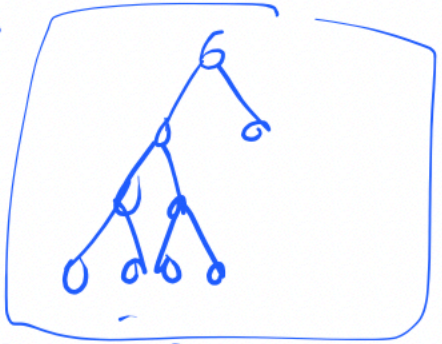
Biological Intelligence: trial & error process, rewards.

Reinforcement Learning

reward-driven learning

solve the problem end-to-end.

Biological greed.



Atari Games (optimistic)

AlphaGo

state → NN → action

Win / Lose / Draw

CSPs: states have a representation

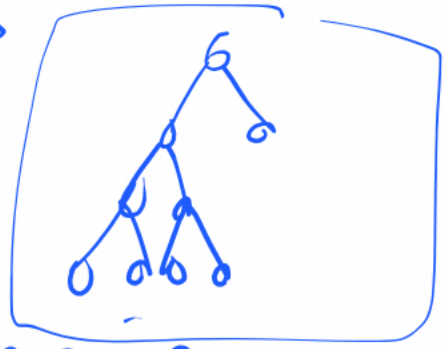
1

state → factored representation
→ atomic state

- In an MDP the agent senses the current state S_t and chooses an action a_t and performs it.

The environment responds by giving a reward $R(S_t$ ...) and producing the ...

process



Stateless

One-arm-bandit

Slotting Machine.

Has States (Game

RL

ven learning

(optimistic)

Games

1) Naive Approach

1) explore] N trials

2) exploitation

NN → action

cost

advantage

Win Lose Draw

certain heuristics

to make the search faster

2) Interleave exploration & exploitation

ϵ] explore —

$1-\epsilon$] exploit the best machine.

Annealing start with large ϵ

3) Upper Bound $\left(\text{Mean} + \frac{\sigma}{\sqrt{n}}\right)$

$r_0 = f(s_1, a_0)$

$s_1 = f(s_0, a_0)$

Alpha 90

CSPs. : states have a representation

- In an MDP the agent senses the current state S_t and chooses an action a_t and performs it.
- The environment responds by giving a reward $R(S_{t+1})$ and producing the succeeding state S_{t+1}
- For a deterministic environment the action outcomes are certain.

state → factored representation

→ atomic state

$s_0 a_0 r_0 \rightarrow s_1 a_1 r_1 \rightarrow s_2 a_2 r_2 \rightarrow s_3$

$s_0 a_0 \rightarrow r_1 s_1 a_1 \rightarrow r_2 s_2 a_2 \rightarrow r_3$ ...

$r_0 = f(s_0, a_0)$

$s_1 = f(s_0, a_0)$

$s_1 = f(s_0, a_0)$

$r_1 = R(s_1)$

the actual cost of

Credit

the

Draw

adv

certain hen

to make the sea

- The agent need to learn a policy for selecting its next action a_t based on the current observed state s_t

$$\pi(s_t) = a_t$$

- The agent would like to learn the best policy, i.e. the policy that produces the greatest possible cumulative reward

$$U^{\pi}(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots$$

γ : discount factor

- The best (optimal) policy chosen by the agent is the one which maximises the $U^{\pi}(s)$ for all states s.

$$\pi^* \equiv \arg\max_{\pi} U^{\pi}(s) \quad \forall s$$

- The value function $U^{\pi^*}(s)$ of such an optimal policy is referred to as $U^*(s)$

Select the best action

best action $Q(s,a)$

$U(s)$ best action

know the env. $P(s'|s,a)$

Q learning

- How can an agent learn an optimal policy π^* ?
- The training information is available to the learner as a sequence of immediate rewards $r(s_t, a_t)$ for $t = 0,1,2,\ldots$
- The agent's preference over states can be captured by the function $U^*(s)$
- However, the agent's policy must choose among actions, not among states.

- We need an evaluation function over actions
- We define an evaluation function $Q(s, a)$ so that its value is the maximum discounted cumulative reward that can be achieved starting from state s and applying action a as the first action.

$$Q(s, a) \equiv R(s) + \gamma U^*(\delta(s, a))$$

We rethink about rewards as associated with actions taken in a given state

$$Q(s, a) \equiv R(s, a) + \gamma U^*(\delta(s, a))$$

Known environment
δ is known.
$\delta(s,a)$ gives the next state s'

- The agent's policy can be worked out as

$$\pi^*(s) = \arg\max_a Q(s, a)$$

- If the agent learns the Q function instead of the value function $U^*(s)$, it will be able to select an optimal action even if it has no knowledge of the functions R and δ

- The agent can thus choose a globally optimal action sequence by utilising repeatedly the local values of Q for the current state.

- A lookahead search is not required.

- The Q function summarizes all the information needed to determine the discounted cumulative reward that will be gained in the future if action a is selected in state s.

provided we know the Q function

$s^0 a_0 s^1 a_1 s^2$.

Q

dont need R & δ.

