Reinforcement Learning

An Introductory Note

Jingye Wang

wangjy5@shanghaitech.edu.cn

Spring 2020

Contents

8 Value Function Approximation

For some problems, the state spaces may huge or even infinity, *e.g.*:

- Backgammon: 10^{20} states;
- Chess: 10^{47} states;
- Came of Go: 10^{170} states;
- Robot Arm and Helicopter: continuous state space;

How can we scale up the model-free methods for these problems?

A solution for these is *function approximation*, which is an instance of *supervised learning*. Concretely, it follows that:

- $\hat{v}(s;\boldsymbol{w}) \approx v^{\pi}(s)$,
- $\hat{q}(s,a;\boldsymbol{w}) \approx q^{\pi}(s,a)$,
- $\hat{\pi}(s,a;\boldsymbol{w}) \approx \pi(a|s)$,

where $\boldsymbol{w}$ is the parameter to be learned. With such approximation, we can generalize from seen states to unseen states. Obviously, we have many available function approximator:

- Linear combinations of features,
- Neural networks,
- Decision trees,
- Nearest neighbors.

Among those, we will focus on the first two as they are differentiable and thus easy to be optimized. It is worth to notice that though such generalization makes the learning potentially more powerful (it may even applicable to partially observable problems), it is potentially more difficult to manage and understand.

8.1 Semi-gradient Method

Now we consider a naive case where we have an oracle for knowing the true value $v^{\pi}(s)$ for any given state s under policy π. Then the object is to find the estimator $\hat{v}(s,\boldsymbol{w})$ approximating $v^{\pi}(s)$. We can define the loss function by the mean squared error:

$$J(\boldsymbol{w}) = \mathbb{E}_{\pi}[(v^{\pi}(s) - \hat{v}(s,\boldsymbol{w}))^2],$$

and the gradient descend for the loss function is:

$$\Delta\boldsymbol{w} = -\frac{1}{2}\alpha\nabla_{\boldsymbol{w}}J(\boldsymbol{w}),$$

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

Follow the gradient descend, we can find a local minimum. Further, if the value function is represented by a linear combination of features, then such method can converge to the global optimum.

8.1.1 Semi-gradient Method for Prediction

However, in practice, no access to the oracle of the true value $v^\pi(s)$. What we have is only the reward, or, the bootstrapping estimation. Thus we can substitute the target for $v^\pi(s)$:

- For MC, the target is the actual return G_t and hence

$$\Delta \boldsymbol{w} = \alpha(G_t - \hat{v}(s_t, \boldsymbol{w}))\nabla_{\boldsymbol{w}}\hat{v}(s_t, \boldsymbol{w});$$

- For TD(0), the target is the TD target $R_{t+1} + \gamma\hat{v}(s_{t+1}, \boldsymbol{w})$ and hence

$$\Delta \boldsymbol{w} = \alpha(r_{t+1} + \gamma\hat{v}(s_{t+1}, \boldsymbol{w}) - \hat{v}(s_t, \boldsymbol{w}))\nabla_{\boldsymbol{w}}\hat{v}(s_t, \boldsymbol{w})$$

For TD(0), the gradient is also called as semi-gradient, as the target $r_{t+1} + \gamma\hat{v}(s_{t+1}, \boldsymbol{w})$ depends on the current value of the weight vector $\boldsymbol{w}$ and we just ignore such effect.

Algorithm 16 Gradient Monte Carlo Policy Evaluation

initialize the differentiable function $\hat{v}(s, \boldsymbol{w}) \in \mathbb{R}$ arbitrarily for all $s \in \mathcal{S}$;
input the policy π to be evaluated; the step size α;
for true **do**:
variants for this alg. can be start with $s_0 \in \mathcal{S}, a_0 \in \mathcal{A}(s_0)$ randomly, $\varepsilon-$greedy, *etc.*
    Generate a complete episode $\tau = (s_0, a_0, r_1, ..., s_{T-1}, a_{T-1}, r_T)$;
    for $t = 0, 1, ..., T-1$ **do**:
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha[G_t - \hat{v}(s_t, \boldsymbol{w})]\nabla\hat{v}(s_t, \boldsymbol{w})$;
    end for
end for

