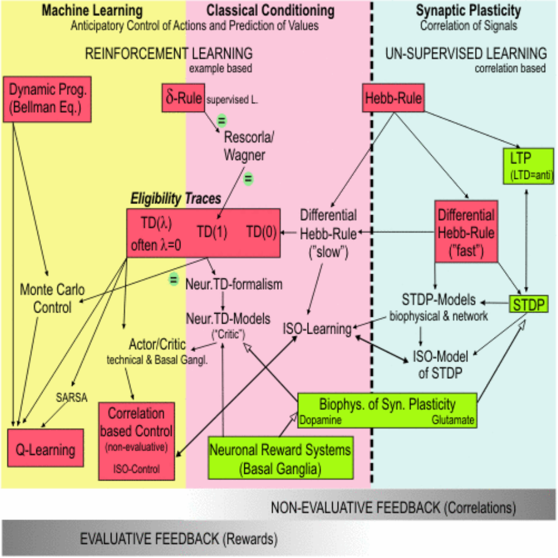
Reinforcement learning: day 2

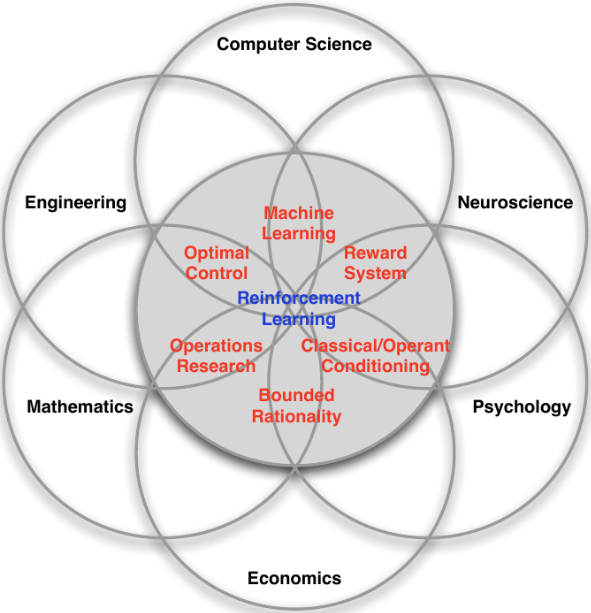
Variability in modular interchangeable RL methods



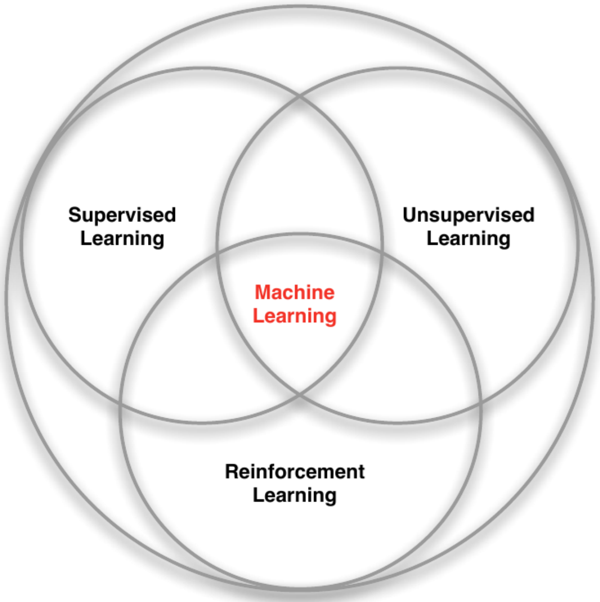
Applications

Fly stunt manoeuvres in a helicopter
Defeat the world champion at Backgammon
Manage an investment portfolio
Control a power station
Make a humanoid robot walk
Play many different Atari games better than humans
Industrial control
Production control
Automotive control
Autonomous vehicles control
Logistics
Telecommunication networks
Sensor networks
Finance
Games

