Ruggero Rossi

r.rossi@opencomplexity.com

A Gentle Principled Introduction to Deep Reinforcement Learning

version 1.0.16b

Preface

This book is an attempt to introduce Deep Reinforcement Learning to those who possess a solid knowledge of Deep Learning. The aim is to explain the functioning of selected Deep Reinforcement Learning algorithms that are both practically effective and didactically representative, providing the reader with a theoretical justification and a balanced level of proofs and mathematical details, without losing sight of the general picture. The prerequisite is the knowledge of Deep Learning, that implies knowledge of basic Calculus, Linear Algebra, Probability and Statistics.

Contents

1. Introduction

1.1 Reinforcement Learning Definition

Reinforcement Learning is a method for an artificial agent to do automated learning using the outcomes caused by its actions. A definition from [Sutton & Barto 2018] reports: "Reinforcement learning is learning what to do - how to map situations to actions - so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them".

So, paraphrasing Sutton and Barto, Reinforcement Learning is: learning which action to take, depending on the current situation, with the purpose of maximizing the total reward.

In that definition we have some of the base components of a Reinforcement Learning system:

- an *agent* able to decide which action to take
- a *state* of the world – or better an *observation* about the state of the world
- a *reward*

The *agent* is obviously the artificial intelligence that is trying to learn how to act, it may be just a program inside a simulation as well as a physical robot in a real environment.

The *observation* about the state of the world is the data that the agent possesses about the state of world, and that may be potentially incomplete, noisy, or delayed. The observation may be given to the agent by the simulator software, in case of simulated environment, or may be obtained by the agent through sensors in case of physical robot.

The *reward* is a number that represents how well or how badly the agent is behaving (usually it is positive for good rewards and negative for bad rewards). It is something similar to the concept of utility. In case of a simulated environment the reward is given to the agent by the simulator itself, together with the timed information updates such as the observation of the state of the world. In case of physical robot in a real environment instead, a software layer must be programmed in the robot in order to understand what is happening in the world and calculate a reward that is related to how much good or bad the situation is.

In both cases the reward is a number whose value is communicated to the agent by a human-made program: either in the simulation or in the robot software. There is not such a thing in the environment as a “reward number”, the reward is just a fictitious number that permits to do numerical optimization (humans and animals feel pain and pleasure and that may be seen as a reward system that has been evolved, but for our artificial agents it has to be built by us). Anyway in some cases that number may be somehow directly available from the real world or from the final purpose objective: think about an artificial intelligence trying to maximize the gains in the stock market, the reward may be directly the market returns, or think about of a robot trying to pick up trash from the ground, a reward proportional to garbage weight may be given every time that garbage is actually collected.

If the task to be accomplished by the agent is never ending it is said to be “infinite-horizon”, otherwise if a begin and an end of the sequence of actions can be identified, it is said “finite-horizon” and we may call “episode” what happens within the begin and the end. For instance, if a robot is trying to shoot a basketball inside a hoop, an episode starts when the robot tries to grab the ball and ends when the ball, after being shot, lands, either passing through the hoop or outside it. An episode contains a “*trajectory*” made of a sequence of states and actions. The sum of the rewards obtained in an “*episode*”, possibly discounted by a time distance value, is called “*return*” (the time discount represents the preference of receiving rewards immediately rather than in the future).

Another useful concept when talking about reinforcement learning is the concept of “*Policy*”: with that term we mean a strategy that associates an action to a world state, for each possible state. That means that in each situation an agent is, the Policy will tell him what action to take. A good Policy is one that makes the agent collect a good amount of rewards, an optimal policy is one that makes the agent collect the maximum amount of rewards (there may be more than one optimal policy). A policy may be stochastic and define, for each state, a probabilistic mixture of actions to take, such as “take action A with probability 80%, take action B with probability 20%”.

So, what finally an agent wants to learn is an optimal policy (or at least one good enough ! If finding the optimal policy takes too much time or resources, it may be better to find just a decent policy and use that, which is the topic “exploration vs exploitation” discussed later).

The agent may or may not have knowledge of how the world works and what happens if he takes a certain action, in which state he will end up and what reward will be obtained because of that action. When the agent has that kind of knowledge it is possible to apply the so-called *model-based* reinforcement learning algorithms. The model of the world is generally considered stochastic, with deterministic models being just particular cases. The model may be a fixed, prior knowledge of the agent. If instead the agent has not a prior model of the world mechanics, it is possible either to learn one in the process, or to use *model-free* algorithms that don't require a model of the world. In some case the model of the world is available (at least partially), in some other cases the learning algorithm may be trying to learn a model of the world through experience and then use that model to apply a model-based algorithm. When the model of the world is complete, and when there are a finite number of states it is theoretically possible to calculate the best action for each state just with numerical optimization (e.g. with dynamic programming), using the "Bellman Equation" [Bellman 1952] [Sutton and Barto 2018] without the need to make the agent taking real actions. Usually that is not the case because often the model of the world is not known (and sometimes difficult to be learned) and there is an infinite number of states, hence agents must advance through trial and errors to discover what is the best action in which situation.

Trying actions in the environment with the purpose of learning exposes the agents to the risk of obtaining very bad rewards: in a real environment that would mean damaging seriously the robot, or even worse creating risky situations outside the experimental environment. So, there is a trade-off between exploration (trying new actions to figure out if they bring better rewards) and exploitation (doing only the actions that so far showed to be sufficiently rewarding). Exploration exposes to risks but permits to optimize the policy, exploitation permits to gain the fruits of past exploration but avoids further learning. Agents usually are given a greater degree of exploration at the beginning, and the exploration degree is decreased as the learning progresses, favouring exploitation.

Some Reinforcement Learning algorithms (but not all) use the concept of "Value Function": a function that takes the current state (or current observation) as input and evaluates the hypothetical goodness of it.
The evaluation score returned by the Value Function is related to how good it is expected to be the future situation in the long run, not just in the immediate next moments. If a certain state is expected to make the agent obtaining a reward, the value function will take in account the reward expected in that situation and all the rewards expected from what hypothetically may be the future situations if the agent acts as its "policy" suggests, possibly discounting the future rewards by a time-dependent distance value (rewards further in time may weight less than immediate rewards).

So, as I will describe later formally with the Bellman Equation, the Value function is a recursive concept: saying that the value at some state depends not only on the reward obtained in that state but also on all the rewards that may be obtained afterwards, is the same of saying that the value at a certain state depends not only on the goodness of that state "per se" but also on the values of the states that are reachable from there. This also implies that the value function depends on the policy: two different policies may reach different states from the same starting point and hence they may have two different value functions. So, in general each value function is tied to a certain policy: changing the policy changes the value function.

There are two different types of value functions: proper “value functions” and “action-value functions”.
The proper “value function”, returns the value of a state, considering that the agent will follow the policy from that state, so its only input is the state (and implicitly the policy). It is sometimes called also “state-value” function.
The “action-value” function takes as input the state and also an action, and returns a value considering that the agent will take that particular action in that state (even if it’s a different action with respect to what the policy would suggest), and after that it will follow his policy.

It has to be noted that when in literature it is used the term "Value Function” it may be referred to two very different things: the first is the value function as described above, the second is the estimate of the value function that an agent is approximating, and that may be far from the real value (and it should be better referred to with the terms “value function estimate”). When there is a finite number of states the estimate of the value function is often saved in a table with an estimate for each state (or, in case of “action-value”, an estimate for each state-action pair), when instead there is a great number of states that cannot be done and a function approximator is used, such as a neural network. That may be the case when the state/observation is described by continuous variables which if discretized in a table would need too much memory, or which would lose relevant details.

The term “Deep Reinforcement Learning” is used to describe Reinforcement Learning methods that use neural networks, especially neural networks with many hidden layers (“deep”). For instance, neural networks may be used to approximate value function estimates, or, as we will see later, for policy functions. The usage of deep neural networks allows to learn complex policies that are not possible with linear methods as function approximators. Moreover, they allow to learn end-to-end, that means having a neural network which, without the hand-made engineering of different other software modules, receives the sensory input and internally learns all the necessary functionalities to process it and take the output action, as opposed to having a separate module that analyses the input, a next module that plans what to do, a further module that learns how to actuate the robot engines, etc.

1.2 Reinforcement Learning Formalization

A formalization of the Reinforcement Learning problem may be done expressing it as a Markov Decision Process (MDP). A MDP is a way to describe how an agent passes from one state to another and possibly obtains a reward as a consequence of its actions. One requirement is that states must have the “Markov Property”: the probabilities to move from one state to another and to obtain rewards depend only on the knowledge of the current state and the chosen action, and not on any previously accessed state. In other words, a state description has the Markov property if it contains all the necessary information from the past and from the present to determine the transition probabilities to other states and the rewards.
This means that theoretically, non-Markov states can be turned in Markov states if all the history of past states and interactions are added to them (but this way to describe states as long sequences of the past are not simple to be managed by algorithms).
In a MDP, the mechanism that makes an agents passing from a state to another as a consequence of its actions is described by a “*transition function*” that is generally probabilistic and it is not necessarily known by the agent (the agent may just experience the change in state after an action). The mechanism that assigns a reward depending on a certain state, an action executed in that state, and a consequent state, is described by a “*reward function*”, that is not necessarily known by the agent (the agent may just notice the obtained reward after every action).
The theoretical exposition of RL that follows, when a different source is not referenced, is taken from [Sutton & Barto 2018] and [OpenAi 2018A], with some change of variables names to have a consistent description.

Formally, following OpenAI definition, a MDP is a 5-tuple $\langle S, A, R, P, \rho_0 \rangle$:

- S is the set of all states
- A is the set of actions
- $R : S \times A \times S \rightarrow \mathbb{R}$ is the reward function, with $r_t = R(s_t, a_t, s_{t+1})$
- $P : S \times A \rightarrow \mathcal{P}(S)$ is the transition function, with $P(s'|s, a)$ being the probability of transitioning from state s to state s' after taking action a, and $\mathcal{P}(S)$ is the set of probability measures on S
- ρ_0 is the distribution of initial state

The policy that decides which action to take depending on the state, is a stochastic function π , such that if at time t the state is s_t , it will return a probability distribution over the actions, from which it may be sampled the action to take a_t:

$$a_t \sim \pi(\cdot\ |s_t) \tag{1.1}$$

A sequence of states and actions is called "trajectory", identified by the symbol τ

$$\tau = (\ s_0, a_0, s_1, a_1, \dots) \tag{1.2}$$

The return $G(\tau)$ of a trajectory is the (potentially infinite) sum of all rewards of the trajectory, with $\gamma \in [0,1]$, and it is time-discounted if $\gamma < 1$:

$$G(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t \tag{1.3}$$

Given a policy π , the probability of following any trajectory τ made of T steps is:

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t) \tag{1.4}$$

And the expected return $J(\pi)$ is :

$$J(\pi) = \int_\tau P(\tau|\pi)G(\tau) = E_{\tau\sim\pi}[G(\tau)] \tag{1.5}$$

Where a notation like $E_{x\sim z}[f(x)]$ means: Expected Value of $f(x)$ where x follows the probability density function (or probability mass function) $z(x)$.

Later we will see also a notation like $E_{x\sim z}[f(x)|y = m,\ u = n]$ that means: Expected Value of $f(x)$ where x follows the pdf/pmf $z(x)$, given that y is equal to m and u is equal to n .

The problem of Reinforcement Learning is to find the policy that maximizes that expectation in eq. 1.5. Hence:

$$\pi^* = arg \max_{\pi} J(\pi) \tag{1.6}$$

Where π^* is the optimal policy.
The Value Function for a given policy π is:

$$V^{\pi}(s) = E_{\tau\sim\pi}[G(\tau)|s_0 = s] \tag{1.7}$$

The Action-Value function (also named “Q-Value”) is:

$$Q^{\pi}(s,a) = E_{\tau\sim\pi}[G(\tau)|s_0 = s, a_0 = a] \tag{1.8}$$

It has to be noted that

$$V^{\pi}(s) = E_{a\sim\pi}[Q^{\pi}(s,a)] \tag{1.9}$$

As previously anticipated, the value function is a recursive concept because the value of a state is dependent on the value of the next state, recursively. The Bellman Equations express this relationship:

$$V^{\pi}(s) = E_{\substack{a\sim\pi \\ s'\sim P}}[R(s,a,s') + \ \gamma V^{\pi}(s')] \tag{1.10}$$

$$Q^{\pi}(s,a) = E_{s'\sim P}\left[R(s,a,s') + \gamma E_{a'\sim\pi}[Q^{\pi}(s',a')]\right] \tag{1.11}$$

Using eq. 1.9 in 1.11 we obtain:

$$Q^{\pi}(s,a) = E_{s' \sim P}[R(s,a,s') + \gamma V^{\pi}(s')]$$
(1.12)

Some algorithms use the "Advantage Function" [Baird 1993], that is a function indicating how much it is better or worse to take a certain action (and after that follow the policy) instead of taking the action suggested by the policy, i.e. it indicates the relative advantage of taking a certain action with respect to the policy. The advantage function is the following:

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$
(1.13)

1.3 Variety of Methods

Among RL methods, a first distinction can be made between **model based** and **model free** methods. Model based methods use a model of the world, consisting generally in the knowledge of the reward and transition functions, and the initial state distribution: R, P, ρ_0 .
This model is either already existing from a priori knowledge or learnt by the agent.
Model free methods instead do not use a model (generally in real environments a model is not available), and are agnostic of initial state distribution, transition function, and reward function (they are aware only of the rewards that the agent receive when it receives any, without knowing the underlying function or rule).

A second distinction is between on-policy and off-policy methods: **on-policy** methods evaluate and improve the same policy that is followed by the agent during training, while in **off-policy** methods the training is done following a policy that is different from the one that is evaluated or improved. In off-policy methods the policy that is being learned is named "*target policy*" and the policy that is used to generate samples is named "*behavior policy*". A particular version of off-policy reinforcement learning is the **offline** reinforcement learning, where the data needed to feed the algorithm has been previously collected and no new data is produced [Levine et al. 2020].

A third distinction may be made between value based methods and policy based methods. In **value based** methods the algorithm aims at estimating a value function or an action-value function and consequently modify the policy to prefer actions that lead to better values. The "*Policy Improvement Theorem*" [Sutton & Barto 2018, Ch.4] guarantees that modifying the policy for a state in a way that obtains a value improvement in that state, strictly improves the expected return $J(\pi)$ and hence improves the overall policy.

If a world model is not available, “proper” value functions (i.e. state-only values) do not contain enough information to improve the policy, hence the only model-free value based methods are those that use action-values: so they can choose the action that is associated with the best outcome for each state.

In **policy based** methods instead, the action is directly chosen by a policy function (that outputs a probability distribution over actions) and value or action-value functions may not be present, or may be used only to improve learning.

A fourth distinction of RL methods can be done between “tabular” and “approximate” methods. **Tabular** methods are those whose state space is composed by a finite set of discrete states, that are hence representable in memory by tables, for instance the Value Function estimation may be simply a table that associates a state to a value. **Approximate** methods are those whose state space is continuous or too big, and so its representation in the algorithm is made by some function approximation. For instance, the Value Function could be approximated (or estimated) by a neural network that takes the information about the state/observation (e.g. sensor data, webcam frame image, etc.) as input, and computes an approximate value as output. It is common to use convolutional neural networks as approximators when the state/observation input is an image. While tabular and approximate methods share much theory and algorithms, they also have important differences, the most notable of which is the fact that an approximate value based algorithm, if it is also using bootstrapping (i.e. using the approximate value function estimation as a replacement for the true value of value function in the Bellman equation, or equivalently using the approximated q-value estimation instead of true q-value) and if it is off-policy, it is not guaranteed to be convergent, while a tabular method would be, see [Sutton and Barto 2018, Ch. 11].
When a deep neural network is used as a function approximator in a RL algorithm, it is said to be Deep Reinforcement Learning.

2. Q-Learning

2.1 Essential Algorithm

An example of (action-)value based method is the Q-Learning [Watkins 1989], an algorithm for discrete actions. Let us see how it works.

The basic idea of Q-Learning is to estimate the value of any action-state pair, and then the target policy would be the policy that, given a state, selects the action with greater action-state value estimate. But the important detail in all this is that during learning, Q-Learning "bootstraps", that means that it estimates the value of a past action summing the obtained reward for the past action and the estimate of the value function of the next state. This will be explained better later.

Q-Learning is an off-policy method, that means that the behavior policy (the policy used to generate trajectories) does not have to coincide with the target policy (the policy that we want to improve until it is optimal). In fact they could be very different, but for practical reasons it is better that they are not too much: since a function approximator is used to evaluate state-action value estimates, it is better to train that approximator on state-action samples coming from a state-action distribution that is similar to the one of the target policy. This is because the approximation capacity of a function approximator is limited and it is more precise when it is run on inputs similar to the ones used for training.

For this reason in Q-Learning the policy followed by agents during training (the *behavior policy*) is often made not too different from the target policy: the behavior policy sometimes chooses the action with the greatest action-value (exploitation) and sometimes a random action (exploration). A way to do that is to have an ϵ-greedy policy: with probability ϵ a random action is taken, with probability $1-\epsilon$ instead the action with greater action-value (Q-value) is taken. Usually, ϵ is progressively decreased during learning.

In Q-Learning the Q-value is computed with respect to a policy (the *target policy*) that would always choose the best action, and never a random one. This is expressed with the formula:

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

(2.1)

I wrote Q^* instead of Q^π to make explicit that it is not computed over a generic policy π but ideally over the optimal policy (the asterisk means "optimal policy"). You can compare the differences between equations 2.1 and 1.11, where it was:

$$Q^\pi(s,a) = E_{s'\sim P}\big[R(s,a,s') + \gamma E_{a'\sim\pi}[Q^\pi(s',a')]\big]$$

(remarkably, in Q-Learning the target policy always selects the best action, so it has $\gamma\, max_{a'}\, Q^*(s',a')$ instead of $E_{a'\sim\pi}[Q^\pi(s',a')]$).
Hence the Q-Value is referred to a tentatively optimal policy, that is asymptotically better or equal than the behavior policy. This makes Q-Learning formally an off-policy method because it evaluates a different policy than the one used to generate trajectories.

In Q-Learning the true Q^* function is unknown, and the algorithm tries to estimate it during the course of learning. So, for the Q-value estimate instead of Q^* we will use the symbol Q_ψ that means that it is an estimate using parameters ψ .

$$Q_\psi(s,a) = E_{s'\sim P}\big[R(s,a,s') + \gamma\, max_{a'}\, Q_\psi(s',a')\big]$$

(2.2)

The execution of Q-Learning is as follows: at the beginning all $Q_\psi(s,a)$ for all states and actions are set to arbitrary values (if Q_ψ is in tabular form the values can be set close to zero for faster convergence, while if Q_ψ is a function approximator such as a neural network the parameters ψ may be set to a small random noise such as in [Glorot and Bengio 2010] or [Saxe et al. 2013]). Then using the behavior policy (for instance the ϵ-greedy policy specified before) an action a is taken, new state s' is reached, and the reward $r = R(s,a,s')$ is obtained.

Now, if we want our estimate of the Q function to respect the Bellman Equation 2.2, we have to make $Q_\psi(s,a)$ move towards the right-hand side value, to be more similar to it. So, calling the "target value" as y :

$$y = r + \gamma\, max_{a'} Q_\psi(s',a')$$

(2.3)

Then if Q_ψ is in tabular format, to move it closer to the target value, it is updated in the following way:

$$Q_\psi(s,a) \leftarrow Q_\psi(s,a) + \eta\ (\ y - Q_\psi(s,a)\ ) \tag{2.4}$$

Where η is the learning rate. This is called "*Temporal-Difference Learning*" (or "TD Learning") because the estimate of the q-values Q_ψ is updated using the difference between the bootstrapped target value after having done the action and the q-value estimate before doing the action.
If instead Q_ψ is not in tabular format but it is estimated using a function approximator such as a neural network, a Loss is computed for the discrepancy of the approximated Q-value :

$$L = (\ y - Q_\psi(s,a)\ )^2 \tag{2.5}$$

Then the parameters of Q_ψ are updated with a gradient descent step (or with batch gradient descent) using the gradient of that Loss function.

It must be noted that in equation 2.5 the value y is to be considered as a constant, not as a value depending on Q_ψ , so that the gradient of the loss with respect to ψ will not include the effect of $max_{a'} Q_\psi(s',a')$ but only of $Q_\psi(s,a)$. For this reason, the Q-Learning update is considered "semi-gradient", and not complete gradient descent. In other words, the update in case of function approximation would be the following:

$$\psi \leftarrow \psi + \eta\ \ (\ y - Q_\psi(s,a)\ )\nabla_\psi Q_\psi(s,a) \tag{2.6}$$

Then, the algorithm continues in the same way both in tabular and approximated version: a new action a' is taken following the policy, and Q_ψ is updated either with the tabular or the gradient descent method using the new $y' = r' + \gamma\ max_{a''} Q_\psi(s'',a'')$ as described above, and so on.

Algorithm 2.1 **Deep Q-Learning (simple version)**

Require: Step size η

Require: Initialize parameters ψ of network Q_ψ with small random values

Do until (supposed) convergence:

using some behavior policy collect a transition $\langle s_i, a_i, s_{i+1}, r_i \rangle$

$y_i = r + \gamma \; max_{a_{i+1}} Q_\psi(s_{i+1}, a_{i+1})$

$\psi \leftarrow \psi + \eta \; (y_i - Q_\psi(s_i, a_i) \;) \nabla_\psi Q_\psi(s_i, a_i)$

end of do

When the agent is not in the learning phase, i.e. when it must use the Q-value to choose an action during its operational task, each q-value can be read directly from the q-function estimate Q_ψ , that is just calling $Q_\psi(s, a)$. The target policy in q-learning is the greedy policy, that mean that in a state s , the q-value for all actions is read, and the action with greater q-value is the chosen one:

$$a_{chosen} = \; argmax_a Q_\psi(s, a) \tag{2.7}$$

2.2 Minibatch Improvement

The algorithm above is a simple version and in the next part of the chapter we will see how to improve it. The first thing to notice is that it collects a transition and computes the respective change of Q-value estimate immediately. But to improve learning speed it is possible to collect a certain number of transitions, called "minibatch", and calculate at once all the q-value changes for all the transitions in the minibatch by vectorized computations in a GPU or in a Tensor Processing Unit (note: in Machine Learning literature some authors use the word "batch" as a synonym for "minibatch", some authors instead reserve the word "batch" to describe the whole set of samples available. Hence for clarity's sake I used the word "minibatch", to indicate a certain number of samples but not the whole set).

2.3 Saving Transitions into a Buffer

As wrote above, Q-Learning is an off-policy algorithm because it trains under a policy that includes exploration, but computes a Q-function estimate for a policy that only assumes exploitation. In fact, because of how the algorithm is designed, theoretically the behavior policy could be even a policy very different from the target policy, we are not obliged to use an ϵ-greedy policy to generate samples. That means that all past trajectories may be retained and reused in future optimization iterations. Saving past trajectories in a buffer allows us to subsequently sample them randomly so to have uncorrelated transitions instead of transitions that are consecutives and hence correlated and close in state space (when transitions are correlated, learning can be unstable). Nonetheless there is a practical limit: if the q-value estimate is computed through a neural network, it has a finite approximation capacity. A neural network can approximate better when the inputs at inference time are of the same distribution of the training samples (this, very trivially, because the number of parameters is finite and the network cannot retain all the information that is presented during training, and it will approximate better for the kind of inputs that went through a greater number of optimization steps). So, it is better to train the q-network with input states that are on the same distribution that the q-network would experience if using the target policy (the tentatively optimal/greedy policy). To have similar distribution of states it is necessary to use a behavior policy that is similar to the target policy. That motivates the usage of an ϵ-greedy policy and motivates the practice of re-using the old samples (saved in a buffer) only for a certain number of steps and then discard them. A simple mechanism to achieve that is, when the buffer is full, to discard eldest samples to make room for new ones, namely a FIFO (first-in-first-out) buffer. Nonetheless, the bigger is your neural network, and hence the network approximation capacity, the longer the old samples may be retained and reused by the algorithm.

2.4 Bootstrapping and Convergence

Q-learning uses “Bootstrapping” in Q-value updates: it uses Q-value estimate of next state to update the Q-value estimate of current state, instead of using only rewards or complete returns (methods that only use complete returns are instead called “Monte Carlo methods”).

Since Q-Learning is both off-policy and bootstrapping, it is not guaranteed to converge when using function approximators such as neural networks for the Q-function [Szepesvári 2010].

In fact, the three conditions “function approximation”, “bootstrapping”, “off-policy training” are named “The Deadly Triad” when they occur together in value based methods, see [Sutton and Barto 2018, Ch.11]. Despite that, it is believed that Q-Learning may converge if the behavior policy is sufficiently close to the target policy, like in ϵ-greedy policies with small ϵ. In many experiments Q-Learning showed to work well, such as in [Mnih et al. 2013] where a variant of Q-Learning with a convolutional neural network as a function approximator was trained to play Atari video games.

2.5 Increasing Gradient Stability: Target Network

When Q_ψ is a function approximator such as a neural network, the fact that it is used both to compute the prediction (and hence changes at each gradient descent iteration) and the target, makes the bootstrapped value y a “moving target” (literally!). This slows down learning because the gradient descent trajectory is wandering too much. To stabilize the trajectory of the gradient descent it is better to do a minibatch update with a big number of samples. To further stabilize it, it is useful to observe the fact that it is an off-policy algorithm: each tuple of $\langle s_t, a_t, s_{t+1}, r_t \rangle$ can be saved to a “replay buffer” and used in future (as we wrote in section 2.3): sampling randomly from the replay buffer instead of using the recently experienced trajectory will avoid to having correlated samples that would bias the gradient. In addition to this, to decrease even more the “moving target” effect it would be better to run some cycles of minibatch updates using a “target network” which is a copy of the Q-network that is not updated at every gradient descent step, only after a certain number of steps, and hence it is more stable.

Algorithm 2.2 **Deep Q-Learning with Replay Buffer and Target Network**

Require: an empty replay buffer B

Require: Step size η

Require: Initialize parameters ψ of network Q_ψ with small random values

do:

copy target network parameters $\psi' \leftarrow \psi$

do N times:

using some behavior policy collect transitions $\{\langle s_i, a_i, s_{i+1}, r_i \rangle\}$, add it to B

do K times:

sample a minibatch of M transitions $\langle s_i, a_i, s_{i+1}, r_i \rangle$ from B

for each i of M transitions do (all can be done at once if vectorized):

$y_i = r + \gamma \; max_{a_{i+1}} Q_{\psi'}(s_{i+1}, a_{i+1})$

$\psi \leftarrow \psi + \eta \; (\; y_i - Q_\psi(s_i, a_i) \;) \nabla_\psi Q_\psi(s_i, a_i)$

end of for each

end of do

end of do

end of do

2.6 Polyak Averaging

The stability of the algorithm with target network can be improved if, instead of copying completely the parameters from original q-value network to target network after some round of updating, the target network gets a small update towards the original q-network every time that the original q-network is updated. The target network updates are such to make the target network a little more similar to the original network, but not too fast. For instance an update of the target network may be:

$$\psi' \leftarrow \varsigma \psi' + (1 - \varsigma) \, \psi$$
$$with \; 0 \leq \varsigma \leq 1 \quad (a \; good \; value \; may \; be \; \varsigma = 0.99) \tag{2.8}$$

This is named Polyak Averaging.

Algorithm 2.3 **Deep Q-Learning with Replay Buffer, Target Network, Polyak Averaging**

Require: an empty replay buffer B

Require: Step size η

Require: Polyak Step size ς

Require: Initialize parameters ψ of network Q_ψ with small random values

copy target network parameters $\psi' \leftarrow \psi$

do N times:

 using some behavior policy collect transitions $\{\langle s_i, a_i, s_{i+1}, r_i\rangle\}$, add it to B

 do K times:

  sample a minibatch of M transitions $\langle s_i, a_i, s_{i+1}, r_i\rangle$ from B

  for each i of M transitions do (all can be done at once if vectorized):

   $y_i = r + \gamma\ max_{a_{i+1}} Q_{\psi'}(s_{i+1}, a_{i+1})$

   $\psi \leftarrow \psi + \eta\ \ (\ y_i - Q_\psi(s_i, a_i)\ \ )\nabla_\psi Q_\psi(s_i, a_i)$

   $\psi' \leftarrow \varsigma\psi' + (1-\varsigma)\ \psi$

  end of for each

 end of do

end of do

2.7 Action-Value Overestimation

Another problem of Q-Learning is that it learns a Q-value that instead of being the maximum of the expectation of the action-value is biased towards the estimate of the maximum of the action-value and hence overestimates the action-value (the maximum of an expectation is different from the expectation of the maximum).

To see how that happens, let us create a toy scenario. Imagine we are in a certain state s_0 in which you have five actions available: a_1, a_2, a_3, a_4, a_5. Taking a_1 will give a very low reward and lead to a state s_1 with $max_{a'}\ Q(s_1, a')$ equal to some constant K. Taking a_2 will give sometimes a very high reward (50% of the times) and sometimes a medium reward (50% of the times) and lead to a state s_2 with a $max_{a'}\ Q(s_2, a')$ equal to the same K as s_1. Actions a_3, a_4 and a_5 will behave in the same way as a_2, except they will lead respectively to states s_3, s_4

and s_5, which have the same $max_{a'}\, Q(s', a')$ as s_1 and s_2 (that is K). Let us put that in numbers:

Action in s_0	Reward	Leads to State	$max_{a'}\, Q(s', a')$
a_1	0 (Probability 1.0)	s_1	$max_{a'}\, Q(s_1, a') = K$
a_2	30 (Probability 0.5)	s_2	$max_{a'}\, Q(s_2, a') = K$
	10 (Probability 0.5)		
a_3	30 (Probability 0.5)	s_3	$max_{a'}\, Q(s_3, a') = K$
	10 (Probability 0.5)		
a_4	30 (Probability 0.5)	s_4	$max_{a'}\, Q(s_4, a') = K$
	10 (Probability 0.5)		
a_5	30 (Probability 0.5)	s_5	$max_{a'}\, Q(s_5, a') = K$
	10 (Probability 0.5)		

Table 2.1

So we can compute the expected values for each action:

$$EQ\ (s_0, a_1) = 0 * 1.0 + \gamma\ max_{a'}\ Q(s_1, a') = 0 + \gamma K = \gamma K$$
$$EQ\ (s_0, a_2) = 30 * 0.5 + 10 * 0.5 + \gamma\ max_{a'}\ Q(s_2, a') = 20 + \gamma K$$
$$EQ\ (s_0, a_3) = 30 * 0.5 + 10 * 0.5 + \gamma\ max_{a'}\ Q(s_3, a') = 20 + \gamma K$$
$$EQ\ (s_0, a_4) = 30 * 0.5 + 10 * 0.5 + \gamma\ max_{a'}\ Q(s_4, a') = 20 + \gamma K$$
$$EQ\ (s_0, a_5) = 30 * 0.5 + 10 * 0.5 + \gamma\ max_{a'}\ Q(s_5, a') = 20 + \gamma K$$

Hence, the $maximum\ of\ expectation\ of\ Q\ with\ respect\ to\ the\ actions$ for the state s_0 is:

$$max_{a'}\ EQ(s_0, a') = 20 + \gamma K$$

But if instead we run the Q-Learning algorithm we will find a different value. For instance imagine we do 15 runs starting from state s_0 , we choose each action three times, and because of random chances we get the following rewards:

Run	Chosen action	Reward r
1	a_1	0
2	a_1	0
3	a_1	0
4	a_2	10
5	a_2	30
6	a_2	30
7	a_3	10
8	a_3	10
9	a_3	10
10	a_4	30
11	a_4	30
12	a_4	30
13	a_5	10
14	a_5	30
15	a_5	10

Table 2.2

Given these outcomes, we can see that randomly, due to the probabilistic nature of rewards, in certain cases the target value $y = r + \gamma\, max_{a'} Q_\psi(s', a')$ (eq. 2.3) is greater than the expected value $EQ\ (s_0, a')$ and sometimes it is smaller. For instance for action a_5 the target value is twice $y = 10 + \gamma K$ and only once $y = 30 + \gamma K$. The Q-learning algorithm will then tend to compute a value for $Q_\psi(s_0, a_5)$ that is smaller than $EQ\ (s_0, a_5)$. For action a_4 instead, all three runs obtained the maximum reward, hence the target value is $y = 30 + \gamma K$ all the three times, that is greater than the expected value of $20 + \gamma K$, so the Q-learning algorithm will tend to compute a value for $Q_\psi(s_0, a_4)$ that is bigger than $EQ\ (s_0, a_4)$. At first glance this may seem ok, because we expect that due to random nature of runs, some values will have an estimate greater than their true expectation, and some a lesser one. But the problem is that the Q-Value is the maximum among all the estimates, so if the Q-value estimate for the optimal action (the one that has actually the greater $EQ\ (s_t, a')$) is greater than its expected value, it will drive the value of s_t up. But if instead the Q-value estimate for the optimal action is lesser than its expected value, it is not certain that it will drive the value of s_t down, because another

action (with expected value lower or equal), due to randomness, may have had an estimate greater than the estimate of the optimal action, and that will still drive the value of s_t up. The value of a state (computed as the maximum among its Q-values) is used to bootstrap the Q-values of the states that reach it, propagating the value overestimation to other states.
In our example we know that:

$$max_a\, EQ(s_0, a) = EQ\,(s_0, a_2) = EQ\,(s_0, a_3) = EQ\,(s_0, a_4) = EQ\,(s_0, a_5)$$

We also know that, from the samples in our example:

$$Q_\psi(s_0, a_2) > EQ\,(s_0, a_2)$$

$$Q_\psi(s_0, a_3) < EQ\,(s_0, a_3)$$

$$Q_\psi(s_0, a_4) > EQ\,(s_0, a_4)$$

$$Q_\psi(s_0, a_5) < EQ\,(s_0, a_5)$$

$$Q_\psi(s_0, a_4) > Q_\psi(s_0, a_2)$$

And we know that $V(s_0)$ is estimated by:

$$max_a\, Q_\psi(s_0, a) = Q_\psi(s_0, a_4) > max_a\, EQ(s_0, a)$$

Hence even if for optimal actions we had an equal number of runs with target greater and with target lesser than expected, the estimate of s_0 value $max_a\, Q_\psi(s_0, a)$ is greater than the real value, and through bootstrapping will propagate the overestimation to any other Q-values of states that reach s_0.
So in this toy example we see how random fluctuations may give returns that are either lesser or greater than true average returns but while all times in which the return of optimal action is greater than its expectation it drives the $max_a\, Q_\psi(s, a)$ up, not all times in which the return of optimal action is lesser than its expectation it drives the $max_a\, Q_\psi(s, a)$ down. This creates an overestimation bias that propagates to other Q-values through bootstrapping.

2.8 Double Q-Learning

A formal mathematical proof of the overestimation, as well as the proposal of an algorithm named "*Double Q-Learning*" that mitigates the problem can be found in [van Hasselt 2010], the application of that algorithm to deep neural networks can be found in [van Hasselt et al. 2016].

In Double Q-Learning the algorithm uses two different action-value function estimators, e.g. two neural networks that we may call A and B. When computing the target value, one neural network (let us say A) is used to check which action has the maximum value, while the other network (let us say B) is used to actually obtain the Q-value of that action. Then the target value is used to update the Q-value of the former network (A, in this case).

At each iteration the estimators may swap roles: the one that previously was used to evaluate which action has the maximum value may now used to obtain the Q-value of the optimal action, while the estimator who was previously used to get the Q-value of the optimal action may now used to find which action has the maximum value, and be subject to Q-value update. The roles of the two networks A and B may be decided randomly or in a fixed way after a certain number of iterations. For clarity's sake in the pseudocode algorithm we use the names "network $Q_{\psi 1}$" and "network $Q_{\psi 1}$" instead of A and B.

