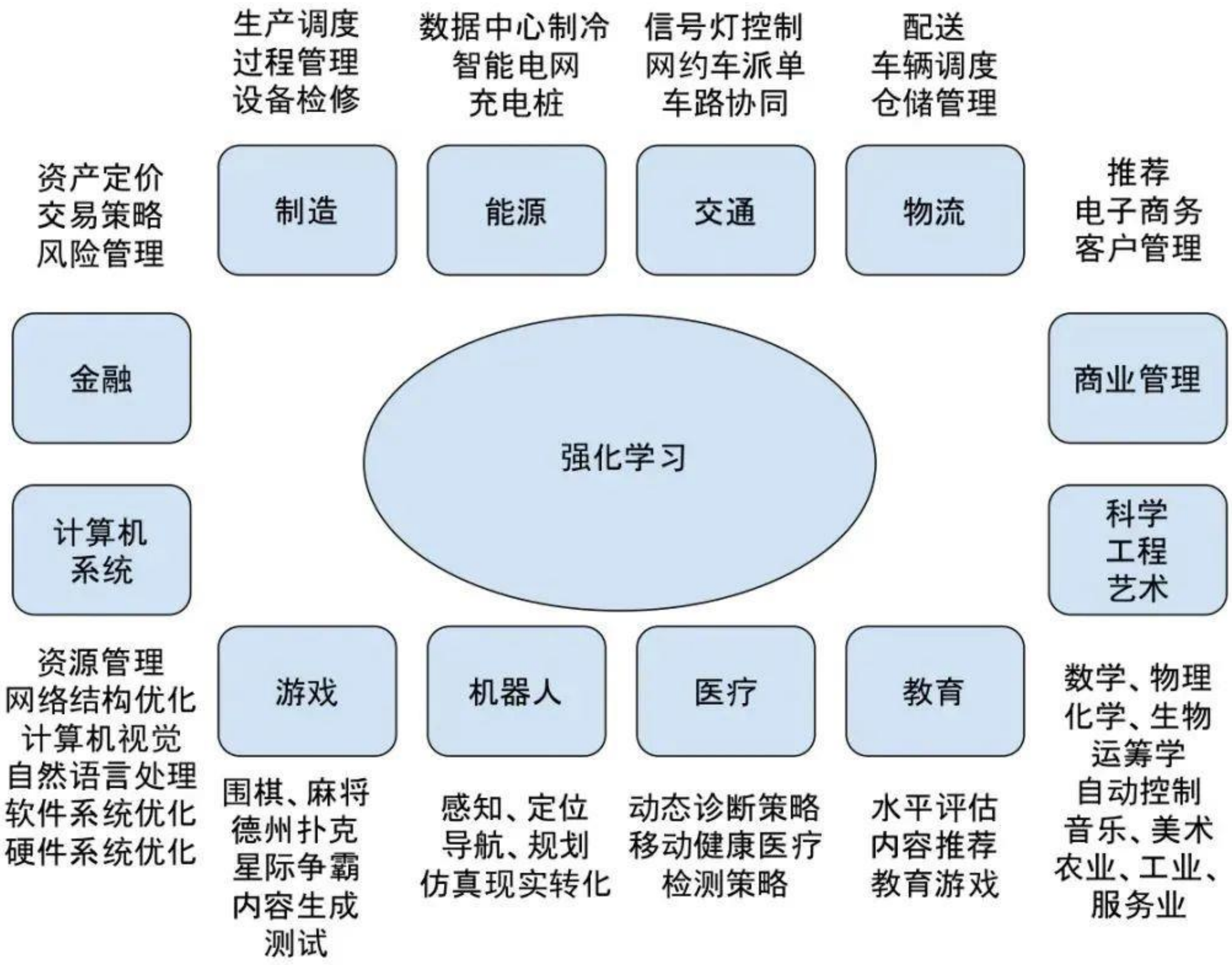
第七章 强化学习V—深度强化学习

Chao Yu （余超）

School of Computer Science and Engineering
Sun Yat-Sen University

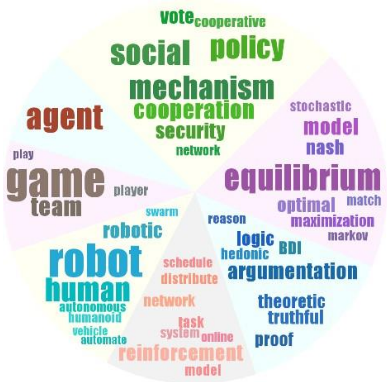
[based on David Silver and Sergey Levine's course]

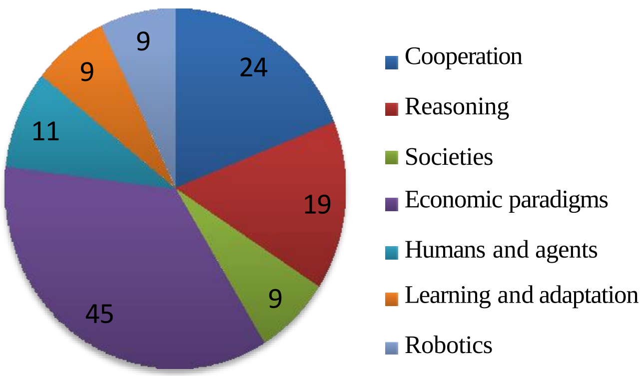
Application of Deep RL

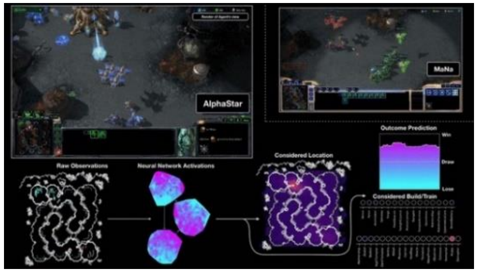


Application of Deep RL

- Game: Go、Texas Hold’em Poker、Honor of Kings、Dota、StarCraft、Atari、Football, etc.
- UAV(Unmanned Aerial Vehicle)
- Autonomous Driving
- Traffic flow control
- Finance
- Medicine
- Robotics

