Lecture 10: Value-Based Control with Function Approximation

Oliver Wallscheid

Preface

Problem space: it is further assumed that

- the states $\boldsymbol{x}$ are (quasi-)continuous and
- the actions u are discrete.

Today's focus:

- valued-based control tasks, i.e., transferring the established tabular methods to work with function approximation.
- Hence, we need to extend the previous prediction methods to action values

$$\hat{q}(\boldsymbol{x}, u, \boldsymbol{w}) \approx q_\pi(\boldsymbol{x}, u). \tag{10.1}$$

- And apply the well-known generalized policy iteration scheme (GPI) to find optimal actions:

$$\hat{q}(\boldsymbol{x}, u, \boldsymbol{w}) \approx q^*(\boldsymbol{x}, u). \tag{10.2}$$

Types of action-value function approximation

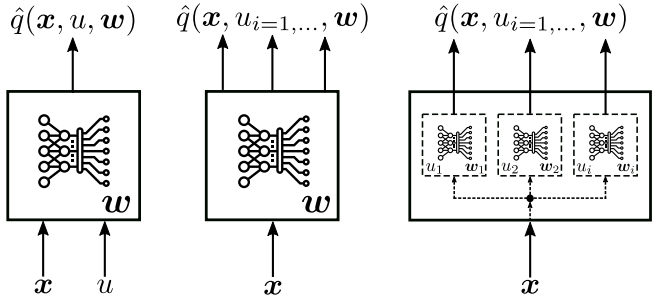


Fig. 10.1: Possible function approximation settings for discrete actions

- Left: one function with both states and actions as input
- Middle: one function with $i = 1, 2, \dots$ outputs covering the action space (e.g., ANN with appropriate output layer)
- Right: multiple (sub-)functions one for each possible action u_i (e.g., multitude of linear approximators in small action spaces)

Feature engineering

- Also for action-value estimation a proper feature engineering (FE) is of vital importance.
- Compared to the state-value prediction, the action becomes part of the FE processing:

$$\hat{q}(\boldsymbol{x}, u, \boldsymbol{w}) = \hat{q}\left(\boldsymbol{f}\left(\boldsymbol{x}, u\right), \boldsymbol{w}\right). \tag{10.3}$$

- Above, $\boldsymbol{f}(\boldsymbol{x}, u) \in \mathbb{R}^{\kappa}$ is the FE function.
- For sake of notation simplicity we write $\hat{q}(\boldsymbol{x}, u, \boldsymbol{w})$ and understand that FE has already been considered (i.e., is a part of $\hat{q}$).

Table of contents

Gradient-based action-value learning

- Transferring the objective (9.3) from on-policy prediction to control yields:

$$J(\boldsymbol{w}) = \sum_k \left[q_\pi(\boldsymbol{x}_k, u_k) - \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w})\right]^2. \tag{10.4}$$

- Analogous, the (semi-)gradient-based parameter update from (9.7) is also applied to action values:

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \alpha \left[q_\pi(\boldsymbol{x}_k, u_k) - \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w}_k)\right] \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w}_k). \tag{10.5}$$

- Depending on the control approach, the true target $q_\pi(\boldsymbol{x}_k, u_k)$ is approximated by:
  - Monte Carlo: full episodic return $q_\pi(\boldsymbol{x}_k, u_k) \approx g$,
  - SARSA: one-step bootstrapped estimate $q_\pi(\boldsymbol{x}_k, u_k) \approx r_{k+1} + \gamma \hat{q}(\boldsymbol{x}_{k+1}, u_{k+1}, \boldsymbol{w}_k)$,
  - n-step SARSA: $q_\pi(\boldsymbol{x}_k, u_k) \approx r_{k+1} + \gamma r_{k+2} + \cdots + \gamma^{n-1} r_{k+n} + \gamma^n \hat{q}(\boldsymbol{x}_{k+n}, u_{k+n}, \boldsymbol{w}_{k+n-1})$.

Houston: we have a problem

- Recall tabular policy improvement theorem (Theo. 3.1): guarantee to find a globally better or equally good policy in each update step.
- With parameter updates (10.5) generalization applies.
- Hence, when reacting to one specific state-action transition other parts of the state-action space within $\hat{q}$ are affected too.

