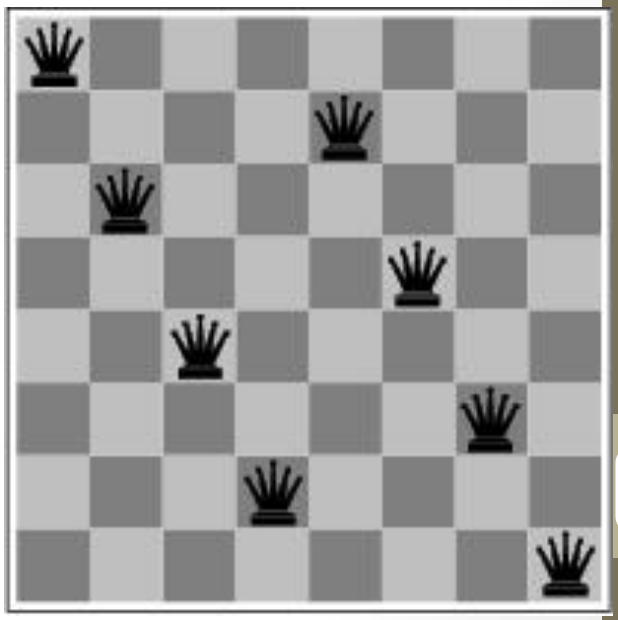
Learning & Search

Part I:

Local Search

Local search and optimization

- Previously: systematic exploration of search space.
  - Path to goal is solution to problem
- YET, for some problems path is irrelevant.
  - E.g., 8-queens puzzle
- Different algorithms can be used
  - Local search



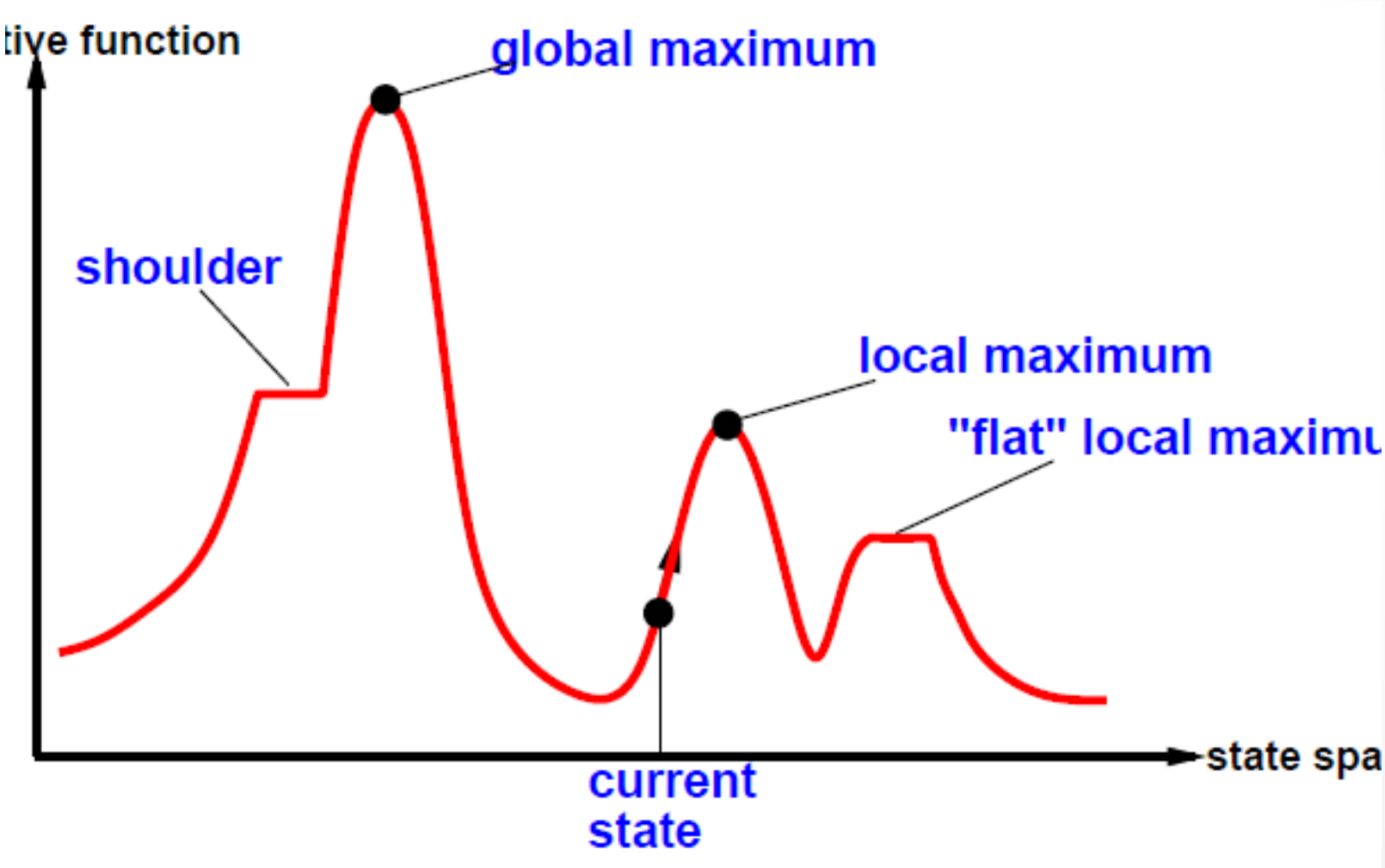
Local search and optimization

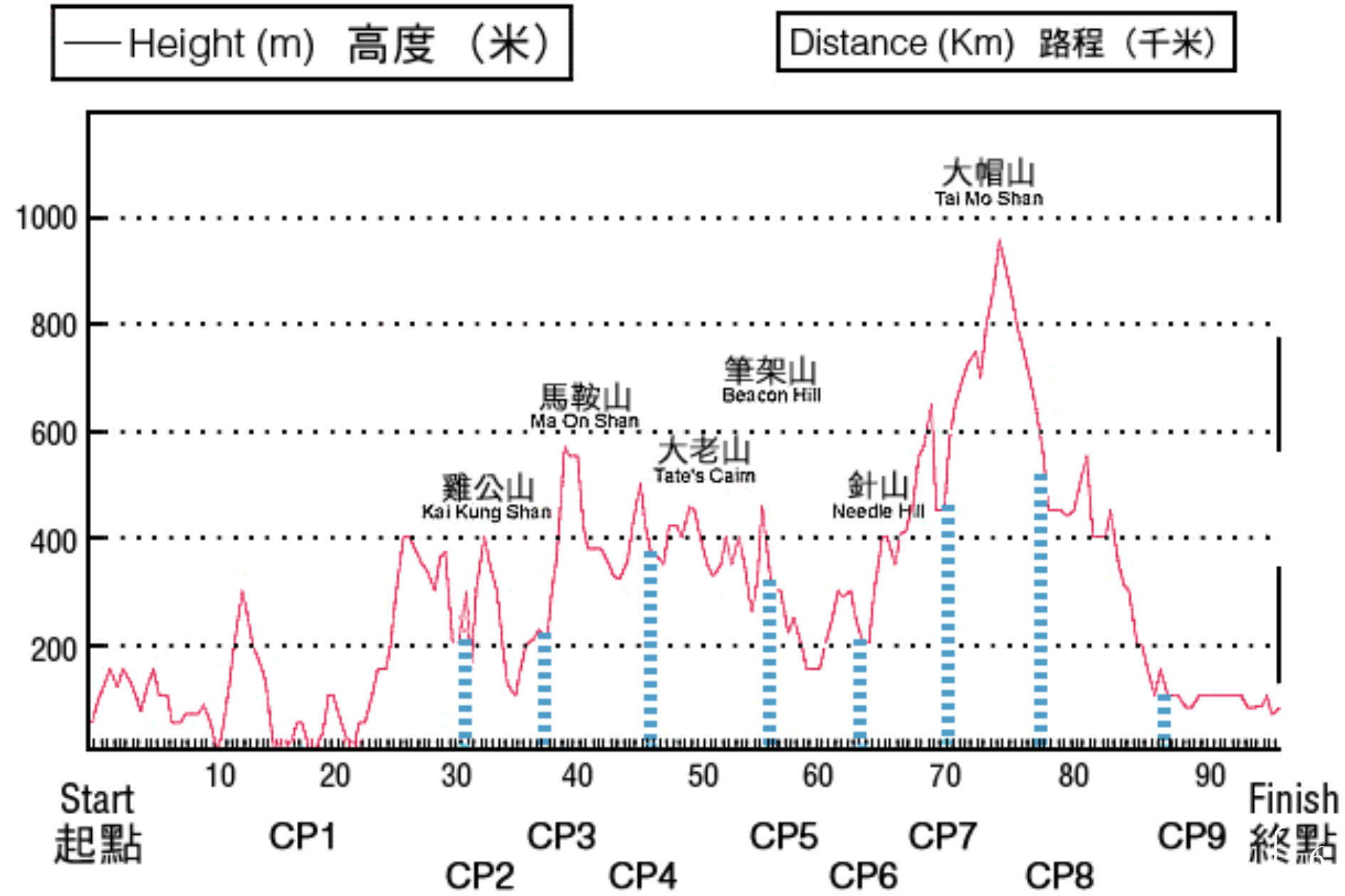
- Local search
  - Keep track of single current state
  - Move only to neighboring states
  - Ignore paths

- Advantages:
  - Use very little memory
  - Can often find reasonable solutions in large or infinite (continuous) state spaces.

