Preface

Problem space: it is further assumed that

- the states $x$ 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

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

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

$$\hat{q}(x, u, w) \approx q^*(x, u). \quad (10.2)$$
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 $i = 1, 2, \ldots$ 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)
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:

\[
\hat{q}(x, u, w) = \hat{q}(f(x, u), w).
\] (10.3)

Above, \( f(x, u) \in \mathbb{R}^\kappa \) is the FE function.

For sake of notation simplicity we write \( \hat{q}(x, u, w) \) and understand that FE has already been considered (i.e., is a part of \( \hat{q} \)).
Table of Contents

1. On-Policy Control With (Semi-)Gradients
2. Average Reward: An Alternative for Continuing Tasks
3. Least Squares Policy Iteration (LSPI)
4. Deep $Q$-Networks (DQN)
Gradient-Based Action-Value Learning

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

\[ J(w) = \sum_k \left[ q_\pi(x_k, u_k) - \hat{q}(x_k, u_k, w) \right]^2. \]  

(10.4)

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

\[ w_{k+1} = w_k + \alpha \left[ q_\pi(x_k, u_k) - \hat{q}(x_k, u_k, w_k) \right] \nabla_w \hat{q}(x_k, u_k, w_k). \]  

(10.5)

- Depending on the control approach, the true target \( q_\pi(x_k, u_k) \) is approximated by:
  - Monte Carlo: full episodic return \( q_\pi(x_k, u_k) \approx g \),
  - Sarsa: one-step bootstrapped estimate \( q_\pi(x_k, u_k) \approx r_{k+1} + \gamma \hat{q}(x_{k+1}, u_{k+1}, w_k) \),
  - \( n \)-step Sarsa: \( q_\pi(x_k, u_k) \approx r_{k+1} + \gamma r_{k+2} + \cdots + \gamma^{n-1} r_{k+n} + \gamma^n \hat{q}(x_{k+n}, u_{k+n}, w_{k+n-1}) \).
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.

![Diagram showing GPI](image)

**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!
The number of games during test with two algorithms is displayed in Figs. 6 and 7. It reflects the convergent trends of Fig. 3. This game expects that the agent should obtain as many scores as possible by saving divers and killing fish. The agent can control the submarine with five basic actions as mentioned above. Once the submarine saves the diver or kills the fish, the agent gets 20 and 40 points. If the submarine runs into the images of the two games. The CNN contains 3 convolution layers and two full connected layers. All the settings in these experiments and results.

**Fig. 10.3:** Learning curves with drastic performance dips when applying Sarsa with function approximation. Left: Atari Breakout, right: Atari Seaquest (source: D. Zhao et al., *Deep reinforcement learning with experience replay based on SARSA*, IEEE Symposium Series on Computational Intelligence, 2016)
Algorithmic Implementation: Gradient MC Control

- Direct transfer from tabular case to function approximation
- Update target becomes the sampled return $q_\pi(x_k, u_k) \approx g_k$
- If operating $\varepsilon$-greedy on $\hat{q}$: baseline policy (given by $w_0$) must (successfully) terminate the episode!

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

**Input:** a differentiable function $\hat{q} : \mathbb{R}^\kappa \times \mathbb{R}^\zeta \to \mathbb{R}$

**Input:** a policy $\pi$ (only if estimating $q_\pi$)

**Parameter:** step size $\alpha \in \{\mathbb{R} | 0 < \alpha < 1\}$, $\varepsilon \in \{\mathbb{R} | 0 < \varepsilon << 1\}$

**Init:** parameter vector $w \in \mathbb{R}^\zeta$ arbitrarily

**For** $j = 1, 2, \ldots$, episodes **do**

- generate episode following $\pi$ or $\varepsilon$-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**

- $w \leftarrow w + \alpha \left[ g_k - \hat{q}(x_k, u_k, w) \right] \nabla_w \hat{q}(x_k, u_k, w)$;
Algorithmic Implementation: Semi-Gradient Sarsa

