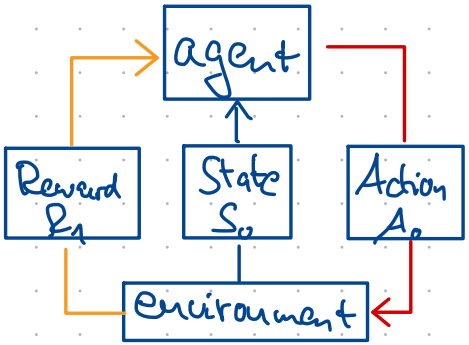
What is RL?

Reinforcement learning is a computational approach to understanding and automating goal-directed learning and decision making. It is distinguished from other computational approaches by its emphasis on learning by an agent from direct interactions with its environment.



$S_0\ A_0\ R_1\ S_1\ A_1\ R_2 \dots$

Goal: Maximize cumulative expected reward

example "walking": $r = \min(v_x, v_{max}) - 0.5 \cdot (v_y^2 + v_z^2) - 0.5\,\Sigma T + 0.02$

fast — straight — elegant — long

Discounted Reward is a trade-off between immediate reward and long-term goals:

$$G_t = \sum_{k=t}^{\infty} \gamma^k \cdot R_{t+k+1}, \quad \gamma = [0,1]$$

Other definitions:

A **policy** is a mapping from percieved states to actions to be taken.

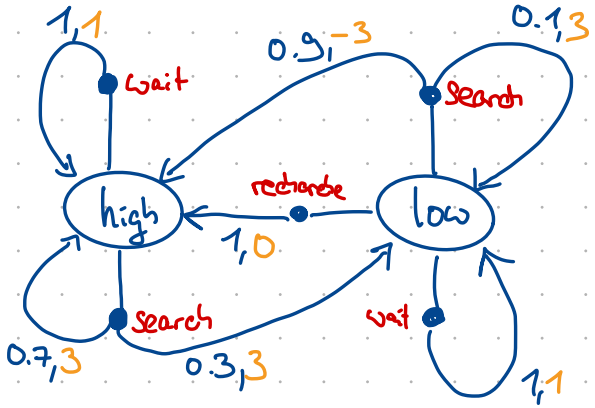
The **value** of a state is the total amount of reward an agent can expect to accumulate, starting from that state.

An **episodic task** has a temporal end, whereas a **continuous task** hasn't.

(Finite) Markov Decision Processes ("The problem")

MDPs are a formalization of sequential decision making, where the action does not only influence the immediate reward, but also subsequent states and, therefore, future rewards.

Example: transition diagram of a "garbage collector robot"



A finite MDP is defined by:

- a finite set of states S
- a finite set of actions A
- a finite set of rewards R
- the one-step dynamics of the environment
- a discount rate $\gamma \in [0,1]$

The dynamics of a finite MDP is a probability distribution for random values $s' \in S$ and $r \in R$ occuring at time t, given particular values for the previous state and action:

$$p(s', r | s, a) \doteq \mathbb{P}\{S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a\}$$
(for all $s, s', a \& r$)

→ $p: S \times R \times S \times A$ is an ordinary deterministic function of four arguments. $\sum_{s} \sum_{R} p = 1$

→ based on the dynamics, one can compute facts of interest about the environment, such as:

- the state-transition probabilities: $p(s' | s, a) \doteq \mathbb{P}\{S_{t+1} = s' \mid S_t, A_t = a) = \sum_{r \in R} p(s', r | s, a)$
- the expected rewards for state-action pairs: $r(s,a) \doteq \mathbb{E}[R_{t+1} | S_t = s, A_t = a] = \sum_{r \in R} r \sum_{s' \in S} p(s', r | s, a)$
- the expected rewards for state-action-next-state triplets

Policies and Value Functions ("The solution")

Gridworld Example:

...	...	...
-1	...	...
2	0	5

A policy is a mapping from states to probabilities of selecting each possible action.