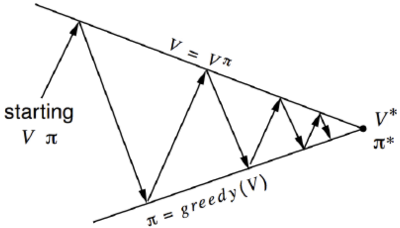
Interdisciplinary context

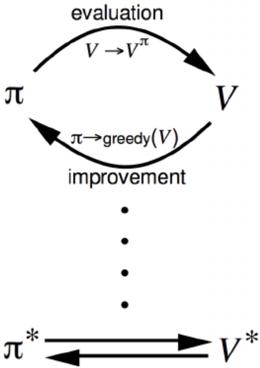


CS context


Supervised Learning
Unsupervised Learning
Machine Learning
Reinforcement Learning

Review: Policy iteration





Policy evaluation Estimate v_π

Any policy evaluation algorithm

Policy improvement Generate $\pi' \geq \pi$

Any policy improvement algorithm

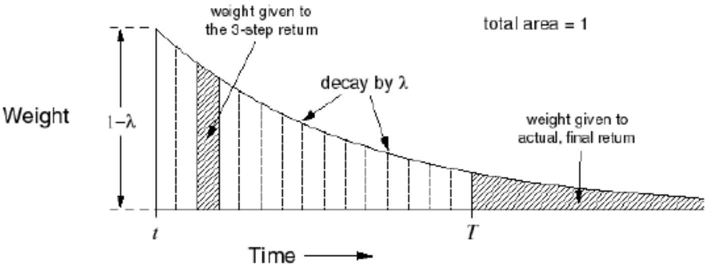
Review: Q-learning

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$;
  until S is terminal

TD(λ), SARSA(λ), etc

Recall: To maximally use all your data, it makes sense to update as many of your states as possible for each new data point.

TD(λ)



Change Q or V values for states visited farther back less.

SARSA(λ) keeps an eligibility trace

initialize $Q[S,A]$ arbitrarily

initialize $e[s,a]=0$ for all s,a

observe current state s

select action a using a policy based on Q

repeat forever:

carry out an action a

observe reward r and state s'

select action a' using a policy based on Q

$\delta \leftarrow r + \gamma Q[s',a'] - Q[s,a]$

$e[s,a] \leftarrow e[s,a]+1$

for all s'',a'':

$Q[s'',a''] \leftarrow Q[s'',a''] + \alpha\delta e[s'',a'']$

$e[s'',a''] \leftarrow \gamma\lambda e[s'',a'']$

$s \leftarrow s'$

$a \leftarrow a'$

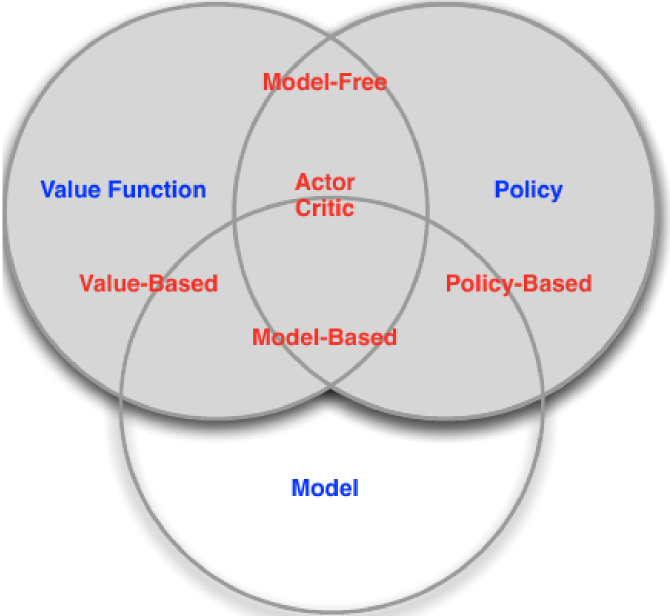
end-repeat

SARSA(λ) keeps an eligibility trace

Initialize $Q(s, a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Repeat (for each episode):
  $E(s, a) = 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
  Initialize S, A
  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)
    $\delta \leftarrow R + \gamma Q(S', A') - Q(S, A)$
    $E(S, A) \leftarrow E(S, A) + 1$
    For all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
      $Q(s, a) \leftarrow Q(s, a) + \alpha \delta E(s, a)$
      $E(s, a) \leftarrow \gamma \lambda E(s, a)$
    $S \leftarrow S'$; $A \leftarrow A'$
  until S is terminal

This will be important for your assignment!

