Computational Assignment #6
Policy Gradient: REINFORCE

Aleksei Petrenko
University of Southern California
petrenko@usc.edu

Abstract

In this project we implement the standard policy gradient algorithm called REINFORCE and evaluate its performance on classical control tasks such as OpenAI Gym MountainCar-v0 and CartPole-v0. We further analyze the advantages and shortcomings of this algorithm and show the implementation details that are necessary to achieve high performance in practice.

1 Introduction

Constructing an algorithm that can learn to control an agent in a variety of environments and achieve high utility has been a long-standing goal in artificial intelligence. The field of reinforcement learning has produced some of the most promising approaches to this problem, and the recent advances in Deep RL [1] show that applications of these methods can span beyond the simple low-dimensional toy problems. Modern reinforcement learning techniques have a tendency to share a set of building blocks which are combined and modified to achieve high robustness. REINFORCE policy gradient algorithm (initially proposed by Williams et al. in 1992 [2]) is one of those fundamental building blocks, and is a precursor to many of contemporary state-of-the-art methods [3, 4]. Therefore for anyone who seeks to conduct novel research in the field of reinforcement learning it is crucial to study and understand the REINFORCE policy gradient algorithm.

2 Monte Carlo Policy Gradient Algorithm

We will consider a standard problem of finding good control policy in a Markov Decision Process (MDP). The basic scheme of interaction between the agent and the environment is shown on the Fig. 1 [5].

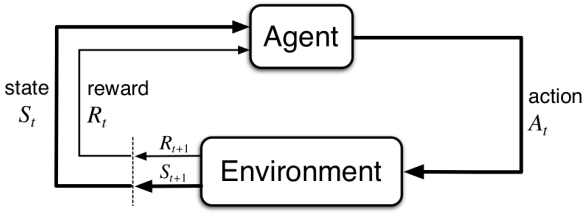

Figure 1: The agent–environment interaction in a Markov decision process [5].

The MDP and agent together give rise to a sequence or *trajectory* or states, actions and rewards that can be written down as following:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ... \tag{1}$$

The actions A_i are sampled from the stochastic policy $\pi(A_i|S_i, \theta)$ parameterized by the parameter vector θ. The performance or utility of the policy can be formalized as an expectation of the discounted sum of rewards from the current state onward:

$$J_\pi = \mathbb{E}_\pi \left[\sum_{i=0}^{\infty} \gamma^i R_i \right] \tag{2}$$

In the discounted setting the REINFORCE algorithm provides the following update rule for the parameters of the policy θ, derived from the policy gradient theorem [5]:

$$\theta \leftarrow \theta + \alpha \gamma^t G_t \nabla \ln \pi \left(A_t | S_t, \theta \right) \tag{3}$$

where G_t is a discounted cumulative return from time t all the way to the end of the episode (Monte-Carlo return).

2.1 REINFORCE with Baseline

In the original formulation given by the equation (3) the REINFORCE algorithm can be very sensitive to the structure of rewards in the environment. For example, consider the goal-based environment where the agent is given a reward of $R = 1000$ at every timestep and $R_{goal} = 1001$ for reaching a goal. In this case, the update formula will *increase* the action probability for every action taken in the environment. Furthermore, the magnitude of the update will be very similar for all actions, regardless of whether the agent ended up reaching a goal or not. To counteract this, the REINFORCE algorithm can be modified to include the baseline function $b(S_t)$. The corresponding update rule is given by the equation below:

$$\theta \leftarrow \theta + \alpha \gamma^t \left(G_t - b(S_t) \right) \nabla \ln \pi \left(A_t | S_t, \theta \right) \tag{4}$$

Note that this is a strict generalization of (3) because baseline function can be uniformly zero $b(S_t) = 0$.

Baseline function can be anything, as long as it does not depend on the action. One popular choice is to subtract the mean rewards observed over a large batch of trajectories. Baseline function can also be learned to predict the return G_t for the current state more accurately.

