Theory and Methods for Reinforcement Learning

Prof. Volkan Cevher
volkan.cevher@epfl.ch

Lecture 6: Value-based Methods for Deep RL

Laboratory for Information and Inference Systems (LIONS)
École Polytechnique Fédérale de Lausanne (EPFL)

EE-618 (Spring 2020)

License Information for Reinforcement Learning Slides

- This work is released under a Creative Commons License with the following terms:
- **Attribution**
  - The licensor permits others to copy, distribute, display, and perform the work. In return, licensees must give the original authors credit.
- **Non-Commercial**
  - The licensor permits others to copy, distribute, display, and perform the work. In return, licensees may not use the work for commercial purposes – unless they get the licensor's permission.
- **Share Alike**
  - The licensor permits others to distribute derivative works only under a license identical to the one that governs the licensor's work.
- Full Text of the License

Outline

- This class:
  1. Value-based Methods for Deep RL
- Next class:
  1. Policy Gradient Methods for Deep RL

Recommended reading

- V. Mnih, et al, *Human-level control through deep reinforcement learning*, Nature 518.7540 (2015): 529.
- H. Van Hasselt, A. Guez, and D. Silver, *Deep reinforcement learning with double q-learning*, AAAI, 2016.
- T. Schaul, et al, *Prioritized experience replay*, arXiv preprint arXiv:1511.05952 (2015).

Motivation

Motivation

How to implement Q-learning so that it can be used with complex function approximators like deep neural networks?

Recap

- Discounted return: $R_t = \sum_{\tau=t}^{\infty} \gamma^{\tau-t} r_\tau$
- State-action value function: $Q^\pi(s,a) = \mathbb{E}\left[R_t \mid s_t = s, a_t = a, \pi\right]$
- State value function: $V^\pi(s) = \mathbb{E}_{a\sim\pi(s)}\left[Q^\pi(s,a)\right]$
- Advantage function: $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$
- Optimal state-action value function: $Q^*(s,a) = \max_\pi Q^\pi(s,a)$
- Optimal policy: $\pi^*(s) = \arg\max_{a'} Q^*(s,a')$
- Optimal state value function: $V^*(s) = \max_a Q^*(s,a)$
- Bellman expectation: $Q^\pi(s,a) = \mathbb{E}_{s'}\left[r + \gamma \mathbb{E}_{a'\sim\pi(s')}\left[Q^\pi(s',a')\right] \mid s,a,\pi\right]$
- Bellman optimality: $Q^*(s,a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q^*(s',a') \mid s,a\right]$

Recap : Tabular Q-learning Algorithm

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm Parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$

Initialize: $Q(s, a)$, for all $s \in \mathcal{S}^+$ and $a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$; $i \leftarrow 0$

Loop for each episode:

Initialize state s_i

Loop for each step of episode:

Choose a_i from s_i using policy based on Q (e.g., ϵ-greedy)

Take action a_i, observe reward $r_i = r(s_i, a_i)$ and next state s_i'

$y_i \leftarrow r_i + \gamma \max_a Q(s_i', a)$

$Q(s_i, a_i) \leftarrow Q(s_i, a_i) - \alpha \{Q(s_i, a_i) - y_i\}$

$s_{i+1} \leftarrow s_i'$

$i \leftarrow i + 1$

until s_i is terminal

Motivation of Q-learning with Function Approximator

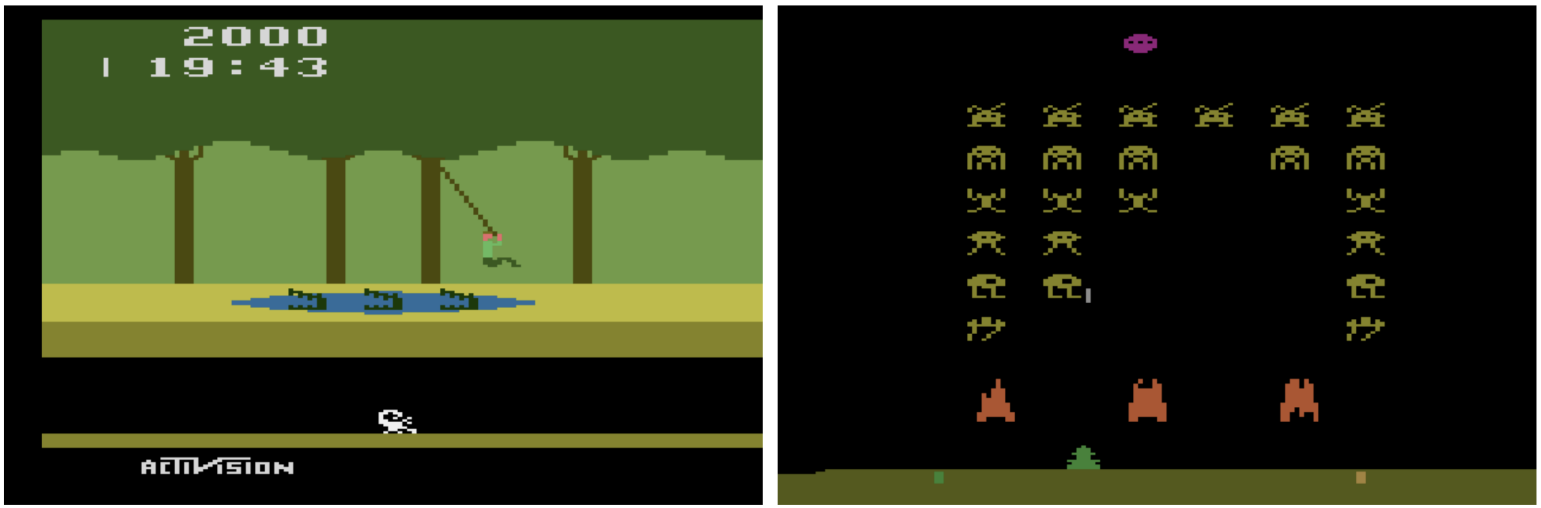


Figure: Screenshots of PITFALL! and SPACE INVADERS.

- In real-life case such as Atari, the size of state space prohibits exhaustive search
  - Tabular methods become insufficient and unsuitable
- Looking ahead one second requires $18^{60} \approx 10^{75}$ simulation steps

Motivation

Consider Q function as a parameteric function Q_θ and learn θ, instead of learning $Q(s, a)$ in tabular Q-learning.

Q-Learning with Function Approximator

- Learning a parametric Q-function: $Q_\theta(s,a)$
- Define: $\text{target}(s') = r(s,a) + \gamma \max_{a'} Q_\theta(s',a')$
- Update: $\theta_{k+1} \leftarrow \theta_k - \alpha \nabla_\theta \mathbb{E}_{s' \sim P(s'|s,a)} \left[\left(Q_\theta(s,a) - \text{target}(s') \right)^2 \right] \Big|_{\theta=\theta_k}$
- For tabular function, $\theta \in \mathbb{R}^{\mathcal{S} \times \mathcal{A}}$, we recover:

$$Q_{k+1}(s,a) \leftarrow (1-\alpha)\, Q_k(s,a) + \alpha\, \text{target}(s'),$$

which converges to optimal values under the conditions discussed in Lecture 4.

