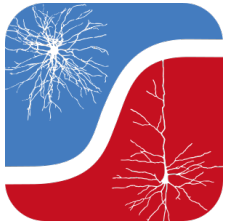
# A Tutorial on Deep-Q-Learning

Robert Lange

January 2020

RobertTLange

@RobertTLange



@SprekelerLab


@ECNBerlin


@SCIOI

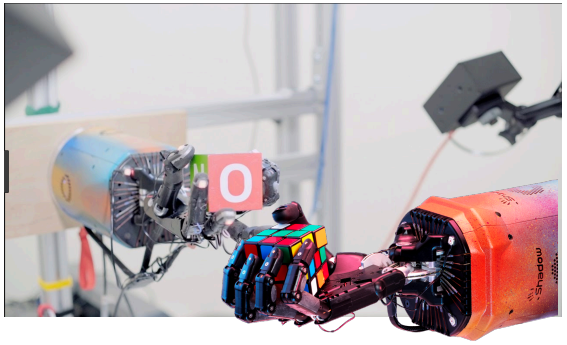
# The Success Story.

DM - Atari DQN
(2013, 2015)


DM - AlphaGo
(2016, 2017)

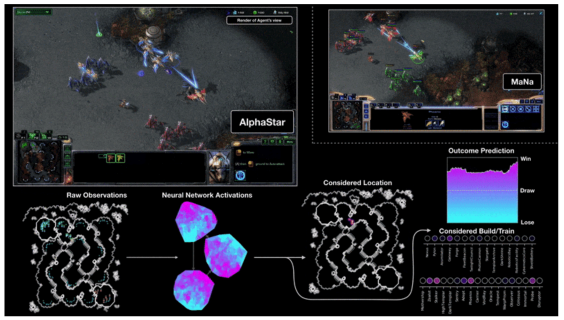

DM - AlphaZero
(2018)


OpenAI - Dexterity
(2018, 2019)

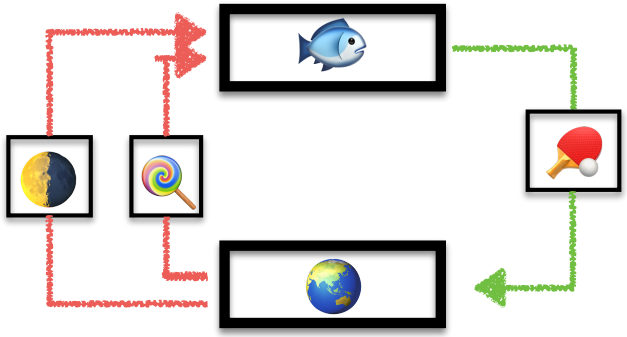


OpenAI - Five
(Dota 2 - 2019)

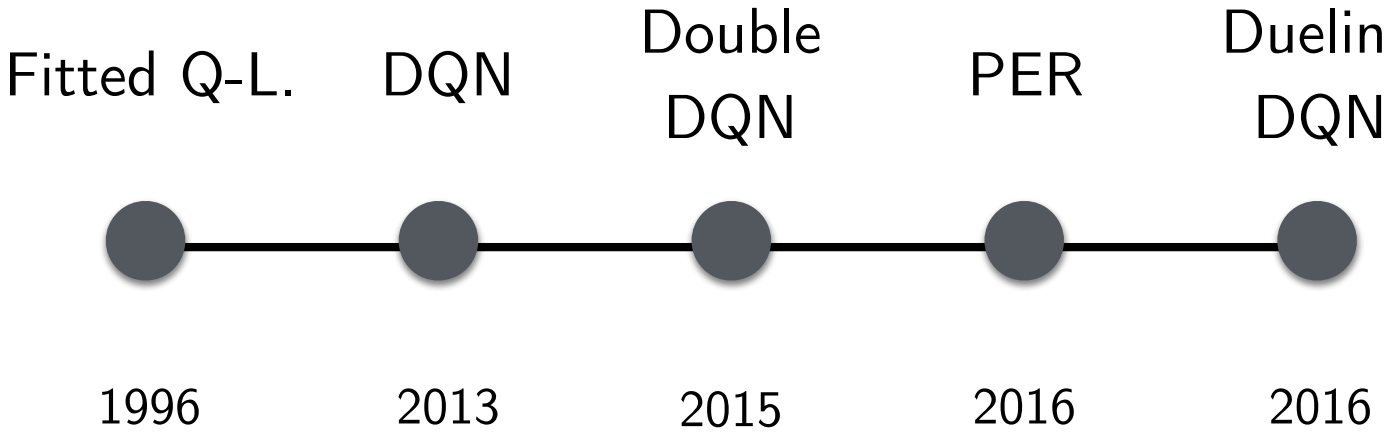

DM - AlphaStar
(StarCraft II -2019)

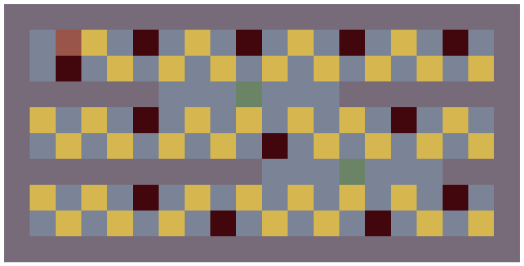


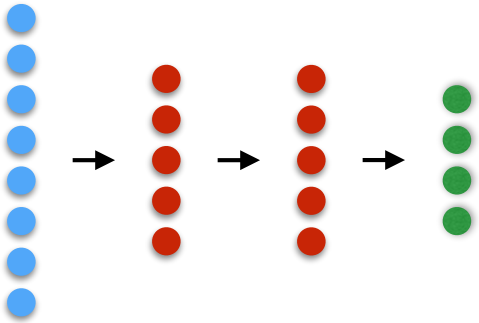
# A Roadmap for Today.

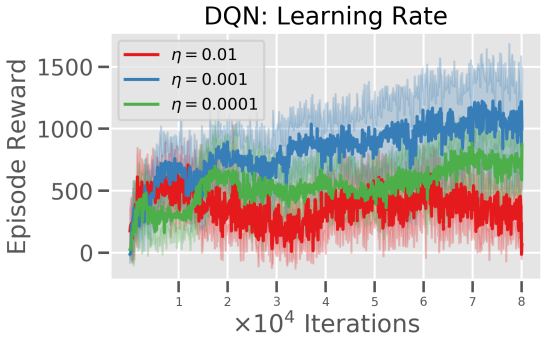


- MDP Formalism
- The RL Problem
- Value-Based RL

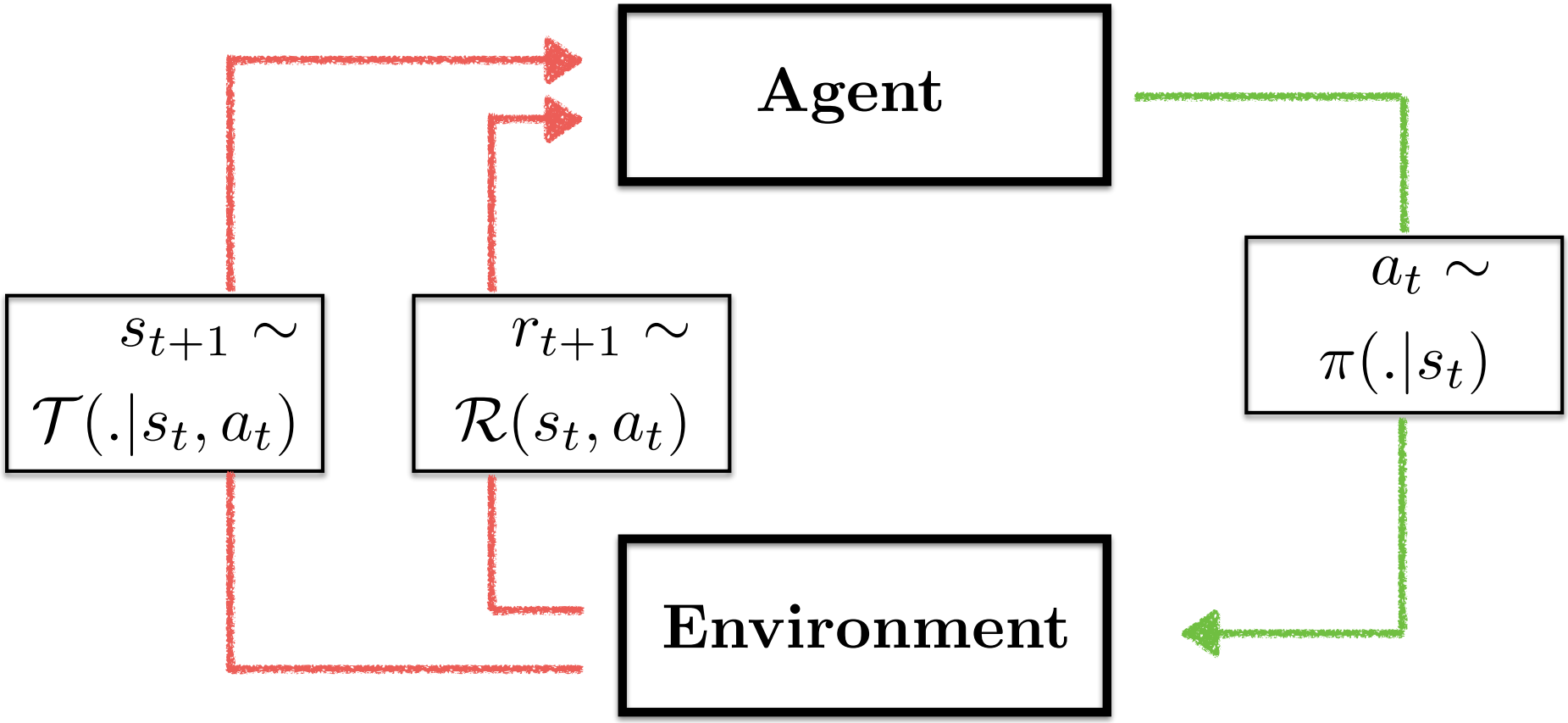






# The Action Perception Loop of RL.




- MDP Formalism:
  $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma)$
- Deterministic Policy:
  $\pi(s) : \mathcal{S} \rightarrow \mathcal{A}$

# The Reinforcement Learning Problem.

- Value function: $V^{\pi}(s) = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t = s, \pi]$

- The RL problem: $V^{\star}(s) = \max_{\pi \in \Pi} V^{\pi}(s)$

R. Bellman

- Action-Value function:

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

$$= \sum_{s' \in \mathcal{S}} T(s,a,s')[R(s,a,s') + \gamma Q^{\pi}(s', a = \pi(s'))]$$

# Temporal Difference Learning in MDPs.

$$Q^{\pi}(s,a) = \sum_{s' \in \mathcal{S}} T(s,\cancel{a},s')[R(s,\cancel{a},s') + \gamma Q^{\pi}(s', a = \pi(s'))]$$

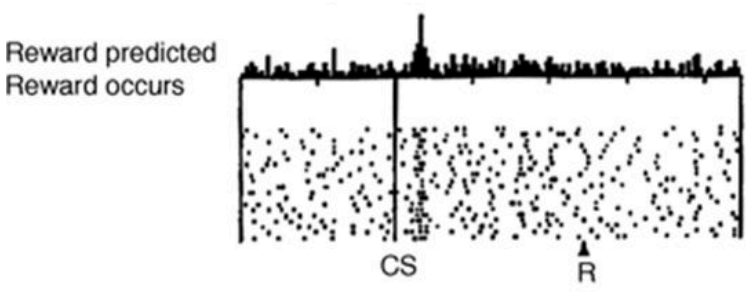
- Bootstrapping/TD-Learning:

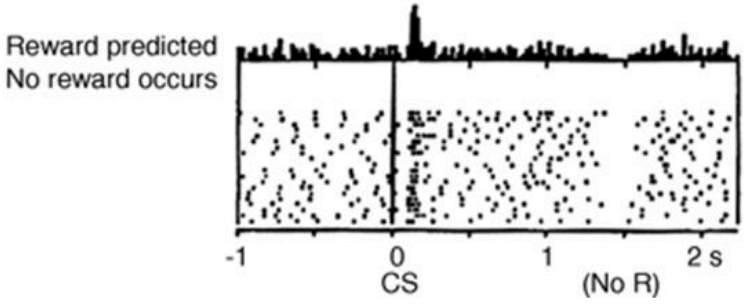
TD Error: $\delta$

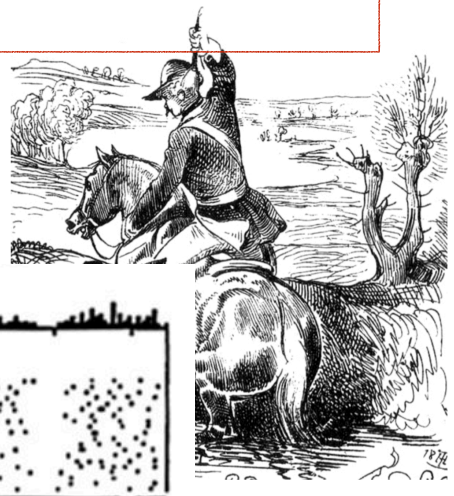
$$Q(s,a)_{k+1} = Q(s,a)_k + \eta(r + \gamma \max_{a' \in \mathcal{A}} Q(s',a')_k - Q(s,a)_k)$$

