

Deep Q-Networks

Roberto Capobianco



SAPIENZA
UNIVERSITÀ DI ROMA

Recap

Value function approximators

Instead of using tables, we will represent V and Q with parameterized functions

$$V(s, \mathbf{w}) \approx V^\pi(s) \\ (\text{or } Q(s, a, \mathbf{w}) \approx Q^\pi(s, a))$$

where $V(s, \mathbf{w})$ can be a **linear combination of features**, a decision tree, K-nearest neighbor, a neural network.

How do we optimize V ?

We iteratively optimize the approx. value function, minimizing the mean squared value error

$$\text{MSVE}(\mathbf{w}) = \sum_s d(s) [Q(s,a) - Q_{\pi}(s,a,\mathbf{w})]^2$$

where $d: S \rightarrow [0,1]$, such that $\sum_s d(s)=1$,
is a distribution over the states

We seek for \mathbf{w}^* s.t. $\text{RMSE}(\mathbf{w}^*) \leq \text{RMSE}(\mathbf{w})$ for all \mathbf{w}

Cool, but how?



Optimization through Gradient Descent

Gradient Descent optimize a function $f(x, \mathbf{w})$ by minimizing an error $J(\mathbf{w})$. This is done by iteratively update w in the following way:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \nabla_{\mathbf{w}} J(\mathbf{w}_t)$$

Using the mean squared error (MSE):

$$\begin{aligned} \mathbf{w}_{t+1} &= \mathbf{w}_t - \alpha \nabla_{\mathbf{w}} [\tfrac{1}{2}(f^* - f(x_t, \mathbf{w}_t))^2] = \\ &= \mathbf{w}_t - \alpha (f^* - f(x_t, \mathbf{w}_t)) \nabla_{\mathbf{w}} f(x_t, \mathbf{w}_t) \end{aligned}$$



Optimization through Gradient Descent

Remembering that our function f is $Q(s, a, \mathbf{w})$ and that our f^* is Q^π we obtain:

$$\begin{aligned}\mathbf{w}_{t+1} &= \mathbf{w}_t - \alpha \nabla_{\mathbf{w}} [\tfrac{1}{2} (Q^\pi_t - Q(s_t, a_t, \mathbf{w}_t))^2] = \\ &= \mathbf{w}_t - \alpha (Q^\pi_t - Q(s_t, a_t, \mathbf{w}_t)) \nabla_{\mathbf{w}} Q(s_t, a_t, \mathbf{w}_t)\end{aligned}$$

Ok cool again, but what about $Q(s_t, a_t, \mathbf{w}_t)$?



Linear Combination State-Action Value Approx.

— — —

If we represent our state with a feature vector

$$\mathbf{x}(s,a) = (x_1(s,a), \dots, x_n(s,a))^T$$

And consider a parametric Q function in the form

$$Q(s,a,\mathbf{w}) = \mathbf{w}^T \mathbf{x}(s,a)$$

Then

$$\begin{aligned} J(\mathbf{w}_t) &= \mathbb{E}(Q^\pi(s_t, a_t) - \mathbf{w}_t^T \mathbf{x}(s_t, a_t))^2 \\ \mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha(Q^\pi_t - \mathbf{w}_t^T \mathbf{x}(s_t, a_t)) \mathbf{x}(s_t, a_t) \end{aligned}$$



Incremental Prediction Algorithms (forward-view)

What exactly is the target Q_t^π ?

- For MC, the target is the return G_t
- For TD(0) the TD target is $R_{t+1} + \gamma Q(s_{t+1}, a_{t+1}, \mathbf{w})$
- For forward-view TD(λ), the target is G_t^λ
- For backward-view TD(λ) the update is

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \delta_t \mathbf{e}_t$$

$$\delta_t = R_{t+1} + \gamma Q(s_{t+1}, a_{t+1}, \mathbf{w}) - Q(s_t, a_t, \mathbf{w})$$

$$\mathbf{e}_t = \gamma \lambda \mathbf{e}_{t-1} + \nabla_{\mathbf{w}} Q(s_t, a_t, \mathbf{w})$$



Least Squares

Given a Q function approx. $Q(s,a,\mathbf{w}) \sim Q^\pi(s,a)$

And an experience $D = \{(s_1, a_1, Q^\pi_1), (s_2, a_2, Q^\pi_2), \dots, (s_n, a_n, Q^\pi_n)\}$

The least squares algorithm finds a parameter vector \mathbf{w} minimizing the error:

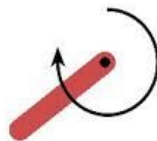
$$J(\mathbf{w}) = \sum_i (Q^\pi_i - Q(s_i, a_i, \mathbf{w}))^2$$

with the Stochastic Gradient Descent or with the closed form solution (linear version)

What about the features?

— — —

- Feature representation plays an important role
- Most times features should be inspected in combination:



Is the pendulum swinging up or down? The angle and the angular velocity shouldn't be used independently



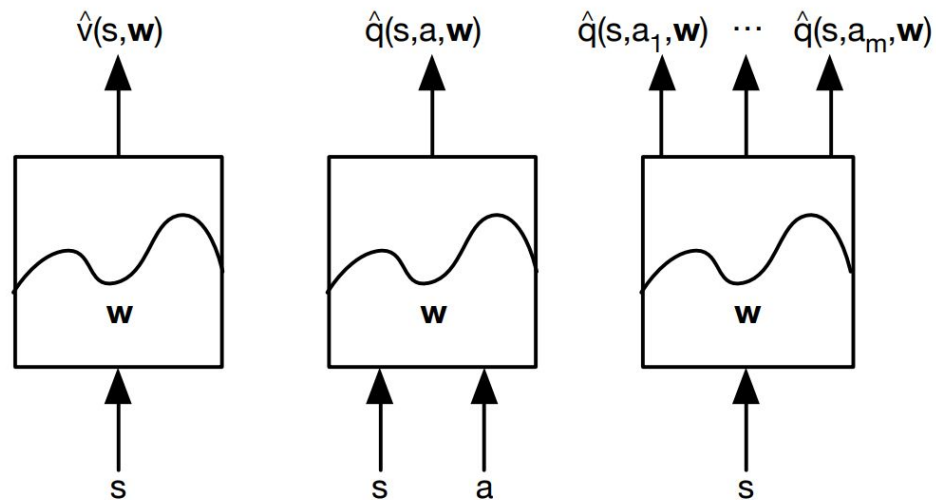
End Recap



SAPIENZA
UNIVERSITÀ DI ROMA

Types of VFA

— — —



Credits: David Silver



SAPIENZA
UNIVERSITÀ DI ROMA

Stochastic Gradient Descent with Experience Replay

Given an experience $D =$

$$\{(s_1, a_1, Q^{\pi_1}), (s_2, a_2, Q^{\pi_2}), \dots, (s_n, a_n, Q^{\pi_n})\}$$

