RL @ DL Study Group

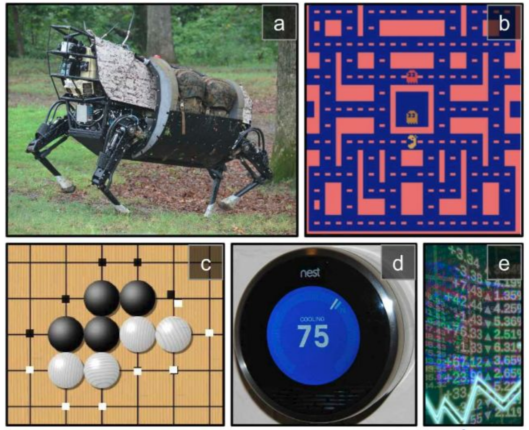
Thomas Balestri

Chapter Summary

- What is RL?
- Techniques in deep RL
  - policy gradients
  - deep Q-networks (DQN)
  - Temporal-difference learning

Learning to optimize rewards

- an *agent* interacts with the *environment*
- the environment provides a *reward* to the agent
- *objective*: maximize the long-term *expected reward*
    - example: walking robot; environment = world; interact via sensors; positive reward for reaching a destination; negative reward for getting lost or falling down



[illegible]

Policy

- *Policy*: the algorithm the agent uses to choose an action given its environment

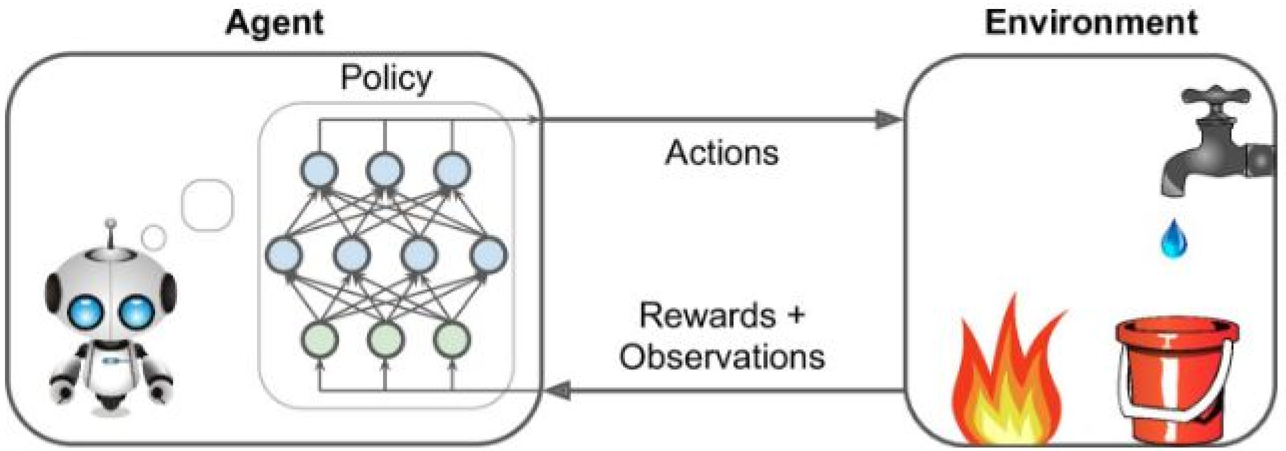


Figure 16-2. Reinforcement Learning using a neural network policy

Policy search

- Ex: Vacuum whose reward is how much dust picked up in 30 minutes
    - policy: move forward with probability *p*, randomly rotate left or right with probability *1-p* with rotation angle *+r* to *-r*
- Policy = try different combinations of *(p,r)* and learn which collects the most dust
    - Problem: brute-force approach has a huge search space
    - Solution: use optimization method by evaluating gradients of rewards w.r.t. policy parameters, changing parameters in direction that maximizes the reward (*Policy Gradients)*