Target: $Y_k$

- Dopamine Reward Prediction Error Hypothesis:

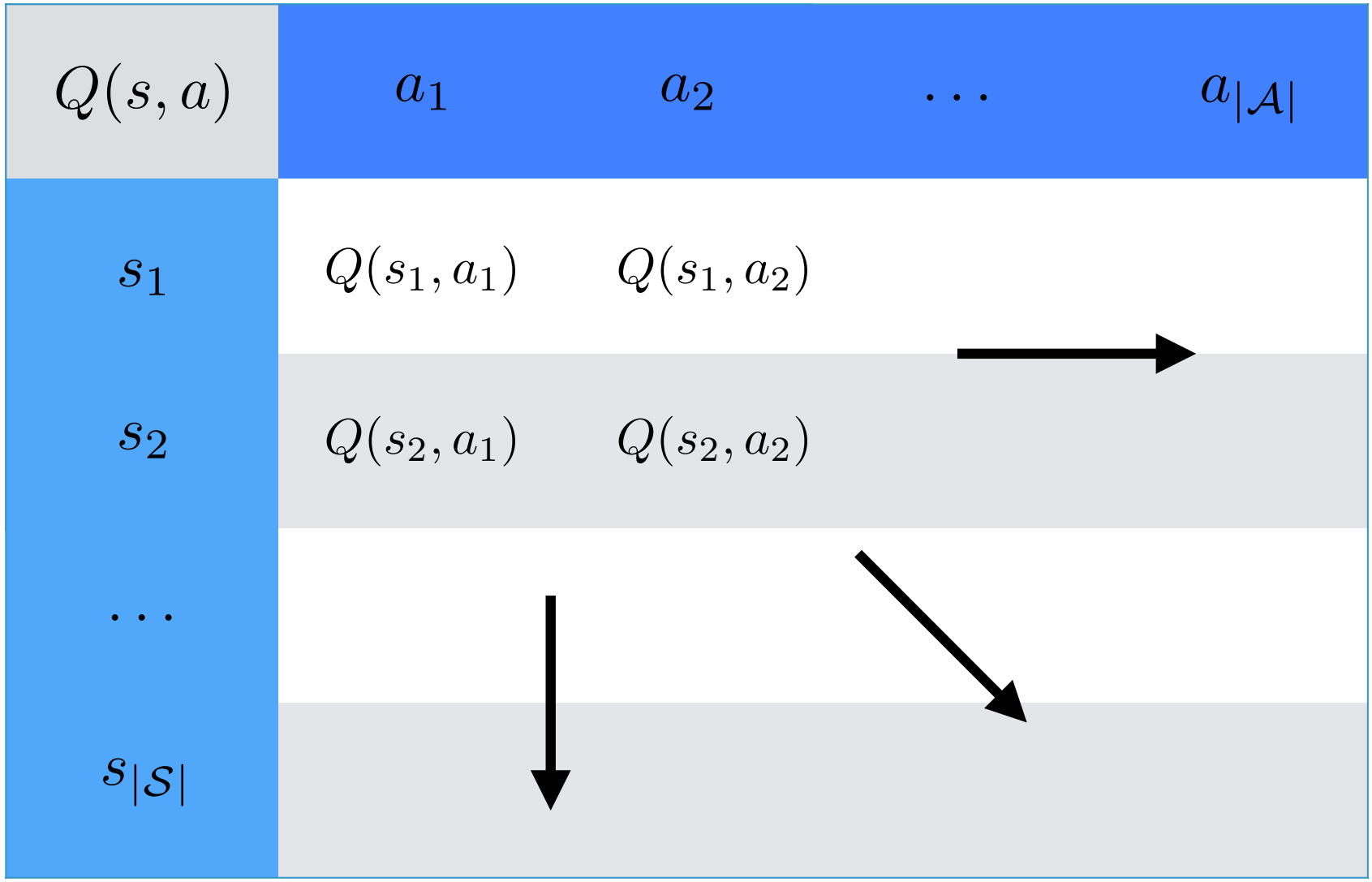


Schultz, Dayan, Montague (1997)

# The Curse of Dimensionality in DRL.

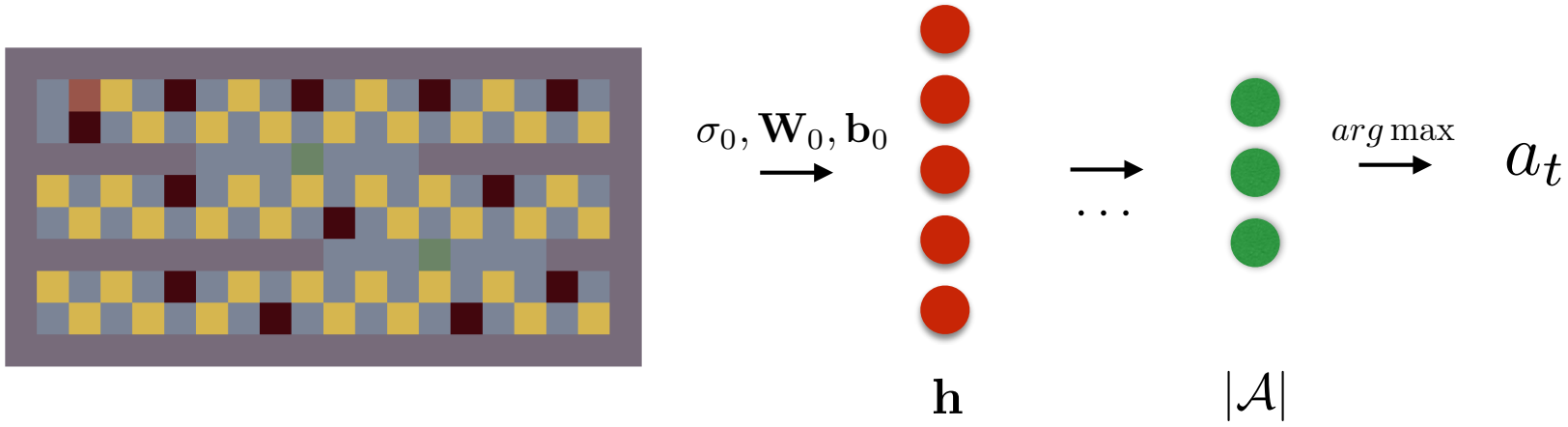


# Overcoming the Curse of Dimensionality.

Problem: What do we do if the state space is too large!?

Idea: Combat via generalisation by function approximation

$$Q(s,a) \longrightarrow Q(s,a;\theta)$$



# Fitted Q-Learning - Gordon (1996).

- Regression Problem - Mean Squared Bellman/TD Error:

$$\mathcal{L}_{MSBE} = \mathbb{E}_{s,a,r,s'}[(Q(s,a;\theta_k) - Y_k)^2]$$

$$Y_k = r + \gamma \max_{a' \in \mathcal{A}} Q(s', a'; \theta_k)$$

- Update Weights with SGD + Backprop:

$$\theta_{k+1} = \theta_k + \alpha(Y_k - Q(s,a;\theta_k))\nabla_{\theta_k} Q(s,a;\theta_k)$$

No convergence guarantees

Wasteful Online Updates

Weight updates change targets

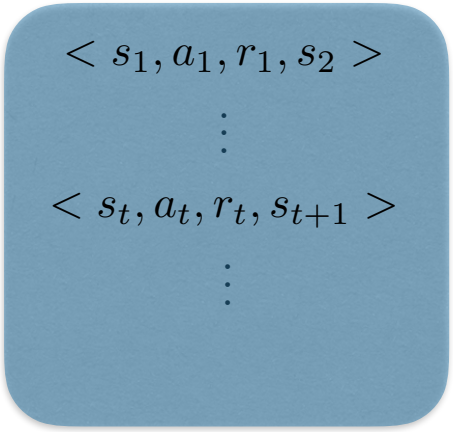
# DQNs - Mnih et al (2013, 2015).

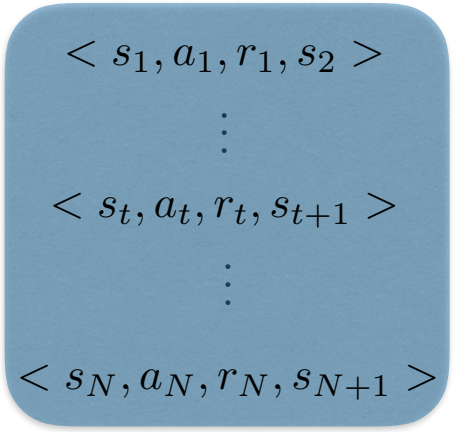
Wasteful Online Updates

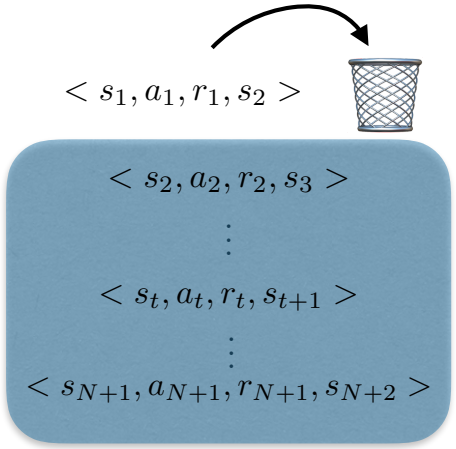
Store & reuse past transitions.

- Experience Replay: $\mathcal{D} = \{< s, a, r, s' >\}$ („Dataset“)

$$\mathcal{L}_{MSBE} = \mathbb{E}_{s,a,r,s' \sim \mathcal{U}(\mathcal{D})}[(Q(s, a; \theta_k) - Y_k]^2$$



