CS 440/ECE 448
Margaret Fleck

Lecture Notes (Fall 2021)

- 01 Intro
- 02 Search
- 03 A-star
- 04 Robots (including geometry cheat sheet)
- 05 Probability
- 06 Naive Bayes
- 07 Bayes Nets
- 08 Language
- 09 HMMs
- 10 Vision
- 11 Classifiers
- 12 Linear
- 13 Neural Nets
- 14 Vector Semantics
- 15 Reinforcement Learning
- 16 CSP
- 17 Planning
- 18 Games

CS 440/ECE 448
Margaret Fleck

01 Introduction

Layout of the field

Core reasoning techniques

- discrete/logic-based
- statistical (e.g. Bayesian)
- neural nets
- engineering (e.g. signal processing, 3D geometry, optics, kinematics, ...)

Application areas

- "Core AI" (basic general reasoning)
- Mathematical applications (e.g. theorem proving)
- Computer Vision: describing what's in a digitized picture
- Natural Language/Speech: understanding text or speech input, also generating language that is fluent and keeps track of context.
- Robotics: planning high-level sequences of actions down to mechanical design
- Games (e.g. Chess, Poker)
- Other (computational biology, recognition of music and other non-speech sounds, smart agriculture, ...)

Mathematical applications have largely become their own separate field, separate from AI. The same will probably happen soon to game playing. However, both applications played a large role in early AI.

We're doing to see popular current techniques (e.g. neural nets) but also selected historical techniques that may still be relevant.

Intelligent agents

Viewed in very general terms, the goal of AI is to build intelligent agents. An agent has three basic components:

- sensory inputs (cameras, microphones, touch, kbd input)
- output actions (move left, say "pumpernickel", place call to Alistair)
- goal: what actions are reasonable given specific inputs

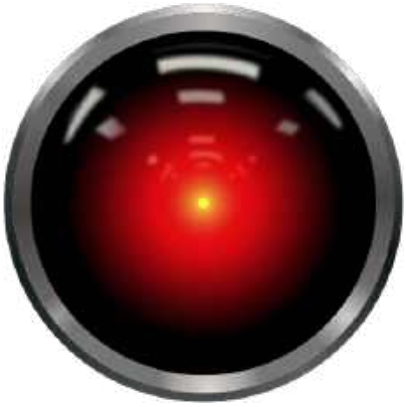
Some agents are extremely simple, e.g. a Roomba:


Roomba (from IRobot)

We often imagine intelligent agents that have a sophisticated physical presence, like a human or animal. Current robots are starting to move like people/animals but have only modest high-level planning ability.


Boston dynamics robot (video)

Other agents may live mostly or entirely inside the computer, communicating via text or simple media (e.g. voice commands). This would be true for a chess playing program or an intelligent assistant to book airplane tickets. A well-known fictional example is the HAL 9000 system from the movie "2001: A Space Odyssey" which appears mostly as a voice plus a camera lens (below).


(from Wikipedia) (dialog) (video)

When designing an agent, we need to consider

- what environment is it intended to operate in?
- exactly what inputs and actions does it have?
- what do we mean by good/bad performance and how do we quantify it?

Many AI systems work well only because they operate in limited environments. A famous sheep-shearingn robot from the 1980's depended on the sheep being tied down to limit their motion. Face identification systems at security checkpoints may depend on the fact that people consistently look straight forwards when herded through the gates.

Simple agents like the Roomba may have very direct connections between sensing and action, with very fast response and almost nothing that could qualify as intelligence. Smarter agents may be able to spend a lot of time thinking, e.g. as in playing chess. They may maintain an explicit model of the world and the set of alternative actions that it is choosing between.

AI researchers often plan to build a fully-featured end-to-end system, e.g. a robot child. However, most research projects and published papers consider only one small component of the task, e.g.

- identify the objects in a picture
- transcribe this audio clip into written English
- plan motion for robot arm to fit two parts together

History: major trends

Over the decades, AI algorithms have gradually improved. Two main reasons:

- Improvements in scientific theories

- Increase in computing power and amount of training data.

The second has been the most important. Some approaches popular today were originally proposed in the 1940's but could not be made to work without sufficient computing power.

AI has cycled between two general approaches to algorithm design:

- "Smart" models, discrete and logic-based.
- "Blind" statistical algorithms

Most likely neither extreme position is correct.

AI results have consistently been overly hyped, sometimes creating major tension with grant agencies.

History: details

Early 20th century

No clear notion of a computer (though interesting philosophy)

1930's and 1940's

First computers (Atanasoff 1941, Eniac 1943-44), able to do almost nothing.

ENIAC, mid 1940's (from Wikipedia)

On-paper models that are insightful but can't be implemented. Some of these foreshadow approaches that work now.

- Alan Turing 1930's and 1940's
- Zellig Harris 1940's owards
- McCulloch and Pitts (early neural nets)

1950-1970's

Computers with functioning CPU but (by today's standards) very slow with absurdly little memory. E.g. the IBM 1130 (1965-72) came with up to 11M of main memory. The VAX 11/780 had many terminals sharing one CPU.

 IBM 1130, late 1960's (from Wikipedia)

Graphis mostly used pen plotters, or perhaps graphics paper that had to be stored in a fridge. Mailers avoided linux gateways because they were unreliable. Prototypes of great tools like the mouse, GUI, Unix, refresh-screen graphics.

AI algorithms are very stripped down and discrete/symbolic. Chomsky's Syntactic Structures (1957) proposed very tight constraints on linguistic models in an attempt to make them learnable from tiny datasets.

1980's-1990's

Computers now look like tower PC's, eventually also laptops. Memory and disk space are still tight. Horrible peripherals like binary (black/white) screens and monochrome surveillance cameras. HTTP servers and internet browsers appear in the 1990's:

 Lisp Machine, 1980's (from Wikipedia

NCSA Mosaic, 1990's (from Wikipedia)

AI starts to use more big-data algorithms, especially at sites with strong resources. First LDC (linguistic data consortium) datasets: 1993 Switchboard 1, TIMIT. Fred Jelinek starts to get speech recognition working usefully using statistical ngram models. Linguistic theories were still very discrete/logic based. Jelinek famously said in the mid/late 1980's: "Every time I fire a linguist, the performance of our speech recognition system goes up."

This century

Modern computers and internet.

Algorithms achieve strong performance by gobbling vast quantities of training data, e.g. Google's algorithms use the entire web to simulate a toddler. Neural nets "learn to the test," i.e. learn what's in the dataset but not always with constraints that we assume implicitly as humans. So strange unexpected lossage ("adversarial examples"). Speculation that learning might need to be guided by some structural constraints.

What still doesn't work


("Three men drinking tea" by a Microsoft AI program, from New Scientist, 2019)

(from CNN, 2019)


Boston Dynamics robot falling down (from The Guardian, 2017)

CS 440/ECE 448
Margaret Fleck

02 Uninformed Search

Many problems in AI can be represented as state graphs. So graph search apprears frequently as a component of AI algorithms. Because AI graphs tend to be very large, efficient design of search algorithms is often critical for good performance.

Since this class depends on data structures, which has discrete math as a prerequisite, most people have probably seen some version of BFS, DFS, and state graphs before. If this isn't true for you, this lecture probably went too fast. Aside from the textbook, you may wish to browse web pages for CS 173, CS 225, and/or web tutorials on BFS/DFS.

State graph representations

Key parts of a state graph:

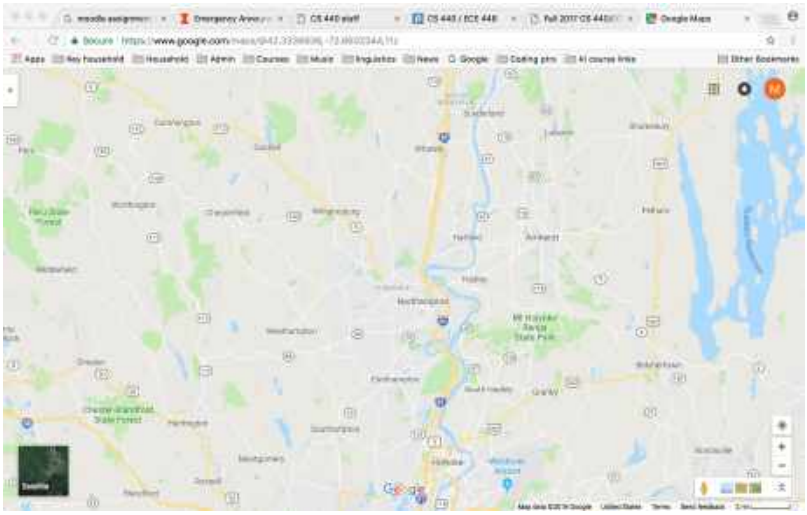
- states (graph nodes)
- actions (graph edges, with costs)
- start state
- goal states (explicit list, or a goal condition to test)

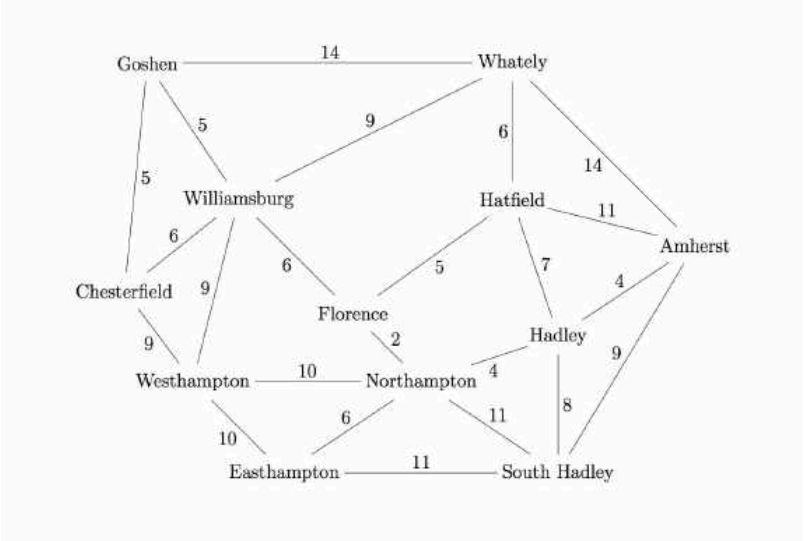
Task: find a low-cost path from start state to a goal state.

Some applications want a minimum-cost path. Others may be ok with a path whose cost isn't stupidly bad (e.g. no loops).

Road maps

We can take a real-world map (below left) and turn it into a graph (below right). Notice that the graph is not even close to being a tree. So search algorithms have to actively prevent looping, e.g. by remember which locations they've already visited.





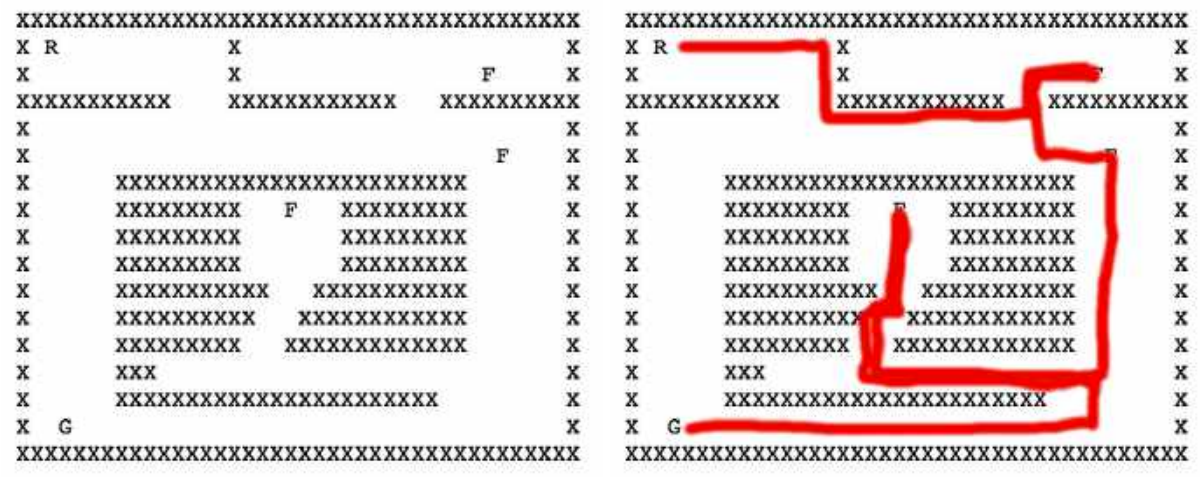
Mapping this onto a state graph:

- states are towns
- actions are going down a road
- edge costs are distances

On the graph, it's easy to see that there are many paths from Northampton to Amherst, e.g. 8 miles via Hadley, 31 miles via Williamsburg and Whately.

Mazes

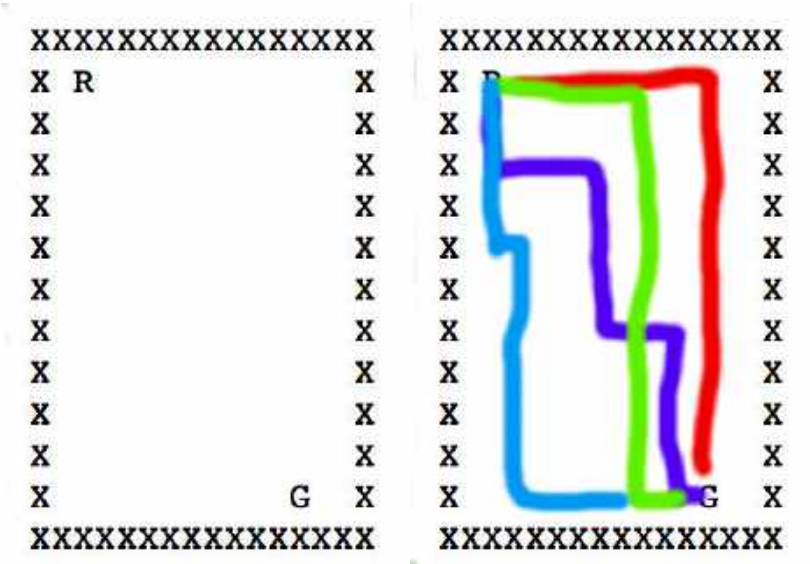
Left below is an ASCII maze of the sort used in 1980's computer games. R is the robot, G is the goal, F is a food pellet to collect on the way to the goal. We no longer play games of this style, but you can still find search problems with this kind of simple structure. On the right is one path to the goal. Notice that it has to double back on itself in two places.



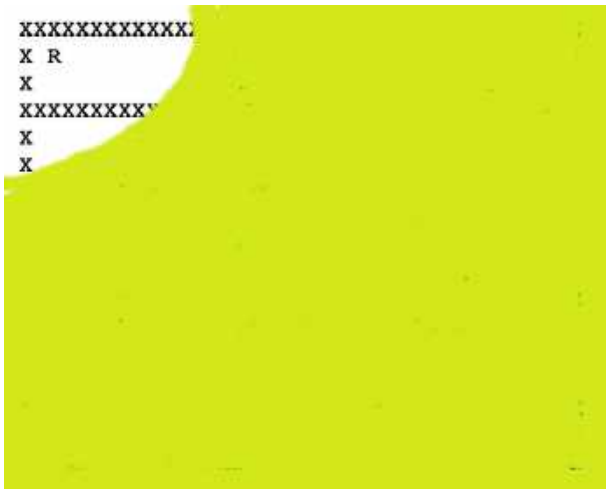
Modelling this as a state graph:

- states are (x,y) positions
- actions are moves left, right, up, down
- edge costs are constant

Some mazes impose tight constraints on where the robot can go, e.g. the bottom corridor in the above maze is only one cell wide. Mazes with large open areas can have vast numbers of possible solutions. For example, the maze below has many paths from start to goal. In this case, the robot needs to move 10 steps right and 10 steps down. Since there are no walls constraining the order of right vs. downward moves, there are $\binom{20}{10}$ paths of shortest length (20 steps), which is about 185,000 paths. We need to quickly choose one of these paths rather than wasting time exploring all of them individually.



Game mazes also illustrate another feature that appears in some AI problems: the AI gains access to the state graph as it explores. E.g. at the start of the game, the map may actually look like this:



Puzzle

States may also be descriptins of the real world in terms of feature values. For example, consider the Missionaries and Cannibals puzzle. In the starting state, there are three missionaries, three cannibals, and a boat on the left side of a river. We need to move all six people to the right side, subject to the following constraints:

- The boat cannot carry more than two people at a time.
- The boat cannot move without at least one person on it.
- If there are both missionaries and cannibals on one side of the river, the cannibals cannot outnumber the missionaries. (Otherwise they will eat the missionaries.)

The state of the world can be described by three variables: how many missionaries on the left bank, how many cannibals on the left bank, and which bank the boat is on. The state space is shown below:

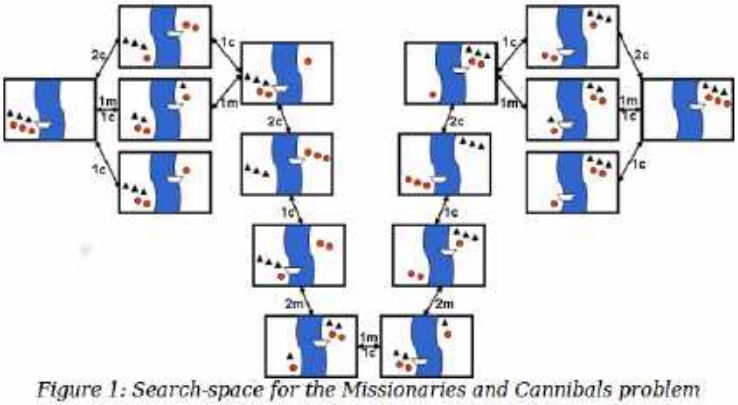


Figure 1: Search-space for the Missionaries and Cannibals problem (from Gerhard Wickler, U. Edinburgh)

In this example

- states are sets of variable values
- actions are legal changes to these values
- edge costs often constant

Speech recognition

In speech recognition, we need to transcribe an acoustic waveform into to produce written text. We'll see details of this process later in the course. For now, just notice that this proceeds word-by-word. So we have a set of candidate transcriptions for the first part of the waveform and wish to extend those by one more word. For

example, one candidate transcription might start with "The book was very" and the acoustic matcher says that the next word sounds like "short" or "sort."

In this example

- states are partial transcriptions of the input
- actions extend the transcription (e.g. by one word)
- edge costs reflect how well the new word matches the acoustic signal and whether the new word seems appropriate given the previous context.

Speech recognition systems have vast state spaces. A recognition dictionary may know about 100,000 different words. Because acoustic matching is still somewhat error-prone, there may be many words that seem to match the next section of the input. So the currently-relevant states are constructed on demand, as we search. Chess is another example of an AI problem with a vast state space.

Some high-level points

For all but the simplest AI problems, it's easy for a search algorithm to get lost.

- The goal is not present, or not reachable.
- The graph has cycles.
- The state space is very large.
- There are many paths of similar quality.

The first two of these can send the program into an infinite loop. The third and fourth can cause it to get lost, exploring states very far from a reasonable path from the start to the goal.

Recap

Our first few search algorithms will be **uninformed**. That is, they rely only on computed distances from the starting state. Next week, we'll see A^*. It's an **informed** search algorithm because it also uses an estimate of the distance remaining to reach the goal.

Basic search outline

A state is exactly one of

- not yet seen (can be a very large set in some applications)
- frontier, i.e. seen but we haven't explored its outgoing edges
- completely done (all outgoing edges followed)

Basic outline for search code

> Loop until we find a solution
>
> - take a state off frontier
> - follow outgoing edges to find neighbors
> - add neighbors to frontier if not already seen

Data structure for frontier determines type of search

- BFS: queue
- DFS: stack (or implicit stack via recursion)

- UCS (uniform cost search) and A^*: priority queue

BFS example

Let's find a route from Easthampton to Whately Using BFS. The actual best route is via Northampton, Florence, Hatfield (19 miles)

The frontier is stored as a queue. So we explore locations in rings, gradually moving outwards from the starting location. Let's assume we always explore edges counterclockwise (right to left)

Start
    Easthampton
Add neighbors of Easthampton
    South Hadley, Northampton, Westhampton
Add neighbors of South Hadley
    Amherst, Hadley [Northampton is already seen]
Add neighbors of Northampton
    Florence, Westhampton
Add neighbors of Westhampton
    Williamsburg, Chesterfield
Add neighbors of Amherst
    ... FOUND WHATELY

We return the path Easthampton, South Hadley, Amherst, Whately (length 34). This isn't the shortest path by mileage, but it's one of the shortest paths by number of edges.

BFS properties:

- Solution is always optimal if and only if all edges have same cost.
- Frontier can get very large, often grows linearly with distance outwars from start.

Here is a fun animation of BFS set to music.

DFS example

The frontier is stored as a stack (explicitly or implicitly via recursive function calls). Again, we'll explore edges counterclockwise. The frontier looks like:

Start
    Easthampton
Add neighbors of Easthampton
    South Hadley, Northampton, Westhampton
Add neighbors of Westhampton
    Williamsburg, Chesterfield
Add neighbors of Chesterfield
    Goshen
Add neighbors of Goshen
    ... FOUND WHATELY

We return the path: Westhampton, Chesterfield, Goshen, Whately (length 38)

DFS properties:

- Solution can be very far from optimal.

- Frontier stays small.
- Not guaranteed to find the goal even if it's reachable

Animation of DFS set to music. Compare this to the one for BFS.

Basic tricks

There are several basic tricks required to make DFS and BFS work right. First, store the list of seen states, so that your code can tell when it has returned to a state that has already been explored. Otherwise, your code may end up in an infinite loop and/or be massively inefficient.

Second, return as soon as a goal state has been found. The entire space of states may be infinite. Even when it is finite, searching the entire set of states is inefficient.

Third, don't use recursion to implement DFS unless paths are known to be short. A small number of programming languages make this safe by optimizing tail recursion. More commonly, each recursive call allocates stack space and you are likely to exceed limits on stack size and/or recursion depth.

Returning paths

When a search algorithm finds the goal, it must be able to return best path to the goal. So, during search, each seen state must remember the best path back to the start state. To do this efficiently, each state stores a pointer to the previous state in the best path. When the goal is found, we retrieve the best path by chaining backwards until we reach the start state.

Length-bounded DFS

DFS performs very poorly when applied directly to most search problems. However, it becomes the method of choice when we have a tight bound on the length of the solution path. This happens in two types of situations, which we'll see later in the course:

- The search space has a small natural depth (e.g. constraint-based problems).
- Our system resources impose a limit on how far away we can search (e.g. complex games or large robot environments).

In this situation, DFS can be implemented so as to use very little memory. In AI problems, the set of seen states can often become very large, so that programs can run out of memory or put enough pressure on the system memory management that programs slow down. Instead of storing and checking this large set, length-bounded DFS checks only that the current path contains no duplicate states. There is a tradeoff here. Checking only the current path will cause the algorithm to re-do some of its search work, because it can't remember what states has already searched. In practice, the wisdom of this approach depends on the application. When there are tight bounds on path length, it's safe to implement DFS using recursion rather than a stack.

Iterative deepening

We can dramatically improve the low-memory version of DFS by running it with a progressively increasing bound on the maximum path length. This is a bit like having a dog running around you on a leash, and gradually letting the leash out. Specifically:

    For k = 1 to infinity

Run DFS but stop whenever a path contains at least k states.
Halt when we find a goal state

This is called "iterative deepening." Its properties

- Overall search behavior looks like BFS.
- Each iteration starts from scratch, forgetting all previous work.
- Uses very little memory.

Each iteration repeats the work of the previous one. So iterative deepening is most useful when the new search area at distance k is large compared to the area search in previous iterations, so that the search at distance k is dominated by the paths of length k.

These conditions don't hold for 2D mazes. The search area at radius k is proportional to k, whereas the area inside that radius is proportional to k^2.

However, suppose that the search space looks like a 4-ary tree. We know from discrete math that there are 4^k nodes at level k. Using a geometric series, we find that there are $1/3 \cdot (4^k - 1)$ nodes in levels 1 through k-1. So the nodes in the lowest level dominate the search. This type of situation holds (approximately) for big games such as chess.

Backwards and Bidirectional Search

Backwards and bidirectional search are basically what they sound like. Backwards search is like forwards search except that you start from the goal. In bidirectional search, you run two parallel searches, starting from start and goal states, in hopes that they will connect at a shared state. These approaches are useful only in very specific situations.

Backwards search makes sense when you have relatively few goal states, known at the start. This often fails, e.g. chess has a huge number of winning board configurations. Also, actions must be easy to run backwards (often true).

Example: James Bond is driving his Lexus in central Boston and wants to get to Dr. Evil's lair in Hawley, MA (population 337). There's a vast number of ways to get out of Boston but only one road of significant size going through Hawley. In that situation, working backwards from the goal might be much better than working forwards from the starting state.

Uniform-cost search

Breadth-first search only returns an optimal path if all edges have the same cost (e.g. length). Uniform-cost search (UCS) is a similar algorithm that finds optimal paths even when edge costs vary.

Recall that the **frontier** in a search algorithm contains states that have been seen but their outgoing edges have not yet been explored. BFS stores the frontier as a queue; DFS stores it as a stack. Uniform-cost search stores the frontier as a priority queue. So, in each iteration of our loop, we explore neighbors of the best state (i.e. lowest cost, shortest path) seen so far.

From now on, we will assume that

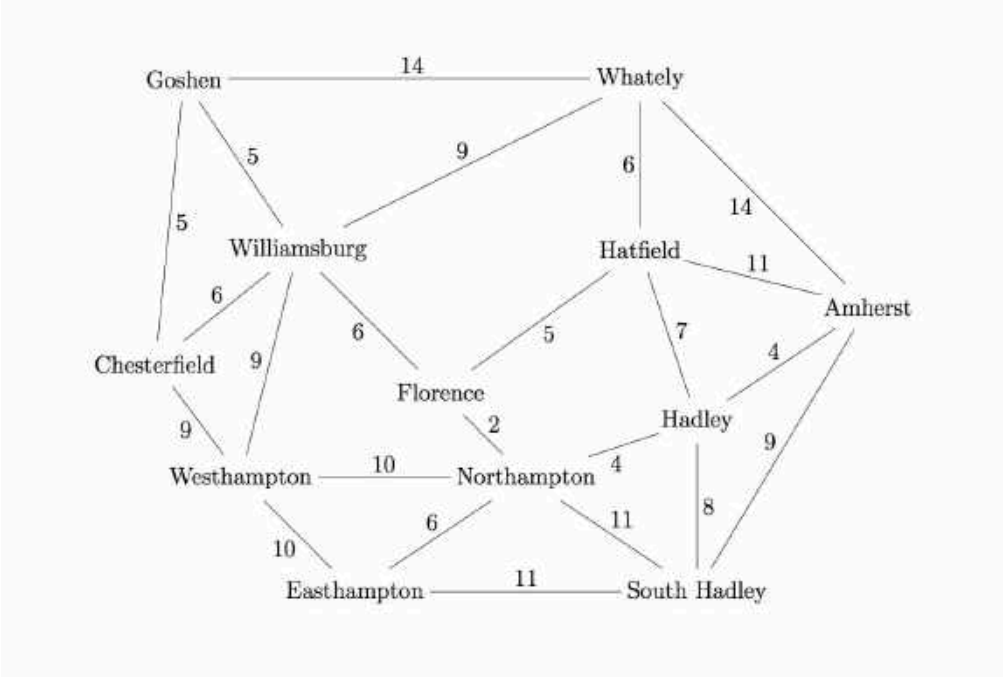
- edge costs are always positive, and
- all edge costs are >= b, where b is some minimum non-zero cost.

These assumptions are almost always true. If they aren't true, algorithm design and analysis can get much harder in ways that are beyond the scope of this class.

The details

When we're using BFS on a problem where all edges have the same cost, we can safely return as soon as we see a goal state. There's no need to put the goal state back into the frontier for further processing. However, consider the following example:

> Suppose we are going from Westhampton to Florence in the road graph below. When we look at Westhampton's neighbors, Williamsburg is closer than Northampton. So we'll first get to Florence via Williamsburg, a route that takes 15 miles. However, the best route is 12 miles, via Northampton.



Evidently, this method of finishing up our search doesn't guarantee an optimal path when edge costs vary. So we need to make a couple changes to ensure that we have really found the best path when we halt and declare success:

- When exploring a state, neighbors are put into the frontier even if they are goal states.
- We stop when goal state reaches the top of the priority queue, **not** when it is first seen.
- When we re-discover a state (i.e. via a different path), we update our record of its path length.

Also notice that our representation of a state should include the cost to reach it, along the best path seen so far. Computing this dynamically would be very slow.

Longer Example

As an example, consider going from Easthampton to Hatfield in our road map.

As with any of these search algorithms, our priority queue starts off containing Easthampton and we then remove Easthampton from the frontier and add its neighbors. However, since we're doing UCS, we add the neighbors in order of distance, not some arbitrary search ordering (e.g. left to right). Our search (states with distances) now looks like this:

```
DONE
    Easthampton 0

FRONTIER
    Northampton 6
    Westhampton 10
    South Hadley 11
```

Now we add Northampton's neighbors:

```
DONE
    Easthampton 0
    Northampton 6

FRONTIER
    Florence 8
    Westhampton 10  <-- Not clear if it's before or after Hadley
    Hadley 10
    South Hadley 11
```

Now we add Florence's neighbors:

```
DONE
    Easthampton 0
    Northampton 6
    Florence 8

FRONTIER
    Westhampton 10
    Hadley 10
    South Hadley 11
    Hatfield 13    <-- GOAL!  But don't stop yet
    Williamsburg 14
```

We have a path to Hatfield, but we can't yet be sure that it's the best one. So we keep going, exploring routes outwards from Westhampton, Hadley, and South Hadley.

```
DONE
    Easthampton 0
    Northampton 6
    Florence 8
    Westhampton 10
    Hadley 10
    South Hadley 11

FRONTIER
    Hatfield 13  <-- GOAL
    Williamsburg 14
    Amherst 14
    Chesterfield 19
```

Now our goal is at the top of the priority queue, so there can't be a better path. So we return the path: Easthampton, Northampton, Florence, Hatfield.

Final comments on UCS

UCS returns an optimal path even if edge costs vary. Memory requirements for UCS are similar to BFS. They can be worse than BFS in some cases, because UCS can get stuck in extended areas of short edges rather than exploring longer but more promising edges.

You may have seen Dijkstra's algorithm in data structures class. It is basically similar to UCS, except that it finds the distance from the start to all states rather than stopping when it reaches a designated goal state.

Dijkstra animation

Edit distance

A good search algorithm must be coupled with a good representation of the problem. So transforming to a different state graph model may dramatically improve performance. We'll illustrate this with finding edit distance, and then see another example when we get to classical planning.

Problem: find the best alignment between two strings, e.g. "ALGORITHM" and "ALTRUISTIC". By "best," we mean the alignment that requires the fewest editing operations (delete char, add char, substitute char) to transform the first word into the second. We can get from ALGORITHM to ALTRUISTIC with six edits, producing the following optimal alignment:

```
 A L G O R   I   T H M
 A L   T R U I S T I C
     D S   A   A   S S

D = delete character
A = add character
S = substitute
```

Versions of this task appear in

- theory classes (example of dynamic programming)
- computational biology (matching DNA sequences)
- evaluating speech recognizers (comparing machine transcription to correct one)
- machine translation (aligning text in two different languages to match related words)

Naive method

Each state is a string. So the start state would be "algorithm" and the end state is "altruistic." Each action applies one edit operation, anywhere in the word.

In this model, each state has a very large number of successors. So there are about 28^9 single-character changes you could make to ALGORITHM, resulting in words like:

```
LGORITHM, AGORITHM, ..., ALGORIHM, ...
BLGORITHM, CLGORITHM, ....
AAGORITHM, ....
...
ALBORJTHM, ...
....
```

This massive fan-out will cause search to be very slow.

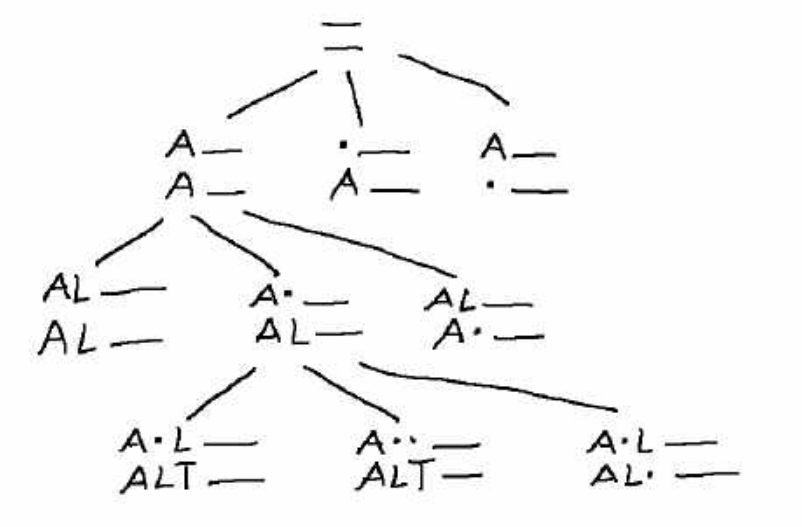
Better method

A better representation for this problem explores the space of partial matches. Each state is an alignment of the initial sections (prefixes) of the two strings. So the start state is an alignment of two empty strings, and the end state is an alignment of the full strings.

In this representation, each action extends the alignment by one character. From each state, there are only three legal moves:

- pair the next characters in the two strings
- skip a character in the first string
- skip a character in the second string

Here's part of the search tree, showing how we extend the alignment with each step. (A dot represents a skipped character.)



In this specific case, we can pack our work into a nifty compact 2D table.

This algorithm can be adapted for situations where the search space is too large to maintain the full alignment table. for example, we may need to match DNA sequences that are 10^5 characters long.

- We can an approximate search in which we keep only the best k alignments of each length (e.g. speech recognition).
- We can compute the edit distance but not the optimal alignment, which requires storing only two columns from the table.

For more detail, see Wikipedia on edit distance and Jeff Erickson's notes.

CS 440/ECE 448
Margaret Fleck

03 A* Search

Shakey the robot (1966-72), from Wired Magazine

*A** search

*A** search was developed around 1968 as part of the Shakey project at Stanford. It's still a critical component of many AI system.s The Shakey project ran from 1966 to 1972. Here's some videos showing Shakey in action.

- Shakey (video from the time)
- Shakey (retrospective video)
- Remembering Shakey

Towards the end of the project, it was using a PDP-10 with about 800K bytes of RAM. The programs occupied 1.35M of disk space. This kind of memory starvation was typical of the time. For example, the guidance computer for Apollo 11 (first moon landing) in 1969 had only 70K bytes of storage.

The Problem

Cost so far is a bad predictor of how quickly a route will reach a goal. If you play around with this search technique demo by Xueqiao Xu, you can see that BFS and UCS ("Dijkstra") search a widening circular space, most of which is in the direction opposite to the goal. When they assess states, they consider only distance from the starting position, but not the distance to reach the goal.

Recall UCS

Recall that uniform-cost search(UCS) has three types of states:

- unseen
- done/closed = neighbors all explored, allegedly finished with this state
- open/frontier = seen but neighbors need to be explored

The frontier is stored in a priority queue, with states ordered by their distance g(s) from the start state. At each step of the search, we extract the state S with lowest g value. For each of S's neighbors, we check if it has already been seen. If it's new, add it to the frontier.

Whe we return to a previously-seen state S via a new, shorter path, we update the stored distance g(S) and S's position in the priority queue.

Search stops when the goal reaches the top of the queue (not when the goal is first found).

A^* search

The idea behind A^* search is use a better estimate of how promising each state is. Specifically, we sum

- cost so far, g(x), and
- estimated ("heuristic") distance to goal, h(x).

So A^* search is essentially the UCS algorithm but prioritizing states based on f(x) = g(x) + h(x) rather than just g(x).

With a reasonable choice of heuristic, A^* does a better job of directing its attention towards the goal. Try the search demo again, but select A^*.

What makes a heuristic "reasonable"?

Heuristic functions for A* need to have two properties

- quick to compute
- underestimates of the true distance to the goal.

In the context of A^* search, being an underestimate is also known as being "admissible."

Heuristics are usually created by simplifying ("relaxing") the task, i.e. removing some constraints. For search in mazes/maps, we can use straight line distance, so ignoring

- obstacles (e.g. in a maze), and
- the fact that roads may be missing or not straight or have different speed limits (e.g. in a graph-like road map).

All other things being equal, an admissible heuristic will work better if it is closer to the true cost, i.e. larger. A heuristic function h is said to "dominate" a heuristic function with lower values.

For example, suppose that we're searching a digitized maze and the robot can only move in left-right or up-down (not diagonally). Then Manhattan distance is a better heuristic than straight line distance. The Manhattan distance between two points is the difference in their x coordinates plus the difference in their y coordinates. If the robot is allowed to move diagonally, we can't use Manhattan distance because it can overestimate the distance to the goal.

Another classical example for A^* is the 15-puzzle. A smaller variant of this is the 8-puzzle..

Two simple heuristics for the 15-puzzle are

- Number of misplaced tiles
- Sum of the Manhattan distances between each tile's current location and goal location.

See 8/15-puzzle demo by Julien Dramaix

Why Admissibility?

Admissibility guarantees that the output solution is optimal. Here's a sketch of the proof:

> Search stops when goal state becomes top option in frontier (not when goal is first discovered). Suppose we have found goal with path P and cost C. Consider a partial path X in the frontier. Its estimated cost must be $\geq C$. Since our heuristic is admissible, the true cost of extending X to reach the goal must also be $\geq C$. So extending X can't give us a better path than P.

Major variants on A*

A^* decays gracefully if admissibility is relaxed. If the heuristic only slightly overestimates the distance, then the return path should be close to optimal. This is called "suboptimal" search. Non-admissible heuristics are particularly useful when there are many alternate paths of similar (or even the same) cost, as in a very open maze. The heuristic is adjusted slightly to create a small preference for certain of these similar paths, so that search focuses on extending the preferred paths all the way out to the goal. See Amit Patel's notes for details.

In some search problems, notably speech recognition, the frontier can get extremely large. "Beam search" algorithms prune states with high values of f. Like suboptimal search, this sabotages guarantees of optimality. However, in practice, it may be extremely unlikely that these poorly-rated states can be extended to a good solution.

Recap of A*

States are evaluated by a function f which estimates from start to goal via this state.

> f(s) = g(s) + h(s)

g(s) is (actual) distance from the start state to s. h(s) is a heuristic estimate of the distance from s to the goal. f(s) is an estimate of the length of a path from start to goal via s.

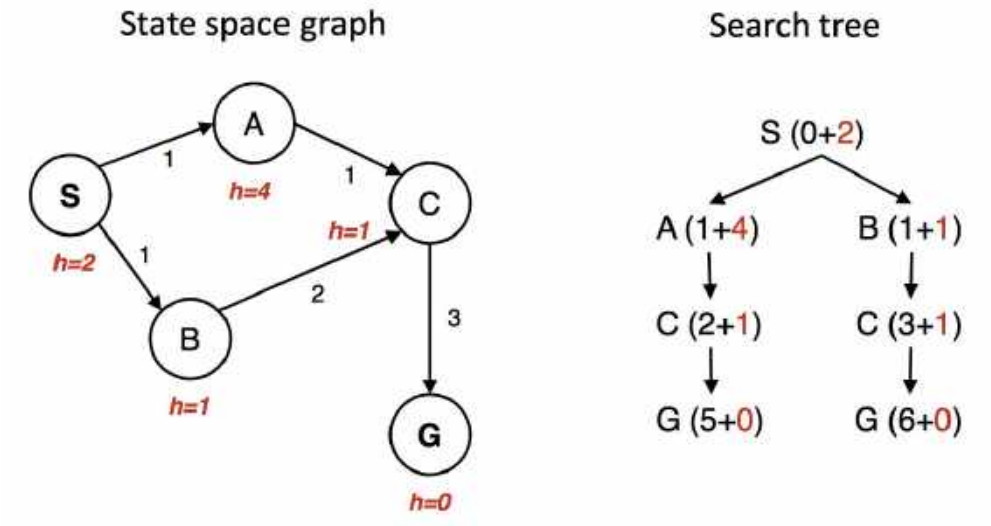
Three types of states:

- unseen
- done/closed = neighbors all explored, allegedly finished with this state
- open/frontier = seen but neighbors need to be explored

The frontier is stored in a priority queue, organized by the f values.

Correct bookkeeping isn't obvious

*A** finds the optimal path only when you do the bookkeeping correctly. The details are somewhat like UCS, but there are some new issues. Consider the following example (artificially cooked up for the purpoose of illustration).



(from Berkeley CS 188x materials)

The state graph is shown on the left, with the h value under each state. The right shows a quick picture of the search process. Walking through the search in more detail:

> First we put S in the queue with estimated cost 2, i.e. g(S) = 0 plus h(S) = 2.
>
> We remove S from the queue and add its neighbors. A has estimated cost 5 and B has estimated cost 2. S is now done/closed.
>
> We remove B from the queue and add its neighbor C. C has estimated cost 4. B is now done/closed.
>
> We remove C from the queue. One of its neighbors (A) is already in the queue. So we only need to add G. G has cost 6. C is now done/closed.
>
> We remove A from the queue. Its neighbors are all done, so we have nothing to do. A is now done.
>
> We remove G from the queue. It's the goal, so we halt, reporting that the best path is SBCG with cost 6.

Oops. The best path is through A and has actual cost 5. But heuristic was admissible. What went wrong? Obviously we didn't do our bookkeeping correctly.

Fixing our bookkeeping

The algorithm we inherited from from UCS only updates costs for nodes that are in the frontier. In this example, paths through B look better at the start. So SBCG reaches C and G first. C and G are marked done/closed. And

then we see the shorter path SAC. Updating the cost of C won't help because it's in the done/closed set.

> Fix: when we update the cost of a node in the closed/done set, put that node back into the frontier (priority queue).

There is also a second issue, that applies also to UCS. Efficient priority queue implementations (e.g. the ones supplied with your favorite programming language) don't necessarily support updates to the priority numbers.

> Workaround: it may be more efficient to put a second copy of the state (with the new cost) into the priority queue.

In many applications, decreases in state cost aren't very frequent. So this creates less inefficiency than using a priority queue implementation with more features.

Consistency

The problem with shorter paths to old states is simplified if our heuristic is "consistent." That is, if we can get from n to n' with cost c(n,n'), we must have

$$h(s) \leq c(s, s') + h(s')$$

So as we follow a path away from start state, our estimated cost f never decreases. Useful heuristics are often consistent. For example, the straight-line distance is consistent because of the triangle inequality.

When the heuristic is consistent, we never have to update the path length for a node that is in the done/closed set. (Though we may still have to update path lengths for nodes in the frontier.)

We can view UCS as A* with a heuristic function h that always returns zero. This heuristic is consistent, so updates to costs in UCS involve only states in the frontier.

Other hacks and variants

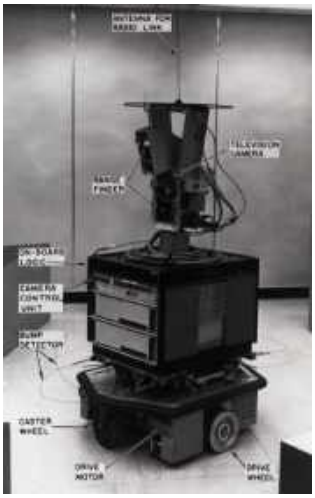
There are many variants of A*, mostly hacks that may help for some problems and not others. See this set of pages for extensive implementation detail Amit Patel's notes on A*.

One well-known variant is iterative deepening A*. Remember that iterative deepening is based on a version of DFS ("length-bounded") in which search is stopped when the path reaches a target length k. Iterative deepening A* is similar, but search stops if the A* estimated cost f(s) is larger than k.

Another optimization involves building a "pattern database." This is a database of partial problems, together with the cost to solve the partial problem. The true solution cost for a search state must be at least as high as the cost for any pattern it contains.

CS 440/ECE 448
Margaret Fleck

04 Robot planning



(from Wikipedia)

Parts of a robot

Even though Shakey was built 50 years ago, it still had the basic components of a mobile robot. That is:

- battery (or power cord) and motors
- brain (aka computer)
  - on-board
  - off-board via radio link or control cable
- propulsion (wheels)
- long-distance sensors
  - passive, e.g. cameras
  - active e.g. IR, sonar
- touch sensors (bumpers, whiskers)

Newer robots may also have features like

- more types of sensors
  - special-purpose (e.g. has cutter penetrated skin)
  - GPS
- legs
- arms, grippers (with force feedback and "skin")

Batteries are a surprisingly critical part of the design. Even now, they are heavy compared to how much power they deliver. So they can be a significant percentage of the robot's weight.

A robot usually has several redundant sensors because they fail in different ways, e.g. whiskers operate only at very short distances, IR sensors fail on dark surfaces.

There is a tradeoff between extra features and better safety. Robots working close to humans have to be less ambitious so that they are more reliably safe.

Mobile robots

How does it move?

- wheels
- legs
- propellers (flying, underwater)
- rolling
- snake-like motion
- floats in space (e.g. on the International Space Station)

Food delivery robot from Techcrunch May 26, 2018

Waymo (Google) self-driving car

legged robot from ETH Zurich

Cimon floating on the ISS from NBC

Guard robot from The Telegraph

Snakebot from NASA

Cool videos

Robots generate the best videos in the field.

CMU snake robot climbs a tree

Guardbot

Legged robots (Marc Raibert and Boston Dynamics): early ones and newish humanoid one

Google self-driving bike (fake)

Adeept robot arm for Arduino (from Amazon)

Robot arms

Robots can also be jointed arms, mounted on a fixed base or a mobile one. This were originally used primarily for industrial assembly, due to safety concerns. As they have become smaller and more reliable, we also see human-facing applications.

Highlights of the Hannover MESSE 2017

Parts assembly: Allied-Technology.com Tabletop Demo of Small Parts Assembly

Some robots hang from the ceiling, e.g. this chef robot from Samsung:

 from the Robb Report

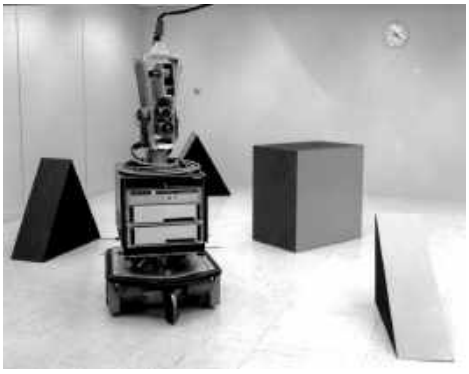
Samsung chef robot video at CES 2020

Planning motions of for jointed arms is made complex by the need to translate between real-world coordinates and joint angles. We'll see the details soon.

Also, very accurate positioning requires that links remain stiff, which ends up requiring very heavy robots for absurdly small payloads. E.g. a very accurate person-sized robot may have only a 10lb payload. Both high-level and low-level planning are more complicated when links may flex and/or there is significant error in positioning.

The environment

Very early robots such as Shakey moved around a world that was largely static and easy to figure out from camera images. Objects moved only when the robot made them move. (Any grad student helpers would be sneaking around behind its back.)

 (from New Atlas)

As AI systems have gotten more capable, they have moved into more relaistic environments:

- The environment must be learned gradually via imperfect sensors.
- The environment may change over time.
- There may be other agents (robots or people)
    - working with the robot, or
    - working against the robot.
- The obstacles may be
    - things you can bump into,
    - areas that are too steep to drive on (e.g. the side of a tree), or
    - surfaces that your wheels/legs can't handle (e.g. gravel).

Final video

Video of robots working as a team: U. Penn drones play James Bond

Path Planning

Planning motions for robots isn't too difficult if the robot is very tiny, so we can approximate it as a point. However, that's usually not the case. So a key problem in robotics is

> Where can the robot move when the robot has significant size?

We'll model robot and obstacles as simple geometrical shapes to develop the ideas.

Moving a circular robot

Suppose that we're moving a circular robot in 2D. We can simplify planning using the following idea:

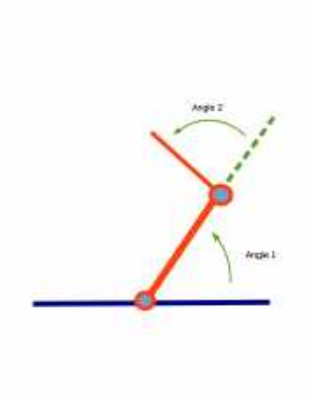
> Idea: expand obstacles so we can shrink robot to a point.

This modified space is called a configuration space. It shows all the possible positions for the center of the robot.

Obstacles for a circular robot (from Howie Choset)

Motion of robot arms

Another standard simple example is a robot arm with two links connected by rotating joints, like this:



To get a sense of how the arm can move around obstacles, play around with this 2-link robot simulator by Jeffrey Ichnowski. Or look at these pictures of a physical robot model .

Our high-level planning task is expressed in normal 2D coordinates, e.g. move the tip from one (x,y) location to another. However, we control the robot via the angles of the joints. These (α, β) pairs form a different 2D space, called the configuration space of the problem:

> Big idea: do planning in configuration space (Lozano-Perez around 1980).

Configuration space for a 2-link arm (from Howie Choset)

Notice that a goal in real space might correspond to more than one point in configuration space. For 2-link arm, many 2D points can be reached with the elbow pointing in two ways. For arms with more links, we can find multiple solutions to goals that specify the angle, as well as the position, of the hand. Animals can exploit this to maintain several constraints simultaneously, e.g. keep head in a fixed position while feet hold onto a moving branch.

Kingfisher video from the Audobon Society

Useful references

The figures above credited to Howie Choset came from Howie Choset, CMU 16-311, Spring 2018 and 16-735, Fall 2010. For more details, check out his book **Principles of Robot Motion**.

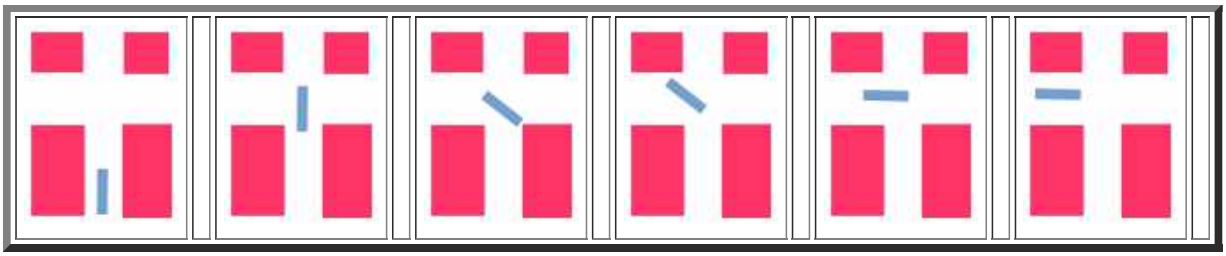
Fun book on collision detection by Jeff Thompson

A 2D scene

 (from Google maps)

Motion of rotating polygon (car)

Now consider a polygonal robot moving around in 2D. For example, a car or truck might be moving around the roads near the Siebel Center shown in the areal view above. In the simplified map below , we have two crossing roads defined by four obstacles (red). The car (blue) can move forwards and rotate.



To specify the robot's position, we need to give its (x,y) position and also its orientation. So the robot's configuration space is three dimensional.

See the following slides for two worked examples from Howie Choset:

- Configuration space obstacle for a rectangular object with a moving rectangular robot
- Moving an L-shaped object through some tight doorways

Issues

So this 2D problem with simple polygonal obstacles has a 3D configuration space with quite complex obstacles.

The dimensionality of the configuration space gets even larger in many real-world situations. For example, if we

- move objects in 3D (e.g. drones, assembly of 3D parts)
- robots with more degrees of freedom, e.g. arm with more than one joint, arm mounted on mobile base

Searching configuration space

How do we compute a path in a configuration space?

For simple 2D examples, e.g. MP 2, we can digitize the configuration space positions and use a digitized maze search algorithm. However, this approach scales very badly.

- Digitized size increases exponentially in the number of dimensions
- Free space tends to be mostly large open areas, where many paths are possible
- Precision needs are variable: low in free space, high near obstacles

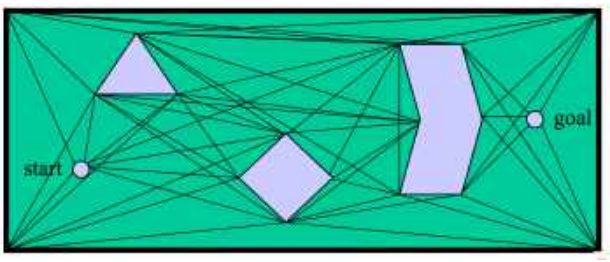
Normally we convert configuration space into a graph representation containing

- waypoints
- edges joining them

These graph representations are compact and can be searched efficiently.

Visibility graphs

One graph-based approach uses vertices of objects as waypoints. Edges are straight lines connecting vertices. This "visibility graph" can be constructed incrementally, directly from a polygonal representation of the obstacles:



(from Howie Choset's book)

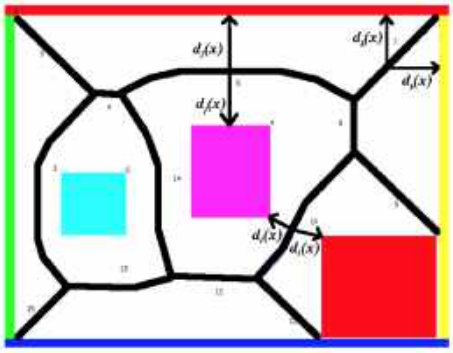
Visibility Graph Construction

This type of representation was popular in early configuration space algorithms. It produces paths of minimal length. However, these paths aren't very safe because they skim the edges of obstacles. It's risky to get too close to obstacles. We could easily hit the obstacle if there are any errors in our model of our shape, our position, and/or the obstacle position. For most practical applications, it's better to have a somewhat longer path that is reasonably short and doesn't come too close to obstacles.

Skeletonization

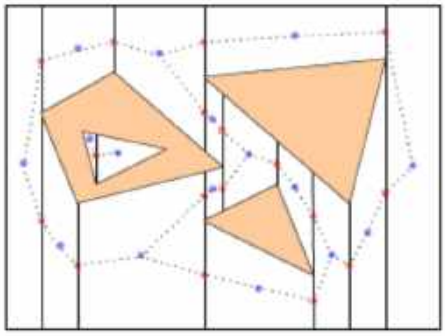
Another class of methods converts freespace into a skeleton that goes through the middle of regions, called a "roadmap". Paths via the skeleton stay far from obstacles, so they tend to be safe but perhaps overly long.

The "Generalized Voronoi Diagram" places the skeleton along curves that are equidistant from two or more obstacles. The waypoints are the places where these curves intersect.



(from Howie Choset)

The cell decomposition below divides free space somewhat more arbitrarily into trapezoids aligned with the coordinate axes. Waypoints are then located in the middle of each trapezoid and the middles of its edges.



(from Philippe Cheng, MIT, 2001)

Details of path construction

A roadmap for a configuration space is usually precomputed and then used to plan many paths, e.g. as the robot moves around its environment. The start and goal of the path will typically not be on the roadmap. So the path is constructed in three parts:

- from start onto nearest point on skeleton
- along skeleton
- to goal from nearest point on skeleton

Paths assembled from roadmap edges may be longer than needed, and the edges aren't joined together smoothly. Sharp changes in orientation may be impossible for the robot to execute and/or cause excessive wear on the joints. So it is standard practice to optimize them (e.g. by smoothing the curve) in a post-processing step.

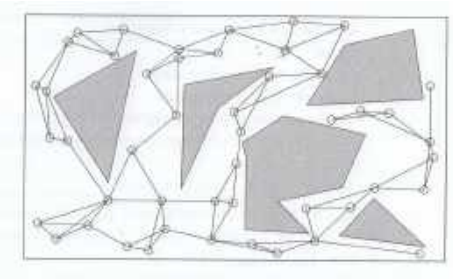
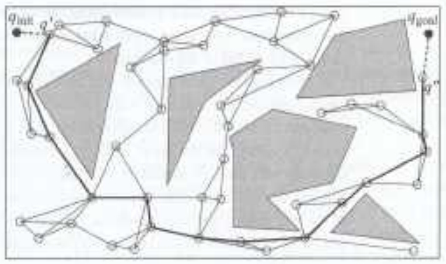
Probabilistic Roadmap (PRM) Planners

Recent algorithms create roadmaps more efficiently by sampling the configuration space without every computing the details of all the obstacles. Specifically we

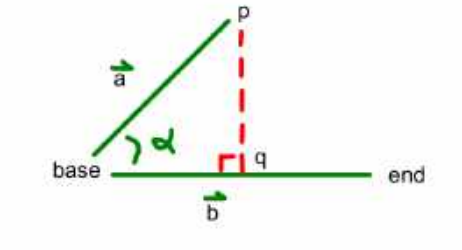
- Generate a large number of sample points in configuration space.
- Keep (as our waypoints) only the points that are in free space.
- Try to connect each waypoint to nearby waypoints.

There are fast algorithms for deciding whether a specific point is inside any obstacle. If waypoints are close together, there are also simple methods for connecting them, e.g. check whether the straight line between them is in free space.

Here is a probabilistic roadmap for a simple planning problem (left) and a plan constructed using it (right).



(from Howie Choset book)

Geometry Cheat Sheet



This picture shows a situation from MP 2. Vector $\vec{b}$ goes from the base of a robot arm's link to the end of the link. Vector $\vec{a}$ goes from the base to some other point p. Both vectors live in 2D and could point in any direction. We'll see some similar situations in higher dimensions when we get to vector semantics.

We've dropped a perpendicular from p to the line containing $\vec{b}$, intersecting the line at point q. Depending on the problem, we'd like the distance from p to q and/or the distance from the base to q. Let's think of these distances as signed, because that's often what you need. For example, the sign of the distance from the base to q tells you whether q on the correct side of the base to be in the line segment defined by base and end.

You'll recal from high school geometry that

- the distance from p to q is $|\vec{a}| \sin \alpha$, and
- the distance from the base to q is $|\vec{a}| \cos \alpha$

However, we'd prefer not to convert to/from angles because those library functions tend to be slow. So, instead, you should exploit the following equations:

- $\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}| \cos \alpha$
- $\vec{a} \times \vec{b} = |\vec{a}||\vec{b}| \sin \alpha$

$\vec{a} \cdot \vec{b}$ is the dot product. $\vec{a} \times \vec{b}$ is sometimes called the "cross product" in 2D geometrical algorithms. More exactly, it is the sign and magnitude of the cross product for two vectors that live in the plane $z = 0$. More concretely, if $\vec{a} = (a_1, a_2)$ and $\vec{b} = (b_1, b_2)$, then

- $\vec{a} \cdot \vec{b} = a_1 b_1 + a_2 b_2$
- $\vec{a} \times \vec{b} = a_1 b_2 - a_2 b_1$

Now you have to do some algebra. So you'll need to substitute one equation into another. Also you'll often need to calculate $|\vec{b}|$ so you can remove it from equations.

In higher dimensions, you can still use the dot product to compute cosine.

Here's one example of using a signed result. The vector $\vec{b}$ defines a line L. The sign of sin(α) tells you which side of this line p is on. If you have a line segment from p to another point, you can tell whether it crosses L by testing whether this sign changes between the two endpoints.

CS 440/ECE 448
Margaret Fleck

05 Probability

Motivation for statistical methods

Mulberry trees have edible berries and are critical for raising silkworms, The picture above shows a silk mill from a brief attempt (around 1834) to establish a silk industry in Northampton Massachusetts. But would you be able to tell a mulberry leaf from a birch leaf (below)? How would you even go about describing how they look different?

Which is mulberry and which is birch?

These pictures come Guide to Practical Plants of New England (Pesha Black and Micah Hahn, 2004). This guide includes a variety of established descrete features for identifying plants, such as

- leaf shape is broad or narrow
- attachment to stem is opposite or alternate
- edge of leaf is smooth or toothed

Unfortunately, both of these leaves are broad, alternate, and toothed. But they look different.

These descrete features work moderately well for plants. However, modelling categories with discrete (logic-like) features doesn't generalize well to many real-world situations.

- too complex on realistic situations

- too much need for hand crafting representations
- too much need for expert knowledge

In practice, this means that logic models are incomplete and fragile A better approach is to learn models from examples, e.g. pictures, audio data, big text collections. However, this input data presents a number of challenges:

- It has uncertainty (e.g. from sensors).
- It has errors (e.g. "Department of Computer Silence").
- It is incomplete, often selectively so. E.g. we have much more training data for English than Zulu.
- It may lack annotation (e.g. explicit type labels).

So we use probabilistic models to capture what we know, and how well we know it.

Discrete random variables

Suppose we're modelling the traffic light at University and Lincoln (two crossing streets plus a diagonal train track).

Some useful random variables might be:

variable	domain (possible values)
time	{morning, afternoon, evening, night}
is-train	{yes, no}
traffic-light	{red, yellow, green}
barrier-arm	{up, down}

Also exist continuous random variables, e.g. temperature (domain is all positive real numbers) or course-average (domain is [0-100]). We'll ignore those for now.

A state/event is represented by the values for all random variables that we care about.