- “Pure optimization” problems
  - All states have an objective function
  - Goal is to find state with max (or min) objective value
  - Does not quite fit into path-cost/goal-state formulation
  - Local search can do quite well on these problems.

“Landscape” of search





Copyright © 2006 by Oxfamtrailwalker.org.hk.

Hill-climbing search

function HILL-CLIMBING(*problem*) **return** a state that is a local maximum

input: *problem*, a problem

local variables: *current*, **a node.**

neighbor, **a node.**

current ← MAKE-NODE(INITIAL-STATE[*problem*])

loop do

neighbor ← a highest valued successor of *current*

if VALUE [*neighbor*] ≤ VALUE[*current*] **then return** STATE[*current*]

current ← *neighbor*

Hill-climbing search

- "a loop that continuously moves in the direction of increasing value"
  - terminates when a peak is reached
  - Aka greedy local search

- Value can be either
  - Objective function value
  - Heuristic function value (minimized)

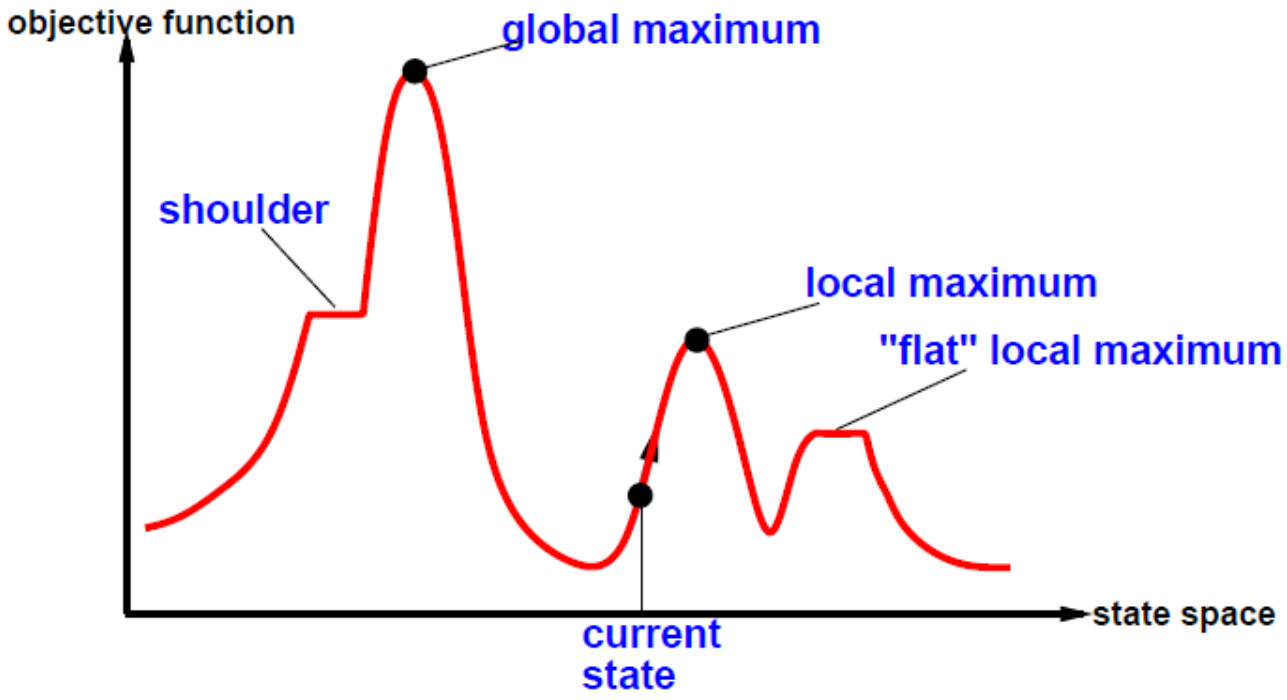
- Hill climbing does not look ahead of the immediate neighbors of the current state.

- Can randomly choose among the set of best successors, if multiple have the best value

- Characterized as "trying to find the top of Mount Everest while in a thick fog"

Hill climbing and local maxima

- When local maxima exist, hill climbing is suboptimal
- Simple (often effective) solution
  - Multiple random restarts



Hill-climbing example

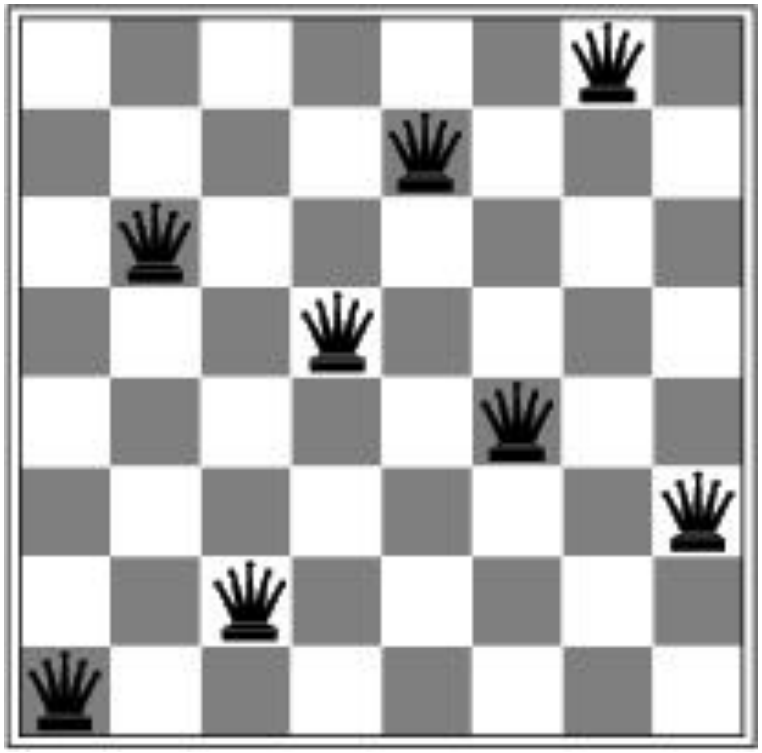
- 8-queens problem, complete-state formulation
  - All 8 queens on the board in some configuration
- Successor function:
  - move a single queen to another square in the same column.
- Example of a heuristic function *h(n)*:
  - the number of pairs of queens that are attacking each other (directly or indirectly)
  - (so we want to minimize this)

Hill-climbing example

Current state:
h=17

Shown is the h-value for each possible successor in each column

A local minimum for 8-queens



A local minimum in the 8-queens state space (h=1)

Performance of hill-climbing on 8-queens

- Randomly generated 8-queens starting states…
  - 14% the time it solves the problem
  - 86% of the time it get stuck at a local minimum

- However…
  - Takes only 4 steps on average when it succeeds
  - And 3 on average when it gets stuck
  - (for a state space with ~17 million states)

Possible solution…sideways moves

- If no downhill (uphill) moves, allow sideways moves in hope that algorithm can escape
  - Need to place a limit on the possible number of sideways moves to avoid infinite loops
- For 8-queens
  - Now allow sideways moves with a limit of 100
  - Raises percentage of problem instances solved from 14 to 94%
  - However….
    - 21 steps for every successful solution
    - 64 for each failure

Genetic algorithms

- Different approach to other search algorithms
  - A successor state is generated by combining two parent states

- A state is represented as a string over a finite alphabet (e.g., binary)
  - 8-queens
    - State = position of 8 queens each in a column

      => 8 x log(8) bits = 24 bits (for binary representation)