Learning the Q function

- For training we only have a sequence of immediate rewards
- We learn the Q function using an iterative approximation scheme.
- We have

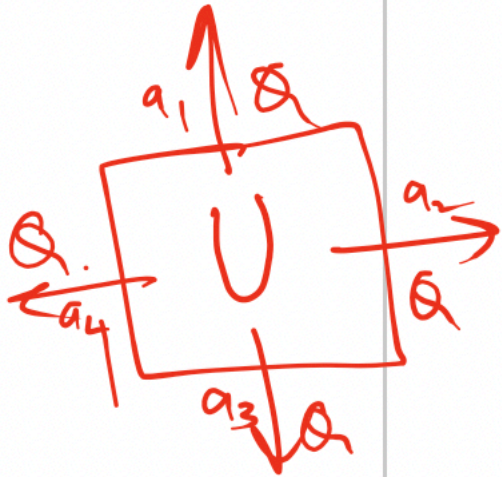
$$Q(s,a) \equiv R(s,a) + \gamma U^*(\delta(s,a))$$

and

$$U^*(s) = \max_{a'} Q(s,a')$$

So we have

$$Q(s,a) = R(s,a) + \gamma \max_{a'} Q(\delta(s,a),a')$$



Q learning algorithm

Env is unknown.

- For each s, a initialise the table entry $\hat{Q}(s, a)$ to zero

Observe the current state s

Do Forever

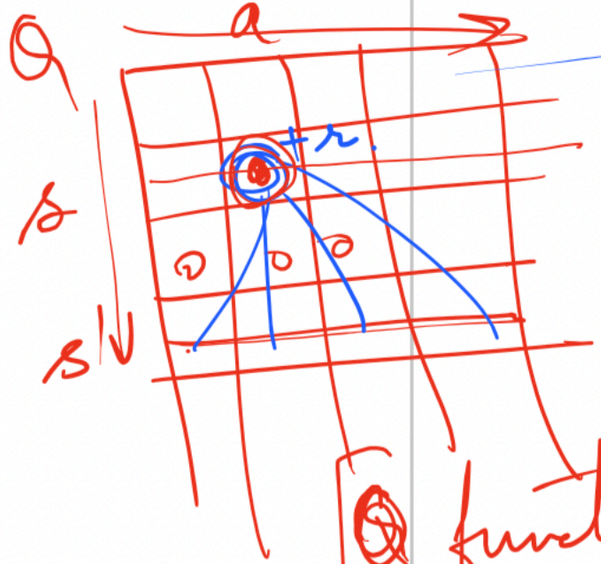
- Select an action a and execute it
- Receive immediate reward r
- Observe the new state s'
- Update the table entry for $\hat{Q}(s, a)$ as follows:

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

- $s \leftarrow s'$

$\hat{Q}$ estimated

Correct



deterministic $s \xrightarrow{a} s'$

Non deterministic $s \xrightarrow{a} P(s'|s,a)$

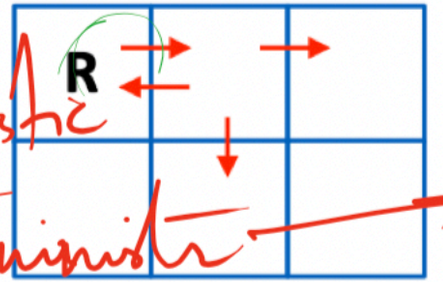
R: stochastic

Q function

Value function: V U^{π}

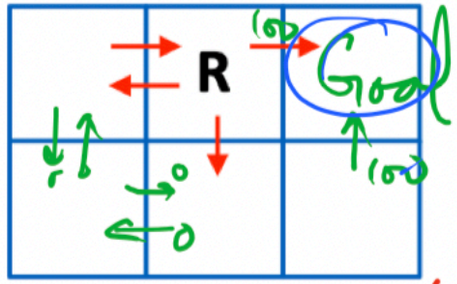
$Q(s,a)$

$U^{\pi}(b)$



Deterministic

Non deterministic



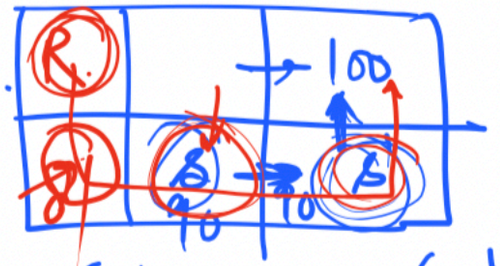
Env is known → transition $(s, a) \rightarrow s'$

Env is unknown → ??

