Function Approximation, Deep Q Network, Rainbow

Milan Straka

March 12, 2025

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

Refresh

Function Approximation

We approximate value function v and/or action-value function q, selecting it 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 the on-policy distribution.

Function Approximation

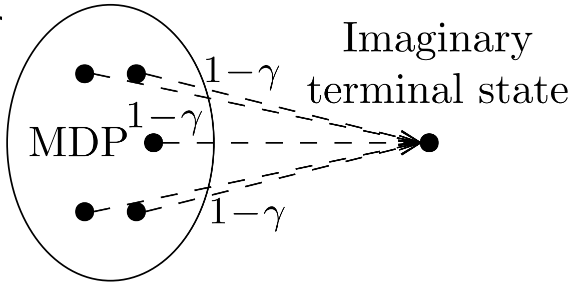
The on-policy distribution is defined as:

- For **episodic tasks**, let $h(s)$ be the probability that an episodes starts in state s, and let $\eta(s)$ denote the number of time steps spent, on average, in state s in a single episode:

$$\eta(s) = h(s) + \sum\nolimits_{s'} \eta(s') \sum\nolimits_a \pi(a|s')p(s|s', a).$$

  The on-policy distribution is then obtained by normalizing: $\mu(s) \overset{\text{def}}{=} \frac{\eta(s)}{\sum_{s'} \eta(s')}$.

  If there is discounting ($\gamma < 1$), it should be treated as a form of termination, by including a factor γ to the second term of the $\eta(s)$ equation.



- For **continuing tasks**, we require $\gamma < 1$, and employ the same definition as in the episodic case.

Gradient and Semi-Gradient Methods

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

$$\begin{aligned} \boldsymbol{w}_{t+1} &\leftarrow \boldsymbol{w}_t - \tfrac{1}{2}\alpha\nabla_{\boldsymbol{w}_t}\big(v_\pi(S_t) - \hat v(S_t;\boldsymbol{w}_t)\big)^2 \\ &\leftarrow \boldsymbol{w}_t + \alpha\big(v_\pi(S_t) - \hat v(S_t;\boldsymbol{w}_t)\big)\nabla_{\boldsymbol{w}_t}\hat v(S_t;\boldsymbol{w}_t). \end{aligned}$$

As usual, the $v_\pi(S_t)$ is estimated by a suitable sample of a return:

- in Monte Carlo methods, we use episodic return G_t,
- in temporal difference methods, we employ bootstrapping and use one-step return

$$R_{t+1} + [\neg\mathrm{done}]\cdot\gamma\hat v(S_{t+1};\boldsymbol{w})$$

  or an n-step return.

Monte Carlo Gradient Policy Evaluation

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

```
Input: the policy $\pi$ to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \rightarrow \mathbb{R}$
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 $\pi$
    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".

If the return estimate G_t is unbiased (which it is in a Monte Carlo method), the policy evaluation algorithm is guaranteed to converge to a local optimum of the mean squared value error under the usual SGD conditions.

Gradient and Semi-Gradient Methods

Temporal Difference Semi-Gradient Policy Evaluation

In TD methods, we again bootstrap the estimate $v_\pi(S_t)$ as $R_{t+1} + [\neg\text{done}] \cdot \gamma\hat v(S_{t+1}; \boldsymbol{w})$.

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

```
Input: the policy $\pi$ 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".

Why Semi-Gradient TD

Note that the above algorithm is called **semi-gradient**, because it does not backpropagate through $\hat{v}(S_{t+1}; \boldsymbol{w})$:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha\Big(R_{t+1} + [\neg\text{done}] \cdot \gamma\hat{v}(S_{t+1}; \boldsymbol{w}) - \hat{v}(S_t; \boldsymbol{w})\Big)\nabla_{\boldsymbol{w}}\hat{v}(S_t; \boldsymbol{w}).$$

In other words, the above rule is in fact not an SGD update, because there does not exist a sufficiently continuous function $J(\boldsymbol{w})$, for which we would get the above update.

To sketch a proof, consider a linear $\hat{v}(S_t; \boldsymbol{w}) = \sum_i x(S_t)_i w_i$ and assume such a $J(\boldsymbol{w})$ exists. Then

$$\frac{\partial}{\partial w_i} J(\boldsymbol{w}) = \Big(R_{t+1} + \gamma\hat{v}(S_{t+1}; \boldsymbol{w}) - \hat{v}(S_t; \boldsymbol{w})\Big)x(S_t)_i.$$

We now verify that the second derivatives are not equal, which is a contradiction with the Schwarz's theorem (stating that partial derivatives commute as long as they are differentiable):

$$\frac{\partial}{\partial w_i}\frac{\partial}{\partial w_j} J(\boldsymbol{w}) = \Big(\gamma x(S_{t+1})_i - x(S_t)_i\Big)x(S_t)_j = \gamma x(S_{t+1})_i x(S_t)_j - x(S_t)_i x(S_t)_j$$
$$\frac{\partial}{\partial w_j}\frac{\partial}{\partial w_i} J(\boldsymbol{w}) = \Big(\gamma x(S_{t+1})_j - x(S_t)_j\Big)x(S_t)_i = \gamma x(S_{t+1})_j x(S_t)_i - x(S_t)_i x(S_t)_j$$

Gradient and Semi-Gradient Methods

Note that “fixing” the algorithm by allowing to backpropagate through the bootstrap estimate $R_{t+1} + \hat{v}(S_{t+1}; \boldsymbol{w})$ would not work at all. If we consider such an update

$$\begin{aligned}\boldsymbol{w}_{t+1} &\leftarrow \boldsymbol{w}_t - \tfrac{1}{2}\alpha\nabla_{\boldsymbol{w}_t}\big(R_{t+1} + \hat{v}(S_{t+1}; \boldsymbol{w}) - \hat{v}(S_t; \boldsymbol{w}_t)\big)^2 \\ &\leftarrow \boldsymbol{w}_t + \alpha\big(R_{t+1} + \hat{v}(S_{t+1}; \boldsymbol{w}) - \hat{v}(S_t; \boldsymbol{w}_t)\big)\nabla_{\boldsymbol{w}_t}\big(\hat{v}(S_t; \boldsymbol{w}_t) - \hat{v}(S_{t+1}; \boldsymbol{w})\big),\end{aligned}$$

then for a linear method $\hat{v}\big(\boldsymbol{x}(s); \boldsymbol{w}\big) \overset{\text{def}}{=} \boldsymbol{x}(s)^T\boldsymbol{w}$ we would get

$$\boldsymbol{w}_{t+1} \leftarrow \boldsymbol{w}_t + \alpha\big(R_{t+1} + \hat{v}(S_{t+1}; \boldsymbol{w}) - \hat{v}(S_t; \boldsymbol{w}_t)\big)\big(\boldsymbol{x}(S_t) - \boldsymbol{x}(S_{t+1})\big).$$

To consider a concrete case, assume the $x(S_t)$ are one-hot encoded, so the update is in fact equal to a tabular method. Then we would update not only the value estimate for state S_t, but also the value estimate for S_{t+1} in the opposite direction.

Temporal Difference Semi-Gradient Convergence

It can be proven (by using separate theory than for SGD) that the linear semi-gradient TD methods do converge.

However, they do not converge to the optimum of $\overline{VE}$. Instead, they converge to a different **TD fixed point $\boldsymbol{w}_\mathrm{TD}$**.

It can be proven that

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

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

Temporal Difference Semi-Gradient Policy Evaluation

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

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

```
Input: the policy $\pi$ 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 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 $T \leftarrow t+1$
    |   $\tau \leftarrow t - n + 1$    ($\tau$ 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".

Temporal Difference Semi-Gradient Policy Evaluation

Recall the previous described 1000-state random walk, where transitions lead 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 result is on the right:

The results using one-step TD(0) are presented below (left); the effect of increasing n in an n-step variant is on the right.

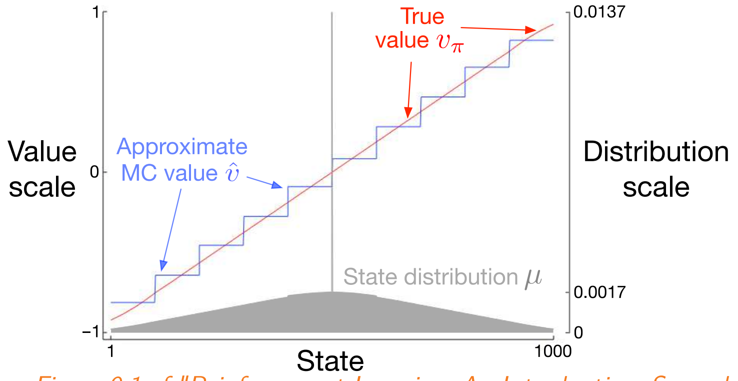
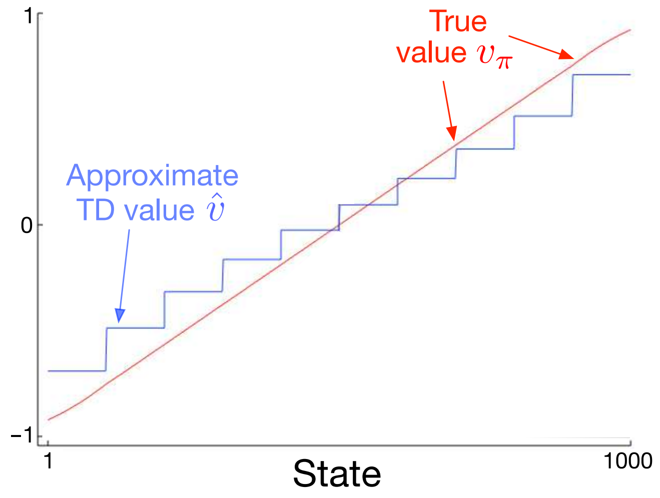


Figure 9.1 of "Reinforcement Learning: An Introduction, Second Edition".



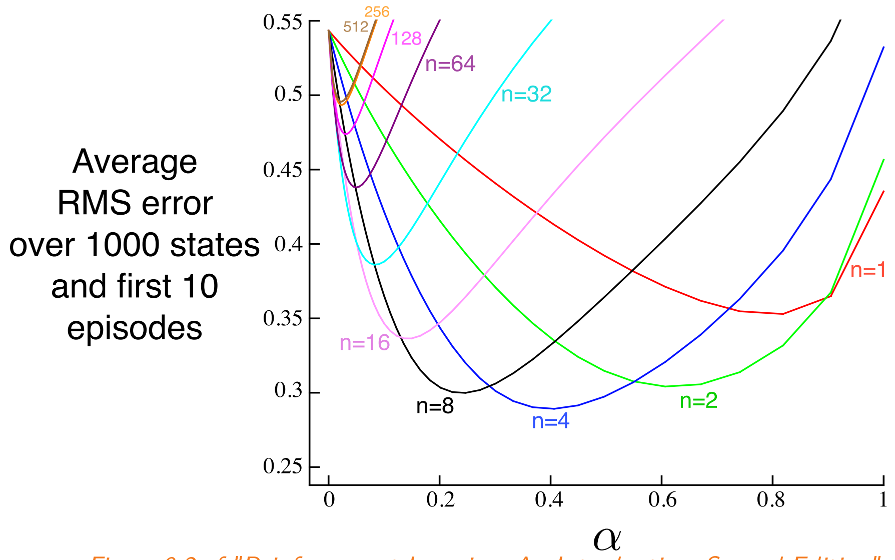


Figure 9.2 of "Reinforcement Learning: An Introduction, Second Edition".

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

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

Sarsa with Function Approximation

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 $\pi$ (if estimating $q_\pi$)
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$, a positive integer $n$
Initialize action-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 $\varepsilon$-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 $\varepsilon$-greedy wrt $\hat{q}(S_{t+1}, \cdot, \mathbf{w})$
    |   $\tau \leftarrow t - n + 1$    ($\tau$ 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 $\tau = T - 1$
```

