



An introduction to RL and its applications at CERN



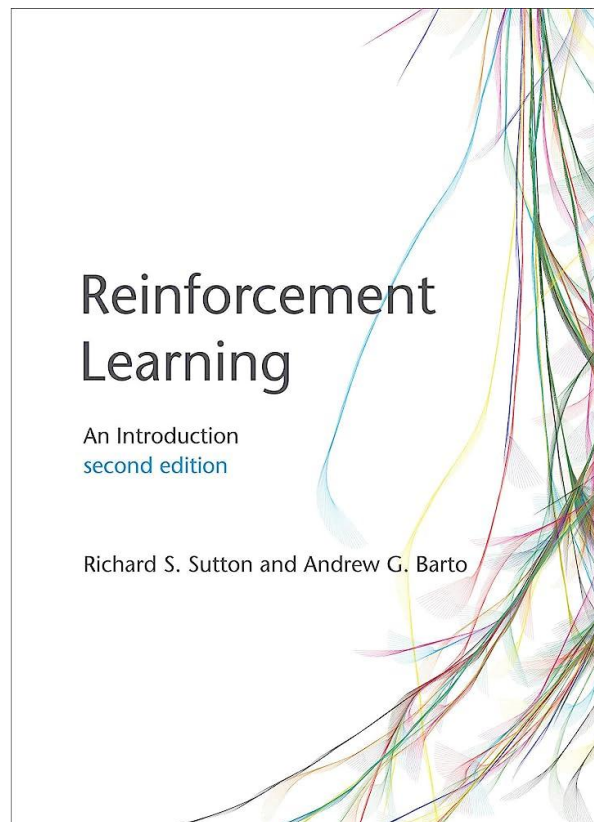
Matteo Bunino (matteo.bunino@cern.ch) - Fellow @ CERN openlab

Acknowledgements

RL theory: Sutton and Barto book "[Reinforcement Learning: an introduction](#)", Prof. David Silver [lectures](#), Prof. Marios Kountouris (EURECOM) notes, [Felix Wagner](#).

RL use cases at CERN: M. Schenk, J. Wulff, N. Bruchon, B. Goddard, S. Hirlander, V. Kain, N. Madysa, G. Valentino, F. Velotti, CERN Openlab, and the ML Community Forum.

If you find some of your materials without the proper credits, let me know and I will update the slides accordingly. Send me an email to matteo.bunino@cern.ch



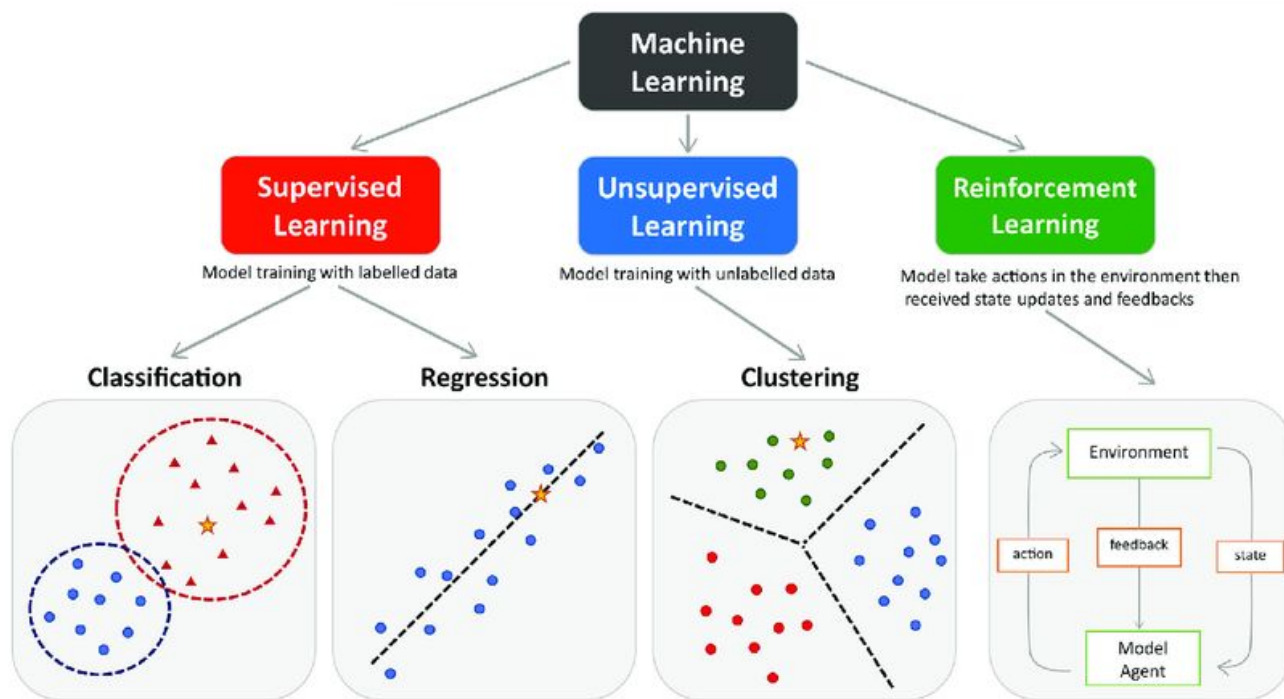


Image [credits](#)

Use cases motivating reinforcement learning (RL)

Examples from Sutton and Barto book:

- A master **chess** player makes a move. The choice is informed both by planning – anticipating possible replies and counterreplies – and by immediate, intuitive judgments of the desirability of particular positions and moves.
- An **adaptive controller** adjusts parameters of a petroleum refinery's operation in real time. The controller optimizes the yield/cost/quality trade-off on the basis of specified marginal costs without sticking strictly to the set points originally suggested by engineers.
- A **gazelle** calf struggles to its feet minutes after being born. Half an hour later it is running at 20 miles per hour.
- A **mobile robot** decides whether it should enter a new room in search of more trash to collect or start trying to find its way back to its battery recharging station. It makes its decision based on the current charge level of its battery and how quickly and easily it has been able to find the recharger in the past.

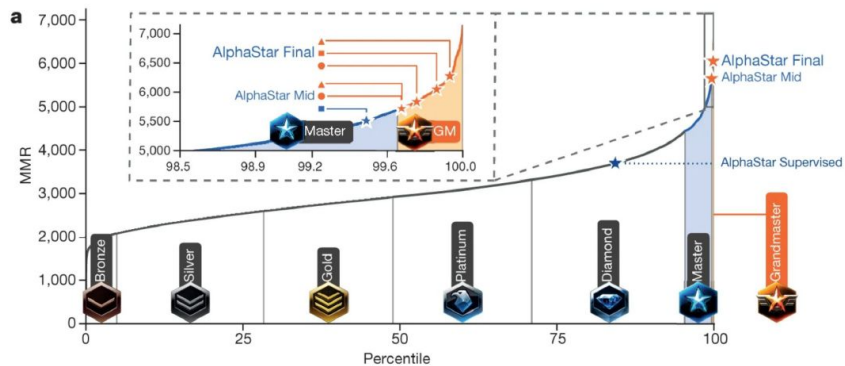
RL in games



“AlphaGo” winning against the Go world champion (2016).

Look for “AlphaGo’s move 37” on the web...

<https://www.nature.com/articles/nature16961>



“AlphaStar” wins Starcraft against 99.85% of human players (2019).

<https://www.nature.com/articles/s41586-019-1724-z>

Reinforcement learning concepts



In a nutshell: learn a **policy** which maximizes the **total expected reward** over time.

Multistage decision-making process: the learner is not told which actions to take - it discovers which actions yield the most reward by trying them.

Not *supervised learning*: the agent **learns from its own experience**, not from representative examples.

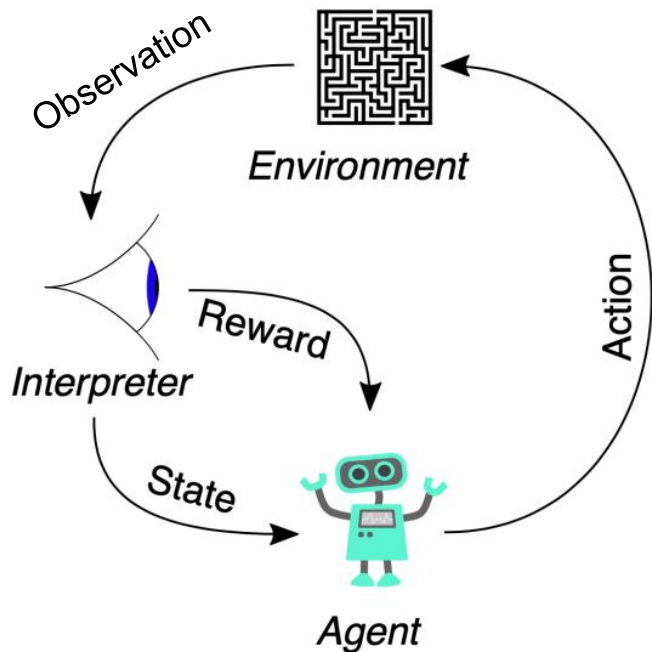
Not *unsupervised learning*: maximize a reward signal instead of trying to find hidden structure in data.

Reinforcement learning (RL) peculiarities:

- Learn by **trial-and-error** search
- **Delayed reward**

...similar to human learning

Reinforcement learning concepts



Goal: learn the **optimal policy** which maximizes the **total expected reward** over time.

Environment: can be accessible only partially. Some dynamics may remain obscure, and we get only what we can *observe*.

Interpreter: that's defined by us. Sort of pre-processing. It builds the state based on the history of previous observations and interactions. It also implements the reward function.

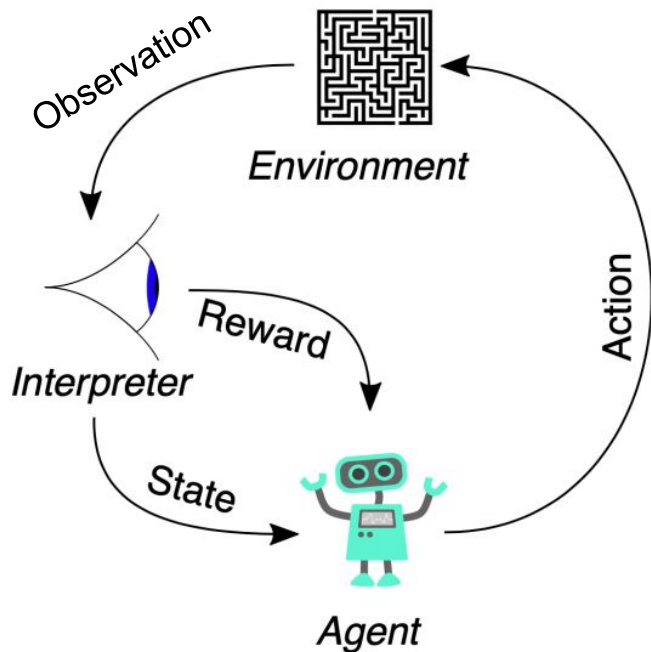
State: describes the environment. It belongs to the states space $s_t \in \mathcal{S}$

Reward (scalar number) is the only feedback the agent receives, which describes the "goodness" of the trajectory so far. $r_t \in \mathcal{R}$

Action: sampled by the agent from the actions space $a_t \in \mathcal{A}$

Interaction defines trajectories: $S_0, A_0, R_1, S_1, A_1 \dots$

Reinforcement learning concepts

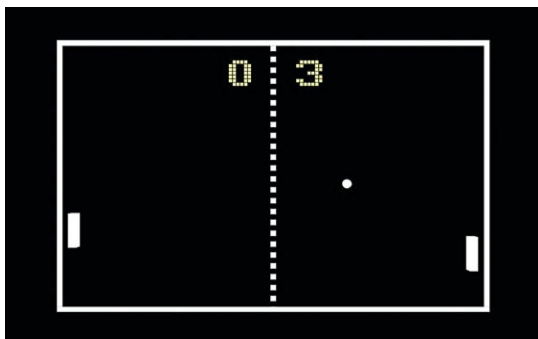


Goal: learn the **optimal policy** which maximizes the **total expected reward** over time.

Policy is a mapping from perceived states of the environment to actions to be taken: $\pi : \mathcal{S} \rightarrow \mathcal{A}$

State value function: value of a state = total amount of reward an agent can expect to accumulate over the future, starting from that state (specifies what is good in the long run). $v : \mathcal{S} \rightarrow \mathbb{R}$
It estimates *how good* is for an agent to be in a given state.

Reinforcement learning concepts - example



Atari's "Pong"

Goal: learn the **optimal policy** which maximizes the **total expected reward** over time.

State: tuple (ball_x, ball_y, cursor_h, opponent_cursor_h), for each t .

Action: up or down of 1cm. $\mathcal{A} = \{\text{'up'}, \text{'down'}\}$

Reward: e.g., scored points, or +1 if agent scored, -1 if opponent scored, 0 otherwise. The design of the reward function is often tricky and shall be tuned.

Optimal policy: find the best mapping between the state and the action to take. E.g., go up when the ball is coming top right.

State value function (informal): how many points am I expecting to score given that now I am in state S_t ?

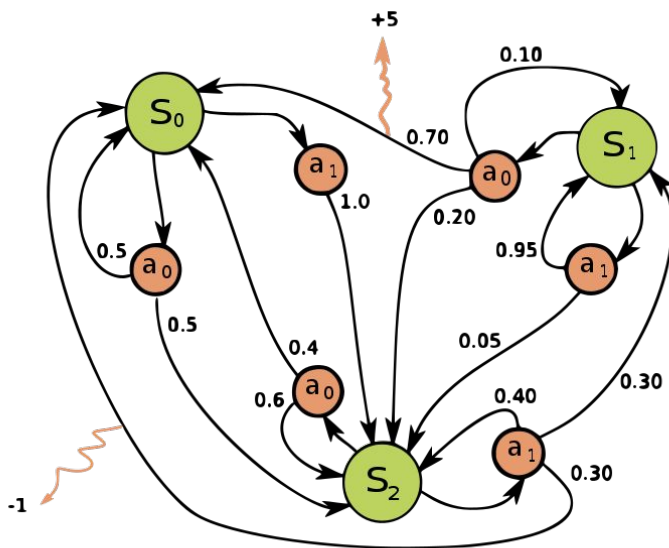
Reinforcement learning challenges

- Often a long sequence of actions before we discover consequences of the actions.
 - e.g., win or lose game only after moves are complete.
- Never see the result of actions not taken.
- Never told what the best action was.
- The outcome of our actions may be uncertain.
- We may not be able to perfectly sense the state of the world.
- The **reward** may be **stochastic or delayed**.
- We may have no clue (model) about how the world responds to our actions.
- We may have no clue (model) of how rewards are being paid off.
- World may change while you try to learn it: **dynamic environment**.
- How much time for **exploration** (of an uncharted territory) before **exploitation** of what we have learned?

