Generalization in Reinforcement Learning

Large State Spaces

- When a problem has a large state space we can not longer represent the V or Q functions as explicit tables
- Even if we had enough memory
  - Never enough training data!
  - Learning takes too long
- What to do?? …. Generalize situations

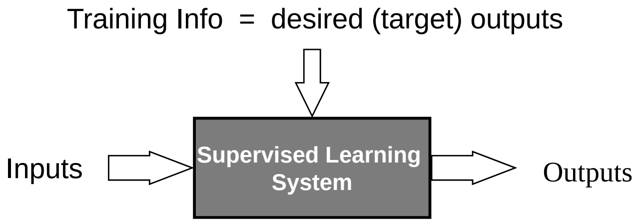
Approximate Reinforcement Learning

- Why?
  - To learn in reasonable time and space
    (avoid Bellman's curse of dimensionality)
  - To generalize to new situations
- Solutions
  - Approximate the value function
  - Search in the policy space

Approximate Reinforcement Learning

- Why?
  - To learn in reasonable time and space
    (avoid Bellman's curse of dimensionality)
  - To generalize to new situations
- Solutions
  - Approximate the value function
  - Search in the policy space

Adapt Supervised Learning Algorithms



Training example = {input, target output}

Error = (target output – actual output)

Backups as Training Examples

e.g., the TD(0) backup :

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

As a training example:

$$\{ \underbrace{\text{description of } s_t}_{\text{input}}, \; \underbrace{r_{t+1} + \gamma V(s_{t+1})}_{\text{target output}} \}$$

Any FA Method?

- In principle, yes:
  - artificial neural networks
  - decision trees
  - multivariate regression methods
  - etc.
- But RL has some special requirements:
  - usually want to learn while interacting
  - ability to handle nonstationarity
  - other?

Gradient Descent Methods

$$\vec{\theta}_t = \left(\theta_t(1), \theta_t(2), \ldots, \theta_t(n) \right)^T$$

Assume V_t is a (sufficiently smooth) differentiable function of $\vec{\theta}_t$, for all $s \in S$.

Assume, for now, training examples of this form :

$$\{ \text{description of } s_t, \; V^{\pi}(s_t) \}$$

Performance Measures

- Many are applicable but…
- a common and simple one is the mean-squared error (MSE) over a distribution *P* :

$$MSE(\theta_t) = \sum_{s \in S} P(s)\left[V^{\pi}(s) - V_t(s)\right]^2$$

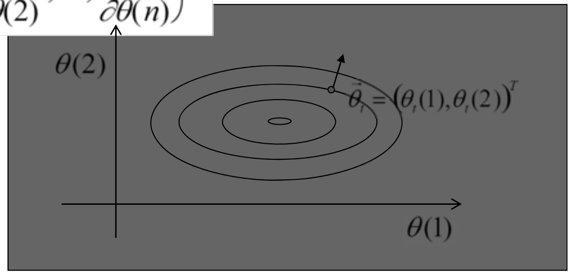
- Why *P* ?
- Why minimize MSE?
- Let us assume that *P* is always the distribution of states at which backups are done.
- The **on-policy distribution**: the distribution created while following the policy being evaluated. Stronger results are available for this distribution.

Gradient Descent

Let f be any function of the parameter space.

Its gradient at any point $\vec{\theta}_t$ in this space is :

$$\nabla_{\vec{\theta}} f(\vec{\theta}_t) = \left(\frac{\partial f(\vec{\theta}_t)}{\partial \theta(1)}, \frac{\partial f(\vec{\theta}_t)}{\partial \theta(2)}, \ldots, \frac{\partial f(\vec{\theta}_t)}{\partial \theta(n)}\right)^T.$$



Iteratively move down the gradient:

$$\vec{\theta}_{t+1} = \vec{\theta}_t - \alpha \nabla_{\vec{\theta}} f(\vec{\theta}_t)$$

Function Approximation

- Never enough training data!
  - Must generalize what is learned from one situation to other “similar” new situations
- Idea:
  - Instead of using large table to represent U or Q, use a parameterized function
    - The number of parameters should be small compared to number of states
  - Learn parameters from experience
  - When we update the parameters based on observations in one state, then our U or Q estimate will also change for other similar states
    - I.e. the parameterization facilitates generalization of experience