Repeat:

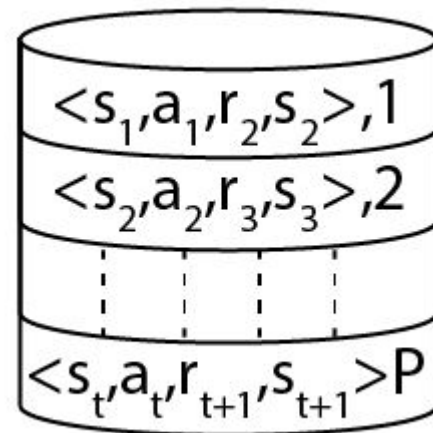
- Sample $(s, a, Q^{\pi}) \sim D$
- Apply the stochastic gradient descent update

$$\Delta \mathbf{w}_t = \alpha (Q^{\pi} - Q(s, a, \mathbf{w})) \nabla_{\mathbf{w}} Q(s, a, \mathbf{w})$$



Replay Buffer

- Used in most recent RL algorithms
 - Stores previous transitions experienced by an agent
 - Transitions are used later in time for training
-
- **Capacity:** total number of transitions stored in the buffer
 - **Age of transitions:** number of gradient steps taken by the learner since the transition was generated



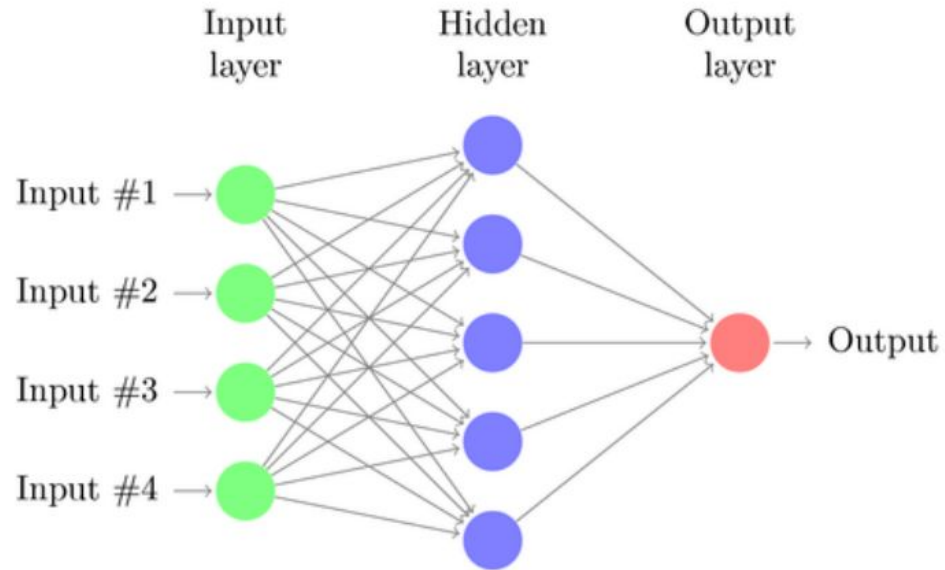
Replay Buffer - Why?

- Efficient use of previous experience
 - Same data used multiple times
 - Collecting real data costs
- Better convergence with function approximators
 - Makes data more i.i.d.
 - More similar to a supervised learning task



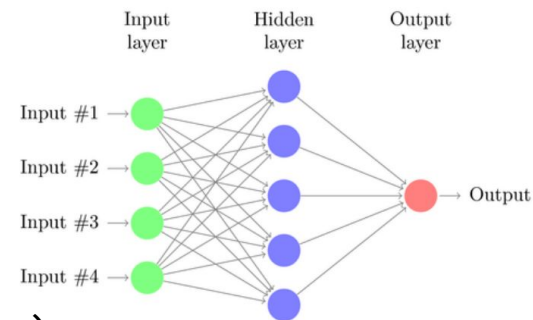
Neural Networks

— — —



Deep Neural Networks

— — —



- Composition of multiple layers (functions)
- To fit the parameters, require a loss function (a measure of error)
- Use chain rule to propagate the error
- Combine linear and non-linear transformations

Deep Neural Networks

— — —

Why DNNs instead of feature engineering + other function approximators?



Deep Neural Networks

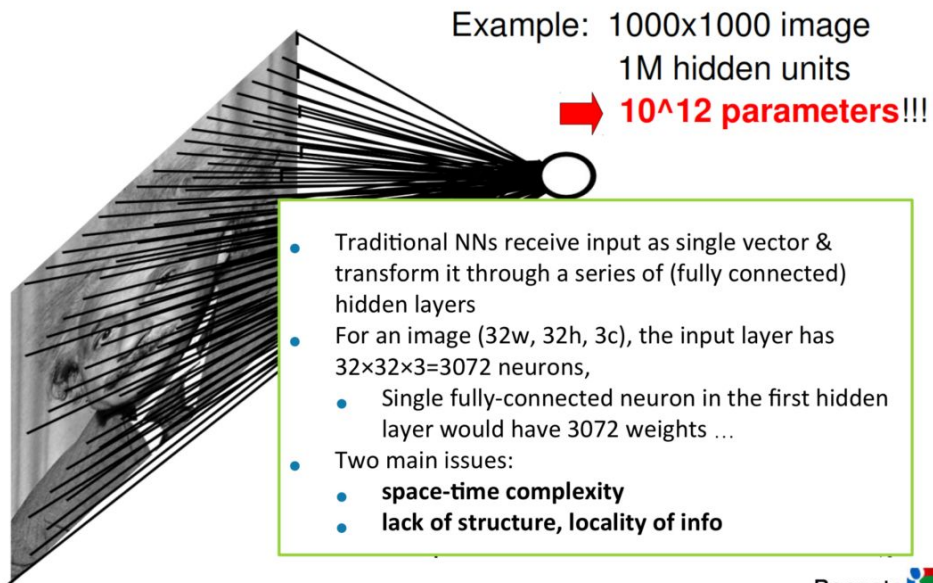
Why DNNs instead of feature engineering + other function approximators?