Foster innovative solutions, e.g., "AlphaGo's move 37"



Markov Decision Process



Maths prerequisites

Given \mathbf{X} , \mathbf{Y} and \mathbf{Z} (discrete) random variables:

$$\mathbb{E}_p[\mathbf{X}] \doteq \sum_{x_i \sim p(x)} x_i p(x_i) \quad (\text{Expectation, for some discrete probability distribution } p(x))$$

$$\mathbb{E}[\alpha\mathbf{X} + \beta\mathbf{Y}] \doteq \alpha\mathbb{E}[\mathbf{X}] + \beta\mathbb{E}[\mathbf{Y}] \quad (\text{Linearity})$$

$$\text{Var}(\mathbf{X} + \mathbf{Y}) = \text{Var}(\mathbf{X}) + \text{Var}(\mathbf{Y}) + 2\text{Cov}(\mathbf{X}, \mathbf{Y})$$

$$p(x|y) \doteq \frac{p(x, y)}{p(y)} \quad (\text{Conditional probability})$$

$$p(x, y) = p(x|y) p(y) \quad (\text{Joint probability})$$

$$p(x, y|z) = p(x|y, z) p(y|z) \quad (\text{Add conditional term to previous one})$$

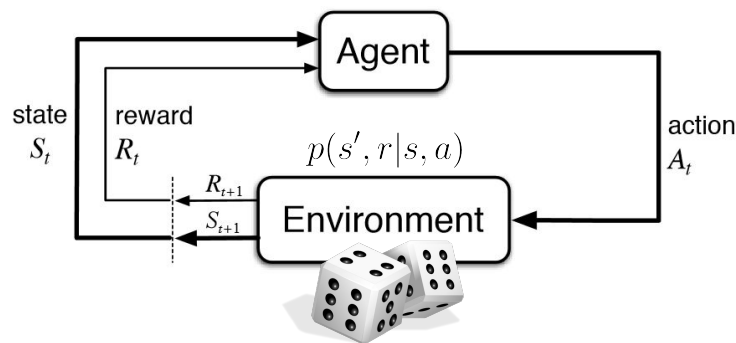
$$p(x) \doteq \sum_i p(x, y_i) = \sum_i p(x|y_i) p(y_i) \quad (\text{Marginal probability})$$

Example:

\mathbf{X}	0	1	2	3
$\mathbf{p}(\mathbf{x})$	0.1	0.3	0.2	0.4

$$\mathbb{E}_p[\mathbf{X}] = 0 \cdot 0.1 + 1 \cdot 0.3 + 2 \cdot 0.2 + 3 \cdot 0.4 = 1.9$$

Markov Decision Process (MDP)



$$p(s', r | s, a) = \Pr\{S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a\}$$

Random variables

The environment can be modeled as an MDP when the following are known:

- Action space: \mathcal{S}
- State space: \mathcal{A}
- Reward space: \mathcal{R}
- MDP dynamics: $p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \times \mathcal{R} \rightarrow [0, 1]$

The MDP/env dynamics fully describes the MDP under analysis, allowing for analytical solutions.

MDP is *finite* if $\mathcal{S}, \mathcal{A}, \mathcal{R}$ are *finite sets*.

Under this formulation, we say that the agent interacts with the environment by performing some **action** A_t , transitioning to a **new state** S_{t+1} and receiving a scalar feedback called **reward** R_{t+1} .

This results in a **trajectory**: $S_0, A_0, R_1, S_1, \dots, R_T, S_T$ (where S_T is the *terminal state*).

Markov property:

$$\begin{aligned} p(s', r | s, a) &= \Pr\{S_{t+1}, R_{t+1} | S_t, A_t, S_{t-1}, A_{t-1}, \dots, S_0, A_0\} \\ &= \Pr\{S_{t+1}, R_{t+1} | S_t, A_t\} \end{aligned}$$

Environment dynamics

When the environment dynamics function $p(s', r|s, a)$ is known, we can compute everything else one may want to know about the environment:

- State-transition probabilities $p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$

$$p(s'|s, a) \doteq \Pr\{S_t = s' \mid S_{t-1} = s, A_{t-1} = a\} = \sum_{r \in \mathcal{R}} p(s', r|s, a)$$

- Expected rewards for state-action pairs $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$

$$r(s, a) \doteq \mathbb{E}[R_t \mid S_{t-1} = s, A_{t-1} = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r|s, a)$$

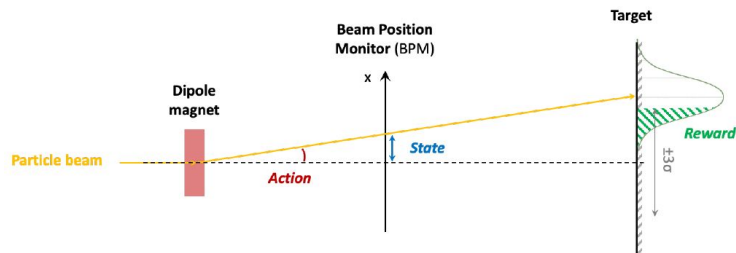
- Expected rewards for state-action-next-state triples $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$

$$r(s, a, s') \doteq \mathbb{E}[R_t \mid S_{t-1} = s, A_{t-1} = a, S_t = s'] = \sum_{r \in \mathcal{R}} r \frac{p(s', r|s, a)}{p(s'|s, a)}$$

Episodic and continuing tasks

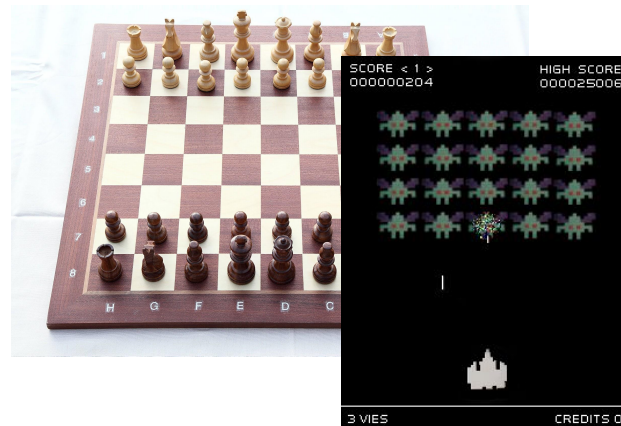
Continuing tasks:

- The agent-environment interaction could **go on forever**.
- There is no terminal state.
- Trajectories can reach *infinite* length.
- Examples: thermostat keeping the room temperature stable, steering the beam in the LHC.



Episodic tasks:

- The agent-environment interaction is **limited in time**.
- At some point a **terminal state** S_T is reached.
- Examples: board games, video games, robotic arm manipulating objects.



Return

The goal of the agent is to find the **optimal policy**: “what is the best action I should take in state S_t ?”

To assess the “goodness” of a state (an action), the agent tries to estimate the cumulative future reward of a *trajectory* starting from that state (and taking that action). More formally, we call this property **return**, and we define it as the cumulative future reward:

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-t} R_T \\ &= \sum_{k=t+1}^T \gamma^{k-t} R_k \\ &= R_{t+1} + \gamma G_{t+1} \quad (\text{recursive definition}) \end{aligned}$$

$\gamma \in [0, 1]$ is the **discount factor**. For continuing tasks $T = \infty$, thus the discount factor has to be $\gamma < 1$ for the sum to converge.

Policy

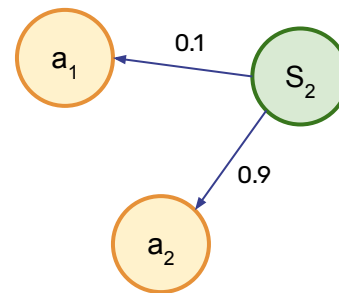
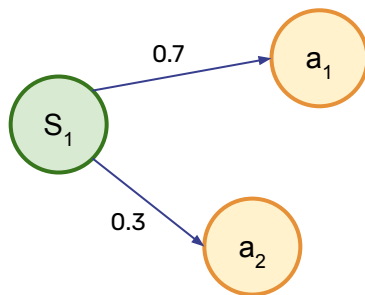
A policy describes the *behavior* of the RL agent, mapping from state to probabilities of selecting each possible action.

Policy $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$

$$\pi(a|s) = \Pr\{A_t = a | S_t = s\}$$

Example:

	$p(a_1)$	$p(a_2)$
State s_1	0.7	0.3
State s_2	0.1	0.9



The first step to find the **optimal policy** is to assess how good is the current one...

Value Functions

Allow to assess the “goodness” of some policy π .

The state value function is the *expected return* when starting in state s and following π thereafter:

$$\begin{aligned}v_{\pi}(s) &= \mathbb{E}_{\pi} [G_t \mid S_t = s], \forall s \in \mathcal{S} \\ &= \mathbb{E}_{\pi} [\underbrace{R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots}_{\text{Expected return}} \mid S_t = s]\end{aligned}$$

How to generalize this?

Value Functions

Allow to assess the “goodness” of some policy π .

The state value function is the *expected return* when starting in state S and following π thereafter:

$$\begin{aligned}v_{\pi}(s) &= \mathbb{E}_{\pi} [G_t \mid S_t = s], \forall s \in \mathcal{S} \\ &= \mathbb{E}_{\pi} [R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \mid S_t = s] \\ &= \mathbb{E}_{\pi} [R_{t+1} + \gamma G_{t+1} \mid S_t = s] \quad (\text{Recursive formulation of } G_t)\end{aligned}$$

Still not so useful... What do we do with G_{t+1} ?

Value Functions

Allow to assess the “goodness” of some policy π .

The state value function is the *expected return* when starting in state s and following π thereafter:

$$\begin{aligned}v_{\pi}(s) &= \mathbb{E}_{\pi} [G_t \mid S_t = s], \forall s \in \mathcal{S} \\&= \mathbb{E}_{\pi} [R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \mid S_t = s] \\&= \mathbb{E}_{\pi} [R_{t+1} + \gamma G_{t+1} \mid S_t = s] \quad (\text{Recursive formulation of } G_t) \\&= \sum_a \pi(a|s) \sum_{s', r} p(s', r \mid s, a) \left[r + \gamma \underbrace{\mathbb{E}_{\pi} [G_{t+1} \mid S_{t+1} = s']}_{v_{\pi}(s')} \right] \quad (\text{Expectation properties})\end{aligned}$$

Interesting recursive relationship to “remove” the return... thus the explicit dependency on the future.

Value Functions

Allow to assess the “goodness” of some policy π .

The state value function is the *expected return* when starting in state s and following π thereafter:

$$\begin{aligned}v_{\pi}(s) &= \mathbb{E}_{\pi} [G_t \mid S_t = s], \forall s \in \mathcal{S} \\&= \mathbb{E}_{\pi} [R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \mid S_t = s] \\&= \mathbb{E}_{\pi} [R_{t+1} + \gamma G_{t+1} \mid S_t = s] \quad (\text{Recursive formulation of } G_t) \\&= \sum_a \pi(a|s) \sum_{s',r} p(s',r \mid s,a) \left[r + \gamma \underbrace{\mathbb{E}_{\pi} [G_{t+1} \mid S_{t+1} = s']}_{v_{\pi}(s')} \right] \quad (\text{Expectation properties}) \\&= \sum_a \pi(a|s) \sum_{s',r} p(s',r \mid s,a) [r + \gamma v_{\pi}(s')], \forall s \in \mathcal{S} \quad (\text{Bellman equation})\end{aligned}$$

Value Functions

Allow to assess the “goodness” of some policy π .

The state value function is the *expected return* when starting in state s and following π thereafter:

$$v_{\pi}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r | s,a) [r + \gamma v_{\pi}(s')], \quad \forall s \in \mathcal{S} \quad (\text{Bellman equation})$$

When \mathcal{S} is finite, we can solve it directly as a linear system in $|\mathcal{S}|$ unknowns:

$$\mathbf{v} = \mathbf{R}^{\pi} + \gamma \mathbf{P}_{s,s'}^{\pi} \mathbf{v} \quad \longrightarrow \quad \mathbf{v} = (\mathbf{I} - \gamma \mathbf{P}_{s,s'}^{\pi})^{-1} \mathbf{R}$$

Where R is the immediate expected reward and P is the state transition matrix.

However, this is expensive also for small MDPs! $\mathcal{O}(|\mathcal{S}|^3)$

Value Functions

Allow to assess the “goodness” of some policy π .

The state value function is the *expected return* when starting in state S and following π thereafter:

$$v_{\pi}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r | s,a) [r + \gamma v_{\pi}(s')], \quad \forall s \in \mathcal{S} \quad (\text{Bellman equation})$$

The state-action value function is the *expected return* when starting in state S , taking action a , and following π thereafter:

$$\begin{aligned} q_{\pi}(s,a) &= \mathbb{E}_{\pi} [G_t | S_t = s, A_t = a], \quad \forall s \in \mathcal{S}, a \in \mathcal{A} \\ &= \sum_{s',r} p(s',r | s,a) [r + \gamma v_{\pi}(s')] \end{aligned}$$

Very similar to the state value function...
but not recursive

Value Functions

Allow to assess the “goodness” of some policy π .

The state value function is the *expected return* when starting in state S and following π thereafter:

$$v_{\pi}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r | s,a) [r + \gamma v_{\pi}(s')], \quad \forall s \in \mathcal{S} \quad (\text{Bellman equation})$$

The state-action value function is the *expected return* when starting in state S , taking action a , and following π thereafter:

$$\begin{aligned} q_{\pi}(s,a) &= \mathbb{E}_{\pi} [G_t | S_t = s, A_t = a], \quad \forall s \in \mathcal{S}, a \in \mathcal{A} \\ &= \sum_{s',r} p(s',r | s,a) [r + \gamma v_{\pi}(s')] \\ &= \sum_{s',r} p(s',r | s,a) \left[r + \gamma \sum_{a'} \pi(a'|s') q_{\pi}(s',a') \right], \quad \forall s \in \mathcal{S}, a \in \mathcal{A} \quad (\text{Bellman equation}) \end{aligned}$$

Bellman optimality equations

So far, we “evaluated” some policy π by computing its associated value functions $v_\pi(s)$ and $q_\pi(s, a)$.
How can we compute directly the optimal policy π_* ?

a.k.a. value function

The optimal policy is the one which maximizes the **expected cumulative future reward** in each state s .

$$v_*(s) \doteq \max_{\pi} v_\pi(s), \text{ for all } s \in \mathcal{S}.$$

$$q_*(s, a) \doteq \max_{\pi} q_\pi(s, a), \text{ for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A}(s).$$

The goal of the RL agent is to find the optimal policy which maximizes the value functions.



