Review II

CMPSCI 383
December 8, 2011

General Information about the Final

- Closed book closed notes
- Includes midterm material too
- But expect more emphasis on later material

What you should know

Chapter 16: Utility Theory

- Maximum expected utility (MEU) principle
- Utility and preferences: rational preferences
- Utility of money
  - Risk averse
  - Risk seeking
- What do humans do?
- Multiattribute utility functions
  - Pure dominance
  - Preference structure
- Decision networks

MEU

$$EU(a \mid e) = \sum_{s'} P(\text{Result}(a) = s' \mid a, e) U(s')$$

$$\text{action} = \arg\max_{a} EU(a \mid e)$$

Constraints

Idea: preferences of a rational agent must obey constraints.
Rational preferences $\Rightarrow$
behavior describable as maximization of expected utility

Constraints:

Orderability

$$(A \succ B) \vee (B \succ A) \vee (A \sim B)$$

Transitivity

$$(A \succ B) \wedge (B \succ C) \Rightarrow (A \succ C)$$

Continuity

$$A \succ B \succ C \Rightarrow \exists p \ [p, A;\ 1-p, C] \sim B$$

Substitutability

$$A \sim B \Rightarrow [p, A;\ 1-p, C] \sim [p, B; 1-p, C]$$

Monotonicity

$$A \succ B \Rightarrow (p \geq q \Leftrightarrow [p, A;\ 1-p, B] \succsim [q, A;\ 1-q, B])$$

You don't have to memorize these, but understand them.

Relationship between preferences and utilities

Theorem (Ramsey, 1931; von Neumann and Morgenstern, 1944): Given preferences satisfying the constraints there exists a real-valued function U such that

$$U(A) \geq U(B) \quad \Leftrightarrow \quad A \succsim B$$

$$U([p_1, S_1; \ \ldots \ ; \ p_n, S_n]) = \sum_i p_i U(S_i)$$

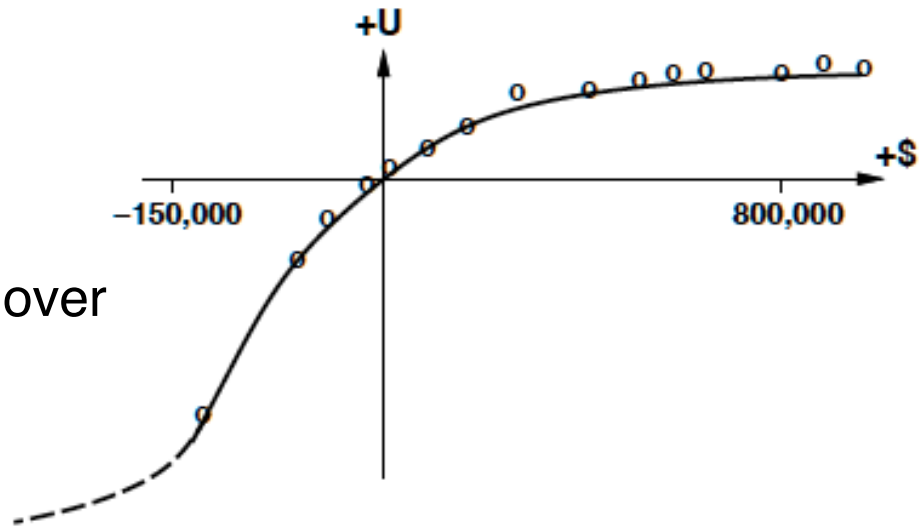
Money and Risk

Money does not behave as a utility function

Given a lottery L with expected monetary value $EMV(L)$, usually $U(L) < U(EMV(L))$, i.e., people are risk-averse

Prefer a sure thing with a payoff less than the EMV

Typical empirical data, extrapolated with risk-prone behavior:



Prefer a gamble L over EMV(L)

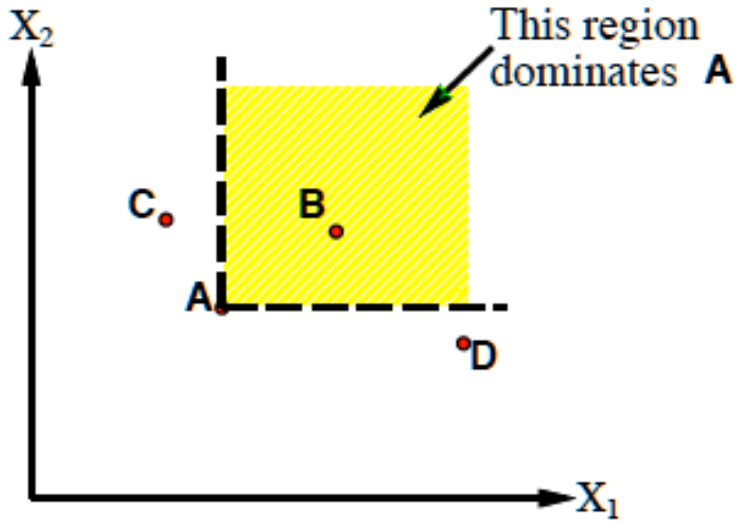
Humans?

- Normative theory
- Descriptive theory
- Give an example of “irrational” human behavior.
  - Certainty effect
  - Ambiguity effect
  - Framing effect
  - Anchoring effect
- Tversky and Kahneman

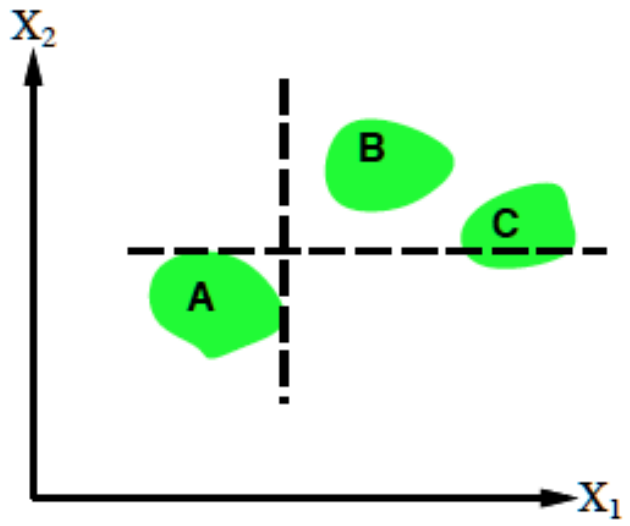
Multiattribute Utility

Typically define attributes such that U is monotonic in each

Strict dominance: choice B strictly dominates choice A iff
$\forall i \; X_i(B) \geq X_i(A)$ (and hence $U(B) \geq U(A)$)



Deterministic attributes



Uncertain attributes

Strict dominance seldom holds in practice

Additive value functions

- What is?

$$V(S) = \sum_i V_i(X_i(S))$$

- Why?
  - Often a good way to describe preferences when multiple attributes are involved

What you don't need to know for the exam

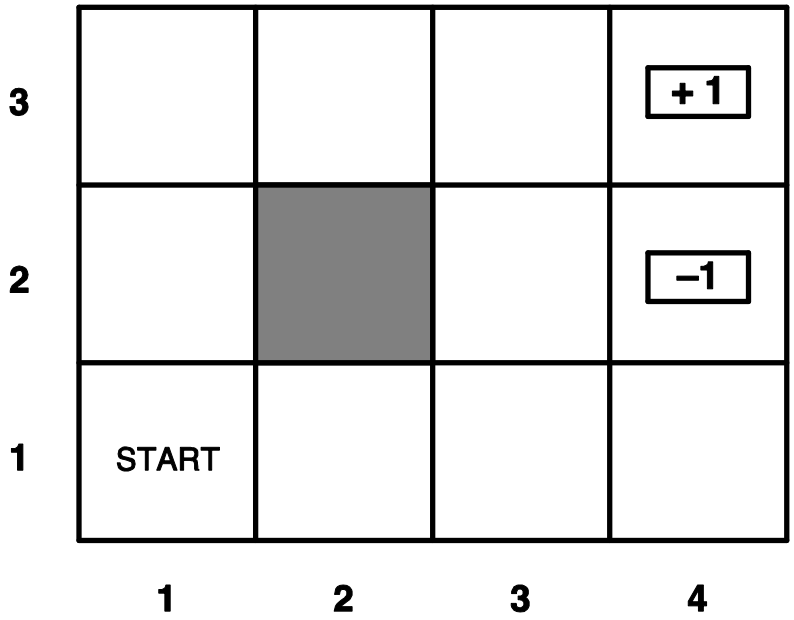
- The 6 constraints for preference relation (but be able to understand one if given)
- Micromort, QALY
- Certainty equivalent, insurance premium
- Optimizer's curse
- Stochastic dominance
- Mutual preference independence
- Preferences with uncertainty (sec. heading on p. 625)
- Secs. 16.5-16.7

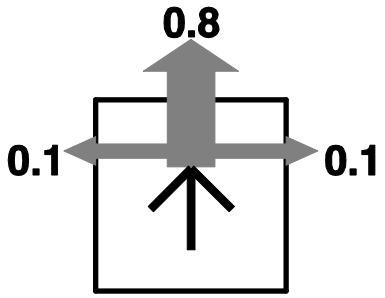
Chapter 17: Making Complex Decisions

- Sequential decision problems: MDPs
- Policy
- Optimal policy
- Utilities over time
  - Additive
  - Discounted
- Utilities of states
  - Policy Evaluation
- Bellman equation
- Value Iteration
- Policy Iteration
  - Modified policy iteration
  - Asynchronous policy iteration

A Simple Example

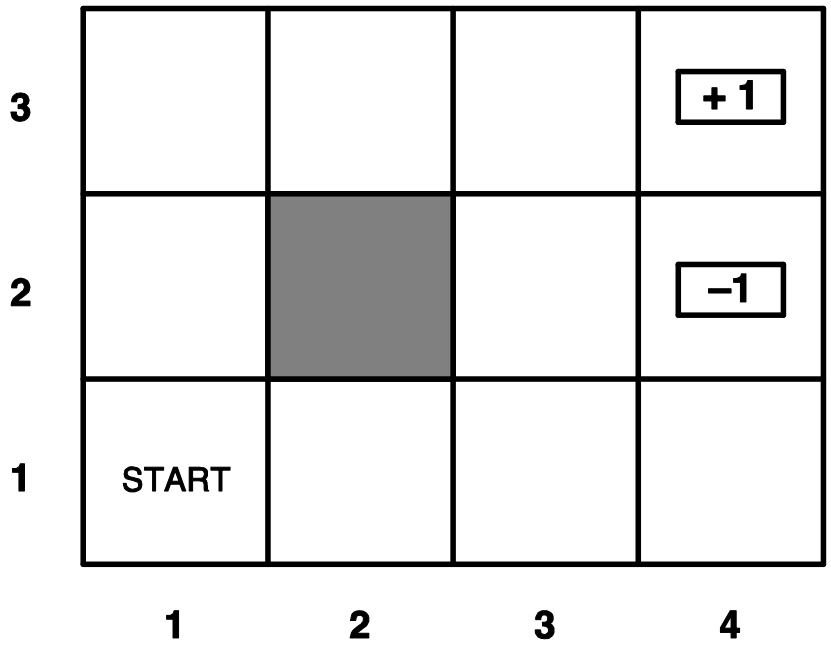
- “Gridworld” with 2 Goal states
- Actions: Up, Down, Left, Right
- Fully observable: Agent knows where it is

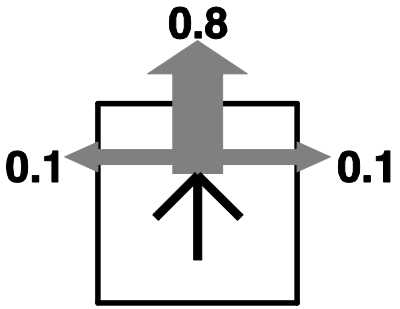




Transition Model

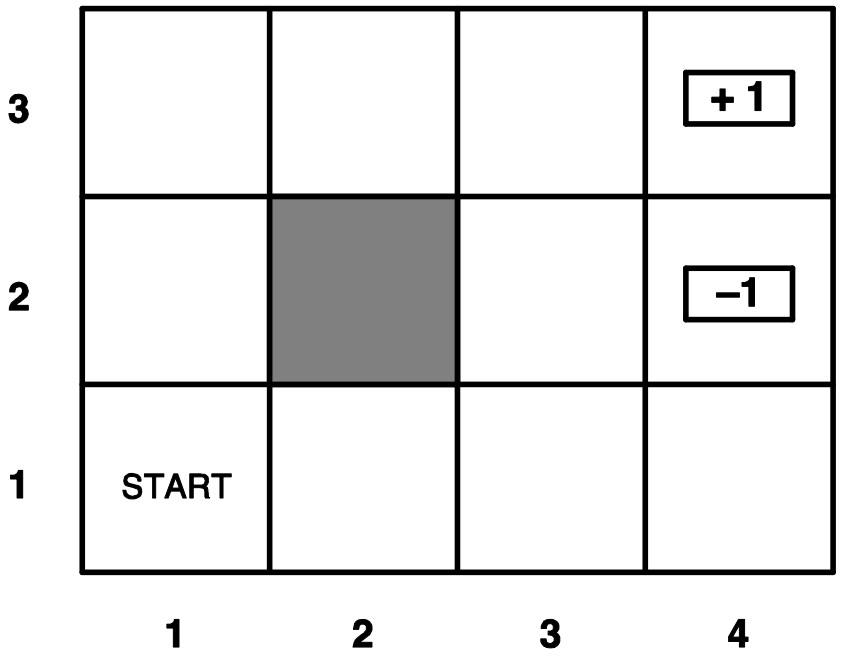
$$P(s' \mid s, a)$$

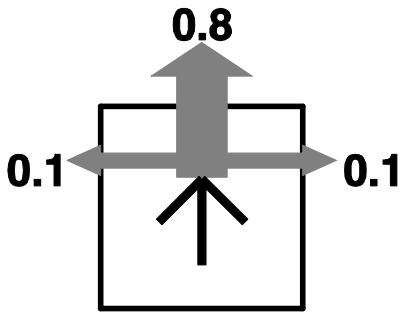




Markov Assumption

$$P(s' \mid s, a)$$





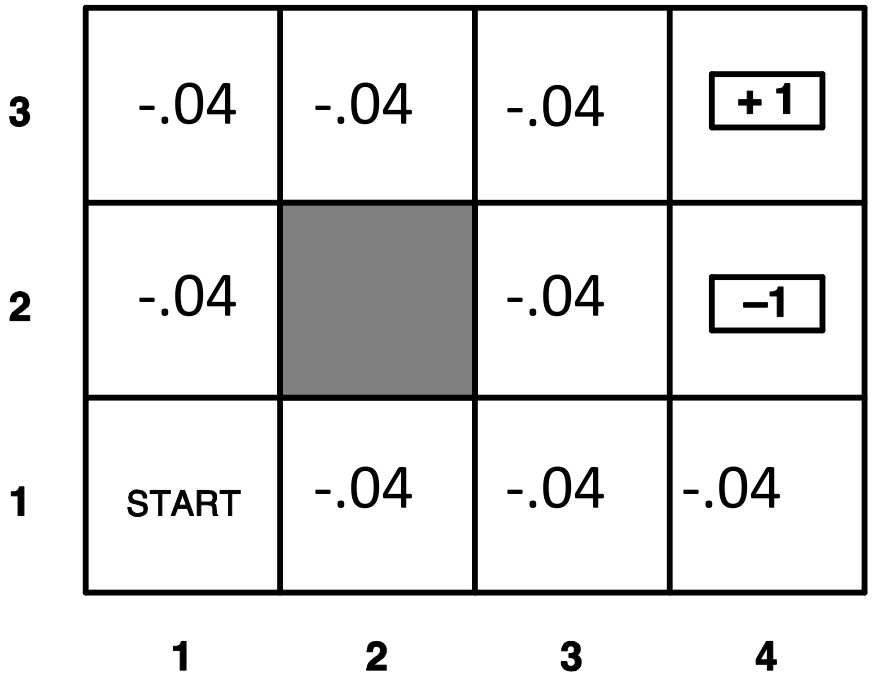
Agent's Utility Function

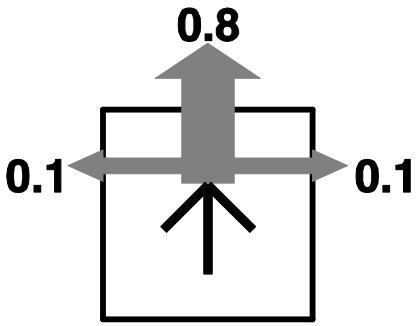
- Performance depends on the entire sequence of states and actions.
  - "Environment history"
- In each state, the agent receives a reward $R(s)$.
  - The reward is real-valued. It may be positive or negative.
- Utility of environment history = sum of reward received.

Reward function

$$P(s' \mid s, a)$$

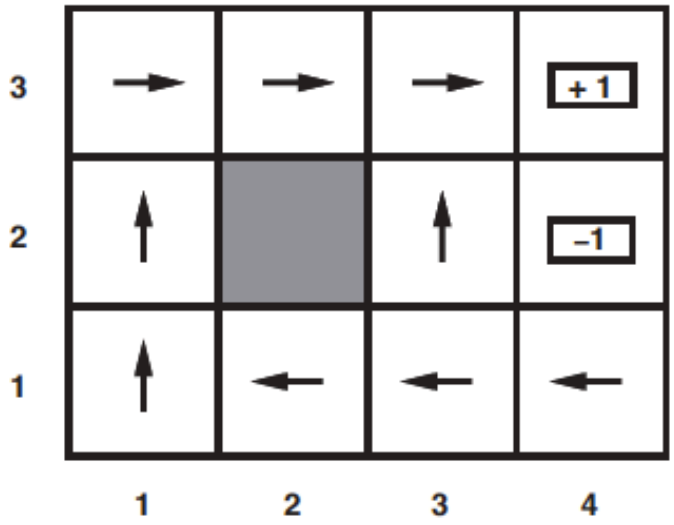
$$R(s)$$

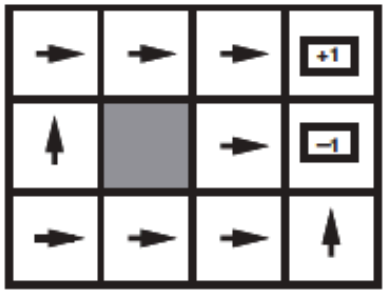




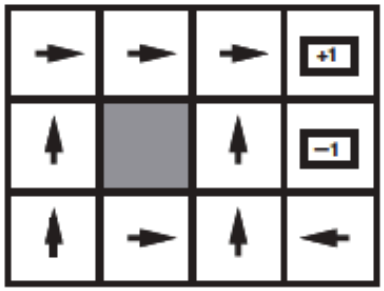
Decision Rules

- Decision rules say what to do in each state.
- Often called policies, π.
- Action for state s is given by $\pi(s)$.

