deepRL WS 20/21 - Summary

Janosch Bajorath

Feb 25, 2021

1 Intro to RL and MDPs

Machine Learning paradigm like supervised learning and unsupervised learning, which is divided into a set of techniques that can be used to approach a set of different RL problems.

1. no supervision, only reward signal - learn from its interaction with the environment (closed loop system)
2. learns patterns from exploration (trial and error search)
3. discrete time-steps with delayed feedback/reward (not instantaneous)
4. current actions influence subsequent data/states the agent receives
5. soft targets with indirect instructions
6. Goal: select actions to maximise total future reward (immediate and future reward important)

Difference between learning and planing
learning:

- environment initially unknown
- agent interacts with environment
- agent improves policy (direct or indirect via value functions)

planing:

- model of the environment is known
- agent performs computations with the model (no interaction with the environment)
- agent improves policy

Difference between Model based and Model free
Model based:

- prior knowledge about the model (i.e. state-transition dynamics and reward dynamics) or learning of an approximation
- learn the model -¿ behave optimal accordingly

Model free:

- no explicit knowledge about the model and no modeling, only implicit knowledge about the model dynamics
- Policy based algorithms directly optimize a policy, while value based approaches learn a value for states or actions

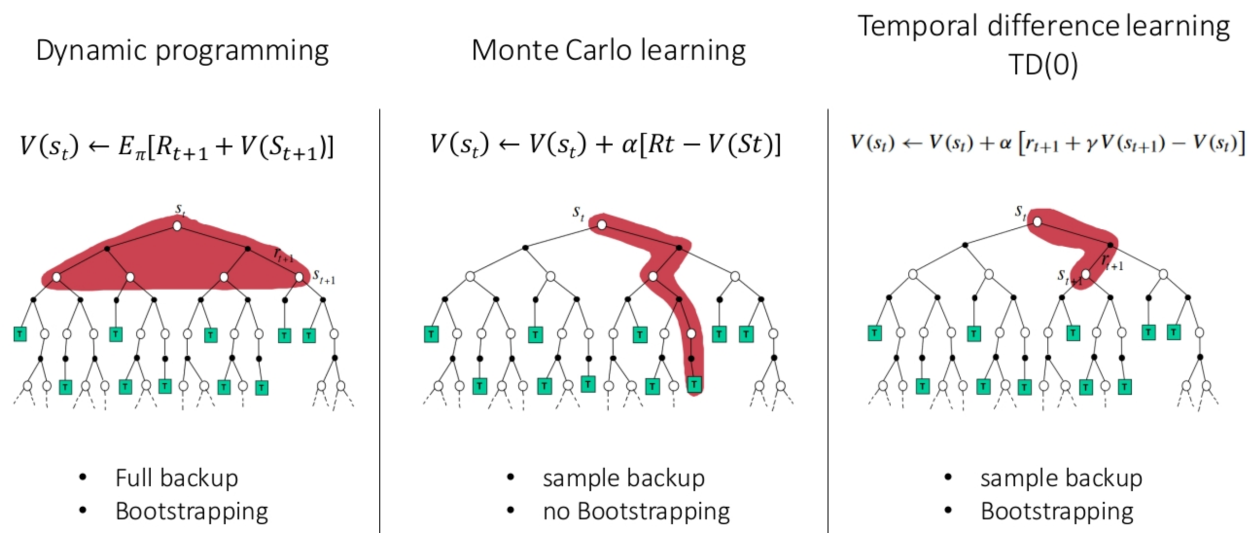

Dynamic programming
$V(s_t) \leftarrow E_\pi[R_{t+1} + V(S_{t+1})]$
s_t
r_{t+1}
s_{t+1}
T
Full backup
Bootstrapping
Monte Carlo learning
$V(s_t) \leftarrow V(s_t) + \alpha[Rt - V(St)]$
sample backup
no Bootstrapping
Temporal difference learning TD(0)
$V(s_t) \leftarrow V(s_t) + \alpha\left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right]$
sample backup
Bootstrapping

Figure 1: Backup Diagrams for common RL algorithms.

1.1 Markov Decision Process - MDP

The **Markov Property** states that: "The future is independent of the past given the present" - Independence of path property.

- state signal that retains all relevant information from the history, i.e. the state is a sufficient statistic of the future, is said to be Markov

A **Markov process/chain** is a memory less random process, i.e. a sequence of random states S1, S2, ... with the Markov property.

- tuple of $(s, p(s'|s))$ - state and state transition probability

A **Markov reward process** is a Markov chain with values.

- tuple of $(s, p(s'|s), \mathbb{E}(r'|s), \gamma)$ - additionally reward function $(R = \mathbb{E}(r'|s))$ and discount factor
- can be solved by the Bellman equation

A **Markov decision processes** formally describes a fully observable environment for reinforcement learning and is defined by:

- tuple of $(s, a, p(s'|s, a), \mathbb{E}(r|s, a), \gamma)$
- a set of States S - finite or infinite
- a set of Actions A - finite or infinite
- environment dynamics (Function that maps the behavior of the environment; also referred to as $p(s', r|s, a)$):
    1. state transition probability $p(s'|s, a)$
    2. probabilistic reward dynamics $p(r|s, a)$ (often expected reward instead of probabilistic; a reward Rt is a scalar feedback signal)
- can be solved by the Bellman expectation eqaution

1.1.1 Return

Reward Hypothesis - All goals can be described by the maximisation of expected cumulative reward (scalar feedback signal)

The return G_t is the total discounted reward from time-step t until the Episode terminates (T-timesteps):

$$G_t = \sum_{i=0}^{T} \gamma^i r_{t+i+1} \tag{1}$$

- **the reward for time-step t is received at time-step $t+1$**
- γ close to 0 leads to "myopic" evaluation and close to 1 leads to 'far-sighted' evaluation
- **bounds** of the Return, due to it's formula being a **geometric series** (only if the reward $r(s,a)$ is bounded):
$$G_{max/min} = r_{max/min} \frac{1}{1-\gamma} \tag{2}$$
- **recursive definition** of the Return:

$$\begin{aligned} G_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... \\ &= r_{t+1} + \gamma(r_{t+2} + \gamma r_{t+3} + ...) \\ &= r_{t+1} + \gamma G_{t+1} \end{aligned} \tag{3}$$

$$G_t^s = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)r + \gamma G_{t'}^{s'}) \tag{4}$$

Why **discount** the reward by a factor γ?

- Avoids infinite returns in cyclic Markov processes
- immediate rewards may earn more interest than delayed rewards
- uncertainty about the future

1.1.2 Value Functions

- The optimal state/action-value function $v_*(s)/q_*(s,a)$ is the maximum state/action-value function over all policies, accordingly $\pi = \pi_*$
- an optimal value function specifies the best possible performance in the MDP, thus is the solution

1. **State-Value** Function
    - The state value $v_\pi(s)$ describes the expected Return, i.e. the expected (discounted) accumulated reward, of being in the state s and following the policy π

$$v_\pi(s_t) = \mathbb{E}_\pi \left[\sum_{i=0}^{\infty} \gamma^i r_{t+i+1} \middle| s_t = s \right] = \mathbb{E}_\pi[G_t | s_t = s] \tag{5}$$

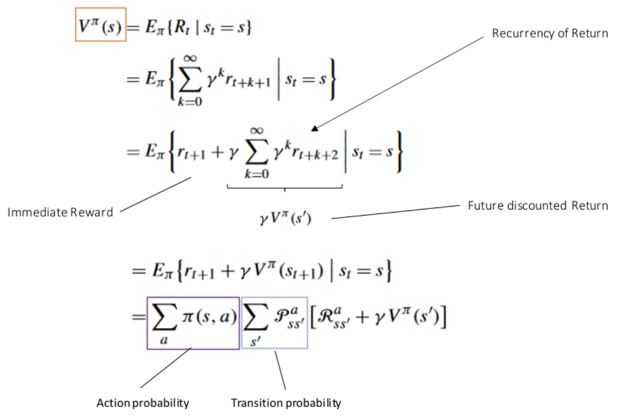
2. (State-)**Action-Value** Function
    - The action-value function $q_\pi(s,a)$ is the expected return starting from state s, taking action a, and then following policy π

$$q_\pi(s_t, a_t) = \mathbb{E}_\pi \left[\sum_{i=0} \gamma^i r_{t+i+1} \middle| s_t = s, a_t = a \right] = \mathbb{E}_\pi[G_t | s_t = s, a_t = a] \tag{6}$$

1.1.3 Bellman Expectation equation

1. **State-Value** Function:
   The state-value function can be decomposed into two parts (recursiveness of Return):

   - immediate reward r_{t+1}
   - value of successor state $\gamma v(s_{t+1})$



$$\begin{aligned}
\boxed{V^\pi(s)} &= E_\pi\{R_t \mid s_t = s\} \\
&= E_\pi\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1} \,\middle|\, s_t = s\right\} \\
&= E_\pi\left\{r_{t+1} + \gamma \underbrace{\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}}_{\gamma V^\pi(s')} \,\middle|\, s_t = s\right\} \\
&= E_\pi\{r_{t+1} + \gamma V^\pi(s_{t+1}) \mid s_t = s\} \\
&= \boxed{\sum_a \pi(s,a)}\boxed{\sum_{s'} \mathcal{P}^a_{ss'}}\left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')\right]
\end{aligned}$$

Figure 2: Bellmann expectation equation for state-values.

$$\begin{aligned}
v_\pi(s_t) &= \mathbb{E}_\pi[G_t | s_t = s] \\
&= \mathbb{E}_\pi\left[r_{t+1} + \gamma v_\pi(s_{t+1}) \middle| s_t = s\right]
\end{aligned} \tag{7}$$

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)(r(s, a) + \gamma v(s')) \tag{8}$$

   Equation 8

   - the probability of the first reward is based on the policy (probability of that action), thus the first reward is also probabilistic (that's why first reward is inside)

2. (State-)**Action-Value** Function The action-value function can, similarly to the state-value function, be decomposed into two parts (recursiveness of Return):

   - immediate reward r_{t+1}
   - action-value of successor state $\gamma q(s_{t+1})$

$$\begin{aligned}
q_\pi(s_t, a_t) &= \mathbb{E}_\pi[G_t | s_t = s, a_t = a] \\
&= \sum_{s',r} p(s', r|s, a)\left(r_{t+1} + \gamma \sum_{a'} \pi(a'|s') q_\pi(s_{t+1}, a_{t+1})\right) \\
&= \sum_{s',r} p(s', r|s, a)\left(r_{t+1} + \gamma v_\pi(s_{t+1})\right)
\end{aligned} \tag{9}$$

$$q_\pi(s, a) = \sum_{s',r} p(s', r|s, a)\left(r(s, a) + \gamma \sum_{a'} \pi(a'|s') q(s', a')\right) \tag{10}$$

   Equation 10

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi\left[G_t|S_t=s, A_t=a\right] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t=s, A_t=a\right] \\ &= \mathbb{E}_\pi\left[R_{t+1}+\sum_{k=0}^{\infty}\gamma^k R_{t+k+2}|S_t=s, A_t=a\right] \\ &= \sum_{s',r} p(s',r|s,a)\left[r+\mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+2}|S_{t+1}=s'\right]\right] \\ &= \sum_{s',r} p(s',r|s,a)\left[r+\gamma V_\pi(s')\right] \end{aligned}$$

Figure 3: Bellmann expectation equation for state-values.

- the first reward is not dependent on the policy only on the environment dynamics (no action to be chosen; value for different actions), thus the first action is implicit for the action-value (already defined) and only successor rewards are influence by taking a specific action (that's why first reward without $\pi(a|s)$)
- makes knowledge about model dynamics unnecessary as entailed within the value-function

1.1.4 Bellman Optimality equation

Defines partial ordering over the value space, which implies and ordering over the policies derived from them. An optimal value function specifies the best possible performance in the MDP, thus is the solution.

$$v_*(s) = \max_\pi v_\pi(s) \quad \text{and} \quad q_*(s,a) = \max_\pi q_\pi(s,a) \tag{11}$$

1. **State-Value** Function:

$$v_*(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)(r(s,a) + \gamma v(s')) \tag{12}$$

   Equation 8

   - the probability of the first reward is based on the policy (probability of that action), thus the first reward is also probabilistic (that's why first reward is inside)

2. (State-)**Action-Value** Function The action-value function can, similarly to the state-value function, be decomposed into two parts (recursiveness of Return):

   - immediate reward r_{t+1}
   - action-value of successor state $\gamma q(s_{t+1})$

$$q_\pi(s_t,a_t) = \mathbb{E}_\pi\left[r_{t+1} + \gamma q_\pi(s_{t+1},a_{t+1})\middle| s_t = s, a_t = a\right] = \mathbb{E}_\pi[G_t|s_t = s, a_t = a] \tag{13}$$

$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)\left(r(s,a) + \gamma \sum_{a'} \pi(a'|s')q(s',a')\right) \tag{14}$$

   Equation 9

   - the first reward is not dependent on the policy only on the environment dynamics (no action to be chosen; value for different actions), thus the first action is implicit for the action-value (already defined) and only successor rewards are influence by taking a specific action (that's why first reward without $\pi(a|s)$)
   - makes knowledge about model dynamics unnecessary as entailed within the value-function

1.1.5 Policy

A policy π is a probabilistic distribution over actions given states, it fully defines the behaviour of an agent and depends only on the current state (i.e. are stationary/time independent):

$$\pi(a|s) = \mathbb{P}(a|s), \qquad \text{with the } \textbf{optimal} \text{ policy } \pi_*(s|a) \tag{15}$$

Given an MDP and a policy π, the state sequence $S_1, S_2...$ is defined as a Markov Process and the state reward sequence $S_1, R_1, S_2...$ is defined as a Markov Reward Process (both dependent on the policy π).

Principle of Optimality - A policy $\pi(a|s)$ achieves the optimal value from state s, $v_\pi(s) = v_*(s)$, if and only if the following applies

- $v_\pi(s)$ is the optimal value $v_*(s)$ for any successor state s' reachable from s
- π achieves the optimal value from state s'

1.2 Dynamic Programming - solve a known MDP

Dynamic - sequential or temporal component within the problem
Programming - optimising a "program", i.e. a policy

A method for solving complex problems, by breaking them down into sub-problems (solve sub-problems, then combine solutions of sub-problems to solve higher order problems). A very general solution method for problems which have two properties:

- optimal sub-structure (principle of optimallity applies)
- overlapping sub-problems (sub-problems recur)

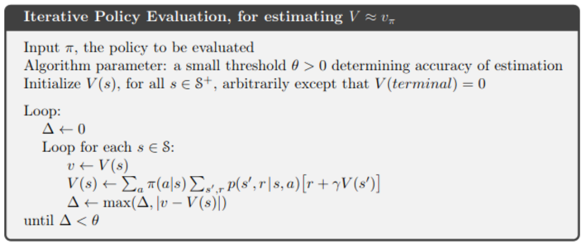
Markov decision processes satisfy both properties with the optimal Bellman equation, which gives an optimal solution to a problem, as well as a recursive decomposition.

Key facts:

- requires full knowledge of the MDP (environment dynamics)
- planning for MDPs
- **policy-evaluation** (prediction) estimates the current *value-function* based on a provided *policy-function*
- **policy-improvement** (control) improves the current *policy-function* (possibility of no improvement, but never worse $\pi \geq \pi'$) based on a provided *value-function*
- full-width backups (Every successor state and action is considered, using knowledge of the MDP transitions and reward function)
- bootstrapping
- not applicable for large state spaces - course of dimensionality (number of states grows exponentialy with every state-dimension added)

1.2.1 Iterative policy evaluation

- Problem: evaluate a given policy π
- Solution: iterative application of the Bellman Expectation equation
- Bellman Expectation backup is a Contraction (Contraction Mapping Theorem - converges to a unique fixed point with a linear convergence rate)
- synchronous backups



Iterative Policy Evaluation, for estimating $V \approx v_\pi$

```
Input $\pi$, the policy to be evaluated
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
      $v \leftarrow V(s)$
      $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$
      $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$
```

Figure 4: The Policy Evaluation algorithm.

1.2.2 Policy improvement

Improve the policy by acting e.g. greedily with respect to v_π/q_π. After improvement of the policy π to π', one can not use the old value estimates v_π/q_π anymore, as $v_\pi \neq v_{\pi'}$ (same for q_π).

