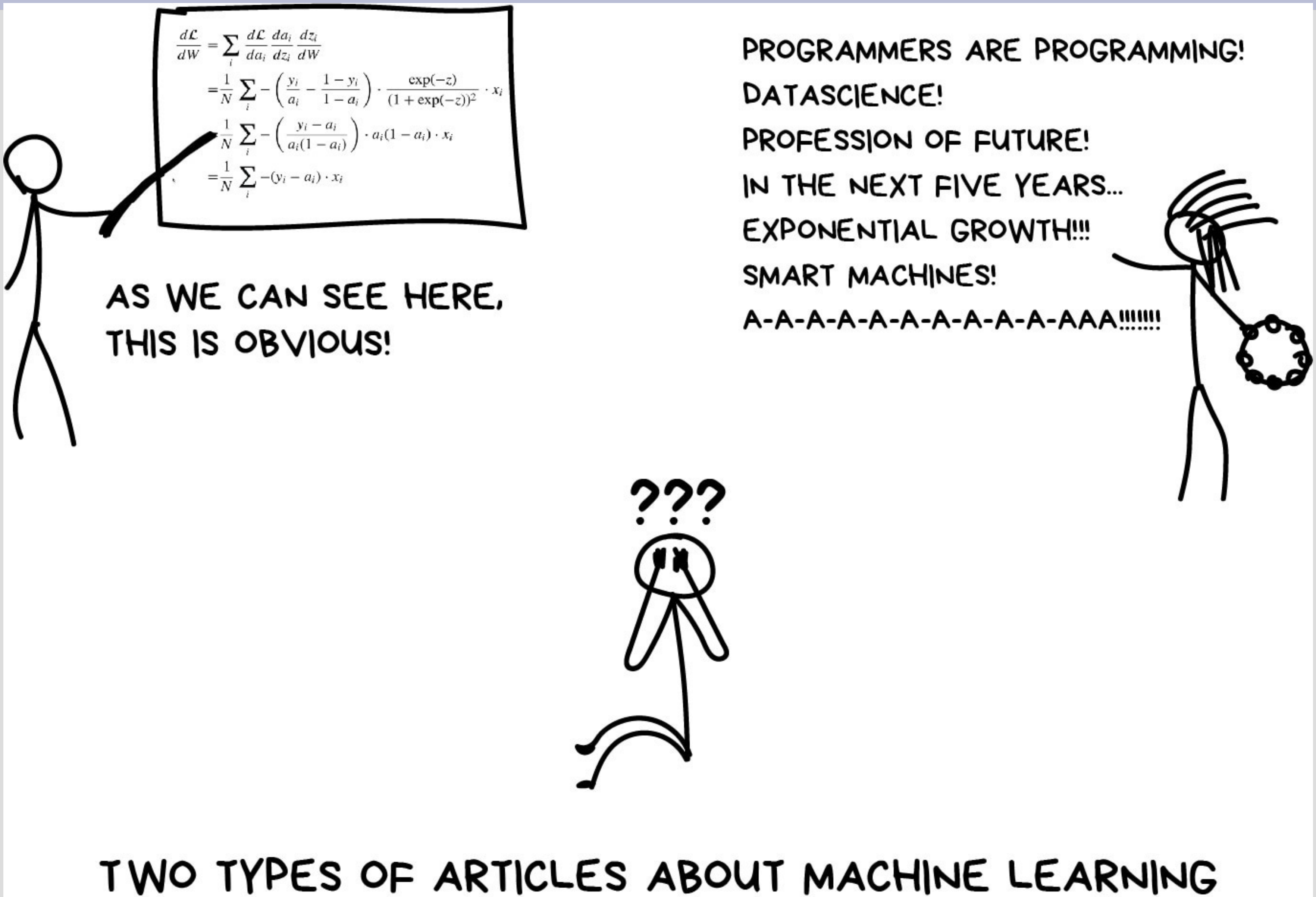
Active Learning (Ch. 21.3-21.5)

