IFT-7014: Directed reading

Deep Reinforcement Learning – Actor-Critic

v:1.1

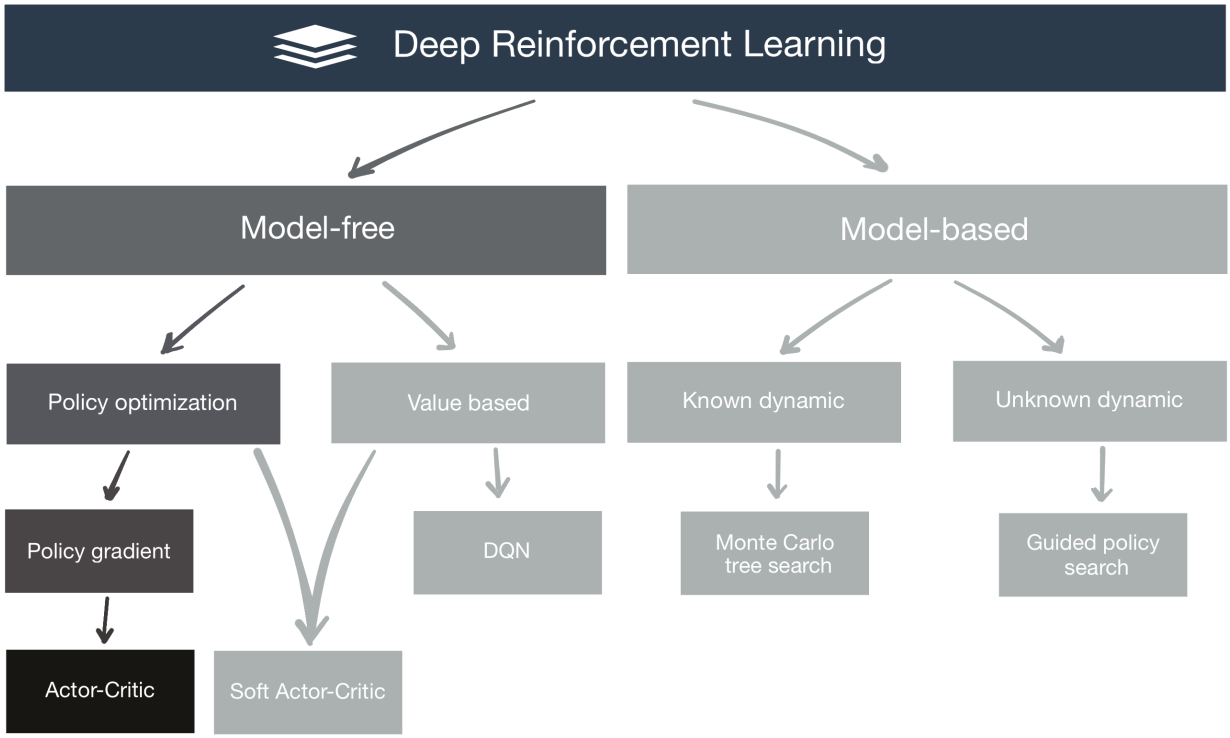
Luc Coupal
Université Laval
Montréal, QC, Canada
Luc.Coupal.1@uLaval.ca

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

November 7, 2019

CONTENTS



Advantage Actor-Critic

Advantage Actor-Critic method are close cousin of *Policy Gradient* class algorithm. The difference is that they use two neural networks instead of one: the **actor** who has the responsibility of finding the best action given a observation and the **critic** who has the responsibility of assessing if the actor does a good job.

The two main goals of this essay were to first, get a deeper understanding of *Actor-Critic* method theoric aspect and second, to acquire a practical understanding of it's beavior, limitation and requirement in order to work. In order to reach this second goal, I felt it was nescessary to implement multiple design & architectural variation commonly found in the litterature.

With this in mind, I've focused on the following practical aspect:

- **Algorithm type:** batch vs online;
- **Computation graph:** split network vs split network (with shared lower layer) vs shared network [1];
- **Critic target:** Monte-Carlo vs bootstrap estimate;
- **Math computation:** element wise vs graph computed;
- **Data collection:** $(\mathbf{s}_t, \mathbf{s}_{t+1}, r_t)$ vs $(\mathbf{s}_t, r_t, \widehat{Q}_t)$ vs $(\mathbf{s}_t, r_t, \widehat{A}_t)$;

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

In parallel, I writen a second essay *A reflexion on design, architecture and implementation details* where I go further in my study of somme aspect of DRL algortihm from a software engineering perspective applied to research by covering question like:

Does implementation details realy matters? Which one does, when & why?

Regarding *Reinforcement Learning (RL)* background:

In this essay, I will cover some *Deep Reinforcement Learning (DRL)* basic version of *Advantage Actor-Critic*, which are sometime refered as *one-step Actor-Critic*.

Regarding his *RL* cousin, Sutton & Barto point out that it's a

> *" analogue of the Temporal-Difference (TD) methode ... such as TD(0), SARSA and Q-Learning "*

Also, keep in mind that Advantage Actor-Critic methodes are a evolution of *Basic Policy Gradient* wich fall in the larger class of *Monte Carlo (MC)* algorithm.

Since *Advantage Actor-Critic* method and it's multiple variations leverage a lot of idea inherited from *RL-MC* and *RL-TD* approach, I founded that it helped a lot to dig deeper and build a good intuition on those two topics. In that regard, for a in depth comparaison, chapter 5: Monte Carlo Methods and chapter 6: Temporal-Difference Learning from the classic book *Reinforcement Learning: An Introduction* [2] are tough to beat.

However, it was recently highlithed by Amiranashvili et al. (2018) [3] that ***RL* theorical and empirical result might not systematicaly hold up in the context of *DRL***. They point out that modern problem tackled in *DRL* research deal with a much more rich and complex state space than those use for experiment at the time in *RL*. As such, those new empirical result show that, in the context of *DRL*, *MC* methods might be back being a top contender in certain setting. Those result contrast with how they where performing in the *RL* setting where they were left in the dust by *TD* methods.

I would argue that, maybe what was considered a hard problem in 1980-1990, like *cart-pole* (1983) [4] and *mountain car* (1990) [5], could be considered today like toy problem compare to solving an environment like *Starcraft* (2019) [6], their like two different animal.

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

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

Comment on notation:

Since their is 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 [2]

$(\mathcal{S}, \mathcal{A}, T, \mathcal{R})$	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 parameter $\theta = (\theta_1, \ldots, \theta_m)$
$\pi_\theta(\tau)$	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 informations 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 a action compose of multiple informations 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 term 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 (optimisation) the function to optimize, sometime synonimous with loss/cost/error function
$\mathbb{E}_{\tau \sim \pi_\theta(\tau)}[X]$	Expectation of X with respect to the trajectory τ conditionned on the policy π_θ
γ	discount factor penality to uncertainty of futur rewards; $0 < \gamma \leq 1$
G_t or r_t^γ	return or total discounted futur reward $G_t = r_t^\gamma = \sum_{k=t}^{\infty} \gamma^{k-t} r(\mathbf{s}_k, \mathbf{a}_k)$

Building bloc of Actor-Critic method

Let's recall somme key formula. The gradient of the objective of *Basic Policy Gradient* with **reward-to-go** $\widehat{Q}$ and **baseline** b is

$$\nabla_\theta J(\theta) \quad \approx \quad \frac{1}{N}\sum_{i}^{N}\sum_{t}^{\mathsf{T}} \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t}) \left(\widehat{Q}_{i,t} - b\right)$$

with

$$\widehat{Q}_{i,t} \quad = \quad \sum_{t'=t}^{\mathsf{T}} r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'})$$

the empirical return of a trajectory at timestep t and

$$b \quad = \quad \frac{1}{N}\sum_{i=1}^{N} r(\tau_i)$$

the average reward baseline.

Widening the scope of *Basic Policy Gradient* foresight:

The job of $\widehat{Q}_{i,t}$ Telling how good is taking action $\mathbf{a}_t$ in state $\mathbf{s}_t$ and going foward following π_θ

How? By estimating *the true expected reward-to-go* Q^π

$$\widehat{Q}_t \quad \approx \quad Q^\pi(\mathbf{s}_t, \mathbf{a}_t) \quad = \quad \sum_{t'=t}^{\mathsf{T}} \mathbb{E}_{\pi_\theta}\left[\, r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) \,|\, \mathbf{s}_t, \mathbf{a}_t \,\right]$$

Problem no 1: We know that **a single sample estimator of a expectation has high variance** and that a infinit sample estimator as lower variance. Since $\widehat{Q}_{i,t}$ **is a single sample estimate**, no wonder why Basic Policy Gradient method suffer from high variance problem.

