Outline

- Defining an RL problem
  - Markov Decision Processes

- Solving an RL problem
  - Dynamic Programming
  - Monte Carlo methods
  - Temporal-Difference learning

- Miscellaneous
  - state representation
  - function approximation
  - rewards
Markov Decision Process (MDP)

- set of states $S$, set of actions $A$, initial state $S_0$
- transition model $P(s,a,s')$
  - $P(\{1,1\}, \text{up}, \{1,2\}) = 0.8$
- reward function $r(s)$
  - $r(\{4,3\}) = +1$
- goal: maximize cumulative reward in the long run
- policy: mapping from $S$ to $A$
  - $\pi(s)$ or $\pi(s,a)$

- reinforcement learning
  - transitions and rewards usually not available
  - how to change the policy based on experience
  - how to explore the environment

Dynamic programming

- Main idea
  - use value functions to structure the search for good policies
  - need a perfect model of the environment

- Two main components
  - policy evaluation: compute $V^\pi$ from $\pi$
  - policy improvement: improve $\pi$ based on $V^\pi$

- start with an arbitrary policy
- repeat evaluation/improvement until convergence
Policy/Value iteration

- **Policy iteration**
  \[ \pi_0 \rightarrow E V^{\pi_0} \rightarrow I \pi_1 \rightarrow E V^{\pi_1} \rightarrow I \pi^* \rightarrow E V^* \]
  - two nested iterations; too slow
  - don’t need to converge to \( V^* \)
    - just move towards it

- **Value iteration**
  \[ V_{k+1}(s) = \max_a \sum_{s'} P_{ss'}^a \left[ r_{ss'}^a + \gamma V_k(s') \right] \]
  - use Bellman optimality equation as an update
  - converges to \( V^* \)

Using DP

- need complete model of the environment and rewards
  - robot in a room
    - state space, action space, transition model

- can we use DP to solve
  - robot in a room?
  - back gammon?
  - helicopter?

- DP bootstraps
  - updates estimates on the basis of other estimates
Monte Carlo methods

- don’t need full knowledge of environment
  - just experience, or
  - simulated experience

- but similar to DP
  - policy evaluation, policy improvement

- averaging sample returns
  - defined only for episodic tasks
  - episodic (vs. continuing) tasks
    - “game over” after N steps
    - optimal policy depends on N; harder to analyze

Monte Carlo policy evaluation

- Want to estimate $V^\pi(s)$
  - expected return starting from $s$ and following $\pi$
  - estimate as average of observed returns in state $s$

- First-visit MC
  - average returns following the first visit to state $s$

\[
V^\pi(s) = \frac{1}{4} (2 + 1 - 5 + 4) = 0.5
\]
Monte Carlo control

- $V^\pi$ not enough for policy improvement
  - need exact model of environment
  \[
  \pi^*(s) = \arg\max_{\pi \in \Pi} \sum_{s' \in S} P_{ss'}V^\pi(s')
  \]
- Estimate $Q^\pi(s,a)$
  \[
  \pi'(s) = \arg\max_{\pi} Q^\pi(s, a)
  \]
- MC control
  \[
  \pi_0 \rightarrow E Q_{\pi_0} \rightarrow I \pi_1 \rightarrow E Q_{\pi_1} \rightarrow I \ldots \rightarrow I \pi^* \rightarrow E Q^*
  \]
  - update after each episode
- Non-stationary environment
  \[
  V(s) \leftarrow V(s) + \alpha[R - V(s)]
  \]
- A problem
  - greedy policy won’t explore all actions

Maintaining exploration

- Deterministic/greedy policy won’t explore all actions
  - don’t know anything about the environment at the beginning
  - need to try all actions to find the optimal one

- Maintain exploration
  - use soft policies instead: $\pi(s,a) > 0$ (for all s,a)

- $\epsilon$-greedy policy
  - with probability $1-\epsilon$ perform the optimal/greedy action
  - with probability $\epsilon$ perform a random action
  - will keep exploring the environment
  - slowly move it towards greedy policy: $\epsilon \to 0$
Simulated experience

- 5-card draw poker
  - s0: A♣, A♦, 6♠, A♥, 2♠
  - a0: discard 6♠, 2♠
  - s1: A♣, A♦, A♥, A♠, 9♠ + dealer takes 4 cards
  - return: +1 (probably)

