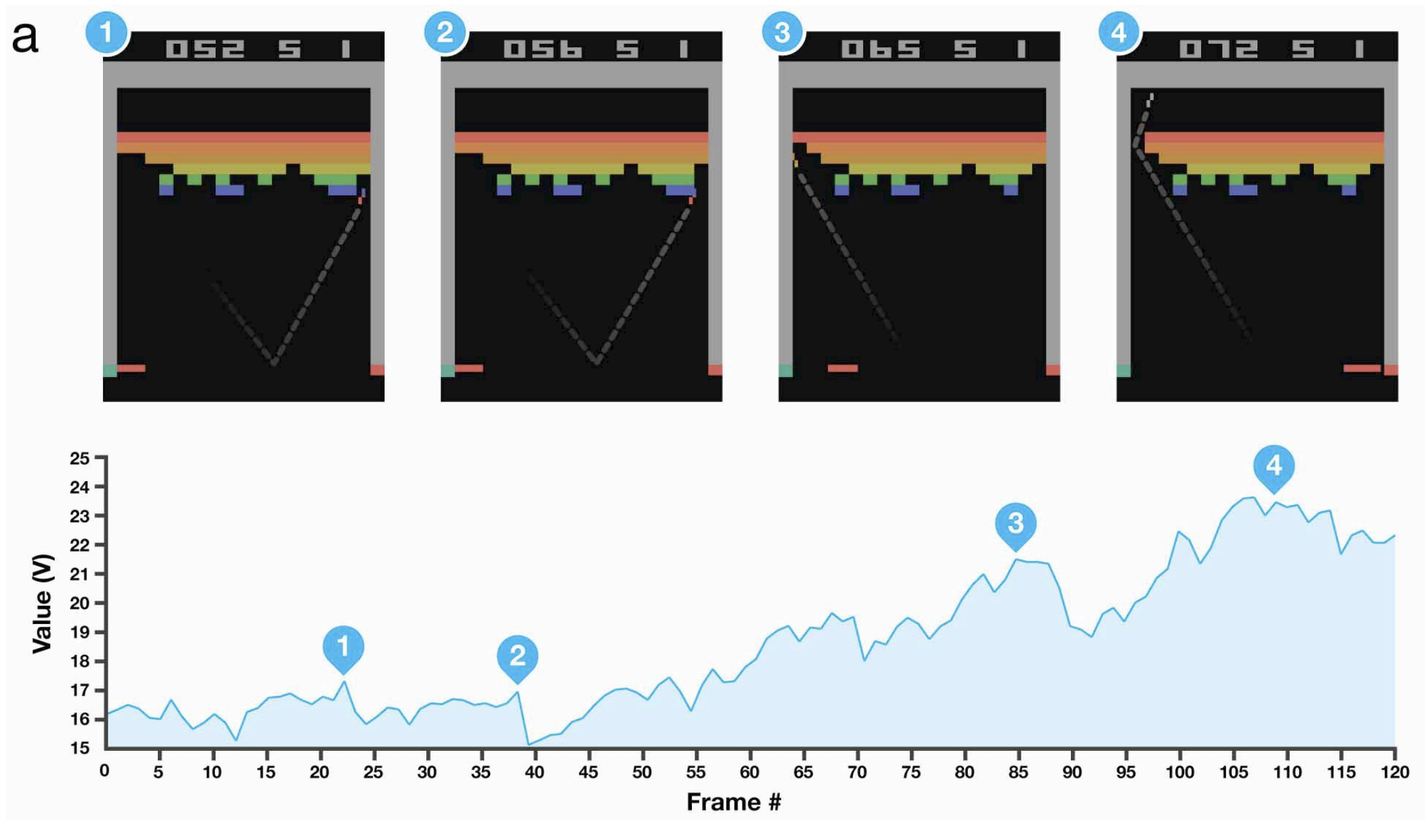
HSLU Lucerne University of Applied Sciences and Arts

Function Approximation Methods



Reinforcement Learning
November 24, 2022

Learning Objectives

- Motivation for using function approximation over tabular methods
- Understanding the objective that should be learned
- Stochastic gradient descent and semi-gradient algorithms
- Understanding Sarsa and Q-Learning with function approximation

Tabular methods

So far, we have used tabular methods:

- Value Functions (and sometimes policies) were stored in a table for each state or state-action

Problems:

- State spaces are arbitrarily large in many RL methods
- State spaces and/or action spaces might be continuous and not discrete
- Many states will not be encountered, we need to **generalize** from other states, i.e. we would like to have a resonable response from a trained agent, even if a state has never been encountered

Function Approximation

- We want to approximate the state-values or the state-action-values by a function that is parametrized by some parameter w

$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$$

$$\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$$

- The dimensionality d of w will generally be much lower than the dimensionality of the state space

$$d \ll |\mathcal{S}|$$

Target Update

Different (tabular) RL methods have different target functions:

- MC Target: G_t
- TD(0) Target: $R_{t+1} + \gamma V(S_{t+1})$
- n-step TD Target: $G_{t:t+n}$

Each update moves the value function in direction of the target

$$V(S_t) \leftarrow V(S_t) + \alpha[Target - V(S_t)]$$

Updates

Tabular Methods:

- Updates for state values were calculated directly to get a better estimate of the target

Function Approximation:

- Each update can be viewed as a training example
- We can then use **supervised learning** on those training examples