- Use distributed representations instead of local representations (like Kernels in SVMs)
- Universal function approximators
- Can potentially need exponentially less nodes/parameters (compared to a shallow net) to represent the same function
- Can learn the parameters using stochastic gradient descent



Convolutional Neural Networks

— — —



Ranzato 



SAPIENZA
UNIVERSITÀ DI ROMA

Convolutional Neural Networks

— — —

But images have structure!



SAPIENZA
UNIVERSITÀ DI ROMA

Convolutional Neural Networks

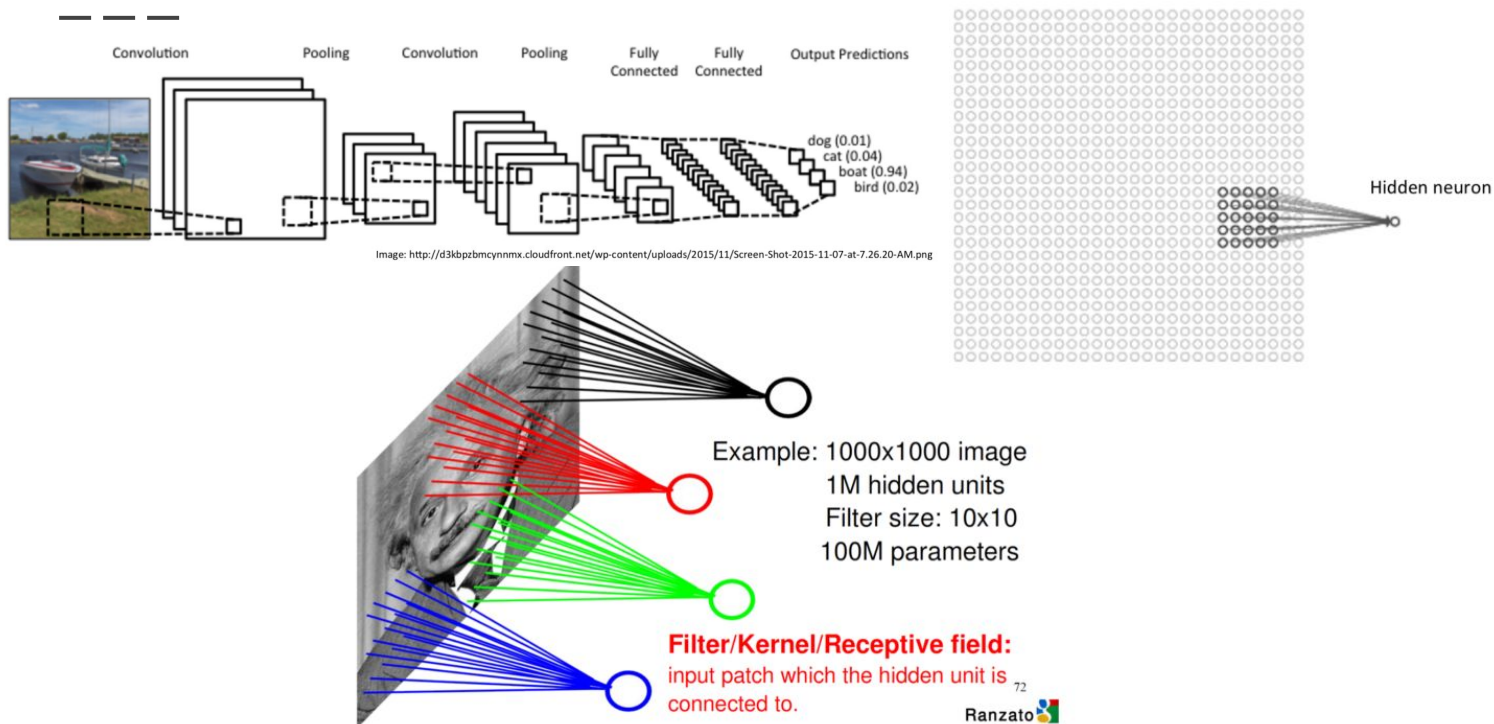
— — —

But images have structure!

Consider local structure and common extraction
of features



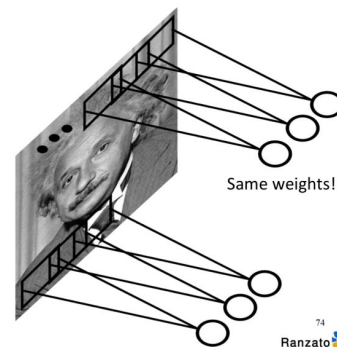
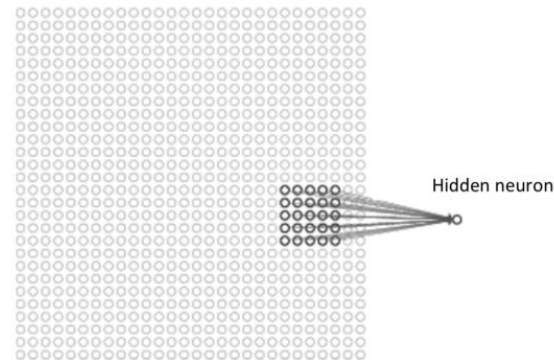
Convolutional Neural Networks



