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

Name: Eli Andrew

- Estimate value function with function approximation:

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

- 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:

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

  - 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*

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

- **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:

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

- **Table lookup features** (special case of linear function approximation)
  - Table lookup features just indicate if we're in a given state

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

  - 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:

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

  - TD(0) does this through:

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

- And TD(λ) does this through:

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

- 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 epsiode (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

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

    - 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 $D = \{(s_1, v_1^\pi), \ldots, (s_T, v_T^\pi)\}$
    - 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 stablizes function approx because it de-corelates 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

- **On Policy Distribution**
  - A distribution of time spent in each state (normalized to sum to one) can be calculated as

$$\mu(s) = \frac{\eta(s)}{\sum_{s'} \eta(s')}, s \in S$$

  - Where $\eta(s)$ denotes the number of time steps spent, on average, in state s in a single episode

- **Coarse Coding**
  - Cover state space in overlapping spaces
  - Features are then just binary (or non-binary) values indicating if the agent is in a particular space
  - Since the spaces are overlapping, the features may have more than a single positive value
  - The larger each individual space is then larger the update area will be
  - While the size of each individual space causes generalizations to be larger/smaller, the final function learned is only slightly affected by the size of the spaces.

- **Tile Coding**
  - Form of coarse coding for multi-dimensional continuous spaces
  - Uses multiple overlapping grid-tilings that are offset from one another by a fraction of tile width in each dimension
  - Each state then falls into exactly one tile of each tiling

  - Using asymmetric tiling offsets allows for better feature generalization
  - Within small squares of size $\frac{w}{n}$ where w is tile width and n is number of tilings, all states activate the same tiles, have the same feature representation, and the same approximated value
  - Often makes sense to use different shape tiles for different tilings

- **Radial Basis Function**
  - Generalization of coarse coding to continuous valued features
  - Typical feature x_i has Gaussian response $x_i(s)$ dependent only on the distance between the state, s, and the feature's prototypical or center state, c_i, relative to the feature's width, σ_i

$$x_i(s) = \exp(-\frac{||s - c_i||^2}{2\sigma_i^2})$$

- Setting α to $(\tau E[x^\top x])^{-1}$ is a good rule of thumb
- Kernel trick
- **Interest and Emphasis**
  - On-policy distribution is distribution of states encountered while following the target policy
  - **Interest**: non-negative scaler measure I_t that indicates the degree to which we are interested in accurately valuing the state (or state-action pair) at time t
  - Distribution μ following target-policy is then weighted by I
  - **Emphasis**: scaler M_t that multiplies the learning update therefore emphasizing or de-emphasizing updates at time t
  - Emphasis is determined recursively from the interest by:

$$M_t = I_t + \gamma^n M_{t-n}$$

- Average reward defined as:

$$r(\pi) = \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)$$

- In average reward setting, the returns are defined in terms of differences between rewards and the average reward (this is called *differential return*):

$$G_t = R_{t+1} - r(\pi) + R_{t+2} - r(\pi) + R_{t+3} - r(\pi) + \dots$$