Problem no 2: $\widehat{Q}_{i,t}$ only acount for one possible futur, the one that was sampled in that trajectory τ_i. In reality their is multiple possible futur. It' like taking advice from someone that is deeply stuburn and only foresight one possible scenario, it's not realy helpfull.

Why does it matters?

- Because lower variance imply convergence to a better sollution & potentialy faster convergence.
- Taking into acount more possible futur will give rise to better estimation.

Solution: Find a better estimator that **take more information into acount**

How: Instead of using the *single estimate reward-to-go* $\widehat{Q}_{i,t}$, lets use the *true expected reward-to-go* $Q^\pi(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})$ (aka the *Q-function*) and instead of using the *average reward baseline* b let use the *average true expected reward-to-go* as a baseline (aka the value-function) $V^\pi(\mathbf{s}_t)$.

The true expected reward-to-go (aka the *Q-function* or the *action-value-function*)

The total reward from taking $\mathbf{a}_t$ in $\mathbf{s}_t$ and going foward with respect to policy π_θ

$$Q^\pi(\mathbf{s}_t, \mathbf{a}_t) \quad = \quad \sum_{t'=t}^{\mathsf{T}} \mathbb{E}_{\pi_\theta} \left[r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) \,|\, \mathbf{s}_t, \mathbf{a}_t \right]$$

The *value-function*

The total reward from state $\mathbf{s}_t$ (the average *Q-function*) and going foward with respect to policy π_θ

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

Using those, let's define the ***Advantage*** *function*

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

Put together, starting from the gradient objective definition of *Basic Policy Gradient* with *reward-to-go* $\widehat{Q}$ and *baseline* b, we get the following:

$$\begin{aligned}
\nabla_\theta J(\theta) \quad &\approx \quad \frac{1}{N} \sum_i^N \sum_t^{\mathsf{T}} \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t}) \left(\widehat{Q}_{i,t} - b \right) \\
&\qquad\qquad \left\langle \text{ Subsituting } \quad \widehat{Q}_{i,t} := Q^\pi(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \text{ and } b := V^\pi(\mathbf{s}_{i,t}) \right\rangle \\
&\approx \quad \frac{1}{N} \sum_i^N \sum_t^{\mathsf{T}} \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t}) \left(Q^\pi(\mathbf{s}_t, \mathbf{a}_t) - V^\pi(\mathbf{s}_t) \right) \\
&\qquad\qquad \left\langle \text{ Definition of the Advantage } \right\rangle \\
&= \quad \frac{1}{N} \sum_i^N \sum_t^{\mathsf{T}} \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t}) \, A^\pi(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})
\end{aligned}$$

The *Advantage* A^π answer the question: for every action $\mathbf{a}_t$ in state $\mathbf{s}_t$, **how much better it would perform** compared to the average over all possible action in that state.

The name *Advantage* ***Actor-Critic*** come from the relation between those two:

- $\pi_\theta \longrightarrow$ the **actor**: the one reponsible for making acting decision in the environment;
- $V^\pi \longrightarrow$ the **critic**: the one responsible for evaluating if π_θ is doing a good job;

This new formulation of the Policy Gradient using the *Advantage* evidently leverage more information, but is it computable?

■ Computing the *Advantage* $A^\pi(\mathbf{s}_t, \mathbf{a}_t)$

Problem: Considering either the case of infinit horizon, contious action space or the typicaly extremely high-dimensionnal state spaces encountered in the context of DRL, computing the exact $A^\pi(\mathbf{s}_t, \mathbf{a}_t)$ become intractable in general.

Solution: We need to aproximate it and we need to do it **in a way that is a good compromise** between computing efficiency and exactitude.

How: By approximating $Q^\pi(\mathbf{s}_t, \mathbf{a}_t)$

$$
\begin{aligned}
Q^\pi(\mathbf{s}_t, \mathbf{a}_t) &= \sum_{t'=t}^{\mathsf{T}} \mathbb{E}_{\pi_\theta}\left[\, r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) \,|\mathbf{s}_t, \mathbf{a}_t \,\right] \\
&\qquad \langle \text{ Since the first term is already determined we can rewrite like this } \rangle \\
&= r(\mathbf{s}_t, \mathbf{a}_t) \;+\; \sum_{t'=t+1}^{\mathsf{T}} \mathbb{E}_{\pi_\theta}\left[\, r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) \,|\mathbf{s}_t, \mathbf{a}_t \,\right] \\
&\qquad \langle \text{ Rewriting } \rangle \\
&= r(\mathbf{s}_t, \mathbf{a}_t) \;+\; \mathbb{E}_{\mathbf{s}_{t+1}\sim p(\mathbf{s}_{t+1}|\mathbf{s}_t,\mathbf{a}_t)}\left[\, V^\pi(\mathbf{s}_{t+1}) \,\right] \\
&\qquad \langle \text{ Use a single sample estimate for just the next time step } {}^{*} \rangle \quad (1.1) \\
&\approx r(\mathbf{s}_t, \mathbf{a}_t) \;+ V^\pi(\mathbf{s}_{t+1})
\end{aligned}
$$

Then, the *Advantage* become

$$
\begin{aligned}
A^\pi(\mathbf{s}_t, \mathbf{a}_t) &= Q^\pi(\mathbf{s}_t, \mathbf{a}_t) \;-\; V^\pi(\mathbf{s}_t) \\
&\approx r(\mathbf{s}_t, \mathbf{a}_t) \;+\; V^\pi(\mathbf{s}_{t+1}) \;-\; V^\pi(\mathbf{s}_t)
\end{aligned}
$$

With this approximation of A^π, we now only need to care about V^π instead of both Q^π and V^π. This simplify a lot the algorithm.

Pro 👍:
- The scope of possibility is smaller;
- It's simpler to learn a function of $\mathbf{s}_t$ than $\mathbf{s}_t, \mathbf{a}_t$;
- We only need to compute $V^\pi(\mathbf{s})$;

Con 👎: We lose a little bit of information,
but it's only one timestep worth of information.

It appears to be a good compromise ... so how do we find V^π?

*This is also refered as *one-step return* in the litterature. See: Reinforcement Learning: An Introduction, section 15.5: Actor-Critic Methods by Sutton & Barto [2]

■ Learning $V^\pi(\mathbf{s})$: Policy evaluation using a neural net

Policy evaluation is the process of fitting the value-function $V^\pi(\mathbf{s})$ to a given policy π.

In the context of Deep Reinforcement Learning, the Actor-Critic approach to policy evaluation is to approximate the value-function using a second neural net

$$\widehat{V}_\phi^\pi(\mathbf{s}_t) \approx V^\pi(\mathbf{s}_t)$$

with parameter ϕ.

Job of $\widehat{V}_\phi^\pi(\mathbf{s}_t)$: Evaluate a given policy π_θ ... not to improve it (that's the job of the other network).

What it does: Since a neural net is a universal function approximator, then for every state $\mathbf{s}_t$ and $\mathbf{s}'_t$ sharing similar feature, $\widehat{V}_\phi^\pi$ should output similar value

$$\mathbf{s}_t \approx \mathbf{s}'_t$$

$$\widehat{V}_\phi^\pi(\mathbf{s}_t) \approx \widehat{V}_\phi^\pi(\mathbf{s}'_t)$$

Under the hood, the neural net is averaging multiple trajectories in a way that it give him the hability to generalize over never seen before trajectory. But, it do it at a cost. Since it's a function approximation of V^π, it's not exact anymore, so it will introduce some bias, especialy earlier during training.

Pro 👍:
- It compute a estimation of V^π;
- The critic A^π will still be able to do is job of reducing the variance;

Con 👎: It introduce some bias because the neural net is not a perfect approximation.

In the end we rewrite the *Advantage* and the *gradient of the objective* like this

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})\,\overbrace{\left(r(\mathbf{s}_{i,t},\mathbf{a}_{i,t}) + \widehat{V}_\phi^\pi(\mathbf{s}_{i,t+1}) - \widehat{V}_\phi^\pi(\mathbf{s}_{i,t})\right)}^{\widehat{A}^\pi(\mathbf{s}_{i,t},\mathbf{a}_{i,t})}$$

■ Training $\widehat{V}_\phi^\pi$

Training $\widehat{V}_\phi^\pi$ is a supervised regression problem. So we need to define

- a training set: $\mathcal{D}^{\text{train}} = \{ (\mathbf{x}_i, y_i) \}$ with input data $\mathbf{x}_i$ and label y_i (aka target values);
- and a loss function: $L(f(\mathbf{x}_i \,|\, \theta), y_i)$.