**input:** a differentiable function $\hat{q} : \mathbb{R}^\kappa \times \mathbb{R}^\zeta \rightarrow \mathbb{R}$

**input:** a policy $\pi$ (only if estimating $q_\pi$)

**parameter:** step size $\alpha \in \{\mathbb{R} | 0 < \alpha < 1\}$, $\varepsilon \in \{\mathbb{R} | 0 < \varepsilon << 1\}$

**init:** parameter vector $w \in \mathbb{R}^\zeta$ arbitrarily

for $j = 1, 2, \ldots$ episodes do

initialize $x_0$;

for $k = 0, 1, 2 \ldots$ time steps do

$u_k \leftarrow$ apply action from $\pi(x_k)$ or $\varepsilon$-greedy on $\hat{q}(x_k, \cdot, w)$;

observe $x_{k+1}$ and $r_{k+1}$;

if $x_{k+1}$ is terminal then

$w \leftarrow w + \alpha \left[ r_{k+1} - \hat{q}(x_k, u_k, w) \right] \nabla_w \hat{q}(x_k, u_k, w)$;

go to next episode;

choose $u'$ from $\pi(x_{k+1})$ or $\varepsilon$-greedy on $\hat{q}(x_{k+1}, \cdot, w)$;

$w \leftarrow w + \alpha \left[ r_{k+1} + \gamma \hat{q}(x_{k+1}, u', w) - \hat{q}(x_k, u_k, w) \right] \nabla_w \hat{q}(x_k, u_k, w)$;

end if

end for

end for

Algo. 10.2: Semi-gradient Sarsa action-value estimation (output: parameter vector $w$ for $\hat{q}_\pi$ or $\hat{q}^*$)
Sarsa Application Example: Mountain Car (1)

Two cont. states: position, velocity
One discrete action: acceleration given by \{left, none, 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

Fig. 10.4: Classic RL control example: mountain car (derivative work based on https://github.com/openai/gym, MIT license)
Fig. 10.5: Cost-to-go function $\max_u \hat{q}(x, u, w)$ 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. 10.6: Tile coding example in 2D (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)
The feature vectors $x(s, a)$ created by tile coding were then combined linearly with the parameter vector to approximate the action-value function:

$$
\hat{q}(s, a, w) = w^T x(s, a) = \sum_{i=1}^{d} w_i \cdot x_i(s, a),
$$

for each pair of state, $s$, and action, $a$.

Figure 10.1 shows what typically happens while learning to solve this task with this form of function approximation. Shown is the negative of the value function (the cost-to-go function) learned on a single run. The initial action values were all zero, which was optimistic (all true values are negative in this task), causing extensive exploration to occur even though the exploration parameter, $\epsilon$, was 0. This can be seen in the middle-top panel of the figure, labeled "Step 428". At this time not even one episode had been completed, but the car has oscillated back and forth in the valley, following circular trajectories in state space. All the states visited frequently are valued worse than unexplored states, because the actual rewards have been worse than what was (unrealistically) expected. This continually drives the agent away from wherever it has been, to explore new states, until a solution is found.

Figure 10.2 shows several learning curves for semi-gradient Sarsa on this problem, with various step sizes.

![Mountain Car Learning Curves](source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)
**input:** a differentiable function $\hat{q} : \mathbb{R}^κ × \mathbb{R}^ζ \to \mathbb{R}$

**input:** a policy $\pi$ (only if estimating $q_\pi$)

**parameter:** $\alpha \in \mathbb{R} | 0 < \alpha < 1$, $\varepsilon \in \mathbb{R} | 0 < \varepsilon << 1$, $n \in \mathbb{Z}^+$

**init:** parameter vector $\mathbf{w} \in \mathbb{R}^ζ$ arbitrarily

**for** $j = 1, 2 \ldots$ **episodes** **do**

  **initialize and store** $\mathbf{x}_0$;
  **select and store** $\mathbf{u}_0 \sim \pi(\mathbf{x}_0)$ or $\varepsilon$-greedy w.r.t. $\hat{q}(\mathbf{x}_0, \cdot, \mathbf{w})$;
  $T \leftarrow \infty$;

