Prioritized Experience Replay

PER

Google DeepMind 2016 ICLR

https://arxiv.org/abs/1511.05952

20180521 구태훈

Why PER?

- DQN으로 Atari 돌려보니 49개 중 41개에서 Replay Buffer에 비해 높은 점수 기록 (Uniform sampling)
- Rainbow 중 하나
- 간단하게 적용 가능

PER

- The main idea is that we prefer transitions that does not fit well to our current estimate of the Q function
- using a clever prioritization scheme to label the experiences in replay memory, learning can be carried out much faster and more effectively

PER

- The magnitude of the TD-error indicates how unexpected a certain transition was
- The TD-error can be a poor estimate about the amount an agent can learn from a transition when rewards are noisy

Problems of greedy-selecting

- High-error transitions are replayed too frequently
- Low-error transitions are almost entirely ignored
- Expensive to update entire replay memory, so errors are only updated for transitions that are replayed
- Lack of diversity leads to over-fitting

A stochastic sampling method

- To find a balance between greedy prioritization and random sampling

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

P is the probability of sampling transition i, $p_i > 0$ is the priority of transition i, and the exponent α determines how much prioritization is used, with α=0 the uniform case

* 참고 Softmax

$$P(y = j \mid \mathbf{x}) = \frac{e^{\mathbf{x}^\top \mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\top \mathbf{w}_k}}$$

Two variants

Variant 1: proportional prioritization

$$p_i = |\delta_i| + \epsilon$$

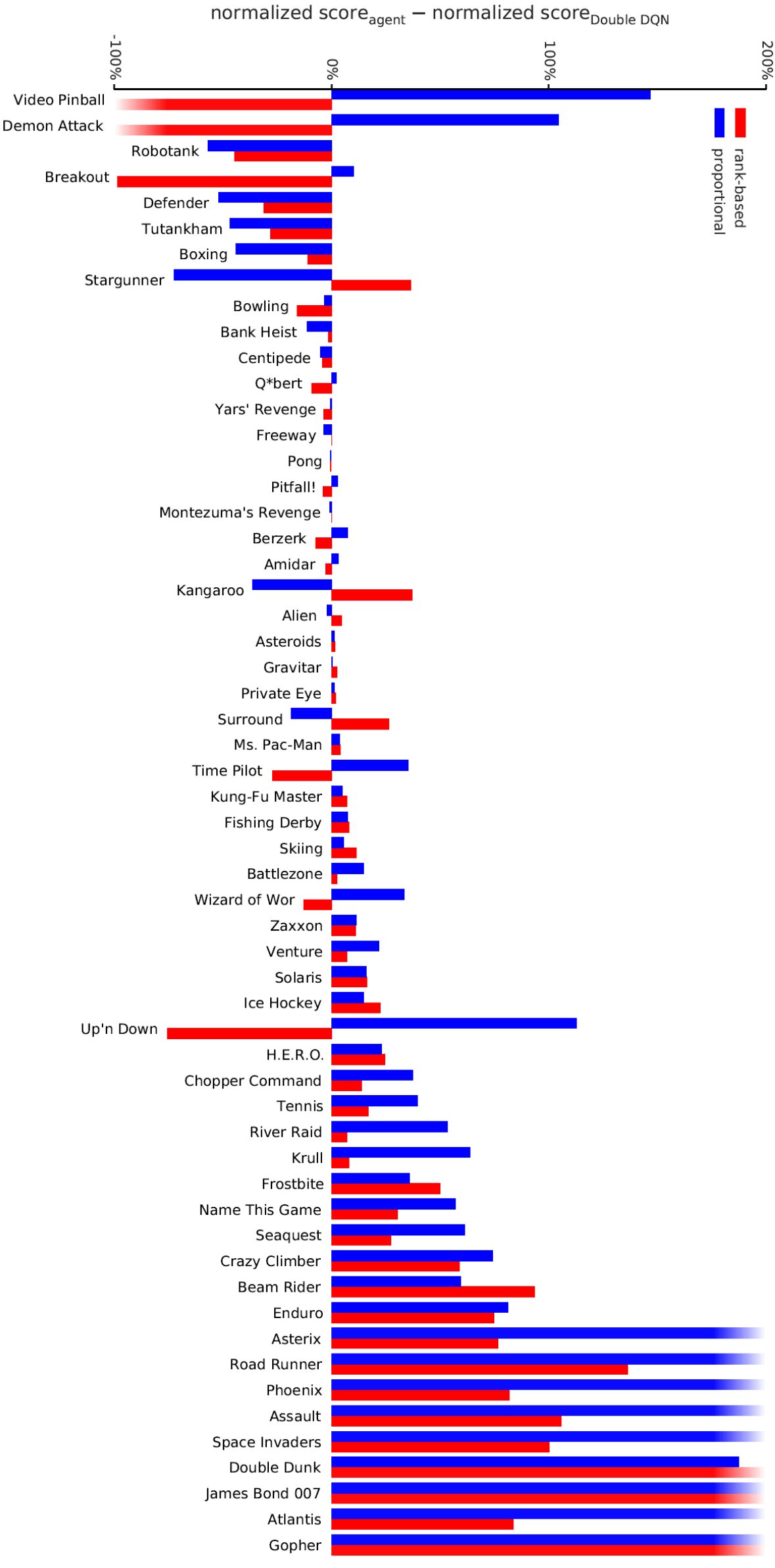
ϵ is a small positive constant that prevents the edge-case of transitions not being revisited once their error is zero. δ is the TD-error

