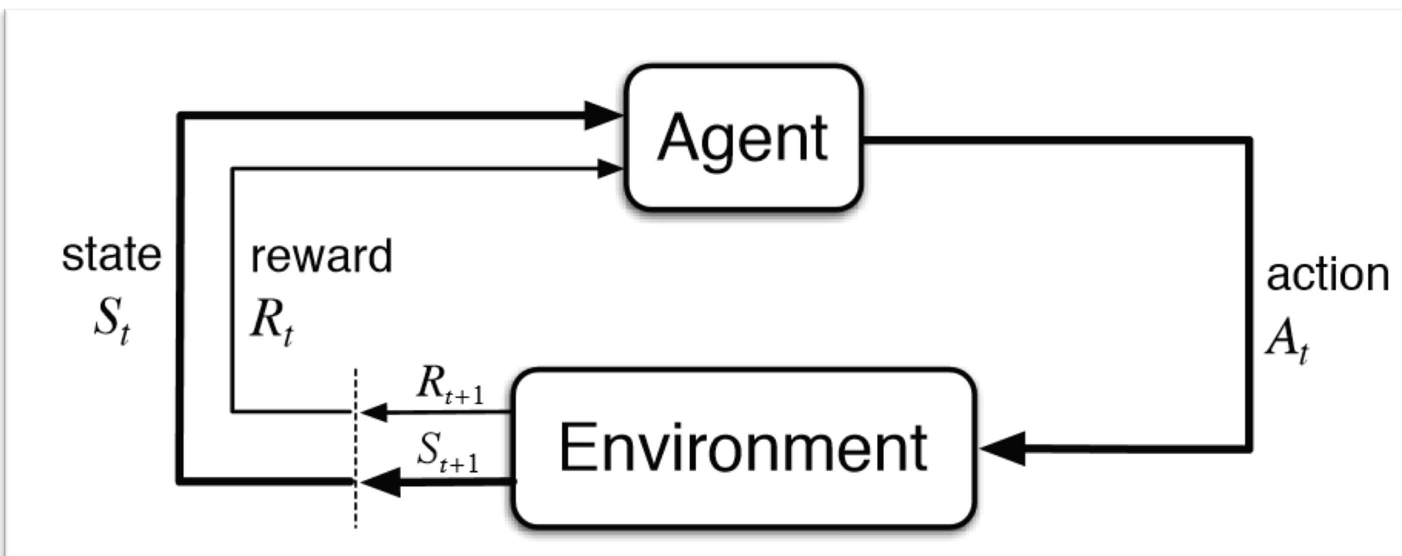
Introduction to Reinforcement Learning

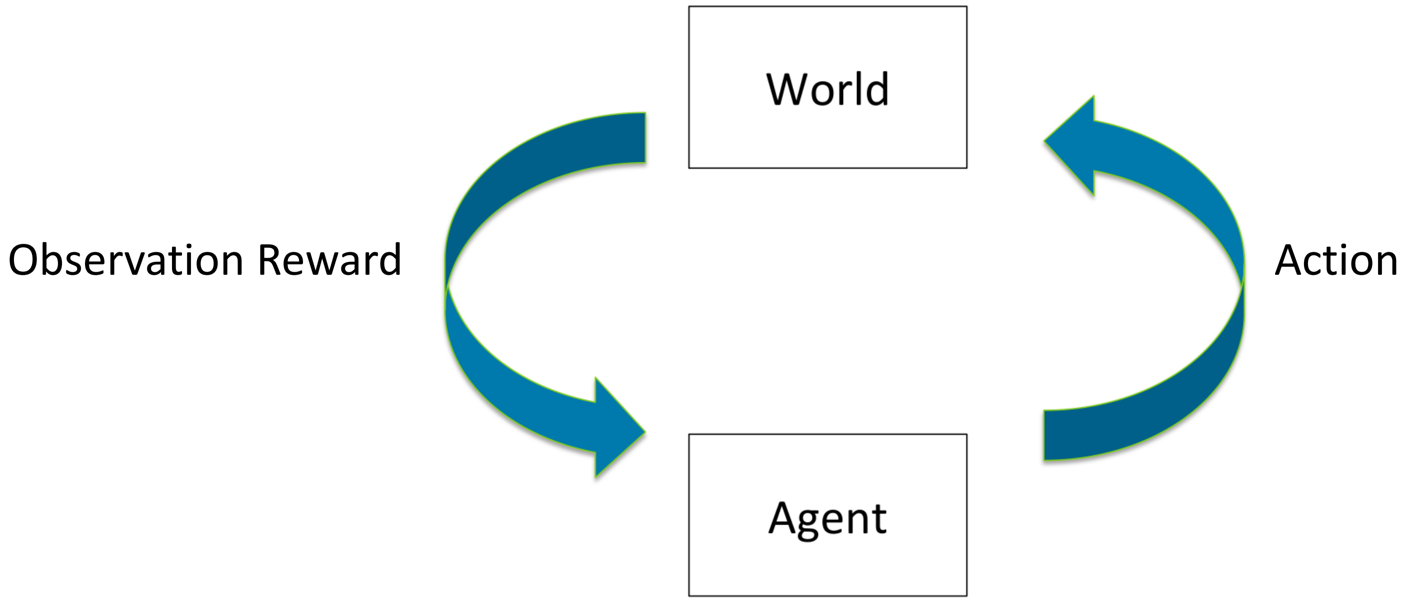
Overview

- Learn to make good sequences of decisions
- Don’t know in advance how world works
- Repeated interactions with environment
- Reward for sequence of decisions



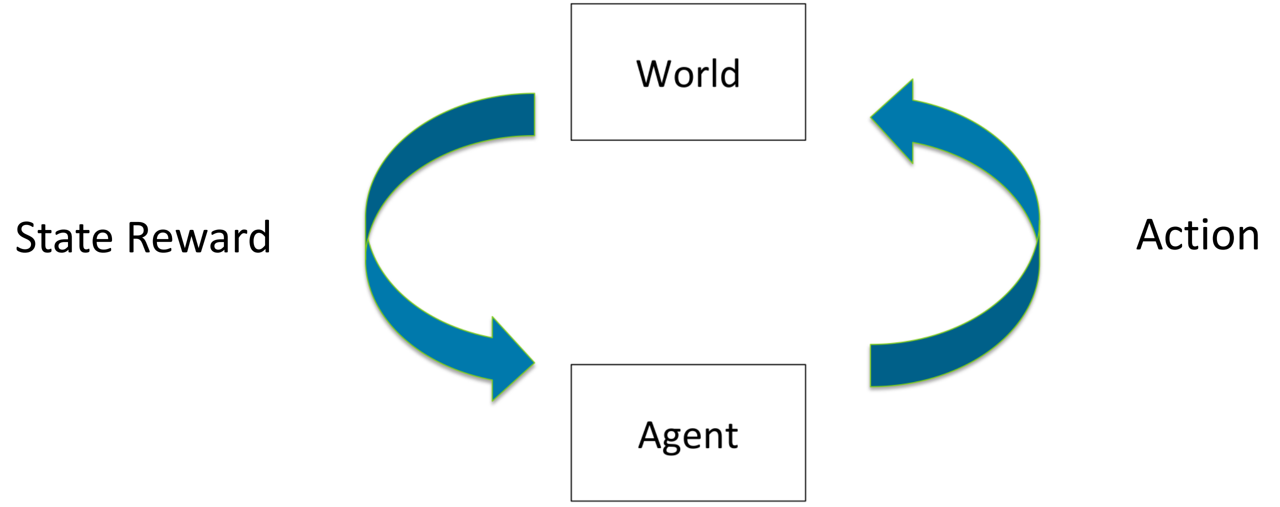
Decision Making Under Uncertainty

- Multi-armed bandit
- A set of arms to select
    - Obtain reward which follows unknown distribution
- Actions do not change the state of the world



Markov Decision Process

- Actions change the state of the world
- State = Observation
- Sufficient statistic that captures how world behaves
- Policy: mapping from state to action



Markov Decision Process: $< S, A, R, T, \gamma >$

- S: set of states
- A: set of actions
- R: immediate reward $R(s)$ / $R(s,a)$ / $R(s,a,s')$
- T: dynamics model $p(s_{t+1}|s_t, a_t)$
- γ: discount factor
  - the difference in importance between future rewards and present rewards
- Memoryless
  - The outcome of an action depends only on the current state (vs entire history)
- Policy $\pi: S \rightarrow A$
  - Specifies what action to take in each state

