* RL in continous spaces.

↓

difference between discrete & continuous

↓

finite state & action space → discrete

↓

simplifies things (what we've worked on till now)

↓

allows us to employ lookup table for a table (action value function) for mapping between states & action, & dictionary for policy, or state value function

→ not restricted to a set of values. (range of values, rather)

optimization problem

↓

finding the best solution from all feasible solns. can be classified on the basis of types of values

discrete space is critical to some RL algorithms, for example value iteration

↓

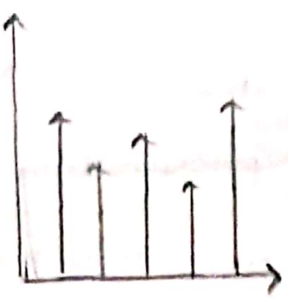
goes over all states and updates the value function values.

↓

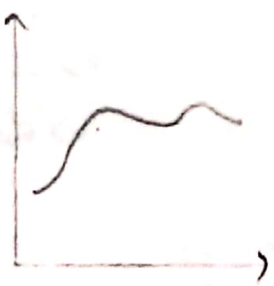
such an algorithm can't be implemented on a cont. state space or even on a very large finite space

↓

bar plot for discrete, needs to be thought of as density plot for continous.



discrete



continuos.

→ same notion extends to environments with multiple cont. parameters. (vector of n real values)

↓

cont.-space with >1 dimension

continous spaces are encountered in real ~~...~~ world like situation and not all environments be plainly represented in a grid like fashion.

↓

no cell 5, 3 for the robot to go to.

(so very popular in reinforcement learning)

↓

rather the problem in real-life will be changed to moving from its current position to about 2.5 m west & 1.8 m north from respective walls.

↓

agent also needs to keep track of its pose and where it is headed

(real-valued problem)

↓

Actions → continous → robotic arm that plays darts

↓

most actions to be taken in an physical environment are real in nature

↓

will have to change and play around with throwing angle, force, etc

↓

modification of algorithms & representation to ~~both~~ accomodate continous spaces.

↓

a) discretization

b) functionapproximation

Discretization

(continuous space into a discrete one)

↓

i) grid positions with discrete positions identified

↓

agent does not need to be in the center of these positions though.

↓

position will be rounded off to the grid cell representation (closest)

(3.1, 2.4)
↓
3, 2

↓

always a little incorrect, but for some environments, it may work very well without requiring any major modifications to already understood algorithms.

← actions can be discretized as well
↓
angles can be divided into whole degrees.

↓

obstacles introduced in the environment, that need to be avoided.

↓

with current discretization operation, we can mark off all the cells where the obstacles are present in any quantity. (even by a little)

↓ will cut off possible routes

↓ occupancy grid

↓

but if grid could be varied as per the obstacles, then this issue might not come up.

(non-uniform discretization)

Alternate approach : grid into smaller cells where required

↓

better than dividing entire state space into finer cells (more states, more computation reqd.)

↓

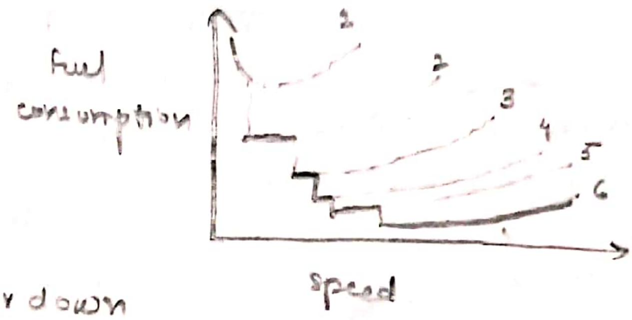
binary space partitioning , quad trees

↓

another example : automatic transmission

state : speed & gear

Reward : $\frac{1}{\text{fuel consumption}}$



Actions : switching up or down

↓

different speed ranges can be discretized , and the ranges may not be of same size

↓

another example of non-uniform discretization

* Tile coding : prior knowledge about state

↓

possible to design an appropriate discretization scheme

↓

like in automatic transmission, relation between fuel consumption & speed was known.

to function in a more arbitrary env

↓

requirement of a more generic approach.

↓

tile coding

↓

overlay multiple grids on top of the surface, and see which grids or tiles are currently activated, which can further be represented by a bit vector, and hence can then be discretized. (coarsely)

← at different offset to each other

↓

very efficient representation

↓

state value representation (instead of storing separate value for each state v of s, it is defined in terms of the bit vector for the state, and a weight for each tile in the state)

↓

tile coding algorithm updates these weights iteratively

↓

This ensures nearby locations that share tiles also share some component of state value effectively smoothing the learned state value function.

↓

Issues : a) tile sizes
b) offsets
c) no. of tilings

need to be decided ahead of time

(45)

adaptive tile coding

↓

does not req. another agent to tell approach for discretization

starts with large tiles, divides each tile into two whenever appropriate

↓

when to split? → heuristic?