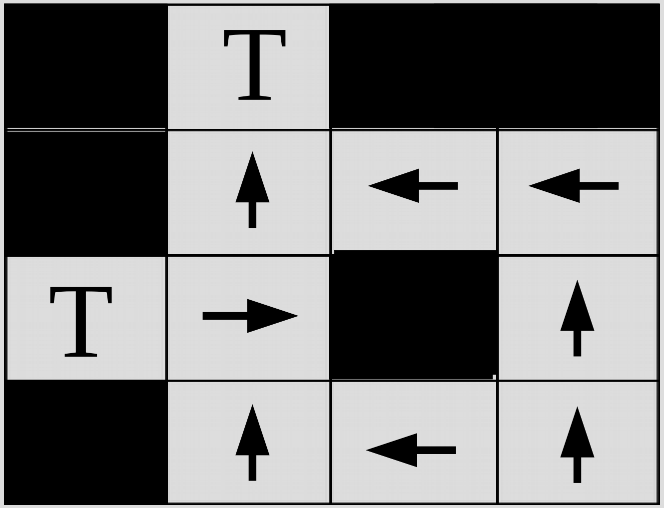


Reinforcement Learning

Last time we looked at passive reinforcement learning (i.e. policy/actions decided already)

We used an MDP (but they are pretty general with “states” and “actions”)

Assume arrows, learn action outcomes & estimate utility



Reinforcement Learning

This time, we need to find the best actions (active learning) in addition to estimating the utility along the way

This may seem much more difficult, but it can be reduced down to one additional part:

Balancing exploitation and exploration

taking the greedy choice
(best action known)

trying new actions to see
if they are any good

Reinforcement Learning

The balance between exploitation and exploring is quite delicate

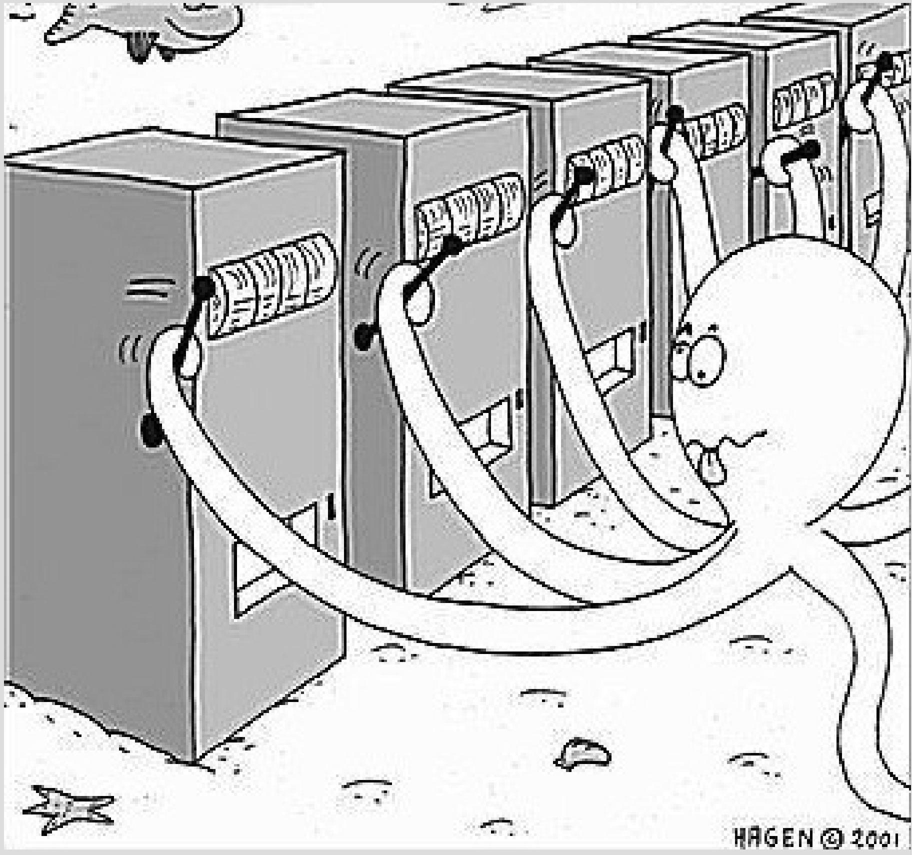
If the agent only exploits, once any “solution” is found they will keep doing it (even if bad)

