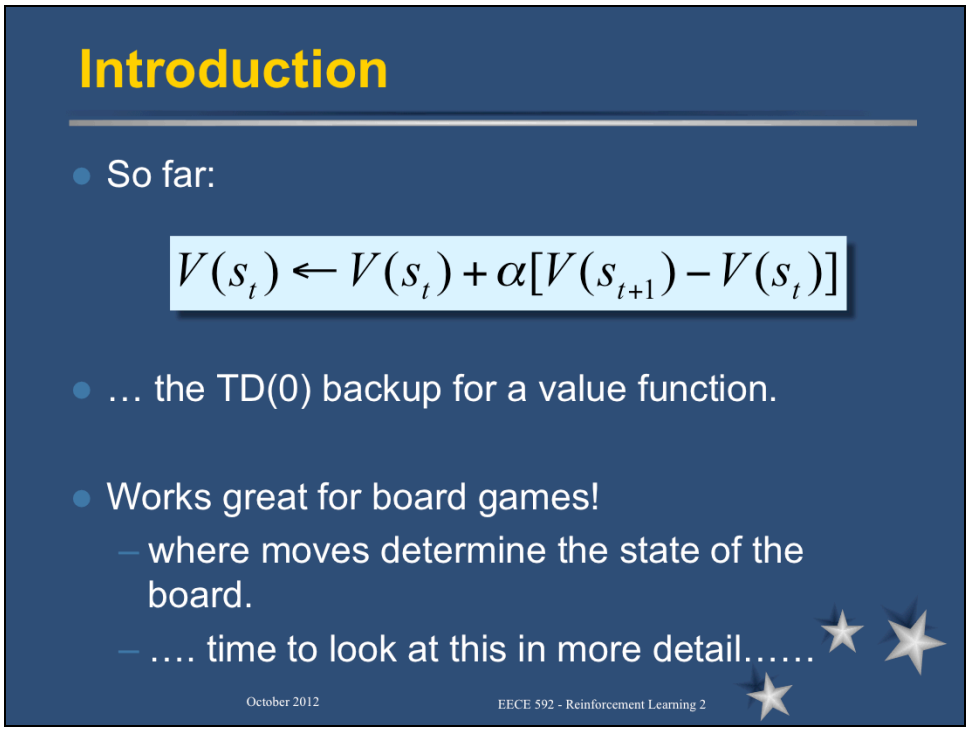
Again, much (but not all) of this chapter is based upon Sutton and Barto, 1998, "*Reinforcement Learning*. An Introduction". The MIT Press



Introduction

In the previous class on RL (reinforcement learning), we saw how a value function could be applied to a board game such as Tic-Tac-Toe. The equation, shown, implements an instance of temporal difference learning applicable to Tic-Tac-Toe.

$$V(s_t) \leftarrow V(s_t) + \alpha[V(s_{t+1}) - V(s_t)]$$

This backup step is defined in terms of transitions from one state to another. In board games, both the state and reward are quite well defined. However, this is not always the case.

Reinforcement learning can also be applied to situations where transitions from one state to another are not as well defined. I.e. the dynamics of the environment are not predictable. This is the case with Robocode which is not a turn-based game! This chapter attempts to provide a more detailed treatment of reinforcement learning as well as to suggest how RL might be applied in Robocode!

Policy

In RL, the notation π, is used to refer to policy. The policy defines the learning agents way of behaving at any given time. It is a function of the rules of the environment, the current learned state and how the agent wishes to select future states or actions. It provides a mapping from perceived states to actions to be taken.

It is said that an optimal policy is one which accumulates the greatest rewards over the long term.

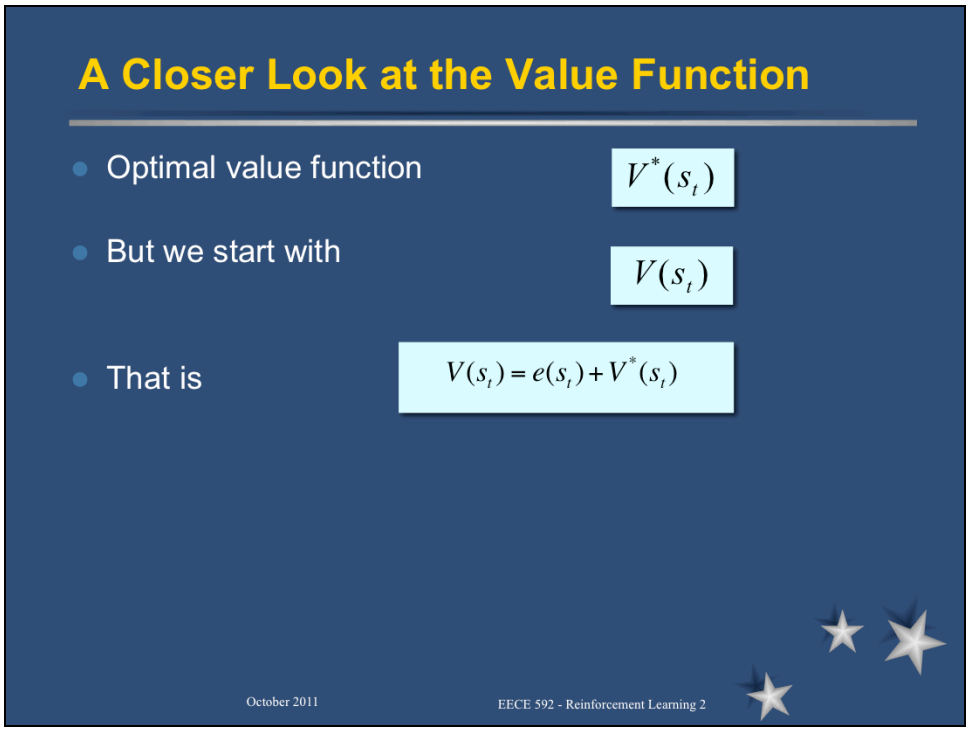
So policy is more than the "rules of the game". The policy is in effect realised by the value function that is being learned using RL and changes as the value function is being learned.

You may come across the terms "on-policy" & "off-policy". The difference is simply that on-policy is both learning and using (to control actions) the same policy that is being learned. "Off-policy" methods learn one policy, while using another to decide what actions to take while learning. E.g. making a exploratory move, but performing updates from greedy moves. The next slide explains why off-policy methods are better than on-policy methods.

Importance of Off-Policy Methods

- Quote from Rich Sutton:
  http://spaces.facsci.ualberta.ca/rlai/files/2010/06/Annual-Report-2013.pdf

- *"Off-policy learning is key to our strategy for scaling reinforcement learning methods to address the more ambitious goals of AI. Off-policy learning means learning about a possible way of behaving (a policy) from following it for a period of time, perhaps only for a fraction of a second, without following it to completion. Moment to moment, the actions an AI agent selects can be seen as parts of many policies, but at most one will be followed to completion. If the agent can learn in parallel about all of the policies, then it can learn vastly more than if it is restricted to learning about only the one that is followed to completion, as it is with classical "on-policy" temporal-difference (TD) algorithms."*



Optimal and Approximate Value Function

When we start learning, the value function is typically randomly initialized.

$V(s_t)$

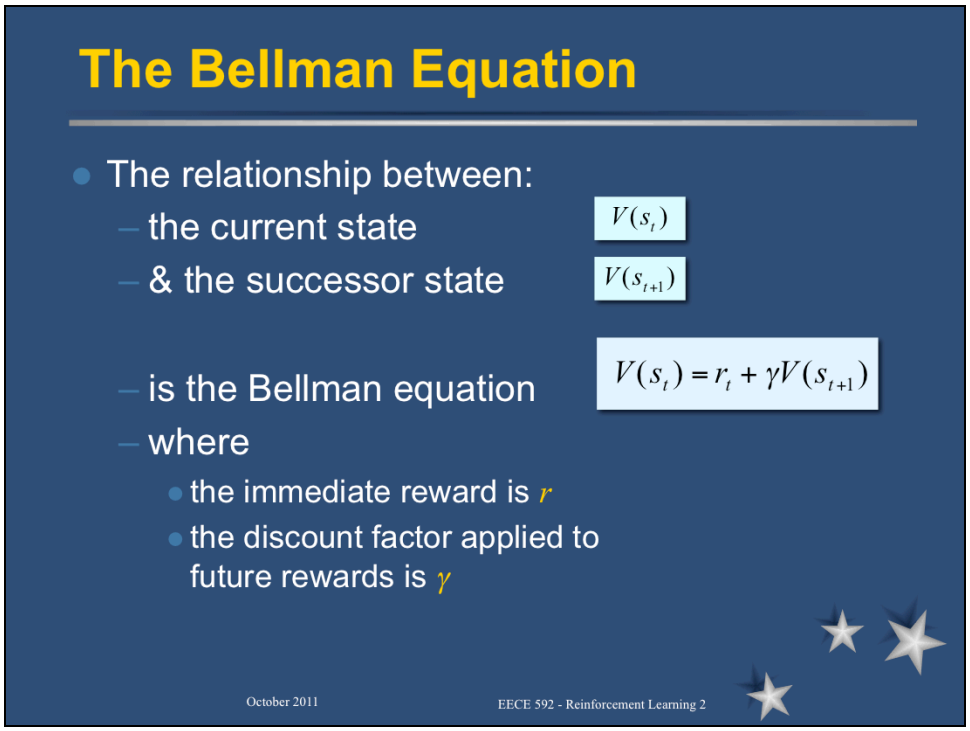
It will be a poor approximation to what we actually want, which is the optimal value function denoted as follows:

$V^*(s_t)$

This is the value function that will lead to us making the best decisions. By best, we mean the best or most reward possible. Thus the approximate value at some state s_t is equal to the true value at that state, plus some error in the approximation. We could write this as follows:

$$V(s_t) = e(s_t) + V^*(s_t)$$

By the way, note that in some literature on the subject, the value function is also referred to more generically as a *utility* function.



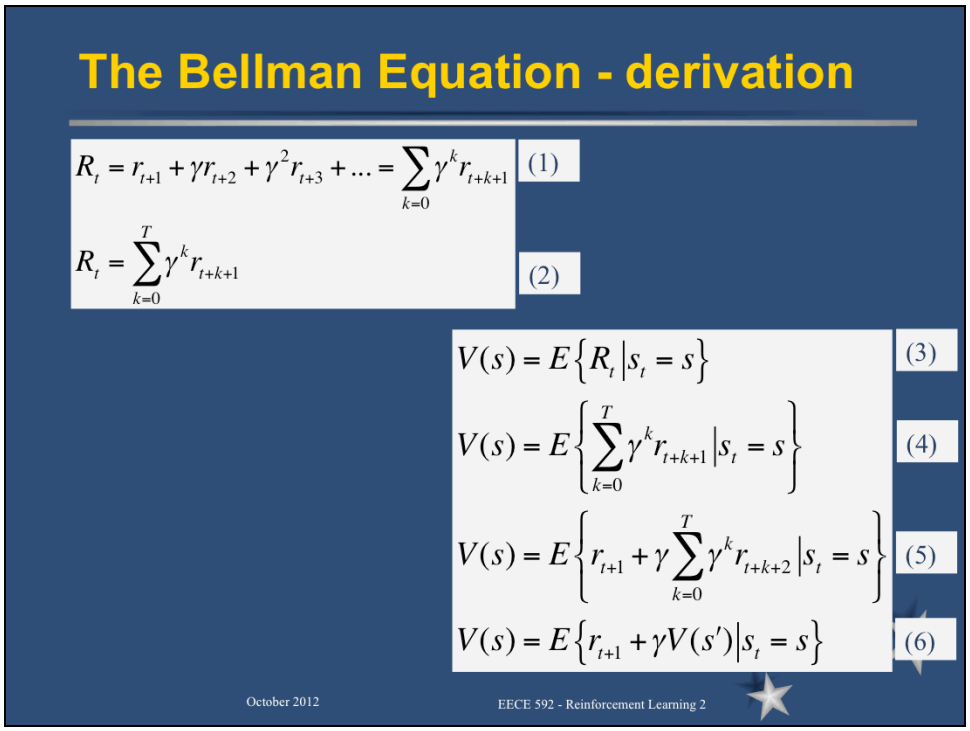
The Bellman Equation

Richard Bellman (1957) defined the relationship between a current state and its successor state. It is often applied to optimization problems that fall in a branch of mathematics known as dynamic programming.