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

Mountain Car Example via Sarsa

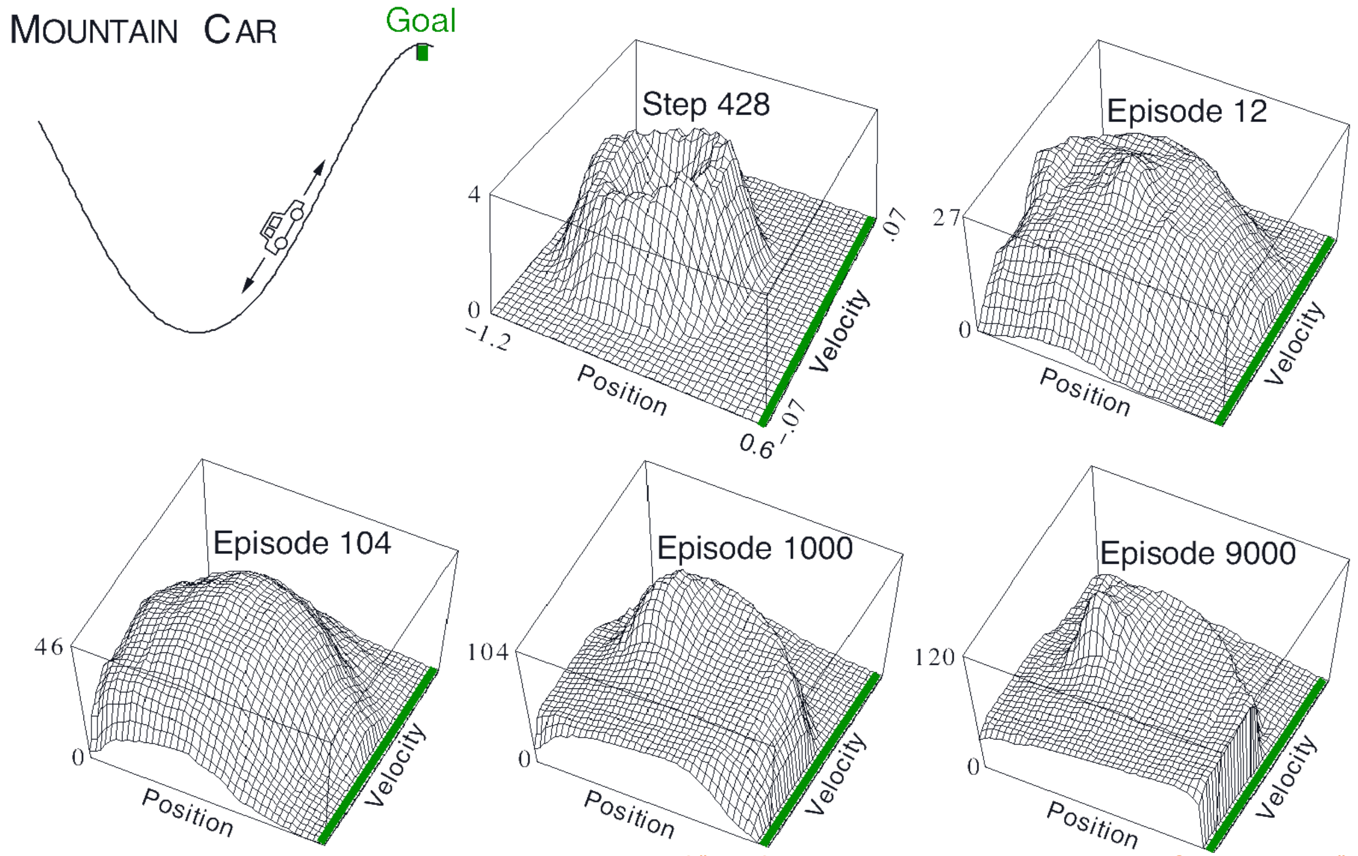


Figure 10.1 of "Reinforcement Learning: An Introduction, Second Edition".

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 via Sarsa

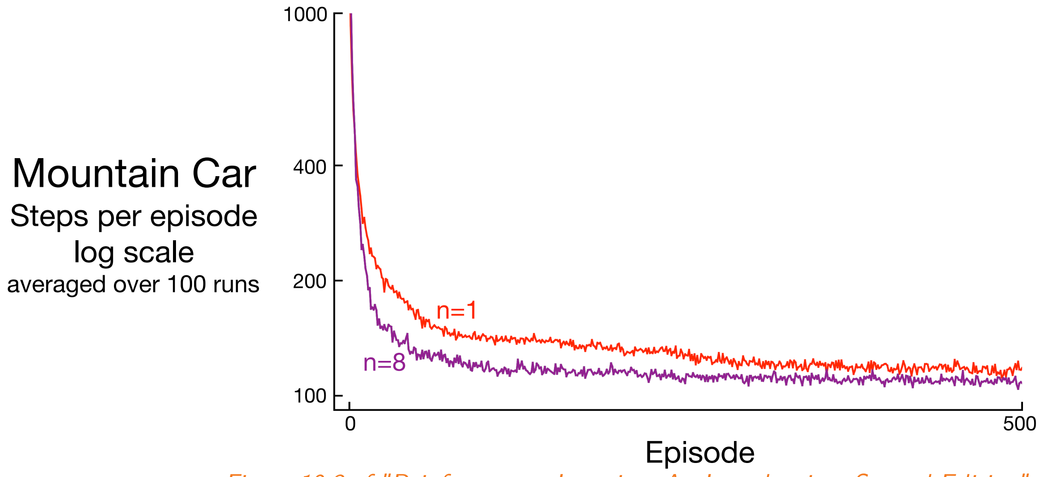


Figure 10.3 of "Reinforcement Learning: An Introduction, Second Edition".

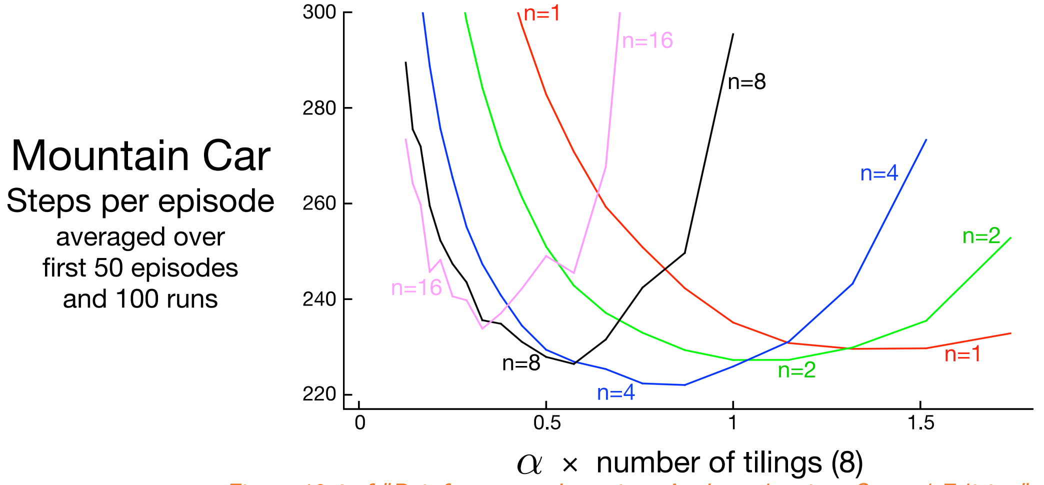


Figure 10.4 of "Reinforcement Learning: An Introduction, Second Edition".

Off-policy Divergence With Function Approximation

Off-policy Divergence With Function Approximation

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

$$\begin{tikzcd} w \arrow[r] & 2w \end{tikzcd}$$

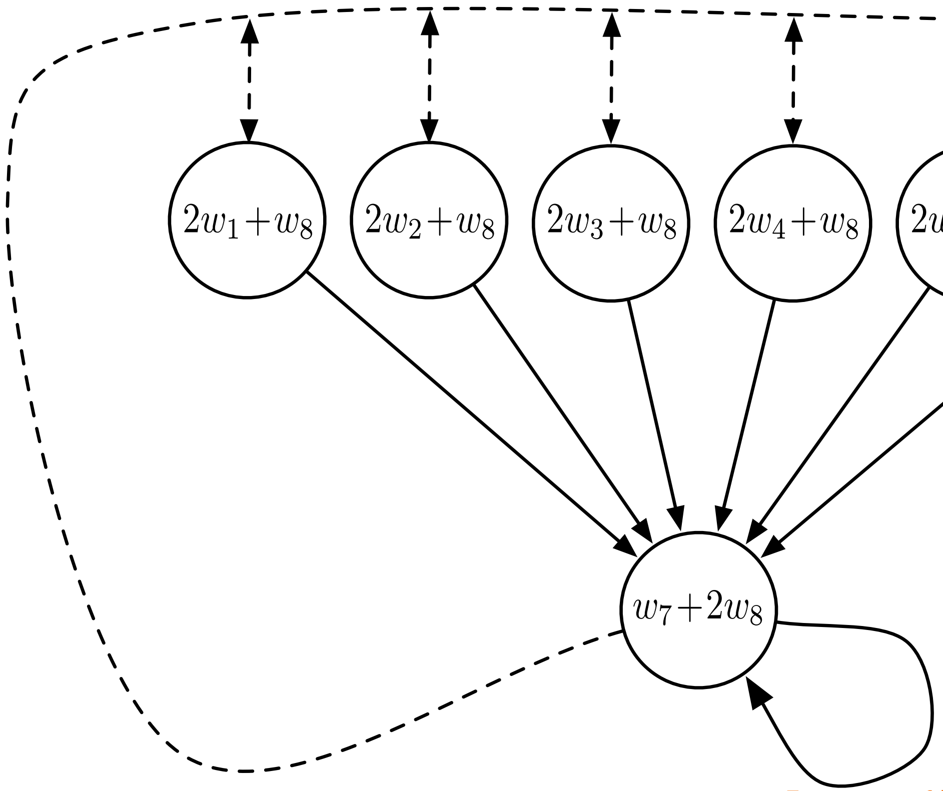
Figure from Section 11.2 of "Reinforcement Learning: An Introduction, Second Edition".

- If initially $w = 10$, the TD error will be also 10 (or nearly 10 if $\gamma < 1$).
- If for example $\alpha = 0.1$, w will be increased to 11 (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.

Off-policy Divergence With Function Approximation

The previous idea can be implemented for instance by the following **Baird's counterexample**:



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

$b(\text{dashed}|\cdot) = 6/7$
$b(\text{solid}|\cdot) = 1/7$

$\gamma = 0.99$

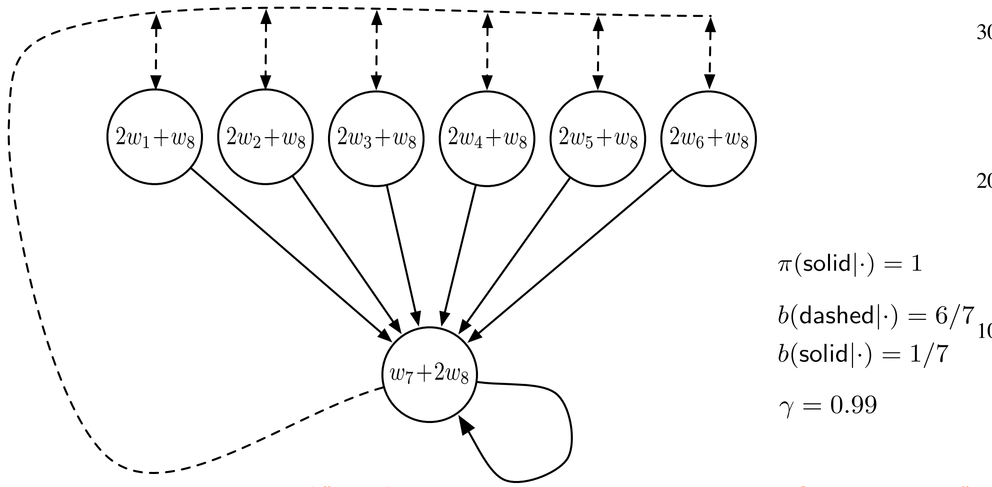
Figure 11.1 of "Reinforcement Learning: An Introduction, Second Edition".

