Model Predictive Control and Reinforcement Learning
– On-Policy Control with Function Approximation –

Joschka Boedecker and Moritz Diehl

University Freiburg

October 6, 2022

Lecture Overview

Acknowledgement

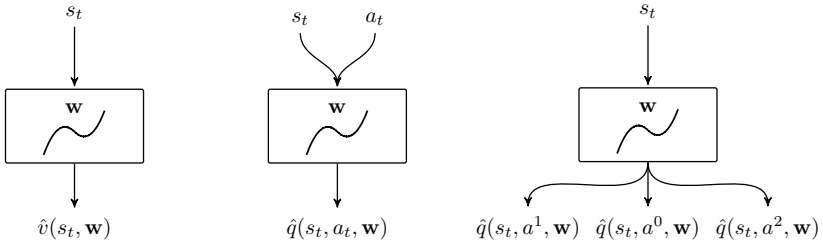
Slide contents are partially based on *Reinforcement Learning: An Introduction* by Sutton and Barto and the Reinforcement Learning lecture by David Silver.

Function Approximation in Reinforcement Learning

- Up to this point, we represented all elements of our RL systems by tables (value functions, models and policies)
- If the state and action spaces are very large or infinite, this is not a feasible solution
- We can apply function approximation to find a more compact representation of RL components and to generalize over states and actions
- Reinforcement Learning with function approximation comes with new issues that do not arise in Supervised Learning – such as non-stationarity, bootstrapping and delayed targets

Function Approximation in Reinforcement Learning

- Here: we estimate value-functions $v_\pi(\cdot)$ and $q_\pi(\cdot,\cdot)$ by function approximators $\hat{v}(\cdot,\mathbf{w})$ and $\hat{q}(\cdot,\cdot,\mathbf{w})$, parameterized by weights $\mathbf{w}$



- But we can also represent models or policies

Function Approximation in Reinforcement Learning

We can use different types of function approximators:

- Linear combinations of features
- Neural networks
- Decision trees
- Gaussian processes
- Nearest neighbor methods
- ...

Here: We focus on differentiable FAs and update the weights via gradient descent.

Function Approximation in Reinforcement Learning

We want to update our weights w.r.t. the *Mean Squared Value Error* of our prediction:

$$\begin{aligned}\mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{1}{2}\alpha\nabla[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)]^2 \\ &= \mathbf{w}_t + \alpha[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)]\nabla\hat{v}(S_t, \mathbf{w}_t)\end{aligned}$$

However, we don't have $v_\pi(S_t)$.

Function Approximation in Reinforcement Learning

Gradient MC

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[G_t - \hat{v}(S_t, \mathbf{w})]\nabla\hat{v}(S_t, \mathbf{w})$$

Semi-gradient TD(0)

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})]\nabla\hat{v}(S_t, \mathbf{w})$$

Why are bootstrapping methods, defined this way, called *semi-gradient methods*?

Function Approximation in Reinforcement Learning

Gradient MC

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[G_t - \hat{v}(S_t, \mathbf{w})]\nabla\hat{v}(S_t, \mathbf{w})$$

Semi-gradient TD(0)

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})]\nabla\hat{v}(S_t, \mathbf{w})$$

Why are bootstrapping methods, defined this way, called *semi-gradient methods*?
They take into account the effects of changing $\mathbf{w}$ w.r.t. the prediction, but not w.r.t. the target!