      => or 8 digits, each in the range from 1 to 8

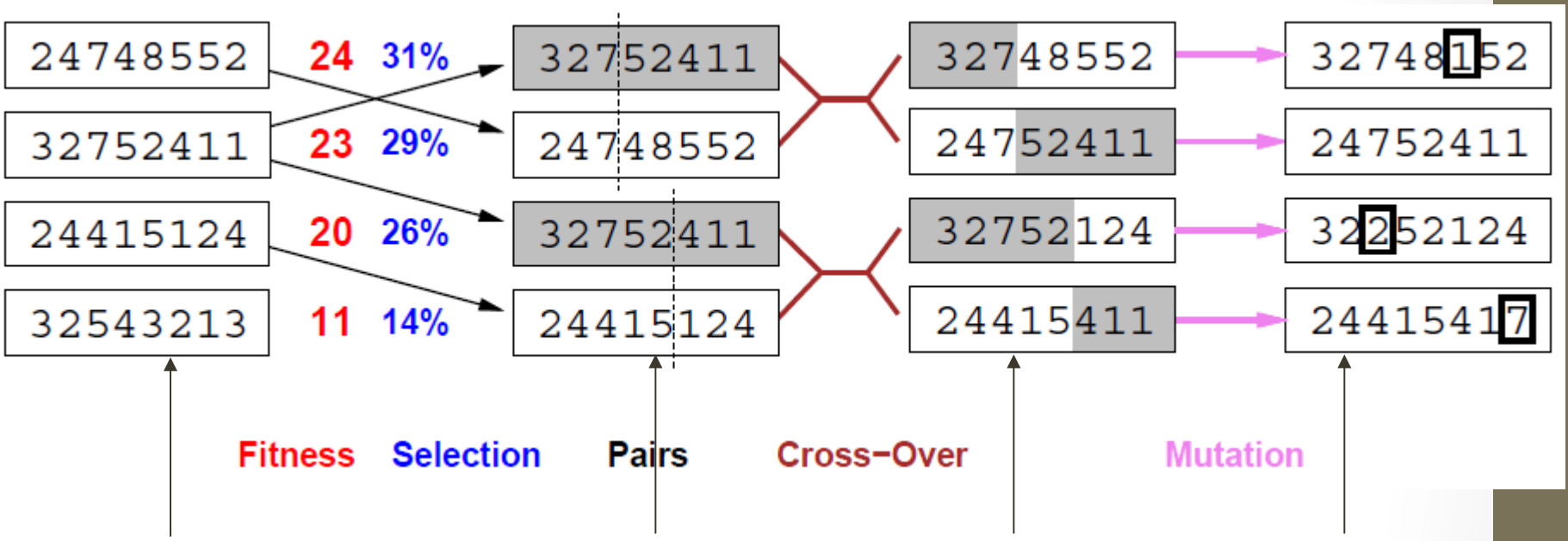
Genetic algorithms

- Start with *k* randomly generated states (population)
- Evaluation function (fitness function).
  - Higher values for better states.
  - Opposite to heuristic function, e.g., # non-attacking pairs in 8-queens
- Produce the next generation of states by "simulated evolution"
  - Random selection
  - Crossover
  - Random mutation

Genetic algorithm pseudocode

function GENETIC_ALGORITHM(*population,* FITNESS-FN)
  return an individual
  input: *population*, a set of individuals
    FITNESS-FN, a function which determines the quality of the individual
  repeat
    new_population ← empty set
    loop for i **from** 1 **to** SIZE(*population*) **do**
      x ← RANDOM_SELECTION(*population*, FITNESS_FN)
      y ← RANDOM_SELECTION(*population*, FITNESS_FN)
      child ← REPRODUCE(*x,y*)
      if (small random probability) **then** *child* ← MUTATE(*child*)
      add *child* to *new_population*
    population ← *new_population*
  until some individual is fit enough or enough time has elapsed
  return the best individual

Genetic algorithms



4 states for 8-queens problem

2 pairs of 2 states randomly selected based on fitness. Random crossover points selected

New states after crossover

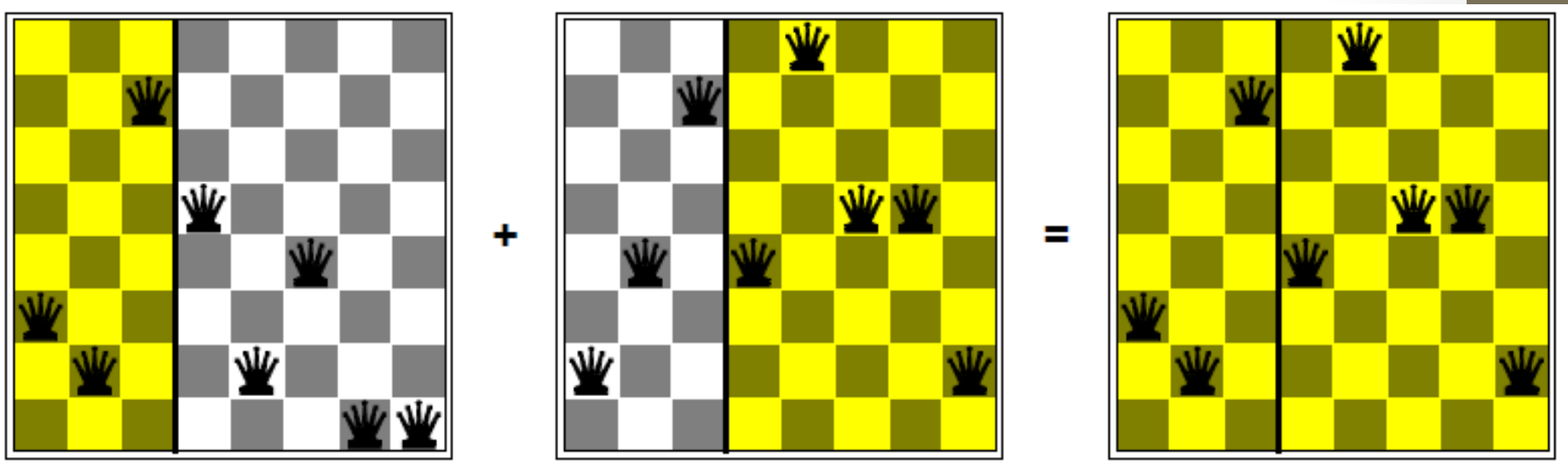
Random mutation applied

Fitness function: number of non-attacking pairs of queens (min = 0, max = 8 × 7/2 = 28)

24/(24+23+20+11) = 31%

23/(24+23+20+11) = 29% etc

Genetic algorithms



Has the effect of “jumping” to a completely different new part of the search space (quite non-local)

Comments on genetic algorithms

- Positive points
  - Random exploration can find solutions that local search can't (via crossover primarily)
  - Appealing connection to human evolution
- Negative points
  - Large number of "tunable" parameters
    - Difficult to replicate performance from one problem to another
  - Lack of good empirical studies comparing to simpler methods
  - Useful on some (small?) set of problems but no convincing evidence that GAs are better than hill-climbing with random restarts in general

Learning

Learning

- Supervised learning
  - the algorithm learns to map given inputs to desired outputs
    - Classification: target variable is discrete (e.g., spam email)
    - Regression: target variable is real-valued (e.g., stock market)
- Unsupervised learning
  - training data does not come with explicit output values; the agent learns to classify inputs on its own
    - Clustering: grouping data into K groups
- Reinforcement learning
  - Learning from feedback/reward, e.g., game-playing agent

Outline

- Different types of learning problems
- Different types of learning algorithms
- Supervised learning
  - Decision trees
  - Perceptrons, Multi-layer Neural Networks

Simple illustrative learning problem

Problem:

decide whether to wait for a table at a restaurant, based on the following attributes:

1. Alternate: is there an alternative restaurant nearby?
2. Bar: is there a comfortable bar area to wait in?
3. Fri/Sat: is today Friday or Saturday?
4. Hungry: are we hungry?
5. Patrons: number of people in the restaurant (None, Some, Full)
6. Price: price range ($, $$, $$$)
7. Raining: is it raining outside?
8. Reservation: have we made a reservation?
9. Type: kind of restaurant (French, Italian, Thai, Burger)
10. WaitEstimate: estimated waiting time (0-10, 10-30, 30-60, >60)

Training Data for Supervised Learning

Example	Attributes										Target
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*Wait*
X_1	T	F	F	T	Some	$$$	F	T	French	0–10	T
X_2	T	F	F	T	Full	$	F	F	Thai	30–60	F
X_3	F	T	F	F	Some	$	F	F	Burger	0–10	T
X_4	T	F	T	T	Full	$	F	F	Thai	10–30	T
X_5	T	F	T	F	Full	$$$	F	T	French	>60	F
X_6	F	T	F	T	Some	$$	T	T	Italian	0–10	T
X_7	F	T	F	F	None	$	T	F	Burger	0–10	F
X_8	F	F	F	T	Some	$$	T	T	Thai	0–10	T
X_9	F	T	T	F	Full	$	T	F	Burger	>60	F
X_{10}	T	T	T	T	Full	$$$	F	T	Italian	10–30	F
X_{11}	F	F	F	F	None	$	F	F	Thai	0–10	F
X_{12}	T	T	T	T	Full	$	F	F	Burger	30–60	T