The rewards are zero everywhere, so the value function is also zero everywhere. We assume the initial values of weights are 1, except for $w_7 = 10$, and that the learning rate $\alpha = 0.01$.

Off-policy Divergence With Function Approximation

For off-policy semi-gradient Sarsa, or even for off-policy dynamic-programming update (where we compute expectation over all following states and actions), the weights diverge to $+\infty$. Using on-policy distribution converges fine.

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \frac{\alpha}{|\mathcal{S}|} \sum_{s} \Big(\mathbb{E}_\pi \big[R_{t+1} + \gamma \hat{v}(S_{t+1}; \boldsymbol{w}) \big| S_t = s \big] - \hat{v}(s; \boldsymbol{w}) \Big) \nabla \hat{v}(s; \boldsymbol{w})$$



$2w_1+w_8$
$2w_2+w_8$
$2w_3+w_8$
$2w_4+w_8$
$2w_5+w_8$
$2w_6+w_8$
w_7+2w_8
$\pi(\text{solid}|\cdot) = 1$
$b(\text{dashed}|\cdot) = 6/7$
$b(\text{solid}|\cdot) = 1/7$
$\gamma = 0.99$

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

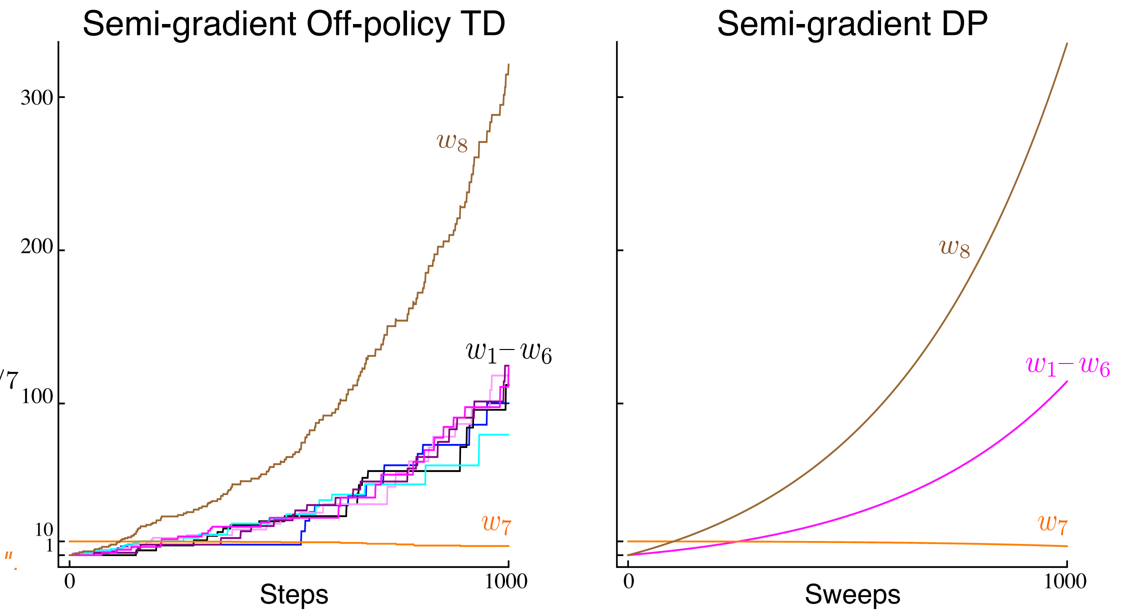


Figure 11.2 of "Reinforcement Learning: An Introduction, Second Edition".

Off-policy Divergence With Function Approximation

The divergence can happen when all following elements are combined:

- functional approximation;
- bootstrapping;
- off-policy training.

In the Sutton's and Barto's book, these are called **the deadly triad**.

Deep Q Networks

Deep Q Networks

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

Deep Q Network

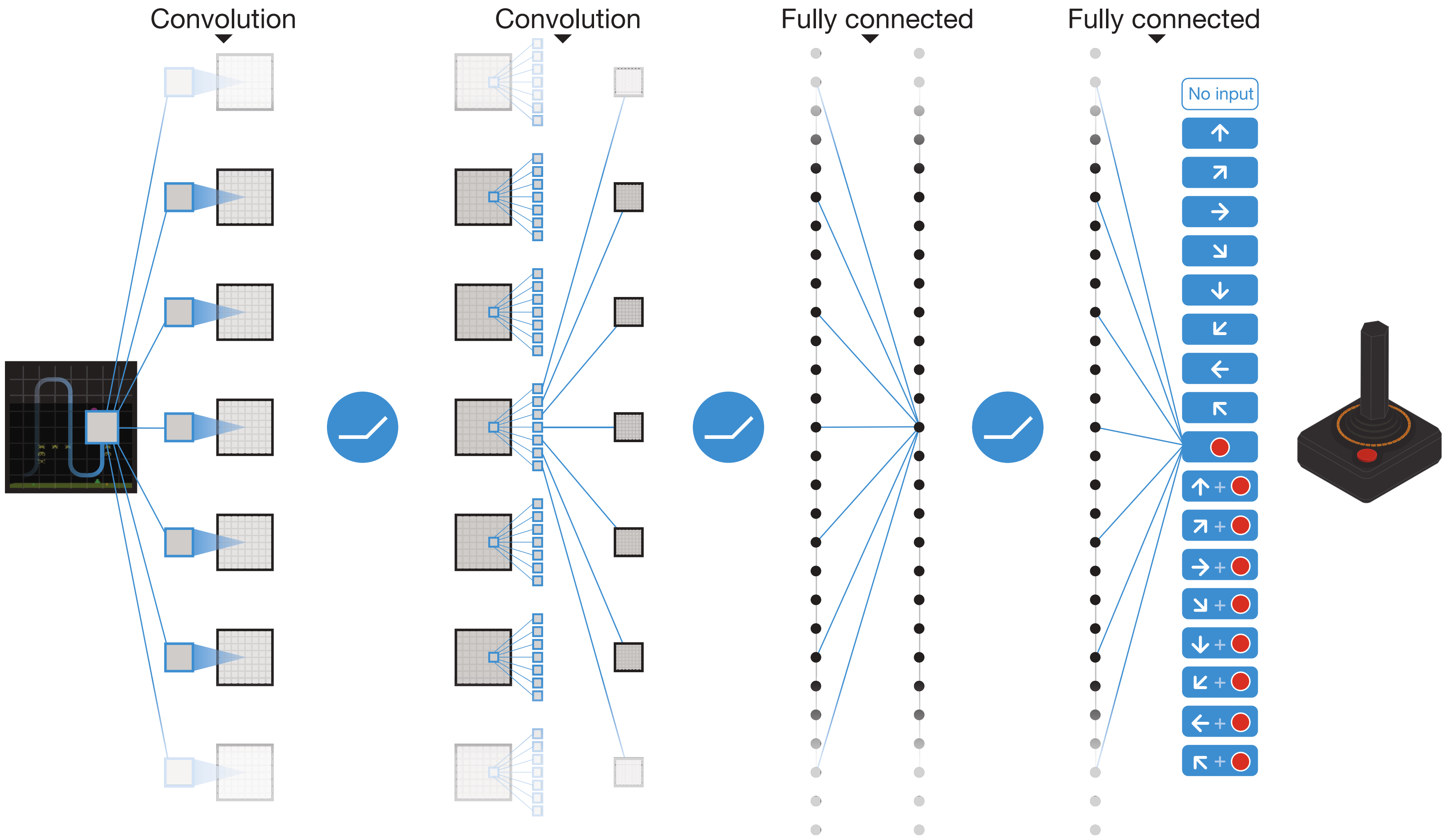


Figure 1 of "Human-level control through deep reinforcement learning" by Volodymyr Mnih et al.

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.
- **Frame stacking** is utilizied – the input to the network are the last 4 frames (considering only the frames kept by frame skipping), i.e., the network inpus is 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:

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

- An ε-greedy behavior policy is utilized (starts at $\varepsilon = 1$ and gradually decreases to 0.1).

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 (off-policy training);
- separate **target network** $\bar{\boldsymbol\theta}$: to prevent instabilities, a separate *target network* is used to estimate one-step returns. The weights are not trained, but copied from the trained network after a fixed number of gradient updates;
- reward clipping: because rewards have wildly different scale in different games, all positive rewards are replaced by $+1$ and negative by -1; life loss is used as an end of episode.
  - furthermore, $(r + [\neg\text{done}]\cdot\gamma\max_{a'} Q(s',a';\bar{\boldsymbol\theta}) - Q(s,a;\boldsymbol\theta))$ is also clipped to $[-1,1]$ (i.e., a smooth_{L_1} loss or Huber loss).

Deep Q Networks

```
**Algorithm 1: deep Q-learning with experience replay.**
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, $M$ do
   Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
   For $t = 1$,T do
      With probability $\varepsilon$ select a random action $a_t$
      otherwise select $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$
      Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
      Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
      Store transition $\left(\phi_t, a_t, r_t, \phi_{t+1}\right)$ in $D$
      Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from $D$
      Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^-\right) & \text{otherwise} \end{cases}$
      Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the
      network parameters $\theta$
      Every $C$ steps reset $\hat{Q} = Q$
   End For
End For
```