Agent must act: $s \xrightarrow{a} s'$

episode

$Q(s, a)$



$Q(s,a) = 0 + \gamma(100)$

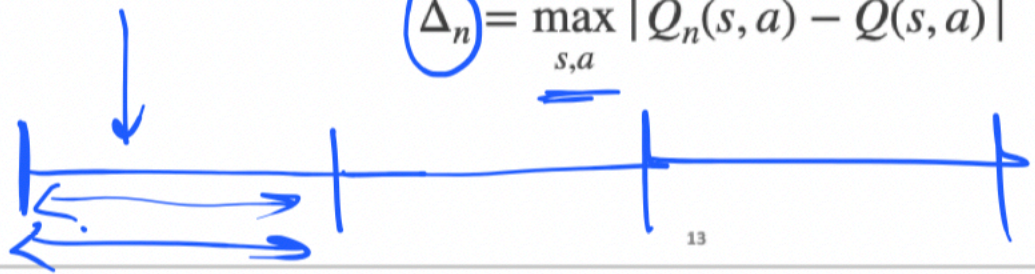
$\gamma = 0$ /

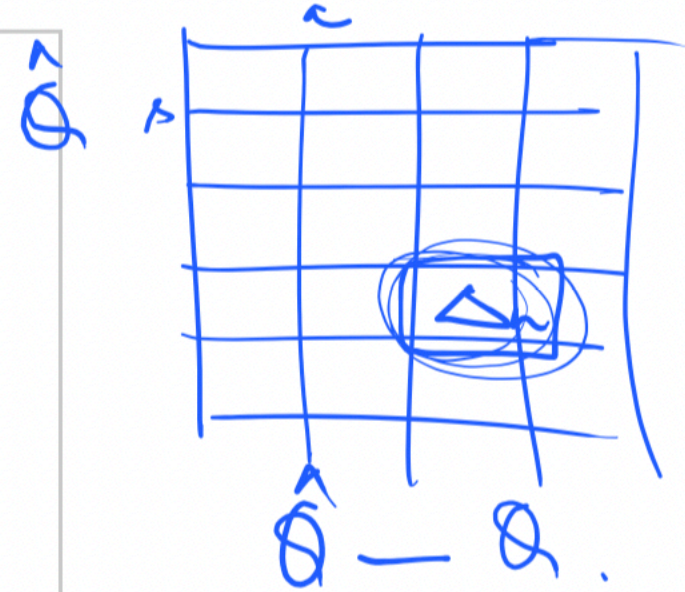
Convergence proof of Q Learning

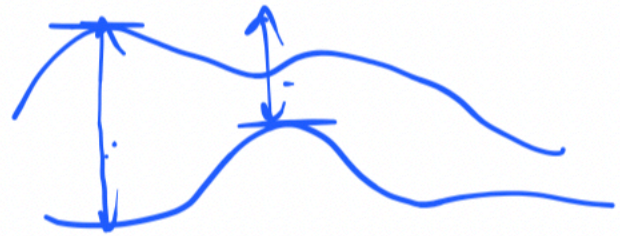
- The Q learning algorithm converges to the true Q function under certain conditions listed below:
  - The system is a deterministic MDP.
  - The immediate reward values are bounded. $|R(s,a)| < c$
  - If action a is a legal action from state s, then over time as the length of the action sequence approaches infinity, the agent must execute action a from state s with non-zero frequency.
    That is, every state-action transition is visited infinite often.

- The proof consists of showing that the maximum error over all entries in the Q table is reduced by a factor at least of γ whenever it is updated.
- Let $\hat{Q}_n(s,a)$ denote the agent's hypothesis $\hat{Q}(s,a)$ after the nth update.
- Since each state-action transition occurs infinitely often, consider consecutive intervals during which each state-action transition occurs at least once.
- Let Δ_n be the maximum error in $\hat{Q}_n$

$$\Delta_n = \max_{s,a} |\hat{Q}_n(s,a) - Q(s,a)|$$







- $|\hat{Q}_{n+1}(s,a) - Q(s,a)| = |(r + \gamma \max_{a'} \hat{Q}_n(s',a')) - (r + \gamma \max_{a'} Q(s',a'))|$

  update to Q_n · true

$$= \gamma |\max_{a'} \hat{Q}_n(s',a') - \max_{a'} Q(s',a')|$$

$$\leq \gamma \max_{a'} |\hat{Q}_n(s',a') - Q(s',a')|$$

$$\leq \gamma \max_{s'',a'} |\hat{Q}_n(s'',a') - Q(s'',a')|$$

$|\max f_1(x) - \max f_2(x)| \leq \max |f_1(x) - f_2(x)|$

$|\hat{Q}_{n+1}(s,a) - Q(s,a)| \leq \gamma \Delta_n$

Δ_n in the Q-table

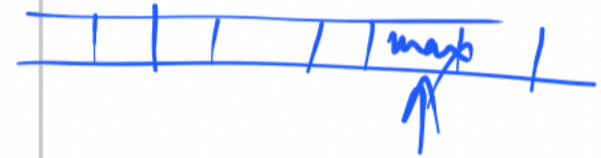
new diff

$\hat{Q}_{n+1}$ is more closer to Q_n

- The largest error in the initial table, Δ_0 is bounded because values of $\hat{Q}_0(s,a)$ and $Q(s,a)$ are bounded for all s, a.
- After the first interval during which each s, a is visited, the largest error in the table will be at most $\gamma\Delta_0$.
- After k such intervals, the error will be at most $\gamma^k\Delta_0$. Since each state is visited infinitely often, the number of such intervals is infinite , and $\Delta_n \to 0$ as $n \to \infty$.

Q table

Best strategy



$\gamma < 1$

Proof of convergence.

- Reinforcement Learning deals with how an agent can become proficient in an unknown environment

- The agent can make use of only the percepts and the occasional rewards.

RL

- The 3 main designs of the agent are:
  - The model-based design, using a transition model P and a value function U
  - The model-free design using an action-utility function Q
  - Reflex design, using a policy π

(known env)

reward

get the strategy

$Q(s,a)$

$\pi(s) \rightarrow$ action (strategy)

Passive Reinforcement Learning

- We first consider direct (utility) value estimation in the context of passive reinforcement learning
- In passive learning, the agent's policy π is fixed.
- Assuming a fully observable environment, the agent's task is to learn the (utility) values of the states.
- The agent does not know the transition model $P(s' \mid s, a)$ and the reward function $R(s)$

