Deep Q-learning: an overview

A. Khachiyants D. Sharafyan

Faculty of Computer Science
Higher School of Economics

December 18

Table of contents

Main definitions

- s for state, a for action, r for reward
- π is policy — function that chooses action given the state (may be probablistic)
- Discounted return

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}, \quad r_t = r(s_t, a_t), \quad \gamma \in [0, 1]$$

  γ can be interpreted as patience (cake tomorrow is γ times better then now)
- $Q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid s_t = s, a_t = a]$ — expected discounted return given that game starts in position s, take player takes action a and after that follows policy π

Q-learning: motivation

$Q_\pi(s, a)$ is called action-value function.
Optimal action-value function is $Q_*(s, a) = \max_\pi Q_\pi(s, a)$
Optimal policy is $\pi^* = \arg\max_\pi Q_\pi(s, a)$

If we know Q_*, we know π^*. But how can we calculate Q_*?
Watkins [2] suggested approximating Q_* with the following rule:

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

Q-Learning: algorithm

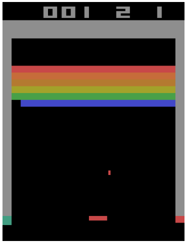
```
Initialize $Q(s, a)$ arbitrarily, and for all terminal states $s$ set $Q(s, \cdot) = 0$
**for** episode = 1 to M **do**
    Initialize $s$
    **while** $s$ is not terminal **do**
        Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $a$, observe $r$, $s'$
        $Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$
        $s \leftarrow s'$
```

State representation

Consider Breakout. How can we represent the state?

- Location of the paddle
- Location/direction of the ball
- Presence/absence of each individual brick

But this approach is really bad at generalizing. Then what? Let's use screen pixels.



Breakout

Value Iteration is impractical

In practice, value iteration is not that useful:

- Very limited states/actions
- Cannot generalize to unobserved states

Consider Breakout. Even after preprocessing:

- 88×88 pixels per frame
- Grayscale with 256 levels
- 4 frames per state

That is a lot of states ($\approx 10^{74600}$).

Then what should we do?

Create a function with some parameters that approximates true Q-values:
$Q(s, a; \theta) \approx Q^*(s, a)$
For example, using neural network.

Table of contents

Q-learning with NN: what could possibly go wrong?

The idea of approximation $Q(s, a)$ using neural networks is not new. But naive approach didn't work — learning was unstable or even impossible. Why?

Q-learning with NN: what could possibly go wrong?

The idea of approximation $Q(s, a)$ using neural networks is not new. But naive approach didn't work — learning was unstable or even impossible. Why?

- The correlations present in the sequence of observations
- Small updates of Q may significantly change the policy and therefore change the data distribution
- Correlations between the action-values and the target values

Experience Replay: main idea

Mnih et al. [3] suggested using experience replay.

- Main idea: remember all the transitions (s_t, a_t, r_t, s_{t+1}) we've seen so far in replay memory $\mathcal{D}$
- In DQN it is used to update weights with mini-batch stochastic gradient descend

Experience Replay: pros

- More efficient use of previous experience, by learning with it multiple times.
- Better convergence behavior when training value function approximator (partly because data becomes somewhat closer to being i.i.d.).

DQN: algorithm

Input: the pixels and the game score
Output: Q action value function (from which we obtain policy and select action)

```
Initialize replay memory $\mathcal{D}$, action-value function $Q$ with random weights $\theta$ and target
action-value function $\hat{Q}$ with weights $\theta^- = \theta$
**for** episode = 1 to M **do**
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\varphi_1 = \varphi(s_1)$
    **for** t = 1 to $T$ **do**
        Select $a_t$ using $\varepsilon$-greedy policy
        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 $\varphi_{t+1} = \varphi(s_{t+1})$
        Store transition $(\varphi_t, a_t, r_t, \varphi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch $D$ of transitions $(\varphi_j, a_j, r_j, \varphi_{j+1})$ from $\mathcal{D}$
        Set
            $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\varphi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
        Perform a GD step on $(y_j - Q(\varphi_j, a_j; \theta))^2$ w.r.t. the network parameter $\theta$
        Every $C$ steps update target network: $\hat{Q} = Q$, i.e., set $\theta^- = \theta$
```

How to train DQN

Well, previous slide lied a bit. In reality we compute loss function

$$\mathcal{L}(\theta) = \mathbb{E}_{(\varphi,a,r,\varphi')\sim D}\left[\frac{1}{2}(y - Q(\varphi, a; \theta))^2\right]$$

Next, the gradient w.r.t. θ is equal to

$$\nabla_\theta \mathcal{L}(\theta) = -\mathbb{E}_{(\varphi,a,r,\varphi')\sim D}\left[(y - Q(\varphi, a; \theta))\nabla_\theta Q(\varphi, a; \theta)\right]$$

And we do the usual gradient descend step:

$$\theta = \theta - \alpha \nabla_\theta \mathcal{L}(\theta)$$

Target Network

Notice that we used two networks: the usual one and the so called target network. What for?