Linear Methods

- Represent state s by feature vector $\mathbf{x}(s) = (x^1(s), x^2(s), \ldots, x^d(s))^\top$
- These features can also be non-linear functions/combinations of state dimensions
- Linear methods approximate the value function by a linear combination of these features

$$\hat{v}(s, \mathbf{w}) = \mathbf{w}^\top \mathbf{x}(s) = \sum_{i=1}^{d} w^i x^i(s)$$

- Therefore, $\nabla_{\mathbf{w}} \hat{v}(s, \mathbf{w}) = \mathbf{x}(s)$
- Gradient MC prediction converges under linear FA
- On-policy linear semi-gradient TD(0) is stable
- Unfortunately, this does not hold for non-linear FA

Fixed point of on-policy linear semi-gradient TD

- The update at each time step t is:

$$\begin{aligned}\mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha \left(R_{t+1} + \gamma \mathbf{w}_t^\top \mathbf{x}_{t+1} - \mathbf{w}_t^\top \mathbf{x}_t\right) \mathbf{x}_t \\ &= \mathbf{w}_t + \alpha \left(R_{t+1}\mathbf{x}_t - \mathbf{x}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^\top \mathbf{w}_t\right)\end{aligned}$$

- The expected next weight vector can thus be written:

$$\mathbb{E}[\mathbf{w}_{t+1}|\mathbf{w}_t] = \mathbf{w}_t + \alpha(\mathbf{b} - \mathbf{A}\mathbf{w}_t),$$

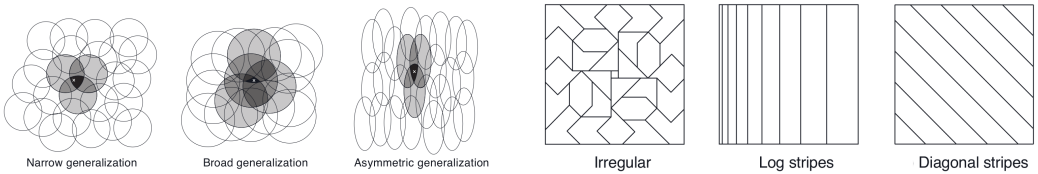
  where $\mathbf{b} = \mathbb{E}[R_{t+1}\mathbf{x}_t]$ and $\mathbf{A} = \mathbb{E}[\mathbf{x}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^\top]$

- If the system converges, it has to converge to the *fixed point*:

$$\mathbf{w}_{\text{TD}} = \mathbf{A}^{-1}\mathbf{b}$$

Coarse Coding

Divide the state space in circles/tiles/shapes and check in which some state is inside. This is a binary representation of the location of a state and leads to generalization.



On-policy Control with Function Approximation

- Again, up to this point we discussed Policy Evaluation based on state value functions
- In order to apply FA in control, we parameterize the action-value function

Semi-gradient SARSA

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w})]\nabla \hat{q}(S_t, A_t, \mathbf{w})$$

Semi-gradient SARSA

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

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    If S' is terminal:
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
      Go to next episode
    Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S'$
    $A \leftarrow A'$

Off-policy Learning

- We want to learn the optimal policy, but we have to account for the problem of *maintaining exploration*
- We call the (optimal) policy to be learned the *target policy* π and the policy used to generate behaviour the *behaviour policy* b
- We say that learning is from data *off* the target policy – thus, those methods are referred to as *off-policy learning*

Importance Sampling

- Weight returns according to the relative probability of target and behaviour policy
- Define state-transition probabilities $p(s'|s,a)$ as
  $p(s'|s,a) = \mathsf{Pr}\{S_t = s'|S_{t-1} = s, A_{t-1} = a\} = \sum_{r\in\mathcal{R}} p(s',r|s,a)$
- The probability of the subsequent trajectory under any policy π, starting in S_t, then is:

$$\begin{aligned}
&\mathsf{Pr}\{A_t, S_{t+1}, A_{t+1}, \dots S_T|S_t, A_{t:T-1} \sim \pi\} \\
&= \pi(A_t|S_t)p(S_{t+1}|S_t,A_t)\pi(A_{t+1}|S_{t+1})\cdots p(S_T|S_{T-1},A_{T-1}) \\
&= \prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k,A_k)
\end{aligned}$$

Importance Sampling

The relative probability therefore is:

Definition: Importance Sampling Ratio

$$\rho_{t:T-1} = \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k)p(S_{k+1}|S_k, A_k)} = \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k)}{\prod_{k=t}^{T-1} b(A_k|S_k)}$$

The expectation of the returns by b is $\mathbb{E}[G_t|S_t = s] = v_b(s)$. However, we want to estimate the expectation under π. Given the importance sampling ratio, we can transform the MC returns by b to yield the expectation under π:

$$\mathbb{E}[\rho_{t:T-1}G_t|S_t = s] = v_\pi(s).$$

Importance Sampling can come with a vast increase in variance.