1. Greedy policy improvement over $V(s)$ requires model of MDP
2. Greedy policy improvement over $Q(s,a)$ is model-free
3. next $V(s)$ is dependent on the transition probability, whereas $Q(s,a)$ isn't

1. State-Value function

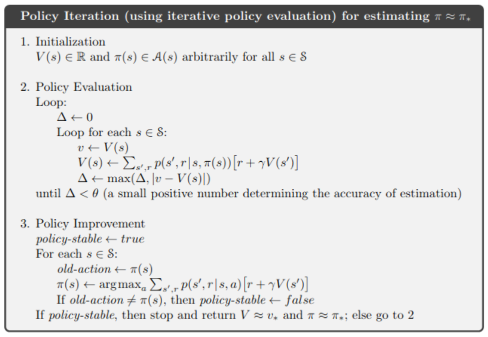
$$\begin{aligned} &\text{deterministic env dynamics:} \quad \pi'(s,a) = argmax_a[r(s,a) + \gamma V(s')] \\ &\text{non-deterministic env dynamics:} \quad \pi'(s,a) = argmax_a[r(s,a) + p(s'|s,a)\gamma V(s')] \end{aligned} \tag{16}$$

2. Action-Value function

$$\pi'(s,a) = argmax_a Q(s,a) \tag{17}$$

1.2.3 Policy iteration

Combination of Policy Evaluation and Policy Improvement, which always converges to π_*



Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

```
1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
       $\Delta \leftarrow 0$
       Loop for each $s \in \mathcal{S}$:
           $v \leftarrow V(s)$
           $V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s))[r + \gamma V(s')]$
           $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   $policy\text{-}stable \leftarrow true$
   For each $s \in \mathcal{S}$:
       $old\text{-}action \leftarrow \pi(s)$
       $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$
       If $old\text{-}action \neq \pi(s)$, then $policy\text{-}stable \leftarrow false$
   If $policy\text{-}stable$, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2
```

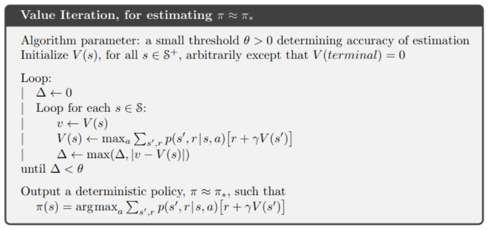
Figure 5: The Policy Iteration algorithm.

1.2.4 Value iteration

Does policy evaluation need to converge to v_π? Value iteration introduce a stopping condition, to stop the policy evaluation after one iteration (policy iteration where the policy is updated every iteration).

- one-step lookahead search

- applies Bellman optimality equation using synchronous backups
- no explicit policy
- Intermediate value functions may not correspond to any policy



```
Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
|   $\Delta \leftarrow 0$
|   Loop for each $s \in \mathcal{S}$:
|       $v \leftarrow V(s)$
|       $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
|       $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
    $\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
```

Figure 6: The Value Iteration algorithm.

1.3 Monte Carlo Methods - solve an unknown MDP by value-function estimation

- learns from complete episodes: no bootstrapping (estimated values are independent of each other; adjust expectation based on the actual Return)
- downside: only applicable for episodic MDPs, episodes must terminate
- does not exploit the Markov property (efficient in non-Markov environments)
- Every-visit Backup vs. First-visit Backup (update of the action/state-value)
- simple Idea: value = mean Return
- high variance, zero bias (Return G_t is unbiased estimate of $v_\pi(s_t)$, but has high variance)
  - Return depends on many random actions, transitions, rewards
  - good convergence properties
  - not sensitive to initial value
- MC-Estimate: Monte-Carlo policy evaluation uses the **empirical mean Return** for updates
- the expected mean value of the sample Returns is actually the expected value of the returns (convergence with growing amount of samples over time)

  State-Value function estimator:

$$V(s) = \frac{1}{k} \sum_k G_\pi^k(s) \tag{18}$$

  Action-Value function estimator:

$$Q(s, a) = \frac{1}{k} [r(s, a) + \gamma G_\pi^k(s)] \tag{19}$$

- **converges** to solution with minimum mean-squared error (best fit of the observed Returns)

$$\sum_{k=0}^{K} \sum_{t=1}^{T_k} = (G_t^k - v(s_t^k))^2 \tag{20}$$

Sample-backups in contrast to full-width backups:

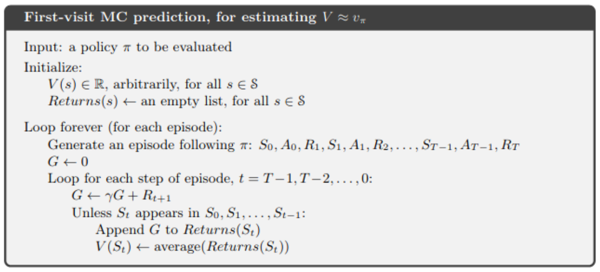
- Model-free: no advance knowledge of MDP required
- uses sample rewards and transitions for backups instead of using known environment dynamics (reward function and transition dynamics)
- curse of dimensionality diminished through sampling (Cost of backup is constant, independent of state space dim)

1.3.1 First visit on-Policy MC-policy evaluation (prediction) - estimating $v = v_\pi$

- estimates state-values for a given policy

Trajectory with 3 time-steps (t=[0, 1, 2]):

- return for the final time-step $t = 2$ is: $G_2 = r_2$
- return for the time-step $t = 1$ is: $G_1 = r_1 + \gamma r_2 = r_1 + \gamma G_2$
- return for the time-step $t = 0$ is: $G_0 = r_0 + \gamma r_1 + \gamma^2 r_2 = r_0 + \gamma[r_1 + \gamma r_2] = r_0 + \gamma G_1$

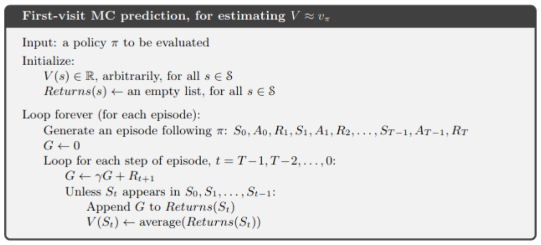


```
First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy $\pi$ to be evaluated
Initialize:
    $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
    $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):
    Generate an episode following $\pi$: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        Unless $S_t$ appears in $S_0, S_1, \ldots, S_{t-1}$:
            Append $G$ to $Returns(S_t)$
            $V(S_t) \leftarrow \text{average}(Returns(S_t))$
```

Figure 7: The Policy Evaluation algorithm.

1.3.2 First visit MC-Control - estimating $q = q_\pi$ with ϵ soft

- does MC-policy evaluation (prediction) for action-values and uses epsilon soft-policy (control) for sampling from the environment
- sampling and optimizing one policy: exploration given by the epsilon-soft policy, **exploring starts** can also be useful
- one step of MC-prediction followed by one step of policy improvement (e.g. epsilon-greedy)

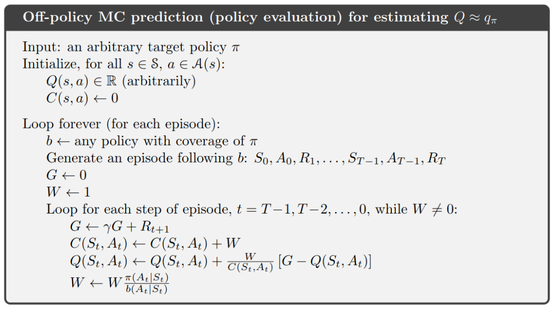


```
First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy $\pi$ to be evaluated
Initialize:
    $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
    $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):
    Generate an episode following $\pi$: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        Unless $S_t$ appears in $S_0, S_1, \ldots, S_{t-1}$:
            Append $G$ to $Returns(S_t)$
            $V(S_t) \leftarrow$ average($Returns(S_t)$)
```

Figure 8: The Policy Evaluation algorithm.

1.3.3 Off-policy MC algorithms

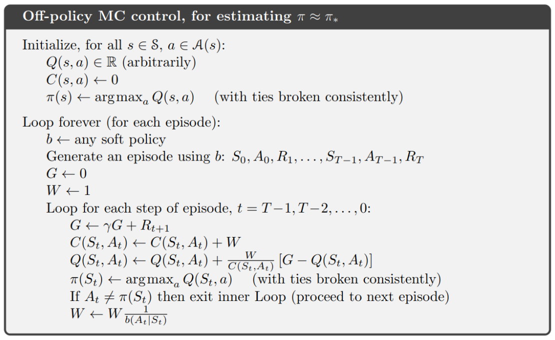
1. Off-policy MC prediction



```
Off-policy MC prediction (policy evaluation) for estimating $Q \approx q_\pi$

Input: an arbitrary target policy $\pi$
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
    $Q(s,a) \in \mathbb{R}$ (arbitrarily)
    $C(s,a) \leftarrow 0$

Loop forever (for each episode):
    $b \leftarrow$ any policy with coverage of $\pi$
    Generate an episode following $b$: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    $W \leftarrow 1$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$, while $W \neq 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
        $W \leftarrow W \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$
```

Figure 9: The off-Policy MC prediction algorithm.

2. Off-policy MC control



```
Off-policy MC control, for estimating $\pi \approx \pi_*$

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
    $Q(s,a) \in \mathbb{R}$ (arbitrarily)
    $C(s,a) \leftarrow 0$
    $\pi(s) \leftarrow \operatorname{argmax}_a Q(s,a)$    (with ties broken consistently)

Loop forever (for each episode):
    $b \leftarrow$ any soft policy
    Generate an episode using $b$: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    $W \leftarrow 1$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
        $\pi(S_t) \leftarrow \operatorname{argmax}_a Q(S_t, a)$    (with ties broken consistently)
        If $A_t \neq \pi(S_t)$ then exit inner Loop (proceed to next episode)
        $W \leftarrow W \frac{1}{b(A_t|S_t)}$
```

Figure 10: The off-Policy MC control algorithm.

1.3.4 Incremental MC methods

- Update $V(s)$ incrementally after each episode by:

$$\begin{aligned} N(s_t) &\leftarrow N(s_t) + 1 \\ V(s_t) &\leftarrow V(s_t) + \frac{1}{N(s_t)}(G_t - V(s_t)) \end{aligned} \tag{21}$$

- in non-stationary problems (slow changes in the environment dynamics), it can be useful to track a exponentially weighted moving average (running mean, i.e. to forget old episodes)

$$V(s_t) \leftarrow V(s) + \alpha[G(s) - V(s)] = (1-\alpha)V(s) + \alpha G(s) \tag{22}$$

  with the part inside the []-brackets as error (δ); $\delta_{MC} = G(s) - V(s)$ (very useful when training ANNs, as target is necessary)

1.4 TD-learning

- TD methods learn directly from episodes of experience
- model-free: no knowledge of MDP transitions / rewards
- TD-policy evaluation uses TD-target (immediate reward plus expected Return) for updates
- exploits the Markov property (good for Markov environments)
- does bootstrap (update estimated based on other learned estimates)
- can learn learn online after every step
- due to bootstrapping (updates a guess towards a guess), TD-algorithms can learn from incomplete sequences and from continuous environments that do not terminate
- low variance, some bias (TD-target is a biased estimate of $v_\pi(s_t)$, but a much lower variance one)
  - TD target depends on one (random/policy based action, transition and reward)
  - TD(0) converges to $v_\pi(s)$
  - sensitive to initial values
- updates values towards estimated Return $r_{t+1} + \gamma V(s_{t+1})$
- TD-target

$$r_{t+1} + \gamma V(s_{t+1}) \tag{23}$$

- TD-Error

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{24}$$

1.4.1 TD(n) - n-Step prediction

- n-Step TD learning:

$$Q(S_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[G_t^{(n)} - Q(s_t, a_t)] \tag{25}$$

- n-Step Target (the n-step return):

$$G_t^{(n)} = \left(\sum_{t=0}^{n-1} \gamma^t r_t \right) + \gamma^n Q(s_n, a_n) \tag{26}$$

- n-Step error:

$$\delta TD_{t,n} = \left(\sum_{t}^{n-1} \gamma^t r_t \right) + \gamma^n Q(s_n, a_n) - Q(s_t, a_t) \tag{27}$$

- Increasing n in n-step SARSA shifts the variance-bias trade-off towards less bias and more variance
- Increasing n in n-step SARSA can help to speed up the learning of long-term reward dynamics
- Effectively n-step SARSA interpolates between MC and SARSA(1)

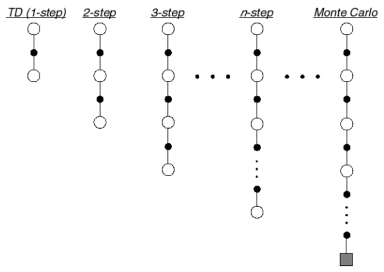


Figure 11: TD(n) illustration

1.4.2 TD(λ) - averaging all n-Step Returns

- updates value function towards the λ-discounted return
- episodic environments with offline updates
- vanilla version of TD(λ) is like MC, can only be computed from complete episodes, but n-step TD(λ) circumvents this (truncating TD(λ) to n-Steps)
- TD(λ) learning:

$$Q(s_t, a_t)) \leftarrow Q(s_t, a_t)) + \alpha[G_t^\lambda - Q(s_t, a_t))] \tag{28}$$

- TD(λ) Target:

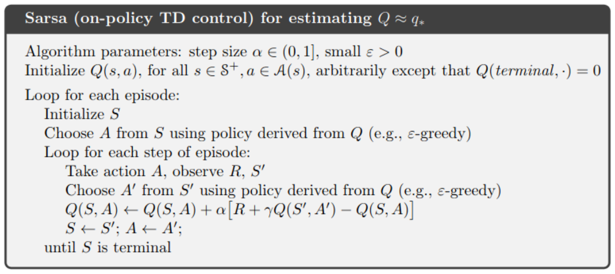
$$\begin{aligned} G_t^\lambda &= (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)} \\ &= [(1-\lambda) \sum_{n}^{T-1} \lambda^{n-1} G_t^{(n)}] + [\lambda^T Q(s_t, a_t)] \end{aligned} \tag{29}$$

- TD(λ) error:

$$\delta TD_{t,\lambda} = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)} - Q(s_t, a_t) \tag{30}$$

1.4.3 SARSA (state, action, reward, state, action) - on-policy TD-control

- one step SARSA Return defined as $r(s_t, a_t) + \gamma V(s')$ or $r(s_t, a_t) + \gamma Q(s', a')$
- with the respective error $\delta_{TD(1)} = r(s, a) + \gamma V(s') - V(s)$ or $\delta_{TD(1)} = r(s, a) + \gamma Q(s') - Q(s)$
- compared to MC-sample trajectory $r_{t+1}...r_T$, SARSA introduces a bias-variance trade-off through exchange of the sample trajectory with an value Estimate $V(s')$
- importance sampling is used for off-policy SARSA approaches



```
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 $Q(terminal, \cdot) = 0$

Loop for each episode:
   Initialize $S$
   Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
   Loop for each step of episode:
      Take action $A$, observe $R$, $S'$
      Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
      $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
      $S \leftarrow S'$; $A \leftarrow A'$;
   until $S$ is terminal
```

Figure 12: SARSA - The on-policy TD-control algorithm.

