Policy gradient methods in Reinforcement Learning

Olexiy Nagirny, Natalia Novosad

March 2, 2020

Abstract

This project is concentrated on the research of Policy gradient methods (PGM) properties as an approach to solve Reinforcement learning (RL) problems. The main goal of the project is to show how the convergence of PGMs highly depends on the baseline choice. Our secondary goal was to make an overview of PGM methods and emphasize the value of learning rate in PGMs convergence. PGMs represent only part of methods used in RL, though a very important one. In this paper, we will uncover the general RL problem setting and the motivation behind the usage of PGMs in RL in comparison to other methods. In the context of our main goal we will compare REINFORCE and A2C algorithms within the PGM family. As an experimental setting , we will use a virtual environment 'LunarLander' in the form of a spaceship landing, which is amongst the most famous ones in OpenAI Gym library.

1 Introduction

Generally, RL problem setting is based on objective optimization in the environments (processes), based on Markov property. Markov property means that future states conditional probability distribution is relied only upon the current state in the process. For instance, the distribution of future possible drone air positions depends only on the current position, but not on where the drone was flying before now. This theoretical framework is especially useful in formalization of sequential decision-making processes where we need to find an optimal sequence of actions in the process to get the biggest total return in a lot of real-world problems (e.g., for successful drone landing etc.). Hence, expected return maximization is an objective of all RL optimization algorithms.

Although RL is an optimization algorithm, it is not possible to use standard approach because objective function depends on future rewards, which are unknown at the beginning. So the only option here is to investigate. An agent visits various environment states, makes actions, gets rewards and reaches other states in a sequential manner. It can be done either in an episodic setting (i.e., the environment has a finite amount of steps and starts in a new episode after some time) or in setting with infinite steps.

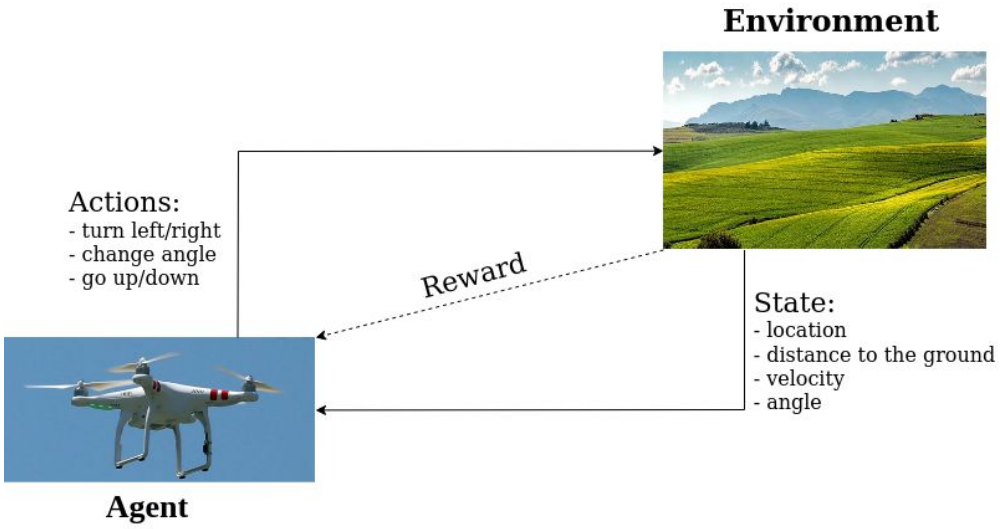
The application of Reinforcement Learning is various: chemistry, finance, robotics, games, traffic control, engineering, etc. Importantly, all RL methods application, including

PGMs, depends not on a particular area of industry/life, but on how certain environment is defined, whether it is finite or infinite, discrete or continuous etc. PGMs have an advantage in the environment with stochasticity. Since PGMs work with the probabilities, it implies some randomness in the decision-making. As a result, the agent can explore more options in the action space and get better rewards with faster process convergence. Moreover, PGMs work with continuous action space that gives much more possibilities in the real world. Here are just some examples, where PGM were used:

1. Robotics: in 2004 the researchers worked on the Sony Aibo robot walk [2];
2. Computer games (AlphaGo, Atari, DOOM etc.);
3. Risk and finance: [3][4][5] finding the best action risks/portfolio modeling.

