Ruggero Rossi
r.rossi@opencomplexity.com

A Gentle Principled Introduction to Deep Reinforcement Learning

Contents

version 1.0.3

0. Introduction

This paper 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.

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 !). 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 may be stochastic (it is considered generally stochastic, with the deterministic model being a special version). 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) because the task to be executed by the agent is to solve a game, such as Chess or Go, and the rules and mechanics of the game are known and may be taken in account by the algorithm. 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 "Bell Equation" [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), there is an infinite number of states, and 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 hypothetically takes the current state (or current observation) as input and evaluates the 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 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, 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.

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 possible 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
- 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:

$$a_t \sim \pi(\cdot\,|s_t)$$

Equation 1

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

$$\tau = (\,s_0, a_0, s_1, a_1, \dots)$$

Equation 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 $\gamma < 1$:

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

Equation 3

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

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

Equation 4

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

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

Equation 5

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

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

Equation 6

Where π^* is the optimal policy.

The Value Function for a given policy π is:

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

Equation 7

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

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

Equation 8

It has to be noted that

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

Equation 9

As anticipated before, 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:

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

Equation 10

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

Equation 11

Some algorithms use the “Advantage Function”, 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:

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

Equation 12

3. Variety of Methods

Among RL methods, a first distinction may be 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.

A second 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 (the reward and transition functions, and the initial state distribution: R, P, ρ_0) is not available, proper value functions 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 third 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.

4. Q-Learning

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

In Q-Learning the policy followed by agents during training sometimes chooses the action with the greater 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 that would always choose the best action, and never a random one. This is expressed with the formula:

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

Equation 13

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 13 and 11.
Hence the Q-Value is referred to a tentatively optimal policy, that is asymptotically better or equal than the training policy. This makes Q-Learning formally an off-policy method because it evaluates a different policy with respect to the one used in learning.

In the Q-Learning algorithm the true Q^* function is unknown, and the program 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 .

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

Equation 14

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, 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 training 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 that our estimate of the Q function respects the Bellman Equation 14, 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 :

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

Equation 15

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

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

Equation 16

Where η is the learning rate.
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 :

$$L = (\, y - Q_\psi(s,a) \,)^2$$

Equation 17

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 17 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:

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

Equation 18

Then, the algorithm continues in the same way: 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.

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 training policy could be even a policy very different from the optimal policy, we are not obliged to use an ϵ-greedy policy to train. That means that all past trajectories may be retained and reused in future optimization iterations. 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 on the same distribution used at training time (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 one the same distribution that the q-network would experience if using the tentatively optimal/greedy policy. To have similar distribution of states it is necessary to use in training a policy that is similar to the one to be improved, that motivates the usage of an ϵ-greedy policy and motivates the practice of re-using the old samples only for a certain number of steps and then discard them.

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 behaviour 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.

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: 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" that is a copy of the q-network but it is not updated at every gradient descent step, and hence it is more stable.

Algorithm 1 **Deep Q-Learning Algorithm**

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

do K times:

sample a batch 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):

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

$$\psi \leftarrow \psi + \eta \; (y - Q_\psi(s'_i, a'_i) \;) \nabla_\psi Q_\psi(s'_i, a'_i)$$

end of for each

end of do

end of do

end of do

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). This can be mitigated with "*Double Q-Learning*" [van Hasselt 2010]: using two different action-value function estimates, one to check the action with the maximum value and the other to actually obtain the Q-value, swapping the roles of the estimators at each step.

It has to be noted that the Q-Learning algorithm 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 have 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. A better way to do Reinforcement Learning with continuous actions is to use Policy Based methods, that are detailed in the following chapter.

5. Policy Based Methods

Policy based methods, also called "Policy Gradient methods" and "Policy Optimization methods" (those two expressions have slightly 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 returns 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 :

$$a_t \sim \pi_\theta(\cdot\,|s_t)$$

Equation 19

That is like equation 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 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 gaussian usually works well).

The problem of Reinforcement Learning is still the same, we want to maximize the expected return $J(\pi)$, (detailed in equation 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:

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

Equation 20

With η learning rate.

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

6. Policy Gradient

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 the latter updates the policy parameters at each step while the former updates parameters using minibatches of steps (some use the term "vanilla policy gradient" as a family of algorithms, with REINFORCE being one of them, see [Peters and Schaal 2008]). It is an on-policy algorithm.

