Reinforcement Learning **Lecture–6**

Function Approximation

Anubhav Gupta

Reinforcement learning methods such as dynamic programming, Monte-Carlo and TD methods are called tabular methods, because these methods require calculating and storing the value functions for all state (state-action pairs). These methods don't scale well for problems with large number of states, because calculating and storing the value functions require a lot of memory and time, and large amount of data is required to accurately compute the the value functions. So, we want to able to learn from limited target data for states and generalize well to other, possibly unseen states.

Function approximation provides this generalization by approximating the value function v_π from the experience generated using policy π.

$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$$
$$\hat{v}(q, a, \mathbf{w}) \approx q_\pi(s, a)$$

The approximate value function is represented by the parameter $\mathbf{w} \in \mathbb{R}^d$. Here d is the dimensionality of our feature space, i.e. all states/state-action pairs are represented by a d-dimensional vector. The objective of value function approximation is then, to learn a weight for each of these dimensions.

Any supervised learning model can be used as a function approximator such as linear model, neural networks, decision trees and others. In reinforcement learning, the model should be able to learn online, while the agent is interacting with the environment. The function approximator should handle the issues such as *non-stationarity* (data comes in sequential form), *bootstrapping* and *delayed targets* present in reinforcement learning.

1 Objective Function

Given a policy π, the objective is to find weight vector $\mathbf{w}$ which approximate the value function v_π for most of the states correctly. Since it is not always possible to learn the exact value for all the states, a state distribution μ is specified which represents the importance of each state. i.e.

$$\mu(s) \geq 0, \ \sum_s \mu(s) = 1$$

$\mu(s)$ is usually chosen to be the fraction of time spent in state s. The objective function that we will use is called **Mean Squared Value Error**:

$$\overline{VE}(w) = \sum_{s \in \mathcal{S}} \mu(s)[v_\pi(s) - \hat{v}(s, w)]^2$$

In all the algorithms discussed below, we will assume that all states are equally important, and thus the objective is to find the weights $\mathbf{w}$ such that

$$J(\mathbf{w}) = \mathbb{E}_\pi \left[(v_\pi(s) - \hat{v}(s, w))^2\right] \tag{1.1}$$

2 Overview of Stochastic Gradient Descent

Gradient descent is a widely used method to optimize (minimize) any differential objective function $J(\mathbf{w})$. Gradient descent finds the optimal parameter weights $\mathbf{w}$ by repeatedly adjusting it in the direction of negative gradient of objective J. Stochastic gradient descent (SGD) is a variant of gradient descent where weights are updated after every time step. The gradient of a differential function J w.r.t. $\mathbf{w}$ is given by

$$\nabla_w J(\mathbf{w}) = \begin{bmatrix} \frac{\partial J(\mathbf{w}}{\partial w_1} \\ \frac{\partial J(\mathbf{w}}{\partial w_2} \\ \vdots \\ \frac{\partial J(\mathbf{w}}{\partial w_d} \end{bmatrix}$$

Then, SGD updates the weights in the direction of negative gradient. That is at each time step

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

and weights are updated as

$$\mathbf{w} \leftarrow \mathbf{w} + \Delta \mathbf{w}$$

Note. *Gradient descent always converges to local optimum, given step size α is chosen to be sufficiently small.*

3 Incremental Methods for Prediction using VFA

Incremental methods to approximate the value function use gradient descent based learning algorithms such as stochastic gradient descent (SGD). The objective is to find the weight vector $\mathbf{w}$ minimizing the objective function given by 1.1. Full gradient descent makes the update

$$\begin{aligned}\Delta \mathbf{w} &= -\frac{1}{2}\alpha \nabla_w J(\mathbf{w}) \\ &= \alpha \mathbb{E}_\pi \left[(v_\pi(s) - \hat{v}(s, w)) \nabla_w \hat{v}(s, w) \right]\end{aligned}$$

