[Notes] Reinforcement Learning Specialization @University of Alberta

Xu, Bangyao
bangyaoxu0321@gmail.com

July 12, 2021

1 Fundamentals of Reinforcement Learning

1.1 An Introduction to Sequential Decision-Making

- In reinforcement learning (RL), the *agent* generates its own training data by interacting with the world
- The agent must learn the consequences of his own actions through trial and error, rather than being told the correct action
- In the *k-armed bandit problem*, we have an agent who chooses between *k actions* and receives a *rewards* based on the action it chooses
- *Expected reward*: $q^*(a) = \mathbb{E}[R_t|A_t = a], \forall a \in \{1, ..., k\} \Rightarrow \sum_r \mathbb{P}(r|a)$
- The goal is to maximize the expected reward $\Rightarrow \text{argmax}_a\, q^*(a)$
- Fundamentals ideas behind RL: actions, rewards, value functions
- Sample-Average method to estimate $q^*(a)$:
- $Q_t(a) = \frac{\text{sum of rewards when } a \text{ taken prior to } t}{\text{number of times } a \text{ taken prior to } t} = \frac{\sum_{i=1}^{t-1} R_i}{t-1}$
- The *greedy action* is the action that currently has the largest estimated value $\Rightarrow$ selecting the greedy action means the agent is *exploiting* its current knowledge
- The agent may choose to *explore* by choosing a non-greedy action $\Rightarrow$ sacrifice immediate reward hoping to gain more information about the other actions

- The *exploration-exploitation dilemma*: the agent can not choose to both explore and exploit at the same time $\Rightarrow$ one of the fundamental problems in RL

- Incremental update rule: New Estimate = Old Estimate + Step Size $\times$ (Target - Old Estimate)$\Rightarrow Q_{n+1} = Q_n + \alpha_n(R_n - Q_n), \alpha_n \in [0, 1]$

- Non-stationary bandit problem $\Rightarrow$ decaying past rewards by a constant step size: $Q_{n+1} = (1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} R_i$

- Exploration $\Rightarrow$ improve knowledge for long-term benefit

- Exploitation $\Rightarrow$ exploit knowledge for short-term benefit

- Epsilon-greedy action selection:

$$A_t = \begin{cases} \operatorname{argmax}_a Q_t(a), & \text{with probability } 1-\epsilon \\ a \sim Uniform(\{a_1, ..., a_k\}), & \text{with probability } \epsilon \end{cases}$$

- *Optimistic initial values* encourage exploration early in learning

- Limitations of optimistic initial values: 1) only drive early exploration; 2) not well-suited for non-stationary problems; 3) we may not always know how to set the optimistic initial values

- Upper-Confidence Bound (UCB) action selection:

- $A_t = \text{argmax}\left[Q_t(a) + c\sqrt{\frac{\ln(t)}{N_t(a)}}\right]$, where t is time steps, $N_t(a)$ is times action a taken, and c is user defined parameter

1.2 Markov Decision Processes

- Two aspects of real-world problems: 1) different situations call for different responses; 2) actions we choose now affect the amount of reward we can get into the future $\Rightarrow$ *Markov decision processes* (MDPs)

- In the MDP framework, the agent environment interaction generates a trajectory of experience consisting of states ($S_{t+1} \in \mathcal{S}$), actions ($A_t \in \mathcal{A}(S_t)$), and rewards ($R_{t+1} \in \mathcal{R}$)

- Actions influence immediate rewards as well as future states and through those, future rewards