Variant 2: rank-based prioritization

$$p_i = \frac{1}{\text{rank}(i)}$$

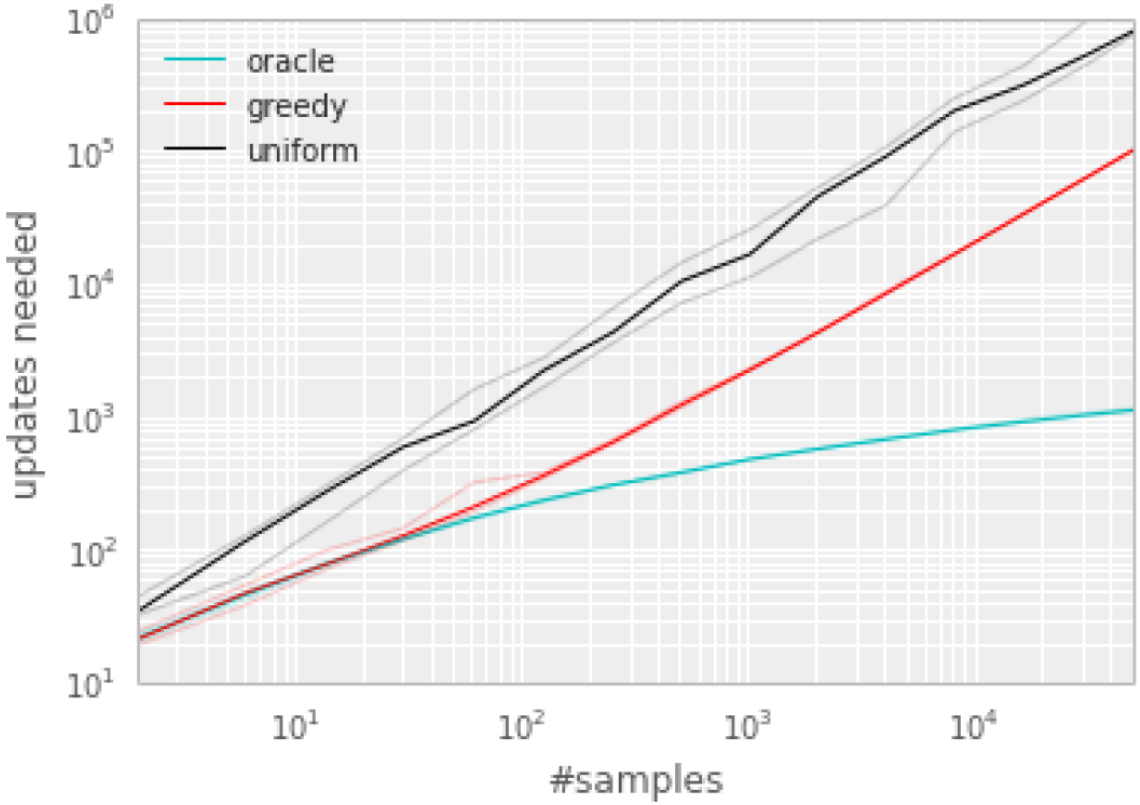
rank(i) is the rank of transition i when the replay memory is sorted according to δi