$$V(s_t) = r_t + \gamma V(s_{t+1})$$

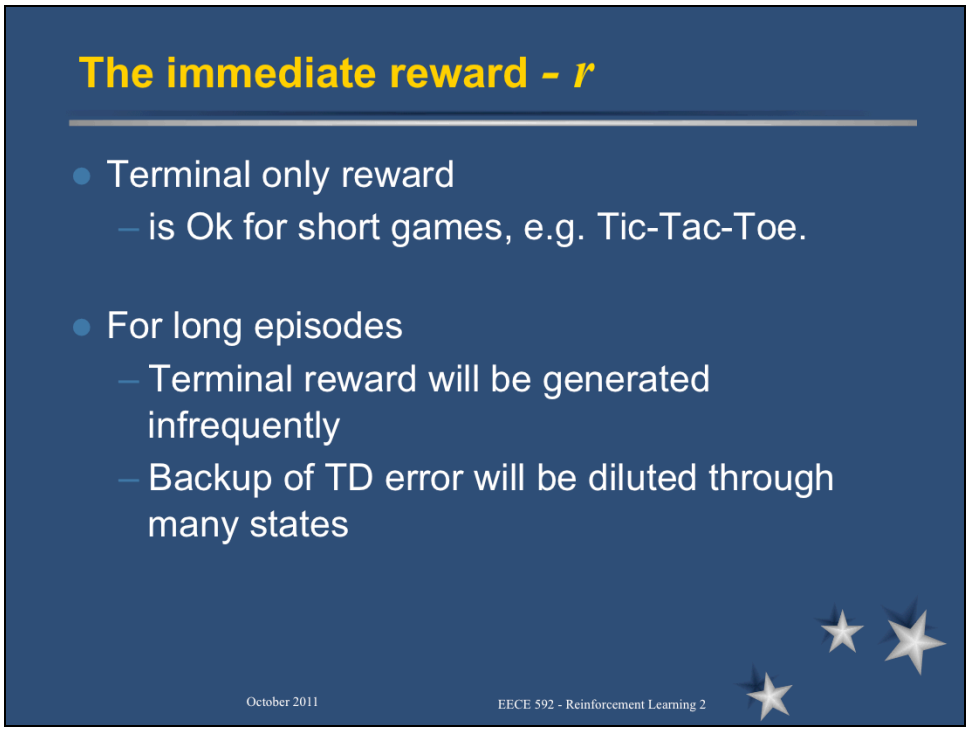
As you can see, the equation is recursive in nature and suggests that the values of successive states are related. In RL, the Bellman equation computes the total expected reward. Typically we want to follow a sequence of events (a policy) that generate the maximum rewards (the optimal policy).

Here we introduce two new terms. The immediate reward *r* and the discount factor γ.



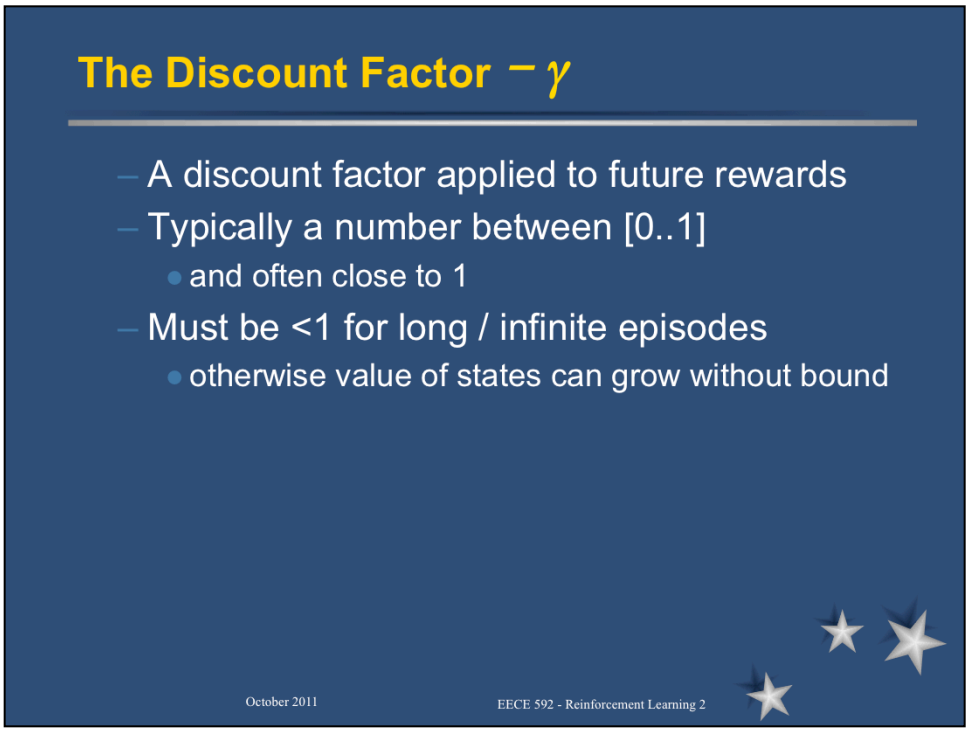
Recall that a value function is attempting to predict the long term accumulated reward that may be expected from rewards encountered in future states. (1). This can be summarized by (2).

We can define the value function in terms of these rewards. (3). Here *E* represents the *expected return*. It depends upon how the backups are performed and also the policy being followed. E.g. *full* backups or *sample* backups. We note that R_t in (3) and (4) can be replaced by (1) and (2). If we separate out current (*s*) and successor states (*s'*), we get (5). Leading to the form of the Bellman equation (6).



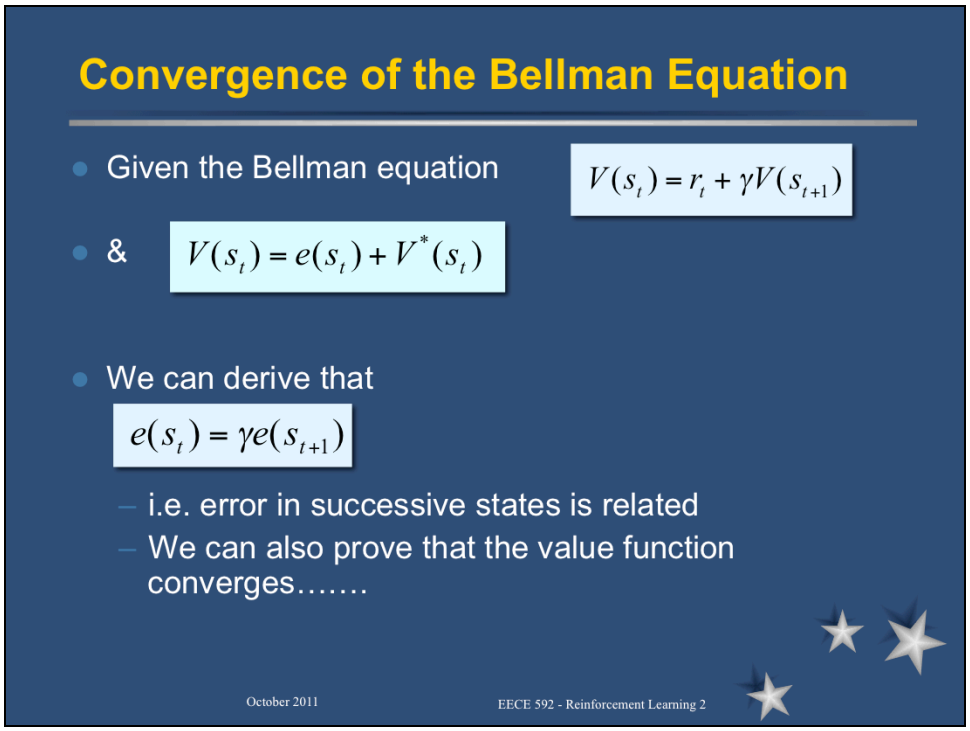
The immediate reward - r

The immediate reward is an optional element of reinforcement learning. If you consider RL as a credit (or blame) assignment problem, you begin to appreciate some of the difficulties associated with learning from just a terminal reward signal. Say for example you come home and find your dog has made a mess inside the house. If you reproach the dog, how does it know which of its hundreds of actions throughout the day is the reason for being told it's a bad dog! On the other hand, if you told the dog immediately after the offending act, the dog is more likely to understand why you're upset. This scenario describes the potential problems with long episodes, where a terminal reward is generated after many hundreds of actions. In some cases, there may even be no terminal, if say the task continues or is meant to continue forever (so called infinite horizon problems). E.g. in Robocode, generating a reward only upon winning a battle is unlikely lead to good learning. This is because the reward will be rarely generated unless the Robot is already smart enough to stay alive!



The Discount Factor - γ

The discount factor is applied to the total accumulated reward represented by $V(s_{t+1})$. It is a number in the range 0 to 1, typically close to 1 and is used to basically weight future rewards. When γ is 0, only the immediate rewards are used in the determination of the next action/state as any future rewards are ignored. When γ is 1, future rewards are as significant as immediate ones. Note that if the decision process has many steps, or is in fact infinite, then we have to have a discount factor less than one. Otherwise the sum of the future reinforcements for each state would also approach infinity.



Convergence

Let the error in the value function at any given state be represented by

$e(s_t)$

Then

$V(s_t) = e(s_t) + V^*(s_t)$

and

$V(s_{t+1}) = e(s_{t+1}) + V^*(s_{t+1})$

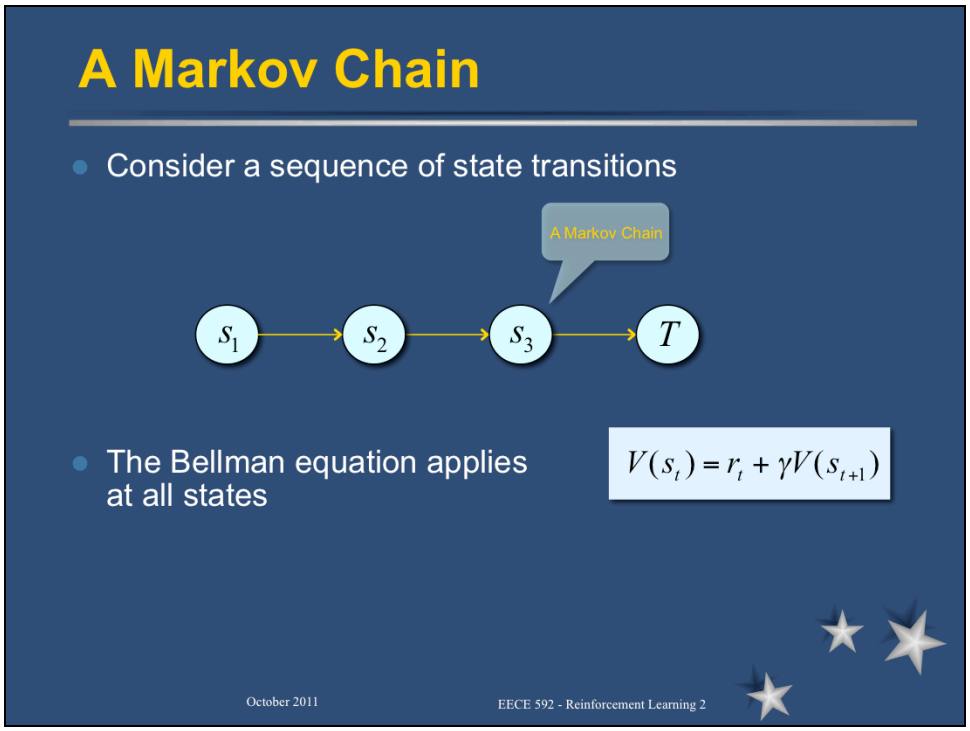
Using the Bellman equation

$V(s_t) = r_t + \gamma V(s_{t+1})$

We can derive that

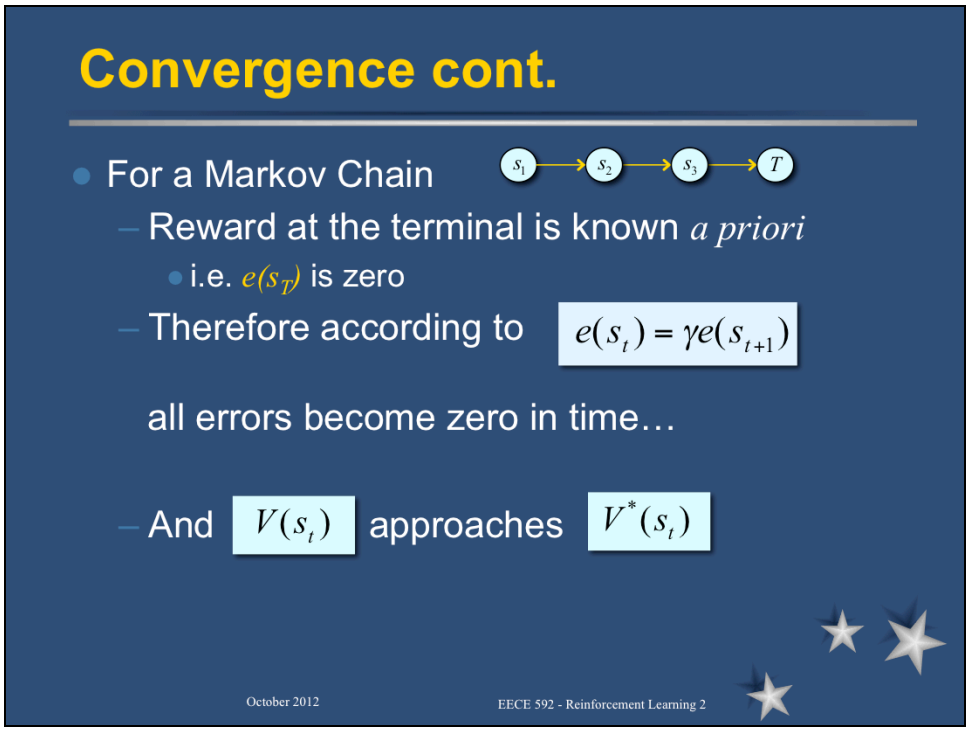
$e(s_t) = \gamma e(s_{t+1})$

Concluding that the error in successive states is related. But we know that already, since we know that the future rewards $V(s)$ are themselves are related. However, the interesting thing is that we can show that these rewards actually converge. I.e. after enough training no longer causes them to change. See the next two pages.