Bellman optimality equations

State value functions:

$$v_{\pi}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r | s,a) [r + \gamma v_{\pi}(s')], \forall s \in \mathcal{S} \quad (\text{Bellman equation})$$

$$v_{*}(s) = \max_{a \in \mathcal{A}(s)} \sum_{s',r} p(s',r | s,a) [r + \gamma v_{*}(s')], \forall s \in \mathcal{S} \quad (\text{Bellman optimality equation})$$

In both cases, we replace the expectation with a max over the action space

State-action value functions:

$$q_{\pi}(s,a) = \sum_{s',r} p(s',r | s,a) \left[r + \gamma \sum_{a'} \pi(a'|s') q_{\pi}(s',a') \right], \forall s \in \mathcal{S}, a \in \mathcal{A} \quad (\text{Bellman equation})$$

$$q_{*}(s,a) = \sum_{s',r} p(s',r | s,a) \left[r + \gamma \max_{a' \in \mathcal{A}(s')} q_{*}(s',a') \right], \forall s \in \mathcal{S}, a \in \mathcal{A} \quad (\text{Bellman optimality equation})$$

Optimal policies

The optimal policy is the policy that assigns non-zero probabilities only to the actions that maximize the the value function in some state, for all states.

Optimal state-action value function

$$q_*(a, s)$$

	a1	a2	a3
s1	20	30	30.1
s2	15	15	3

Optimal policy #1

	Pr(a1)	Pr(a2)	Pr(a3)
s1	0	0	1.0
s2	0.5	0.5	0

Optimal policy #2

	Pr(a1)	Pr(a2)	Pr(a3)
s1	0	0	1.0
s2	0.99	0.01	0

Greedy policies

Optimal policy #3

	Pr(a1)	Pr(a2)	Pr(a3)
s1	0	0	1.0
s2	1.0	0	0

Optimal policy #4

	Pr(a1)	Pr(a2)	Pr(a3)
s1	0	0	1.0
s2	0	1.0	0

MDP summary

In some situations, we can assume that the environment can be modeled as a Markov decision process.

The environment dynamics $p(s', r | s, a) = \Pr\{S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a\}$ are fully known as a common function. In the discrete case, you can imagine p as a 4-dimensional lookup table for probabilities.

This allows us to easily compute Bellman equations and Bellman optimality equations:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r | s, a) [r + \gamma v_\pi(s')], \forall s \in \mathcal{S} \quad (\text{Bellman equation})$$

$$v_*(s) = \max_{a \in \mathcal{A}(s)} \sum_{s',r} p(s', r | s, a) [r + \gamma v_*(s')], \forall s \in \mathcal{S} \quad (\text{Bellman optimality equation})$$

$$q_\pi(s, a) = \sum_{s',r} p(s', r | s, a) \left[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a') \right], \forall s \in \mathcal{S}, a \in \mathcal{A} \quad (\text{Bellman equation})$$

$$q_*(s, a) = \sum_{s',r} p(s', r | s, a) \left[r + \gamma \max_{a' \in \mathcal{A}(s')} q_*(s', a') \right], \forall s \in \mathcal{S}, a \in \mathcal{A} \quad (\text{Bellman optimality equation})$$

These recursive equations can be solved

- As system of (non)linear equations
- Iteratively by means of **dynamic programming** (e.g., policy iteration, value iteration algorithms). Not covered in this lecture.

From the Bellman optimality equations, it is easy to obtain the **optimal policy** which maximizes future rewards.

SOCIETY IF ENVIRONMENT DYNAMICS WERE ALWAYS KNOWN

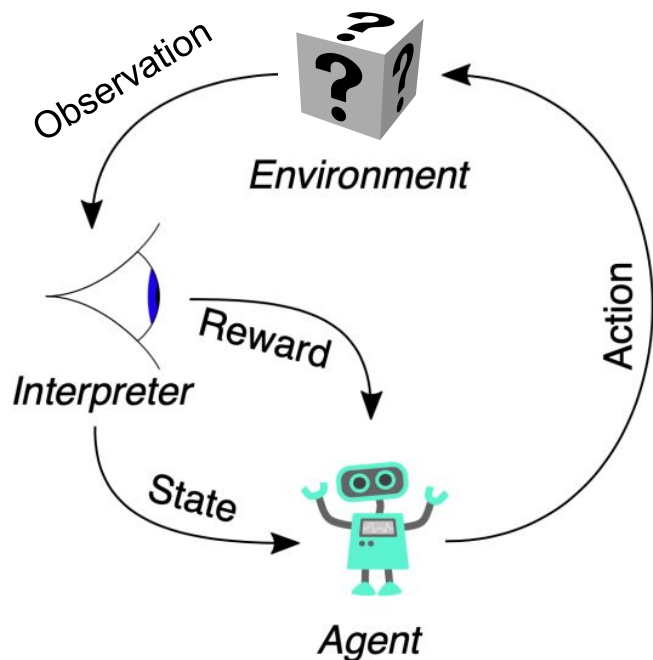


imgflip.com

Sample-based methods



Sample-based methods



When the environment dynamics are not known, we can *simulate them* by interacting directly with the environment.

Again we can have **episodic** and **continuing** tasks.

In this case, episodes are characterized by trajectories of finite length, terminated by some *terminal state* S_T :

$$S_0, A_0, R_1, S_1, \dots, R_T, S_T$$

- The longer we interact with the environment,
- the more data we collect,
- the better our *estimate* of the underlying dynamics will be precise...

...at the cost of taking very long time.

Sample efficiency (informally): how many interactions with the environment do we need before being able to *exploit* the gained knowledge for our goals?

Monte Carlo methods

Base on *experience*: sample sequences of states, actions and rewards from *actual* or *simulated* interactions with the environment: $S_0, A_0, R_1, S_1, \dots, R_T, S_T$

“Monte Carlo” replace expectation on the return with **average**:

$$q_{\pi}(s, a) = \mathbb{E}_{\pi}[G_t | S_t = s, A_t = a] \approx Q_{\pi}(S_t, A_t) = \frac{1}{N} \sum_{i=1}^N G_{t,i}(S_t, A_t)$$

$G_t \doteq R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-1} R_T$ is known only **at the end of an episode**, thus we can only apply Monte Carlo methods to **episodic tasks**, which terminate at some point (reach some terminal state S_T).

Therefore, *value functions* and *policies* are updated only at the end of each episode. Monte carlo is incremental in an **episode-by-episode** sense, but not in a step-by-step sense (on-line):

$$\underbrace{S_0, A_0, R_1, S_1, \dots, S_{T_1}}_{\text{Step}} ; S_0, A_0, R_1, S_1, \dots, S_{T_2}$$

Episode

Monte Carlo prediction

Prediction = estimating the value function

First-visit MC prediction, for estimating $Q \approx q_*$

Input: a policy π to be evaluated

Initialize:

$Q(s, a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

$Returns(s, a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):

Generate an episode following π : $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \dots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

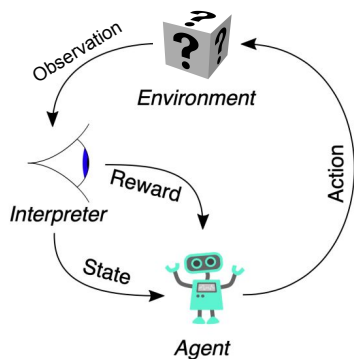
Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1, \dots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

Monte Carlo prediction - example

Interact with the env...



...according to some policy.

$$\pi(a|s) = \Pr\{A_t = a | S_t = s\}$$

	p(a ₁)	p(a ₂)
State s ₁	0.7	0.3
State s ₂	1.0	0

Initial Q table:

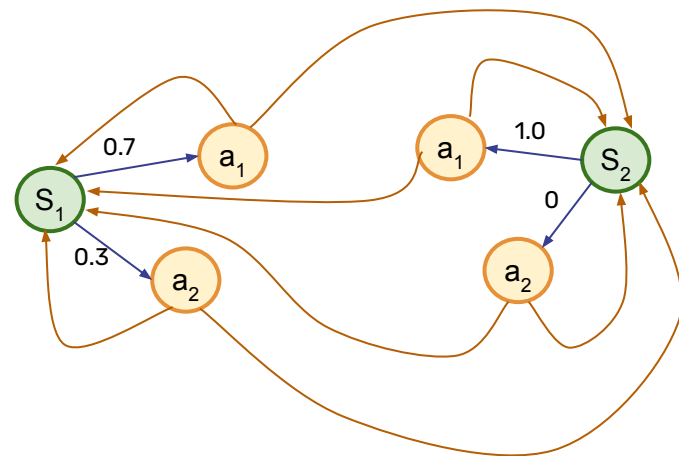
Q(s,a)	a ₁	a ₂
State s ₁	???	???
State s ₂	???	???

Upon convergence



Resulting Q table:

Q(s,a)	a ₁	a ₂
State s ₁	23.1	12.09
State s ₂	2.57	???



We never visited (s₂, a₂)!

Exploration v. exploitation

When the env dynamics are not known, we need to sample from the environment, at the risk of incurring in bias.

Exploration requires devoting some interactions budget to low-rewarding interactions, however in the long run it can result in better rewards.

Exploiting too early can lead to *suboptimal* policies, which are too shortsighted. They prefer small immediate rewards versus big delayed rewards.

Too few exploration in favour of exploitation may bias the agent, with the risk of locking him sub-optimal policies forever!

To find the best policy, the agent may have to explore a lot before, at a greater computational cost. Trade-off!



Maintaining exploration

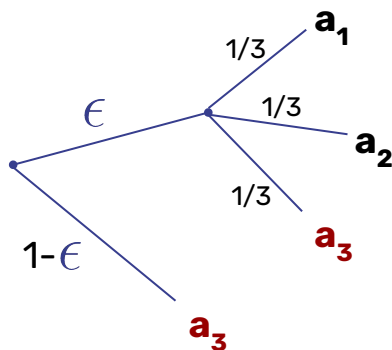
In practice, a popular way to maintain exploration is resorting to ϵ -greedy policies.

$$\pi_{\epsilon}(a|S_t) = \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}|, & \text{if } a = A^* \text{ (Greedy action)} \\ \epsilon/|\mathcal{A}|, & \text{otherwise} \end{cases}$$

ϵ is usually small (e.g., 0.1) and $\epsilon < 1$.

Example, given 3 actions a_1, a_2, a_3 where $A^* = a_3$:

Let's visualize it with the help of a **probability tree**...



$$\Pr(\mathbf{a}_1) = \epsilon/3$$

$$\Pr(\mathbf{a}_2) = \epsilon/3$$

$$\Pr(\mathbf{a}_3) = \epsilon/3 + (1 - \epsilon)$$

Monte Carlo control

Control = improving the current policy

On-policy first-visit MC control (for ε -soft policies), estimates $\pi \approx \pi_*$

Algorithm parameter: small $\varepsilon > 0$

Initialize:

$\pi \leftarrow$ an arbitrary ε -soft policy

$Q(s, a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

$Returns(s, a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):

Generate an episode following π : $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \dots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1, \dots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow$ average($Returns(S_t, A_t)$)

$A^* \leftarrow \operatorname{argmax}_a Q(S_t, a)$

(with ties broken arbitrarily)

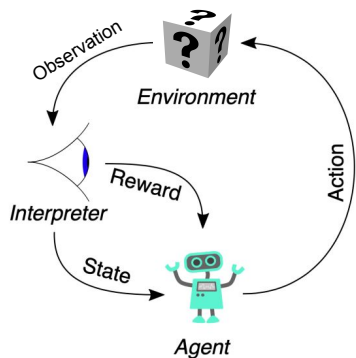
For all $a \in \mathcal{A}(S_t)$:

Novelty

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

Monte Carlo control- example

Interact with the env...



...according to some policy.

$$\pi(a|s) = \Pr\{A_t = a | S_t = s\}$$

	p(a ₁)	p(a ₂)
State s ₁	ϵ/2 + 1 - ϵ	ϵ/2
State s ₂	ϵ/2 + 1 - ϵ	ϵ/2

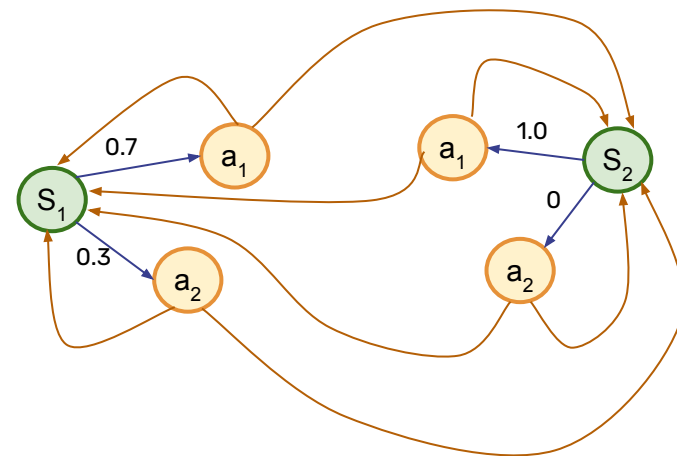
Initial Q table:

Q(s,a)	a ₁	a ₂
State s ₁	???	???
State s ₂	???	???

Estimate Q table



Loading...





One Eternity Later

MC has high variance + off-line -> slow learning

Monte Carlo control- example

Initial Q table:

Q(s,a)	a ₁	a ₂
State s ₁	???	???
State s ₂	???	???

