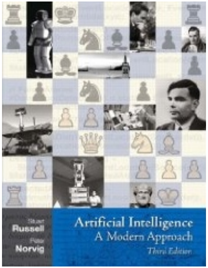
Reinforcement Learning 2

Oana Cocarascu & Helen Yannakoudakis

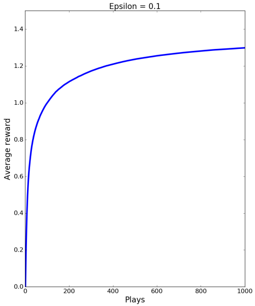
Department of Informatics
King's College London

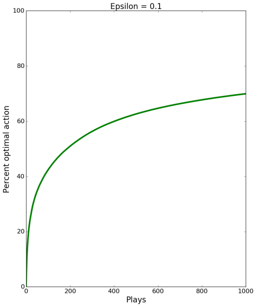
Today

- Adaptive dynamic programming
- Temporal difference learning
- Active reinforcement learning
- Q-learning
- SARSA
- Function approximation



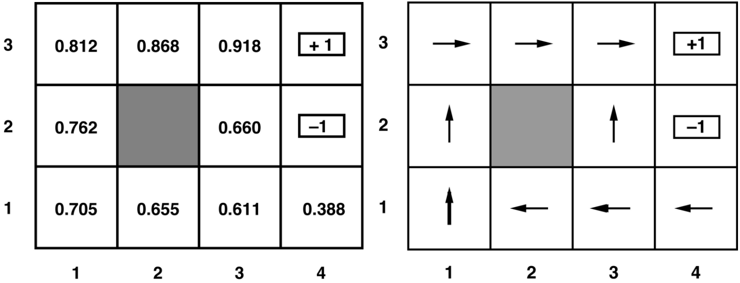
Recap - ϵ-greedy





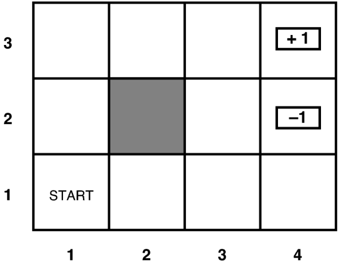
$$a_t = \begin{cases} a_t^* & \text{with probability} \quad 1-\epsilon \\ \textit{random action} & \text{with probability} \quad \epsilon \end{cases}$$

Recap - Value iteration

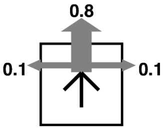


$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} \Pr(s'|s,a) U_i(s')$$

Recap - Direct utility estimation



(a)



(b)

$$(1,1)_{-0.04} \xrightarrow{Up} (1,2)_{-0.04} \xrightarrow{Up} (1,3)_{-0.04} \xrightarrow{Right}$$
$$(1,2)_{-0.04} \xrightarrow{Up} (1,3)_{-0.04} \xrightarrow{Right} (2,3)_{-0.04} \ldots$$

Problem with direct utility estimation

- Treats utilities of states as independent.
- But we know that they are connected:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a) U_i(s')$$

- Ignoring the connection means that learning may converge slowly.
- Another approach to utility estimation: adaptive dynamic programming (ADP).
  - Still doing passive reinforcement learning.
  - But doing it *smarter*.

Adaptive dynamic programming

- We can improve on direct utility estimation by applying a version of the Bellman equation.
- The utility of a state is the reward for being in that state plus the expected discounted reward of being in the next state, assuming that the agent chooses the optimal action:

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} \Pr(s'|s, a) U(s')$$

- For n states, there are n Bellman equations (one per state).
- We want to solve these simultaneous equations to find the utilities.
  - But the equations are nonlinear because of the *max* operator.

Adaptive dynamic programming

- In passive learning, we have π so we know what action we will carry out.
- Because of this, the *max* operator is removed so the equations become linear:

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

- Bellman states a constraint on utilities, but what does that mean in practice?
- Two approaches:
  1. Directly solve the Bellman equations
  2. Apply value iteration

Solving the Bellman equations

- The fixed policy version of the Bellman equation is:

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

- This is just a set of simultaneous equations that can be solved with a Linear Programming solver.
- Updates all the utilities of all the states where we have experienced the transitions.
- Note that updated values are *estimates*.
    - They are no better than the estimated values of utility and probability we had before.
    - We just get quicker convergence because the utilities are consistent.