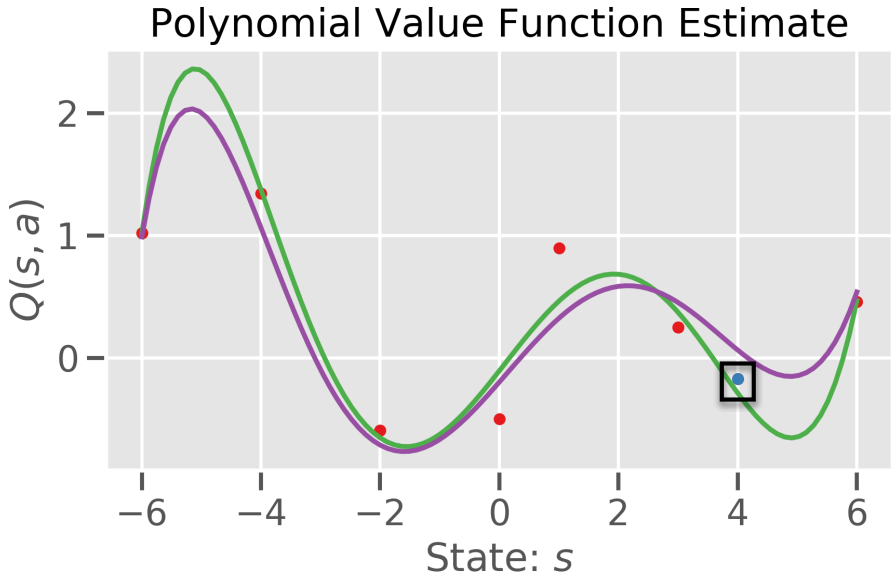



# DQNs - Mnih et al (2013, 2015).

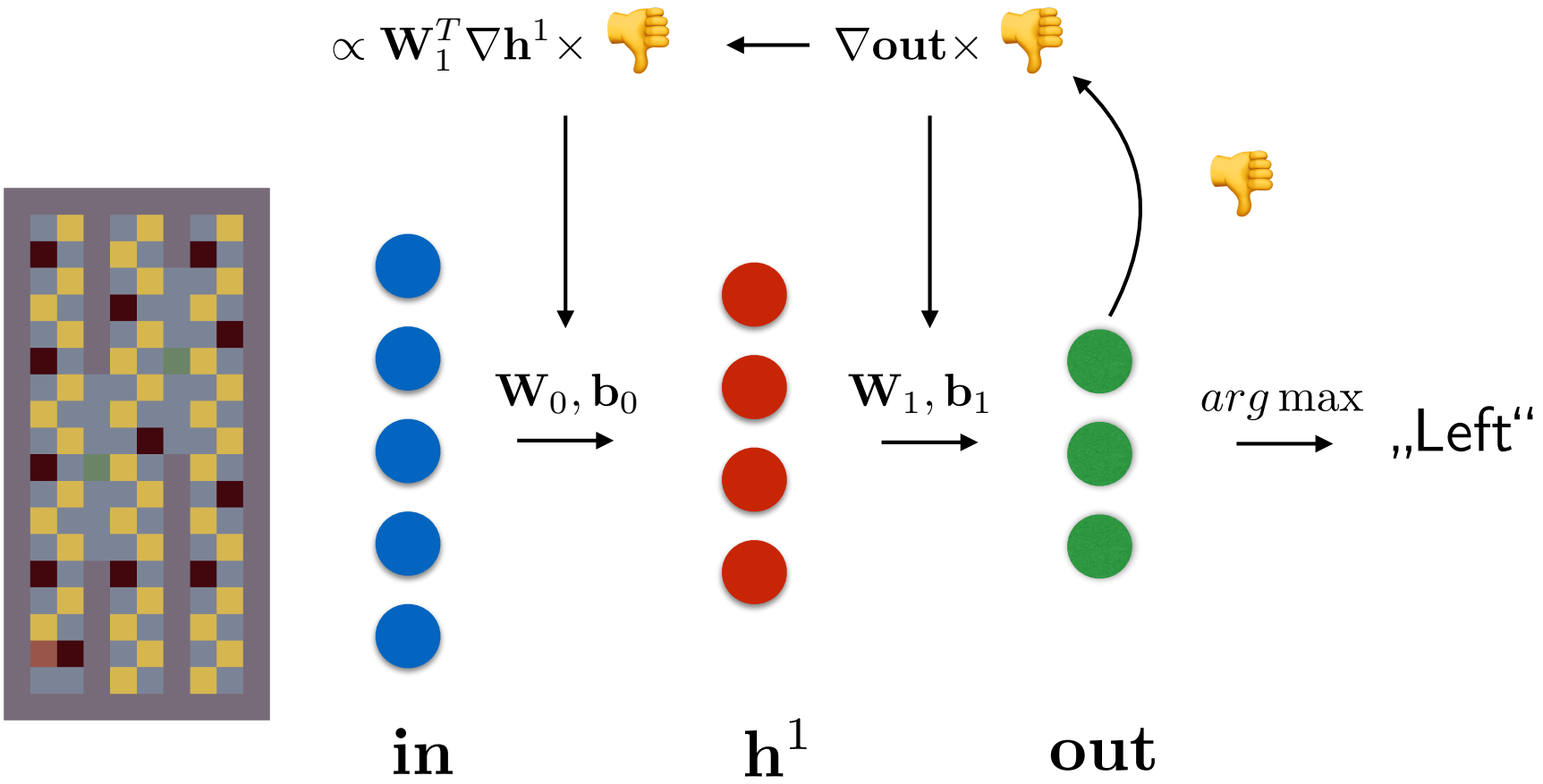
Weight updates change targets

Slowly changing target network.




- Target Networks: $Y_k = r + \gamma \max_{a' \in \mathcal{A}} Q(s', a'; \theta_k^-)$
  - Update every C iterations: $k \mod C = 0 : \theta_k^- \leftarrow \theta_k$

# An Intermezzo - Reverse Mode Automatic Differentiation.

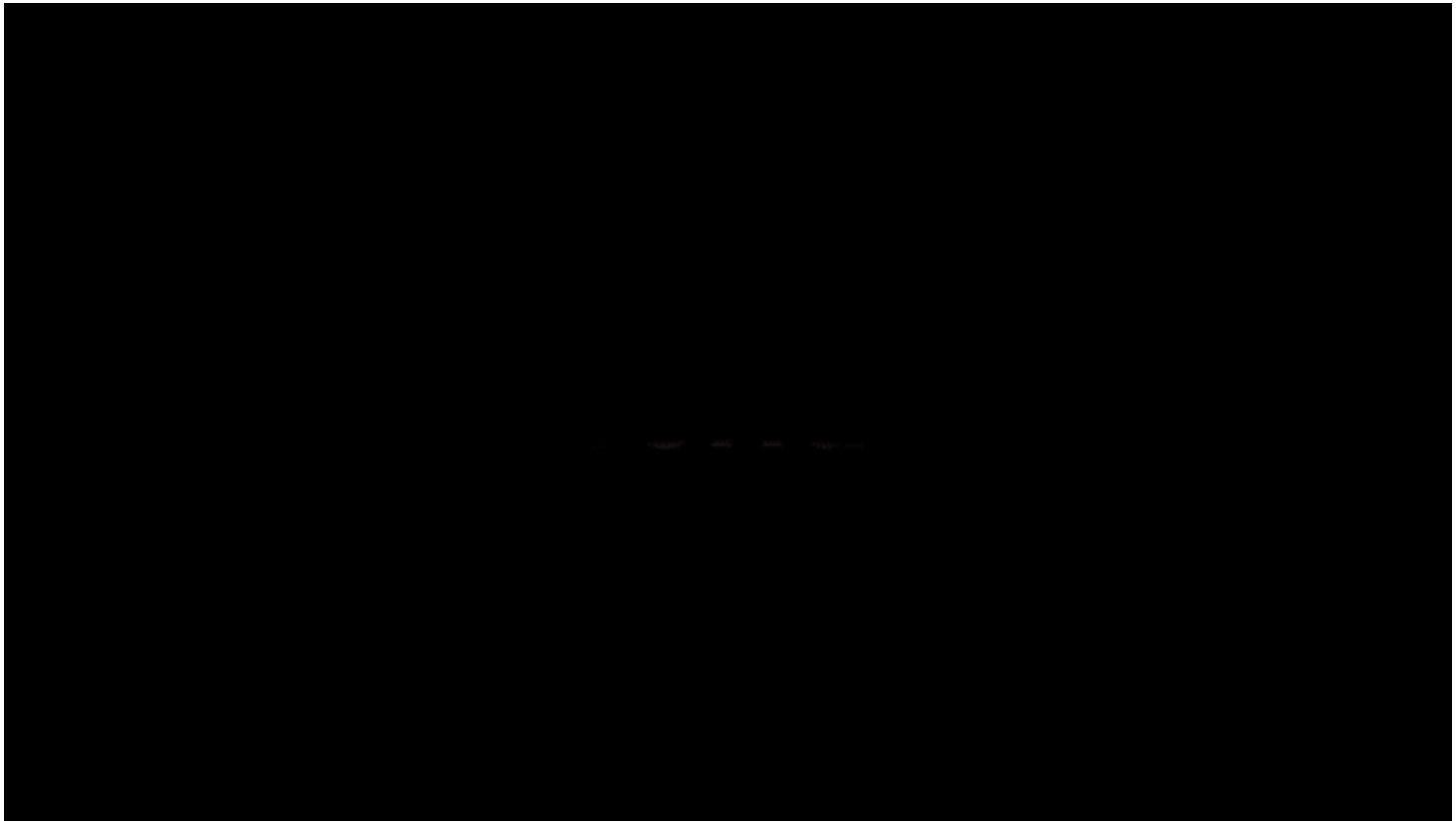



PyTorch

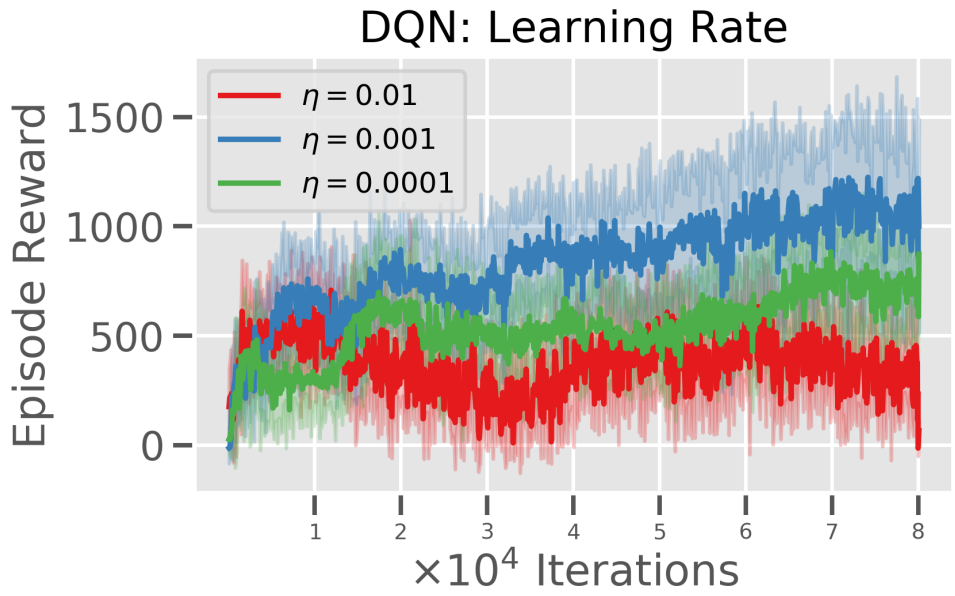
TensorFlow

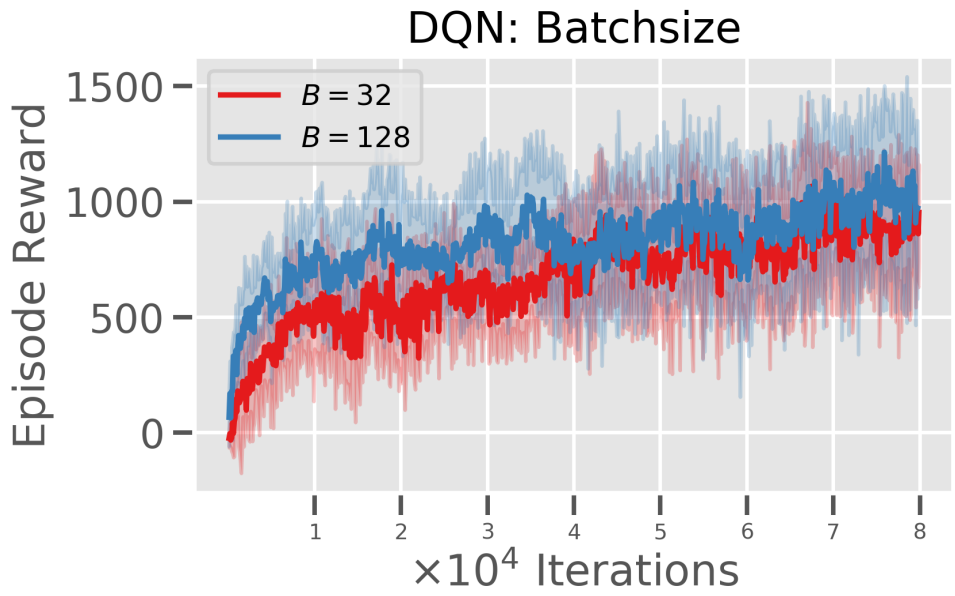
```
import torch
import torch.nn as nn
import torch.optim as optim
```

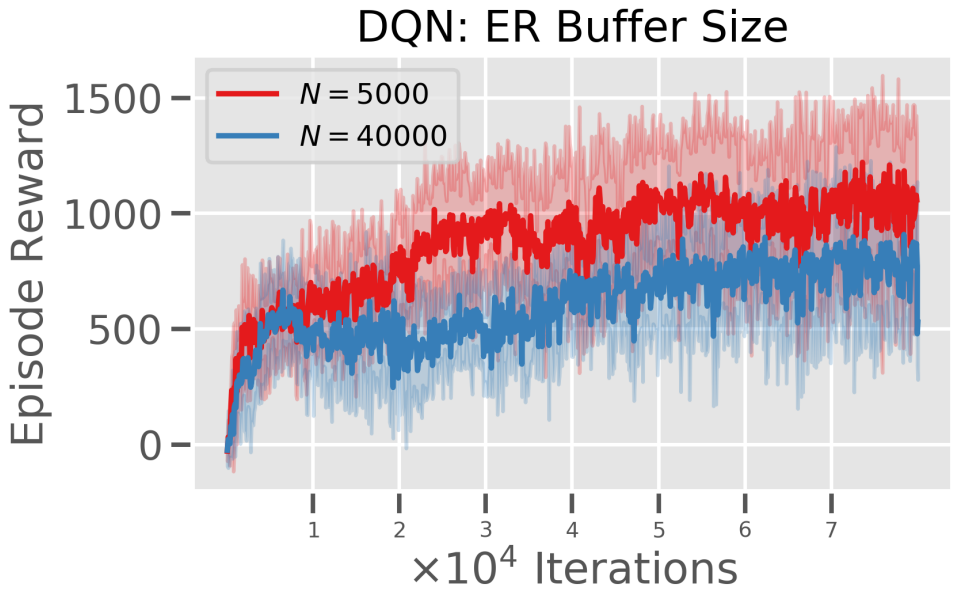
# A Toy Gridworld Example.