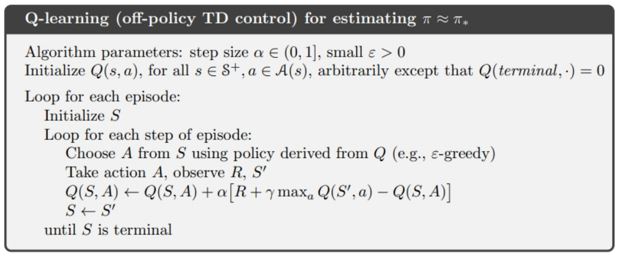
1.4.4 Q-Learning - off-policy learning of action-values $Q(s,a)$

- no importance sampling required
- next action is chosen using behaviour policy $a_t \sim \mu(\cdot|s_t)$
- consider alternative successor action $a_{t+1} \sim \mu(\cdot|s_t)$
- Q-updates are not estimating the current policy $Q_\pi(s,a)$, bur rather try to directly estimate $Q_{(s,a)}$
- allows off-policy updates without importance sampling, and updating greedily can be very efficient
- Q-learning:

$$Q(s_t,a_t)) \leftarrow Q(s_t,a_t) + \alpha \left[r_{t+1} \gamma \max_{a_{t+1}} Q(s_{t+1},a_{t+1}) - Q(s_t,a_t) \right] \tag{31}$$

- Q-learning error:

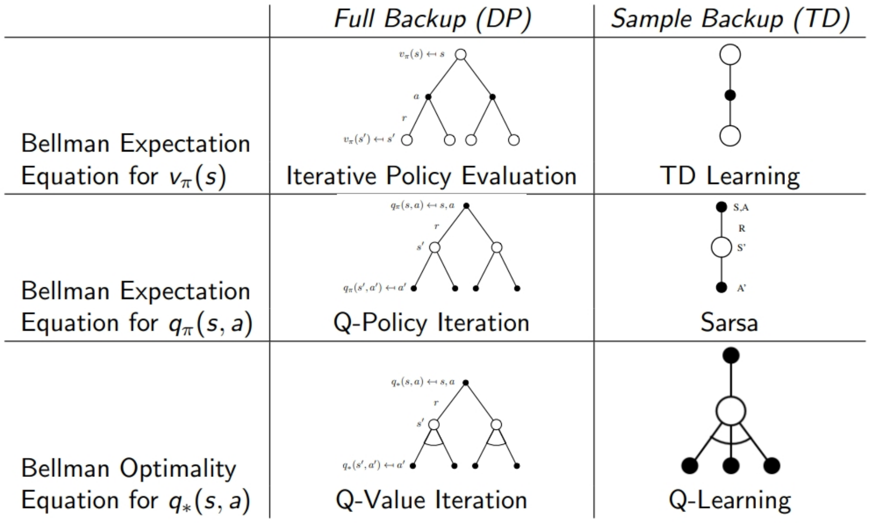
$$\delta_{TD-Watkins-Q} = r(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a) \tag{32}$$



```
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 $Q(terminal, \cdot) = 0$

Loop for each episode:
   Initialize $S$
   Loop for each step of episode:
      Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
      Take action $A$, observe $R$, $S'$
      $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
      $S \leftarrow S'$
   until $S$ is terminal
```

Figure 13: Q-learning - off-policy action value estimation.

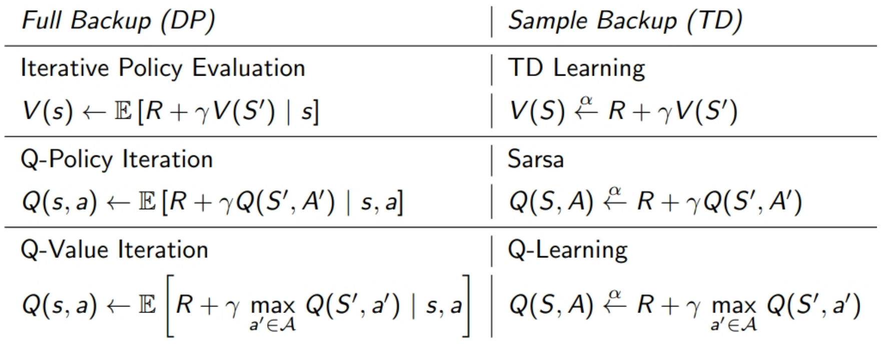
1.5 Difference between DP and TD



	Full Backup (DP)	*Sample Backup (TD)*
Bellman Expectation Equation for $v_\pi(s)$	Iterative Policy Evaluation	TD Learning
Bellman Expectation Equation for $q_\pi(s,a)$	Q-Policy Iteration	Sarsa
Bellman Optimality Equation for $q_*(s,a)$	Q-Value Iteration	Q-Learning

$v_\pi(s) \leftrightarrow s$
a
r
$v_\pi(s') \leftrightarrow s'$
$q_\pi(s,a) \leftrightarrow s,a$
r
s'
$q_\pi(s',a') \leftrightarrow a'$
S,A
R
S'
A'
$q_*(s,a) \leftrightarrow s,a$
r
s'
$q_*(s',a') \leftrightarrow a'$

Figure 14: Relationship between TD and DP (1).



Full Backup (DP)	*Sample Backup (TD)*
Iterative Policy Evaluation	TD Learning
$V(s) \leftarrow \mathbb{E}[R + \gamma V(S') \mid s]$	$V(S) \overset{\alpha}{\leftarrow} R + \gamma V(S')$
Q-Policy Iteration	Sarsa
$Q(s,a) \leftarrow \mathbb{E}[R + \gamma Q(S',A') \mid s,a]$	$Q(S,A) \overset{\alpha}{\leftarrow} R + \gamma Q(S',A')$
Q-Value Iteration	Q-Learning
$Q(s,a) \leftarrow \mathbb{E}\left[R + \gamma \max_{a' \in \mathcal{A}} Q(S',a') \mid s,a\right]$	$Q(S,A) \overset{\alpha}{\leftarrow} R + \gamma \max_{a' \in \mathcal{A}} Q(S',a')$

Figure 15: Relationship between TD and DP (2).

1.6 Exploration-Exploitation trade-off

The problem of choosing between maximizing reward given the current information (exploitation) or obtaining more information (exploration), which might offer more reward in the future.

- Exploration finds more information about the environment
- Exploitation exploits known information to maximise reward
- important to have a good balance between both worlds

1.6.1 epsilon greedy

Takes the optimal action with a probability of $1 - \epsilon$ and a random action with a probability of ϵ. Only focuses on optimal action, while giving all the others the same probability.

- Simplest idea for ensuring continual exploration
- All actions are tried with non-zero probability

1.6.2 Thompson sampling

$$\pi(a|s) = \frac{e^{\tau^{-1}Q(s,a)}}{\sum_{a' \in A} e^{\tau^{-1}Q(s,a')}} \tag{33}$$

- scales values for each action via a Boltzmann-distribution to achieve a relative score
- each action has it's own probability of being chosen, based on their value (relative-score)
- can be influence via a temperature parameter τ (large τ results in a small impact (more exploration) and small τ results in a large impact (more exploitation))

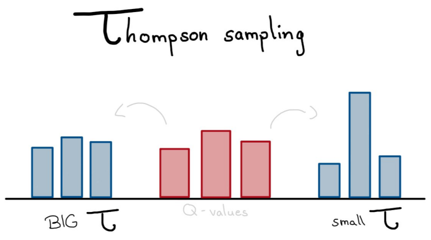


Figure 16: Thompson sampling.

1.6.3 Off-policy learning

behaviour policy $\beta(a|s)$ - used to sample from the environment (e.g., epsilon-soft policy)
target policy $\pi(a|s)$- used for control after training (greedy policy)

Advantageous compared to on-policy learning, as exploration is encoded within the behaviour policy so that the target policy can be used for optimal control.

Why off-policy learning?

- Learn from observing other agents
- Re-use experience generated from old policies

- Learn about optimal policy while following exploratory policy
- Learn about multiple policies while following one policy

1. **Importance sampling**:
   - adresses the problem, that the probability of trajectories occurring under $\pi(a|s)$ or $\beta(a|s)$ shifts
   $$\mathbb{E}[G_\pi(s)] \neq \mathbb{E}[G_\beta(s)] \tag{34}$$
   - change the probability of policy b with it's respective probability under the target policy π (for each time-step this would be $\frac{\pi(a_t|s_t)}{\beta(a_t|s_t)}$)
   $$G_\beta(s) = \prod_t \frac{\pi(a_t|s_t)}{\beta(a_t|s_t)} \tag{35}$$
   - to make use of importance sampling the support of β must contain the support of π; i.e. if the probability for any action under π is nonzero, β must also be nonzero and vice versa (accordingly in off-policy MC control with an greedy policy, one can not use $\frac{\pi(a_t|s_t)}{\beta(a_t|s_t)}$ as a weight, since the greedy policy has either a probability of one for an action or 0, hence $\frac{1}{\beta(a_t|s_t)}$ is used)
2. **Batch Methods - experience replay memory**
   Gradient descent is simple and appealing, but is not sample efficient. Batch methods seek to find the best fitting value function given the agent's experience. Updates to the value function are then executed via mini-batch stochastic-gradient descent.

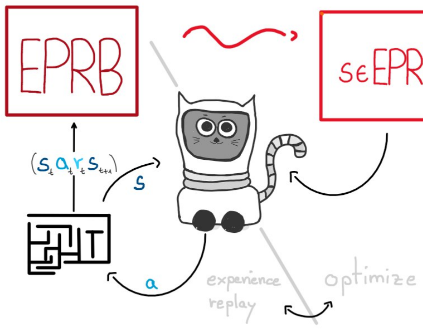


Figure 17: Experience replay illustration.

   - implemented as a deque, that stores n amount of samples and discards the oldest samples when appending newer samples (best would be to prioritize newer samples for older ones, when sampling one batch from the deque)
   - Least squares algorithms find parameter vector θ minimising sum-squared error between $\hat{v}(s_t, \theta)$ and target values v_t^π
   $$LS(\theta) = \sum_{t=1}^{T} (v_t^\pi - \hat{v}(s_t, \theta))^2 \quad = \mathbb{E}_D\left[(v_t^\pi - \hat{v}(s_t, \theta))^2\right] \tag{36}$$

1.7 The deadly triad

1. **Function approximation** - powerful and scalable way to approximate value functions with feasible memory and computational complexity compared to tabular methods (course of dimensionality)

2. **Bootstrapping** - update targets at t_0 based on existing estimates from t_{-1} (e.g. DP or TD), compared to update targets based on the actual Return G_t (MC)
3. **off-policy learning** - training on a distribution of transitions that wasn't produced by the target policy (plus env. dynamics)

Combination of all will cause instability and divergence during training.

2 Function Approximation

Scale up the model-free methods for prediction and control. Lecture regarding function approximation

Problem with large MDPs:

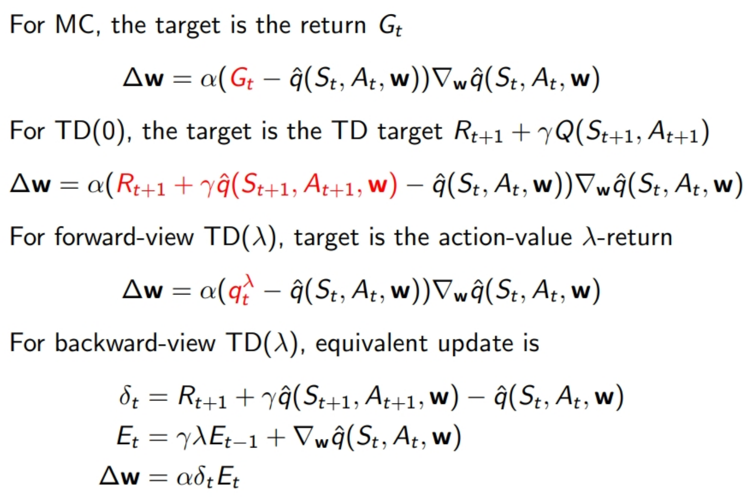
1. too many states and/or actions to store in memory (representation of Value functions by lookup tables not feasible; $V(s)$ represented as a vector of size S and $Q(s)$ as a matrix of size SxA)
2. too slow to learn the value of each state individually (necessary to compute a few hundred or thousand of updates for for each state)
3. curse of dimensionality (most states are quite similar, could be treated as the same)

Solution for large MDPs:

- estimate value function via function approximation (linear combination of features or artificial neural networks)

$$\begin{aligned} \hat{v}(s, \theta) &\approx v_\pi(s) \\ \hat{q}(s, a, \theta) &\approx q_\pi(s, a) \end{aligned} \tag{37}$$

- generalises from seen states to unseen
- parameters θ can be updated using MC or TD learning
- approximate policies can approach a deterministic policy, whereas in epsilon-greedy there is always an epsilon probability of selecting a random action
- enables selection of actions with arbitrary probabilities
- action probabilities change smoothly as a function of the learned policy, whereas in epsilon-greedy action probabilities might change dramatically for an arbitrary small change in the estimated action-value, if that change results in a different action having the maximal value



For MC, the target is the return G_t

$$\Delta \mathbf{w} = \alpha(G_t - \hat{q}(S_t, A_t, \mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(S_t, A_t, \mathbf{w})$$

For TD(0), the target is the TD target $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$

$$\Delta \mathbf{w} = \alpha(R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(S_t, A_t, \mathbf{w})$$

For forward-view TD(λ), target is the action-value λ-return

$$\Delta \mathbf{w} = \alpha(q_t^\lambda - \hat{q}(S_t, A_t, \mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(S_t, A_t, \mathbf{w})$$

For backward-view TD(λ), equivalent update is

$$\begin{aligned} \delta_t &= R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w}) \\ E_t &= \gamma\lambda E_{t-1} + \nabla_{\mathbf{w}}\hat{q}(S_t, A_t, \mathbf{w}) \\ \Delta \mathbf{w} &= \alpha\delta_t E_t \end{aligned}$$

Figure 18: Incremental Prediction Algorithms assume true value function provided by the supervisor, but in RL that's not possible, accordingly the target for $q_\pi(s, a)$ is substituted and can then be used with e.g. stochastic gradient descent.

What can be general issues with combining function approximation and Reinforcement Learning?

3 Deep Q-Network

- substitutes the Q-function in Q-learning with a deep neural network called Q-network
- **shared network** with a different action-value for each single action is preferable to one network for each action (learn underlying structures (like image recognition - perceptive fields; embedding of the state space) shared between all actions)
- mostly MLP; for state spaces that are represented by images, a CNN is preferred

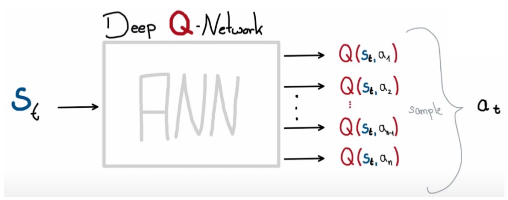


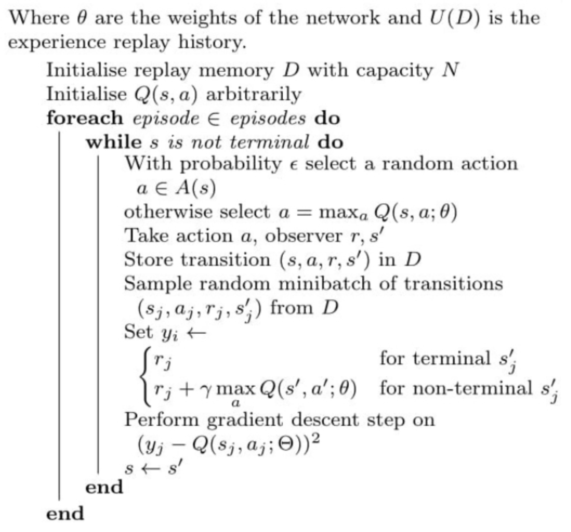
Figure 19: Deep Q-learning Q-network.

- exploration granted by epsilongreedy or Thompson-Sampling strategy (action for sampling)
- uses experience replay (samples random mini-batch of transitions for update) and fixed Q-targets (computes Q-learning targets w.r.t. old, fixed parameters θ_{t-1})
- uses variant of stochastic gradient descent to update parameters θ
- Loss function:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\left[r + \gamma \max_a Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)^2\right] \tag{38}$$

  with the target $\hat{t}$ in red and the prediction $\hat{y}$ in blue
- the target is an estimate of the real Q-value, as it's based on the previous network parameters (bootstrapping)

The Algorithm:



Where θ are the weights of the network and $U(D)$ is the experience replay history.

```
Initialise replay memory $D$ with capacity $N$
Initialise $Q(s,a)$ arbitrarily
foreach $episode \in episodes$ do
    while $s$ is not terminal do
        With probability $\epsilon$ select a random action
          $a \in A(s)$
        otherwise select $a = \max_a Q(s,a;\theta)$
        Take action $a$, observer $r, s'$
        Store transition $(s,a,r,s')$ in $D$
        Sample random minibatch of transitions
          $(s_j, a_j, r_j, s'_j)$ from $D$
        Set $y_i \leftarrow$
          $\begin{cases} r_j & \text{for terminal } s'_j \\ r_j + \gamma \max_a Q(s', a'; \theta) & \text{for non-terminal } s'_j \end{cases}$
        Perform gradient descent step on
          $(y_j - Q(s_j, a_j; \Theta))^2$
        $s \leftarrow s'$
    end
end
```

Figure 20: The Deep (Watkins) Q-learning Algorithm; note, the calculation of the target for terminal states.

