Lecture notes - Introduction to Reinforcement learning with David Silver

Lecture 6

By - Amal Sunny

Value function approximation

Need for it

- In real world applications of RL, we come across many large problems, for example like-
  - Backgammon: 10^{20} states
  - Computer Go(the game): 10^{170} states
  - Robots: continous state space
- We cannot possibly store all the state space values for these problems(even backgammon, a small game- has an really state space).
- So therein, we would have the **need** for an function approximator for these large problems that deals with the states we visit and end in.
- This approximator *generalizes* across those states, so that minute differences need not be considered and stored seperately. We want our value functions to understand and work with those.
- So essentially the question is how do we achieve this for large problems- i.e scale up the model-free methods covered before for both prediction and control.

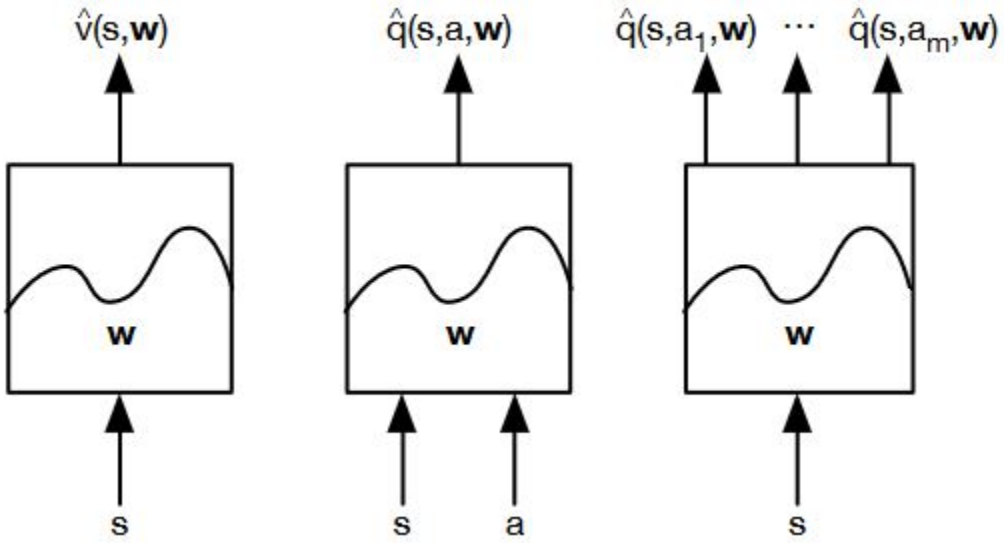
Value function Approximation (VFA)

- Until now, we just represented value functions in a look up table(either as value functions or action-value functions)
- But with large MDPs
  - Too many states too store
  - Would be too slow
- Thus our solution for large MDPS:
  - Estimate the value function with function approximation

- $$\hat{v}(s, w) \approx v_\pi(s)$$
  $$\text{or } \hat{q}(s, a, w) \approx q_\pi(s, a)$$
  - w - is the weights of your function approximator
    - could be parameters of the Neural Network, or weights if linear combination of features
  - This function approximator is for a small set of states/weights
  - Same thing can be done for action-value function too
- It also lets us generalize, as in we fit this function to seen states and query unseen states based on them.
- And how this works, is we update the parameter of the function approximator **using** the methods we use for RL - MC or TD learning
- This gives us the target for fitting the function approximator to, so we can in turn get better estimates.

Types of VFA

- For convienience just assume the function approximator as a black box, which given certain inputs gives us the approximated value function there.
- Here, the box is the VFA.



- The first one takes in the state, and outputs the approximated value function of the state
- Second one takes in a state and action, and tells us the approximated action-value of that state-action pair.
- Third ones takes in a state, and approximates **all** state-action pairs possible from that state, and gives their action-value function.

- We want that tunable VFA, to give approximated values as close to the actual value function as possible.

Which Function approximators ?