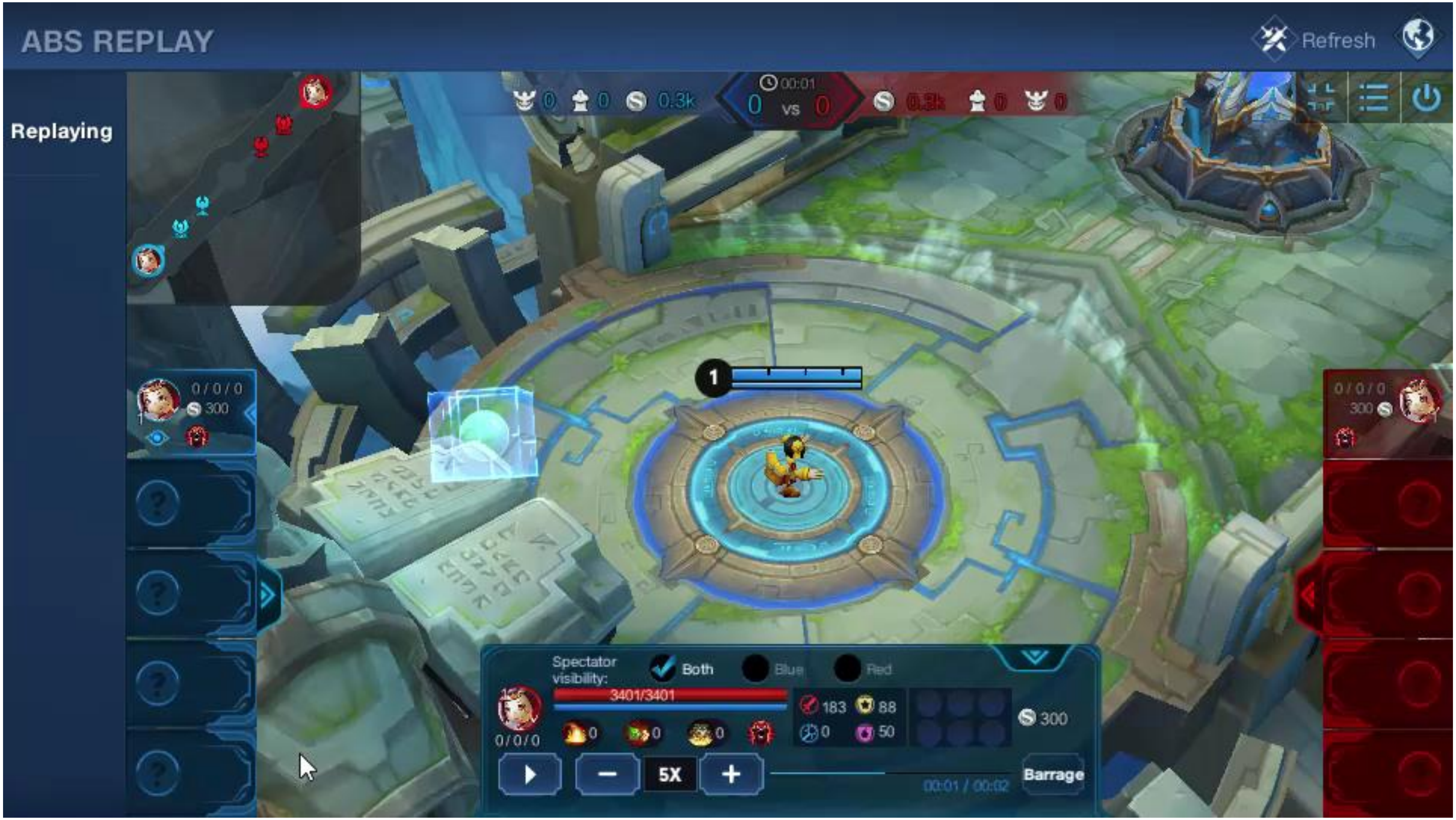






Application of Deep RL

- Honor of Kings



Value Function Approximation

- So far we have represented value function by a *lookup table*
  - Every state s has an entry $V(s)$
  - Or every state-action pair s, a has an entry $Q(s, a)$
- Problem with large MDPs:
  - There are too many states and/or actions to store in memory
  - It is too slow to learn the value of each state individually
- Solution for large MDPs:
  - Estimate value function with *function approximation*

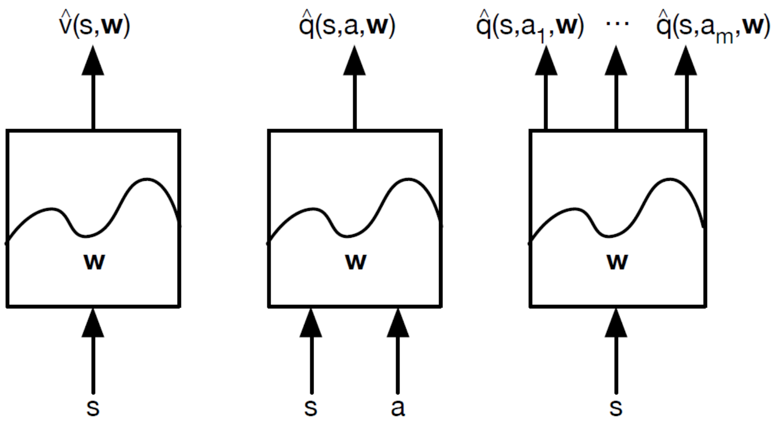
$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$$
$$\text{or } \hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$$

  - *Generalise* from seen states to unseen states
  - *Update* parameter $\mathbf{w}$ using MC or TD learning

Large-Scale Reinforcement Learning

- Reinforcement learning can be used to solve large problems, e.g.
  - Backgammon: 10^{20} states
  - Computer Go: 10^{170} states
  - Helicopter: continuous state space

Types of Value Function Approximation



There are many function approximators, e.g.

- Linear combinations of features
- Neural network
- Decision tree
- Nearest neighbour
- Fourier / wavelet bases
- ...

Value Function Approx. By Stochastic Gradient Descent

- Goal: find parameter vector $\mathbf{w}$ minimising mean-squared error between approximate value fn $\hat{v}(s, \mathbf{w})$ and true value fn $v_\pi(s)$

$$J(\mathbf{w}) = \mathbb{E}_\pi \left[(v_\pi(S) - \hat{v}(S, \mathbf{w}))^2 \right]$$

- Gradient descent finds a local minimum

$$\begin{aligned} \Delta \mathbf{w} &= -\frac{1}{2}\alpha \nabla_{\mathbf{w}} J(\mathbf{w}) \\ &= \alpha \mathbb{E}_\pi \left[(v_\pi(S) - \hat{v}(S, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w}) \right] \end{aligned}$$

- Stochastic gradient descent *samples* the gradient

$$\Delta \mathbf{w} = \alpha (v_\pi(S) - \hat{v}(S, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w})$$

- Expected update is equal to full gradient update

Linear Value Function Approximation

- Represent value function by a linear combination of features

$$\hat{v}(S, \mathbf{w}) = \mathbf{x}(S)^\top \mathbf{w} = \sum_{j=1}^{n} \mathbf{x}_j(S)\mathbf{w}_j$$

- Objective function is quadratic in parameters $\mathbf{w}$

$$J(\mathbf{w}) = \mathbb{E}_\pi\left[(v_\pi(S) - \mathbf{x}(S)^\top \mathbf{w})^2\right]$$

- Stochastic gradient descent converges on *global* optimum
- Update rule is particularly simple

$$\nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w}) = \mathbf{x}(S)$$
$$\Delta \mathbf{w} = \alpha(v_\pi(S) - \hat{v}(S, \mathbf{w}))\mathbf{x}(S)$$

Update = *step-size* × *prediction error* × *feature value*

TD Learning with Value Function Approximation

- The TD-target $R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$ is a *biased* sample of true value $v_\pi(S_t)$
- Can still apply supervised learning to "training data":

$$\langle S_1, R_2 + \gamma \hat{v}(S_2, \mathbf{w})\rangle, \langle S_2, R_3 + \gamma \hat{v}(S_3, \mathbf{w})\rangle, ..., \langle S_{T-1}, R_T\rangle$$

- For example, using *linear TD(0)*

$$\begin{aligned}\Delta \mathbf{w} &= \alpha(R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w}))\nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w}) \\ &= \alpha \delta \mathbf{x}(S)\end{aligned}$$

- Linear TD(0) converges (close) to global optimum

Action-Value Function Approximation

- Approximate the action-value function

$$\hat{q}(S, A, \mathbf{w}) \approx q_\pi(S, A)$$

- Minimise mean-squared error between approximate action-value fn $\hat{q}(S, A, \mathbf{w})$ and true action-value fn $q_\pi(S, A)$

$$J(\mathbf{w}) = \mathbb{E}_\pi\left[(q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}))^2\right]$$

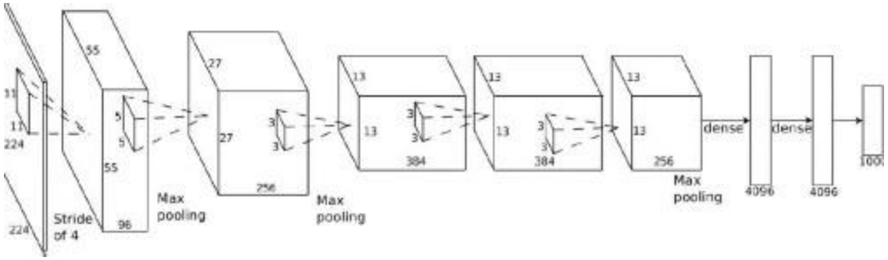
- Use stochastic gradient descent to find a local minimum

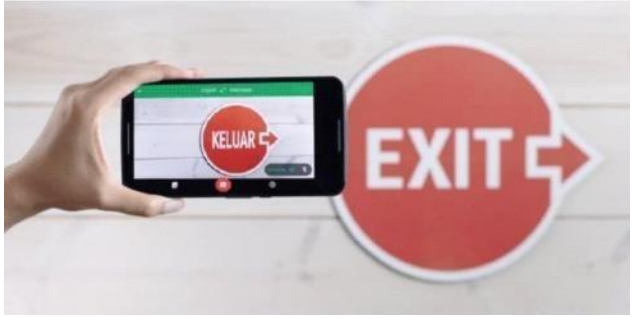
$$-\frac{1}{2}\nabla_{\mathbf{w}} J(\mathbf{w}) = (q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(S, A, \mathbf{w})$$

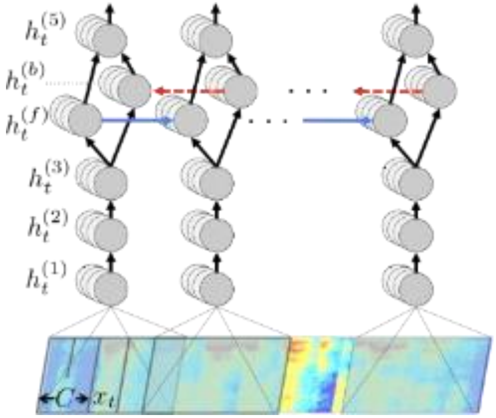
$$\Delta\mathbf{w} = \alpha(q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(S, A, \mathbf{w})$$

Deep Learning

- Deep Learning is part of a broader family of machine learning methods based on artificial neural networks with representation learning