Update made by SGD at each time step is:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha (v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)) \nabla_w \hat{v}(S_t, \mathbf{w}_t) \tag{3.1}$$

Let feature vector for state S is denoted by the column vector $x(S)$. i.e.

$$x(S) = \begin{bmatrix} x_1(S) \\ x_2(S) \\ \vdots \\ x_d(S) \end{bmatrix}$$

When the value function is represented by the linear combination of features, the method is called linear VFA.

$$\hat{v}(S, \mathbf{w}) = x(S)^T \mathbf{w} = \sum_{j=1}^{d} x_j(S) \mathbf{w}_j$$

Then for linear VFA, SGD update rule becomes:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha (v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)) x(S_t) \tag{3.2}$$

3.1 Prediction using Monte-Carlo with VFA

We have seen how to approximate the value function if we know the true value function v_π. However, in practice there is no supervision and true value function is not given. Monte-Carlo method uses the expected return G_t as the true value of state S_t. So, SGD update for Monte-Carlo VFA with linear function is given as:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha (G_t - \hat{v}(S_t, \mathbf{w}_t)) x(S_t)$$

The overall algorithm for gradient MC algorithm is given by algorithm-1.

Algorithm 1: Gradient Monte-Carlo for estimating $\hat{v} \approx v_\pi$

Input: The policy π to be evaluated
A differential function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm Parameters: Step size $\alpha \in (0, 1]$
Initialization: Initialize value function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily

```
for each episode until convergence do
    Generate an episode S_0, A_0, R_1, S_1, ..., S_{T-1}, A_{T-1}, R_T following policy π
    for each time-step t = 0, 1, ..., T − 1 of episode do
        w ← w + α(G_t − v̂(S_t, w)) ∇_w v̂(S_t, w)
    end
end
```

Note. *Since return G_t is an unbiased estimate of true value function $v_\pi(S_t)$, the MC estimate of v_π using SGD converges to local optimum. In case of linear VFA, it converges to global optimum.*

3.2 Prediction using TD Learning with VFA

In MC, we substituted the expected return in place of true value v_π of value function. In TD methods, this quantity is approximated by *TD target* $R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w})$. TD(0) method with linear function approximation makes the update:

$$\begin{aligned}\mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha(R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t))\ \nabla_w\hat{v}(S_t, w) \\ &= \mathbf{w}_t + \alpha(R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t))\ x(S_t) \\ &= \alpha\delta x(S_t)\end{aligned}$$

Bootstrapping methods such as TD are not instances of true gradient descent. This is because they only take into consideration the effect of weight vector $\mathbf{w}_t$ on estimates and ignore it's effect on the target. Such methods are called **semi-gradient methods** as they only include a part of the gradient rather than full gradient. Algorithm-2 gives the pseudocode of semi-gradient TD(0) for prediction using VFA.

Algorithm 2: Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: The policy π to be evaluated
A differential function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$, with $\hat{v}(terminal, \cdot) = 0$
Algorithm Parameters: Step size $\alpha \in (0, 1]$
Initialization: Initialize value function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily

```
for each episode until convergence do
    Initialize S
    for each step of episode do
        choose A ~ π(·|S)
        Take action A, observe reward R and next state S'
        w ← w + α(R + γv̂(S', w) − v̂(S, w)) ∇_w v̂(S, w)
    end
end
```

Similar to TD(0), TD(λ) is also a semi-gradient method for estimating v_π. TD(λ) uses the λ-return G_t^λ instead of true value function $v_\pi(S_t)$. Updates made by forward view and backward view TD(λ) are given as follows:

- **Forward view linear TD(λ)**

$$\begin{aligned}\mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha(G_t^\lambda - \hat{v}(S_t, \mathbf{w}_t))\ \nabla_w\hat{v}(S_t, \mathbf{w}) \\ &= \mathbf{w}_t + \alpha(R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t))\ x(S_t)\end{aligned}$$

