RL - Week - 7

L1 Intro to Func^n Approxima^n

- q func^n is stored as a lookup table.

s	$q(s, a_1)$	$q(s, a_2)$
s_1	—	—
s_2	—	—
s_3	—	—

- if they are large no. of states → not efficient, data sparsity, contin. state / ac^n spaces, we can't generalize it.
- ∴, we can use a parameterized represenṭa^n → value func^n, policies, models, etc. → $f(x) = \alpha x + \beta^2 x + \gamma$ ← parameters
- a NN → fully parameterized object

L2 Linear Func^n Approxima^n

Least sq. $Q(s_t, a_t) = f(s_t, a_t; w_t)$ ← params

$$w_{t+1} = w_t - \frac{1}{2}\alpha \nabla_{w_t} [q_*(s_t, a_t) - Q(s_t, a_t)]^2$$

but the problem, I don't the target itself.

maybe use the TD target.

$$w_{t+1} = w_t - \frac{1}{2}\alpha \nabla w_t \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]^2$$

TD error, δ_t

$f(s_{t+1}, a_t; w_t)$ $f(s_t, a_t; w_t)$

- GD → 1st order iterative optimizaⁿ algo. to find the local minima of a diff. funcⁿ. It is in the direcⁿ of steepest descent.

$$f(x_0 + h) \approx f(x_0) + h \cdot f'(x_0)$$

$$L(\theta + \alpha \Delta\theta) \approx L(\theta) + \alpha \Delta\theta^T \nabla_\theta L(\theta)$$

- Gradient ascent — in the direcⁿ of steepest ascent.
- so we use semi-gradient method (we ignore the gradient of TD target.

$$Q(s_t, a_t) = \phi^T(s_t, a_t) \times w_t$$

$$\delta_t = r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)$$

$$\boxed{w_{t+1} = w_t + \alpha \delta_t \phi(s_t, a_t)}$$

taking gradient w.r.t only predicⁿ valu

L3 State & Action Representation

$\phi(s) = \langle x_1, y_3, 1000 \rangle$ $N = 1000$, $S = 0100$, $W = 0010$

↳ kind of a 6-dimensional

$Q(s, W) = w_1 x_1 + w_2 y_2 + w_3 z + w_4$

$\phi(s, N) = \langle x_3, y_3, x_3^2, y_3^2, x_3 y_3, 0 \rangle$

↳ this is still a linear approxima^n. even though square terms are, we do not have to solve for it.

$\phi_1(s, a) = \text{is}(\text{block\_on\_table}) = \begin{cases} 1 & T \\ 0 & F \end{cases}$

↳ these are features of state-action value pairs.

L4 More on Representations

Lookup table

	a_1	a_2
hot	$q(\text{hot}, \text{ad})$	$\neg\text{ad}$
cold	$q(\text{cold}, \text{res})$	$\neg\text{res}$

using linear func^n approximator

$\phi_1\ \phi_2\ \phi_3\ \phi_4$

$\phi(\text{hot}, \text{ad}) = \langle 1000 \rangle$

$\phi(\text{hot}, \neg\text{ad}) = \langle 0100 \rangle$

$\vdots$

$\phi(\text{cold}, \neg\text{res}) = \langle 0001 \rangle$

$$q(s,a) = w_1 \phi_1(s,a) + w_2 \phi_2(s,a) + w_3 \phi_3(s,a) + w_4 \phi_4(s,a)$$