3.1 Improvements to DQN

Rainbow Paper - all improvements mentioned listed here
Key problem in DQN: due to the recursive target definition for the Q_net update (taking the max of the successive q-values), an overestimated Q-value leads to overestimated targets and thus propagates through the state-space. For tabular q-learning, initial values are set to 0, which counteracts the problem of overestimation as now all q-values are underestimated, as well as updates to the q-values are based on the exponential weighted sum (weighted mean; α defines how far we look into the past), which uses underestimated q-values to update the current one (leads to underestimation of the to be updated q-value)
Recursive target definition DQN:

$$q_{target} = \hat{t} = r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \tag{39}$$

1. **Delayed DQN**
   - uses two q-networks for updating q-values
   - circumvents overestimation partially (rather slowing down) by using old q-value estimate for target calculation, thus the overestimation does not spread as fast (if target network overestimates, it will only affect the same states for the delay amount of updates, but after the target q-network update spread further)
   - the actual q-network $Q(s,a)$ and a delayed target q-network $\hat{Q}(s,a)$, used for the recursive target estimate
   - the target network parameters are fixed during computation and only updated after a specific delay by the actual q-networks parameters (copying the weights of one to the other netwrork)

$$qq_{target} = \hat{t} = r_t + \gamma \max_{a_{t+1}} \hat{Q}(s_{t+1}, a_{t+1}) \tag{40}$$

2. **Double Q-learning** - Paper
   - addresses the Problem, that $Q_{net}(s,a)$ systematically overestimates $Q_\pi(s,a)$ due to the recursive definition of the regression target
   - utilizes two DQNs $Q_1(s,a)$ and $Q_2(s,a)$, and changes the regression target to:

$$q_{target} = \hat{t} = r_t + \gamma min_{Q_1,Q_2}[\max_a Q_{net}(s_{t+1}, a_{t+1})] \tag{41}$$

   - reduces overestimation, as minimum of both Q-networks action-value is taken

3. **Prioritized Replay** - Paper
   - optimizer for replay buffer (sampling random mini-batches from experience reply memory for mini-batch SGD), that focuses on the priority at which a sample is considered for a q-net update
   - memorizes the last square error after each update
   - samples that have a larger error are drawn more often (e.g. via Thompson sampling), these are samples we can learn more from

$$\delta_{TD}^2 = (y - Q_{net}(s,a))^2 \tag{42}$$

4. **Dueling Networks** - Paper
   - split of action-value into state-value baseline and advantage-value, make estimations more stable and it can be used to understand which actions are actually meaning full (those with high variance in $A(s,a)$)
   - addresses the Problem, that often action-values $Q(s,a)$ for one state s are close to identical to each other (e.g. no specific action needed for the given state, as there is little to no advantage for any action)

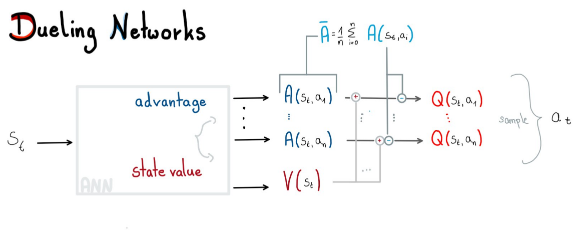


Figure 21: Illustration of the Duelling Networks architecture.

- the advantage-value $A(s,a)$ of an action a in a state s is defined as its deviation from the state-value $V(s)$
$$A(s,a) = Q(s,a) - V(s) \tag{43}$$
- network output: the state-value $V(s)$ and for each action an advantage-value $A(s,a)$
- training of the network is similar to training a Q-Network (regression), as the action-value $Q(s,a)$ can be recovered from the advantage-value $A(s,a)$ and state-value $V(s)$
$$Q(s,a) = V(s) + A(s,a) \tag{44}$$
- **Regularization:**
Subtracting the sum of all advantage-values from the action-value calculation ensures that networks output is actually the state-values and one advantage-value, not just the action-values encoded in the advantage-values, while the state-value remains 0.
$$Q(s,a) = V(s) + A(s,a) - \sum_{a'} A(s,a') \tag{45}$$

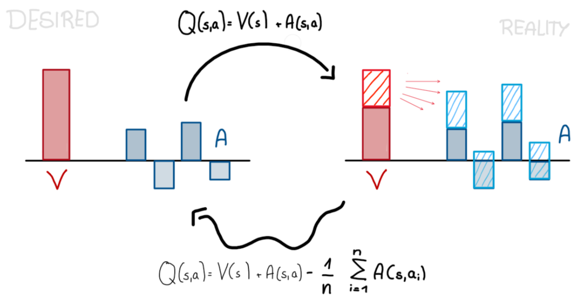

DESIRED
$Q(s,a) = V(s) + A(s,a)$
REALITY
V
A
V
A
$Q(s,a) = V(s) + A(s,a) - \frac{1}{n}\sum_{i=1}^{n} A(s,a_i)$

Figure 22: Split of the action-value $Q(s,a)$ into the state-value $V(s)$ and an advantage-value $A(s,a)$.

5. **Multi-step Learning - the 'SARSA' of DQN**
   - n-step TD-learning combined with the Q-learning estimate instead of the sampled action-value $Q(s,a)$ to define a new target:
$$q_{target} = \hat{t} = \sum_{i=0}^{n-1} [\gamma^i r_{t+i}] + \gamma^n \max_a Q_{net}(s_{t+n}, a_{t+n}) \tag{46}$$
   - faster propagation of Q-Values through state-action space

- more explicit variance-bias trade-off (like n-step SARSA compared to MC)

6. **Distributional RL** - Paper

   - estimates the distribution over state- or action-values and not Q_π directly
   - the expected value of that distribution is the respective state- or action-value
   - utilized for explicit usage like risk-aware behavior and provides a more stable value estimation

7. **Noisy Nets** - Paper

   - addresses the idea, that exploration of states without rigid action-value estimates should be increased
   - noisy nets enforce more exploration of rarely encountered states by introducing noise into the computation within network layers, which delays the actual learning process (needs to ignore noise first)

4 Policy Gradient Methods

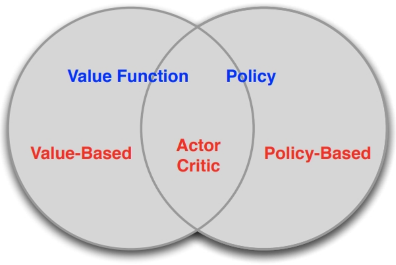


Figure 23: Sub-classes of value-function and policy based Reinforcement Learning.

- policy function should output the action that achieves the maximal action-value $Q(s, a)$ for a given state
- parameterizes the policy and learn/optimize the policy directly, without the usage/consolation of a value-function (value-function can still be used for updating policy parameter, but has no influence on the action-selection)
- problematic due to full batch style update, as trajectories need to be sampled with the same policy parameterization π_θ (on-policy: less efficient, would need approximation for batch-style learning (makes it off-policy) like importance sampling (introduces side effects))

$$\pi_\theta(s, a) = \mathbb{P}[a|s, \theta] \tag{47}$$

- can learn probabilities for taking an action, which allows for continuous action spaces, as we can output μ and σ according to a normal distribution over the action space
  - here value-based approaches would be computationally infeasible or too expensive, as one would need to compute the value estimates over an infinite amount of actions and/or states
  - for policy improvement in GPI (generalized policy iteration), it would require a full sweep trough the action space, as the improvement is achieved by $\arg\max_{a \in \mathcal{A}} Q^\pi(s, a)$

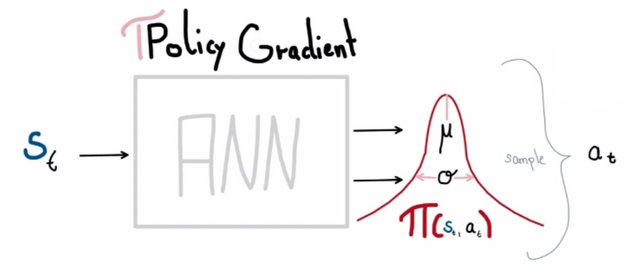


Figure 24: The policy-gradient network illustrated.

- **advantages**: effective in high-dimensional or continuous action spaces, can learn stochastic policies and has better convergence properties
- **disadvantages**: tends to converge to local optima, evaluating policies is generally inefficient and high in variance

Scalar performance measures $J(\theta)$ for the parameters of the policy π_θ:

- the sate-value function ($V^\pi(s,a) = \mathbb{E}_\pi[G_t|s_t]$) depends on a policy, that's why optimizing the parameter of the policy function is equal to optimizing the state-value function corresponding to that policy
- reward/performance function $J\theta$ as measure of quality for a given policy π_θ (Goal here: increase the cumulative reward (Return) the agent acquires)
- the gradient $\nabla J(\theta)$ is a stochastic estimate and approximates the gradient of the performance measure with respect to its argument θ
$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t) \tag{48}$$

Performance measures/ Reward functions:

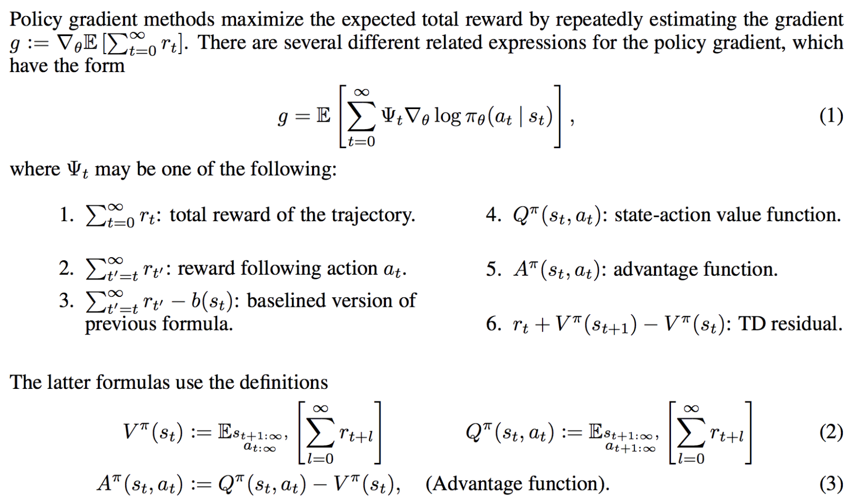
1. $J(\theta)$ defined as the **sate-value at** $s_{t=0}$ (for episodic environments; just the expected return starting in the particular state):
$$J_1(\theta) = \sum_{s \in \mathcal{S}} V^{\pi_\theta}(s) \tag{49}$$
2. $J(\theta)$ defined as the **average state-value** (for continuous environments):
$$J_{avV}(\theta) = \sum_{s \in \mathcal{S}} d^{\pi_\theta}(s) V^{\pi_\theta}(s) \tag{50}$$
3. $J(\theta)$ defined as the **average action-value** (for continuous environments):
$$J_{avV}(\theta) = \sum_{s \in \mathcal{S}} d^{\pi_\theta}(s) \sum_{a \in \mathcal{A}} \pi_\theta(a|s) Q^{\pi_\theta}(s,a) \tag{51}$$
with d^π as the **stationary distribution of Markov chain** for π_θ
$$d^\pi(s) = \lim_{t \to \infty} P(s_t = s | s_0, \pi_\theta) \tag{52}$$
   - the probability that $s_t = s$, when starting in state s_0 and following policy π_θ for t time-steps
   - more abstract: the probability for ending up in a specific state s when starting in a state s_0 and following policy π_θ

4.1 Generalized version of policy gradient methods

In actor-critic algorithms, the Critic would use policy evaluation (MC or TD learning) to estimate $Q^\pi(s,a)$, $A^\pi(s,a)$ or $V^\pi(s)$

4.2 Policy Gradient Theorem - How does it work, why do we need it?

- gives an exact formula for how performance is affected by the policy parameter that does not involve derivatives of the state distribution (foundation for all policy gradient algorithms)
- Policy based reinforcement learning is an optimisation problem, find θ for the policy π_θ that maximises the reward-function $J(\theta)$
- **Improve a policy-function by gradient ascent**: move the policy-function parameters θ into the direction suggested by the gradient of the reward-function $\nabla_\theta J(\theta)$
- $\nabla_\theta J(\theta)$ depends on two things:
   1. the action selected by the policy-function π_θ
   2. the stationary distribution over states (indirectly determined by the policy-function π_θ, as well as the state-transition dynamics of the environment $p(s'|s,a)$)



Policy gradient methods maximize the expected total reward by repeatedly estimating the gradient $g := \nabla_\theta \mathbb{E}\left[\sum_{t=0}^{\infty} r_t\right]$. There are several different related expressions for the policy gradient, which have the form

$$g = \mathbb{E}\left[\sum_{t=0}^{\infty} \Psi_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\right], \tag{1}$$

where Ψ_t may be one of the following:

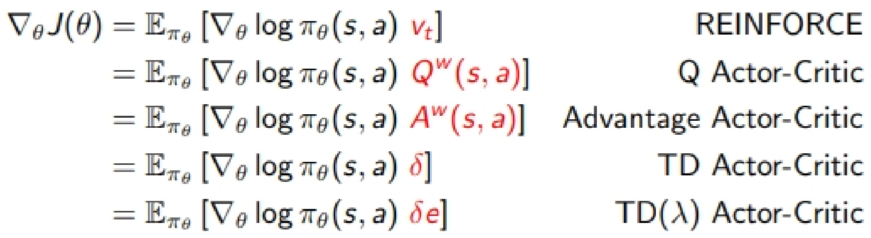
1. $\sum_{t=0}^{\infty} r_t$: total reward of the trajectory.
2. $\sum_{t'=t}^{\infty} r_{t'}$: reward following action a_t.
3. $\sum_{t'=t}^{\infty} r_{t'} - b(s_t)$: baselined version of previous formula.
4. $Q^\pi(s_t, a_t)$: state-action value function.
5. $A^\pi(s_t, a_t)$: advantage function.
6. $r_t + V^\pi(s_{t+1}) - V^\pi(s_t)$: TD residual.

The latter formulas use the definitions

$$V^\pi(s_t) := \mathbb{E}_{\substack{s_{t+1:\infty},\\ a_{t:\infty}}}\left[\sum_{l=0}^{\infty} r_{t+l}\right] \qquad Q^\pi(s_t, a_t) := \mathbb{E}_{\substack{s_{t+1:\infty},\\ a_{t+1:\infty}}}\left[\sum_{l=0}^{\infty} r_{t+l}\right] \tag{2}$$

$$A^\pi(s_t, a_t) := Q^\pi(s_t, a_t) - V^\pi(s_t), \quad \text{(Advantage function).} \tag{3}$$

Figure 25: Paper



$$\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ v_t\right] && \text{REINFORCE}\\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ Q^w(s,a)\right] && \text{Q Actor-Critic}\\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ A^w(s,a)\right] && \text{Advantage Actor-Critic}\\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ \delta\right] && \text{TD Actor-Critic}\\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ \delta e\right] && \text{TD}(\lambda)\text{ Actor-Critic}
\end{aligned}$$

Figure 26: Different types of policy gradient algorithms follow equivalent forms.

