A Tutorial on Deep-Q-Learning

Robert Tjarko Lange
robert.lange17@ic.ac.uk

July 2019

RobertTLange

Robert Tjarko Lange

@RobertTLange

roberttlange.github.io

@Sprekelerlab

@ECNBerlin

@SCIoI

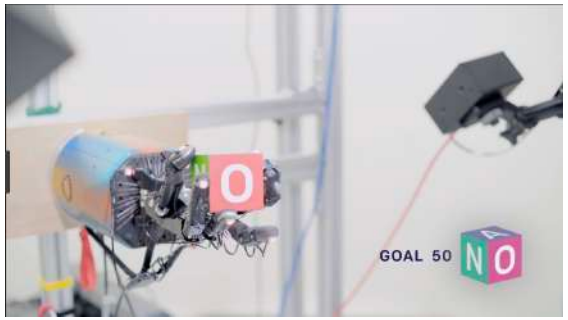
The Success Story.

DM - Atari DQN (2013, 2015)

DM - AlphaGo (2016, 2017)

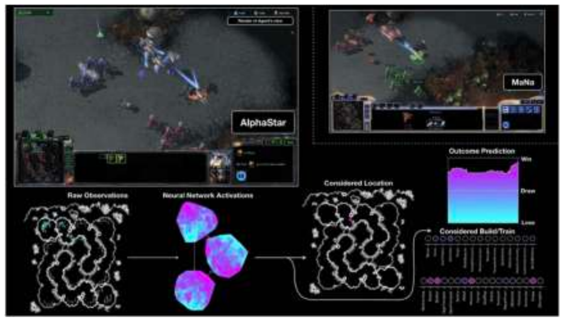
DM - AlphaZero (2018)

OpenAI - Dexterity (2018)

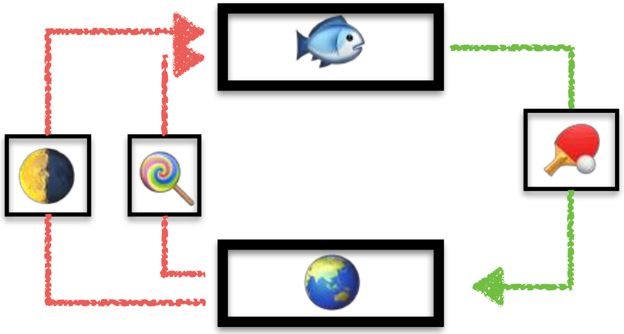


OpenAI - Five (Dota 2 - 2018)

DM - AlphaStar (StarCraft II -2019)

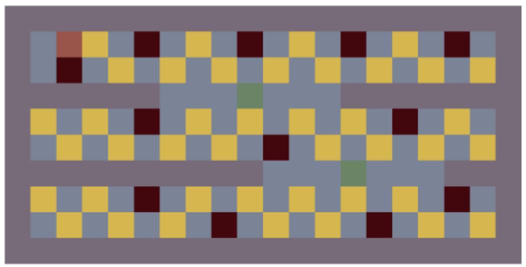


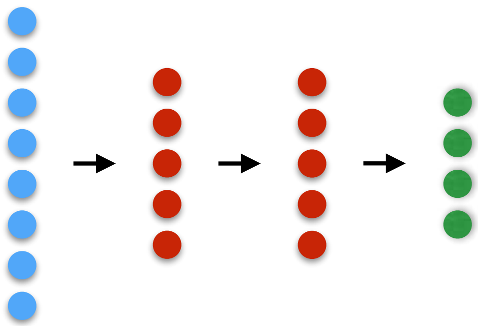
A Roadmap for Today.

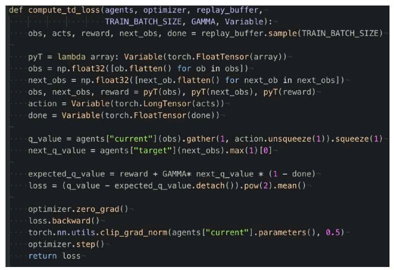


- MDP Formalism
- The RL Problem
- MF & Value-Based

Fitted Q-L.	DQN	Double DQN	PER	Dueling DQN
1996	2013	2015	2016	2016



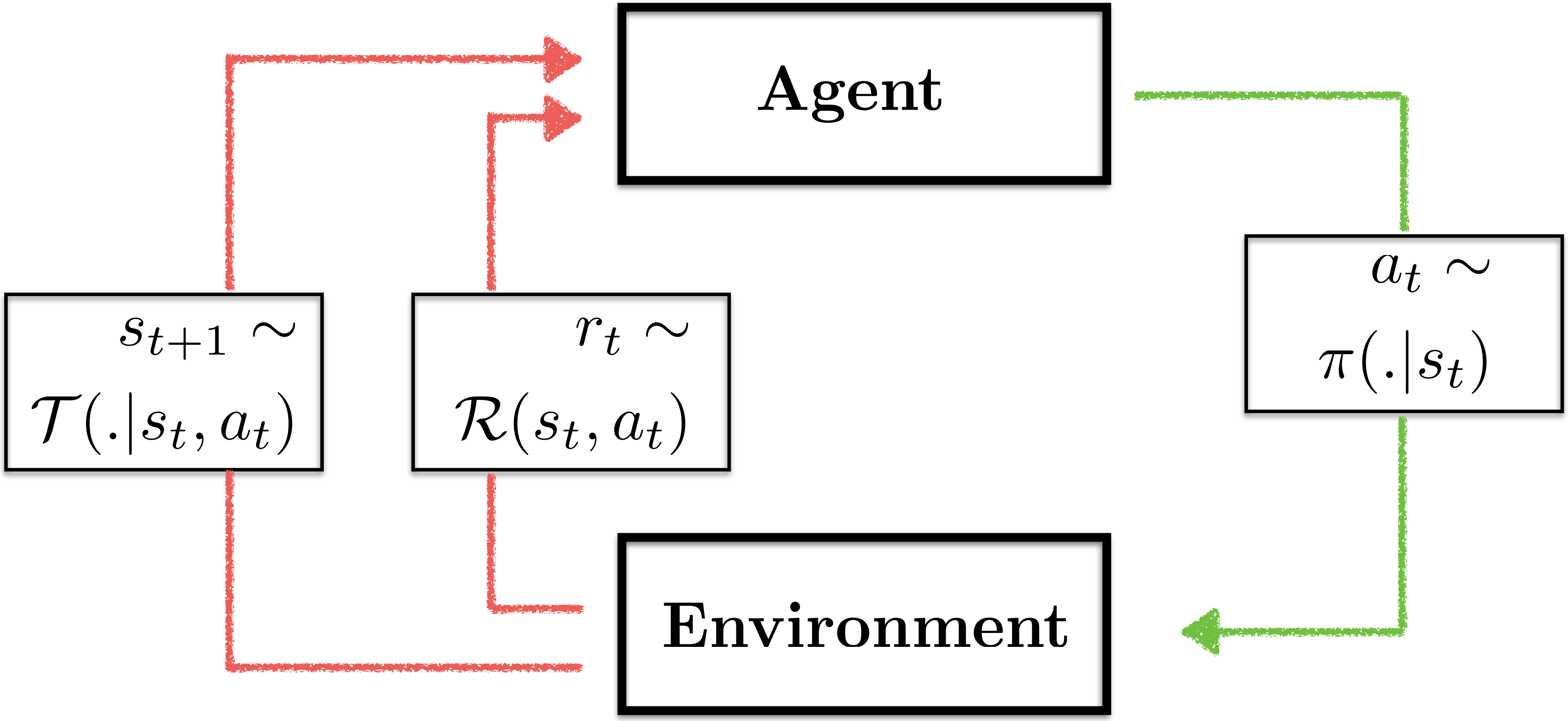




```python
def compute_td_loss(agents, optimizer, replay_buffer,
                    TRAIN_BATCH_SIZE, GAMMA, Variable):
    obs, acts, reward, next_obs, done = replay_buffer.sample(TRAIN_BATCH_SIZE)

    pyT = lambda array: Variable(torch.FloatTensor(array))
    obs = np.float32([ob.flatten() for ob in obs])
    next_obs = np.float32([next_ob.flatten() for next_ob in next_obs])
    obs, next_obs, reward = pyT(obs), pyT(next_obs), pyT(reward)
    action = Variable(torch.LongTensor(acts))
    done = Variable(torch.FloatTensor(done))

    q_value = agents["current"](obs).gather(1, action.unsqueeze(1)).squeeze(1)
    next_q_value = agents["target"](next_obs).max(1)[0]

    expected_q_value = reward + GAMMA* next_q_value * (1 - done)
    loss = (q_value - expected_q_value.detach()).pow(2).mean()

    optimizer.zero_grad()
    loss.backward()
    torch.nn.utils.clip_grad_norm(agents["current"].parameters(), 0.5)
    optimizer.step()
    return loss
```

The Action Perception Loop of RL.

