Machine Learning

Greedy Local Search

With slides from Bryan Pardo, Stuart Russell
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)

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

- \( h = 17 \) = 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

```python
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
        ΔE ← VALUE[next] − VALUE[current]
        if ΔE > 0 then current ← next
        else current ← next only with probability e^ΔE/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

INSERT DEMO HERE
## Results on 8-queens

<table>
<thead>
<tr>
<th></th>
<th>Random</th>
<th>Sim Anneal</th>
<th>Greedy</th>
</tr>
</thead>
<tbody>
<tr>
<td>600+</td>
<td>173</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>119</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>154</td>
<td>114</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>256+</strong></td>
<td><strong>135</strong></td>
<td><strong>4</strong></td>
</tr>
</tbody>
</table>

- Note: on other problems, your mileage may vary
Continuous Optimization

- Many AI problems require optimizing a function $f(x)$, which takes continuous values for input vector $x$

- 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)

\[ \mathbf{x}_k = \mathbf{x}_{k-1} + \eta \nabla f(\mathbf{x}_{k-1}) \]
<table>
<thead>
<tr>
<th>$x$</th>
<th>$f(x)$</th>
<th>gradient</th>
<th>$x_{new}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.000</td>
<td>9.000</td>
<td>6.000</td>
<td>4.000</td>
</tr>
<tr>
<td>4.000</td>
<td>4.000</td>
<td>4.000</td>
<td>3.333</td>
</tr>
<tr>
<td>3.333</td>
<td>1.778</td>
<td>2.667</td>
<td>2.889</td>
</tr>
<tr>
<td>2.889</td>
<td>0.790</td>
<td>1.778</td>
<td>2.593</td>
</tr>
<tr>
<td>2.593</td>
<td>0.351</td>
<td>1.185</td>
<td>2.395</td>
</tr>
<tr>
<td>2.395</td>
<td>0.156</td>
<td>0.790</td>
<td>2.263</td>
</tr>
<tr>
<td>2.263</td>
<td>0.069</td>
<td>0.527</td>
<td>2.176</td>
</tr>
<tr>
<td>2.176</td>
<td>0.031</td>
<td>0.351</td>
<td>2.117</td>
</tr>
<tr>
<td>2.117</td>
<td>0.014</td>
<td>0.234</td>
<td>2.078</td>
</tr>
<tr>
<td>2.078</td>
<td>0.006</td>
<td>0.156</td>
<td>2.052</td>
</tr>
<tr>
<td>2.052</td>
<td>0.003</td>
<td>0.104</td>
<td>2.035</td>
</tr>
<tr>
<td>2.035</td>
<td>0.001</td>
<td>0.069</td>
<td>2.023</td>
</tr>
<tr>
<td>2.023</td>
<td>0.001</td>
<td>0.046</td>
<td>2.015</td>
</tr>
<tr>
<td>2.015</td>
<td>0.000</td>
<td>0.031</td>
<td>2.010</td>
</tr>
</tbody>
</table>
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

\[ d(x, y) = (x_1 - y_1)^2 + (x_2 - y_2)^2 \]

\[ d(x, y) = (x_1 - y_1)^2 + (3x_2 - 3y_2)^2 \]
Reading

- **Gradient Descent**
  - See e.g. the python example midway down the page

- Previous:
  - Nearest neighbor (*Elements of Statistical Learning* 2.3, 13.3)