- **Problem:** in model-free RL the environment dynamics are unknown, accordingly it's not possible to estimate the effect on the stationary distribution when calculating the gradient ($\nabla_\theta J(\theta)$)
- Policy gradient theorem applies to start state objective, average reward and average value objective (different definitions of the reward-function)
- the policy gradient theorem generalises the likelihood ratio approach to multi-step MDPs
- **Generally:** reformation/transformation of the objective functions derivative to not include the derivative of the stationary distribution over states $d^{\pi_\theta}(s)$ (enables model-free policy-function updates)
- on-policy algorithm (pol-theorem)
  policy gradient theorem to find an estimate for the gradient of a given policy (estimate includes an expectation over states encountered when following that policy pi)
  policy gradient theorem only holds when your expectation (or in this case samples) are from that same distribution π (can be solved with for example importance sampling)

4.2.1 Proof - by lil'log

1. Derivative of the state-value

$$\begin{aligned}
\nabla_\theta V^\pi(s) =& \nabla_\theta \Big(\sum_{a\in\mathcal{A}} \pi_\theta(a|s) Q^\pi(s,a) \Big) \\
=& \sum_{a\in\mathcal{A}} \Big(\nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) + \pi_\theta(a|s) \nabla_\theta Q^\pi(s,a) \Big) && \text{;Derivative product rule.} \\
=& \sum_{a\in\mathcal{A}} \Big(\nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) + \pi_\theta(a|s) \nabla_\theta \sum_{s',r} P(s',r|s,a)(r + V^\pi(s')) \Big) && \text{;Extend } Q^\pi \text{ with future state value.} \\
=& \sum_{a\in\mathcal{A}} \Big(\nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) + \pi_\theta(a|s) \sum_{s',r} P(s',r|s,a) \nabla_\theta V^\pi(s') \Big) && \text{;}P(s',r|s,a) \text{ and } P(s',r|s,a)r \text{ is not a func of } \theta \\
=& \sum_{a\in\mathcal{A}} \Big(\nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) + \pi_\theta(a|s) \sum_{s'} P(s'|s,a) \nabla_\theta V^\pi(s') \Big) && \text{;}P(s'|s,a)=\textstyle\sum_r P(s',r|s,a)
\end{aligned} \tag{53}$$

Solution:

$$\nabla_\theta V^\pi(s) = \sum_{a\in\mathcal{A}} \Big(\nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) + \pi_\theta(a|s) \sum_{s'} P(s'|s,a) \nabla_\theta V^\pi(s') \Big) \tag{54}$$

- term for each time-step
- recursive definition, state-value function can be repetitively unrolled (bootstrapping can be used here)

2. Unrolling of the recursive representation

- the state transition probability (probability of transitioning from state s to state s' with policy π_θ in k time-steps) is defined as:

$$\rho^\pi(s \to s', k=1) = \sum_a \pi_\theta(a|s) P(s'|s,a) \tag{55}$$

with k=0

$$\rho^\pi(s \to s, k=0) = 1 \tag{56}$$

- extending $J(\theta)$ by exploiting it's recursiveness:

$$\phi(s) = \sum_{a\in\mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) \tag{57}$$

$$\rho^\pi(s \to x, k+1) = \sum_{s'} \rho^\pi(s \to s', k) \rho^\pi(s' \to x, 1) \tag{58}$$

stationary distribution $d^\pi(s)$:

$$d^\pi(s) = \frac{\eta(s)}{\sum_s \eta(s)} = \frac{\sum_{k=0}^\infty \rho^\pi(s_0 \to s, k)}{\sum_s \sum_{k=0}^\infty \rho^\pi(s_0 \to s, k)} \tag{59}$$

$$\begin{aligned}
&\nabla_\theta V^\pi(s)\\
=&\phi(s) + \sum_a \pi_\theta(a|s)\sum_{s'} P(s'|s,a)\nabla_\theta V^\pi(s')\\
=&\phi(s) + \sum_{s'}\sum_a \pi_\theta(a|s) P(s'|s,a)\nabla_\theta V^\pi(s')\\
=&\phi(s) + \sum_{s'} \rho^\pi(s\to s',1)\nabla_\theta V^\pi(s')\\
=&\phi(s) + \sum_{s'} \rho^\pi(s\to s',1)\nabla_\theta V^\pi(s')\\
=&\phi(s) + \sum_{s'} \rho^\pi(s\to s',1)[\phi(s') + \sum_{s''}\rho^\pi(s'\to s'',1)\nabla_\theta V^\pi(s'')]\\
=&\phi(s) + \sum_{s'} \rho^\pi(s\to s',1)\phi(s') + \sum_{s''}\rho^\pi(s\to s'',2)\nabla_\theta V^\pi(s'') \text{ ; Consider } s' \text{ as the middle point for } s\to s'' \textbf{ eq. 57}\\
=&\rho^\pi(s\to s,0)\phi(s) + \sum_{s'} \rho^\pi(s\to s',1)\phi(s') + \sum_{s''}\rho^\pi(s\to s'',2)\phi(s'') + \sum_{s'''}\rho^\pi(s\to s''',3)\nabla_\theta V^\pi(s''')\\
=&\ldots\text{; Repeatedly unrolling the part of } \nabla_\theta V^\pi(.)\\
=&\sum_{x\in\mathcal{S}}\sum_{k=0}^{\infty}\rho^\pi(s\to x,k)\phi(x)
\end{aligned} \tag{60}$$

extending $\nabla_\theta V^\pi(\cdot)$ infinitely, it is easy to find out that we can transition from the starting state s to any successor state, after any number of steps in this unrolling process and by summing up all the visitation probabilities, we get $\nabla_\theta V^\pi(s)$

3. now Recap and fancy re-structuring

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta V^\pi(s_0) && \text{; Starting from a random state } s_0\\
&= \nabla_\theta\Big(\sum_{a\in\mathcal{A}}\pi_\theta(a|s)Q^\pi(s,a)\Big)\\
&= \sum_s\sum_{k=0}^{\infty}\rho^\pi(s_0\to s,k)\phi(s) && \text{; Let } \eta(s)=\textstyle\sum_{k=0}^{\infty}\rho^\pi(s_0\to s,k)\\
&= \sum_s \eta(s)\sum_{a\in\mathcal{A}}\nabla_\theta\pi_\theta(a|s)Q^\pi(s,a)\\
&= \Big(\sum_s\eta(s)\Big)\sum_s\frac{\eta(s)}{\sum_s\eta(s)}\sum_{a\in\mathcal{A}}\nabla_\theta\pi_\theta(a|s)Q^\pi(s,a) && \text{; Normalize } \eta(s), s\in\mathcal{S} \text{ to be a probability distribution.}\\
&\propto \sum_s\frac{\eta(s)}{\sum_s\eta(s)}\sum_{a\in\mathcal{A}}\nabla_\theta\pi_\theta(a|s)Q^\pi(s,a) && \textstyle\sum_s\eta(s) \text{ is a constant}\\
&= \sum_s d^\pi(s)\sum_a\nabla_\theta\pi_\theta(a|s)Q^\pi(s,a) && d^\pi(s)=\tfrac{\eta(s)}{\sum_s\eta(s)} \text{ is stationary distribution.}
\end{aligned} \tag{61}$$

4. introducing another policy $\beta(a|s)$ (used for off-policy learning, whereas the same would be on-policy learning $\beta(a|s) = \pi(a|s)$)
on- or off-policy depending on what policy the stationary state distribution and action distributions

follow

$$\begin{aligned}\nabla_\theta J(\theta) &\propto \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} Q^\pi(s,a)\nabla_\theta \pi_\theta(a|s) \\ &= \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} \pi_\theta(a|s) Q^\pi(s,a)\frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} \\ &= \mathbb{E}_\pi[Q^\pi(s,a)\nabla_\theta \ln \pi_\theta(a|s)] \qquad \text{; Because } (\ln x)'=1/x\text{, we only consider the actual action taken and } d\to\mathbb{E}\end{aligned} \tag{62}$$

4.2.2 Problems with Policy-gradient updates

- problematic due to full batch style update, as trajectories need to be sampled with the same policy parameterization π_θ (**on-policy**: less efficient, would need approximation for batch-style learning (makes it off-policy) like importance sampling (introduces side effects))

$$\nabla v_\pi(s_0) \propto \sum_{s\in S}[\mu(s)\sum_a[\nabla\pi(a|s)q(s,a)]] \tag{63}$$

  with the stationary distribution $\mu(s)$ and following policy π_θ, this is an on-policy approach

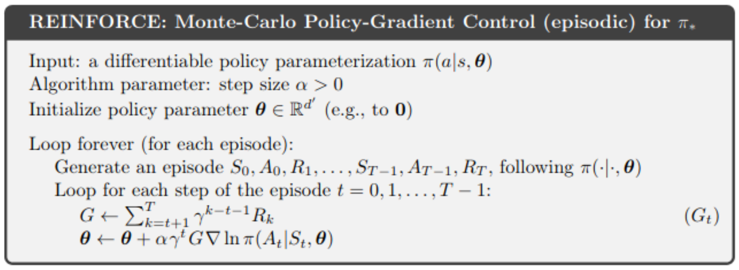
4.3 Vanilla Policy Gradient algorithm - REINFORCE

- implementation of the policy gradient theorem with Monte-Carlo estimate (return G_t) for the action- or state-value $Q^{\pi}(s,a)$
- uses gradient ascend to maximize $\max_\theta \mathbb{E}[G_t|s_t,a_t]$
- relies on sampling full trajectories from the environment (needs to sample all future rewards), to calculate G_t (only environments that terminate)
- no bias but high variance (like MC-learning) which slows down learning
- expectation of the sample gradient is equal to the actual gradient:

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_\pi[Q^{\pi_\theta}(s,a)\nabla_\theta \ln \pi_\theta(a|s)] \\ &= \mathbb{E}_\pi[G_t \nabla_\theta \ln \pi_\theta(A_t|S_t)] \qquad \text{; Because } Q^{\pi}(S_t,A_t)=\mathbb{E}_\pi[G_t|S_t,A_t]\end{aligned} \tag{64}$$

$$\theta_{t+1} = \theta_t + \alpha G_t \nabla \ln \pi(a_t|s_t,\theta) \tag{65}$$

without γ, as we do not discount (equivalent to $\gamma = 1$)



```
REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for $\pi_*$

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 $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
    Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$                    $(G_t)$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t G \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$
```

Figure 27: The REINFORCE algorithm.

γ^t in the algorithm, as here the discounted case of MC-learning is used, whereas for deriving the update equation an non-discounted ($\gamma = 1$) case was used.

REINFORCE with baseline:

- reduce the variance of gradient estimation while keeping the bias unchanged
- baseline leaves the expected value for the update unchanged, but it has an impact on the variance
- good baseline is the state-value function $b(s) = V^{\pi_\theta}(s)$, which turns the action-value estimate for the gradient computation into the Advantage of an action $A(s,a) = Q(s,a) - V(s)$

$$\begin{aligned}\theta_{t+1} &= \theta_t + \alpha(G_t - b(s_t))\nabla \ln \pi(a_t|s_t,\theta) \\ &= \theta_t + \alpha A(s,a)\nabla \ln \pi(a_t|s_t,\theta)\end{aligned} \tag{66}$$

without γ, as we do not discount (equivalent to $\gamma = 1$)
$G_t - b(s_t) = A(s,a) = Q(s,a) - V(s)$

4.4 Output-structure of pol-network - parameterization of policy $\pi_\theta^{net}(a|s)$

- also possible to use soft-max with or without temperature as target for the network, but here application for continuous action spaces
- learn statistics of a (normal) distribution over the real-valued action space (one or several actions)
- for 1-dim action space: one μ and one σ
- for n-dim action space: one μ and co-variance matrix for σ (lower triangle)
- μ can be approximated as linear function, but should be bounded by the bounds of the action space (tanH as output, to bound between -1 and 1; any other regularization technique to fix μ in a specific range)
- σ needs to be always positive (no linear output possible), use $\ln(\sigma)$ as network output and recover by $\exp(output)$
- re-parameterization trick just use μ

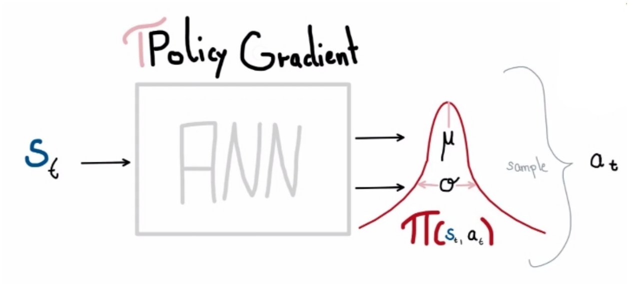
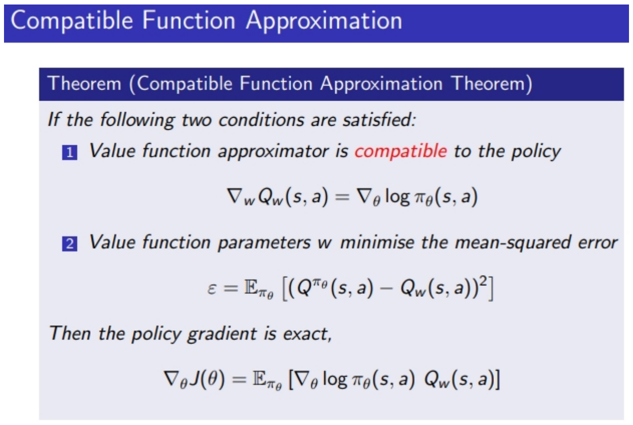


Figure 28: The policy-gradient network illustrated.

5 Actor-Critic Methods

- addresses the problem of high variance in sole policy-networks ('Monte-Carlo' - policy gradient with high variance)
- critic introduces bias into the actor's gradient estimates (approximates $\max_a Q(s,a)$), but substantially reduces variance in policy network updates (gradient calculation more stable; same as TD methods that use bootstrapping)
- critic has to be chosen carefully, as biased policy gradient may not find the right solution


Compatible Function Approximation

Theorem (Compatible Function Approximation Theorem)

If the following two conditions are satisfied:

1. *Value function approximator is compatible to the policy*

$$\nabla_w Q_w(s,a) = \nabla_\theta \log \pi_\theta(s,a)$$

2. *Value function parameters w minimise the mean-squared error*

$$\varepsilon = \mathbb{E}_{\pi_\theta}\left[(Q^{\pi_\theta}(s,a) - Q_w(s,a))^2\right]$$

Then the policy gradient is exact,

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ Q_w(s,a)\right]$$

Figure 29: The Compatibility function approximation Theorem proofs, that introducing a **compatible** value-function approximator for the policy gradient computation, will not make the gradient diverge from the actual one; by David Silver

- **Critic learns the value (value-function)** updates the value-function parameters w ($Q_w(a|s)$ or $V_w(s)$)
- **Actor learns the policy (policy function)** updates the policy parameters θ for $\pi_\theta(a|s)$, in the direction suggested by the critic ($Q_w(a|s)$ or $V_w(s)$)
- critic is actually doing policy evaluation (GPI - generalized policy iteration), by estimating the action-value function (evaluates goodness of the policy π_θ with current parameters θ)

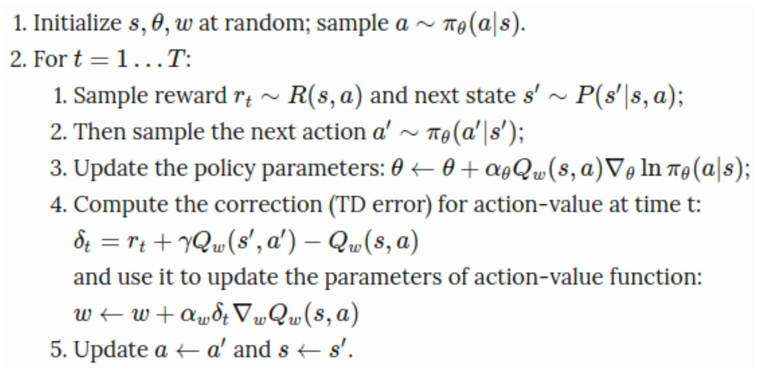
$$Q^{\pi_\theta}(s,a) \approx Q_w(s,a) \tag{67}$$

- Actor-critic algorithms follow an approximate policy gradient:

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_\pi[Q^{\pi_\theta}(s,a)\nabla_\theta \ln \pi_\theta(a|s)] \\ &= \mathbb{E}_\pi[Q_w(s,a)\nabla_\theta \ln \pi_\theta(A_t|S_t)]\end{aligned} \tag{68}$$

$$\delta_\theta = \alpha \nabla \log \pi_\theta(s,a) Q_w(s,a) \tag{69}$$

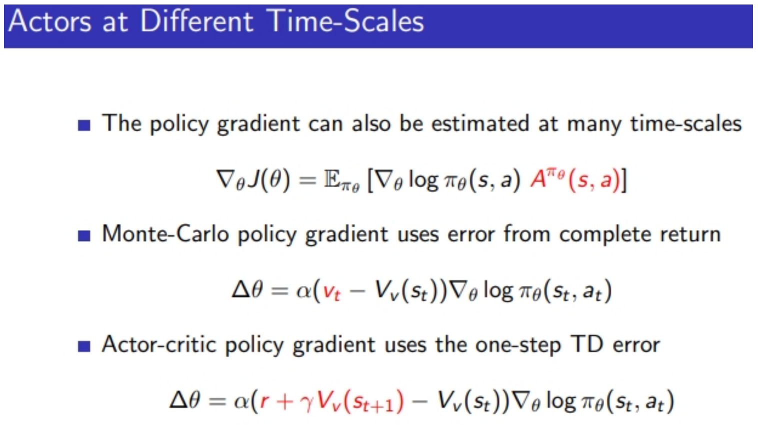
Vanilla actor-critic algorithm:



```
1. Initialize $s, \theta, w$ at random; sample $a \sim \pi_\theta(a|s)$.
2. For $t = 1 \dots T$:
    1. Sample reward $r_t \sim R(s, a)$ and next state $s' \sim P(s'|s, a)$;
    2. Then sample the next action $a' \sim \pi_\theta(a'|s')$;
    3. Update the policy parameters: $\theta \leftarrow \theta + \alpha_\theta Q_w(s, a) \nabla_\theta \ln \pi_\theta(a|s)$;
    4. Compute the correction (TD error) for action-value at time t:
       $\delta_t = r_t + \gamma Q_w(s', a') - Q_w(s, a)$
       and use it to update the parameters of action-value function:
       $w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s, a)$
    5. Update $a \leftarrow a'$ and $s \leftarrow s'$.
```

Figure 30: The vanilla action-value Actor-Critic algorithm. **Critic** Updates w by linear TD(0), **Actor** Updates θ by policy gradient.