Linear Function Approximation

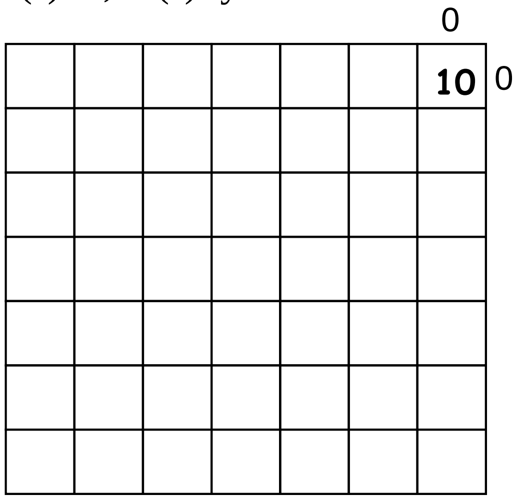
- Define a set of features f1(s), …, fn(s)
  - The features are used as our representation of states
  - States with similar feature values will be treated similarly
- A common approximation is to represent *U(s)* as a weighted sum of the features features (i.e. a linear approximation)

$$\hat{U}_{\theta}(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \ldots + \theta_n f_n(s)$$

- The approximation accuracy is fundamentally limited by the information provided by the features
- Can we always define features that allow for a perfect linear approximation?
  - Yes. Assign each state an indicator feature.
  - Of course this requires far to many features and gives no generalization.

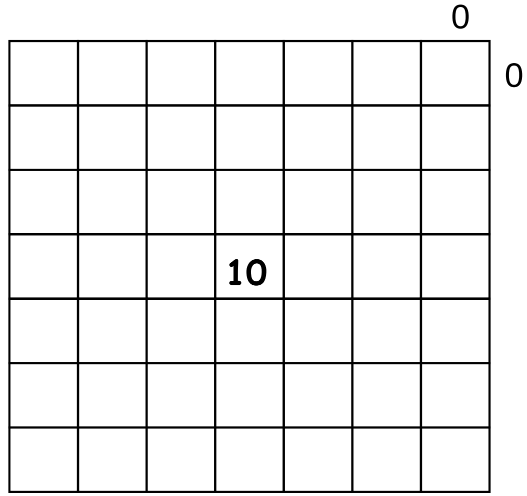
Example

- Consider grid problem with no obstacles
- Features for state s=(x,y): f1(s)=x, f2(s)=y
- $U(s) = \theta_0 + \theta_1 x + \theta_2 y$
- Is there a good linear approximation?
  - Yes.
  - $\theta_0 = 10$, $\theta_1 = -1$, $\theta_2 = -1$
  - (note upper right is origin)



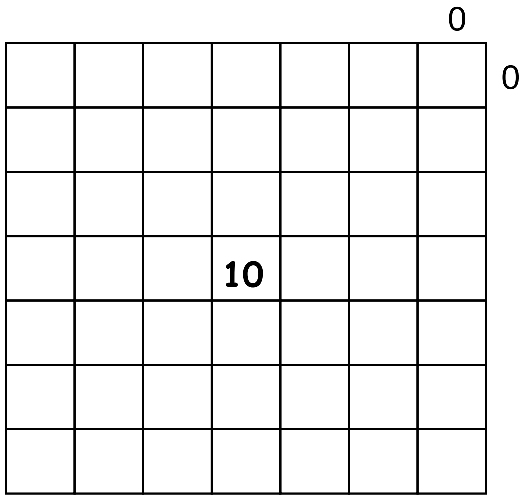
But What If…

- $U(s) = \theta_0 + \theta_1 x + \theta_2 y$
- Is there a good linear approximation?
  - No.



But What If…

- $U(s) = \theta_0 + \theta_1 x + \theta_2 y + \theta_3 z$
- Include new feature z
  - $z = |x_g - x| + |y_g - y|$
- Does this allow a good linear approx?
  - $\theta_0 = 10$, $\theta_1 = \theta_2 = 0$, $\theta_0 = -1$



Linear Function Approximation

- Define a set of features f1(s), …, fn(s)
  - The features are used as our representation of states
  - States with similar feature values will be treated similarly

$$\hat{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \ldots + \theta_n f_n(s)$$

- Our goal is to learn good parameter values (feature weights).
  - How can we do this?
  - Use TD-based RL and somehow update parameters based on each experience.