Off-policy MC Prediction and Semi-gradient TD(0)

To use importance sampling with function approximation, replace the update to an array to an update to weight vector $\mathbf{w}$, and correct it with the importance sampling weight.

Off-policy MC Prediction

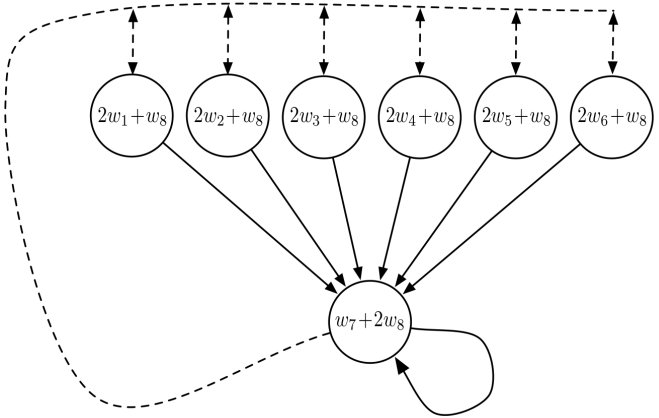
$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \rho_{t:T-1} [G_t - \hat{v}(S_t, \mathbf{w})] \nabla \hat{v}(S_t, \mathbf{w})$$

Semi-gradient Off-policy TD(0)

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

where $\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})$

Baird's Counterexample



$\pi(\text{solid}|\cdot) = 1$

$b(\text{dashed}|\cdot) = 6/7$

$b(\text{solid}|\cdot) = 1/7$

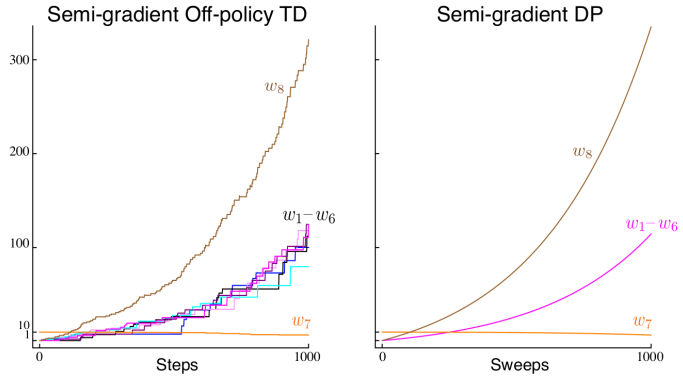
$\gamma = 0.99$

The reward is 0 for all transitions, hence $v_\pi(s) = 0$. This could be exactly approximated by $\mathbf{w} = \mathbf{0}$.

Baird's Counterexample

Semi-gradient DP

$$\mathbf{w} \leftarrow \mathbf{w} + \frac{\alpha}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} (\mathbb{E}[R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) | S_t = s] - \hat{v}(s, \mathbf{w})) \nabla \hat{v}(s, \mathbf{w})$$



The Deadly Triad

The combination of

- Function Approximation,
- Bootstrapping and
- Off-policy Learning

is known as the *Deadly Triad*, since it can lead to stability issues and divergence.

