

Reinforcement Learning David Silver - Lecture 6 Notes: Value Function Approximation

Name: Eli Andrew

- Estimate value function with function approximation:

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

- Update parameter \mathbf{w} using MC and TD methods

- **Value Function Approximation with SGD**

- Goal: find parameter vector \mathbf{w} minimizing MSE between approximate value function $\hat{v}(s, \mathbf{w})$ and true value function $v_{\pi}(s)$

$$J(\mathbf{w}) = E_{\pi}[(v_{\pi}(S) - \hat{v}(S, \mathbf{w}))^2]$$

- Gradient descent finds local minimum:

$$\begin{aligned}\nabla \mathbf{w} &= -\frac{1}{2}\alpha \nabla_{\mathbf{w}} J(\mathbf{w}) \\ &= \alpha E_{\pi}[(v_{\pi}(S) - \hat{v}(S, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w})]\end{aligned}$$

- SGD *samples* the gradient (removes expectation):

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

- α gives step size: how much to adjust current estimate towards the true value
- $v_{\pi}(S) - \hat{v}(S, \mathbf{w})$ gives the total error of our current estimate
- $\nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w})$ gives how much each component of \mathbf{w} contributed to the overall error

- State representation is done using *feature vectors*

$$\mathbf{x}(S) = (\mathbf{x}_1(S), \dots, \mathbf{x}_n(S))$$

- **Linear Value Function Approximation**

- Represent value function by linear combination of features:

$$\hat{v}(S, \mathbf{w}) = \mathbf{x}(S)^{\top} \mathbf{w} = \sum_{j=1}^n \mathbf{x}_j(S) * \mathbf{w}_j$$

- Objective function is quadratic in parameters \mathbf{w} :

$$J(\mathbf{w}) = E_{\pi}[(v_{\pi}(S) - \mathbf{x}(S)^{\top} \mathbf{w})^2]$$

- SGD converges on global optimum using update rule:

$$\begin{aligned}\nabla_{\mathbf{w}}\hat{v}(S, \mathbf{w}) &= \mathbf{x}(S) \\ \Delta\mathbf{w} &= \alpha(v_{\pi}(S) - \hat{v}(S, \mathbf{w}))\mathbf{x}(S)\end{aligned}$$

- **Table lookup features** (special case of linear function approximation)

- Table lookup features just indicate if we're in a given state

$$x^{table}(S) = (\mathbf{1}(S = s_1), \dots, \mathbf{1}(S = s_n))$$

- Essentially a one-hot encoded vector with length equal to the number of states
- When we multiple this feature vector by our weights \mathbf{w} we are just selecting the weight at the same entry as our active state (s_n selects w_n)
- In the previous methods, this weight that we're selecting is the current estimate of the value of that state

- **Incremental Prediction Algorithms**

- There is no *true* value function ever actually available
- So, we must use our current target $v_{\pi}(s)$ instead
- This gives our $\Delta\mathbf{w}$ as:

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

- Where the $v_{target}(S)$ is defined by the algorithm we're using:
 - * Monte-Carlo: G_t
 - * TD(0): $R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w})$
 - * TD(λ): G_t^{λ}

- **Learning with Function Approximation**

- All algorithms (MC, TD, etc.) are used to generate training data that can then be used for updating our \mathbf{w} to better predict our $v_{target}(S)$
- Monte Carlo samples training data and performs updates through:

$$\begin{aligned}(S_1, G_1), (S_2, G_2), \dots, (S_T, G_T) \\ \Delta\mathbf{w} = \alpha(G_t - \hat{v}(S, \mathbf{w}))\mathbf{x}(S)\end{aligned}$$

- TD(0) does this through:

$$\begin{aligned}(S_1, R_2 + \gamma\hat{v}(S_2, \mathbf{w})), \dots, (S_{T-1}, R_T) \\ \Delta\mathbf{w} = \alpha(R_{t+1} + \gamma\hat{v}(S_{t+2}))\mathbf{x}(S) \\ = \alpha\delta\mathbf{x}(S)\end{aligned}$$

- And TD(λ) does this through:

$$\begin{aligned} & (S_1, G_1^\lambda), \dots, (S_{T-1}, G_{T-1}^\lambda) \\ \delta_t &= R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}) \\ E_t &= \gamma \lambda E_{t-1} + \mathbf{x}(S_t) \\ \Delta \mathbf{w} &= \alpha \delta_t E_t \end{aligned}$$

- The eligibility trace here is for features now (instead of states)
- In the incremental case we perform these updates to our weights on every single step of the episode (as determined by the algorithm we're using). For example, for Monte-Carlo it's one update at the end of every episode and for TD(0) its at every step.

• Control with Function Approximation

- Action-Value Function Approximation
 - * Approximate the action-value function

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

- * Minimize MSE between \hat{q} and q

$$J(w) = E_\pi[(q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}))^2]$$

- * Use SGD to find local minimum

$$\begin{aligned} -\frac{1}{2} \nabla J(\mathbf{w}) &= (q_\pi(S, A) - \hat{q}(S, A, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S, A, \mathbf{w}) \\ \Delta_{\mathbf{w}} &= \alpha (q_\pi(S, A) - \hat{q}(S, A, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S, A, \mathbf{w}) \end{aligned}$$

- * Feature vector \mathbf{x} is now a function of both S and A

• Convergence of Prediction Algorithms

Algorithm	Table Lookup	Linear	Non-Linear
On-policy MC	YES	YES	YES
On-policy TD(0)	YES	YES	NO
On-policy TD(λ)	YES	YES	NO
Off-policy MC	YES	YES	YES
Off-policy TD(0)	YES	NO	NO
Off-policy TD(λ)	YES	NO	NO

- Gradient TD is an algorithm that fixes these issues and converges for all 3 situations

- Gradient TD follows the true gradient of the projected Bellman error unlike TD
- **Convergence of Control Algorithms**

Algorithm	Table Lookup	Linear	Non-linear
MC Control	YES	(YES)	NO
Sarsa	YES	(YES)	NO
Q-learning	YES	NO	NO
Gradient Q-learning	YES	YES	NO

- **Batch Reinforcement Learning**

- Gradient descent is not sample efficient
- Batch methods seek to find the best fitting value function to the whole batch of training data
- **Least Squares Prediction**
 - * Use oracle data for value of each state $D = \{(s_1, v_1^\pi), \dots, (s_T, v_T^\pi)\}$
 - * Calculate least squares error as:

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

- **SGD with Experience Replay**

- * Given an experience of data in the same format as D
- * Repeat two steps:
- * (1) Sample state, value from experience

$$(s, v^\pi) \sim D$$

- * (2) Apply SGD update:

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

- * This removes the correlation of regular SGD where each update was sequentially related to one another
- * Another way to look at this is that we are storing our experienced data rather than throwing it away after each step
- * This eventually converges to least squares solution

- **Experience Replay in Deep-Q Networks**

- * Take action a_t 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 \mathbf{w}'
- * Optimize MSE between Q-network and Q-learning targets

$$L_i(w_i) = E_{s,a,r,s' \sim D}[(r + \gamma \max_{a'} Q(s', a'; \mathbf{w}'_i) - Q(s, a; \mathbf{w}_i))^2]$$

- * Using variant of SGD
- * This is stable vs. naive Q-learning because (1) it uses experience replay and (2) uses fixed Q-targets
- * Experience replay stabilizes function approx because it de-correlates the trajectories
- * The fixed Q-targets are obtained by having two separate networks where the “frozen” one (not being updated) is used as the target to bootstrap to