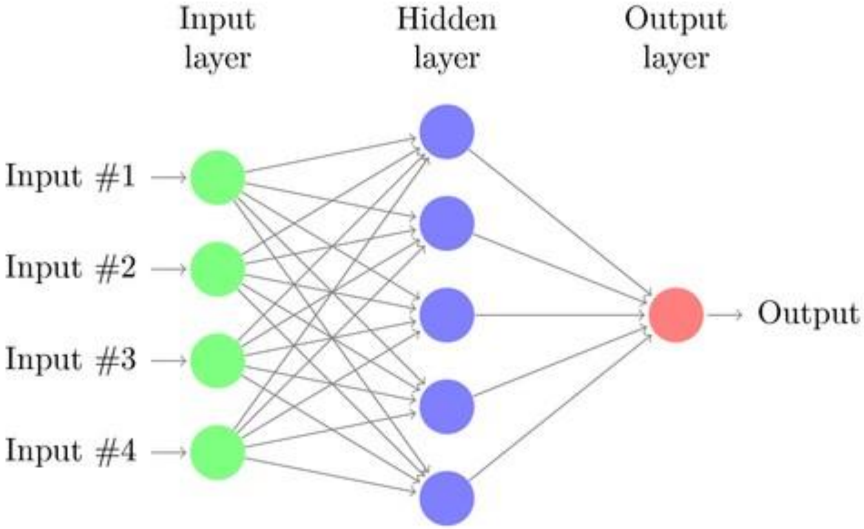






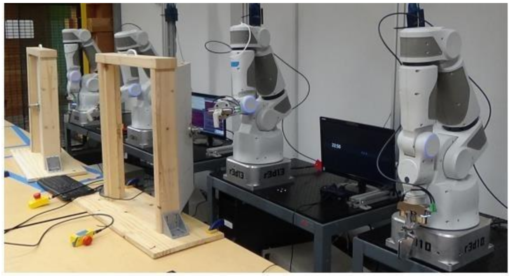
Deep Neural Networks(DNN)

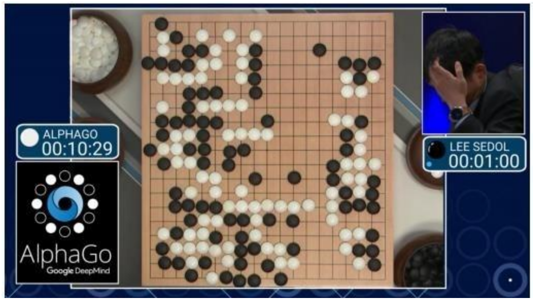
- Composition of multiple functions
- Can use the chain rule to backpropagate the gradient
- Generally combines both linear and non-linear transformations
- To fit the parameters, require a loss function(MSE, log likelihood, etc.)
- Major innovation: tools to automatically compute gradients for a DNN
- Deep Learning helps us handle unstructured environments



Deep Reinforcement Learning

- *What is deep RL, and why should we care?*
  - Deep models are what allow reinforcement learning algorithms to solve complex problems
    - Deep = can process complex sensory input
    - RL = can choose complex actions
  - Use deep neural networks to represent Value, Q function, Policy, Model





Atari games:

Q-learning:

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, et al. "Playing Atari with Deep Reinforcement Learning". (2013).

Policy gradients:

J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel. "Trust Region Policy Optimization". (2015).
V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, et al. "Asynchronous methods for deep reinforcement learning". (2016).

Real-world robots:

Guided policy search:

S. Levine*, C. Finn*, T. Darrell, P. Abbeel. "End-to-end training of deep visuomotor policies". (2015).

Q-learning:

D. Kalashnikov et al. "QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation". (2018).

Beating Go champions:

Supervised learning + policy gradients + value functions + Monte Carlo tree search:

D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, et al. "Mastering the game of Go with deep neural networks and tree search". Nature (2016).

Deep RL with Q-Functions

- Naïve deep Q-learning
  - Represent state-action value function by Q-network

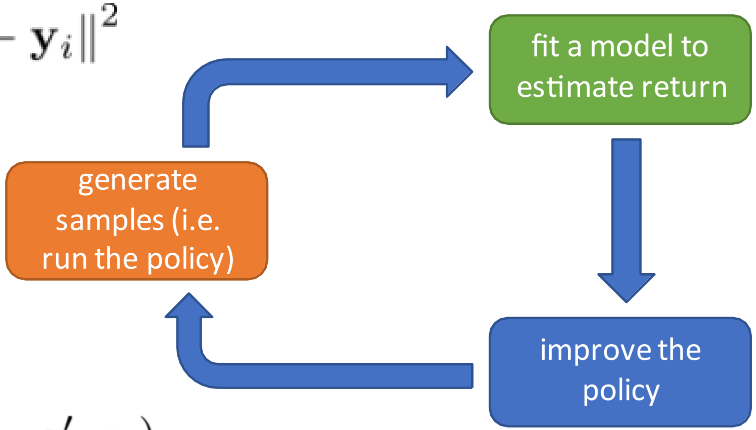
full fitted Q-iteration algorithm:

1. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy

$K\times$

2. set $\mathbf{y}_i \leftarrow r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'_i} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
3. set $\phi \leftarrow \arg\min_\phi \frac{1}{2} \sum_i \|Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i\|^2$

$Q_\phi(\mathbf{s}, \mathbf{a}) \leftarrow r(\mathbf{s}, \mathbf{a}) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}', \mathbf{a}')$



$\mathbf{a} = \arg\max_{\mathbf{a}} Q_\phi(\mathbf{s}, \mathbf{a})$

online Q iteration algorithm:

1. take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$
2. $\mathbf{y}_i = r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
3. $\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i)$

Deep RL with Q-Functions

- Two of the issues:
  - Correlations between samples - sequential states are strongly correlated
  - Non-stationary targets - target value is always changing

online Q iteration algorithm:

1. take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$
2. $\mathbf{y}_i = r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
3. $\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i)$

these are correlated!

isn't this just gradient descent? that converges, right?

Q-learning is *not* gradient descent!

$$\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)])$$

no gradient through target value

Deep RL with Q-Functions

Solution: replay buffers

full Q-learning with replay buffer:

+ samples are no longer correlated

1. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy, add it to $\mathcal{B}$

$K\times$

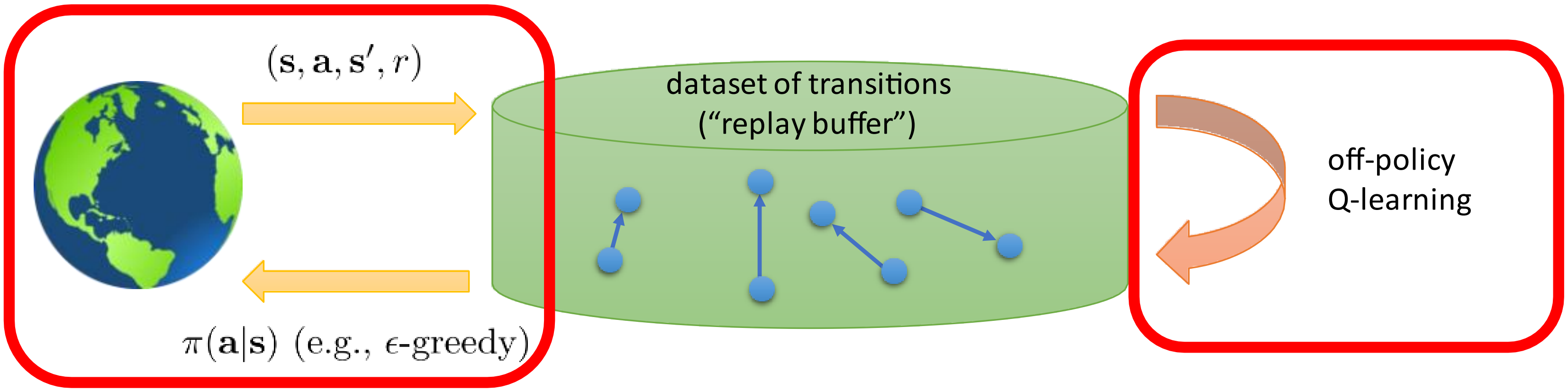
2. sample a batch $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$ from $\mathcal{B}$
3. $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)])$

+ multiple samples in the batch (low-variance gradient)

but where does the data come from?

need to periodically feed the replay buffer...