→ $\pi(a|s)$ is the probability for choosing $A_t = a$ when in state $S_t = s$ under the policy π.

A state-value function is an estimate of the expected reward when following a policy, starting from that state.

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

The Bellmann Expectation Equation estimates the value of any state as the sum of the intermediate reward and the total discounted reward under policy π:

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma \cdot G_{t+1} | S_t = s]$$

Value functions satisfy recursive relationships!

$$= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma \cdot v_\pi(s')] \quad \text{for all } s \in S$$

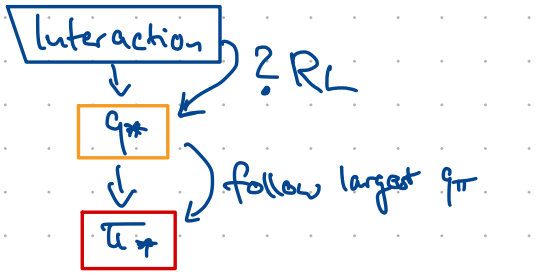
≙ multiply the sum of the reward r and discounted value $\gamma \cdot v_\pi$ of the next state s' by its corresponding probability $\pi \cdot p$.

Summing over all possibilities yields the expected values.

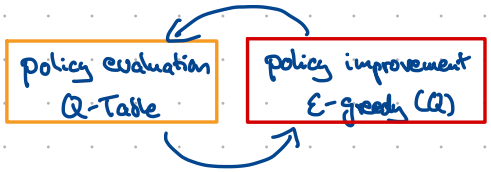
The action-value function yields the return if the agent starts in s, takes action a and then follows policy π:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$

An optimal policy π_* is a policy where $v_{\pi_*}(s) \geq v_\pi(s)$ for all $s \in S$.



Monte Carlo Methods



Prediction Problem: Given a policy, how might the agent estimate the value function for that policy?

→ With MC prediction methods, the goal is to collect state-action pairs for many episodes and track their reward.

Taking the mean reward of state-action pairs over all episodes yields the estimated Q-Table.

For multiple visits of state-action pairs within an episode, there is the first-visit-MC (untrained, better) or every-visit-MC.

→ pseudocode: Input: policy π, num_episodes, Output: value-function $Q(s,a) \approx q_\pi$ for large num_episodes

Generate an episode and calculate the cumulative reward for the first occurence of a state-action pair

$Q(s,a) \leftarrow$ returns_sum(s,a) / $N(s,a)$ for all $s \in S, a \in A$

Control Problem: Estimate the optimal policy.

→ With MC control methods, the policy is updated based on the generated Q-Table.

This is done with an ε-greedy update: $\pi' \leftarrow \varepsilon$-greedy$(Q)$ where ε sets the exploration-exploitation trade-off.

$$\pi(a|s) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|A(s)| & \text{if } a \text{ maximizes } Q(a,s) \\ \varepsilon/|A(s)| & \text{else} \end{cases}$$

($\varepsilon/|A(s)|$ is the equiprobable random policy including the optimal action)

An optimal solution is found with the "greedy in the Limit with Infinite Exploration" (GLIE) algorithm.

In order to update more frequently, one can implement incremental mean or constant alpha approach.

→ constant alpha example: $Q(s,a) \leftarrow Q(s,a) + \alpha(G_t - Q(s,a))$, where $\alpha \in [0,1]$

$G_t :=$ ("error" between estimated reward and actual reward)

Temporal Difference Methods

Review of Monte Carlo Methods:

- In order to learn, the agent has to terminate an episode.
- For most RL tasks, this is either slow (chess) or not desired (car → crash).
- Constant-alpha update equation: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha (G_t - Q(S_t, A_t))$

TD Control Methods update the value function after each time step.

