IFT-7014: Directed reading

Deep Reinforcement Learning - Soft Actor-Critic

v:1.0

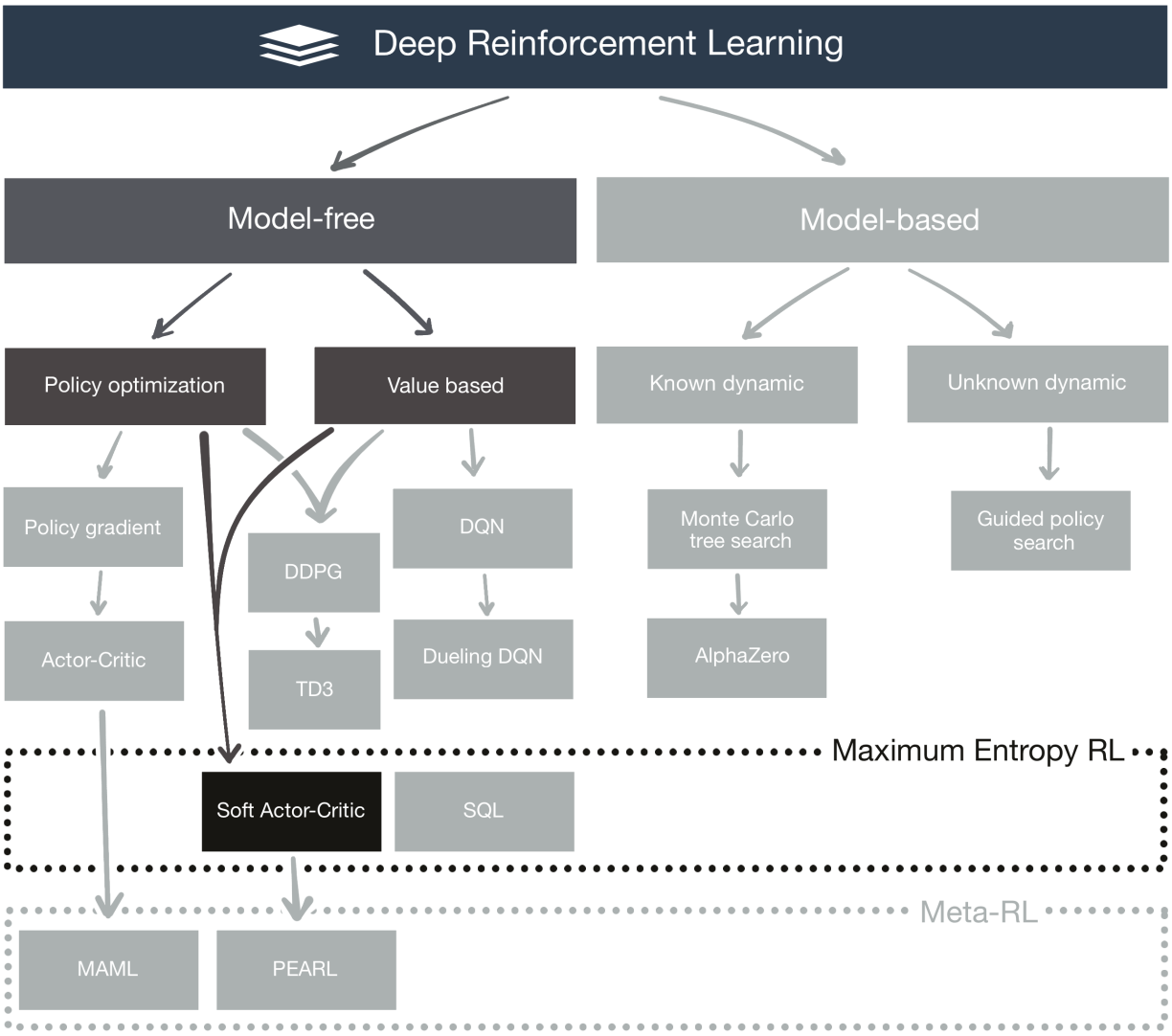
Luc Coupal
Université Laval
Montréal, QC, Canada
Luc.Coupal.1@uLaval.ca

Under the supervision of:

Professor **Brahim Chaib-draa**
Directeur du programme de baccalauréat en génie logiciel de l'Université Laval
Québec, QC, Canada
Brahim.Chaib-draa@ift.ulaval.ca

January 26, 2020

CONTENTS



Soft Actor-Critic

Soft Actor-Critic (*SAC*) is an off-policy algorithm based on the *Maximum Entropy Reinforcement Learning* framework. The main idea behind *Maximum Entropy RL* is to frame the decision-making problem as a graphical model from top to bottom and then solve it using tools borrowed from the field of *Probabilistic Graphical Model*. Under this framework, a learning agent seeks to maximize both the return and the entropy simultaneously. This approach benefit *Deep Reinforcement Learning* algorithm by giving them the capacity to consider and learn many alternate paths leading to an optimal goal and the capacity to learn how to act optimally despite adverse circumstances.

Since *SAC* is an off-policy algorithm, then it has the ability to train on samples coming from a different policy. What is particular though is that contrary to other off-policy algortihm, it's stable. This mean that the algorithm is much less picky in term of hyperparameter tuning.

SAC [1] is curently **the state of the art** *Deep Reinforcement Learning* algorithm together with Twin Delayed Deep Deterministic policy gradient (*TD3*) [2].

The learning curve of the *Maximum Entropy RL* framework is quite steep due to it's depth and to how much it re-think the RL problem. It was definitavely required in order to understand how SAC work. Tackling the applied part was arguably the most difficult project I did to date, both in term of component to implement & silent bug dificulties. Never the less I'm particularly proud of the result.

You can find my implementaion at https://github.com/RedLeader962/LectureDirigeDRLimplementation

Reading material:

- Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor [1]
- Reinforcement Learning and Control as Probabilistic Inference: Tutorial and Review [3]
- Soft Actor-Critic Algorithms and Applications [4]
- Reinforcement Learning with Deep Energy-Based Policies [5]
- Deterministic Policy Gradient Algorithms [6]
- Reinforcement learning: An introduction [7]

I've also complemented my reading with the following resources:

- CS 294–112 *Deep Reinforcement Learning*: lectures 14-15 by Sergey Levine from University Berkeley;
- *OpenAI: Spinning Up: Soft Actor-Critic*, by Josh Achiam;
- and also *Lil' Log blog:Policy Gradient Algorithms* by Lilian Weng, research intern at *OpenAI*;

Comment on notation:

Since there are a lot of different notations across paper, I've decided to follow (for the most part) the convention established by Sutton & Barto in their book Reinforcement Learning: An Introduction [7]

$(\mathcal{S}, \mathcal{A}, T, \mathcal{R})$	Markov decision process with $\mathcal{S}$ the state space, $\mathcal{A}$ the action space, the transition fct T and a reward space $\mathcal{R}$
$t \in [1, \mathsf{T}]$	time step
T	Time horizon time step $t \in [1, \mathsf{T}]$; Case 1: T is finite, case 2: T is infinite. ($\mathsf{T} = \infty$)
τ	a trajectory $\mathbf{s}_1, \mathbf{a}_1, r_2, \mathbf{s}_2, \mathbf{a}_2, \ldots, r_\mathsf{T}, \mathbf{s}_\mathsf{T}, \mathbf{a}_\mathsf{T}$ aka: episode, trial, rollout
θ	Parameters vector a vector of m parameters $\theta = (\theta_1, \ldots, \theta_m)$
$\pi_\theta(\tau)$	Policy over trajectory τ
$\pi_\theta(\mathbf{a}_t \vert \mathbf{s}_t)$	Policy of parameter θ
$s, \mathbf{s}, s_t,$ or $\mathbf{s}_t$	State or state vector $s, \mathbf{s} \in \mathcal{S}$. $\mathbf{s}$ is a state compose of multiple information ex: $\langle \text{coord} : (x, y, z), \text{heading} : 10\,\text{deg} \rangle$.
$a, \mathbf{a}, a_t,$ or $\mathbf{a}_t$	Action $a, \mathbf{a} \in \mathcal{A}$. $\mathbf{a}$ is an action compose of multiple information ex: $\langle x : +1, y : +2, z : 0,$ say:hello$\rangle$.
r or r_t	the reward at time step t $r, r_t \in \mathcal{R}$. r_t is a shorthand for $R_t = r$ at time step $t \in [1, \mathsf{T}]$
$p(\mathbf{s}' \vert \mathbf{s}, \mathbf{a})$	unknown transition dynamic - 3 argument (aka transition fct of unknown model) model of the world express in terms of conditional probability of getting from $(\mathbf{s}_t, \mathbf{a}_t)$ to $\mathbf{s}_{t+1}$
$r(\mathbf{s}_t, \mathbf{a}_t)$	expected immediate reward from state $\mathbf{s}$ after action $\mathbf{a}$ $r : \mathcal{S} \times \mathcal{A} \longrightarrow \mathbb{R}$;
$J(\theta)$	Objective function (optimization) the function to optimize, sometimes synonymous with loss/cost/error function
$\mathbb{E}_{\tau \sim \pi_\theta(\tau)}[X]$	Expectation of X with respect to the trajectory τ conditioned on the policy π_θ
γ	discount factor penalty to uncertainty of future rewards; $0 < \gamma \leq 1$
G_t or r_t^γ	return or total discounted future reward $G_t = r_t^\gamma = \sum_{k=t}^{\infty} \gamma^{k-t} r(\mathbf{s}_k, \mathbf{a}_k)$

An overview of the *Maximum Entropy* framework

Goal: Solving decision-making problems in a way that define optimality as a context-dependent and multi-solution concept.

How? By learning a Maximum Entropy model.

The *Maximum Entropy* framework is an approach to solving decision-making problem that uses the formalism and tools of the field of *Probabilistic Graphical Model*. The difference between *Classical RL* and *Maximum Entropy RL* is not obvious at first because in both cases, it's all about dealing with probabilities, random variable, expectation maximization and so on. As we will see, those two are fundamentally different.

A other way of framing the decision-making problem

The classical approach used in RL is to formalize the decision-making problem **using a** *Probabilistic Graphical Model* **augmented with a reward input** and then seek to maximize the sum of cumulative reward using some kind of tools borrowed from *Stochastic Optimization* (in the broad sense).

The *Maximum Entropy RL* approach on the other end formalize the decision-making problem **as a** *Probabilistic Graphical Model* and then solve a learning and inference problem using *Probabilistic Graphical Model* tools. While the former can use Probabilistic Graphical Model to describe the RL problem, the later formalize the complete RL problem as *Probabilistic Graphical Model*. In other word, the *Maximum Entropy* framework **formalize and solve the entirety of the RL problem using probability theory**.

Note: This approach to tackling decision-making problem is not new in the literature and has many names: *Maximum Entropy RL*, *KL-divergence control*, *stochastic optimal control*. In this essay we will use the name *Maximum Entropy RL*.

How does it make the RL problem different (an intuition): Consider an environment with a continuous action space. The *Classical RL* approach would specify the agent policy π as a unimodal probability distribution for which the center is the maximal Q-value and indicate the optimal action for a given state. In contrast, the *Maximum Entropy RL* approach would specify the policy as a multimodal distribution for which all mode centers are local maxima Q-values that each indicates good alternative action for a given state.

Why does it matter? Because since in any given real life task, there is in general more than one way to do things, then an RL agent should be able to handle the following scenario:

- Suppose the optimal way is simply impractical at a given time, meaning there is no choice to fallback to a lesser optimal way. Does he know how handle non-optimal alternative way to do things?
- Suppose there is more than one optimal way to do things all leading to the same end result, how does he choose between them?
- Suppose now that there exist multiple equally optimal but different end result, how does he proceed now?
- What about the case where there are many ways to do things and only one optimal way but we want the agent to relax is expectation regarding the end result, in other words, we don't care whether the end result is optimal or near optimal? Will he be able to make good use of that relaxed requirement of near optimality?

Those are all legitimate scenario that an agent should be required to successfully handle in order to become effective, skillful, agile, nimble, resilient and capable of handling adverse condition.

The problem with *Classical RL* is that it converges (in expectation) to a deterministic policy π. This is one of the keys takes away proof from the *Deterministic Policy Gradient* (DPG) [6] paper (see appendix C in the supplementary material). They show that for a wide range of stochastic policy, policy gradient algorithms converge to a deterministic gradient as the variance goes to zero. The same idea goes for value-based algorithms [7] This mean that the algorithm will optimize for one and only one way to do things. Once it starts *believing* that a path is more promising than the others, it will start to optimize for that *believed-best* path and it will discard all the alternate ones. Even if the algorithm is forced to explore using whatever trick, those trick only promote *believed unpromising* path for consideration but it still results in an algorithm learn to optimize for one and only way to do things.

On the other end, *Maximum Entropy RL* optimize for multiple alternate ways of doing things which lead to algorithms that exhibit the following property:

- effective exploration
- transfer learning capabilities out of the box
- robustness and adaptability

■ Nuts and bolt (Key component related to SAC)

The *MaxEnt* policy π:

Recall the *Classical RL* policy π definition

$$\pi_{classical}(\mathbf{a}_t|\mathbf{s}_t) \quad = \quad \mathbb{P}\left[\, \mathcal{A}_t = \mathbf{a} \,|\, \mathcal{S}_t = \mathbf{s} \,\right]$$

which is modelled either as categorical or a Gaussian distribution.