- DP
  - list all states, actions, compute P(s,a,s')
  - P( [A♣,A♦,6♠,A♥,2♠], [6♠,2♠], [A♠,9♠,4] ) = 0.00192

- MC
  - all you need are sample episodes
  - let MC play against a random policy, or itself, or another algorithm

Temporal Difference Learning

- Combines ideas from MC and DP
  - like MC: learn directly from experience (don’t need a model)
  - like DP: bootstrap
  - works for continuous tasks, usually faster than MC

- Constant-alpha MC:
  - have to wait until the end of episode to update
    \[ V(s_t) \leftarrow V(s_t) + \alpha [R_t - V(s_t)] \]

- simplest TD
  - update after every step, based on the successor
    \[ V(s_t) \leftarrow V(s_t) + \alpha [r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \]
TD in passive learning

- TD(0) key idea:
  - adjust the estimated utility value of the current state based on its immediately reward and the estimated value of the next state.

- The updating rule
$$V(s_t) \leftarrow V(s_t) + \alpha [r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

- $\alpha$ is the learning rate parameter
- Only when $\alpha$ is a function that decreases as the number of times a state has been visited increased, then can $V(s)$ converge to the correct value.

Algorithm TD($\lambda$)
(not in Russell & Norvig book)

- Idea: update from the whole epoch, not just on state transition.
$$V(s_t) \leftarrow V(s_t) + \alpha \sum_{k=1}^{\infty} \lambda^{t-k} [r_{t+k+1} + V(s_{t+k+1}) - V(s_t)]$$

- Special cases:
  - $\lambda=1$: LMS
  - $\lambda=0$: TD
- Intermediate choice of $\lambda$ (between 0 and 1) is best.
- Interplay with $\alpha$ …
**MC vs. TD**

- Observed the following 8 episodes:
  - A – 0, B – 0  B – 1  B – 1  B - 1
  - B – 1  B – 1  B – 1  B – 0

- MC and TD agree on V(B) = 3/4

- MC: V(A) = 0
  - converges to values that minimize the error on training data

- TD: V(A) = 3/4
  - converges to ML estimate of the Markov process
The TD learning curve

![Graph showing the TD learning curve](image)

Another model free method—TD-Q learning

- Define Q-value function
  \[ V(s) = \max_a Q(s, a) \]

- Q-value function updating rule
  - See subsequent slides

- Key idea of TD-Q learning
  - Combined with temporal difference approach

- Rule to choose the action to take
  \[ a = \arg\max_a Q(s, a) \]

\[ \Pi(s) = V(s) + V \arg\max_a \sum_s P_{sa}(s) V(s) \]
Sarsa

- Again, need $Q(s,a)$, not just $V(s)$

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

$$V(s_t) \leftarrow V(s_t) + \alpha [r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

- Control
  - start with a random policy
  - update $Q$ and $\pi$ after each step
  - again, need $\epsilon$-soft policies

Q-learning

- Before: on-policy algorithms
  - start with a random policy, iteratively improve
  - converge to optimal

- Q-learning: off-policy
  - use any policy to estimate $Q$
    $$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

  - $Q$ directly approximates $Q^*$ (Bellman optimality eqn)
  - independent of the policy being followed
  - only requirement: keep updating each $(s,a)$ pair

- Sarsa
  $$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$
For each pair \((s, a)\), initialize \(Q(s,a)\)

Observe the current state \(s\)

Loop forever

\[
\begin{align*}
&\text{Select an action } a \text{ (optionally with } \epsilon\text{-exploration) and execute it} \\
&a = \arg \max_a Q(s, a) \\
&\text{Receive immediate reward } r \text{ and observe the new state } s' \\
&\text{Update } Q(s,a) \\
Q(s, a) &\leftarrow Q(s,a) + \alpha[r_{t+1} + \gamma \max_{a'} Q(s', a') - Q(s,a)] \\
&s' = s' \\
\end{align*}
\]

### Exploration

- Tradeoff between exploitation (control) and exploration (identification)

- Extremes: greedy vs. random acting (n-armed bandit models)

Q-learning converges to optimal Q-values if

- Every state is visited infinitely often (due to exploration),
- The action selection becomes greedy as time approaches infinity, and
- The learning rate \(\alpha\) is decreased fast enough but not too fast (as we discussed in TD learning)
State representation

