Tufts

Class #25: Abstractions in Reinforcement Learning

Machine Learning (COMP 135): M. Allen, 04 Dec. 19

Extending the Value Update Procedure

function Q-LEARNING(mdp) **returns** a policy
  inputs: mdp, an MDP

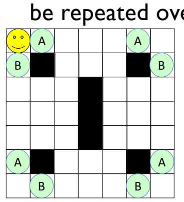
  $\forall s \in S, \forall a \in A, Q(s,a) = 0$
  repeat for each episode E:
    set start-state $s \leftarrow s_0$
    repeat for each time-step t of episode E, until s is terminal:
      set action a, chosen ϵ-greedily based on $Q(s,a)$
      take action a
      observe next state s', one-step reward r
      $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
      $s \leftarrow s'$

  return policy π, set greedily for every state $s \in S$, based upon $Q(s,a)$

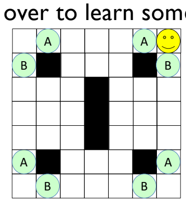
- Basic reinforcement learning algorithms update the value of a ***single state*** (or state-action pair) at a time

Lack of Generalization

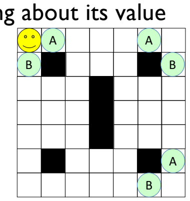
- When doing learning, each state is treated as unique, and must be repeated over and over to learn something about its value



Even if we learn that going right here is best, because type-A objects are better than type-B ones…



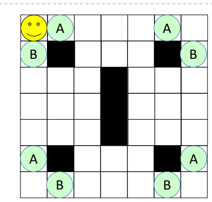
But this really tells us nothing about what to do in a ***similar state*** like this one…



Might even say nothing about what to do in a ***different environment*** with some states ***exactly the same!***

Feature Vectors

- Rather than use every single detail of a state space, we can try to generalize over multiple states at once, by selecting some ***finite number of features*** and learning based upon only those
- States (or state-action pairs) that share the same features are thus treated the same, even if they differ in other ways that we don't pay attention to
- Using the right features can speed learning significantly



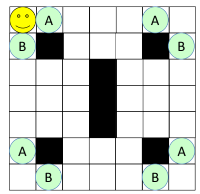
- Here, we might try representing state-action pairs (s, a) in terms of just four features:

$$f_X(s,a) = \frac{x}{\max_x} \text{ for } x\text{-coordinate after } a \text{ in } s$$
$$f_Y(s,a) = \frac{y}{\max_y} \text{ for } y\text{-coordinate after } a \text{ in } s$$
$$f_A(s,a) = \frac{1}{d_A + 1}, \text{ where } d_A \text{ is the distance to nearest } A \text{ after } a \text{ in } s$$
$$f_B(s,a) = \frac{1}{d_B + 1}, \text{ where } d_B \text{ is the distance to nearest } B \text{ after } a \text{ in } s$$

Choosing Feature Vectors

- We want to choose values that seem to be important to problem success
- When we represent them, it has been shown that we get better results if we ***normalize*** features, ensuring that each is in same, unit range:

$$0 \leq f_i(s,a) \leq 1$$



$$f_X(s,a) = \frac{x}{\max_x} \text{ for } x\text{-coordinate after } a \text{ in } s$$
$$f_Y(s,a) = \frac{y}{\max_y} \text{ for } y\text{-coordinate after } a \text{ in } s$$
$$f_A(s,a) = \frac{1}{d_A + 1}, \text{ where } d_A \text{ is the distance to nearest } A \text{ after } a \text{ in } s$$
$$f_B(s,a) = \frac{1}{d_B + 1}, \text{ where } d_B \text{ is the distance to nearest } B \text{ after } a \text{ in } s$$

Value Functions over Features

- One issue is that when we use simpler features, we don't always know which ones to use
  - States may share features and still have very different values
  - Some features may turn out to be more or less important
  - We want to ***learn*** a proper function that tells us how much we should pay attention to each feature
- We may assume this function is ***linear***
  - This means that the value of a state is a simple combination of weights applied to each feature
  - While this assumption may not be the right one in some domains, it can often be the basis of a good approximation