2 Reinforcement Learning Framework

The pipeline of agent learning is shown in Pic.1. The environment is like a black box for the agent, but it can be explored by doing actions and getting back the observations with rewards. Usually, the observations are in the form of states. To get the agent to know if the action had good or bad consequences, the reward is given (it can be negative or positive). In order to maximize agent's expected total return RL algorithms make use of notions of policies and value functions. Value functions are used in order to find what value certain action or state has under any policy. Policy, in simple words, defines what action should be taken in every state and is the end goal of RL-agent which take actions in the environment.



Picture 1: Scheme of the Reinforcement Learning framework

Thus, there are main definitions in RL:

1. State S_t - the properties of the situation in some moment of time t;
2. Action A_t - an action that was done at some moment of time t;

3. Reward R_t - reward at some moment of time t;
4. Episode - the finite or infinite sequence of the steps made by the agent. The process can be terminated by the environment. In this case, we have T - the number of steps in the episode. When the episode is terminated, the new episode starts from the initial state; It is also possible to have a continuous problem setting
5. Policy $\pi(a|s)$ - policy(decision-making function), that can be either deterministic (i.e., chooses exact action in particular state) or stochastic(i.e., represent probability distribution over actions in particular state) which defines probability
6. Return G_t - total return at the moment of the time t, which is a discounted sum of experienced rewards, γ - discounted rate

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T \qquad (1)$$

7. Action-value function $q_\pi(s, a)$ - the value (expected return) of taking action a being at the state s over some policy π;

$$q_\pi(s, a) = \mathbb{E}[G_t | S_t = s, A_t = a] \qquad (2)$$

8. State-value function $v_\pi(s)$ - the value (expected return) of the state s over some policy π;

$$v_\pi(s) = \mathbb{E}[G_t | S_t = s] \qquad (3)$$

We can differentiate between the two main approaches in Reinforcement learning: action-value and policy-based methods. The latter one will be at the focus of our project, but their essence will become more clear after comparison with action-value methods.

Action-value methods concentrate on iteratively finding the value $q_\pi(s, a)$ of action a, given state s under initially random current policy π, which is called *policy evaluation step.* After that, methods improve (update) current policy by taking greedy-action selection $\pi(s) = argmax_a(q(s, a))$, choosing the action with the highest value at a particular state. This way, action-value methods are similar to successive approximation algorithms, where one array holds old values while the second one is updated [6].

In contrast, PGMs directly concentrate on finding optimal policy π without the need to estimate action-value functions. It is done by policy parametrization and using stochastic gradient ascent (or other optimizer, such as Adam) in order to maximize policy performance function $J(\theta)$[6]:

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t), \qquad (4)$$

where α - is learning rate, θ - parameters of $J(\theta_t)$. Our parameterized policy is usually approximated by neural network in practice.

3 Policy Gradient methods classification and overview

3.1 General considerations

Our main classification of policy gradient made in this paper is based upon whether it has a baseline in its update rule (this represents majority of nowadays PGM algorithms). The baseline can work mainly as a scale of gradient, being an essential part of update rule. The essence of the baseline approach will be explained later.

Policy gradient methods can also be classified by whether they belong to *on-policy* or *off-policy* type. In the case of policy gradient methods, on-policy algorithms compute the gradient of performance function, $\nabla J(\theta)$, using only target policy. Basically, it means that samples from agent interaction with the environment are collected according to the policy we are looking for. Off-policy methods also use behavior policy, currently followed by an agent [7]. We will use only on-policy algorithms in our further analysis.

As was said before, the learning process can be episodic (there is condition under which the process can finite and the agent has to start again) and continuing (the number of steps is infinite). Depending on the chosen approach, the performance function is different. Till the end of the paper, we consider only episodic learning, when the parameters θ_t updates at the end of the episode.