Convolutional Neural Networks

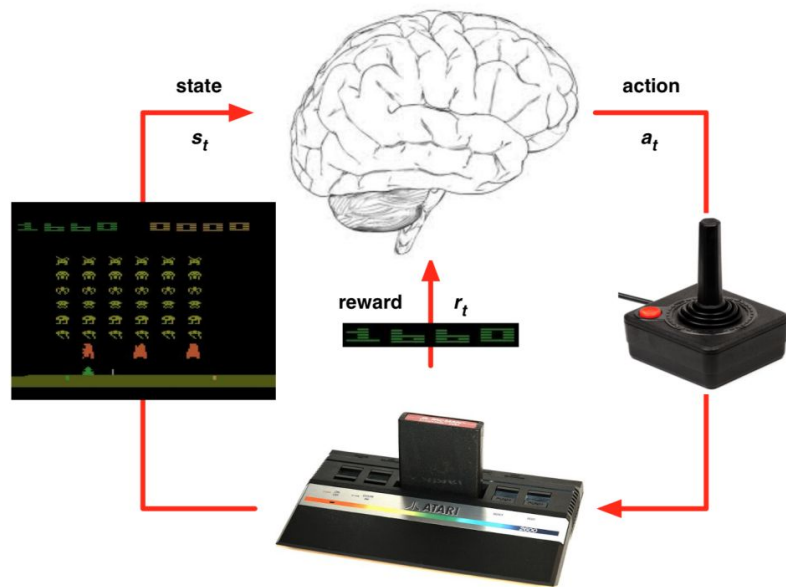
The same weights and bias are used for each of the hidden neurons

All the neurons in the first hidden layer detect exactly the same feature, just at different locations in the input image



CNNs in Atari

— — —



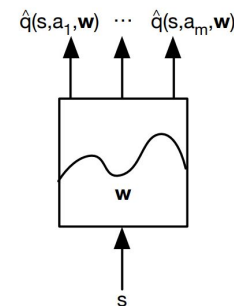
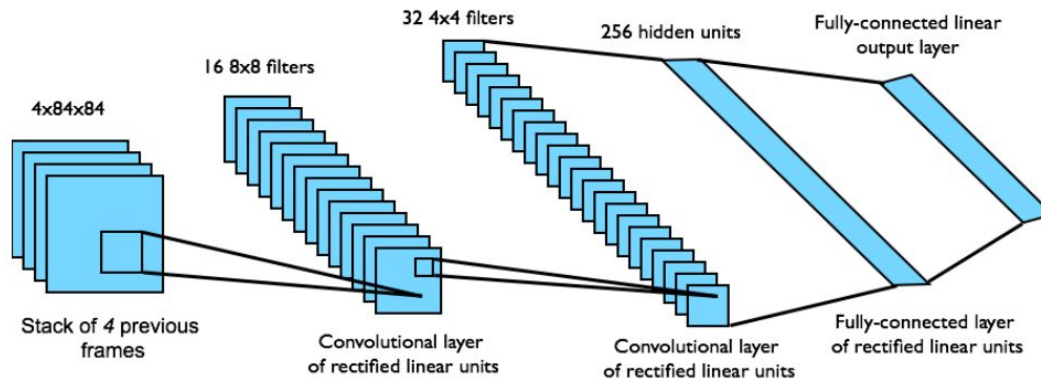
Credits: Emma Brunskill



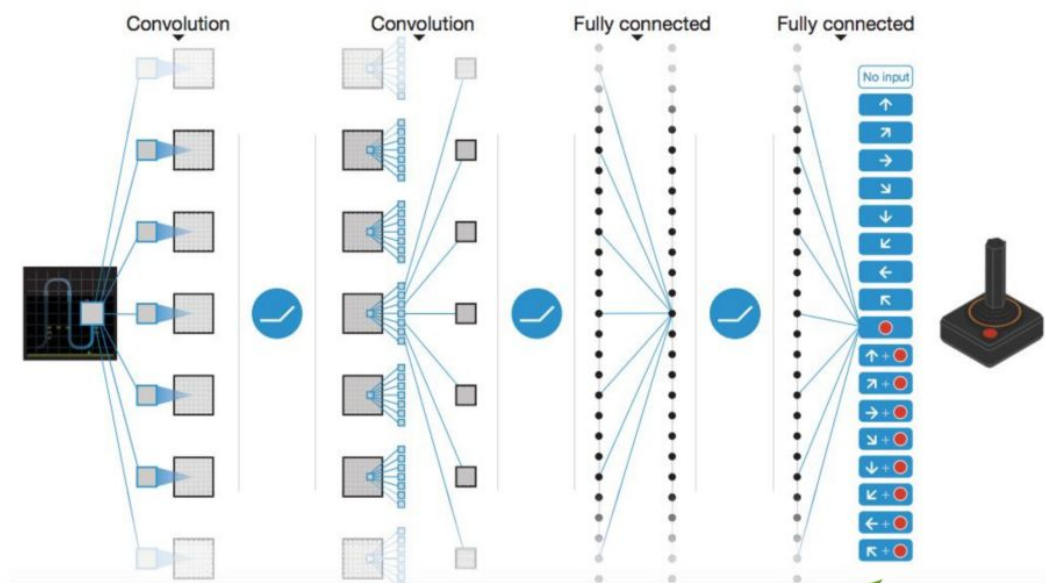
SAPIENZA
UNIVERSITÀ DI ROMA

DQN

- End-to-end learning of values $Q(s, a)$ from pixels s
- Input state s is stack of raw pixels from last 4 frames
- Output is $Q(s, a)$ for 18 joystick/button positions
- Reward is change in score for that step



DQN



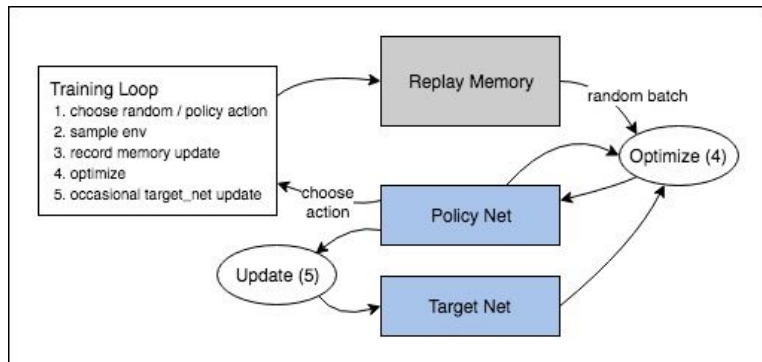
1 network, outputs Q value for each action



DQN

DQN uses experience replay:

- The sampled tuple is used to compute the update targets



s_1, a_1, r_2, s_2
s_2, a_2, r_3, s_3
s_3, a_3, r_4, s_4
...
$s_t, a_t, r_{t+1}, s_{t+1}$