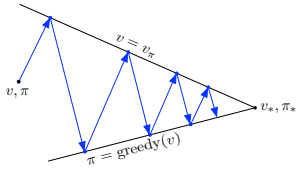


Fig. 10.2: GPI

Loss of policy improvement theorem

- Is not applicable with function approximation!
- We may improve and impair the policy at the same time!

Algorithmic implementation: gradient MC control

- Direct transfer from tabular case to function approximation
- Update target becomes the sampled return $q_\pi(\boldsymbol{x}_k, u_k) \approx g_k$
- If operating ε-greedy on $\hat{q}$: baseline policy (given by $\boldsymbol{w}_0$) must (successfully) terminate the episode!

input: a differentiable function $\hat{q} : \mathbb{R}^\kappa \times \mathbb{R}^\zeta \to \mathbb{R}$
input: a policy π (only if estimating q_π)
parameter: step size $\alpha \in \{\mathbb{R}|0 < \alpha < 1\}$, $\varepsilon \in \{\mathbb{R}|0 < \varepsilon << 1\}$
init: parameter vector $\boldsymbol{w} \in \mathbb{R}^\zeta$ arbitrarily
for $j = 1, 2, \ldots,$ *episodes* **do**
  generate episode following π or ε-greedy on $\hat{q}$: $x_0, u_0, r_1, \ldots, x_T$;
  calculate every-visit return g_k;
  for $k = 0, 1, \ldots, T-1$ *time steps* **do**
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[g_k - \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w})\right] \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w})$;

Algo. 10.1: Every-visit gradient MC-based action-value estimation (output: parameter vector $\boldsymbol{w}$ for $\hat{q}_\pi$ or $\hat{q}^*$)

Algorithmic implementation: semi-gradient SARSA

input: a differentiable function $\hat{q} : \mathbb{R}^{\kappa} \times \mathbb{R}^{\zeta} \to \mathbb{R}$
input: a policy π (only if estimating q_π)
parameter: step size $\alpha \in \{\mathbb{R}|0 < \alpha < 1\}$, $\varepsilon \in \{\mathbb{R}|0 < \varepsilon << 1\}$
init: parameter vector $\boldsymbol{w} \in \mathbb{R}^{\zeta}$ arbitrarily
for $j = 1, 2, \ldots$ *episodes* **do**
    initialize $\boldsymbol{x}_0$;
    for $k = 0, 1, 2 \ldots$ *time steps* **do**
        $u_k \leftarrow$ apply action from $\pi(\boldsymbol{x}_k)$ or ε-greedy on $\hat{q}(\boldsymbol{x}_k, \cdot, \boldsymbol{w})$;
        observe $\boldsymbol{x}_{k+1}$ and r_{k+1};
        if $\boldsymbol{x}_{k+1}$ *is terminal* **then**
            $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[r_{k+1} - \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w})\right] \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w})$;
            go to next episode;
        choose u' from $\pi(\boldsymbol{x}_{k+1})$ or ε-greedy on $\hat{q}(\boldsymbol{x}_{k+1}, \cdot, \boldsymbol{w})$;
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[r_{k+1} + \gamma \hat{q}(\boldsymbol{x}_{k+1}, u', \boldsymbol{w}) - \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w})\right] \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w})$;

Algo. 10.2: Semi-gradient SARSA action-value estimation (output: parameter vector $\boldsymbol{w}$ for $\hat{q}_\pi$ or $\hat{q}^*$)

SARSA application example: mountain car (1)

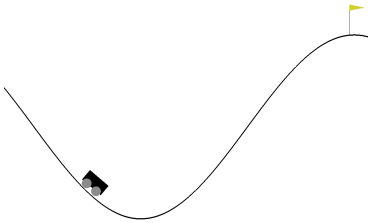


Fig. 10.3: Classic RL control example: mountain car (derivative work based on https://github.com/openai/gym, MIT license)

- Two cont. states: position, velocity
- One discrete action: acceleration given by $\{\text{left}, \text{none}, \text{right}\}$
- $r_k = -1$, i.e., goal is to terminate episode as quick as possible
- Episode terminates when car reaches the flag (or max steps)
- Simplified longitudinal car physics with state constraints
- Position initialized randomly within valley, zero initial velocity
- Car is underpowered and requires swing-up

SARSA application example: mountain car (2)