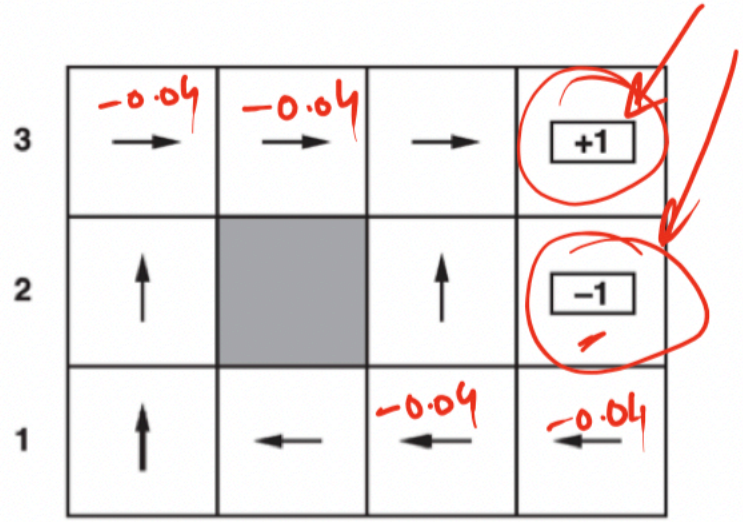
unknown env.

need to estimate.

- There are 3 approaches to learning the utility values for a state
  - Direct (Utility) Value Estimation
  - Adaptive Dynamic Programming (ADP)
  - Temporal-Difference (TD)

Bellman equations

Direct (Utility) Value Estimation



3	0.812	0.868	0.918	+1
2	0.762		0.660	−1
1	0.705	0.655	0.611	0.388
	1	2	3	4

policy

- The agent executes a set of trials in the environment using its policy π.

$(1,1)_{\text{-.04}} \rightsquigarrow (1,2)_{\text{-.04}} \rightsquigarrow (1,3)_{\text{-.04}} \rightsquigarrow (1,2)_{\text{-.04}} \rightsquigarrow (1,3)_{\text{-.04}} \rightsquigarrow (2,3)_{\text{-.04}} \rightsquigarrow (3,3)_{\text{-.04}} \rightsquigarrow (4,3)_{\text{+1}}$

$(1,1)_{\text{-.04}} \rightsquigarrow (1,2)_{\text{-.04}} \rightsquigarrow (1,3)_{\text{-.04}} \rightsquigarrow (2,3)_{\text{-.04}} \rightsquigarrow (3,3)_{\text{-.04}} \rightsquigarrow (3,2)_{\text{-.04}} \rightsquigarrow (3,3)_{\text{-.04}} \rightsquigarrow (4,3)_{\text{+1}}$

$(1,1)_{\text{-.04}} \rightsquigarrow (2,1)_{\text{-.04}} \rightsquigarrow (3,1)_{\text{-.04}} \rightsquigarrow (3,2)_{\text{-.04}} \rightsquigarrow (4,2)_{\text{-1}}$.

- The information about the award received in each visited state is used to learn the expected value function $U^{\pi}(s)$ for the non-terminal states s

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t)\right]$$

(Model-free) → Reward, transition

Direct Value Estimation

- The value of a state is the expected total reward from that state onwards (expected reward-to-go).
- Each trial provides a sample of this quantity for each state visited.

For the first trial/episode

For (1, 1) we get a sample total reward is 0.72

For (1, 2) we get 2 samples of total reward 0.76 and 0.84

For (1,3) we get 2 samples of total reward 0.80 and 0.88

$\lambda = 1$

$1 \to \lambda \cdot 04 \to \lambda^2 \cdot 04$

$1 - 7(.04$

$= 1 - 0.28$

First.

$$(1,1)_{-.04} \rightsquigarrow (1,2)_{-.04} \rightsquigarrow (1,3)_{-.04} \rightsquigarrow (1,2)_{-.04} \rightsquigarrow (1,3)_{-.04} \rightsquigarrow (2,3)_{-.04} \rightsquigarrow (3,3)_{-.04} \rightsquigarrow (4,3)_{+1}$$
$$(1,1)_{-.04} \rightsquigarrow (1,2)_{-.04} \rightsquigarrow (1,3)_{-.04} \rightsquigarrow (2,3)_{-.04} \rightsquigarrow (3,3)_{-.04} \rightsquigarrow (3,2)_{-.04} \rightsquigarrow (3,3)_{-.04} \rightsquigarrow (4,3)_{+1}$$
$$(1,1)_{-.04} \rightsquigarrow (2,1)_{-.04} \rightsquigarrow (3,1)_{-.04} \rightsquigarrow (3,2)_{-.04} \rightsquigarrow (4,2)_{-1} .$$

0.76 0.8 0.84 0.88

→ get the transition rewards → utilities.

- We keep a running average of the value estimates for each state
- This method assumes that the estimate of value of each state is independent of the other states!
- But in fact, the values obey the Bellman's equations for a fixed policy

$$U^{\pi}(s) = R(s) + \gamma \sum_{s'} P(s' \mid s, \pi(s)) U^{\pi}(s')$$

Adaptive Dynamic Programming

- The agent learns the transition model of the environment and uses the simplified value iteration process to update the utility estimates after each change to the learned model

$$U_{i+1}(s) \leftarrow R(s) + \gamma \sum_{s'} P(s' \mid s, \pi_i(s)) U_i(s')$$

- We assume the environment to be fully observable and represent the transition model as a table of probabilities

function PASSIVE-ADP-AGENT($percept$) **returns** an action

inputs: $percept$, a percept indicating the current state s' and reward signal r'

persistent: π, a fixed policy

mdp, an MDP with model P, rewards R, discount γ

U, a table of utilities, initially empty

N_{sa}, a table of frequencies for state–action pairs, initially zero

$N_{s'|sa}$, a table of outcome frequencies given state–action pairs, initially zero