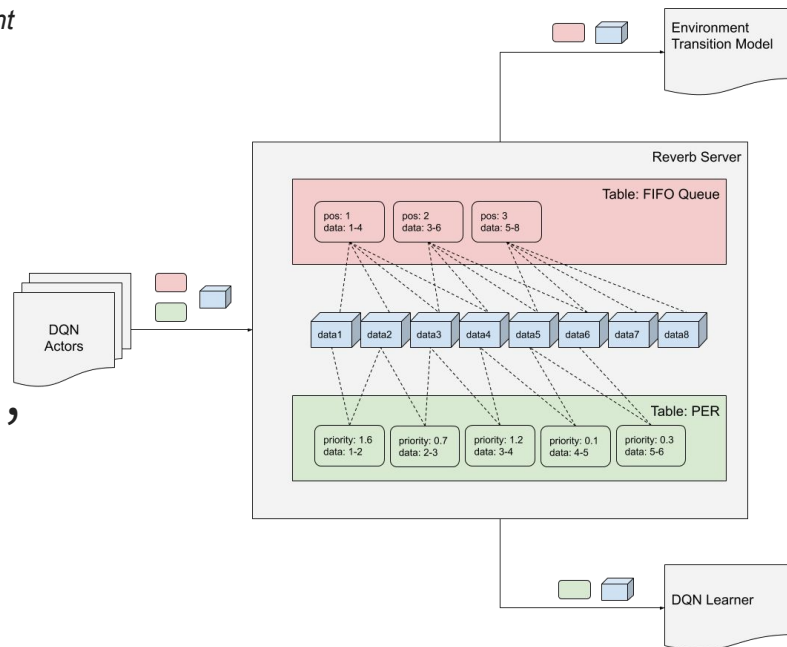
→ s, a, r, s'



Reverb

Cassirer, Albin, et al. "Reverb: A Framework For Experience Replay." *arXiv preprint arXiv:2102.04736* (2021). - Code: <https://github.com/deepmind/reverb>

- Efficient and easy-to-use data storage and transport system
- Experience replay system for distributed reinforcement learning
- Other data-structures for FIFO, LIFO and queues



DQN

DQN uses experience replay:

- The sampled tuple is used to compute the update targets
- The target is computed by using the estimation itself
- Since the weights get updated on the next round, the target value also changes

s_1, a_1, r_2, s_2
s_2, a_2, r_3, s_3
s_3, a_3, r_4, s_4
\dots
$s_t, a_t, r_{t+1}, s_{t+1}$

→ s, a, r, s'



DQN

DQN uses experience replay:

- The sampled tuple is used to compute the update targets
- To improve stability, have a specific **target network** used in the target calculation for multiple updates and fix its weights
- Update such weights every once a while copying the Q network in it

s_1, a_1, r_2, s_2
s_2, a_2, r_3, s_3
s_3, a_3, r_4, s_4
...
$s_t, a_t, r_{t+1}, s_{t+1}$

→ s, a, r, s'



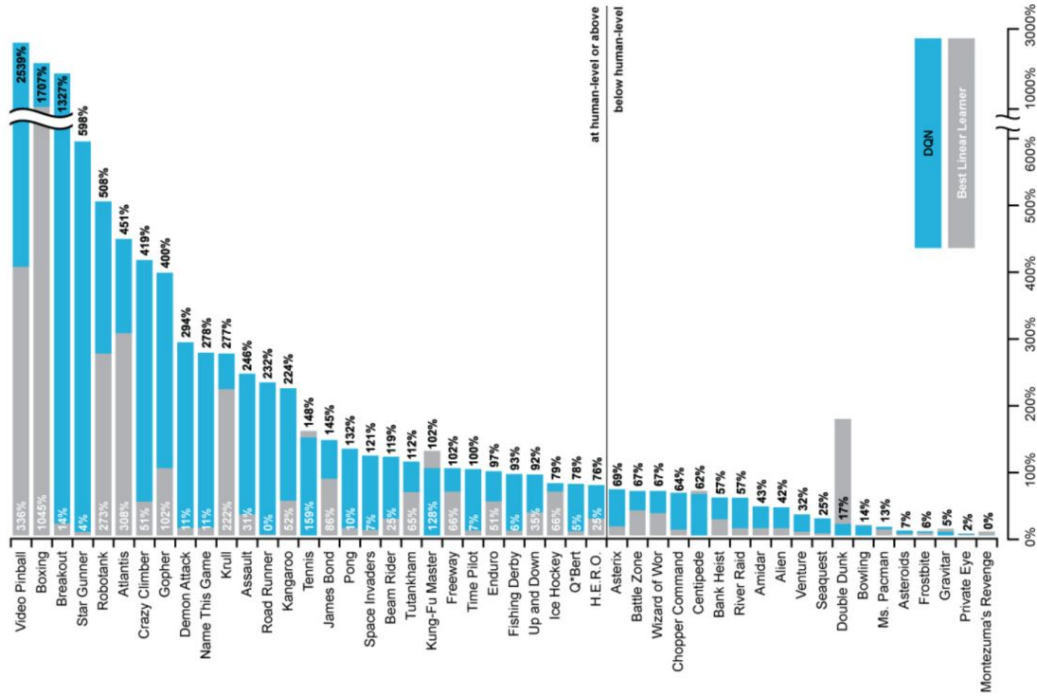
DQN Pseudo-code

```
1: Input  $C, \alpha, D = \{\}$ , Initialize  $\mathbf{w}, \mathbf{w}^- = \mathbf{w}, t = 0$ 
2: Get initial state  $s_0$ 
3: loop
4:   Sample action  $a_t$  given  $\epsilon$ -greedy policy for current  $\hat{Q}(s_t, a; \mathbf{w})$ 
5:   Observe reward  $r_t$  and next state  $s_{t+1}$ 
6:   Store transition  $(s_t, a_t, r_t, s_{t+1})$  in replay buffer  $D$ 
7:   Sample random minibatch of tuples  $(s_i, a_i, r_i, s_{i+1})$  from  $D$ 
8:   for  $j$  in minibatch do
9:     if episode terminated at step  $i + 1$  then
10:       $y_i = r_i$ 
11:     else
12:       $y_i = r_i + \gamma \max_{a'} \hat{Q}(s_{i+1}, a'; \mathbf{w}^-)$ 
13:     end if
14:     Do gradient descent step on  $(y_i - \hat{Q}(s_i, a_i; \mathbf{w}))^2$  for parameters  $\mathbf{w}$ :  $\Delta \mathbf{w} = \alpha(y_i - \hat{Q}(s_i, a_i; \mathbf{w})) \nabla_{\mathbf{w}} \hat{Q}(s_i, a_i; \mathbf{w})$ 
15:   end for
16:    $t = t + 1$ 
17:   if  $\text{mod}(t, C) == 0$  then
18:      $\mathbf{w}^- \leftarrow \mathbf{w}$ 
19:   end if
20: end loop
```