RL context


Model-Free
Value Function
Actor
Critic
Policy
Value-Based
Policy-Based
Model-Based
Model

Limits of RL

What about very large state spaces and continuous problems?

Large scale RL

Reinforcement learning can be used to solve large problems, e.g.,
Backgammon: 10^{20} states
Computer Go: 10^{170} states
Helicopter: continuous state space

How can we scale up the model-free methods for prediction and control?

Reinforcement Learning with features

- **flat** or modular or hierarchical
- explicit states or **features** or individuals and relations
- static or finite stage or **indefinite stage or infinite stage**
- **fully observable** or partially observable
- deterministic or **stochastic** dynamics
- goals or **complex preferences**
- **single agent** or multiple agents
- knowledge is given or **knowledge is learned**
- **perfect rationality** or bounded rationality

Reinforcement Learning with Features

- Usually we don't want to reason in terms of states, but in terms of features.
- In state-based methods, information about one state cannot be used by similar states.
- If there are too many parameters to learn, it takes too long.
- **Idea:** Express the value function as a function of the features. Most typical is a linear function of the features.
- $Q(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \ldots + w_n f_n(s, a)$
- $V(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$

Example Features

- $F_1(s, a) = 1$ if a goes from state s into a monster location and is 0 otherwise.
- $F_2(s, a) = 1$ if a goes into a wall, is 0 otherwise.
- $F_3(s, a) = 1$ if a goes toward a prize.
- $F_4(s, a) = 1$ if the agent is damaged in state s and action a takes it toward the repair station.
- $F_5(s, a) = 1$ if the agent is damaged and action a goes into a monster location.
- $F_6(s, a) = 1$ if the agent is damaged.
- $F_7(s, a) = 1$ if the agent is not damaged.
- $F_8(s, a) = 1$ if the agent is damaged and there is a prize in direction a.
- $F_9(s, a) = 1$ if the agent is not damaged and there is a prize in direction a.

Example Features

- $F_{10}(s, a)$ is the distance from the left wall if there is a prize at location P_0, and is 0 otherwise.
- $F_{11}(s, a)$ has the value $4 - x$, where x is the horizontal position of state s if there is a prize at location P_0; otherwise is 0.
- $F_{12}(s, a)$ to $F_{29}(s, a)$ are like F_{10} and F_{11} for different combinations of the prize location and the distance from each of the four walls.
  For the case where the prize is at location P_0, the y-distance could take into account the wall.

Example function:

$Q(s, a) = 2.0 - 1.0 * F_1(s, a) - 0.4 * F_2(s, a) - 1.3 * F_3(s, a) - 0.5 * F_4(s, a) - 1.2 * F_5(s, a) - 1.6 * F_6(s, a) + 3.5 * F_7(s, a) \ldots$

Value Function Approximation

So far we have represented value function by a lookup table

- Every state s has an entry V (s)
- Or every state-action pair s, a has an entry Q(s, a)

Problem with large MDPs:

- There are too many states and/or actions to store in memory
- It is too slow to learn the value of each state individually

Solution for large MDPs:

- Estimate value function with function approximation
  $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$
  or $\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$
- Generalise from seen states to unseen states
- Update parameter w using MC or TD learning

Which Function Approximator?

There are many function approximators, e.g.

