CS 161: Fundamental of Artificial Intelligence Lecture Notes (Spring 2022)

Rahul Reddy

Table of Contents

Lecture 1: LISP

LISP

- Functional Language
- Uses Prefix notation in the form of: (op arg1 … argn)
    - Operator can be one of two things: Function (User-Defined or Built In) OR Special Operation
- Lisp expressions
    - Numeric Expressions (evaluates to numbers)
    - Symbolic Expressions
    - Boolean Expressions (evaluates to true or false)
    - Branching Expressions
    - Function Expressions (you get the function definition when you evaluate these expressions)

An atom is a number or a symbol

Examples of Lisp expressions:

> (+ 137 349)
> 486

> (-1000 334)
> 666

> 3
> 3

> (* (* 3 5 1) (- 10 6))
> 60

> (+ 1)
> 0

> (*)
> 1

Two Important Operators:

1. Quote
2. Setq

Quote

If you add a “quote” or a ‘, then:
> (quote (+ 3 1))
> (+ 3 1)

> (‘(+ 3 1))
> (+ 3 1)

Setq
> (setq x 3)
> 3
> x
> 3

> (+ x 2)
> 5

> (+ x y)
> error (because y is not bound to anything)

Examples

> (setq x (+ 1 2 3))
> 6

> x
> 6 (now, x has been bound)

> (setq z ‘(+ x y))
> (+ x y)

> (setq x ‘(+ 1 2 3))
> (+ 1 2 3) => this is a symbolic expression

> x
> (+ 1 2 3) => x is now bound to this expression

> (setq z ‘(+ x y))
> error (because y is not bound even though x is)

LIST Manipulators

List Examples

- (a b c d)
- (a (bx) 17 d)

Accessors/Selectors:

- car (first)
- cdr (rest)

More Examples

> (setq x ‘(a b c d))

> (a b c d) => Here, head is “a” and tail is “(b c d)”

> (car x)
> a

> (cdr x)
> (b c d)

> (car (cdr x)) OR (cadr x)
> b

> (caddr x)
> c

> (setq x ‘(a (b c) d e))
> (a (b c) d e))

> (caadr x)
> b

> (cadadr x)
> c

List Constructors

Cons takes a head and a tail

> (cons ‘a ‘(b c))
> (a b c)

< (setq a 7)
< cons a ‘(b c))
< (7 b c)

> (cons 1 (cons ‘a ‘(3)))
> (1 a 3)

- 1 => This is an element (an atom)
- (1) => This is a list that that has one element

List (e1, … en)

(1 2 3)

Atom

3 (like the base case)

Also a List: ((a b) 3 (2)) where (a b) is a list, 3 is an atom, and (2) is a list

> (cons ‘(a) ‘(b c)) => > ((a) b c)
> (setq x ‘(3))
> (car x) > (car x)
> 3 > NIL which is the empty list or ()

> ‘()
> NIL

NIL is both an atom and a list

- (car NIL) returns NIL
- (car NIL) returns NIL

Questions
> (setq x ‘(3))
> (3)

> (setq x (3))
> error because it is going to try to evaluate “(3)” and there is no function called “(3)”

Lecture 2: LISP Continued

Review

Lisp expression:

- Atom
- List
- (op arg1 … argn) where op is a function: built in or user
- Special Operators: Quote and Setq

Selectors: car (first), cdr (rest)
Constructors: cons, list

Examples

> (setq x '(a b c))
> (car x)
> a

> (cdr x)
> (b c)

(cons 1 (cons 2 NIL)) => (1 2)

- Specifically, (cons 2 NIL) constructs (2)

(setq x (cons 2 (cons 3 NIL)))
(cons 1 (cons x (cons 4 NIL)))

- These two create (1 (2 3) 4)

List

> (list 1 2 3)
> (1 2 3)

> (list 1 (list 2 3) 4)
> (1 (2 3) 4)

Boolean Expressions

- False represented by NIL
- True represented by t (anything that is NOT NIL is also true just like how in other languages: everything considered not 0 is true)

Predicates

(> …)
(< …)

(= …)
(>= …)
(<= …)

> (< 3 1)
> t
> (< 3 1)
> NIL

Functions: atom, listp, null, equal

- The “atom” function checks if something is an atom,
- The “listp” function checks if something is a list.
- The “null” function checks if something is an empty list.
- The “equal” function checks if something is equal.

> (atom 3)
> t
> (atom ‘x)
> t
> (atom ‘(a b))
> NIL

> (listp ‘(a b))
> t

> (setq x ‘(a b))
> (setq y ‘(a b))
> (equal x y)
> t since x and y’s lists are equal (both are “(a b)”)
> (equal ‘x ‘y)
> NIL since x and y themselves are not equal

Questions:
> (atom ‘(a))
> NIL

Boolean Connectives

NOT, OR, AND (not, or and)

- The original form of Lisp is NOT case sensitive, but newer versions of lisp are.
- "AND" and "OR" evaluate from left to right

Examples:

> (NOT t)
> NIL

> (NOT NIL)
> t

> (NOT 3)
> NIL

> (NOT (< 3 1))
> NIL

More Examples: AND returns true when all arguments are true

> (AND (+ 2 3) (+ 1 3))
> 4 because "AND" returns the value of the last expression if the whole expression is true.

> (AND (+ 2 3) (cdr '(a)) (+ 1 3))
> NIL => ("(+ 2 3)" evaluates to 5 and "(cdr '(a))" evaluates to NIL because "AND" sees a NIL, it stops and returns NIL)

In general:

- (car NIL) returns NIL and(cdr NIL) returns NIL
- NIL is BOTH an atom and a list

More Examples: OR returns true when >= 1 argument is true, otherwise false

> (OR (+ 2 3) (+ 1 3))
> 5 because it returns the value of the first expression that is true

> (OR NIL t (+2 3))
> t

> (OR NIL (+2 3) t)
> 5

Branching

You can think of the statement below as a case statement:

```
(cond (bexp exp1… expn)
        (bexp exp1… expn)
        …
        (bexp exp1… expk))
```

The first "bexp" means "boolean expression" and this is evaluated and if it is true, then it evaluates exp1…expn, but if this fails then it goes onto the second one

Examples:

```
> (setq x 3) => This will bound x to 3 before we do the branching statement down below
> (cond (( == x 0) 'Zero)
        ((< x 0) 'Negative)
        ((> x 0) 'Positive))
```

The above branching statement can also be written as how it is down bwlo because "t" acts as the statement that catches the other case that is not considered:

```
> (cond (( == x 0) 'Zero)
        ((< x 0) 'Negative)
        (t 'Positive))
```

Function Expressions

```
> (defun square (x)
        (* x x))
> (square 3)
> 9
> (square (square 3))
> 81
```

```
(defun <function> (arg1… argn)
        <body>)
```

> (square '3)

> error because this "3" is not a number, but it is a symbol. The function will be typing to multiply a symbol by a symbol rather than what the function is actually defined to do)which is multiple a number by a number).

```
> (defun abs(x)
        (cond (( = x 0) 0)
              ((> = x 0) x)
              ((> x 0) x)
              ((t (- x)) …)
```

(-x): returns the negative value of x

Binding Variables

```
> (let ((x 3)
       (y 4))
    (+ x y))
```

> 7 => x is set to 3, y is set to 4, then we evaluate (+ x y) and get 7)

```
> x
```

> error (since the "let" binding is only temporary and x is not bound afterwards)

> (setq x 5) => setq is a global setting for a variable which is why we do not use it in the homeworks and let is a local setting for a variable which is recommended in general programming languages

```
> (+ (let ((x 3))
        (+ x (* x 10)))
      x)
> 38
```

Let does assignments in parallel

```
> (setq x 2)
> (let ((x 3)          ; (x 3) sets x to 3
        (y (+ x 2))) ; y is being set to x + 2, but we use x = 2
   (* x y))
> 12
```

```
> (let ((x 3)          ; (x 3) sets x to 3
        (y (+ x 2))) ; y is being set to x + 2 => 3 + 2 = 5
   (* x y))
> 15
```

Functions (recursive)

```
(defun odd? (n)
        (cond ((= x 0) NIL)
        ((= x 1) t)
        (t (odd? (- x 2) …)
```

(odd? 5) returns t and (odd? 4) returns NIL

Factorial: n! = {1 if n = 0
n(n-1)! Otherwise }

```
(defun factorial (n)
        (cond ((= n 0) 1)
                (t (* n (fact (- n 1) …)
```

Examples:

```
(defun sum-list (L)
        (cond (null L) nil)
                (t (+ (car L)
                     (sum-list (cdr L) ))))
```

```
(defun member? (x L)
        (cond (null L) nil)
                ((equal x (car L)) t)
                (t (member? x (cdr L)))…)
```

General Question:

- "=" is general and it works on numbers too. It can check whether two expressions are the same. They could be atoms or lists.
- "equal" is

```
(defun Last (L)
        (cond (null (rest L) first L) ; if the list is empty or only has 1 element, then (cdr L) is null
                (t (Last(cdr L)) )…)
```

```
(defun nth (L n)
        (cond ( (= n 0) (car L))
                (t (nth (cdr L) (- n 1))) … )
```

```
(defun remove (e L)
        (cond ((null L) nil)
                (equal e (car L)) (remove e (cdr L)))
                (t (cons (car L) (remove e (cdr L)))...)

(defun append (L1 L2)
        (cond ((null L1) L2)
                (t (cons (car L1)
                        (append (cdr L1) L2)))...)
```

Lecture 3: Problem Solving Using Search

Major Components of the Course:

1. Problem Solving Using Search
2. Symbolic (Logic) - Part of Knowledge Representation & Reasoning
3. Numeric (Probability) - Part of Knowledge Representation & Reasoning
4. Machine Learning

Problem -> Search Formulations -> Search Engine -> Solution

Search Formulations:

1. Initial State
2. Final State
3. Actions
4. Heuristic

Search Strategy:

- Blind/Uninformed
- Heuristic/Informed

Other Problems: Constraint Satisfaction Problems (CSPs) and Two-Player Games

8-Puzzle

The numbers can slide and sliding one number from one square to an open square is an action

- There are four possible ways to move (up, down, left, right)

IS:

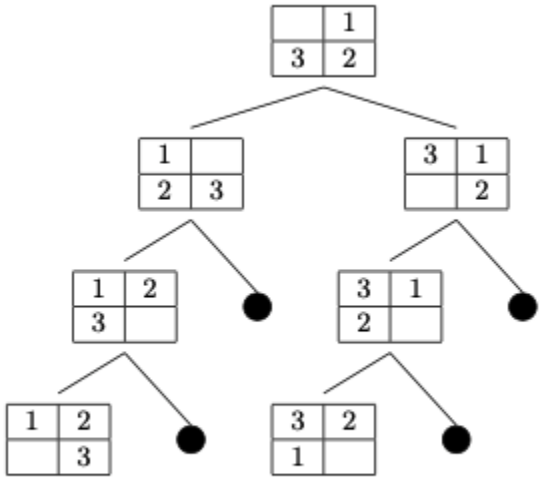
7	6	1
4	5	
8	3	2

FS:

1	2	3
8		4
7	6	5

Search Formulations:

1. Initial State (IS)
2. Final State (FS)
3. Actions



9

For the following search tree below, we can see that each dot represents dead-end repeated paths. Also, we can see that:

- Each path has a cost
- This search tree is infinitely large:

Overall, we are trying to optimize two different things

1. Cost of a solution
2. Cost of finding a solution

Note: The cost of an action does not have to be 1, so in principle, a cost with 5 actions may be worse than a cost with 10 actions.

Planning

Actions are **deterministic**: If we are given a state and an action, then there is another state that results from it (only one state).

- We know the initial state

Missionaries & Cannibals OR River/Boat Problem

- Initial State (IS): We have three missionaries and cannibals with a boat on the east side of a river
- Final State (FS): We want the three missionaries and cannibals AND the boat to end up on the west side of a river
- Boat can take at most 2
- We cannot have more cannibals than missionaries on any side at any point
- We can have all all cannibals on one side
- Boat cannot travel empty

Actions:

- We can move a cannibal on the boat
- We can move a missionary on the boat
- We can move two cannibals on the boat
- We can move two missionaries on the boat
- We can move a cannibal and a missionary on the boat

When going from state_i -> state_j the "->" represents an action

- State_j must be a legal state
- The action that causes state_i to go to state_j must be a legal action

Communicating with a Search Engine

1. Initial State
2. Final State
3. Actions: legals states & actions
    - Successor function takes one state and then creates other states from it (state1, state2, … staten) in the form of tree

State (as a LISP problem)
(A B C)

- A: number of X at boat location
- B: number of O at boat location
- C: boat location (t = boat on right, nil = boat on left)

Initial State (IS): (3 3 t)
Final State (FS): (3 3 NIL)

> (succ_fn '(3 3 t))
> ((0 1 NIL) (1 1 NIL) (0 2 NIL)) => this is what is returned

What Contributes to the difficulty of a search problem

1. Number of possible states (state space)
2. Branching factor: the upper bound of the choices you can go from one node to another
3. Depth of solution

Constraint-Satisfaction Problems (CSPs)

N-Queens Problem

We want to place N queens on an NxN Chess Board such that none can capture the other (no two Queens can be in the same row, column, or diagonal)

Initial State (IS): Empty Board
Final State (FS): All N Queens added to the board such that none attacks the other queens
Actions:

Observations: The search tree is finite and the depth is fixed.
All paths have the same cost.

Other Real-World Applications of CSPS

- Pathfinding, Navigation, Scheduling (this is a big one!)

Lecture 4: Uninformed (Blind) Search Strategies

Review
Problem -> Search Formulations -> Search Engine -> Solution

Search Formulations:
1. Initial State
2. Final State
3. Actions (successor function)
4. Heuristic (require more information)

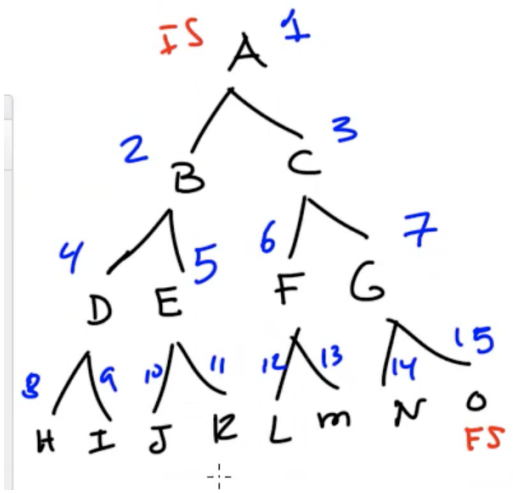
Search Strategy:
- Blind/Uninformed
- Heuristic/Informed (more powerful)

The Search Process
Terminology:
- **Fringe/Frontier**: Nodes that have not yet been checked for if it is the goal (or Final State - FS)
- **Expanding** a node: 1) Check if Final State (FS), 2) Generate Children
    - Nodes on the fringe are waiting to be expanded
- **Generating Children**
- A "**search strategy**" is a criteria for decide which node on the fringe to expand next

Evaluating Search Strategies
1. Completeness: Does it always find a solution if one exists?
2. Time Complexity: number of nodes generated/expanded
3. Space Complexity: maximum number of nodes in memory
4. Optimality: does it always find a least-cost solution?