- **Backward view linear TD(λ)**

$$\begin{aligned}\delta_t &= R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t)) \\ E_t &= \gamma\delta E_{t-1} + x(S_t) \\ \mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha\delta_t E_t\end{aligned}$$

Note. *Following facts are known about TD estimates:*

- *Both forward view and backward view of TD(λ) are equivalent*
- *Both TD target and λ-return are biased estimate of v_π*
- *Linear TD(0) converges (close) to global optimum*

4 Incremental Methods for Control using VFA

So far we have seen how parametric function approximation can be used for prediction, i.e. to estimate the state value functions. Let us now consider the problem of control, where we want to find the optimal policy π_*. For control, we use the idea of generalized policy iteration (GPI). We now approximate action value function $\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$ for estimating the value of state-action pairs, and then ϵ-greedy policy is used to obtain an improvement of policy π.

For approximating the action value functions by the parameter vector $\mathbf{w}$, every state-action pair is represented by a feature vector:

$$x(S, A) = \begin{bmatrix} x_1(S, A) \\ x_2(S, A) \\ \vdots \\ x_d(S, A) \end{bmatrix}$$

A linear VFA, then approximates the action value function with linear combination of features:

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

The semi-gradient update rule for control is given by:

$$\begin{aligned} \mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha(q_\pi(S_t, A_t) - \hat{q}(S_t, A_t, \mathbf{w}_t))\ \nabla_w \hat{v}(S_t, A_t, \mathbf{w}_t) \\ &= \mathbf{w}_t + \alpha(q_\pi(S_t, A_t) - \hat{q}(S_t, A_t, \mathbf{w}_t))\ x(S_t, A_t) \end{aligned}$$

Just like in prediction, Monte-Carlo and TD methods use different values as true target $q_\pi(S_t, A_t)$. **Monte-Carlo** uses the return G_t as target. i.e.

$$\Delta\mathbf{w}_t = \alpha(G_t - \hat{q}(S_t, A_t, \mathbf{w}_t))\ \nabla_w \hat{q}(S_t, A_t, \mathbf{w}_t)$$

4.1 Episodic Control using Semi-gradient Sarsa

The semi-gradient Sarsa uses the TD target $R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t)$ as the true action value $q_\pi(S_t, A_t)$. The update for semi-gradient Sarsa is given by following equation:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha(R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t))\ \nabla_w \hat{q}(S_t, A_t, \mathbf{w}_t)$$

Pseudocode for the complete semi-gradient Sarsa is given by algorithm-3.

Algorithm 3: Semi-gradient Sarsa for estimating $\hat{q} \approx q_*$

Input: A differential action value function $\hat{q} : \mathcal{S}^+ \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm Parameters: Step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialization: Initialize value function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily

```
for each episode do
    Initialize S
    Choose A from S using policy π
    for each time-step t of episode do
        Take action A, observe R, S'
        if S' is terminal then
            w ← w + α(R − q̂(S, A, w)) ∇_w q̂(S, A, w)
            Go to next episode
        end
        Choose A' as a function of q̂(S', ·, w) (e.g. ε-greedy)
        w ← w + α(R + γq̂(S', A', w) − q̂(S, A, w)) ∇_w q̂(S, A, w)
        S ← S'
        A ← A'
    end
end
```

4.2 Semi-gradient Sarsa(λ)

The effect of bootstrapping in function approximation methods is shown many times. Methods such as gradient Monte-Carlo, that don't bootstrap are rarely used in practice. Although semi-gradient Sarsa gives much better results, TD(λ) methods perform even better when λ is tuned carefully. Sarsa(λ) is one such algorithm which uses exponential weighting for n-step returns. The update equations for semi-gradient Sarsa(λ) algorithm for episodic control is given as follows:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \Delta\mathbf{w}_t$$

where $\Delta\mathbf{w}_t$ is calculated for forward and backward views as follows:

- For **forward view Sarsa(λ)**, target is action-value λ return

$$\Delta \mathbf{w}_t = \alpha(G_t^\lambda - \hat{q}(S_t, A_t, \mathbf{w}_t))\ \nabla_w \hat{q}(S_t, A_t, \mathbf{w}_t)$$

- For **backward view Sarsa(λ)**, equivalent update is

$$\begin{aligned}\delta_t &= R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)) \\ E_t &= \gamma \delta E_{t-1} + \nabla_w \hat{q}(S_t, A_t, \mathbf{w}_t) \\ \Delta \mathbf{w}_t &= \alpha \delta_t E_t\end{aligned}$$

4.3 Semi-gradient Q-learning

Q-learning can also be used with function approximation in the same way as Sarsa. Semi-gradient Q-learning uses **Q-learning target** $R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, \mathbf{w}_t)$ as the true action value $q_\pi(S_t, A_t)$. Then the update for semi-gradient Q-learning is given as

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha(R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t))\ \nabla_w \hat{q}(S_t, A_t, \mathbf{w}_t) \tag{4.1}$$

5 Batch Methods for Function Approximation

Gradient descent based methods discussed so far are simple to understand and very appealing, but these methods are not sample efficient. After looking at the experience, the function approximator is updated in the direction of negative gradient, and then the experience is thrown away. Batch methods on the other hand, aim to find the best fitting value function to all the experience available (agent's "training data").

One such method is **least squares** prediction which, given function approximation $\hat{v}(s, \mathbf{w})$ and training data $\mathcal{D} = \{\langle s_1, v_1^\pi \rangle, \langle s_2, v_2^\pi \rangle, \cdots \langle s_T, v_T^\pi \rangle\}$, finds parameter vector $\mathbf{w}$ which minimizes the sum of squared error between $\hat{v}(S_t, \mathbf{w})$ and target values v^π.

$$\begin{aligned}LS(\mathbf{w}) &= \sum_{t=1}^{T} (v_t^\pi - \hat{v}(S_t, \mathbf{w}))^2 \\ &= \mathbb{E}_{\mathcal{D}} \left[(v^\pi - \hat{v}(s, \mathbf{w}))^2 \right]\end{aligned}$$

5.1 SGD with Experience Replay

Experience replay is one way of finding the least squares solution. Instead of using the experience for just one update, the idea of experience replay is to store (cache) the entire experience in the form of a training data (or replay buffer) $\mathcal{D}$.

$$\mathcal{D} = \{\langle s_1, v_1^\pi \rangle, \langle s_2, v_2^\pi \rangle, \cdots \langle s_T, v_T^\pi \rangle\}$$

and then sample this data to make SGD updates. The algorithm can be described in two simple steps:

- Sample state,value from experience

$$\langle s, v^\pi \rangle \sim \mathcal{D}$$

- Apply SGD update based on this sample

$$\Delta w = \alpha(v^\pi - \hat{v}(s, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(s, \mathbf{w}) \tag{5.1}$$

Experience replay method has several advantages over standard online learning methods:

- Experience replay provides greater data efficiency, since each step of experience is potentially used for making several updates.
- Randomly sampling the data removes correlations between consecutive samples, and therefore reduces the variance of updates.

The stochastic gradient descent method with experience replay converges to least squares solution. i.e.

$$w^\pi = \operatorname*{argmin}_{\mathbf{w}} LS(\mathbf{w})$$

Note. *Refer to DQN algorithm discussed in section 6 for a full algorithm using experience replay.*

5.2 Linear Least Squares Prediction

Experience replay is one method of finding the least squares solution, but it may take many iterations in general to do so. For linear VFA, i.e. $\hat{v}(s, \mathbf{w}) = \mathbf{w}^T x(s)$, it turns out we can directly find least squares solution.

