

---

# Machine Learning

## Greedy Local Search

# ML in a Nutshell

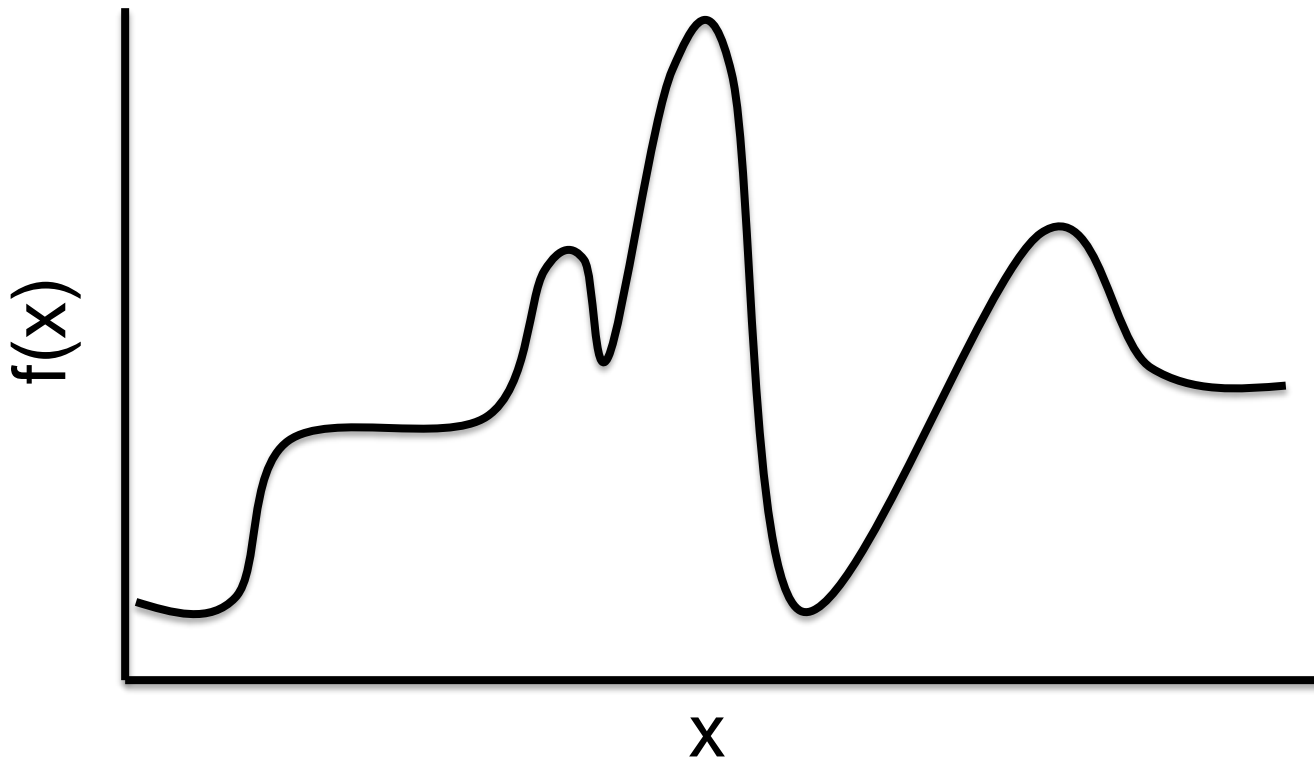
---

- Every machine learning algorithm has three components:
  - Representation
    - E.g., Decision trees, instances
  - Evaluation
    - E.g., accuracy on test set
  - **Optimization**
    - How do you **find** the best hypothesis?