For example, comparing two webpages with a single tab

Multi-Armed Bandit

On the other hand, an agent that explores 100% of the time will have a great idea of the problem, but will cost a lot

One of the famous ways about thinking of this is the multi-armed bandit (i.e. multiple slot machines)



Multi-Armed Bandit

Suppose you walk into a casino see 3 types of slow machines (low, med and high risk)

Suppose playing all three machines will make you money overall (not realistic)

Do you play the one that gave the best average outcome so far?(exploitation, probably “low”)

Do you play each 1/3 of the time? (explore)

Multi-Armed Bandit

The common way to measure this is regret:

$$\underbrace{N \cdot \mu^*}_{\text{best option N times}} \quad - \quad \underbrace{\sum_{i=1}^{N} r_i}_{\text{actual rewards}}$$

In other words, if we play the 3 machines N times... we want to get as close to the possible maximum reward if we knew machine payouts

(i.e. minimize equation above... which is hard to do exactly so we will approximate)

Multi-Armed Bandit

The theory is a bit easier in the case where N=∞ (i.e. can play forever)

In this case you want the regret per round to be zero: $\lim_{N \to \infty} \frac{N \cdot \mu^* - \sum_i r_i}{N} = 0$

This means that you have to play each slot machine an infinite amount of times (or else there is a non-zero probability your estimates were just “unlucky” for some machine)

Multi-Armed Bandit

A fairly simple strategy (that does **not** accomplish this) is called ε-greedy:

(1) Generate random number [0,1]
(2) If random < ε: play random machine
(3) Else: play best machine

Since you will play each machine (with 3 machines) an infinite amount of times: $\frac{1}{3} \cdot \sum_{i=1}^{\infty} \epsilon = \infty$
... but the probability of play suboptimal is ε

Multi-Armed Bandit

A slight modification of ε-greedy can cause the regret per round to be zero:

Instead of having ε as a fixed value, have ε decrease over time (like ε/i for round i)

Each machines is still played infinite: $\frac{\epsilon}{3} \cdot \sum_{i=1}^{\infty} \frac{1}{i} = \infty$

Yet per round (lots of math): $\lim_{N \to \infty} \frac{N \cdot \mu^* - \sum_i r_i}{N} = 0$

Multi-Armed Bandit

While ε-greedy with decreasing ε has better theoretical bounds, in practice it is quite often slow to converge (exploits a bit too much)

Quite often basic ε-greedy is used or...

SoftMax:

$$p(\text{pick machine } i) = \frac{e^{\hat{R}_i}}{\sum_j e^{\hat{R}_j}}$$

$\hat{R}$ is the estimated reward up to this point

probabilistic... will pick best exponentially more

Upper Confidence Bound (UCB):

N=total times played (so far)

$$\text{pick machine } i : \arg\max_i \left(\hat{R}_i + \sqrt{\frac{2 \ln N}{n_i}}\right)$$

n_i=times played machine i (so far)