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 $f(x)$. Applying that to the trajectory distribution, and rearranging, we obtain the so-called "*log derivative trick*":

$$\nabla_\theta P(\tau|\theta) = P(\tau|\theta)\nabla_\theta \log P(\tau|\theta)$$

Equation 21

Now, taking the definition of the probability of a trajectory τ made of T steps from equation 4, we compute the log of it:

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

Equation 22

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:

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

Equation 23

Now, computing the gradient of equation 5 we obtain:

$$\nabla_\theta J(\pi_\theta) = \nabla_\theta E_{\tau\sim\pi_\theta}[G(\tau)]$$

$$= \nabla_\theta \int_\tau P(\tau|\theta) G(\tau)$$

$$= \int_\tau \nabla_\theta P(\tau|\theta) G(\tau)$$

Equation 24

And applying equation 21 (log derivative trick):

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

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

Equation 25

Applying equation 23 :

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

Equation 26

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:

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

Equation 27

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

$$\theta \leftarrow \theta + \eta\Delta\theta$$

Equation 28

A notable fact is that the gradient of the log probability of each action $a_{i,t}$ is multiplied by the complete return $G(\tau_i)$. 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:

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

Equation 29

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

$$\nabla_\theta \int_x f_\theta(x) = \nabla_\theta 1 = 0$$

Equation 30

Now, applying the “log derivative trick” equation 21 :

$$0 = \nabla_\theta \int_x f_\theta(x)$$

$$= \int_x \nabla_\theta f_\theta(x)$$

$$= \int_x f_\theta(x)\, \nabla_\theta \log f_\theta(x)$$

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

Equation 31

(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 29 , 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:

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

Equation 32

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

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

Equation 33

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

Equation 34

We can then decompose equation 26 in the same way:

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

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

Equation 35

Now, this can be rewritten as:

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

Equation 36

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

Equation 37

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

Equation 38

Now, because of EGLP lemma:

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

Equation 39

Now, from 36 we have:

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

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

Equation 40

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:

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

Equation 41

Now we plug equation 39 in 41

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

$$\therefore part1 = 0$$

Equation 42

So we obtain:

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

Equation 43

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

Equation 44

This permits 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). 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 state value.
For instance, in infinite-horizon, a k-step bootstrap would be:

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

Equation 45

$\widehat{G_n}(\tau_t)$ 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:

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

Equation 46

In this way the finite and infinite horizon are included in the same framework.
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 43).

So, more generally:

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

Equation 47

With Φ_t that may be one of:

$$G(\tau_t)$$

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

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

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

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

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

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

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

Equation 48

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 $V^{\pi_\theta}(s_t)$.

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 12 .

The advantage function 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 equation 15 and 16 , 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:

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

$$or$$

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

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

$$\psi \leftarrow \psi + \eta\ \ (\ y - V_\psi(s)\ )\nabla_\psi V_\psi(s)$$

Equation 49

Policy gradient methods that use a value function to bootstrap the estimate of the total return such in equation 45 and 46 are called "**actor-critic**", where actor is referred to the policy function and critic to the value function [Sutton and Barto 2018]. Some authors call "actor-critic" every policy method that uses a 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.

Algorithm 2 **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 $G(\tau_t)$

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:

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

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

As it is evident from the 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 returns (equation 47) 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 and cannot be used again to improve the next policy $\pi_{\theta'}$ (the new policy resulted from the optimization step). 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 allows to use each trajectory more than once is the Proximal Policy Optimization, described in Ch. 9 .

A remarkable aspect of policy gradient methods is that compared to value based methods they have a more "natural" distribution on 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 bigger 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.

7. Generalized Advantage Estimation

[Schulman et al. 2016] introduced a particular version of the Advantage Function, the “Generalized Advantage Estimation”, with proven variance reduction property. The setting is 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 downweight 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:

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

Equation 50

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

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

Equation 51

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. :

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

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

...

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

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

Equation 52

In infinite horizon, k may tend to ∞. In that case the $\gamma^k V(s_{t+k})$ at each time will be canceled by the discounted $-V(s_t)$ of the next time and we would have:

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

Equation 53

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 $\sum_{k=0}^{\infty} \lambda^k$ is equal to $1/(1-\lambda)$, so $(1-\lambda)\sum_{k=0}^{\infty} \lambda^k = 1$. So, a summation of infinite terms, each weighted by $(1-\lambda)\lambda^k$, with $\lambda \in [0,1]$ and k from 0 to ∞, has the total sum of weights = 1.
The "Generalized Advantage Function" hence is:

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

Equation 54

And a more practical formulation may be obtained:

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

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

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

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

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

Equation 55

This GAE hence can be used instead of the Advantage Function and put in place of Φ_t in the policy gradient (see equation 47) . This permits to tune variance and bias with two parameters: γ controls how far in time to consider the rewards relevant, and λ that decides how much the return component of the advantage has to be similar to a 1-step bootstrap (if λ=0) or progressively to a Monte Carlo return (if λ=1).

8. Natural Policy Gradient

To improve the speed of training one right thing to do can be to scale the gradient in a way that gradient descent 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 [Kakade 2002].

In fact, some parameters of the neural network affect the policy more than others: when we are doing gradient descent 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.

