Deep Reinforcement Learning

Armin Biess

Ben-Gurion University of the Negev

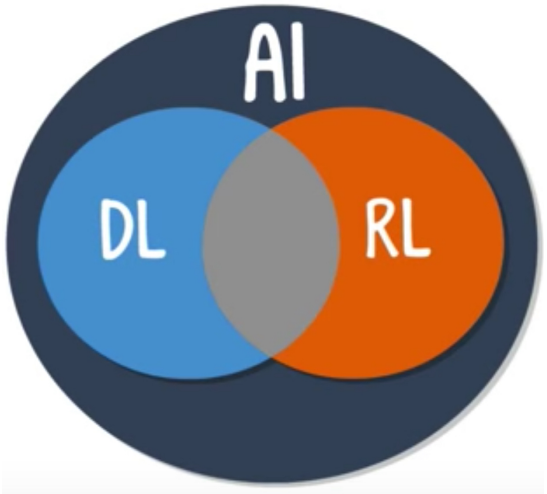
Deep Q-Network (DQN)

Deep Reinforcement Learning (DRL)

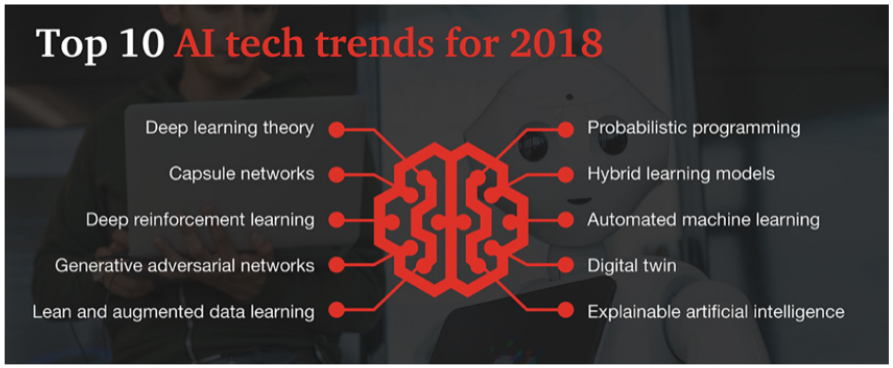
- **Deep reinforcement learning (DRL)** refers to the combination of reinforcement learning (RL) with **deep learning (DL)**, i.e., methods of supervised learning using deep neural networks for function approximation

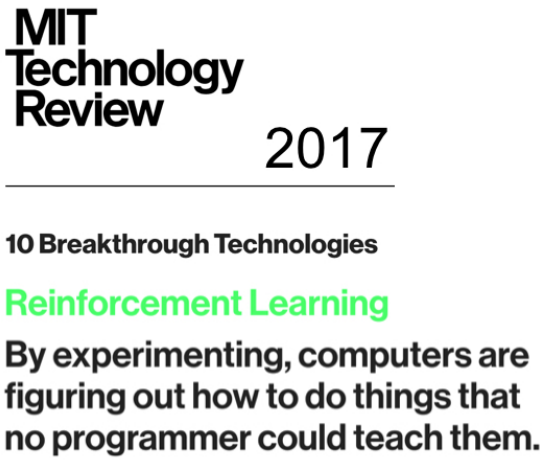
  S+B: DRL = reinforcement learning with function approximation by deep artificial neural networks

- DRL is a general-purpose framework for **artificial intelligence (AI)**



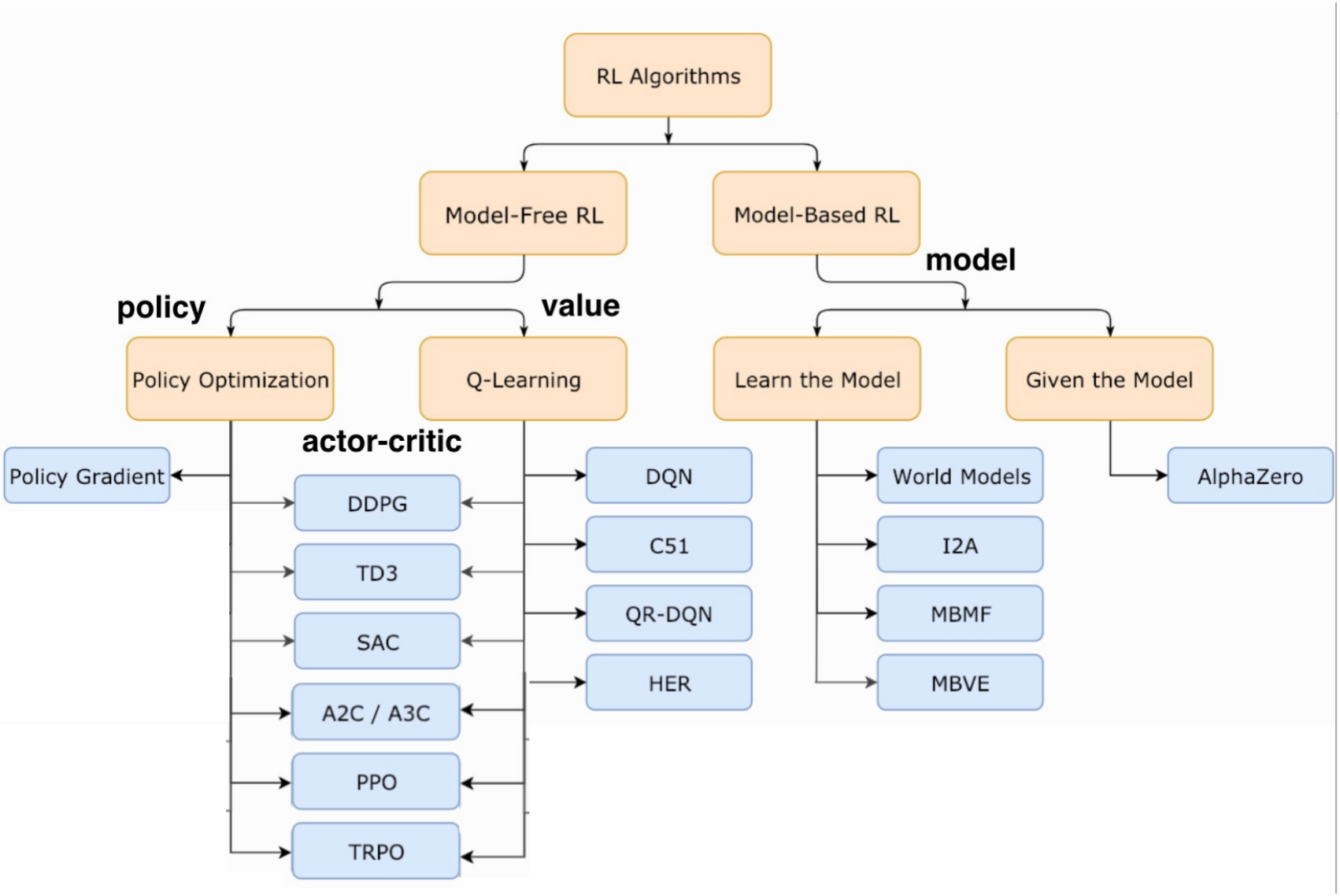
Deep Reinforcement Learning Revolution? Why now?





- theoretical: methodological **breakthroughs in deep learning**
- hardware: **exponential increase of computational power** with the use of GPUs and distributed computing (**movie**)
- software: **a growing eco-system of softwares** such as Tensorflow, PyTorch, OpenAI, MuJoCo and **datasets** such as ImageNet, MNIST
- money: **big players** are involved (Google, Facebook, Microsoft, Amazon, Tesla, . . .)

RL Taxonomy

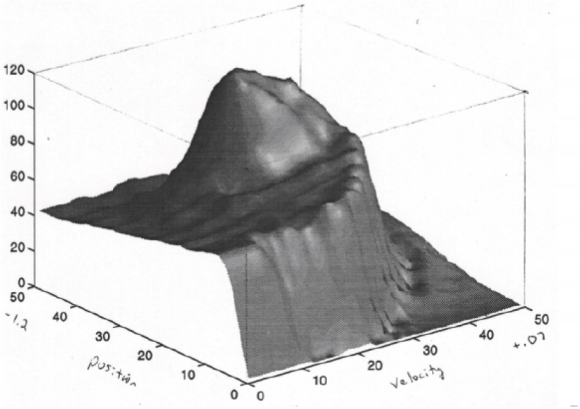

RL Algorithms
Model-Free RL
Model-Based RL
model
policy
value
Policy Optimization
Q-Learning
Learn the Model
Given the Model
actor-critic
Policy Gradient
DDPG
TD3
SAC
A2C / A3C
PPO
TRPO
DQN
C51
QR-DQN
HER
World Models
I2A
MBMF
MBVE
AlphaZero

DRL: Function Approximation

- DRL = reinforcement learning with function approximation by deep artificial neural networks
- in DRL: state and actions become **continuous**; need to represent a policy and/or value function over continuous domains
- we cannot enumerate all values and use tabular solutions