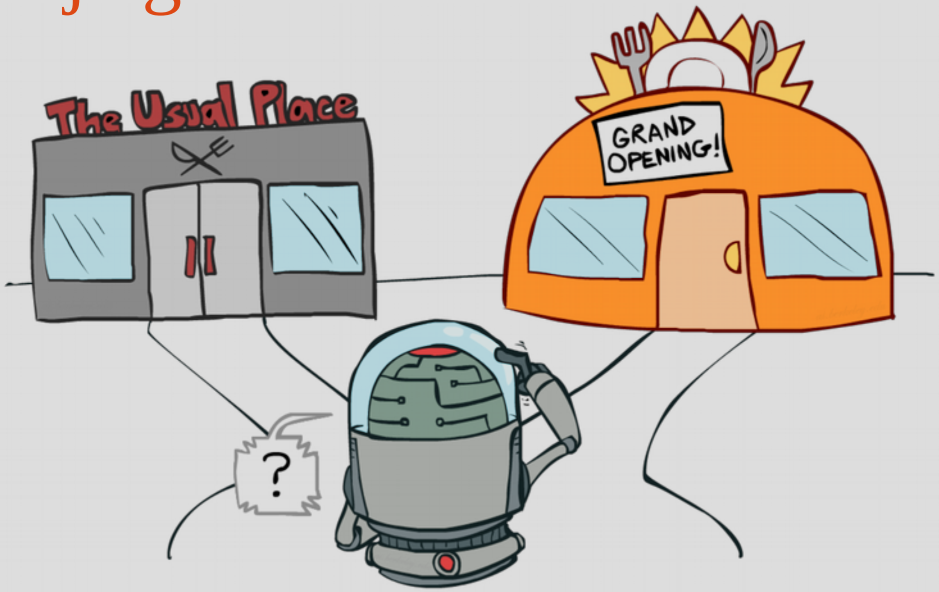
Multi-Armed Bandit

Multi-Armed Bandit problem research is quite deep... so we will stop here

Here are some good links to info:

https://lilianweng.github.io/lil-log/2018/01/23/the-multi-armed-bandit-problem-and-its-solutions.html

https://sudeepraja.github.io/Bandits/



Reinforcement Learning

Now that we can balance exploit & explore, we can modify the two passive algorithms:

Specifically, Adaptive Dyn. Prog. (ADP):

Count transitions to estimate P(s'|s,a)

Use Bellman: $U(s) = \underbrace{R(s)}_{\text{rewards}} + \gamma \cdot \sum_{s'} \underbrace{P(s'|a,s)}_{\text{transition}} \cdot U(s')$

solve system linear equations for
all states when P(s'|s,a) changes

Temporal-difference (TD):

Localized Bellman (estimate utility directly)

$$U(s) \leftarrow U(s) + \alpha \cdot (R(s) + \gamma \cdot U(s') - U(s))$$

Recap: ADP

So given the same first example:

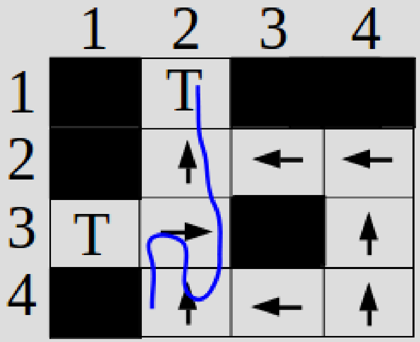
$(4,2)_{-1} \uparrow (3,2)_{-1} \rightarrow (4,2)_{-1} \uparrow (3,2)_{-1} \rightarrow (2,2)_{-1} \uparrow (1,2)_{50}$

We'd estimate the following transitions:

(4,2) + ↑ = 100% ↑ (2 of 2)

(3,2) + → = 50% ↑, 50% ↓

(2,2) + ↑ = 100% ↑



... and we can easily see the rewards from sequence, so policy/value iteration time!

better as actions fixed no iteration

Modified ADP

Unlike before, we have to pick the arrows first... but then it reduces down to past ADP

To choose arrows, we just need any balance between exploit & explore into Bellman:

$$U(s) \leftarrow R(s) + \gamma \cdot \max_a f(utility, explore)$$

value iteration update
start initial guesses high
to encourage exploration

utility=normal Bellman update

$$utility = \sum_{s'} P(s'|s,a) \cdot U(s')$$

... where f(utility, explore) can be any multi-armed bandit function

(book suggests something simple)

$$f(u,e) = \begin{cases} R^+\text{, if visited less than k times} \\ utility\text{, otherwise} \end{cases}$$

Modified ADP

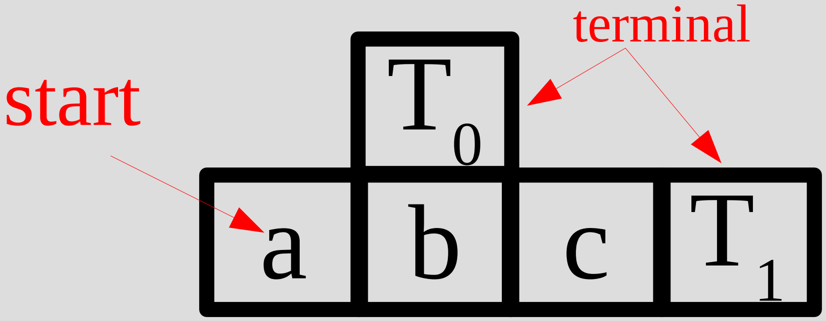
Before we were calling the inputs to the bandit problems “rewards”