Markov Chain

The diagram shows a sequence of state transitions. Definition: When the decision made at any given state does not require any previous knowledge or history of prior states, this is known as the Markov property. Here we see a sequence of states in which there is no choice but to go from one state to the next. This is a Markov chain.



Convergence continued

For a decision process where a reward is available at a terminal state, we know that the reward is known precisely. I.e. there is no error in the reinforcement signal. I.e. $e(s_T)$=0 where s_T is terminal.

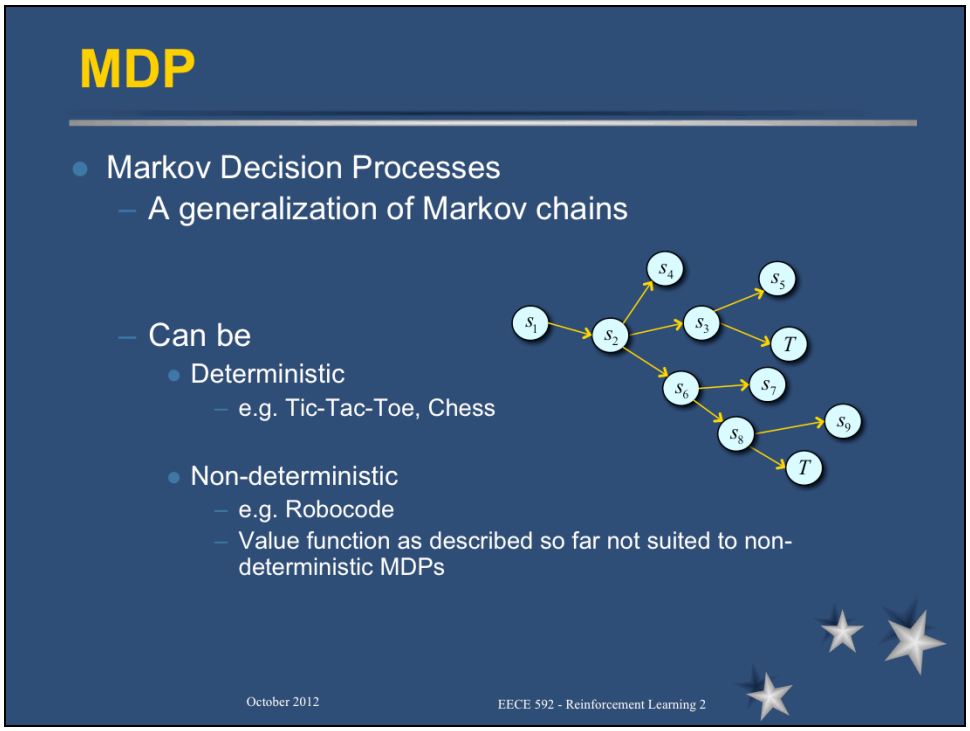
Given $e(s_t) = \gamma e(s_{t+1})$

we deduce that with enough sweeps through the state space, I.e. enough opportunities to learn, we will eventually reach the condition where $e(s_t)$ is zero for all t. When this is the case, then we know we have the optimal value function. So $e(s_t)$ is zero in the following equation.

$$V(s_t) = e(s_t) + V^*(s_t)$$

For the optimal value function, we can say that we have state values that satisfy the Bellman equation, for all t:

$$V(s_t) = r + \gamma V(s_{t+1})$$

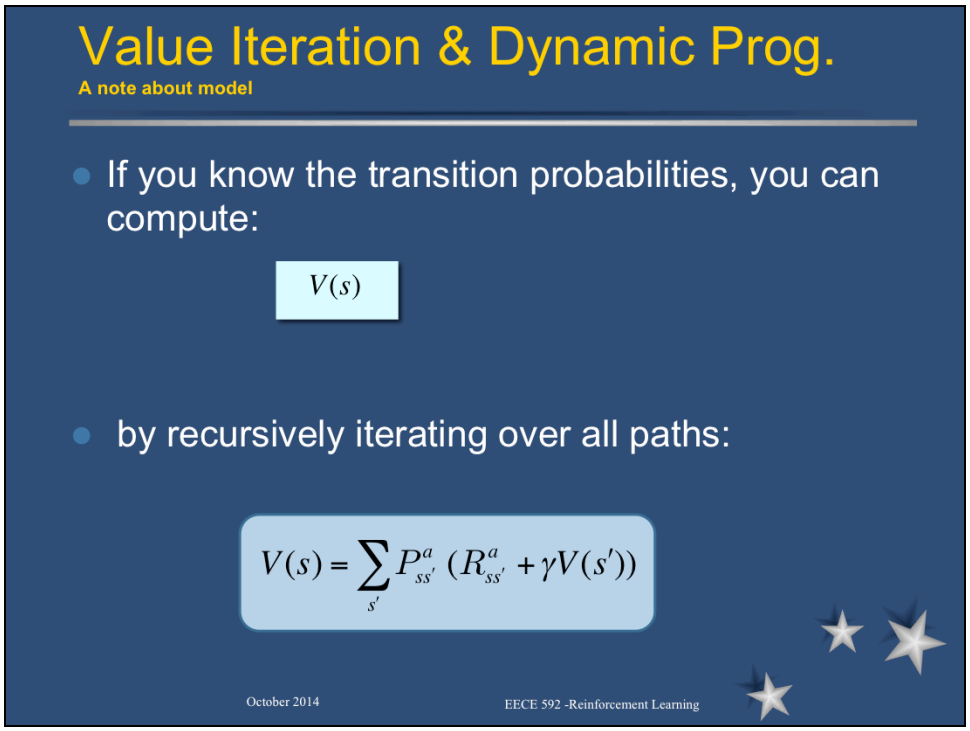


MDP

Markov Decision Processes, or MDPs as they are referred to, describe any class of problems where some state *s*, belonging to a set of states *S*, upon taking some action *a*, will lead to some other state *s'* with probability *P*.

When the probability is 1, we say the MDP is deterministic. I.e. the RL agent will always, when in state *s*, after taking action *a*, end up in the same state *s'*. However the MDP may also be non-deterministic. In this case, given state *s* and action *a*, the agent may not always end up in the same state *s'*. This is actually the case with any dynamic environment such as Robocode, where the actions of other tanks will also affect the resultant state.

For non-deterministic decision processes it turns out that the value function as described is not really suitable. I'll refer you to http://courses.ece.ubc.ca/592/PDFfiles/rltutorial.pdf for the details.

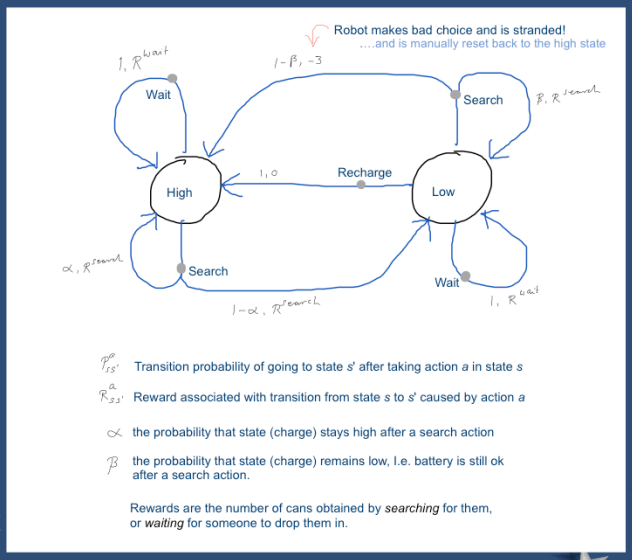


The bottom equation is the Bellman equation for value iteration. P is the transition probability that an action a, taken in state s will lead to a successor state s'. R is the reward that follows a state transition from s to s' after taking action a. and γ the discount factor. The summation is performed over all transitions, each path weighted by its transition probability.

Doing so can determine the optimal value function. The problem with this approach is that requires *apriori* knowledge of the transition probabilities and the rewards. Without a model of the environment, this may not be possible.

A key element of Dynamic Programming methods is the use of such a model. I.e. knowledge of the complete probability distributions P of all possible transitions.

Recycling Robot Example



Sutton and Barto, 1998, "*Reinforcement Learning*. An Introduction". The MIT Press

For any policy π $$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} P^{a}_{ss'} \left[R^{a}_{ss'} + \gamma V^{\pi}(s') \right]$$

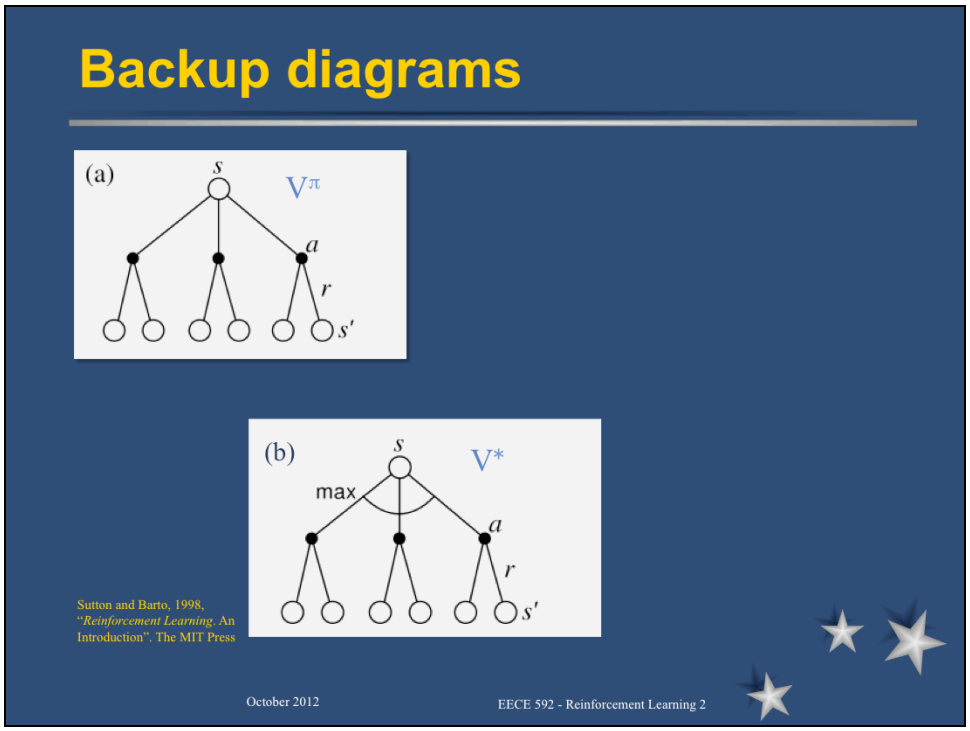
For the optimal policy $$V^{*}(s) = \max_{a} \sum_{s'} P^{a}_{ss'} \left[R^{a}_{ss'} + \gamma V^{\pi}(s') \right]$$

For example, $V^*(s)$ for the robot states is:

$$V^{*}(\mathrm{h}) = \max \begin{cases} P^{\mathrm{s}}_{\mathrm{hh}} \left[R^{\mathrm{s}}_{\mathrm{hh}} + \gamma V^{*}(\mathrm{h}) \right] + P^{\mathrm{s}}_{\mathrm{hl}} \left[R^{\mathrm{s}}_{\mathrm{hl}} + \gamma V^{*}(\mathrm{l}) \right], \\ P^{\mathrm{w}}_{\mathrm{hh}} \left[R^{\mathrm{w}}_{\mathrm{hh}} + \gamma V^{*}(\mathrm{h}) \right] + P^{\mathrm{w}}_{\mathrm{hl}} \left[R^{\mathrm{w}}_{\mathrm{hl}} + \gamma V^{*}(\mathrm{l}) \right] \end{cases}$$

$$V^{*}(\mathrm{l}) = \max \begin{cases} P^{\mathrm{s}}_{\mathrm{ll}} \left[R^{\mathrm{s}}_{\mathrm{ll}} + \gamma V^{*}(\mathrm{l}) \right] + P^{\mathrm{s}}_{\mathrm{lh}} \left[R^{\mathrm{s}}_{\mathrm{lh}} + \gamma V^{*}(\mathrm{h}) \right], \\ P^{\mathrm{w}}_{\mathrm{ll}} \left[R^{\mathrm{w}}_{\mathrm{ll}} + \gamma V^{*}(\mathrm{l}) \right], \\ P^{\mathrm{re}}_{\mathrm{lh}} \left[R^{\mathrm{re}}_{\mathrm{lh}} + \gamma V^{*}(\mathrm{h}) \right] \end{cases}$$

Where states={h,l} (high, low) and actions={s,w,re} (search, wait, recharge)
For fun, try substituting *P* for the actual probabilities as shown in the diagram.