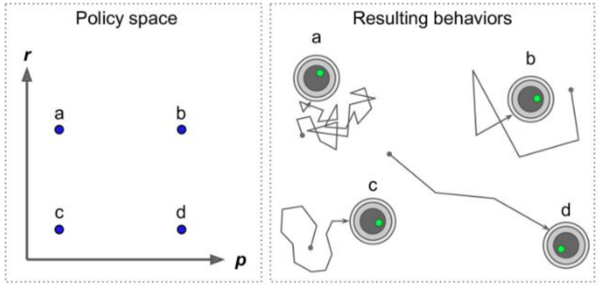


Figure 16-3. Four points in policy space and the agent's corresponding behavior

Neural Network Policies

- input = observation, output = execution of an action according to a probability distribution over the actions
- e.g. Cart pole has 2 actions (move left or move right)
- choosing actions based on probs allows for *exploration* and *exploitation*

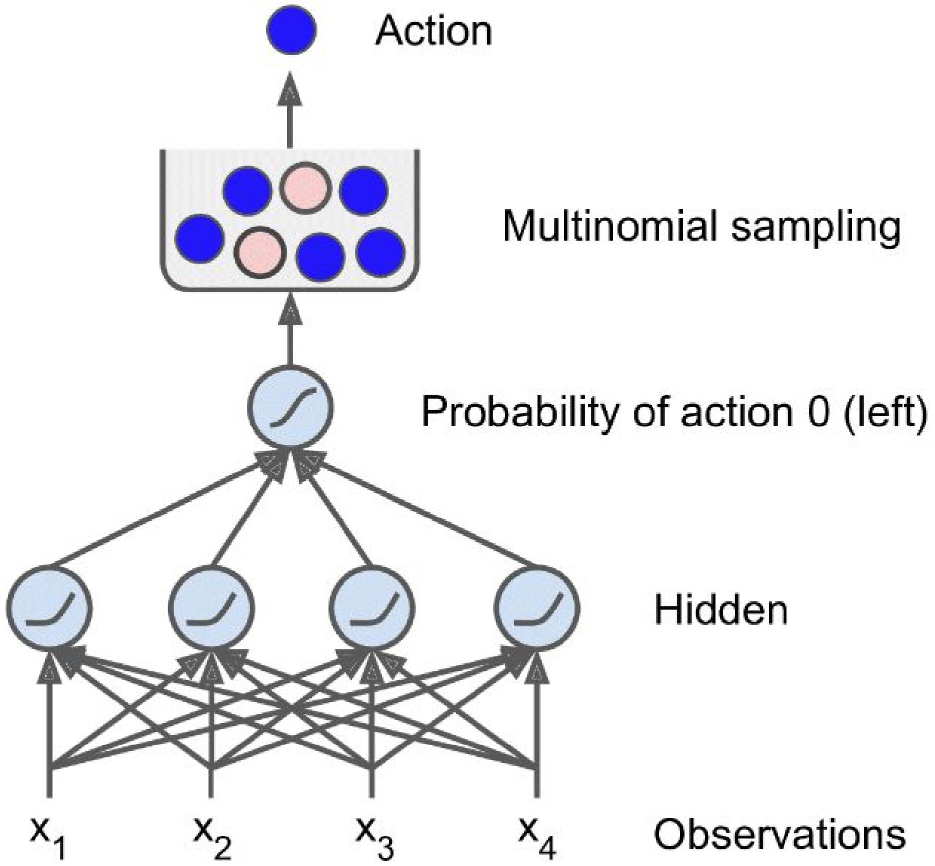


Figure 16-5. Neural network policy

Credit Assignment Problem

- If we knew correct action at each time step *t,* would just be a supervised learning problem (minimize cross-entropy loss between prediction and target)
- But we only have rewards provided by the environment (sparse and delayed)
- cart balancing a pole: pole falls after 100 actions, but which actions were good and which were bad?
- *credit-assignment problem*: summation of rewards that came after an action *a* at timestep *t* at discount ratio *r*
    - on average, good action should get better score than bad ones