Instead, *Maximum Entropy RL* defines it either in terms of *Q-function* Q^π as

$$\pi_{MaxEnt}(\mathbf{a}_t|\mathbf{s}_t) \quad \propto \quad \exp\left(\frac{1}{\alpha} Q^\pi(\mathbf{s}_t, \mathbf{a}_t)\right)$$

or in terms of *advantage* A^π as

$$\pi_{MaxEnt}(\mathbf{a}_t|\mathbf{s}_t) \quad = \quad \exp\left(\frac{1}{\alpha} Q^\pi(\mathbf{s}_t, \mathbf{a}_t) \;-\; \frac{1}{\alpha} V^\pi(\mathbf{s}_t)\right) \quad = \quad \exp\left(\frac{1}{\alpha} A^\pi(\mathbf{s}_t, \mathbf{a}_t)\right)$$

with α being the *temperature* and $\propto$ the symbol of proportionality

We can observe that it's an analogue of the Boltzmann distribution (aka Gibbs distribution) with the *advantage* being the negative energy found in the Boltzmann distribution. Equivalently, it gives us a *probability measure* of a RL agent doing action $\mathbf{a}_t$ in a given state $\mathbf{s}_t$ as a function of that state energy $A(\mathbf{s}_t, \mathbf{a}_t)$ (*the advantage*). As the *temperature* α decreases and approach to zero, the policy behaves like a standard *greedy policy*

$$\pi_{greedy}(\mathbf{a}_t|\mathbf{s}_t) \;=\; \arg\max_{\mathbf{a}} A^\pi(\mathbf{s}_t, \mathbf{a}_t)$$

This hyperparameter α control the stochasticity of the policy and become very useful later on during training in order to adjust the trade-off between exploration and exploitation.

The *MaxEnt* objective $J(\pi)$:

The RL objective derived from the *Maximum Entropy RL* framework is similar to the *Classical RL* one with the exception of an added entropy term $\mathcal{H}$ and the temperature. α

$$J(\pi_{MaxEnt}) = \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\mathsf{T}} r(\mathbf{s}_t, \mathbf{a}_t) + \alpha\mathcal{H}(\pi(\cdot|\mathbf{s}_t))\right] \tag{1}$$

The *return* ◀ ▶ The temperature

The *action entropy* under the current policy

Key idea: This objective seeks to maximize the expected *return* **and** maximize the action *entropy* at the same time.

Moving part

- **The return:** Same as in *Classical RL*
- **The entropy term:** Can be viewed as regularizer, an uninformative prior over the policy or a way to handle exploration/exploitation.
- **The temperature α:** Control the trade-off between exploration/exploitation.

Entropy

$$\mathcal{H}(p) = -\mathbb{E}_{x\sim p(x)}\left[\log p(x)\right] = -\int_x p(x)\log p(x)\,dx$$

It's a measure of the randomness of a random variable.

- $0 \leq \mathcal{H} \leq 1$
- $\mathcal{H} = 0 \implies$ the variable is deterministic.

Additional information

Intuition:

$\mathcal{H}$ tel us how wide is the distribution from which are sampled the random variables.

- A wide distribution will produce high entropy *random variable.*
- A narrow distribution will produce low entropy *random variable.*

The *soft* value function Q^π and V^π

Under the *Maximum Entropy* framework, both value function are redefined to handle the added entropy term.

First we need to rewrite the *MaxEnt* objective by expanding the *entropy* term and using the *Q-function* definition such that

$$J(\pi_{MaxEnt}) = \mathbb{E}_{(\mathbf{s}_t,\mathbf{a}_t)\sim\pi(\tau)}\left[\sum_{t'=t}^{\mathsf{T}} r(\mathbf{s}_{t'},\mathbf{a}_{t'}) + \alpha\mathcal{H}\left(\pi(\cdot|\mathbf{s}_t)\right)\right] \tag{2}$$

$$\langle \text{ definition of Q-function and entropy } \rangle$$

$$= \mathbb{E}_{(\mathbf{s}_t,\mathbf{a}_t)\sim\pi(\tau)}\Big[Q^\pi(\mathbf{s}_t,\mathbf{a}_t) + \alpha\,\mathbb{E}_{\mathbf{a}\sim\pi}\left[-\log\pi(\mathbf{a}_t|\mathbf{s}_t)\right]\Big] \tag{3}$$

$$\langle \text{ } \mathbf{a} \text{ is condition on the same policy, so the inner expectation can be push outside } \rangle$$

$$= \mathbb{E}_{(\mathbf{s}_t,\mathbf{a}_t)\sim\pi(\tau)}\Big[Q^\pi(\mathbf{s}_t,\mathbf{a}_t) - \alpha\log\pi(\mathbf{a}_t|\mathbf{s}_t)\Big] \tag{4}$$

This leads us to the definition of both *value function*. The *state-action value function* is defined as

$$Q^\pi_{soft}(\mathbf{s}_t,\mathbf{a}_t) = r(\mathbf{s}_t,\mathbf{a}_t) + \mathbb{E}_{(\mathbf{s}_{t+1},\mathbf{a}_{t+1})\sim\pi}\left[\sum_{t'=t+1}^{\mathsf{T}} r(\mathbf{s}_{t'},\mathbf{a}_{t'}) - \alpha\log\pi(\mathbf{a}_{t'}|\mathbf{s}_{t'}))\right] \tag{5}$$

and the *state value function* is defined as

$$V^\pi_{soft}(\mathbf{s}_t) = \mathbb{E}_{\mathbf{a}_t\sim\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)}\left[Q^\pi_{soft}(\mathbf{s}_t,\mathbf{a}_t) - \alpha\log\pi(\mathbf{a}_t|\mathbf{s}_t)\right] \tag{6}$$

or alternatively

$$V^\pi_{soft}(\mathbf{s}_t) = \alpha\log\int_{a_t}\exp\left(\frac{1}{\alpha}Q(\mathbf{s}_t,\mathbf{a}_t)\right)d\mathbf{a}_t \tag{7}$$

We can also rewrite the *Bellman* equation in terms of Q^π_{soft} and V^π_{soft}

$$Q^\pi_{soft}(\mathbf{s}_t,\mathbf{a}_t) = r(\mathbf{s}_t,\mathbf{a}_t) + \mathbb{E}_{\mathbf{s}_{t+1}\sim p(\mathbf{s}_{t+1}|\mathbf{s}_t,\mathbf{a}_t)}\left[V^\pi_{soft}(\mathbf{s}_{t+1})\right] \tag{8}$$

with p being the *transition dynamic*.

Without going into the details about the nuts and bolt of *Maximum Entropy* framework, it's valuable to point out that Q^π_{soft} is what is called in *Probabilistic Graphical Model* a *backward message*. It's not working like a *Classical RL reward-to-go* Q^π nor is it having the same properties.

By definition, $Q^\pi_{soft}(\mathbf{s}_t,\mathbf{a}_t)$ **is the probability of being optimal at timestep** t **until the trajectory end.** It's a measure of the quality of an ongoing trajectory as in the *measure-theory* definition of *measure* contrary to the *Classical RL reward-to-go*, which is not since a $Q^\pi_{classical}(\cdot,\mathbf{s}) = 0$ does not mean an empty set of *reward* magnitude of a trajectory. Take the case of an environment where the *reward* signal upper-bound is 0 by design, then 0 would be the highest possible *return*.

(Key idea) Building bloc of *Maximum Entropy RL*

Framing the RL problem: The *Maximum Entropy* framework **formalize and solve the entirety of the RL problem using probability theory**

Key benefits:

- effective exploration
- transfer learning capabilities out of the box
- robustness and adaptability

Policy

$$\pi_{MaxEnt}(\mathbf{a}_t|\mathbf{s}_t) \quad \propto \quad \exp\left(\frac{1}{\alpha}Q^{\pi}(\mathbf{s}_t,\mathbf{a}_t)\right)$$

State value function

$$V^{\pi}_{soft}(\mathbf{s}_t) \quad = \quad \mathbb{E}_{\mathbf{a}_t\sim\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)}\left[Q^{\pi}_{soft}(\mathbf{s}_t,\mathbf{a}_t) \; - \; \alpha\log\pi(\mathbf{a}_t|\mathbf{s}_t)\right]$$

Q-function

$$Q^{\pi}_{soft}(\mathbf{s}_t,\mathbf{a}_t) \quad = \quad r(\mathbf{s}_t,\mathbf{a}_t) + \mathbb{E}_{(\mathbf{s}_{t+1},\mathbf{a}_{t+1})\sim\pi}\left[\sum_{t'=t+1}^{\mathsf{T}} r(\mathbf{s}_{t'},\mathbf{a}_{t'}) \; - \; \alpha\log\pi(\mathbf{a}_{t'}|\mathbf{s}_{t'}))\right]$$

It's the probability of being optimal at timestep t until the trajectory end.

Objective

$$J(\pi_{MaxEnt}) \quad = \quad \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\mathsf{T}} r(\mathbf{s}_t,\mathbf{a}_t) \; + \; \alpha\mathcal{H}(\pi(\cdot|\mathbf{s}_t))\right]$$

What it does: The *MaxEnt* objective seek to maximize the expected *return* **and** maximize the action *entropy* at the same time.

Building bloc of Soft Actor-Critic

The algorithm seeks to maximize the maximum entropy objective by $J(\pi_{MaxEnt})$ doing *soft policy iteration*, which is similar to regular policy iteration (more on this in the algorithm anatomy section). To do so, the algorithm will have to learn simultaneously the *soft Q-function* Q^{π}_{θ} and *Maximum Entropy policy* π_{MaxEnt}.

Because it's learning both value and policy at the same time, *Soft Actor-Critic* (*SAC* for short) **is considered a *value-based Actor-Critic* algorithm**. This also means that it can be trained using *off-policy* samples.

off-policy learning capability is a very valuable and coveted ability: it means that **the algorithm can learn from samples generated by another policy π distribution than the current one**. Those samples could be coming from the same but older policy π_{older} (in other word samples generated earlier) or they could be coming from a totally different policy π' that is producing them elsewhere.

The key benefit here is that it can **speed up training by reducing the overhead of having to produce new sample at every gradient step**. It's particularly useful in environment where producing new samples is a long process, like in real life robotic.

■ Learning the *soft Q-function*

Recall that we talk earlier about Q^{π}_{soft} being a *Probabilistic Graphical Model* backward message. In order to be able to compute that value efficiently, we need to approximate it. We can do this by representing it has a parametrized function $Q^{\pi}_{\theta}(\mathbf{s}_t, \mathbf{a}_t)$ of parameters θ. We then learn θ by minimizing the squared soft Bellman residual error

$$J_Q(\theta) = \mathbb{E}_{(\mathbf{s}_t,\mathbf{a}_t)\sim\pi}\left[\frac{1}{2}\Big(Q^{\pi}_{\theta}(\mathbf{s}_t,\mathbf{a}_t) - \underbrace{\widehat{Q}^{\pi}_{soft}(\mathbf{s}_t,\mathbf{a}_t)}_{\vdots}\Big)^2\right] \tag{9}$$

$$r(\mathbf{s}_t,\mathbf{a}_t) + \mathbb{E}_{\mathbf{s}_{t+1}\sim p(\mathbf{s}_{t+1}|\mathbf{s}_t,\mathbf{a}_t)}\Big[V^{\pi}_{soft}(\mathbf{s}_{t+1})\Big]$$

with $\widehat{Q}^{\pi}_{soft}$ being the target.

In theory, $V^{\pi}_{soft}(\mathbf{s}_t)$ value can be estimated directly using Equation 6. However, representing V^{π}_{soft} explicitly has the added benefit of helping stabilize learning. We can represent it has a parametrized function $V^{\pi}_{\psi}(\mathbf{s}_t)$ of parameters ψ. We then learn ψ by minimizing the squared residual error

$$J_V(\psi) = \mathbb{E}_{\mathbf{s}_t\sim p}\left[\frac{1}{2}\Big(V^{\pi}_{\phi}(\mathbf{s}_t) - \underbrace{\widehat{V}^{\pi}_{soft}(\mathbf{s}_t)}_{\vdots}\Big)^2\right] \tag{10}$$

$$\mathbb{E}_{\mathbf{a}_t\sim\pi(\mathbf{a}_t|\mathbf{s}_t)}\Big[Q^{\pi}_{\theta}(\mathbf{s}_t,\mathbf{a}_t) - \log\pi_{MaxEnt}(\mathbf{a}_t|\mathbf{s}_t)\Big]$$

with $\widehat{V}^{\pi}_{soft}$ being the target.

We now need a way to represent and learn $\pi_{MaxEnt}(\mathbf{a}_t|\mathbf{s}_t)$.