RL for Linear Approximators

1. Start with initial parameter values
2. Take action according to an explore/exploit policy (should converge to greedy policy, e.g. soft-max)
3. Update estimated model
4. Perform TD update for each parameter

$$\theta_i \leftarrow ?$$

5. Goto 2

What is a "TD update" for a parameter?

Aside: Gradient Descent for Squared Error

- Suppose that we have a sequence of states and target values/utilities for each state
  - E.g. produced by the TD-based RL loop
- Our goal is minimize the squared error between our estimated function and each example:

$$\underbrace{E_j(s)}_{\text{squared error of example j}} = \frac{1}{2}\left(\underbrace{\hat{U}_\theta(s)}_{\text{current estimate}} - \underbrace{u_j(s)}_{\text{target utility for j'th example}}\right)^2$$

- Gradient descent rule tells us to update parameters by:

$$\theta_i \leftarrow \theta_i - \underbrace{\alpha}_{\text{learning rate}} \frac{\partial E_j(s)}{\partial \theta_i} = \theta_i + \underbrace{\alpha}_{\text{learning rate}} \underbrace{\left(u_j(s) - \hat{U}_\theta(s)\right)}_{\frac{\partial E_j(s)}{\partial \hat{U}_\theta(s)}} \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$

Aside: continued

$$\theta_i \leftarrow \theta_i + \underbrace{\alpha}_{\text{learning rate}} \frac{\partial E_j(s)}{\partial \theta_i} = \theta_i + \underbrace{\alpha}_{\text{learning rate}} \underbrace{\left(u_j(s) - \hat{U}_\theta(s)\right)}_{\frac{\partial E_j(s)}{\partial \hat{U}_\theta(s)}} \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$

- For a linear approximation function:

$$\hat{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \ldots + \theta_n f_n(s)$$

$$\frac{\partial \hat{U}_\theta(s)}{\partial \theta_i} = f_i(s)$$

- Thus the update becomes: $\theta_i \leftarrow \theta_i + \alpha\left(u_j(s) - \hat{U}_\theta(s)\right) f_i(s)$
- For linear functions this update is guaranteed to converge to best approximation for suitable learning rate schedule

RL for Linear Approximators

1. Start with initial parameter values
2. Take action according to an explore/exploit policy (should converge to greedy policy, e.g. soft-max)
3. Perform TD update for each parameter

$$\theta_i \leftarrow \theta_i + \alpha\left(u_j(s) - \hat{U}_\theta(s)\right) f_i(s)$$

4. Goto 2

What should we use for $u_j(s)$?

$$u_j(s) = R(s) + \beta \hat{U}_\theta(s')$$

RL for Linear Approximators

1. Start with initial parameter values
2. Take action according to an explore/exploit policy (should converge to greedy policy, e.g. soft-max)
3. Perform TD update for each parameter

$$\theta_i \leftarrow \theta_i + \alpha\left(R(s) + \beta \hat{U}_\theta(s') - \hat{U}_\theta(s)\right) f_i(s)$$

4. Goto 2

- Note that step 2 still requires model to select action using one-step look-ahead.
- For applications such as Backgammon it is easy to get a simulation-based model
- But we can do the same thing for model-free Q-learning

Q-learning with Linear Approximators

$$\hat{Q}_\theta(s,a) = \theta_1 f_1(s,a) + \theta_2 f_2(s,a) + \ldots + \theta_n f_n(s,a)$$

1. Start with initial parameter values
2. Take action according to an explore/exploit policy (should converge to greedy policy, i.e. soft-max)
3. Perform TD update for each parameter

$$\theta_i \leftarrow \theta_i + \alpha\left(R(s) + \beta \max_{a'} \hat{Q}_\theta(s',a') - \hat{Q}_\theta(s,a)\right) f_i(s)$$

4. Goto 2

- For both Q and U learning these algorithms converge to the closest linear approximation to optimal Q or U.

