Function Approximation, Deep Q Network

Milan Straka

November 12, 2018

EUROPEAN UNION
European Structural and Investment Fund
Operational Programme Research, Development and Education

Charles University in Prague
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics

n-step Methods

Full return is

$$G_t = \sum_{k=t}^{\infty} R_{k+1},$$

one-step return is

$$G_{t:t+1} = R_{t+1} + \gamma V_t(S_{t+1}).$$

We can generalize both into n-step returns:

$$G_{t:t+n} \stackrel{\text{def}}{=} \left(\sum_{k=t}^{t+n-1} \gamma^{k-t} R_{k+1}\right) + \gamma^n V_{t+n-1}(S_{t+n}).$$

with $G_{t:t+n} \stackrel{\text{def}}{=} G_t$ if $t+n \geq T$.

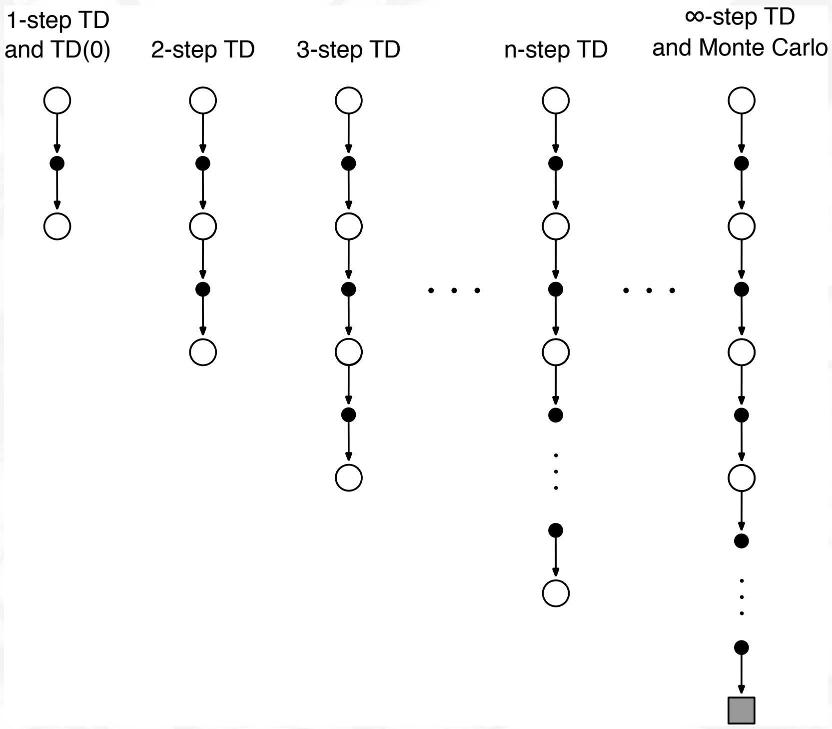


Figure 7.1 of "Reinforcement Learning: An Introduction, Second Edition".

Defining the n-step return to utilize action-value function as

$$G_{t:t+n} \stackrel{\text{def}}{=} \left(\sum_{k=t}^{t+n-1} \gamma^{k-t} R_{k+1}\right) + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n})$$

with $G_{t:t+n} \stackrel{\text{def}}{=} G_t$ if $t + n \geq T$, we get the following straightforward update rule:

$$Q_{t+n}(S_t, A_t) \stackrel{\text{def}}{=} Q_{t+n-1}(S_t, A_t) + \alpha \left[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)\right].$$

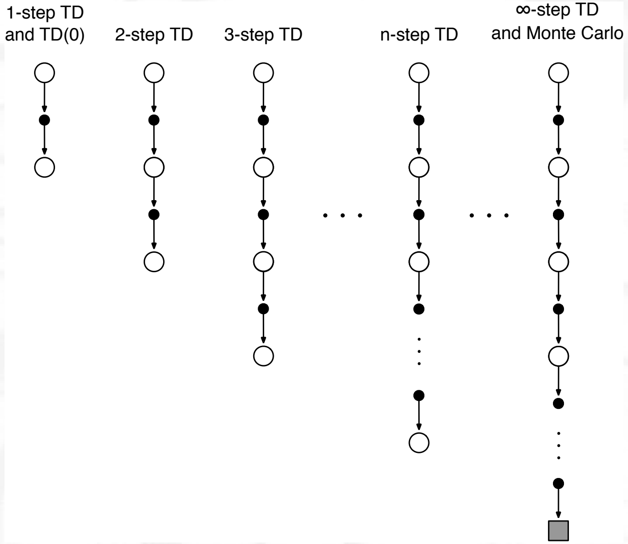


Figure 7.1 of "Reinforcement Learning: An Introduction, Second Edition".

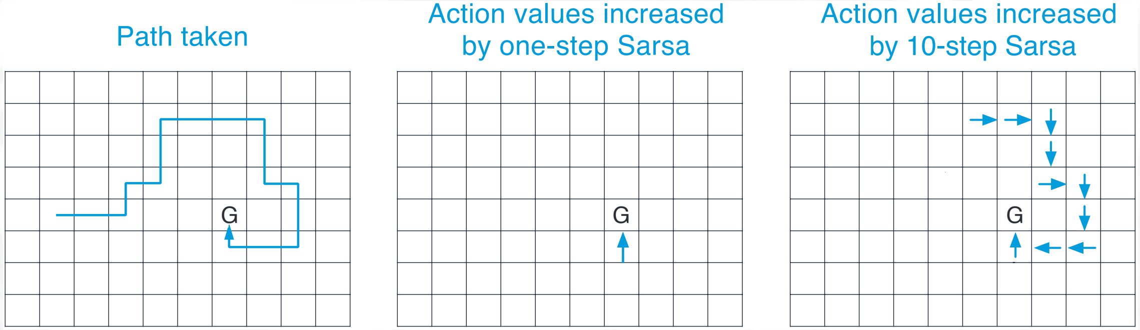


Figure 7.4 of "Reinforcement Learning: An Introduction, Second Edition".

Off-policy n-step Sarsa

Recall the relative probability of a trajectory under the target and behaviour policies, which we now generalize as

$$\rho_{t:t+n} \stackrel{\text{def}}{=} \prod_{k=t}^{\min(t+n,T-1)} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}.$$

Then a simple off-policy n-step TD can be computed as

$$V_{t+n}(S_t) \stackrel{\text{def}}{=} V_{t+n-1}(S_t) + \alpha\rho_{t:t+n-1}\left[G_{t:t+n} - V_{t+n-1}(S_t)\right].$$

Similarly, n-step Sarsa becomes

$$Q_{t+n}(S_t, A_t) \stackrel{\text{def}}{=} Q_{t+n-1}(S_t, A_t) + \alpha\boldsymbol{\rho_{t+1:t+n}}\left[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)\right].$$

Off-policy n-step Without Importance Sampling

We now derive the n-step reward, starting from one-step:

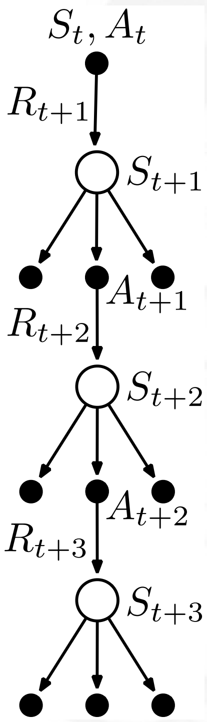
$$G_{t:t+1} \stackrel{\text{def}}{=} R_{t+1} + \sum\nolimits_a \pi(a|S_{t+1}) Q_t(S_{t+1}, a).$$

For two-step, we get:

$$G_{t:t+2} \stackrel{\text{def}}{=} R_{t+1} + \gamma \sum\nolimits_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_t(S_{t+1}, a) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:t+2}.$$

Therefore, we can generalize to:

$$G_{t:t+n} \stackrel{\text{def}}{=} R_{t+1} + \gamma \sum\nolimits_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_t(S_{t+1}, a) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:t+n}.$$



the 3-step tree-backup update

Example in Section 7.5 of "Reinforcement Learning: An Introduction, Second Edition".