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

Deep Q Network

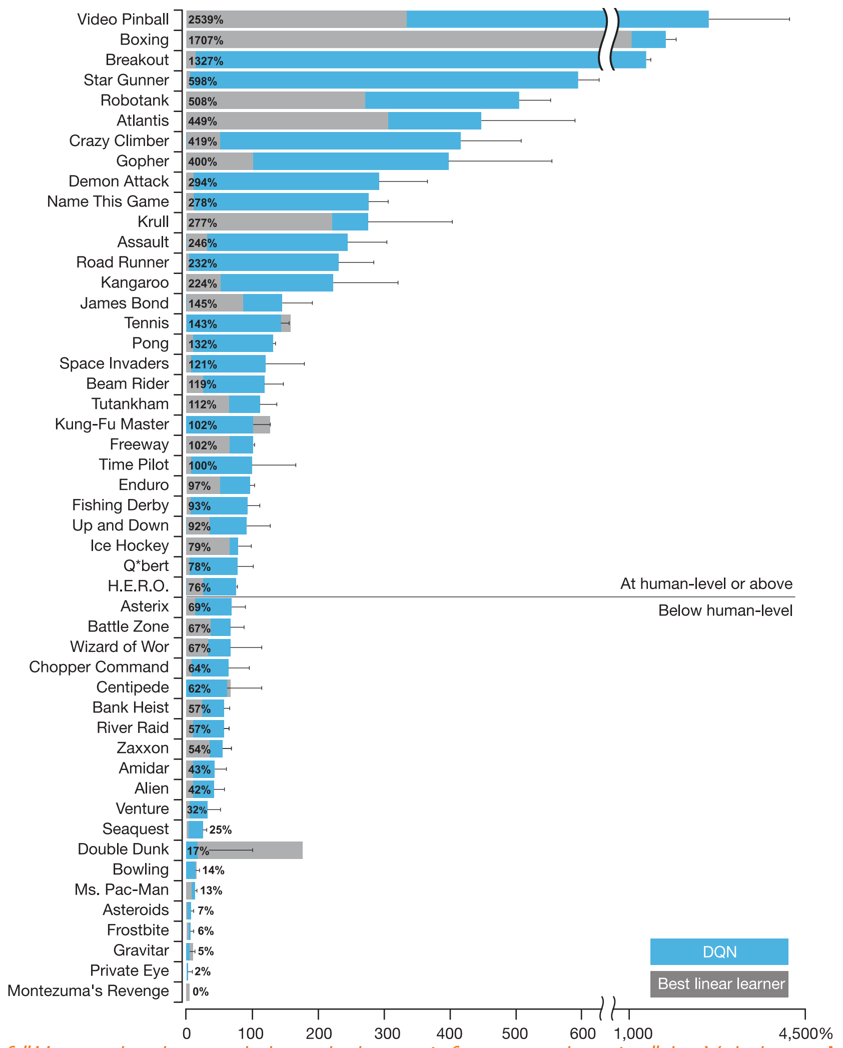
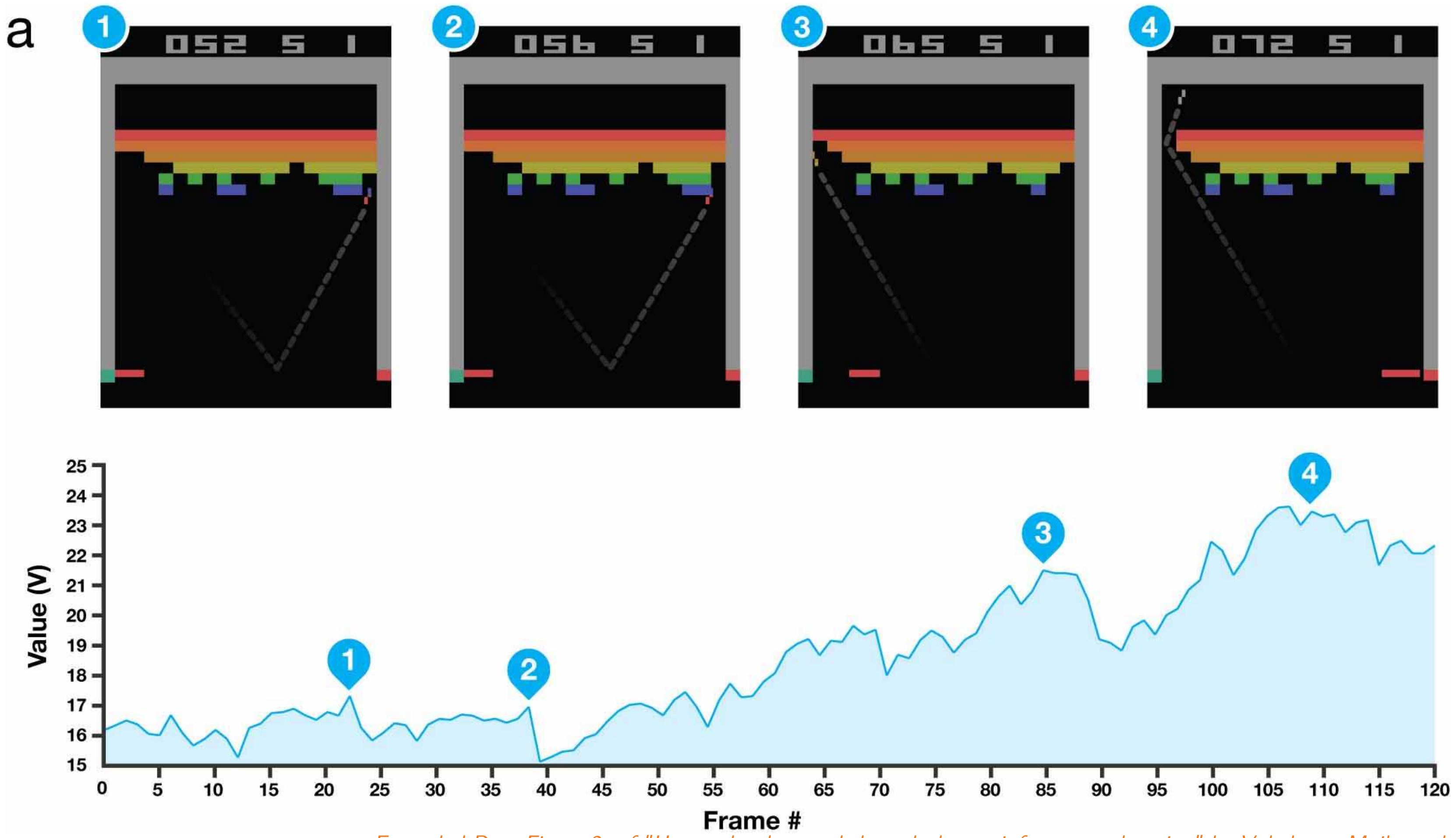


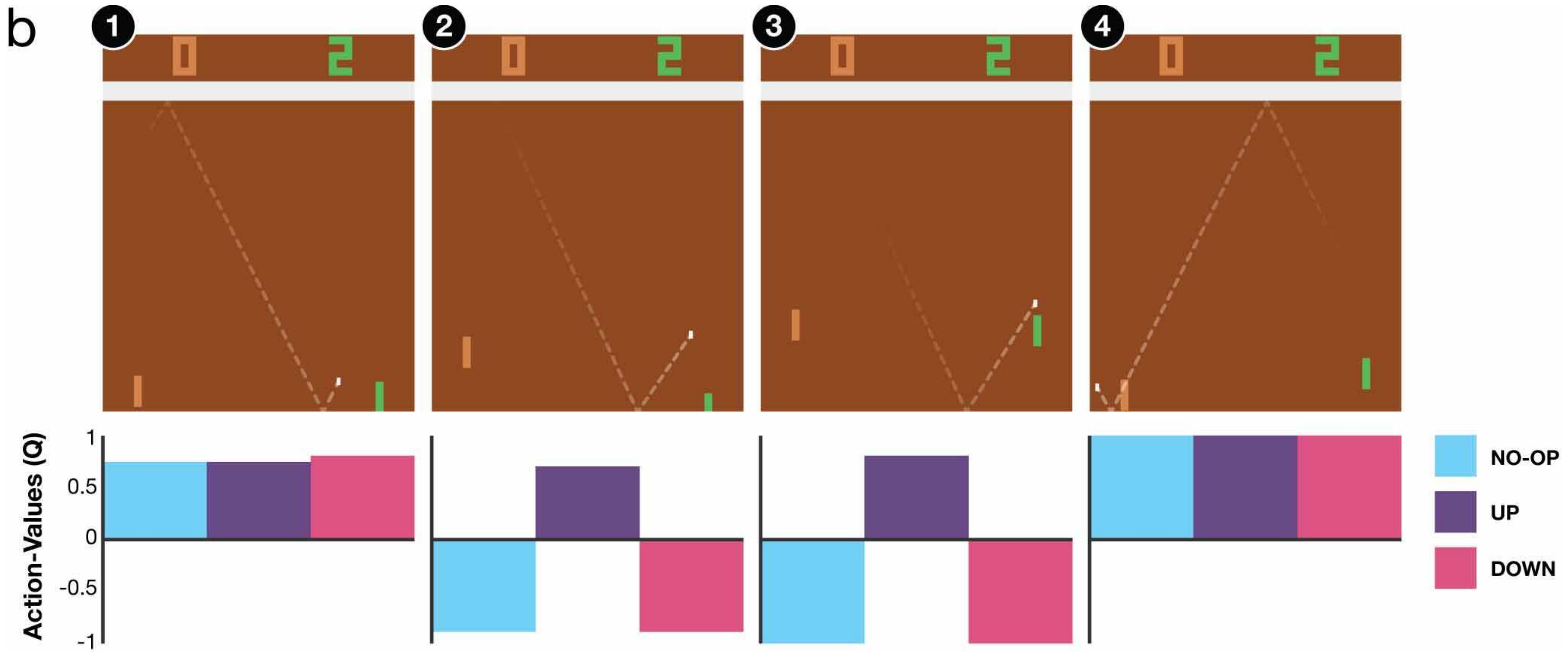
Figure 3 of "Human-level control through deep reinforcement learning" by Volodymyr Mnih et al.

Deep Q Network



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

Deep Q Network



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

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 both momentums	0.00025, 0.95
initial ε, final ε (linear decay) and frame of final ε	1.0, 0.1, 1M
replay start size	50k
no-op max	30

Rainbow

Rainbow

There have been many suggested improvements to the DQN architecture. In the end of 2017, the *Rainbow: Combining Improvements in Deep Reinforcement Learning* paper combines 6 of them into a single architecture they call **Rainbow**.

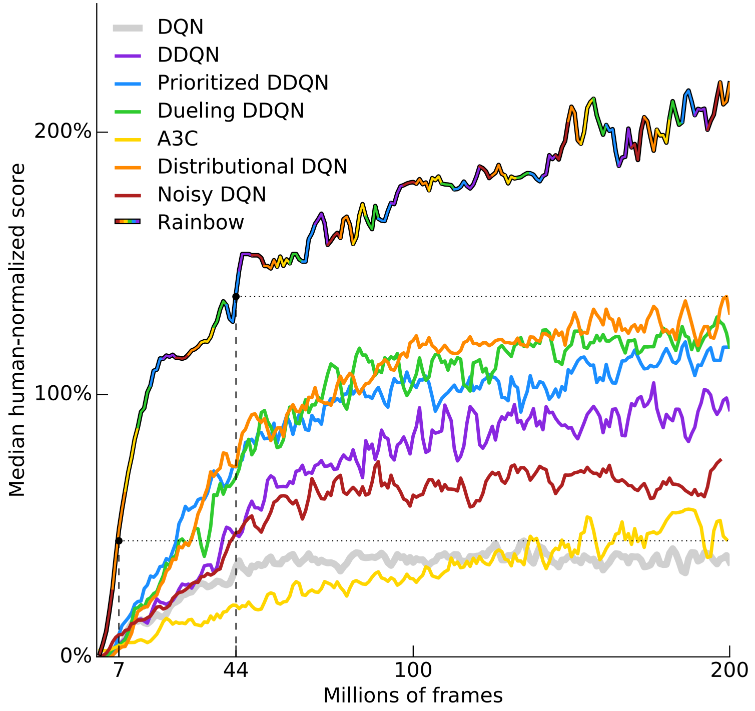


Figure 1 of "Rainbow: Combining Improvements in Deep Reinforcement Learning" by Matteo Hessel et al.

Double Deep Q-Network

Q-learning and Maximization Bias

Because behaviour policy in Q-learning is ε-greedy variant of the target policy, the same samples (up to ε-greedy) determine both the maximizing action and estimate its value.