Linear Functions over Features

- What we want is to learn the set of ***weights*** needed to properly calculate our U- or Q-values

$$U(s) = w_1\, f_1(s) + w_2\, f_2(s) + \cdots + w_n\, f_n(s)$$
$$Q(s,a) = w_1\, f_1(s,a) + w_2\, f_2(s,a) + \cdots + w_n\, f_n(s,a)$$

- For instance, for our grid problem:

$$Q(s,a) = w_X\, f_X(s,a) + w_Y\, f_Y(s,a) + w_A\, f_A(s,a) + w_B\, f_B(s,a)$$

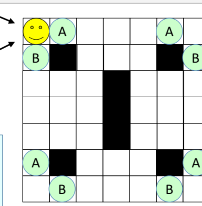
Setting the Weights

- Initially, we may not know which features really matter
  - In some cases, we may have knowledge that tells us some are more important than others, and we will weight them more
  - In other cases, we may treat them all the same
- For instance, in our grid problem, we might start with all weights the same (1.0)

$$\begin{aligned} Q(s,\textsc{Right}) &= w_X\, f_X(s,a) + w_Y\, f_Y(s,a) + w_A\, f_A(s,a) + w_B\, f_B(s,a) \\ &= \boxed{(1.0 \times 2/7) + (1.0 \times 1/7) + (1.0 \times 1) + (1.0 \times 1/3)} \\ &= 1.76 \end{aligned}$$

After Right, at $(x,y) = (2,1)$, distance 0 to A, distance 2 to B

$$\begin{aligned} Q(s,\textsc{Down}) &= w_X\, f_X(s,a) + w_Y\, f_Y(s,a) + w_A\, f_A(s,a) + w_B\, f_B(s,a) \\ &= (1.0 \times 1/7) + (1.0 \times 2/7) + (1.0 \times 1/3) + (1.0 \times 1) \\ &= 1.76 \end{aligned}$$



Initially, many states may end up with identical value estimates. If it turned out that position didn't matter, and both A- and B-type objects were equally valuable, this would be fine. Typically, however, this is not the case, and we will need to adjust our weights dynamically as we go.

Q-Learning with Function Approximation

- In normal Q-learning, we evaluate a state-action pair (s,a) based on the results we get (r and s') and update the ***single pair value***:

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

$$Q(s,a) \leftarrow Q(s,a) + \alpha\,\delta$$

- Now, we will instead update ***each of the weights*** on our features
  - If outcomes are particularly good or bad, we change weights accordingly
  - This affects ***all*** (state, action) pairs that share features with current one

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

$$\forall i,\ w_i \leftarrow w_i + \alpha\,\delta\,f_i(s,a)$$

Adjusting the Weights

- Now, when we take an action, we adjust weight-values and then compute Q-values
- For example: we take the RIGHT action and get a large positive reward, which means we could increase weights on contributing features

$$\delta = r + \gamma \max_{a'} Q(s',a') - Q(s,\mathrm{Right}) = 10$$

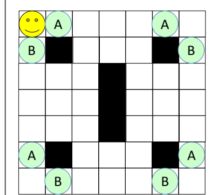
$$w_X \leftarrow w_X + \alpha\,\delta\,f_X(s,a)$$
$$\leftarrow 1.0 + 0.9 \times 10 \times 2/7 = 3.57$$

$$w_Y \leftarrow w_Y + \alpha\,\delta\,f_Y(s,a)$$
$$\leftarrow 1.0 + 0.9 \times 10 \times 1/7 = 2.29$$

$$w_A \leftarrow w_A + \alpha\,\delta\,f_A(s,a)$$
$$\leftarrow 1.0 + 0.9 \times 10 \times 1 = 10.0$$

$$w_B \leftarrow w_B + \alpha\,\delta\,f_B(s,a)$$
$$\leftarrow 1.0 + 0.9 \times 10 \times 1/3 = 4.0$$

$$Q(s,\textsc{Right}) = (3.57 \times 2/7) + (2.29 \times 1/7) + (10.0 \times 1) + (4.0 \times 1/3)$$
$$= 12.69$$