Algorithm 2.4 **Deep Double Q-Learning with Replay Buffer & Target Network**

Require: an empty replay buffer B

Require: Step size η

Require: Initialize parameters $\psi 1$ of network $Q_{\psi 1}$ with small random values

Require: Initialize parameters $\psi 2$ of network $Q_{\psi 2}$ with small random values

do:

  copy target network parameters $\psi 1' \leftarrow \psi 1$

  copy target network parameters $\psi 2' \leftarrow \psi 2$

  do N times:

    using some behavior policy collect transitions $\{\langle s_i, a_i, s_{i+1}, r_i \rangle\}$, add it to B

    do K times:

      sample a minibatch of M transitions $\langle s_i, a_i, s_{i+1}, r_i \rangle$ from B

      choose (e.g. randomly) either UPDATE(1) or UPDATE(2)

      for each i of M transitions do (all can be done at once if vectorized):

        if UPDATE(1) then:

          $a_{i+1}^* = \underset{a_{i+1}}{arg\,max}\ Q_{\psi 1'}(s_{i+1}, a_{i+1})$

          $y_i = r + \gamma Q_{\psi 2'}(s_{i+1}, a_{i+1}^*)$

          $\psi 1 \leftarrow \psi 1 + \eta\ \left(y_i - Q_{\psi 1}(s_i, a_i) \right) \nabla_{\psi 1} Q_{\psi 1}(s_i, a_i)$

        else if UPDATE(2) then:

          $a_{i+1}^* = \underset{a_{i+1}}{arg\,max}\ Q_{\psi 2'}(s_{i+1}, a_{i+1})$

          $y_i = r + \gamma Q_{\psi 1'}(s_{i+1}, a_{i+1}^*)$

          $\psi 2 \leftarrow \psi 2 + \eta\ \left(y_i - Q_{\psi 2}(s_i, a_i) \right) \nabla_{\psi 2} Q_{\psi 2}(s_i, a_i)$

        end if

      end of for each

    end of do

  end of do

end of do

2.9 Issues with Continuous Actions

It has to be noted that the Q-Learning algorithms presented above can be used only with a set of discrete actions. The problem with continuous actions is the computation of the maximum q-value among all possible actions: when we have a set of discrete actions it is possible to check the q-value of all actions and pick the maximum, but if the actions are continuous we would need an analytic way to compute the maximum of the q-function, and that is not possible if the q-function estimate is represented by a neural network. There have been many attempts to use workarounds to extend Q-Learning to continuous actions, but for a matter of space we will not describe them here: some of them are viable in certain scenarios but not effective in others, such as discretizing the action space which suffers of the curse of dimensionality. A better way to do Reinforcement Learning with continuous actions is to use Policy Based methods, that are detailed in the following chapter. In particular, one of those methods named Deep Deterministic Policy Gradient (Ch. 9), is considered to be applying to continuous actions the same basic strategy that Q-Learning applies to discrete actions: the ideas of having a deterministic target policy and of improving it greedily.

2.10 Sarsa

If you do not need to learn the optimal policy but you only want to evaluate the q-values for a fixed policy π , you can use Sarsa: an algorithm that is similar to Q-Learning where instead of bootstrapping with the maximum among the $Q_\psi(s_{i+1}, a_{i+1})$ for all possible $a_{i+1} \in A$, you just sample a_{i+1} following the policy $\pi(\cdot\,|s_{t+1})$. The trajectory of actual steps followed by the agent can be decided by the fixed policy π (on-policy) but can also follow a different behavior policy (off-policy), but a behavior policy too different from target policy may increase divergence risk. Given a sample transition $\langle s, a, s', r\rangle$ obtained by any policy (same of π , or different from π), the update rule is:

$$
\begin{gathered}
a' \sim \pi(\cdot\,|s') \\
y = r + \gamma\, Q_\psi(s', a')
\end{gathered}
\tag{2.9}
$$

The update rule for the tabular version (TD Learning) is:

$$Q_\psi(s,a) \leftarrow Q_\psi(s,a) + \eta\ (\ y - Q_\psi(s,a)\ )$$

(2.10)

The update rule for the function approximator version is:

$$L = (\ y - Q_\psi(s,a)\ )^2$$
$$\psi \leftarrow \psi + \eta\ \ (\ y - Q_\psi(s,a)\ )\nabla_\psi Q_\psi(s,a)$$

(2.11)

Since to estimate q-values we use a tuple $\langle s, a, r, s', a' \rangle$, we understand where the name of the algorithm comes from.

Algorithm 2.5 **Deep Sarsa – Q-Value Estimation Only (simple version)**

Require: Step size η

Require: Initialize parameters ψ of network Q_ψ with small random values

Do until (supposed) convergence:

    using some behavior policy collect a transition $\langle s_i, a_i, s_{i+1}, r_i \rangle$

    sample $a_{i+1} \sim \pi(\cdot\ |s_{i+1})$

    $y_i = r_i + \gamma\ Q_\psi(s_{i+1}, a_{i+1})$

    $\psi \leftarrow \psi + \eta\ \ (\ y_i - Q_\psi(s_i, a_i)\ \ )\nabla_\psi Q_\psi(s_i, a_i)$

end of do

If the actions are discrete, it is also possible to improve the policy π : for each state the improved policy can select the action that gives the greater q-value, as we were doing with q-value in eq. 2.7 : $a_{chose}\ = argmax_a Q_\psi(s,a)$

This way of improving the policy anyway is much slower than q-learning because it bootstraps the q-value using next-state q-value of the action chosen by the fixed policy and not the maximum q-value, so the information of maximum q-value is slower to propagate.

It is not possible (or better, it is not easy nor recommended) to improve policies in this way when actions are continuous, as we already discussed about q-learning that would be possible only with workarounds and there are better alternatives.

If we have discrete actions, an improved version of Sarsa is possible. "Expected Sarsa" bootstraps with the expectation of $Q_\psi(s_{i+1}, a_{i+1})$: instead of just sampling a_{i+1} from the target policy and compute $Q_\psi(s_{i+1}, a_{i+1})$.

$$y = r + \gamma \sum_{a} \pi(a|s')\, Q_{\psi}(s', a') \tag{2.12}$$

This implies a lower variance and faster convergence.

When the target policy is the greedy policy and the exploration policy is e-greedy, Expected Sarsa (with policy improvement) coincides with Q-Learning.

Algorithm 2.6 **Deep Expected Sarsa – Q-Value Estimation Only (simple version)**

Require: Step size η

Require: Initialize parameters ψ of network Q_{ψ} with small random values

Do until (supposed) convergence:

  using some behavior policy collect a transition $\langle s_i, a_i, s_{i+1}, r_i \rangle$

  $y_i = r_i + \gamma \sum_a \pi(a|s_{i+1})\, Q_{\psi}(s_{i+1}, a_{i+1})$

  $\psi \leftarrow \psi + \eta \ (y_i - Q_{\psi}(s_i, a_i) \ ) \nabla_{\psi} Q_{\psi}(s_i, a_i)$

end of do

If we do not need to improve/learn a policy, but we only need to estimate its q-values, we can use Sarsa estimation also with continuous actions (but we must use a function approximator such as a neural network, since we cannot use tabular methods with continuous actions). This can be useful in other algorithms, such as in Deep Deterministic Policy Gradient (Ch. 9).

3. Policy Gradient

3.1 Policy Based Methods

Policy based methods, also called "Policy Gradient methods" and "Policy Optimization methods" (those two expressions have diverse meaning but often are used interchangeably), differently from the value based methods, may or may not use value or action-value functions, and are characterized instead by having a parametrized policy function that, given the state/observation as input, returns a probability for each action, without explicitly assigning expected return values.

In other terms, the policy function $\pi(\cdot\,|s_t)$ is a function that is explicitly parametrized by θ (for instance θ may be the weights of a neural network), such that if at time t the state is s_t , it will return the probability distribution over which action to take a_t :

$$a_t \sim \pi_\theta(\cdot\,|s_t) \tag{3.1}$$

That is like equation 1.1 with the addition of parameters θ.

One way in which a neural network can output a probability distribution over a set of discrete actions is to have a last layer with as many neurons as the number of different actions, and on top of that operate a Softmax. If instead the actions are continuous then the last layer of the neural network will have as many neurons as continuous parameters in the action, and each neuron will be considered the mean of the distribution of that action parameter (the distribution type may be arbitrary, for instance a normal usually works well). When actions are continuous the covariance matrix of the distribution is usually considered diagonal (so to sample each action parameter independently form the others), and it may be of a fixed value established a priori, or it may be a value outputted by other neurons of the network. If the variance of each continuous action is outputted by neurons, it is better to output it in form of log-standard deviation, so there is no need to do any workaround to ensure that it is non-negative. When deciding which action to take, the actual action to be taken will be sampled from the distribution outputted by the neural network as we just described.

The problem of Reinforcement Learning is still the same, we want to maximize the expected return $J(\pi)$, (detailed in equation 1.5), but this time we would like to modify directly the policy parameters θ. One way of doing that could be to do gradient ascent using the gradient of the expected return $J(\pi)$ with respect to the parameters θ:

$$\theta_{k+1} \leftarrow \theta_k + \eta \nabla_\theta J(\pi_{\theta_k}) \tag{3.2}$$

With η learning rate.
So it is necessary to compute the gradient of the expected return $J(\pi_{\theta_k})$ with respect to θ .

Apparently that could seem problematic because the expected return $J(\pi_{\theta_k})$ is an expectation with respect to the distribution of the trajectories, that depends both on the policy and on the transition function, but we assume to not know the transition function so we do not know the distribution of trajectories (or the distribution of states). Hence, we do not even know how a change of the policy may change the distribution of trajectories. But we need to estimate how the expected return changes as a consequence of changes of policy (due to changes of θ), and this seems to be problematic because, as just said, the effect of the policy change on the trajectories distribution is unknown. Fortunately, as we will see in next chapter, there is a way to derive the gradient of the expected return $J(\pi_{\theta_k})$ that does not involve the effect of the change of policy on states/trajectories distribution, so this is not really a problem.

3.2 Policy Gradient Basics

A basic algorithm that improves the policy through gradient ascent on the expected return $J(\pi_\theta)$ with respect to the policy parameters θ is the one called "**Vanilla Policy Gradient**" [OpenAi 2018A] (or just "Policy Gradient"), that I describe below. It is very similar to the earlier "REINFORCE" [Williams 1992], with the difference that REINFORCE updates the policy parameters at each step while Vanilla Policy Gradient updates parameters using minibatches of steps. It is an on-policy algorithm. Some use the term "Vanilla Policy Gradient" as a family of algorithms, with REINFORCE being one of them (see [Peters and Schaal 2008]), where the policy gradient is not manipulated (as it happens instead in Natural Policy Gradient or in Proximal Policy Optimization, which I describe in later chapters).

Following the method in [OpenAi 2018A] it is useful to recall from calculus that the derivative of $\log(x)$ with respect to x is $1/x$. Then, because of chain rule, the derivative of $\log(f(x))$ with respect to x is the derivative of $f(x)$ divided by $f(x)$, that is $\log\big(f(x)\big)' = f(x)'/f(x)$. Applying that using the trajectory distribution $P(\tau|\theta)$ instead of $f(x)$, and rearranging, we obtain the so-called "*log derivative trick*":

$$\nabla_\theta P(\tau|\theta) = P(\tau|\theta)\nabla_\theta \log P(\tau|\theta) \tag{3.3}$$

Now, taking the definition of the probability of a trajectory τ made of T steps from equation 1.4, we compute the log of it:

$$\log P(\tau|\theta) = \log \rho_0(s_0) + \sum_{t=0}^{T-1} \log P(s_{t+1}|s_t, a_t) + \log \pi_\theta(a_t|s_t) \tag{3.4}$$

At this point we want to compute $\nabla_\theta \log P(\tau|\pi)$. We note that the terms $P(s_{t+1}|s_t, a_t)$ and $\rho_0(s_0)$ are environmental and that do not depend on θ, so their gradient is zero. Hence:

$$\nabla_\theta \log P(\tau|\theta) = \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \tag{3.5}$$

Now, computing the gradient of equation 1.5 we obtain:

$$\begin{aligned} \nabla_\theta J(\pi_\theta) &= \nabla_\theta E_{\tau\sim\pi_\theta}[G(\tau)] \\ &= \nabla_\theta \int_\tau P(\tau|\theta)G(\tau) \\ &= \int_\tau \nabla_\theta P(\tau|\theta)G(\tau) \end{aligned} \tag{3.6}$$

And applying equation 3.3 (log derivative trick):

$$= \int_{\tau} P(\tau|\theta) \nabla_\theta \log P(\tau|\theta)\, G(\tau)$$

$$= E_{\tau\sim\pi_\theta}[\nabla_\theta \log P(\tau|\theta)\, G(\tau)]$$

(3.7)

Applying equation 3.5 :

$$\nabla_\theta J(\pi_\theta) = E_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, G(\tau)\right]$$

■

(3.8)

The same formula can be obtained also in a different way through the *Policy Gradient Theorem* [Sutton et al. 1999], [Sutton and Barto 2018, Ch.13] and then *REINFORCE* [Williams 1992] which I will not detail here.

What is remarkable here is that even though the expected return depends on the state distribution (which in turn depends on the policy and hence on the parameters θ), the gradient of the expected return does not depend on the gradient of the state distribution (which we do not know).

We obtained the gradient of the expected return in form of an expectation. This means that if we can sample what is contained inside the expectation, we can use it to estimate the gradient. It turns out that we can do that: the expectation is with respect to the trajectory τ , whose probability distribution is determined by the environment and by our policy π_θ , so by making the agent follow the policy we can collect the returns $G(\tau)$ and compute the estimate of the expectation. Since the agent must follow the policy π_θ the algorithm is on-policy .

If we run N different episodes or trajectories $\{\tau_0, \tau_1, .., \tau_{N-1}\}$, with respective trajectory lengths $\{T_0, T_1, .., T_{N-1}\}$ we can do minibatch gradient ascent. If we call the actions and states of trajectory i at time t respectively with $a_{i,t}$ and $s_{i,t}$ the following is mean gradient:

$$\hat{g} = \frac{1}{N}\sum_{i=0}^{N}\sum_{t=0}^{T_i-1} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\, G(\tau_i)$$

(3.9)

Then we can update the weights of the policy neural network:

$$\theta' \leftarrow \theta + \eta \hat{g}$$

(3.10)

Algorithm 3.1 **Policy Gradient**

Require: Policy network step size η

Require: Initialize parameters θ of network π_θ with small random values

For $k = 0, 1, 2, \ldots$ do:

collect trajectories $D_k = \{ \tau_i \}$ using policy $\pi_k(\theta_k)$

compute total rewards $G(\tau_i)$ for each τ_i

estimate policy gradient as:

$$\widehat{g_k} = \frac{1}{|D_k|} \sum_{\tau \epsilon \quad k} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, G(\tau)$$

update the policy network with gradient ascent (or other methods like Adam):

$$\theta_{k+1} \leftarrow \theta_k + \eta \widehat{g_k}$$

end of for

3.3 Rewards-To-Go

A notable fact is that the gradient of the log probability of each action a_t is multiplied by the complete return $G(\tau)$. The complete return of the trajectory includes the rewards that have been received before that action, and for which the action has not any responsibility. From a logical standpoint we would expect that the gradient of the log probability of a certain action is multiplied only by the sum of the rewards obtained after that action, instead also the rewards obtained before that action are used. As we will see, the part of gradient of the expected return that involves the multiplication for the rewards obtained before the action is equal to zero in expectation, so we can use only the rewards obtained after the action. To prove it, I must digress a little and introduce the Expected Grad-Log-Prob (EGLP) lemma, from [OpenAi 2018A].

EGLP Lemma: suppose that f_θ is a parametrized distribution over a random variable x , then:

$$E_{x\sim f_\theta}[\nabla_\theta \log f_\theta(x)] = 0 \tag{3.11}$$

Proof (noting that the integral of a probability distribution is always = 1 by definition):

$$\nabla_\theta \int_x f_\theta(x) = \nabla_\theta 1 = 0 \tag{3.12}$$

Now, applying the “log derivative trick” equation 3.3 :

$$\begin{aligned} 0 &= \nabla_\theta \int_x f_\theta(x) \\ &= \int_x \nabla_\theta f_\theta(x) \\ &= \int_x f_\theta(x)\, \nabla_\theta \log f_\theta(x) \\ \therefore\ 0 &= E_{x\sim f_\theta}[\nabla_\theta \log f_\theta(x)] \end{aligned}$$

$\blacksquare$

(3.13)

(Optional: this proof is analogous to the proof that the expected value of statistical score of a distribution conditioned on the true value of its parameters is equal to zero. In fact since the statistical score is the gradient of the log-likelihood, when θ are the true values of the parameters of the distribution we could substitute the likelihood function $\mathcal{L}(\theta; X)$ in place of $f_\theta(x)$ in equation 3.11 , that means that the probability of X would be $\mathcal{L}(\theta; X)$, and we will obtain the same result, that in this case would mean that expected value of statistical score conditioned on true θ is zero).

We can decompose $G(\tau)$ in 2 sums, one of the rewards before the action and one after the action happened at time t (this proof is elaborated from the one by [Soemers 2019], there exists also another proof by [OpenAi 2018B]).

We use $G(\tau_{0:t-1})$ for the past rewards:

$$G(\tau_{0:t-1}) = \sum_{k=0}^{t-1} \gamma^k r_k \tag{3.14}$$

And we call $G(\tau_t)$ “reward-to-go”.

$$G(\tau_t) = \sum_{k=t}^{T-1} \gamma^k r_k \tag{3.15}$$

$$G(\tau) = G(\tau_{0:t-1}) + \; G(\tau_t) \tag{3.16}$$

We can then decompose equation 3.8 in the same way:

$$\begin{aligned}
\nabla_\theta J(\pi_\theta) &= E_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, G(\tau_{0:t-1}) + \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, G(\tau_t)\right] \\
&= E_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, G(\tau_{0:t-1})\right] + E_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, G(\tau_t)\right]
\end{aligned} \tag{3.17}$$

Now, this can be rewritten as:

$$part1 \; = E_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, G(\tau_{0:t-1})\right] \tag{3.18}$$

$$part2 = E_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, G(\tau_t)\right] \tag{3.19}$$

$$\nabla_\theta J(\pi_\theta) = part1 + part2$$

(3.20)

Now, because of EGLP lemma:

$$E_{a_t \sim \pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t)] = 0$$

(3.21)

Now, from 3.18 we have:

$$part1 \ = E_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, G(\tau_{0:t-1})\right]$$

$$= \sum_{t=0}^{T-1} E_{\tau\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t)\, G(\tau_{0:t-1})]$$

(3.22)

We know that $\nabla_\theta \log \pi_\theta(a_t|s_t)$ and $G(\tau_{0:t-1})$ are independent because the action depends only on the state at time t by definition, and $G(\tau_{0:t-1})$ depends only on states and actions before time t . So, we can separate the expectation of the multiplication into a multiplication of expectations:

$$= \sum_{t=0}^{T-1} E_{\tau\sim\pi_\theta}[G(\tau_{0:t-1})]\, E_{\tau\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t)]$$

(3.23)

Now we plug equation 3.21 in 3.23:

$$= \sum_{t=0}^{T-1} E_{\tau\sim\pi_\theta}[G(\tau_{0:t-1})] \cdot 0$$

$$\therefore part1 = 0$$

(3.24)

So we obtain:

$$\nabla_\theta J(\pi_\theta) = E_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, G(\tau_t)\right]$$

(3.25)

$$\nabla_\theta J(\pi_\theta) = E_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{l=t}^{T-1} \gamma^l r_l\right]$$

(3.26)

This permits us to use, for each action, only the rewards obtained after it (the rewards-to-go) to calculate the gradient of the expected return. Using the complete returns would not be wrong, because it is equal in expectation, but it would introduce much more variance in the gradient and that would slow down learning. Using $\gamma = 1$ would assure that each action has the same importance, it can be used in finite-horizon (episodic learning).

Algorithm 3.2 **Policy Gradient with Rewards-To-Go**

Require: Policy network step size η

Require: Initialize parameters θ of network π_θ with small random values

For $k = 0, 1, 2, \ldots$ do:

collect trajectories $D_k = \{\tau_i\}$ using policy $\pi_k(\theta_k)$

compute rewards-to-go $G(\tau_{i,t})$, for each t of each τ_i

estimate policy gradient as:

$$\widehat{g_k} = \frac{1}{|D_k|} \sum_{\tau \epsilon D_k} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, G(\tau_t)$$

update the policy network with gradient ascent (or other methods like Adam):

$$\theta_{k+1} \leftarrow \theta_k + \eta \widehat{g_k}$$

end of for

3.4 Bootstrapping the Value

In infinite time horizon it is not possible to collect the whole return (the trajectory never ends), so in that case it is necessary to bootstrap $G(\tau_t)$ using only the reward for the first step (or for some steps) and then summing a discounted estimate of the state value.
For instance, in infinite-horizon, a k-step bootstrap, using a function approximator $V_\psi(s)$, parametrized by ψ , to estimate the value function, would be:

$$\widehat{G_k}(\tau_t) = \sum_{l=t}^{t+k-1} \gamma^l r_l + \gamma^{t+k} V_\psi(s_{t+k}) \tag{3.27}$$

$\widehat{G_n}(\tau_t)$ is equal in expectation to $G(\tau_t)$, so it is theoretically correct to use it, but since in any algorithm we are not using the real state value but an approximation, it will introduce bias (but it will allow to work with infinite-horizon cases).

In infinite horizon case moreover, since we don't use the whole infinite trajectory at once but only n-step of it a time, we have to consider the infinite trajectory as equivalent to an infinite set of n-step episodes. So, every n-step we will reset t to 0, that is also necessary to avoid that the discount makes progressively less relevant the subsequent rewards. Or, if we don't want to reset t to 0, just change the equation a little, subtracting t to the exponent:

$$\widehat{\widehat{G_k}}(\tau_t) = \sum_{l=t}^{t+k-1} \gamma^{l-t} r_l + \gamma^{k} V_\psi(s_{t+k}) \tag{3.28}$$

In this way the finite and infinite horizon are included in the same framework.

3.5 Beyond Rewards-To-Go

Also $Q^{\pi_\theta}(s_t, a_t)$ is theoretically correct to be used instead of $G(\tau_t)$, because since inside the expectation the actions are distributed following the policy π_θ , it implies that $Q^{\pi_\theta}(s_t, a_t)$ is equal in expectation to $G(\tau_t)$ (another formal proof by OpenAI may be found in [OpenAi 2018C]).

In fact, it is theoretically correct also to use, instead of $G(\tau_t)$, any other expression that is equal in expectation. Every expression that starts with $G(\tau_t)$ or $\widehat{G_n}(\tau_t)$ and then adds any other expression that does not depend on the current action (but may depend on current state, since the current action and any function depending only on the current state are conditionally independent given the current state), is equal in expectation. That is because of EGLP lemma, and we already used it to show that $G(\tau_{0:t-1})$ may be or may be not added to $G(\tau_t)$ without changing the expectation (see how we obtained equation 3.25).
So, more generally:

$$\nabla_\theta J(\pi_\theta) = E_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, \Phi_t\right] \tag{3.29}$$

With Φ_t that may be one of:

$$G(\tau_t)$$

$$\widehat{G_k}(\tau_t)$$

$$\widehat{\widehat{G_k}}(\tau_t)$$

$$Q^{\pi_\theta}(s_t, a_t)$$

$$G(\tau_t) + b(s_t)$$

$$\widehat{G_k}(\tau_t) + b(s_t)$$

$$\widehat{\widehat{G_k}}(\tau_t) + b(s_t)$$

$$Q^{\pi_\theta}(s_t, a_t) + b(s_t) \tag{3.30}$$

Where $b(s_t)$, usually called “baseline”, is an additional expression that may depend on state and may be negative. For example, it may be $V^{\pi_\theta}(s_t)$ (in practice we would use its estimate $V_\psi(s_t)$).
So, many different Φ_t are possible. For instance if we start from $Q^{\pi_\theta}(s_t, a_t)$ and we add $b(s_t)$ = $-V^{\pi_\theta}(s_t)$, we end up with the Advantage function $\Phi_t = A^{\pi_\theta}(s_t, a_t) = Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t)$, see equation 1.13 .

3.6 Using the Advantage Function

Using the advantage function as Φ_t makes sense because the policy gradient update is intended to increase the probability of actions that perform better than the policy and to decrease the probability of actions that perform worse than the policy, if we use the Advantage function as Φ_t we are sure to have positive updates when the action is better than current policy choice, and negative updates when it is worse. This will have the effect of reducing variance in the gradient updates and hence speed up learning.

In case of using an advantage function, it is necessary to have a neural network $V_\psi(s)$, parametrized by ψ , that estimates the value function. That neural network will be updated with a minibatch gradient descent, similar to the Q-network gradient descent of equations 2.3 and 2.4 , where the target value y may be computed by $G(\tau_t)$, the Monte Carlo rewards-to-go of state t (this would have a great variance) or may be computed bootstrapping the value using $V_\psi(s')$ (in that case it will not be full gradient descent but semi-gradient, since the value of the next state will be considered as a constant and not as a function of ψ).
The value network update will be the following:

$$y = r + \gamma V_\psi(s')\ if\ bootstrapping$$

$$or$$

$$y = G(\tau_t) = \sum_{k=t}^{T-1} \gamma^k r_k\ \ if\ using\ Monte\ Carlo$$

$$L = (\ y - V_\psi(s)\ )^2$$

$$\psi \leftarrow \psi + \eta\ \ (\ y - V_\psi(s)\ )\nabla_\psi V_\psi(s) \tag{3.31}$$

Policy gradient methods that use an estimate of the value function to bootstrap the estimate of the total return such in equation 3.27 and 3.28 are called "**actor-critic**", where actor is referred to the policy function and critic to the value function [Sutton and Barto 2018, Ch.13]. Some authors call "actor-critic" every policy method that uses an estimate of the value function as component of Φ_t , but Sutton and Barto specify that the term should be exclusive for those that use value function to bootstrap the estimate of the total return. That means when generally, for a fixed k and a certain baseline $b(s_t)$:

$$\Phi_t = \sum_{l=t}^{t+k-1} \gamma^{l-t} r_l + \gamma^k V_\psi(s_{t+k}) - b(s_t)$$

(3.32)

It is common to do that with $k = 1$ and with the estimate of the value function used as baseline, so that:

$$\Phi_t = r_t + \gamma V_\psi(s_{t+1}) - V_\psi(s_t) ,$$

(3.33)

Which is equivalent to a bootstrapped advantage function.

Algorithm 3.3 **Policy Gradient with rewards-to-go and advantage function**

Require: Policy network step size η

Require: Value network step size ω

Require: Initialize parameters θ of network π_θ with small random values

Require: Initialize parameters ψ of network V_ψ with small random values

For $k = 0, 1, 2, \ldots$ do:

collect trajectories $D_k = \{ \tau_i \}$ using policy $\pi_k(\theta_k)$

compute rewards-to-go $G(\tau_{i,t})$, for each t of each τ_i

compute advantage estimates $\widehat{A_t}$ using current estimate of value function $V_{\psi k}$:

$\widehat{A_t} = G(\tau_t) - V_{\psi k}(s_t)$

estimate policy gradient as:

$$\widehat{g_k} = \frac{1}{|D_k|} \sum_{\tau \epsilon D_k} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, \hat{A}_t$$

update the policy network with gradient ascent (or other methods like Adam):

$\theta_{k+1} \leftarrow \theta_k + \eta \widehat{g_k}$

For $n = 0, 1, 2, \ldots, N-1$ do a value function gradient descent iteration:

estimate the value function gradients as:

$$\widehat{h_k} = \frac{1}{|D_k|} \sum_{\tau \epsilon D_k} \frac{1}{T} \sum_{t=0}^{T-1} \nabla_\psi (V_\psi(s_t) - G(\tau_t))^2$$

update the value function with a gradient descent step (or other method):

$\psi_{k+1} \leftarrow \psi_k + \omega \widehat{h_k}$

end of for

end of for

3.7 Benefits of Policy Gradients

A remarkable aspect of policy gradient methods is that compared to value based methods they have a more "natural" distribution over actions: the policy function outputs "by design" a distribution on actions, and during training the distribution smoothly changes towards one that improves the returns. In value based methods instead the estimate of value function or q-function only gives expected (action-)state returns, so we have to choose an artificial way to create a distribution on actions, such as with ϵ-greedy. Hence in value base methods when some action that previously seemed not optimal starts to seem better than all the other actions (that is when its q-value estimate becomes the greater among all actions), we have an abrupt change in the policy: the policy stops suggesting the previously-considered-best action (except for a small probability depending on ϵ) and starts suggesting almost greedily the new best action. In policy gradient methods instead the change in distribution is smooth because it is due to gradual changes through the policy gradient ascent, and this also implies an automatic gradual passage from exploration to exploitation.

This “natural” distribution over actions of policy methods, is also great in adversarial applications where an antagonist may understand that the agent is always deterministically taking a certain action which is deemed the best, and prepare a counter-action. If the agent has a value based policy, it will have a deterministic or a quasi-deterministic policy, so it may be predictable, while if the agent instead has a probability distribution over actions (a stochastic policy) the antagonist cannot be sure about which action is going to be taken by the agent (in Game Theory distributions over actions at state s are named “*mixed actions*”).

3.8 Improvements

It must be noted that in Policy Gradient algorithm the policy ascent direction is often not the most efficient because each parameter of θ may have a different magnitude of influence over the final policy distribution, but we are updating all parameters by the same step size. We will describe the problem and present an algorithm that computes a better direction in Ch. 6 with Natural Policy Gradient.

One thing to be careful of in Policy Gradient is the fact that the when the policy step size η is too little, the learning will be slow. So, it should be big enough to make the learning progress quick, but not so big to make the new policy worse than the old. In fact the new policy may end up being quite close in parameters space to the old policy, but even small differences in parameters may result in big differences in policies. Hence a small step in parameter space may correspond to a big step in policy space, in a way that worsens the policy instead of improving it. An algorithm that aims at computing the right step size (together with a better direction) is Trust Region Policy Optimization, in Ch. 7 .

A further improvement for all Policy Gradient methods is the Generalized Advantage Estimation, that is a way to compute a better Advantage Function which can be tuned in bias and variance. It is not a policy gradient algorithm in its own, it is just an improved Advantage Function that can be used in any policy gradient algorithm. It is presented in Chapter 5.

4. Off-Policy Policy Gradient

4.1 Motivations for Off-Policy

As it is evident watching the Policy Gradient algorithm, each trajectory is used only for one step of optimization in the policy network gradient ascent. This is because it is an on-policy algorithm: since the gradient of the expected return (equation 3.29) is an expectation with respect to the distribution of samples generated by the policy π_θ , the generated samples can be used only once, to improve π_θ . They cannot be used again to improve the next policy $\pi_{\theta'}$ (the new policy resulted from the optimization step) because they come from a different distribution, the one generated by the old policy. It may be tempting to use each trajectory more than once, but as [Schulman et al. 2017] report: "*doing so is not well-justified, and empirically it often leads to destructively large policy updates*". A method that instead allows to use each trajectory more than once is Proximal Policy Optimization, described in Ch. 8, which permits us to use the samples generated by the previous policy distribution if the new optimized policy distribution is not too different. It is still considered an on-policy algorithm, but you will see that it is also *slightly off-policy*, because it reuses recent samples, if the policy is not too different. Proximal Policy Optimization is based on the same theoretical framework of Trust Region Policy optimization, so before reading Ch. 8 it is necessary to read Ch. 7 at least until eq. 7.31

It would be even better if we could use not only the trajectories generated by recent policies (which do not differ much from the current policy), but also use trajectories generated by any policy, without caring of the difference with our target policy. In other words, it would be great to do off-policy learning. That could allow us to reuse all past trajectories, or to choose behavior policies that differ from the target policy in order to explore some particular state space, or even to use trajectories obtained by just observing other agents (Offline Reinforcement Learning). This could greatly speed up learning.
Intuitively, to do policy gradient using samples from a different policy, it is necessary to do *importance sampling* (explained in Appendix C), as described by [Degris et al. 2012], [Levine and Koltun 2013] and [Levine 2021b]: since the trajectory has been generated by the behavior policy but has a different probability of occurring if we used the target policy, we must do importance sampling using the ratio of the probability of the target policy π_θ to the behavior policy π_β (where β are the parameters of the neural network for the behavior policy).

4.2 The Approximated Off-Policy Policy Gradient

Let us derive the off-policy gradient formally following [Levine 2021c]:

Recalling from equation 1.3 the total return of a trajectory τ:

$$G(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t$$

And the probability of a trajectory τ from eq. 1.4:

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t)$$

The expected return under the behavior policy π_β is :

$$J(\pi_\beta) = E_{\tau\sim\pi_\beta}[G(\tau)] \tag{4.1}$$

Using importance sampling under the behavior policy π_β we can express the expected return for the target policy π_θ as:

$$J(\pi_\theta) = E_{\tau\sim\pi_\beta}\left[\frac{P(\tau|\pi_\theta)}{P(\tau|\pi_\beta)} G(\tau)\right] \tag{4.2}$$

From that we can compute the gradient:

$$\nabla_\theta J(\pi_\theta) = \nabla_\theta E_{\tau\sim\pi_\beta}\left[\frac{P(\tau|\pi_\theta)}{P(\tau|\pi_\beta)} G(\tau)\right]$$

$$= E_{\tau\sim\pi_\beta}\left[\nabla_\theta \left(\frac{P(\tau|\pi_\theta)}{P(\tau|\pi_\beta)} G(\tau)\right)\right]$$

$$\textit{and since only } P(\tau|\pi_\theta) \textit{ depends on } \theta:$$

$$= E_{\tau\sim\pi_\beta}\left[\frac{\nabla_\theta P(\tau|\pi_\theta)}{P(\tau|\pi_\beta)} G(\tau)\right]$$

$$applying\ the\ log-derivative\ trick\ of\ eq. 3.1:$$

$$= E_{\tau\sim\pi_\beta}\left[\frac{P(\tau|\pi_\theta)}{P(\tau|\pi_\beta)}\nabla_\theta logP(\tau|\pi_\theta)G(\tau)\right]$$

$$= E_{\tau\sim\pi_\beta}\left[\frac{\rho_0(s_0)\prod_{t=0}^{T-1}P(s_{t+1}|s_t,a_t)\pi_\theta(a_t|s_t)}{\rho_0(s_0)\prod_{t=0}^{T-1}P(s_{t+1}|s_t,a_t)\pi_\beta(a_t|s_t)}\nabla_\theta logP(\tau|\pi_\theta)G(\tau)\right]$$

$$initial\ state\ and\ transition\ probabilities\ are\ the\ same\ for\ both\ policies\ and\ get\ cancelled:$$

$$= E_{\tau\sim\pi_\beta}\left[\left(\prod_{t=0}^{T-1}\frac{\pi_\theta(a_t|s_t)}{\pi_\beta(a_t|s_t)}\right)\nabla_\theta logP(\tau|\pi_\theta)G(\tau)\right]$$

$$= E_{\tau\sim\pi_\beta}\left[\left(\prod_{t=0}^{T-1}\frac{\pi_\theta(a_t|s_t)}{\pi_\beta(a_t|s_t)}\right)\left(\sum_{t=0}^{T-1}\nabla_\theta \log\pi_\theta(a_t|s_t)\right)G(\tau)\right]$$

(4.3)

That is basically the common policy gradient of eq. 3.8 with importance sampling applied using the trajectory probabilities, where only the policy probabilities for actions are used in the ratio. We can improve that formula using the rewards-to-go (as in section 3.2) to decrease variance: instead of multiplying the weighted gradient of log probability by the total return $G(\tau)$, we multiply it only by the sum of the rewards in which the current action has a causality relationship, that is from the current time t to the end of the trajectory. This means that also the weighing must be changed accordingly, multiplying each reward by the multiplication of only the policy ratios from the beginning of the trajectory to the time of the reward:

$$\nabla_\theta J(\pi_\theta) = E_{\tau\sim\pi_\beta}\left[\sum_{t=0}^{T-1}\nabla_\theta \log\pi_\theta(a_t|s_t)\left(\sum_{l=t}^{T-1}\gamma^l r_l\left(\prod_{t\prime=0}^{l}\frac{\pi_\theta(a_{t\prime}|s_{t\prime})}{\pi_\beta(a_{t\prime}|s_{t\prime})}\right)\right)\right]$$

$$common\ multiplication\ terms\ before\ t\ may\ be\ grouped\ together\ out\ of\ summation$$

$$= E_{\tau\sim\pi_\beta}\left[\sum_{t=0}^{T-1}\nabla_\theta \log\pi_\theta(a_t|s_t)\left(\prod_{t\prime=0}^{t}\frac{\pi_\theta(a_{t\prime}|s_{t\prime})}{\pi_\beta(a_{t\prime}|s_{t\prime})}\right)\left(\sum_{l=t}^{T-1}\gamma^l r_l\left(\prod_{t''=t+1}^{l}\frac{\pi_\theta(a_{t\prime\prime}|s_{t\prime\prime})}{\pi_\beta(a_{t\prime\prime}|s_{t\prime}\ )}\right)\right)\right]$$

(4.4)

Let us name:

$$\Pi_{current} = \prod_{t'=0}^{t} \frac{\pi_\theta(a_{t'}|s_{t'})}{\pi_\beta(a_{t'}|s_{t'})}$$

$$\Pi_{after} = \prod_{t''=t+1}^{l} \frac{\pi_\theta(a_{t''}|s_{t''})}{\pi_\beta(a_{t''}|s_{t'}\ )}$$

(4.5)

The first multiplication grouping $\Pi_{current}$ is the importance weight for the actions until current time t, without any future action weighing, while the second multiplication grouping Π_{after} is inside the inner of the summation and multiplies each reward r_l by the importance weight of actions after the current time t until the time of the reward l (included).
This way of computing the gradient has still some issue: the multiplication of the policy probability ratios $\Pi_{current}$ and Π_{after} can grow exponentially large.