Nice Properties of Linear FA Methods

- The gradient is very simple: $\nabla_{\vec{\theta}} V_t(s) = \vec{\phi}_s$
- For MSE, the error surface is simple: quadratic surface with a single minumum.
- Linear gradient descent TD(λ) converges:
  - Step size decreases appropriately
  - On-line sampling (states sampled from the on-policy distribution)
  - Converges to parameter vector $\vec{\theta}_\infty$ with property:

$$MSE(\vec{\theta}_\infty) \le \frac{1-\gamma\lambda}{1-\gamma} MSE(\vec{\theta}^*)$$

best parameter vector

(Tsitsiklis & Van Roy, 1997)

Q-l w/ Non-linear Approximators

$\hat{Q}_\theta(s,a)$ is sometimes represented by a non-linear approximator such as a neural network

1. Start with initial parameter values
2. Take action according to an explore/exploit policy (should converge to greedy policy, i.e. soft-max)
3. Perform TD update for each parameter

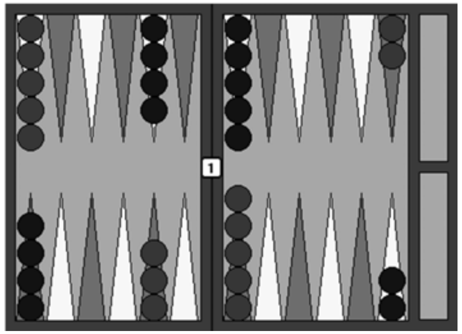
$$\theta_i \leftarrow \theta_i + \alpha\left(R(s) + \beta \max_{a'} \hat{Q}_\theta(s',a') - \hat{Q}_\theta(s,a)\right) \frac{\partial \hat{Q}_\theta(s,a)}{\partial \theta_i}$$

4. Goto 2

calculate closed-form

- Typically the space has many local minima and we no longer guarantee convergence
- Often works well in practice

One of the Worlds Best Backgammon Players



- Neural network with 80 hidden units
  - Used computed features
- Used TD-updates for 300,000 games against self
- Is one of the top (2 or 3) players in the world!

Other successful RL applications

- Checker Player
- Elevator Control (Barto & Crites)
- Space shuttle job scheduling (Zhang & Dietterich)
- Dynamic channel allocation in cellphone networks (Singh & Bertsekas)
- Robot Control
- Supply Chain Management

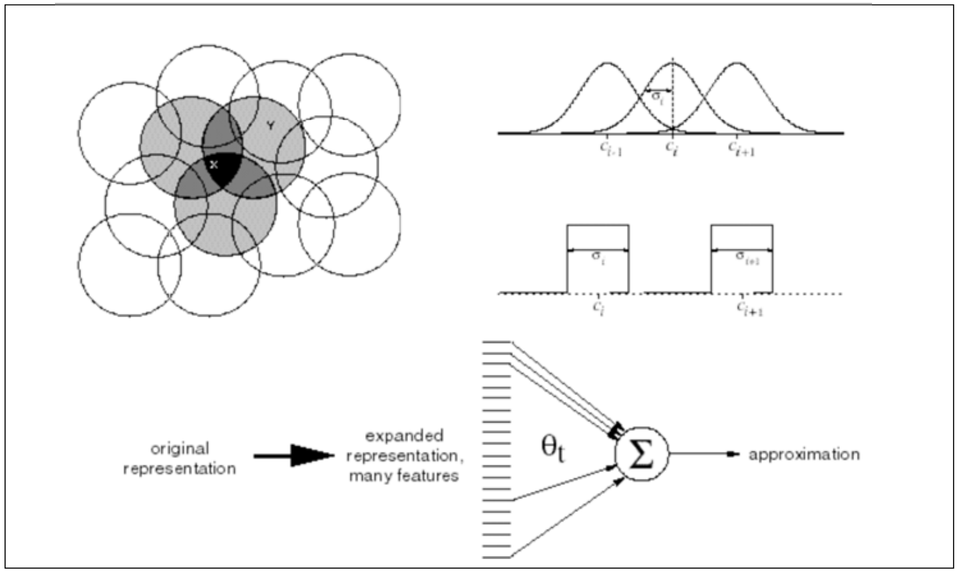
RL Function Approximation

- High-dimensionality addressed by
  - replacing v(s) or Q(s,a) by representation

$$\widetilde{Q}(s,a) = \sum_{i=1}^{k} w_i \phi_i(s,a)$$

    and then applying Q-learning algorithm updating weights w_i at each iteration, or
  - approximating v(s) or Q(s,a) by a neural network
- Issue: choose “basis functions” $\varphi_i(s,a)$ to reflect problem structure