Adjusting the Weights

- Later, if we take the DOWN action from the same state, and get a large negative cost-value, we might down-grade weights on the contributing features

$$\delta = r + \gamma \max_{a'} Q(s',a') - Q(s,\textsc{Down}) = -20$$

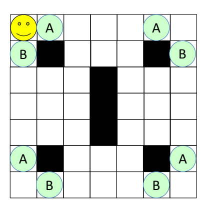
$$w_X \leftarrow w_X + \alpha\,\delta\,f_X(s,a)$$
$$\leftarrow 3.57 + 0.9 \times -20 \times 1/7 = 1.0$$

$$w_Y \leftarrow w_Y + \alpha\,\delta\,f_Y(s,a)$$
$$\leftarrow 2.29 + 0.9 \times -20 \times 2/7 = -2.85$$

$$w_A \leftarrow w_A + \alpha\,\delta\,f_A(s,a)$$
$$\leftarrow 10.0 + 0.9 \times -20 \times 1/3 = 4.0$$

$$w_B \leftarrow w_B + \alpha\,\delta\,f_B(s,a)$$
$$\leftarrow 4.0 + 0.9 \times -20 \times 1 = -14.0$$

$$Q(s,\textsc{Down}) = (1.0 \times 1/7) + (-2.85 \times 2/7) + (4.0 \times 1/3) + (-14.0 \times 1)$$
$$= -13.34$$



Adjusting the Weights

- Note: since we adjust the weights that are used to calculate the Q-values of any (state, action) pairs, what happens when we encounter one new outcome actually affects the Q-value of ***all*** the pairs at once
- We are thus potentially learning a value function over our ***entire space***, even though it is based only on a single outcome at a time, which can speed up learning

$$\delta = r + \gamma \max_{a'} Q(s',a') - Q(s,\textsc{Down}) = -20$$

$$Q(s,\textsc{Down}) = (1.0 \times 1/7) + (-2.85 \times 2/7) + (4.0 \times 1/3) + (-14.0 \times 1)$$
$$= -13.34$$

$$Q(s,\textsc{Right}) = (1.0 \times 2/7) + (-2.85 \times 1/7) + (4.0 \times 1) + (-14.0 \times 1/3)$$
$$= -0.79$$

After Down, we have changed weights, which changes the Q-value of not only one state-action pair, but ***all of them***.

Here, we see the updated value for going Right (this was 12.69 before).

Where Do Features Come From?

- A variety of approaches can be used to generate useful features for a given learning problem
  - Sometimes we have good intuitions about what features are useful, and sometimes we don't
- For example, suppose we have a problem in which states are characterized by (x, y) points in the plane:
  1. We might just use the features (x, y) themselves, with 2 matching weights…
  2. Or a simple polynomial combination, like $(1, x, y, xy)$, with 4 weights to go along with those…
  3. Or a more complex polynomial, like:

$$(1, x, y, xy, x^2, y^2, xy^2, x^2y, x^2y^2)$$

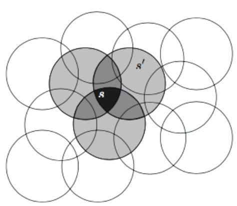
13

Various Feature Functions

- Experimentation has been performed with many different functional forms for features:
  1. Polynomial features: linear weights over polynomial combinations of numeric features
  2. Fourier features: combinations of sine and cosine functions over the underlying numbers
  3. Radial basis functions: real-valued features based on computed distances from chosen points in the state-space
- A common issue is the complexity growth of the features as the dimensionality of the state space increases
  - A variety of techniques exist to use sparser, binary features

14

Coarse Coding



- Suppose we have a state-space consisting of points in the plane
- A binary coding scheme is to use features that each correspond to circles in the state-space
  - A point has a given feature if it lies inside the circle
  - Two points share all the features that overlap them both

Image: Sutton & Barto, 2018

15

Advantages of Binary Features

- Given a coarse coding scheme (set of n circles), we get a feature-vector, and corresponding weight-vector for state s:

$$\mathbf{x}_s = (c_1, c_2, \ldots, c_n)$$
$$\mathbf{w}_s = (w_1, w_2, \ldots, w_n)$$

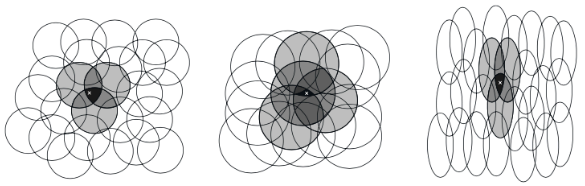
- Each feature is a binary value:

$$c_i = \begin{cases} 1 & \text{if } s \text{ lies inside circle } i \\ 0 & \text{else} \end{cases}$$