The goal is to find weight vector $\mathbf{w}$ which minimizes sum of squared error as defined earlier. Consider the update given by equation 5.1. If we have found the minimum value of $LS(\mathbf{w})$, then for this $\mathbf{w}$, the expected next update must be zero. i.e.

$$\begin{aligned}
\mathbb{E}_{\mathcal{D}}\left[\Delta \mathbf{w}\right] &= 0 \\
\alpha \frac{1}{T} \sum_{t=1}^{T} x(S_t)(v_t^\pi - x(S_t)^T \mathbf{w}) &= 0 \\
\sum_{t=1}^{T} x(S_t) v_t^\pi &= \sum_{t=1}^{T} x(S_t) x(S_t)^T \mathbf{w} \\
\mathbf{w} &= \left(\sum_{t=1}^{T} x(S_t) x(S_t)^T \right)^{-1} \sum_{t=1}^{T} x(S_t) v_t^\pi
\end{aligned}$$

For feature vector of size d, direct solution can be found in $\mathcal{O}(d^3)$ time. A $\mathcal{O}(d^2)$ solution also exists by using *Shermann-Morrison* formula.

Different methods use different values for true target v^π. Some of these linear least squares prediction algorithms are as given below.

5.2.1 LSTD (Least Squares Temporal Difference)

LSTD approximates the true value v_t^π by TD target $R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t)$. Then, using above argument optimal weight vector $\mathbf{w}$ is given by

$$\begin{aligned}
\sum_{t=1}^{T} x(S_t)(R_{t+1} + \gamma x(S_{t+1})^T \mathbf{w} - x(S_t)^T \mathbf{w}) &= 0 \\
\sum_{t=1}^{T} x(S_t) \left(x(S_t) - \gamma x(S_{t+1}) \right)^T \mathbf{w} &= \sum_{t=1}^{T} x(S_t) R_{t+1}
\end{aligned}$$

and optimum $\mathbf{w}$ is given by

$$\mathbf{w}_{LSTD} = \left(\sum_{t=1}^{T} x(S_t) \left(x(S_t) - \gamma x(S_{t+1}) \right)^T \right)^{-1} \sum_{t=1}^{T} x(S_t) R_{t+1} \tag{5.2}$$

The quantity given in equation 5.2 is called **TD fixed point**. This is also where the linear semi-gradient TD(0) algorithm (discussed earlier) converges.

5.2.2 LSMC (Least Squares Monte Carlo)

LSMC approximates the true value by expected return G_t. Therefore estimated parameter vector given by LSMC is

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

5.2.3 LSTD(λ) (Least Squares TD(λ))

It uses λ-return in place of true values v^π.

$$\begin{aligned}
\delta_t &= R_{t+1} + \gamma x(S_{t+1})^T \mathbf{w} - x(S_t)^T \mathbf{w} \\
E_t &= \gamma \delta E_{t-1} + x(S_t) \\
\Delta \mathbf{w} &= \alpha \delta_t E_t
\end{aligned}$$

On/Off-Policy	Algorithm	Tabular	Linear VFA	Nonlinear VFA
On-policy	MC	yes	yes	yes
	LSMC	yes	yes	-
	TD(Sarsa)	yes	yes	no
	LSTD	yes	yes	-
Off-policy	MC	yes	yes	yes
	LSMC	yes	yes	-
	TD(Q-Learning)	yes	no	no
	LSTD	yes	yes	-

Table 1: Convergence properties of VFA based prediction algorithms.