Training set of N=12 example input-output pairs

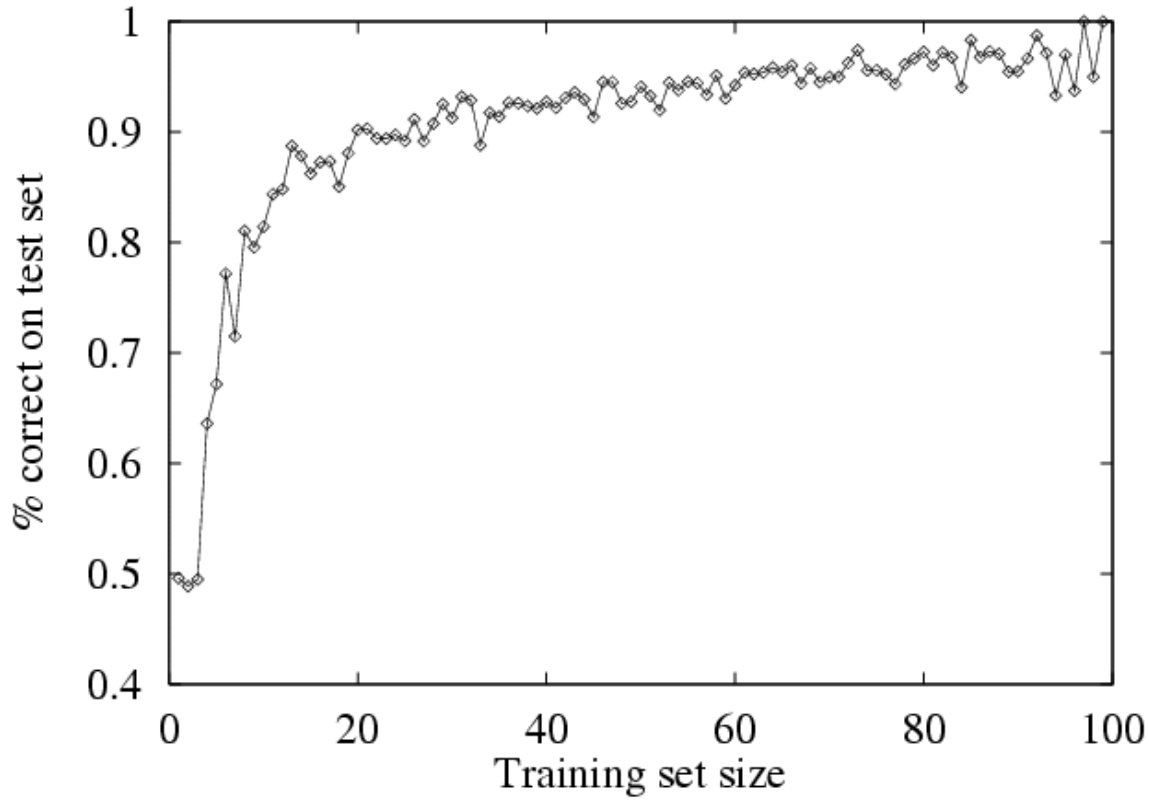
Training & Testing

- To show how more data eliminates the problems of sparse data and noise, separate data into training and test sets.
  - After creating model based on examples in training set, put test cases through decision tree and record the percentage that get classified accurately.
  - The result is that performance improves as training size increases

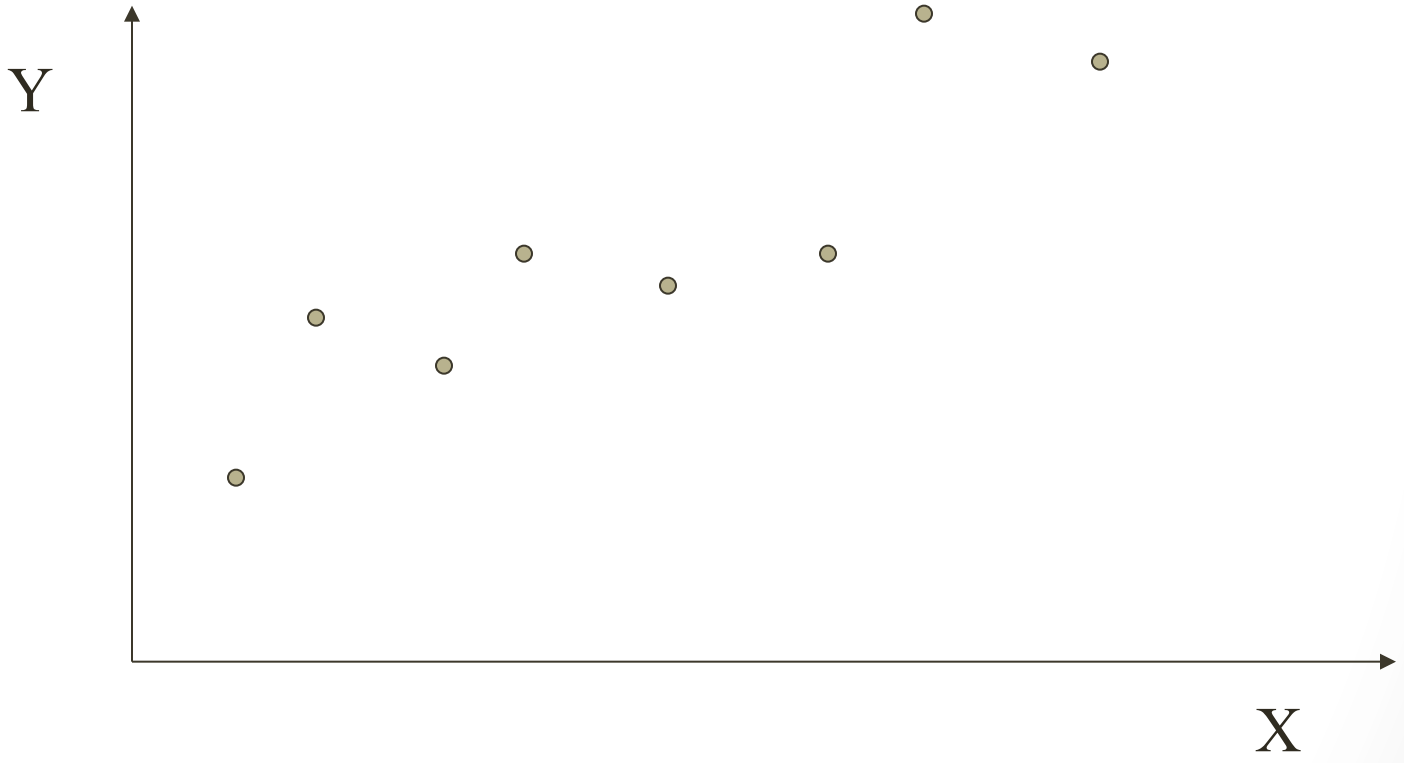
Example of Test Performance

Restaurant problem

- simulate 100 data sets of different sizes
- train on this data, and assess performance on an independent test set
- learning curve = plotting accuracy as a function of training set size

