# TML Assignment 3*
# Policy Gradient Methods

Soumyasis Gun
20171002

## Introduction

Reinforcement learning problems may have either a discrete or continuous action space that greatly affects the algorithm used. Deep reinforcement learning algorithms have already been applied to both discrete and continuous action spaces. In this work, we compare the performance of two well established model-free DRL algorithms: Deep Q Network for discrete action spaces, and the continuous action space-variant Deep Deterministic Policy Gradient on two classic RL problems, CartPole and Pendulum Problem using OpenAI gym.

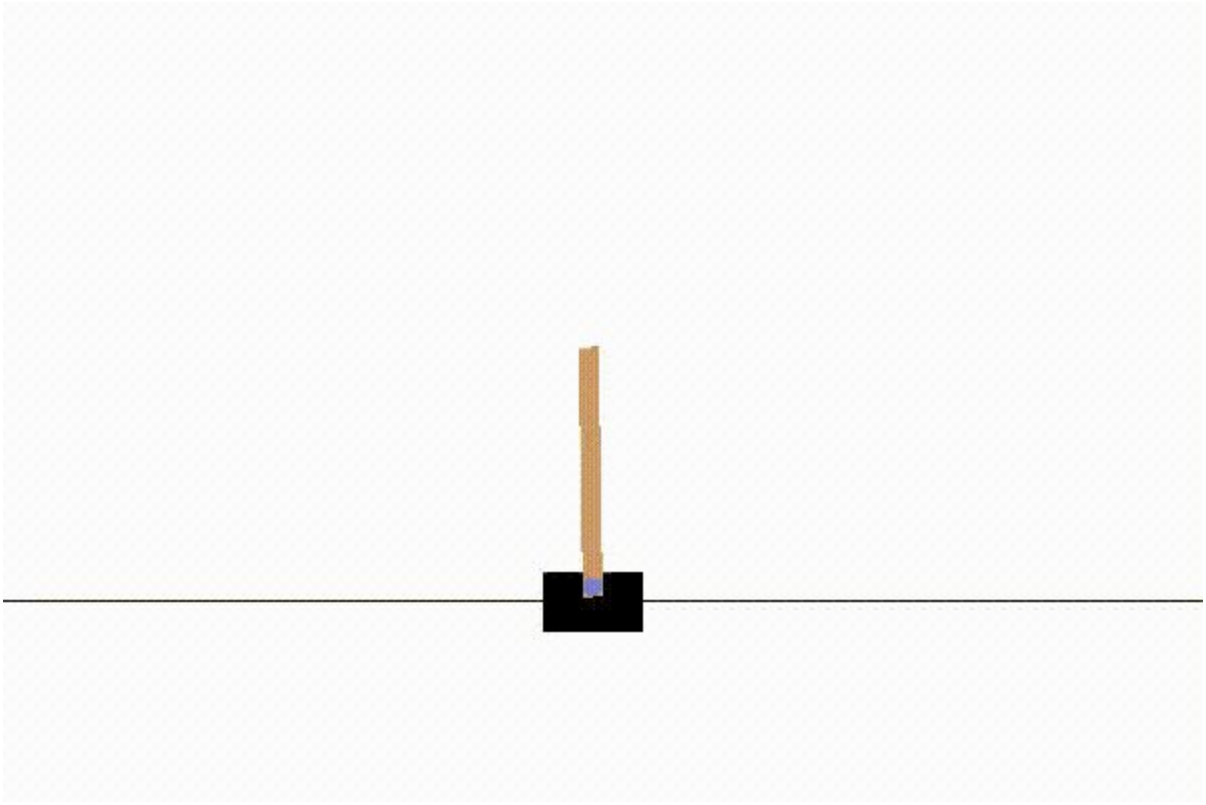
## Cartpole Problem

(*Note that results and analysis for this are added in the jupyter notebook titled "Discrete Action Spaces.ipynb"*)

Cartpole - known also as an Inverted Pendulum is a pendulum with a centre of gravity above its pivot point. It's unstable, but can be controlled by moving the pivot point under the centre of mass. The goal is to keep the cartpole balanced by applying appropriate forces to a pivot point.