Credits: Emma Brunskill



DQN Results



DQN Result Analysis

Game	Linear	Deep Network	DQN w/ fixed Q	DQN w/ replay	DQN w/replay and fixed Q
Breakout	3	3	10	241	317
Enduro	62	29	141	831	1006
River Raid	2345	1453	2868	4102	7447
Seaquest	656	275	1003	823	2894
Space Invaders	301	302	373	826	1089

Credits: Emma Brunskill



SAPIENZA
UNIVERSITÀ DI ROMA

Maximization Bias

Greedy and epsilon greedy require a maximization step

This can lead to a positive bias:

- Consider s where many actions have $Q(s,a)$ that are all zero
- Estimated values are uncertain, some above and some below zero
- Maximum of the true values is zero, but max over estimates is positive

Maximization Bias

— — —

Problem is related to using same samples both to determine max action and estimate its value

We could learn two independent estimates:

- One could be used to determine maximizing action
- Other could be used to estimate value
- Only one estimate is updated on each episode

Memory requirement is doubled; computation is unchanged

Double Q-Learning

— — —

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$

Initialize $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, such that $Q(\text{terminal}, \cdot) = 0$

Loop for each episode:

Initialize S

Loop for each step of episode:

Choose A from S using the policy ε -greedy in $Q_1 + Q_2$

Take action A , observe R, S'

With 0.5 probability:

$$Q_1(S, A) \leftarrow Q_1(S, A) + \alpha \left(R + \gamma Q_2(S', \arg \max_a Q_1(S', a)) - Q_1(S, A) \right)$$

else:

$$Q_2(S, A) \leftarrow Q_2(S, A) + \alpha \left(R + \gamma Q_1(S', \arg \max_a Q_2(S', a)) - Q_2(S, A) \right)$$

$S \leftarrow S'$

until S is terminal



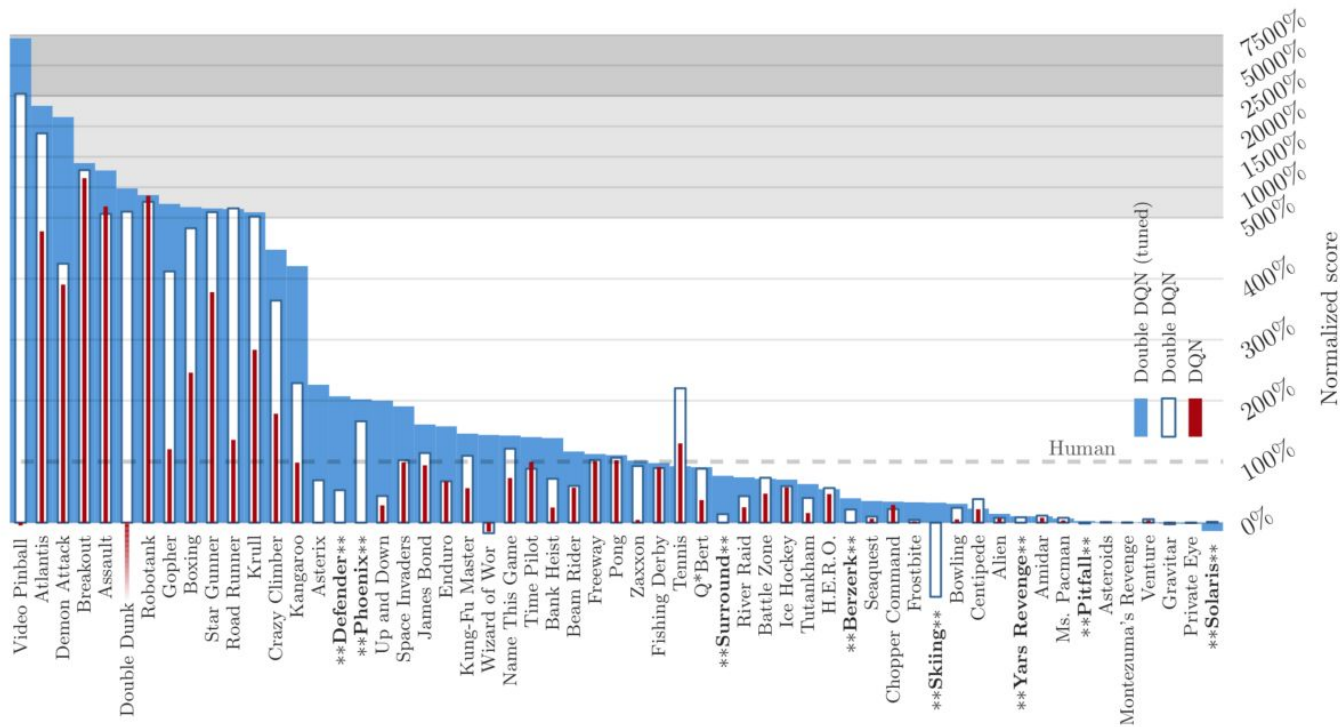
DDQN

- Extend Double Q-Learning to DQN
- Use current Q-network for action selection
- Use older (the target) Q-network (w^-) for action evaluation

$$\Delta \mathbf{w} = \alpha(r + \gamma \underbrace{\hat{Q}(\arg \max_{a'} \hat{Q}(s', a'; \mathbf{w}); \mathbf{w}^-)}_{\text{Action selection: } \mathbf{w}} - \hat{Q}(s, a; \mathbf{w}))$$



DDQN Results



Prioritized Experience Replay (PER)

— — —

Schaul, Tom, et al. "Prioritized experience replay." *arXiv preprint arXiv:1511.05952* (2015). - Code:

https://github.com/google/dopamine/tree/master/dopamine/replay_memory

- **Idea:** *“more frequently replay transitions with high expected learning progress, as measured by the magnitude of their temporal-difference (TD) error”*
- Prioritization can lead to:
 - Loss of diversity:
 - Use stochastic prioritization
 - Bias
 - Use importance sampling



