Q-learning

Dmitry Ligay

Higher School of Economics

Definition of Q-function

Weighted sum of rewards for fixed policy.

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}[\sum_{k=0}^{\infty} r_{t+k+1}\gamma^{k} | s_t = s, a_t = a] = \mathbb{E}_{\pi}[R_t | s_t = s, a_t = a]$$

Maybe we want to try to maximize this over all policies?

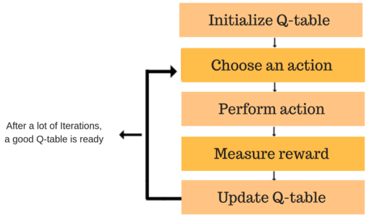
Definition of Q-function

$$Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a)$$

Bellman equation:

$$Q^*(s, a) = \mathbb{E}[r + \gamma \max_{a'} Q^*(s', a')|s, a]$$

Basic Q-learning



$$a = \begin{cases} \text{random action, with probability } \varepsilon \\ \arg\max_a Q_i(s,a), \text{otherwise} \end{cases}$$

Transition for Q table in state s after taking action a and receiving reward r:

$$Q_{i+1}(s,a) = Q_i(s,a) + \alpha(\mathbb{E}_{s'}[r + \gamma \max_{a'} Q_i(s',a')] - Q_i(s,a))$$

Problems with naive approach

- keeps all pairs (state, action) that we've met
- takes a lot of runs to find a good solution

How to solve these problems?

Currently $Q(s, a)$ ”exist independently”, but in reality there are dependencies between them.
So we can find a smaller function $Q(s, a, \theta) \approx Q^*(s, a)$.

Deep Q-learning

Define loss function at each step as follows:

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

The gradient of this function:

$$\nabla_{\theta_i} L_i(\theta_i) =$$

$$\mathbb{E}_{s,a\sim\rho,s'}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)\right) \nabla_{\theta_i} Q(s, a; \theta_i))\right]$$

Example: Atari games

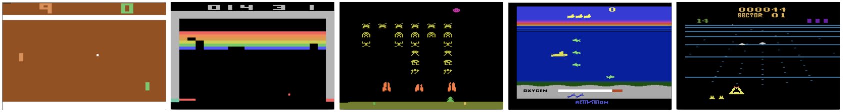


Figure 1: Screen shots from five Atari 2600 Games: (*Left-to-right*) Pong, Breakout, Space Invaders, Seaquest, Beam Rider

Input: image.
Output: action (small variety).

Theory to practice

We can approximate expected values with minibatches (basically SGD).
Experience replay: store agent's experiences in a dataset $\mathcal{D} = e_1, \ldots, e_N$, $e_t = (s_t, a_t, r_t, s_{t+1})$. We apply Q-learning updates (minibatch updates) to samples from $\mathcal{D}$.
Also our state is not an image, it's $(s_1, a_1, s_2 \ldots)$. So we need to convert this state to a fixed length vector (denoted as ϕ).

THE ALGORITHM

Algorithm 1 Deep Q-learning with Experience Replay

```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        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 $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
    end for
end for
```

Algorithm advantages

- we don’t forget about past states, as they can participate in weight updates after they had passed
- it’s innefficient to learn only from current state because adjacent states are strongly correlated
- smoother learning process because we learn not only from current optimal paths

Deeper implementation details

$\mathcal{D}$ is uniform and stores only last $N = 10^6$ states, ϕ stacks last 4 frames of history.
Image pipeline: gray-scale, down-sample to 110x84, square crop, 2D convolutions from AlexNet
Q neural network receives ϕ, and returns Q-value for all possible actions (for optimization).
For all games the neural network was the same, except for last FC linear layer (games have different number of possible actions).
All positive scores were replaced with 1, negative with -1 (games had different score scales).

Experiments

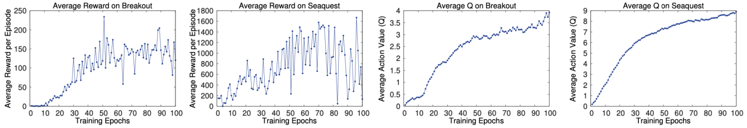


Figure 2: The two plots on the left show average reward per episode on Breakout and Seaquest respectively during training. The statistics were computed by running an ϵ-greedy policy with $\epsilon = 0.05$ for 10000 steps. The two plots on the right show the average maximum predicted action-value of a held out set of states on Breakout and Seaquest respectively. One epoch corresponds to 50000 minibatch weight updates or roughly 30 minutes of training time.

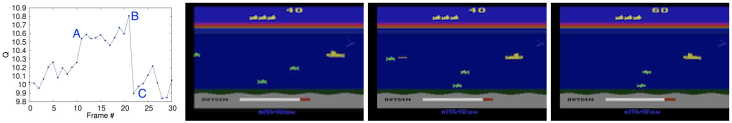


Figure 3: The leftmost plot shows the predicted value function for a 30 frame segment of the game Seaquest. The three screenshots correspond to the frames labeled by A, B, and C respectively.

Experiments

	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
HNeat Best [8]	3616	52	106	19	1800	920	**1720**
HNeat Pixel [8]	1332	4	91	-16	1325	800	1145
DQN Best	**5184**	**225**	**661**	**21**	**4500**	**1740**	1075

Table 1: The upper table compares average total reward for various learning methods by running an ϵ-greedy policy with $\epsilon = 0.05$ for a fixed number of steps. The lower table reports results of the single best performing episode for HNeat and DQN. HNeat produces deterministic policies that always get the same score while DQN used an ϵ-greedy policy with $\epsilon = 0.05$.