Online Q-Iteration Algorithm

Online Q-Iteration

Algorithm Parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$

Initialize: $Q_\theta(s, a)$, for all $s \in \mathcal{S}^+$ and $a \in \mathcal{A}(s)$, arbitrarily except that $Q_\theta(\text{terminal}, \cdot) = 0$; $i \leftarrow 0$

Loop for each episode:

Initialize state s_i

Loop for each step of episode:

Choose a_i from s_i using policy based on Q_θ (e.g., ϵ-greedy)

Take action a_i, observe reward $r_i = r(s_i, a_i)$ and next state s'_i

$y_i \leftarrow r_i + \gamma \max_a Q_\theta(s'_i, a)$

$\theta \leftarrow \theta - \alpha \{Q_\theta(s_i, a_i) - y_i\} \frac{dQ_\theta}{d\theta}(s_i, a_i)$

$s_{i+1} \leftarrow s'_i$

$i \leftarrow i + 1$

until s_i is terminal

Full Fitted Q-Iteration Algorithm

Full Fitted Q-Iteration [4]

Algorithm Parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$

Initialize: $Q_\theta(s, a)$, for all $s \in \mathcal{S}^+$ and $a \in \mathcal{A}(s)$, arbitrarily except that $Q_\theta(\text{terminal}, \cdot) = 0$; $\mathcal{D} \leftarrow$ empty; $i \leftarrow 0$

Loop for each episode:

Initialize state s_i

Loop for each step of episode:

Choose a_i from s_i using policy based on Q_θ (e.g., ϵ-greedy)

Take action a_i, observe reward $r_i = r(s_i, a_i)$ and next state s'_i

Collect samples $\mathcal{D} \leftarrow \mathcal{D} \cup (s_i, a_i, r_i, s'_i)$

$s_{i+1} \leftarrow s'_i$

$i \leftarrow i + 1$

If it's time to update then:

For however many updates do:

$y_j \leftarrow r_j + \gamma \max_a Q_\theta(s'_j, a)$, for all $(s_j, a_j, r_j, s'_j) \in \mathcal{D}$

$\theta \leftarrow \arg\min_\theta \frac{1}{2} \sum_j \left(Q_\theta(s_j, a_j) - y_j\right)^2$

$\mathcal{D} \leftarrow$ empty

until s_i is terminal

Q-function with linear function approximation (on ATARI games)

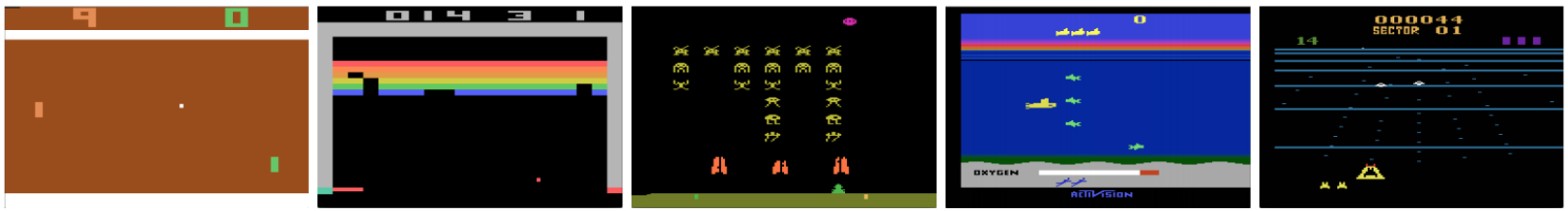


Figure: Screenshots from five Atari 2600 Games: (Left-to-right) Pong, Breakout, Space Invaders, Seaquest, Beam Rider

- 49 ATARI 2600 games.
- From pixels to actions.
- The change in score is the reward.
- Same algorithm.
- Same function approximator.
- Same hyperparameters.

Q-function with linear function approximation (on ATARI games)

Game	Basic	BASS	DISCO	LSH	RAM	Random	Const	Perturb	Human
ASTERIX	862	860	755	**987**	943	288	650	338	620
SEAQUEST	579	**665**	422	509	594	108	160	451	156
BOXING	-3	16	12	10	**44**	-1	-25	-10	-2
H.E.R.O.	6053	**6459**	2720	3836	3281	712	0	148	6087
ZAXXON	1392	2069	70	**3365**	304	0	0	2	820

Figure: Reinforcement Learning results for selected games [1]

- BASIC, BASS, DISCO, LSH, RAM are five different sets of features to use with linear function approximation.
- State-of-the art feature sets are heavily depends on the game.
- Better or similar performance comparing with Human agent (who had never previously played Atari), not providing accurate human-level benchmarks.

- Feature engineering is hard and boring, can't we just use neural networks?
  - No, it is unstable or divergent with a nonlinear function approximation
  - Why, and can we solve the issue?

Issues in Online Q-Iteration

- Online Q-iteration:
  1. take some action a_i and observe (s_i, a_i, r_i, s_i')
  2. $\theta \leftarrow \theta - \alpha \left\{ Q_\theta(s_i, a_i) - \left[r_i + \gamma \max_a Q_\theta(s_i', a) \right] \right\} \frac{dQ_\theta}{d\theta}(s_i, a_i)$
- Issues:
  1. Sequential states are strongly correlated
  2. Q-learning is not gradient descent (through target value)

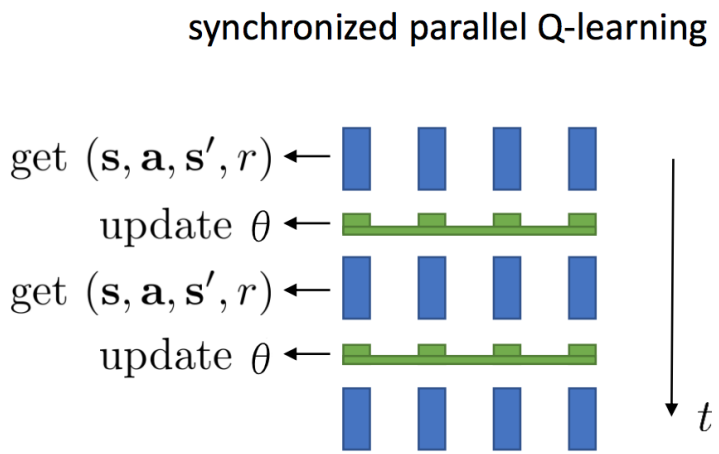


Figure: Addressing correlated samples issues in online Q-learning.

