Lecture 07: On-Policy Prediction with Function Approximation

Paul Swoboda

Table of Contents

Prediction Framework with Function Approximation (1)

- Estimate true value function $v_\pi(\boldsymbol{x})$ using a parametrizable approximate value function

$$\hat{v}(\tilde{\boldsymbol{x}}, \boldsymbol{w}) \approx v_\pi(\boldsymbol{x}). \tag{1.1}$$

- The state $\boldsymbol{x}$ might be enhanced by feature engineering (i.e., additional signal inputs are derived in the feature vector $\tilde{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}) \in \mathbb{R}^\kappa$).
- Above, $\boldsymbol{w} \in \mathbb{R}^\zeta$ is the parameter vector.
- Typically, $\zeta << |\mathcal{X}|$ applies (otherwise approximation is pointless).

Generalization

Due to the usage of function approximation one incremental learning step changes at least one element $w_i \in \boldsymbol{w}$ which

- affects the estimated value of many states compared to
- the tabular case where one update step affects only one state.

Types of Action-Value Function Approximation

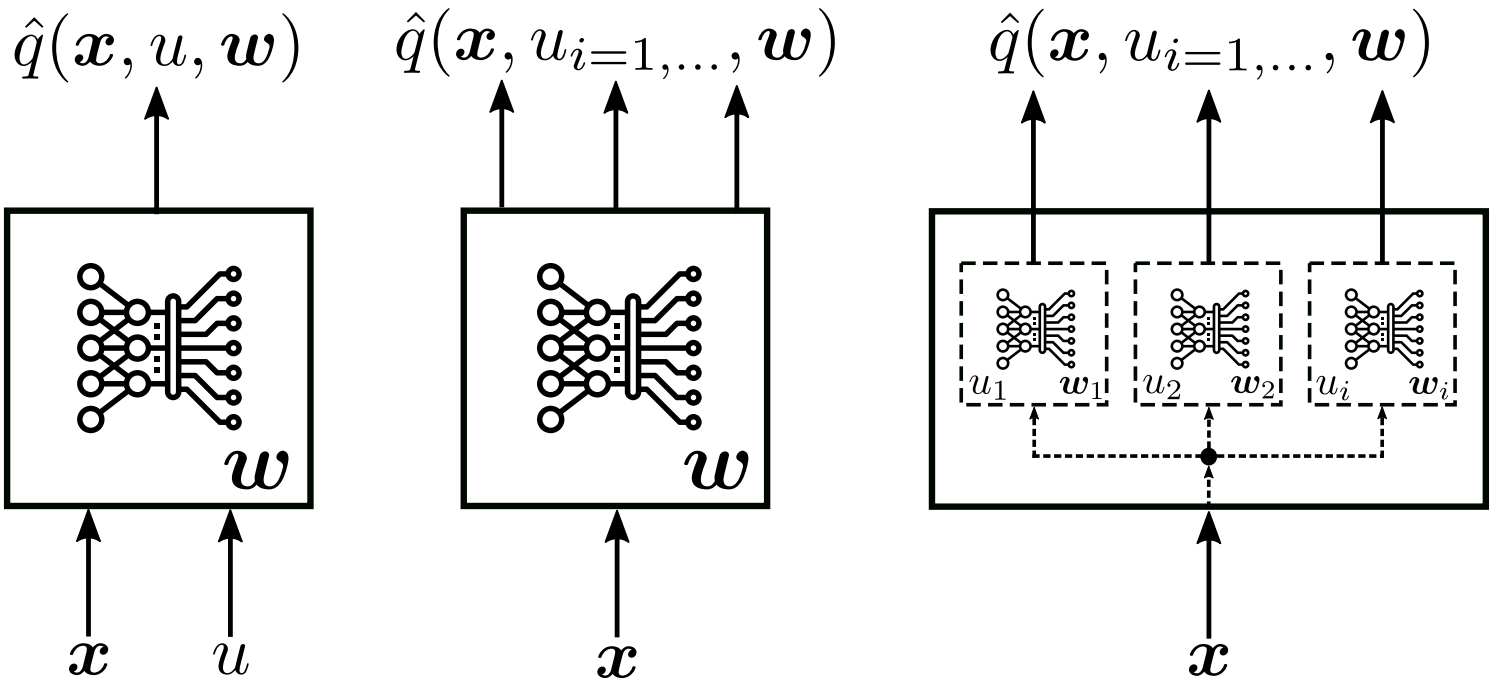


Fig. 1.1: Possible function approximation settings for discrete actions

- 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)

Prediction Framework with Function Approximation (2)

- In the tabular case a specific prediction objective was not needed:
  - The learned value function could exactly match the true value.
  - The value estimate at each state was decoupled from other states.
- Due to generalization impact we need to define an accuracy metric on the entire state space (the RL prediction goal):

Definition 1.1: Mean Squared Value Error

The RL prediction objective is defined as the mean squared value error

$$\overline{\text{VE}}(\boldsymbol{w}) = \int_{\mathcal{X}} \mu(\boldsymbol{x}) \left[v_\pi(\boldsymbol{x}) - \hat{v}(\tilde{\boldsymbol{x}}, \boldsymbol{w})\right]^2 \tag{1.2}$$

with $\mu(\boldsymbol{x}) \in \{\mathbb{R} | \mu(\boldsymbol{x}) \geq 0\}$ being a state distribution weight with $\int_{\mathcal{X}} \mu = 1$.

- Practical note: As the true value $v_\pi(\boldsymbol{x})$ is most likely unknown in most tasks, (1.2) cannot be computed exactly but only estimated.

Simplification for On-Policy Prediction

- For prediction we focus entirely on the on-policy case.
- Hence, $\mu(\boldsymbol{x})$ is the on-policy distribution under π.
- For practical usage we can therefore approximate the weighted integration over the entire state space $\mathcal{X}$ in (1.2) by the sampled MSE of the visited state trajectory:

$$\overline{\mathrm{VE}}(\boldsymbol{w}) \approx J(\boldsymbol{w}) = \sum_k \left[v_\pi(\boldsymbol{x}_k) - \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w})\right]^2 \tag{1.3}$$

- If we would perform off-policy prediction we have to transform the sampled value (estimates) from the behavior to the target policy.
- Likewise when doing this for tabular methods, this increases the prediction variance.
- In combination with generalization errors due to function approximation, the overall risk of diverging is significantly higher compared to the on-policy case.

Prediction Challenges with Function Approximation

Summarizing the two previous slides:

- The goal is to find

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} J(\boldsymbol{w}). \tag{1.4}$$

First challenge:

- Function approximator $\hat{v}(\tilde{\boldsymbol{x}}, \boldsymbol{w})$ requires certain form to fit $v_\pi(\boldsymbol{x})$.

Second challenge:

- If $\hat{v}(\tilde{\boldsymbol{x}}, \boldsymbol{w})$ is linear: convex optimization problem.
  - The nice case: the local optimum equals the global optimum and is uniquely discoverable. But requires linear feature dependence.
- If $\hat{v}(\tilde{\boldsymbol{x}}, \boldsymbol{w})$ is non-linear: non-linear optimization problem.
  - The ugly case: possible multitude of local optima with no guarantee to locate the global one.
  - Depending on optimization strategy the RL algorithm may diverge.