s, a, the previous state and action, initially null

if s' is new **then** $U[s'] \leftarrow r'$; $R[s'] \leftarrow r'$

if s is not null **then**

  increment $N_{sa}[s, a]$ and $N_{s'|sa}[s', s, a]$

  for each t such that $N_{s'|sa}[t, s, a]$ is nonzero **do**

    $P(t \mid s, a) \leftarrow N_{s'|sa}[t, s, a] \,/\, N_{sa}[s, a]$

$U \leftarrow$ POLICY-EVALUATION(π, U, mdp)

if s'.TERMINAL? **then** $s, a \leftarrow$ null **else** $s, a \leftarrow s', \pi[s']$

return a

Temporal Difference Learning

- Consider a transition (1,3) to (2,3) in the second trial

$$(1,1)_{-.04} \rightsquigarrow (1,2)_{-.04} \rightsquigarrow (1,3)_{-.04} \rightsquigarrow (1,2)_{-.04} \rightsquigarrow (1,3)_{-.04} \rightsquigarrow (2,3)_{-.04} \rightsquigarrow (3,3)_{-.04} \rightsquigarrow (4,3)_{+1}$$
$$(1,1)_{-.04} \rightsquigarrow (1,2)_{-.04} \rightsquigarrow (1,3)_{-.04} \rightsquigarrow (2,3)_{-.04} \rightsquigarrow (3,3)_{-.04} \rightsquigarrow (3,2)_{-.04} \rightsquigarrow (3,3)_{-.04} \rightsquigarrow (4,3)_{+1}$$
$$(1,1)_{-.04} \rightsquigarrow (2,1)_{-.04} \rightsquigarrow (3,1)_{-.04} \rightsquigarrow (3,2)_{-.04} \rightsquigarrow (4,2)_{-1} .$$

- Suppose that as a result of the first trial, the utility estimates are

$U^\pi(1,3) = 0.84$ and $U^\pi(2,3) = 0.92$

- If this transition occurred all the time, we would expect the utilities to obey

$$U^\pi(1,3) = -0.04 + U^\pi(2,3)$$

- That is, $U^\pi(1,3) = 0.88$

- So, in general we can apply the following update to $U^{\pi}(s)$

$$U^{\pi}(s) \leftarrow U^{\pi}(s) + \alpha(R(s) + \gamma U^{\pi}(s') - U^{\pi}(s))$$

- When using such an update the Temporal Differencing method does not need a transition model.

function PASSIVE-TD-AGENT(*percept*) **returns** an action

inputs: *percept*, a percept indicating the current state s' and reward signal r'

persistent: π, a fixed policy

U, a table of utilities, initially empty

N_s, a table of frequencies for states, initially zero

s, a, r, the previous state, action, and reward, initially null

if s' is new **then** $U[s'] \leftarrow r'$

if s is not null **then**

increment $N_s[s]$

$U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma\, U[s'] - U[s])$

if s'.TERMINAL? **then** $s, a, r \leftarrow$ null **else** $s, a, r \leftarrow s', \pi[s'], r'$

return a

No transition model.

$\alpha(n) \downarrow$ if $n \uparrow$

look-ahead Q-table

- We can also apply the temporal difference method to the Q learning algorithm.
- The Q learning training rule calculates the training value for $\hat{Q}(s_t, a_t)$ in terms of the values for $\hat{Q}(s_{t+1}, a_{t+1})$ where s_{t+1} is the result of applying the action a_t to the state s_t
- Let $Q^{(1)}(s_t, a_t)$ denote the training value calculated by this one-step lookahead.

$$Q^{(1)}(s_t, a_t) \equiv r_t + \gamma \max_a \hat{Q}(s_{t+1}, a)$$

$\max_a \hat{Q}(s_{t+1}, a)$: $U(s_{t+1})$

γ: discount factor

$s_t \to s_{t+1} \to s_{t+2}$

r_t . r_{t+1} U.

(s_t, a_t) . s_{t+1} U.

- Similarly we can apply multiple step lookahead

$$Q^{(2)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 \max_a \hat{Q}(s_{t+2}, a)$$

$$Q^{(n)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \ldots + \gamma^{(n-1)} r_{t+n-1} + \gamma^{(n)} \max_a \hat{Q}(s_{t+n}, a)$$

- These alternative training estimates can be blended as

$$Q^{\lambda}(s_t, a_t) = (1 - \lambda) \left[Q^{(1)}(s_t, a_t) + \lambda Q^{(2)}(s_t, a_t) + \lambda^2 Q^{(3)}(s_t, a_t) + \ldots \right]$$

- An equivalent recursive definition for Q^{λ} is

$$Q^{\lambda}(s_t, a_t) = r_t + \gamma \left[(1-\lambda) \max_a \hat{Q}(s_t, a_t) + \lambda Q^{\lambda}(s_{t+1}, a_{t+1}) \right]$$

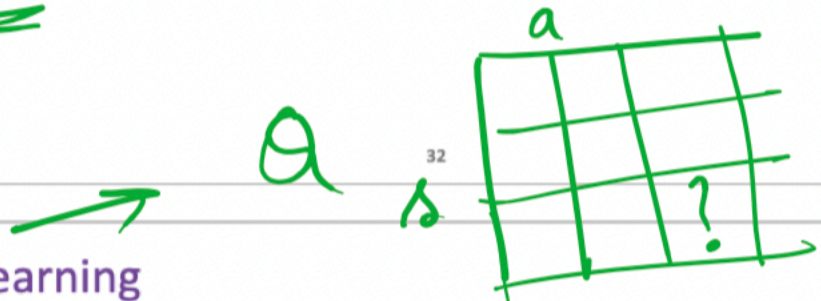
- Compare this with

$$Q^{(1)}(s_t, a_t) \equiv r_t + \gamma \max_a \hat{Q}(s_{t+1}, a)$$

As λ is increased the algorithm places increasing emphasis on discrepancies based on more distant lookahead.