Using value iteration

- Can also use value iteration to update the utilities we have for each state.
- Update until convergence using:

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

- Again, the results are still estimates, and no better than the estimates we got from direct estimation or solving the Bellman equations.

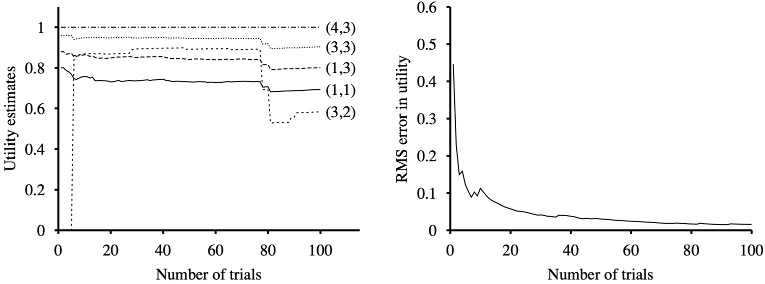
Adaptive dynamic programming

- In all cases:
  1. Direct utility estimation
  2. ADP: solving Bellman equations
  3. ADP: applying value iteration

  the quality of the utility estimates will depend on how well we have *explored* the space.
- Roughly, this is how many times we have encountered each state.

Adaptive dynamic programming

- Results:



- Typically quicker than direct utility estimation.
- At trial 80, the agent falls into the -1 terminal state (4,2) for the first time.
- Error is for $U(1,1)$.

Adaptive dynamic programming: solution

- Still passive learning, so a solution is as before:
- A list of states s_i.
- Each state has a utility estimate associated with it, $U(s)$.
- Each state has an action associated with it, $\pi(s)$.
- Each state action pair has a probability distribution:

$$P(S'|s, \pi(s))$$

over the states S' that it gets to from s by doing $\pi(s)$.

After learning

- To get the utilities, the agent started with a fixed policy, so it always knew what action to take.
- Having gotten the utilities, it could use them to choose actions.
    - Pick the action with the best expected utility in a given state.
- However, there is a problem with doing this.

Problems

- The transition model is a maximum likelihood estimate (just the sample average).
- Recall from previous lectures that these models tend to overfit.
- Maximum likelihood action selection can be dangerous.
  - Might not yet have experienced the bad effects of an action:

  - Maybe your autonomous car learnt that running a red light saves time.
- There is no way to be sure that the action the reinforcement learner is picking does not have possible bad outcomes.

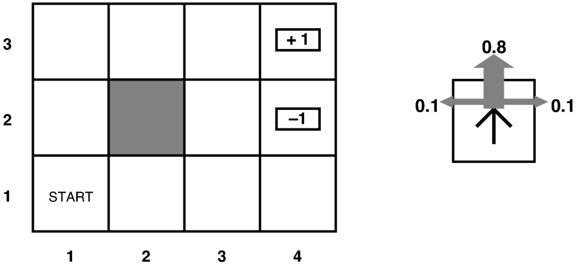
Solutions

1. As discussed in earlier lectures we can use better priors. Consider transitions to all reasonable states.
2. Can also learn the probability that a particular model (a set of probability and utility values) is true and make decisions based on:
   - either the expected value of actions across all models; or
   - the worst case outcome across the models.
3. Can try to ensure that the learner explores widely.

Temporal difference learning

- In the previous approach we used the fact that we are learning in the context of an MDP.
- Temporal difference learning uses Bellman (= constraints between states).
- Use the observed transitions to adjust the utilities of the states.
- Let's look at an example.