‚A transition is the atomic unit of interaction in RL' - Schaul et al. (2016)



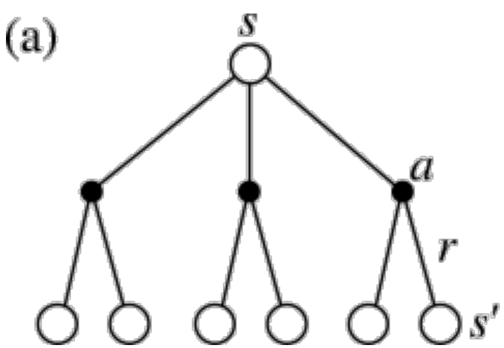
- MDP + Deterministic Policy: $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma)$ & $\pi(s) : \mathcal{S} \rightarrow \mathcal{A}$

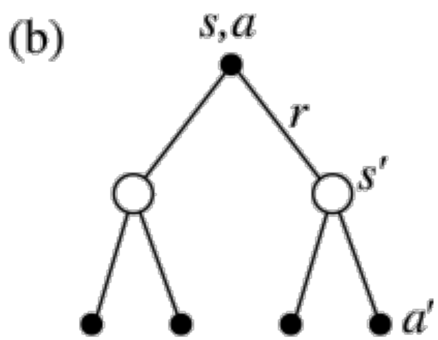
The Reinforcement Learning Problem.

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

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

$$
\begin{aligned}
Q^\pi(s,a) &= \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t = s, a_t = a, \pi] \\
&= \sum_{s' \in \mathcal{S}} T(s,a,s')[R(s,a,s') + \gamma Q^\pi(s', a = \pi(s'))]
\end{aligned}
$$





A Zoo of Different Approaches.

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

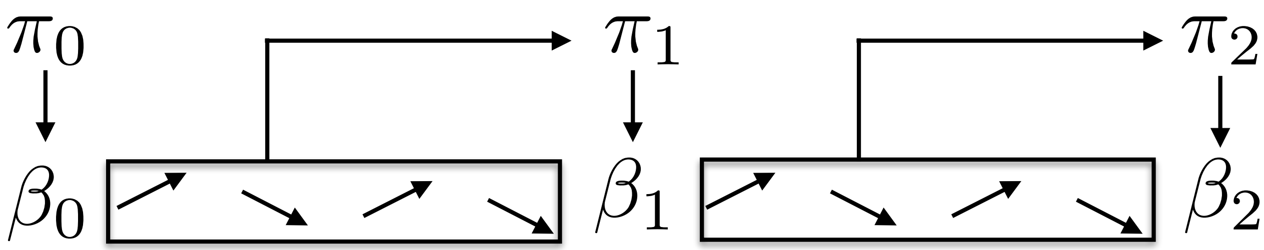
- Bootstrapping/TD-Learning: $< s, a, r, s' >$

Reward Prediction/TD Error: δ

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

Target: Y_k

- „Off-Policy“ Exploration:

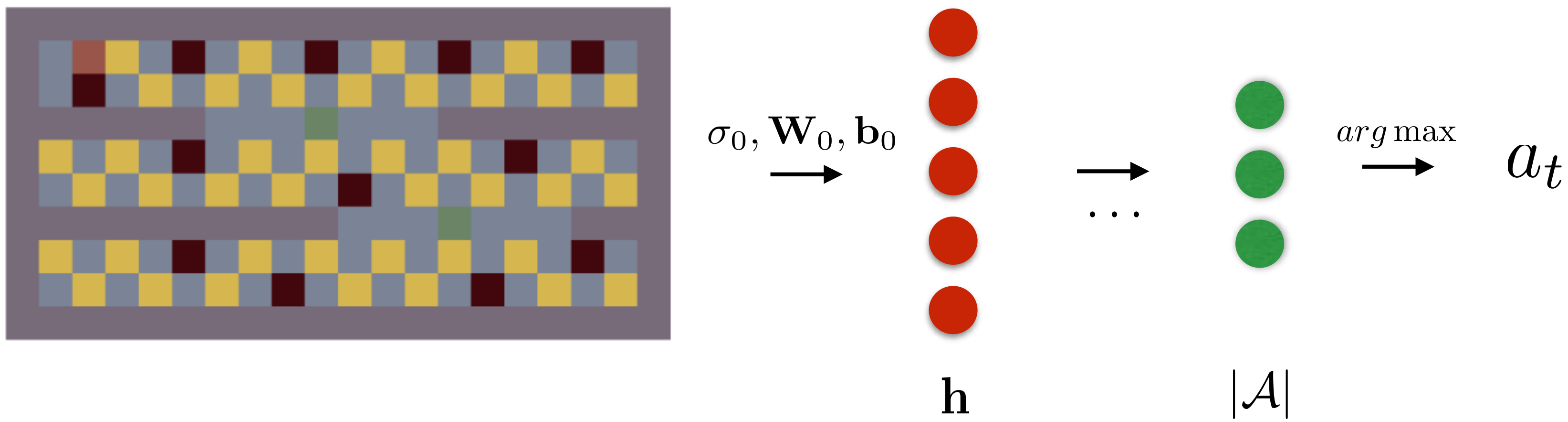


Overcoming the Curse of Dimensionality.

$\boxed{\text{Problem: What do we do if state space is too large!?}}$

Idea: Combat via generalization by function approximation

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



Fitted Q-Learning - Gordon (1996).

- Regression Problem - Mean Squared Bellman/TD Error:

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

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

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

1	2	3
No convergence guarantees	Wasteful Online Updates	Weight updates change targets
Meh.	Store & reuse past transitions.	Slowly changing target network.

DQNs - Mnih et al (2013, 2015).

- Target Networks: $Y_k = r + \gamma \max_{a' \in \mathcal{A}} Q(s', a'; \theta_k^-)$
  - Update every C iterations: $k \mod C = 0 : \theta_k^- \leftarrow \theta_k$
  - Polyak Averaging: $\theta_k^- \leftarrow \tau \theta_k + (1 - \tau)\theta_{k-1}^-$

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

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

$$\begin{matrix} < s_1, a_1, r_1, s_2 > \\ \vdots \\ < s_t, a_t, r_t, s_{t+1} > \\ \vdots \end{matrix} \qquad \begin{matrix} < s_1, a_1, r_1, s_2 > \\ \vdots \\ < s_t, a_t, r_t, s_{t+1} > \\ \vdots \\ < s_N, a_N, r_N, s_{N+1} > \end{matrix} \qquad \begin{matrix} < s_2, a_2, r_2, s_3 > \\ \vdots \\ < s_t, a_t, r_t, s_{t+1} > \\ \vdots \\ < s_{N+1}, a_{N+1}, r_{N+1}, s_{N+2} > \end{matrix}$$

$< s_1, a_1, r_1, s_2 >$

DQNs in 120 Lines of Code (60/120).