PER - Stochastic VS Greedy Prioritization

- Greedy: Directly store magnitude of TD error in transitions

$$\delta_i = r_t + \gamma \max_{a \in \mathcal{A}} Q_{\theta}(s_{t+1}, a) - Q_{\theta}(s_t, a_t) \quad (s_t, a_t, r_t, s_{t+1}, |\delta_t|)$$

Problem:

- We would need to keep all of them up-to-date
- ER contains a lot of data

Solution:

- Only update items that are sampled during updates (stochastic)



PER - Sampling Probability

- Compute priorities p :
 - **Rank-based method:** sort items based on TD error magnitude $p_i = 1/\text{rank}(i)$
 - **Proportional method:** (epsilon to ensure non-zero) $p_i = |\delta_i| + \epsilon$
 - Priorities initialized at max value for new samples
- Compute probability of sampling the i -th data-point:

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

- alpha determines level of prioritization (0 means none)



PER - Bias Correction

Bias arises from the fact that we are changing the same distribution as the one that we are using for our updates

- Changes the distribution & solution

Solution:

- Use importance sampling for updates
- If $P(i)$ becomes 0, w becomes large
- If $P(i) = 1/N$, we have uniform sampling and $w = 1$
- Beta defines amount of correction

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



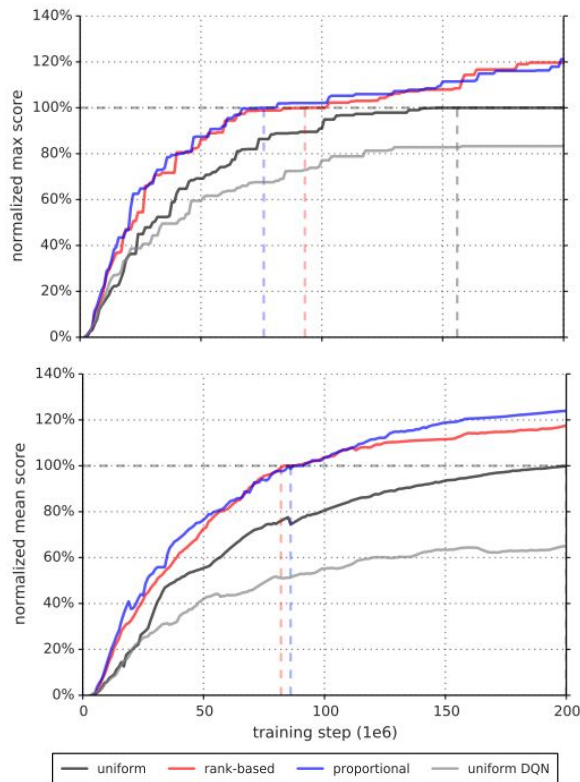
DDQN with PER

Algorithm 1 Double DQN with proportional prioritization

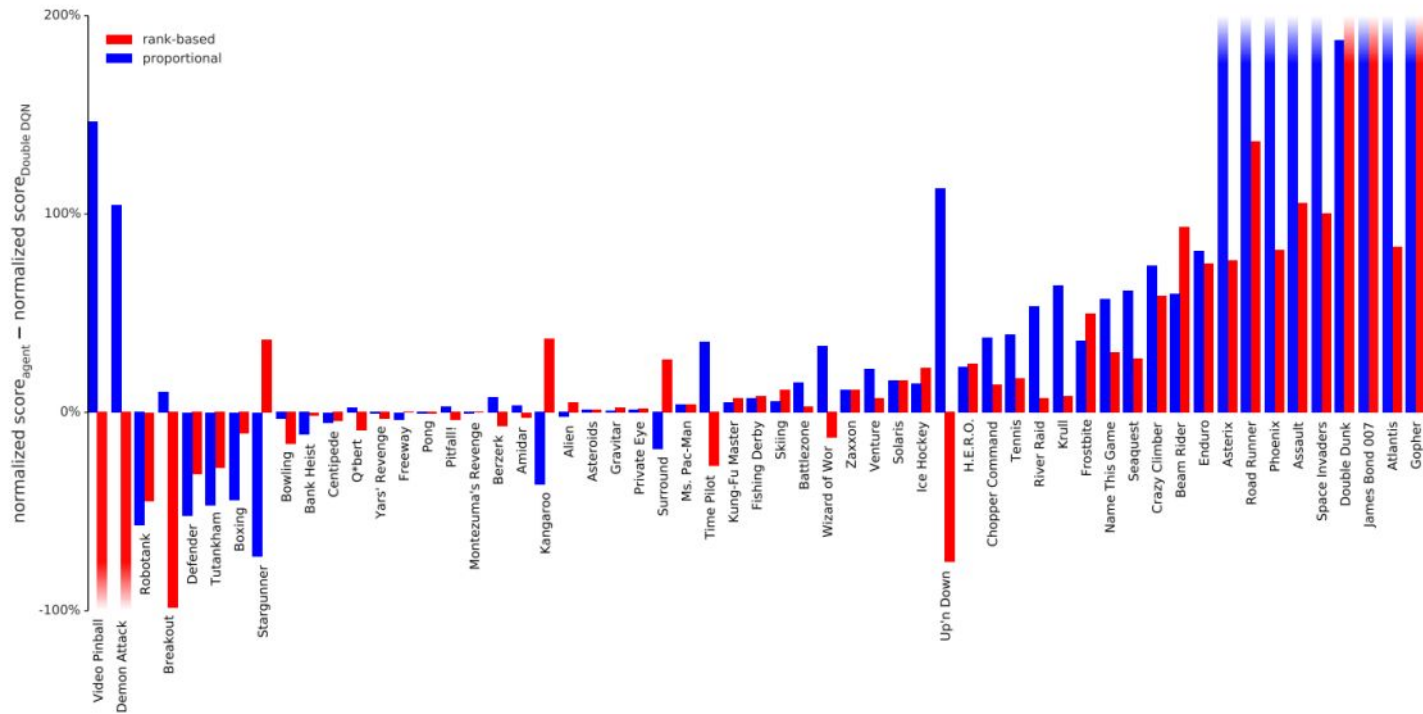
```

1: Input: minibatch  $k$ , step-size  $\eta$ , replay period  $K$  and size  $N$ , exponents  $\alpha$  and  $\beta$ , budget  $T$ .
2: Initialize replay memory  $\mathcal{H} = \emptyset$ ,  $\Delta = 0$ ,  $p_1 = 1$ 
3: Observe  $S_0$  and choose  $A_0 \sim \pi_\theta(S_0)$ 
4: for  $t = 1$  to  $T$  do
5:   Observe  $S_t, R_t, \gamma_t$ 
6:   Store transition  $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$  in  $\mathcal{H}$  with maximal priority  $p_t = \max_{i < t} p_i$ 
7:   if  $t \equiv 0 \pmod K$  then
8:     for  $j = 1$  to  $k$  do
9:       Sample transition  $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$ 
10:      Compute importance-sampling weight  $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$ 
11:      Compute TD-error  $\delta_j = R_j + \gamma_j Q_{\text{target}}(S_j, \arg \max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$ 
12:      Update transition priority  $p_j \leftarrow |\delta_j|$ 
13:      Accumulate weight-change  $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$ 
14:    end for
15:    Update weights  $\theta \leftarrow \theta + \eta \cdot \Delta$ , reset  $\Delta = 0$ 
16:    From time to time copy weights into target network  $\theta_{\text{target}} \leftarrow \theta$ 
17:  end if
18:  Choose action  $A_t \sim \pi_\theta(S_t)$ 
19: end for