Temporal difference learning



(a) (b)

- Consider this run:

$$(1,1)_{-0.04} \xrightarrow{Up} (1,2)_{-0.04} \xrightarrow{Up} (1,3)_{-0.04} \xrightarrow{Right} (2,3)_{-0.04} \xrightarrow{Right}$$
$$(3,3)_{-0.04} \xrightarrow{Right} (3,2)_{-0.04} \xrightarrow{Up} (3,3)_{-0.04} \xrightarrow{Right} (4,3)_{+1}$$

Temporal difference learning

- Consider the transition from $(1,3)$ to $(2,3)$.

$$(1,1)_{-0.04} \stackrel{Up}{\rightarrow} (1,2)_{-0.04} \stackrel{Up}{\rightarrow} (1,3)_{-0.04} \stackrel{Right}{\rightarrow} (2,3)_{-0.04} \stackrel{Right}{\rightarrow}$$
$$(3,3)_{-0.04} \stackrel{Right}{\rightarrow} (3,2)_{-0.04} \stackrel{Up}{\rightarrow} (3,3)_{-0.04} \stackrel{Right}{\rightarrow} (4,3)_{+1}$$

- Assume that we have utility estimates:

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

(These values are from the run we considered last lecture.)

- These results should be linked by a Bellman-type update.

Temporal difference learning

- In other words, we should expect ($\gamma = 1$ to simplify calculations):

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

  and so $U^{\pi}(1,3) = 0.88$
- Currently have $U^{\pi}(1,3) = 0.84$
- So maybe the current estimate is too low.
- Can generalise the idea.

Temporal difference learning

- The temporal difference update for a transition from s to s' is:

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

- α is a learning rate.
  - Controls how quickly we update the utility when we have new information.
  - Like the learning rate in gradient descent.
- α is another parameter that is problem specific.
- The rule is called “temporal difference” (TD) because the update occurs between successive states.

Note

- The temporal difference update:

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

  looks rather similar to the computation that we used to update the estimated value of an action in the ϵ-greedy learner.
- (Slide 17, Reinforcement Learning 1 lecture)

Temporal difference learning

- Compare ADP

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

with TD

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

Temporal difference learning

- ADP can be read as a statement about the stopping condition:

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

- No change in values when both sides of the equation are equal.
- Connects the utility of s with that of ***all*** its successor states.

Temporal difference learning

- The TD update only adjusts the utility of s with that of a single successor s':

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

- Yet it manages to reach the same equilibrium.
- How?

Temporal difference learning

- TD update:

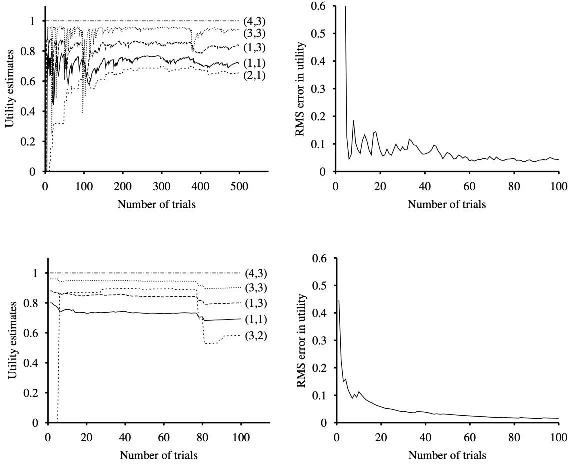
$$U^{\pi}(s) \leftarrow U^{\pi}(s) + \alpha(R(s) + \gamma U^{\pi}(s') - U^{\pi}(s))$$

- In the long run, the transition from s to s' will happen *exactly* in proportion to:

$$P(s'|s, \pi(s))$$

- So $U^{\pi}(s')$ will be averaged into $U^{\pi}(s)$ exactly the right amount, but we need to adjust α over time.

Temporal difference learning



- Error is for $U(1,1)$.
- Temporal difference learning (top) is a bit slower and noisier than ADP (bottom).

Temporal difference learning

- Note that TD learning is model free.
- There is no transition model.
- That makes it easier to apply (no need to count transition probabilities).
- Learning reduces to applying the TD rule on transition from one state to another.

Active reinforcement learning

- The passive reinforcement learning agent is told what to do. (Fixed policy)
- Active reinforcement learning agents must decide what to do. (While learning)
- We will think about how to do this by adapting the passive ADP learner.

Active reinforcement learning

- We can use exactly the same approach to estimating the transition function.
- Sample average of the transitions we observe.
- But computing utilities is more complex.
- When we had a policy, we could use the simple version of the Bellman equation:

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

- When we have to choose actions, we need the utility values to base our choice of action on.