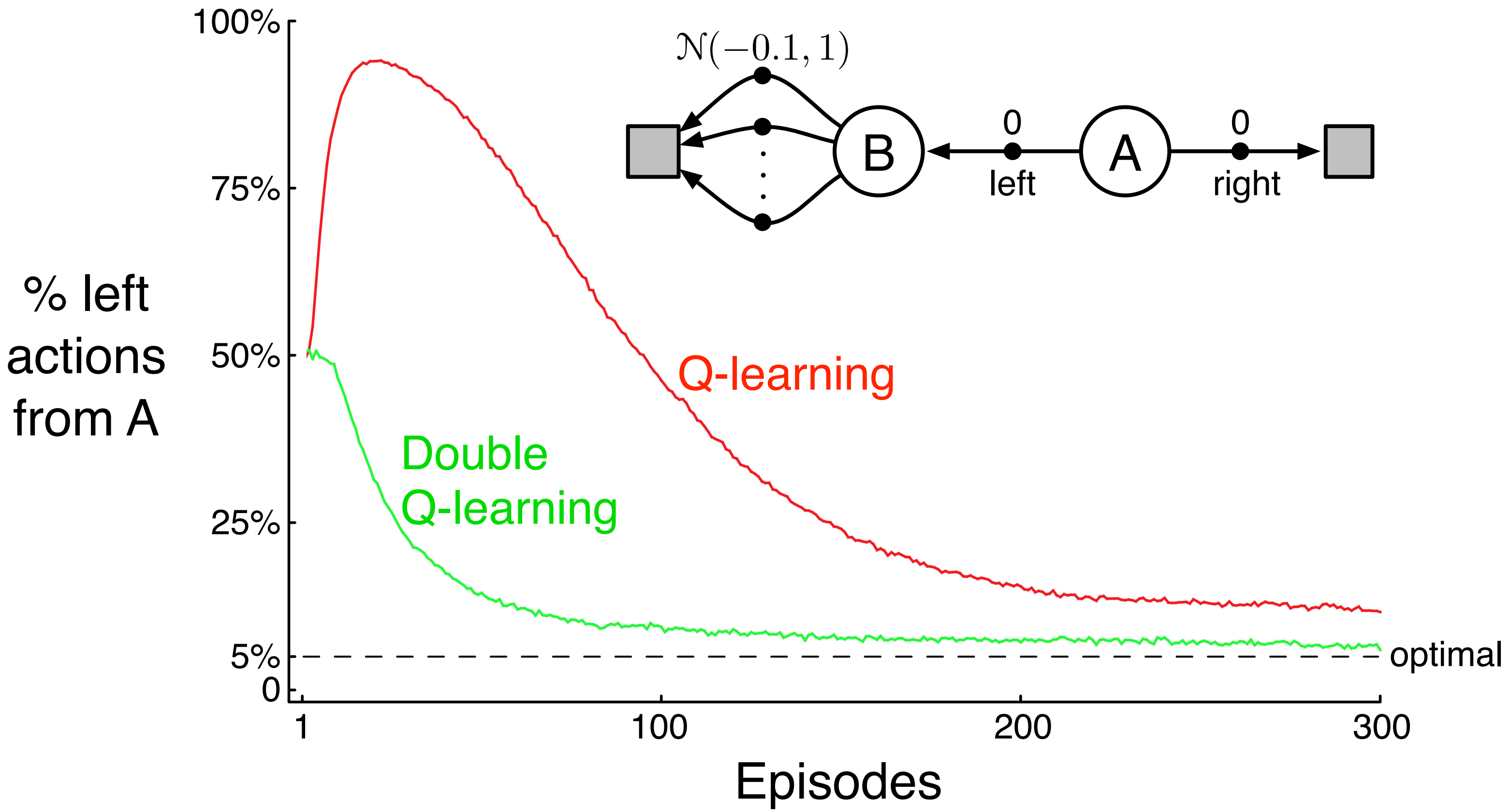


Figure 6.5 of "Reinforcement Learning: An Introduction, Second Edition".

Double Q-learning

Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q_*$

```
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, such that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
  Initialize $S$
  Loop for each step of episode:
    Choose $A$ from $S$ using the policy $\varepsilon$-greedy in $Q_1 + Q_2$
    Take action $A$, observe $R$, $S'$
    With 0.5 probabilility:
      $Q_1(S, A) \leftarrow Q_1(S, A) + \alpha\Big(R + \gamma Q_2\big(S', \arg\max_a Q_1(S', a)\big) - Q_1(S, A)\Big)$
    else:
      $Q_2(S, A) \leftarrow Q_2(S, A) + \alpha\Big(R + \gamma Q_1\big(S', \arg\max_a Q_2(S', a)\big) - Q_2(S, A)\Big)$
    $S \leftarrow S'$
  until $S$ is terminal
```

Modification of Algorithm 6.7 of "Reinforcement Learning: An Introduction, Second Edition" (replacing S+ by S).

Double Deep Q-Network

Similarly to double Q-learning, instead of

$$r + \gamma \max_{a'} Q(s', a'; \bar{\boldsymbol\theta}) - Q(s, a; \boldsymbol\theta),$$

we minimize

$$r + \gamma Q(s', \arg\max_{a'} Q(s', a'; \boldsymbol\theta); \bar{\boldsymbol\theta}) - Q(s, a; \boldsymbol\theta).$$

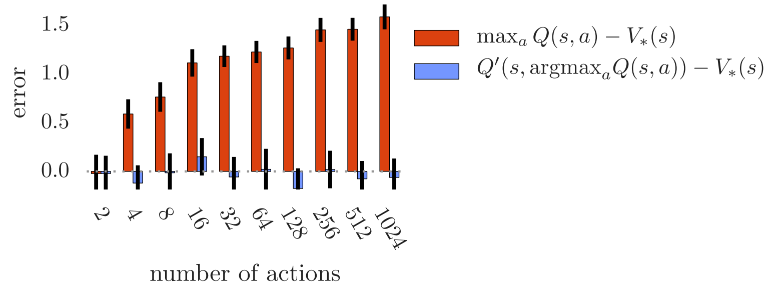


Figure 1: The orange bars show the bias in a single Q-learning update when the action values are $Q(s,a) = V_*(s) + \epsilon_a$ and the errors $\{\epsilon_a\}_{a=1}^m$ are independent standard normal random variables. The second set of action values Q', used for the blue bars, was generated identically and independently. All bars are the average of 100 repetitions.

Figure 1 of "Deep Reinforcement Learning with Double Q-learning" by Hado van Hasselt et al.

Double Deep Q-Network

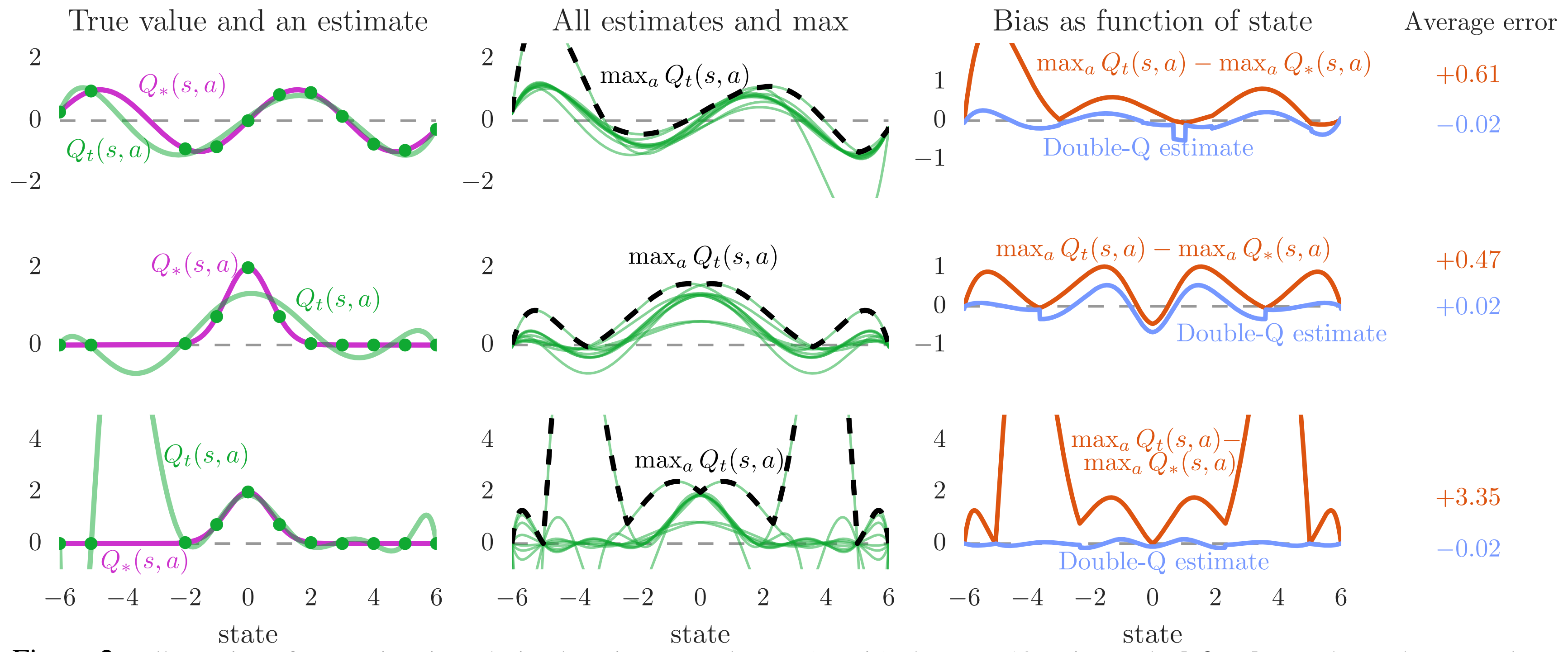


Figure 2: Illustration of overestimations during learning. In each state (x-axis), there are 10 actions. The **left column** shows the true values $V_*(s)$ (purple line). All true action values are defined by $Q_*(s,a) = V_*(s)$. The green line shows estimated values $Q(s,a)$ for one action as a function of state, fitted to the true value at several sampled states (green dots). The **middle column** plots show all the estimated values (green), and the maximum of these values (dashed black). The maximum is higher than the true value (purple, left plot) almost everywhere. The **right column** plots shows the difference in orange. The blue line in the right plots is the estimate used by Double Q-learning with a second set of samples for each state. The blue line is much closer to zero, indicating less bias. The three **rows** correspond to different true functions (left, purple) or capacities of the fitted function (left, green). (Details in the text)

Figure 2 of "Deep Reinforcement Learning with Double Q-learning" by Hado van Hasselt et al.