↓

split when learning none from current representation

↓

i.e. value function isn't changing

↓

stop when an upper limit is reached on number of splits or max iterations or some other parameter

↓

which one to split? → one with greatest effect on the value function

↓

keep track of subtiles and their projected weights

↓

pick tile with greatest diff. in subtile weights.

↓

coarse coding : like tile coding but uses a sparser set of features to encode the state space

Drop many circles on the state space. At any location, see how many circles that point belongs to

↓

bit vector

↓

sparse coding representation of the state

↓

can also be extending to higher dimensions, where circles becomes spheres and hyperspheres

↓

using smaller circles, results in less generalization across the space, and vice versa

↓

more computation, more resolution

↓

not necessary to use circles, they can be stretched along one dimension to get higher resolution along that dimension, while shrinking some other ~~bypone~~ to reduce resolution

↓

resulting state representations as a bit vector

↓

extension to this idea is to see the distance from the centre of the circle, to measure how active that feature is.

↓

this measure can be made to fall smoothly using gaussian or bell shaped curve centred on the circle (Radial basis function)

↓

cont. state vector again → no. of features can be reduced

50 Function approximation

values nearby in a state space might be similar as smoothly changing. (for example, cliff walking)

↓

need to approximate the state value and action value function which might be vast over the entire space.

↓

being able to find it perfectly, is practically not feasible, hence approximation

↓

a method to do this, is to introduce a parameter vector w that shapes the function,

↓

our task becomes of tweaking this parameter vector until function can be appropriately approximated

↓

different possibilities here can be, that s can be made to its state value, (s, a) together can result in their q value, or s alone can be mapped to different q values for corresponding actions.

↓

useful for q learning

*) concept of state value function and its function approximation rather than value iteration

• approximating the state value function

$x(s)$

↓

i) firstly, we need to convert our state representation to a feature vector representation.

↓

also gives us more flexibility, since we donot need to operate on raw state values

↓

we have a feature vector $x(s)$, and parameter vector w, and we need a scalar value.

(dot product) ←

↓

linear combination

↓

linear function approximation → using a linear function to approximate the underlying value function, with a linear function.

↓

$$\hat{v}(s,\omega) = x(s)^T \cdot \omega.$$

(assuming random weight initialization, and computation of $\hat{v}$)

↓

tweaking ω to bring $\hat{v}$ to the true function value

↓

numerical optimization problem

↓

gradient descent

(52)

(1) value function

$$\hat{v}(s,w) = x(s)^T \cdot w \quad \ldots (i)$$

$$\nabla_w \hat{v}(s,w) = x(s)$$ (derivative of state value function w.r.t. w is the feature vector)

we are trying to optimize the difference between the true value function & the approximate value function.

↓

optimal value

current approximation

expected squared error

$$J(w) = E_\pi \left[(v_\pi(s) - x(s)^T w)^2 \right] \quad \ldots (ii)$$

expected difference (stochastic nature)

(2) minimize error

↓

error gradient

$$\nabla_w J(w) = -2\,(v_\pi(s) - x(s)^T \cdot w) \cdot x(s)$$

gradient of error w.r.t. weight

E has been removed as we focus on state s chosen stochastically

↓

sampling enough states

↓

coming close to the expected value

update rule : $\Delta w = -\alpha \frac{1}{2} \nabla_w J(w)$

↓

gradient from previous step.

$$\Delta w = \alpha \cdot (v_\pi(s) - x(s)^T \cdot w)\, x(s)$$

for each time step, we change the weights by a small step moving away from error, away from error direction, pointed by the feature vector ?

↓

for $\hat{q}(s,a,w)$: feature vector as a combination (transformation) of s, a.

↓

then some gradient descent method.

↓

computing all action values at once.

↓

can be thought of producing an action vector

$\hat{q}(s, a_1, w)$

$\vdots$

$\hat{q}(s, a_m, w)$

↓

its like feeding them one by one, but by employing matrices, this can be done at once

$\hat{q}(s,a,w)$

$$= \underbrace{(x_1(s,a), \dots x_n(s,a))}_{1\times n} \cdot \underbrace{\begin{pmatrix} w_{11} & \cdots & w_{1m} \\ \vdots & & \vdots \\ w_{n1} & \cdots & w_{nm} \end{pmatrix}}_{n\times m}$$

$$= \underbrace{(\hat{q}(s,a_1,w) \dots \hat{q}(s,a_m,w))}_{1\times m}$$

for discrete action space → select the action with max action value in the action vector

54

y continous → allows us to o/p multiple value at the same time

↓

issue : only linear relationships can be represented properly between i/ps & o/ps.

↓

for non-linearity → kernel function (dummy features)

Radial basis function

↓

feature transformation

(each element of the vector can be produced by a separate ~~vector~~ function, which can be non-linear)

* ~~N~~Non-linear approximation

↓

pass through artificial neural networks to get non linear approximations