Autonomous Agents

Assault game-A3C agent

2016030010 - Kosmas Pinitas

Introduction

The purpose of this work is to create an agent who can take the correct actions in order to maximize the game's score. Nowdays it is common practice to solve such problems with reinforcement learning techniques and more specifically with Q or Deep Q learning. But in this work we will test a fairly new model developed by DeepMind in 2016. This model is called Asynchronous Advantage Actor-Critic (A3C) model.

Background

Environment

We will use "Assault-v0" from OpenAI gym as our environment. In this environment, the observation is an RGB image of the screen, which is an array of shape $(210, 160, 3)$. Each action is repeatedly performed for a duration of k frames, in our case $k = 4$. The environment supports 7 actions $\{\alpha_0, \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6\}$ where:

- α_0 is do nothing,
- α_1 is do nothing,
- α_2 is shoot,
- α_3 is move left,
- α_4 is move right,
- α_5 is shoot left,
- α_6 is shoot right

We can see that actions α_0 and α_1 are exactly the same, so in reality the permitted actions are 6.

Markov Decision Process

We want our agent to perform the best action for a given state. So we want to find a function that matches states to actions. This function is called policy and we define policy as follows:

$$\pi(s) = \alpha$$

Now we can define a Markov Decision Process (MDP) as a set $(S, A, P_\alpha, R_\alpha)$ where:

- S is a finite set of states,
- A is a finite set of actions,

- P_α is the probability that action α in state s at time t will lead to state s' at time $t+1$,
- R_α is the immediate reward (or expected immediate reward) received after transitioning from state s to state s', due to action α

Q-Learning

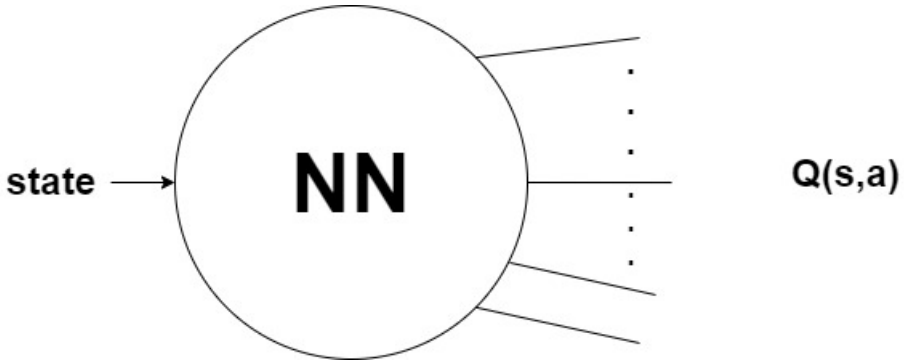
Q-learning is a model-free reinforcement learning algorithm. The goal of Q-learning is to learn a policy, which tells an agent what action to take under what circumstances. It does not require a model (hence the connotation "model-free") of the environment, and it can handle problems with stochastic transitions and rewards, without requiring adaptations.

For any finite Markov decision process (FMDP), Q-learning finds a policy that is optimal in the sense that it maximizes the expected value of the total reward over any and all successive steps, starting from the current state. Q-learning can identify an optimal action-selection policy for any given FMDP, given infinite exploration time and a partly-random policy."Q" names the function that returns the reward used to provide the reinforcement and can be said to stand for the "quality" of an action taken in a given state.
The core of the Q-Learning algorithm is a simple value iteration update, using the weighted average of the old value and the new information:

$$Q^{new}(s_t, \alpha_t) = Q(s_t, \alpha_t) + a \cdot (r_t + \gamma \cdot max_\alpha\{Q(s_{t+1}, \alpha)\} - Q(s_t, \alpha_t))$$

- r_t is the reward received when moving from state s_t to state s_{t+1} ,
- a is the learning rate or step size and determines to what extent newly acquired information overrides old information,
- γ is the discount factor and determines the importance of future rewards.