To mitigate that it is possible to ignore Π_{after} and drop that multiplication: the resulting formula will not be a proper gradient anymore but it may still improve the policy because it will be equivalent to an update of policy iteration [Levine 2021c], even if with some exception that we will analyze later.

$$\nabla_{\theta}^{\approx} J(\pi_\theta) = E_{\tau\sim\pi_\beta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\left(\prod_{t'=0}^{t} \frac{\pi_\theta(a_{t'}|s_{t'})}{\pi_\beta(a_{t'}|s_{t'})}\right)\left(\sum_{l=t}^{T-1} \gamma^l r_l\right)\right]$$

(4.6)

I use the symbol $\nabla_{\theta}^{\approx}$ to make it clear that it is an approximation. That is equivalent to the gradient of expected return following the target policy π_θ until (and including) the "current" timestamp t and then supposing that the agent will follow the behavior policy π_β (because importance sampling is not applied to those steps).

To see if this still improves the target policy, even if after t there is not importance sampling and it is like the expectation follows the behavior policy β, let us examine three cases:

A) $V^\theta(s_{t+1}) = V^\beta(s_{t+1})$
B) $V^\theta(s_{t+1}) < V^\beta(s_{t+1})$
C) $V^\theta(s_{t+1}) > V^\beta(s_{t+1})$

In case (A) the value of the next state will be the same, so the approximate gradient will be equal to the true gradient.
In case (B) the approximated gradient is a half-way between the gradient of the target policy and the gradient of a better policy ! So, in some sense it will update the current policy towards choosing the action in s_t that would be the best if the actions for states after t were chosen by the better policy. So basically, in case (B) the update will make the target policy behave more like a better policy, as if actions for next states were already improved, “believing” that in future also the actions taken in next states will converge towards the ones of the better policy.
In case (C) the approximate gradient is smaller than the true gradient. We can identify three subcases. Subcase (C1): if the approximate gradient has the same sign of the true gradient, but a smaller magnitude, the target policy will still improve, even if slower. Subcase (C2): if the approximate gradient has the same sign but a bigger magnitude, that may happen only if the true gradient is negative, the update may be too big, even if in the right direction, and that may slow down learning (maybe even distort the policy). Subcase (C3): if the approximate gradient has the opposite sign of the true gradient, that may happen when $V^{\beta}(s_{t+1})$ is negative, and so big in magnitude to cancel the positivity of the part of the approximate gradient computed following the target policy. In this case the update would go in the opposite direction of the correct update, and this will jeopardize learning.

So, we see that in the two subcases (C2) and (C3) we may incur in some issues.
We may wonder if, adding $V^{\beta}(s_{t+1})$ (that is the expectation of $\sum_{l=t+1}^{T-1} \gamma^l r_l$) instead of $V^{\theta}(s_{t+1})$ (that is the expectation of $\sum_{l=t+1}^{T-1} \gamma^l r_l \left(\prod_{t''=t+1}^{l} \frac{\pi_\theta(a_{t''}|s_{t''})}{\pi_\beta(a_{t''}|s_{t''})}\right)$) to $\gamma^t r_t$, it is like adding $V^{\theta}(s_{t+1})$ plus a baseline = ($V^{\beta}(s_{t+1}) - V^{\theta}(s_{t+1})$) that would mean that we are just using a baseline and so what we add would be zero in expectation. But this is not the case, because what we are adding depends on a_t , not only on s_t, (because we area adding the expected value of s_{t+1} , not of s_t) so we are not adding a proper baseline and this mean that we are actually distorting the gradient (as in subcase C2, or in subcase C3 with the possibility of an upgrade in the wrong direction !).

Now, a further approximation would be to use as importance sampling ratio only the ratio of the probability of choosing the action a_t under the target policy to the behavior policy, dropping the multiplication of the ratios for the previous time. That is:

$$\nabla_{\theta}^{\approx} J(\pi_\theta) = E_{\tau\sim\pi_\beta}\left[\sum_{t=0}^{T-1} \frac{\pi_\theta(a_t|s_t)}{\pi_\beta(a_t|s_t)} \nabla_\theta \log \pi_\theta(a_t|s_t) \left(\sum_{l=t}^{T-1} \gamma^l r_l\right)\right] \tag{4.7}$$

Since $E_{\tau\sim\pi_\beta}\left[\sum_{l=t}^{T-1} \gamma^l r_l\right]$ is equivalent to $E_{\tau\sim\pi_\beta}\left[Q^\beta(a_t|s_t)\right]$ we can write it also as:

$$\nabla_{\theta}^{\approx} J(\pi_\theta) = E_{\tau\sim\pi_\beta}\left[\sum_{t=0}^{T-1} \frac{\pi_\theta(a_t|s_t)}{\pi_\beta(a_t|s_t)} \nabla_\theta \log \pi_\theta(a_t|s_t)\, Q^\beta(a_t|s_t)\right] \tag{4.8}$$

Also this approximation has not a theoretical justification.

4.3 Alternative Derivation

There is another way of computing the same approximation of the off-policy policy gradient, presented by [Degris et al. 2012].

If we name $d_\beta(s)$ the hypothetical stationary distribution of behavior policy π_β :

$$d_\beta(s) = lim_{t\to\infty} Prob(s_t = s|s_0; \beta) \tag{4.9}$$

Let us define a performance indicator where the states are occurring following the behavior policy π_β but the action-value is computed as if actions followed the target policy π_θ:

$$\begin{aligned} J_\beta(\pi_\theta) &= \sum_{s}^{S} d_\beta(s) \sum_{a}^{A} Q^\theta(a,s)\, \pi_\theta(a|s) \\ &= E_{s\sim d_\beta}\left[\sum_{a}^{A} Q^\theta(a,s)\, \pi_\theta(a|s)\right] \end{aligned} \tag{4.10}$$

We can rewrite it to make more evident that in $Q^\theta(a,s)$ the action a is distributed with policy π_θ as well as any subsequent action in the computation of Q^θ :

$$J_\beta(\pi_\theta) = E_{s\sim d_\beta}\left[\sum_a^A E_{s'\sim P}\left[R(s,a,s') + \gamma E_{a'\sim\theta}\left[Q^\theta(s',a')\right]\right]\pi_\theta(a|s)\right]$$

(4.11)

Its gradient with respect to θ is:

$$\nabla_\theta J_\beta(\pi_\theta) = \nabla_\theta E_{s\sim d_\beta}\left[\sum_a^A Q^\theta(a,s)\,\pi_\theta(a|s)\right]$$

$$\textit{applying the chain rule of calculus}:$$

$$= E_{s\sim d_\beta}\left[\sum_a^A \left(Q^\theta(a,s)\nabla_\theta\pi_\theta(a|s) + \pi_\theta(a|s)\nabla_\theta Q^\theta(a,s)\right)\right]$$

(4.12)

The equation 4.12 above is the basis on which [Degris et al. 2012] derive their version of off-policy policy gradient. Unfortunately, it is not theoretically justified to compute the gradient in that way if the policy function is approximated (as with a neural network).
Let us give an intuitive explanation about that.
Affirming that doing gradient ascent on $\nabla_\theta J_\beta(\pi_\theta)$ optimizes policy π_θ is like asserting that:

$$J_\beta(\pi_{\theta'}) \geq J_\beta(\pi_\theta)\ \textit{implies}\ J(\pi_{\theta'}) \geq J(\pi_\theta)$$

(4.13)

Where $J(\pi_\theta)$ is, as defined in equation 1.5, the expected return with states obtained following policy π_θ , while $J(\pi_{\theta'})$ is the expected return with states obtained following policy $\pi_{\theta'}$.
We know that when the policy function is approximated, as when using a neural network to express the policy function, a change in parameters may not only change the policy function for the states in the trajectory used by the algorithm, but also for other states (this is called "aliasing"). Maybe that parameters change will improve the returns from a state that is frequently visited under that policy, but at the same time it will worsen the returns from a state that is rarely visited under the same policy, and as a whole the policy will improve because the rarely visited state has little impact to the whole expected return.
Imagine that to increase $J_\beta(\pi_\theta)$ the changes we make in the policy π_θ are such that they increase the value $V^\theta(s_a)$ of some state s_a that occurs very frequently on π_β and very rarely

on π_θ but at the same time those policy changes decrease the value $V^\theta(s_b)$ of some state s_b that occurs very rarely on π_β and very frequently on π_θ , that means we would have:

$$J_\beta(\pi_{\theta'}) > J_\beta(\pi_\theta) \; and \; J(\pi_{\theta'}) < J(\pi_\theta) \tag{4.14}$$

Which would contradict eq. 4.13.

Is that possible to happen with a gradient ascent if we have tabular policy ? No, because in tabular policies each state policy is independent from every other states' policies: changing the policy of a state will not change the policy of other states (there is not "aliasing"). That implies that increasing the value of a state s_a through a change in the policy on that state s_a will not decrease the value of another state s_b: either it will let the value of s_b the same (if from the state s_b it is not possible to reach the state s_a) or it will increase the value of s_b (if from the state s_b it is possible to reach the state s_a following the policy π_θ , that means that from s_b there is a probability greater than zero to reach a state that has increased its values, so also the total $V^\theta(s_b)$ has increased).

On the opposite, when using a function approximator like a neural network as a policy function, the change of the parameters θ resulting from a gradient ascent step that increases the value of some state s_a may result in modifying also the way the function approximator computes the policy for another state s_b (as already wrote, a phenomenon named by some authors "*aliasing*" or "*state-aliasing*"), possibly also decreasing the value $V^\theta(s_b)$.

So, the case of eq. 4.14 may happen and contradict eq. 4.13, that means that doing gradient ascent on $\nabla_\theta J_\beta(\pi_\theta)$ is not theoretically justified to improve policy π_θ when using neural networks as policy function approximators.

We do not need to do a dedicated analysis of the cases in which, as result of maximizing $J_\beta(\pi_\theta)$ (eq. 4.10), some states of policy π_θ decrease their frequency while other increase it. The reason is that if a state s_a decreases its frequency, another state s_b must increase its own, and if $V^\theta(s_a) > V^\theta(s_b)$ it will lead to decreasing the values of some states from which s_a was reachable, because some of them will reach more frequently than before the lower value state s_b and less frequently s_a. On the opposite, if $V^\theta(s_a) < V^\theta(s_b)$, some states from which s_a was reachable will increase their value. Anyway all those occurrences are just cases of either state value increase or decrease, and so they are already included in the analysis we just made when discussing if $J_\beta(\pi_{\theta'}) \geq J_\beta(\pi_\theta)$ implies $J(\pi_{\theta'}) \geq J(\pi_\theta)$.

Another discussion of the side effects of using a behavior policy to estimate the target policy in DPG, together with a proposed solution, can be found in [Liu et al. 2019].

Let us continue now with the derivation of off-policy policy gradient by [Degris et al. 2012] from eq 4.12 . Now, it is quite difficult to compute $\nabla_\theta Q^\theta(a,s)$, so an easier approximation could be to drop the second term in the summation: $\pi_\theta(a|s)\nabla_\theta Q^\theta(a,s)$. This will give us the following approximation:

$$\nabla_{\theta}^{\approx} J_\beta(\pi_\theta) = E_{s\sim d_\beta}\left[\sum_a^A \left(Q^\theta(a,s)\nabla_\theta \pi_\theta(a|s)\right)\right]$$

$$= E_{s\sim d_\beta}\left[\sum_a^A \left(\frac{\pi_\beta(a,s)}{\pi_\beta(a,s)}\frac{\pi_\theta(a,s)}{\pi_\theta(a,s)} Q^\theta(a,s)\nabla_\theta \pi_\theta(a|s)\right)\right]$$

$$= E_{s\sim d_\beta}\left[\sum_a^A \left(\pi_\beta(a,s)\frac{\pi_\theta(a,s)}{\pi_\beta(a,s)} Q^\theta(a,s)\frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a,s)}\right)\right]$$

$$= E_{s\sim d_\beta, a\sim\pi_\beta}\left[\frac{\pi_\theta}{\pi_\beta} Q^\theta(a,s)\nabla_\theta \log \pi_\theta(a|s)\right]$$

$$= E_\beta\left[\frac{\pi_\theta(a,s)}{\pi_\beta(a,s)} Q^\theta(a,s)\nabla_\theta \log \pi_\theta(a|s)\right] \tag{4.15}$$

At this point [Degris et al. 2012] introduce a further approximation replacing $Q^\theta(a,s)$ with the off-policy return, that in expectation is equivalent to $Q^\beta(a,s)$. So we obtain:

$$\nabla_{\theta}^{\approx} J_\beta(\pi_\theta) = E_\beta\left[\frac{\pi_\theta(a,s)}{\pi_\beta(a,s)} Q^\beta(a,s)\nabla_\theta \log \pi_\theta(a|s)\right] \tag{4.16}$$

It is easy to see that sampling that is equivalent to sampling $\nabla_{\theta}^{\approx} J(\pi_\theta)$ in eq. 4.8.

[Degris et al. 2012] in their “Off Policy Actor Critic” algorithm (also named “OffPAC”) give a partial justification for this approximation that is correct only for tabular representations of the policy and not for versions that use function approximations such as neural networks, that interests us when doing Deep Reinforcement Learning.

In Ch. 9 we will see the Deterministic Policy Gradient algorithm [Silver et al. 2014], which is an off-policy policy gradient method that does not use importance sampling.

Algorithm 4.1 **Off-Policy Policy Gradient with Rewards-To-Go**

Require: Policy network step size η

Require: Initialize parameters θ of network π_θ with small random values

Require: A behaviour policy π_β

For $k = 0, 1, 2, \dots$ do:

collect trajectories $D_k = \{ \tau_i \}$ using policy π_β

compute rewards-to-go $G(\tau_{i,t})$, for each t of each τ_i

estimate policy gradient as:

$$\widehat{g_k} = \frac{1}{|D_k|} \sum_{\tau \epsilon D_k} \sum_{t=0}^{T-1} \frac{\pi_\theta(a_t|s_t)}{\pi_\beta(a_t|s_t)} \nabla_\theta \log \pi_\theta(a_t|s_t)\, G(\tau_t)$$

update the policy network with gradient ascent (or other methods like Adam):

$$\theta_{k+1} \leftarrow \theta_k + \eta \widehat{g_k}$$

end of for

5. Generalized Advantage Estimation

[Schulman et al. 2016] introduced a particular version of the Advantage Function, the "Generalized Advantage Estimation", with proven variance reduction property. It is intended to be used with the policy gradient family of algorithms, using undiscounted rewards. So there is not a time-dependent parameter γ to discount the rewards, but there is another parameter named γ with a different meaning but same mathematical behaviour: it is meant to downweigh rewards corresponding to delayed effects. Thus, the meaning is different from the "discount", it does not mean that delayed effects are less "useful" at current time (that would be the meaning of the classical time-discount), but it still discounts the delayed effects, and doing so it introduces bias, because in this way the total return will be smaller in absolute value. On the other hand, it will decrease variance, for the same reason: smaller absolute value means smaller variance among different trajectories. So, the meaning of this parameter $\gamma \in [0,1]$ here is to be a switch for smoothly parametrize a trade-off between bias and variance, with bias increasing and variance decreasing when γ decreases, and bias decreasing and variance increasing when γ tends to 1.

Then this Generalized Advantage Estimation has another characteristic that decreases variance but introduces bias: it computes the estimate of the returns as an exponentially weighted average of k different n-step bootstrap estimates, where n goes from 1 to k. To be clearer, if we define:

$$\delta_t^V = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{5.1}$$

Imagine having a trajectory of length at least k>1, then we could build a 1-step bootstrap estimate of the advantage function at time t :

$$\hat{A}_t^{(1)} = r_t + \gamma V(s_{t+1}) - V(s_t) = \delta_t^V \tag{5.2}$$

But, if k>2 we could also build a 2-step bootstrap estimate, or if k>3 a 3-step estimate, or a k-step estimate etc. :

$$\hat{A}_t^{(2)} = r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) - V(s_t) = \delta_t^V + \gamma \delta_{t+1}^V$$

$$\hat{A}_t^{(3)} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ + \gamma^3 V(s_{t+3}) - V(s_t) = \delta_t^V + \gamma \delta_{t+1}^V + \gamma^2 \delta_{t+2}^V$$

...

$$\hat{A}_t^{(k)} = -V(s_t) + \sum_{l=t}^{t+k-1} \gamma^{l-t} r_l + \gamma^k V(s_{t+k}) = \sum_{l=t}^{t+k-1} \gamma^{l-t} \delta_t^V$$

(5.3)

If we recall eq. 3.28, we may also notice that:

$$\hat{A}_t^{(k)} = \widehat{\widehat{G_k}}(\tau_t) - V(s_t)$$

(5.4)

In infinite horizon, k may tend to ∞.

$$\hat{A}_t^{(\infty)} = -V(s_t) + \sum_{l=t}^{\infty} \gamma^{l-t} r_l = \sum_{l=t}^{\infty} \gamma^{l-t} \delta_l^V$$

(5.5)

Now, a 1-step estimate has low variance but high bias, while a 5-step estimate has bigger variance and lower bias, the more steps we include in the estimate the bigger the variance and the lower the bias. It is possible to compute an exponentially weighted average of all n-steps estimators, parametrized by λ such that when $\lambda = 0$ the weighted average coincides with $A_t^{(1)}$ and when $\lambda = 1$ it coincides with $A_t^{(\infty)}$, so that λ is another parameter that may be used to tune the bias/variance trade-off.

To obtain that, let us note that the series $\sum_{k=0}^{\infty} \lambda^k$, with $\lambda \in [0,1]$, is equal to $1/(1-\lambda)$. Hence $(1-\lambda)\sum_{k=0}^{\infty} \lambda^k = 1$. So, a summation of infinite terms, each weighted by $(1-\lambda)\lambda^k$, has the total sum of weights = 1.

The “Generalized Advantage Function” hence is:

$$\hat{A}_t^{GAE(\gamma,\lambda)} = (1-\lambda) \sum_{k=1}^{\infty} \lambda^{k-1} \hat{A}_t^{(k)}$$

(5.6)

And a more practical formulation may be obtained:

$$\hat{A}_t^{GAE(\gamma,\lambda)} = (1-\lambda)(\hat{A}_t^{(1)} + \lambda\hat{A}_t^{(2)} + \lambda^2\hat{A}_t^{(3)} + \cdots)$$

$$= (1-\lambda)(\delta_t^V + \lambda(\delta_t^V + \gamma\delta_{t+1}^V) + \lambda^2(\delta_t^V + \gamma\delta_{t+1}^V + \gamma^2\delta_{t+2}^V) + \cdots)$$

$$= (1-\lambda)(\delta_t^V(1+\lambda+\lambda^2+\lambda^3+\cdots) + \gamma\delta_{t+1}^V(\lambda+\lambda^2+\lambda^3+\lambda^4+\cdots) + \gamma^2\delta_{t+2}^V(\lambda^2+\lambda^3+\lambda^4+\lambda^5+\cdots) + \cdots)$$

$$= (1-\lambda)(\delta_t^V\left(\frac{1}{1-\lambda}\right) + \gamma\delta_{t+1}^V\left(\frac{\lambda}{1-\lambda}\right) + \gamma^2\delta_{t+2}^V\left(\frac{\lambda^2}{1-\lambda}\right) + \cdots)$$

$$= \sum_{l=t}^{\infty}(\lambda\gamma)^{l-t}\delta_t^V \tag{5.7}$$

This GAE hence can be used instead of the Advantage Function and put in place of Φ_t in the policy gradient (see equation 3.29) . This permits to tune variance and bias with two parameters: γ controls how far in time to consider the rewards relevant (the closer to 1, the farther in time), and λ that decides how much the return component of the advantage has to be similar to a 1-step bootstrap (if $\lambda=0$) or progressively to a Monte Carlo return (if $\lambda=1$).

6. Natural Policy Gradient

6.1 Motivation

To improve the speed of training in Policy Gradient one right thing to do can be to scale the gradient in a way that gradient ascent works better. It is possible that the gradient is not pointing to the direction of steepest ascent, and it may be convenient to modify or scale the gradient differently for each parameter in order to make it point to the direction of greater policy improvement, through the so-called "Natural Policy Gradient" [Kakade 2002], [Peters et al. 2003], [Bagnell and Schneider 2003] (also known as "Covariant Policy Gradient"), similarly to what has been studied in supervised learning [Amari 1998].

In fact, some parameters of the neural network affect the policy more than others: when we are doing gradient ascent with a certain learning rate, we basically are putting a limit on how much parameters can be changed, but this is not equivalent to put a limit on how much the policy can be changed. A faster learning would happen if we were able to fix the rate at which the policy is changed, and have larger rates for the parameters with little influence on the policy and smaller rates for the parameters with greater influence on the policy.

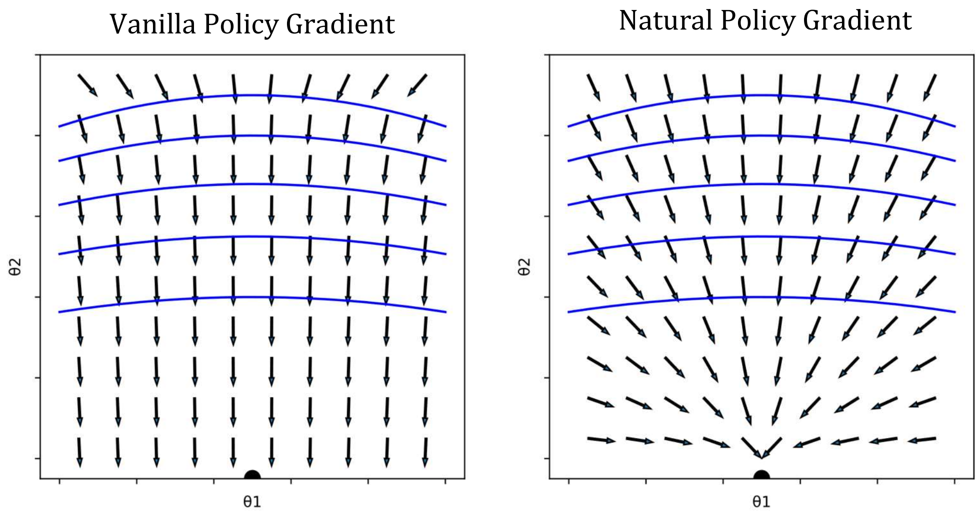

Figure 6.1

An hypothetical example of how gradients are computed in standard/vanilla Policy Gradient (on the left), and in Natural Policy Gradient (on the right). We have only two parameters here: $\theta 1$, represented on the horizontal axis, and $\theta 2$, represented on the vertical axis. Blue lines are zones of equivalent expected return (return landscape). Black arrows are normalized gradients. The black half-circle in the middle of the bottom is representative of the optimum expected return. In standard Policy Gradient we see how the gradient used for the update is not directed at the optimum. The original example can be found in [Peters et al. 2003].

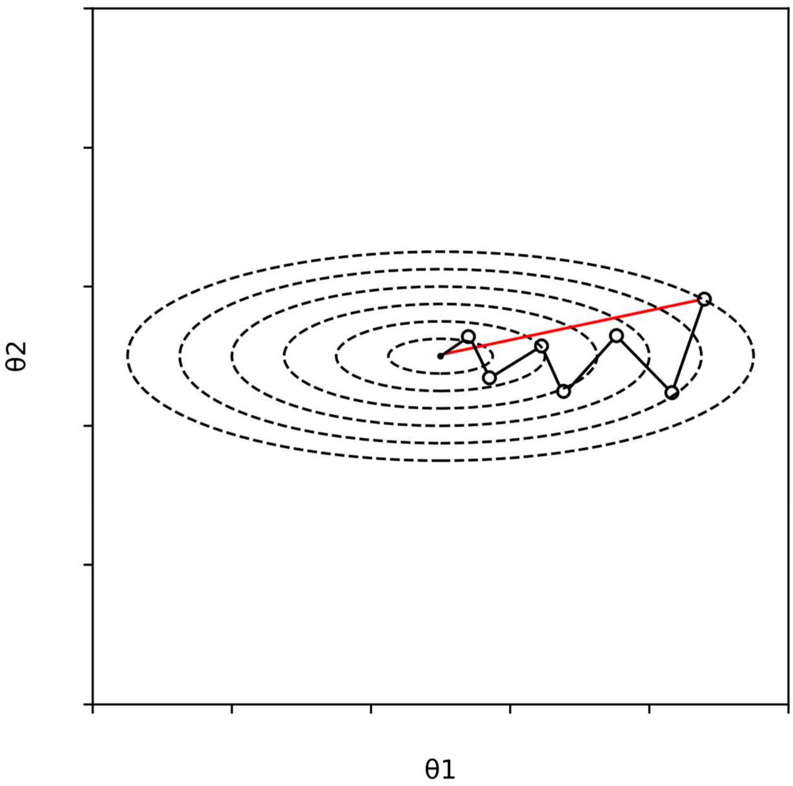

Figure 6.2

The analogous problem as it is usually depicted in Supervised Learning using gradient descent, where the dashed lines are zones of equivalent error function, the central dot is the optimum, the red line would be the steepest descent, while the taken path usually follows a more wandering trajectory, represented by the small circles connected by segments. The two axes represent the two parameters $\theta 1$ and $\theta 2$ of the machine learning system.

6.2 Constraining the New Policy

One way to compute a better gradient consists of putting a constraint to each gradient ascent step, such that the new policy and the old policy are not too different as distributions. An

appropriate computation for that could be the Kullback-Leibler divergence (see Appendix A.1), enforcing the constraint $D_{KL}(\pi_{\theta'}||\pi_{\theta}) < \epsilon$.
The Kullback-Leibler divergence is defined as $D_{KL}(\pi_{\theta'}||\pi_{\theta}) = E_{\theta'}[\log \pi_{\theta'} - \log \pi_{\theta}]$.
Second order Taylor expansion of KL divergence is approximated by (see Appendix A.6, eq. a.25):

$$D_{KL}(\pi_{\theta'}||\pi_{\theta}) \approx \frac{1}{2}(\theta' - \theta)^T \boldsymbol{F}_{\boldsymbol{\pi_\theta}}(\theta' - \theta) \tag{6.1}$$

Where $\boldsymbol{F}_{\boldsymbol{\pi_\theta}}$ is the Fisher-information matrix of the policy π_θ .
The Fisher-information matrix of a distribution is the expected value of the covariance matrix of the score, given parameters θ . The score is the gradient of log-likelihood with respect to the parameters (see Appendix A.3 and A.4). Hence:

$$\boldsymbol{F}_{\boldsymbol{\pi_\theta}} = E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s)\, \nabla_\theta \log \pi_\theta(a|s)^{\mathrm{T}}] \tag{6.2}$$

That can be approximated with samples from the trajectory:

$$\widehat{\boldsymbol{F}_{\boldsymbol{\pi_\theta}}} = \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta \log \pi_\theta(a_i|s_i)\ \nabla_\theta \log \pi_\theta(a_i|s_i)^{\mathrm{T}} \tag{6.3}$$

The common gradient ascent step for policy gradient is:

$$\theta' \leftarrow \theta + \eta\, \nabla_\theta J(\pi_\theta) \tag{6.4}$$

That can be seen as optimizing a constrained first order Taylor expansion of $J(\pi_\theta)$:

$$\theta' \leftarrow \underset{\theta'}{\arg max}\ \nabla_\theta J(\pi_\theta)^T(\theta' - \theta)$$

$$such\ that\ ||\theta' - \theta||^2 \leq \epsilon \tag{6.5}$$

That would imply:

$$\theta' \leftarrow \theta + \sqrt{\frac{\epsilon}{\|\nabla_\theta J(\pi_\theta)\|^2}}\, \nabla_\theta J(\pi_\theta)$$

(6.6)

But as I wrote before, instead of imposing a constraint over $\|\theta' - \theta\|^2$ it would be better to impose a constraint over the distributions, such as $D_{KL}(\pi_{\theta'}||\pi_\theta) < \epsilon$.

Given the relation between $D_{KL}(\pi_{\theta'}||\pi_\theta)$ and Fisher-information matrix from equation 6.1, and given that we can compute an approximation of $\boldsymbol{F}_{\boldsymbol{\pi}_{\boldsymbol{\theta}}}$ from samples (as in eq. 6.3), such a constraint in the KL divergence can be imposed in a gradient descent step with the following update [Peters and Schaal 2008], [Levine 2021]:

$$\theta' \leftarrow \theta + \eta\, \boldsymbol{F}_{\boldsymbol{\pi\theta}}^{-1} \nabla_\theta J(\pi_\theta)$$

$$with \;\; \eta = \sqrt{\frac{2\epsilon}{\nabla_\theta J(\pi_\theta)^T \boldsymbol{F}_{\boldsymbol{\pi}_{\boldsymbol{\theta}}}{}^{-1} \nabla_\theta J(\pi_\theta)}}$$

(6.7)

This is called “Natural gradient”, and its usage permits a faster learning with a smoother gradient ascent.

The computation of η as in eq. 6.7 is necessary only if you want to adhere to a strict a-priori fixed ϵ , otherwise you can use an arbitrarily small value for η : doing so will not change the direction of the change.

6.3 Derivation of the Formula

The derivation of η as in eq. 6.7 can be found in Appendix A.1 of [Peters 2007] and we propose it here in an equivalent form in the following lines, showing how to obtain eq. 6.7.

Let us define $\delta = (\theta' - \theta)$. So we can redefine the optimization problem, starting from eq. 6.5, using the Kullback-Leibler constraint, and the eq. 6.1 approximation, as:

$$\delta \leftarrow \underset{\delta}{\arg max} \; \nabla_\theta J(\pi_\theta)^T \; \delta$$

$$such\ that\ \frac{1}{2}(\theta' - \theta)^T \boldsymbol{F}_{\boldsymbol{\pi}_{\boldsymbol{\theta}}}(\theta' - \theta) < \epsilon \tag{6.8}$$

This is a constrained optimization problem, solvable by Lagrangian Multiplier method. The resulting Lagrangian is the following:

$$\Lambda(\delta, \lambda) \; = \; \nabla_\theta J(\pi_\theta)^T \; \delta + \; \lambda(\epsilon - \frac{1}{2}\delta^T \boldsymbol{F}_{\boldsymbol{\pi}_{\boldsymbol{\theta}}}\delta) \tag{6.9}$$

Now, all partial derivatives of $\Lambda(\delta, \lambda)$ should be zero.

$$\nabla_\delta \Lambda(\delta, \lambda) = \; \nabla_\theta J(\pi_\theta) - \lambda \frac{1}{2} 2 \; \boldsymbol{F}_{\boldsymbol{\pi}_{\boldsymbol{\theta}}}\delta = 0$$

$$\nabla_\theta J(\pi_\theta) = \lambda \; \boldsymbol{F}_{\boldsymbol{\pi}_{\boldsymbol{\theta}}}\delta$$

$$\lambda^{-1} {\boldsymbol{F}_{\boldsymbol{\pi}_{\boldsymbol{\theta}}}}^{-1} \nabla_\theta J(\pi_\theta) = \lambda^{-1} {\boldsymbol{F}_{\boldsymbol{\pi}_{\boldsymbol{\theta}}}}^{-1} \lambda \; \boldsymbol{F}_{\boldsymbol{\pi}_{\boldsymbol{\theta}}}\delta$$

$$\delta = \lambda^{-1} {\boldsymbol{F}_{\boldsymbol{\pi}_{\boldsymbol{\theta}}}}^{-1} \; \nabla_\theta J(\pi_\theta) \tag{6.10}$$

Now we plug the δ found in eq. 6.10 into the constraint $\frac{1}{2}\delta^T \boldsymbol{F}_{\boldsymbol{\pi}_{\boldsymbol{\theta}}}\delta = \epsilon$ and we have the dual function:

$$\frac{1}{2}\lambda^{-1}({\boldsymbol{F}_{\boldsymbol{\pi}_{\boldsymbol{\theta}}}}^{-1} \; \nabla_\theta J(\pi_\theta))^T \boldsymbol{F}_{\boldsymbol{\pi}_{\boldsymbol{\theta}}} \lambda^{-1} {\boldsymbol{F}_{\boldsymbol{\pi}_{\boldsymbol{\theta}}}}^{-1} \; \nabla_\theta J(\pi_\theta) = \epsilon$$

$$\frac{1}{2\epsilon}({\boldsymbol{F}_{\boldsymbol{\pi}_{\boldsymbol{\theta}}}}^{-1} \; \nabla_\theta J(\pi_\theta))^T \; \nabla_\theta J(\pi_\theta) = \lambda^2$$

$$because\ of\ matrix\ properties\ AB^T \; = B^T A^T$$

$$\lambda^2 = \frac{1}{2\epsilon} \nabla_\theta J(\pi_\theta)^T ({\boldsymbol{F}_{\boldsymbol{\pi}_{\boldsymbol{\theta}}}}^{-1})^T \; \nabla_\theta J(\pi_\theta)$$

$$since\ \boldsymbol{F}_{\boldsymbol{\pi_\theta}}\ is\ symmetric, also\ {\boldsymbol{F}_{\boldsymbol{\pi_\theta}}}^{-1}\ is\ symmetric, and\ for\ symmetric\ matrices\ A = A^T$$

$$\lambda = \sqrt{\frac{\nabla_\theta J(\pi_\theta)^T {\boldsymbol{F}_{\boldsymbol{\pi_\theta}}}^{-1}\, \nabla_\theta J(\pi_\theta)}{2\epsilon}} \tag{6.11}$$

With λ the Lagrange multiplier. Now we plug it into eq. 6.10:

$$\delta = \sqrt{\frac{2\epsilon}{\nabla_\theta J(\pi_\theta)^T {\boldsymbol{F}_{\boldsymbol{\pi_\theta}}}^{-1} \nabla_\theta J(\pi_\theta)}}\, {\boldsymbol{F}_{\boldsymbol{\pi_\theta}}}^{-1}\, \nabla_\theta J(\pi_\theta) \tag{6.12}$$

Since δ is the name we used for $(\theta' - \theta)$, we know that:

$$\theta' - \theta = \sqrt{\frac{2\epsilon}{\nabla_\theta J(\pi_\theta)^T {\boldsymbol{F}_{\boldsymbol{\pi_\theta}}}^{-1} \nabla_\theta J(\pi_\theta)}}\, {\boldsymbol{F}_{\boldsymbol{\pi_\theta}}}^{-1}\, \nabla_\theta J(\pi_\theta)$$

$$\theta' = \theta + \sqrt{\frac{2\epsilon}{\nabla_\theta J(\pi_\theta)^T {\boldsymbol{F}_{\boldsymbol{\pi_\theta}}}^{-1} \nabla_\theta J(\pi_\theta)}}\, {\boldsymbol{F}_{\boldsymbol{\pi_\theta}}}^{-1}\, \nabla_\theta J(\pi_\theta)$$

$$now\ if\ we\ name\ \eta = \sqrt{\frac{2\epsilon}{\nabla_\theta J(\pi_\theta)^T \boldsymbol{F}_{\boldsymbol{\pi_\theta}} \nabla_\theta J(\pi_\theta)}}\ we\ have:$$

$$\theta' = \theta + \eta {\boldsymbol{F}_{\boldsymbol{\pi_\theta}}}^{-1}\, \nabla_\theta J(\pi_\theta)$$

■

$$\tag{6.13}$$

The final 2 lines of eq. 6.13 are equivalent to eq. 6.7 .