■ Deriving the objective $J(\pi_{MaxEnt})$ of SAC:

Let first rewrite the *Maximum Entropy RL* objective for a single timestep in terms of Q_θ^π

$$
\begin{aligned}
J(\pi_{MaxEnt}) &= \mathbb{E}_{\mathbf{s}\sim p,\, \mathbf{a}\sim\pi(\mathbf{a}|\mathbf{s})}\Big[\sum_{t'=1}^{\mathsf{T}} r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) \;+\; \alpha\mathcal{H}\left(\pi(\cdot|\mathbf{s})\right)\Big] && (11)\\
&\qquad\qquad \langle \text{ definition of soft Q function and entropy } \rangle \\
&= \mathbb{E}_{\mathbf{s}\sim p,\, \mathbf{a}\sim\pi(\mathbf{a}|\mathbf{s})}\Big[Q_\theta^\pi(\mathbf{s},\mathbf{a}) \;+\; \alpha\,\mathbb{E}_{\mathbf{a}\sim\pi}\left[-\log\pi(\mathbf{a}|\mathbf{s})\right]\Big] && (12)\\
&\qquad\qquad \langle \text{ } \mathbf{a} \text{ is condition on the same policy, so the inner expectation can be push outside } \rangle \\
&= \mathbb{E}_{\mathbf{s}\sim p,\, \mathbf{a}\sim\pi(\mathbf{a}|\mathbf{s})}\Big[Q_\theta^\pi(\mathbf{s},\mathbf{a}) \;-\; \alpha\log\pi(\mathbf{a}|\mathbf{s})\Big] && (13)\\
&= -\,\mathbb{E}_{\mathbf{s}\sim p,\, \mathbf{a}\sim\pi(\mathbf{a}|\mathbf{s})}\Big[\alpha\log\pi(\mathbf{a}|\mathbf{s}) \;-\; Q_\theta^\pi(\mathbf{s},\mathbf{a})\Big] && (14)\\
&= -\,\mathbb{E}_{\mathbf{s}\sim p}\Big[\,\mathbb{E}_{\mathbf{a}\sim\pi(\mathbf{a}|\mathbf{s})}\left[\alpha\log\pi(\mathbf{a}|\mathbf{s}) \;-\; Q_\theta^\pi(\mathbf{s},\mathbf{a})\right]\Big] && (15)\\
&\qquad\qquad \langle \text{ rewriting the inner expectation as a KL-divergence } \rangle \\
J(\pi_{SAC}) &= -\,\mathbb{E}_{\mathbf{s}\sim p}\left[D_{KL}\left(\pi(\cdot|\mathbf{s})\,\middle\|\,\exp\left(\frac{1}{\alpha}Q_\theta^\pi(\mathbf{s}_t,\cdot)\right)\right)\right] + \textit{constant} && (16)
\end{aligned}
$$

with the *constant* being the partition function that is used to normalize the distribution.

We can then learn this objective by minimizing the expected *KL-divergence* directly using this update rule

$$
\pi_{new} \longleftarrow \arg\min_{\pi'\in\Pi} D_{KL}\left(\pi'(\cdot|\mathbf{s})\,\middle\|\,\exp\left(\frac{1}{\alpha}Q_\theta^{\pi_{old}}(\mathbf{s}_t,\cdot)\right)\right) \tag{17}
$$

with Π being a family of policy.

Note: The authors of the SAC paper as demonstrated that the constant can be omitted since it does not contribute to the gradient of $J(\pi_{MaxEnt})$ [1] (see appendix B).

KL-divergence (aka. relative entropy)

$$
D_{KL}\big(P\|Q\big) = \mathbb{E}_{x\sim P(x)}\left[\log\frac{P(x)}{Q(x)}\right] = -\,\mathbb{E}_{x\sim P(x)}\left[\log\frac{Q(x)}{P(x)}\right] = -\,\mathbb{E}_{x\sim P(x)}\left[\,\log Q(x)\,\right] - \mathcal{H}\big(P(x)\big)
$$

It tel us how much different are to distribution

- $0 \le D_{KL}\big(q\|p\big)$
- $D_{KL}\big(q\|p\big) = 0 \implies q$ and p are similar

■ Learning the policy π_{MaxEnt}

Concretely, we are going to approximate the policy π_{SAC} by representing it has parametrized Gaussian distribution $\pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$ of parameters ϕ with a learnable mean and covariance. We cannot directly differentiate a probability distribution but using the *reparameterization trick*, we can remodel the policy so that it exposes different parameters: in our case the mean and covariance of a Gaussian distribution. Using this trick, we can express the action

$$\mathbf{a}_t$$

as

$$f_\phi(\epsilon_t; \mathbf{s}_t) \longmapsto \mathbf{a}_t \tag{18}$$

where $\epsilon \sim \mathcal{N}(\mu, \Sigma)$ and define implicitly the policy π_ϕ in terms of f_ϕ

$$\pi_\phi\Big(f_\phi(\epsilon_t; \mathbf{s}_t) \,|\mathbf{s}_t\Big) \longmapsto \mathbf{a}_t \tag{19}$$

We then rewrite Equation 14 as

$$J_\pi(\phi) \;=\; \mathbb{E}_{\mathbf{s}_t \sim p,\, \epsilon \sim \mathcal{N}(\mu,\Sigma)} \Bigg[\alpha \log \pi_\phi\Big(\overbrace{f_\phi(\epsilon_t; \mathbf{s}_t)}^{\mathbf{a}_t} \,|\mathbf{s}_t\Big) \;-\; Q_\theta^\pi\Big(\mathbf{s}_t,\, \overbrace{f_\phi(\epsilon_t; \mathbf{s}_t)}^{\mathbf{a}_t} \Big) \Bigg] \tag{20}$$

(Key idea) Building bloc of Soft Actor-Critic

Algo. type: *value-based Actor-Critic* algorithm

How does it work? SAC learns simultaneously a *soft Q-function* Q^{π}_{soft} and a *Maximum Entropy policy* π_{MaxEnt}

Capability: *off-policy* learning

- the algorithm can learn from samples generated by another policy π distribution than the current one;
- Key benefit: speed up training by reducing the overhead of having to produce new samples often;

Learning objective:

$$J_Q(\theta) = \mathbb{E}_{(\mathbf{s}_t,\mathbf{a}_t)\sim\pi}\left[\frac{1}{2}\Big(Q^{\pi}_{\theta}(\mathbf{s}_t,\mathbf{a}_t) - \underbrace{\widehat{Q}^{\pi}_{soft}(\mathbf{s}_t,\mathbf{a}_t)}_{\vdots}\Big)^2\right]$$

$$r(\mathbf{s}_t,\mathbf{a}_t) + \mathbb{E}_{\mathbf{s}_{t+1}\sim p(\mathbf{s}_{t+1}|\mathbf{s}_t,\mathbf{a}_t)}\left[V^{\pi}_{soft}(\mathbf{s}_{t+1})\right]$$

$$J(\pi_{SAC}) = -\mathbb{E}_{\mathbf{s}\sim p}\left[D_{KL}\left(\pi(\cdot|\mathbf{s})\,\middle\|\,\exp\left(\frac{1}{\alpha}Q^{\pi}_{soft}(\mathbf{s}_t,\cdot)\right)\right)\right]$$

Trick: Learning V^{π}_{soft} separately is not required in principle but it can be useful to help stabilize learning 👍

$$J_V(\psi) = \mathbb{E}_{\mathbf{s}_t\sim p}\left[\frac{1}{2}\Big(V^{\pi}_{\phi}(\mathbf{s}_t) - \underbrace{\widehat{V}^{\pi}_{soft}(\mathbf{s}_t)}_{\vdots}\Big)^2\right]$$

$$\mathbb{E}_{\mathbf{a}_t\sim\pi(\mathbf{a}_t|\mathbf{s}_t)}\left[Q^{\pi}_{\theta}(\mathbf{s}_t,\mathbf{a}_t) - \log\pi_{MaxEnt}(\mathbf{a}_t|\mathbf{s}_t)\right]$$

Trick: Apply the reparameterization trick to transform the policy π_{MaxEnt} as a learnable Gaussian distribution

$$J_{\pi}(\phi) = \mathbb{E}_{\mathbf{s}_t\sim p,\,\epsilon\sim\mathcal{N}(\mu,\Sigma)}\left[\alpha\log\pi_{\phi}\Big(f_{\phi}(\epsilon_t;\mathbf{s}_t)\,|\mathbf{s}_t\Big) - Q^{\pi}_{\theta}\Big(\mathbf{s}_t,\,f_{\phi}(\epsilon_t;\mathbf{s}_t)\Big)\right]$$

🔁 Algorithm anatomy:

In order to be effective while tackling large continuous domain, the *SAC* algorithm uses an approximation of the *soft policy iteration* algorithm:

1. It used function approximator for the *soft Q-function* Q^{π}_{soft} and the policy π_{SAC}
2. It alternates between *soft policy evaluation* and *soft policy improvement* instead of running each one to convergence separately like in *Classical RL policy iteration*.

Algorithm 1: *soft policy iteration*

repeat

1. collect sample by acting in the environment with respect to the policy π_{SAC}
2. *soft policy evaluation* step: evaluating Q^{π}_{soft} for a fixed policy π at each step of a trajectory using the *soft Bellman* Equation 8
3. *soft policy improvement* step: Update the policy by minimizing the expected KL-divergence using Equation 16

until *convergence*

🏗 Training soft function approximator: implementation details

The algorithm will learn

1. a parametrized soft *state-value function* $V^{\pi}_{\psi}(\mathbf{s}_t)$;
2. two parametrized *soft Q-function* $Q^{\pi}_{\theta}(\mathbf{s}_t, \mathbf{a}_t)$;
3. and a *maximum entropy policy* $\pi_{\phi}(\mathbf{a}_t|\mathbf{s}_t)$;

ψ and θ will be modelled as neural networks. ψ will be reparameterized as a Gaussian distribution with a mean and covariance learnable by a neural network.

The algorithm also uses a **replay buffer** D to collect and accumulate samples $(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1}, done_{t+1})$. One of the key benefits of sampling tuple $(\mathbf{s}_t, \mathbf{a}_t, r_t, \mathbf{s}_{t+1})$ randomly from a *replay buffer* is that it breaks temporal correlation which helps learning 👍

Why learn the soft *state-value function* V^{π}_{ψ} ? As we talked earlier there is no theoretical requirement for learning $V_{\psi}(\mathbf{s}_t)$ since it can be recovered from Equation 6 directly using $Q_{\theta}(\mathbf{s}_t, \mathbf{a}_t)$ and $\pi_{\phi}(\mathbf{a}_t|\mathbf{s}_t)$. In practice, it can stabilize training 👍

Why learn the two soft *Q-function* Q^{π}_{θ} ? The policy improvement step is known to produce positive bias that reduces the performance in value-based method. This is a trick (aka *clipped double-Q*) that help reduce the impact of this problem. How it work is that the algorithm learn $Q^{\pi}_{\theta 1}$ and $Q^{\pi}_{\theta 2}$ separately then take the minimum between the two when training V^{π}_{ψ}. In practice, the SAC authors founded that "it significantly speed up training, especially on harder task" 👍

Detail regarding the Soft state value function

Like we explained earlier we can train the *soft state value function* by least squared regression

$$J_V(\psi) = \mathbb{E}_{\mathbf{s}_t \sim D}\left[\frac{1}{2}\left(V_\psi^\pi(\mathbf{s}_t) - \mathbb{E}_{\mathbf{a}_t \sim \pi_\phi}\left[Q_\theta^\pi(\mathbf{s}_t, \mathbf{a}_t) - \log \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)\right]\right)^2\right] \tag{21}$$

$\min\left(Q_{\theta 1}^\pi, Q_{\theta 2}^\pi\right)$

⚠ Those action are not sampled from the replay buffer!

Observe that state $\mathbf{s}_t$ is sampled from the *replay buffer* but not action $\mathbf{a}_t$ which is sampled from the current policy π. It's a critical detail that is easy to miss. Also, this is where we make use of our two learned *soft Q-function*.

Detail regarding the Soft Q-function

Again, we can train the *soft Q-function* by least squared regression

$$J_Q(\theta) = \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \pi}\left[\frac{1}{2}\left(Q_\theta^\pi(\mathbf{s}_t, \mathbf{a}_t) - \widehat{Q}_{soft}^\pi(\mathbf{s}_t, \mathbf{a}_t)\right)^2\right] \tag{22}$$

$$r(\mathbf{s}_t, \mathbf{a}_t) + \mathbb{E}_{\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)}\left[V_{\bar{\psi}}^\pi(\mathbf{s}_{t+1})\right]$$

Target value network

The learning target is represented by a copy of the main $V_\psi^\pi(\mathbf{s}_t)$ with parameter noted $\bar{\psi}$. Network weight from V_ψ^π are copied in a controlled manner to $V_{\bar{\psi}}^\pi$ using exponential moving average and adjusted by a target smoothing coefficient hyperparameter τ.

Detail regarding the *Soft Actor-Critic* policy

Policy π_ϕ is trained using Equation 0.1.3

$$J_\pi(\phi) = \mathbb{E}_{\mathbf{s}_t \sim p,\, \epsilon \sim \mathcal{N}(\mu, \Sigma)}\left[\alpha \log \pi_\phi\Big(f_\phi(\epsilon_t; \mathbf{s}_t)\,|\mathbf{s}_t\Big) - Q_\theta^\pi\Big(\mathbf{s}_t,\, f_\phi(\epsilon_t; \mathbf{s}_t)\Big)\right]$$

Like we have explained earlier, the policy π is modelled using a Gaussian distribution. It's important to consider the fact that Gaussian distributions are unbounded contrary to our policy which needs to produce action inside a bound reflecting the environment *action space*. Thus, enforcing those bounds is done by applying a squashing function $\sum_{i=1}^{|\mathcal{A}|} \log(1 - \tanh^2(action))$.