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

    def forward(self, x):
        return self.layers(x)

    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
```

```python
class ReplayBuffer(object):
    def __init__(self, capacity, record_macros=False):
        self.buffer = deque(maxlen=capacity)

    def push(self, ep_id, step, state, action,
             reward, next_state, done):
        self.buffer.append((ep_id, step, state, action, reward, next_state, done))

    def sample(self, batch_size):
        _, step, s, act, rew, next_s, done = zip(*random.sample(self.buffer,
                                                                 batch_size))
        return np.stack(s), act, rew, np.stack(next_s), done

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

```python
def init_dqn(model, L_RATE, USE_CUDA, INPUT_DIM, HIDDEN_SIZE, NUM_ACTIONS):
    agents = {"current": model(INPUT_DIM, HIDDEN_SIZE, NUM_ACTIONS),
              "target": model(INPUT_DIM, HIDDEN_SIZE, NUM_ACTIONS)}
    optimizers = optim.Adam(params=agents["current"].parameters(), lr=L_RATE)
    return agents, optimizers

def update_target(current_model, target_model):
    target_model.load_state_dict(current_model.state_dict())

def polyak_update_target(current_model, target_model, soft_tau):
    for target_param, current_param in zip(target_model.parameters(),
                                           current_model.parameters()):
        target_param.data.copy_(
            target_param.data * (1. - soft_tau) + current_param.data * soft_tau
        )

def epsilon_by_episode(eps_id, epsilon_start, epsilon_final, epsilon_decay):
    eps = (epsilon_final + (epsilon_start - epsilon_final)
           * math.exp(-1. * eps_id / epsilon_decay))
    return eps
```


```python
def init_dqn(model, L_RATE, USE_CUDA, INPUT_DIM, HIDDEN_SIZE, NUM_ACTIONS):
    agents = {"current": model(INPUT_DIM, HIDDEN_SIZE, NUM_ACTIONS),
              "target": model(INPUT_DIM, HIDDEN_SIZE, NUM_ACTIONS)}
    optimizers = optim.Adam(params=agents["current"].parameters(), lr=L_RATE)
    return agents, optimizers

def update_target(current_model, target_model):
    target_model.load_state_dict(current_model.state_dict())

def polyak_update_target(current_model, target_model, soft_tau):
    for target_param, current_param in zip(target_model.parameters(),
                                           current_model.parameters()):
        target_param.data.copy_(
            target_param.data * (1. - soft_tau) + current_param.data * soft_tau
        )

def epsilon_by_episode(eps_id, epsilon_start, epsilon_final, epsilon_decay):
    eps = (epsilon_final + (epsilon_start - epsilon_final)
           * math.exp(-1. * eps_id / epsilon_decay))
    return eps
```

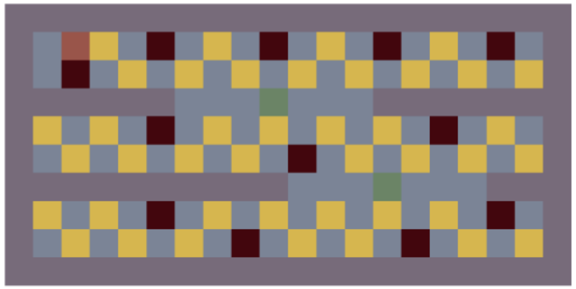
DQNs in 120 Lines of Code (120/120).

```python
def run_dqn_learning(args):
    agents, optimizer = init_dqn(MLP_DQN, args.L_RATE, USE_CUDA,
                                 args.INPUT_DIM, args.HIDDEN_SIZE, args.NUM_ACTIONS)
    replay_buffer = ReplayBuffer(capacity=args.CAPACITY)

    opt_counter = 0
    env = gym.make("dense-v0")
    ep_id = 0

    while opt_counter < args.NUM_UPDATES:
        epsilon = epsilon_by_episode(ep_id + 1, args.EPS_START, args.EPS_STOP,
                                     args.EPS_DECAY)
        obs = env.reset()
        steps = 0
        while steps < args.MAX_STEPS:
            action = agents["current"].act(obs.flatten(), epsilon)
            next_obs, rew, done, _  = env.step(action)
            steps += 1

            # Push transition to ER Buffer
            replay_buffer.push(ep_id, steps, obs, action,
                               rew, next_obs, done)

            if len(replay_buffer) > args.TRAIN_BATCH_SIZE:
                opt_counter += 1
                loss = compute_td_loss(agents, optimizer, replay_buffer,
                                       args.TRAIN_BATCH_SIZE, args.GAMMA, Variable)

            # Go to next episode if current one terminated or update obs
            if done: break
            else: obs = next_obs

            if (opt_counter+1) % args.UPDATE_EVERY == 0:
                update_target(agents["current"], agents["target"])
        ep_id += 1
    return
```

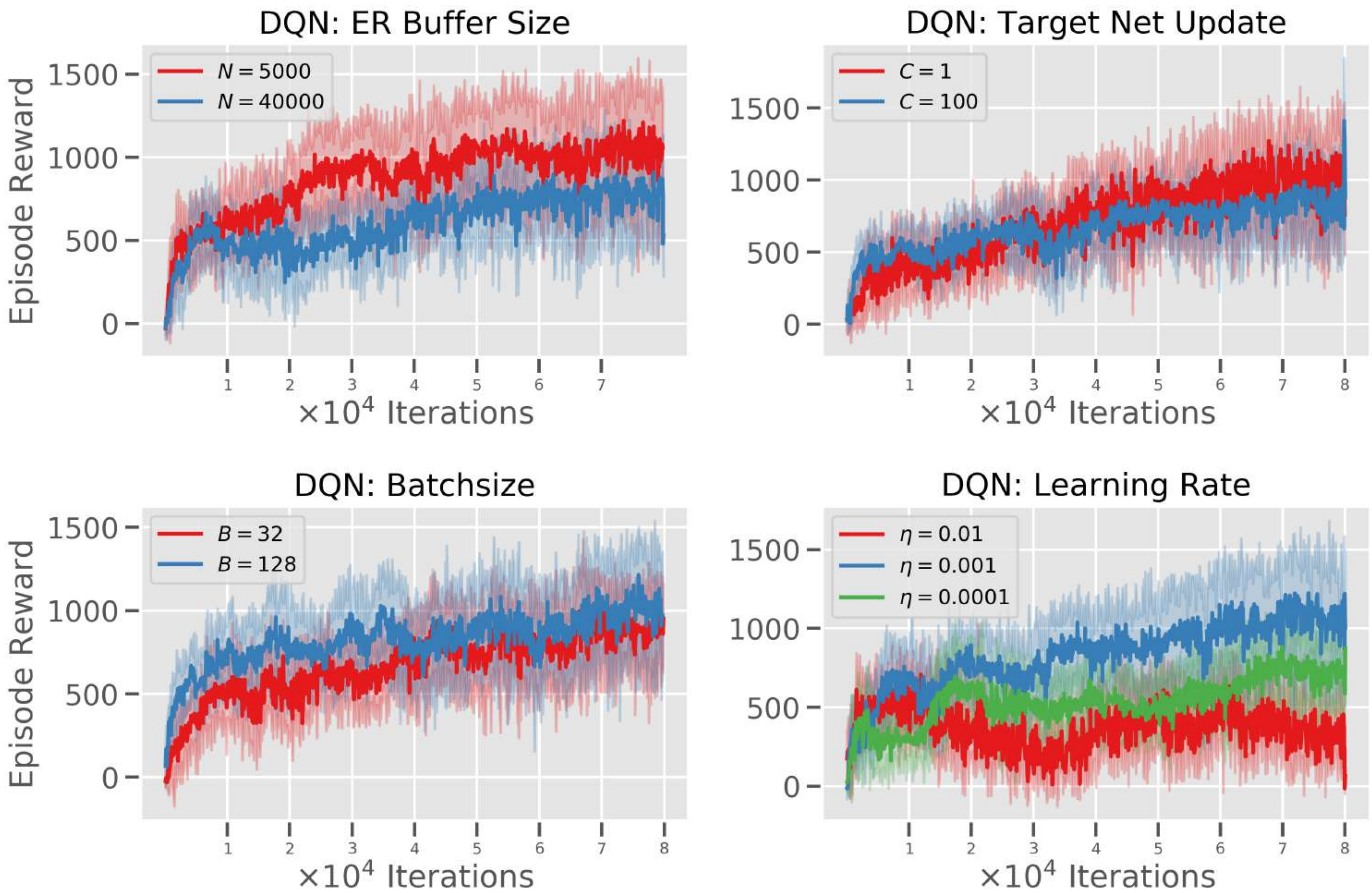
```python
def compute_td_loss(agents, optimizer, replay_buffer,
                    TRAIN_BATCH_SIZE, GAMMA, Variable):
    obs, acts, reward, next_obs, done = replay_buffer.sample(TRAIN_BATCH_SIZE)

    pyT = lambda array: Variable(torch.FloatTensor(array))
    obs = np.float32([ob.flatten() for ob in obs])
    next_obs = np.float32([next_ob.flatten() for next_ob in next_obs])
    obs, next_obs, reward = pyT(obs), pyT(next_obs), pyT(reward)
    action = Variable(torch.LongTensor(acts))
    done = Variable(torch.FloatTensor(done))

    q_value = agents["current"](obs).gather(1, action.unsqueeze(1)).squeeze(1)
    next_q_value = agents["target"](next_obs).max(1)[0]

    expected_q_value = reward + GAMMA* next_q_value * (1 - done)
    loss = (q_value - expected_q_value.detach()).pow(2).mean()

    optimizer.zero_grad()
    loss.backward()
    torch.nn.utils.clip_grad_norm(agents["current"].parameters(), 0.5)
    optimizer.step()
    return loss
```


```python
def compute_td_loss(agents, optimizer, replay_buffer,
                    TRAIN_BATCH_SIZE, GAMMA, Variable):
    obs, acts, reward, next_obs, done = replay_buffer.sample(TRAIN_BATCH_SIZE)

    pyT = lambda array: Variable(torch.FloatTensor(array))
    obs = np.float32([ob.flatten() for ob in obs])
    next_obs = np.float32([next_ob.flatten() for next_ob in next_obs])
    obs, next_obs, reward = pyT(obs), pyT(next_obs), pyT(reward)
    action = Variable(torch.LongTensor(acts))
    done = Variable(torch.FloatTensor(done))

    q_value = agents["current"](obs).gather(1, action.unsqueeze(1)).squeeze(1)
    next_q_value = agents["target"](next_obs).max(1)[0]

    expected_q_value = reward + GAMMA* next_q_value * (1 - done)
    loss = (q_value - expected_q_value.detach()).pow(2).mean()

    optimizer.zero_grad()
    loss.backward()
    torch.nn.utils.clip_grad_norm(agents["current"].parameters(), 0.5)
    optimizer.step()
    return loss
```