- Sarsa is an on-policy TD method. It uses the current reward and value of the next state-action pair to estimate Q:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha (R_{t+1} + \gamma \cdot Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$$

- Sarsamax is an off-policy TD method. It uses the current reward and greedy value of the next state to estimate Q: (Q-Learning)

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha (R_{t+1} + \gamma \cdot \max_{a \in A} Q(S_{t+1}, a) - Q(S_t, A_t))$$

- Expected Sarsa is an on-policy TD method. It uses the current reward and probability distribution for the next action to estimate Q:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha (R_{t+1} + \gamma \cdot \sum_{a \in A} \pi(a|S_{t+1}) \cdot Q(S_{t+1}, a) - Q(S_t, A_t))$$

- All Sarsa methods are guaranteed to converge for small α and ϵ under the GLIE condition (in praxis, also w/o GLIE)
- In Praxis:
  - It shows that Optimism when initializing Q yields better results.
  - On-policy methods have a better online learning performance. Expected Sarsa often works best.

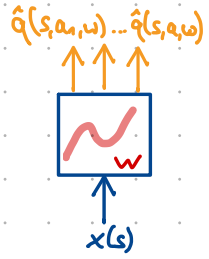
RL in Continuous Spaces

The previous (model-free) methods work with finite MDPs, where state-action pairs can be represented in a discrete table.

In continuous spaces, states and/or actions are infinite: $s \in \mathbb{R}^n$, $a \in \mathbb{R}^m$. There are two ways of applying RL methods:

- Discretization (only works for small underlying spaces → "curse of dimensionality")
  - → The infinite state space gets mapped to a discrete representation, e.g. by uniformly binning the space.
  - → Tile-Coding overlays multiple grids to the space, resulting in a bit-vector representation.
    The estimated value function is the average of the weights of the activated tiles.
  - → More advanced discretization methods are adaptive tile-coding or coarse coding with a radial-basis function.
- Function Approximation

  - → the state is represented as a feature vector $x(s) = \begin{pmatrix} x_1(s) \\ \vdots \\ x_n(s) \end{pmatrix}$.
  - → the mapping from s to x(s) can make use of kernel functions to derive non-linear features.
  - → the scalar value output is the dot product of feature vector and weights: $\hat{v}(s,w) = x(s)^T \cdot w$



  - → the weights can be learned using gradient descent: $J(w) = \mathbb{E}_\pi\left[(v_\pi(s) - x(s)^T \cdot w)^2\right]$ → $\Delta w = \alpha\,(v_\pi(s) - x(s)^T \cdot w) \cdot x(s)$
  - → for discrete action-values, the value function of each action $\hat{q}(s,a_i,w)$ can be approximated with a weight matrix.
  - → non-linear value functions can be approximated using non-linear activation functions:

    $\hat{v}(s,w) = f(x(s)^T \cdot w)$ → $\Delta w = \alpha\,(v_\pi(s) - \hat{v}_\pi(s,w)) \cdot \nabla_w \hat{v}(s,w)$, $\nabla_w \hat{v}(s,w)$ is the partial derivative!

  ⇒ This is the basis of using neural networks for RL!

Deep Q-Networks

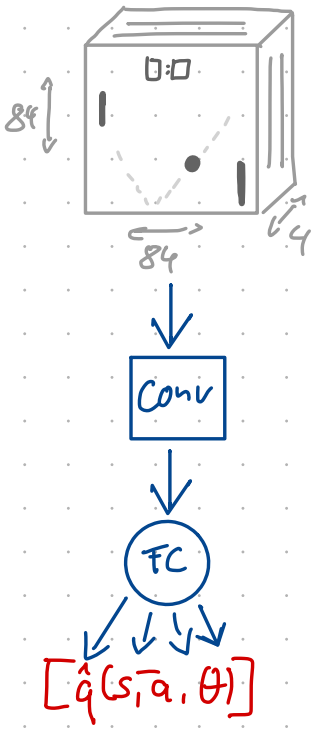
Review: It is unfeasible to discretize large state-spaces in a Q-table, i.e. the SLLE requirement cannot be fulfilled.

Therefore, (non-linear) function approximation is used to generalize the value-functions from sparse samples.