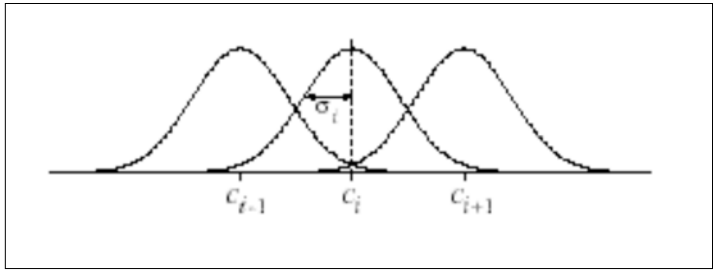
Coarse Coding



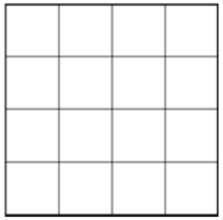
Radial Basis Functions (RBFs)

e.g., Gaussians

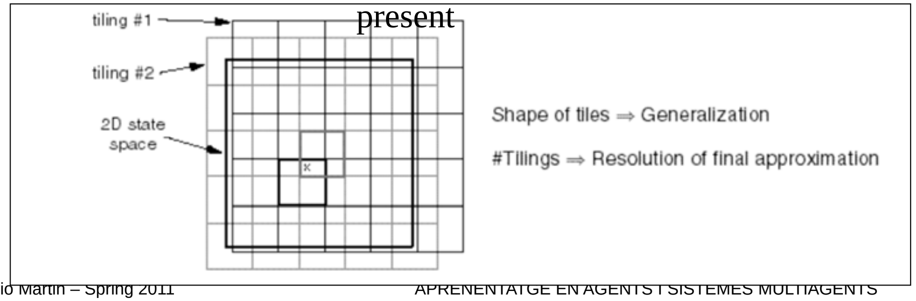
$$\phi_s(i) = \exp\left(-\frac{\left\|s - c_i\right\|^2}{2\sigma_i^2}\right)$$



Tile Coding

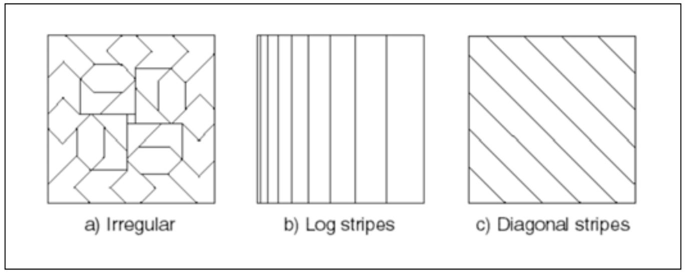


- Binary feature for each tile
- Number of features present at any one time is constant
- Binary features means weighted sum easy to compute
- Easy to compute indices of the features present

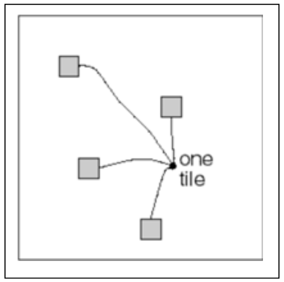


Tile Coding Cont.

Irregular tilings



Hashing



CMAC
"Cerebellar model arithmetic computer"
Albus 1971

Can you beat the "curse of dimensionality"?

- Can you keep the number of features from going up exponentially with the dimension?
- Function complexity, not dimensionality, is the problem.
- Kanerva coding:
  - Select a bunch of binary **prototypes**
  - Use hamming distance as distance measure
  - Dimensionality is no longer a problem, only complexity
- "Lazy learning" schemes:
  - Remember all the data
  - To get new value, find nearest neighbors and interpolate
  - e.g., locally-weighted regression

Neuro-Dynamic Programming Reinforcement Learning

"*It is unclear which algorithms and parameter settings will work on a particular problem, and when a method does work, it is still unclear which ingredients are actually necessary for success. As a result, applications often require trial and error in a long process of a parameter tweaking and experimentation.*"

van Roy - 2002

Value Function Approximation Convergence results

- Linear TD(λ) converges if we visit states using the on-policy distribution
- Off policy Linear TD(λ) and linear Q learning are known to diverge in some cases
- Q-learning, and value iteration used with some averagers (including k-Nearest Neighbour and decision trees) has almost sure convergence if particular exploration policies are used
- A special case of policy iteration with Sarsa style updates and linear function approximation converges