Updating the Parameter Vector to Find (Local) Optimum

Transferring the idea of incremental learning steps from the tabular case

$$\hat{v}(s) \leftarrow \hat{v}(s) + \alpha \left[v_\pi(s) - \hat{v}(s) \right] \tag{1.5}$$

to function approximation using a gradient descent update:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \nabla_{\boldsymbol{w}} J(\boldsymbol{w}). \tag{1.6}$$

- The search direction is the prediction objective gradient $\nabla_{\boldsymbol{w}} J(\boldsymbol{w})$.
- The learning rate α determines the step size of one update.

How to Retrieve the Gradient?

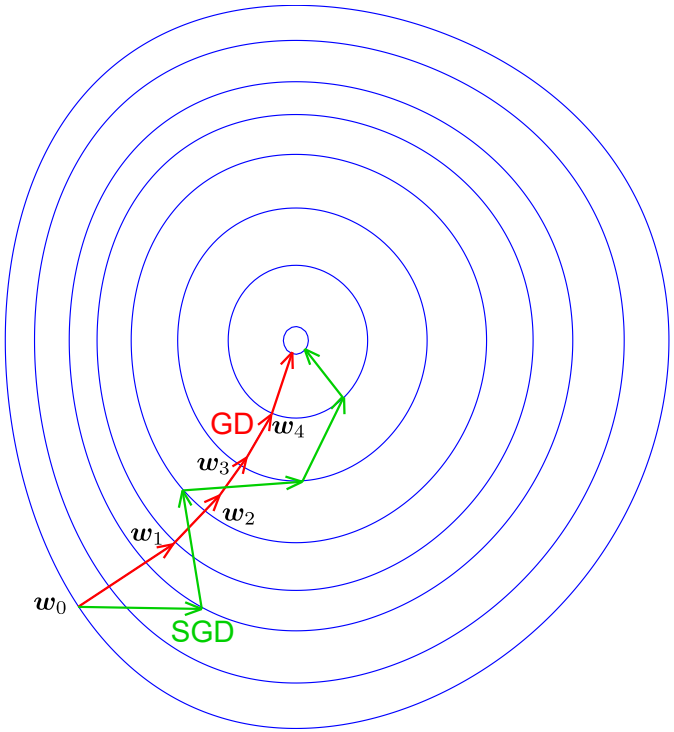


Fig. 1.2: Exemplary optimization paths for (stochastic) gradient descent (derivative work of www.wikipedia.org, CC0 1.0)

- Full calculus of $\nabla_{\boldsymbol{w}} J(\boldsymbol{w})$:
  - Batch evaluation on sampled sequence $\boldsymbol{s}_0, \boldsymbol{s}_1, \boldsymbol{s}_2, \ldots$ might be computationally costly.
  - In RL control: since π changes over time, past data in batch is not fully representative.
- SGD: sample gradient at a given state $\boldsymbol{s}_k$ and parameter vector $\boldsymbol{w}_k$:

$$\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) \approx -\left[v_\pi(\boldsymbol{s}_k) - \hat{v}(\tilde{\boldsymbol{s}}_k, \boldsymbol{w}_k)\right] \nabla_{\boldsymbol{w}} \hat{v}(\tilde{\boldsymbol{s}}_k, \boldsymbol{w}_k).$$

- Regular gradient descent leads to same result as SGD in expectation (averaging of samples).

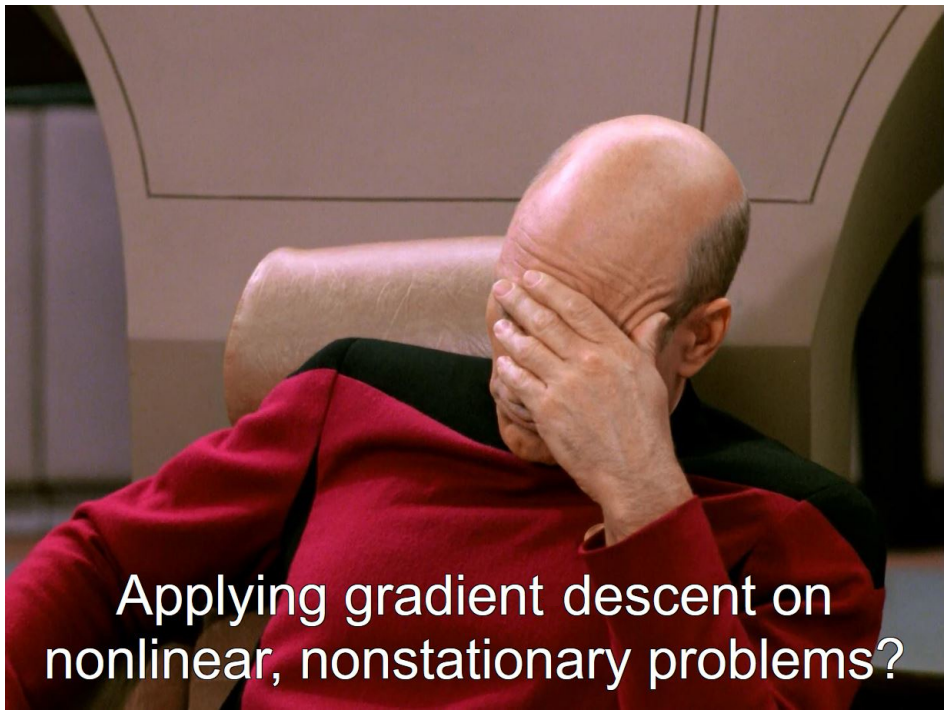
Asking an Expert on Convergence Properties

The optimization task could be

- non-linear,
- multidimensional and
- non-stationary.

Applying gradient descent to such a problem requires:

- Enormous luck to initialize $\boldsymbol{w}_0$ close to the global optimum.
- Cautious tuning of α to prevent diverging or chattering of $\boldsymbol{w}_k$.



SGD-Based Learning Step

Despite the possible problems we apply SGD-based learning due to its striking simplicity (and wide distribution in the literature):

Gradient-based parameter update

To optimize $J(w)$ by an appropriate function approximator $\hat{v}(\tilde{\boldsymbol{s}}, \boldsymbol{w})$ the incremental learning update per step is

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \alpha \left[v_\pi(\boldsymbol{s}_k) - \hat{v}(\tilde{\boldsymbol{s}}_k, \boldsymbol{w}_k) \right] \nabla_{\boldsymbol{w}} \hat{v}(\tilde{\boldsymbol{s}}_k, \boldsymbol{w}_k). \tag{1.7}$$

Nevertheless, the true update target $v_\pi(\boldsymbol{s}_k)$ is often unknown due to

- noise or
- the learning process itself (e.g. bootstrapping estimates).