Neural networks are powerful non-linear function approximators, but two pitfalls in combination of RL need to be solved:

- there is a high correlation between state-spaces and actions that lead to oscillations and divergence in online learning
- the target needs to be decoupled from the action, because Q-learning "updates a guess with a guess".

The **Deep-Q-Network** algorithm avoids these pitfalls with experience replay and fixed Q-targets:



- In experience replay, the agent samples $<S_t, A_t, R_t, S_{t+1}>$ tuples and saves them in a replay buffer
  → randomly sampling from the experience buffer when learning tackles state-action-correlation and increases data efficiency, because the samples can be used multiple times for updates.
- With fixed Q-targets, the TD-target is calculated from a copy of the network and only updated after C learning steps.

$$\Delta w = \alpha \cdot \underbrace{\left(R + \gamma \cdot \max_a \hat{q}(S', a, w^-) - \hat{q}(S', a, w)\right)}_{\text{TD error}} \cdot \nabla_w \hat{q}(S, A, w), \quad w^- = w \text{ after } C \text{ steps}$$

TD target: $R + \gamma \cdot \max_a \hat{q}(S', a, w^-)$ — current estimation: $\hat{q}(S', a, w)$

- algorithm: initialize replay memory D to capacity N and target value function $\hat{Q}$ with $\theta^- = \theta$

  for episode = 1 to M

  → for $t = 1, T$: take x samples with ε-greedy policy and store them in replay buffer,

  sample in mini-batches and perform gradient descent on θ, set θ^- to θ after C steps

Improving DQNs

- **Double DQNs**: The TD target depends on the action that maximizes the $\hat{q}(s,a,w)$ function for state s'.

  This estimation tends to be too high (overestimation of Q-values), especially in the early stages of training

  It can be avoided by choosing $\max_a q(s',a,w)$ with network w, and evaluating $q(s',a,w^-)$ with the fixed target network w^-

  $$\text{TD target} = R + \gamma \cdot \hat{q}\left(S', \underbrace{\arg\max_a \hat{q}(S',a,w)}_{\text{select a}}, \underbrace{w^-}_{\text{evaluate } \hat{q}}\right)$$

- **Prioritized Experience Replay**: Some experiences are more important than others → sample experience based on TD error δ_t! (+ small min. priority e)

  The priority-random-trade-off is controlled with hyperparameter a.

  The update rule needs to be adapted, because the underlying distribution of the experience sampling causes a bias

  $$p_t = \delta_t + e \quad , \quad P(i) = \frac{p_i^a}{\sum_k p_k^a} \quad , \quad \Delta w = \alpha \cdot \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^b \cdot \delta_i \cdot \nabla_w \cdot \hat{q}(S, A_i, w)$$

- **Dueling Networks**: In many RL tasks, the choice of action is not significant in many states.

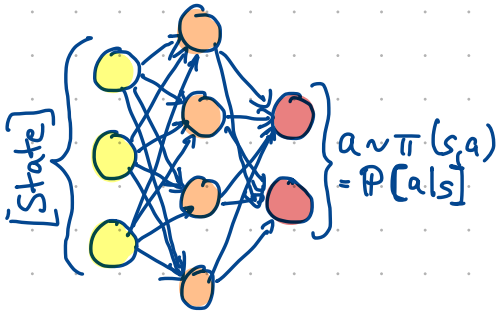
  Dueling networks adress this by using a network with "two streams" that estimate the state value $V^\pi(s)$ and the "advantage" $A^\pi(s,a)$ seperately, where $A(s,a) = Q(a,s) - V(s)$.

  For a stable gradient computation, the output $Q(s,a,\theta)$ of a dueling network is calculated as follows:

  $$Q(s,a,\theta,\alpha,\beta) = V(s,\theta,\beta) + \left(A(s,a,\theta,\alpha) - \frac{1}{|A|} \cdot \sum_{a'} A(s,a',\theta,\alpha)\right)$$

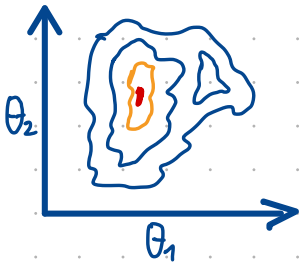
- **Rainbow DQN**: A "Rainbow" DQN combines the three improvements above as well as multi-step learning, distributional RL and Noisy Net in a single network. This lead to super-human performance.

