COMP90054 — AI Planning for Autonomy

10. More Efficient Reinforcement Learning

Reward shaping and Q-function approximation

Adrian Pearce

THE UNIVERSITY OF MELBOURNE

Semester 2, 2021

Agenda

1 Motivation

2 Approximating Q-functions

3 Shaping Rewards and Initial Q-Functions

4 Conclusion

Learning Outcomes

1. Manually apply linear Q-function approximation to solve small-scale MDP problems given some known features
2. Select suitable features and design & implement Q-function approximation for model-free reinforcement learning techniques to solve medium-scale MDP problems automatically
3. Argue the strengths and weaknesses of function approximation approaches
4. Compare and contrast linear function approximation with function approximation using deep neural networks
5. Explain how reward shaping can be used to help model-free reinforcement learning methods to converge
6. Manually apply reward shaping for a given potential function to solve small-scale MDP problems
7. Design and implement potential functions to solve medium-scale MDP problems automatically
8. Compare and contrast reward shaping with Q-function initialisation

Reinforcement Learning – Some Weaknesses

In the previous lectures, we looked at fundamental temporal difference (TD) methods for reinforcement learning: Q-learning and SARSA.
These two methods have some weaknesses in this basic format:

1. Unlike Monte-Carlo methods, which reach a reward and then backpropagate this reward, TD methods use bootstrapping (they estimate the future discounted reward using $Q(s, a)$), which means that for problems with sparse rewards, it can take a long time to for rewards to propagate throughout a Q-function.
2. Both methods estimate a Q-function $Q(s, a)$, and the simplest way to model this is via a Q-table. However, this requires us to maintain a table of size $|A| \times |S|$, which is prohibitively large for any non-trivial problem.
3. Using a Q-table requires that we visit every reachable state many times and apply every action many times to get a good estimate of $Q(s, a)$. Thus, if we never visit a state s, we have no estimate of $Q(s, a)$, even if we have visited states that are very similar to s.
4. Rewards can be sparse, meaning that there are few state/actions that lead to non-zero rewards. This is problematic because initially, reinforcement learning algorithms behave entirely randomly and will struggle to find good rewards. Remember the Freeway demo from the previous lecture?

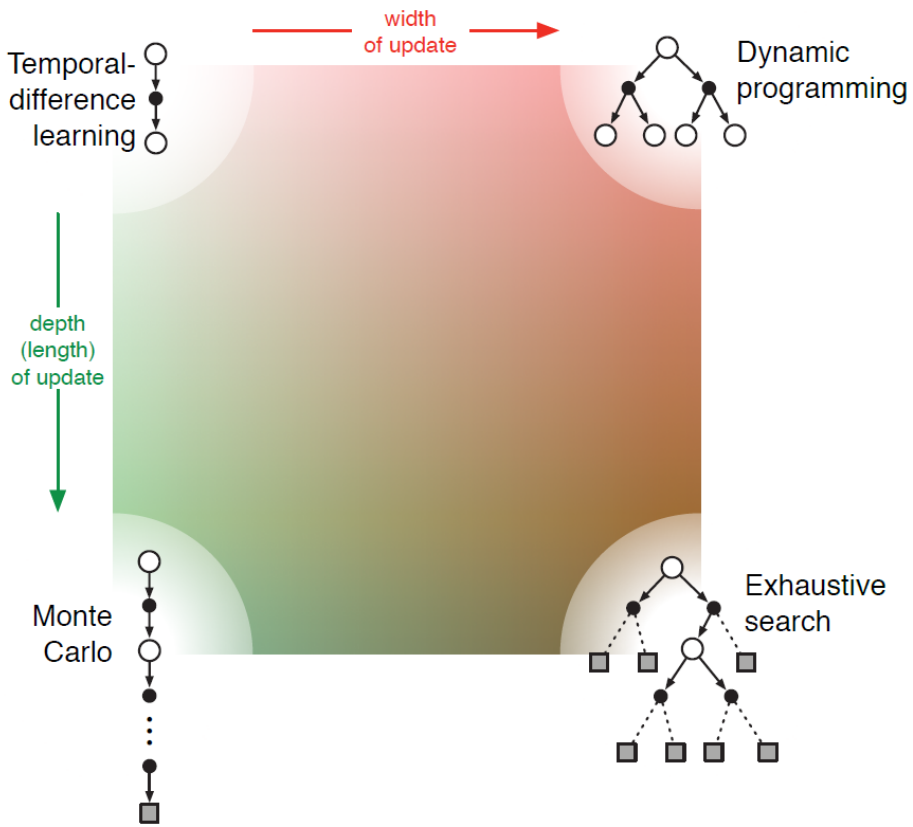
Reinforcement Learning – Some Improvements