Neural Fitted-Q Iteration (NFQ) [Riedmiller 2005]

- Model-free off-policy RL algorithm that works on continuous state and discrete action spaces
- Q-function is represented by a multi-layer perceptron
- One of the first approaches that combined RL with ANNs, predecessor of DQN

Neural Fitted-Q Iteration (NFQ) [Riedmiller 2005]

for *iteration* $i = 1, .., N$ **do**

sample trajectory with ϵ-greedy exploration and add to memory D

initialize network weights randomly

generate pattern set $P = \{(x_j, y_j) | j = 1..|D|\}$ with

$x_j = (s_j, a_j)$ and $y_j = \begin{cases} r_j & \text{if } s_j \text{ is terminal} \\ r_j + \gamma \max_{a'} Q(s_{j+1}, a', \mathbf{w}_i) & \text{else} \end{cases}$

for *iteration* $k = 1, .., K$ **do**

Fit weights according to:

$$L(\mathbf{w}_i) = \frac{1}{|D|} \sum_{j=1}^{|D|} (y_j - Q(x_j, \mathbf{w}_i))^2$$

end

end

Algorithm 1: NFQ

Deep Q-Networks (DQN)

DQN provides a stable solution to deep RL:

- Use experience replay (as in NFQ)
- Sample minibatches (as opposed to Full Batch in NFQ)
- Freeze target Q-networks (no target networks in NFQ)
- Optional: Clip rewards or normalize network adaptively to sensible range

Deep Q-Networks: Experience Replay

To remove correlations, build data set from agent's own experience

- Take action a_t according to ϵ-greedy policy
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory D
- Sample random mini-batch of transitions (s, a, r, s') from D
- Optimize MSE between Q-network and Q-learning targets, e.g.

$$L(\mathbf{w}) = \mathbb{E}_{s,a,r,s' \sim D}\left[(r + \gamma \max_{a'} Q(s', a', \mathbf{w}) - Q(s, a, \mathbf{w}))^2\right]$$

Deep Q-Networks: Target Networks

To avoid oscillations, fix parameters used in Q-learning target

- Compute Q-learning targets w.r.t. old, fixed parameters $\mathbf{w}^-$

$$r + \gamma \arg\max_{a'} Q(s', a', \mathbf{w}^-)$$

- Optimize MSE between Q-network and Q-learning targets

$$L(\mathbf{w}) = \mathbb{E}_{s,a,r,s' \sim D}\left[(r + \gamma \max_{a'} Q(s', a', \mathbf{w}^-) - Q(s, a, \mathbf{w}))^2\right]$$

- Periodically update fixed parameters $\mathbf{w}^- \leftarrow \mathbf{w}$
  - hard update: update target network every N steps
  - slow update: slowly update weights of target network every step by

$$\mathbf{w}^- \leftarrow (1 - \tau)\mathbf{w}^- + \tau\mathbf{w}$$

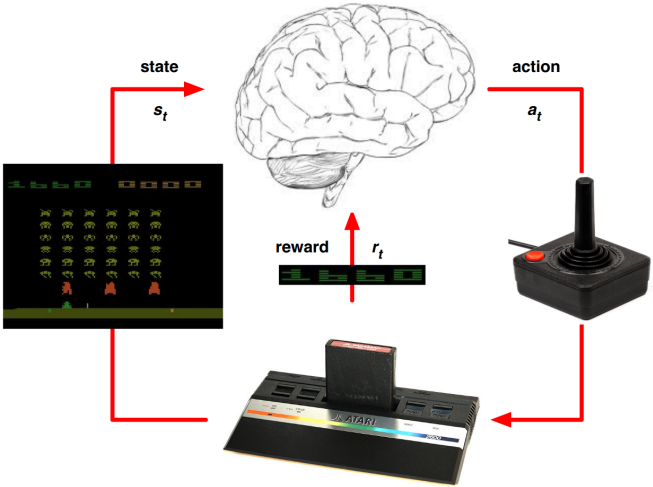
Deep Q-Networks (DQN)