Policy Based Methods



- In value-based methods, we approximate a value function $\hat{q}(s,a)$ to determine the best policy π^*.
- Instead, we can directly learn the **policy mapping $\pi: s \to a$** (deterministic) or the **action distribution $a \sim \pi(s,a)$** (stochastic).
- In discrete action spaces, the output is a probability for action, in continuous spaces it is a value eg. $[-1, 1]$.
- Benefits of policy-based methods are:
  - Simplicity: P.B.M. directly get to the problem at hand
  - Stochastic policies: P.B.M. can learn true stochastic policies (unlike ε-greedy) → good for "aliasing" states
  - Continuous action spaces: P.B.M. tend to work well here.

Hill climbing



- Hill climbing is an iterative algorithm that slightly perturbes the weights θ_{new} with gaussian noise and evaluates $\theta_{best} \leftarrow \theta_{new}$.
- It is a set of black box algorithms that require no back-propagation.
- Improvements of the vanilla algorithm are:
  - Steepest-Hill-Ascent: A small number of policies are evaluated and the best among them is chosen.
  - Simulated annealing: The search radius is reduced gradually
  - Adaptive noise scaling: Decrease the search radius when a new best policy is found, else increase

- These algorithms can also be used with **Evolutionary Strategies** where the next weights are determined by a weighted-distribution of the "previous" generation. This allows efficient, massive parallel computing and tends to work well with RL tasks with "sparse reward".

Policy Gradient Methods

- Policy Gradient Methods try to estimate the weights of an optimal policy through gradient ascent: $\theta \leftarrow \theta + \alpha \cdot \hat{g}$
- The ultimate goal is to make (state, action) pairs more likely when the reward is positive, and less likely otherwise
- In order to work with episodic and continuous tasks, we define a trajectory τ as: $(s_0, a_0, s_1, a_1, \dots, s_H, a_H, s_{H+1})$, H: "Horizon"
- The return of a trajectory is denoted as $R(\tau)$, the corresponding expected return $U(\theta)$ is: $U(\theta) := \sum_\tau P(\tau;\theta) \cdot R(\tau)$
- The algorithm we use to optimize the policy is called REINFORCE:

1) Use policy π_θ to collect m trajectories $\{\tau^{(1)}, \tau^{(2)}, \dots, \tau^{(m)}\}$ with horizon H. ("policy rollout")

2) Use the trajectories to estimate the gradient $\nabla_\theta U(\theta)$:

"likelihood ratio (policy) gradient trick":

$$\nabla_\theta \cdot U(\theta) \approx \hat{g} := \frac{1}{m} \cdot \sum_{i=1}^{m} \sum_{t=0}^{H} \nabla_\theta \cdot \log \pi_\theta(a_t^{(i)} | s_t^{(i)}) \cdot R(\tau^{(i)}) \qquad P(\tau;\theta) \cdot \frac{\nabla_\theta P(\tau;\theta)}{P(\tau;\theta)} = P \cdot \nabla_\theta \log P$$

3) Use the estimated gradient to update the weights of the policy:

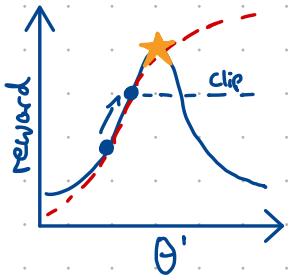
$$\theta \leftarrow \theta + \alpha \cdot \hat{g}$$

(For continuous action spaces, we can try to output a parameterized probability distribution and sample an action $a_t \sim \mathcal{N}(\mu, \sigma^2)$.

Modifications to make this work are described below.)