Time and space complexity are expressed in terms
- b - branching factor
- d - depth of least-cost solution
- m - maximum depth of search tree (could be infinite)

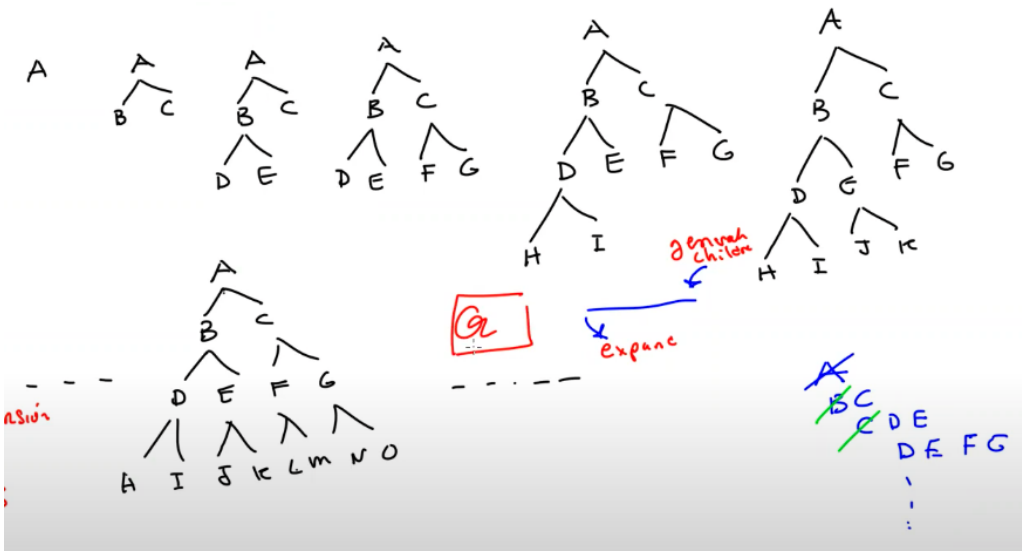
(BFS) Breadth-First Search ("Shallowest Node First")
- Overall Idea: It picks the one with the smallest depth to expand next



The picture above is how we would number nodes according to expansion using BFS. We break ties with alphabetical order in this specific case.

Observation:

If we maintain the fringe as a **queue**, we will get something like the picture down below:



Using a queue for BFS will yield us something like this:

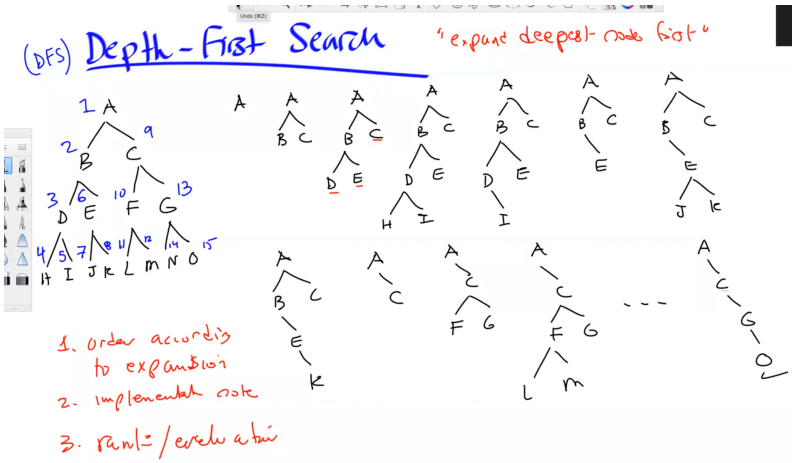
A
BC
CDE
DEFG
EFGHI…

Evaluating BFS

1. Complete? **Yes** (because of the systematic fashion in which the search algorithm works - it goes level by level)
2. Time Complexity? **O(b^d)**
3. Space Complexity? **O(b^d)**
4. Optimal? **Yes** (because it will not touch something at level 17 before level 16)

(DFS) Depth-First Search ("Expand Deepest Node First")

- As opposed to BFS (where we use a queue), we now use a stack!



Evaluating DFS

1. Complete? **No** (because we could have an infinite tree)
2. Time Complexity? **O(b^m)**
3. Space Complexity? **O(bm)**
4. Optimal? **No**

Depth-Limited Search

- This is effectively DFS with depth limit e

Evaluating Depth-Limited Search

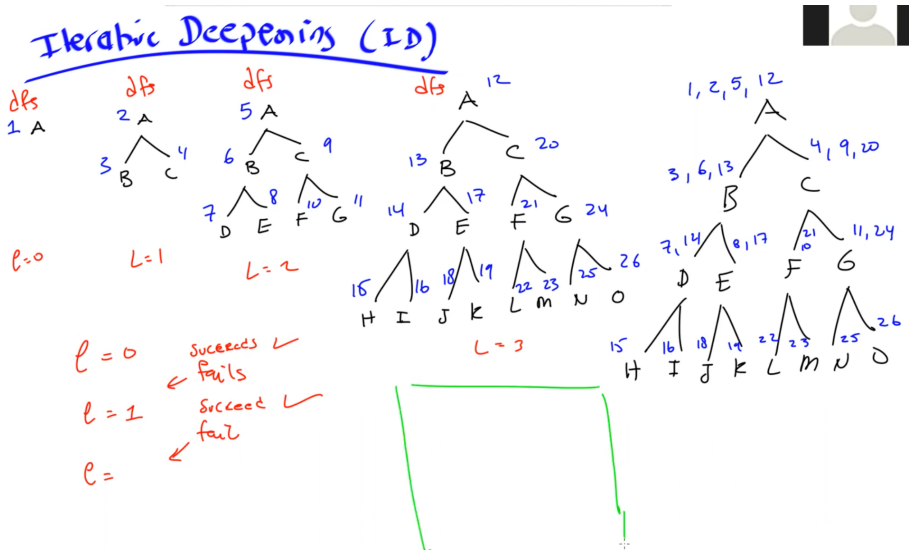
1. Complete? **No** (Yes, but only if d <= e)
2. Time Complexity? **O(b^e)**
3. Space Complexity? **O(be)**
4. Optimal? **No**

Iterative Deepening Search (ID)

- This is effectively Depth-Limited Search, but we start from l = 0 and increase the depth to l = 1, then l = 2, etc until we reach a goal state

Evaluating Iterative Deepening Search

1. Complete? **Yes**
2. Time Complexity? **O(b^d)**
3. Space Complexity? **O(bd)**
4. Optimal? **Yes**



Time Complexity of Iterative Deepening (ID) Search

ID will expand

L = 0: b^0(b/b-1)
L = 1: b^1(b/b-1)
L = 2: b^2(b/b-1)
…
L = d: b^d(b/b-1)

---

b^d((b/b-1))^2 => this is after adding them all up

Tree

- depth d and branching factor b leads to: b^d(b/b-1)

---

b^d ((b/b-1))^2: Only ratio = b/(b-1)

B = 2 => 2, B = 3 => 1.3, B = 4 => 1.33

Lecture 5: Informed (Heuristic) Search

Review

Search Problem -> Search Formulations -> Search Engine -> Solution

Search Problem:

1. Initial State
2. Final State
3. Successor Function
4. Heuristic

Search Strategy:

- Blind/Uninformed
- Heuristic/Informed

Uniform-Cost Search (UCS)

- **This generalizes Breadth-First Search (BFS) which allows arbitrary costs for actions > 0.**
- This chooses whatever is on the fringe with the current least cost and expands that node. Expanding includes checking if we are at the goal state and then generating the children.
- UCS is also known as Dijkstra's Algorithm.
- We goal check on expansion and UCS is based on **g(n)**

When we expand a node, we:

- Check if goal
- If not, generate children

Evaluating UCS

Supposed epsilon (e) is the smallest cost of any action and the optimal solution has cost c*, then we may have to go down to depth ceiling(c*/e).

1. Complete? **Yes**
2. Time Complexity? **O(b^(c*/e))** (This can NOT be better than BFS since BFS is a special case of UCS)
3. Space Complexity? **O(b^(c*/e))**
4. Optimal? **Yes**

Greedy (Best-First Search)

- This is based on straight-line distance, **h(n)**, which is also essentially an estimate of the distance to the goal state or how close we are to the goal itself.
- We do the saem search strategy as UCS, but instead, we use **h(n)** or the "heuristic" only and not **g(n)**.

Evaluating Greedy (Best-First) Search

1. Complete? **No**
2. Time Complexity? **O(b^m)**
3. Space Complexity? **O(b^m)**
4. Optimal? **No**

***A** Search**

- This is based on **f(n)** = **g(n)** + **h(n)** as the evaluation function
    - **g(n)**: cost so far to reach n
    - **h(n)**: estimated cost to goal from n ("heuristic")
    - **f(n)**: estimated total cost of a path that goes through n
- If you are doing goal checking during generation instead of during expansion, then you will have stopped at the less optimal solution (like in Greedy Best-First Search)

The heuristic must be "admissible" and furthermore:

h(n) <= h*(n) where h*(n) is the least cost to go from n to goal
h(n) >= 0
h(G) = 0 for a goal state G

*A** with an admissible heuristic (h(n)) is optimal:

The higher that h(n) becomes, the faster that *A** becomes
Also, when h(n) = 0, *A** Search just becomes Uniform Cost Search

Heuristic: h(n)

If we consider the 8 PUZZLE, there are two heuristics we can consider (which are both admissible):

h1(n): Number of tiles that are not in the correct position (admissible b/c you can NOT overestimate w/this heuristic)
h2(n): Manhattan distance = number of moves or horizontal + vertical distance
Here, h2(n) >= h1(n) where h2 dominates h1

Comparing Search Strategies

IDS is uninformed while *A**(h1) is informed and we can already see the massive difference.

In addition, we can also see that *A**(h2) is the better heuristic compared to *A**(h1).

Node Count**:**

D (depth)	IDS	*A**(h1)	*A**(h2)
2	10	6	6
6	680	20	18
8	6384	39	25
12	36404	227	73
14	3473941	539	113
16	…	1301	211
…	…	…	…
24	…	39135	1641

If we have h1 and h2 that are both admissible, BUT none dominates the other, then we would want to take h(n) = max(h1(n), h2(n)).

This is because we want a bigger heuristic! The larger the number, the better the estimate is. The estimate is trying to tell us how much we are going to pay or how much we expect to pay.

In general, it is NOT hard to give an admissible heuristic since we could just return 0 for every node (h(n) = 0). In fact, *A** search with h(n) = 0 is really just UCS. It is harder, however, to create a heuristic that is good as well.

Lecture 6: Constraint Satisfaction

Constraint Satisfaction Problems (CSPs)

- Standard formulation of a search
- Standard search strategy
- Improvements: six total

Constraint Satisfaction

- We have a set of discrete variables (x1… xn) with values (having the domain D1… Dn)
- Constraints: allowable combinations of value

Constraint Graph (Primal Graph)

This is an abstraction of the Constraint Satisfaction Problem

1. We can create a node for every variable (so if we have 7 variables, then we have 7 nodes)
2. If two variables are involved in the same constraint, then we put an edge between them.

If the constraint graph is a tree, then the CSP can be solved efficiently (in polynomial time)

Types of Constraints

1. Unary Constraints: An example would be if some state can NOT be colored green (unary means it only involves one variable).
2. Binary Constraints (Binary CSP): An example would be if some state can NOT be colored the same color as another state (binary means it involves two variables).
3. Higher Order Constraints: An example would be if it involves greater than two (multiple) variables.

Two Categories of Problems

- Hard Constraints: These constraints NEED to be satisfied and any solution can not satisfy these constraints, period!
- Soft Constraints (preferences): An example would be like saying if someone preferred red to green or if there existed weights on constraints (constraint 1: w1, constraint 2: w2). We may not be able to satisfy all of these constraints, but we try our best to.

Satisfiability (SAT)

Variables: X1, … , Xn
Values: 1, 0 (True, False)
Constraints: [X1 $\vee$ $\neg$X2 $\vee$ X4], [X2 $\vee$ $\neg$X3 $\vee$ X5]

For reference:

"$\vee$" = or

"¬" = not

"^" = and

Looking back above at the constraints, each bracket contains a clause

3-SAT is NP-complete: 3-SAT refers to a problem where every clause has three variables.

- Complete refers to the "hardest" problem in a given complexity class. If you can provide a solution, you can solve all problems in the class.
- SAT is prototypical for NP

CSP as Search

We need the following for a general search problem

- Initial State
- Final State: Final state test (if we do not know the final state)
- Actions/successor function

State: Partial variable assignment

Initial State: Empty variable assignment - { }

Example of Complete States: Given X, Y, Z

State {X = True, Z = False}

State {X = True, = False, Z = False}

If we create a tree with variables *x, y, z* & D being 0 or 1, then we get this analysis:

Depth (d) = number of variables

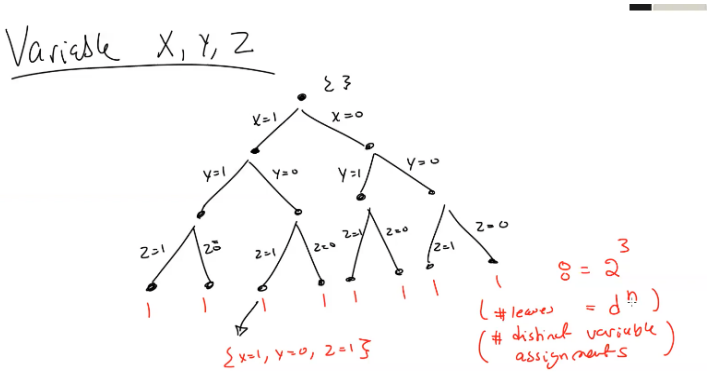
Distinct States = d^n

Leaves = n! * d^n

This is bad since we have many repeated states!

In this case: the number of leave is d^n and the number of leaves now corresponds to this (as show in the picture down below). The depth is still the number of variables.

For the below picture, we should use DFS b/c we will not miss a solution if one exists. There is no notion of optimality (if the depth is finite) b/c we just need to find a solution satisfying constraints.



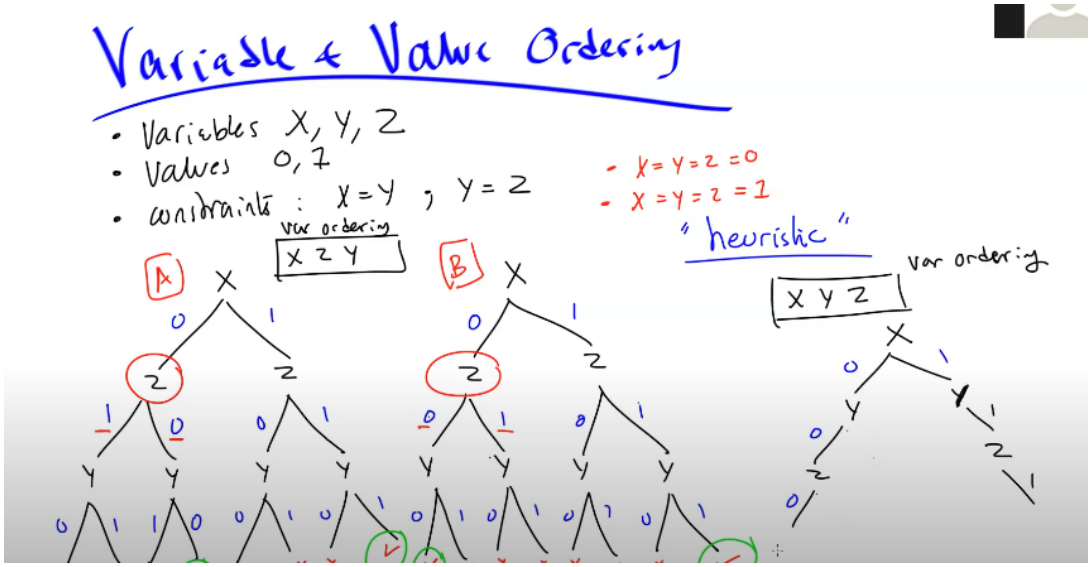
Improvements of DFS for CSP

1. Backtrack Search (DFS + generating feasible children)
2. Value Ordering
3. Variable Ordering
4. Forward Checking
5. Arc Consistency

Forward Checking and Arc Consistency do more sophisticated checks of violation

- Forward Checking will see violations that Backtrack Search can NOT see immediately
- Arc Consistency will see even more violations that even Forward Checking can NOT see immediately

Variable Ordering



This changes the branch count/affects tree size b/c we aren't showing nodes violating constraints.