Q-Learning with Replay Buffer (I)

- Full fitted Q-iteration:
  1. collect dataset $\mathcal{D} = \left\{(s_i, a_i, r_i, s_i')\right\}$ using some policy
  2. for $k = 1, \ldots, K$:
     2.1 $y_j \leftarrow r_j + \gamma \max_a Q_\theta(s_j', a)$, for all $(s_j, a_j, r_j, s_j') \in \mathcal{D}$
     2.2 $\theta \leftarrow \arg\min_\theta \frac{1}{2} \sum_j (Q_\theta(s_j, a_j) - y_j)^2$
  3. $\mathcal{D} \leftarrow$ empty

- Full Q-learning with replay buffer:
  1. collect dataset $\left\{(s_i, a_i, r_i, s_i')\right\}$ using some policy, add it to $\mathcal{D}$
  2. for $k = 1, \ldots, K$:
     2.1 sample a batch (s_j, a_j, r_j, s_j') from $\mathcal{D}$
     2.2 $\theta \leftarrow \theta - \alpha \sum_j \left\{Q_\theta(s_j, a_j) - \left[r_j + \gamma \max_a Q_\theta(s_j', a)\right]\right\} \frac{dQ_\theta}{d\theta}(s_j, a_j)$

Q-Learning with Replay Buffer (II)

- Samples are no longer correlated
- Multiple samples in the batch (low-variance gradient)
- Q-learning is not gradient descent - this is still a problem!

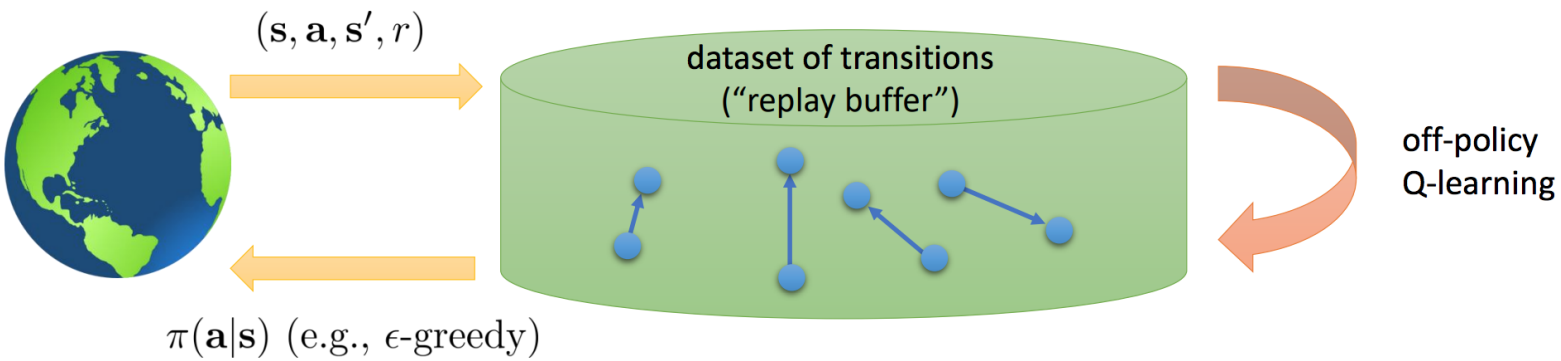


Figure: Full Q-learning with replay buffer.

Q-Learning and Regression

- Full Q-learning with replay buffer:

  1. collect dataset $\left\{(s_i, a_i, r_i, s_i')\right\}$ using some policy, add it to $\mathcal{D}$
  2. for $k = 1, \ldots, K$:
     2.1 sample a batch (s_j, a_j, r_j, s_j') from $\mathcal{D}$

     2.2 $\theta \leftarrow \theta - \underbrace{\alpha \sum_j \left\{ Q_\theta(s_j, a_j) - \left[r_j + \gamma \max_a Q_\theta(s_j', a) \right] \right\} \frac{dQ_\theta}{d\theta}(s_j, a_j)}_{\text{one gradient step, moving target}}$

- Full fitted Q-iteration:

  1. collect dataset $\mathcal{D} = \left\{(s_i, a_i, r_i, s_i')\right\}$ using some policy
  2. for $k = 1, \ldots, K$:
     2.1 $y_j \leftarrow r_j + \gamma \max_a Q_\theta(s_j', a)$, for all $(s_j, a_j, r_j, s_j') \in \mathcal{D}$

     2.2 $\theta \leftarrow \underbrace{\arg\min_\theta \frac{1}{2} \sum_j \left(Q_\theta(s_j, a_j) - y_j\right)^2}_{\text{perfectly well-defined, stable regression}}$
  3. $\mathcal{D} \leftarrow$ empty

Q-Learning with Target Network

- Changing the value of one action will change the value of other actions and similar states.
- The network can end up chasing its own tail because of bootstrapping.
- Q-learning with replay buffer and target network:
  1. save target network parameters $\theta' \leftarrow \theta$
  2. for $k = 1, \ldots, K$: % supervised regression
     - 2.1 collect dataset $\left\{(s_i, a_i, r_i, s_i')\right\}$ using some policy, add it to $\mathcal{D}$
     - 2.2 for $\ell = 1, \ldots, L$:
       - 2.2.1 sample a batch (s_j, a_j, r_j, s_j') from $\mathcal{D}$
       - 2.2.2 $\theta \leftarrow \theta - \alpha \sum_j \left\{ Q_\theta(s_j, a_j) - \left[r_j + \gamma \max_a Q_{\theta'}(s_j', a) \right] \right\} \frac{dQ_\theta}{d\theta}(s_j, a_j)$
- Targets don't change in inner loop!

Deep Q-Learning Algorithm (DQN)

- High-level idea – **make Q-learning look like supervised learning**.
- Two main ideas for stabilizing Q-learning.
- Apply Q-updates on batches of past experience instead of online:
  - **Experience replay** [2]
  - Previously used for better data efficiency.
  - Makes the data distribution more stationary.
- Use an older set of weights to compute the targets (**target network**):
  - Keeps the target function from changing too quickly.

$$L_i(\theta) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta) \right)^2 \right]$$

Deep Q-Learning Algorithm (DQN)

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

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 ε select a random action a_t