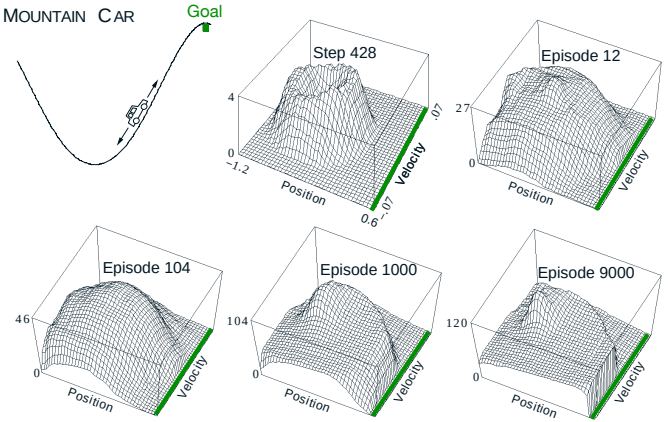


Fig. 10.4: Cost-to-go function $-\max_u \hat{q}(\boldsymbol{x}, u, \boldsymbol{w})$ for mountain car task using linear approximation with SARSA and tile coding (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Tile coding

- Problem space is grouped into (overlapping) partitions / tiles.
- Performs a discretization of the problem space.
- Function approximation serves as interpolation between tiles.
- Find an example here: https://github.com/MeepMoop/tilecoding .

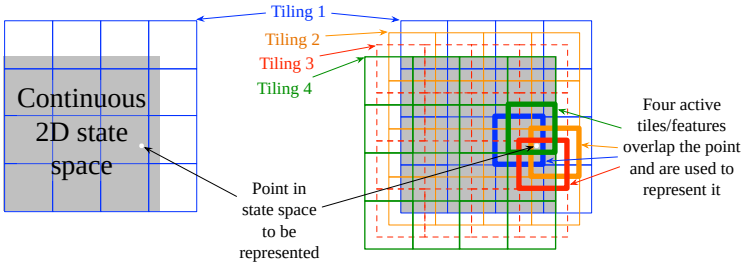


Fig. 10.5: Tile coding example in 2D (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

input: a differentiable function $\hat{q} : \mathbb{R}^{\kappa} \times \mathbb{R}^{\zeta} \to \mathbb{R}$
input: a policy π (only if estimating q_{π})
parameter: $\alpha \in \{\mathbb{R} | 0 < \alpha < 1\}$, $\varepsilon \in \{\mathbb{R} | 0 < \varepsilon << 1\}$, $n \in \mathbb{Z}^{+}$
init: parameter vector $\boldsymbol{w} \in \mathbb{R}^{\zeta}$ arbitrarily
for $j = 1, 2 \ldots$ *episodes* **do**
    initialize and store $\boldsymbol{x}_0$;
    select and store $u_0 \sim \pi(\boldsymbol{x}_0)$ or ε-greedy w.r.t. $\hat{q}(\boldsymbol{x}_0, \cdot, \boldsymbol{w})$;
    $T \leftarrow \infty$;
    repeat $k = 0, 1, 2, \ldots$
        if $k < T$ **then**
            take action u_k observe and store $\boldsymbol{x}_{k+1}$ and r_{k+1};
            if $\boldsymbol{x}_{k+1}$ *is terminal* **then** $T \leftarrow k + 1$;
            else select & store $u_{k+1} \sim \pi(\boldsymbol{x}_{k+1})$ or ε-greedy w.r.t. $\hat{q}(\boldsymbol{x}_{k+1}, \cdot, \boldsymbol{w})$;
        $\tau \leftarrow k - n + 1$ (τ time index for estimate update);
        if $\tau \geq 0$ **then**
            $g \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} r_i$;
            if $\tau + n < T$: $g \leftarrow g + \gamma^n \hat{q}(\boldsymbol{x}_{\tau+n}, u_{\tau+n}, \boldsymbol{w})$;
            $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[g - \hat{q}(\boldsymbol{x}_{\tau}, u_{\tau}, \boldsymbol{w}) \right] \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{x}_{\tau}, u_{\tau}, , \boldsymbol{w})$;
    until $\tau = T - 1$;

Algo. 10.3: n-step semi-gradient SARSA (output: parameter vector $\boldsymbol{w}$ for $\hat{q}_{\pi}$ or $\hat{q}^{*}$)