# Hill-climbing (greedy local search)

---

$$\text{find } x_{\max} = \arg \max_{x \in X} (f(x))$$



# Greedy local search needs

---

- A “successor” function
  - Says what states I can reach from the current one.
  - Often implicitly a distance measure.
- An objective (error) function
  - Tells me how good a state is
- Enough memory to hold
  - The best state found so far
  - The current state
  - The state it’s considering moving to

# Hill-climbing search

---

- "Like climbing Everest in thick fog with amnesia"

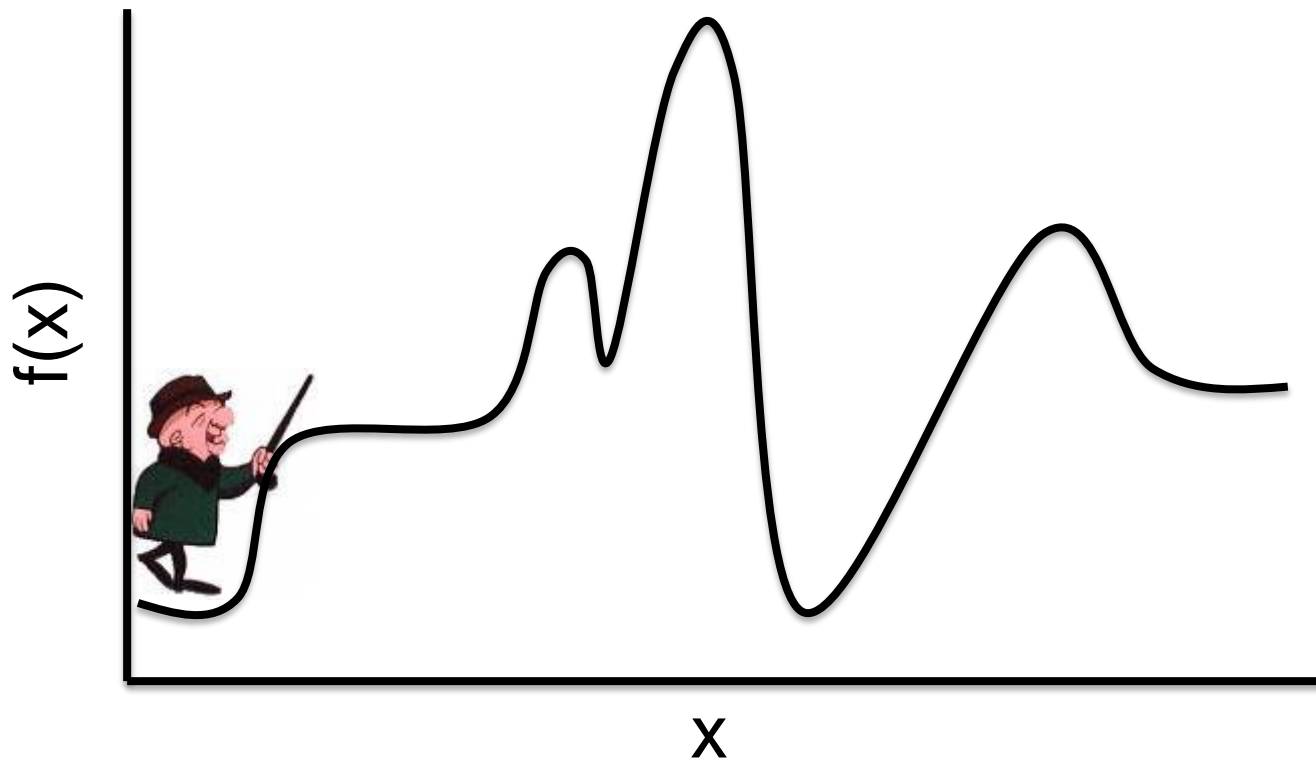
```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  inputs: problem, a problem
  local variables: current, a node
                  neighbor, a node

  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 (greedy local search)

---

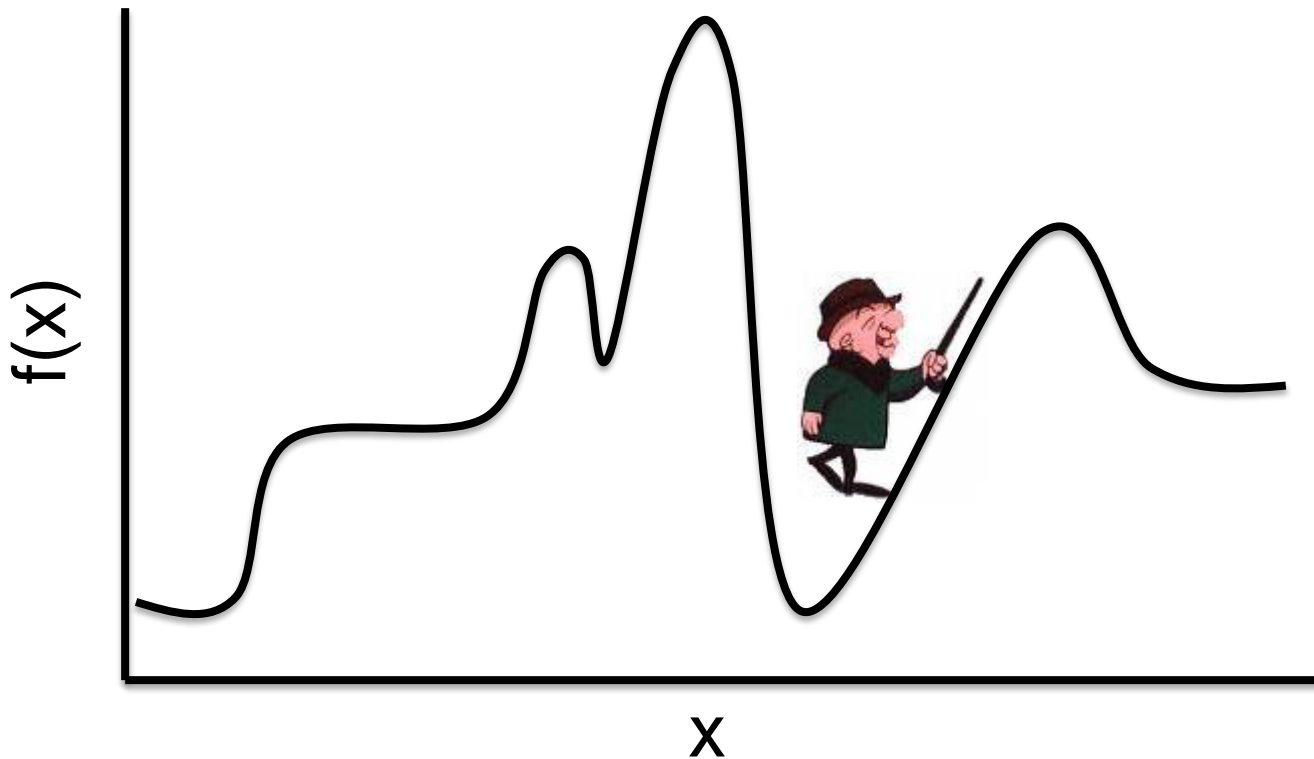