  **repeat** $k = 0, 1, 2, \ldots$

  **if** $k < T$ **then**
  
  take action $\mathbf{u}_k$ observe and store $\mathbf{x}_{k+1}$ and $r_{k+1}$;
  
  **if** $\mathbf{x}_{k+1}$ **is terminal** **then** $T \leftarrow k + 1$;
  
  **else** select & store $\mathbf{u}_{k+1} \sim \pi(\mathbf{x}_{k+1})$ or $\varepsilon$-greedy w.r.t. $\hat{q}(\mathbf{x}_{k+1}, \cdot, \mathbf{w})$;

  $\tau \leftarrow k - n + 1$ ($\tau$ 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 $\tau + n < T$: $g \leftarrow g + \gamma^n \hat{q}(\mathbf{x}_{\tau+n}, \mathbf{u}_{\tau+n}, \mathbf{w})$;

  $\mathbf{w} \leftarrow \mathbf{w} + \alpha [g - \hat{q}(\mathbf{x}_\tau, \mathbf{u}_\tau, \mathbf{w})] \nabla \hat{q}(\mathbf{x}_\tau, \mathbf{u}_\tau, \mathbf{w})$;

  **until** $\tau = T - 1$;

**Algo. 10.3:** $n$-step semi-gradient Sarsa (output: parameter vector $\mathbf{w}$ for $\hat{q}_\pi$ or $\hat{q}^*$)
Fig. 10.4: Effect of the $n$ and $\alpha$ on early performance of $n$-step semi-gradient Sarsa and tile-coding function approximation on the Mountain Car task. As usual, an intermediate level of bootstrapping ($n = 4$) performed best. These results are for selected $\alpha$ values, on a log scale, and then connected by straight lines. The standard errors ranged from 0.5 (less than the line width) for $n = 1$ to about 4 for $n = 16$, so the main effects are all statistically significant.

Exercise 10.1: We have not explicitly considered or given pseudocode for any Monte Carlo methods or in this chapter. What would they be like? Why is it reasonable not to give pseudocode for them? How would they perform on the Mountain Car task? 

Exercise 10.2: Give pseudocode for semi-gradient one-step Expected Sarsa for control.

Exercise 10.3: Why do the results shown in Figure 10.4 have higher standard errors at large $n$ than at small $n$?
Mountain Car: Early Performance Evaluation

Fig. 10.9: Effect of learning rate $\alpha$ and step variable $n$ on early performance of $n$-step semi-gradient Sarsa and tile coding function approximation on the mountain car task. Abscissa is scaled with the number of used tiles, here equal to eight (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)
Table of Contents

1. On-Policy Control With (Semi-)Gradients

2. Average Reward: An Alternative for Continuing Tasks

3. Least Squares Policy Iteration (LSPI)

4. Deep Q-Networks (DQN)
The Average Reward Definition

- Is an alternative performance metric in continuing tasks.
  - Recall: discounting helped us to limit $v < \infty$.
  - Alternative viewpoint: why value rewards further in the future less?

**Definition 10.1: Average reward**

In continuing tasks the average reward is defined as

\[
\bar{r}_\pi = \lim_{h \to \infty} \frac{1}{h} \sum_{k=1}^{h} \mathbb{E} \left[ R_k \mid X_0, U_{0:k-1} \sim \pi \right],
\]

\[
= \lim_{k \to \infty} \mathbb{E} \left[ R_k \mid X_0, U_{0:k-1} \sim \pi \right],
\]

\[
= \int_{X} \mu_\pi(x) \sum_{u} \pi(u \mid x) \int_{X \times R} p(x', r \mid x, u)r.
\]

for policy $\pi$ with $\mu_\pi$ being the steady-state distribution $\mu_\pi(x) = \lim_{k \to \infty} \mathbb{P} \left[ X_k = x \mid U_{0:k-1} \sim \pi \right]$ (long-term state distribution).
To compare the average reward against the usual discounted reward setting in continuing tasks when utilizing function approximation, we introduce the performance metric

$$J_\pi = \int_\mathcal{X} \mu_\pi(x) v_\pi^\gamma(x).$$  \hfill (10.7)  