However: in RL the updates must be online as the agents learns while interacting with the environment

An agent must be able to learn efficiently from incrementally acquired data

Prediction Objective

As an objective, we would like to learn to approximate the value function at each state:

$$[v_\pi(s) - \hat{v}(s, \mathbf{w})]^2$$

However, an update to one state, will affect the other states

We want to intruduce a function $\mu(s)$ that measures how important each error is and then minimize the mean weighted error between the value function and its approximation

$$\overline{\mathrm{VE}}(\mathbf{w}) = \sum_{s \in \mathcal{S}} \mu(s)[v_\pi(s) - \hat{v}(s, \mathbf{w})]^2$$

Prediction Objective: Caveats

(It is not completely clear that VE is the best objective to optimize for RL:

- The purpose of finding the value function, is to find a better policy
- The best value function for that, might not be the best for minimizing VE.
- Further more, a *global optimum* in minimizing VE is unlikely to be found)

On-policy distribution

$\mu(s)$ can be calculated as the fraction of how much time is spent in s, i.e. how often is s visited

in on-policy methods, this is called the on-policy distribution, for episodic tasks, that also depends on how the starting states are chosen, let $h(s)$ be the probability of choosing starting state s, then $\eta(s)$ is the average time spent in s in a single episode

$$\eta(s) = h(s) + \sum_{\bar{s}} \eta(s) \sum_{a} \pi(a|\bar{s}) p(s|\bar{s}, a), \quad \text{for all } s \in \mathcal{S}$$

$$\mu(s) = \frac{\eta(s)}{\sum_{s'} \eta(s')} \quad \text{for all } s \in \mathcal{S}$$

Stochastic gradient descent

Assume that in each step we observe a state s and its (true) value under the policy.

We can use stochastic gradient-descend (SGD) by adjusting the weights vector to minimize the error in the observed examples

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

Monte Carlo Prediction

Gradient Monte Carlo Prediction

Input:
  a policy π
  a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$ with parameters $\mathbf{w}$
  a step size parameter $\alpha > 0$
Initialize:
  $\mathbf{w}$ arbitrarily
Loop forever:
  Generate episode following π: $S_0, A_0, R_0, S_1, ..., R_T$
  $G \leftarrow 0$
  Loop for each step of the episode, $t = T-1, T-2, ..., 0$:
    $G \leftarrow \gamma G + R_{t+1}$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha[G_t - \hat{v}(S_t, \mathbf{w})]\nabla\hat{v}(S_t, \mathbf{w})$

Semi Gradient Methods

Bootstrapping methods are not true gradient descent methods, as the *targets* themselves depend on the weights w

They only contain part of the gradient and are therefor called *semi-gradient* methods

However, the calculation works just as before....

TD(0) Prediction

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

Input:
  a policy π
  a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$ with parameters $\mathbf{w}$
  ($\hat{v}(\text{terminal}, \cdot)$) must be 0)
  a step size parameter $\alpha > 0$
Initialize:
  $\mathbf{w}$ arbitrarily

Loop for each episode)
  Initialize S
  Loop for each step of the episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha[R + \gamma\hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})]\nabla\hat{v}(S, \mathbf{w})$
    $S \leftarrow S'$
  until S is terminal

Episodic Semi-gradient Control

For control, we want to approximate action-value functions

$$\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$$

The update target is again any approximation of q, for example for (1-step) Sarsa:

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

Episodic Semi-gradient Control

- Generalized policy iteration (GPI) can be used again
- For on policy methods a soft approximation to the greedy policy can be used for policy improvement
- If the action set is discrete and not too large, the greedy action can be calculated directly

$$A^*_{t+1} = \arg\max_a \ \hat{q}(S_{t+1}, a, w_t)$$

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

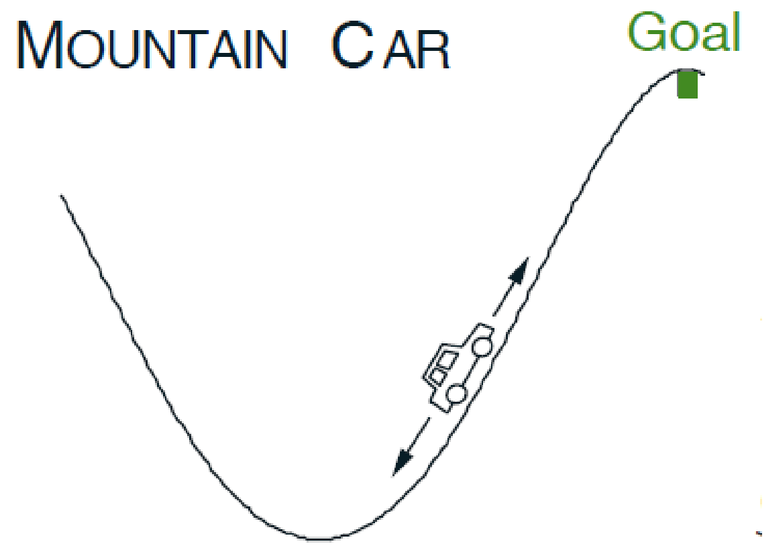
Input:
a differentiable function $\hat{q} : \mathcal{S} \times A \times \mathbb{R}^d \to \mathbb{R}$ with parameters $\mathbf{w}$
a step size parameter $\alpha > 0$, small $\epsilon > 0$
Initialize:
$\mathbf{w}$ arbitrarily