Algorithm 2: *Soft Actor-Critic*

hParam(*Target smoothing* τ) $\leftarrow 0.005$ "SAC paper"

hParam(*GS-INTERVALE*) $\leftarrow 1$ "SAC paper"

hParam(*TARGET-INTERVALLE*) $\leftarrow 1$ "SAC paper"

hParam(*MAXBUFFERSIZE*) $\leftarrow 10e6$ "SAC paper"

hParam(*MINIBATCHSIZE*) $\leftarrow 256$ "SAC paper"

Input: Learning environment

Network *initialize* $\psi, \theta_1, \theta_2, \phi$

Network $\bar{\psi} \longleftarrow \psi$

ReplayBuffer *initialize to MAXBUFFERSIZE*

repeat environment step t

— Collect sample ———

$\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$ "Run the policy"

$r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$ "Observe reaction"

$\mathcal{D} \longleftarrow \mathcal{D} \cup (\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})$ "Collect experience in replay buffer"

if *GS-INTERVALE reached* **then**

MiniBatch$[(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1}), \ \ldots \ ] \longleftarrow \mathcal{D}$ "Sample from replay buffer"

— *soft policy evaluation* step ——

$\psi \longleftarrow \psi - \lambda_V \widehat{\nabla}_\psi J_V(\psi)$ "Update the state value network"

$\theta_1 \longleftarrow \theta_1 - \lambda_Q \widehat{\nabla}_\theta J_Q(\theta_1)$ "Update the state-action value network 1"

$\theta_2 \longleftarrow \theta_2 - \lambda_Q \widehat{\nabla}_\theta J_Q(\theta_2)$ "Update the state-action value network 2"

— *soft policy improvement* step ——

$\phi \longleftarrow \phi - \lambda_\pi \widehat{\nabla}_\phi J_\pi(\phi)$ "Update the policy network"

if *TARGET-INTERVALLE reached* **then** update target

$\bar{\psi} \longleftarrow \tau\psi + (1-\tau)\bar{\psi}$ "Update the state value target network"

until *learning goal reached*

Soft Actor-Critic - In Practice

Implementing *SAC* was definitely not an easy venture. There was a lot of moving pieces, interacting part and many small important details. It was a new framework for me, so the challenge was both on a theoretical and applied level. Of course every time one adds a complexity layer to a problem, narrower the margin for error become and the steepest the fall. Nevertheless it was thrilling and very fulfilling especially the moment I finally started seeing results.

There were many tasks to handle in order to implement SAC, among others:

- expanding my previous learning agent implementation to support: a new architecture, a new data acquisition requirement and new hyperparameters;
- implementing an agent brain comprised of **5 neural nets**;
- 3 different optimization operations;
- a neural net weight update operation that **required to periodically fetch parameters from his older brother, tensor by tensor**;
- a **continuous action space policy** with learnable mean and covariance;
- a way to convert this stochastic policy on the fly to a deterministic one;
- a **replay buffer** of arbitrary size for storing collected samples with the capability to renew is stock along the way;
- a **minibatch sampler** that aggregates randomly selected samples from the replay buffer to feed the learning agent;
- **collecting additional metrics** from those 5 neural nets and the agent in order to evaluate performance and understand how *Maximum Entropy RL* behave and work;
- expand my tools for running experiment and organize results and, **of course, . . .**
  solving silent bugs;

Once the implementation was working and the agent was exhibiting some timid proof of life, the second challenge was to understand how this new framework behave and get acquainted with very sensitive hyperparameter that has the power to make or break the algorithm if poorly understood. Reading about it is one thing, being able to recognize their effects among the noise of other things during experimentation is something else.

From theory to practice

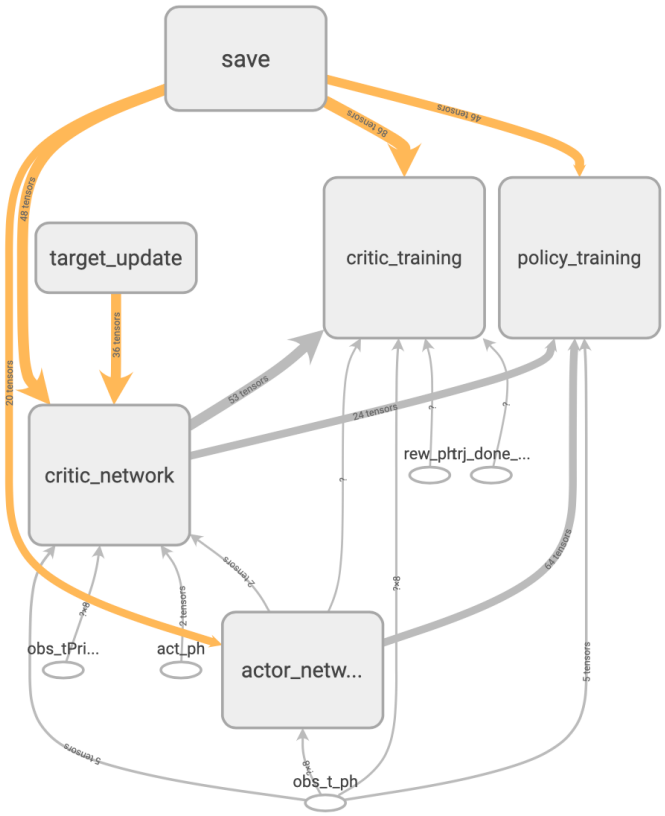
Implementing a policy for continuous action space

Like every other algorithm capable of handling continuous action space, the common strategy is to model the policy has a Gaussian distribution with the center of the distribution μ representing the best action $\mathbf{a}$ given the $\mathbf{s}$. μ is learned with a function approximator and we don't care about the covariance since we are optimizing for 'the' optimal path.

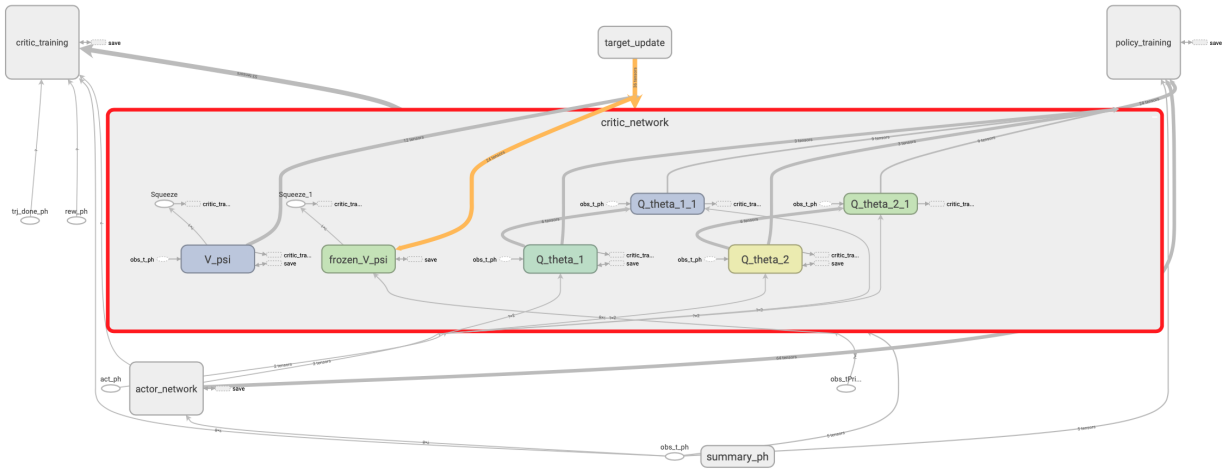
Under the *Maximum Entropy* framework, things are a little bit different as we are optimizing for every optimal behaviour, even those under adverse condition. In that regard, the policy cannot just rely on knowing the position of the peak under the distribution. The policy must learn multiple modes and to do so, he must have control over learning the covariance as well as μ. Thus, it's critical to validate that the gradient is able to flow throughout all those component during back propagation.

Keeping up with 5 separate neural networks

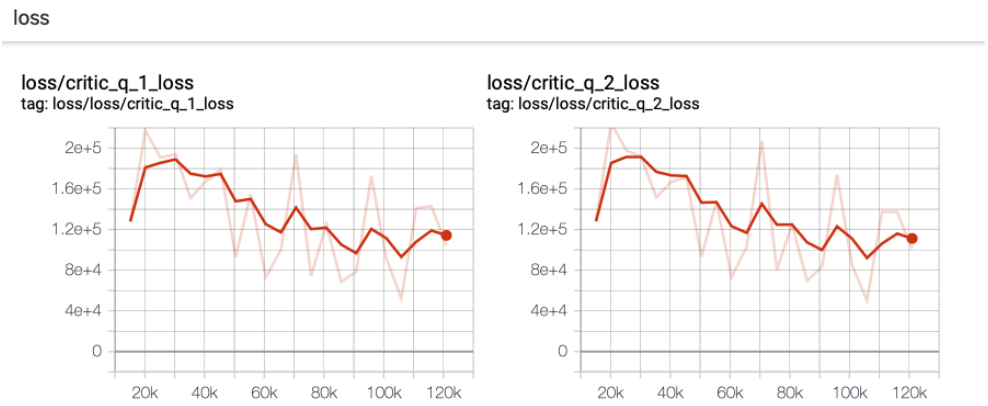
One of the main challenges was to handle the construction of those graph, their interaction, make sure the data was piped in the right place. Because of the growing number of *TensorFlow* `Tensors` and `Operations`, keeping things organized became a necessity. This required the uses of a standardize vocabulary and re-grouping `Tensors` and `Operations` under logical `name scope`.



Like I have talked previously, *Soft Actor-Critic* implementation makes use of a *Gaussian distribution* parametrized by a neural net ϕ, two *Q-function* θ of same architecture behaving closely and two *state value functions* of same architecture but with one always behind in time in terms of optimization.



The fact that both θ networks were never far apart in terms of optimization added a bit of confusion during debugging since the feedback you get from watching their collected metrics on TensorBoard is that they are the same! Then you are faced with the doubt that maybe, you did something wrong and that the first one is cloned into the other for whatever reason.



</> Implementing the target update operation

Note on target network

This idea of using two slightly different copy of the same network to optimize the mean squared error is common among value-based algorithms.

The problem arises from the fact that we are trying to minimize the loss of two functions depending on the same parameters. Performing a delayed update on the target network keep them closely related but different enough so that evaluating their differences as they evolve can give a useful learning signal. In effect, it stabilizes training.

This is one of the key contributions of *DQN* in terms of architecture.

This was a tricky part for me since I was not familiar with that kind of task involving low-level TensorFlow component. So the story goes like this.: SAC need a way to keep track of two closely evolving neural nets ψ and $\bar{\psi}$ with one always lagging behind the other one in terms of optimization. They are both named `V_psi` and `frozen_V_psi` in the snapshot of *SAC* trainable variable.

```
:: TRAINABLE_VARIABLES
[   <tf.Variable 'actor_network/phi/hidden_1/kernel:0' shape=(8, 64) dtype=float32_ref>
    <tf.Variable 'actor_network/phi/hidden_1/bias:0' shape=(64,) dtype=float32_ref>
    <tf.Variable 'actor_network/phi/logits/kernel:0' shape=(64, 2) dtype=float32_ref>
    <tf.Variable 'actor_network/phi/logits/bias:0' shape=(2,) dtype=float32_ref>
    <tf.Variable 'actor_network/policy_mu/kernel:0' shape=(2, 2) dtype=float32_ref>
    <tf.Variable 'actor_network/policy_mu/bias:0' shape=(2,) dtype=float32_ref>
    <tf.Variable 'actor_network/policy_log_std/kernel:0' shape=(2, 2) dtype=float32_ref>
    <tf.Variable 'actor_network/policy_log_std/bias:0' shape=(2,) dtype=float32_ref>
    <tf.Variable 'critic_network/V_psi/hidden_1/kernel:0' shape=(8, 64) dtype=float32_ref>
    <tf.Variable 'critic_network/V_psi/hidden_1/bias:0' shape=(64,) dtype=float32_ref>
    <tf.Variable 'critic_network/V_psi/logits/kernel:0' shape=(64, 1) dtype=float32_ref>
    <tf.Variable 'critic_network/V_psi/logits/bias:0' shape=(1,) dtype=float32_ref>
    <tf.Variable 'critic_network/frozen_V_psi/hidden_1/kernel:0' shape=(8, 64) dtype=float32_ref>
    <tf.Variable 'critic_network/frozen_V_psi/hidden_1/bias:0' shape=(64,) dtype=float32_ref>
    <tf.Variable 'critic_network/frozen_V_psi/logits/kernel:0' shape=(64, 1) dtype=float32_ref>
    <tf.Variable 'critic_network/frozen_V_psi/logits/bias:0' shape=(1,) dtype=float32_ref>
    <tf.Variable 'critic_network/Q_theta_1/mlp/hidden_1/kernel:0' shape=(10, 64) dtype=float32_ref>
    <tf.Variable 'critic_network/Q_theta_1/mlp/hidden_1/bias:0' shape=(64,) dtype=float32_ref>
    <tf.Variable 'critic_network/Q_theta_1/mlp/logits/kernel:0' shape=(64, 1) dtype=float32_ref>
    <tf.Variable 'critic_network/Q_theta_1/mlp/logits/bias:0' shape=(1,) dtype=float32_ref>
    <tf.Variable 'critic_network/Q_theta_2/mlp/hidden_1/kernel:0' shape=(10, 64) dtype=float32_ref>
    <tf.Variable 'critic_network/Q_theta_2/mlp/hidden_1/bias:0' shape=(64,) dtype=float32_ref>
    <tf.Variable 'critic_network/Q_theta_2/mlp/logits/kernel:0' shape=(64, 1) dtype=float32_ref>
    <tf.Variable 'critic_network/Q_theta_2/mlp/logits/bias:0' shape=(1,) dtype=float32_ref>
]
```

We can't copy the entire network at once because of the way *TensorFlow* computation graph are designed. We can see it as a custom-made aggregation of *tensor* with full control over their component. In other words, *TensorFlow* is not designed to be user-friendly, it is designed for efficiency and computation power. So there is no easy solution out of the box to do this like a `newNet = easyCopyMyNeuralNetwork(oldNet)` function.