Generalization Example for Parameter Update

- Function approximation $\hat{v}(\tilde{\boldsymbol{s}}, \boldsymbol{w}) = \begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix} \begin{bmatrix} s_1 & s_2 & 1 \end{bmatrix}^\mathsf{T}$
- Initial parameter: $\boldsymbol{w}_0^\mathsf{T} = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$, $v_\pi(\boldsymbol{s}_0 = \begin{bmatrix} 1 & 1 \end{bmatrix}^\mathsf{T}) = 1$, $\alpha = 0.1$
- New parameter set:

$$\begin{aligned} \boldsymbol{w}_1^\mathsf{T} &= \boldsymbol{w}_0^\mathsf{T} + \alpha \left[v_\pi(\boldsymbol{s}_0) - \hat{v}(\tilde{\boldsymbol{s}}_0, \boldsymbol{w}_0) \right] \left(\nabla_{\boldsymbol{w}} \hat{v}(\tilde{\boldsymbol{s}}_0, \boldsymbol{w}_0) \right)^\mathsf{T} \\ &= \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} + 0.1 \, (1 - 3) \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 0.8 & 0.8 & 0.8 \end{bmatrix} \end{aligned}$$

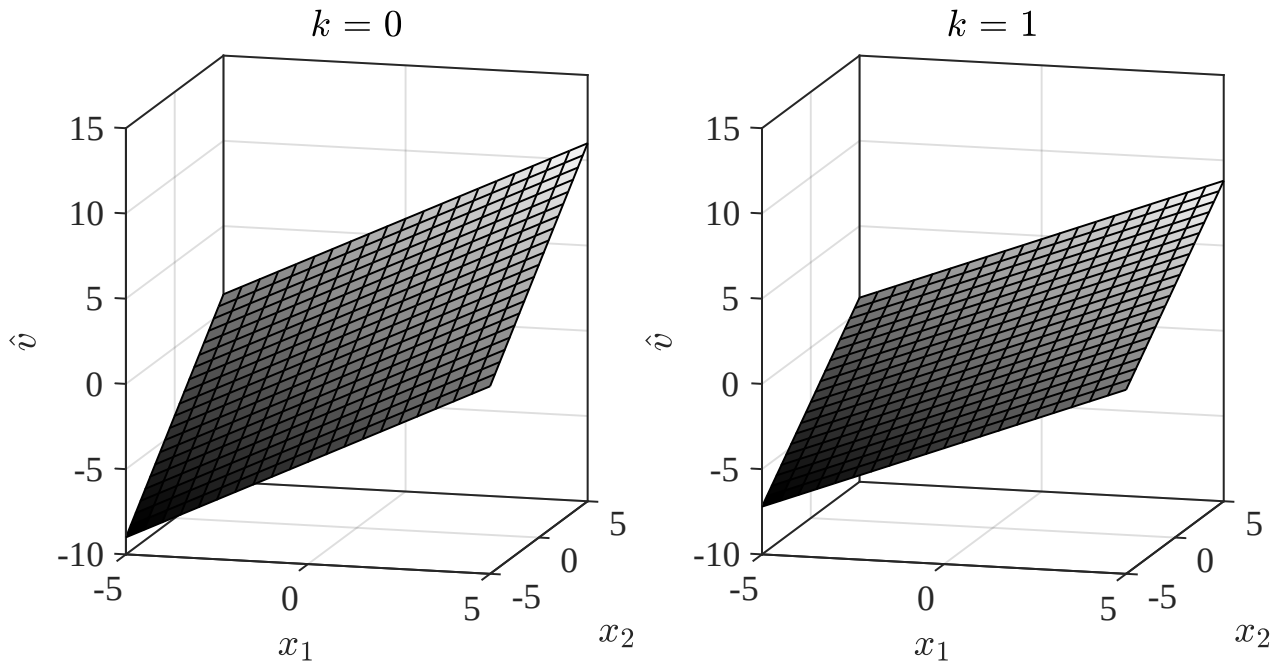


Fig. 1.3: Exemplary state-value estimation update with linear regression model

Algorithmic Implementation: Gradient Monte Carlo Prediction

- Direct transfer from tabular case to function approximation
- Update target becomes the sampled return $v_\pi(\boldsymbol{s}_k) \approx g_k$

input: a policy π to be evaluated, a feature representation $\tilde{\boldsymbol{s}} = \boldsymbol{f}(\boldsymbol{s})$
input: a differentiable function $\hat{v} : \mathbb{R}^\kappa \times \mathbb{R}^\zeta \to \mathbb{R}$
parameter: step size $\alpha \in \{\mathbb{R} | 0 < \alpha < 1\}$
init: value-function weights $\boldsymbol{w} \in \mathbb{R}^\zeta$ arbitrarily
for $j = 1, 2, \ldots,$ *episodes* **do**
    generate an episode following π: $s_0, a_0, r_1, \ldots, s_T$;
    calculate every-visit return g_k;
    for $k = 0, 1, \ldots, T-1$ *time steps* **do**
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[g_k - \hat{v}(\tilde{\boldsymbol{s}}_k, \boldsymbol{w}) \right] \nabla_{\boldsymbol{w}} \hat{v}(\tilde{\boldsymbol{s}}_k, \boldsymbol{w})$;

Algo. 1.1: Every-visit gradient MC prediction(output: parameter vector $\boldsymbol{w}$ for $\hat{v}_\pi$)

Semi-Gradient Methods

- If bootstrapping is applied, the true target $v_\pi(\boldsymbol{s}_k)$ is approximated by a target depending on the estimate $\hat{v}(\tilde{\boldsymbol{s}}_k, \boldsymbol{w})$.
- If $\hat{v}(\tilde{\boldsymbol{s}}_k, \boldsymbol{w})$ does not perfectly fit $v_\pi(\boldsymbol{s}_k)$, the update target becomes a biased estimate of $v_\pi(\boldsymbol{s}_k)$.
  - For example, in the TD(0) case applying SGD we receive:

$$
\begin{aligned}
v_\pi(\boldsymbol{s}) &\approx r + \gamma \hat{v}(\tilde{\boldsymbol{s}}', \boldsymbol{w}), \\
J(\boldsymbol{w}) &\approx \sum_k \left[r_{k+1} + \gamma \hat{v}(\tilde{\boldsymbol{s}}_{k+1}, \boldsymbol{w}_k) - \hat{v}(\tilde{\boldsymbol{s}}_k, \boldsymbol{w}_k) \right]^2, \\
\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) &\approx \left[r_{k+1} + \gamma \hat{v}(\tilde{\boldsymbol{s}}_{k+1}, \boldsymbol{w}_k) - \hat{v}(\tilde{\boldsymbol{s}}_k, \boldsymbol{w}_k) \right] \\
&\quad \nabla_{\boldsymbol{w}} \left[\gamma \hat{v}(\tilde{\boldsymbol{s}}_{k+1}, \boldsymbol{w}_k) - \hat{v}(\tilde{\boldsymbol{s}}_k, \boldsymbol{w}_k) \right].
\end{aligned}
\tag{1.8}
$$

Semi-gradient methods

When bootstrapping is applied, the gradient does not take into account any gradient component of the bootstrapped target estimate.

- Motivation: speed up gradient calculation while assuming that the simplification error is small (e.g. due to discounting).

Algorithmic Implementation: Semi-Gradient TD(0)

The semi-gradient of $J(\boldsymbol{w})$ for TD(0) from prev. slide is then

$$\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) \approx -\left[r_{k+1} + \gamma \hat{v}(\tilde{\boldsymbol{s}}_{k+1}, \boldsymbol{w}_k) - \hat{v}(\tilde{\boldsymbol{s}}_k, \boldsymbol{w}_k)\right] \nabla_{\boldsymbol{w}} \hat{v}(\tilde{\boldsymbol{s}}_k, \boldsymbol{w}_k). \quad (1.9)$$

input: a policy π to be evaluated, a feature representation $\tilde{\boldsymbol{s}} = \boldsymbol{f}(\boldsymbol{s})$
input: a differentiable function $\hat{v} : \mathbb{R}^{\kappa} \times \mathbb{R}^{\zeta} \to \mathbb{R}$ with $\hat{v}(\tilde{\boldsymbol{s}}_T, \cdot) = 0$
parameter: step size $\alpha \in \{\mathbb{R} | 0 < \alpha < 1\}$
init: value-function weights $\boldsymbol{w} \in \mathbb{R}^{\zeta}$ arbitrarily
for $j = 1, 2, \ldots$ *episodes* **do**
    initialize $\boldsymbol{s}_0$;
    for $k = 0, 1, 2 \ldots$ *time steps* **do**
        $a_k \leftarrow$ apply action from $\pi(\boldsymbol{s}_k)$;
        observe $\boldsymbol{s}_{k+1}$ and r_{k+1};
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[r_{k+1} + \gamma \hat{v}(\tilde{\boldsymbol{s}}_{k+1}, \boldsymbol{w}) - \hat{v}(\tilde{\boldsymbol{s}}_k, \boldsymbol{w})\right] \nabla_{\boldsymbol{w}} \hat{v}(\tilde{\boldsymbol{s}}_k, \boldsymbol{w})$;
        exit loop if $\boldsymbol{s}_{k+1}$ is terminal;