Actor-critic learning timescales (Episodic or online)



Actors at Different Time-Scales

- The policy gradient can also be estimated at many time-scales

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s, a)\ A^{\pi_\theta}(s, a)\right]$$

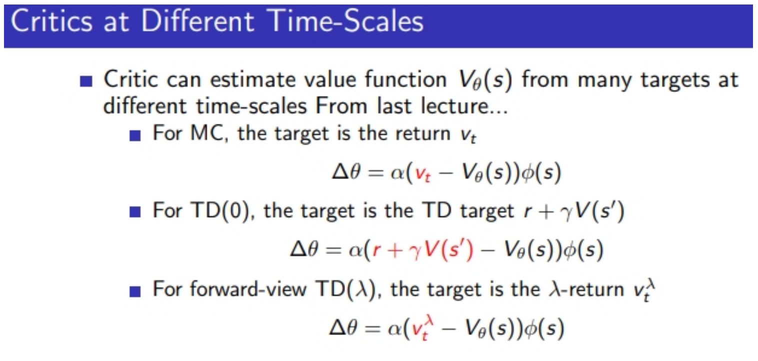
- Monte-Carlo policy gradient uses error from complete return

$$\Delta\theta = \alpha(v_t - V_v(s_t))\nabla_\theta \log \pi_\theta(s_t, a_t)$$

- Actor-critic policy gradient uses the one-step TD error

$$\Delta\theta = \alpha(r + \gamma V_v(s_{t+1}) - V_v(s_t))\nabla_\theta \log \pi_\theta(s_t, a_t)$$

Figure 31: Critic learning regimes.



Critics at Different Time-Scales

- Critic can estimate value function $V_\theta(s)$ from many targets at different time-scales From last lecture...
  - For MC, the target is the return v_t

$$\Delta\theta = \alpha(v_t - V_\theta(s))\phi(s)$$

  - For TD(0), the target is the TD target $r + \gamma V(s')$

$$\Delta\theta = \alpha(r + \gamma V(s') - V_\theta(s))\phi(s)$$

  - For forward-view TD(λ), the target is the λ-return v_t^λ

$$\Delta\theta = \alpha(v_t^\lambda - V_\theta(s))\phi(s)$$

Figure 32: Actor learning regimes.

5.1 Combination of policy-networks with deep Q-networks

5.1.1 DDPG (Deep Deterministic Policy Gradients)

- off-policy algorithm, which is constrained to environments with continuous action spaces
- combines the deterministic policy-network (DPG) with a DQN (improvements: experience replay and delayed DQN), which stabilizes the policy network training by estimating the action values via the Q-network (extension of DQN from discrete action spaces to continuous ones, while still outputting an deterministic action)
- **Deterministic policy gradient** (DPG) models the policy as a deterministic decision $a = \mu(s)$ (only one non-zero value, where only the preferred action has a probability of 1), instead of outputting a probability distribution over the actions (PG; stochastic policy)
- deterministic policy is more data efficient, as one does not learn action probabilities over the entire action space
- stochastic policy is eventually equivalent to the deterministic case when $\sigma = 0$
- in practice very sensitive to hyper-parameter choices (can't rescue itself from a 'failure-state' of the parameters)
- DDPS is off-policy learning compared to on-policy learning of vanilla pol-gradient, as the critic (q-net) is an off-policy q-value estimator (watkins-q-target makes it off-policy)
  - makes it sample efficient, as samples can be used several times (consider hard to get samples, e.g. real life, expensive, risky)

Network structure and training:

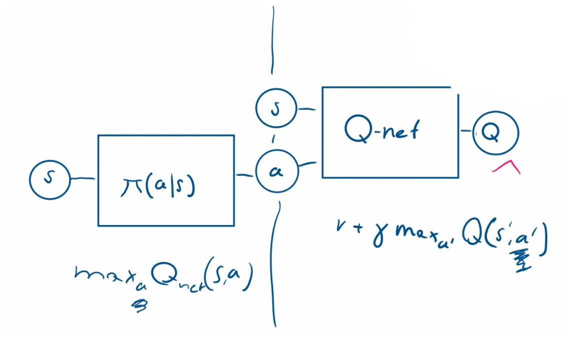


Figure 33: The DDPG-network illustrated.

- the **policy network** $\pi_\theta(a|s)$ is trained by maximizing $max_\theta \mathbb{E}_s[Q_\phi(s,a)]$
- the **Q-network** $Q_\pi(s,a)$ is trained by minimizin $max_\theta \mathbb{E}_s[Q_\phi(s,a)]$, with $q_{target} = r + \gamma max_{a'} Q(s',a')$ where $max_{a'} Q(s',a') : a' \ \sim \pi(a'|s')$ (max. q-value obtained by the output of the pol-network, as the network should resemble a greedy policy (output only μ))
- Q-net struggles to get optimal action (now achieved by pol-net) and pol-net struggles to get the derivative for the q-value (now achieved by Q-net)

Derivative of the performance measure $J(\theta)$:

$$\begin{aligned}\nabla_\theta J(\theta) &= \int_{\mathcal{S}} \rho^\mu(s) \nabla_a Q^\mu(s,a) \nabla_\theta \mu_\theta(s)|_{a=\mu_\theta(s)} ds \\ &= \mathbb{E}_{s\sim\rho^\mu}[\nabla_a Q^\mu(s,a) \nabla_\theta \mu_\theta(s)|_{a=\mu_\theta(s)}]\end{aligned} \tag{70}$$

Example with on-policy actor-critic algorithm, that uses SARSA update on the value-network:

$$\begin{aligned}
\delta_t &= R_t + \gamma Q_w(s_{t+1}, a_{t+1}) - Q_w(s_t, a_t) && \text{; TD error in SARSA} \\
w_{t+1} &= w_t + \alpha_w \delta_t \nabla_w Q_w(s_t, a_t) \\
\theta_{t+1} &= \theta_t + \alpha_\theta \nabla_a Q_w(s_t, a_t) \nabla_\theta \mu_\theta(s)|_{a=\mu_\theta(s)} && \text{; Deterministic policy gradient theorem}
\end{aligned} \tag{71}$$

Problems that make the learning unstable:

1. **'deterministic' action selection by the policy-network**

   - relying on a deterministic action might yield lower exploration, as network can collapse towards one action, which might be sub-optimal action and never recover

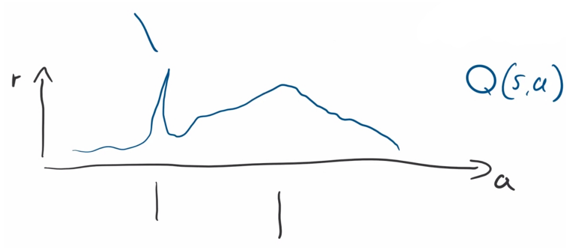


Figure 34: Bias in action selection of the pol-network when overestimating q-values by the q-network.

   - solution: add random noise to the output of the policy network

$$\mu'(s) = \mu_\theta(s) + \mathcal{N}(\mu_\mathcal{N}, \sigma_\mathcal{N}) \tag{72}$$

   where the added noise can be seen as the actual sigma, which results in sampling from the policy network according to a probability distribution parameterized by μ and σ, as noise is sampled from a probability distribution (actions will be sampled around μ (actions not considered with (s, a))); could also be fixed value

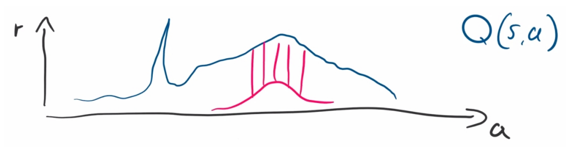


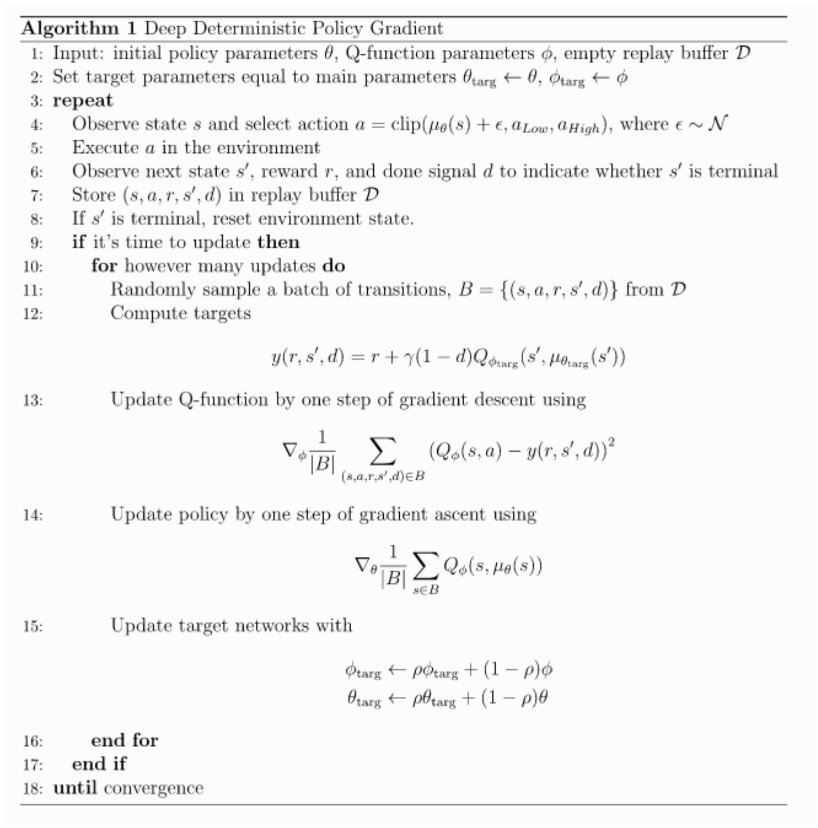
Figure 35: Make pol-net output μ probabilistic by adding noise.

2. **overestimation of the q-values by the q-network**

   - affects action selection by the policy network (will collapse towards max q-value output)
   - introduced by policy-network with it selecting the most rewarding action and the q-network, estimating all the q-values for a specific state
   - **solution:** delayed/slow update of both networks by **polyak averaging** (soft updates)

$$\theta_{target} \leftarrow \tau\theta_{target} + (1 - \tau)\theta \tag{73}$$

   with $0 < \tau < 1$ (can be seen as percentage of the actual target kept for the update)
   - target network values are constrained to change slowly, different from the design in DQN that the target network stays frozen for some period of time



Algorithm 1 Deep Deterministic Policy Gradient

```
1: Input: initial policy parameters $\theta$, Q-function parameters $\phi$, empty replay buffer $\mathcal{D}$
2: Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
3: repeat
4:   Observe state $s$ and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
5:   Execute $a$ in the environment
6:   Observe next state $s'$, reward $r$, and done signal $d$ to indicate whether $s'$ is terminal
7:   Store $(s, a, r, s', d)$ in replay buffer $\mathcal{D}$
8:   If $s'$ is terminal, reset environment state.
9:   if it's time to update then
10:    for however many updates do
11:      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
12:      Compute targets
             $y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$
13:      Update Q-function by one step of gradient descent using
             $\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q_\phi(s, a) - y(r, s', d))^2$
14:      Update policy by one step of gradient ascent using
             $\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$
15:      Update target networks with
             $\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi$
             $\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta$
16:    end for
17:  end if
18: until convergence
```

Figure 36: The DDPG algorithm with added noise to the pol-net output and polyak averaging.

5.1.2 TD3 (Twin delayed DDPG)

- off-policy algorithm
- constrained to environments with continuous action spaces
- continuation of the DDPG algorithm with major performance and stability boosts
- addresses the problem of DDPG being brittle to hyper-parameters and other kind of tuning
- generally DDPG fails as follows: first q-net overestimates q-values, second the policy-net exploits these errors/overestimates, which in turn breaks the policy
- critic-learns by minimizing Bellman Error

Three tricks that improve DDPG:

1. Clipped double Q-learning
   - same as in RainbowDQN, uses two q-networks to estimate the $Q(s, a)$ for the respective q-net update and uses the smaller value for the target computation $\min_{\phi_1,\phi_2} Q(s', a')$
   - reduces/delays q-value overestimation by the q-net

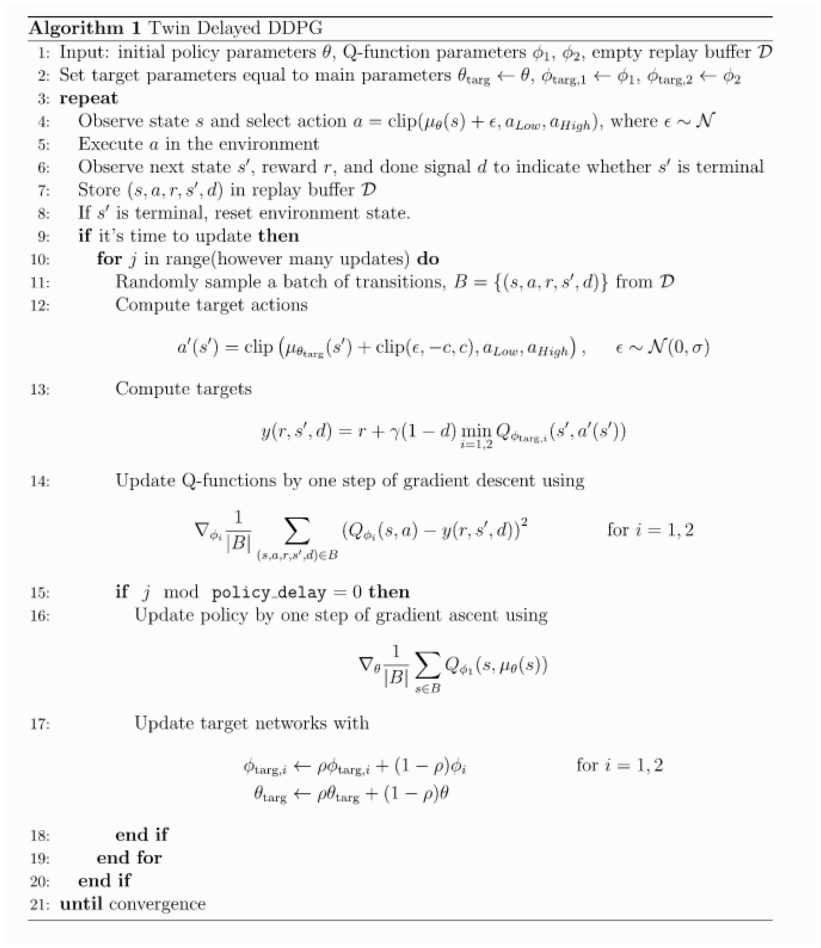
$$y_{target} = r + \gamma \min_{i=1,2} Q_{w_i}(s', \mu_\theta(s')) \tag{74}$$

2. Delayed policy updates
   - update policy-net less frequent then the q-net (one pol-net update per two q-net updates)

- reduces overestimation of q-network
  - Value estimates diverge through overestimation when the policy is poor
  - policy will become poor if the value estimate itself is inaccurate

3. Target policy smoothing
   - addresses the problem, that deterministic policies can over-fit to narrow peaks in the value function
   - add noise to the target action $a'(s')$
   - makes it harder for the policy network to exploit q-value overestimation, by smoothing out Q along changes in action

$$\begin{aligned} &y = r + \gamma Q_w(s', \mu_\theta(s') + \epsilon) \\ &\epsilon \sim \text{clip}(\mathcal{N}(0,\sigma), -c, +c) \qquad \text{; clipped random noises.} \end{aligned} \tag{75}$$



Algorithm 1 Twin Delayed DDPG

```
 1: Input: initial policy parameters $\theta$, Q-function parameters $\phi_1$, $\phi_2$, empty replay buffer $\mathcal{D}$
 2: Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
 3: repeat
 4:   Observe state $s$ and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
 5:   Execute $a$ in the environment
 6:   Observe next state $s'$, reward $r$, and done signal $d$ to indicate whether $s'$ is terminal
 7:   Store $(s, a, r, s', d)$ in replay buffer $\mathcal{D}$
 8:   If $s'$ is terminal, reset environment state.
 9:   if it's time to update then
10:     for $j$ in range(however many updates) do
11:       Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
12:       Compute target actions
              $a'(s') = \text{clip}\left(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{Low}, a_{High}\right), \quad \epsilon \sim \mathcal{N}(0, \sigma)$
13:       Compute targets
              $y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', a'(s'))$
14:       Update Q-functions by one step of gradient descent using
              $\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$
15:       if $j$ mod policy_delay $= 0$ then
16:         Update policy by one step of gradient ascent using
              $\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_{\phi_1}(s, \mu_\theta(s))$
17:         Update target networks with
              $\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2$
              $\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1 - \rho)\theta$
18:       end if
19:     end for
20:   end if
21: until convergence
```

Figure 37: The TD3 algorithm.