Probabilities

P(variable = value) or P(A) where A is an event

> What percentage of the time does [variable] occur with [value]? E.g. P(barrier-arm = up) = 0.95

P(X=v and Y=w) or P(X=v,Y=w)

> How often do we see X=v and Y=w at the same time? E.g. P(barrier-arm=up, time=morning) would be the probability that we see a barrier arm in the up position in the morning.

P(v) or P(v,w)

> The author hopes that the reader can guess what variables these values belong to. For example, in context, it may be obvious that P(morning,up) is shorthand for P(barrier-arm=up, time=morning).

Probability notation does a poor job of distinguishing variables and values. So it is very important to keep an eye on types of variable and values, as well as the general context of what an author is trying to say. A useful heuristic is that

- variables are usually capital letters, and
- values are usually lowercase letters.

A distribution is an assignment of probability values to all events of interest, e.g. all values for particular random variable or pair of random variables.

Properties of probabilities

The key mathematical properities of a probability distribution can be derived from **Kolmogorov's axioms of probability**:

> $0 \leq P(A)$
>
> P(True) = 1
>
> P(A or B) = P(A) + P(B), if A and B are mutually exclusive events

It's easy to expand these three axioms into a more complete set of basic rules, e.g.

> $0 \leq P(A) \leq 1$
>
> P(True) = 1 and P(False) = 0
>
> P(A or B) = P(A) + P(B) - P(A and B) [inclusion/exclusion, same as set theory]
>
> If X has possible values p,q,r then P(X=p or X=q or X=r) = 1.

Where do probabilities come from?

Probabilities can be estimated from direct observation of examples (e.g. large corpus of text data). Constraints on probabilities may also come from sceientific beliefs

- Theory connects observable probabilities to ones we can't observe. (e.g. icard access fails --> icard broken or access list wrong, because EngIT told me)
- The underlying model must have some simple form (e.g. virus infections grow exponentially) which can be used to fill in data from incomplete observations.
- Anything is possible, i.e. $P(A) > 0$. (We'll see **smoothing** techniques that fill zero values with small numbers.)
- Nothing is guaranteed, i.e. $P(A) < 1$.

The assumption that anything is possible is usually made to deal with the fact that our training data is incomplete, so we need to reserve some probability for all the possible events that we haven't happened to see yet.

An intersection

Joint probabilities

Here's a model of two variables for the University/Goodwin intersection:

		E/W light		
		green	yellow	red
N/S light	green	0	0	0.2
	yellow	0	0	0.1
	red	0.5	0.1	0.1

To be a probability distribution, the numbers must add up to 1 (which they do in this example).

Most model-builders assume that probabilities aren't actually zero. That is, unobserved events do occur but they just happen so infrequently that we haven't yet observed one. So a more realistic model might be

		E/W light		
		green	yellow	red
N/S light	green	e	e	0.2-f
	yellow	e	e	0.1-f
	red	0.5-f	0.1-f	0.1-f

To make this a proper probability distribution, we need to set f=(4/5)e so all the values add up to 1.

Suppose we are given a joint distribution like the one above, but we want to pay attention to only one variable. To get its distribution, we sum probabilities across all values of the other variable.

		E/W light			marginals
		green	yellow	red	
N/S light	green	0	0	0.2	0.2
	yellow	0	0	0.1	0.1
	red	0.5	0.1	0.1	0.7
marginals		0.5	0.1	0.4	

So the marginal distribution of the N/S light is

P(green) = 0.2
P(yellow) = 0.1
P(red) = 0.7

To write this in formal notation suppose Y has values $y_1 \ldots y_n$. Then we compute the marginal probability P(X=x) using the formula $P(X = x) = \sum_{k=1}^{n} P(x, y_k)$.

Conditional probabilities

Suppose we know that the N/S light is red, what are the probabilities for the E/W light? Let's just extract that line of our joint distribution.

		E/W light		
		green	yellow	red
N/S light	red	0.5	0.1	0.1

So we have a distribution that looks like this:

P(E/W=green | N/S = red) = 0.5
P(E/W=yellow | N/S = red) = 0.1
P(E/W=red | N/S = red) = 0.1

Oops, these three probabilities don't sum to 1. So this isn't a legit probability distribution (see Kolmogorov's Axioms above). To make them sum to 1, divide each one by the sum they currently have (which is 0.7). This gives us

P(E/W=green | N/S = red) = 0.5/0.7 = 5/7
P(E/W=yellow | N/S = red) = 0.1/0.7 = 1/7
P(E/W=red | N/S = red) = 0.1/0.7 = 1/7

Conditional probability equations

Conditional probability models how frequently we see each variable value in some context (e.g. how often is the barrier-arm down if it's nighttime). The conditional probability of A in a context C is defined to be

$$P(A \mid C) = P(A,C)/P(C)$$

Many other useful formulas can be derived from this definition plus the basic formulas given above. In particular, we can transform this definition into

$$P(A,C) = P(C) * P(A \mid C)$$
$$P(A,C) = P(A) * P(C \mid A)$$

These formulas extend to multiple inputs like this:

$$P(A,B,C) = P(A) * P(B \mid A) * P(C \mid A,B)$$

Independence

Two events A and B are independent iff

$$P(A,B) = P(A) * P(B)$$

It's equivalent to show that this equation is equivalent to each of the following equations:

$$P(A \mid B) = P(A)$$
$$P(B \mid A) = P(B)$$

Exercise for the reader: why are these three equations all equivalent? Hint: use definition of conditional probability. Figure this out for yourself, because it will help you become familiar with the definitions.

Bikes


Red and teal veoride bikes from /u/civicsquid reddit post

Example: campus bikes

We have two types of bike (veoride and standard). Veorides are mostly teal (with a few exceptions), but other (privately owned) bikes rarely are.

Joint distribution for these two variables

	veoride	standard	
teal	0.19	0.02	0.21
red	0.01	0.78	0.79
	0.20	0.80	

For this toy example, we can fill out the joint distribution table. However, when we have more variables (typical of AI problems), this isn't practical. The joint distribution contains

- too many probability values to estimate (2^n values for n variables)
- too many probability values to store

The estimation is the more important problem because we usually have only limited amounts of data, esp. clean annotated data. Many value combinations won't be observed simply because we haven't seen enough examples.

Bayes Rule

Remember that P(A | C) is the probability of A in a context where C is true. For example, the probability of a bike being teal increases dramatically if we happen to know that it's a veobike.

P(teal) = 0.21
P(teal | veoride) = 0.95 (19/20)

The formal definition:of conditional probability is

P(A | C) = P(A,C)/P(C)

Let's write this in the equivalent form P(A,C) = P(C) * P(A | C).

Flipping the roles of A and C (seems like middle school but just keep reading):

equivalently P(C,A) = P(A) * P(C | A)

P(A,C) and P(C,A) are the same quantity. (AND is commutative.) So we have

P(A) * P(C | A) = P(A,C) = P(C) * P(A | C)

So

P(A) * P(C | A) = P(C) * P(A | C)

So we have Bayes rule

P(C | A) = P(A | C) * P(C) / P(A)

Here's how to think about the pieces of this formula

```
P(cause | evidence) =  P(evidence | cause) *  P(cause) / P(evidence)
  posterior            likelihood              prior     normalization
```

The normalization doesn't have a proper name because we'll typically organize our computation so as to eliminate it. Or, sometimes, we won't be able to measure it directly, so the number will be set at whatever is required to make all the probability values add up to 1.

Example: we see a teal bike. What is the chance that it's a veoride? We know

P(teal | veoride) = 0.95 [likelihood]
P(veoride) = 0.20 [prior]
P(teal) = 0.21 [normalization]

So we can calculate

P(veoride | teal) = 0.95 x 0.2 / 0.21 = 0.19/0.21 = 0.905

Why will this be useful?

In non-toy examples, it will be easier to get estimates for P(evidence | cause) than direct estimates for P(cause | evidence).

P(evidence | cause) tends to be stable, because it is due to some underlying mechanism. In this example, veoride likes teal and regular bike owners don't. In a disease scenario, we might be looking at the tendency of measles to cause spots, which is due to a property of the virus that is fairly stable over time.

P(cause | evidence) is less stable, because it depends on the set of possible causes and how common they are right now.

For example, suppose that a critical brake problem is discovered and Veoride suddenly has to pull bikes into the shop for repairs. So then only 2% of the bikes are veorides. Then we might have the following joint distribution. Now a teal bike is more likely to be standard rather than veoride.

```
              veoride        standard     |
teal             0.019          0.025     |   0.044
red              0.001          0.955     |   0.956
```

```
                          ---------------------------
                             0.02           0.98
```

Dividing P(cause | evidence) into P(evidence | cause) and P(cause) allows us to easily adjust for changes in P(cause).

Recall: Bayes Rule

```
P(cause | evidence) =  P(evidence | cause) *  P(cause) / P(evidence)

posterior            likelihood             prior     normalization
```

The MAP estimate

Example: We have two underlying types of bikes: veoride and standard. We observe a teal bike. What type should we guess that it is?

"Maximum a posteriori" (MAP) estimate chooses the type with the highest posterior probability:

> pick the type X such that P(X | evidence) is highest

Suppose that type X comes from a set of types T. Then MAP chooses

> $\text{argmax}_{X \in T}\, P(X|\text{evidence})$

The argmax operator iterates through a series of values for the dummy variable (X in this case). When it determines the maximum value for the quantity (P(X | evidence) in this example), it returns the value for X that produced that maximum value.

Completing our example, we compute two posterior probabilities:

> P(veoride | teal) = 0.905
> P(standard | teal) = 0.095

The first probability is bigger, so we guess that it's a veoride. Notice that these two numbers add up to 1, as probabilities should.

Ignoring the normalizing factor

P(evidence) can usually be factored out of these equations, because we are only trying to determine the relative probabilities. For the MAP estimate, in fact, we are only trying to find the largest probability. In our equation

> P(cause | evidence) = P(evidence | cause) * P(cause) / P(evidence)

P(evidence) is the same for all the causes we are considering. Said another way, P(evidence) is the probability that we would see this evidence if we did a lot of observations. But our current situation is that we've actually seen this particular evidence and we don't really care if we're analyzing a common or unusual situation.

So Bayesian estimation often works with the equation

> P(cause | evidence) $\propto$ P(evidence | cause) * P(cause)

where we know (but may not always say) that the normalizing constant is the same across various such quantities.

Specifically, for our example

> P(veoride | teal) = P(teal | veoride) * P(veoride) / P(teal)
> P(standard | teal) = P(teal | standard) * P(standard) / P(teal)

P(teal) is the same in both quantities. So we can remove it, giving us

> P(veoride | teal) $\propto$ P(teal | veoride) * P(veoride) = 0.95 * 0.2 = 0.19
> P(standard | teal) $\propto$ P(teal | standard) * P(standard) = 0.025 * 0.8 = 0.02

So veoride is the MAP choice, and we never needed to know P(teal). Notice that the two numbers (0.19 and 0.02) don't add up to 1. So they aren't probabilities, even though their ratio is the ratio of the probabilities we're interested in.

The MLE estimate

As we saw above, our joint probabilities depend on the relative frequencies of the two underlying causes/types. Our MAP estimate reflects this by including the prior probability.

If we know that all causes are equally likely, we can set all P(cause) to the same value for all causes. In that case, we have

> P(cause | evidence) $\propto$ P(evidence | cause)

So we can pick the cause that maximizes P(evidence | cause). This is called the "Maximum Likelihood Estimate" (MLE).

The MLE estimate can be very inaccurate if the prior probabilities of different causes are very different. On the other hand, it can be a sensible choice if we have poor information about the prior probabilities.

CS 440/ECE 448
Margaret Fleck

06 Naive Bayes


Red and teal veoride bikes from /u/civicsquid reddit post

Suppose we are building a more complete bike recognition system. It might observe (e.g. with its camera) features like:

- where the bike is parked
- whether it has a kickstand.
- whether it has a filled triangular part

How do we combine these types of evidence into one decision about whether the bike is a veoride?

As we saw earlier, a large number of parameters would be required to model all of the 2^n interactions between n variables. The Naive Bayes algorithm reduces the number of parameters to O(n) by making assumptions about how the variables interact,

Conditional independence

Remember that two events A and B are independent iff

$$P(A,B) = P(A) * P(B)$$

Sadly, independence rarely holds. A more useful notion is conditional independence. That is, are two variables independent when we're in some limited context. Formally, two events A and B are conditionally independent given event C if and only if

$$P(A, B \mid C) = P(A|C) * P(B|C)$$

Conditional independence is also unlikely to hold exactly, but it is a sensible approximation in many modelling problems. For example, having a music stand is highly correlated with owning a violin. However, we may be able to treat the two as independent (without too much modelling error) in the context where the owner is known to be a musician.

The definiton of conditional independence is equivalent to either of the following equations:

$$P(A \mid B,C) = P(A \mid C)$$
$$P(B \mid A,C) = P(B \mid C)$$

It's a useful exercise to prove that this is true, because it will help you become familiar with the definitions and notation.

Basic Naive Bayes model

Suppose we're observing two types of evidence A and B related to cause C. So we have

$$P(C \mid A,B) \propto P(A,B \mid C) * P(C)$$

Suppose that A and B are conditionally independent given C. Then

$$P(A, B \mid C) = P(A|C) * P(B|C)$$

Substituting, we get

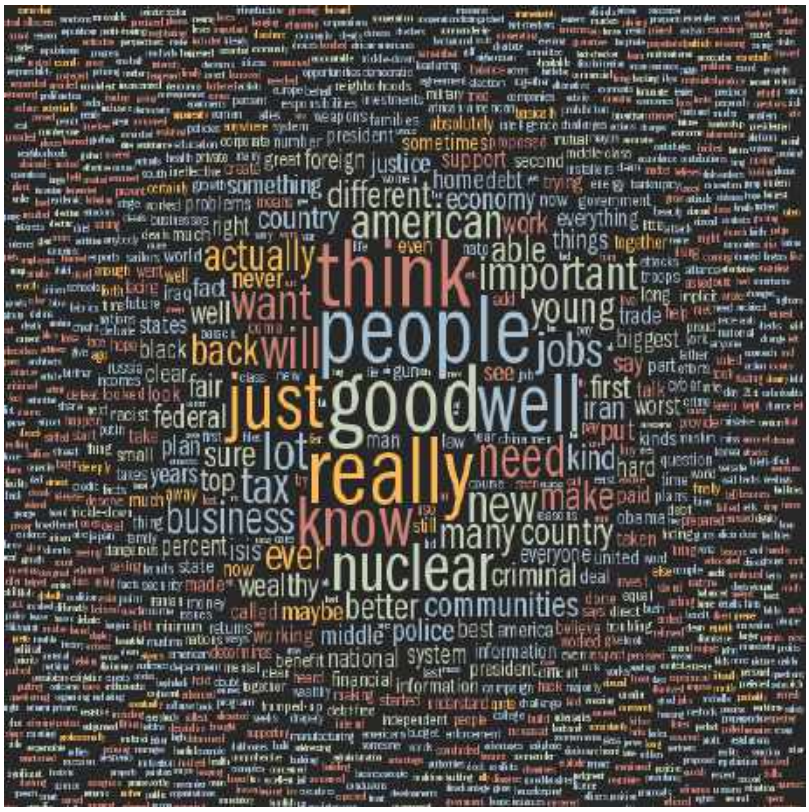
$$P(C \mid A,B) \propto P(A|C) * P(B|C) * P(C)$$

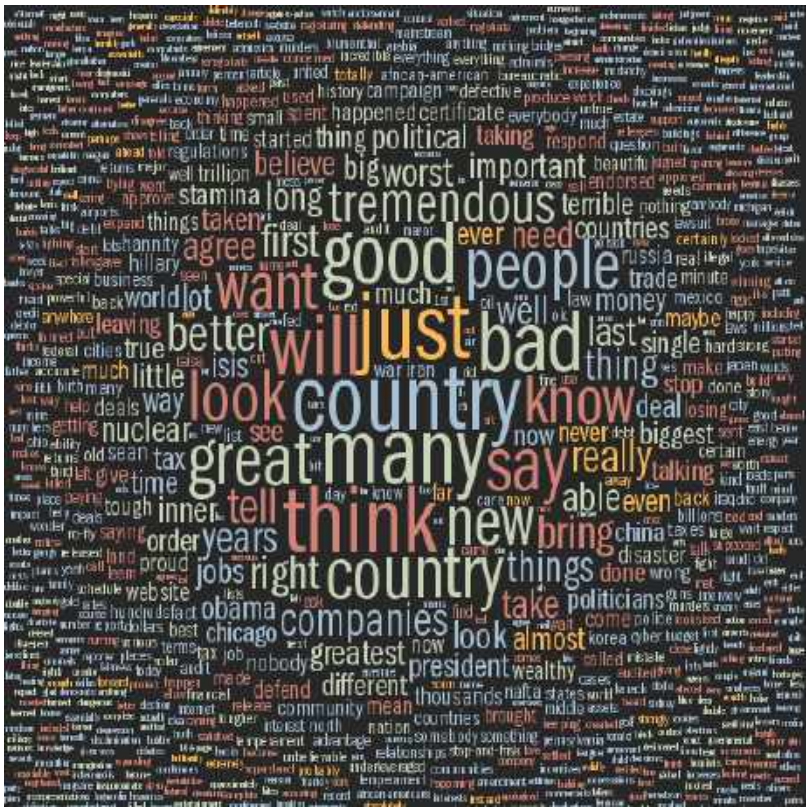
This model generalizes in the obvious way when we have more than two types of evidence to observe. Suppose our cause C has n types of evidence, i.e. random variables $E_1, \ldots, E_n$. Then we have

$$P(C|E_1, \ldots E_n) \propto P(E_1|C) * \ldots * P(E_n|C) * P(C) = P(C) * \prod_{k=1}^{n} P(E_k|C)$$

So we can estimate the relationship to the cause separately for each type of evidence, then combine these pieces of information to get our MAP estimate. This means we don't have to estimate (i.e. from observational data) as many numbers. For n variables, a naive Bayes model has only O(n) parameters to estimate, whereas the full joint distribution has 2^n parameters.

Two word clouds





Classification

The basic task is

> **Input:** a collection of objects of several types
> **Output:** correct type label for each object

In practice, this means

> **Input:** a collection of objects of several types
> Extract features from each object
> **Output:** Use the features to label each object with its correct type

For example, we might wish to take a collection of documents (i.e. text files) and classify each document as

- spam or not spam
- politics vs. science vs. recipe
- British vs. American English

Bag of words model

Let's look at the British vs. American classification task. Certain words are characteristic of one dialect or the other. For example

- UK: boffin, petrol, chips, lorry, queue, pushchair
- US: egghead, gas, French fries, truck, line, stroller

See this list of British and American words.

The word "boffin" differs from "egghead" because it can easily be modified to make words like "boffinry" and "astroboffin".

Dialects have special words for local foods, e.g. the South African English words "biltong," "walkie-talkie," "bunny chow."

The word clouds at the top of this page show which words (content words only) were said most by Clinton (left) and Trump (right) in their first debate.

See more details at Martin Krzywinski's Word Analysis of 2016 Presidential Debates Bag of words model: We can use the individual words as features.

A bag of words model determines the class of a document based on how frequently each individual word occurs. It ignores the order in which words occur, which ones occur together, etc. So it will miss some details, e.g. the difference between "poisonous" and "not poisonous,"

Text classification with Naive Bayes.

Suppose that cause C produces n observable effects $E_1, \ldots, E_n$. Then our Naive Bayes model calculates the MAP estimate as

$$P(C|E_1, \ldots E_n) \ \propto \ P(E_1|C) * \ldots * P(E_n|C) * P(C) \ = \ P(C) * \prod_{k=1}^{n} P(E_k|C)$$

Let's map this idea over to our Bag of Words model of text classification. Suppose our input document contains the string of words $W_1, \ldots, W_n$

Notice that some of the words might be identical. E.g the sentence "The big cat saw the small cat" contains two pairs of repeated words. For example, W_2 is "cat" and W_7 is also "cat". Our estimation process will consider the two copies of "cat" separately

Jargon:

- word type (aka unique word): a dictionary entry such as "cat"
- word token: a word in a specific position in the text

So our example sentence has seven tokens drawn from five types.

To classify the document, naive Bayes compares these two values

$$P(\text{UK}|W_1, \ldots, W_n) \propto P(\text{UK}) * \prod_{k=1}^{n} P(W_k|\text{UK})$$
$$P(\text{US}|W_1, \ldots, W_n) \propto P(\text{US}) * \prod_{k=1}^{n} P(W_k|\text{US})$$

To do this, we need to estimate two types of parameters

- the likelihoods P(W | UK) and P(W | US) for each word type W.
- the priors P(UK) and P(US)

Headline results

Naive Bayes works quite well in practice, especially when coupled with a number of tricks that we'll see later. The Naive Bayes spam classifier SpamCop, by Patrick Pantel and Dekang Lin (1998) got down to a 4.33% error rate using bigrams and some other tweaks.

Suppose we have n word types, then creating our bag of words model requires estimating O(n) probabilities. This is much smaller than the full conditional probability table which has 2^n entries. The big problem with too many parameters is having enough training data to estimate them reliably. (I said this last lecture but it deserves a repeat.)

Gender classification example

A Quantitative Analysis of Lexical Differences Between Genders in Telephone Conversations Constantinos Boulis and Mari Ostendorf, ACL 2005

Fisher corpus of more than 16,000 transcribed telephone conversations (about 10 minutes each). Speakers were told what subject to discuss. Can we determine gender of the speaker from what words they used?

Most frequent words in any spoken corpus are very common words that are usually ignored in estimation, called "stop words." These include

- function words: the, a, you, in,....
- fillers: um, uh, I mean, so, ...
- backchannel: uh-uh, yeah

Stop words are often omitted in classification, because they don't tell us much about the topic. However, this is task dependent. Stop words can provide useful information about the speaker's mental state, relation to the listener, etc. See James Pennebaker "The secret life of pronouns".

Words most useful for discriminating speaker's gender (i.e. used much more by men than women or vice versa)

- **men** dude [expletive] [expletive] wife wife's matt steve bass ben [expletive]
- **women** husband husband's refunding goodness boyfriend coupons crafts linda gosh cute

This paper compares the accuracy of Naive Bayes to that of an SVM classifier, which is a more sophisticated classification technique.

- Naive Bayes 83%

- SVM 88.6%

The more sophisticated classifier makes an improvement, but not a dramatic one.

A animal that's rare in Illinois

 Armadillo (picture from Prairie Rivers Network)

Forward Outline

The high level classification model is simple, but a lot of details are required to get high quality results. Over the next three videos, we'll look at four aspects of this problem:

- Testing and evaluation
- Defining the word tokens
- Estimating the probabilities
- Extending the model to sequences of adjacent words

The process of testing

Training data: estimate the probability values we need for Naive Bayes

Development data: tuning algorithm details

Test data: final evalution (not always available to developer) or the data seen only when system is deployed

Training data only contains a sampling of possible inputs, hopefully typically but nowhere near exhaustive. So development and test data will contain items that weren't in the training data. E.g. the training data might focus on apples where the development data has more discussion of oranges. Development data is used to make sure the system isn't too sensitive to these differences. The developer typically tests the algorithm both on the training data (where it should do well) and the development data. The final test is on the test data.

Evaluation metrics

Results of classification experiments are often summarized into a few key numbers. There is often an implicit assumption that the problem is asymmetrical: one of the two classes (e.g. Spam) is the target class that we're trying to identify.

	Labels from Algorithm	
	Spam	**Not Spam**
Correct = Spam	True Positive (TP)	False Negative (FN)

Correct = Not Spam	False Positive (FP)	True Negative (TN)

We can summarize performance using the rates at which errors occur:

- False positive rate = FP/(FP+TN) [how many wrong things are in the negative outputs]
- False negative rate = FN/(TP+FN) [how many wrong things are in the positive outputs]
- Accuracy = (TP+TN)/(TP+TN+FP+FN)
- Error rate = 1-accuracy

Or we can ask how well our output set contains all, and only, the desired items:

- precision (p) = TP/(TP+FP) [how many of our outputs were correct?]
- recall (r) = TP/(TP+FN) [how many of the correct answers did we find?]
- F1 = 2pr/(p+r)

F1 is the harmonic mean of precision and recall. Both recall and precision need to be good to get a high F1 value.

For more details, see the Wikipedia page on recall and precision

We can also display a confusion matrix, showing how often each class is mislabelled as a different class. These usually appear when there are more than two class labels. They are most informative when there is some type of normalization, either in the original test data or in constructing the table. So in the table below, each row sums to 100. This makes it easy to see that the algorithm is producing label A more often than it should, and label C less often.

	Labels from Algorithm		
	A	**B**	**C**
Correct = A	95	0	5
Correct = B	15	83	2
Correct = C	18	22	60

Forward Outline

We're now going to look at details required to make this classification algorithm work well.

Be aware that we're now talking about tweaks that might help for one application and hurt performance for another. "Your mileage may vary."

For a typical example of the picky detail involved in this data processing, see the method's section from the word cloud example in the previous lecture.

Defining the word tokens

Our first job is to produce a clean string of words, suitable for use in the classifier. Suppose we're dealing with a language like English. Then we would need to

- divide at whitespace
- normalize punctuation, html tags, capitalization, etc

This process is called tokenization. Format features such as punctuation and capitalization may be helpful or harmful, depending on the task. E.g. it may be helpful to know that "Bird" is different from "bird," because the former might be a proper name. But perhaps we'd like to convert "Bird" to "CAP bird" to make its structure explicit. Similarly, "tyre" vs. "tire" might be a distraction or an important cue for distinguishing US from UK English. So "normalize" may either involve throwing away the information or converting it into a format that's easier to process.

After that, it often helps to normalize the form of each word using a process called stemming.

Stemming

Stemming converts inflected forms of a word to a single standardized form. It removes prefixes and suffixes, leaving only the content part of the word (its "stem"). For example

```
help, helping, helpful --->  help
happy, happiness --> happi
```

Output is not always an actual word.

Stemmers are surprisingly old technology.

- first one: Julie Beth Lovins 1968 (!!)
- standard one: Martin Porter 1980

Porter made modest revisions to his stemmer over the years. But the version found in nltk (a currently popular natural language toolkit) is still very similar to the original 1980 algorithm.

Languages unlike English

English has relatively short words, with convenient spaces between them. Tokenization requires slightly different steps for languages with longer words or different writing system. For example, Chinese writing does not put spaces between words. So "I went to the store" is written as a string of characters, each representing one syllable (below). So programs processing Chinese text typically run a "word segmenter" that groups adjacent characters (usually 2-3) into a word.

In German, we have the opposite problem: long words. So we might divide words like "Wintermantel" into "winter" (winter) and "mantel" (coat).

In either case, the output tokens should usually form coherent units, whose meaning cannot easily be predicted from smaller pieces.

Very frequent and very rare words

Classification algorithms frequently ignore very frequent and very rare words. Very frequent words ("stop words") often convey very little information about the topic. So they are typically deleted before doing a bag of

words analysis.

Rare words include words for uncommon concepts (e.g. armadillo), proper names, foreign words, and simple mistakes (e.g. "Computer Silence"). Words that occur only one often represent a large fraction (e.g. 30%) of the word types in a dataset. So rare words cause two problems

- Their probabilities are hard to estimate accurately.
- They are very numerous, so they make our data tables much larger.

Rare words may be deleted. More often, all all rare words are mapped into a single placeholder value (e.g. UNK). This allows us to treat them all as a single item, with a reasonable probability of being observed.

N-gram models

The Bag of Words model uses individual words as features, ignoring the order in which they appeared. Text classification is often improved by using pairs of words (bigrams) as features. Some applications (notably speech recognition) use trigrams (triples of words) or even longer n-grams. E.g. "Tony the Tiger" conveys much more topic information than either "Tony" or "tiger". In the Boulis and Ostendorf study of gender classification bigram features proved significantly significantly more powerful.

For n words, there are n^2 possible bigrams. But our amount of training data stays the same. So the training data is even less adequate for bigrams than it was for single words.

Recap: naive Bayes document classification

Recall our document classification problem. Our input document is a string of words (perhaps not all identical) $W_1, \ldots, W_n$

Suppose that we have cleaned up our words, as discussed in the last video and now we're trying to label input documents as either CLA or CLB. To classify the document, our naive Bayes, bag of words model, compares these two values:

$$P(\text{CLA}) * \prod_{k=1}^{n} P(W_k|\text{CLA})$$
$$P(\text{CLB}) * \prod_{k=1}^{n} P(W_k|\text{CLB})$$

We need to estimate the parameters P(W | CLA) and P(W | CLB) for each word type W.

Simplistic parameter estimation

Now, let's look at estimating the probabilities. Suppose that we have a set of documents from a particular class C (e.g. CLA English) and suppose that W is a word type. We can define:

count(W) = number of times W occurs in the documents
n = number of total words in the documents

A naive estimate of P(W|C) would be $P(W|\text{C}) = \frac{\text{count}(W)}{n}$. This is not a stupid estimate, but a few adjustments will make it more accurate.

Problem 1: underflow

The probability of most words is very small. For example, the word "Markov" is uncommon outside technical discussions. (It was mistranscribed as "mark off" in the original Switchboard corpus.) Worse, our estimate for P(W|C) is the product of many small numbers. Our estimation process can produce numbers too small for standard floating point storage.

To avoid numerical problems, researchers typically convert all probabilities to logs. That is, our naive Bayes algorithm will be maximizing

$$\log(P(\mathrm{C}|W_1, \ldots W_k)) \quad \propto \quad \log(P(\mathrm{C})) + \sum_{k=1}^{n} \log(P(W_k|\mathrm{C}))$$

Notice that when we do the log transform, we need to replace multiplication with addition.

Problem 2: overfitting the training data

Recall that we train on one set of documents and then test on a different set of documents. For example, in the Boulis and Ostendorf gender classifcation experiment, there were about 14M words of training data and about 3M words of test data.

Training data doesn't contain all words of English and is too small to get a representative sampling of the words it contains. So for uncommon words:

- Words that didn't appear in the training data get estimated zero probability.
- Words that were uncommon in the training data get inaccurate estimates:

Smoothing

Smoothing assigns non-zero probabiities to unseen words. Since all probabilities have to add up to 1, this means we need to reduce the estimated probabilities of the words we have seen. Think of a probability distribution as being composed of stuff, so there is a certain amount of "probability mass" assigned to each word. Smoothing moves mass from the seen words to the unseen words.

Questions:

- Which words should lose mass: mostly infrequent ones (esp. seen only once)
- Which words should gain mass? How much?

Some unseen words might be more likely than others, e.g. if it looks like a legit English word (e.g. boffin) it might be more likely than something using non-English characters (e.g. bête).

Smoothing is critical to the success of many algorithms, but current smoothing methods are witchcraft. We don't fully understand how to model the (large) set of words that haven't been seen yet, and how often different types might be expected to appear. Popular standard methods work well in practice but their theoretical underpinnings are suspect.

Laplace smoothing

Laplace smoothing is a simple technique for addressing the problem:

All unseen words are represented by a single word UNK. So the probability of UNK should represent all of these words collectively.

n = number of words in our Class C training data
count(W) = number of times W appeared in Class C training data

To add some probability mass to UNK, we increase all of our estimates by α, where α is a tuning constant between 0 and 1 (typically small). So our probabilities for UNK and for some other word W are:

$$P(\text{UNK} \mid C) = \frac{\alpha}{n}$$
$$P(W \mid C) = \frac{count(W)+\alpha}{n}$$

This isn't quite right, because probabilities need to add up to 1. So we revise the denominators to make this be the case. Our final estimates of the probabilities are

V = number of word TYPES seen in training data

$$P(\text{UNK} \mid C) = \frac{\alpha}{n+\alpha(V+1)}$$
$$P(W \mid C) = \frac{\text{count}(W)+\alpha}{n+\alpha(V+1)}$$

Performance of Laplace smoothing

- overestimates probability of unseen words
- underestimates probability of common words

Deleted estimation

A technique called deleted estimation (or cross-validation or held-out estimation) can be used to directly measure these estimation errors.

Let's empirically map out how much counts change between two samples of text from the same source. So we divide our training data into two halves 1 and 2. Here's the table of word counts:

- **Experiment: Divide 3M words of Wall Street Journal into 2 halves. Compare word counts across the two halves.**

Count 1	Count 2	Word	Count 1	Count 2	Word
1	0	abacuses	1	2	abilities
11	6	abandon	86	72	ability
29	21	abandoned	1	0	ability...
4	8	abandoning	0	1	ablaze
0	2	abandonment	192	149	able
2	0	abandons	0	1	able-bodied
1	0	abashed	4	9	abnormal
0	2	abate	0	2	abnormalities
1	1	abated	0	2	abnormality

(from Mark Liberman via Mitch Marcus)

Now, let's pick a specific count r in the first half. Suppose that $W_1, \ldots, W_k$ are the words that occur r times in the first half. We can estimate the corrected count as

Corr(r) = average count of $W_1, \ldots, W_k$ in Half 2

This gives us the following correction table:

Half 1	Half 2	Half 1	Half 2
0	1.60491	8	7.53499
1	0.639544	9	8.27005
2	1.59014	10	9.50197
3	2.55045	11	10.0348
4	3.49306	12	11.2292
5	4.45996	13	12.7391
6	5.23295	14	12.5298
7	6.28311	15	14.1646

(from Mark Liberman via Mitch Marcus)

Now, if there were n_1 words in the first half of our data, we can use $\frac{\text{Corr}(r)}{n_1}$ as estimate of the true probability for each word W_i whose raw count was r.

Tweaks on deleted estimation

Make the estimate symmetrical. Let's assume for simplicity that both halves contain the same number of words (at least approximately). Then compute

> Corr(r) as above
> Corr'(r) reversing the roles of the two datasets
> $\frac{\text{Corr}(r)+\text{Corr}'(r)}{2}$ estimate of the true count for a word with observed count r

Effectiveness of this depends on how we split the training data in half. Text tends to stay on one topic for a while. For example, if we see "boffin," it's likely that we'll see a repeat of that word soon afterwards. Therefore

- Splitting the data in the middle works poorly.
- Better to split by even/odd sentences.

These corrected estimates still have biases, but not as bad as the original ones. The method assumes that our eventual test data will be very similar to our training data, which is not always true.

N-gram smoothing

Because training data is sparse, smoothing is a major component of any algorithm that uses n-grams. For single words, smoothing assigns the same probability to all unseen words. However, n-grams have internal structure which we can exploit to compute better estimates.

Example: how often do we see "cat" vs. "armadillo" on local subreddit? Yes, an armadillo has been seen in Urbana. But mentions of cats are much more common. So we would expect "the angry cat" to be much more common than "the angry armadillo". So

> Idea 1: If we haven't seen an ngram, guess its probability from the probabilites of its prefix (e.g. "the angry") and the last word ("armadillo").

Certain contexts allow many words, e.g. "Give me the ...". Other contexts strongly determine what goes in them, e.g. "scrambled" is very often followed by "eggs".

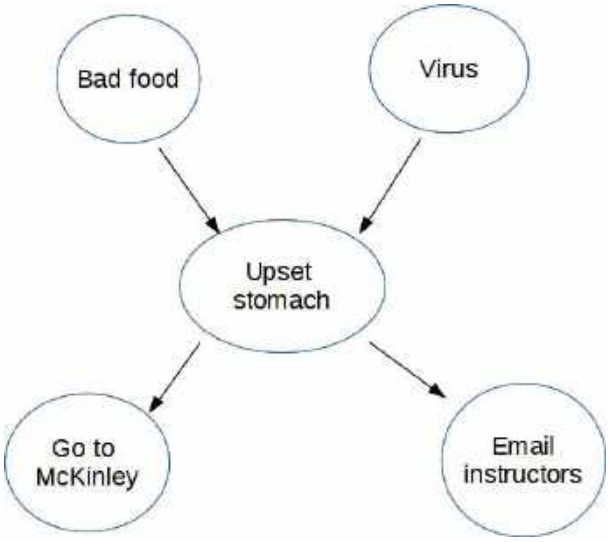
Idea 2: Guess that an unseen word is more likely in contexts where we've seen many different words.

There's a number of specific ways to implement these ideas, largely beyond the scope of this course.

07 Bayes Nets

Main idea

Bayes nets (a type of Graphical Model) compactly represent how variables influence one another Here's a Bayes net with five variables.

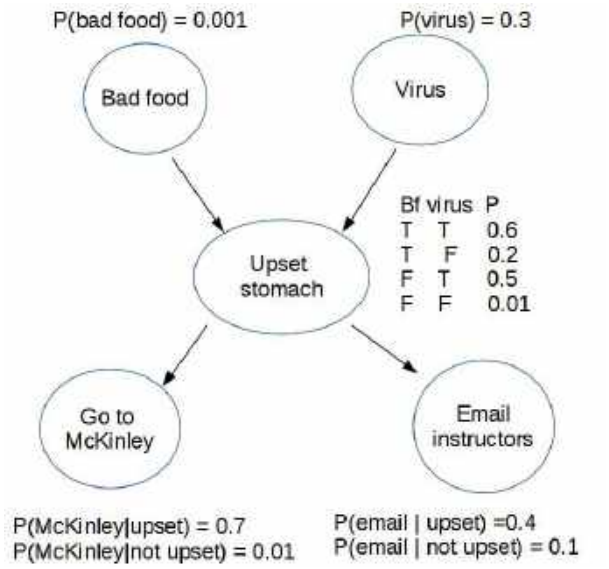


For simplicity, suppose these are all boolean (T/F) variables.

Notice that this diagram shows the causal structure of the situation, i.e. which events are likely to cause which other events. For example, we know that a virus can cause an upset stomach. Technically the design of a Bayes net isn't required to reflect causal structure. But they are more compact (therefore more useful) when they do.

Add probability information

For each node in the diagram, we write its probability in terms of the probabilities of its parents.



"Bad food" and "Virus" don't have parents, so we give them non-conditional probabilities. "Go to McKinley" and "Email instructors" have only one parent, so we need to spell out how likely they are for both possible values of the parent node.

"Upset stomach" has two parents, i.e. two possible causes. In this case, the two causes might interact. So we need to give a probability for every possible combination of the two parent values.

Recap: independence and conditional independence

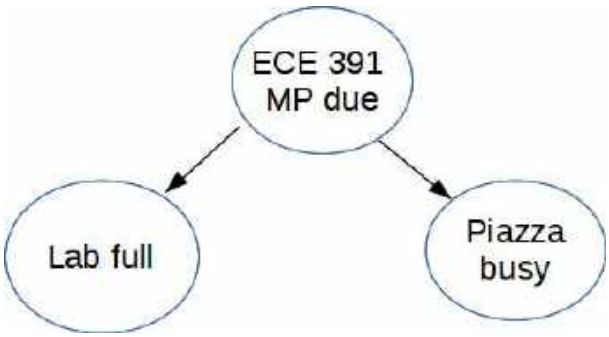
Two random variables A and B are independent iff P(A,B) = P(A) * P(B).

A and B are conditionally independent given C iff

P(A, B | C) = P(A|C) * P(B|C)
equivalently P(A | B,C) = P(A | C)
equivalently P(B | A,C) = P(B | C)

Conditional independence vs. independence #1

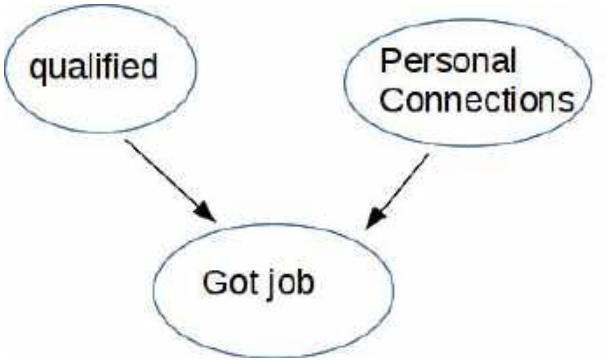
Here's a Bayes net modelling two effects with a common cause:



Busy piazza and a full lab aren't independent, but they are conditionally independent given that there's a project due.

Conditional independence vs. independence #2

Here's an example of two causes jointly helping create a single effect:



Being qualified is independent of having personal connections (more or less) But they aren't conditionally independent given that the person got the job. Supppose we know that they got the job but weren't qualified, then

(given this model of the causal relationships) they must have personal connections. This is called "explaining away" or "selection bias."

A high-level point: conditional independence isn't some weakened variant of independence. Neither property implies the other.

Conditional independence vs. independence #3

Here's a causal chain, i.e. a sequence of events in which each causes the following one.

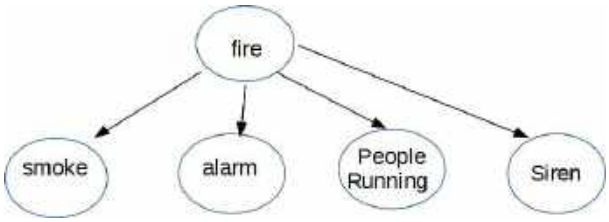
Candle and alarm are conditionally independent given fire. But they aren't independent, because P(alarm | candle) is much larger than P(alarm).

Example: independent events

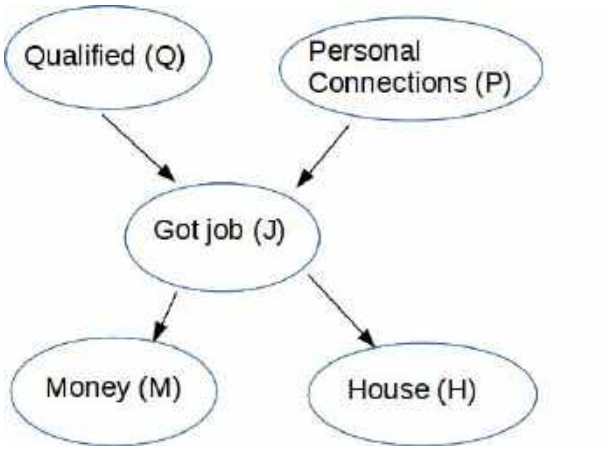
A set of independent variables generates a graphical model where the nodes aren't connected.

Example: Naive Bayes

Naive Bayes can be modelled with a Bayes net that has one ancestor (cause) and a number of children (effects).



A jobs-related Bayes net



Recap

Let's remember how to understand the Bayes net diagram above. We have some number of boolean (true/false) variables, five in this case. The edges in the diagram show which probabilities depend on one anohter. For each node, we store the probability of that variable in terms of the probabilities of its parents. For example

- Q has no parents, so we store the probability of Q being true.
- J has two parents, so we store the conditional probability of J being true given each possible combination of values for P and Q.

Partial orders

Recall from discrete math: a partial order orders certain pairs of objects but not others. It has to be consistent, i.e. no loops. So we can use it to draw a picture of the ordering with ancestors always higher on the page than descendents.

The arrows in a Bayes net are like a skeleton of a partial order: just relating parents to their children. The more general ancestor relationship is a true partial order.

For any set with a partial order, we can create a "topological sort." This is a total/linear order of all the objects which respects the partial order. That is, a node always follows its ancestors. Usually there is more than one possible topological sort and we don't care which one is chosen.

Exactly what are Bayes nets?

A Bayes net is a directed acyclic (no cycles) graph (called a DAG for short). So nodes are partially ordered by the ancestor/descendent relationships.

Each node is conditionally independent of its ancestors, given its parents. That is, the ancestors can ONLY influence node A by working via parents(A)

Suppose we linearize the nodes in a way that's consistent with the arrows (aka a topological sort). Then we have an ordered list of nodes $X_1, \ldots, X_n$. Our conditional independence condition is then

> For each k, $P(X_k | X_1, \ldots, X_{k-1}) = P(X_k | \text{parents}(X_k))$

Reconstructing the joint distribution

We can build the probability for any specific set of values by working through the graph from top to bottom. Suppose (as above) that we have the nodes ordered $X_1, \ldots, X_n$. Then

$$P(X_1, \ldots, X_n) = \prod_{k=1}^{n} P(X_k | \text{parents}(X_k))$$

$P(X_1, \ldots, X_n)$ is the joint distribution for all values of the variables.

In $\prod_{k=1}^{n} P(X_k | \text{parents}(X_k))$, each term $P(X_k | \text{parents}(X_k))$ is the information stored with one node of the Bayes net.

In this computation, we're starting from the causes (also called "query variables") such as bad food and virus above. Values propagate through the intermediate ("unobserved") nodes, and eventually we produce values for the visible effects (go to McKinley, email instructors). Much the time, however, we would like to work backwards from the visible effects to find the most likely cause(s).

Quantify Compactness

Suppose we have n variables. Suppose the maximum number of parents of any node in our Bayes net is m. Typically m will be much smaller than n.

- Full joint distribution requires 2^n probabilities
- Bayes net requires at most $n2^m$ probabilities

Building a Bayes net

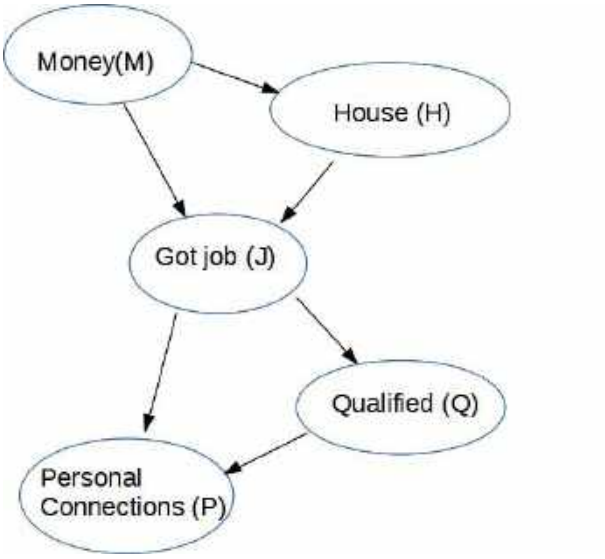
Suppose we know the full full joint probability distribution for our variables and want to convert this to a Bayes net.. We can construct a Bayes net as follows:

> Put the variables in any order $X_1, \ldots, X_n$.
>
> For k from 1 to n
>
> > Choose parents(X_k) to contain all variables from $X_1, \ldots, X_{k-1}$ that $P(X_k)$ depends on.

The output Bayes net depends on the order in which we process the variables. If our variable order is Q, P, J, M, H, then we get the above Bayes net, which contains 10 probability values. (1 each for Q and P, 2 each for M and H, 4 for J).

Suppose we use the variable order M, H, J, Q, P. Then we proceed as follows:

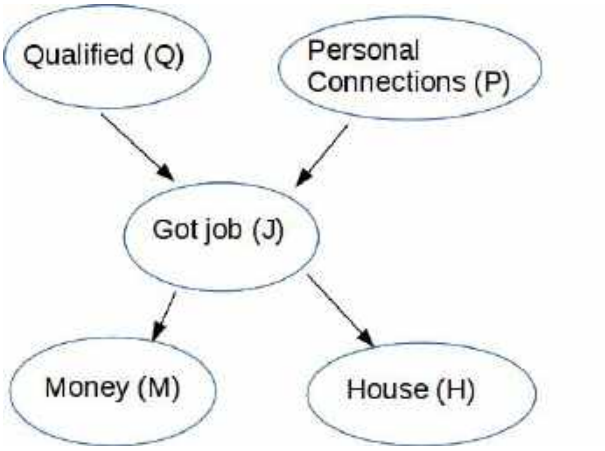
> M: no parents
> H: isn't independent of M (they were only conditionally independent given J), so M is its parent
> J: P(J) isn't independent of P(M) or P(H), nor is P(J | M,H) = P(J | H) so both of these nodes have to be its parents
> Q: P(Q | J,M,H) = P(Q | J) so just list J as parent
> P: probility depends on both J and Q



This requires that we store 13 probability numbers: 1 for M, 2 for H, 4 for J, 2 for Q, 4 for P.

So the choice of variable order affects whether we get a better or worse Bayes net representation. We could imagine a learning algorithm that tries to find the best Bayes net, e.g. try different variable orders and see which is most compact. In practice, we usually depend on domain experts to supply the causal structure.

The jobs graph again



What is Bayes net inference?

We'd like to use Bayes nets for computing most likely state of one node (the "query variable") from observed states of some other nodes. For example:

- We observe the state of M and H. (Does the person have money and/or a house?)
- We'd like to predict whether they were qualified (Q).

In addition to the query and observed nodes, we also have all the other nodes in the Bayes net. These unobserved/hidden nodes need to be part of the probability calculation, even though we don't know or care about their values.

Brute force inference

We'd like to do a MAP estimate. That is, figure out which value of the query variable has the highest probability given the observed effects. Recall that we use Bayes rule but omit P(effect) because it's constant across values of the query variable. So we're trying to optimize

$$\begin{aligned} P(\text{ cause } \mid \text{ effect }) &\propto P(\text{ effect } \mid \text{ cause }) P(\text{ cause }) \\ &= P(\text{ effect }, \text{ cause }) \end{aligned}$$

So, to decide if someone was qualified given that we have noticed that they have money and a house, we need to estimate $P(q \mid m, h) \propto P(q, m, h)$. We'll evaluate P(q,m,h) for both q=True and q=False, and pick the value (True or False) that maximizes P(q,m,h).

To compute this, $P(q, m, h)$, we need to consider all possibilities for how J and P might be set. So what we need to compute is

$$P(q, m, h) = \sum_{J=j, P=p} P(q, m, h, p, j)$$

P(q,m,h,p,j) contains five specific values for the variables Q,M,H,P,J. Two of them (m,h) are what we have observed. One (q) is the variable we're trying to predict. We'll do this computation twice, once for Q=true and once for Q=false. The last two values (j and p) are bound by the summation.

Notice that $P(Q = true, m, h)$ and $P(Q = false, m, h)$ do not necessarily sum to 1. These values are only proportional to $P(Q = true|m, h)$ and $P(Q = false|m, h)$, not equal to them.

If we start from the top of our Bayes net and work downwards using the probability values stored in the net, we get

$$P(q,m,h) = \sum_{J=j,P=p} P(q,m,h,p,j)$$
$$= \sum_{J=j,P=p} P(q) * P(p) * P(j \mid p,q) * P(m \mid j) * P(h \mid j)$$

Notice that we're expanding out all possible values for three variables (q,j,p). This will become a problem for problems involving more variables, because k binary variables have 2^k possible value assignments.

How to compute efficiently

Let's look at how we can organize our work. First, we can simplify the equation, e.g. move variables outside summations like this

$$P(q,m,h) = P(q) * \sum_{P=p} P(p) * \sum_{J=j} P(j \mid p,q) * P(m \mid j) * P(h \mid j)$$

Second, if you think through that computation, we'll have to compute $P(j \mid p,q) * P(m \mid j) * P(h \mid j)$ four times, i.e. once for each of the possible combinations of p and j values.

But some of the constituent probabilities are used by more than one branch. E.g. $P(h \mid J = true)$ is required by the branch where P=true and separately by the branch where P=false. Se we can save significant work by caching such intermediate values so they don't have to be recomputed. This is called memoization or dynamic programming.

If work is organized carefully (a long story), inference takes polynomial time for Bayes nets that are "polytrees." A polytree is a graph that has no cycles if you consider the edges to be non-directional (i.e. you can follow the arrows backwards or forwards).

Recap

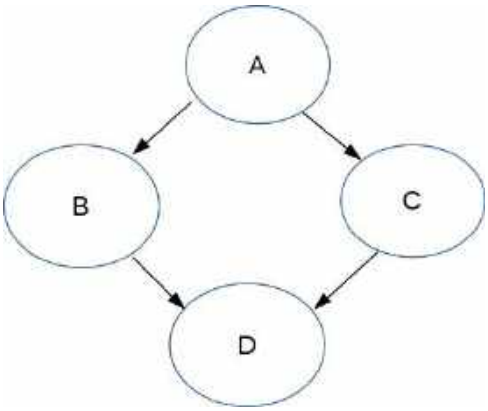
Recall that Bayes net inference is polynomial time (i.e. reasonably efficient) Bayes nets that are "polytrees." A polytree is a graph that has no cycles if you consider the edges to be non-directional. But what if our Bayes net wasn't a polytree?

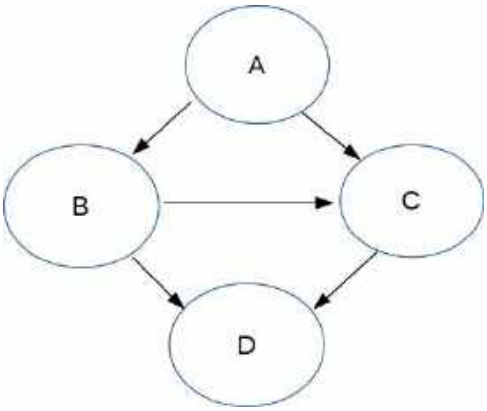
Junction tree algorithm

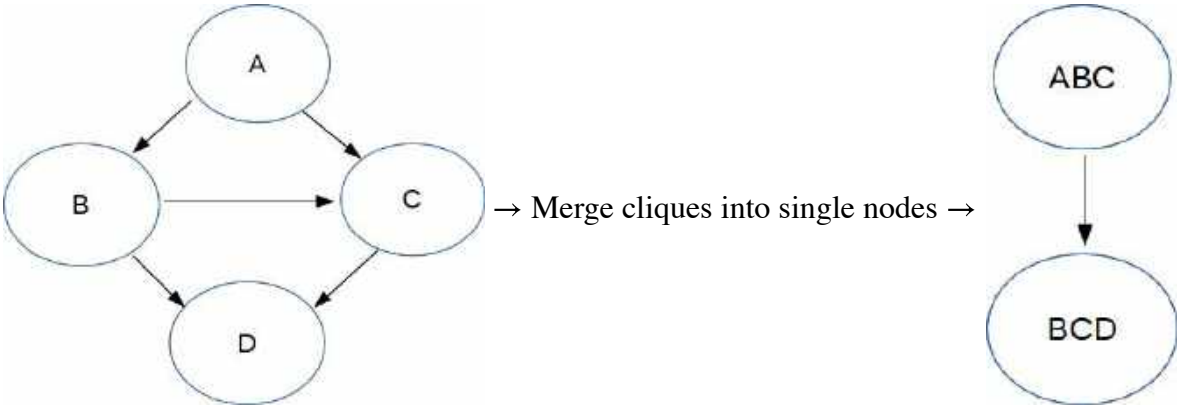
If your Bayes net is not a polytree, the junction tree algorithm will convert it to a polytree. This can allow efficient inference for a wider class of Bayes nets.

At a very high level, this algorithm first removes the directionality on the edges of the diagram. It then adds edges between nodes which aren't directly connected but are statistically connected (e.g. parents with a common child). This is called "moralization." It also adds additional edges to break up larger cycles (triangulaization). Fully-connected groups of nodes (cliques) are then merged into single nodes.

→ Moralize →







When the output diagrams are reasonably simple, they can then be used to generate solutions for the original diagram. (Long story, beyond the scope of this class.) However, this isn't guaranteed to work: the output graphs can be as complex as the original Bayes net. For example, notice that the new composite variable ABC has 8 possible values, because it's standing in for three boolean variables.

Quick review: NP complete

A decision problem is in NP (non-deterministic polynomial time) if you can provide a succinct (polynomial time verifiable) justification that a "yes" answer is correct. An example is whether a graph can be colored with k colors: just show us the coloring.

Problems in NP are widely believed to require exponential time. However, it is an open question whether there might be a better algorithm using only polynomial time.

A problem X is NP complete if a polynomial time algorithm for X would allow us to solve all problems in NP in polynomial time.

A very standard NP complete problem is 3SAT. The input 3SAT is a logical expression like this:

$$P = (X \text{ or } Y \text{ or } Z) \text{ and } (X \text{ or } \neg Y \text{ or } \neg Z) \text{ and } (X \text{ or } W \text{ or } Y)$$

Our algorithm must determine whether there is a set of input true/false values (for W, X, Y, Z) that will make the whole expression true. The input expression (P) must have the following form

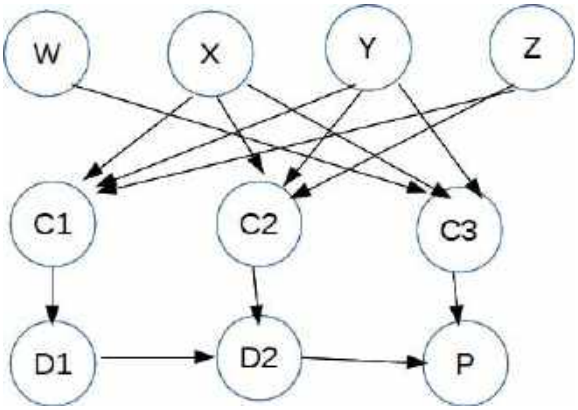
- It's the AND of a series of terms.
- Each term is the OR of up to 3 variables (plain or negated).

Bayes net inference is NP complete

We can prove that Bayes net inference is NP complete, by showing how to use Bayes Net inference to construct an algorithm for 3SAT. Suppose we have a 3SAT logical expression:

$$P = (X \text{ or } Y \text{ or } Z) \text{ and } (X \text{ or } \neg Y \text{ or } \neg Z) \text{ and } (X \text{ or } W \text{ or } Y)$$

We convert it to a Bayes net that looks like this



The top row of nodes are the input variables. The nodes C1, C2, C3 represent the terms. E.g. C1 represents (X or Y or Z). Their associated probability tables simulate an OR of their inputs. The nodes D1, D2, and P assemble the full logical expression, using probability tables that simulate the AND of their inputs. So D2 represents

$$(X \text{ or } Y \text{ or } Z) \text{ and } (X \text{ or } \neg Y \text{ or } \neg Z)$$

Our inference problem is whether we can find values for the input variables W,X,Y,Z that will give us a non-zero output probability value for P.

Bottom line

So, we have good and bad news about Bayes net inference:

- A useful range of examples can be solved efficiently. These include hidden Markov models (HMM's) which we'll see next week.
- Other examples probably require exponential time. Their values can be approximated using sampling methods beyond the scope of this course.

Further reading

A Brief Introduction to Graphical Models and Bayesian Networks by Kevin Murphy (1998)

Person: PICK UP A BIG RED BLOCK.
Computer: OK. (does it)
Person: GRASP THE PYRAMID.
Computer: I DON'T UNDERSTAND WHICH PYRAMID YOU MEAN.

Natural language main outline

The area of natural language and speech processing covers algorithms for three types of tasks:

- convert text/speech into structured information usable by an AI reasoning system
- generate fluent text/speech from structured information
- translate directly between two types of text/speech (e.g. two languages, simple types of question answering)

In the first two cases, a system might contain a complete end-to-end pipeline (e.g. speech to structured information). But it's also common for see applications that do only part of it, e.g. transcribing speech into written text. A human then takes over (e.g. reads and understands the transcripts). Generation is easier than understanding, but still hard to do well.

Prehistory

Interactive systems using natural language have been around since the early days of AI and computer games. For example:

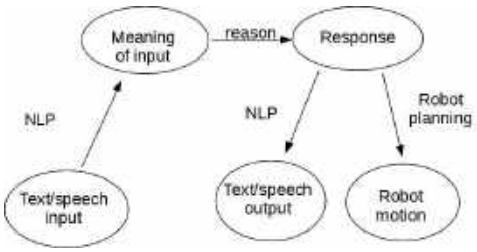
- SHRDLU (Terry Winograd, 1971). (from Stanford)
- Zork transcript Infocom, around 1980. (from Marleigh Norton's former page at Georgia Tech)

These older systems used simplified text input. The computer's output seems to be sophisticated, but actually involves a limited set of generation rules stringing together canned text. Most of the sophistication is in the backend world model. This enabled the systems to carry on a coherent dialog.

These systems also depend on social engineering: teaching the user what they can and cannot understand. People are very good at adapting to a system with limited, but consistent, linguistic skills.

Deep vs. systems

A "shallow" system converts fairly directly between its input and output, e.g. a translation system that transforms phrases into new phrases without much understanding of what the input means. A "deep" system uses a high-level representation as an intermediate step between its input and output. There is no hard-and-fast boundary between the two types of design. Both approaches can be useful in an appropriate context.



A deep system

In classical AI systems, the high-level representation might look like mathematical logic. E.g.

```
in(Joe,kitchen) AND holding(Joe,cheese)
```

Or perhaps we have a set of objects and events, each containing values for a number of named slots:

```
event
   name = "World War II"
   type = war
   years = (1939 1945)
   participants = (Germany, France, ....)
```

In modern systems based on neural nets, the high-level representation may look partly like a set of mysterious floating-point numbers. But these are intended to represent information at much the same level as the symbolic representations of meaning.

Shallow systems

Many of the useful current AI systems are shallow. That is, they process their input with little or no understanding of what it actually means. For example, suppose that you ask Google "How do I make zucchini kimchi?" or "Are tomatoes fruits?" Google doesn't actually answer the question but, instead, returns some paragraphs that seem likely to contain the answer. Shallow tools work well primarily because of the massive amount of data they have scraped off the web.

Similarly, summarization systems are typically shallow. These systems combine and select text from several stories, to produce a single short abstract. The user may read just the abstract, or decide to drill down and read some/all of the full stories. Google news is a good example of a robust summarization system. These systems make effective use of shallow methods, depending on the human to finish the job of understanding.

Text translations systems are often surprisingly shallow, with just enough depth of representation to handle changes in word order between languages.