Algo. 1.2: Semi-gradient TD(0) (output: parameter vector $\boldsymbol{w}$ for $\hat{v}_{\pi}$)

Table of Contents

Background and Motivation

- As already discussed in the tabular case: incremental learning is not data efficient (cf. example Fig. **??**).
  - During one incremental learning step we are not utilizing the given information to the maximum possible extent.
  - Also applies to SGD-based updates with function approximation.

- Alternative: batch learning methods
  - Find $\boldsymbol{w}^*$ given a fixed, consistent data set $\boldsymbol{\mathcal{D}}$
  - $\boldsymbol{\mathcal{D}} = \{\langle \boldsymbol{x}_0, v_\pi(\boldsymbol{x}_0)\rangle, \langle \boldsymbol{x}_1, v_\pi(\boldsymbol{x}_1)\rangle, \ldots\}$

- What batch learning options do we have?
  - Experience replay (cf. planning and learning lecture e.g. Fig. **??**)
  - If $\hat{v}(\tilde{\boldsymbol{x}}, \boldsymbol{w})$ is linear: closed-form least-squares solution

SGD with Experience Replay

Based on the data set

$$\mathcal{D} = \{\langle \boldsymbol{x}_0, v_\pi(\boldsymbol{x}_0)\rangle, \langle \boldsymbol{x}_1, v_\pi(\boldsymbol{x}_1)\rangle, \ldots\}$$

repeat:

1. Sample uniformly $i = 1, \ldots, b$ state-value pairs from experience (so-called mini batch)

$$\langle \boldsymbol{x}_i, v_\pi(\boldsymbol{x}_i)\rangle \sim \mathcal{D}.$$

2. Apply (semi) SGD update step:

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \frac{\alpha}{b} \sum_{i=1}^{b} \left[v_\pi(\boldsymbol{x}_i) - \hat{v}(\tilde{\boldsymbol{x}}_i, \boldsymbol{w}_i)\right] \nabla_{\boldsymbol{w}} \hat{v}(\tilde{\boldsymbol{x}}_i, \boldsymbol{w}_i).$$

- Universally applicable: $\hat{v}(\tilde{\boldsymbol{x}}, \boldsymbol{w})$ can be any differentiable function.
- The usual technical tuning requirements regarding α apply.
- True target $v_\pi(\boldsymbol{x})$ is usually approximated by MC or TD targets.

(Ordinary) Least Squares

Assuming the following applies:

- $\hat{v}(\tilde{\boldsymbol{x}}, \boldsymbol{w})$ is a linear estimator and
- $\boldsymbol{\mathcal{D}}$ a fixed, representative data set following the on-policy distribution.

Then, minimizing the quadratic cost function (1.3) becomes

- an ordinary least squares (OLS) / linear regression problem.

We focus on the combination of OLS and TD(0) (so-called LSTD), but the following can be equally extended to n-step learning or MC.

- Rewriting $J(\boldsymbol{w})$ from (1.3) using linear approximation TD(0) target:

$$v_\pi(\boldsymbol{x}_k) \approx r_{k+1} + \gamma \hat{v}(\boldsymbol{x}_{k+1}) = r_{k+1} + \gamma \tilde{\boldsymbol{x}}_{k+1}^\mathsf{T} \boldsymbol{w} \tag{1.10}$$

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

Table of Contents

Gradient-Based Action-Value Learning

- Transferring the objective $J(w)$ from on-policy prediction to control yields:

$$J(\boldsymbol{w}) = \sum_k \left[q_\pi(\boldsymbol{s}_k, a_k) - \hat{q}(\boldsymbol{s}_k, a_k, \boldsymbol{w})\right]^2. \tag{1.11}$$

- Analogous, the (semi-)gradient-based parameter update from (1.7) is also applied to action values:

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \alpha \left[q_\pi(\boldsymbol{s}_k, a_k) - \hat{q}(\boldsymbol{s}_k, a_k, \boldsymbol{w}_k)\right] \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{s}_k, a_k, \boldsymbol{w}_k). \tag{1.12}$$

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

Houston: We have a Problem (1)

- Recall tabular policy improvement theorem guarantees to find a globally better or equally good policy in each update step.
- With parameter updates (1.12) generalization applies.
- Hence, when reacting to one specific state-action transition other parts of the state-action space within $\hat{q}$ are affected too.

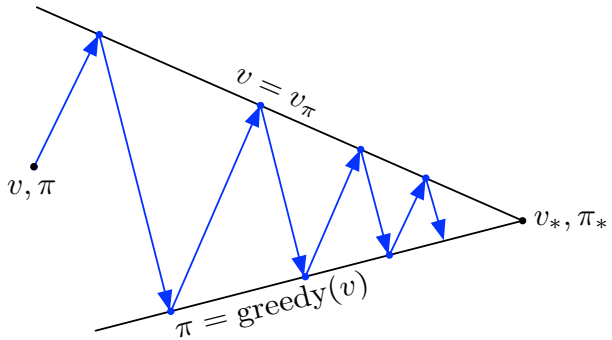


Fig. 1.4: GPI

Loss of policy improvement theorem

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

Houston: We have a Problem (2)