At optimum $\mathbf{w}$,

$$\begin{aligned}
\mathbb{E}\left[\Delta \mathbf{w}\right] &= 0 \\
\alpha \frac{1}{T} \sum_{t=1}^{T} \delta_t E_t &= 0 \\
\sum_{t=1}^{T} \left(R_{t+1} + \gamma x(S_{t+1})^T \mathbf{w} - x(S_t)^T \mathbf{w}\right) E_t &= 0 \\
\sum_{t=1}^{T} E_t R_{t+1} &= \sum_{t=1}^{T} E_t (x(S_t) - \gamma x(S_{t+1}))^T \mathbf{w}
\end{aligned}$$

and optimal $\mathbf{w}$ is given as

$$\mathbf{w}_{LSTD(\lambda)} = \left(\sum_{t=1}^{T} E_t (x(S_t) - \gamma x(S_{t+1}))^T \right)^{-1} \sum_{t=1}^{T} E_t R_{t+1} \tag{5.3}$$

Table-1 summarizes the convergence of on-policy and off-policy prediction methods.

5.3 Least Squares Policy Iteration

Least squares policy iteration (LSPI) again uses the idea of GPI with policy evaluation using least squares Q-learning and greedy policy improvement. A linear least squares approximator, then approximates the action value function with linear combination of features:

$$\hat{q}(S, A, \mathbf{w}) = x(S, A)^T \mathbf{w} \approx q_\pi(S, A)$$

For performing policy evaluation, we need to utilize all of agent's experience, while we also want to improve the policy at the same time. Therefore we use the off-policy Q-learning for approximating the value function. The experience $\mathcal{D}$ is generated from many different policies. The idea of Q-learning is as follows:

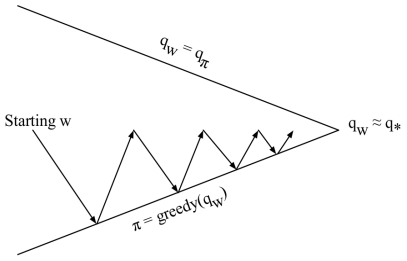

Figure 1: Least squares policy iteration

- Use experience generated by old policy

$$S_t, A_t, R_{t+1}, S_{t+1} \sim \pi_{old}$$

- Consider alternative successor state $A' = \pi_{new}(S_{t+1})$
- Update $\hat{q}(S_t, A_t, \mathbf{w})$ toward value of alternative successor state $R_{t+1} + \gamma \hat{q}(S_{t+1}, A', \mathbf{w})$

This gives us the Q-learning update given in equation 4.1. Since π_{new} is a greedy policy, it is same as

$$\begin{aligned}
\delta_t &= R_{t+1} + \gamma \hat{q}(S_{t+1}, \pi_{new}(S_{t+1}), \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)) \\
\Delta \mathbf{w}_t &= \alpha \delta_t x(S_t, A_t)
\end{aligned}$$

Algorithm	Tabular	Linear VFA	Nonlinear VFA
Monte-Carlo	yes	(yes)	no
Sarsa	yes	(yes)	no
Q-learning	yes	no	no
LSPI	yes	(yes)	-

Table 2: Convergence properties of VFA based control algorithms. (yes) means algorithm chatters around near-optimal value function.

LSTDQ Algorithm: Using linear least squares methods, LSTDQ algorithm learns the weight vector $\mathbf{w}$, which satisfies following:

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

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

$$\mathbf{w}_{LSTDQ} = \left(\sum_{t=1}^{T} x(S_t, A_t) \left(x(S_t, A_t) - \gamma x(S_{t+1}, \pi(S_{t+1}))\right)^T\right)^{-1} \sum_{t=1}^{T} x(S_t, A_t) R_{t+1} \tag{5.4}$$

LSPI algorithm uses LSTDQ for policy evaluation and repeatedly re-evaluates experience $\mathcal{D}$ with different policies. Algorithm-4 gives the pseudocode of LSPI algorithm with LSTDQ policy evaluation.