```

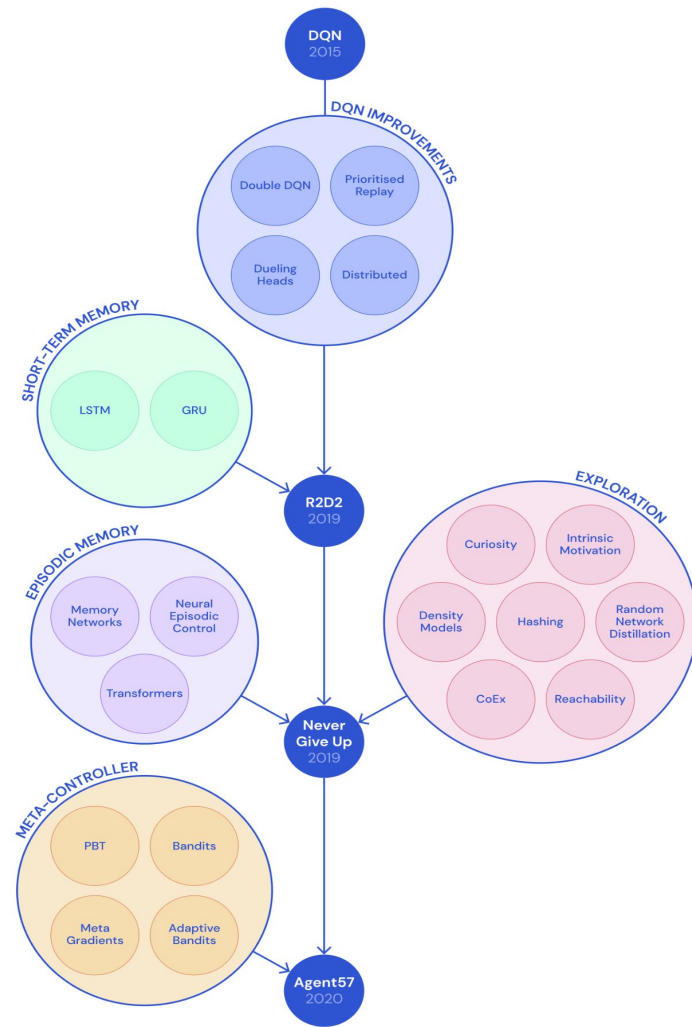


DDQN with PER Results



From DQN to Agent57

- DQN is the first approach beating humans in many Atari games
- Agent57 is the first approach beating humans in all Atari games



More on Off-Policy RL



On-Policy vs Off-Policy

— — —

On-policy: learn by what you do

Off-policy: learn by looking at someone else

- learn from observing other agents or humans
- reuse experience
- learn about optimal policy while following exploratory behaviors
- learn multiple policies while following a single policy



Expectation of Different Distribution

— — —

How do you estimate the expectation of a different distribution?



Expectation of Different Distribution

— — —
How do you estimate the expectation of a different distribution?

$$\begin{aligned}\mathbb{E}_{X \sim P}[f(X)] &= \sum P(X)f(X) \\ &= \sum Q(X) \frac{P(X)}{Q(X)} f(X) \\ &= \mathbb{E}_{X \sim Q} \left[\frac{P(X)}{Q(X)} f(X) \right]\end{aligned}$$



Importance Sampling

The same idea is applied to RL for off-policy learning



Importance Sampling for MC

The same idea is applied to RL for off-policy learning

Consider the MC setting: we want to use the returns from policy μ to evaluate π

Compute $G_t^{\pi/\mu}$ by multiplying **importance sampling corrections**

$$G_t^{\pi/\mu} = \frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} \frac{\pi(A_{t+1}|S_{t+1})}{\mu(A_{t+1}|S_{t+1})} \cdots \frac{\pi(A_T|S_T)}{\mu(A_T|S_T)} G_t$$



Importance Sampling for MC

The same idea is applied to RL for off-policy learning

Consider the MC setting: we want to use the returns from policy μ to evaluate π

Compute $G_t^{\pi/\mu}$ by multiplying **importance sampling corrections**

$$G_t^{\pi/\mu} = \frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} \frac{\pi(A_{t+1}|S_{t+1})}{\mu(A_{t+1}|S_{t+1})} \cdots \frac{\pi(A_T|S_T)}{\mu(A_T|S_T)} G_t$$

if μ is zero and π non-zero this cannot be used



Importance Sampling for TD

The same idea is applied to RL for off-policy learning

Consider the TD setting: we want to weight the TD target

$$\frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} (R_{t+1} + \gamma V(S_{t+1}))$$

Q-Learning does not need it, why?



Importance Sampling for TD

The same idea is applied to RL for off-policy learning

Consider the TD setting: we want to weight the TD target

$$\frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} (R_{t+1} + \gamma V(S_{t+1}))$$

Q-Learning does not need it, why?

We directly use the action from the target policy