Google translate is impressive, but can also fail catastrophically. A classic test is a circular translation: X to Y and then back to X. Google translate sweeps up text from the entire internet but can still be made to fail. It's best to pick somewhat uncommon topics (making kimchi, Brexit) and/or a less-commmon language, so that the translation system can't simply regurgitate memorized chunks of text.

Here is a Failure created by translating into a less-common language (Zulu) and back into English:

```
Google translate run 20 May 2019
   The input text is a headline from the Daily Maverick on the same date

[Input English]

Britain's slow suicide as the Brexit Euro-chickens come home to roost

[Translate into Chinese characters and back into English]

With the arrival of Brexit European chickens, the suicide rate in the UK is slow

[Translate into pinyin]

Suízhe yīngguó tuō ōu ōuzhōu jī de dàolái, yīngguó de zìshā sùdù huǎnmàn

[Put this back into Google.  Google autodetects it as Maori.
Explicitly selecting Chinese causes translate to give up and copy
input to output unchanged.   OK, translate it from Maori...]

Find out how to get rid of your ice cream gases, and how to get rid of zoo
```

A serious issue with current AI systems is that they have no idea when they are confused. A human is more likely to understand that they don't know the answer, and say so rather than bluffing.

Deep systems

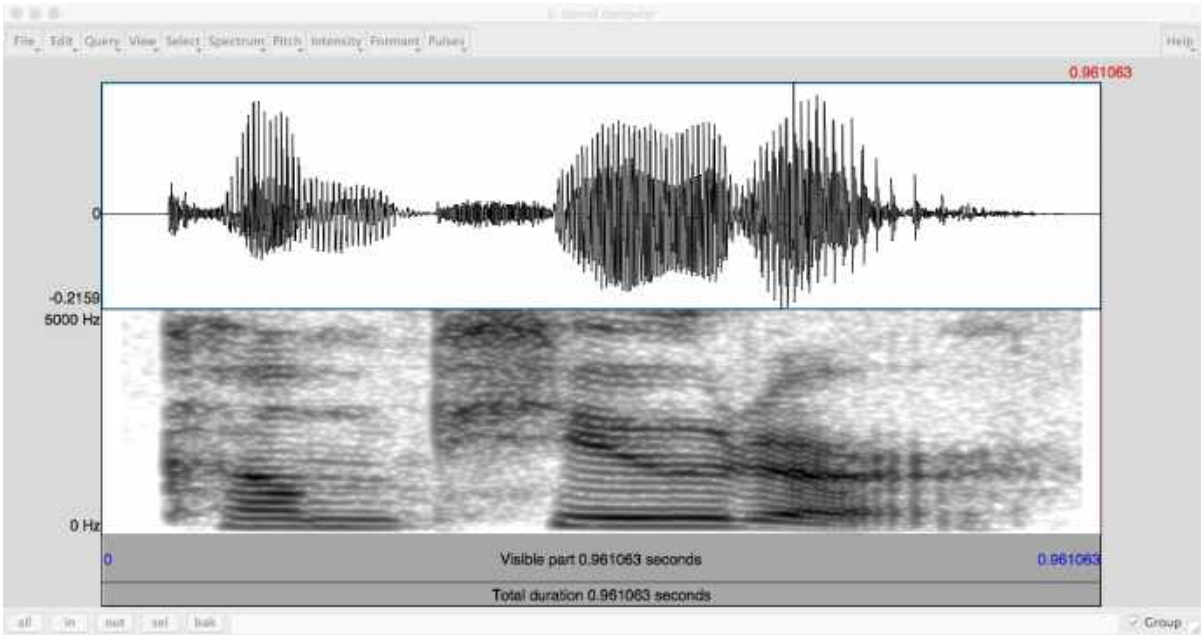
Deeper processing has usually been limited to focused domains, for which we can easily convert text or speech into a representation for underlying meaning. A simple example would when Google returns structured information for a restaurant, e.g. its address and opening hours. In this case, the hard part of the task is extracting the structured information from web pages, because people present this information in many different formats. It's typically much easier to generate natural language text, or even speech, from structured representations.

End-to-end systems have been built for some highly-structured customer service tasks, e.g. speech-based airline reservations (a task that was popular historically in AI) or utility company help lines. Google had a recent demo of a computer assistant making a restaurant reservation by phone. There have also been demo systems for tutoring, e.g. physics.

Many of these systems depend on the fact that people will adapt their language when they know they are talking to a computer. It's often critical that the computer's responses not sound too sophisticated, so that the human doesn't over-estimate its capabilities. (This may have happened to you when attempting a foreign language, if you are good at mimicking pronunciation.) These systems often steer humans towards using certain words. E.g. when the computer suggests several alternatives ("Do you want to report an outage? pay a bill?") the human is likely to use similar words in their reponse.

If the AI system is capable of relatively fluent output (possible in a limited domain), the user may mistakenly assume it understands more than it does. Many of us have had this problem when travelling in a foreign country: a simple question pronounced will may be answered with a flood of fluent language that can't be understood. Designers of AI systems often include cues to make sure humans understand that they're dealing with a computer, so that they will make appropriate allowances for what it's capable of doing.

A spectrogram



Overview of processing pipeline

The NLP processing pipeline looks roughly like this:

- Speech
- Low-level text processing (e.g. forming words, finding morphemes)
- Mid-level processing (e.g. part of speech, parsing)
- Semantics (meaning, dialog structure)

Text-only processing omits the speech step. Shallow systems operate directly on the low-level or mid-level representations, omitting semantics.

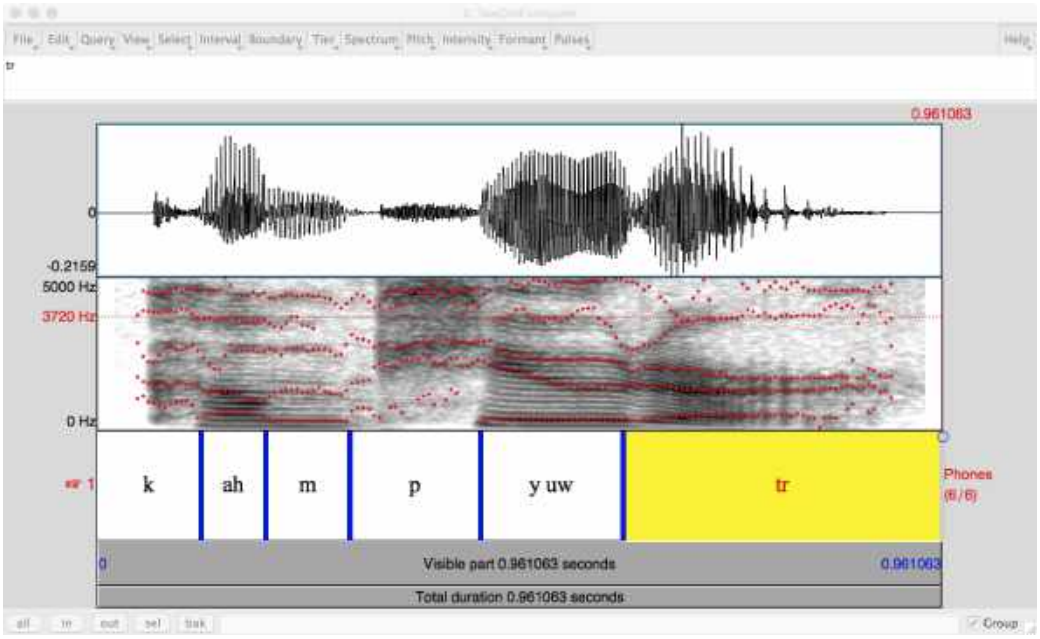
Some useful definitions

- phone: unit in a high-level transcription of speech
- morpheme: smallest meaningful chunk (e.g. "talk" or "-ing")
- word: smallest chunk that's said as a unit (no pauses) and/or written without internal spaces e.g. "talking"

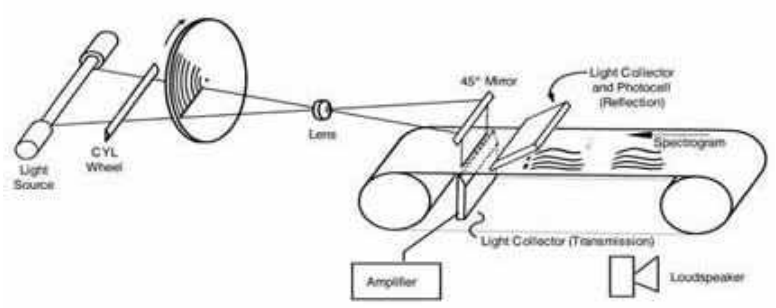
Speech recognition/generation

The first "speech recognition" layer of processing turns speech (e.g. the waveform at the top of this page) into a sequence of phones (basic units of sound). We first process the waveform to bring out features important for recognition. The spectogram (bottom part of the figure) shows the energy at each frequency as a function of time, creating a human-friendly picture of the signal. Computer programs use something similar, but further processed into a small set of numerical features.

Each phone shows up in characteristic ways in the spectrogram. Stop consonants (t, p, ...) create a brief silence. Fricatives (e.g. s) have high-frequency random noise. Vowels and semi-vowels (e.g. n, r) have strong bands ("formants") at a small set of frequencies. The location of these bands and their changes over time tell you which vowel or semi-vowel it is.



Recognition is harder than generation. However, generation is not as easy as you might think. Very early speech synthesis (1940's, Haskins Lab) was done by painting pictures of formants onto a sheet of plastic and using a physical transducer called the Pattern Playback to convert them into sounds.



Synthesis moved to computers once these became usable. One technique (formant synthesis) reconstructs sounds from formats and/or a model of the vocal tract. Another technique (concatenative synthesis) splices together short samples of actual speech, to form extended utterances.

- A synthesizer from 1961 sings a song (1961, John Kelly and Louis Gerstman, sound file from Chilin Shih)
- Speech generation still sounds odd, especially the timing/intonation. (e.g. Stephen Hawking's infamous voice synthesizer) but also mispronunciations due to limited dictionaries.

The voice of Hal singing "Bicycle built for two" in the movie "2001" (1968) was actually a fake, rather than real synthesizer output from the time.

Making words

Models of human language understanding typically assume that a first stage that produces a reasonably accurate sequence of phones. The sequence of phones must then be segmented into a sequence of words by a "word segmentation" algorithm. The process might look like this, where # marks a literal pause in speech (e.g. the speaker taking a breath).

> INPUT: ohlThikidsinner # ahrpiyp@lThA?HAvkids # ohrThADurHAviynqkids
> OUTPUT: ohl Thi kids inner # ahr piyp@l ThA? HAv kids # ohr ThADur HAviynq kids

In standard written English, this would be "all the kids in there # are people that have kids # or that are having kids".

With current technology, the phone-level transcripts aren't accurate enough to do things this way. Instead, the final choice for each phone is determined by a combination of bottom-up signal processing and a statistical model of words and short word sequences. The methods are similar to the HMMs that we'll see soon for part of speech tagging.

If you have seen live transcripts of speeches (e.g. Obama's 2018 speech at UIUC) you'll know that there is still a significant error rate. This seems to be true whether they are produced by computers, humans, or (most likely) the two in collaboration. Highly-accurate systems (e.g. dictation) depend on knowing a lot about the speaker (how they normally pronounce words) and/or their acoustic environment (e.g. background noise) and/or what they will be talking about

A major challenge for speech recognition is that actual pronunciations of words frequently differ from their dictionary pronunciations. For example,

- The words "can't" and "can" appear as [K AE N] and [K AE N T] in CMU's standard recognizer dictionary. But, in some dialects, "can" is pronounced with a different vowel [K EH N].
- In the segmentation example above, notice that "in there" has fused together into "inner" with the "th" sound changing to become like the "n" preceding it.

This kind of "phonological" change is part of the natural evolution of languages and dialects. Spelling conventions often reflect an older version of the language. Also, spelling systems often try to strike a middle ground among diverse dialects, so that written documents can be shared. The same is true, albeit to a lesser extent, of the pronunciations in dictionaries. These sound changes make it much harder to turn sequences of phones into words.

As we saw in the Naive Bayes lectures, different sorts of cleaning are required to make a clean stream of words from typical written text. We may also need to do similar post-processing to a speech recognizer's output. For example, the recognizer may be configured to transcribe into a sequence of short words (e.g. "base", "ball", "computer", "science") even when they form a tight meaning unit ("baseball" or "computer science"). Recognizers also faithfully reproduce disfluencies from the speaker (e.g. "um") and backchannels (e.g. "uh huh") from the listener. We may wish to remove these before later processing.

Boaty McBoatface


Boaty McBoatface (from the BBC)

Finding morphemes

So, now, let's assume that we have a clean sequence of words. By "word," I mean a chunk of a size that's convenient for later algorithms (e.g. parsing, translation). This might be the output of a speech recognizer. Or, because many natural language systems don't start with speech, the stream of words might be (cleaned-up) written text.

These words then need to be divided into morphemes by a "morphology" algorithm. Words in some languages can get extremely long (e.g. Turkish), making it hard to do further processing unless they are (at least partly) subdivided into morphemes. For example:

> unanswerable --> un-answer-able
> preconditions --> pre-condition-s

Or, consider the following well-known Chinese word:


Zhōng + guó

In modern Chinese, this is one word ("China"). And our AI system would likely be able to accumulate enough information about China by treating it as one two-syllable item. But it is historically (and in the writing system) made from two morphemes ("middle" and "country"). It is what's called a "transparent compound," i.e. a speaker of the language would be aware of the separate pieces. This knowledge of the internal word structure would allow a human to guess that a less common word ending in "guo" might also be the name of a country.

Sometimes related words are formed by processes other than concatenation, e.g. the internal vowel change in English "foot" vs. "feet". When this is common in a language, we may wish to use a more abstract feature-based representation, e.g.

> foot --> foot+SINGULAR
> feet --> foot+PLURAL

Sometimes the appropriate division depends on the application. So the English possessive ending "'s" is traditionally written as part of the preceding word. However, it is often convenient to split it off as a separate word in later processing.

POS tagging

In order to group words into larger units (e.g. prepositional phrases), the first step is typically to assign a "part of speech" (POS) tag to each word. Here some sample text from the Brown corpus, which contains very clean written text.

> Northern liberals are the chief supporters of civil rights and of integration. They have also led the nation in the direction of a welfare state.

Here's the tagged version. For example, "liberals" is an NNS which is a plural noun. "Northern" is an adjective (JJ). Tag sets need to distinguish major types of words (e.g. nouns vs. adjectives) and major variations e.g. singular vs plural nouns, present vs. past tense verbs. There are also some ad-hoc tags key function words, such as HV for "have," and punctuation (e.g. period).

> Northern/jj liberals/nns are/ber the/at chief/jjs supporters/nns of/in civil/jj rights/nns and/cc of/in integration/nn ./. They/ppss have/hv also/rb led/vbn the/at nation/nn in/in the/at direction/nn of/in a/at welfare/nn state/nn ./.

Most words have only one common part of speech. So we can get up to about 91% accuracy using a simple "baseline" algoirthm that always guesses the most common part of speech for each word. However, some words have more than one possible tag, e.g. "liberal" could be either a noun or an adjective. To get these right, we can use algorithms like hidden Markov models (next lecture). On clean text, a highly tuned POS tagger can get about 97% accuracy. In other words, POS taggers are quite reliable and mostly used as a stable starting place for further analysis.

Shallow systems

Many useful systems can be built using only a simple local analysis of the word stream, e.g. words, word bigrams, morphemes, POS tags, perhaps a shallow parse (see below). These algorithms typically make a "Markov assumption," i.e. assume that only the last few items matter to the next decision. E.g.

- Deciding whether to insert a word boundary? Look at the last 5-7 characters.
- Deciding what POS tag to put on the next word? Look at the last 1-3 words.

Specific techniques include finite-state automata, hidden Markov models (HMMs), and recurrent neural nets (RNNs). We'll see hidden Markov models later in this course.

One very old type of shallow system is spelling correction. Useful upgrades (which often work) include detection of grammar errors, adding vowels or diacritics in language (e.g. Arabic) where they are often omitted. Cutting-edge research problems involve automatic scoring of essay answers on standardized tests. Speech recognition has been used for evaluating English fluency and children learning to read (e.g. aloud). Success depends on very strong expectations about what the person will say.

Translation is often done by shallow algorithms. To learn how to do translation, we might align pairs of sentences in different languages, matching corresponding words.

> English: 18-year-olds can't buy alcohol.
> French: Les 18 ans ne peuvent pas acheter d'alcool

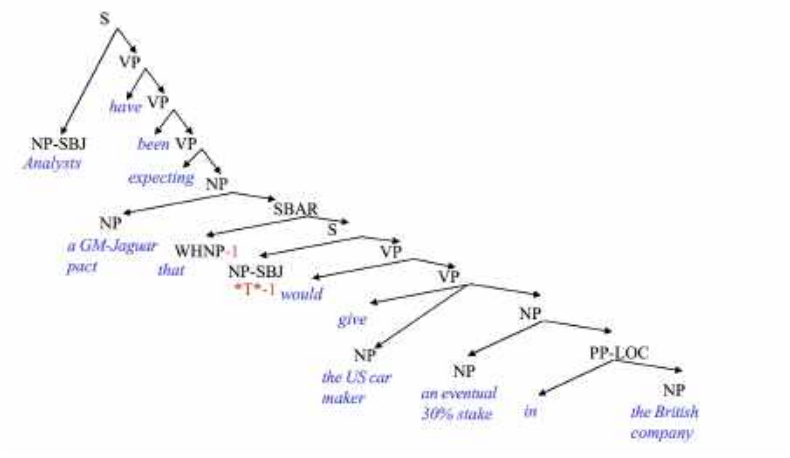
	18	year	olds		can	't	buy		alcohol
Les	18	ans		ne	peuvent	pas	acheter	d'	alcool

Notice that some words don't have matches in the other language. For other pairs of languages, there could be radical changes in word order.

A corpus of matched sentence pairs can be used to build translation dictionaries (for phrases as well as words) and extract general knowledge about changes in word order.

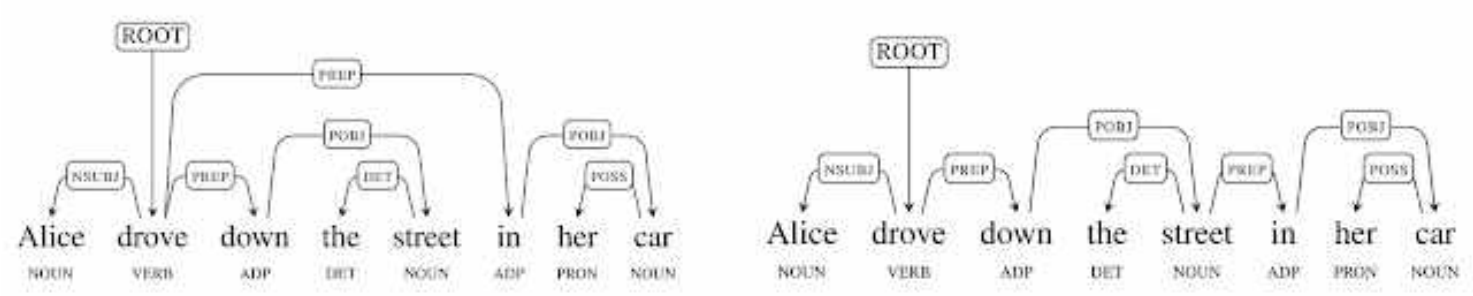
Parsing

One we have POS tags for the words, we can assemble the words into a parse tree. There's many styles from building parse trees. Here is a **constituency tree** from the Penn treebank (from Mitch Marcus).



Penn treebank style parse tree (from Mitch Marcus)

An alternative is a **dependency tree** like the following ones below from Google labs. A fairly recent parser from them is called "Parsey McParseface," after the UK's Boaty McBoatface shown above.



In this example, the lefthand tree shows the correct attachment for "in her car," i.e. modifying "drove." The righthand tree shows an interpretation in which the street is in the car.

Linguists (computational and otherwise) have extended fights about the best way to draw these trees. However, the information encoded is always fairly similar and primarily involves grouping together words that form coherent phrases e.g. "a welfare state." It's fairly similar to parsing computer programming languages, except that programming languages have been designed to make parsing easy.

A "shallow parser" builds only the lowest parts of such a tree. So it might group words into noun phrases, prepositional phrases, and complex verbs (e.g. "is walking"). Extracting these phrases is much easier than building the whole parse tree, and can be extremely useful for building shallow systems.

Like low-level algorithms, parsers often make their decisions using only a small amount of prior (and perhaps lookahead) context. However, "one item" of context might be a whole chunk of the parse tree. For example, "the young lady" might be considered a single unit. Parsing algorithms must typically consider a wide space of options, e.g. many options for a partly-built tree. So they typically use beam search, i.e. keep only a fixed number of hypotheses with the best rating. Newer methods also try to share tree structure among competing alternatives (like dynamic programming) so they can store more hypotheses and avoid duplicative work.

Parsers fall into three categories

- Unlexicalized: use only the POS tags to build the tree.
- Class-based: beyond the part of speech, identify the general type of object or action (e.g. a person vs. a vehicle)
- Lexicalized: also include some information about word identity/meaning

The value of lexical information is illustrated by sentences like this, in which changing the noun phrase changes what the prepositional phrase modifies:

> She was going down a street ..
>
> > in her truck. (modifies going)
> > in her new outfit. (modifies the subject)
> > in South Chicago. (modifies street)

The best parsers are lexicalized (accuracies up to 94% from Google's "Parsey McParseface" parser). But it's not clear how much information to include about the words and their meanings. For example, should "car" always behaves like "truck"? More detailed information helps make decisions (esp. attachment) but requires more training data.

More on tag sets

Notice that the tag set for the Brown corpus was somewhat specialized for English, in which forms of "have" and "to be" play a critical syntactic role. Tag sets for other languages would need some of the same tags (e.g. for nouns) but also categories for types of function words that English doesn't use. For example, a tag set for Chinese or Mayan would need a tag for numeral classifiers, which are words that go with numbers (e.g. "three") to indicate the approximate type of object being enumerated (e.g. "table" might require a classifier for big flat objects). It's not clear whether it's better to have specialized tag sets for specific languages or one universal tag sets that includes major functional categories for all languages.

Tag sets vary in size, depending on the theoretical biases of the folks making the annotated data. Smaller sets of labels convey only basic information about word type. Larger sets include information about what role the word plays in the surrounding context. Sample sizes

- Penn treebank 36
- Brown corpus 87
- "universal" 12

Conversational spoken language also includes features not found in written language. In the example below (from the Switchboard corpus), you can see the filled pause "uh", also a broken off word "t-". Also, notice that the first sentence is broken up by a paranthetical comment ("you know") and the third sentence trails off at the end. Such features make spoken conversation harder to parse than written text.

> I 'd be very very careful and , uh , you know , checking them out . Uh , our , had t- , place my mother in a nursing home . She had a rather massive stroke about , uh, about ---
>
> I/PRP 'd/MD be/VB very/RB very/RB careful/JJ and/CC ./, uh/UH ./, you/PRP know/VBP ./, checking/VBG them/PRP out/RP ./. Uh/UH ./, our/PRP$ ./, had/VBD t-/TO ./, place/VB my/PRP$ mother/NN in/IN a/DT nursing/NN home/NN ./. She/PRP had/VBD a/DT rather/RB massive/JJ stroke/NN about/RB ./, uh/UH ./, about/RB --/:

Semantics

Representation of meaning is less well understood. In modern systems, meaning of individual word stems (e.g. "cat" or "walk") is based on their observed context. We'll see details later in the term. But these meanings are not tied, or are only weakly tied, to the physical world. Methods for combining the meanings of individual words into a single meaning (e.g. "the red cat") are similarly fragile. Very few systems attempt to understand sophisticated constructions involving quantifiers ("How many arrows didn't hit the target?") or relative clauses. (An example of a relative clause is "that would give ..." in the Penn treebank parse example above.)

A specific issue for even simple applications is that negation is easy for humans but difficult for computers. For example, a google query for "Africa, not francophone" returns information on French-speaking parts of Africa. And this circular translation example shows Google making the following conversion which reverses the polarity of the advice.

- Input: "so oops on the trying it after 3 hours."
- Output: "Try to try it after 3 hours."

Applications that can work well with limited understanding include

- grouping documents into topics, dividing documents at places where they change topic
- sentiment analysis: does the writer like this movie or this restaurant?

Three types of shallow semantic analysis have proved useful and almost within current capabilities:

- word classes: which words are similar to one another (e.g. people vs. vegetables)?
- word sense disambiguation: which was the intended reading of a word with multiple meanings (e.g. "bank")?
- semantic role labelling: we know that a noun phrase X relates to a verb Y. Is X the subject/actor? the object that the action was done to? a tool used to help with the action?
- co-reference resolution (see below)

Semantic role labelling involves deciding how the main noun phrases in a clause relate to the verb. For example, in "John drove the car," "John" is the subject/agent and "the car" is the object being driven. These relationships aren't always object, e.g. who is eating who in a "truck eating bridge"? (Google it.)

Right now, the most popular representation for word classes is a "word embedding." Word embeddings give each word a unique location in a high dimensional Euclidean space, set up so that similar words are close together. We'll see the details later. A popular algorithm is word2vec.

Running text contains a number of "named entities," i.e. nouns, pronouns, and noun phrases referring to people, organizations, and places. Co-reference resolution tries to identify which named entities refer to the same thing. For example, in this text from Wikipedia, we have identified three entities as referring to Michelle Obama, two as Barack Obama, and three as places that are neither of them. One source of difficulty is items such as the last "Obama," which looks superficially like it could be either of them.

> [Michelle LaVaughn Robinson Obama] (born January 17, 1964) is an American lawyer, university administrator, and writer who served as the [First Lady of the United States] from 2009 to 2017. She is married to the [44th U.S. President], [Barack Obama], and was the first African-American First Lady. Raised on the South Side of [Chicago, Illinois], [Obama] is a graduate of [Princeton University] and [Harvard Law School].

Dialog coherency

From a very young age, people have a strong ability to manage interactions over a long period of time. E.g. we can keep talking about a coherent theme (e.g. politics) for an entire evening. We can ask follow-on questions to build up a mental model in which all the pieces fit together. We expect all the pieces to fit together when reading a novel. Even the best modern AI systems cannot yet do this.

One specific task is "dialog coherency." For example, suppose you ask Google home a question like "How many calories in a banana?" You might like to follow on with "How about an orange?" But most of these systems process each query separately. The exceptions can keep only a very short theory of context or (e.g. customer service systems) work in a very restricted domain. A more fun example of maintaining coherency in a restricted domain was the ILEX system for providing information about museum exhibits. (original paper link)

The recent neural net system GPT-3 has a strong ability to maintain local coherency in its generated text and dialogs. However, this comes at a cost. Because it has no actual model of the world or its own beliefs, it can be led astray by leading questions.

CS 440/ECE 448
Margaret Fleck

09 Hidden Markov Models

a real-world trellis (supporting a passionfruit vine)

Hidden Markov Models are used in a wide range of applications. They are similar to Bayes Nets (both being types of "Graphical Models"). We'll see them in the context Part of Speech tagging. Uses in other applications (e.g. speech) look very similar.

Part of Speech tagging

We saw part-of-speech (POS) tagging in a previous lecture. Recall that we start with a sequence of words. The tagging algorithm needs to decorate each word with a part-of-speech tag. Here's an example:

> INPUT: Northern liberals are the chief supporters of civil rights and of integration. They have also led the nation in the direction of a welfare state.
>
> OUTPUT: Northern/jj liberals/nns are/ber the/at chief/jjs supporters/nns of/in civil/jj rights/nns and/cc of/in integration/nn ./. They/ppss have/hv also/rb led/vbn the/at nation/nn in/in the/at direction/nn of/in a/at welfare/nn state/nn ./.

Identifying a word's part of speech provides strong constraints on how later processing should treat the word. Or, viewed another way, the POS tags group together words that should be treated similarly in later processing. For example, in building parse trees, the POS tags povide direct information about the local syntactic structure, reducing the options that the parse must choose from.

Many early algorithms exploit POS tags directly (without building a parse tree) because the POS tag can help resolve ambiguities. For example, when translating, the noun "set" (i.e. collection) would be translated very differently from the verb "set" (i.e. put). The part of speech can distinguish words that are written the same but pronounced differently, such as:

- OBject (noun) vs. obJECT (verb)
- does (modal verb vs. plural noun)
- read (past vs. present tense)

Tag sets

POS tag sets vary in size, e.g. 87 in the Brown corpus down to 12 in the "universal" tag set from Petrov et all 2012 (Google research). Even when the number of tags is similar, different designers may choose to call out different linguistic distinctions.

our MP	"Universal"	
NOUN	Noun	noun
PRON	Pronoun	pronoun
VERB	Verb	verb
MODAL		modal verb
ADJ	Adj	adjective
DET	Det	determiner or article
PERIOD	. (just a period)	end of sentence punctuation
PUNCT		other punctuation
NUM	Num	number
IN	Adp	preposition or postposition
PART		particle e.g. after verb, looks like a preposition)
TO		infinitive marker
ADV	Adv	adverb
	Prt	sentence particle (e.g. "not")
UH	[not clear what tag]	filler, exclamation
CONJ	Conj	conjunction
X	X	other

Here's a more complex set of 36 tags from the Penn Treebank tagset . The larger tag sets are a mixed blessing. On the one hand, the tagger must make finer distinctions when it chooses the tag. On the other hand, a more specific tag provides more information for tagging later words and also for subsequent processing (e.g. parsing).

Tags and words

Each word has several possible tags, with different probabilities.

- in principle, if we had perfect knowledge of English
- as seen in our training set

For example, "can" appears using in several roles:

- Monica can swim. (modal verb)
- A can of beans. (noun)
- I will can the beans. (verb)

Some other words with multiple uses

- cover: noun, verb, adjective (cover crop)

- over: preposition, noun (e.g. cricket)
- pot: noun, verb

Consider the sentence "Heat oil in a large pot." The correct tag sequence is "verb noun prep det adj noun". Here's the tags seen when training on the Brown corpus.

Word	POS listings in Brown		
heat	noun/89	**verb/5**	
oil	**noun/87**		
in	**prep/20731**	noun/1	adv/462
a	**det/22943**	noun/50	noun-proper/30
large	**adj/354**	noun/2	adv/5
pot	**noun/27**		

(from Mitch Marcus)

Notice that

- An individual word may have more than one plausible tag
- The most commonly seen tag may not be correct choice for analyzing any particular input.
- A word may have possible tags that weren't seen in our training input.

An example of that last would be "over" used as a noun (e.g. in cricket). That is fairly common in British text but might never occur in a training corpus from the US.

Baseline algorithm

Most testing starts with a "baseline" algorithm, i.e. a simple method that does do the job, but typically not very well. In this case, our baseline algorithm might pick the most common tag for each word, ignoring context. In the table above, the most common tag for each word is shown in bold: it made one error out of six words.

The test/development data will typically contain some words that weren't seen in the training data. The baseline algorithm guesses that these are nouns, since that's the most common tag for infrequent words. An algorithm can figure this out by examining new words that appear in the development set, or by examining the set of words that occur only once in the training data ("hapax") words.

On larger tests, this baseline algorithm has an accuracy around 91%. It's essential for a proposed new algorithm to beat the baseline. In this case, that's not so easy because the baseline performance is unusually high.

Formulating the estimation problem

Suppose that W is our input word sequence (e.g. a sentence) and T is an input tag sequence. That is

$$W = w_1, w_2, \ldots w_n$$
$$T = t_1, t_2, \ldots t_n$$

Our goal is to find a tag sequence T that maximizes P(T | W). Using Bayes rule, this is equivalent to maximizing $P(W \mid T) * P(T)/P(W)$ And this is equivalent to maximizing $P(W \mid T) * P(T)$ because P(W) doesn't depend on T.

So we need to estimate P(W | T) and P(T) for one fixed word sequence W but for all choices for the tag sequenceT.

What can we realistically estimate?

Obviously we can't directly measure the probability of an entire sentence given a similarly long tag sequence. Using (say) the Brown corpus, we can get reasonable estimates for

- individual word frequencies
- tag usage for each word
- tag bigrams

If we have enough tagged data, we can also estimate tag trigrams. Tagged data is hard to get in quantity, because human intervention is needed for checking and correction.

Markov assumptions

To simplify the problem, we make "Markov assumptions." A Markov assumption is that the value of interest depends only on a short window of context. For example, we might assume that the probability of the next word depends only on the two previous words. You'll often see "Markov assumption" defined to allow only a single item of context. However, short finite contexts can easily be simulated using one-item contexts.

For POS tagging, we make two Markov assumptions. For each position k:

$P(w_k)$ depends only on $P(t_k)$
$P(t_k)$ depends only on $P(t_{k-1})$

Therefore

$$P(W \mid T) = \prod_{i=1}^{n} P(w_i \mid t_i)$$
$$P(T) = P(t_1 | START) * \prod_{i=1}^{n} P(t_k \mid t_{k-1})$$

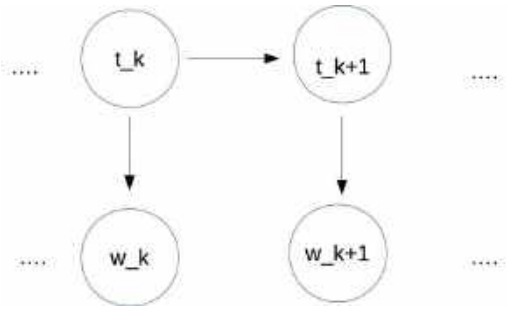
where $P(t_1 | START)$ is the probability of tag t_1 at the start of a sentence.

Folding this all together, we're finding the T that maximizes

$$P(T \mid W)$$
$$\propto P(W \mid T) * P(T)$$
$$= \prod_{i=1}^{n} P(w_i \mid t_i) \quad * \quad P(t_1 \mid \text{START}) \quad * \quad \prod_{k=2}^{n} P(t_k \mid t_{k-1})$$

HMM picture

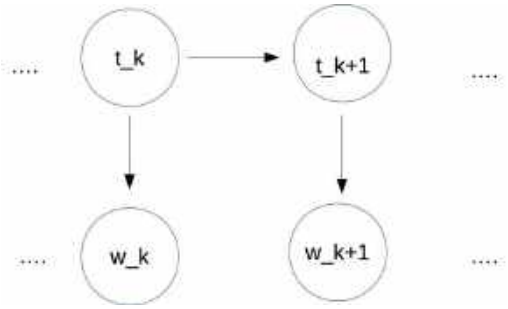
Like Bayes nets, HMMs are a type of graphical model and so we can use similar pictures to visualize the statistical dependencies:



In this case, we have a sequence of tags at the top. Each tag depends only on the previous tag. Each word (bottom row) depends only on the corresonding tag.

Recap

In an HMM, we have a sequence of tags at the top. Each tag depends only on the previous tag. Each word (bottom row) depends only on the corresonding tag.



At each time position k, we assume that

$P(w_k)$ depends only on $P(t_k)$
$P(t_k)$ depends only on $P(t_{k-1})$)

Given an input word sequence W, our goal is to find the tag sequence T that maximizes

$$P(T \mid W)$$
$$\propto P(W \mid T) * P(T)$$
$$= \prod_{i=1}^{n} P(w_i \mid t_i) \quad * \quad P(t_1 \mid \text{START}) \quad * \quad \prod_{k=2}^{n} P(t_k \mid t_{k-1})$$

Let's label our probabilities as P_S, P_E, and P_T to make their roles more explicit. Using this notation, we're trying to maximize

$$\prod_{i=1}^{n} P_E(w_i \mid t_i) \quad * \quad P_S(t_1) \quad * \quad \prod_{k=2}^{n} P_T(t_k \mid t_{k-1})$$

So we need to estimate three types of parameters:

- $P_S(t_1)$ initial probabilities
- $P_T(t_k|t_{k-1})$ transition probabilities
- $P_E(w_i|t_i)$ emission probabilities

Toy model

For simplicity, let's assume that we have only three tags: Noun, Verb, and Determiner (Det). Our input will contain sentences that use only those types of words, e.g.

```
Det Noun     Verb    Det Noun   Noun
The student  ate     the ramen  noodles

Det Noun   Noun  Verb    Det Noun
The maniac rider crashed his bike.
```

We can look up our initial probabilities, i.e. the probability of various values for $P_S(t_1)$ in a table like this.

tag	$P_S(tag)$
Verb	0.01
Noun	0.18
Det	0.81

We can estimate these by counting how often each tag occurs at the start of a sentence in our training data.

Transition probabilities

In the (correct) tag sequence for this sentence, the tags t_1 and t_5 have the same value Det. So the possibilities for the following tags (t_2 and t_6) should be similar. In POS tagging, we assume that $P_T(t_{k+1} \mid t_k)$ depends only on the identities of t_{k+1} and t_k, not on their time position k. (This is a common assumption for Markov models.)

This is not an entirely safe assumption. Noun phrases early in a sentence are systematically different from those toward the end. The early ones are more likely to be pronouns or short noun phrases (e.g. "the bike"). The later ones more likely to contain nouns that are new to the conversation and, thus, tend to be more elaborate (e.g. "John's fancy new racing bike"). However, abstracting away from time position is a very useful simplifying assumption which allows us to estimate parameters with realistic amounts of training data.

So, suppose we need to find a transition probability $P_T(t_k|t_{k-1})$ for two tags t_{k-1} and t_k in our tag sequence. We consider only the identities of the two tags, e.g. t_{k-1} might be DET and t_k might be NOUN. And we look up the pair of tags in a table like this

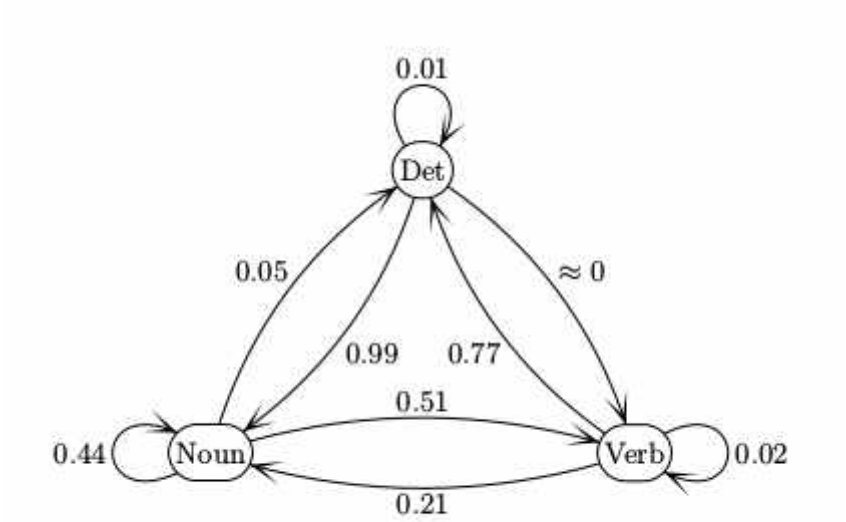
	Next Tag		
Previous tag	**Verb**	**Noun**	**Det**
Verb	0.02	0.21	0.77
Noun	0.51	0.44	0.05
Det	≈ 0	0.99	0.01

We can estimate these by counting how often a 2-tag sequence (e.g. Det Noun) occurs and dividing by how often the first tag (e.g. Det) occurs.

Notice the near-zero probability of a transition from Det to Verb. If this is actually zero in our training data, do we want to use smoothing to fill in a non-zero number? Yes, because even "impossible" transitions may occur in real data due to typos or (with larger tagsets) very rare tags. Laplace smoothing should work well.

Finite-state automaton

We can view the transitions between tags as a finite-state automaton (FSA). The numbers on each arc is the probability of that transition happening.



These are conceptually the same as the FSA's found in CS theory courses. But beware of small differences in conventions.

- You can start in any state (probability given by the initial probability table).
- A transition always exists between every pair of states, but each transition has an associated probability of happening.
- Output (words) is generated when you enter a state, not on the transition between states.

Emission probabilities

Our final set of parameters are the emission probabilities, i.e. $P_E(w_k|t_k)$ for each position k in our tag sequence. Suppose we have a very small vocabulary of nouns and verbs to go with our toy model. Then we might have probabilities like

```
Verb   played 0.01
       ate    0.05
       wrote  0.05
       saw    0.03
       said   0.03
       tree   0.001

Noun   boy    0.01
       girl   0.01
       cat    0.02
       ball   0.01
       tree   0.01
       saw    0.01

Det    a      0.4
       the    0.4
       some   0.02
       many   0.02
```

Because real vocabularies are very large, it's common to see new words in test data. So it's essential to smooth, i.e. reserve some of each tag's probability for

- new words
- familiar words seen with a new tag

However, the details may need to depend on the type of tag. Tags divide (approximately) into two types

- open class, aka content words (e.g. verbs, nouns, adjectives)
- closed class, aka function words (e.g. prepositions, determiners)

It's very common to see new open-class words. In English, open-class tags for existing words are also likely, because nouns are often repurposed as verbs, and vice versa. E.g. the proper name MacGyver spawned a verb, based on the improvisational skills of the TV character with that name. So open class tags need to reserve significant probability mass for unseen words, e.g. larger Laplace smoothing constant. It's much less common to encounter new closed-class words, e.g. a new conjunction. So these tags should have a smaller Laplace smoothing constant. The amount of probability mass to reserve for each tag can be estimated by looking at the tags of "hapax" words (words that occur only once) in our training data.

Decoding (Viterbi algorithm)

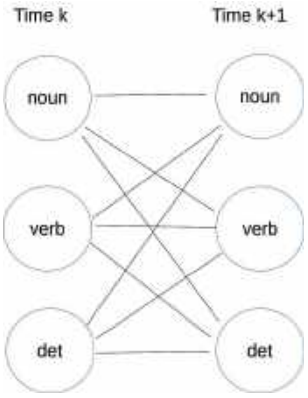
Once we have estimated the model parameters, the Viterbi algorithm finds the highest-probability tag sequence for each input word sequence. Suppose that our input word sequence is $w_1, \ldots w_n$.

The basic data structure is an n by m array (v), where n is the length of the input word sequence and m is the number of different (unique) tags. Each cell (k,t) in the array contains the probability v(k,t) of the best sequence of tags for $w_1 \ldots w_k$ that ends with tag t. We also store b(k,t), which is the previous tag in that best sequence. This data structure is called a "trellis."

The Viterbi algorithm fills the trellis from left to right, as follows:

> **Initialization:** We fill the first column using the initial probabilities. Specifically, the cell for tag t gets the value $v(1, t) = P_S(t) * P_E(w_1 \mid t)$.
>
> **Moving forwards in time:** Use the values in column k to fill column k+1. Specifically
>
> > For each tag tag_B
> >
> > $$v(k+1, tag_B) = \max_{tag_A} \; v(k, tag_A) * P_T(tag_B \mid tag_A) * P_E(w_{k+1} \mid tag_B)$$
> > $$b(k+1, tag_B) = \operatorname{argmax}_{tag_A} \; v(k, tag_A) * P_T(tag_B \mid tag_A) * P_E(w_{k+1} \mid tag_B)$$
>
> That is, we compute $v(k, tag_A) * P_T(tag_B \mid tag_A) * P_E(w_{k+1} \mid tag_B)$ for all possible tags tag_A. The maximum value goes into trellis cell $v(k+1, tag_B)$ and the corresponding value of tag_A is stored in $b(k+1, tag_B)$.
>
> **Finishing:** When we've filled the entire trellis, pick the best tag B in the final (time=n) column. Trace backwards from B, using the values in b, to produce the output tag sequence.

Here's a picture of one section of the trellis, drawn as a graph. A path ending in some specific tag at time k might continue to any one of the possible tags at time k+1. So each cell in column k is connected to all cells in column k+1. The name "trellis" comes from this dense pattern of connections between each timestep and the next.



It's like a Maze

The Viterbi algorithm depends on this mystery rule for computing a value at time k+1 from a value at time k.

$$v(k+1, tag_b) \;=\; v(k, tag_a) * P_T(tag_b \mid tag_a) * P_E(w_{k+1} \mid tag_b)$$

What is it?

This about searching the trellis like a maze, but consistently moving from left to right (i.e. following the progression of time). The righthand side of our equation is the product of two terms:

- $v(k, tag_a)$ is the total cost to reach state (k, tag_a)
- $P_T(tag_b \mid tag_a) * P_E(w_{k+1} \mid tag_b)$ is the cost to get from (k, tag_a) to (k_{k+1}, tag_b)

As we move through the trellis, we're multiplying costs, where we would add them in a normal maze. But the rest of the method is similar. And we produce the final output path by tracing backwards, just as we would in a maze search algorithm.

More details

The probabilities in the Viterbi algorithm can get very small, creating a real possibility of underflow. So the actual implementation needs to convert to logs and replace multiplication with addition. After the switch to addition, the Viterbi computation looks even more like edit distance or maze search. That is, each cell contains the cost of the best path from the start to our current timestep. As we move to the next timestep, we're adding some additional cost to the path. However, for Viterbi we prefer larger, not smaller, values.

The asymptotic running time for this algorithm is $O(m^2 n)$ where n is the number of input words and m is the number of different tags. In typical use, n would be very large (the length of our test set) and m fairly small (e.g. 36 tags in the Penn treebank). Moreover, the computation is quite simple. So Viterbi ends up with good constants and therefore a fast running time in practice.

Extensions

The Viterbi algorithm can be extended in several ways. First, high accuracy taggers typically use two tags of context to predict each new tag. This increases the height of our trellis from m to m^2. So the full trellis could become large and expensive to computer. These methods typically use beam search, i.e. keep only the most promising trellis states for each timestep.

So far, we've been assuming that all unseen words should be treated as unanalyzable unknown objects. However, the form of a word often contains strong cues to its part of speech. For example, English words ending in "-ly"

are typically adverbs. Words ending in "-ing" are either nouns or verbs. We can use these patterns to make more accurate guesses for emission probabilities of unseen words.

Finally, suppose that we have only very limited amounts of tagged training data, so that our estimated model parameters are not very accurate. We can use untagged training data to refine the model parameters, using a technique called the forward-backward (or Baum-Welch) algorithm. This is an application of a general method called "Expectation Maximization" (EM) and it iteratively tunes the HMM's parameters so as to improve the computed likelihood of the training data.

Here is an interesting example from 1980 of how an HMM can automatically learn sound classes: R. L. Cave and L. P. Neuwirth, "Hidden Markov models for English."

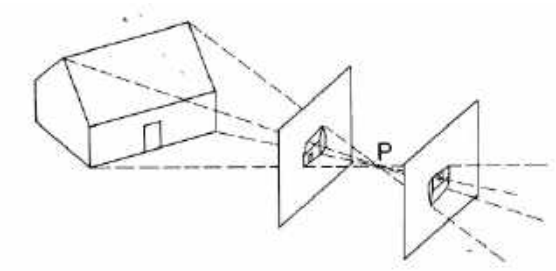
CS 440/ECE 448
Margaret Fleck

10 Computer Vision

Computer vision algorithms relate image data to high-level descriptions. Although the techniques are generally similar to those in natural language, the objects being manipulated are in 2D or 3D.

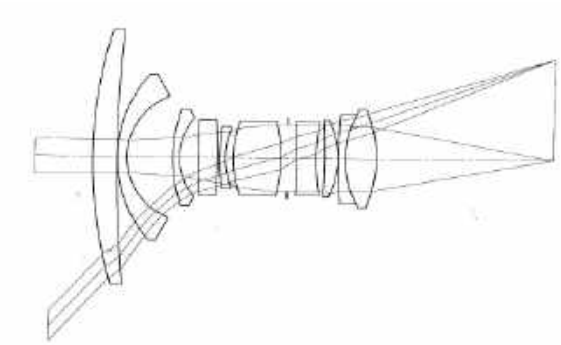
Image formation

Cameras and human eyes map part of the 3D world onto a 2D picture. The picture below shows the basic process for a pinhole camera. Light rays pass through the pinhole (P) and form an image on the camera's sensor array. When we're discussing geometry rather than camera hardware, it's often convenient to think of a mathematical image that lives in front of the pinhole. The two images are similar, except that the one behind the pinhole is upside down.



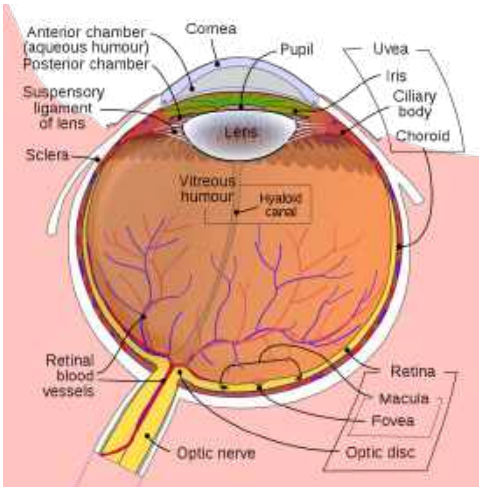
from Kingslake, Optics in Photography

Pinhole cameras are mathematically simple but don't allow much light through. So cameras use a lens to focus light coming in from a wider aperture. Typical camera lenses are built up out of multiple simple lenses. In the wide-angle lens shown below, a bundle of light rays coming from the original 3D location enters the front element at the left and is gradually focused into a single point on the image at the right.



from Kingslake, Optics in Photography

The human eye has a simpler lens, but also an easier task. At any given moment, only a tiny area in the center of your field of view is in sharp focus and, even within that area, objects are only in focus if they are at the depth you're paying attention to. The illusion of a wide field of view in sharp focus is created by the fact that your eyes constantly change their direction and focal depth, as your interest moves around the scene.



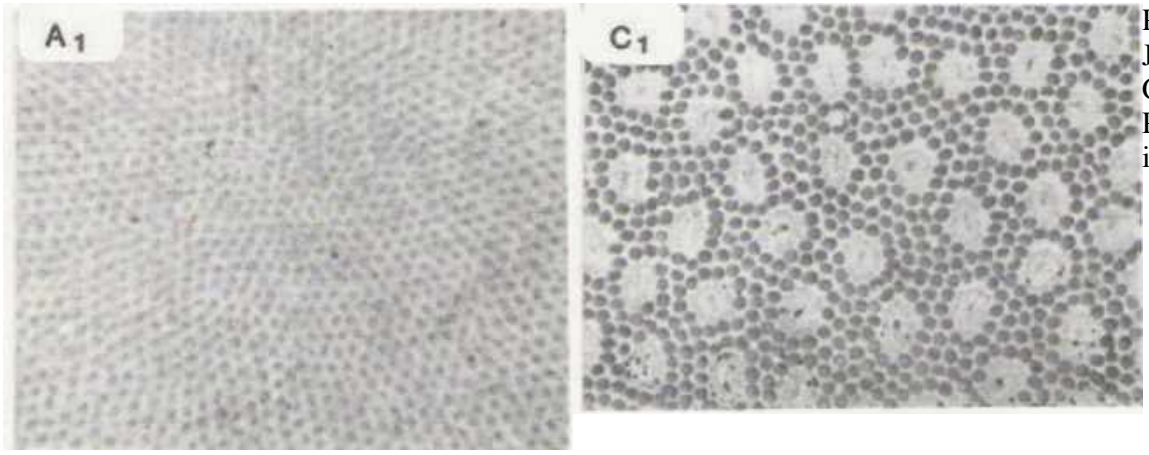
from Wikipedia

Sampling

The lens produces a continuous image on the camera. Digital cameras sample this to produce a 2D array of intensity values. A filter array sends red, green, or blue light preferentially to each (x,y) location in the sensor array, in an alternating pattern. This information is reformatted into an output in which each (x,y) position contains a "pixel" with three color values (red, green, and blue). Modern cameras produce images that are several thousand pixels wide and high, i.e. much more resolution than AI algorithms can actually handle.

The human retina uses variable-resolution sampling. The central region (fovea) is used for seeing fine details of the object that you're paying attention to. The periphery provides only a coarse resolution picture and is used primarily for navigational tasks such as not running into tables and staying 6' away from other people.

The lefthand photomicrograph below shows the pattern of cone cells in the fovea. The righthand picture shows the cones (large cells) in the periphery, interspersed with rods (small cells). Notice that the pattern is slightly irregular, which prevents aliasing artifacts. "Aliasing" is the creation of spurious low-frequency patterns from high-frequency patterns that have been sampled at too low a frequency.



Photomicrographs from John Yellott, "Spectral Consequences of Photoreceptor Sampling in the Rhesus Retina"

Each type of receptor responds preferentially to one range of light colors, as shown below. The red, green, and blue cones produce outputs similar to the ones from a digital camera, except that blue cones are much less common than blue and green ones. The rods (only one color type) are used for seeing in low-light conditions.

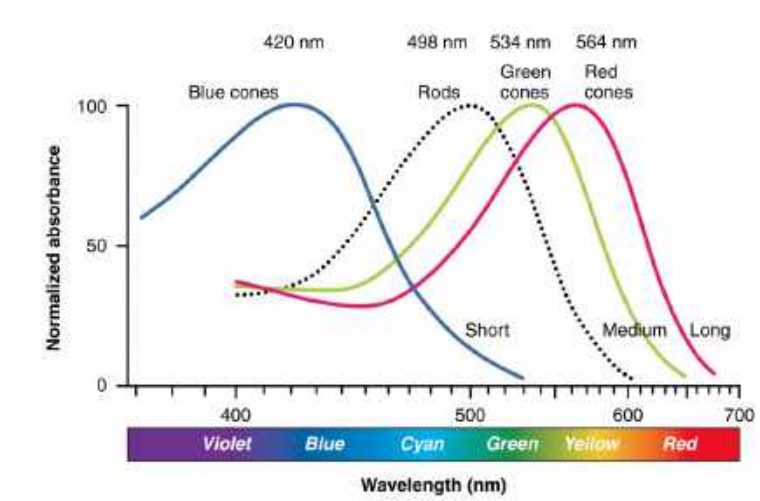


Illustration from Anatomy & Physiology, Connexions Web site. http://cnx.org/content/col11496/1.6/, Jun 19, 2013.

Objects to images

Recognizing objects from pictures is difficult because "the same" object never actually has the same pattern of pixel values. Most obviously, we could rotate or translate the position of the object in the picture. But, in addition, we might be seeing the object from two different viewpoints. E.g. compare the two views of the lemon tree (in the planter) and the fig tree (left plant in the righthand picture). Depending on specifics of the camera lens, simply moving the object from the middle to the edge of the view can also cause its shape to be distorted. The shape of an object looks different when it's seen from very close, e.g. a selfie vs. a professional photograph.

Parts of objects can be "occluded," i.e. disappear behind other objects (apple in the lefthand picture). A flexible object can be bent into different shapes (right below).

Differences in lighting can change the apparent color of an object, as well as the positions of highlights and shadows. The two pictures below are the same apple, photographed in different rooms. The apparent colors are different. Also, the lefthand picture has a dark shadow above the apple, whereas the righthand picture (taken in more diffuse lighting) has a light shadow below the apple.

Finally, natural objects are rarely identical, even when they are nominally of the same type, e.g. the two honeycrisp apples below.

Applications of Computer Vision

Computer vision can used for four basic types of tasks:

- Classification
- Reconstruction
- Image generation
- Making predictions

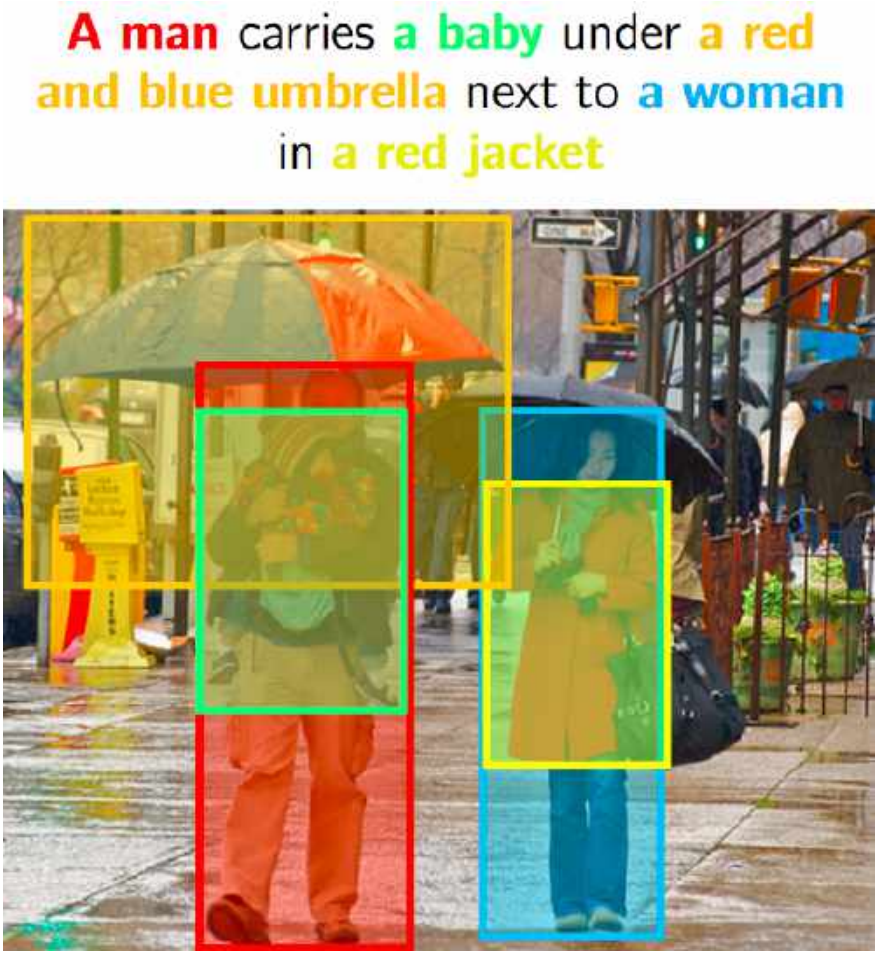
Classification

In its most basic form, classification involves coming up with a single label to describe the contents of an image. For example, the lefthand picture below shows an apple; the righthand picture shows a mug.


Using deep neural networks, top-5 error rates for classification on Imagenet (one standard benchmark set of labelled images) have gone from 27% in 2010 to <5% in 2015. Unfortunately, we have a rather poor model of why these networks work well, and how to construct good ones. As we will see later on, neural nets also have a tendency to fail unexpectedly.

Applications of classification

Recently, classifiers have been extended to work on more complex scenes. This involves two tasks which could theoretically be done as one step but frequently are not. First, we detect objects along with their approximate location in the image, producing a labelled image like the one below. This uses a lot of classifiers and works only on smallish sets of object types (e.g. tens, hundreds).



from Lana Lazebnik

When an object is mentioned in an image caption, similar methods can be used to locate the object in the image. After an object has been detected and approximately located, we can "register" it. That is, we can produce its exact position, pose, orientation.

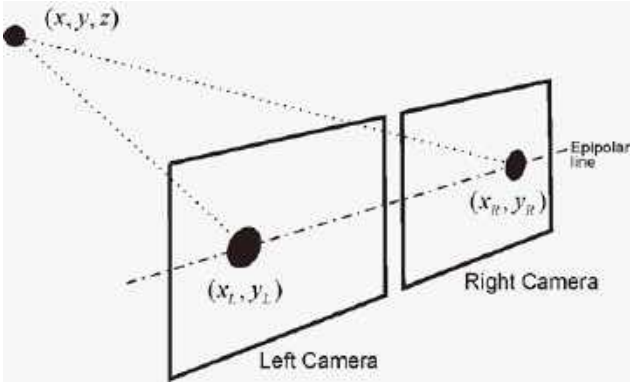
There has also been some success at labelling "semantic" roles in images, e.g. who is the actor? what are they acting on? what tool are they using? In the examples below, you can see that many roles are successfully recovered (green) but some are not (red). This kind of scene analysis can be used to generate captions for images and answer questions about what's in a picture.

from Yatskar, Zettlemoyer, Farhadi, "Situation Recognition" (2016)

Reconstruction

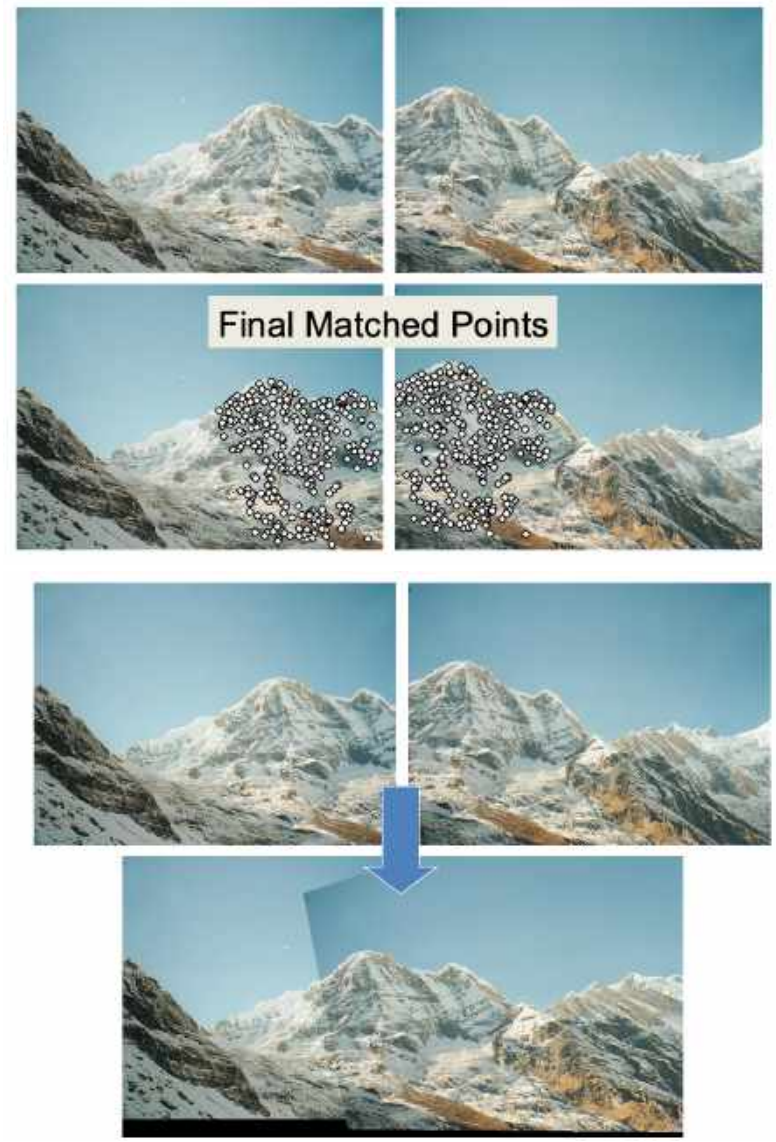
We can also reconstruct aspects of the 3D scene, without necessarily doing a deeper analysis or associating any natural language labels. Suppose, for example, that we have two pictures of a scene, either from a stereo pair of cameras or a moving camera. Notice that distant objects stay in approximately the same image position but closer objects move significantly.