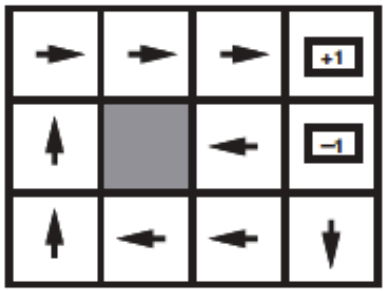




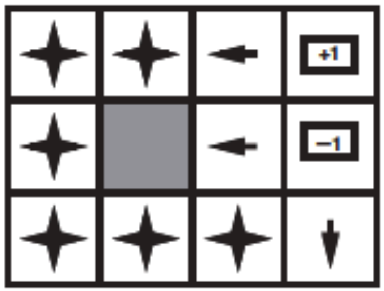
$R(s) < -1.6284$



$-0.4278 < R(s) < -0.0850$



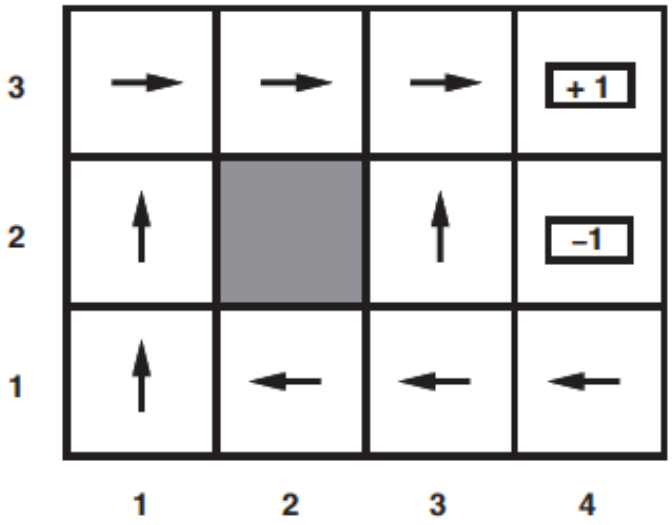
$-0.0221 < R(s) < 0$

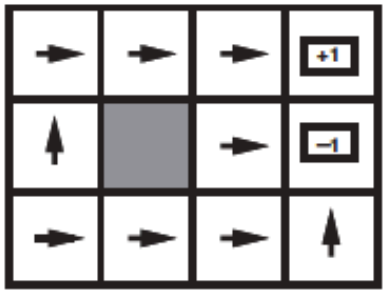


$R(s) > 0$

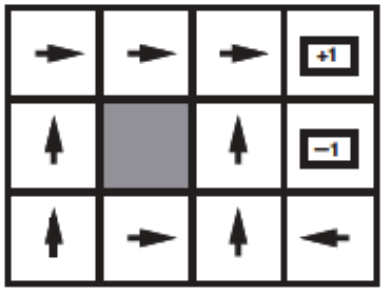
Our Goal

- Find the policy that maximizes the expected sum of rewards.
- Called an *optimal policy*.

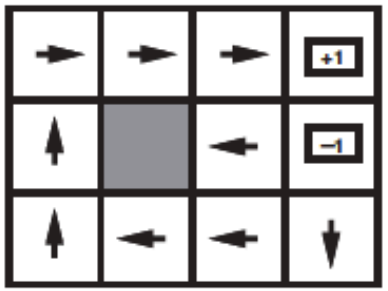




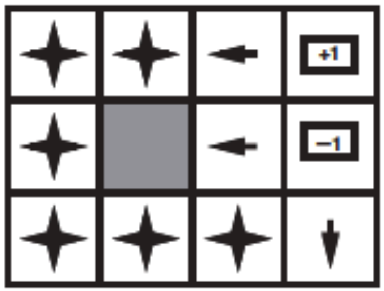
$R(s) < -1.6284$



$-0.4278 < R(s) < -0.0850$



$-0.0221 < R(s) < 0$



$R(s) > 0$

Markov Decision Process (MDP)

- M=(S,A,P,R)
- S = set of possible states
- A = set of possible actions
- P(s'|s,a) gives transition probabilities
- R = reward function

- Goal: find an optimal policy, π^*.

Finite/Infinite Horizon

- Finite horizon: the “game” ends after N steps.
- Infinite horizon: the “game” never ends
- “With a finite horizon, the optimal action in a given state could change over time.”
  - The optimal policy is *nonstationary*.
- With infinite horizon, the optimal policy is *stationary*.

Utilities over time

- Additive rewards:

$$U_h([s_0, s_1, s_2, \ldots]) = R(s_0) + R(s_1) + R(s_2) + \ldots$$

- Discounted rewards:

$$U_h([s_0, s_1, s_2, \ldots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots$$

- Discount factor:

$$\gamma \in [0,1]$$

Utility of States

- Given a policy, we can define the utility of a state:

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^{t} R(s_t)\right]$$

Policy Evaluation

- Finding the utility of states for a given policy.
- Solve a system of linear equations:

$$U^{\pi}(s) = R(s) + \gamma \sum_{s'} P(s' \mid s, \pi(s)) U^{\pi}(s')$$

- An instance of a “Bellman Equation”.

Policy Evaluation

- The *n* linear equations can be solved in $O(n^3)$ with standard linear algebra methods.
- If $O(n^3)$ is still too much, we can do it iteratively:

$$U_{i+1}^{\pi} \leftarrow R(s) + \gamma \sum_{s'} P(s' \mid s, \pi(s)) U_i^{\pi}(s')$$

- An instance of value iteration---for a fixed policy

Optimal Policy

$$\pi_s^* = \arg\max_{\pi} U^{\pi}(s)$$

- Optimal policy doesn't depend on what state you start in (for infinite horizon discounted case).
- Optimal policy: π^*
- True utility of a state:

$$U^{\pi^*}(s) = U(s)$$

"optimal value function"

Selecting optimal actions

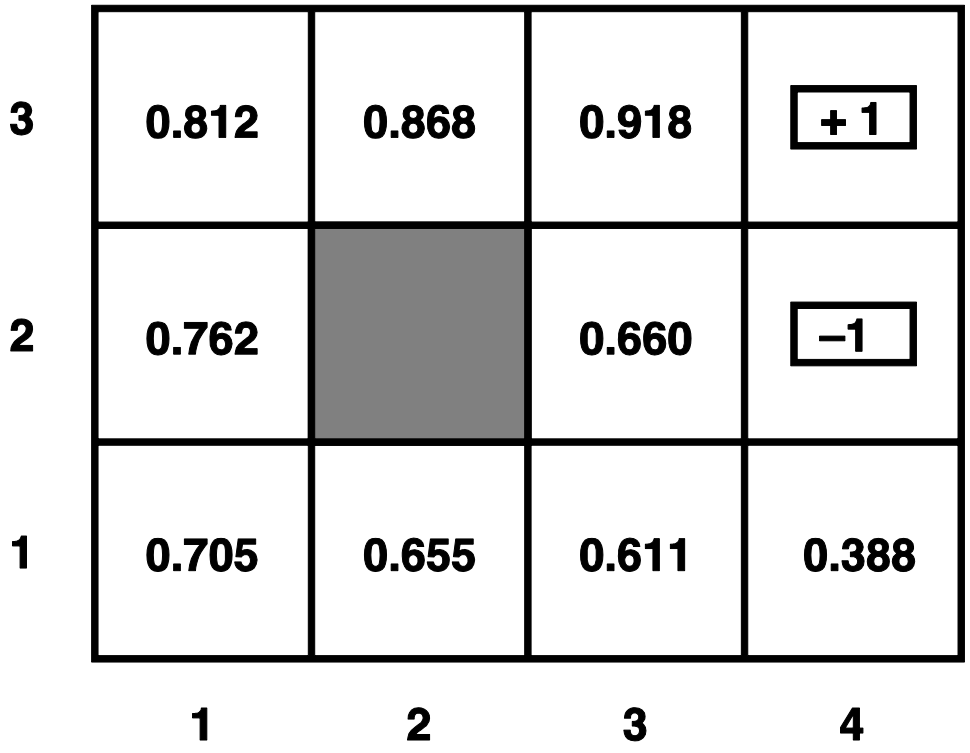
- Given the true $U(s)$ values, how can we select actions? (Maximum expected utility – MEU)

$$a_t = \arg\max_{a \in A(s)} \sum_{s'} P(s' \mid s_t, a) U(s')$$

Utility and Rewards

- Utility = long term total reward from s onwards
- Reward = short term reward from s

Utility



Searching for Optimal Policies

- Bellman Equation

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U(s')$$

- If we write out the Bellman equation for all n states, we get n equations, with n unknowns: $U(s)$.
- We can solve this system of equations to determine the Utility of every state.

Value Iteration

- The equations are non-linear, so we can't use standard linear algebra methods.

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U(s')$$

- Value iteration: start with random initial values for each U(s), iteratively update each value to fit the fight-hand side of the equation:

$$U_{i+1} \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U_i(s')$$

Value Iteration

- The update is applied simultaneously to every state.
- If this update is applied infinitely often, we are guaranteed to find the true $U(s)$ values.
  - There is one unique solution
- Given the true $U(s)$ values, how can we select actions? (Maximum expected utility – MEU)

$$a_t = \arg\max_{a \in A(s)} \sum_{s'} P(s' \mid s_t, a) U_i(s')$$

Policy Iteration

- Policy iteration interleaves two steps:
  - Policy evaluation: Given a policy, compute the utility of each state for that policy
    - Modified policy iteration: don't do this to completion
  - Policy improvement: Calculate a new MEU policy
- Terminate when the policy doesn't change the utilities.
- Guaranteed to converge to an optimal policy

Asynchronous Policy Iteration

- We said the utility of every state is updated simultaneously. This isn't necessary.
- You can pick and subset of the states and apply either policy improvement or value iteration to that subset.
- Given certain conditions, this is also guaranteed to converge to an optimal policy.

What you don't have to know for the exam

- Average reward case
- Convergence details for VI: contractions (Sec. 17.2.3)
- Secs. 17.4 -17.6: POMDPs, Games, Mechanism design

Chapter 18: Learning from Examples

- Types of learning
- Supervised learning
  - Decision tree induction
- Univariate Linear Regression
  - Batch gradient descent
  - Stochastic gradient descent
- Multivariate Linear Regression
  - Regularization
- Linear Classifiers
  - Perceptron learning rule
- Logistic Regression

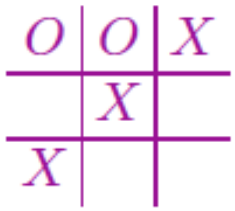
Types of learning

- Unsupervised learning
- Reinforcement learning
- Supervised learning
- Semi-supervised learning

Supervised Learning

Simplest form: learn a function from examples (tabula rasa)

f is the target function

An example is a pair x, $f(x)$, e.g.,  , $+1$

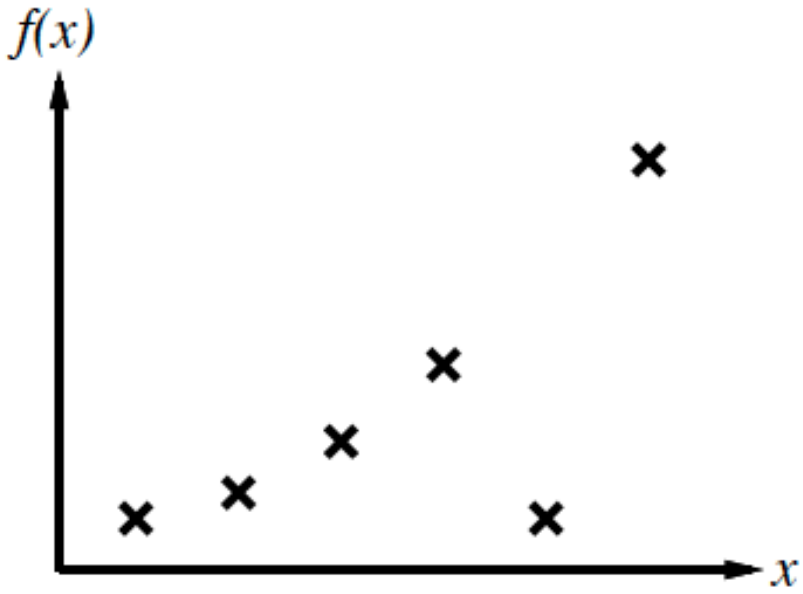
Problem: find a(n) hypothesis h
such that $h \approx f$
given a training set of examples

(This is a highly simplified model of real learning:
- Ignores prior knowledge
- Assumes a deterministic, observable "environment"
- Assumes examples are given
- Assumes that the agent wants to learn f—why?)

Supervised Learning

Construct/adjust h to agree with f on training set
(h is consistent if it agrees with f on all examples)

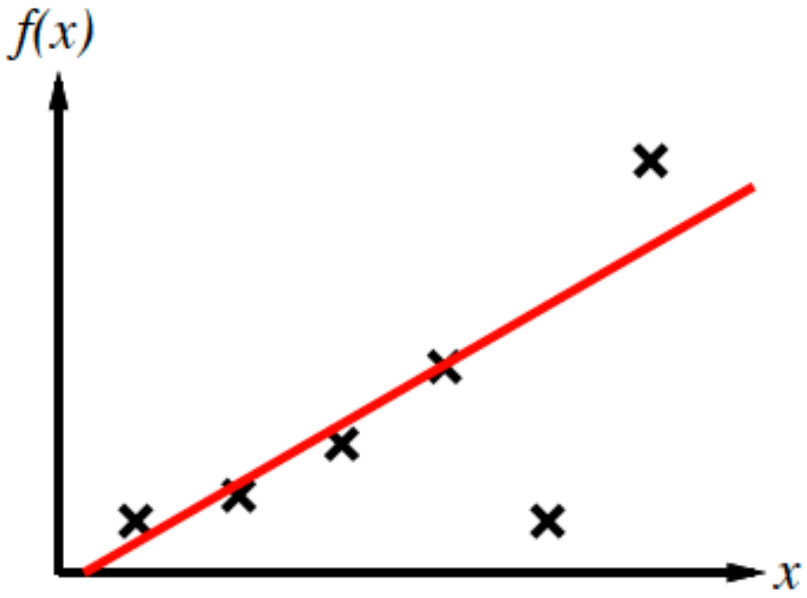
E.g., curve fitting:



Supervised Learning

Construct/adjust h to agree with f on training set
(h is consistent if it agrees with f on all examples)

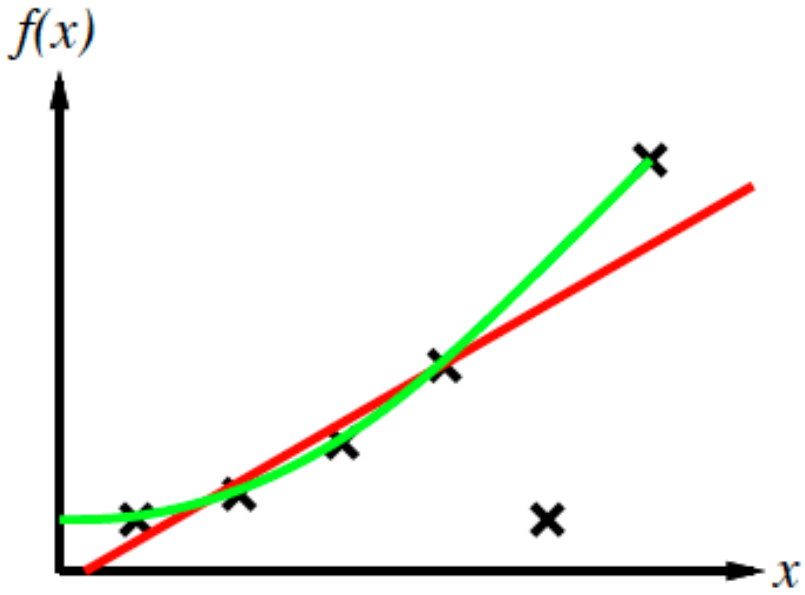
E.g., curve fitting:



Supervised Learning

Construct/adjust h to agree with f on training set
(h is consistent if it agrees with f on all examples)

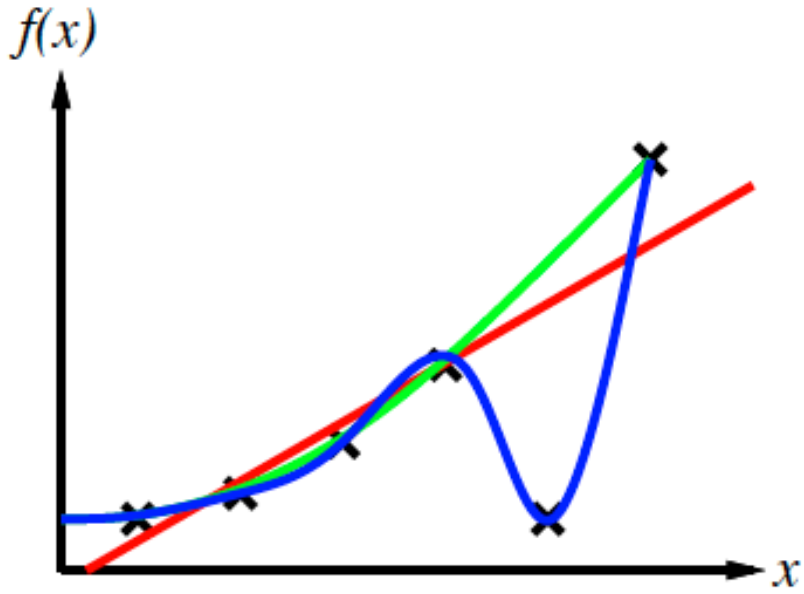
E.g., curve fitting:



Supervised Learning

Construct/adjust h to agree with f on training set
(h is consistent if it agrees with f on all examples)

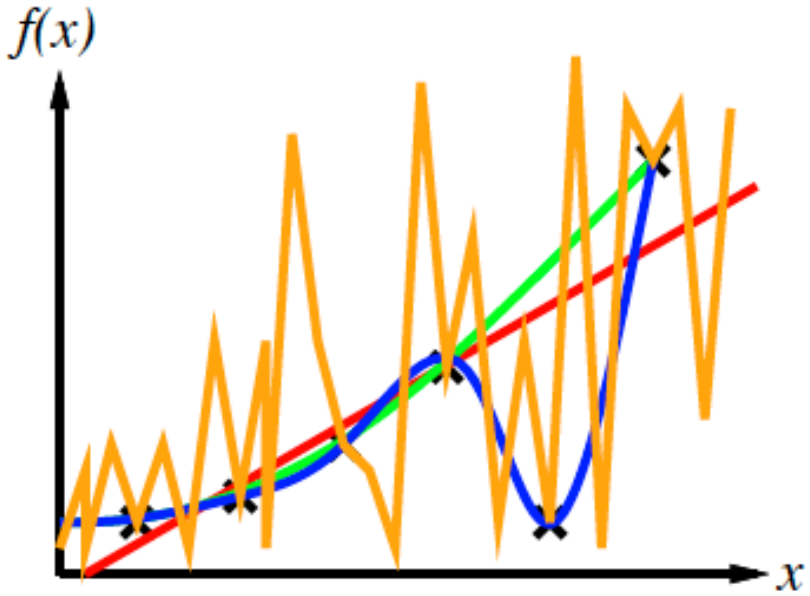
E.g., curve fitting:



Supervised Learning

Construct/adjust h to agree with f on training set
(h is consistent if it agrees with f on all examples)

E.g., curve fitting:

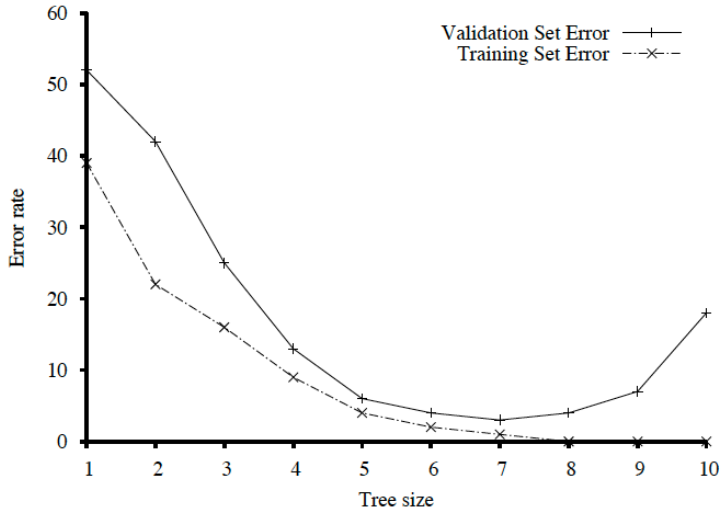


Terms to know for supervised learning

- Training set
- Test set
- Generalization
- Classification
- Regression
- Hypothesis space
- Consistent hypothesis
- Ockham's razor
- Overfitting
- Cross-validation
- Regularization
- Model selection
- Loss function
- Generalization loss
- Empirical loss

Important issues

- Generalization
- Overfitting
- Cross-validation
  - Holdout cross validation
  - K-fold cross validation
  - Leave-one-out cross-validation
- Model selection



Recall Notation

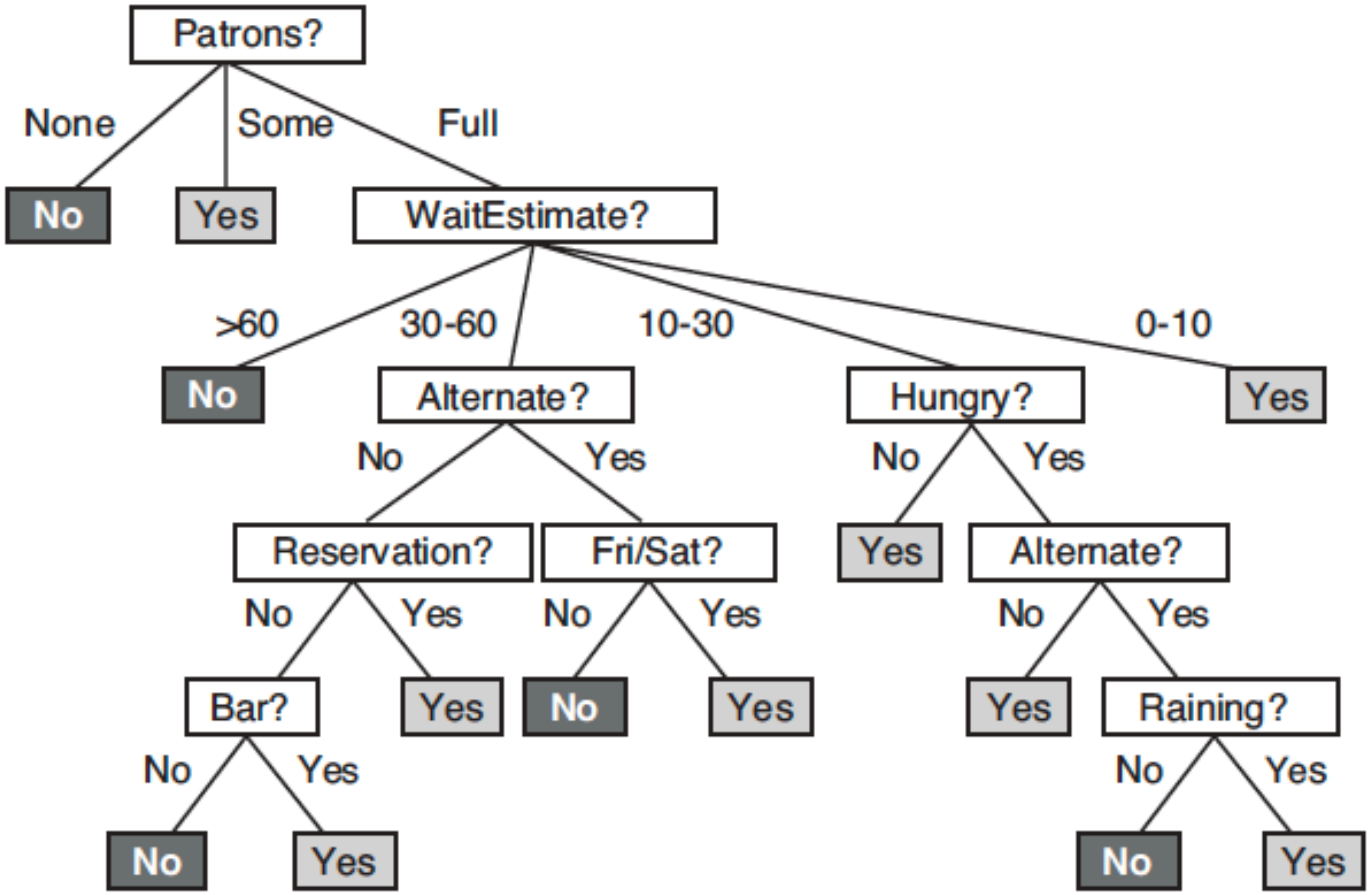
$(x_1, y_1), (x_2, y_2), \ldots (x_N, y_N)$ training set

Where each y_j was generated by an unknown function $y = f(\mathbf{x})$

Discover a function h that best approximates the true function f

hypothesis

Decision trees



Expressiveness?

- Consider only Boolean case
- How many Boolean functions are there of n Boolean attributes?
- Functions that can't be compactly represented by a decision tree?

Splitting the examples

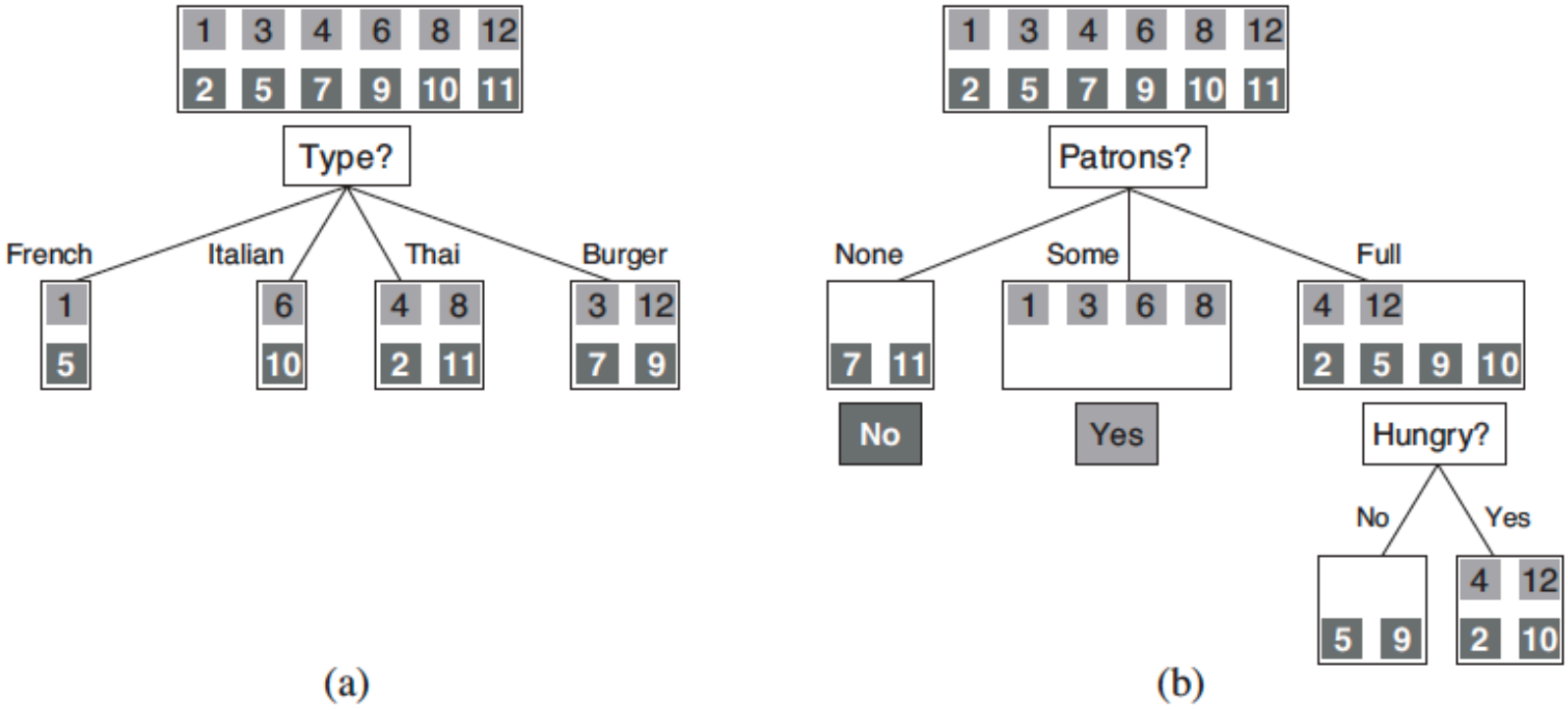


Figure 18.4 **FILES: figures/restaurant-stub.eps (Tue Nov 3 16:23:28 2009).** Splitting the examples by testing on attributes. At each node we show the positive (light boxes) and negative (dark boxes) examples remaining. (a) Splitting on *Type* brings us no nearer to distinguishing between positive and negative examples. (b) Splitting on *Patrons* does a good job of separating positive and negative examples. After splitting on *Patrons*, *Hungry* is a fairly good second test.

Final decision tree from the examples

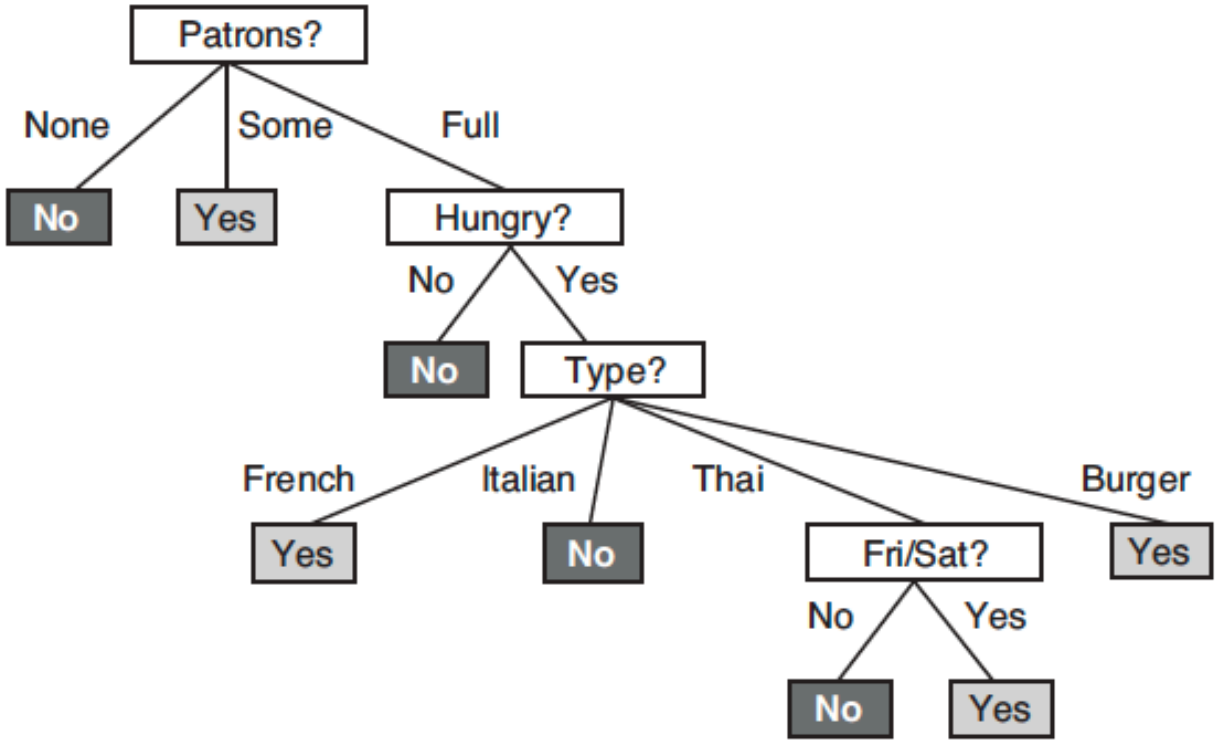


Figure 18.6 FILES: figures/induced-restaurant-tree.eps (Tue Nov 3 16:23:04 2009). The decision tree induced from the 12-example training set.

Choosing Attribute Tests

- Entropy
  - What does it measure?
- Information gain
  - What does it measure?

Loss Functions

Suppose the true prediction for input $\mathbf{x}$ is $f(\mathbf{x}) = y$
but the hypothesis gives $h(\mathbf{x}) = \hat{y}$

$L(\mathbf{x}, y, \hat{y}) = Utility(\text{result of using } y \text{ given input } \mathbf{x})$
$\quad - Utility(\text{result of using } \hat{y} \text{ given input } \mathbf{x})$

Simplified version: $L(y, \hat{y})$

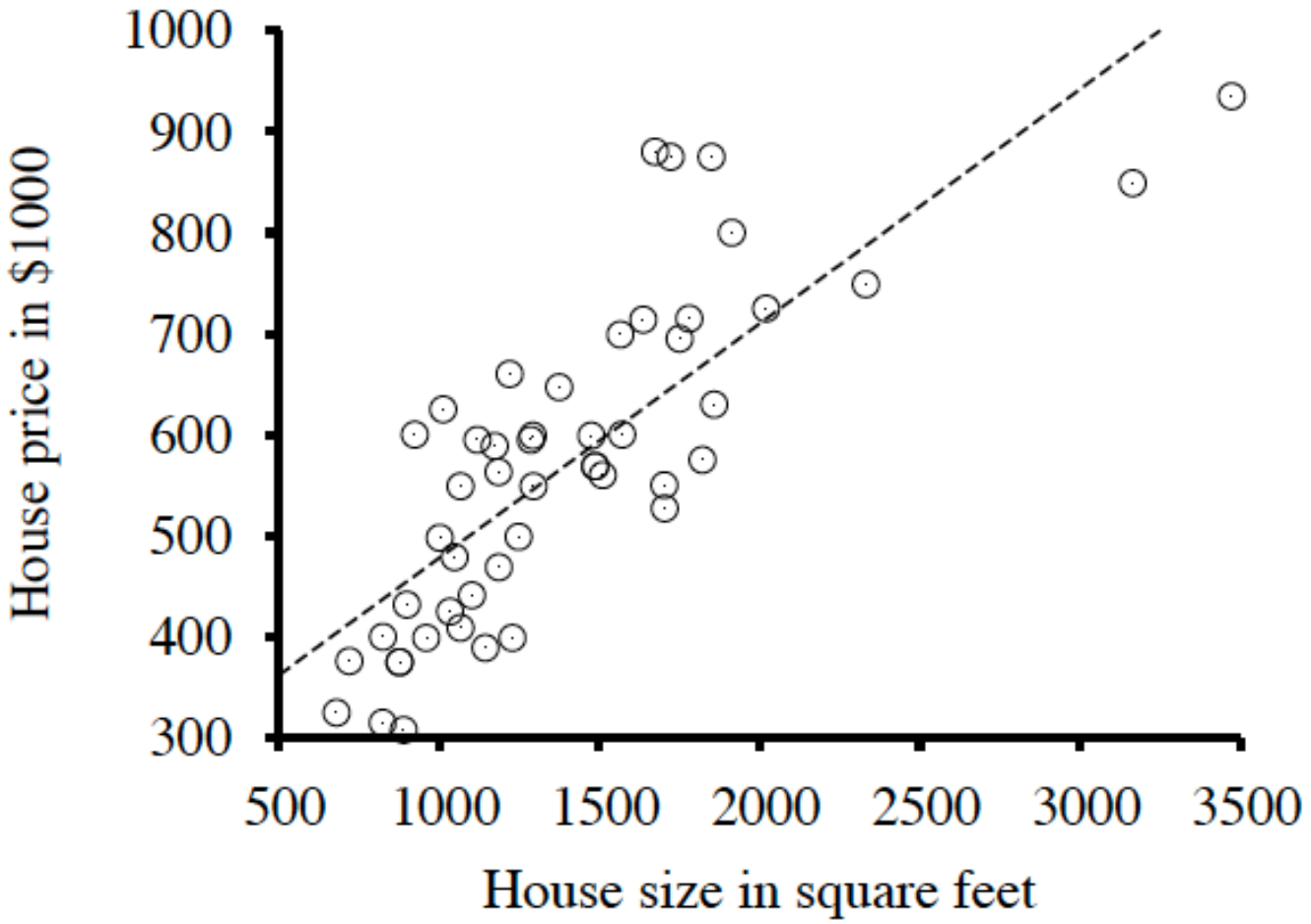
Absolute value loss: $L_1(y, \hat{y}) = |y - \hat{y}|$

Squared error loss: $L_2(y, \hat{y}) = (y - \hat{y})^2$

0/1 loss: $L_{0/1}(y, \hat{y}) = 0$ if $y = \hat{y}$, else 1

Generalization loss: expected loss over all possible examples
Empirical loss: average loss over available examples

Univariate Linear Regression



Univariate Linear Regression contd.