Target Network

Notice that we used two networks: the usual one and the so called target network. What for?

To make training more stable. At every step of training, the Q-network's values shift, and if we are using a constantly shifting set of values to adjust our network values, then the value estimations can easily spiral out of control.

ER and target network: are they worth it?

The following table says it all.

Game	+ER, +TN	+ER, -TN	-ER, +TN	-ER, -TN
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

Reward range: yet another modification

We can modify DQN by clipping rewards to $[-1, 1]$. What for?

- This prevents Q-values from becoming too large
- Ensures gradients are well-conditioned

Table of contents

Overestimating...

When we update Q-values, we use $\max_{a'} \hat{Q}(\varphi_{j+1}, a'; \theta^{-})$. But that leads to overestimation. Why?

Overestimating...

When we update Q-values, we use $\max_{a'} \hat{Q}(\varphi_{j+1}, a'; \theta^-)$. But that leads to overestimation. Why?

Consider the following situation:

- For one particular state there is a set of actions, all of which have the same true Q value.
- But the estimate is noisy.
- max selects the action with the highest positive error.
- Subsequent propagation leads to positive bias.

...and how to fix it

van Hasselt et al. [4] proposed double Q-learning as the way of fixing overestimation:

- Let's take a look at DQN algorithm gradient descent step:

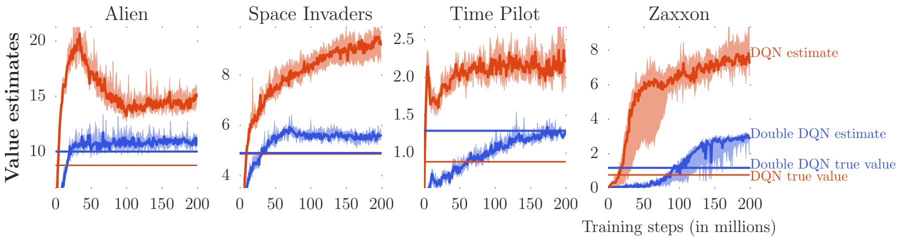
$$\theta_{t+1} = \theta_t + \alpha(y_t^{\text{DQN}} - Q(s_t, a_t; \theta_t))\nabla_{\theta_t} Q(s_t, a_t; \theta_t),$$

  where $y_t^{\text{DQN}} = r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta_t^-)$.
- Next, change y_t^{DQN} to

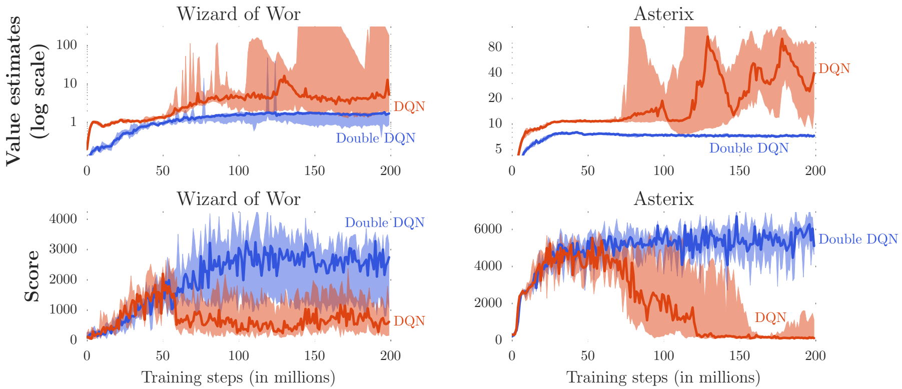
$$y_t^{\text{D-DQN}} = r_{t+1} + \gamma Q(s_{t+1}, \arg\max_a Q(s_{t+1}, a; \theta_t); \theta_t^-)$$

  In other words, we evaluate the greedy policy using the online network, but use the target network to estimate its value.