Backup diagrams

Sutton and Barto introduce the notion of backup diagrams. These can be helpful in understanding RL, but first you have to understand their notation! Take these for example. (a) shows the backup diagrams for V^π whereas (b) applies to only the optimal (greedy) policy. In (a), the update of the root node, i.e. the backup, is a function of all successor nodes. A so-called *full* backup. In (b) the backup is from a *sample* of future nodes. The arcs indicate the point at which a choice has to be made. For the optimal policy, this is the *greedy* choice. Both upper and lower diagrams are applicable to dynamic programming and not TD learning.

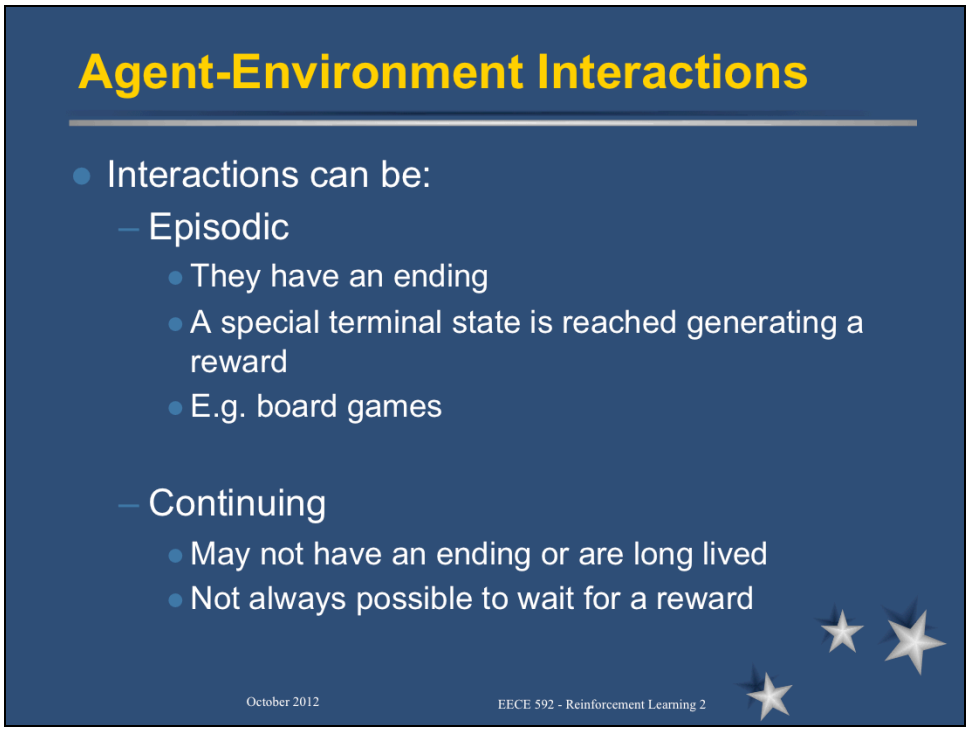
Which Value Function?

$$V(s_t) \leftarrow V(s_t) + \alpha[V(s_{t+1}) - V(s_t)]$$

- This update function does not work for Robocode. (*Why not?*)
- Then what does?

So first of all, a good question to ask is why does this update function not work work for Robocode?

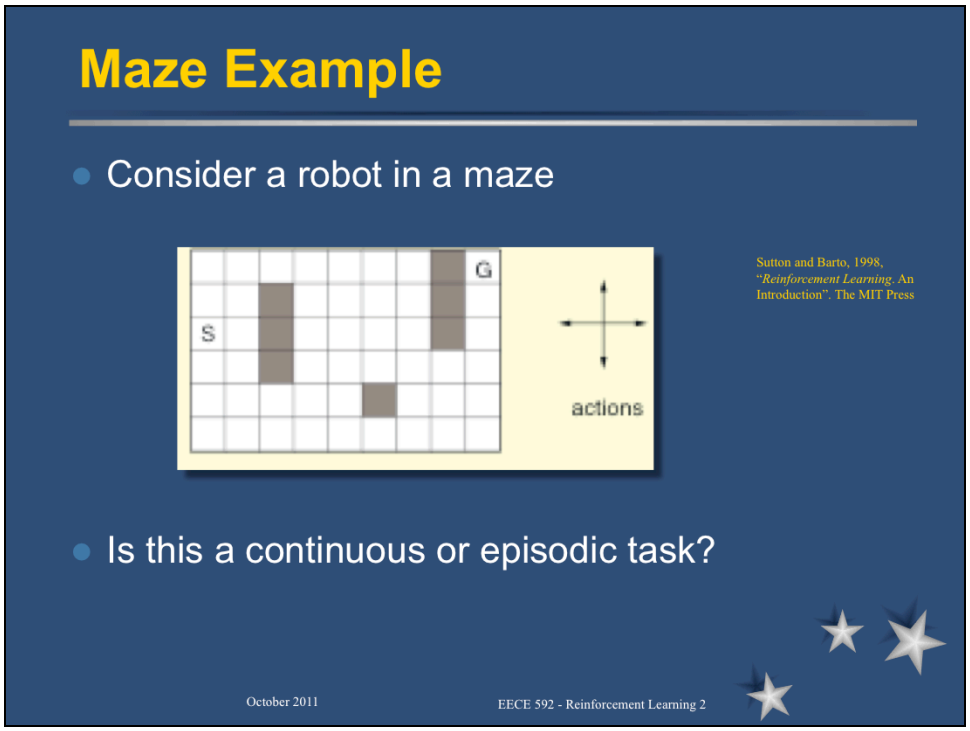
Then the next question is, what does the *utility* function that can be applied to Robocode look like?



Agent-Environment Interactions

There are essentially two categories of interactions that an agent may have with its environment. An agent learning to play a board game for example will always come to an end. The game is either won, lost or drawn. A reward signal is generated upon reaching one of these terminal states. Of course, you could play again and continue learning in this next game too. Each game in this case is considered to be an episode. Interactions within each episode in this case, proceed in a step-by-step fashion.

Then there are so called continuous tasks. Like the futuristic soda can recycling robot from a previous slide. Its goal is to wander around an office environment looking for and collecting empty soda cans. Such a task could theoretically continue forever. Assuming that the robot runs on rechargeable batteries, there is of course one terminal state reached when the robot runs out of charge and is left stranded. This would obviously generate a large negative reward. However, we don't really want to reach this state too often if at all! So relying on this terminal reward signal may not be so effective if used as the only reward. Instead, we may want to inject reward signals upon certain strategic actions such as finding a can, reaching a charging point or if the charge level reaches dangerously low.

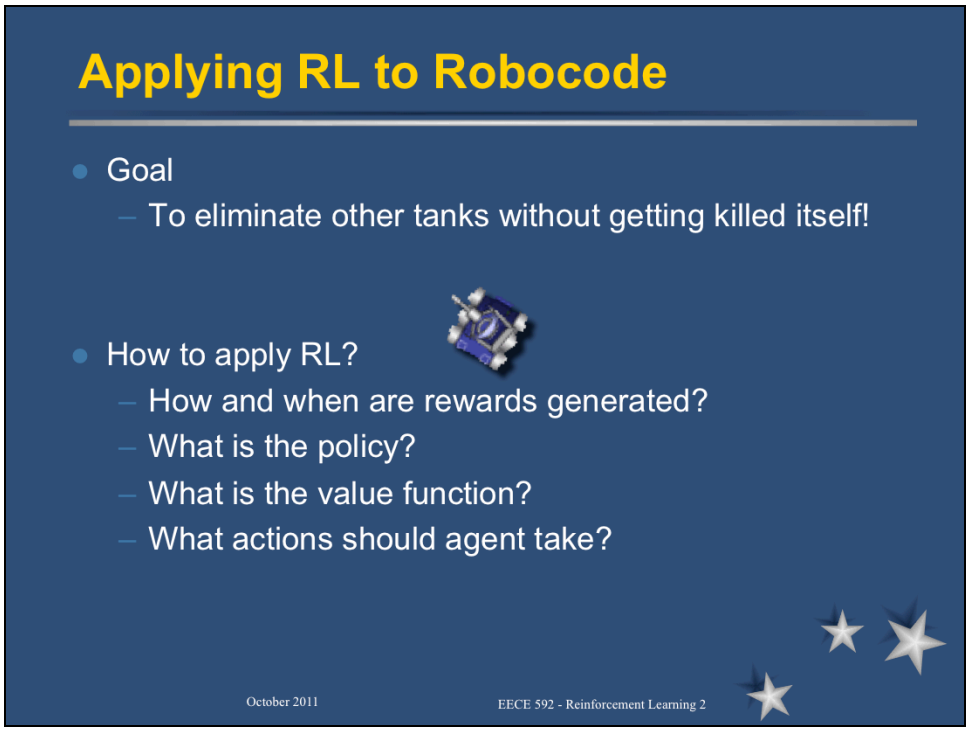


Maze example

The diagram shows a very simple maze. The goal of the robot is to go from the starting point **S**, to the exit **G** in the shortest number of steps. In each square, the actions available to the robot are to go left, right, up or down, except of course where blocked either by the shaded areas or the walls of the maze.

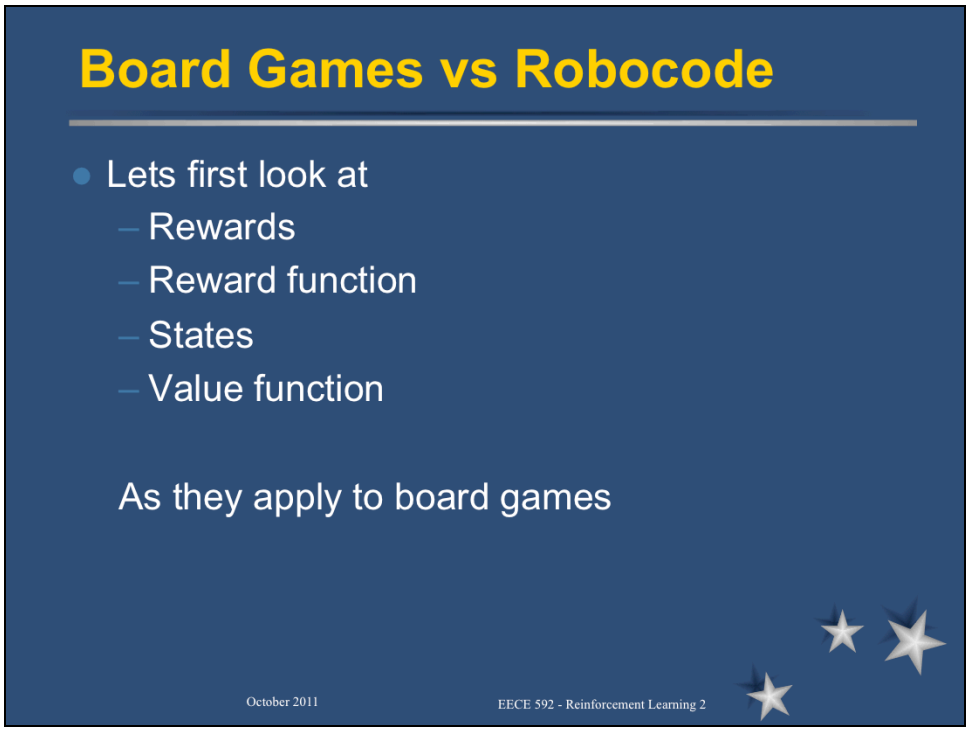
We want to use reinforcement learning to find an optimal policy and one might consider that the problem can be naturally broken down into episodes. Where a reward of +1 is generated upon reaching **G** and 0 at all other times. TD learning is used with greedy action selection based upon 1-ε. However, it turns out that such an approach is quite slow in reaching a suitable policy. Why might this be?