How do we define the label y_i ? There is many ways to do this. Chosing which one realy depend on the DRL task to solve, but in the end, what we want is to substitute y_i with some kind of surogate for V^π. A good estimate would be an average over multiple trajectories of *reward-to-go*

$$V^\pi(\mathbf{s}_t) \;\approx\; \frac{1}{N} \sum_{i=1}^{N} \sum_{t'=t}^{\mathsf{T}} r(\mathbf{s}_{t'}, \mathbf{a}_{t'})$$

but this require us to have multiple completed trajectories and it imply that we delay learning $\widehat{V}_\phi^\pi(\mathbf{s}_t)$ to the end of each of those batch collection cycles. It's good in some setting but it's restrictive.

Do we have other options?

■ Function approximation with *Monte Carlo target values*

While we are inside the trajectories collection cycle, we have acces to $\widehat{Q}_{i,t} = \sum_{t'=t}^{\mathsf{T}} r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'})$ at each trajectories ends. We could use those as an estimate of $V^\pi(\mathbf{s}_t)$. It' not as good but it could be good enough.

Why not use this estimate as label to define our supervised regression problem:

$$y_i \;:=\; \widehat{Q}_{i,t} \;\approx\; V^\pi(\mathbf{s}_t)$$

The input $\mathbf{x}$: the state t of the i-em sample ◄ ► The labels y_i as *Monte Carlo target values*

$$\mathcal{D}^{\text{train}} \;=\; \left\{ \left(\mathbf{s}_{i,t} \,,\, \widehat{Q}_{i,t} \right) \right\}$$

$$L\left(\widehat{V}_\phi^\pi(\mathbf{s}_i) \,\middle|\, \widehat{Q}_i \right) \;=\; \frac{1}{2} \sum_{i=1}^{N} \left\| \widehat{V}_\phi^\pi(\mathbf{s}_i) \;-\; \widehat{Q}_i \right\|^2$$

Caveat: This approch use a single sample estimate, so for the same reason stated earlier, we are loosing information (see: Widening the scope of Basic Policy Gradient foresight/Problem no 2).

Is their a way that we would not be constraint to **batch learning**
and we could do **online learning** instead?

■ Function approximation with bootstrap estimate

" They learn a guess from a guess — they bootstrap."

Sutton & Barto †

What about the ideal target value $V^{\pi}(\mathbf{s}_{i,t}) = \sum_{t'=t}^{\mathsf{T}} \mathbb{E}_{\pi_\theta}\left[\, r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) \,|\mathbf{s}_{i,t}\,\right]$.

Could we get closer to it?

$$
\begin{aligned}
V^{\pi}(\mathbf{s}_{i,t}) \quad &= \quad \sum_{t'=t}^{\mathsf{T}} \mathbb{E}_{\pi_\theta}\left[\, r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) \,|\mathbf{s}_{i,t}\,\right] \\
&\qquad\qquad \langle \text{ Use the same single sample estimate trick as in (1.1) } \rangle \\
&\approx \quad r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \;+\; V^{\pi}(\mathbf{s}_{i,\,t+1}) \\
&\qquad\qquad \langle \text{ Use what the value-function approximator has learned so far } \rangle \\
&\approx \quad r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \;+\; \widehat{V}^{\pi}_{\phi}(\mathbf{s}_{i,t+1})
\end{aligned}
$$

So now we can define our supervised regression problem as

$$
y_i \;:=\; r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \;+\; \widehat{V}^{\pi}_{\phi}(\mathbf{s}_{i,t+1}) \;\approx\; V^{\pi}(\mathbf{s}_t)
$$

The input $\mathbf{x}$: the state t of the i-em sample ◄ ► The labels y_i as a *Bootstrap estimate*

$$
\mathcal{D}^{\text{train}} \;=\; \left\{ \left(\mathbf{s}_{i,t}\,,\; r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \;+\; \widehat{V}^{\pi}_{\phi}(\mathbf{s}_{i,\,t+1}) \right) \right\}
$$

$$
L\left(\widehat{V}^{\pi}_{\phi}(\mathbf{s}_i) \,\middle|\, y_i \right) \;=\; \frac{1}{2} \sum_{i=1}^{N} \left\| \widehat{V}^{\pi}_{\phi}(\mathbf{s}_i) \;-\; \left(r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \;+\; \widehat{V}^{\pi}_{\phi}(\mathbf{s}_{i,\,t+1}) \right) \right\|^2
$$

Why does it work?

Again, it's a compomise. We use a small amount of collected knowledge from the environment while leveraging what the value-function approximator as already learned. So it

- introduce some bias because of the nature of this approximator 👎;
- but still reduce variance 👍;

Note: This is called a *bootstrap estimate* because the *value-function approximator* is re-using it's own estimate along the learning process, therefore lifting himself up by pulling the strap of it's own boot.

†Reinforcement Learning: An Introduction, chapter 6: Temporal-Difference Learning, page 124 [2]

(Key idea) Building bloc of *Advantage Actor-Critic* method

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{\mathsf{T}} \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})\, \widehat{A}^\pi(\mathbf{s}_{i,t},\mathbf{a}_{i,t})$$

Learn two neural network:

- $\pi_\theta \longrightarrow$ the **actor**: the one reponsible for making acting decision in the environment;
- $\widehat{V}^\pi_\phi \longrightarrow$ the **critic**: the one responsible for evaluating if π_θ is doing a good job;

Make approximation of:

$$Q^\pi(\mathbf{s}_t,\mathbf{a}_t) \approx r(\mathbf{s}_{i,t},\mathbf{a}_{i,t}) + \widehat{V}^\pi_\phi(\mathbf{s}_{i,t+1})$$
$$V^\pi(\mathbf{s}_t,\mathbf{a}_t) \approx \widehat{V}^\pi_\phi(\mathbf{s}_{i,t})$$

Use a Advantage function: It tell, for every action $\mathbf{a}_t$ in state $\mathbf{s}_t$, **how much better it would perform** compared to the average over all possible action in that state.

$$\widehat{A}^\pi(\mathbf{s}_{i,t},\mathbf{a}_{i,t}) = r(\mathbf{s}_{i,t},\mathbf{a}_{i,t}) + \widehat{V}^\pi_\phi(\mathbf{s}_{i,t+1}) - \widehat{V}^\pi_\phi(\mathbf{s}_{i,t})$$

Learn $\widehat{V}^\pi_\phi$ using either:

Monte Carlo target values $y_i := \widehat{Q}_{i,t}$

or ***Bootstrap estimate*** $y_i := r(\mathbf{s}_{i,t},\mathbf{a}_{i,t}) + \widehat{V}^\pi_\phi(\mathbf{s}_{i,t+1})$

Algorithm anatomy: Actor-Critic method

What it does: Like *Basic Policy Gradient*, it optimize directly the policy + the added benefit of using a *value-function* approximator.

Meaning:
- It as **lower variance** then Basic Policy Gradient
- It's **not unbias anymore** because the neural net is a approximation

value-function approximator for the next state ◄ ► *value-function* approximator for current state

The *Advantage* part: $r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) + \widehat{V}^{\pi}_{\phi}(\mathbf{s}_{i,t+1}) - \widehat{V}^{\pi}_{\phi}(\mathbf{s}_{i,t})$

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{\mathsf{T}} \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})\, \widehat{A}^{\pi}(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})$$

Policy Gradient part

From formula to algorithm

There is a lot of different design & architecture related to *Actor-Critic* method. Some author actualy use term *Actor-Critic class* [‡] which make sense to me now, given the number of improvement and variation I have readed about since I've started studying on the topic. I must admit it was a little bit overwelming at first, especialy at moment my conceptual tree of the subject started to morph to a conceptual graph. In order to help structure that conceptual graph, I feel it's natural to pull the *algorithm type* node at the top and make all other aspect fall under it.

There is two *algorithm type*:

1. **Batch *Actor-Critic*** $\Longrightarrow$ do computation on a big batch of trajectories;
2. **Online *Actor-Critic*** $\Longrightarrow$ execute one step in the environment, then compute;

Choosing one over the other will have a incidence on *computation graph architecture*, available *critic target type* options, *data collection requirement* and available *improvment method* option.

Both *algorithm type* work very well [1, 7], but they have different requirement and limitation. We will cover those applied consideration later in Advantage Actor-Critic - In Practice.

For now let's compare batch vs online high level algorithm.