Function Approximation

We will approximate value function v and/or state-value function q, choosing from a family of functions parametrized by a weight vector $\boldsymbol{w} \in \mathbb{R}^d$.

We denote the approximations as

$$\hat{v}(s, \boldsymbol{w}),$$
$$\hat{q}(s, a, \boldsymbol{w}).$$

We utilize the *Mean Squared Value Error* objective, denoted $\overline{VE}$:

$$\overline{VE}(\boldsymbol{w}) \stackrel{\text{def}}{=} \sum_{s \in \mathcal{S}} \mu(s) \left[v_\pi(s) - \hat{v}(s, \boldsymbol{w}) \right]^2,$$

where the state distribution $\mu(s)$ is usually on-policy distribution.

Gradient and Semi-Gradient Methods

The functional approximation (i.e., the weight vector $\boldsymbol{w}$) is usually optimized using gradient methods, for example as

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

As usual, the $v_\pi(S_t)$ is estimated by a suitable sample. For example in Monte Carlo methods, we use episodic return G_t, and in temporal difference methods, we employ bootstrapping and use $R_{t+1} + \gamma\hat{v}(S_{t+1}, \boldsymbol{w})$.

Monte Carlo Gradient Policy Evaluation

Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using π
  Loop for each step of episode, $t = 0, 1, \ldots, T-1$:
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[G_t - \hat{v}(S_t,\mathbf{w})\big]\nabla\hat{v}(S_t,\mathbf{w})$

Algorithm 9.3 of "Reinforcement Learning: An Introduction, Second Edition".

Linear Methods

A simple special case of function approximation are linear methods, where

$$\hat v(\boldsymbol x(s), \boldsymbol w) \stackrel{\text{def}}{=} \boldsymbol x(s)^T \boldsymbol w = \sum x(s)_i w_i.$$

The $\boldsymbol x(s)$ is a representation of state s, which is a vector of the same size as $\boldsymbol w$. It is sometimes called a *feature vector*.

The SGD update rule then becomes

$$\boldsymbol w_{t+1} \leftarrow \boldsymbol w_t - \alpha \left[v_\pi(S_t) - \hat v(\boldsymbol x(S_t), \boldsymbol w_t)\right] \boldsymbol x(S_t).$$

Feature Construction for Linear Methods

Many methods developed in the past:

- state aggregation,
- polynomials
- Fourier basis
- tile coding
- radial basis functions

But of course, nowadays we use deep neural networks which construct a suitable feature vector automatically as a latent variable (the last hidden layer).

Simple way of generating a feature vector is *state aggregation*, where several neighboring states are grouped together.

For example, consider a 1000-state random walk, where transitions go uniformly randomly to any of 100 neighboring states on the left or on the right. Using state aggregation, we can partition the 1000 states into 10 groups of 100 states. Monte Carlo policy evaluation then computes the following:

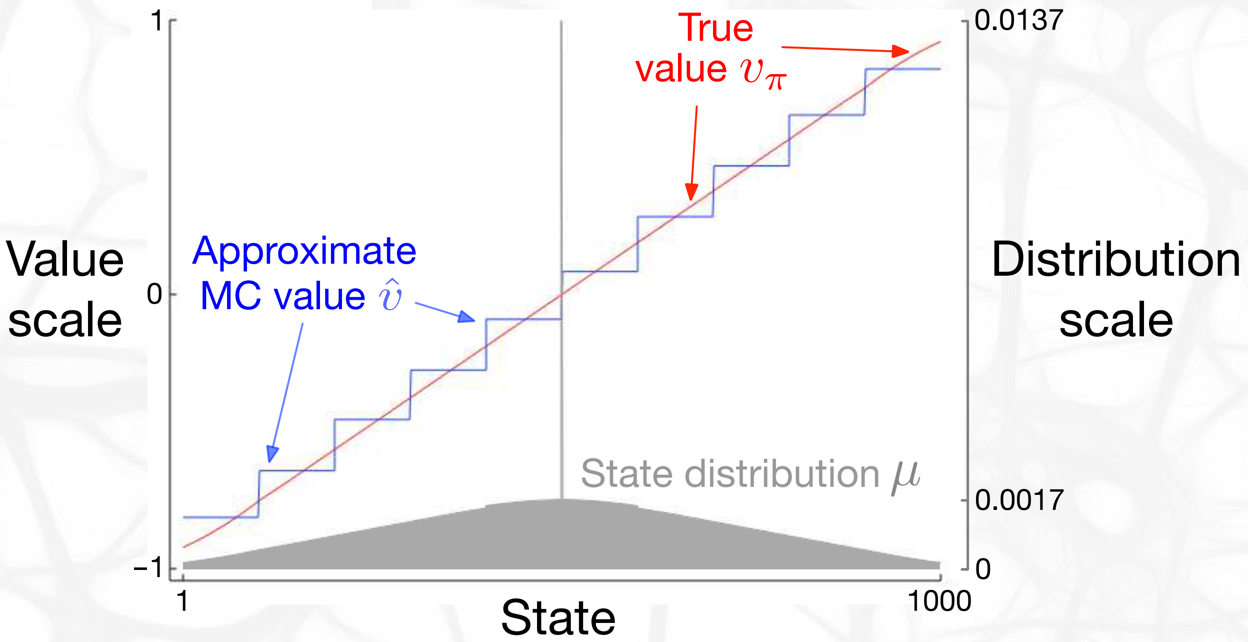


Figure 9.1 of "Reinforcement Learning: An Introduction, Second Edition".

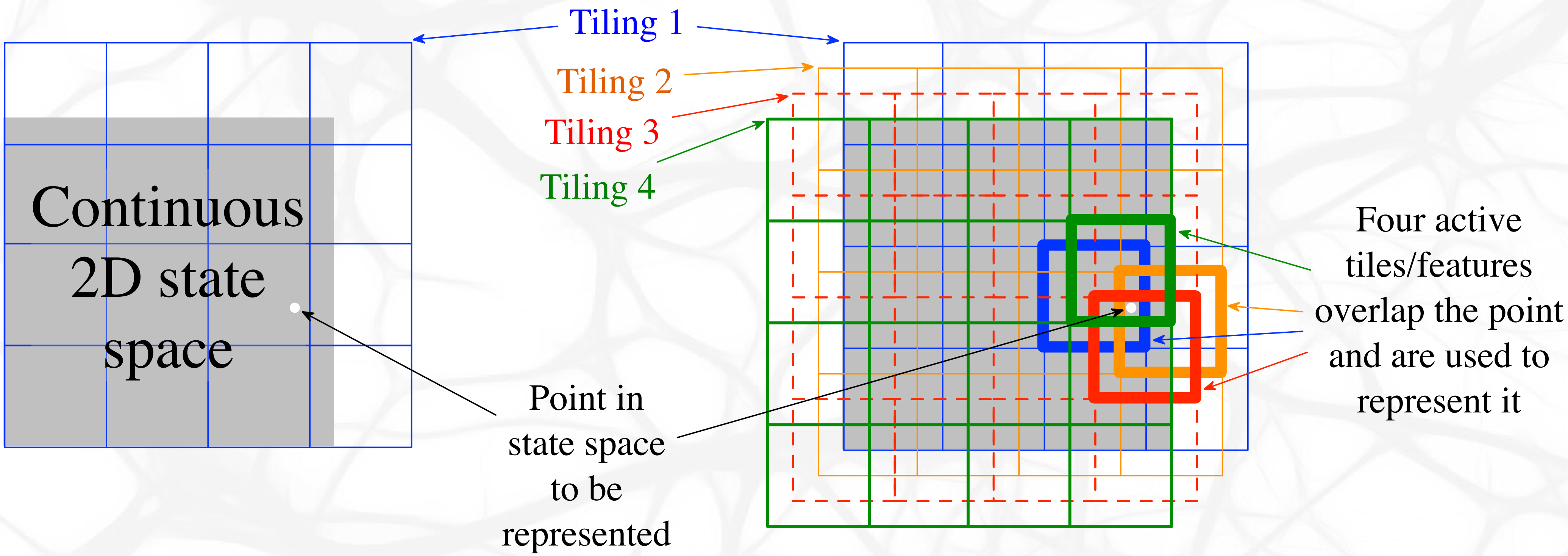


Figure 9.9 of "Reinforcement Learning: An Introduction, Second Edition".