One reason may be that the terminal state is not frequently or easily reached, or that each episode is quite long running. In this case it might be worth considering treating this as a continuous task. At least provided some immediate rewards. Can you think of suitable events that could trigger short term rewards?



To understand how to apply RL, we need to understand what rewards, states, policy, etc. mean in Robocode. Since Robocode is not a turn-based game, it is fair to say that when and how rewards are generated, what the terminal states are, how a policy is learned is not trivial.

In the previous class on RL we looked at a Tic-Tac-Toe and saw how RL learning could be applied to a simple turn-based game board game. To help our understanding, this chapter compares various aspects of RL as they apply to board games and as they apply to Robocode.

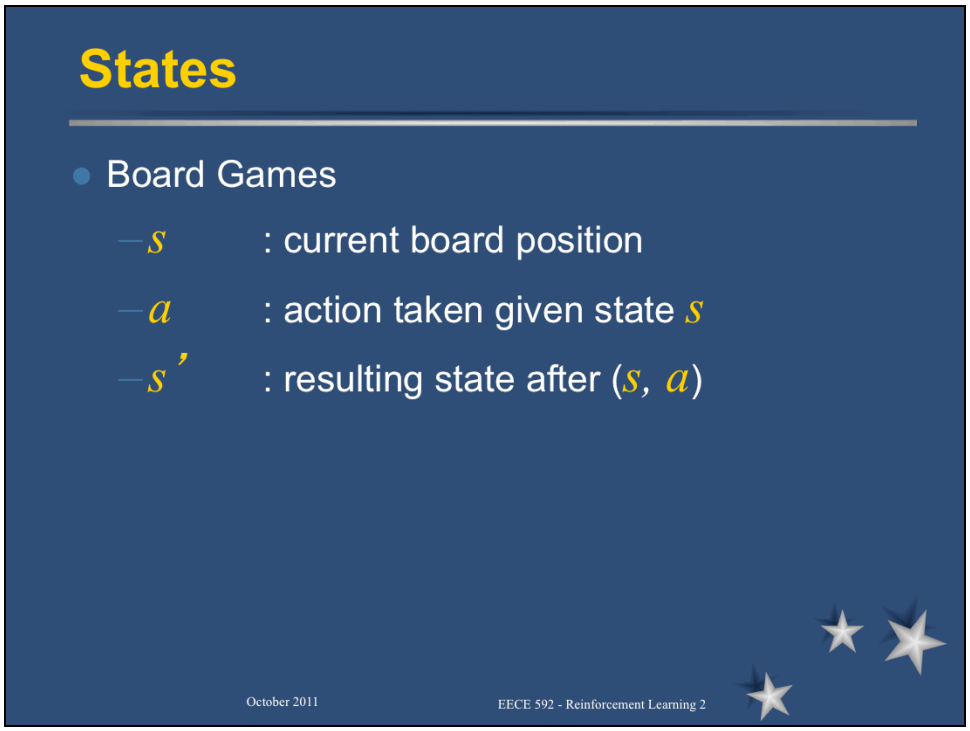


Lets take a look at rewards, the reward function, states and the value function as applied to a board game.

Board Game Rewards

In board games, the reward signal is typically generated at the end of the game. In RL this is also called an episode and board games fall into the class of problems known as episodic tasks. Terminal states of an episode are typically the sources of reward.

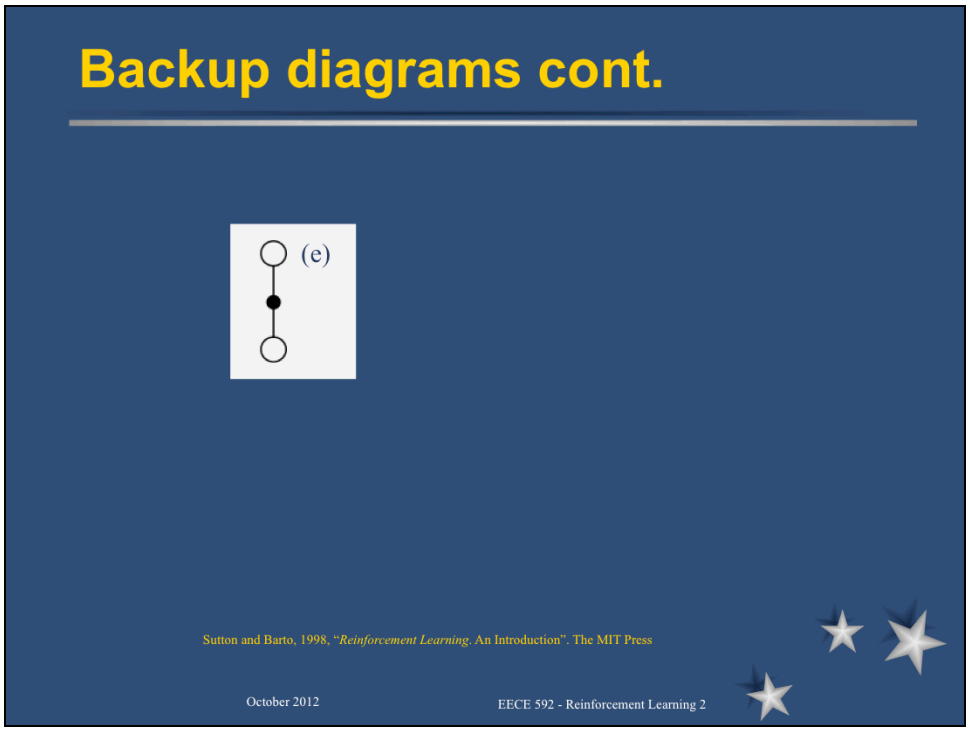
Note that this does not preclude generating rewards in non-terminal states. We could if we wanted to, say in a chess game, generate a positive reward upon capturing the opponents Queen. (Of course, by doing so we are assuming that such a move is always going to lead to a stronger position.)



Board Game States

For a board game, the states and actions are well defined. The state of a board game is for example, the configuration of the board after an opponent has taken a turn.

Here s represents the position for which the agent must select some action a. The environment, or rather the opponent, will then leave the agent in state s' after taking their turn. The function $V(s)=V(s) + \alpha[V(s') - V(s)]$ backs up the reward signal r, from a future state s' back to the present state s following selection of action a. This is how we arrive at a policy that defines the set of actions for all states that lead to an optimized policy.

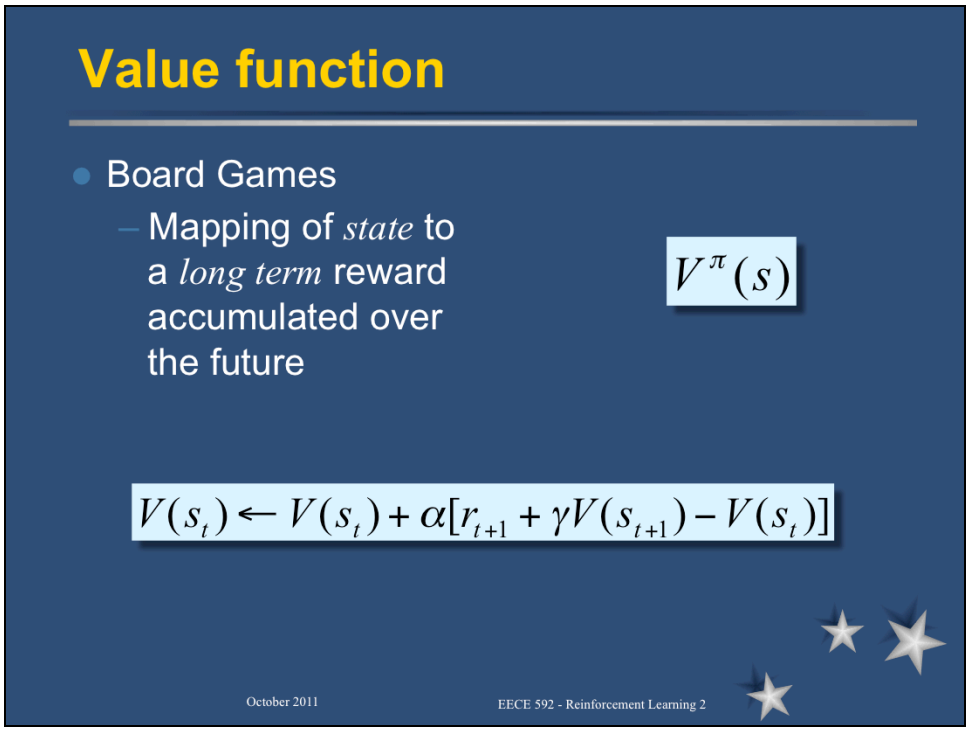


(e) Is the backup diagram for TD(0) (value function). It shows that there is a single backup from *state* to *state* as would be applicable for a board game.

Reward function

The difference between a reward function and a value function in reinforcement learning is worth noting. A reward function maps the perceived state of the environment to a single *immediately* available reward.

In general, the value function on the other hand maps the perceived state of the environment to the long term reward that can be expected to be accumulated over the future, starting from that state.



Board game value function

Formally in reinforcement learning, $V^{\pi}(s)$ represents the state-value function for policy π. It represents the long term reward, predicted to be accumulated over the future from the current state s, until a terminal state is reached.

As with the reward function, more generally, the value function applies not just to mapping of state to reward but to a state-action pair to a reward.

The equation above shows the form of the state-value function as applied to temporal difference based learning TD(0). The term r_{t+1} represents a potential immediate reward and α as before, represents a sort of learning rate coefficient.

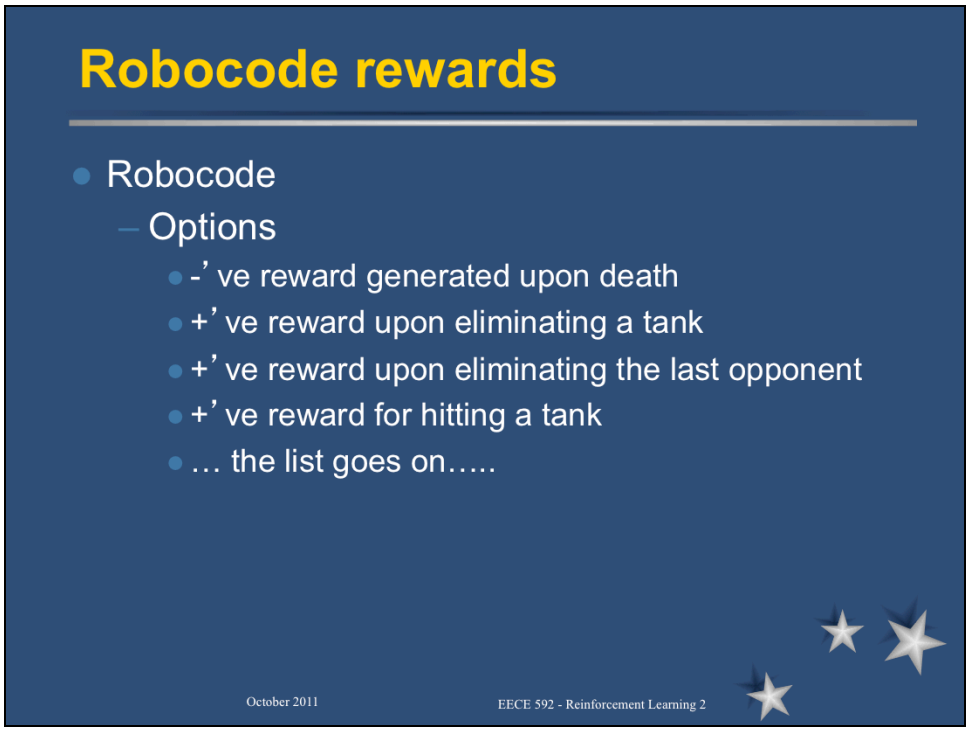
In practice, whether the term r_{t+1} is used or not depends really on the application. For example in the tic-tac-toe, there is no other reward signal except the value from a future state.

As we mentioned before, the term $\gamma V(s_{t+1})$ represents a discounted future reward that can be expected by making a choice which includes moving to state s_{t+1}.