5.1.3 Entropy Methods and Soft Actor Critic

- off-policy algorithm, that incorporates the entropy measure of the policy into the reward to encourage exploration (maximum entropy reinforcement learning framework; maximise entropy for a given policy) and uses an actor-critic architecture for separate policy and value function networks
- addresses the problem of continuous action spaces and controlling the exploration-exploitation dilemma (policy-network has control over the policy location (where prob. distribution is located in the action space) and scale/probability of each action)
- pol-network that outputs μ and σ or a covariance-matrix, must decrease the learned σ to converge (increase the actions probability, while lowering other actions probabilities) to a optimal action (after μ is learned), which eventually decreases the exploration
- one simple solution: pol-net outputs μ and a hyper-parameter fixes/anneals σ to a specific value
- entropy as measure for how different the action-probability distribution is from a uniform-distribution

  Entropy for discrete valued actions spaces:

$$\begin{aligned}\mathcal{H} &= -\sum_a [\pi(a|s) log[\pi(a|s)]] \\ &= -\mathbb{E}_\pi[log[\pi(a|s)]]\end{aligned} \tag{76}$$

  with $-log[\pi(a|s)]$ as an estimate of $\mathcal{H}$
  Entropy for continuous valued actions spaces:

$$\mathcal{H} = -\int_a [\pi(a|s) log[\pi(a|s)]] \tag{77}$$

- entropy penalization/regularization of the objective function (Loss), by increasing the entropy over the action space, which is similar to making the probability of all the actions more equal (highest entropy when all actions are drawn with the same probability)

$$J(\theta) = \sum_{t=1}^{T} \mathbb{E}_{(s_t,a_t)\sim\rho_{\pi_\theta}}[r(s_t, a_t) + \alpha\mathcal{H}(\pi_\theta(.|s_t))] \tag{78}$$

  with $\mathcal{H}$ as entropy and α as weight for the impact of the entropy (also known as temperature parameter)

Entropy maximization RL

- **idea**: leave probability for all actions more equal, while giving the optimal action a preference
- entropy maximization leads to policies that can explore more and capture multiple modes of near-optimal strategies (does not necessary converge to one single optimal action, leaves possibility of equally good actions obtaining equal probabilities)
- maximize the (discounted, accumulated) reward and the (discounted, accumulated) entropy in combination (will lead to explore state-spaces which haven't been explored so far, as these states have high entropy (no preferred action)):

$$\hat{r}(s,a) = r(s,a) + \alpha H(\pi(a|s)) \tag{79}$$

  accordingly the return:

$$G = \sum_t [\gamma^t (r(s,a) + H(\pi(s,a)))] \tag{80}$$

  and the 1-step sarsa Q-estimate

$$r(s,a) + \alpha H(\pi(a|s)) + \gamma Q(s', a') \tag{81}$$

Training the Networks (lil'log):

1. policy π_θ with parameter θ
2. Soft Q-value function $Q_w(s,a)$ parameterized by w
3. Soft state-value V_ψ function parameterized by ψ (not necessary, could infer V from Q and π, but more stable when learned)

Soft Q-value and soft state value defined as:

$$\begin{aligned} Q(s_t,a_t) &= r(s_t,a_t) + \gamma \mathbb{E}_{s_{t+1}\sim\rho_\pi(s)}[V(s_{t+1})] && \text{; according to Bellman equation.} \\ \text{where } V(s_t) &= \mathbb{E}_{a_t\sim\pi}[Q(s_t,a_t) - \alpha \log \pi(a_t|s_t)] && \text{; soft state value function.} \end{aligned} \tag{82}$$

soft q-function updates:

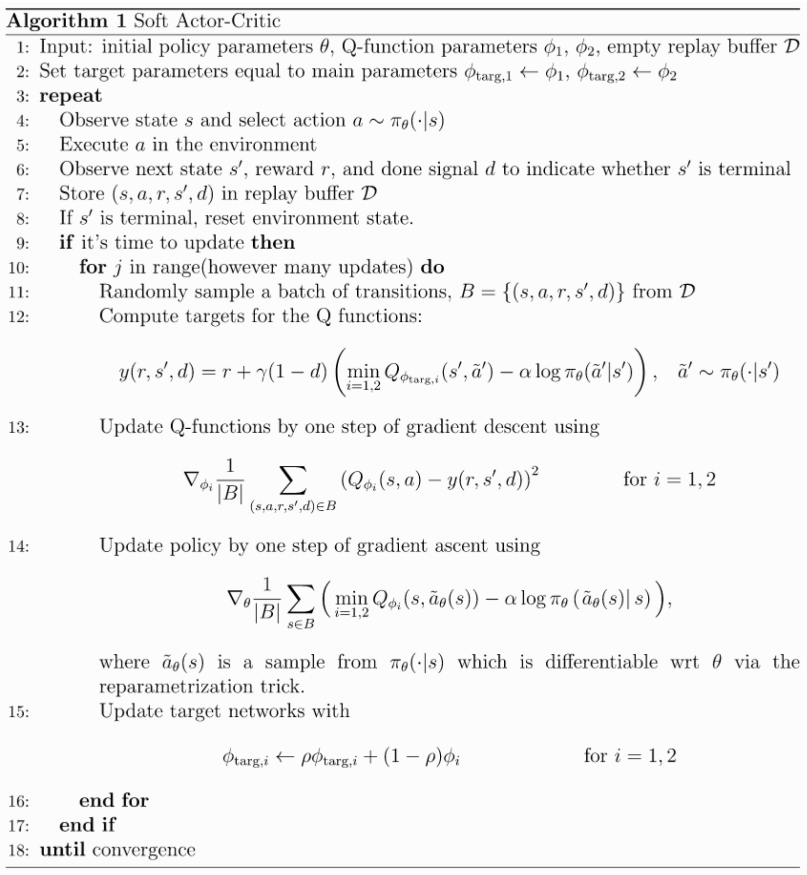
$$\begin{aligned} J_Q(w) &= \mathbb{E}_{(s_t,a_t)\sim\mathcal{D}}[\frac{1}{2}\big(Q_w(s_t,a_t) - (r(s_t,a_t) + \gamma \mathbb{E}_{s_{t+1}\sim\rho_\pi(s)}[V_{\bar{\psi}}(s_{t+1})])\big)^2] \\ \text{with gradient: } \nabla_w J_Q(w) &= \nabla_w Q_w(s_t,a_t)\big(Q_w(s_t,a_t) - r(s_t,a_t) - \gamma V_{\bar{\psi}}(s_{t+1})\big) \end{aligned} \tag{83}$$

soft value-function updates:

$$\begin{aligned} J_V(\psi) &= \mathbb{E}_{s_t\sim\mathcal{D}}[\frac{1}{2}\big(V_\psi(s_t) - \mathbb{E}[Q_w(s_t,a_t) - \log \pi_\theta(a_t|s_t)]\big)^2] \\ \text{with gradient: } \nabla_\psi J_V(\psi) &= \nabla_\psi V_\psi(s_t)\big(V_\psi(s_t) - Q_w(s_t,a_t) + \log \pi_\theta(a_t|s_t)\big) \end{aligned} \tag{84}$$

The SAC algorithm:

- SAC is a DDPG-style algorithm
- clipped double-Q Learning and delayed Q-networks
- no policy delay, because we want to use the current policy to obtain the recursive Q-target to obtain the Entropy estimate
- Q-estimations do not include the entropy term for the first time-step
- policy recovers the term for the entropy of the first time-step during policy optimization by maximizing $min_{Q_1,Q_2} Q(s,a) - \alpha log \pi(a|s)$



```
Algorithm 1 Soft Actor-Critic
1: Input: initial policy parameters $\theta$, Q-function parameters $\phi_1$, $\phi_2$, empty replay buffer $\mathcal{D}$
2: Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
3: repeat
4:   Observe state $s$ and select action $a \sim \pi_\theta(\cdot|s)$
5:   Execute $a$ in the environment
6:   Observe next state $s'$, reward $r$, and done signal $d$ to indicate whether $s'$ is terminal
7:   Store $(s, a, r, s', d)$ in replay buffer $\mathcal{D}$
8:   If $s'$ is terminal, reset environment state.
9:   if it's time to update then
10:    for $j$ in range(however many updates) do
11:      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
12:      Compute targets for the Q functions:
             $y(r, s', d) = r + \gamma(1-d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi_\theta(\cdot|s')$
13:      Update Q-functions by one step of gradient descent using
             $\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s,a) - y(r,s',d)\right)^2 \qquad \text{for } i = 1, 2$
14:      Update policy by one step of gradient ascent using
             $\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \left(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta\left(\tilde{a}_\theta(s)|\, s\right)\right),$
         where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt $\theta$ via the
         reparametrization trick.
15:      Update target networks with
             $\phi_{\text{targ},i} \leftarrow \rho\phi_{\text{targ},i} + (1-\rho)\phi_i \qquad \text{for } i = 1, 2$
16:    end for
17:  end if
18: until convergence
```

Figure 38: The SAC algorithm

with $-\alpha log \pi(a|s)$ as entropy estimate

5.2 Advantage Actor Critics

5.2.1 TD-residual learning (one step SARSA) with advantage and pol-gradient

- on-policy algorithm (because it uses the policy gradient theorem to find an estimate for the gradient of a given policy π (includes expectation over states encountered when following that policy))

Addresses the problem of vanilla policy gradients having high variance due to the Monte-Carlo estimate $Q(s,a) = \mathbb{E}[G_t|s,a]$.

$$\begin{aligned} \theta_t &= \theta_{t-1} + \alpha \mathbb{E}_\pi[q(s,a)\nabla_\theta \ln \pi_\theta(A_t|S_t)] \\ &= \theta_{t-1} + \alpha \mathbb{E}_\pi[G_t(s)\nabla_\theta \ln \pi_\theta(A_t|S_t)] \end{aligned} \tag{85}$$

1. **exchange of MC-estimate with TD-estimate to lower variance**

$$q(s,a) = \mathbb{E}_\pi[r(s,a) + v(s')] \tag{86}$$

which results in the estimator:

$$\theta_t = \theta_{t-1} + \alpha \mathbb{E}_\pi[(r + V_\phi(s'))\nabla_\theta \ln \pi_\theta(A_t|S_t)] \tag{87}$$

with $V_\phi(s')$ as an estimator of $v(s')$ (q-net parameterized by ϕ)

- q-net trained via MSE (regression), with the targets obtained from sampled trajectories

$$MSE = (G_t - V_\phi(s))^2 \tag{88}$$

2. **Introducing a state-dependent baseline (Advantage)** for the value-estimate when calculating the pol-gradient
The advantage of an action a in a state s is the deviation of that action from the state-value (another version of Q-value with lower variance by taking the state-value off the action-value as the baseline)

$$\begin{aligned} A(s,a) &= Q(s,a) - V(s) \\ &= (r + v(s')) - v(s) \\ &= q(s,a) - b(s) \end{aligned} \tag{89}$$

$$\begin{aligned} \nabla v_\pi(s_0) &\propto \mathbb{E}_\pi[(q(s,a) - b(s))\nabla ln\pi(a|s)] \\ &= \mathbb{E}_\pi[A(s,a))\nabla ln\pi(a|s)] \\ &= \mathbb{E}_\pi[(r + v(s') - v(s))\nabla ln\pi(a|s)] \end{aligned} \tag{90}$$

- using the state-value as a baseline has the advantage of the Advantage being zero centered
- expected value of the policy-gradient does not change, when applying a state-depended (action-independent) baseline, because rewriting the of the policy-gradient theorem with a baseline added will yield the original policy-gradient formulation (sum over all probabilities is 1, which's derivative is 0 and will cancel out the baseline)
- makes actions more comparable, so the policy-network can be trained more stable and efficiently, which results in faster convergence
- **general idea**: How much better is the action a then the average action $\sum_{a=0}^{A} a$ (average over all actions should be 0)
- Advantage can be positive or negative, just needs to sum up to 0
makes it easier to distinguish the best action for a given state, compared to q-values, which are more or less the same (either very big or very small) in many cases (e.g. state where each action is good, or state where it's not known which action will be the most rewarding in the future)
- makes states much more comparable, as Advantage values for different states do not diverge as drastically, which results in gradients with comparable sizes - good for ANNs - and that the policy-network can generalize much better

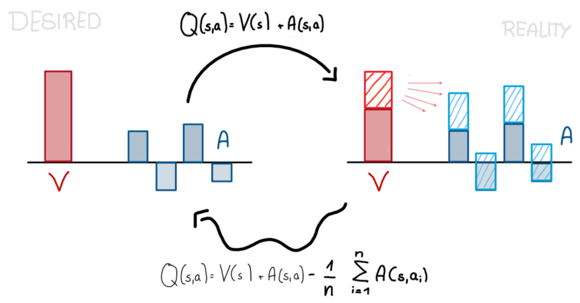
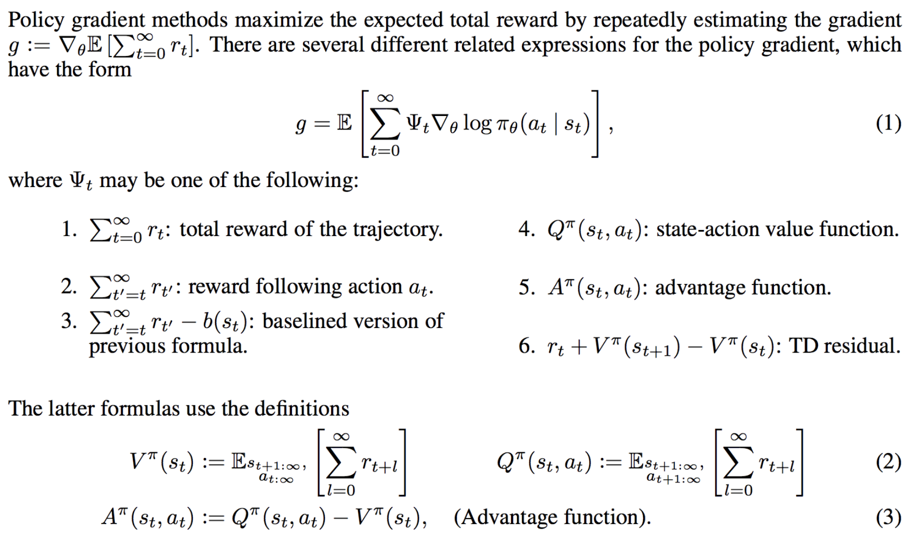


Figure 39: Split of the action-value $Q(s, a)$ into the state-value $V(s)$ and an advantage-value $A(s, a)$.

5.2.2 Generalized Advantage estimation - GAE

GAE - paper



Policy gradient methods maximize the expected total reward by repeatedly estimating the gradient $g := \nabla_\theta \mathbb{E}\left[\sum_{t=0}^{\infty} r_t\right]$. There are several different related expressions for the policy gradient, which have the form

$$g = \mathbb{E}\left[\sum_{t=0}^{\infty} \Psi_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\right], \tag{1}$$

where Ψ_t may be one of the following:

1. $\sum_{t=0}^{\infty} r_t$: total reward of the trajectory.
2. $\sum_{t'=t}^{\infty} r_{t'}$: reward following action a_t.
3. $\sum_{t'=t}^{\infty} r_{t'} - b(s_t)$: baselined version of previous formula.
4. $Q^\pi(s_t, a_t)$: state-action value function.
5. $A^\pi(s_t, a_t)$: advantage function.
6. $r_t + V^\pi(s_{t+1}) - V^\pi(s_t)$: TD residual.

The latter formulas use the definitions

$$V^\pi(s_t) := \mathbb{E}_{\substack{s_{t+1:\infty},\\ a_{t:\infty}}}\left[\sum_{l=0}^{\infty} r_{t+l}\right] \qquad Q^\pi(s_t, a_t) := \mathbb{E}_{\substack{s_{t+1:\infty},\\ a_{t+1:\infty}}}\left[\sum_{l=0}^{\infty} r_{t+l}\right] \tag{2}$$

$$A^\pi(s_t, a_t) := Q^\pi(s_t, a_t) - V^\pi(s_t), \quad \text{(Advantage function).} \tag{3}$$

Figure 40: Overview of policy gradient methods.