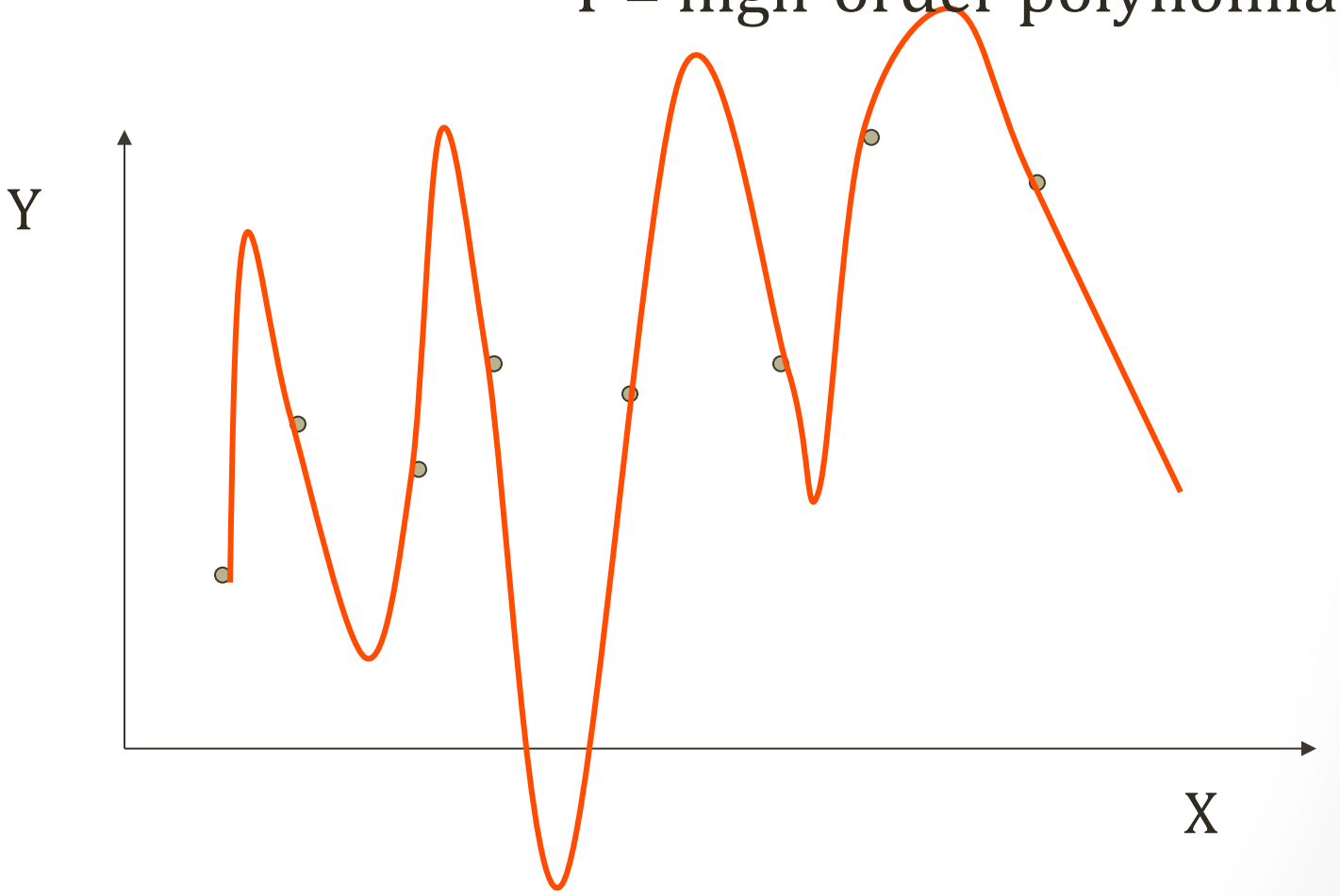


Overfitting and Underfitting

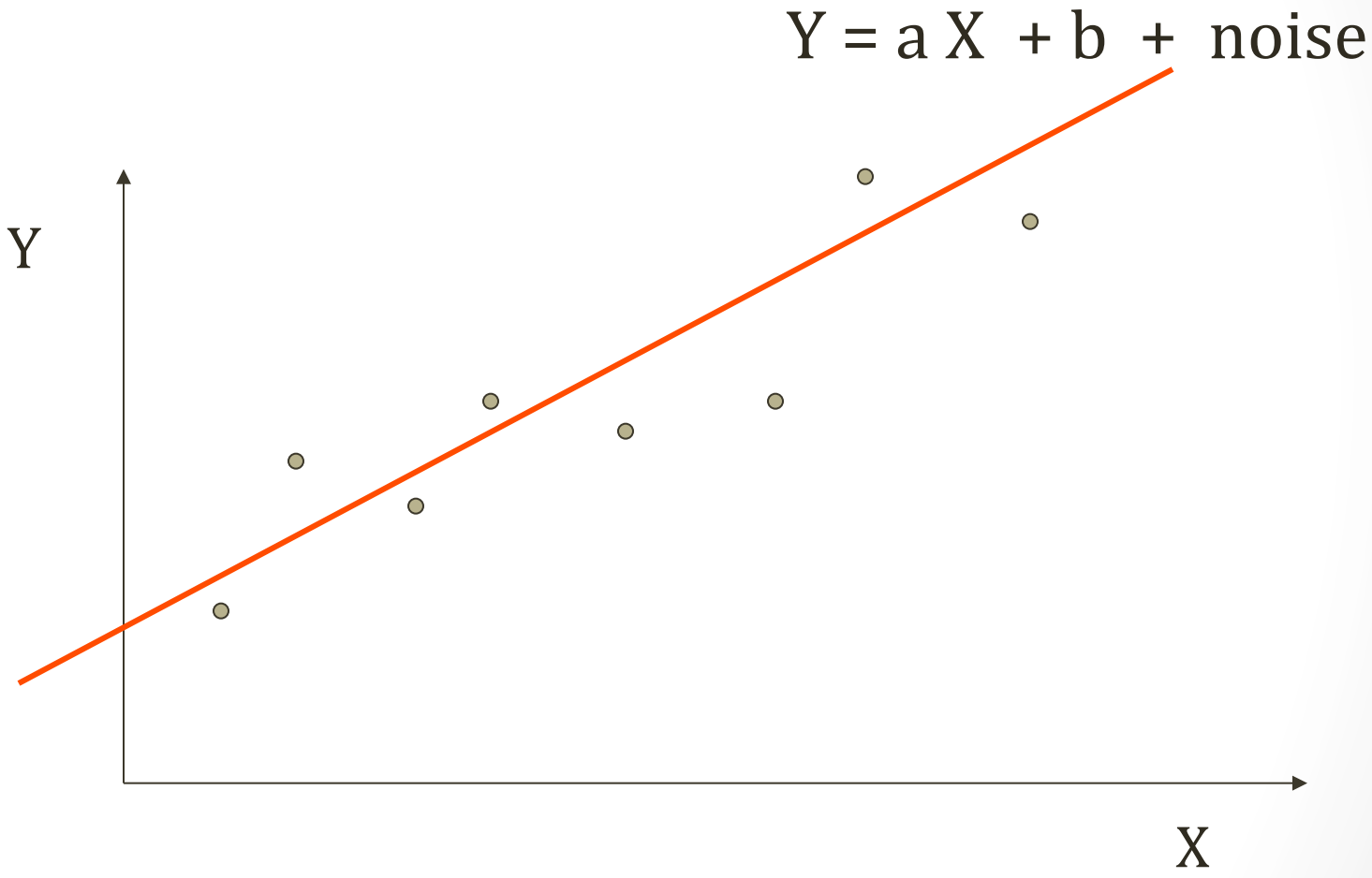


A Complex Model

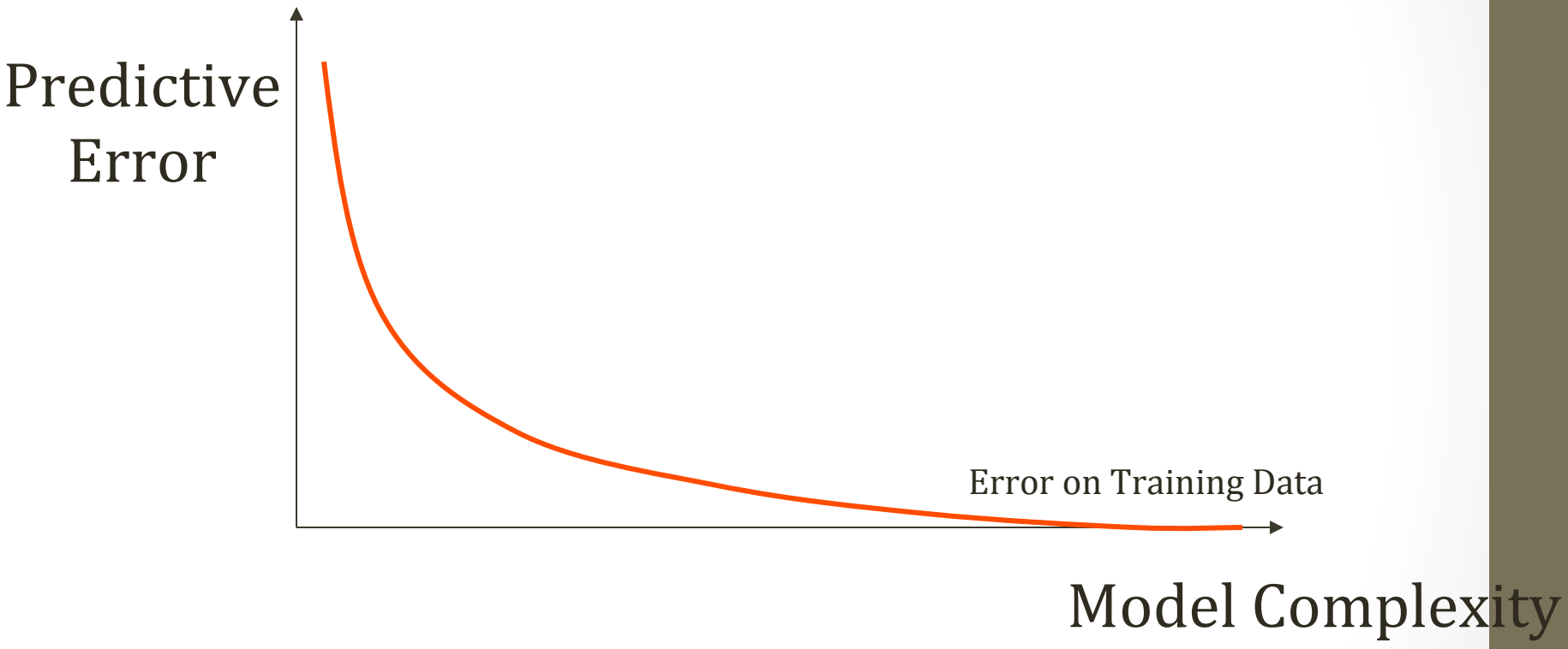
Y = high-order polynomial in X



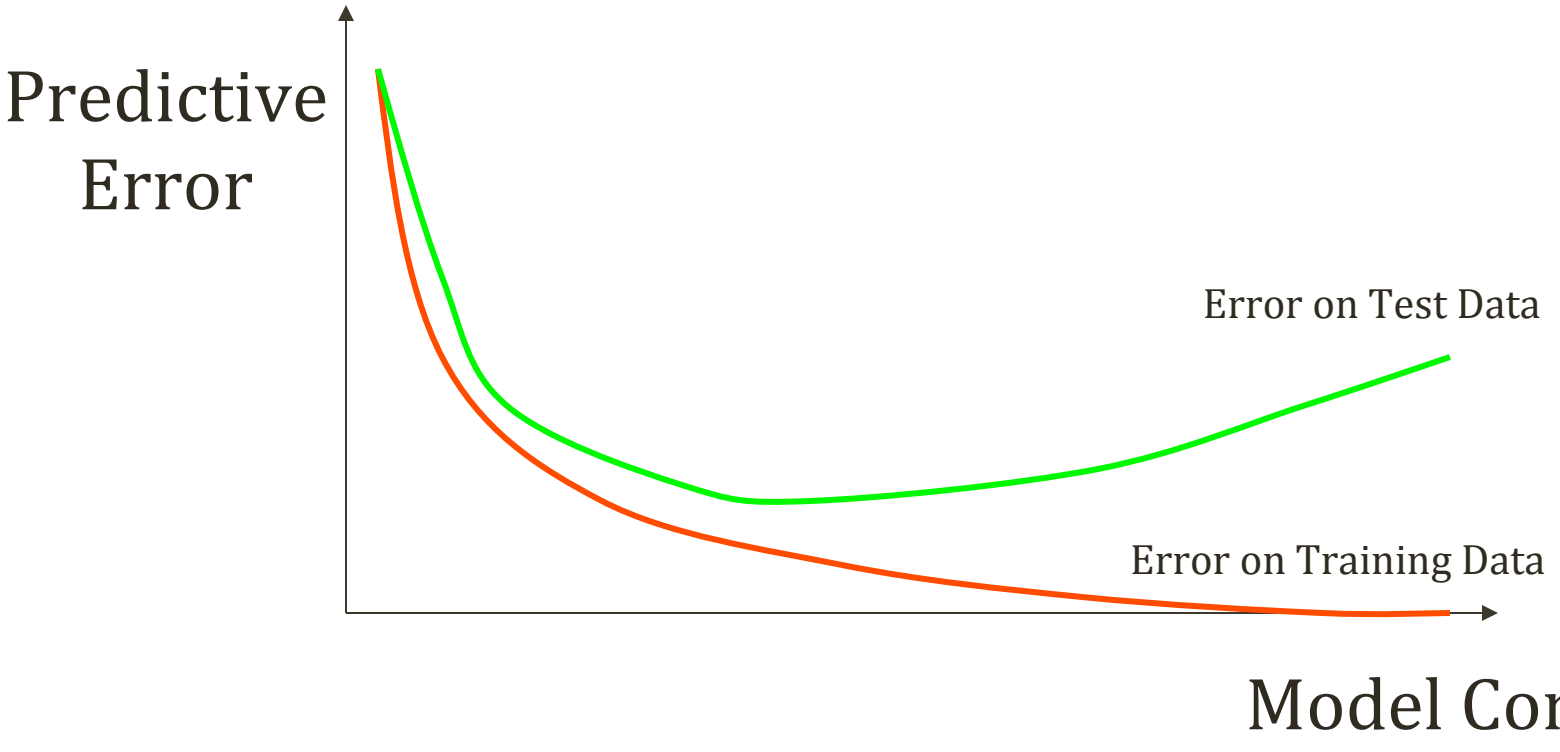
A Much Simpler Model



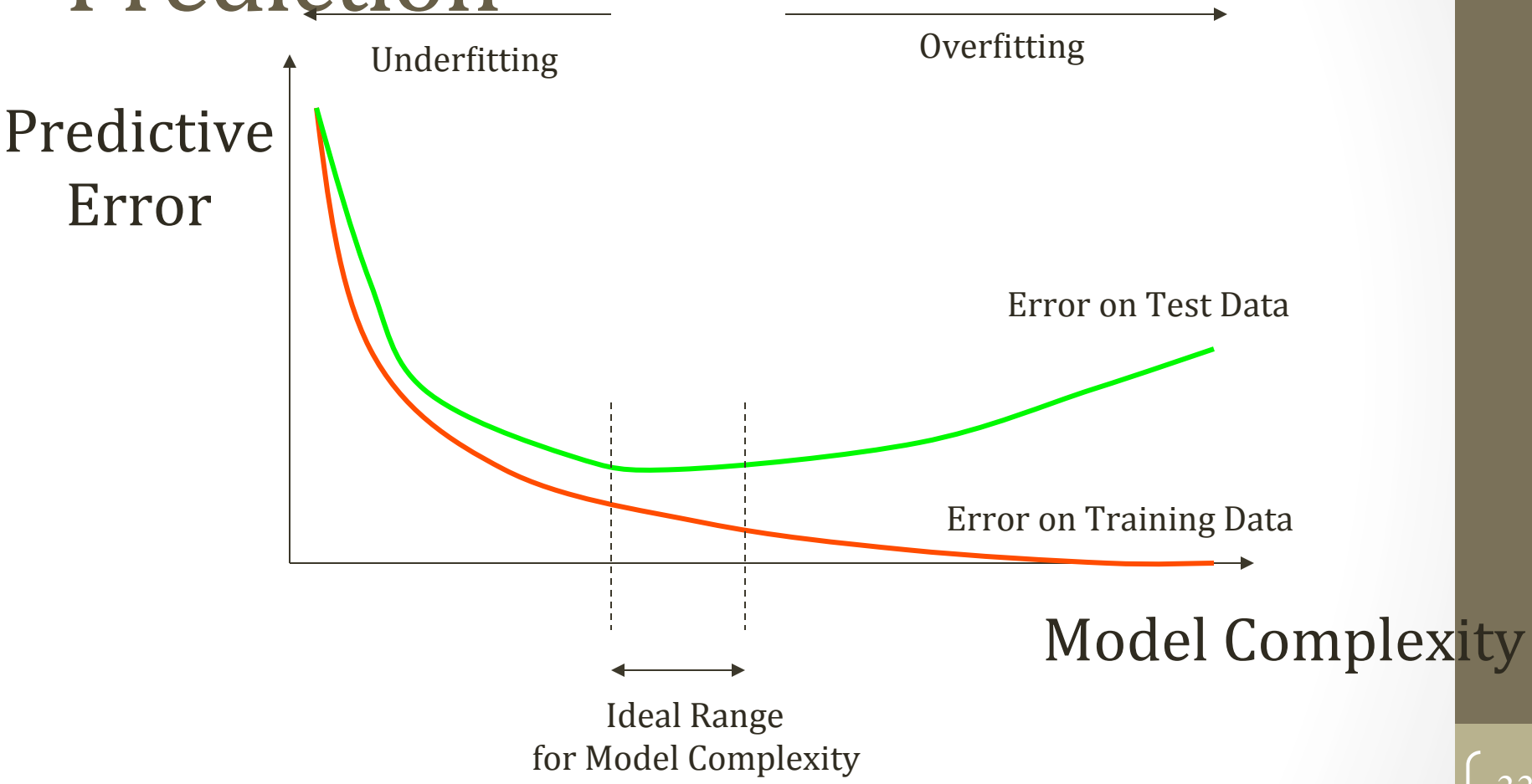
How Overfitting affects Prediction



How Overfitting affects Prediction



How Overfitting affects Prediction



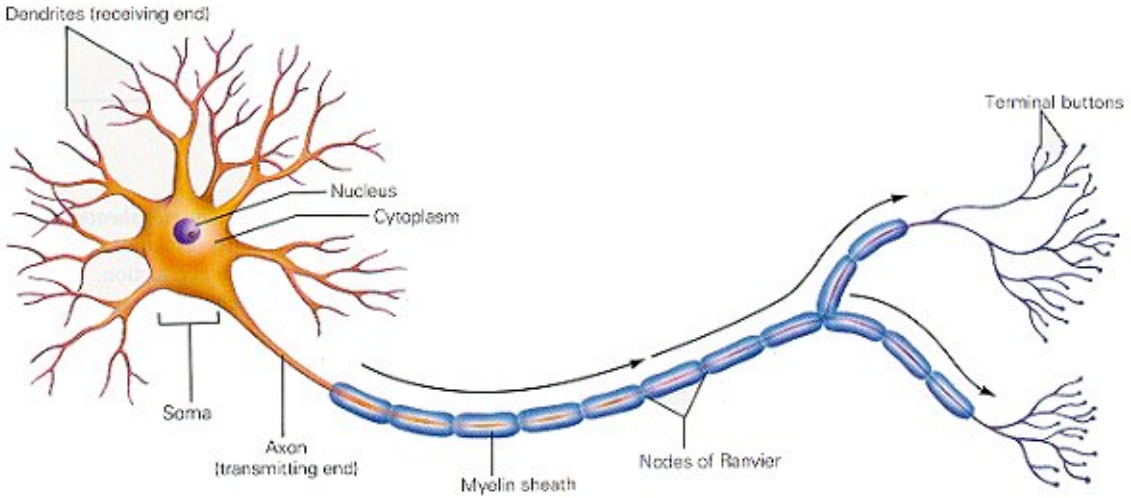
Part II:

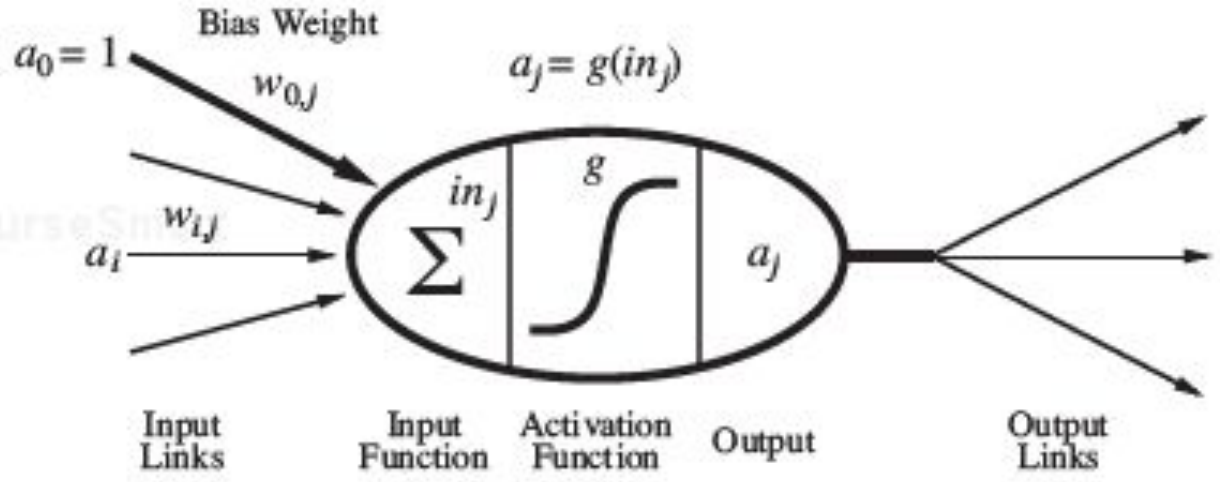
Artificial Neural Networks

Artificial Neural Networks

- Artificial neural networks (or simply neural networks) are made up of a series of nodes that represent the neurons of the brain
  - Each node (or unit) is connected to other nodes through directed links
  - Each node is defined by its input function, which sums the activations energies from other nodes, and an activation function that produces an output value for this node, based on the result of the input function.