Now, let us see the algorithm for Natural Policy Gradient, that as you may already understood is an on-policy algorithm because it updates the policy that has been used to generate samples and must use the latest policy to generate new trajectories.

Algorithm 6.1 **Natural Policy Gradient**

Require: Policy network step size η (optional)

Require: Value network step size ω

Require: Initialize parameters θ of network π_θ with small random values

Require: Initialize parameters ψ of network V_ψ with small random values

For $k = 0, 1, 2, \dots$ do:

collect trajectories $D_k = \{ \tau_i \}$ using policy $\pi_k(\theta_k)$

compute rewards-to-go $G(\tau_{i,t})$, for each t of each τ_i

compute advantage estimates $\widehat{A_t}$ using current estimate of value function $V_{\psi k}$:

$$\widehat{A_t} = G(\tau_t) - V_{\psi k}(s_t)$$

estimate policy gradient as:

$$\widehat{g_k} = \frac{1}{|D_k|} \sum_{\tau \epsilon D_k} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, \hat{A}_t$$

compute approximate average Fisher Information Matrix:

$$\widehat{\boldsymbol{F}_{\boldsymbol{\pi_\theta}}} = \frac{1}{\sum_{\tau \epsilon D_k} N_\tau} \sum_{\tau \epsilon D_k} \sum_{t=0}^{N_\tau - 1} \nabla_\theta \log \pi_\theta(a_t|s_t) \nabla_\theta \log \pi_\theta(a_t|s_t)^{\mathrm{T}}$$

compute update vector for policy parameters

$$\Delta_k = \boldsymbol{F}_{\boldsymbol{\pi\theta}}^{-\mathbf{1}}\, \widehat{g_k}$$

compute the step size η either with npg $\eta = \sqrt{\frac{2\epsilon}{\widehat{g_k}^T \boldsymbol{F}_{\boldsymbol{\pi_\theta}} \widehat{g_k}}}$ or with other method

update the policy network:

$$\theta_{k+1} \leftarrow \theta_k + \eta \Delta_k$$

For $z = 0, 1, 2, \dots, Z-1$ do a value function gradient descent iteration:

estimate the value function gradients as:

$$\widehat{h_k} = \frac{1}{|D_k|} \sum_{\tau \epsilon D_k} \frac{1}{T} \sum_{t=0}^{T-1} \nabla_\psi (V_\psi(s_t) - G(\tau_t))^2$$

update the value function with a gradient descent step (or other method):

$$\psi_{k+1} \leftarrow \psi_k + \omega \widehat{h_k}$$

end of for

end of for

7. Trust Region Policy Optimization

7.1 Problem formalization

The Trust Region Policy Optimization is an on-policy algorithm in which an alternative optimization problem is posed, such that allows to maximize the length of the step of the policy gradient ascent.

The Reinforcement Learning problem here is framed in a different formulation.
In the classical Policy Gradient formulation, we aim at maximizing $J(\pi_\theta)$, that for ease of reading we may now write $J(\theta)$. But equivalently, we may maximize the performance of a policy (the one to be maximized) with respect to a fixed policy (the one used until now to generate trajectories). If we call $\pi_{\theta'}$ the optimized policy (parametrized by the new parameters θ') and π_θ the fixed policy, we may want to maximize $J(\theta') - J(\theta)$.

Following [Schulman et al. 2015], it is useful to note that:

$$J(\theta') - J(\theta) \ = E_{\tau\sim\pi_{\theta'}}\left[\sum_t \gamma^t A^{\pi_\theta}(s_t, a_t)\right] \tag{7.1}$$

Proof:
Recall equation 1.12 and 1.13:

$$Q^{\pi_\theta}(s_t, a) = E_{s_{t+1}\sim P}[R(s_t, a, s_{t+1}) + \gamma V^{\pi_\theta}(s_{t+1})]$$

$$A^{\pi_\theta}(s_t, a) = Q^{\pi_\theta}(s_t, a) - V^{\pi_\theta}(s_t)$$

Then:

$$E_{\tau\sim\pi_{\theta'}}\left[\sum_t \gamma^t A^{\pi_\theta}(s_t, a_t)\right]$$

$$= E_{\tau \sim \pi_{\theta'}} \left[\sum_t \gamma^t (R(s_t, a, s_{t+1}) + \gamma V^{\pi_\theta}(s_{t+1}) - V^{\pi_\theta}(s_t)) \right]$$

$$= E_{\tau \sim \pi_{\theta'}} \left[-V^{\pi_\theta}(s_0) + \sum_t \gamma^t R(s_t, a, s_{t+1}) \right]$$

$$= E_{\tau \sim \pi_{\theta'}}[-V^{\pi_\theta}(s_0)] + E_{\tau \sim \pi_{\theta'}} \left[\sum_t \gamma^t R(s_t, a, s_{t+1}) \right]$$

$$distribution\ of\ s_0\ does\ not\ depend\ on\ policy\ but\ only\ on\ initial\ state\ distribution\ \rho_0 \Longrightarrow$$

$$= E_{s_0 \sim \rho_0}[-V^{\pi_\theta}(s_0)] + E_{\tau \sim \pi_{\theta'}} \left[\sum_t \gamma^t R(s_t, a, s_{t+1}) \right]$$

$$= E_{\tau \sim \pi_\theta}[-V^{\pi_\theta}(s_0)] + E_{\tau \sim \pi_{\theta'}} \left[\sum_t \gamma^t R(s_t, a, s_{t+1}) \right]$$

$$= -J(\theta) + J(\theta')$$

■

(7.2)

Now, equation 7.1 means that the difference between the expected return of the policy parametrized by θ' and the expected return of the policy parametrized by θ is equal to the expectation of the advantage function $A^{\pi_\theta}(s_t, a_t)$ over trajectories distributed by the policy parametrized by θ' . (Since the expectation of the trajectories follows the distribution by policy θ' , it implies that the actions a_t are distributed by $\pi_{\theta'}$, so $A^{\pi_\theta}(s_t, a_t)$ in eq. 7.1 it is the advantage function of the actions selected by the new policy θ' with respect to the fixed policy θ) .

We can improve the policy if we can maximize the right-hand side of eq. 7.1 (because the greater is the difference between the expected return of new policy and the expected return of the old policy, the greater is the right-hand side). To do that, one strategy is to have an equivalent equation in a form that we can sample, so that we can run gradient ascent with respect to the policy parameters.

7.2 Ideal Objective

We can define the probability of being in state s at time t, depending on policy parametrized by θ as $Prob(s_t = s \mid \theta)$.

Then we can define the frequency $\xi_\theta(s)$ of a state s as the number of times that s is expected to be visited, computed as unnormalized (it is a frequency, not a probability) and time-discounted, under the policy π_θ parametrized by θ :

$$\xi_\theta(s) = Prob(s_0 = s|\,\theta) + \gamma Prob(s_1 = s|\,\theta) + \gamma^2 Prob(s_2 = s|\,\theta) + \cdots \tag{7.3}$$

We can rewrite eq. 7.1 in a way to sum over time, then over states, and then over actions. Then in a second passage we can write it in a way to sum over states first, and in the last passage we plug eq. 7.3, referred to policy θ' , into it:

$$J(\theta') - J(\theta) \ = \sum_t \sum_s Prob(s_t = s \mid \theta') \sum_a \pi_{\theta'}(a_t|s_t)\, \gamma^t A^{\pi_\theta}(s_t, a_t)$$

$$J(\theta') - J(\theta) \ = \sum_s \sum_t \gamma^t Prob(s_t = s \mid \theta') \sum_a \pi_{\theta'}(a_t|s_t)\, A^{\pi_\theta}(s_t, a_t)$$

$$J(\theta') - J(\theta) \ = \sum_s \xi_{\theta'}(s) \sum_a \pi_{\theta'}(a_t|s_t)\, A^{\pi_\theta}(s_t, a_t)$$

$$J(\theta') \ = J(\theta) + \sum_s \xi_{\theta'}(s) \sum_a \pi_{\theta'}(a_t|s_t)\, A^{\pi_\theta}(s_t, a_t) \tag{7.4}$$

This equation entails that any new policy parametrized by θ' which in every state has a better or equal expected advantage function with respect to the previous policy parametrized by θ , i.e. $\sum_a \pi_{\theta'}(a_t|s_t)\, A^{\pi_\theta}(s_t, a_t) \geq 0$, improves the total expected return (or leaves the total expected return unchanged if the expected advantage function is zero at each state).

7.3 Local Approximation of Objective

Now, the right-hand side seems in a more manageable form to be maximized than eq. 7.1, but if we look carefully, we see that to sample it we would need to follow the frequency of states $\xi_{\theta'}(s)$, referred to the new policy θ' , but at that point we only know the old policy θ and we can sample only with that. Hence, we use a local approximation that uses the old policy for the frequencies (please note the usage of $\xi_{\theta}(s)$ instead of $\xi_{\theta'}(s)$):

$$L_{\xi\theta}(\theta') = J(\theta) + \sum_{s} \xi_{\theta}(s) \sum_{a} \pi_{\theta'}(a_t|s_t)\, A^{\pi_\theta}(s_t, a_t) \tag{7.5}$$

It has to be noted that (from eq. 7.1, 7.4 and eq. 7.5):

$$J(\theta') = J(\theta) + E_{\tau\sim\pi_{\theta'}}\left[\sum_{t} \gamma^t A^{\pi_\theta}(s_t, a_t)\right] = J(\theta) + E_{s\sim P(\tau|\pi_{\theta'})}\left[\sum_{t} \gamma^t E_{a\sim\pi_{\theta'}(a|s)}[A^{\pi_\theta}(s_t, a)]\right] \tag{7.6}$$

$$L_{\xi\theta}(\theta') = J(\theta) + E_{s\sim P(\tau|\pi_{\theta})}\left[\sum_{t} \gamma^t E_{a\sim\pi_{\theta'}(a|s)}[A^{\pi_\theta}(s_t, a)]\right] \tag{7.7}$$

This approximation $L_{\xi\theta}(\theta')$ can be used in place of $J(\theta')$ only when the new distribution $\pi_{\theta'}(a_t|s_t)$ is not too different from the old distribution $\pi_{\theta}(a_t|s_t)$, it must stay inside a "trust region" (hence the name of the algorithm).

7.4 Objective Lower Bound

Now we need to use the concept of Kullback-Leibler divergence so the reader who does not know about it may read Appendix A.1 before proceeding.
Considering the status s where the Kullback-Leibler divergence of $\pi_{\theta}(\cdot\,|s)$ from $\pi_{\theta'}(\cdot\,|s)$ is maximal, we define the value of such divergence as D_{KL}^{max} :

$$D_{KL}^{max}(\pi_\theta, \pi_{\theta'}) = max_s[D_{KL}(\pi_\theta(\cdot\,|s)||\pi_{\theta'}(\cdot\,|s))]$$

(7.8)

Then, we define the maximum absolute value that the Advantage function can possibly assume, in any possible pairs of state and action, as υ :

$$\upsilon = max_{s,a}|A^{\pi_\theta}(s,a)|$$

(7.9)

Then it can be proved, that

$$J(\theta') \geq L_{\xi\theta}(\theta') - C\ D_{KL}^{max}(\pi_\theta, \pi_{\theta'})$$

$$where\ C = \frac{4\,\upsilon\,\gamma}{(1-\gamma)^2}$$

(7.10)

We may consider C as a penalty coefficient: the bigger the difference between π_θ and $\pi_{\theta'}$, the bigger the penalty.
Now we define:

$$\bar{A}(s) = E_{a'\sim\pi_{\theta'}(\cdot|s)}[A^{\pi_\theta}(s,a')]$$

(7.11)

So we can rewrite eq. 7.6 and 7.7 as:

$$J(\theta') = J(\theta) + E_{s\sim P(\tau|\pi_{\theta'})}\left[\sum_t \gamma^t \bar{A}(s_t)\right]$$

(7.12)

$$L_{\xi\theta}(\theta') = J(\theta) + E_{s\sim P(\tau|\pi_\theta)}\left[\sum_t \gamma^t \bar{A}(s_t)\right]$$

(7.13)

To prove eq. 7.10, as in [Schulman et al. 2017], we must bound the difference between $J(\theta')$ and $L_{\xi\theta}(\theta')$.

To do that we begin measuring how much the two policies agree: we create a "coupled policy pair" that means that we create a joint distribution over a pair of policy actions (a, a'), where $a \sim \pi_\theta(\cdot\,|s)$ and $a' \sim \pi_{\theta'}(\cdot\,|s)$.

We define $(\pi_\theta\,,\ \pi_{\theta'})$ as an "$\alpha - coupled\,policy\,pair$" if for all states s the joint distribution $(a, a')|s$ is such that $Prob(a \neq a'|s) \leq\ \alpha$.

(Please note the graphical difference between a , the first letter of latin alphabet used to denote an action, and α , that is "alpha" from greek alphabet, used to denote the value of a probability boundary).

Now:

$$\bar{A}(s) = E_{a' \sim \pi_{\theta'}(\cdot|s)}[A^{\pi_\theta}(s, a')] = E_{(a,a') \sim (\pi_\theta\ ,\ \pi_{\theta'})}[A^{\pi_\theta}(s, a')]$$

$$since\ E_{a \sim \pi_\theta(\cdot|s)}[A^{\pi_\theta}(s, a)] = 0\ we\ can\ insert\ it\ into\ the\ expectation$$

$$\bar{A}(s) = E_{(a,a') \sim (\pi_\theta\ ,\ \pi_{\theta'})}[A^{\pi_\theta}(s, a') - A^{\pi_\theta}(s, a)]$$

$$when\ a = a', we\ have\ A^{\pi_\theta}(s, a') - A^{\pi_\theta}(s, a) = 0, so\ we\ can\ consider\ only\ when\ a \neq a'$$

$$\bar{A}(s) = Prob(a \neq a'|s) E_{(a,a') \sim (\pi_\theta,\ \pi_{\theta'})|a \neq a'}[A^{\pi_\theta}(s, a') - A^{\pi_\theta}(s, a)]$$

$$\bar{A}(s) \leq\ \alpha\, E_{(a,a') \sim (\pi_\theta,\ \pi_{\theta'})|a \neq a'}[A^{\pi_\theta}(s, a') - A^{\pi_\theta}(s, a)]$$

$$|\bar{A}(s)| \leq \alpha \cdot 2\ max_{s,a}(|A^{\pi_\theta}(s, a)|)$$

(7.14)

Now, if $(\pi_\theta\,,\ \pi_{\theta'})$ is an $\alpha - coupled\,policy\,pair$ also the trajectories τ and τ' (respectively generated by π_θ and $\pi_{\theta'}$, using the same random seed) can be coupled. We can compute the advantage of $\pi_{\theta'}$ over π_θ at each timestamp t and decompose this expectation depending on whether $\pi_{\theta'}$ agrees with π_θ at all timestamps $i < t$. We use n_t to count the occurrences in which $\pi_{\theta'}$ disagrees with π_θ (i.e. $a \neq a'$) before t . To simplify the notation, we use $s_t \sim \pi_\theta$ instead of the previous $s \sim P(\tau|\pi_\theta)$.

$$E_{s_t \sim \pi_{\theta'}}[\bar{A}(s_t)] = Prob(n_t = 0) E_{s_t \sim \pi_{\theta'}|n_t=0}[\bar{A}(s_t)] + Prob(n_t > 0) E_{s_t \sim \pi_{\theta'}|n_t>0}[\bar{A}(s_t)]$$

(7.15)

We can write the same for the trajectories generated by π_θ :

$$E_{s_t \sim \pi_\theta}[\bar{A}(s_t)] = Prob(n_t = 0)E_{s_t \sim \pi_\theta | n_t = 0}[\bar{A}(s_t)] + Prob(n_t > 0)E_{s_t \sim \pi_\theta | n_t > 0}[\bar{A}(s_t)]$$

(7.16)

Note that the terms with $n_t = 0$ are equals, because in there the two policies agree at all timestamps:

$$E_{s_t \sim \pi_{\theta'} | n_t = 0}[\bar{A}(s_t)] = E_{s_t \sim \pi_\theta | n_t = 0}[\bar{A}(s_t)]$$

$$\Longrightarrow$$

$$E_{s_t \sim \pi_{\theta'}}[\bar{A}(s_t)] - E_{s_t \sim \pi_\theta}[\bar{A}(s_t)] =$$

$$= Prob(n_t > 0)(E_{s_t \sim \pi_{\theta'} | n_t > 0}[\bar{A}(s_t)] - E_{s_t \sim \pi_\theta | n_t > 0}[\bar{A}(s_t)])$$

(7.17)

Now, by definition, $\pi_{\theta'}$ agrees with π_θ at each timestamp with probability $\geq 1 - \alpha$, hence we have that:

$$Prob(n_t = 0) \geq (1 - \alpha)^t$$

$$Prob(n_t > 0) \leq 1 - (1 - \alpha)^t$$

(7.18)

Consider that:

$$|E_{s_t \sim \pi_{\theta'} | n_t > 0}[\bar{A}(s_t)] - E_{s_t \sim \pi_\theta | n_t > 0}[\bar{A}(s_t)]| \leq |E_{s_t \sim \pi_{\theta'} | n_t > 0}[\bar{A}(s_t)]| + |E_{s_t \sim \pi_\theta | n_t > 0}[\bar{A}(s_t)]|$$

(7.19)

And, by eq 7.14:

$$|E_{s_t \sim \pi_{\theta'} | n_t > 0}[\bar{A}(s_t)]| + |E_{s_t \sim \pi_\theta | n_t > 0}[\bar{A}(s_t)]| \leq \ \alpha \cdot 4 \, max_{s,a}(|A^{\pi_\theta}(s, a)|)$$

(7.20)

Now if we compute the absolute value of eq. 7.17 and plug eq. 7.18 and 7.20 into it:

$$|E_{s_t \sim \pi_{\theta'} = 0}[\bar{A}(s_t)] - E_{s_t \sim \pi_\theta}[\bar{A}(s_t)]| = Prob(n_t > 0)|E_{s_t \sim \pi_{\theta'} | n_t > 0}[\bar{A}(s_t)] - E_{s_t \sim \pi_\theta | n_t > 0}[\bar{A}(s_t)]|$$

$$|E_{s_t \sim \pi_{\theta'}=0}[\bar{A}(s_t)] - E_{s_t \sim \pi_\theta}[\bar{A}(s_t)]| \leq (1-(1-\alpha)^t) \cdot \alpha \cdot 4\, max_{s,a}(|A^{\pi_\theta}(s,a)|)$$

(7.21)

This bounds the difference in expected advantage at each timestep t. If we sum over time, we bound the difference between $J(\theta')$ and $L_{\xi\theta}(\theta')$. Subtracting eq. 7.13 from eq. 7.12:

$$\left|J(\theta') - L_{\xi\theta}(\theta')\right| = \sum_t \gamma^t\, |E_{s_t \sim \pi_{\theta'}}[\bar{A}(s_t)] - E_{s_t \sim \pi_\theta}[\bar{A}(s_t)]|$$

(7.22)

Now, using $v = max_{s,a}|A^{\pi_\theta}(s,a)|$ as we already defined in eq. 7.9 with eq. 7.21 and eq. 7.22:

$$\left|J(\theta') - L_{\xi\theta}(\theta')\right| \leq \sum_t^{\infty} \gamma^t (1-(1-\alpha)^t) 4\, \alpha\, v$$

$$any\ series \sum_{k=0}^{\infty} \gamma^k\ with\ \gamma \in [0,1]\ converges\ to\ \frac{1}{1-\gamma} \Rightarrow$$

$$\sum_t^{\infty} \gamma^t (1-(1-\alpha)^t) 4\, \alpha\, v = 4\, \alpha\, v \left(\sum_t^{\infty} \gamma^t - \sum_t^{\infty} (\gamma(1-\alpha))^t \right) = 4\, \alpha\, v \left(\frac{1}{1-\gamma} - \frac{1}{1-\gamma(1-\alpha)} \right) \Rightarrow$$

$$\left|J(\theta') - L_{\xi\theta}(\theta')\right| \leq 4\, \alpha\, v \left(\frac{1}{1-\gamma} - \frac{1}{1-\gamma(1-\alpha)} \right)$$

$$\left|J(\theta') - L_{\xi\theta}(\theta')\right| \leq \frac{4\, \alpha^2\, \gamma\, v}{(1-\gamma)(1-\gamma(1-\alpha))}$$

$$\left|J(\theta') - L_{\xi\theta}(\theta')\right| \leq \frac{4\, \alpha^2\, \gamma\, v}{(1-\gamma)^2}$$

(7.23)

Now, the right-hand side of eq. 7.23 is ≥ 0 , that implies that:

$$if\ J(\theta') \geq L_{\xi\theta}(\theta') \Rightarrow J(\theta') - L_{\xi\theta}(\theta') \leq \frac{4\, \alpha^2\, \gamma\, v}{(1-\gamma)^2} \Rightarrow J(\theta') \leq L_{\xi\theta}(\theta') + \frac{4\, \alpha^2\, \gamma\, v}{(1-\gamma)^2}$$

(7.24)

$$if\ J(\theta') \leq L_{\xi\theta}(\theta') \implies L_{\xi\theta}(\theta') - J(\theta') \leq \frac{4\,\alpha^2\,\gamma\,\upsilon}{(1-\gamma)^2} \Rightarrow J(\theta') \geq L_{\xi\theta}(\theta') - \frac{4\,\alpha^2\,\gamma\,\upsilon}{(1-\gamma)^2}$$

(7.25)

If the premise of eq. 7.24 is true ($if\ J(\theta') \geq L_{\xi\theta}(\theta')$), then of course it is true also $J(\theta') \geq (L_{\xi\theta}(\theta') - anything\ positive)$, that means that it is true also $J(\theta') \geq L_{\xi\theta}(\theta') - \frac{4\,\alpha^2\,\gamma\,\upsilon}{(1-\gamma)^2}$ (the result of 7.25), because $\frac{4\,\alpha^2\,\gamma\,\upsilon}{(1-\gamma)^2}$ is always positive. So, in any case it holds:

$$J(\theta') \geq L_{\xi\theta}(\theta') - \frac{4\,\alpha^2\,\gamma\,\upsilon}{(1-\gamma)^2}$$

(7.26)

Now, following [Levin et al. 2009], proposition 2.7, we know that if we have two distributions $P(x)$ and $Q(y)$ whose Total Variation Distance is α (see Appendix A.2), formally $D_{TV}(P||Q) = \alpha$, then there exists a joint distribution (X, Y) whose marginals are $P(x)$ and $Q(y)$, for which the probability of $X = Y$ is equal to $1 - \alpha$.
Hence if we have two policies $\pi_{\theta'}$ and π_θ such that $max_s[D_{TV}(\pi_\theta(\cdot\,|s)||\pi_{\theta'}(\cdot\,|s))] \leq \alpha$ then it is possible to define an $\alpha - coupled\ policy\ pair\ \ (\pi_{\theta'}, \pi_\theta)$.
So if we name:

$$D_{TV}^{max} = max_s[D_{TV}(\pi_\theta(\cdot\,|s)||\pi_{\theta'}(\cdot\,|s))]$$

(7.27)

And if we choose $\alpha = D_{TV}^{max}$, we can rewrite eq. 7.26:

$$J(\theta') \geq L_{\xi\theta}(\theta') - \frac{4\ \gamma\,\upsilon}{(1-\gamma)^2}(D_{TV}^{max})^2$$

(7.28)

Now, it is known (see [Pollard 2000] Ch.3) that the Total Variation distance has the following relationship with Kullback-Leibler divergence: $D_{TV}(P||Q)^2 \leq D_{KL}(P||Q)$. So in eq. 7.28 we can plug in $D_{KL}^{max}(\pi_\theta, \pi_{\theta'})$:

$$J(\theta') \geq L_{\xi\theta}(\theta') - \frac{4\ \gamma\,\upsilon}{(1-\gamma)^2} D_{KL}^{max}(\pi_\theta, \pi_{\theta'})$$

(7.29)

And we see that eq. 7.29 is equivalent to eq. 7.10, so we just proved eq. 7.10.

■

So this proves that $L_{\xi\theta}(\theta') - C\ D_{KL}^{max}(\pi_\theta, \pi_{\theta'})$ is a lower bound of $J(\theta')$.

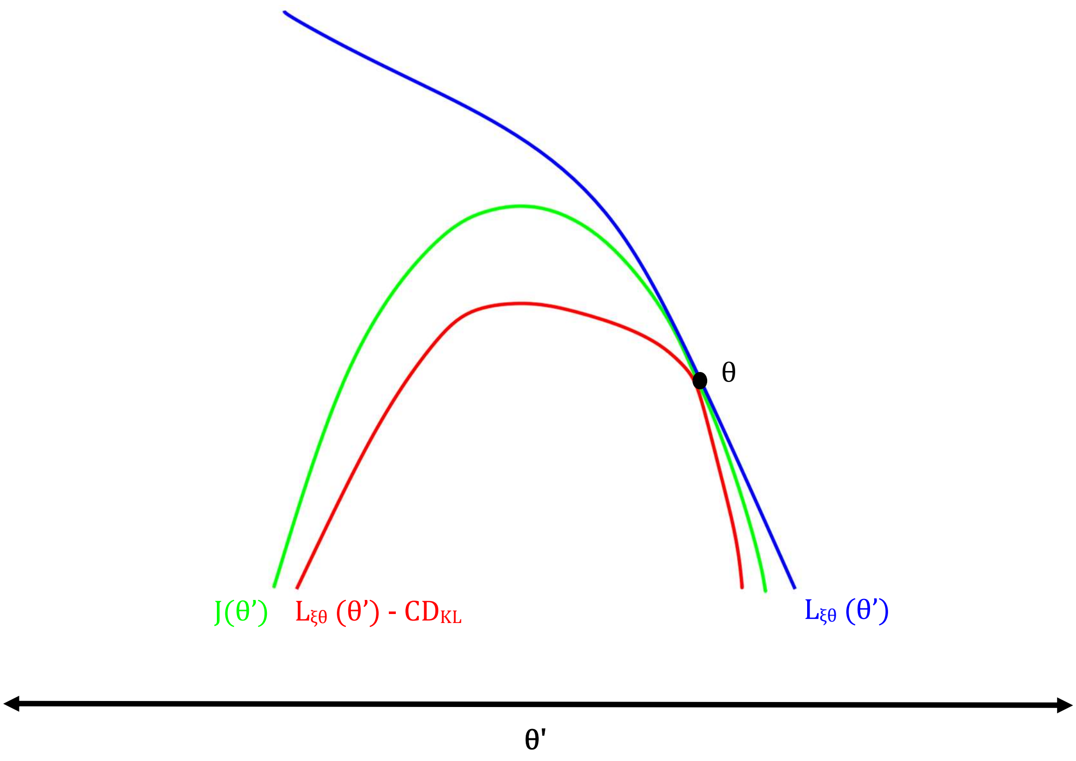

Figure 7.1

Graphic rendition of $L_{\xi\theta}(\theta') - C\ D_{KL}^{max}(\pi_\theta, \pi_{\theta'})$ as a lower bound for $J(\theta')$. When $\theta' = \theta$ the three curves coincide.

7.5 Monotonic Improvement

Now, looking back at eq. 7.10, if we call θ_i the policy parameters at time i , and we call ω the policy parameters for any other different policy (for instance a policy to be optimized), we define:

$$M_i(\omega) = L_{\xi\theta_i}(\omega) - C\ D_{KL}^{max}(\pi_{\theta_i}, \pi_\omega) \tag{7.30}$$

Then, if we start from eq. 7.10 , substituting θ' with ω and θ with θ_i , we have:

$$J(\omega) \geq L_{\xi\theta_i}(\omega) - C\ D_{KL}^{max}(\pi_{\theta_i}, \pi_\omega)$$

$$\Longrightarrow J(\omega) \geq M_i(\omega)$$

(7.31)

When $\omega = \theta_i$ it is easy to see that since the Kullbach-Leibler divergence of two identical distribution is zero, and the advantage function of two identical policies is zero (that makes $L_{\xi\theta_i}(\theta_i) = J(\theta_i)$) :

$$J(\theta_i) = M_i(\theta_i)$$

(7.32)

Now, let us say that at time $i + 1$ we have a policy parametrized by θ_{i+1} . This can be the case such as in a loop in which at each iteration we change the policy parameters. By eq. 7.31 we have:

$$J(\theta_{i+1}) \geq M_i(\theta_{i+1})$$

(7.33)

We can combine 7.32 with 7.33:

$$J(\theta_{i+1}) - J(\theta_i) \geq M_i(\theta_{i+1}) - M_i(\theta_i)$$

(7.34)

This means that improving M_i implies monotonically improving the expected return ! This is also called *“Monotonic improvement theorem”*.

7.6 Surrogate Objective

So, we know what we want to optimize: $L_{\xi\theta}(\theta') - C\ D_{KL}^{max}(\pi_\theta, \pi_{\theta'})$ with respect to θ'.

As [Schulman et al. 2017] notice, using that C penalty coefficient is theoretically justified but leads to very small steps.

Alternatively, we could be able to take larger steps if we just maximize $L_{\xi\theta}(\theta')$ and put as a condition that the difference between the new policy distribution and the old policy distribution stays under a certain threshold δ (the "trust region"), measuring that difference with a Kullback-Leibler divergence.

$$\underset{\theta'}{maximize}\ L_{\xi\theta}(\theta')$$

$$s.t. \quad D_{KL}^{max}(\pi_\theta, \pi_{\theta'}) \leq \beta$$

(7.35)

But we need also to express $L_{\xi\theta}(\theta')$ as an expectation with respect to a known policy, to be able to sample it: our agent must follow a policy with respect to which the expectation is expressed. If we look at eq. 7.7 we see that the first term of $L_{\xi\theta}(\theta')$ is $J(\theta)$, that does not depend on θ' so it is not getting optimized (we could actually get rid of it from the equation). The second term $E_{s\sim P(\tau|\pi_\theta)}\left[E_{a\sim\pi_{\theta'}(a|s)}[\gamma^t A^{\pi_\theta}(s,a)]\right]$ has the outer part that is an expectation with respect to policy π_θ (the old policy, that we know), and the inner part that is an expectation with respect to policy $\pi_{\theta'}$ that is an unknown. To be able to sample actions, we want to have all expectations in the formula being with respect to the old policy π_θ.

To achieve that, let us start from the formula of $L_{\xi\theta}(\theta')$ in eq. 7.5:

$$L_{\xi\theta}(\theta') = J(\theta) + \sum_s \xi_\theta(s) \sum_a \pi_{\theta'}(a_t|s_t)\, A^{\pi_\theta}(s_t, a_t)$$

Again, $J(\theta)$ is fixed because we optimize over the unknown parameters $\theta'$of the new policy, so we do not worry about $J(\theta)$ now. Let us focus on the second term.

If you recall from eq. 7.3 and eq. 7.4: $\xi_\theta(s) = \sum_t \gamma^t Prob(s_t = s \mid \theta)$.

Any series $\sum_{k=0}^{\infty} \gamma^k$ with $\gamma \in [0,1]$ converges to $1/(1-\gamma)$.

Hence, the sum over the frequencies $\sum_s \xi_\theta(s)$ can then be replaced by the surrogate expectation $\frac{1}{1-\gamma} E_{s\sim\xi_\theta(s)}$.

Since the formula will be used for optimization, we could even get rid of the constant $\frac{1}{1-\gamma}$ when optimizing with respect to θ' (but for now we will keep it).

The sum over actions can be replaced by an expectation over actions, and we want those actions to be generated by the old policy θ , while now they are multiplied by the probabilities of the new policy θ', so we need to apply importance sampling. So, the new objective to be maximized becomes:

$$L_{\xi\theta}(\theta') = J(\theta) + \frac{1}{1-\gamma} E_{s\sim\xi_\theta(s)}\left[E_{a\sim\pi_\theta(a|s)}\left[\frac{\pi_{\theta'}(a|s)}{\pi_\theta(a|s)} A^{\pi_\theta}(s,a)\right]\right]$$

(7.36)

Now, if we look at equation 7.36 we note that the expectation with respect to the states $E_{s\sim\xi_\theta(s)}$ is with respect to the discounted frequency of the states $\xi_\theta(s)$ (e.q. 7.3), that even when normalized through a multiplication by $1/(1-\gamma)$ is not generally exactly the same as the state distribution that happens during training, which is determined by the transition function $P(s'|s,a)$ and the policy π_θ, or equivalently by the trajectory distribution $P(\tau|\pi_\theta)$ (eq. 1.4). This is because of the presence of γ^t term in $\xi_\theta(s)$: the expectation $\frac{1}{1-\gamma} E_{s\sim\xi_\theta(s)}$ is the same of $E_{s\sim P(\tau|\pi_\theta)}$ only in case of some non-general assumptions, such as when the visiting probability of each state never changes, being the same at each time t. Or, in a more relaxed assumption, if the state distribution is ergodic they converge for $t \longrightarrow \infty$.

But when we sample our trajectories we actually sample from $P(\tau|\pi_\theta)$.

So, we need to write a slightly different surrogate objective, using the expectation that is consistent with our sampling, obtaining the following function:

$$L_\theta(\theta') = J(\theta) + E_{s\sim P(\tau|\pi_\theta)}\left[E_{a\sim\pi_\theta(a|s)}\left[\frac{\pi_{\theta'}(a|s)}{\pi_\theta(a|s)} A^{\pi_\theta}(s,a)\right]\right]$$

$$= J(\theta) + E_{\tau\sim\pi_\theta}\left[\frac{\pi_{\theta'}(a|s)}{\pi_\theta(a|s)} A^{\pi_\theta}(s,a)\right]$$

(7.37)

Fortunately, it turns out that we can use that as objective. This is justified by the fact that the gradient of equation 7.37 matches the gradient of $J(\theta')$ if evaluated locally at the point $\theta' = \theta$:

$$\nabla_{\theta'} L_\theta(\theta')\big|_{\theta'=\theta} = E_{\tau\sim\pi_\theta}\left[\frac{\nabla_{\theta'}\pi_{\theta'}(a|s)}{\pi_\theta(a|s)} A^{\pi_\theta}(s,a)\right]\Bigg|_{\theta'=\theta}$$

$$= E_{\tau\sim\pi_\theta}[\log\nabla_{\theta'}\pi_{\theta'}(a|s)\, A^{\pi_\theta}(s,a)]\big|_{\theta'=\theta}$$

$$= E_{\tau\sim\pi_{\theta'}}[\log\nabla_{\theta'}\pi_{\theta'}(a|s)\, A^{\pi_\theta}(s,a)]\big|_{\theta'=\theta}$$

$$= \nabla_{\theta'} J(\theta')\big|_{\theta' = \theta}$$

(7.38)

So this means that a small change in the policy parameters from θ to θ' such that $L_\theta(\theta')$ increases, makes also the new policy's expected return $J(\theta')$ increase with respect to the old policy expected return $J(\theta)$. This "small change" in policy parameters is expressed as the constraint on the Kullback-Leibler divergence, because the two policy distribution must be close.