With a learned baseline function the REINFORCE method starts to resemble an actor-critic algorithm, the only difference between them is the type of return G_t used in the update rule. REINFORCE algorithm uses Monte-Carlo return which is an (optionally) discounted sum of rewards from the current state to the end of trajectory, while actor-critic methods use a bootstrapped return based on the value estimate. Actor-critic is therefore a TD-learning method, while REINFORCE is not. As a consequence of this property, REINFORCE policy gradients are unbiased but have high variance, while bootstrapped gradients are biased and have lower variance. Actor-critic methods tend to be more sample-efficient and also, unlike REINFORCE, have ability to operate on very long trajectories, effectively learning online.

3 Experiments

In this project the REINFORCE algorithm was evaluated on two classic problems: MountainCar [6] and CartPole (OpenAI Gym implementations were used [7]). To an unaided eye these problems may seem benign, while in reality they pose a major challenge for policy gradient algorithms. In both MountainCar and CartPole the rewards received by the agent are the same for all states and actions, and the only source of learning signal is the length of the episode. Because of that, a very careful choice of the baseline is required.

3.1 MountainCar Environment

In this environment the goal of the agent is to move the car to the top of the hill on the right. Three actions are available to the agent: accelerating to the left, accelerating to the right and coasting. The reward of -1 is received for every action in the environment, including the terminal state. The episode is terminated after 200 steps or when the goal is reached. The screenshot of the environment is shown on the figure below:

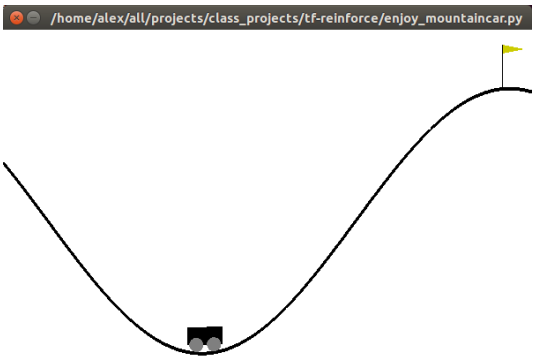

Figure 2: A screenshot of the OpenAI Gym MountainCar-v0 learning environment.

The structure of the reward in this environment makes MountainCar a hard exploration problem. In the beginning of training while actions are mostly random it is almost impossible to achieve the top of the mountain by mere chance, therefore the agent receives no positive reinforcement whatsoever. In this work we used additional tricks to simplify the exploration in the beginning:

1. Each action sampled from the policy is repeated 4 times. This trick alone makes it possible to solve MountainCar without additional reward shaping, albeit the progress is very slow in the beginning.
2. The environment reward is modified in a following way: $R_{modified} = R_{original} + |v|$ where v is a car velocity. The contribution of this v-reward is very small (measured on the order of 0.01), so it does not change the overall objective, but allows the algorithm to find the good policy faster.

Additional implementation details that helped to improve performance on MountainCar:

- The following baseline function was used: $b(S_t) = R_{avg}/L_{episode}$ where R_{avg} is the total episode reward averaged over 100 most recent episodes and $L_{episode}$ is the length of the current episode.
- The experience was collected into batches of size 512 for more stable training.
- The clipping of the gradients helps to prevent the policy from collapsing to a bad local optimum in the beginning of training, especially when large learning rate is used.

The figures below illustrate the learning process for four different runs: stochastic linear model and MLP (multi-layer perceptron) with different learning rates. Interesting observations:

- Linear models are more robust to high learning rate, but at equal learning rate they train slower than MLP models.
- MLP model trained with high learning rate can achieve a very high reward quickly, but then start to degrade (large learning rate can lead to bad updates). Note though that the policy still reaches a goal in 100% of the episodes.
- MLP policies tend to converge to lower entropy, almost zero (which means almost deterministic action selection at all states).