So, the performance function is:

$$J(\theta) = \mathbb{E}[G_t|S_t = s_0] = \sum_a \pi_\theta(a|s) q_{\pi_\theta}(s,a) \qquad (5)$$

This performance function is quite expected. We want to maximize the performance function - the total return of the process from the first step. Such parametrization makes possible smoother convergence and learning in environments with very long episodes. The derivative of performance function is described by the Policy Gradient Theorem.

Theorem 1 (Policy Gradient Theorem)[6]:

The derivative of the performance function (5) is:

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_{\pi_\theta}(s,a) \nabla \pi_\theta(a|s) \qquad (6)$$

where $\mu(s)$ - is the distribution of the states under the policy and symbol $\propto$ - mean 'proportional to'.

Proof: see in [6].

The theorem gives us the value which is proportional to performance function, not the function itself. But it is enough for us because we just need the correct direction for the ascent (4) and the scale is set by the learning rate.

Now we need to approximate expressions (6). We will do this by using the learned action-value function $q(S_t, a)$ during the episode:

$$\nabla J(\theta) = \mathbb{E}_\pi[\sum_a q_\pi(S_t, a)\nabla_\theta \pi_\theta(a|S_t)] \qquad (7)$$

We will consider the implementation of two on-policy policy gradient algorithms (REINFORCE and A2C), showcasing the difference between approaches with and without baseline in policy gradient framework.

Given policy gradient theorem, REINFORCE is guaranteed to converge to a local minimum. In its core it relies on sampling from many episodes in order to find the optimal policy, i.e. is a Monte-Carlo method. However, we need to wait until the end of each episode. AlI that leads to REINFORCE having high variance and, thus, slower convergence properties [6] [7].

Actor-critic algorithms, introducing value approximation, improve the speed of convergence to the optimal policy by allowing updates at each step without waiting until the whole episode ends.

3.2 REINFORCE

Derivative of performance function (7) builds a family of *all-actions* methods because it involves all the actions that were made during the episode [6][1]. The first algorithm we consider is classical REINFORCE. This algorithm involves calculation of all actions A_t in the episode for updating the parameters θ at the time point t. The performance function for REINFORCE can be directly derived from Policy Gradient Theorem and is the following:

$$\nabla J(\theta) = \mathbb{E}_\pi[\frac{q_\pi(S_t, A_t)}{\pi_\theta(A_t|S_t)}\nabla_\theta \pi_\theta(A_t|S_t)] = \mathbb{E}_\pi[q_\pi(S_t, A_t)\frac{\nabla_\theta \pi_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)}] =$$

$$= \mathbb{E}_\pi[G_t \nabla_\theta \ln \pi_\theta(A_t|S_t)] \qquad (8)$$

where

- π_θ - policy, targeted by an agent, parameterized by θ;
- $q_\pi(S_t, A_t)$ - action-value function under policy π;
- $\pi_\theta(A_t|S_t)$ - probability distribution of random variable A_t (actions), conditioned on S_t (states) under policy π_θ.

To start implementing the REINFORCE we need to randomly choose the policy function. It has to return the value of probability of making action a on the state s, so the domain of the function must be [0,1].

REINFORCE algorithm starts with generating an episode of MDP (i.e. game) following policy π, using randomly initialized weight vector θ of the function approximator (can be neural network). Then it iteratively finds total discounted return for each step (action) of the episode and multiplies gradient of log probability vector $\nabla \ln \pi(A_t|S_t, \theta)$ by that discounted return. Afterwards we perform gradient ascent changing our weight vector θ, which defines our policy at a given step. That multiplication by total discounted return is a key idea of REINFORCE, as we more want episodes having total return of 100 to contribute to the gradient than those with a return of 5. Note that gradient here is with respect to weight vector θ. The pseudocode is given below:

Algorithm 1: REINFORCE

Input: a differentiable policy parametrization $\pi(a|s, \theta)$
Algorithm parameter: step size $\alpha > 0$
Initialize: policy parameter $\theta \in R^n$
loop(for each episode):
Generate and episode $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \theta)$;
loop(for each step of the episode t=0,1,...T-1):

$$G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$$

$$\theta \leftarrow \theta + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \theta)$$

---

[6]

'Pure' policy gradient method REINFORCE is theoretically guaranteed to local optima convergence, but in practice there are several serious limitations which make it restricted in its usage:

1) One iteration of training requires waiting until the whole episode ends;

2) High gradient variance. As an example, in the 'LunarLander' environment in one episode we can get +100 points and in other -100. The difference between successful and unsuccessful attempts becomes too large and only few 'good' episodes will dominate gradient;
3) Very high correlation between training samples in one episode can lead to problems in gradient descent. [6] [7]

3.3 Baseline approach. Advantage Actor-Critic method.

In order to correct the variance of our gradients we usually use baseline function $b(S_t)$, which helps to scale our gradient vector appropriately. Generally, with baseline approach we update weights θ vector in the following manner:

$\theta_{t+1} \leftarrow \theta + \alpha(G_t^*)\nabla \ln \pi(A_t|S_t,\theta)$,

where

$G_t^* = G_t - b(S_t)$.

Basically $b(S_t)$ can any function. The only requirement is that baseline should not directly stem from parameterized policy π. The most popular and efficient gradient scales now are considered to be state/action-dependent. This group of policy gradient methods further improves gradient variance problems and also deals with waiting till the end of the whole episode per iteration problem.

It represents the so-called Actor-Critic approach and it uses a mix of policy gradient and action-value methods. One of the most members of such a group is Advantage Actor-Critic Algorithm (A2C). Namely, A2C method use state value v_π as a baseline, exploiting a fact that our $q(s,a)$ function can be represented as $v_\pi + Adv(s,a)$

,where

$Adv(s,a)$ - advantage of the action.

This way v_π can be a baseline and $Adv(s,a)$ will be the scale of the gradient dependent on the baseline state. But since we don't know v_π we can use function approximators, such as neural networks, and update its parameters on each step in order to direct ourselves for the true value of state. Such a state-value function approximation is called critic, evaluating our actions under policy, while our function approximator of policy is called 'actor'.

It then means that we have two function approximators: one is used as an "actor", which acts environment (samples actions from policy parametrized distribution) and the other one works like "critic", updating current action-value function.

In our work we would use Advantage Actor-Critic Algorithm in representing baseline approach. Pseudocode for it looks is as follows:

Algorithm 2: Advantage Actor-Critic (A2C)

Input: a differentiable policy parametrization $\pi(a|s,\theta)$

Input: a differentiable state-value parameterization $v(s,w)$

Parameters: step sizes $\alpha > 0$, $\beta > 0$

Initialize: policy parameter $\theta \in R^n$, state-value weights $w \in R^n$

loop(for each episode):

Play t steps in the environment using the current policy, saving triples (s,a,r);

If s is terminal:

$R \leftarrow 0$

For $i = t-1, ..., 1$:

$R \leftarrow r_i + \gamma R$

Accumulate PG:

$$\partial\theta_\pi \leftarrow \partial\theta_\pi + \nabla_\theta \log \pi_\theta(a_i \mid s_i)(R - v_(s_i)) =$$

$$= \partial\theta_\pi + \nabla_\theta \log \pi_\theta(a_i \mid s_i)(Adv(s,a))$$

Accumulate state-value gradients:

$$\partial w \leftarrow \partial w + \frac{\partial (R - v_s)^2}{\partial w}$$

Update parameters using accumulated gradients in the direction of PG $\partial\theta_\pi$ and in the opposite direction of the value gradients ∂w

4 Experimental setup

4.1 Goals

Investigate how baseline approach and learning rate to PGMs influence policy gradient convergence.