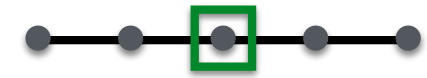
# Some DQN Hyperparameter Intuition.

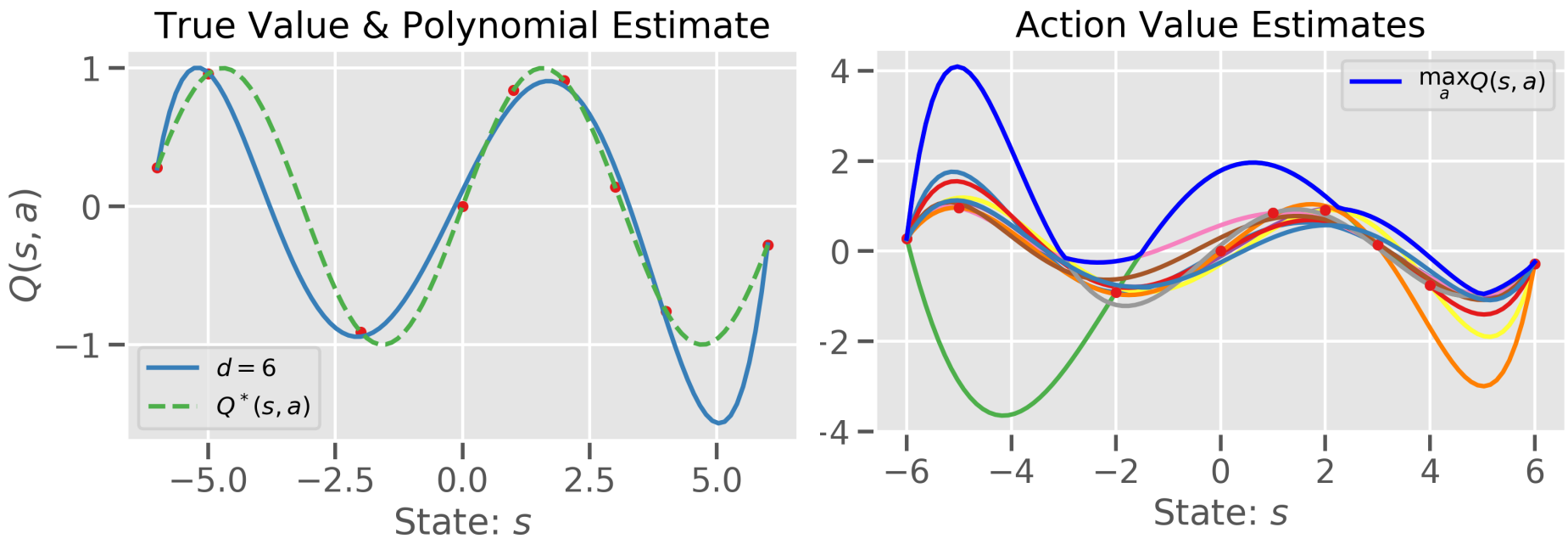


# Overestimation Bias in Q-Learning.





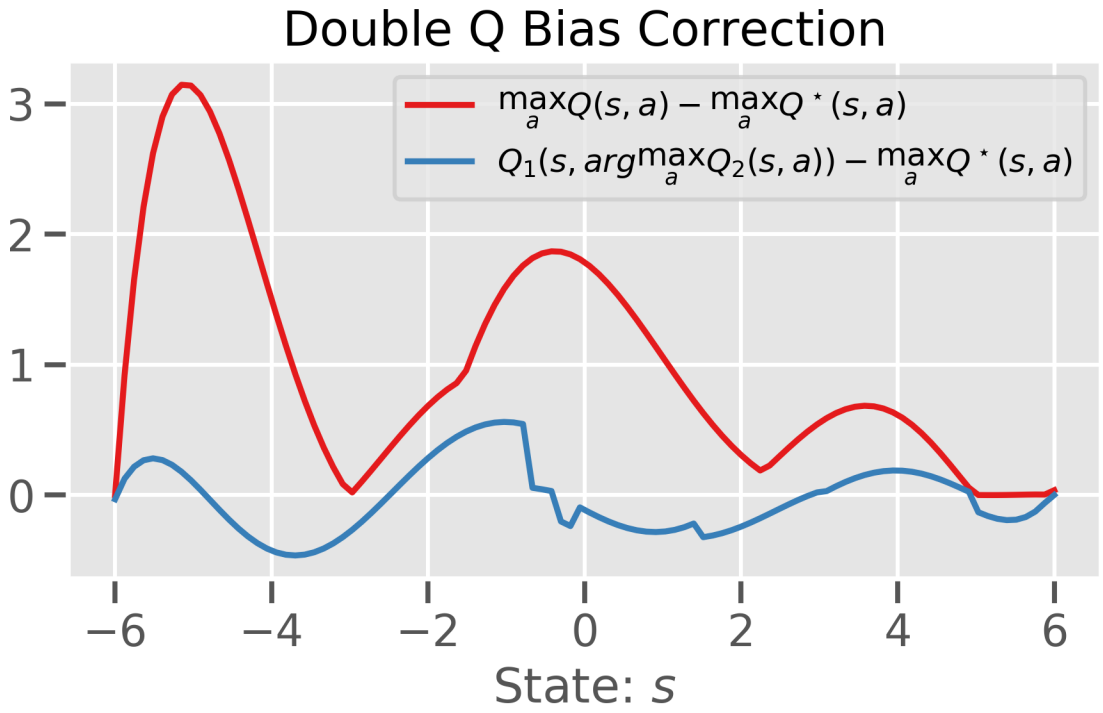

Reason: Maximization prefers overestimated values!

$$Y_k^{DQN} = r + \gamma Q(s', arg\max_{a' \in \mathcal{A}} Q(s', a'; \theta_k^-); \theta_k^-)$$

- Intuitive fix: Disentangle evaluation & action selection

# Double DQN - Van Hasselt et al (2015).

Double Q-Learning (Van Hasselt, 2010): Separate Q functions!

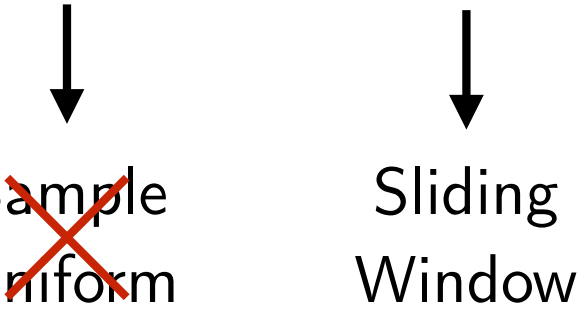


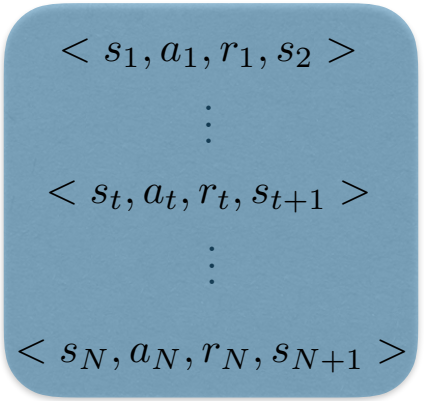

In DQN setting: Evaluation via target net & selection via online net

$$Y_k^{DDQN} = r + \gamma Q(s', arg\max_{a \in \mathcal{A}} Q(s', a; \theta_k); \theta_k^-)$$

# Prioritized ER - Schaul et al (2016).

Memory Replay = Computation + Storage

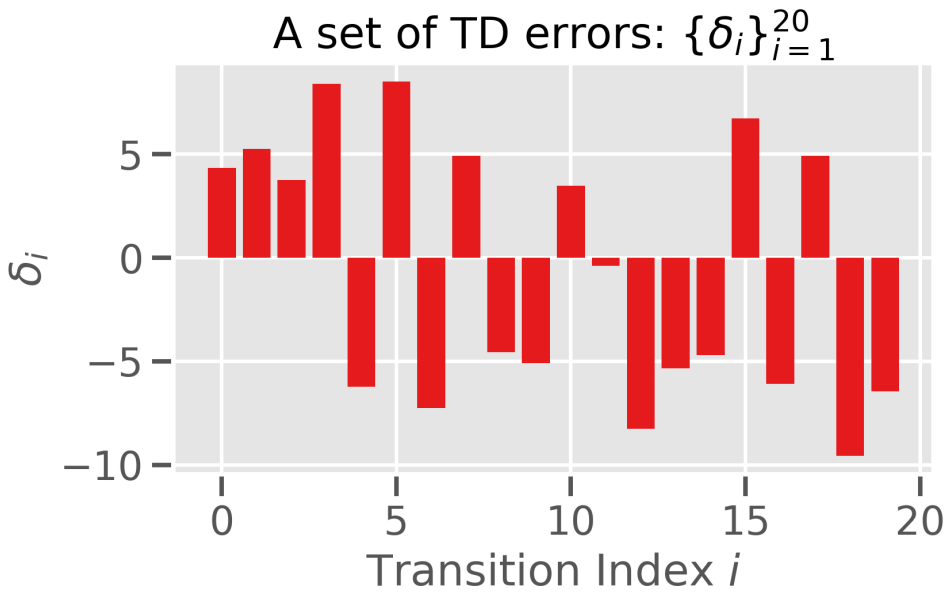


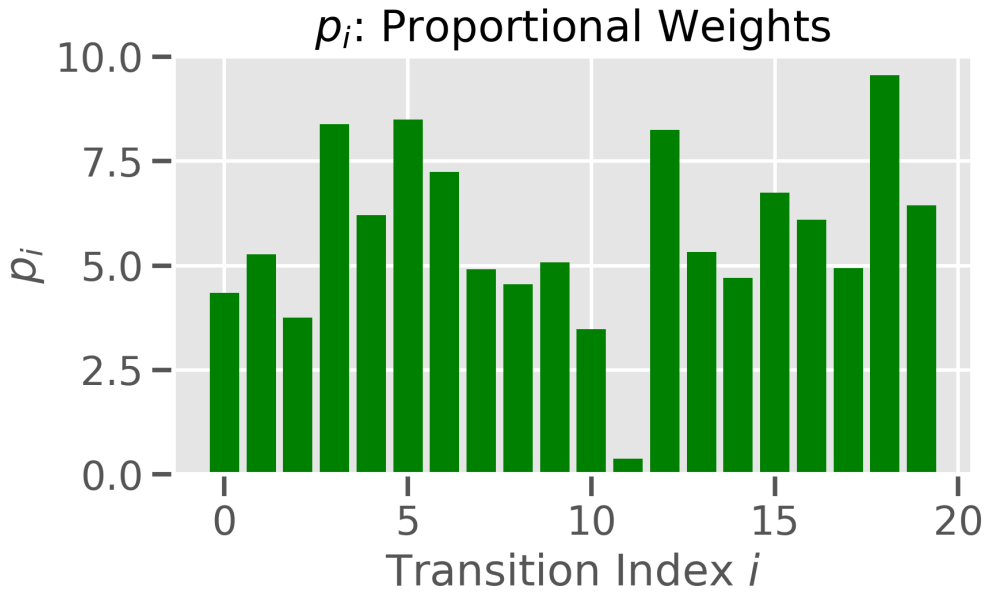



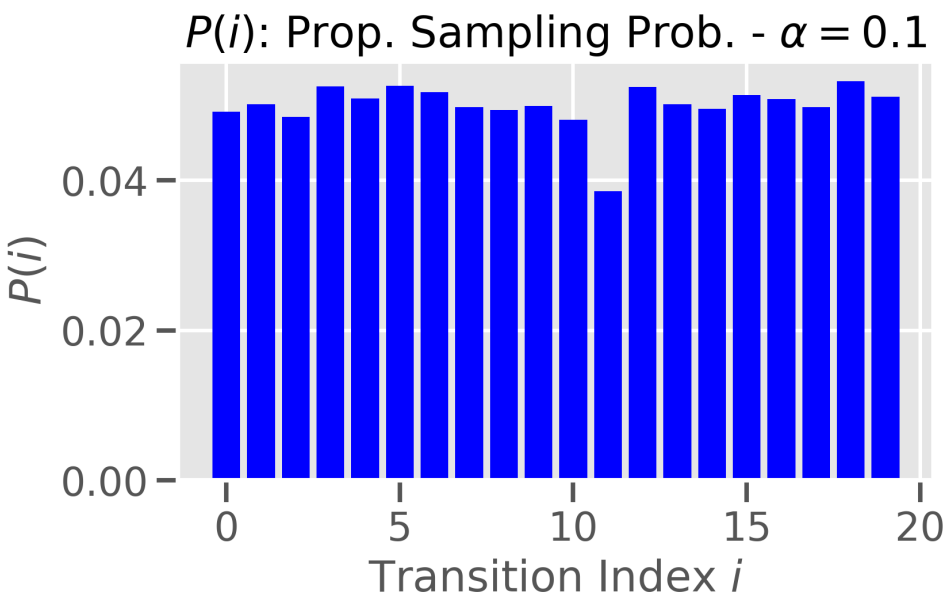

- Prioritization - Sample proportionately to learning progress: $|\delta|$
  - Rank-based: $p_i = \frac{1}{rank(i)}$
  - Proportional: $p_i = |\delta_i| + \epsilon$