It's fairly common that we know the relative positions of the two cameras. For example, they may both be mounted on the same robot platform. Human eyes are another example. In this situation, it's easy to turn a pair of matching features into a depth estimate. (See diagram below.) These depth estimates are highly accurate for objects near the observer, which conveniently matches what we need for manipulating objects that are within grabbing distance. People who lack stereo vision can get similar information by moving their heads.

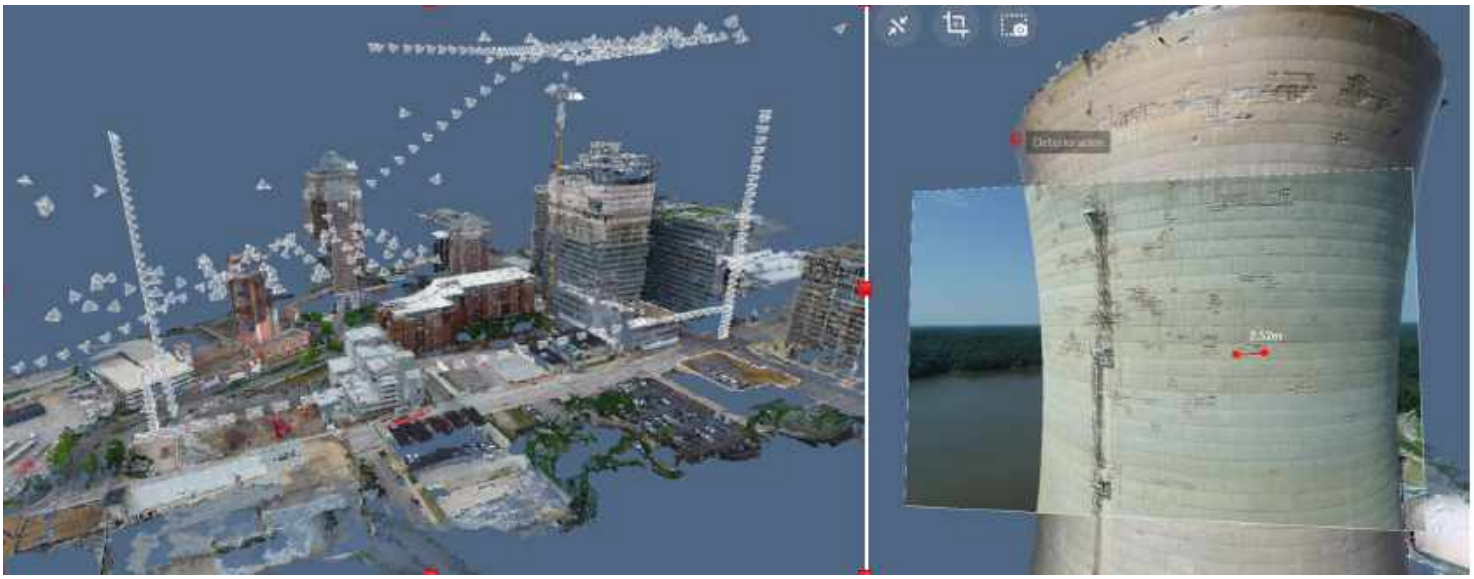


from Derek Hoiem

More interestingly, we can reconstruct the camera motion if we have enough matched features. For example, we can extract local features from a pair of images and use these as guidepoints to merge the two images into one. We can reconstruct a 3D scene down to very fine details if we have enough pictures. So we can send a drone into a space that is hazardous or inconvenient for humans to visit, and then direct a human inspector to problematic features (e.g. evidence of cracking in bridge supports).



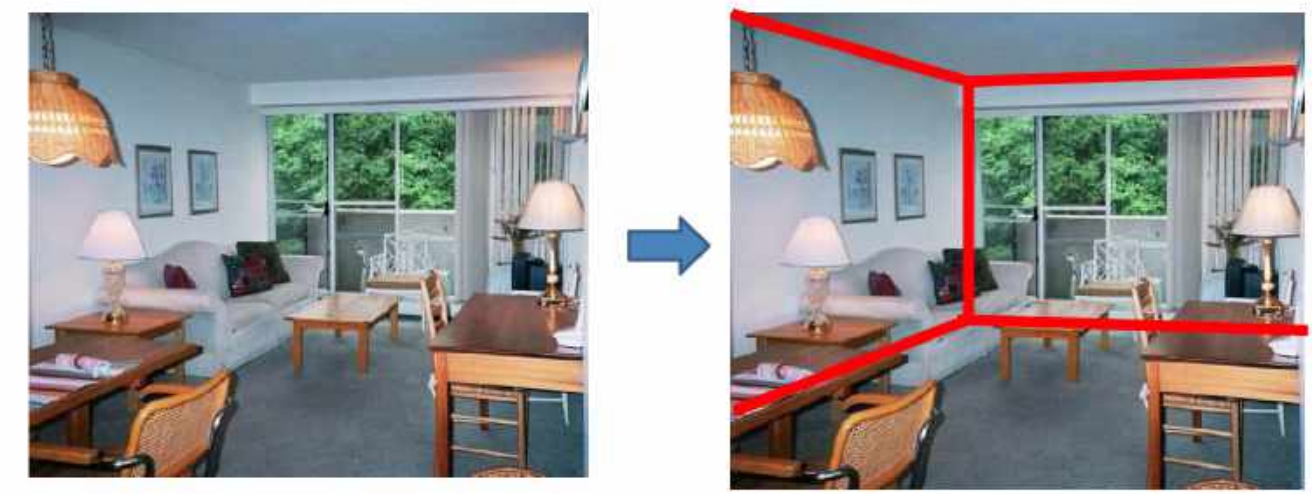
from Derek Hoiem



from Derek Hoiem

When you have enough training examples, or enough prior knowledge, it's possible to reconstruct 3D information from a single image. For example, a mobile robot might extract the sides of this path, use their vanishing point to guess the tilt, and figure out that the blue object is something that might block its route. In this case, it's a jacket, so some robots might be able to drive over it. But it could have been a large rock.

More interestingly, we can extract a 3D model for indoor scenes using our knowledge of typical room geometry, likely objects, etc. In the picture below, a program has outlined the edges delimiting the floor, ceiling, and walls.



from Derek Hoiem

Generation

Relatively recent neural net algorithms can create random images similar to a set of training images. This can be done for fun (e.g. the faces below). A serious application is creating medical images that can be for teaching/research, without giving away private medical data. Can work very well: doctors find it very hard to tell real xray images from fake ones.

from StyleGAN by Karras, Laine, Aila (NVIDIA) 2019

More interestingly, suppose that we have a 3D model for the scene. We now have enough information to add objects to the scene and make the result look right. (See picture below.) The 3D model is required to make objects sit correctly on surfaces (rather than hovering somewhere near them), give the object appropriate shading, and make light from (or bouncing off of) the object affect the shading on nearby objects. We can also remove objects, by making assumptions about what surface(s) probably continue behind them. And we can move objects from one picture into another.

Karsh, Hedau, Forsyth, Hoiem "Rendering Synthetic Objects into Legacy Photographs" (2011)

Here is a cool video showing details of how this algorithm works (from Kevin Karsh).

Predicting the future

A much harder, but potentially more useful, task would be to predict the future from our current observations. For example, what's about to happen to the piece of ginger in the picture below?

Some predictions require sophisticated understanding of the scene. But, interestingly, others do not. If an object is moving directly towards you, it expands in your field of view without moving sideways. Moreover, the time to collision is its size in the image times the derivative of the size. Humans and many other animals use this trick to quickly detect potential collisions. Collision avoidance is one of the main roles of the low-resolution peripheral vision in humans.

11 Classifiers

The big idea

Classifiers turn sets of input features into labels. At a very high level:

- Train the classifier on the training data.
- Tune any parameters (e.g. the smoothing constant in Naive Bayes) by trying to classify the development data.
- Discover how well we did using the final test data.

Classifiers can be used to classify static objects and to make decisions.

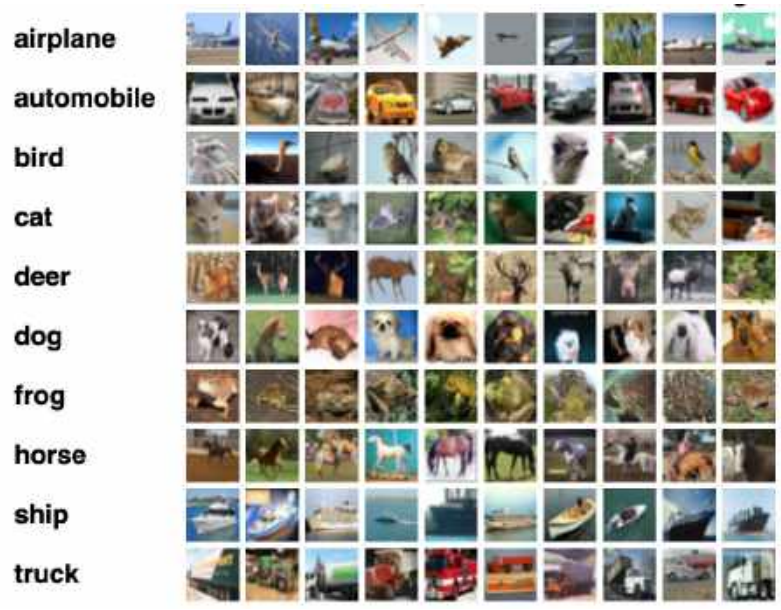
What is the topic?

We've seen a naive Bayes classifier used to identify the type of text, e.g. label extracts such as the following as math text, news text, or historical document.

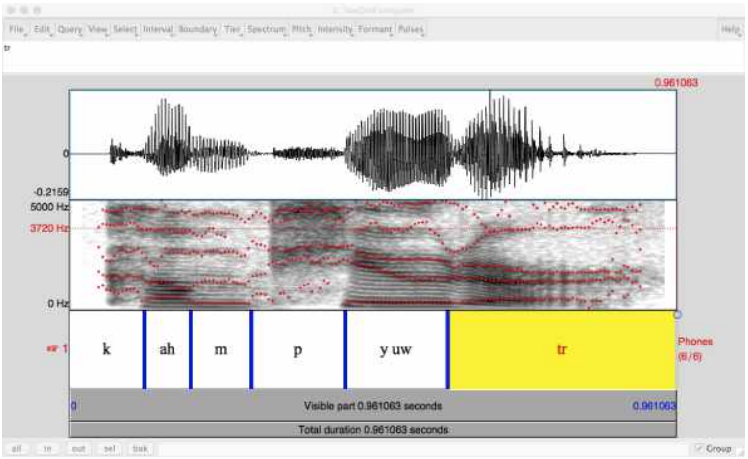
> Let V be a particular set of vertices of cardinality n. How many different (simple, undirected) graphs are there with these n vertices? Here we will assume that we are naming the edges of the graph using a pair of vertices (so you cannot get new graphs by simply renaming edges). Briefly justify your answer. (from CS 173 study problems)
>
> Senate Majority Leader Mitch McConnell did a round of interviews last week in which he said repealing Obamacare and making cuts to entitlement programs remain on his agenda. Democrats are already using his words against him. (from CNN, 21 October 2018)
>
> Congress shall make no law respecting an establishment of religion, or prohibiting the free exercise thereof; or abridging the freedom of speech, or of the press; or the right of the people peaceably to assemble, and to petition the Government for a redress of grievances.

Labelling low-level input

Classifiers can also be used label low-level input. For example, we can label what kind of object is in a picture. Or we can identify what phone has been said in each timeslice of a speech signal.



from Alex Krizhevsky's web page

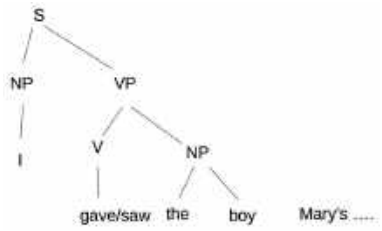


"computer"

Making decisions

Classifiers can also be used to make decisions. For example, we might examine the state of a video game and decide whether to move left, move right, or jump. In the pictures below, we might want to decide if it's safe to move forwards along the path or predict what will happen to the piece of ginger.


Classifiers may also be used to make decisions internally to an AI system. Suppose that we have constructed the following parse tree and the next word we see is "Mary's".



We have to decide how to attach "Mary's" into the parse tree. In "I gave the boy Mary's hat," "Mary's hat" will be attached to the VP as a direct object. In "I saw the boy Mary's going out with," the phrase "Mary's going out with" is a modifier on "boy." The right choice here depends on the verb (gave vs. saw) and lookahead at the words immediately following "Mary's".

What kind of answer do we want?

Current algorithms require a clear (and fairly small) set of categories to classify into. That's largely what we'll assume in later lectures. But this assumption is problematic, and could create issues for deploying algorithms into the unconstrained real world.

For example, the four objects below are all quite similar, but nominally belong to three categories (peach, Asian pear, and two apples). How specific a label are we looking for, e.g. "fruit", "pear", or "Asian pear"? Should the two apples be given distinct labels, given that one is a Honeycrisp and one is a Granny Smith? Is the label "apple" correct for a plastic apple, or an apple core? Are some classification mistakes more/less acceptable than others? E.g. mislabelling the Asian pear as an apple seems better than calling it a banana, and a lot better than calling it a car.

What happens if we run into an unfamiliar type of object? Suppose, for example, that we have never seen an Asian pair. This problem comes up a lot, because people can be surprisingly dim about naming common objects, even in their first language. Do you know what a parsnip looks like? How about a plastron? Here is an amusing story about informant reliability from Robert M. Laughlin (1975, The Great Tzotzil Dictionary). In this situation, people often use the name of a familiar object, perhaps with a modifier, e.g. a parsnip might be "like a white carrot but it tastes different." Or they may identify the general type of object, e.g. "vegetable."

Some unfamiliar objects and activities can be described in terms of their components. E.g. the wind chime below has a big metal disk and some mysterious wood parts. The middle picture below is "a house with a shark sticking out of it." The bottom picture depicts ironing under water.

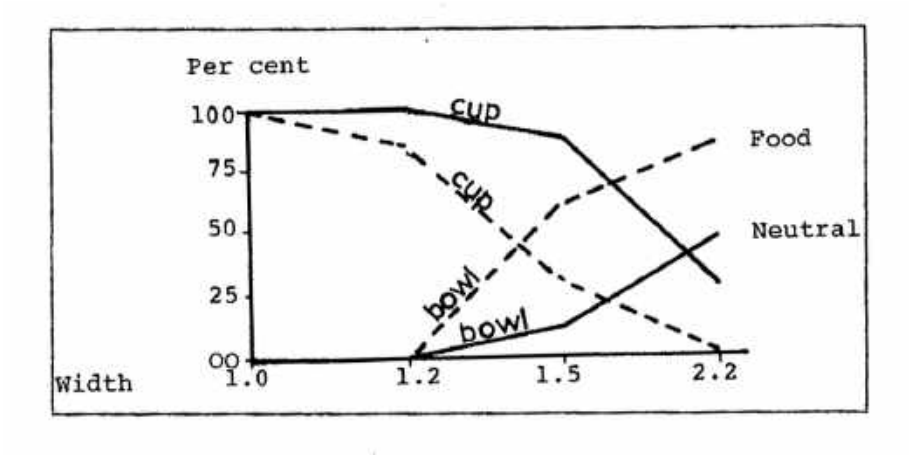


shark from Wikipedia, diver from Sad and Useless

When labelling objects, people use context as well as intrinsic properties. This can be particularly important when naming a type of object whose appearance can vary substantially, e.g. the vases in the picture below.

A famous experiment by William Labov (1975, "The boundaries of words and their meanings") showed that the relative probabilities of two labels (e.g. cup vs. bowl) change gradually as properties (e.g. aspect ratio) are varied. For example, the tall container with no handle (left) is probably a vase. The round container on a stem (right) is probably a glass. But what are the two in the middle?

The very narrow container was sold as a vase. But filling it with liquid would encourage people to call it a glass (perhaps for a fancy liquor). Installing flowers in the narrow wine glass makes it seem more like a vase. And you probably didn't worry when I called it a "vase" above. The graph below (from the original paper) shows that imagining a food context makes the same picture more likely to be labelled as a bowl rather than a cup.



With complex scenes, it's not clear how many objects to explicitly call out. E.g. is the picture below showing a garden or a street? Are we supposed to be labelling each plant? Should we mark out the street lamp? The sidewalk?

When you're talking to another human, it's much more clear what aspects of the scene are important. First, there is usually some larger context that the conversation fits into (e.g. "gardening" or "traffic"). Second, people look at things they are interested in, e.g. compare the two pictures below. People have very strong ability to track another person's gaze as we watch what they are doing. Because of their close association with humans, the same applies to dogs. Using gaze to identify interest is believed to be an integral part of how children associate names with objects. Lack of appropriate eye contact is one reason that conversations feel glitchy on video conferencing systems.

It's tempting to see this problem as specific to computer vision. However, there are similar examples from other domains. E.g. in natural language processing, it's common to encounter new words. These can be unfamiliar proper nouns (e.g. "Gauteng", "covid"). Or they may be similar to familiar words, e.g. typos ("coputer") or new compounds ("astroboffin") or melds ("brexit", "splinch"). Phone recognition may involve more or less detail and mistakes may be minor (e.g. "t" vs. "d") or major (e.g. "t" vs. "a"). And people use prosodic features (e.g. length, stress) to indicate which words are most important for the listener to pay attention to.

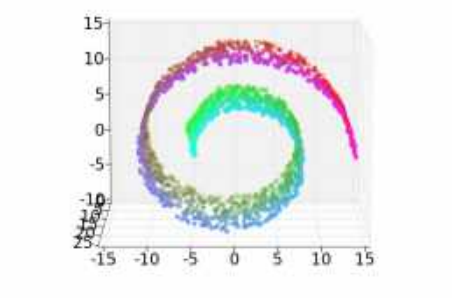
Overall system

Classifier systems often have several stages. "Deep" models typically have more distinct layers. Each layer in a deep model does a simpler task and the representation type more changes gradually from layer to layer.

	speech recognizer	**object identification**	**scene reconstruction**	**parsing**	**word meaning**
raw input	waveform	image	images	text	text
low-level features	MFCC features	edges, colors	corners, SIFT features	stem, suffixes word bigrams	word neighbors
low-level labels	phone probabilities	local object ID	raw matchpoints	POS tags	word embeddings
output labels	word sequence	image description	scene geometry	parse tree	word classes

Representations for the raw input objects (or sometimes the low-level features) have a large number of bits. E.g. a color image with 1000x1000 pixels, each containing three color values. Early processing often tries to reduce the dimensionality by picking out a smaller set of "most important" dimensions. This may be accomplished by linear algebra (e.g. principal components analysis) or by classifiers (e.g. word embedding algorithms).

When the data is known to live on a subset of lower dimension, vectors may be mapped onto this subset by so-called "manifold learning" algorithms. For example, color is used to enclose class (output label). Distance along the spiral correctly encodes class similarity. Straight-line distance in 3D does the wrong thing: some red-green pairs of points are closer than some red-red pairs.



Algorithm overview

There are two basic approaches to each stage of processing:

- deterministic transformations (e.g. edge finders, MFCC transformation)
- classifiers with learned parameters

Many real systems are a mix of the two. A likely architecture for a speech recognizer (e.g. the Kaldi system) would start by converting the raw signal to MFCC features, then use a neural net to identify phones, then use an HMM-type decoder to extract the most likely word sequence.

There are three basic types of classifiers with learned parameters. We've already seen the first type and we're about to move onto the other three

- Statistical (e.g. Naive Bayes, HMMs)
- Simple traditional classifiers: decision trees, k-nearest neighbor
- Linear classifiers (perceptrons)
- Neural nets

What can we tune?

Classifiers typically have a number of overt variables that can be manipulated to produce the best outputs. These fall into two types:

- "Parameters": values learned directly from the training set (e.g. probabilities in Bayes nets, weights on the elements in neural nets)
- "Hyperparameters": tuning constants adjusted using the development data (e.g. the Laplace smoothing constant in naive Bayes).

Researchers also tune the structure of the model.

- General design of the algorithm, e.g. neural net vs. HMM
- Geometry of the model, e.g. the number of units and the connections in a Bayes net or neural net
- Theory-based parameters, e.g. "a word must have at least one vowel."

It is now common to use automated scripts to tune hyperparameters and some aspects of the model structure. Tuning these additional values is a big reason why state-of-the-art neural nets are so expensive to tune. All of these tunable values should be considered when assessing whether the model could potentially be over-fit to the training/development data, and therefore might not generalize well to slightly different test data.

Training and testing can be interleaved to varying extents:

- Batch: train first, then test
- Incremental: test as we train

Incremental training is typically used by algorithms that must interact with the world before they have finished learning. E.g. children need to interact with their parents from birth, long before their language skills are fully mature. Sometimes interaction is to get their training data. E.g. a video game player may need to play experimental games. Incremental training algorithms are typically evaluated based on their performance towards the end of training.

Supervised training

Traditionally, most classifiers used "supervised" training. That is, someone gives the learning algorithm a set of training examples, together with correct ("gold") output for each one. This is very convenient, and it is the setting in which we'll learn classifier techniques. But this only works when you have enough labelled training data. Usually the amount of training data is inadequate, because manual annotations are very slow to produce.

Moreover, labelled data is always noisy. Algorithms must be robust to annotation mistakes. It may be impossible to reach 100% accuracy because the gold standard answers aren't entirely correct.

- Even trained annotators make errors.
- Much annotation is done "quick and dirty" by amateurs (e.g. Mechanical Turk).
- The supposedly correct answers may have been scraped off the web and not fully examined for correctedness.

Finally, large quantities of correct answers may be available only for the system as a whole, not for individual layers. Marking up correct answers for intermediate layers frequently requires expert knowledge, so cannot easily be farmed out to students or internet workers.

Workarounds for limited training data

Researchers have a used a number of methods to work around lack of explicit labelled training data. For example, current neural net software lets us train the entire multi-layer as a unit. Then we need correct outputs only for the final layer. This type of training optimizes the early layers only for this specific task. However, it may be possible to quickly adapt this "pre-trained" layer for use in another task, using only a modest amount of new training data for the new task.

For example, the image colorization algorithm shown below learns how to add color to black and white images. The features it exploits in the monochrome images can also (as it turns out) be used for object tracking.

from Zhang, Isola, Efros Colorful Image Colorization

For this colorization task, the training pairs were produced by removing color from color pictures. That is, we subtracted information and learned to restore it. This trick for producing training pairs has been used elsewhere. For example, the BERT system trains natural language models by masking off words in training sentences, and learning to predict what they were. Similarly, the pictures below illustrate that we can learn to fill in missing image content, by creating training images with part of the content masked off.

from Deepak Pathak et al

Another approach is self-supervision, in which the learner experiments with the world to find out new information. Examples of self-supervision include

- calibrating a camera by making motions specifically intended to reveal calibration problems
- curiosity-driven learning: the AI tries to make novel things happen (like a toddler)

The main challenge for self-supervised methods is designing ways for a robot to interact with a real, or at least realistic, world without making it too easy for the robot to break things, injury itself, or hurt people.

"Semi-supervised" algorithms extract patterns from unannotated data, combining this with limited amounts of annotated data. For example, a translation algorithm needs some parallel data from the two languages to create rough translations. The output can be made more fluent by using unannotated data to learn more about what word patterns are/aren't appropriate for the output language. Similarly, we can take observed pauses in speech and extrapolate the full set of locations where it is appropriate to pause (i.e. word boundaries).

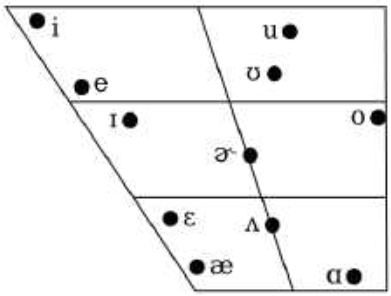
"Unsupervised" algorithms extract patterns entirely from unannotated data. These are much less common than semi-supervised methods. Sometimes so-called unsupervised methos actually have some implicit supervision (e.g. sentence boundaries for a parsing algorithm). Sometimes they are based on very strong theoretical assumptions about how the input must be structured.

Recap

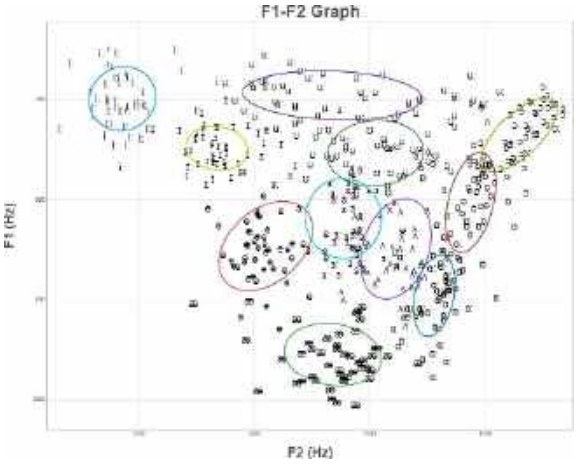
Recall: a classifier maps input examples (each a set of feature values) to class labels. Our task is to train a model on a set of examples with labels, and then assign the correct labels to new input examples.

Messy classification example

Here's a schematic representation of English vowels (from Wikipedia). From this, you might reasonable expect to experimental data to look like several tight clusters of points, well-separated. If that were true, then classification would be easy: model each cluster with a regeion of simple shape (e.g. ellipse), compare a new sample to these region, and pick the one that contains it.



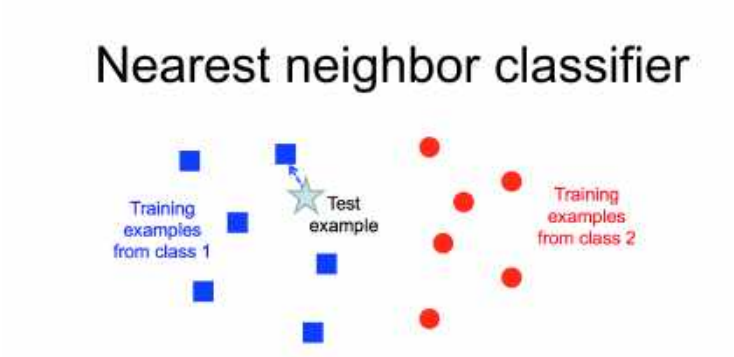
However, experimental datapoints for these vowels (even after some normalization) actually look more like the following. If we can't improve the picture by clever normalization (sometimes possible), we need to use more robust methods of classification.



(These are the tirst two formants of English vowels by 50 British speakers, normalized by the mean and standard deviation of each speaker's productions. From the Experimental Phonetics course PALSG304 at UCL.)

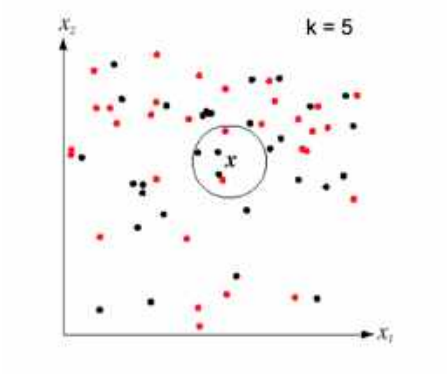
K nearest neighbors

Another way to classify an input example is to find the most similar training example and copy its label.



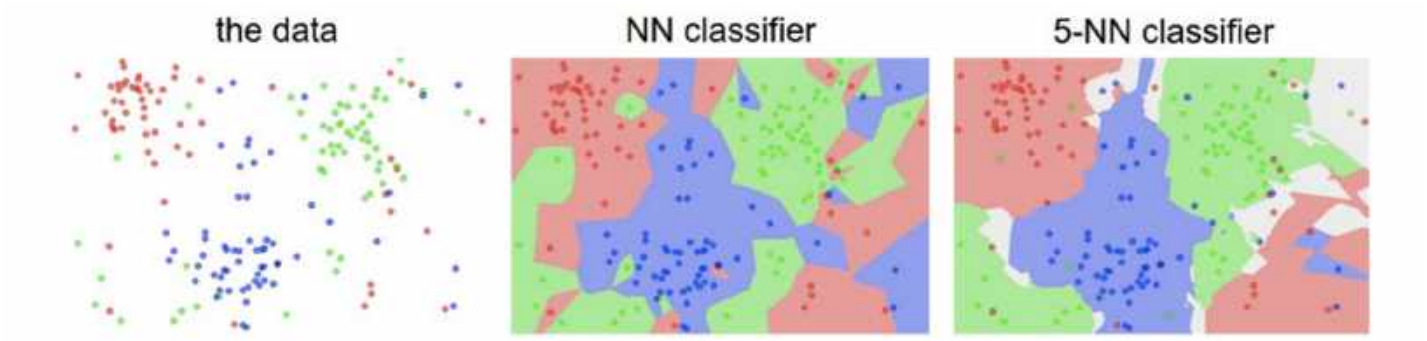
from Lana Lazebnik

This "nearest neighbor" method is prone to overfitting, especially if the data is messy and so examples from different classes are somewhat intermixed. However, performance can be greatly improved by finding the k nearest neighbors for some small value of k (e.g. 5-10).



from Lana Lazebnik

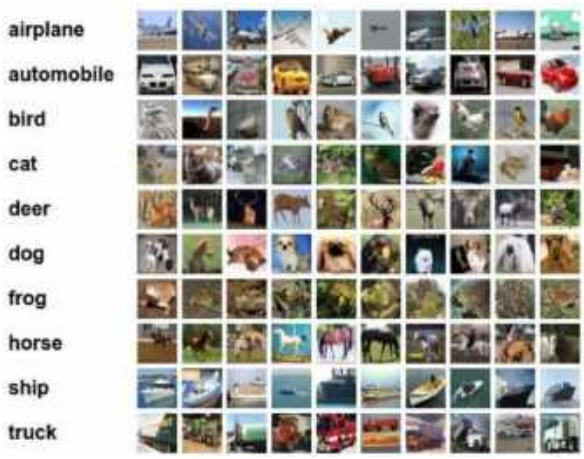
K nearest neighbors can model complex patterns of data, as in the following picture. Notice that larger values of k smooth the classifier regions, making it less likely that we're overfitting to the training data.



from Andrej Karpathy

A limitation of k nearest neighbors is that the algorithm requires us to remember all the training examples, so it takes a lot of memory if we have a lot of training data. Moreover, the obvious algorithm for classifying an input image X requires computing the distance from X to every stored image. In low dimensions, a k-D tree can be used to find efficiently find the nearest neighbors for an input example. However, efficiently handling higher dimensions is more tricky and beyond the scope of this course.

This method can be applied to somewhat more complex types of examples. Consider the CFAR-10 dataset below. (Pictures are from Andrej Karpathy cs231n course notes.) Each image is 32 by 32, with 3 color values at each pixel. So we have 3072 feature values for each example image.



We can calculate the distance between two examples by subtracting corresponding feature values. Here is a picture of how this works for a 4x4 toy image.

test image			
56	32	10	18
90	23	128	133
24	26	178	200
2	0	255	220

-

training image			
10	20	24	17
8	10	89	100
12	16	178	170
4	32	233	112

=

pixel-wise absolute value differences			
46	12	14	1
82	13	39	33
12	10	0	30
2	32	22	108

→ 456

The following picture shows how some sample test images map to their 10 nearest neighbors.

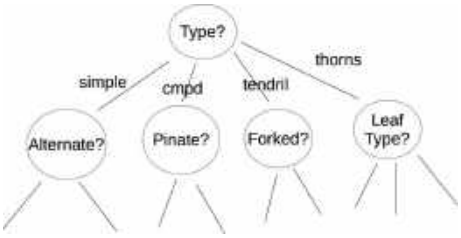
More precisely, suppose that $\vec{x}$ and $\vec{y}$ are two feature vectors (e.g. pixel values for two images). Then there are two different simple ways to calculate the distance between two feature vectors:

- L1 norm or Manhattan distance: $\sum_i |x_i - y_i|$
- L2 norm or straight-line distance: $\sqrt{\sum_i (x_i - y_i)^2}$

As we saw with A* search for robot motion planning, the best choice depends on the details of your dataset. The L1 norm is less sensitive to outlier values. The L2 norm is continuous and differentiable, which can be handy for certain mathematical derivations.

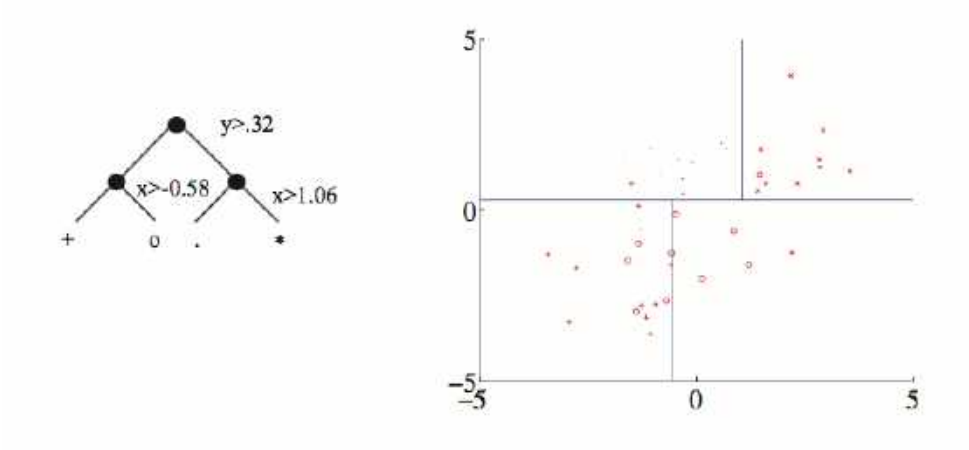
Decision trees

A decision tree makes its decision by asking a series of questions. For example, here is a a decision tree user interface for identifying vines (by Mara Cohen, Brandeis University). The features, in this case, are macroscopic properties of the vine that are easy for a human to determine, e.g. whether it has thorns. The top levels of this tree look like



Depending on the application, the nodes of the tree may ask yes/no questions, multiple choice (e.g. simple vs. complex vs. tendril vs thorns), or set a threshold on a continuous space of values (e.g. is the tree below or above 6' tall?). The output label may be binary (e.g. is this poisonous?), from a small set (edible, other non-poisonous, poisonous), or from a large set (e.g. all vines found in Massachusetts). The strength of decision trees lies in this ability to use a variety of different sorts of features, not limited to well-behaved numbers.

Here's an example of a tree created by thresholding two continuous variable values (from David Forsyth, Applied Machine Learning). Notice that the split on the second variable (x) is different for the two main branches.

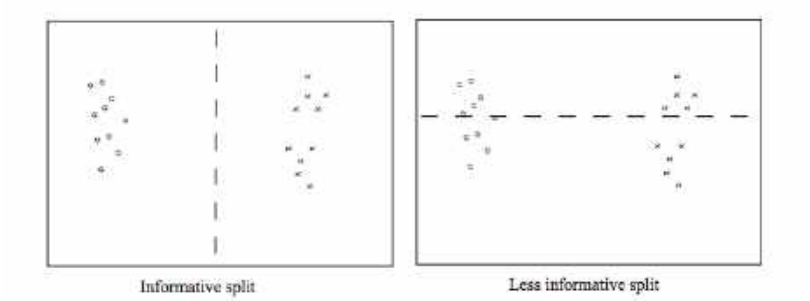


Each node in the tree represents a pool of examples. Ideally, each leaf node contains only objects of the same type. However, that isn't always true if the classes aren't cleanly distinguished by the available features (see the vowel picture above). So, when we reach a leaf node, the algorithm returns the most frequent label in its pool.

If we have enough decision levels in our tree, we can force each pool to contain objects of only one type. However, this is not a wise decision. Deep trees are difficult to construct (see the algorithm in Russell and Norvig) and prone to overfitting their training data. It is more effective to create a "random forest", i.e. a set of shallow trees (bounded depth) and have these trees vote on the best label to return. The set of shallow trees is created by choosing variables and possible split locations randomly.

Informative splits

Whether we're building a deep or shallow tree, some possible splits are a bad idea. For example, the righthand split leaves us with two half-size pools that are no more useful than the original big pool.



From David Forsyth, Applied Machine Learning.

A good split should produce subpools that are less diverse than the original pool. If we're choosing between two candidate splits (e.g. two thresholds for the same numerical variable), it's better to pick the one that produces subpools that are more uniform. Diversity/uniformity is measured using entropy.

Huffman codes

Suppose that we are trying to compress English text, by translating each character into a string of bits. We'll get better compression if more frequent characters are given shorter bit strings. For example, the table below shows the frequencies and Huffman codes for letters in a toy example from Wikipedia.

Char	Freq	Code	Char	Freq	Code	Char	Freq	Code	Char	Freq	Code	Char	Freq	Code	Char	Freq	Code	Char	Freq	Code
space	7	111	a	4	010	e	4	000												
f	3	1101	h	2	1010	i	2	1000	m	2	0111	n	2	0010	s	2	1011	t	2	0110
l	1	11001	o	1	00110	p	1	10011	r	1	11000	u	1	00111	x	1	10010			

Entropy

Suppose that we have a (finite) set of letter types S. Suppose that M is a finite list of letters from S, with each letter type (e.g. "e") possibly included more than once in M. Each letter type c has a probability P(c) (i.e. its fraction of the members of M). An optimal compression scheme would require $\log(\frac{1}{P(c)})$ bits in each code. (All logs in this discussion are base 2.)

To understand why this is plausible, suppose that all the characters have the same probability. That means that our collection contains $\frac{1}{P(c)}$ different characters. So we need $\frac{1}{P(c)}$ different codes, therefore $\log(\frac{1}{P(c)})$ bits in each code. For example, suppose S has 8 types of letter and M contains one of each. Then we'll need 8 distinct codes (one per letter), so each code must have 3 bits. This squares with the fact that each letter has probability 1/8 and $\log(\frac{1}{P(c)}) = \log(\frac{1}{1/8}) = \log(8) = 3$.

The entropy of the list M is the average number of bits per character when we replace the letters in M with their codes. To compute this, we average the lengths of the codes, weighting the contribution of each letter c by its probability P(c). This give us

$$H(M) = \sum_{c \in S} P(c) \log(\frac{1}{P(c)})$$

Recall that $\log(\frac{1}{P(c)}) = -\log(P(c))$. Applying this, we get the standard form of the entropy definition.

$$H(M) = -\sum_{c \in S} P(c) \log(P(c))$$

In practice, good compression schemes don't operate on individual characters and also they don't achieve this theoretical optimum bit rate. However, entropy is a very good way to measure the homogeneity of a collection of discrete values, such as the values in the decision tree pools.

For more information, look up Shannon's source coding theorem.

CS 440/ECE 448
Margaret Fleck

12 Linear Classifiers

Big picture

Linear classifiers classify input items using a linear combination of feature values. Depending on the details, and the mood and theoretical biases of the author, a linear classifier may be known as

- perceptrons
- single units of a neural net
- logistic regression
- a support vector machine (SVM)

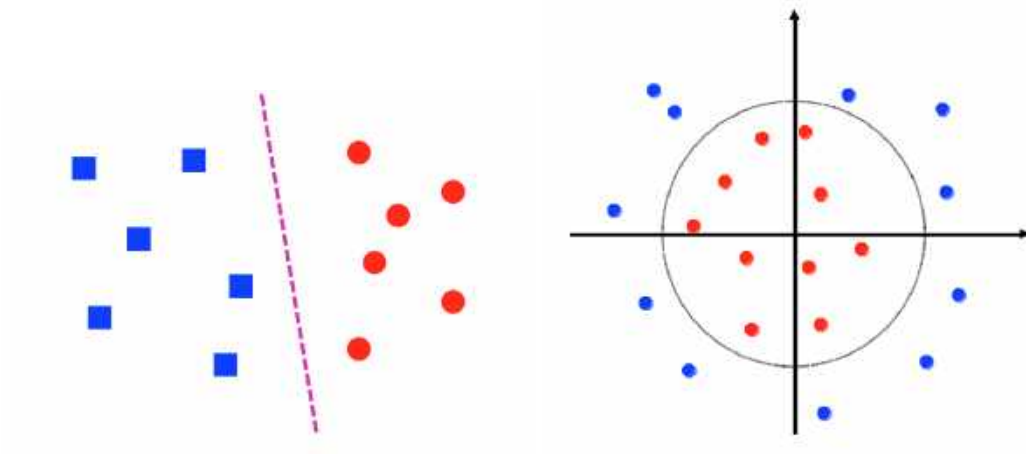
The variable terminology was created by the convergence of two lines of research. One (neural nets, perceptrons) attempts to model neurons in the human brain. The other (logistic regression, SVM) evolved from standard methods for linear regression.

A key limitation of linear units is that class boundaries must be linear. Real class boundaries are frequently not linear. So linear units typically

- used in groups (e.g. neural nets) and/or
- fed data after some prior processing (e.g. non-linear or dimension reducing)

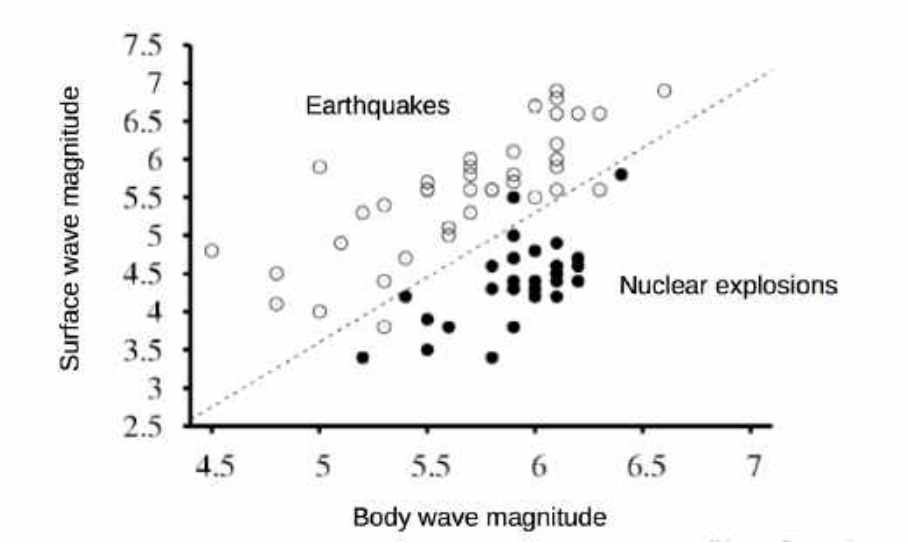
What is linearly separable?

A dataset is linearly separable if we can draw a line (or, in higher dimensions, a hyperplane) that divides one class from the other. The lefthand dataset below is linearly separable; the righthand one is not.



from Lana Lazebnik

A common situation is that data may be almost linearly separable. That is, we can draw a line that correctly predicts the class for most of the data points, and we can't do any better by adjusting the line. That's often the best that can be done when the data is messy.

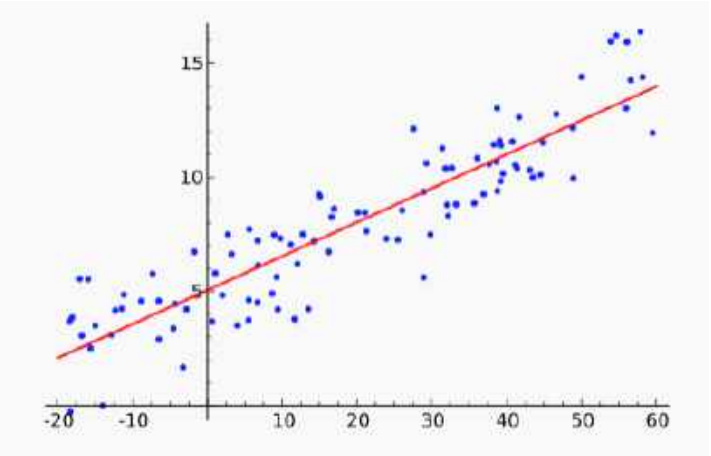


from Lana Lazebnik

A single linear unit will try to separate the data with a line/hyperplane, whether or not the data really looks like that.

Review: Linear regression

You've probably seen linear regression or, at least, used a canned function that performs it. However, you may well not remember what the term refers to. Linear regression is the process of fitting a linear function f to a set of (x,y) points. That is, we are given a set of data values x_i, y_i. We stipulate that our model f must have the form $f(x) = ax + b$. We'd like to find values for a and b that minimize the distance between the $f(x_i)$ and y_i, averaged over all the datapoints. For example, the red line in this figure shows the line f that approximates the set of blue datapoints.



From Wikipedia

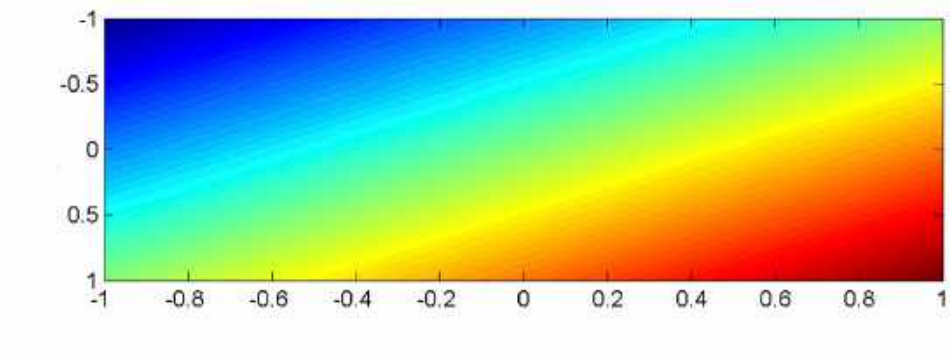
There are two common ways to "measure the distance between $f(x_i)$ and y_i":

- L1 norm or Manhattan distance: $\sum_i |f(x_i) - y_i|$
- L2 norm or straight-line distance: $\sqrt{\sum_i (f(x_i) - y_i)^2}$

The popular "least squares" method uses the L2 norm. Because this error function is a simple differentiable function of the input, there is a well-known analytic solution for finding the minimum. (See the textbook or many other places.) There are also so-called "robust" methods that minimize the L1 norm. These are a bit more complex but less sensitive to outliers (aka bad input values).

What does a linear function look like?

In most classification tasks, our input values are a vector $\vec{x}$. Suppose that $\vec{x}$ is two-dimensional, so we're fitting a model function $f(x_1, x_2)$. A linear function of two input variables must look as follows, where color shows values of the output value $f(x_1, x_2)$. (red high to blue low).

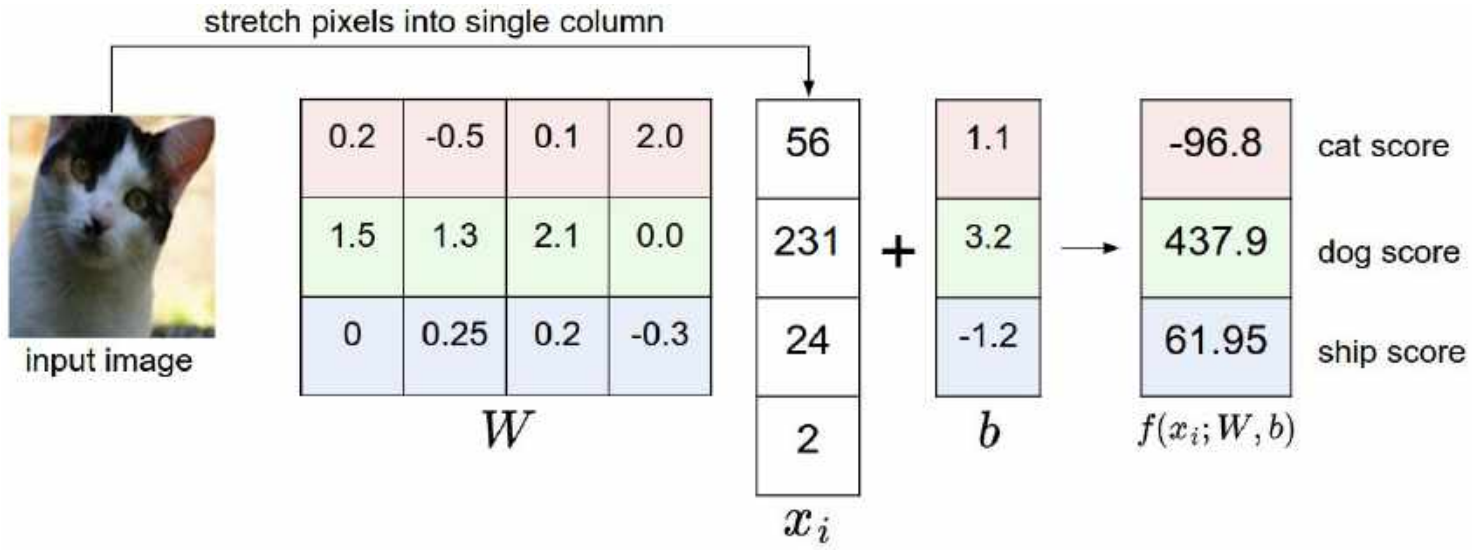


from Mark Hasegawa-Johnson, CS 440, Spring 2018

In classification, we will typically be comparing the output value $f(x_1, x_2)$. against a threshold T. In this picture, $f(x_1, x_2) = T$ is a color contour. E.g. perhaps it's the middle of the yellow stripe. The division boundary (i.e. $f(x_1, x_2) = T$) is always linear: a straight line for a 2D input, a plane for inputs of higher dimension.

Basic linear Classifier

The basic set-up for a linear classifier is shown below. Look at only one class (e.g. pink for cat). Each input example generates a feature vector (x). The four pixel values x_1, x_2, x_3, x_4 are multiplied by the weights for that class (e.g. the pink cells) to produce a score for that class. A bias term (b) is added to produce the weighted sum $w_1 x_1 + \ldots . w_n x_n + b$.



(from Andrej Karpathy cs231n course notes via Lana Lazebnik)

In this toy example, the input features are are individual pixel values. In a real working system, they would probably be something more sophisticated, such as SIFT features or the output of an early stage of neural processing. In a text classifcation task, one might use word or bigram probabilities.

To make a multi-way decision (as in this example), we compare the output values for several classes and pick the class with the highest value. In this toy example, the algorithm makes the wrong choice (dog). Clearly the weights w_i are key to success.

To make a yes/no decision, we compare the weighted sum to a threshold. The bias term adjusts the output so that we can use a standard threshold: 0 for neural nets and 0.5 for logistic regression. (We'll assume we want 0 in what follows.) In situations where the bias term is notationally inconvenient, we can make it disappear (superficially) by introducing a new input feature x_0 which is tied to the constant 1. Its weight w_0 is the bias term.

Why isn't this Naive Bayes?

In Naive Bayes, we fed a feature vector into a somewhat similar equation (esp. if you think about everything in log space). What's different here?

First, the feature values in Naive Payes were probabilities. E.g. a single number might be the probability of seeing the word "elephant" in some type of text. The values here are just numbers, not in the range [0,1]. We don't know whether all of them are on the same scale.

In naive Bayes, we assumed that features were conditionally independent, given the class label. This may be a reasonable approximation for some sets of features, but it's frequently wrong enough to cause problems. For example, in a bag of words model, "San" and "Francisco" do not occur independently even if you know that the type of text is "restaurant reviews." Logistic regression (a natural upgrade for naive Bayes) allows us to handle features that may be correlated. For example, the individual weights for "San" and "Francisco" can be adjusted so that the pair's overall contribution is more like that of a single word.

Basic perceptron

Recall that a linear classifier makes its decisions based on a linear combination of the input feature values. That is, if our input features values are $x_1, x_2, \ldots, x_n$, it tests whether

$$w_1 x_1 + \ldots . w_n x_n + b \geq 0$$

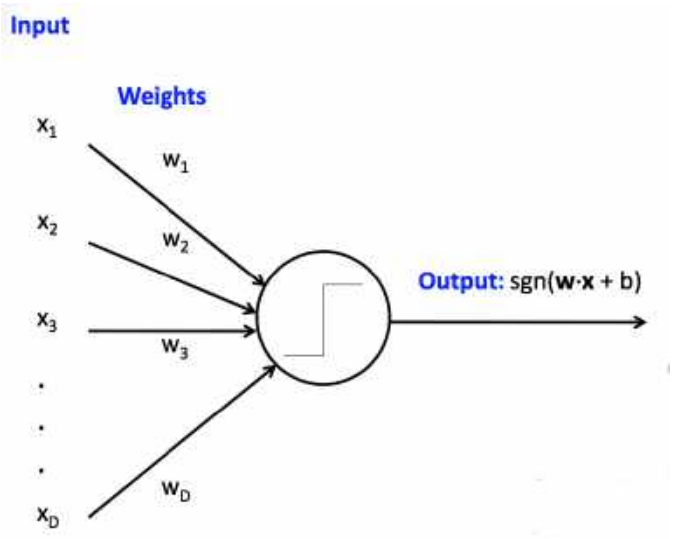
To avoid having to deal separately with the bias term b, we replace this with a fake feature x_0 that is always set to 1, plus an additional weight w_0. So then our decision equation becomes

$$w_0 x_0 + w_1 x_1 + \ldots . w_n x_n \geq 0$$

Reformulating a bit further, let's pass this weighted sum through the sign function (sgn). We'll follow the neural net terminology and call this final function the "activation function." So then our classification is based on this test:

Is $\text{sgn}(w_0 x_0 + w_1 x_1 + \ldots . w_n x_n)$ equal to 1 or -1?

We can make one of these units look like a neuron or a circuit component by drawing it like this:



(from Mark Hasegawa-Johnson, CS 440, Spring 2018)

This particular type of linear unit unit is called a perceptron. Perceptrons have limited capabilities but are particularly easy to train.

The training process

So far, we've assumed that the weights $w_0, w_1, \ldots, w_n$ were given to us by magic. In practice, they are learned from training data. Suppose that we have a set T of training data pairs (x,y), where x is a feature vector and y is the (correct) class label. Our training algorithm looks like:

- Initialize weights $w_0, \ldots, w_n$ to zero
- For each training example (x,y), update the weights to improve the changes of classifying x correctly.

Perceptron Learning Rule

Let's suppose that our unit is a classical perceptron, i.e. our activation function is the sign function. Then we can update the weights using the "perceptron learning rule". Suppose that x is a feature vector, y is the correct class label, and $\hat{y}$ is the class label that was computed using our current weights. Then our update rule is:

- If $\hat{y}$ = y, do nothing.
- Otherwise $w_i = w_i + \alpha \cdot y \cdot x_i$

The "learning rate" constant α controls how quickly the update process changes in response to new data.

There's several variants of this equation, e.g.

$$w_i = w_i + \alpha \cdot (y - \hat{y}) \cdot x_i$$

When trying to understand why they are equivalent, notice that the only possible values for y and $\hat{y}$ are 1 and -1. So if $y = \hat{y}$, then $(y - \hat{y})$ is zero and our update rule does nothing. Otherwise, $(y - \hat{y})$ is equal to either 2 or -2, with the sign matching that of the correct answer (y). The factor of 2 is irrelevant, because we can tune α to whatever we wish.

This training procedure will converge if

- data are linearly separable or
- we throttle the size of the updates as training proceeds by decreasing α.

Apparently convergence is guaranteed if the learning rate is proportional to $\frac{1}{t}$ where t is the current iteration number. The book recommends a gentler formula: make α proportional to $\frac{1000}{1000+t}$.

Processing the training data

One run through training data through the update rule is called an "epoch." Training a linear classifier generally uses multiple epochs, i.e. the algorithm sees each training pair several times.

Datasets often have strong local correlations, because they have been formed by concatenating sets of data (e.g. documents) that each have a strong topic focus. So it's often best to process the individual training examples in a random order.

Limitations

Perceptrons can be very useful but have significant limitations.

- The decision boundary can only be a line.

  Fixes:
  - use multiple units (neural net)
  - massage input features to make the boundary linear

- If there is overlap between the two categories, the learning process can thrash between different boundary positions.

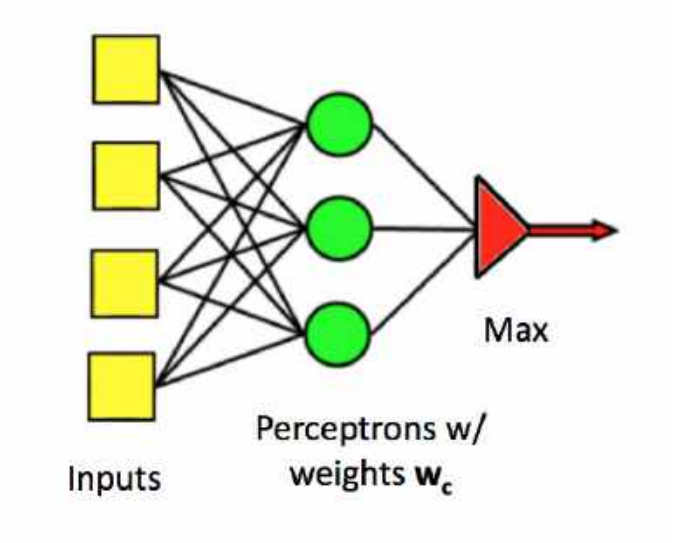
  Fixes:
  - reduce learning rate as learning progresses
  - don't update weights any more than needed to fix mistake in current example
  - cap the maximum change in weights (e.g. this example may have been a mistake)

- If there is a gap between the two categories, the training process may have trouble deciding where to place the boundary line.

Fix: switch to a closely-related learning method called a "support vector machines" (SVM's). The big idea for an SVM is that only examples near the boundary matter. So we try to maximize the distance between the closest sample point and the boundary (the "margin").

Multi-class perceptrons

Perceptrons can easily be generalized to produce one of a set of class labels, rather than a yes/no answer. We use several parallel classifier units. Each has its own set of weights. Join the individual outputs into one output using argmax (i.e. pick the class that has the largest output value).

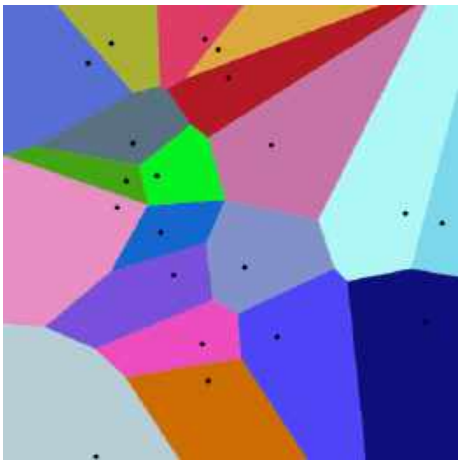


(from Mark Hasegawa-Johnson, CS 440, Spring 2018)

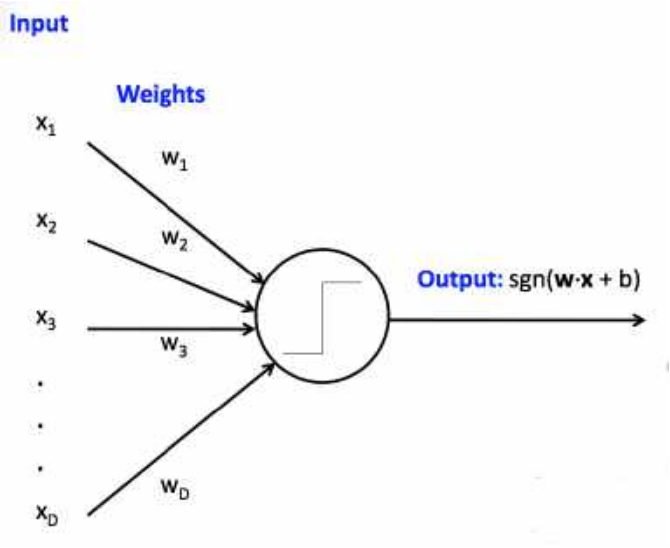
Update rule: Suppose we see (x,y) as a training example, but our current output class is $\hat{y}$. Update rule:

- Do nothing if y and $\hat{y}$ are the same.
- Do nothing to the units for classes other than y and $\hat{y}$.
- Each weight in class y: $w_i = w_i + \alpha \cdot x_i$
- Each weight in class $\hat{y}$: $w_i = w_i - \alpha \cdot x_i$

The classification regions for a multi-class perceptron look like the picture below. That is, each pair of classes is separated by a linear boundary. But the region for each class is defined by several linear boundaries.

 from Wikipedia

A perceptron



(from Mark Hasegawa-Johnson, CS 440, Spring 2018)

Differentiable units

Classical perceptrons have a step function activation function, so they return only 1 or -1. Our accuracy is the count of how many times we got the answer right. When we generalize to multi-layer networks, it will be much more convenient to have a differentiable output function, so that we can use gradient descent to update the weights.