To get around these limitations, we are going to look at three simple approaches that can improve temporal difference methods:

1. *n-step temporal difference learning*: Monte Carlo techniques execute entire traces and then backpropagate the reward, while basic TD methods only look at the reward in the next step, estimating the future wards. n-step methods instead look n steps ahead for the reward before updating the reward, and then estimate the remainder. *Last lecture!*
2. *Approximate methods*: Instead of calculating an exact Q-function, we approximate it using simple methods that both eliminate the need for a large Q-table (therefore the methods scale better), and also allowing use to provide reasonable estimates of $Q(s, a)$ *even if we have not applied action a in state s previously*. *This lecture!*
3. *Reward shaping and Q-Value Initialisation*: If rewards are sparse, we can modify/augment our reward function to reward behaviour that we think moves us closer to the solution, or we can guess the optimal Q-function and initial $Q(s, a)$ to be this. *This lecture!*

Approaches to AI Planning and Reinforcement Learning

n-step TD learning comes from the idea used in the image below. Monte Carlo methods uses 'deep' updates and backups, where entire traces are executed and the reward backpropagated. Whereas temporal difference learning methods such as Q-learning and SARSA use 'shallow' updates and backups, only using the reward from the 1-step ahead. n-step learning finds the middle ground: only update the Q-function after having explored ahead n steps.



n-step Q-learning and SARSA

Both Q-learning and SARSA have n-step version.

Problem: Large State Spaces

As discussed, the problem is that the size of the state space increases exponentially with every variable added. This causes two main issues:

1. Storing a value of $Q(s, a)$ for every s and a, such as in Q-tables, is infeasible.
2. Propagation of Q-values takes a long time.

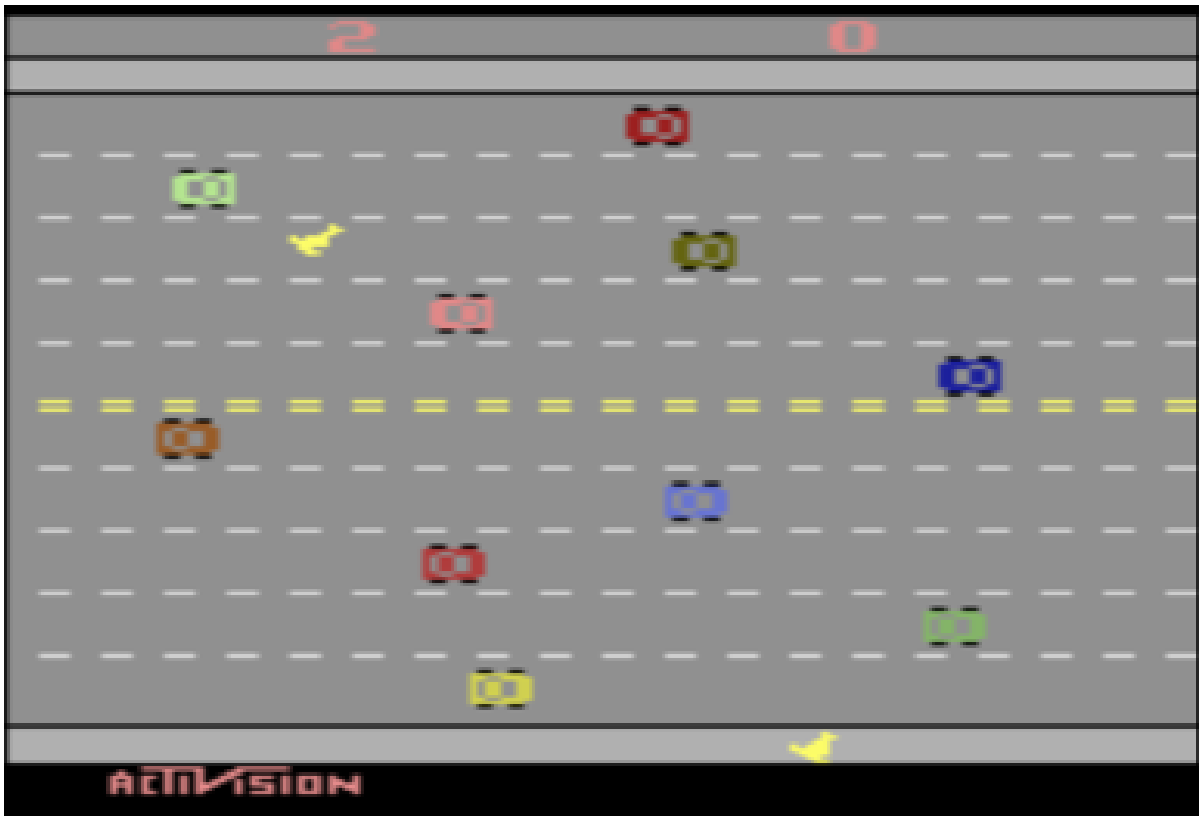
Example: *Freeway*. Let's say there are 12 rows and about 40 columns (this underestimates because the cars move a few pixels at a time, not in columns), so about 480 different positions. There are two chickens, plus we need to store whether there is a car in each location (at the very least). This leads to:

$$480^2 + 2^{480} = 3.12 \times 10^{144} \text{ states}$$

And therefore we would need a Q-table this size times the number of actions (Four actions I think: left, right, up, down?).

Feature Selection Exercise: Freeway

What would be some good features for the Freeway game to learn how to get the chicken across the freeway?



Linear Function Approximation

The key idea is to *approximate* the Q-function using a linear combination of *features* and their weights. Instead of recording everything in detail, we think about what is most important to know, and model that.

The overall process is:

1. For the states, consider what are the features that determine its representation.
2. During learning, perform updates based on the *weights* of *features* instead of states.
3. Estimate $Q(s, a)$ by summing the features and their weights.

Example: *Freeway*. Instead of recording the position of both players and whether there is a car in every position, just record how far away each player is from the other side of the road and how far away the *closest* car is in each row. This is probably enough. This requires just 14 features. In approximate Q-learning, we store features and weights, not states. What we need to learn is how important each feature is (its *weight*) for each action.

Approximate Q-function Representation

As noted, a Q-function is represented using the features and their weights, instead of a Q-table.

To represent this, we have two vectors:

1. A *feature vector*, $f(s,a)$, which is a vector of $n \cdot |A|$ different functions, where n is the number of state features and $|A|$ the number of actions. Each function extracts the value of a feature for state-action pair (s,a). We say $f_i(s,a)$ extracts the ith feature from the state-action pair (s,a):

$$f(s,a) = \begin{pmatrix} f_1(s,a) \\ f_2(s,a) \\ \dots \\ f_n(s,a) \end{pmatrix}$$

   In the Freeway example, we have a vector with 14 state features times four actions. The function $f_i(s, Up)$ returns the closest car for row $i+1$ (for each of the 12 rows) if going up, $f_{13}(s, Up)$ is the distance to the goal for the first chicken, and $f_{14}(s, Up)$ the second chicken; e.g. $f_{13}(s,a) = \frac{row(s,chicken)}{max\_row}$.
2. A *weight* vector w of size $n \times |A|$: one weight for each feature-action pair. w_i^a defines the weight of a feature i for action a.

Defining State-Action Features

Often it is easier to just define features for states, rather than state-action pairs. The features are just a vector of n functions of the form $f_i(s)$.

However, for most applications, the weight of a feature is related to the action. The weight of being one step away from the end in Freeway is different if we go Up than if we go Right.

It is straightforward to construct $n \times |A|$ state-pair features from just n state features:

$$f_{ik}(s,a) = \begin{cases} f_i(s) & \text{if } a = a_k \\ 0 & \text{otherwise} \end{cases} \quad 1 \leq i \leq n, 1 \leq k \leq |A|$$

This effectively results in $|A|$ different weight vectors: (s,a):

$$f(s,a_1) = \begin{pmatrix} f_{1,a_1}(s,a) \\ f_{2,a_1}(s,a) \\ 0 \\ 0 \\ 0 \\ 0 \\ \dots \end{pmatrix} \quad f(s,a_2) = \begin{pmatrix} 0 \\ 0 \\ f_{1,a_2}(s,a) \\ f_{2,a_2}(s,a) \\ 0 \\ 0 \\ \dots \end{pmatrix} \quad f(s,a_3) = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ f_{1,a_3}(s,a) \\ f_{2,a_3}(s,a) \\ \dots \end{pmatrix} \quad \dots$$