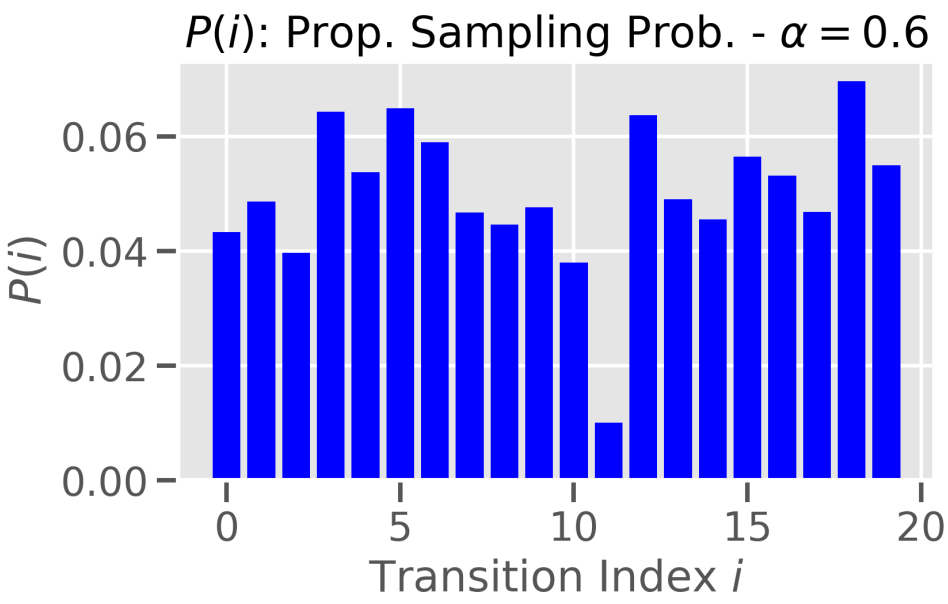
$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

# Prioritized Experience Replay - Schaul et al (2016).



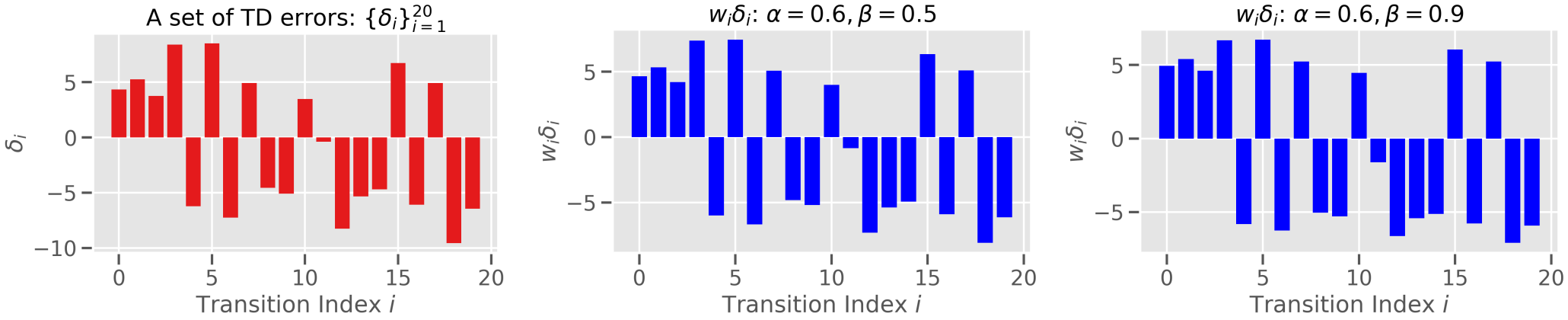
# Prioritized Experience Replay - Schaul et al (2016).

$$s, a, r, s' \sim \mathcal{U}(\mathcal{D}) \quad \neq \quad s, a, r, s' \sim P(\mathcal{D})$$

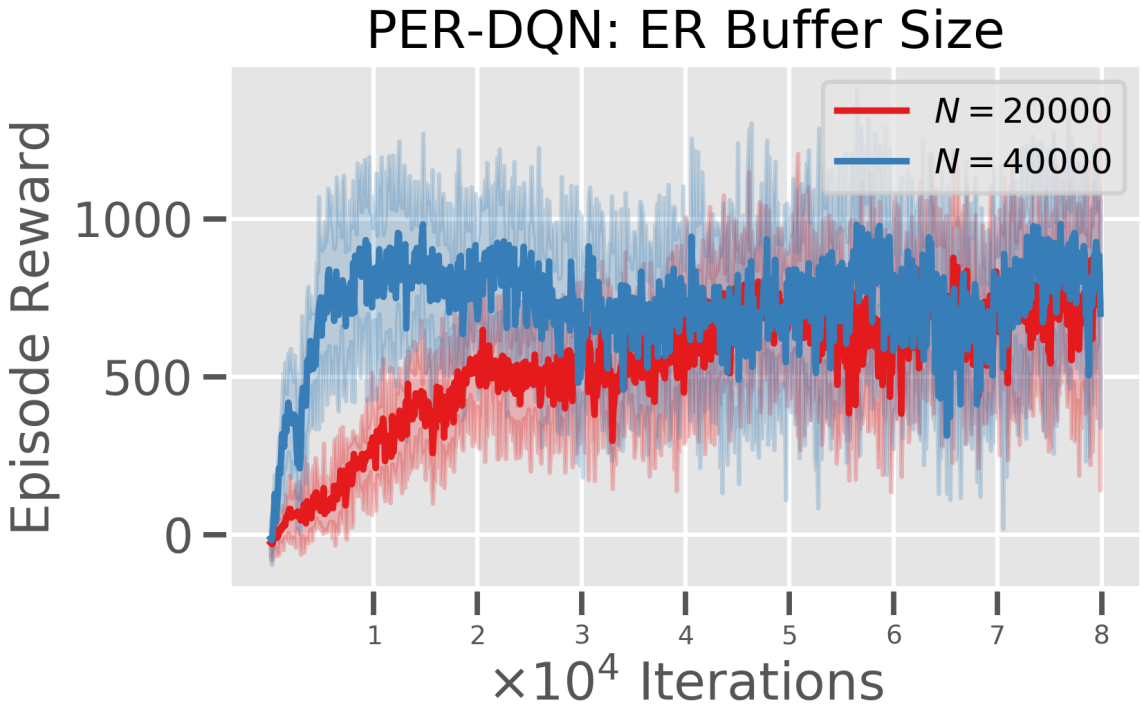
- Solution: Importance sampling!
- Starting from 0.4 linearly anneal $\beta$ to 1.

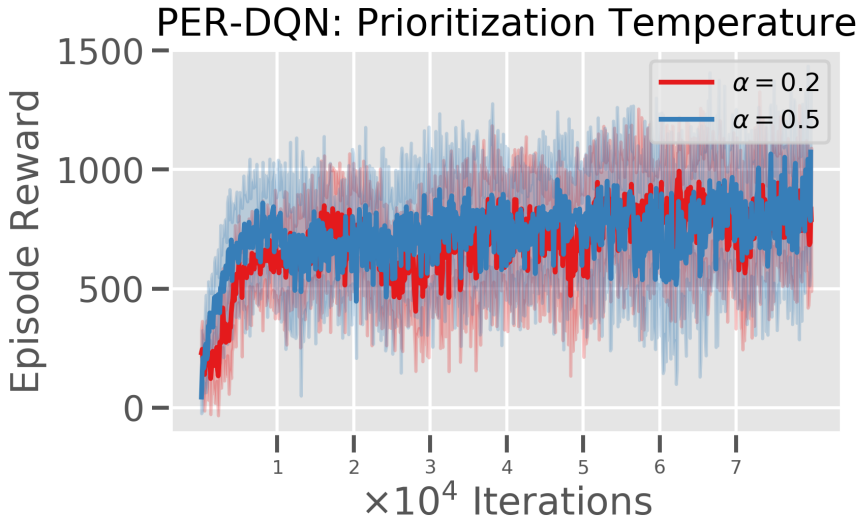
$$w_i = \left( \frac{1}{N} \frac{1}{P(i)} \right)^{\beta}$$

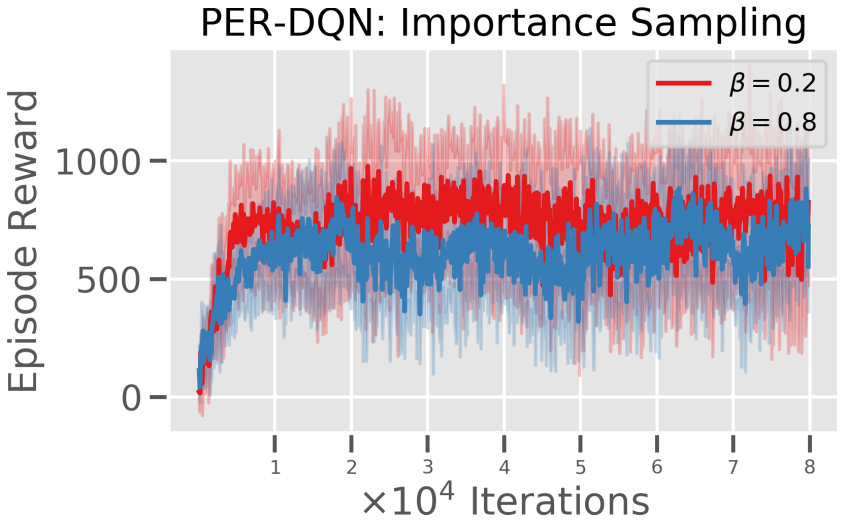
$$\mathcal{L}_{IS-MSBE} = \mathbb{E}_{s,a,r,s' \sim \mathcal{P}}[w(Q(s, a; \theta_k) - Y_k)^2]$$