So what we want to optimize is:

$$\underset{\theta'}{maximize}\ L_\theta(\theta') = J(\theta) + E_{\tau \sim \pi_\theta}\left[\frac{\pi_{\theta'}(a|s)}{\pi_\theta(a|s)} A^{\pi_\theta}(s,a)\right]$$

$$s.t. \quad D_{KL}^{max}(\pi_\theta, \pi_{\theta'}) \leq \beta$$

(7.39)

This constraint on the KL divergence applies to all points in space and it is not practical to be concretely applied, so an heuristic approximation may be used instead, computing the average Kullback-Leibler divergence (which at computation time will be the average of the samples).

$$\underset{\theta'}{maximize}\ L_\theta(\theta') = J(\theta) + E_{\tau \sim \pi_\theta}\left[\frac{\pi_{\theta'}(a|s)}{\pi_\theta(a|s)} A^{\pi_\theta}(s,a)\right]$$

$$s.t. \quad E_{\tau \sim \pi_\theta}[D_{KL}(\pi_\theta(\cdot\,|s)||\pi_{\theta'}(\cdot\,|s))] \leq \beta$$

(7.40)

When maximizing eq. 7.40 with respect to θ' it is easy to see that fixed term $J(\theta)$ is not dependent on θ' so we can get rid of it:

$$\underset{\theta'}{maximize}\ L_\theta(\theta') = E_{\tau \sim \pi_\theta}\left[\frac{\pi_{\theta'}(a|s)}{\pi_\theta(a|s)} A^{\pi_\theta}(s,a)\right]$$

$$s.t. \quad E_{\tau \sim \pi_\theta}[D_{KL}(\pi_\theta(\cdot\,|s)||\pi_{\theta'}(\cdot\,|s))] \leq \beta$$

(7.41)

At this point the optimization problem is completely defined. We need only an effective way to solve it. The original authors [Schulman et al. 2015] used $Q^{\pi_\theta}(s,a)$ instead of $A^{\pi_\theta}(s,a)$, which leads to an equivalent optimization problem because it changes the objective only by a constant, but saves from the burden of computing $V^{\pi_\theta}(s)$ to compute $A^{\pi_\theta}(s,a)$.

$$\underset{\theta'}{maximize}\ L_\theta(\theta') = E_{\tau\sim\pi_\theta}\left[\frac{\pi_{\theta'}(a|s)}{\pi_\theta(a|s)} Q^{\pi_\theta}(s,a)\right]$$

$$s.t. \quad E_{\tau\sim\pi_\theta}[D_{KL}(\pi_\theta(\cdot\,|s)||\pi_{\theta'}(\cdot\,|s))] \le \beta \tag{7.42}$$

$Q^{\pi_\theta}(s,a)$ is computed by an empirical estimate, such as the sum of discounted rewards of the sampled trajectory that starts with (s,a), that is actually $G(\tau)$ of eq. 1.3 . In our pseudocode algorithm 7.1 at the end of the chapter we anyway use the advantage function instead of the Q function, because it converges faster, so we use eq. 7.41 instead of eq. 7.42 (they both are theoretically right and should converge to the same policy).

The parameters θ are then updated with the θ' found solving the constrained optimization problem, using the conjugate gradient algorithm followed by line search (point 3 of paragraph 6, and Appendix C of [Schulman et al. 2015]): we are going to explain that in detail in the following part.

7.7 Constraining the Kullback-Leibler Divergence

Now, in order to simplify the procedure to analytically find the optimization update rule satisfying the Kullback-Leibler constraint, we approximate the objective $L_\theta(\theta')$ by its first order Taylor expansion $\widehat{L_\theta}(\theta')$. Note that we do this only to find a formula that considers the Kullback-Leibler constraint, but then at computation time we will use the real $L_\theta(\theta')$ gradient, and not the one simplified using its first order Taylor expansion $\widehat{L_\theta}(\theta')$. Since in $L_\theta(\theta')$ the parameters θ are fixed and the variable parameters are θ', and the Taylor expansion is built around the point $\theta' = \theta$, we have that:

$$\widehat{L_\theta}(\theta') = L_\theta(\theta) + \nabla_{\theta'} L_\theta(\theta')^T\big|_{\theta'=\theta}(\theta' - \theta)$$

$$since\ L_\theta(\theta) = 0 \implies$$

$$\widehat{L_\theta}(\theta') = \nabla_{\theta'} L_\theta(\theta')^T \big|_{\theta' = \theta}(\theta' - \theta)$$

$$we\ plug\ eq. 7.38\ in$$

$$\widehat{L_\theta}(\theta') = \nabla_{\theta'} J(\theta')^T \big|_{\theta' = \theta}(\theta' - \theta)$$

(7.43)

The Kullback-Leibler divergence is formulated by the approximation described in Appendix A.6 (eq. a.25).

$$\widehat{D_{KL}}(\pi(\cdot\ |s_t;\theta)||\pi(\cdot\ |s_t;\theta + \delta)) = \frac{1}{2}\delta^{\mathrm{T}}\widehat{\boldsymbol{F_{\pi\theta}}}\delta$$

(7.44)

What we are doing until now, that is maximizing eq. 7.43 while keeping the KL divergence constrained, seems very similar to the Natural Policy Gradient (Ch. 6). In fact at this point we are maximizing:

$$\nabla_{\theta'} J(\theta')^T \big|_{\theta' = \theta}(\theta' - \theta)$$

$$s.t.\ \frac{1}{2}\delta^{\mathrm{T}}\widehat{\boldsymbol{F_{\pi\theta}}}\delta \le \beta$$

(7.45)

And it is easy to see that eq. 7.45 is equivalent to eq. 6.8 of Natural Policy Gradient ! Solving 7.45 with the method of Lagrangian multipliers would be the same as we did in Ch. 6 and will give the same solution as eq. 6.7 and eq. 6.13, with the following parameters update rule:

$$\theta' \leftarrow \theta + \sqrt{\frac{2\beta}{\nabla_\theta J(\pi_\theta)^T \boldsymbol{F_{\pi_\theta}}^{-1} \nabla_\theta J(\pi_\theta)}}\boldsymbol{F_{\pi\theta}^{-1}}\nabla_\theta J(\pi_\theta)$$

(7.46)

But TRPO differs in the way the computation is executed. To begin with, at computation time in TRPO instead of $\nabla_\theta J(\pi_\theta)$ (which was obtained from the first order Taylor approximation) we use $\nabla_{\theta'}\left(\frac{\pi_{\theta'}(a|s)}{\pi_\theta(a|s)}A^{\pi_\theta}(s,a)\right)$ that is the gradient of the surrogate objective (see eq. 7.41).

Secondly, the Fisher Information Matrix is calculated in a different way compared to Natural Policy Gradient, but we will detail it in next paragraph. In addition, there are two problematic aspects: the former is that big matrices like $\boldsymbol{F}_{\boldsymbol{\pi}_{\boldsymbol{\theta}}}$ are costly to invert, and the latter is that the approximations used may cause a destructive update of the policy. Let us see how TRPO tackles those problems.

7.8 Inverting the Fisher Information Matrix via Conjugate Gradient

To compute the update rule of eq. 7.46 we need to find the inverse of the Fisher information Matrix $\boldsymbol{F}_{\boldsymbol{\pi}_{\boldsymbol{\theta}}}$. Computing the inverse of a big matrix can be very expensive in terms of computation and memory, but if the matrix is symmetrical and positive definite it is possible to do it in a way that is faster and less memory demanding than with traditional linear algebra methods (especially when the matrix is sparse): with the Conjugate Gradient algorithm [Hestenes and Stiefel 1952]. It turns out that the Fisher Information Matrix is symmetrical, and it is positive semidefinite: even when a matrix is not guaranteed to be definite but only semidefinite the Conjugate Gradient algorithm is able to converge [Hayami 2018]. The Conjugate Gradients can solve a linear system:

$$\mathbf{A}x = b \tag{7.47}$$

(with $\mathbf{A}$ and b given) exploiting the fact that, if $\mathbf{A}$ is symmetrical and positive definite, the solution of such a system (that is $x = \mathbf{A}^{-1}b$) can be found solving the minimization problem of the equation:

$$f(x) = \frac{1}{2}x^T\mathbf{A}\,x - x^T b \tag{7.48}$$

The Conjugate Gradient allows us to do it fast because it computes a particular version of steepest descent in which the direction of descent is chosen efficiently. A sketch of the Gradient Descent algorithm is reported in Appendix B.

Now, if in place of matrix $\mathbf{A}$ we use the approximation of Fisher Information Matrix

$\widehat{\boldsymbol{F_{\pi\theta}}}$, and if in place of b we use $\nabla_\theta J(\pi_\theta)$, we obtain that the solution found by the Conjugate Gradient algorithm is (substituting in $x = \mathbf{A}^{-1}b$):

$$x = \widehat{\boldsymbol{F_{\pi\theta}}}^{-1} \nabla_\theta J(\pi_\theta)$$

(7.49)

If we watch eq. 7.46 we notice that we do not need to find also $\widehat{\boldsymbol{F_{\pi\theta}}}^{-1}$ alone: we can just use our x, that is $\widehat{\boldsymbol{F_{\pi\theta}}}^{-1} \nabla_\theta J(\pi_\theta)$. In fact:

$$\sqrt{\frac{2\beta}{\nabla_\theta J(\pi_\theta)^T \boldsymbol{F_{\pi_\theta}}^{-1} \nabla_\theta J(\pi_\theta)}}\, \boldsymbol{F_{\pi\theta}^{-1}} \nabla_\theta J(\pi_\theta) = \sqrt{\frac{2\beta}{x^T \widehat{\boldsymbol{F_{\pi\theta}}}\, x}}\; x$$

(7.50)

Proof:

$$\sqrt{\frac{2\beta}{x^T \widehat{\boldsymbol{F_{\pi\theta}}}\, x}}\; x =$$

$$\sqrt{\frac{2\beta}{\left(\widehat{\boldsymbol{F_{\pi\theta}}}^{-1} \nabla_\theta J(\pi_\theta)\right)^T \widehat{\boldsymbol{F_{\pi\theta}}}\, \widehat{\boldsymbol{F_{\pi\theta}}}^{-1} \nabla_\theta J(\pi_\theta)}}\; \widehat{\boldsymbol{F_{\pi\theta}}}^{-1} \nabla_\theta J(\pi_\theta) =$$

$$\sqrt{\frac{2\beta}{\nabla_\theta J(\pi_\theta)^T (\widehat{\boldsymbol{F_{\pi\theta}}}^{-1})^T \widehat{\boldsymbol{F_{\pi\theta}}}\, \widehat{\boldsymbol{F_{\pi\theta}}}^{-1} \nabla_\theta J(\pi_\theta)}}\; \widehat{\boldsymbol{F_{\pi\theta}}}^{-1} \nabla_\theta J(\pi_\theta) =$$

$$since\ the\ Fisher\ Information\ Matrix\ is\ symmetrical, its\ inverse\ is\ symmetrical \Longrightarrow$$

$$= \sqrt{\frac{2\beta}{\nabla_\theta J(\pi_\theta)^T\, \widehat{\boldsymbol{F_{\pi\theta}}}^{-1} \nabla_\theta J(\pi_\theta)}}\; \widehat{\boldsymbol{F_{\pi\theta}}}^{-1} \nabla_\theta J(\pi_\theta)$$

(7.51)

This means that we do not need to compute $\widehat{\boldsymbol{F_{\pi\theta}}}^{-1}$, we can be happy enough if we find $\widehat{\boldsymbol{F_{\pi\theta}}}^{-1} \nabla_\theta J(\pi_\theta)$, and this is something that the Conjugate Gradient algorithm can do for us.

To use the Conjugate Gradient algorithm all we need is a function that takes vector v as input, and returns the multiplication $\mathbf{A}v$, where in our case $\mathbf{A}$ would be our approximated Fisher Information Matrix $\widehat{\boldsymbol{F_{\pi\theta}}}$.

Now, this function that returns the multiplication $\widehat{\boldsymbol{F_{\pi\theta}}} \cdot v$ can be realized in at least two ways: one is described by original authors [Schulman et al. 2015] and consists in finding a manageable version of an approximation of the Fisher Information Matrix. We detail it in the next section. The other method is suggested by [OpenAi 2018D] and does not require to compute an approximation of the Fisher Information Matrix. It is a faster method, so it is recommended, and we detail it in section 7.8.2

7.8.1 Product of Average Hessian of KL

In this section we will detail the method of computing the approximated Fisher Information Matrix as described in the original paper [appendix C1 of Schulman et al. 2015]. This section is optional and can be skipped, since there is a faster way to compute the product between Fisher Information Matrix and a vector v without using directly the Fisher Information Matrix, and it is described in next section (section 7.8.2 “Average Gradient of Product by KL Gradient”).

As explained in our Appendix A.7, given two distributions of the same family $\pi_\theta(x)$ and $\pi_{\theta'}(x)$, the Hessian of Kullback-Leibler divergence of $\pi_\theta(x)$ from $\pi_{\theta'}(x)$, with respect to θ' , evaluated at $\theta' = \theta$ is equal to the Fisher Information Matrix of $\pi_\theta(x)$. Hence this can be used to approximate the Fisher Information Matrix: using the Hessian of Kullback-Leibler divergence instead of commonly using $E_{x\sim P_\theta(x)}[\nabla_\theta \log P_\theta(x)\ \nabla_\theta \log P_\theta(x)^{\mathrm{T}}]$ (as in Appendix A.4).

$$\widehat{\boldsymbol{F_{P_\theta}}}[i,j] = \frac{1}{N}\sum_{t}^{N} \frac{\partial^2}{\partial\theta'_i\theta'_j} D_{KL}(\pi_\theta(\cdot\,|s_t)||\pi_{\theta'}(\cdot\,|s_t)) \tag{7.52}$$

To compute the Hessian of Kullback-Leibler divergence of $\pi_\theta(a,s)$ from $\pi_{\theta'}(x)$, for each sampled state s we need to take in consideration all possible actions a , not only the one that has been actually taken by the sampled trajectory. An analytic estimator must integrate over a for each sampled state s.

Let us detail an analytical way to compute the Fisher Information Matrix using the Hessian of Kullback-Leibler divergence. The KL divergence of old policy $\pi_{\theta}(\cdot\,|s)$ from new policy $\pi_{\theta'}(\cdot\,|s)$ is $D_{KL}(\pi_{\theta}(\cdot\,|s)||\pi_{\theta'}(\cdot\,|s))$. The new policy is unknown, it is what we aim to find with the TRPO algorithm. Each policy function outputs a vector of real numbers, that may be the mean of a distribution (e.g. a normal) for each action parameter when the actions are continuous, while when the actions are discrete the policy function outputs a vector of probability for each action (for each action dimension). With continuous actions if you do not have fixed variances you will output the variance values too with the policy function.

So let us call the outputs of the old and new policy functions respectively $\mu_{OLD}(s)$ and $\mu_{\theta'}(s)$. You may notice that for the old policy we do not put the parameters in the notation, we just put "old", to explicit the fact that the old policy is unaffected by θ' parameters optimization because it depends on fixed parameters (that were indicated with θ in eq. 7.41 and 7.42).

We denote the Hessian of KL divergence $\pi_{\theta}(\cdot\,|s)$ from $\pi_{\theta'}(\cdot\,|s)$ in a simplified way, using the outputs of the policy functions, so to make evident what are the variables of the KL divergence function:

$$D_{KL}(\pi_{\theta}(\cdot\,|s)||\pi_{\theta'}(\cdot\,|s)) = kl(\,\mu_{OLD}(s),\,\mu_{\theta'}(s)\,) \tag{7.53}$$

Now, to compute the Hessian of $kl(\,\mu_{OLD}(s),\,\mu_{\theta'}(s)\,)$ we first compute the gradient with respect to θ'. Since $kl()$ is a function whose parameters are functions themselves, we apply the chain rule of calculus:

$$\frac{\partial(kl(\,\mu_{OLD}(s),\,\mu_{\theta'}(s)\,))}{\partial\theta'} =$$

$$\frac{\partial(kl(\,\mu_{OLD}(s),\,\mu_{\theta'}(s)\,))}{\partial(\mu_{OLD}(s))}\frac{\partial(\mu_{OLD}(s))}{\partial\theta'} + \frac{\partial(kl(\,\mu_{OLD}(s),\,\mu_{\theta'}(s)\,))}{\partial(\mu_{\theta'}(s))}\frac{\partial(\mu_{\theta'}(s))}{\partial\theta'} =$$

$$since\ \frac{\partial(\mu_{OLD}(s))}{\partial\theta'}\ equals\ to\ 0\ because\ \mu_{OLD}(s)\ does\ not\ depend\ on\ \theta', the\ first\ term\ disappears$$

$$= \frac{\partial(kl(\,\mu_{OLD}(s),\,\mu_{\theta'}(s)\,))}{\partial(\mu_{\theta'}(s))}\frac{\partial(\mu_{\theta'}(s))}{\partial\theta'} \tag{7.54}$$

This may be rewritten in a clearer way making evident that $\mu_{\theta'}(s)$ may be a vector function, and θ' is (usually) a vector, using subscripts to enumerate the element of the vector, with $\mu_{\theta'}(s)$ having U elements, and θ' having V elements. Since we are applying the chain rule of calculus, the partial derivative with respect to an element of θ' must sum the derivatives of every function parameter with respect to it, that is why on every row we have a summation. To ease the reading, we write μ_{OLD} instead of $\mu_{OLD}(s)$.

$$\frac{\partial(kl(\,\mu_{OLD},\,\mu_{\theta'}(s)\,))}{\partial\theta'}=$$

$$\begin{bmatrix} \frac{\partial kl\left(\mu_{OLD},\mu_{\theta'}(s)\right)}{\partial(\mu_{\theta'}(s)_1)}\frac{\partial(\mu_{\theta'}(s)_1)}{\partial\theta'_1}+\frac{\partial kl\left(\mu_{OLD},\mu_{\theta'}(s)\right)}{\partial(\mu_{\theta'}(s)_2)}\frac{\partial(\mu_{\theta'}(s)_2)}{\partial\theta'_1}+\cdots+\frac{\partial kl\left(\mu_{OLD},\mu_{\theta'}(s)\right)}{\partial(\mu_{\theta'}(s)_U)}\frac{\partial(\mu_{\theta'}(s)_U)}{\partial\theta'_1} \\ \frac{\partial kl\left(\mu_{OLD},\mu_{\theta'}(s)\right)}{\partial(\mu_{\theta'}(s)_1)}\frac{\partial(\mu_{\theta'}(s)_1)}{\partial\theta'_2}+\frac{\partial kl\left(\mu_{OLD},\mu_{\theta'}(s)\right)}{\partial(\mu_{\theta'}(s)_2)}\frac{\partial(\mu_{\theta'}(s)_2)}{\partial\theta'_2}+\cdots+\frac{\partial kl\left(\mu_{OLD},\mu_{\theta'}(s)\right)}{\partial(\mu_{\theta'}(s)_U)}\frac{\partial(\mu_{\theta'}(s)_U)}{\partial\theta'_2} \\ \cdots \\ \frac{\partial kl\left(\mu_{OLD},\mu_{\theta'}(s)\right)}{\partial(\mu_{\theta'}(s)_1)}\frac{\partial(\mu_{\theta'}(s)_1)}{\partial\theta'_V}+\frac{\partial kl\left(\mu_{OLD},\mu_{\theta'}(s)\right)}{\partial(\mu_{\theta'}(s)_2)}\frac{\partial(\mu_{\theta'}(s)_2)}{\partial\theta'_V}+\cdots+\frac{\partial kl\left(\mu_{OLD},\mu_{\theta'}(s)\right)}{\partial(\mu_{\theta'}(s)_U)}\frac{\partial(\mu_{\theta'}(s)_U)}{\partial\theta'_V} \end{bmatrix}$$

(7.55)

This may be compactly rewritten in a way that is vector-element wise, with row index i , as:

$$\left[\frac{\partial kl\left(\mu_{OLD},\mu_{\theta'}(s)\right)}{\partial\theta'_i}\right]=$$

$$\left[\frac{\partial kl\left(\mu_{OLD},\mu_{\theta'}(s)\right)}{\partial(\mu_{\theta'}(s)_a)}\frac{\partial(\mu_{\theta'}(s)_a)}{\partial\theta'_i}\right]$$

(7.56)

Where i stands for the element of the parameters vector θ', and a stands for "compute this equation over all elements of $\mu_{\theta'}(s)$ using a as index, and sum all the results together", as it is evident in eq. 7.55.

If we acknowledge that $\mu_{\theta'}(s)$ is a vector function and θ' is a vector we could also write it as the product of the Jacobian of $\mu_{\theta'}(s)$ wrt to θ' for the gradient of $kl(\,\mu_{OLD}(s),\,\mu_{\theta'}(s)$ wrt to $\mu_{\theta'}(s)$:

$$= \left(\frac{\partial(\mu_{\theta'}(s))}{\partial\theta'}\right)^T \frac{\partial(kl(\ \mu_{OLD}(s),\ \mu_{\theta'}(s)\ ))}{\partial(\mu_{\theta'}(s))}$$

$$= \mathbf{J}(\mu_{\theta'}(s))^T \frac{\partial(kl(\ \mu_{OLD}(s),\ \mu_{\theta'}(s)\ ))}{\partial(\mu_{\theta'}(s))}$$

(7.57)

Eq. 7.57 is like eq. 7.54 but with the Jacobian of $\mu_{\theta'}(s)$ and the gradient of $kl(\ \mu_{OLD}(s),\ \mu_{\theta'}(s))$ in switched order, to make it possible to have the matrix multiplication. Also we used $\mathbf{J}()$ to denote a Jacobian.

Now if we compute the Jacobian of 7.56 (or equivalently 7.57 or 7.55 or 7.54) wrt θ' we have the Hessian of $kl\big(\ \mu_{OLD}(s),\ \mu_{\theta'}(s)\big)$ wrt θ', here presented for any element of row i and column j.

$$\left[H\left(kl\big(\mu_{OLD},\mu_{\theta'}(s)\big)\right)_{i,j}\right] =$$

$$\left[\frac{\partial(\partial kl\big(\mu_{OLD},\mu_{\theta'}(s)\big))}{\partial\theta'_j\partial\theta'_i}\right] =$$

$$\left[\frac{\partial}{\partial\theta'_j}\left(\frac{\partial kl\big(\mu_{OLD},\mu_{\theta'}(s)\big)}{\partial(\mu_{\theta'}(s)_a)}\frac{\partial(\mu_{\theta'}(s)_a)}{\partial\theta'_i}\right)\right] =$$

$$\left[\frac{\partial}{\partial\theta'_j}\left(\frac{\partial kl\big(\mu_{OLD},\mu_{\theta'}(s)\big)}{\partial(\mu_{\theta'}(s)_a)}\right)\frac{\partial(\mu_{\theta'}(s)_a)}{\partial\theta'_i} + \frac{\partial kl\big(\mu_{OLD},\mu_{\theta'}(s)\big)}{\partial(\mu_{\theta'}(s)_a)}\frac{\partial}{\partial\theta'_j}\left(\frac{\partial(\mu_{\theta'}(s)_a)}{\partial\theta'_i}\right)\right] =$$

$$\left[\frac{\partial^2 kl\big(\mu_{OLD},\mu_{\theta'}(s)\big)}{\partial(\mu_{\theta'}(s)_b)\partial(\mu_{\theta'}(s)_a)}\frac{\partial(\mu_{\theta'}(s)_b)}{\partial\theta'_j}\frac{\partial(\mu_{\theta'}(s)_a)}{\partial\theta'_i} + \frac{\partial kl\big(\mu_{OLD},\mu_{\theta'}(s)\big)}{\partial(\mu_{\theta'}(s)_a)}\frac{\partial^2(\mu_{\theta'}(s)_a)}{\partial\theta'_j\partial\theta'_i}\right] =$$

$$\textit{the second term vanishes so only the first term remains}$$

$$= \left[\frac{\partial^2 kl\big(\mu_{OLD},\mu_{\theta'}(s)\big)}{\partial(\mu_{\theta'}(s)_b)\partial(\mu_{\theta'}(s)_a)}\frac{\partial(\mu_{\theta'}(s)_b)}{\partial\theta'_j}\frac{\partial(\mu_{\theta'}(s)_a)}{\partial\theta'_i}\right]$$

(7.58)

In the equation above i and j are the indices for each element of the parameters vector θ', and so identify the row and column of the element in the Hessian. Also, both a and b stand for "compute this equation over all elements of $\mu_{\theta'}(s)$ using a and b as indices and sum all the results together", that since they both appear at the same time is to be interpreted as: "to compute an element of the Hessian at position (i,j) , compute eq. 7.57 for every combination of a and b, and sum together all the results (considering a and b as two separate indices of the vector $\mu_{\theta'}(s)$)".
Writing it in a Matrix-wise way we have:

$$H\left(kl\big(\mu_{OLD},\mu_{\theta'}(s)\big)\right)_{wrt\ \theta'} =$$

$$\left(\frac{\partial(\mu_{\theta'}(s))}{\partial\theta'}\right)^T \frac{\partial^2(kl(\ \mu_{OLD}(s),\ \mu_{\theta'}(s)\ ))}{\partial\big(\mu_{\theta'}(s)_j\big)\partial(\mu_{\theta'}(s)_i)} \frac{\partial(\mu_{\theta'}(s))}{\partial\theta'} =$$

$$\mathbf{J}(\mu_{\theta'}(s))^{\boldsymbol{T}} \frac{\partial^2(kl(\ \mu_{OLD}(s),\ \mu_{\theta'}(s)\ ))}{\partial\big(\mu_{\theta'}(s)_j\big)\partial(\mu_{\theta'}(s)_i)} \mathbf{J}(\mu_{\theta'}(s)) \tag{7.59}$$

A shorter way to write it is:

$$H\left(kl\big(\mu_{OLD},\mu_{\theta'}(s)\big)\right)_{wrt\ \theta'} = \mathbf{J}^{\boldsymbol{T}}\mathbf{M}\,\mathbf{J}$$

$$with\ \mathbf{J}\ Jacobian\ of\ \mu_{\theta'}(s)\ wrt\ \theta'\ and\ \mathbf{M}\ hessian\ of\ kl\big(\ \mu_{OLD}(s),\ \mu_{\theta'}(s)\big)\ wrt\ \ \mu_{\theta'}(s) \tag{7.60}$$

This is a much more compact representation than the full computed hessian of $kl\big(\ \mu_{OLD}(s),\ \mu_{\theta'}(s)\big)$ wrt θ'.

If you recall, we found this manageable approximation of Fisher Information Matrix with the intention of using it within the Conjugate Gradient algorithm: for that algorithm we need to have a function that multiplies the Hessian/Fisher Information Matrix for a vector v, and since we have built $\widehat{\boldsymbol{F_{\pi\theta}}}$ as the average of $\mathbf{J}^{\boldsymbol{T}}\mathbf{M}\,\mathbf{J}$ over all states of the transitions (i.e. we plugged 7.60 into 7.52), we can just do the matrix-vector multiplication.

$$\widehat{F_{\pi\theta}} \cdot v = \left(\frac{1}{N} \sum_{t}^{N} \left(\mathbf{J}(s_t)\right)^{T} \mathbf{M}(s_t)\, \mathbf{J}(s_t) \right) \cdot v$$

(7.61)

Since this multiplication $\widehat{F_{\pi\theta}} \cdot v$ is used in the conjugate gradient algorithm to compute the direction of a descent vector for an iteration, it can even be an approximate computation, and original authors [Schulman et al. 2015] suggest to compute the approximate $\widehat{F_{P_\theta}}$ using only 10% of samples, in order to have a faster execution.

Anyway, there seems to be a faster way to obtain the Fisher Information Matrix product, and we see it in next section.

7.8.2 Average Gradient of Product by KL Gradient

A faster way of computing the product between Fisher Information Matrix and a vector v, is explained in [OpenAi 2018D], and it does not directly computes the Fisher Information Matrix. It consists in computing the average of the gradient of the Kullback-Leibler divergence for all states, transpose it, multiply it by the vector v (it would be a dot product between vectors), and compute the gradient of that result wrt to θ. Expressed in formulas:

$$\widehat{F_{\pi\theta}} \cdot v = \frac{1}{N} \sum_{t}^{N} \nabla_{\theta'} \left(\left(\frac{\partial (kl(\,\mu_{OLD},\, \mu_{\theta'}(s_t)\,))}{\partial \theta'} \right)^{T} \cdot v \right)$$

$$= \nabla_{\theta'} \left(\left(\frac{1}{N} \sum_{t}^{N} \left(\frac{\partial (kl(\,\mu_{OLD},\, \mu_{\theta'}(s_t)\,))}{\partial \theta'} \right)^{T} \right) \cdot v \right)$$

(7.62)

Proof (considering the vector v given, not depending on θ'):

$$\nabla_{\theta'} \left(\left(\frac{\partial (kl(\,\mu_{OLD},\, \mu_{\theta'}(s)\,))}{\partial \theta'} \right)^{T} \cdot v \right) =$$

$$\nabla_{\theta'} \left(\left(\frac{\partial (kl(\,\mu_{OLD},\, \mu_{\theta'}(s)\,))}{\partial \theta'} \right)^{T} \right) \cdot v \; + \; \left(\frac{\partial (kl(\,\mu_{OLD},\, \mu_{\theta'}(s)\,))}{\partial \theta'} \right)^{T} \nabla_{\theta'}(v) =$$

$$since \ \nabla_{\theta'}(v) \ is \ zero \ \Longrightarrow$$

$$= \nabla_{\theta'}\left(\left(\frac{\partial(kl(\mu_{OLD}, \mu_{\theta'}(s)))}{\partial\theta'}\right)^T\right) \cdot v$$

$$= H\left(kl(\mu_{OLD}, \mu_{\theta'}(s))\right) \cdot v$$

(7.63)

Where H denotes the Hessian.

To use this procedure for the matrix-vector multiplication, when writing the code with an automatic differentiation package it is necessary to have the values of v disconnected from the parameters θ' in the differentiation graph, so that the gradient of v wrt θ' is zero.

7.10 Using the Conjugate Gradient

Now that we have a function to do the product between the Fisher Information Matrix and a vector, we can run the Conjugate Gradient method and obtain the vector x (see eq. 7.49), that is a vector that gives the direction in which we should be going from the old parameters θ to reach the new parameters θ'. In other words we should multiply the vector x by a constant η and sum the result to old parameters θ to obtain new parameters θ' . According to eq. 7.46 and 7.50, η should be equal to $\sqrt{\frac{2\beta}{x^T \widehat{\boldsymbol{F}_{\pi\theta}}\, x}}$.

At this point it seems that we have everything we need for the update rule. But because of the approximations, it is not sure that with $\eta = \sqrt{\frac{2\beta}{x^T \widehat{\boldsymbol{F}_{\pi\theta}}\, x}}$ the constraint on the average Kullback-Leibler distance will be respected. If the new and old distribution differ too much there is the possibility of a destructive update. So, we should better check if $\eta = \sqrt{\frac{2\beta}{x^T \widehat{\boldsymbol{F}_{\pi\theta}}\, x}}$ makes the new distribution complying with the Kullback-Leibler divergence constraint. Also, we need to check that the updates are not worsening the policy, so we compute for each step of the trajectories the value of the surrogate objective $\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}\hat{A}_t$ (see eq. 7.41) and we check that the average of it computed for the new policy is greater or equal than the average of it computed for the old policy. If either the KL divergence constraint is not satisfied, or the average surrogate objective has worsened, we exponentially decrease η until we find a value that satisfies those conditions. This is basically a line search. If in a reasonable number of iterations, we can't find any decreased value of η that satisfies those conditions, we reject this policy update.

Let us now see all the pieces together with the TRPO algorithm pseudocode.

Algorithm 7.1 **Trust Region Policy Optimization**

Require: Kullback-Leibler divergence constraint β

Require: backtracking coefficient b , with $0 < b < 1$

Require: maximum number of backtracking steps U

Require: Value network step size ω

Require: Initialize parameters θ of network π_θ with small random values

Require: Initialize parameters ψ of network V_ψ with small random values

For $k = 0, 1, 2, \dots$ do:

collect trajectories $D_k = \{ \tau_i \}$ using policy $\pi_k(\theta_k)$

compute rewards-to-go $G(\tau_{i,t})$, for each t of each τ_i

compute advantage estimates $\widehat{A_t}$ using current estimate of value function $V_{\psi k}$:

$$\widehat{A_t} = G(\tau_t) - V_{\psi k}(s_t)$$

estimate policy gradient as:

$$\widehat{g_k} = \frac{1}{|D_k|} \sum_{\tau \epsilon \quad k} \sum_{t=0}^{T-1} \nabla_\theta \left(\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \hat{A}_t \right)$$

set a function that returns the multiplication of a vector v by Fisher Inf. Matrix: $\widehat{\boldsymbol{F_{\pi\theta}}} \cdot v$

either by $\widehat{\boldsymbol{F_{\pi\theta}}} \cdot v = \left(\frac{1}{\sum_{\tau \epsilon D_k} N_\tau} \sum_{\tau \epsilon \quad k} \sum_{t=0}^{N_\tau - 1} \left(\mathbf{J}(s_t)\right)^{\boldsymbol{T}} \mathbf{M}(s_t)\, \mathbf{J}(s_t) \right) \cdot v$

or (faster) by $\widehat{\boldsymbol{F_{\pi\theta}}} \cdot v = \nabla_{\theta'} \left(\frac{1}{\sum_{\tau \epsilon D_k} N_\tau} \left(\sum_{\tau \epsilon \quad k} \sum_{t=0}^{N_\tau - 1} \left(\frac{\partial (kl(\, \mu_{OLD}, \mu_{\theta'}(s_t)\,))}{\partial \theta'} \right)^T \right) \cdot v \right)$

run the Conjugate Gradient algorithm with that function to find update direction vector

$$x_k = \widehat{\boldsymbol{F_{\pi\theta}}}^{-1} \widehat{g_k}$$

compute the step size upper limit $\eta = \sqrt{\frac{2\beta}{{x_k}^T \widehat{\boldsymbol{F_{\pi\theta}}}\, x_k}}$

update the policy network by backtracking line search, with $j \in \{0,1,2 \dots U\}$ being the smallest number that satisfies the Kullback-Leibler divergence and that improves the average surrogate objective $\frac{\pi_{\theta'}(a|s)}{\pi_\theta(a|s)} \hat{A}_t$ on the samples:

$\theta_{k+1} \leftarrow \theta_k + b^j \eta x_k$

(no policy update if conditions are not satisfied)

For $z = 0, 1, 2, \dots, Z-1$ do a value function gradient descent iteration:

estimate the value function gradients as:

$$\widehat{h_k} = \frac{1}{|D_k|} \sum_{\tau \epsilon D_k} \frac{1}{T} \sum_{t=0}^{T-1} \nabla_\psi (V_\psi(s_t) - G(\tau_t))^2$$

update the value function with a gradient descent step (or other method):

$$\psi_{k+1} \leftarrow \psi_k + \omega \widehat{h_k}$$

end of for

end of for

7.12 Vine Sampling Scheme

The original authors [Schulman et al. 2015] experimented also a sampling scheme named "Vine", applicable on any environment in which it is possible to restart from a certain well-defined state (as in a simulated environment). The Vine sampling consists of trying different actions from the same starting point, generating different trajectories to compute an average Q estimate, which in this way will have a lower variance.

8. Proximal Policy Optimization

The Proximal Policy Optimization algorithm [Schulman et al. 2017] builds on the ideas and theoretical framework of Trust Region Policy Optimization. To understand this chapter it is necessary to have read the TRPO chapter (Ch. 7) until equation 7.41 included. In fact, PPO aims at optimizing the same eq. 7.41:

$$\underset{\theta'}{maximize}\ L_{\theta}(\theta') \ = E_{\tau\sim\pi_{\theta}}\left[\frac{\pi_{\theta'}(a|s)}{\pi_{\theta}(a|s)}A^{\pi_{\theta}}(s,a)\right]$$

$$s.t. \qquad E_{\tau\sim\pi_{\theta}}[D_{KL}(\pi_{\theta}(\cdot\ |s)||\pi_{\theta'}(\cdot\ |s))] \leq \beta$$

While in TRPO there was only one policy gradient ascent iteration in each optimization round (in which the step length was computed so to not make the new policy too different from the old), in PPO the algorithm does more than one policy gradient ascent iteration in each optimization round using the same set of trajectories: the trajectories are re-used as long as the new policy is not too different from the old.