- pole-balancing
  - move car left/right to keep the pole balanced

- state representation
  - position and velocity of car
  - angle and angular velocity of pole

- what about Markov property?
  - would need more info
  - noise in sensors, temperature, bending of pole

- solution
  - coarse discretization of 4 state variables
    - left, center, right
  - totally non-Markov, but still works
Function approximation

- represent $V_t$ as a parameterized function
  - linear regression, decision tree, neural net, ...
  - linear regression:
    $$V_t(s) = \theta_t^T \phi(s) = \sum_{i=1}^{n} \theta_t(i) \phi_s(i)$$

- update parameters instead of entries in a table
  - better generalization
    - fewer parameters and updates affect "similar" states as well

- TD update
  $$V(s_t) \leftarrow V(s_t) + \alpha \left[ r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \right]$$
  $$V(s_t) \times r_{t+1} + \gamma V(s_{t+1})$$
  - treat as one data point for regression
  - want method that can learn on-line (update after each step)

Features

- tile coding, coarse coding
  - binary features
    - a) Irregular
    - b) Log stripes
    - c) Diagonal stripes

- radial basis functions
  - typically a Gaussian
  - between 0 and 1

[ Sutton & Barto, Reinforcement Learning ]
Splitting and aggregation

- want to discretize the state space
  - learn the best discretization during training

- splitting of state space
  - start with a single state
  - split a state when different parts of that state have different values

- state aggregation
  - start with many states
  - merge states with similar values

Designing rewards

- robot in a maze
  - episodic task, not discounted, +1 when out, 0 for each step

- chess
  - GOOD: +1 for winning, -1 losing
  - BAD: +0.25 for taking opponent’s pieces
    - high reward even when lose

- rewards
  - rewards indicate what we want to accomplish
  - NOT how we want to accomplish it

- shaping
  - positive reward often very “far away”
  - rewards for achieving subgoals (domain knowledge)
  - also: adjust initial policy or initial value function
A Success Story

- **TD Gammon** (Tesauro, G., 1992)
  - A Backgammon playing program.
  - Application of **temporal difference** learning.
  - The basic learner is a neural network.
  - It trained itself to the world class level by playing against itself and learning from the outcome. So smart!!

---

**Case study: Back gammon**

- **rules**
  - 30 pieces, 24 locations
  - roll 2, 5: move 2, 5
  - hitting, blocking
  - branching factor: 400

- **implementation**
  - use TD(\( \lambda \)) and neural nets
  - 4 binary features for each
  - no BG expert knowledge

- **results**
  - TD-Gammon 0.0: trained against itself (300,000 games)
    - as good as best previous BG computer program (also by Tesauro)
    - lot of expert input, hand-crafted features
  - TD-Gammon 1.0: add special features
  - TD-Gammon 2 and 3 (2-ply and 3-ply search)
    - 1.5M games, beat human champion
Summary

- Reinforcement learning
  - use when need to make decisions in uncertain environment

- Solution methods
  - dynamic programming
    - need complete model
  - Monte Carlo
  - time difference learning (Sarsa, Q-learning)

- most work
  - algorithms simple
  - need to design features, state representation, rewards

Future research in RL

- Function approximation (& convergence results)
- On-line experience vs. simulated experience
- Amount of search in action selection
- Exploration method (safe?)
- Kind of backups
  - Full (DP) vs. sample backups (TD)
  - Shallow (Monte Carlo) vs. deep (exhaustive)
    - $\lambda$ controls this in TD($\lambda$)
- Macros
  - Advantages
    - Reduce complexity of learning by learning subgoals (macros) first
    - Can be learned by TD($\lambda$)
  - Problems
    - Selection of macro action
    - Learn models of macro actions (predict their outcome)
    - How do you come up with subgoals
Types of Learning

- **Supervised Learning**
  - Training data: \((X, Y)\) (features, label)
  - Predict \(Y\), minimizing some loss.
  - Regression, Classification.

- **Unsupervised Learning**
  - Training data: \(X\) (features only)
  - Find "similar" points in high-dim \(X\)-space.
  - Clustering.

- **Reinforcement Learning**
  - Training data: \((S, A, R)\) (State-Action-Reward)
  - Develop an optimal policy (sequence of decision rules) for the learner so as to maximize its long-term reward.
  - Robotics, Board game playing programs

Where Machine Learning is being used or can be useful?