To do this, we need to separate two functions at the output end of the unit:

- the activation function: our model of uncertainty about the decision
- the loss function: the penalty for getting the decision wrong.

For the perceptron, the activation function is the sign function function: 1 if above the threshold, -1 if below it. The loss function for a perceptron is "zero-one loss". We get 1 for each wrong answer, 0 for each correct one.

So our whole classification function is the composition of three functions:

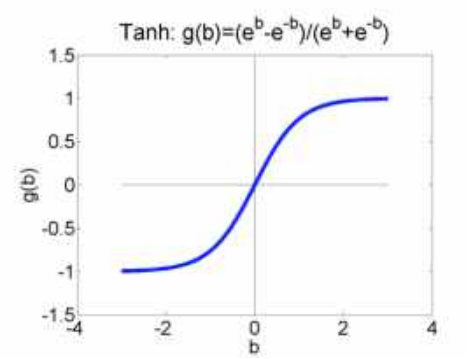
- the weighted sum of input feature values
- the activation function
- the loss function

In a multi-layer network, there will be activation functions at each layer and one loss function at the very end.

Activation Functions

When we move to differentiable units for making neural nets, the popular choices for differentiable activation functions include the logistic (aka sigmoid), tanh, rectified linear unit (ReLu). Wikipedia has an excessively comprehensive list of alternatives.

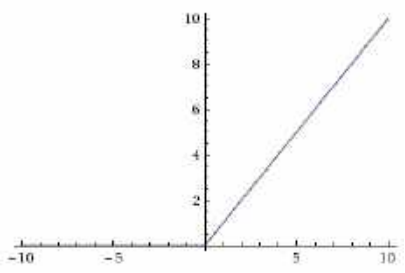
The tanh function $\frac{e^x - e^{-x}}{e^x + e^{-x}}$ looks like this:



from Mark Hasegawa-Johnson

The logistic/sigmoid function has equation $\frac{1}{(1+e^{-x})}$. It is just like tanh, except rescaled so that it returns values between 0 and 1 (rather than between -1 and 1). The logistic form would be used when the designer wants to think of the values as probabilities (e.g. logistic regression). Otherwise one would choose whichever is more convenient for later stages of processing.

A rectified linear unit (ReLU) uses the function f, where f(x) = x when positive, f(x) = 0 if negative. It looks like this:


from Andrej Karpathy

The activation does not affect the decision from our single, pre-trained unit. Rather, the choice has two less obvious consequences:

- The output of the logistic function is in the right range to be interpreted as a probability, if that is desired.
- A slow transition across the decision boundary lets us treat values near the decision boundary as uncertain, both when training our unit and when feeding its value to later processing (e.g. in a neural net).

Loss Functions

Suppose that we have a training pair (x,y). So y is the correct output. Suppose that $\hat{y}$ is the output of our unit (weighted average and then activation function). Popular differentiable loss functions include

- L2 norm: $(y - \hat{y})^2$
- cross-entropy loss: $-(y \log \hat{y} + (1 - y) \log(1 - \hat{y}))$

In both cases, our goal is to minimize this loss averaged over all our training pairs.

When using the L2 norm to measure distances (e.g. in a 2D map), one would normally

- take the square of the difference for each individual coordinate (as shown above),
- add the squares up across all coordinates, and then
- take the square root of the result.

However, we're just trying to minimize the distance, not actually compute the distance. So we can skip the last (square root) step, because square root is monotonically increasing.

Recall that entropy is the number of bits required to (optimally) compress a pool of values. The high-level idea of cross-entropy is that we have built a model of probabilities (e.g. using some training data) and then use that model to compress a different set of data. Cross-entropy measures how well our model succeeds on this new data. This is one way to measure the difference between the model and the new data.

If you are curious, here is a longer list of loss functions.

Review: the chain rule

Remember the chain rule from calculus. If $h(x) = f(g(x))$, the chain rule says that $h'(x) = f'(g(x)) \cdot g'(x)$. In other words, to compute $h'(x)$, we'll need the derivatives of the two individual functions, plus the value of of them applied to the input.

Example of differentiable updating

Let's assume we use the logistic/sigmoid for our activation function and the L2 norm for our loss function.

To work out the update rule for learning weight values, suppose we have a training example (x,y), where x is a vector of feature values. The output with our current weights is

$\hat{y} = \sigma(w \cdot x)$ where σ is the sigmoid function.

Using the L2 norm as our loss function, our loss from just this one training sample is

$$L(w, x) = (y - \hat{y})^2$$

For gradient descent, we adjust our weights in the opposite of the gradient direction. That is, our update rule for one weight w_i should look like:

$$w_i = w_i - \alpha \cdot \frac{\partial L(w,x)}{\partial w_i}$$

Let's figure out what $\frac{\partial L(w,x)}{\partial w_i}$ is. This requires some patience but is entirely mechanical. It's a calculation that you'll typically ask a software package (e.g. neural net utilities) to do for you.

$\frac{\partial L(w,x)}{\partial w_i} = \frac{\partial (y-\hat{y})^2}{\partial w_i}$ (definition of L(w,x))
$= 2(y - \hat{y}) \cdot \frac{\partial (y-\hat{y})}{\partial w_i}$ (chain rule)
$= 2(y - \hat{y}) \cdot \frac{\partial (-\hat{y})}{\partial w_i}$ (y doesn't depend on w_i) $= -2(y - \hat{y}) \cdot \frac{\partial \hat{y}}{\partial w_i}$ (y doesn't depend on w_i)

OK, so now what is $\frac{\partial \hat{y}}{\partial w_i}$? If we look up the derivative of the sigmoid function, we find that $\frac{\partial \sigma(z)}{\partial z} = \sigma(z)(1 - \sigma(z))$. Using this fact:

$\frac{\partial \hat{y}}{\partial w_i} = \frac{\partial \sigma(w \cdot x)}{\partial w_i}$ (definition of $\hat{y}$)
$= \sigma(w \cdot x) \cdot (1 - \sigma(w \cdot x)) \cdot \frac{\partial w \cdot x}{\partial w_i}$ (chain rule and the derivative of σ)
$= \sigma(w \cdot x) \cdot (1 - \sigma(w \cdot x)) \cdot x_i$ (it's just a polynomial)
$= \hat{y} \cdot (1 - \hat{y}) \cdot x_i$ (definition of $\hat{y}$)

Substituting this into our earlier formula, we get

$$\frac{\partial L(w,x)}{\partial w_i} = -2 \cdot (y - \hat{y}) \cdot \hat{y} \cdot (1 - \hat{y}) \cdot x_i$$

So our update formula is

$$w_i = w_i + 2\alpha \cdot (y - \hat{y}) \cdot \hat{y} \cdot (1 - \hat{y}) \cdot x_i$$

α is just a tuning parameter, so we can fold 2 into it. This gives us our final update equation:

$$w_i = w_i + \alpha \cdot (y - \hat{y}) \cdot \hat{y} \cdot (1 - \hat{y}) \cdot x_i$$

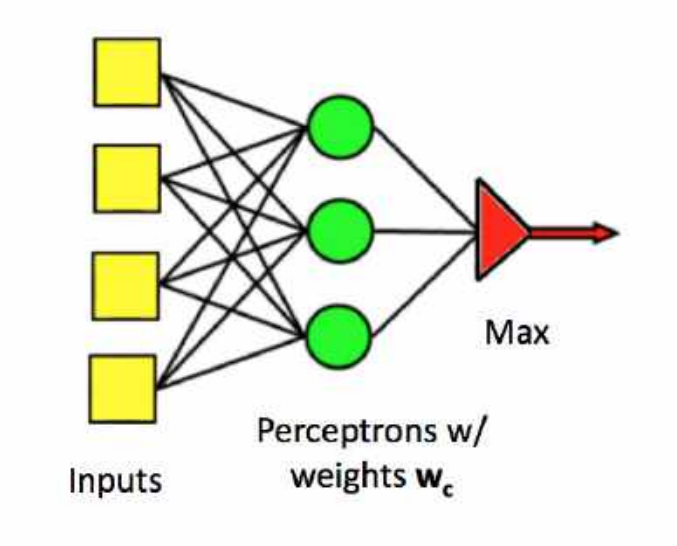
Ugly, unintuitive, but straightforward to code.

Recap

We've seen how to define and train a single differentiable unit. It's useful to know about a couple more details before we move on to looking at multi-layer networks formed from these units.

Softmax

Recall how we built a multi-class perceptron. We fed each input feature to all of the units. Then we joined their outputs together with an argmax to select the class with the highest output.



(from Mark Hasegawa-Johnson, CS 440, Spring 2018)

We'd like to do something similar with differentiable units.

There is no (reasonable) way to produce a single output variable that takes discrete values (one per class). So we use a "one-hot" representation of the correct output. A one-hot representation is a vector with one element per class. The target class gets the value 1 and other classes get zero. E.g. if we have 8 classes and we want to specify the third one, our vector will look like:

```
[0, 0, 1, 0, 0, 0, 0, 0]
```

One-hot doesn't scale well to large sets of classes, but works well for medium-sized collections.

Now, suppose we have a sequence of classifier outputs, one per class. $v_1, \ldots, v_n$. We often have little control over the range of numbers in the v_i values. Some might even be negative. We'd like to map these outputs to a one-hot vector. Softmax is a differentiable function that approximates this discrete behavior. It's best thought of as a version of argmax.

Specifically, softmax maps each v_i to $\frac{e^{v_i}}{\sum_i e^{v_i}}$. The exponentiation accentuates the contrast between the largest value and the others, and forces all the values to be positive. The denominator normalizes all the values into the range [0,1], so they look like probabilities (for folks who want to see them as probabilities).

Regularization

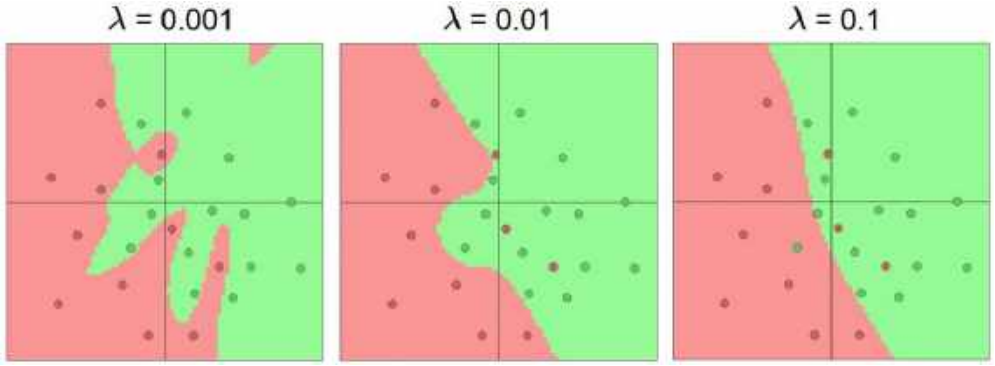
Any type of model fitting is subject to possible overfitting. When this seems to be creating problems, a useful trick is "regularization." Suppose that our set of training data is T. Then, rather than minimizing loss(model,T), we minimize

loss(model,T) + λ * complexity(model)

The right way to measure model complexity depends somewhat on the task. A simple method for a linear classifier would be $\sum_i (w_i)^2$, where the w_i are the classifier's weights. Recall that squaring makes the L2 norm sensitive to outlier (aka wrong) input values. In this case, this behavior is a feature: this measure of model complexity will pay strong attention to unusually

large weights and make the classifier less likely to use them. Linguistic methods based on "minimumum description length" are using a variant of the same idea.

λ is a constant that balances the desire to fit the data with the desire to produce a simple model. This is yet another parameter that would need to be tuned experimentally using our development data. The picture below illustrates how the output of a neural net classifier becomes simpler as λ is increased. (The neural net in this example is more complex, with hidden units.)

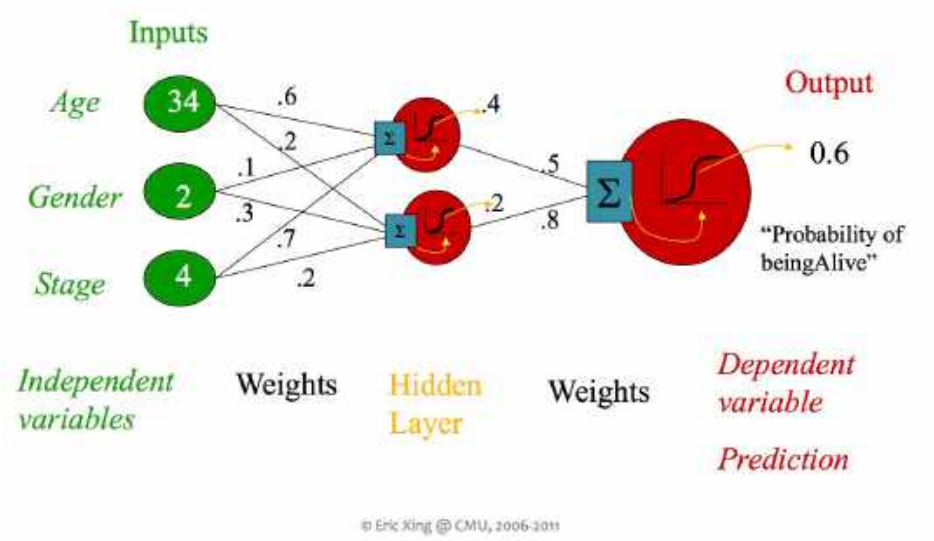


from Andrej Karpathy course notes

13 Neural Nets

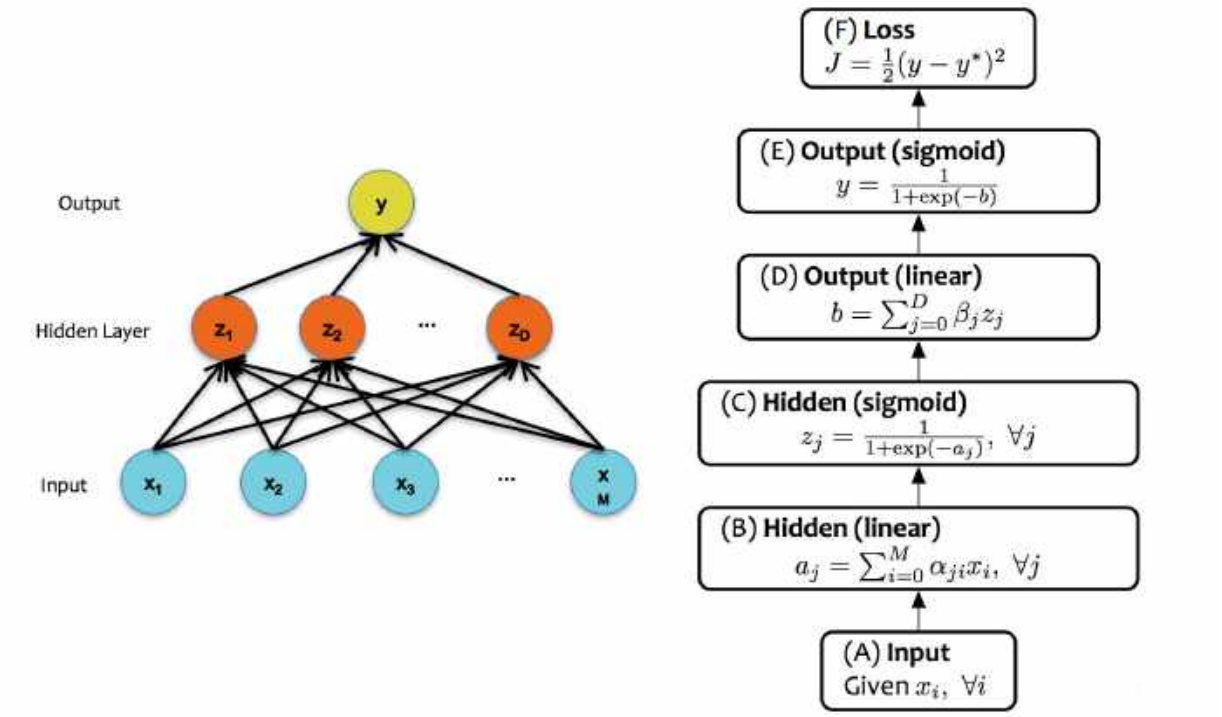
Hidden layers

So far, we've seen some systems containing more than one classifier unit, but each unit was always directly connected to the inputs and an output. When people say "neural net," they usually mean a design with more than layer of units, including "hidden" units which aren't directly connected to the output. Early neural nets tended to have at most a couple hidden layers (as in the picture below). Modern designs typically use many layers.



from Eric Xing via Matt Gormley

A single unit in a neural net can look like any of the linear classifiers we've seen previously. Moreover, different layers within a neural net design may use types of units (e.g. different activation functions). However, modern neural nets typically use entirely differentiable functions, so that gradient descent can be used to tune the weights.



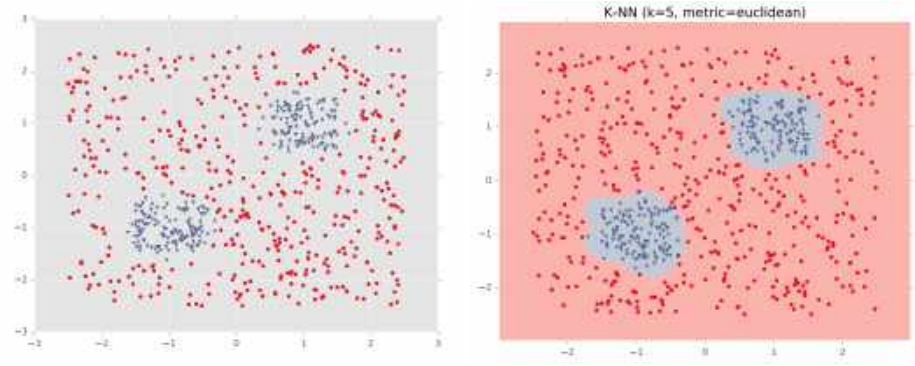
from Matt Gormley

Notice two things about this design. First, each layer has its own activation function, but there is only one loss function, at the very end. Second, the activation functions are non-linear. If we had linear activation functions, we could squash the whole network down into one linear function, so we'd be back to having only linear classification boundaries.

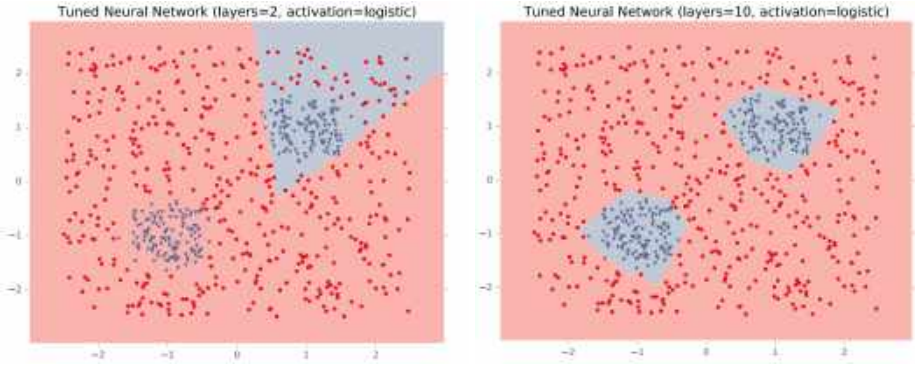
The core ideas in neural networks go back decades. Some of their recent success is due to better understanding of design and/or theory. But perhaps the biggest reasons have to do with better computer hardware and better library support for the annoying mechanical parts of the computation.

Approximating functions

In theory, a single hidden layer is sufficient to approximate any continuous function, assuming you have enough hidden units. However, shallow networks may require very large numbers of hidden units. Deeper neural nets seem to be easier to design and train. Here's an example (from Matt Gormley) of a class boundary with a somewhat complex shape. K-nearest neighbor does a good job of approximating it (but is inefficient in higher dimensions).

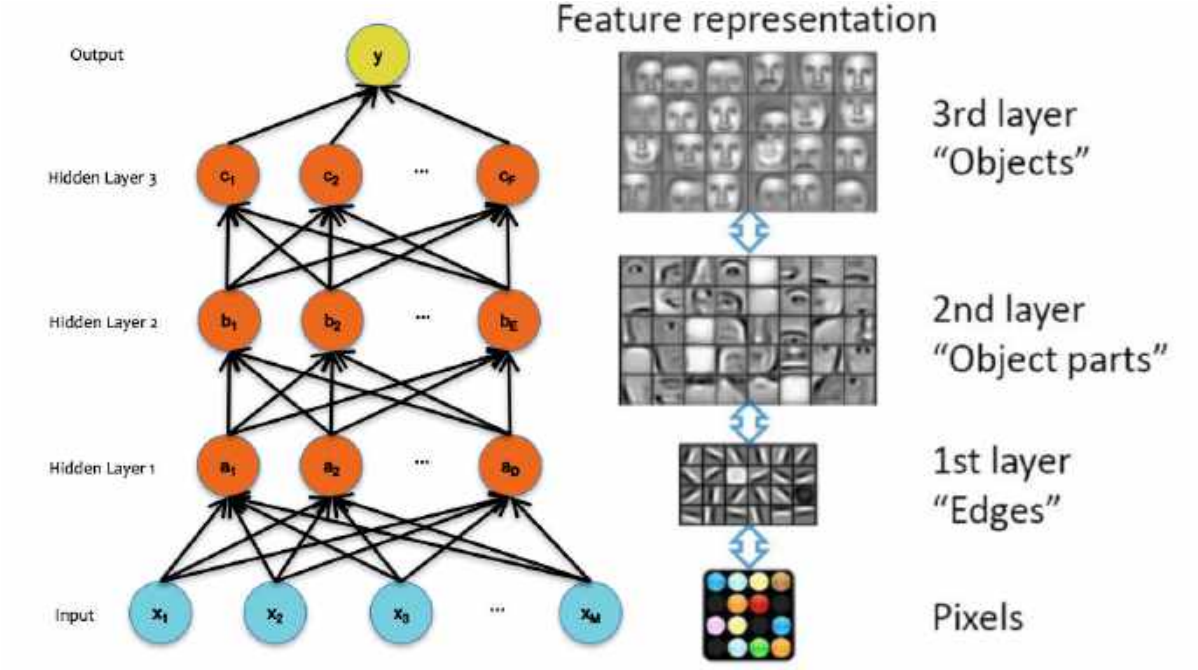


A two-layer network gives us a poor approximation (left), but a ten-layer network does a good job (right). Notice that our final network model is compact: proportional to the number of weights int the network. The size of k-nearest neighbor model grows with the number of training samples.



Deep networks

In image processing applications, deep networks normally emulate the layers of processing found in previous human-designed classifiers. So each layer transforms the input so as to make it more sophisticated ("high level"), compacts large inputs (e.g. huge pictures) into a more manageable set of features, etc. The picture below shows a face recognizer in which the bottom units detect edges, later units detect small pieces of the face (e.g. eyes), and the last level (before the output) finds faces.



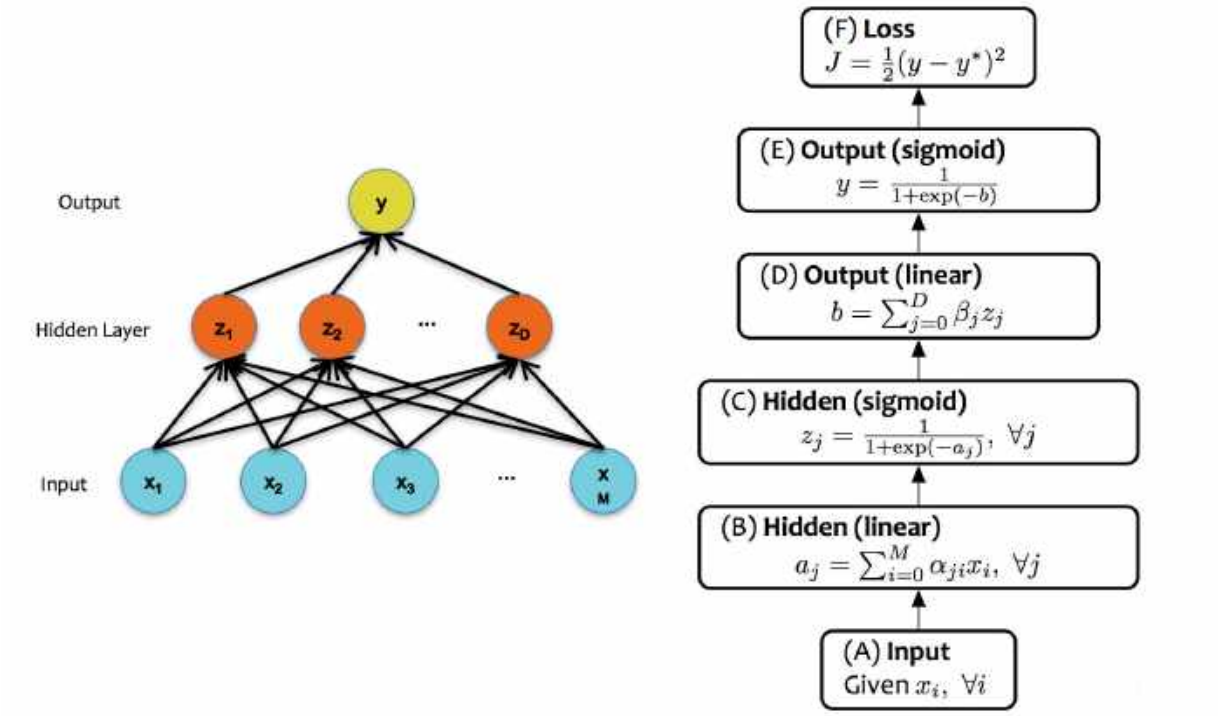
(from Matt Gormley)

Here are layers 1, 3, and 5 from a neural net trained on ImageNet (from M. Zeiler and R. Fergus [Visualizing and Understanding Convolutional Networks]).

Notes

Credits to Matt Gormley are from [10-601 CMU 2017]

A 2-layer network



from Matt Gormley

Training: high level idea

Neural nets are trained in much the same way as individual classifiers. That is, we initialize the weights, then sweep through the input data multiple types updating the weights. When we see a training pair, we updated the weights, moving each weight w_i in the direction that decreases the loss (J):

$$w_i = w_i - \alpha * \frac{\partial J}{\partial w_i}$$

Notice that the loss is available at the output end of the network, but the weight w_i might be in an early layer. So the two are related by a chain of composed functions, often a long chain of functions. We have to compute the derivative of this composition.

Review: the chain rule

Remember the chain rule from calculus. If $h(x) = f(g(x))$, the chain rule says that $h'(x) = f'(g(x)) \cdot g'(x)$. In other words, to compute $h'(x)$, we'll need the derivatives of the two individual functions, plus the value of of them applied to the input.

Let's write out the same thing in Liebniz notation, with some explicit variables for the intermediate results:

- y = g(x)
- z = f(y) = f(g(x))
- chain rule: $\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$

This version of the chain rule now looks very simple: it's just a product. However, we need to remember that it depends implicitly on the equations defining y in terms of x, and z in terms of y. Let's use the term "forward values" for values like y and z.

We can now see that the derivative for a whole composed chain of functions is, essentially, the product of the derivatives for the individual functions. So, for a weight w at some level of the network, we can assemble the value for $\frac{\partial J}{\partial w}$ using

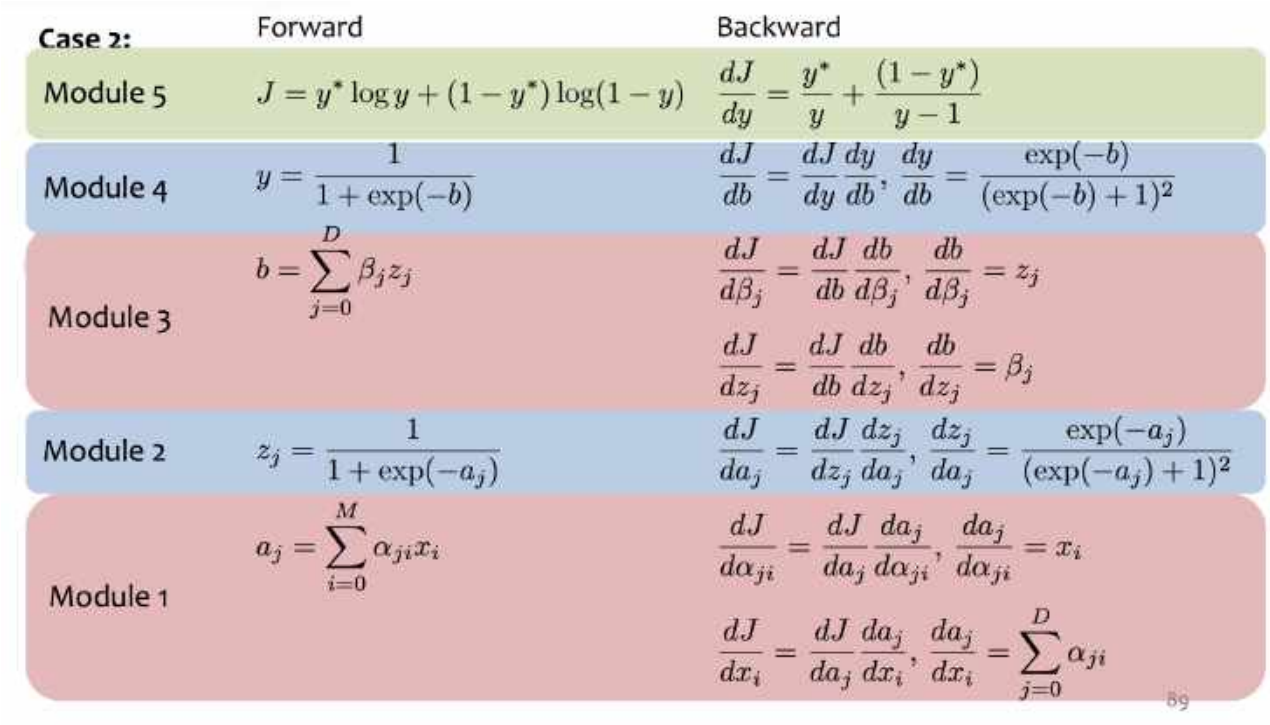
- the derivative of each individual function
- the chain rule: multiply those derivatives
- the forward values: substitute to eliminate all the intermediate variables

Neural net update

Now, suppose we have a training pair $(\vec{x}, y)$. (That is, y is the correct class for $\vec{x}$.) Updating the neural net weights happens as follows:

- Forward pass: starting from the input $\vec{x}$, calculate the output values for all units. We work forwards through the network, using our current weights.
- Suppose the final output is y^*. We put y and y^* into our loss function to calculate the final loss value.
- Backpropagation: starting from the loss at the end of the network and working backwards, calculate how changes to each weight will affect the loss. This combines the derivatives of each individual function with the values from the forward pass.

The diagram below shows the forward and backward values for our example network:



(from Matt Gormley)

Backpropagation is essentially a mechanical exercise in applying the chain rule repeatedly. Humans make mistakes, and direct manual coding will have bugs. So, as you might expect, computers have taken over most of the work as they for (say) register allocation. Read the very tiny example in Jurafsky and Martin (7.4.3 and 7.4.4) to get a sense of the process, but then assume you'll use TensorFlow or PyTorch to make this happen for a real network.

Three challenges in training

Unfortunately, training neural nets is somewhat of a black art because the process isn't entirely stable. Three issues are prominent:

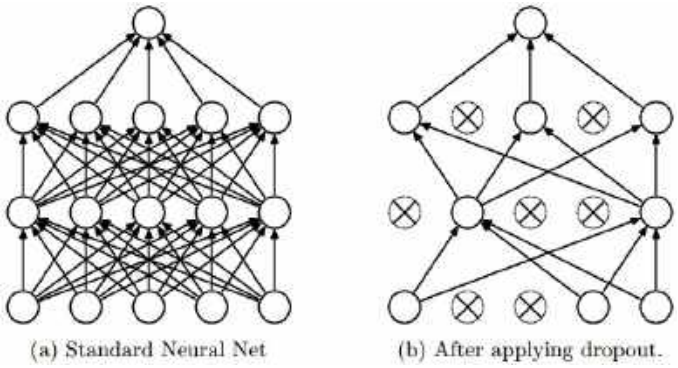
- Symmetry breaking
- Overfitting
- Vanishing/exploding gradients

Symmetry Breaking

Perceptron training works fine with all weights initialized to zero. This won't work in a neural net, because each layer typically has many neurons connected in parallel. We'd like parallel units to look for complementary features but the naive training algorithm will cause them to have identical behavior. At that point, we might as well economize by just having one unit. Two approaches to symmetry breaking:

- Set the initial weights to random values.
- Randomize some aspect of the later training, e.g. ignore some randomly-chosen units on each update.

One specific proposal for randomization is **dropout**: Within the network, each unit pays attention to training data only with probability p. On other training inputs, it stops listening and starts reading its email or something. The units that aren't asleep have to classify that input on their own. This can help prevent overfitting.



(a) Standard Neural Net (b) After applying dropout.

from Srivastava et al.

Overfitting

Neural nets infamously tend to tune themselves to peculiarities of the dataset. This kind of overfitting will make them less able to deal with similar real-world data. The dropout technique will reduce this problem. Another method is "data augmentation".

Data augmentation tackles the fact that training data is always very sparse, but we have additional domain knowledge that can help fill in the gaps. We can make more training examples by perturbing existing ones in ways that shouldn't (ideally) change the network's output. For example, if you have one picture of a cat, make more by translating or rotating the cat. See this paper by Taylor and Nitschke.

Vanishing/exploding gradients

In order for training to work right, gradients computed during backprojection need to stay in a sensible range of sizes. A sigmoid activation function only works well when output numbers tend to stay in the middle area that has a significant slope.

- gradients too small: numbers can underflow, also training can become slow
- gradients too large: numbers can overflow

The underflow/overflow issues happen because numbers that are somewhat too small/large tend to become smaller/larger.

Several approaches to mitigating this problem, none of which looks (to me) like a solid, complete solution.

- ReLU units are less prone to these problems. However, they stop training if inputs force their outputs negative. So people often use a "leaky ReLU" function which has a very small slope on the negative side, e.g. f(x) = x for positive inputs, f(x) = 0.01x for negative ones.
- Initialize weights so that different layers end up with the same variance in gradients. For example, the variance may be set proportional to $\frac{1}{N_{in}}$ or $\frac{2}{N_{in}+N_{out}}$ where N_{in} is the number of incoming connections for this layer and N_{out} is the number of outgoing connections. There are several variants on this idea, e.g. Xavier and Kaiming initialization.
- Gradient clipping: detect excessively high gradients and reduce them.
- Weight regularization: many of the problematic situations involve excessively large weights. So add a regularization term to the network's loss function that measures the size of the weights (e.g. sum of the squares or magnitudes of the weights).

Notes

Credits to Matt Gormley are from 10-601 CMU 2017

Convolutional neural nets

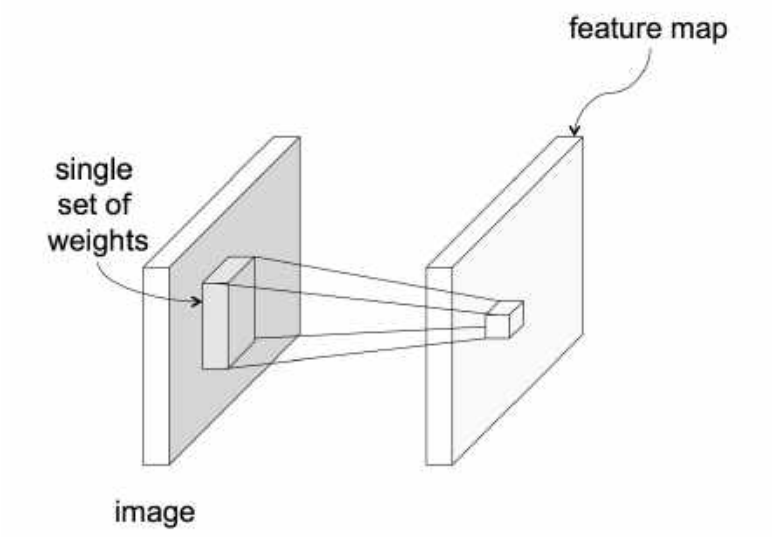
Convolutional neural nets are a specialized architecture designed to work well on image data (also apparently used somewhat for speech data). Images have two distinctive properties:

- They are large, e.g. 1000 by 1000 pixels.
- They have spatial coherence.

We'd like to be able to do some processing at high-resolution, to pick out features such as written text. But other objects (notably faces) can be recognized with only poor resolution. So we'd like to use different amounts of resolution at different stages of processing.

The large size of each layer makes it infeasible to connect units to every unit in the previous layer. Full interconnection can be done for artificially small (e.g. 32x32) input images. For larger images, this will create too many weights to train effectively with available training data. For physical networks (e.g. the human brain), there is also a direct hardware cost for each connection.

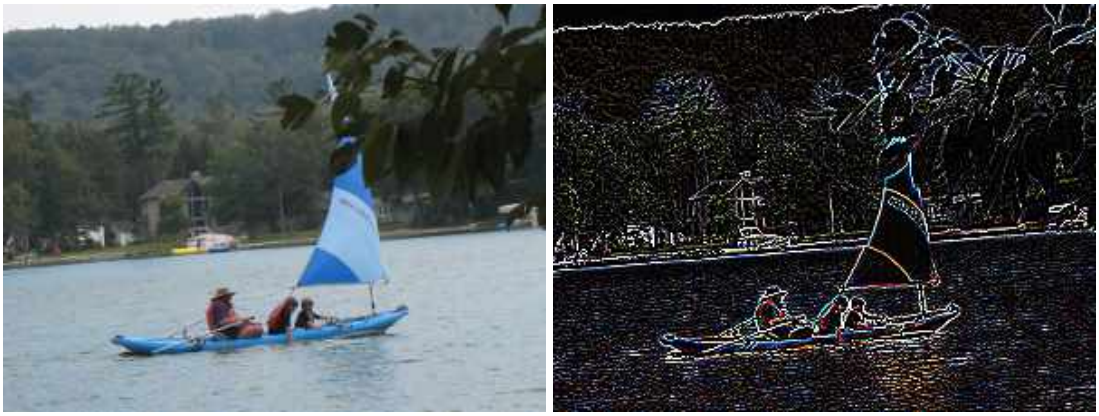
In a CNN, each unit reads input only from a local region of the preceding layer:



from Lana Lazebnik Fall 2017

This means that each unit computes a weighted sum of the values in that local region. In signal processing, this is known as "convolution" and the set of weights is known as a "mask." For example, the following mask will locate sharp edges in the image.

```
 0   -4    0
-4   16   -4
 0   -4    0
```



The following mask detects horizontal edges but not vertical ones.

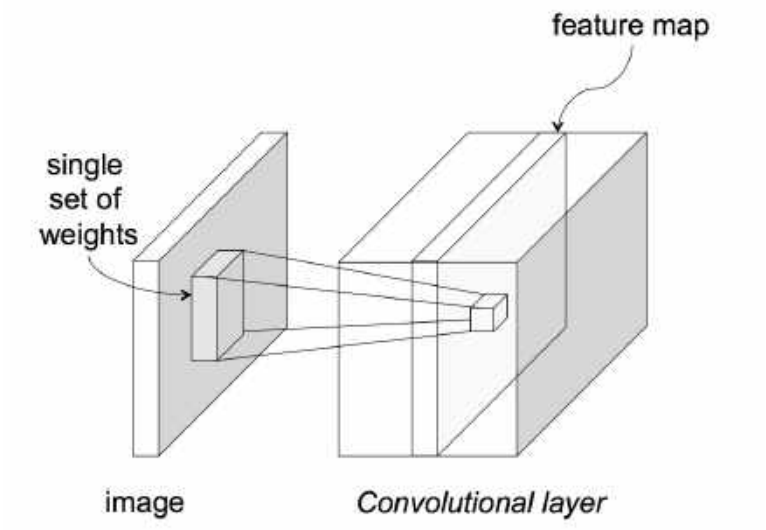
```
 2    4    8    4    2
 0    0    0    0    0
-2   -4   -8   -4   -2
```


These examples were made with gimp (select filters, then generic).

Normally, all units in the same layer would share a common set of weights and bias terms, called "parameter sharing". This reduces the number of parameters we need to train. However, parameter sharing may worsen performance if different regions in the input images are expected to have different properties, e.g. the object of interest is always centered. (This might be true in a security application where people were coming through a door.) So there are also neural nets in which separate parameters are trained for each unit in a convolutional layer.

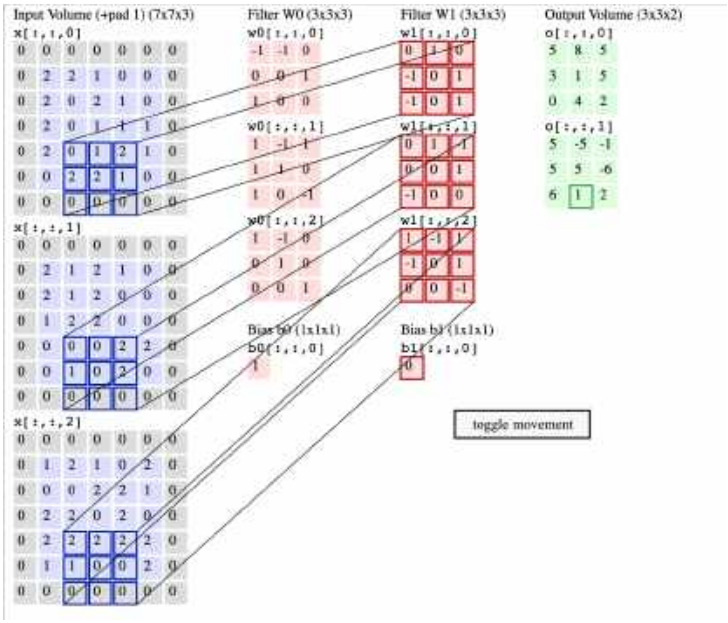
Convolutional layer

The above picture assumes that each layer of the network has only one value at each (x,y) position. This is typically not the case. An input image often has three values (red, green, blue) at each pixel. Going from the input to the first hidden layer, one might imagine that a number of different convolution masks would be useful to apply, each picking out a different type of feature. So, in reality, each network layer has a significant thickness, i.e. a number of different values at each (x,y) location.



from Lana Lazebnik Fall 2017

Click on the image below to see an animation from Andrej Karpathy shows how one layer of processing might work:



- blue: a 7x7 input image (3 color values at each location)
- red: the weights for two classifier units
- green: the outputs of the two units

In this example, each unit is producing values only at every third input location. So the output layer is a 3x3 image, with has two values at each (x,y) position.

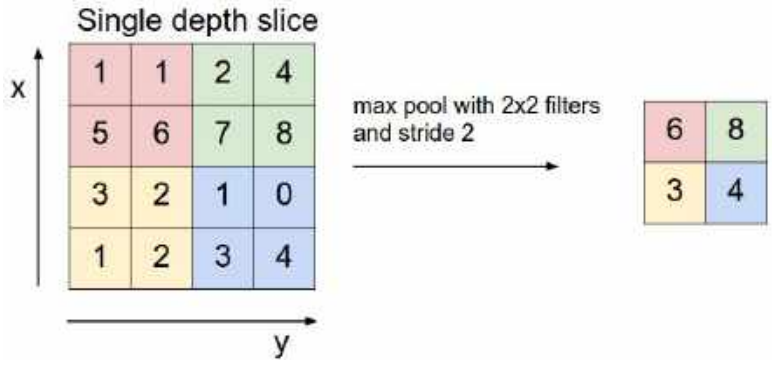
Two useful bits of jargon:

- "stride" = how many pixels we shift mask sideways between output units
- "depth" = how many stacked features in each level

If we're processing color images, the initial input would have depth 3. But the depth might get significantly larger if we are extracting several different types of features, e.g. edges in a variety of orientations.

Pooling

A third type of neural net layer reduces the size of the data, by producting an output value only for every kth input value in each dimension. This is called a "pooling" layer. The output values may be either selected input values, or the average over a group of inputs, or the maximum over a group of inputs.



from [Andrej Karpathy]

This kind of reduction in size ("downsampling") is especially sensible when data values are changing only slowly across the image. For example, color often changes very slowly except at object boundaries, and the human visual system represents color at a much lower resolution than brightness.

A pooling layer that chooses the maximum value can be very useful when we wish to detect high-resolution features but don't care about their precise location. E.g. perhaps we want to report that we found a corner, together with its approximate

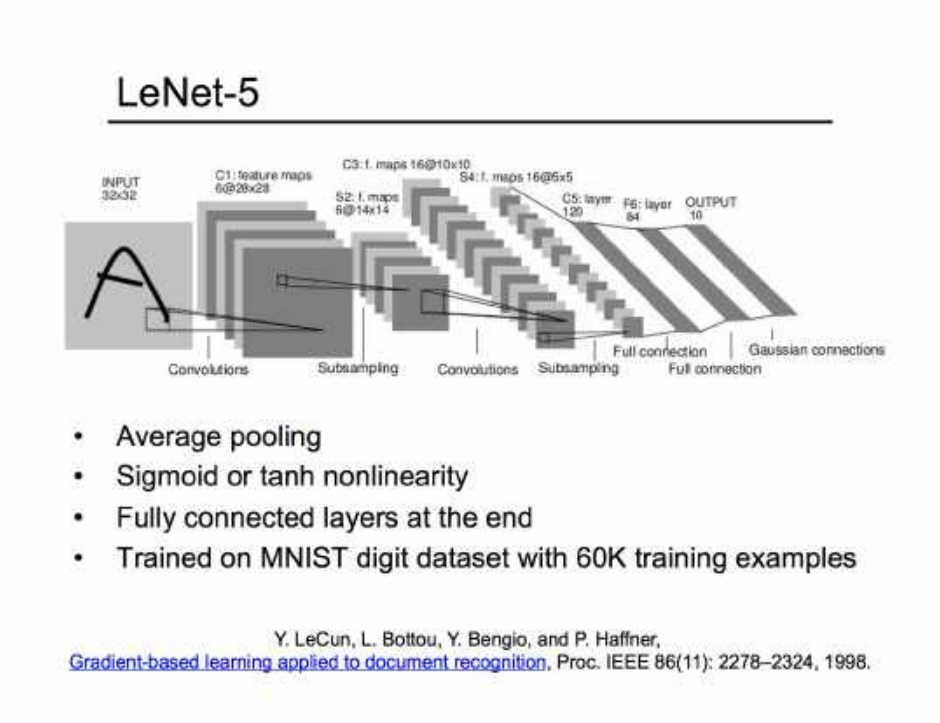
location.

Architecture

A complete CNN typically contains three types of layers

- convolutional
- pooling
- fully-connected

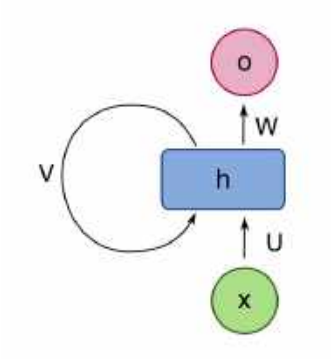
Convolutional layers would typically be found in the early parts of the network, where we're looking at large amounts of image data. Fully-connected layers would make final decisions towards the end of the process, when we've reduced the image data to a smallish set of high-level features.



The specific network in this picture is from 1998, when neural nets were just starting to be able to do something useful. Its input is very small, black and white, pictures of handwritten digits. The original application was reading zip codes for the post office.

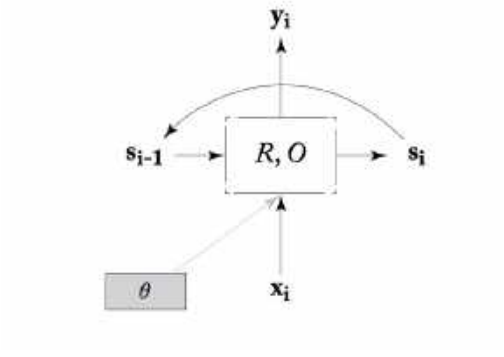
Recurrent Neural Nets

Recurrent neural nets (RNNs) are neural nets that have connections that loop back from a layer to the same layer. The RNN shown below has a single hidden layer. We can think of the self-connected layer as containing multiple processing units, similar to the hidden layer of a normal neural network. The intent of the feedback loop in the picture is that each unit is connected to all the other units in the layer.



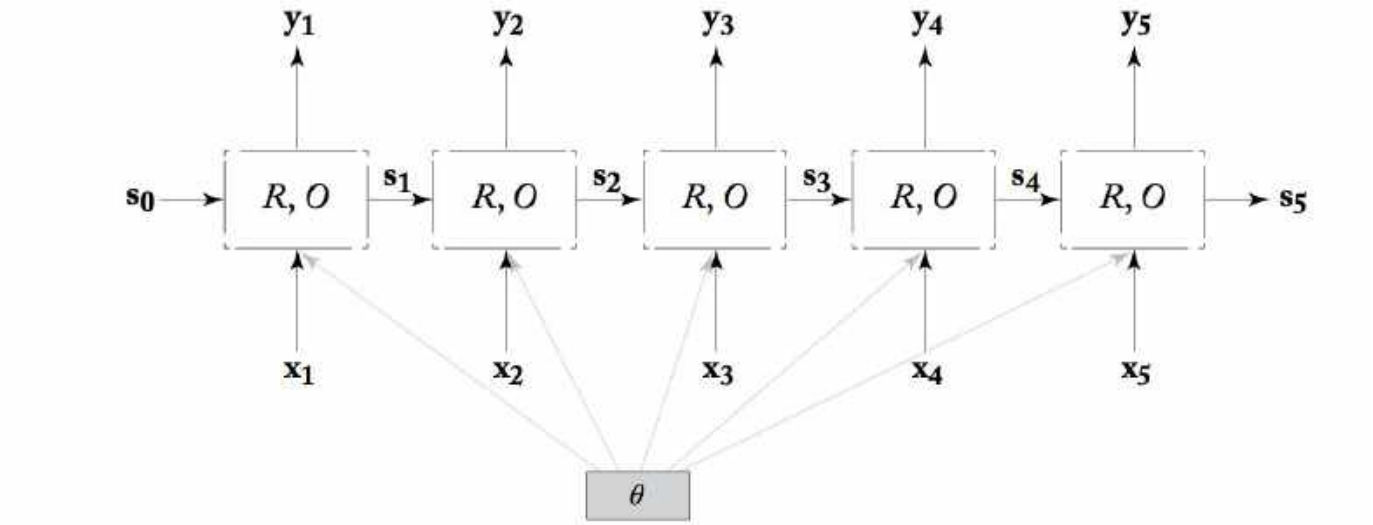
from Wikipedia

Alternatively, we can think of an RNN as a state machine. That is, we think of the values from all the units in the layer as bundled up into a state vector, s_i in the picture below. At each timestep, the RNN reads an input vector and the current state. It produces an output vector and a new state, using a state transition function (R), an output function (O), and some tunable parameters θ.



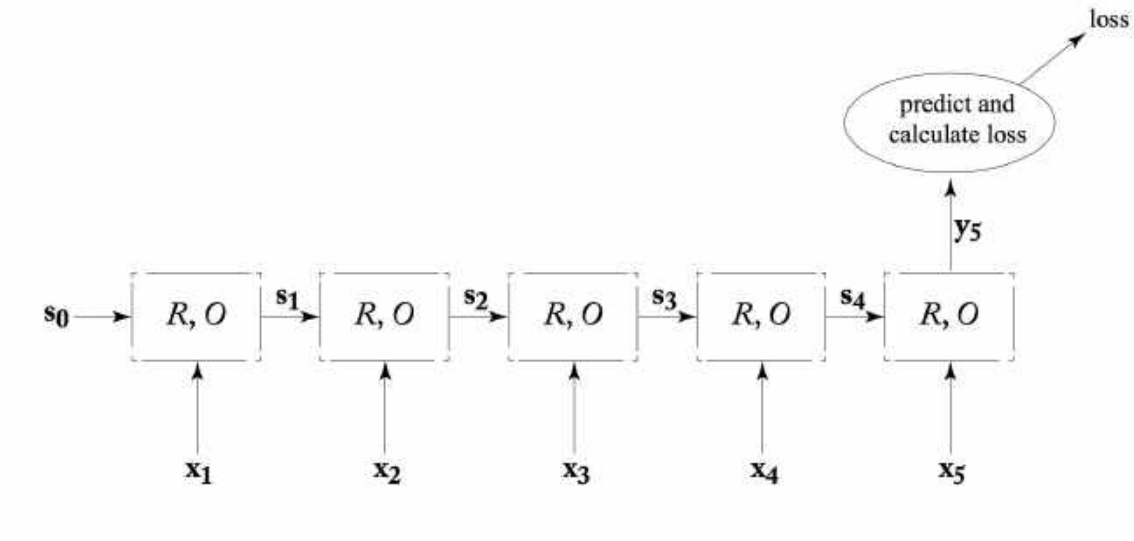
from Yoav Goldberg

To see how computation proceeds and, more importantly, how training works, we unroll the RNN. That is, we make a clone of the RNN for each timestep, creating the diagram below. Notice that all copies of the unit share the same parameter values.



from Yoav Goldberg

An RNN can be used as a classifier. That is, the system using the RNN only cares about the output from the last timestep, as shown below. In an NLP system, the final output value may actually be complex, e.g. a summary of an entire sentence. Either way, the final output value is fed into a later processing that provides feedback about its performance (the loss value).



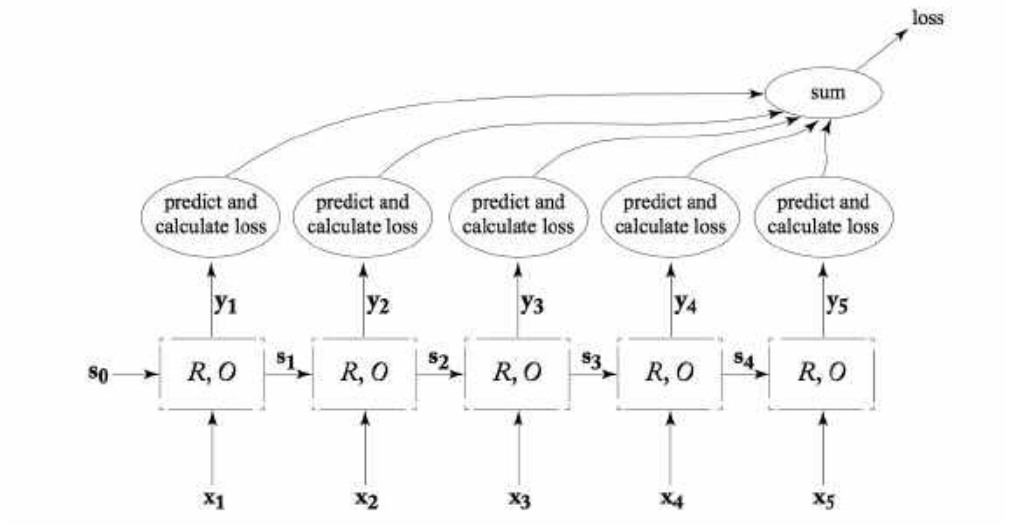
from Yoav Goldberg

This kind of RNN can be trained in much the same way as a standard ("feedforward") neural net. However, values and error signals propagate in the time direction. The forward pass calculates values moving to the right. Backpropagation starts at the final loss node and moves back to the left. This is often called "backpropagation through time."

Like convolutional layers, RNNs rarely do an entire task by themselves. They are typically combined with other processing layers, either other types of neural net machinery or non-neural processing.

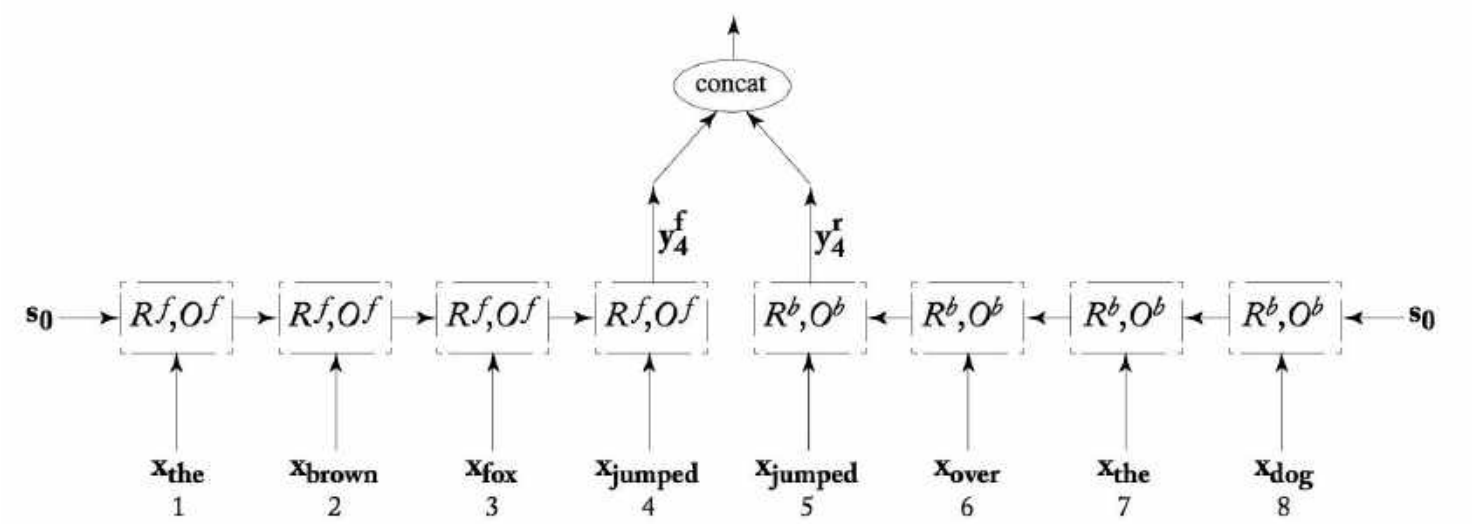
Other types of RNNs

RNNs have proved most useful for modelling sequential data, which is common in natural language understanding and generation tasks. Some of these tasks use variations on the classifier design shown above. For example, many tasks require mapping the input sequence into a corresponding output sequence, e.g. mapping words to part-of-speech tags. In this case, we would care about getting the correct output at each timestep (not just the final one). So the connection to our loss function would look like this:



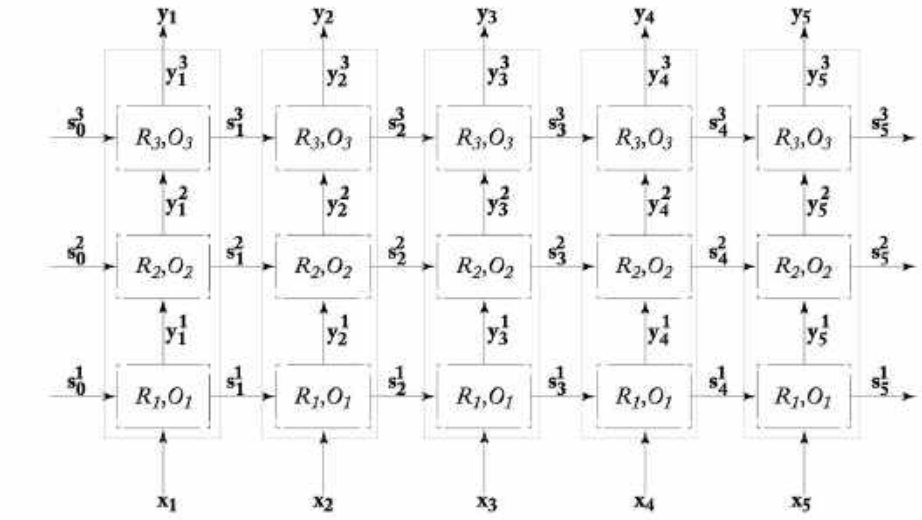
from Yoav Goldberg

We can join two RNN's together into a "bidirectional RNN." One RNN works forwards from the start of the input, the second works backwards from the end of the input. The output at each position is the concatenation of the two RNN outputs. This combined output would typically receive further processing (not show) and then eventually be evaluated to produce a loss. When the loss information propagaes backwards, both RNNs receive feedback based on their joint answer.



from Yoav Goldberg

We can also build "deep" RNN's, which have more than one processing layer between the input and output streams. The one shown below has three layers. Because each processing unit in an RNN is already complex (equivalent to multiple units in a standard neural net), it's unusual to see many layers.



from Yoav Goldberg

Gated RNNs

In theory, an RNN can remember the entire stream of input values. However, gradient magnitude tends to decay as you move backwards from the loss signal. So earlier inputs may not contributed much to the RNN's final answer at the end. For a transducer, inputs may contribute little to the output for locations some distance away. This is a problem, because many linguistic tasks benefit from an extended context.