The way to handle this is to build a *TensorFlow operation* that fetches the relevant *tensor* part of the`V_psi` graph among all the available *Trainable Variables* and assign their weight to `frozen_V_psi` by performing polyak averaging. This operation once built became part of the computation graph.

(See on next page, Figure: My target network update *TensorFlow operation* implementation detail)

```python

def update_nn_weights(graph_variables_from: List[tf_cv1.Variable], graph_variable_to: List[tf_cv1.Variable],
                      target_smoothing_coefficient: float) -> tf.Operation:
    """
    Fetch all tensor in list graph_key_from and update tensor weight of those in graph_key_to
    Pre condition: Batch tensor graph key list must match
    """
    with tf_cv1.variable_scope('update_nn_weights_op', reuse=True):
        op = tf.group(
            [tf_cv1.assign(
                updated_tensor,
                target_smoothing_coefficient * source_tensor + (1 - target_smoothing_coefficient) * updated_tensor)
                for source_tensor, updated_tensor in zip(graph_variables_from, graph_variable_to)]
            )
    return op


def get_current_scope_variables(name: str) -> List[tf_cv1.Variable]:
    """
    Fetch the list of all parameter graph key under a specific variable name
    Pass to argument 'name':
        - If called INSIDE a scope --> only the relevant key ex: 'V_psi'
        - If called OUTSIDE a scope --> the full key ex: 'Tatget_network/V_psi'

        ''' with tf_cv1.variable_scope('Tatget_network'):
                the_V = build_MLP_computation_graph(obs_t_ph, 1, (4, 4), name='V_psi')
                the_frozen_V = build_MLP_computation_graph(obs_t_prime_ph, 1, (4, 4), name='frozen_V_psi')
            v_psy_key = get_current_scope_variables('Tatget_network/V_psi')
            print(v_psy_key)
        '''

        [<tf.Variable 'V_psi/hidden_1/kernel:0' shape=(4, 4) dtype=float32_ref>,
         <tf.Variable 'V_psi/hidden_1/bias:0' shape=(4,) dtype=float32_ref>,
         <tf.Variable 'V_psi/hidden_2/kernel:0' shape=(4, 4) dtype=float32_ref>,
         <tf.Variable 'V_psi/hidden_2/bias:0' shape=(4,) dtype=float32_ref>,
         <tf.Variable 'V_psi/logits/kernel:0' shape=(4, 1) dtype=float32_ref>,
         <tf.Variable 'V_psi/logits/bias:0' shape=(1,) dtype=float32_ref>]

    :param name: variable name
    :return: a list of all variable under 'name' graph key
    """
    assert not name.endswith('/')

    scope_name = tf_cv1.get_variable_scope().name
    if len(scope_name) > 0:
        scope_name += '/' + name + '/'
    else:
        scope_name = name + '/'

    return get_variable_explicitely_by_graph_key_from(scope_name)


def get_variable_explicitely_by_graph_key_from(name: str) -> List[tf_cv1.Variable]:
    assert not name.endswith('/')
    param_key = tf_cv1.get_collection(
        tf_cv1.GraphKeys.TRAINABLE_VARIABLES, scope=name)
    return param_key

```

My target network update *TensorFlow operation* implementation detail
`tensorflowbloc.py` module from the `blocAndTools` package

</> Implementing the replay buffer

This is "not a big challenge to implement" per se but it is a very crucial one. A one for which a careless design and implementation can have a detrimental effect on the agent learning wall clock speed and also, on another level, impede the development process because of the delayed feedback between experimentation cycles. So it's essential to take as much time as possible to optimize the replay buffer for speed as some of its operation are going to be performed millions of times over a experimentations set.

Let suppose you setup an experimentation to perform training of a SAC algorithm over a period of `50 epoch` of `5000 timestep` each. Then SAC will have to perform `250000 gradient steps` during an experimentation run. Now you also decide to set a `batch size of 100`, which is a relatively conservative value. Since *SAC* standard design (the one with target update by exponential moving average) performs `1 gradient step` for every timestep it collects, this means that *SAC* will handle 25 million samples during the course of one experimentation. The price of a poor design choice can be quite significative when performed $25e^6$ times.

One design experiment I have conducted showed that, given that we want to compute data at the end:

- writing to a prefixed size list is close in speed to writing to a numpy array;
- appending to a list is 45 times slower than writing to a prefixed size container;

```
Experiment result:

    Sizeof (with 250000 items):
        numpy array 2.000096 Mb
        python list 2.000064 Mb

    Timeit result:
        5 loops,

        Write directly to numpy array then compute:                  5.91130 usec per loop
        Append to list, convert to numpy array then compute:         266.94095 usec per loop
        Write to fixed size list, convert to numpy array then compute:  8.11612 usec per loop
```

write_vs_apend_data_structure_benchmark.py

In another benchmark experiment regarding the design of an `EpisodeData` class proof of concept, I observed the following:

- numpy array indexing (READ) is 2X slower compared to (READ) on list of the same size;
- overridden methods are a little slower to execute than an original one;
- execution behaves differently with respect to container size (size: 4000 vs 4000000);
- result change drastically if TensorFlow is running in another python process at the same time;

```
Experiment result:

    === 4 000 000 items ===========================================================

    --- Timeit WRITE result (with 4000000 items) ----------------------------------
        250 step per episode, over 20 timeit loops

        populate_numpy_array: ›› › › › › › › › › ›  8.24359 usec per loop
        populate_append_to_list: › › › › › › › › › ›  13.24392 usec per loop
        populate_fix_size_list: › › › › › › › › › ›  10.36419 usec per loop
        populate_EpisodeData(EpisodeData): ›› › › › › › ›  3.31320 usec per loop
        populate_EpisodeData(EpisodeDataFixSizeList): › › › › ›  2.65059 usec per loop

        Overrided (WRITE):
        overrided_populate_EpisodeData(EpisodeData): › › › › ›  3.66189 usec per loop
        overrided_populate_EpisodeData(EpisodeDataFixSizeList): › › ›  2.91738 usec per loop


    --- Timeit READ result (with 4000000 items) -----------------------------------
        250 step per episode, over 20 timeit loops

        numpy_array_container: ›› › › › › › › › › ›  72.16929 usec per loop
        append_to_list: › › › › › › › › › › ›  24.34082 usec per loop
        fix_size_list: ›› › › › › › › › › ›  23.58640 usec per loop
        read_and_write_EpisodeData(EpisodeData): › › › › › ›  28.45965 usec per loop
        read_and_write_EpisodeData(EpisodeDataFixSizeList): › › › ›  28.79260 usec per loop

        Overrided (READ):
        overrided_read_and_write_EpisodeData(EpisodeData): ›› › › ›  26.65566 usec per loop
        overrided_read_and_write_EpisodeData(EpisodeDataFixSizeList): › 26.11270 usec per loop


    --- Timeit READ&WRITE result (with 4000000 items) -----------------------------
        250 step per episode, over 20 timeit loops

        numpy_array_container: ›› › › › › › › › › ›  80.41288 usec per loop
        append_to_list: › › › › › › › › › › ›  37.58474 usec per loop
        fix_size_list: ›› › › › › › › › › ›  33.95059 usec per loop
        read_and_write_EpisodeData(EpisodeData): › › › › › ›  31.77285 usec per loop
        read_and_write_EpisodeData(EpisodeDataFixSizeList): › › › ›  31.44319 usec per loop

        Overrided (READ):
        overrided_read_and_write_EpisodeData(EpisodeData): ›› › › ›  30.31756 usec per loop
        overrided_read_and_write_EpisodeData(EpisodeDataFixSizeList): › 29.03008 usec per loop


    --- Sizeof (with 4000000 items) -----------------------------------------------
        numpy array›› › › › › ›  512.00011 Mb
        python list›› › › › › ›  512.00102 Mb
        EpisodeData›› › › › › ›  1287.36000 Mb
        EpisodeDataFixSizeList› › › › ›  1282.49600 Mb
```

`EpisodeData` class proof of concept

`data_structure_prof_of_concept_benchmark.py`

>_ Important component that make or break SAC

Evaluating the learned policy by making it deterministic

Recall that *SAC* optimize a stochastic policy π_{MaxEnt}, so when it comes to evaluating the performance it is useful to make it deterministic. This same strategy can be used at deployment time of a trained *SAC* agent.

How? By using a variant of the π_{MaxEnt} where instead of returning a sampled action from the learned distribution, we return it's mean μ (like in a *Classical RL* policy for continuous action space).

$$\pi_{Stochastic} \longrightarrow \mathbf{a}_t \sim \pi_t$$

vs

$$\pi_{Deterministic} \longrightarrow \mathbf{a}_t = \mu_t$$

</> Understanding the reward scaling and the quest for finding the right one

Reward scaling is the process of raising or lowering the magnitude of the reward by multiplying each reward term by a constant: `reward scaling` coefficients that we will note η.

$$\eta \cdot r(\mathbf{s}_t, \mathbf{a}_t) \qquad \forall t \in [1, \mathsf{T}]$$

It's a common trick that is useful when an algorithm is sensitive to the reward magnitude.

The *Maximum Entropy RL* policy definition with `reward scaling` coefficient is

$$\begin{aligned} \pi_{MaxEnt}(\mathbf{a}_t|\mathbf{s}_t) &\propto \exp\left(\frac{1}{\alpha} Q^{\pi}_{soft}(\mathbf{s}_t, \mathbf{a}_t)\right) \\ &\qquad \langle \text{ Definition of the } \textit{soft Bellman equation} \rangle \\ &\propto \exp\left(\frac{\eta \cdot r(\mathbf{s}_t, \mathbf{a}_t) + \mathbb{E}_{\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)}\left[V^{\pi}_{soft}(\mathbf{s}_{t+1})\right]}{\alpha}\right) \end{aligned}$$

with α being the `temperature` .

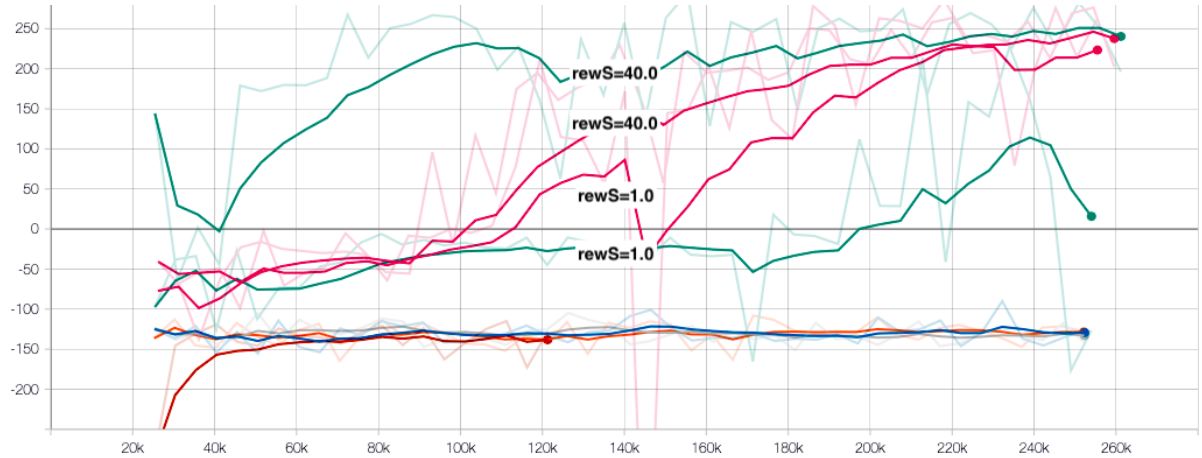
As the authors of *SAC* paper mentioned, *SAC* is very sensitive to this hyperparameter setting since the reward plays the same role as the inverse of the energy in its analogue, the Boltzmann distribution. So `reward scaling` coefficient has the opposite effect of the temperature α thus tuning both parameters in the same experiment is useless, it's one or the other.