- The dynamics of an MDP: $\mathbb{P}(s^{'}, r|s, a)$, where $\mathbb{P} : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ and $\sum_{s^{'} \in \mathcal{S}} \sum_{r \in \mathcal{R}} \mathbb{P}(s^{'}, r|s, a) = 1, \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$
- The present state contains all information necessary to predict future
- MDP formalism is abstract and flexible (high-level vs. low-level)
- Goal of an agent: $G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T \Rightarrow$ maximize the expected return $\mathbb{E}[G_t]$
- Episode tasks: 1) Each episode begins independently of how the previous one ended; 2) At termination, the agent is reset to a start state; 3) Every episode has a final state called the *terminal state*
- Continuing tasks: 1) Cannot be broken up into independent episodes; 2) The interaction goes on continually; 3) There are no terminal states
- Discounting: $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... + \gamma^{k-1} R_{t+k} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$, where $\gamma \in [0, 1) \Rightarrow$ make sure G_t is finite
- Effect if γ on agent behavior:
- $\gamma \rightarrow 0 \Rightarrow$ short-sighted agent; $\gamma \rightarrow 1 \Rightarrow$ far-sighted agent
- Recursive nature of returns: $G_t = R_{t+1} + \gamma G_{t+1}$

1.3 Value Functions & Bellman Equations

- A *policy* maps the current state onto a set of probabilities for taking each action
- Deterministic policy notation: $\pi(s) = a$ represents action selected in state s by the policy π
- The agent can select the same action in multiple states, and some actions might not be selected in any state
- Stochastic policy notation: $\pi(a|s) \geq 0$, where $\sum_{a \in \mathcal{A}(s)} \pi(a|s) = 1$
- The policies depend only on the current state, not on other things
- In MDPs, state includes all information required for decision-making
- State-value functions: $v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$, where $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$
- Action-value functions: $q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$

- Value functions predict rewards into the future $\Rightarrow$ judge the quality of different policies
- Bellman Equation $\Rightarrow$ formalize the connection between the value of a state and its possible successors
- State-value Bellman equation:

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[G_t | S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r | s, a) \Big[r + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s'] \Big] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r | s, a) \Big[r + \gamma v_\pi(s') \Big]
\end{aligned}$$

- Action-value Bellman equation:

$$\begin{aligned}
q_\pi(s, a) &= \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \\
&= \sum_{s'} \sum_r p(s', r | s, a) \Big[r + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s'] \Big] \\
&= \sum_{s'} \sum_r p(s', r | s, a) \Big[r + \gamma \sum_{a'} \pi(a'|s') \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s', A_{t+1} = a'] \Big] \\
&= \sum_{s'} \sum_r p(s', r | s, a) \Big[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a') \Big]
\end{aligned}$$

- Bellman equation reduced an unmanageable infinite sum over possible futures, to a simple linear algebra problem
- An *optimal policy* π_* is as good as or better than all the other policies
- There must always exist at least one optimal deterministic policy
- Only for deterministic policies, the number of possible policies is equal to the number of possible actions to the power of the number of states $\Rightarrow$ brute-force search is intractable
- $\pi_1 \geq \pi_2$ if and only if $v_{\pi_1}(s) \geq v_{\pi_2}(s) \quad \forall s \in \mathcal{S}$

- $v_{\pi_*}(s) = \mathbb{E}_{\pi_*}[G_t|S_t = s] = \max_\pi v_\pi(s) \quad \forall s \in \mathcal{S} \qquad$ (Equation v_*)
- $q_{\pi_*}(s,a) = \max_\pi q_\pi(s,a) \quad \forall s \in \mathcal{S}, a \in \mathcal{A} \qquad$ (Equation q_*)
- Bellman Optimal Equation for v_*:

$$\begin{aligned} v_*(s) &= \sum_a \pi_*(a|s) \sum_{s'} \sum_r p(s',r|s,a)\Big[r + \gamma v_*(s')\Big] \\ &= \max_a \sum_{s'} \sum_r p(s',r|s,a)\Big[r + \gamma v_*(s')\Big] \end{aligned}$$

- Bellman Optimal Equation for q_*:

$$\begin{aligned} q_*(s,a) &= \sum_{s'} \sum_r p(s',r|s,a)\Big[r + \gamma \sum_{a'} \pi_*(a'|s') q_*(s',a')\Big] \\ &= \sum_{s'} \sum_r p(s',r|s,a)\Big[r + \gamma \max_{a'} q_*(s',a')\Big] \end{aligned}$$

