Introduction to AI

Index

1. Search problems

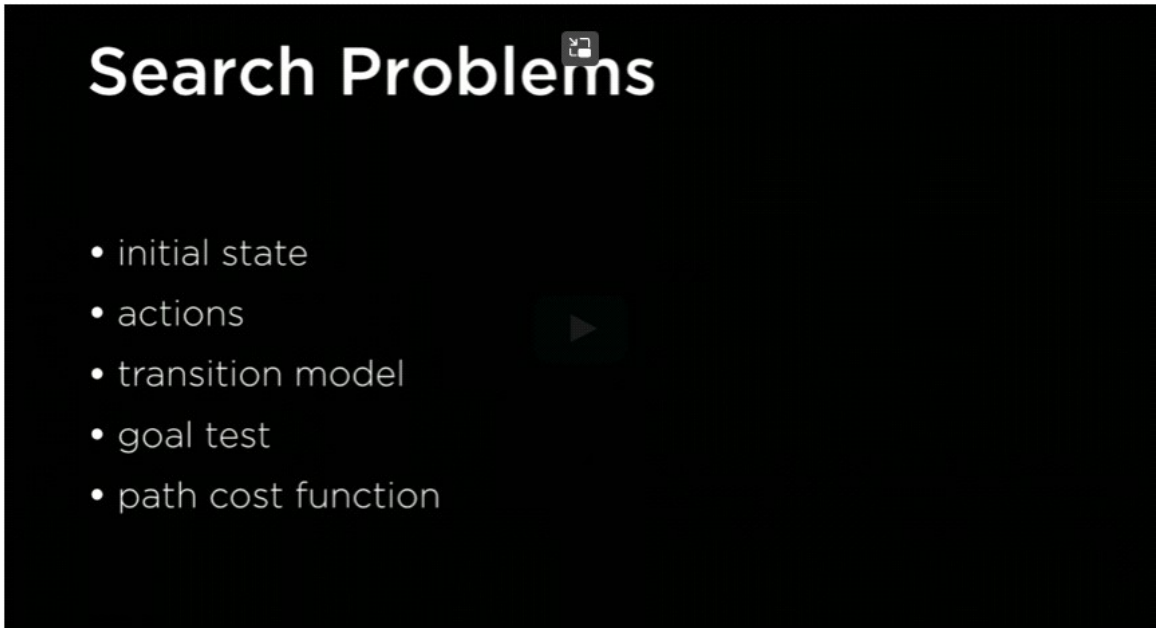
1.1. Basic terms of search problems

State space: the set of all states reachable from the initial state by any sequence of actions. It is represented by nodes and arrows, the arrows represent the actions that we could take.

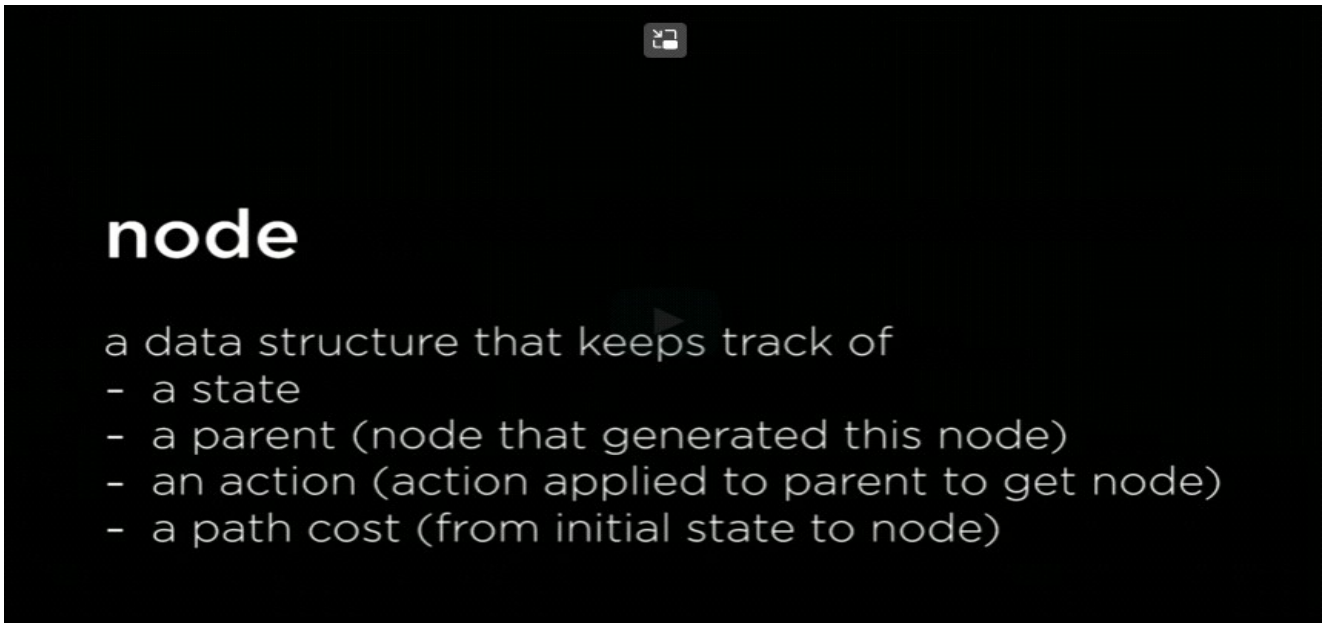
Transition model: what happens to x state after we apply y action.

Way state: way to determine whether a given state is a goal state.

Path cost: numerical cost associated with any given path.



1.2. Nodes



1.3. Types of Search

Depth first search: search algorithm that always expands the deepest node in the frontier.

Breadth-first search: search algorithm that always expands the shallowest node in the frontier.

Uninformed search: search strategy that uses no problem-specific knowledge.

Informed search: search strategy that uses problem-specific knowledge.

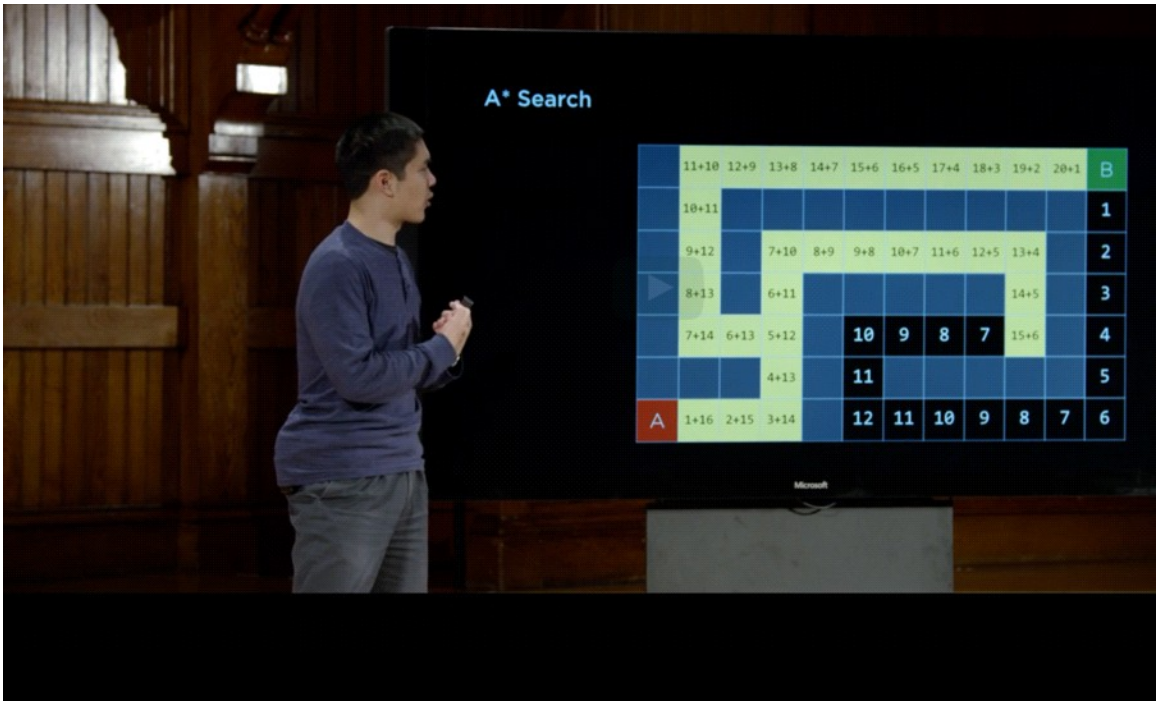
Greedy best first search: Search algorithm that expands the node that is closest to the goal, as estimated by a heuristic function h(n). It finds the solution according to what's best locally, not necessarily in the big picture, that's why it's called greedy.

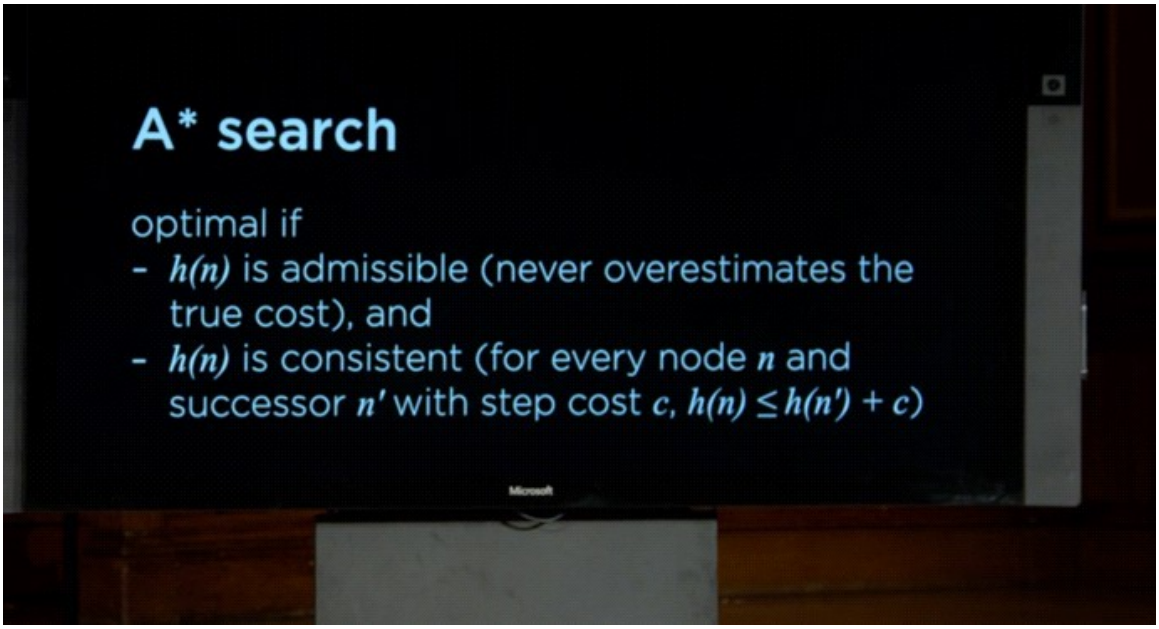
Manhattan distance: The number of squares up and to the right that you would need to travel to get to the destination in the maze.

A* search: search algorithm that expands node with lowest value of g(n) + h(n)

g(n) = cost to reach node

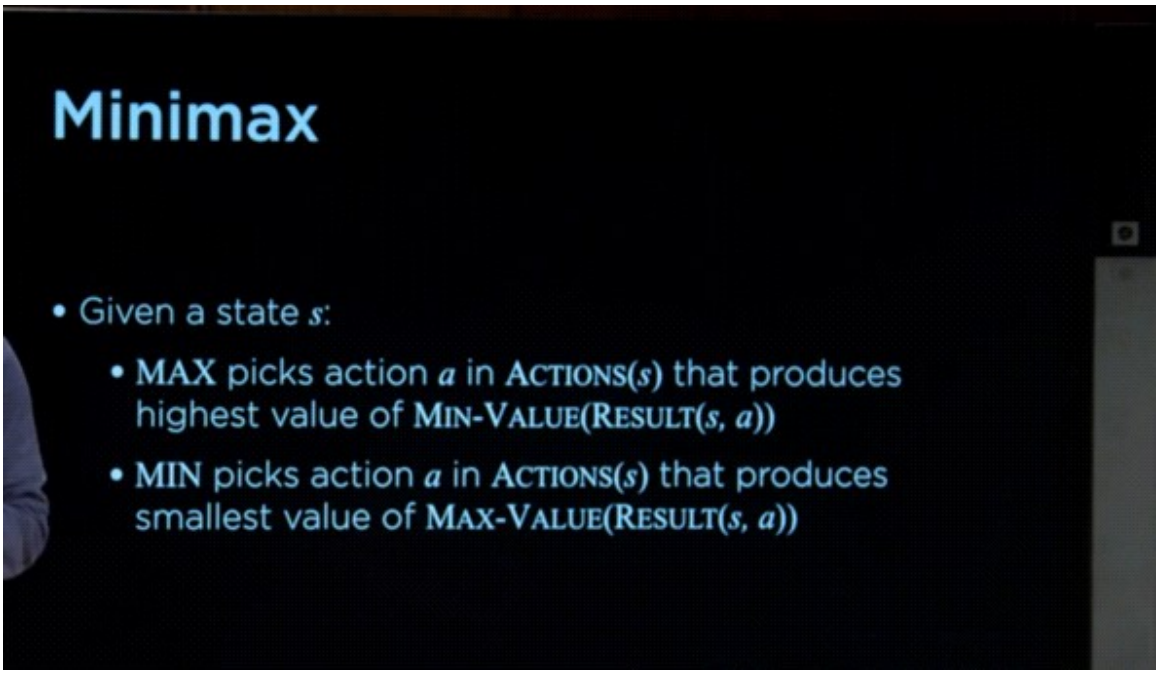
h(n) = estimated cost to goal





1.4. Minimax

Minimax: MAX (X) aims to maximize the score. MIN (O) aims to minimize score.



Minimax

```
function MAX-VALUE(state):
    if TERMINAL(state):
        return UTILITY(state)
    v = -∞
    for action in ACTIONS(state):
        v = MAX(v, MIN-VALUE(RESULT(state, action)))
    return v
```

Minimax

```
function MIN-VALUE(state):
    if TERMINAL(state):
        return UTILITY(state)
    v = ∞
    for action in ACTIONS(state):
        v = MIN(v, MAX-VALUE(RESULT(state, action)))
    return v
```

We can do Alpha-Beta pruning in order to optimize minimax, pruning is the idea that if you have long deep search tree you don't have to look at everything to know what the best course of action is.

Depth-Limited Minimax: self-explanatory name.

Evaluation function: function that estimates the expected utility of the game from a given state. Used for depth-limited minimax.

1.5. Quiz

✓ Between depth first search (DFS) and breadth first search (BFS), which will find a shorter path through a maze? *1/1

- ○ DFS will always find a shorter path than BFS
- ○ BFS will always find a shorter path than DFS
- ○ DFS will sometimes, but not always, find a shorter path than BFS
- ◉ BFS will sometimes, but not always, find a shorter path than DFS ✓
- ○ Both algorithms will always find paths of the same length

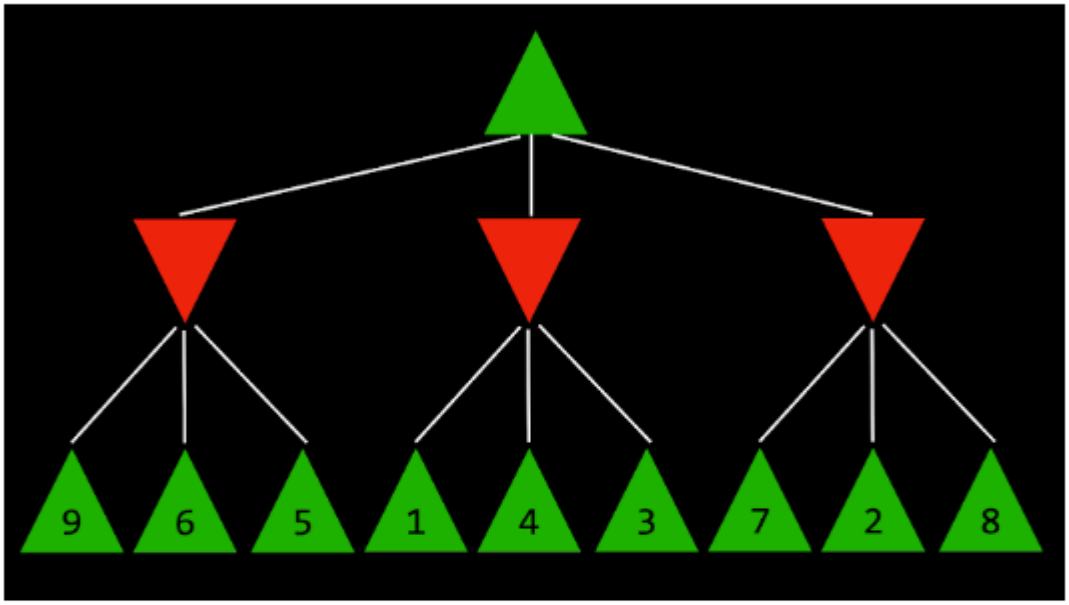
✓ Of the four search algorithms discussed in lecture — depth-first search, breadth-first search, greedy best-first search with Manhattan distance heuristic, and A* search with Manhattan distance heuristic — which one (or multiple, if multiple are possible) could be the algorithm used? *1/1

- ○ Could only be A*
- ○ Could only be greedy best-first search
- ◉ Could only be DFS ✓
- ○ Could only be BFS
- ○ Could be either A* or greedy best-first search
- ○ Could be either DFS or BFS
- ○ Could be any of the four algorithms
- ○ Could not be any of the four algorithms

✓ Why is depth-limited minimax sometimes preferable to minimax without a depth limit? *1/1

- ◉ Depth-limited minimax can arrive at a decision more quickly because it explores fewer states ✓
- ○ Depth-limited minimax will achieve the same output as minimax without a depth limit, but can sometimes use less memory
- ○ Depth-limited minimax can make a more optimal decision by not exploring states known to be suboptimal
- ○ Depth-limited minimax is never preferable to minimax without a depth limit

The following question will ask you about the Minimax tree below, where the green up arrows indicate the MAX player and red down arrows indicate the MIN player. The leaf nodes are each labelled with their value.



✓ What is the value of the root node? * 1/1

- 2
- 3
- 4
- 5 ✓

2. Knowledge

2.1. Basic terms of knowledge

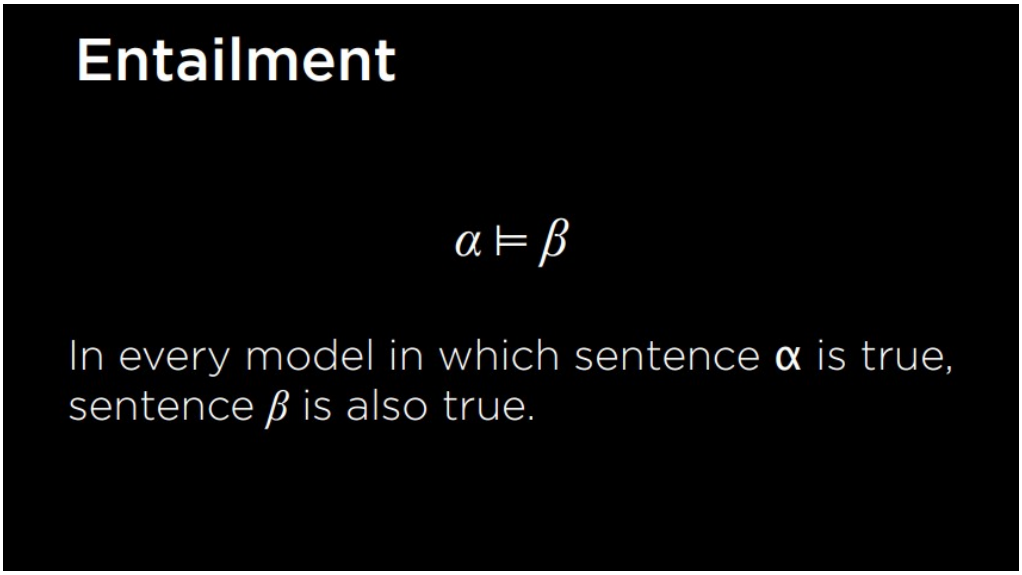
- Knowledge-based agents: agents that reason and act by operating on internal representations of knowledge.
- Sentence: an assertion about the world in a knowledge representation lenguage.
- Propositional Logic: is based on a logic of statements about the world.
  These statements are represented with something called propositional symbols, *P, Q, R*, etc…
- There are also truth tables, which you already know. You have AND, OR, NOT, IMPLICATION, BICONDITIONAL.