Boards Games vs Robocode

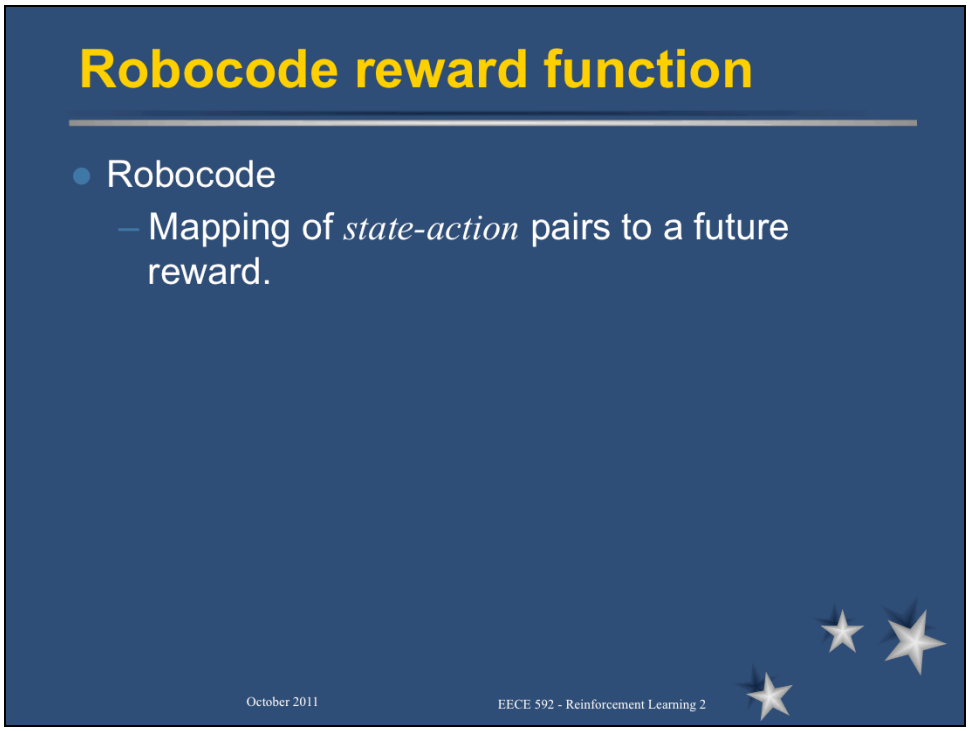
- Now lets look at
  - Rewards
  - Reward function
  - States
  - Value function

  As they apply to Robocode



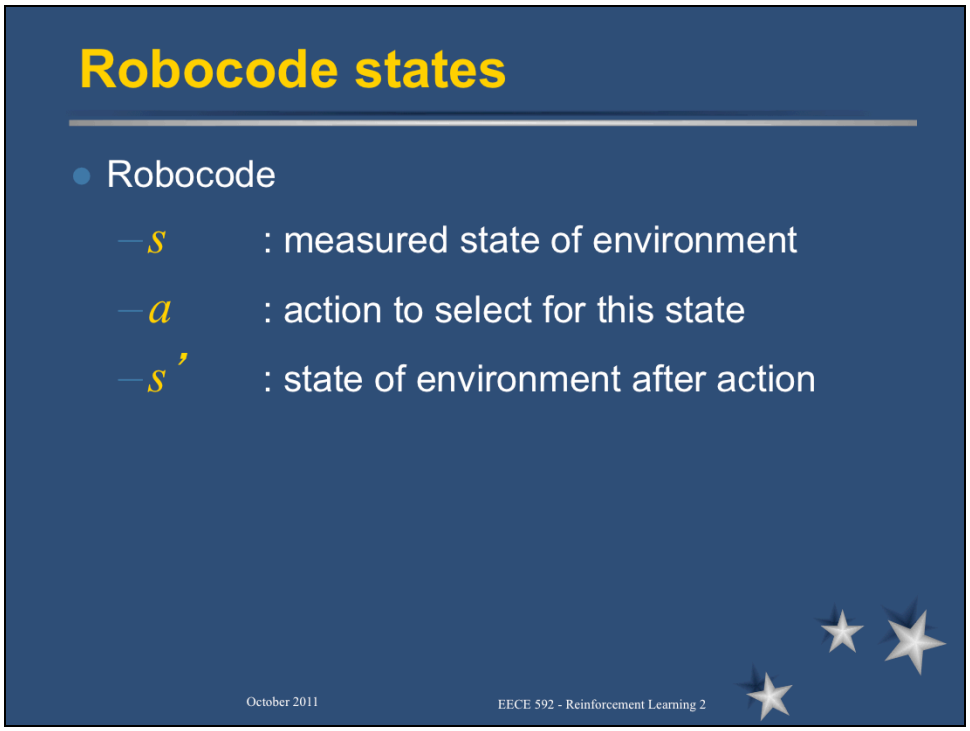
Robocode rewards

Robocode does not quite fit the pattern of episodic tasks. There are many potential sources of rewards. These may occur either during a battle or possibly at the end of the battle (terminal states). Which maybe either due to being eliminated or eliminating all other opponents. Theoretically, a robocode battle could go on indefinitely or at least in practice, a very long time. This has implications if we rely on a single reward signal generated at the end.



Robocode reward function

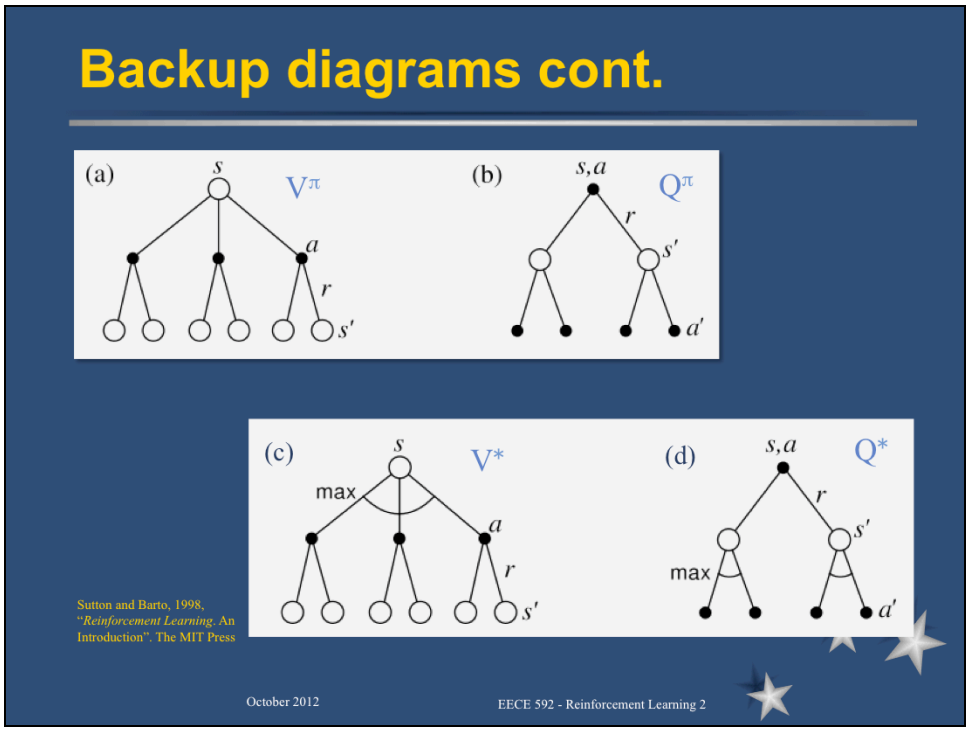
Important! Generally, the reward function applies not just to mapping of state to a reward but also mapping a state-action pair to a reward. What we are saying is that the utility function will map state-action pairs to future rewards or utilities. This kind of function is also referred to as an action-value or *Q*-function. (Q-Learning, Watkins 1989)



Robocode states

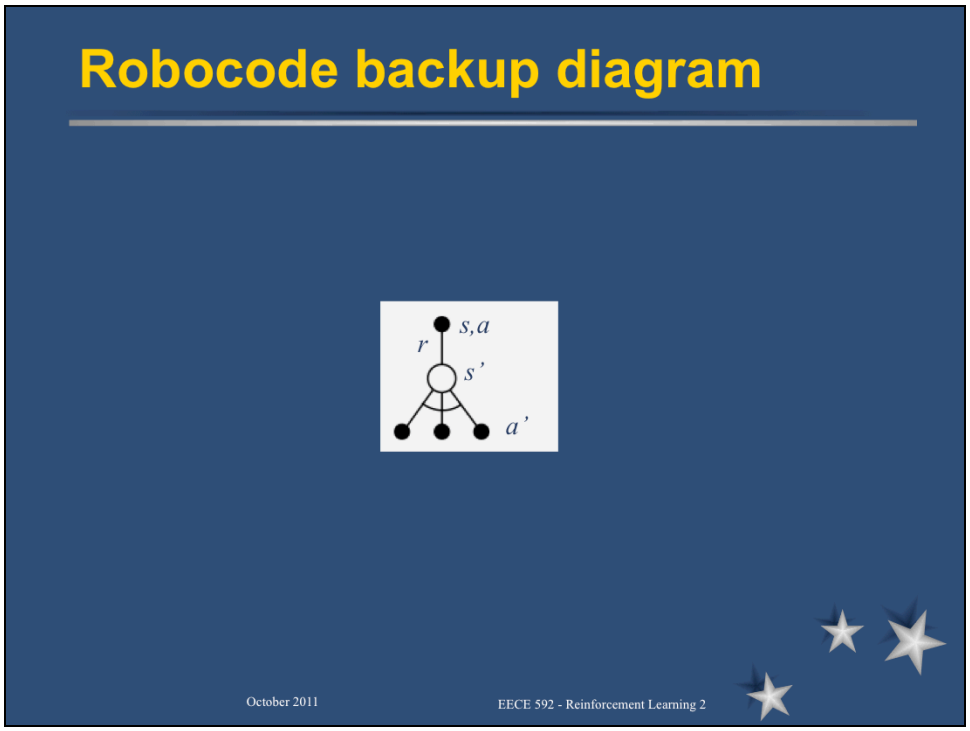
In a board game, the state *s'*, that represents the state of the game after the opponent has reacted can be measured. The game state is static and does not change until the agent again makes a move.

In Robocode it is not possible to determine the state that represents the environment after our tank has taken an action. I.e. *s'* **cannot** be predicted. The environment is dynamic. Other tanks maybe active in the environment and continually altering the environment. A value function based only on states is not useful. Since the tank has no way of forcing the environment into a desired state. Thus the temporal difference learning step $V(s)=V(s) + \alpha[V(s') - V(s)]$ is also not useful. I.e. modelling the environment as a series of state transitions will not work for Robocode.



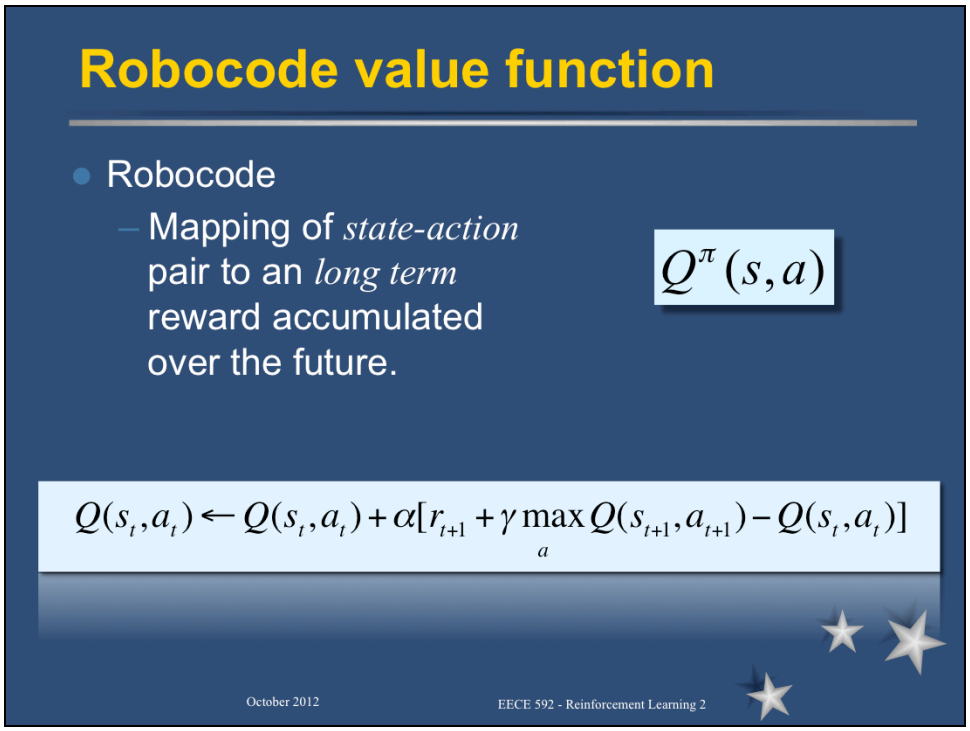
Backup diagrams – state-action transitions

We've already seen some of these in an earlier slide. The two new ones (b) and (d) represent transitions from *state-action* to *state-action* when applying dynamic programming techniques such as value iteration.