- Determining an Optimal Policy:
- $\pi_*(s) = \text{argmax}_a \sum_{s'} \sum_r p(s',r|s,a)\Big[r + \gamma v_*(s')\Big]$;
- $\pi_*(s) = \text{argmax}_a q_*(s,a)$

1.4 Dynamic Programming

- *Dynamic programming* algorithms use the Bellman equations to define iterative algorithms for both policy evaluation and control
- The goal of the control task is to modify a policy to produce a new one which is strictly better, moreover, to improve the policy repeatedly to obtain a sequence of better and better policies
- Iterative policy evaluation:
- $v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s',r|s,a)\Big[r + \gamma v_k(s')\Big]$;
- If $v_{k+1} = v_k$, $\forall s \in \mathcal{S}$, then $v_k = v_\pi$ since v_π is the unique solution to the Bellman equation, and for any v_0, $\lim_{k \to \infty} v_k = v_\pi$
- Policy improvement theorem:

- $q_\pi(s, \pi^{'}(s)) \geq q_\pi(s, \pi(s))$ for all $s \in \mathcal{S} \Rightarrow \pi^{'} \geq \pi$;
- $q_\pi(s, \pi^{'}(s)) > q_\pi(s, \pi(s))$ for at least one $s \in \mathcal{S} \Rightarrow \pi^{'} > \pi$
- The policy improvement theorem only guarantees that the new policy is an improvement on the original
- Policy iteration = Evaluation + Improvement:
- Evaluation: $V \rightarrow v_\pi$; Improvement: $\pi \rightarrow greedy(V)$; until $\pi_* \leftrightarrow v_*$
- Generalized policy iteration ⇒ value iteration: perform *just one sweep* over all the states and greedify with respect to current value function
- *Synchronous* DP algorithms perform systematic sweeps
- *Asynchronous* DP algorithms update the values of states in any order
- In order to guarantee convergence, asynchronous algorithms must continue to update the values of all states
- Asynchronous algorithms can propagate value information quickly through selective updates ⇒ sometimes more efficient than a systematic sweep
- Monte Carlo ⇒ gather a large number of returns under pi and take their average, eventually this will converge to the state value
- Bootstrapping ⇒ using the value estimates of successor states to improve our current value estimate
- Brute-Force ⇒ number of deterministic policies can be huge: $|\mathcal{A}|^{|\mathcal{S}|}$
- Policy iteration is guaranteed to find the optimal policy in polynomial time in $|\mathcal{S}|$ and $|\mathcal{A}|$ ⇒ exponentially faster than Brute-Force
- The curse of dimensionality ⇒ the size of the states space grows *exponentially* as the number of relevant features increases

2 Sample-based Learning Methods

2.1 Monte Carlo Methods for Prediction & Control

- To use a pure dynamic programming approach, the agent needs to know the environment's transition probabilities

- Monte Carlo (MC) doesn't need a model of the environment dynamics
- The MC estimates the value of individual state *independently* of the values of other states
- The computation MC needed to update the value of each state does not depend on the size of the MDP, but the length of the episode
- One way to maintain exploration is called *exploring starts* $\Rightarrow$ guarantee the episodes start in every state-action pair
- Monte Carlo Generalized Policy Iteration (GPI):
- Improvement: $\pi_{k+1}(s) = \text{argmax}_a\, q_{\pi_k}(s, a)$;
- Evaluation: Monte Carlo prediction
- The ϵ-Soft policies take each action with probability at least $\frac{\epsilon}{|\mathcal{A}|}$
- ϵ-Soft policies may not be optimal $\Rightarrow$ the optimal ϵ-Soft policy
- On-Policy $\Rightarrow$ improve & evaluate the policy being used to select actions
- Off-Policy $\Rightarrow$ improve & evaluate a *different* policy from the one used to select actions
- Target policy $\pi(a|s) \Rightarrow$ learn values for this policy (e.g. optimal policy)
- Behavior policy $b(a|s) \Rightarrow$ select actions from this policy (exploratory)
- $b(a|s)$ must cover $\pi(a|s) \Rightarrow \pi(a|s) > 0$ where $b(a|s) > 0$
- On-Policy $\Rightarrow \pi(a|s) = b(a|s)$
- Importance sampling $\Rightarrow$ estimate expected value of a distribution using samples from a different distribution: sample $x \sim b$; estimate $\mathbb{E}_\pi[X]$
- Derivation of importance sampling:

$$\begin{aligned}
\mathbb{E}_\pi[X] &= \sum_{x \in X} x\pi(x) \\
&= \sum_{x \in X} x \frac{\pi(x)}{b(x)} b(x) = \sum_{x \in X} x\rho(x)b(x) \\
&= \mathbb{E}_b[X\rho(X)] \\
&\approx \frac{1}{n}\sum_{i=1}^{n} x_i\rho(x_i),\ x_i \sim b,\ \rho \text{ is called importance sampling ratio}
\end{aligned}$$

- Off-Policy Monte Carlo $\Rightarrow V_\pi(s) = \mathbb{E}_b[\rho G_t | S_t = s]$:
- $\rho_{t:T-1} = \frac{\mathbb{P}(\text{trajectory under } \pi)}{\mathbb{P}(\text{trajectory under } b)} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)p(S_{k+1}|S_k,A_k)}{b(A_k|S_k)p(S_{k+1}|S_k,A_k)} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}$
- ρ is computed incrementally: $\rho_{t:T-1} = \rho_t \rho_{t+1} \rho_{t+2} \dots \rho_{T-2} \rho_{T-1}$

2.2 Temporal Difference Learning Methods for Prediction

- Recursive value function ($G_t = R_{t+1} + \gamma G_{t+1}$):

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t | S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] \\ &= R_{t+1} + \gamma v_\pi(S_{t+1}) \end{aligned}$$

- Temporal Difference ($G_t \approx R_{t+1} + \gamma V(S_{t+1})$):
- $V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)] = V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$
- TD error: $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$
- TD updates the value of one state towards its own estimate of the value in the next state
- TD(0) algorithm $\Rightarrow$ update the values with the TD learning rule on each step of the episode (only need to keep track of the previous state)
- TD elegantly combines key ideas from dynamic programming (bootstrap) and Monte Carlo methods (learn directly from experience)
- TD asymptotically converges to the correct predictions, and usually converges faster than Monte Carlo methods
- Comparing TD and Monte Carlo:
- 1) TD agent makes updates to the values on every step vs. Monte Carlo agent only updates at the end of each episode;
- 2) TD converges faster than Monte Carlo and achieves better error

2.3 Temporal Difference Learning Methods for Control

- GPI with Monte Carlo evaluates and improves after each episode $\Rightarrow$ improve the policy after just one policy evaluation step with TD
- Sarsa makes predictions about the values of state action pairs:
- $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$
- Q-learning uses the Bellman's Optimality Equation for action values:
- $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a^{'}) - Q(S_t, A_t))$
- Sarsa is sample-based version of policy iteration which uses Bellman equations for action values, that each depend on a fixed policy
- Q-learning is a sample-based version of value iteration which iteratively applies the Bellman optimality equation
- Q-learning converges to the optimal value function as long as the aging continues to explore and samples all areas of the state action space
- In reinforcement learning, α, ϵ, initial values, and the length of the experiment can all influence the final result
- In Sarsa, the agent bootstraps off of the value of the action it's going to take next, which is sampled from its behavior policy $\Rightarrow$ on-policy
- Q-learning bootstraps off of the largest action value in its next state, which is like sampling an action under an estimate of the optimal policy rather than the behavior policy $\Rightarrow$ off-policy
- Q-learning learns about the best action it could possibly take rather than the actions it actually takes
- The Q-learning agent is estimating action values with unknown policy $\Rightarrow$ does not need important sampling ratios to correct for the difference in action selection
- The Expected Sarsa explicitly computes expectation over next actions:
- $Q(S_t, A_t) \leftarrow Q(S_t, A_t)+\alpha(R_{t+1}+\gamma \sum_{a'} \pi(a^{'}|S_{t+1})Q(S_{t+1}, a^{'})-Q(S_t, A_t))$
- Expected Sarsa's updates are deterministic for a given state and action vs. Sarsa's updates can be significantly depending on next action