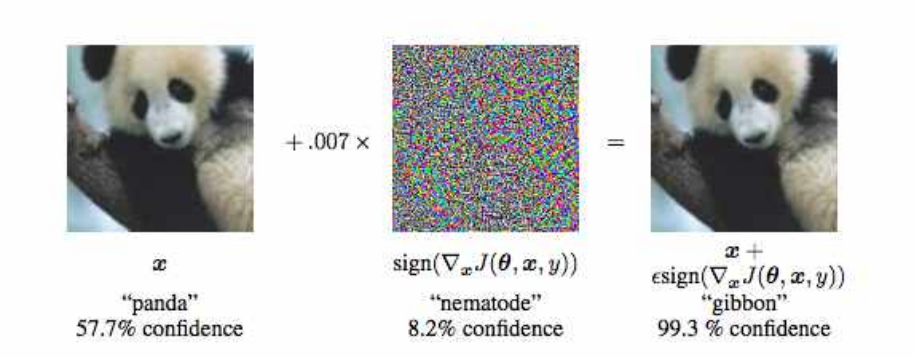
To allow RNNs to store information more effectively, researchers use "gated" versions of RNNs. These RNNs include separate vectors (the "gates") which control which parts of the unit's state will be updated at each timestep. The gates make the RNN's behavior easier to control, but create yet more tunable parameters that must be learned. Two popular gated RNN models are the "Long Short-Term Memory" (LSTM) and the "Gated Recurrent Unit" (GRU).

Notes

Figures credited to Yoav Goldberg are from his book "Neural Network Methods for Natural Language processing." (If you're on the U. Illinois VPN, you can download it for free.)

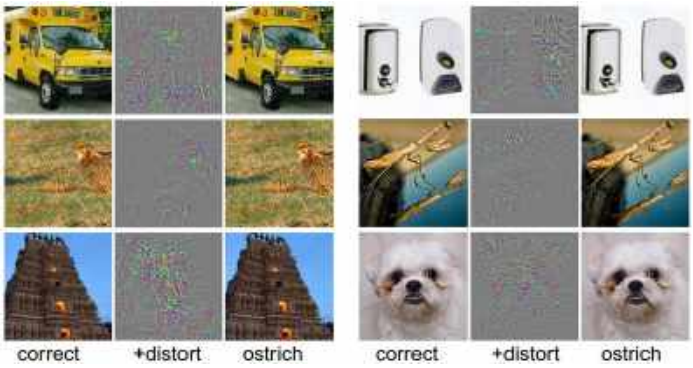
Adversarial Examples

Current training procedures for neural nets still leave them excessively sensitive to small changes in the input data. So it is possible to cook up patterns that are fairly close to random noise but push the network's values towards or away from a particular output classification. Adding these patterns to an input image creates an "adversarial example" that seems almost identical to a human but gets a radically different classification from the network. For example, the following shows the creation of an image that looks like a panda but will be misrecognized as a gibbon.



from Goodfellow et al

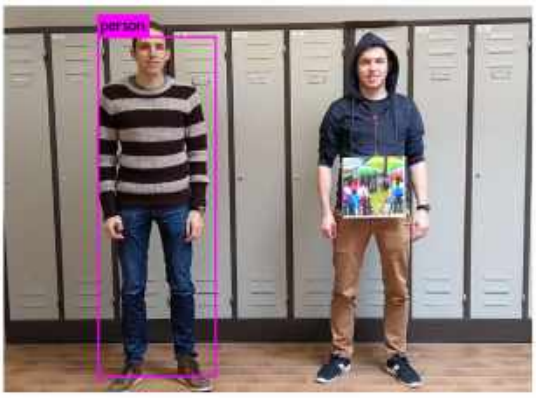
The pictures below show patterns of small distortions being used to persuade the network that images from six different types are all ostriches.



from Szegedy et al.

These pictures come from Andrej Karpathy's blog, which has more detailed discussion.

Clever patterns placed on an object can cause it to disappear, e.g. only the lefthand person is recognized in the picture below.



from Thys, Van Ranst, Goedeme 2019

Disturbingly, the classifier output can be changed by adding a disruptive pattern near the target object. In the example below, a banana is recognized as a toaster.

from [Brown, Mane, Roy, Abadi, Gilmer, 2018]

And here are a couple more examples of fooling neural net recognizers:

- [The elephant in the room]
- [Fooling a recognizer using subtle makeup]

In the words of one researcher (David Forsyth), we need to figure out how to "make this nonsense stop" without sacrificing accuracy or speed. This is currently an active area of research.

NLP Adversarial Examples

Similar adversarial examples can be created purely with text data. In the examples below, the output of a natural language classifier can be changed by replacing words with synonyms. The top example is from a sentiment analysis task, i.e. was this review positive or negative? The bottom example is from a textual entailment task, in which the algorithm is asked to decide how the two sentences are logically related. That is, does one imply the other? Does one contradict the other?

Original Text Prediction = **Negative**. (Confidence = 78.0%)
This movie had terrible acting, terrible plot, and terrible choice of actors. (Leslie Nielsen ...come on!!!) the one part I considered slightly funny was the battling FBI/CIA agents, but because the audience was mainly kids they didn't understand that theme.
Adversarial Text Prediction = **Positive**. (Confidence = 59.8%)
This movie had horrific acting, horrific plot, and horrifying choice of actors. (Leslie Nielsen ...come on!!!) the one part I regarded slightly funny was the battling FBI/CIA agents, but because the audience was mainly youngsters they didn't understand that theme.

Table 1: Example of attack results for the sentiment analysis task. Modified words are highlighted in green and red for the original and adversarial texts, respectively.

Original Text Prediction: **Entailment** (Confidence = 86%)
Premise: *A runner wearing purple strives for the finish line.* **Hypothesis:** *A runner wants to head for the finish line.*
Adversarial Text Prediction: **Contradiction** (Confidence = 43%)
Premise: *A runner wearing purple strives for the finish line.* **Hypothesis:** *A racer wants to head for the finish line.*

from [Alzantot et al 2018]

Generative Adversarial Networks

A generative adversarial network (GAN) consists of two neural nets that jointly learn a model of input data. The classifier tries to distinguish real training images from similar fake images. The adversary tries to produce convincing fake images. These networks can produce photorealistic pictures that can be stunningly good (e.g. the dog pictures below) but fail in strange ways (e.g. some of the frogs below).

pictures from New Scientist article on Andrew Brock et al research paper

Good outputs are common. However, large enough collections contain some catastrophically bad outputs, such as the frankencat below right. The neural nets seem to be very good at reproducing the texture and local features (e.g. eyes). But they are missing some type of high-level knowledge that tells people that, for example, dogs have four legs.

from medium.com generated using the tool https://thiscatdoesnotexist.com/

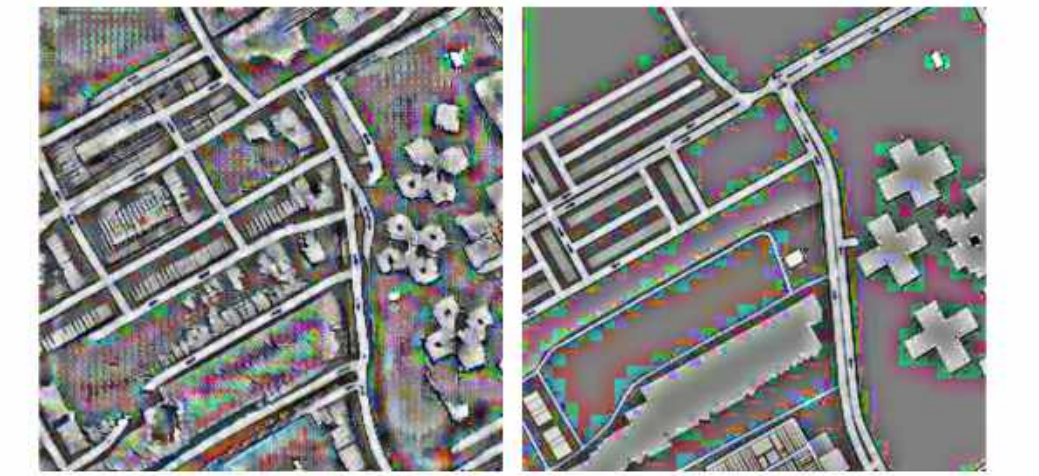
A recent paper exploited this lack of anatomical understanding to detect GAN-generated faces using the fact that the pupils in the eyes had irregular shapes.

GAN cheating

Another fun thing about GANs is that they can learn to hide information in the fine details of images, exploiting the same sensitivity to detail that enables adversarial examples. This GAN was supposedly trained to convert map into arial photographs, by doing a circular task. One half of the GAN translates pictures into arial photographs into maps and the other half translates maps into arial photographs. The output results below are too good to be true:

(a) Aerial photograph: x. (b) Generated map: Fx. (c) Aerial reconstruction: GFx.

The map-producing half of the GAN is hiding information in the fine details of the maps it produces. The other half of the GAN is using this information to populate the arial photograph with details not present in the training version of the map. Effectively, they have set up their own private communication channel invisible to the researchers (until they got suspicious about the quality of the output images.).



(a) Generated map. (b) Training map, for comparison.

More details are in this Techcrunch summary of Chu, Zhmoginov, Sandler, CycleGAN, NIPS 2017.

CS 440/ECE 448
Margaret Fleck

15 Reinforcement Learning

Gridworld pictures are from the U.C. Berkeley CS 188 materials (Pieter Abbeel and Dan Klein) except where noted.

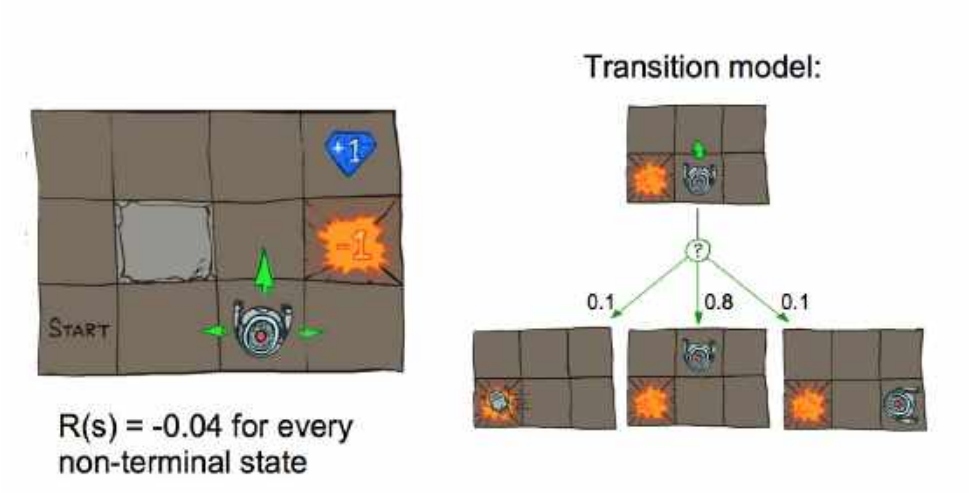
Reinforcement Learning

Reinforcement learning is a technique for deciding which action to take in an environment where your feedback is delayed and your knowledge of the environment is uncertain. Consider, for example, a driving exam. You excute a long series of maneuvers, with a stony-faced examiner sitting next to you. At the end (hopefully) he tells you that you've passed. Reinforcement Learning can be used on a variety of AI tasks, notably board games. But it is frequently associated with learning how to control mechanical systems. It is often one component in an AI system that also uses other techniques that we've seen.

For example, we can use reinforcement learning (plus neural networks) to learn how to park a car. The car starts off behaving randomly, but is rewarded whenever it parks the car, or comes close to doing so. Eventually it learns what's required to get this reward.

Markov Decision Processes

The environment for reinforcement learning is typically modelled as a Markov Decision Process (MDP). The basic set-up is an agent (e.g. imagine a robot) moving around a world like the one shown below. Positions add or deduct points from your score. The Agent's goal is to accumulate the most points.

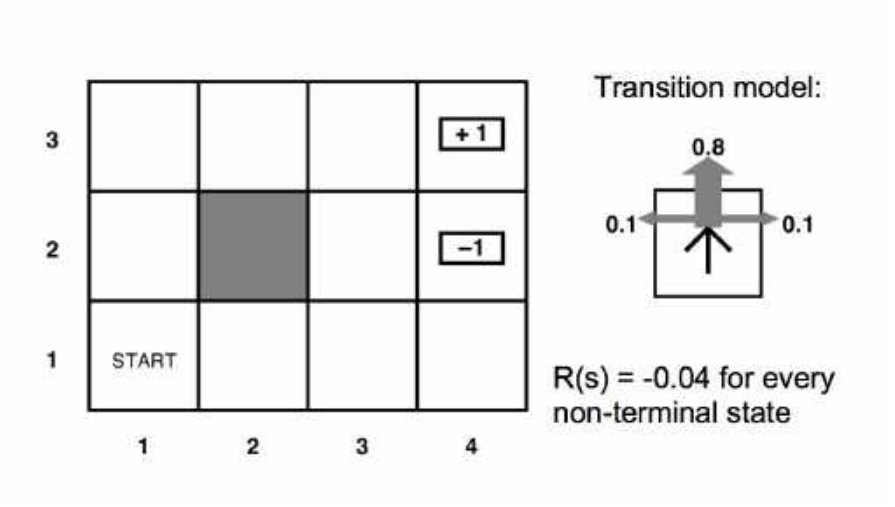


It's best to imagine this as a game that goes on forever. Many real-world games are finite (e.g. a board game) or have actions that stop the game (e.g. robot falls down the stairs). We'll assume that reaching an end state will automatically reset you back to the start of a new game, so the process continues forever.

Actions are executed with errors. So if our agent commands a forward motion, there's some chance that they will instead move sideways. In real life, actions might not happen as commanded due to a combination of errors in the motion control and errors in our geometrical model of the environment.

Mathematical setup

Eliminating the pretty details gives us a mathematician-friendly diagram:

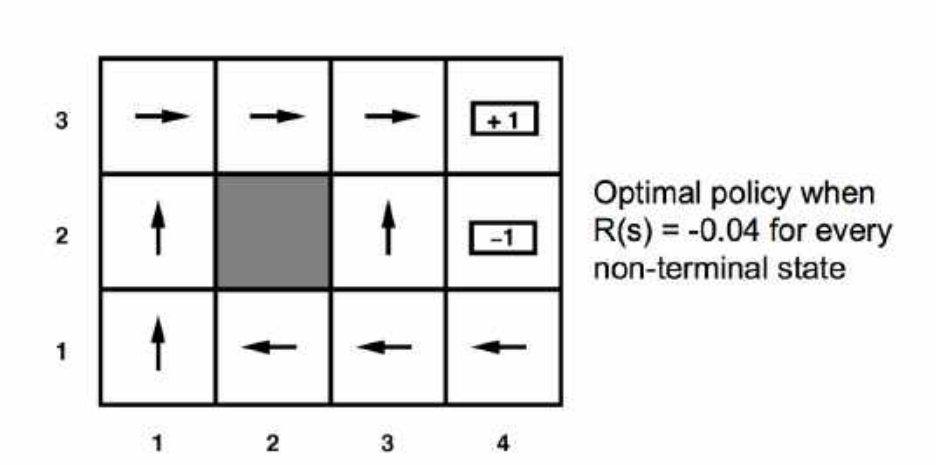


Our MDP's specification contains

- set of states $s \in S$
- set of actions $a \in A$
- reward function R(s)
- transition function P(s' | s,a)

The transition function tells us the probability that a commanded action a in a state s will cause a transition to state s'.

Our solution will be a policy $\pi(s)$ which specifies which action to command when we are in each state. For our small example, the arrows in the diagram below show the optimal policy.



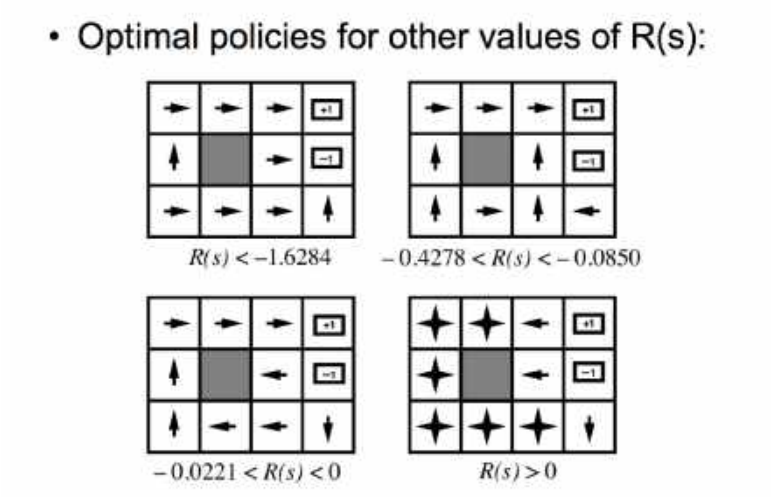
The reward function

The reward function could have any pattern of negative and positive values. However, the intended pattern is

- A few states have big rewards or negative consequences.
- The rest of the states have some small background reward, often constant across all of these background states.

For our example above, the unmarked states have reward -0.04.

The background reward for the unmarked states changes the personality of the MDP. If the background reward is high (lower right below), the agent has no strong incentive to look for the high-reward states. If the background is strongly negative (upper left), the agent will head aggressively for the high-reward states, even at the risk of landing in the -1 location.



What makes a policy good?

We'd like our policy to maximize reward over time. So something like

$$\sum_{\text{sequences of states}} \text{P(sequence)R(sequence)}$$

R(sequence) is the total reward for the sequence of states. P(sequence) is how often this sequence of states happens.

However, sequences of states might be extremely long or (for the mathematicians in the audience) infinitely long. Our agent needs to be able to learn and react in a reasonable amount of time. Worse, infinite sequences of values might not necessarily converge to finite values, even with the sequence probabilities taken into account.

So we make the assumption that rewards are better if they occur sooner. The equations in the next section will define a "utility" for each state that takes this into account. The utility of a state is based on its own reward and, also, on rewards that can be obtained from nearby states. That is, being near a big reward is almost as good as being at the big reward.

So what we're actually trying to maximize is

$$\sum_{\text{sequences of states}} \text{P(sequence)U(sequence)}$$

U(sequence) is the sum of the utilities of the states in the sequence and P(sequence) is how often this sequence of states happens.

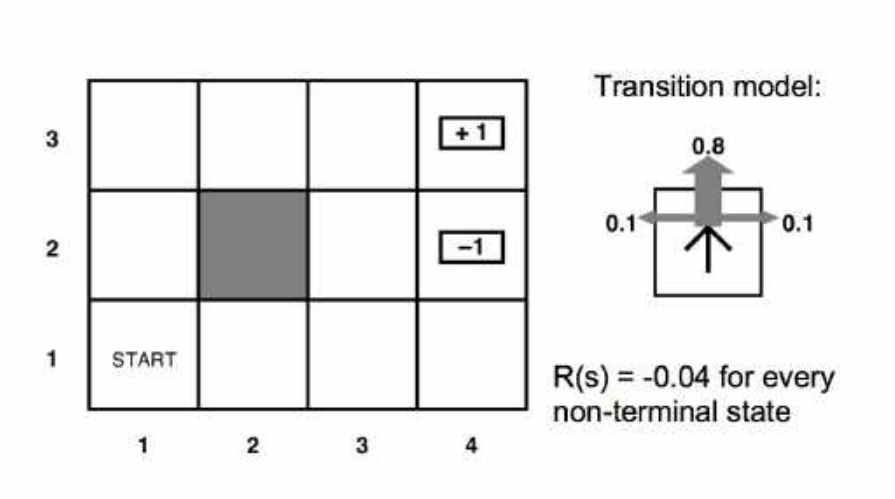
Note

Gridworld pictures are from the U.C. Berkeley CS 188 materials (Pieter Abbeel and Dan Klein) except where noted.

Recap

Recall that we have a Markov Decision Process defined by

- set of states $s \in S$
- set of actions $a \in A$
- reward function R(s)
- transition function P(s' | s,a)



Recall that we are trying to maximize this sum. And we haven't actually defined the utility function U. But it should depends on the reward function R and give less weight to more distant rewards.

$$\sum_{\text{sequences of states}} \text{P(sequence)U(sequence)}$$

The Bellman equation

Suppose our actions were deterministic, then we could express the utility of each state in terms of the utilities of adjacent states like this:

$$U(s) = R(s) + \max_{a \in A} U(move(a, s))$$

where move(a,s) is the state that results if we perform action a in state s. We are assuming that the agent always picks the best move (optimal policy).

Since our action does not, in fact, entirely control the next state s', we need to compute a sum that's weighted by the probability of each next state. The expected utility of the next state would be $\sum_{s' \in S} P(s' | s, a)U(s')$, so this would give us a recursive equation

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

The actual equation should also include a reward delay multiplier γ. Downweighting the contribution of neighboring states by γ causes their rewards to be considered less important than the immediate reward R(s). It also causes the system of equations to converge. So the final "Bellman equation" looks like this:

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

Specifically, this is the Bellman equation for the optimal policy.

Short version of why it converges. Since our MDP is finite, our rewards all have absolute value $\leq$ some bound M. If we repeatedly apply the Bellman equation to write out the utility of an extended sequence of actions, the

kth action in the sequence will have a discounted reward $\leq \gamma^k M$. So the sum is a geometric series and thus converges if $0 \leq \gamma < 1$.

Solving the MDP

There's a variety of ways to solve MDP's and closely-related reinforcement learning problems:

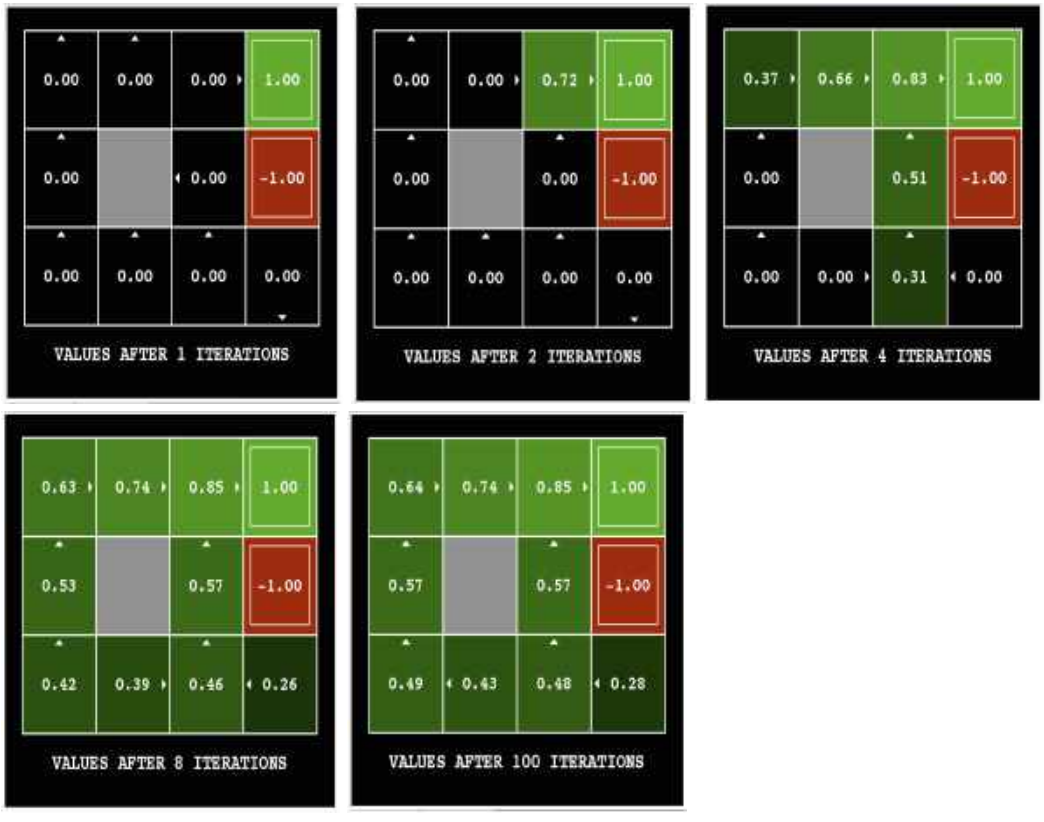
- Value iteration
- Policy iteration
- Dynamic methods that look a lot like perceptron training

Value iteration

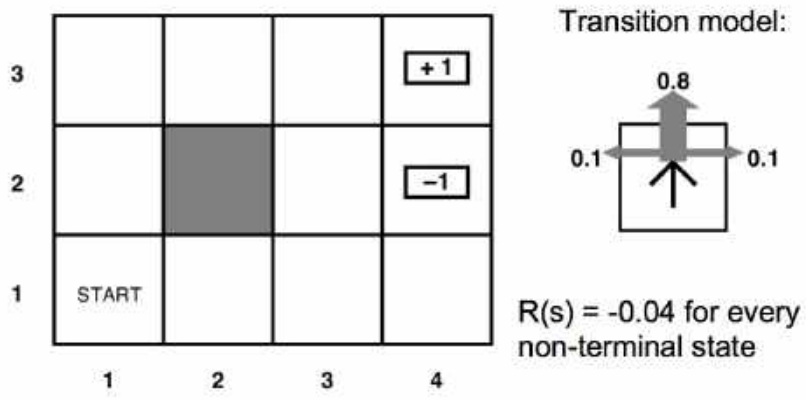
The simplest solution method is value iteration. This method repeatedly applies the Bellman equation to update utility values for each state. Suppose $U_k(s)$ is the utility value for state s in iteration k. Then

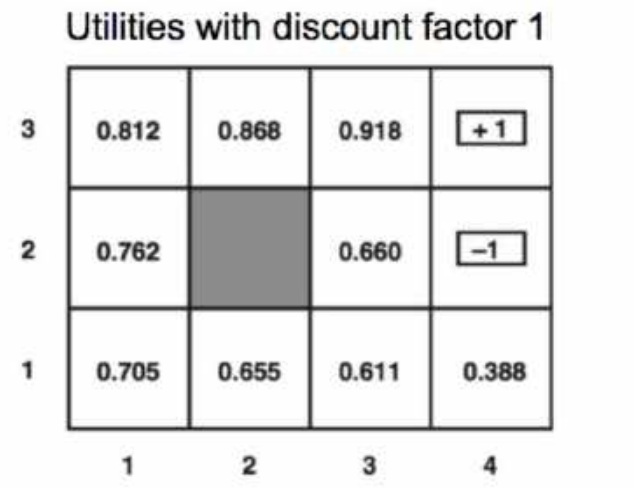
- initialize $U_1(s) = 0$, for all states s
- loop for k = 1 until the values converge
  - $U_{k+1}(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} P(s' \mid s, a) U_k(s')$

The following pictures show how the utility values change as the iteration progresses. The discount factor γ is set to 0.9 and the background reward is set to 0.



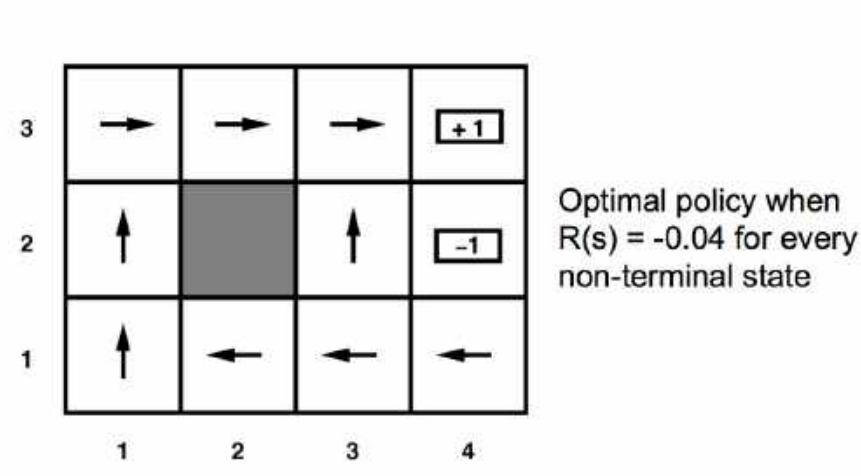
Here's the final output utilities for a variant of the same problem (discount factor $\gamma = 1$ and the background reward is -0.04).





We can read a policy off the final utility values. We can't simply move towards the neighbor with the highest utility, because our agent doesn't always move in the direction we commanded. So the optimal policy requires examining a probabilitic sum:

$$\pi(s) = \text{argmax}_a \sum_{s'} P(s' | s, a) U(s')$$



You can play with a larger example in Andrej Karpathy's gridworld demo.

Warning: Notational variant

Some authors (notably the Berkeley course) use variants of the Bellman equation like this:

$$V(s) = \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V(s')]$$

There is no deep change involved here. This is partly just pushing around the algebra. Also, passing more arguments to the reward function R allows a more complicated theory of rewards, i.e. depending not only on the new state but also the previous state and the action you used. If we use our simpler model of rewards, then this V(s) these can be related to our U(s) as follows

U(s) = R(s) + V(s)

Recap

Pieces of an MDP

- states s in S
- actions a in A
- transition probabilities P(s' | s,a)
- reward function R(s)
- policy $\pi(s)$ returns action

When we're in state s, we command an action $\pi(s)$. However, our buggy controller may put us into a variety of choices for the next state s', with probabilities given by the transition function P.

Bellman equation for optimal policy

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

Recap: Value iteration

Recall how we solve the Bellman equation using value iteration. Let U_t be the utility values at iteration step t.

- Initialize $U_0(s) = 0$, for all states s
- For i=0 until values converge, update U using the equation

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

Then extract the corresponding policy:

$$\pi(s) = \text{argmax}_a \sum_{s'} P(s'|s,a)U(s')$$

Value iteration eventually converges to the solution. Notice that the optimal utility values are uniquely determined, but there may be more than one policy consistent with them.

Policy Iteration

Suppose that we have picked some policy π telling us what move to command in each state. Then the Bellman equation for this fixed policy is simpler because we know exactly what action we'll command:

Bellman equation for a fixed policy: $U(s) = R(s) + \gamma \sum_{s' \in S} P(s'|s,\pi(s))U(s')$

Because the optimal policy is tightly coupled to the correct utility values, we can rephrase our optimization problem as finding the best policy. This is "policy iteration". It produces the same solution as value iteration, but

faster.

Specifically, the policy iteration algorithm looks like this:

- Start with an initial guess for policy π.
- Alternate two steps:
  - Policy evaluation: use policy π to estimate utility values U
  - Policy improvement: use utility values U to calculate a new policy π

Policy iteration makes the emerging policy values explicit, so they can help guide the process of refining the utility values.

The policy improvement step is easy. Just use this equation:

$$\pi(s) = \text{argmax}_a \sum_{s'} P(s' | s, a) U(s')$$

We still need to understand how to do the policy evaluation step.

Policy evaluation

Since we have a draft policy $\pi(s)$ when doing policy evaluation, we have a simplified Bellman equation (below).

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

We have one of these equations for each state s. The equations are still recursive (like the original Bellman equation) but they are now linear. So have two options for adjusting our utility function:

- linear algebra
- a few iterations of value iteration

The value estimation approach is usually faster. We don't need an exact (fully converged) solution, because we'll be repeating this calculation each time we refine our policy π.

Asynchronous dynamic programming

One useful weak to solving Markov Decision Process is "asynchronous dynamic programming." In each iteration, it's not necessary to update all states. We can select only certain states for updating. E.g.

- states frequently seen in some application (e.g. a game)
- states for which the Bellman equation has a large error (i.e. compare values for left and right sides of the equation)

The details can be spelled out in a wide variety of ways.

Intro to reinforcement learning

So far, we've been solving an MDP under the assumption that we started off knowing all details of P (transition probability) and R (reward function). So we were simply finding a closed form for the recursive Bellman equation. Reinforcement learning (RL) involves the same basic model of states and actions, but our learning agent starts off knowing nothing about P and R. It must find out about P and R (as well as U and π) by taking actions and seeing what happens.

Obviously, the reinforcement learner has terrible performance right at the beginning. The goal is to make it eventually improve to a reasonable level of performance. A reinforcement learning agent must start taking actions with incomplete (initially almost no) information, hoping to gradually learn how to perform well. We typically imagine that it does a very long sequence of actions, returning to the same state multiple types. Its performance starts out very bad but gradually improves to a reasonable level.

A reinforcement learner may be

- Online: interacting directly with the world or
- Offline: interacting with a simulation of some sort.

The latter would be safer for situations where real-world hazards could cause real damage to the robot or the environment.

An important hybrid is "experience replay." In this case, we have an online learner. But instead of forgetting its old experiences, it will

- remember old training sequences,
- spend some training time replaying these experiences rather than taking new actions in the world

This is a way to make the most use of training experiences that could be limited by practical considerations, e.g. slow, costly, dangerous.

The reinforcement learning loop

A reinforcement learner repeats the following sequence of steps:

- take an action
- observe the outcome (state and reward)
- update internal representation

There are two basic choices for the "internal representation":

- Model-based: explicitly estimate values for P(s'|s,a) and R(s)
- Model-free: estimate "Q" values, which sidestep the need to estimate P and R

We'll look at model-free learning next lecture.

Model-based RL

A model based learner operates much like a naive Bayes learning algorithm, except that it has to start making decisions as it trains. Initially it would use some default method for picking actions (e.g. random). As it moves around taking actions, it tracks counts for what rewards it got in different states and what state transitions resulted from its commanded actions. Periodically it uses these counts to

- Estimate P(S' | s,a) and R(s), and then
- Use values for P and R to estimate U(s) and $\pi(s)$

Gradually transition to using our learned policy π.

Adding exploration

This obvious implementation of a model-based learner tends to be risk-averse. Once a clear policy starts to emerge, it has a strong incentive to stick to its familiar strategy rather than exploring the rest of its environment. So it could miss very good possibilities (e.g. large rewards) if it didn't happen to see them early.

To improve performance, we modify our method of selecting actions:

- with probability p, pick $\pi(s)$
- with probability 1-p, explore

"Explore" could be implemented in various ways, such as

- make a uniform random choice among the actions
- try actions that we haven't tried "enough" times in the past

The probability of exploring would typically be set high at the start of learning and gradually decreased, allowing the agent to settle into a specific final policy.

The decision about how often to explore must depend on the state s. States that are easy to reach from the starting state(s) end up explored very early, when more distant states may have barely been reached. For each state s, it's important to continue doing significant amounts of exploration until each action has been tried (in s) enough times to have a clear sense of what it does.

Model-free reinforcement learning

We've see model-based reinforcement learning. Let's now look at model-free learning. For reasons that will become obvious, this is also known as Q-learning.

A useful tutorial from Travis DeWolf: Q learning

Recap: Bellman equation

$U(s) = R(s) + \gamma \max_a \sum_{s'} P(s'|s,a)U(s')$

Where

- s is the current state
- a is the commanded action
- s' is new state we actually end up in

Model-free reinforcement learning (aka Q-learning)

For Q learning, we split up and recombine the Bellman equation. First, notice that we can rewrite the equation as follows, because γ is a constant and R(s) doesn't depend on which action we choose in the maximization.

$$U(s) = \max_a [\, R(s) + \gamma \sum_{s'} P(s'|s,a)U(s') \,]$$

Then divide the equation into two parts:

- $Q(s,a) = R(s) + \gamma \sum_{s'} P(s'|s,a)U(s')$
- $U(s) = \max_a Q(s,a)$

Q(s,a) tells us the value of commanding action a when the agent is in state s. Q(s,a) is much like U(s), except that the action is fixed via an input argument. As we'll see below, we can directly estimate Q(s,a) based on

experience.

To get the analog of the Bellman equation for Q values, we recombine the two sections of the Bellman equation in the opposite order. So let's make a version of the second equation referring to the following state s' (since it's true for all states). a' is an action taken starting in this next state s'.

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

Then substitute the second equation into the first

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

This formulation entirely removes the utility function U in favor of the function Q. But we still have an inconvenient dependence on the transition probabilities P.

Removing the transition probabilities

To figure out how to remove the transition probabilities from our update equation, assume we're looking at one fixed choice of state s and commanded action a and consider just the critical inner part of the expression:

$$\sum_{s'} P(s' | s, a) \max_{a'} Q(s', a')$$

We're weighting the value of each possible next state s' by the probability P(s' | s,a) that it will happen when we command action a in state s.

Suppose that we command action a from state s many times. For example, we roam around our environment, occasionally returning to state s. Then, over time, we'll transition into s' a number of times proportional to its transition probability P(s' | s,a). So if we just average

$$\max_{a'} Q(s', a')$$

over an extended sequence of actions, we'll get an estimate of

$$\sum_{s'} P(s' | s, a) \max_{a'} Q(s', a')$$

So we can estimate Q(s,a) by averaging the following quantity over an extended sequence of timesteps t

$$Q(s, a) = R(s) + \gamma \max_{a'} Q(s', a')$$

We need these estimates for all pairs of state and action. In a realistic training sequence, we'll move from state to state, but keep returning periodically to each state. So this averaging process should still work, as long as we do enough exploration.

Q-value update equations

The above assumes implicitly that we have a finite sequence of training data, processed as a batch, so that we can average up groups of values in the obvious way. How do we do this incrementally as the agent moves around?

One way to calculate the average of a series of values $V_1 \ldots V_n$ is to start with an initial estimate of zero and average in each incoming value with a an appropriate discount. That is

- V = 0
- loop for t=1 to n
    - $V = V + V_t/n$

Now suppose that we have an ongoing sequence with no obvious endpoint. Or perhaps the sequence is so long that the world might gradually change, so we'd like to slowly adapt to change. Then we can do a similar calculation called a moving average:

- V = 0
- loop for t=1 to forever
    - $V = (1 - \alpha)V + \alpha V_t$

The influence of each input decays gradually over time. The effective size of the averaging window depends on the constant α.

If we use this to average our values Q(s,a), we get an update method that looks like:

- Q(s,a) = 0 for all s and all a
- for t = 1 to forever
    - $Q(s, a) = (1 - \alpha)Q(s, a) + \alpha[R(s) + \gamma \max_{a'} Q(s', a')]$

Our update equation can also be written as

$$Q(s, a) = Q(s, a) + \alpha[R(s) - Q(s, a) + \gamma \max_{a'} Q(s', a')]$$

Temporal difference (TD) learning

We can now set up the full "temporal difference" algorithm for choosing actions while learning the Q values.

- Q(s,a) = 0 for all s and all a
- for t = 1 to forever
    1. in the current state s, select an action a
    2. observe the reward R(s) and the next state s'
    3. update our Q values $Q(s, a) = Q(s, a) + \alpha[R(s) - Q(s, a) + \gamma \max_{a'} Q(s', a')]$

The obvious choice for an action to command in step 1 is

$$\pi(s) = \text{argmax}_a Q(s, a)$$

But recall from a previous lecture that the agent also needs to explore. So a better method is

- Sometimes choose $\pi(s) = \text{argmax}_a Q(s, a)$
- Sometimes choose a random (or semi-random) action

The best balance between these two possibilities depends on the state and action. We should do more random exploration in states that haven't been sufficiently explored, with a bias towards choosing actions that haven't been sufficiently explored.

What's wrong with TD?

Adding exploration creates an inconsistency in our TD algorithm.

- The maximization $\max_{a'} Q(s', a')$ in step 3 assumes we pick the action that will yield the highest value, but
- Step 1 doesn't always choose the action that currently seems to have the highest value.

So the update in step 3 is based on a different action from the one actually chosen by the agent. In particular, our construction of the update method assumes that we always followed our currently best policy pi, but our action

selection method also includes exploration. So occasionally the agent does random things that don't follow policy.

SARSA

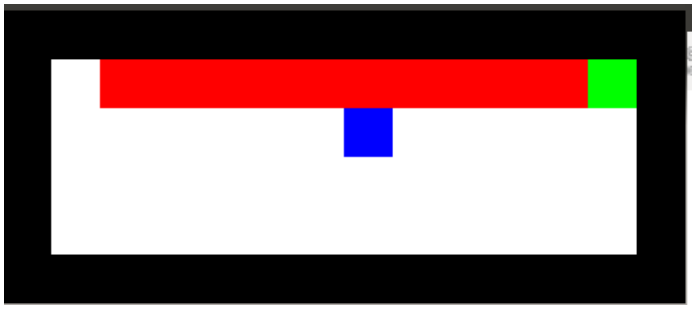
The SARSA ("State-Action-Reward-State-Action") algorithm adjusts the TD algorithm to align the update with the actual choice of action. The algorithm looks like this, where (s,a) is our current state and action and (s',a') is the following state and action.

- Q(s,a) = 0 for all s and all a
- pick an initial state s and initial action a
- for t = 1 to forever
    - observe the reward R(s) and the next state s'
    - from next state s', select next action a'
    - update our Q values using $Q(s,a) = Q(s,a) + \alpha[R(s) - Q(s,a) + \gamma Q(s',a')]$
    - (s,a) = (s',a')

So we're unrolling a bit more of the state-action sequence before doing the update. Our update uses the actions that the agent chose rather than maximizing over all possible actions that the agent might choose.

SARSA vs. TD update

Here's an example from Travis DeWolf that nicely illustrates the difference between SARSA and TD update. The agent is prone to occasional random motions, because it likes to explore rather than just following its optimal policy. The TD algorithm assumes it will do a better job of following policy, so it sends the agent along the edge of the hazard and it regularly falls in. The SARSA agent (correctly) doesn't trust that it will stick to its own policy. So it stays further from the hazard, so that the occasional random motion isn't likely to be damaging.



Mouse (blue) tries to get cheese (green) without falling in the pit of fire (red).

Trained using TD update

Trained using SARSA

Tweaks and extensions

There are many variations on reinforcement learning. Especially interesting ones include

- use features to model states, so we can predict values for unseen states based on familiar states that are similar in some way
- deploy deep learning (e.g. to learn a mapping from features to Q values)
- imitate an expert rather than having numerical rewards.

Reinforcement Learning Demos and Applications

Stanford helicopter

Spider robot from Yamabuchi et al 2013

playing Atari games (V. Mnih et al, Nature, February 2015)

CS 440/ECE 448
Margaret Fleck

14 Vector Semantics


from Wikipedia

Word meanings

What do words mean?
Why do we care?
What are they useful for?

In Russell and Norvig style: what concepts does a rational agent use to do its reasoning?

Seen from a neural net perspective, the key issue is that one-hot representations of words don't scale well. E.g. if we have a 100,000 word vocabulary, we use vectors of length 100,000. We should be able to give each word a distinct representation using a lot fewer parameters.

We can use models of word meaning for a variety of practical tasks.

- model human intelligence
- answering factual questions
- finding word classes (e.g. for parsing)
    - parts of speech (e.g. noun vs. verb)
    - semantic classes: person? surface? soft material we can spread?
- word prediction (e.g. for speech recognition)
- judging fluency of text (e.g. translation, speech recognition)

Word meaning is complex

For each word, we'd like to learn

- information about its basic sense (e.g. "bird", "apple")
- distinctions between similar words (e.g. "apples" vs. "orange")
- grammatical properties (e.g. noun vs. verb, singular vs. plural)
- connotations (e.g. complimentary or pejorative, fancy vs. common word)

Many meanings can be expressed by a word that's more fancy or more plain. E.g. bellicose (fancy) and warlike (plain) mean the same thing. Words can also describe the same property in more or less complimentary terms, e.g. plump (nice) vs. fat (not nice). This leads to jokes about "irregular conjugations" such as

- I'm confident
- You're assertive
- She's aggressive

Logic-style logic-based representations

Here is the definition of "bird" from Oxford Living Dictionaries (Oxford University Press)

> "A warm-blooded egg-laying vertebrate animal distinguished by the possession of feathers, wings, a beak, and typically by being able to fly."

Back in the Day, people tried to turn such definitions into representations with the look and feel of formal logic:

> isa(bird, animal)
> AND has(bird, wings)
> AND flies(bird)
> AND if female(bird), then lays(bird,eggs) ....

The above example is not only incompete, but also has a bug: not all birds fly. It also contains very little information on what birds look like or how they act, so not much help for recognition. Recall from our early classifier lectures that people rely heavily on context to decide what name to give an object.

General problems with this style of meaning representation:

- Very hard to build (mostly by experts)
- Capture the meaning poorly
- Based on elementary predicates (e.g. flies) whose meaning isn't defined.
- Words fit into a larger vocabulary

Contrast

A more subtle problem with these logic-style representations of meaning is that people are are sensitive to the overall structure of the vocabulary. Although some words are seen as synonyms, it's more common that distinct words are seen as having distinct senses, even when the speaker cannot explain what is different about their meaning.

The "Principle of Contrast" states that differences in form imply differences in meaning. (Eve Clark 1987, though idea goes back earlier). For example, kids will say things like "that's not an animal, that's a dog." Adults have a better model that words can refer to more or less general categories of objects, but still make mistakes like "that's not a real number, it's an integer." Apparent synonyms seem to inspire a search for some small difference in meaning. For example, how is "graveyard" different from "cemetery"? Perhaps cemeteries are prettier, or not adjacent to a church, or fancier, or ...

People can also be convinced that two meanings are distinct because an expert says they are, even when they cannot explain how they differ (an observation originally due to Hilary Putnam). For example, pewter (used to make food dishes) and nickel silver (used to make keys for instruments) are similar looking dull silver-colored metals used as replacements for silver. The difference in name tells people that they must be different. But most people would have to trust an expert because they can't tell them apart. Even fewer could tell you that they are alloys with these compositions:

- nickel silver: 60% copper, 20% nickel and 20% zinc
- pewter: 91% tin, 7.5% antimony, and 1.5% copper

Context-based representations

A different approach to representing word meanings, which has recently been working better in practice is to observe how people use each word.

> "You shall know a word by the company it keeps" (J. R. Firth, 1957)

People have a lot of information to work with, e.g. what's going on when the word was said, how other people react to it, even actual explanations from other people. Most computer algorithms only get to see running text. So we approximate this idea by observing which words occur in the same textual context. That is words that occur together often share elements of meaning.

- hour test, syllabus, assignment, lecture
- milk, flour, spoon, oven, cook

We can also figure out an unfamiliar word from examples in context:

> Authentic biltong is flavored with coriander.
> John took biltong on his hike.
> Antelope biltong is better than ostrich biltong .

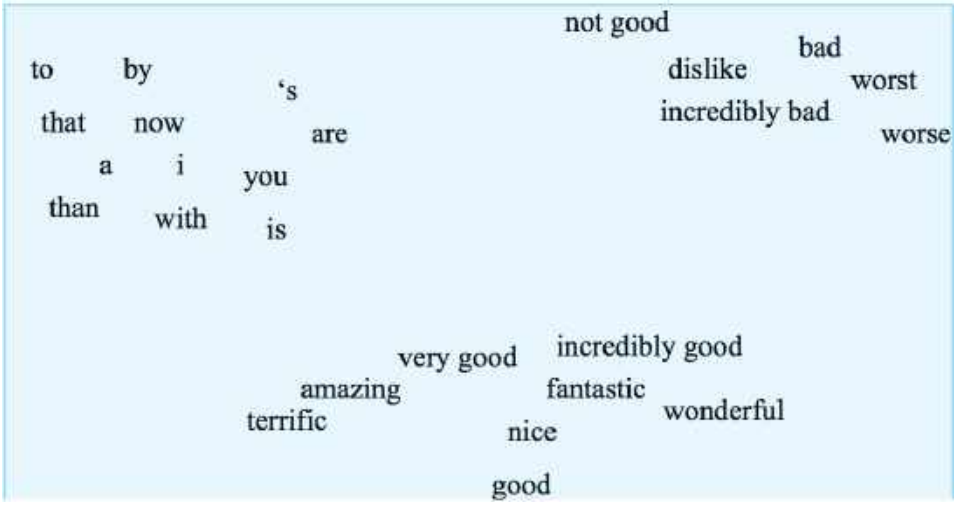
The first context suggests that biltong is food. The second context suggests that it isn't perishable. The third suggests it involves meat.

This would probably be enough information for you to use the word biltong in a sentence, as if you completely understood what it meant. For some tasks, that might be enough. However, you might want to know a bit more detail before deciding whether to eat some. We've all heard stories of travellers who ordered some mystery dish off a foreign language menu and it turned out to be something they hated.

Word Embeddings (vector semantics)

Idea: let's represent each word as a vector of numerical feature values.

These feature vectors are called word embeddings. In an ideal world, the embeddings might look like this embedding of words into 2D, except that the space has a lot more dimensions. (In practice, a lot of massaging would be required to get a picture this clean.) Our measures of similarity will be based on what words occur near one another in a text data corpus, because that's the type of data that's available in quantity.



from Jurafsky and Martin

How to get vectors

Feature vectors are based on some context that the focus word occurs in. There are a continuum of possible contexts, which will give us different information:

- close neighbors (one or two on each side of focus)
- nearby words (e.g. +/- 7 word window)
- whole document

Close neighbors tend to tell you about the word's syntactic properties (e.g. part of speech). Nearby words tend to tell you about its main meaning. Features from the whole document tend to tell you what topic area the word comes from (e.g. math letures vs. Harry Potter).

For example, we might count how often each word occurs in each of a group of documents, e.g. the following table of selected words that occur in various plays by Shakespeare. These counts give each word a vector of numbers, one per document.

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
food	36	58	1	4
wit	20	15	2	3

word counts from Jurafsky and Martin

We can also used these word-document matrices to produce a feature vector for each document, describing its topic. These vectors would typically be very long (e.g. one value for every word that is used in any of the documents). This representation is primarily used in document retrieval, which is historically the application for which many vector methods were developed.

Alternative, we can use nearby words as context. So this example from Jurafsky and Martin shows sets of 7 context words to each side of the focus word.

sugar, a sliced lemon, a tablespoonful of	**apricot**	jam, a pinch each of,
their enjoyment. Cautiously she sampled her first	**pineapple**	and another fruit whose taste she likened
well suited to programming on the digital	**computer**.	In finding the optimal R-stage policy from
for the purpose of gathering data and	**information**	necessary for the study authorized in the

Our 2D data table then relates each focus word (row) to each context word (column). So it might look like this, after processing a number of documents.

	aardvark	computer	data	pinch	result	sugar	...
apricot	0	0	0	1	0	1	
pineapple	0	0	0	1	0	1	
digital	0	2	1	0	1	0	
information	0	1	6	0	4	0	

These raw count vectors will require some massaging to make them into nice feature representations. (We'll see the details later.) And they don't provide a complete model of word meanings. However, as we've seen earlier with neural nets, they provide representations that are good enough to work well in many practical tasks.

Measuring similarity?

Suppose we have a vector of word counts for each document. How do we measure similarity between documents?

Notice that short documents have fewer words than longer ones. So we normally consider only the direction of these feature vectors, not their lengths. So we actually want to compare the directions of these vectors. The smaller the angle between them, the more similar they are.

(2, 1, 0, 4, ...)
(6, 3, 0, 8, ...) same as previous
(2, 3, 1, 2, ...) different

This analysis applies also to feature vectors for words: rare words have fewer observations than common ones. So, again, we'll measure similarity using the angle between feature vectors.

Forgotten linear algebra

The actual angle is hard to work with, e.g. we can't compute it without calling an inverse trig function. It's much easier to approximate the angle with its cosine.

Recall what the cosine function looks like

- 1 at 0 degrees (i.e. the two vectors match)
- 0 at 90 degrees (i.e. the two vectors are perpendicular)

What happens past 90 degrees? Oh, word counts can't be negative! So our vectors all live in the first quadrant, so their angles can't differ by more than 90 degrees.

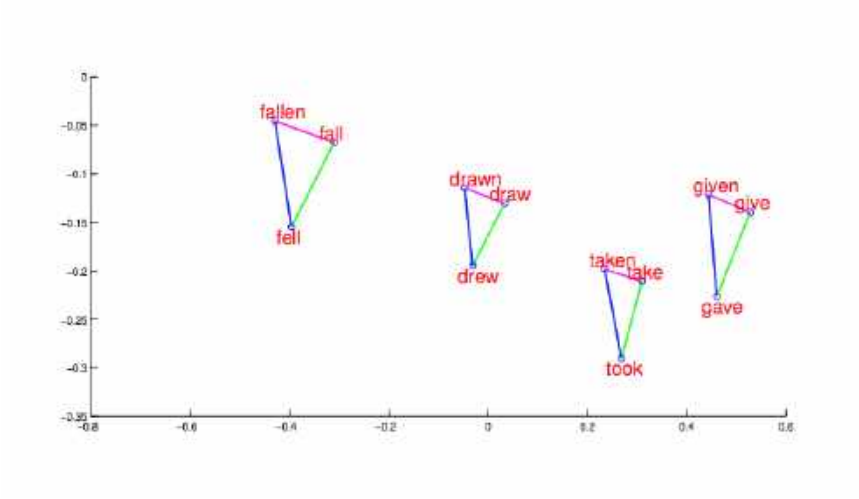
Recall a handy formula for computing the cosine of the angle θ between two vectors:

input vectors $v = (v_1, v_2, \ldots, v_n)$ and $w = (w_1, w_2, \ldots, w_n)$
dot product: $v \cdot w = v_1 w_1 + \ldots + v_n w_n$
$\cos(\theta) = \frac{v \cdot w}{|v||w|}$

So you'll see both "cosine" and "dot product" used to describe this measure of similarity.

Output feature vectors

After significant cleanup (which we'll get to), we can produce embeddings of words in which proximity in embedding space mirrors similarity of meaning. For example, in the picture below (from word2vec), morphologically related words end up clustered. And, more interesting, the shape and position of the clusters are the same for different stems.



from Mikolov, NIPS 2014

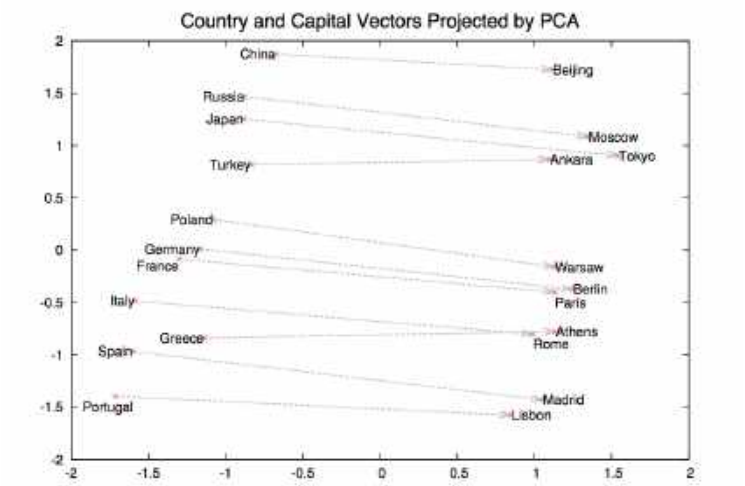
Notice that this picture is a projection of high-dimensional feature vectors onto two dimensions. And they authors are showing us only selected words. It's very difficult to judge the overall quality of the embeddings.

Evaluation

Word embedding models are evaluated in two ways:

- Similarity to human judgement in word analogy tasks.
- Comparing the embeddings to other feature representations, when used as the first stage for solving a larger task (e.g. question answering)

Suppose we find a bunch of pairs of words illustrating a common relation, e.g. countries and their capitals in the figure below. The vectors relating the first word (country) to the second word (capital) all go roughly in the same direction.



from Mikolov, NIPS 2014

We can get the meaning of a short phrase by adding up the vectors for the words it contains. This gives us a simple version of "compositional semantics," i.e. deriving the meaning of a phrase from the meanings of the words in it. We can also do simple analogical reasoning by vector addition and subtraction. So if we add up

```
vector("Paris") - vector("France") + vector("Italy")
```

We should get the answer to the analogy question "Paris is to France as ?? is to Italy". The table below shows examples where this calculation produced an embedding vector close to the right answer:

Expression	*Nearest token*
Paris - France + Italy	Rome
bigger - big + cold	colder
sushi - Japan + Germany	bratwurst
Cu - copper + gold	Au
Windows - Microsoft + Google	Android
Montreal Canadiens - Montreal + Toronto	Toronto Maple Leafs

from Mikolov, NIPS 2014

Some Words in Shakespeare Plays

	As You Like It	**Twelfth Night**	**Julius Caesar**	**Henry V**
battle	1	0	7	13
good	114	80	62	89
food	36	58	1	4
wit	20	15	2	3

word counts from Jurafsky and Martin

Two issues

Raw word count vectors don't work very well. Two basic problems need to be fixed:

- The raw counts require normalization.
- Word-count vectors are too long and sparse.

We'll first see older methods in which these two steps are distinct. We'll then see word2vec, which accomplishes both via one procedure.

Normalizing word counts

Why don't raw word counts work well?

In terms of the item we're trying to classify

- common words have larger feature values than rare words
- long documents have larger feature values than short ones

This problem is handled (as we saw in the last video) by using the direction (not the magnitude) of each count vector. When comparing two vectors, we look at the angle between them (or its cosine or a normalized dot product).

But there's also a related issue, which isn't fixed by comparing angles. Consider the context words, i.e. the words whose counts are entries in our vectors. Then

- Very frequent words (e.g. function words) don't tell you much about the meaning.
- Very rare words provide unreliable context, e.g. an uncommon word like "anteater" might appear in only certain documents about animals, in a random way.
- Local observations (e.g. within a document) are highly correlated, so importance isn't proportional to frequency.

We need to normalize our individual counts to minimize the impact of these effects.

TF-IDF

TF-IDF normalization maps word counts into a better measure of their importance for classification. It is a bit of a hack, but one that has proved very useful in document retrieval. To match the historical development, suppose we're trying to model the topic of a document based on counts of how often each word occurs in it.

Suppose that we are looking at a particular focus word in a particular focus document. The TF-IDF feature is the product of TF (normalized word count) and IDF (inverse document frequency).

> Warning: neither TF nor IDF has a standard definition, and the most common definitions don't match what you might guess from the names. Here is one of many variations on how to define them.

To keep things simple, let's assume that our word occurs at least once. It's pretty much universal that these normalization methods leave 0 counts unchanged.

To compute TF, we transform the raw count c for the word onto a log scale. That is, we replace it by $\log c$. Across many different types of perceptual domains (e.g. word frequency, intensity of sound or light) studies of humans suggest that our perceptions are well modelled by a log scale. Also, the log transformation reduces the impact of correlations (repeated words) within a document.

However, $\log c$ maps 1 to zero and exaggerates the importance of very rare words. So it's more typical to use

$$\text{TF} = 1 + \log_{10}(c)$$

The document frequency (DF) of the word, is df/N, where N is the total number of documents and df is the number of documents that our word appears in. When DF is small, our word provides a lot of information about the topic. When DF is large, our word is used in a lot of different contexts and so provides little information about the topic.

The normalizing factor IDF is also typically put through a log transformation, for the same reason that we did this to TF:

$$\text{IDF} = \log_{10}(N/df)$$

To avoid exaggerating the importance of very small values of N/df, it's typically better to use this:

$$\text{IDF} = \log_{10}(1 + N/df)$$

The final number that goes into our vector representation is TF $\times$ IDF. That is, we multiply the two quantities even though we've already put them both onto a log scale. I did say this was a bit of a hack, didn't I?

Pointwise mutual information (PMI)

Let's switch to our other example of feature vectors: modelling a word's meaning on the basis of words seen near it in running text. We still have the same kinds of normalization/smoothing issues.

sugar, a sliced lemon, a tablespoonful of	**apricot**	jam, a pinch each of,
their enjoyment. Cautiously she sampled her first	**pineapple**	and another fruit whose taste she likened
well suited to programming on the digital	**computer.**	In finding the optimal R-stage policy from
for the purpose of gathering data and	**information**	necessary for the study authorized in the

	aardvark	computer	data	pinch	result	sugar	...
apricot	0	0	0	1	0	1	
pineapple	0	0	0	1	0	1	
digital	0	2	1	0	1	0	
information	0	1	6	0	4	0	

Here's a different (but also long-standing) approach to normalization. We've picked a focus word w and a context word c. We'd like to know how closely our two words are connected. That is, do they occur together more often than one might expect for independent draws?

Suppose that w occurs with probability P(w) and c with probability P(c). If the two were appearing independently in the text, the probability of seeing them together would be P(w)P(c). Our actual observed probability is P(w,c). So we can use the following fraction to gauge how far away they are from independent:

$$\frac{P(w,c)}{P(w)P(c)}$$

Example: consider "of the" and "three-toed sloth." The former occurs a lot more often. However, that's mostly because its two constituent words are both very frequent. The PMI normalizes by the frequencies of the constituent words.