Artificial Neural Networks





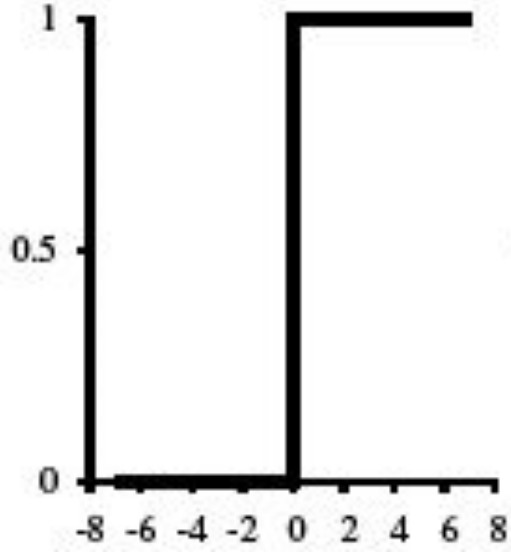
Neural Units

- Input function
  - the input function is a sum of the weighted inputs coming into the neural unit *j*
  - $in_j = \Sigma_i w_{i,j} a_i$
  - the weights on each link are adjusted to achieve the learning mechanism for the neural network

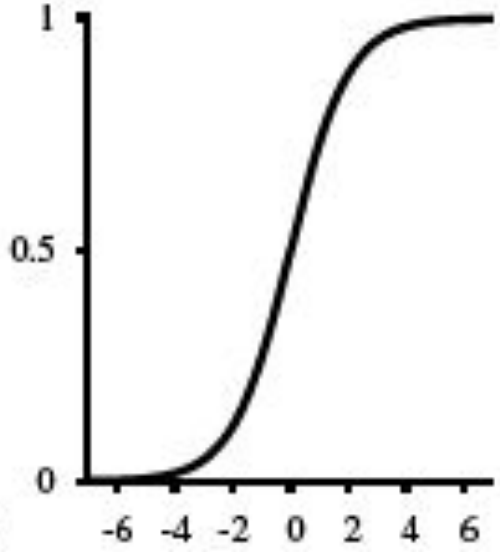
Neural Units (cont'd)

- Activation function : $g(in_j)$

Thresholded output, takes values 0 or 1



Sigmoid output, takes real values between 0 and 1



$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{else} \end{cases}$$

sigmoid(x) = $1 / (1+e^{-x})$

Neural Units (cont'd)

- Activation function : $g(in_j)$

Thresholded output, takes values +1 or -1

$a_j = \text{sign}(\Sigma\, w_{i,j}\, a_i)$

Sigmoid output, takes real values between -1 and +1

tanh(x)

sigmoid(x) = $2 / (1+e^{-x}) - 1$

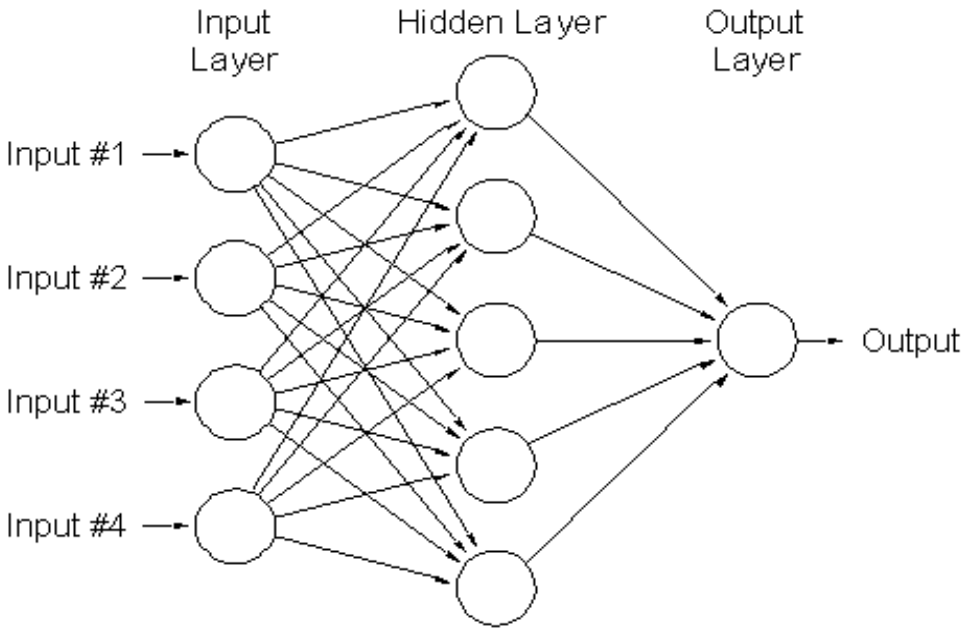
Neural Network Structure

- Two major kinds of neural networks:
  - Feed-Forward (acyclic) networks: node structures where the directed links between them do not form any loops
  - Recurrent (cyclic) networks: some nodes link back to previous nodes in the structure
    - important for maintaining state information from one reading of the neural network to the other

Neural Network Structure

- Every neural network has three layers:
  - input layer: the nodes that record the input values for the training case, one node per input value (no processing)
  - output layer: the nodes whose activation functions produce the output values for the neural network
  - hidden layer(s): the nodes between the input and output layers

Hidden Layer "Rules"



- Hidden layers of neural networks store internal "knowledge", used in processing the output
- Rules of Thumb:
  - no universal rule for number of no
    more nodes implies more detail in
    overfitting, in sparse data cases
  - some problems require no hidden layer (i.e., linear separation problems)
    - use single-layer feed-forward neural network → perceptron

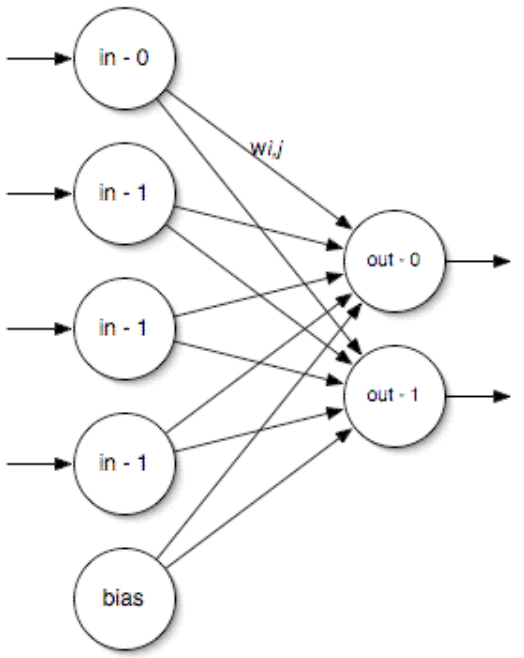
Perceptrons

- No hidden layer
- How does learning occur within the network?
  - want to minimize the squared error of the neural network

$$E = \tfrac{1}{2}Err^2 \equiv \tfrac{1}{2}(y - h_w(x))^2$$

  - *y* is the true output that should be

expected, given the input *x*,
weights *w* and actual output $h_w(x)$



Perceptron Learning

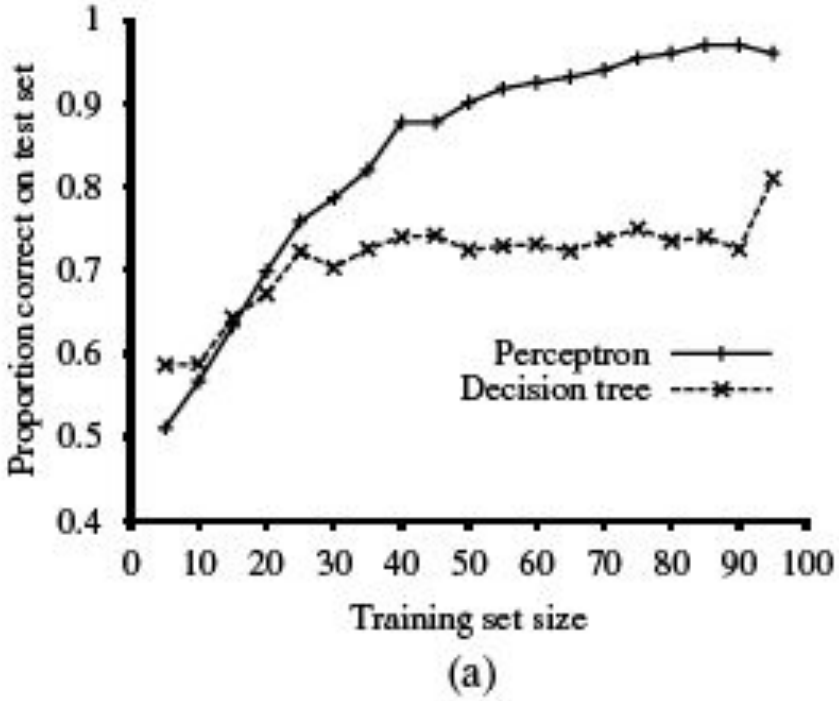
- In order to minimize this error, find the difference between the expected and actual output, and use that to adjust the weights that affect that output