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

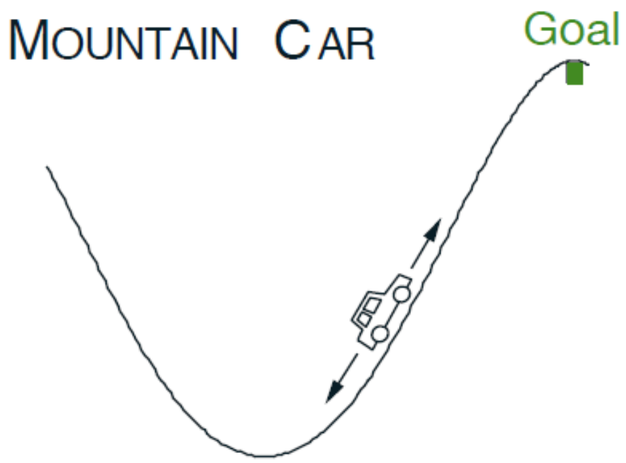
Mountain Car Example

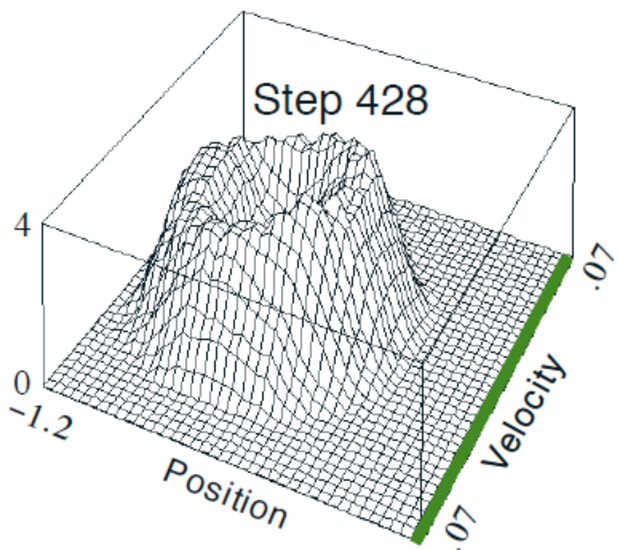
The car wants to move up the mountain to reach the goal, but the gravity is larger than its acceleration

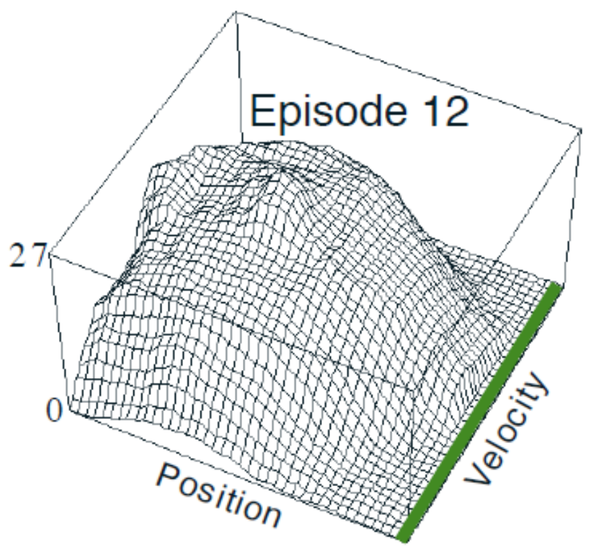
- 3 Actions: Full throttle forward, full throttle backwards or no throttle
- Reward is -1 for each time step (0 for reaching goal)
- Each episode starts at random position (with velocity 0)