What distinguishes Proximal Policy Optimization from TRPO is that it uses a different strategy to constraint the KL divergence: (1) it uses gradient clipping as a way to never have a too big gradient update, and (2) it checks when the old and new policies are too divergent so to stop reusing the samples generated by the old policy.

For ease of reading, let us call the ratio between the two probabilities $d(\theta) = \frac{\pi_{\theta'}(a|s)}{\pi_{\theta}(a|s)}$. A new surrogate objective function that uses clipping, called L_{θ}^{CLIP} may be devised:

$$L_{\theta}^{CLIP}(\theta') = E_{\tau\sim\pi_{\theta}}[\ \min\ (\ d(\theta)A^{\pi_{\theta}}(s,a)\ ,clip(\ d(\theta),1-\epsilon,1+\epsilon\ )A^{\pi_{\theta}}(s,a)\ )\quad] \tag{8.1}$$

written in a simplified way:

$$L_{\theta}^{CLIP}(\theta') = E_{\tau\sim\theta}[\ \min\ (\quad d(\theta)A^{\pi_{\theta}}\quad ,\quad clip(\ d(\theta),1-\epsilon,1+\epsilon\ )\ A^{\pi_{\theta}}\quad)\ \ ] \tag{8.2}$$

That means that, generating actions with policy θ, the objective is the expectation of the minimum among $d(\theta)A^{\pi_\theta}$ and $clip(\, d(\theta), 1-\epsilon, 1+\epsilon\,)\ A^{\pi_\theta}$.

The $clip$ function imposes both a ceiling and a floor to $d(\theta)$: if $d(\theta) > 1+\epsilon$ it returns $1+\epsilon$, if $d(\theta) < 1-\epsilon$ it returns $1-\epsilon$, otherwise it returns $d(\theta)$.

The $\min$ function then has the result of returning a lower bound on the unclipped objective. In this way, when the probability ratio would make the objective improve, it is bounded to be $< 1+\epsilon$, so to not have too big steps. While when it would make the objective worse it is unbounded towards negative infinity. When the objective is negative, it is bounded towards zero so to always have some negative value that has an impact on the policy update.

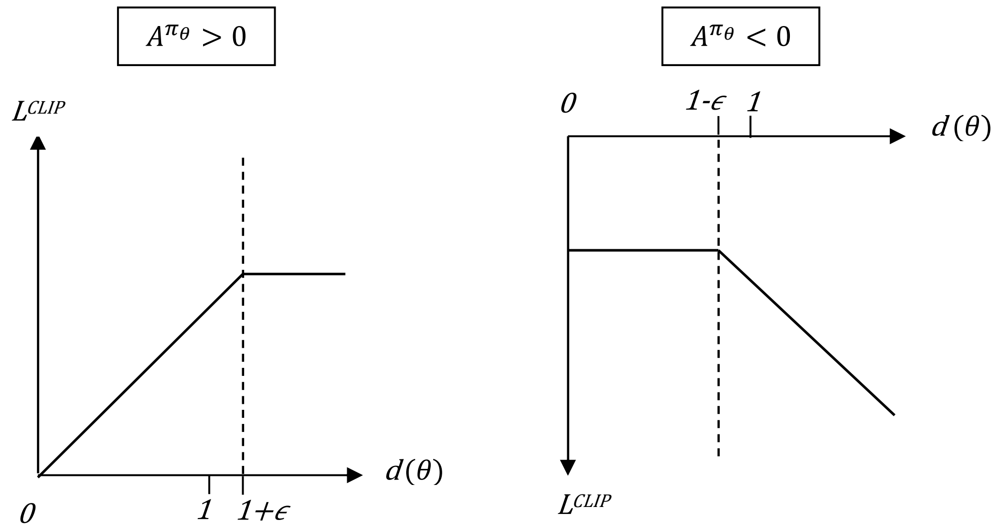

Figure 8.1 Surrogate objective with policy ratio clipping in Proximal Policy Optimization

To practically use it, we have to do gradient descent with respect to θ' on equation 8.1 or 8.2 , using an automatic differentiation library that supports clipping.

This clipping does not guarantee that the new policy is not too different from the old policy, so it is necessary to check the Kullback-Leibler divergence of the old policy from the new policy, using the taken actions and computing their probability both under the old policy and under the new policy. Then stop the policy gradient ascent iterations in case the KL divergence is over a certain threshold (“early stopping”). This avoids using samples generated from a policy that has a distribution too different from the one that gets optimized. The KL divergence computed in this way is just an approximation because is the average of the KL divergence for each sample. Since it is sampled on the distribution of actions from the old policy, it is already the empirical expectation over the old policy (so there is no need to multiply for $\pi_{\theta old}(a_t|s_t)$ in the divergence formula):

$$\widehat{Kl} = \frac{1}{T}\sum_{t}^{T} \log\left(\pi_{\theta old}(a_t|s_t)\right) - \log\left(\pi_{\theta}(a_t|s_t)\right) \tag{8.3}$$

Algorithm 8.1 **Proximal Policy Optimization with ratio clipping and early stopping**

Require: Policy network step size η

Require: Value network step size ω

Require: Threshold for Kullback-Leibler divergence based early stopping ζ

Require: Initialize parameters θ of network $\pi_{\theta}$with small random values

Require: Copy parameters θ_{old} of network $\pi_{\theta old}$ from θ

Require: Initialize parameters ψ of network $V_{\psi}$with small random values

For $k = 0, 1, 2, \ldots$ do:

collect trajectories $D_k = \{\tau_i\}$ using policy $\pi_{\theta old}$

compute rewards-to-go $G(\tau_{i,t})$, for each t of each τ_i

compute advantage estimates $\widehat{A_t}$ using current estimate of value function $V_{\psi k}$:

$\widehat{A_t} = G(\tau_t) - V_{\psi k}(s_t)$

For $m = 0, 1, 2, \ldots, M-1$ do a policy gradient ascent iteration:

compute policy ratios:

$$d_t(\theta) = \frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)}$$

compute clipped surrogate losses:

$$L_{\theta}^{CLIP}(\theta')_t = \min\left(d_t(\theta)\widehat{A_t}, \quad clip(d_t(\theta), 1-\epsilon, 1+\epsilon)\ \widehat{A_t} \right)$$

compute KL divergence between $\pi_{\theta old}(a_t|s_t)$ and $\pi_{\theta}(a_t|s_t)$ on taken actions:

$$\widehat{Kl} = \frac{1}{|D_k|}\sum_{\tau\epsilon\ k}\frac{1}{T}\sum_{t=0}^{T-1} \log\left(\pi_{\theta old}(a_t|s_t)\right) - \log\left(\pi_{\theta}(a_t|s_t)\right)$$

If $\widehat{Kl} >= \zeta$:

stop doing policy gradient ascent and exit this internal For-cycle

end of If

estimate policy gradient as:

$$\widehat{g_k} = \frac{1}{|D_k|}\sum_{\tau\epsilon\ k}\sum_{t=0}^{T-1} \nabla_{\theta} L_{\theta}^{CLIP}(\theta')_t$$

update the policy network with gradient ascent (or other methods like Adam):

$\theta \leftarrow \theta + \eta\widehat{g_k}$

end of for

Copy optimized policy into fixed policy: $\theta_{old} \leftarrow \theta$

For $n = 0, 1, 2, \ldots, N-1$ do a value function gradient descent iteration:

estimate the value function gradients as:

$$\widehat{h_k} = \frac{1}{|D_k|} \sum_{\tau \epsilon \ k} \frac{1}{T} \sum_{t=0}^{T-1} \nabla_\psi (V_\psi(s_t) - G(\tau_t))^2$$

update the value function with a gradient descent step (or other method):

$\psi_{k+1} \leftarrow \psi_k + \omega\widehat{h_k}$

end of for

end of for

PPO is considered on-policy because it aims at optimizing the same policy function that generates the training trajectories (and it is not valid to use an arbitrary policy function to generate trajectories). But actually, it is also *slightly* off-policy because it does possibly more than one step of optimization with the same trajectories, and at each step further than the first it is optimizing a policy that is not *exactly* the same that generated trajectory: it is very similar (small KL-divergence) but not the same.

9. Deep Deterministic Policy Gradient

9.1 Base Idea

Deterministic Policy Gradient is an algorithm invented by [Silver et al. 2014] (with similarities to [Hafner and Riedmiller 2011]), and then adapted to deep networks by [Lillicrap et al. 2016], that allows us to do an effective off-policy policy gradient optimization for tasks with continuous actions. The basic ideas are (a) to have a target policy that is deterministic, which means that for each state the policy chooses only one well-determined action, and (b) to improve the policy greedily, that here is done by moving the policy in the direction of the gradient of $Q^{\pi}(s,a)$. This is considered to be a policy gradient algorithm that is similar in spirit to Q-Learning because of the greediness of learning and determinism of the target policy. The full algorithm is off-policy, but we will present an on-policy version at first, before describing the changes to make it off-policy.

Let us digress a little to prepare field for the full picture.

If you recall, the "*Expected return*" as used until now is the one expressed in eq. 1.5 , in section 1.2 :

$$J(\pi) = E_{\tau \sim \pi}[G(\tau)] = E_{a \sim \pi,\ s_0 \sim \rho_0,\ s \sim P}\ [G(\tau)]$$

where $G(\tau)$ is the total return $G(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t$.

In that formula, $J(\pi)$ is an expectation over the trajectory τ, that means that it is actually an expectation over actions and over states, with actions a distributed by policy π , initial state s_0 distributed by ρ_0 , and the other states s distributed by the transition function P. Until now, the policy π has been considered stochastic: we selected an action basing on a random sample that followed that distribution. For discrete actions that means that every possible discrete action occupies a subset of [0,1], right next to the subset of the previous action, as big as the probability assigned by the policy function. Then a random number between 0 and 1 is sampled, which permits us to select one of those subsets and hence the corresponding action. For continuous actions instead, for each action dimension the policy π gives the average of the value and the variance (which may be fixed, a priori), and those parameters are used within a certain distribution (usually a normal distribution) to sample each action dimension.

This kind of stochastic behaviour is responsible for some variety of exploration, and for a minimum level of robustness in adversarial games (but beware: for some optimal policy the level of exploration and variety of action may be very little).
It is also the reason why the Expected Return $J(\pi)$ is an expectation also over actions and not only over states: we could make the computation of its gradient simpler if it was only over states. How to do it ? We could decide that our target policy π is deterministic instead of stochastic, and hence instead of sampling the action, we just choose the action with maximum probability. So, we would not have any expectation over actions anymore. As we will see below, this will also allow us to derive a policy gradient ascent that is "greedy": in the direction of the gradient of $Q^{\pi}(s, a)$. That is the basic idea of Deterministic Policy Gradient.
Now, for discrete actions there is already an algorithm that includes a similar idea: Q-Learning (see Ch.2), which is not a policy based algorithm but essentially applies the same idea of having the target policy choosing only the action with maximum probability (which in Q-Learning is the action with the maximum action-state value, since Q-Learning is a value-based algorithm), hence being both deterministic and greedy at the same time.
As we already saw, applying Q-Learning to continuous actions instead is complicated, since it would entail finding the maximum of a function approximator (the one used for the action-value function) which is very problematic to do if the function approximator is a deep neural network. But it is still possible to apply the idea of deterministic policy to policy-gradient systems with continuous actions: instead of sampling the action using the average and the variance given by the policy function, we can just choose always that average as action. This will make the expectation over the actions collapse to just one action, and simplify the computation of the gradient of $J(\pi)$. A second benefit of having a deterministic policy is that it is applicable to environments where stochastic policies cannot be used, such as in robots with a differentiable control policy where it is not possible to inject noise.
Since a deterministic policy does not offer any level of exploration per se, a different exploratory policy ("*behavior policy*") must be used which guarantees some variety of actions, so the whole algorithm will be off-policy.

9.2 Action-Value Gradient

Let us rewrite the expected return $J(\pi)$ in a way that can be handy with deterministic policy. The first thing to note is that the expectation of $\sum_{t=0}^{\infty} \gamma^t r_t$,over actions a distributed by π and states s distributed by ρ_0 (if initial) and P, is equal to the expectation of $Q^\pi(s,a)$, as already showed in eq. 1.8.

$$E_{\tau\sim\pi}[G(\tau)] = E_{\tau\sim\pi}[Q^\pi(s,a)\,] = E_{a\sim\pi,\ s_0\sim\rho_0,\ s\sim P}\ [Q^\pi(s,a)] \tag{9.1}$$

Then, analogously to what we did in section 7.2 , we can define the probability of being in state s at time t, depending on policy π as $Prob^\pi(s_t = s\,)$.
And we can define the frequency $\xi_\theta(s)$ of a state s as the number of times that s is expected to be visited, computed as unnormalized (it is a frequency, not a probability) and time-discounted, under the policy π :

$$\xi_\pi(s) = Prob^\pi(s_0 = s\,)\ + \gamma Prob^\pi(s_1 = s\,)\ + \gamma^2 Prob^\pi(s_2 = s\,)\ + \cdots \tag{9.2}$$

To be noted that $\xi_\pi(s)$ it is not a probability distribution, but it is a time-discounted frequency (where γ is the discount factor), even if in the original paper [Silver et al. 2014] that is defined "*the (improper) discounted state distribution*" and is denoted as $\rho^\pi(s')$.

So, the expected return $J(\pi)$ can be rewritten as:

$$J(\pi)\ = \sum_t \sum_s Prob^\pi(s_t = s\,)\ \sum_a \pi(a_t|s_t)\,\gamma^t Q^\pi(s_t,a_t)$$

$$J(\pi)\ = \sum_s \sum_t \gamma^t Prob^\pi(s_t = s\,)\ \sum_a \pi(a_t|s_t)\, Q^\pi(s_t,a_t)$$

$$J(\pi)\ = \sum_s \xi_\pi(s) \sum_a \pi(a_t|s_t)\, Q^\pi(s_t,a_t) \tag{9.3}$$

Which can be written more generally also with integral notation as:

$$J(\pi) = E_{\tau\sim\pi}[Q^\pi(s,a)\,] = \int_S \xi_\pi(s) \int_a \pi(a|s) Q^\pi(s,a) \tag{9.4}$$

At this point if the policy π is deterministic, we have that the integral over actions disappears because we have only one possible action for each state, and the value of $\pi(a|s)$ for that action is a is $= 1$. We can name μ_θ the vector of outputs by the policy neural network which computes the averages for each action dimension, parametrized by weights θ . We can also make explicit that the policy π depends on θ, writing it as π_θ , and writing $J(\mu_\theta)$ instead of $J(\pi)$. The expected return then is:

$$J(\mu_\theta) = \int_S \xi_{\pi_\theta}(s) Q^{\mu_\theta}(s, \mu_\theta(s)\ ) \tag{9.5}$$

Now, in that form it is evident that we can do gradient ascent on $J(\mu_\theta)$ with respect to θ , sampling the states that appear while the agent is following policy π (those states will appear with discounted frequency $\xi_\pi(s)$).

$$\nabla_\theta J(\mu_\theta) = \nabla_\theta \int_S \xi_{\pi_\theta}(s) Q^{\mu_\theta}(s, \mu_\theta(s)\ ) \tag{9.6}$$

Naming η the step size of parameters' updates, the update rule is:

$$\theta' \leftarrow \theta + \eta \nabla_\theta \int_S \xi_{\pi_\theta}(s) Q^{\mu_\theta}(s, \mu_\theta(s)\ ) \tag{9.7}$$

Applying the chain rule of calculus, we find the gradient composed by the gradient of μ_θ wrt θ , multiplied by the gradient of $Q^{\mu_\theta}(s, a)$, with respect to a , having a valued at point $a = \mu_\theta(s)$, that is the a chosen by the deterministic policy .

$$\theta' \leftarrow \theta + \eta \nabla_\theta \int_S \xi_{\pi_\theta}(s) \nabla_\theta \mu_\theta(s) \cdot \nabla_a Q^{\mu_\theta}(s, a) \Big|_{a\, =\, \mu_\theta(s)} \tag{9.8}$$

Now, when we sample a process where a state has (time-discounted) frequency $\xi_{\pi_\theta}(s)$, it means that we are sampling a state which is distributed by the probability $Prob^{\pi_\theta}(s_t = s\ )$. That means:

$$\nabla_\theta \int_S \xi_{\pi_\theta}(s) Q^{\mu_\theta}(s, \mu_\theta(s)) \;\propto\; E_{s\sim Prob^{\mu_\theta}}\left[\nabla_\theta \mu_\theta(s) \cdot \nabla_a Q^{\mu_\theta}(s,a)\Big|_{a\,=\,\mu_\theta(s)}\right]$$

(9.9)

With a little abuse of notation, we can say that from eq. 9.7 we can deduce eq. 9.9 below (the abuse of notation consists in the fact that η of eq. 9.7 is not the same η of eq. 9.9).

$$\theta' \leftarrow \theta + \eta E_{s\sim Prob^{\mu_\theta}}\left[\nabla_\theta \mu_\theta(s) \cdot \nabla_a Q^{\mu_\theta}(s,a)\Big|_{a\,=\,\mu_\theta(s)}\right]$$

(9.10)

To be noted that, following original paper [Silver et al. 2014] description, $\nabla_\theta \mu_\theta(s)$ is a transposed Jacobian matrix where each column d is the gradient $\nabla_\theta [\mu_\theta(s)]_d$ of the dth action dimension with respect to the policy parameters θ . It becomes clear then that $\nabla_a Q^{\mu_\theta}(s,a)$ is a column vector, and both θ and θ' are column vectors as well. (In usual notation, $\nabla_\theta \mu_\theta(s)$ would have made the reader think that it was a Jacobian matrix, and not a transposed Jacobian matrix, but original authors specified the orientation of columns and rows, and following it makes it a transposed Jacobian).

Eq. 9.9 shows that it is irrelevant the fact that changing the policy μ_θ by the gradient ascent step in eq. 9.9 could change in turn the distribution of the states: the gradient of the expected return does not depend on the gradient of the state distribution (which we do not know), similarly to what we saw with stochastic on-policy policy gradient in section 3.2 . In other words: we are already optimizing the performance $J(\mu_\theta)$, hence the optimization step in 9.9 will indeed improve the policy and we do not need to care about change in state distribution. Original authors [Silver et al. 2014, supplement material] use a slightly different proof that originates as a deterministic version of Policy Gradient Theorem.

One remarkable thing is that the deterministic policy gradient represents the limit of the stochastic policy gradient, when the variance of the stochastic policy tends to zero, for a family of stochastic policies. This is important because it means that many techniques applicable to (stochastic) policy gradient algorithms may be applied also to deterministic policy gradient. We are not writing the proof here because it is quite long, and it is already written in a detailed and clear way in the supplementary material of [Silver et. al 2014]. We will just outline the theorem definition.

Suppose that $\mu_\theta(s)$ is a deterministic policy function, parametrized by θ, which outputs a vector of deterministic action dimensions (i.e. $\mu_\theta(s)$ has the same meaning as it had above in this chapter). Consider a stochastic policy function $\pi_{\mu_\theta,\sigma}(a|s) = \upsilon_\sigma(\mu_\theta(s), a)$ satisfying some conditions called "regular delta-approximation" (detailed in [Silver et. al 2014]), where σ is the variance. An example of such a stochastic policy could be a gaussian policy function with mean $\mu_\theta(s)$ and variance σ.
Then:

$$\lim_{\sigma \to 0^+} \nabla_\theta J(\pi_{\mu_\theta,\sigma}) = \nabla_\theta J(\mu_\theta) \tag{9.11}$$

On the left-hand side there is the gradient of the stochastic policy $\pi_{\mu_\theta,\sigma}(a|s)$, and on the right-hand side the gradient of the deterministic policy $\mu_\theta(s)$, so as we anticipated it means that the deterministic policy gradient is the limit of the stochastic policy gradient when the variance of the stochastic policy tends to zero.

9.3 On-Policy Deterministic Actor-Critic

Now, we have all elements to start build a learning algorithm.
If the agent follows the same policy that we optimize, that means that we are doing on-policy optimization, and we can use an approximation function parametrized by ψ as critic $Q_\psi(s, a)$ to estimate $Q^{\mu_\theta}(s, \mu_\theta(s)\,)$ with a Sarsa algorithm (see section 2.10). This on-policy version is slower than the off-policy one because it needs to spend more time generating trajectories, and it is meant only as a preliminary example to later introduce the off-line algorithm.

Algorithm 9.1 **On-Policy Deterministic Actor-Critic**

Require: Policy network step size η

Require: Q-Value network step size ω

Require: Initialize parameters θ of network μ_{θ_k} with small random values

Require: Initialize parameters ψ of network Q_ψ with small random values

For $k = 0, 1, 2, \ldots$ do:

collect trajectories $D_k = \{ \tau_i \}$ using policy μ_{θ_k}

For $n = 0, 1, 2, \ldots, N-1$ do a q-value function gradient descent iteration:

For $m = 0, 1, 2, \ldots, M-1$ do:

From trajectories D_k sample a transition $\langle s_{m,t}, a_{m,t}, s_{m,t+1}, r_{m,t} \rangle$

compute $a_{m,t+1} = \mu_{\theta_k}(s_{m,t+1})$

$y_{m,t} = r_{m,t} + \gamma\, Q_\psi(s_{m,t+1}, a_{m,t+1})$

End of for

estimate the q-value network gradients as:

$$\widehat{h_k} = \frac{1}{M} \sum_{m=0}^{M-1} (\, y_{m,t} - Q_\psi(s_{m,t}, a_{m,t})\,) \nabla_\psi Q_\psi(s_{m,t}, a_{m,t})$$

update the q-value network with a gradient descent step (or other method):

$$\psi \leftarrow \psi + \omega \widehat{h_k}$$

end of for

estimate policy gradient as:

$$\widehat{g_k} = \frac{1}{|D_k|} \sum_{\tau \epsilon \; k} \frac{1}{T} \sum_{t=0}^{T-1} \nabla_\theta \mu_\theta(s_t) \cdot \nabla_a Q_\psi(s_t, a) \Big|_{a = \mu_{\theta_k}(s_t)}$$

update the policy network with gradient ascent (or other methods like Adam):

$$\theta_{k+1} \leftarrow \theta_k + \eta \widehat{g_k}$$

end of for

9.4 Off-Policy Deterministic Actor-Critic

If instead we want to build an off-policy algorithm, we have a behavior policy parametrized by β, so under that policy we denote $\xi_\beta(s)$ the time-discounted frequency of a state s. The performance to be optimized, following [Silver et al. 2014], then is the total expected value where the frequency of the visited states is the one under the behavior policy, while the q-value of each state is computed as if we followed the target policy:

$$J_\beta(\mu_\theta) = \int_S \xi_\beta(s) Q^{\pi_\theta}(s, \mu_\theta(s)) \tag{9.11}$$

We should be aware that this goal performance is not theoretically justified, because analogously as we discussed in section 4.3 about off-policy policy gradients we know that when using function approximators:

$$J_\beta(\mu_\theta{}') \geq J_\beta(\mu_\theta) \; does\ not\ imply \; J(\mu_\theta{}') \geq J(\mu_\theta) \tag{9.12}$$

As with the on-policy algorithm, we can use an approximation function parametrized by ψ as critic $Q_\psi(s, a)$ to estimate $Q^{\pi_\theta}(s, \mu_\theta(s))$ with a Sarsa algorithm (see section 2.10). The algorithm then is very similar to On-Policy Deterministic Actor-Critic (algorithm 9.1), with the difference that the trajectories are generated following the behavior policy and not the target policy. Since the policy is deterministic, this Sarsa evaluation coincides with an off-policy Expected Sarsa evaluation.

Algorithm 9.2 **Off-Policy Deterministic Actor-Critic**

Require: Policy network step size η

Require: Q-Value network step size ω

Require: Behavior policy β

Require: an empty replay buffer B

Require: Initialize parameters θ of network μ_{θ_k} with small random values

Require: Initialize parameters ψ of network Q_ψ with small random values

For $k = 0, 1, 2, \ldots$ do:

collect trajectories $D_k = \{\tau_i\}$ using behavior policy β

put all transitions $\langle s, a, s', r\rangle$ of all trajectories in replay buffer B

For $n = 0, 1, 2, \ldots, N-1$ do a q-value function gradient descent iteration:

For $m = 0, 1, 2, \ldots, M-1$ do:

From replay buffer B sample a transition $\langle s_{m,t}, a_{m,t}, s_{m,t+1}, r_{m,t}\rangle$

compute $a_{m,t+1} = \mu_{\theta_k}(s_{m,t+1})$

$y_{m,t} = r_{m,t} + \gamma\, Q_\psi(s_{m,t+1}, a_{m,t+1})$

End of for

estimate the q-value network gradients as:

$$\widehat{h_k} = \frac{1}{M}\sum_{m=0}^{M-1}(\, y_{m,t} - Q_\psi(s_{m,t}, a_{m,t})\,)\nabla_\psi Q_\psi(s_{m,t}, a_{m,t})$$

update the q-value network with a gradient descent step (or other method):

$\psi \leftarrow \psi + \omega\widehat{h_k}$

end of for

estimate policy gradient as:

From replay buffer B sample U transitions (you need only the state s_u)

$$\widehat{g_k} = \frac{1}{U}\sum_{u=0}^{U-1}\nabla_\theta\mu_\theta(s_u)\cdot\nabla_a Q_\psi(s_u, a)\Big|_{a=\mu_{\theta_k}(s_u)}$$

update the policy network with gradient ascent (or other methods like Adam):

$\theta_{k+1} \leftarrow \theta_k + \eta\widehat{g_k}$

end of for

9.5 Compatible Function Approximator

The gradient of the approximated q-value function is used in the policy gradient update formula instead of the real, unknown, q-value gradient, and this implies that the resulting policy gradient may not follow the true gradient (in the off-policy algorithm this is even worsened by the presence of the three conditions “function approximation”, “bootstrapping”, “off-policy training” that may lead the function approximator to diverge instead of converge, as discussed in section 2.4).

For this reason, we should find a way to recognize if a certain function approximator will have an approximated q-value gradient that follows the true q-value gradient. The original authors call it “Compatible Function Approximator”, and discover that there is a class of such approximators characterized by two well-defined properties enunciated below in the theorem. Please note that the following theorem is valid both for on-policy and off-policy deterministic actor critic, and that with the symbol E we denote both the expectation $E_{s\sim Prob^{\mu_\theta}}$ over states distributed by the target policy μ_θ (on-policy case) and the expectation $E_{s\sim Prob^{\beta}}$ over states distributed by the behavior policy β (off-policy case). In the same way, in the theorem when we write the performance gradient $\nabla_\theta J(\mu_\theta)$, we mean both the on-policy gradient (properly: $\nabla_\theta J(\mu_\theta)$) and the off-policy gradient (it would be properly denoted as $J_\beta(\mu_\theta)$ as previously seen).

Compatible function approximation theorem: saying that a function approximator $Q_\psi(s,a)$ parametrized by ψ is *compatible* with a deterministic policy $\mu_\theta(s)$ means that:

$$\nabla_\theta J(\mu_\theta) = E\left[\nabla_\theta \mu_\theta(s) \cdot \nabla_a Q_\psi(s,a) \Big|_{a\,=\,\mu_\theta(s)}\right] \tag{9.12}$$

Two conditions imply the compatibility:

$$1.\ \ \nabla_a Q_\psi(s,a) \Big|_{a\,=\,\mu_\theta(s)} = \left(\nabla_\theta \mu_\theta(s)\right)^{\mathrm{T}} \psi$$

$$where\,\psi\ is\ a\ column\ vector\ of\ same\ length\ of\ \theta \tag{9.13}$$

$$2.\ \ \psi\ minimizes\ the\ mean-squared\ error\ MSE(\theta,\psi) = E[\epsilon(s,\theta,\psi)^{\mathrm{T}}\epsilon(s,\theta,\psi)] \tag{9.14}$$

$$where\ \epsilon(s,\theta,\psi) = \nabla_a Q_\psi(s,a)\Big|_{a\,=\,\mu_\theta(s)} - \nabla_a Q^{\mu_\theta}(s,a)\Big|_{a\,=\,\mu_\theta(s)} \tag{9.15}$$

Proof:

By condition 2 (eq. 9.14), ψ minimizes $MSE(\theta,\psi)$, then the gradient of ϵ^2 w.r.t. ψ must be zero.

$$\nabla_\psi\, MSE(\theta,\psi) = 0$$

$$E\left[\nabla_\psi\big(\epsilon(s,\theta,\psi)^{\mathrm{T}}\epsilon(s,\theta,\psi)\big)\right] = 0$$

applying derivative product rule:

$$E\left[\left(\nabla_\psi\epsilon(s,\theta,\psi)\right)^{\mathrm{T}}\epsilon(s,\theta,\psi) + \left(\epsilon(s,\theta,\psi)^{\mathrm{T}}\nabla_\psi\epsilon(s,\theta,\psi)\right)^{\mathrm{T}}\right] = 0$$

$$E\left[2\left(\nabla_\psi\epsilon(s,\theta,\psi)\right)^{\mathrm{T}}\epsilon(s,\theta,\psi)\right] = 0$$

$$E\left[\left(\nabla_\psi\epsilon(s,\theta,\psi)\right)^{\mathrm{T}}\epsilon(s,\theta,\psi)\right] = 0 \tag{9.16}$$

By condition 1 (eq. 9.13), we find that $\nabla_\psi\epsilon(s,\theta,\psi) = \nabla_\theta\mu_\theta(s)$. Let us show it starting from eq. 9.15

$$\epsilon(s,\theta,\psi) = \nabla_a Q_\psi(s,a)\Big|_{a\,=\,\mu_\theta(s)} - \nabla_a Q^{\mu_\theta}(s,a)\Big|_{a\,=\,\mu_\theta(s)}$$

$$\nabla_\psi\epsilon(s,\theta,\psi) = \nabla_\psi\left(\nabla_a Q_\psi(s,a)\Big|_{a\,=\,\mu_\theta(s)} - \nabla_a Q^{\mu_\theta}(s,a)\Big|_{a\,=\,\mu_\theta(s)}\right)$$

$$\nabla_\psi\epsilon(s,\theta,\psi) = \nabla_\psi\nabla_a Q_\psi(s,a)\Big|_{a\,=\,\mu_\theta(s)}$$

plugging-in eq. 9.13:

$$\nabla_{\psi}\epsilon(s,\theta,\psi) = \nabla_{\psi}\left(\left(\nabla_{\theta}\mu_{\theta}(s)\right)^{\mathrm{T}}\psi\right)$$

$$\nabla_{\psi}\epsilon(s,\theta,\psi) = \nabla_{\theta}\mu_{\theta}(s) \tag{9.17}$$

plugging eq. 9.17 into eq. 9.16:

$$E\left[\left(\nabla_{\theta}\mu_{\theta}(s)\right)^{\mathrm{T}}\epsilon(s,\theta,\psi)\right] = 0$$

plugging-in eq. 9.15:

$$E\left[\left(\nabla_{\theta}\mu_{\theta}(s)\right)^{\mathrm{T}}\left(\nabla_{a}\, Q_{\psi}(s,a)\Big|_{a\,=\,\mu_{\theta}(s)} - \nabla_{a}\, Q^{\mu_{\theta}}(s,a)\Big|_{a\,=\,\mu_{\theta}(s)}\right)\right] = 0$$

$$E\left[\left(\nabla_{\theta}\mu_{\theta}(s)\right)^{\mathrm{T}}\nabla_{a}\, Q_{\psi}(s,a)\Big|_{a\,=\,\mu_{\theta}(s)}\right] = E\left[\left(\nabla_{\theta}\mu_{\theta}(s)\right)^{\mathrm{T}}\nabla_{a}\, Q^{\mu_{\theta}}(s,a)\Big|_{a\,=\,\mu_{\theta}(s)}\right]$$

$$E\left[\left(\nabla_{\theta}\mu_{\theta}(s)\right)^{\mathrm{T}}\nabla_{a}\, Q_{\psi}(s,a)\Big|_{a\,=\,\mu_{\theta}(s)}\right] = \nabla_{\theta}J(\mu_{\theta}) \tag{9.18}$$

Which proves the theorem (statement 9.12).

■

It turns out that for any policy $\mu_{\theta}(s)$ it is possible to find a function approximator that satisfies the condition 1 (eq. 9.13) of compatible function approximators, with the form:

$$Q_{\psi}(s,a) = \left(a - \mu_{\theta}(s)\right)^{\mathrm{T}}\left(\nabla_{\theta}\mu_{\theta}(s)\right)^{\mathrm{T}}\psi + V_{v}(s) \tag{9.18}$$

Where $V_{v}(s)$ is any differentiable baseline independent from the action a , parametrized by v. A way to see this is to think of $V_{v}(s)$ as a value function, for instance a linear one, where, if called the state features as $\phi(s)$, we can have $V_{v}(s) = v^{\mathrm{T}}\,\phi(s)$.
Then, if we consider that the q-value function is equivalent to the state value function plus the advantage function (eq. 1.13), we can think of $\left(a - \mu_{\theta}(s)\right)^{\mathrm{T}}\left(\nabla_{\theta}\mu_{\theta}(s)\right)^{\mathrm{T}}\psi$ as a linear advantage function. Such advantage function would have state-action features $\phi(s,a) = \left(\nabla_{\theta}\mu_{\theta}(s)\right)\left(a - \mu_{\theta}(s)\right)$ and parameters ψ , so to have $A^{\psi}(s,a) = \phi(s,a)^{\mathrm{T}}\psi$.

We can also denote the deviation of the action from current policy as $\Delta_a = a - \mu_\theta(s)$, in this way the advantage function may be written as $A^{\psi}(s, \mu_\theta(s) + \Delta_a) = {\Delta_a}^{\mathrm{T}}\psi$.
To indicate that $Q_\psi(s,a)$ is now parametrized not only by ψ but also by v, because it is the sum of the advantage function A^{ψ} and the value function V_v , we can write it as $Q_{\psi,v}(s,a)$.

Linear approximation is not very precise to predict action-value globally (it can diverge towards infinity), but it can work well as local critic. Here it is used in an advantage function which indicates the local advantage with respect to the current policy: a linear approximator may be enough to find the right direction in which the policy parameters should be updated.

Let us examine the second condition (eq. 9.14 and 9.15): ψ should minimize the mean squared error between the approximated gradient of Q_ψ and the true gradient of Q^{μ_θ}. To fulfill this condition with a linear approximator, we should build a linear regression to approximate $\nabla_a Q^{\mu_\theta}(s,a)\Big|_{a=\mu_\theta(s)}$, with $\phi(s,a)$ as features, and ψ as parameters. But we are not doing exactly that, because we are learning parameters ψ through Sarsa/Q-Learning evaluation, so ψ is not guaranteed to actually minimize $MSE(\theta,\psi)$, hence the condition 2 is not satisfied (also, it is hard to obtain unbiased samples of the real gradient of Q^{μ_θ}). We can only say that despite not having the condition 2 satisfied, in practice our policy evaluation may be good enough to make Q_ψ an acceptable approximation.
This algorithm with linear approximation and quasi-compliance with compatible function approximation is named “Compatible Off-Policy Deterministic Actor-Critic”, COPDAC in short.