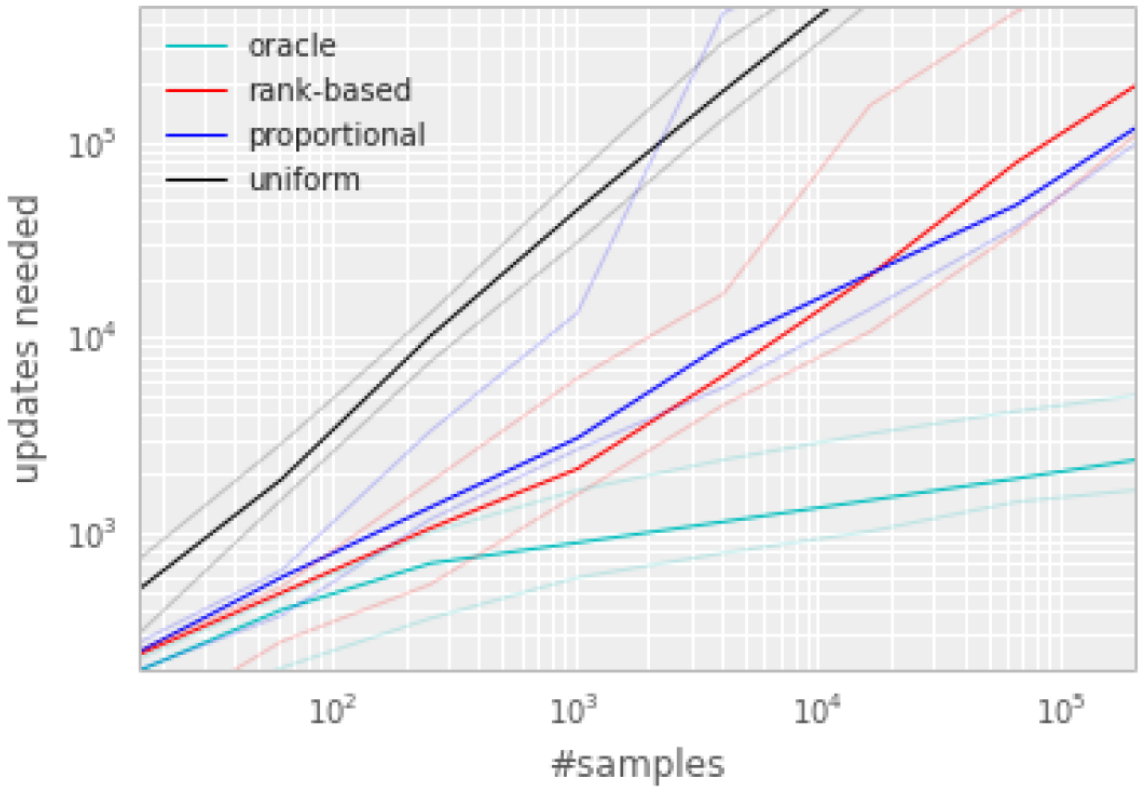
latter is likely to be more robust, as it is insensitive to outliers


normalized score$_{agent}$ − normalized score$_{Double\ DQN}$
-100%
0%
100%
200%
proportional
rank-based
Video Pinball
Demon Attack
Robotank
Breakout
Defender
Tutankham
Boxing
Stargunner
Bowling
Bank Heist
Centipede
Q*bert
Yars' Revenge
Freeway
Pong
Pitfall!
Montezuma's Revenge
Berzerk
Amidar
Kangaroo
Alien
Asteroids
Gravitar
Private Eye
Surround
Ms. Pac-Man
Time Pilot
Kung-Fu Master
Fishing Derby
Skiing
Battlezone
Wizard of Wor
Zaxxon
Venture
Solaris
Ice Hockey
Up'n Down
H.E.R.O.
Chopper Command
Tennis
River Raid
Krull
Frostbite
Name This Game
Seaquest
Crazy Climber
Beam Rider
Enduro
Asterix
Road Runner
Phoenix
Assault
Space Invaders
Double Dunk
James Bond 007
Atlantis
Gopher

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


oracle
greedy
uniform
updates needed
#samples
10^1
10^2
10^3
10^4
10^5
10^6


oracle
rank-based
proportional
uniform
updates needed
#samples
10^2
10^3
10^4
10^5