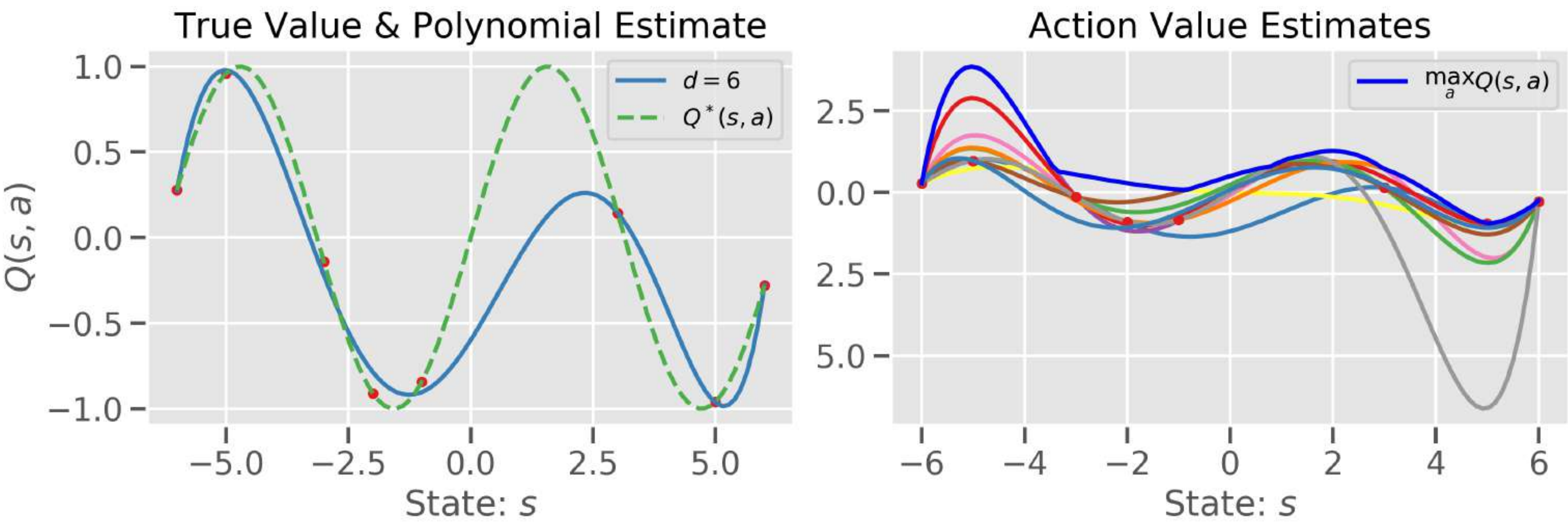

- Back to our toy gridworld:



Some DQN Hyperparameter Intuition.



Overestimation Bias in Q-Learning.



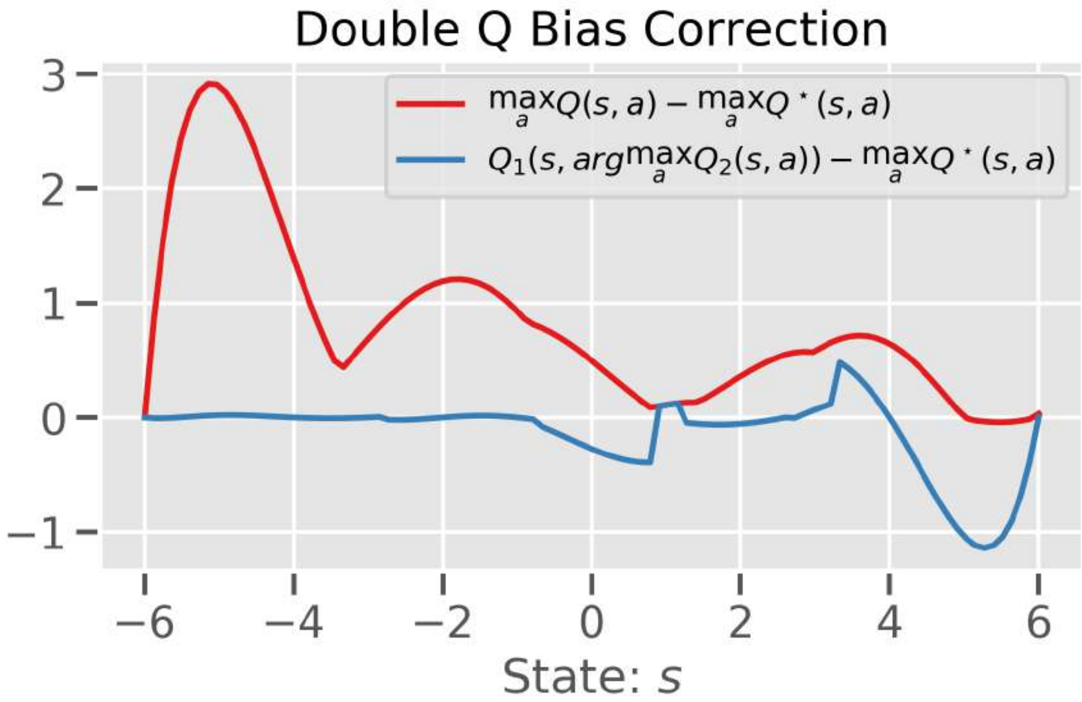
Reason: Maximization prefers overestimated values!

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

- 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

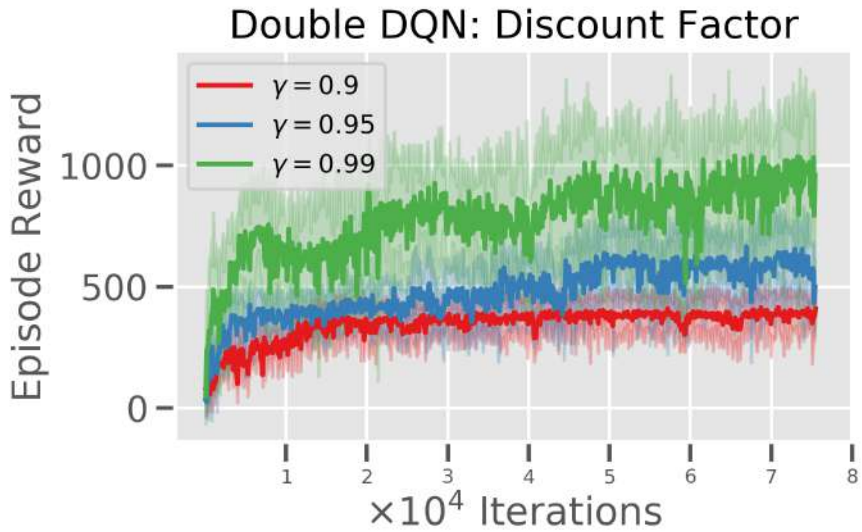
$$Y_k^{DDQN} = r + \gamma Q(s', arg \max_{a \in \mathcal{A}} Q(s', a; \theta_k); \theta_k^-)$$