Here, $v_\pi^\gamma$ is the usual discounted value and, therefore, (10.7) evaluates the long-term value following $\pi$. Applying the Bellman equation we receive:

$$J_\pi = \int_\mathcal{X} \mu_\pi(x) \sum_u \pi(u|x) \int_{\mathcal{X},\mathcal{R}} p(x', r|x, u) \left[ r + \gamma v_\pi^\gamma(x') \right].$$ \hfill (10.8)  

Inserting the average reward $\bar{r}_\pi$ definition (10.6) produces:

$$J_\pi = \bar{r}_\pi + \int_\mathcal{X} \mu_\pi(x) \sum_u \pi(u|x) \int_{\mathcal{X},\mathcal{R}} p(x', r|x, u) \gamma v_\pi^\gamma(x').$$ \hfill (10.9)
Equivalence in Terms of Optimality (2)

In the last equation (10.9) the term

$$\int_{\mathcal{X}} \mu_\pi(x) \sum_u \pi(u|x) \int_{\mathcal{X}, \mathcal{R}} p(x', r|x, u)$$

combines the steady-state distribution $\mu_\pi(x)$ with the likelihood of transitioning to $x'$ and receiving $r$ when taking an action $u \sim \pi(x)$. If one assumes that the long-term distribution $\mu_\pi(x)$ is independent from the starting state $x_0$, i.e., the underlying MDP is an ergodic process, then

$$\int_{\mathcal{X}} \mu_\pi(x) \sum_u \pi(u|x) \int_{\mathcal{X}, \mathcal{R}} p(x', r|x, u) \gamma v_\pi^\gamma(x') = \gamma \int_{\mathcal{X}} v_\pi^\gamma(x') \mu_\pi(x')$$

(10.10)

holds since it does not make any difference if we start from $x$ or $x'$. 
Equivalence in Terms of Optimality (3)

Inserting (10.10) into (10.9) yields:

\[
J_\pi = \bar{r}_\pi + \gamma \int_X \nu_\pi^\gamma(x') \mu_\pi(x'),
\]

\[
= \bar{r}_\pi + \gamma J_\pi,
\]

\[
= \bar{r}_\pi + \gamma \bar{r}_\pi + \gamma^2 J_\pi,
\]

\[
= \ldots
\]

\[
= \frac{1}{1 - \gamma} \bar{r}_\pi.
\]

(10.11)

Take away observations:

- The on-policy discounted value equals the scaled discounted average reward (for on-policy continuing task methods).
- Hence, ordering of all policies based on \( \bar{r}_\pi \) would be exactly the same as using the discounted value \( \nu_\pi^\gamma \).
- Discount factor \( \gamma \) changes from a problem to a solution parameter.
Introducing the Differential Return

► Since discounting is not used in the average reward definition we have to reformulate the return and value definitions.

► Intuitive approach: introduce differential quantities describing the deviations towards $r_{\pi}$.

► Differential return:

$$G_k = R_{k+1} - \bar{r}_{\pi} + R_{k+2} - \bar{r}_{\pi} + R_{k+3} - \bar{r}_{\pi} + \ldots$$  \hspace{1cm} (10.12)

► Differential value functions:

$$v_\pi(x) = \mathbb{E}_\pi [G_k | X_k = x],$$

$$q_\pi(x, u) = \mathbb{E}_\pi [G_k | X_k = x, U_k = u].$$  \hspace{1cm} (10.13)

► Accordingly, the Bellman equations require adaption (cf. chapter 10.3 in lecture book of Barto/Sutton, 2018).
Integrating the Average Reward into RL Solutions

Based on the differential return (10.12) we can reintroduce the TD error in combination with function approximation:

\[
\delta_k = r_{k+1} - \hat{r}_k + \hat{v}(x_{k+1}, w_k) - \hat{v}(x_k, w_k),
\]

\[
\delta_k = r_{k+1} - \hat{r}_k + \hat{q}(x_{k+1}, u_{k+1}, w_k) - \hat{q}(x_k, u_k, w_k).
\]

Here, \(\hat{r}_k\) is an estimate of \(\bar{r}_\pi\) at time step \(k\) (e.g., by a moving average filter).