[‡] Daniel Seita as a very elegant way of explaning this:

> " The term "actor-critic" is best thought of as a framework or a class of algorithms satisfying the criteria that there exists parameterized actors and critics"

Algorithm 1: *Batch Actor-Critic algorithm*

Note: to keep the pseudocode concise, iteration loop over timestep t are implicit.

repeat

  repeat

    1. **sample** $\tau_i \sim \pi_\theta$ **and collect to** *batch* "run the policy"

  until i = BATCH SIZE MAX

  repeat

    2. **fit** $\widehat{V}_\phi^\pi$ using *Monte-Carlo target* or *bootstrap estimate*

  until j = CRITIC MAX FIT ITERATION

  foreach τ_i **in** *batch* **do**

    3. **evaluate** $\widehat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i) = r(\mathbf{s}_i, \mathbf{a}_i) + \widehat{V}_\phi^\pi(\mathbf{s'}_i) - \widehat{V}_\phi^\pi(\mathbf{s}_i)$ "estimate the return"

  4. $\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \log \pi_\theta(\mathbf{a}_i|\mathbf{s}_i)\, \widehat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i)$

  5. $\theta \longleftarrow \theta + \alpha \nabla_\theta J(\theta)$ "improve the policy"

  reset *batch*

until *learning goal reached* **or** MAX EPOCH *reached*

Performing computation over a **batch** require collecting a large amount of samples. This can become a problem when you're dealing with environment where producing a single FULL trajectory take a considerable amount of time or imply some risk eg. training a quadrocopter to do aerobatic manoeuvre.

Also collection & retrival of samples require designing so kind of handling strategy, especialy with some type of observation space eg. dealing with the *MsPacman-v0* Gym environment where each observation $\mathbf{s}_t$ is a RGB image represented as an (260, 160, 3) array take a lot more space on RAM than a 4 digits observation like in *CartPole-v1* Gym environment Box(4) .

Algorithm 2: *Online Actor-Critic algorithm*

repeat

  1. **take one step** $\mathbf{a}_t \sim \pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ **and collect** $(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1}, r_t)$ "run the policy"

  2. **update** $\widehat{V}_\phi^\pi$ **with bootstrap estimate target** $r_t + \widehat{V}_\phi^\pi(\mathbf{s}_{t+1})$

  3. **evaluate** $\widehat{A}^\pi(\mathbf{s}_t, \mathbf{a}_t) = r_t + \widehat{V}_\phi^\pi(\mathbf{s}_{t+1}) - \widehat{V}_\phi^\pi(\mathbf{s}_t)$ "estimate the return"

  4. $\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t)\, \widehat{A}^\pi(\mathbf{s}_t, \mathbf{a}_t)$

  5. $\theta \longleftarrow \theta + \alpha \nabla_\theta J(\theta)$ "improve the policy"

until *learning goal reached* **or** MAX EPOCH *reached*

Performing **online** computation require doing a lot of gradient step, which mean that computation speed might become a limitation without the proper hardware. Sampling can also be a bottle neck for that algorithm type if the speed at which the state $(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1}, r_t)$ are produce is slower than performing both update π_θ and $\widehat{V}_\phi^\pi$.

(Key idea) Algorithm - Actor-Critic method

It's *Basic Policy Gradient* with *value-function* approximator:

- It as **lower variance**
- It's **not unbias anymore**

There is numerous architecture variation and implementation design choice;

Two algorithm type:

1. **batch *Actor-Critic*** $\Longrightarrow$ delay learning to epoch end ;
2. **online *Actor-Critic*** $\Longrightarrow$ learn at each timestep;

Improvment

Actor-Critic class is very diverse. Let's mention that there is other very important improvement and architectural aspect not covered in this essay:

Computing the return: *one-step return* vs *n-step returns* vs weighted combination *n-step returns* [7] (GAE)

Paralellel architecture: *online synchronized* parallel (A2C) vs *online asynchronous* parallel [1] (A3C)

Advantage Actor-Critic - In Practice

›_ From theory to practice

Following my essay *A reflexion on design, architecture and implementation details*, I came to realize that learning to recognize when an implementation detail become important or critical was a valuable skill. Since earning that skill can only be done by facing challenge I decided to implement the following variations of *Actor-Critic* algorithm:

1. Batch algorithm, split network with Monte-Carlo target or Bootstrap estimate target;
2. Batch algorithm, bootstrap estimate target with split network or shared network;
3. Online algorithm with split network or shared network or split network with shared lower layer graph;
4. Graph computed $\widehat{Q}^{\pi}$ or element wize computed $\widehat{Q}^{\pi}$
5. Discounted or undiscouted $\widehat{Q}^{\pi}$

This process helped me a lot to understand the various key pieces, experiment different data handling requirement and to sharped my habilites at planning DRL project.

</> Choosing an algorithm type

Like I said earlier, both **batch** and **online** algorithm perform well but each have different limiting factor. So it's important to consider the way they work and evaluate the pro & con before making a design decision given the folowing:

- **batch** *Actor-Critic* require collecting and handling a large amount of new samples at each update cycle, but need relatively few of those cycles to converge;
- **online** *Actor-Critic* require doing a lot of graph updates, one at each collected step, but require storing a minimum amout of data along the way;

As an example, training a **robot hand** to fold a t-shirt **knowing that you only have one robot is a critical detail** as the simulator will become a sampling bottleneck. So choosing a batch *Actor-Critic* design might mean days before getting some result on a ongoing experiment since learning is delayed at batch collection end.

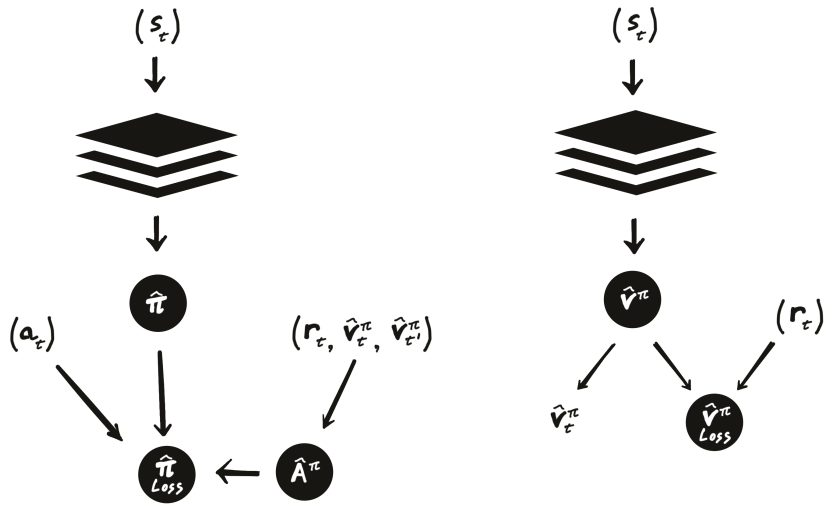
Also you have to consider the impact the choosen *algorithm type* will have over other design aspect like choosing a *critic target type*. **Batch** *Actor-Critic* will give you the option of using either Monte-Carlo target or bootstrap estimate target to fit $\widehat{V}_{\phi}^{\pi}$. On the other hand, **online** *Actor-Critic* will restrict you to bootstrap estimate target only.

Choosing the amount of critic training iteration is also a design choice but it can be made has an hyper-parameter. Also whether the critic training is done before or after the policy update is apparently also a design choice but it's unclear to me if it has any repercution on performance.

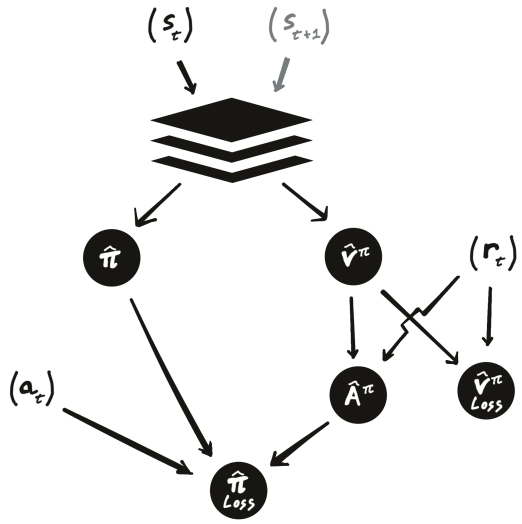
</> Choosing a computation graph architecture

There is 3 high level network architecture for *Actor-Critic* class:

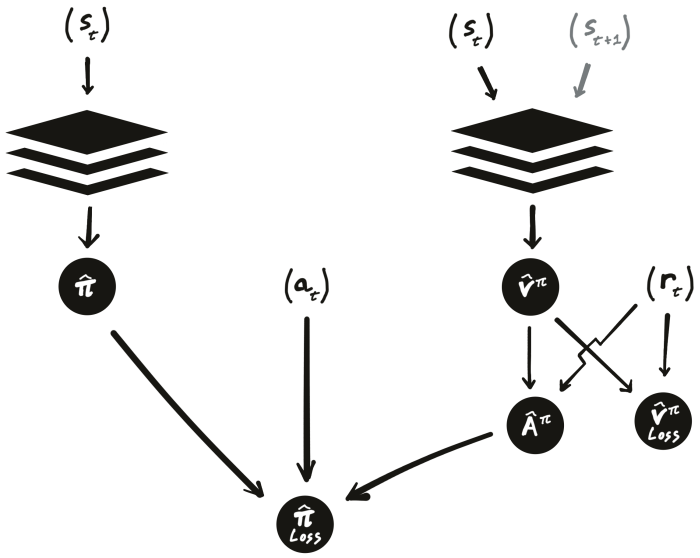
1. *Split network*: Straight foward to set up, stabilize faster during training;
2. *Shared network*: Can learn faster (. . . in principle) but is harder to stabilize in training;
3. *Split network with shared graph lower layer*: Require the simplest data manipulation strategy of all 3;



Split network



Shared network



Split network (with shared graph lower layer)

From my (limited) experimentation on *CartPole-v0*:

1. *Split network*: was working with almost any hyperparameter and seems more robust to some setting change.
2. *Shared network*: was indead a lot harder to stabilize in training, was very sensible to hyperparameter small modification and was working on a very narrow range of hyperparameter;
3. *Split network with shared graph lower layer*: The data handling is minimal since all the math computation is handle in the computation graph. It was the easiest and fastest way to implement the algorithm. On the other hand it's maybe a less versatile design when it come to implement *n-step return* improvement variant.