"Like climbing Everest in thick fog with amnesia"



# Hill-climbing (greedy local search)

---

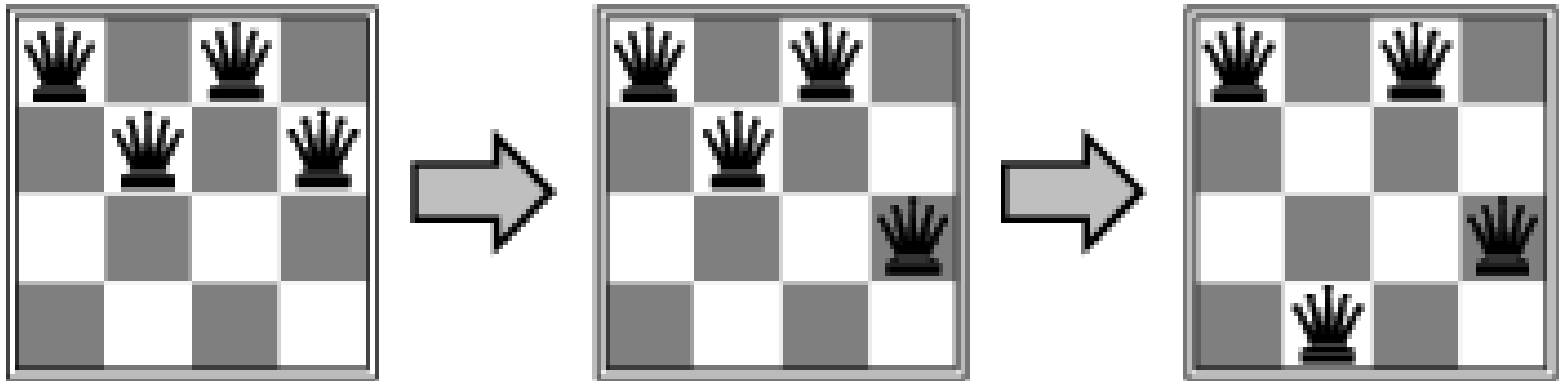
It is easy to get stuck in local maxima



# Example: $n$ -queens

---

- Put  $n$  queens on an  $n \times n$  board with no two queens on the same row, column, or diagonal
- 





# Greedy local search needs









---

- A “successor” (distance?) function  
Any board position that is reachable by moving one queen in her column.
- An optimality (error?) measure  
How many queen pairs can attack each other?

# Hill-climbing search: 8-queens problem

---

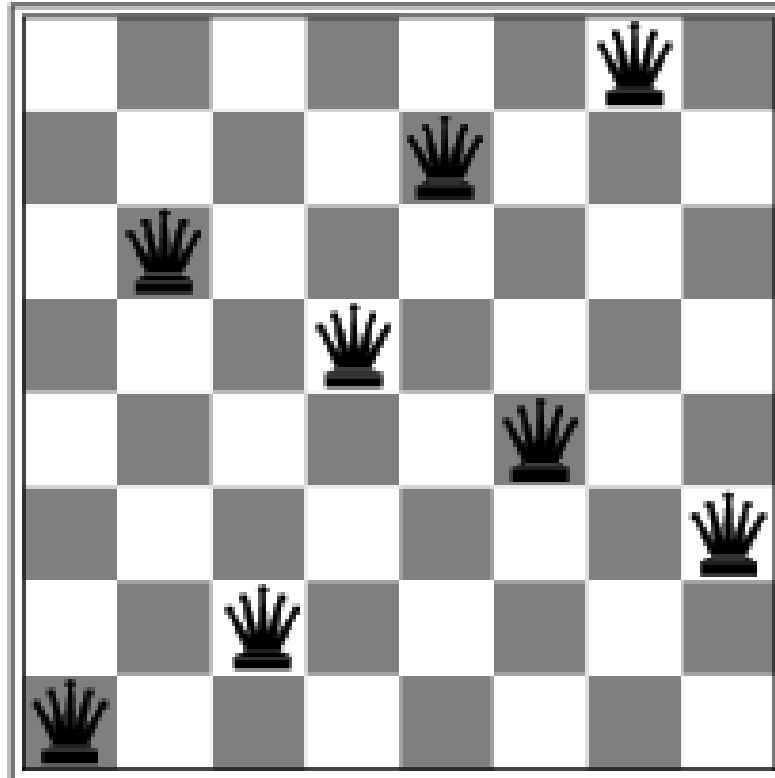
$h = 17 \rightarrow$

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14		13	16	13	16
	14	17	15		14	16	16
17		16	18	15		15	
18	14		15	15	14		16
14	14	13	17	12	14	12	18

- $h$  = number of pairs of queens that are attacking each other, either directly or indirectly

# Hill-climbing search: 8-queens problem

---



- A local minimum with  $h = 1$

# Simulated annealing search

---

- Idea: escape local maxima by allowing some "bad" moves but **gradually decrease** their frequency

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
           schedule, a mapping from time to "temperature"
  local variables: current, a node
                   next, a node
                   T, a "temperature" controlling prob. of downward steps

  current ← MAKE-NODE(INITIAL-STATE[problem])
  for t ← 1 to ∞ do
    T ← schedule[t]
    if T = 0 then return current
    next ← a randomly selected successor of current
     $\Delta E \leftarrow \text{VALUE}[\textit{next}] - \text{VALUE}[\textit{current}]$ 
    if  $\Delta E > 0$  then current ← next
    else current ← next only with probability  $e^{\Delta E/T}$ 
```