MDP Policy Value

- For a given state s
- Value of policy $V^{\pi}(s)$: Expected discounted sum of rewards obtain if the agent follows policy π starting in state s
  - $V^{\pi}(s) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r\big(s_t, \pi(s_t)\big) | s_0 = s]$
- Optimal policy: $argmax_{\pi} V^{\pi}(s)$
- Immediate reward + Discounted sum of future rewards
  - $V^{\pi}(s) = r\big(s, \pi(s)\big) + \gamma \sum_{s' \in S} p(s' | \pi(s), s) V^{\pi}(s')$

Q: state-policy value

- Expected immediate reward for taking action a and expected future reward get after taking that action from that state and following π
- $Q^{\pi}(s,a) = r(s,a) + \gamma \sum_{s' \in S} p(s'|a,s)\, V^{\pi}(s')$

Optimal Value, Q & Policy

- Optimal V
  - Highest possible value for each s (under any possible policy)
  - Satisfies the Bellman Equation
  - $V^*(s) = \max_a [r(s,a) + \gamma \sum_{s' \in S} p(s'|a,s)\, V^*(s')]$
- Optimal Q function
  - $Q^*(s,a) = r(s,a) + \gamma \sum_{s' \in S} p(s'|a,s)\, V^*(s')$
- Optimal policy
  - $\pi^*(s) = argmax_a\, Q^*(s,a)$

MDP Planning

- How to compute π^*?
- Know full MDP
- Given the dynamics and reward model
  - Reward and state transition probability
- Computational challenge
  - Not learning

Value Iteration

- Bellman equation inspires an update rule
  - $V^*(s) = \max_a [r(s,a) + \gamma \sum_{s' \in S} p(s'|a,s)\, V^*(s')]$
- First compute value for each state as if only get to take 1 action
- Then what if take 2 actions…
- Estimate the optimal value
- Output
  - The solution: derive the optimal policy π^* from the optimal value
  - The discounted sum of the rewards to be earned (on average) by following that solution from state s

Value Iteration

1. Initialize $V_0(s) = 0$ for all states s,
2. Set $k = 1$
3. Loop until [finite horizon, convergence]
   - For each state s
   - $V_{k+1}(s) = \max_a [r(s,a) + \gamma \sum_{s' \in S} p(s'|a,s) \, V_k(s')]$
4. Extract Policy $\pi(s)$

Policy Iteration

- Search directly for the optimal policy π^*
- Compute infinite horizon value of a policy
- Use to select another (better) policy
- Closely related to a very popular method in RL
  - policy gradient

Policy Iteration

1. Initialize $\pi_0(s)$ randomly for all states s
2. Loop until [finite horizon, convergence]
   - Policy evaluation: Compute V^{π_i}
   - Policy improvement:
     - Compute Q value of different 1st action and then following π_i
     - $Q^{\pi_i}(s,a) = r(s,a) + \gamma \sum_{s' \in S} p(s'|a,s)\, V^{\pi_i}(s')$
     - Use to extract a new policy
     - $\pi_{i+1}(s) = argmax_a\, Q^{\pi_i}(s,a)$

Convergence

- Converge to a unique solution
  - for discrete state and action space
  - when $\gamma < 1$
  - all state-action pairs are visited infinitely often

Model-based Passive Reinforcement Learning

- Estimate MDP model parameters from data
  - Reward
  - State transition probability
- If finite set of states and actions
  - count & average
- Use estimated MDP to do policy evaluation of π

Model-free Passive Reinforcement Learning

- Only maintain estimate of Q value
- Temporal Difference learning
  - Approximate expectation with samples
  - Approximate future reward with estimate
- Maintain estimate of $V^{\pi}(s)$ for all states
  - Update $V^{\pi}(s)$ each time after each transition (s, a, s', r)
  - Likely outcomes s' will contribute updates more often
  - Approximate expectation over next state with samples