  Implication: (then)

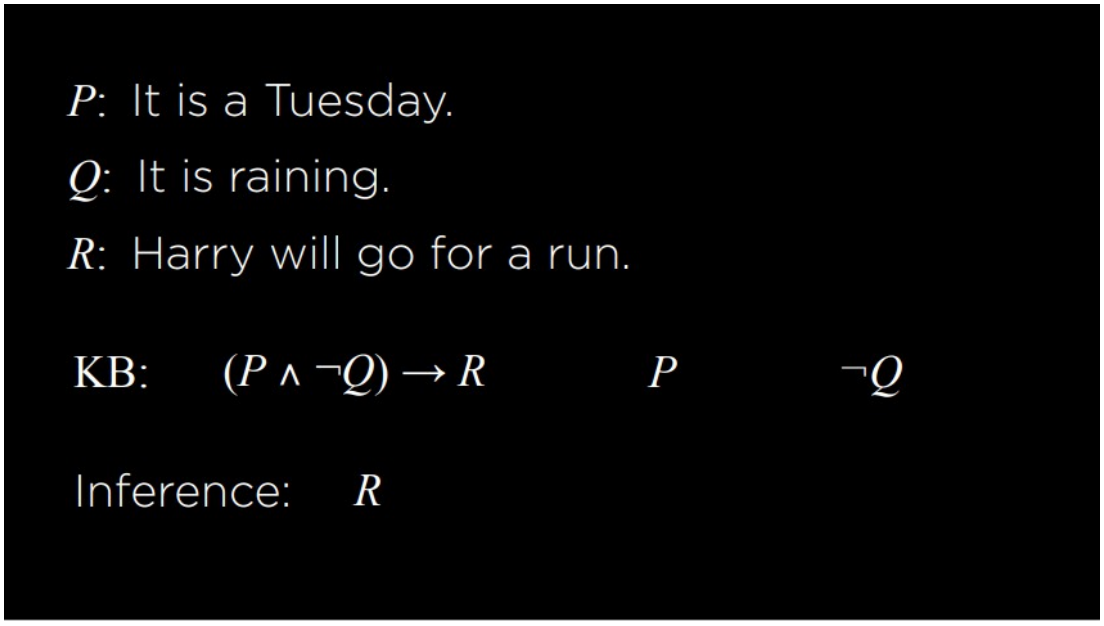
Implication ($\rightarrow$)

P	Q	$P \rightarrow Q$
false	false	true
false	true	true
true	false	false
true	true	true

- Model: assigns a true value to every propositional sentence (a “possible world”).
- Knowledge base: things that our AI knows about the world, a set of sentences, known by a knowledge-based agent.
- Entailment:



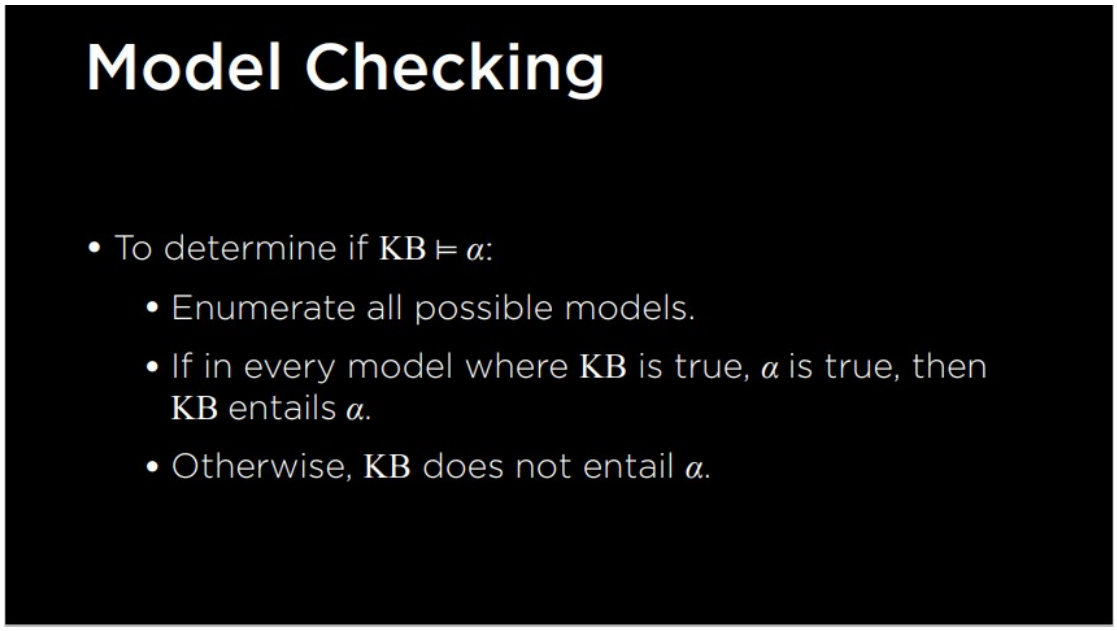
- Inference: the act of deriving new sentences from new ones.

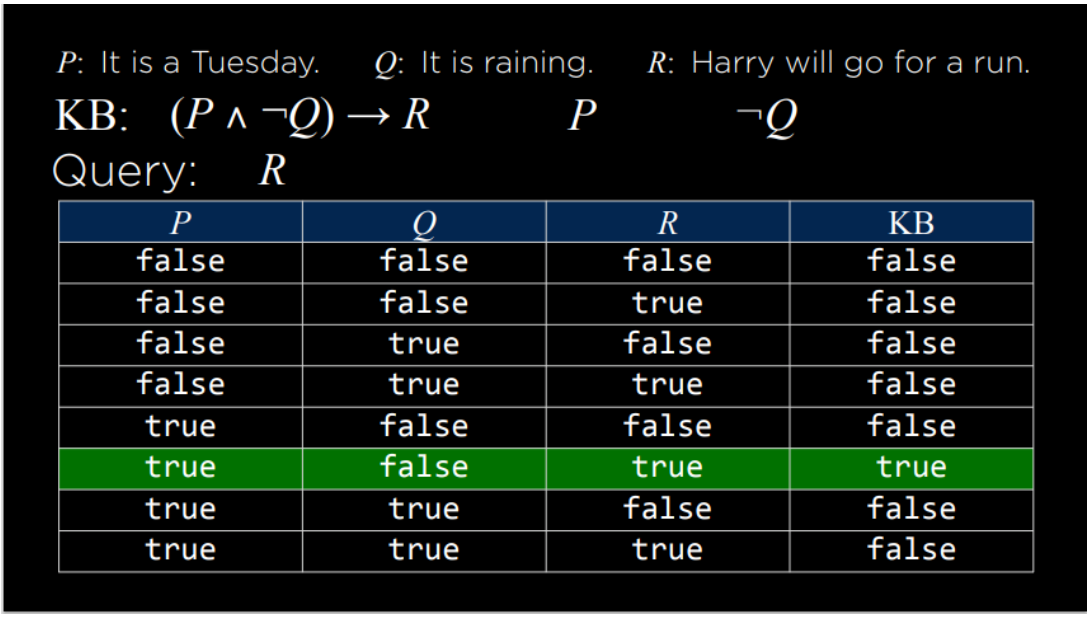


2.2. Inference Algorithms

The goal in all inference algorithms is going to be: Does KB $\vDash$ α ?

2.2.1. Model Checking





P	Q	R	KB
false	false	false	false
false	false	true	false
false	true	false	false
false	true	true	false
true	false	false	false
true	false	true	true
true	true	false	false
true	true	true	false

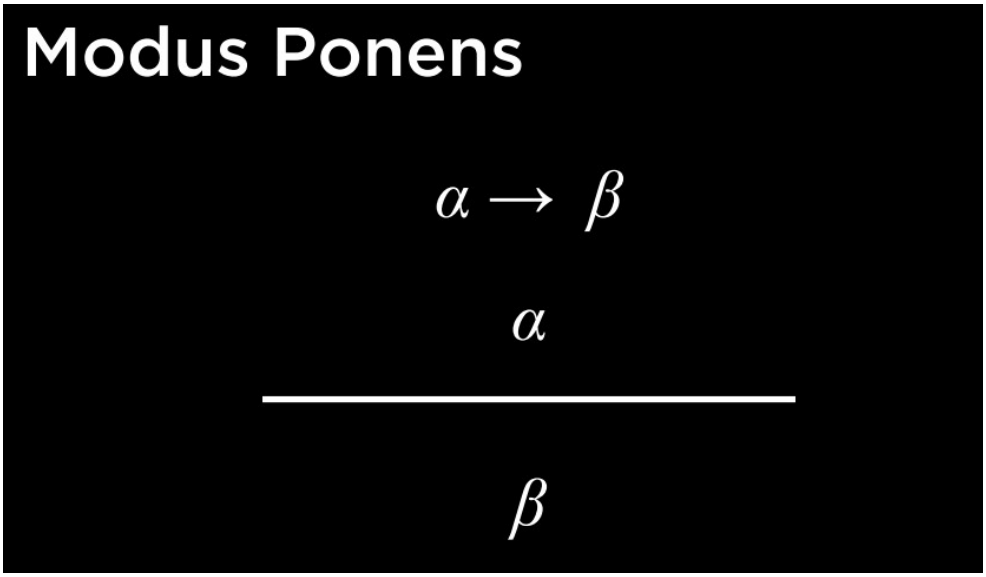
Model checking is not so efficient as an algorithm, because you need to take all of the variables and check all the possibilities that they could be in, so if I have n variables we would have 2^n possible worlds.

2.3. Knowledge engineering

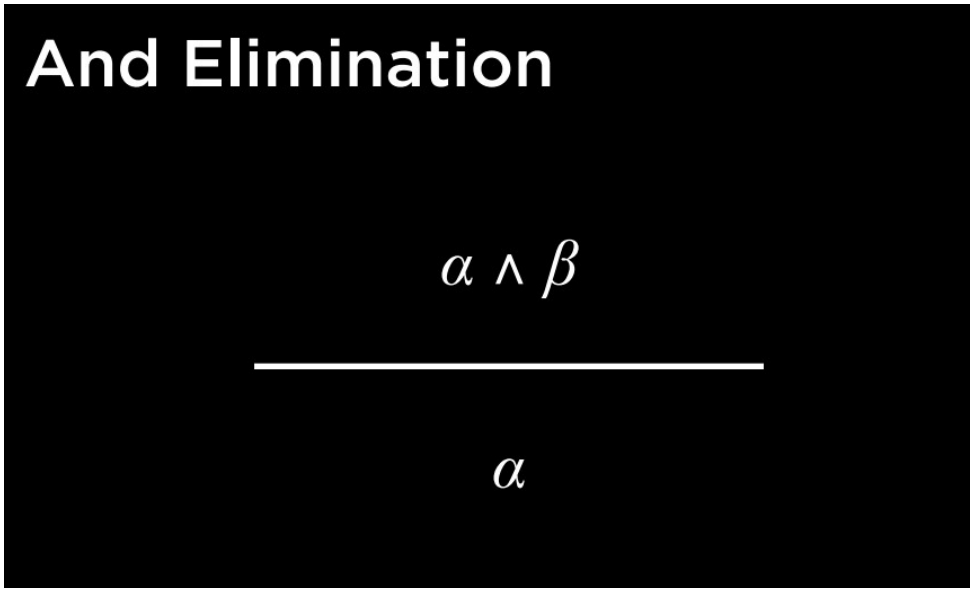
Take a problem to figure out how to distill it down into knowledge representable to a computer.