# Properties of simulated annealing

---

- One can prove: If  $T$  decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1
- Widely used in VLSI layout, airline scheduling, etc

# Let's look at a demo

---

[yuval.bar-or.org/index.php?item=9](http://yuval.bar-or.org/index.php?item=9)



# Results on 8-queens

---

	Random	Sim Anneal	Greedy
	600+	173	4
	15	119	4
	154	114	5
<b>Average</b>	<b>256+</b>	<b>135</b>	<b>4</b>

- Note: on other problems, your mileage may vary

# Continuous Optimization

---

- Many AI problems require optimizing a function  $f(\mathbf{x})$ , which takes continuous values for input vector  $\mathbf{x}$
- Huge research area
- Examples:
  - **Machine Learning**
  - Signal/Image Processing
  - Computational biology
  - Finance
  - Weather forecasting
  - Etc., etc.



# Local beam search

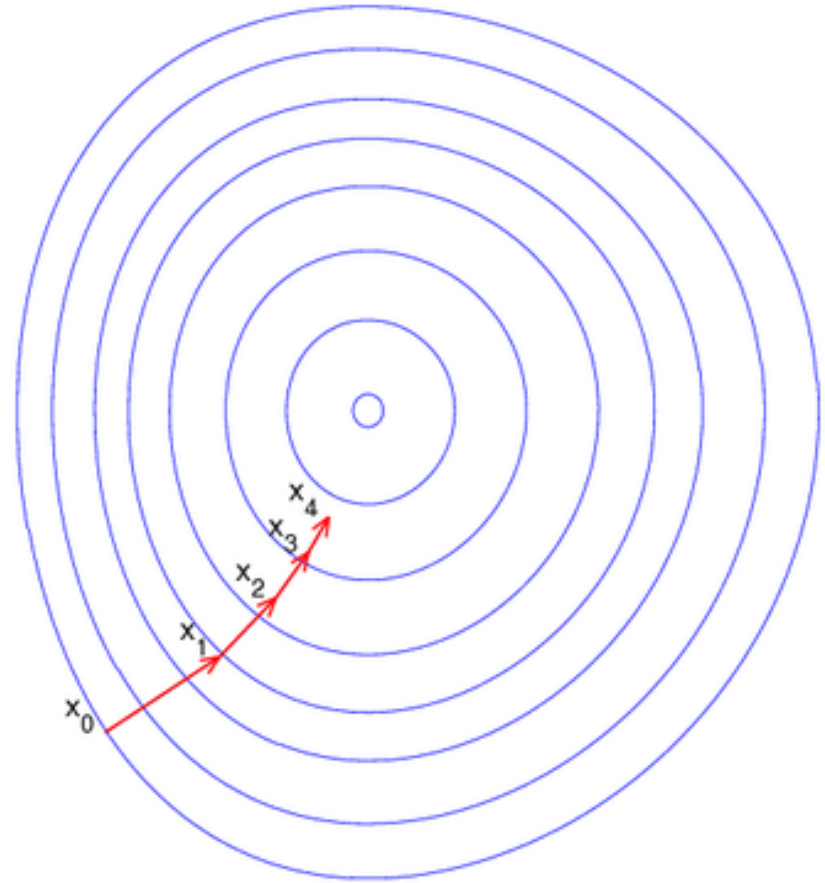
---

- Keep track of  $k$  states rather than just one
- Start with  $k$  randomly generated states
- At each iteration, all the successors of all  $k$  states are generated
- If any one is a goal state, stop; else select the  $k$  best successors from the complete list and repeat.