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

  - current value of weight $w_{j,k}$
  - learning rate α → affects learning steps in algorithm

- *Note:* sigmoid activation function makes the rate of change calculation easy, since derivative is well-formed

Comparing Performance of perceptrons and decision trees



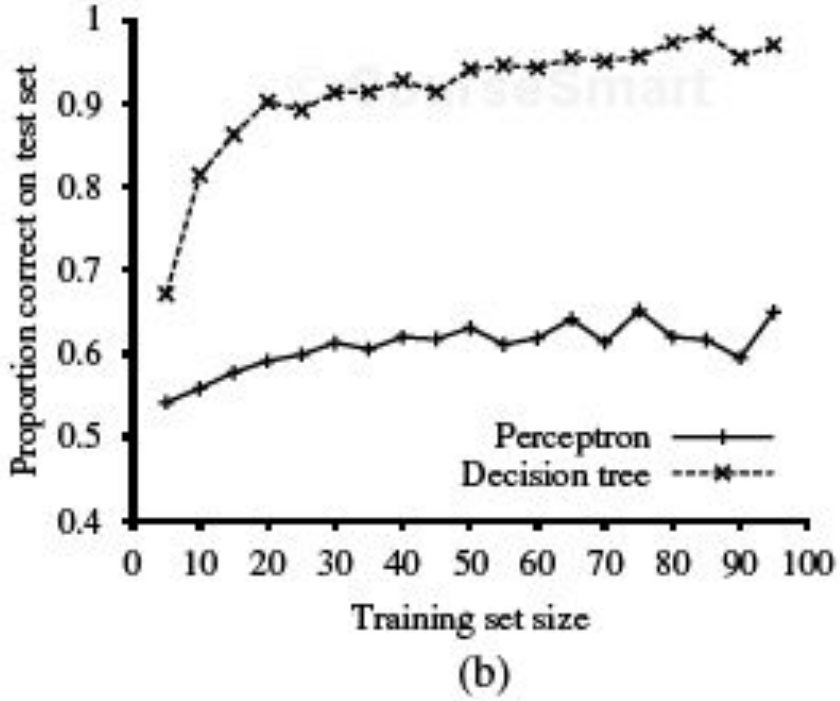


Figure 18.22 Comparing the performance of perceptrons and decision trees. (a) Perceptrons are better at learning the majority function of 11 inputs. (b) Decision trees are better at learning the *WillWait* predicate in the restaurant example.

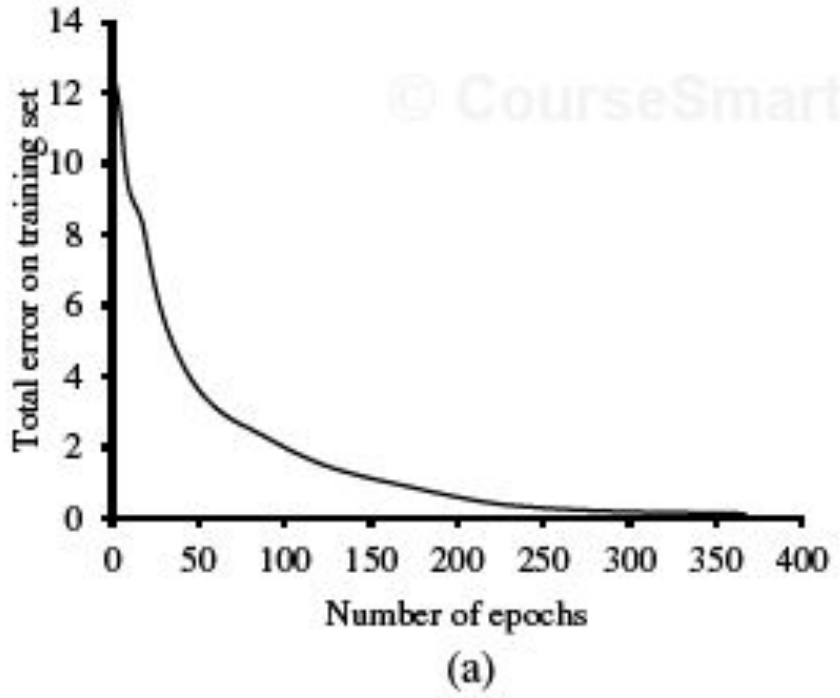
Multilayer Learning Back-Propagation

function BACK-PROP-LEARNING(*examples*, *network*) **returns** a neural network
inputs: *examples*, a set of examples, each with input vector **x** and output vector **y**
network, a multilayer network with L layers, weights $w_{i,j}$, activation function g
local variables: Δ, a vector of errors, indexed by network node

repeat
for each weight $w_{i,j}$ in *network* **do**
$w_{i,j} \leftarrow$ a small random number
for each example (**x**, **y**) **in** *examples* **do**
/* *Propagate the inputs forward to compute the outputs* */
for each node i in the input layer **do**
$a_i \leftarrow x_i$
for $\ell = 2$ **to** L **do**
for each node j in layer ℓ **do**
$in_j \leftarrow \sum_i w_{i,j}\, a_i$
$a_j \leftarrow g(in_j)$
/* *Propagate deltas backward from output layer to input layer* */
for each node j in the output layer **do**
$\Delta[j] \leftarrow g'(in_j) \times (y_j - a_j)$
for $\ell = L - 1$ **to** 1 **do**
for each node i in layer ℓ **do**
$\Delta[i] \leftarrow g'(in_i) \sum_j w_{i,j}\, \Delta[j]$
/* *Update every weight in network using deltas* */
for each weight $w_{i,j}$ in *network* **do**
$w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j]$
until some stopping criterion is satisfied
return *network*

Figure 18.24 The back-propagation algorithm for learning in multilayer networks.

Comparing Performance



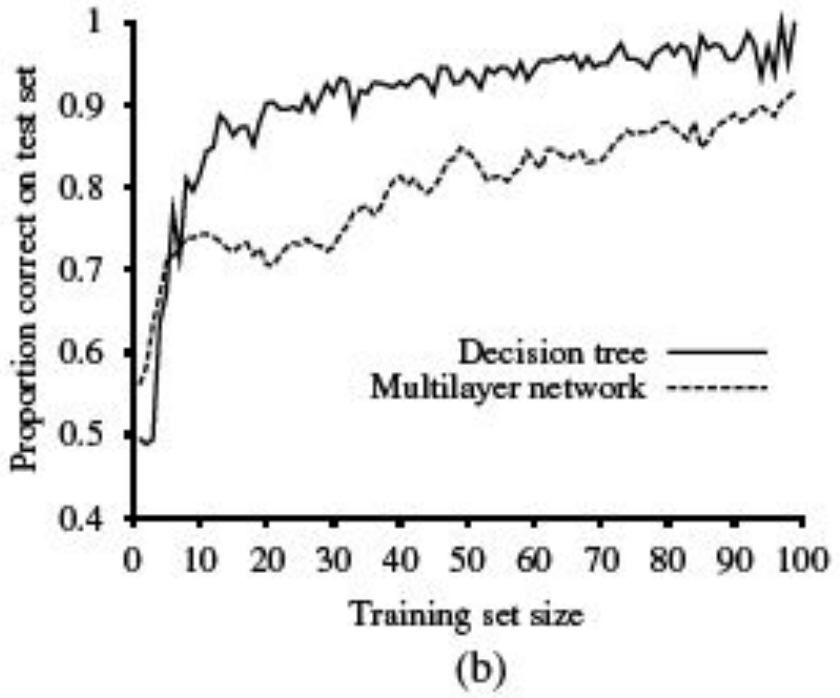


Figure 18.25 (a) Training curve showing the gradual reduction in error as weights are modified over several epochs, for a given set of examples in the restaurant domain. (b) Comparative learning curves showing that decision-tree learning does slightly better on the restaurant problem than back-propagation in a multilayer network.

Neural Network Structure – Part II

- Instead of only using fully connected networks and changing weights and trying different numbers of layers and their sizes we could try the following:
  - optimal brain damage: start with a fully connected network and remove connections and/or units from it if this does not decrease its performance
  - growing larger networks, e.g., tiling: start with a single unit and add units that take care of examples the first unit got wrong

Qualities of Neural Networks

- Advantages:
  - Biological basis
    - learning models human learning techniques; more sound than other contrived techniques
  - Auto-organization
    - internal representation of knowledge is not outlined or confined to specific structure
  - Fault tolerance
    - partial destruction of network structure is compensated for by parallel operation of remaining network
  - Flexibility
    - learned model can work on unseen data
  - Speed
    - once model is trained, responses are obtained in real-time
- Disadvantages
  - Arcane quality
    - blackbox: can't inspect internal workings to determine what it learned as hidden state values

Readings

- Russell & Norvig
  - Chapter 4
  - Chapter 18