Algorithm 9.3 **Compatible Off-Policy Deterministic Actor-Critic**

Require: Policy network step size η

Require: Advantage linear approximator step size ω

Require: Value linear approximator step size ς

Require: Behavior policy β

Require: an empty replay buffer B

Require: Init parameters θ of Policy network μ_{θ_k} with small random values

Require: Init parameters ψ of Advantage linear approximator A_ψ with small random values

Require: Init parameters v of Value linear approximator V_v with small random values

For $k = 0, 1, 2, \dots$ do:

collect trajectories $D_k = \{ \tau_i \}$ using behavior policy β

put all transitions $\langle s, a, s', r \rangle$ of all trajectories D_k in replay buffer B

For $n = 0, 1, 2, \dots, N-1$ do a q-value function gradient descent iteration:

For $m = 0, 1, 2, \dots, M-1$ do:

From replay buffer B sample a transition $\langle s_{m,t}, a_{m,t}, s_{m,t+1}, r_{m,t} \rangle$

$\phi(s_{m,t}, a_{m,t}) = \left(\nabla_\theta \mu_{\theta_k}(s_{m,t})\right)\left(a_{m,t} - \mu_{\theta_k}(s_{m,t})\right)$

$A^\psi(s_{m,t}, a_{m,t}) = \phi(s_{m,t}, a_{m,t})^{\mathrm{T}} \psi$

extract state features $\phi(s_{m,t})$

$V_v(s_{m,t}) = v^{\mathrm{T}} \phi(s_{m,t})$

$Q_{\psi,v}(s_{m,t}, a_{m,t}) = A^\psi(s_{m,t}, a_{m,t}) + V_v(s_{m,t})$

compute $a_{m,t+1} = \mu_{\theta_k}(s_{m,t+1})$

$\phi(s_{m,t+1}, a_{m,t+1}) = \left(\nabla_\theta \mu_{\theta_k}(s_{m,t+1})\right)\left(a_{m,t+1} - \mu_{\theta_k}(s_{m,t+1})\right)$

$A^\psi(s_{m,t+1}, a_{m,t+1}) = \phi(s_{m,t+1}, a_{m,t+1})^{\mathrm{T}} \psi$

extract state features $\phi(s_{m,t+1})$

$V_v(s_{m,t+1}) = v^{\mathrm{T}} \phi(s_{m,t+1})$

$Q_{\psi,v}(s_{m,t+1}, a_{m,t+1}) = A^\psi(s_{m,t+1}, a_{m,t+1}) + V_v(s_{m,t+1})$

$y_{m,t} = r_{m,t} + \gamma \, Q_\psi(s_{m,t+1}, a_{m,t+1})$

$\delta_{m,t} = y_{m,t} - Q_\psi(s_{m,t}, a_{m,t})$

End of for

estimate the advantage and value functions gradients as:

$$\widehat{h_k} = \frac{1}{M}\sum_{m=0}^{M-1} \delta_{m,t}\phi\left(s_{m,t}, a_{m,t}\right)$$

$$\widehat{d_k} = \frac{1}{M}\sum_{m=0}^{M-1} \delta_{m,t}\phi\left(s_{m,t}\right)$$

update advantage and value functions by gradient descent step (or other):

$$\psi \leftarrow \psi + \omega\widehat{h_k}$$

$$v \leftarrow v + \varsigma\widehat{d_k}$$

end of for

estimate policy gradient as:

From replay buffer B sample U transitions (you only need the state s_u)

$$\widehat{g_k} = \frac{1}{U}\sum_{u=0}^{U-1} \nabla_\theta \mu_{\theta_k}(s_u) \cdot \left(\nabla_\theta \mu_{\theta_k}(s_u)\right)^{\mathrm{T}} \psi$$

update the policy network with gradient ascent (or other methods like Adam):

$$\theta_{k+1} \leftarrow \theta_k + \eta\widehat{g_k}$$

end of for

9.6 Deep Deterministic Policy Gradient

Compatible Function Approximator provides a way to have a q-value gradient closer to the true one, but at the expense of having only a linear approximator of the q-value function. If we want to have a richer and more complex capacity of processing the input state, we need to use a large neural network as function approximator for the q-value, but this is known to be unstable, which precisely motivated the usage of COPDAC. [Lillicrap et al. 2016] found an effective way to use big neural networks as function approximators for the q-value, using the following expedients:

- Using a replay buffer (see section 2.3). We already did this in previous DPG algorithms.
- Target Network for Q-Value function (see section 2.5) with Polyak averaging (see section 2.6). This means to use a different Q-Value network to compute target Q-Values, which gets updated slowly to give more stable results.
- Target policy network with Polyak averaging, to be used to compute $a_{t+1} = \mu_\theta(s_{t+1})$ in the target Q-Value computation.

- Batch normalization
- The Behavior policy is the Target policy plus gaussian noise on actions (i.e. is a stochastic gaussian policy, whose average is the deterministic target policy).

Algorithm 9.4 **Deep Deterministic Policy Gradient (Actor-Critic)**

Require: Policy network step size η

Require: Q-Value network step size ω

Require: Behavior policy β

Require: an empty replay buffer B

Require: Initialize parameters θ of network μ_{θ_k} with small random values

Require: Initialize parameters ψ of network Q_ψ with small random values

Require: Polyak Q-Value Step size ς

Require: Polyak Policy Step size ζ

copy Q-Value target network parameters $\psi' \leftarrow \psi$

copy Policy target network parameters $\theta' \leftarrow \theta$

For $k = 0, 1, 2, \ldots$ do:

    collect trajectories $D_k = \{ \tau_i \}$ using behavior policy β

    put all transitions $\langle s, a, s', r \rangle$ of all trajectories in replay buffer B

    For $n = 0, 1, 2, \ldots, N-1$ do a q-value function gradient descent iteration:

        For $m = 0, 1, 2, \ldots, M-1$ do:

            From replay buffer B sample a transition $\langle s_{m,t}, a_{m,t}, s_{m,t+1}, r_{m,t} \rangle$

            compute $a_{m,t+1} = \mu_{\theta'_{k+1}}(s_{m,t+1})$

            $y_{m,t} = r_{m,t} + \gamma\, Q_{\psi'}(s_{m,t+1}, a_{m,t+1})$

        end of for

        estimate the q-value network gradients as:

$$\widehat{h_k} = \frac{1}{M} \sum_{m=0}^{M-1} \left(y_{m,t} - Q_\psi(s_{m,t}, a_{m,t}) \right) \nabla_\psi Q_\psi(s_{m,t}, a_{m,t})$$

        update the q-value network with a gradient descent step (or other method):

            $\psi \leftarrow \psi + \omega \widehat{h_k}$

    update the q-value target network

            $\psi' \leftarrow \varsigma \psi' + (1 - \varsigma)\, \psi$

        estimate policy gradient as:

$$\widehat{g_k} = \frac{1}{M}\sum_{m=0}^{M-1} \nabla_\theta \mu_{\theta_k}(s_{m,t}) \cdot \nabla_a Q_\psi(s_{m,t}, a)\Big|_{a = \mu_{\theta_k}(s_{m,t})}$$

update the policy network with gradient ascent (or other methods like Adam):

$$\theta_{k+1} \leftarrow \theta_k + \eta \widehat{g_k}$$

update the policy target network

$$\theta'_{k+1} \leftarrow \zeta \theta'_k + (1-\zeta)\,\theta_{k+1}$$

end of for

end of for

---

10. Reward Shaping and Curriculum Learning

Reinforcement Learning may be defined as a method to automatically learn from experience without human help. In fact, this is not completely correct. One of the crucial aspects of actual Reinforcement Learning algorithms is the modelling of the rewards: it is up to the human designer to decide when a reward should be given and how high (or low) the reward is, and that must be done in a way that reflects the actual goal that the agent is supposed to learn. Reward modelling can be made in a simple way if the problem has a single, unique goal: a positive reward is given if the goal is reached, no other rewards are given while the goal is not reached. While this seems to make sense, it may work badly: the agent may reach a state very close to goal without fulfilling the goal, so it would not receive any reward for it and it would not learn that the reached state was good and desirable to be reached again. Since RL algorithms “propagate back” the information obtained from a state that obtained reward to the previous states that lead to it, agents are not able to learn when a state is ideally good if there is not a reward deriving from that state or future reached states. So, it is clear that for some tasks, having only a final reward when the goal is reached is not the best way to assign rewards, and the reward system should instead be designed and engineered. This means that the human choices about how to model the rewards may change a lot the performance of the same algorithm.

Designing the reward system is called “Reward Shaping” and it has to be done in a way that “guides” the agent into intermediate subgoals. It must be noted that Reward Shaping is somehow like “cheating”: the more you shape rewards, the more you are inserting human knowledge in a system that was meant to learn automatically, without prior knowledge. That does not mean it is a bad thing, it just means that we have to be aware that we are inserting external knowledge, and potentially any bias or flaw that comes with it.

In more complicated settings, where the goal may not be unique and there are different “good things to do”, there may be rewards for minor goals, and sometimes this may lead to the fact that the agent learns only to solve the minor goals because the obtained rewards distract it from solve the bigger goal (small rewards may be much smaller than big rewards but much easier to get).

Another method used to make agents learn in complicated task is the “curriculum learning”: a simplified version of the task (or the environment) is used at the beginning, and when the agent has learnt to reach the goal in that setting, a progressively more complete version is used.

Curriculum learning may also be intended as making the agent learn a set of certain skills or behaviours that are preparatory for the complete goal.

Also curriculum learning, like reward shaping, is injecting human knowledge into the system, because a human is deciding which simplified task or skill is necessary to be learnt before learning the true task, and that opens the possibility to bias the learning (i.e. without a certain curriculum learning maybe the agent would learn a better policy that is not based on what the curriculum designer thought were the necessary skills).

11. Imitation Learning

At this point it may be due a brief digression about Imitation Learning, a method that, in common with Reinforcement Learning, aims at selecting the best action depending on the state. Differently from RL, Imitation Learning does not use reinforcement signals such as rewards, but rather it is a supervised learning method in which examples reproduce the behaviour of an expert (that usually is a human). The input data X represent the state, and the label Y is the action taken by the expert in that state. Imitation Learning has the valuable characteristic of being able to use directly human knowledge in form of examples. Unfortunately, one of the issues of IL is that training examples are usually created in a limited subset of the state space, and a trained agent may, during his functioning, exit from that subspace, because even small differences in action responses or in starting states may accumulate and lead to very different states. Once the agent is outside the subspace of states in which training examples have been produced (outside the training set distribution), it likely will not be able to generalize the new states, and the actions selected will not be optimal, or may even be disastrous. For this reason, it is important to cover a big part of the state space with training samples, and that could be difficult.

An Imitation Learning system may be built as a neural network, with a number of input neurons equal to the dimension of a sample, a certain number of hidden layers with a variety of possible architectures, then a last layer whose output neurons are the ones that indicate the action to take (so if the actions are discrete and there are K different actions there will be K final neurons with a Softmax applied to them, if the actions are continuous there will be other K output neurons outputting the average of the computed distributions on the continuous values, etc.). In other words, it will be a neural network just like the one that you would build for a policy network in RL, except that this network will be trained as a supervised network with usual (minibatch) gradient descent algorithm, and not with any policy gradient algorithm.

A first thing to note is that the fact that the policy network may have the same architecture both in imitation learning and in policy gradient RL makes possible to train the same network both with IL and RL, so it is theoretically possible to integrate human knowledge with the RL process.

An interesting characteristic of Imitation Learning training is its mathematical relationship with Reinforcement Learning, in particular with policy gradient methods, as we will see it below.

Let us start with the example of discrete actions. To use the same notation of Reinforcement Learning, we call $\pi_\theta(a_t|s_t)$ the Imitation Learning neural network that outputs a probability distribution over actions taking the state as input, and we imagine of having N example trajectories of length T_i each.

Now, the last layer of the network is a Softmax which implies that the gradient of the Loss is the negative of the log of the probability of the correct action. So, it turns out that this Imitation Learning minibatch gradient (of Loss function) will be computed as:

$$\nabla_\theta J_{IL}(\pi_\theta) = -\frac{1}{N}\sum_{i=0}^{N}\sum_{t=0}^{T_i-1} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \tag{11.1}$$

While in case of policy gradient Reinforcement Learning the minibatch gradient (of expected return) is (from eq. 3.9 and eq. 3.29):

$$\nabla_\theta J_{RL}(\pi_\theta) = \frac{1}{N}\sum_{i=0}^{N}\sum_{t=0}^{T_i-1} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\, \Phi_{i,t} \tag{11.2}$$

Remarkably, they are similar (except for the initial minus in IL because in IL we minimize the objective while in RL we maximize the objective), the Reinforcement Learning gradient is just like the Imitation Learning gradient in which each sample gradient is also multiplied by $\Phi_{i,t}$, that is related to the rewards (it may be the rewards-to-go, the Advantage function etc.).

The same happens if the actions are not discrete but continuous, represented by a real number for each action parameter, so the output of the network is not a Softmax operator but real numbers, intended to be the means of a distribution, usually a Gaussian. For simplicity let us deal with the case of just one action parameter. In that configuration the Loss function of the Imitation Learning system would be the mean square error between the network output and the parameter chosen by the expert (from the sample). The RL formula for the log probability of the policy network will again be similar to the one of IL. To see that, for the RL policy network let us call σ^2 the variance of the gaussian of the action parameter and call $\pi_\theta(s_t)$ the output of the policy network, whose value is to be intended as the mean of the gaussian distribution of the parameter of the action. It is necessary to carefully not confuse the policy output $\pi_\theta(s_t)$ with Pi, the transcendental number π that is necessary for the gaussian formula and appearing in the equation. The agent will select the value of action a_t by sampling from a gaussian distribution with mean $\pi_\theta(s_t)$ and variance σ^2 . So, the probability of the action taken by the agent will be the probability of the action a_t considering that it is distributed by that gaussian.

$$\nabla_\theta J_{RL}(\pi_\theta) = \frac{1}{N}\sum_{i=0}^{N}\sum_{t=0}^{T_i-1} \nabla_\theta \log \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(\pi_\theta(s_{i,t}) - a_{i,t})^2/(2\pi\sigma^2)}\, \Phi_{i,t} =$$

$$\frac{1}{N}\sum_{i=0}^{N}\sum_{t=0}^{T_i-1} \nabla_\theta (-\frac{(\pi_\theta(s_{i,t}) - a_{i,t})^2}{2\pi\sigma^2} - \log\sqrt{2\pi\sigma^2})\Phi_{i,t} =$$

$$-\frac{1}{2\pi\sigma^2}\frac{1}{N}\sum_{i=0}^{N}\sum_{t=0}^{T_i-1} \nabla_\theta(\pi_\theta(s_{i,t}) - a_{i,t})^2\Phi_{i,t}$$

(11.3)

Now, $1/(2\pi\sigma^2)$ can be cancelled from the equation because it is just a constant that may be incorporated with the learning rate. The same would apply to the Imitation Learning gradient if we derived it within the maximum log-likelihood framework: a $1/(2\pi\sigma^2)$ constant would appear, and we would incorporate it with the learning rate. So, the gradient for Reinforcement Learning policy is:

$$\nabla_\theta J_{RL}(\pi_\theta) = -\frac{1}{N}\sum_{i=0}^{N}\sum_{t=0}^{T_i-1} \nabla_\theta(\pi_\theta(s_{i,t}) - a_{i,t})^2\Phi_{i,t}$$

(11.4)

While the gradient for the Imitation Learning is given by the usual supervised regression loss gradient, the gradient of the square error:

$$\nabla_\theta J_{IL}(\pi_\theta) = \frac{1}{N}\sum_{i=0}^{N}\sum_{t=0}^{T_i-1} \nabla_\theta(\pi_\theta(s_{i,t}) - a_{i,t})^2$$

(11.5)

Again, the two formulas are almost the same, apart from the minus sign (depending on ascending or descending the gradient) and the multiplication by $\Phi_{i,t}$.

So, intuitively, Reinforcement Learning policy gradient is like an Imitation Learning gradient that also uses the information of the rewards $\Phi_{i,t}$ to inform “how good” or “how bad” the action is.

I think it is surprising to find out that two different optimization problems related to two different underlying tasks end up with such similar computations.

11. Afterword

This has been just an introduction to the theory of Deep Reinforcement Learning. To not make it too boring, some mathematical passages and proofs have been skipped, some others have been simplified, but still some have been necessarily reported in detail, to create a consistent presentation that follows a principled thread. This introduction has focused on “Deep” RL, that means that I have not dealt with tabular methods, even if there has been a great amount of research on them and they cannot be ignored. Even among Deep RL methods, since this is just an introduction, I did not examine all algorithms, but only a didactically representative small subset of them.

The reader who wants the complete math and proofs, as well as the reader that wants to know more about tabular methods or about other deep RL methods, is left to the reference literature.

Also, while algorithms are described in detail, I acknowledge that it may not be easy for the primer to understand exactly how to implement a RL system after reading this introduction. To that purpose I suggest the reader to follow one of the many introductory courses on the internet (there are excellent ones both from MOOCs platforms and from famous brick and mortar universities), that focus on the practical side. A valid help to understand RL algorithms may come from checking the open source implementations of the algorithms available online, some of which are made by the same authors of the reference literature.

A topic that I did not discuss but it is worth a final mention is the fact the Reinforcement Learning is not “sample efficient”, that means that it needs a lot of samples (a lot of experience, or trajectories, or “trial and error”) to learn good policies in complex environments. Often training time may be much longer than expected, for instance longer than (usually) with supervised learning systems. This and other difficulties of Reinforcement Learning practice are well detailed in an article written by [Irpan 2018], which I suggest every RL practitioner to read.

Appendix A: Information Theory Refresh

A.1 Kullback-Leibler Divergence

It is a way to compute how different two distributions are. Given two distributions P and Q, the Kullback-Leibler divergence (also named *Relative Entropy*) of P from Q is defined as:

$$continuous \quad D_{KL}(P||Q) = \int_x P(x) \log \frac{P(x)}{Q(x)}$$

$$discrete \quad D_{KL}(P||Q) = \sum_x P(x) \log \frac{P(x)}{Q(x)} \tag{a.1}$$

Where "$\log$" is the natural logarithm (even if it is possible to compute it also using different bases for the logarithm, such as base 2).

- KL divergence is always non-negative (as a consequence of *Gibb's Inequality*): $D_{KL}(P||Q) \geq 0$.
- When the two distributions are the same, KL divergence is zero $D_{KL}(P||P) = 0$.
- KL divergence is not symmetrical: it is easy to see that in general $D_{KL}(P||Q) \neq D_{KL}(Q||P)$, hence it is not a metric. This is the reason why it is more precise to say "KL divergence of P from Q" than just "KL divergence between P and Q".
- Anyway a symmetric computation may be devised: $Symmetric_{KL} = D_{KL}(P||Q) + D_{KL}(Q||P)$

In Deep Reinforcement Learning, when using policy gradient methods, you may have either discrete (categorical) actions or continuous actions. If actions are categorical you have as many output neurons as the number of possible actions within which to choose. Each output neuron represents the probability of choosing the related action. In this case the distribution of actions is discrete, and the KL divergence may be computed with the formula for discrete distributions (the second formula in eq. a.1), where x is the action.

If instead actions are continuous, the policy neural network outputs a value for each dimension of the action. This output value usually represents the mean of a normal distribution from which it is possible to sample the action to take. The standard deviation of such a normal distribution may be a fixed value (decided ex ante), or it may be another output of the neural network (one for each dimension of the action). So it may be useful to recall an analytical computation of Kullback-Leibler divergence for normal distributions.

For univariate normal distributions (see proof in [Statproofbook 2021A]) where variable x is unidimensional and distributed normally with mean μ and variance σ^2 :

$$P: x \sim \mathcal{N}(\mu_1, \sigma_1^2)$$
$$Q: x \sim \mathcal{N}(\mu_2, \sigma_2^2)$$

$$KL(P||Q) = \frac{1}{2}\left[\frac{(\mu_2 - \mu_1)^2}{\sigma_2^2} + \frac{\sigma_1^2}{\sigma_2^2} - \log\frac{\sigma_1^2}{\sigma_2^2} - 1\right] \tag{a.2}$$

Now let us see the version for multivariate normal distributions (see proof in [Statproofbook 2021B]), using the notation $|\mathbf{M}|$ for the determinant of a matrix $\mathbf{M}$, and the notation $tr(\mathbf{M})$ for the trace of a matrix $\mathbf{M}$. The variable x is a vector of dimension n distributed normally with mean vector μ (analogously of dimension n) and covariance matrix $\mathbf{\Sigma}$ (of dimension $n \times n$).:

$$P: x \sim \mathcal{N}(\mu_1, \mathbf{\Sigma_1})$$
$$Q: x \sim \mathcal{N}(\mu_2, \mathbf{\Sigma_2})$$

$$KL(P||Q) = \frac{1}{2}\left[(\mu_2 - \mu_1)^T \mathbf{\Sigma_2^{-1}} (\mu_2 - \mu_1) + tr(\mathbf{\Sigma_2^{-1}\Sigma_1}) - \log\frac{|\mathbf{\Sigma_1}|}{|\mathbf{\Sigma_2}|} - n\right] \tag{a.3}$$

When the two covariance matrices $\mathbf{\Sigma_1}$ and $\mathbf{\Sigma_2}$ are diagonal, i.e. the covariances between variables are zero, the computation of eq. a.3 is equivalent to compute eq. a.2 for each variable and sum all the results together. In Deep Reinforcement Learning usually the covariances between action dimensions are supposed to be zero, so this simplification may be applied (either the variances for each action are fixed, with covariances equal to zero, or the variances are outputted by an output neuron of the policy neural network that outputs only a standard deviation -or a log-standard deviation- for each action dimensions and no covariances).

A.2 Total Variation Distance

It is another way to compute how different two distributions are. Given two distributions P and Q, the Total Variation distance $D_{TV}(P||Q)$ is defined as:

$$continuous \quad D_{TV}(P||Q) = \frac{1}{2}\int_x |P(x) - Q(x)|$$

$$discrete \quad D_{TV}(P||Q) = \frac{1}{2}\sum_x |P(x) - Q(x)|$$

(a.4)

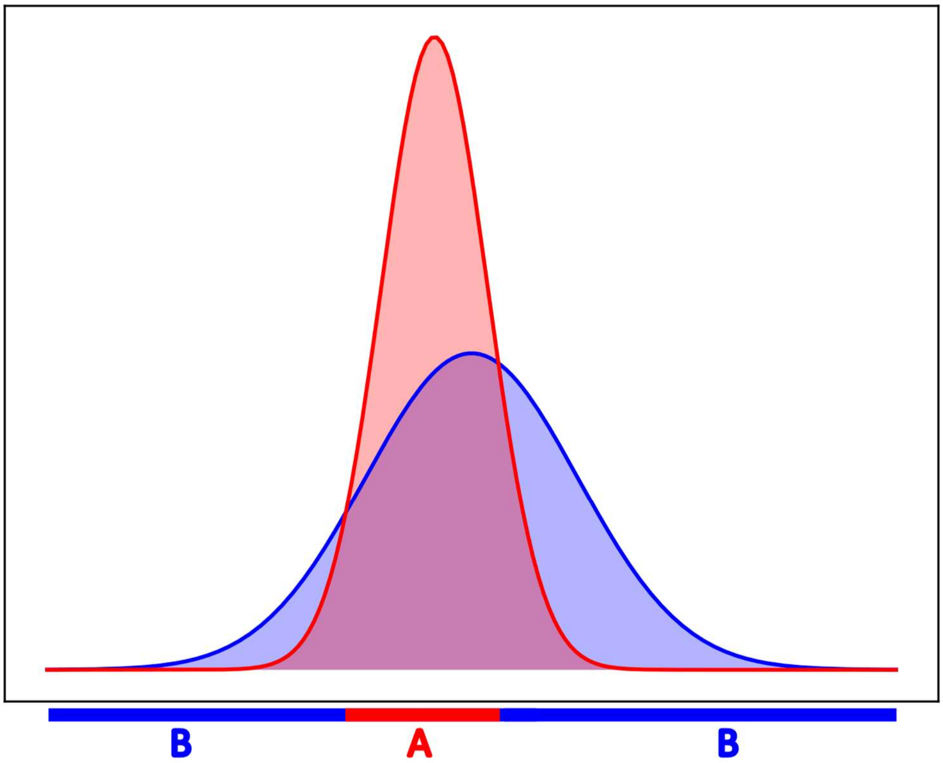

Figure a.1

Given two distributions P and Q , here depicted in their probability density functions, we notice that by definition the Total Variation Distance is equal to the sum of all areas of P above the area of Q , plus the sum of all areas of Q above the area of P, all divided by 2. That is, in the figure, the sum of the red-only area and the two blue-only areas, excluding the purple area, all divided by two.

If you watch Figure a.1 you can see that the Total Variation Distance is equal to the sum of all areas in which either distribution overwhelms the other (without the overwhelmed areas), divided by two. This is by definition.

But, since the area of every distribution is always equal to one, if a distribution overwhelms the other for a certain area, it must be overwhelmed for an equivalent area in a different subset of the domain. This means that the Total Variation Distance is also equal to the area of any of the distributions that is overwhelming the other, without summing the other, and without dividing by two.

Hence, since the Total Variation Distance may be computed measuring only the area of a distribution that overwhelms the area of the other distribution, another definition is possible. Watching Figure a.1 if we name the red distribution P and the blue distribution Q , we can see that where the red distribution is above the blue one, we named the support as A (the subset of the domain where $P > Q$), and where the blue distribution is above the red one we named the support as B (the subset of the domain where $Q > P$).
We just noticed that the Total Variation Distance may be computed measuring the area where the probability density function of a distribution is greater than the pdf of the other.
Hence, another definition of Total Variation distance may be based on finding the subset A where $P > Q$, and then computing the difference between the probabilities of P and Q on that subset.

That is equivalent to say that we want to find the subset A that maximizes the difference in probabilities between P and Q. If we want to use statistical language, we may call the domain of P as "*sample space*", hence the subset A is an "*event*", and any $x \in A$ is an "*outcome*" included in that event. So, to compute the Total Variation distance we find the event for which the difference between the probabilities of the two distributions is greater, and then compute that difference.

$$D_{TV}(P||Q) = \max_{A \subseteq S}\left(Prob_P(A) - Prob_Q(A)\right)$$

$$with\ S\ support\ of\ P$$

(a.5)

That is equivalent to:

$$continuous \quad D_{TV}(P||Q) = \frac{max}{A \subseteq S}\left(\int_{x \in A} P(x) - \int_{x \in A} Q(x)\right)$$

$$discrete \quad D_{TV}(P||Q) = \frac{max}{A \subseteq S}\left(\sum_{x \in A} P(x) - \sum_{x \in A} Q(x)\right)$$

$$with\ S\ support\ of\ P$$

(a.6)

Those can be equivalently written as:

$$continuous \quad D_{TV}(P||Q) = \frac{max}{A \subseteq S}\left(\int_{x \in A} (P(x) - Q(x))\right)$$

$$discrete \quad D_{TV}(P||Q) = \frac{max}{A \subseteq S}\left(\sum_{x \in A} (P(x) - Q(x))\right)$$

$$with\ S\ support\ of\ P$$

(a.7)

The Total Variation distance has the following relationship with Kullback-Leibler divergence (see [Pollard 2000] Ch.3):

$$D_{TV}(P||Q)^2 \leq D_{KL}(P||Q)$$

(a.8)

A.3 Score (of the Log-Likelihood)

It is the gradient of the log-likelihood function with respect to the parameters vector. Hence, given a certain parameters vector, the score denotes the steepness of the log-likelihood at that point in parameters space, or in other terms how much the log-likelihood would change for an infinitesimal change in the parameters.
Naming θ the parameters vector:

$$s(\theta) = \frac{\partial \log \mathcal{L}(\theta)}{\partial \theta}$$

(a.9)

Since the log-likelihood is a function of the samples X , we can make that explicit:

$$s(\theta; x) = \frac{\partial \log \mathcal{L}(\theta; x)}{\partial \theta}$$

(a.10)

If θ are the true parameters of the distribution $P_\theta(x)$ of samples X, the expectation of the score with respect of the distribution of X is the zero vector:

$$E_{x \sim P_\theta(x)}[s(\theta; x)] = \mathbf{0}$$

(a.11)

To show it:

$$E_{x \sim P_\theta(x)}[s(\theta; x)] = \int_x P_\theta(x) \frac{\partial \log \mathcal{L}(\theta; x)}{\partial \theta}$$

$$since\ \theta\ are\ the\ true\ parameters,$$

$$P_\theta(x)\ is\ \mathcal{L}(\theta; x)\ .\ Substitute\ it, and\ \ apply\ log - derivative\ trick\ (eq.\ 3.3)$$

$$= \int_x P_\theta(x) \frac{1}{P_\theta(x)} \frac{\partial P_\theta(x)}{\partial \theta}$$

$$= \int_x \frac{\partial P_\theta(x)}{\partial \theta}$$

$$under\ regularity\ conditions\ it\ is\ possible\ to\ interchange\ the\ derivative\ and\ the\ integral$$

$$(Leibniz\ Integral\ Rule)$$

$$= \frac{\partial}{\partial \theta} \int_x P_\theta(x) = \frac{\partial}{\partial \theta} 1\ = 0$$

■

(a.12)

A.4 Fisher Information Matrix

The Fisher Information is the variance of the score with respect to the distribution of the samples. If the parameters θ of the distribution are more than one, the Fisher Information is a matrix of covariances of the elements of the score vector, and we name it $\boldsymbol{F_{P_\theta}}$.
Assuming that θ are the true parameters of the distribution of X :

$$\boldsymbol{F_{P_\theta}} = Var_{x\sim P_\theta(x)}[s(\theta;x)]$$

$$= E_{x\sim P_\theta(x)}\left[\left(s(\theta;x) - E_{x\sim P_\theta(x)}[s(\theta;x)]\right) \left(s(\theta;x) - E_{x\sim P_\theta(x)}[s(\theta;x)]\right)^{\mathrm{T}} \right]$$

$$from\ equation\ a.11\ we\ know\ that\ \ E_{x\sim P_\theta(x)}[s(\theta;x)] = \mathbf{0}\ \ ,hence:$$

$$\boldsymbol{F_{P_\theta}} = E_{x\sim P_\theta(x)}[s(\theta;x)\ s(\theta;x)^{\mathrm{T}}]$$

(a.13)

That may also be written as (recall that $s(\theta;x) = \frac{\partial \log \mathcal{L}(\theta;x)}{\partial\theta} = \nabla_\theta \log P_\theta(x)$ by definition):

$$\boldsymbol{F_{P_\theta}} = E_{x\sim P_\theta(x)}[\nabla_\theta \log P_\theta(x)\ \ \nabla_\theta \log P_\theta(x)^{\mathrm{T}}]$$

(a.14)

Hence, each matrix element $\boldsymbol{F_{P_\theta}}[i,j]$ has the following form:

$$\boldsymbol{F_{P_\theta}}[i,j] = E_{x\sim P_\theta(x)}\left[\left(\frac{\partial}{\partial\theta_i}\log P_\theta(x)\right)\left(\frac{\partial}{\partial\theta_j}\log P_\theta(x)\right)\right]$$

(a.15)

It is clearly a square symmetric matrix. It is also positive semidefinite.
It becomes evident that such a matrix may be estimated by samples x_i which follow the P_θ distribution :

$$\widehat{\boldsymbol{F_{P_\theta}}} = \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta \log P_\theta(x_i)\ \ \nabla_\theta \log P_\theta(x_i)^{\mathrm{T}}$$

(a.16)

(That is actually what we do when approximating the Fisher Information Matrix for the Natural Policy Gradient in Ch.6, equation 6.3, where instead of P_θ there is the policy function π_θ .).

The Fisher information matrix is always symmetric and positive semi-definite [Watanabe 2009].

Intuitively and informally speaking, the Fisher Information Matrix measures how much information the random variable x carries about any parameter in θ.

A.5 Fisher Information Matrix equivalence to negative expectation of Hessian Matrix of log-probability

Now, for the next part we need to use the concept of the Hessian Matrix. Recall that the Hessian is the square matrix of second order partial derivatives of a scalar function (shortly, the Hessian is the Jacobian of the gradient). In our case the Hessian would be with respect to the parameters θ of a probability function $P_\theta(x)$, that is $\frac{\partial^2 P_\theta(x)}{\partial_i \partial_j}$. To simplify reading I will use the symbol $H_{P_\theta(x)}$ for it, while for the Jacobian instead I will use the symbol $\mathbf{J}()$.

If the log-likelihood function is twice differentiable with respect to θ, and under certain regularity conditions it can be shown that the Fisher Information Matrix is equivalent to the negative expected value of the Hessian matrix of the log-likelihood.

$$\boldsymbol{F_{P_\theta}} = -E_{x \sim P_\theta(x)}\left[H_{\log P_\theta(x)}\right] \tag{a.17}$$

So, each matrix element $F_{P_\theta}[i,j]$ is:

$$F_{P_\theta}[i,j] = -E_{x \sim P_\theta(x)}\left[\left(\frac{\partial^2}{\partial \theta_i \theta_j} \log P_\theta(x)\right)\right] \tag{a.18}$$

To show it let us start from the Hessian of the log-likelihood, following [Kristiadi 2018]:

$$H_{\log P_\theta(x)} = \mathbf{J}(\nabla_\theta \log P_\theta(x))$$

$$= \mathbf{J}\left(\frac{\nabla_\theta P_\theta(x)}{P_\theta(x)}\right)$$

$$\textit{applying the quotient rule for derivatives}$$

$$= \frac{H_{P_\theta(x)} P_\theta(x) - \nabla_\theta P_\theta(x) \nabla_\theta P_\theta(x)^{\mathrm{T}}}{P_\theta(x) P_\theta(x)}$$

$$= \frac{H_{P_\theta(x)}}{P_\theta(x)} - \frac{\nabla_\theta P_\theta(x)}{P_\theta(x)} \frac{\nabla_\theta P_\theta(x)^{\mathrm{T}}}{P_\theta(x)}$$

$$\textit{let us take the expectation with respect to } x \textit{ distributed by } P_\theta(x)$$

$$E_{x \sim P_\theta(x)}\left[H_{\log P_\theta(x)}\right] = E_{x \sim P_\theta(x)}\left[\frac{H_{P_\theta(x)}}{P_\theta(x)} - \frac{\nabla_\theta P_\theta(x)}{P_\theta(x)} \frac{\nabla_\theta P_\theta(x)^{\mathrm{T}}}{P_\theta(x)}\right]$$

$$= \int_x \frac{H_{P_\theta(x)}}{P_\theta(x)} P_\theta(x) - E_{x \sim P_\theta(x)}\left[\frac{\nabla_\theta P_\theta(x)}{P_\theta(x)} \frac{\nabla_\theta P_\theta(x)^{\mathrm{T}}}{P_\theta(x)}\right]$$

$$\textit{under regularity conditions it is possible to interchange the derivative and the integral}$$

$$(\textit{Leibniz Integral Rule})$$

$$= H_{\int_x P_\theta(x)} - E_{x \sim P_\theta(x)}\left[\frac{\nabla_\theta P_\theta(x)}{P_\theta(x)} \frac{\nabla_\theta P_\theta(x)^{\mathrm{T}}}{P_\theta(x)}\right]$$

$$\textit{the integral of a distribuion is} = 1 \textit{ and its derivative is} = 0, \qquad \textit{hence } H_{\int_x P_\theta(x)} = \mathbf{0}$$

$$= -E_{x \sim P_\theta(x)}\left[\frac{\nabla_\theta P_\theta(x)}{P_\theta(x)} \frac{\nabla_\theta P_\theta(x)^{\mathrm{T}}}{P_\theta(x)}\right]$$

$$= -E_{x \sim P_\theta(x)}[\nabla_\theta \log P_\theta(x) \; \nabla_\theta \log P_\theta(x)^{\mathrm{T}}]$$

$$= -\boldsymbol{F}_{\boldsymbol{P_\theta}}$$

■

(a.19)

Since the Hessian Matrix is commonly used to study the curvature of a function at a critical point, the equivalence above tells us that the Fisher Information Matrix contains information about the curvature of the expectation of the log-likelihood.