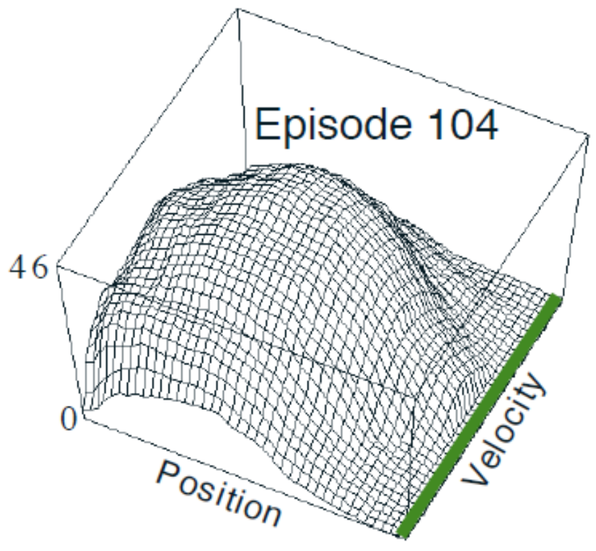


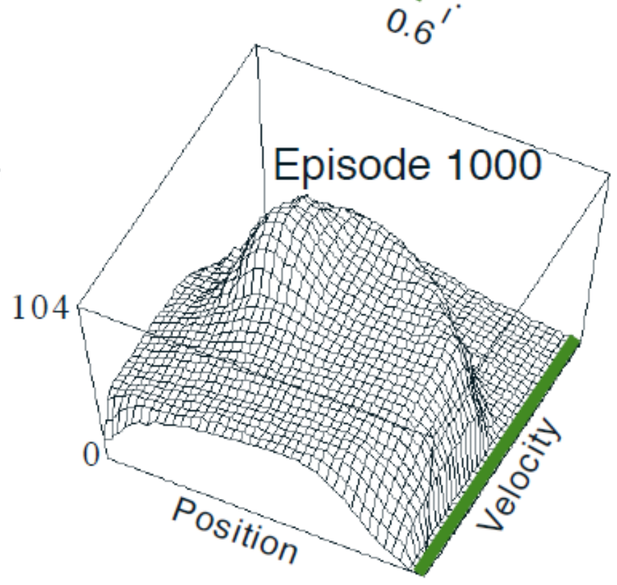
Mountain Car Example

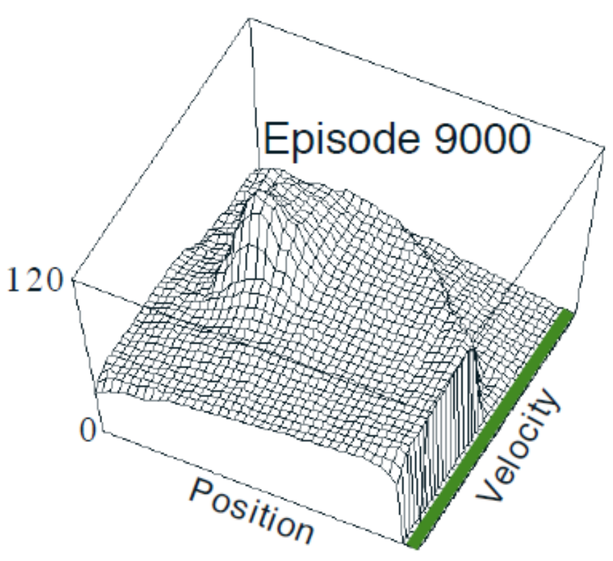






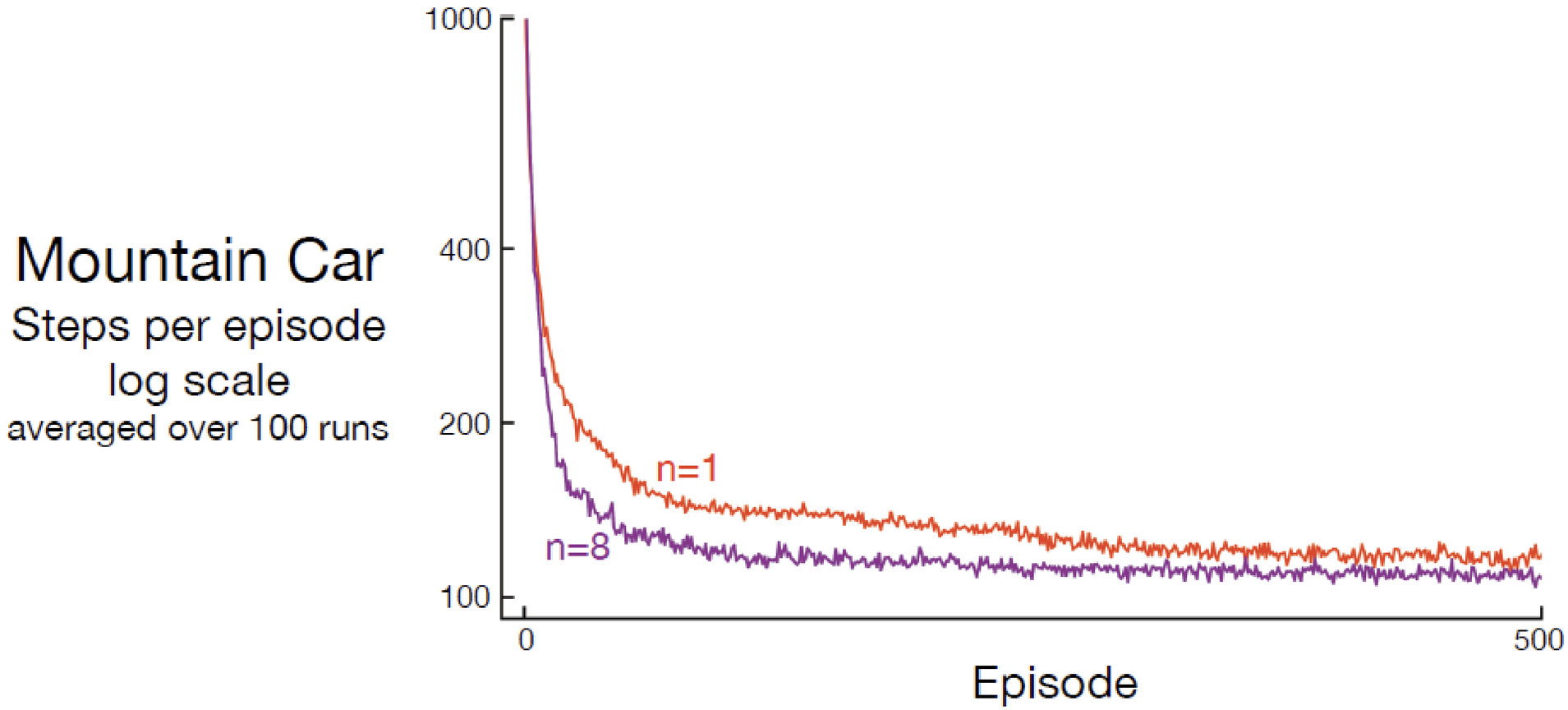




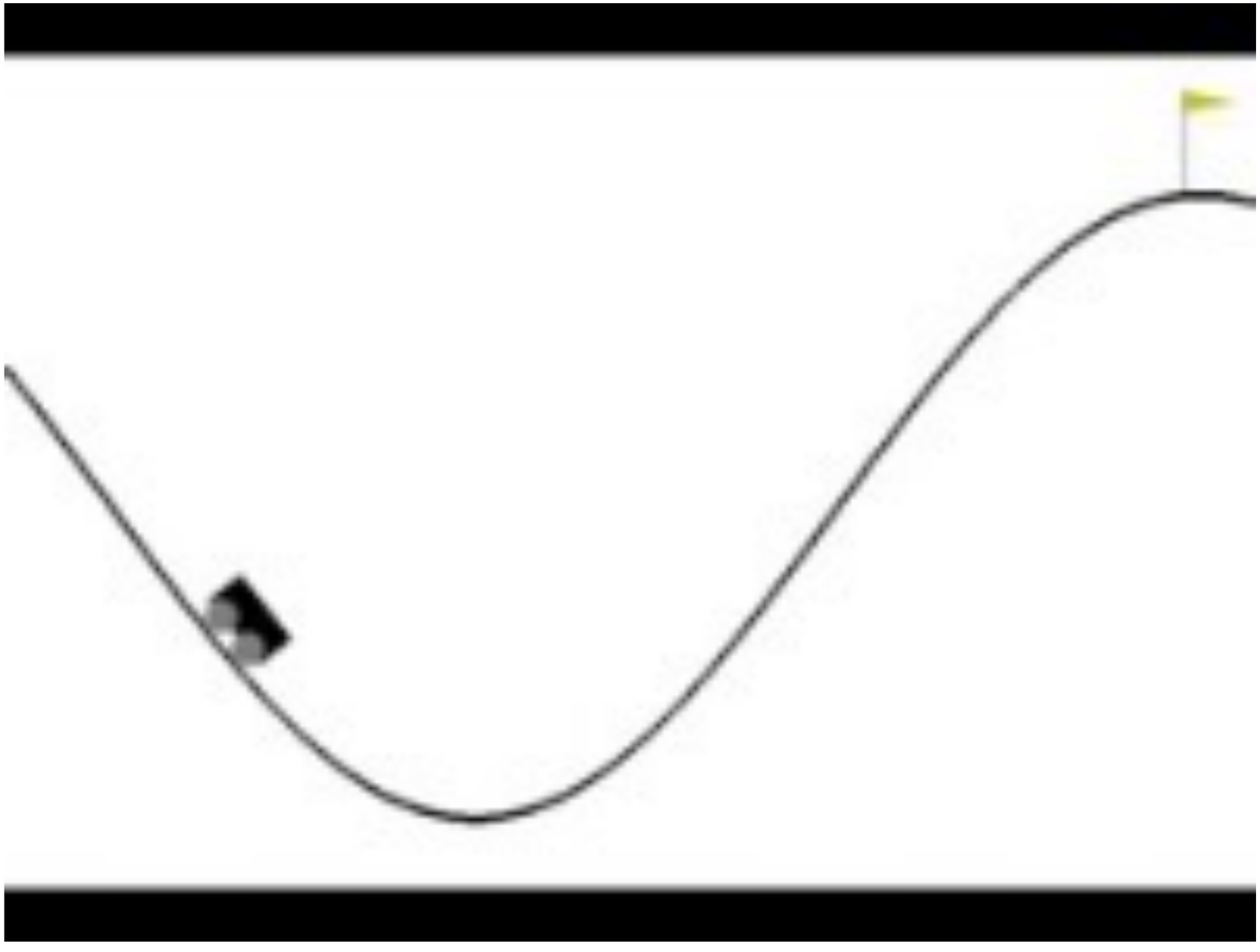


$$-\max_a \ \hat{q}(s, a, \mathbf{w})$$

Mountain Car Example



1-step vs. 8-step semi-gradient Sarsa



Off-Policy Learning

Off-policy learning can be unstable or even diverge with nonlinear function approximation:

- Correlations in the sequence of observations
- Small updates to q may significantly change the policy
- Correlations between the action values q and the target values

Deep Q-Learning

Playing Atari with Deep Reinforcement Learning

Volodymyr Mnih Koray Kavukcuoglu David Silver Alex Graves Ioannis Antonoglou

Daan Wierstra Martin Riedmiller

DeepMind Technologies

Human-level control through deep reinforcement learning

Volodymyr Mnih[1]*, Koray Kavukcuoglu[1]*, David Silver[1]*, Andrei A. Rusu[1], Joel Veness[1], Marc G. Bellemare[1], Alex Graves[1], Martin Riedmiller[1], Andreas K. Fidjeland[1], Georg Ostrovski[1], Stig Petersen[1], Charles Beattie[1], Amir Sadik[1], Ioannis Antonoglou[1], Helen King[1], Dharshan Kumaran[1], Daan Wierstra[1], Shane Legg[1] & Demis Hassabis[1]

Deep Q-Learning

Deep Q-Learning uses a deep Q network (DQN), that estimates the action value function

- Uses experience replay
- Updates the action-values iteratively towards target
- Target values are only updated periodically

$$J(\mathbf{w}_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\Big[(r + \gamma \max_a \; Q(s', a', \mathbf{w}_i^-) - Q(s, a, \mathbf{w}_i)^2\Big]$$