One way to do that is consists of putting a constraint to each gradient descent step, such that the new policy and the old policy are not too different as distributions. An appropriate computation for that could be the Kullbach-Leibler divergence, enforcing the constraint $D_{KL}(\pi_{\theta'}||\pi_\theta) < \epsilon$.
The Kullbach-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:

$$D_{KL}(\pi_{\theta'}||\pi_\theta) \approx \frac{1}{2}(\theta' - \theta)\boldsymbol{F}_{\boldsymbol{\pi_\theta}}(\theta' - \theta)$$

Equation 56

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 (and the score is the gradient of log-likelihood with respect to the parameters). Hence:

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

Equation 57

That can be approximated with samples from the trajectory.

Having the Fisher-information matrix, it is easy to compute the correct learning rate for the update such that the Kullbach-Leibler divergence is constrained.

The common gradient ascent step for policy gradient is:

$$\theta \leftarrow \theta + \eta \nabla_\theta J(\pi_\theta)$$

Equation 58

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

$$\theta' \leftarrow \arg max_\theta \ \nabla_\theta J(\pi_\theta)^T(\theta' - \theta)$$

$$such\ that \left|\left|\theta' - \theta\right|\right| < \epsilon$$

Equation 59

That would imply:

$$\theta' \leftarrow \theta + \sqrt{\frac{\epsilon}{|| \nabla_\theta J(\pi_\theta)^T||^2}} \nabla_\theta J(\pi_\theta)$$

Equation 60

But as I wrote before, instead of imposing a constraint over $\left|\left|\theta' - \theta\right|\right|$ 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 56, and given that we can compute $\boldsymbol{F_{\pi_\theta}}$ as in equation 57 from samples, such a constraint in the KL divergence can be imposed in a gradient descent step with the following update [Peters and Schaal 2008]:

$$\theta \leftarrow \theta + \eta \boldsymbol{F_{\pi\theta}^{-1}} \nabla_\theta J(\pi_\theta)$$

Equation 61

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

9. Proximal Policy Optimization

In the Proximal Policy Optimization an alternative optimization problem is posed, such that allows to do more than one policy gradient ascent iteration in each optimization round using the same set of trajectories.

The Reinforcement Learning problem here is framed in a different (but equivalent) formulation. In the classical 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 π_θ, 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 possible to show that

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

Equation 62

That 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 are distributed by π_θ, , so $A^{\pi_\theta}(s_t, a_t)$ 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. 62 . 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 θ .

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

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 parametrized by θ , as:

$$\xi_\theta(s) = P(s_0 = s|\,\theta) + \gamma P(s_1 = s|\,\theta) + \gamma^2 P(s_1 = s|\,\theta) + \cdots$$

Equation 63

We can write eq. 62 in a way to sum over time, then over states, and then over actions, and in the last row we plug eq. 63, referred to policy θ' , into it:

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

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

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

Equation 64

Now, the right-hand side seems in a more manageable form to be maximized, 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:

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

Equation 65

This approximation can be used only when the new distribution $\pi_{\theta'}(a_t|s_t)$ is not too different from the old distribution $\pi_\theta(a_t|s_t)$, so a constraint may be enforced for that, computing the difference with the Kullback-Leibler divergence: $D_{KL}(\pi_\theta||\pi_{\theta'}) < \epsilon$

We need also to express this formula as an expectation to sample it.
The sum over the frequencies $\sum_s \xi_\theta(s)$ can be replaced by the expectation $\frac{1}{1-\gamma} E_{s\sim\xi_\theta(s)}$.
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:

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

$$s.t. \quad E_{s\sim\xi_\theta(s)}[D_{KL}(\pi_\theta(\cdot\,|s)||\pi_{\theta'}(\cdot\,|s))] < \epsilon$$

Equation 66

The constraint on the KL divergence may also be inserted as a penalty:

$$L(\theta, \theta') = E_{s\sim\xi_\theta(s)}\left[E_{a\sim\pi_\theta(a|s)}\left[\frac{\pi_{\theta'}(a|s)}{\pi_\theta(a|s)}A^{\pi_\theta}(s,a)\right] - \beta D_{KL}(\pi_\theta(\cdot\,|s)||\pi_{\theta'}(\cdot\,|s))\right]$$

Equation 67

For some coefficient β.
But it is hard to find the right β, so [Schulman et al. 2017] in their Proximal Policy optimization suggest using clipping as a way to bound the difference in distribution. 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^{CLIP} may be devised:

$$L^{CLIP}(\theta, \theta') = E_{\substack{s\sim\xi_\theta(s),\\ a\sim\pi_\theta(a|s)}}[\,\min(\,d(\theta)A^{\pi_\theta}(s,a)\,, clip(\,d(\theta), 1-\epsilon, 1+\epsilon\,)A^{\pi_\theta}(s,a)\,)\quad]$$

Equation 69

written in a simplified way:

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

Equation 70

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.