Upon convergence



Resulting Q table:

Q(s,a)	a ₁	a ₂
State s ₁	23.1	12.09
State s ₂	2.57	42.5

Red: max action value

Initial policy:

	p(a ₁)	p(a ₂)
State s ₁	$\epsilon/2 + 1 - \epsilon$	$\epsilon/2$
State s ₂	$\epsilon/2 + 1 - \epsilon$	$\epsilon/2$

Control



Final policy:

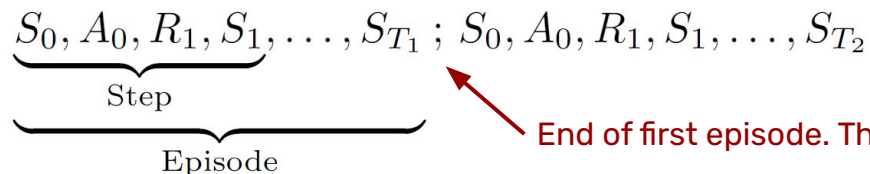
	p(a ₁)	p(a ₂)
State s ₁	$\epsilon/2 + 1 - \epsilon$	$\epsilon/2$
State s ₂	$\epsilon/2$	$\epsilon/2 + 1 - \epsilon$

Red background: preferred action

Monte Carlo (MC) - summary

The return $G_t \doteq R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-1} R_T$ is known only **at the end of an episode**, thus we can only apply Monte Carlo methods to **episodic tasks**, which terminate at some point (reach some terminal state).

Therefore, *value functions* and *policies* are updated only at the end of each episode. Monte carlo is incremental in an **episode-by-episode** sense (off-line), but not in a step-by-step sense (on-line):



Drawbacks:

- Off-line method: **policies** and **value functions** can be updated **only at the end of an episode** -> **Low sample efficiency**: need many interactions with the environment to converge.
- Not applicable to **continuing tasks**.
- it is subject to relatively **high variance**, since it estimates the expected return as the (weighted) sum of the rewards (random variables):

$$Q_\pi(S_t, A_t) = \frac{1}{N} \sum_{i=1}^N \underbrace{(R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots + \gamma^{T-1} R_T)}_i$$

TD(0) methods

“If one had to identify one idea as central and novel to reinforcement learning, it would undoubtedly be temporal-difference (TD) learning.”

– Barto–Sutton RL book.

How to improve sample efficiency of MC methods?

Idea: turn **off-line** into **on-line**!

Recall that the expression of the return can be rewritten **recursively**:

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-1} R_T \\ &= R_{t+1} + \gamma G_{t+1} \quad (\text{recursive definition}) \end{aligned}$$

Now, at each step (interaction) estimate the return by means of **bootstrapping**:

$$V(S_t) = R_{t+1} + \gamma V(S_{t+1})$$

The incremental update is possible as soon as $S_t, A_t, R_{t+1}, S_{t+1}$ are available.

Don't need to wait for the end of the episode to update value functions and policies.

TD(0) methods

Don't need to wait for the end of the episode to update value functions and policies.

The *on-line* update rule becomes, for some small learning rate $0 < \alpha < 1$:

$$\begin{aligned} V^{k+1}(S_t) &\leftarrow (1 - \alpha) V^k(S_t) + \alpha [R_{t+1} + \gamma V^k(S_{t+1})] \\ &\leftarrow V^k(S_t) + \alpha \underbrace{[R_{t+1} + \gamma V^k(S_{t+1}) - V^k(S_t)]}_{\text{TD error}} \end{aligned}$$

$$Q^{k+1}(S_t, A_t) \leftarrow Q^k(S_t, A_t) + \alpha \underbrace{[R_{t+1} + \gamma Q^k(S_{t+1}, A_{t+1}) - Q^k(S_t, A_t)]}_{\text{TD error}}$$

In both cases, the new estimate of the value function is a linear combination of the previous estimate and the “TD error”.

TD(0) control - SARSA

It **estimates the expected return** for state-action pairs assuming the current policy continues to be followed: **on-policy** update for Q .

$$Q(S_t, A_t) = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$$

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$

Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$

Loop for each episode:

Initialize S

Choose A from S using policy derived from Q (e.g., ε -greedy)

Loop for each step of episode:

Take action A , observe R, S'

Choose A' from S' using policy derived from Q (e.g., ε -greedy)

$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma Q(S', A') - Q(S, A)]$

$S \leftarrow S'; A \leftarrow A';$

until S is terminal

TD(0) control - Q-learning

It **estimates the expected return** for state-action pairs **assuming a greedy policy** were followed despite the fact that it's not necessarily following a greedy policy: **off-policy** update for Q.

$$Q(S_t, A_t) = R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$$

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$

Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$

Loop for each episode:

Initialize S

Loop for each step of episode:

Choose A from S using policy derived from Q (e.g., ε -greedy)

Take action A , observe R, S'

$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$

$S \leftarrow S'$

until S is terminal

More generally, off-policy means that the return is computed using a different policy from the one used to choose the next action (i.e., the policy through which we “explore” the environment).

TD(0) summary

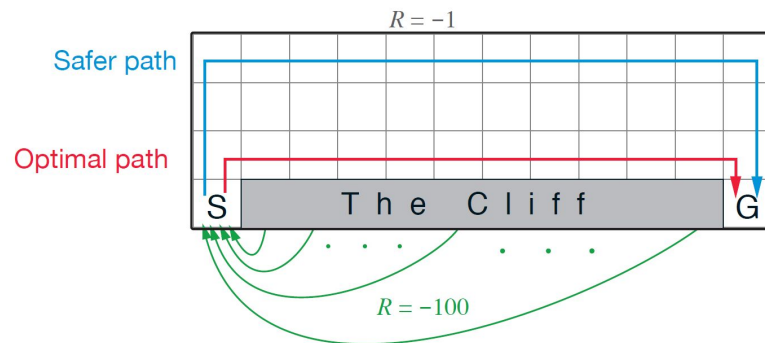
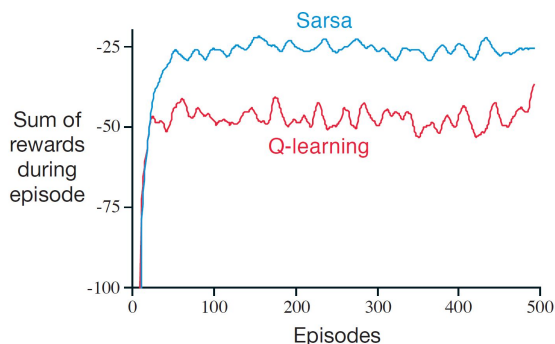
Pros (on-line method):

- Improved sample efficiency: converges faster than MC.
- Applicable to continuing tasks.

Cons (due to bootstrapping): $V(S_t) = R_{t+1} + \gamma V(S_{t+1})$

- **Biased estimate** of the return.
- Difficult to **propagate sparse rewards** through *bootstrapped returns*.
- More susceptible to the **violation of Markov property**. Harder to reconstruct the whole interactions “storyline”.

SARSA vs. Q-learning:



Model-based methods

The interaction with the environment may be **expensive**:

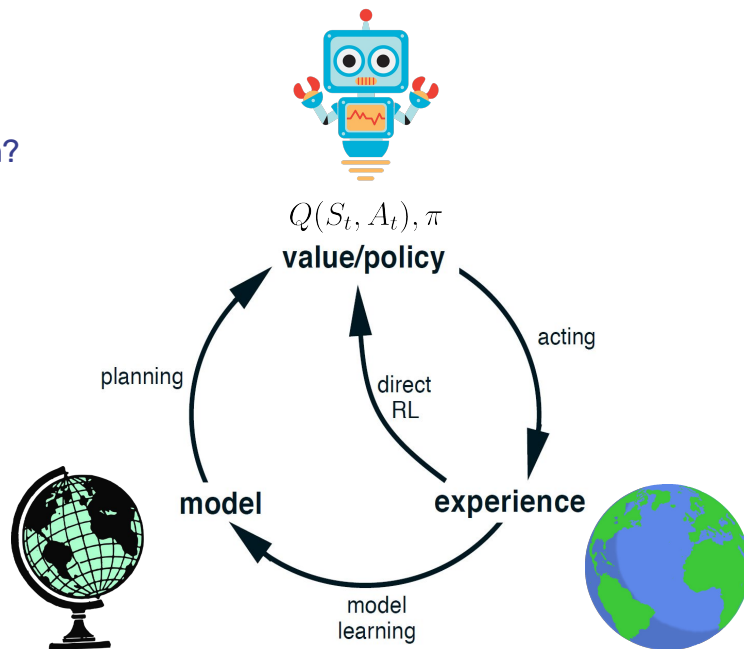
- **Slow** to respond (e.g., human)
- **Costs** of operating the environment (e.g., LHC)
- Env can be **hardly reachable** (e.g., Mars)

...how can we train our agent well, **without interacting too much**?

Keep a model of the environment and sample (also) from it!

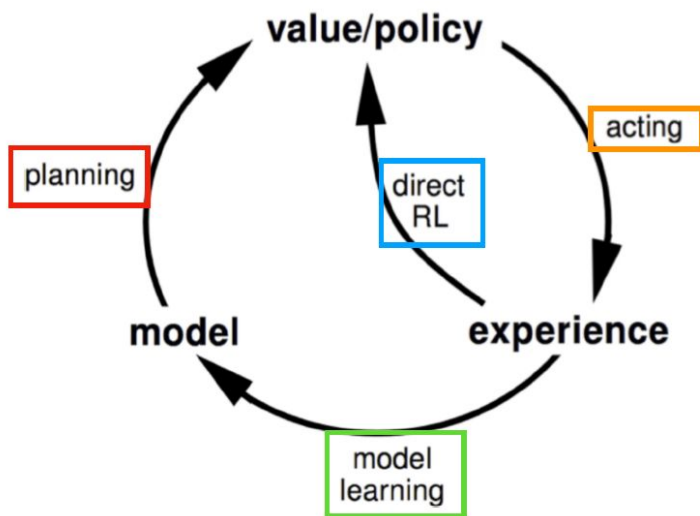
General idea:

1. Interact with the real env: $S_t, A_t, R_{t+1}, S_{t+1}$
2. Update $Q(s,a)$ based on real env
3. Update the model
4. For N times:
 - a. Sample from model: $S_t, A_t, R_{t+1}, S_{t+1}$
 - b. Update $Q(s,a)$ based on model



Tabular Dyna-Q

Model: store previous interactions with the environment. Can only sample state-action pairs visited previously.



Initialize $Q(s, a)$ and $Model(s, a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$

Do forever:

(a) $S \leftarrow$ current (nonterminal) state

(b) $A \leftarrow \varepsilon$ -greedy(S, Q)

(c) Execute action A ; observe resultant reward, R , and state, S'

(d) $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$

(e) $Model(S, A) \leftarrow R, S'$ (assuming deterministic environment)

(f) Repeat n times:

$S \leftarrow$ random previously observed state

$A \leftarrow$ random action previously taken in S

$R, S' \leftarrow Model(S, A)$

$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$

Tabular Dyna-Q

Model: store previous interactions with the environment. Can only sample state-action pairs visited previously.

Tabular Dyna-Q

Initialize $Q(s, a)$ and $Model(s, a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$

Loop forever:

- (a) $S \leftarrow$ current (nonterminal) state
- (b) $A \leftarrow \varepsilon$ -greedy(S, Q)
- (c) Take action A ; observe resultant reward, R , and state, S'
- (d) $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$
- (e) $Model(S, A) \leftarrow R, S'$ (assuming deterministic environment)
- (f) Loop repeat n times:
 - $S \leftarrow$ random previously observed state
 - $A \leftarrow$ random action previously taken in S
 - $R, S' \leftarrow Model(S, A)$
 - $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$

Model-based methods - summary

Pros:

- Improve sample efficiency: **reduce the number of costly interactions** with the environment.

Cons:

- The **model** may be **wrong** or **outdated** (non-stationary environment). How often update the model?

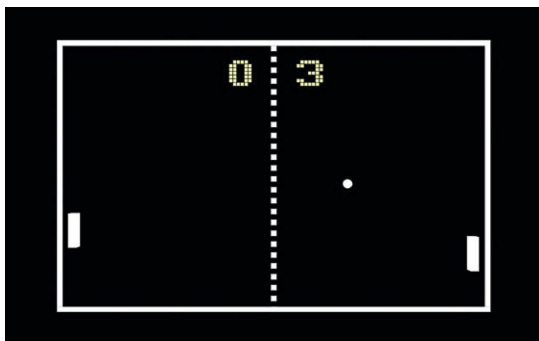
Function approximation



Limitations of tabular methods

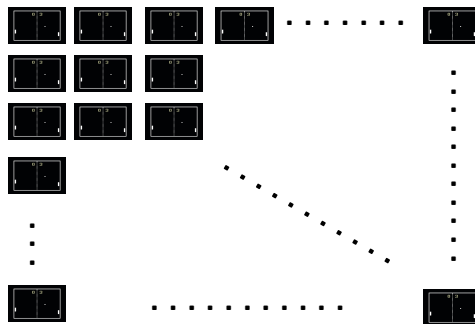
Tabular methods encountered: Monte Carlo, SARSA, Q-learning, tabular Dyna-Q.

- State space \mathcal{S} and action space \mathcal{A} **must be discrete**: define rows and columns in the Q-table.
- Q-table values are learned independently: requires many interactions and cannot infer the value from a “similar” state: **there is no generalization across similar states** (or state-action pair).
- **Don't scale** to problems with a large number (millions) of states.



Atari's "Pong"

How do we represent this in a table?
One entry for each unique combination
of the colors of all pixels?



Can we find a
better way?

Function approximation - value function

Function approximation shifts the task of learning the values for each state or state-action pair to learning a parameterized version of the value functions that minimizes a given objective.

The parametric state value function is $\hat{v}(s; \mathbf{w})$ with parameters \mathbf{w} .

The objective to minimize is the Mean Squared Error in the approximation of $v_\pi(s)$ by $\hat{v}(s; \mathbf{w})$.

$$\overline{VE}(\mathbf{w}) \doteq \sum_s \mu(s) \left[v_\pi(s) - \hat{v}(s; \mathbf{w}) \right]^2$$

where $\mu(s)$ is the proportion of times state s was visited.

The objective above can be minimized by means of Stochastic Gradient Descent (SGD), obtaining the update rule:

$$\begin{aligned} \mathbf{w}_{k+1} &\doteq \mathbf{w}_k - \frac{1}{2} \alpha_k \nabla_{\mathbf{w}} \left[v_\pi(s) - \hat{v}(s; \mathbf{w}) \right]^2 \\ &= \mathbf{w}_k + \alpha_k \left[v_\pi(s) - \hat{v}(s; \mathbf{w}) \right] \nabla_{\mathbf{w}} \hat{v}(s; \mathbf{w}) \end{aligned}$$


A similar reasoning holds for the parametric state-action value function: $\hat{q}(s, a; \mathbf{w})$

Function approximation - value function

Since the value function $v_\pi(s)$ is unknown, we substitute it with an unbiased estimator of it: U_t .