    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

    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 θ

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

  End For

End For

General View of DQN

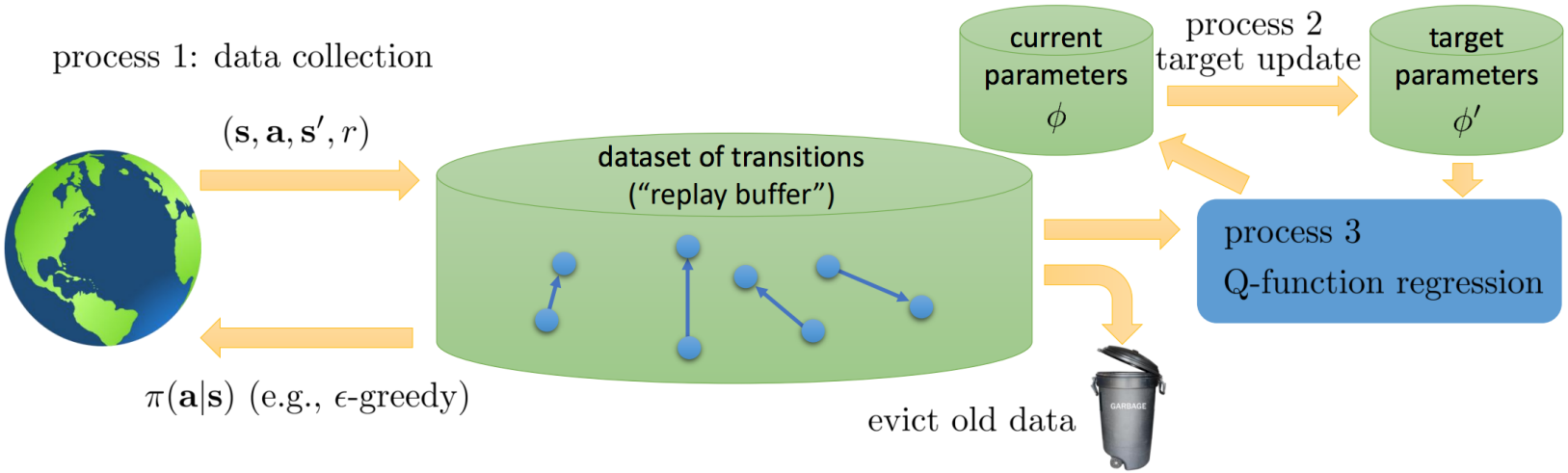


Figure: A more general view of DQN.

- Online Q-learning: evict immediately, process 1, process 2, and process 3 all run at the same speed
- DQN: process 1 and process 3 run at the same speed, process 2 is slow

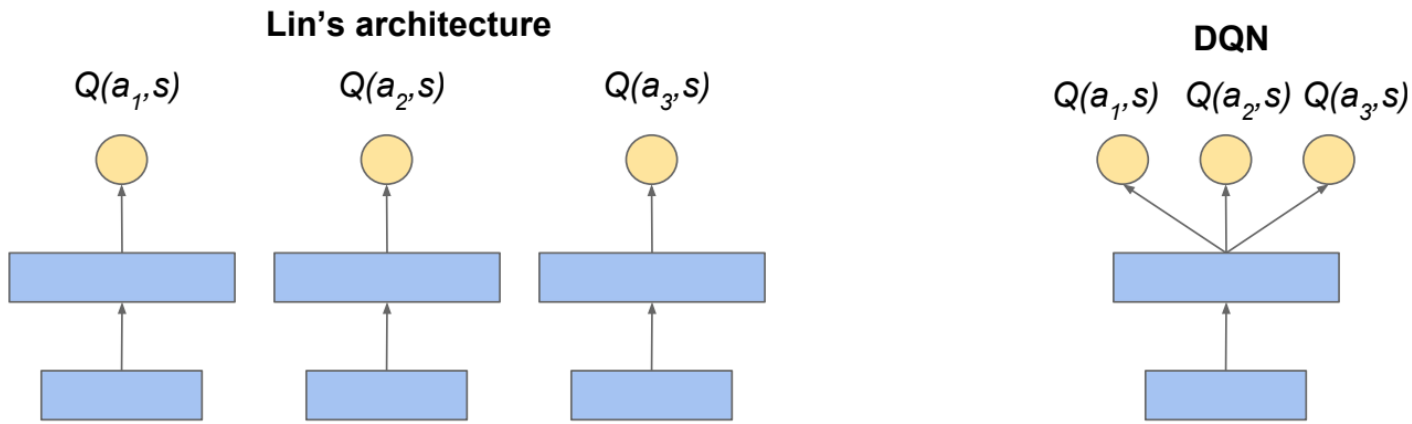
Implementation details of DQN

- Uses Huber loss instead of squared loss on Bellman error:

$$L_\delta(x) = \begin{cases} \frac{1}{2}x^2 & \text{for } |x| \leq \delta, \\ \delta\left(|x| - \frac{1}{2}\delta\right) & \text{otherwise.} \end{cases}$$

- Uses RMSProp instead of vanilla SGD.
  - Optimization in RL really matters.
- It helps to anneal the exploration rate.
  - Start ϵ at 1 and anneal it to 0.1 or 0.05 over the first million frames.

DQN Architecture



- Lin's networks did not share parameters among actions.

- Lin's architecture requires a separate forward pass to compute the Q-value of each action, resulting in a cost that scales linearly with the number of actions.

- DQN architecture has a separate output unit for each possible action.

- DQN architecture enables to compute Q-values for all possible actions in a given state with only a single forward pass through the network.