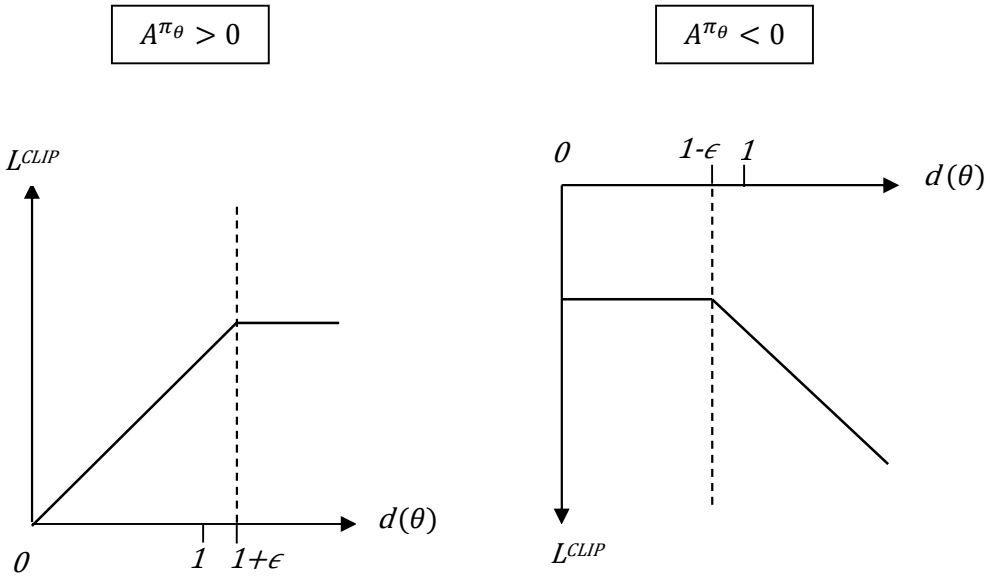

Figure 1. Surrogate objective with policy ratio clipping in Proximal Policy Optimization

To practically use it, we have to do gradient descent with respect to θ' on equation 69 or 70 , 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 may be useful to check the Kullback-Leibler divergence between the probabilities of the taken actions under the old policy and under the new policy, and stop the policy gradient ascent iterations in case it 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):

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

Equation 71

Algorithm 3 **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 $G(\tau_t)$

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:

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

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

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

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

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

end of If

estimate policy gradient as:

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

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:

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

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

PPO is considered on-policy because it aims at optimizing the same policy function that generates the training 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.

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.

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.

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's 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:

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

Equation 72

While in case of Reinforcement Learning the minibatch gradient (of return expectation) is:

$$\nabla_\theta J_{RL}(\pi_\theta) = \frac{1}{N}\sum_{i=0}^{N}\sum_{t=0}^{T_i-1} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\, \Phi_{i,t}$$

Equation 73

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.

$$\nabla_\theta J_{RL}(\pi_\theta) = \frac{1}{N}\sum_{i=0}^{N}\sum_{t=0}^{T_i-1} \nabla_\theta \log \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(\pi_\theta(s_{i,t})-a_{i,t})^2/(2\pi\sigma^2)}\, \Phi_{i,t} =$$

$$\frac{1}{N}\sum_{i=0}^{N}\sum_{t=0}^{T_i-1} \nabla_\theta (-\frac{(\pi_\theta(s_{i,t})-a_{i,t})^2}{2\pi\sigma^2} - \log\sqrt{2\pi\sigma^2})\Phi_{i,t} =$$

$$-\frac{1}{2\pi\sigma^2}\frac{1}{N}\sum_{i=0}^{N}\sum_{t=0}^{T_i-1} \nabla_\theta(\pi_\theta(s_{i,t}) - a_{i,t})^2 \Phi_{i,t}$$

Equation 74

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:

$$\nabla_\theta J_{RL}(\pi_\theta) = -\frac{1}{N}\sum_{i=0}^{N}\sum_{t=0}^{T_i-1}\nabla_\theta(\pi_\theta(s_{i,t}) - a_{i,t})^2\Phi_{i,t}$$

Equation 75

While the gradient for the Imitation Learning is given by the usual supervised regression loss gradient, the gradient of the square error:

$$\nabla_\theta J_{IL}(\pi_\theta) = \frac{1}{N}\sum_{i=0}^{N}\sum_{t=0}^{T_i-1}\nabla_\theta(\pi_\theta(s_{i,t}) - a_{i,t})^2$$

Equation 76

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.

12. 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.

References

[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)

[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.

[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

[Peters and Schaal 2008] Jan Peters, Stefan Schaal , “Reinforcement learning of motor skills with policy gradients.” , Neural networks, 21(4), 682-697. (2008)

[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.

[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

[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

[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)

[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)