- With $\lambda = 1$, only the observed r_{t+i} values are considered, with no contribution from the current $\hat{Q}$ estimate.
- When $\hat{Q} = Q$, the training values given by Q^{λ} will be identical for all values of λ such that $0 \leq \lambda \leq 1$
- When the agent follows an optimal policy for choosing actions, then Q^{λ} with $\lambda = 1$ will provide a perfect estimate for the true Q value, regardless of any inaccuracies in $\hat{Q}$

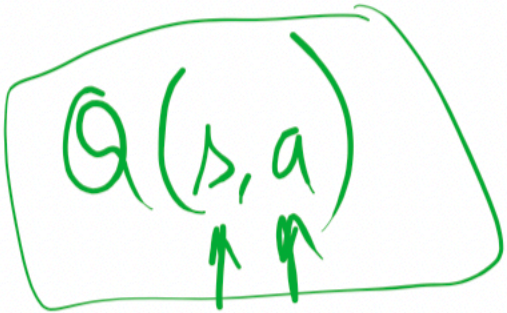
all look-ahead estimates



Limitations of Q learning

Limitations of Q learning

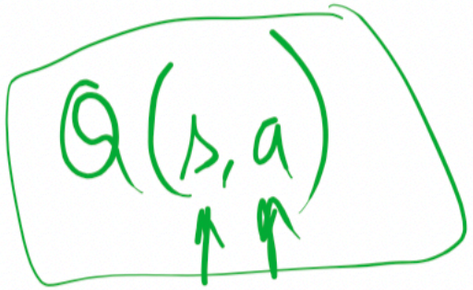
- Algorithms that implement the Q function in the form of explicit lookup table cannot generalise to unseen state-action pairs.
- The convergence proof of Q learning is applicable only when Q is represented as an explicit function.
- Also the convergence proof of Q learning requires every possible state-action pair is visited infinitely often during training



Neural Network.

pair is visited infinitely often

during training



Neural Network.

Deep Reinforcement Learning

- Neural Network to implement the Q function
- We can encode the state s and action a as network inputs and train the network to output the target values of $\hat{Q}$ given by the training rules.
- We can train separate networks for each action, using the state as input and $\hat{Q}$ as output.
- We can train one network with state as input, but with $\hat{Q}$ output for each action.

Loss Function for Deep Q Learning

Loss Function for Deep Q Learning

- As we make moves, particularly in the early learning stages, we do not know whether they are good or bad.

We don't have any ground truth for Q values.

- However, we do know the following:

  On average

$$R(s,a) + \gamma \max_{a'} \hat{Q}(s',a')$$

better than $\hat{Q}(s,a)$.

is a more accurate estimate of $Q(s,a)$ than the current value, because we are looking one move ahead.

deterministic

- So we can use the loss

$$\left[\hat{Q}(s,a) - (R(s,a) + \gamma \max_{a'} Q(s',a'))\right]^2$$

- The first term is the Q calculated by the network and the second term is the actual reward for the next action plus the Q value one step in the future.
- This difference in the brackets is called the temporal difference error TD(0).
- If we looked 2 steps into the future, it will be TD(1).

- In deep-Q learning we make one move at a time.

- Having made the move and received the reward we end up in a new state. This improves our knowledge of the current local environment.

guides the loss.

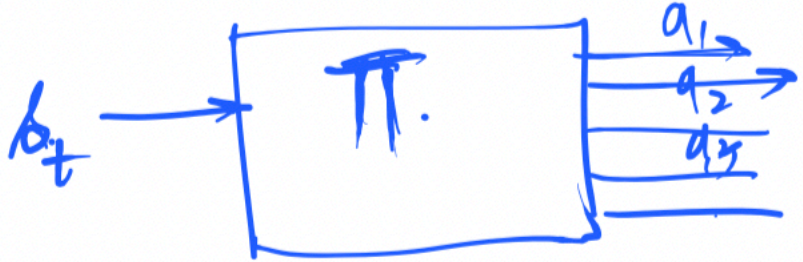
- The loss is the difference between what was predicted (e.g. the Q function) on the basis of the old knowledge and what, in fact, happened.

↓ after taking the action.

GT is never available.

Policy Gradient Method

- We play an entire episode using the current policy.
- The accumulated reward tells us how good the current sequence of actions was.



action
$P(a_t|s_t)$
loss.