- Expected Sarsa is more robust than Sarsa to large step sizes
- Q-Learning is a special case of Expected Sarsa:
- $\sum_{a'} \pi(a^{'}|S_{t+1})Q(S_{t+1}, a^{'}) = \max_{a'} Q(S_{t+1}, a^{'})$

2.4 Planning, Learning & Acting

- Planning refers to the process of using a model to improve a policy $\Rightarrow$ use simulated experience and perform value function updates
- *Sample model* produces an actual outcome drawn from some underlying probabilities (computationally inexpensive)
- *Distribution model* which completely specifies the likelihood or probability of every outcome (difficult to specify & can become very large)
- Sample models require less memory vs. Distribution models can be used to compute the exact expected outcome
- *Planning* $\Rightarrow$ leverage a model to better inform decision-making without having to interact with the world
- Random-sample one-step tabular Q-planning:
- 1) choose a state-action pair at random from set of all states & actions
- 2) query the sample model with this state action pair to produce a sample of the next state and reward
- 3) perform a Q-Learning update on this model transition
- 4) improve policy by beautifying with respect to updated action values
- Dyna architecture = Q-learning (performs updates using environment experience) + Q-planning (performs updates using simulated experience from the model)
- Dyna-Q performs many planning updates for each environment transition and makes better use of its limited interaction with environment
- For the same number of environment interactions, Dyna-Q can learn a lot more $\Rightarrow$ planning makes better use of environment experience if the model is correct

- Models are inaccurate when transitions they store are different from transitions that happen in the environment $\Rightarrow$ 1) incomplete model; 2) changing environment
- Dyna-Q can plan with an incomplete model by only sampling state action pairs that had been previously visited
- In general, an agent might want to double-check that all its models transitions are correct $\Rightarrow$ double-checking transitions with low valued actions will often lead to low reward
- The trade-off for the agent: *explore* to make sure it's model is accurate vs. *exploit* the model to compute the optimal policy assuming the model is correct
- Bonus reward for exploration: New reward $= r + \kappa\sqrt{\tau}$, where τ is the time steps since transition was last tried
- Dyna-Q algorithm + reward bonus $\Rightarrow$ Dyna-Q+ algorithm

3 Prediction and Control with Function Approximation

3.1 On-policy Prediction with Approximation

- Parameterizing the value function: $\hat{v}(s, \mathbf{w}) \approx v_\pi(s) \Rightarrow$ modify the weights, instead of the individual state values
- Linear value function approximation: $\hat{v}(s, \mathbf{w}) \approx \sum w_i x_i(s) =< \mathbf{w}, \mathbf{x}(s) >$
- The tabular value functions are special cases of linear value function approximations $\Rightarrow$ indicator functions for weights
- *Generalization* $\Rightarrow$ Updates to one state affect the value of other states
- *Discrimination* $\Rightarrow$ The ability to make the value of two states different
- The tabular representations have provided good discrimination, but no generalization
- Framing policy evaluation as supervised learning: $s \Rightarrow v_\pi(s)$
- The function approximator should be compatible with:

- 1) Online updating; 2) Bootstrapping (e.g. TD)
- The mean squared value error: $\overline{VE} = \sum_s \mu(s)[v_\pi(s) - \hat{v}(s, \mathbf{w})]^2$, where μ should be the fraction of time spent in s when following policy π
- Gradient descent: $\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \nabla J(\mathbf{w}_t) \Rightarrow$ Global minimum
- $\hat{v}(s, \mathbf{w}) =< \mathbf{w}, \mathbf{x}(s) > \Rightarrow \nabla \hat{v}(s, \mathbf{w}) = \mathbf{x}(s)$
- Stochastic gradient descent $\Rightarrow$ efficiently update the weights on every step by sampling the gradient
- Gradient Monte Carlo: $\mathbf{w} \leftarrow \mathbf{w} + \alpha[G_t - \hat{v}(S_t, \mathbf{w})]\nabla \hat{v}(S_t, \mathbf{w})$
- *State aggregation* treats certain states as the same $\Rightarrow$ another example of linear function approximation
- TD update for function approximation: $\mathbf{w} \leftarrow \mathbf{w} + \alpha[U_t - \hat{v}(S_t, \mathbf{w})]\nabla \hat{v}(S_t, \mathbf{w})$, where $U_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) \Rightarrow U_t$ is biased since $\mathbf{w}$ may not converge to a local optimum
- TD is a semi-gradient method $\Rightarrow \nabla U_t = \nabla(R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})) \neq 0$
- TD can learn during the episode and has lower variance updates $\Rightarrow$ TD often learns faster than Monte Carlo
- TD update with linear function approximation: $\mathbf{w} \leftarrow \mathbf{w} + \alpha \delta_t \mathbf{x}(S_t)$, where $\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})$
- With good features, linear methods can learn quickly and achieve good prediction accuracy $\Rightarrow$ expert knowledge to design good features
- The expected TD update: $\mathbb{E}[\Delta \mathbf{w}_t] = \alpha(\mathbf{b} - \mathbf{A}\mathbf{w}_t)$, where $\mathbf{b} = \mathbb{E}[R_{t+1}\mathbf{x}_t]$ and $\mathbf{A} = \mathbb{E}[\mathbf{x}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^T]$
- The TD fixed point: $\mathbb{E}[\Delta \mathbf{w}_{TD}] = \alpha(\mathbf{b} - \mathbf{A}\mathbf{w}_{TD}) = 0 \Rightarrow \mathbf{w}_{TD} = \mathbf{A}^{-1}\mathbf{b}$
- $\overline{VE}(\mathbf{w}_{TD}) \leq \frac{1}{1-\gamma} \min_w \overline{VE}(\mathbf{w})$
- TD does converge to the minimum of a principled objective, based on Bellman equations
- If γ is very close to zero, the TD fixed point is very close to the minimum value error solution

3.2 Constructing Features for Prediction

- State aggregation $\Rightarrow$ obtain a more flexible class of feature representations by allowing overlap $\Rightarrow$ Coarse coding
- Coarse coding can also be applied to higher dimensional inputs
- The shape and size of the receptive fields impact the *broadness* and *direction* of generalization and so the speed of learning
- The ability to distinguish between values for two different states is called *discrimination*
- The size, number, and shape of the features all affect the discriminative ability of the representation
- Each task requires different feature properties $\Rightarrow$ no general solution
- Coarse coding using overlapping grids $\Rightarrow$ Tile coding
- The feature vectors produced by tile coding may query in the value function cheap computationally
- Both of neural network and tile coding use prior knowledge to help in constructing features
- The neural network can use data to improve the features, whereas the tile coder cannot incorporate new information from data
- The depth allows *composition* of features $\Rightarrow$ Composition can produce more specialized features by combining modular components
- Depth can also be helpful for obtaining *abstractions*
- The choice of starting point can play a big role in the performance of the neural network $\Rightarrow$ randomly sample the initial weights from a normal distribution with small variance: $\mathbf{w}_{init} = \frac{\mathcal{N}(0,1)}{\sqrt{n_{input}}}$
- Update momentum:
- 1) $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \alpha \nabla_{\mathbf{w}} L(\mathbf{w}_t) + \lambda \mathbf{M}_t$;
- 2) $\mathbf{M}_{t+1} \leftarrow \lambda \mathbf{M}_t - \alpha \nabla_{\mathbf{w}} L$
- Adapting the step sizes for each weight, based on statistics about the learning process in practice results in much better performance $\Rightarrow$ Each dimension of the gradient is scaled by its corresponding step size instead of the global step size