	1	2	3	4
a	0.812	0.868	0.918	+1
b	0.762		0.660	-1
c	0.705	0.655	0.611	0.388

optimal values for gridworld



negative optimal values (cost) for mountain car

DRL: Linear vs Non-Linear FAP

linear function approximation (see lecture 6)

- for example, Gaussian RBF $\boldsymbol{\phi}(\boldsymbol{x}) = (\phi_1(\boldsymbol{x}), \phi_2(\boldsymbol{x}), \ldots, \phi_n(\boldsymbol{x}))^T$
- $\boldsymbol{y}(\boldsymbol{x}) = \boldsymbol{w}^T \cdot \boldsymbol{\phi}(\boldsymbol{x})$
- learning process tunes weights $\boldsymbol{w}$, features $\boldsymbol{\phi}(\boldsymbol{x})$ remain fixed

nonlinear function approximation

- for example, ANN
- fwd 2-layer NN: $\boldsymbol{y}(\boldsymbol{x}) = \tilde{\boldsymbol{f}}(\tilde{\boldsymbol{W}} \cdot \boldsymbol{f}(\boldsymbol{W}\boldsymbol{x}))$,
  $\boldsymbol{W}, \boldsymbol{W}'$ weight matrices
- consider special case: $\tilde{\boldsymbol{f}} = \mathbf{1}$ gives $\boldsymbol{y}(\boldsymbol{x}) = \tilde{\boldsymbol{W}} \cdot \boldsymbol{f}(\boldsymbol{W}\boldsymbol{x})$
- learning (through backpropagation) process tunes weights
  $\boldsymbol{\theta} = (\tilde{\boldsymbol{W}}, \boldsymbol{W})$ **and** nonlinear features $\boldsymbol{f}(\boldsymbol{W}\boldsymbol{x})$

RL: Early Success with ANN

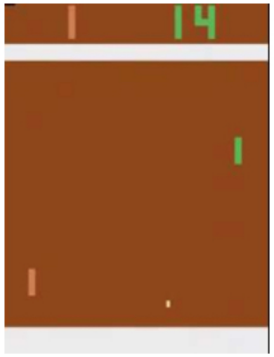
- RL agent using NN beats world champion in backgammon (TD-gammon)
- but is was shown that RL with function appoximation can diverge

Tesauro, G. (1995) *Temporal learning in TD-gammon*. Communications of the ACM, 38(3):58-68

Baird, L., (1994) *Reinforcement learning in continuous time: advantage updating*, Proc. International Conference on Neural Networks, Orlando, FL

DQN: Atari Games

- Atari 2600 game console, popular in the 80's
- challenging AI platform
- many games (Pong, Breakout, Space Invaders, Pacman, ...)
- **end-to-end reinforcement learning**: from images to actions
- **continuous** states, **discrete** actions
- DeepMind: 49 games, same algorithm, same network architecture, same hyperparameters (γ, α, exploration parameters, etc) for all games
- learns a different policy for each of the 49 game (weights of ANN are reset to random values before learning on each game)
- original paper: Minh et al, 7 games, 2013 (without target network) *Playing Atari with Deep Reinforcement Learning*
- revised paper: Minh et al, 49 games, 2015 (wih target network) *Human-Level Control Through Deep Reinforcement Learning*



Q-learning

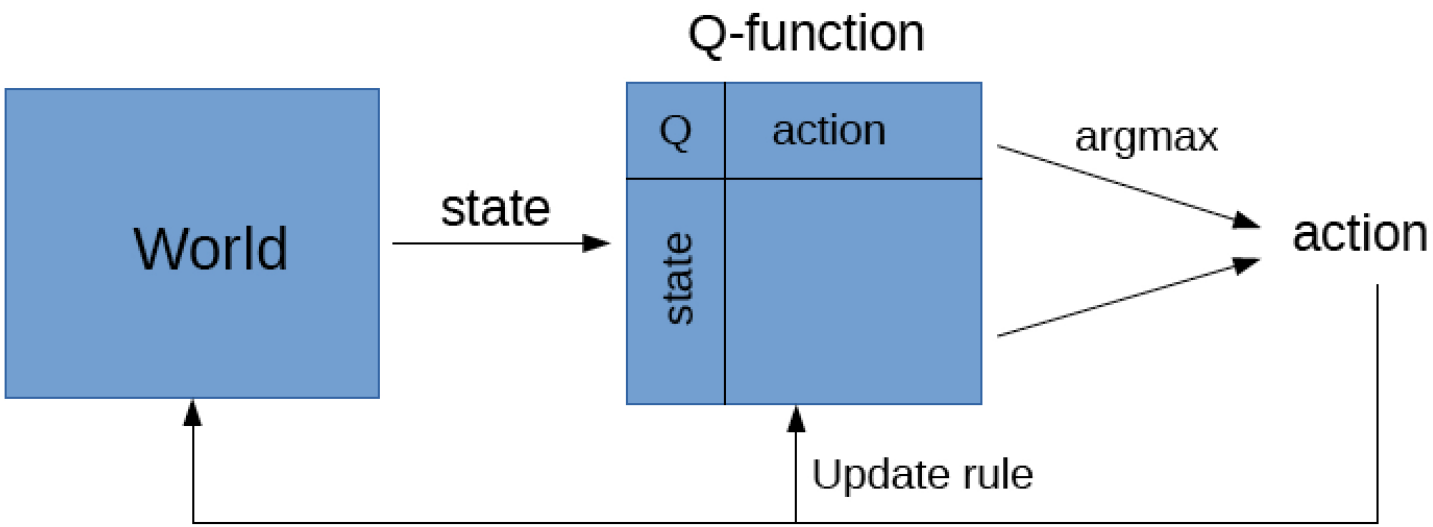
Algorithm: Q-learning (Sarsamax)

1: **Input:** small positive fraction α, GLIE $\{\varepsilon_i\}$
2: **Output:** value function $Q(\approx q^*$ if *num_episodes* is large enough)
3: Initialize Q arbitrarily (e.g., $Q(s,a) = 0$ for all $s \in S^+$ and $a \in \mathcal{A}(s)$ and $Q(terminal_states, \cdot) = 0$)
4: **for** $i \leftarrow$ to num_episodes **do**
5: $\quad \varepsilon \leftarrow \varepsilon_i$
5: $\quad$ Observe S_0
6: $\quad$ Choose action A_0 using policy derived from Q (e.g., ε-greedy)
7: $\quad t \leftarrow 0$
8: $\quad$ **repeat**
10: $\quad\quad$ Choose action A_0 using policy derived from Q (e.g., ε-greedy)
11: $\quad\quad Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t))$
12: $\quad\quad t \leftarrow t + 1$
13: $\quad$ **until** S_t is terminal
14: **end**
15: **return** Q

Comments:

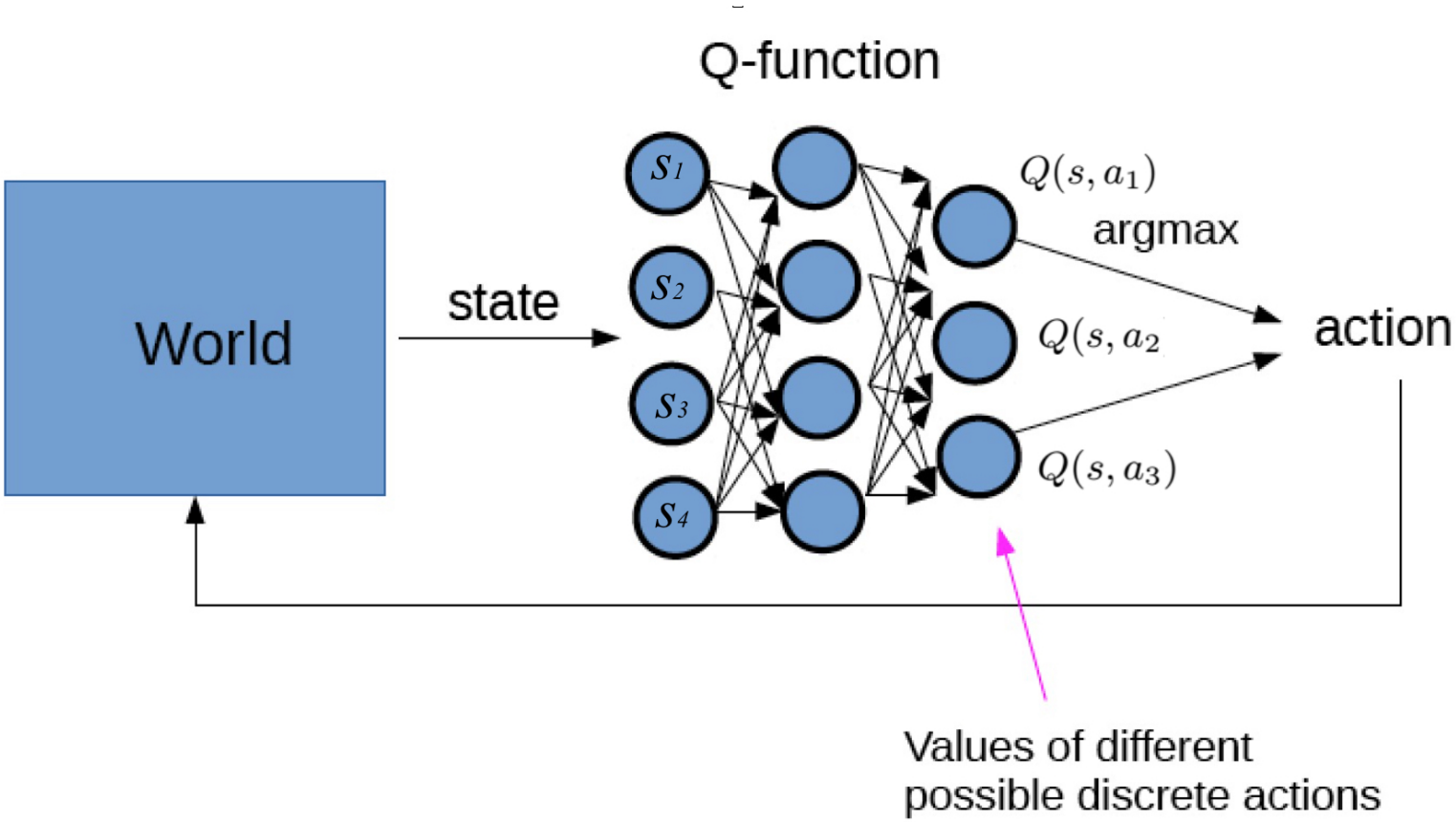
6,10: move one step using the behavioral policy $\mu(A_t|S_t)$
11: observe R_{t+1}, S_{t+1}
11: update $Q(S_t, A_t)$ (using greedy target policy)
11: note that A_{t+1} does **not** occur in update $\rightarrow$ off-policy
6,10: instead of ε-greedy behavioral policy a Boltzmann strategy can be used