If t overlapping tiles are used, the learning rate is usually normalized as α/t.

For example, on the 1000-state random walk example, the performance of tile coding surpasses state aggregation:

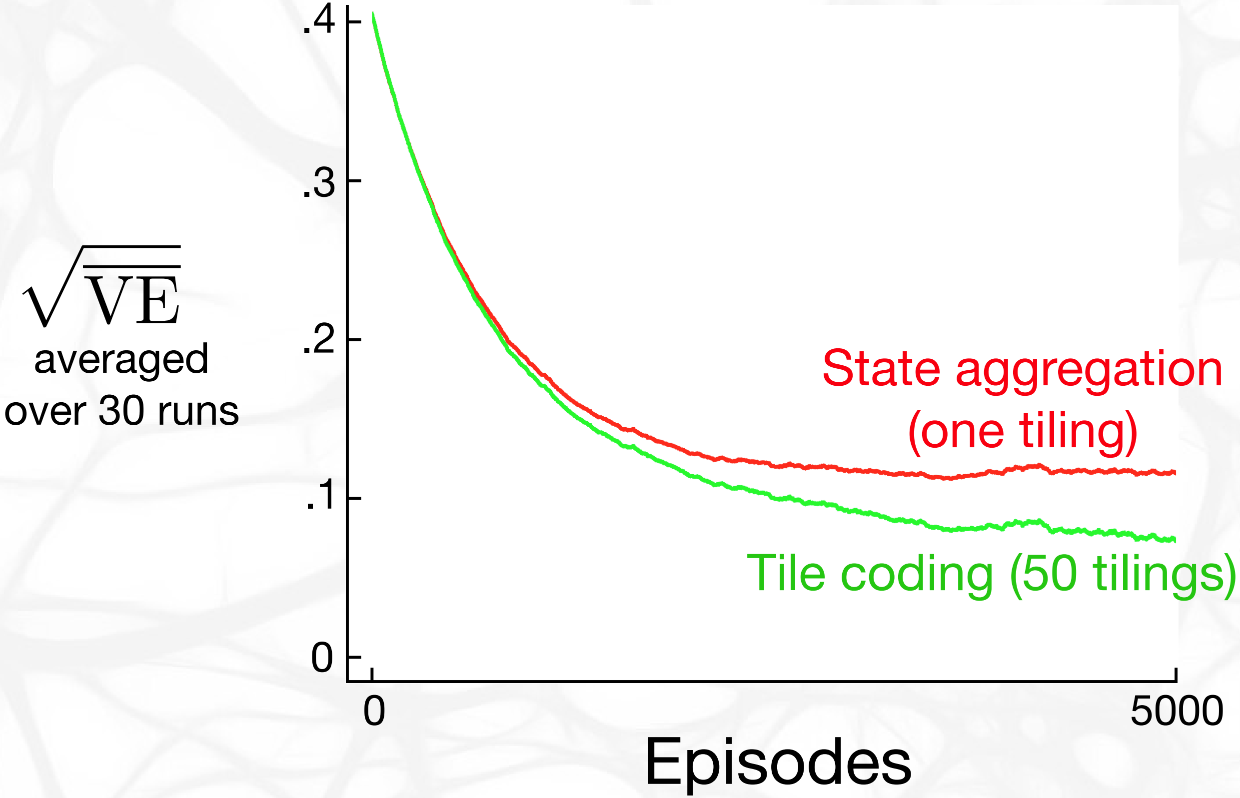


Figure 9.10 of "Reinforcement Learning: An Introduction, Second Edition".

Asymmetrical Tile Coding

In higher dimensions, the tiles should have asymmetrical offsets, with a sequence of $(1, 3, 5, \ldots, 2d-1)$ being a good choice.

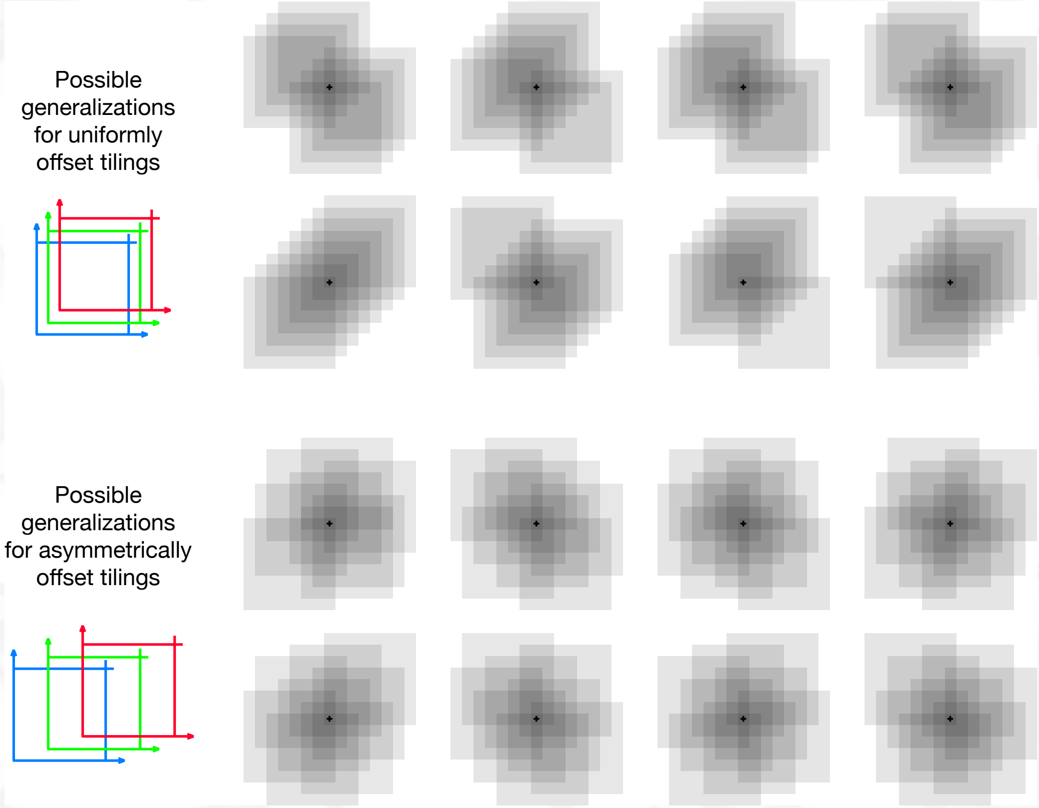


Figure 9.11 of "Reinforcement Learning: An Introduction, Second Edition".

Temporal Difference Semi-Gradient Policy Evaluation

In TD methods, we again use bootstrapping to estimate $v_\pi(S_t)$ as $R_{t+1} + \gamma\hat{v}(S_{t+1}, \boldsymbol{w})$.

Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
    Initialize S
    Loop for each step of episode:
        Choose $A \sim \pi(\cdot|S)$
        Take action A, observe R, S'
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big]\nabla\hat{v}(S,\mathbf{w})$
        $S \leftarrow S'$
    until S is terminal

Algorithm 9.3 of "Reinforcement Learning: An Introduction, Second Edition".

Note that such algorithm is called *semi-gradient*, because it does not backpropagate through $\hat{v}(S', \boldsymbol{w})$.

Temporal Difference Semi-Gradient Policy Evaluation

An important fact is that linear semi-gradient TD methods do not converge to $\overline{VE}$. Instead, they converge to a different *TD fixed point* $\boldsymbol{w}_\mathrm{TD}$.

It can be proven that

$$\overline{VE}(\boldsymbol{w}_\mathrm{TD}) \leq \frac{1}{1-\gamma} \min_{\boldsymbol{w}} \overline{VE}(\boldsymbol{w}).$$

However, when γ is close to one, the multiplication factor in the above bound is quite large.