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha_k \left[U_t - \hat{v}(s; \mathbf{w}) \right] \nabla_{\mathbf{w}} \hat{v}(s; \mathbf{w})$$

- Monte Carlo: $U_t = G_t$
- TD(0): $U_t = R_{t+1} + \gamma \hat{v}(S_{t+1}; \mathbf{w})$. Is bootstrap legit???



Depends on the
trainable
parameters!

Function approximation - value function

Since the value function $v_\pi(s)$ is unknown, we substitute it with an unbiased estimator of it: U_t .

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha_k \left[U_t - \hat{v}(s; \mathbf{w}) \right] \nabla_{\mathbf{w}} \hat{v}(s; \mathbf{w})$$

- Monte Carlo: $U_t = G_t$
- TD(0): performs a *semi-gradient* update (do not use the full gradient information).

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w}) \right] \nabla \hat{v}(S, \mathbf{w})$$

Nice but... how to put everything together?

- Gradient update
- RL interactions
- Policy update

Function approximation - value function

Example of **on-policy** control with **function approximation**: SARSA.

Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \rightarrow \mathbb{R}$

Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$

Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:

$S, A \leftarrow$ initial state and action of episode (e.g., ε -greedy)

Loop for each step of episode:

Take action A , observe R, S'

If S' is terminal:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha [R - \hat{q}(S, A, \mathbf{w})] \nabla \hat{q}(S, A, \mathbf{w})$$

Go to next episode

Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε -greedy)

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha [R + \gamma \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})] \nabla \hat{q}(S, A, \mathbf{w})$$

$S \leftarrow S'$

$A \leftarrow A'$

Function approximation - DQN

Train RL agent to play Atari games. **Feature extractor**: convolutional neural network (CNN). **Reward**: +1 if scored a point, -1 otherwise.

Example of **off-policy** control with **function approximation**. The Q function is approximated with a neural network

Gameplay



Image [credits](#).

Experience replay

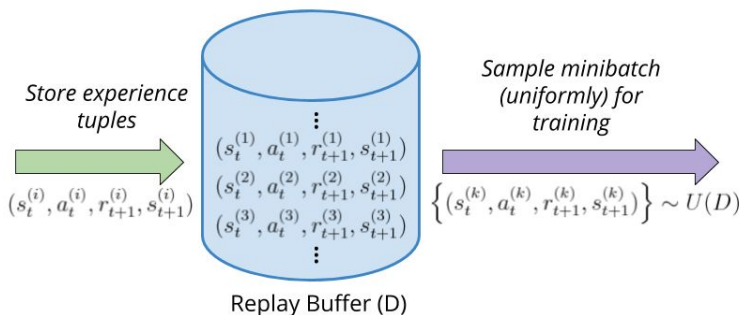


Image [credits](#).

DQN

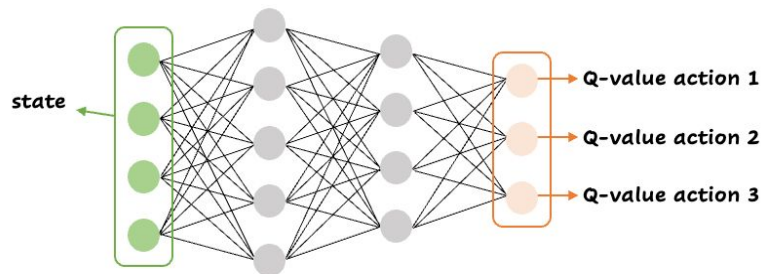


Image [credits](#).

Off-policy: DQN is updated using "old" transitions sampled from the replay buffer

Function approximation - DQN

Continuous state space: embedding vector produced by the feature extractor (CNN, part of DQN).

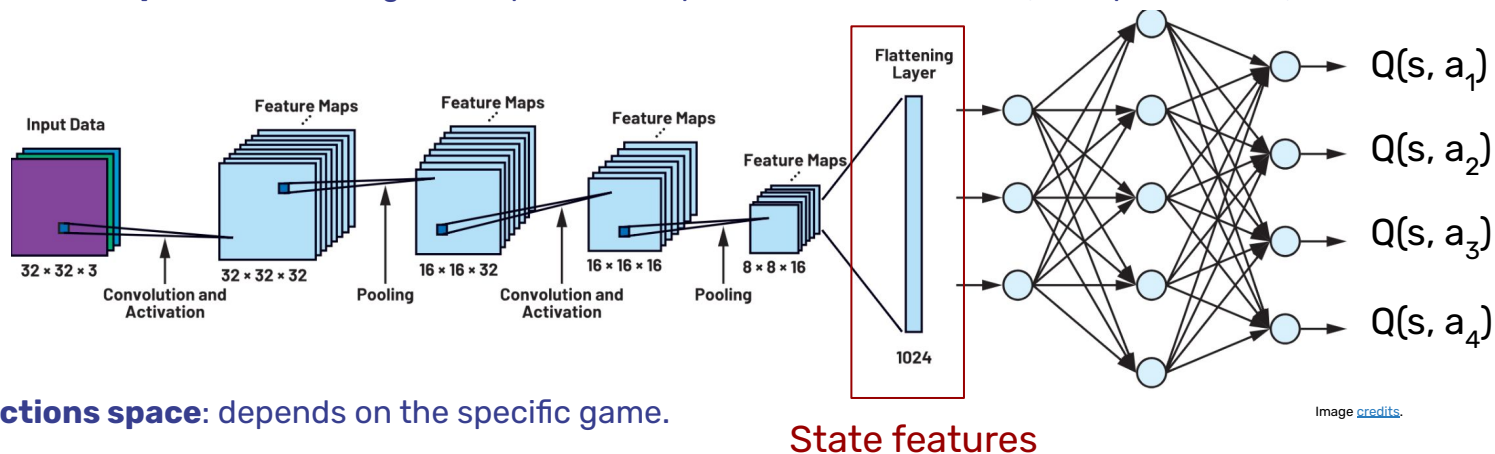


Image [credits](#).

Discrete actions space: depends on the specific game.

Experience replay:

- Similar idea as Dyna-Q for model-based RL.
- Improves sample efficiency: DNNs are data hungry.
- **Decorrelates samples** in the training batch. Good for convergence properties and training stability.

Function approximation - DQN

Update rule:

1. Sample a batch of B transitions $\{(s_i, a_i, r_i, s'_i), \dots\}$ from the replay buffer (hereafter #B=1).
2. Compute the loss:

$$\mathcal{L} = \left(r + \gamma \max_a \tilde{q}(S_{t+1}, a; \mathbf{w}_t^-) - q(S_t, A_t; \mathbf{w}_t) \right)^2 = (\text{TD error})^2$$

3. Gradient update:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[R_{t+1} + \gamma \max_a \tilde{q}(S_{t+1}, a, \mathbf{w}_t^-) - \hat{q}(S_t, A_t, \mathbf{w}_t) \right] \nabla \hat{q}(S_t, A_t, \mathbf{w}_t)$$

“Target” network

“Policy” network

Function approximation - DQN

Target and policy networks visualized:

Copy parameters every K steps

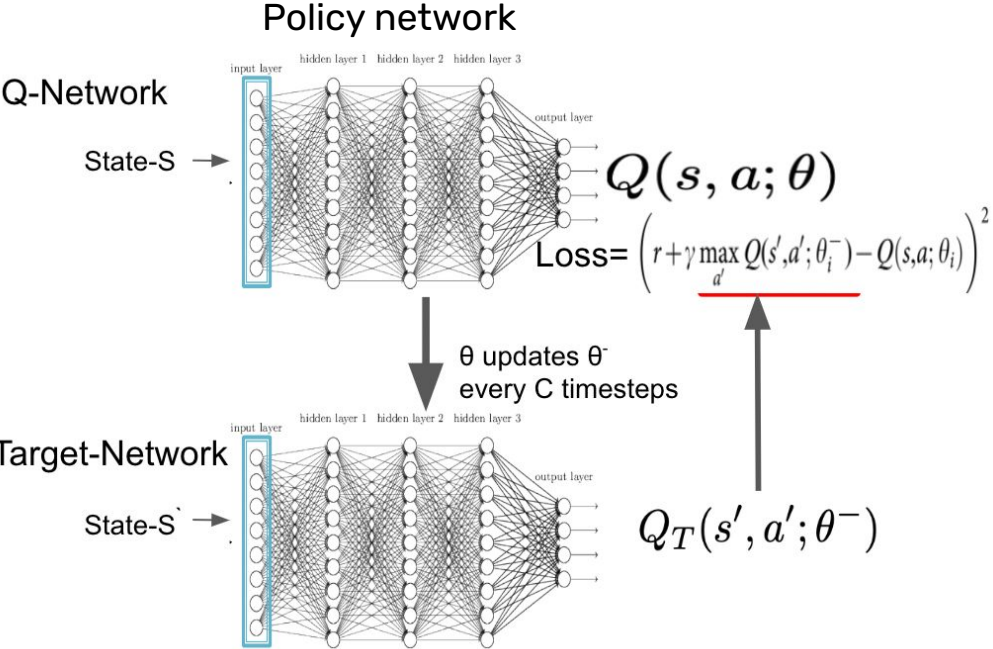


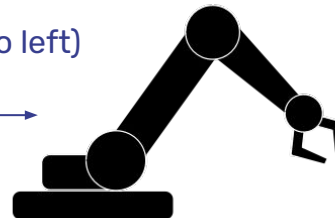
Image credits.

Function approximation – policy gradient

So far we obtained the optimal greedy policy from the (approx) optimal value function $q_*(s, a) \approx Q_*(s, a)$

However, in some cases it would be more convenient to learn directly the policy!

- State space may be “complex”, whereas the **policy could be “easy”** (e.g., always go left)
- ϵ -greedy is generally suboptimal: with ϵ probability **take random action**.
- **Continuous action space** (e.g., robotic arm control).
- Greedy policy may be suboptimal (e.g., “rock, paper, scissor”).



A greedy policy is easily exploited by the opponent.

Example:
Always draw paper

Function approximation - policy gradient

Write the policy as a parametrized function:

$$\pi(a|s, \boldsymbol{\theta}) = \Pr\{A_t = a \mid S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}\}$$

For instance:

$$\pi(a|s, \boldsymbol{\theta}) \doteq \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_b e^{h(s,b,\boldsymbol{\theta})}} \quad h(s, a, \boldsymbol{\theta}) \text{ Is called } \textit{actions preferences}. \text{ Can by a neural network.}$$

Define an objective, for instance:

$$J(\boldsymbol{\theta}) \doteq v_{\pi_{\boldsymbol{\theta}}}(s_0)$$

And the corresponding param. update rule:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha G_t \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)}$$

Function approximation - policy gradient

Episodic case

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$

Algorithm parameter: step size $\alpha > 0$

Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):

Generate an episode $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$

Loop for each step of the episode $t = 0, 1, \dots, T - 1$:

$$G \leftarrow \sum_{k=t+1}^T \gamma^{k-t-1} R_k \quad (G_t)$$

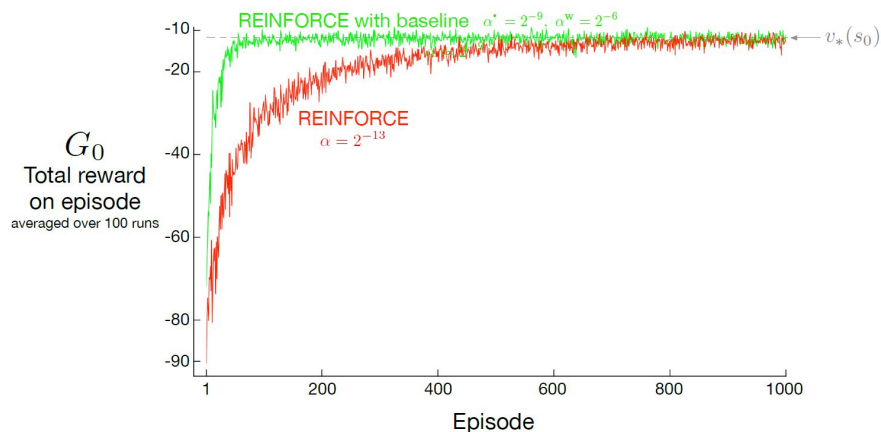
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$$

Function approximation - policy gradient

REINFORCE is based on MC -> high variance.

Adding a “baseline” leaves the expected value of the update unchanged, but it can have a large effect on its variance.

$$\theta_{t+1} \doteq \theta_t + \alpha G_t \frac{\nabla \pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)} \quad \longrightarrow \quad \theta_{t+1} \doteq \theta_t + \alpha \left(G_t - b(S_t) \right) \frac{\nabla \pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)}$$



For instance:
 $b(S_t) = \hat{v}(S_t; \mathbf{w})$

Figure 13.2: Adding a baseline to REINFORCE can make it learn much faster, as illustrated here on the short-corridor gridworld (Example 13.1). The step size used here for plain REINFORCE is that at which it performs best (to the nearest power of two; see Figure 13.1).

Function approximation - policy gradient

REINFORCE with baseline

REINFORCE with Baseline (episodic), for estimating $\pi_* \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s, \theta)$

Input: a differentiable state-value function parameterization $\hat{v}(s, \mathbf{w})$

Algorithm parameters: step sizes $\alpha^{\cdot} > 0$, $\alpha^{\mathbf{w}} > 0$

Initialize policy parameter $\theta \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):

Generate an episode $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \theta)$

Loop for each step of the episode $t = 0, 1, \dots, T - 1$:

$$G \leftarrow \sum_{k=t+1}^T \gamma^{k-t-1} R_k \quad (G_t)$$

$$\delta \leftarrow G - \hat{v}(S_t, \mathbf{w}) \quad \longleftarrow \text{Kind of "loss", depends on } G_t$$

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S_t, \mathbf{w})$$

$$\theta \leftarrow \theta + \alpha^{\cdot} \gamma^t \delta \nabla \ln \pi(A_t | S_t, \theta)$$

Function approximation - actor-critic

REINFORCE based on MC -> **off-line**.

Solution: substitute MC return with its bootstrapped version. Becomes **on-line**.