Q-learning


Q-function
World
state
Q
action
state
argmax
action
Update rule

Deep Q-learning (DQN)

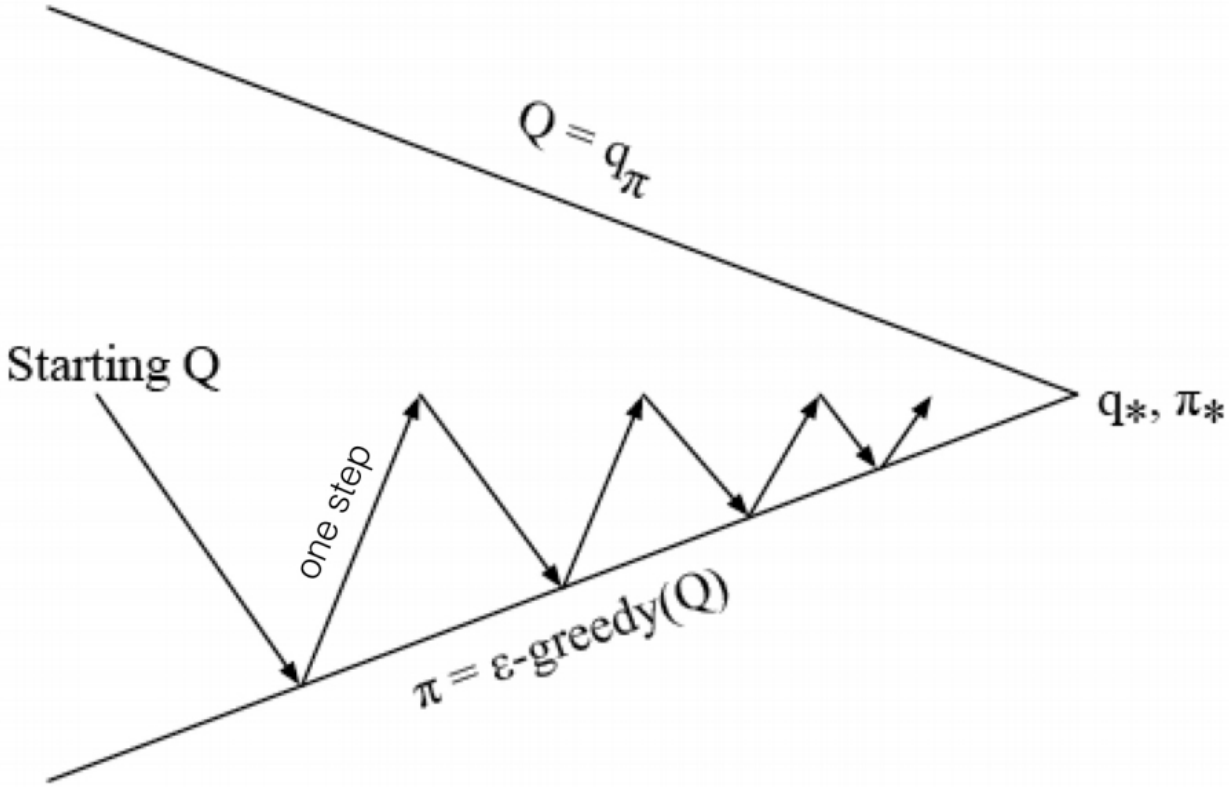
- Q-function is approximated using a NN



- input neuron: pixel value [0,255]
- one output neuron per action: Q-value

Deep Q-learning (DQN)

- Generalized Policy Iteration (GPI)



every step:
policy evaluation: gradient step for weight update of $Q_{\boldsymbol{\theta}}(s, a)$
policy improvement: ε-greedy policy improvement

DQN: Algorithm

Algorithm: Deep Q-Network (DQN) with Experience Replay

1: **Input:** the pixels and the game score, a preprocessing map ϕ
2: **Output:** Q action value function (from which we obtain policy and select action)
3: Initialize replay buffer $\mathcal{D}$ to capacity N
4: Initialize action-value function Q with random weights $\boldsymbol{\theta}$
5: Initialize target action-value function Q with random weights $\boldsymbol{\theta}^- = \boldsymbol{\theta}$
5: **for** $episode = 1, to\ M$ **do**
6: Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
7: **for** $t = 1, T$ **do**
8: ε-greedy: select $a_t = \begin{cases} \text{a random action}, & \text{with prob } \varepsilon \\ \operatorname{argmax}_a Q(\phi(s_t), a; \boldsymbol{\theta}), & \text{otherwise} \end{cases}$
9: Execute action a_i in emulator and observe reward r_t and image x_{t+1}
10: Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
11: Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
12: Sample random minibatch B of transitions $(\phi_j, a_j, r_j, \phi_j)$ from $\mathcal{D}$
13: Set $y_j = \begin{cases} r_j, & \text{episode terminates at } j+1 \\ r_j, +\gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \boldsymbol{\theta}^-) & \text{otherwise} \end{cases}$
14: Perform a gradient descent step on $(y_i - Q(\phi_j, a_j; \boldsymbol{\theta}))^2$ wrt to the network parameter $\boldsymbol{\theta}$
15: Every C steps reset $\hat{Q} = Q$, $\boldsymbol{\theta} = \boldsymbol{\theta}^-$
16: **end**
17: **end**

Comments: 11: experience: $e_t = (\phi_t, a_t, r_t, \phi_{t+1})$; 11: replay buffer $\mathcal{D} = \{e_1, \ldots, e_t\}$; 12-14: experience replay; 15: periodic update of target network, 8: policy improvement; 9-15: policy evaluation

DQN: States

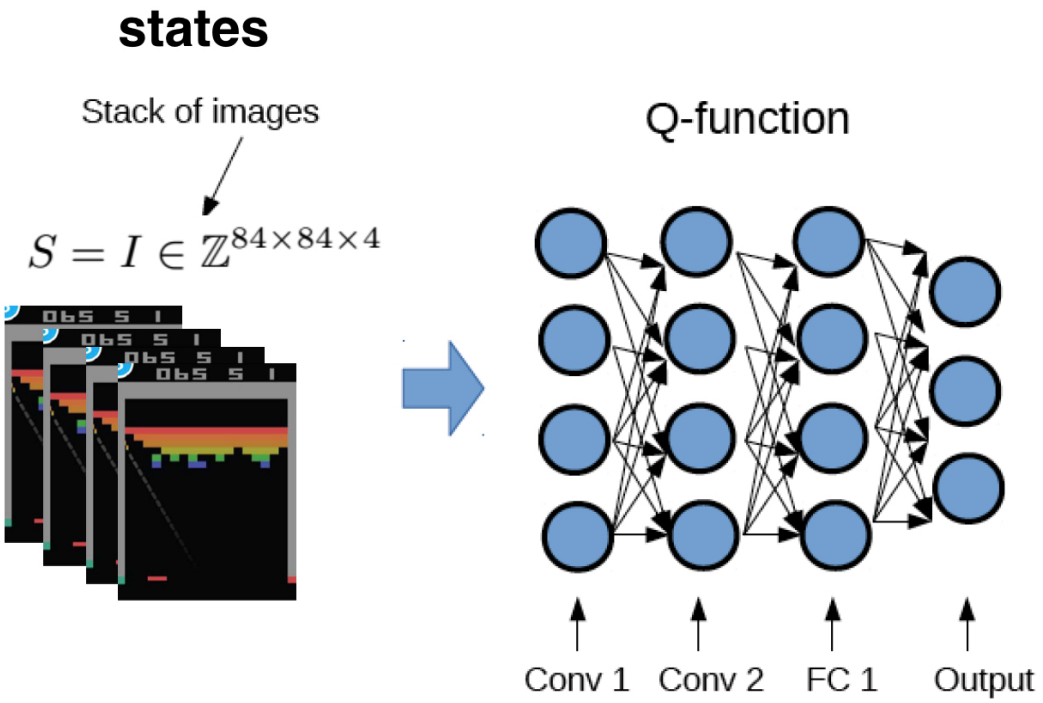
- input: single image (?)
- screen resolution: 210×160 pixels with 128 colors
- number of states: $210 \times 160 = 33600$, $|\mathcal{S}| = 128^{33600} \approx 10^{708002}$
- more states than number of atoms in the universe (10^{78}–10^{82})

Problems:

- **cannot** use tabular solution $\Rightarrow$ function approximation methods (NN)
- single image $\stackrel{?}{=}$ state: **violates** the Markov property (for example, no information about speed and direction of ball in Pong is available) $\Rightarrow$ POMDP (?)