And finally the semi-gradient-based parameter update is:

\[
\begin{align*}
  w_{k+1} &= w_k + \alpha \delta_k \nabla_w \hat{v}(x_k, w_k), \\
  w_{k+1} &= w_k + \alpha \delta_k \nabla_w \hat{q}(x_k, u_k, w_k).
\end{align*}
\]

In the following we provide details on the differential Sarsa implementation, but nevertheless, the TD modifications for state-value prediction are straightforward.
▶ For incremental average estimation of $\bar{r}_\pi$ we introduce $\beta$.
▶ Similar to the discussion around $\gamma$, the averaging step size $\beta$ is also a solution parameter of the RL algorithm.

**Input:** a differentiable function $\hat{q} : \mathbb{R}^\kappa \times \mathbb{R}^\zeta \rightarrow \mathbb{R}$

**Input:** a policy $\pi$ (only if estimating $q_\pi$)

**Parameter:** step sizes $\{\alpha, \beta\} \in \{\mathbb{R}|0 < \alpha, \beta < 1\}$, $\varepsilon \in \{\mathbb{R}|0 < \varepsilon << 1\}$

**Init:** parameter vector $\mathbf{w} \in \mathbb{R}^\zeta$ arbitrarily, average return estimate $\hat{r} \in \mathbb{R}$

**Init:** $x_0$ and $u_0$ from $\pi(x_0)$ or $\varepsilon$-greedy on $\hat{q}(x_0, \cdot, \mathbf{w})$

**For $k = 0, 1, 2 \ldots$ time steps do**

- apply $u_k$ and observe $x_{k+1}$ and $r_{k+1}$;
- choose $u_{k+1}$ from $\pi(x_{k+1})$ or $\varepsilon$-greedy on $\hat{q}(x_{k+1}, \cdot, \mathbf{w})$;
- $\delta \leftarrow r_{k+1} - \hat{r} + \hat{q}(x_{k+1}, u_{k+1}, \mathbf{w}) - \hat{q}(x_k, u_k, \mathbf{w})$;
- $\hat{r} \leftarrow \hat{r} + \beta \delta$;
- $\mathbf{w} \leftarrow \mathbf{w} + \alpha \delta \nabla_{\mathbf{w}} \hat{q}(x_k, u_k, \mathbf{w})$;

**Algo. 10.4: Differential semi-gradient Sarsa action-value estimation**

(output: parameter vector $\mathbf{w}$ for $\hat{q}_\pi$ or $\hat{q}^*$)
Remarks on Differential Semi-Gradient Sarsa

What is the key motivation to use the differential returns and not the discounted returns in continuing tasks?

- If one is interested in the long-term control behavior setting the discount close to one ($\gamma \to 1$) is required.
- Depending on the application (reward feedback), the estimated values might become unfeasibly large numbers.
- Even if the numeric stability can be maintained, learning might become slow.
  - Recall again: discounting in continuing tasks is numerically limiting $u$.
  - Hence we have to face a trade off: numeric stability vs. long-term estimation capabilities.
- Average rewards can be considered numerically more robust.