Approximate Q-function Computation

Give a feature vector f and a weight vector w, the Q-value of a state is a simple linear combination of features and weights:

$$\begin{aligned} Q(s,a) &= f_1(s,a) \cdot w_1^a + f_2(s,a) \cdot w_2^a + \ldots + f_n(s,a) \cdot w_n^a \\ &= \sum_{i=0}^{n} f_i(s,a) w_i^a \end{aligned}$$

Example: For the Freeway example, we would assume that moving up would give a better score than moving down, all else equal (that is, if the closest car in the next row up is the same distance away than the closest in the next row down). So, for state s where the chicken is in row 1:

$$Q(s, Up) = f_1(s, Up) \cdot 0.31 + \ldots + f_{14}(s, Up) \cdot 0.04$$

Note that to be effective, our feature values should be *normalised* using e.g. min-max normalisation or mean normalisation.

Approximate Q-function Update

To use approximate Q-functions in reinforcement learning, there are two steps we need to change from the standard algorithms: (1) initialisation; and (2) update.

For initialisation, initialise all weights to 0. Alternatively, you can try Q-function initialisation and assign weights that you think will be 'good' weights.

For update, we now need to update the weights instead of the actions. For Q-learning, the update rule is now:

$$w_i^a \leftarrow w_i^a + \alpha[r + \gamma max_a' Q(s', a') - Q(s, a)] f_i(s, a)$$

For SARSA:

$$w_i^a \leftarrow w_i^a + \alpha[r + \gamma Q(s', a') - Q(s, a)] f_i(s, a)$$

Note: we need to update for each feature i for the last executed action a.

As this is linear, it will eventually converge!

Q-value Propagation

Note that this has the effect of updating Q-values to states that have never been visited!

In Freeway, for example, if we receive our first reward by crossing the road (going Up from the final row), this will update the weight all features for Up, and now we have a Q-value for going Up from *any* position on the final row.

Assume that all weights are 0, therefore, $Q(s,a) = 0$ for every state and action. Now, we receive the reward of 10 for getting to the other side of the road. If feature 14 is has the value $\frac{r}{D}$, where r is the current row and D is the distance to the other side, then we have:

$$\begin{array}{lcl} w_i^a & \leftarrow & w_i^a + \alpha[r + \gamma \max_a Q(s',a') - Q(s,a)]f_i(s,a) \\ w_{14}^{Up} & \leftarrow & 0 + 0.5[10 + 0.9 \times 0]\frac{10}{10} \\ & = & 5 \end{array}$$

From this, we now can get an estimate of $Q(s, Up)$ from any state because we have some weights in our linear function. Those that are closer to the other size of the road will get a higher Q-value than those further away (all other things being equal).

Q-function Approximation with Neural Networks: Deep Q-learning

The latest hype in reinforcement learning is all about the use of deep neural networks to approximate value and Q-functions.

In brief: instead of selecting features and training weights, we learn the parameters θ to a neural network. The Q-function is $Q(s, a; \theta)$, so takes the parameters as an argument.

The TD update for Q-learning is just:

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

Advantage (compared to linear Q functions): we do not need to select features – the 'features' will be learnt as part of the hidden layers.

Disadvantages: there are no convergence guarantees; and very data hungry because they need to learn features as well as Q-function.

Despite this, deep Q-learning often works remarkably well in practice, especially for tasks that require vision (see the robotic arm grasping unknown objects in the lecture on Q-learning).

Strengths and Limitations of Q-function Approximation

Approximating Q-functions using machine learning techniques has advantages and disadvantages.

Advantages:

- More efficient representation compared with Q-tables.
- Q-value propagation

Disadvantages:

- The Q-function is now only an approximation of the real Q-function: states that share feature values may have different actual values

Applications of Deep Reinforcement Learning

A great application of using off-policy updates in deep Q-learning for robotic arms to learn how to grasp unknown objects. The only input for the problem is the camera data:

`https://www.youtube.com/watch?v=cXaic_k80uM&feature=youtu.be`

This is using policy iteration (policy gradient descent) rather than standard Q-learning.

What is reward shaping?

In many applications, you will have some idea of what a good solution should look like. For example, in our simple navigation task, it is clear that moving towards the reward of +1 and away from the reward of -1 are likely to be good solutions.