$\pi(s)$

Act by selecting an action from the prob distribution

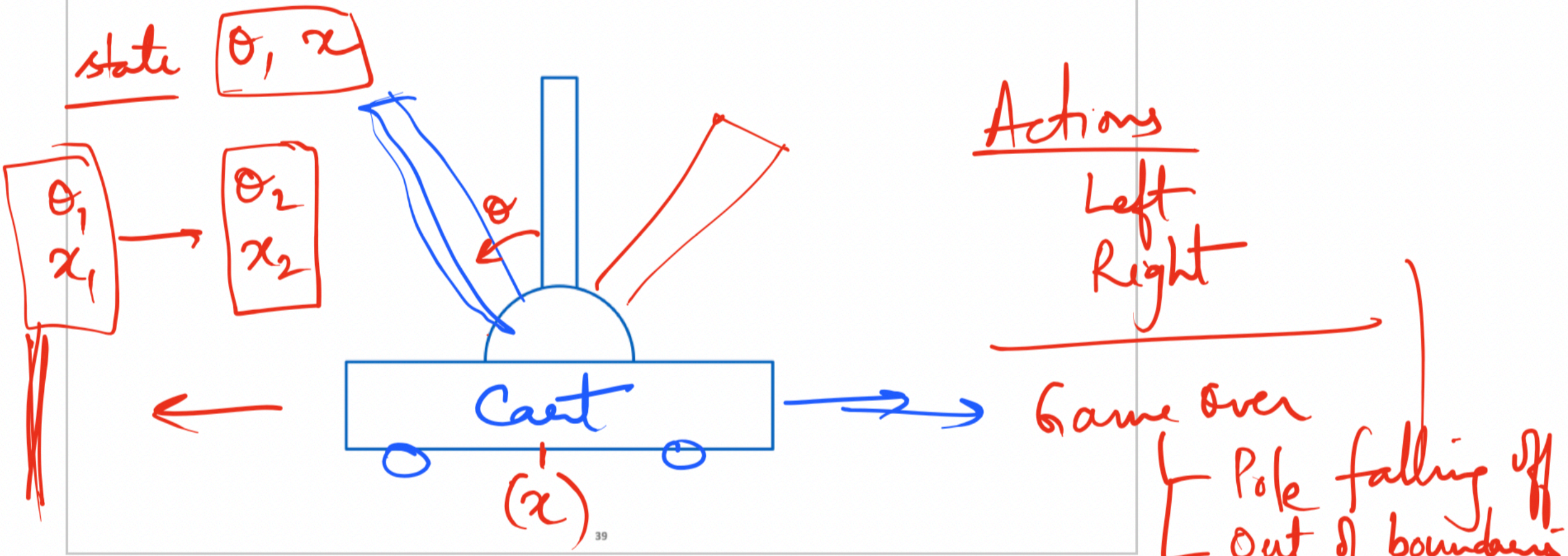
s_0 s_1 s_2 - - - s_T
R R R

$D(s\ a)$

discounted reward.

discounted reward

A cart Pole



Cart Pole

- The state is the position and the velocity of the cart and the pole head.
  - The number of possible states is infinite.
- The game is over if the cart moves too far to the right or to the left OR the pole head moves too far from the perpendicular.
- We get one unit of reward for every move we make before failing.
- The goal is to keep the cart and the pole well positioned for as long as possible.

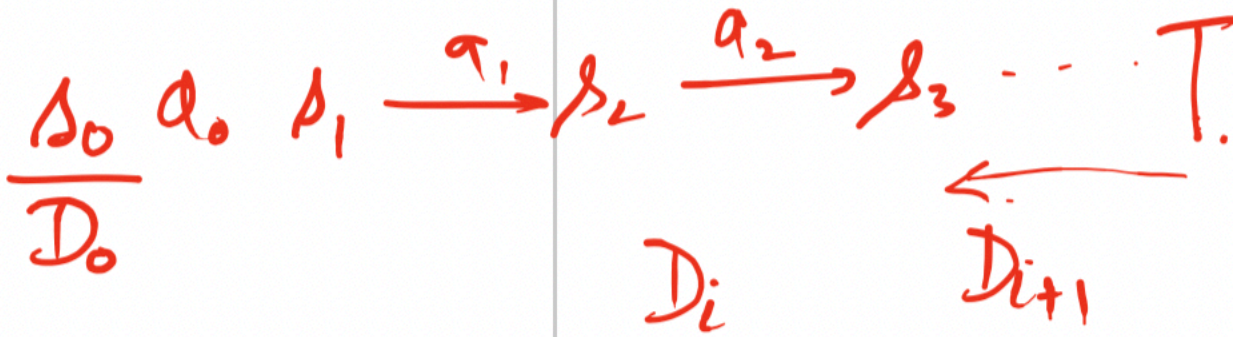
Q-table learning is not possible

- We need to learn a NN to approximate the Q function
- In deep-Q learning we
    - make one move at a time
    - gain some knowledge about the environment
    - formulate the loss based on the gained knowledge
- In Policy-gradient method, we play an entire episode (till the game ends) without making any modification to our network.
- Actions are chosen randomly as per the probability distribution arrived from the Q function

- We compute the discounted reward for the first state $D_0(\mathbf{s}, \mathbf{a})$ when it is followed by all the states and actions we just tried out

vectors

$$D_0(\mathbf{s}, \mathbf{a}) = \sum_{t=0}^{n-1} \gamma^t \, R(s_t, a_t, s_{t+1})$$



- If we took n steps we can compute the future discounted reward for any of the state-action combinations s_i, a_i from the recurrence relation

$$D_n(\mathbf{s}, \mathbf{a}) = 0 \qquad D_i(\mathbf{s}, \mathbf{a}) = R(s_i, a_i, s_{i+1}) + \gamma D_{i+1}(\mathbf{s}, \mathbf{a})$$

- The discounted reward depends on the quality of the actions performed.
- A good loss function can be

$$L(\mathbf{s}, \mathbf{a}) = \sum_{t=0}^{n-1} D_t(\mathbf{s}, \mathbf{a})(-\log P(a_t \mid s_t))$$

- The basic architecture to learn the Q function using the above loss formulation is known as REINFORCE.
- The policy can be trivially extracted from the action probability matrix produced by the NN.