Table of contents

Transferring LSTD-style batch learning to action values

- In the previous lecture we developed a closed-form batch learning tool: LSTD.
  - Linear function approximation.
  - Fixed, representative data set $\mathcal{D}$.
- Same idea can be transferred to action values when bootstrapping with one-step Sarsa, called LS-SARSA (or sometimes LSTDQ):

$$\begin{aligned} q_\pi(\boldsymbol{x}_k, u_k) &\approx r_{k+1} + \gamma \hat{q}(\boldsymbol{x}_{k+1}, u_{k+1}, \boldsymbol{w}_k), \\ \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w}_k) &= \hat{q}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w}_k) = \tilde{\boldsymbol{x}}_k^\mathsf{T} \boldsymbol{w}_k. \end{aligned} \tag{10.6}$$

- The cost function for action-value prediction is then:

$$J(\boldsymbol{w}) = \sum_k \left[r_{k+1} - \left(\tilde{\boldsymbol{x}}_k^\mathsf{T} - \gamma \tilde{\boldsymbol{x}}_{k+1}^\mathsf{T} \right) \boldsymbol{w} \right]^2. \tag{10.7}$$

- Hence, the closed-form least squares solution for the action values is the same as for the state value case but the feature vector depends also on the actions:

$$\tilde{\boldsymbol{x}}_k = \boldsymbol{f}(\boldsymbol{x}_k, u_k).$$

On and off-policy LS-SARSA

With b samples we can form a target vector $\boldsymbol{y}$ and regressor matrix $\boldsymbol{\Xi}$:

$$\boldsymbol{y} = \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_b \end{bmatrix}, \quad \boldsymbol{\Xi} = \begin{bmatrix} \left(\tilde{\boldsymbol{x}}_0^\mathsf{T} - \gamma \tilde{\boldsymbol{x}}_1^\mathsf{T}\right) \\ \left(\tilde{\boldsymbol{x}}_1^\mathsf{T} - \gamma \tilde{\boldsymbol{x}}_2^\mathsf{T}\right) \\ \vdots \\ \left(\tilde{\boldsymbol{x}}_{b-1}^\mathsf{T} - \gamma \tilde{\boldsymbol{x}}_b^\mathsf{T}\right) \end{bmatrix}. \tag{10.8}$$

Regarding the data input to $\boldsymbol{\Xi}$ we can distinguish two cases: The actions u_k and u_{k+1} in the feature pair $\left(\tilde{\boldsymbol{x}}_k^\mathsf{T} - \gamma \tilde{\boldsymbol{x}}_{k+1}^\mathsf{T}\right)$ per row in $\boldsymbol{\Xi}$ either descends from the

- same policy π (on-policy learning) or
- the action u_{k+1} in $\tilde{\boldsymbol{x}}_{k+1} = \boldsymbol{f}(\boldsymbol{x}_{k+1}, u_{k+1})$ is chosen based on an arbitrary policy π' (off-policy learning).

If we apply off-policy LS-SARSA then

- we retrieve the flexibility to collect training samples arbitrarily
- at the cost of an estimation bias based on the sampling distribution.

LS-SARSA

LS-SARSA solution

Having arranged $i = 1, \ldots, b$ samples $\langle \boldsymbol{x}_i, u_i, r_{i+1}, \boldsymbol{x}_{i+1}, u_{i+1} \rangle \sim \mathcal{D}$ using one-step bootstrapping (10.6) and linear function approximation as in (10.8), the LS-SARSA solution is

$$\boldsymbol{w}^* = (\boldsymbol{\Xi}^\mathsf{T} \boldsymbol{\Xi})^{-1} \boldsymbol{\Xi}^\mathsf{T} \boldsymbol{y}. \tag{10.9}$$

Again, basic usage distinction:

- If $\{u_i, u_{i+1}\} \sim \pi$: on-policy prediction (as in LSTD)
- If $u_i \sim \pi$ and $u_{i+1} \sim \pi'$: off-policy prediction (useful for control)

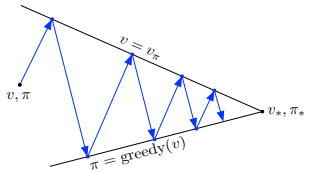
Possible modifications:

- To prevent numeric instability regularization is possible, cf. (9.13)
- Recursive implementation for online usage straightforward, cf. (9.14)

Least squares policy iteration (LSPI)

General idea:

- Apply general policy improvement (GPI) based on data set $\mathcal{D}$,
- Policy evaluation by off-policy LS-SARSA,
- Policy improvement by greedy choices on predicted action values.