Active reinforcement learning

- We know what to do to get utility values — we use value iteration.
- At any stage, we can run:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a) U_i(s')$$

to stability to compute a new set of utilities.

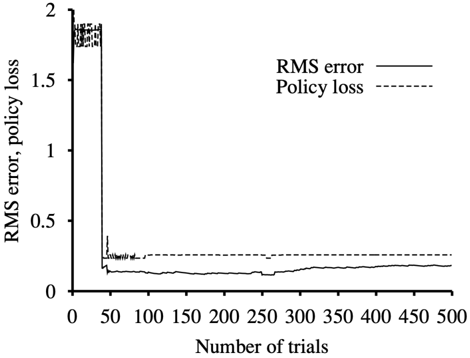
- So establishing utilities is not so hard after all.

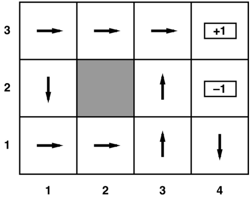
Active reinforcement learning

- Deciding what to do, what action to take, is the next issue.
- Normally after running value iteration we would choose the action with the highest expected utility.
- Greedy agent.
- Could do that while we are learning.
- This turns out not to be so great an idea.
- Typically a greedy agent will not learn the optimal policy.

Active reinforcement learning

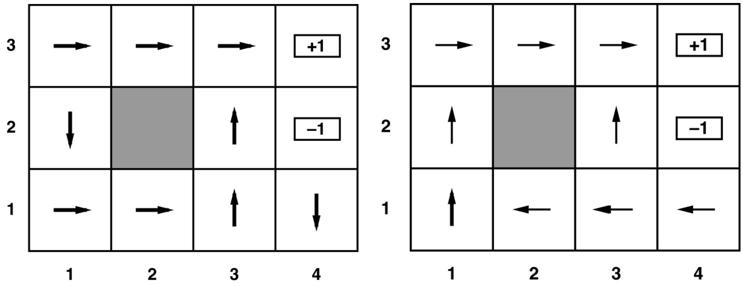
- On the usual example:





- Graph is error compared with optimal utility values.

Greedy vs optimal



- Remember: we start in $(1, 1)$.
- Greedy (left) and optimal (right).
- Greedy prefers the lower route, despite the danger of -1.

Exploration

- The issue is that once the agent finds a run that leads to a good reward, it tends to stick to it, i.e. it stops exploring.
- To do better, we can go back to the idea of bandit learning.
- Could just do ϵ-greedy exploration/exploitation.
- Advantage: Simple, and we know it works.
- Disadvantage: As we saw earlier it can be slow.

Exploration

- A better approach is to change the estimated utility assigned to states in value iteration.
- Manipulate the values to force the learner to explore.
- Then, once exploration is sufficient, we just let it choose the best possible action.
- To do this, we can use:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} f\left(\sum_{s'} P(s'|s,a) U_i(s'), N(s,a) \right)$$

where:

- $N(s,a)$ counts how many times we have done a in s,
- $f(u,n)$ is the exploration function.

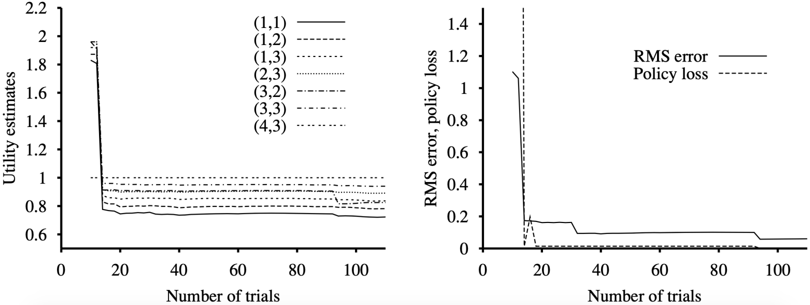
Exploration

- For example:

$$f(u,n) = \begin{cases} R^+ & \text{if } n < N_e \\ u & \text{otherwise} \end{cases}$$

  R^+ is an optimistic reward, and N_e is the number of times we want the agent to be forced to pick an action in every state.
- We force the learner to pick each state/action pair N_e times.
- N_e becomes another parameter that has to be adjusted until we find good solutions.