Putting this on a log scale gives us the pointwise mutual information (PMI)

$$I(w,c) = \log_2\left(\frac{P(w,c)}{P(w)P(c)}\right)$$

When one or both words are rare, there is high sampling error in their probabilities. E.g. if we've seen a word only once, we don't know if it occurs once in 10,000 words or once in 1 million words. So negative values of PMI are frequently not reliable. This observation leads some researchers to use the positive PMI (PPMI):

$$\text{PPMI} = \max(0,\text{PMI})$$

Warning: negative PMI values may be statistically significant, and informative in practice, if both words are quite common. For example, "the of" is infrequent because it violates English grammar. There have been some computational linguistics algorithms that exploit these significant zeroes.

Singular value decomposition (SVD)

Both TF-IDF and PMI methods create vectors that contain informative numbers but still very long and sparse. For most applications, we would like to reduce these to short dense vectors that capture the most important

dimensions of variation. A traditional method of doing this uses the singular value decomposition (SVD).

Theorem: any rectangular matrix can be decomposed into three matrices:

- W: maps original coordinates into new ones
- S: diagonal matrix of "singular values"
- C: maps new coordinates back to original ones

$$\underset{|V|\times|V|}{\begin{bmatrix} X \end{bmatrix}} = \underset{|V|\times|V|}{\begin{bmatrix} W \end{bmatrix}} \underset{|V|\times|V|}{\begin{bmatrix} \sigma_1 & 0 & 0 & \dots & 0 \\ 0 & \sigma_2 & 0 & \dots & 0 \\ 0 & 0 & \sigma_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & \sigma_V \end{bmatrix}} \underset{|V|\times|V|}{\begin{bmatrix} C \end{bmatrix}}$$

from Dan Jurafsky at Stanford.

In the new coordinate system (the input/output of the S matrix), the information in our original vectors is represented by a set of orthogonal vectors. The values in the S matrix tell you how important each basis vector is, for representing the data. We can make it so that weights in S are in decreasing order top to bottom.

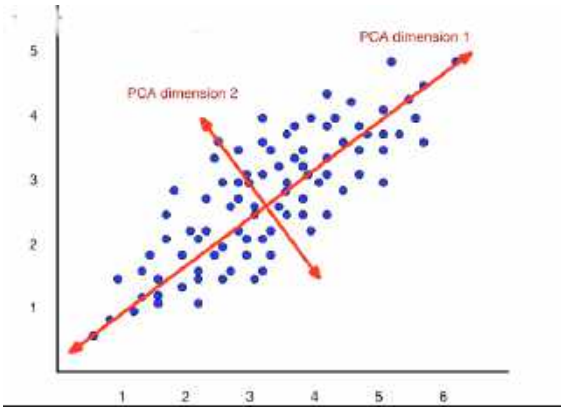
It is well-known how to compute this decomposition. See a scientific computing text for methods to do it accurately and fast. Or, much better, get a standard statistics or numerical analysis package to do the work for you.

Recall that our goal was to create shorter vectors. The most important information lies in the top-left part of the S matrix, where the S values are high. So let's consider only the top k dimensions from our new coordinate system: The matrix W tells us how to map our input sparse feature vectors into these k-dimensional dense feature vectors.

$$\underset{|V|\times|V|}{\begin{bmatrix} X \end{bmatrix}} = \underset{|V|\times k}{\begin{bmatrix} W \end{bmatrix}} \underset{k\times k}{\begin{bmatrix} \sigma_1 & 0 & 0 & \dots & 0 \\ 0 & \sigma_2 & 0 & \dots & 0 \\ 0 & 0 & \sigma_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & \sigma_k \end{bmatrix}} \underset{k\times|V|}{\begin{bmatrix} C \end{bmatrix}}$$

from Dan Jurafsky at Stanford.

The new dimensions are called the "principal components." So this technique is often called a principal component analysis (PCA). When this approach is applied to analysis of document topics, it is often called "latent semantic analysis" (LSA).



from Dan Jurafsky at Stanford.

Bottom line on SVD/PCA

- Been around for ages. Lots of packages to do it for you.
- Helps human analysts figure out the important dimensions of variation.
- Works pretty well. Only the best newer methods do better.

Word2vec

The word2vec (aka skip-gram) algorithm is a newer algorithm that does normalization and dimensionality reduction in one step. That is, it learns how to embed words into an n-dimensional feature space. The algorithm was introduced in a couple papers by Mikolov et al in 2013. However, it's pretty much impossible to understand the algorithm from those papers, so the following derivation is from later papers by Yoav Goldberg and Omer Levy. Two similar embedding algorithms (which we won't cover) are the CBOW algorithm from word2vec and the GloVe algorithm from Stanford.

Overview of word2vec

In broad outline word2vec has three steps

- Gather pairs (w,c) where w is our focus word and c is a nearby context word,
- Randomly embed all words into $\mathbb{R}^k$
- Iteratively adjust the embedding so that pairs (w,c) end up with similar coordinates.

A pair (w,c) is considered "similar" if the dot product $w \cdot c$ is large.

For example, our initial random embedding might put "elephant" near "matchbox" and far from "rhino." Our iterative adjustment would gradually move the embedding for "rhino" nearer to the embedding for "elephant."

Two obvious parameters to select. The dimension of the embedding space would depend on how detailed a representation the end user wants. The size of the context window would be tuned for good performance.

Negative sampling

Problem: a great solution for the above optimization problem is to map all words to the same embedding vector. That defeats the purpose of having word embeddings.

Suppose we had negative examples (w,c'), in which c' is not a likely context word for w. Then we could make the optimization move w and c' away from each other. Revised problem: our data doesn't provide negative examples.

Solution: Train against random noise, i.e. randomly generate "bad" context words for each focus word.

- Called "negative sampling" (not a very evocative term)
- For a fixed focus word w, negative context words are picked with a probability based on how often words occur in the training data.
- Depends on good example pairs being relatively sparse in the space of all word pairs (probably correct)

The negative training data will be corrupted by containing some good examples, but this corruption should be a small percentage of the negative training data.

Two embeddings

Another mathematical problem happens when we consider a word w with itself as context. The dot product $w \cdot w$ ls large, by definition. But, for most words, w is not a common context for itself, e.g. "dog dog" is not very common compared to "dog." So setting up the optimization process in the obvious way causes the algorithm to want to move w away from itself, which it obviously cannot do.

To avoid this degeneracy, word2vec builds two embeddings of each word w, one for w seen as a focus word and one for w used as a context word. The two embeddings are closely connected, but not identical. The final representation for w will be a concatenation of the two embedding vectors.

Details of algorithm

Our classifier tries to predict yes/no from pairs (w,c), where "yes" means c is a good context word for w.

We'd like to make this into a probabilistic model. So we run dot product through a sigmoid to produce a "probability" that (w,c) is a good word-context pair. These numbers probably aren't the actual probabilities, but we're about to treat them as if they are. That is, we approximate

$$P(\text{good}|w, c) \approx \sigma(w \cdot c).$$

By the magic of exponentials (see below), this means that the probability that (w,c) is not a good word-context pair is

$$P(\text{bad}|w, c) \approx \sigma(-w \cdot c).$$

Now we switch to log probabilities, so we can use addition rather than multiplication. So we'll be looking at e.g.

$$\log(P(\text{good}|w, c)) \approx \log(\sigma(w \cdot c))$$

Suppose that D is the positive training pairs and D' is the set of negative training pairs. So our iterative refinement algorithm adjusts the embeddings (both context and word) so as to maximize

$$\sum_{(w,c)\in D} \log(\sigma(w \cdot c)) + \sum_{(w,c)\in D'} \log(\sigma(-w \cdot c))$$

That is, each time we read a new focus word w from the training data, we

- find its context words,
- generate some negative context words, and
- adjust the embeddings of w and the context words in the direction that increase the above sum.

Unexplained exponentials

Ok, so how did we get from $P(\text{good}|w, c) = \sigma(w \cdot c)$ to $P(\text{bad}|w, c) = \sigma(-w \cdot c)$?

"Good" and "bad" are supposed to be opposites. So $P(\text{bad}|w,c) = \sigma(-w \cdot c)$ should be equal to $1 - P(\text{good}|w,c) = 1 - \sigma(w \cdot c)$. I claim that $1 - \sigma(w \cdot c) = \sigma(-w \cdot c)$.

This claim actually has nothing to do with the dot products. As a general thing $1 - \sigma(x) = \sigma(-x)$. Here's the math.

Recall the definition of the sigmoid function: $\sigma(x) = \frac{1}{1+e^{-x}}$.

$$
\begin{aligned}
1 - \sigma(x) &= 1 - \frac{1}{1+e^{-x}} = \frac{e^{-x}}{1+e^{-x}} \quad \text{(add the fractions)} \\
&= \frac{1}{1/e^{-x}+1} \quad \text{(divide top and bottom by } e^{-x}\text{)} \\
&= \frac{1}{e^{x}+1} \quad \text{(what does a negative exponent mean?)} \\
&= \frac{1}{1+e^{x}} = \sigma(-x)
\end{aligned}
$$

Tweaking word2vec

In order to get nice results from word2vec, the basic algorithm needs to be tweaked a bit to produce this good performance. Similar tweaks are frequently required by other similar algorithms.

Building the set of training examples

In the basic algorithm, we consider the input (focus) words one by one. For each focus word, we extract all words within +/- k positions as positive context words. We also randomly generate a set of negative context words. This produces a set of positive pairs (w,c) and a set of negative pairs (w,c') that are used to update the embeddings of w, c, and c'.

> Tweak 1: Word2vec uses more negative training pairs than positive pairs, by a factor of 2 up to 20 (depending on the amount of training data available).

You might think that the positive and negative pairs should be roughly balanced. However, that apparently doesn't work. One reason may be that the positive context words are definite indications of similarity, whereas the negative words are random choices that may be more neutral than actively negative.

> Tweak 2: Positive training examples are weighted by 1/m, where m is the distance between the focus and context word. I.e. so adjacent context words are more important than words with a bit of separation.

The closer two words are, the more likely their relationship is strong. This is a common heuristic in similar algorithms.

Smoothing negative context counts

For a fixed focus word w, negative context words are picked with a probability based on how often words occur in the training data. However, if we compute P(c) = count(c)/N (N is total words in data), rare words aren't picked often enough as context words. So instead we replace each raw count count(c) with $(\text{count}(c))^{\alpha}$. The probabilities used for selecting negative training examples are computed from these smoothed counts.

α is usually set to 0.75. But to see how this brings up the probabilities of rare words compared to the common ones, it's a bit easier if you look at $\alpha = 0.5$, i.e. we're computing the square root of the input. In the table below, you can see that large probabilities stay large, but very small ones are increased by quite a lot. After this transformation, you need to normalize the numbers so that the probabilities add up to one again.

x	$x^{0.75}$	$\sqrt{x}$
.99	.992	.995
.9	.924	.949
.1	.178	.316
.01	.032	.1
.0001	.001	.01

This trick can also be used on PMI values (e.g. if using the methods from the previous lecture).

Deletion, subsampling

Ah, but apparently they are still unhappy with the treatment of very common and very rare words. So, when we first read the input training data, word2vec modifies it as follows:

- very rare words are deleted from the text, and
- very common words are deleted with a probability that increases with how frequent they are.

This improves the balance between rare and common words. Also, deleting a word brings the other words closer together, which improves the effectiveness of our context windows.

Evaluation

The 2014 version of word2vec uses use 1 billion words to train embeddings for basic task.

For the word analogy tasks, they used an embedding with 1000 dimensions and about 33 billion words of training data. Performance on word analogies is about 66%.

By coomparison: children hear about 2-10 million words per year. Assuming the high end of that range of estimates, they've heard about 170 million words by the time they take the SAT. So the algorithm is performing well, but still seems to be underperforming given the amount of data it's consuming.

A more recent embedding method, BERT large, is trained using a 24-layer network with 340M parameters. This has somewhat improved performance but apparently can't be reproduced on a standard GPU. Again, a direction for future research is to figure out why ok performance seems to require so much training data and compute power.

Some follow-on papers

Original Mikolov et all papers:

- Efficient Estimation of Word Representations in Vector Space
- Distributed Representations of Words and Phrases and their Compositionality

Goldberg and Levy papers (easier and more explicit)

- word2vec Explained
- Dependency-Based Word Embeddings
- Neural Word Embedding as Implicit Matrix Factorization
- Improving Distributional Similarity with Lessons learned from Word Embeddings

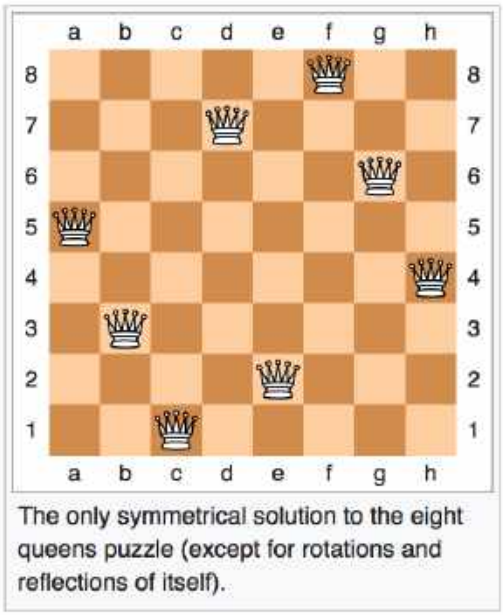
CS 440/ECE 448
Margaret Fleck

16 Constraint Satisfaction Problems

The last few topics in this course cover specialized types of search. These were once major components of AI systems. They are are still worth knowing about because they still appear in the higher-level (more symbolic) parts of AI systems. We'll start by looking at constraint satisfaction problems.

The n-queens problem

Remember that a queen in chess threatens another piece if there is a direct path between the two either horizontally, or vertically, or diagonally. In the **n queens problem,** we have to place n queens on a n by n board so that no queen threatens another. Here is a solution for n=8:



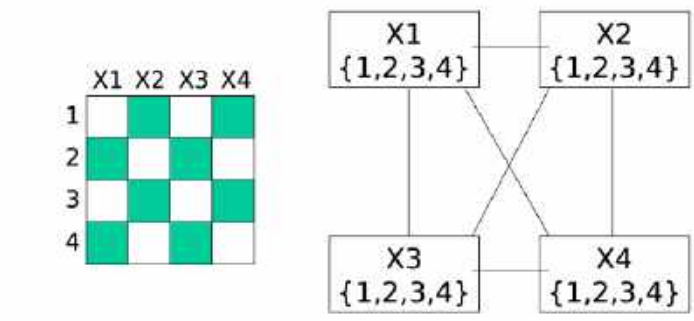
The only symmetrical solution to the eight queens puzzle (except for rotations and reflections of itself).

From Wikipedia

Since each column must contain exactly one queen, we can model this as follows

- One variable per column.
- Possible values: the n vertical positions within the column.
- Constraints: can't have two in the same row, can't have two in the same diagonal.

We can visualize this as a graph in which each variable is a node and edges link variables involved in binary constraints. So X1 in the following diagram is a variable representing a column. Its possible values {1,2,3,4} are the row to place the queen in.



from Bonnie Dorr, U. Maryland (via Mitch Marcus at U. Penn).

What is a CSP problem?

A constraint-satisfaction problem (CSP) consists of

- Variables
- Set of allowed values (for each variable)
- Constraints

In many examples, all variables have the same set of allowed values. But there are problems where different variables may have different possible values.

Need to find a complete assignment (one value for each variable) that satisfies all the constraints

In basic search problems, we are explicitly given a goal (or perhaps a small set of goals) and need to find a path to it. In a constraint-satisfaction problem, we're given only a description of what constraints a goal state must satisfy. Our problem is to find a goal state that satisfies these constraints. We do end up finding a path to this goal, but the path is only temporary scaffolding.

Map Coloring

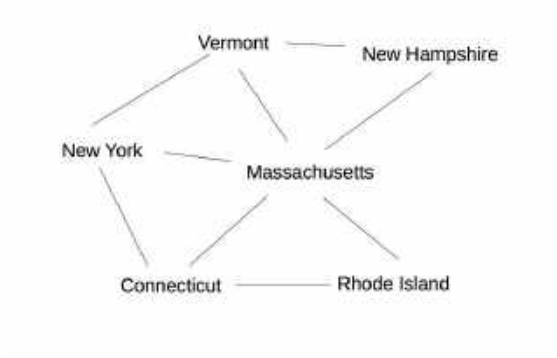
The map coloring problem requires that we color regions of the plan such that neighboring regions never have the same color. (Touching only at a corner doesn't count as neighboring.) The map below shows an example.

From Wikipedia

Viewed as a CSP problem

- Variables: states/countries
- Values: {pink, orange, green, yellow}
- Constraints: extended boundary --> different color

Pictured as a constraint graph, the states/countries are nodes, and edges connect adjacent states. A small example is shown below. Theoretical work on this topic is found mostly under the keyword "graph coloring." Applications of graph coloring include register allocation, final exam scheduling.



The "4 color Theorem" states that any planar map can be colored with only four colors. It was proved at U. Illinois in 1976 by Kenneth Appel and Wolfgang Haken. It required a computer program to check almost 2000 special cases. This computer-assisted proof was controversial at the time, but became accepted after the computerized search had been replicated a couple times.

Notice that graph coloring is NP complete. We don't know for sure if NP problems require polynomial or exponential time, but we suspect they require exponential time. However, many practical applications can exploit domain-specific heuristics (e.g. linear scan for register allocation) or loopholes (e.g. ok to have small conflicts in final exams) to produce fast approximate algorithms.

Cryptarithmetic

Cryptarithmetic is another classic CSP problem. Here's an example:

```
    T W O
  + T W O
---------
= F O U R
```

We can make this into a CSP model as follows:

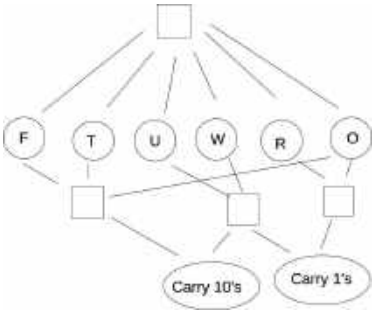
- Variables: the letters
- Possible values: {0,1,..,9}
- Constraints: all letters have different values, addition works as intended, leading digits aren't zero.

"Addition works as intended" can be made concrete with these equations, where C_1 and C_10 are the carry values.

```
O + O = R + 10 * C_1
W + W + C_1 = U + 10*C_10
T + T + C_10 = O + 10*F
```

"All different" and the equational constraints involve multiple variables. Usually best to keep them in that form, rather than converting to binary. We can visualize this using a slightly different type of graph with some extra nodes:

- nodes for the basic variables
- nodes for auxiliary variables (carry values)
- square boxes for constraints, attached to variables they involve



Sudoku

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

from Wikipedia

Sudoku can be modelled as a CSP problem,

- Variables: cells x_ij
- Values: {1,2,...,9}
- Constraints: no duplicates in row, no duplicates in column, no duplicates in block

Scheduling problems

We can also use this framework to model scheduling problems, e.g. planning timing and sequence of operations in factory assembly.

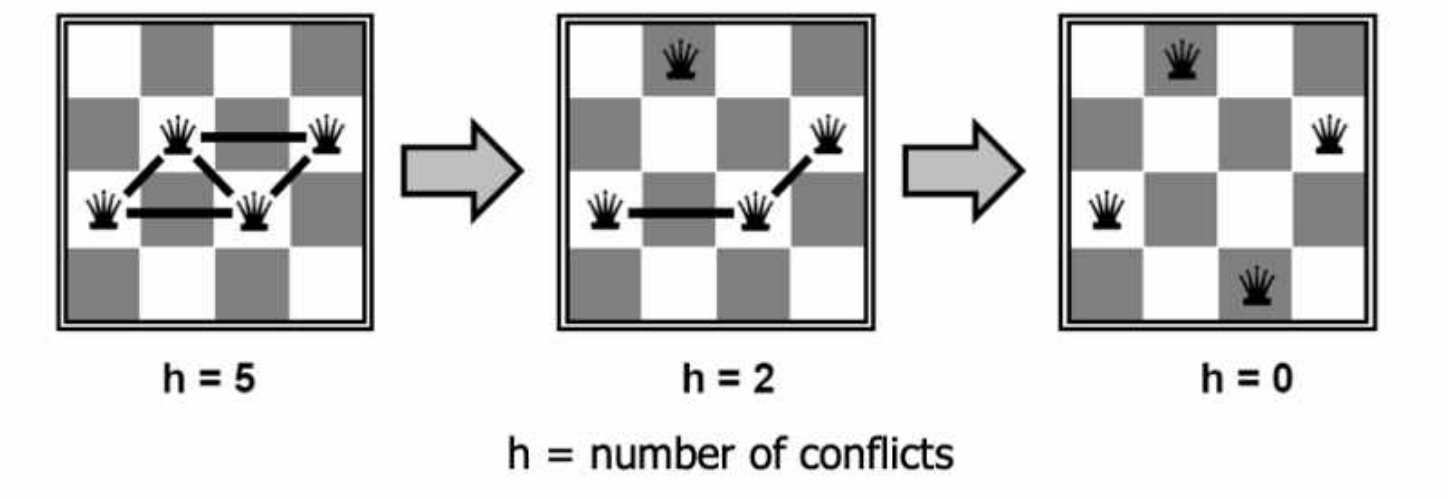
- Variables: tasks (with how long they take)
- Values: start times
- Constraints: certain tasks must/must not run at the same time, certain tasks must precede/follow other tasks.

Approaches to CSP solving

There are two basic approaches to solving constraint satisfaction problems: hill climbing and backtracking search. CSP problems may include multi-variate constraints. E.g. sudoku requires that there be no duplicate values within each row/column/block. For this class, we'll stick to constraints involving two variables.

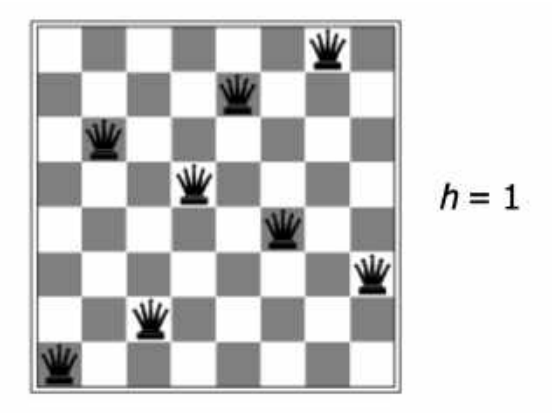
Hill climbing

In the hill-climbing approach, we randomly pick a full set of variable assignments, then try to tweak it into a legal solution. At each step, our draft solution will have some constraints that are still violated. We change the value of one variable to try to reduce the number of conflicts as much as possible. (If there are several options, pick one randomly.) In the following example, this strategy works well:



from Lana Lazebnik, Fall 2017

Hill climbing can be very effective on certain problems, and people frequently use it to solve complicated constraint satisfaction problems, e.g. scheduling classes or final exams. However, this method can get stuck in local optima. For example, the following configuration for 8 Queens has a single constraint violation, but no adjustment of a single piece will reduce the number of violations to zero.



from Lana Lazebnik, Fall 2017

To avoid being stuck in local optima, hill-climbing algorithms often add some randomization:

- Randomly pick a new starting configuration when apparently stuck.
- Add a random component to the movement algorithm, so it sometimes moves to a higher-cost configuration ("simulated annealing").

Backtracking search

The other basic approach to a constraint satisfaction problem is to choose variable values one at a time, undoing assignments ("backtracking") when we discover that our current set of assignments can't work. Specifically, we set up search as follows:

- start: no variables have a value
- each step: assign a value to one variable
- end: all variables have a value

CSP search has some special properties which will affect our choice of search method. If our problem has n variables, then all solutions are n steps away. So DFS is a good choice for our search algorithm.

- We don't have to worry about finding a shortest path. (They are all length n.)
- It's not possible to loop. Each step adds a variable value and search cuts off at depth n.

We can use a very stripped down DFS implementation, without explicit loop detection. This can be made to use only a very small amount of memory. In this type of AI context, DFS is often called "backtracking search" because it spends much of its time getting blocked and retreating back up the search tree to try other options.

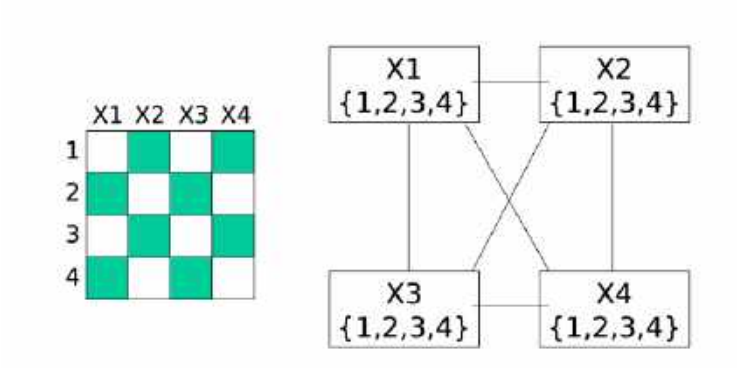
Forward checking

A variety of heuristics can be used to speed up backtracking search. Let's start by looking at when to detect that our path is failing and we need to backtrack.

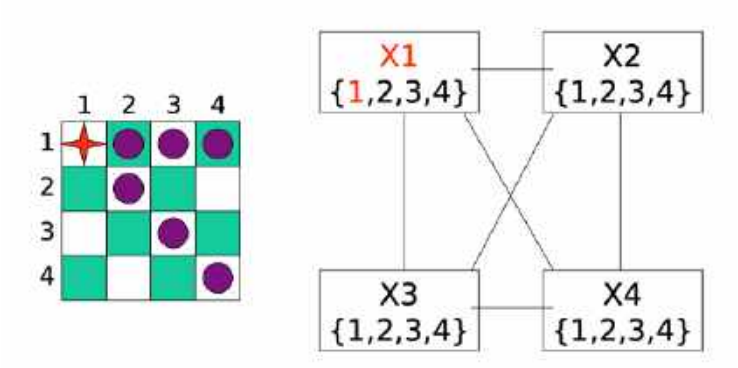
Stupid method
: Always go down n levels to create a complete solution. Then check if it meets the constraints. If not, retreat back up the DFS search tree.

Smart method (forward checking)
: During search, each variable x keeps a list D(x) containing its possible values. At each search step, remove values from these lists if they violate constraints, given the values we've already assigned to other variables. Back up if any variable has no possible values left.

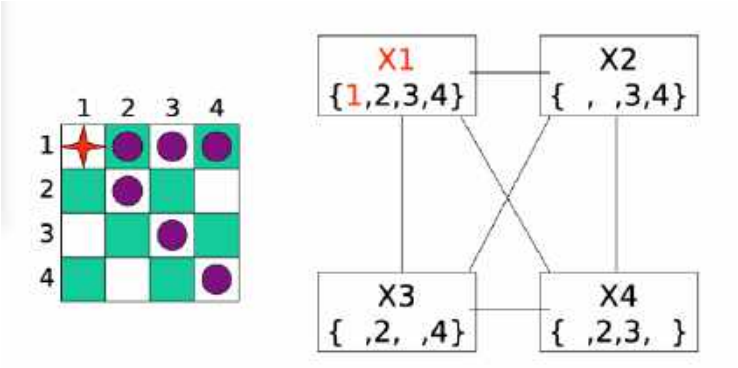
Here is a little demo of forward checking, from Bonnie Dorr, U. Maryland, via Mitch Marcus at U. Penn. We start with each variable x having a full list of possible values D(x) and no assignments.



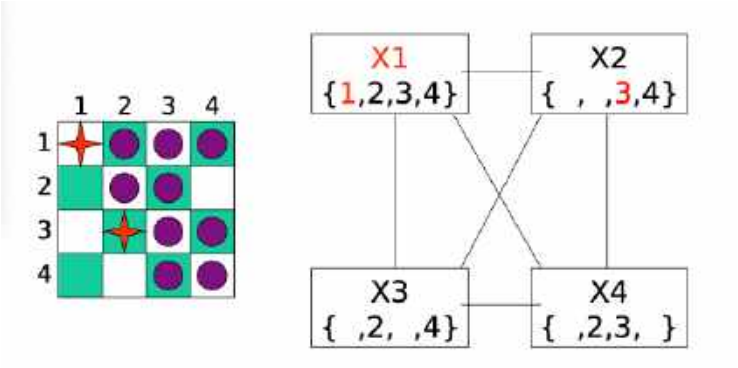
First, we assign a value to the first variable. The purple cells show which other assignments are now problematic.



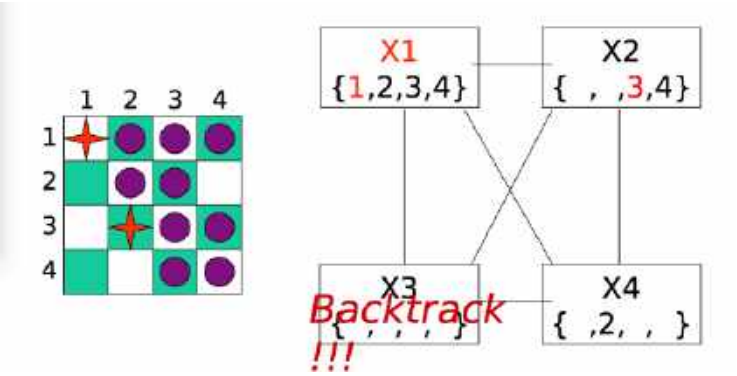
Forward checking now removes conflicting values from each list D(x).



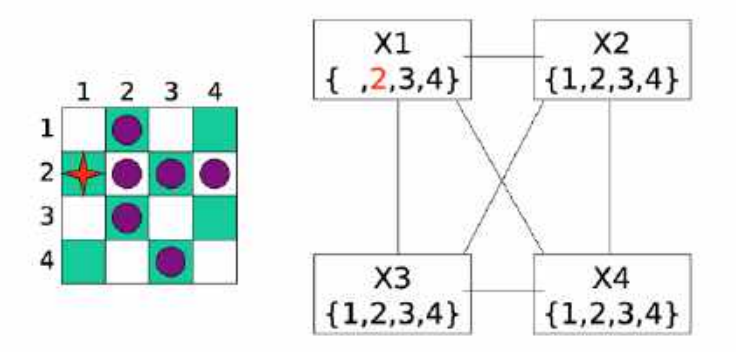
We now pick a value for our second variable.



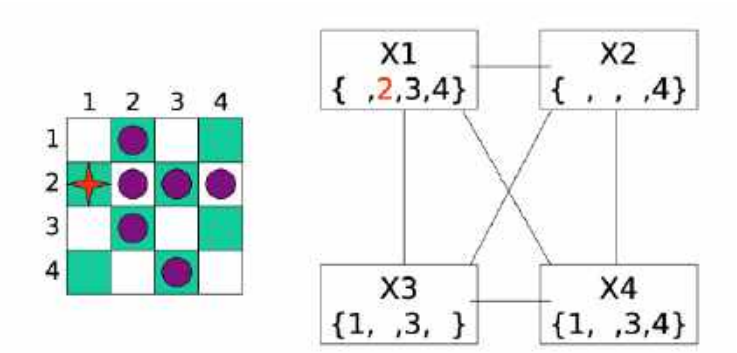
However, when we do forward checking, we discover that variable X3 has no possible values left. So we have to backtrack and undo the most recent assignment (to X2).



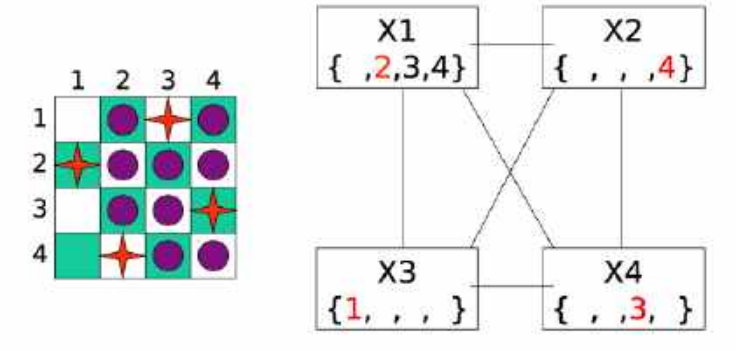
After some more assignments and backtracking, we end up giving up on X1=1 and moving on to exploring X1=2.



Forward checking reduces the sets of possible values to this:



After a few more steps of assignments and forward checking, we reach a full solution to the problem:

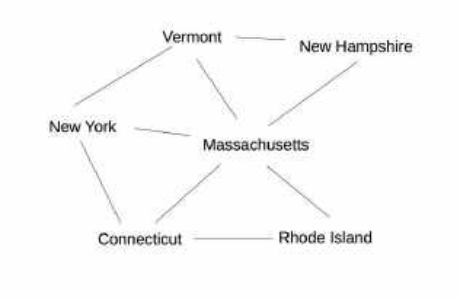


Heuristics for variable assignments

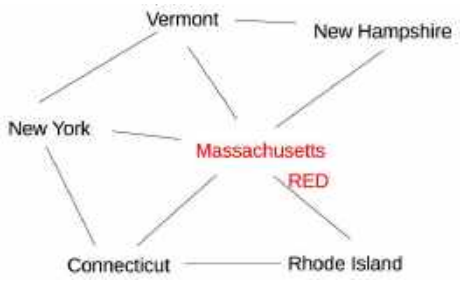
In the previous example, we tried variables in numerical order, i.e. how they came to us in the problem statement. When solving more complex problems, it can be important to use a smarter selection method. For example:

a. Choose the variable with fewest remaining values.
b. If there are ties, choose the variable that constrains the most other variables (i.e. node with highest degree in the constraint graph).

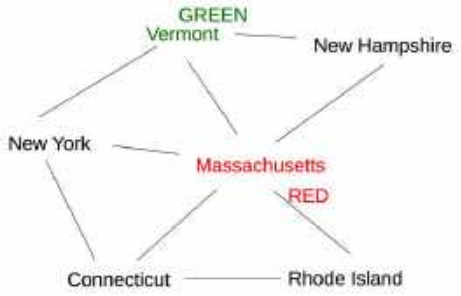
Suppose we are coloring the following graph with 3 colors (red, green, blue).



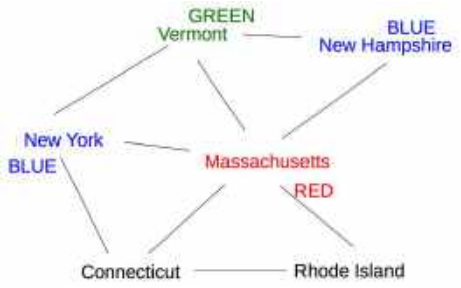
All states have three possible color values, so we use heuristic (b) and pick a color for Massachusetts:



The five remaining states are still tied on (a), so we use heuristic (b) to pick one of the states with three neighbors (Vermont):



Now two states (New York and New Hampshire) have only one possible color value (blue), so we color one of them next:

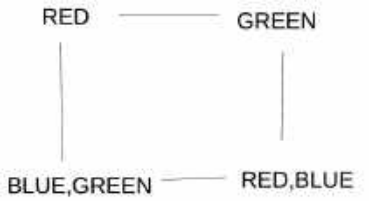


Choosing a value

Once we've chosen a variable, we can be smart about the order in which we explore its values. A good heuristic is:

- Start with the value that leaves the most options open for other variables.

For example, suppose we're trying to pick a value for the lower left node in the graph below. We should try Green first, because it leaves two color options open for the lower right node.



If this preferred choice doesn't work out, then we'll back up and try the other choice (Blue).

Symmetries

Constraint solving code should be designed to exploit any internal symmetries in the problem. For example, the map coloring problem doesn't care which color is which. So it doesn't matter which value we pick for the first variable (sometimes also some later choices). Similarly, there aren't really four distinct choices for the first column of n-queens, because reflecting the board vertically (or horizontally) doesn't change the essential form of the solution.

Recap: CSP algorithm

For each variable X, maintain set D(X) containing all possible values for X.

Run DFS. At each search step:

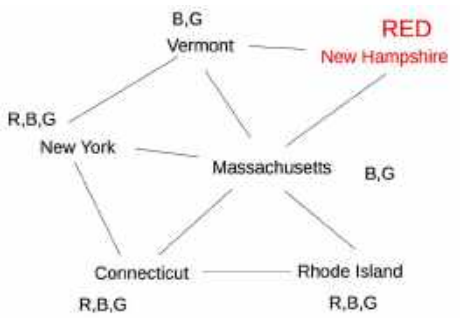
1. Pick a value for one variable. (Back up if no choices are available.)
2. Forward checking: prune all the sets D(X) to remove violations of constraints

And we saw some heuristics for choosing variables and values in step 1.

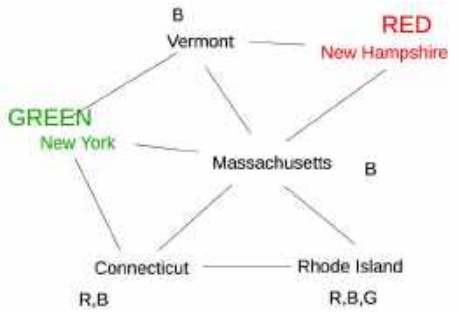
Constraint propagation

When we assign a value to variable X, forward checking only checks variables that share a constraint with X, i.e. are adjacent in the constraint graph. This is helpful, but we can do more to exploit the constraints. Constraint propagation works its way outwards from X's neighbors to their neighbors, continuing until it runs out of nodes that need to be updated.

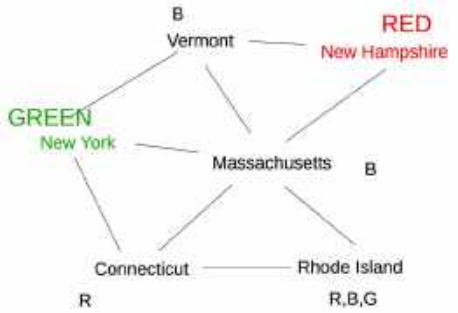
Suppose we are coloring the same state graph with {Red,Green,Blue} and we decide to first color New Hampshire red:



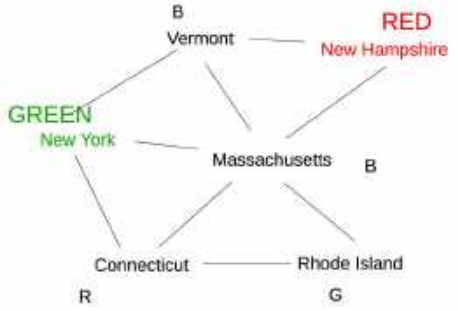
Forward checking then removes Red from Vermont and Massachusetts. Suppose we now color New York Green. Forward checking removes Green from its immediate neighbors, giving us this:



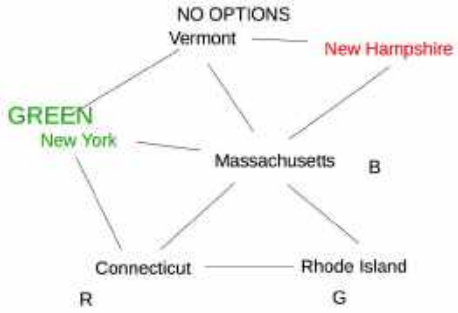
At this point, forward checking stops. Constraint propagation continues. Since Massachusetts now has fewer possible values, its neighbor Connecticut also needs to be updated.



Rhode Island is adjacent to Massachusetts and Connecticut, so its options need to be updated:



Finally, Vermont is a neighbor of Massachusetts, so update its options.

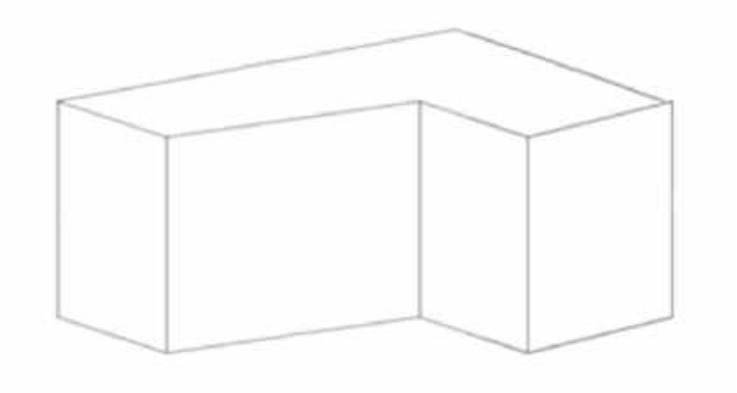


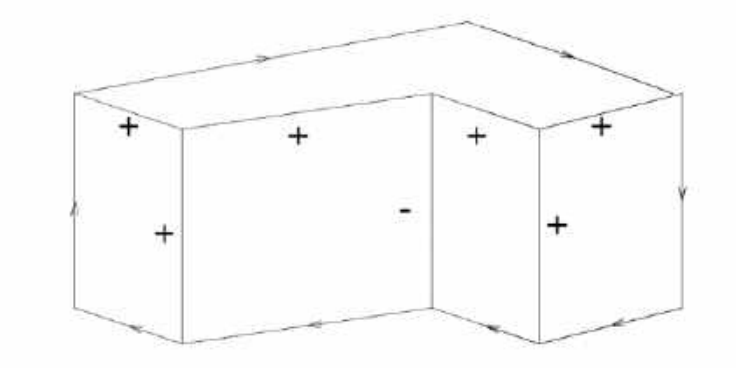
Since this leaves Vermont with no possible values, we backtrack to our last decision point (coloring New York Green).

The exact sequence of updates depends on the order in which we happen to have stored our nodes. However, constraint propagation typically makes major reductions in the number of options to be explored, and allows us to figure out early that we need to backtrack.

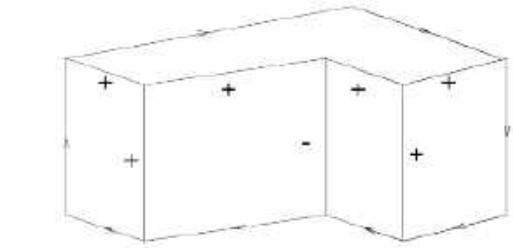
Waltz labelling

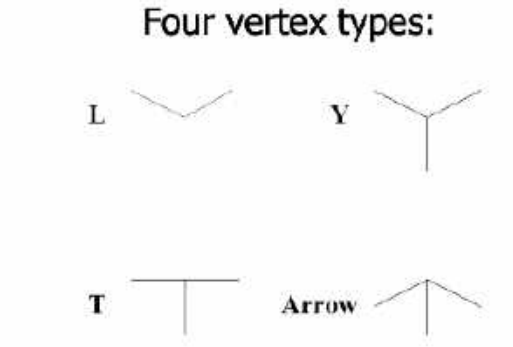
Constraint propagation was developed by David Waltz in the early 1970's for the Shakey project (e.g. see David Waltz's 1972 thesis). The original task was to take a wireframe image of blocks and decorate each edge with a labels indicating its type. Object boundaries are indicated by arrows, which need to point clockwise around the bounary. Plus and minus signs mark internal edges that are convex and concave.

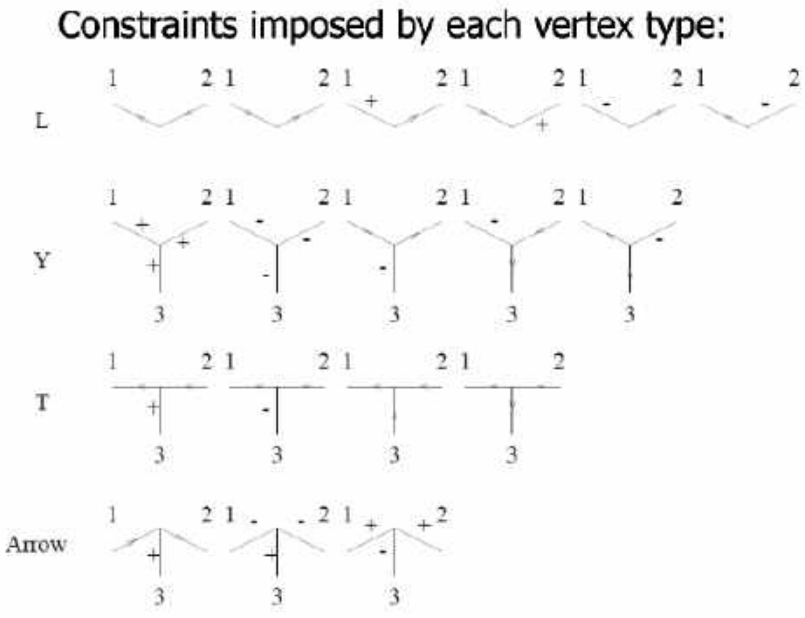




In this problem, the constraints are provided by the vertices. Each vertex is classified into one of four types, based on the number of edges and their angles. For each type, there are only a small set of legal ways to label the incoming edges.







See this contemporary video of the labeller in action.

Using constraint propagation

Because line labelling has strong local constraints, constraint propagation can sometimes nail down the exact solution without any search. A more typical situation is that we need to use a combination of backtracking search and constraint propagation. Constraint propagation can be added at two points in our backtracking search:

- At the start of search, to prune the initial sets of possible values.
- Replacing forward checking during the search.

AC-3

It takes a bit of care to implement constraint propagation correctly. We'll look at the AC-3 algorithm, developed by David Waltz and Alan Mackworth (1977).

The constraint relationship between two variables ("some constraint relates the values of X and Y") is symmetric. For this algorithm, however, we will treat constraints between variables ("arcs") as ordered pairs. So the pair (X,Y) will represent contraint flowing from Y to X.

The function Revise(X,Y) prunes values from D(X) that are inconsistent with what's currently in D(Y).

The main datastructure in AC-3 is a queue of pairs ("arcs") that still need attention.

Given these definitions, AC-3 works like this:

> Initialize queue. Two options
>
> - all constraint arcs in problem [for starting up a new CSP search]
> - all arcs (X,Y) where Y's value was just set [for use during CSP search]
>
> Loop:
>
> - Remove (X,Y) from queue. Call Revise(X,Y).
> - If D(X) has become empty, halt returning a value that forces main algorithm to backtrack.
> - If D(X) has been changed (but isn't empty), push all arcs (C,X) onto the queue. Exception: don't push (Y,X).
>
> Stop when queue is empty

Exercise for the reader

Near the end of AC-3, we push all arcs (C,X) onto the queue, but we don't push (Y,X). Why don't we need to push (Y,X)?

Do attempt to solve this yourself first. It will help you internalize the details of the AC-3 algorithm. Then have a look at the solution.

More information

See the CSPLib web page for more details and examples.

Constraint Satisfaction Problems
Answer to Exercise

In the CSP lectures, I left the following question unanswered:

> Near the end of AC-3, we push all arcs (C,X) onto the queue, but we don't push (Y,X). Why don't we need to push (Y,X)?

There's two cases. First (Y,X) might already be in the queue.

- This is our first pass through all the arcs, when we start working on the constraint problem.
- Some other part of constraint propagation has modified D(X) since the last time we checked (Y,X).

If (Y,X) is already in the queue, then we don't need to add it.

If (Y,X) is not already in the queue, then we've already checked (Y,X) at some point in the past. That check ensured that every value in D(Y) had a matching value in D(X). And, moreover, nothing has happened to D(X) since then, until what we did in the current step of AC-3.

The current step of AC-3 is repairing a problem caused by D(Y) having become smaller. So the situation at the start of our step is that:

- Every value in D(Y) has a matching value in D(X), but
- Some values in D(X) don't have a matching value in D(Y).

Our call to Revise(X,Y) removes that second group of unmatched values. But it doesn't do anything to the first group of values with matches, because the constraint is symmetric. So, after our call to Revise(X,Y), D(X) and D(Y) are now completely consistent with one another.

CS 440/ECE 448
Margaret Fleck

17 Classical Planning

Making toast

We'd like to end up with a toasted and buttered slice of bread. So our goal is something like holding(slice) AND toasted(slice) AND buttered(slice). The individual conjuncts, e.g. toasted(slice), are the main subgoals.

To get toasted(slice) AND buttered(slice), we might decide we need to do this:

> Heat(slice,toaster).
> Spread-on(butter,slice)

But each of these actions requires some set-up:

> Put-in(slice,toaster)
> Heat(slice,toaster).
> Take-out(slice,toaster)
> Spread-on(butter,slice)

But, we can't do any of this without first getting the slice of bread and the butter. So before anything else we should do

> Open(fridge)
> Open(breadbag)
> Take-out(slice,breadbag)
> Take-out(butter,fridge)

Here's a more-or-less final plan:

> Make-Open(breadbag) --> achieves open(breadbag)
> Take-out(slice,breadbag) ---> requires open(breadbag), achieves holding(slice)
> Make-Open(fridge) --> achieves open(fridge)
> Take-out(butter,fridge) ---> requires open(fridge), achieves holding(butter)
> Put-in(slice,toaster) --> requires holding(slice), achieves in(slice,toaster)
> Heat(slice,toaster) --> requires slice in toaster, achieves toasted(slice)
> Take-out(slice, toaster) --> achieves holding(slice)
> Put-on(slice,plate) --> requires holding(slice)
> Spread(butter,slice) --> requires slice on plate, holding(butter), achieves buttered(toast)

Upgraded representation of the world

Plan construction is yet another application of search, but this time with a more powerful representation of states and actions. Previously our world was represented by a small set of variables bound to concrete values such as numbers. E.g.

> Q1 = 3 (The queen in column 1 is in row 3.)

In classical planning, features of the world are encoded using predicates that take states as objects, e.g.

> open(fridge)
> in(slice,toaster)

The state of the world is represented by a logical combination of predicates (e.g. AND, OR, NOT)

> open(fridge) AND NOT open(breadbag)

This style of representation makes our planner look smarter than it is. By writing "spread," we've implied that the computer has some ability to recognize or execute the corresponding real-world action. That connection would have to be made by other parts of our AI system, e.g. the image recognition system and the low-level robotics system. Frequently those systems are a lot less general and robust than one might hope.

Upgraded representation of actions

In our previous search examples, each action set or reset the value of one variable. Actions now take objects as parameters, e.g.

> Take-out(butter,fridge)

Each action has two sets of predicates:

- Preconditions: logical conditions required for an action to happen
- Effects (postconditions): changes that the action makes to the state of the world

So the action take-out(butter,fridge) might have

- Preconditions: in(butter,fridge), is-open(fridge)
- Effects: holding(butter)

The preconditions and postconditions determine how actions can fit together into a coherent plan. They also encode some very limited real-world knowledge about the meaning of each action.

Partial Ordering

Another source of difficulty in classical planning is that the action order is only partly constrained. In our toast example, we could have this ordering:

> Open(fridge)
> Open(breadbag)
> Take-out(slice,breadbag)
> Take-out(butter,fridge)

But this ordering works just as well.

> Open(breadbag)
> Take-out(slice,breadbag)
> Open(fridge)
> Take-out(butter,fridge)

History

Classical planning is called that because it goes back to the early days of AI. A simple version popularized by the STRIPS planner, created by Richard Fikes and Nils Nilsson 1971 at SRI. (STRIPS is short for Stanford Research Institute Problem Solver.)

These days, a planner would most likely be used to manage the high-level goals and constraints of a complex planning problem. For example, it might set the main way points for a mobile robot or a robot doing assembly/disassembly tasks. A neural net or specialized robotics code would then be used to fill in the details.

Stories

A planner can also be used to generate stories. These may be freestanding stories (e.g. like a novel). More often, these systems also include human interaction.

- Storytelling for entertainment, e.g. Automated storytelling projects by Mark Riedl, GA Tech.
- Randomized plots and non-player characters for computer games
- Personalization and randomization of training scenarios (e.g. military exercises, online ethics training)
- Chatbots

Neural nets can do a good job of filling in details, e.g. generating sentences given a sketch of what they should say. But they aren't (at least so far) good at keeping the high-level plot under control. E.g. see this article on automatically-generated Harry Potter stories.

A closely-related problem is story understanding. To understand stories written for a human audience, it helps to have models of what typically happens in them. The observation goes back to Roger Schank's work in the 1970's and 80's. E.g.

- In news reports, common events can be summarized in terms of standard features. E.g. an earthquake has a location, a date, death toll, major aid agencies assisting.
- Everyday actions often follow "scripts" that have only a few variations, e.g. the main events of having a meal in a restaurant.

Understanding a story often requires filling in missing details that are obvious to humans. E.g. two people leave a store and the cashier runs out after them. Why? Well maybe to return a forgotten credit card. A human knows the payment script well enough to realize that it involves returning the card.

The frame problem

Classical planning representations quickly get complex as you move beyond toy problems, so we have to make some simplifying assumptions. For example, the high-level vision of action representations is that they describe how the action changes the world. However, a complete logical representation also requires "frame axioms" that specify which predicates an action does **not** change. For example

> heat(substance,pan) has in(substance,pan) as both a precondition and effect

If we handle such constraints explicitly, our action descriptions end up needing a lot of frame axioms. This is a waste of memory. But, much more important, it is an invitation to bugs: writing out long lists of frame axioms means that some will inevitably be left out. This nuisance is called the "frame problem." It's usually solved by incorporating a default assumption that properties don't change unless actions explicitly change them. (We've been tacitly assuming that.)

Extending our representation

The representation of objects in classical planning leaves much to be desired. Extensions of this framework are beyond the scope of this class. But it's worth thinking about some of the issues, because they continue to be problems even when planning is combined with more modern technology.

First, we need ways to represent type hierarchies for objects, e.g. pie is a type of desert and deserts are a type of food. For example, washing a tablecloth should use the same action as washing a napkin. Eating an apple is similar to, but not identical to, eating rice. So perhaps they should inherit some of their representation from a shared ancestor. Then we might define actions that can be used on any object from a broad range of types.

A closely-related problem is making it possible to have multiple objects of the same type (e.g. two napkins).

Some object properties should be called out as predicates. In a kitchen scenario, many types of object are movable (e.g. eggs, kettle) but some are not (e.g. the counters) and some (e.g. the stove) should be considered non-movable for most routine planning. However, it's not immediately obvious when we should create an explicit predicate (e.g. movable) vs. having rules that target only certain types of objects.

We might want to specify relationships among predicates. For example, how are cold and hot related? Also might want a "Truth Maintenance System" to spell out the consequences of each proposition. For example, automatically account for the fact that the teabag cannot be in two places at once. Such knowledge and inference would allow more succinct (and likely more correct) state and action definitions.

We also have no way to represent substances that must be measured:

- liquids and substances (e.g. flour)
- counting objects of the same type (e.g. ten energy bars)
- actions that can continue for an extended time (e.g. reading)
- actions that take time to finish (e.g. cooking oatmeal)
- properties that decay over time, e.g. milk left in the fridge eventually spoils

But we need to keep the overall representation simple enough to do planning at a reasonable speed.

Harder environments

A final issue is that planning may need to work in less tightly managed environments. These might include:

- Incomplete knowledge of environment (e.g. via sensors)
- Obstacles, other actors might move/change
- What if actions fail?
- How long before I have to finish planning and act?

This might be best handled by a combination of a classical planning and an adaptive technology such as reinforcement learning. But it's not always clear how to make them work well together.

Recap

In a classical planning problem, we have

- States defined by logical combinations of predicates, e.g. open(fridge) AND NOT open(breadbag)
- Actions take objects as arguments, e.g. Take-out(butter,fridge)
- Actions have preconditions and effects (postconditions)

Situation space planning

The most obvious approach to classical planning is "situation space planning". In this method

- Each state is a complete model of the world.
- Each edge is an action that changes the state of the world.

Suppose that we're using situation space planning to make a cake. Our starting state might look like

> start: have(flour), not(have(eggs)), number-of(spoons,3), not(have(cakepan)), ...

What action should we take first? Perhaps we should go buy the eggs. Then the start of our plan might look like:

> start: have(flour), not(have(eggs)), number-of(spoons,3), not(have(cakepan)), ...
> next: have(flour), have(eggs), number-of(spoons,3), not(have(cakepan)), ...

Or we might buy the cakepan next, in which case the next state would be

> start: have(flour), not(have(eggs)), number-of(spoons,3), not(have(cakepan)), ...
> next: have(flour), not(have(eggs)), number-of(spoons,3), have(cakepan), ...

Issues with situation space planning

- very large branching factor
- unnecessarily specific about ordering of unrelated actions

Any planner will eventually need to pin down the order of all actions, so that it can execute the plan. However, it makes sense to make unimportant ordering decisions as late as possible.

Forward and backward planning

We saw that situation space planning can get into trouble because there's too many possible actions at each step. We might try applying backwards or bidirctional search. For example:

> start: typical student apartment
> goal #1: make dinner

It's sensible to see what we can make with available ingredients. (Not ramen AGAIN!!!) But suppose we switch to

> start: typical student apartment
> goal #2: make a birthday cake

We probably don't have the ingredients. So we should start by checking the recipe, so we can plan what to go out and buy.

In planning problems, the goal is often very specific compared to possible successors of the starting state. So it makes sense to do backwards or bidirectional planning. However, we'll see a different approach (partial order planning) that makes a more significant difference.

Hierarchical Planning

Hierarchical planning is another useful idea that doesn't work as well as you might hope. The idea here is that planning problems often have a natural arrangement of tasks into main goals and subgoals. In a complex problem, there might be a multi-layer hierarchy of tasks.

For example, you might sketch out the overall plan for buying groceries:

- Make a list
- Drive to the store
- Buy everything on the list

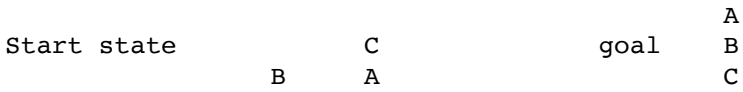
- Drive home
- Put everything away

However, actually doing each of these tasks requires quite a bit more detail. E.g. buying the items might require

- Enter the store
- Get a cart
- Go through the aisles picking up items
- Wait for a cashier
- Pay for the items
- Go to the car
- Unload the cart
- Return the cart

And obviously we could go into even more detail. At the lowest levels, e.g. moving the car around the produce displays, we might wish to switch control over to a neural net or a reinforcement learning model. But some of these details (e.g. making sure we bought everything on the list) might be best handled symbolically.

It is tempting to think that we can base our planning algorithm on this hierarchical structure. That is true to some extent. Unfortunately, strictly hierarchical planning doesn't always work A famous example is the Sussman anomaly (Gerry Sussman, 1975) This comes from a variant of the blocks world. Blocks can be put on an (infinite) table or stacked on top of one another. The robot can move only one block at a time. Suppose we have the following starting state and goal:

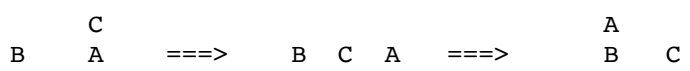


The intended solution is

- move(C,table)
- move(B,C)
- move(A,B)

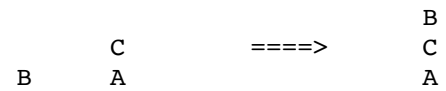
Let's try to do this hierarchically. We have two subgoals: on(A,B) AND on(B,C). Let's first deal with one subgoal and then deal with the other.

Attempt #1: Suppose we try to deal with on(A,B) first. To do this, we clear the block A (so we can pick it up) and then put it on top of B.



Now we turn to goal on(B,C). But we can't put B onto C without unpiling A and B. So we've wasted all our previous work putting A onto B.

Attempt #2: Suppose we try to deal with on(B,C) first. So we make the following move:



Now we turn to goal on(A,B). But we can't put A onto B without unpiling all three blocks, thus undoing on(B,C).

So strictly hierarchical planning doesn't work. We need a more flexible approach to planning. For example, expand what's required to meet subgoals, then try to order all the little tasks. So you can "interleave" sub-tasks from more than one main goal.

Recap

Classical planning problem

- States defined by logical combinations of predicates (e.g. AND, OR, NOT), e.g. open(fridge) AND NOT open(breadbag)
- Actions take objects as arguments, e.g. Take-out(butter,fridge)
- Actions have preconditions and effects (postconditions)

Plan space (partial order) planning

So far, we've used a "situation space" approach to planning, in which our search space looks like:

- Each state is a complete model of the world.
- Each edge is an action that changes the state of the world.

Searching in situation space has a massive branching factor. Patches such as backwards planning and hierarchical planning don't properly address this problem.

When looking at the edit distance problem near the start of the term, we built a much more efficient solution by replacing a search through complete matches with a search through partial matches. A similar idea is used in "plan space" planning (also called "partial order" planning). Our search space now looks like:

- Each state is a partial plan, perhaps missing some actions or ordering links.
- Each edge adds detail to the partial plan.

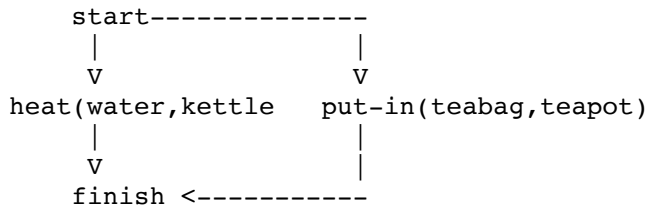
Specifically, in plan space planning, each node in the search graph contains a partial plan. A partial plan includes

- actions
- orderings between actions

This reformulation of the planning problem gives us a search problem with a more manageable set of options at each step.