Proximal Policy Optimization

- PPO is a state-of-the-art RL algorithm for complex environments and continuous action spaces
- Drawbacks of the REINFORCE algorithm that motivates PPO are: 1) data inefficiency (trajectories are thrown away) 2) noisy gradients 3) no credit assignement
- Noise reduction is reached by sampling multiple trajectories and averaging their gradients and by applying "batch normalization" on the rewards.
- Credit assignement (+ also noise reduction) is done by changing the coefficient from total rewards $R(\tau)$ to future rewards $R_{(\tau)}^{future}$ $\rightarrow g = \sum R_t^{future} \cdot \nabla_\theta \cdot \log \pi(a_t|s_t)$
- In order to re-use old trajectories, we can compensate the change in the underlying distribution by importance sampling $\rightarrow \sum \underbrace{P(\tau;\theta)}_{\text{"old } \pi\text{"}} \cdot \underbrace{\frac{P(\tau;\theta')}{P(\tau;\theta)}}_{\text{re-weighting factor}} \cdot f(\tau)$



- PPO consists of two key principles:
  - The surrogate function: We use importance sampling to estimate the gradients of θ' based on trajectories that were sampled from an old policy $P(\tau;\theta)$. We further assume that the old and new policies are similar to each other (proximity). Hence we can define a surrogate function as optimization objective for gradient ascent:

$$g = \nabla_\theta \cdot L_{sur}(\theta', \theta), \text{ where } L_{sur} = \sum_t \frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \cdot R_t^{future}$$

  - Clipping policy updates: The gradient estimation approximation is only valid if the policies are similar enough. To avoid over-shooting (and possibly landing in a very bad policy), we clip the surrogate objective if the ratio of the policies exceed $1 \pm \varepsilon$.

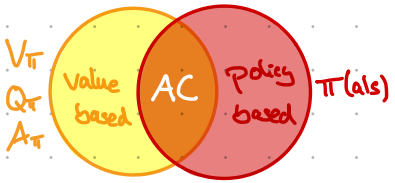
$$L_{sur}^{clip} = \sum_t \min\left\{\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \cdot R_t^{future}, \text{clip}\left(\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}\right) \cdot R_t^{future}\right\}$$

⇒ PPO algorithm: First collect trajectories based on θ and set $\theta' = \theta$.

↺ repeat: Compute the gradient $\nabla_{\theta'} \cdot L_{sur}^{clip}(\theta', \theta)$ and update $\theta' \leftarrow \theta' + \alpha \cdot \nabla_{\theta'} \cdot L_{sur}^{clip}(\theta', \theta)$.

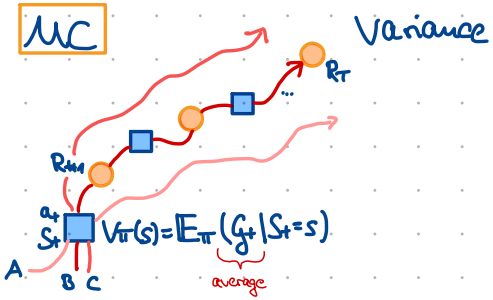
After a few iterations, set $\theta' = \theta$ and repeat from the beginning ("policy rollout").

Actor-Critic-Methods

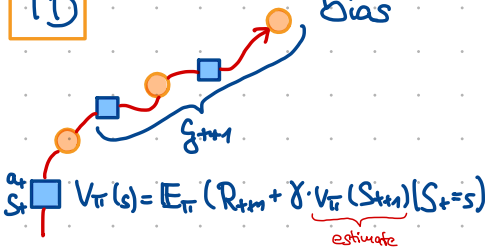


- AC-methods try to use value-based techniques to reduce the variance of policy-based methods.
- In policy-based methods, the agent learns to "act", but actions are reinforced based on the outcome (win/loss).
  → good actions are discouraged when the total reward is negative, so many trajectories are needed! (MC)
- In value-based methods, the agent learns to "estimate" a state-value, e.g. via TD-methods.
  → combining both approaches will eventually speed up learning and reduce the variance!
- Bias / Variance in RL algorithms come from the approaches to estimate expected total reward:
  - Monte-Carlo-Methods average the reward of multiple trajectories transitioning through state S ⇒ variance
  - Temporal-Difference-Methods use an estimate of the next state-value to estimate $v_\pi(s_t)$ ⇒ bias