Robocode backup diagram

The backup diagram for Q-learning (e.g. for robocode) is shown. Each time the backup updates the root node, in this case representing a state-action pair. You then sense the next state *s'* and select one action, *a'* allowed in that state (according to some policy, probably the ε-greedy policy). The backup is then from (*s'*,*a'*) to (*s*,*a*). There maybe a reward r_{t+1} (not shown) associated with (*s'*,*a'*).

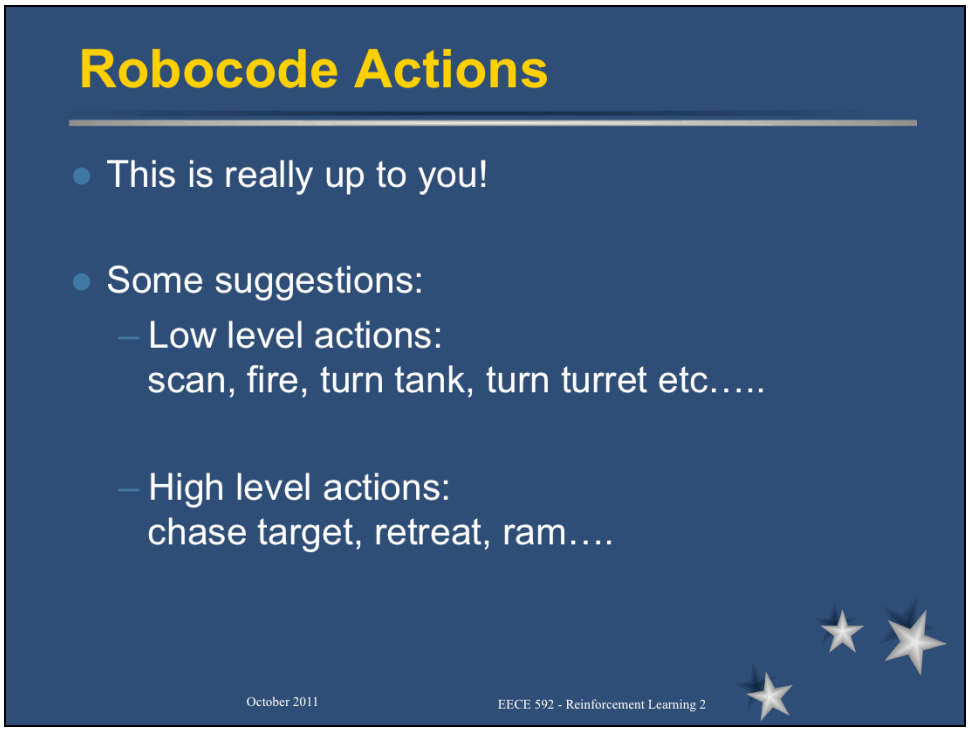


Robocode value (utility) function

For Robocode, we need a value function that maps state-action pairs to future rewards. Formally, $Q^{\pi}(s,a)$ represents the action-value function for policy π. This is the general definition of a reward function. It permits a reward to be backed up from a future state-action selected. Unlike the state-value function, it does not require knowledge of the state of the environment following an action. It backs up reward from state-actions defined by the agent itself.

The equation above show the general form of the action-value function as applicable to temporal difference based learning TD(*0*). Again the term γ, represents a discount factor and is applied to a future reward. The term r_{t+1} represents a potential immediate reward and α as before, represents a sort of learning rate coefficient.

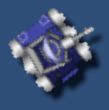
In practice, whether the term r_{t+1} is used or not depends really on the application. For example in the tic-tac-toe, there is no other reward signal except the value from a future state. Which also is not discounted (i.e. γ=0). In robocode, as well as a discounted value from a future state, $\gamma \max_a Q(s_{t+1}, a_{t+1})$ you may wish to generate an immediate reward r_{t+1}, for example in response to a fired shell impacting its target.



In a board game, the range of actions is quite well defined and typically determined by the set of rules that apply to any given game. In robocode, the set of low level actions supported are provided by the robocode environment itself. Any of these could be used for as actions for your own robocode project. However, you may also want to define some high level actions, or preprogrammed procedures in your code. For example you may try to implement a "chase" algorithm which will attempt to chase after a locked on target. Or some preprogrammed steps for taking evasive action. Each of these could be treated as separate actions within robocode.

Robocode and Episodic Tasks

- Consider Robocode

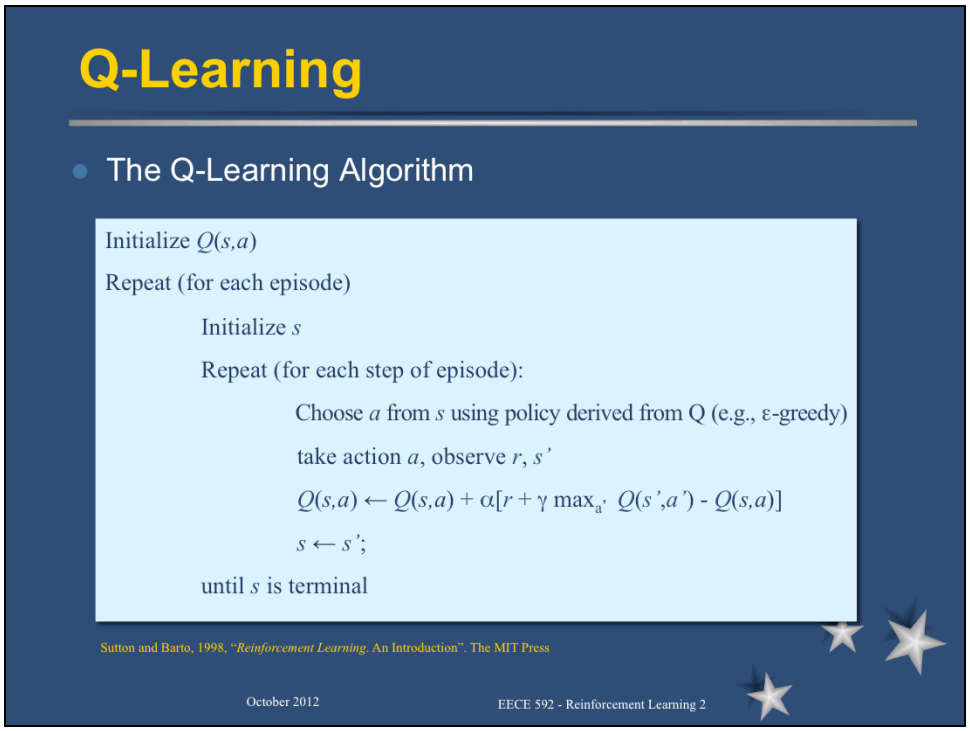


- Is this a continuous or episodic task?
- Is it a deterministic MDP?
  - Most likely a continuous task
  - Most likely a non-deterministic process

Like the maze example, with Robocode too it seems possible to formulate the problem into a series of episodes. Terminal states would be when the tank is either eliminated (a negative reward) or is the last surviving tank (positive reward). Also like the maze example, encountering a positive terminal state might be too infrequent or too unlikely to result in fast learning.

As such the task is better described as a continuous learning problem.

The decision process appears to be non-deterministic. I.e. it is not possible with certainty to say if an action in any given state will always lead to the same resulting state.

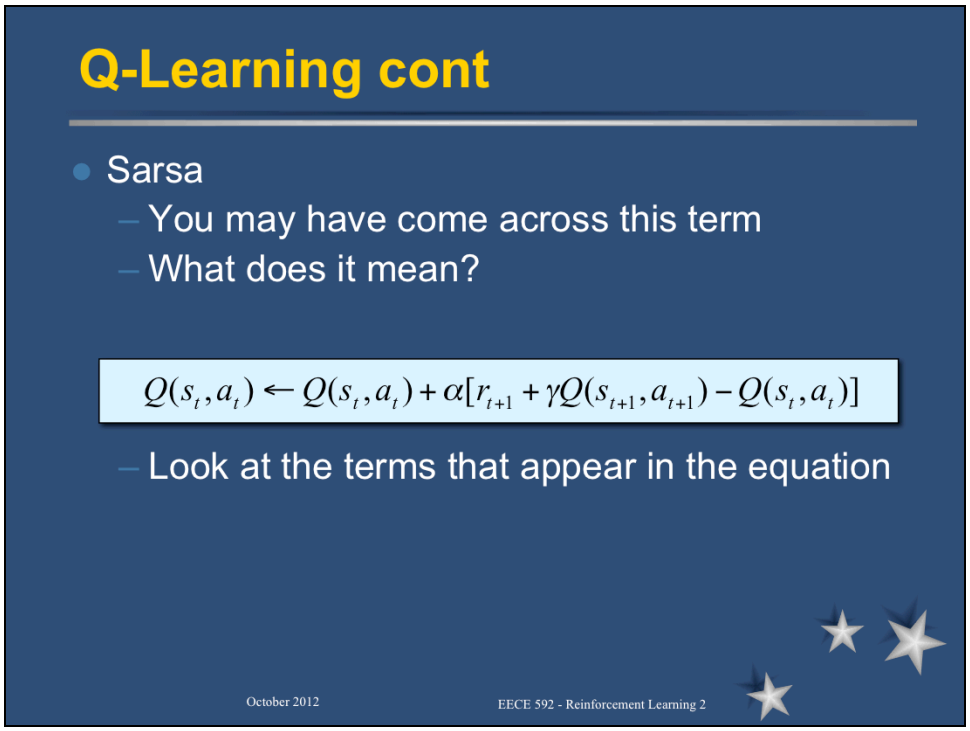


The above algorithm is from Figure 6.12, Sutton & Barto and represents what is known as an "off-policy" TD Control algorithm.

It is the algorithm that is recommended for us by Robocode. The basic idea then is that you determine (measure, scan whatever), the environment to determine the current state. Based upon this you determine the action to take. (I.e. look up in your Q-value table, all rows for this state and pick the one that gives the highest Q-value according to epsilon-greedy policy). You must also remember to backup this Q-value to the previous Q-value was selected.

Note that when making an exploratory move, according to Q-learning, the backup is still **as-if** you had taken the greedy move. This what makes Q-learning an off-policy algorithm. Its not always following the policy that it is learning.

In the literature you may also see similar algorithms referred to by the name *Sarsa*. You might be puzzled where this name comes from…….



Sarsa

The equation that defines TD(0) control algorithm uses the whole set of terms that apply in the transition from one state-action pair to the next state-action pair. This set looks something like this (s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}) and it is these that give rise to the name *Sarsa*. Now you know!

The SARSA algorithm, shown below, is subtly different from Q-Learning. Here it is (taken from Sutton & Barto,, Fig 6.9): Note the difference between the Q-Learning algorithm.

Initialize $Q(s,a)$

Repeat (for each episode)

  Initialize s

  Choose a from s using policy derived from Q (e.g., ε-greedy)

  Repeat (for each step of episode):

    Take action a, observe r, s'

    Choose a' from s' using policy derived from Q (e.g., ε-greedy)

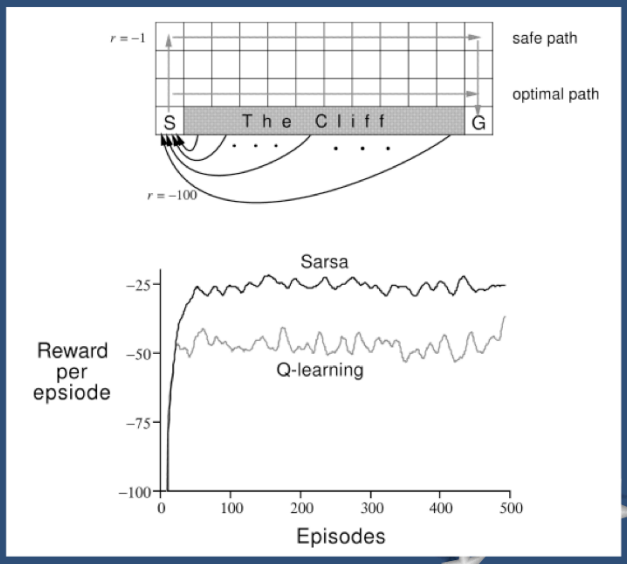
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$