DQN really overestimates



DQN vs D-DQN: why overestimation breaks everything



DQN vs D-DQN: overview

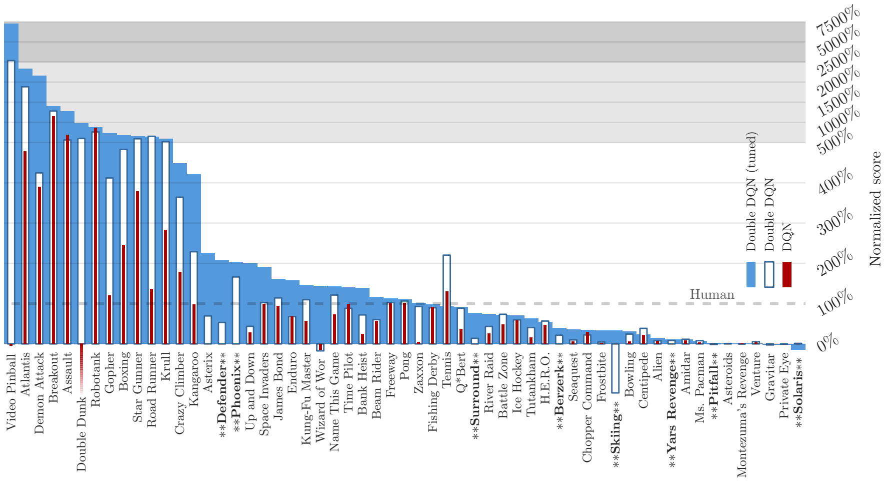


Table of contents

Back to experience replay

Earlier we discussed experience replay. Using it leads to design choices at two levels:

- which experiences to store
- which experiences to replay (and how to do so)

In original ER we ignore any differences in importance of transitions and suggest that they are uniform. But is that really alright?

Back to experience replay

Earlier we discussed experience replay. Using it leads to design choices at two levels:

- which experiences to store
- which experiences to replay (and how to do so)

In original ER we ignore any differences in importance of transitions and suggest that they are uniform. But is that really alright?

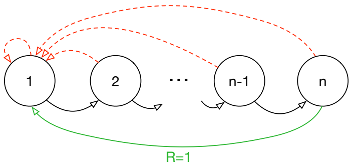
Schaul et al. [5] show that it isn't.

Blind Cliffwalk

Consider the following environment:

- There are two actions: a “right” and a “wrong” one.
- The episode is terminated whenever the agent takes the “wrong” action.
- Taking the “right” action progresses through a sequence of n states
- Reaching the end gives final reward of 1; reward is 0 elsewhere.

That environment exemplifies the challenge of exploration when rewards are rare.



Blind Cliffwalk

How about using oracle?

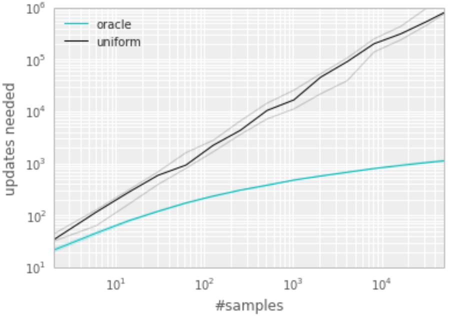
Let's compare two different agents:

- The first one replays transitions uniformly at random
- The second one invokes an oracle to prioritize transitions: this oracle greedily selects the transition that maximally reduces the global loss in its current state (in hindsight, after the parameter update).

Results

Picking the transitions in a good order can lead to exponential speed-ups over uniform choice.

Such an oracle is of course not realistic, yet the large gap motivates our search for a practical approach that improves on uniform random replay.



Median number of learning steps required to learn the value function

Prioritizing with TD-error

How can we prioritize transitions?

- Ideally, by the amount the RL agent can learn from a transition in its current state
- But we can't access it directly

Prioritizing with TD-error

How can we prioritize transitions?

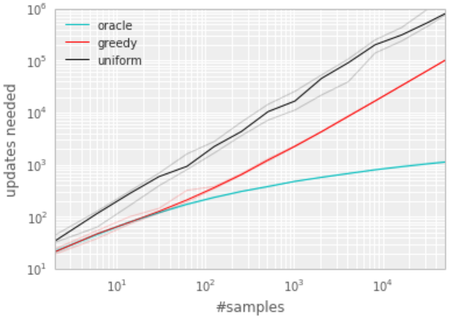
- Ideally, by the amount the RL agent can learn from a transition in its current state
- But we can't access it directly
- A reasonable proxy is the magnitude of a transition's TD error

$$\delta = r + \gamma \max_{a'} Q(s', a') - Q(s, a)$$

- This is particularly suitable for incremental, online RL algorithms, such as SARSA or Q-learning.
- This prioritizing is greedy

Comparison

This algorithm results in a substantial reduction in the effort required to solve the Blind Cliffwalk task.