DQN Architecture

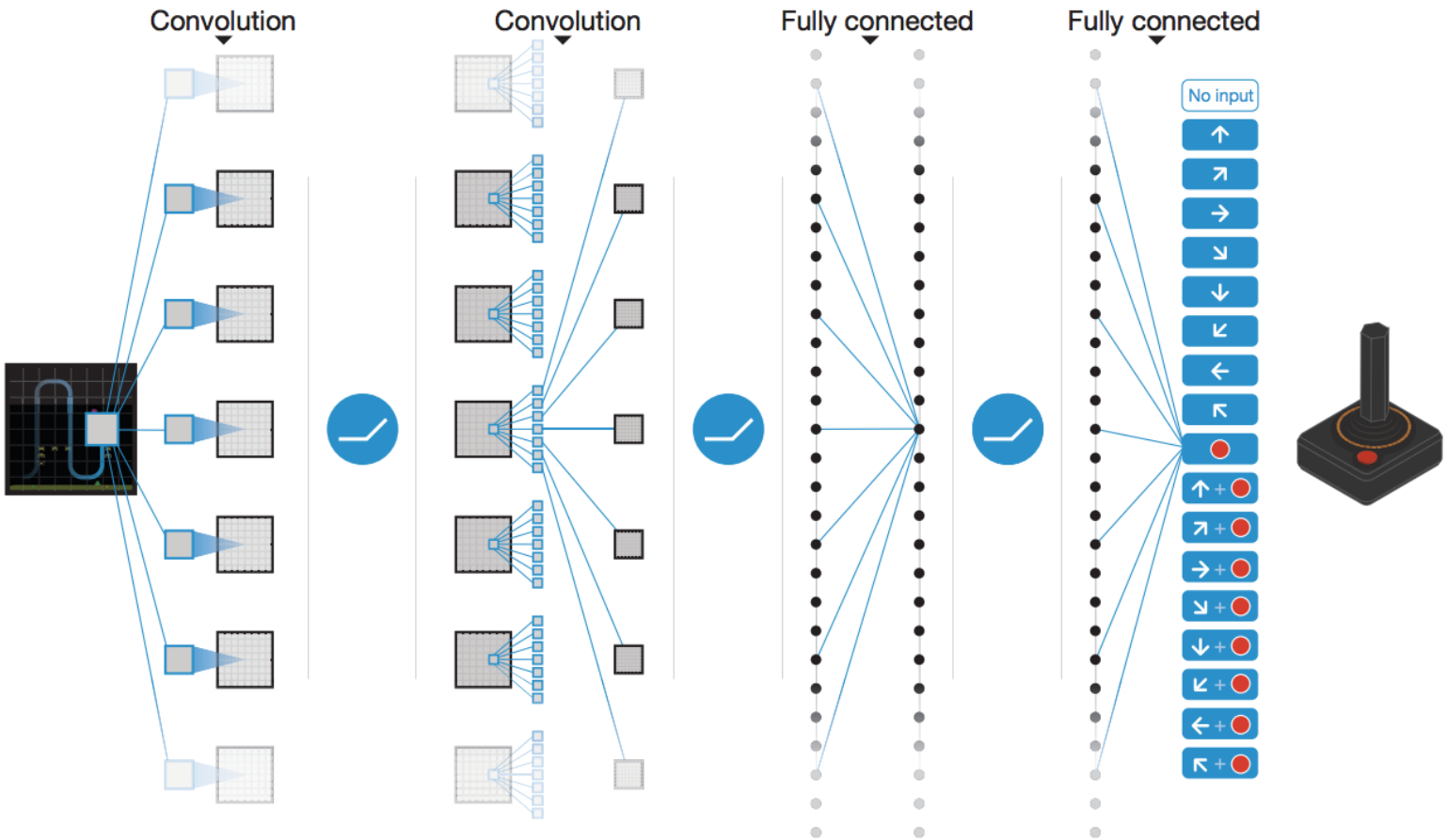
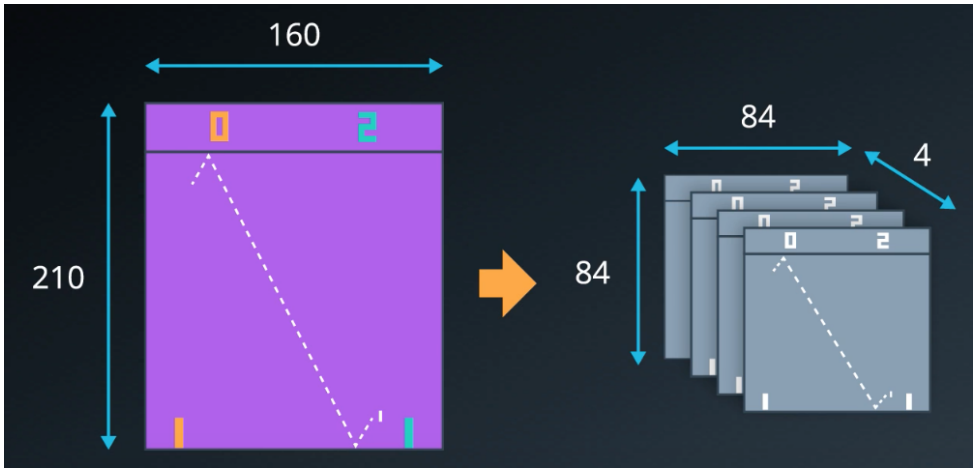
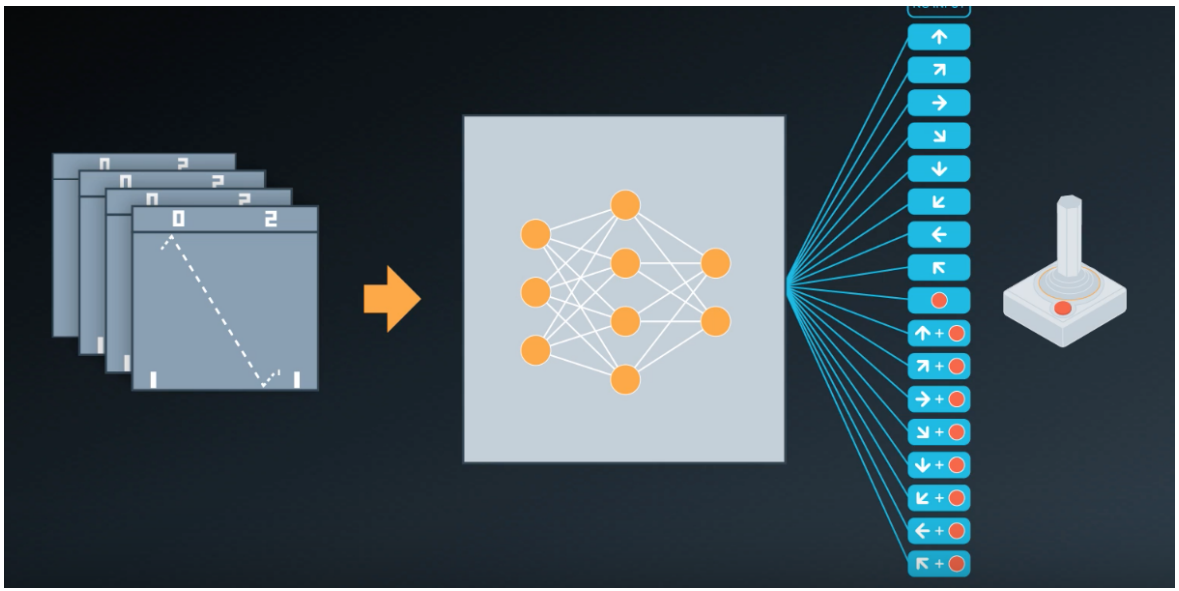


Figure: ATARI Network Architecture: History of frames as input. One output per action - expected reward for that action $Q(s, a)$.

DQN on ATARI [3]





DQN on ATARI

• DQN achieves much better performance than any other methods, even an expert human player

	B. Rider	Breakout	Enduro	Pong	Q*bert	Seaquest	S. Invaders
Random	354	1.2	0	−20.4	157	110	179
Sarsa [3]	996	5.2	129	−19	614	665	271
Contingency [4]	1743	6	159	−17	960	723	268
DQN	**4092**	**168**	**470**	**20**	**1952**	**1705**	**581**
Human	7456	31	368	−3	18900	28010	3690

Figure: Average total reward for various learning methods for a fixed number of steps.