Function Approximation in RL

- Represent State by a finite number of <u>Features (Observations)</u>
- Represent Q-Function as a <u>parameterized function</u> of these features
  – (Parameter-Vector θ)
- Learn optimal parameter-vector θ^* with <u>Gradient Descent</u> Optimization at each time step

Problems of Value Function Approximation

- No <u>Convergence</u> Proofs
  – Exception: Linear Approximators
- <u>Instabilities</u> in Approximation
  – "Forgetting" of Policies
- Very high <u>Learning Time</u>
- Still it works in many Environments
  – TD-Gammon (Neural Network Approximator)

Summary of Value Function Approximation

- Generalization
- Adapting supervised-learning function approximation methods
- Gradient-descent methods
- Linear gradient-descent methods
  - Radial basis functions
  - Tile coding
  - Kanerva coding
- Nonlinear gradient-descent methods? Backpropation?
- Subleties involving function approximation, bootstrapping and the on-policy/off-policy distinction

Policy Search

- Why not search directly for a policy?
- Policy gradient methods and Evolutionary methods
- Particularly good for problems with hidden state

Approximate Reinforcement Learning

- Why?
  - To learn in reasonable time and space
    (avoid Bellman's curse of dimensionality)
  - To generalise to new situations
- Solutions
  - Approximate the value function
  - Search in the policy space

RL via Policy Search

- So far all of our RL techniques have tried to learn an exact or approximate utility function or Q-function
  - I.e. learn the optimal "value" of being in a state, or taking an action from a state.
- Another approach is to search directly in a parameterized policy space
- This general approach has the following components
  - Select a space of parameterized policies:
  - Compute the gradient of the utility function of the policy wrt parameters
  - Move parameters in the direction of the gradient
  - Repeat these steps until we reach a local maxima
- So we must answer the following questions:
  - How should we represent parameterized policies?
  - How can we compute the gradient?

Parameterized Policies

- One example of a space of parametric policies is:

$$\pi_\theta(s) = \arg\max_a \hat{Q}_\theta(s,a)$$

where $\hat{Q}_\theta(s,a)$ may be a linear function, e.g.

$$\hat{Q}_\theta(s,a) = \theta_1 f_1(s,a) + \theta_2 f_2(s,a) + \ldots + \theta_n f_n(s,a)$$

- The goal is to learn parameters θ that give a good policy
- Note that it is not important that $\hat{Q}_\theta(s,a)$ be close to the actual Q-function
  - Rather we only require $\hat{Q}_\theta(s,a)$ is good at ranking actions in order of goodness

Policy Gradient Search

- Let $\rho(\theta)$ be the value of policy π_θ.
  - $\rho(\theta)$ is just the expected discounted total reward for a trajectory of π_θ.
  - For simplicity assume each trajectory starts at a single initial state.
- Our objective is to find a θ that maximizes $\rho(\theta)$
- Policy gradient search computes the gradient $\nabla_\theta \rho(\theta)$ and then update the parameters by $\theta \leftarrow \theta + \alpha \nabla_\theta \rho(\theta)$

  we add the gradient since we are trying maximize $\rho(\theta)$
- In theory with the right learning rate schedule this will converge to a locally optimal solution
- It is rare that we can compute a closed form for the gradient, so it must be estimated

Gradient Estimation

- **Problem**: for our example parametric policy

$$\pi_\theta(s) = \arg\max_a \hat{Q}_\theta(s,a)$$

  is $\rho(\theta)$ continuous?
- No.
  - There are values of θ where arbitrarily small changes, cause the policy to change.
  - Since different policies can have different values this means that changing θ can cause discontinuous jump of $\rho(\theta)$.
- Computing or estimating the gradient of discontinuous functions can be problematic.
- What can we do about this?
  - Consider a space of parametric policies that smoothly vary with θ

Probabilistic Policies

- We would like to avoid policies that drastically change with small parameter changes
- A **probabilistic policy** π_θ is takes a state as input and returns a distribution over actions
  - Given a state s $\pi_\theta(s,a)$ returns the probability that π_θ selects action a in s
- Note that $\rho(\theta)$ is still well defined for probabilistic policies
  - Importantly if $\pi_\theta(s,a)$ is continuous relative to changing θ then $\rho(\theta)$ is also continuous
- A common form for probabilistic policies is the **softmax function**

$$\pi_\theta(s,a) = \Pr(a \mid s) = \frac{\exp\left(\hat{Q}_\theta(s,a)\right)}{\sum_{a' \in A} \exp\left(\hat{Q}_\theta(s,a')\right)}$$