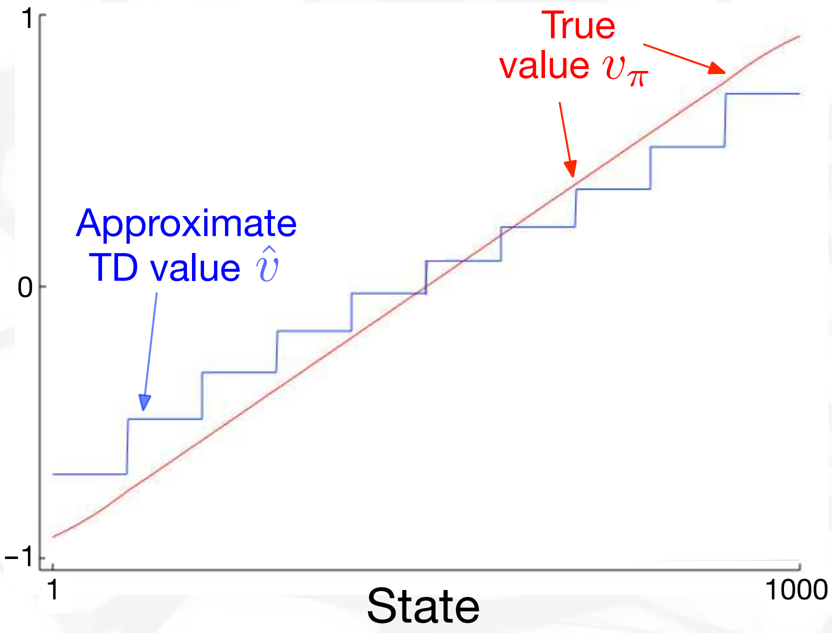
As before, we can utilize n-step TD methods.

n-step semi-gradient TD for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \rightarrow \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameters: step size $\alpha > 0$, a positive integer n
Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
All store and access operations (S_t and R_t) can take their index mod $n+1$

Loop for each episode:
    Initialize and store $S_0 \neq$ terminal
    $T \leftarrow \infty$
    Loop for $t = 0, 1, 2, \ldots$:
    | If $t < T$, then:
    | Take an action according to $\pi(\cdot|S_t)$
    | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
    | If S_{t+1} is terminal, then $T \leftarrow t + 1$
    | $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
    | If $\tau \geq 0$:
    | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
    | If $\tau + n < T$, then: $G \leftarrow G + \gamma^n \hat{v}(S_{\tau+n},\mathbf{w})$ $(G_{\tau:\tau+n})$
    | $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[G - \hat{v}(S_\tau,\mathbf{w})\right] \nabla \hat{v}(S_\tau,\mathbf{w})$
    Until $\tau = T - 1$

Algorithm 9.5 of "Reinforcement Learning: An Introduction, Second Edition".



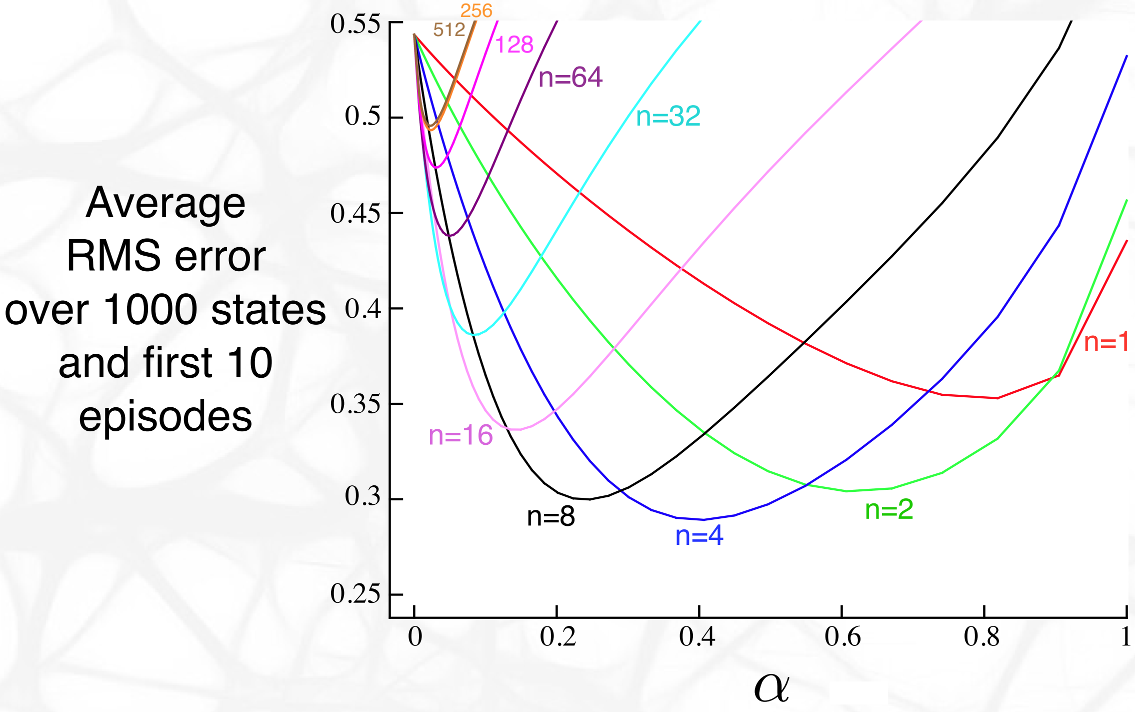


Figure 9.2 of "Reinforcement Learning: An Introduction, Second Edition".

Sarsa with Function Approximation

Until now, we talked only about policy evaluation. Naturally, we can extend it to a full Sarsa algorithm:

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)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S'$
    $A \leftarrow A'$

Algorithm 10.1 of "Reinforcement Learning: An Introduction, Second Edition".

Additionally, we can incorporate n-step returns:

Episodic semi-gradient n-step Sarsa for estimating $\hat{q} \approx q_*$ or q_π

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Input: a policy π (if estimating q_π)
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$, a positive integer n
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
All store and access operations (S_t, A_t, and R_t) can take their index mod $n + 1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  Select and store an action $A_0 \sim \pi(\cdot|S_0)$ or ε-greedy wrt $\hat{q}(S_0, \cdot, \mathbf{w})$
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
  | If $t < T$, then:
  | Take action A_t
  | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  | If S_{t+1} is terminal, then:
  | $T \leftarrow t + 1$
  | else:
  | Select and store $A_{t+1} \sim \pi(\cdot|S_{t+1})$ or ε-greedy wrt $\hat{q}(S_{t+1}, \cdot, \mathbf{w})$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose estimate is being updated)
  | If $\tau \geq 0$:
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  | If $\tau + n < T$, then $G \leftarrow G + \gamma^n \hat{q}(S_{\tau+n}, A_{\tau+n}, \mathbf{w})$ $(G_{\tau:\tau+n})$
  | $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[G - \hat{q}(S_\tau, A_\tau, \mathbf{w})\right] \nabla \hat{q}(S_\tau, A_\tau, \mathbf{w})$
  Until $\tau = T - 1$

Algorithm 10.2 of "Reinforcement Learning: An Introduction, Second Edition".