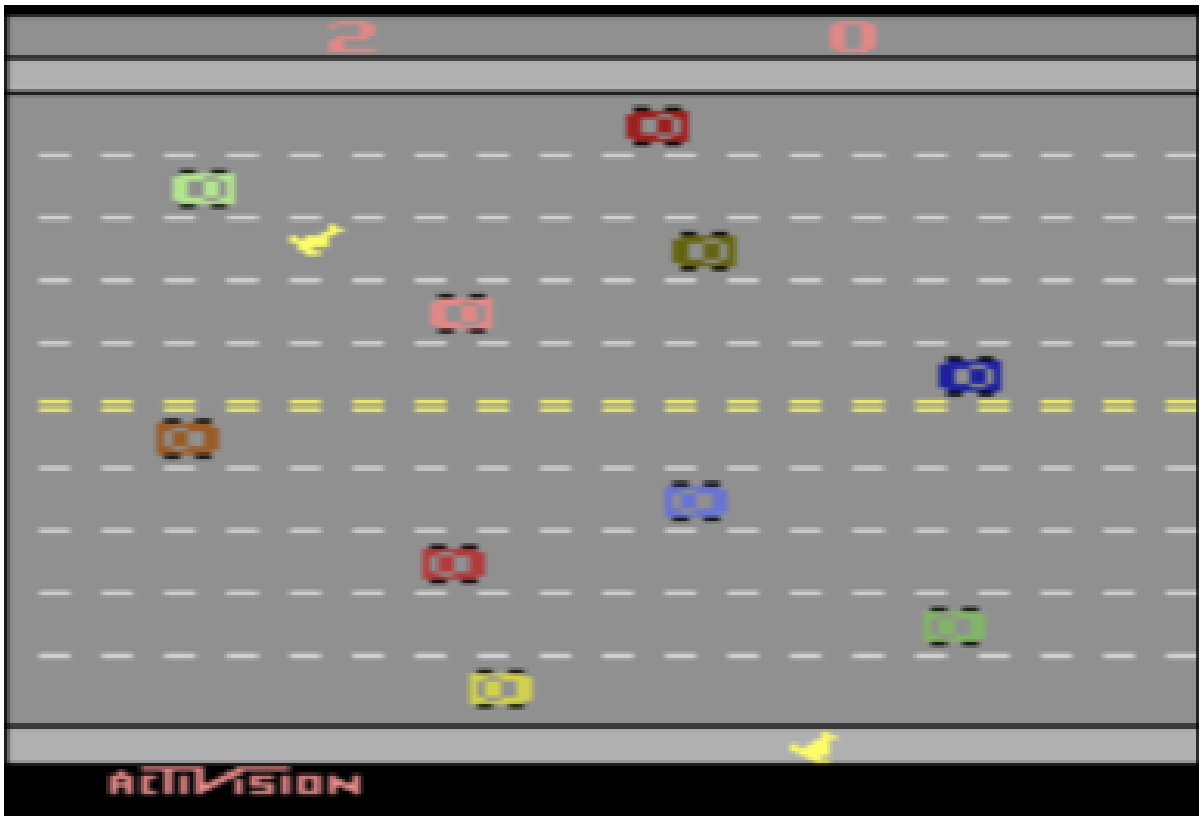
Can we then speed up learning and/or improve our final solution by nudging our reinforcement learner towards this behaviour?

The answer is: Yes! We can modify our reinforcement learning algorithm slightly to add domain knowledge, while also guaranteeing optimality.

This is known as *domain knowledge* — that is, stuff about the domain that the human modeller knows about while constructing the model to be solved.