# Prioritized Experience Replay - Schaul et al (2016).

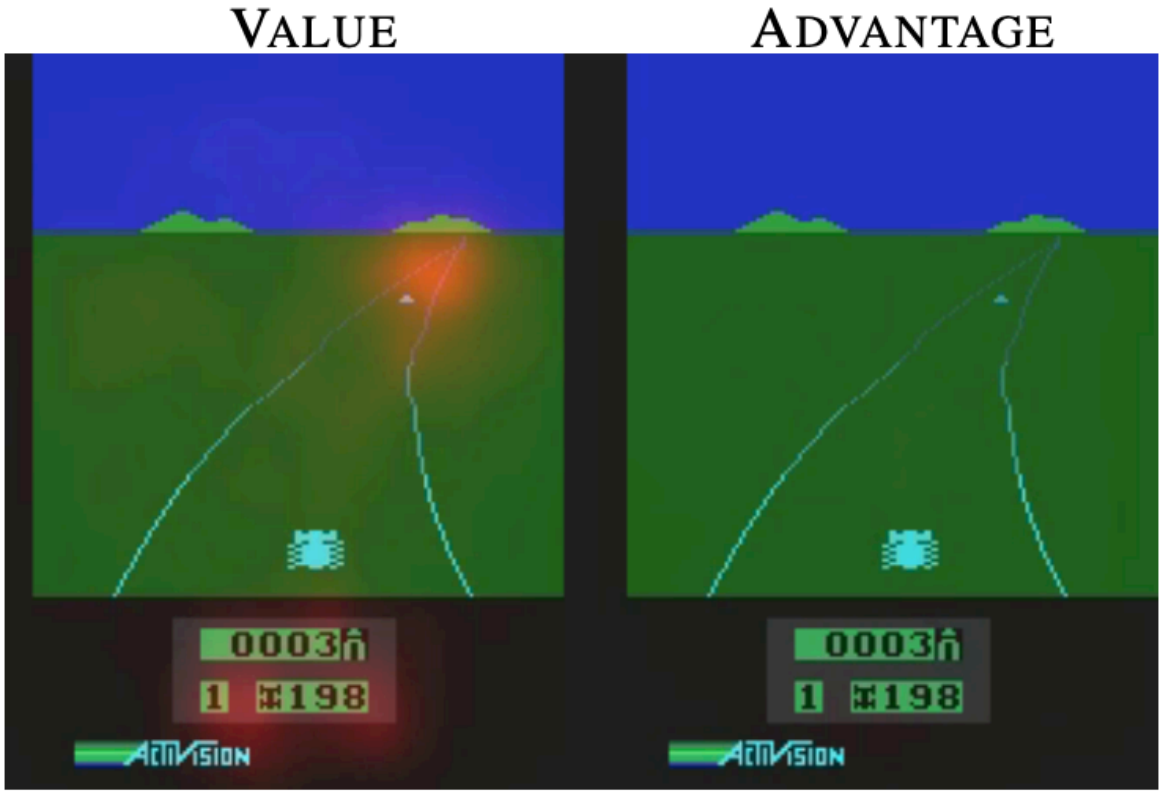


# Dueling Streams = Better Propagation

- The Advantage: $A(s,a)^\pi = Q(s,a)^\pi - V^\pi(s)$



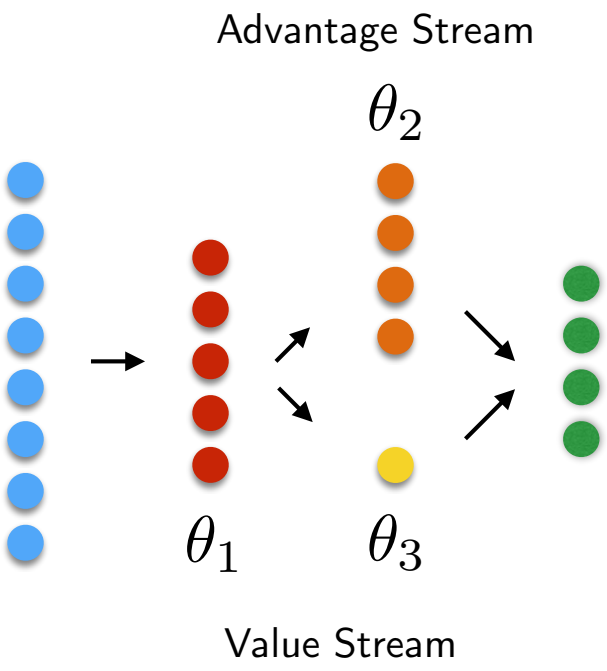

Wang et al. (2016)

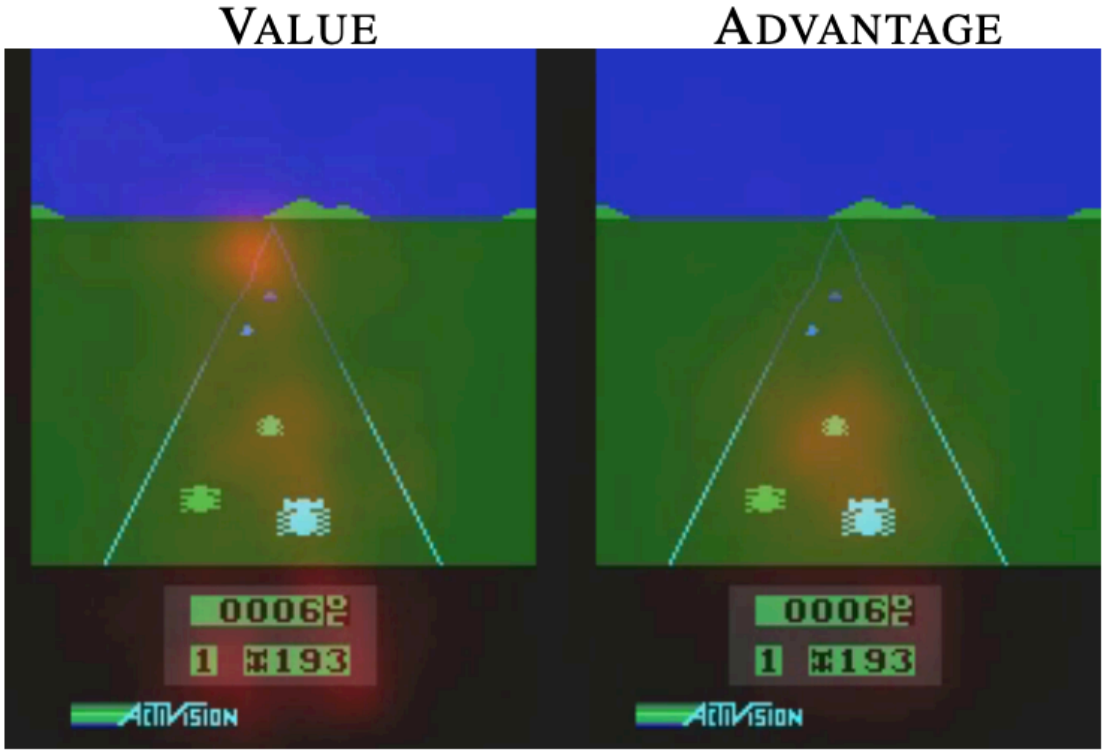
- Problem: Some times action doesn't affect env!
- Solution: Split action value into two streams!

# Dueling Streams = Better Propagation

$$Q(s,a;\theta^1,\theta^2,\theta^3) = V(s;\theta^1,\theta^3) + (A(s,a;\theta^1,\theta^2) - \frac{1}{|\mathcal{A}|}\sum A(s,a;\theta^1,\theta^2))$$

Easy architectural change! No effort whatsoever




Wang et al. (2016)

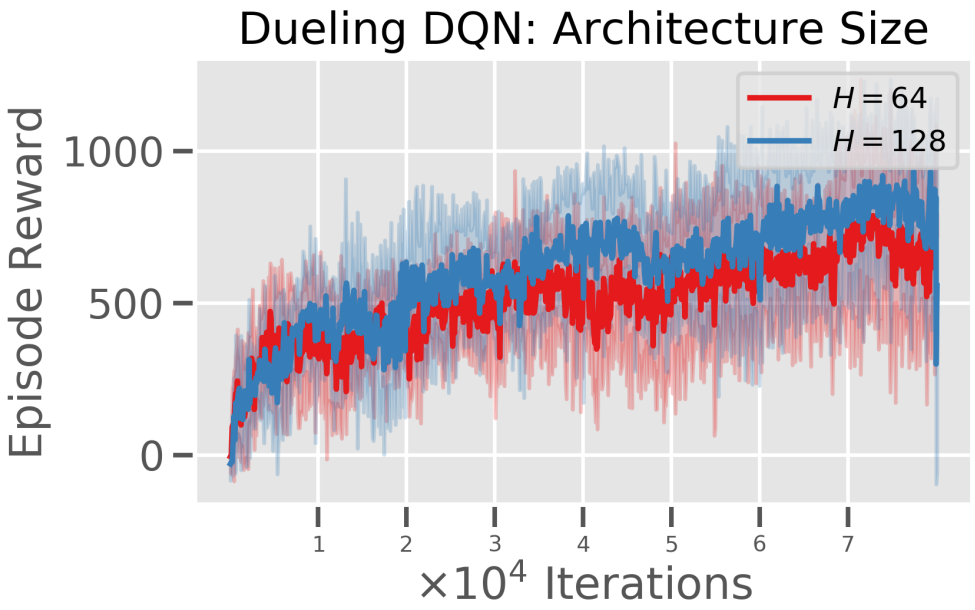
# The Dueling DQN Architecture - Wang et al (2016).

```python
class MLP_DuelingDQN(nn.Module):
    def __init__(self, INPUT_DIM, HIDDEN_SIZE, NUM_ACTIONS):
        super(MLP_DuelingDQN, self).__init__()
        self.action_space_size = NUM_ACTIONS
        self.feature = nn.Sequential(
            nn.Linear(INPUT_DIM, HIDDEN_SIZE),
            nn.ReLU()
        )

        self.advantage = nn.Sequential(
            nn.Linear(HIDDEN_SIZE, HIDDEN_SIZE),
            nn.ReLU(),
            nn.Linear(HIDDEN_SIZE, self.action_space_size)
        )
        self.value = nn.Sequential(
            nn.Linear(HIDDEN_SIZE, HIDDEN_SIZE),
            nn.ReLU(),
            nn.Linear(HIDDEN_SIZE, 1)
        )

    def forward(self, x):
        x = self.feature(x)
        advantage = self.advantage(x)
        value = self.value(x)
        return value + advantage - advantage.mean()

    def act(self, state, epsilon):
        if random.random() > epsilon:
            state = Variable(torch.FloatTensor(state).unsqueeze(0))
            q_value = self.forward(state)
            action = q_value.max(1)[1].data[0]
        else:
            action = random.randrange(self.action_space_size)
        return action
```



# The One Equation Summary.

Motivation: Curse of Dimensionality in large state spaces + efficiency

Fitted Q-L. $Y_k = r + \gamma \max_{a' \in \mathcal{A}} Q(s', a'; \theta_k)$

DQN $Y_k = r + \gamma \max_{a' \in \mathcal{A}} Q(s', a'; \theta_k^-)$ ⊕ ER

Double DQN $Y_k^{DDQN} = r + \gamma Q(s', arg\max_{a \in \mathcal{A}} Q(s', a; \theta_k); \theta_k^-)$

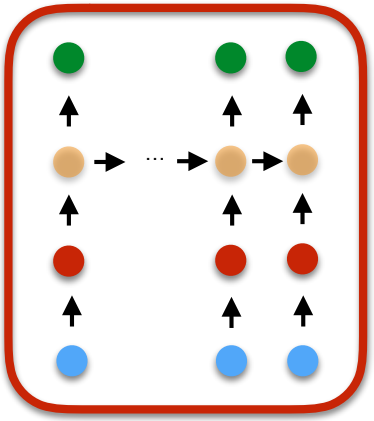
PER $p_i = |\delta_i| + \epsilon$ ⊕ $P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$ ⊕ $w_i = \left(\frac{1}{N}\frac{1}{P(i)}\right)^\beta$

Dueling DQN $Q(s, a; \theta^1, \theta^2, \theta^3) = V(s; \theta^1, \theta^3) + (A(s, a; \theta^1, \theta^2) - \frac{1}{|\mathcal{A}|}\sum A(s, a; \theta^1, \theta^2))$

Many Open Qs: Intrinsic Motivation, Partial Obs., Multi-Agent, Transfer

# A Zoo of Extensions.

Hausknecht & Stone (2015) - Deep Recurrent Q Networks.

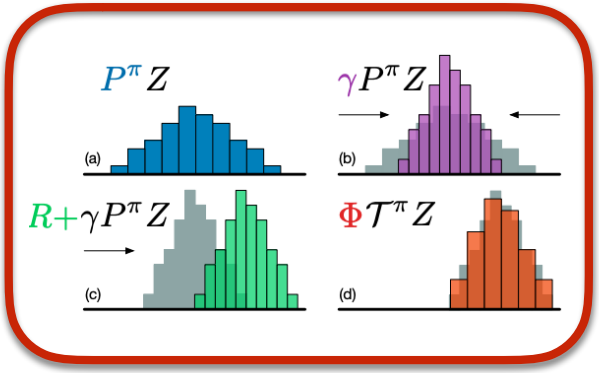


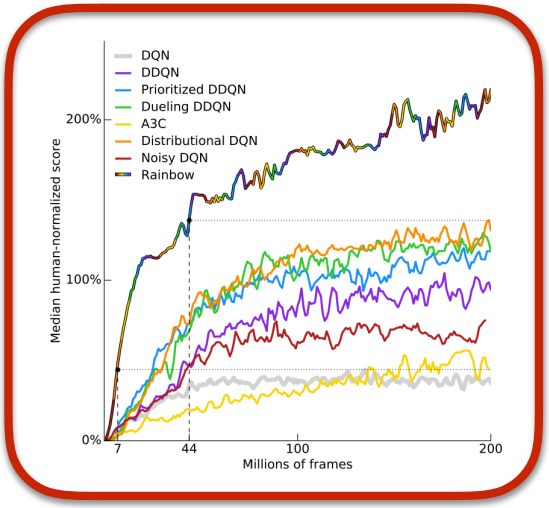
$$y = (b + Wx) + (\tilde{b} \odot \epsilon^b + (\tilde{W} \odot \epsilon^W)x)$$

Net learns E-to-E noise to explore

Fortunato et al (2018) - Noisy Networks.