$q(\text{hot}, \text{ad}) = w_1 \cdot (1) + 0 + 0 + 0 = w_1$

$q(\text{hot}, \neg\text{ad}) = w_2$

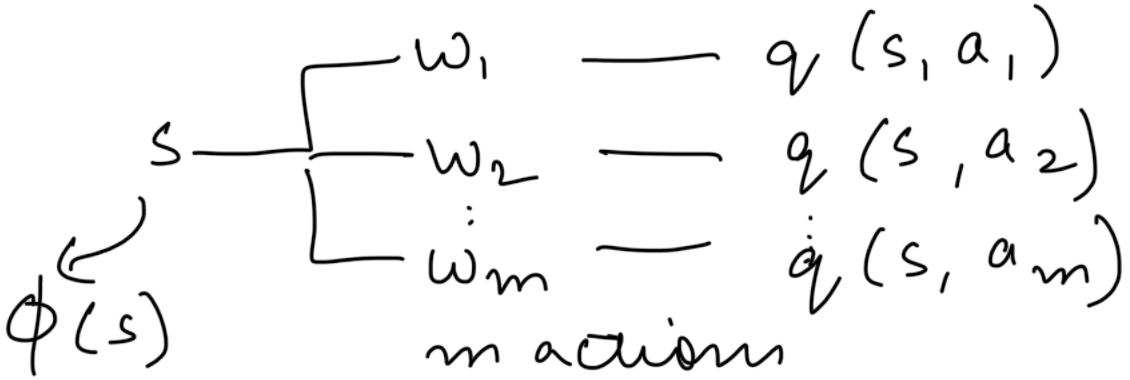
$q(\text{cold}, \text{res}) = w_3$

$q(\text{cold}, \neg\text{res}) = w_4$

another way,

$\phi^T(s,a)\, w$

or

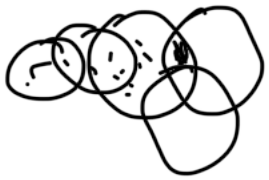


↳ these ϕ_s will be slowly learned in the process.

- we have to exploit the possibility the q values of near states would not Δ much.

- Naive →
  1. divide the grid 10 x 10.
  2. abrupt Δ in q value at the boundary.

- Coarse Coding → avoid abrupt Δs. Consider a circular world. Smoothes the q values during the transition.



problem → no uniform no. of "ON" bits representing states.

- Tile Coding → like coarse but in grid world, more systematic.

No. of ON bits = no. of tiles used.
↳ and each one of tile represents a feature.

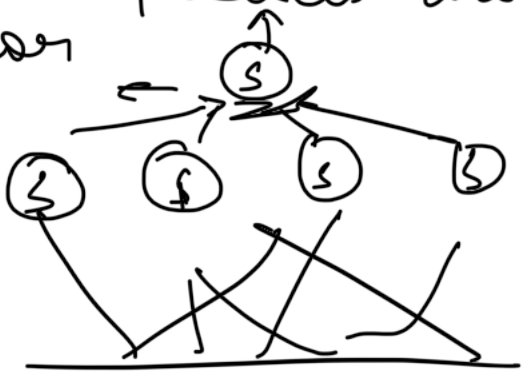
↳ leads to CMAC (cerebral model articulation controller)
powerful classifier, linear, no complex forward pass, in backward pass just update what is required.

- In SGD → the true gradient of loss func^n is approximated by the gradient at a single eg.
- In usually, we do mini-batch GD, use small egs. This allows for efficient

computa^n & smooth convergence (approx. of gradient, ∴, still SGD)!

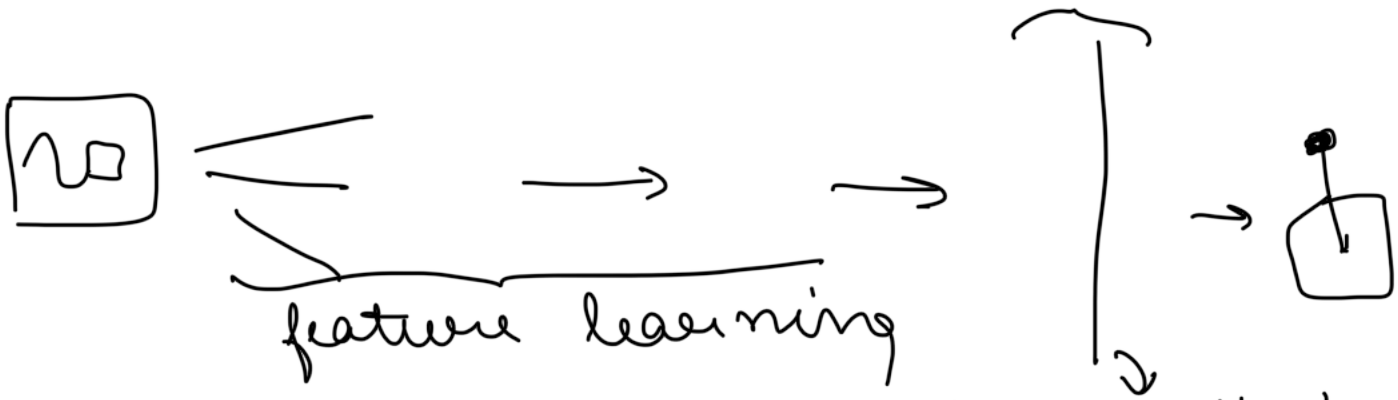
L5 DQN (Deep Q-Networks)