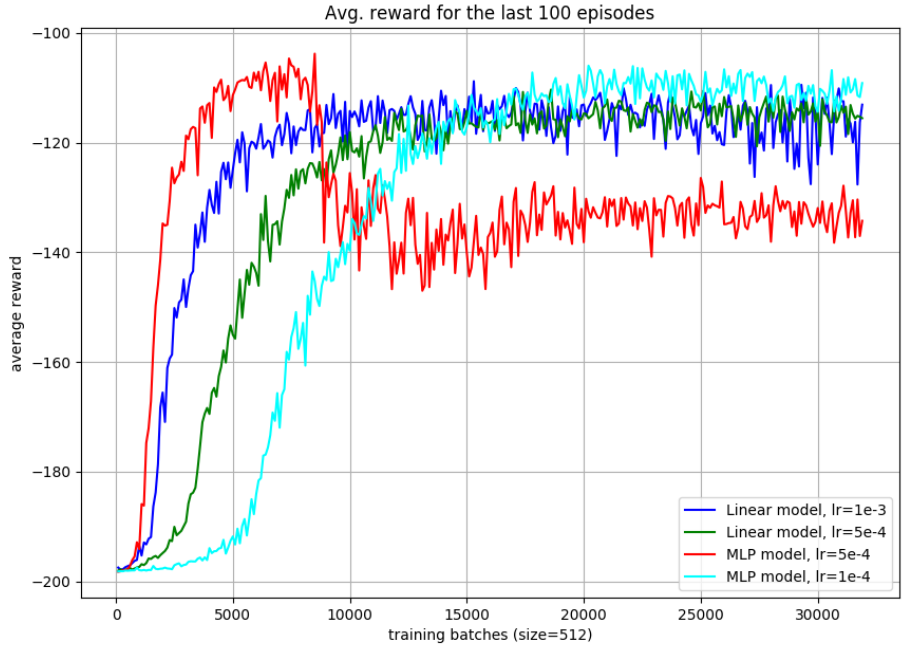

Figure 3: Average reward during training.

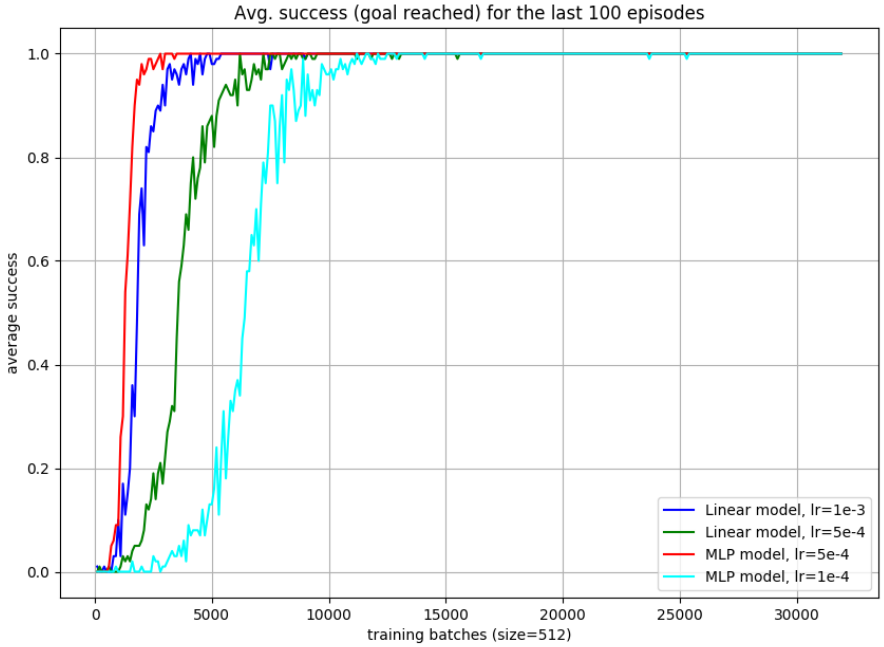

Figure 4: Fraction of the episodes where a goal was reached during training.