Most Constrained Variable: Chose the variable with the fewest legal values (this is a type of "heuristic" for variable ordering)

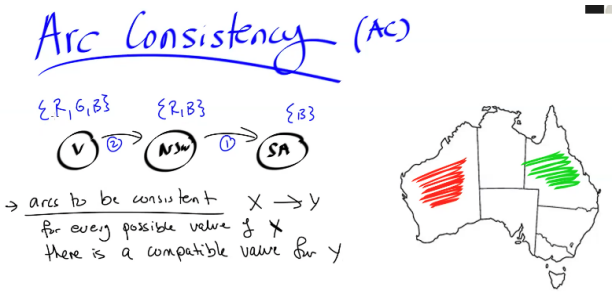
Value Ordering Heuristic

Least Constraining Variable: Chose the value that rules out the fewest values of remaining variables.

Forward Checking (FC)

-> Maintain set of possible values for each variable
-> When you assign a value to a variable, update possible values for other variable
-> Declare "bad state" if some variable loses all its values

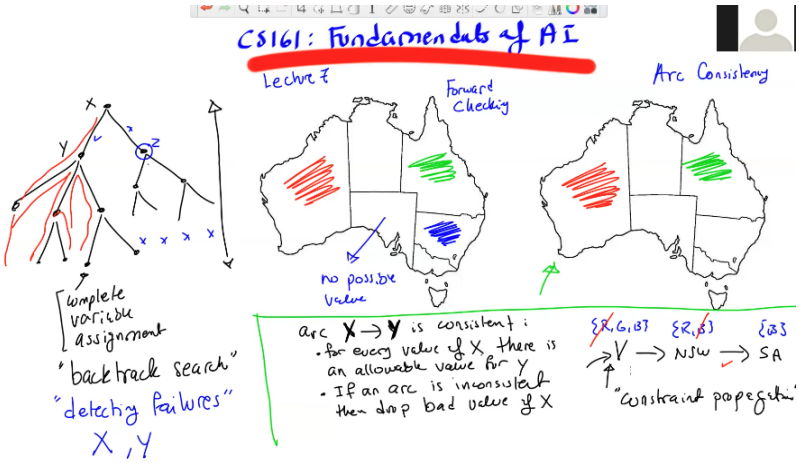
Arc Consistency (AC)



In the picture above, arc 1 is not consistent, so we must remove B on top of the NSW node.

Arc **X** -> **Y** is consistent:

- For every value of X, there is an allowable value for Y.
- If an arc is inconsistent, then drop the bad value of X ("constraint propagation")



*Complexity of enforcing a single-arc consistency (each variable has at most d values): **O(d^2)**
*Binary CSPs (n variables): **O(n^2)** because there are n^2 constraints or arcs since we can choose two variables.

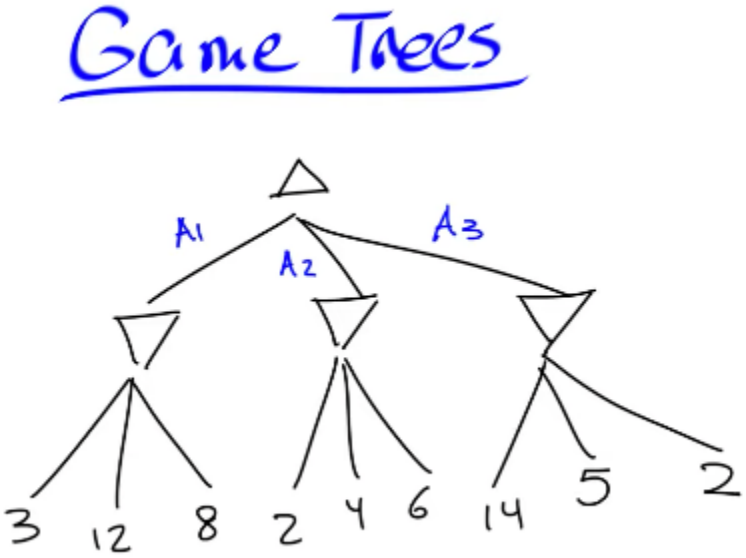
*Complexity of Arc Consistency: O(n^2 d^2 d) => **O(n^2 d^3)**

- n^2 is how many arcs we have and d^2 is from how much it takes to check an arc (essentially from the two previous complexities up above, respectively)
- If Y loses a value, then we have to check an edge and Y can lose at most d values, therefore we will not have to check it more than d times.

Lecture 7: Two-Player Games

	Deterministic	Chance
Perfect Information	Chess, Go, Othello	Backgammon, Monopoly
Imperfect Information	…	Poker, Bridge

Minimax Algorithm



Terminology

A "Ply" is one player's move

A "Move" is when both players move

Game Tree has "utilities" which are the numbers.

The right side up triangle acts as the "max" node.

The upside down triangles act as the "min" nodes.

In this tree, the upside down triangles (min nodes) will have utilities of 3, 2, and 3, respectively from left to right.

In this tree, the right side up triangle (max node) will have a utility of 3.

The Algorithm we use is DFS

Time Complexity is **O(b^m)** where b is the branching factor and m is the depth of tree

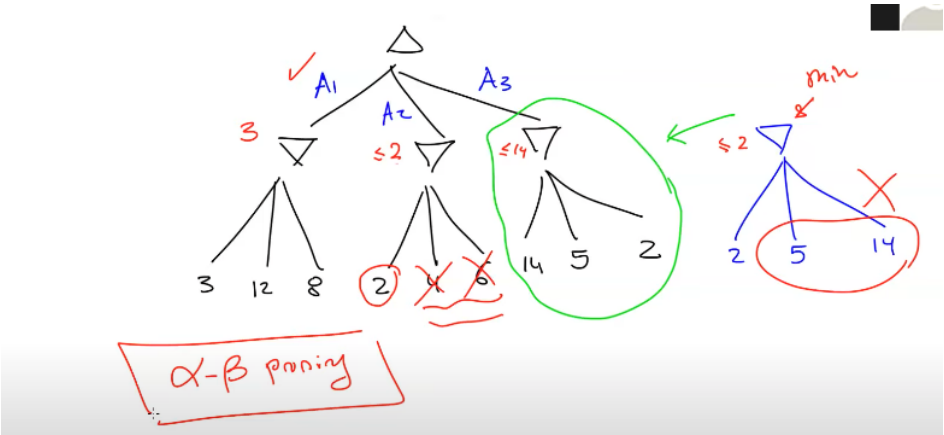
Space Complexity is **O(bm)** where b is the branching factor and m is the depth of tree

Going all the way down to the terminal nodes using minimax will not be efficient for a game like chess. It will take too long due to the fact that chess has a branching factor at around 35.

So, we can use an **evaluation function (fn(n)** that estimates the utility.

Alpha-Beta Pruning

- With Alpha-Beta pruning, you can come to the same conclusion as the Minimax algorithm without having to looking at everything

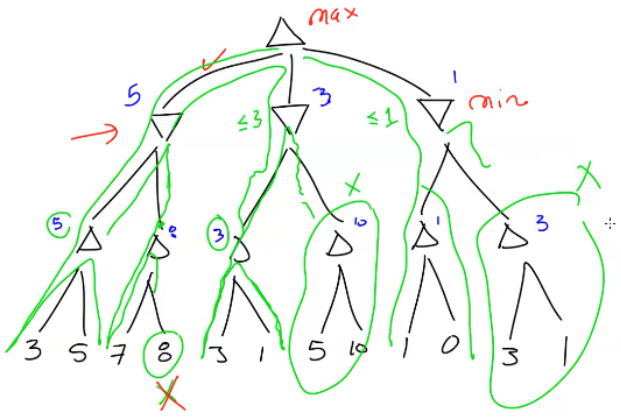


- For min, you are exploring things from the smallest to highest score
- For max, you are exploring things from the highest value children to the lowest value children
- Uses DFS (Depth-First Search)

<u>Analysis of Alpha-Beta Pruning</u>

Considering these two factors: Depth **d** and Branching factor **b**

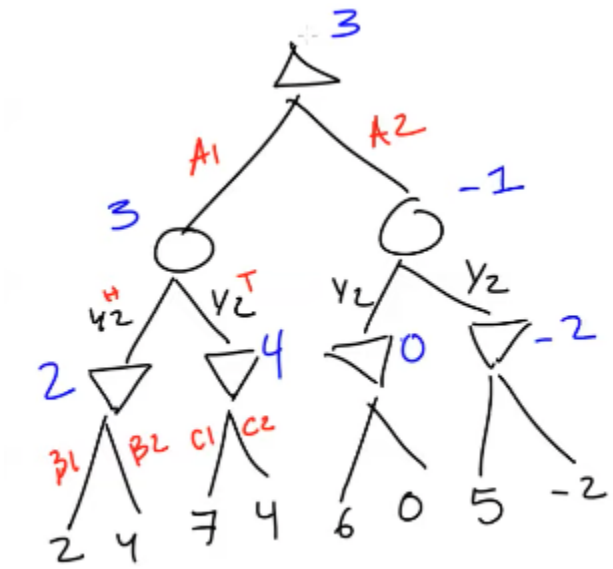
- Standard Minimax: **O(b^d)**
- Best Case Scenario for Alpha-Beta Pruning: **(O(b^(d/2))**



Other than minimax or alpha-beta pruning, we can also use machine learning and train the network on itself in order to produce very powerful models.

Games with chance

When we have games with chance (coin flips, etc), we want to take the weighted average to calculate minimax



For example, to calculate the first node on the left under the edge A1, we would take 2(½) + 4(½) = 3.

Lecture 8: Propositional Logic

Knowledge Representation & Reasoning

Statements in Logic (Knowledge base) -> Logical Deduction (Observation) -> Conclusion

The two representations when AI first started as a field were:

1. Symbolic Representations
2. Numerical Representations

Numerical Representation is now used in probability theory, specifically with Bayesian Networks.

After, there was a decision between modeling knowledge and learning knowledge from information in the world. The former is used in AI now while the latter is used in ML.

Logic (Specifically Propositional Logic or Boolean Logic)

There are two parts:

1. Syntax (Grammar)
2. Semantics (Meaning): Example - Alpha is equivalent to Beta or Alpha implies Beta (the later is deduction)

Syntax

There are two parts:

1. Basic Syntax (a simple story)
2. Normal Forms

Ingredients to Syntax:

1. Variables: x1, x2, … xn where each xi is true (1) or false (0)
2. Boolean Variables: aka atoms, atomic variables, propositional symbols
3. Logical Connective:
    a. $\wedge$: and
    b. $\vee$: or
    c. $\neg$: not
    d. $\Rightarrow$: imply or implies
    e. $\Leftrightarrow$: equivalent or iff
4. Variables is a sentences
    a. If S is a sentence, then $\neg$S is also a sentence

   b. If S1 and S2 are sentences, then S1 ∧ S2, S1 ∨ S2, S1 ⇒ S2, and S1 ⇐⇒ S2 are all sentences, too!
5. Terminology
   a. S1 ∧ S2 is a **conjunction** and S1 and S2 are both **conjuncts**
   b. S1 ∨ S2 is a **disjunction** and S1 and S2 are both **disjuncts**
   c. S1 ⇒ S2 is a **premise** whereas S1 is the **premise antecedent** and S2 is the **consequent**

If we have α => β, then the **contrapositive** would be ¬β => ¬α.

Logical Rules:

α => β = ¬α ∨ β

¬ (α ∧ β) = ¬α ∨ ¬β

¬ (α ∨ β) = ¬α ∧ ¬β

The latter two rules are known as DeMorgan's Law

If we have: ¬(B1 ∧ B2 ∧ B3 ∧ B4) => ¬P

This equals ¬B1 ∨ ¬B2 ∨ ¬B3 ∨ ¬B4 => ¬P

Normal Forms

1. **Conjunctive Normal Form (CNF)**: A conjunction of clauses (<u>universal</u>)
   (A ∨ ¬B) ∧ (B ∨ ¬C ∨ ¬D) ∧ …
   A disjunction of literals is a **clause**.
   A **positive literal** is just a variable (x) and a **negative literal** is its negation (¬x).

2. **Disjunctive Normal Form (DNF)**: A disjunction of terms (<u>universal</u> and <u>tractable</u>)
   (A ∨ ¬B) ∧ (B ∨ ¬C ∨ ¬D) ∧ …
   A conjunction of literals is a **term**.

3. **Horn Form**: a conjunction of horn clauses (<u>tractable</u>)
   Horn Clause (special type of clause): clause with at most one positive literal
   (A ∨ ¬B ∨ ¬C) satisfies this condition.
   (A ∨ B ∨ ¬C) does NOT satisfy this condition.
   Doing SAT on Horn Clauses can also be done easily in linear time => it is tractable.

4. **Negation Normal Form (NNF)**: Negation (¬) only appears next to vars, not sentences.
   Example: ((¬X ∨ Y) ∨ (X ∧ Z)) ∧ …

They are often expressed as circuits. They are tractable iff all entries to all AND gates share no variables.

This subset of NNFs are called **Decomposable Negation Normal Form Circuits (DNNF Circuits)** where SAT can be done in linear time.

Universal

- In general, a form is universal if we are able to express anything we know according to that form.
- CNF and DNF are universal.

Tractable

- In general, a form is tractable if some tasks that are generally hard to do on arbitrary propositional sentences become easy if our knowledge is expressed in this form.
- Tractable means SAT is easy.
- Horn Clauses and Negation Normal Form are tractable.

Summary of Normal Form

	Universal	Tractable
CNF	✓	
DNF	✓	✓
Horn		✓
NNF	✓	
DNNF Circuits	✓	✓

Semantics (meaning)

Semantics are concerned with the notion of worlds.

–> Sentence α holds in some world w.

w |= α (equivalent to "α is true at w.")

M(α) = set of worlds where α holds at: {w: w |= α}

M(α): Meaning of α

W: **Models** of α

- α equivalent to β
  => $M(\alpha) = M(\beta)$
- α is contradictory/inconsistent
  => $M(\alpha)$ = empty set/α has no possibilities
- α is a **tautology**/is valid
  => $M(\alpha)$ = all worlds
- α and β are “mutually exclusive”
  => $M(\alpha) \cap M(\beta)$ = empty set
- α implies β (or $\alpha \Rightarrow \beta$)
  => $M(\alpha) \subseteq M(\beta)$

Lecture 9: Propositional Logic Inference I

Inference

Knowledge Base $\Delta = \{\beta 1, \beta 2, \ldots, \beta n\}$: $\beta 1 \wedge \beta 2 \wedge \ldots \wedge \beta n$

Query α

Does Δ imply α (or $\Delta \models \alpha$)?

$\Delta \models \alpha$ can be thought of as Δ implies α, Δ entails α, or α follows from Δ, …

Inference Methods:

1. **Truth-Table (Model Enumeration)**: We can check valid equivalence through this method.

   It is essentially the "brute-force" method.

   Δ implies α: $M(\Delta) \subseteq M(\beta)$.
2. **Inference Rules**: We can keep applying inference rules until we get what we want.

   $\alpha, \alpha \Rightarrow \beta$

   β

   - This rule essentially says that if we have "α" and "$\alpha \Rightarrow \beta$" in our knowledge base, then we can conclude β.
   - Resolution is the rule we will be focusing on.
3. **By Reduction SAT (Satisfiability) - Search**: For SAT
4. **Converting KB (Knowledge Base) into "tractable forms"**: For advance queries

Method 1: Truth Tables, Enumerating Mode

If we have the following:

Knowledge Base (KB) Δ: $A, A \vee B \Rightarrow C$

Query α: C

Does $\Delta \models \alpha$? This is true iff $M(\Delta) \subseteq M(\alpha)$.

Truth Table for the example above:

This is show down below

=> (at the beginning

of the next page)

	A	B	C	A	A ∨ B => C	Δ	α
1	t	t	t	✓	✓	✓	✓
2	t	t	f	✓	x	x	x
3	t	f	t	✓	✓	✓	✓
4	t	f	f	✓	x	x	x
5	f	t	t	x	✓	x	✓
6	f	t	f	x	x	x	x
7	f	f	t	x	✓	x	✓
8	f	f	f	x	✓	x	x

✓ = true
X = false

M(Δ): 1, 3
M(α): 1, 3, 5, 7

Because M(Δ) ⊆ M(α), then Δ |= α (Δ does, indeed, imply alpha).