$\mathbf{w} = [w_0, w_1]$ weight vector

$h_{\mathbf{w}}(x) = w_1 x + w_0$

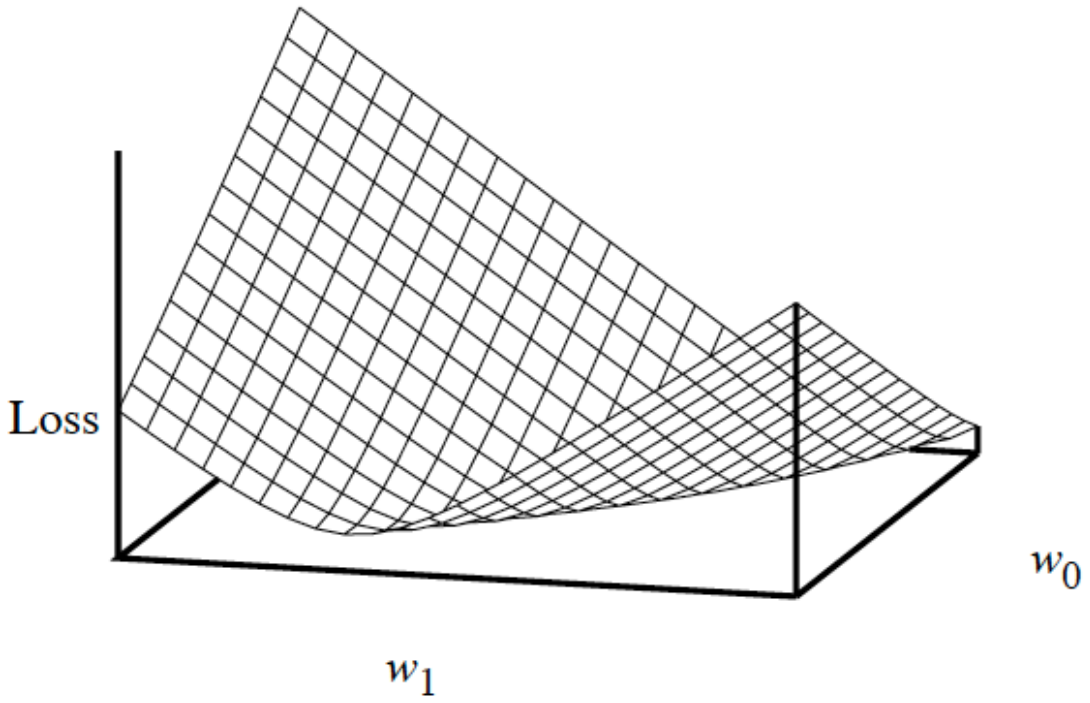
Find weight vector that minimizes empirical loss, e.g., L_2:

$$Loss(h_{\mathbf{w}}) = \sum_{j=1}^{N} L_2(y_j, h_{\mathbf{w}}(x_j)) = \sum_{j=1}^{N} (y_j - h_{\mathbf{w}}(x_j))^2 = \sum_{j=1}^{N} (y_j - (w_1 x_j + w_0))^2$$

i.e., find $\mathbf{w}^*$ such that

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} Loss(h_{\mathbf{w}})$$

Weight Space



Finding w*

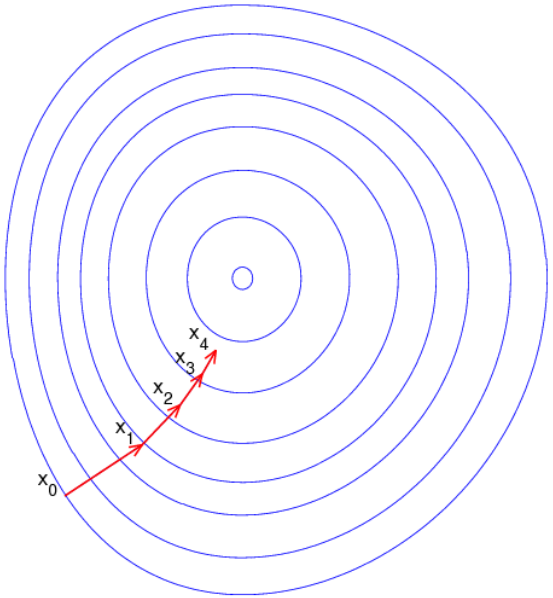
Find weights such that:

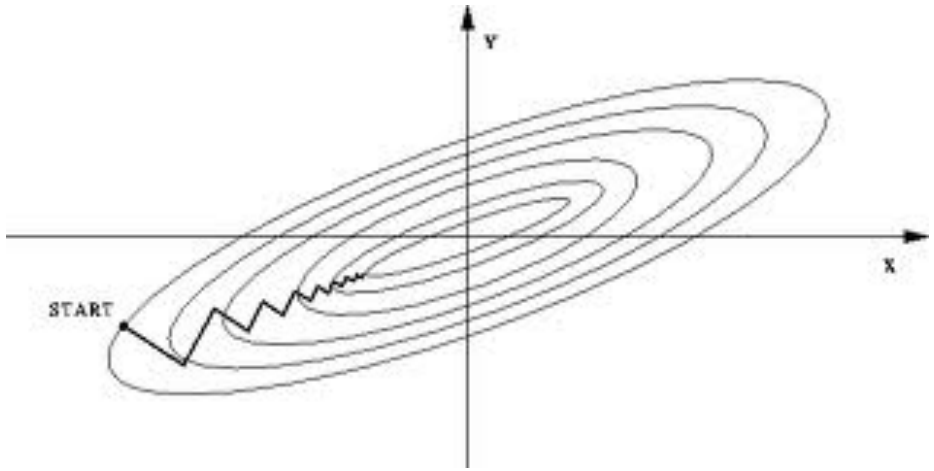
$$\frac{\partial}{\partial w_0}\sum_{j=1}^{N}(y_j-(w_1x_j+w_0))^2=0 \quad \text{and} \quad \frac{\partial}{\partial w_1}\sum_{j=1}^{N}(y_j-(w_1x_j+w_0))^2=0$$

Gradient Descent

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} Loss(\mathbf{w})$$

step size or learning rate





Gradient Descent contd.

For one training example (x, y):

$$w_0 \leftarrow w_0 + \alpha(y - h_{\mathbf{w}}(x)) \quad \text{and} \quad w_1 \leftarrow w_1 + \alpha(y - h_{\mathbf{w}}(x))x$$

For *N* training examples:

$$w_0 \leftarrow w_0 + \alpha \sum_j (y_j - h_{\mathbf{w}}(x_j)) \quad \text{and} \quad w_1 \leftarrow w_1 + \alpha \sum_j (y_j - h_{\mathbf{w}}(x_j))x_j$$

batch gradient descent

stochastic gradient descent: take a step for one training example at a time

The Multivariate case

$$h_{sw}(\mathbf{x}_j) = w_0 + w_1 x_{j,1} + \cdots + w_n x_{j,n} = w_0 + \sum_i w_i x_{j,i}$$

Augmented vectors: add a feature to each $\mathbf{x}$ by tacking on a 1: $x_{j,0} = 1$

Then:

$$h_{sw}(\mathbf{x}_j) = \mathbf{w} \cdot \mathbf{x}_j = \mathbf{w}^T \mathbf{x}_j = \sum_i w_i x_{j,i}$$

And batch gradient descent update becomes:

$$w_i \leftarrow w_i + \alpha \sum_j (y_j - h_{\mathbf{w}}(\mathbf{x}_j)) x_{j,i}$$

Done after each example: Widrow-Hoff rule

The Multivariate case contd.

Or, solving analytically:

Let $\mathbf{y}$ be the vector of outputs for the training examples

$\mathbf{X}$ data matrix: each row is an input vector

Solving this for $\mathbf{w}^*$: $\mathbf{y} = \mathbf{X}\mathbf{w}$

$$\mathbf{w}^* = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{y}$$

pseudo inverse

Regularization

The process of explicitly penalizing hypothesis complexity

$$Cost(h) = EmpLoss(h) + \lambda Complexity(h)$$

$$\text{For example : } Complexity(h_{\mathbf{w}}) = L_q(\mathbf{w}) = \sum_i |w_i|^q$$

Linear Classification: hard thresholds

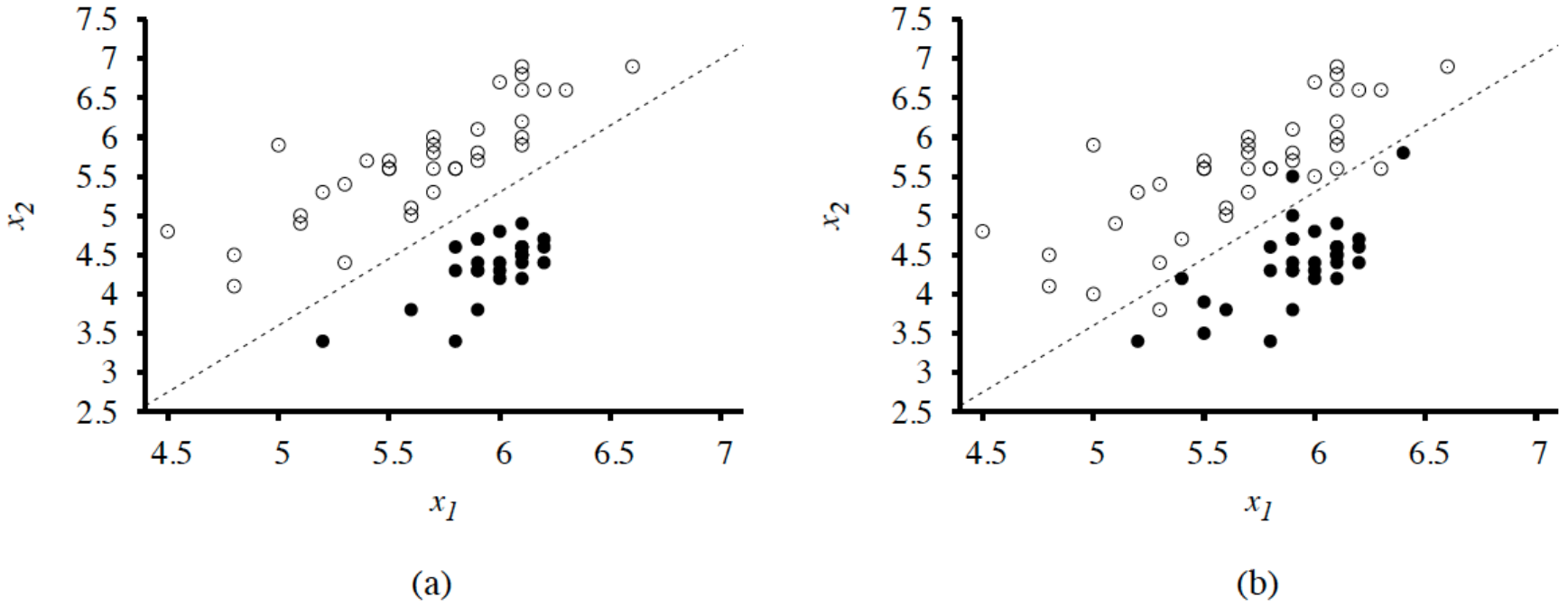
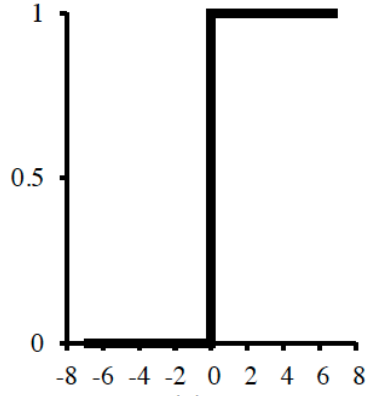


Figure 18.15 **FILES: .** (a) Plot of two seismic data parameters, body wave magnitude x_1 and surface wave magnitude x_2, for earthquakes (white circles) and nuclear explosions (black circles) occurring between 1982 and 1990 in Asia and the Middle East (?). Also shown is a decision boundary between the classes. (b) The same domain with more data points. The earthquakes and explosions are no longer linearly separable.

Linear Classification: hard thresholds contd.

- Decision Boundary:
  - In linear case: linear separator, a hyperplane
- Linearly separable:
  - data is linearly separable if the classes can be separated by a linear separator
- Classification hypothesis:

$$h_{\mathbf{w}}(x) = Threshold(\mathbf{w} \cdot \mathbf{x}) \quad \text{where} \quad Threshold(z) = 1 \text{ if } z \geq 0 \text{ and } 0 \text{ otherwise}$$



Perceptron Learning Rule

For a single sample $(\mathbf{x}, y)$:

$$w_i \leftarrow w_i + \alpha\left(y - h_{\mathbf{w}}(\mathbf{x})\right)x_i$$

- If the output is correct, i.e., y $=h_{\mathbf{w}}(\mathbf{x})$, then the weights don't change
- If $y = 1$ but $h_{\mathbf{w}}(\mathbf{x}) = 0$, then w_i is *increased* when x_i is positive and *decreased* when x_i is negative.
- If $y = 0$ but $h_{\mathbf{w}}(\mathbf{x}) = 1$, then w_i is *decreased* when x_i is positive and *increased* when x_i is negative.

Perceptron Convergence Theorem: For any data set that's linearly separable and any training procedure that continues to present each training example, the learning rule is guaranteed to find a solution in a finite number of steps.

Linear Classification with Logistic Regression

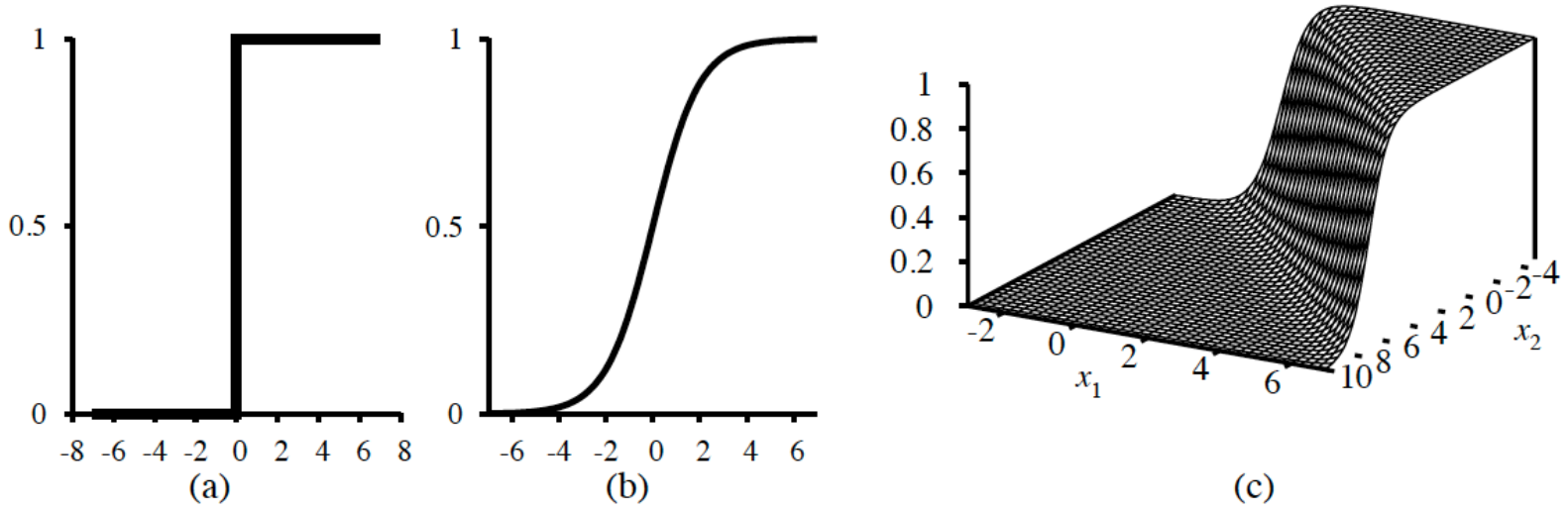


Figure 18.17 **FILES: .** (a) The hard threshold function $Threshold(z)$ with 0/1 output. Note that the function is nondifferentiable at $z = 0$. (b) The logistic function, $Logistic(z) = \frac{1}{1+e^{-z}}$, also known as the sigmoid function. (c) Plot of a logistic regression hypothesis $h_{\mathbf{w}}(\mathbf{x}) = Logistic(\mathbf{w} \cdot \mathbf{x})$ for the data shown in Figure 18.14(b).

An important function!

Logistic Regression

$$h_{\mathbf{w}}(\mathbf{x}) = Logistic(\mathbf{w}\cdot\ \mathbf{x}) = \frac{1}{1+e^{-\mathbf{w}\cdot\mathbf{x}}}$$

For a single sample $(\mathbf{x}, y)$ and L_2 loss function:

$$w_i \leftarrow w_i + \alpha\left(y - h_{\mathbf{w}}(\mathbf{x})\right)h_{\mathbf{w}}(\mathbf{x})\left(1 - h_{\mathbf{w}}(\mathbf{x})\right)x_i$$

derivative of logistic function

What you don't have to know for the exam

- Information gain formula
- Broadening applicability of decision trees
- Wrapper
- Small and large-scale learning
- Minimum description length (MDL)
- Learning theory: Sec. 18.5
- Formula for analytic solution of regression problem
- L1 vs. L2 regularization
- Logistic regression learning rule

Artificial Neural Networks

- Networks of relatively simple processing units, which are very abstract models of neurons; the network does the computation more than the units.

Neuron-like units

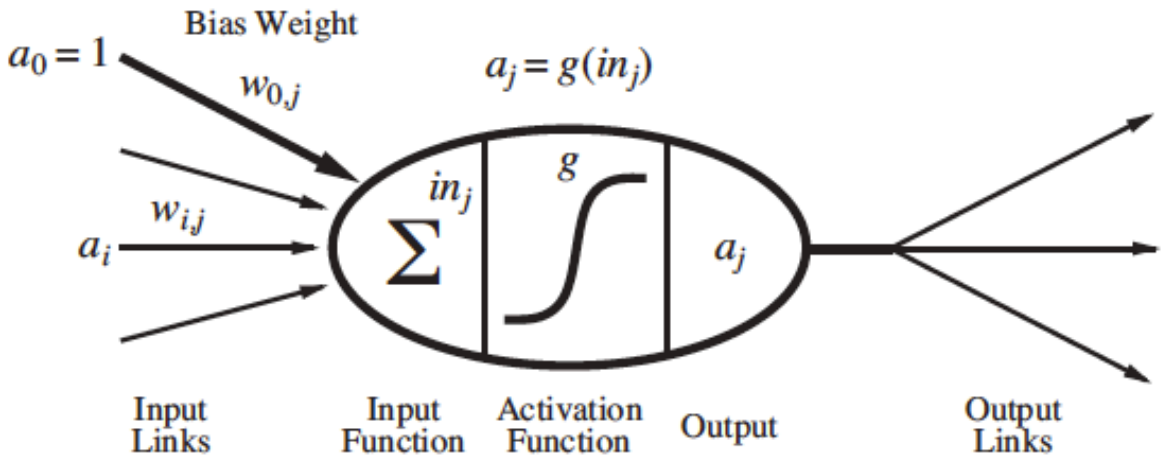
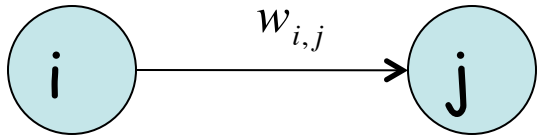
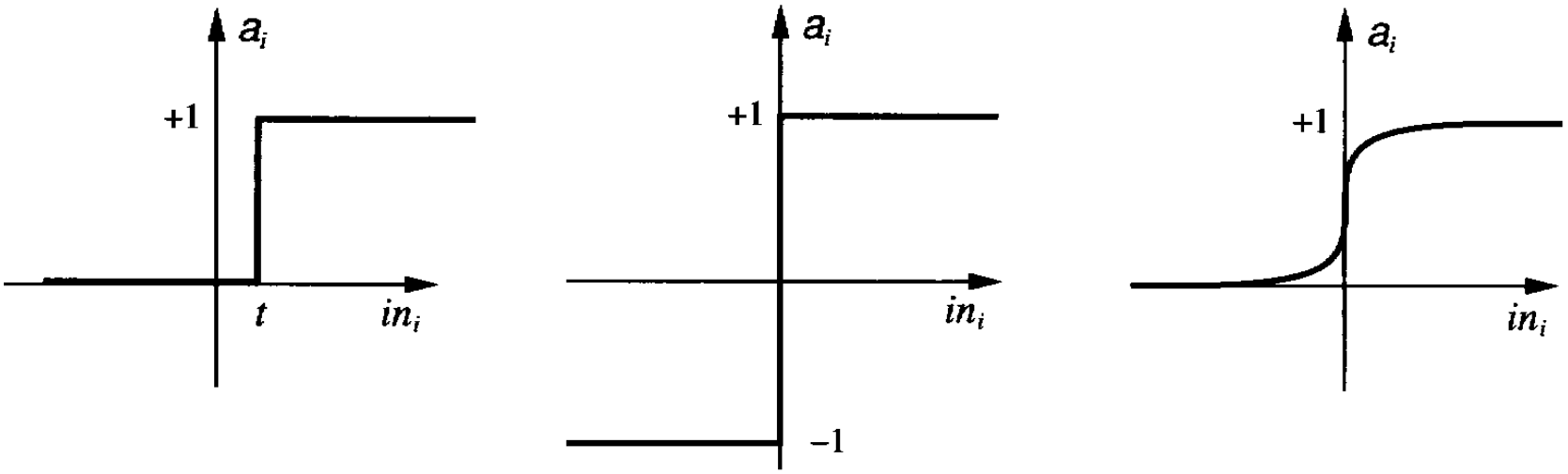


Figure 18.19 **FILES: figures/neuron-unit.eps (Wed Nov 4 11:23:13 2009).** A simple mathematical model for a neuron. The unit's output activation is $a_j = g(\sum_{i=0}^{n} w_{i,j} a_i)$, where a_i is the output activation of unit i and $w_{i,j}$ is the weight on the link from unit i to this unit.