2.4. Inference Rules

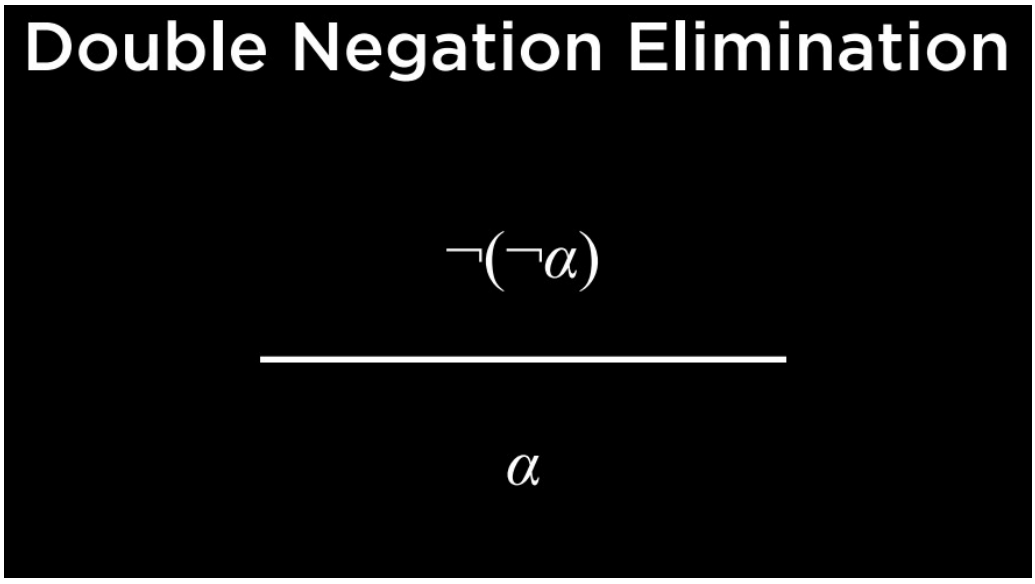
2.4.1. Modus Ponens



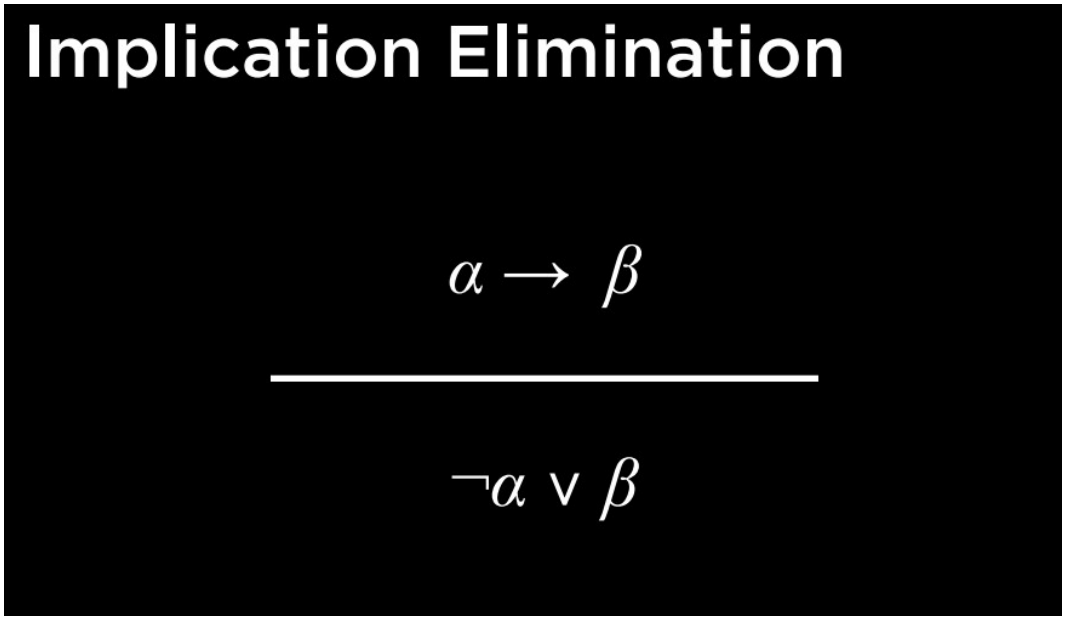
2.4.2. And Elimination



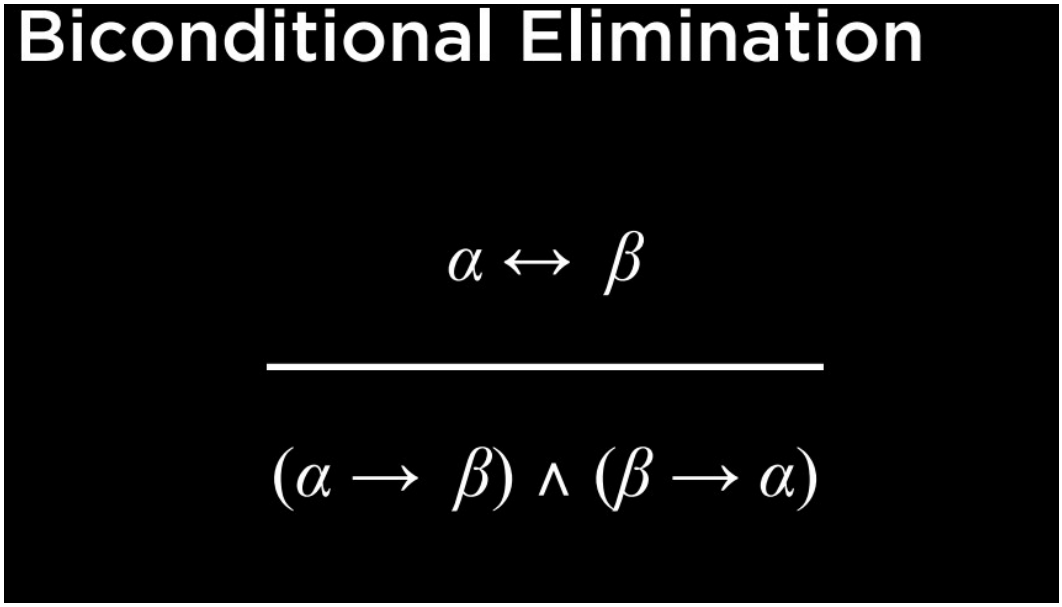
2.4.3. Double Negation Elimination



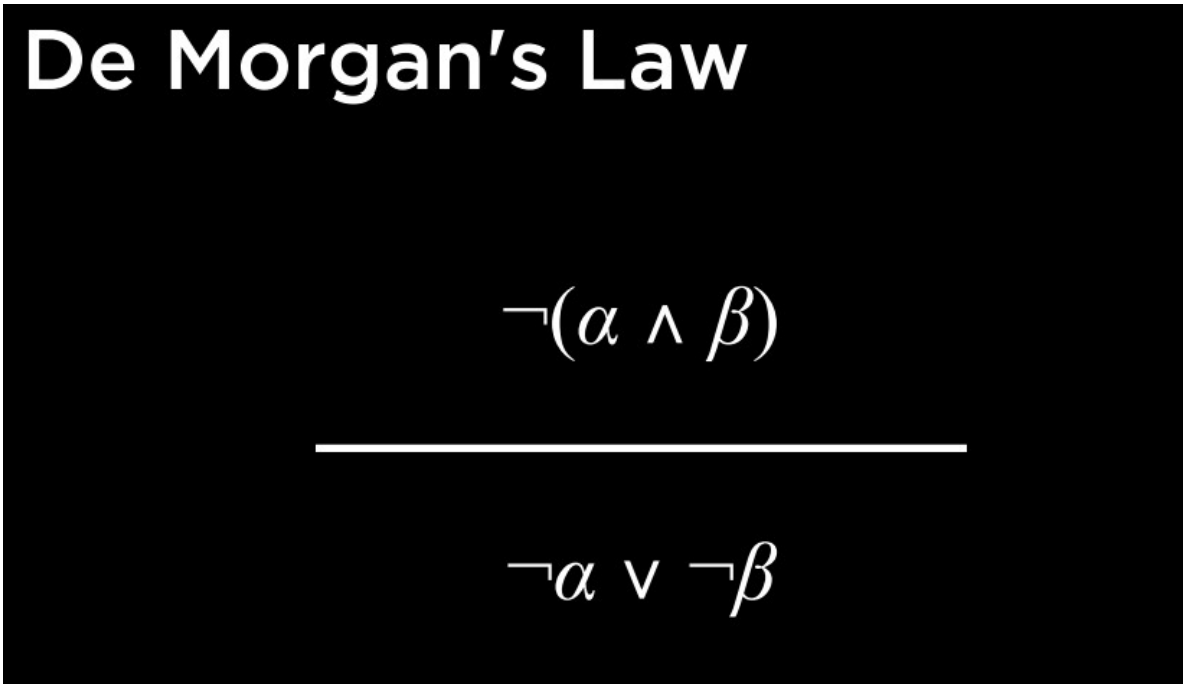
2.4.4. Implication Elimination



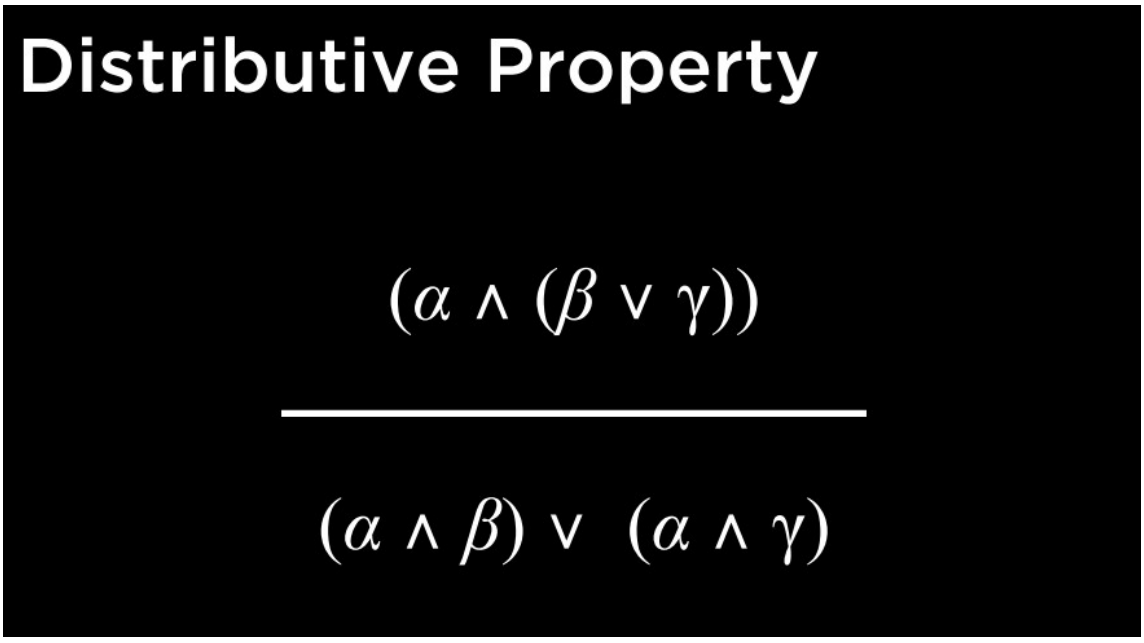
2.4.5. Biconditional Elimination



2.4.6. De Morgan's Law



2.4.7. Distributive Property



2.5. Theorem Proving

Theorem Proving

- initial state: starting knowledge base
- actions: inference rules
- transition model: new knowledge base after inference
- goal test: check statement we're trying to prove
- path cost function: number of steps in proof

2.6. Resolution

$$\frac{P \vee Q \qquad \neg P}{Q}$$

$$\frac{P \vee Q_1 \vee Q_2 \vee \ldots \vee Q_n \qquad \neg P}{Q_1 \vee Q_2 \vee \ldots \vee Q_n}$$

$$\frac{P \vee Q \qquad \neg P \vee R}{Q \vee R}$$

$$\frac{P \vee Q_1 \vee Q_2 \vee \ldots \vee Q_n \qquad \neg P \vee R_1 \vee R_2 \vee \ldots \vee R_m}{Q_1 \vee Q_2 \vee \ldots \vee Q_n \vee R_1 \vee R_2 \vee \ldots \vee R_m}$$

$$\frac{P \vee Q \vee S \qquad \neg P \vee R \vee S}{(Q \vee R \vee S)}$$

$$\frac{P \qquad \neg P}{( )}$$

2.6.1. Clause

- A disjunction of literals. (Disjunction means a bunch of things connected with or, conjunction connected with and) e.g. P v Q v R.
- Conjunctive normal form: logical sentence that is a conjunction of clauses. e.g. (A v B v C) ^ (D v E v F)

Conversion to CNF

- Eliminate biconditionals
  - turn $(\alpha \leftrightarrow \beta)$ into $(\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$
- Eliminate implications
  - turn $(\alpha \rightarrow \beta)$ into $\neg\alpha \vee \beta$
- Move $\neg$ inwards using De Morgan's Laws
  - e.g. turn $\neg(\alpha \wedge \beta)$ into $\neg\alpha \vee \neg\beta$
- Use distributive law to distribute $\vee$ wherever possible

Conversion to CNF

$(P \vee Q) \rightarrow R$

$\neg(P \vee Q) \vee R$ — eliminate implication

$(\neg P \wedge \neg Q) \vee R$ — De Morgan's Law

$(\neg P \vee R) \wedge (\neg Q \vee R)$ — distributive law

Once these clauses are in this form, these clauses are the input to the resolution inference rule.

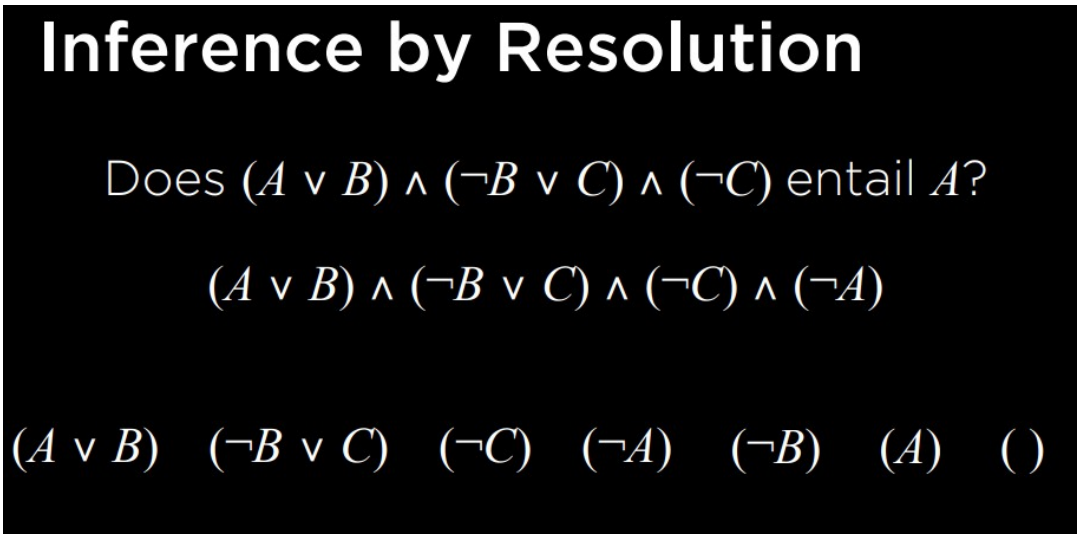
2.6.2. Inference By Resolution

Inference by Resolution

- To determine if $\mathrm{KB} \models \alpha$:
  - Check if $(\mathrm{KB} \wedge \neg\alpha)$ is a contradiction?
    - If so, then $\mathrm{KB} \models \alpha$.
    - Otherwise, no entailment.

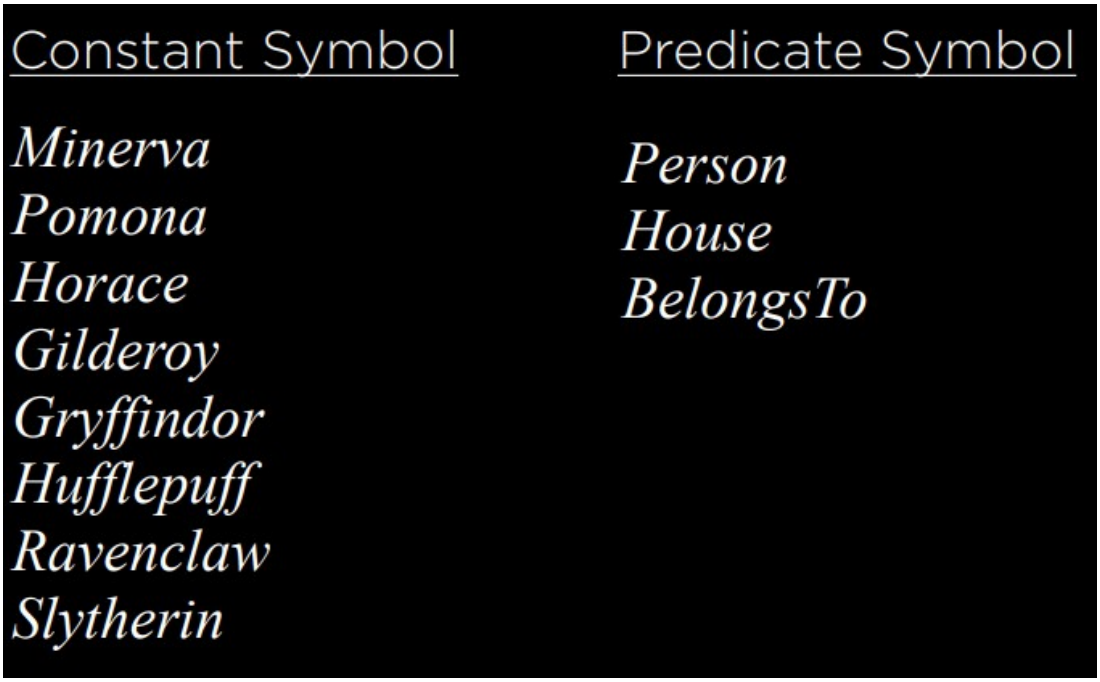
Inference by Resolution

- To determine if $\mathrm{KB} \models \alpha$:
  - Convert $(\mathrm{KB} \wedge \neg\alpha)$ to Conjunctive Normal Form.
  - Keep checking to see if we can use resolution to produce a new clause.
    - If ever we produce the **empty** clause (equivalent to False), we have a contradiction, and $\mathrm{KB} \models \alpha$.
    - Otherwise, if we can't add new clauses, no entailment.



2.7. First-Order Logic

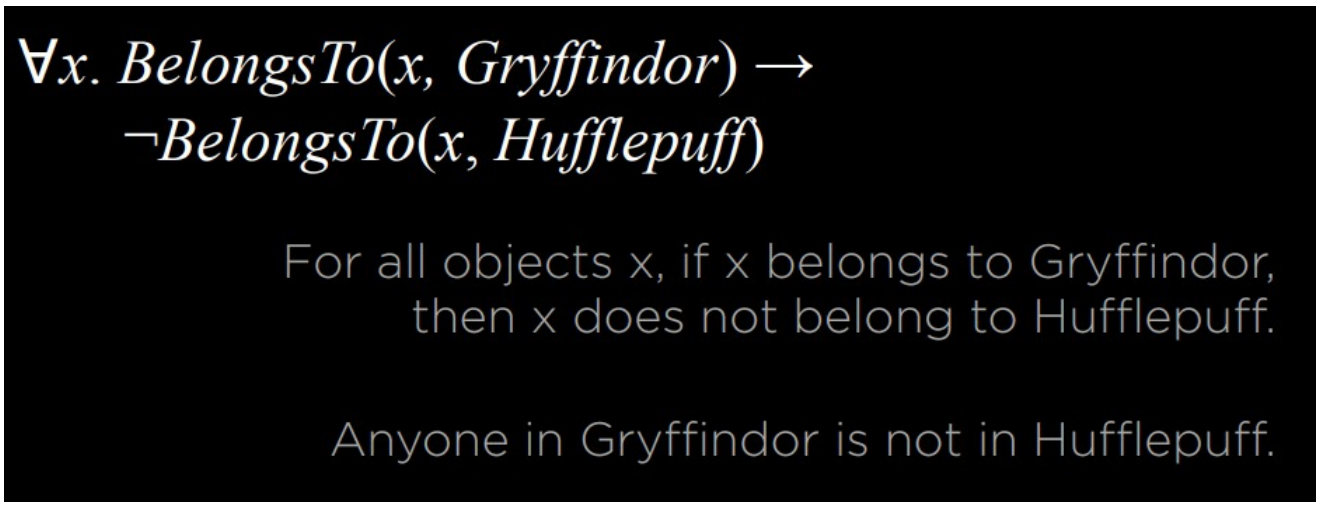
In this kind of logic we're going to have constant symbols that are going to represent objects, and then predicate symbols, which you can think of as relations or functions, that take an input and evaluate them to, like, true or false, for example.



There are a couple of additional features that we can use to express even more complex ideas, these additional features are generally known as quantifiers. There are two main quantifiers: universal quantification and existential quantification.

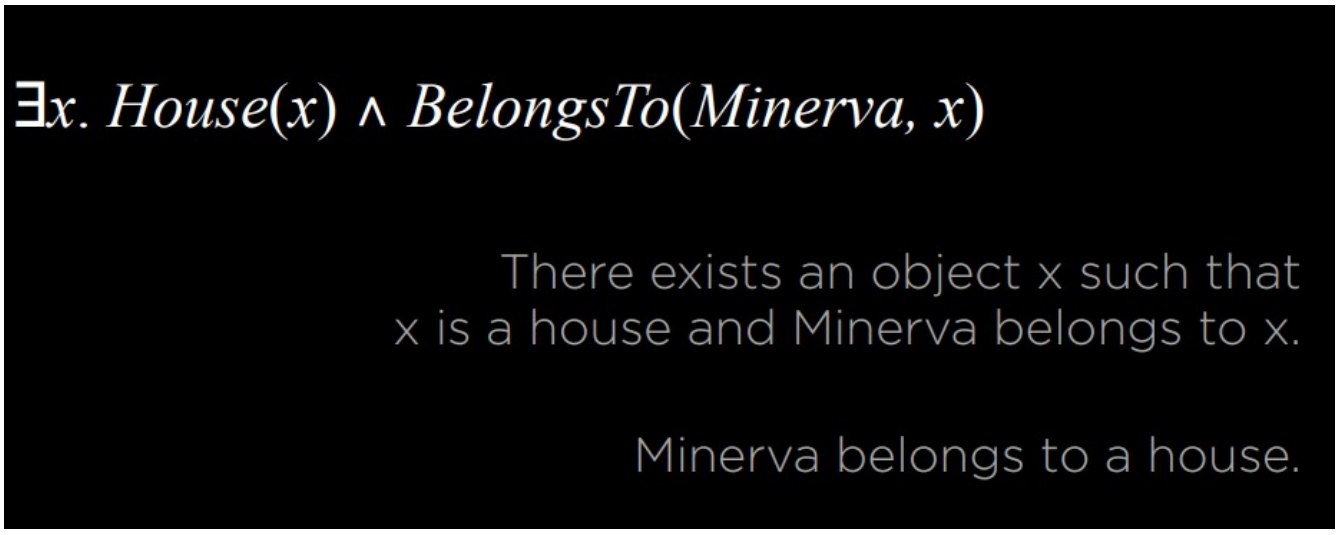
2.7.1. Universal Quantification

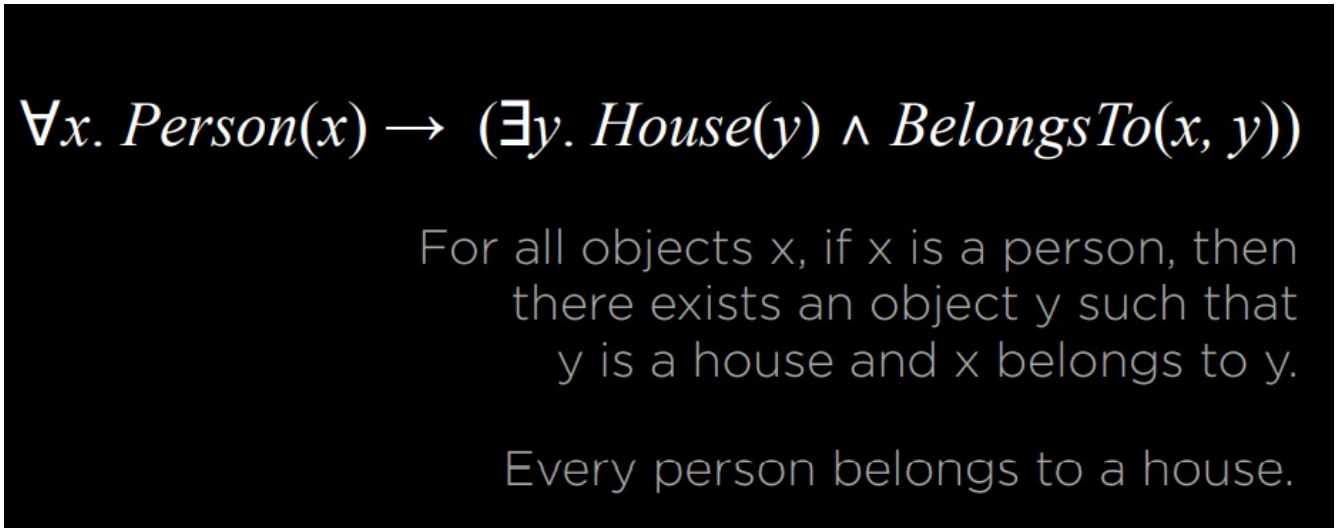
Universal quantification says that something is going to be true for all values of variable.



2.7.2. Existential Quantification

Whereas universal quantification said that something is going to be true for all values of variable, existential quantification says that some expression is going to be true for some value of a variable.





2.8. Quiz

The following question will ask you about the following logical sentences.
1. If Hermione is in the library, then Harry is in the library.
2. Hermione is in the library.
3. Ron is in the library and Ron is not in the library.
4. Harry is in the library.
5. Harry is not in the library or Hermione is in the library.
6. Ron is in the library or Hermione is in the library.

✓ Which of the following logical entailments is true? * 1/1

- ○ Sentence 1 entails Sentence 4
- ◉ Sentence 2 entails Sentence 5 ✓
- ○ Sentence 1 entails Sentence 2
- ○ Sentence 6 entails Sentence 3
- ○ Sentence 6 entails Sentence 2
- ○ Sentence 5 entails Sentence 6

✓ There are other logical connectives that exist, other than the ones discussed in lecture. One of the most common is "Exclusive Or" (represented using the symbol $\oplus$). The expression $A \oplus B$ represents the sentence "A or B, but not both." Which of the following is logically equivalent to $A \oplus B$? *1/1

- ○ $(A \vee B) \wedge \neg (A \vee B)$
- ○ $(A \vee B) \wedge (A \wedge B)$
- ○ $(A \wedge B) \vee \neg (A \vee B)$
- ◉ $(A \vee B) \wedge \neg (A \wedge B)$ ✓

✓ Let propositional variable R be that "It is raining," the variable C be that "It is cloudy," and the variable S be that "It is sunny." Which of the following a propositional logic representation of the sentence "If it is raining, then it is cloudy and not sunny."? *1/1

- ○ $(R \rightarrow C) \wedge \neg S$
- ○ $R \rightarrow C \rightarrow \neg S$
- ○ $R \wedge C \wedge \neg S$
- ◉ $R \rightarrow (C \wedge \neg S)$ ✓
- ○ $(C \vee \neg S) \rightarrow R$

✓ Consider, in first-order logic, the following predicate symbols. Student(x) represents the predicate that "x is a student." Course(x) represents the predicate that "x is a course." Enrolled(x, y) represents the predicate that "x is enrolled in y." Which of the following is a first-order logic translation of the sentence "There is a course that Harry and Hermione are both enrolled in."? *1/1

- ○ $\forall x.\ \text{Enrolled(Harry, x)} \wedge \forall y.\ \text{Enrolled(Hermione, y)}$
- ○ $\forall x.\ \text{Course(x)} \wedge \text{Enrolled(Harry, x)} \wedge \text{Enrolled(Hermione, x)}$
- ○ $\exists x.\ \text{Enrolled(Harry, x)} \vee \text{Enrolled(Hermione, x)}$
- ○ $\exists x.\ \text{Enrolled(Harry, x)} \wedge \exists y.\ \text{Enrolled(Hermione, y)}$
- ○ $\forall x.\ \text{Enrolled(Harry, x)} \vee \text{Enrolled(Hermione, x)}$
- ◉ $\exists x.\ \text{Course(x)} \wedge \text{Enrolled(Harry, x)} \wedge \text{Enrolled(Hermione, x)}$ ✓

3. Uncertainty

3.1. Probability

Every probability must range between 0 and 1 inclusive. 0 is an impossible event.