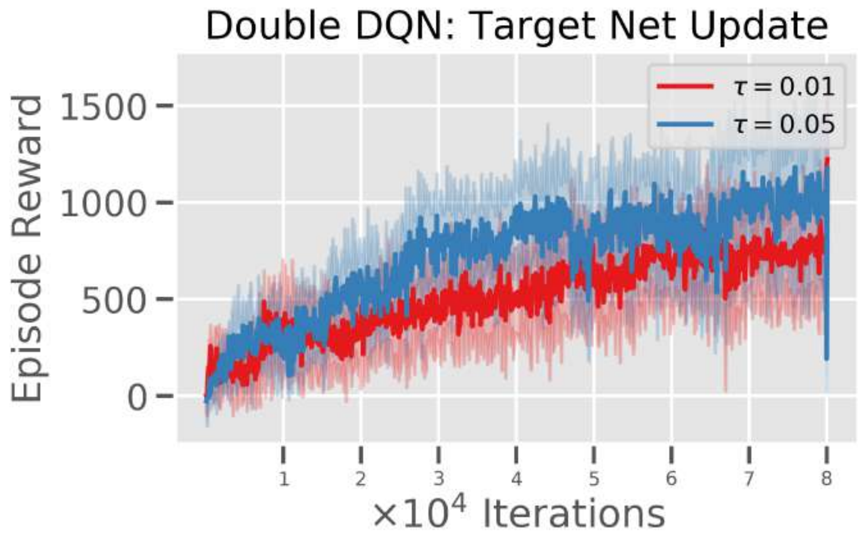
So what do we have to change in our DQN code?

```
q_value = agents["current"](obs).gather(1, action.unsqueeze(1)).squeeze(1)
online_next_q_values = agents["current"](next_obs)
online_action = torch.max(next_q_values, 1)[1].unsqueeze(1)
next_q_value = agents["target"](obs).gather(1, online_action).squeeze(1)
```

new_obs

```
def compute_td_loss(agents, optimizer, replay_buffer,
                    TRAIN_BATCH_SIZE, GAMMA, Variable):
    obs, acts, reward, next_obs, done = replay_buffer.sample(TRAIN_BATCH_SIZE)

    pyT = lambda array: Variable(torch.FloatTensor(array))
    obs = np.float32([ob.flatten() for ob in obs])
    next_obs = np.float32([next_ob.flatten() for next_ob in next_obs])
    obs, next_obs, reward = pyT(obs), pyT(next_obs), pyT(reward)
    action = Variable(torch.LongTensor(acts))
    done = Variable(torch.FloatTensor(done))

    q_value = agents["current"](obs).gather(1, action.unsqueeze(1)).squeeze(1)
    next_q_value = agents["target"](next_obs).max(1)[0]

    expected_q_value = reward + GAMMA* next_q_value * (1 - done)
    loss = (q_value - expected_q_value.detach()).pow(2).mean()

    optimizer.zero_grad()
    loss.backward()
    torch.nn.utils.clip_grad_norm(agents["current"].parameters(), 0.5)
    optimizer.step()
    return loss
```





Prioritized Experience Replay - Schaul et al (2016).

Memory Replay = Computation + Storage

$\downarrow$ ~~Sample Uniform~~

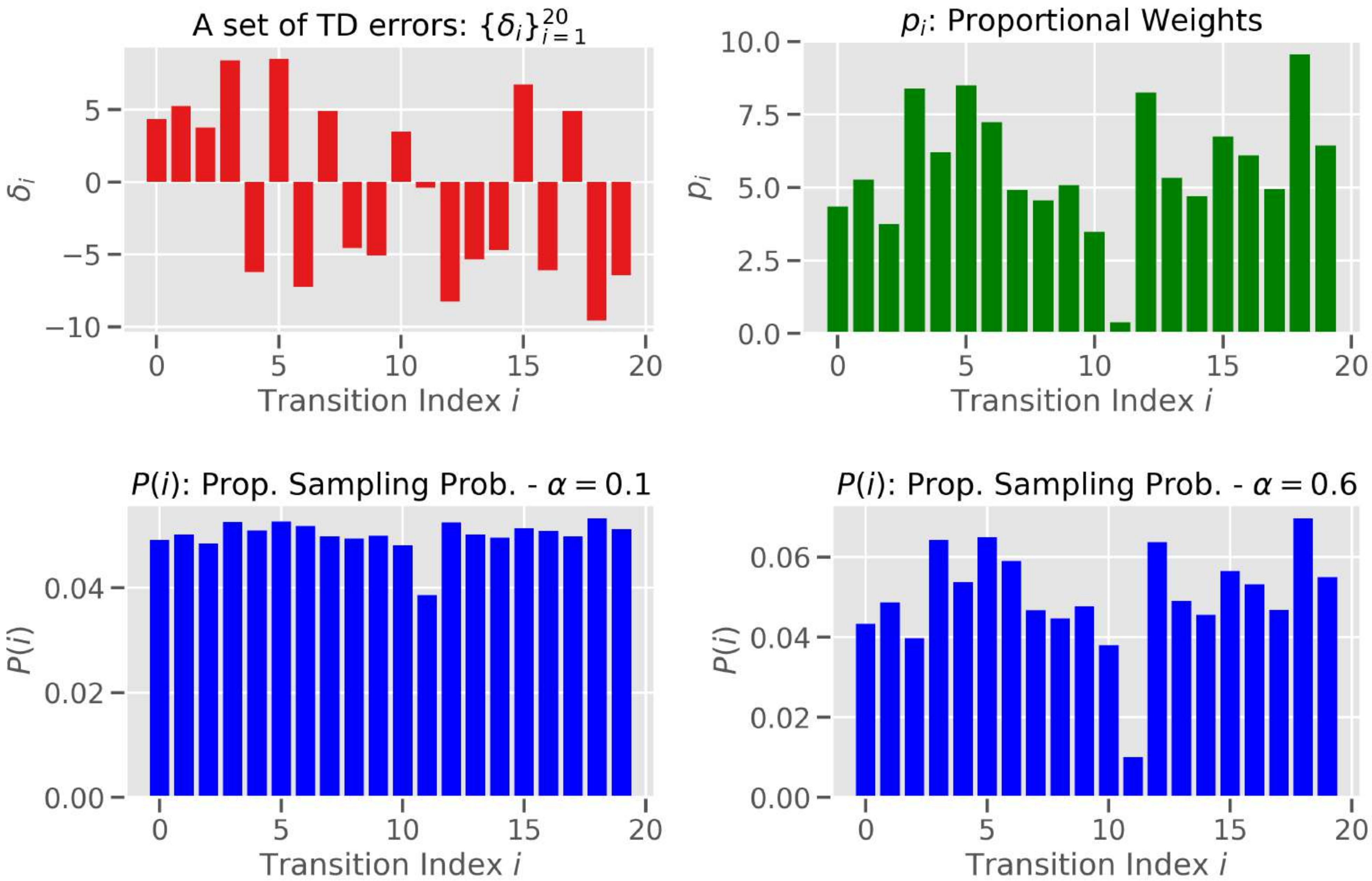
$\downarrow$ Sliding Window

$$< s_1, a_1, r_1, s_2 >$$
$$\vdots$$
$$< s_t, a_t, r_t, s_{t+1} >$$
$$\vdots$$
$$< s_N, a_N, r_N, s_{N+1} >$$

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

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

Prioritized Experience Replay - Schaul et al (2016).



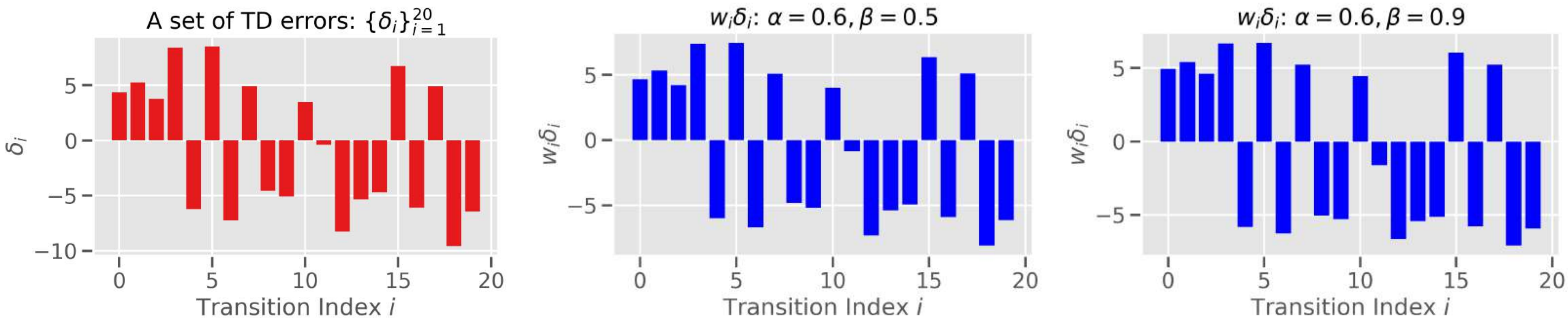
Prioritized Experience Replay - Schaul et al (2016).