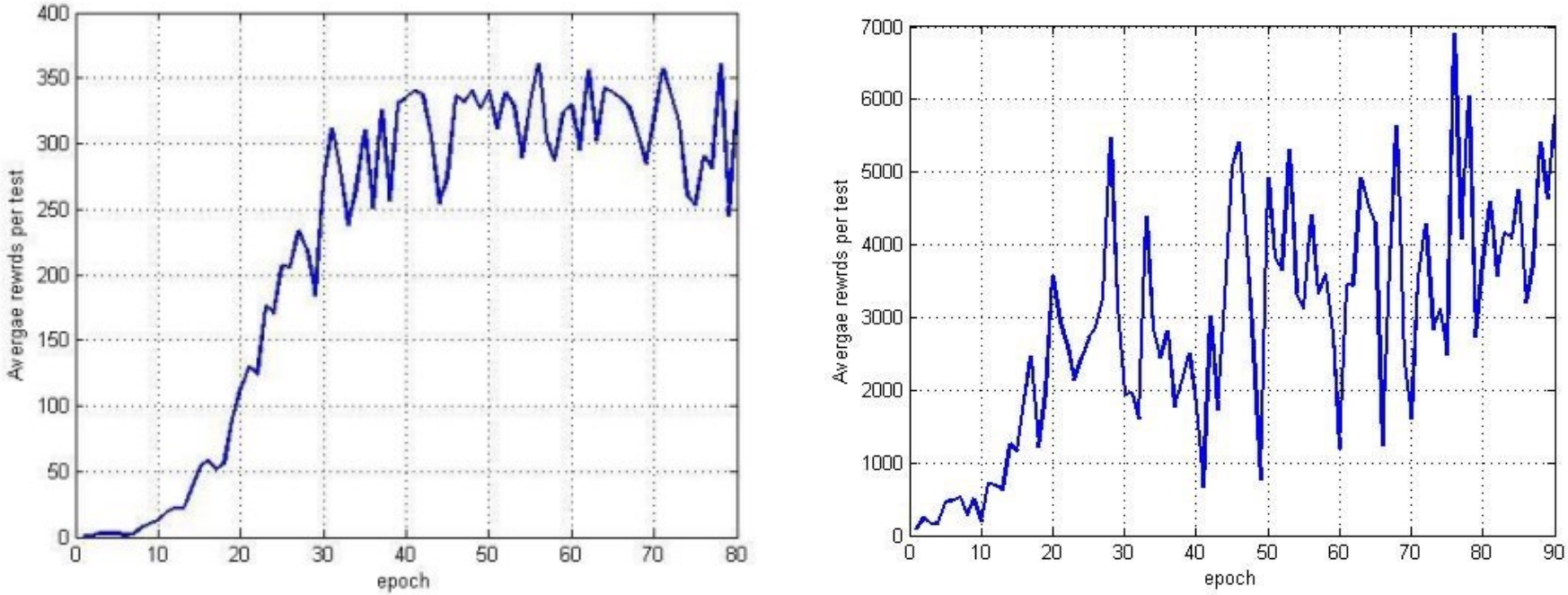


Fig. 1.5: Learning curves with drastic performance dips when applying Sarsa with function approximation. Left: Atari Breakout, right: Atari Seaquest (source: D. Zhao et al., *Deep reinforcement learning with experience replay based on SARSA*, IEEE Symposium Series on Computational Intelligence, 2016)

Algorithmic Implementation: Gradient MC Control

- Direct transfer from tabular case to function approximation
- Update target becomes the sampled return $q_\pi(\boldsymbol{s}_k, a_k) \approx g_k$
- 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 $\hat{q}$: $s_0, a_0, r_1, \ldots, s_T$;
    calculate every-visit return g_k;
    for $k = 0, 1, \ldots, T-1$ *time steps* **do**
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[g_k - \hat{q}(\boldsymbol{s}_k, a_k, \boldsymbol{w})\right] \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{s}_k, a_k, \boldsymbol{w})$;

Algo. 1.3: 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 $\boldsymbol{s}_0$;
    for $k = 0, 1, 2 \ldots$ *time steps* **do**
        $u_k \leftarrow$ apply action from $\pi(\boldsymbol{s}_k)$ or ε-greedy on $\hat{q}(\boldsymbol{s}_k, \cdot, \boldsymbol{w})$;
        observe $\boldsymbol{s}_{k+1}$ and r_{k+1};
        if $\boldsymbol{s}_{k+1}$ *is terminal* **then**
            $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[r_{k+1} - \hat{q}(\boldsymbol{s}_k, a_k, \boldsymbol{w})\right] \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{s}_k, a_k, \boldsymbol{w})$;
            go to next episode;
        choose u' from $\pi(\boldsymbol{s}_{k+1})$ or ε-greedy on $\hat{q}(\boldsymbol{s}_{k+1}, \cdot, \boldsymbol{w})$;
        $\boldsymbol{w} \leftarrow$
        $\boldsymbol{w} + \alpha \left[r_{k+1} + \gamma \hat{q}(\boldsymbol{s}_{k+1}, a', \boldsymbol{w}) - \hat{q}(\boldsymbol{s}_k, a_k, \boldsymbol{w})\right] \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{s}_k, a_k, \boldsymbol{w})$;

Algo. 1.4: Semi-gradient Sarsa action-value estimation (output: parameter vector $\boldsymbol{w}$ for $\hat{q}_\pi$ or $\hat{q}^*$)

Sarsa Application Example: Mountain Car (1)

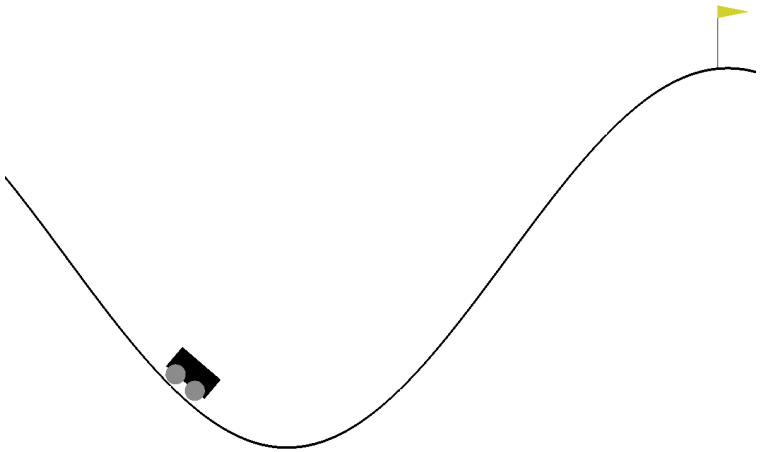


Fig. 1.6: Classic RL control example: mountain car (derivative work based on https://github.com/openai/gym, MIT license)

- 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)