Double Q-learning

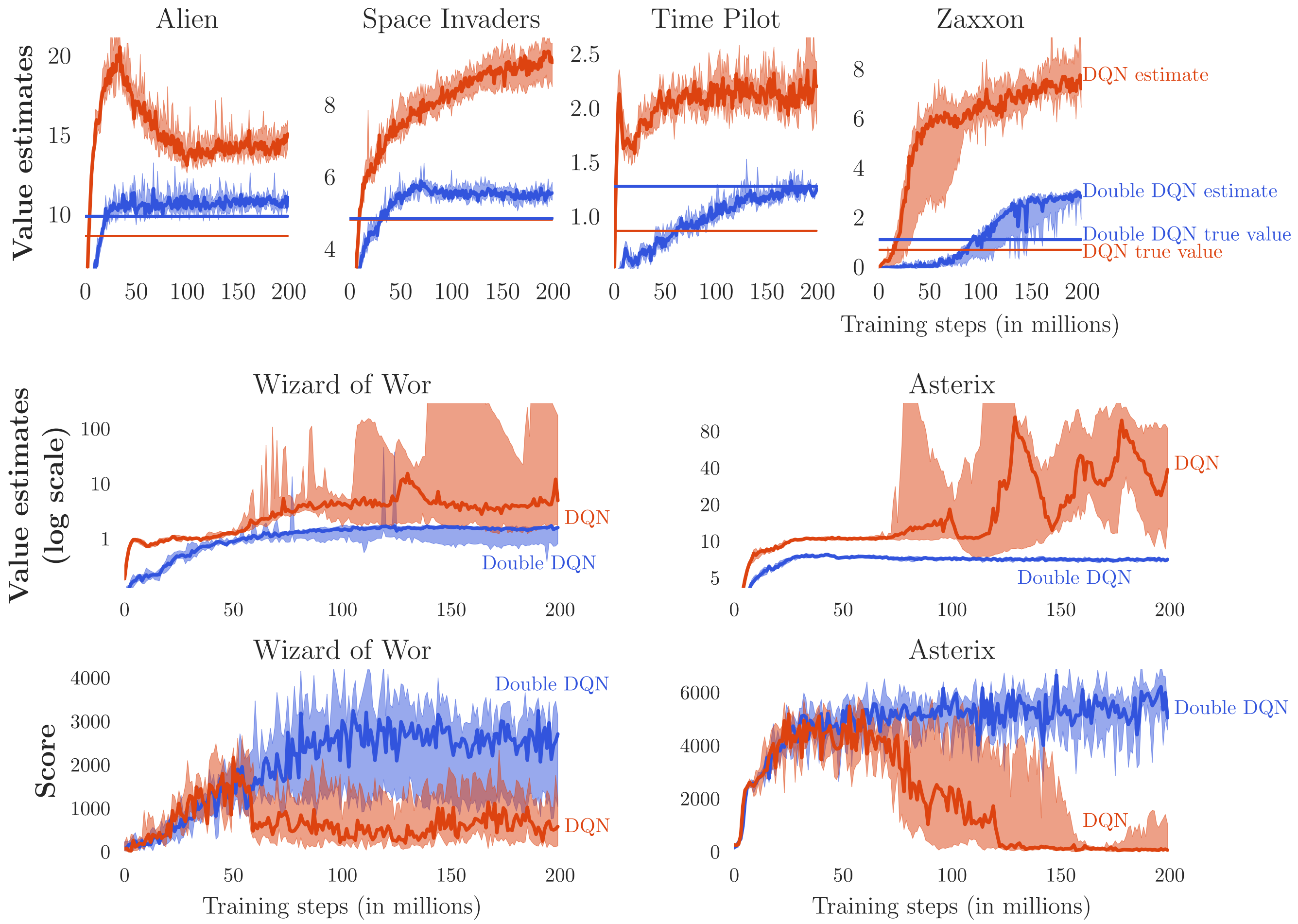


Figure 3 of "Deep Reinforcement Learning with Double Q-learning" by Hado van Hasselt et al.

Double Q-learning

Performance on episodes taking at most 5 minutes and no-op starts on 49 games:

	DQN	Double DQN
Median	93.5%	114.7%
Mean	241.1%	330.3%

Table 1 of "Deep Reinforcement Learning with Double Q-learning" by Hado van Hasselt et al.

Performance on episodes taking at most 30 minutes and using 100 human starts on each of the 49 games:

	DQN	Double DQN	Double DQN (tuned)
Median	47.5%	88.4%	116.7%
Mean	122.0%	273.1%	475.2%

Table 2 of "Deep Reinforcement Learning with Double Q-learning" by Hado van Hasselt et al.

The Double DQN follows the training protocol of DQN; the tuned version increases the target network update from 10k to 30k steps, decreases exploration during training from $\varepsilon = 0.1$ to $\varepsilon = 0.01$, and uses a shared bias for all action values in the output layer of the network.

Prioritized Replay

Prioritized Replay

Instead of sampling the transitions uniformly from the replay buffer, we instead prefer those with a large TD error. Therefore, we sample transitions according to their probability

$$p_t \propto \Big|r + \gamma \max_{a'} Q(s', a'; \bar{\boldsymbol\theta}) - Q(s, a; \boldsymbol\theta)\Big|^\omega,$$

where ω controls the shape of the distribution (which is uniform for $\omega = 0$ and corresponds to TD error for $\omega = 1$).

New transitions are inserted into the replay buffer with maximum probability to support exploration of all encountered transitions.

When combined with DDQN, the probabilities are naturally computed as

$$p_t \propto \Big|r + \gamma Q(s', \arg\max_{a'} Q(s', a'; \boldsymbol\theta); \bar{\boldsymbol\theta}) - Q(s, a; \boldsymbol\theta)\Big|^\omega,$$

Prioritized Replay

Because we now sample transitions according to p_t instead of uniformly, on-policy distribution and sampling distribution differ. To compensate, we therefore utilize importance sampling with ratio

$$\rho_t = \left(\frac{1/N}{p_t}\right)^\beta.$$

The authors utilize in fact "for stability reasons"

$$\rho_t / \max_i \rho_i.$$

Prioritized Replay

Algorithm 1 Double DQN with proportional prioritization

```
1: **Input:** minibatch $k$, step-size $\eta$, replay period $K$ and size $N$, exponents $\alpha$ and $\beta$, budget $T$.
2: Initialize replay memory $\mathcal{H} = \emptyset$, $\Delta = 0$, $p_1 = 1$
3: Observe $S_0$ and choose $A_0 \sim \pi_\theta(S_0)$
4: **for** $t = 1$ **to** $T$ **do**
5:     Observe $S_t, R_t, \gamma_t$
6:     Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in $\mathcal{H}$ with maximal priority $p_t = \max_{i<t} p_i$
7:     **if** $t \equiv 0 \mod K$ **then**
8:         **for** $j = 1$ **to** $k$ **do**
9:             Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
10:            Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$
11:            Compute TD-error $\delta_j = R_j + \gamma_j Q_{\text{target}}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$
12:            Update transition priority $p_j \leftarrow |\delta_j|$
13:            Accumulate weight-change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$
14:        **end for**
15:        Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$
16:        From time to time copy weights into target network $\theta_{\text{target}} \leftarrow \theta$
17:    **end if**
18:    Choose action $A_t \sim \pi_\theta(S_t)$
19: **end for**
```

Algorithm 1 of "Prioritized Experience Replay" by Tom Schaul et al.

Dueling Networks

Dueling Networks

Instead of computing directly $Q(s, a; \boldsymbol\theta)$, we compose it from the following quantities:

- average return in a given state s, $V(s; \boldsymbol\theta) = \frac{1}{|\mathcal{A}|} \sum_a Q(s, a; \boldsymbol\theta)$,
- advantage function computing an **advantage** $Q(s, a; \boldsymbol\theta) - V(s; \theta)$ of action a in state s.

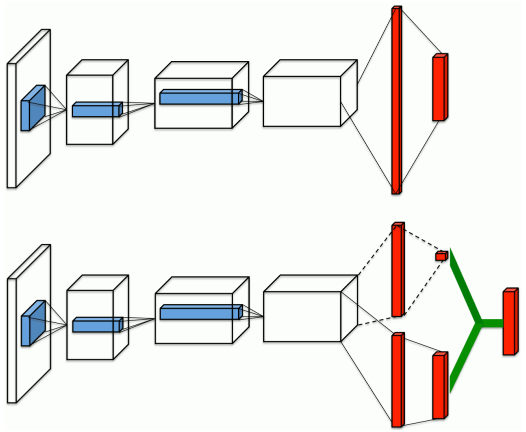
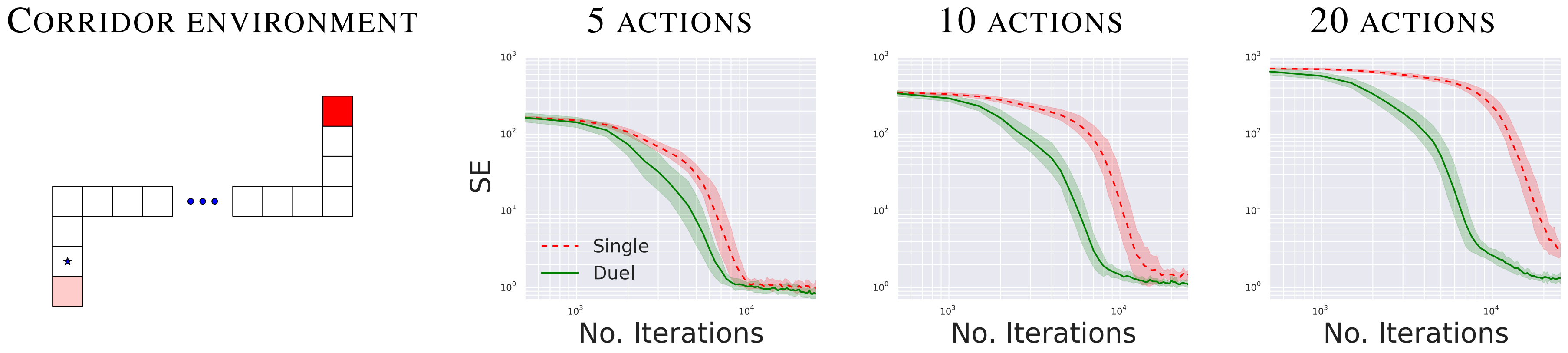


Figure 1 of "Dueling Network Architectures for Deep Reinforcement Learning" by Ziyu Wang et al.

$$Q(s, a) \stackrel{\text{def}}{=} V\big(f(s; \zeta); \eta\big) + A\big(f(s; \zeta), a; \psi\big) - \frac{\sum_{a' \in \mathcal{A}} A(f(s; \zeta), a'; \psi)}{|\mathcal{A}|}$$

Dueling Networks



(a) (b) (c) (d)

Figure 3. **(a)** The corridor environment. The star marks the starting state. The redness of a state signifies the reward the agent receives upon arrival. The game terminates upon reaching either reward state. The agent's actions are going up, down, left, right and no action. Plots **(b), (c)** and **(d)** shows squared error for policy evaluation with 5, 10, and 20 actions on a log-log scale. The dueling network (Duel) consistently outperforms a conventional single-stream network (Single), with the performance gap increasing with the number of actions.

Figure 3 of "Dueling Network Architectures for Deep Reinforcement Learning" by Ziyu Wang et al.

Evaluation is performed using ε-greedy exploration with $\varepsilon = 0.001$; in the experiment, the horizontal corridor has a length of 50 steps, while the vertical sections have both 10 steps.

Dueling Networks

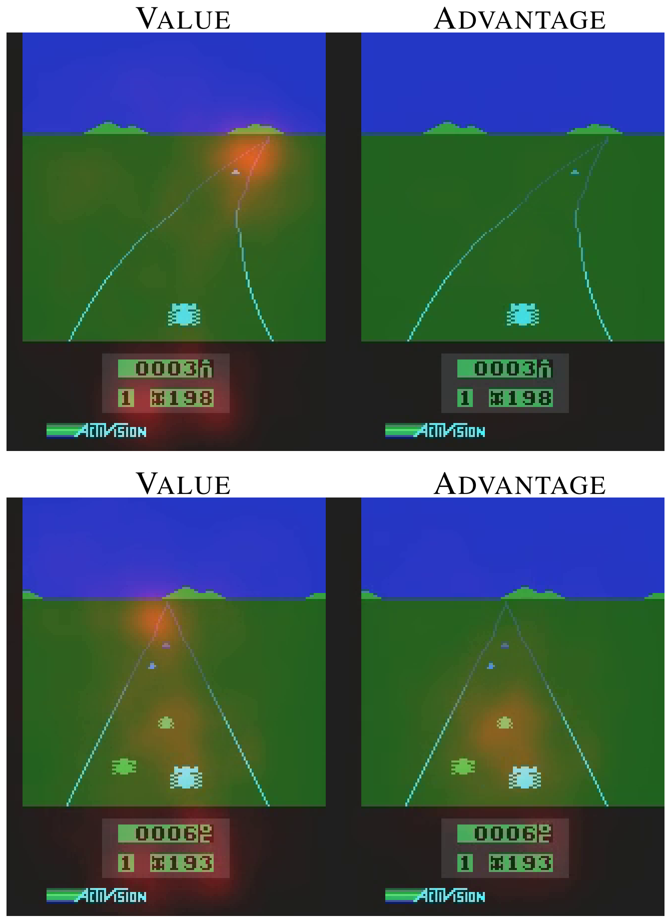


Figure 2 of "Dueling Network Architectures for Deep Reinforcement Learning" by Ziyu Wang et al.