K = 1 is common, though larger K more efficient



Deep RL with Q-Functions

- Solution: Target Networks

Q-learning with replay buffer and target network:

$N\times$

1. save target network parameters: $\phi' \leftarrow \phi$
2. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy, add it to $\mathcal{B}$

$K\times$

3. sample a batch $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$ from $\mathcal{B}$
4. $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \underline{[r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_{\phi'}(\mathbf{s}'_i, \mathbf{a}'_i)]})$

targets don't change in inner loop!

supervised regression

Deep RL with Q-Functions

Deep Q-Network(DQN)

Q-learning with replay buffer and target network:

1. save target network parameters: $\phi' \leftarrow \phi$
2. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy, add it to $\mathcal{B}$
3. sample a batch $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$ from $\mathcal{B}$
4. $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_{\phi'}(\mathbf{s}'_i, \mathbf{a}'_i)])$

$N\times$ $K\times$

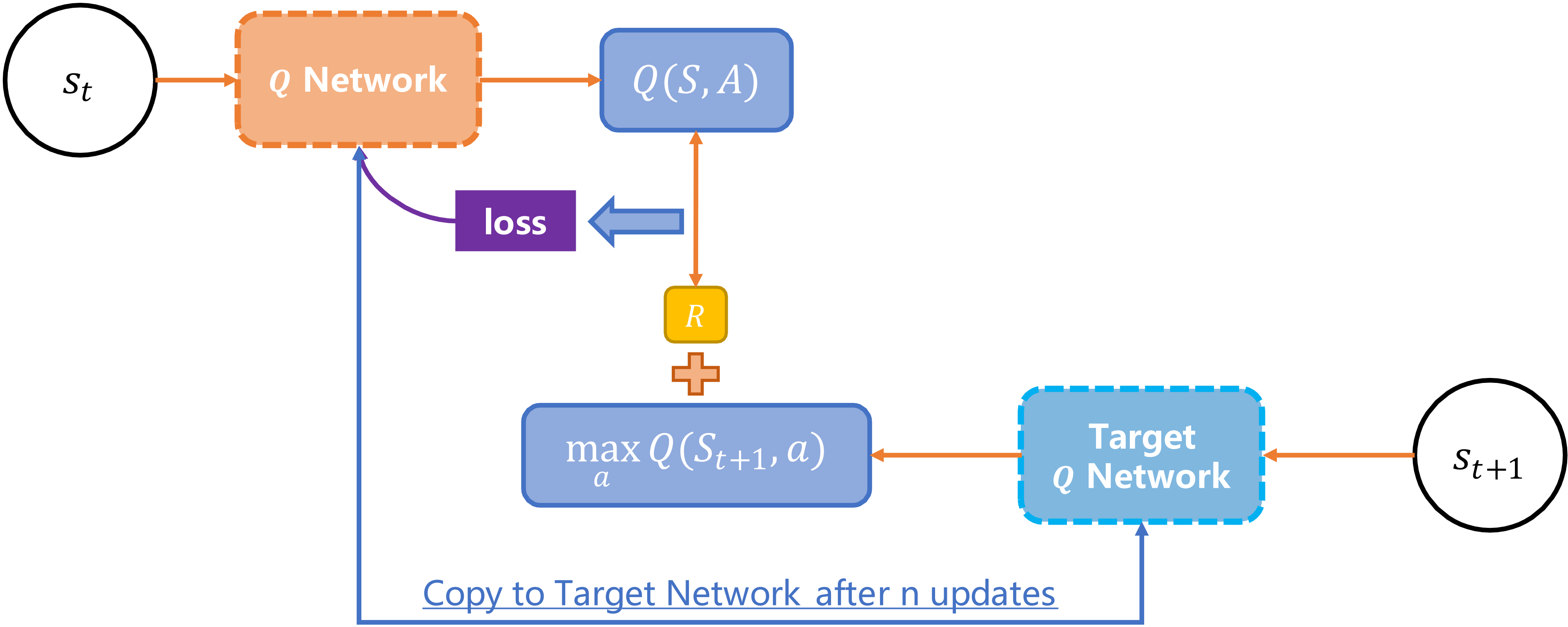
“classic” deep Q-learning algorithm:

1. take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$, add it to $\mathcal{B}$
2. sample mini-batch $\{\mathbf{s}_j, \mathbf{a}_j, \mathbf{s}'_j, r_j\}$ from $\mathcal{B}$ uniformly
3. compute $y_j = r_j + \gamma \max_{\mathbf{a}'_j} Q_{\phi'}(\mathbf{s}'_j, \mathbf{a}'_j)$ using *target* network $Q_{\phi'}$
4. $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}(\mathbf{s}_j, \mathbf{a}_j)(Q_\phi(\mathbf{s}_j, \mathbf{a}_j) - y_j)$
5. update ϕ': copy ϕ every N steps

$K = 1$

Deep RL with Q-Functions

Deep Q-Network(DQN)



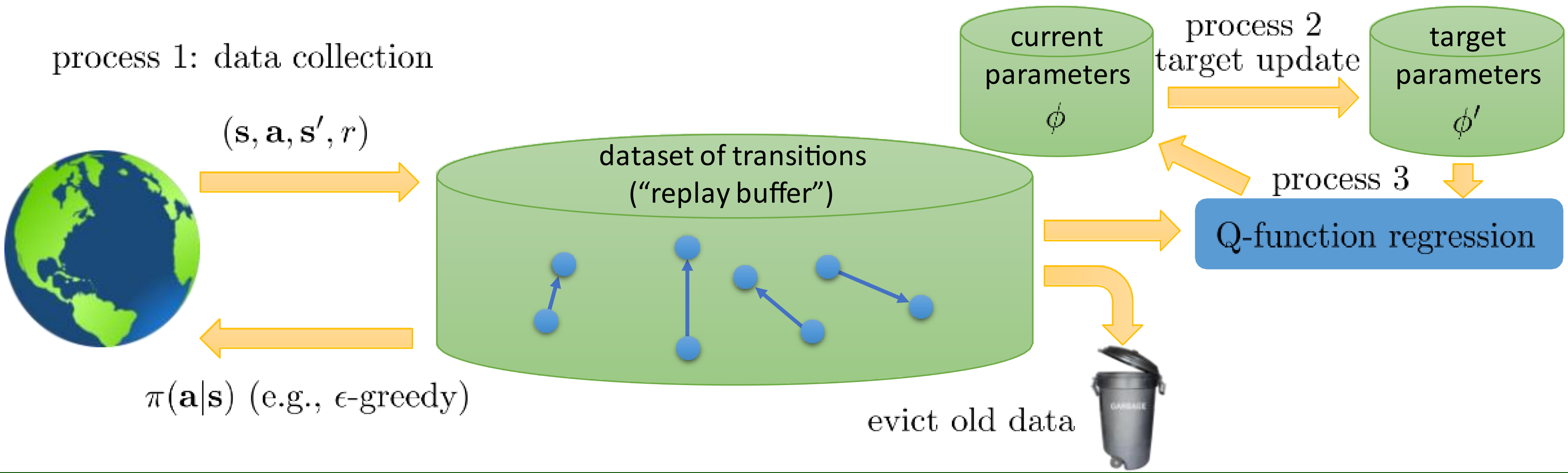
Deep RL with Q-Functions

- Deep Q-Network(DQN) Summary
  - Use experience replay and target network
  - The target network is time-delayed
  - Sample random mini-batch from replay buffer
  - Use stochastic gradient descent

Q-learning with replay buffer and target network:

1. save target network parameters: $\phi' \leftarrow \phi$
2. collect M datapoints $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy, add them to $\mathcal{B}$
3. sample a batch $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$ from $\mathcal{B}$
4. $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_{\phi'}(\mathbf{s}'_i, \mathbf{a}'_i)])$

$N\times$ $K\times$



Deep RL with Q-Functions

```
def main():
    env = gym.make(args.env)
    o_dim = env.observation_space.shape[0]
    a_dim = env.action_space.n
    agent = DQN(env, o_dim, args.hidden, a_dim)
    for i_episode in range(args.n_episodes):
        obs = env.reset()
        episode_reward = 0
        done = False
        while not done:
            action = agent.choose_action(obs)
            next_obs, reward, done, info = env.step(action)
            agent.store_transition(obs, action, reward, next_obs, done)
            episode_reward += reward
            obs = next_obs
            if agent.buffer.len() >= args.capacity:
                agent.learn()
```

Deep RL with Q-Functions

```
class DQN:
    def __init__(self, env, input_size, hidden_size, output_size):
        self.env = env
        self.eval_net = QNet(input_size, hidden_size, output_size)
        self.target_net = QNet(input_size, hidden_size, output_size)
        self.optim = optim.Adam(self.eval_net.parameters(), lr=args.lr)
        self.eps = args.eps
        self.buffer = ReplayBuffer(args.capacity)
        self.loss_fn = nn.MSELoss()
        self.learn_step = 0

    def choose_action(self, obs):
        if np.random.uniform() <= self.eps:
            action = np.random.randint(0, self.env.action_space.n)
        else:
            action_value = self.eval_net(obs)
            action = torch.max(action_value, dim=-1)[1].numpy()
        return int(action)

    def store_transition(self, *transition):
        self.buffer.push(*transition)
```

Deep RL with Q-Functions

```
def learn(self):
    if self.eps > args.eps_min:
        self.eps *= args.eps_decay

    if self.learn_step % args.update_target == 0:
        self.target_net.load_state_dict(self.eval_net.state_dict())
    self.learn_step += 1

    obs, actions, rewards, next_obs, dones = self.buffer.sample(args.batch_size)
    actions = torch.LongTensor(actions)  # LongTensor to use gather latter
    dones = torch.IntTensor(dones)
    rewards = torch.FloatTensor(rewards)

    q_eval = self.eval_net(obs).gather(-1, actions.unsqueeze(-1)).squeeze(-1)
    q_next = self.target_net(next_obs).detach()
    q_target = rewards + args.gamma * (1 - dones) * torch.max(q_next, dim=-1)[0]
    loss = self.loss_fn(q_eval, q_target)
    self.optim.zero_grad()
    loss.backward()
    self.optim.step()
```