Intuitively:

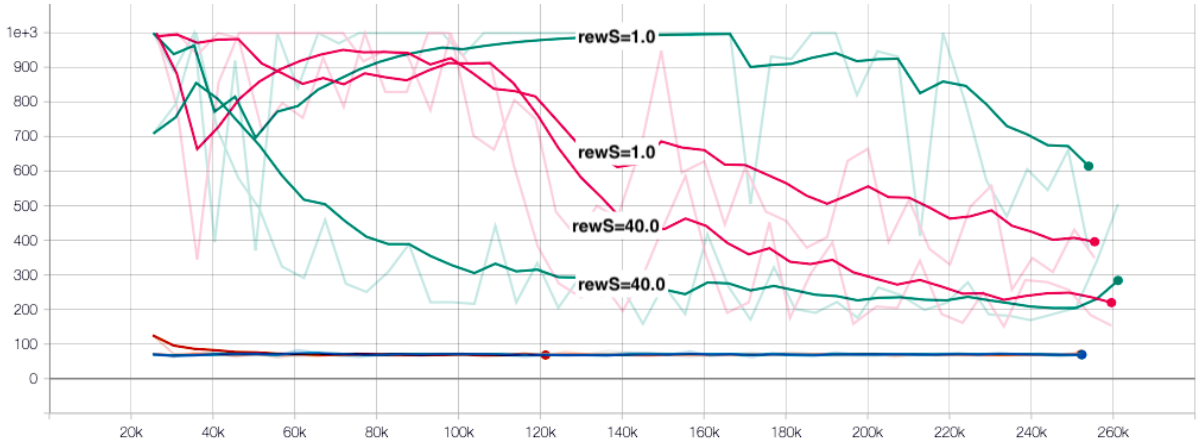
- Low reward magnitudes will make all rewards look identical to the agent, so the agent won't be able to see a difference between events;
- High reward magnitudes will make the agent completely discard event linked to smaller reward making the policy almost deterministic;

We can observe the effect of `reward scaling` in the following experiment done on the `LunarLanderContinuous-v2` environment where I performed 8 training run divided over 4 different `reward scaling` coefficient settings (1.0, 5.0, 20.0, 40.0) using no random seed. We can see how 40.0 is the only setting leading to consistent results where the agent succeeded to learn 2 times out of 2.

Note: This experiment was performed to give a general idea and need to be rerun at least 3 more times to have significant empirical value.



Deterministic policy π



Trajectory lenght

Reward scaling experiment over 1.0, 5.0, 20.0, 40.0

ExperimentSpec {

'Rerun tag': `LunarLander-EMA-MinPool-RewSFour-reward_scaling=(1.0|5.0|20.0|40.0)`
'Max epoch': 50
'Timestep per epoch': 5000
'Discount factor': 0.99
'Learning rate': 0.003
'All Neural Nets topo': (200X200)
'Hidden layers activation': relu
'Replay buffer capacity': 50000
'Batch size in timestep': 200
'Minimum replay buffer size': 200000
'reward scaling': [1.0, 5.0, 20.0, 40.0]
'Random seed': None

}

</> The effect of the replay buffer size

This component is composed of two aspects:

- The **minimum** `replay buffer` size before starting to take gradient step
- The **maximum** `replay buffer` size before starting to overwrite older samples

This is an underrated component in the *SAC* literature. In fact, the part regarding the **minimum** `replay buffer` size is not discussed at all.

The effect in principle is that **the bigger the** `replay buffer`**, the more diversify the experience pool will be for the learning agent**. The caveat, though, is that the bigger the `replay buffer`, the slower samples in the replay buffer get renewed, the slower the learning agent gets access to samples coming from trajectories that are closer to optimal one, which is what the algorithm is optimizing for. Well, that's **the common assumption made in the case or *Classical RL***, where you usually want to **prioritized sample leading to the optimal trajectories**.

Things are different in the case of *Maximum Entropy RL*. Intuitively, the bigger and diversify the experience pool, the wider and deeper the learning scope gets. As *Maximum Entropy RL* optimize for many optimal path and alternative scenario, learning from a large `replay buffer` become a requirement. Finding the right `maximum buffer size` setting is significant because it affects how old sample get renewed.

The common assumption coming from *Classical RL* is that since newer trajectory samples come from a policy π distribution closer to the optimal goal, then they must have more learning value. It's important to point out that **this logic does not apply to *Maximum Entropy RL* as non-optimal trajectory samples are essential to learning how to handle adverse condition. So getting solely expose to optimal condition won't cut it.**

The agent needs to have a hard time, need to be beaten up, shaken, rattle and so on. Putting it in a way that relates to human performances in sport:

No one ever became strong by having it easy.

That's the general idea about how **the upper bound** on the `replay buffer` size has huge consequences on *SAC* performance. Now let's talk about the underrated one, the **lower bound**, as in the **minimum** `replay buffer` size before starting to take gradient steps.

Suppose we start training early, right at the end of the first trajectory at $t = 200$. We choose a batch size of 100 for training, so gradient step will look like this:

1 is done with samples from $\frac{100}{200}$ of the replay buffer;

2 is done with samples from $\frac{100}{201}$ of the replay buffer;

3 is done with samples from $\frac{100}{202}$ of the replay buffer;

4 is done with samples from $\frac{100}{203}$ of the replay buffer;

5 ...

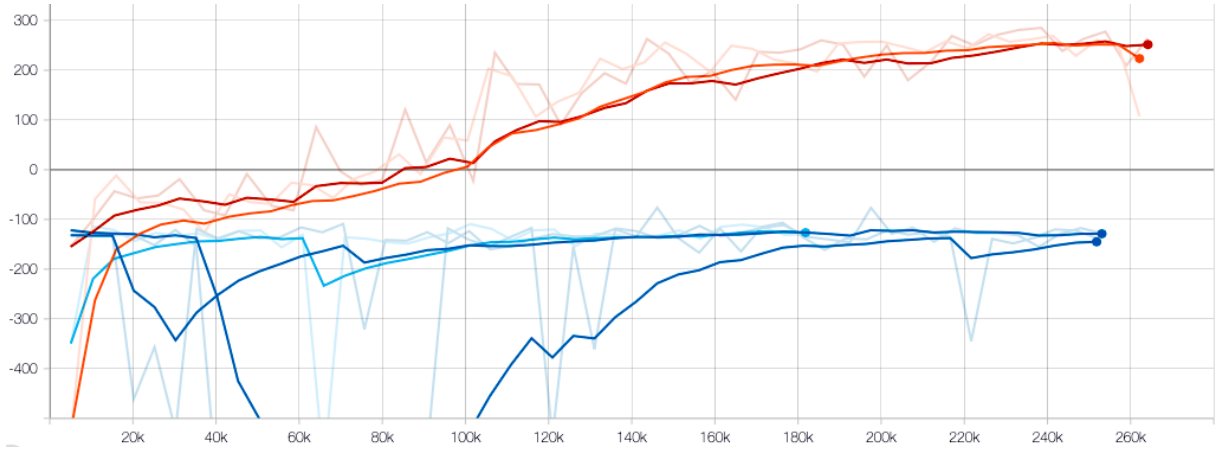
so since the `replay buffer` size grow slower than the speed at which the agent train on them, we can see how he will probably overfit those first samples and reach a local maxima.

The **best-case scenario** is that at every timestep, the sampler pick the ones that were less used for training;
The **worst-case scenario** is that at every timestep, the sampler pick all the same one consecutively;

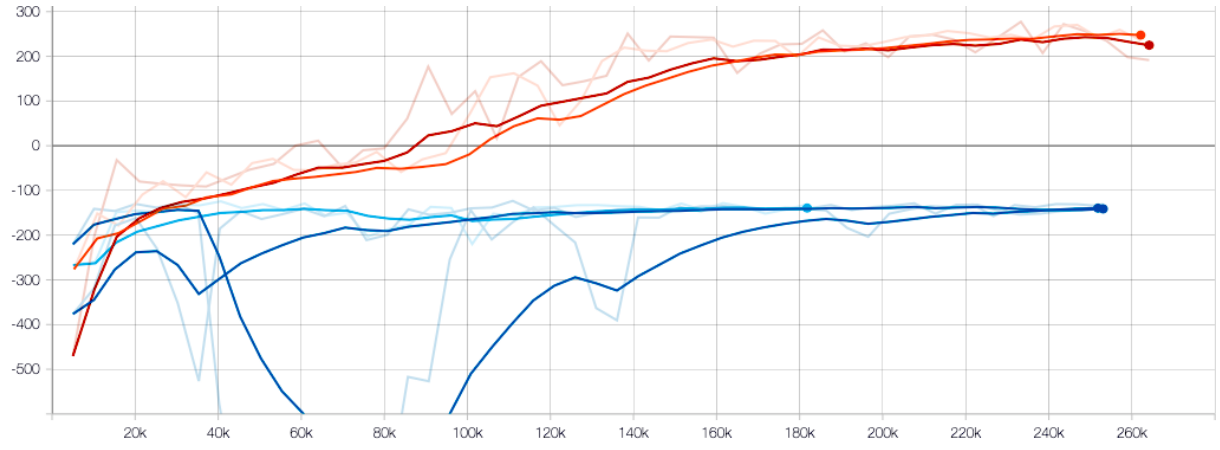
It's true that both the worst case and best-case scenario can happen for every `minimum replay buffer` size setting. The problem is that the smaller the `minimum replay buffer` size, the greater the chances of encountering the worst-case scenario and the lesser the chances of encountering the best-case scenario.

This intuition is aligned with what I have observed during experimentation. When the `minimum replay buffer` size was too small for the training environment, then the agent training performance was inconsistent. On the other end, when the `minimum replay buffer` size was large enough for the training environment, then the agent training was succeeding at every run.

In the following experiment done on the `LunarLanderContinuous-v2` environment, I performed 15 training run divided over 3 different `minimum replay buffer` settings (300, 10000 and 20000) using no random seed. We can observe how the lower settings 300 and 10000 gave inconsistent results as the agent failed to learn 3 times out of 5 as opposed to a large enough `minimum replay buffer` setting of 20000, which gave consistent results as the agent succeeded to learn 5 times out of 5.