- If we use these features to compute a utility-value for s, rather than a series of multiplications and additions, we get the simpler summation: $U(s) = \mathbf{w}_s \cdot \mathbf{x}_s = \sum_i \{w_i \mid c_i = 1\}$

16

Feature Variation

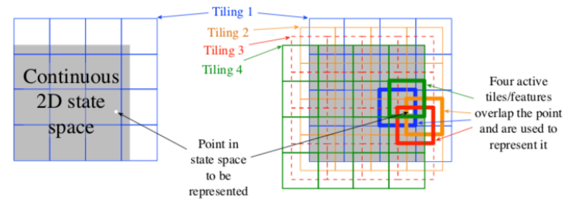


Narrow generalization Broad generalization Asymmetric generalization

- A simple idea, circular coarse coding can still generalize over domains in a variety of ways
  - We get generalization, since the ***same*** weight vector is used for every state (only the ***particular features*** change)

Image: Sutton & Barto, 2018

Tile Coding



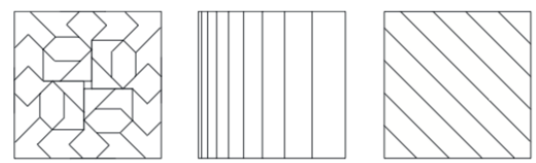
- Another form of coarse coding is to tile the state space
  - A single, simple tiling is just a partition, dividing the state space into uniform regions
  - A more complex tiling uses ***multiple overlapping*** partitions
  - Again, each individual tile corresponds to a binary feature

Image: Sutton & Barto, 2018

Placement of Tiles

- Much research has gone into good ways to choose how many tilings to use, and how they should overlap
  - It has been found that choosing ***uniform offsets*** of the tiles (e.g. moving each new tiling over by 1 unit in each direction) can introduce certain numerical biases
- Best practices, as recommended by Miller and Glanz (96):
  1. For a state space of dimensionality k, use $2^n \geq 4k$ tilings
  2. Offset each dimension using numbers $(1, 3, 5, \ldots, 2k-1)$
- For example, in 2 dimensions, use 8 or more tilings, offsetting each by 1 unit in the x dimension, and 3 in y
- In 3 dimensions, use at least 12 tilings, offsetting by 1 unit in x, 3 in y, 5 in z …

Other Approaches to Tiling



Irregular Log stripes Diagonal stripes

- Any number of tiling patterns and offsets can be used
  - Tilings need not be regular in shape or size
  - "Striped" tilings generalize along only certain dimensions
  - Different sizes of tiles allow finer/coarser discrimination in certain parts of the state space

Image: Sutton & Barto, 2018

Sparse Distributed Memory

- Developed by NASA Ames researcher Pentti Kanerva while working on a model of human long-term memory
- In RL, often now called Kanerva coding
- A model that generalizes over states based on a similarity measure
- State-features are represented again as binary vectors, which can be regarded as lists of ***other*** states to which they are or are not similar

21

Basic Kanerva Coding for Q-Learning

- Rather than save Q-values for all state-action pairs, a Kanerva coding selects a subset of prototypes:

$$P = \{p_1, p_2, ..., p_n\} \subsetneq S$$

- For any state s and prototype p_i, we say the two are adjacent if s and p_i differ by at most 1 feature (other such similarity measures are possible)
- For example, if we have two state feature-variables:

$$f_1 \in \{1, 2\} \qquad f_2 \in \{a, b, c\}$$

- Then the state $s = (1, a)$ would be:
  1. Adjacent to prototypes $p_i = (2, a), (1, b)$, or $(1, c)$
  2. ***Not*** adjacent to $p_j = (2, b)$ or $(2, c)$

22

From Prototypes to Binary Features

- For our vector of prototypes we then get a vector of binary features for every state:

$$P = \{p_1, p_2, ..., p_n\}$$

$$\mathbf{x}_s = (f(s)_1, f(s)_2, ..., f(s)_n)$$

$$f(s)_i = \begin{cases} 1 & \text{if } s \text{ is adjacent to } p_i \\ 0 & \text{else} \end{cases}$$