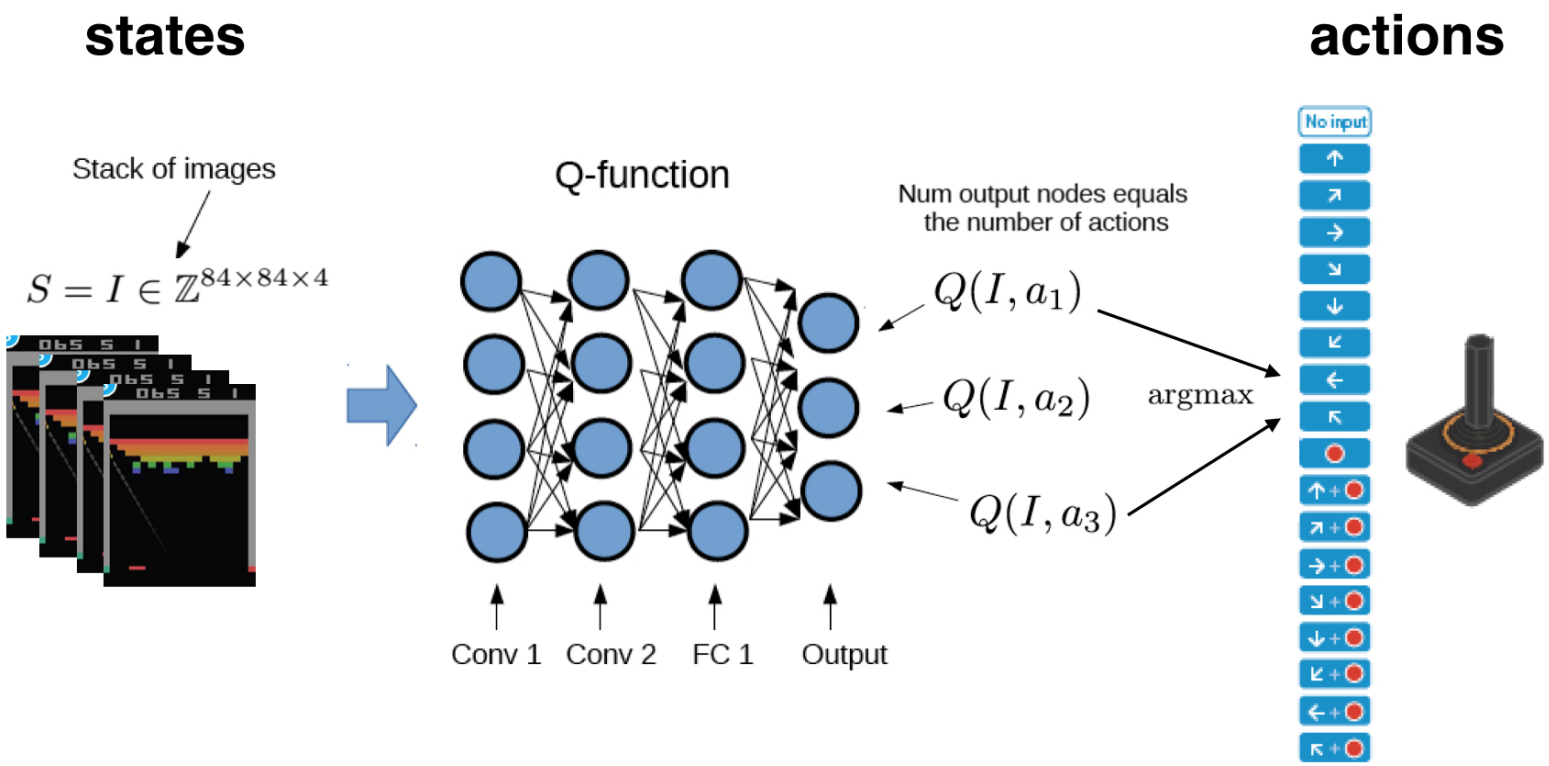
DQN: States

- **states $=$ sequence of images**
- use history of the k most recent images (DQN: $k = 4$)
- can deduct speed and direction of ball
- preprocessing operation ϕ: stacks of four most recent 4 single-color images 84×84
- makes it more Markovian
- same preprocessing for all games, no



DQN: Actions

- **discrete** number of actions
- Atari: $|\mathcal{A}| = 4 - 18$



DQN: Rewards

- change of game score from one time step to the next:

  +1 : increase
  -1 : decrease
  0 : otherwise

Q-function updates in Q-learning

- standard Q-learning update

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

  This equations adjust $Q(s,a)$ in the direction of the target $r + \gamma \max_{a'} Q(s',a')$.

- same update can be accomplished using an ANN and **supervised learning**

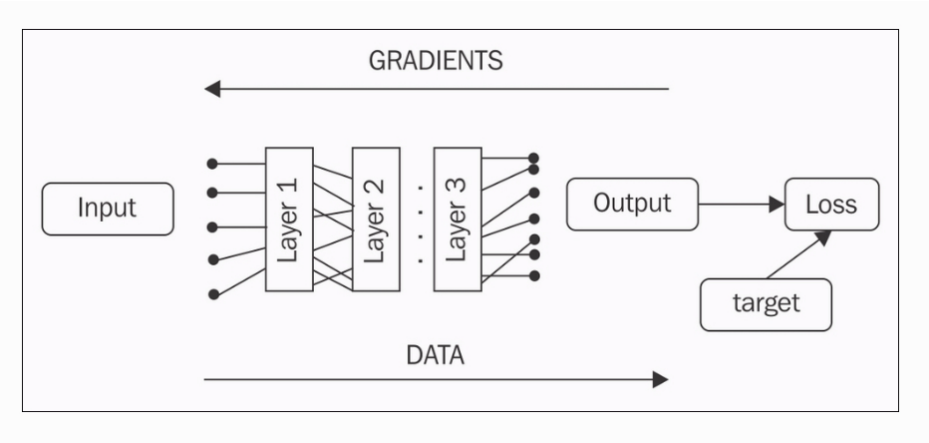
DQN: Q-function updates in DQN

- define **loss function**

$$\mathcal{L}(s, a, s'; \boldsymbol{\theta}) = \frac{1}{2}\Big(r + \gamma \max Q_{\boldsymbol{\theta}}(s', a') - Q_{\boldsymbol{\theta}}(s, a)\Big)^2$$

  where $Q_{\boldsymbol{\theta}}(s, a) \approx Q(s, a)$.
- **prediction**: $Q_{\boldsymbol{\theta}}(s, a)$
- **target**: $r + \gamma \max Q_{\boldsymbol{\theta}}(s', a')$
- optimize loss function using **semi-gradient** and backpropagation on $\boldsymbol{\theta}$

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(s, a, s'; \boldsymbol{\theta}) \approx -\Big(r + \gamma \max Q_{\boldsymbol{\theta}}(s', a') - Q_{\boldsymbol{\theta}}(s, a)\Big) \nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(s, a)$$



DQN: "Barebone"

Initialize $Q_{\boldsymbol{\theta}}(s, a)$ with random weights
Repeat (for each episode):
    Initialize *s*
    Repeat (for each step of the episode):
        Choose *a* from *s* using policy derived from Q (e.g. ε-greedy)
        Take action *a*, observe *r, s'*

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\boldsymbol{\theta}}(s, a, s'; \boldsymbol{\theta})$$
$$s \leftarrow s'$$

only difference
to standard q-learning

    Until *s* is terminal

where

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(s, a, s'; \boldsymbol{\theta}) \approx -\Big(r + \gamma \max Q_{\boldsymbol{\theta}}(s', a') - Q_{\boldsymbol{\theta}}(s, a)\Big) \nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(s, a)$$

- does this algorithm **converge**?
- generally not $\Rightarrow$ DQN introduces two tricks to achieve convergence: **experience replay** and **target network**