- There are a lot of choices availible to us, such as:
  - Linear combination of features
  - Neural Networks
  - Decision tree
  - Nearest neighbour
- But we'd only be considering certain specific ones, namely differentiable function approximators - because it'll be easier to adjust the parameters if we can calculate the gradient.
- i.e.
  - Linear combination of features
  - Neural network
- Secondly, our training method should be one that is suitable for non-stationary and non-iid(independent identical distributed) data, due the how the policy keeps improving and how the data is closely correlated to each other.

Incremental Methods

Methods that improve incrementally over each step, rather than batchwise.

Gradient Descent (or Stochastic Gradient Descent - SGD)

- If we keep doing this repeatedly, then the stochastic approximation theory tells us this will eventually minimize the mean square error b/w approximated and actual value function

Feature Vectors

- Features for a state, are nothing but information about the state space- anything that tells you something about the state. Eg:
  - A landmark, and its distance from your robot

    - Configurations of pieces on a chess board
- Essentially it compresses information about the state into a few features
- Represented by a feature vector

$$x(S) = \begin{pmatrix} x_1(S) \\ \vdots \\ x_n(S) \end{pmatrix}$$

Linear Value function approximation

- The value function is represented by a linear feature vector as mentioned above.

$$\hat{v}(S, w) = x(S)^T w = \sum_{j=1}^{n} x_j(S) w_j$$

- Thus, we get the objective function as:

$$J(w) = E_\pi[(v_\pi(S) - x(S)^T w)^2]$$

- The objective functions ends up in a quadratic on w. That ends up in a convex shape, thats easy to optimize.
    - You never get stuck in local optimum, always end up converging to global optimum.
- Update rule is simple as well;

$$\nabla_w \hat{v}(S, w) = x(S)$$
$$\Delta w = \alpha(v_\pi(S) - \hat{v}(S, w)) x(S)$$

i.e update = step-size x prediction error x feature value

Table lookup

- Table lookup is a special case of linear VFA where there's enough features to correspond to each state.
- Each feature corresponds to unique state, thus feature vector always has only one 1 and rest all 0s

$$x^{\text{table}}(S) = \begin{pmatrix} 1(S = s_1) \\ \vdots \\ 1(S = s_n) \end{pmatrix}$$

- Parameter vector w gives value of each individual state

$$\hat{v}(S,w) = \begin{pmatrix} 1(S=s_1) \\ \vdots \\ 1(S=s_n) \end{pmatrix} \cdot \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix}$$

Incremental prediction algorithms

- We normally assume the true value function $v_\pi(s)$ is given to us by a supervisor.
- However in RL, no supervisor
- Thus we substitute this *true* value, with some target depending on the algorithm
- We essentially do supervised learning for VFA, using respective target of whichever algorithm we're following.
- For MC, target is return
  - $$\Delta w = \alpha(G_t - \hat{v}(S_t,w))\nabla w \hat{v}(S_t,w)$$
- For TD(0), target is the TD target - one step look ahead reward + next state value function(here predicted)
  - $$\Delta w = \alpha(R_{t+1} + \gamma\hat{v}(S_{t+1},w) - \hat{v}(S_t,w))\nabla w \hat{v}(S_t,w)$$
- For TD(λ), target is λ-return G_t^λ
  - $$\Delta w = \alpha(G_t^\lambda - \hat{v}(S_t,w))\nabla w \hat{v}(S_t,w)$$

MC with VFA

- Return G_t is unbiased noisy sample of true value function
- Thus, we can consider it as a supervised learning problem, given training data
  - $\langle S_1, G_1 \rangle$, $\langle S_2, G_2 \rangle$, ..., $\langle S_T, G_T >$
- For example, with linear MC-policy evaluation
  - $$\Delta w = \alpha(G_t - \hat{v}(S_t,w))x(S_t)$$
- MC VFA will definitely work, it uses an unbiased estimate(noisy) and since its a linear regression SGD will eventually converge. Only issue is it might be slow, due to return being a noisy target.