Some remarks:

- LSPI is an offline and off-policy control approach.
- Exploration is required by feeding suitable sampling distributions in $\mathcal{D}$:
  - Such as ε-greedy choices based on $\hat{q}$.
  - But also complete random samples are conceivable.

Algorithmic implementation: LSPI

input: a feature representation $\tilde{\boldsymbol{x}}$ with $\tilde{\boldsymbol{x}}_T = 0$ (i.e., $\hat{q}(\tilde{\boldsymbol{x}}_T, \cdot) = 0$)
input: a data set $\langle \boldsymbol{x}_i, u_i, r_{i+1}, \boldsymbol{x}_{i+1} \rangle \sim \boldsymbol{\mathcal{D}}$ with $i = 1, \ldots, b$ samples
parameter: an accuracy threshold $\Delta \in \{\mathbb{R} | 0 < \Delta\}$
init: linear approximation function weights $\boldsymbol{w} \in \mathbb{R}^{\zeta}$ arbitrarily
$\pi \leftarrow \arg\max_u \hat{q}(\cdot, u, \boldsymbol{w})$ (greedy choices based on $\hat{q}(\boldsymbol{w})$);
repeat
    $\boldsymbol{w}' \leftarrow \boldsymbol{w}$;
    $\boldsymbol{w} \leftarrow$ LS-SARSA$(\boldsymbol{\mathcal{D}}, u_{i+1} \sim \pi)$;
    $\pi \leftarrow \arg\max_u \hat{q}(\cdot, u, \boldsymbol{w})$;
until $||\boldsymbol{w}' - \boldsymbol{w}|| < \Delta$;

Algo. 10.4: Least squares policy iteration (output: $\boldsymbol{w}$ for $\hat{q}^*$)

- In a (small) discrete action space the $\arg\max_u$ operation is straightforward.
- After one full LSPI evaluation the data set $\boldsymbol{\mathcal{D}}$ might be altered to include new data obtained based on the updated $\boldsymbol{w}$ vector.
- Source: M. Lagoudakis and R. Parr, *Least-Squares Policy Iteration*, Journal of Machine Learning Research 4, pp. 1107-1149, 2003

LSPI application example: inverted pendulum (1)

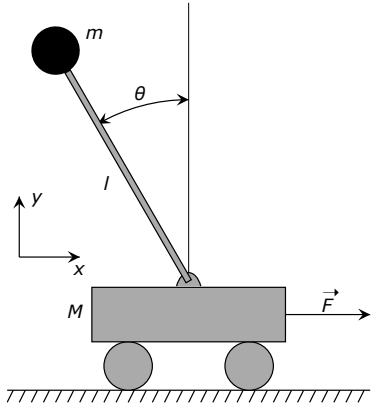


Fig. 10.6: Classic RL control example: inverted pendulum (source: www.wikipedia.org, CC0 1.0)

- Two continuous states: angular position θ and velocity $\dot{\theta}$
- One discrete action: acceleration force (i.e., torque at shaft)
- Action noise as disturbance
- Non-linear system dynamics
- State initialization randomly close to upper equilibrium
- $r_k = 0$ if pendulum is above horizontal line
- $r_k = -1$ if below horizontal line and episode terminates
- $\gamma = 0.95$

LSPI application example: inverted pendulum (2)

- Initial training samples for $\mathcal{D}$ following a policy selecting actions at uniform probability
- Additional samples have been manually added during the training
- Radial basis function as feature engineering

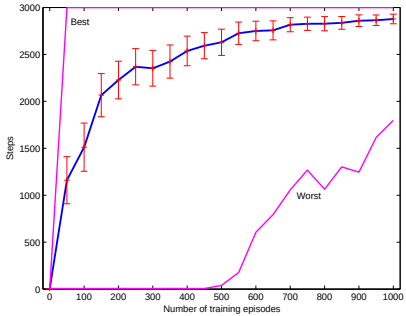


Fig. 10.7: Balancing steps before episode termination with a clipping of maximum 3000 steps (source: M. Lagoudakis and R. Parr, *Least-Squares Policy Iteration*, Journal of Machine Learning Research 4, pp. 1107-1149, 2003)