Typical activation functions



(a) Step function **(b) Sign function** **(c) Sigmoid function**

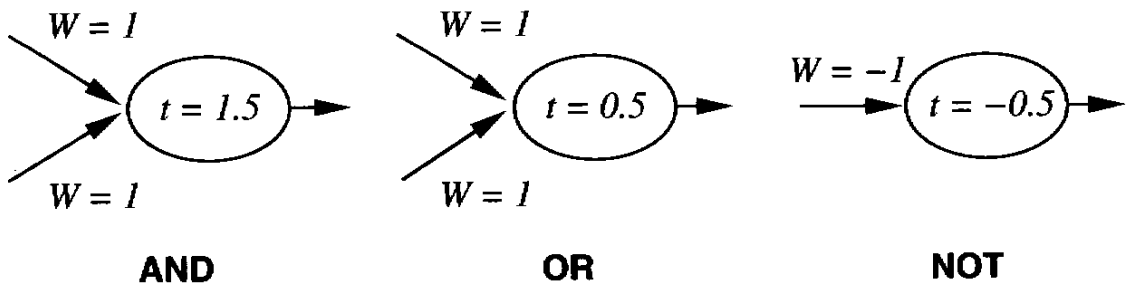
Figure 19.5 Three different activation functions for units.

$$\text{step}_t(x) = \begin{cases} 1, & \text{if } x \geq t \\ 0, & \text{if } x < t \end{cases} \quad \text{sign}(x) = \begin{cases} +1, & \text{if } x \geq 0 \\ -1, & \text{if } x < 0 \end{cases} \quad \text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

NNs and logic

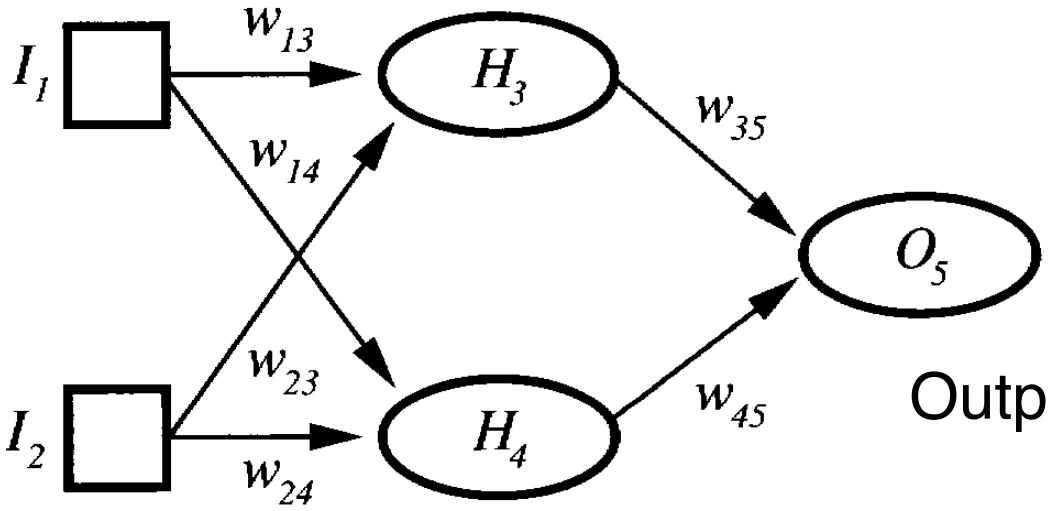
McCulloch and Pitts, 1943: showed that whatever you can do with logic networks, you can do with networks of abstract neuron-like units.

Units with step function activation functions



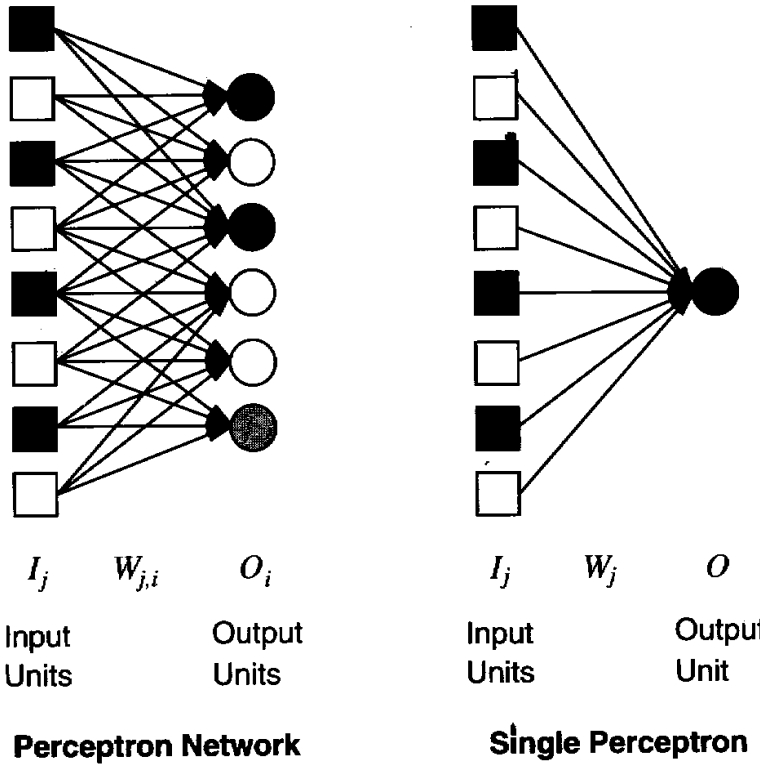
Network structures

- Feed-forward vs. recurrent networks
- Multi-layer feed-forward networks



Perceptrons

- Name given in 1950s to layered feed-forward networks.

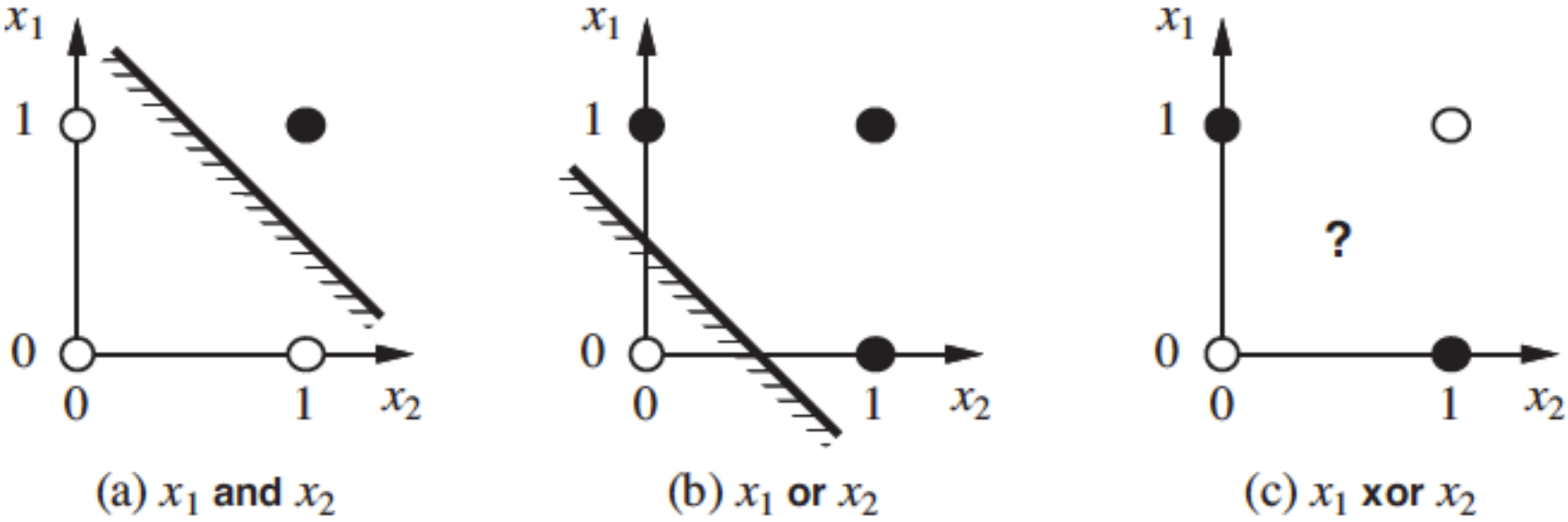


$$O = Step_0(\sum_j W_j I_j)$$

$$= Step_0(W \cdot I)$$

What can perceptrons represent

- Only **linearly separable** functions



Back-propagation learning

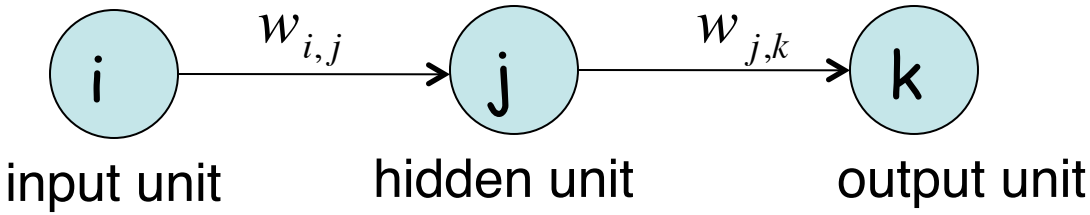
To update weights from hidden units to output unit

$$Err_k = k^{th} \text{ component of } \mathbf{y} - \mathbf{h_w}$$

$$w_{j,k} \leftarrow w_{j,k} + \alpha \times a_j \times Err_k \times g'(in_k)$$

letting $\Delta_k = Err_k \, g'(in_k)$ this becomes

$$w_{j,k} \leftarrow w_{j,k} + \alpha \times a_j \times \Delta_k$$

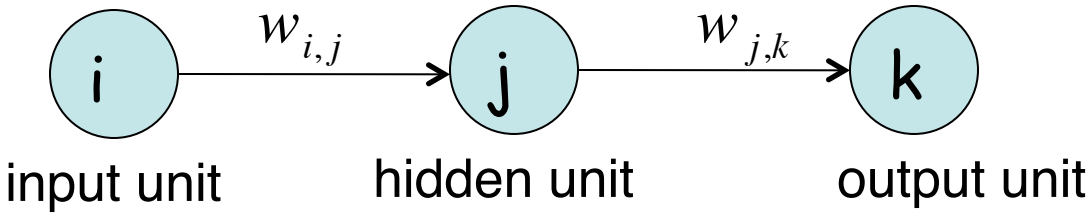


Back-prop contd.

To update weights from input units to hidden units

$$\Delta_j = g'(in_j)\sum_k w_{j,k}\Delta_k$$

$$w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta_j$$

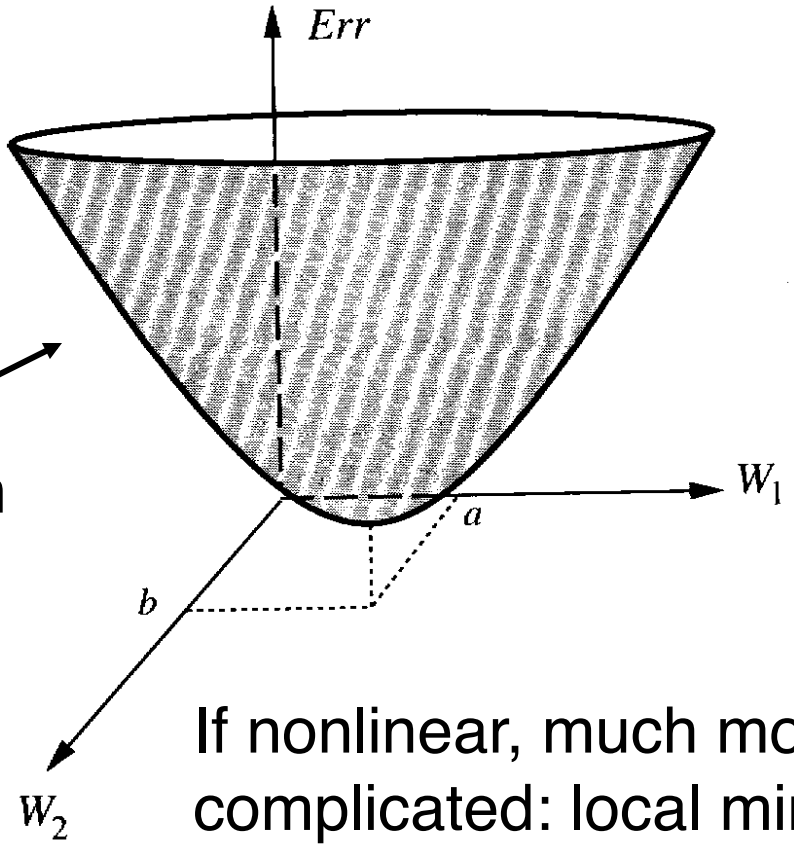


Back-prop as gradient descent

$$Err = \sum_k (y_k - a_k)^2$$

sum squared error

An error surface for net with linear activation functions



If nonlinear, much more complicated: local minima

Comments on network learning

- **Expressiveness**: given enough hidden units, can represent any function (almost).
- **Computational efficiency**: generally slow to train, but fast to use once trained.
- **Generalization**: good success in a number of real-world problems.
- **Sensitivity to noise**: very tolerant to noise in data

Comments contd.

- **Transparency**: not good!
- **Prior knowledge**: not particularly easy to insert prior knowledge, although possible.

Nonparametric Methods

- Parametric model: a learning model that has a set of parameters of fixed size
  - e.g., linear models, neural networks (of fixed size)
- Nonparametric model: a learning model whose set of parameters is not bounded
  - Parameter set grows with the number of training examples
  - e.g., just save the examples in a lookup table

Nearest Neighbor Models

- K-nearest neighbors algorithm:
  - Save all the training examples
  - For classification: find k nearest neighbors of the input and take a vote (make k odd)
  - For regression: take mean or median of the k nearest neighbors, or do a local regression on them
- How do you measure distance?
- How do you efficiently find the k nearest neighbors?

Distance Measures

Minkowski distance

$$L^p(\mathbf{x}_j, \mathbf{x}_q) = \left(\sum_{\mathrm{i}} \left| x_{j,i} - x_{q,i} \right|^p \right)^{1/p}$$

$p = 1$ Manhattan distance

$p = 2$ Euclidean distance

Hamming distance for Boolean attribute values

k-nearest neighbor for k=1 and k=5

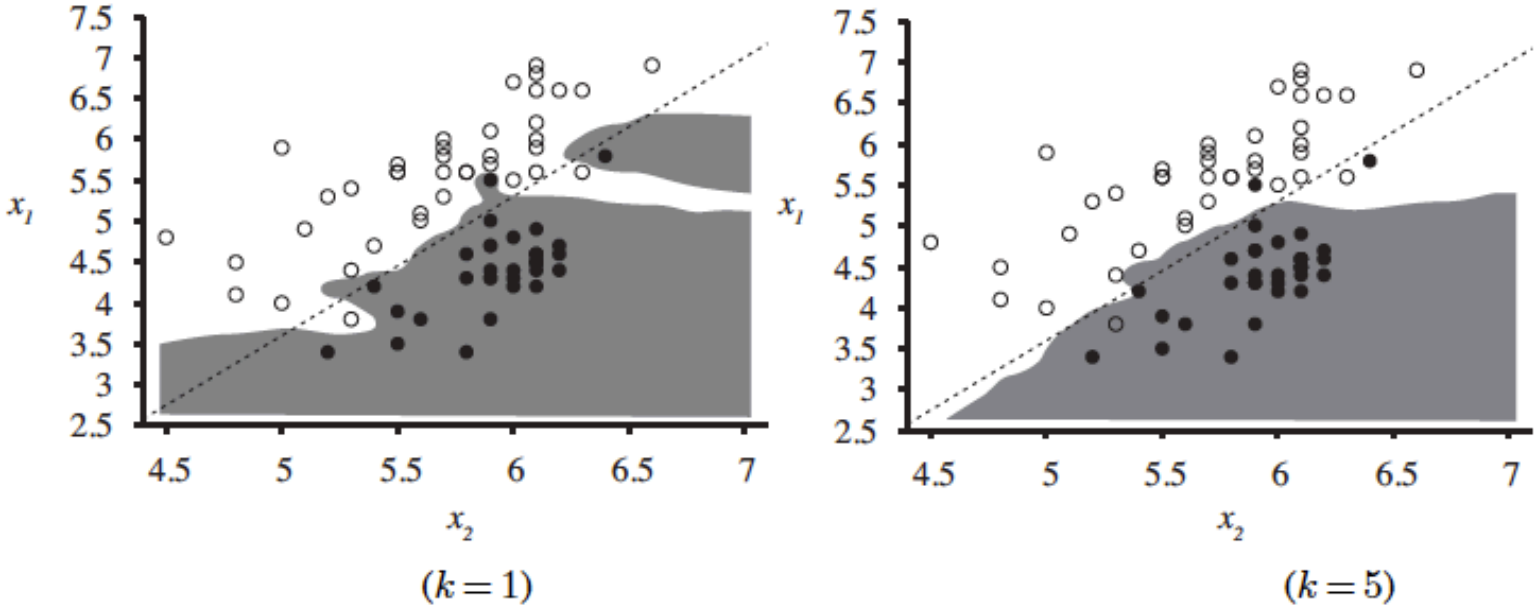


Figure 18.26 FILES: figures/earthquake-nn1.eps (Tue Nov 3 16:22:38 2009) figures/earthquake-nn5.eps (Tue Nov 3 16:22:38 2009). (a) A k-nearest-neighbor model showing the extent of the explosion class for the data in Figure 18.14, with $k=1$. Overfitting is apparent. (b) With $k=5$, the overfitting problem goes away for this data set.