We should recall/remember that:

Δ implies α: M(Δ) ⊆ M(α)
Δ1 equivalent to Δ2: M(Δ1) = M(Δ2)
…
M(α ∧ β) = M(α) ∩ M(β)
M(α ∨ β) = M(α) U M(β)
M(¬α) = complement of M(α)

In general, with all of these things, we are trying to map a variable to its models.

Refutation Theorem

This is a very general way to establish Δ |= α (Δ implies α).

Essentially, we want to show that Δ ∧ ¬α is contradictory or more specifically, NOT satisfiable (this is where M(Δ ∧ ¬α) = empty. Here, we are reducing a logical implication to SAT solving.

Method 2: Inference Rules

Inference Rules are of the following form:

Pattern 1, Pattern 2
Pattern 3

1. $\frac{\alpha,\ \alpha \Rightarrow \beta}{\beta}$ (This is called Modus Ponens, which is NOT a complete inference rule).

2. OR-Introduction: $\frac{\alpha,\ \beta}{\alpha \vee \beta}$

3. And-Introduction: $\frac{\alpha,\ \beta}{\alpha \wedge \beta}$

Assume a set of Inference rules R:

$\Delta \vdash_R$ means α is derived from Δ using inference rules R
Inference Rules "R" are **complete**: if $\Delta \models \alpha$, then $\Delta \vdash_R$.

Example:

- $\Delta = A \vee B$
- $\alpha = A \vee B \vee C$

Then, $\Delta \models \alpha$. BUT, we can NOT use Modus Ponens for this and therefore, Modus Ponens is **incomplete**.

Inference Rules are **sound**: if $\Delta \vdash_R$, then $\Delta \models \alpha$.

Questions:

$\Delta \vdash_R$: In general, this is a mechanical way for establishing $\Delta \models \alpha$.
$\alpha \models \beta$: This is a relation.
$\alpha \Rightarrow \beta$: This is a sentence.

Resolution

Example: $\frac{\alpha \vee \beta,\ \neg\beta \vee \gamma}{\alpha \vee \gamma}$: we can "cross out/cancel" β and $\neg\beta$

- Resolution is typically applied to CNF, but can be applied towards other things.
- We resolved $\alpha \vee \beta$ with $\neg\beta \vee \gamma$ over variable β. The resolvent was $\alpha \vee \gamma$.

- Resolution is NOT complete! But resolution is refutation complete if applied to a CNF.
- In general, we will now only apply resolution to CNF and if some logical form is not in CNF, then we will convert it to CNF.

Refutation Complete: It will derive a contradiction is CNF is unsatisfiable or if the CNF has a contradiction.

Unit Resolution

Unit Resolution is a specialization of resolution and when you resolve two clauses and one of them is unit.

Example:

$A \vee \neg B \vee C \rightarrow$ length 3 (literals)
$A \vee \neg B \rightarrow$ length 2 (binary clause)
$\neg B \rightarrow$ length 1 (unit clause)

$A, \neg A \vee B \vee C$:
$B \vee C$

Right up above, we cross out "A and ¬A" to "resolve" this. This is unti resolution because one of them (A) is a unit clause.

In general, resolution when run on CNF can take exponential time.

However, unit resolution can be applied in linear time. The catch is, however, that unti resolution is NOT refutation complete (while general resolution is).

Full Resolution Example

Δ consists of these three things:

- $A \vee \neg B \Rightarrow C$
- $C \Rightarrow D \vee \neg E$
- $E \vee D$

α consists of this one thing:

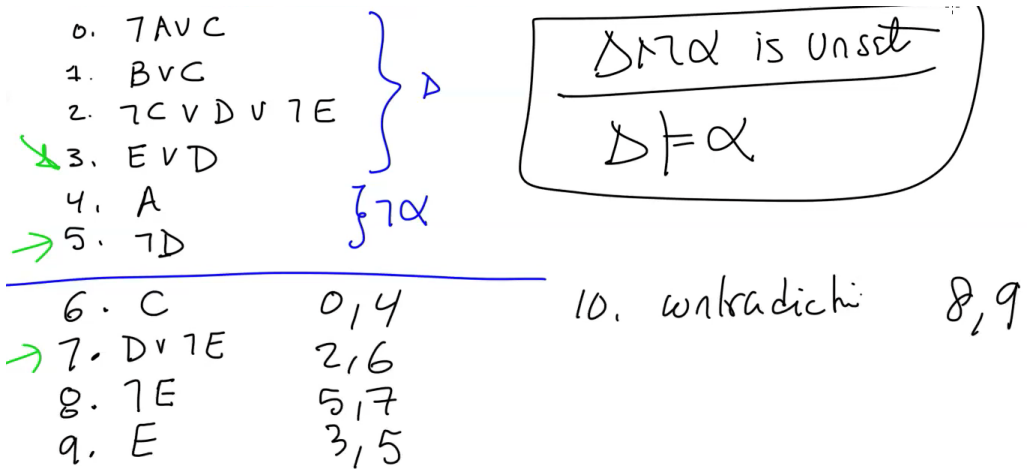
- $A \Rightarrow D$

Is $\Delta \models \alpha$?

The first thing we want to do is check if $\Delta \wedge \neg\alpha$ is unsatisfiable.

We want to primarily take Δ and the negation of α and convert it to CNF.

If we get a contradiction, then Δ |= α. Otherwise, if there is no contraction, then Δ does not imply α.



Since we have a contradiction here, we now have proved that Δ |= α! This is the **resolution proof** or the **resolution derivation**.

Converting Sentences into CNF

Any sentence can be converted into CNF through the following process:

1. Get rid of all connectives but for $\vee \ \wedge \ \neg$
   $\alpha \Rightarrow \beta \rightarrow \neg\alpha \vee \beta$
   $\alpha \Leftrightarrow \beta \rightarrow (\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$
2. Use DeMorgan's law to push negations inside
   $\neg(\alpha \wedge \beta) \rightarrow \neg\alpha \vee \neg\beta$
   $\neg(\alpha \vee \beta) \rightarrow \neg\alpha \wedge \neg\beta$
3. Distribute $\vee$ over $\wedge$ (Distribute OR over AND)
   $(\alpha \wedge \beta) \vee \gamma \rightarrow (\alpha \wedge \gamma) \wedge (\beta \vee \gamma)$

Lecture 10: Propositional Logic Inference II

Review of Inference Methods

1. Truth-Table (Enumerating Models)
2. Inference: Resolution … Converting to CNF
3. By Reduction to SAT
    a. Systematic Search (Backtrack Search)
    b. Local Search (best effort methods - not guaranteed to find a solution, but can be very fast. They apply more broadly to CSPs.)
4. Tractable Circuits (aka Knowledge Compilation) - #SAT (Sharp-SAT)

We will be covering 3 and 4 today: Reduction to SAT and Tractable Circuits

Method 3: Reduction to SAT

$\Delta \models \alpha$ iff $\Delta \wedge \alpha$ is unsatisfiable.

a. Complete Method: Systematic Search (DFS)
b. Incomplete Method: Local Search

Consider the sentence $\Delta = (A \vee B \vee \neg C) \wedge (\neg A \vee C) \wedge (A \wedge C \wedge \neg D)$ We treat each clause as a constraint, and solve it as a CSP.

→ if $w = \{A = T, B = F, C = T, D = F\}$, then this comes out as T and therefore, $w \models \alpha$.

Δ is valid if $M(\Delta)$ = the set of all worlds.

More specifically in terms of SAT, Δ is valid if $\neg\Delta$ is unsatisfiable (this is the query).

Δ and α are mutually exclusive if: $\neg\Delta \wedge \alpha$ is unsatisfiable.

Complete Methods

DFS, Backtracking Search
→ Value/Variable Ordering
→ Detecting Failures Early (unit resolution)

In the context of SAT, backtrack search/DFS is called the **DPLL Algorithm**.

Unit Resolution is used for detecting failures early.

If we get a contradiction, then there is no solution and we do not need to search any further. Unit Resolutions can be done in linear time in the context of DPLL.

General Resolution is refutation complete which means that if there is a contradiction, it is guaranteed to discover it. Even if it runs and there is no contradiction, it is still guaranteed that there is no contradiction.

On the other hand, unit resolution is NOT refutation complete. This means that if you did not find a contradiction at a certain point (when going down the tree), but that does not mean that the input formula is consistent. It may be that there is a contradiction, but that unit resolution can NOT find it.

- You can add clauses to the input formula that are implied in order to improve unit resolution. You can build "Learning Clauses."

Local Search (Incomplete Method)

General Remarks: Even though this is faster and uses little space (has both good time and space complexity), it can not prove if something is unsatisfiable and it is also incomplete. The method is as follows:

1. Guess some truth assignment
2. If the world in which you guessed the truth assignments turns out to be the solution and makes the CNF satisfiable, then we are done. Otherwise, we try again.

   One problem is that it does NOT have memory, so it may not know if it has already visited everything. If you start in a good place, then you will solve it faster, but on the other hand, if you start in a bad place, it will take a very long time to solve.

If we have n neighbors and we violate a certain number of constraints, then we want to make sure that the next state we choose violates the least number of constraints.

This is called "min-conflicts".

The algorithm is called "hill-climbing".

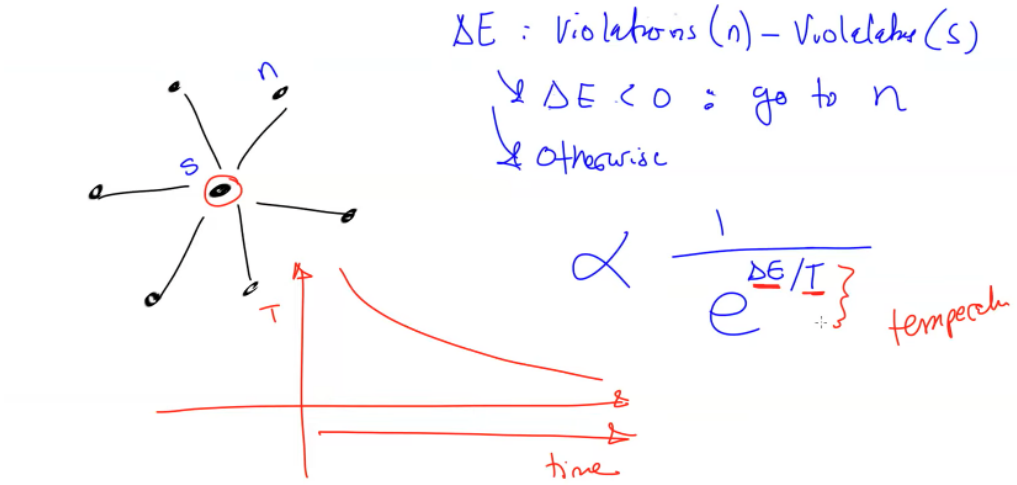
Hill-Climbing (Local Search)

There's an overall difficulty in how we choose our truth assignment and we can do this by imagining the assignments in a graph.

Hill-climbing chooses the next node by heuristic. For local maxima and minima (or local extrema in general), we can utilize random restarts to solve this issue. Hill-Climbing is like a loop that may not visit every world, so we can solve this potential issue in one of two different ways:

1. must allow it to loop forever
   OR
2. terminate after some time.

Simulated Annealing



As ΔE grows, then the whole denominator grows and therefore, the whole probability decreases.

- T, on the other hand, would decrease the overall probability since T increases with time.
- T stands for **temperature**.
- The probability is proportional to 1/(e^(ΔE/T))

The algorithm for simulated annealing is essentially as follows and is <u>probabilistic</u> in nature:

```
While not at goal:
        pick a neighbor randomly
        if it is better
                go there
        else:
                there is a % chance based on how much worse it is and the depth of the
        neighbor
End
```

Method 4: Tractable Circuits and Knowledge Compilation

We are going to take our input formula which is a CNF and "compile it" into a "<u>tractable circuit</u>"

With this, we can do more sophisticated queries than SAT, aka #SAT. These theories are useful for probabilistic methods.

- The #SAT problem is counting the number of satisfiable assignments. #SAT is also known as model counting because it symbolizes probabilistic reasoning.

Complexity:

SAT is NP complete: it is not solvable in polynomial time

#SAT is #P complete: it involves counting the # of acceptable NP solutions

Maj-SAT is PP complete: probabilistically run some number of times for polynomial time

Tractable NNF Circuits

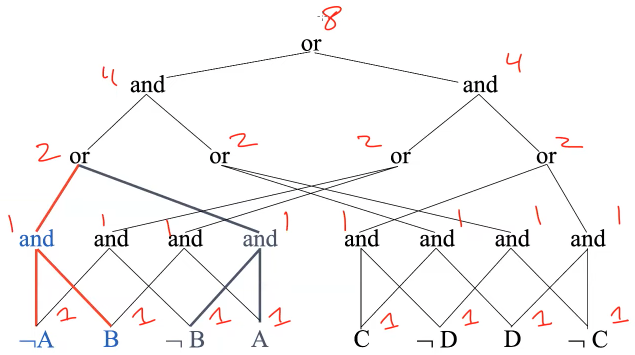
This allows #SAT to be done in linear time. We know it is a circuit because there are AND gates and OR gates and NNF stands for negation normal form.

There are three properties for this circuit: if these are all satisfied, we can do #SAT in linear time.

1. **Decomposability (DNNF Circuits)**: If you look at the children of any "AND" gate in the circuit, they can NOT share variables.
2. **Determinism**: If you look at the children of any "OR" gate in the circuit (let's call them α, β if there are two children), then α ∧ β must be unsatisfiable (they conflict - aka, you can cancel everything out). Essentially, the inputs to the OR gate must be mutually exclusive.

    d-DNNF Circuits: have both decomposability and deterministic properties
3. **Smoothness**: If you look at the children of any "OR" gate in the circuit (let's call them α, β if there are two children), then they must share NO variables.

The total number of variable assignments where a variable can only be true or false (binary) would be 2^n where n represents the number of variables.



The picture up above shows how we can solve the #SAT problem in linear time. Here, we assume that the variables at the bottom take true (1) as input and none of them take false (0).

At AND gates, we multiply the children/inputs and at OR gates, we add children/inputs.

At the top, we get "8" which signifies the number of SAT assignments and is the answer to the #SAT problem. Even though we have 16 possible total variable assignments, only 8 of them satisfy what we are looking for.

Lecture 11: First-Order Logic

FOL: First Order logic (aka PC or Predicate Calculus)

In general, first order logic is more expressive and succinct than propositional logic.

If we use the Wumpus World as an example:

First Order Logic would look like this: $\forall$ r Pit(r) $\Longrightarrow$ [$\forall$ s Adjacent(r, s) $\Longrightarrow$ Breezy(s)]

This means that for all squares r, if there is a Pit in that cell r, that implies for all other cells that are adjacent to r, those cells must be breezy.

In terms of representing a world:

Propositional Logic does something like this:

Variables: {A, B, …}

Values: {T, F}

First Order Logic:

Objects: People, Houses, Wumpus, Numbers, Colors, …

Properties: Breezy, Large, Red, …

Relations: Inside, Adjacent, Larger, …

Functions: Father of, Best Friend, …

Example:

*One plus one equals two

Objects: one, two

Properties: –

Relations: equals

Functions: plus

*Squares adjacent to the wumpus are smelly.

Objects: Squares, wumpus

Properties: smelly

Relations: adjacent

Functions: –

Syntax

1. Constants: 2, Jack, UCLA, … (direct names of objects)
2. Predicates: adjacent
3. Functions: Left of, …

Numbers 1, 2, and 3 are all domain specific and there could be multiple choices ("vocabulary").

4. Variables: x, y, z (lower case)
5. Connectives: $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftarrow\Rightarrow$
6. Equality: =
7. Quantifiers: $\forall$ (For all), $\exists$ (There exists)

Atomic Sentences

They are the simplest sentences you can write in first order logic

Term: constant or variable or function (term_1, … term_n)

It is an expression that you can use to refer to an object.