Algorithm 17 Semi-gradient TD(0) Policy Evaluation

initialize the differentiable function $\hat{v}(s, \boldsymbol{w}) \in \mathbb{R}$ arbitrarily for all $s \in \mathcal{S}$;
input the policy π to be evaluated; the step size α;
for true **do**:
    Generate a start state s;
    while s is not terminal **do**:
        Choose action $a \leftarrow \pi(s)$;
        $r, s' \leftarrow environment(s, a)$;
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha[r + \gamma\hat{v}(s', \boldsymbol{w}) - \hat{v}(s, \boldsymbol{w})]\nabla\hat{v}(s, \boldsymbol{w})$;
        $s \leftarrow s'$;
    end while
end for

8.1.2 Semi-gradient Method for Control

The extension of the semi-gradient prediction methods to state-action values is straightforward. In this case it is the approximate action-value function, $\hat{q} \approx q^\pi$, that is represented as a parameterized functional

form with weight vector $\boldsymbol{w}$. The control version is derived from the *generalized policy iteration*. It quite like the way we make prediction:

- For MC, the target is return G_t

$$\Delta \boldsymbol{w} = \alpha(G_t - \hat{q}(s_t, a_t, \boldsymbol{w}))\nabla \hat{q}(s_t, a_t, \boldsymbol{w});$$

- For Sarsa, the target is TD target $r_{t+1} + \gamma \hat{q}(s_{t+1}, a_{t+1}, \boldsymbol{w})$:

$$\Delta \mathbf{w} = \alpha \left(r_{t+1} + \gamma \hat{q}\left(s_{t+1}, a_{t+1}, \mathbf{w}\right) - \hat{q}\left(s_t, a_t, \mathbf{w}\right)\right) \nabla_{\mathbf{w}} \hat{q}\left(s_t, a_t, \mathbf{w}\right);$$

- For Q-learning, the target is TD target $r_{t+1} + \gamma \max_a \hat{q}\left(s_{t+1}, a, \mathbf{w}\right)$:

$$\Delta \mathbf{w} = \alpha \left(r_{t+1} + \gamma \max_a \hat{q}\left(s_{t+1}, a, \mathbf{w}\right) - \hat{q}\left(s_t, a_t, \mathbf{w}\right)\right) \nabla_{\mathbf{w}} \hat{q}\left(s_t, a_t, \mathbf{w}\right).$$

One can get the corresponding control algorithm by modifying the update rule in the prediction version and then extracting the optimal policy according to $\hat{q}$, which is $\pi(s) = \arg\max_a \hat{q}(s, a, \boldsymbol{w})$.

Though TD control is powerful and practical in most case, there is no guarantees in the convergence of TD. Firstly, TD with VFA follows the *semi-gradient* rather than the true gradient of any objective function. Secondly, the updates involve doing an approximate Bellman backup (model-free) followed by fitting the underlying value function (approximation). Further, for off-policy, behavior policy and target policy are not identical, thus value function approximation may diverge.

The Deadly Triad for *Danger of Instability and Divergence*:

- Function approximation: A scalable way of generalizing from a state space much larger than the memory and computational resources;
- Bootstrapping: Update target that includes existing estimates (as in dynamic programming or TD methods) rather than relying exclusively on actual rewards and complete returns (as in MC methods);
- Off-policy learning: training on a distribution of transitions other than that produced by the target policy.

8.2 Deep Q-Learning

Deep Q-learning leverages nonlinear function approximator, deep neural networks, to avoid manual designing of features. Such method is also called DQN, and it is a well-known case of *deep reinforcement learning*.

Deep Reinforcement Learning:

- Frontier in machine learning and artificial intelligence;
- Deep neural networks can be used to represent value function, policy function, and even model;
- Optimize loss function by stochastic gradient descent.

