Lecture 6 → RL. Value function approximations

→ scaling RL to real-world problems.

→ Incremental methods → take some functn approximator (NN) and when see some new data, online → update your value functn.

→ Batch methods

→ look at set of data / history of data.

Large Scale RL

→ Backgammon → 10^{20} states

→ go → 10^{170} states. → Can't build a table → not practical

→ want methods to work inspite of large # states.

eg don't want separate values for some position X, and another position at 1mm to X. → ideally value functn should capture / understand that.

→ This is for value functions

→ Scale up for prediction & control.

Value function Approximators.

$q(s,a)$ → used in the model free regime.

(X) Too many s/a to store in memory

(X) Slow to learn the state's value individually.

parametric functn approximator → $\hat{v}(s, w) \simeq v_\pi(s)$ → parameter. gives estimate of $v_\pi(s)$ for any s.

$\hat{q}(s, a, w) \simeq q(s, a)$ → estimate for any state / action pair

(+) reduce memory

(+) Generalize. → to states that we haven't seen.

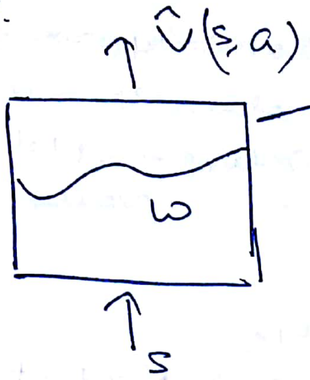
eg. w → weights of NN

→ update using earlier algos → Monte Carlo, TD learning

Meaning of function approx with a value function

Types

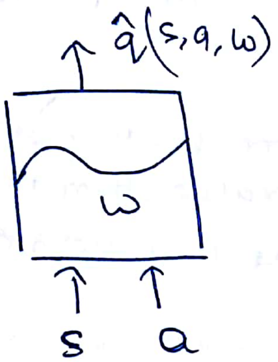
(1)



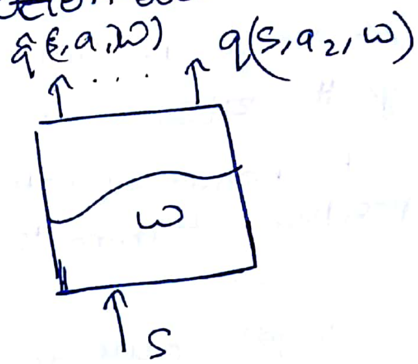
black box function approximator with parameter → w.

→ value for a state s.

(2) i) Action-in



ii) Action-out



Using the tunable parameter → w, we get a universal function approximator for all states /(or/and) action

Many FAs:

→ NN, decision-tree, etc.

How to choose?

→ Now focus on differentiable FAs → so that when we change w, we know how $V(s,w)$ affected.

- Linear combination of features
- Neural Nets.

Challenge compared to supervised Learning

↳ data is non-iid like the $\hat{v}(s,w)$ → approximating $v_\pi(s)$ → but the π here is constantly being improved

Gradient Descent

use to find the minimum of $J(\omega)$

$J(\omega) \rightarrow$ differentiable $\rightarrow$ adjust parameter in the downhill direction

$$\nabla_\omega J(\omega) = \begin{bmatrix} \frac{\partial J(\omega)}{\partial \omega_1} \\ \vdots \\ \frac{\partial J(\omega)}{\partial \omega_n} \end{bmatrix}$$

$\rightarrow \Delta \omega = -\frac{1}{2}\alpha \nabla_\omega J(\omega)$

If $\rightarrow$ by magic, we had actual $V_\pi(s)$ that should have been given $\rightarrow$ simply do the mean square error as the cost func^n and find the minimum.

use SGD $\rightarrow$ as a replacement for (Expectation) (substitute)

Feature vector

$$x(s) = \begin{bmatrix} x_1(s) \\ \vdots \\ x_n(s) \end{bmatrix}$$

$\rightarrow$ everything ~~that~~ says something about the state. (one thing)

Linear VFA $\rightarrow$ $\hat{v}(s,\omega) = x(s)^T \omega = \sum_{j=1}^{n} x_j(s)\omega_j$

(+) mean square error is quadratic and hence, SGD converges to ~~optimal~~ global optimum

$$\Delta \omega = \alpha (V_\pi(s) - \hat{v}(s,\omega)) x(s)$$

$\rightarrow$ adjust only where features are on/active

update = step size $\times$ error $\times$ feature value

Table lookup

$\rightarrow$ essentially $\rightarrow$ special case of linear VFA

$$x^{table}(s) = \begin{bmatrix} I(s_1 = s) \\ \vdots \\ I(s = s_n) \end{bmatrix}$$

$\rightarrow$ only 1 is non-zero.