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

- Solution: Importance sampling!
- Starting from 0.4 linearly anneal β to 1.

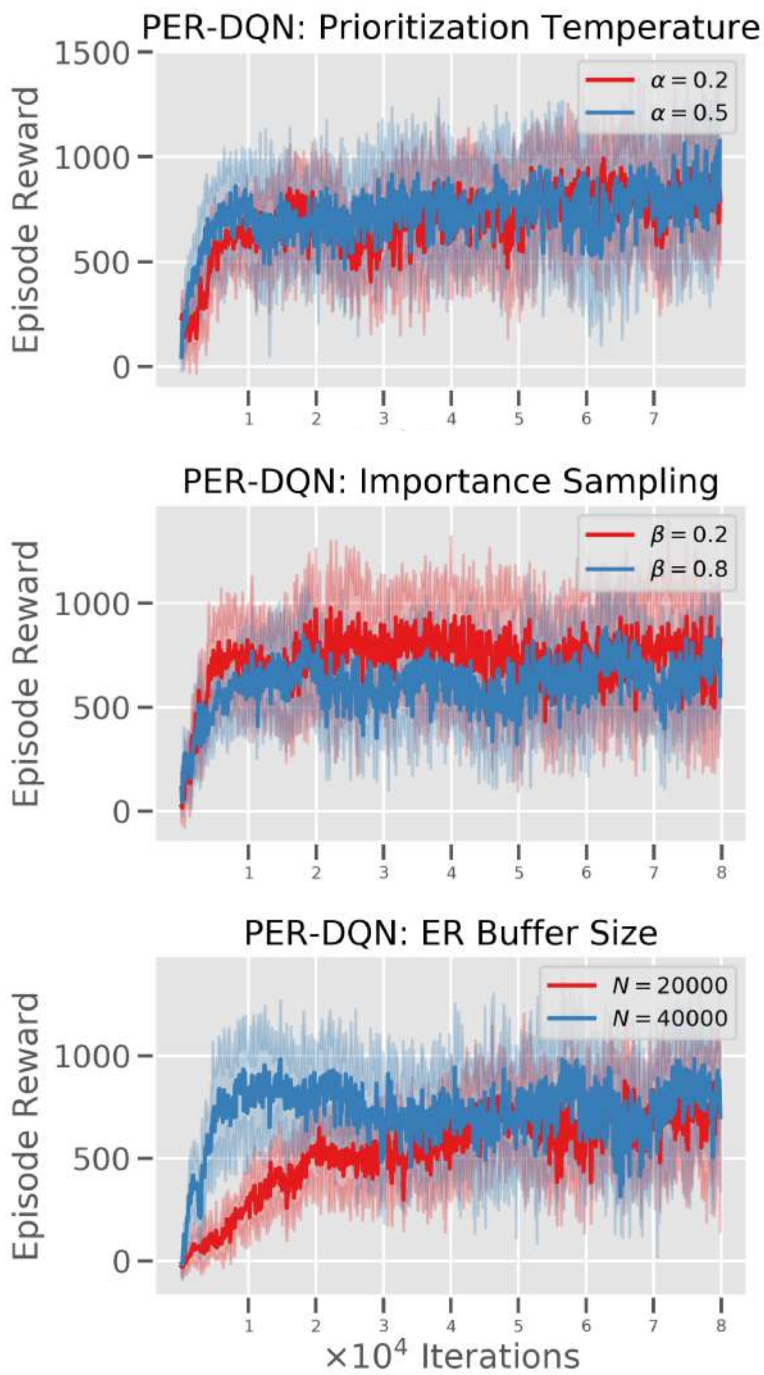
$$\boxed{w_i = \left(\frac{1}{N}\frac{1}{P(i)}\right)^{\beta}}$$

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



Prioritized Experience Replay - Schaul et al (2016).

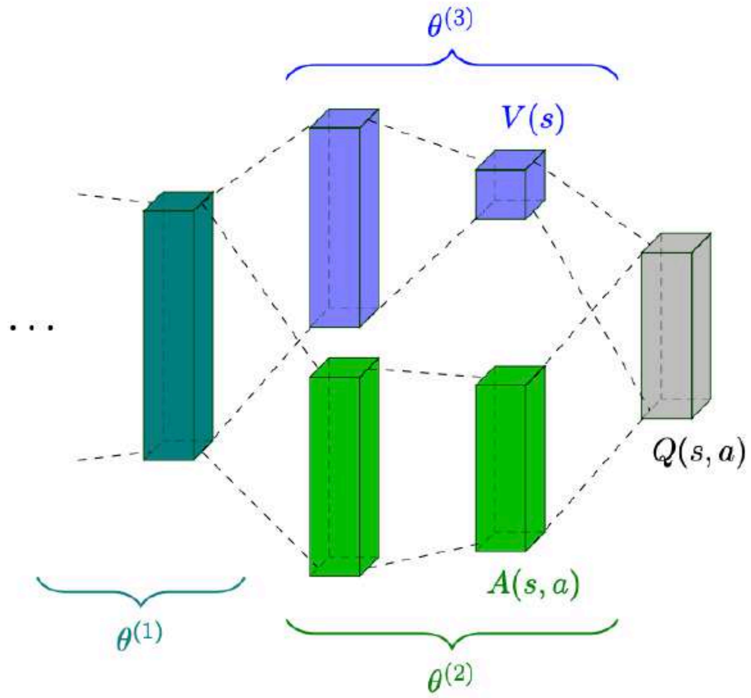
```python
class NaivePrioritizedBuffer(object):
    def __init__(self, capacity, prob_alpha=0.6):
        self.prob_alpha = prob_alpha
        self.capacity   = capacity
        self.buffer     = []
        self.pos        = 0
        self.priorities = np.zeros((capacity,), dtype=np.float32)

    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)
```



Splitting Action Values in State Value & Advantage

- Problem: Some times action doesn't affect env!

$$A(s,a)^{\pi} = Q(s,a)^{\pi} - V^{\pi}(s)$$



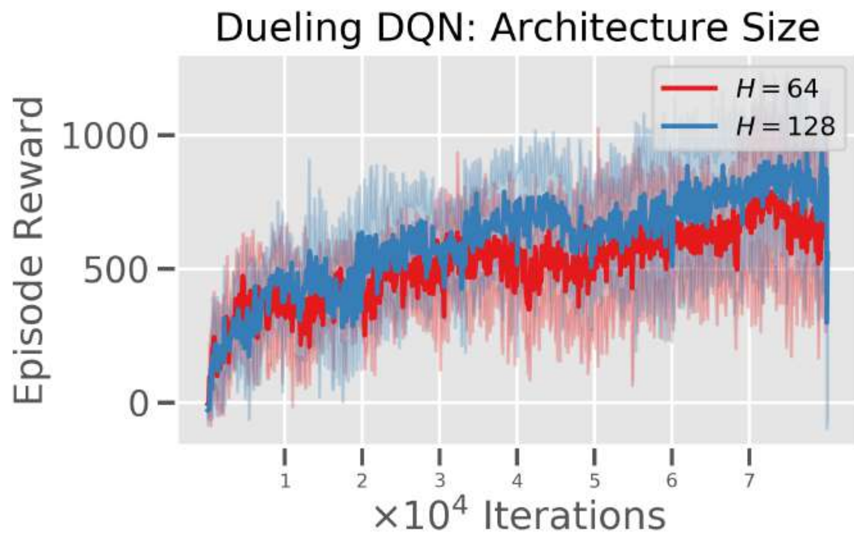
- Solution: Split streams!
- Net can now learn which states are valuable regardless of actions!

