

## Some terms

Model-free: not learning the transition and reward, only need to get observations from the real world or simulator

Prediction to control - prediction into a control policy

## Monte Carlo

$$\pi_{UCT}(n) = \operatorname{argmax}_a (Q(n, A) + c \sqrt{\frac{\log_2(N(n))}{N(n, a)}})$$

$$V(A) = \frac{\#A|A=1}{\#A + \#B}$$

## Generalized policy iteration

Take argmax get the policy

Run the policy and estimate a new Q function

Don't have to evaluate the policy fully

Do one step and update one data (Q function)

Update greedily wrt the Q function

By being greedy (get a new policy) and improve it

## Monte Carlo

Take many trajectories and average the trajectories

Temporal difference learning

Immediately update our prediction just after one step

But there is no return from the long trajectory

Current return of u

U is the estimate

Monte Carlo error: considers all the previous steps

vs.

TD error: how difference are my difference at this time step

Do gradient descent and minimize squared loss

Gradient is the direction of maximum increase

So we need to find the opposite to find the min which is why it is called gradient descent

## **n step TD**

better tradeoff between bias and variance

When  $n$  is  $\infty$ , it becomes Monte Carlo

Average can somewhat reduce variance (need less data to get good estimate, learn faster as it converges faster)

$TD(\lambda)$

$n$  step is weighted by  $\frac{\lambda}{n-1}$

0 forget the future take 1 step

1 close to 1 important for a long time

We are averaging the estimates (move expensive)

## **ADP**

Run  $v_i$  or  $\pi_i$

not trying to estimate the model only the values

need to have simulator of the real world to go on with RL

Build a model using function approximation

What is the transition function to use?

What reward to use

Successful has been parameterised Q function, supervised learning etc.

Model-free are easier to work with

We usually mix MC with TD in practice, less estimate and not much variance

## **TD**

Assumption is it is always markovian in real RL tasks in the environment

TD learning applies assumption to estimate

Initializer can be 0

A, 0, B, 0

Markovian assumption, if it sees A and B then it gives the reward of 0. Does not give the right values of the states because it makes an assumption that is not true of the environment

If it is true, then it will converge to correct value functions

Update equation

$$V^\pi(s) \leftarrow V^\pi(s) + \alpha(R(s) + \gamma V^\pi(s') - V^\pi(s))$$

$1-\alpha$  can be obtain by extracting -  $V^\pi(s)$  out and simplifying  
 $1 - \alpha$  can be obtain by extracting -  $V^\pi(s)$  out and simplifying

## SARSA

TD control

Uses GPI, on policy (improve the same policy)

target =  $R(s) + \gamma Q(s', a')$

SARSA will converge to optimal policy on tabular cases

MC on the windy gridworld

Converges slower but eventually reaches optimal

Have to use old policy until end of the trajectory

SARSA changes policy at every time step, may wander around at the start, but will continue learning

$\epsilon$  greedy: Take the action suggested by the Q function or random exploration

Improving the current policy with  $\epsilon$  greedy, when present does not converging to the optimal policy

trying to improve the policy

environment might change, world isn't quite a MDP, it can be robust, model may change over time

Might still fall off the cliff

Predictive problem learning (passive TD) has some restrictions

Update only at next state

## Q-learning

Only have the q function (acts as policy on its own)

Not in SARSA:  $\gamma \max_a$

best action at the next action of Q function

E.g currently have a deployed policy, at wild change the policy with SARSA, train the target policy

Behavior policy can be some other policy that is running, generate data to train the policy

If stop learning then Q-learning will outperform Sarsa

Similar concept to  $\epsilon$  greedy

Let's go to part of the space that is exploit less

Issues when state space is large

Update policy then take argmax - given the current policy from the Q function, estimate a new Q function and construct a new policy by taking argmax of Q function

Q learning can learn off policy (doesn't follow the policy given, uses different policy)

Take the max over the next state

Minimizing the difference between the target and the current estimate

If the count is small give bonus, if we try many times reduce bonus

Why don't u update all the Q at each state?