1. **Advancing the question of what (n)-step SARSA to use, to what (n)-step advantage term to use**
   1-step TD-error for pol gradient:

$$\delta_t^V = -V(s_t) + r_t + \gamma V(s_{t+1}) \tag{91}$$

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

$$
\hat{A}_t^{(k)} := \sum_{l=0}^{k-1} \gamma^l \delta_{t+l}^V = -V(s_t) + r_t + \gamma r_{t+1} + \cdots + \gamma^{k-1} r_{t+k-1} + \gamma^k V(s_{t+k})
$$

Figure 41: Advantage for n-steps.

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

$$
\hat{A}_t^{(k)} := \sum_{l=0}^{k-1} \gamma^l \delta_{t+l}^V = -V(s_t) + r_t + \gamma r_{t+1} + \cdots + \gamma^{k-1} r_{t+k-1} + \gamma^k V(s_{t+k})
$$

Figure 42: Advantage for n-steps in a concise form.

2. **expressing the n-step advantage in terms as a discounted exponentially weighted average with the scalar/decay factor λ over the n-step advantages/estimator s**

$$
\begin{aligned}
\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} &:= (1-\lambda)\Big(\hat{A}_t^{(1)} + \lambda \hat{A}_t^{(2)} + \lambda^2 \hat{A}_t^{(3)} + \dots\Big) \\
&= (1-\lambda)(\delta_t^V + \lambda(\delta_t^V + \gamma \delta_{t+1}^V) + \lambda^2(\delta_t^V + \gamma \delta_{t+1}^V + \gamma^2 \delta_{t+2}^V) + \dots) \\
&= (1-\lambda)(\delta_t^V(1 + \lambda + \lambda^2 + \dots) + \gamma \delta_{t+1}^V(\lambda + \lambda^2 + \lambda^3 + \dots) \\
&\qquad + \gamma^2 \delta_{t+2}^V(\lambda^2 + \lambda^3 + \lambda^4 + \dots) + \dots) \\
&= (1-\lambda)\left(\delta_t^V\left(\frac{1}{1-\lambda}\right) + \gamma \delta_{t+1}^V\left(\frac{\lambda}{1-\lambda}\right) + \gamma^2 \delta_{t+2}^V\left(\frac{\lambda^2}{1-\lambda}\right) + \dots\right) \\
&= \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V
\end{aligned}
$$

Figure 43: Generalised advantage estimation, SRASA(λ)

- formula for the generalized advantage estimation

$$
GAE(\gamma, \lambda) = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{TD(1)_l} \tag{92}
$$

  with $\delta_{TD(1)_l}$ as the one-step TD-error/ one-step SARSA error
- **advantages**: soft choice of the n-step term for our advantages as well as computationally efficient way between bias and variance trade-of

3. **Examples**

- $GAE(\gamma, 1)$ is γ-just regardless of the accuracy of V, but it has high variance due to the sum of terms (MC-estimate)

$$\text{GAE}(\gamma,0): \quad \hat{A}_t := \delta_t \qquad\qquad = r_t + \gamma V(s_{t+1}) - V(s_t)$$

$$\text{GAE}(\gamma,1): \quad \hat{A}_t := \sum_{l=0}^{\infty} \gamma^l \delta_{t+l} = \sum_{l=0}^{\infty} \gamma^l r_{t+l} - V(s_t)$$

Figure 44: Examples for GAE with $0 < \lambda < 1$, making compromise between bias an variance

- $GAE(\gamma, 0)$ is γ-just for $V = V_{\pi,\gamma}$ and otherwise induces bias, but it typically has much lower variance (one-step SARSA estimate)
- γ determines the scale of the value function $V^{\pi,\gamma}$; $\gamma < 1$ introduces bias into the policy gradient estimate
- $\lambda < 1$ introduces bias only if the value function is inaccurate
- λ introduces far less bias than γ for a reasonably accurate value function
- rule of thumb: $\lambda < \gamma$ ($\lambda = .95$ and $\gamma = .99$)

5.3 Trust region constrained actors

- mini-batch style updates for pol-gradient methods, constrained within local trust region
- on-policy updates (importance sampling)
- used for environments with either discrete or continuous action spaces

Generalization of trust region constraining methods

```
Algorithm 1 PPO, Actor-Critic Style
for iteration=1, 2, ... do
    for actor=1, 2, ..., $N$ do
        Run policy $\pi_{\theta_{\text{old}}}$ in environment for $T$ timesteps
        Compute advantage estimates $\hat{A}_1, \dots, \hat{A}_T$
    end for
    Optimize surrogate $L$ wrt $\theta$, with $K$ epochs and minibatch size $M \leq NT$
    $\theta_{\text{old}} \leftarrow \theta$
end for
```

Figure 45: The general Trust region constrained algorithm, input Loss from constrained loss functions below.

$$\begin{aligned}
\mathcal{L}^{\text{CLIP}}(\theta) &:= \mathop{\mathbb{E}}_{a,s\sim\pi_{\text{old}}}\left[\min\left(\frac{\pi_\theta(a|s)}{\pi_{\text{old}}(a|s)}\hat{A}^{\pi_{\text{old}}}(a,s),\ \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\text{old}}(a|s)}, 1-\epsilon, 1+\epsilon\right)\hat{A}^{\pi_{\text{old}}}(a,s)\right)\right] \\
\mathcal{L}^{\text{KL,forward}}(\theta) &:= \mathop{\mathbb{E}}_{a,s\sim\pi_{\text{old}}}\left[\frac{\pi_\theta(a|s)}{\pi_{\text{old}}(a|s)}\hat{A}^{\pi_{\text{old}}}(a,s)\right] - \beta D_{\text{KL}}(\pi_{\text{old}}||\pi_\theta) \\
\mathcal{L}^{\text{KL,reverse}}(\theta) &:= \mathop{\mathbb{E}}_{a,s\sim\pi_{\text{old}}}\left[\frac{\pi_\theta(a|s)}{\pi_{\text{old}}(a|s)}\hat{A}^{\pi_{\text{old}}}(a,s)\right] - \beta D_{\text{KL}}(\pi_\theta||\pi_{\text{old}})
\end{aligned}$$

Figure 46: Loss function with local trust region constrains (PPO, TRPO)

5.3.1 Trust Region Policy Optimization

- on-policy algorithm
- local constrained (trust region constraint) optimization algorithm (make multiple updated with the same data as long as we fulfill the KL constrain), to avoid parameter updates that change the policy too much at one step
- techniques are more or less unique to TRPO in DRL
- addresses the problem, that if doing several updates with trajectories sampled by the same policy, the stationary state distribution $\mu(s)$ also changes making the calculated gradient diverge from the actual (good approximation, but for how many steps ok?)
- TRPO uses KL-divergence to calculate how much the policy π_θ changes over updates and stops updating when the KL-divergence $KL(\pi_{new}(a|s)||\pi(a|s))$ (distance between both policies) is larger then a parameter δ

Optimize (Importance sampling style updates of pol-grad.):

$$\begin{aligned}
J(\theta) &= \sum_{s\in\mathcal{S}} \rho^{\pi_{\theta_{\text{old}}}} \sum_{a\in\mathcal{A}} \big(\pi_\theta(a|s)\hat{A}_{\theta_{\text{old}}}(s,a)\big) \\
&= \sum_{s\in\mathcal{S}} \rho^{\pi_{\theta_{\text{old}}}} \sum_{a\in\mathcal{A}} \big(\beta(a|s)\frac{\pi_\theta(a|s)}{\beta(a|s)}\hat{A}_{\theta_{\text{old}}}(s,a)\big) \quad \text{; Importance sampling} \\
&= \mathbb{E}_{s\sim\rho^{\pi_{\theta_{\text{old}}}}, a\sim\beta}\big[\frac{\pi_\theta(a|s)}{\beta(a|s)}\hat{A}_{\theta_{\text{old}}}(s,a)\big] \\
&= \mathbb{E}_{s\sim\rho^{\pi_{\theta_{\text{old}}}}, a\sim\pi_{\theta_{\text{old}}}}\big[\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}\hat{A}_{\theta_{\text{old}}}(s,a)\big]
\end{aligned} \tag{93}$$

mismatch between the training data distribution and the true policy state distribution is compensated by importance sampling estimator

while keeping KL-small:

$$\mathbb{E}_{s\sim\rho^{\pi_{\theta_{\text{old}}}}}[D_{\text{KL}}(\pi_{\theta_{\text{old}}}(.|s)\|\pi_\theta(.|s)] \leq \delta \tag{94}$$

old and new policies diverge not too much when this hard constraint is met

Combined to:

$$\nabla v_\pi(s_0) \propto \mathbb{E}_{s,a\sim\pi_{old}}[(\frac{\pi_{new}(a|s)}{\pi_{old}(a|s)}A) - \beta KL(\pi_{new}(a|s)||\pi_{old}(a|s))] \tag{95}$$

difficult to find appropriate hyper-parameter β (would need to be adapted throughout training)
constrained optimization not favoured in ANN/Deep learning

- TRPO reformulates this goal using a Taylor expansion approximation
- Using Lagrangian duality we can incorporate the optimization problem and the constraint into one new objective function
- this is then optimized using a combination of backtracking line search and conjugate gradient algorithm, where β, δ are hyper-parameters for the weighting the KL objective/constraint against the original optimization objective

5.3.2 PPO - proximal policy optimization

- on-policy algorithm
- in the continuous action domain quite stable against changes in hyper-parameters or environments
- 'lightweight' TRPO approach -apply multiply update steps (e.g. with mini-batches) from samples that were obtained by the current policy-, can be combined with other RL-approaches (GAE - general advantage estimation)
- simplifies TRPO by using a clipped surrogate objective while retaining similar performance
- good to combine with early stopping (based on KL-divergence between new and old policy)
- ratio between old and new policy (also used in TRPO)

$$r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} \tag{96}$$

- the objective function of PPO

$$J^{\text{CLIP}}(\theta) = \mathbb{E}[\min((r(\theta)\hat{A}_{\theta_{\text{old}}}(s,a)), (\text{clip}(r(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_{\theta_{\text{old}}}(s,a)))] \tag{97}$$

probability ratio between policies $r(\theta)$ constrained to a specific interval $[1-\epsilon, 1+\epsilon]$
when clipping the loss there is no gradient any more, which results in no changes made

Clipping in PPO: - Paper

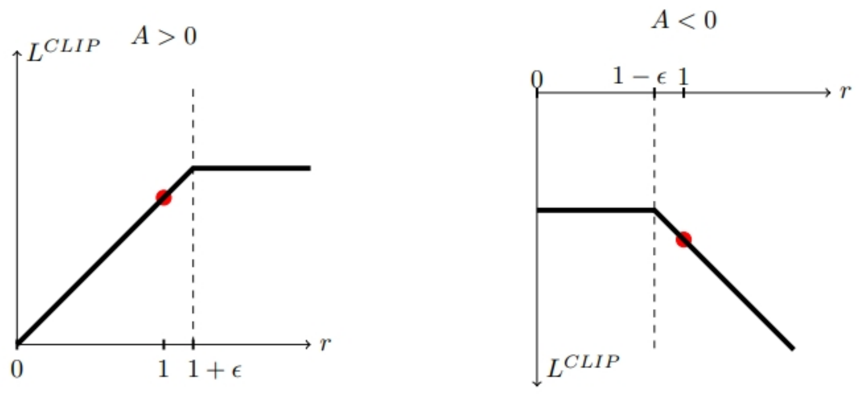


Figure 47: Visualization of the clipping achieved by PPO, see how the Loss gets constant outside the trust region constrain, which results in a gradient of 0. If the policy is outside of the specified confidence region (parameter ϵ), no updates are made (min term in loss function would now take the clipped version, as it refers to a gradient of 0)

- $A(s,a)$ - positive and $\frac{\pi_{new}(a|s)}{\pi_{old}(a|s)} > 1+\epsilon$
  probability ratio is out of trust-region: min operation chooses the clipped probability ratio and the advantage of that action (probability of taking the action) is not further increased
- $A(s,a)$ - positive and $\frac{\pi_{new}(a|s)}{\pi_{old}(a|s)} \leq 1+\epsilon$
  probability ratio is inside of trust-region: min operation chooses the non-clipped version of the gradient, the advantage of that action is further increased
- $A(s,a)$ - negative and $\frac{\pi_{new}(a|s)}{\pi_{old}(a|s)} < 1-\epsilon$
  probability ratio is out of our trust-region: min operation chooses the clipped probability ratio and the advantage of the action is not further decreased

- $A(s,a)$ - negative and $\frac{\pi_{new}(a|s)}{\pi_{old}(a|s)} \geq 1 - \epsilon$
  probability ratio is inside of our trust-region: min operation chooses the non-clipped version of the loss and the advantage of the action is further decreased

6 Regularization in Deep Reinforcement Learning

paper

7 Topics in dRL

1. **hindsight**
   - give failed trajectories still a reward and behave as if the goal was reached (learns dynamics from that)
2. **curriculum learning**
   - stepwise making tasks more challenging
   - Auto: two agents, one makes task, other one tries to solve it (rewards based on solvers performance; not too difficult, but not too easy)
3. **intrinsic motivation** (curiosity driven)
   - no extrinsic reward (or sparse), but agent has intrinsic motivation to explore state space (curiosity driven)
   - 1) encourage exploration of states not seen so far
   - 2) take Actions where it's not know what happens in the next state
   - Problem: some states/actions behave just noisy/random (every time new state/action)
4. **Sim2real**
   - Real world dynamics not perfectly modelled in simulation (reality gap)
   - No generalisation to different domains
   - Most of the time domain mismatch occurs
   - Domain randomisation (good combined with adversarial training) to make better generalisation
5. **Model based rl**
   Model known
   - optimise policy, while gathering trajectories
   - better training data observation and more efficient training Model unknown

   Model unknown
   - Vision model/Autoencoder (for dim reduction (bad to learn about large state spaces in RL); latent space representation of state)
   - environment model/RNN (predict next state based om the current state and action; learns env. dynamics)
   - controller model (agent; policy)
   - train agent directly (model free), train via model based approach, or do some fancy stuff like dreaming (output of the RNN is input for the next iteration) and training of the agent based on dreams
6. **Multi agent RL - MARL**

- make agents work together (interact)
- network of agents, that optimise together

Competitive self-play

- exploiting RL agents that did not explore the entire state space (do not know what to do in this state)
- training agents against each other makes rewards more prominent (slight updates in a policy might result in better performance, which is directly transferred into the self-play and will grant immediate reward)
- emergent complexity (fooling opponents; competitive self-play makes training tasks more complex by simple introduction of an opponent - kinda curriculum learning on it's own mediated through competition)
- Problem: adversarial exploitation of the opponents policy (fooling), by behaving unexpected which results in the opponent failing (state unknown, so no optimal behaviour; not learned during training)

Cooperative marl

- curriculum learning on its own
- 'one policy to rule them all' - paper that uses one policy to control all limbs of an artificial object (several agents with the same policy for each limb; consider different state spaces each agent is exposed to), outstanding generalisation, exchange of information between policies

Language emergence

- language emergence communication studies
- agents need to communicate in sequence (fixed set of tokens), as the goals of one agent can be entangled with one from another agent
- make agents interact on difficult/high level tasks; gives a training method that can make agents efficiently communicate with each other; outlook - make agents communicate with humans to execute their tasks efficiently (emergence towards natural languages); understand how natural language developed

7. **hierarchical RL**

- split policy into several sub-policies, that perform each a specific task and are managed by a higher-order main/manager-policy (both, main- and sub-policies need to be trained)
- alternative, not activation of policies managed by main-policy, but manager-policy assigns goals to specific sub-policies
- re-use of sub-policies; splitting of the environment goal into several sub-tasks, which can be spread across policies and managed by main-policy (question is, when to activate each policy; higher policy not so many times activate as sub-policies);

Glossary

A2C Advantage Actor-Critic

A3C Asynchronous Advantage Actor-Critic

DDPG Deep Deterministic Policy Gradient

DDQN Dueling DQN

DQN Deep Q-Network

DRL Deep Reinforcement Learning

ERP Experience Replay Buffer

HER Hindsight Experience Replay

PER Prioritized Experience Replay

Policy Gradient Policy Gradient (e.g. REINFORCE algorithm)

PPO Proximal Policy Optimization

SAC Soft Actor-Critic

SARSA State-Action-Reward-State-Action

TD Temporal Difference

TD3 Twin Delayed DDPG

TRPO Trust Region Policy Optimization