A.6 Kullback-Leibler divergence approximation by second order Taylor expansion using Fisher Information Matrix

The equivalence of the Fisher Information Matrix with the negative expected value of the Hessian matrix of the log-likelihood allows to compute an approximation of the Kullback-Leibler divergence between two distributions where one is the "perturbed" version of the other (i.e. they share the same parametric form, and the parameters vector of the second distribution are obtained adding a small vector to the parameters of the first), using second order Taylor expansion for a part of the KL divergence formula.
Recall that second order Taylor series expansion for a function $f(\theta)$ is an approximation used to evaluate the function $f(\theta)$ around a certain point θ_0, at the point $\theta_0 + \delta$:

$$f(\theta_0 + \delta\ ) \approx f(\theta_0) + \nabla_\theta f(\theta_0)^{\mathrm{T}}\delta + \frac{1}{2}\delta^{\mathrm{T}}(\nabla_\theta^2 f(\theta_0))\delta \tag{a.20}$$

Consider having the distributions $P_\theta(x) = f(x;\theta)$ and $Q_{\theta,\delta}(x) = f(x;\theta+\delta)$, that means that both share the same function f, and the parameters vector of P is θ , while the parameters vector of Q is $(\theta+\delta)$, with small δ. In other terms, Q is a perturbed version of P . For instance in Reinforcement Learning P may be the old policy function and Q the new policy function obtained by an optimization step on P.
I follow the proof by [Ratliff 2013]:

$$D_{KL}(P||Q) = D_{KL}(P_\theta(x)||Q_{\theta,\delta}(x)) = D_{KL}(f(x;\theta)||f(x;\theta+\delta)) =$$

$$= \int_x f(x;\theta)\log\frac{f(x;\theta)}{f(x;\theta+\delta)}$$

$$= \int_x f(x;\theta)\log f(x;\theta) - \int_x f(x;\theta)\log f(x;\theta+\delta) \tag{a.21}$$

Now we apply the second order Taylor expansion only to $\log f(x;\theta+\delta)$:

$$\log f(x;\theta+\delta) \approx \log f(x;\theta) + \nabla_\theta \log f(x;\theta)^{\mathrm{T}}\delta + \frac{1}{2}\delta^{\mathrm{T}}\left(\nabla_\theta^2 \log f(x;\theta)\right)\delta$$

$$= \log f(x;\theta) + \left(\frac{\nabla_\theta f(x;\theta)}{f(x;\theta)}\right)^{\mathrm{T}}\delta + \frac{1}{2}\delta^{\mathrm{T}}(\nabla_\theta^2 \log f(x;\theta))\delta$$

(a.22)

Now we plug a.22 into a.21:

$$D_{KL}(f(x;\theta)||f(x;\theta+\delta)) \approx$$

$$\approx \int_x f(x;\theta)\log f(x;\theta) - \int_x f(x;\theta)\left(\log f(x;\theta) + \left(\frac{\nabla_\theta f(x;\theta)}{f(x;\theta)}\right)^{\mathrm{T}}\delta + \frac{1}{2}\delta^{\mathrm{T}}(\nabla_\theta^2 \log f(x;\theta))\delta\right)$$

$$= \int_x f(x;\theta)\log\frac{f(x;\theta)}{f(x;\theta)} - \left(\int_x \nabla_\theta f(x;\theta)\right)^{\mathrm{T}}\delta - \frac{1}{2}\delta^{\mathrm{T}}\left(\int_x f(x;\theta)\nabla_\theta^2 \log f(x;\theta)\right)\delta$$

(a.23)

Now: $\int_x f(x;\theta)\log\frac{f(x;\theta)}{f(x;\theta)}$ is equal to 0 (easy to do the math or to see it as a KL divergence of two equal distributions).

Also $\left(\int_x \nabla_\theta f(x;\theta)\right)^{\mathrm{T}}$ is equal to 0: it is possible to swap the gradient and derivative signs (under regularities) to obtain $\left(\nabla_\theta \int_x f(x;\theta)\right)^{\mathrm{T}}$.

The integral of any distribution is 1, so $\nabla_\theta 1 = 0$.

So it turns out:

$$D_{KL}(f(x;\theta)||f(x;\theta+\delta)) \approx -\frac{1}{2}\delta^{\mathrm{T}}\left(\int_x f(x;\theta)\nabla_\theta^2 \log f(x;\theta)\right)\delta$$

$$= -\frac{1}{2}\delta^{\mathrm{T}}\left(\int_x f(x;\theta)H_{\log f(x;\theta)}\right)\delta$$

$$= -\frac{1}{2}\delta^{\mathrm{T}}E_{x\sim f(x;\theta)}\left[H_{\log f(x;\theta)}\right]\delta$$

(a.24)

Now, we already know that the expected value of the Hessian matrix of the log-likelihood is equal to the negative of Fisher Information Matrix (eq. a.17). So we substitute it:

$$D_{KL}(f(x;\theta)||f(x;\theta+\delta)) \approx \frac{1}{2}\delta^{\mathrm{T}}\boldsymbol{F}_{\boldsymbol{P_\theta}}\delta$$

(a.25)

A.7 Relationship between the Hessian of Kullback-Leibler divergence and Fisher Information Matrix

Again, consider having two distributions with the same function but different parameters $P_\theta(x)$ and $P_{\theta'}(x)$.

Then the gradient of Kullback-Leibler divergence between P_θ and $P_{\theta'}$ with respect to θ' would be:

$$\nabla_{\theta'} D_{KL}(P_\theta||P_{\theta'}) = \nabla_{\theta'}\int_x P_\theta(x)\log\frac{P_\theta(x)}{P_{\theta'}(x)}$$

(a.26)

That gradient is a vector of partial derivatives with respect to all parameters of θ'. To denote the partial derivative with respect to parameter j of parametrization θ' I use the symbol $\partial_{\theta'j}d$. We have that:

$$\partial_{\theta'j}d = \partial_{\theta'j}\int_x P_\theta(x)\log\frac{P_\theta(x)}{P_{\theta'}(x)}$$

$$\textit{under regularity conditions the derivative and integral symbol can be exchanged}$$

$$= \int_x P_\theta(x)\,\partial_{\theta'j}\left(\log\frac{P_\theta(x)}{P_{\theta'}(x)}\right)$$

$$= \int_x P_\theta(x)\frac{P_{\theta'}(x)}{P_\theta(x)}P_\theta(x)\frac{-1}{P_{\theta'}(x)^2}\partial_{\theta'j}P_{\theta'}(x)$$

$$= -\int_x \frac{P_\theta(x)}{P_{\theta'}(x)}\partial_{\theta'j}P_{\theta'}(x)$$

(a.27)

Now let us take the derivative of eq. a.27, that is the second derivative of the KL divergence (computing at each parameter i, j): $\partial_{\theta' i}\partial_{\theta' j} d$.

$$\partial_{\theta' i}\partial_{\theta' j} d = \partial_{\theta' i} \int_x -\frac{P_\theta(x)}{P_{\theta'}(x)} \partial_{\theta' j} P_{\theta'}(x)$$

$$under\ regularity\ conditions\ the\ derivative\ and\ integral\ symbol\ can\ be\ exchanged$$

$$= \int_x -P_\theta(x) \partial_{\theta' i} \left(\frac{\partial_{\theta' j} P_{\theta'}(x)}{P_{\theta'}(x)} \right)$$

$$= \int_x - P_\theta(x) \left(\frac{\partial_{\theta' i}\partial_{\theta' j} P_{\theta'}(x) P_{\theta'}(x) - (\partial_{\theta' i} P_{\theta'}(x))(\partial_{\theta' j} P_{\theta'}(x))}{P_{\theta'}(x)^2} \right)$$

$$= \int_x P_\theta(x) \left(\frac{(\partial_{\theta' i} P_{\theta'}(x))(\partial_{\theta' j} P_{\theta'}(x))}{P_{\theta'}(x)^2} - \frac{\partial_{\theta' i}\partial_{\theta' j} P_{\theta'}(x)}{P_{\theta'}(x)} \right)$$

$$now, if\ we\ evaluate\ that\ integral\ at\ the\ parameter\ point\ \theta' = \theta\ we\ obtain:$$

$$\partial_{\theta' i}\partial_{\theta' j} d \Big|_{\theta' = \theta} = \int_x P_\theta(x) \left(\frac{(\partial_{\theta' i} P_{\theta'}(x))(\partial_{\theta' j} P_{\theta'}(x))}{P_{\theta'}(x)^2} - \frac{\partial_{\theta' i}\partial_{\theta' j} P_{\theta'}(x)}{P_{\theta'}(x)} \right) \Big|_{\theta' = \theta}$$

$$= \int_x \frac{(\partial_{\theta' i} P_{\theta'}(x))(\partial_{\theta' j} P_{\theta'}(x))}{P_{\theta'}(x)} - \partial_{\theta' i}\partial_{\theta' j} P_{\theta'}(x) \Big|_{\theta' = \theta}$$

$$it\ is\ easy\ to\ see\ that\ \int_x \partial_{\theta' i}\partial_{\theta' j} P_{\theta'}(x) = \partial_{\theta' i}\partial_{\theta' j} \int_x P_{\theta'}(x) = \partial_{\theta' i}\partial_{\theta' j} 1 = 0$$

$$= \int_x \frac{(\partial_{\theta' i} P_{\theta'}(x))(\partial_{\theta' j} P_{\theta'}(x))}{P_{\theta'}(x)} \Big|_{\theta' = \theta}$$

$$= \int_x \frac{P_{\theta'}(x)}{P_{\theta'}(x)} \frac{(\partial_{\theta' i} P_{\theta'}(x))(\partial_{\theta' j} P_{\theta'}(x))}{P_{\theta'}(x)} \Big|_{\theta' = \theta}$$

$$= \int_x P_{\theta'}(x)\ (\partial_{\theta' i} \log P_{\theta'}(x))(\ \partial_{\theta' j} \log P_{\theta'}(x)) \Big|_{\theta' = \theta}$$

$$= E_{x \sim P_\theta(x)} \left[(\partial_{\theta i} \log P_\theta(x)) \big(\partial_{\theta j} \log P_\theta(x) \big) \right]$$

$$= \boldsymbol{F}_{\boldsymbol{P}_{\boldsymbol{\theta}}}[i,j] \quad (see\ eq.a.8)$$

$$\Rightarrow H(D_{KL}(P_\theta||P_{\theta'}))_{\theta'}\big|_{\theta' = \theta} = \boldsymbol{F}_{\boldsymbol{P}_{\boldsymbol{\theta}}}$$

(a.28)

That means that given two distributions of the same family $P_\theta(x)$ and $P_{\theta'}(x)$, the Hessian with respect to θ' of Kullback-Leibler divergence from P_θ to $P_{\theta'}$, evaluated at $\theta' = \theta$ is equal to the Fisher Information Matrix of $P_\theta(x)$.

Appendix B: Conjugate Gradient Algorithm

This Appendix provides a sketch of the Conjugate Gradient Algorithm [Hestenes and Stiefel 1952]. It will not be described in full detail, just what it is enough to understand its usage in Reinforcement Learning (as in Trust Region Policy Optimization). The reader is encouraged to find more on the topic in literature, starting for instance from the guides consulted to prepare this appendix like [Shewchuk 1994] and [Refsnæs 2009].

B.1 Aim

Conjugate Gradient is an algorithm that is typically used to solve linear systems such as:

$$\mathbf{A}x = b \tag{b.1}$$

Where $\mathbf{A}$ is a matrix, b is a (column) vector, and x is the unknown (column) vector.
This problem may be solved also by other methods such as Gaussian Elimination, LU Decomposition, or Cholesky Decomposition, but when the matrix $\mathbf{A}$ is big, those methods are slow and demand much memory. Conjugate Gradient allows to find an (approximate) solution in a faster way and with less memory usage, especially when the matrix $\mathbf{A}$ is sparse. You do not even need to know the full matrix $\mathbf{A}$ if somehow you manage to have a function that, given a vector v, returns the matrix-vector product $\mathbf{A} \cdot v$.
To use Conjugate Gradient, the matrix $\mathbf{A}$ must be symmetrical and positive definite (the algorithm is able to converge also in case of symmetrical and positive semidefinite matrices [Hayami 2018]).
To have a quick refresh of algebra: a symmetric matrix $\mathbf{A}$ of dimension $n \times n$ is positive definite if and only if:

$$v^T \cdot \mathbf{A} \cdot v > 0$$
$$for\ every\ column\ vector\ v \neq 0\ in\ \mathbb{R}^n \tag{b.2}$$

An alternative definition is: a square matrix is positive definite if it is symmetric and all its eigenvalues are positive.
Property: a positive definite matrix is invertible, and its determinant is positive.

B.2 The Equivalent Problem

When the matrix $\mathbf{A}$ is symmetrical and positive definite, finding the solution for $\mathbf{A}x = b$ is equivalent to minimize the function:

$$f(x) = \frac{1}{2}x^T \mathbf{A}\, x - x^T b \tag{b.3}$$

The equivalence of the two problems is evident noting that one condition to find the minimum x^* of $f(x)$ is that is that it must be a critical point, which means that the gradient of $f(x)$ with respect to x must be the zero vector:

$$\nabla_x f(x^*) = \mathbf{A}\, x^* - b = 0 \tag{b.4}$$

That is equivalent to eq. b.1 .
To see that this critical point is also a minimum it is sufficient to note that the Hessian of $f(x)$ has all positive eigenvalues, hence it is a “concave up” function, that implies that a critical point must be the minimum. In fact the Hessian is:

$$\nabla_{x_i x_j} f(x^*) = \mathbf{A} \tag{b.5}$$

We know by assumption that $\mathbf{A}$ is positive definite, that implies that all its eigenvalues are positive.
Therefore, we just proved that the vector x^* that minimizes eq. b.3 is also the solution to the linear system of eq. b.1.
The Conjugate Gradient algorithm allows us to solve the minimization of function b.3 through a particular flavour of steepest descent algorithm, and that can be used whenever we want to solve a linear system in which the matrix is symmetrical positive definite.

B.3 The Steepest Descent

The Conjugate Gradient algorithm can be seen as a special kind of steepest descent. Let us see first thing a standard steepest descent algorithm and then we will see how the Conjugate Gradient improves on that.

In the steepest descent algorithm, we begin assigning a starting value for the vector x, and we iteratively add another vector to it until it reaches the solution x^*, or a close approximation of it. In formulas, starting with initial iteration $k = 0$ and an arbitrary vector $x_n \in \mathbb{R}^n$:

$$x_{k+1} = x_k + \alpha_k p_k \tag{b.6}$$

where a_k is a scalar and $p_k \in \mathbb{R}^n$ is a direction vector. The initial vector x_0 may be set to the zero vector if we do not know any starting point who is allegedly closer to the solution than the zero vector.

The issue here is to find the right a_k and p_k such that x_{k+1} improves the objective compared to x_k . Since we are minimizing $f(x)$, that means:

$$f(x_{k+1}) < f(x_k) \tag{b.7}$$

We do not want just that, we would also like the improvement to be the biggest that can be done, so to do as few iterations as possible.

If you already know methods like gradient descent (and you should, as a prerequisite to study Deep Reinforcement Learning), you are aware that the gradient of the function $f(x)$ is the direction of greatest increase of $f(x)$, hence the negative gradient is the direction of steepest descent. So we have found the direction, and we also already know that the gradient of $f(x)$ is $\mathbf{A}\,x - b$.

$$p_k = -(\mathbf{A}\,x_k - b) \tag{b.8}$$

When the problem is solved we have $\mathbf{A}\,x = b$, when the problem is not yet solved $\mathbf{A}\,x$ is a vector different from b. The difference between b and this vector is called *residual*, and indicated with r_k :

$$r_k = b - \mathbf{A}\,x_k = p_k \tag{b.9}$$

That implies:

$$x_{k+1} = x_k - \alpha_k \nabla_x f(x_k) = x_k - \alpha_k (\mathbf{A}\, x_k - b\ ) = x_k + \alpha_k r_k$$

(b.10)

Now we need to find the scalar α_k. To do that, we plug x_{k+1} of eq. b.6 into $f(x)$ of eq. b.3 and we minimize for α_k.

Doing so, we obtain:

$$f(x_k + \alpha_k p_k) = \frac{1}{2}(x_k + \alpha_k p_k)^T \mathbf{A}\,(x_k + \alpha_k p_k) - (x_k + \alpha_k p_k)^T b$$

$$= \frac{1}{2} {p_k}^T \mathbf{A}\, p_k {\alpha_k}^2 + \alpha_k {p_k}^T (\mathbf{A}\, x_k - b) + \cdots$$

$$\textit{where the three dots "..." represent other terms that do not contain } \alpha_k$$

(b.11)

So, we obtained a quadratic at α_k , that is concave up, so we just need to set its first derivative to zero to minimize it:

$${p_k}^T \mathbf{A}\, p_k \alpha_k + {p_k}^T (\mathbf{A}\, x_k - b) = 0$$

$$\alpha_k = -\frac{{p_k}^T (\mathbf{A}\, x_k - b)}{{p_k}^T \mathbf{A}\, p_k} = \frac{{p_k}^T r_k}{{p_k}^T \mathbf{A}\, p_k}$$

(b.12)

This is a general formula. In steepest gradient we have also that $p_k = r_k$, so we have:

$$\alpha_k = \frac{{r_k}^T r_k}{{r_k}^T \mathbf{A}\, r_k}$$

(b.13)

Doing this for enough iterations will converge to the solution.

One issue with this method is that it often produces a jagged path toward the solution (such as in fig. 6.2, but possibly even more jagged since in steepest descent two consecutive directions are orthogonal) that means that it may not be the fastest. The Conjugate Gradient algorithm, while following a similar iterative increment strategy, adopts some principled improvements that usually allows us to reach the solution quicker.

B.4 The Conjugate Gradient

Now we need to introduce the concept of "*conjugate vectors*".

Given a positive definite $n \times n$ matrix $\mathbf{A}$, two vectors u and $v \in \mathbb{R}^n$ are said to be mutually $\mathbf{A} - \text{conjugate}$ if and only if:

$$u^T \cdot \mathbf{A} \cdot v = 0$$

(b.15)

If u and v are $\mathbf{A} - \text{conjugate}$, then any two other vectors respectively parallel to u and v are also $\mathbf{A} - \text{conjugate}$: the direction matters, not the length. For this reason it is sometimes used the term "*conjugate directions*" rather than "*conjugate vectors*". Also, sometimes it is used the term $\mathbf{A} - \text{orthogonal}$ instead of $\mathbf{A} - \text{conjugate}$, that is motivated by the fact that u is orthogonal to the vector resulting from the multiplication $\mathbf{A} \cdot v$ (their dot product equals 0, by definition, and this implies orthogonality).

Property: any set of mutually $\mathbf{A} - \text{conjugate}$ vectors is linearly independent.

So, if we have a set of n mutually $\mathbf{A} - \text{conjugate}$ vectors $H = \{h_0, h_1, \dots, h_{n-1}\}$, that set is also a basis for $\mathbb{R}^n$. This allows us to express any vector as a linear combination of that basis, for instance we can express the difference between the solution x^* and the initial point x_0 as a linear combination:

$$x^* - x_0 = \gamma_0 h_0 + \gamma_1 h_1 + \cdots + \gamma_{n-1} h_{n-1}$$

(b.16)

With $\gamma_0 \dots \gamma_{n-1}$ scalars.

This may be rewritten as:

$$x^* = x_0 + \gamma_0 h_0 + \gamma_1 h_1 + \cdots + \gamma_{n-1} h_{n-1}$$

(b.17)

Written in this way it is noticeable that, starting from x_0 , the solution x^* can be reached in an iterative way summing a vector at each iteration, exactly like we were doing in steepest descent, but this time the vectors are forming a basis so we know that we need just n iterations (if we know the basis and the scalars). If we compare eq. b.17 with eq. b.6 we have γ_k in place of α_k and h_k in place of p_k . So, we can rewrite the formula replacing them (but keep note that they now will be computed in a different manner than in steepest descent):

$$x^* = x_0 + \alpha_0 p_0 + \alpha_1 p_1 + \cdots + \alpha_{n-1} p_{n-1}$$

(b.18)

At this point we need to find the basis p_k and the coefficients α_k . One way to find a basis could be to compute the eigenvectors, but that would be computationally expensive.
We introduced the concept of conjugate vectors because they have a useful property: each new $\mathbf{A}-\mathrm{conjugate}$ vector p_k can be calculated using the previous $\mathbf{A}-\mathrm{conjugate}$ vector p_{k-1} and the residual r_k , without needing any other previous $\mathbf{A}-\mathrm{conjugate}$ vector. The whole derivation of this property and its consequences would be very long so we will just write down the formulas here, and the reader may find proofs and theory in [Shewchuk 1994].
Remembering that $r_k = b - \mathbf{A}\, x_k$, we have:

$$p_k = r_k + \beta_k p_{k-1}$$

(b.19)

Since r_k is the gradient of original function (b.3) to be minimized, and we are using conjugate vectors, the algorithm has been named “Conjugate Gradient”.
Because at the beginning of the computations we do not have any already existing vector to conjugate with, the formula b.15 does not apply to p_0 (clearly p_{-1} does not exists): to compute p_0 we just compute it as in steepest descent:

$$p_0 = r_0$$

(b.20)

Because p_k is now computed differently than in steepest descent, also α_k is different:

$$\alpha_k = \frac{{r_k}^T r_k}{{p_k}^T \mathbf{A}\, p_k}$$

(b.21)

Now we need to compute β_k , using the condition of $\mathbf{A}-\mathrm{conjugacy}$, and again the reader may find the detailed explanation on it in [Shewchuk 1994]:

$$\beta_k = \frac{{r_k}^T \mathbf{A}\, p_{k-1}}{{p_{k-1}}^T \mathbf{A}\, p_{k-1}} = \frac{{r_k}^T r_k}{{r_{k-1}}^T\, r_{k-1}}$$

(b.22)

Moreover, please note that:

$$r_{k+1} = b - \mathbf{A}\, x_{k+1} = b - \mathbf{A}\, x_k - \alpha_k \mathbf{A}\, p_k = r_k - \alpha_k \mathbf{A} p_k$$

(b.23)

Wrapping up, we may now write the algorithm:

Algorithm b.1 **Conjugate Gradient**

Require: $n \times n$ symmetric positive definite matrix $\mathbf{A}$

Require: b , a (column) vector of size n

Require: x_0 , a (column) vector of size n (optional, if absent assign the zero vector)

$r_0 = b - \mathbf{A}x_0$

$p_0 = r_0$

Do for $k = 0,1,\dots$ until (or close to) convergence:

$$\alpha_k = \frac{{r_k}^T r_k}{{p_k}^T \mathbf{A}\, p_k}$$

$$x_{k+1} = x_k + \alpha_k p_k$$

$$r_{k+1} = r_k - \alpha_k \mathbf{A}\, p_k$$

$$\beta_k = \frac{{r_{k+1}}^T r_{k+1}}{{r_k}^T\, r_k}$$

$$p_{k+1} = r_{k+1} + \beta_k p_k$$

end of do

One notable aspect of the algorithm is that the matrix $\mathbf{A}$ is used only to do one thing: multiply vectors of size n. So, hypothetically it is possible to use the Conjugate Gradient algorithm even if we do not know the full matrix $\mathbf{A}$, as long as we have the availability of a function that receives a vector v of size n as input and returns the multiplication $\mathbf{A} \cdot v$.

To speed up a little the algorithm it is easy to see in the inner loop that the matrix multiplication $\mathbf{A}\, p_k$ is computed twice, so it is possible to compute it once and store in a variable for reuse, and the same can be said of ${r_k}^T r_k$.

Appendix C: Importance Sampling

If we have a random variable X with probability density function $P(x)$, and we have a function of X named $g(x)$ (which is itself a random variable too, being a function of a random variable), we can compute the expectation of $g(x)$ through an integral:

$$E_{x\sim P(x)}[\ g(x)\ ] = \int_{-\infty}^{\infty} g(x)\ P(x)\ dx$$

(c.1)

If for some reason we cannot compute the integral or if it would be too expensive to do it, we can approximate it by sampling. We need the possibility to observe a process where X is distributed following $P(x)$ and then it is processed by $g(x)$, where for each sample x_i we can observe what is the resulting value of $g(x_i)$.
In that way we can compute the sample average of $g(x)$:

$$\bar{g}(x) = \frac{1}{N}\sum_{i=1}^{N} g(x_i)$$

$$with\ x_i \sim P(x)$$

(c.2)

Now, imagine that there is not any process to observe where X is distributed following $P(x)$. So, we cannot sample $g(x_i)$ and compute the average as in c.2. But suppose that we can instead observe a process where X is distributed following a known, different, probability density function $Q(x)$, and we can observe the corresponding samples x_i and the result of $g(x_i)$. Can we use that process to compute an approximation for the expected value $E_{x\sim P(x)}[\ g(x)\ ]$? Sure we can ! We just need to weigh each $g(x_i)$ by the ratio of $P(x_i)$ to $Q(x_i)$, to balance the fact that any sample may occur with different probability in $P(x_i)$ than in $Q(x_i)$. Intuitively, samples that would occur more frequently in $P(x_i)$ than in $Q(x_i)$ should be taken more in account than samples that would occur more rarely in $P(x_i)$ than in $Q(x_i)$. So, the ratio $P(x_i)/\ Q(x_i)$ is like representing the *importance* of the sample, hence the name “importance sampling”.

$$\bar{\bar{g}}(x) = \frac{1}{N}\sum_{i=1}^{N}\frac{P(x_i)}{Q(x_i)}g(x_i)$$

$$with\ x_i \sim Q(x)$$

(c.3)

A mathematical proof to justify that can be derived from c.1:

$$E_{x\sim P(x)}[\ g(x)\ ] = \int_{-\infty}^{\infty} g(x)\ P(x)\ dx$$

$$since\ \frac{Q(x)}{Q(x)}\ is\ equal\ to\ 1, we\ can\ insert\ it\ into\ the\ integral\ as\ a\ multiplicative\ factor:$$

$$= \int_{-\infty}^{\infty} g(x)\ P(x)\frac{Q(x)}{Q(x)}\ dx$$

$$= \int_{-\infty}^{\infty} \frac{P(x)}{Q(x)}g(x)\ Q(x)\ dx$$

$$= E_{x\sim Q(x)}\left[\frac{P(x)}{Q(x)}g(x)\right]$$

(c.4)

Now, if you sample from c.4 you obtain the equation c.3 .

There are some caveats with importance sampling: since the sampling distribution $Q(x)$ is different from the target distribution $P(x)$, importance sampling is theoretically guaranteed to give the right result only with an infinite number of samples, because it is unbiased but may have a big variance. In practice, with limited samples, if the two distributions differ much you could end up not having samples for values of x that have a great probability density function $P(x)$ but are sampled rarely because they have a low probability density function $Q(x)$. This will distort the result. The opposite is also bad: if an x with very low $Q(x)$ and very high $P(x)$ is sampled by chance, the ratio $P(x)/Q(x)$ could be a number very big and distort the computation of $\bar{\bar{g}}(x)$.

References

[Amari 1998] Amari, S. “Natural gradient works efficiently in learning.” Neural Computation, 10, 251. (1998).

[Bagnell and Schneider 2003] Bagnell, J., & Schneider, J. "Covariant policy search." In Proceedings of the international joint conference on artificial intelligence (pp. 1019–1024). (2003).

[Baird 1993] Baird, L. C. "Advantage Updating." Wright Lab. 1993, Technical Report WL-TR-93-1l46.

[Bellman 1952] Bellman, R "On the Theory of Dynamic Programming". Proc Natl Acad Sci U S A. 38 (8): 716–9 , August 1952. doi:10.1073/pnas.38.8.716. PMC 1063639. PMID 16589166.

[Degris et al. 2012] Degris, T., White, M., and Sutton, R. S. "Linear off-policy actor-critic." In 29th International Conference
on Machine Learning.

[Glorot and Bengio 2010] Glorot, X., & Bengio, Y. “Understanding the difficulty of training deep feedforward neural networks. “, In Proceedings of the thirteenth international conference on artificial intelligence and statistics (pp. 249-256). JMLR Workshop and Conference Proceedings. (2010)

[Hafner and Riedmiller 2011] Roland Hafner, Martin Riedmiller
"Reinforcement learning in feedback control", Mach Learn (2011) 84:137–169, DOI 10.1007/s10994-011-5235-x
https://link.springer.com/content/pdf/10.1007/s10994-011-5235-x.pdf

[Hayami 2018] Ken Hayami, "Convergence of the Conjugate Gradient Method on Singular Systems", https://arxiv.org/abs/1809.00793

[Hestenes and Stiefel 1952] Hestenes, Magnus R.; Stiefel, Eduard. "Methods of Conjugate Gradients for Solving Linear Systems" (PDF). Journal of Research of the National Bureau of Standards. 49 (6): 409. (December 1952) doi:10.6028/jres.049.044.
http://nvlpubs.nist.gov/nistpubs/jres/049/6/V49.N06.A08.pdf

[Irpan 2018] Alex Irpan, “Deep Reinforcement Learning Doesn't Work Yet”, (2018)
https://www.alexirpan.com/2018/02/14/rl-hard.html

[Kakade 2002] Kakade Sham, “A natural policy gradient”, Advances in Neural Information Processing Systems, pp. 1057–1063. MIT Press, 2002.

[Kakade and Langford 2002] Kakade, Sham and Langford, John. Approximately optimal approximate reinforcement learning. In ICML, volume 2, pp. 267–274, 2002.

[Kristiadi 2018] Kristiadi Augustinus,
https://agustinus.kristia.de/techblog/2018/03/11/fisher-information/

[Levine 2021a] Sergey Levine, CS 285 at UC Berkeley, Deep Reinforcement Learning, Lecture 9 (part 4)
http://rail.eecs.berkeley.edu/deeprlcourse-fa21/static/slides/lec-9.pdf

https://www.youtube.com/watch?v=QWnpF0FaKL4

[Levine 2021b] Sergey Levine, CS 285 at UC Berkeley, Deep Reinforcement Learning, Lecture 15 (part 2)
https://rail.eecs.berkeley.edu/deeprlcourse/static/slides/lec-15.pdf
https://www.youtube.com/watch?v=9HrN6nHoxD8

[Levine 2021c] Sergey Levine, CS 285 at UC Berkeley, Deep Reinforcement Learning, Lecture 5
https://rail.eecs.berkeley.edu/deeprlcourse/static/slides/lec-5.pdf
https://www.youtube.com/watch?v=KZd508qGFt0

[Levine and Koltun 2013] Sergey Levine and Vladlen Koltun, "Guided Policy Search", ICML 2013, PMLR 28(3):1-9, 2013.
http://proceedings.mlr.press/v28/levine13.pdf

[Levin et al. 2009] Levin, D. A., Peres, Y., and Wilmer, E. L. Markov chains and mixing times. American Mathematical Society, 2009.

[Levine et al. 2020] Sergey Levine, Aviral Kumar, George Tucker, Justin Fu, “Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems”
https://arxiv.org/abs/2005.01643

[Lillicrap et al. 2016] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver & Daan Wierstra
"Continuous Control with Deep Reinforcement Learning"
https://arxiv.org/abs/1509.02971

[Liu et al. 2019] Yao Liu, Adith Swaminathan, Alekh Agarwal, Emma Brunskill
"Off-Policy Policy Gradient with State Distribution Correction"
https://arxiv.org/abs/1904.08473

[Mnih et al. 2013] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. , “Playing Atari with Deep Reinforcement Learning.” (2013), arXiv:1312.5602.

[OpenAi 2018A] OpenAi
Vanilla policy gradient and Expected Grad-Log-Prog Lemma:
https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html

[OpenAi 2018B] OpenAi
Reward-to-go proof
https://spinningup.openai.com/en/latest/spinningup/extra_pg_proof1.html

[OpenAi 2018C] OpenAI
Q-function in policy gradient proof:
https://spinningup.openai.com/en/latest/spinningup/extra_pg_proof2.html

[OpenAi 2018D] OpenAI
TRPO: Matrix-vector product Hx
https://spinningup.openai.com/en/latest/algorithms/trpo.html

[Peters 2007] Peters, J. “Machine learning of motor skills for robotics. “Ph.D. thesis University of Southern California, Los Angeles, CA, 90089, USA. (2007).

[Peters and Schaal 2008] Jan Peters, Stefan Schaal , “Reinforcement learning of motor skills with policy gradients.” , Neural networks, 21(4), 682-697. (2008)

[Peters et al. 2003] Peters, J., Vijayakumar, S., & Schaal, S. "Reinforcement learning for humanoid robotics." In Proceedings of the IEEE-RAS international conference on humanoid robots (HUMANOIDS) (pp. 103–123).(2003)

[Pollard 2000] Pollard, David. Asymptopia: an exposition of statistical asymptotic theory. 2000. URL http://www.stat.yale.edu/˜pollard/Books/Asymptopia.

[Ratliff 2013] Ratliff, Nathan. "Information Geometry and Natural Gradients" in Mathematics for Intelligent Systems.
https://www.nathanratliff.com/pedagogy/mathematics-for-intelligent-systems
https://drive.google.com/file/d/1jDM9yZI1KrtH3JJznPE7S1QzNZZB2xs2/view
https://web.archive.org/web/20200924055016/https://ipvs.informatik.uni-stuttgart.de/mlr/wp-content/uploads/2015/01/mathematics_for_intelligent_systems_lecture12_notes_I.pdf

[Refsnæs 2009] Runar Heggelien Refsnæs, "A Brief Introduction to the Conjugate Gradient Method", 2009, https://www.math.hkust.edu.hk/~mamu/courses/531/cg.pdf

[Saxe et al. 2013] Saxe, A. M., McClelland, J. L., & Ganguli, S. ,”Exact solutions to the nonlinear dynamics of learning in deep linear neural networks.” ICLR (2013), arXiv preprint arXiv:1312.6120.

[Schulman et al. 2015] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, Pieter Abbeel , “Trust Region Policy Optimization”, In International conference on machine learning (pp. 1889-1897). PMLR, arXiv:1502.05477

[Schulman et al. 2016] John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan and Pieter Abbeel, “High-Dimensional Continuous Control Using Generalized Advantage Estimation”, ICLR 2016, arXiv:1506.02438

[Schulman et al. 2017] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). , “Proximal policy optimization algorithms” , arXiv preprint arXiv:1707.06347.

[Shewchuk 1994] Jonathan Richard Shewchuk, "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain" (1994)
https://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf

[Silver et al. 2014] Silver D., Lever G., Heess N., Degris T., Wierstra D., & Riedmiller M. “Deterministic policy gradient algorithms”. In International conference on machine learning (pp. 387-395). PMLR. (2014, January)
https://proceedings.mlr.press/v32/silver14.pdf
https://proceedings.mlr.press/v32/silver14-supp.pdf

[Soemers 2019] Dennis Soemers, “Reward-to-go proof”
https://ai.stackexchange.com/questions/9614/why-does-the-reward-to-go-trick-in-policy-gradient-methods-work/10369

[Statproofbook 2021A] "Proof: Kullback-Leibler divergence for the normal distribution", The Book of Statistical Proofs https://dx.doi.org/10.5281/zenodo.5820411
https://statproofbook.github.io/P/norm-kl.html

[Statproofbook 2021B] "Proof: Kullback-Leibler divergence for the multivariate normal distribution", The Book of Statistical Proofs https://dx.doi.org/10.5281/zenodo.5820411 https://statproofbook.github.io/P/mvn-kl.html

[Sutton & Barto 2018] Richard C. Sutton, Andrew C. Barto, “Reinforcement Learning - An introduction. 2nd Ed.”, MIT Press - ISBN: 9780262039246
http://incompleteideas.net/book/RLbook2020.pdf

[Sutton et al. 1999] Sutton, R. S., McAllester, D. A., Singh, S. P., and Mansour, Y. (1999). "Policy gradient methods for reinforcement learning with function approximation." In Neural Information Processing Systems 12, pages 1057–1063.

[Szepesvári 2010] Csaba Szepesvári, “Algorithms for Reinforcement Learning”, Morgan & Claypool 2010, ISBN: 9781608454921

[van Hasselt 2010] Hado van Hasselt, “Double Q-learning.”
Advances in neural information processing systems, 23, 2613-2621. (2010)

[van Hasselt et al. 2016] Van Hasselt, Hado, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning." Proceedings of the AAAI conference on artificial intelligence. Vol. 30. No. 1. 2016

[Watanabe 2009] Watanabe Sumio, "Algebraic Geometry and Statistical Learning Theory", Cambridge University Press, 2009. ISBN-13 978-0-521-86467-1

[Watkins 1989] Christopher J. C. H. Watkins, “Learning from delayed rewards.”, PhD Thesis, King’s College (1989).

[Williams 1992] Williams, R. J. , “Simple statistical gradient-following algorithms for connectionist reinforcement learning.”, Machine learning, 8(3), 229-256. (1992)