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights
for *episode* $i = 1, .., M$ **do**
  for $t = 1, .., T$ **do**
    select action a_t ϵ-greedily
    Store transition (s_t, a_t, s_{t+1}, r_t) in D
    Sample minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from D
    Set $y_j = \begin{cases} r_j & \text{if } s_{j+1} \text{ is terminal} \\ r_j + \gamma \max_{a'} Q(s_{j+1}, a', \mathbf{w}^-) & \text{else} \end{cases}$
    Update the parameters of Q according to:

$$\nabla \mathbf{w}_i L_i(\mathbf{w}_i) = \mathbb{E}_{s,a,s,r \sim D}[(r + \gamma \max_{a'} Q(s', a', \mathbf{w}_i) - Q(s, a, \mathbf{w}_i)) \nabla_{\mathbf{w}_i} Q(s, a, \mathbf{w}_i)]$$

    Update target network
  end
end

Deep Q-Networks: Reinforcement Learning in Atari



Deep Q-Networks: Reinforcement Learning in Atari

- End-to-end learning of values $Q(s, a)$ from pixels s
- Input state s is a stack of raw pixels from the last 4 frames
- Output is $Q(s, a)$ for 18 joystick/button positions
- Reward is change in score for that step

How much does DQN help?

	Q-Learning	Q-Learning + Target Q	Q-Learning + Replay	DQN Q-learning + Replay + Target Q
Breakout	3	10	241	**317**
Enduro	29	142	831	**1006**
River Raid	1453	2868	4103	**7447**
Seaquest	276	1003	831	**2894**
Space Invaders	302	373	826	**1089**

Deep Deterministic Policy Gradient

- DDPG is an actor-critic method (*Continuous DQN*)
- Recall the DQN-target: $y_j = r_j + \gamma \max_a Q(s_{j+1}, a, \mathbf{w}^-)$
- In case of continuous actions, the maximization step is not trivial
- Therefore, we approximate deterministic actor μ representing the $\arg\max_a Q(s_{j+1}, a, \mathbf{w})$ by a neural network and update its parameters following the *Deterministic Policy Gradient Theorem*:

$$\nabla_\theta J \approx \frac{1}{N} \sum_j \nabla_a Q(s_j, a, \mathbf{w})|_{a=\mu(s_j)} \nabla_\theta \mu(s_j, \boldsymbol{\theta})$$

- Exploration by adding Gaussian noise to the output of μ

Deep Deterministic Policy Gradient

- The Q-function is fitted to the adapted TD-target:

$$y_j = r_j + \gamma Q(s_{j+1}, \mu(s_{j+1}, \boldsymbol{\theta}^-), \mathbf{w}^-)$$

- The parameters of target networks $\mu(\cdot, \boldsymbol{\theta}^-)$ and $Q(\cdot, \cdot, \mathbf{w}^-)$ are then adjusted with a soft update

$$\mathbf{w}^- \leftarrow (1-\tau)\mathbf{w}^- + \tau \mathbf{w} \text{ and } \boldsymbol{\theta}^- \leftarrow (1-\tau)\boldsymbol{\theta}^- + \tau \boldsymbol{\theta}$$

  with $\tau \in [0, 1]$
- DDPG is very popular and builds the basis for more SOTA actor-critic algorithms
- However, it can be quite unstable and sensitive to its hyperparameters