Moreover, $n$-step bootstrapping can be implemented as well:

$$G_{k:k+n} = r_{k+1} - \hat{r}_{k+n-1} + \cdots + r_{n+1} - \hat{r}_{k+n-1} + \hat{q}(x_{k+n}, u_{k+n}, w_{k+n-1}),$$
$$\delta_k = G_{k:k+n} - \hat{q}(x_k, u_k, w_k).$$
Algorithmic Impl.: Differential $n$-Step Semi-Gradient Sarsa

**input:** a differentiable function $\hat{q} : \mathbb{R}^\kappa \times \mathbb{R}^\zeta \rightarrow \mathbb{R}$

**input:** a policy $\pi$ (only if estimating $q_\pi$)

**parameter:** $\{\alpha, \beta\} \in \{\mathbb{R}|0 < \alpha, \beta < 1\}$, $\varepsilon \in \{\mathbb{R}|0 < \varepsilon << 1\}$, $n \in \mathbb{Z}^+$

**init:** parameter vector $w \in \mathbb{R}^\zeta$ arbitrarily, average return estimate $\hat{r} \in \mathbb{R}$

**init:** $x_0$ and $u_0$ from $\pi(x_0)$ or $\varepsilon$-greedy on $\hat{q}(x_0, \cdot, w)$

**for** $k = 0, 1, 2 \ldots$ **time steps** **do**

- apply $u_k$ and observe $x_{k+1}$ and $r_{k+1}$;
- choose $u_{k+1}$ from $\pi(x_{k+1})$ or $\varepsilon$-greedy on $\hat{q}(x_{k+1}, \cdot, w)$;
- $\tau \leftarrow k - n + 1$;
- **if** $\tau \geq 0$ **then**
  - $\delta \leftarrow \sum_{i=\tau+1}^{\tau+n} (r_i - \hat{r}) + \hat{q}(x_{\tau+n}, u_{\tau+n}, w) - \hat{q}(x_{\tau}, u_{\tau}, w)$;
  - $\hat{r} \leftarrow \hat{r} + \beta \delta$;
  - $w \leftarrow w + \alpha \delta \nabla_w \hat{q}(x_{\tau}, u_{\tau}, w)$;

**Algo. 10.5:** Differential $n$-step semi-gradient Sarsa action-value estimation (output: parameter vector $w$ for $\hat{q}_\pi$ or $\hat{q}^*$)
In the previous lecture we developed a closed-form batch learning tool, LSTD, which estimated the state values of a given policy if we use linear function approximation and a fixed, representative data set $\mathcal{D}$ is given.

Same idea can be transferred to action values when bootstrapping with one-step Sarsa – called LS-Sarsa (or sometimes LSTD$Q$):

$$q_{\pi}(x_k, u_k) \approx r_{k+1} + \gamma \hat{q}(x_{k+1}, u_{k+1}, w_k),$$

$$\hat{q}(x_k, u_k, w_k) = \hat{q}(\tilde{x}_k, w_k) = \tilde{x}_k^T w_k.$$  \hspace{1cm} (10.16)

The cost function for action-value prediction is then:

$$J(w) = \sum_k \left[ r_{k+1} - \left( \tilde{x}_k^T \gamma \tilde{x}_{k+1}^T \right) w \right]^2.$$ \hspace{1cm} (10.17)

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:

$$\tilde{x}_k = f(x_k, u_k).$$
On and Off-Policy LS-Sarsa

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

$$
\mathbf{y} = \begin{bmatrix}
br_1 \\
r_2 \\
\vdots \\
r_b
\end{bmatrix}, \quad 
\Xi = \begin{bmatrix}
(\tilde{x}_0^T - \gamma \tilde{x}_1^T) \\
(\tilde{x}_1^T - \gamma \tilde{x}_2^T) \\
\vdots \\
(\tilde{x}_{b-1}^T - \gamma \tilde{x}_b^T)
\end{bmatrix}.
$$ (10.18)

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

- same policy $\pi$ (on-policy learning) or
- the action $u_{k+1}$ in $\tilde{x}_{k+1} = f(x_{k+1}, u_{k+1})$ is chosen based on an arbitrary policy $\pi'$ (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.
Having arranged \( i = 1, \ldots, b \) samples \( \langle x_i, u_i, r_{i+1}, x_{i+1}, u_{i+1} \rangle \sim \mathcal{D} \) using one-step bootstrapping (10.16) and linear function approximation as in (10.18), the LS-Sarsa solution is

\[
\mathbf{w}^* = (\Xi^T \Xi)^{-1} \Xi^T \mathbf{y}.
\]  