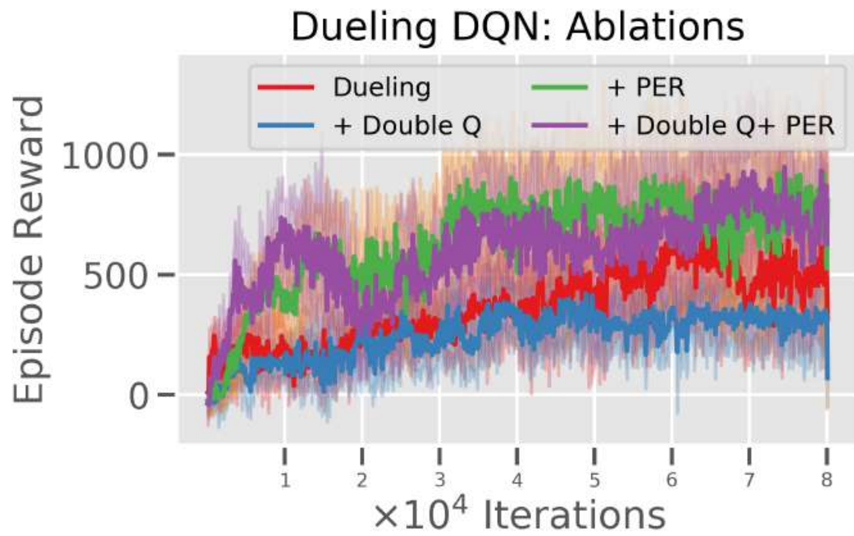
Easy architectural change!
No effort whatsoever

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

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

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

Double DQN

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

PER

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

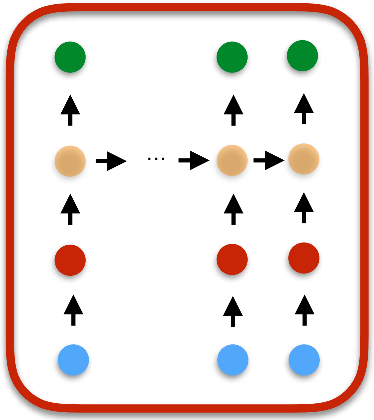
Dueling DQN

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

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

A Zoo of Extensions.

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

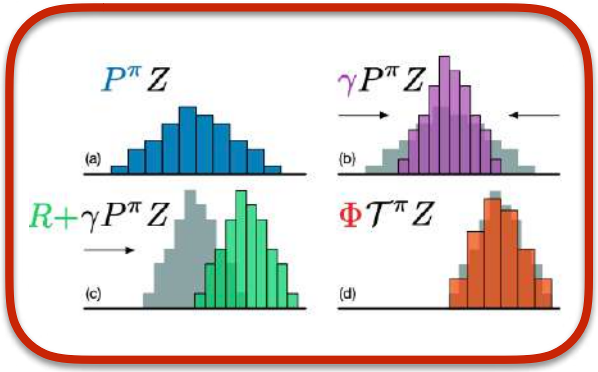


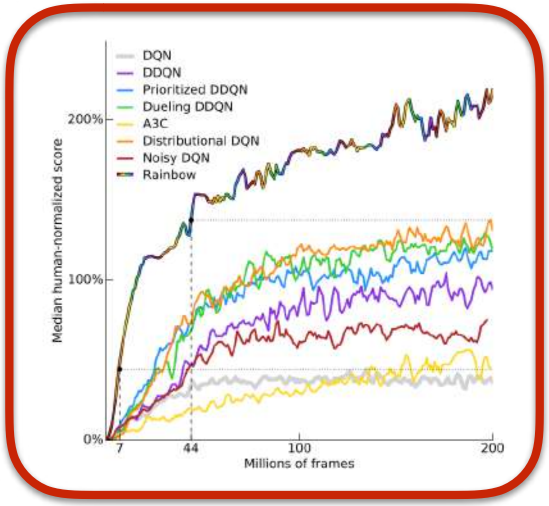
$$y = (b + Wx) + (\tilde{b} \odot \epsilon^b + (\tilde{W} \odot \epsilon^W)x)$$

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.

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.

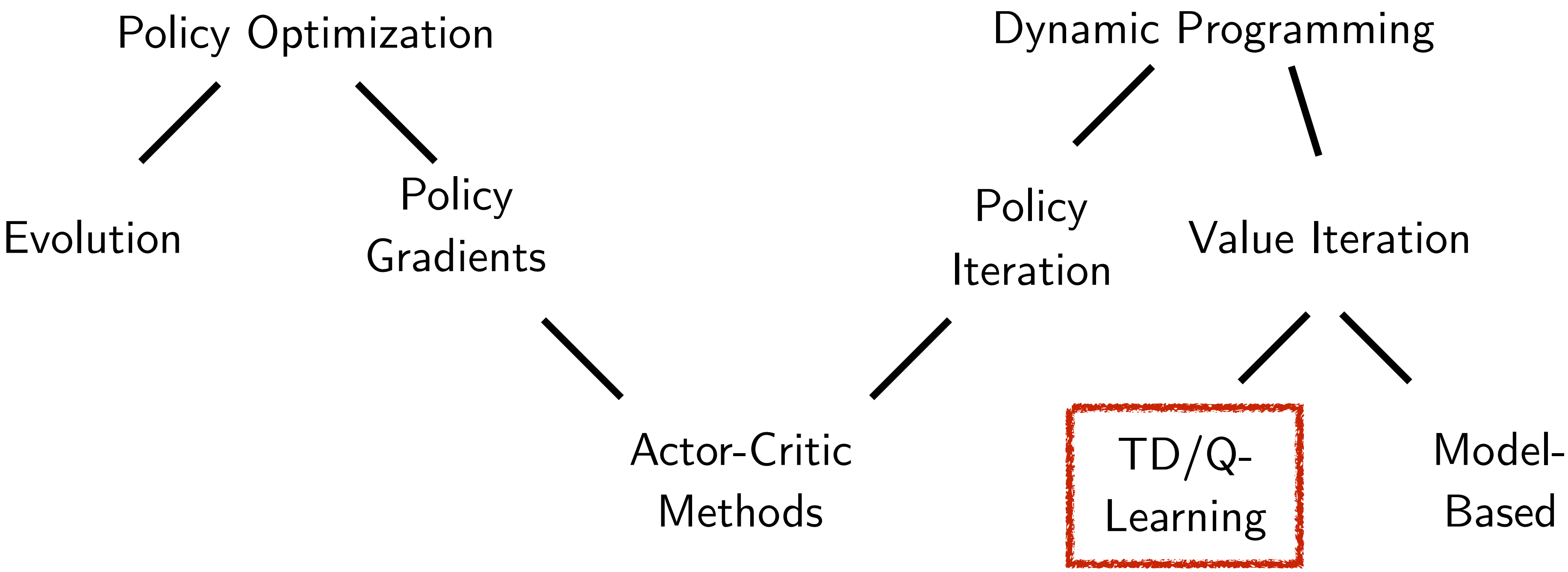
Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015). Prioritized experience replay. arXiv preprint arXiv: 1511.05952.

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

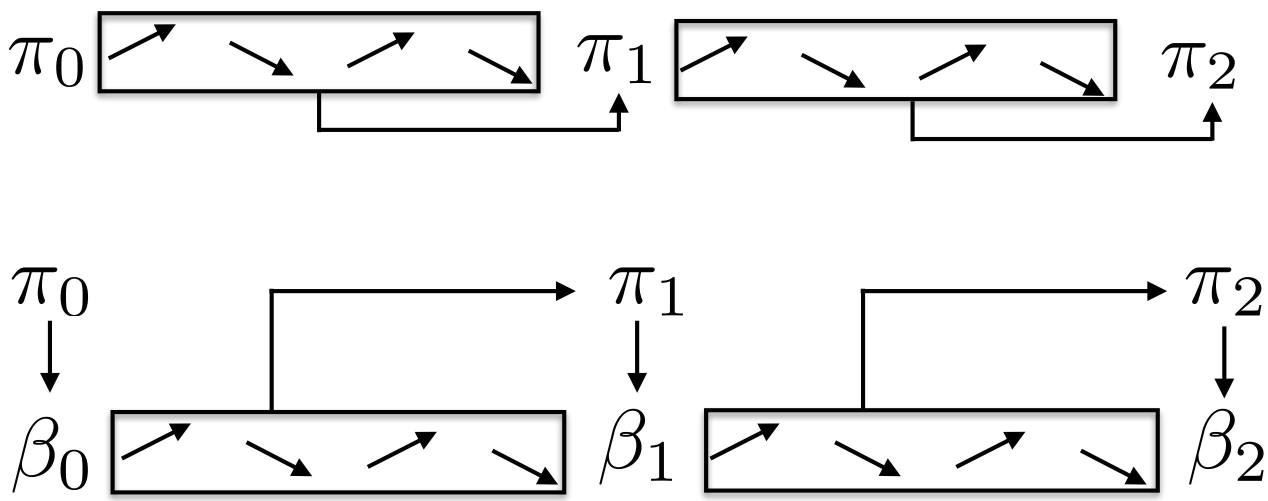
Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., & De Freitas, N. (2015). Dueling network architectures for deep reinforcement learning. arXiv preprint arXiv:1511.06581.