DQN on ATARI

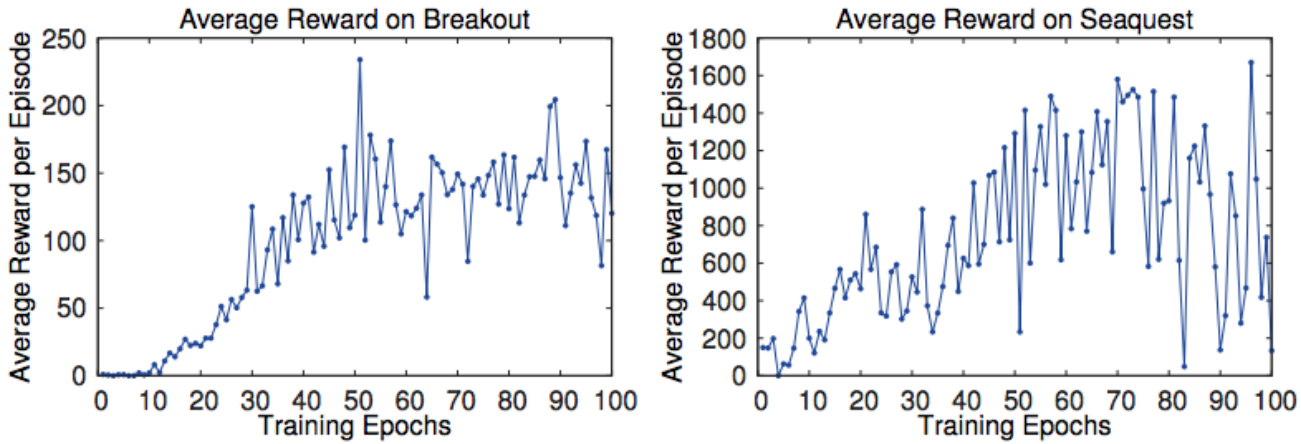


Figure: Average reward per episode on Breakout and Seaquest respectively during training. The statistics were computed by running an ϵ-greedy policy with $\epsilon = 0.05$ for 10000 steps.

DQN on ATARI

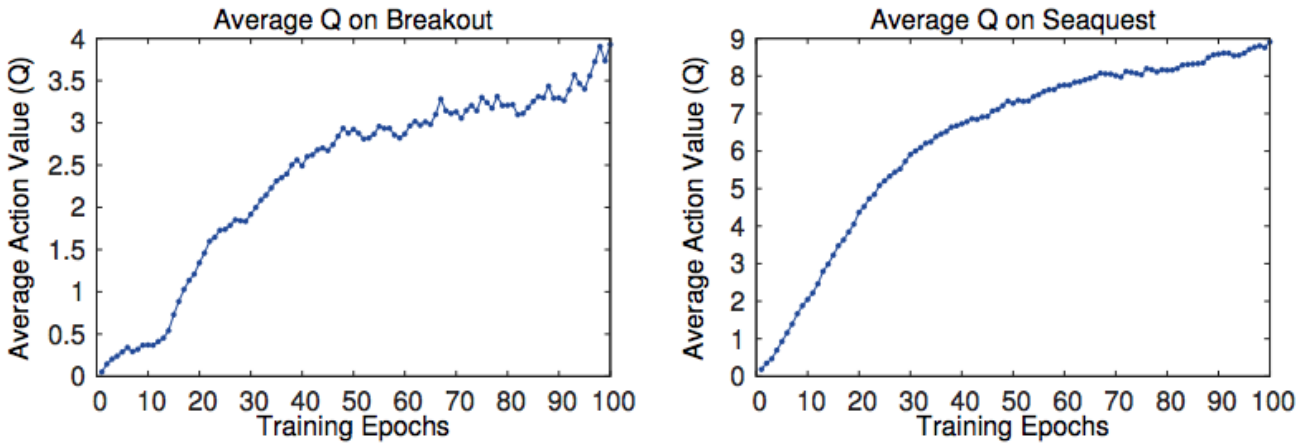


Figure: Average maximum predicted action-value of a held out set of states on Breakout and Seaquest respectively.

Motivation

Motivation

Now, we will study some improved variants of DQN.

Overestimation in Q-learning

- Target value $y_j \leftarrow r_j + \gamma \max_{a'} Q_{\theta'}(s'_j, a')$
- Given two random variables X_1 and X_2:

$$\mathbb{E}\left[\max\left(X_1, X_2\right)\right] \geq \max\left(\mathbb{E}\left[X_1\right], \mathbb{E}\left[X_2\right]\right)$$

- $Q_{\theta'}(s', a')$ is not perfect – hence $\max_{a'} Q_{\theta'}(s', a')$ *overestimates* the next value.
- Note that $\max_{a'} Q_{\theta'}(s', a') = Q_{\theta'}(s', \arg\max_{a'} Q_{\theta'}(s', a'))$
  - action selected according to $Q_{\theta'}$
  - value also comes from $Q_{\theta'}$
- Don't use the same network to choose the action and evaluate the value – use two networks!