# Gradient Ascent

---

- Idea: move in direction of steepest ascent (gradient)
- $\mathbf{x}_k = \mathbf{x}_{k-1} + \eta \nabla f(\mathbf{x}_{k-1})$



<b>x</b>	<b>f(x)</b>	<b>gradient</b>	<b>xnew</b>
5.000	9.000	6.000	4.000
4.000	4.000	4.000	3.333
3.333	1.778	2.667	2.889
2.889	0.790	1.778	2.593
2.593	0.351	1.185	2.395
2.395	0.156	0.790	2.263
2.263	0.069	0.527	2.176
2.176	0.031	0.351	2.117
2.117	0.014	0.234	2.078
2.078	0.006	0.156	2.052
2.052	0.003	0.104	2.035
2.035	0.001	0.069	2.023
2.023	0.001	0.046	2.015
2.015	0.000	0.031	2.010

# Types of Optimization

---

- Linear vs. non-linear
- Analytic vs. Empirical Gradient
- Convex vs. non-convex
- Constrained vs. unconstrained

# Continuous Optimization in Practice

---

- *Lots* of previous work on this
- Use packages

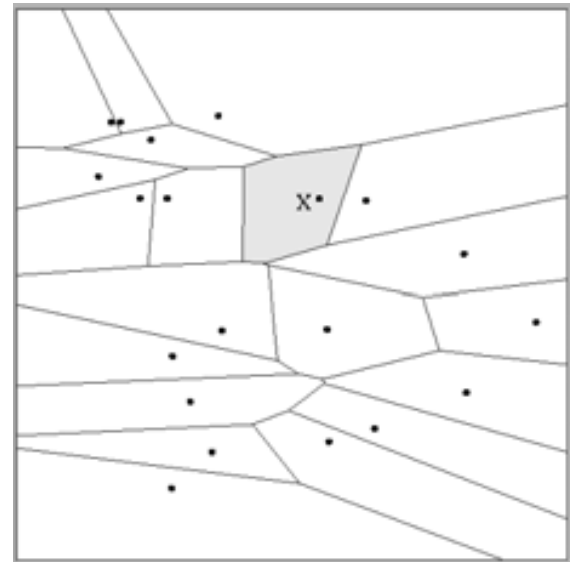
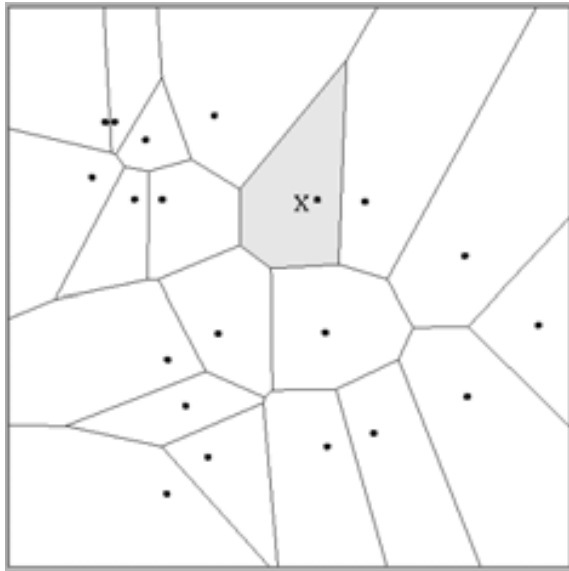
# Rules of Thumb: Gradient Descent

---

- Stochastic vs. Batch
  - Try stochastic first
- Analytic Gradients are hard to debug
  - Use packages with gradients built in (e.g. Tensorflow)
  - Do gradient checking

# Final example: weights in NN

---



$$d(x, y) = (x_1 - y_1)^2 + (x_2 - y_2)^2 \quad d(x, y) = (x_1 - y_1)^2 + (3x_2 - 3y_2)^2$$

# Reading

---

- [Gradient Descent](#)
  - See e.g. the python example midway down the page
- Previous:
  - Nearest neighbor ([Elements of Statistical Learning](#) 2.3, 13.3)