3.3 Control with Approximation

- Stacking features $\Rightarrow$ use the same state features for each action, but only activate the features corresponding to that action
- Parameterized action value functions for the action value estimates $\Rightarrow$ Episodic Sarsa with function approximation
- Sarsa: $\mathbf{w} \leftarrow \mathbf{w}+\alpha(R_{t+1}+\gamma\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w})-\hat{q}(S_t, A_t, \mathbf{w}))\nabla\hat{q}(S_t, A_t, \mathbf{w})$
- Expected Sarsa: $\mathbf{w} \leftarrow \mathbf{w} + \alpha(R_{t+1} + \gamma \sum_{a'} \pi(a^{'}|S_{t+1})\hat{q}(S_{t+1}, a^{'}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w}))\nabla\hat{q}(S_t, A_t, \mathbf{w})$
- Q-learning: $\mathbf{w} \leftarrow \mathbf{w}+\alpha(R_{t+1}+\gamma \max_{a'} \hat{q}(S_{t+1}, a^{'}, \mathbf{w})-\hat{q}(S_t, A_t, \mathbf{w}))\nabla\hat{q}(S_t, A_t, \mathbf{w})$
- $q_\pi(s, a) \approx \hat{q}(s, a, \mathbf{w}) = \mathbf{w}^T\mathbf{x}(s, a)$
- To facilitate systematic exploration, changes to the value function need to be more localized
- Epsilon greedy is generally applicable and easy to use even in cases with non-linear function approximation
- The average reward objective: $r(\pi) = \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s^{'},r} p(s^{'}, r|s, a)r$
- The average reward puts preference on the policy that receives more reward in total without having to consider larger and larger discounts
- Differential return: $G_t = R_{t+1} - r(\pi) + R_{t+2} - r(\pi) + R_{t+3} - r(\pi) + ...$ $\Rightarrow$ represents how much more reward the agent will receive from the current state in action compared to the average reward of the policy
- The differential return is only a convergent sum if the subtracted constant is equal to the true average reward $\Rightarrow$ If a lower or higher number is subtracted, the sum will diverge to positive or negative infinity
- Value functions for average reward:

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \\ &= \sum_{s^{'}}\sum_{r} p(s^{'}, r|s, a)\Big[r - r(\pi) + \sum_{a^{'}} \pi(a^{'}|s^{'})q_\pi(s^{'}, a^{'})\Big] \end{aligned}$$