Deep Deterministic Policy Gradient

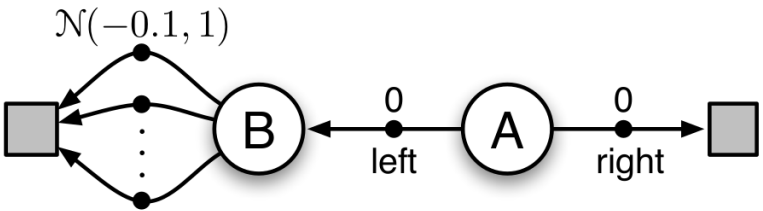
Initialize replay memory D to capacity N
Initialize critic Q and actor μ with random weights
for *episode* $i = 1, .., M$ **do**
  for $t = 1, .., T$ **do**
    select action $a_t = \mu(s_t, \boldsymbol{\theta}) + \epsilon$, where $\epsilon \sim \mathcal{N}(0, \sigma)$
    Store transition (s_t, a_t, s_{t+1}, r_t) in D
    Sample minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from D
    Set $y_j = \begin{cases} r_j & \text{if } s_{j+1} \text{ is terminal} \\ r_j + \gamma\ Q(s_{j+1}, \mu(s_{j+1}, \boldsymbol{\theta}^-), \mathbf{w}^-) & \text{else} \end{cases}$
    Update the parameters of Q according to the TD-error
    Update the parameters of μ according to:

$$\nabla_\theta J \approx \frac{1}{N} \sum_j \nabla_a Q(s_j, a, \mathbf{w})|_{a=\mu(s_j)} \nabla_\theta \mu(s_j, \boldsymbol{\theta})$$

    Adjust the parameters of the target networks via a soft update
  end
end

Overestimation Bias

- In all control algorithms so far, the target policy is created by the *maximization* of a value-function
- We thus consider the maximum over estimated values as an estimate of the maximum value
- This can lead to the so-called *overestimation bias*



Overestimation Bias

- Recall the Q-learning target: $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$
- Imagine two random variables X_1 and X_2:

$$\mathbb{E}[\max(X_1, X_2)] \geq \max(\mathbb{E}[X_1], \mathbb{E}[X_2])$$

- $Q(S_{t+1}, a)$ is not perfect – it can be *noisy*:

$$\max_a Q(S_{t+1}, a) = \overbrace{Q(S_{t+1}, \underbrace{\arg\max_a Q(S_{t+1}, a)}_{\text{action comes from } Q})}^{\text{value comes from } Q}$$

- If the noise in these is decorrelated, the problem goes away!

Double Q-learning

Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, such that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using the policy ε-greedy in $Q_1 + Q_2$
    Take action A, observe R, S'
    With 0.5 probabilility:

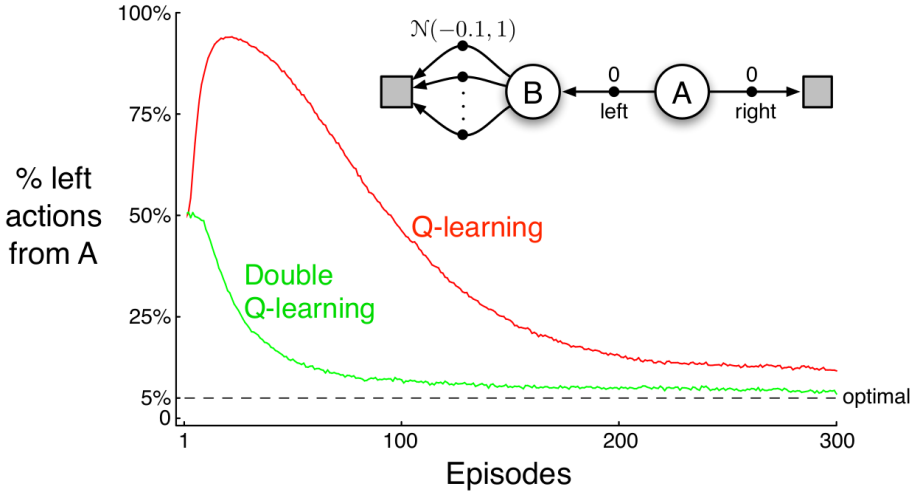
$$Q_1(S, A) \leftarrow Q_1(S, A) + \alpha\Big(R + \gamma Q_2\big(S', \operatorname{argmax}_a Q_1(S', a)\big) - Q_1(S, A)\Big)$$

    else:

$$Q_2(S, A) \leftarrow Q_2(S, A) + \alpha\Big(R + \gamma Q_1\big(S', \operatorname{argmax}_a Q_2(S', a)\big) - Q_2(S, A)\Big)$$