DQN: Trick 1 - Experience Replay

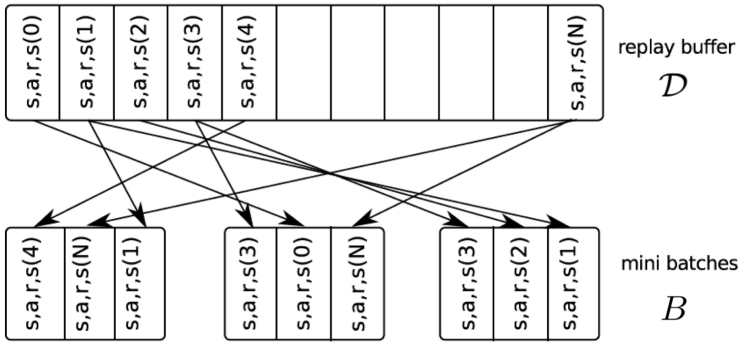
- ML/DL assumes **independent and identical distributed** (i.i.d.) training data

problem:

- samples $(S_t, A_t, R_{t+1}, S_{t+1})$ are **not independent**: samples from subsequent steps are highly correlated
- distribution of training data is **not identical**: data are generated of some policy that is changing during training (e.g. by ε-greedy policy)

solution:

- save transitions $(S_t, A_t, R_{t+1}, S_{t+1})$ in **replay buffer** of size N (e.g., $N = 10^6$) and *uniformly* sample **mini-batch** B (e.g., $|B| = 32$)
- use mini-batch B to train agent, i.e. update Q on experiences of **mini-batch** B



DQN: Experience Replay

Algorithm: Deep Q-Network (DQN) with Experience Replay

Input: the pixels and the game score
Output: Q action value function (from which we obtain policy and select action)
Initialize replay buffer $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights $\boldsymbol{\theta}$
Initialize target action-value function Q with random weights $\boldsymbol{\theta}^- = \boldsymbol{\theta}$
for $episode = 1$ to M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for $t = 1; T$ **do**
    ε-greedy: select $a_t = \begin{cases} \text{a random action}, & \text{with prob } \varepsilon \\ \operatorname{argmax}_a Q(\phi(s_t), a; \boldsymbol{\theta}), & \text{otherwise} \end{cases}$
    Execute action a_i 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 $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_j)$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j, & \text{episode terminates at } j+1 \\ r_j, +\gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \boldsymbol{\theta}^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_i - Q(\phi_j, a_j; \boldsymbol{\theta}))^2$ wrt to the network parameter $\boldsymbol{\theta}$
    Every C steps reset $\hat{Q} = Q$, $\boldsymbol{\theta} = \boldsymbol{\theta}^-$
  end
end

Lin, L.-J. (1992) *Self-Improving Reactive Agents based on Reinforcement Learning, Planning and Teaching*. Machine Learning, 8(3/4), 293-321

de Bruin, T et al (2015) *The importance of experience replay database composition in deep reinforcement learning*. In Deep RL workshop at NIPS

Zhang, S. and Sutton, R. S. (2017) *A deeper look at experience replay*. arXiv preprint arXiv:1712.01275

DQN: Q-update with Experience Replay (no Target Network)

sample batch B *uniformly* from $\mathcal{D}$:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}(B; \boldsymbol{\theta})$$

where
minibatch gradient descent:

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(B; \boldsymbol{\theta}) \approx -\frac{1}{|B|} \sum_{(s,a,s',r)} \big(target(s'; \boldsymbol{\theta}) - Q_{\boldsymbol{\theta}}(s,a)\big) \nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(s,a)$$

stochastic gradient descent over mini-batch:

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(B; \boldsymbol{\theta}) \approx -\big(target(s'; \boldsymbol{\theta}) - Q_{\boldsymbol{\theta}}(s,a)\big) \nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(s,a)$$

with

$$target(s'; \boldsymbol{\theta}) = r + \gamma \max_{a'} Q_{\boldsymbol{\theta}}(s', a')$$

DQN: Trick 2 - Target Network

problem:

- update of $Q_{\boldsymbol{\theta}}(s,a)$ is intended to make it closer to the target $r + \gamma \max_{a'} Q_{\boldsymbol{\theta}}(s',a')$, but target is itself a function of $Q_{\boldsymbol{\theta}}$. Update is done via $\boldsymbol{\theta}$:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}(B;\boldsymbol{\theta})$$

  with

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(B;\boldsymbol{\theta}) \approx -\big(target(s';\boldsymbol{\theta}) - Q_{\boldsymbol{\theta}}(s,a)\big)\nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(s,a)$$

- update also alters the value for $Q_{\boldsymbol{\theta}}(s',a')$ and other states nearby
- not truly supervised learning (in SL target constant)
- unstable ("chasing our own tail")

solution:

- key idea: "periods of supervised learning"
- to make training more stable add **target network**
- copy of original network for $Q_{\boldsymbol{\theta}}(s,a)$
- use it to represent $Q_{\boldsymbol{\theta}^-}(s',a')$
- update target network only periodically, every C iterations synchronize: $\boldsymbol{\theta}^- \leftarrow \boldsymbol{\theta}$

DQN: Q-update with Experience Replay and Target Network

sample batch B *uniformly* from $\mathcal{D}$

$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}(B; \boldsymbol{\theta}, \boldsymbol{\theta}^-)$

if mod(step, $C = 1000$) == 0

$\boldsymbol{\theta}^- \leftarrow \boldsymbol{\theta}$

where
minibatch gradient descent:

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(B; \boldsymbol{\theta}, \boldsymbol{\theta}^-) \approx -\frac{1}{|B|} \sum_{(s,a,s',r)} \big(target(s'; \boldsymbol{\theta}^-) - Q_{\boldsymbol{\theta}}(s,a)\big) \nabla_{\boldsymbol{\theta}} Q(s,a)$$

stochastic gradient descent:

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(B; \boldsymbol{\theta}, \boldsymbol{\theta}^-) \approx -\big(target(s'; \boldsymbol{\theta}^-) - Q_{\boldsymbol{\theta}}(s,a)\big) \nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(s,a)$$

with

$$target(s'; \boldsymbol{\theta}^-) = r + \gamma \max_{a'} Q_{\boldsymbol{\theta}^-}(s', a')$$

DQN: Q-update

Algorithm: Deep Q-Network (DQN) with Experience Replay

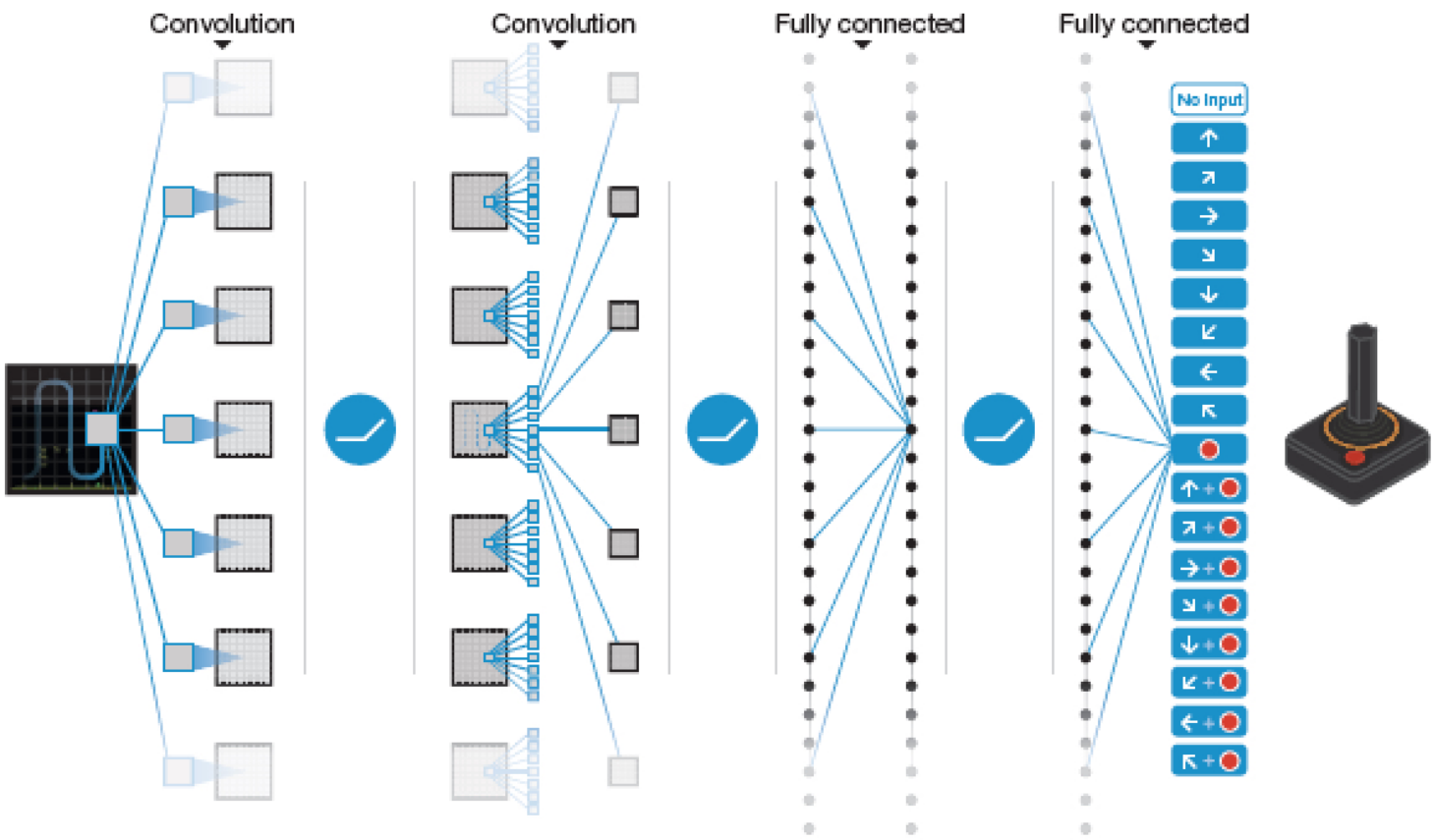
Input: the pixels and the game score
Output: Q action value function (from which we obtain policy and select action)
Initialize replay buffer $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights $\boldsymbol{\theta}$
Initialize target action-value function Q with random weights $\boldsymbol{\theta}^- = \boldsymbol{\theta}$
for $episode = 1\ to\ M$ **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for $t = 1; T$ **do**
    ε-greedy: select $a_t = \begin{cases} \text{a random action}, & \text{with prob } \varepsilon \\ \operatorname{argmax}_a Q(\phi(s_t), a; \boldsymbol{\theta}), & \text{otherwise} \end{cases}$
    Execute action a_i 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 $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_j)$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j, & \text{episode terminates at } j+1 \\ r_j, +\gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \boldsymbol{\theta}^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_i - Q(\phi_j, a_j; \boldsymbol{\theta}))^2$ wrt to the network parameter $\boldsymbol{\theta}$
    Every C steps reset $\hat{Q} = Q$, $\boldsymbol{\theta} = \boldsymbol{\theta}^-$
  end
end