$$10 + r*0 + r^2*(-50) = -22$$

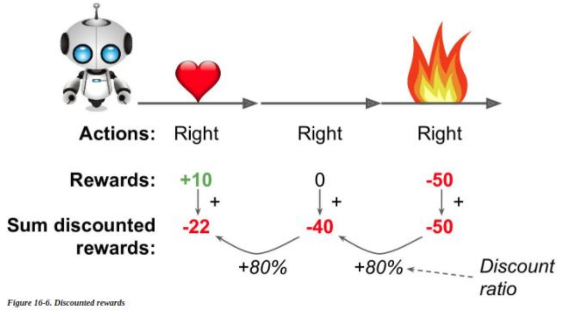


Figure 16-6. Discounted rewards

Policy Gradients

- follow gradients towards higher rewards
- *REINFORCE* algorithms
  - let nn policy play the game several times; each time, compute gradients that would make chosen action more likely (but don’t apply gradient updates)
  - after several games, comput each action’s score (credit assignment)
  - if action is positive, appy gradients calculated for this action to make action more likely; if action is negative, apply opposite gradients to make this action less likely; multiply gradient vectors by action score
  - compute mean of resulting gradient vectors; use this to perform gradient step and update nn weights

Markov Chains

- PG algos try to directly optimize policy to increase rewards
- other classes of algos can learn to estimate expected sum of discounted rewards for each state, then use this to choose an action
- *Markov Chains*:
    - fixed number of states
    - randomly evolve to next state
    - probability to evolve from state *s* to *s'* is fixed and depends only on state pair (*s,s'*), not on past states (memoryless)

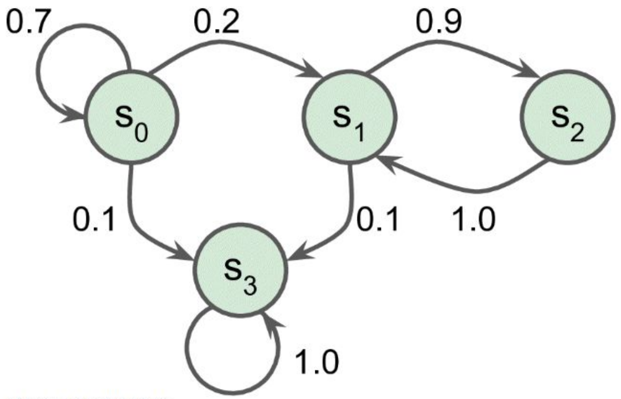


Figure 16-7. Example of a Markov chain

Markov Decision Process (MDP)

- similar to Markov chains, but at each step, agent can choose several actions and transition probability depends on action taken
- some transition states result in reward (pos/neg)
- agent goal: find policy to maximize reward over time
- Which strategy will gain the most reward over time => Bellman Eqn