</> Choosing the critic target

There is a tradeof for both target type:

- Monte-Carlo target is unbias in expectation but has more variance. It also appear to work with a wider range of hyperparameter than bootstrap estimate;
- Bootstrap estimate has less variance which mean that π_θ might converge to a better solution or that we could use the bigger learning rate which mean a faster convergence. But all this comme at the cost of introducing bias, especialy at a earlier training stage;

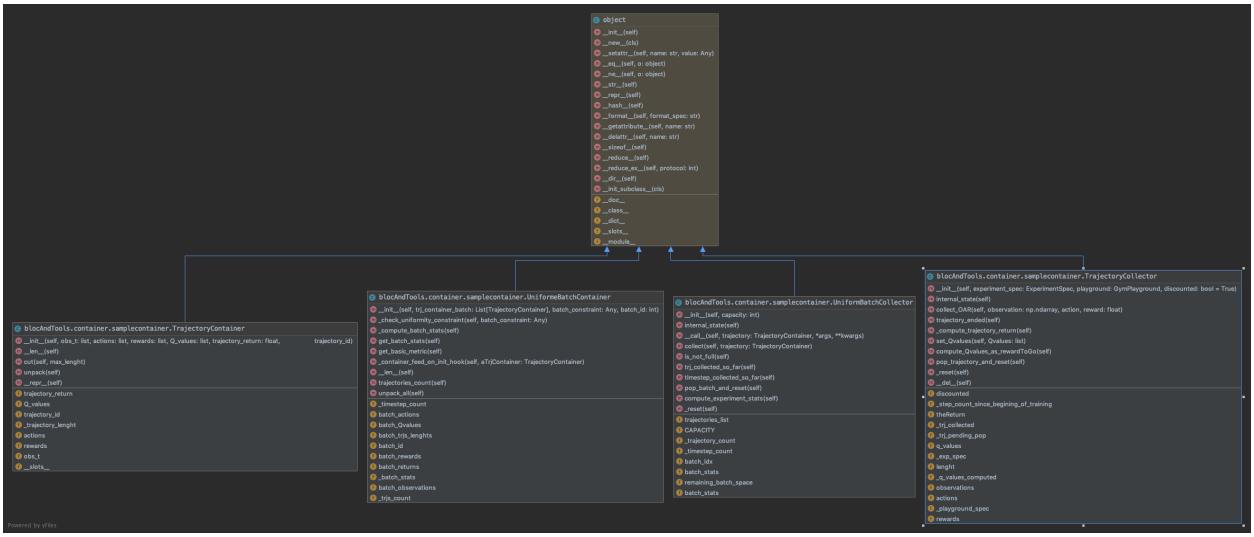
</> Choosing a data handling strategy

There is multiple possible approach regarding data collection and retrivial but the most appropriate one will depend on multiple factor like the algorithm type, network architecture and the setting. As an example those are alternative strategy related to computing the Advantage:

Collected data		Advantage computation		Result	
$\left(r_t,\, \mathbf{s}_t,\, \mathbf{s}_{t+1}\right)$	$\longmapsto$	$\widehat{A}^{\pi}\left(r_t + \widehat{V}_{\phi}^{\pi}(\mathbf{s}_{t+1}) - \widehat{V}_{\phi}^{\pi}(\mathbf{s}_t)\right)$	$=$	$\widehat{A}_t^{\pi}$	(1)
$\left(r_t,\, \widehat{V}_t^{\pi},\, \widehat{V}_{t+1}^{\pi}\right)$	$\longmapsto$	$\widehat{A}^{\pi}\left(r_t + \widehat{V}_{t+1}^{\pi} - \widehat{V}_t^{\pi}\right)$	$=$	$\widehat{A}_t^{\pi}$	(2)
$\left(\widehat{Q}_t^{\pi}, \mathbf{s}_t\right)$	$\longmapsto$	$\widehat{A}^{\pi}\left(\widehat{Q}_t^{\pi} - \widehat{V}_{\phi}^{\pi}(\mathbf{s}_t)\right)$	$=$	$\widehat{A}_t^{\pi}$	(3)
$\left(r_{[1:\mathsf{T}]},\, \mathbf{s}_{[1:\mathsf{T}]},\, t\right)$	$\longmapsto$	$\widehat{A}^{\pi}\left(r_{[t]} + \widehat{V}_{\phi}^{\pi}(\mathbf{s}_{[t+1]}) - \widehat{V}_{\phi}^{\pi}(\mathbf{s}_{[t]})\right)$	$=$	$\widehat{A}_t^{\pi}$	(4)
$\left(r_{[1:\mathsf{T}]},\, \widehat{V}_{[1:\mathsf{T}]}^{\pi},\, t\right)$	$\longmapsto$	$\widehat{A}^{\pi}\left(r_{[t]} + \widehat{V}_{[t+1]}^{\pi} - \widehat{V}_{[t]}^{\pi}\right)$	$=$	$\widehat{A}_t^{\pi}$	(5)
$\left(\widehat{Q}_{[1:\mathsf{T}]}^{\pi}, \mathbf{s}_{[1:\mathsf{T}]},\, t\right)$	$\longmapsto$	$\widehat{A}^{\pi}\left(\widehat{Q}_{[t]}^{\pi} - \widehat{V}_{\phi}^{\pi}(\mathbf{s}_{[t]})\right)$	$=$	$\widehat{A}_t^{\pi}$	(6)

Strategy 1 is use in *online Actor-Critic* algorithm with a *shared network* design.
Strategy 2 is better suited for setting where handling observation $\mathbf{s}_t$ are costly.
Strategy 3 and 6 are well suited for *bootstrap estimate critic target* design since $\widehat{Q}^{\pi}$ is already computed.
Strategy 4 is used in *batch Actor-Critic* algorithm with a *split network with shared lower layer graph* design
Strategy 5 is used for design that leverage *n-step return* or *GAE* improvment technique.

To manage the changing requirement posed by the different *Actor-Critic* variation I was implementing, I've opted for a *object oriented* approch to data handling. It added a overhead at first, but once the parent classes were done and well tested, it gave me the hability to create subclasses variations effectively and the freedom to experiment with different design faster. My basic idea was to separate the responsabilities in the following way: a trajectory collector, a trajectory container, a batch collector and a batch container.



›_ Experimentation

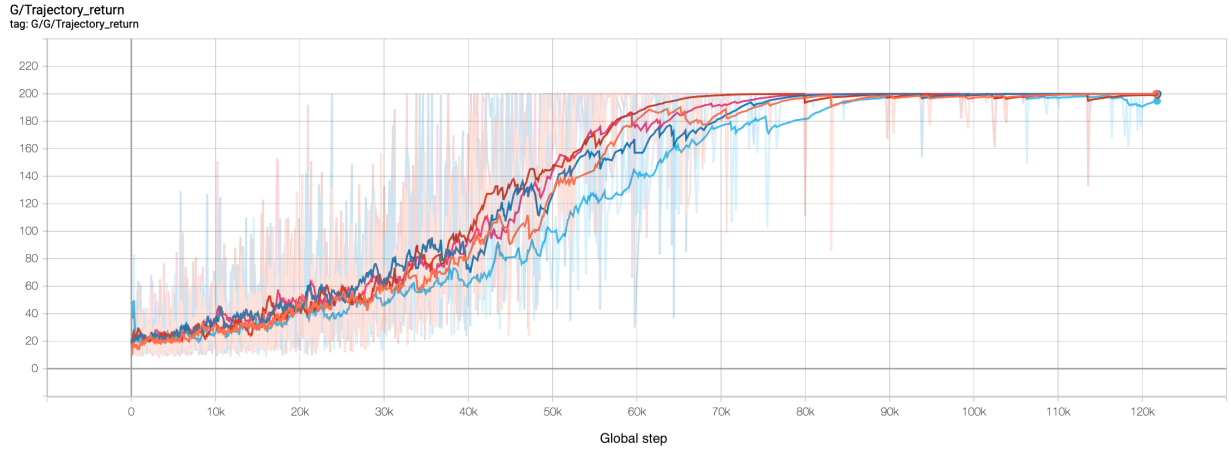
All experiments were run 5 times each with no random seed.
They were all executed with the Gym environment 'CartPole-v0' (expected return goal of 200).