DQN: Policy Improvement

Algorithm: Deep Q-Network (DQN) with Experience Replay

Input: the pixels and the game score
Output: Q action value function (from which we obtain policy and select action)
Initialize replay buffer $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights $\boldsymbol{\theta}$
Initialize target action-value function Q with random weights $\boldsymbol{\theta}^- = \boldsymbol{\theta}$
for $episode = 1$ to M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for $t = 1; T$ **do**
    ε-greedy: select $a_t = \begin{cases} \text{a random action}, & \text{with prob } \varepsilon \\ \operatorname{argmax}_a Q(\phi(s_t), a; \boldsymbol{\theta}), & \text{otherwise} \end{cases}$
    Execute action a_i 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 $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_j)$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j, & \text{episode terminates at } j+1 \\ r_j, +\gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \boldsymbol{\theta}^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_i - Q(\phi_j, a_j; \boldsymbol{\theta}))^2$ wrt to the network parameter $\boldsymbol{\theta}$
    Every C steps reset $\hat{Q} = Q$, $\boldsymbol{\theta} = \boldsymbol{\theta}^-$
  end
end

DQN: Network Architecture



DQN: Network Architecture

The input to the neural network consists of an 84 x 84 x 4 image produced by the preprocessing map, The first hidden layer convolves 32 filters of 8 x 8 with stride 4 with the input image and applies a rectifier nonlinearity. The second hidden layer convolves 64 filters of 4 x 4 with stride 2, again followed by a rectifier nonlinearity. This is followed by a third convolutional layer that convolves 64 filters of 3 x 3 with stride 1 followed by a rectifier. The final hidden layer is fully-connected and consists of 512 rectifier units. The output layer is a fully-connected linear layer with a single output for each valid action. The number of valid actions varied between 4 and 18 on the games we considered.

input: $84 \times 84 \times 4$
3 convolution layers
2 fully connected layers
output: one value for each action

DQN: Hyperparameters

Extended Data Table 1 | List of hyperparameters and their values

Hyperparameter	Value	Description
minibatch size	32	Number of training cases over which each stochastic gradient descent (SGD) update is computed.
replay memory size	1000000	SGD updates are sampled from this number of most recent frames.
agent history length	4	The number of most recent frames experienced by the agent that are given as input to the Q network.
target network update frequency	10000	The frequency (measured in the number of parameter updates) with which the target network is updated (this corresponds to the parameter *C* from Algorithm 1).
discount factor	0.99	Discount factor gamma used in the Q-learning update.
action repeat	4	Repeat each action selected by the agent this many times. Using a value of 4 results in the agent seeing only every 4th input frame.
update frequency	4	The number of actions selected by the agent between successive SGD updates. Using a value of 4 results in the agent selecting 4 actions between each pair of successive updates.
learning rate	0.00025	The learning rate used by RMSProp.
gradient momentum	0.95	Gradient momentum used by RMSProp.
squared gradient momentum	0.95	Squared gradient (denominator) momentum used by RMSProp.
min squared gradient	0.01	Constant added to the squared gradient in the denominator of the RMSProp update.
initial exploration	1	Initial value of ε in ε-greedy exploration.
final exploration	0.1	Final value of ε in ε-greedy exploration.
final exploration frame	1000000	The number of frames over which the initial value of ε is linearly annealed to its final value.
replay start size	50000	A uniform random policy is run for this number of frames before learning starts and the resulting experience is used to populate the replay memory.
no-op max	30	Maximum number of "do nothing" actions to be performed by the agent at the start of an episode.

The values of all the hyperparameters were selected by performing an informal search on the games Pong, Breakout, Seaquest, Space Invaders and Beam Rider. We did not perform a systematic grid search owing to the high computational cost, although it is conceivable that even better results could be obtained by systematically tuning the hyperparameter values.

DQN: Atari

Atari

DQN: Performance

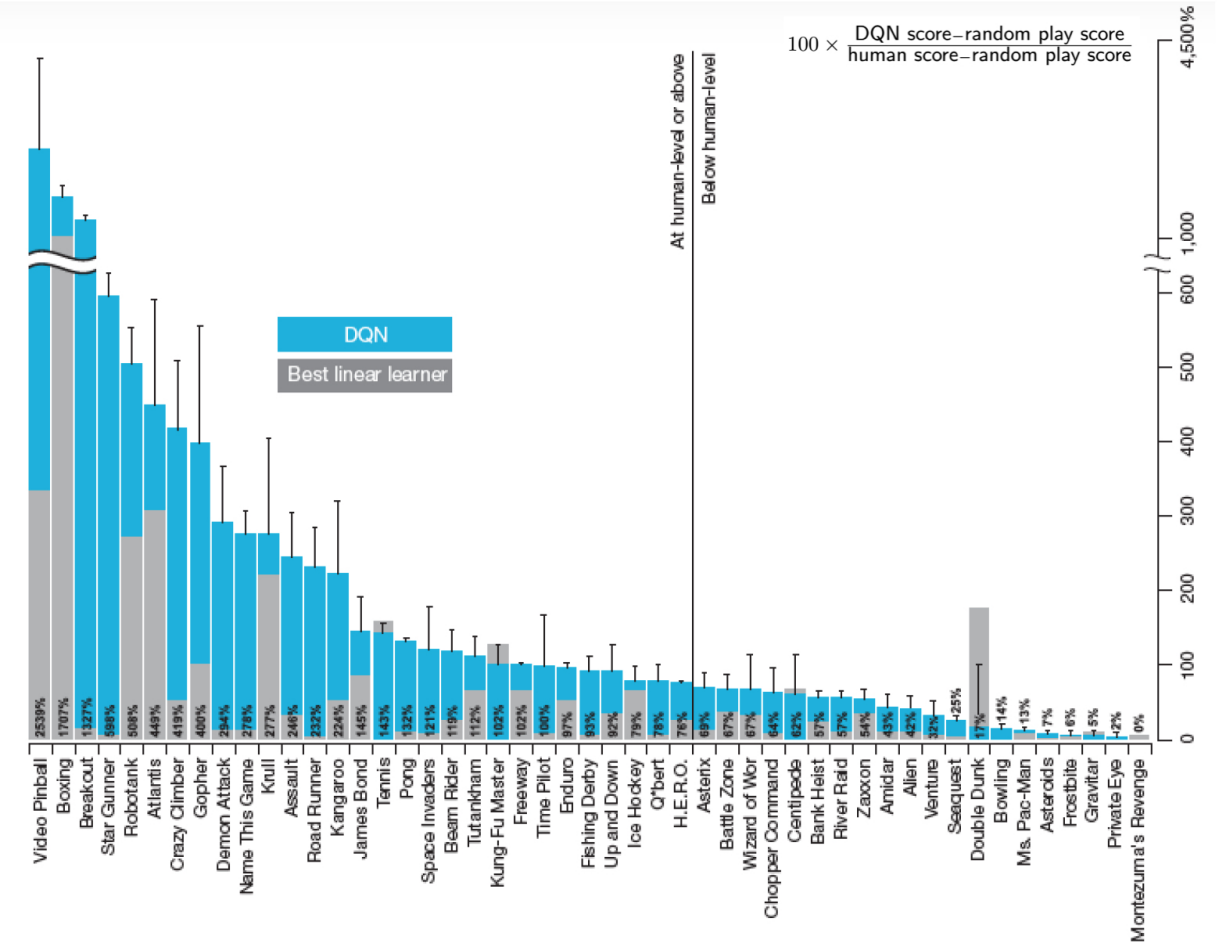
agent

- training: 50 million frames (38 days) for each game
- score: after learning, averaged over 30 sessions (up to 5 min each) on each game, beginning with random initial game state
- comparison to human player and linear learner (= best known RL agent with linear value function approximation)

human

- after 2h of practice, human played 20 episodes of each game for up to 5 min (without sound), no breaks allowed