A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the centre.

**Files hosted at https://github.com/DefUs3r/Policy-Gradient-Methods*



In short,
Action space (Discrete)
0 - Apply 1 unit of force in the left direction on the Cart
1 - Apply 1 unit force in the right direction on the cart

State space (Continuous)
0 - Cart Position
1 - Cart Velocity
2 - Pole Angle
3 - Pole Velocity At Tip

We use several methods to solve this, notably a Vanilla Policy Gradient (with and without baseline) and an Actor-Critic Method which are described below.

**Vanilla Policy Gradient(VPG):**

The key idea underlying policy gradients is to push up the probabilities of actions that lead to higher return, and push down the probabilities of actions that lead to lower return, until you arrive at the optimal policy. VPG is an on-policy algorithm and can be used for environments with either discrete or continuous action spaces. VPG trains a stochastic policy in an on-policy way. This means that it explores by sampling actions according to the latest version of its stochastic policy. The amount of randomness in action selection depends on both initial conditions and the training procedure. Over the course of training, the policy typically becomes progressively less random, as the update rule encourages it to exploit rewards that it has already found. This may cause the policy to get trapped in local optima.

**Algorithm 1** Vanilla Policy Gradient Algorithm

1: Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function $V_{\phi_k}$.
6: Estimate policy gradient as

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$$

7: Compute policy update, either using standard gradient ascent,

$$\theta_{k+1} = \theta_k + \alpha_k \hat{g}_k,$$

or via another gradient ascent algorithm like Adam.
8: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left( V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.
9: **end for**

After one batch, we may exhibit a wide range of results: much better performance, equal performance, or worse performance. The high variance

of these gradient estimates is precisely why there has been so much effort devoted to variance reduction techniques.
The standard way to reduce the variance of the above gradient estimates is to insert a baseline function $b(s_t)$ inside the expectation like this.

$$\nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_t|s_t)\left(\sum_{t'=t}^{T-1} r_{t'} - b(s_t)\right)\right]$$

Here we define something called as Advantage function $A(s,a)$:

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

It can be considered as another version of Q-value with lower variance by taking the state-value off as the baseline.
Results in the jupyter notebook will support the claim that the policy gradient method has lesser variance after adding a baseline of the mean.

**Actor-Critic Policy Gradient Algorithms**

Two main components in policy gradient are the policy model and the value function. It makes a lot of sense to learn the value function in addition to the policy, since knowing the value function can assist the policy update, such as by reducing gradient variance in vanilla policy gradients, and that is exactly what the Actor-Critic method does.

Actor-critic methods consist of two models, which may optionally share parameters:

- *Critic* updates the value function parameters w and depending on the algorithm it could be action-value $Q_w(a|s)$ or state-value $V_w(s)$.
- *Actor* updates the policy parameters $\theta$ for $\pi_\theta(a|s)$, in the direction suggested by the critic.

The algorithm is:

1. Initialize s,θ,w at random; sample a~πθ(a|s).
2. For $t = 1\ldots T$:
    a. Sample reward $r_t \sim R(s,a)$ and next state $s' \sim P(s'|s,a)$;

b. Then sample the next action $a' \sim \pi_\theta(a'|s')$;
c. Update the policy parameters: $\theta \leftarrow \theta + \alpha_\theta Q_w(s,a) \nabla_\theta \, ln \, \pi_\theta(a|s)$;
d. Compute the correction (TD error) for action-value at time $t$: $\delta_t = r_t + \gamma Q_w(s',a') - Q_w(s,a)$ and use it to update the parameters of action-value function: $w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s,a)$
e. Update $a \leftarrow a'$ and $s \leftarrow s'$.