Gradient Estimation

- For stochastic policies it is possible to estimate the gradient of ρ(θ) directly from trajectories of π_θ.
- First consider the simplified case where trials have length 1
  - ρ(θ) is just the expected discounted total reward for a trajectory of π_θ.
  - For simplicity assume each trajectory starts at a single initial state.

$$\nabla_\theta \rho(\theta) = \nabla_\theta \sum_a \pi_\theta(s_o, a) R(a) = \sum_a \left(\nabla_\theta \pi_\theta(s_o, a)\right) R(a)$$

where s_0 is the initial state, and R(a) is reward received after taking action a. A simple rewrite gives,

$$\nabla_\theta \rho(\theta) = \sum_a \pi_\theta(s_o, a) \underbrace{\frac{\left(\nabla_\theta \pi_\theta(s_o, a)\right) R(a)}{\pi_\theta(s_o, a)}}_{\text{can get closed form } f(s_0,a)}$$

- Estimate the gradient by estimating the expected value of $f(s_0,a)$!

Gradient Estimation

$$\nabla_\theta \rho(\theta) = \sum_a \pi_\theta(s_o, a) \underbrace{\frac{\left(\nabla_\theta \pi_\theta(s_o, a)\right)}{\pi_\theta(s_o, a)}}_{\text{can get closed form } f(s_0,a)} R(a)$$

- Estimate the gradient by estimating the expected value of f(s0,a)R(a) !
- We already learned how to estimate expected values by sampling (just average a set of N samples)

$$\nabla_\theta \rho(\theta) \approx \frac{1}{N} \sum_{j=1}^{N} f(s_o, a_j) R(a)$$

Gradient Estimation

- So for the case of a length 1 trajectories we got:

$$\nabla_\theta \rho(\theta) \approx \frac{1}{N} \sum_{j=1}^{N} f(s_o, a_j) R(a)$$

- For the general case where trajectories have length greater than 1 we get:

$$\nabla_\theta \rho(\theta) \approx \frac{1}{N} \sum_{j=1}^{N} \sum_{t=1}^{T_j} f(s_{j,t}, a_{j,t}) R_j(s_{j,t})$$

Total reward in trial j
from step t to end

- This gradient estimation converges rather slowly. There have been many recent improvements.

Policy Gradient Theorem[1]

- Theorem:

  If the value-function parameterization is *compatible* with the policy parameterization, then the true policy gradient can be estimated, the *variance of the estimation* can be controlled by a reinforcement baseline, and policy iteration *converges to a locally optimal* policy.

- Significance:
  - Shows first convergence proof for policy iteration with function approximation.

[1] Sutton,McAllester, Singh, Mansour: Policy Gradient Methods for RL with Function Approximation

What else exists?

- Memory-based RL
- Fuzzy RL
- Multi-objective RL
- Inverse RL
- ...

- Could all be used for Motor Learning

Memory-based RL

- Use a short-term Memory to store important Observations over a long time
  - Overcome Violations of Markov Property
  - Avoid storing finite histories

- Memory Bits [Peshkin et.al.]
  - Additional Actions that change memory bits
- Long Short-Term Memory [Bakker]
  - Recurrent Neural Networks

Fuzzy RL

- Learn a Fuzzy Logic Controller via Reinforcement Learning [Gu, Hu]

- Optimize Parameters of Membership Functions and Composition of Fuzzy Rules

- Adaptive Heuristic Critic Framework

Inverse RL

- Learn the Reward Function from observation of optimal Policy [Russell]
  - Goal: Understand, which optimality principle underlies a policy

- Problems:
  - Most algorithms need full policy (not trajectories)
  - Ambiguity: Many different reward functions could be responsible for the same policy

- Few results exist until now

Multi-objective RL

- Reward-Function is a Vector
  - Agent has to fulfill multiple tasks (e.g. reach goal and stay alive)
  - Makes design of Reward function more natural
- Algorithms are complicated and make strong assumptions
  - E.g. total ordering on reward vectors [Gabor]
  - Game theoretic Principles [Shelton]