DQN: Performance



- DQN better than linear learner on all but 6 games
- DQN better than human player on 29 of 46 games ($> 75\%$ human score), on 22 ($> 100\%$ human score)

DQN: Performance

AI breakthrough:

"Same learning system achieved these levels of play on widely varying games without relying on any game-specific modifications." (S+B)

DQN: Effect of Experience Replay and Target Network

Extended Data Table 3 | The effects of replay and separating the target Q-network

Game	With replay, with target Q	With replay, without target Q	Without replay, with target Q	Without replay, without target Q
Breakout	316.8	240.7	10.2	3.2
Enduro	1006.3	831.4	141.9	29.1
River Raid	7446.6	4102.8	2867.7	1453.0
Seaquest	2894.4	822.6	1003.0	275.8
Space Invaders	1088.9	826.3	373.2	302.0

DQN Extensions

- many improvements have been proposed to the vanilla-DQN algorithm
- paper: Hessel at al, 2017 *Rainbow: Combining Improvements in DRL*, seven most important improvements

Double Q-learning

problem:

- basic DQN (and standard Q-learning) is over-estimating Q
- $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
  $\leftarrow Q(s,a) + \alpha[r + \gamma Q(s', \mathrm{argmax}_{a'} Q(s',a')) - Q(s,a)]$
- maximization bias: max operator is used both for action choice and value propagation

solution: for standard Q-learning

- **Double Q-learning**: separate both calculations
- two estimates for Q: Q_1 and Q_2
- use one estimate, say Q_1, for maximizing action $A^* = \mathrm{argmax}_a Q_1(s,a)$
- use the other one, say Q_2 to estimate its value $Q_2(s, A^*) = Q_2(s, \mathrm{argmax}_a Q_1(s,a))$

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 $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using the policy ε-greedy in $Q_1 + Q_2$
    Take action A, observe R, S'
    With 0.5 probabilility:

$$Q_1(S, A) \leftarrow Q_1(S, A) + \alpha\Big(R + \gamma Q_2\big(S', \operatorname{argmax}_a Q_1(S', a)\big) - Q_1(S, A)\Big)$$

    else:

$$Q_2(S, A) \leftarrow Q_2(S, A) + \alpha\Big(R + \gamma Q_1\big(S', \operatorname{argmax}_a Q_2(S', a)\big) - Q_2(S, A)\Big)$$

    $S \leftarrow S'$
  until S is terminal

Double DQN

- DQN: make profit of the target network by propagating on target Q-network, selecting $\max$ on original Q-network
- DQN target:

$$target(s'; \boldsymbol{\theta}^-) = r + \gamma \max_{a'} Q_{\boldsymbol{\theta}^-}(s', a') = r + \gamma Q_{\boldsymbol{\theta}^-}(s', \operatorname*{argmax}_{a'} Q_{\boldsymbol{\theta}^-}(s', a'))$$

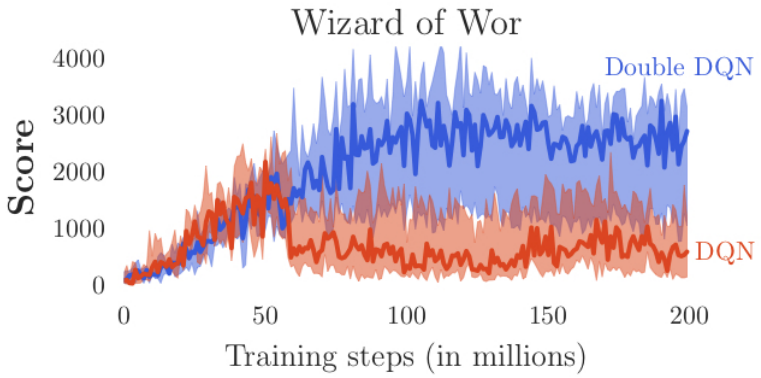
- **Double DQN** target:

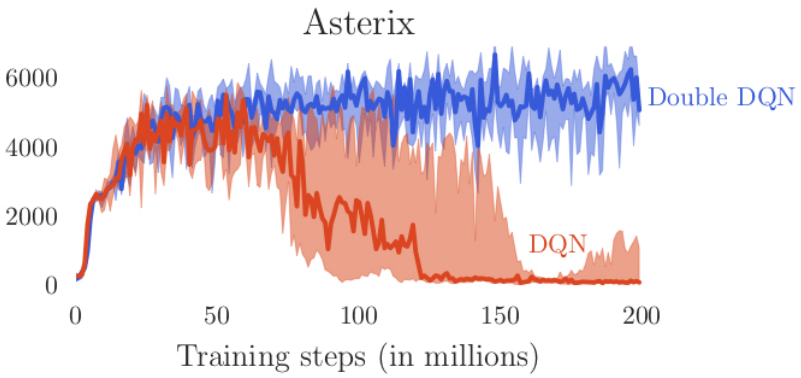
$$target(s'; \boldsymbol{\theta}, \boldsymbol{\theta}^-) = r + \gamma Q_{\boldsymbol{\theta}^-}(s', \operatorname*{argmax}_{a'} Q_{\boldsymbol{\theta}}(s', a'))$$

- minor changes to DQN
- converges twice faster

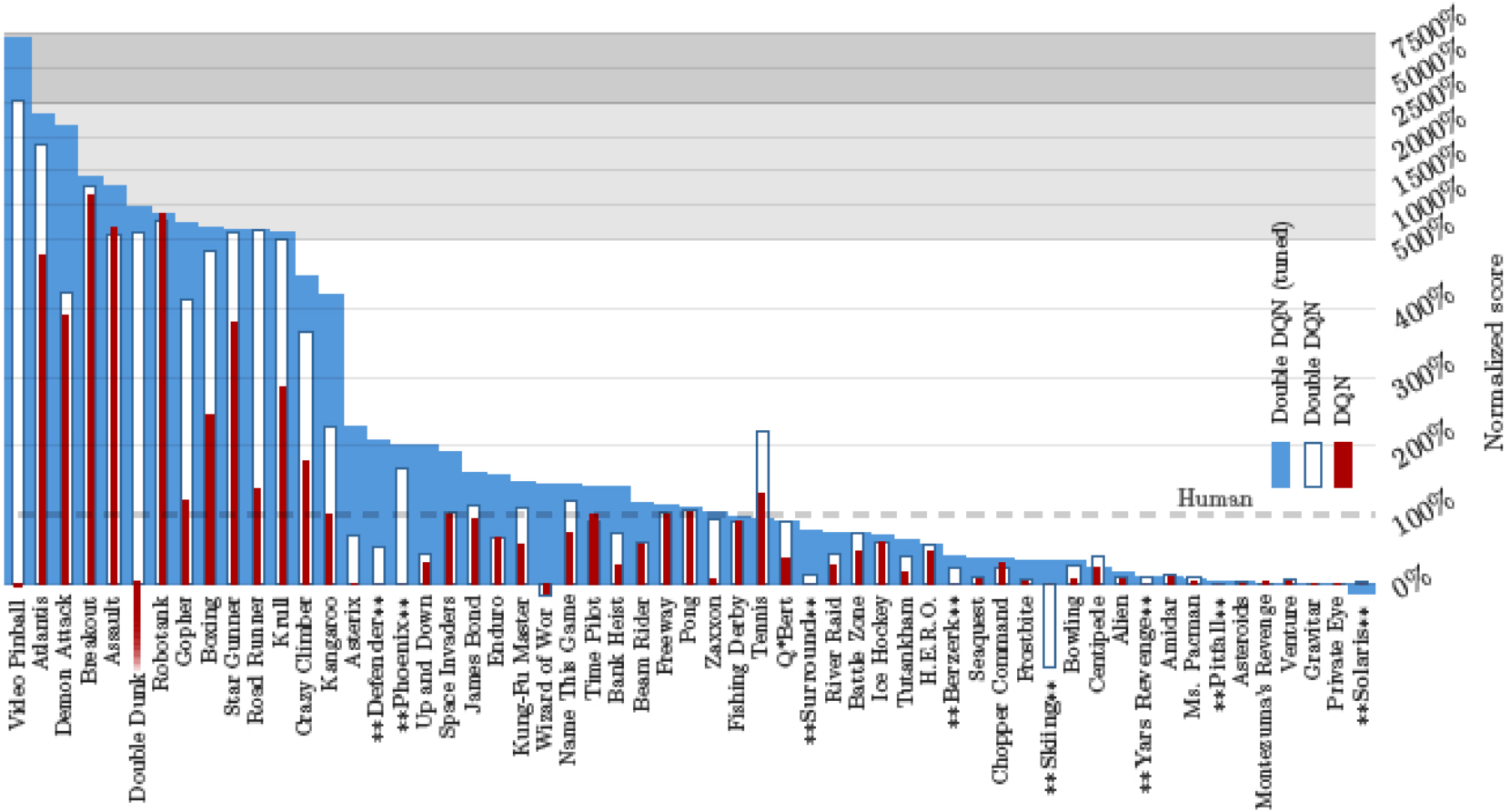
von Hasselt et al, 2015, *DRL with Double Q-Learning*, CoRL, abs/1509.06461