- Our Q-learning algorithm then learns weight values over prototypes only:

$$\theta(p_i, a), \; \forall p_i \in P, \; \forall a \in A$$

23

Adjusting Prototype Weights

- For any state-action pair, we can now compute an approximate Q-value, based only on adjacent prototypes:

$$\hat{Q}(s, a) = \sum_i \theta(p_i, a) \, f(s)_i$$

- Furthermore, when our learning algorithm takes action a in state s_1, receives reward r, and ends up in state s_2 we update:

$$\theta(p_i, a) \leftarrow \theta(p_i, a) + f(s)_i \, \alpha(r + \gamma \max_{a_2} \hat{Q}(s_2, a_2))$$

- Doing it this way also means that we only update those weights on pairs featuring prototypes that are adjacent to s, since otherwise $f(s)_i = 0$

24

Choosing Prototypes

- In the limit, we could use ***all*** states as prototypes ($P = S$), and impose strict identity on the measure of adjacency:

$$f(s)_i = \begin{cases} 1 & \text{if } s = p_i \\ 0 & \text{else} \end{cases}$$

  - In this case, the algorithm is just normal Q-learning, without any generalization at all

- If we make the set of prototypes very small, on the other hand, then most states will not be adjacent to any prototype, and we won't learn ***anything*** about those states at all

25

Choosing Prototypes

- In between the extremes, the usefulness of Kanerva encoding is maximized when:
  1. No prototype is visited too much (such a state is effectively ***too abstract***)
  2. No prototype is visited too little (such a state is effectively ***too specific***)
- Achieving this balance with an initial set or prototypes, which is often chosen randomly, or according to some heuristic, is challenging, leading to interest in adaptive Kanerva Coding, where we ***change*** the prototype set over time as we learn

26

Adaptive Kanerva Coding

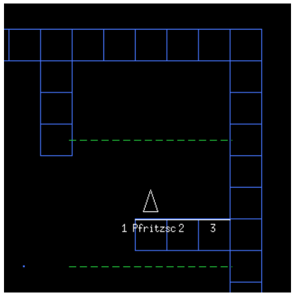
- We can modify the basic approach as follows:

1. We start with a set of randomly chosen prototypes
2. For each prototype, we keep track of how many times we visit a state that is adjacent to it
3. Periodically, we modify the prototype set in two ways:
   a. Deletion: if a prototype has been visited t times, we decide whether or not to delete it randomly, with probability:
   $$P_{del} = e^{-t}$$
   b. Splitting: whenever some prototypes are deleted, they are replaced by taking the **most-visited** prototypes and generating new, adjacent ones by changing one of their feature-values

27

An Application: Xpilot Navigation

- Xpilot is a space-shooter game with the ability to add complex physics and environments
- Basic Q-learning is thwarted by the enormous state-space of the full game (even when only trying to learn to navigate successfully without crashing)



- Even on a small map of size (500×500), a ship with 10 possible speeds and full rotation will correspond to approximately 3.24×10^{10} states!

28

An Application: Xpilot Navigation

TABLE I
STATE VARIABLES FOR XPILOT, WITH RANGES. THE STATE SPACE IS MADE UP OF 5 VARIABLES AND 31104 STATES. THESE COMBINE WITH 4 ACTIONS FOR 124416 STATE-ACTION PAIRS (s, a). 1024 STATES ARE CHOSEN AS PROTOTYPES, FOR 4096 PROTOTYPE-ACTION PAIRS (p, a).

State Variable	Range
Heading	1–36
Tracking	1–36
Speed	1–6
Near Wall	$\{1, 0\}$
Near Corner	$\{1, 0\}$

TABLE III
THE REWARD-STRUCTURE.