Algorithmic implementation: online LSPI

input: a feature representation $\tilde{\boldsymbol{x}}$ with $\tilde{\boldsymbol{x}}_T = 0$ (i.e., $\hat{q}(\tilde{\boldsymbol{x}}_T, \cdot, \cdot) = 0$)
parameter: forgetting factor $\lambda \in \{\mathbb{R} | 0 < \lambda \leq 1\}$, $\varepsilon \in \{\mathbb{R} | 0 < \varepsilon << 1\}$, update factor $k_w \in \{\mathbb{N} | 1 \leq k_w\}$
init: weights $\boldsymbol{w} \in \mathbb{R}^\zeta$ arbitrarily, policy π being ε-greedy w.r.t. $\hat{q}(\boldsymbol{w})$, covariance $\boldsymbol{P} > 0$
for $j = 1, 2, \ldots$ *episodes* **do**
  initialize $\boldsymbol{x}_0$ and set $u_0 \sim \pi(\boldsymbol{x}_0)$;
  for $k = 0, 1, 2 \ldots$ *time steps* **do**
    apply action u_k, observe $\boldsymbol{x}_{k+1}$ and r_{k+1}, set $u_{k+1} \sim \pi(\boldsymbol{x}_{k+1})$;
    $y \leftarrow r_{k+1}$;
    $\boldsymbol{\xi}^\mathsf{T} \leftarrow \tilde{\boldsymbol{x}}_k^\mathsf{T}(\boldsymbol{x}_k, u_k) - \gamma \tilde{\boldsymbol{x}}_{k+1}^\mathsf{T}(\boldsymbol{x}_{k+1}, u_{k+1})$;
    $\boldsymbol{c} \leftarrow (\boldsymbol{P}\boldsymbol{\xi}) \,/\, \left(\lambda + \boldsymbol{\xi}^\mathsf{T}\boldsymbol{P}\boldsymbol{\xi}\right)$;
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \boldsymbol{c}\left(y - \boldsymbol{\xi}^\mathsf{T}\boldsymbol{w}\right)$;
    $\boldsymbol{P} \leftarrow \left(\boldsymbol{I} - \boldsymbol{c}\boldsymbol{\xi}^\mathsf{T}\right)\boldsymbol{P}/\lambda$;
    if $k \mod k_w = 0$ **then**
      $\pi \leftarrow \varepsilon$-greedy w.r.t. $\hat{q} = \tilde{\boldsymbol{x}}^\mathsf{T}(\boldsymbol{x}, u)\boldsymbol{w}$;
    exit loop if $\boldsymbol{x}_{k+1}$ is terminal;

Algo. 10.5: Online LSPI with RLS-SARSA (output: $\boldsymbol{w}$ for $\hat{q}^*$)

Remarks on online LSPI

- k_w depicts the number of steps between policy improvement cycles.
- Forgetting factor λ and k_w require mutual tuning:
  - After each policy improvement the policy evaluation requires sample updates to accurately predict the altered policy.
  - Numerically instability may occur for $\lambda < 1$ and requires regularization.
- Hence, the algorithm is online-capable but its policy is normally not updated in a step-by-step fashion.
- Alternative online LSPI with OLS-SARSA can be found in L. Buşoniu et al., *Online least-squares policy iteration for reinforcement learning control*, American Control Conference, 2010.

Table of contents

General background on DQN

- Recall incremental learning step from tabular Q-learning:

$$\hat{q}(x,u) \leftarrow \hat{q}(x,u) + \alpha \left[r + \gamma \max_u \hat{q}(x',u) - \hat{q}(x,u) \right].$$

- Deep Q-networks (DQN) transfer this to an approximate solution:

$$\boldsymbol{w} = \boldsymbol{w} + \alpha \left[r + \gamma \max_u \hat{q}(\boldsymbol{x}',u,\boldsymbol{w}) - \hat{q}(\boldsymbol{x},u,\boldsymbol{w}) \right] \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{x},u,\boldsymbol{w}). \tag{10.10}$$

However, instead of using above semi-gradient step-by-step updates, DQN is characterized by

- an experience replay buffer for batch learning (cf. prev. lectures),
- a separate set of weights $\boldsymbol{w}^-$ for the bootstrapped Q-target.

Motivation behind:

- Efficiently use available data (experience replay).
- Stabilize learning by trying to make targets and feature inputs more like i.i.d. data from a stationary process (prevent windup of values).