Curse of Dimensionality

In high dimensions, the nearest points tend to be far away.

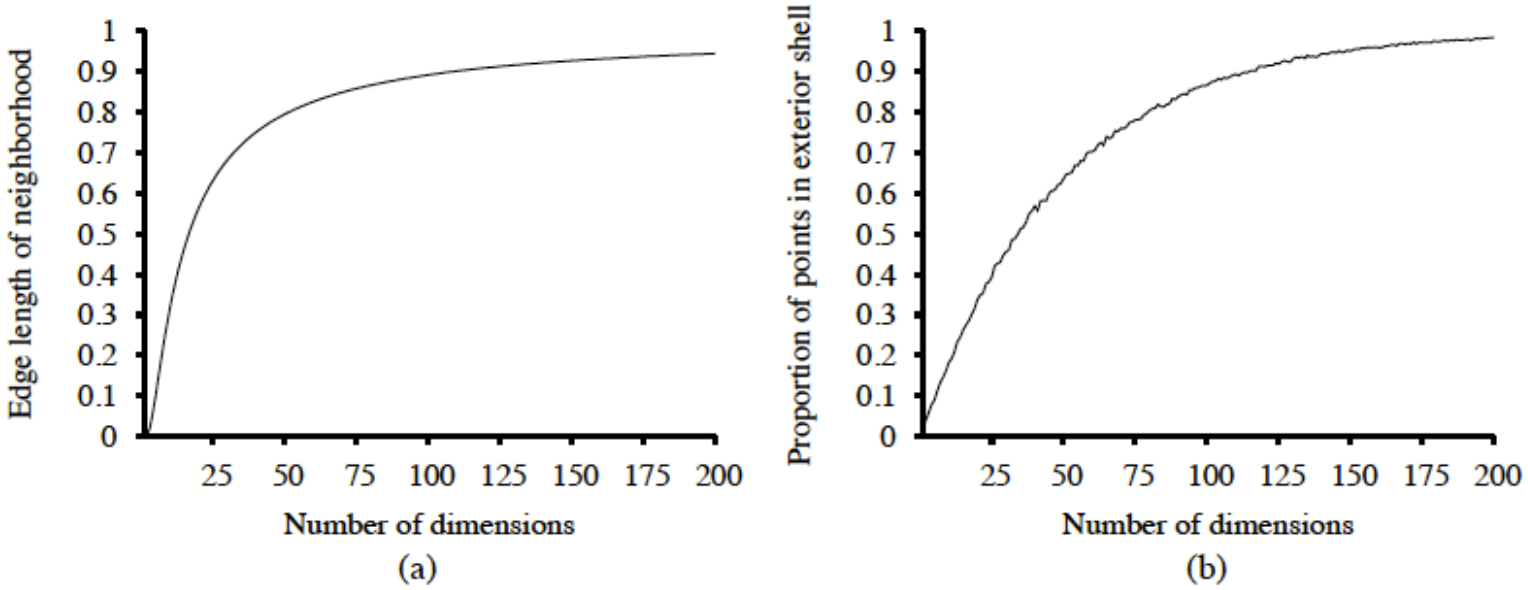


Figure 18.27 **FILES: .** The curse of dimensionality: (a) The length of the average neighborhood for 10-nearest-neighbors in a unit hypercube with 1,000,000 points, as a function of the number of dimensions. (b) The proportion of points that fall within a thin shell consisting of the outer 1% of the hypercube, as a function of the number of dimensions. Sampled from 10,000 randomly distributed points.

Nonparametric Regression

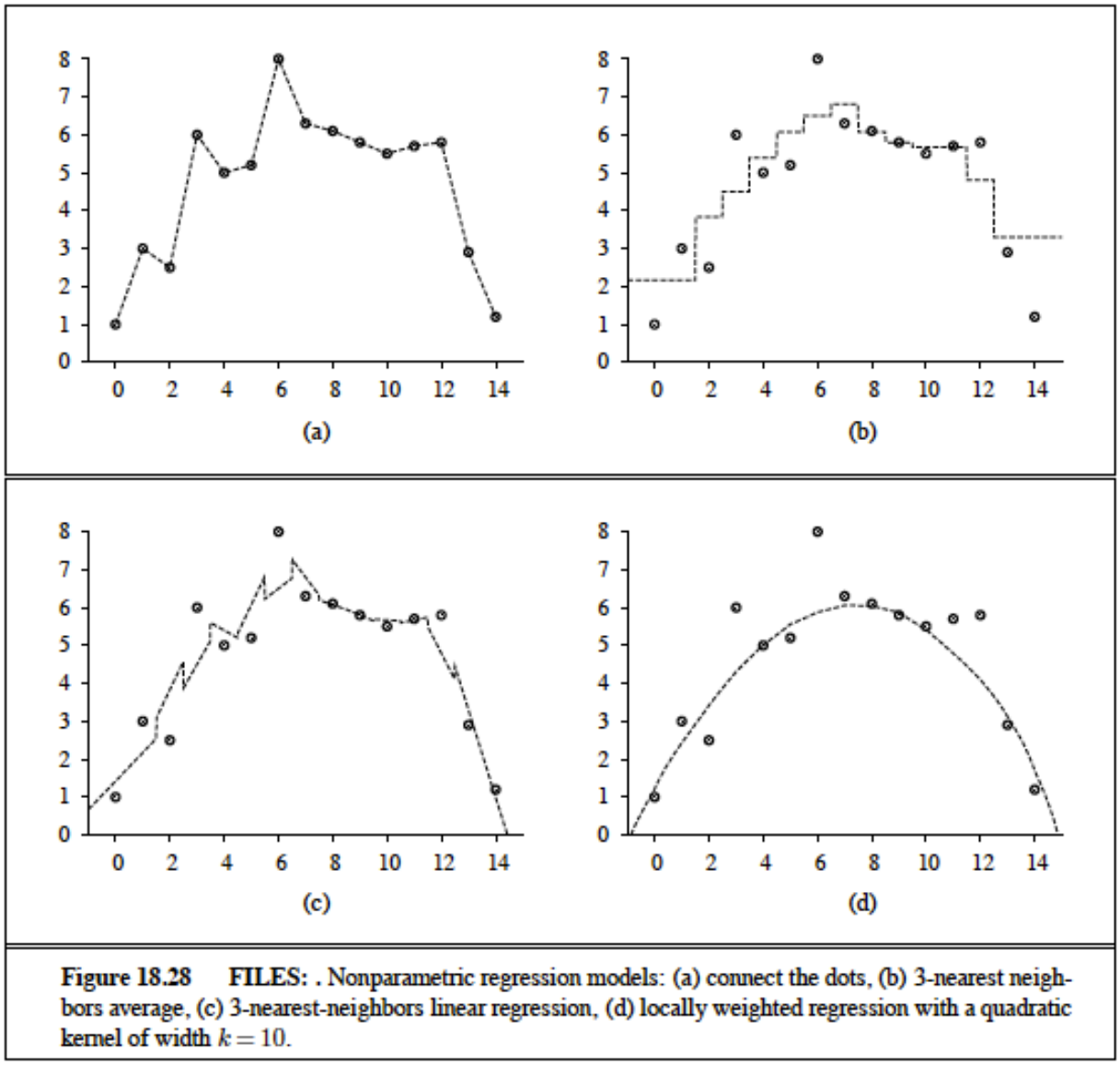


Figure 18.28 **FILES:** . Nonparametric regression models: (a) connect the dots, (b) 3-nearest neighbors average, (c) 3-nearest-neighbors linear regression, (d) locally weighted regression with a quadratic kernel of width $k = 10$.

What you don't have to know for the exam

- Details of backpropagation learning method (but know the general idea…)
- Optimal brain damage
- Locality-sensitive hashing
- Locally weighted regression
- Secs. 18.9—18.11

Chapter 20: Learning Probabilistic Models

- Full Bayesian Learning
- MAP approximation
- ML approximation
- ML parameter learning in Bayes nets
  - Naïve Bayes Model
- Bayesian parameter learning
  - Beta family of distributions
  - Conjugate families
- Latent variables

Full Bayesian Learning

View learning as Bayesian updating of a probability distribution over the hypothesis space

H is the hypothesis variable, values $h_1, h_2, \ldots$, prior $\mathbf{P}(H)$

jth observation d_j gives the outcome of random variable D_j
training data $\mathbf{d} = d_1, \ldots, d_N$

Given the data so far, each hypothesis has a posterior probability:

$$P(h_i|\mathbf{d}) = \alpha P(\mathbf{d}|h_i)P(h_i)$$

where $P(\mathbf{d}|h_i)$ is called the likelihood

Predictions use a likelihood-weighted average over the hypotheses:

$$\mathbf{P}(X|\mathbf{d}) = \Sigma_i \, \mathbf{P}(X|\mathbf{d}, h_i)P(h_i|\mathbf{d}) = \Sigma_i \, \mathbf{P}(X|h_i)P(h_i|\mathbf{d})$$

No need to pick one best-guess hypothesis!

MAP approximation

Summing over the hypothesis space is often intractable
(e.g., 18,446,744,073,709,551,616 Boolean functions of 6 attributes)

Maximum a posteriori (MAP) learning: choose h_{MAP} maximizing $P(h_i|\mathbf{d})$

I.e., maximize $P(\mathbf{d}|h_i)P(h_i)$ or $\log P(\mathbf{d}|h_i) + \log P(h_i)$

For deterministic hypotheses, $P(\mathbf{d}|h_i)$ is 1 if consistent, 0 otherwise

$\Rightarrow$ MAP = simplest consistent hypothesis (cf. science)

ML approximation

For large data sets, prior becomes irrelevant

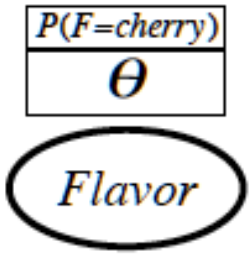
Maximum likelihood (ML) learning: choose h_{ML} maximizing $P(\mathbf{d}|h_i)$

I.e., simply get the best fit to the data; identical to MAP for uniform prior (which is reasonable if all hypotheses are of the same complexity)

ML is the "standard" (non-Bayesian) statistical learning method

ML parameter learning in Bayes nets

Bag from a new manufacturer; fraction θ of cherry candies?
Any θ is possible: continuum of hypotheses h_θ
θ is a parameter for this simple (binomial) family of models



Suppose we unwrap N candies, c cherries and $\ell = N - c$ limes
These are i.i.d. (independent, identically distributed) observations, so

$$P(\mathbf{d}|h_\theta) = \prod_{j=1}^{N} P(d_j|h_\theta) = \theta^c \cdot (1-\theta)^\ell$$

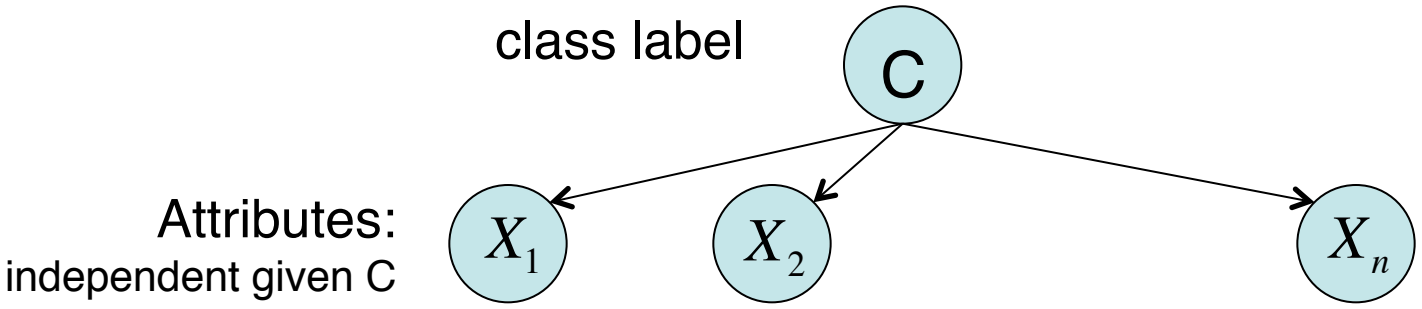
Maximize this w.r.t. θ—which is easier for the log-likelihood:

$$L(\mathbf{d}|h_\theta) = \log P(\mathbf{d}|h_\theta) = \sum_{j=1}^{N} \log P(d_j|h_\theta) = c\log\theta + \ell\log(1-\theta)$$

$$\frac{dL(\mathbf{d}|h_\theta)}{d\theta} = \frac{c}{\theta} - \frac{\ell}{1-\theta} = 0 \quad \Rightarrow \quad \theta = \frac{c}{c+\ell} = \frac{c}{N}$$

Seems sensible, but causes problems with 0 counts!

Naïve Bayes Model



$$\mathbf{P}(\mathrm{C} \mid \mathrm{x}_1, x_2, \ldots, x_n) = \alpha\, \mathbf{P}(C) \prod_i \mathbf{P}(x_i \mid C)$$

Naïve Bayes Classifier:

$$\mathrm{C}_{\mathrm{NB}} = \mathrm{argmax}_{\mathrm{C} \in \mathrm{lables}} \mathbf{P}(\mathrm{C} \mid \mathrm{x}_1, x_2, \ldots, x_n) = \arg\max \alpha\, \mathbf{P}(C) \prod_i \mathbf{P}(x_i \mid C)$$

Naïve Bayes contd.

$$C_{NB} = \text{argmax}_{C \in \text{lables}} \mathbf{P}(C \mid x_1, x_2, \ldots, x_n) = \arg\max \alpha \mathbf{P}(C) \prod_i \mathbf{P}(x_i \mid C)$$

Or, taking logs and dropping α :

$$C_{NB} = \text{argmax}_{C \in \text{lables}} \log \mathbf{P}(C \mid x_1, x_2, \ldots, x_n) = \log \mathbf{P}(C) \prod_i \mathbf{P}(x_i \mid C)$$

$$= \log P(c) + \sum_i \log \mathbf{P}(x_i \mid C)$$

→ a linear classifier

Full Bayesian parameter learning

- ML learning is simple but has some problems:
  - e.g., after seeing one sample, the ML estimate is %100 that sample
- Bayesian approach starts with a **hypothesis prior,** which is revised using Bayes rule as more data comes in.
- E.g., consider one unknown parameter θ

  We start with a prob. distribution over values of θ : e.g., the prior probability that a bag has a fraction θ of cherries.

Beta family of distributions

$$\text{beta}[a,b](\theta) = \alpha\,\theta^{a-1}(1-\theta)^{b-1}$$

a and b are called **hyperparameters**

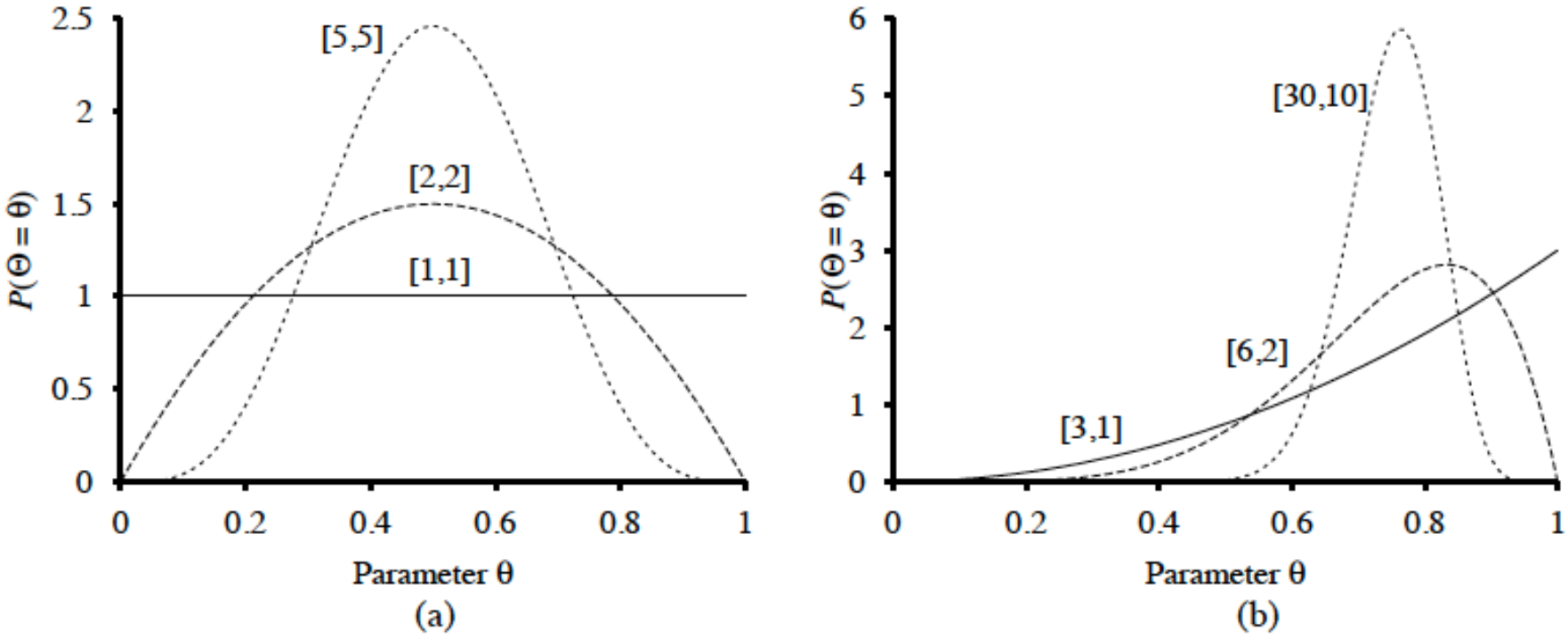


Figure 20.5 **FILES: .** Examples of the beta$[a, b]$ distribution for different values of $[a, b]$.