- linear func^{ns} are restrictive. They can only model linear func^n / extrapola^n.
- non linear approx. can model complex funcs, very powerful, compute & data (disadv) hungry. Features are learnt on the fly and not like tile coding.
- It generalizes well on unseen cases.
- Eg. TD-Gammon. Classical AI players can't play that good. Self play (1.4 million games)



- what about alaai games? learnt from scratch, transfer, ϕ_s were also learnt from scratch. ϕ was initialized randomly

- Deep Q-Learning Network (DQN)



- finally we train using the TD error that also calculates the ϕs.

- Q-Network learning

$$w_{t+1} = w_t - \frac{1}{2}\alpha \nabla w_t \left[r_{t+1} + \gamma \max_a q(s_{t+1}, a) - q(s_t, a_t) \right]^2$$

(target)

  - target is non-stationary.
  - correlation btw. samples.
  - divergence issue → current network is used to decide its own target.

- How to address this? → replay memory, freeze target network.
- Replay memory → fixes correlation.
  ↳ build a data-set from the

agent's experience.

$$s_1, a_1, r_2, s_2$$

$$\vdots$$

$$s_t, a_t, r_{t+1}, s_{t+1}$$

- frozen target network – non-stationary
  ↳ sample experience from dataset.
  slow down the process. w⁻ – frozen for some time period.

$$\left[\left\{r_{t+1} + \gamma \max_a q(s_{t+1}, a; w^-)\right\} - q(s_t, a_t; w)\right]$$

- except for few games, DQNs do well than humans.
- Architecture of DQN
  1. set of conv. layers which act as feature extractors
  2. features are passed through FC layers.
  3. output layer has $|A|$ no. of nodes.

uses now huber loss not reg. least sq. loss.

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \leq \delta \\ \delta\left(|a| - \frac{1}{2}\delta\right) & \text{otherwise} \end{cases}$$

L6 Advanced Value-Based Methods

- Double DQN (DDQN)
  1. add replay memory & target network.
  2. make Q network a NN
  3. add 2 more networks. Then we can use the target network to estimate the value while the online network is used for selecting the action..

$$Y_t^{DDQN} = R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\text{argmax}}, a; Q_t^-)$$

Q_t^- ← target network params

- dueling DQN → sometimes its not necessary to estimate the value of each action choices in many states.
  - ↳ Architecture —
    1. Value network attends on both horizon of track as well as scene.
    2. the attention network concentrate only when there are inbound cars.

- so we estimate the Q value using both $A(s,a)$ & $V(s)$.
- Advantage → calculates the benefit of taking an action.
- What is the maximum value of advantage func^n?

$$Q(s,a) = V(s) + A(s,a)$$

$Q(s,a)$ → to update all other things

$V(s)$ → value func^n

$A(s,a)$ → advantage func^n

(for most of the func^{ns} A will be 0, only V has to learnt.)

- Aggrega^n layer → a sep. layer is req. to combine $V(s)$ & $A(s,a)$

Issue → it is impossible to recover $V(s)$ & $A(s,a)$ from $Q(s,a)$

Overcome → avg. advantage is calculated!

$$Q(s,a;\theta,a,b) = V(s;\theta,b) + \left(A(s,a;\theta,\alpha) - \frac{1}{A}\sum_{a'} A(s,a';\theta,\alpha)\right)$$

Thus, we can now apply Q-learning techniques such as DDQN & PER with the dueling architecture.

- SARSA

$Q(s_t, a_t)$ ------ $\gamma Q(s_{t+1}, a_{t+1})$

a_{t+1} introduces variance which slows convergence.

⇓

Expected SARSA

We are approximating by just 1 sample.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \left[R_{t+1} + \gamma \sum_a \pi(a|s_{t+1}) Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

it converges faster.

- Expected SARSA is ON Policy. The expectation is related to π! We are looking at the distribution of each action.