Munos et al (2016) - Distributional DQN.




Hessel et al (2018) - Rainbow.

**Robert Tjarko Lange**

Deep Reinforcement Learning PhD Student @SprekelerLab

Berlin, London

ECN Berlin

Email

Twitter

LinkedIn

Github

Google Scholar

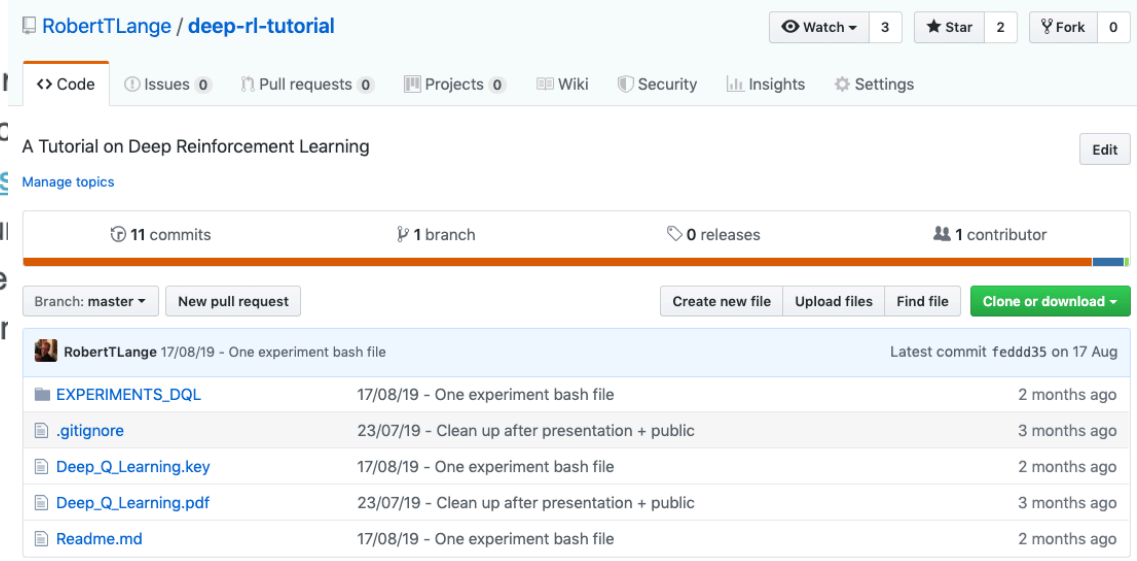
# A Primer on Deep Q-Learning - Part 1/2

31 minute read

**Published:** August 11, 2019

Before starting to write a blog post I always ask myself - “What is the added value?”. There is a lot of awesome ML material out there. And a lot of duplicates as well. Especially when it comes to all the flavors of Deep Reinforcement Learning. So you might wonder what is the added value of this two part blog post on Deep Q-Learning? It is threefold.

1. We will r
   Reinforc
   (Watkins
   value fu
   ingredie
   algorithr



# References.


Gordon, G. J. (1996). Stable fitted reinforcement learning. In Advances in neural information processing systems (pp. 1052-1058).

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., ... & Silver, D. (2018, April). Rainbow: Combining improvements in deep reinforcement learning. In Thirty-Second AAAI Conference on Artificial Intelligence.

Lin, L. J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. Machine learning, 8(3-4), 293-321.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Petersen, S. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529.

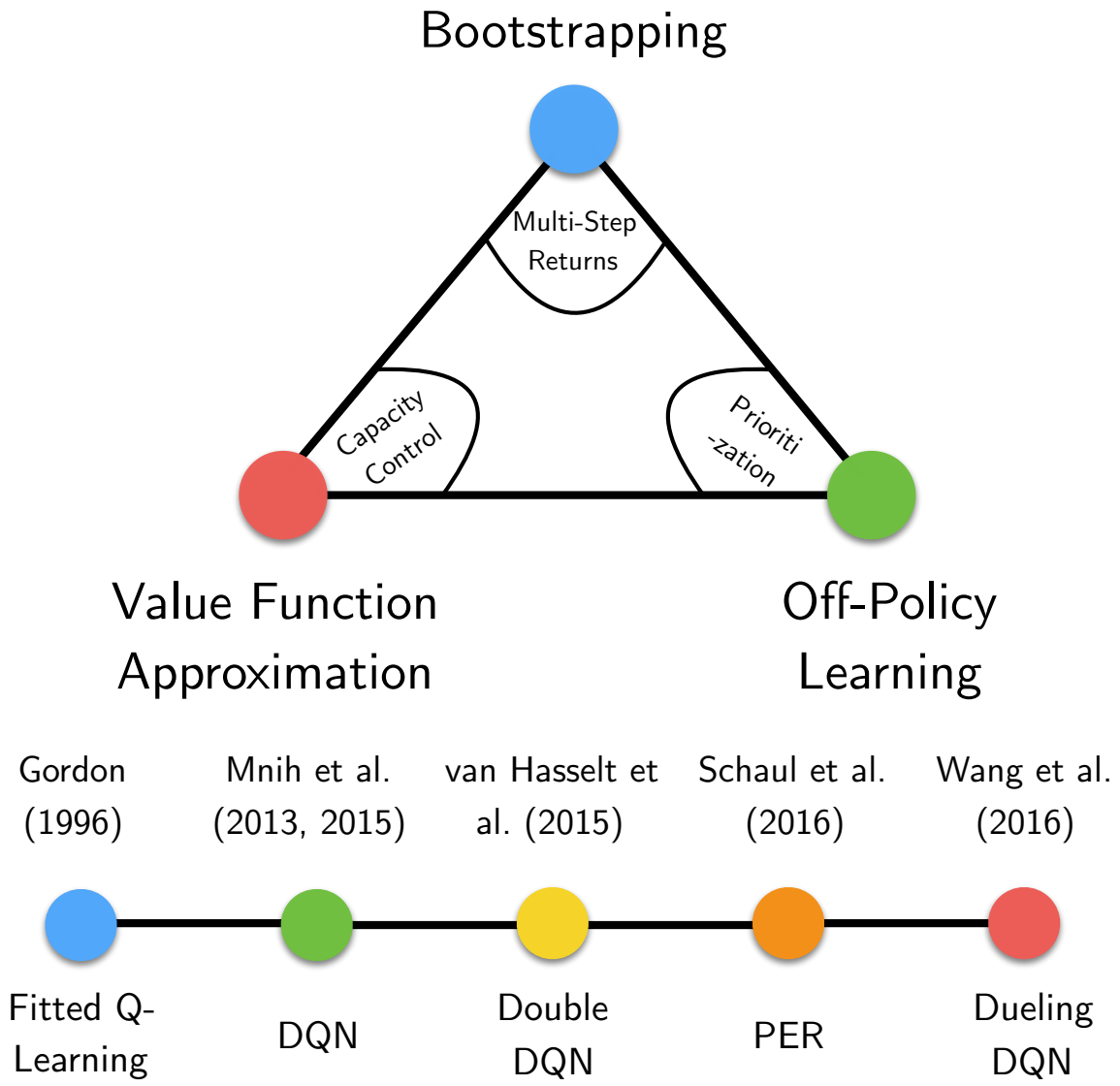
Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015). Prioritized experience replay. arXiv preprint arXiv:1511.05952.

Van Hasselt, H., Guez, A., & Silver, D. (2016, March). Deep reinforcement learning with double q-learning. In Thirtieth AAAI conference on artificial intelligence.

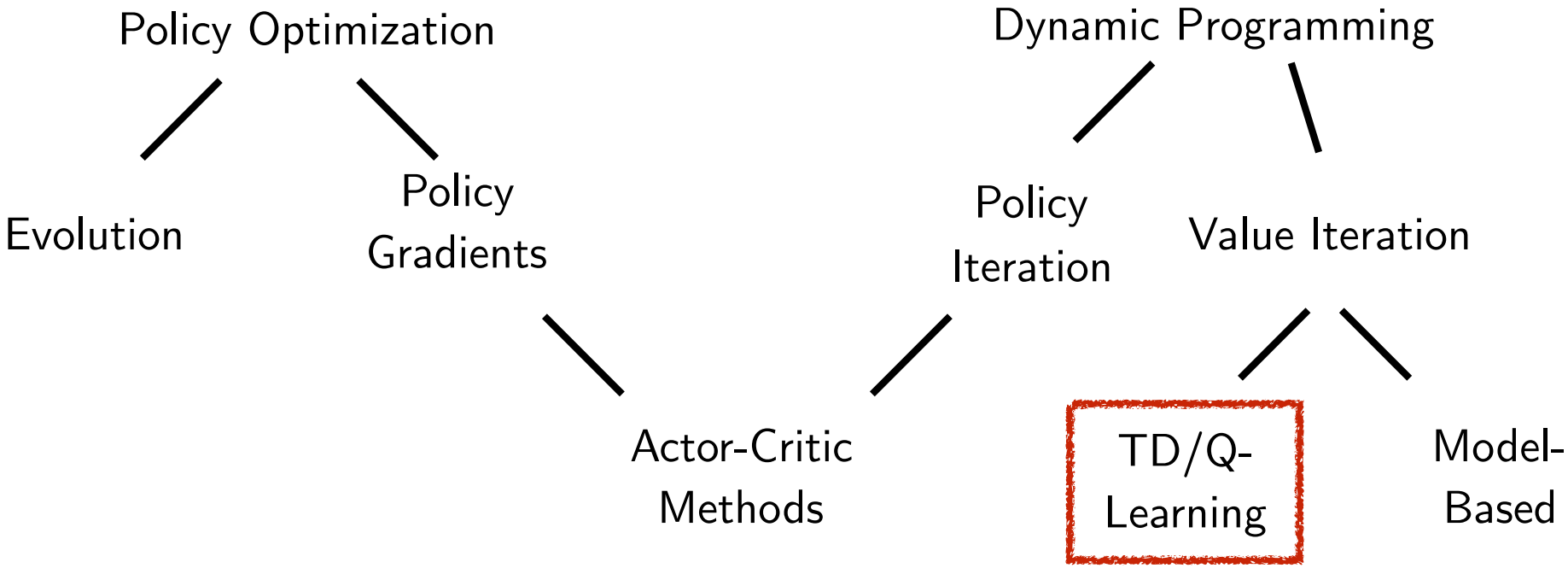
Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., & De Freitas, N. (2015). Dueling network architectures for deep reinforcement learning. arXiv preprint arXiv:1511.06581.

## Supplementary Material

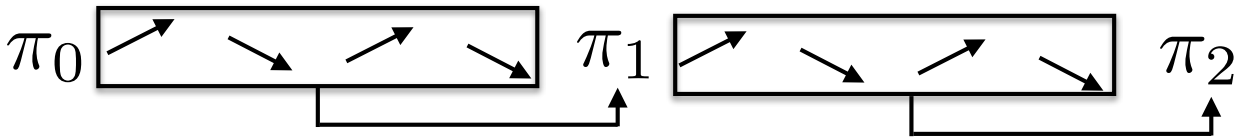
# The Deadly Triad of DRL



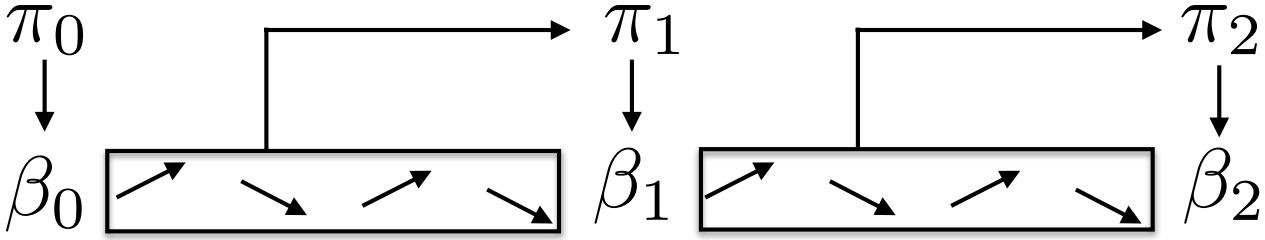
# A Zoo of Different Approaches.