TD with VFA

- Each step, reward + query own VFA to get val function for next step
- Biased - due to going through our VFA to get estimate
- Can still apply supervised learning, with same data like we did above(even if biased here)
- For example, with linear TD(0)
  - $$\Delta w = \alpha(R + \gamma\hat{v}(S\prime, w) - \hat{v}(S, w))\nabla w\hat{v}(S, w) \\ = \alpha\delta x(S)$$
- Despite there being bias, it has been demonstrated that linear TD(0) converges *close* to the global optimum.

TD(λ) with VFA

- The λ-return G_t^λ is also a biased sample of true value $v_\pi(s)$
- Again, applying supervised learning to the required data
- Forward view linear TD(λ)
  - $$\Delta w = \alpha(G_t^\lambda - \hat{v}(St, w))\nabla w\hat{v}(St, w) \\ = \alpha(G_t^\lambda - \hat{v}(S_t, w))x(S_t)$$
- Backward view linear TD(λ)
  - $$\delta t = R_{t+1} + \gamma\hat{v}(S_{t+1}, w) - \hat{v}(S_t, w) \\ E_t = \gamma\lambda E_{t-1} + x(S_t) \\ \Delta w = \alpha\delta_t E_t$$
- If we take our changes at the end of both fwd and backward at the end of an episode, we would find both equivalent.

Control with VFA

- Policy evaluation: Approximate policy evaluation $\hat{q}(\cdot, \cdot, w) \approx q_\pi$
- Policy improvement: ϵ-greedy policy improvemen
- Same as before, but with approximate policy
- We update our VFA, and then immediately act greedily w.r.t it(ϵ-greedy) - then update VFA and

so on.

- Does it get to optimal q_* ? No, but we can't even be sure if q_* is representable anymore(given the large size and function approximation)

Action-value function approximation

- Similar to the state-value function approximation, we now do it for action-value function
  - $$\hat{q}(S, A, w) \approx q_\pi(S, A)$$
- To calculate it, we go with the same metric of weights that minimize mean-squared error
  - $J(w) = E_\pi[(q_\pi(S, A) - \hat{q}(S, A, w))^2]$
- We apply SGD here too, to find the local minimum

$$-1/2 \nabla_w J(w) = (q_\pi(S, A) - \hat{q}(S, A, w)) \nabla_w \hat{q}(S, A, w)$$
$$\Delta w = \alpha(q_\pi(S, A) - \hat{q}(S, A, w)) \nabla_w \hat{q}(S, A, w)$$

Linear Action-Value Function Approximation

- We represent state and action by a feature vector (just like for state approximation)

$$x(S, A) = \begin{pmatrix} x_1(S, A) \\ \vdots \\ x_n(S, A) \end{pmatrix}$$

- Following up, to represent action-value function by linear combination of features

$$\hat{q}(S, A, w) = x(S, A)^T w = \sum_{j=1}^{n} x_j(S, A) w_j$$

- SGD update

$$\nabla_w \hat{q}(S, A, w) = x(S, A)$$
$$\Delta w = \alpha(q_\pi(S, A) - \hat{q}(S, A, w)) x(S, A)$$

Incremental Control Algorithms

- Like we did in prediction, we substitute targets for $q_\pi(S, A)$
  - For MC, target is return G_t
    - $$\Delta w = \alpha(G_t - \hat{q}(S_t, A_t, w)) \nabla w \hat{q}(S_t, A_t, w)$$

- For TD(0), the target is the TD target $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$
  - $$\Delta w = \alpha(R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, w) - \hat{q}(S_t, A_t, w)) \nabla w \hat{q}(S_t, A_t, w)$$
- For forward-view TD(λ):
  - $$\Delta w = \alpha(q_t^\lambda - \hat{q}(S_t, A_t, w)) \nabla w \hat{q}(S_t, A_t, w)$$
- For backward-view TD(λ):
  - $$\begin{gathered} \delta_t = R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, w) - \hat{q}(S_t, A_t, w) \\ E_t = \gamma \lambda E_{t-1} + \nabla w \hat{q}(S_t, A_t, w) \\ \Delta w = \alpha \delta_t E_t \end{gathered}$$