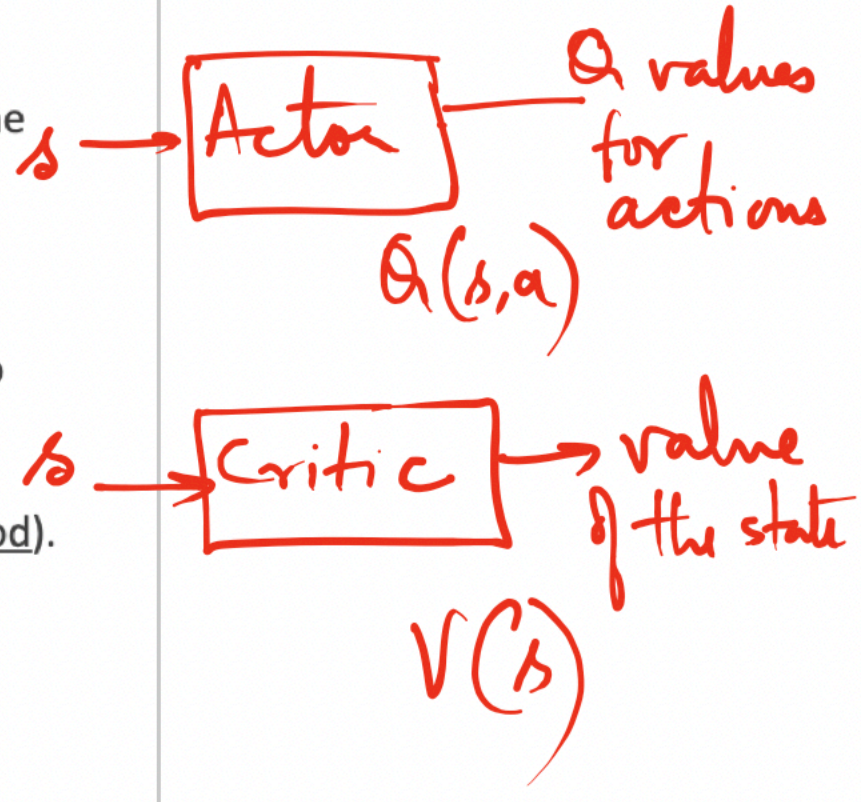
- The REINFORCE method makes the parameter changes less often compared to the deep Q-learning method.
- But REINFORCE makes better changes because we are computing the actual discounted reward.

Actor-Critic Method

- Both Q learning and REINFORCE make use of a single NN for approximating the Q function.
- Such RL programs are called actor methods.
- Another class of RL programs, known as actor-critic methods make use of two NN subcomponents, each with its own loss function.
- One such actor-critic method is the advantage actor-critic method (a2c method).

a a c a2c

$$V(s) = \max_a Q(s,a)$$



- The advantage of a state-action pair is the difference between the state-action Q value and the state's value

$$A(s,a) = Q(s,a) - V(s)$$

- Advantage is a negative number
- For good actions A has a smaller magnitude
- The loss that a2c incurs from exploring a sequence of actions from a start state to the end of a game is as follows:

$$L_A(\mathbf{s},\mathbf{a}) = \sum_{t=0}^{n-1} A(s_t,a_t)\,(-\log P(a_t \mid s_t))$$

- Note that REINFORCE encourages actions that lead to larger reward.
- The actor loss L_A loss encourages actions that are better than alternative actions

Network tries to make the 2nd term large positive

- Note that REINFORCE encourages actions that lead to larger reward.
- The actor loss L_A loss encourages actions that are better than alternative actions from the same state.
- The actor component of the NN is used to estimate Q
- The critic component of the NN is used to estimate the value function V
- The critic loss L_C is the is the quadratic loss on the disparity between the actual rewards and the output of the critic component of the NN

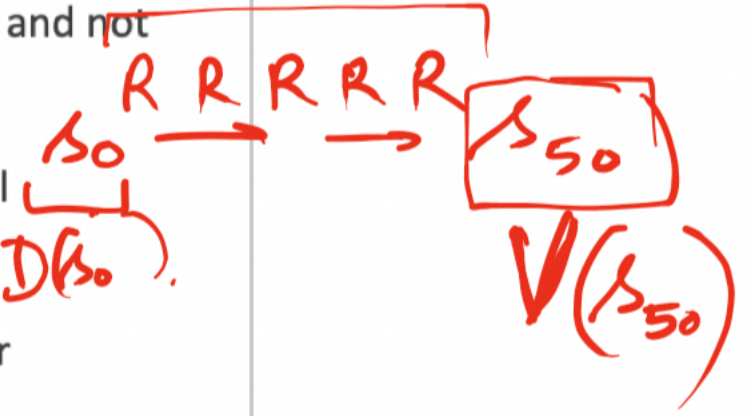
⇒ make action a_t unlikely

critic → $\hat{V}(s)$

L_A loss has got less variance compared to loss that uses $D(s,a)$

Improvement over REINFORCE

- A2C makes use of a trick to update the model's parameters much earlier and not waiting till the end of the game.
- The game execution is paused every, say 50, actions to update the model parameters.
- A2C allows us to make an estimate of Q values by simply adding together
  - (a) the actual rewards we accumulated over the last 50 moves
  - (b) the V value of the state we end up in
- A2C then restarts from scratch from the 51st move and applies the update again after another 50 moves.



- Multiple games can be played simultaneously.
- This is equivalent to using a batch of examples to take advantage of parallel processing hardware.