Atomic Sentence: Example: FatherOf(Jack) is a function
Brother(Jack, Tom) also a function
{Predicate(term_1, …, term_n)}

Function vs Relation:

Function: maps object to object

Relation: holds between objects

Universal Quantification

$\forall$ <variables> <sentence>

$\forall$ At(x, UCLA) => Smart(x) → equivalent to the conjunction of all instantiations of P ($\forall$x P). From this, we could say:

[At(John, UCLA) $\Rightarrow$ Smart(John)] $\wedge$
[At(FatherOf(John), UCLA) $\Rightarrow$ smart(fatherOf(John))] $\wedge$
… (This is conjunctions with the "ANDs" ($\wedge$)
Predicates: At (Relation), Smart (Property)
Constant: UCLA

Existential Quantification

$\exists$ <variables> <sentence>

$\exists$x At(x, UCLA) $\wedge$ Tall(x) → equivalent to the conjunction of all instantiations of P ($\forall$x P). From this, we could say:

[At(Ed, UCLA) $\wedge$ Tall(Ed)] $\vee$
[At(Sandy, UCLA) $\wedge$ Tall(Sandy)] $\vee$

… (This is disjunction with the "OR" ($\vee$)

In general, for quantification, this is the typical use case:

- Universal quantification: $\forall$ At(x, UCLA) => Smart(x)
  Uses "=>"
- Existential quantification: At(Ed, UCLA) $\wedge$ Tall(Ed)
  Uses "$\wedge$"

Properties of Quantifiers

- $\exists x\ \exists y = \exists y\ \exists x$
- $\forall x\ \forall y = \forall y\ \forall x$
- $\exists x\ \forall y$!= $\forall y\ \exists x$

$\forall y\ \exists x$ Loves(x, y): For all y, there exists x such that x loves y:
Everyone in the world is loved by at least one person.

$\exists x\ \forall y$ Loves(x, y): For all x, there exists y such that x loves y:
There is a person who loves everyone in the world.

---

*$\forall x$ likes(x, Ice Cream): For all x, x loves Ice Cream OR (Everybody likes ice cream.)
The equivalent saying using "$\exists$" would be: $\neg \exists x \neg$likes(x, Ice Cream)
DeMorgan's Law: $\forall x \neg\neg$likes(x, Ice Cream) = $\forall x$ likes(x, Ice Cream)

Equality

Spot has two sisters: $\exists x\ \exists y$ sister(x, spot) $\wedge$ sister(y, spot) $\wedge \neg$ (x = y)
The above statement is essentially saying that Spot has 2 or more sisters, BUT if we wanted to say that Spot has ONLY two sisters, then we would say the following:

- $\exists x\ \exists y$ sister(x, spot) $\wedge$ sister(y, spot) $\wedge \neg$(x = y) $\wedge$ [$\forall z$ sister(z, spot) $\Rightarrow$ (z = x $\vee$ z = y)]

Uniqueness Quantifier

$\exists !$ = There exists a unique

- $\exists !$ x king(x) = There is exactly one king
- [$\exists !$ x king(x) ^ [$\forall$x king(y)) $\Rightarrow$ (y = x)]]
  In general, we want to avoid having free variables, which are variables with no Quantifiers.

We also want a well-formed formula which is an expression with no free variables.

Lecture 12: First-Order Logic Inference

Topics for this lecture:

**Reducing FOL inference to Propositional Inference*

**Simple/Restricted Method:*

- *Forward Chaining*
- *Backward Chaining*

**Resolution*

Universal Instantiation (UI)

Taking a universally quantified sentence and replacing it with a propositional sentence(s).

Rule: ($\forall$ variable) (sentence) subst(variable/ground-term, sentence) for constants {John, Richard} in the below case

$\forall$ x King(x) $\wedge$ Greedy(x) $\Rightarrow$ Evil(x)
$\rightarrow$ King(John) $\wedge$ Greedy(John) $\Rightarrow$ Evil(John)
$\rightarrow$ King(Richard) $\wedge$ Greedy(Richard) $\Rightarrow$ Evil(Richard)
$\rightarrow$ King(Father(Richard)) $\wedge$ Greedy(Father(Richard)) $\Rightarrow$ Evil(Father(Richard))

In general, a term is a variable, constant, or function(term_1, … , term_n).

Ground term: It does not have variables, so constants are ground terms but not variables. Each individual term_i in function is also a ground term.

Existential Quantification

Rule: ($\exists$ variable) (sentence) subst(variable/new-constant, sentence) for constants {C}
$\exists$ x Crown(x) $\wedge$ On-Head(x, John)
$\rightarrow$ Crown(C1) $\wedge$ On-Head(C1, John)

C1 is the "skolem constant"

We start with a FOL KB (First-Order Logic Knowledge Base) and finish with a PL (Propositional Logic Knowledge base).

Satisfiability is preserved.

Entailment in FOL is semidecidable

If a sentence is entailed by a first order logic knowledge base, then it is entailed by a finite subset of that propositional knowledge base.

Here, we bring in the idea of nesting: where we can bring in infinite instances, however the KB (KNowledge Base would be too large)

- John, Father(John), Father(Father(John))

Algorithm:

For n = 0 to ∞, do:

- Create a proposition KB by instantiation with depth-n terms.
- See if the stance is entailed by this KB.

However, an issue that comes up is that we need to make sure that it terminates and it only terminates if the sentence is entailed.

This inference is considered **semi-decidable**.

Definite Clauses

Horn Clause: at most one positive literal (0 or 1 positive literal is fine, but no more).
Definite Clause: exactly one positive literal

Forward Chaining

We build this from the ground up, starting with ground literals and eventually getting in our case ⇒ Criminal(West).

Backward Chaining

In this case, we start with Criminal(West) and then do the same thing that we did in forward chaining, but backwards. We try to end up with the ground literals at the end.

- Applications: Used (with improvements) in logic programming such as Prolog.

Resolution in First Order Logic (FOL)

Unification: If we start with *Knows(John, x)*, we want to unify it to become *Knows (John, Jane)* by setting the unifier to {x/Jane}.

- In general, if we have α, β, and θ, we can unify(α, β) = θ. Then αθ = βθ.

More Examples:

Knows(John, x) & Knows (y, OJ), θ = {y/John, x/OJ} ⇒ *Knows (John, OJ)*
Knows(John, x) & Knows (y, Mother(y)), θ = {*y/John, x/Mother(John)*} ⇒ *Knows(John, Mother(John)*

FOL Simple Resolution Examples

∀x Rich(x) ⇒ Unhappy(x)
Rich(Ken)

_____

¬Rich(x) ∨ Unhappy(x),

Rich(Ken)

_____

If we do {x/Ken}:

¬Rich(Ken) ∨ Unhappy(Ken),
Rich(Ken)

_____

Unhappy(Ken)

Just like with Propositional Logic, we need to show that "Delta ^ Not Alpha" is unsatisfiable or that there is a contradiction (aka it "cancels out" in the end).

Conversion to CNF

1. Eliminate ⇒ and ⇐⇒
2. Move Negation Inward: $\neg \forall x a \rightarrow \exists x \neg a$ OR $\neg \exists x a \rightarrow \forall x \neg a$
3. Standardize Variables: Each Quantifier should use a different one
4. "Skolem"-ize: Replace the variable (such as y) with something like F(x) and drop the existential quantifier before it. F(x) is a **skolem function**.
5. Drop Universal Quantifiers
6. Distribute

Lecture 13: Probability & Beliefs

Model, Reasoning, & Learning Under Uncertainty

This is the broader topic that we will start now. Classic Logical is "**monotonic**", but human reasoning is NOT.

If $\Delta \models \alpha$,

Then $\Delta \wedge \beta \models \alpha$

- α continues to follow even when we learn β, which is the new formulation

The problem with monotonicity is shown in the example down below:

If Tweety is a bird (Δ), then we would say it flies (α)

However, if we add that Tweety is a penguin (Δ), then we would say it does NOT fly (α).

In this case, our α changes!

If we were to create the example up above with FOL (First-Order Logic):

Δ: $\forall$ x bird(x) $\Rightarrow$ flies(x)

Good: $\Delta \wedge$ bird(Tweety) $\Rightarrow$ flies(Tweety)

Bad: $\Delta \wedge$ bird(Tweety) $\Rightarrow \neg$flies(Tweety)

This is a contradiction!

Because of the example up above, we should have written:

Δ : $\forall$ x bird(x) $\wedge \neg$abnormal(x) $\Rightarrow$ flies (x)

Good: "$\Delta \wedge$ bird(Tweety) $\wedge \neg$flies(Tweety)" is fine and there is no contradiction, but…

Bad: "$\Delta \wedge$ bird(Tweety)" does NOT imply "$\neg$flies(Tweety)"

This is a contradiction!

Belief Revision

- Degrees of Belief is between [0,1] which are going to each be represented by probabilities

Probability as a basis for representing belief

world	Earthquake	Burglary	Alarm	Pr(.)
ω_1	true	true	true	.0190
ω_2	true	true	false	.0010
ω_3	true	false	true	.0560
ω_4	true	false	false	.0240
ω_5	false	true	true	.1620
ω_6	false	true	false	.0180
ω_7	false	false	true	.0072
ω_8	false	false	false	.7128

Looking at the table above, we can represent the probability of each world happening.

f: sentences → degree of certainty; $Pr(\alpha) = \Sigma\ Pr(w)$ for $w \models \alpha$

- For example: $Pr(E) = .1$, $Pr(B) = .2$, $Pr(\neg B) = .8$, $Pr(A) = .2442$, etc…

Assuming α is a sentence, we will have the following properties:

1. $0 \leq Pr(\alpha) \leq 1$
2. If α is inconsistent (unsatisfiable or contradictory), then $Pr(\alpha) = 0$
3. If α is valid, then $Pr(\alpha) = 1$
4. $Pr(\alpha) + Pr(\neg\alpha) = 1$
5. $Pr(\alpha \vee \beta) = Pr(A) + Pr(\beta) - Pr(\alpha \wedge \beta)$
6. If α, β are mutually exclusive, $Pr(\alpha \vee \beta) = Pr(\alpha) + Pr(\beta)$
   Saying that α, β are mutually exclusive implies that $M(\alpha) \cap M(\beta) = 0$

Belief Change

This is what happens when we get new information (or evidence):

Initial: Pr(.)

Updated β: Pr (.|β) where β is new information

$Pr(w|\beta) = \{0$ if $w \models \neg\beta$

$Pr(w)/Pr(\beta)$ if $w \models \beta\}$

Now, we have to zero out the worlds that are no longer valid given the new information. In this case, from the formula right above, we get **Bayes Conditioning**: $Pr(\alpha|\beta) = Pr(\alpha \wedge \beta) / Pr(\beta.)$

Using an example of $Pr(B) = 0.2$, $Pr(B|E) = 0.2$,$Pr(E) = 0.1$, and $Pr(E|B) = 0.1$:

We can concludes that if $P(B) = Pr(B|E)$ AND $P(E) = P(E|B)$, then these two variables are **independent**.

Independence

Alpha (α) is independent of Beta (β) when **$Pr(\alpha|\beta) = Pr(\alpha)$**.

Bayes' Conditioning told us that $Pr(\alpha|\beta)$ is the same thing as $Pr(\alpha \wedge \beta) / Pr(\beta)$ and we can simplify to $Pr(\alpha \wedge \beta) / Pr(\beta) = Pr(\alpha)$.

- Overall, $Pr(\alpha \wedge \beta) = Pr(\alpha)\ Pr(\beta)$
- We can also see that $Pr(\alpha|\beta) = Pr(\alpha)$ and $Pr(\beta|\alpha) = Pr(\beta)$.

One more observation about independence is that it is dynamic!

Pr finds α conditionally independent of β given γ

$Pr(\alpha|\beta \wedge \gamma) = Pr(\alpha|\beta \wedge \gamma)$
Also: $Pr(\alpha \wedge \beta|\gamma) = Pr(\alpha|\gamma)Pr(\beta|\gamma)$

Properties of Beliefs

- **Chain Rule**: $Pr(\alpha 1 \wedge \alpha 2 \wedge ... \wedge \alpha n) = Pr(\alpha 1|\alpha 2 \wedge \alpha 3 \wedge ... \wedge \alpha n) + \ldots + Pr(\alpha n)$
- **Case Analysis**: $\{\beta 1, \beta 2, \ldots, \beta n\}$ are mutually exclusive (no overlap) and exhaustive (they cover everything). Overall, the $Pr(\alpha) = Pr(\alpha \wedge \beta) + Pr(\alpha \wedge \neg\beta)$.

  More intuitively = **$\sum Pr(\alpha|\beta i)\, Pr(\beta i)$**
- **Bayes' Rule**: $Pr(\alpha|\beta) = (Pr(\alpha) / Pr(\beta)) * Pr(\beta|\alpha)$

For Bayes Rule, we can consider the following:

α: cause (like a disease)

β: effect (like a symptom)

- Bayes Rule is essentially finding the probability of effect(symptom) given cause (disease)

One More Observation:

We have built probability calculus on top of propositional logic

Lecture 14: Bayesian Networks

We use probability as a basis for belief and this will serve as the foundation for Bayesian Networks.

We can think of a Bayesian Network as a modeling tool.

a. Directed Acyclic Graph (DAG): Causality
b. Numbers: Probabilities (These can be learned from data)
   We can use (a) and (b) to create a Probability Distribution (as a table with worlds and their corresponding probabilities).
c. Inference
   Compile Bayesian Network into "circuits"
d. Learning

<u>Notation</u>
Variable - X
Value - x
If X has the values: {r, b, g}, then x could be r.

Pr(x): Pr(X = x) => number
Pr(X) => table: distribution over X

Set of Variables - **X**:
A = {X, Y}

Instantiation - **x**:
a: X = r, Y = s
X = r, Y = m

<u>Variable Independence</u>
If we have the $Pr(\alpha|\beta, \gamma) = Pr(\alpha|\gamma)$, then we know that α & β are independent given γ.
In this case, α, β, and γ are sentences.

Similarly, if we have Variable sets **X**, **Y**, and **Z**, and X and Y are independent given Z:
Then, we can represent this as I(X, Z, Y).
$\Rightarrow Pr(x|y, z) = Pr(x|z)$

Another example: if **X** = {A, B}, **Y** = {C}, and **Z** = {D, E} (and they are all binary variables), then there are 4 states for X, 2 states for Y, and 4 states for Z, so…

I(X, Z, Y) implies that:

A ∧ B independent of C given D ∧ E.

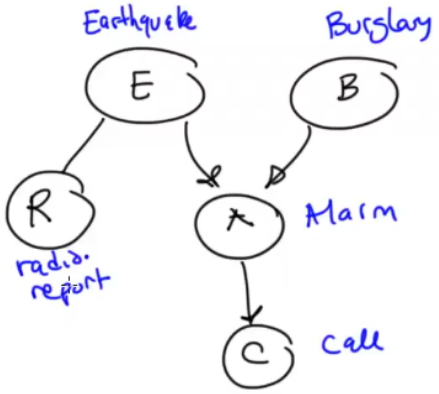
A ∧ ¬B independent of C given D ∧ E.

…

¬A ∧ ¬B independent of C given ¬D ∧ ¬E.

All these make up 32 statements! (4 x 2 x 4 = 32)

Capturing Independence Graphically



A Bayesian Network is a Directed Acyclic Graph (DAG) meaning that each edge is directed and there exists NO cycles.

The Bayesian Network in the case above represents causality. For example, an earthquake could cause an alarm, then our neighbor may call.

If we have I(R, A, C), then R and C are independent given A. Moreover, if we have I(E, Ø, B), then E and B are independent.

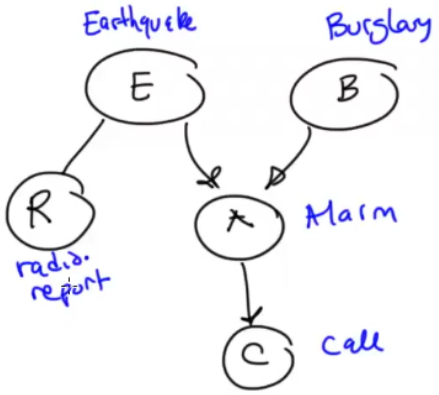
- Parents(V): Self-Explanatory
  - A: E, B
  - R: E
  - C: A

- Descendants(V): All the nodes downstream from V
  - E: R, A, C
  - B: A, C
  - C: Ø

- Non-Descendants(V): The complement of Descendant, but also Exclude V & its parents

- A: E
- E: B

Markovian Assumptions of a Bayesian Network



The Markovian Assumptions are all the independent statements that are implied by a graph of a Bayesian Network and take this form: I(V, Parents(V), Non-Descendants(V)).