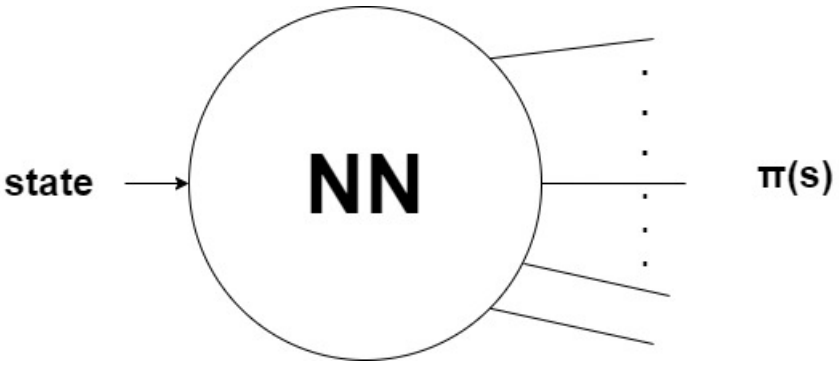
The complexity of computing large Q tables grows exponentially. So in case of a problem with big dimensionality of the states and the actions a nonlinear function approximator such as a neural network is used to represent Q. This technique is called **Deep Q-Learning**.



1: Deep Q Learning neural network

Policy-Gradient

In DQN we approximate Q function but we have already see tha policy is also a function of state. So in this case we will try to approximate directly the policy function. Our neural network with weights θ will now take a state s as an input and output an action probability distribution, for policy π that we will symbolize as π_θ.



2: Policy-Gradient neural network

From now on we will symbolize the distribution of starting states in the environment as ρ^{s_0} and the distribution of states under a policy as ρ^π. Additionaly we will define a function $J(\pi)$ as a discounted reward that a policy π can gain, averaged over all possible starting states s_0.

$$J(\pi) = E_{\rho^{s_0}}[V(s+0)]$$

We can easily find the gradient of the above function by using the Policy-Gradient Theorem:

$$\nabla_\theta \, J(\pi) = E_{s\sim\rho^\pi,\, a\sim\pi(s)}[A(s,a) \cdot \nabla_\theta \, log \, \pi(a\|s)]$$

- $\nabla_\theta \; log \; \pi(a\|s)$ tells us a direction in which logged probability of taking action α in state s rises
- $A(s, a)$ is a scalar value and tells us what's the advantage of taking this action.

If we combine the above terms, we will see that the likelihood of actions that are better than average is increased, and the likelihood of actions worse than average is decreased. Also in order to calculate the expected value we can let our agent run in the environment and record the (s, α, r, s') samples. When we gather enough of them, we can calculate $\nabla_\theta \; J(\pi)$. Finally we can use any of the existing techniques based on gradient descend to improve our policy.

A3C

The Asynchronous Advantage Actor Critic (A3C) algorithm is one of the newest algorithms to be developed under the field of Deep Reinforcement Learning Algorithms.

Asynchronous

In Deep Q-Learning we use a single agent and a single environment, but in A3C we run multiple agents in parallel and each one of them has its own network parameters and a copy of the environment. This agents learn only from their respective environments. Each agent is controlled by a global network. As each agent gains more knowledge, it contributes to the total knowledge of the global network. Different agents will likely experience different states and transitions, thus avoiding the correlation. Another benefit is that this approach needs much less memory, because we don't need to store the samples. Also A3C performs better than the other Reinforcement learning techniques because of the diversification of knowledge as explained above.