Conjugate families of distributions

- E.g., the Beta family

  Closed under Bayesian updates

$$P(\theta \mid D_1 = cherry) = \alpha P(D_1 = cherry \mid \theta) P(\theta)$$
$$= \alpha' \theta \cdot \text{beta}[a,b](\theta) = \alpha' \theta \cdot \theta^{a-1}(1-\theta)^{b-1}$$
$$= \alpha' \theta^{a}(1-\theta)^{b-1} = beta[a+1,b](\theta)$$

Latent variables

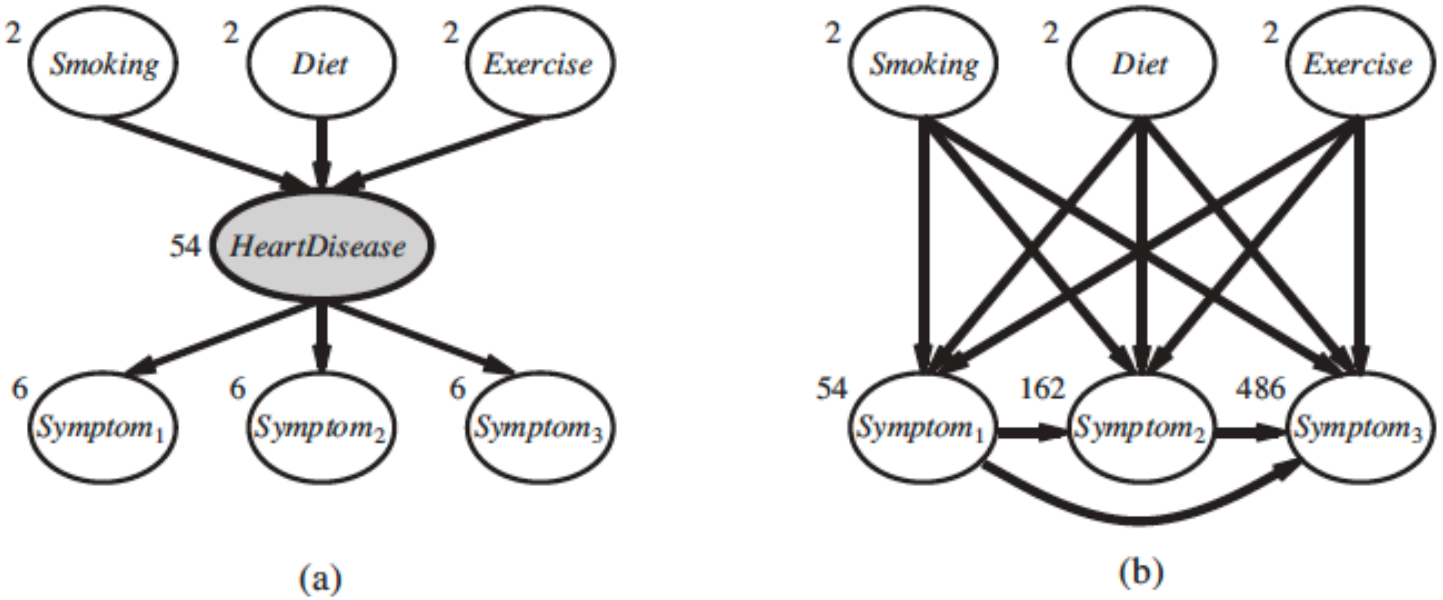


Figure 20.10 FILES: figures/313-heart-disease.eps (Tue Nov 3 16:22:09 2009). (a) A simple diagnostic network for heart disease, which is assumed to be a hidden variable. Each variable has three possible values and is labeled with the number of independent parameters in its conditional distribution; the total number is 78. (b) The equivalent network with *HeartDisease* removed. Note that the symptom variables are no longer conditionally independent given their parents. This network requires 708 parameters.

What you don't have to know for the exam

- MDL method
- ML learning for continuous models (Sec. 20.2.3)
- Learning Bayes net structures (Sec. 20.2.5)
- Density estimation (Sec. 20.2.6)
- Learning with hidden variables (Sec. 20.3): but know what a latent variable is and why useful

Chapter 21: Reinforcement Learning

Chapter 21: Reinforcement Learning

- Some kinds of RL agents
  - Utility-based agent: learns utility function on states and uses it to select actions
    - Needs an environment model to decide on actions
  - Q-learning agent: leans and action-utility functions, or Q-function, giving expected utility for taking each action in each state.
    - Does not need an environment model.
  - Reflex agent: learn a policy without first learning a state-utility function or a Q-function

Passive versus active learning

- A **passive learner** simply watches the world going by and tries to learn the utility of being in various states.
- An **active learner** must also act using the learned information, and can use its problem generator to suggest explorations of unknown portions of the environment.

Passive learning

Given (but agent doesn't know this):

- A Markov model of the environment.
- States, with probabilistic actions.
- Terminal states have rewards/utilities.

Problem:

- Learn expected utility of each state.

Note: if agent knows how the environment and its actions work, can solve the relevant Bellman equation (which would be linear).

Learning utility functions

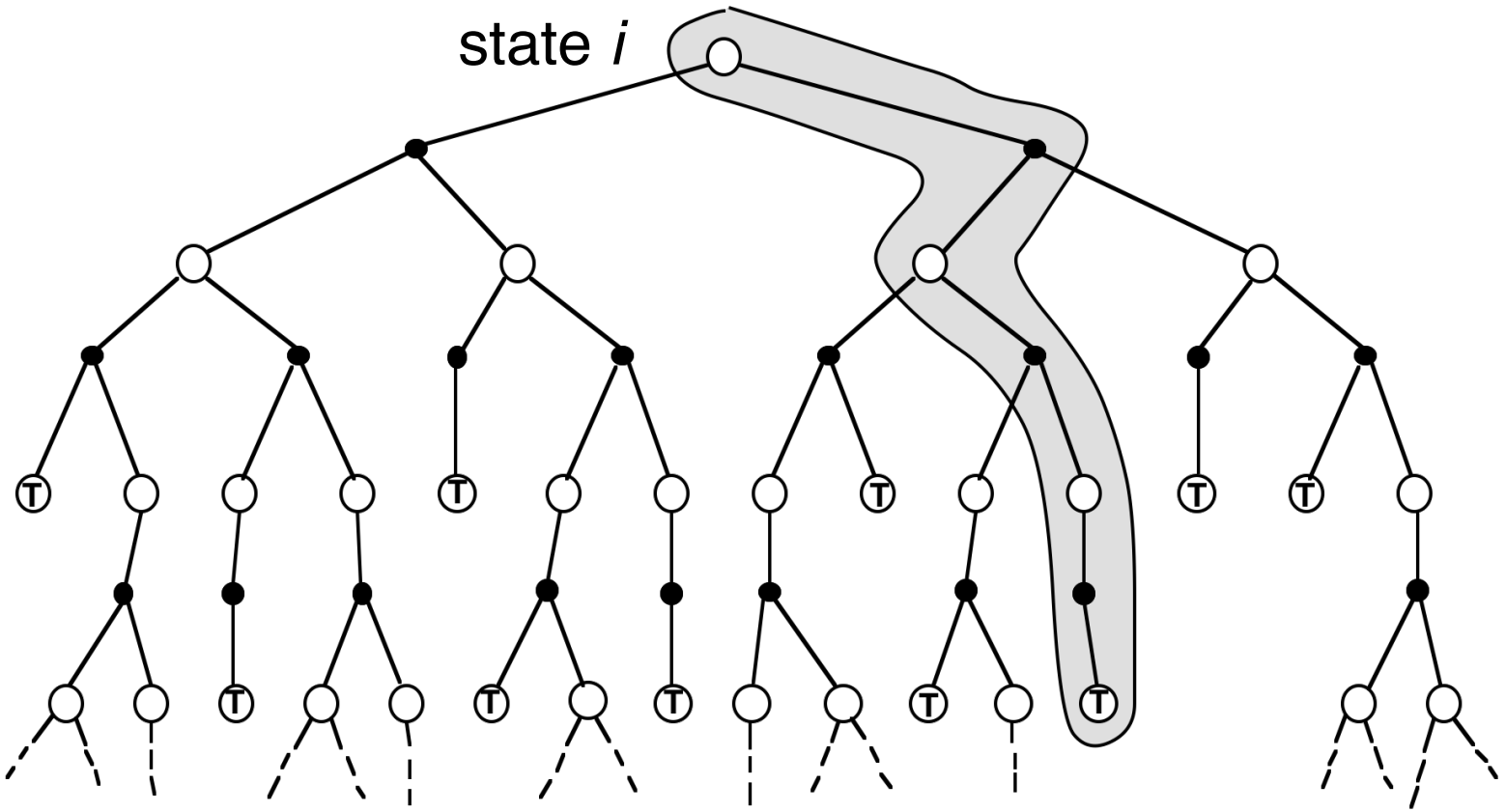
- A **training sequence** (or episode) is an instance of world transitions from an initial state to a terminal state.
- The **additive utility assumption**: utility of a sequence is the sum of the rewards over the states of the sequence.
- Under this assumption, the utility of a state is the expected **reward-to-go** of that state.

Direct Utility Estimation

- For each training sequence, compute the reward-to-go for each state in the sequence and update the utilities.
- This is just learning the utility function from examples.
- Generates utility estimates that minimize the mean square error (LMS-update).

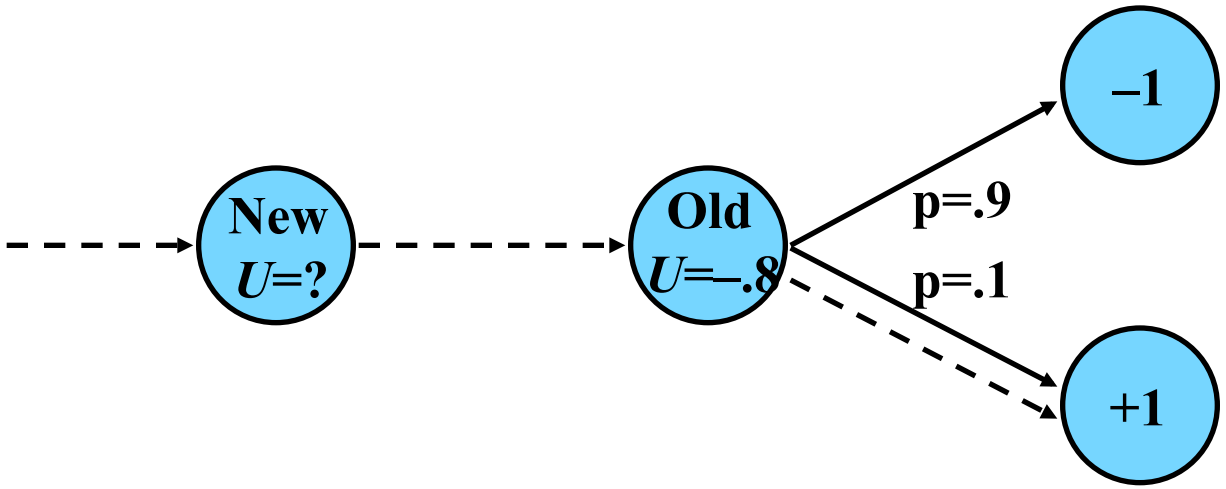
Direct Utility Estimation

$$U(i) \leftarrow (1-\alpha)U(i) + \alpha\, REWARD(\text{training sequence})$$



Problems with direct utility estimation

Converges slowly because it ignores the relationship between neighboring states:



Key Observation: Temporal Consistency

- Utilities of states are not independent
- The utility of each state equals its own reward plus the expected utility of its successor states:

$$U^{\pi}(s) = R(s) + \gamma \sum_{s'} P(s' \mid s,\ \pi(s)) U^{\pi}(s')$$

The key fact:

$$\begin{aligned} U_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} \cdots \\ &= r_{t+1} + \gamma\left(r_{t+2} + \gamma r_{t+3} + \gamma^2 r_{t+4} \cdots \right) \\ &= r_{t+1} + \gamma U_{t+1} \end{aligned}$$

Adaptive dynamic programming

- Learn a model: transition probabilities, reward function
- Do policy evaluation
  - Solve the Bellman equation either directly or iteratively (value iteration without the max)
- Learn model while doing iterative policy evaluation:
  - Update the model of the environment after each step. Since the model changes only slightly after each step, policy evaluation will converge quickly.

Temporal difference (TD) learning

Approximate the constraint equations without solving them for all states.

Modify $U(i)$ whenever we see a transition from i to j using the following rule:

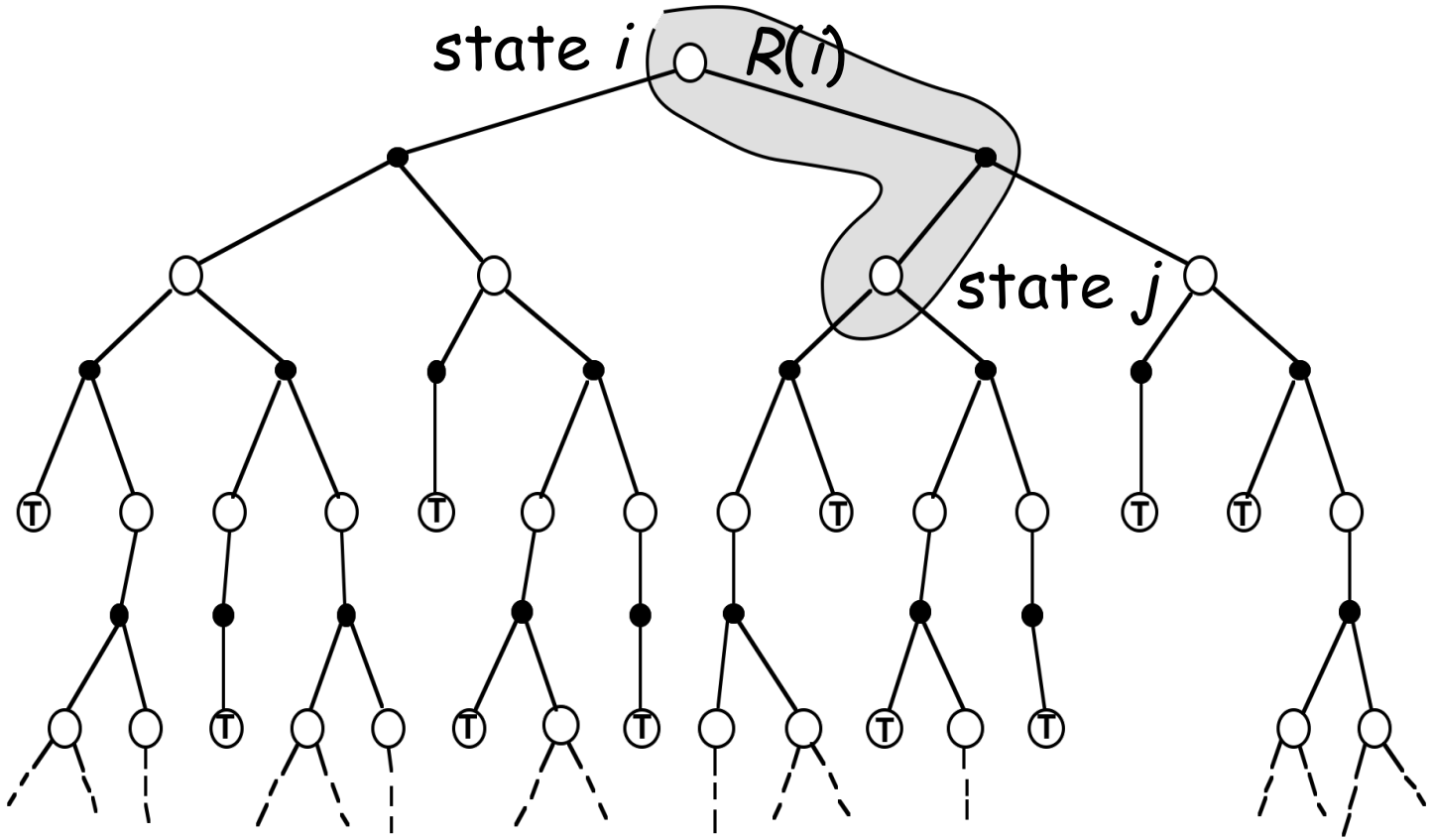
$$U(i) \leftarrow U(i) + \alpha\left[R(i) + U(j) - U(i)\right]$$

The modification moves $U(i)$ closer to satisfying the original equation.

Q. Why does it work?

TD learning contd.

$$U(i) \leftarrow (1-\alpha)U(i) + \alpha\left[R(i) + U(j)\right]$$



Passive RL: review

- Agent has fixed policy and learns utilities
  - Barto: “not really RL: it is prediction”
- Direct utility estimation
  - Collect samples of quantity to be estimated
  - Average them
  - Or use an incremental method….
  - Does not take advantage of relationship between utilities of different states
- Adaptive Dynamic Programming (ADP)
  - Learn a model
  - Do DP on it
  - Can interleave these (modified policy iteration)
- Temporal Difference (TD) Learning
  - Use an error to make estimates adhere to constraint
  - Does not need a model

A Bit of Terminology

- Utilities (U) = Values (V)
- Return: discounted sum of rewards
  - Return from a state: the discounted sum of rewards accumulated after visiting that state
  - Same as “reward-to-go”
  - Utility (or value) of a state is the expected return from that state

More on TD Learning

- TD methods do not require a model of the environment, only experience
- You can learn before knowing the final outcome
  - Less memory
  - Less peak computation
  - You can learn without the final outcome from incomplete sequences

You are the Predictor

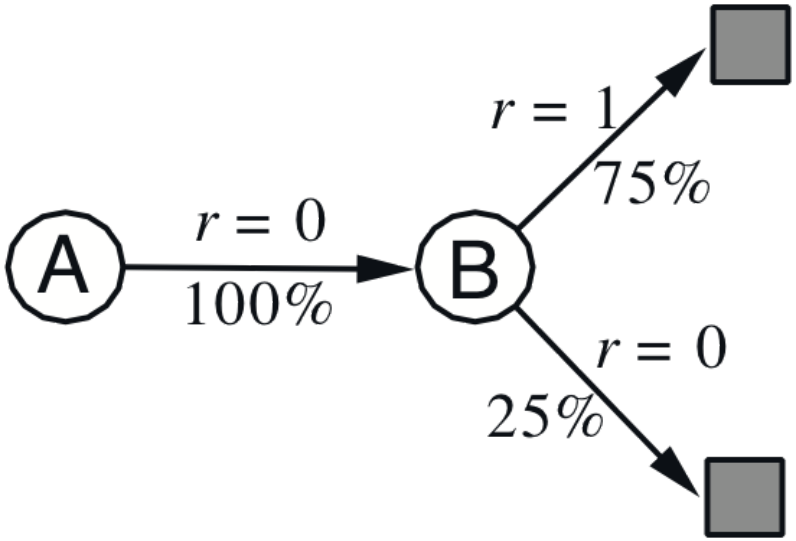
Suppose you observe the following 8 episodes:

A, 0, B, 0
B, 1
B, 1
B, 1
B, 1
B, 1
B, 1
B, 0

$V(\mathrm{A})?$

$V(\mathrm{B})?$

You are the Predictor



$V(\mathrm{A})?$

You are the Predictor

- The prediction that best matches the training data is V(A)=0
  - This minimizes the mean-square-error on the training set
  - This is what direct utility estimation gets
- If we consider the sequentiality of the problem, then we would set V(A)=.75
  - This is correct for the maximum likelihood estimate of a Markov model generating the data
  - i.e, if we do a best fit Markov model, and assume it is exactly correct, and then compute what it predicts (how?)
  - This is called the certainty-equivalence estimate
  - This is what TD gets

117

TD learning of Action-Values (for a given policy)

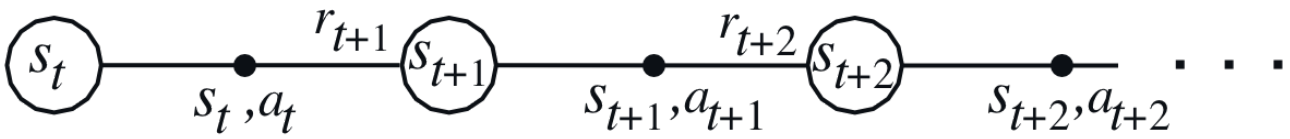
$Q^{\pi}(s,a) =$ Utility of doing action a in state s
i.e.: Total amount of reward expected over the future if you do action *a* in state s and thereafter follow policy π

The utility of a state is the utility of doing the best action from that state:

$$U^{\pi}(s) = \max_{a} Q^{\pi}(s,a)$$

TD Learning of Action-Values (for a given policy)

Estimate Q^{π} for the current behavior policy π.