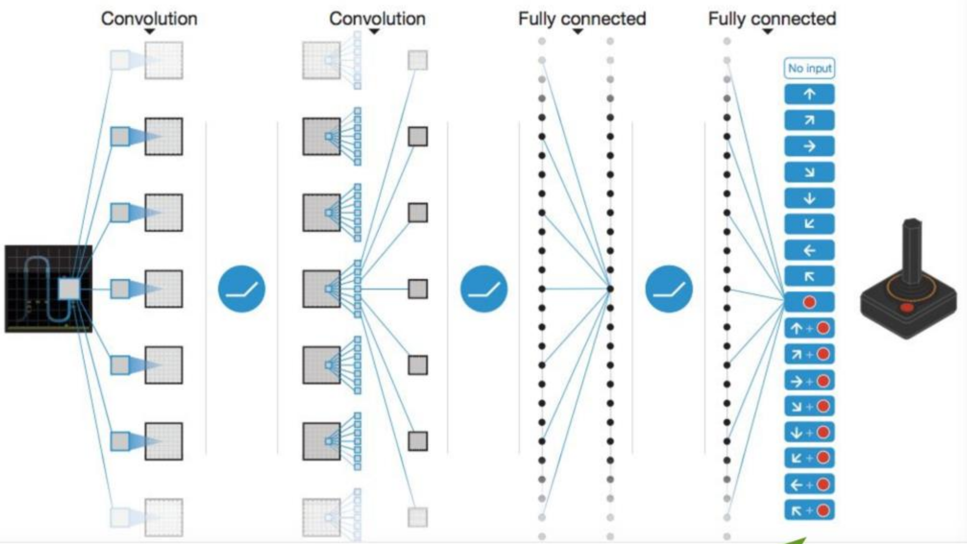
Deep RL with Q-Functions

```python
class ReplayBuffer:
    def __init__(self, capacity):
        self.buffer = []
        self.capacity = capacity

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

    def push(self, *transition):
        if len(self.buffer) == self.capacity:
            self.buffer.pop(0)
        self.buffer.append(transition)

    def sample(self, n):
        index = np.random.choice(len(self.buffer), n)
        batch = [self.buffer[i] for i in index]
        return zip(*batch)

    def clean(self):
        self.buffer.clear()
```