Dueling Networks

Results on all 57 games (retraining the original DQN on the 8 missing games). `Single` refers to DDQN with a direct computation of $Q(s, a; \boldsymbol\theta)$, `Single Clip` corresponds to additional gradient clipping to norm at most 10 and larger first hidden layer (so that duelling and single have roughly the same number of parameters).

	30 no-ops		**Human Starts**	
	Mean	**Median**	**Mean**	**Median**
Prior. Duel Clip	**591.9%**	**172.1%**	**567.0%**	**115.3%**
Prior. Single	434.6%	123.7%	386.7%	112.9%
Duel Clip	**373.1%**	**151.5%**	**343.8%**	**117.1%**
Single Clip	341.2%	132.6%	302.8%	114.1%
Single	307.3%	117.8%	332.9%	110.9%
Nature DQN	227.9%	79.1%	219.6%	68.5%

Table 1 of "Dueling Network Architectures for Deep Reinforcement Learning" by Ziyu Wang et al.

Multi-step DQN

Multi-step DQN

Instead of Q-learning, we use n-step variant of Q-learning, which estimates return as

$$\sum_{i=1}^{n} \gamma^{i-1} R_i + \gamma^n \max_{a'} Q(s', a'; \bar{\boldsymbol\theta}).$$

This changes the off-policy algorithm to on-policy (because the “inner” actions are sampled from the behaviour distribution, but should follow the target distribution); however, it is not discussed in any way by the authors.

Noisy Nets

Noisy Nets

Noisy Nets are neural networks whose weights and biases are perturbed by a parametric function of a noise.

The parameters $\boldsymbol\theta$ of a regular neural network are in Noisy nets represented as

$$\boldsymbol\theta \approx \boldsymbol\mu + \boldsymbol\sigma \odot \boldsymbol\varepsilon,$$

where $\boldsymbol\varepsilon$ is zero-mean noise with fixed statistics. We therefore learn the parameters $(\boldsymbol\mu, \boldsymbol\sigma)$.

A fully connected layer $\boldsymbol y = \boldsymbol w \boldsymbol x + \boldsymbol b$ with parameters $(\boldsymbol w, \boldsymbol b)$ is represented in the following way in Noisy nets:

$$\boldsymbol y = (\boldsymbol\mu_w + \boldsymbol\sigma_w \odot \boldsymbol\varepsilon_w)\boldsymbol x + (\boldsymbol\mu_b + \boldsymbol\sigma_b \odot \boldsymbol\varepsilon_b).$$

Each $\sigma_{i,j}$ is initialized to $\frac{\sigma_0}{\sqrt{n}}$, where n is the number of input neurons of the layer in question, and σ_0 is a hyperparameter; commonly 0.5.

Rainbow DQN Extensions

Noisy Nets

The noise ε can be for example independent Gaussian noise. However, for performance reasons, factorized Gaussian noise is used to generate a matrix of noise. If $\varepsilon_{i,j}$ is noise corresponding to a layer with n inputs and m outputs, we generate independent noise ε_i for input neurons, independent noise ε_j for output neurons, and set

$$\varepsilon_{i,j} = f(\varepsilon_i)f(\varepsilon_j) \quad \text{for} \quad f(x) = \operatorname{sign}(x)\sqrt{|x|}.$$

The authors generate noise samples for every batch, sharing the noise for all batch instances (consequently, during loss computation, online and target network use independent noise).

Deep Q Networks

When training a DQN, ε-greedy is no longer used (all policies are greedy), and all fully connected layers are parametrized as noisy nets in both the current and target network (i.e., networks produce samples from the distribution of returns, and greedy actions still explore).

Noisy Nets

	Baseline		NoisyNet		Improvement
	Mean	Median	Mean	Median	(On median)
DQN	319	83	**379**	**123**	48%
Dueling	524	132	**633**	**172**	30%
A3C	293	80	**347**	**94**	18%

Table 1 of "Noisy Networks for Exploration" by Meire Fortunato et al.

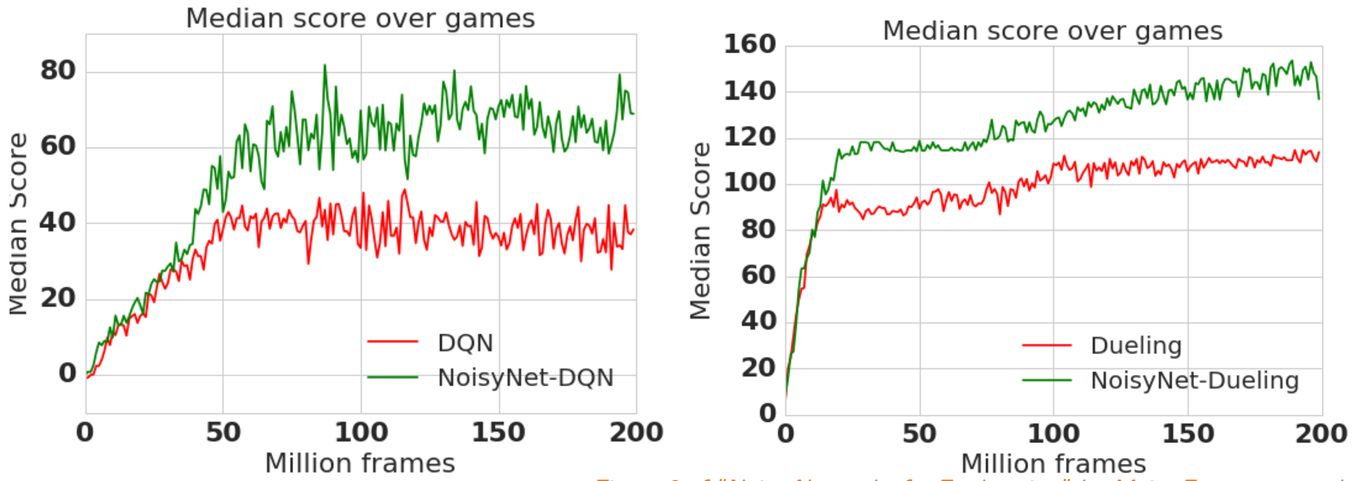


Figure 2 of "Noisy Networks for Exploration" by Meire Fortunato et al.

Noisy Nets

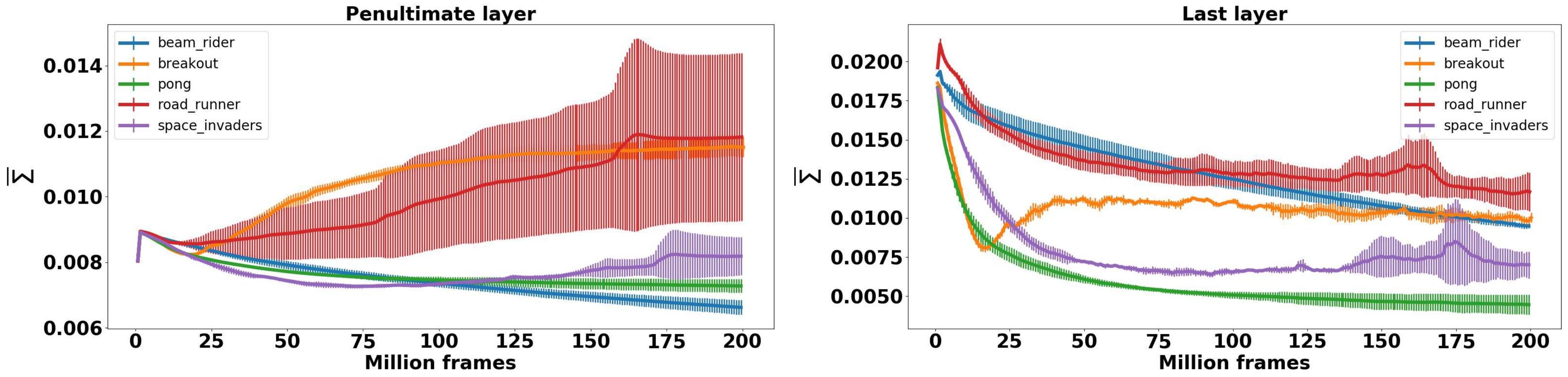


Figure 3: Comparison of the learning curves of the average noise parameter $\bar{\Sigma}$ across five Atari games in NoisyNet-DQN. The results are averaged across 3 seeds and error bars (+/- standard deviation) are plotted.

Figure 3 of "Noisy Networks for Exploration" by Meire Fortunato et al.

The $\bar{\Sigma}$ is the mean-absolute of the noise weights $\boldsymbol{\sigma}_w$, i.e., $\bar{\Sigma} = \frac{1}{\textit{layer size}} \|\boldsymbol{\sigma}_w\|_1$.

Distributional RL

Rainbow DQN Extensions

Distributional RL

Instead of an expected return $Q(s, a)$, we could estimate the distribution of expected returns $Z(s, a)$ – the *value distribution*.

The authors define the distributional Bellman operator $\mathcal{T}^\pi$ as:

$$\mathcal{T}^\pi Z(s, a) \stackrel{\text{def}}{=} R(s, a) + \gamma Z(S', A') \quad \text{for} \quad S' \sim p(s, a), A' \sim \pi(S').$$

The authors of the paper prove similar properties of the distributional Bellman operator compared to the regular Bellman operator, mainly being a contraction under a suitable metric (for Wasserstein metric W_p, the authors define $\bar{W}_p(Z_1, Z_2) \stackrel{\text{def}}{=} \sup_{s,a} W_p\big(Z_1(s, a), Z_2(s, a)\big)$ and prove that $\mathcal{T}^\pi$ is a γ-contraction in $\bar{W}_p$).

Wasserstein Metric

For two probability distributions μ, ν, Wasserstein metric W_p is defined as

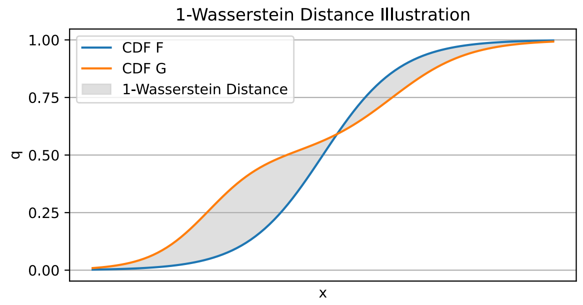
$$W_p(\mu, \nu) \overset{\text{def}}{=} \inf_{\gamma \in \Gamma(\mu,\nu)} \left(\mathbb{E}_{(x,y)\sim\gamma} \|x - y\|^d\right)^{1/p},$$

where $\Gamma(\mu, \nu)$ is a set of all *couplings*, each being a a joint probability distribution whose marginals are μ and ν, respectively. A possible intuition is the optimal transport of probability mass from μ to ν.

For distributions over reals with CDFs F, G, the optimal transport has an analytic solution:

$$W_p(\mu, \nu) = \left(\int_0^1 |F^{-1}(q) - G^{-1}(q)|^p \,\mathrm{d}q\right)^{1/p},$$

where F^{-1} and G^{-1} are *quantile functions*, i.e., inverse CDFs.



For $p = 1$, the 1-Wasserstein metric correspond to area "between" F and G, and in that case we can compute it also as $W_1(\mu, \nu) = \int_x \big|F(x) - G(x)\big| \,\mathrm{d}x$.