Linear combinations of features

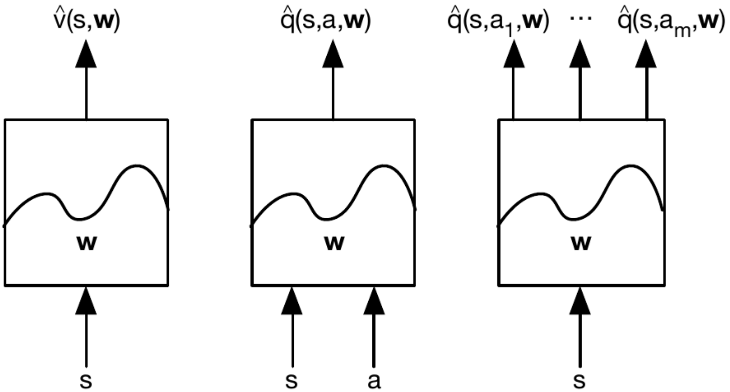
Neural network

Decision tree

Nearest neighbour

Fourier / wavelet bases

Function approximation architectures



1) v 2) q, action-in 3) q, action-out (deepmind)

Algorithm-Function interactions

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	TD(0)	✓	✓	✗
	TD(λ)	✓	✓	✗
Off-Policy	MC	✓	✓	✓
	TD(0)	✓	✗	✗
	TD(λ)	✓	✗	✗

Not all function approximation methods will converge with RL-algorithms. Non-linear methods like multi-layer networks are particularly problematic.

Algorithm-Function interactions

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	TD	✓	✓	✗
	Gradient TD	✓	✓	✓
Off-Policy	MC	✓	✓	✓
	TD	✓	✗	✗
	Gradient TD	✓	✓	✓

Some new TD methods are more robust

Algorithm-Function interactions

Algorithm	Table Lookup	Linear	Non-Linear
Monte-Carlo Control	✓	(✓)	✗
Sarsa	✓	(✓)	✗
Q-learning	✓	✗	✗
Gradient Q-learning	✓	✓	✗

(✓) = chatters around near-optimal value function

Still, the non-linear methods are hard.

Gradient descent

To find a (local) minimum of a real-valued function $f(x)$:

- assign an arbitrary value to x
- repeat

$$x \leftarrow x - \eta \frac{df}{dx}$$

where η is the step size

To find a local minimum of real-valued function $f(x_1, \ldots, x_n)$:

- assign arbitrary values to $x_1, \ldots, x_n$
- repeat:

for each x_i

$$x_i \leftarrow x_i - \eta \frac{\partial f}{\partial x_i}$$

Linear Regression

- A linear function of variables $x_1, \ldots, x_n$ is of the form

$$f^{\overline{w}}(x_1, \ldots, x_n) = w_0 + w_1 x_1 + \cdots + w_n x_n$$

  $\overline{w} = \langle w_0, w_1, \ldots, w_n \rangle$ are weights. (Let $x_0 = 1$).

- Given a set E of examples.
  Example e has input $x_i = e_i$ for each i and observed value, o_e:

$$Error_E(\overline{w}) = \sum_{e \in E} (o_e - f^{\overline{w}}(e_1, \ldots, e_n))^2$$

- Minimizing the error using gradient descent, each example should update w_i using:

$$w_i \leftarrow w_i - \eta \frac{\partial Error_E(\overline{w})}{\partial w_i}$$

Gradient Descent for Linear Regression

Given E: set of examples over n features
each example e has inputs $(e_1, \ldots, e_n)$ and output o_e:
Assign weights $\overline{w} = \langle w_0, \ldots, w_n \rangle$ arbitrarily
repeat:
For each example e in E:
let $\delta = o_e - f^{\overline{w}}(e_1, \ldots, e_n)$
For each weight w_i:
$w_i \leftarrow w_i + \eta \delta e_i$