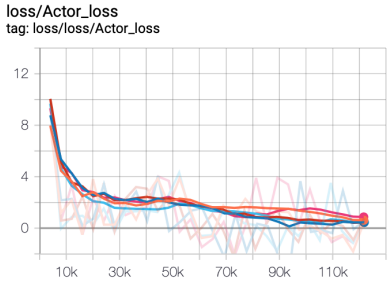
Those were the most interesting one:

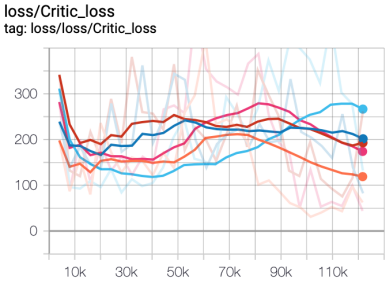
1. Batch algorithm, split network with discounted Monte-Carlo target
2. Batch algorithm, split network with undiscounted Monte-Carlo target
3. Batch algorithm, split network with Bootstrap estimate target
4. Batch algorithm, shared network, mini-batch, with Bootstrap estimate target
5. Online algorithm, split small network with discounted target
6. Online algorithm, split network hl(16, 32, 256) with discounted target
7. Online algorithm, shared network hl(32, 64, 256) with discounted target
8. Online algorithm, split network (shared lower layer) hl(16, 32, 32) with discounted target

</> Batch algorithm, split network with discounted Monte-Carlo target

Observation: Design with the fastest convergence. Also the most stable design once it has converge.







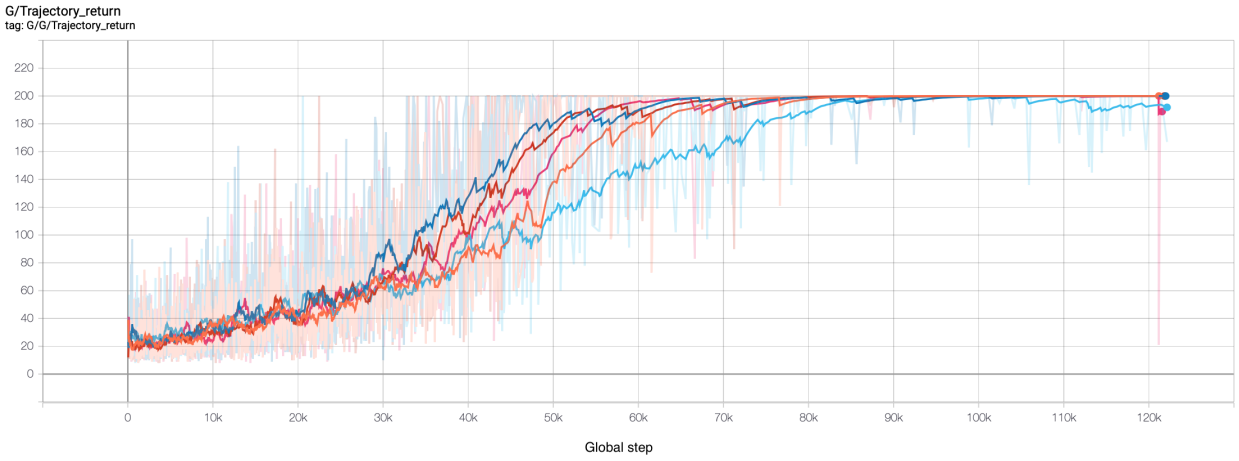
ExperimentSpec{

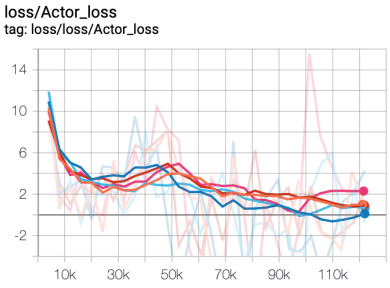
'Rerun tag': BMCSPL-A
'Batch size in ts': 4000
'Max epoch': 30
'Discout factor': 0.99
'Learning rate': 0.01
'Discounted reward to go': True
'Theta nn h layer topo': (32, 32)
'Theta hidden layers activation': relu
'Critic learning rate': 0.01
'Critique loop len': 80

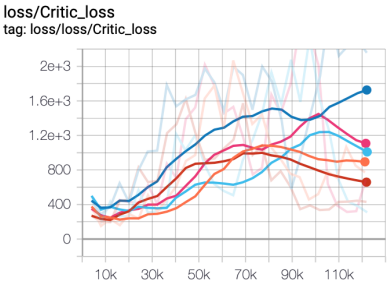
}

</> Batch algorithm, split network with undiscounted Monte-Carlo target

Observation: The actor loss has more variance than the actor loss of the discounted version.







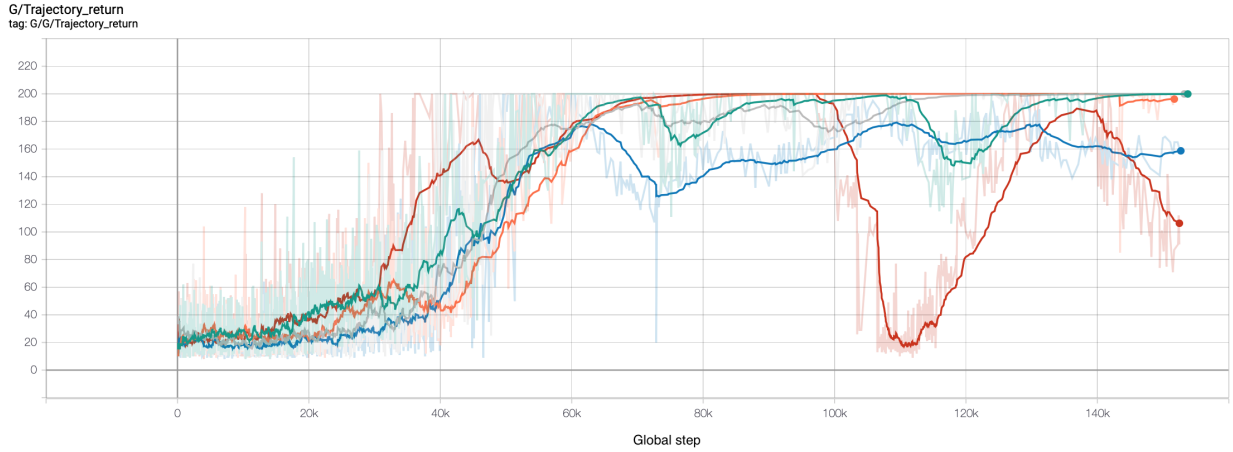
ExperimentSpec{

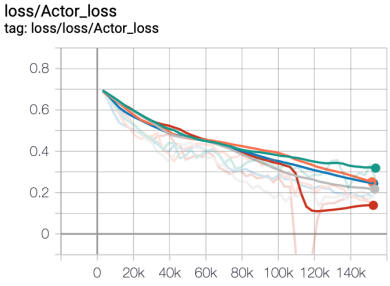
'Rerun tag': BMCSPL-NoD-A
'Batch size in ts': 4000
'Max epoch': 30
'Discout factor': 0.99
'Learning rate': 0.01
'Discounted reward to go': True
'Theta nn h layer topo': (32, 32)
'Theta hidden layers activation': relu
'Critic learning rate': 0.01
'Critique loop len': 80

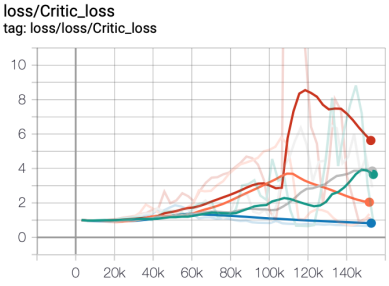
}

</> Batch algorithm, split network with Bootstrap estimate target

Observation: Less stable than Monte-Carlo target design. Relu activation seams to work better. Both actor and critic losses have a lot less variance than the Monte-Carlo target design.