Deep Q-Learning

$$J(\mathbf{w}_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\Big[(r + \gamma \max_a \; Q(s', a', \mathbf{w}_i^-) - Q(s, a, \mathbf{w}_i)^2\Big]$$

- The target network parameters $\mathbf{w}_i$ are only updated with the Q-network parameters every C steps
- The network is actually trained to learn

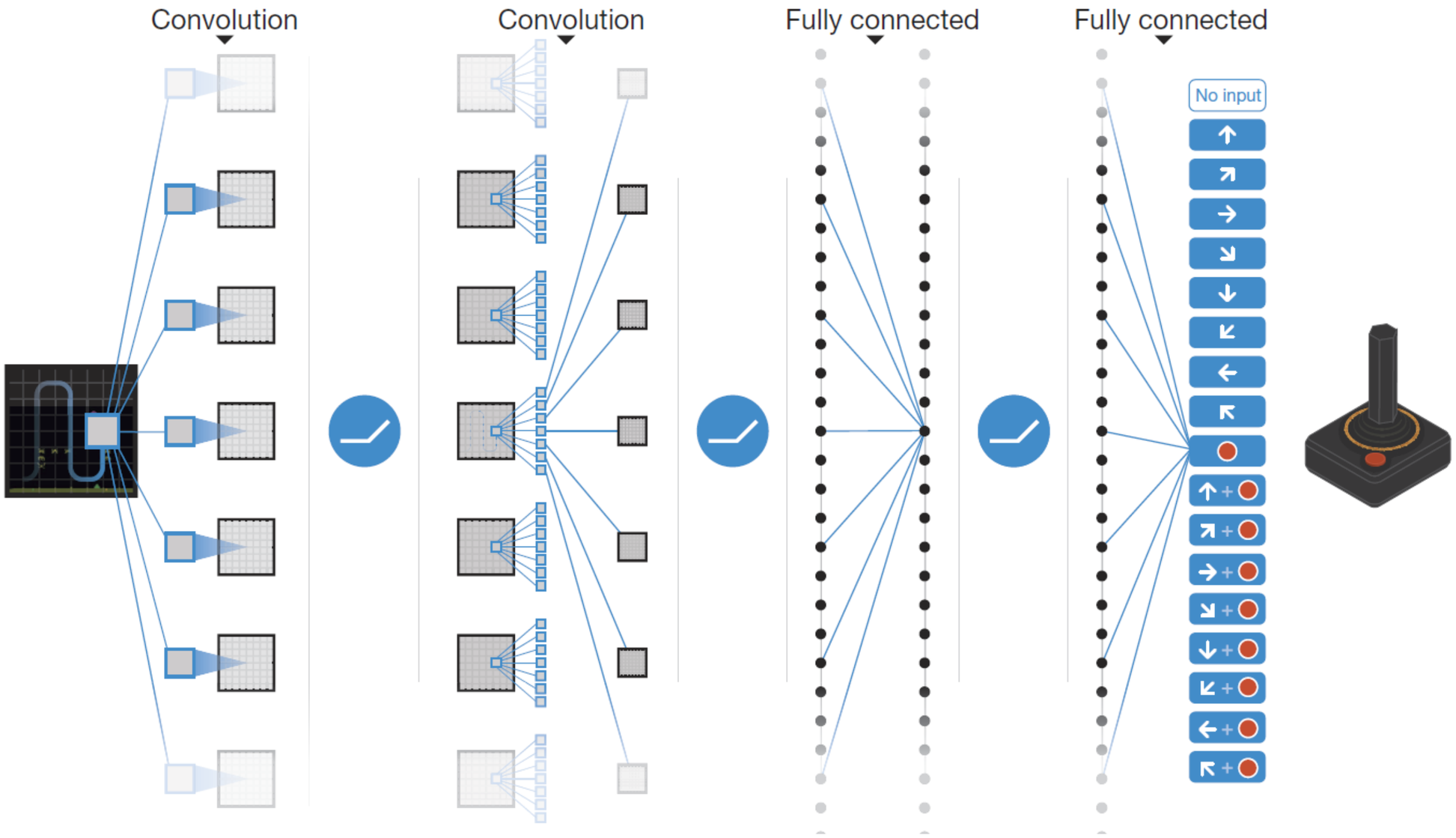
$$q(a|s) = Q(s, \mathbf{w})$$

i.e. it returns the q-value of each action in s given s.

Deep Q-Learning

- For the Atari games, the input was the actual image of the game, so a convolutional neural network was used.
- The images are 210x160 pixels and have been converted to grey-value images, down-sampled and cropped to 84x84
- Input are the last 4 frames
- Rewards are the score of the game (normalized and clipped at +1,-1)
- A new action is calculated every 4-th frame and repeated in between frames

Deep Q-Learning: Network Configuration



Batch Methods

- For function approximation with neural networks, it is not efficient to update the network after one step
- Furthermore, the sample is only used once
- In supervised learning approaches with neural networks, the training goes over many episodes of the same data

- Solution: Collect a batch of experiences and store them in a buffer to use multiple times
- This is called **Experience Replay**

Experience Replay

The experience at each time step is stored into a data set

$$e_t = (s_t, a_t, r_t, s_{t+1})$$

$$\mathcal{D} = e_1, \ldots e_N$$

- A fixed size buffer is used that overwrites the oldest experience
- Training data is then sampled from the data set

Advantages:

- Each experience can be used in many updates
- Learning from consecutive samples would be inefficient as they are correlated

Deep Q-Learning with experience replay

Initialize:

the replay memory D to capacity N

action-value function Q with random weights $\mathbf{w}$

target action-value function $\hat{Q}$ with weights $\mathbf{w}^- = \mathbf{w}$

Loop for each episode:

Initialize S_1

For every step $t = 1, T$ in the episode:

Choose A_t as a function of $Q(S_t, ., \mathbf{w})$ (e.g., ϵ-greedy)

Take action A_t, observe R_t, S_{t+1}

Store transition (S_t, A_t, R_r, S_{t+1}) in D

Sample a random minibatch of transitions (S_j, A_j, R_j, S_{j+1}) from D

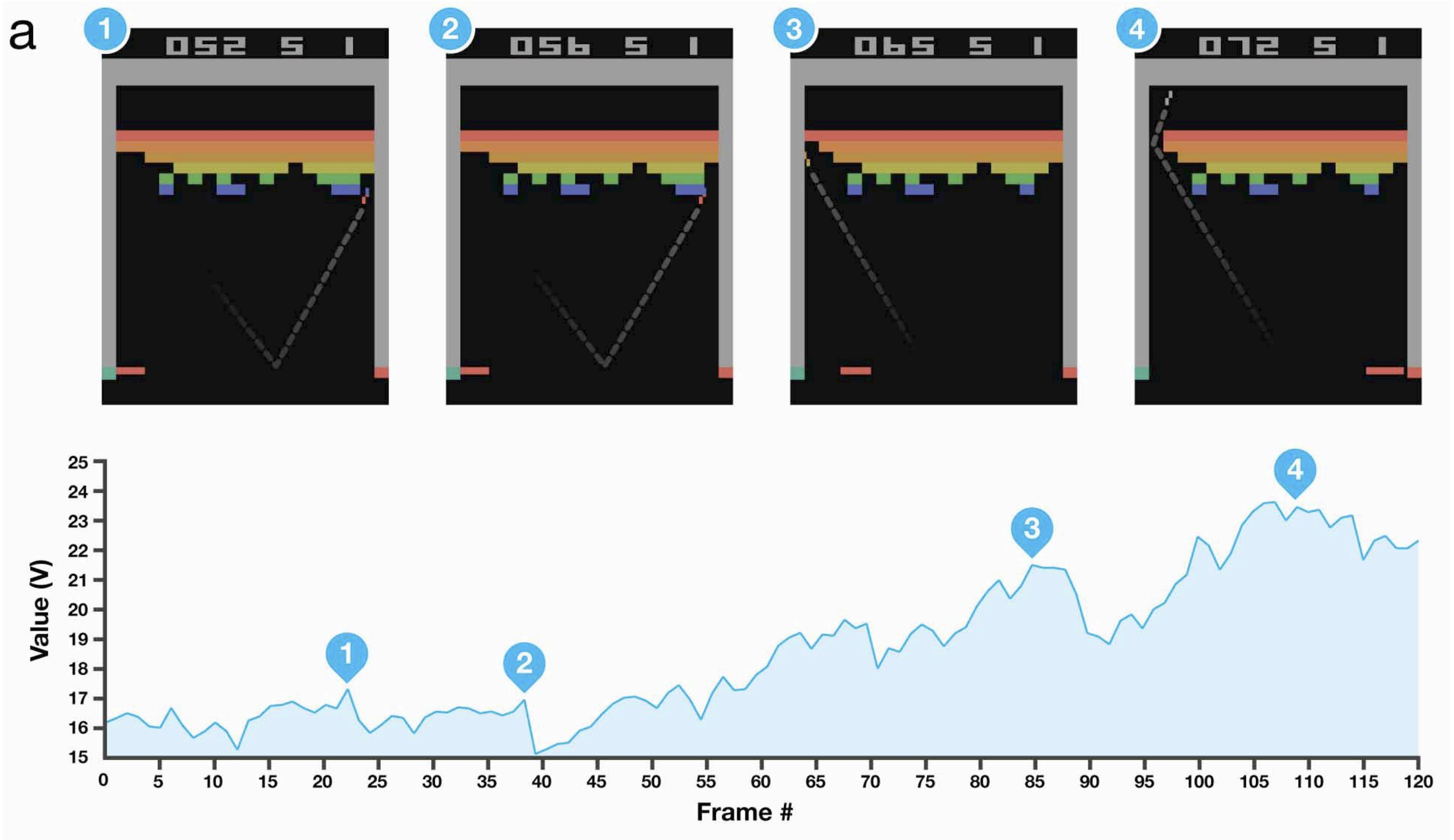
$$y_j = \begin{cases} R_j & \text{if } S_{j+1} \text{ is terminal} \\ R_j + \gamma \max_{A'} \hat{Q}(S_{j+1}, A', \mathbf{w}^-) & \text{otherwise} \end{cases}$$

Perform a gradient step on $(y_j - Q(S_j, A_j, \mathbf{w}))^2$ with respect to $\mathbf{w}$