Update rules for function approximation

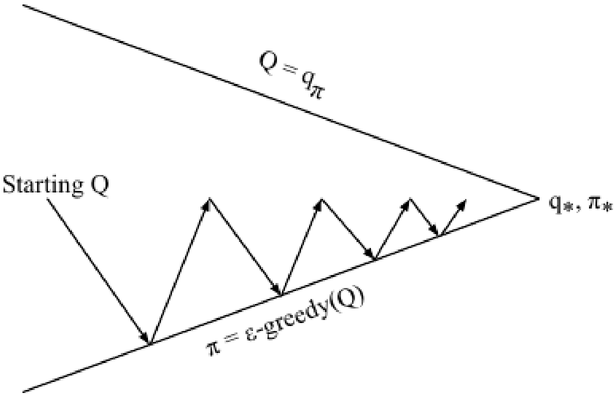
With V:
$\nabla_w \hat{v}(S, w) = x(S)$
Update = step-size (reward + prediction error x feature values)
$\Delta_w = \alpha(R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}))\nabla_w \hat{v}(S_t, \mathbf{w})$
$= \alpha \delta x(S)$

or with Q:

$\Delta_w = \alpha(R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{v}(S_t, A_t, \mathbf{w}))\nabla_w \hat{v}(S, A_t \mathbf{w})$

Note: **w** is a vector, which means this happens on a matrix or array all at once!

Approximate control



Every time-step:

Policy evaluation Sarsa, $Q \approx q_\pi$

Policy improvement ϵ-greedy policy improvement

SARSA with linear function approximation

- One step backup provides the examples that can be used in a linear regression.
- Suppose $F_1, \ldots, F_n$ are the features of the state and the action.
- So $Q_{\overline{w}}(s, a) = w_0 + w_1 F_1(s, a) + \cdots + w_n F_n(s, a)$
- An experience $\langle s, a, r, s', a' \rangle$ provides the "example":
  - **old predicted value:** $Q_{\overline{w}}(s, a)$
  - **new "observed" value:** $r + \gamma Q_{\overline{w}}(s', a')$

SARSA with linear function approximation

Given γ:discount factor; η:step size
Assign weights $\overline{w} = \langle w_0, \ldots, w_n \rangle$ arbitrarily
observe current state s
select action a
repeat forever:

carry out action a
observe reward r and state s'
select action a' (using a policy based on $Q_{\overline{w}}$)
let $\delta = r + \gamma Q_{\overline{w}}(s', a') - Q_{\overline{w}}(s, a)$
For $i = 0$ to n

$w_i \leftarrow w_i + \eta \delta F_i(s, a)$

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

Bonus points: If you can get this working in the assignment

SARSA-FA

controller *SARSA-FA*(F,γ,η)

    Inputs

        $F=\langle F_1,\dots,F_n\rangle$: a set of features

        $\gamma\in[0,1]$: discount factor

        $\eta>0$: step size for gradient descent

    Local

        weights $w=\langle w_0,\dots,w_n\rangle$, initialized arbitrarily

    observe current state s

    select action a

    repeat

        carry out action a

        observe reward r and state s'

        select action a' (using a policy based on Q_w)

        let $\delta= r+\gamma Q_w(s',a')-Q_w(s,a)$

        for $i=0$ **to** n **do**

            $w_i \leftarrow w_i + \eta\delta F_i(s,a)$

        $s \leftarrow s'$

        $a \leftarrow a'$

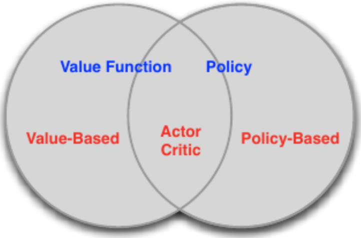
    until termination

Policy-gradient

Value Based
- Learn Value Function
- Implicit policy
- (e.g. ϵ-greedy)

Policy Based
- No Value Function
- Learn Policy

Actor-Critic
- Learn Value Function
- Learn Policy



Improve on evolutionary methods mentioned at beginning of last class

Applications

Games

Fly stunt manoeuvres in a helicopter
Defeat the world champion at Backgammon
Manage an investment portfolio
Control a power station
Make a humanoid robot walk
Play many different Atari games better than humans
Industrial control
Production control
Automotive control
Autonomous vehicles control
Logistics
Telecommunication networks
Sensor networks
Finance