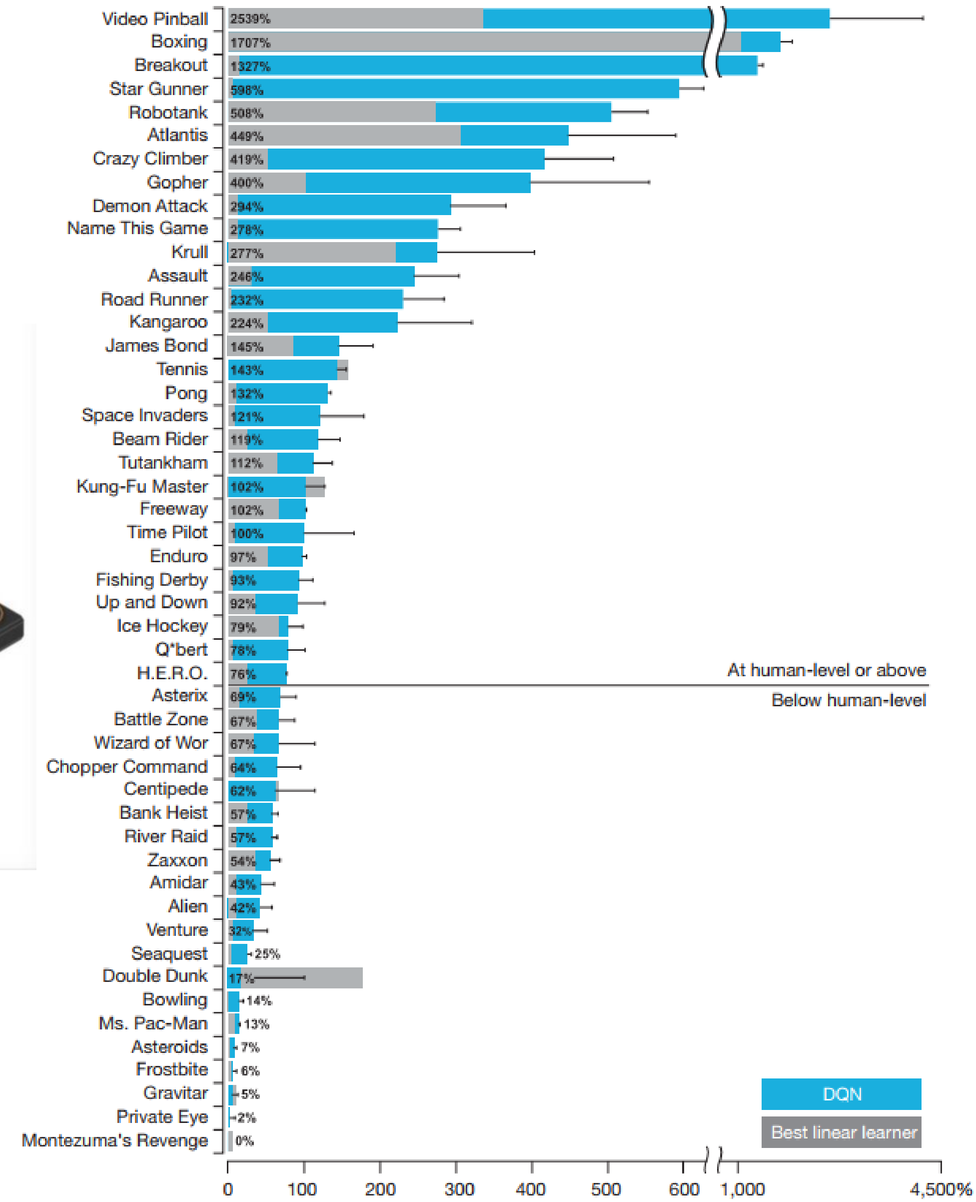
```python
class QNet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(QNet, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = torch.Tensor(x)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Deep RL with Q-Functions

Network and Performance



1 network, outputs Q value for each action



Deep RL with Q-Functions

- Variant
  - Double DQN： solving overestimation in DQN

target value $y_j = r_j + \gamma \max_{\mathbf{a}'_j} Q_{\phi'}(\mathbf{s}'_j, \mathbf{a}'_j)$

this last term is the problem

imagine we have two random variables: X_1 and X_2

$$E[\max(X_1, X_2)] \geq \max(E[X_1], E[X_2])$$

$Q_{\phi'}(\mathbf{s}', \mathbf{a}')$ is not perfect – it looks "noisy"

hence $\max_{\mathbf{a}'} Q_{\phi'}(\mathbf{s}', \mathbf{a}')$ *overestimates* the next value!

idea: don't use the same network to choose the action and evaluate value!

"double" Q-learning: use two networks:

$$Q_{\phi_A}(\mathbf{s}, \mathbf{a}) \leftarrow r + \gamma Q_{\phi_B}(\mathbf{s}', \arg\max_{\mathbf{a}'} Q_{\phi_A}(\mathbf{s}', \mathbf{a}'))$$

$$Q_{\phi_B}(\mathbf{s}, \mathbf{a}) \leftarrow r + \gamma Q_{\phi_A}(\mathbf{s}', \arg\max_{\mathbf{a}'} Q_{\phi_B}(\mathbf{s}', \mathbf{a}'))$$

if the two Q's are noisy in *different* ways, there is no problem

Deep RL with Q-Functions

- Variant
  - Double DQN： solving overestimation in DQN

where to get two Q-functions?

just use the current and target networks!

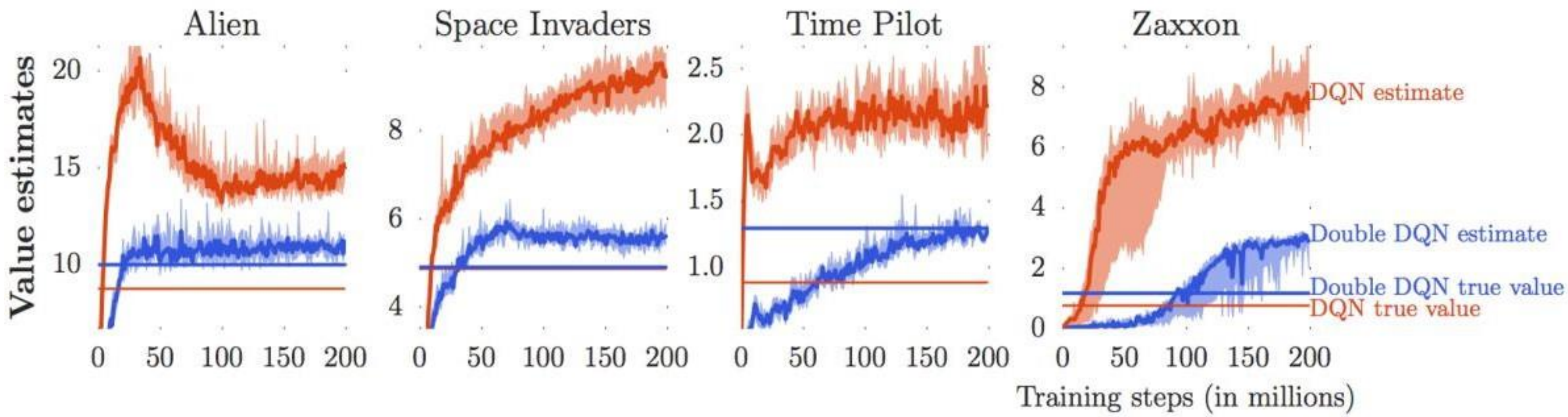
standard Q-learning: $y = r + \gamma Q_{\phi'}(\mathbf{s}', \arg\max_{\mathbf{a}'} Q_{\phi'}(\mathbf{s}', \mathbf{a}'))$

double Q-learning: $y = r + \gamma Q_{\phi'}(\mathbf{s}', \arg\max_{\mathbf{a}'} Q_{\phi}(\mathbf{s}', \mathbf{a}'))$

just use current network (not target network) to evaluate action

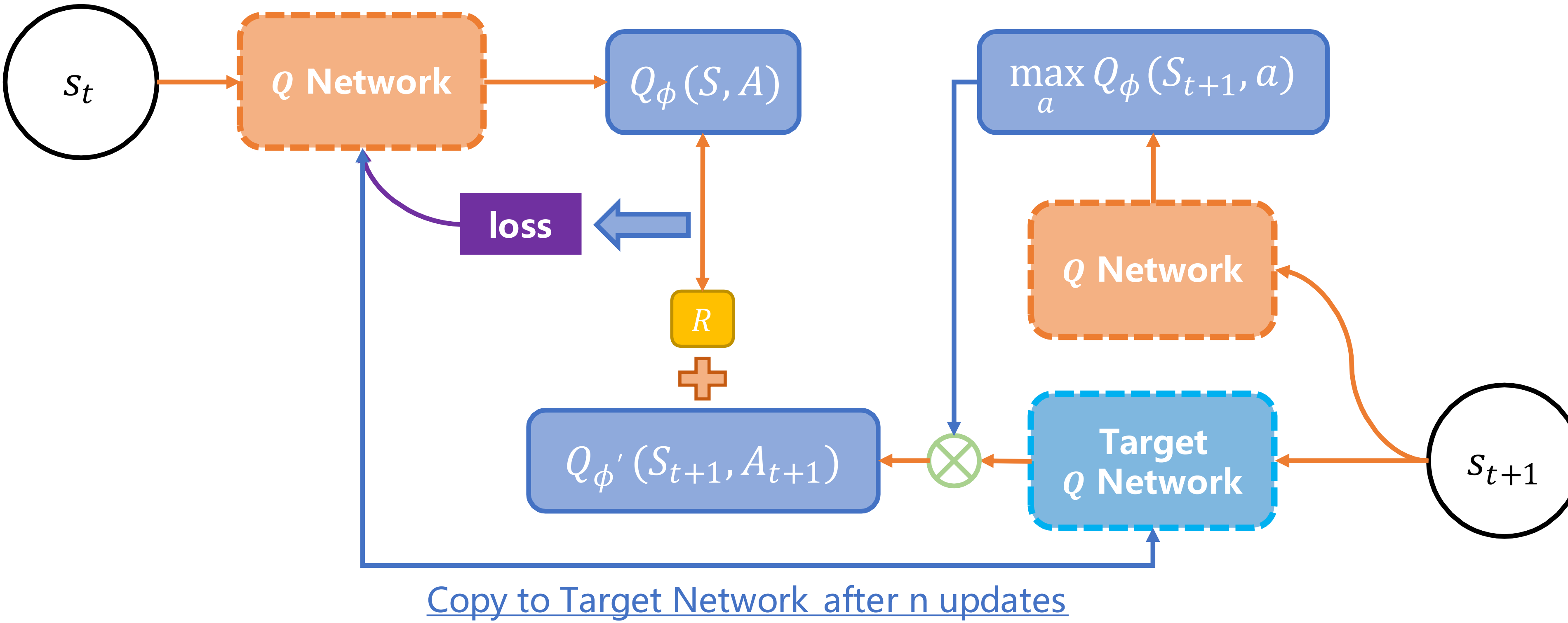
still use target network to evaluate value!

  - Value estimation in Atari



Deep RL with Q-Functions

- Variant
  - Double DQN：solving overestimation in DQN



Deep RL with Q-Functions

☐ Performance of Double DQN in Atari

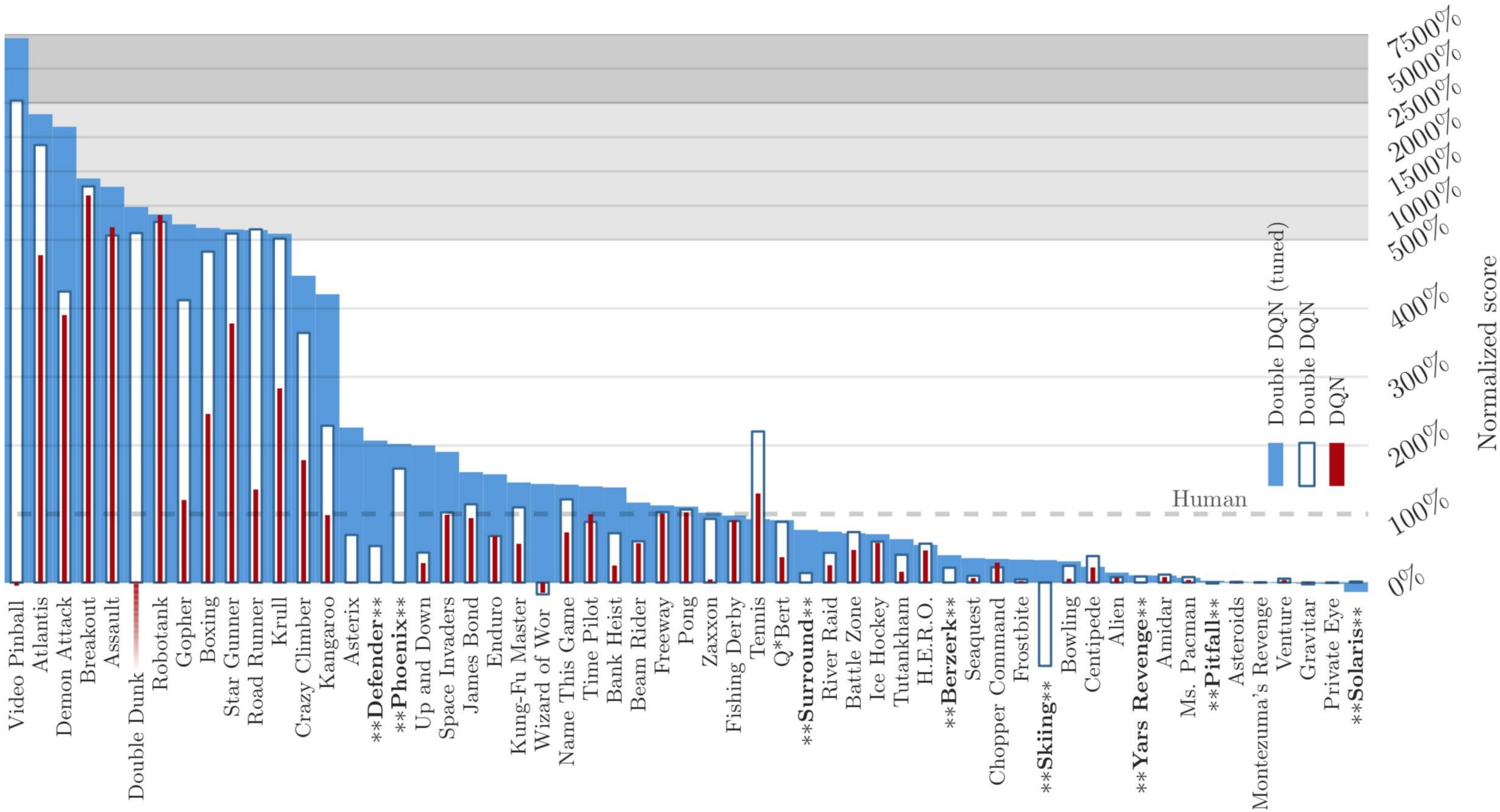
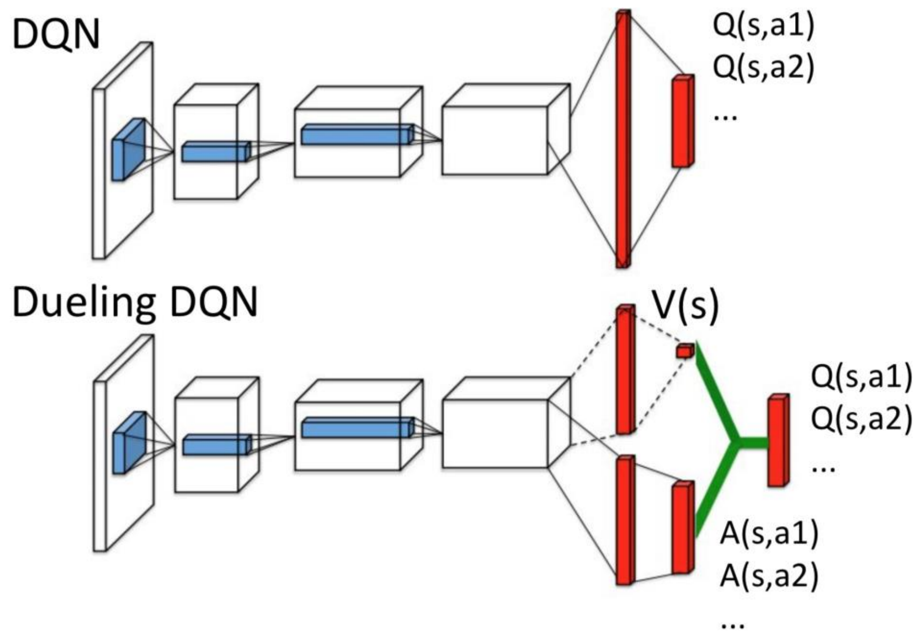


Figure: van Hasselt, Guez, Silver, 2015

Deep RL with Q-Functions

- Variant
  - Dueling DQN
    - Sometimes it is unnecessary to know the exact value of each action