$0 \leq P(w) \geq 1$. We also have that:

$$\sum_{\omega \in \Omega} P(\omega) = 1$$

unconditional probability

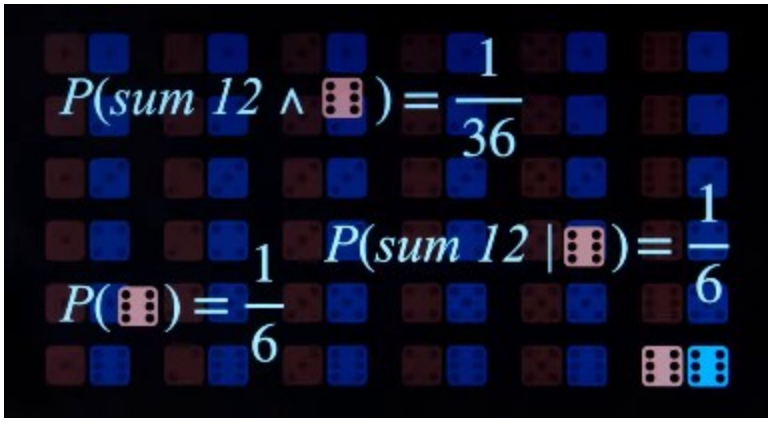
degree of belief in a proposition
in the absence of any other evidence

conditional probability

degree of belief in a proposition
given some evidence that has already
been revealed

P (a|b) a is the condition that we want to check if it's true, and b is the information we know, the evidence.

$$P(a \mid b) = \frac{P(a \wedge b)}{P(b)}$$



random variable

a variable in probability theory with a domain of possible values it can take on

A <u>probability destribution</u> takes a random variable and gives me the probability for each of the possible values in its domain. Example (represented as a vector):

P(flight) = <0.6, 0.3, 0.1>

Example: P (a $\wedge$ b) = P(a)P(b)

3.1.1. Bayes' rules

$$P(a \wedge b) = P(b)\ P(a|b)$$

$$P(a \wedge b) = P(a)\ P(b|a)$$

If both are equal to P(a $\wedge$ b) that must mean that they are equal to each other as well:

$$P(a)\ P(b|a) = P(b)\ P(a|b)$$

Bayes' Rule

$$P(b \mid a) = \frac{P(a \mid b)\ P(b)}{P(a)}$$

Given clouds in the morning,
what's the probability of rain in the afternoon?

- 80% of rainy afternoons start with cloudy mornings.
- 40% of days have cloudy mornings.
- 10% of days have rainy afternoons.

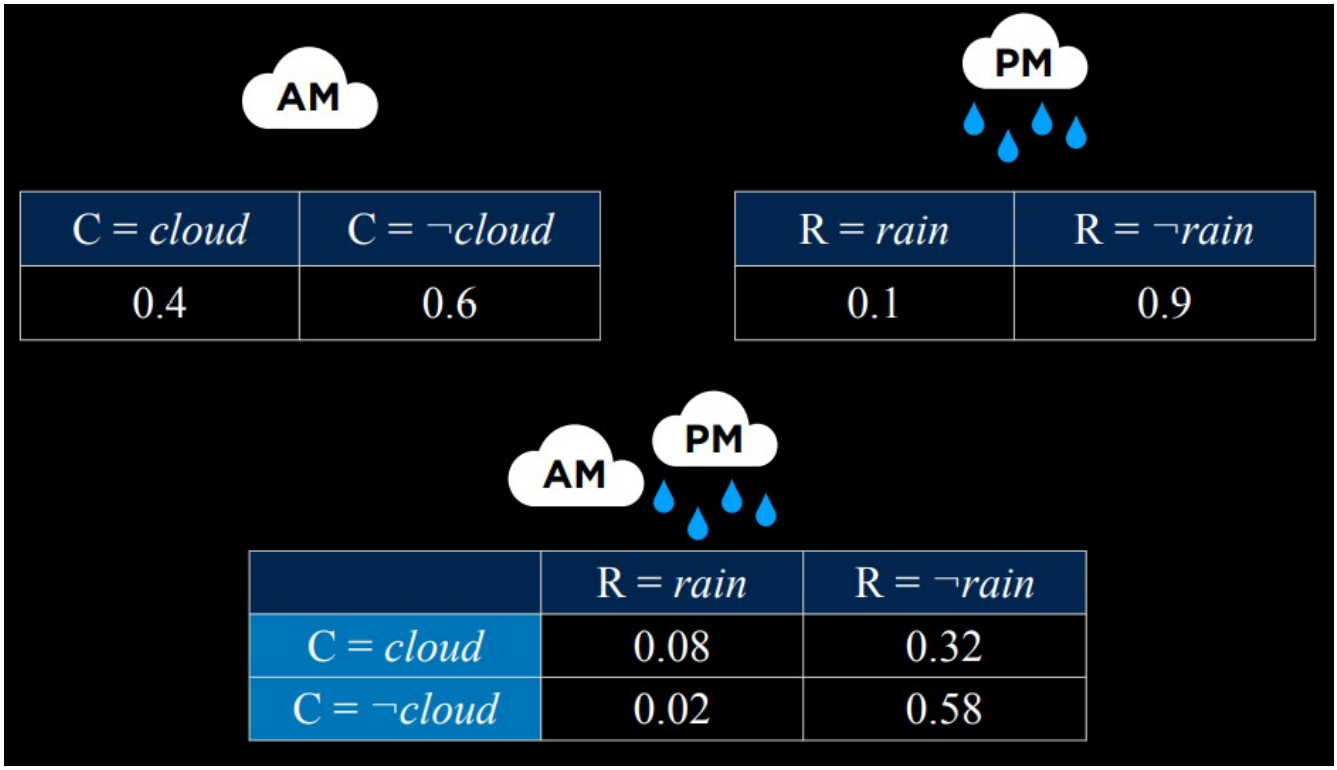
$$P(rain \mid clouds) = \frac{P(clouds \mid rain)P(rain)}{P(clouds)}$$

$$= \frac{(.8)(.1)}{.4}$$

$$= 0.2$$

Knowing P(vissible effect | unknown cause) we can calculate P (unknown cause | vissible effect).

3.1.2. Joint probability



$$\mathbf{P}(C \mid rain)$$

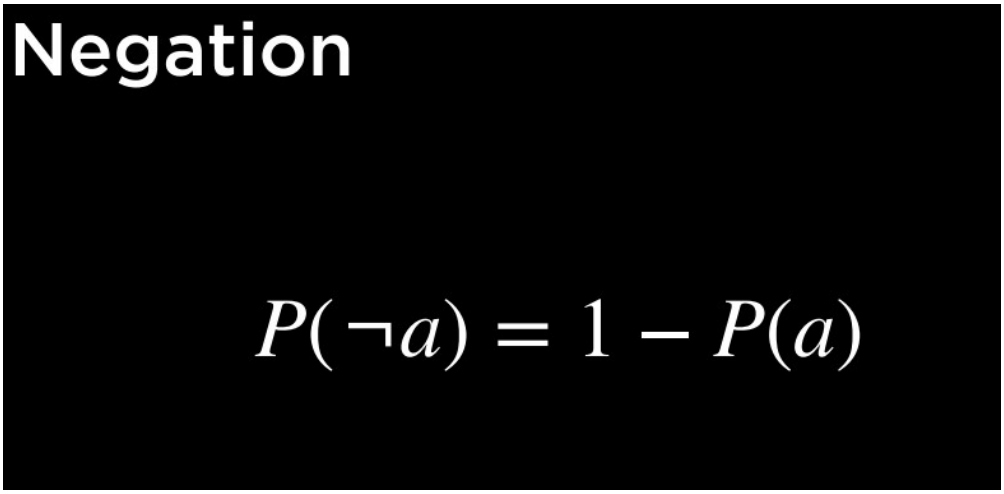
$$\mathbf{P}(C \mid rain) = \frac{\mathbf{P}(C, rain)}{P(rain)} = \alpha \mathbf{P}(C, rain)$$

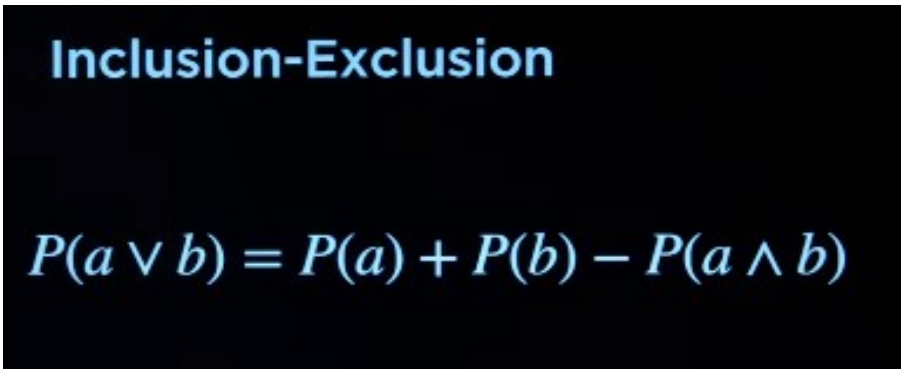
$$= \alpha \langle 0.08, 0.02 \rangle = \langle 0.8, 0.2 \rangle$$

	R = *rain*	R = ¬*rain*
C = *cloud*	0.08	0.32
C = ¬*cloud*	0.02	0.58

The comma also stands for a logical and. Since we need both 0.08 and 0.02 to sum up to 1, the constant must be 10.

3.1.3. Probability rules





Marginalization

$$P(a) = P(a, b) + P(a, \neg b)$$

Marginalization

	R = *rain*	R = ¬*rain*
C = *cloud*	0.08	0.32
C = ¬*cloud*	0.02	0.58

$$P(\text{C} = cloud)$$
$$= P(\text{C} = cloud, R = rain) + P(\text{C} = cloud, R = \neg rain)$$
$$= 0.08 + 0.32$$
$$= 0.40$$

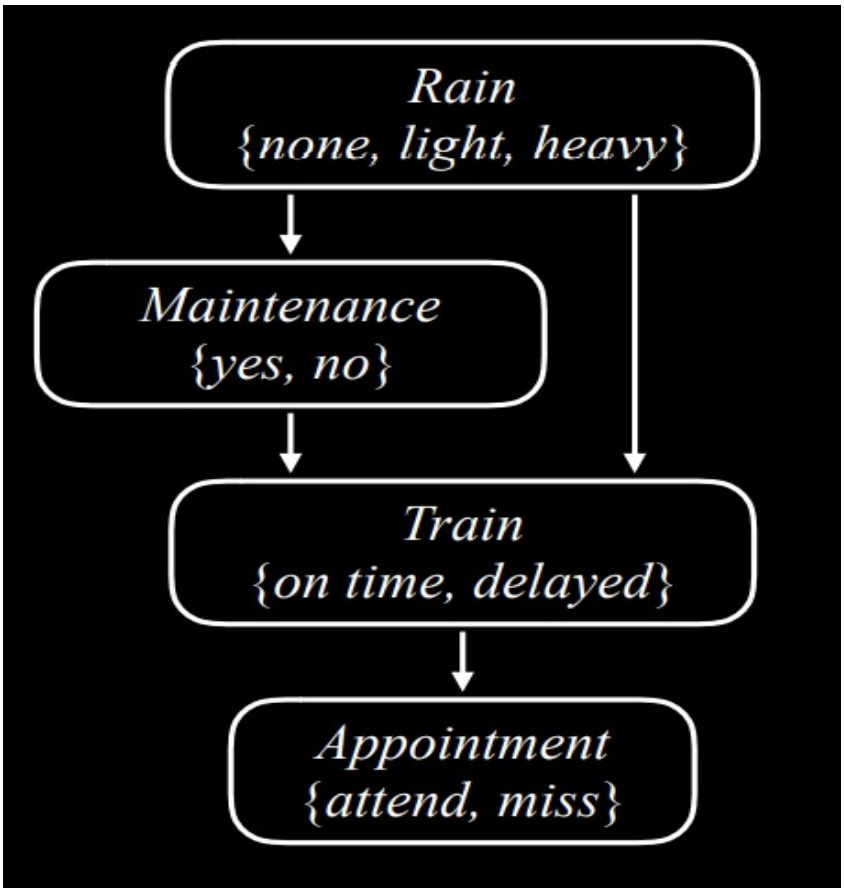
Conditioning

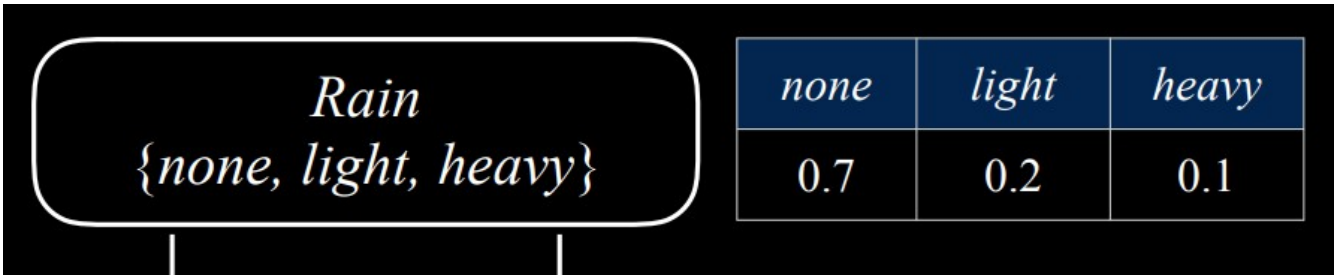
$$P(a) = P(a \mid b)P(b) + P(a \mid \neg b)P(\neg b)$$

3.2. Bayesian Networks

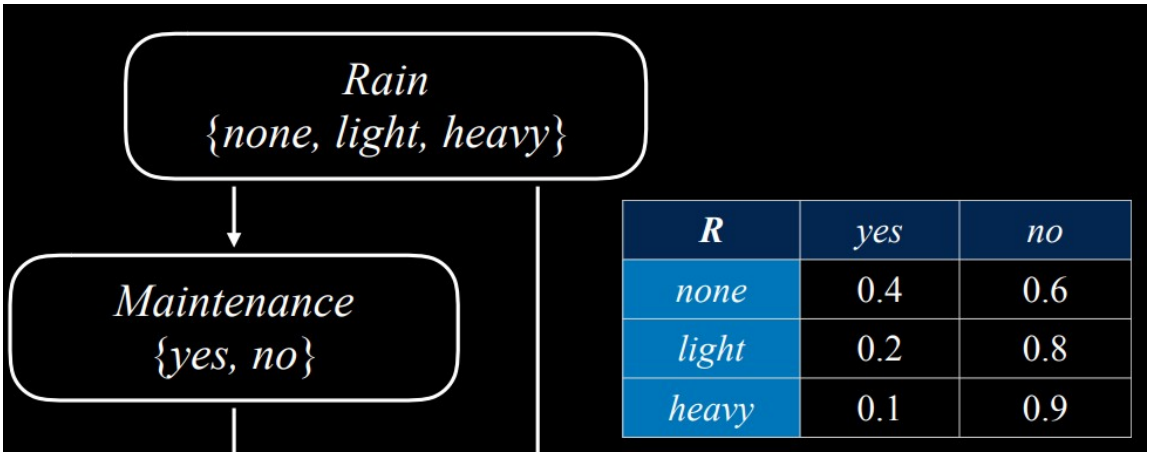
A bayesian network is a data structure that represents the dependencies among random variables.

- directed graph
- each node represents a random variable
- arrow from X to Y means X is a parent of Y
- each node X has probability distribution P(X | Parents(X))