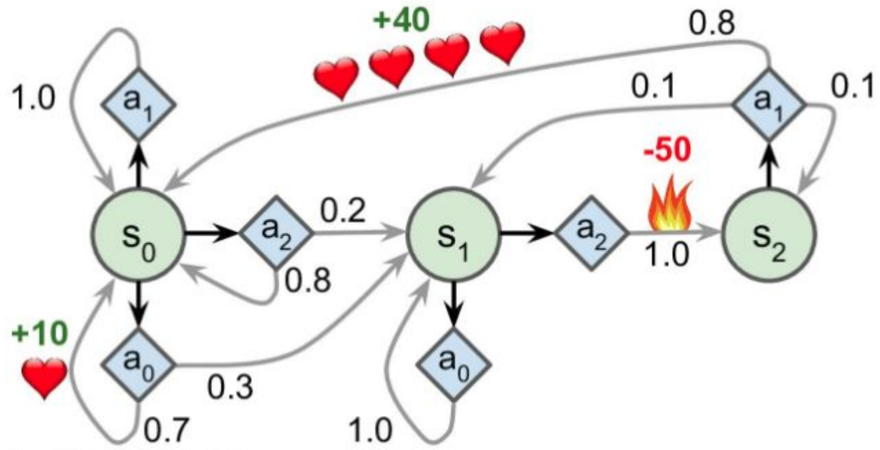
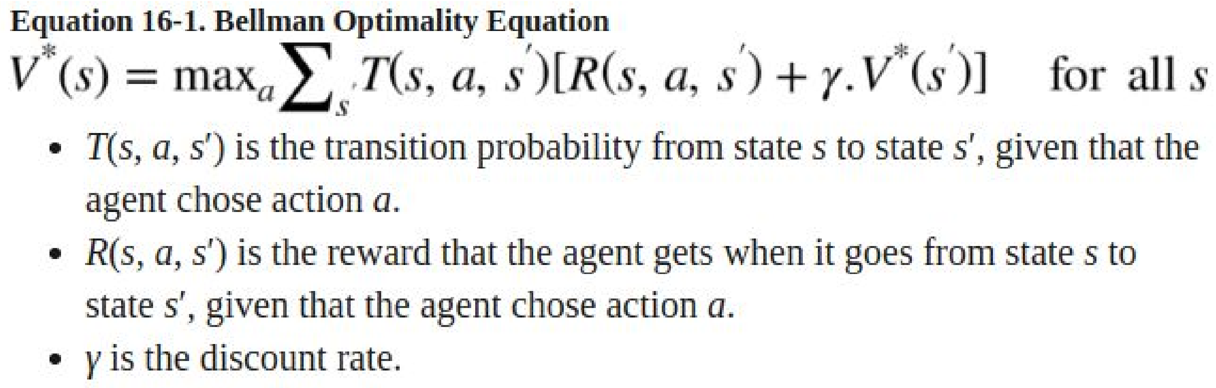


Figure 16-8. Example of a Markov decision process

Bellman Optimality Equation

- Equation to estimate the optimal state value *V**, which is the sum of all future discounted rewards the agent can expect on average from state *s*
- Recursive eqn that says if the agent acts optimally, then optimal value of current state is equal to reward it will get on average after taking one optimal action, plus expected optimal value of all possible next states that this action can lead to
- leads directly to algorithm that can precisely estimate optimal state value of every possible state



Equation 16-1. Bellman Optimality Equation

$$V^*(s) = \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma . V^*(s')] \quad \text{for all } s$$

- $T(s, a, s')$ is the transition probability from state s to state s', given that the agent chose action a.
- $R(s, a, s')$ is the reward that the agent gets when it goes from state s to state s', given that the agent chose action a.
- γ is the discount rate.

Q-values

- knowing optimal state values can help in evaluating policy, but doesn’t tell agent what to do
- optimal Q-value Q* of state-action pair (s,a) is sum of discounted future reward agent can expect on average after it reaches state *s* and chooses action *a*
- can define optimal policy Π* which optimal Q-values Q* by choosing action *a* with highest Q-value in a given state *s*

Equation 16-3. Q-Value Iteration algorithm

$$Q_{k+1}(s, a) \leftarrow \sum_{s'} T(s, a, s')\left[R(s, a, s') + \gamma . \max_{a'} Q_k(s', a')\right] \quad \text{for all } (s, a)$$

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q^*(s, a)$$

Temporal Difference (TD) Learning

- similar to value iteration, but takes into account that the agent has only partial knowledge of the transition probability T(s,a,s’) and the reward R(s,a,s’)
- initially, assume agent knows only possible states and actions
- use exploration policy to explore MDP
- as it progresses, TD algo updates estimates of state values based on transitions and rewards actually observed
- for each state, algo keeps running average of immediate rewards agent gets upon leaving state plus rewards it expects to get later

Equation 16-4. TD Learning algorithm

$$V_{k+1}(s) \leftarrow (1-\alpha)V_k(s) + \alpha\left(r + \gamma . V_k(s')\right)$$

Q-learning

- adapts Q-value iteration to situation where transition probabilities and rewards are initially unknown
- for each state-action pair (s,a), keep track of running average of rewards agent receives upon leaving state *s* with action *a*, plus rewards agent expects to get later

Equation 16-5. Q-Learning algorithm

$$Q_{k+1}(s, a) \leftarrow (1-\alpha)Q_k(s, a) + \alpha\left(r + \gamma.\max_{a'} Q_k(s', a')\right)$$