Summary of DQN working principle (1)

- Take actions u based on $\hat{q}(\boldsymbol{x}, u, \boldsymbol{w})$ (e.g., ε-greedy).
- Store observed tuples $\langle \boldsymbol{x}, u, r, \boldsymbol{x}' \rangle$ in memory buffer $\boldsymbol{\mathcal{D}}$.
- Sample mini-batches $\boldsymbol{\mathcal{D}}_b$ from $\boldsymbol{\mathcal{D}}$.
- Calculate bootstrapped Q-target with a delayed parameter vector $\boldsymbol{w}^-$ (so-called target network):

$$q_\pi(\boldsymbol{x}, u) \approx r + \gamma \max_u \hat{q}(\boldsymbol{x}', u, \boldsymbol{w}^-).$$

- Optimize MSE loss between above targets and the regular approximation $\hat{q}(\boldsymbol{x}, u, \boldsymbol{w})$ using $\boldsymbol{\mathcal{D}}_b$

$$\mathcal{L}(\boldsymbol{w}) = \left[\left(r + \gamma \max_u \hat{q}(\boldsymbol{x}', u, \boldsymbol{w}^-) \right) - \hat{q}(\boldsymbol{x}, u, \boldsymbol{w}) \right]^2_{\boldsymbol{\mathcal{D}}_b}. \tag{10.11}$$

- Update $\boldsymbol{w}^-$ based on $\boldsymbol{w}$ from time to time.

Summary of DQN working principle (2)

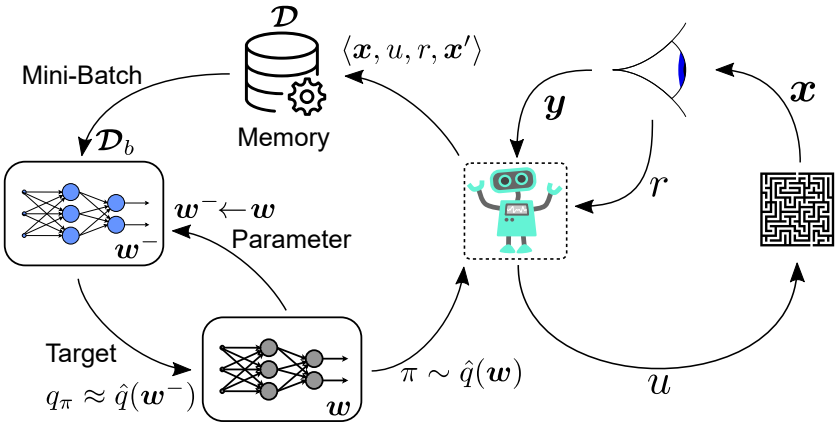


Fig. 10.8: DQN structure from a bird's-eye perspective (derivative work of Fig. 1.1 and wikipedia.org, CC0 1.0)

Algorithmic implementation: DQN

input: a differentiable function $\hat{q} : \mathbb{R}^\kappa \times \mathbb{R}^\zeta \to \mathbb{R}$ (including feature eng.)
parameter: $\varepsilon \in \{\mathbb{R} | 0 < \varepsilon << 1\}$, update factor $k_w \in \{\mathbb{N} | 1 \leq k_w\}$
init: weights $\boldsymbol{w} = \boldsymbol{w}^- \in \mathbb{R}^\zeta$ arbitrarily, memory $\boldsymbol{\mathcal{D}}$ with certain capacity
for $j = 1, 2, \ldots$ *episodes* **do**
    initialize $\boldsymbol{x}_0$;
    for $k = 0, 1, 2 \ldots$ *time steps* **do**
        $u_k \leftarrow$ apply action ε-greedy w.r.t $\hat{q}(\boldsymbol{x}_k, \cdot, \boldsymbol{w})$;
        observe $\boldsymbol{x}_{k+1}$ and r_{k+1};
        store tuple $\langle \boldsymbol{x}_k, u_k, r_{k+1}, \boldsymbol{x}_{k+1} \rangle$ in $\boldsymbol{\mathcal{D}}$;
        sample mini-batch $\boldsymbol{\mathcal{D}}_b$ from $\boldsymbol{\mathcal{D}}$ (after initial memory warmup);
        for $i = 1, \ldots, b$ *samples* **do** calculate Q-targets
            if $\boldsymbol{x}_{i+1}$ *is terminal* **then** $y_i = r_{i+1}$;
            else $y_i = r_{i+1} + \gamma \max_u \hat{q}(\boldsymbol{x}_{i+1}, u, \boldsymbol{w}^-)$;
        fit $\boldsymbol{w}$ on loss $\mathcal{L}(\boldsymbol{w}) = [y_i - \hat{q}(\boldsymbol{x}_i, u_i, \boldsymbol{w})]^2_{\boldsymbol{\mathcal{D}}_b}$;
        if $k \mod k_w = 0$ **then** $\boldsymbol{w}^- \leftarrow \boldsymbol{w}$ (update target weights);