TD-error prioritizing is troublesome

- TD errors are only updated for the transitions that are replayed.
- As a result, transitions that have a low TD error on first visit may not be replayed for a long time
- Sensitive to noise spikes
- Greedy prioritization focuses on a small subset of the experience
- As a result — prone to overfitting

Stochastic Prioritizing

To overcome this issues, let's create distribution that will keep order, but will give non-zero probabilities for every transition.

Stochastic Prioritizing

To overcome this issues, let's create distribution that will keep order, but will give non-zero probabilities for every transition.

For that we define the probability of sampling transition i as

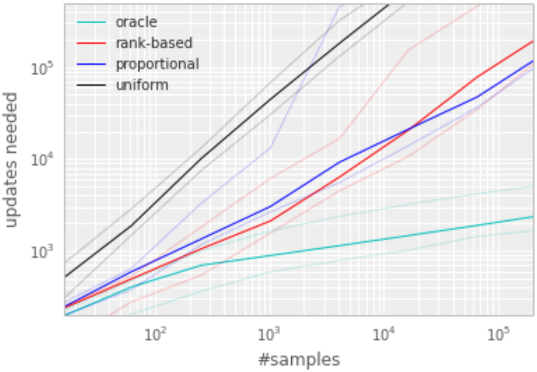
$$\mathrm{P}(i) = \frac{p_i^\alpha}{\sum_i p_i^\alpha}$$

- $p_i > 0$ is the priority of transition i
- α determines how much prioritization is used
- $\alpha = 0$ gives the uniform case

How to define p_i?

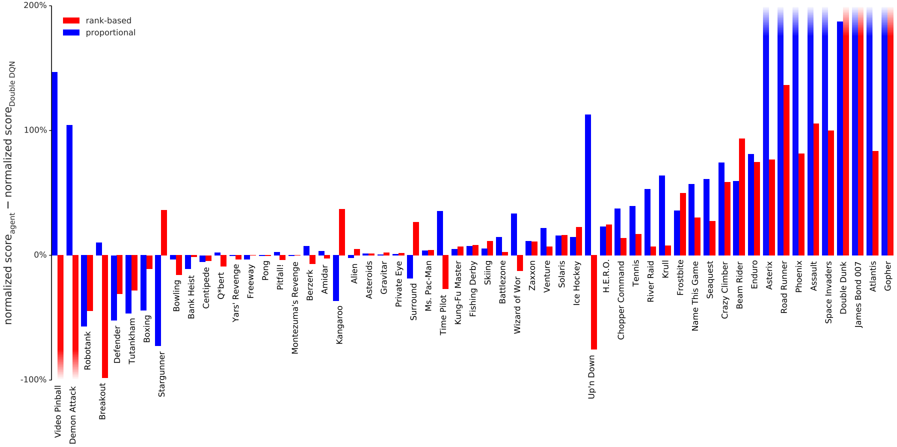
- Proportional: $p_i = |\delta_i| + \varepsilon$, where δ_i is TD-error for transition i and $\varepsilon > 0$
- Rank-based: $p_i = 1/\operatorname{rank}(i)$
- $\operatorname{rank}(i)$ is rank of transition i when the replay memory is sorted according to $|\delta_i|$
- Rank-based method is more robust to outliers

Results: better training time



Faint lines show max/min over 10 iterations

Results: better policies then D-DQN



D-DQN taken as 0%

Table of contents

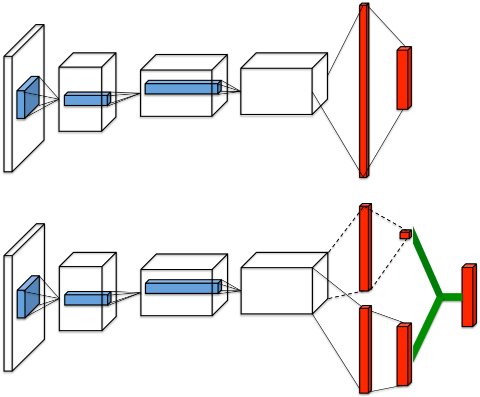
Dueling Network: main idea

Some definitions:

- $V_\pi(s) = \mathbb{E}_\pi[G_t \mid s_t = s]$ — state-value function
- $A_\pi(s, a) = Q_\pi(s, a) - V_\pi(s)$ — advantage function
- A_π measures the relative importance of action

Wang et al. [6] suggested the following idea: estimate $V(s)$ and $A(s, a)$ separately and then get $Q(s, a)$ as $V(s) + A(s, a)$.