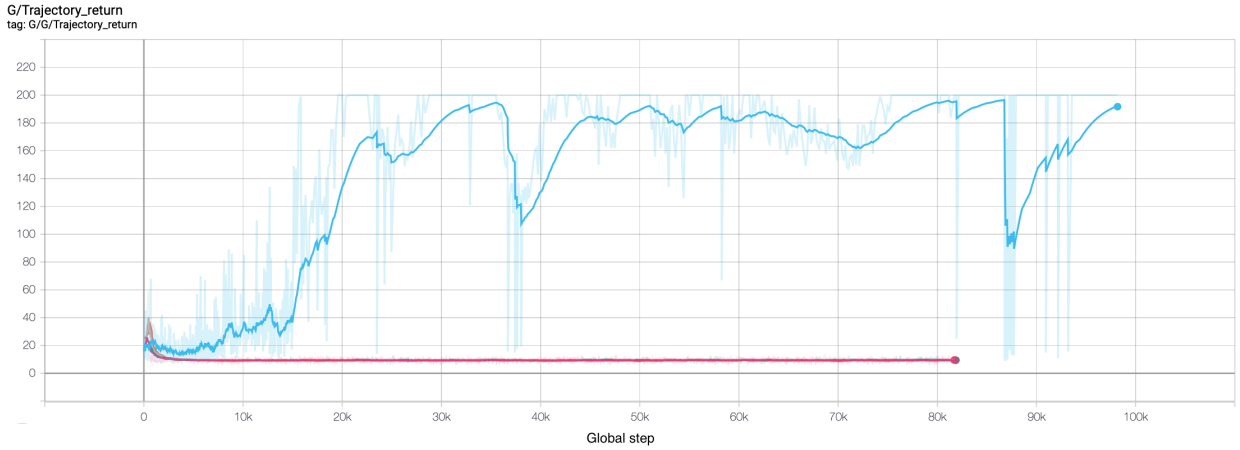
ExperimentSpec{

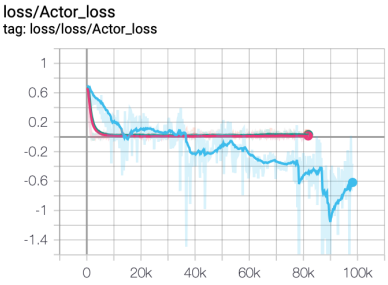
'rerun tag': BBSPL-A
'Batch size in ts': 3000
'Max epoch': 50
'Discout factor': 0.9999
'Learning rate': 0.01
'Discounted reward to go': True
'Theta nn h layer topo': (16, 32, 64)
'Theta hidden layers activation': relu
'Critic learning rate': 0.001
'Critique loop len': 120

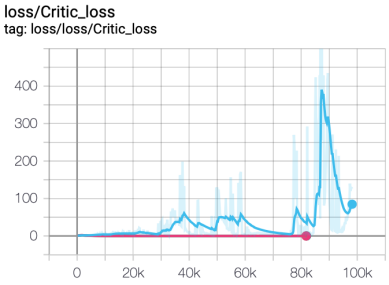
}

</> Batch algorithm, shared network, mini-batch with Bootstrap estimate target

Observation: Fail to learn 4 time out of 5. Does not learn on large batch! Work only on tiny batch (more or less 1 trajectory). Require small hidden layer topology and a small learning rate. Extremely sensible to hyperparameter tuning. Can possibly not learn at all on different run despite using the same hyperparameter. It's probably because of a combinaison of unlucky graph initialisation or unlucky initial state with a very narrow range of working hyperparameter.







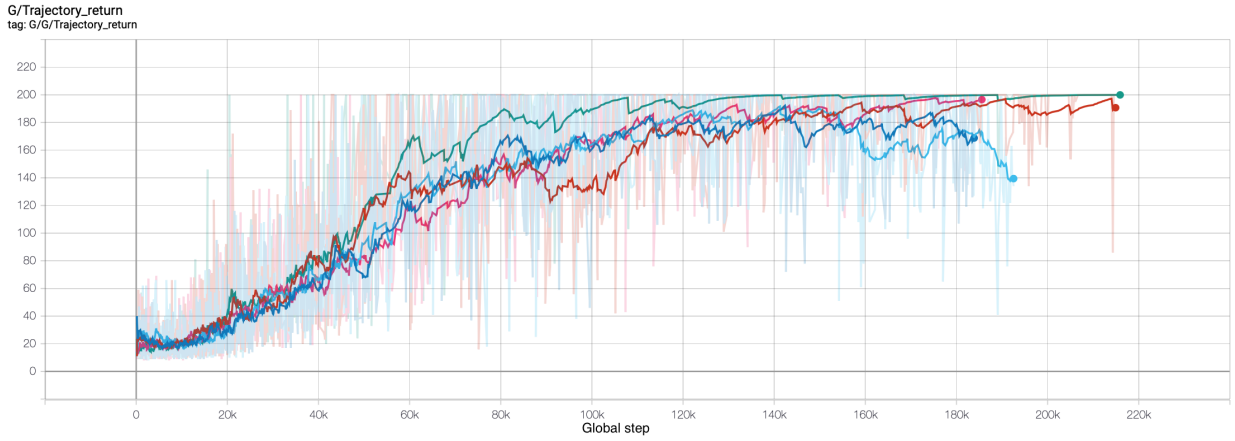
ExperimentSpec{

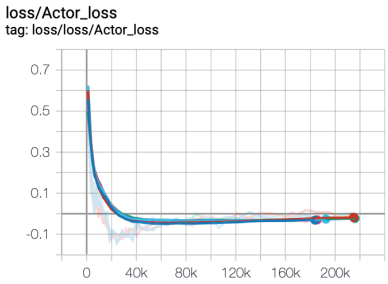
'Rerun tag': BSHA-A
'Rerun idx': 4
'Is test run': False
'Batch size in ts': 200
'Max epoch': 400
'Discout factor': 0.999
'Learning rate': 0.001
'Discounted reward to go': True
'Theta nn h layer topo': (60, 60)
'Theta hidden layers activation': leaky relu
'Critic learning rate': 0.0001
'Critique loop len': 100

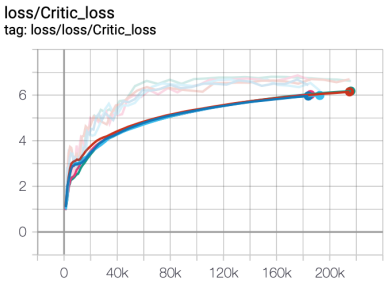
}

</> Online algorithm, split network hl(32, 32) with discounted target

Observation: Difficulte to stabilise. Those hyperparameter were very sensible: learning rate, critic learning rate, discount factor, critique loop lenght and batch size in timestep.







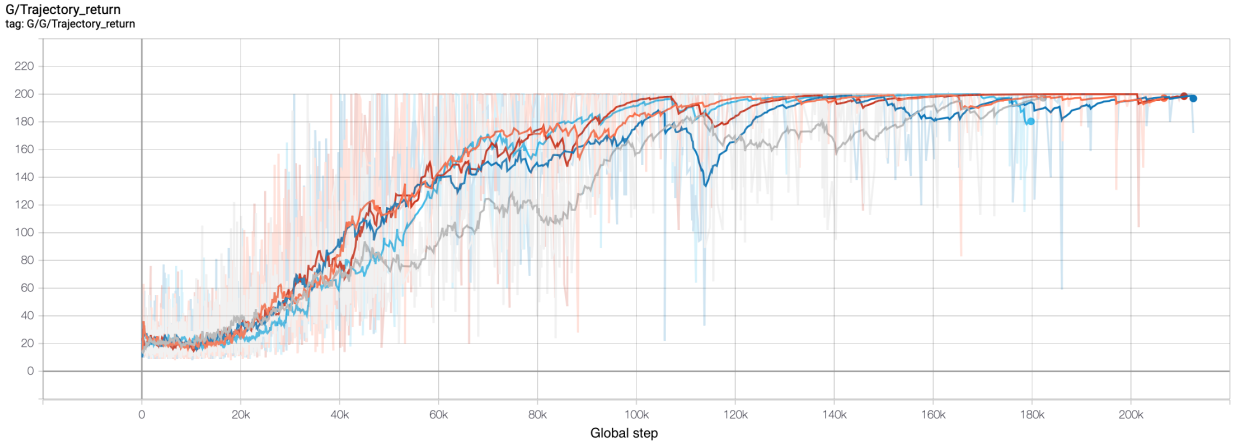
ExperimentSpec{

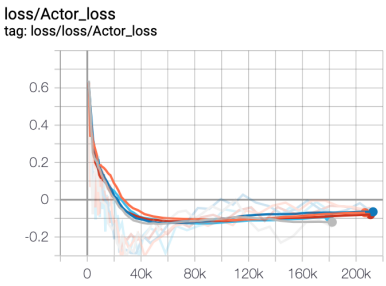
'Rerun tag': OSPL-A
'Batch size in ts': 8
'Max epoch': 45
'Discout factor': 0.999
'Learning rate': 0.0001
'Discounted reward to go': True
'Theta nn h layer topo': (32, 32)
'Theta hidden layers activation': relu
'Stage size in trj': 50
'Critic learning rate': 0.0005
'Critique loop len': 1

}

</> Online algorithm, split network hl(16, 32, 256) with discounted target

Observation: Difficulte to stabilise. Those hyperparameter were very sensible: learning rate, critic learning rate, discount factor, critique loop lenght and batch size in timestep.