Advantage

If we assume that $V(s) = E_{\pi(s)}[r + \gamma V(s')]$ then we can calculate the advantage as follows:

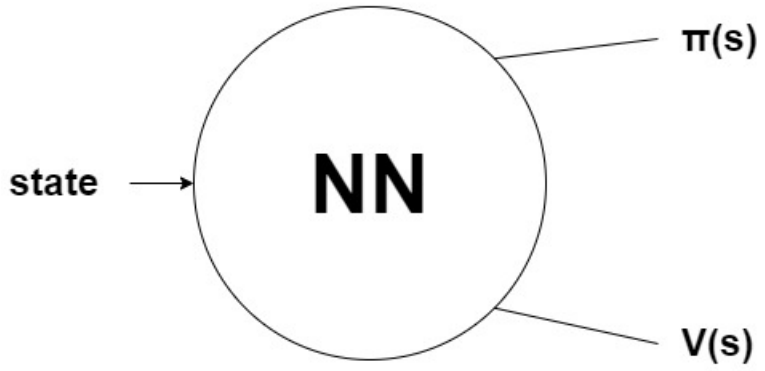
$$A(s, a) = Q(s, a) - V(s) \leftrightarrow$$

$$A(s, a) = Q(s, a) - V(s) = r + \gamma V(s') - V(s)$$

In a Policy-Gradient or a Value-Iteration method we use parametters such as r, γ in order to reward a good action and penalize a bad action. By using the advantage metric the agent also learns how better the rewards are than it's expectation. In other words the advantage function expresses how good it is to take an action α in a state s compared to average.

Actor critic

A3C algorithm combines the best parts of Policy-Gradient and Value-Iteration methods. In this case we use a neural network in order to predict both the value function $V(s)$ as well as the optimal policy function $\pi(s)$.