Double DQN


Wizard of Wor
Score
4000
3000
2000
1000
0
0
50
100
150
200
Training steps (in millions)
Double DQN
DQN

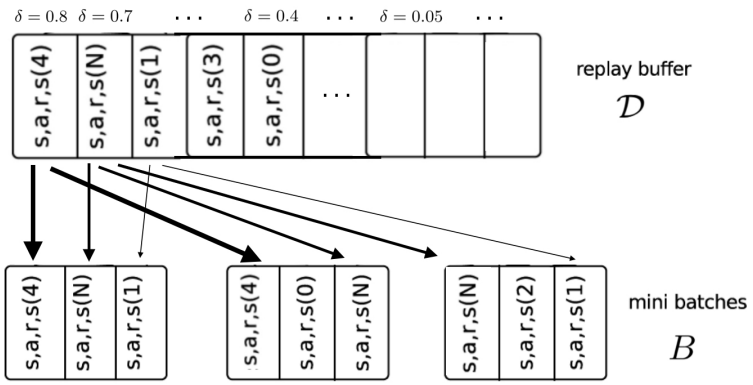

Asterix
6000
4000
2000
0
0
50
100
150
200
Training steps (in millions)
Double DQN
DQN

Double DQN: Atari



Prioritized Experience Replay (PER)

- Q-update in DQN: sample batch B *uniformly* from $\mathcal{D}$
- **Prioritized Experience Replay** (PER) = sample batch B from $\mathcal{D}$ more *intelligently*
- generate minibatch by drawing sample i from $\mathcal{D}$ with probability: $P(i) = p_i / \sum_k p_k$, where p_i denotes the priority of sample
- simplest case: $p_i = |\delta_i| + \epsilon$, δ_i = TD-error
- samples with a greater TD error improve the critic faster



Schaul et al, 2015, *Prioritized Experience Replay*, arXiv preprint arXiv:1511.05952

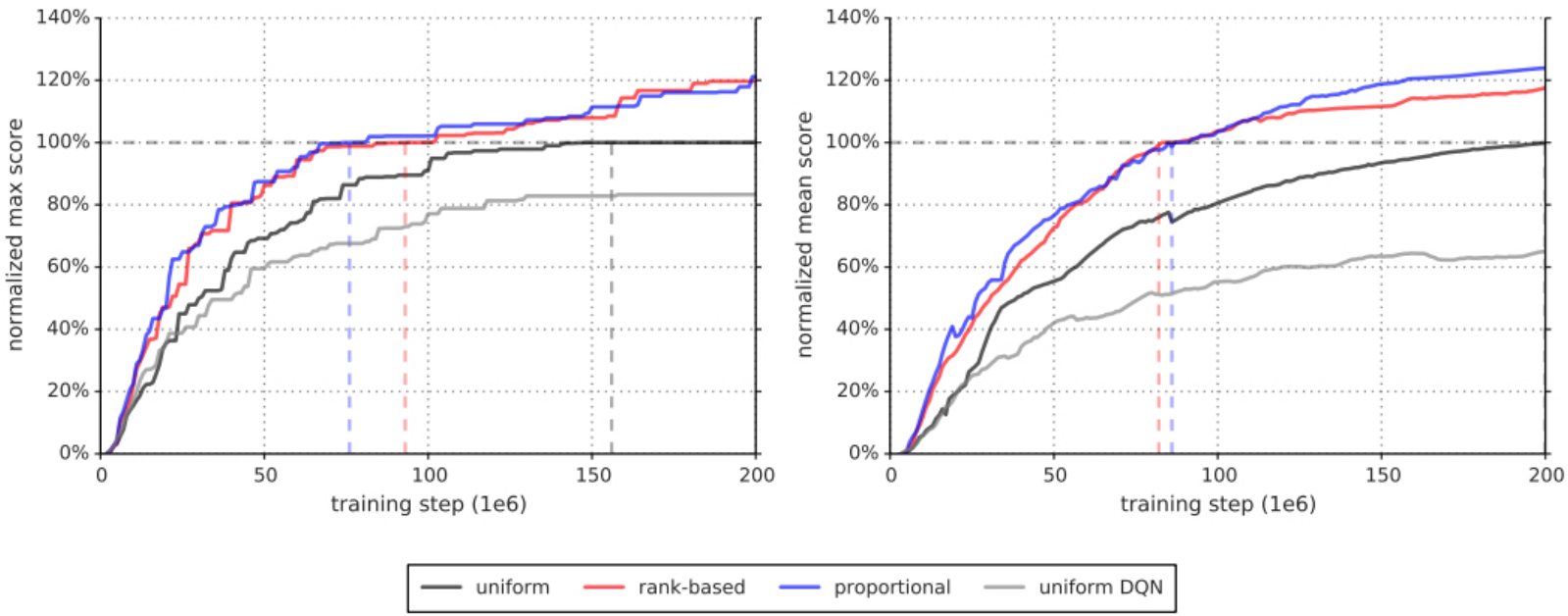
Prioritized Experience Replay (PER)

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 $\delta_j = R_j + \gamma_j Q_{\text{target}}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$
      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

Comments: since the distribution of updates is changing, this is off-policy $\rightarrow$ importance sampling

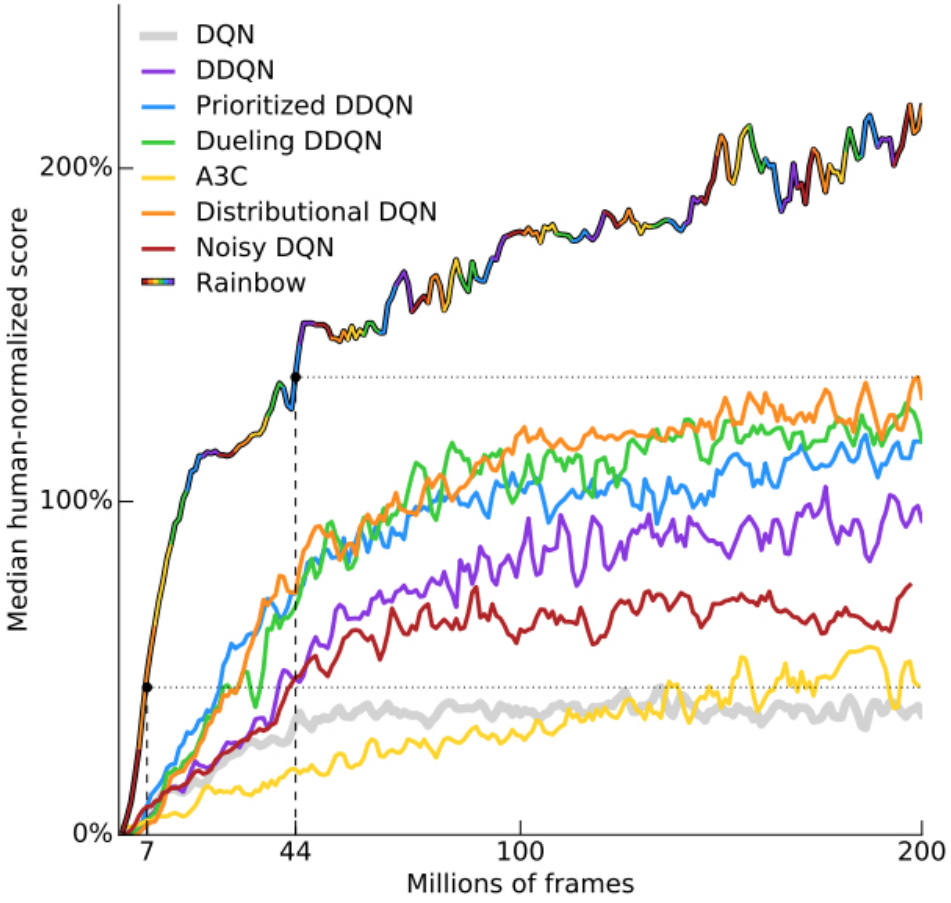
Prioritized Experience Replay (PER)



averaged results over 57 atari games

Combine all the improvements

- paper: Hessel at al, 2017 *Rainbow: Combining Improvements in DRL*, seven most important improvements



averaged results over 57 atari games