    $S \leftarrow S'$
  until S is terminal

Double Q-learning



TD3

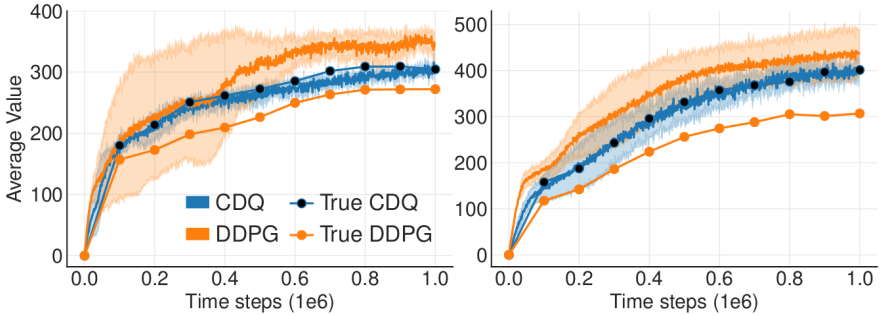
TD3 adds three adjustments to vanilla DDPG

- Clipped Double Q-Learning
- Delayed Policy Updates
- Target-policy smoothing

TD3: Clipped Double Q-Learning

- In order to alleviate the overestimation bias (which is also present in actor-critic methods), TD3 learns two approximations of the action-value function
- It the takes the minimum of both predictions as the second part of the TD-target:

$$y_j = r_j + \gamma \min_{i \in \{1,2\}} Q(s_{j+1}, \mu(s_{j+1}), \mathbf{w}_i^-)$$



TD3: Delayed Policy Updates

- Due to the mutual dependency between actor and critic updates...
  - values can diverge when the policy leads to overestimation and
  - the policy will lead to bad regions of the state-action space when the value estimates lack in (relative) accuracy
- Therefore, policy updates on states where the value-function has a high prediction error can cause divergent behaviour
- We already know how to compensate for that: target networks
- Freeze target and policy networks between d updates of the value function
- This is called a *Delayed Policy Update*

TD3: Target-policy Smoothing

- *Target-policy Smoothing* adds Gaussian noise to the next action in target calculation
- It transforms the Q-update towards an Expected SARSA update fitting the value of a small area around the target-action:

$$y_j = r_j + \gamma \min_{i \in \{1,2\}} Q(s_{j+1}, \mu(s_{j+1}) + \mathsf{clip}(\epsilon, -c, c), \mathbf{w}_i^-),$$

where $\epsilon \sim \mathcal{N}(0, \sigma)$

TD3: Ablation

Table 2. Average return over the last 10 evaluations over 10 trials of 1 million time steps, comparing ablation over delayed policy updates (DP), target policy smoothing (TPS), Clipped Double Q-learning (CDQ) and our architecture, hyper-parameters and exploration (AHE). Maximum value for each task is bolded.

Method	HCheetah	Hopper	Walker2d	Ant
TD3	9532.99	**3304.75**	**4565.24**	**4185.06**
DDPG	3162.50	1731.94	1520.90	816.35
AHE	8401.02	1061.77	2362.13	564.07
AHE + DP	7588.64	1465.11	2459.53	896.13
AHE + TPS	9023.40	907.56	2961.36	872.17
AHE + CDQ	6470.20	1134.14	3979.21	3818.71
TD3 - DP	9590.65	2407.42	**4695.50**	3754.26
TD3 - TPS	8987.69	2392.59	4033.67	**4155.24**
TD3 - CDQ	9792.80	1837.32	2579.39	849.75
DQ-AC	9433.87	1773.71	3100.45	2445.97
DDQN-AC	**10306.90**	2155.75	3116.81	1092.18