(10.19)

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 $D$
- 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 $D$
  - Such as $\varepsilon$-greedy choices based on $\hat{q}$
  - But also complete random samples are conceivable
Algorithmic Implementation: LSPI

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

ié Algo. 10.6: Least squares policy iteration (output: $w$ for $\hat{q}^*$)

\begin{itemize}
    \item In a (small) discrete action space the $\arg \max_u$ operation is straightforward: just compare $\hat{q}$ for all possible actions given the state.
    \item After one full LSPI evaluation the data set $\mathcal{D}$ might be altered to include new data obtained based on the updated $w$ vector.
\end{itemize}
LSPI Application Example: Inverted Pendulum (1)

Fig. 10.10: Classic RL control example: inverted pendulum (source: www.wikipedia.org, CC0 1.0)

- Two continuous states: angular position $\theta$ 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
- $\gamma = 0.95$
Initial training samples for $D$ following a policy selecting actions at uniform probability

Additional samples have been manually added during the training

Radial basis function as feature engineering

Fig. 10.11: 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 $\tilde{x}$ with $\tilde{x}_T = 0$ (i.e., $\hat{q}(\tilde{x}_T, \cdot, \cdot) = 0$)

**parameter:** forgetting factor $\lambda \in \{\mathbb{R} \mid 0 < \lambda \leq 1\}$, $\varepsilon \in \{\mathbb{R} \mid 0 < \varepsilon << 1\}$, update factor $k_w \in \{\mathbb{N} \mid 1 \leq k_w\}$

**init:** weights $w \in \mathbb{R}^\zeta$ arbitrarily, policy $\pi$ being $\varepsilon$-greedy w.r.t. $\hat{q}(w)$, covariance $P > 0$ (e.g., $P = \beta I$)

**for** $j = 1, 2, \ldots$ *episodes* **do**

- initialize $x_0$ and set $u_0 \sim \pi(x_0)$;

  **for** $k = 0, 1, 2 \ldots$ *time steps* **do**

    - apply action $u_k$, observe $x_{k+1}$ and $r_{k+1}$, set $u_{k+1} \sim \pi(x_{k+1})$;
    - $y \leftarrow r_{k+1}$;
    - $\xi^T \leftarrow \tilde{x}_k^T(x_k, u_k) - \gamma \tilde{x}_{k+1}^T(x_{k+1}, u_{k+1})$;
    - $c \leftarrow (P\xi) / (\lambda + \xi^TP\xi)$;
    - $w \leftarrow w + c(y - \xi^Tw)$;
    - $P \leftarrow (I - c\xi^T)P/\lambda$;

  **if** $k \mod k_w = 0$ **then**

    - $\pi \leftarrow \varepsilon$-greedy w.r.t. $\hat{q} = \tilde{x}^T(x, u)w$;
    - exit loop if $x_{k+1}$ is terminal;

**Algo. 10.7:** Online LSPI with RLS-Sarsa (output: $w$ for $\hat{q}^*$)
Remarks on Online LSPI

- $k_w$ depicts the number of steps between policy improvement cycles.
- Forgetting factor $\lambda$ 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.
Online LSPI Application Example: Inverted Pendulum

Fig. 10.12: Inverted pendulum with online LSPI using OLS-Sarsa \((k_w = k_\theta, \text{source: L. Buşoniu et al., } \textit{Online least-squares policy iteration for reinforcement learning control}, \textit{American Control Conference, 2010})\)

- In principle same problem framework as before
- Altered reward:
  \[ r = -x^T N x - m u^2 \]
- \( N = \text{diag}(\begin{bmatrix} 5 & 0.1 \end{bmatrix}) \)
- \( m = 1 \)
- \( x = \begin{bmatrix} \theta & \dot{\theta} \end{bmatrix}^T \)
- \( x = \begin{bmatrix} 0 & 0 \end{bmatrix}^T \) equals pendulum pointing up without movement
Table of Contents