Q-learning

- Update $Q(s, a)$ every time experience (s, a, s', r)
- Create new sample estimate
  - $Q_{samp}(s, a) = r(s, a) + \gamma \max_{a'} Q(s', a')$
- Update estimate of $Q(s, a)$
  - $Q(s, a) = (1 - \alpha)Q(s, a) + \alpha Q_{samp}(s, a)$
- If acting randomly, Q-learning converges Q^*
- Optimal Q values
- Finds optimal policy

Simple Approach: ϵ-greedy

- With probability $1 - \epsilon$
  - Choose $argmax_a Q(s, a)$
- With probability ϵ
  - Select random action
- Even after millions of steps still won't always be following argmax of $Q(s, a)$

Example

- State: the amount of occupied resource in the cloud
- Action: whether to accept newly arrived job
- Reward: revenue earned by the infrastructure provider
- Deterministic state transition
  - Consider the average time the system stays at state s

Scaling Up

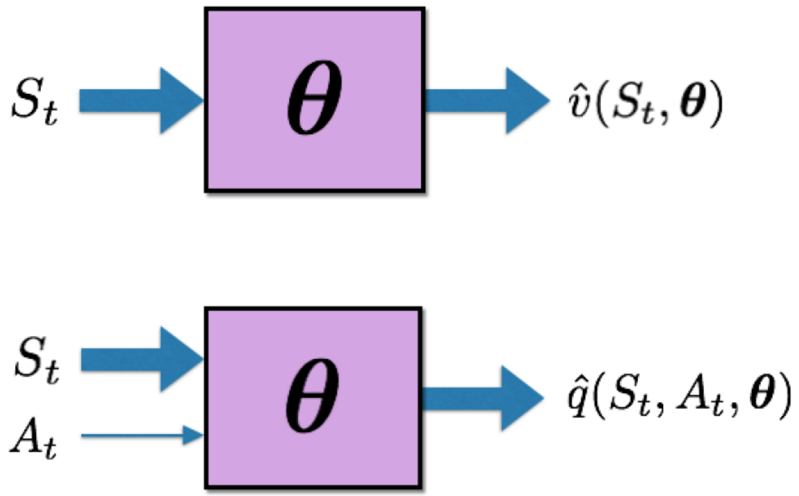
- Want to be able to tackle problems with enormous or infinite state spaces
- Tabular representation is insufficient
- Don't want to have to explicitly store
  - dynamics or reward model
  - value
  - state-action value
  - policy
- for every single state

Generalization

- Smoothness assumption
- If two states are close, then (at least one of)
  - Dynamics are similar
  - Reward is similar
  - Q functions are similar
  - optimal policy is similar
- More generally, dimensionality reduction or compression
  - Unnecessary to individually represent each state
  - Compact representations possible

Function Approximation

- Key idea: replace lookup table with a function
- Replace table with general parameterized form



- Examples:
  - Linear combinations of features
  - Neural networks

Linear Value Function Approximation

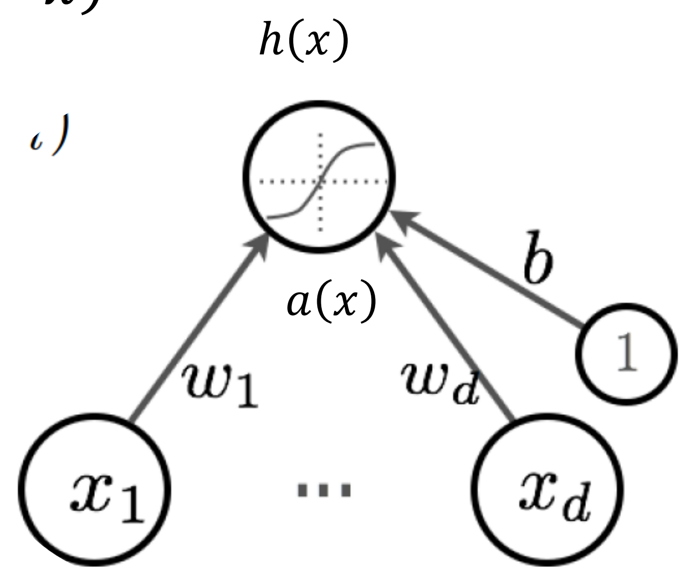
- Represent state by a feature vector $\mathrm{x}(s) = \begin{pmatrix} x_1(s) \\ x_2(s) \end{pmatrix}$
- For example
  - Distance of robot from landmarks
  - Trends in the stock market
  - Piece and pawn configurations in chess
- Represent value function by a linear combination of features
  - $\hat{v}(s, w) = x(s)^T w$