Mountain Car Example

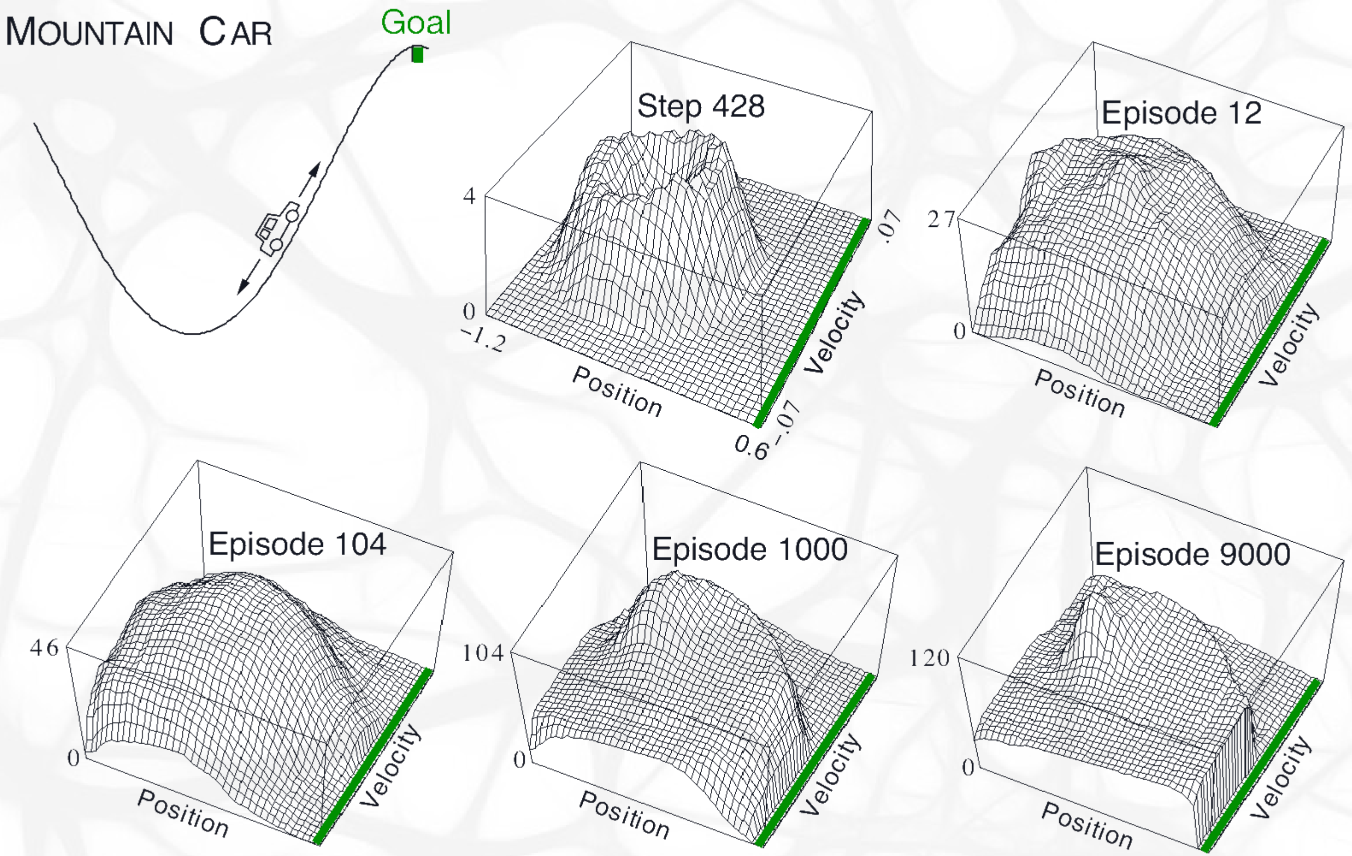


Figure 10.1 of "Reinforcement Learning: An Introduction, Second Edition".

The performances are for semi-gradient Sarsa(λ) algorithm (which we did not talked about yet) with tile coding of 8 overlapping tiles covering position and velocity, with offsets of $(1, 3)$.

Mountain Car Example

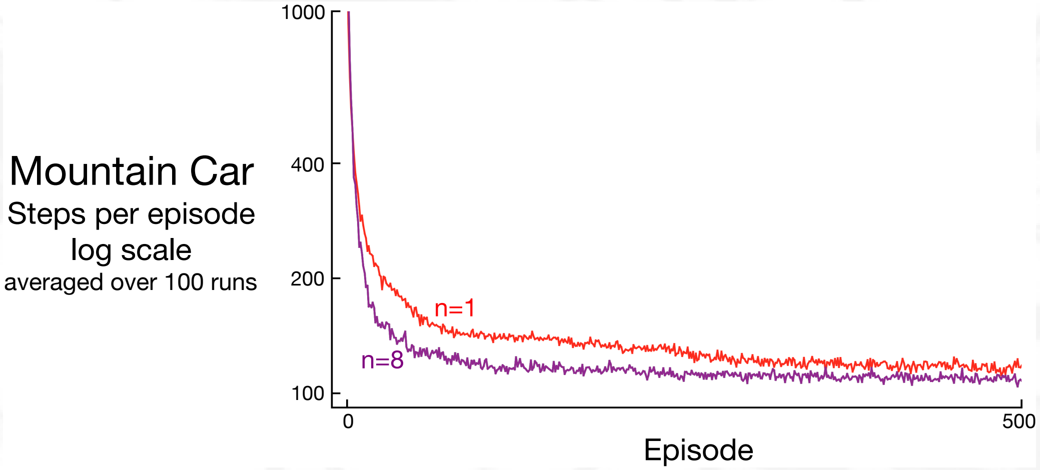


Figure 10.3 of "Reinforcement Learning: An Introduction, Second Edition".

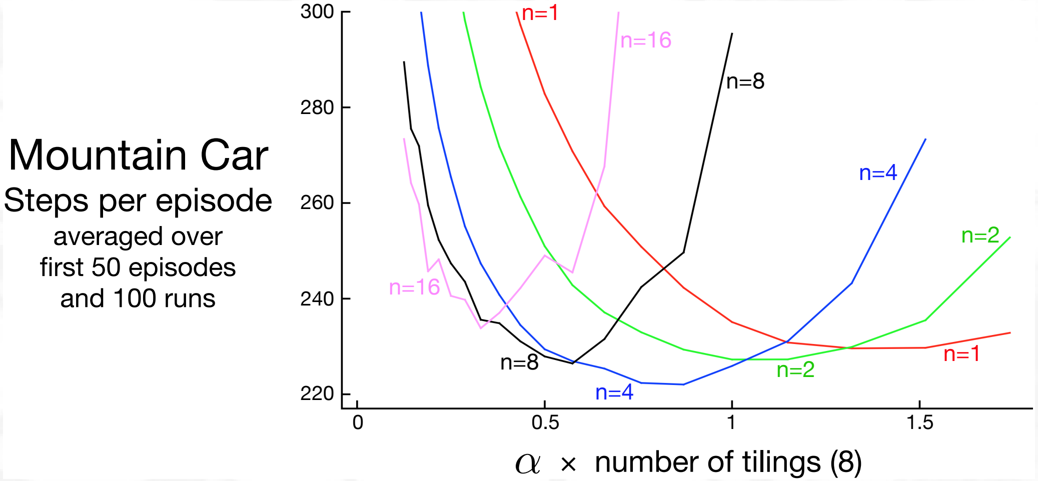


Figure 10.4 of "Reinforcement Learning: An Introduction, Second Edition".

Consider a deterministoc transition between two states whose values are computed using the same weight:

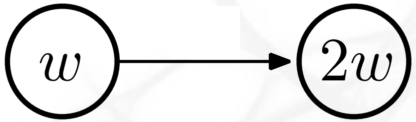
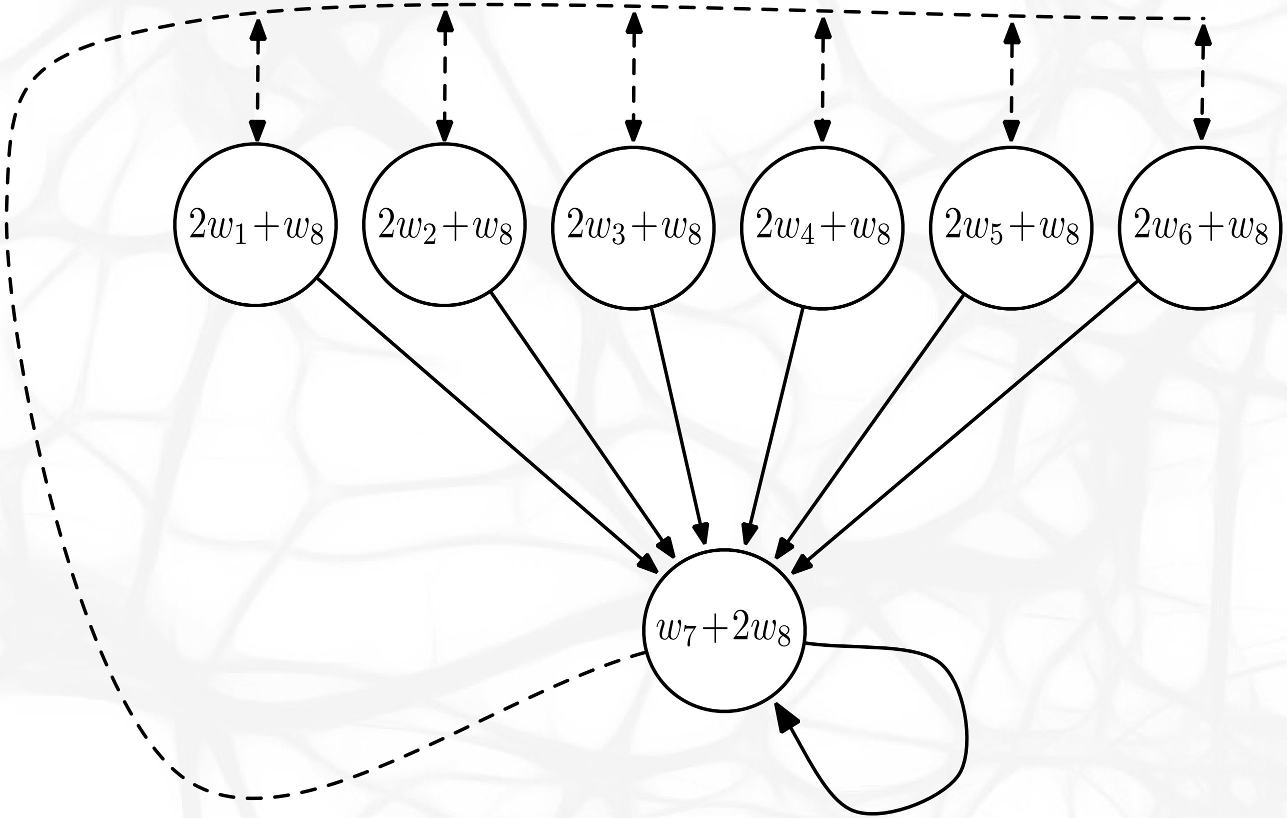


Figure from Section 11.2 of "Reinforcement Learning: An Introduction, Second Edition".

- If initially $w = 10$, TD error will be also 10 (or nearly 10 if $\gamma < 1$).
- If for example $\alpha = 0.1$, w will be increased to 1 (by 10%).
- This process can continue indefinitely.

However, the problem arises only in off-policy setting, where we do not decrease value of the second state from further observation.

The previous idea can be realized for instance by the following example.



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

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

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

$\gamma = 0.99$

Figure 11.1 of "Reinforcement Learning: An Introduction, Second Edition".

Off-policy Divergence With Function Approximation

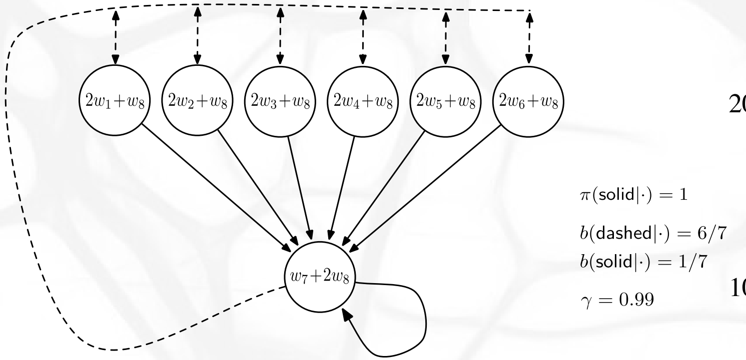


Figure 11.1 of "Reinforcement Learning: An Introduction, Second Edition".

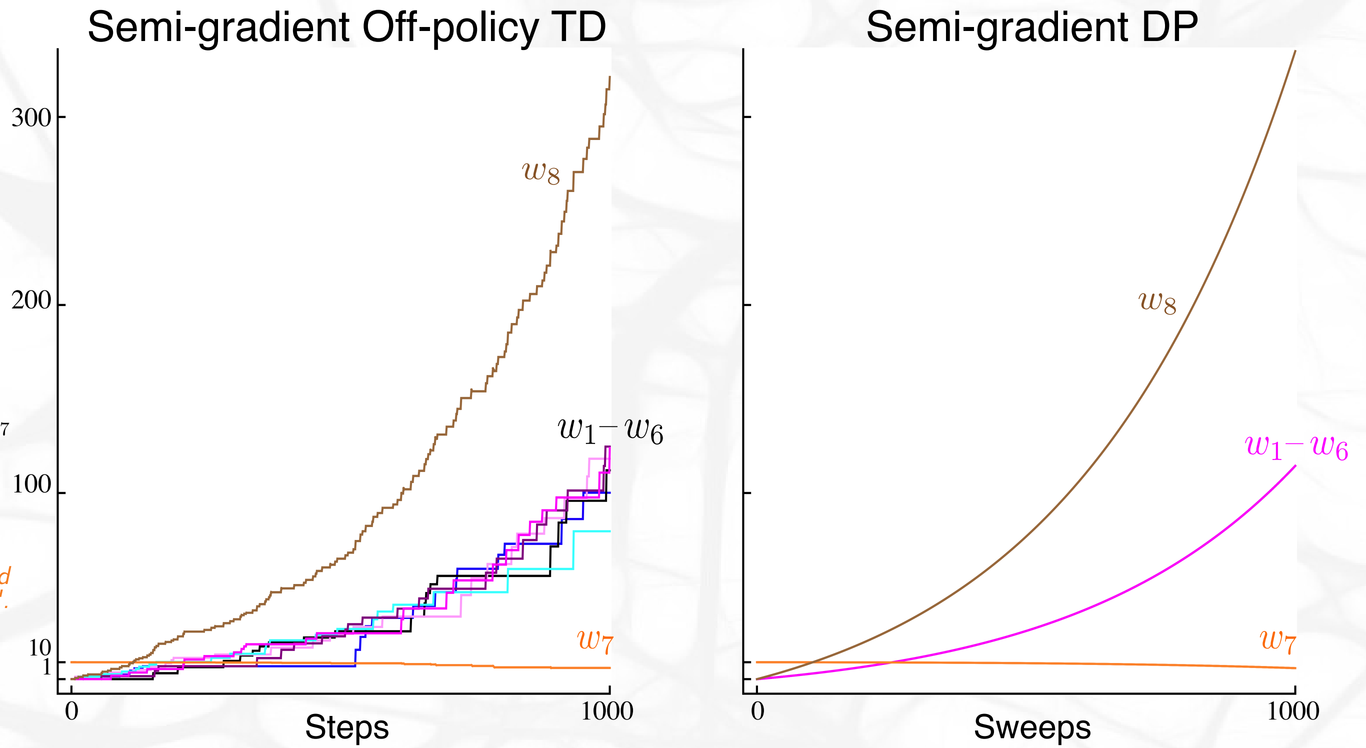


Figure 11.2 of "Reinforcement Learning: An Introduction, Second Edition".

Deep Q Networks

Volodymyr Mnih et al.: *Playing Atari with Deep Reinforcement Learning* (Dec 2013 on arXiv).

In 2015 accepted in Nature, as *Human-level control through deep reinforcement learning*.

Off-policy Q-learning algorithm with a convolutional neural network function approximation of action-value function.

Training can be extremely brittle (and can even diverge as shown earlier).