Keep using value function as a baseline: $b(S_t) = \hat{v}(S_t; \mathbf{w})$

$$G_t = \sum_{k=t+1}^T \gamma^{k-t-1} R_k \quad \longrightarrow \quad G_t = R_{t+1} + \gamma \hat{v}(S_t; \mathbf{w})$$

Update rules become:

Value function params (critic)

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha [R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})] \nabla \hat{v}(S, \mathbf{w})$$

Policy function params (actor)

$$\begin{aligned} \boldsymbol{\theta}_{t+1} &\doteq \boldsymbol{\theta}_t + \alpha \left(G_{t:t+1} - \hat{v}(S_t, \mathbf{w}) \right) \frac{\nabla \pi(A_t | S_t, \boldsymbol{\theta}_t)}{\pi(A_t | S_t, \boldsymbol{\theta}_t)} \\ &= \boldsymbol{\theta}_t + \alpha \left(R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}) \right) \frac{\nabla \pi(A_t | S_t, \boldsymbol{\theta}_t)}{\pi(A_t | S_t, \boldsymbol{\theta}_t)} \end{aligned}$$

Note: they may use different alphas.

Function approximation - actor-critic

One-step Actor-Critic (episodic), for estimating $\pi_* \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s, \theta)$

Input: a differentiable state-value function parameterization $\hat{v}(s, \mathbf{w})$

Parameters: step sizes $\alpha^a > 0$, $\alpha^w > 0$

Initialize policy parameter $\theta \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):

 Initialize S (first state of episode)

$I \leftarrow 1$

 Loop while S is not terminal (for each time step):

$A \sim \pi(\cdot|S, \theta)$

 Take action A , observe S', R

$\delta \leftarrow R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$ (if S' is terminal, then $\hat{v}(S', \mathbf{w}) \doteq 0$)

$\mathbf{w} \leftarrow \mathbf{w} + \alpha^w \delta \nabla \hat{v}(S, \mathbf{w})$

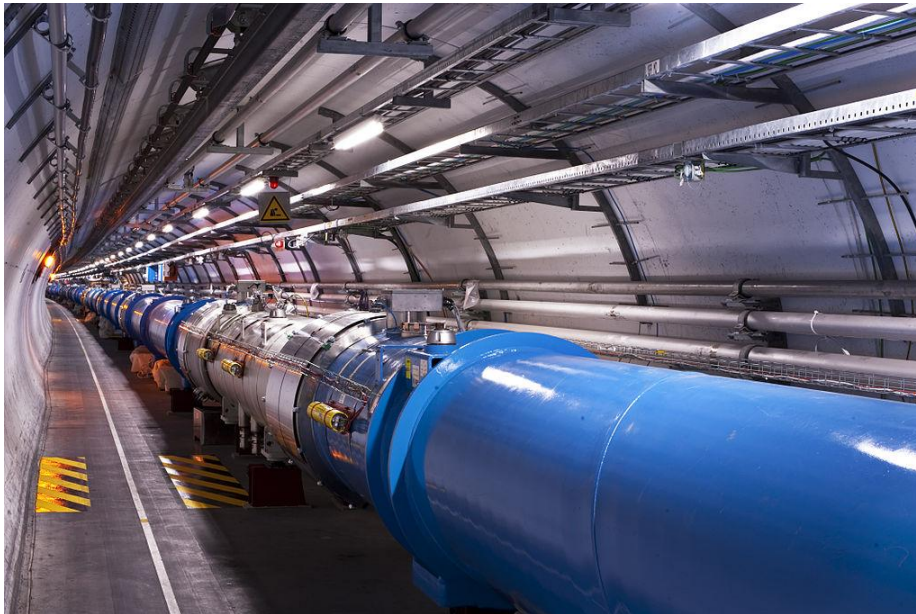
$\theta \leftarrow \theta + \alpha^a I \delta \nabla \ln \pi(A|S, \theta)$

$I \leftarrow \gamma I$

$S \leftarrow S'$

Episodic case because for the continuing case we need to change the objective function $J(\theta)$ (out of the context of this lecture)

RL for particle accelerators



AWAKE - background

Problem: difficult to accelerate electrons in the LHC. Due to curvature, they emit radiation and lose energy. Since the circumference of the LHC is finite, it can provide a finite amount of energy to the particles. This results in a **dynamic equilibrium**.

A possible solution could be using **linear/larger** accelerators (e.g., FCC), or **improving how we accelerate** particles.

Wakefield Acceleration Experiment: experimental beamline exploring innovative acceleration techniques.

Use two beams in the same line:

$e^-e^-e^-e^- p^+p^+p^+p^+$

Go through a plasma cell:

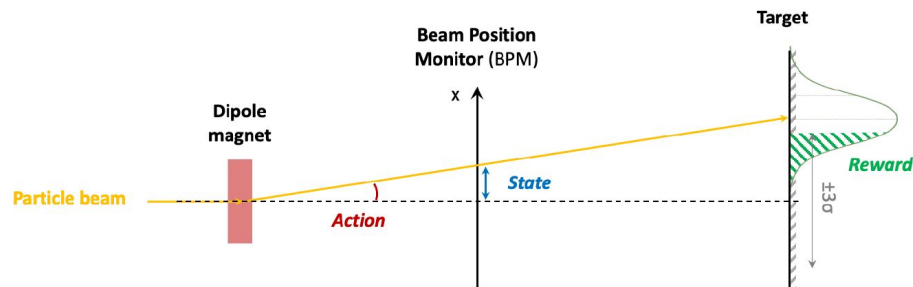
- The protons ionise the plasma, creating a *wakefield*: this corresponds to a **very strong electric field**.
- The electrons “surf” the wakefield like a wave, receiving a **very strong acceleration**.

Wakefield acceleration could replace current **RF cavities** in the far future.



AWAKE - beam steering

RL problem: keep the electrons beam within the beamline and steer them into the plasma to best exploit the wakefield.

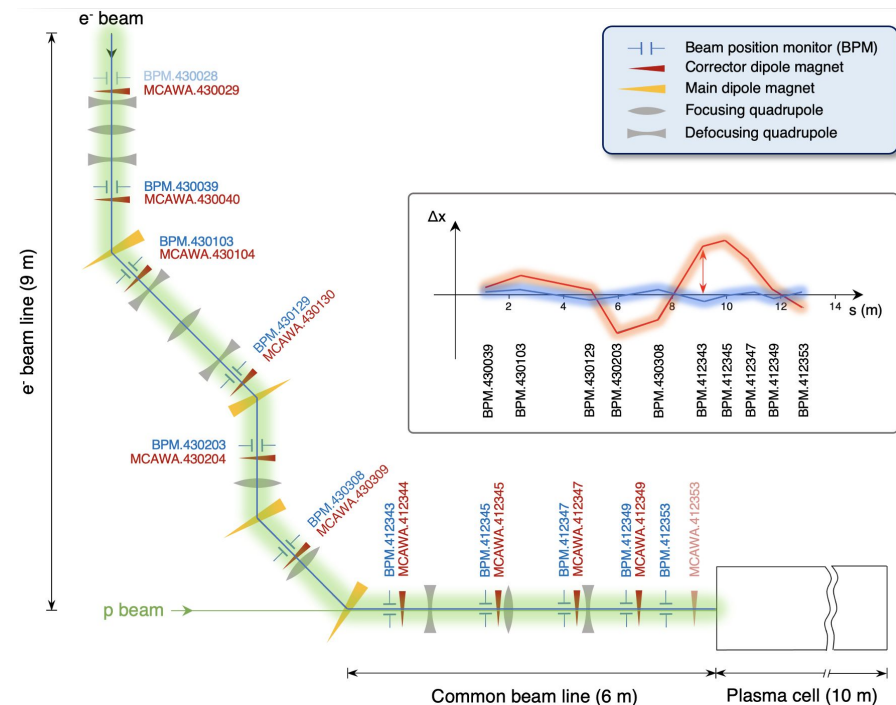


States: readouts from 10 sensors (BPMs)

Actions: 10 correctors

Simulation of the beamline available: train the agent in simulation and fine-tune it on the real machine.

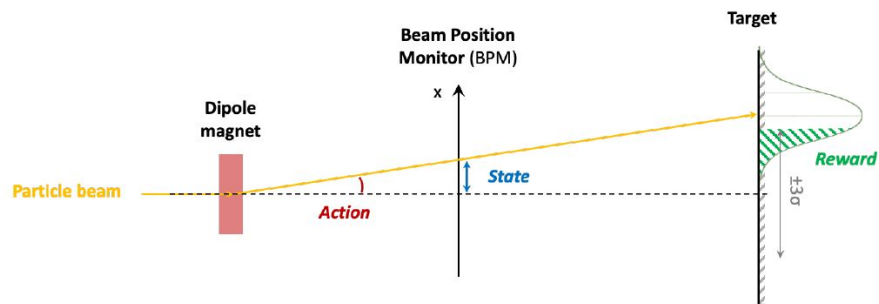
...remember the expensive agent-environment interactions mentioned before?



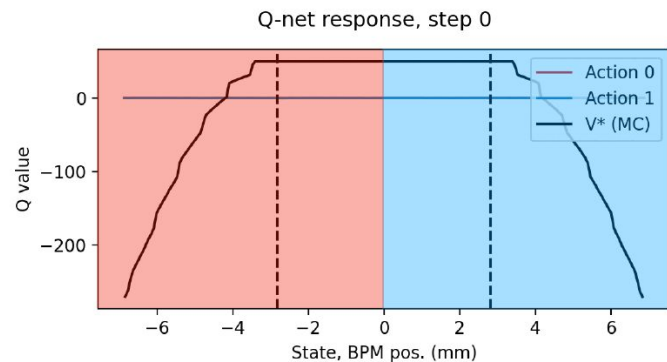
Beam-time is expensive:
the agent has to learn fast!
Sample efficiency is critical.

AWAKE - beam steering

Illustration of Q-learning: 1D beam steering



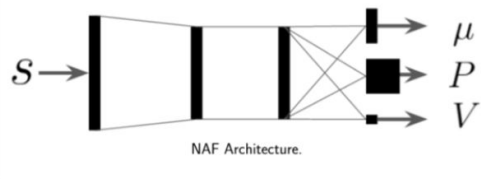
- **Action:** deflection angle
- **State:** beam position at BPM
- **Reward:** integrated beam intensity on target



AWAKE - Q-learning with NAF

- Standard DQN only applicable to discrete action tasks
- Our control tasks: typically **continuous state-action spaces**
- Various ways to extend Q-learning to continuous action tasks
 - Most successful: actor-critic algorithms, e.g. DDPG and extensions
 - If convex problem, can use trick: assume Q-function belongs to function class that is easy to optimise, e.g., NAF (Normalized Advantage Function)

$$Q_{\phi}(\mathbf{s}, \mathbf{a}) = -\frac{1}{2}(\mathbf{a} - \mu_{\phi}(\mathbf{s}))^T P_{\phi}(\mathbf{s})(\mathbf{a} - \mu_{\phi}(\mathbf{s})) + V_{\phi}(\mathbf{s})$$



Assumption:
 $Q(s,a)$ is quadratic in \mathbf{a}

$$\arg \max_{\mathbf{a}} Q_{\phi}(\mathbf{s}, \mathbf{a}) = \mu_{\phi}(\mathbf{s})$$

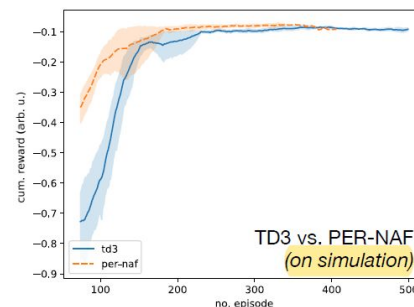
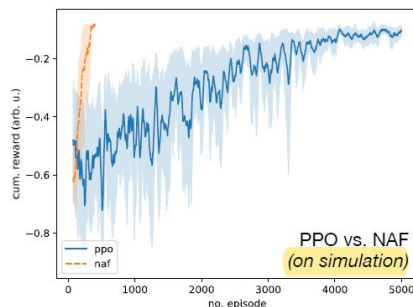
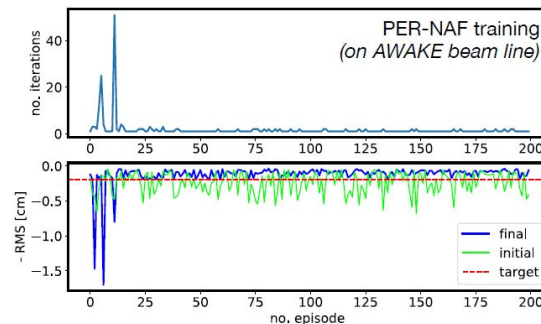
$$\max_{\mathbf{a}} Q_{\phi}(\mathbf{s}, \mathbf{a}) = V_{\phi}(\mathbf{s})$$

Gu, Lillicrap, Sutskever, L., ICML 2016

AWAKE - train Q-learning on simulation

Proof-of-principle: learn to correct AWAKE e^- -line in H (10 DOF)

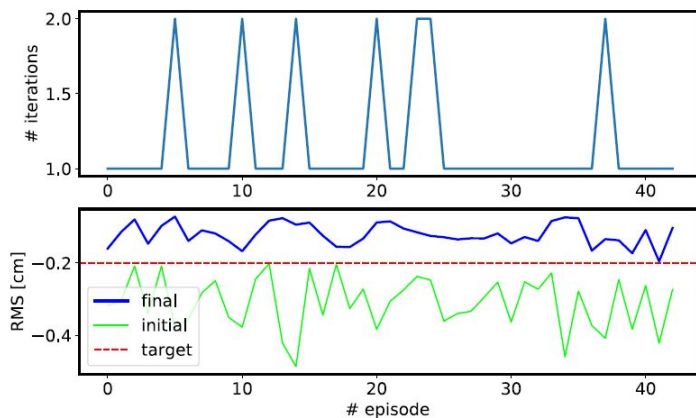
- Implemented **NAF with *Prioritised Experience Replay*** (*arXiv:1511.05952*)
- After training for **~90 iterations**, agent starts correcting well
- Any initial steering is corrected to below target RMS **within 1-2 steps**



→ Q -learning derives more sample efficient than policy gradient algorithm (PPO)