Convergence

- However, all these methods are not guarenteed to converge.
- Specific examples(Baird's counterexample), created to prove the non-convergence of certain algorithms
- They converge for most cases, or hover around optimal value(chattering)

Prediction

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

Gradient TD

- Emphatic TD and Gradient TD are newly discovered methods that fixes the problems the TD algorithm has when it bootstraps.
- Since TD does not follow gradient of any objective function - it can diverge off-policy or when using non-linear function approximation.
- Gradient TD follows true gradient of projected Bellman error.

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

Control

- Suprisingly problamatic, we rarely get guarentees of convergence.
- Chattering - a situation where it converges very close to optimal value, but "improvements" *can* end up degrading value when near to optimal.

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

(✓) = chatters around near-optimal value function

Batch Methods

- Not sample efficient - meaning we take a sample, and then discard it. We do not make full use of it as much as we can. Not data efficient.
- Best fitting value for the entire data in the batch.
- Given the agent's experience

> "Life is one big training set"

- So now our question is how do we measure the goodness of the fit ?

Least Squares Prediction

- So, given value function approximation, and experience D consisting of <state,value> pairs
- How do we find optimal parameter w for VFA ?
- So one measure of best fit would be Least squares prediction.
- Least squares algorithms find parameter vector w minimising sum-squared error between $\hat{v}(s_t, w)$ and target values v_t^π

$$LS(w) = \sum_{t=1}^{T} (v_t^\pi - \hat{v}(s_t, w))^2 = E_D[(v^\pi - \hat{v}(s, w))^2]$$

SGD with Experience Replay

- Instead of throwing away our data, we cache it and call this our experience.
- Every time step we sample (randomly) from this experience.
- And update our SGD accordingly
- Given experience consisting of ⟨state,value⟩ pairs
  - $$D = \{ \langle s_1, v_1^\pi \rangle , \langle s_2, v_2^\pi \rangle , ..., \langle s_T, v_T^\pi \rangle \}$$
- Repeat:
  - 1: Sample state, value from experience
    - $$< s, v^\pi > \ D$$
  - 2: Apply SGD update
    - $$\Delta w = \alpha(v^\pi - \hat{v}(s, w)) \nabla_w \hat{v}(s, w)$$
- This converges to least squares solution.

$$w^\pi = argmin_w LS(w)$$

- Experience relay *decorrelates* trajectories, instead of seeing highly correlated parts of trajectory that follow one after another - we get tuples in random order.

Experience replay in Deep Q-Networks(DQN)

- DQN uses **experience replay** and **fixed Q-targets**.
  - Take action at according to ϵ-greedy policy

- Store transition $(s_t, a_t, r_t + 1, s_{t+1})$ in replay memory D.
- Sample random mini-batch of transitions (s ,a,r ,s ′) from D
- Compute Q-learning targets w.r.t. old, fixed parameters w^-
- Optimise MSE between Q-network and Q-learning targets

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

- In addition to the experience relay advanatage, we preserve two different networks for iterating through the learning process
- We bootstrap towards the old frozen target having old parameters w^- from a few episodes/experiences ago
  - The updation of values happens after a couple thousand episodes, where we equate our old network to the new one and optimise the MSE for it.
  - The reason old frozen values are taken are to prevent further movement of a target, if both networks were the same then our target would keep moving every time and cause further instability.