Problem of bias

- The estimation of the expected value with stochastic updates relies on those updates corresponding to the same distribution as its expectation.
- Prioritized replay introduces bias because it changes this distribution in an uncontrolled fashion, and therefore changes the solution that the estimates will converge to (even if the policy and state distribution are fixed)

IS (Importance-Sampling)

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)} \right)^{\beta}$$

that fully compensates for the non-uniform probabilities P(i) $\text{if } \beta = 1.$ These weights can be folded into the Q-learning update by using $w_i \delta_i \text{ instead of } \delta_i$

(this is thus weighted IS, not ordinary IS, see e.g. Mahmood et al., 2014).

For stability reasons, we always normalize weights by 1= $\max_i w_i$ so that they only scale the update downwards.

IS (Importance-Sampling)

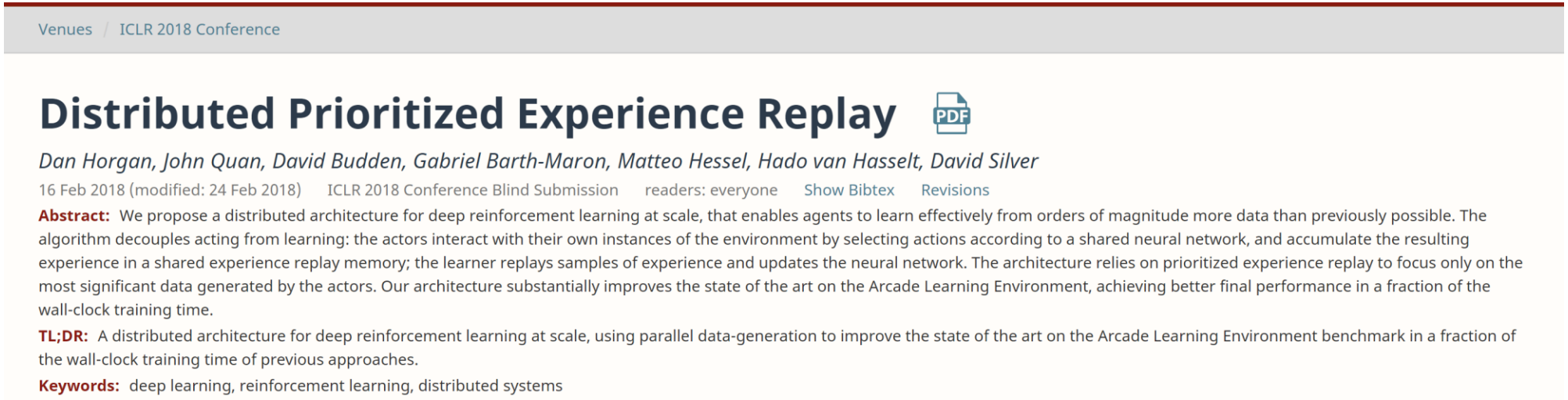
- IS is annealed from β_0 to 1, which means its affect is felt more strongly at the end of the stochastic process; this is because the unbiased nature of the updates in RL is most important near convergence
- IS also reduces the gradient magnitudes which is good for optimization; allows the algorithm to follow the curvature of highly non-linear optimization landscapes because the Taylor expansion (gradient descent) is constantly re-approximated

```
def sample(self, num_samples):
    indices, probs = self.errors.sample(num_samples)
    beta = float(self.beta)
    importance_weights = np.power(probs * self.size, -beta)
    importance_weights /= np.power(self.errors.min() / self.errors.sum() * self.size, -beta)
    samples = []
    for i, weight in zip(indices, importance_weights):
        sample = self.transitions[i].copy()
        sample['weight'] = weight
        sample['id'] = i
        samples.append(sample)
    return samples


 def update_weights(self, samples, new_weights):
        for sample, weight in zip(samples, new_weights):
            self.errors.set_value(sample['id'], self._process_weight(weight))

 def _process_weight(self, weight):
        self._max_weight_arg = max(self._max_weight_arg, weight)
        return (weight + self.epsilon) ** self.alpha
```

참고



https://openreview.net/forum?id=H1Dy---0Z

Rainbow: Combining Improvements in Deep Reinforcement Learning

https://arxiv.org/abs/1710.02298