1. On-Policy Control With (Semi-)Gradients
2. Average Reward: An Alternative for Continuing Tasks
3. Least Squares Policy Iteration (LSPI)
4. Deep Q-Networks (DQN)
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:

$$
\omega = \omega + \alpha \left[ r + \gamma \max_u \hat{q}(x', u, \omega) - \hat{q}(x, u, \omega) \right] \nabla_{\omega} \hat{q}(x, u, \omega). \quad (10.20)
$$

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 $\omega^-$ 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 \( \hat{q}(x, u, w) \) (e.g., \( \varepsilon \)-greedy).
- Store observed tuples \( \langle x, u, r, x' \rangle \) in memory buffer \( D \).
- Sample mini-batches \( D_b \) from \( D \).
- Calculate bootstrapped \( Q \)-target with a delayed parameter vector \( w^- \) (so-called target network):
  \[
  q_\pi(x, u) \approx r + \gamma \max_u \hat{q}(x', u, w^-).
  \]
- Optimize MSE loss between above targets and the regular approximation \( \hat{q}(x, u, w) \) using \( D_b \)
  \[
  \mathcal{L}(w) = \left[ \left( r + \gamma \max_u \hat{q}(x', u, w^-) \right) - \hat{q}(x, u, w) \right]^2_{D_b}.
  \]
- Update \( w^- \) based on \( w \) from time to time.
Summary of DQN Working Principle (2)

Fig. 10.13: 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 $\hat{q}: \mathbb{R}^\kappa \times \mathbb{R}^\zeta \rightarrow \mathbb{R}$ (including feature eng.)

**parameter:** $\varepsilon \in \{\mathbb{R}|0 < \varepsilon << 1\}$, update factor $k_w \in \{\mathbb{N}|1 \leq k_w\}$

**init:** weights $w = w^- \in \mathbb{R}^\zeta$ arbitrarily, memory $D$ with certain capacity

for $j = 1, 2, \ldots$ episodes do
  initialize $x_0$;
  for $k = 0, 1, 2\ldots$ time steps do
    $u_k \leftarrow$ apply action $\varepsilon$-greedy w.r.t $\hat{q}(x_k, \cdot, w)$;
    observe $x_{k+1}$ and $r_{k+1}$;
    store tuple $\langle x_k, u_k, r_{k+1}, x_{k+1} \rangle$ in $D$;
  sample mini-batch $D_b$ from $D$ (after initial memory warmup);
  for $i = 1, \ldots, b$ samples do calculate $Q$-targets
    if $x_{i+1}$ is terminal then $y_i = r_{i+1}$;
    else $y_i = r_{i+1} + \gamma \max_u \hat{q}(x_{i+1}, u, w^-)$;
    fit $w$ on loss $L(w) = [y_i - \hat{q}(x_i, u_i, w)]^2_{D_b}$;
  if $k \mod k_w = 0$ then $w^- \leftarrow w$ (update target weights);

Algo. 10.8: DQN (output: parameter vector $w$ for $\hat{q}^*$)
Remarks on DQN Implementation

- Often ’deep’ artificial neural networks are used as function approximation for DQN.
  - Nevertheless, other model topologies are fully conceivable.
- The fit of $w$ on loss $L$ 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 $D$ 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 $\varepsilon$-greedy approach can be extended.
  - Often a scheduled/annealed trajectory $\varepsilon_k$ 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 $\hat{q}(x, u)$ from monitor pixels $x$
- 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. 10.14: Network architecture overview used for DQN in Atari games (source: D. Silver, Reinforcement learning, 2016. CC BY-NC 4.0)
Fig. 10.15: DQN performance results in Atari games against human performance (source: D. Silver, Reinforcement learning, 2016. CC BY-NC 4.0)
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.

Differential-returns allow an interesting alternative MDP formulation for continuing tasks.
- The usual discounted MDP framework may exhibit numerical problems for $\gamma \approx 1$ (slow learning, unfeasible large returns).
- Discounting is not relevant for the optimal policy order.

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).
The End for Today

Thanks for your attention and have a nice week!