Reward Value	Condition
-10	Agent crashes
$+1$	Agent is alive for one frame

TABLE II
POSSIBLE AGENT ACTIONS

Action	Effect
Avoid Wall	Turn 10°: away from nearest wall; thrust once
Avoid Corner	Turn 10°: direction 180° opposite **Tracking**; thrust once
Thrust	If speed $s < 6$, thrust once
Do Nothing	`null`

Even with a simplified and discretized state-action space, things are still far too complex for effective use of basic RL algorithms

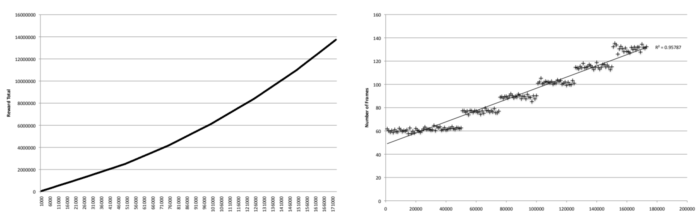
29

Other Parameters

1. # of prototypes: $1{,}024$ initial prototypes (at random)
   - A total of $4{,}096$ (prototype, action) pairs
   - ~3.3% of overall possible (state, action) pairs
2. Learning updates: $\gamma = 0.9$, $\alpha = 0.1$
3. Policy randomness: initially $\varepsilon = 0.9$
   - Every $25{,}000$ games we update: $\varepsilon = \dfrac{0.9}{\lfloor \texttt{totalGames}/25{,}000 \rfloor + 1}$
4. Updating prototypes: every time any prototype is visited (by encountering an adjacent state) 50 times, we:
   a. Delete m of them using randomness based upon number of visits
   b. Split each of the most-visited m prototypes, returning to $1{,}024$
   c. Re-set all counts of how many times each is visited

30

Learning Performance

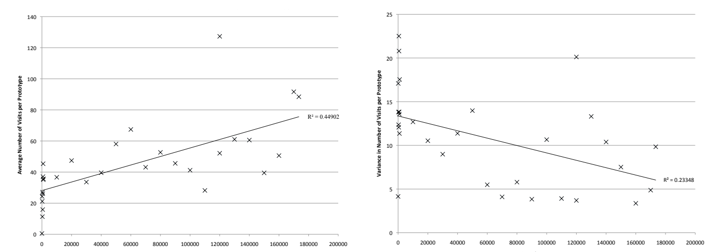


(a) Total Reward Over Time (b) Average Frames Alive Over Time

- The agent continues to improve performance over many hours of learning, comprising over $170{,}000$ episodes
  - Note: as they get better, each episode of learning gets longer as they survive more frames

31

Prototype Visits



(a) Average Visits per Prototype Over Time (b) *Variance*: Average Visits per Prototype Over Time

- As rarely visited prototypes are replaced by more useful ones, overall rate of visits increases
  - At same time, variance of visits to any prototype decreases

32

Adding More Prototypes

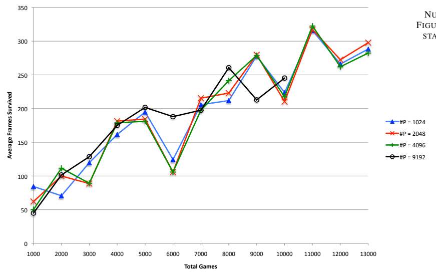


TABLE IV
NUMBERS OF PROTOTYPES USED IN THE EXPERIMENTS DESCRIBED IN FIGURE 4. EACH COMBINES WITH 4 ACTIONS FOR THE GIVEN NUMBER OF STATE-ACTION PAIRS (s, a). ALSO SHOWN IS THE PERCENTAGE OF THE ENTIRE STATE-ACTION SPACE (124416 PAIRS) REPRESENTED.

Prototypes	$(\mathbf{s}, \mathbf{a})$	% Total
1024	4096	3.3%
2048	8192	6.6%
4096	16384	13.2%
9192	36768	29.6%

- Using 2/4/8 times as many prototypes has little to no effect, meaning that the smaller number is as good as any other
- For the largest size of prototype set, learning was much slower, and experiments had to be curtailed somewhat

This Week & Next Week

- **Topics**: Approximation in RL & Review
- **Homework 06**: Last assignment, due Monday, 06 Dec.
- **Final Project**: Essay assignment, due Thursday, 19 Dec.
- **Final Exam**: in regular classroom
  - 7:00–9:00 PM, Thursday, 12 December
  - Practice exam out by end of this week
  - Exam review: last class (Monday, 09 December)
- **Office Hours:** as usual next week
  - By appointment with instructor after that