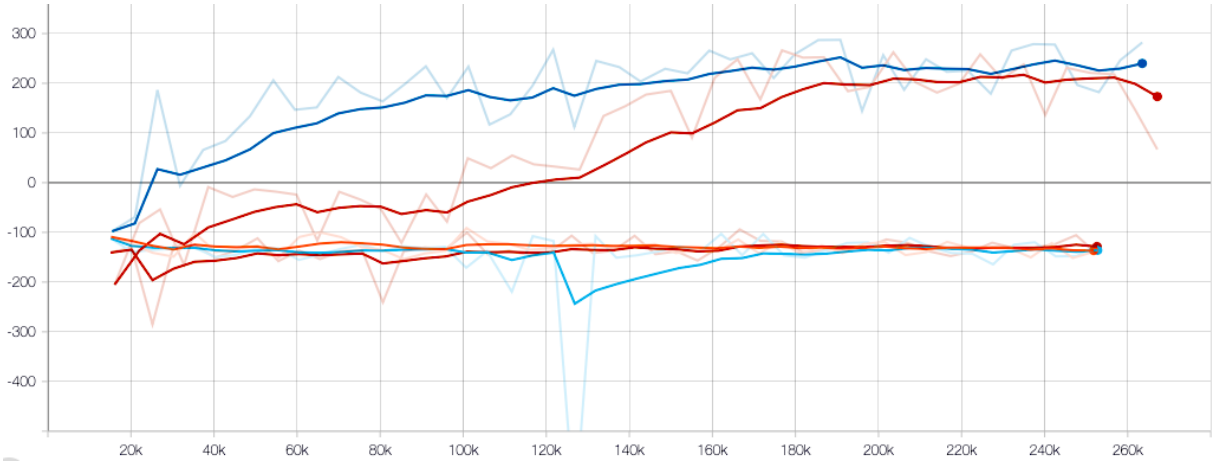


Deterministic policy π

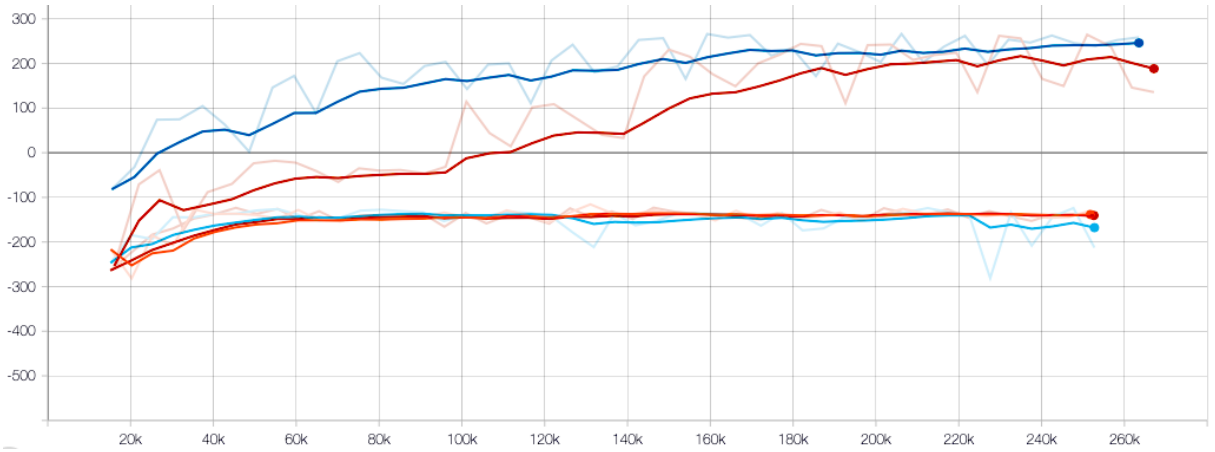


Stochastic policy π

`minimum replay buffer` = **300**

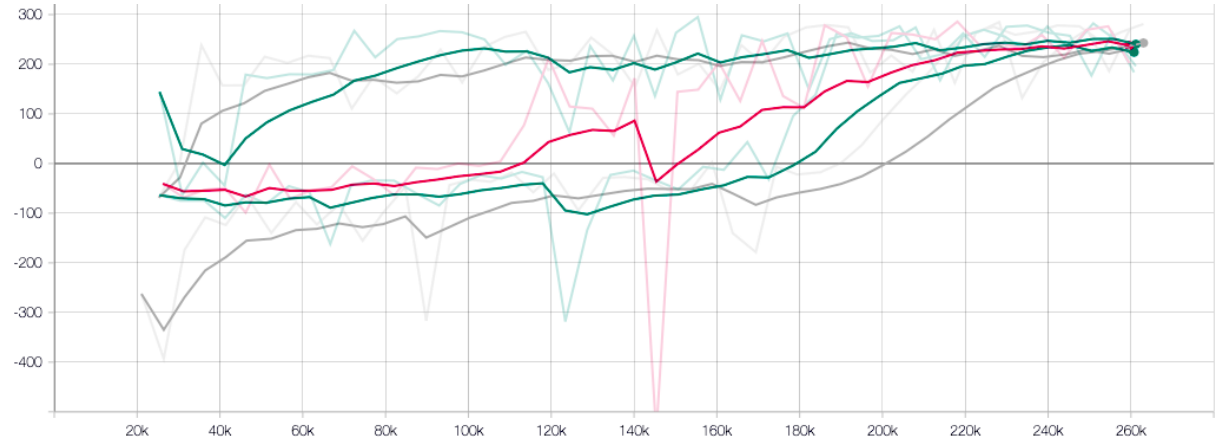


Deterministic policy π

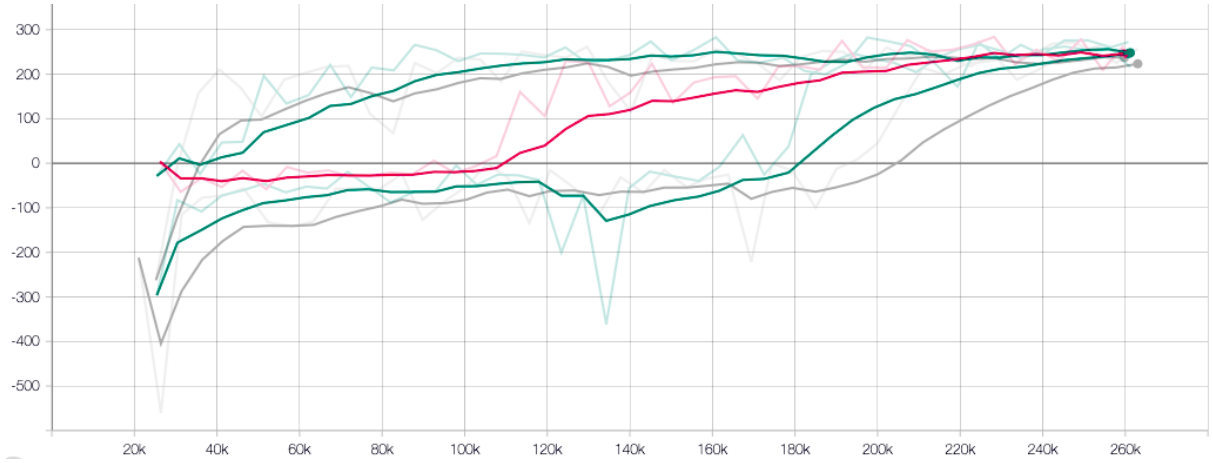


Stochastic policy π

`minimum replay buffer` $=$ **10000**



Deterministic policy π



Stochastic policy π

`minimum replay buffer` = **20000**

ExperimentSpec {

'Rerun tag': `LunarLander-EMA-MinPool-ModBuffer-min_pool_size=(300|20000|10000)`
'Max epoch': 50
'Timestep per epoch': 5000
'Discount factor': 0.99
'Learning rate': 0.003
'All Neural Nets topo': (200X200)
'Hidden layers activation': relu
'reward scaling': 40.0
'Replay buffer capacity': 50000
'Batch size in timestep': 200
'Minimum replay buffer size': [300|10000|20000]
'Random seed': None

}

Experimentation

It took a lot of time and hard work before starting to be able to exhibit the SAC paper results in a consistent manner, meaning over multiple rerun with the same experiment specification. It did so for numerous reasons, among those stated earlier:

- spotting the `replay buffer` sampling defects,
- tunning the `reward scaling` coefficient required for the current environment
- understanding the catastrophic effect that a too small `minimum pool size` hyperparameter value has on training a *Soft Actor-Critic* agent.

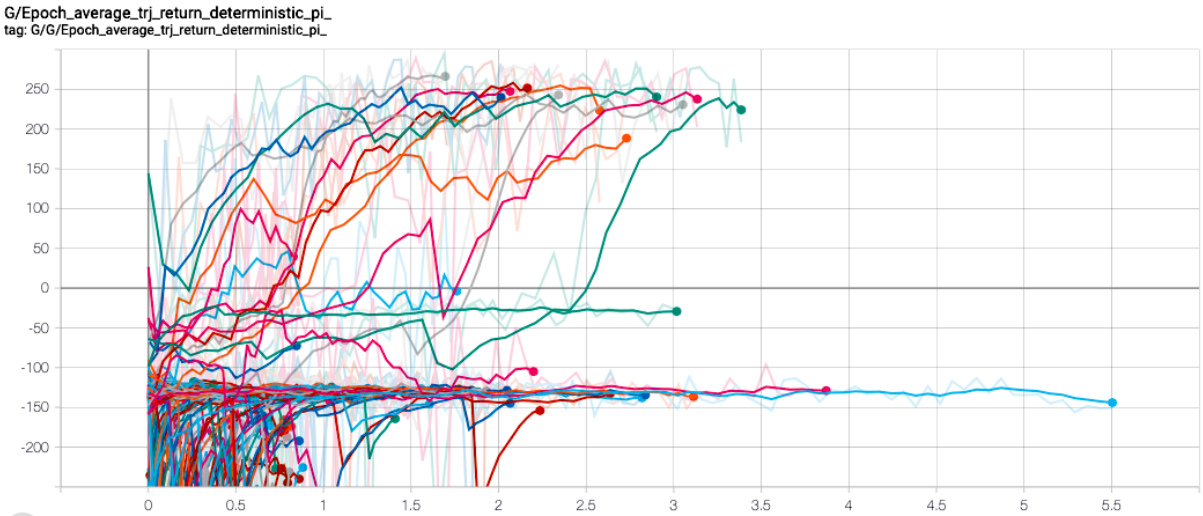
When silent bugs and bad hyperparameters walk hand in hand

What I did not mention about those three previously stated problems is that they had a compounding effect over the same resulting behaviour, naming: **optimizing to a local maxima** where learning an optimal path seems to be **depending on state initialization probabilities**.

This is the opposite of the expected behaviour of *Maximum Entropy RL*.

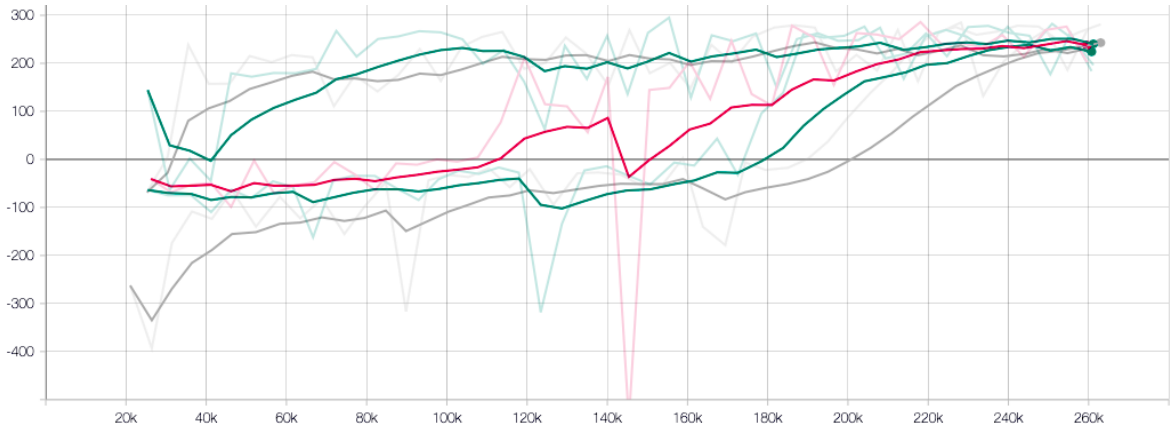
This compounding effect made finding its multiple causes harder, as solving one only diminished the undesired behaviour. The effect of that weak feedback and mix signal gave me doubt for a long time as I was unsure if I was wasting my time or if I was actually heading in the right direction even though I was solving issues.

I have to point out the fact that easier task like `Pendulum` did not highlight those problems, probably because of the smaller observation and action spaces. So once it worked on `Pendulum`, I thought that my implementation was working fine and that the lack of results on harder task was only due to hyperparameter tunning among those discuss in the literature. Also the experimentation cycle become slower and slower as environment get harder. As an example, a single training run on `LunarLanderContinuous-v2` over 250000 timesteps working those *SAC* 5 neural nets with each 200X200 neurons take between one and three and a half hour long on average.

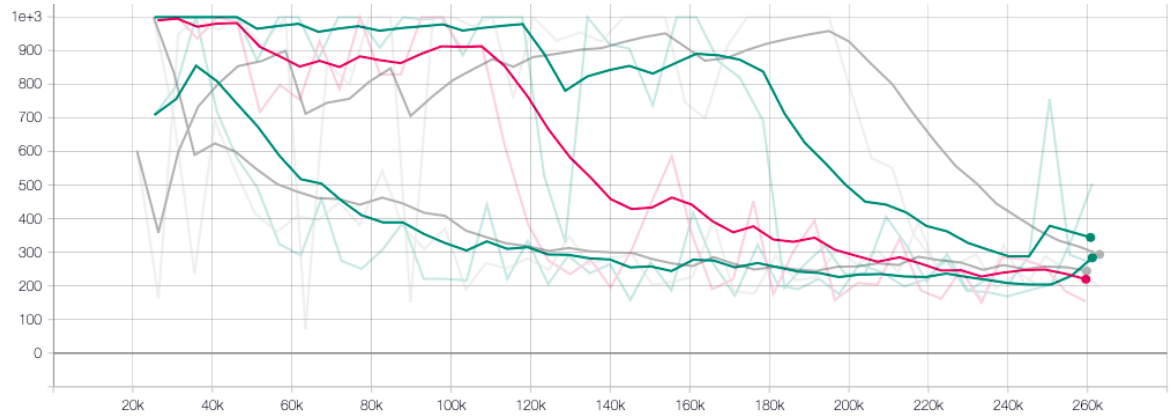


Average return vs **wall clock time**

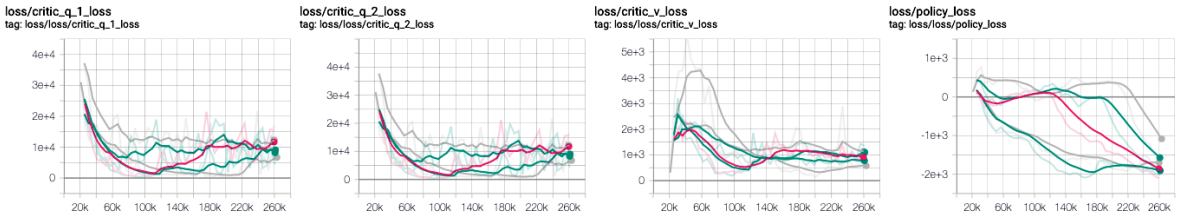
Nevertheless, after a lot of hard work, countless hours, sleepless night, many failed hypotheses and close to 500 experiments run, I am proud to say that I founded all the causes, I now understand why they were causing *SAC* to brake and I have a lot deeper appreciation of the *Maximum Entropy RL* mechanic in general.