I(C, A, BER)
I(R, E, BAC)
I(B, Ø, ER)
I(E, Ø, B)
I(A, EB, R)

These five make up the **Markovian Assumptions**: Markov(G).

Parameterizing the structure

For Variable C, the parents are A.

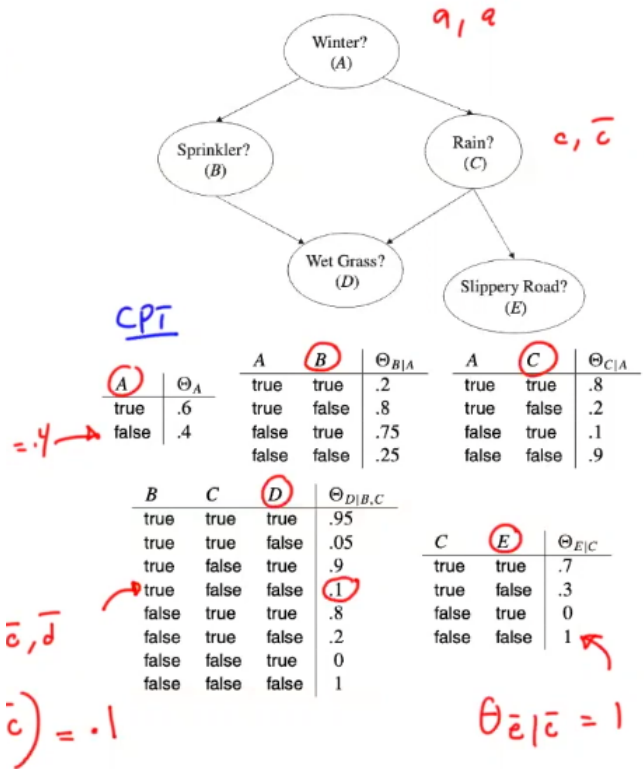
Conditional Probability Table (CPT):

A	C	Pr(c|a)
t	t	.8
t	f	.2
f	t	.001
f	f	.999

.8: Pr(c|a) ⇒ Pr(C = t, A= t): Probability you will receive a call given the alarm has triggered.
.2: Pr(Pr(C = f, A= t): Probability you will receive a call given the alarm hasn't been triggered.

- These make up the distribution on the condition that A is true.

The distribution of a Bayesian Network



A	Θ_A
true	.6
false	.4

A	B	$\Theta_{B\|A}$
true	true	.2
true	false	.8
false	true	.75
false	false	.25

A	C	$\Theta_{C\|A}$
true	true	.8
true	false	.2
false	true	.1
false	false	.9

B	C	D	$\Theta_{D\|B,C}$
true	true	true	.95
true	true	false	.05
true	false	true	.9
true	false	false	.1
false	true	true	.8
false	true	false	.2
false	false	true	0
false	false	false	1

C	E	$\Theta_{E\|C}$
true	true	.7
true	false	.3
false	true	0
false	false	1

The way we fill this out is actually through the chain rule:

Pr(abcde) = Pr(a) * Pr(b|a) * Pr(c|a) * Pr(d|bc) * Pr(e|c) = 0.7

Pr(ab¬cd¬e) = Pr(a) * Pr(b|a) * Pr(¬c|a) * Pr(d|b¬c) * Pr(¬e|c) = 0.216

…

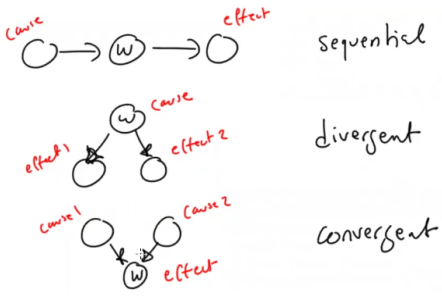
Pr(¬(abcde)) = Pr(¬a) * Pr(¬b|¬a) * Pr(¬c|¬a) * Pr(¬d|¬b¬c) * Pr(¬e|¬c) = 0.09

D-separation

If we are trying to figure out ways/paths between x and y (given that we know z and its states), then we need to check if all paths are blocked.

If any path between x and y is not blocked, then we do not have independence.

There are three different ways in which a node can be connected to its neighbors:



1. **Sequential**: node → w → node – The valve is closed iff w ∈ **Z**.
2. **Divergent**: node ← w → node – The valve is closed iff w ∈ **Z**.
3. **Convergent**: node → w ← node – The valve is closed iff w and Descendants(w) ∉ **Z**.

dsep(X, Z, Y): X and Y are d-separated given Z

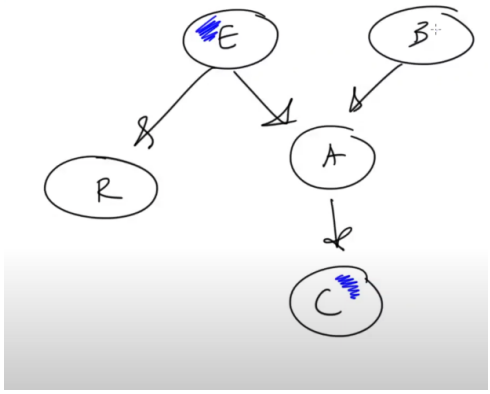
This happens iff every path between a node in X and a node in Y is blocked by Z.

A path is blocked by Z iff at least one value on the path is closed given Z

X → Y (This kind of path is never blocked: this is essentially a base case).

Example:

We are given Z = {E, C}, so this is known/observed!



dsep(B, EC, R)? There is only one path between B and R which goes from B to A to E to R or vice versa.

The first path we have is from R to E to A and this is a divergent path.

The second path we have is from E to R to A and this is a convergent path.

- For this case, we have two paths. If one of them is closed, then we are done and we can say there is d-separation. But, if both of them are open, then there is NOT desperation.
- Since the first path is closed, we can say yes to dsep(B, EC, R)!
- Extra information: the second path is closed because descendants of A (only C) are ∈ **Z**.

Lecture 15A: Bayesian Network Inference

Inference
(1) Queries
(2) Algorithms

Modeling
Sensitivity Analysis

Queries

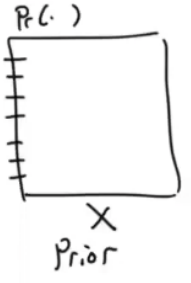
Prior Marginal: For each variable, we are trying to compute the distribution on it. "Marginal" means a distribution over a small set of variables and "prior" means you do NOT have evidence yet and is essentially "before testing."

C = yes	3.2%
C = no	96.8%

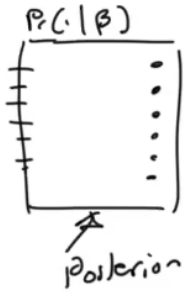
- This table is an example of a prior marginal. It is also called a "monitor" on the SamIam software used by Professor Darwiche's group.

Posterior Marginal: If we are given evidence β which constant of T1 being positive and T2 being negative, then we can have the following: Pr(.|β)

Pr(.):



Pr(.|β):



The goal of inference algorithms is to give answers w/o making joint probability tables.

Most Probable Explanation (MPE)

Based on Lecture Example:
We will assume the evidence A = yes.

We have 5 variables and each one can be in 2 states ⇒ 32 instantiation (since 2^5 = 32). If we are also assuming the evidence A = yes, then the other four variables have 16 instantiations. We want to find the instantiation with the highest probability or the "most probability instantiation" of the four variables given our evidence.

Maximum a Posteriori Hypothesis (MAP)

This is a generalization of Most Probable Explanation (MPE) and MPE will be a special case of this. MPE is also a lot easier to compute than this.

Instead of finding the most likely instantiation of all variable, we (as the user) will pick the subset of the variables and say that we are only interested in these and we want to know the most likely state of instantiation of these variables.

- If all you have is an algorithm for MAP, you can also use this to compute the MPE. In addition, MAP requires the evidence and the set of variables that you need to maximize over while MPE does not.
- If we wanted, we could use the MAP engine and pass it through all engines and to give us an MPE.

Complexity of Inference

- Variable Elimination
- Conditioning

The complexity of these algorithms are tied to the topology of these Bayesian Networks (how connected are these networks). More specifically, the tree width is related to how connected the network/graph is.

If we have the following: **n**: # variables, **d**: # values, **w**: tree width

Then, the complexity for marginals is O(n(d^w)).

A few special cases:

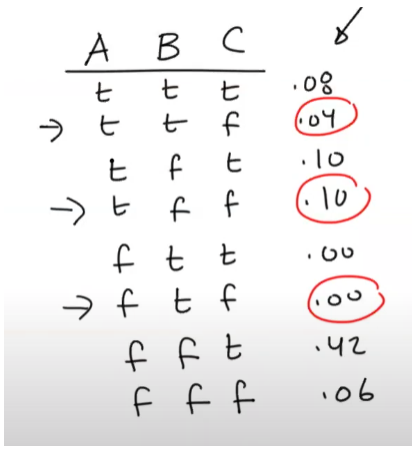
If **w = 1** (which means there is only one path from one node to another node), then we can do inference in linear time.

If we have a **poly-tree (singly-connected network)**, this means we have more than one parent per node. The tree width is **k** (the max number of parents a node can have).

A general DAG is known as **multiply-connected**.

Weighted Model Counting (WMC)

If we have $\Delta = (A \vee B) \wedge \neg C$ described by the following table down below, then we have already talked about how:



SAT is a "yes/no" answer in which we compute whether there exists an instantiation of these variables that satisfy this sentence.

#SAT, more especially, counts how many worlds in which there is an instantiation of these variables that satisfy this sentence (in this case, 3). If a world satisfies a formula, then it's a model of that formula. So, in this case, we have 3 models.

- WMC (Weighted Model Counting) is a generalization of Model Counting (or #SAT) that adds up the probabilities from all the models/worlds that satisfy our query.
- If we can take our formula and compile it (knowledge compilation) into an NNF circuit that is smooth, decomposable, and deterministic, then we can do WMC in linear time!

Another plausible idea is to use probability as our weights: $WMC(\Delta \wedge \alpha) = Pr(\alpha)$.

In general, we would need to change the Δ into a boolean circuit.

We can see that Δ is true in the following worlds in w1, w4, and w7.

We can also say that the weights of literals are $W(A) = W(\neg A) = ... = W(\neg C) = 1$.

Moreover: $W(P\_i) = \theta\_i$, $W(\neg P\_i) = 0$, so $W(A) = W(P\_1) * W(P\_4) * W(P\_7)$.

Analysis of Weighted Model Counting:

Contains many possible representations

Is NOT sensitive to tree width

Is applicable to other application other than Bayesian Networks

Lecture 15B: Bayesian Network Modeling

Analysis of Bayesian Networks

n: # of variables
k: maximum # of parents per node
d: maximum # of values per variable

The size of the Bayesian network = O(n*(d^(k+1)))
The size of the Joint-Probability Table = O(d^n)
⇒ Bayesian Networks have a more efficient (smaller) space complexity.

If we want to model a situation using a Bayesian Network, we can do the following:

Step 1: Define variables, values.
Step 2: Define edges.
Step 3: Specify CPT.
Two types of Variables: Query OR Evidence variables

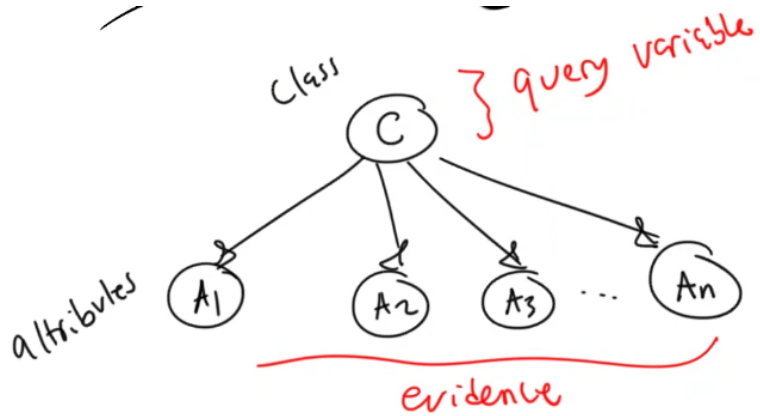
Types of Data:
→ Complete: When data is not missing, learning parameters is easy (closed form: is efficient).
→ Incomplete: When data is missing, we use EM (Expectation-Maximization).

Bayesian Network Structure + CPTs = Bayesian Network

- Bayesian Network Structure + CPT for A = Bayesian Network A
- Bayesian Network Structure + CPT for B = Bayesian Network B

Maximum Likelihood Principle: choosing the network assigning the highest probability of a dataset.

Naive Bayes Structure



Lecture 16A: Learning Bayesian Networks (Unsupervised)

Topics:

Learning can be broken down into two components: Parameters and Structure
Supervised (Query-Oriented) vs Unsupervised (Model-Oriented) Learning

When we have **Complete Data**, the maximum likelihood parameters are unique and it is easier to find them. On the other hand, when having **Incomplete Data**, the maximum likelihood parameters are not unique and the algorithm to find them is more involved.

The distinction matters when we try to learn the parameters for the Bayesian Network.

Using Complete Data

We use the **Empirical Distribution** to get the values in the example down below along with Bayes Conditioning Rule.

For example: $\theta_{\neg S|H} = Pr(\neg S|H) / Pr(H)$ is the maximum likelihood parameter estimate.

Using Incomplete Data

Here, we use EM (Expectation-Maximization) to find the maximum likelihood parameters. This is an iterative (local-search) algorithm that first starts with random estimates and CPTs. It also "completes" the data in a sense and it chooses the probability with higher value.

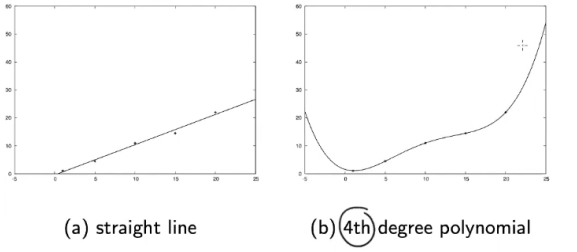
Algorithm:

$CPT_1 \rightarrow BN_1 \rightarrow Pr_1(.)$
$CPT_2 \rightarrow BN_2 \rightarrow Pr_1(.)$
$CPT_3 \rightarrow BN_3 \rightarrow Pr_1(.)$
…
Converges!

The underlying concept behind Expectation-Maximization goes beyond Bayesian Networks.

BUT, why is maximum likelihood not sufficient? Because of **overfitting**.

Overfitting



(a) straight line (b) 4th degree polynomial

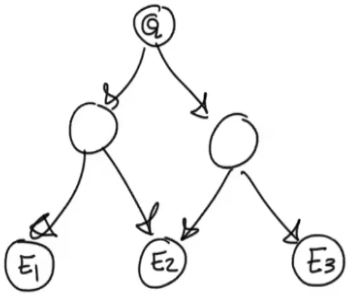
The problem of overfitting: Even though the fit in (b) is perfect, the polynomial does not appear to provide a good generalization of the data beyond the range of the observed data points.

Lecture 16B: Learning Bayesian Networks (Supervised)

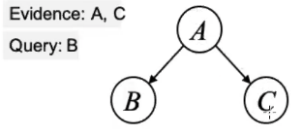
Unsupervised (OR Model-Oriented/Unlabeled) Learning
VS
Supervised (Query-Oriented/Labeled) Learning

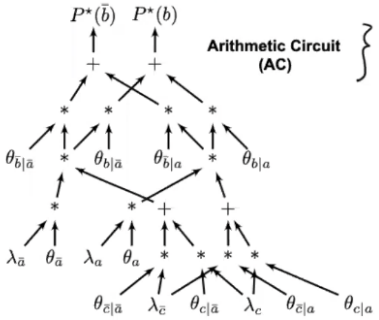
We have mostly done unsupervised or model-oriented learning with unlabeled data, but we will now talk about supervised or query-oriented learning with labeled data.