Exploration



- $R^+ = 2$ and $N_e = 5$.
- Slow to converge on U, but quickly finds a policy that is close to optimal.

Active reinforcement learning: solution

- A list of states $s_1, \ldots s_n$.
- Each state has a utility estimate associated with it $U(s)$.
- Each state has a set of actions associated with it, $a_1, \ldots a_m$.
- Each state/action pair has a probability distribution:

$$P(S'|s, a_i)$$

  over the states S' that it gets to from s by doing a_i.

Model-free active learning

- The form of active reinforcement learning we have just looked at learns a transition model.
- What about a model free version?
- Can quite easily define an active version of temporal difference learning.

Q-learning

- Q-learning is a model-free approach to active reinforcement learning. It does not need to learn $P(s'|s,a)$.
- Revolves around the notion of a Q-value, just like bandit learning.
- The difference is that for bandits we learnt the Q-value of an action.
- Here we learn the Q-value of a state/action pair, $Q(s,a)$.
- $Q(s,a)$ denotes the value of doing a in s, so that:

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

- Easier to learn than $U(s)$.

Q-learning

- We can write:

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

- Note that the sum is over s'.
- Can compute estimates of $Q(s,a)$ by running value-iteration style updates on this.
- But it would not be model-free.

Q-learning

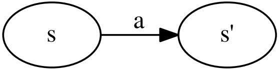
- However, we can write the update rule as:

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

  and recalculate every time that a is executed in s and takes the agent to s'.
- Again, α is the learning rate.
- Note the similarity between this update, and the one for TD-learning (slide 21).

Running updates

- Run an update having moved from s to s':



- Update $Q(s, a)$:

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

where the a' are all the actions we know about in s'.

Action selection

- Since Q-learning is an active approach to reinforcement learning, we have to choose which a' to select in s'.
- Again greedy selection is usually a poor choice.
- Could use ϵ-greedy.
- Or could force exploration as we did before.

Q-learning

function Q-LEARNING-AGENT(*percept*) **returns** an action

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

persistent: Q, a table of action values indexed by state and action, initially zero

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

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

if TERMINAL?(s) **then** $Q[s, None] \leftarrow r'$

if s is not null **then**

increment $N_{sa}[s, a]$

$Q[s, a] \leftarrow Q[s, a] + \alpha(N_{sa}[s, a])(r + \gamma \max_{a'} Q[s', a'] - Q[s, a])$

$s, a, r \leftarrow s', \text{argmax}_{a'} f(Q[s', a'], N_{sa}[s', a']), r'$

return a

- Note that α is a function of the number of visits to s, a.
- Ensures convergence.

Q-learning: solution

- A list of state action pairs $\langle s_i, a_j \rangle$.
- Each state/action pair has $Q(s_i, a_j)$.
- For a given s_i, just pick the a_j to maximise $Q(s_i, a_j)$.

SARSA

- State-Action-Reward-State-Action (SARSA) is a variant of Q-learning.
- Update rule is:

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

  where a' is the actual action taken in s'.
- Rule is applied when the action a' *has been taken*.
- That is after every s, a, r, s', a' cycle. (Hence the name)

SARSA vs Q-learning

- Q-learning computes $Q(s, a)$ using the best $Q(s', a')$ that is accessible from s.
- SARSA uses the actual $Q(s', a')$ once a' has been taken.
- When all the Q values are learnt, there is no difference.
- Greedy agent using the result of Q-learning will do the same as a greedy agent using the results of SARSA.
- Different when exploring.
- Q-learning off-policy. It does not care about the policy being followed. Will update with $Q(s', a')$ even if a' is not the action taken.
- SARSA is on-policy. It will only update with $Q(s', a')$ if a' is taken.

SARSA and Q-learning

- Both slower to converge than the previous approach, active learning with ADP.
- No model means no ability to enforce the Bellman constraint.
- So, is it better to learn the model?
- Depends on what you are doing.