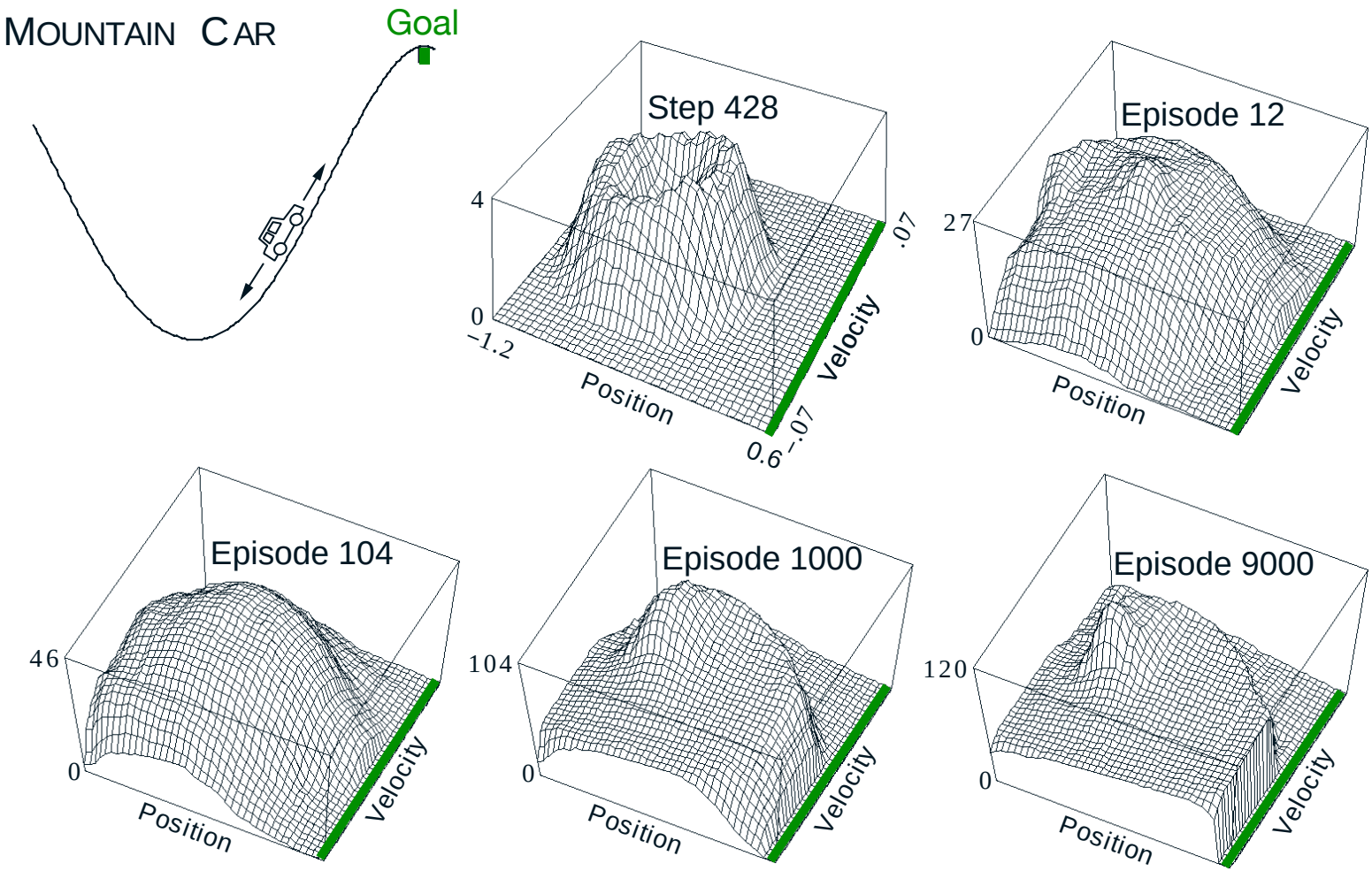


Fig. 1.7: Cost-to-go function $-\max_a \hat{q}(\boldsymbol{s}, a, \boldsymbol{w})$ for mountain car task using linear approximation with Sarsa and tile coding (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

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 .

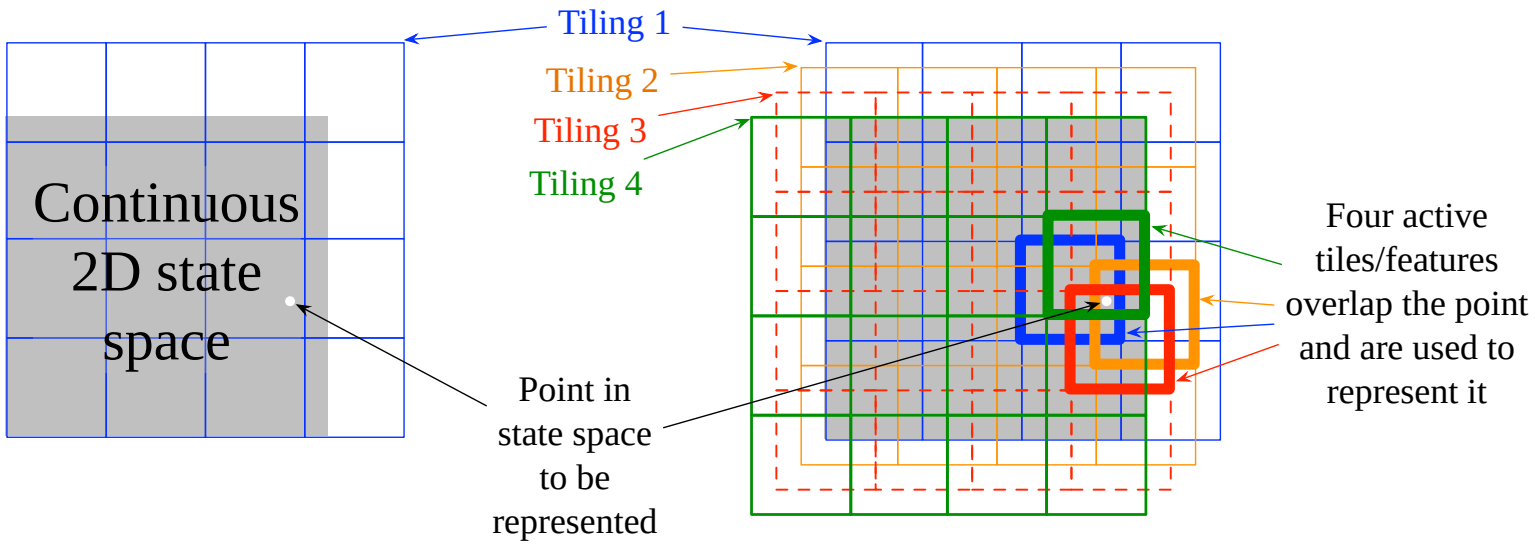


Fig. 1.8: Tile coding example in 2D (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Sarsa Application Example: Mountain Car (3)

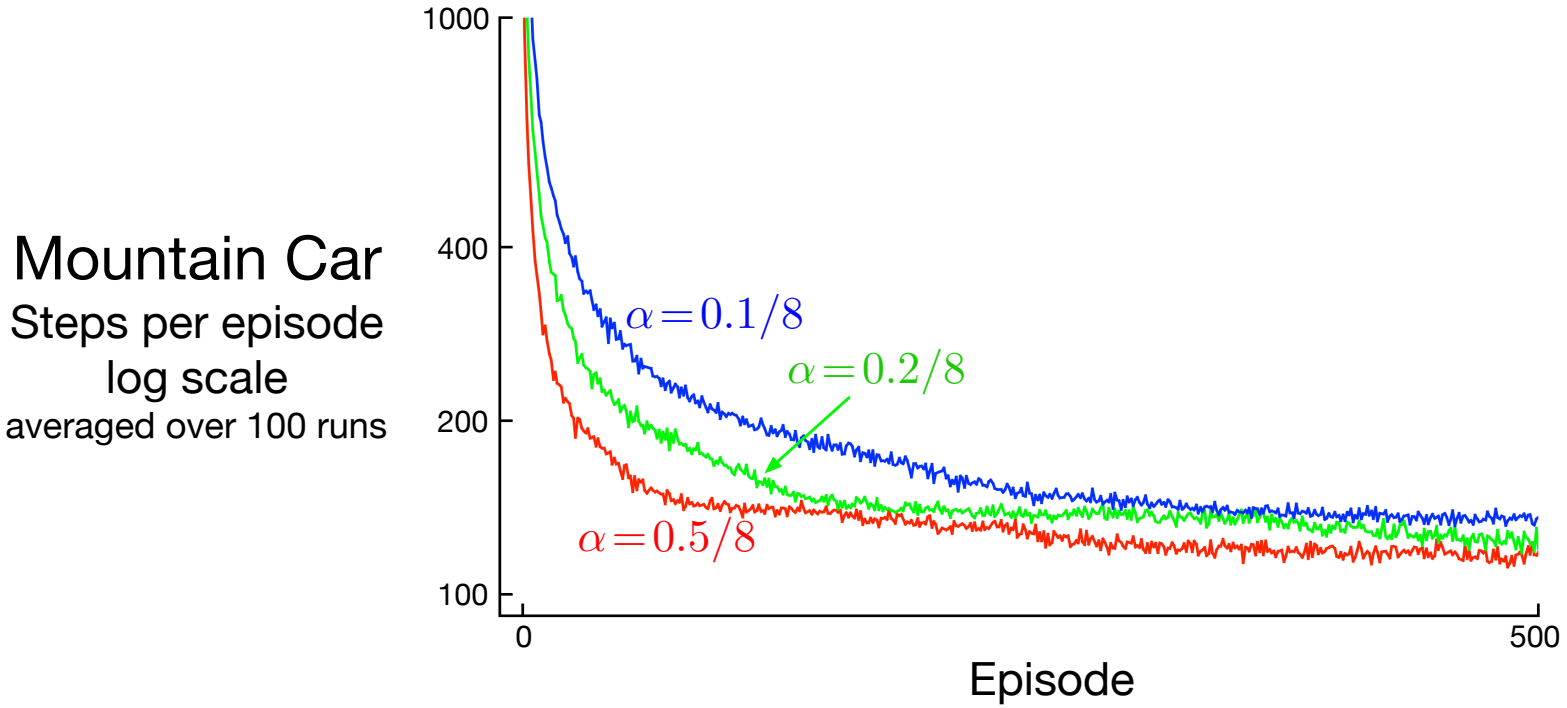


Fig. 1.9: Mountain car learning curves with semi-gradient Sarsa for different learning rates α (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Table of Contents