If we have a graph that looks like this, then we can look for a specific query such as Pr(Q | E_1, E_2, E_3).



Arithmetic Circuits (AC)





If we compile the Bayesian Network + the Query, the result is an Arithmetic Circuit.

P*(b) ← distribution on B

λ is for the inputs

If we take the boolean formula + weights,we can get an NNF circuit and we can convert it to an AC Circuit.

In addition we can evaluate the circuit in linear time with a pass through and with the indicators and pass through.

- The same complexity of a Bayesian Network also applied to an Arithmetic Circuit (AC). It is O(n*(d^w)) where n is the # of variables, d is the # of values, and w is the treewidth.

Labeled Data

If we do not have the parameters, we need to optimize the **loss function**.

This is the optimization function/the criteria that we are trying to optimize.

Cross Entropy takes two distributions and tries to see how close they are to each other.

If we have two distributions (P(X) and Q(X)), then cross entropy would give us back a number that measures how close they are.

For example, if our input/evidence is A, C and we are given A = T and C = T, then this gives us a prediction of Pr(B).

For the dataset, we would get a one-hot distribution that looks like this:

B	(label)
T	0
F	1

Gradient Descent is the main algorithm used that tries to initially guess values of the parameters and it tries to improve on them by computing the derivative of the loss function with respect to the parameters and then walks in the direction of the derivative.

TensorFlow and PyTorch are two tools used for this created by Google and Facebook.

Cross Entropy (CE) takes place between:

P(X): predictions

Q(X): label

Both of these are distributions:

$CE = \Sigma$ (of x) $Q(x)\log_2(P(x))$:

We are trying to minimize this and cross entropy is an example of a "loss function."

Background Knowledge

If we know some of the parameters, we will need less data to get a certain accuracy.

In addition, the more background knowledge we have, the harder it is to break a model and this relates to how nowadays, people may complain about supervised learning and how it requires a lot of data.

Lecture 17A: Decision Trees

Decision Trees and Random Forests are used to build **classifiers**, a type of machine learning system that makes decisions. There exists BOTH Bayesian Network Classifiers and Neural Network Classifiers.

Inputs are called characteristics or "instances."

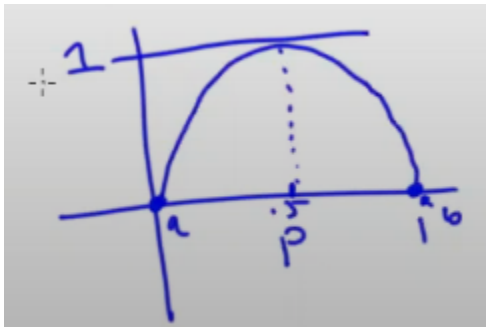
Outputs are called decisions.

Entropy can be used to quantify uncertainty: $ENT(X) = -\Sigma \text{ (of x): } Pr(x)\log_2(Pr(x))$

This formula is the same as the cross entropy between a distribution and itself.

Higher Entropy means more uncertainty.

Lower Entropy means less uncertainty (or more certainty).



- Entropy Graph: Y-Axis is Entropy and X-Axis is Probability

<u>Conditional Entropy</u>

If we start with ENT(X) and we observe that Y = y,

$ENT(X|y) = -\Sigma \text{ (of x) } Pr(x|y)\log_2(Pr(x|y))$

If we start with ENT(X) and we want to observe Y, but do not know its value, then:

$ENT(X|Y) = -\Sigma \text{ (of y) } Pr(y)ENT(X|y)$

Conditional Entropy will never increase as a result of observing another variable/other information. We are specifically talking about the average entropy not necessarily a single value.

$ENT(X|Y) <= ENT(X)$

<u>Classifiers</u> use Supervised Learning and Labeled Data which has feature attributes (input) and a "class" (output)

Decision Trees are "**interpretable**" since they are easy to read compared to a neural network. The more deep (or more **depth** that) that a decision tree is will signify how complex it is.

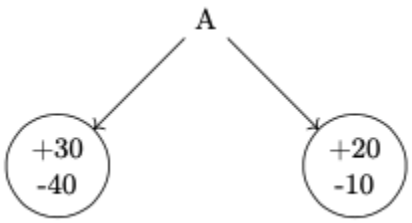
Splitting is when we branch on some sort of variable. It is basically making a choice.

The **leaves** of a decision tree are the decisions.

Lecture 17B: Decision Trees & Random Forests

When we choose an attribute for a decision tree, we want to choose one that splits the tree more (will give us more dramatic results in a sense).

To quantify this, we can use entropy. We use a score computed by conditional entropy in order to choose a split.



M	
Low	30/70
High	40/70

ENT = 0.985 (for first table up above)

M	
Low	20/30
High	10/30

ENT = 0.918 (for second table up above)

ENT(M|A) = (0.7)(0.985) + (0.3)(0.918) = 0.965

Decision Tree Algorithm:

```
function DECISION-TREE-LEARNING(examples, attributes, parent_examples) returns a tree
  if examples is empty then return PLURALITY-VALUE(parent_examples)
  else if all examples have the same classification then return the classification
  else if attributes is empty then return PLURALITY-VALUE(examples)
  else
      A ← argmax_{a ∈ attributes} IMPORTANCE(a, examples)
      tree ← a new decision tree with root test A
      for each value v_k of A do
          exs ← { e : e ∈ examples and e.A = v_k }
          subtree ← DECISION-TREE-LEARNING(exs, attributes − A, examples)
          add a branch to tree with label (A = v_k) and subtree subtree
      return tree
```

Evaluation

If we have a dataset with 100 data points, we can decide to do the following: Split it into 80/20 for the training and testing datasets, respectively and then repeat and take the average of the scores for accuracy.

This is known as **cross validation**.

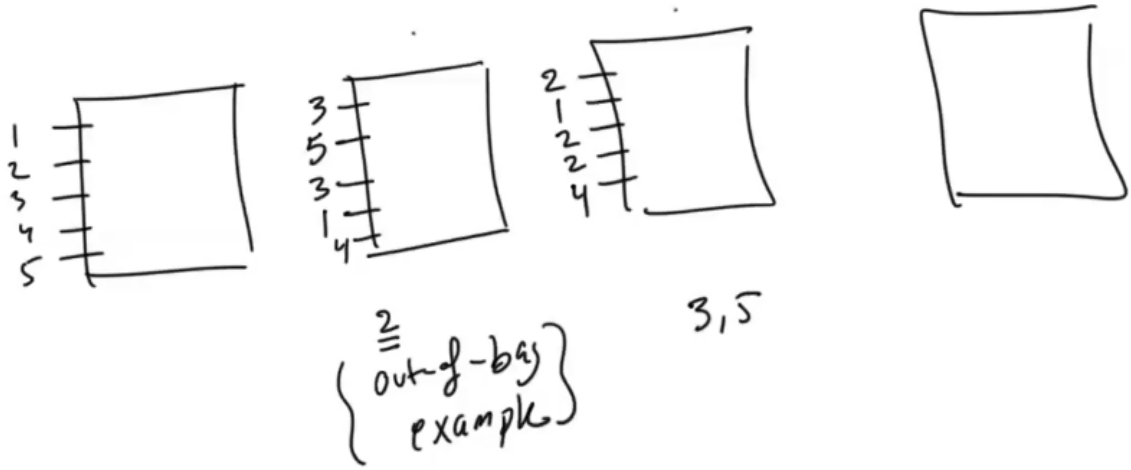
In terms of Interpretability: Decision Trees > Bayesian Networks > Neural Networks.

Random Forests

The main idea is that we are going to build a bunch of decision trees and do a **majority vote** on them to determine the decision/output.

This is an **ensemble learning method**.

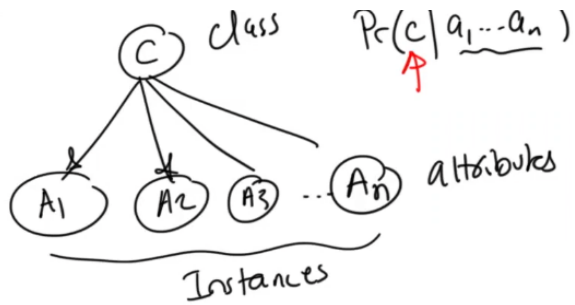
We can also do something called bootstrapping which is shown by the picture down below.



- Choosing a random subset of the attributes when considering what to split on next

Bayesian Network Classifiers

A Naive Bayes structure is as follows: (Note that attributes are also called features)



We have a **Threshold** T for classifying inputs where

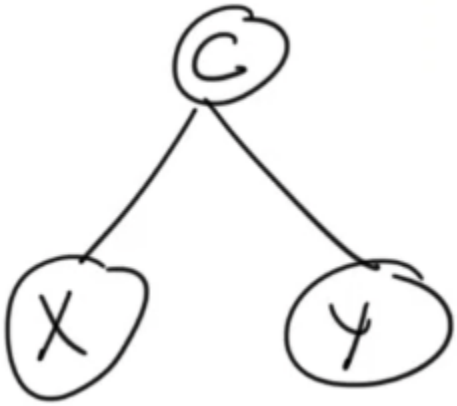
c: Pr(C|a1… an) >= T

¬c: Pr(C|a1… an) < T

From here, we would calculate the probability, Pr(C|a1… an), and if it is >= to the threshold, we can signify that C is true. Otherwise, it is false.

Doing inference in this case is easy because the time complexity is O(n*(d^w)) and we know that trees have a treewidth of w = 1. In addition, we know that Naive Bayes Classifiers are trees.

Naive Bayes Classifier



If we have the following picture up above, then we can say that:

- Pr(c|x, y) = (Pr(x, y|c) * Pr(c)) / Pr(x, y) by Bayes Rule
- We also know that X and Y are independent given because X and Y are desparated by C, therefore Pr(x, y|c) = Pr(x|c) * Pr(y|c).

⇒ (Pr(x|c) * Pr(y|c) * Pr(c)) / Pr(x, y)

We can also compute Pr(x, y) = Pr(x, y|c) * Pr(c) + (Pr(x, y|¬c) * Pr(¬c)

Explainable AI

The table we drew in class can be represented efficiently using Boolean Circuits and we can generally make a tractable circuit from the data.

The data compiles into a circuit with some input-output behavior.

In some cases, some attributes may be more important than others and this is known as **PI-explanation**.

Lecture 18: Neural Networks

There have been lots of ups and downs in the history of neural networks.

The impact has included image analysis (vision) with CNNs (Convolutional Neural Networks), Sequence to Sequence Translation (Speech and Text) and RNNs (Recurrent Neural Networks).

There has also been a debate in AI between these two approaches:

Model-Free (Function/Curve Fitting): The category that neural networks fall under. Most people argue that we have gotten where we are today with neural networks due to advances in computation and labeled data.

Model-Based (Reason approach): This is what we have mostly been doing.

<u>Basics</u>

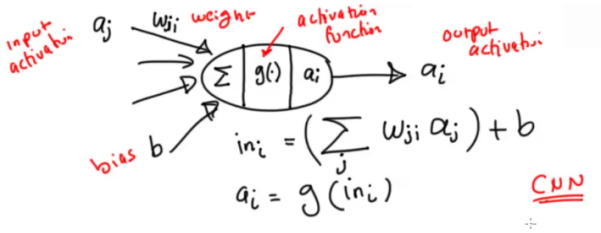
Feed-Forward Neural Networks are trained in a supervised fashion using labeled data. The different parts we will talk about include:

Neurons
Neural Networks (Syntax, Semantics)
How they are trained

<u>Neurons</u>

Neurons are the building block of Neural Networks. We will talk about the following:

Activations (input, output)
Weights
Bias
Activation Functions:
- Step
- Sign
- Sigmoid
- ReLU



Neurons have inputs from other neurons and outputs themselves and the inputs and outputs are numbers. In addition, there exists a weight for each input.

In the picture above, w_{ij} is the weight going from activation j to the neuron i.

Activation Functions

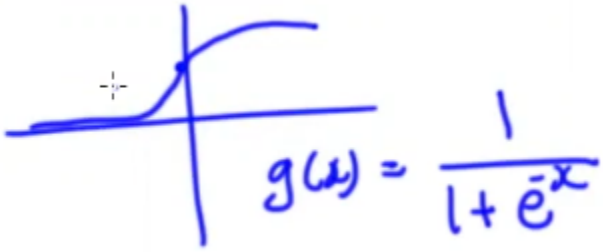
Step Function

g(x) = 1 if x >= t
g(x) = 0 if x < t

Sign Function

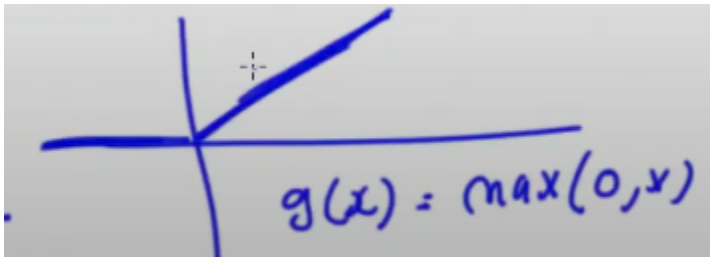
g(x) = 1 if x >= 0
g(x) = -1 if x < 0

Sigmoid Function



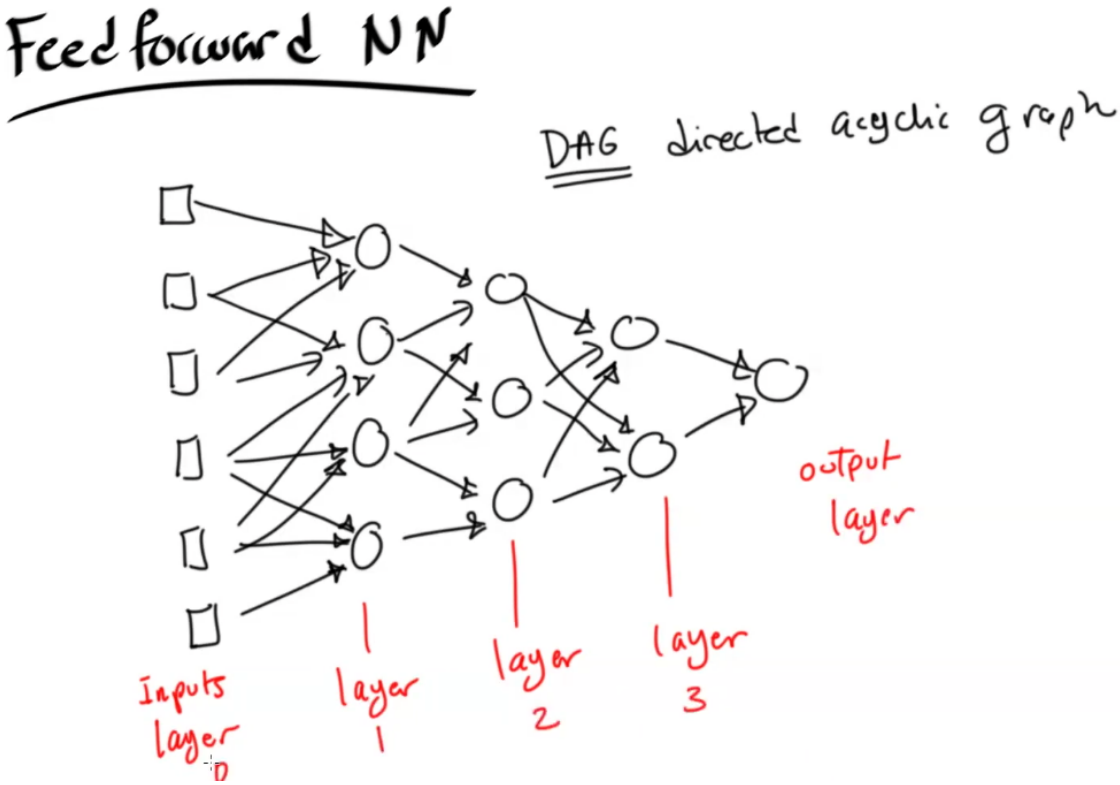
g(x) = 1/(1+e^-x)

ReLU (Rectified Linear Unit) Function



g(x) = max(0, x)

Feedforward Neural Networks

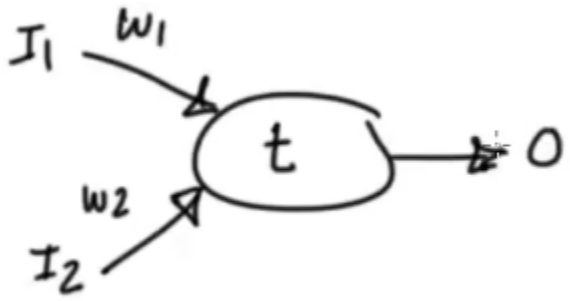


This is represented as a DAG (Directed Acyclic Graph) where we have an input and output layer as well as layers in between.