Rainbow DQN Extensions

Distributional RL

The distribution of returns is modeled as a discrete distribution parametrized by the number of atoms $N \in \mathbb{N}$ and by $V_\textrm{MIN}, V_\textrm{MAX} \in \mathbb{R}$. Support of the distribution are atoms

$$\{z_i \overset{\textrm{def}}{=} V_\textrm{MIN} + i\Delta z : 0 \leq i < N\} \quad \textrm{for } \Delta z \overset{\textrm{def}}{=} \frac{V_\textrm{MAX} - V_\textrm{MIN}}{N-1}.$$

The atom probabilities are predicted using a $\operatorname{softmax}$ distribution as

$$Z_{\boldsymbol\theta}(s, a) = \left\{ z_i \textrm{ with probability } p_i = \frac{e^{f_i(s,a;\boldsymbol\theta)}}{\sum_j e^{f_j(s,a;\boldsymbol\theta)}} \right\}.$$

Rainbow DQN Extensions

Distributional RL

After the Bellman update, the support of the distribution $R(s,a) + \gamma Z(s',a')$ is not the same as the original support. We therefore project it to the original support by proportionally mapping each atom of the Bellman update to immediate neighbors in the original support.

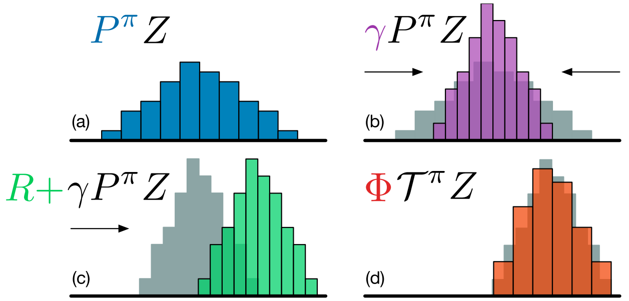


Figure 1 of "A Distributional Perspective on Reinforcement Learning" by Marc G. Bellemare et al.

$$\Phi\big(R(s,a) + \gamma Z(s',a')\big)_i \overset{\text{def}}{=} \sum_{j=1}^N \left[1 - \frac{\left|[r + \gamma z_j]_{V_\text{MIN}}^{V_\text{MAX}} - z_i\right|}{\Delta z}\right]_0^1 p_j(s',a').$$

The network is trained to minimize the Kullbeck-Leibler divergence between the current distribution and the (mapped) distribution of the one-step update

$$D_\text{KL}\Big(\Phi\big(R + \gamma Z_{\bar{\boldsymbol\theta}}\big(s', \arg\max_{a'} \mathbb{E} Z_{\bar{\boldsymbol\theta}}(s',a')\big)\big) \Big\| Z_{\boldsymbol\theta}\big(s,a\big)\Big).$$

Distributional RL

Algorithm 1 Categorical Algorithm

```
input  A transition $x_t, a_t, r_t, x_{t+1}, \gamma_t \in [0, 1]$
  $Q(x_{t+1}, a) := \sum_i z_i p_i(x_{t+1}, a)$
  $a^* \leftarrow \arg\max_a Q(x_{t+1}, a)$
  $m_i = 0, \quad i \in 0, \dots, N-1$
  for $j \in 0, \dots, N-1$ do
    # Compute the projection of $\hat{\mathcal{T}} z_j$ onto the support $\{z_i\}$
    $\hat{\mathcal{T}} z_j \leftarrow [r_t + \gamma_t z_j]_{V_\text{MIN}}^{V_\text{MAX}}$
    $b_j \leftarrow (\hat{\mathcal{T}} z_j - V_\text{MIN})/\Delta z$  # $b_j \in [0, N-1]$
    $l \leftarrow \lfloor b_j \rfloor, u \leftarrow \lceil b_j \rceil$
    # Distribute probability of $\hat{\mathcal{T}} z_j$
    $m_l \leftarrow m_l + p_j(x_{t+1}, a^*)(u - b_j)$
    $m_u \leftarrow m_u + p_j(x_{t+1}, a^*)(b_j - l)$
  end for
output  $-\sum_i m_i \log p_i(x_t, a_t)$  # Cross-entropy loss
```

Algorithm 1 of "A Distributional Perspective on Reinforcement Learning" by Marc G. Bellemare et al.

Distributional RL

	Mean	**Median**	**> H.B.**	**> DQN**
DQN	228%	79%	24	0
DDQN	307%	118%	33	43
DUEL.	373%	151%	37	50
PRIOR.	434%	124%	39	48
PR. DUEL.	592%	172%	39	44
C51	**701%**	**178%**	**40**	**50**

Figure 6 of "A Distributional Perspective on Reinforcement Learning" by Marc G. Bellemare et al.

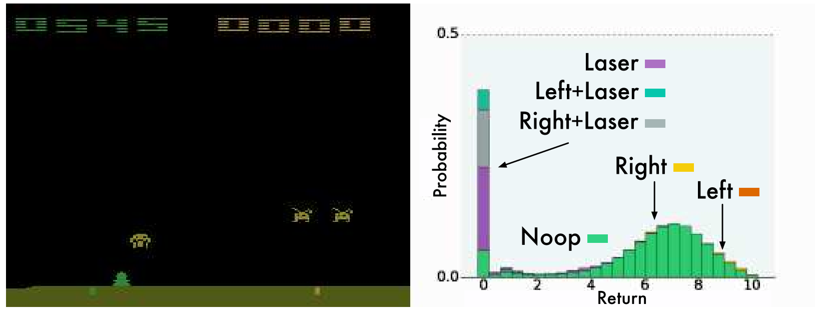


Figure 4. Learned value distribution during an episode of SPACE INVADERS. Different actions are shaded different colours. Returns below 0 (which do not occur in SPACE INVADERS) are not shown here as the agent assigns virtually no probability to them.

Figure 4 of "A Distributional Perspective on Reinforcement Learning" by Marc G. Bellemare et al.

Distributional RL

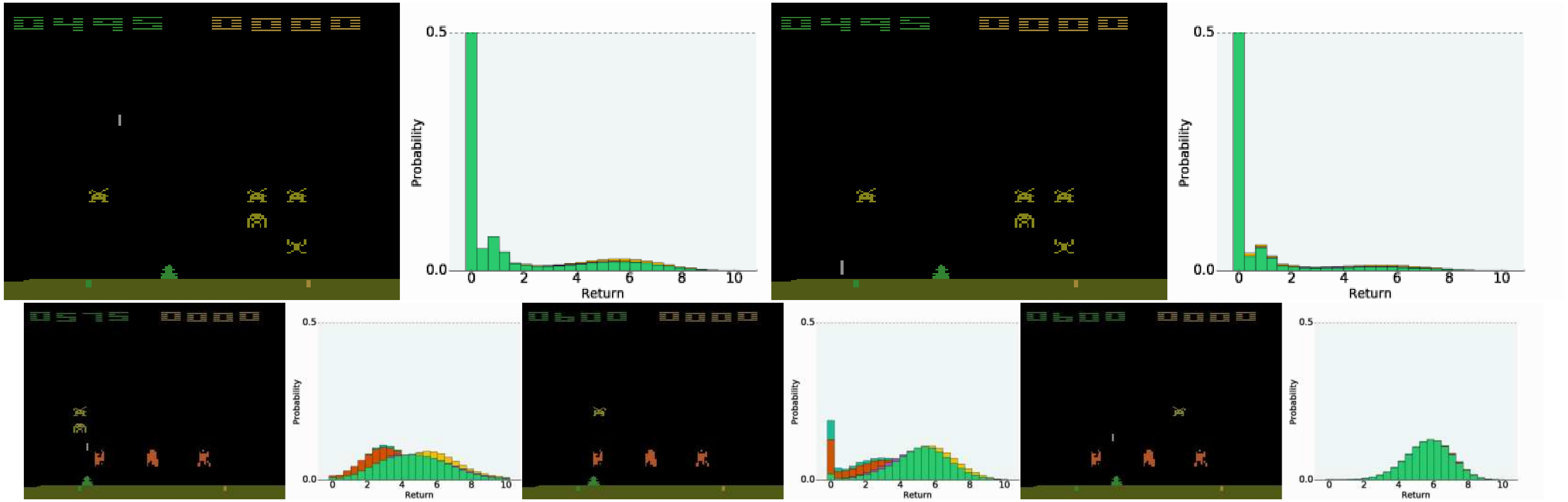


Figure 18. SPACE INVADERS: Top-Left: Multi-modal distribution with high uncertainty. Top-Right: Subsequent frame, a more certain demise. Bottom-Left: Clear difference between actions. Bottom-Middle: Uncertain survival. Bottom-Right: Certain success.

Figure 18 of "A Distributional Perspective on Reinforcement Learning" by Marc G. Bellemare et al.

Rainbow DQN Extensions

Distributional RL

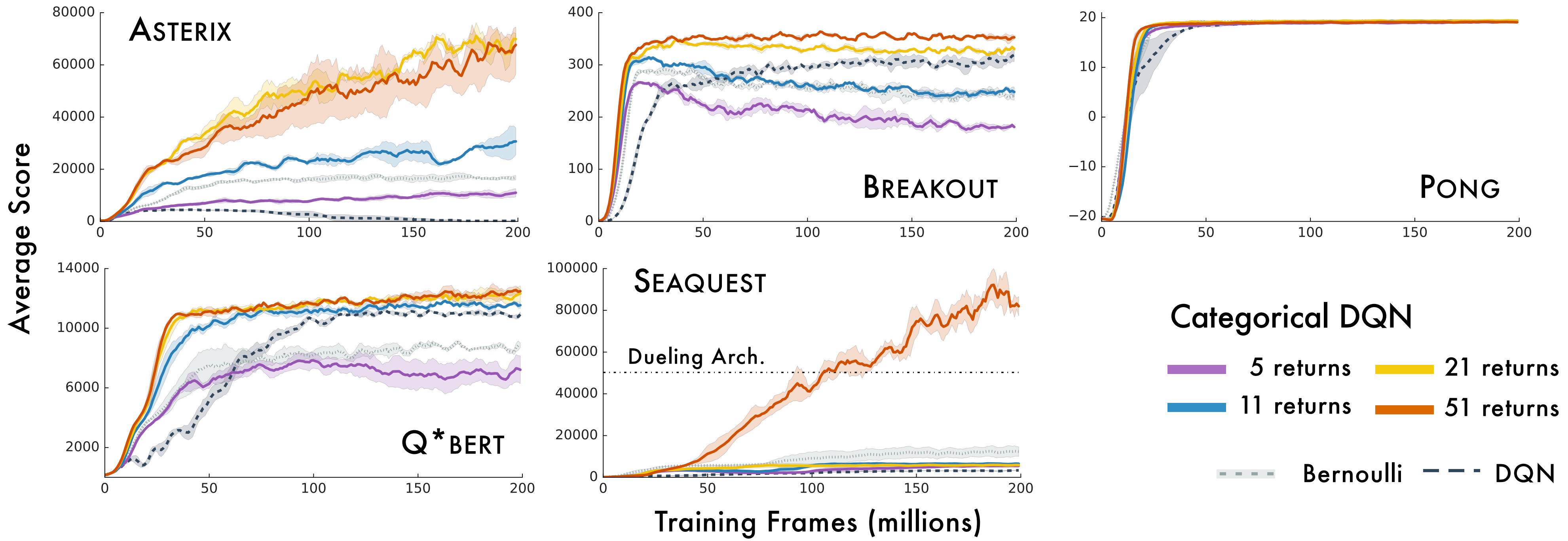


Figure 3. Categorical DQN: Varying number of atoms in the discrete distribution. Scores are moving averages over 5 million frames.

Figure 3 of "A Distributional Perspective on Reinforcement Learning" by Marc G. Bellemare et al.