General Background on DQN

- Recall incremental learning step from tabular Q-learning:

$$\hat{q}(s,a) \leftarrow \hat{q}(s,a) + \alpha \left[r + \gamma \max_a \hat{q}(s',a) - \hat{q}(s,a) \right].$$

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

$$\boldsymbol{w} = \boldsymbol{w} + \alpha \left[r + \gamma \max_a \hat{q}(\boldsymbol{s}',a,\boldsymbol{w}) - \hat{q}(\boldsymbol{s},a,\boldsymbol{w}) \right] \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{s},a,\boldsymbol{w}). \quad (1.13)$$

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 a based on $\hat{q}(\boldsymbol{s}, a, \boldsymbol{w})$ (e.g., ε-greedy).
- Store observed tuples $\langle \boldsymbol{s}, a, r, \boldsymbol{s}' \rangle$ 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):

$$q_\pi(\boldsymbol{s}, a) \approx r + \gamma \max_a \hat{q}(\boldsymbol{s}', a, \boldsymbol{w}^-).$$

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

$$\mathcal{L}(\boldsymbol{w}) = \left[\left(r + \gamma \max_a \hat{q}(\boldsymbol{s}', a, \boldsymbol{w}^-) \right) - \hat{q}(\boldsymbol{s}, a, \boldsymbol{w}) \right]^2_{\boldsymbol{\mathcal{D}}_b} . \tag{1.14}$$

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

Summary of DQN Working Principle (2)

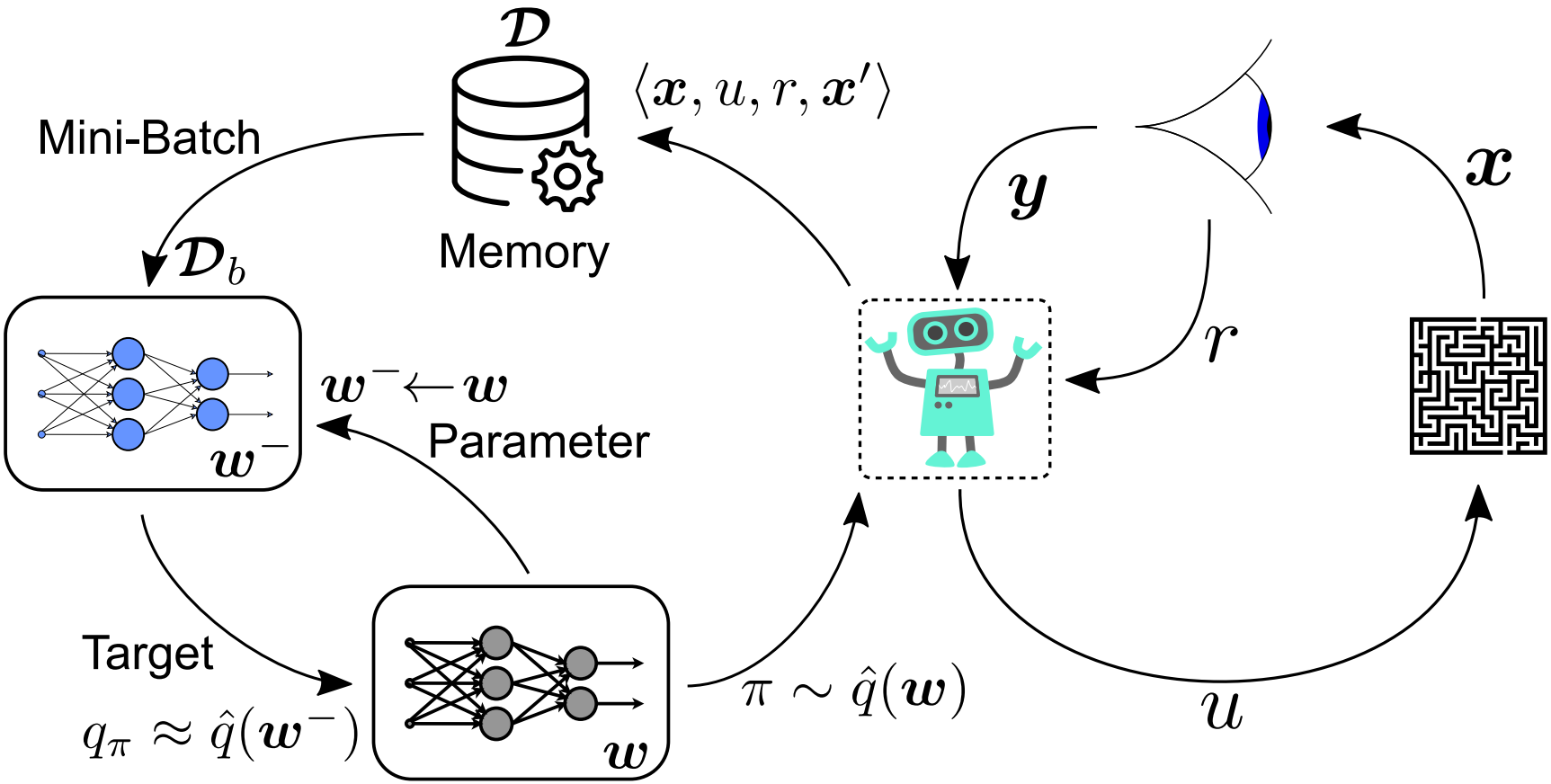


Fig. 1.10: DQN structure from a bird's-eye perspective (derivative work of Fig. **??** and wikipedia.org, CC0 1.0)

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 $\boldsymbol{s}_0$;
    for $k = 0, 1, 2 \ldots$ *time steps* **do**
        $a_k \leftarrow$ apply action ε-greedy w.r.t $\hat{q}(\boldsymbol{s}_k, \cdot, \boldsymbol{w})$;
        observe $\boldsymbol{s}_{k+1}$ and r_{k+1};
        store tuple $\langle \boldsymbol{s}_k, a_k, r_{k+1}, \boldsymbol{s}_{k+1} \rangle$ 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 $\boldsymbol{s}_{i+1}$ *is terminal* **then** $y_i = r_{i+1}$;
            else $y_i = r_{i+1} + \gamma \max_a \hat{q}(\boldsymbol{s}_{i+1}, a, \boldsymbol{w}^-)$;
        fit $\boldsymbol{w}$ on loss $\mathcal{L}(\boldsymbol{w}) = [y_i - \hat{q}(\boldsymbol{s}_i, a_i, \boldsymbol{w})]^2_{\boldsymbol{\mathcal{D}}_b}$;
        if $k \mod k_w = 0$ **then** $\boldsymbol{w}^- \leftarrow \boldsymbol{w}$ (update target weights);

Algo. 1.5: 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
- Likewise the simple ε-greedy approach can be extended.
  - Often a scheduled/annealed trajectory ε_k is used.