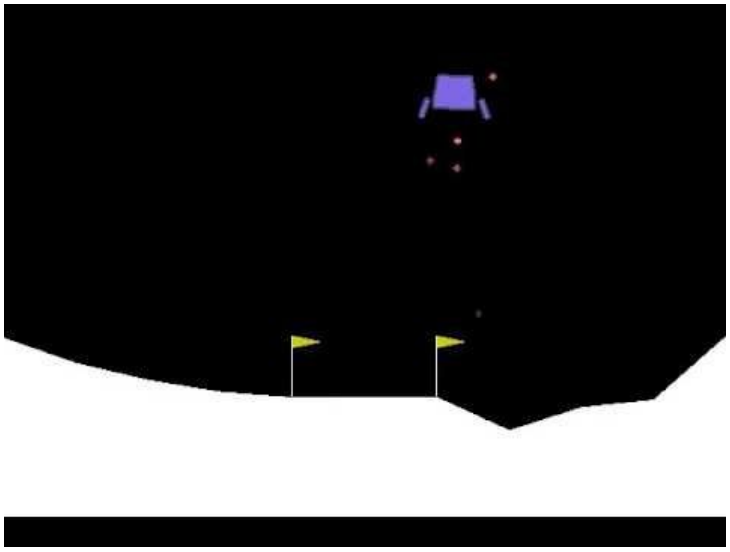
4.2 Environment

In training our RL-agents we used one of the environments provided by OpenAI Gym. Gym is a library created to develop and compare various RL-algorithms [9]. We chose the 'LunarLander-v2' environment, which has the main goal of accurate and fast drone landing onto

the pad [10]. The environment represents a real-world practical problem of constructing successful control systems for drone landing.

In the view of the MDP framework our environment consists of the next elements:

- State-space with 8 possible states: horizontal and vertical position, horizontal and vertical velocity, angle and angular velocity, left and right leg contact;
- Action space with 4 possible actions: push up (use the main engine), push left (left engine), push right (right engine) and do nothing;
- Reward system: every episode finishes if the lander crashes or comes to rest, receiving -100 or +100 points. Each leg ground contact is +10. Firing main engine is -0.3 points each frame. The firing side engine is -0.03 points each frame. Solved is 200 points.



Picture 2: 'LunarLander-v2' environment

Our comparison metric is simply the convergence rate of REINFORCE and A2C algorithms.

4.3 Training settings for REINFORCE and A2C

1) Function approximator:

Function approximator is in the form of one-hidden layer neural network with 128 neurons without softmax unit as this will be calculated separately. The same option is used for two-headed neural network, where we have actors and critics. Probably, this parameter can be tweaked, but neural networks architecture comparison is not a goal of this project.

2) Gradient optimizer:

To update the parameters in (4) we need to choose the optimization algorithm. Adam optimizer is the most popular for RL problems because of its adaptability. Adam uses first-order

momentum (to smooth the direction) and second-order momentum. Adam changes the learning rate in dependance on the parameters of the model. Adam solves the problem of too slow convergence or too big jumps.. We use Adam optimizer for all of the below models. Optimizer comparison is out of scope of this work, concentrating on other aspects, such as gradient scaling. If you want to read more, see [11].

3) Activation function:

In all our neural networks we need to implement activation functions. The most common activation functions are ReLu, Softmax, ELU and others. We have chosen ReLu for our experiments as an activation function to neural network function approximator, as it is considered one the most efficient:

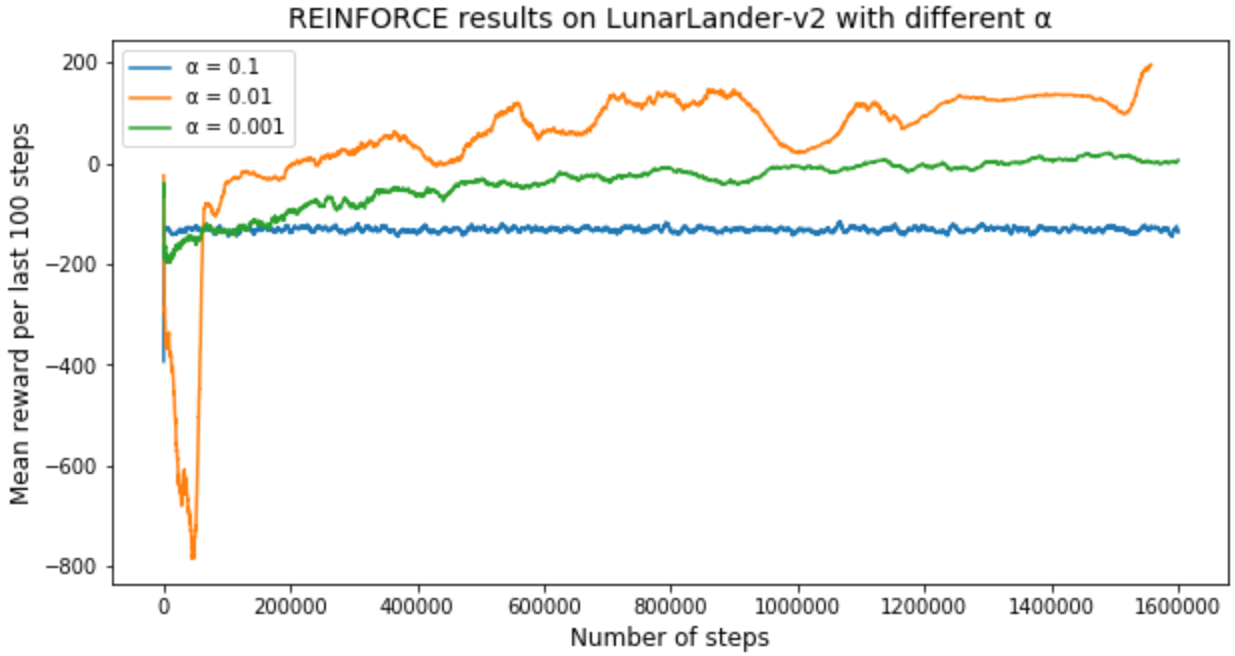
$$\pi_\theta(a|s) = \begin{cases} \theta^\top s & if \quad \theta^\top s > 0 \\ 0 & if \quad \theta^\top s \leq 0 \end{cases}$$

4) Gamma value $\gamma = 0.99$ - this is the above-described discount rate, which we use uniformly in all algorithms.

4.3. Experiments results

So, let's see results of our comparison firstly between learning rates convergence in REINFORCE algorithm.

On the Pic 2 we can see REINFORCE algorithm with different learning rates: 0.1, 0.01, 0.001.