AWAKE - train Q-learning on simulation

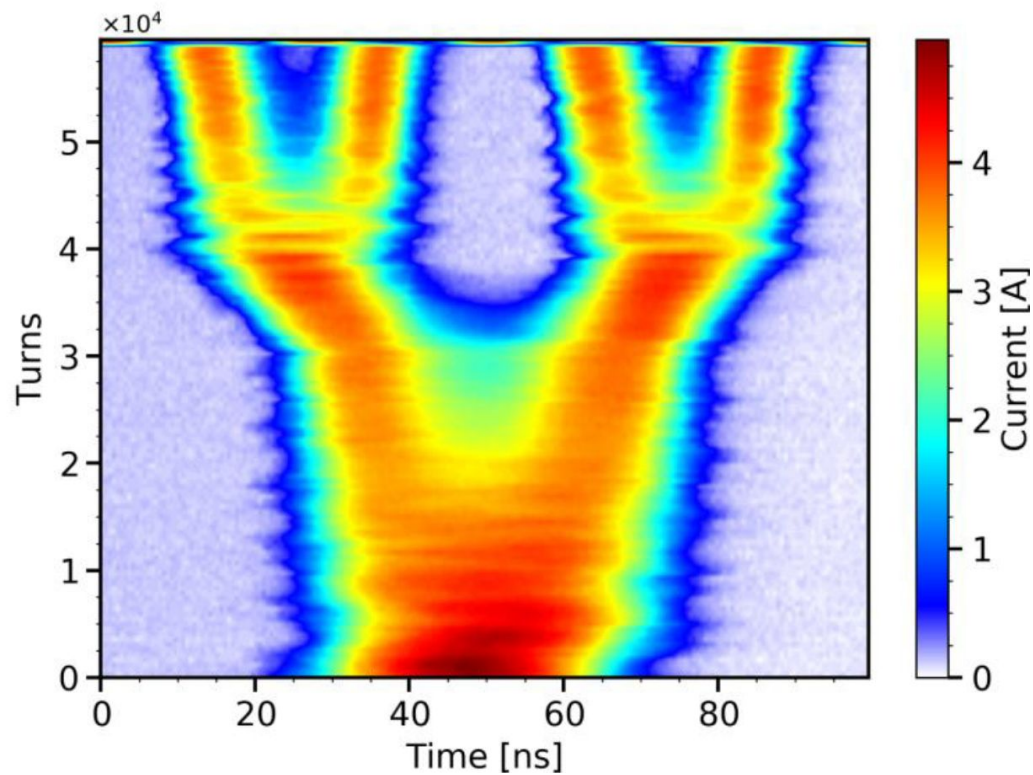
- Train PER-NAF agent *exclusively on simulation*
- For this particular task, have **excellent model** at hand
- **Validate agent on actual beam line**



Performance as good as if trained directly on machine

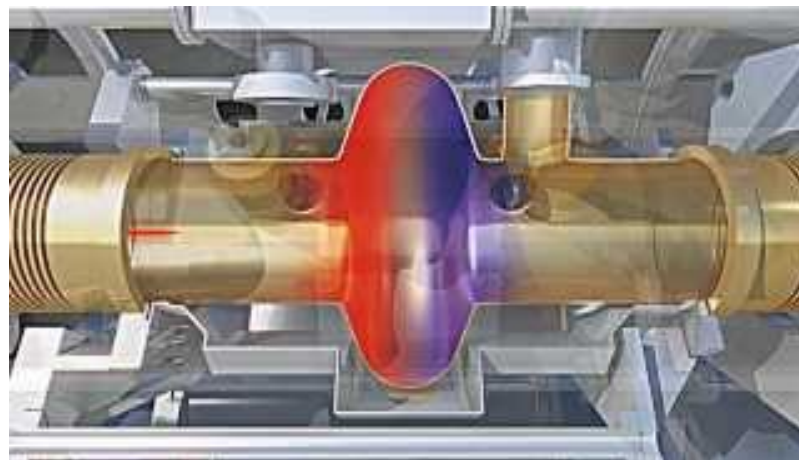
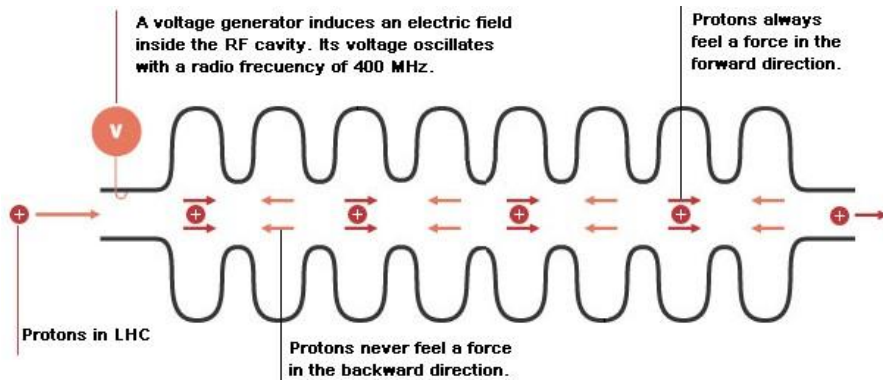
Further experiments improved sample efficiency by using model-based RL, inspired to Dyna-Q

Bunch splitting



Bunch splitting - background

RF cavities: used to accelerated particles in the LHC



Bunch splitting - background

RF cavities: used to accelerated particles... **but also to split bunches!**

The split is done in PS to prepare beams for LHC.

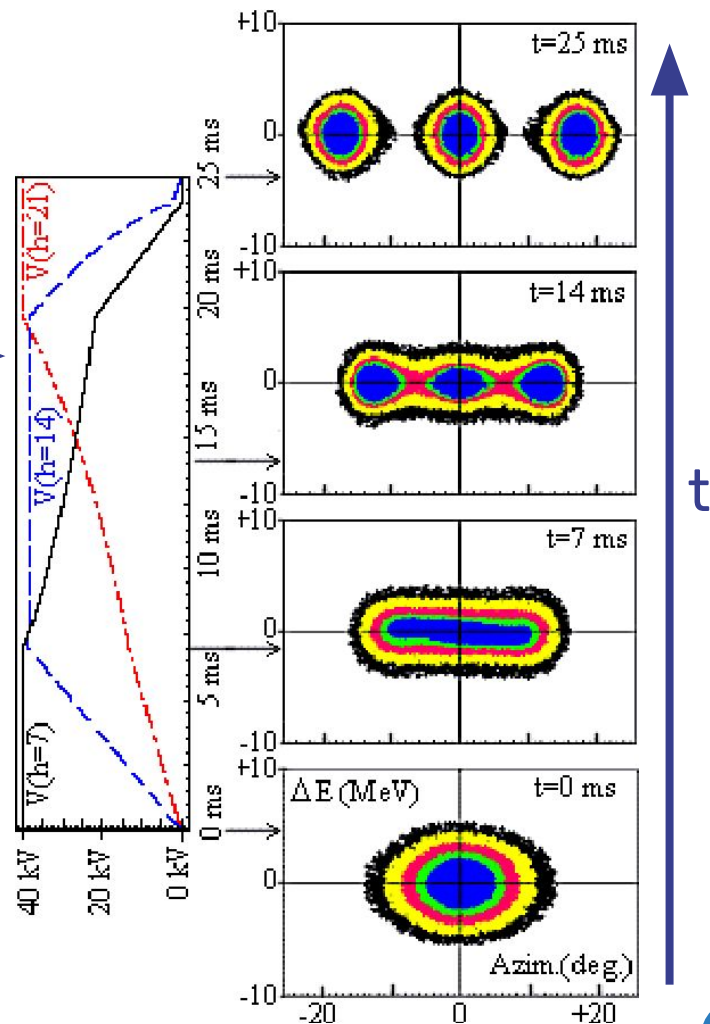
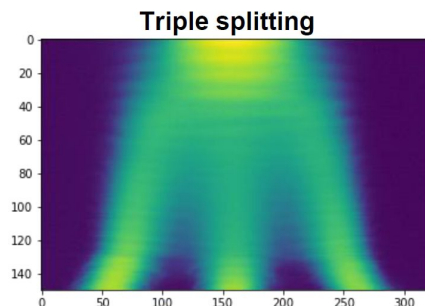
We gradually change the **intensity** (voltage) of **high harmonics** in the RF cavities: h_7, h_{14}, h_{21}

Voltage and phase has to be **dynamically adjusted** for each harmonic:

- Compensate for voltage and phase errors
- Synchronize phase with beam

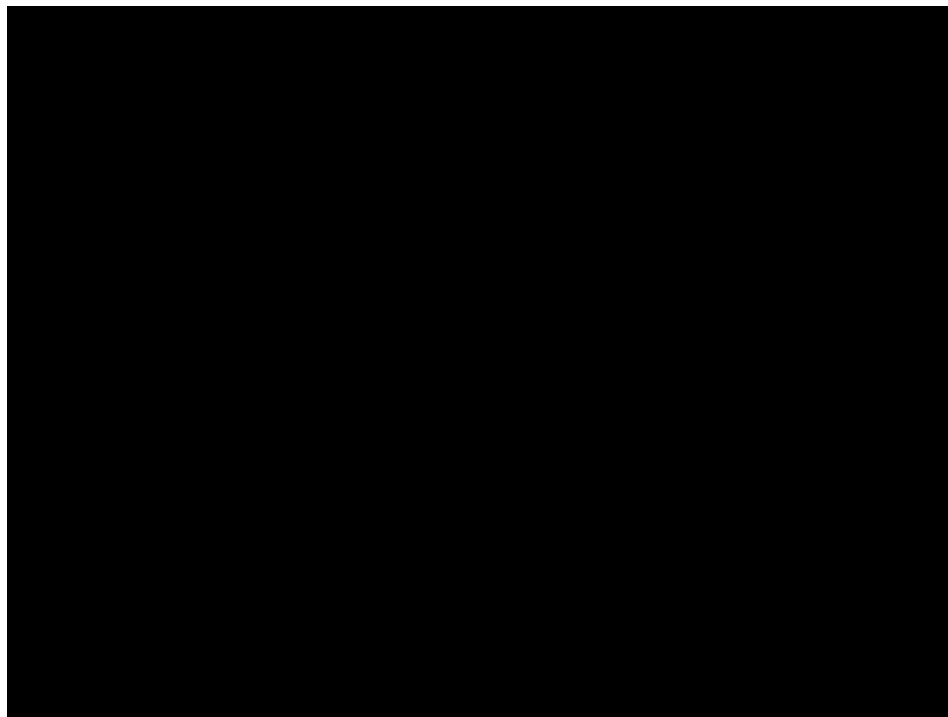
Done manually: not always reproducible.

Task: RL to optimise splittings to produce uniform bunches. *Good for science!*




Bunch splitting - background

Animation of a bunch (triple) splitting. Note the variation of intensity per harmonics.



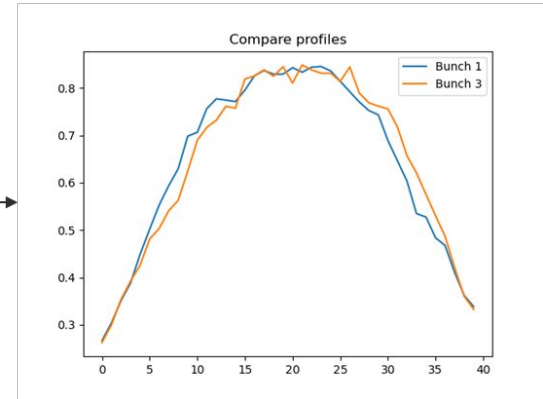
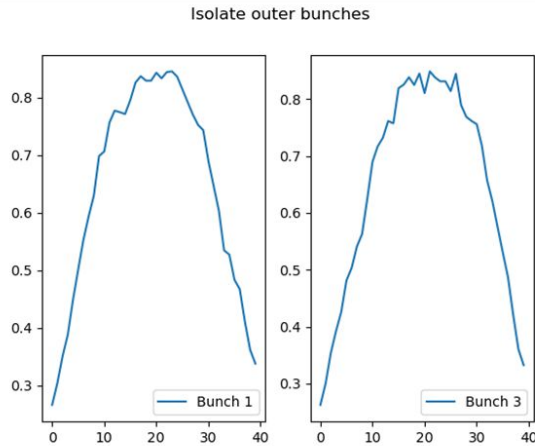
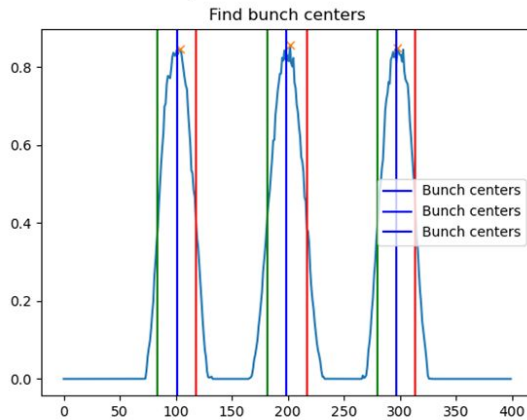
Bunch splitting - RL problem

Automating bunch splitting is good for **reproducibility**. RL-based splitting is “in production” at PS.

Challenge: reward function design.  For both phase and voltage, compare bunch profiles (Means Squared Error).


Phase profiles:

The final profile after the triple split is extracted, bunch centers located.



Bunch splitting - RL problem

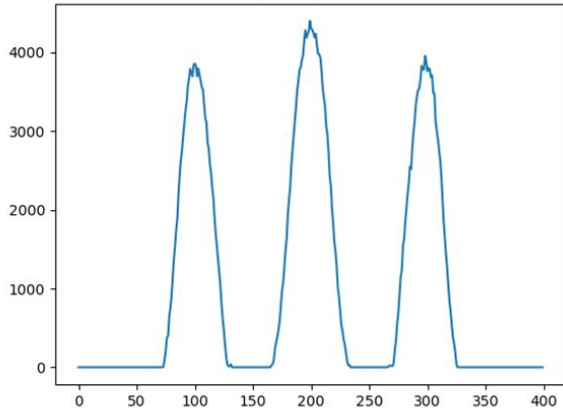
Automating bunch splitting is good for **reproducibility**. RL-based splitting is “in production” at PS.

Challenge: reward function design.  For both phase and voltage, compare bunch profiles (Means Squared Error).