    - Split the Q-values in two different parts, the value function V(s) and the advantage function A(s, a), Q(s, a) = V(s) + A(s, a)
    - Value function V(s) : how much reward we will collect from the state s
    - Advantage function A(s, a) : how much better one action is compared to the other actions.
  - Prioritized experience replay
    - Weigh the samples so that “important” ones are drawn more frequently for training
  - Rainbow
    - Combining improvements : Double DQN、Dueling DQN、Prioritized Replay Buffer、Multi-Step Learning、Distributional DQN（Categorical DQN）、NoisyNet

Network and performance of Dueling DQN



Wang et.al., ICML, 2016

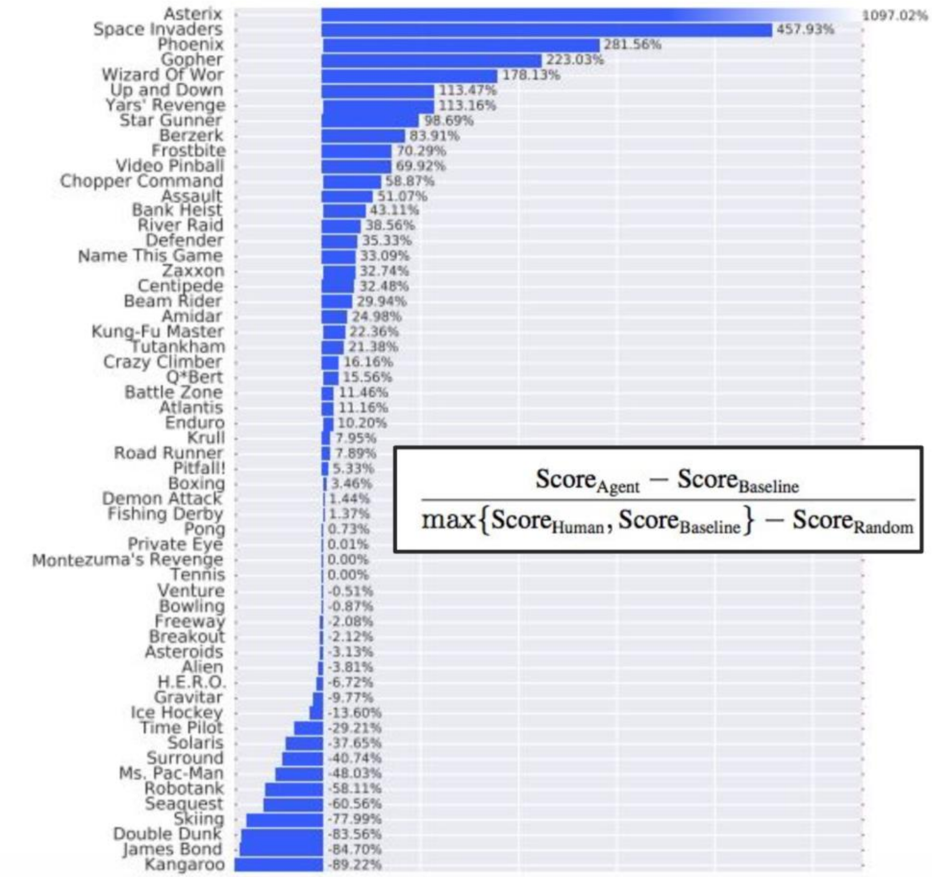


Figure: Wang et al, ICML 2016

Deep RL with Q-Functions

Performance of Prioritized Experience Replay in Atari

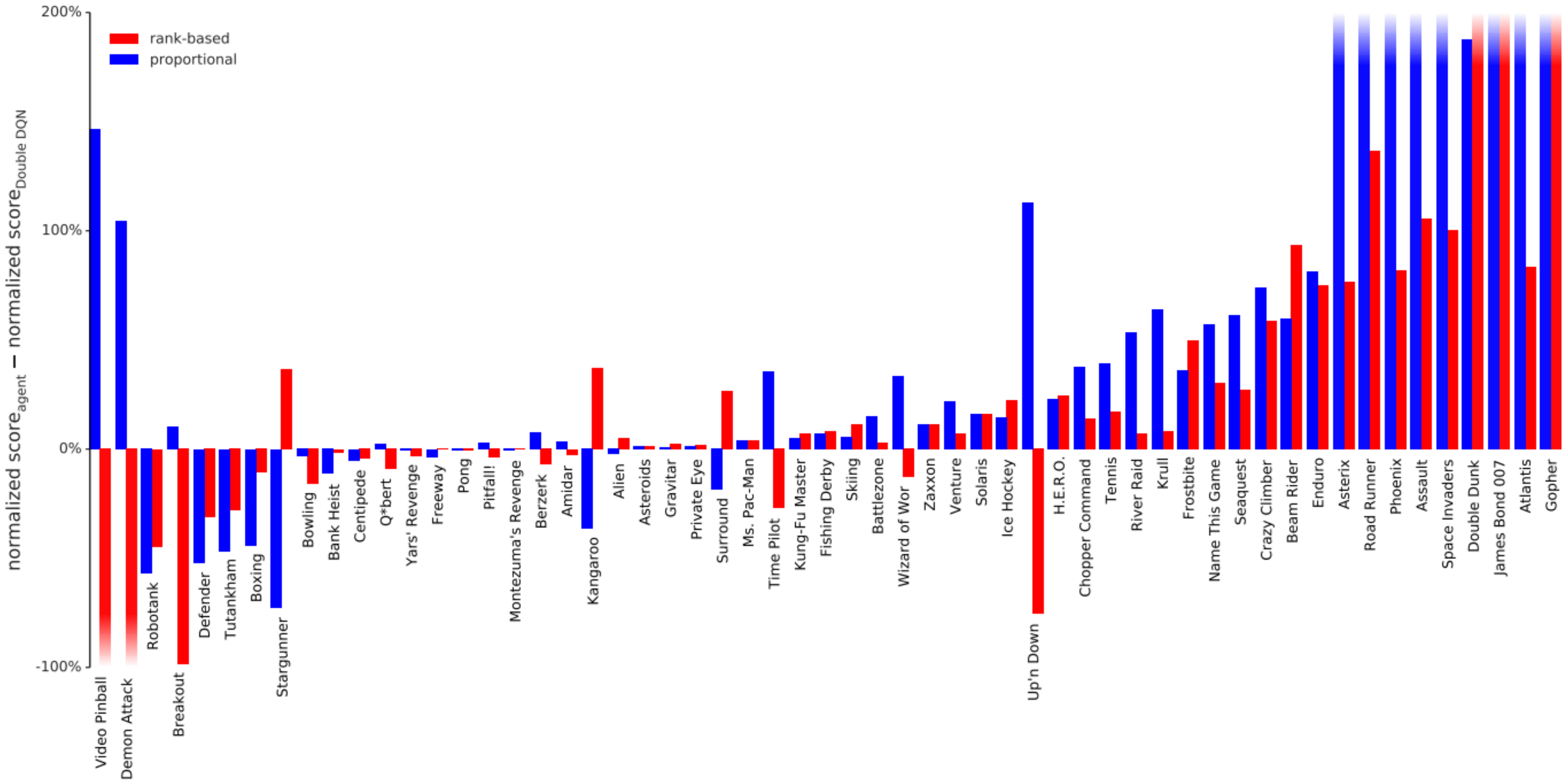


Figure: Schaul, Quan, Antonoglou, Silver ICLR 2016

Deep RL with Q-Functions

◻ Performance of Rainbow

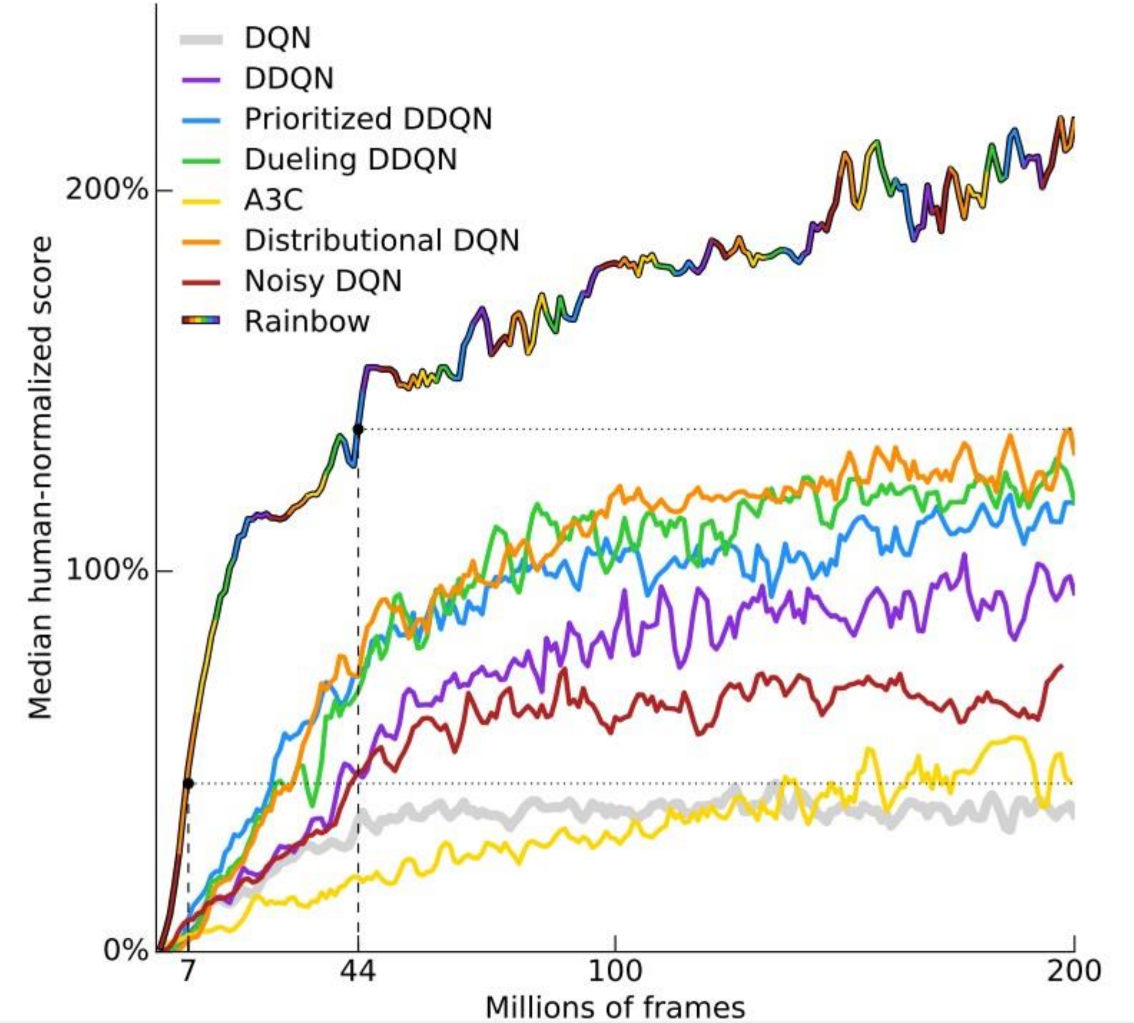


Figure: Hessel, Matteo, et al. "Rainbow: Combining Improvements in Deep Reinforcement Learning."

Deep RL with Q-Functions

- Q-learning with continuous actions
  - Problem

$$\pi(\mathbf{a}_t|\mathbf{s}_t) = \begin{cases} 1 \text{ if } \mathbf{a}_t = \arg\max_{\mathbf{a}_t} Q_\phi(\mathbf{s}_t, \mathbf{a}_t) \\ 0 \text{ otherwise} \end{cases}$$

$$\text{target value } y_j = r_j + \gamma \max_{\mathbf{a}'_j} Q_{\phi'}(\mathbf{s}'_j, \mathbf{a}'_j)$$

  - Solution
    - $\max_a Q(s,a) \approx \max\{Q(s,a_1), \dots, Q(s,a_N)\}$, $(a_1, \dots, a_N)$sampled from some distribution (e.g., uniform, Gaussian), but not very accurate.
    - Learn an approximate maximizer, Policy Gradient algorithm or DDPG ("deterministic" actor-critic, Lillicrap et al., ICLR 2016)

Deep RL with Policy Gradient

- Recap: Policy Gradient
  - Goal: given a policy $\pi_\theta(s,a)$ with parameters θ, find best θ that maximize $V(s,\theta)$
  - Can use gradient free optimization
    - Hill climbing、Cross-Entropy method etc.
  - Assume policy π_θ is differentiable and we can calculate gradient $\nabla_\theta \pi_\theta(s,a)$ analytically
    - Differentiable policy classes including: Softmax、Gaussian、Neural Networks
    - REINFORCE algorithm
    - A2C(Advantage Actor-Critic) algorithm
    - TRPO(Trust Region Policy Optimization) algorithm

Deep RL with policy gradient

DDPG(Deep Deterministic Policy Gradient)