Overestimation in Q-learning

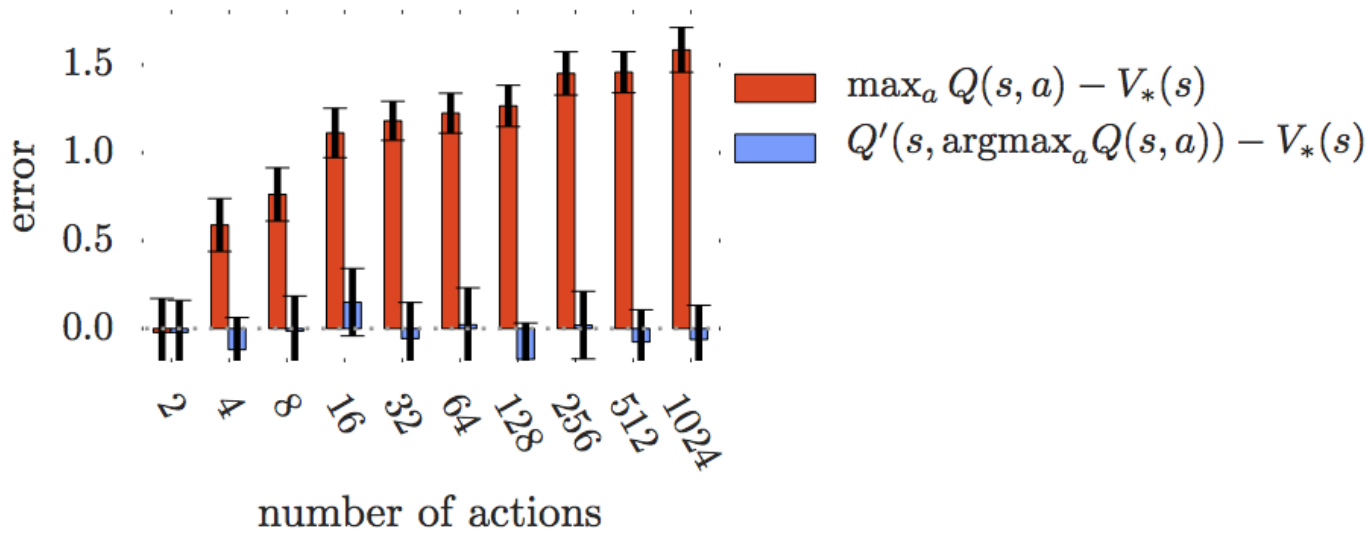
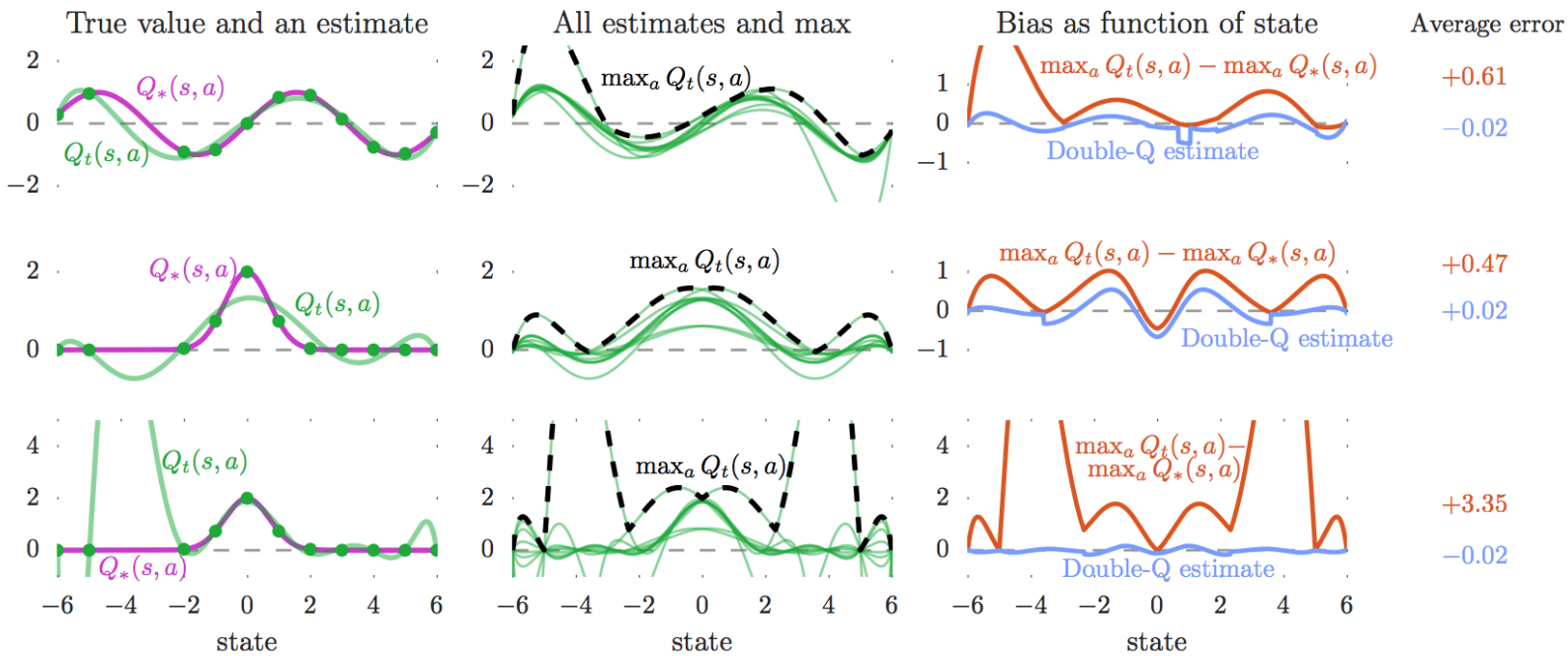


Figure: The action values are $Q(s,a) = V^*(s) + \epsilon_a$ and the errors $\{\epsilon_a\}_{a=1}^m$ are independent standard normal random variables. Q' was generated identically and independently.

Overestimation in Q-learning



Tabular Double Q-Learning Algorithm

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

Algorithm Parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$

Initialize: $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in \mathcal{S}^+$ and $a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$

Loop for each episode:

Initialize state S

Loop for each step of episode:

Choose A from S using policy based on $Q_1 + Q_2$ (e.g., ϵ-greedy)

Take action A, observe reward R and next state S'

With 0.5 probabilility:

$$Q_1(S, A) \leftarrow Q_1(S, A) + \alpha[R + \gamma Q_2(S', \arg\max_a Q_1(S', a)) - Q_1(S, A)]$$

Else:

$$Q_2(S, A) \leftarrow Q_2(S, A) + \alpha[R + \gamma Q_1(S', \arg\max_a Q_2(S', a)) - Q_2(S, A)]$$

$S \leftarrow S'$

until S is terminal

Double DQN [6]

- There is an upward bias in $\max_a Q(s, a; \theta)$
- DQN maintains two sets of weight θ and θ^-, so reduce bias by using:
  - θ for selecting the best action.
  - θ^- for evaluating the best action.
- Double DQN loss:

$$L_i(\theta) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}} \left[\left(r + \gamma Q(s', \arg\max_{a'} Q(s', a'; \theta_i); \theta_i^-) - Q(s, a; \theta) \right)^2 \right]$$

Double DQN Algorithm

Algorithm 1: Double DQN Algorithm.

input : $\mathcal{D}$ – empty replay buffer; θ – initial network parameters, θ^- – copy of θ
input : N_r – replay buffer maximum size; N_b – training batch size; N^- – target network replacement freq.
for *episode* $e \in \{1, 2, \ldots, M\}$ **do**
  Initialize frame sequence $\mathbf{x} \leftarrow ()$
  for $t \in \{0, 1, \ldots\}$ **do**
    Set state $s \leftarrow \mathbf{x}$, sample action $a \sim \pi_{\mathcal{B}}$
    Sample next frame x^t from environment $\mathcal{E}$ given (s, a) and receive reward r, and append x^t to $\mathbf{x}$
    if $|\mathbf{x}| > N_f$ **then** delete oldest frame $x_{t_{min}}$ from $\mathbf{x}$ **end**
    Set $s' \leftarrow \mathbf{x}$, and add transition tuple (s, a, r, s') to $\mathcal{D}$,
      replacing the oldest tuple if $|\mathcal{D}| \geq N_r$
    Sample a minibatch of N_b tuples $(s, a, r, s') \sim \text{Unif}(\mathcal{D})$
    Construct target values, one for each of the N_b tuples:
    Define $a^{\max}(s'; \theta) = \arg\max_{a'} Q(s', a'; \theta)$
    $$y_j = \begin{cases} r & \text{if } s' \text{ is terminal} \\ r + \gamma Q(s', a^{\max}(s'; \theta); \theta^-), & \text{otherwise.} \end{cases}$$
    Do a gradient descent step with loss $\|y_j - Q(s, a; \theta)\|^2$
    Replace target parameters $\theta^- \leftarrow \theta$ every N^- steps
  end
end