Current game software (not all RL)

Program	Level of Play	Program to Achieve Level
Checkers	Perfect	*Chinook*
Chess	Superhuman	*Deep Blue*
Othello	Superhuman	*Logistello*
Backgammon	Superhuman	*TD-Gammon*
Scrabble	Superhuman	*Maven*
Go	Grandmaster	*MoGo*[1], *Crazy Stone*[2], *Zen*[3]
Poker[4]	Superhuman	*Polaris*

[1] 9×9
[2] 9×9 and 19×19
[3] 19×19
[4] Heads-up Limit Texas Hold'em

Current RL game software

Program	Level of Play	RL Program to Achieve Level
Checkers	Perfect	*Chinook*
Chess	International Master	*KnightCap / Meep*
Othello	Superhuman	*Logistello*
Backgammon	Superhuman	*TD-Gammon*
Scrabble	Superhuman	*Maven*
Go	Grandmaster	*MoGo*[1], *Crazy Stone*[2], *Zen*[3]
Poker[4]	Superhuman	*SmooCT*

[1] 9×9
[2] 9×9 and 19×19
[3] 19×19
[4] Heads-up Limit Texas Hold'em

Though this is a recent table, DeepMind and AlphaGo were more recently

RL details for game software

Program	Input features	Value Fn	RL	Training	Search
Chess *Meep*	Binary *Pieces, pawns, ...*	Linear	TreeStrap	Self-Play / Expert	$\alpha\beta$
Checkers *Chinook*	Binary *Pieces, ...*	Linear	TD leaf	Self-Play	$\alpha\beta$
Othello *Logistello*	Binary *Disc configs*	Linear	MC	Self-Play	$\alpha\beta$
Backgammon *TD Gammon*	Binary *Num checkers*	Neural network	TD(λ)	Self-Play	$\alpha\beta$ / MC
Go *MoGo*	Binary *Stone patterns*	Linear	TD	Self-Play	MCTS
Scrabble *Maven*	Binary *Letters on rack*	Linear	MC	Self-Play	MC search
Limit Hold'em *SmooCT*	Binary *Card abstraction*	Linear	MCTS	Self-Play	-

Deep learning

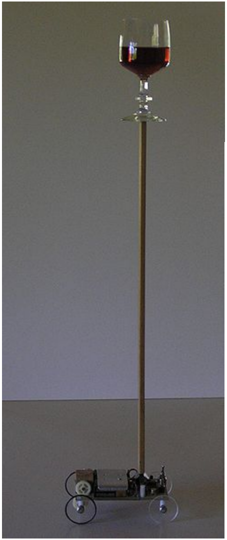
Fashion models and financial models are similar.
They bear a similar relationship to everyday world.
Like supermodels, financial models are idealized representations of the real world,
they are not real, they don't quite work the way that the real world works.
There is celebrity in both worlds.
In the end, there is the same inevitable disappointment"
- Satyajit Das

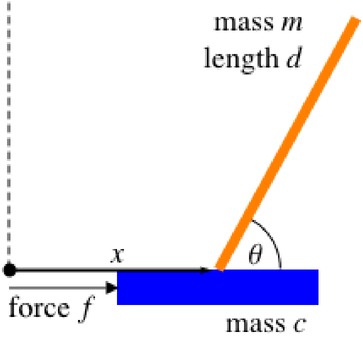
Some popular deep methods are **Q-learning with an action-out convolutional network as the feature approximator**

Applications: Real world practical

Games
Fly stunt manoeuvres in a helicopter
Defeat the world champion at Backgammon
Manage an investment portfolio
Control a power station
Make a humanoid robot **walk**
Play many different Atari games better than humans
Industrial **control**
Production **control**
Automotive **control**
Autonomous vehicles **control**
Logistics
Telecommunication networks
Sensor networks
Finance

Pole-cart / inverted pendulum





https://www.youtube.com/watch?v=Ep2lNMic_fk

Pole-cart / inverted pendulum

Your assignment: Keep the pole vertical at $\pi/2$