Reinforcement learning update rules

- General form of update rule:

$$\text{New\_estimate} \leftarrow \text{Old\_estimate} + \text{step\_size}\left[\text{Target} - \text{Old\_estimate}\right]$$

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

$$\text{Q-learning:} Q(s,a) \leftarrow Q(s,a) + \alpha\left(R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)$$

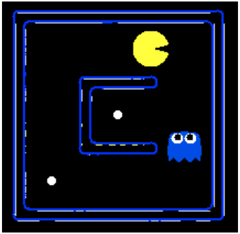
$$\text{SARSA:} Q(s,a) \leftarrow Q(s,a) + \alpha\left(R(s) + \gamma Q(s',a') - Q(s,a)\right)$$

- $\left[\text{Target} - \text{Old\_Estimate}\right]$ is an error in the estimate which is reduced by taking a step towards Target.

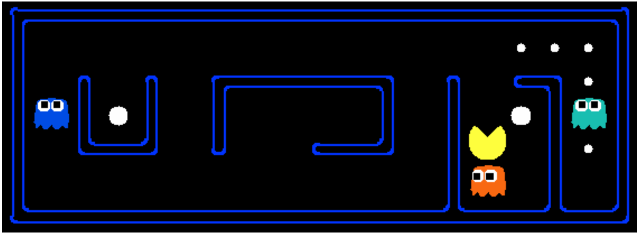
Generalisation

- Up to now, we have explicitly recorded utility functions and Q-values.
- Lookup table.
- This is fine for small numbers of states.
- Time to convergence slows rapidly as the number of states grows.
- For ADP time per iteration also increases.
- Hard limit on the kinds of problem that can be solved.

Generalisation



(UC Berkeley)



Function approximation

- One solution is function approximation.
- That means we use any sort of representation other than a lookup table.
- We have some function on state that provides a value.
- This is an *approximation* because we do not know what the real utility is.

Function approximation

- Given a state s, one reasonable function is a weighted linear function of a set of features:

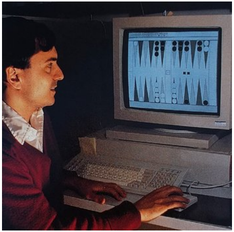
$$\hat{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \ldots + \theta_n f_n(s)$$

- Features are also called basis functions.
- We then try to learn the θ_i.
- Say 20 parameters rather than 10^{40} states.

Function approximation

- We can represent utilities efficiently.
- Can also generalise.
- That is, we can get utilities for states we have not visited.

- TD Gammon, 1992.
- Plays backgammon as well as any human.
- Explores about one state in 10^{12}.



(IBM Research)

- Of course, we might also learn a function that fails completely.
- Trade-off between a function which is likely to span the true utility function and how long it takes to learn.

Function approximation

- Back to:

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

- How could we approximate utility here?

Function approximation

- Assume utility is related to x and y:

$$\hat{U}_\theta(x, y) = \theta_0 + \theta_1 x + \theta_2 y$$

- If, for example:

$$(\theta_0, \theta_1, \theta_2) = (0.5, 0.2, 0.1)$$

then:

$$\hat{U}_\theta(1, 1) = 0.8$$

- To learn this we might run some passive learning trials and collect direct utility estimates of some states.
- Then learning $(\theta_0, \theta_1, \theta_2)$ is a supervised learning problem.
- It is just linear regression (so we know how to solve it).

Function approximation

- What we just described was an *offline* method: do some learning and then do some acting.
- Can do it *online* as well: learn while acting.
- As we learn new estimates of a state, we adjust the weights to reduce the error for that state.

Function approximation

- The real power here is the generalisation.
- Say we get a new utility for $(1, 1)$.
- When we adjust to reduce the error in $(1, 1)$, we change the utility estimate for every other state as well.
- If we have a good function, that will reduce the error for all states with each update.

Function approximation

- Of course we are not limited to linear approximations.
- We can use any of the learning methods we have studied that can output a value.
- Including non-linear approximators like neural networks.
- Deep reinforcement learning.

Summary

- Passive reinforcement learning (again).
  - Adaptive dynamic programming
  - Temporal difference learning
- Active reinforcement learning.
  - Active learning
  - Q-learning
  - SARSA
- Function approximation.