Supplementary Material

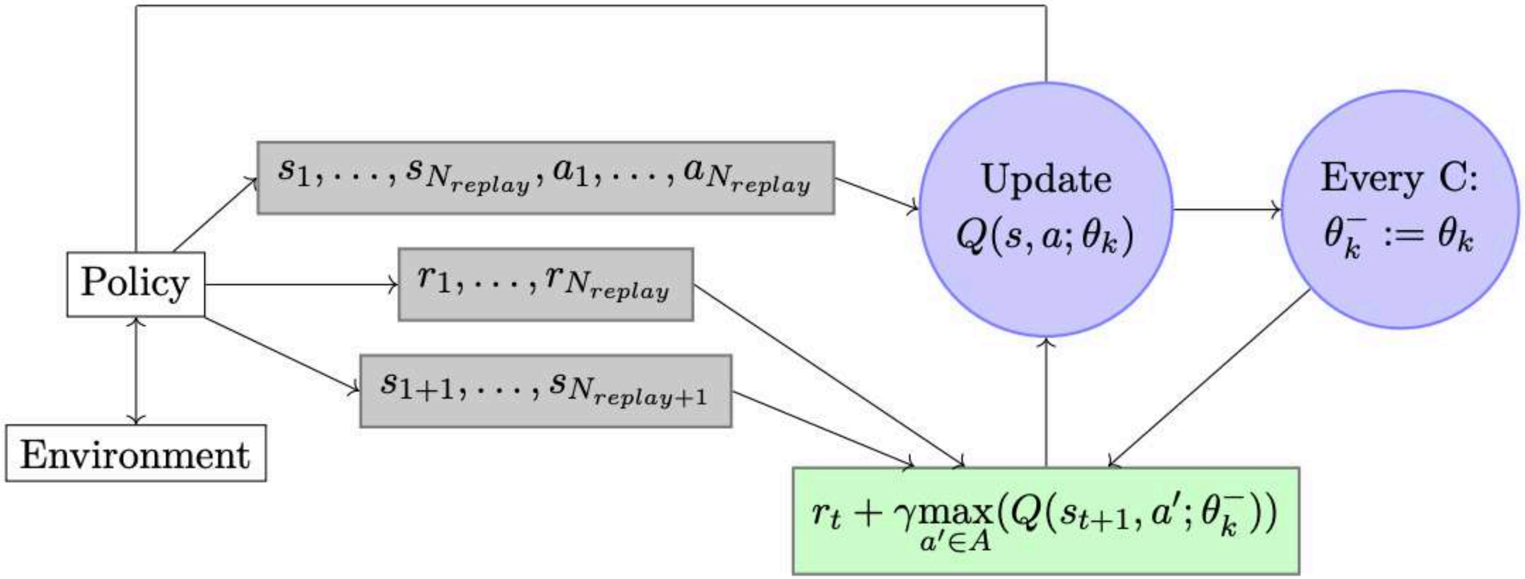
A Zoo of Different Approaches.



- „On-Policy“:
- „Off-Policy“:



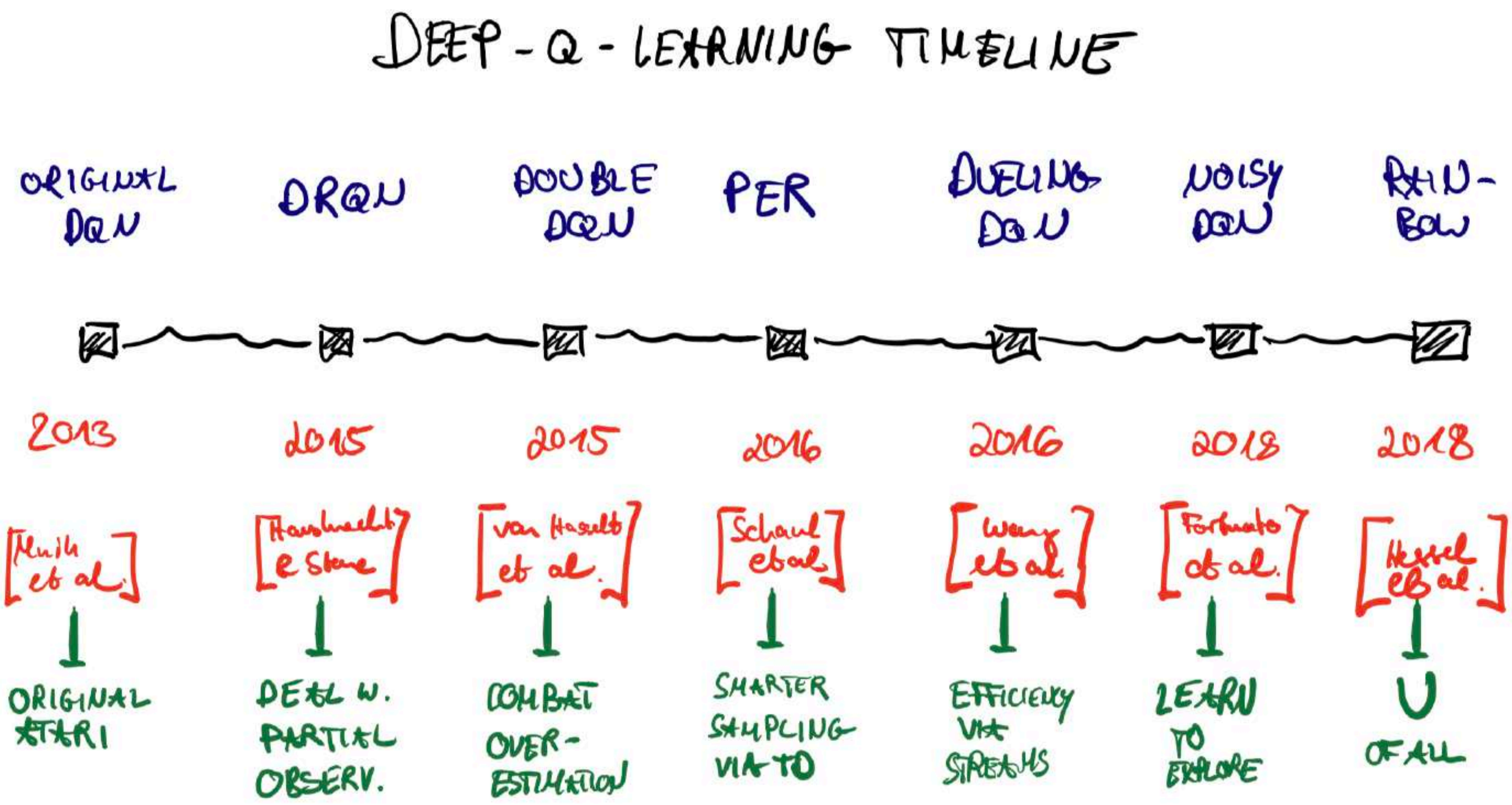
ATARI DQN - Mnih et al (2013, 2015).



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
RAIN-BOW
2013
2015
2015
2016
2016
2018
2018
[Mnih et al.]
[Hausknecht & Stone]
[van Hasselt et al.]
[Schaul et al.]
[Wang et al.]
[Fortunato et al.]
[Hessel et al.]
ORIGINAL ATARI
DEAL W. PARTIAL OBSERV.
COMBAT OVER-ESTIMATION
SMARTER SAMPLING VIA TD
EFFICIENCY VIA STREAMS
LEARN TO EXPLORE
U OF ALL