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

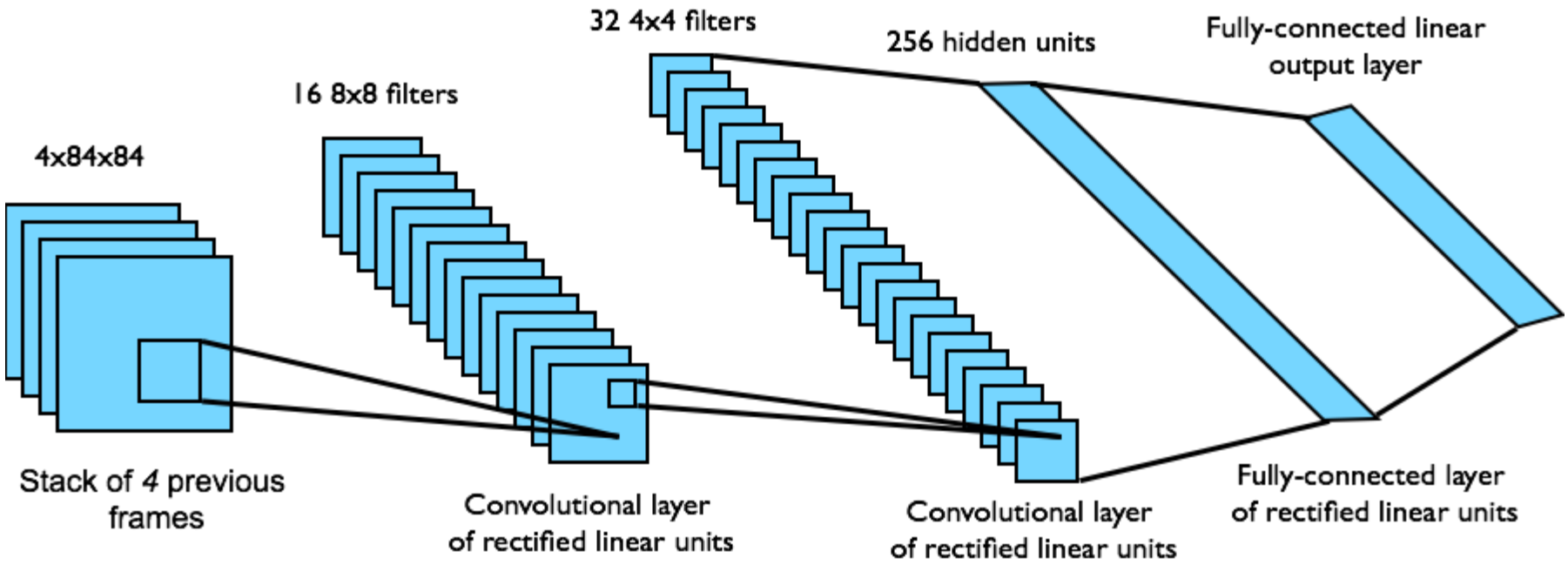


Fig. 1.11: Network architecture overview used for DQN in Atari games (source: D. Silver, Reinforcement learning, 2016. CC BY-NC 4.0)

DQN Application Example: Atari Games (2)

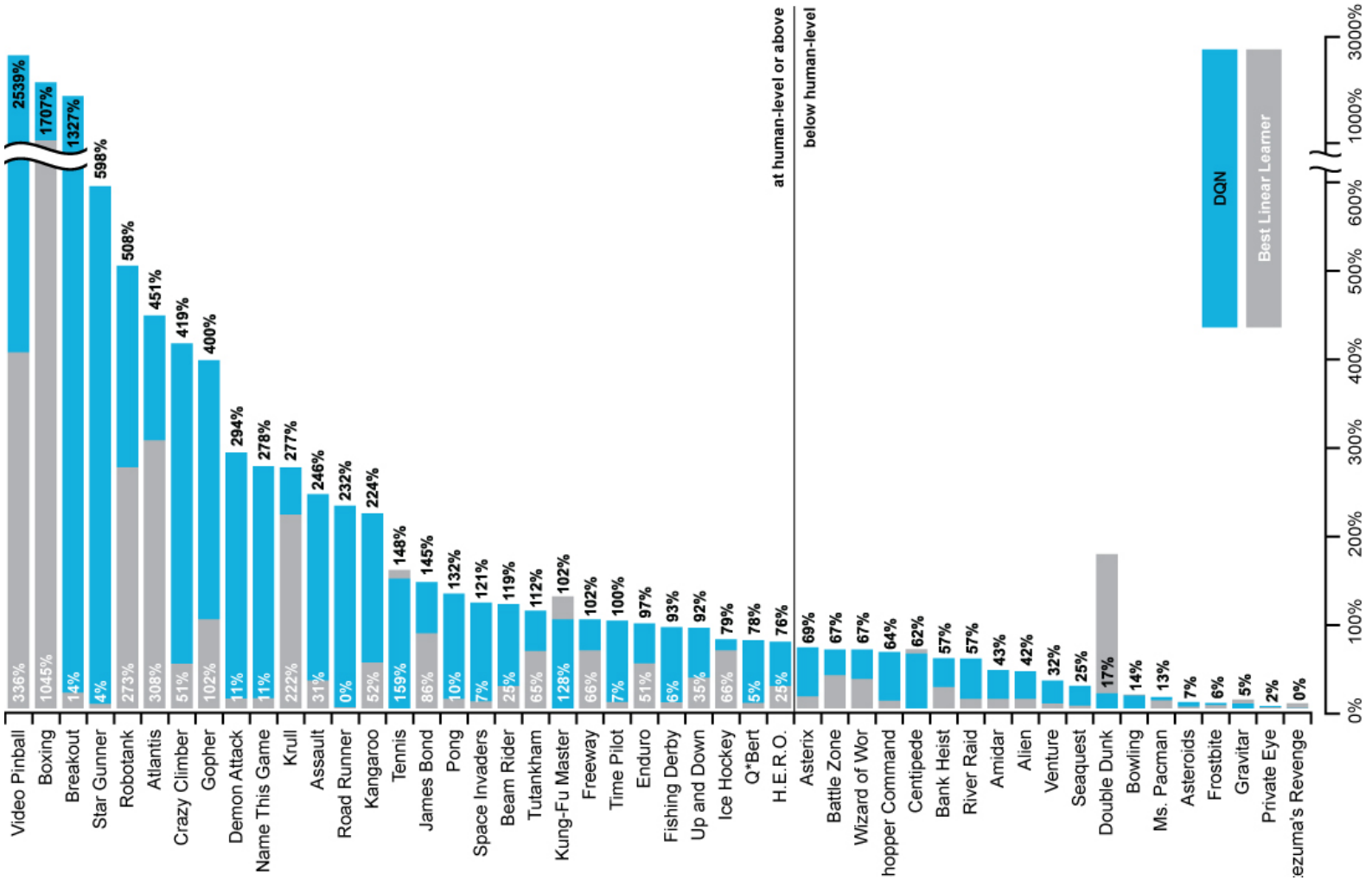


Fig. 1.12: DQN performance results in Atari games against human performance (source: D. Silver, Reinforcement learning, 2016. CC BY-NC 4.0)