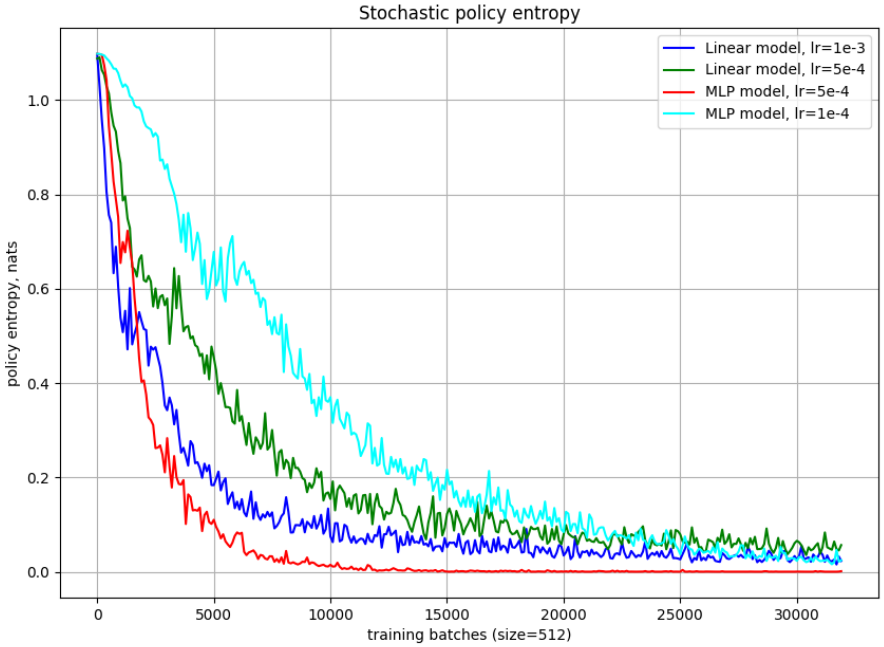

Figure 5: Policy entropy evolution during training.

TensorFlow implementation of the algorithm and the visualization of the fully trained agent can be found on this GitHub page: `https://github.com/alex-petrenko/tf-reinforce`

3.2 CartPole Environment

In the CartPole environment the goal is to keep the pole upright by moving the car left or right. The reward of $+1$ is given for every state, including the terminal state. The episode is terminated after 200 timesteps, or when the pole is sufficiently tilted.

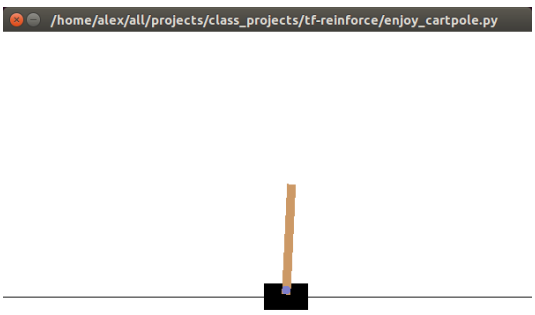

Figure 6: A screenshot of the OpenAI Gym CartPole-v0 learning environment.

CartPole is significantly easier than MountainCar, mainly because it does not require a lot of exploration. From the very beginning of training, the strong learning signal is available: longer episodes give higher reward. Therefore, both linear and MLP models learn very quickly, only in a few hundred episodes. No additional algorithm modifications were required to solve the CartPole environment.