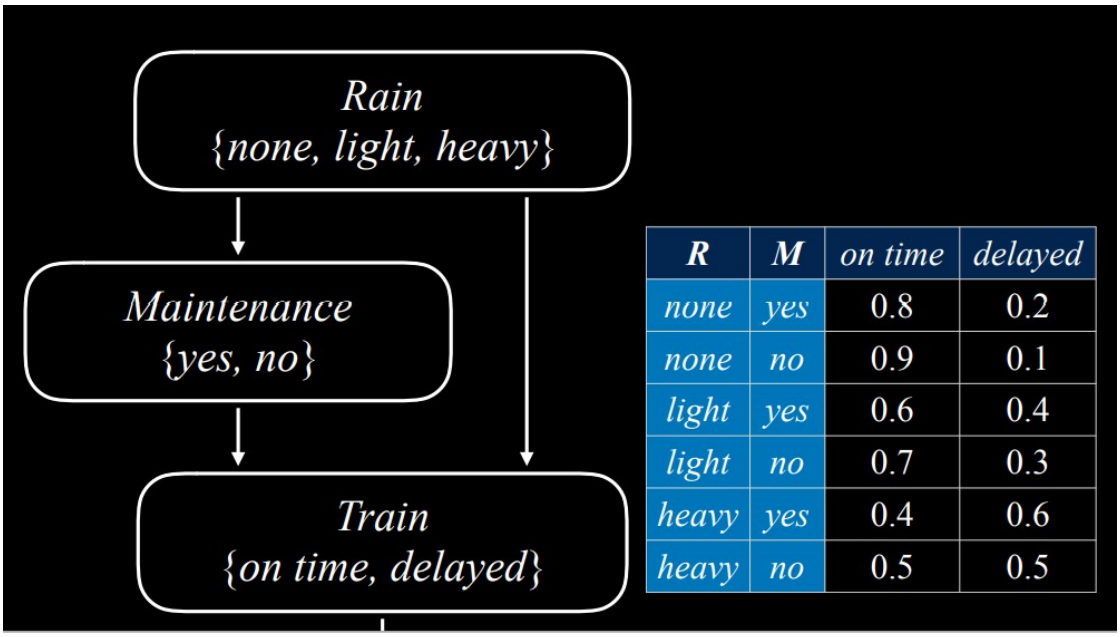




none	light	heavy
0.7	0.2	0.1

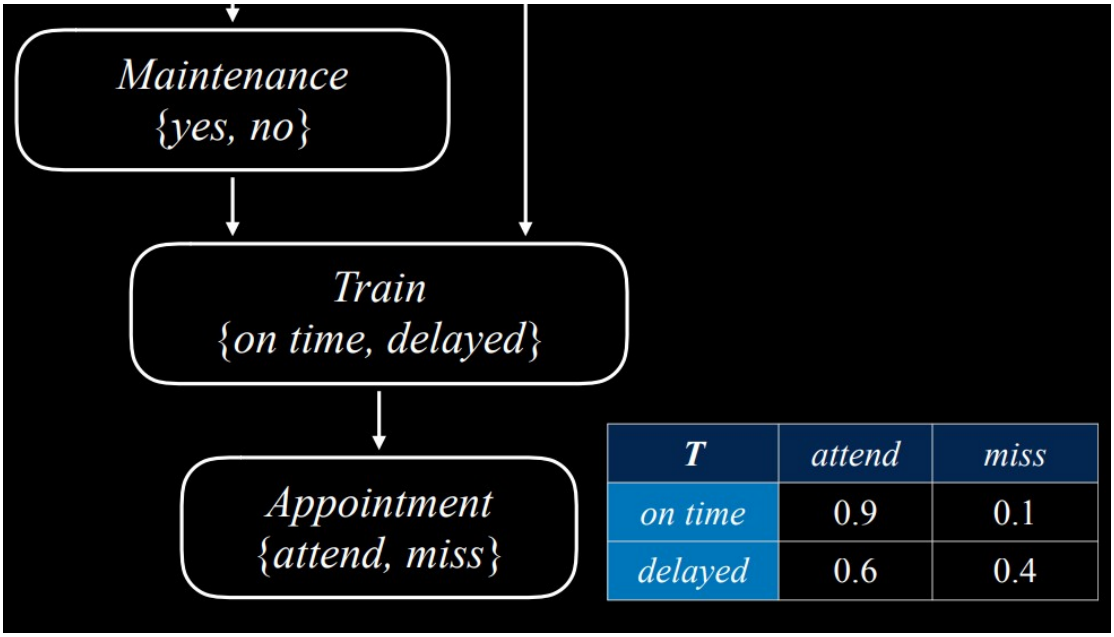


R	yes	no
none	0.4	0.6
light	0.2	0.8
heavy	0.1	0.9

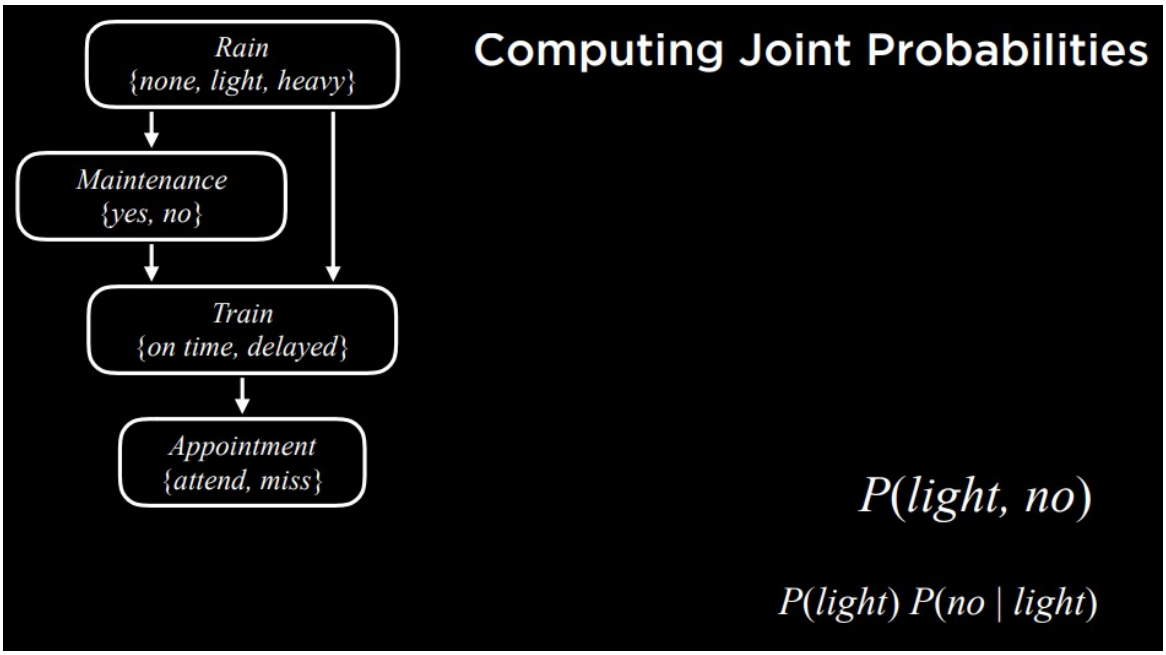


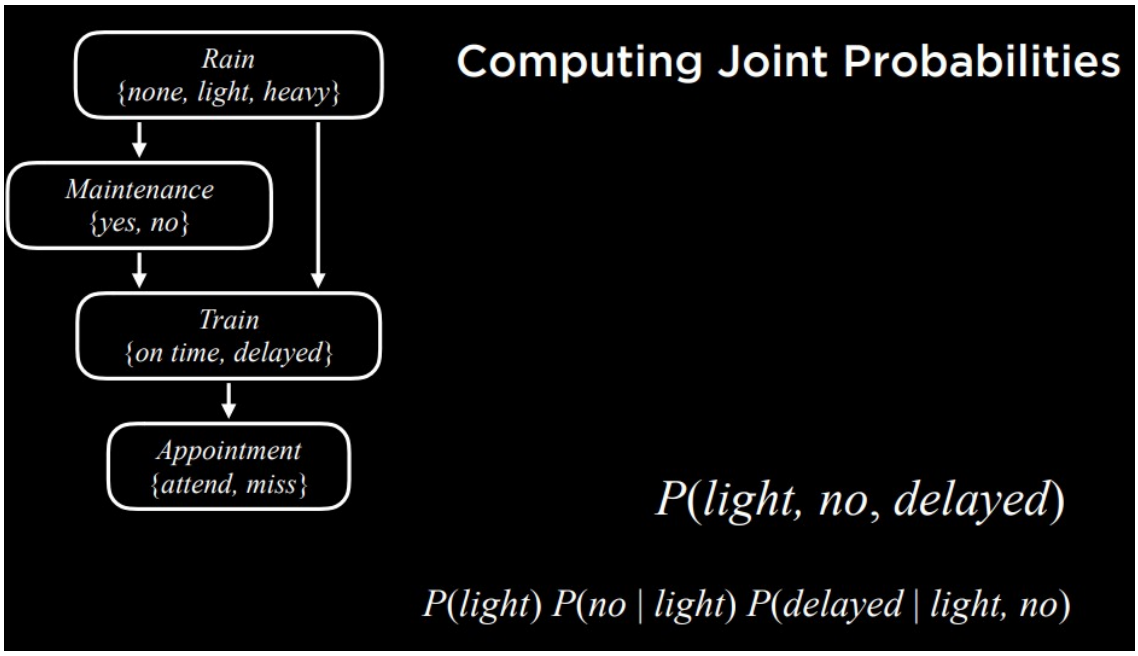
R	M	on time	delayed
none	yes	0.8	0.2
none	no	0.9	0.1
light	yes	0.6	0.4
light	no	0.7	0.3
heavy	yes	0.4	0.6
heavy	no	0.5	0.5

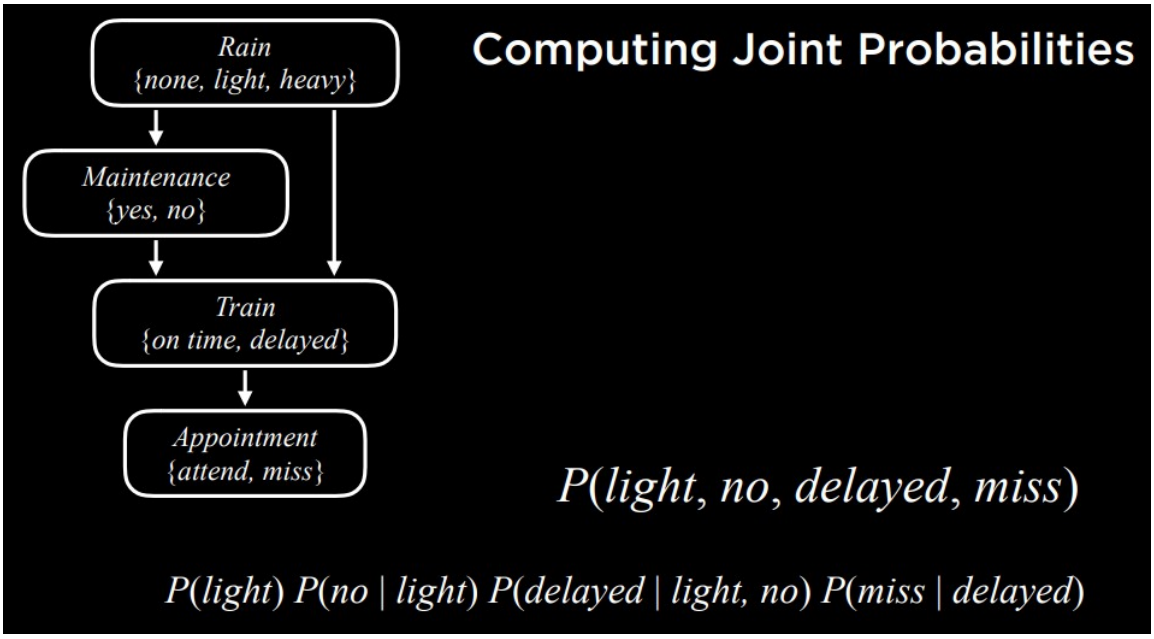
In the final one, there's no need to encode all the other variables, since it will depend exclusively on the train so we represent it like:



T	*attend*	*miss*
on time	0.9	0.1
delayed	0.6	0.4

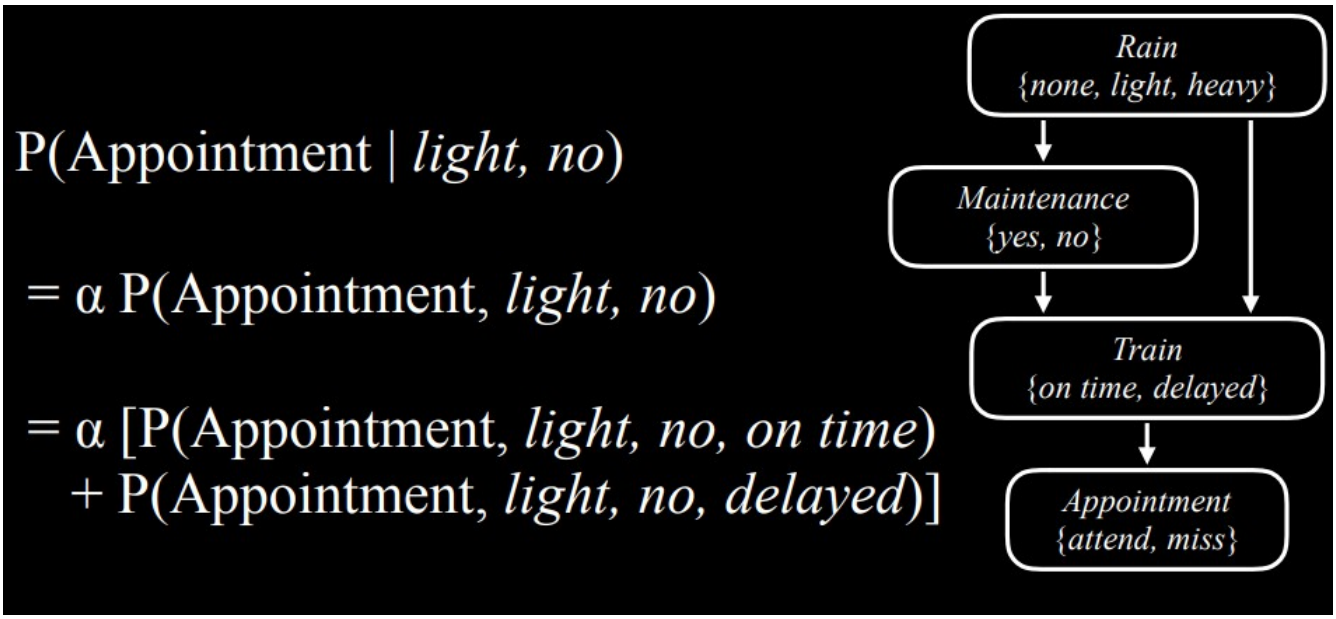



Rain
{none, light, heavy}
Maintenance
{yes, no}
Train
{on time, delayed}
Appointment
{attend, miss}
Computing Joint Probabilities
P(light, no, delayed)
P(light) P(no | light) P(delayed | light, no)

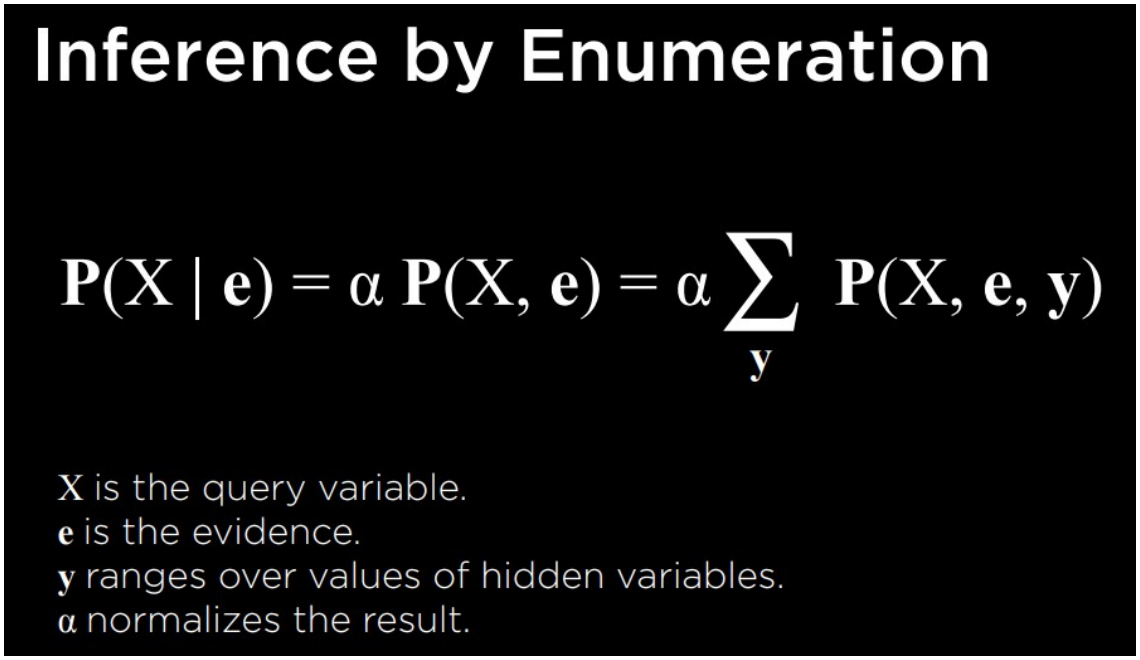

Rain
{none, light, heavy}
Maintenance
{yes, no}
Train
{on time, delayed}
Appointment
{attend, miss}
Computing Joint Probabilities
P(light, no, delayed, miss)
P(light) P(no | light) P(delayed | light, no) P(miss | delayed)

We can also make inferences in a probabilistic setting, and the parts of the inference problem are these:

- Query X: variable for which to compute distribution
- Evidence variables E: observed variables for event e
- Hidden variables Y: non-evidence, non-query variable.
- Goal: Calculate P(X | e)



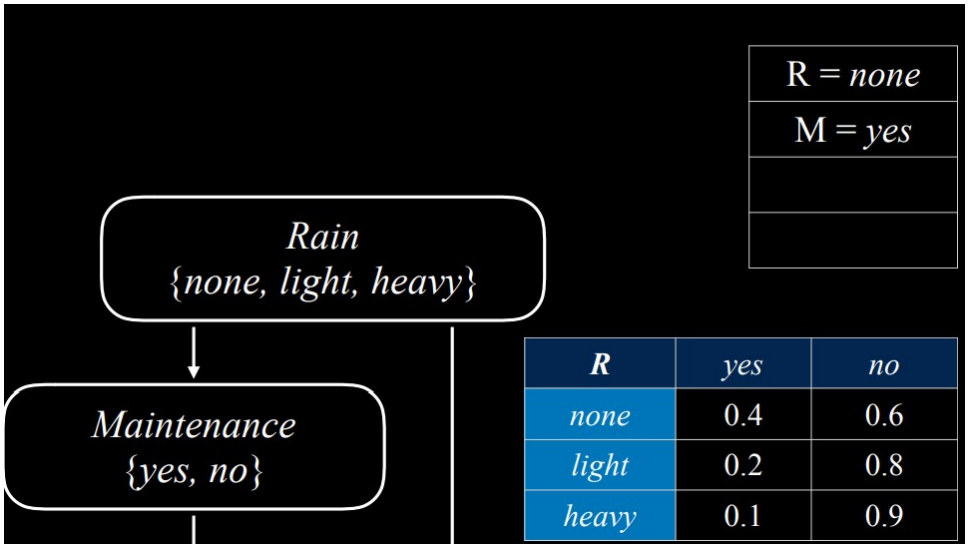
The formula to do inference by enumeration is the following:



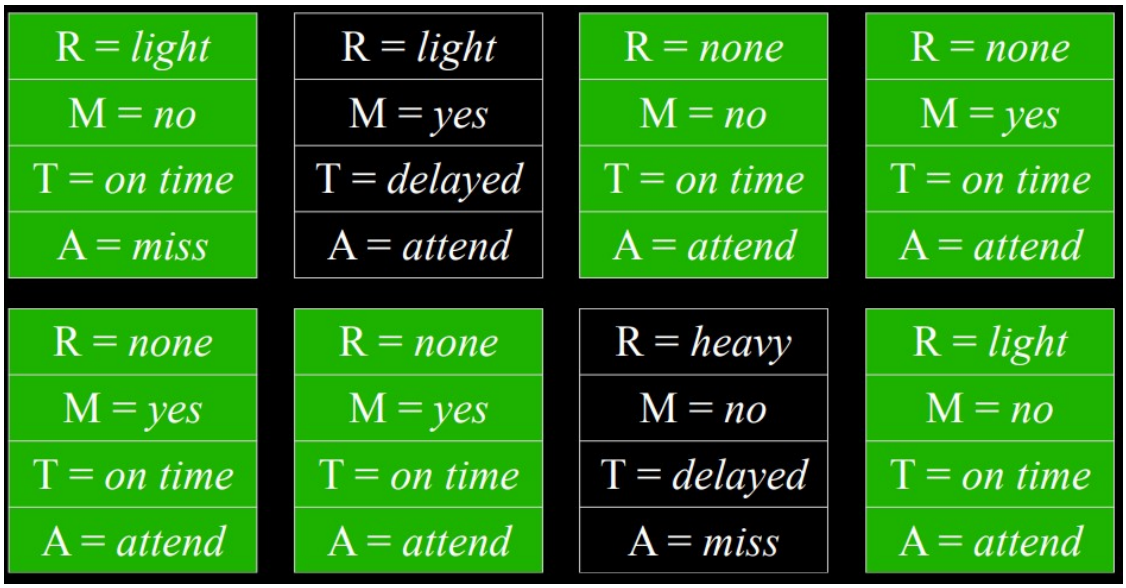
In the first two set of elements we have that a conditional probability is proportional to the joint probability. Some python libraries use these processes to make easier all these formulas, but it's important to know the underlying math.

We can also do Approximate inference. One method is through sampling.