Plan space example

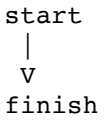
For example, a partial plan for making tea might look like this:



Start and finish are special actions:

- "start": no preconditions, its effects are the information that is true at start of the plan
- "finish": no effects, preconditions are what needs to be true at the end of the plan

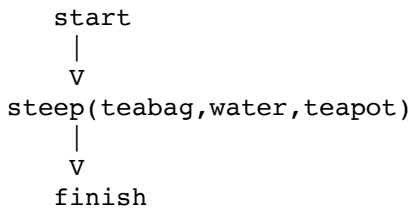
The initial state in the planning process looks like this: the start and finish actions, with start ordered before finish.



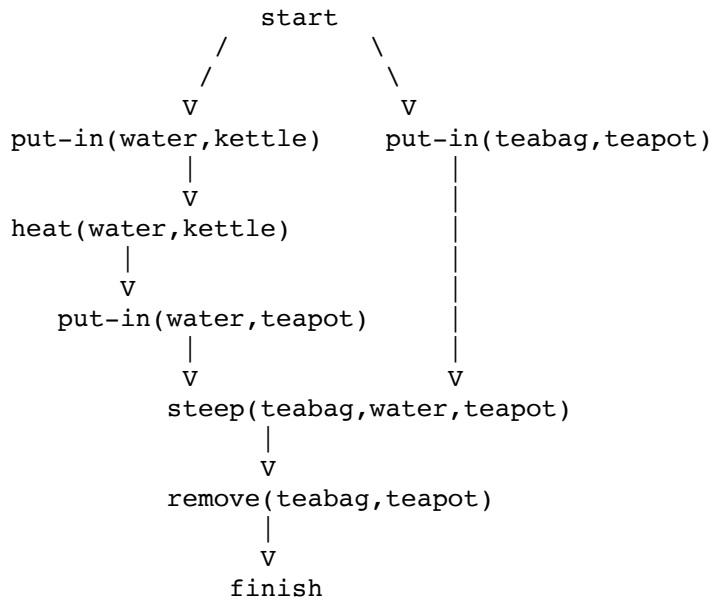
The preconditions and effects are invisible in this diagram. For our tea example, they might be

- effects of start: cold(water), not(in(teabag,teapot)), not(tasty(water), not(in(water,teapot))
- pre-conditions of finish: tasty(water), not(in(teabag,teapot))

Each edge in the planning graph adds more actions and/or ordering constraints. Finish has a precondition tasty(water). We might try to satisfy this using the steep action:



But steeping requires that the water be hot and everything put in the teapot. And then we need to also ensure that the teabag is out of the water at the end. So we add more actions and orderings until we reach a complete plan.



Notice that our path from start to finish may require undoing some properties that we have made true. For example, the teabag is put into the teapot and then later removed.

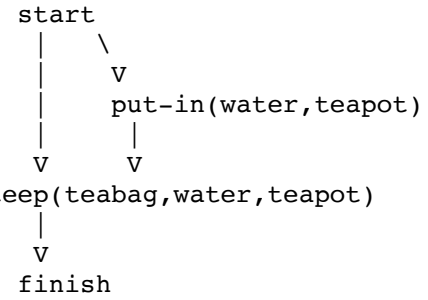
Improving incomplete plans

A key idea in partial order planning is to add actions and orderings of actions only when they are required to ensure that the plan will succeed. The output plan is typically a partial order, in which some pairs of actions are left unordered. When the plan is executed, these pairs can be scheduled in either order or done in parallel. In our example, our final plan leaves several choices for when to put the teabag into the teapot.

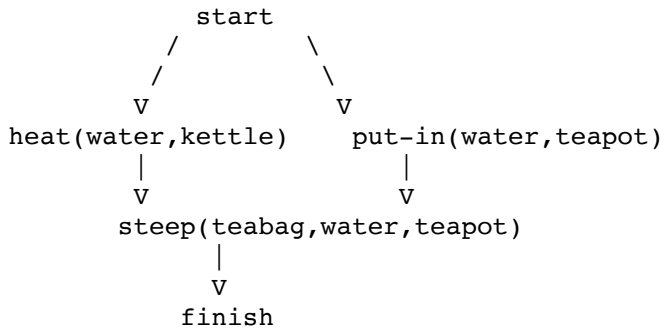
What could be wrong with a partial plan? There are two basic problems for the planner to detect and resolve:

- "open precondition": an action (including finish) has a precondition with no guarantee that it will be true when the action is executed
- "threat": P is true at some point, but an (unordered) action might cause not(P)

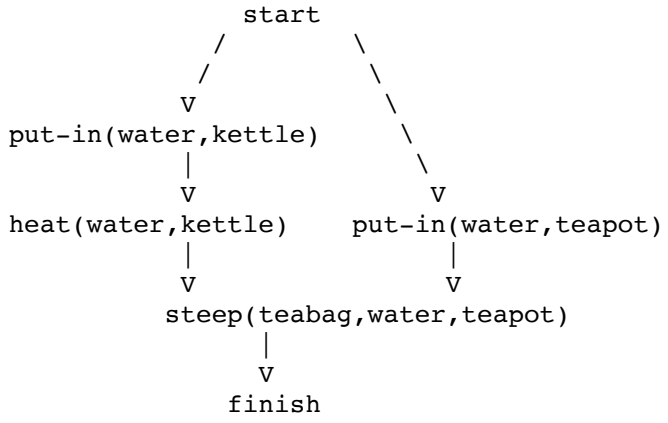
For example, suppose that our partial plan looks as below.



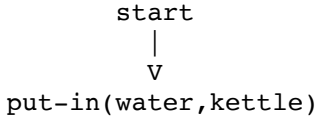
The preconditions of the steep action require that the water be hot. That's an open precondition. We can fix this defect by adding another action that makes the water hot, ordered before the steep action:

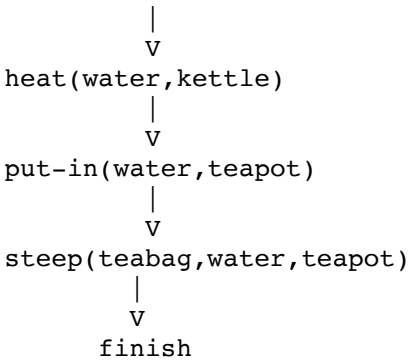


We now notice an open precondition on the heat(water,kettle) action, i.e. that the water needs to be in the kettle. So let's add an action to fix that bug.



Now the heat action has in(water,kettle) as a precondition. As our plan stands right now, this is possible but not guaranteed. Putting the water into the teapot threatens to "clobber" that precondition, if we were do put-in(water,teapot) before heat(water,kettle). We can fix this bug in the plan by adding an ordering link between the two actions.





In general, we have the following set of plan modification operations

- To fix an open precondition
  - add an ordering link to an existing action
  - add a new action

- To fix X threatening to "clobber" Y's preconditions
  - order X after Y
  - order Y after X (leaving Y with an explicitly open precondition)

It's reasonable to expect that a partial plan has only a small set of immediately visible defects. Although our knowledge base may contain a large number of actions, it's reasonable to hope that only a small set would be relevant to filling in an open prerequisite. So our search graph has a much smaller fan-out (typical number of successors to each state) than the search graphs in situation space.

Better Algorithms

In general, classical planning can get very difficult. Consider the Towers of Hanoi puzzle. (Also see demo.) Solutions to this puzzle have length exponential in the number of disks. When modelling realistic situations, planning constraints and background knowledge (e.g. action pre- and post-conditions) can get very large. So it's essential to exploit all available domain constraints. For practical tasks (e.g. robot assembly, face recognition) folks often engineer the task/environment to make the AI problem doable.

A faster algorithm for this type of planning is the the GraphPlan algorithm. Details beyond the scope of this course. See the paper by Blum and Furst (1997) (on the web). Graphplan uses a more complex planning graph with alternating layers of facts and actions. Edges link either facts to actions (preconditions), or actions to facts (effects).

There is also recent work on learning STRIPS-style representations from observations (which may be noisy and/or incomplete). A recent example is Mourao et al 2012 paper (also at Xarchiv).

18 Games

What makes games special?

Game search differs in several ways from the types of search we've seen so far.

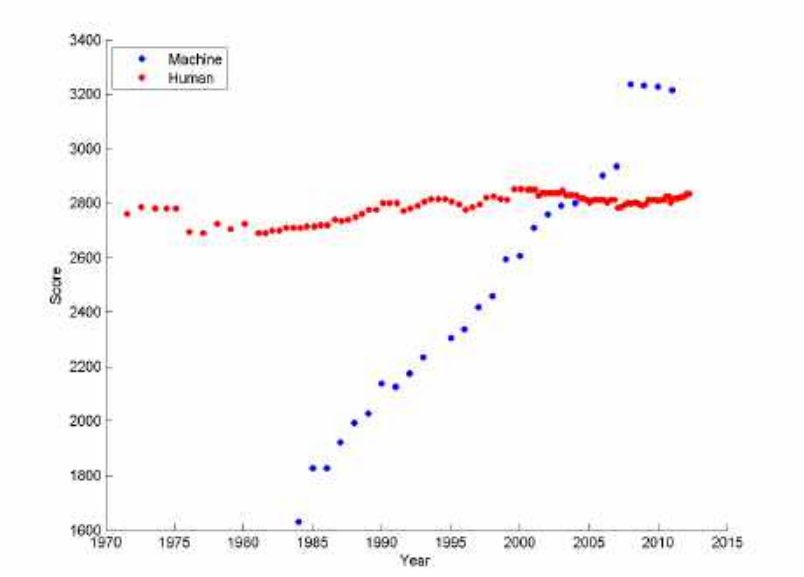
- Adversarial: Two (or more) players competing against one another. Said another way, part of our environment is actively malicious.
- Reward only at the end. (Often just 1=win or 0=lose.)
- Resource intensive (except for the simplest games).
- Very well defined.

For example, chess has about 35 moves from any board configuration. A game may be 100 ply deep. So the full search tree will contain about 35^100 nodes which is about 10^154 nodes.

> → What is a "ply"? This is game computing jargon for one turn by one player. A "move" is a sequence of plies in which each player has one turn. This terminology is consistent with common usage for some, but not all, games.

The large space of possibilities makes games difficult for people, so performance is often seen as a measure of intelligence. In the early days, games were seen as a good domain for computers to demonstrate intelligence. However, it has become clear that the precise definitions of games make them comparatively easy to implement on computers. AI has moved more towards practical tasks (e.g. cooking) as measures of how well machines can model humans.

Game performance is heavily dependent on search and, therefore, high-performance hardware. Scrabble programs beat humans even back in the 1980's. More games have become doable. The graph below shows how computer performance on chess has improved gradually, so that the best chess and go programs now beat humans. High-end game systems have become a way for computer companies to show off their ability to construct an impressive array of hardware.



chess Elo ratings over time from Sarah Constantin

Types of games

Games come in a wide range of types. The most basic questions that affect the choice of computer algorithm are:

- Is it fully observable?
- Is it deterministic or stochastic (involving some random choices)?
- Is it small or big? (So can we do an exhaustive search for best move?)

	fully observable?	**determininistic?**	**size**
Tic-tac-toe	yes	yes	small
Chess	yes	yes	big
Monopoly	yes	no	big
Poker	no	no	big

Some other important properties:

- how many players?
- is reward zero-sum?
- is opponent going to play optimally? badly?
- time limit on how long you can think?
- static or dynamic? (does the world change while you are thinking?)

What is zero sum? Final rewards for the two (or more) players sum to a constant. That is, a gain for one player involves equivalent losses for the other player(s) This would be true for games where there are a fixed number of prizes to accumulate, but not for games such as scrabble where the total score rises with the skill of both players.

Normally we assume that the opponent is going to play optimally, so we're planning for the worst case. Even if the opponent doesn't play optimally, this is the best approach when you have no model of the mistakes they will make. If you do happen to have insight into the limitations of their play, those can be modelled and exploited.

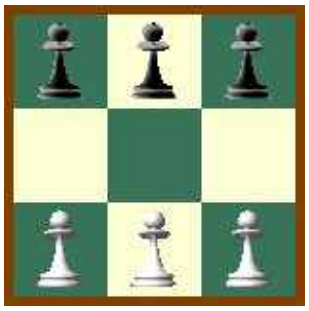
Time limits are most obvious for competitive games. However, even in a friendly game, your opponent will get bored and walk away (or turn off your power!) if you take too long to decide on a move.

Many video games are good examples of dyanmic environments. If the changes are fast, we might need to switch from game search to methods like reinforcement learning, which can deliver a fast (but less accurate) reaction.

We'll normally assume zero sum, two players, static, large (but not infinite) time limit. We'll start with deterministic games and then consider stochastic ones.

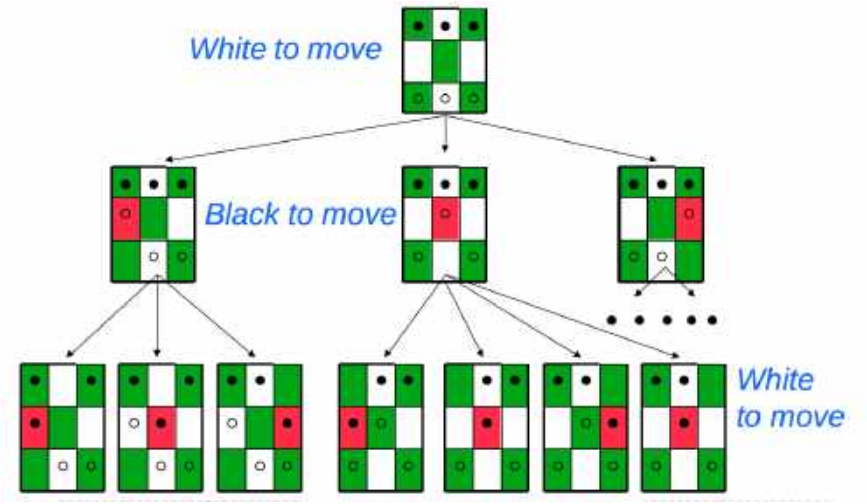
Example: Hexapawn

Games are typically represented using a tree. Here's an example (from Mitch Marcus at U. Penn) of a very simple game called Hexapawn, invented by Martin Gardner in 1962.



This game uses the two pawn moves from chess (move forwards into an empty square or move diagonally to take an opposing piece). A player wins if (a) he gets a pawn onto the far side, (b) his opponent is out of pawns, or (c) his opponent can't move.

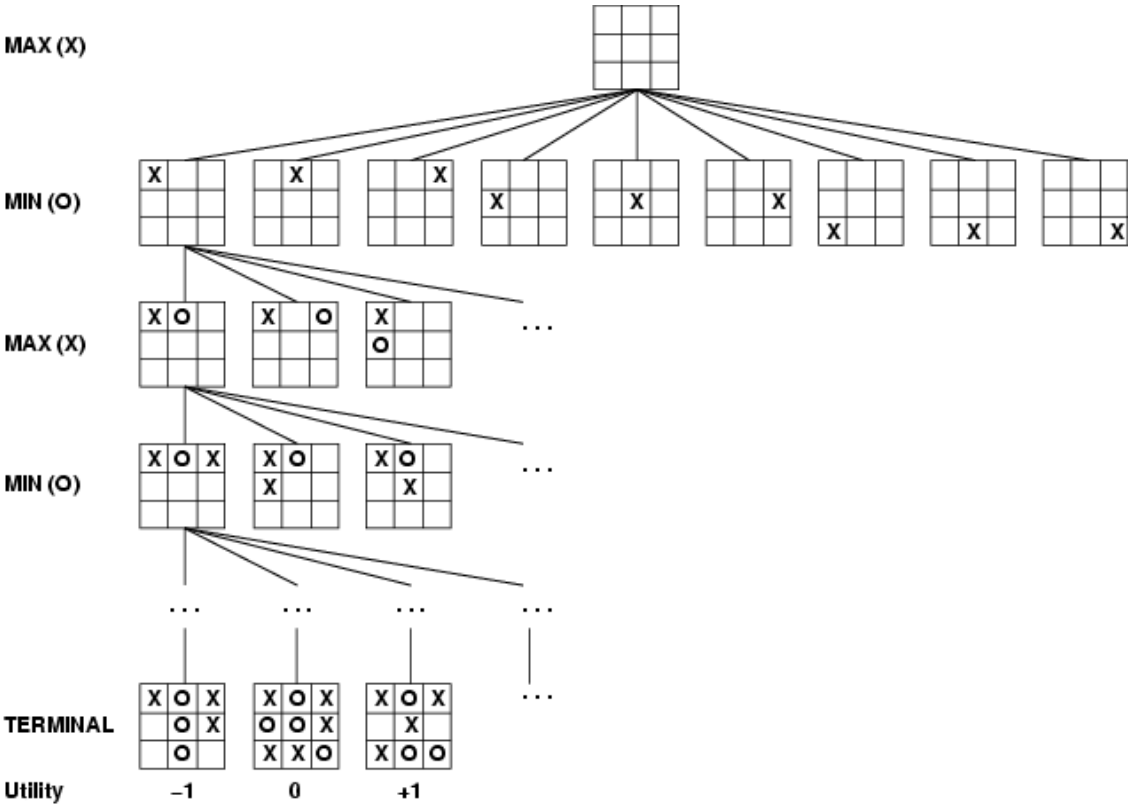
The top of the game tree looks as follows. Notice that each layer of the tree is a ply, so the active player changes between successive levels.



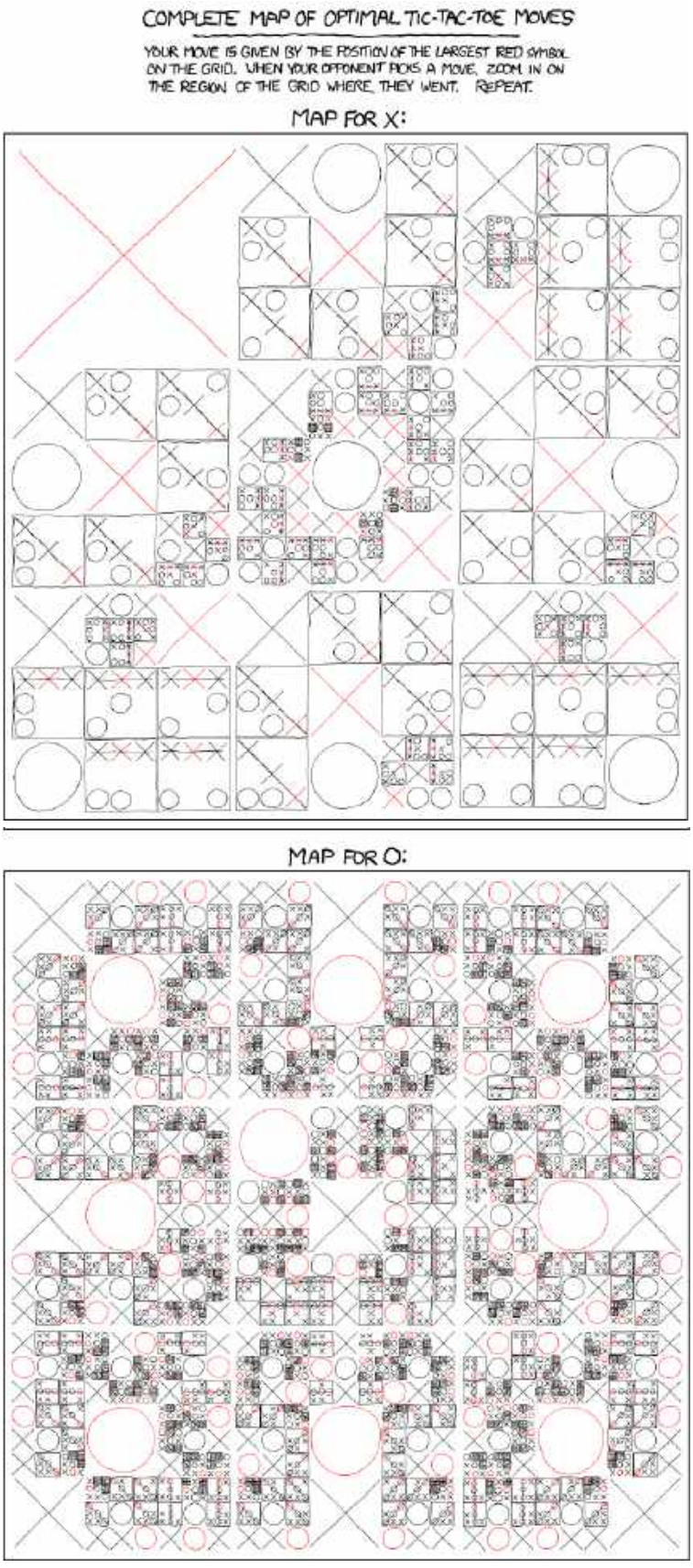
Notice that a game "tree" may contain nodes with identical contents. So we might actually want to treat it as a graph. Or, if we stick to the tree represenation, memoize results to avoid duplicating effort.

Example: tic-tac-toe

Here's the start of the game tree for a more complex game, tic-tac-toe.



Tic-tac-toe is still simple enough that we can expand out the entire tree of possible moves. It's slightly difficult for humans to do this without making mistakes, but computers are good at this kind of bookkeeping. Here's the details, compacted into a convenient diagram by the good folks at xkcd:



Big picture on game algorithms

Game search uses a combination of two techniques:

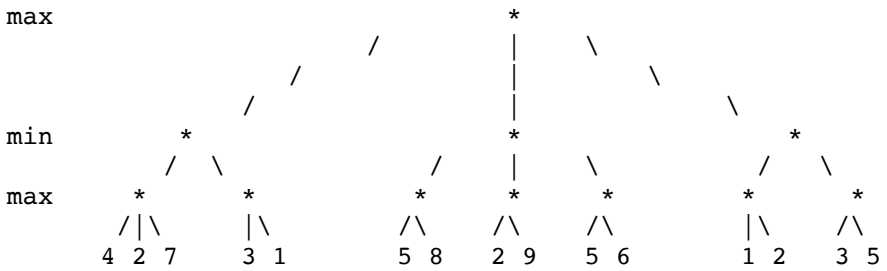
- adversarial search

- evaluation of game positions

Minimax search

Let's consider how to do adversarial search for a two player, deterministic, zero sum game. For the moment, let's suppose the game is simple enough that we can can build the whole game tree down to the bottom (leaf) level. Also suppose we expect our opponent to play optimally (or at least close to).

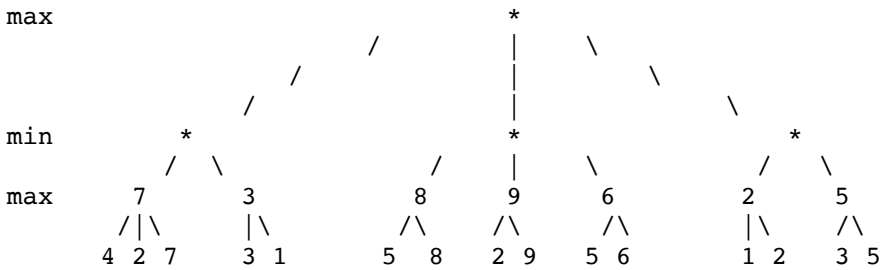
Traditionally we are called max and our opponent is called min. The following shows a (very shallow) game tree for some unspecified game. The final reward values for max are shown at the bottom (leaf) level.



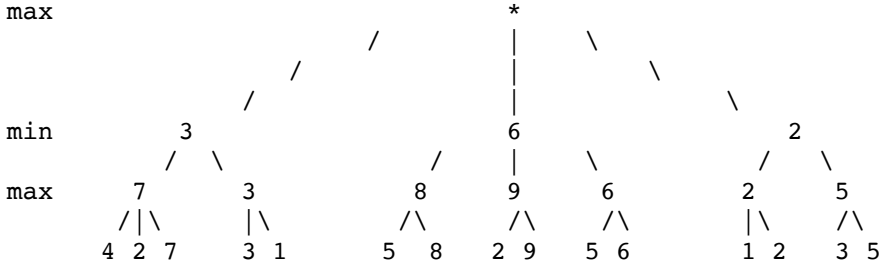
Minimax search propagates values up the tree, level by level, as follows:

- On levels where we move, take the maximum of the child values
- On levels where opponent moves, take the minimum of the child values

In our example, at the first level from the bottom, it's max's move. So he picks the highest of the available values:



At the next level (moving upwards) it's min's turn. Min gains when we lose, so he picks the smallest available value:



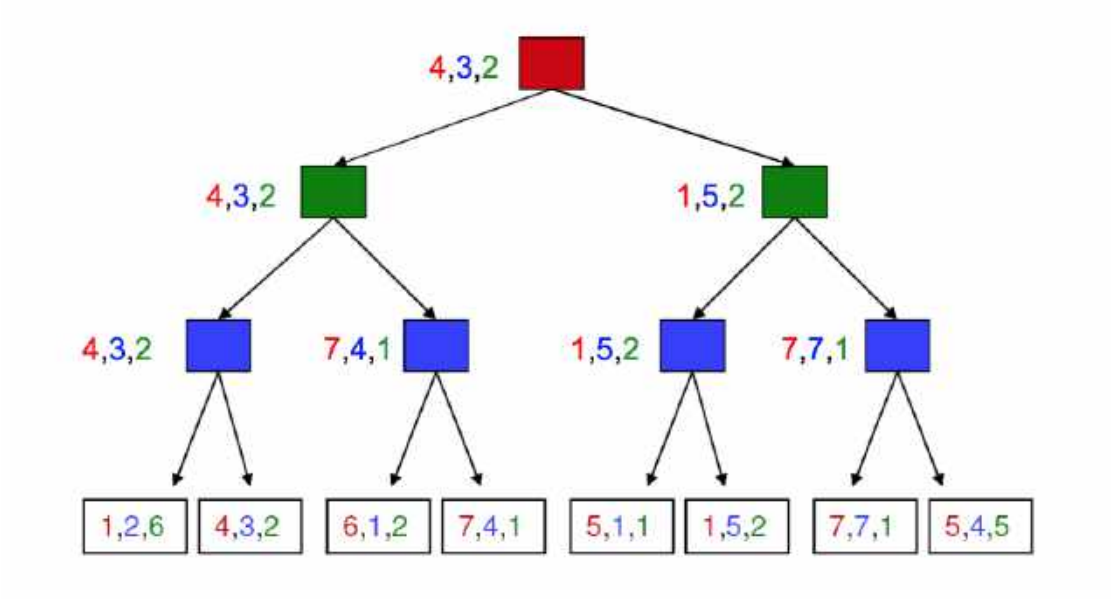
The top level is max's move, so he again picks the largest value. The double hash signs (##) show the sequence of moves that will be chosen, if both players play optimally.

```
                    /             ##          \
                   /              ##               \
min          3                   /    6 ##                 2
            /  \                /     |   ##            /    \
max       7      3            8       9      6         2       5
         /|\     |\          /\      /\     / ##       |\     /\
        4 2 7    3 1       5  8     2 9    5  6       1 2    3 5
```

This procedure is called "minimax" and is optimal against an optimal opponent. Usually our best choice even against suboptimal opponents unless we have special insight into the ways in which they are likely to play badly.

More than two players

Minimax search extends to cases where we have more than two players and/or the game is not zero sum. In the fully general case, each node at the bottom contains a tuple of reward values (one per player). At each level/ply, the appropriate player choose the tuple that maximizes their reward. The chosen tuples propagate upwards in the tree, as shown below.



What if the game tree is too deep?

For more complex games, we won't have the time/memory required to search the entire game tree. So we'll need to cut off search at some chosen depth. In this case, we heuristically evaluate the game configurations in the bottom level, trying to guess how good they will prove. We then use minimax to propagate values up to the top of the tree and decide what move we'll make.

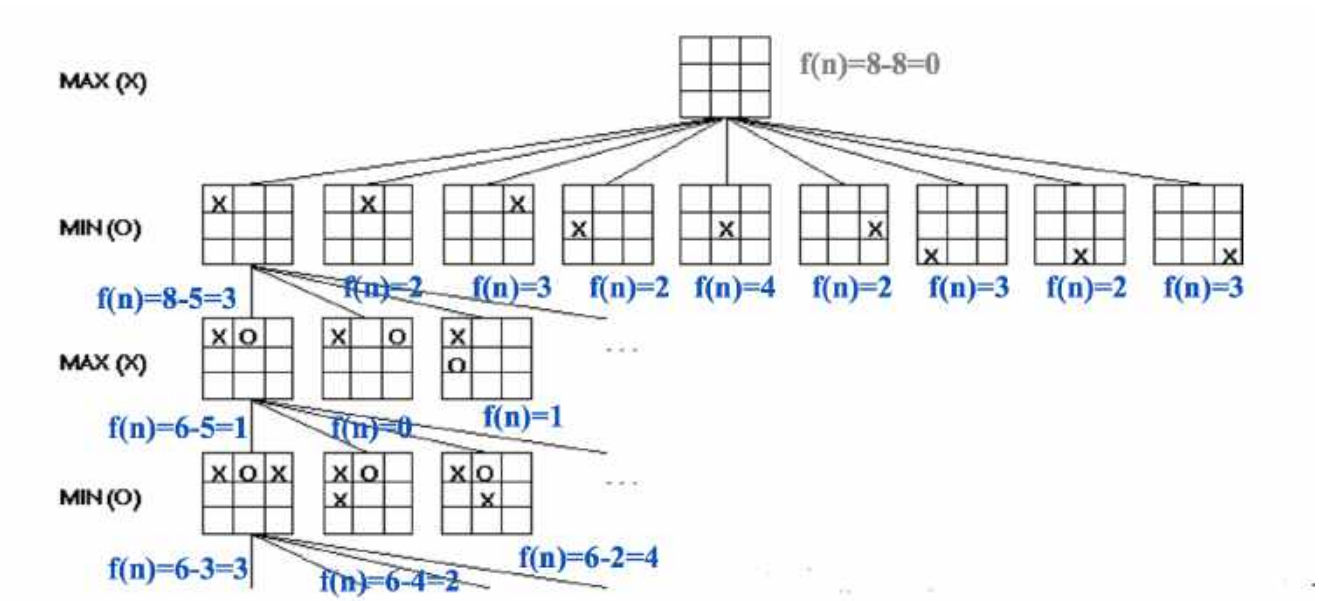
As play progresses, we'll know exactly what moves have been taken near the top of the tree. Therefore, we'll be able to expand nodes further and further down the game tree. When we see the final result of the game, we can use that information to refine our evaluations of the intermediate nodes, to improve performance in future games.

Evaluating tic-tac-toe configurations

Here's a simple way to evaluate positions for tic-tac-toe. At the start of the game, there are 8 ways to place a line of three tokens. As play progresses, some of these are no longer available to one or both players. So we can evaluate the state by

f(state) = [number of lines open to us] - [number of lines open to opponent]

The figure below shows the evaluations for early parts of the game tree:



(from Mitch Marcus again)

Evaluating chess configurations

Evaluations of chess positions dates back to a simple proposal by Alan Turing. In this method, we give a point value to each piece, add up the point values, and consider the difference between our point sum and that of our opponent. Appropriate point values would be

- pawn = 1
- knight = 3
- bishop = 3.25
- rook = 5
- queen = 9

We can do a lot better by considering "positional" features, e.g. whether a piece is being threatened, who has control over the center. A weighted sum of these feature values would be used to evaluate the position. The Deep Blue chess program has over 8000 positional features.

More powerful evaluation functions

More generally, evaluating a game configuration usually involves a weighted sum of features. In early systems, the features were hand-crafted using expert domain knowledge. The weighting would be learned from databases of played games and/or by playing the algorithm against itself. (Playing against humans is also useful, but can't be done nearly as many times.) In more recent systems, both the features and the weighting may be replaced by a deep neural net.

Further optimizations

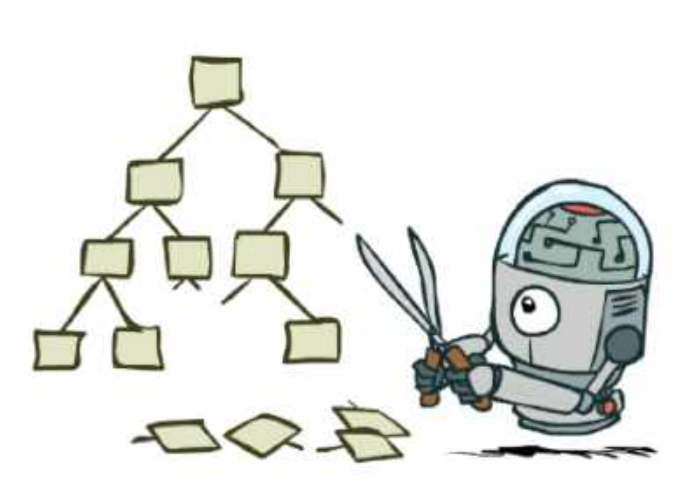
Something really interesting might happen just after the cutoff depth. This is called the "horizon effect." We can't completely stop this from happening. However, we can heuristically choose to stop search somewhat above/below the target depth rather than applying the cutoff rigidly. Some heuristics include:

- "Quiescence search" extend search further if position is "unstable" (e.g. piece in danger).
- "Singular extension": try a few especially strong moves past the cutoff.
- Early pruning: Evaluate states before we reach cutoff depth. Prune unpromising states without expanding them to cutoff depth (as in beam search).

Two other optimizations are common:

- Memoize evaluation of states seen previously. The table of stored positions is often called a "transposition table" because a common way to hit the same state twice is to have two moves (e.g. involving unrelated parts of the board) that can be done in either order.
- Special data tables for opening moves and endgames.

Robot prunes a tree



from Berkeley CS 188 slides

Recap: game search

Each tree node represents a game state. As imagined by a theoretician, we might use the following bottom-up approach to finding our best move:

- Build out game tree to depth d.
- At depth d, use heuristic function to evaluate game positions.
- Use minimax to propagate values from depth d up to root of tree.
- Pick best move available to us at the root node

As we'll see soon, we can often avoid building out the entire game tree (even to depth d). So a better approach to implementation uses recursive depth-first search. So the algorithm for finding our best move actually looks like:

- Do recursive depth-first search, with depth limit d.
- When we hit a node at depth d, use heuristic function to evaluate the game position.
- As we recurse upwards, use minimax to propagate values upwards
- Pick best move available to us at the root node

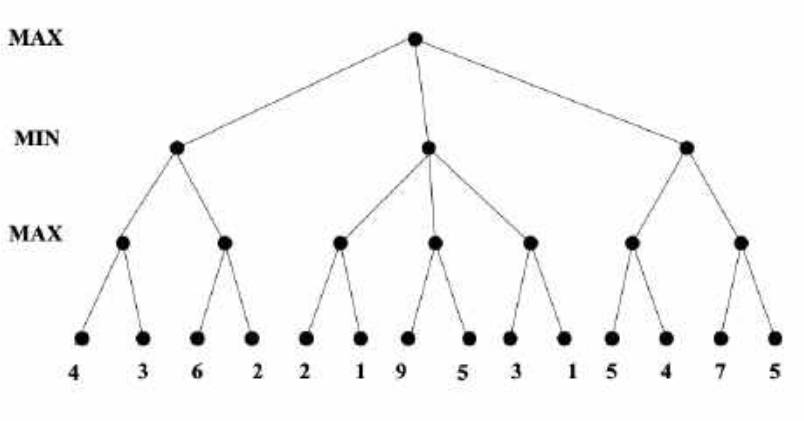
Alpha-beta pruning: main idea

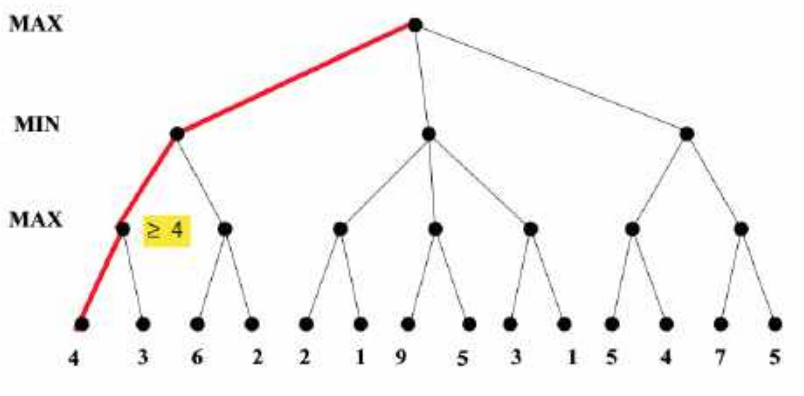
How do we make this process more effective?

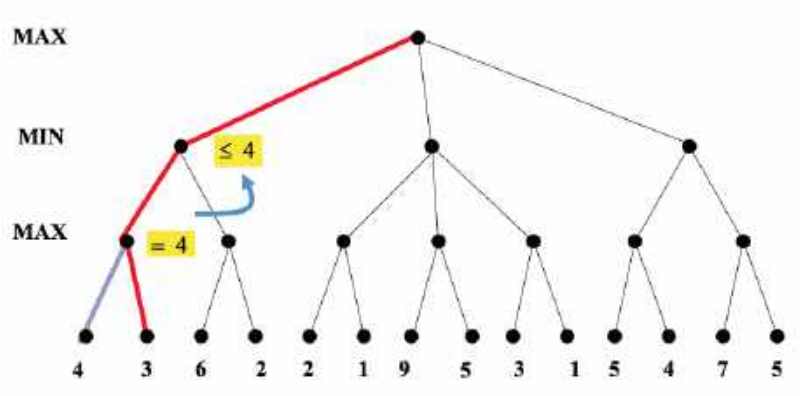
Idea: we can often compute the utility of the root without examining all the nodes in the game tree.

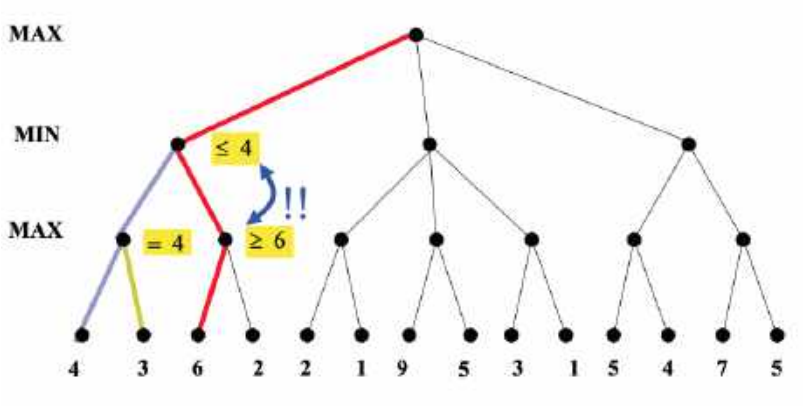
Once we've seen some of the lefthand children of a node, this gives us a preliminary value for that node. For a max node, this is a lower bound on the value at the node. For a min node, this gives us an upper bound. Therefore, as we start to look at additional children, we can abandon the exploration once it becomes clear that the new child can't beat the value that we're currently holding.

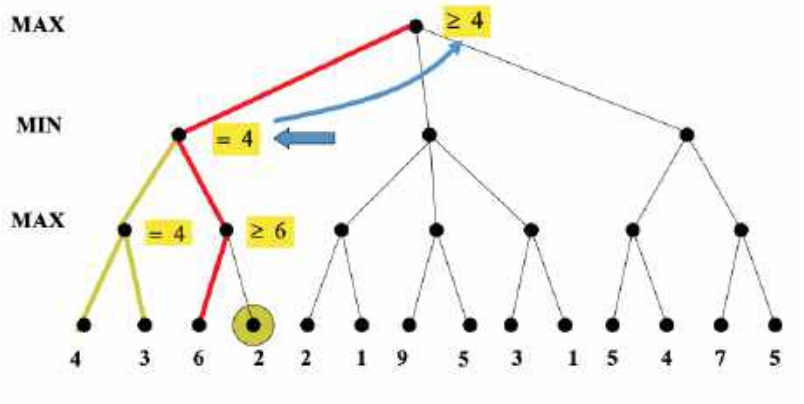
Here's an example from from Diane Litman (U. Pitt).

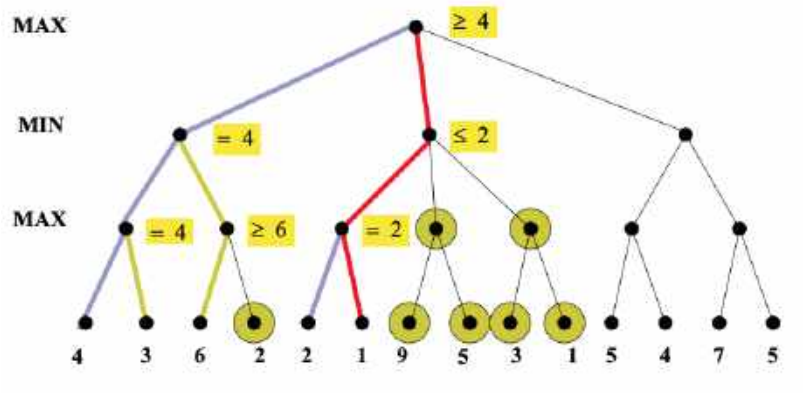


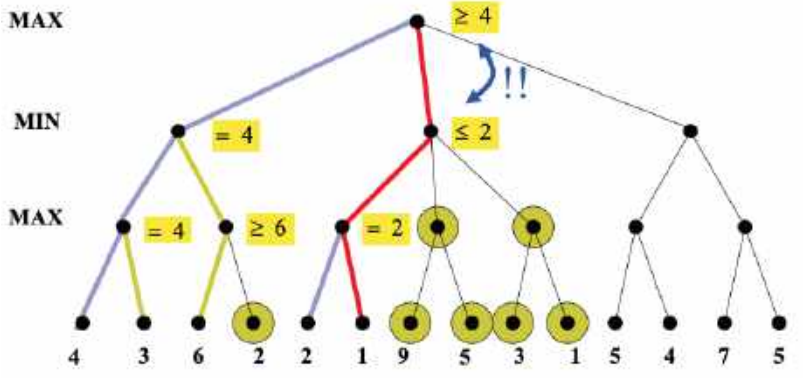


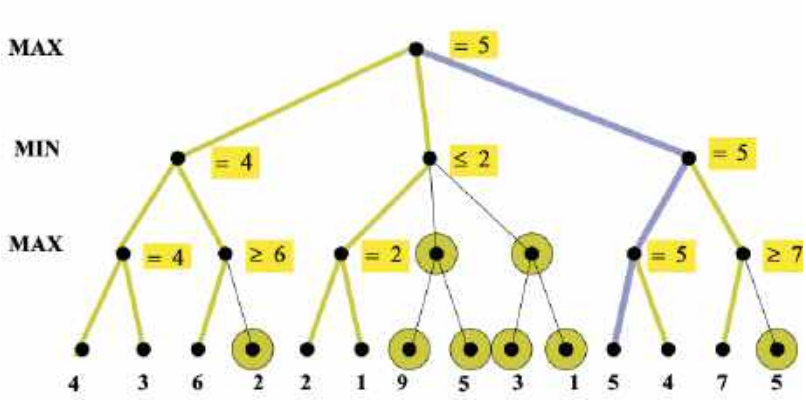



MAX
MIN
MAX
≤ 4
!!
= 4
≥ 6
4 3 6 2 2 1 9 5 3 1 5 4 7 5


MAX
MIN
MAX
≥ 4
= 4
= 4
≥ 6
4 3 6 2 2 1 9 5 3 1 5 4 7 5


MAX
MIN
MAX
≥ 4
= 4
≤ 2
= 4
≥ 6
= 2
4 3 6 2 2 1 9 5 3 1 5 4 7 5


MAX
MIN
MAX
≥ 4
!!
= 4
≤ 2
= 4
≥ 6
= 2
4 3 6 2 2 1 9 5 3 1 5 4 7 5

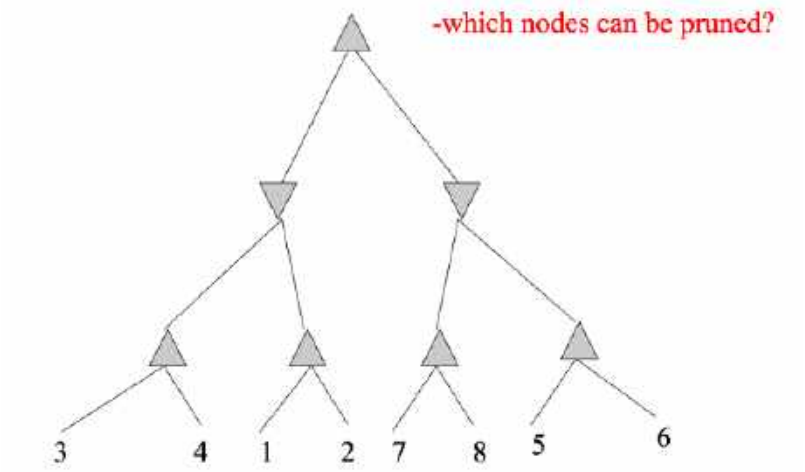


As you can see from this example, this strategy can greatly reduce the amount of the tree we have to explore. Remember that we're building the game tree dynamically as we do depth-first search, so "don't have to explore" means that we never even allocate those nodes of the tree.

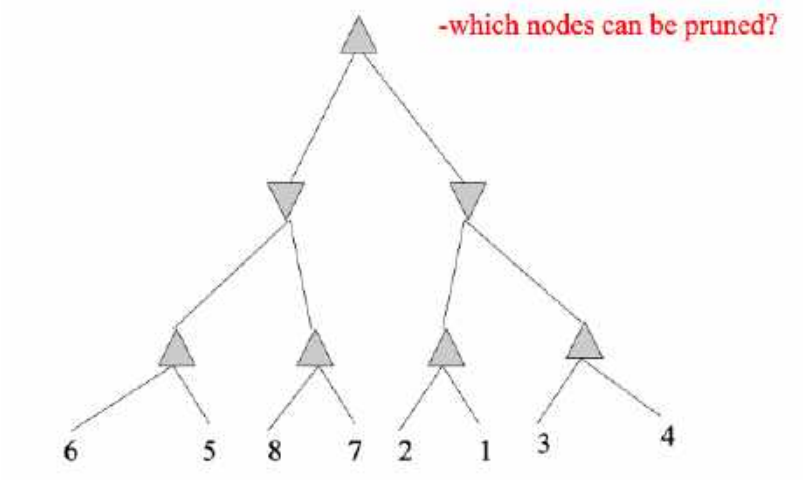
Performance of alpha-beta pruning

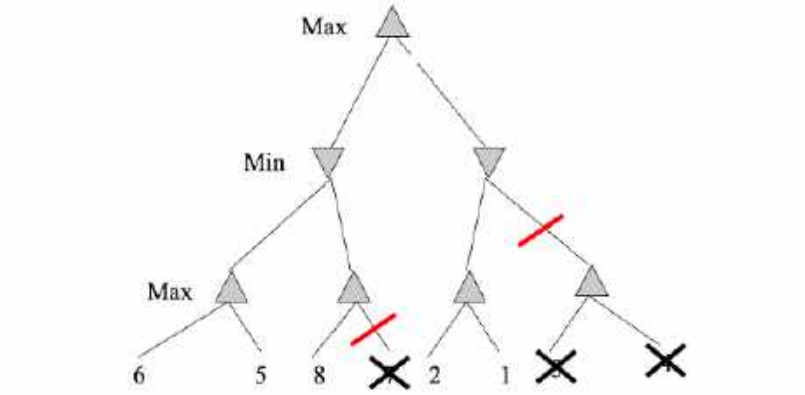
That the left-to-right order of nodes matters to the performance of alpha-beta pruning. Remember that the tree is built dynamically, so this left-to-right ordering is determined by the order in which we consider the various possible actions (moves in the game). So the node order is something we can potentially control.

Consider this example (from Mitch Marcus (U. Penn), adapted from Rick Lathrop (USC)). None of these nodes can be pruned by alpha-beta, because the best values are always on the righthand branch.



If we reverse the left-to-right order of the values, we can do considerable pruning:





Suppose that we have b moves in each game state (i.e. branching factor b), and our tree has height m. Then standard minimax examines $O(b^m)$ nodes. If we visit the nodes in the optimal order, alpha-beta pruning will reduce the number of nodes searched to $O(b^{m/2})$. If we visit the nodes in random order, we'll examine $O(b^{3m/4})$ on average.

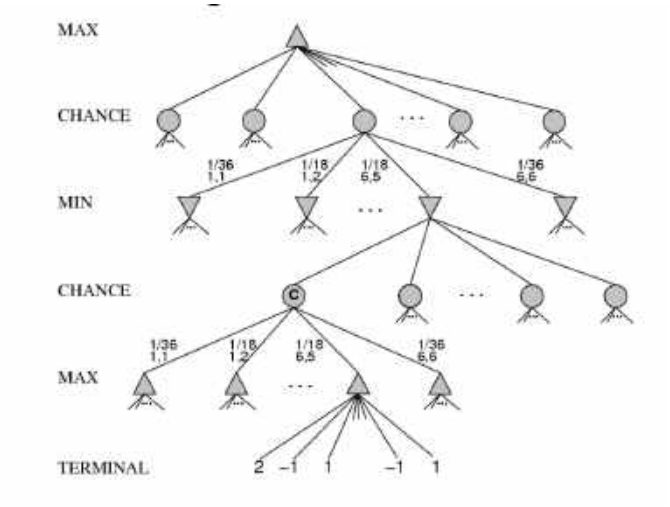
Obviously, we can't magically arrange to have the nodes in exactly the right order, because the whole point of this search is to find out what values we have at the bottom. However, heuristic ordering of actions can get close to the optimal $O(b^{m/2})$ in practice. For example, a good heuristic order for chess examines possible moves in the following order:

- (first) captures
- threats
- forward moves
- (last) backwards moves

Stochastic Games

A stochastic game is one in which some changes in game state are random, e.g. dice rolls, card games. That is to say, most board and card games. These can be modelled using game trees with three types of nodes

- max nodes (our move), upward pointing triangles
- min nodes (opponent's move), downward pointing triangles
- chance nodes (environment's move), circles

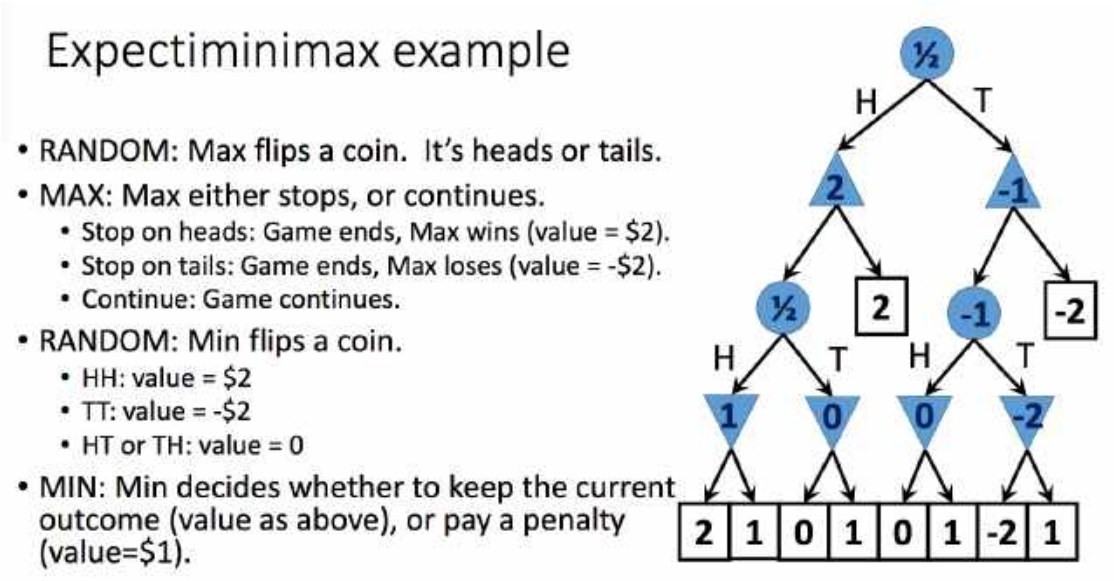


from Russell and Norvig

Expectiminimax

Expectiminimax is an extension of minimax for propagating values upwards in trees with chance nodes. Min and max nodes work as normally. For chance nodes, node value is the expected value over all its children.

That is, suppose the possible outcomes of a random action are $s_1, s_2, \ldots, s_n$. Put each outcome in its own child node. Then the value for the parent node will be $\sum_k P(s_k)v(s_k)$. The toy example below shows how this works:



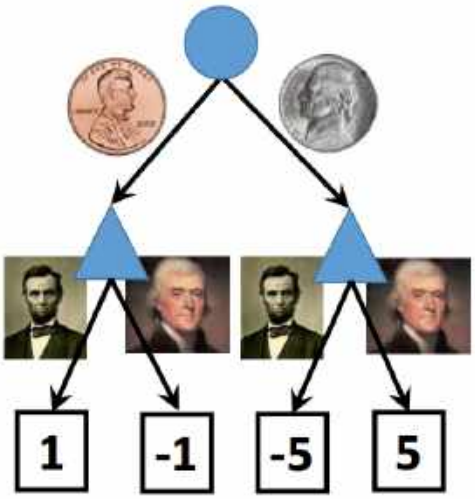
from Lana Lazebnik

This all works in theory. However, chance nodes add enough level to the tree and, in some cases (e.g. card games), they can have a high branching factor. So games involving random section (e.g. poker) can quickly become hard to solve by direct search.

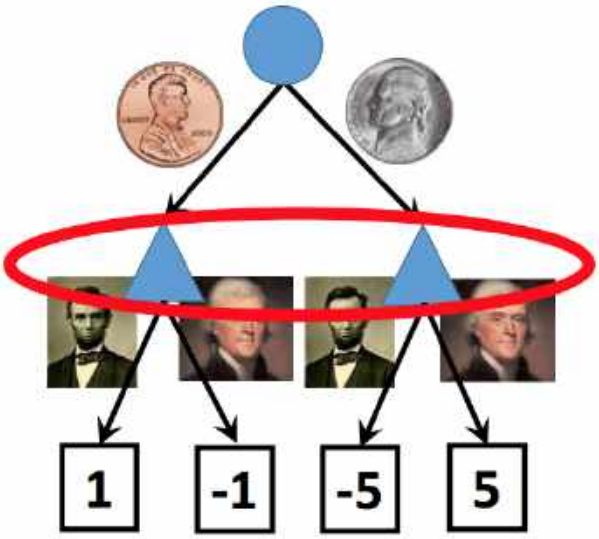
Imperfect information

A related concept is imperfect information. In a game with imperfect information, there are parts of the game state that you can't observe, e.g. what's in the other player's hand. Again, typical of many common board and card games. We can also draw these as chance nodes, as in the example below (from Lana Lazebnik). Notice that Lincoln's face is on the penny and Jefferson's face is on the nickel.

- Min chooses a coin.
- I say the name of a U.S. President.
  - If I guessed right, she gives me the coin.
  - If I guessed wrong, I have to give her a coin to match the one she has.



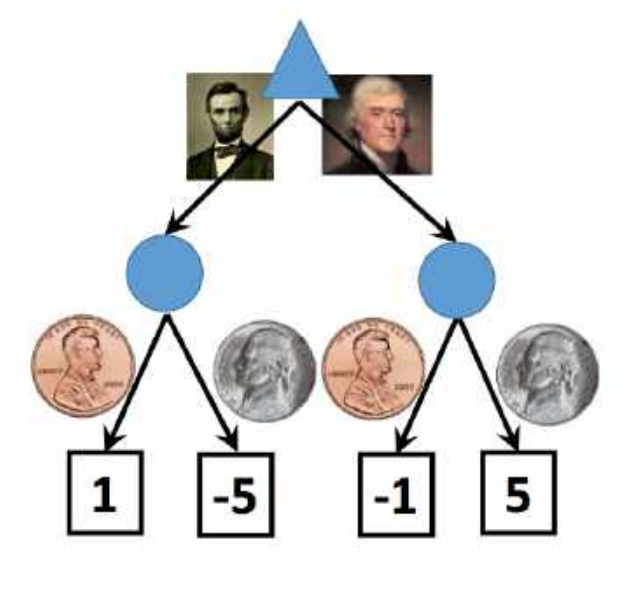
One way to model such a situation is to say that the agent is in one of several states:



In broad outline, this is similar to the standard trick in automata theory for modelling an NFA using a DFA whose states are sets of NFA states. The game playing program would then pick a common policy that will work for the whole set of possible states. For example, if you're playing once and can't afford to lose, you might pick the policy that minimizes your loss in the worst case. In the example above, this approach would pick Jefferson.

In a more complex situation, the group of states might share common features that could be leveraged to make a good decision. For example, suppose you are driving in a city and have gotten lost. You stop to ask for directions because you "have no idea" where you are. Actually, you probably do have some idea. If you're lost in Chicago, you'd ask for directions in English. In Beijing, you'd do this in Chinese. Your group of alternative states shares a common geographic feature. This can be true for complex game states, e.g. you don't know their exact hand but it apparently contains some diamonds.

Alternatively, we could restructure the representation so that we have a stochastic chance node after our move, as in this picture:



This is a good approach if you are playing the game repeatedly and have some theory of how likely the two coins are. If the two coins are equally likely, the corresponding expectiminimax analysis would compute the expected value for each of the two chance nodes (-2 for the left one, 2 for the right one). This again leads to a choice of Jefferson. However, in this case, the best choice will change depending on our theory of how likely the two types of coins are.

But what if our opponent is thinking ahead? How often would she actually choose to pick the penny vs. the nickel? Or, if she's picking randomly from a bag of coins, what percentage of the coins are really pennies?

> >>> Question for the reader: what probability of penny vs. nickel would be required to change our best choice to Lincoln?

Monte Carlo Trees

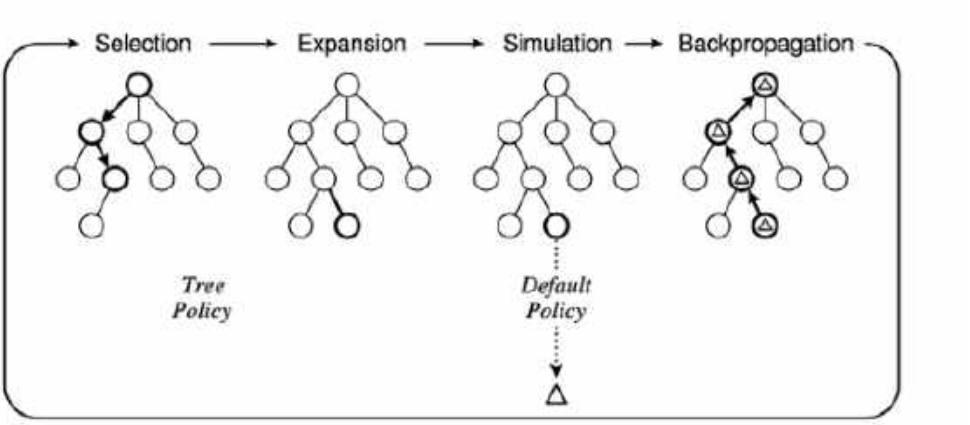
Game trees involving chance nodes can quickly become very large, restricting exhaustive search to a small lookahead in terms of mvoes. One method for getting the most out of limited search time is to randomly select trees. That is

- Build n game trees.
- When building each tree, make a random selection for each chance node.
- Average the resulting node values

This method limits to the correct node values as n gets large. In practice, we make n as large as resources allow and live with the resulting approximation error.

A similar approach can be used on deterministic games where the trees are too large for other reasons. That is

- Build out the tree down to some cutoff level.
- Pick a leaf node to expand further.
- Play game to its end using some default strategy (even random moves).
- Propagate reward values to nodes higher up in the tree.



from C. Browne et al., A survey of Monte Carlo Tree Search Methods, 2012

Poker playing systems

Poker is a good example of a game with both a chance component and incomplete information. It is only recently that poker programs are competitive. CMU's Libratus system recently (2017) beat some of the best human players (geekwire, CMU news).

Libratus uses three modules to handle three key tasks:

- Simplifies game by treating certain hands as equivalent, use this to build initial strategy (more detailed in early parts of game).
- As game progresses, refine strategy for later parts of the game.
- Analyzes opponent's bets to detect and fix issues with current strategy.

It used a vast amount of computing power: 600 of the 846 compute nodes in the "Bridges" supercomputer center. Based on the news articles, this would have 1 pentaflop, i.e. 5147 times as much processing power as a high-end laptop. And the memory was 195 Terabyes, i.e. 12,425 times as much as a high-end laptop. This is a good example of current state-of-the-art game programs requiring excessive amounts of computing resources, because game performance is so closely tied to the raw amount of search that can be done. However, improvements in hardware have moved many AI systems from a similar niche to eventual installation on everyday devices.

For a more theoretical (probably completmentary) approach, check out the CFR* Poker algorithm.

Go

Another very recent development are programs that play Go extremely well. AlphaGo and its successor AlphaZero combine Monte Carlo tree search with neural networks that evaluate the goodness of board positions and estimate the probability of different choices of move. In 2016, AlphaGo beat Lee Se-dol, believed to be the top human player at the time. More recent versions work better and seem to be consistently capable of beating human players.

More details can be found in

- the Alpha Zero paper, and
- from Google's Deepmind website

In a recent news article, the BBC reported that Lee Se-dol quit because AI "cannot be defeated."