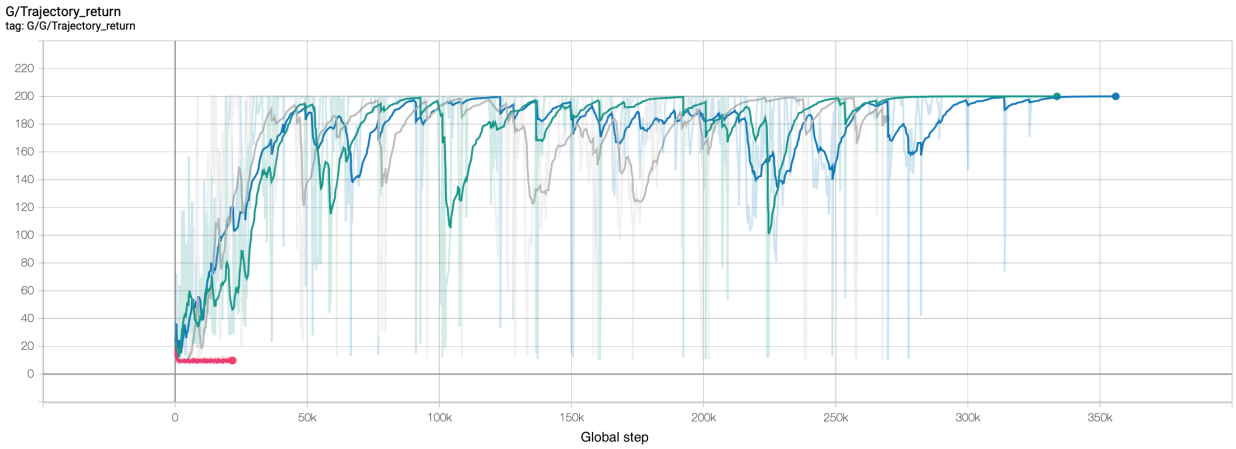
ExperimentSpec{

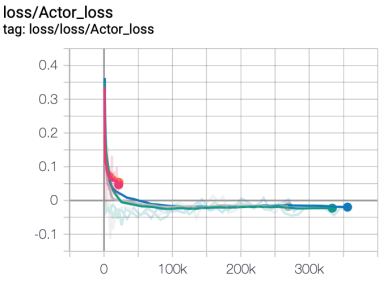
'Rerun tag': OSPL3L-A
'Batch size in ts': 20
'Max epoch': 45
'Discout factor': 0.999
'Learning rate': 5e-05
'Discounted reward to go': True
'Theta nn h layer topo': (16, 32, 256)
'Theta hidden layers activation': relu
'Stage size in trj': 50
'Critic learning rate': 0.0005
'Critique loop len': 1

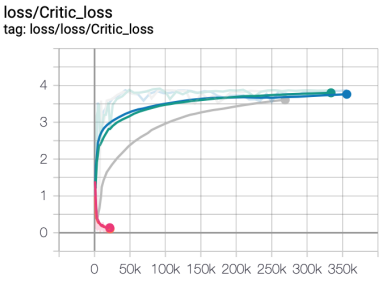
}

</> Online algorithm, shared network hl(32, 64, 256) with discounted target

Observation: Larger hidden layer size work better with shared network. Fail to learn 2 time out of 5 runs





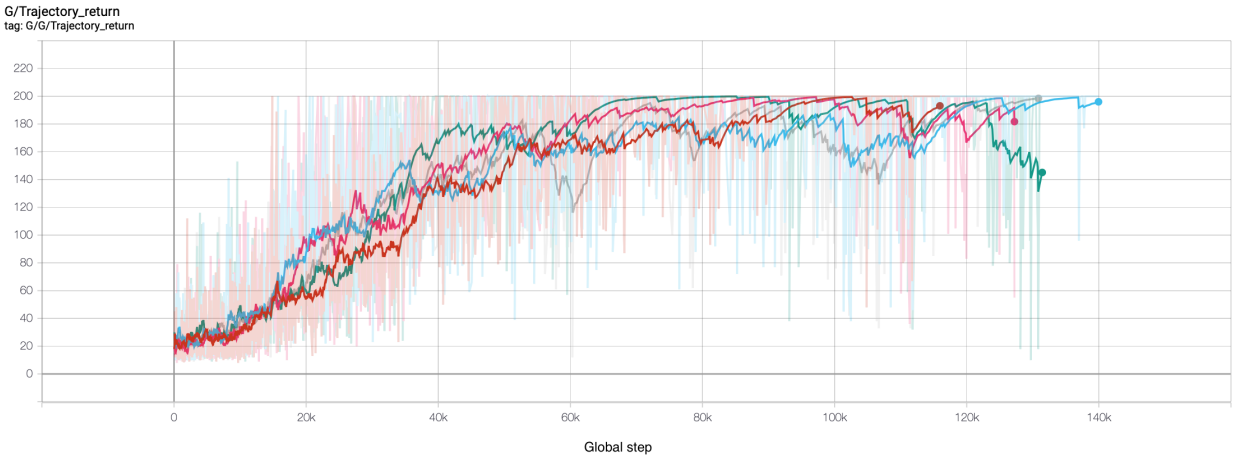


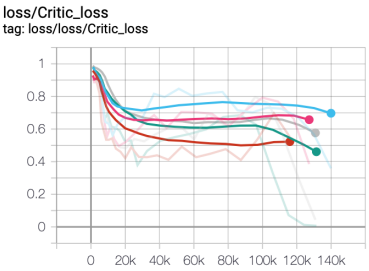
ExperimentSpec{

'Rerun tag': OSHA-A
'Batch size in ts': 10
'Max epoch': 45
'Discout factor': 0.95
'Learning rate': 0.0003
'Discounted reward to go': True
'Theta nn h layer topo': (32, 64, 256)
'Theta hidden layers activation': tanh
'Stage size in trj': 50
'Critic learning rate': 0.0003
'Critique loop len': 2

}

</> Online algorithm, split network (shared lower layer) hl(16, 32, 32) with discounted target





ExperimentSpec{

'Rerun tag': OSTWO-H
'Batch size in ts': 5
'Max epoch': 25
'Discout factor': 0.999
'Learning rate': 0.0001
'Discounted reward to go': True
'Theta nn h layer topo': (16, 32, 32)
'Theta hidden layers activation': relu
'Stage size in trj': 50
'Critic learning rate': 0.0005
'Critique loop len': 1

}

Closing thoughts

Studying *Actor-Critic* method was a very fulfilling experience. DRL is a highly active field and in that regard, it will be very interesting to see what happen with *Actor-Critic* Monte-Carlo variant in the futur. I still have a lot to explore and I'm looking foward to the next project.

</> Futur projects:

- Implement asynchronous *Actor-Critic* method [1] (A3C);
- Investigate setting where Monte-Carlo methode could perform better than boostrap method;
- Work with more chalenging environment;

BIBLIOGRAPHY

1. Mnih, V. *et al.* Asynchronous Methods for Deep Reinforcement Learning. **48.** arXiv: 1602.01783. http://arxiv.org/abs/1602.01783 (2016).
2. 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).
3. Amiranashvili, A., Dosovitskiy, A., Koltun, V. & Brox, T. TD or not TD: Analyzing the Role of Temporal Differencing in Deep Reinforcement Learning, 1–14. arXiv: 1806.01175. http://arxiv.org/abs/1806.01175 (2018).
4. Barto, A. G., Sutton, R. S. & Anderson, C. W. Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems. *IEEE Transactions on Systems, Man and Cybernetics* **SMC-13,** 834–846. ISSN: 21682909 (1983).
5. Moore, A. W. Efficient Memory-based Learning for Robot Control - Dissertation. *Learning.* ISSN: 1932-7420 (1990).
6. DeepMind. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. *DeepMind.* https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/ (2019).
7. Schulman, J., Moritz, P., Levine, S., Jordan, M. & Abbeel, P. High-Dimensional Continuous Control Using Generalized Advantage Estimation, 1–14. arXiv: 1506.02438. http://arxiv.org/abs/1506.02438 (2015).