Using the same previous case, we randomly take one of the values of the random variables that a node has, assume we pick randomly none, and since we picked none, we must also choose one of the yes or no's but only those that belong to the first row (none):



This method is useful if we don't need to be 100% right and only with a probability, we can use this method which is more efficient, these are simply the ones in which the train is on time:

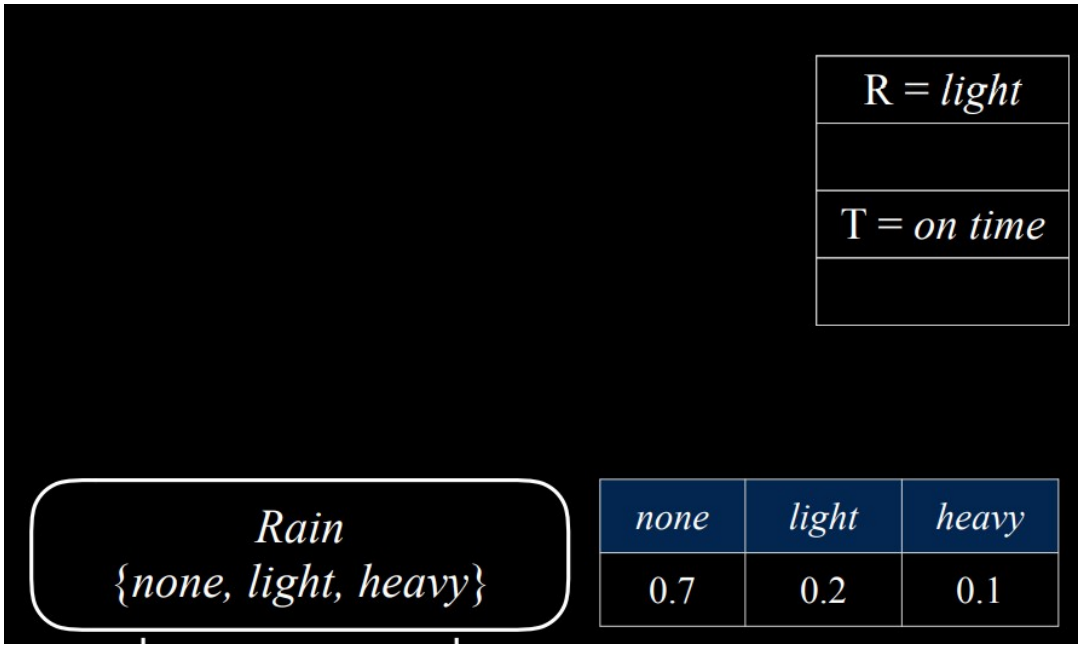


Another way of sampling is called likelihood weighting:

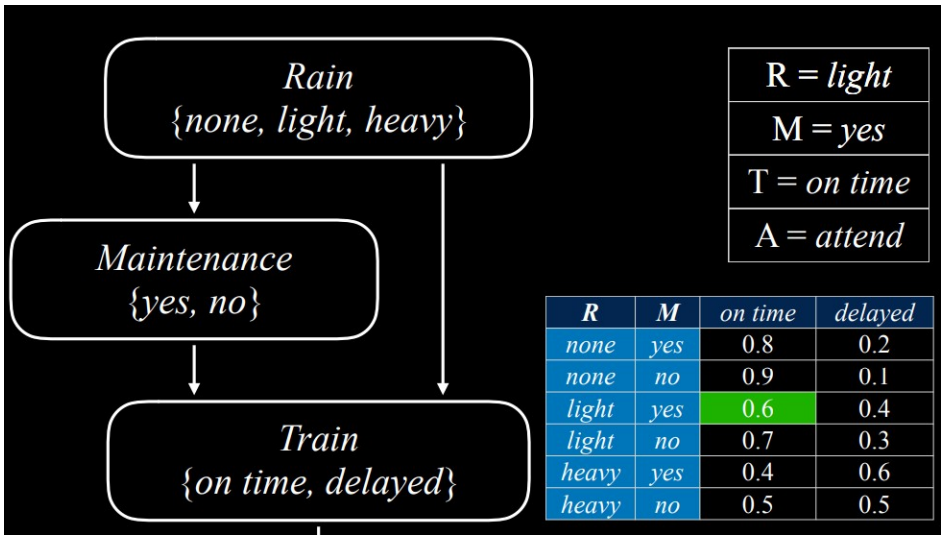
Likelihood Weighting

- Start by fixing the values for evidence variables.
- Sample the non-evidence variables using conditional probabilities in the Bayesian Network.
- Weight each sample by its **likelihood**: the probability of all of the evidence.

Example: P(Rain = light | Train = on time) ?

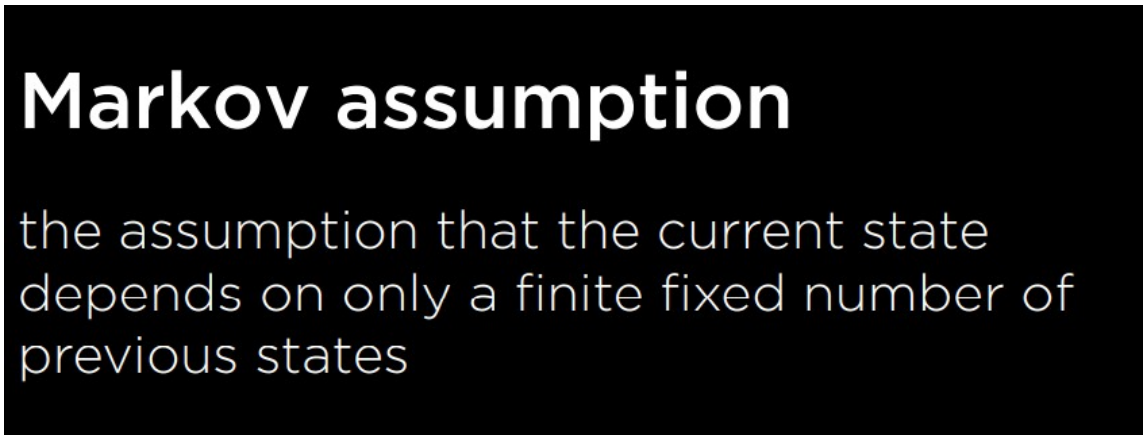


We know that the train is on time, so we don't have to sample that, and now we choose randomly from the other variables:



R	M	on time	delayed
none	yes	0.8	0.2
none	no	0.9	0.1
light	yes	0.6	0.4
light	no	0.7	0.3
heavy	yes	0.4	0.6
heavy	no	0.5	0.5

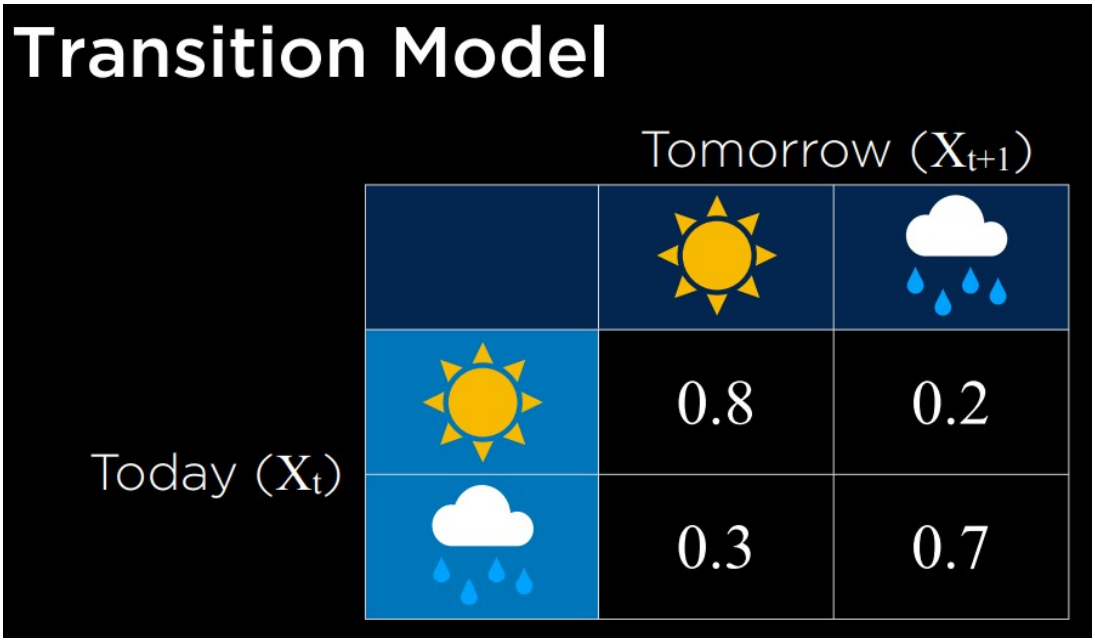
3.3. Markov Models



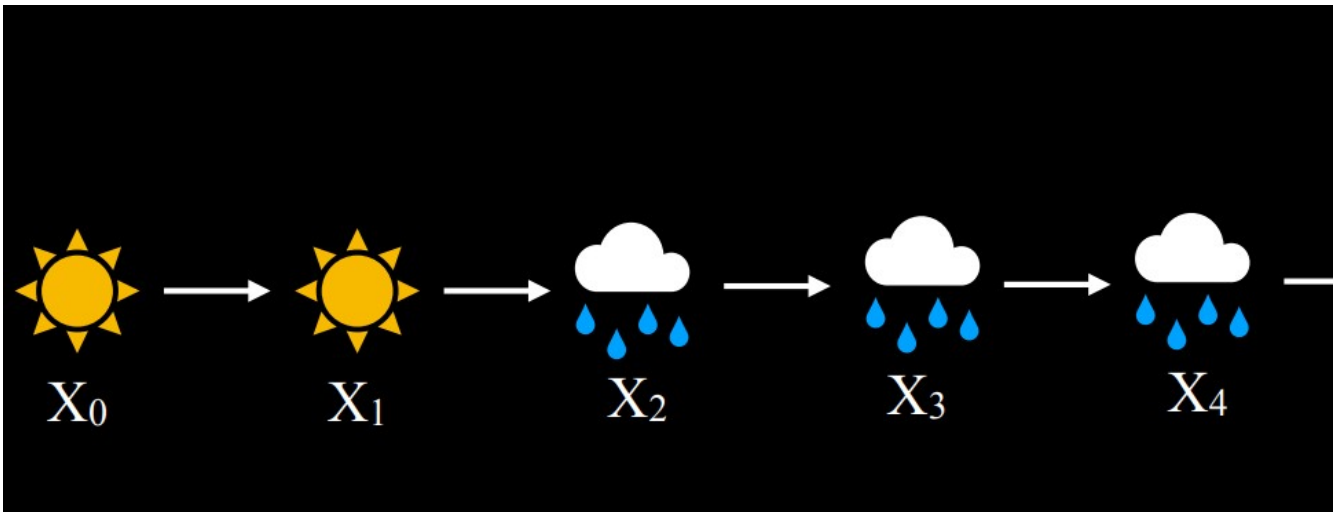
Markov chain

a sequence of random variables where the distribution of each variable follows the Markov assumption

Transition Model

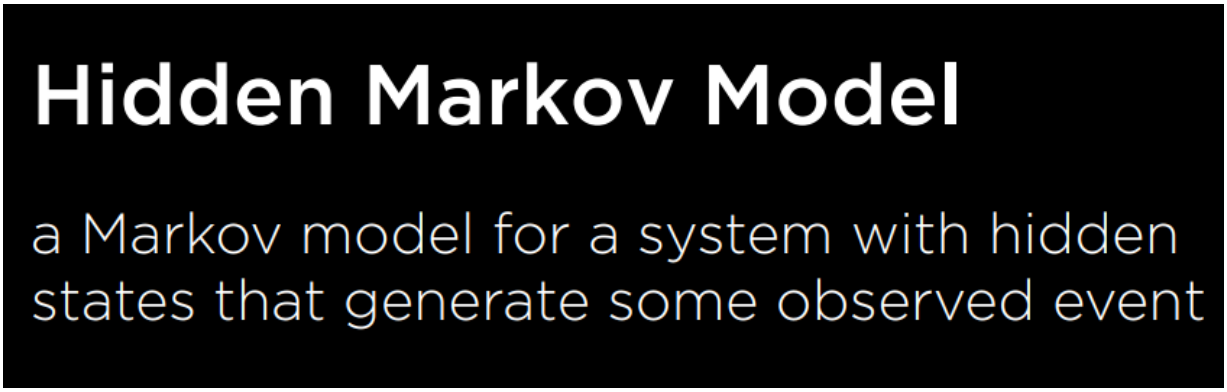


The result of doing this over a extended period of time, is that we can form this markov chain for example:



We move to the sensor models, sensing data, and using that data, that data is somehow related to the state of the world even if it doesn't actually know, our AI doesn't know, what the underlying true state of the world actually is. And that is how we introduce the next topic:

3.4. Hidden Markov Models

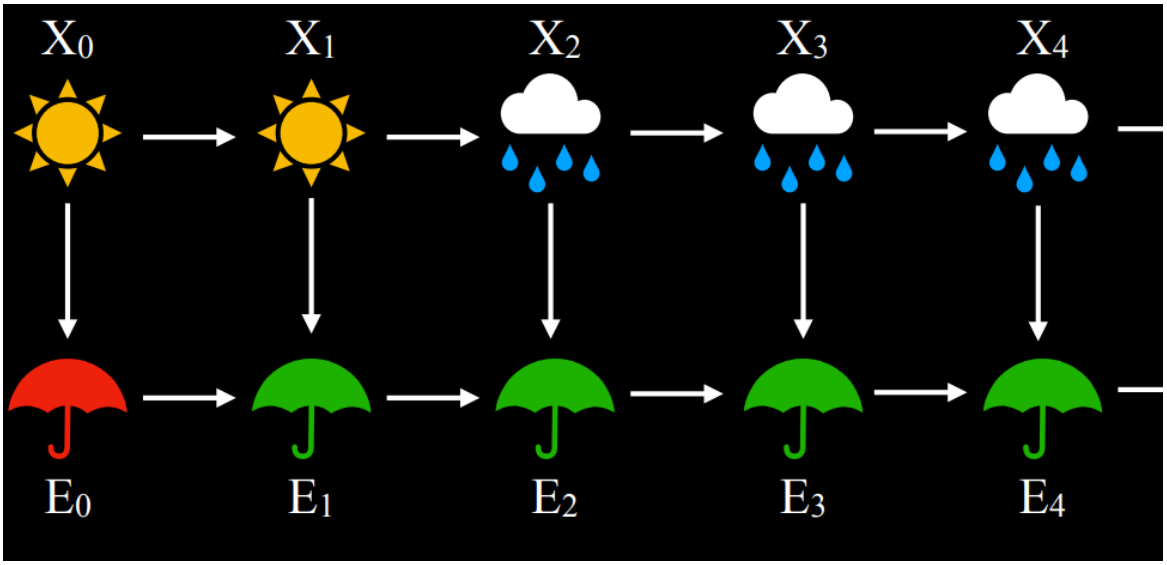


Sensor Model

State (X_t)	Observation (E_t)	
		
	0.2	0.8
	0.9	0.1

sensor Markov assumption

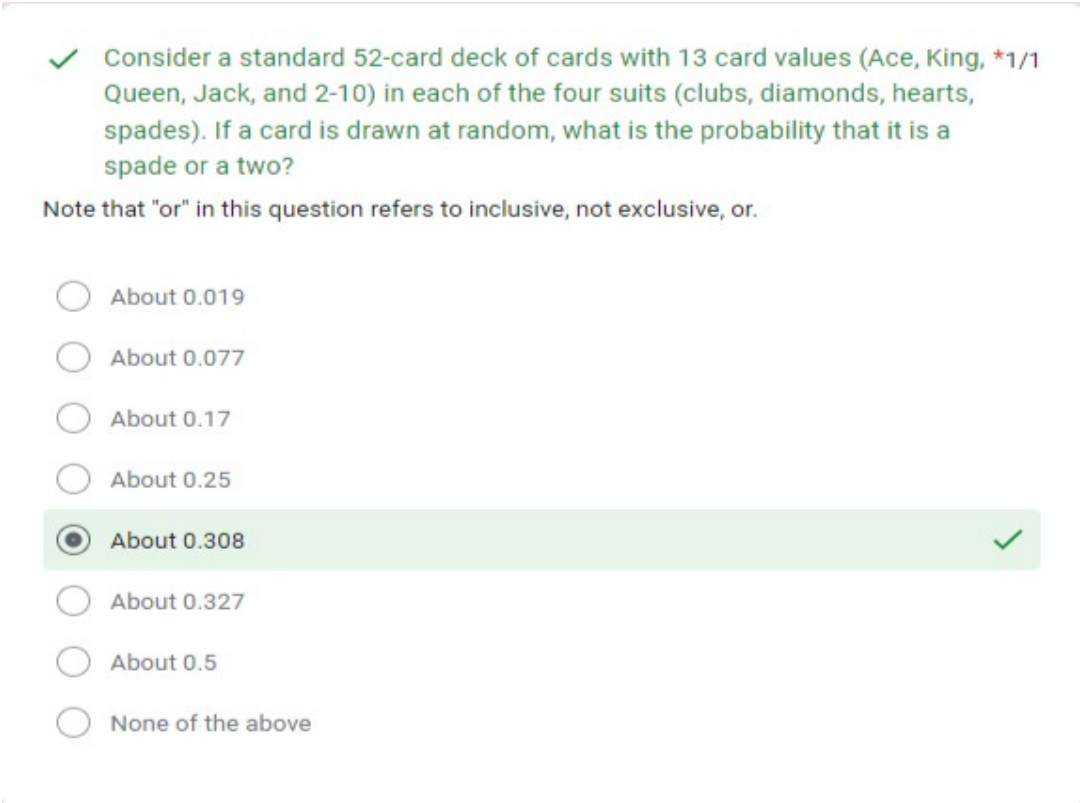
the assumption that the evidence variable depends only the corresponding state



There are a number of possible tasks that you might want to do given this kind of information of Hidden Markov Models:

Task	Definition
filtering	given observations from start until now, calculate distribution for **current** state
prediction	given observations from start until now, calculate distribution for a **future** state
smoothing	given observations from start until now, calculate distribution for **past** state
most likely explanation	given observations from start until now, calculate most likely **sequence** of states

3.5. Quiz



1)- We use inclusion-exclusion rule.

$P(a \text{ or } b) = P(a) + P(b) - P(a \wedge b)$

$P(a \wedge b) = P(a) * P(b)$ (independent events)

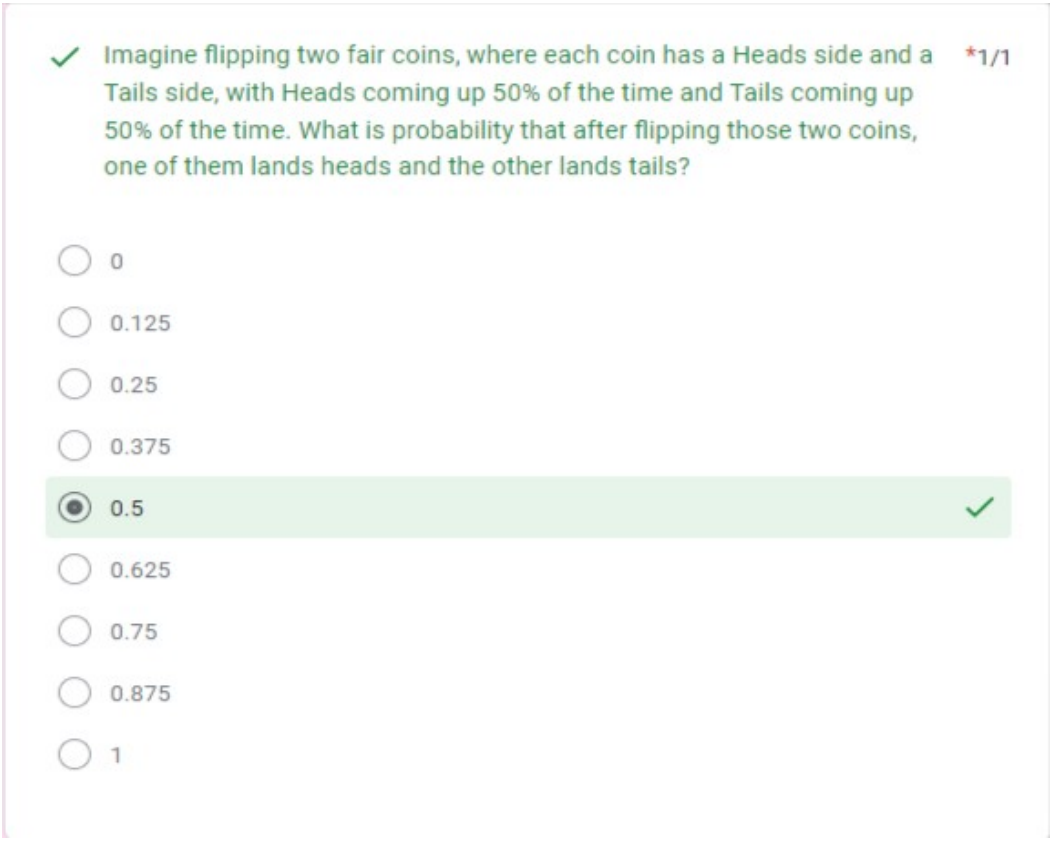
52 cards, divided by 13 possible card values = 4.