Every C steps reset $\hat{Q} = Q$

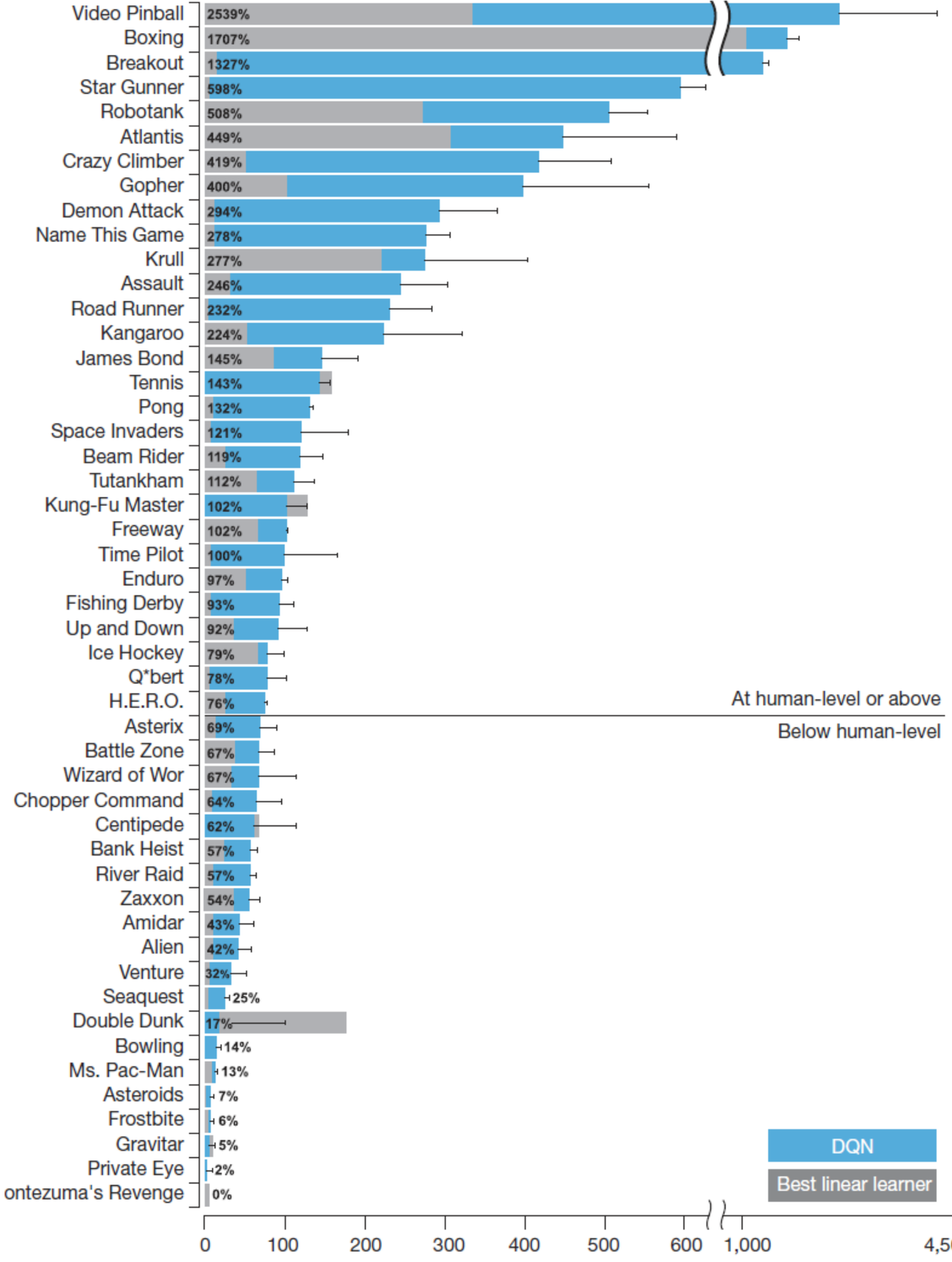
Example: Breakout

Example: Breakout



Results on Atari Games

Comparison of the DQN agent with other reinforcement learning methods and human game tester



Prioritized Experience Replay

Idea:

- An agent should be able to learn more efficiently from some transitions than from others
- By what criterion can we measure the importance of each transition?

TD error indicates how surprising or unexpected the transition is, so samples with a high TD error should be prioritized

However:

- Only using the prioritized transitions can lead to errors too
- Use stochastic sampling

Prioritized Experience Replay

The probability of sampling a transition i is

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

where p_i is the priority of the transition i

Two approaches for calculating p_i:

$$p(i) = |\delta_i| + \epsilon \qquad\qquad p(i) = \frac{1}{\text{rank}(i)}$$

Double Q-Learning

- Q-learning is known to overestimate action-values under certain conditions
- The Q function is used to select the best action and to evaluate it:

$$y_j^{\mathrm{DQN}} \doteq R_j + \gamma Q(S_{j+1}, \arg\max_a Q(S_{j+1}, a, \mathbf{w}^-), \mathbf{w}^-)$$

- it can be shown it is better to use different networks for those two tasks:

$$y_j^{\mathrm{DoubleDQN}} \doteq R_j + \gamma Q(S_{j+1}, \arg\max_a Q(S_{j+1}, a, \mathbf{w}), \mathbf{w}^-)$$

Conclusion

- The methods seen so far, DP, MC and TD Learning can be adapted to function approximation methods, where the state- or action-value function is approximated
- The object is to minimize the **mean squared error** between the function approximation and the target function
- Minimization is done using **stochastic gradient descent**
- Deep Q-Learning Methods use a **neural network** for function approximation and can tackle difficult RL problems by only using *raw* observation as input (no feature design)
- Training time is generally long
- In order to speed up training, replay buffers are used