$\hat{v}(s,\omega) = x^{table}(s)\,\omega \rightarrow$ gives the value for that specific state.

Scanned by CamScanner

But we don't have $\underline{\underline{V_\pi}}$.

Use MC / TD to give a target ~~for~~ our VFA.

→ For MC, target is G_t → unbiased estimator for V_π

$$\Delta w = \alpha (G_t - \hat{V}(S, w)) \nabla_w \hat{V}(S, w).$$

→ For TD(λ) → target is G_t^λ

eg. for TD(0) → $R_{t+1} + \hat{V}(S_{t+1}, w)$

G_t → unbiased, noisy sample of $V_\pi(S)$.

→ we have training data like SL → $\langle S_1, G_1 \rangle \dots \langle S_T, G_T \rangle$

→ Linear MC policy evaluation

$$\Delta w = \alpha (G_t - \hat{V}(S_t, w))\, x(S_t).$$

→ converges to local optimum (eg. with non linear VFA)

TD target → biased sample of true $V_\pi(S)$.

→ Still SL with data → $\langle S_1, R_2 + \gamma \hat{V}(S_2, w) \rangle, \dots$

$\dots \langle S_{T-1}, R_T \rangle$.

$$\Delta w = \alpha \underbrace{(\gamma \hat{V}(s', w) + R - \hat{V}(s, w))} \nabla_w \hat{V}(s, w)$$

$$= \alpha\, \delta\, x(s). \text{ (linear case)}.$$

[All these → incremental online updates]

Linear TD(0) $\xrightarrow{\text{close to}}$ global optimum.

TD(λ) → λ-return G_t^λ

① $\langle S_1, G_1^\lambda \rangle \dots \langle S_{T-1}, G_{T-1}^\lambda \rangle$

② $\Delta w = \alpha (G_t^\lambda - \hat{V}(S_t, w))\, x(S_t)$.

③ Here, Eligibility Trace → ET on the parameters, not the states

$$\delta_t = R_{t+1} + \gamma \hat{V}(S_{t+1}, w) - \hat{V}(S_t, w)$$

$$E_t = \lambda\gamma E_{t-1} + x(S_t)$$

Doubt → gradient of $\hat{v}(s', w)$ not included in optimizing?

$$\Delta w = \alpha (R + \hat{v}(s', w) - \hat{v}(s, w)) \nabla_w \hat{v}(s, w)$$

why not the gradient used here

But, it sort of means pulling both ends of the spring together close →

& its shown to fail (But other modifications are there → residual gradient).

we need to ground stuff / something & we always do that in the future.

Control with VFA

- Approximate policy evaluation → $\hat{q}(s, a, w) \approx q_\pi$
- Policy improvement - ϵ- greedy.

→ update on every step → always use the most fresh experience.

→ In practice, this goes to very close of the optimal π_*.

Action-VFA → same steps → just using Q instead of V.

Feature vector → $x(S, A) = \begin{bmatrix} x_1(S, A) \\ \vdots \\ x_n(S, A) \end{bmatrix}$

using the update eqn, we either pull up / down the functn approximator → to what we actually see → coarse coding

too many examples

Bootstrap → helps. TD(λ) > TD(0). MC → fails badly → too variance.

TD → does not guarantee that TD would converge.

(Follow Table in slide). to get where TD can be applied

Gradient TD → really follows the true gradient of the projected Bellmann error.

fixes the problems with the TD stability

Why TD unstable?

→ TD does not follow the gradient of any objective func and hence, can diverge.

⇛ Control → problematic

	Linear	Non-linear (VFA)
Q-learning	✗	✗
Gradient Q-learning	✓	✗.

→ Can keep chattering around the optimal policy.

Batch RL

→ Problem with GD → throws away experience after one use. → not sample efficient.

→ Find the best-fitting value to the entire batch.