Double DQN

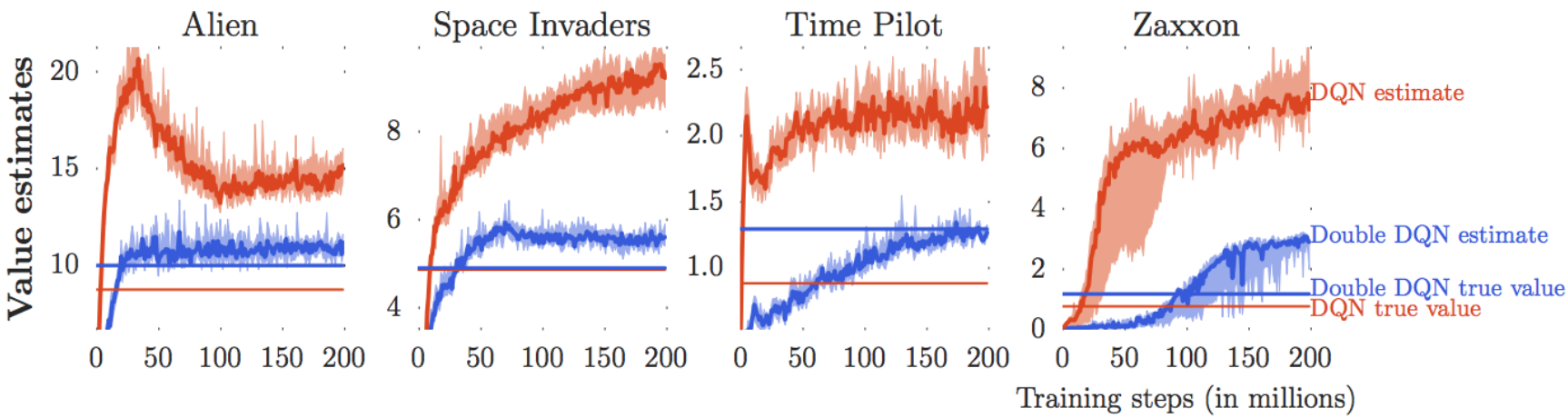


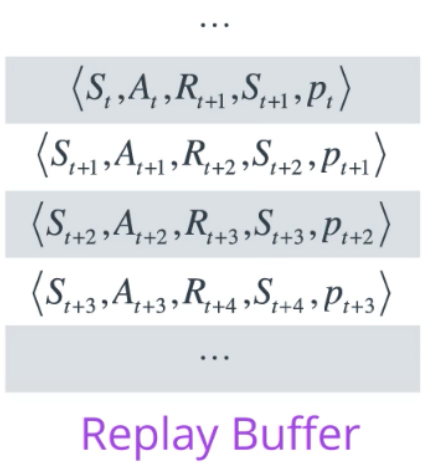
Figure: Value estimates by DQN (orange) and Double DQN (blue) on Atari games. The straight horizontal lines are computed by running the corresponding agents after learning concluded, and averaging the actual discounted return obtained from each visited state. These straight lines would match the learning curves at the right side of the plots if there is no bias.

Prioritized Experience Replay (PER)

- Replaying all transitions with equal probability is highly suboptimal.
- Replay transitions in proportion to absolute Bellman error:

$$\left| r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta) \right|$$

- Leads to much faster learning.



TD Error

$$\delta_t = R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w})$$

Priority

$$p_t = |\delta_t| + e$$

Sampling Probability

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

Modified Update Rule

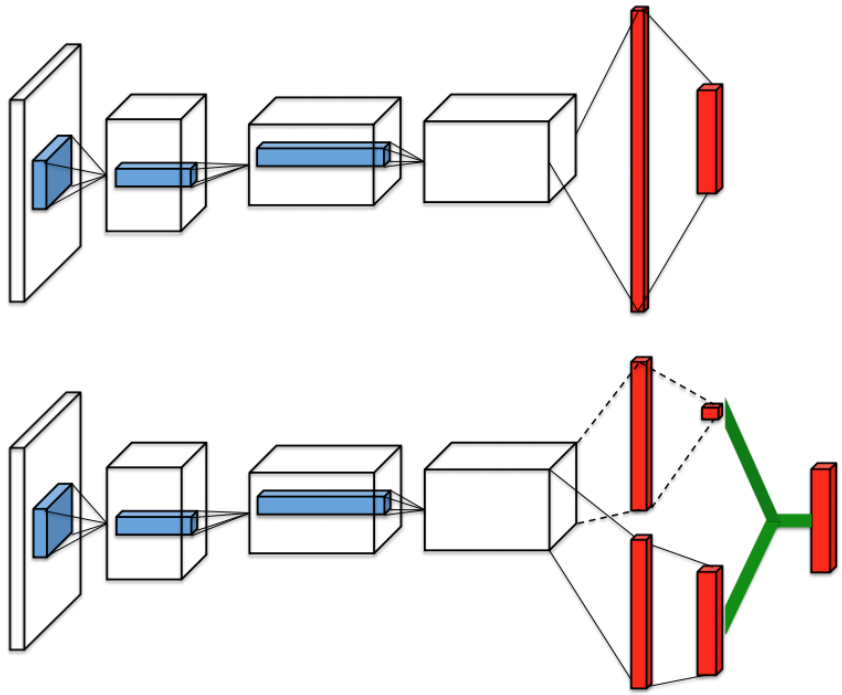
$$\Delta \mathbf{w} = \alpha \underbrace{\left(\frac{1}{N} \cdot \frac{1}{P(i)} \right)^b}_{\text{importance-sampling weight}} \delta_i \nabla_{\mathbf{w}} \hat{q}(S_i, A_i, \mathbf{w})$$

Double DQN with PER [5]

Algorithm 1 Double DQN with proportional prioritization

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

Dueling DQN [7]



- Value-Advantage decomposition of Q: $Q^\pi(s,a) = A^\pi(s,a) + V^\pi(s)$
- θ denotes the parameters of the convolutional layers, while α and β are the parameters of the two streams of fully-connected layers.
- Dueling DQN:

$$Q(s,a;\theta,\alpha,\beta) \;=\; V(s;\theta,\beta) + A(s,a;\theta,\alpha) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a';\theta,\alpha)$$

References

[1] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling.
The arcade learning environment: An evaluation platform for general agents.
Journal of Artificial Intelligence Research, 47:253–279, 2013.

[2] Long-Ji Lin.
Reinforcement learning for robots using neural networks.
Technical report, Carnegie-Mellon University, 1993.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller.
Playing atari with deep reinforcement learning.
arXiv preprint arXiv:1312.5602, 2013.

[4] Martin Riedmiller.
Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method.
In *European Conference on Machine Learning*, pages 317–328. Springer, 2005.

[5] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver.
Prioritized experience replay.
arXiv preprint arXiv:1511.05952, 2015.

[6] Hado Van Hasselt, Arthur Guez, and David Silver.
Deep reinforcement learning with double q-learning.
In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[7] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas.
Dueling network architectures for deep reinforcement learning.
In *International Conference on Machine Learning*, pages 1995–2003, 2016.