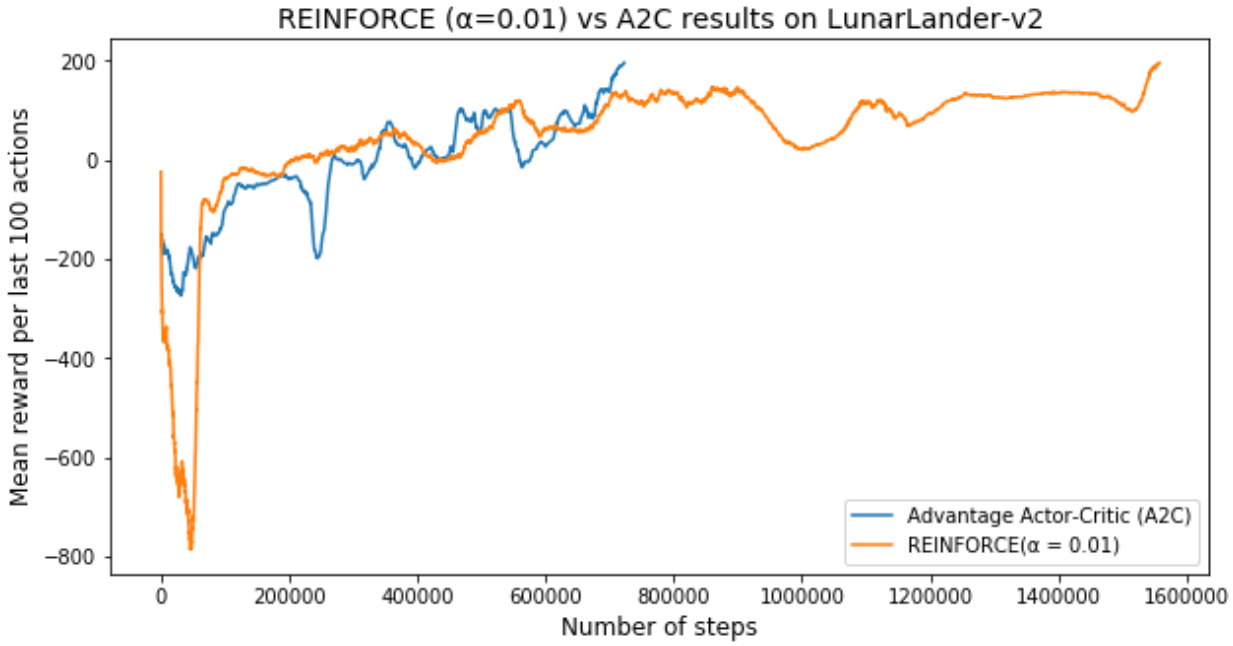


Picture 2: 'LunarLander-v2' environment

We see that the learning rate has a significant impact on REINFORCE convergence properties. Only learning rate with $\alpha = 0.01$ solved the environment. We also see that, independently of the learning rate, REINFORCE training is highly unstable and noisy. This fact is connected with having no baseline included in this algorithm.

Thus, we then implemented Advantage Actor Critic Algorithm (A2C) in order to check whether baseline approach with 2 function approximators will do better.

For our experiment we also included with A2C option that couldn't be done in REINFORCE, such as training with 50 environments. This means basically, that agent collects experience before training iteration on 50 independent environments, so that above-mentioned problem of high correlation between samples of the same environment can be resolved. Thus, our policy gradient iteration is done using results from i.i.d environments which is generally good for neural networks stability.



We can deduce from the above image the following:

- Variance of convergence has stabilized with A2C, though it remains noisy and demands further hyper-parameter tuning;
- Baseline approach in the form of A2C led to faster convergence.

5 Conclusions

We achieved all the main goals that we put before us, such as

- demonstration of the influence of appropriately selected baseline on policy gradient convergence and
- showing learning rate tuning significance for RL agent learning;
- uncovering overview and essence of Policy Gradient methods in RL.

Difficulties in writing this report can be classified into two groups: theoretical difficulties and experiment difficulties.

Theoretical difficulties are connected to lack of formal convergence properties exploration for discussed algorithms.

One of the main difficulties encountered during experimentation is connected with a lack of computational power, which causes real problems in iterative process and evaluation of optimal model hyperparameter, such a learning rate, not saying about models with a lot higher amount of hyperparameters. One CPU barely handles convergence to optima in such a simple environment as 'LunarLander'.

We think that further work in the area of Policy Gradient methods is needed to advance boundaries of RL.

Literature:

1. Deep Reinforcement Learning Hands-On, Maxim Lapan, 2018/Packt Publishing Ltd./UK

2. http://www-anw.cs.umass.edu/~barto/courses/cs687/Kohl-Stone-04.pdf

3.https://arxiv.org/pdf/1911.03618.pdf

4.http://papers.neurips.cc/paper/5923-policy-gradient-for-coherent-risk-measures.pdf

5.https://arxiv.org/pdf/1906.09090.pdf

6. Reinforcement Learning: An Introduction, R. Sutton, A. Barto, The MIT Press, second edition

7. https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html

8. https://arxiv.org/pdf/1707.06347.pdf

9. https://gym.openai.com/docs/

10. https://gym.openai.com/envs/LunarLander-v2/

11. Adam: a method for stochastic optimization, Diederik P. Kingma, Jimmy Lei Ba