In the MDP setting we deal with multiple rewards (i.e. utilities), but same idea “expected utility” instead of “average reward”

The theoretical bounds no longer apply with multiple steps, so approximate methods are often used (ones we discussed)

Modified ADP

Let's do a simple MDP where we have run it a bit and have P(s'|s,a) as shown: (for s=b)



Tried → 10 times:

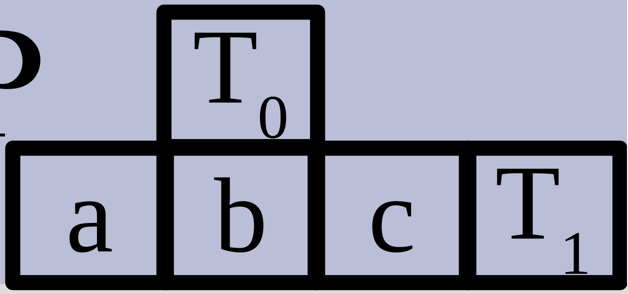
$P(c|b,\rightarrow) = 0.8$

$P(T_0|b,\rightarrow) = 0.2$

Tried ↑ 2 times:

$P(T_0|b,\uparrow) = 1.0$

Modified ADP



Tried → 10 times:

$P(c|b,\rightarrow) = 0.8$

$P(T_0|b,\rightarrow) = 0.2$

Tried ↑ 2 times:

$P(T_0|b,\uparrow) = 1.0$

bit off as UCB made for [0,1]... meh (can rescale)

Assume we found $U(T_0) = -1$, $U(c)=0.7$, and we're at "b" in another training example

$$\arg\max_i E[utility_i] + \sqrt{\frac{2\log N}{n_i}}$$

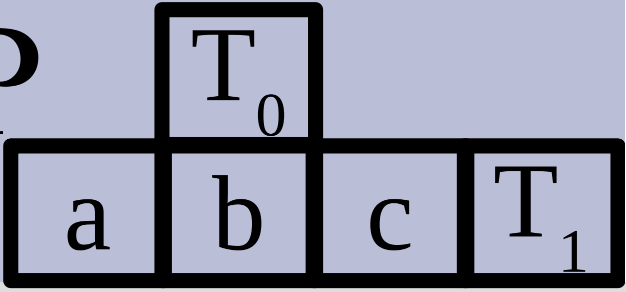
If we use the UCB bandit trade-off:

larger, so go →

Value for $(b,\uparrow) = (1.0\cdot -1) + \sqrt{\frac{2\ln 12}{2}} = 0.576$

Value for $(b,\rightarrow) = (0.8\cdot 0.7 + 0.2\cdot -1) + \sqrt{\frac{2\ln 12}{10}} = 1.06$

Modified ADP



Thus we do (b , →) and say we end up in T_0:

Tried → 11 times:

$P(c|b, \rightarrow) = 0.73$

$P(T_0|b, \rightarrow) = 0.27$

Tried ↑ 2 times:

$P(T_0|b, \uparrow) = 1.0$

exploration function “f” wanted to go right

We then update utility of b:

$$U(s) \leftarrow R(s) + \gamma \cdot \max_a f(utility, explore)$$

$$U(b) \leftarrow R(b) + \gamma \cdot (0.73 \cdot 0.7 + 0.27 \cdot -1)$$

... and run value iteration a bit (has seed value)

Q-Learning = Modified TD

Next we will modify the TD update:

$$U(s) \leftarrow U(s) + \alpha \cdot (R(s) + \gamma \cdot U(s') - U(s))$$

This is commonly called <u>q-learning</u> and uses a Q-function that is **very** related to utility:

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

Q-functions defined in terms of both a state and action (pair)

This modifies Bellman equations to be:

$$Q(s,a) = R(s) + \gamma \cdot \sum_{s'} P(s'|s,a) \cdot \underbrace{\max_{a'} Q(s',a')}_{\text{just } U(s') \text{ by def}}$$

“max” missing in Bellman,
as used later to get utility
same as: r+max(a) = max(r+a)

Q-Learning = Modified TD

Thus we change our update... “old” TD one:

$$U(s) \leftarrow U(s) + \alpha \cdot (R(s) + \gamma \cdot U(s') - U(s))$$

“New” Q-learning one:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot (R(s) + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a))$$

... sure...