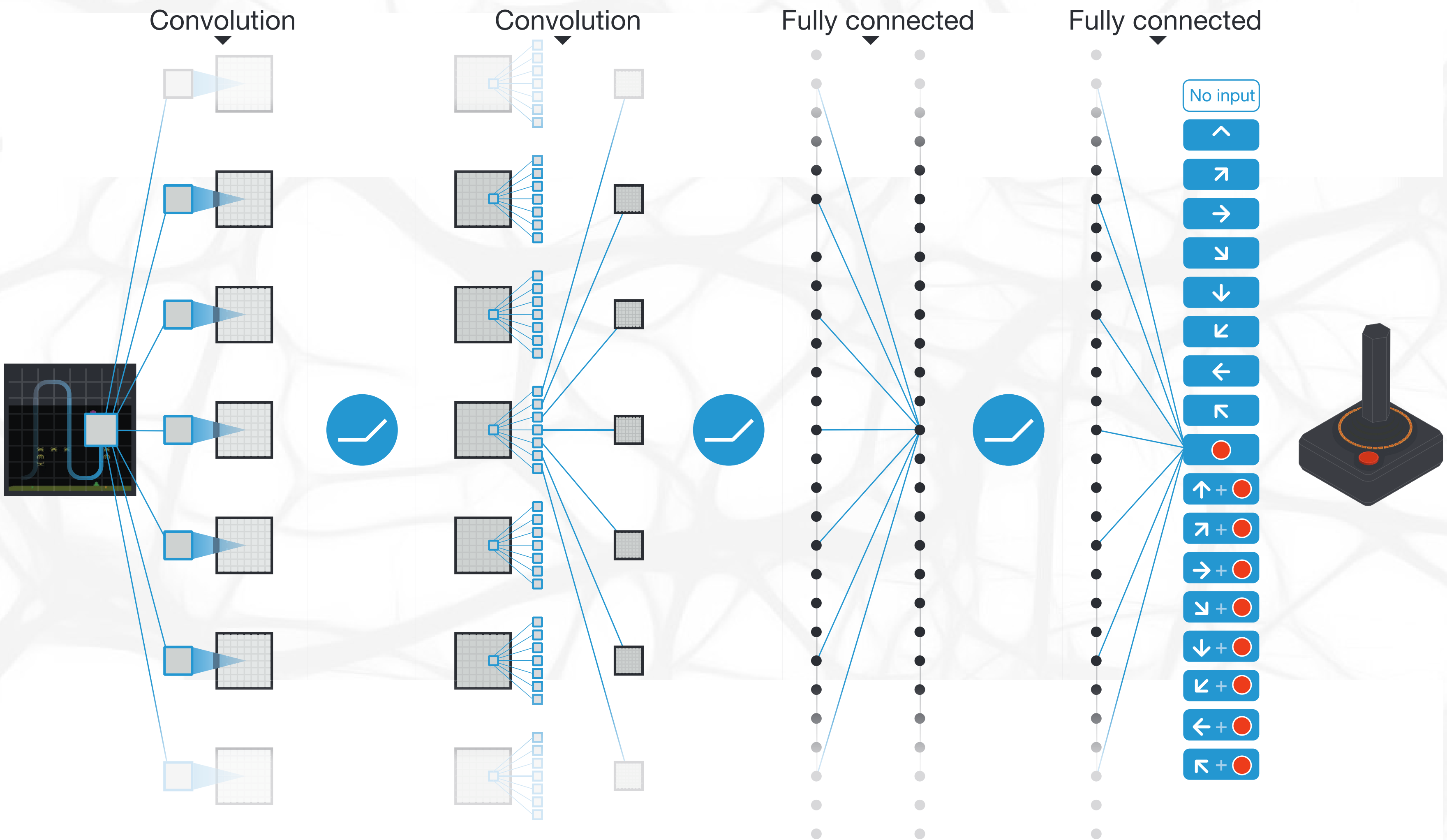


Figure 1 of the paper "Human-level control through deep reinforcement learning" by Volodymyr Mnih et al.

Deep Q Network

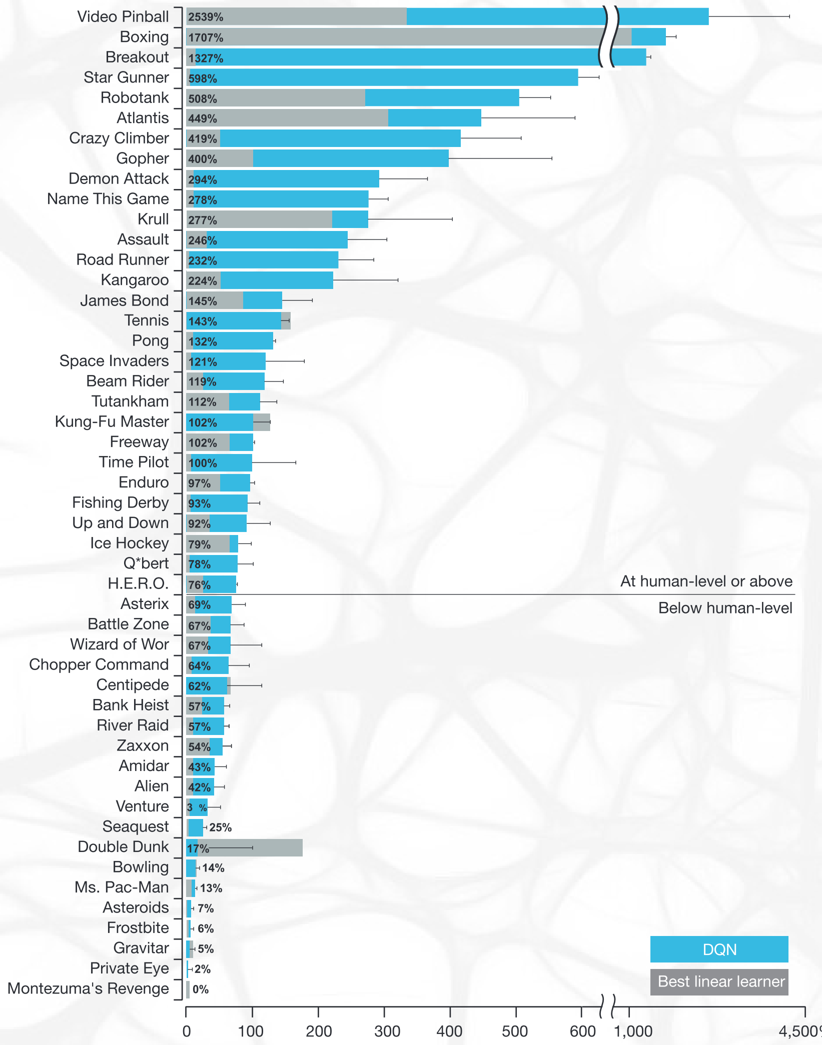
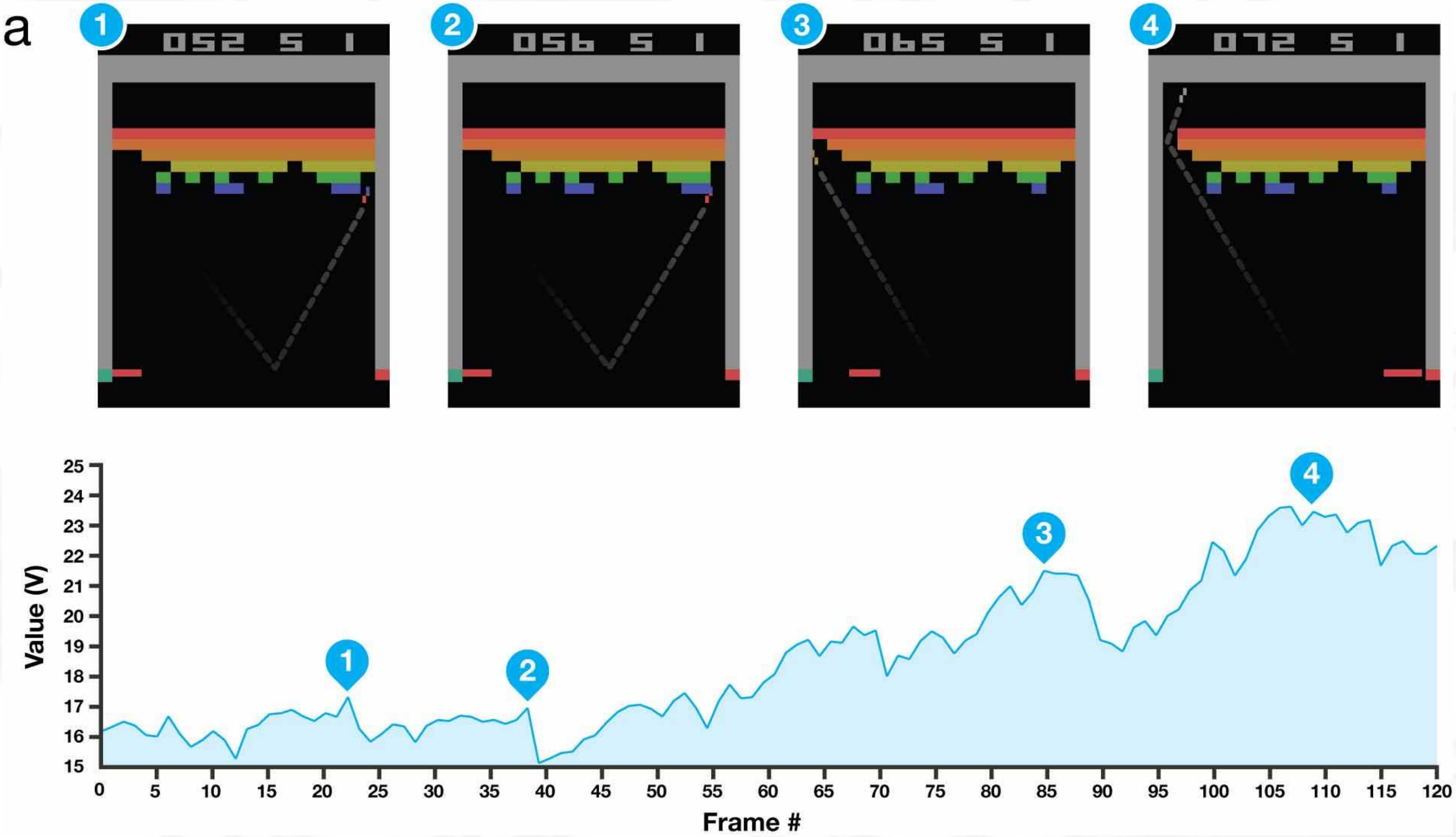