3.4 Policy Gradient

- Policy parameterization $\Rightarrow \pi(a|s,\theta)$:
- $\pi(a|s,\theta) \geq 0$ and $\sum_a \pi(a|s,\theta) = 1$, for $a \in \mathcal{A}, s \in \mathcal{S}$
- Softmax policy parameterization: $\pi(a|s,\theta) = \frac{e^{h(s,a,\theta)}}{\sum_{b\in\mathcal{A}} e^{h(s,b,\theta)}}$
- The action preference, $h(s,a,\theta)$, can be parameterized in any way since the softmax will enforce the constraints of a probability distribution
- Only the relative differences between preferences are important
- Advantages of policy parameterization:
- 1) autonomously decrease exploration over time
- 2) avoid failures due to deterministic policies with limited function approximation
- 3) sometimes the policy is less complicated than the value function
- $r(\pi) = \sum_s \mu(s) \sum_a \pi(a|s,\theta) \sum_{s',r} p(s', r|s,a) r \Rightarrow$ the expected reward across states is a sum over s of the expected reward in a state weighted by $\mu(s) \Rightarrow$ average reward learning objective
- The policy gradient theorem: $\nabla r(\pi) = \sum_s \mu(s) \sum_a \nabla\pi(a|s,\theta) q_\pi(s,a)$
- Stochastic samples of the gradient: $\theta_{t+1} = \theta_t + \alpha \sum_a \nabla\pi(a|S_t,\theta_t) q_\pi(S_t,a)$
- Unbiasedness of the stochastic samples $\Rightarrow$ get stochastic sample with one action: $\theta_{t+1} = \theta_t + \alpha \frac{\nabla\pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)} q_\pi(S_t,A_t) = \theta_t + \alpha\nabla\ln(\pi(A_t|S_t,\theta_t) q_\pi(S_t,A_t)$
- Actor-Critic algorithm:
- Parameterized policy $\Rightarrow$ *actor*; Value function $\Rightarrow$ *critic*
- Approximating the action value:
- $\theta_{t+1} = \theta_t + \alpha\nabla\ln(\pi(A_t|S_t,\theta_t)[R_{t+1} - \bar{R} + \hat{v}(S_{t+1},\mathbf{w})]$
- Subtracting the current state's value estimates:

$$\begin{aligned}\theta_{t+1} &= \theta_t + \alpha\nabla\ln(\pi(A_t|S_t,\theta_t)[R_{t+1} - \bar{R} + \hat{v}(S_{t+1},\mathbf{w}) - \hat{v}(S_t,\mathbf{w})] \\ &= \theta_t + \alpha\nabla\ln(\pi(A_t|S_t,\theta_t)\delta_t\end{aligned}$$

- The *actor* is continually changing the policy to exceed the critics expectation, and the *critic* is constantly updating its value function to evaluate the actors changing policy:
- $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S, \mathbf{w})$ and $\theta \leftarrow \theta + \alpha^{\theta} \delta \nabla \ln(\pi(A|S, \theta)$
- Policy update with a softmax policy: $\pi(a|s, \theta) = \frac{e^{h(s,a,\theta)}}{\sum_{b \in \mathcal{A}} e^{h(s,b,\theta)}}$
- Stacked state features $\Rightarrow$ a copy of state feature vector for each action
- Parameterization and features: $\hat{v}(s, \mathbf{w}) = \mathbf{w}^T \mathbf{x}(s)$; $h(s, a, \theta) = \theta^T \mathbf{x}_h(s, a)$
- $\nabla \hat{v}(s, \mathbf{w}) = \mathbf{x}(s)$ and $\nabla \ln(\pi(a|s, \theta) = \mathbf{x}_h(s, a) - \sum_b \pi(b|s, \theta) \mathbf{x}_h(s, b)$
- Gaussian policy: $\pi(a|s, \theta) = \frac{1}{\sigma(s,\theta)\sqrt{2\pi}} \exp\Big(- \frac{(a - \mu(s,\theta))^2}{2\sigma(s,\theta)^2} \Big)$, where:
- $\mu(s, \theta) = \theta_\mu^T \mathbf{x}(s)$ and $\sigma(s, \theta) = \exp\big(\theta_\sigma^T \mathbf{x}(s)\big)$
- Gradient of the Log of the Gaussian policy:
- $\nabla \ln(\pi(a|s, \theta_\mu)) = \frac{1}{\sigma(s,\theta)^2}(a - \mu(s, \theta))\mathbf{x}(s)$
- $\nabla \ln(\pi(a|s, \theta_\sigma)) = \Big(\frac{(a - \mu(s,\theta))^2}{\sigma(s,\theta)^2} - 1 \Big) \mathbf{x}(s)$
- It might not be straightforward to choose a discrete set of actions
- Continuous actions allow us to generalize over actions

■