    $s \leftarrow s'$; $a \leftarrow a'$

  until s is terminal

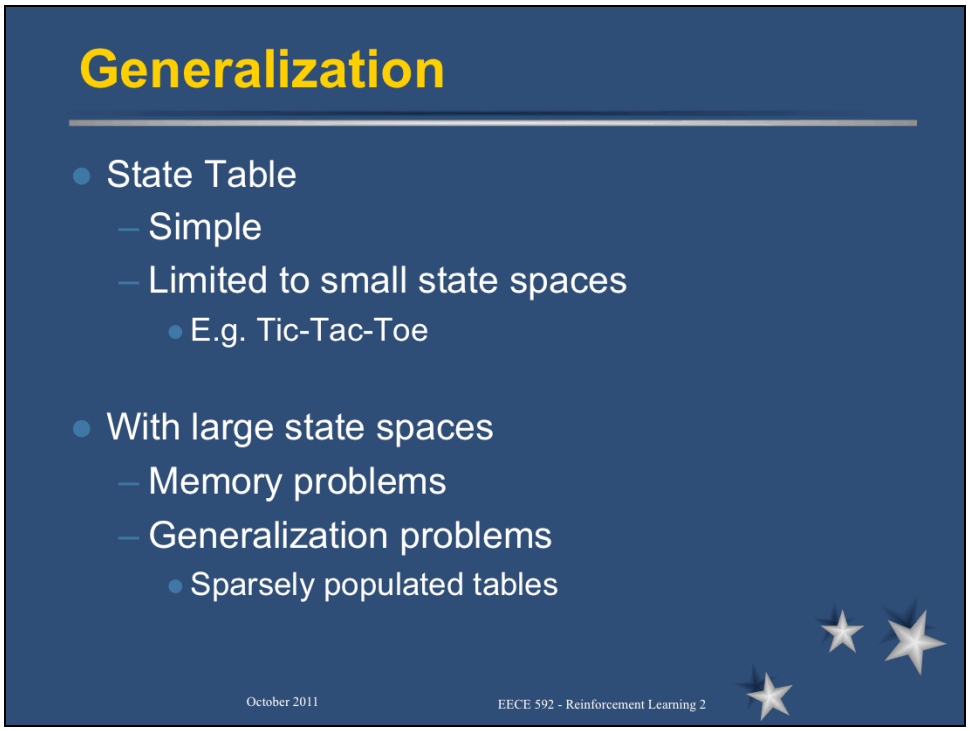
Q-learning vs Sarsa

- Q-Learning is *off-policy*
- Sarsa is *on-policy*



Sutton and Barto, 1998, "*Reinforcement Learning*. An Introduction". The MIT Press

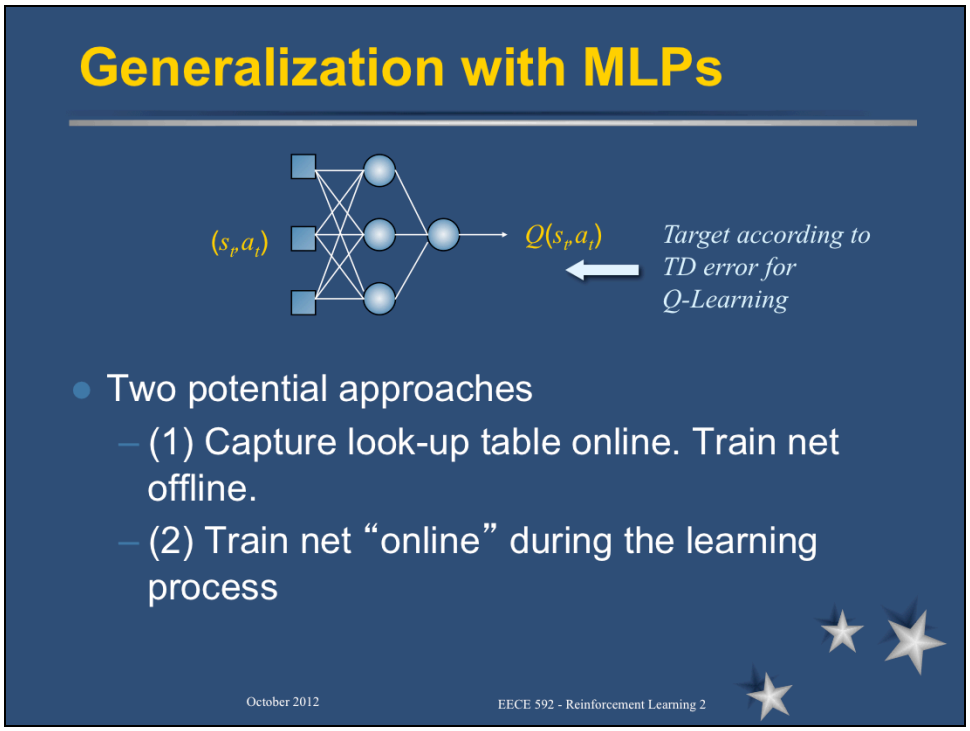
In this example, a reward of –1 is generated upon each transition and a reward of -100 for falling off the edge of the cliff. While Sarsa learns the safest policy, Q-learning finds the most optimal one – i.e. the best long term accumulated reward to reach the goal *G*, even though now and again it falls off the cliff!



Generalization

For instructional purposes, the implementation of RL is by far the easiest using look-up tables. Each value or Q-value is indexed by the states of the problem. Initially randomly initialized, as each state is visited, it is updated according to TD learning. In previous years, students were asked to implement Tic-Tac-Toe using a look-up table and in fact look-up tables have been used for Robocode too. However, as the number of states increases so does the size of the look up table and the number of computations to adequately populate it. In practice, with a world with many dimensions, the number of states will be large. This can be the case with Robocode. It is certainly the case with TD-Gammon (over 10^{20} states!). In such situations, not only is memory a concern, but more so generalization. A sparsely populated table is unlikely to be effective.

The answer is to generalize the value functions. There are many approaches including the application of non-linear approximators such as mutli-layer perceptrons. However, the area is still one of active research. In fact the application of neural nets, although promising and shown to be highly successful in one application (TD-Gammon) is regarded as a delicate art!



Generalization with MLPs

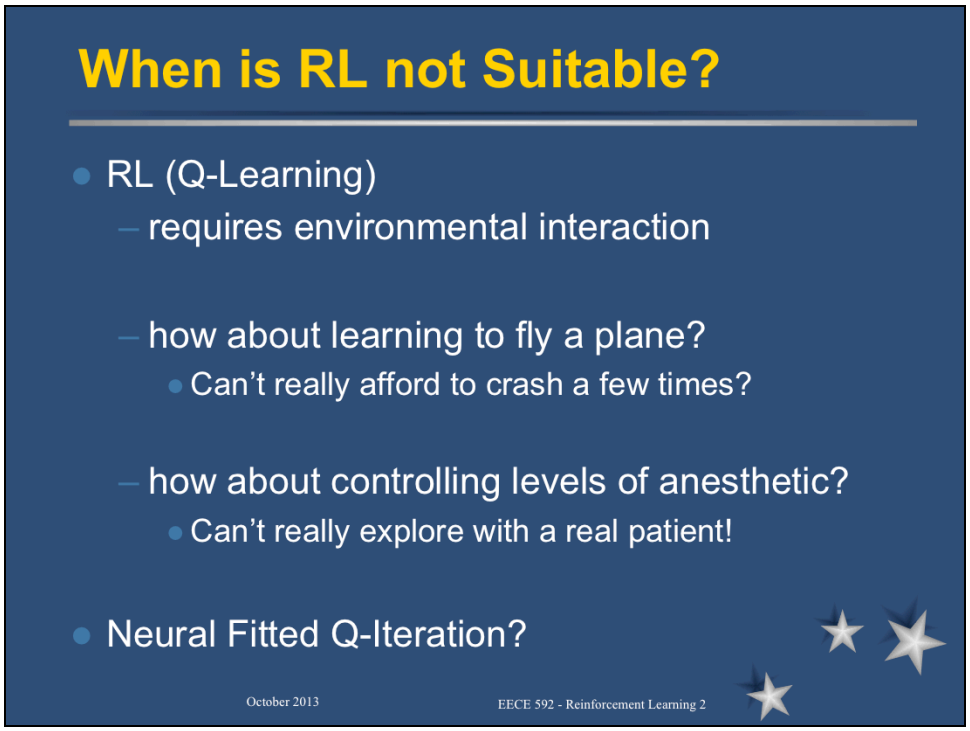
Two approaches to the practical assignment are suggested. Both require the use of a neural network (multi-layer perceptron) trained using the backpropagation algorithm. Note there are issues with both approaches suggested below. It is useful to read sections 8.1 and 11.1 of Sutton ad Barto.

(1) Online training

The Q-function is implemented as an MLP. As TD updates are backed up, the supervised values for the backpropagation training targets are generated according to TD as applied to the Q-function. (See earlier slides). Thus the MLP is undergoing training as it is being used.

(2) Offline training

The Q-function is implemented as a look-up table during training. As TD updates are backed up, the table is updated. Upon completion of RL training, the contents of the look-up table are then used to train a neural net. Once trained, the neural net is used to replaced the look-up table in the RL agent. We then observe improvements in the agents behaviour now using a generalized Q-function.

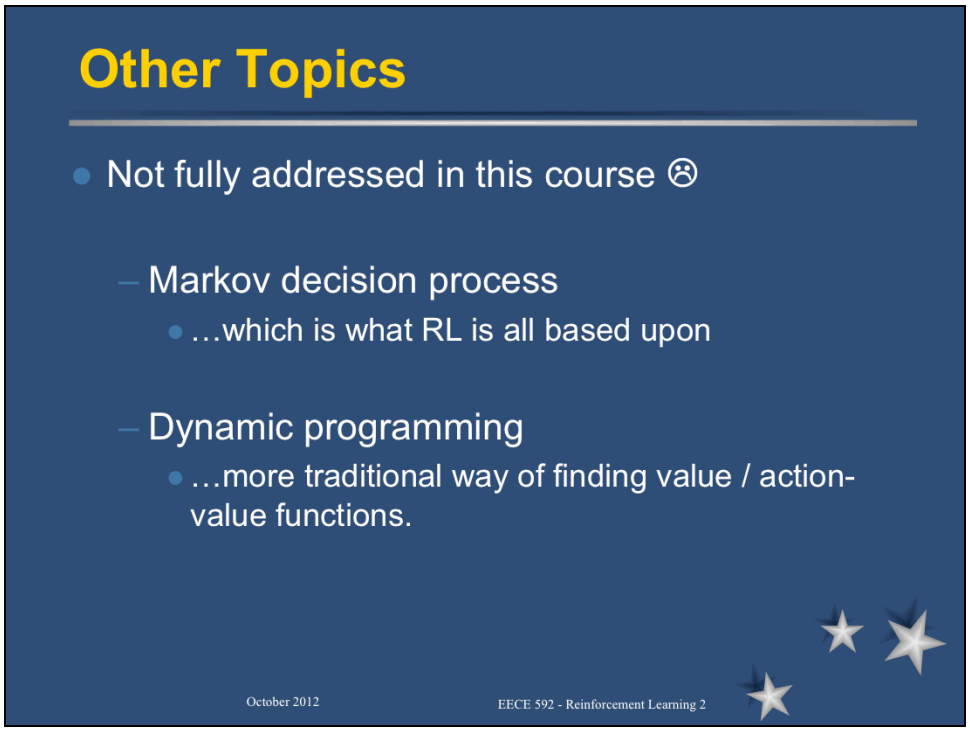


While RL and in particular temporal difference learning, seems generally suitable for learning many varieties of control problems, in order to do so, a learning agent requires interaction with its environment. In certain cases this might not be desirable or possible.

For example, learning to fly an airplane. You would not let an untrained agent take control of a real aircraft. Either some software would be necessary to restrict the agents actions to "safe" actions or possibly, to train it first using a simulator. (An approach which works well with real pilots).

How about applying RL in the health/medical realm? RL might be able to provide personalized care by learning to adjust medications on an individual basis. For example in anesthesiology, where the delivery of anesthetic is carefully governed based upon monitoring a patients vital signs. However again, the application of RL is unacceptable because initially, the agent will require exploration before it can learn to offer optimal dosing. Unlike training a pilot, in this case there is no simulator available for pre-training. With regards to this, one solution from the literature is to adopt an approach which attempts to off-line learn from collected data. Such an approach is described in [1].

Ref: [1] *Neural Fitted Q Iteration - First Experiences with a Data Efficient Neural Reinforcement Learning Method*, Martin Riedmiller, Neuroinformatics Group, University of Onsabru¨ck, 49078 Osnabru¨ck



Reinforcement learning is a broad topic covering many related topics. This course provides a glimpse, but unfortunately not a full or in-depth treatment. Some of the topics not addressed include but are not limited to are:

•Markov decision processes

•Monte carlo methods

•Dynamic programming

•Eligibility traces

Other topics cont.

- Monte Carlo methods
  - …No academic consensus for which is faster, TD or MC?
    - Although empirical results suggest TD.
- Eligibility traces
  - Creates a spectrum of algorithms spanning MC one on side and TD(0) on the other.