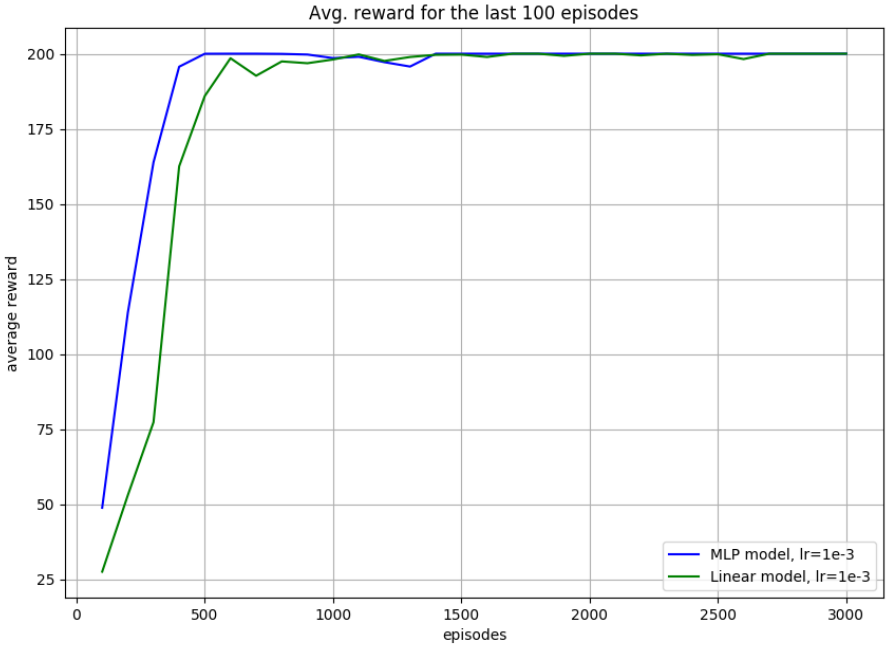

Figure 7: CartPole: average reward during training.

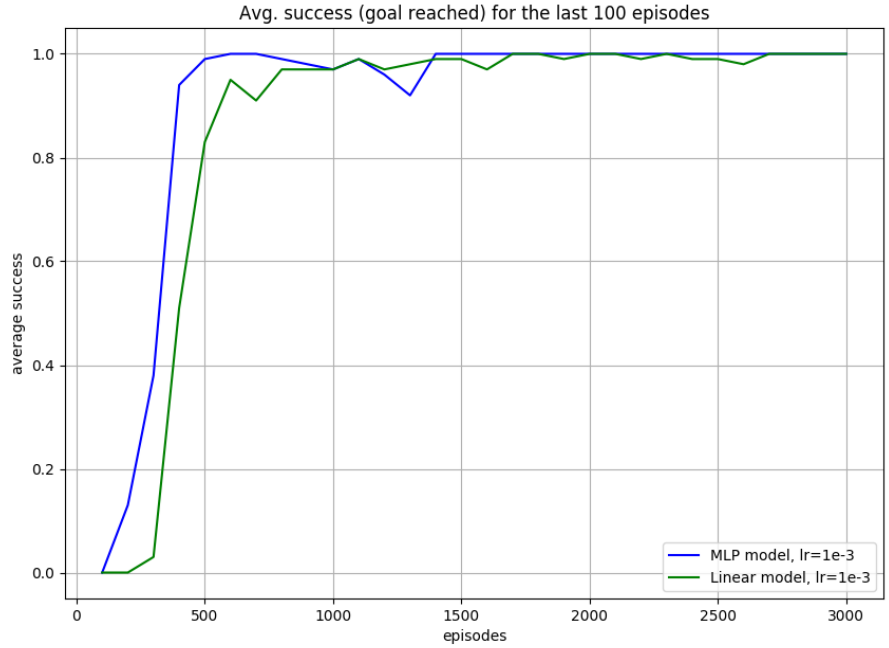

Figure 8: CartPole: fraction of the episodes where the pole was kept upright for 200 steps.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[2] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3-4):229–256, May 1992.

[3] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[4] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[5] R.S. Sutton, A.G. Barto, and F. Bach. *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning. MIT Press, 2018.

[6] Andrew William Moore. Efficient memory-based learning for robot control. Technical report, 1990.

[7] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.