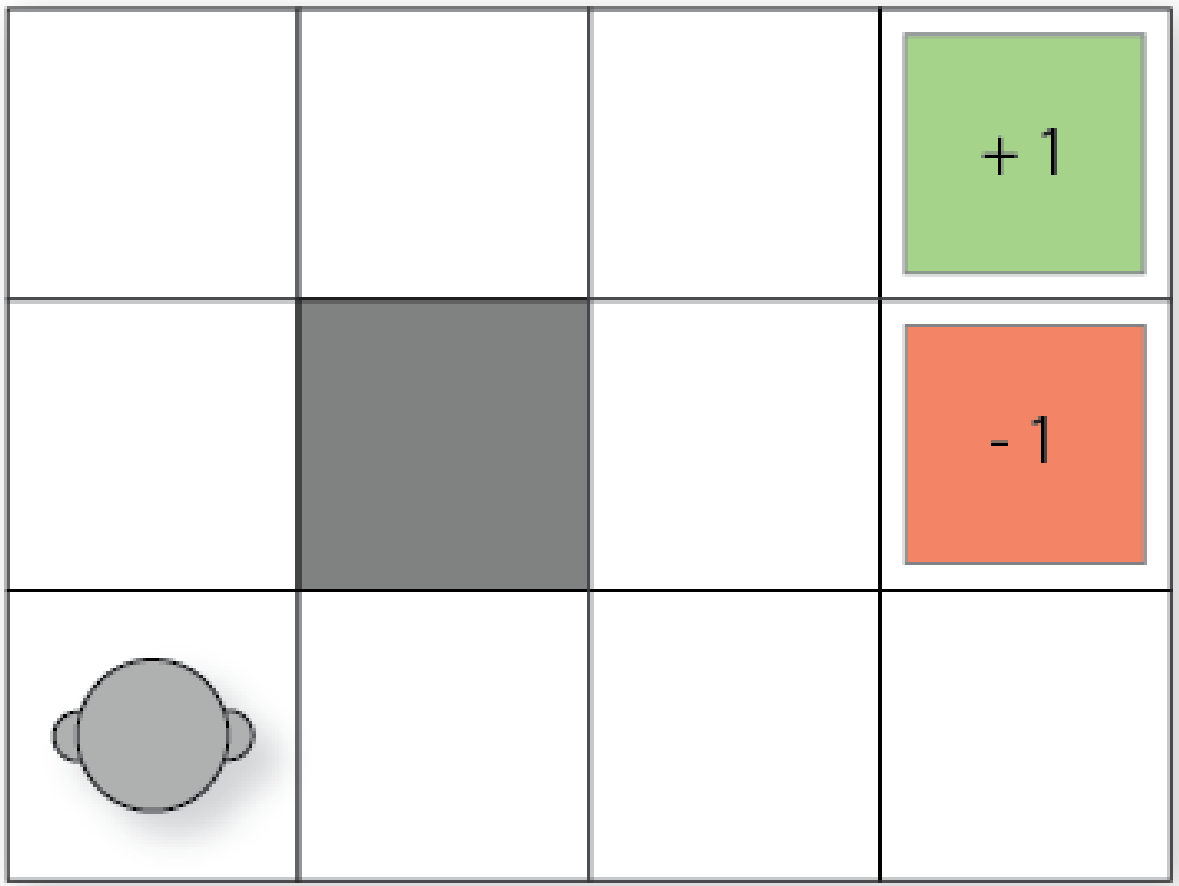
Exercise: Freeway

What would be a good reward for the Freeway game to learn how to get the chicken across the freeway?



Exercise: Gridworld

What would be a good reward for the GridWorld example?



Reward Shaping

In TD learning methods, we update a Q-function when a reward is received. E.g, for 1-step Q-learning:

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

The approach to reward shaping is not to modify the reward function, but to just give additional reward for some moves:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \underbrace{F(s,s')}_{\text{additional reward}} + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

The function F provides *heuristic* domain knowledge to the problem.

$r + F(s,s')$ is the *shaped reward* for an action.

$G^{\Phi} = \sum_{i=0}^{\infty} \gamma^i (r_i + F(s_i, s_{i+1}))$ is the shaped reward for the entire trace.

Potential-based Reward Shaping

Potential-based reward shaping is a particular type of reward shaping with nice theoretical guarantees. In potential-based reward shaping, F is of the form:

$$F(s, s') = \gamma\Phi(s') - \Phi(s)$$

We call Φ the *potential function* and $\Phi(s)$ is the potential of state s.

Theoretical guarantee: this will still converge to the optimal policy.

But! It may either increase or decrease the time taken to learn. A well-designed potential function will decrease the time.

Example: GridWorld

For Grid World, we can use 1 over Manhattan distance to define the potential function:

$$\Phi(s) = \frac{1}{|x(g) - x(s)| + |y(g) - y(s)|}$$

in which $x(s)$ and $y(s)$ return the x and y coordinates of the agent respectively, and g is the goal state.