Once again we just need to incorporate the bandit trade-off (exploit vs. explore)

Q-Learning = Modified TD

This makes the overall algorithm:

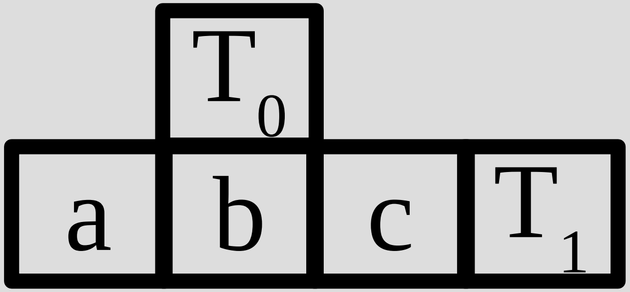
(0) Initialize Q(s,a) to anything (for all s & a)
(1) Pick action based on Multi-Armed Bandit
(2) Once you have action, use Q-update
on the state that you just left:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot (R(s) + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a))$$

(3) Repeat from step 1 until end

Q-Learning = Modified TD

Let’s go back to our simple example:



... but this time let’s do ε-greedy with $\varepsilon=0.05$

Suppose we have Q-values as:

$Q(a, \rightarrow) = 1$

$Q(a, \uparrow) = 0.5$

$Q(b, \rightarrow) = 1.5$

$Q(b, \uparrow) = -0.8$

Q-Learning = Modified TD

1.15

Q(a, →) = 1 Q(b, →) = 1.5

Q(a, ↑) = 0.5 Q(b, ↑) = -0.8

Assume R(a) = -0.2

Start in “a” and generate random number:

0.472 > 0.05, so take “greedy” choice (a, →)

$$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot (R(s) + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a))$$

Say we end up in “b”, then (α=0.5, γ=1):

$$\begin{aligned} Q(a,\rightarrow) &= Q(a,\rightarrow) + \alpha \cdot (R(a) + \gamma \cdot \max_{x} Q(b,x) - Q(a,\rightarrow)) \\ &= 1 + \alpha \cdot (R(a) + \gamma \cdot Q(b,\rightarrow) - 1) \\ &= 1 + 0.5 \cdot (-0.2 + 1 \cdot 1.5 - 1) = 1.15 \end{aligned}$$

Q-Learning = Modified TD

Q(a, →) = 1.15   Q(b, →) = 1.5   -1

Q(a, ↑) = 0.5   Q(b, ↑) = -0.8

Assume R(b) = -0.2

Now we in “b” and generate random number:

0.028 < 0.05, so “explore” (random action=↑)

$$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot (R(s) + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a))$$

Say we end up in “T_0”, then (α=0.5, γ=1):

$$\begin{aligned} Q(b,\uparrow) &= Q(b,\uparrow) + \alpha \cdot (R(b) + \gamma \cdot \max_{x} Q(T_0,x) - Q(b,\uparrow)) \\ &= -0.8 + \alpha \cdot (R(b) + \gamma \cdot Q(T_0, terminal) - (-0.8)) \\ &= -0.8 + 0.5 \cdot (-0.2 + 1 \cdot -1 - (-0.8)) = -1 \end{aligned}$$

Q-Learning vs. SARSA

A slightly different update is called SARSA (state-action-reward-state’-action’):

$$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot (R(s) + \gamma \cdot Q(s',a') - Q(s,a))$$

bye, bye max

(Compared to original:)

$$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot (R(s) + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a))$$

This update is a bit different as:

Q-learn: in state, need find action, result state

SARSA: in state, need find action, result state **and next action**

Q-Learning vs. SARSA

In the “exploitation” phase of the bandit problem, this should be the same

However, in “exploration” things differ as:
Q-learn: assumes you will take “best” action
SARSA: update based on action actually taken

Given you know the Q(s,a) values, you can decide what policy you want to follow (randomly introducing exploration)

Q-Learning vs. SARSA

SARSA updates Q(s,a) values based on this policy you decide you want to follow (thus called on-policy)