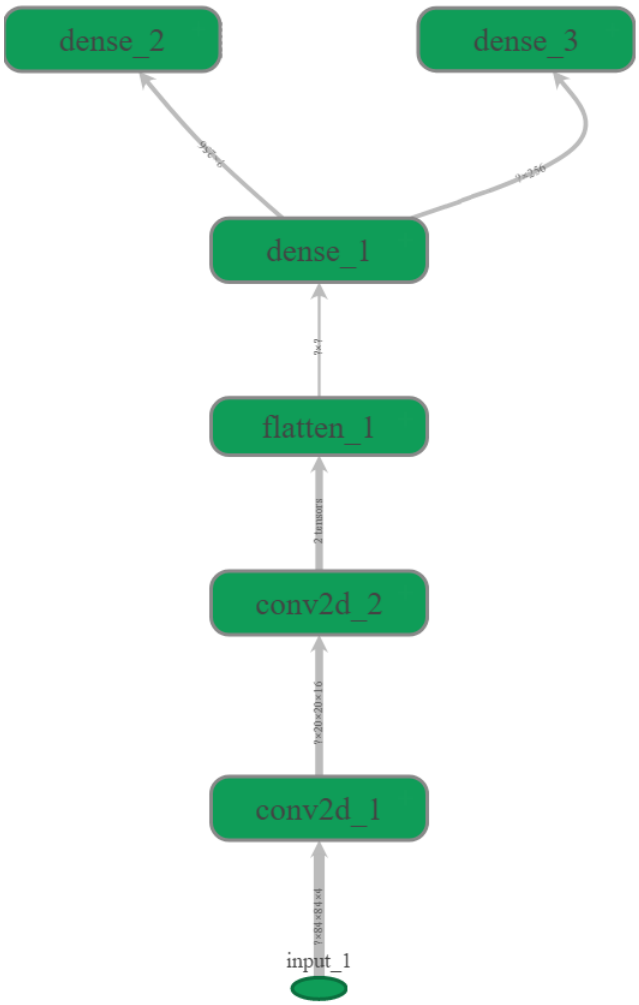


3: Actor-Critic neural network

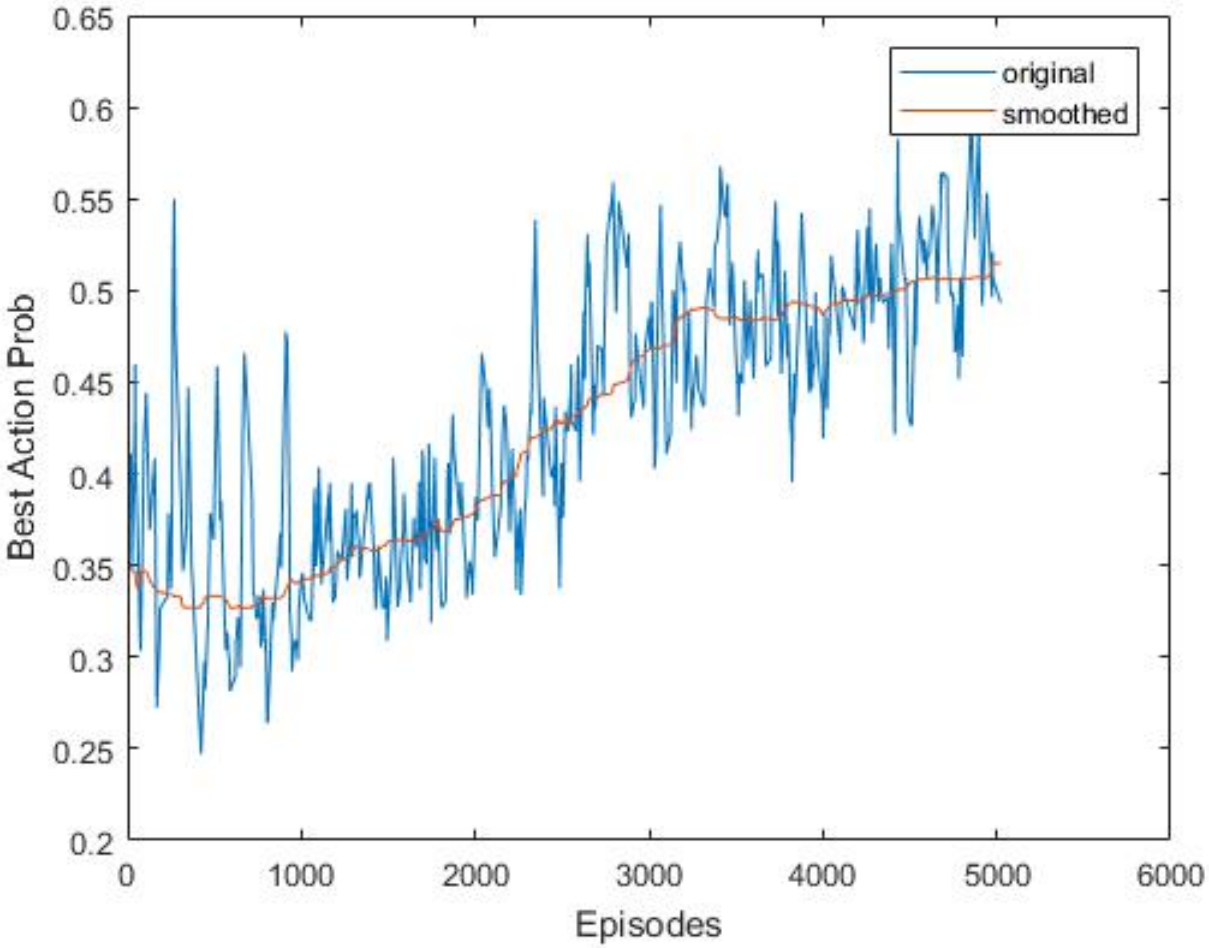
The learning agent uses the value of the Value function (Critic) to update the optimal policy function (Actor). Note that here the policy function means the probabilistic distribution of the action space. Note that this algorithm is faster and more robust than the standard Reinforcement Learning Algorithms and it can be used on discrete as well as continuous action spaces.