Algo. 10.6: DQN (output: parameter vector $\boldsymbol{w}$ for $\hat{q}^*$)

Remarks on DQN implementation

- General framework is based on V. Mnih et al., *Human-level control through deep reinforcement learning*, Nature, pp. 529-533, 2015.
- Often 'deep' artificial neural networks are used as function approximation for DQN.
  - Nevertheless, other model topologies are fully conceivable.
- The fit of $\boldsymbol{w}$ on loss $\mathcal{L}$ is an intermediate supervised learning step.
  - Comes with degrees of freedom regarding solver choice.
  - Has own optimization parameters which are not depicted here in details (many tuning options).
- Mini-batch sampling from $\mathcal{D}$ is often randomly distributed.
  - Nevertheless, guided sampling with useful distributions for a specific control task can be beneficial (cf. Dyna discussion in 7th lecture).
- Likewise, the simple ε-greedy approach can be extended.
  - Often a scheduled/annealed trajectory ε_k is used.
  - Again referring to the Dyna framework, many more exploration strategies are possible.

DQN application example: Atari games (1)

- End-to-end learning of $\hat{q}(\boldsymbol{x}, u)$ from monitor pixels $\boldsymbol{x}$
- Feature engineering obtains stacking of raw pixes from last 4 frames
- Actions u are 18 possible joystick/button combinations
- Reward is the change of highscore per step
- Interesting lecture from V. Minh with more details: YouTube

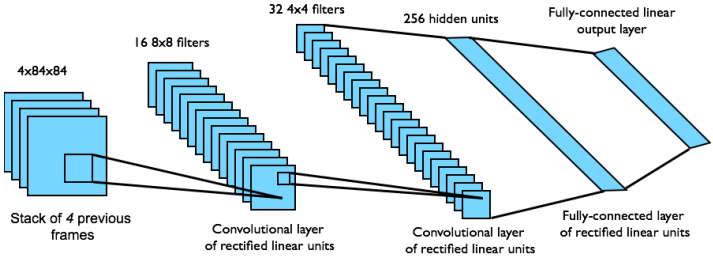


Fig. 10.9: Network architecture overview used for DQN in Atari games (source: D. Silver, Reinforcement learning, 2015. CC BY-NC 4.0)

DQN application example: Atari games (2)

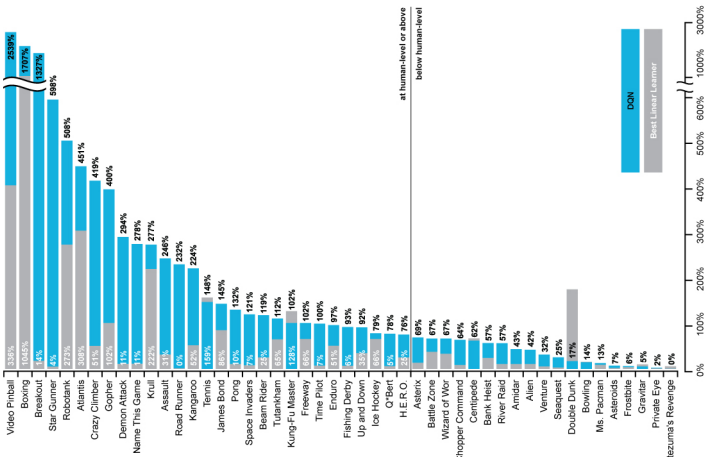


Fig. 10.10: DQN performance results in Atari games against human performance (source: D. Silver, Reinforcement learning, 2015. CC BY-NC 4.0)

Summary: what you've learned today

- From a simplified perspective, the procedures from the approximate prediction can simply be transferred to value-based control.
- On the contrary, the policy improvement theorem no longer applies in the approximate RL case (generalization impact).
  - Control algorithms may diverge completely.
  - Or a performance trade-off between different parts of the problem space could emerge.
- Off-policy batch learning approaches allow for efficient data usage.
  - LSPI uses LS-SARSA on linear function approximation.
  - DQN extends Q-learning on non-linear approximation with additional tweaks (experience replay, target networks,...).
  - However, a prediction bias results (off-policy sampling distribution).