Even on the very first iteration, a greedy policy, such as ϵ-greedy, will feedback those states closer to the +1 reward. From state (1,2) with $\gamma = 0.9$ if we go East, we get:

$$\begin{aligned} F((1,2),(2,2)) &= \gamma\Phi(2,2) - \Phi(1,2) \\ &= 0.9 \cdot \tfrac{1}{1} - \tfrac{1}{2} \\ &= 0.4 \end{aligned}$$

Reward Shaping Example: GridWorld (continued)

We can compare the Q-values for these states for the four different possible moves that could have been taken from (1,2), using and $\alpha = 0.5$ and $\gamma = 0.9$:

Action	r	$F(s, s')$	$\gamma \max_{a'} Q(s', a')$	New $Q(s, a)$
North	0	$0.9\frac{1}{2} - \frac{1}{2} = 0$	0	0
South	0	$0.9\frac{1}{2} - \frac{1}{2} = 0$	0	0
East	0	$0.9\frac{1}{1} - \frac{1}{2} = 0.4$	0	0.4
West	0	$0.9\frac{1}{3} - \frac{1}{2} = -0.2$	0	-0.1

Thus, we can see that our potential reward function rewards actions that go towards the goal and penalises actions that go away from the goal. Recall that state (1,2) is in the top row, so action North just leaves us in state (1,2) and South similarly because we cannot go into the wall.

But! It will not always work. Compare states (0,0) and (0,1). Our potential function will reward (0,1) because it is closer to the goal, but we know from the lecture on MDPs that (0,0) is a higher value state than (0,1). This is because our reward function does not consider the negative reward.

In practice, it is non-trivial to derive a perfect reward function. If we could, we would not need to even use reinforcement learning – we could just do a greedy search over the reward function.

Q-function initialisation

An approach related to reward shaping is *Q-function initialisation*. Recall that TD learning methods can start at any arbitrary Q-function. The closer our Q-function is to the optimal Q-function, the quicker it will converge.

Imagine if we happened to initialise our Q-function to the optimal Q-function. It would converge in one step!

Q-function initialisation is similar to reward shaping: we use heuristics to assign higher values to 'better' states. If we just define $\Phi(s) = V(s)$, then they are equivalent. In fact, if our potential function is *static* (the definition does not change during learning), then Q-function initialisation and reward shaping are equivalent[1].

[1] Wiewiora: ?Potential-based shaping and Q-value initialization are equivalent.? (JAIR, 2003)

Q-function Initialisation Example : GridWorld

Using the idea of inverse Manhattan distance, we can define an initial Q-function as follows for state (1,2):

$$\begin{aligned} Q((1,2),North) &= \tfrac{1}{2} - \tfrac{1}{2} &= 0 \\ Q((1,2),South) &= \tfrac{1}{2} - \tfrac{1}{2} &= 0 \\ Q((1,2),East) &= \tfrac{1}{1} - \tfrac{1}{2} &= 0.5 \\ Q((1,2),West) &= \tfrac{1}{3} - \tfrac{1}{2} &= -0.16^{*} \end{aligned}$$

Once we start learning over episodes, we will select those actions with a higher heuristic value, and also we are already closer to the optimal Q-function, so will will converge faster.

Summary

1. We can scale reinforcement learning by approximating Q-functions, rather than storing complete Q-tables.
2. Using simple linear methods in which we select features and learn weights are effective and guarantee convergence.
3. Using neural networks offer alternatives in which we do not need to select features, but require a lot of training data and have not convergence guarantees.

Some tips for reinforcement learning in the group project

1. Linear Q-function approximation should work well if the features are not strongly dependent on the random initial state.
2. I would be surprised if using deep Q networks for function approximation was effective because deep neural nets are data hungry and generating enough training data could require weeks or months of computation.
3. Design good reward shaping structures will help, but keep them simple.
4. For some advanced techniques, read Sutton and Barto; in particular stuff on: experience replay, approximate methods (even for value iteration!), and generalised policy iteration.

Reading

- (Chapter 9 On-policy Prediction with Approximation) of *Reinforcement Learning: An Introduction (Second Edition)* [*Sutton and Barto*, 2020]

  Available at:

  `http://www.incompleteideas.net/book/RLbook2020.pdf`

- *Playing Atari with Deep Reinforcement Learning* from DeepMind.
  Available at:

  `https://arxiv.org/pdf/1312.5602v1.pdf`

- Before AlphaGo there was TD-gammon, which was the first paper to combine reinforcement learning and neural networks:

  `http://www.aaai.org/Papers/Symposia/Fall/1993/FS-93-02/FS93-02-003.pdf`