Q-learning sorta ignores the policy you are following (off-policy) and still updates off the best action (even if that is not next action)

SARSA works better if you are not in full control of the policy (like bandit explore)

Q-Learning vs. SARSA

1.15
Q(a, →) = 1 Q(b, →) = 1.5 -1
Q(a, ↑) = 0.5 Q(b, ↑) = -0.8

Assume R(a) = -0.2 = R(b), $\alpha=0.5$, $\gamma=1$

In our Q-learning, we updated the Q(s,a) values as shown above (previous slides)

SARSA would disagree on the update for Q(a, →), as it would find max = (b, →), but we did (b, ↑) due to ε-greedy exploration

Q-Learning vs. SARSA

1.15

Q(a, →) = 1 Q(b, →) = 1.5 -1

0

Q(a, ↑) = 0.5 Q(b, ↑) = -0.8

Assume R(a) = -0.2 = R(b), α=0.5, γ=1

$$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot (R(s) + \gamma \cdot Q(s',a') - Q(s,a))$$

Thus SARSA would do:

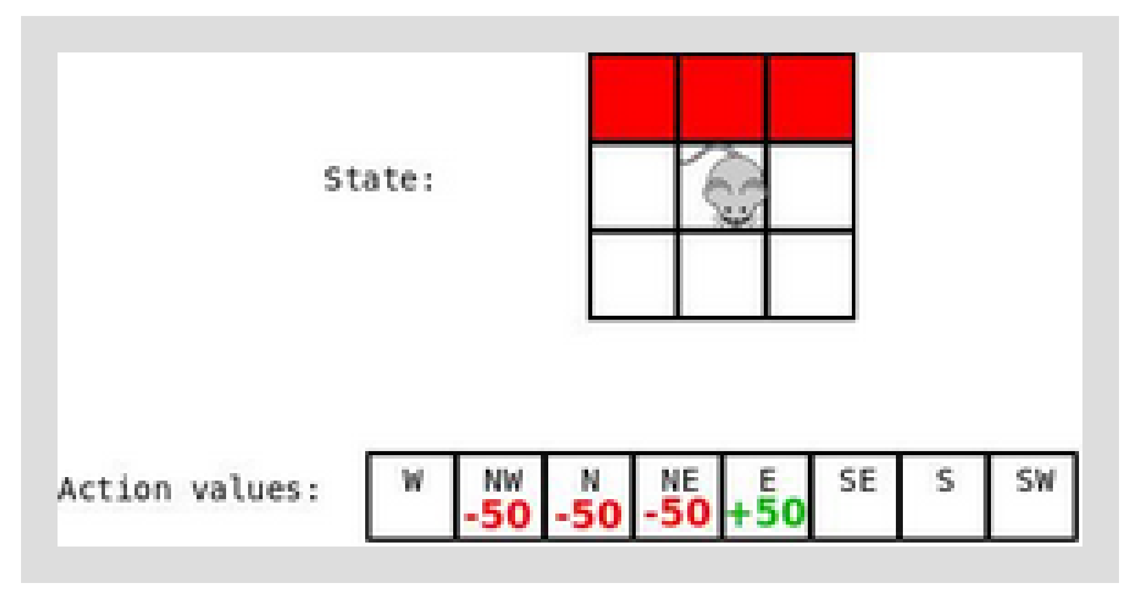
$$\begin{aligned} Q(a,\rightarrow) &= Q(a,\rightarrow) + \alpha \cdot (R(a) + \gamma \cdot Q(b,\uparrow) - Q(a,\rightarrow)) \\ &= 1 + \alpha \cdot (R(a) + \gamma \cdot -0.8 - 1) \\ &= 1 + 0.5 \cdot (-0.2 + 1 \cdot -0.8 - 1) = 0 \end{aligned}$$

... which is a bit more pessimistic

Q-Learning vs. SARSA

A simple mouse & cheese example is here which demonstrates difference graphically:

https://studywolf.wordpress.com/2013/07/01/reinforcement-learning-sarsa-vs-q-learning/



State:

Action values:

W	NW	N	NE	E	SE	S	SW
	-50	-50	-50	+50			