Two learning rates, $\alpha_\theta$ and $\alpha_w$, are predefined for policy and value function parameter updates respectively.
However the vanilla version of actor-critic method are on-policy: training samples are collected according to the target policy — the very same policy that we try to optimize for. Off policy methods, however, result in several additional advantages:

- The off-policy approach does not require full trajectories and can reuse any past episodes ("experience replay") for much better sample efficiency.
- The sample collection follows a behaviour policy different from the target policy, bringing better exploration.

Results corresponding to these are placed in the Jupyter Notebook along with justifications. Gaussian Policy are implementationally same to VPG so the concept has just been explained here.

Let us now look at the next problem, Pendulum-v0

**Pendulum Problem**
(*Note that results and analysis for this are added in the jupyter notebook titled "Continuous Action Spaces.ipynb"*)

The goal of the pendulum-v0 problem is to swing a frictionless pendulum upright so it stays vertical, pointed upwards. The pendulum starts in a random position every time, and, since pendulum-v0 is an unsolved environment, it does not have a specified reward threshold at which it is considered solved nor at which the episode will terminate.

There are two major observation inputs for this environment, representing the angle of the pendulum (0) and its angular velocity (1).
The action is a value between -2.0 and 2.0, representing the amount of torque on the pendulum.

In short,
Action space (Continuous)
0- The torque applied on the pendulum, Range: (-2, 2)

State space (Continuous)
0- Pendulum angle
1- Pendulum speed

We repeat our trials using VPG (with and without baseline) as explained before and use DDPG (Lillicrap, et al., 2015), Deep Deterministic Policy

Gradient which is a model-free off-policy actor-critic algorithm, combining DPG (Deterministic Policy Gradient) with DQN (Deep Q-Network).
DPG models the policy as a deterministic decision: a=μ(s). We can consider the deterministic policy as a special case of the stochastic one, when the probability distribution contains only one extreme non-zero value over one action. In the DPG paper (ICML 2014), the authors have shown that if the stochastic policy $\pi_{\mu_\theta,\sigma}$ is re-parameterized by a deterministic policy $\mu_\theta$ and a variation variable σ, the stochastic policy is eventually equivalent to the deterministic case when σ=0. Compared to the deterministic policy, we expect the stochastic policy to require more samples as it integrates the data over the whole state and action space.

However, unless there is sufficient noise in the environment, it is very hard to guarantee enough exploration due to the determinacy of the policy. We can either add noise into the policy (making it non-deterministic) or learn it using off-policy by following a different stochastic behaviour policy to collect samples.

DQN stabilizes the learning of Q-function by experience replay and the frozen target network. The original DQN works in discrete space, and DDPG extends it to continuous space with the actor-critic framework while learning a deterministic policy.

In order to do better exploration, an exploration policy μ′ is constructed by adding noise $N$:

$$\mu'(s) = \mu_\theta(s) + N$$

In addition, DDPG does conservative policy iteration on the parameters of both actor and critic, with $\tau \ll 1 : \theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$. In this way, the target network values are constrained to change slowly, different from the design in DQN that the target network stays frozen for some period of time.

**Algorithm 1** DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ 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 $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ 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 $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

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

    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**

Results for DDPG are included in the Jupyter Notebook.

References:


1. David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, et al.. Deterministic Policy Gradient Algorithms. ICML, Jun 2014, Beijing, China. ffhal-00938992f
2. Lillicrap, T., J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver and Daan Wierstra. “Continuous control with deep reinforcement learning.” CoRR abs/1509.02971 (2016): n. Pag.
3. Schulman, John, P. Moritz, S. Levine, Michael I. Jordan and P. Abbeel. “High-Dimensional Continuous Control Using Generalized Advantage Estimation.” CoRR abs/1506.02438 (2016): n. pag.
4. https://medium.com/@aniket.tcdav/vanilla-policy-gradient-with-tensorflow-2-9855df271472 (For VPG Theory)
5. OpenAI Gym for CartPole and Pendulum gifs.