Algorithm 4: Least squares policy iteration for estimating $\pi \approx \pi_*$

Input: Initial set of samples $\mathcal{D}_0$, possibly empty
Initialization: Initialize value function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily
Initialize policy π_0 as greedy policy according to value function Q defined by $\mathbf{w}$

$\mathcal{D} \leftarrow \mathcal{D}_0$
$\pi' \leftarrow \pi_0$
while *not converged* **do**
  Update $\mathcal{D}$ (Optional)
  $\pi \leftarrow \pi'$
  $Q \leftarrow LSTDQ(\pi, \mathcal{D})$
  for *all* $s \in \mathcal{S}$ **do**
    $\pi'(s) \leftarrow \text{argmax}\, a \in \mathcal{A} Q(s, a)$
  end
  Stop if $\pi \approx \pi'$
end
return π

Table-2 provides a summary of convergence guarantees of different VFA based control algorithms. Tabular VFA is same as using no approximation. Prediction methods also exhibit similar convergence guarantees. On thing to note here is that, even though convergence for off-policy approximation methods is not theoretically proven, these methods still work well in practice.

6 Deep Q-networks (DQN)

So far we have discussed linear methods for function approximation. The performance of such methods highly depend on the quality of features. Deep Q-learning on the other hand automatically learns the features using deep neural networks. DQN differs from other control methods in that it uses two techniques *experience replay* and a separate *target network*.

The complete DQN algorithm is presented in algorithm-5. In **experience replay**, we store the agent's experience at each time step $e_t = \langle s_t, a_t, r_t, s_{t+1} \rangle$ in a data set $\mathcal{D}$. During the inner loop of algorithm, we sample mini-batches of experience randomly from this data set and make stochastic gradient descent updates.

Algorithm 5: Deep Q-learning with experience replay for estimating $\hat{q} \approx q_*$

Algorithm Parameters: Step size $\alpha \in (0, 1]$, small $\epsilon > 0$, positive integer C
Initialization: Initialize replay memory $\mathcal{D}$ to fixed capacity N
Initialize action-value function $\hat{q}$ with random weights $\mathbf{w}$
Initialize target action-value function $\hat{q}$ with weights $\mathbf{w}^- = \mathbf{w}$

for *each episode* 1 *to* M **do**
  observe initial state S
  for *each time-step* $t = 1, 2, \cdot, T$ **do**
    Choose A as a function of $\hat{q}(S, \cdot, \mathbf{w})$ using ϵ-greedy policy
    Take action A, observe R, S'
    Update replay memory $\mathcal{D}$ with new sample $\langle S, A, R, S' \rangle$
    Sample a random mini-batch of M transitions (s_j, a_j, r_j, s_{j+1}) from $\mathcal{D}$
    for *each* $j = 1, 2, \cdots, M$ **do**
      Set true target $y_j = \begin{cases} r_j, & \text{if episode ends at state } j+1 \\ r_j + \gamma \max_{a'} \hat{q}(s_{j+1}, a', \mathbf{w}^-), & \text{otherwise} \end{cases}$
    end
    Perform SGD step on $J(\mathbf{w}) = \frac{1}{M} \sum_{j=1}^{M} (y_j - \hat{q}(s_j, a_j, \mathbf{w}))^2$ w.r.t. to parameter vector $\mathbf{w}$
    $S \leftarrow S'$
    Every C steps, reset $\mathbf{w}^- = \mathbf{w}$
  end
end

Another technique used in DQN is the use of a separate **target network** for generating the target values y_j in Q-learning updates. The algorithm updates the weights of target network after every C iterations, and it remains unchanged and generates targets y_j for remaining C steps. Fixing the targets in this way makes the algorithm more stable compared to online Q-learning algorithms.

References

[1] Reinforcement Learning: An Introduction (2nd Edition), by Richard S. Sutton, Andrew G. Barto.

[2] David Silver's course "Introduction to Reinforcement Learning" available at `https://www.davidsilver.uk/teaching/`

[3] Lagoudakis, Michail G., and Ronald Parr. "Model-free least-squares policy iteration." *Advances in neural information processing systems*. 2002.

[4] Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." *Nature 518.7540* (2015): 529-533.