When we need more depth, this is known as a multi-layer Neural Network. Advances have led to training deep neural networks or more popularly, "deep learning."

Neural Networks are **universal function approximators** and are expressive enough to where we can approximate them to an arbitrary error (f(x1…xn) → ε.

Neurons with Step Activation Functions

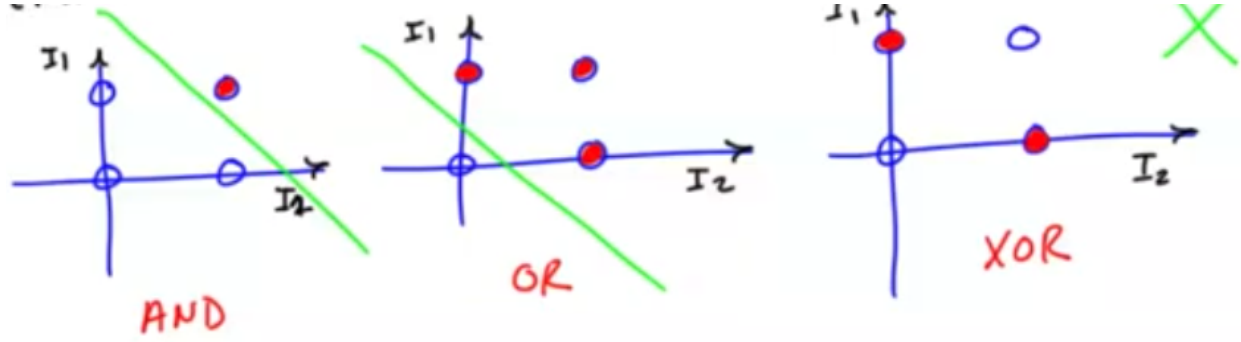


We want to make the above neuron behave like a "AND" gate.

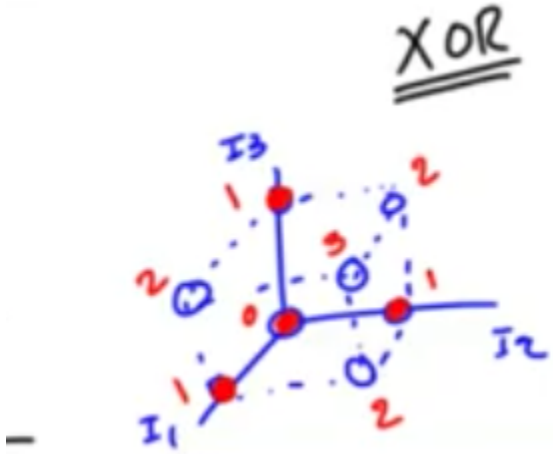
We can set w1 = w2 = 1 and t = 1.5 to satisfy this.

Although we can form an OR gate and an AND gate with a single neuron, we can NOT do this with an XOR gate. This is due to the fact that XOR is NOT linearly separable.

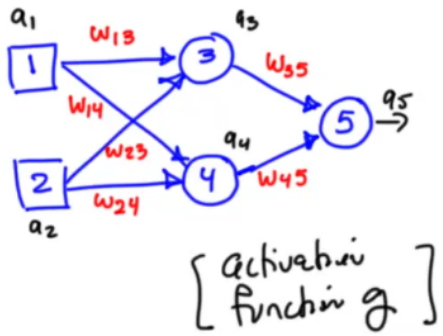
Linearly Separable Functions



Although XOR can NOT be represented by a 2D graph as shown above, it may be able to be represented by a 3D graph as shown below.



Neural Network as a Function



$a_5 = g(w_{35}a_3 + a_4w_{45})$
$= g(w_{35}\, g(w_{13}a_1 + a_2w_{23}) + (w_{45}\, g(w_{14}a_1 + a_2w_{24}))$

$a_5 = f(a_1, a_2, w_{13}, w_{14}, \dots, w_{45})$: a_1, a_2 represent **inputs** while w_{13}, w_{14}, … , w_{45} represent weights.

This neural network can be described as a **complex nonlinear function** because we can control that by setting the values of the weights.

We want to set the weights so that they match the labeled data.

When we set values for a1 and a2:

(1) $a_5 = f_1(w_{13}a_3 + a_4w_{45})$
(2) $a_5 = w_{35}(w_{35}a_3 + a_4w_{45})$

This is what function fitting essentially is.

Loss Function: This is the optimization criteria and what we are trying to optimize. Examples include:

(1) **CE: Cross Entropy**
(2) **MSE: Mean-Squared Error**

MSE (Mean-Squared Error) = **$(1/N) \Sigma$ (from i = 1 to N) $(NN(I_i) - L_i)^2$**

The "$NN(I_i)$" part can be formalized as $f(w_1, \dots, w_k)$.

Gradient Descent

We compute the partial derivative of the function at a particular point and if it is positive, then we can increase the value by going backward. We then increase x0 by epsilon and so on and this is a "step".

$Y = F(x)$
$Y = f(w_1, w_2, \dots, w_k)$ where $w_1, w_2, \dots, w_.$ are all vectors.

This is the loss function,

Since we are computing the partial derivative of f with respect to w_1, the partial derivative of f with respect to w_2 all the way up to the partial derivative of f with respect to w_k, this is where we get the "gradient" part of gradient descent

In general, there are tools such as TensorFlow and ADAM optimizer that automatically do all of this for us.

If we want to check accuracy, we can do the following:

Assume we have 2000 images. We can then break this up into 1600 images (training data) and 400 images (testing data). From this, we will get some sort of accuracy (lets say 98%).

Training Neural Networks

First, we take our data and split it into training data (80%) and testing data (20%).

We run gradient descent on the training data to optimize the loss.

We do the testing by checking the **performance metric**.

A single iteration of gradient descent is an **epoch** which has a loss and a metric.

Generally, we take our data set and randomly divide it into batches (64 is a typical size) and apply 1 step of gradient descent on each batch – all of this is w/in one epoch. Beforehand, we need to decide on our batch size: 32, 64, 128, etc.

- When we go to our second epoch, we split our data into batches again.

BUT, we also need we need to know when to stop:

An approach we could take for this is take our data and split it into training (80%) and testing data (20%). After every epoch, we report the **performance metric** on the validation/test data. We stop when we stop making enough improvements on the validation/test data, then we stop.

- Tools that do this very well include Keras.
- We can also put an upper limit on epoch

Splitting the data into batches proves useful as you can do the following:

Treat them in parallel with GPUs, however there is also randomization where these batches are picked up at random from the dataset and you get Stochastic Gradient Descent which is evaluation on Gradient Descent

Lecture 19A: Convolutional Neural Networks (CNNs)

Topics for Lectures 19A & 19B:

- Computer Vision (for image analysis) is done using **Convolutional Neural Networks (CNNs).**
- Sequence-to-Sequence Translation (done with speech, text) are known as **Recurrent Neural Networks (RNNs).**
- Perspective on "Model-Based" vs "Model-Free" (Neural Networks) approaches to Artificial Intelligence (AI).

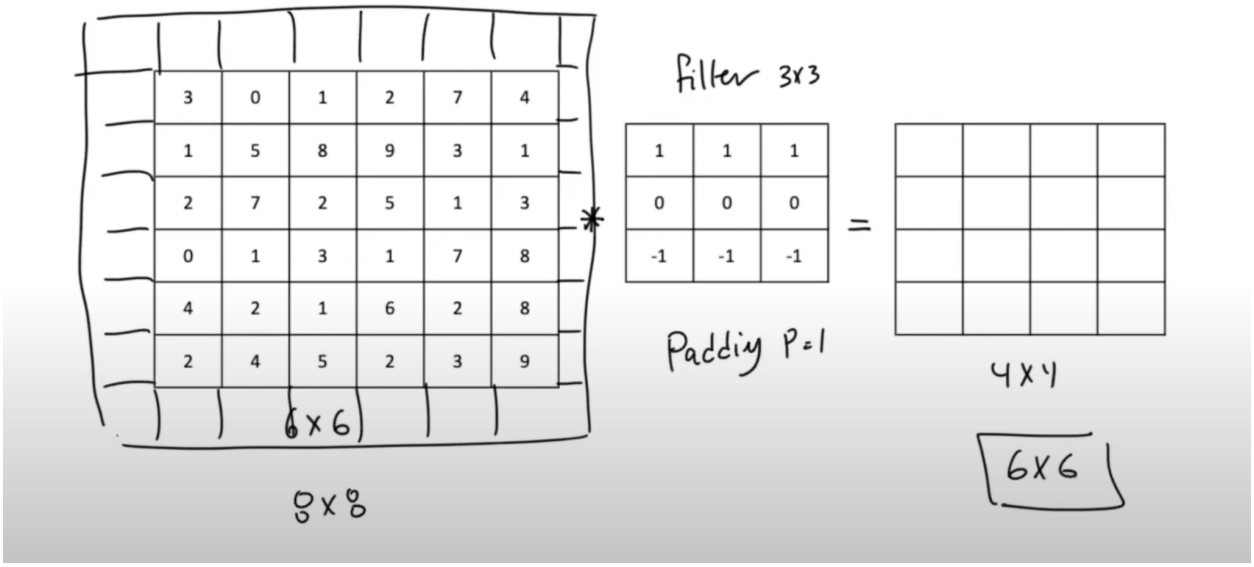
---

(1) Local Detection of features/patterns/concepts
(2) Aggregation/Abstraction

Convolution

Convolution is about having some type of neuron called a **filter** that roams around an image, where cells represent pixels.

If we have a 6x6 image, then it can be gray scaled or colored Neurons are described by a matrix and the inputs.



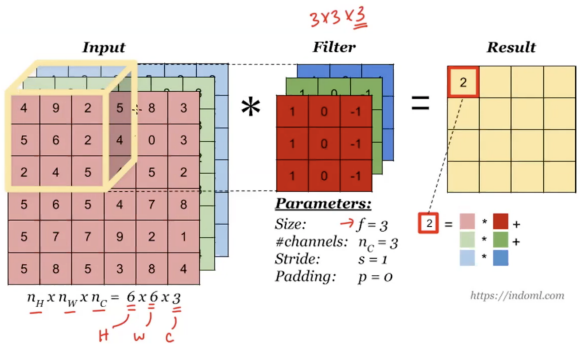
In the above example, **filter size (f)** is a hyperparameter which you have to choose as you are designing a filter.

Padding (p) is something that we can choose which will emphasize the edge features.

We can also have **stride (s)** which is essentially the step size at which we "roam" or the

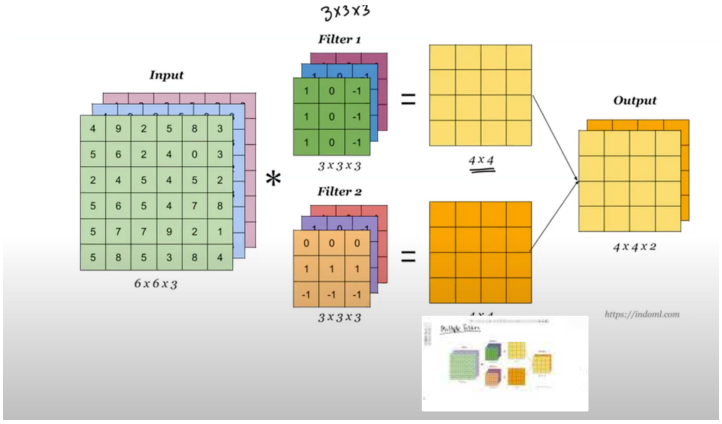
size in the middle of iterations.

If you increase the stride/step size, then you end up shrinking things further.



In the above image, we now have a 3x3 convolutional neural network, but we have to now use a filter of 3x3x3 (note that the last dimension in the filter must match the last dimension in the network).

Multiple Filters



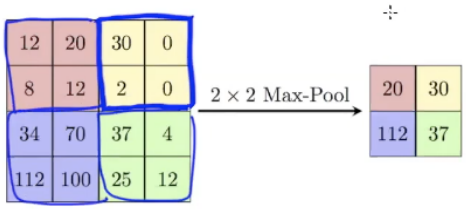
The "orange" boxes are essentially the same neuron in the sense that they have the same weights. We are just taking the same weight and applying it multiple times.

- When the output is "4x4x2", the 2 represents the number of filters we use (which in this case is 2).
- If we were to change the # of filters from 2 to 15, then our output matrix would be 4x4x16.

  If we additionally changed our padding size from 0 to 1, then our output matrix would be 6x6x16.

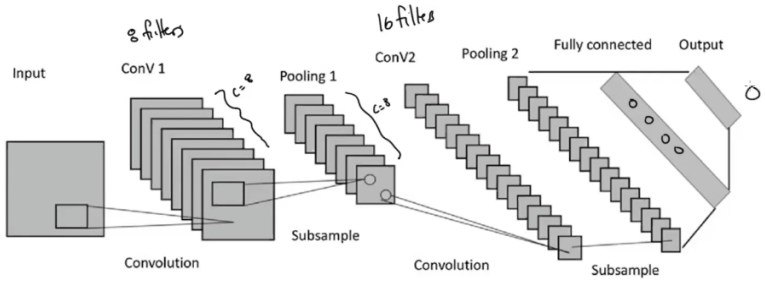
Max Pooling

This is very similar to convolution, but when we are looking at a block, we take the max of a particular set of blocks. We can see this example below.



Here, we use a stride (s) of 2, a padding (p) of 0, and a filter size (f) of 2.

CNN Architecture (Example)



As you move to the right, the number of channels increases, but the height and width starts to shrink.

The number of channels is controlled by the number of filters.

The way to shrink the height and width is through stride (s).

Overall, for the convolution layers, the number of weights is controlled whereas for max pooling, there are no weights!

When CNNs are implemented, **tensors** are multi-dimensional arrays that can be used for implementing neural networks in general.

Lecture 19B: Model-Based vs Model-Free Supervised Learning

Promise & Limitations of Neural Networks

(1) Hungry for data: need lots of data
(2) Brittle/lack of robustness: easy to break
(3) Not as easy to explain: {"why?"} → Explainable AI
    Based on Model-Free Learning (under "function-fitting/curve-fitting)

The answer for the future of AI Researchers is to combine both Model-Free and Model-Based Learning!

Brittle/Lack of Robustness

Small changes to data can lead to the wrong results/outputs which created adversarial attacks.

> These networks do NOT have knowledge of what they are trying to learn (these are the "Model-Free" Learning)

Using a BN (Bayesian Network) + BK (Background Knowledge), the research shows that this Model-Free + Model-Based Learning approach beats out the Convolutional Neural Network (CNN) in both low and high data situations. However, JUST using a BN (Bayesian Network) without BK (Background Knowledge) yields lower accuracy than both of the other approaches mentioned previously.

> The "BN + BK" approach is injecting every piece of knowledge we know into the circuit, particularly the Arithmetic Circuit (AC).

Overall, Model-Based Learning comes under representation and reasoning which is trying to understand what something is and how to identify it.

> Model-Free Learning, on the other hand, comes under "curve-fitting" and tends to be able to identify correctly or produce the desired output by just being given enough data.

We need both of these approaches in the future!