Dueling Network: architecture



Original DQN on top, dueling DQN on bottom

Dueling Network: how to use it?

- Let's say that $Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + A(s, a; \theta, \alpha)$
- θ — parameters of convolutional layers
- α and β — parameters of two dense layers

Seems legit, right?

Dueling Network: how to use it?

- Let's say that $Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + A(s, a; \theta, \alpha)$
- θ — parameters of convolutional layers
- α and β — parameters of two dense layers

Seems legit, right?

Sorry, that won't work.

But why?

- $Q(s, a; \theta, \alpha, \beta)$ is only a parameterized estimate of the true Q-function.
- $V(s; \theta, \beta)$ is not a good estimator of the state-value function
- $A(s, a; \theta, \alpha)$ doesn't provide a reasonable estimate of the advantage function.
- Current equation is unidentifiable.

How to fix it?

- Force the advantage function estimator to have zero advantage at the chosen action:

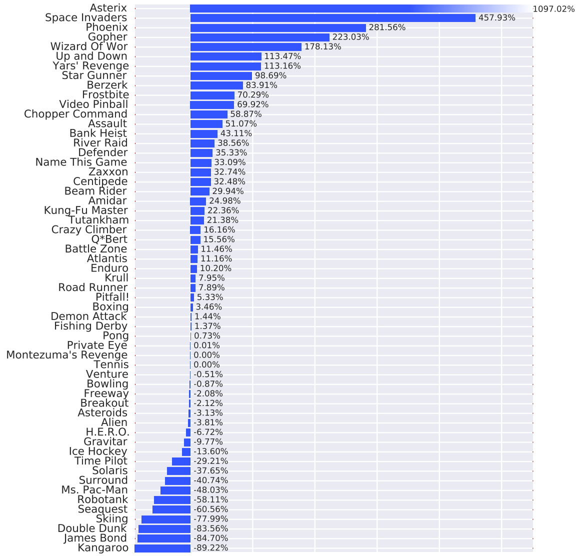
$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + \left(A(s, a; \theta, \alpha) - \max_{a'} A(s, a'; \theta, \alpha) \right)$$

- If we take $a^* = \arg\max_{a'} A(s, a'; \theta, \alpha)$, we get

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta)$$

- Hence, the stream $V(s; \theta, \beta)$ provides an estimate of the value function, while the other stream produces an estimate of the advantage function.

Dueling architecture vs Prioritized D-DQN



In most games dueling architecture performs better then prioritized D-DQN

Table of contents

Is that really all?

Of course not. There are lots of improvements, but we'll just mention four of them and give papers:

- Anschel et al. [7] proposed to reduce variability and instability by an average of previous Q-values estimates.
- Liang et al. [8] attempted to understand the success of DQN and reproduced results with shallow RL.
- Jeong and Lee [9] attempted to use Bayesian approach to Q-Learning (spoiler: how about updating weights using Kullback-Leibler divergence?)
- Hausknecht and Stone [10] suggested using RNN in order to deal with shortcomings of original approach.

Further Reading I

Yuxi Li
Deep Reinforcement Learning: An Overview
arXiv:1701.07274 [cs.LG]

Christopher Watkins
Learning From Delayed Rewards
Thesis (Ph. D.) – *King's College, Cambridge*, 1989

Volodymir Mnih et al.
Human-level control through deep reinforcement learning
Nature 518, 529–533 (26 February 2015)

Hado van Hasselt, Arthur Guez, David Silver
Deep Reinforcement Learning with Double Q-learning
arXiv:1509.06461 [cs.LG]

Further Reading II

Tom Schaul, John Quan, Ioannis Antonoglou, David Silver
Prioritized Experience Replay
arXiv:1511.05952 [cs.LG]

Ziyu Wang et al.
Dueling Network Architectures for Deep Reinforcement Learning
arXiv:1511.06581 [cs.LG]

Oron Anschel, Nir Baram, Nahum Shimkin
Averaged-DQN: Variance Reduction and Stabilization for Deep Reinforcement Learning
arXiv:1611.01929 [cs.AI]

Yitao Liang, Marlos C. Machado, Erik Talvitie, Michael Bowling
State of the Art Control of Atari Games Using Shallow Reinforcement Learning
arXiv:1512.01563 [cs.LG]

Further Reading III

Heejin Jeong, Daniel D. Lee
Bayesian Q-learning with Assumed Density Filtering
arXiv:1712.03333 [cs.LG]

Matthew Hausknecht, Peter Stone
Deep Recurrent Q-Learning for Partially Observable MDPs
arXiv:1507.06527 [cs.LG]