Linear Least Squares Prediction

- Experience replay finds least squares solution
- But it may take many iterations
- Using linear value function approximation $\hat{v}(s, w) = x(s)^T w$
- We can solve the least squares solution directly
- At minimum of LS(w), expected update must be 0 (i.e global minimum - so no update needed)

$$E_D[\Delta w] = 0$$

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

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

- For N features, this would take $O(N^3)$ - where N is number of features
- However there's an incremental solution done in $O(N^2)$ using Shermann-Morrison

Prediction

- Since true value of v_t^π is unknown, we must use targets again:

- LSMC Least Squares Monte-Carlo uses return
- LSTD Least Squares Temporal-Difference uses TD target
- LSTD(λ) Least Squares TD(λ) uses λ-return

LSMC

$$0 = \sum_{t=1}^{T} \alpha(G_t - \hat{v}(S_t, \mathbf{w}))\mathbf{x}(S_t)$$

$$\mathbf{w} = \left(\sum_{t=1}^{T} \mathbf{x}(S_t)\mathbf{x}(S_t)^\top\right)^{-1} \sum_{t=1}^{T} \mathbf{x}(S_t)G_t$$

LSTD

$$0 = \sum_{t=1}^{T} \alpha(R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}))\mathbf{x}(S_t)$$

$$\mathbf{w} = \left(\sum_{t=1}^{T} \mathbf{x}(S_t)(\mathbf{x}(S_t) - \gamma\mathbf{x}(S_{t+1}))^\top\right)^{-1} \sum_{t=1}^{T} \mathbf{x}(S_t)R_{t+1}$$

LSTD(λ)

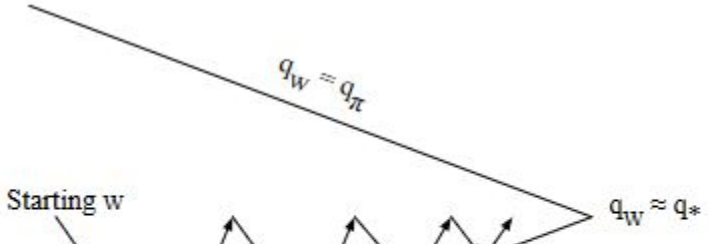
$$0 = \sum_{t=1}^{T} \alpha\delta_t E_t$$

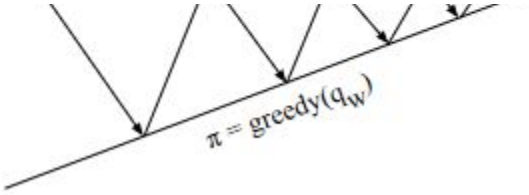
$$\mathbf{w} = \left(\sum_{t=1}^{T} E_t(\mathbf{x}(S_t) - \gamma\mathbf{x}(S_{t+1}))^\top\right)^{-1} \sum_{t=1}^{T} E_t R_{t+1}$$

Convergence

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

Policy iteration





Convergence of control algorithm

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

(✓) = chatters around near-optimal value function

Problems

Problem 1: Taken from Sutton and Barto

Exercise *9.1* Show that tabular methods such as presented in Part I of this book are a special case of linear function approximation. What would the feature vectors be? □

The Part I refers to tabular solution methods.

Ans: The tabular solutions can easily be represented as a feature vector, with each vector corresponding to a state in the table. Then that feature simply needs to point to one state, and display 0 otherwise to still be a valid function approximator.

For eg:

$$x^{\text{table}}(S) = \begin{pmatrix} 1(S = s_1) \\ \vdots \\ 1(S = s_n) \end{pmatrix}$$

Where $s_1, s_2, \cdots, s_n$ are the states in the tabular method.

Problem 2: Corresponds to polynomial feature construction, Sutton and Barto

Exercise 9.3 What n and $c_{i,j}$ produce the feature vectors $\mathbf{x}(s) = (1, s_1, s_2, s_1 s_2, s_1^2, s_2^2, s_1 s_2^2, s_1^2 s_2, s_1^2 s_2^2)^\top$? □

Ans:

We know that for polynomial feature construction,

$$x_i(s) = \prod_{j=1}^{k} s_j^{c_{i,j}}$$

where s_j corresponds to states,
x_i is the polynomial basis function,
n is the dimension of state space,
$c_{i,j}$ is a set of integer values

Clearly from this, we can tell from the feature vector, the powers taken by each component lies within (0,2)

Thus, $c_{i,j} = \{0, 1, 2\}$

And number of states here is 2, thus $k = 2$

Problem of the week: To try and beat the snake game.