Update is only done for all the state-action

Not the final Q-table, it will interact with the environment and update the table

What if the transitions are non-deterministic?

probability to move to a state

Even it is probability, it will move to that state, no longer a probability so take Q value of that state, expected Q learning

## Function approximation

Number of states are huge, huge state space

Approximate (estimate) Q value with function approximator

Figure out what are the important features and extract out to define each state E.g. for a chess, queen is worth 5 points, pawn worth, knight worth

RL will learn the parameter vector  $\Theta$

Allows generalisation observations so that it can work on entire state space

Small fraction of the state

Applications: Go, Deep neural network, OpenAI5

MCT Search on simulator, for online search value function and combine with MCTS

Do learning with function approximation

Runs in the form of ridge regression

## Online learning

One example, learn an example and update the weight vector

Update  $\theta$  to improve estimate, step in the direction of the negative gradient

$\hat{U}(s)$  function approximator, estimate from the function

target  $u_j(s)$  At every time step it is the sum of the reward to the future, measured from the environment

It could be the q function

Semi gradient - don't include in the computation

gradient wrt the parameter  $\Theta$

we treat  $\gamma \hat{U}(s')$  as a fixed target

When we move on-policy we lose most of our policy, have to do lots of tuning

e.g. doesn't converge

Deep Q-learning

Techniques to do better: Experience replay

large buffer  $D$ , old sample does not mix with the recent one

$(s_t, a_t, r_{t+1}, s_{t+1})$  max over  $a$  without the  $k$

learn and freeze it and use as target  $\theta^-$

$\theta^-$  Q function from current -  $c$  steps

High profile achievement as one architecture can work across 50 atari games

Set a new target and change every  $c$  steps

Extract with a linear function of features

E.g. position of the ball

Need two frames to determine the direction we are going

provably good - how long would it take to go to epsilon

## Policy search

Mapping from state to action

Drop the assumption that policy is of  $Q(s, a)$

Deep learning is continuous

Max function is not differentiable everywhere

max function isn't linear

Gradient descent can differentiate that  
 Actions are discrete can't find gradient wrt a

## POMDP

Approximate sometimes doesn't work but usually work well  
 P space hard (as hard as any problem that can be solved with polynomial space, consist of NP)  
 If there is  $P(b', a)$  and  $P(b)$ , we have a MDP  
 Function from belief to action through an observation  $b - o - a$   
 $b$  input vector  $[b(1), b(2), b(3)]$ ,  $\alpha$  is the weights or coefficient of the linear function (alpha vectors) then it becomes a linear function  $b(1)\alpha(1) + \dots + b(n)\alpha(n)$   
 $\alpha(1)$  is the expected return at the state

Optimal policy to choose conditional plan  
 $U(b) = \max_p b \cdot \alpha_p$

Do induction on the depth of the tree

wrt to state action, then if u take an action then u go to a state  
 Zero step plan only take into regard the state, so are initialised with rewards at the state  
 Each 2-step plan can draw a straight line  
 Each region has a corresponding value function

Converge into a convex function

Construct a depth d plan

Transition =  $\sum_{s' P(s'|s,a)}$

Average over observations (with probability of receiving o) =  $\sum_e P(e|s')$

Probability to use alpha vector at  $s' = \alpha_{p,e}(s')$

$\alpha_p(s) = R(s) + \gamma(\sum_s P(s'|s,a)$

Possible subtrees that can put in position d - 1 action

1. Use initialised values to compute alpha vectors
2. Remove dominated vectors for depth 1 (Largest difference between belief to small enough)

3. Use existing alpha vectors to compute alpha vectors at depth 2 and so on

Know the belief instead of being perfect of knowing the state

Be clever on which subset to pick

Intractable - no of state variables grow Also never reach same belief twice, have to enumerate through all the states, a lot of multiplications and integrations

Belief tracking is expensive, generating beliefs is continuous

Instead use history, sample some states for approximating beliefs

Tree search until a certain depth, build the tree from simulation

Particle filtering

Have a simulator, so represent a set of states

If initial belief is bad, then it might affect the approximation

Deterministic will take longer time to compute

## **Partial Observability**

Fog of war in DoTa

single network that controls all 5 players

each agent runs independently

Break into hierarchical operation

RNN to learn what is good to remember and propagate (hidden unit) to memory

Each player has own policy (POMDP)

For model based how is it well defined?

Model of the opponent becomes an environment

For a 2 player, will converge to an optimal structure

Value iteration for MDP, what u observe is different from what the enemy observe

Does not work with multi-agent, belief is no longer meaningful but policy still holds

Optimal collection of trees(policies), each agent to have a tree

Each agent to not do its own observation

Reward shaping might give incorrect reward, go around collecting rewards instead of what u want

It helps bring consequences that are in the distant future closer to the present, so it's easier for an agent to learn.

Where win and loss are very far

Not so much for chess, pawn and queen worth is defined using function approximator

Adding a constant does not change the optimal policy