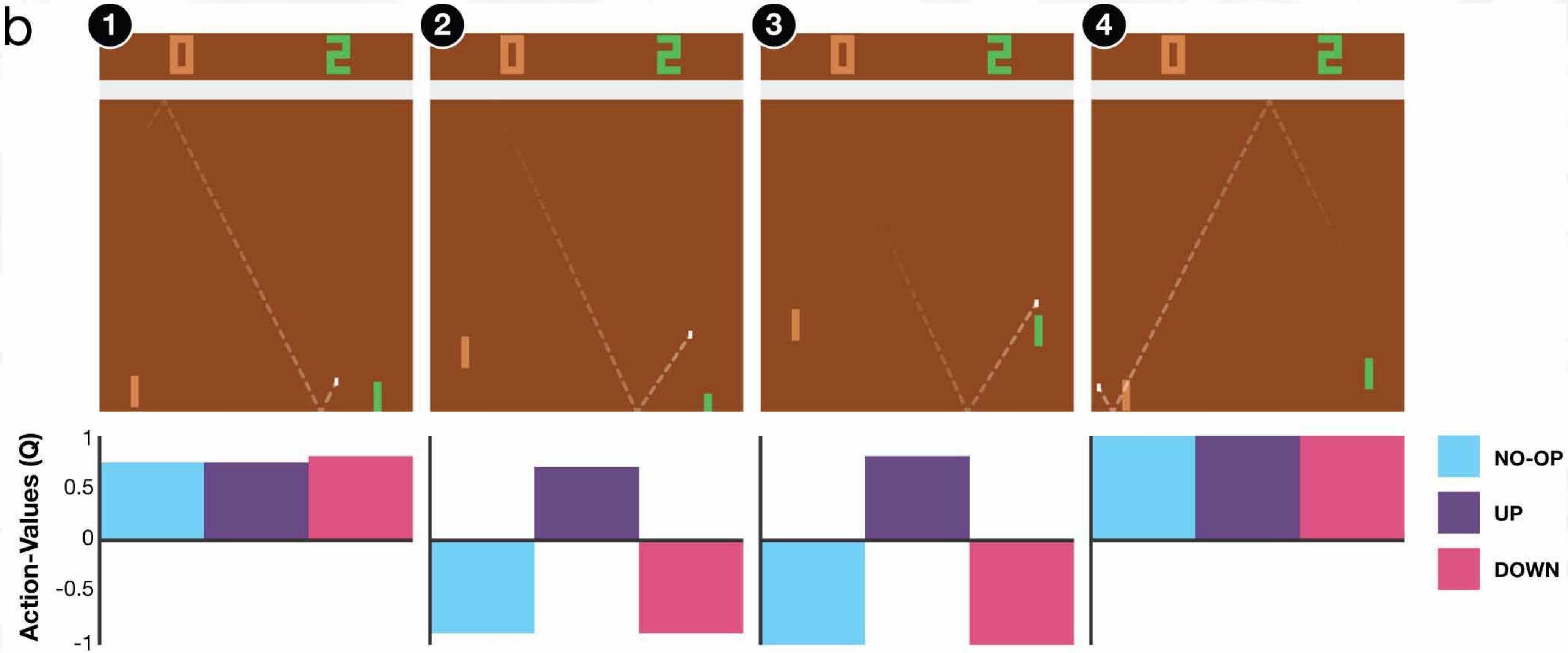
Figure 3 of the paper "Human-level control through deep reinforcement learning" by Volodymyr Mnih et al.

Deep Q Network



Extended Data Figure 2a of the paper "Human-level control through deep reinforcement learning" by Volodymyr Mnih et al.

Deep Q Network



Extended Data Figure 2b of the paper "Human-level control through deep reinforcement learning" by Volodymyr Mnih et al.

Deep Q Networks

- Preprocessing: 210×160 128-color images are converted to grayscale and then resized to 84×84.
- Frame skipping technique is used, i.e., only every 4^{th} frame (out of 60 per second) is considered, and the selected action is repeated on the other frames.
- Input to the network are last 4 frames (considering only the frames kept by frame skipping), i.e., an image with 4 channels.
- The network is fairly standard, performing
  - 32 filters of size 8×8 with stride 4 and ReLU,
  - 64 filters of size 4×4 with stride 2 and ReLU,
  - 64 filters of size 3×3 with stride 1 and ReLU,
  - fully connected layer with 512 units and ReLU,
  - output layer with 18 output units (one for each action)

Deep Q Networks

- Network is trained with RMSProp to minimize the following loss:

$$\mathcal{L} \stackrel{\text{def}}{=} \mathbb{E}_{(s,a,r,s')\sim data}\left[\left(r + \gamma \max_{a'} Q(s', a'; \bar\theta) - Q(s, a; \theta)\right)^2\right].$$

- An ε-greedy behavior policy is utilized.

Important improvements:

- experience replay: the generated episodes are stored in a buffer as (s, a, r, s') quadruples, and for training a transition is sampled uniformly;
- separate target network $\bar\theta$: to prevent instabilities, a separate target network is used to estimate state-value function. The weights are not trained, but copied from the trained network once in a while;
- reward clipping of $(r + \gamma \max_{a'} Q(s', a'; \bar\theta) - Q(s, a; \theta))$ to $[-1, 1]$.

Deep Q Networks Hyperparameters

Hyperparameter	Value
minibatch size	32
replay buffer size	1M
target network update frequency	10k
discount factor	0.99
training frames	50M
RMSProp learning rate and momentum	0.00025, 0.95
initial ε, final ε and frame of final ε	1.0, 0.1, 1M
replay start size	50k
no-op max	30