Algorithm 18 Semi-gradient TD(0) Policy Evaluation

initialize the differentiable function $\hat{v}(s, \boldsymbol{w}) \in \mathbb{R}$ arbitrarily for all $s \in \mathcal{S}$;
input the policy π to be evaluated; the step size α;
for true **do**:
    Generate a start state s;
    while s is not terminal **do**:
        Choose action $a \leftarrow \pi(s)$;
        $r, s' \leftarrow environment(s, a)$;
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha[r + \gamma\hat{v}(s', \boldsymbol{w}) - \hat{v}(s, \boldsymbol{w})]\nabla\hat{v}(s, \boldsymbol{w})$;
        $s \leftarrow s'$;
    end while
end for

The success of DQN lies in two innovative techniques that greatly improve and stabilize the training procedure.

Experience Replay

- To reduce the correlations among samples, DQN stores transition (s_t, a_t, r_t, s_{t+1}) in a replay memory $\mathcal{D}$;
- To perform experience replay, DQN repeats the following
  - samples an experience tuple from the dataset:

$$(s, a, r, s') \sim \mathcal{D};$$

  - computes the target value with the sampled tuple: $r + \gamma \max_{a'} \hat{q}(s', a', \boldsymbol{w})$;
  - uses stochastic gradient descent to update the network weights

$$\Delta\boldsymbol{w} = \alpha(r + \gamma \max_{a'} \hat{q}(s', a', \boldsymbol{w}) - q(s, a, \boldsymbol{w}))\nabla_{\boldsymbol{w}}\hat{q}(s, a, \boldsymbol{w}).$$

Fixed Target

- To help improve stability, DQN fixes the weights used in the target calculation for multiple updates;
- Let a different set of parameter $\boldsymbol{w}^-$ be the set of weights used in the target, and $\boldsymbol{w}$ be the weights to be updated in each step;
- To perform experience replay with fixed target, DQN repeats the following
  - samples an experience tuple from the dataset: $(s, a, r, s') \sim \mathcal{D}$;
  - computes the target value for the sampled tuple:

$$r + \gamma \max_{a'} \hat{q}(s', a', \boldsymbol{w}^-);$$

  - uses stochastic gradient descent to update the network weights:

$$\Delta\boldsymbol{w} = \alpha(r + \gamma \max_{a'} \hat{q}(s', a', \boldsymbol{w}^-) - q(s, a, \boldsymbol{w}))\nabla_{\boldsymbol{w}}\hat{q}(s, a, \boldsymbol{w});$$

– updates $\boldsymbol{w}^- \leftarrow \boldsymbol{w}$ periodically.

Algorithm 19 Deep-Q Network

initialize the replay memory $\mathcal{D}$ with capacity N; initialize the state-action value function q with random weight $\boldsymbol{w}$; initialize the target state-action value function $\hat{q}$ with random weight $\boldsymbol{w}^- = \boldsymbol{w}$;
input the period C;
for true **do**:
    Generate a start state s;
    while s is not terminal **do**:
        Choose action $a = \begin{cases} \arg\max_{a' \in \mathcal{A}} q(s, a'), & \text{with probability } \varepsilon; \\ \text{a random action}, & \text{otherwise}; \end{cases}$
        $r, s' \leftarrow environment(s, a)$;
        Store transition (s, a, r, s') in $\mathcal{D}$;
        Sample random minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from $\mathcal{D}$;
        Set $y_j = \begin{cases} r_j, & \text{if } s_{j+1} \text{ is the terminal state}; \\ r_j + \gamma \max_{a'} \hat{q}(s_{j+1}, a', \boldsymbol{w}^-), & \text{otherwise}; \end{cases}$
        Perform a gradient descent step on $\left(y_j - q(s_j, a_j, \boldsymbol{w})\right)^2$ with respect to the parameter $\boldsymbol{w}$;
        Reset $\hat{q} = q$ every C steps;
    end while
end for