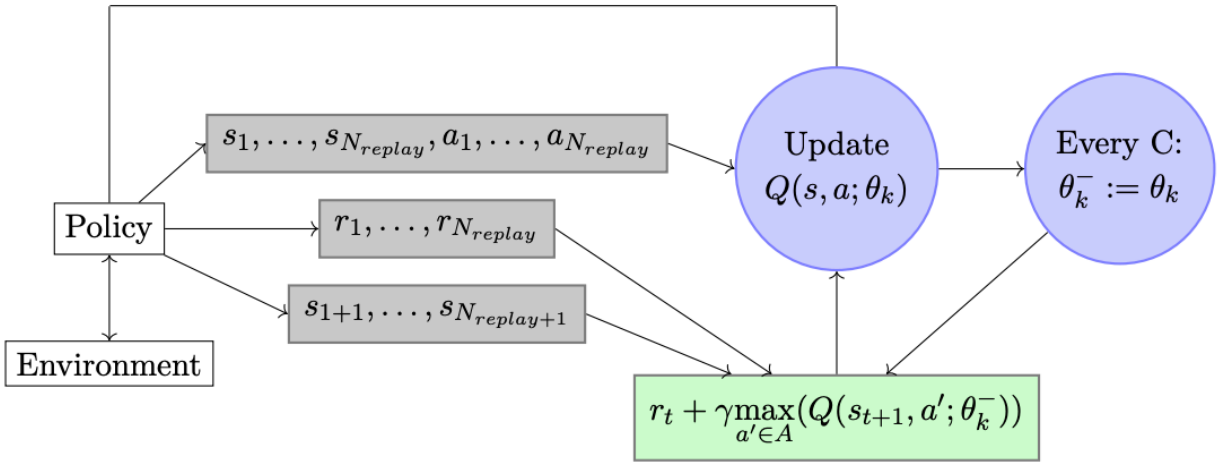

- „On-Policy“:

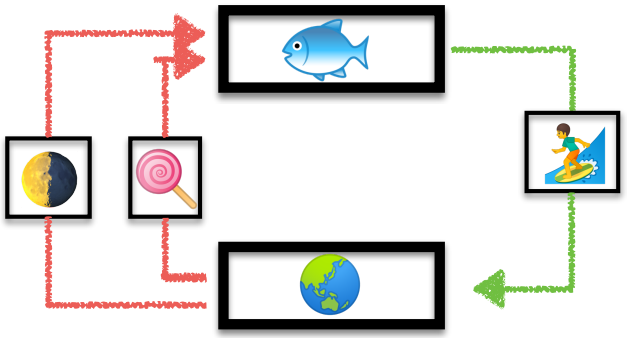



- „Off-Policy“:



# ATARI DQN - Mnih et al (2013, 2015).

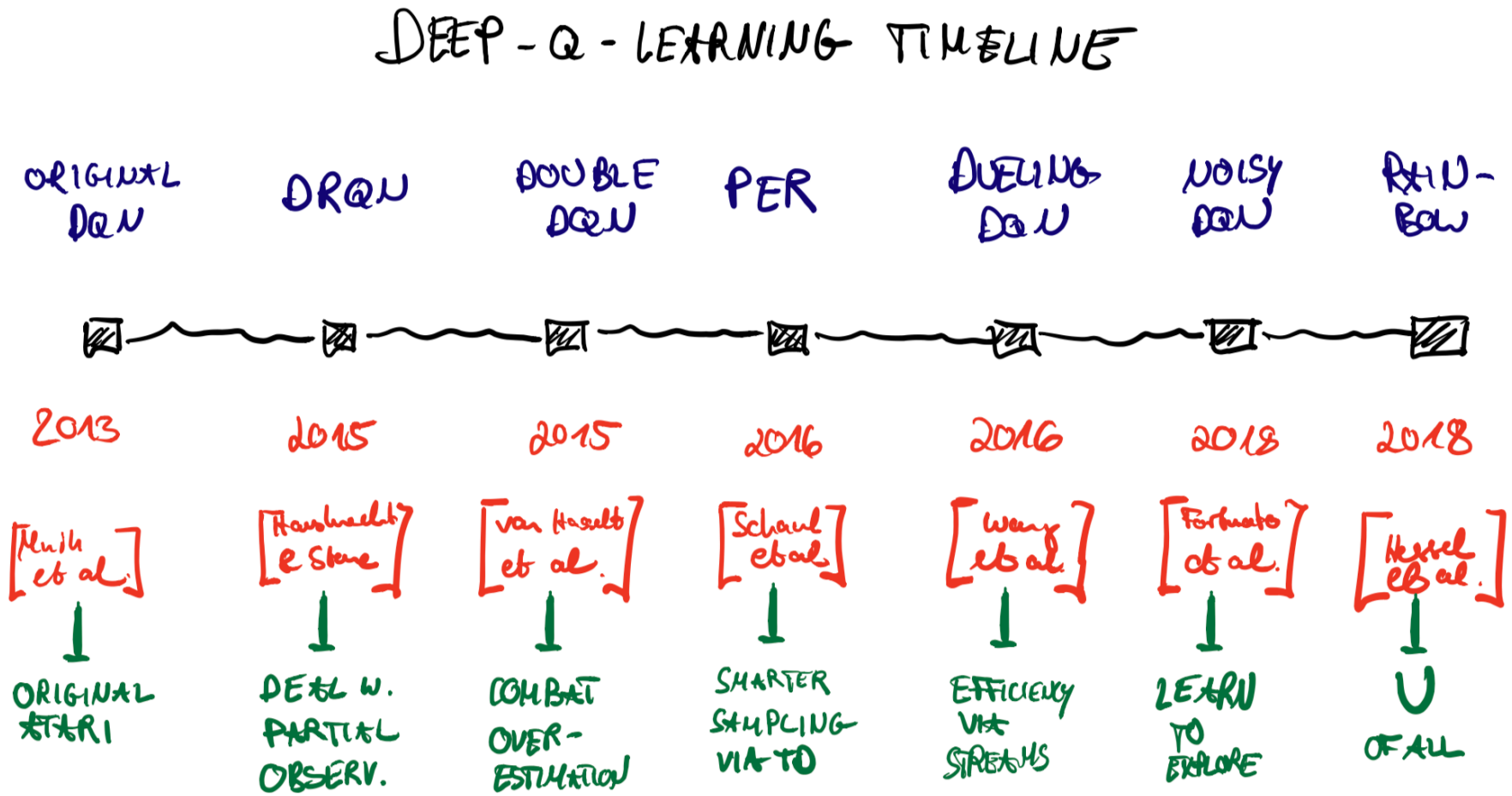


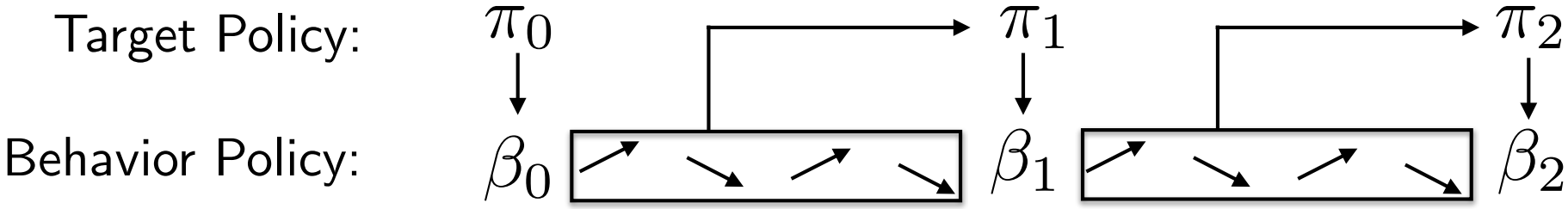



*Source: François-Lavet, Vincent, et al. "An introduction to deep reinforcement learning." Foundations and Trends® in Machine Learning 11.3-4 (2018): 219-354.*

- ATARI-DQN: Directly learn from pixels: "end-to-end".
- Hacks: reward clipping, down-sampling/grey scaling, skipping/concat.
- Hyperparameters: RMSprop Optimizer, Architecture, #iterations between target network update, ER buffer capacity (memory)

# In Summary.


DEEP-Q-LEARNING TIMELINE
ORIGINAL DQN
DRQN
DOUBLE DQN
PER
DUELING DQN
NOISY DQN
RAINBOW
2013
2015
2015
2016
2016
2018
2018
ORIGINAL ATARI
DEAL W. PARTIAL OBSERV.
COMBAT OVER-ESTIMATION
SMARTER SAMPLING VIA TD
EFFICIENCY VIA STREAMS
LEARN TO EXPLORE
U OF ALL


Target Policy:
Behavior Policy:
$\pi_0$
$\pi_1$
$\pi_2$
$\beta_0$
$\beta_1$
$\beta_2$

Fitted Q-L.

$$Y_k = r + \gamma \max_{a' \in \mathcal{A}} Q(s', a'; \theta_k)$$

DQN

$$Y_k = r + \gamma \max_{a' \in \mathcal{A}} Q(s', a'; \theta_k^-) \quad \oplus \quad \text{ER}$$

Double DQN

$$Y_k^{DDQN} = r + \gamma Q(s', arg\max_{a \in \mathcal{A}} Q(s', a; \theta_k); \theta_k^-)$$

PER

$$p_i = |\delta_i| + \epsilon \quad \oplus \quad P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \quad \oplus \quad w_i = \left(\frac{1}{N}\frac{1}{P(i)}\right)^\beta$$

Dueling DQN

$$Q(s, a; \theta^1, \theta^2, \theta^3) = V(s; \theta^1, \theta^3) + (A(s, a; \theta^1, \theta^2) - \frac{1}{|\mathcal{A}|}\sum A(s, a; \theta^1, \theta^2))$$

Noisy Nets

$$A(x) = (b + Wx) + (\tilde{b} \odot \epsilon^b + (\tilde{W} \odot \epsilon^W)x)$$

Categorical DQN

$$Q(x_{t+1}, a; \theta) = \sum_{i=0}^{N-1} z_i p_i(x_{t+1}, a; \theta) \quad \oplus \quad \nabla_\theta KL(m||p_\theta)$$

QR-DN

$$Q(x_{t+1}, a; \theta) = \sum_{i=0}^{N-1} z_i(x_{t+1}, a; \theta) p_i \quad \oplus \quad \nabla_z \sum_{i=0}^{N-1} \mathbb{E}_x \left(|x - z||\tau_i - \delta_{x<z}|\right)$$

# Prioritized Experience Replay - Schaul et al (2016).

```python
class NaivePrioritizedBuffer(object):
    def __init__(self, capacity, prob_alpha=0.6):
        self.prob_alpha = prob_alpha
        self.capacity   = capacity
        self.buffer     = []
        self.pos        = 0
        self.priorities = np.zeros((capacity,), dtype=np.float32)

    def push(self, state, action, reward, next_state, done):
        max_prio = self.priorities.max() if self.buffer else 1.0

        if len(self.buffer) < self.capacity:
            self.buffer.append((state, action, reward, next_state, done))
        else:
            self.buffer[self.pos] = (state, action, reward, next_state, done)

        self.priorities[self.pos] = max_prio
        self.pos = (self.pos + 1) % self.capacity

    def sample(self, batch_size, beta=0.4):
        if len(self.buffer) == self.capacity: prios = self.priorities
        else: prios = self.priorities[:self.pos]

        probs  = prios ** self.prob_alpha
        probs /= probs.sum()

        indices = np.random.choice(len(self.buffer), batch_size, p=probs)
        samples = [self.buffer[idx] for idx in indices]

        total    = len(self.buffer)
        weights  = (total * probs[indices]) ** (-beta)
        weights /= weights.max()
        weights  = np.array(weights, dtype=np.float32)

        batch = zip(*samples)
        states, next_states = np.concatenate(batch[0]), np.concatenate(batch[3])
        actions, rewards, done = batch[1], batch[2], batch[4]
        return states, actions, rewards, next_states, dones, indices, weights

    def update_priorities(self, batch_indices, batch_priorities):
        for idx, prio in zip(batch_indices, batch_priorities):
            self.priorities[idx] = prio

    def __len__(self):
        return len(self.buffer)
```