Deterministic policy π



Trajectory lenght



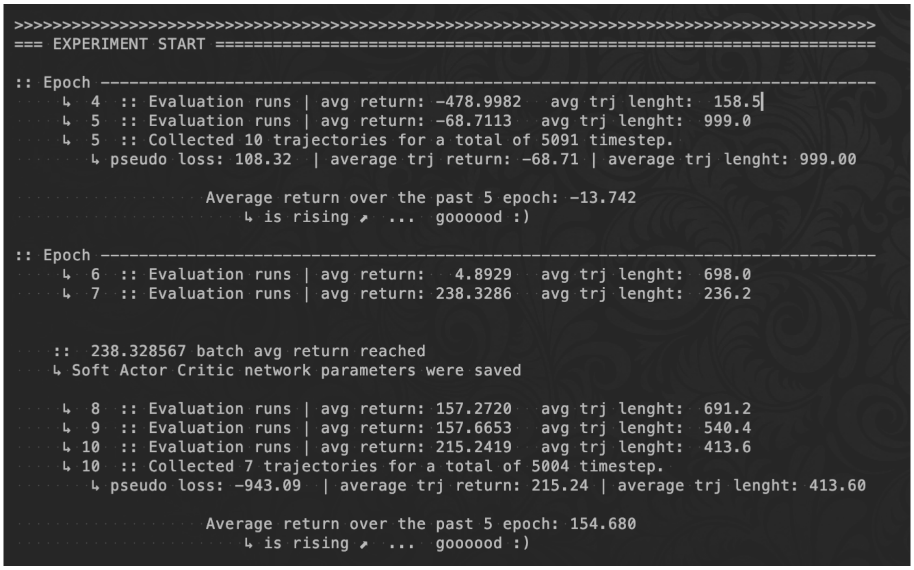
Losses

ExperimentSpec {

'Rerun tag': `LunarLander-EMA-MinPool-ModBuffer-min_pool_size=20000`
'Max epoch': 50
'Timestep per epoch': 5000
'Discout factor': 0.99
'Learning rate': 0.003
'All Neural Nets topo': (200X200)
'Hidden layers activation': relu
'Batch size in timestep': 200
'reward scaling': 40.0
'Replay buffer capacity': 50000
'Minimum replay buffer size': 20000
'Random seed': None

}

I still have a lot of experimentation to do on harder task but so far, my final iteration learn fast and gives powerful result. As you can appreciate, this following *SAC* training runs on `LunarLanderContinuous-v2` exceeded the reward goal in 7 `epoch` of the 50 planned.



```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
=== EXPERIMENT START =======================================================================

:: Epoch -----------------------------------------------------------------------------------
     ↳  4  :: Evaluation runs | avg return: -478.9982   avg trj lenght:  158.5
     ↳  5  :: Evaluation runs | avg return: -68.7113   avg trj lenght:  999.0
     ↳  5  :: Collected 10 trajectories for a total of 5091 timestep.
        ↳ pseudo loss: 108.32  | average trj return: -68.71 | average trj lenght: 999.00

                    Average return over the past 5 epoch: -13.742
                        ↳ is rising ↗  ...  goooood :)

:: Epoch -----------------------------------------------------------------------------------
     ↳  6  :: Evaluation runs | avg return:   4.8929   avg trj lenght:  698.0
     ↳  7  :: Evaluation runs | avg return: 238.3286   avg trj lenght:  236.2


    ::  238.328567 batch avg return reached
    ↳ Soft Actor Critic network parameters were saved

     ↳  8  :: Evaluation runs | avg return: 157.2720   avg trj lenght:  691.2
     ↳  9  :: Evaluation runs | avg return: 157.6653   avg trj lenght:  540.4
     ↳ 10  :: Evaluation runs | avg return: 215.2419   avg trj lenght:  413.6
     ↳ 10  :: Collected 7 trajectories for a total of 5004 timestep.
        ↳ pseudo loss: -943.09  | average trj return: 215.24 | average trj lenght: 413.60

                    Average return over the past 5 epoch: 154.680
                        ↳ is rising ↗  ...  goooood :)
```

A tool that I've built to custom display dynamically learning metric in the console
`visualisationtools.py` and `experiment_runner.py` modules under the `blocAndTools` package

Lesson learned & best practice

The importance of applying *Test-Driven development* best practice in RL

Why? First, because it **gives you confidence over those tested piece of code when tracking evil silent bug** causes. You know for sure they are working as expected so you can focus elsewhere. **Second,** because you can catch small mistakes early so you don't waste time later. **Third,** the "divide and conquer" approach: It's much easier to solve a problem over a small scope of your code base than it is on the entire code base.

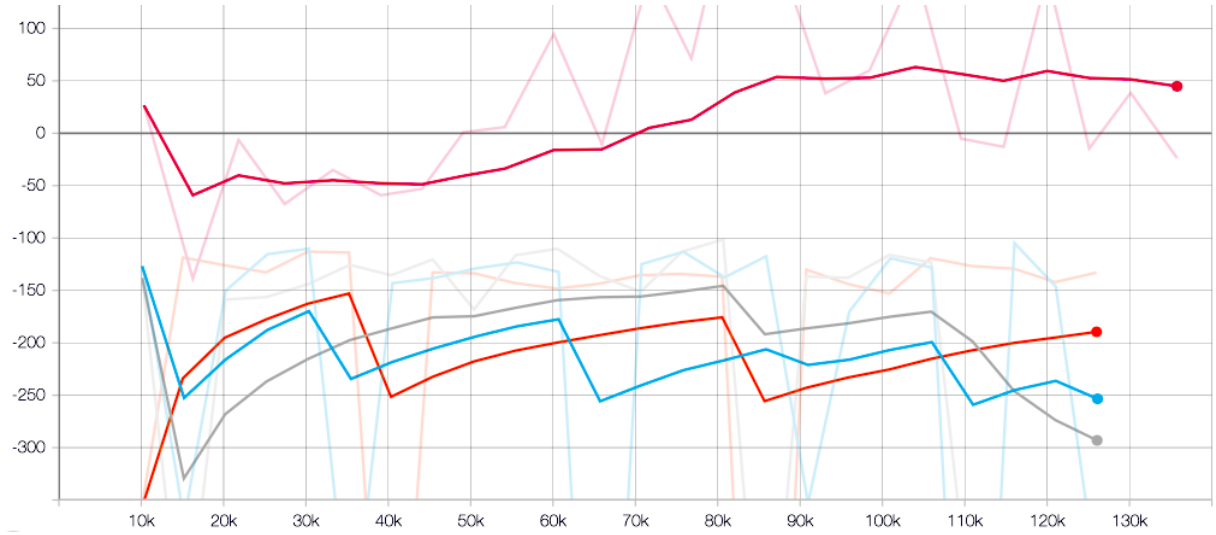
It's easy to get overconfident (as I did) and think that since it's "not a big challenge to implement", I could minimize or delay the *unit-test* part.

As an example, I realized at some point (very late in the process) that my *SAC* implementation seems to behave like it was getting trapped in local maxima more often than not. I thought for weeks that my math implementation was the problem and wasted a lot of valuable time and effort while looking at the wrong place.

The feedback I was getting from experimentation, over and over again, was that: for every 5 to 10 experimentation run, only one was actually learning something to an OK level. The others were never rising above a -150 return.

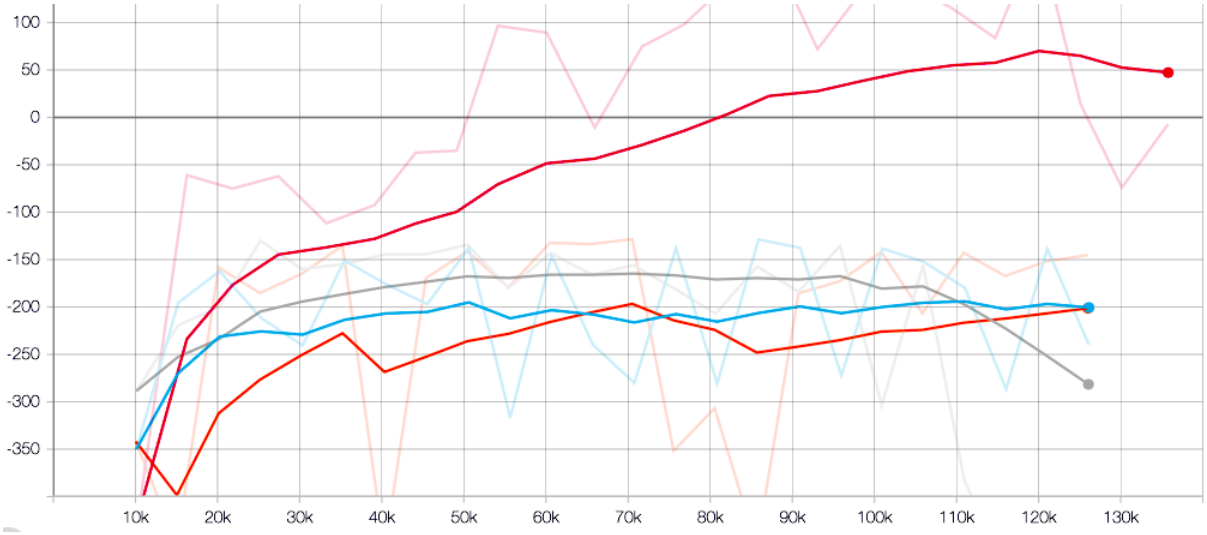
In general, for a experimentation rerun 5 times with the same hyperparameter settings, the average return was looking like this for deterministic policy:

(Note: LunarLander `reward threshold` is 200)



Deterministic policy π
Real average return over timestep
Experiment tag: `Exp-LunarLander-EMA-MedRewS`

and like this for stochastic policy



Stochastic policy π
Real average return over timestep
Experiment tag: `Exp-LunarLander-EMA-MedRewS`

with their learned policy π distribution for the final few `epoch` looking like the following

Policy π distribution

where the pink one is the only run that learned something useful.

Recall that *SAC* seeks to maximize the *return* and *entropy* so the final policy π distribution should be as wide as possible; thus it makes sense that the pink run looks flat.

The actual cause ... a bad implementation detail choice fixed with one method call change.

Why did it happen? Because I decided (unknowingly) to pass samples from the `replay buffer` to the `minibatch sampler` in ways that did not account for the speed at which those were randomly selected. So it was actually selecting random samples faster than it was swapping them in the `minibatch`. In effect, the `replay buffer` was, more often than not, passing identical `minibatch` to the learner periodically.

As a result, the algorithm was overfitting those same samples for multiple gradient step at the expense of the bigger picture, weakening its ability to handle local maxima.

Lesson RE-learned: Whenever I start thinking of myself as a "Top Gun coder" ... think twice, it's a trap.

Closing Thoughts

Where do we go from here? Even though the idea of using *entropy* or *Probabilistic Graphical Model* to solve the decision-making problem is not new in reinforcement learning, their application in practice, on the other hand, was not giving arguably better performance gain so far, until *Soft Actor-Critic* came out in 2018.

This approach is still very young and there is much more to explore on the subject:

- Is there more hyperparameter constraints that did not meet the eyes yet?
- Does it exhibits the same property regardless of the environment?
- To what extent can we model the policy in different ways?

The next few years will be interesting as we will probably discover a lot more on the subject and its capacity.

</> Future related projects:

- Following my observation on the minimum `replay buffer` size problem: explore thoroughly and prove rigorously *SAC replay buffer* requirements;
- Implement *transfer learning* capabilities to *SAC*: the ability to inject prior knowledge such as a policy distribution π_{prior} pre-trained on more general task;
- Implement *PEARL* [8], a Meta-Learning algorithm based on *Maximum Entropy RL*;
- Design an experiment that challenges adaptability skills and highlight agent robustness to adverse condition. My idea is to build on top of *Sven Niederberger* Gym **Rocket Lander** environment by making the state space more challenging and adding sub-goal. More importantly, adding a chaotic test environment to evaluate the robustness level of a trained agent over never-before-seen disaster scenario like those:
  - the loss of thrust during deceleration;
  - a moving landing pad;
  - strong side wind;
  - closing at the wrong angle;
  - losing control in a high angular velocity spin;
  - and so on;
- Dive into *Probabilistic Graphical Model* theory and applications;

BIBLIOGRAPHY

1. Haarnoja, T., Zhou, A., Abbeel, P. & Levine, S. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. arXiv: 1801.01290. http://arxiv.org/abs/1801.01290 (2018).
2. Fujimoto, S., van Hoof, H. & Meger, D. Addressing Function Approximation Error in Actor-Critic Methods. *CoRR* **abs/1802.09477.** arXiv: 1802.09477. http://arxiv.org/abs/1802.09477 (2018).
3. Levine, S. Reinforcement Learning and Control as Probabilistic Inference: Tutorial and Review. arXiv: 1805.00909. http://arxiv.org/abs/1805.00909 (2018).
4. Haarnoja, T. *et al.* Soft Actor-Critic Algorithms and Applications. arXiv: 1812.05905. http://arxiv.org/abs/1812.05905 (2018).
5. Haarnoja, T., Tang, H., Abbeel, P. & Levine, S. Reinforcement Learning with Deep Energy-Based Policies. arXiv: 1702.08165. http://arxiv.org/abs/1702.08165 (2017).
6. Silver, D. *et al.* Deterministic Policy Gradient Algorithms. *Proceedings of the 31st International Conference on Machine Learning (ICML-14)* **1.** http://proceedings.mlr.press/v32/silver14.pdf (2014).
7. Sutton, R. S. & Barto, A. G. *Reinforcement learning: An introduction* 2nd ed. (ed MIT Press) ISBN: 978-0262039246. http://incompleteideas.net/book/RLbook2018.pdf (Cambridge, MA, 2018).
8. Rakelly, K., Zhou, A., Quillen, D., Finn, C. & Levine, S. Efficient Off-Policy Meta-Reinforcement Learning via Probabilistic Context Variables. *CoRR* **abs/1903.08254.** arXiv: 1903.08254. http://arxiv.org/abs/1903.08254 (2019).