Least squares Prediction

→ minimize error over batch/ entire dataset.

$D = \{ \langle s_1, V_1^{\pi} \rangle, \ldots \langle s_T, V_T^{\pi} \rangle \}$ ↳ true value.

Find best LS fit

$$LS(w) = \sum_{t=1}^{T} \left[V_t^{\pi} - \hat{v}(s_t, w) \right]^2.$$

$$= E_D \left[(V^{\pi} - \hat{v}(s, w))^2 \right]$$

Easy way to find the LS soln. → Experience Replay (ER)

- store the entire experience D
- sample from experience → $\langle s, V^{\pi} \rangle \sim D$.
- Apply SGD to get that target. ↘ global minimum obtained is the LS soln.
- $w^{\pi} = \arg\min_{w} LS(w)$.

Idea → Sampling repeatedly from experience → making more use of the data

Deep Q-Networks (DQN) → uses ER & fixed Q-targets (Q-learning) i.e. off policy

→ Big FA → NN

→ take action a_t → ϵ-greedy policy

→ store ~~transition~~ $(s_t, a_t, r_{t+1}, s_{t+1})$ in memory D.

→ sample random minibatches (s, a, r, s') from D. (64).

→ compute Q-learning targets wrt old, fixed params w^-.

→ optimise MSE.

$$L_i(w_i) = E_{s,a,s,r \sim D}\left[\left(r + \gamma \max_{a'} Q(s', a', w_i^-) - Q(s, a_i, w_i)\right)^2\right]$$

→ variant of SGD.

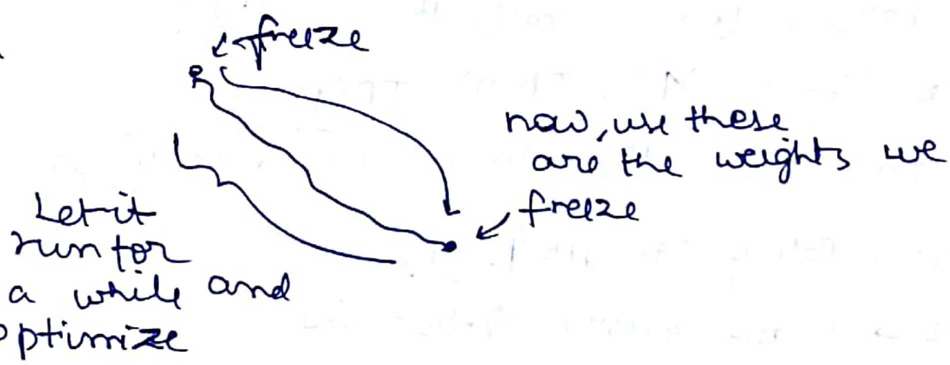
Generally, SARSA (and others) are unstable with SGD.

But this works. → Reasons ① ER. → stabilizes the NN as it decorrelates the trajectory (random sample).

② fixed Q-targets.

2 networks → freeze the old network & freeze the ~~old~~ bootstrapping

Idea



→ End-End Learning of Q(s, a) from pixels s.

→ I/p → s → stack of raw pixels from last 4 frames.

→ O/p → Q(s, a) for 18 possible actions

→ Reward → change in score.

SGD with ER → works.

→ For Linear FA, can jump directly to the minimum (closed form)

policy evaluation in 1 step.

{ Linear Least squares prediction:

→ Wouldn't want any update at optimal point ∴ $E_D(\Delta w) = 0$

$$\alpha \sum_{t=1}^{T} x(s_t)\left(v_t^{\pi} - x(s_t)^T w\right) = 0$$

$$\sum_{t=1}^{T} x(s_t) v_t^{\pi} = \sum_{t=1}^{T} x(s_t)\, x(s_t)^T w$$

$$w = \left(\sum_{t=1}^{T} x(s_t)\, x(s_t)^T\right)^{-1} \sum_{t=1}^{T} x(s_t)\, v_t^{\pi}.$$

Incremental matrix inverse → $O(N^2)$.

Depends on the # parameters now, not the # states

∴ if # parameter is small, this works!

Can replace v_π by MC, TD(0), TD(λ).

Performs much better than incremental → for linear

Least squares Policy Iteration (LSPI).

① Evaluate → Least square Q-learning

② Greedy policy ~~evaluation~~ improvement.