Model

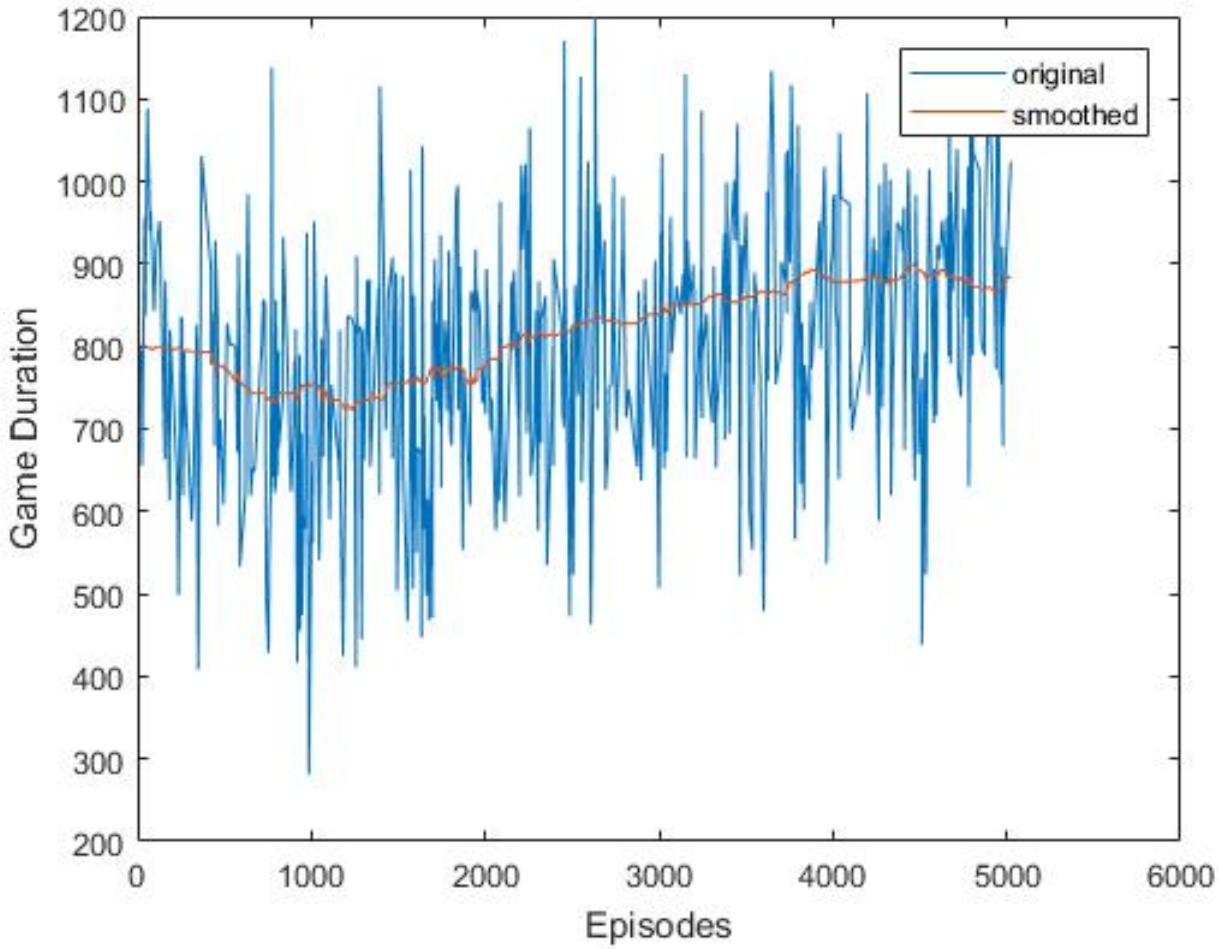
Architecture



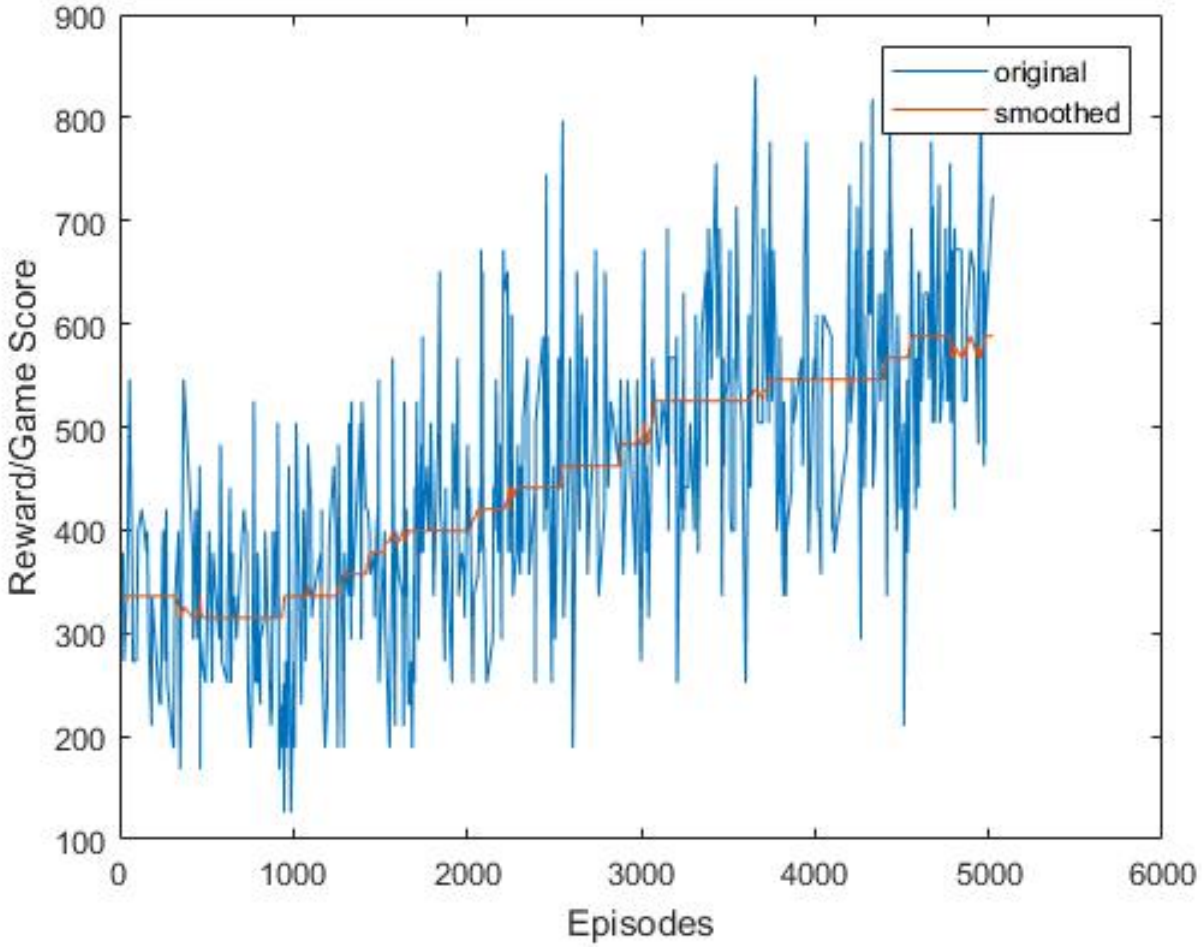
Results



The increase in action probability shows us that our agent is becoming more and more confident over time.



The increase in game duration shows us that our agent learns to protect itself over time.



The increase in reward shows us that our agent learns a winning strategy.

References

- environment: https://gym.openai.com/envs/Assault-ram-v0/
- MDP: https://en.wikipedia.org/wiki/Markov_decision_process
- Q-Learning: https://en.wikipedia.org/wiki/Q-learning
- Policy-Gradients: https://jaromiru.com/2017/02/16/lets-make-an-a3c-theory/
- A3C
  - https://jaromiru.com/2017/02/16/lets-make-an-a3c-theory/
  - https://www.geeksforgeeks.org/asynchronous-advantage-actor-critic-a3c-algorithm/

Dependencies

- python 3.6.5
- keras 2.1.5
- tensorflow 1.6.0
- gym 1.5.0
- numpy 1.14.0