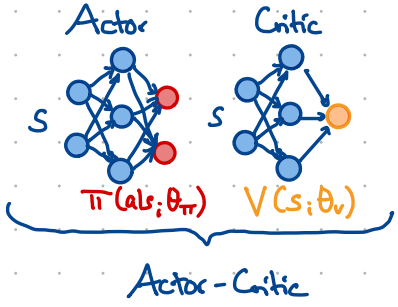
MC — Variance



TD — bias



- A basic Actor-Critic Algorithm works as follows:
  - Use the actor for interaction with the environment to get a (s, a, r, s')-tuple.
  - Use the TD-estimate of that experience tuple $r + \gamma \cdot V(s'; \theta_v)$ to train the critic.
  - Use the critic to calculate the advantage $A(s,a) = r + \gamma \cdot V(s'; \theta_v) - V(s; \theta_v)$ to train the actor ("baseline").



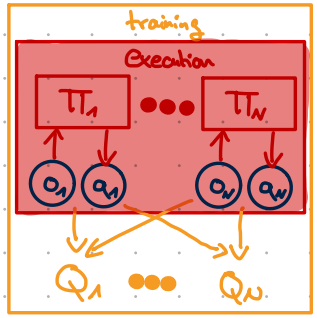
- Using n-step bootstrapping is a trade-off between bias ($n=1$) and variance ($n=\infty$).
  → In general advantage estimation (GAE) all n-step bootstraps are combined: $\sum_{t=1}^{T} G_t \cdot (1-\lambda) \cdot \lambda^t$
- AC-methods are on-policy (=stable) and replace the replay buffer with parallel training.
  → A3C: asynchronous advantage actor-critic; A2C: (synchronous) advantage actor-critic

$$\theta_{t+1} = \theta_t + \alpha \cdot (G - \underbrace{\hat{v}(s)}_{\text{baseline}}) \cdot \nabla \log \pi(a|s)$$

Multi-Agent RL

- Multi-Agent scenarios are very common in RL applications and can either be collaborative, competitive or mixed.
- Learning is very unstable with vanilla RL-methods, because for each individual agent, the environment is not constant because of the other agents policies

→ This problem is solved in **MADDPG** that is using a "shared" critic with a centralized value function that considers all agents policies.



- A Markov Game is a multi-agent extension of MDPs with partially observed states.

  $A = A_1 \dots A_N$, $O = O_1 \dots O_N$, $\pi_{\theta_i} : O_i \times A_i \mapsto [0,1]$, $T : S \times A_1 \times \dots \times A_N \mapsto S$ (transition depends on S + all actions)

- The gradient of a single agent in MADDPG is calculated as follows using the deterministic policy gradient approach:

$$\nabla_{\theta_i} J(\mu_i) = \mathbb{E}_{x,a \sim D}\left[\nabla_{\theta_i} \mu_i(a_i|o_i) \nabla_{a_i} Q_i^{\mu}(x, a_1, \dots, a_N) \big|_{a_i = \mu_i(o_i)} \right]$$

  where: ○ $J(\mu_i)$ is the objective function, i.e. the total expected reward of agent i following policy μ_i,

  ○ $\mu(a_i|o_i)$ is the deterministic policy of agent i, given observation o_i,

  ○ $Q_i^{\mu}(x, a_1, \dots, a_N)$ is the centralized value function considering all observations x and all agents actions a.

- Instead of "knowing" all agents policies, they can also be learned with an entropy regularizer.
- In competitive scenarios, agents tend to overfit to specific strategies. To avoid such behavior, it is more robust to learn K ensemble sub-policies and then choosing uniformly random from the ensemble.