Linear Value Function Approximation

- Objective function is quadratic in parameters w
  - $J(w) = \mathbb{E}_\pi[(v_\pi(s) - x(s)^T w)^2]$
- Update = step-size × prediction error × feature value
  - $\Delta w = \alpha\big(v_\pi(s) - \hat{v}(s, w)\big)x(s)$
  - Use historical average for $v_\pi(s)$

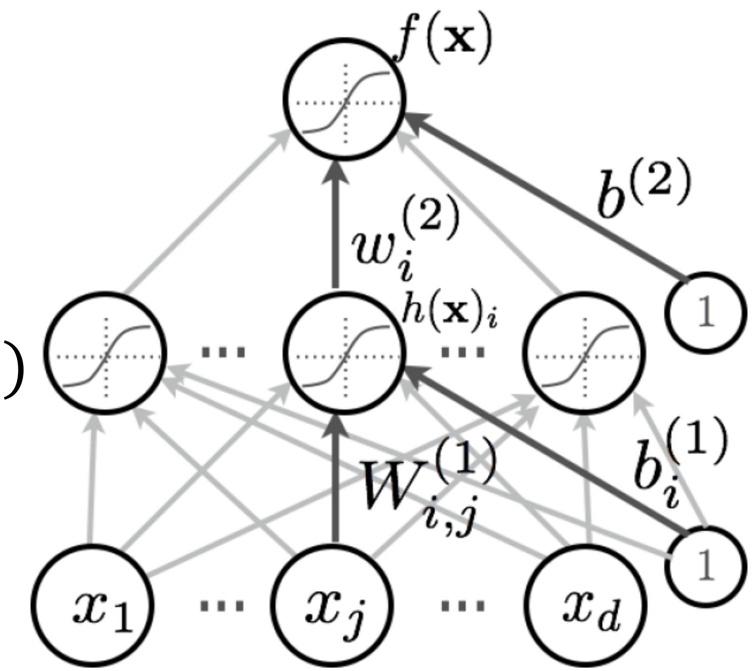
Deep Neural Networks

- Input activation
  - $a(x) = b + w^T x$
- Output activation
  - $h(x) = g\big(a(x)\big) = g(b + w^T x)$
- x: features
- w: weights (parameters)
- b: bias term
- $g(\cdot)$: activation function
  - Sigmoid function
  - Rectified linear function



Single Hidden Layer Neuro Net

- Hidden layer pre-activation
  - $a(x) = b^{(1)} + w^{(1)^T} x$
- Hidden layer activation
  - $h(x) = g\big(a(x)\big)$
- Output layer activation
  - $f(x) = o(b^{(2)} + w^{(2)^T} h^{(1)}(x))$



How to train neural nets

- To train a neural net, we need:
- Loss function
  - $l(f(x^{(t)}, \theta), y^{(t)})$
- A procedure to compute gradients
  - $\nabla_\theta \, l(f(x^{(t)}, \theta), y^{(t)})$
- Regularizer and its gradient
  - $\Omega(\theta), \nabla_\theta \, \Omega(\theta)$
  - Prevent overfitting

Stochastic Gradient Descent

- Perform updates after seeing each example
- For each training epoch
  - For each training example $(x^{(t)}, y^{(t)})$
  - $\Delta = -\nabla_\theta \, l\left(f\left(x^{(t)}, \theta\right), y^{(t)}\right) - \lambda \nabla_\theta \, \Omega(\theta)$
  - $\theta = \theta + \alpha\Delta$
- Backpropagation with gradient descent
  - Calculate the error contribution of each neuron after a batch of data is processed
  - From upper layer to lower layer

Example

- Feature: available resource, job resource profile
- Action: whether to schedule job in one time slot
- Reward: job completion time

Summary

- Standard Value iteration / Q learning is not very useful
  - State space is large
  - Convergence
  - Infinitely visiting each state-action pair
- Other work provide regret bound on modified Q learning
- DNN is useful
  - But there is no theoretical support behind

Thank you! ^^