After every transition from a nonterminal state s_t, do this:

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

If s_{t+1} is terminal, then $Q(s_{t+1}, a_{t+1}) = 0$.

Active RL

- Passive agent follows a fixed policy, estimates expected utilities
- Active agent needs to decide on what actions to perform to maximize expected utility

Passive agent: faces a **prediction** problem
Active agent: faces a **control** problem

Key Fact

- A greedy agent is one that always takes the action that maximizes its current utility estimates

- If its utility estimates are correct, i.e., it has learned the true utility function (or optimal value function), then a greedy agent acts optimally.

Exploration/Exploitation Dilemma

- Exploitation: act according to your current estimates (exploit current “knowledge”).
- Exploration: do something else!

- You can’t do both at the same time.
- How do you handle the tradeoff?

What's the best exploration policy?

Assume you've learned a utility function,
How do you select actions?

Current estimate

Greedy Action Selection:

Always select the action that looks best:

$$\pi(s) = \arg\max_a Q(s,a)$$

ε-Greedy Action Selection:

Be greedy most of the time

Occasionally take a random action

The simplest possible thing!

Other Methods:

Boltzmann distribution, keep track of confidence intervals, etc.

ε-Greedy Action Selection

- Greedy action selection:

$$a_t = a_t^* = \arg\max_a Q_t(a)$$

- ε-Greedy:

$$a_t = \begin{cases} a_t^* & \text{with probability } 1 - \varepsilon \\ \text{random action with probability} & \varepsilon \end{cases}$$

. . . the simplest way to try to balance exploration and exploitation

More on Exploration

- GLIE schemes: “Greedy in the Limit of Infinite Exploration”
  - Simplest maybe: ε-Greedy with decreasing ε
  - Optimistic initial estimates, fading out with increasing visitations
- “Exploration Bonuses”

Sarsa

Turn passive learning of action values into an active method by always updating the policy to be greedy with respect to the current estimate:

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
    Initialize s
    Choose a from s using policy derived from Q (e.g., ϵ-greedy)
    Repeat (for each step of episode):
        Take action a, observe r, s'
        Choose a' from s' using policy derived from Q (e.g., ϵ-greedy)
        $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma Q(s', a') - Q(s, a) \right]$
        $s \leftarrow s'; a \leftarrow a';$
    until s is terminal

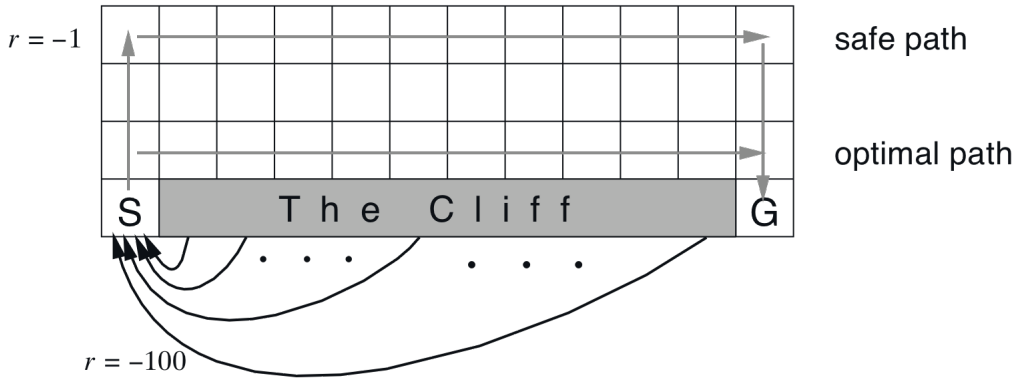
Q-Learning

One - step Q - learning :

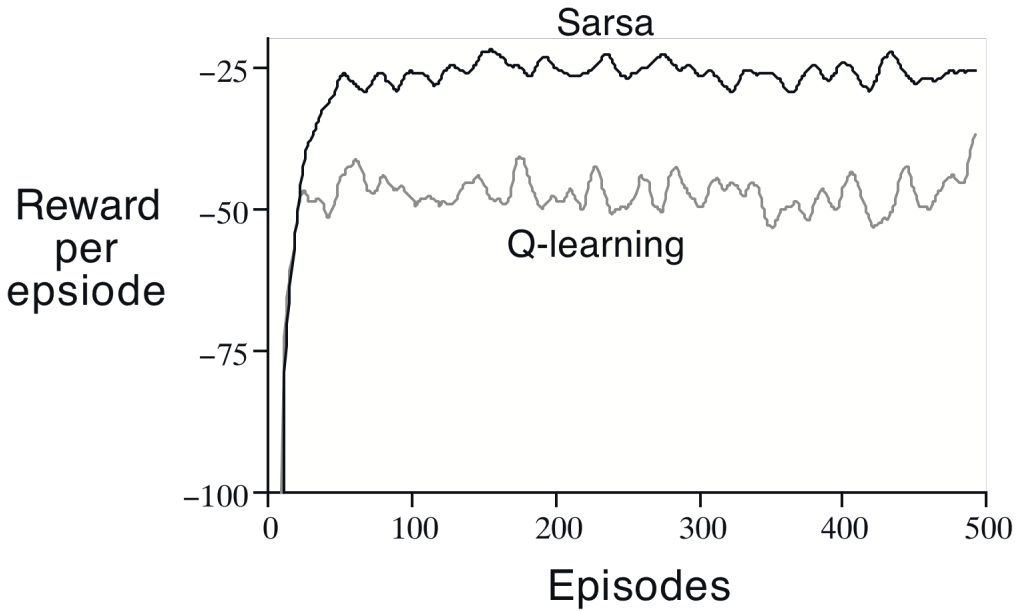
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\right]$$

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
    Initialize s
    Repeat (for each step of episode):
        Choose a from s using policy derived from Q (e.g., ϵ-greedy)
        Take action a, observe r, s'
        $Q(s, a) \leftarrow Q(s, a) + \alpha\left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right]$
        $s \leftarrow s'$;
    until s is terminal

Cliffwalking



ε-greedy, ε = 0.1



On-Policy vs. Off-Policy

- Behavior policy: the policy the agent is using.
- Estimation policy: the policy the agent is evaluating

SARSA

- On-Policy methods:
  - Estimation policy = Behavior policy
- Off-Policy methods:
  - Estimation policy ≠ Behavior policy

Q-learning

Generalization in RL

- So far only considered lookup table representations of utility functions.
- What if the state set is huge? e.g. Backgammon
- Use function approximation methods

Features or Basis Functions

- E.g., linear function approximation: represent U or Q as a linear combination of features (or basis functions $f_1, \ldots f_n$:

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

$n <<$ number of states

Gradient ascent TD learning

For state-value functions:

$$\theta_i \leftarrow \theta_i + \alpha\left[R(s) + \gamma\,\hat{U}_\theta(s') - \hat{U}_\theta(s)\right]\frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$

For action-value functions:

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

For Linear Function Approximation

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

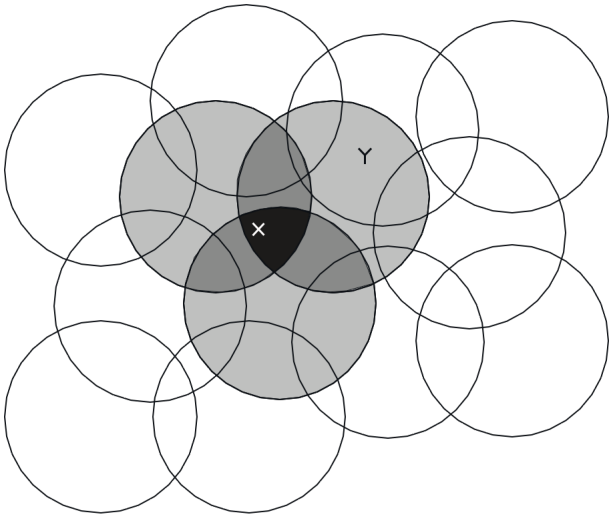
$$\frac{\partial \hat{U}_\theta(s)}{\partial \theta_i} = ?$$

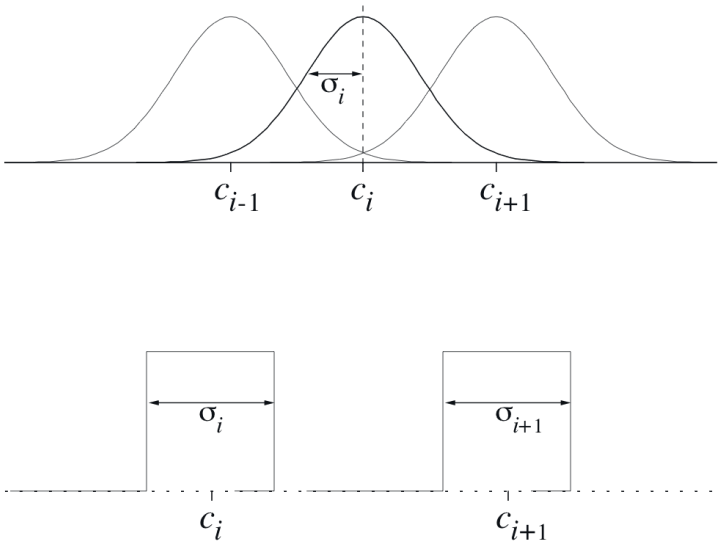
For Linear Function Approximation

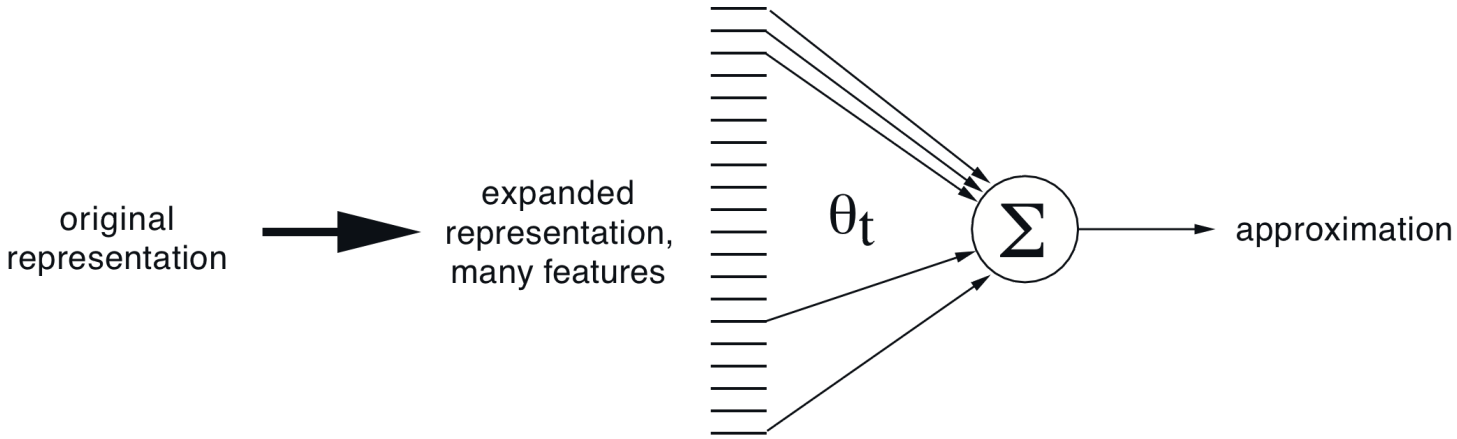
$$\hat{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \cdots + \theta_n f_n(s)$$

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

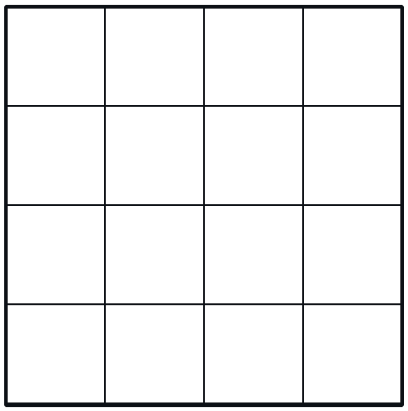
“Coarse Coding”



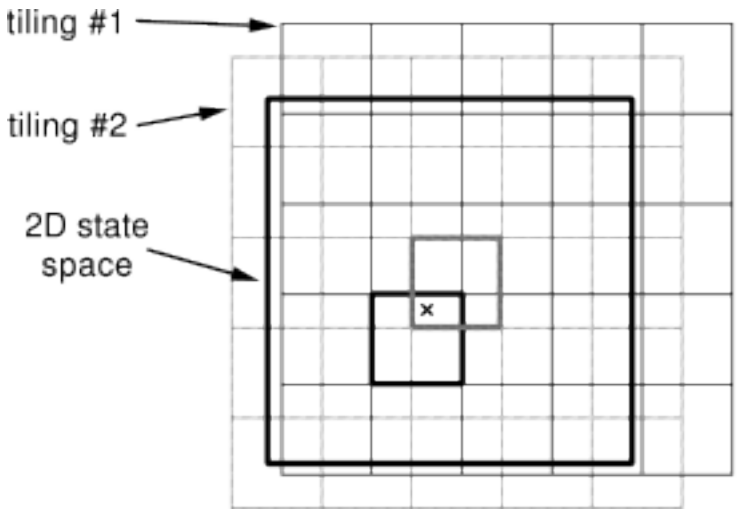




Tile Coding



- 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 freatures present



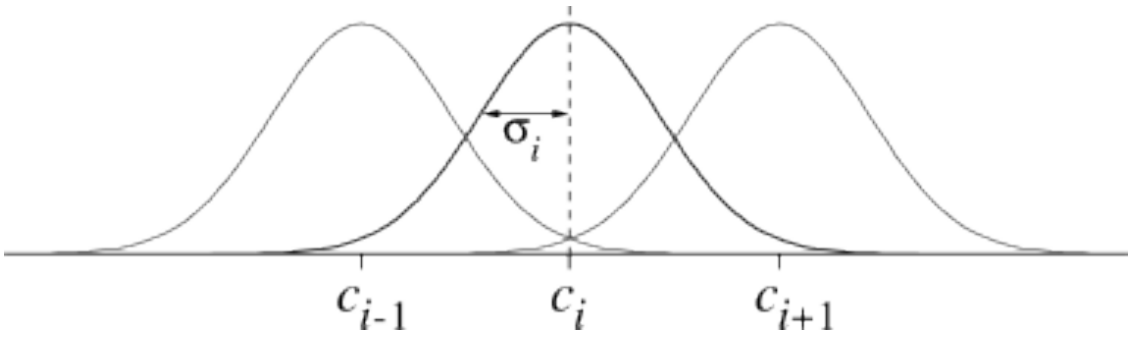
Shape of tiles ⇒ Generalization

#Tilings ⇒ Resolution of final approximation

Radial Basis Functions (RBFs)

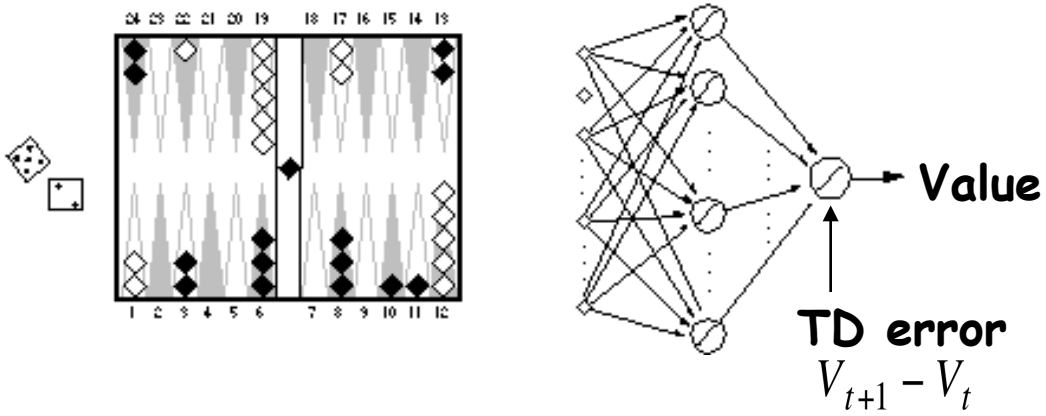
e.g., Gaussians

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



Nonlinear Function Approx.

TD-Gammon **Tesauro, 1992–1995**



Action selection by 2–3 ply search

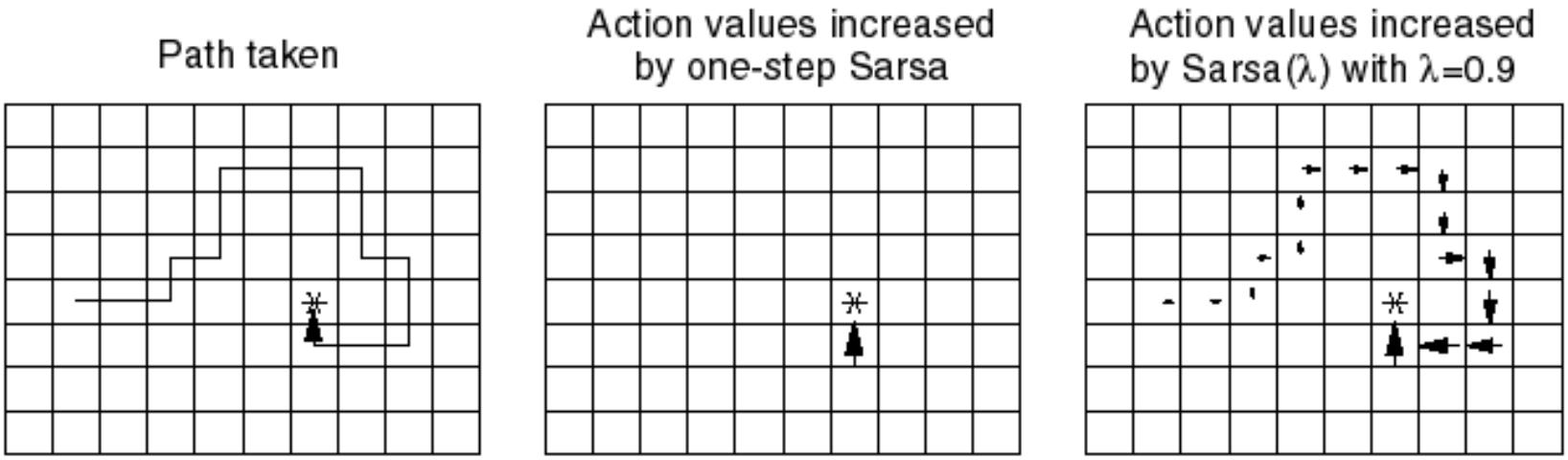
Start with a random network

Play very many games against self

Learn a value function using TD with backpropagation from this simulated experience

This produces arguably the best player in the world

Eligibility traces: Sarsa(λ) Example



- With one trial, the agent has much more information about how to get to the goal
  - not necessarily the *best* way
- Can considerably accelerate learning

What you don't need to know for the exam

- Policy search (Sec. 21.5)
- Formulas for SARSA and Q-Learning update rules
- Exploration and bandits (the box on p. 841)
- Byesian RL
- Robust control theory

That’s all (!)