- Idea: train actor network $\mu_\theta(s) \approx argmax_a Q_\phi(s,a)$

- Use four neural networks: a Q network, a deterministic policy network , a target q network, a target policy network

- The Q network and policy network is similar to actor-critic algorithm. But the Actor directly maps states to actions instead of outputting the probability distribution across an action space.

- Actor network:

$$\theta \leftarrow argmax_\theta Q_\phi(s, \mu_\theta(s)), \frac{dQ_\phi}{d\theta} = \frac{da}{d\theta}\frac{dQ_\phi}{da}$$

- Critic network: $y_j = r_j + \gamma Q_{\phi'}\left(s_j', \mu_{\theta'}(s_j')\right)$

$$\approx r_j + \gamma Q_{\phi'}\left(s_j', argmax_{a'} Q_{\phi'}(s_j', a_j')\right)$$

Lillicrap, Timothy P., et al. "Continuous control with deep reinforcement learning." *arXiv preprint arXiv:1509.02971* (2015).

Deep RL with policy gradient

DDPG

Pseudo Code

1. take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$, add it to $\mathcal{B}$
2. sample mini-batch $\{\mathbf{s}_j, \mathbf{a}_j, \mathbf{s}'_j, r_j\}$ from $\mathcal{B}$ uniformly
3. compute $y_j = r_j + \gamma Q_{\phi'}(\mathbf{s}'_j, \mu_{\theta'}(\mathbf{s}'_j))$ using *target* nets $Q_{\phi'}$ and $\mu_{\theta'}$
4. $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}(\mathbf{s}_j, \mathbf{a}_j)(Q_\phi(\mathbf{s}_j, \mathbf{a}_j) - y_j)$
5. $\theta \leftarrow \theta + \beta \sum_j \frac{d\mu}{d\theta}(\mathbf{s}_j) \frac{dQ_\phi}{d\mathbf{a}}(\mathbf{s}_j, \mu(\mathbf{s}_j))$
6. update ϕ' and θ' (e.g., Polyak averaging)

Soft Updates(different with DQN)

- Slowly track those of the learned networks via "soft updates"

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta'$$
$$\phi' \leftarrow \tau\phi + (1-\tau)\phi'$$

Tricks in Deep RL

- Simplify the problem by using a low-dimensional state space or action space
- Simplify the reward function
- Scaling observation and reward: normalization, clipping, etc.
- GAE, λ-return, etc.
- Exploration and Exploitation: entropy, Epsilon annealing, etc.
- Parallelized environment
- Test your algorithm on a known baseline environment
- Mini-batch update
- Parameter sharing
- Activation function: relu and tanh
- Orthogonal initialization and layer scaling
- Optimizer: Adam or RMSprop
- Global Gradient Clipping
- Value Function Loss Clipping
- Try different random seeds
- Look at episode return closely