Similarly, for **voltage profiles**:

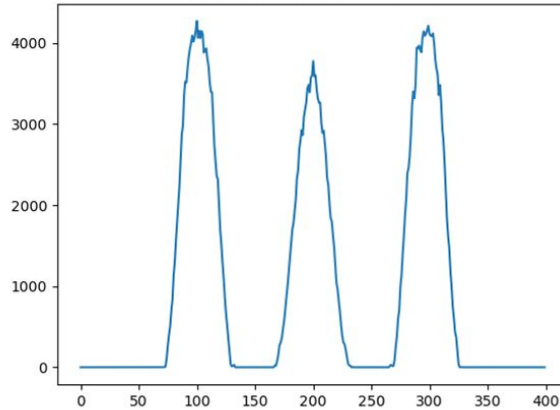
A) Middle profile larger than the others
→ **Voltage too low!**

p14: 0.0, p21: 0.0, v14: 0.95



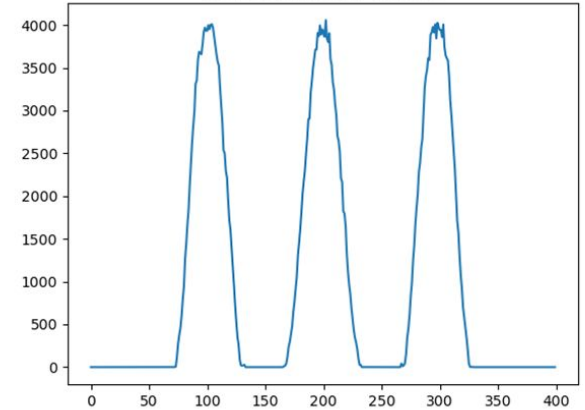
B) Middle profile smaller than the others
→ **Voltage too high!**

p14: 0.0, p21: 0.0, v14: 1.05



C) Middle profile equal to the others
→ **Voltage optimised!**

p14: 0.0, p21: 0.0, v14: 1.0



Bunch splitting - RL problem

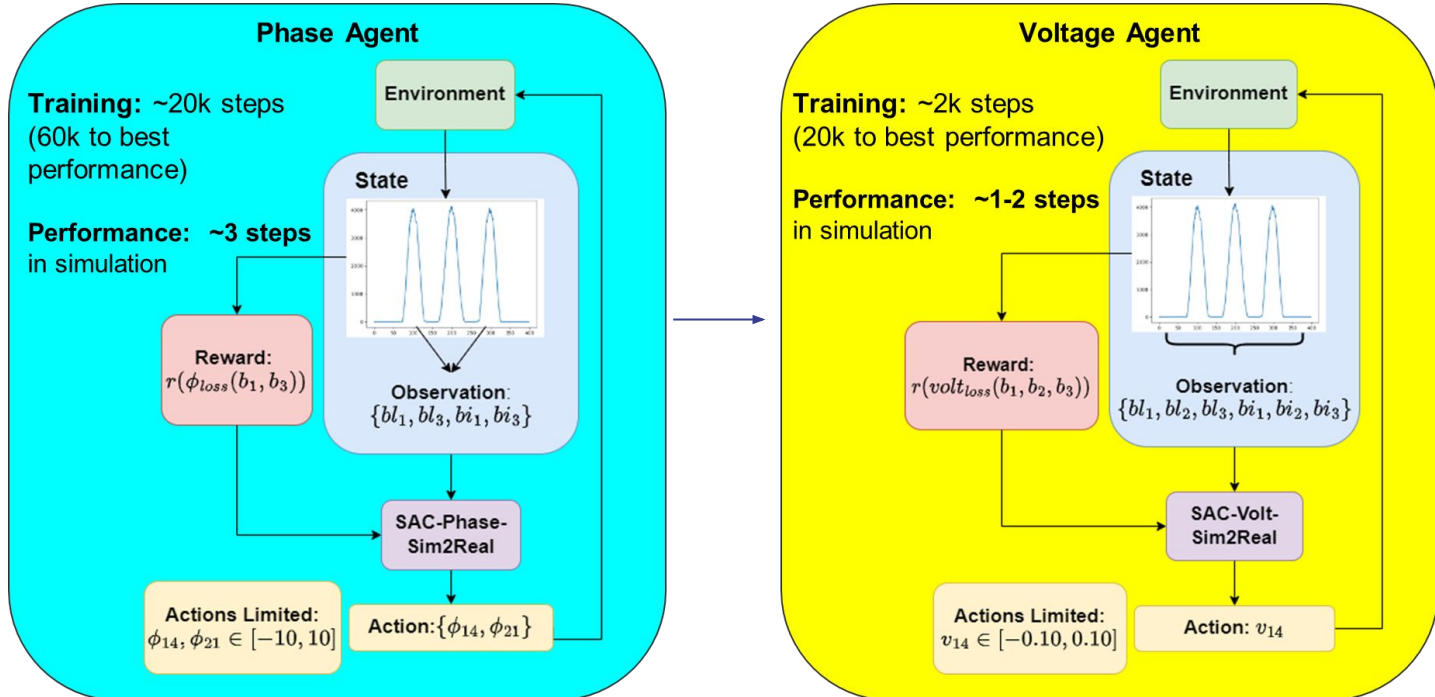
Optimization of voltage profiles is based on the assumption that the phase is already optimal.

Train two RL agents:

1. **SAC-Phase-Sim2Real:** Trained using the phase MSE loss criteria (used to define a step-wise reward) to *optimise the phase only*.
2. **SAC-Volt-Sim2Real:** Trained using the overall MSE loss (used to define a step-wise reward) to optimise the voltage only, *assuming phase is already optimised*.

Bunch splitting - RL problem

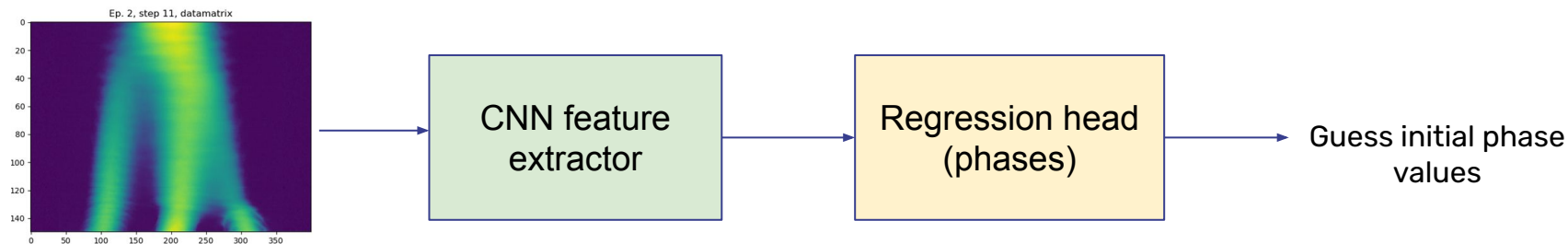
Optimization of voltage profiles is based on the assumption that the phase is already optimal.



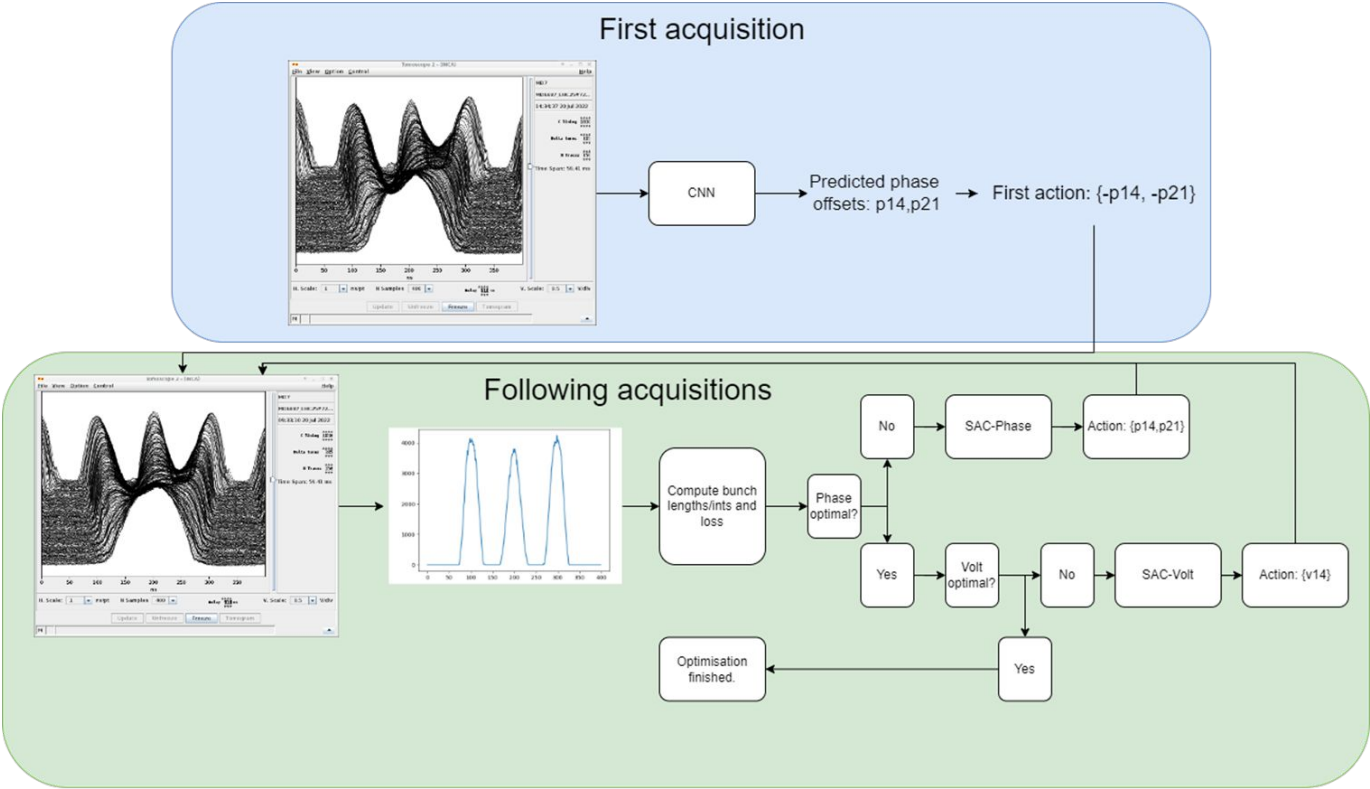
Bunch splitting - RL problem

For stability reasons, it is convenient to “bias” the agent with a prediction from a supervised CNN.

The CNN is trained on simulated data.



Bunch splitting - the big picture



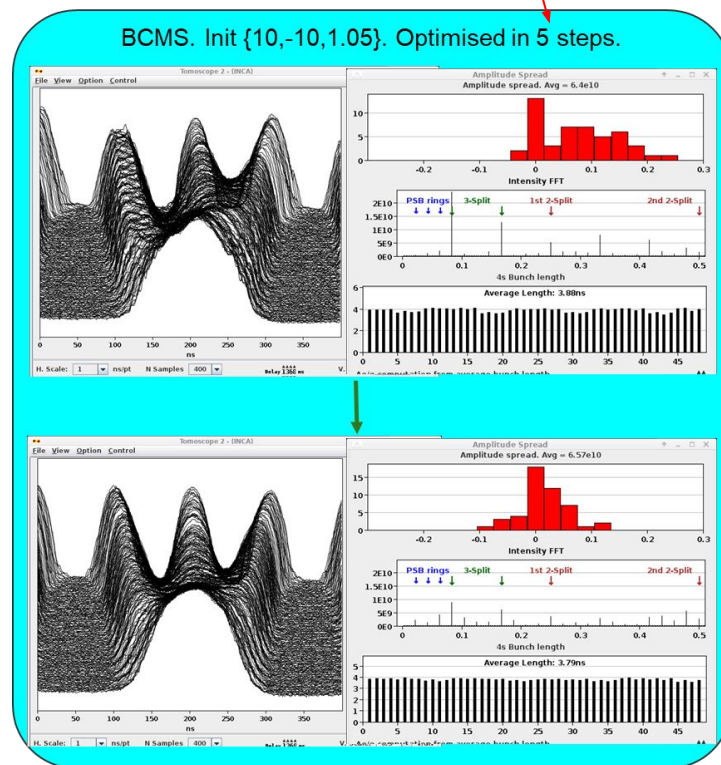
Bunch splitting - conclusions

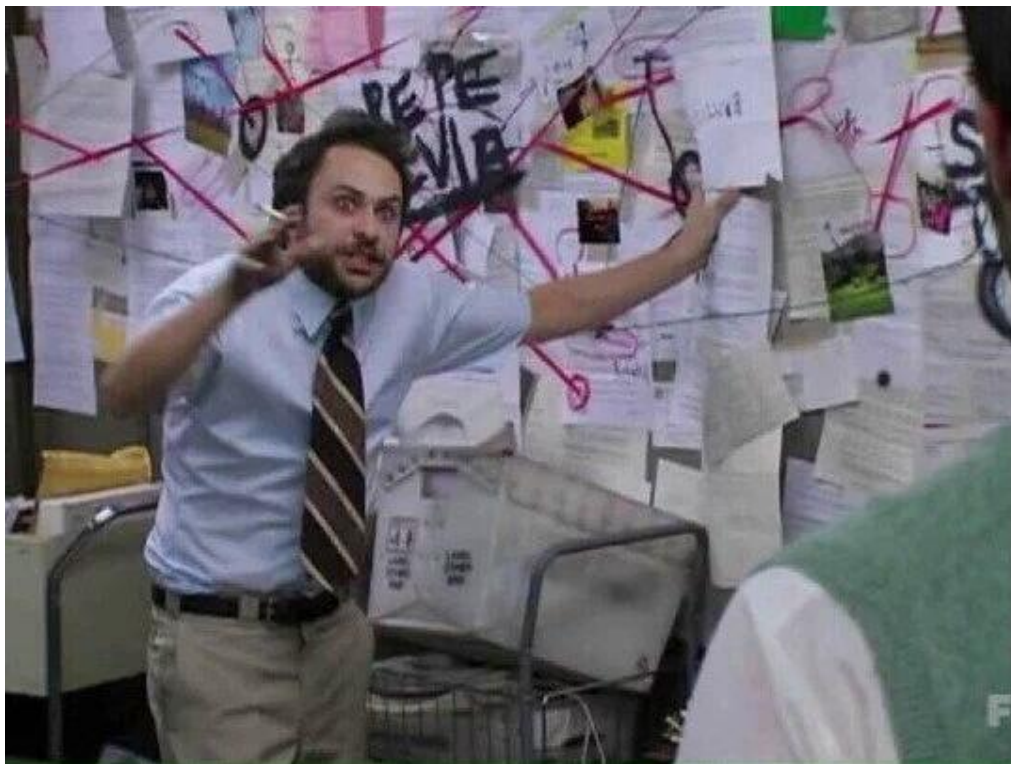
Trained on simulation and applied on machine without re-training.

Consistent good performance for:

- varying intensities ($1.3e11$ - $2.6e11$)
- different beam types (72b, BCMS)

Consistently **rivals experienced operators** in optimisation steps: averaging ~ 8.5 steps per optimisation (depending on initial conditions).





Questions?



Matteo Bunino | An introduction to RL and its applications at CERN