Each 2 can appear 4 times then, 4/52. This is the probability that it is a two.

Now the probability that it is spades, first we calculate the number and

probability of spades cards: 52/4 = 13 , 13/52 = 0.25, and we calculate a and b,

P (a $\wedge$ b) = (4/52) * (13/52) = 0.0192, so: P(a or b) = (4/52) + (13/52) - 0.0192 = 0.3077 ≈ 0.308.



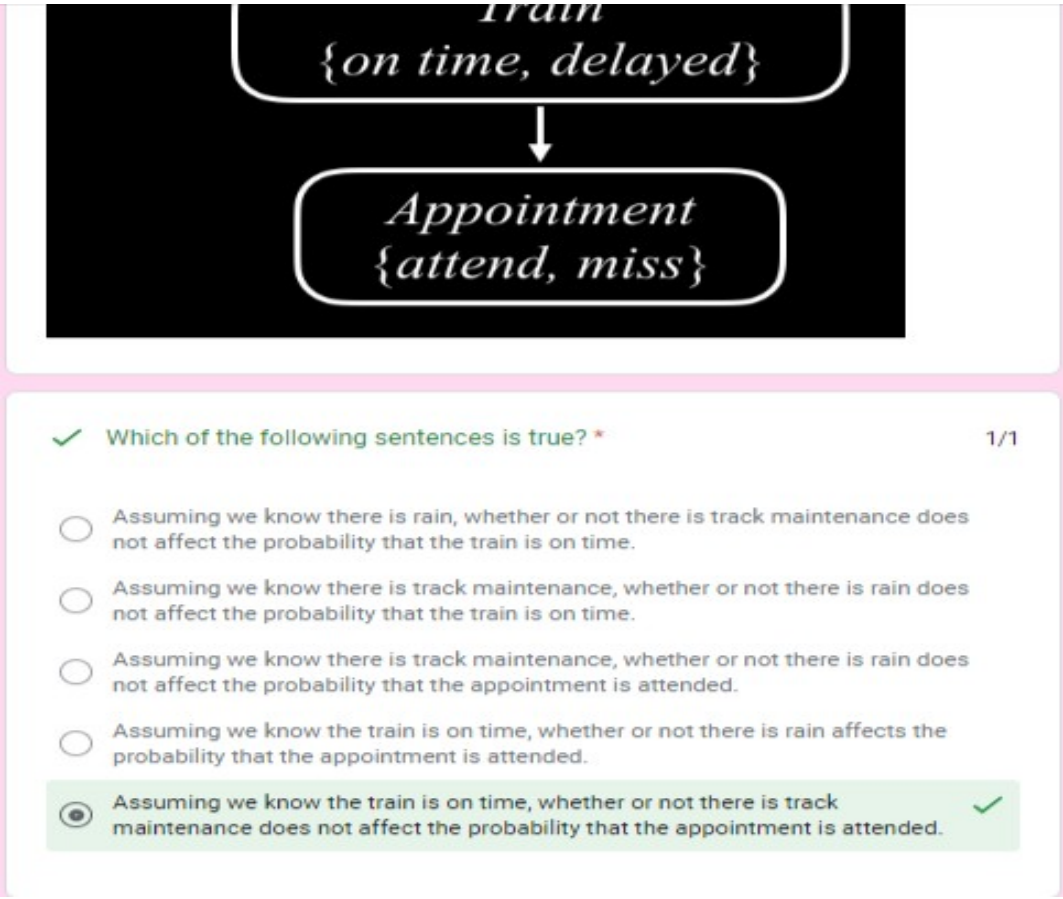
P(heads, tails)

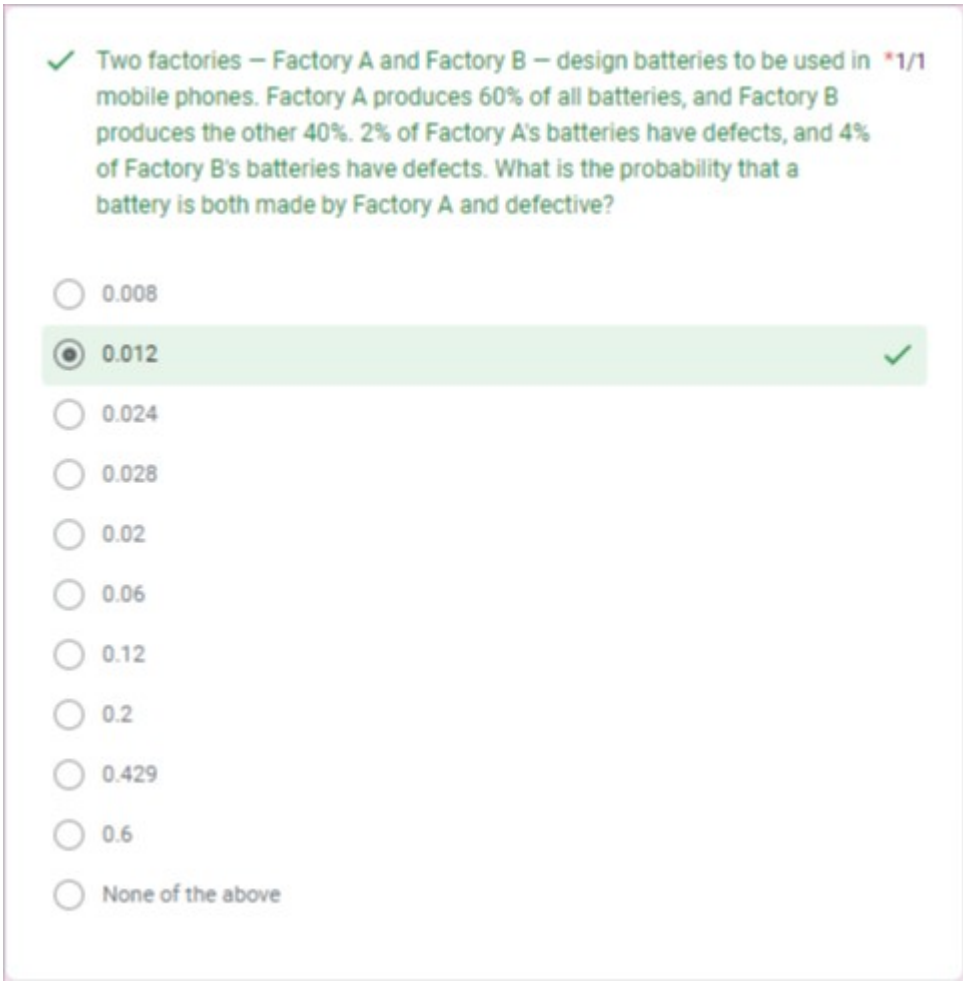
heads heads

heads tails

tails heads

tails tails





We use $P(a \wedge b) = P(a) * P(b \mid a)$ (made by factory A and defective)

Batteries factory A = 0.60, factory B = 0.40, percentage of defectual batteries A: 0.02, percentage of defectual batteries B: 0.04.

$P(a \wedge b) = 0.60 * 0.02 = 0.012$

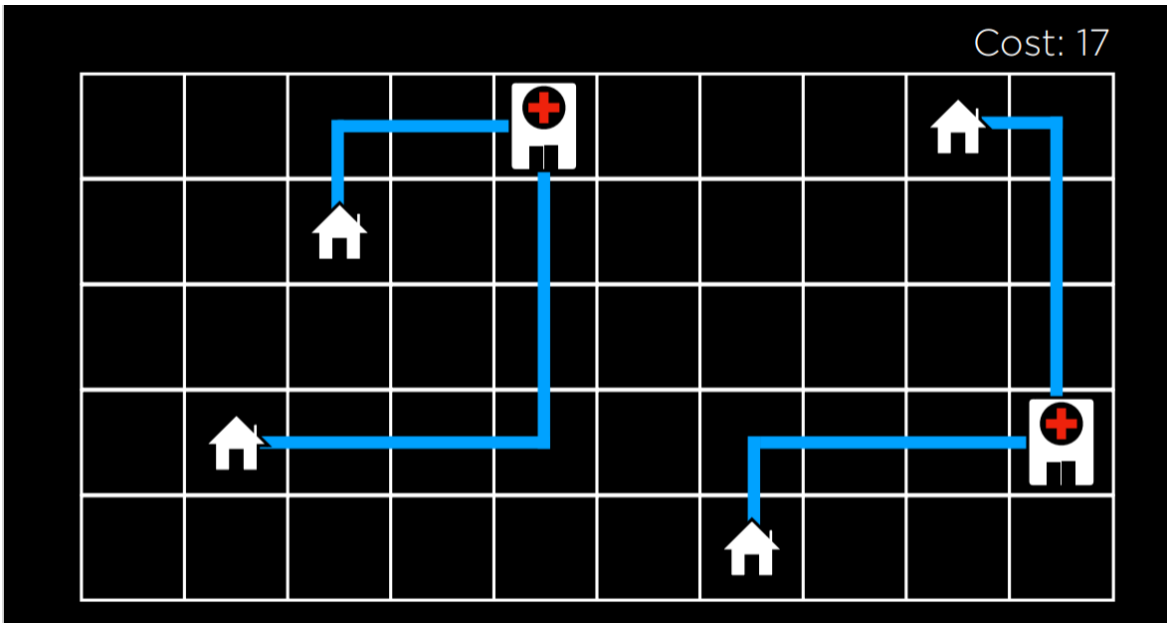
4. Optimization

Optimization is about choosing the best option from a set of options. Let's start looking at algorithms used for optimization.

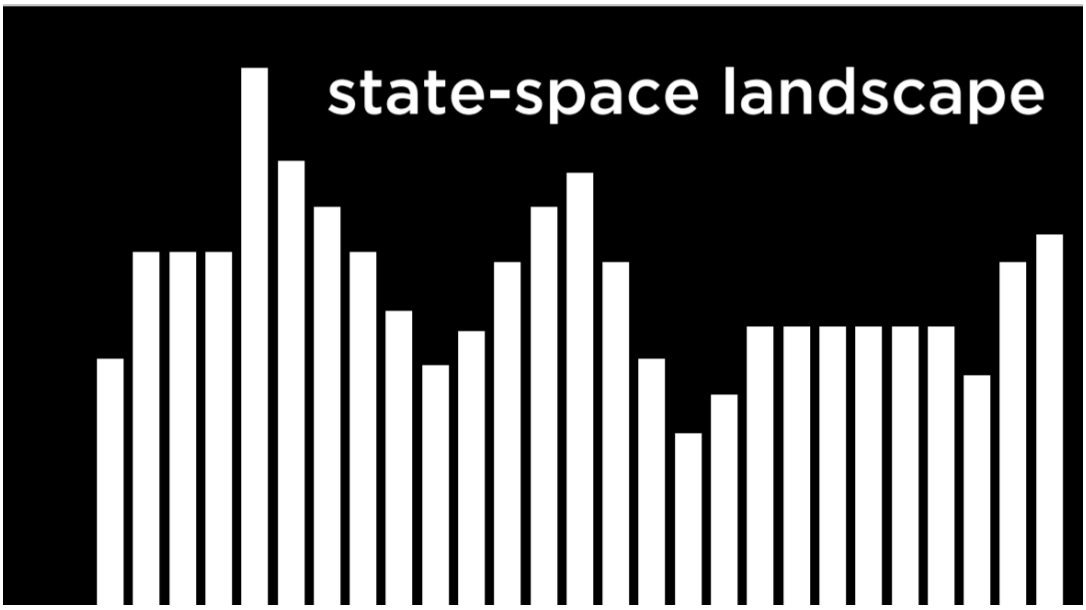
4.1. Local Search

Search algorithms that maintain a single node and searches by moving to a neighboring node. The difference between these type of algorithms and the other types of searches is that in the other types we follow some path to go from the initial state to the goal, in this one we don't really care about the path at all, just to know what the solution is.

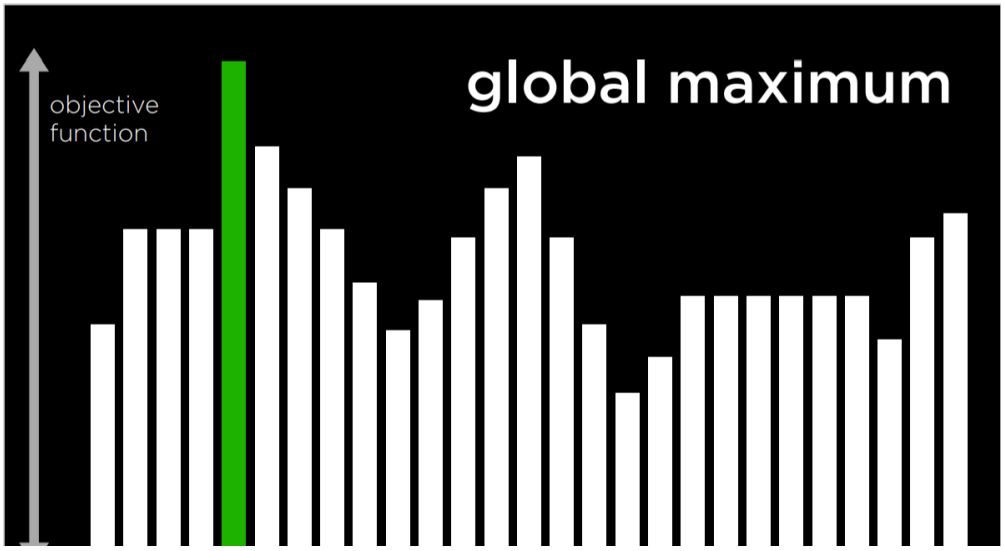
For example, to find the point where the two hospitals would be the closest to the houses, reducing the "cost" (Manhattan distance):



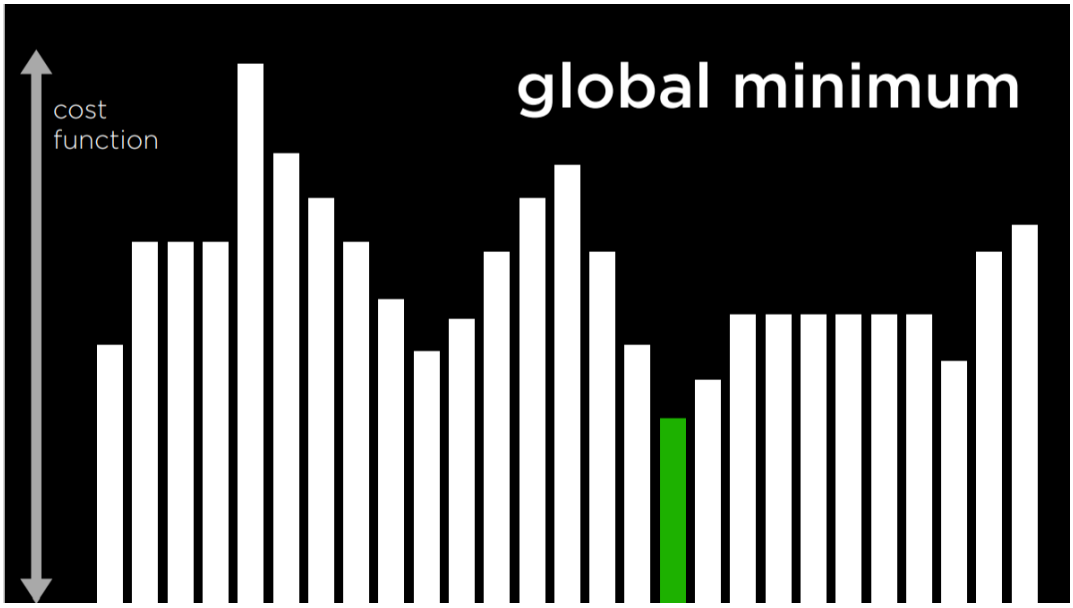
We can think of these problem as a state-space landscape, each of the bars represents a state that our world could be in, the height of these bars is going to represent some value of that state:



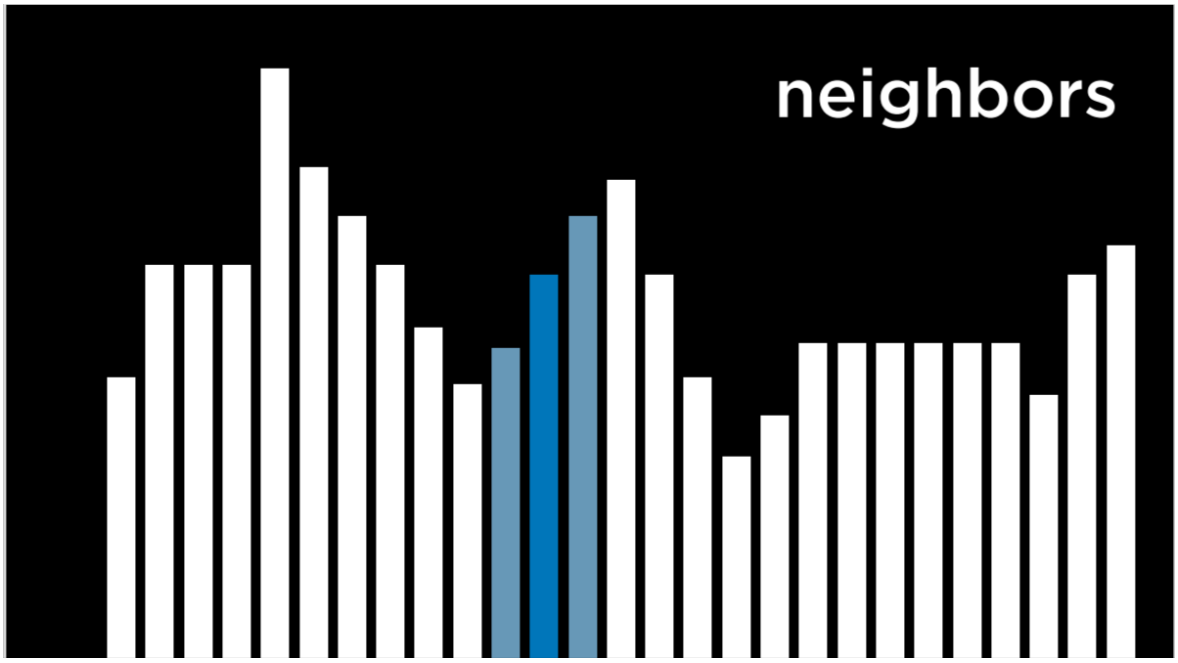
When we're working with a state-space landscape we might want to either find the global maximum (we call the function that we want to optimize in this case: the objective function):



Or viceversa:

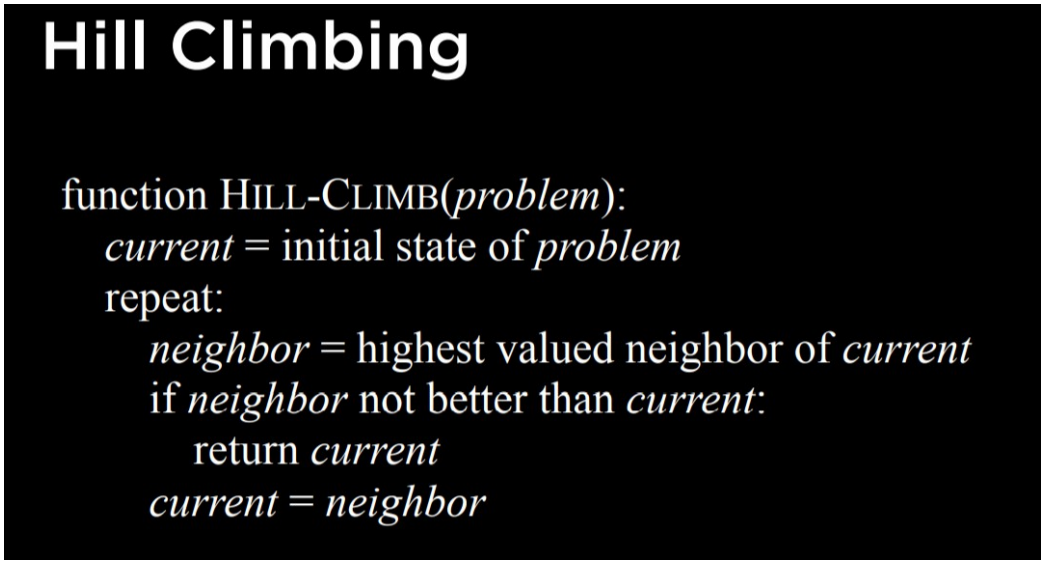


And our strategy to find the global maximum or global minimum will be to take a state mantaining some current node and look at each of its neighboring states:

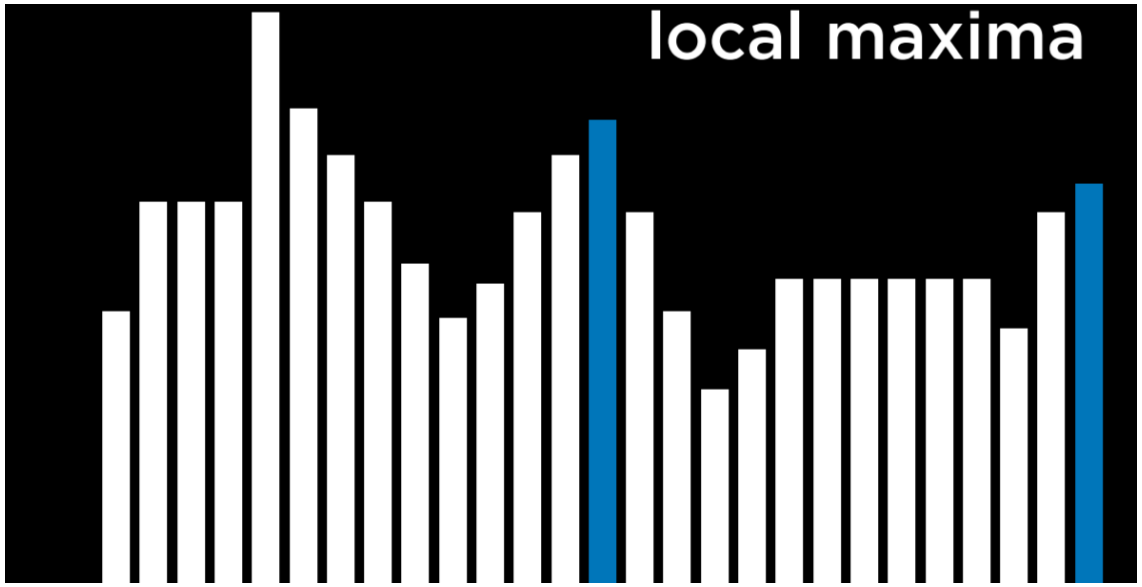


4.1.1. Hill Climbing

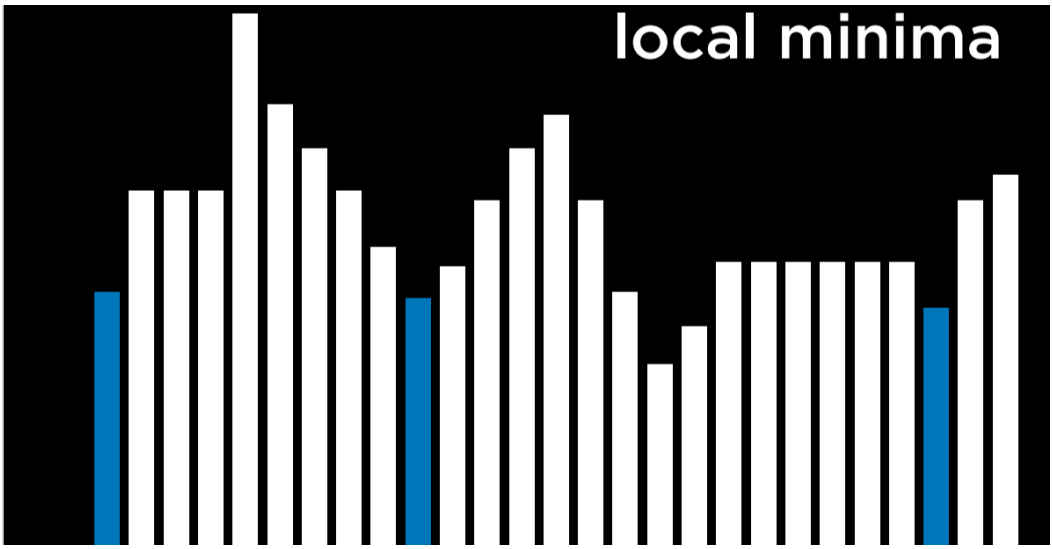
This is one of the algorithms that we can use to implement local search. And this is a function that demonstrates how it works (in this case to find the global maximum, in the case of global minimum you can substitute “highest” with “lowest”):

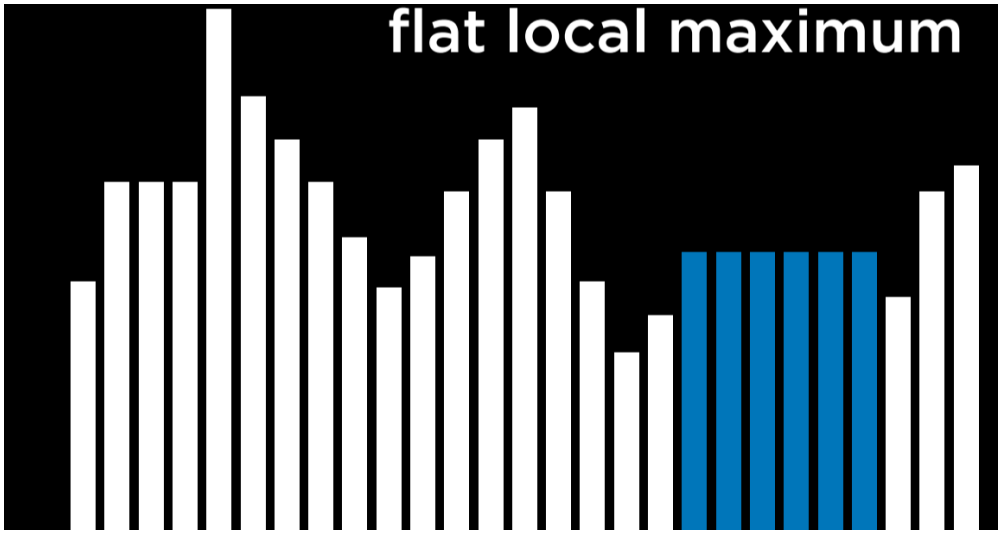


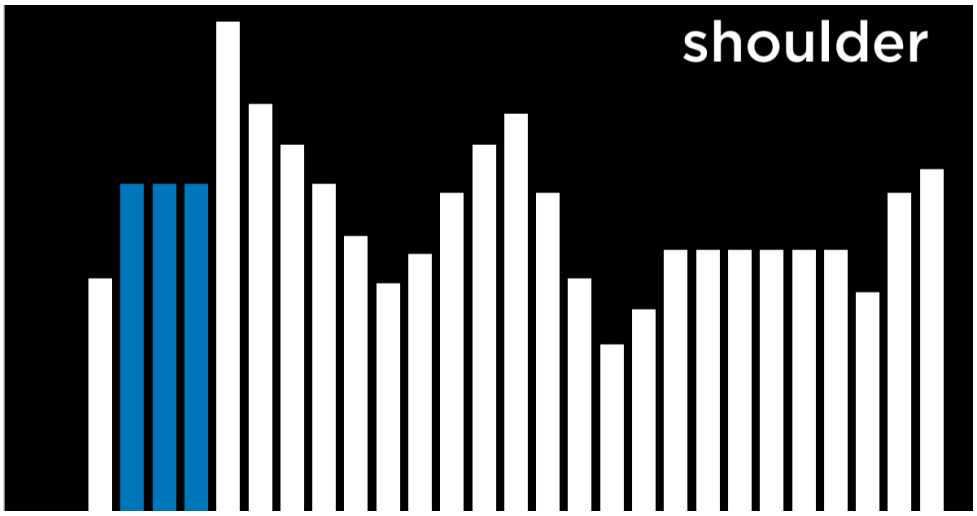
One of the limitations of this algorithms is the possibility of getting stuck in a “local maxima” for example:



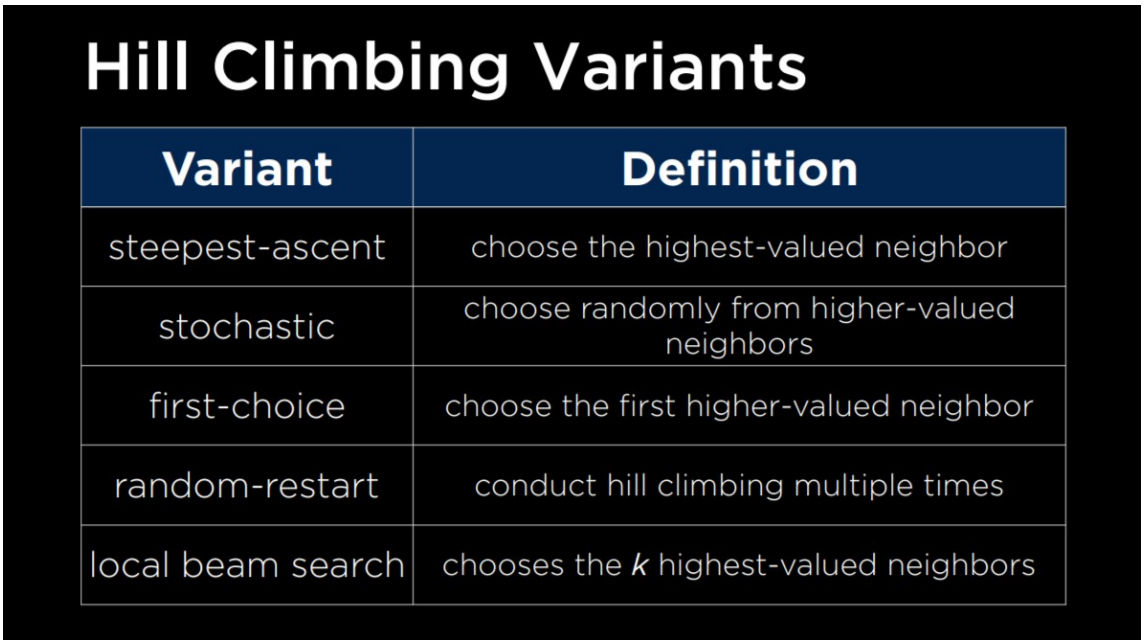
or:







There are also a number of variants of hill climbing algorithms, these are some of them:



4.1.1.1. Simulated Annealing

Annealing is the process of heating metal and allowing it to cool slowly, which serves to toughen the metal. This is used as a metaphor for the simulated annealing algorithm, which starts with a high temperature, being more likely to make random decisions, and, as the temperature decreases, it becomes less likely to make random decisions, becoming more "firm." This mechanism allows the algorithm to change its state to a neighbor that's worse than the current state, which is how it can escape from local maxima.

Simulated Annealing

- Early on, higher "temperature": more likely to accept neighbors that are worse than current state
- Later on, lower "temperature": less likely to accept neighbors that are worse than current state

Simulated Annealing

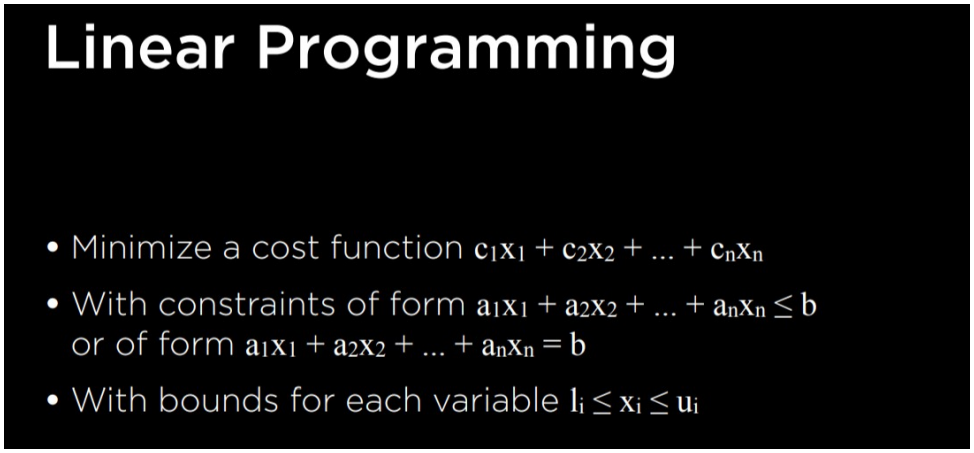
```
function SIMULATED-ANNEALING(problem, max):
    current = initial state of problem
    for t = 1 to max:
        T = TEMPERATURE(t)
        neighbor = random neighbor of current
        ΔE = how much better neighbor is than current
        if ΔE > 0:
            current = neighbor
        with probability e^(ΔE/T) set current = neighbor
    return current
```

Refer to the lecture to learn about the traveling salesman problem, which can offer a good solution used simulated annealing.

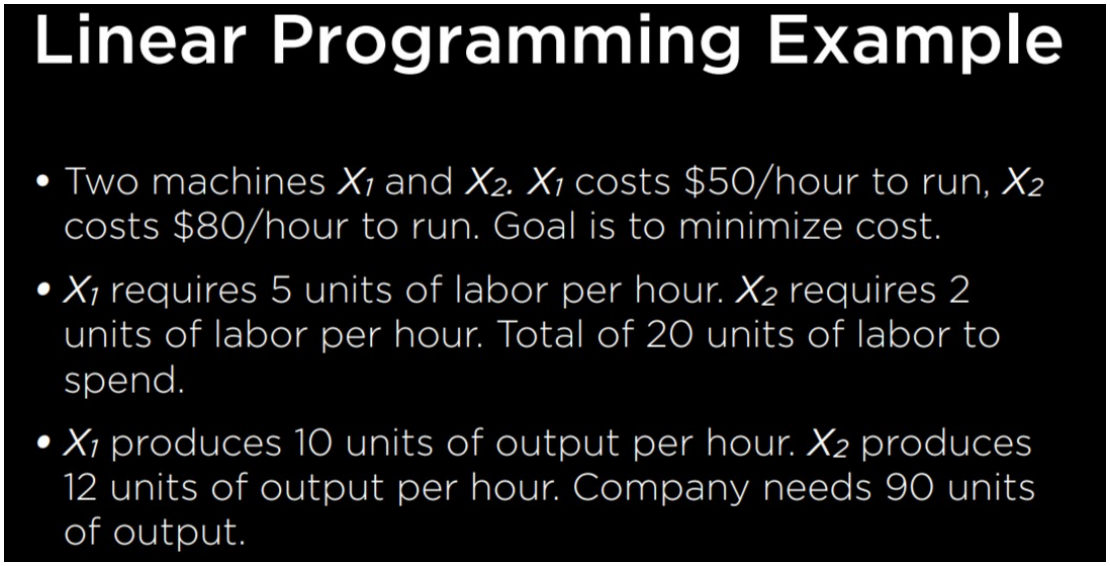
4.2. Linear Programming

Linear programming is a family of problems that optimize a linear equation (an equation of the form $y = ax_1 + bx_2 + \ldots$).

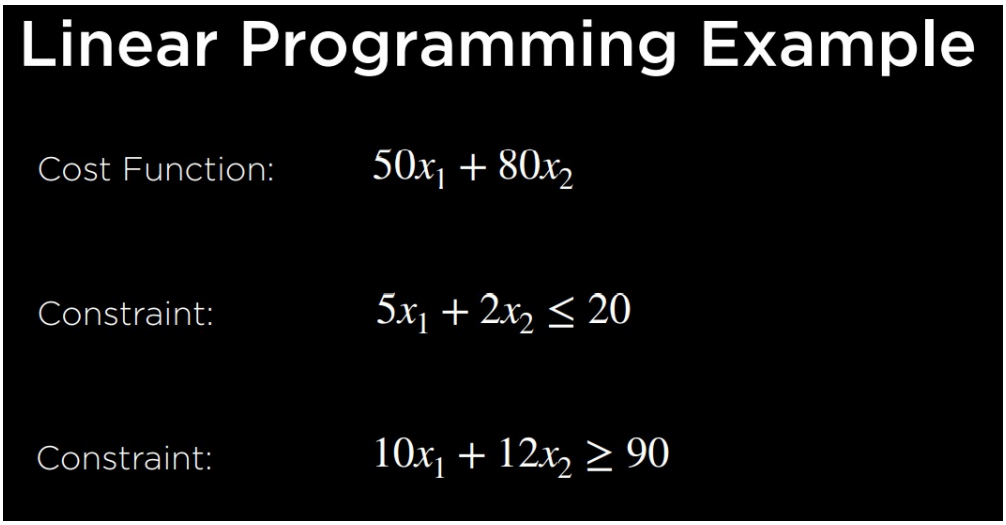
Linear programming will have the following components:



Let's look at an example:



We can translate that to linear equations:



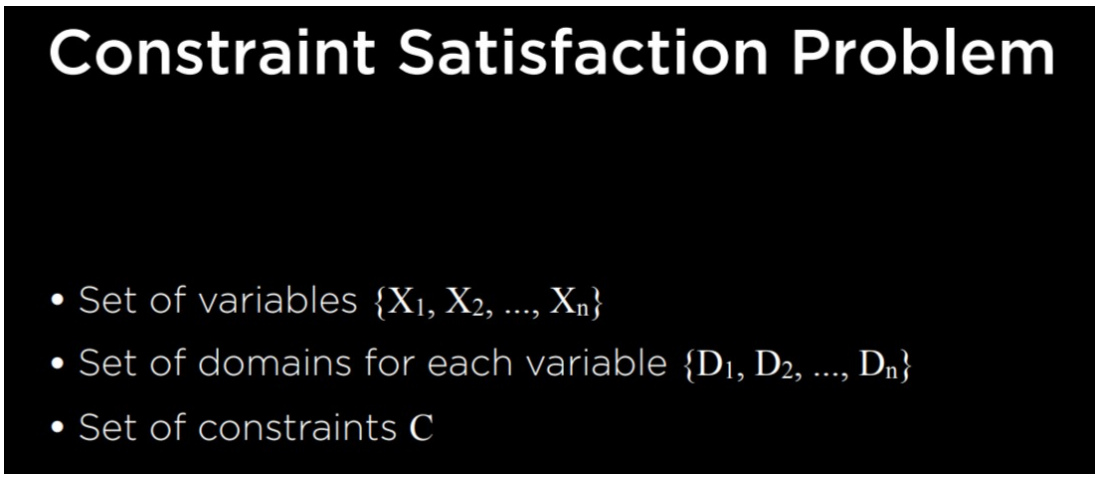
However in the last constraint we see that it is higher than or equal to; constraints need to be of the form ($a_1x_1 + a_2x_2 + \ldots + a_nx_n \leq b$) or ($a_1x_1 + a_2x_2 + \ldots + a_nx_n = b$). Therefore, we multiply by (-1) to get to an equivalent equation of the desired form: ($-10x_1$) + ($-12x_2$) ≤ -90.

There are two types of linear programming algorithms, *Simplex* or *Interior Point.* You can read more in depth these methods in your linear algebra book Jordi.

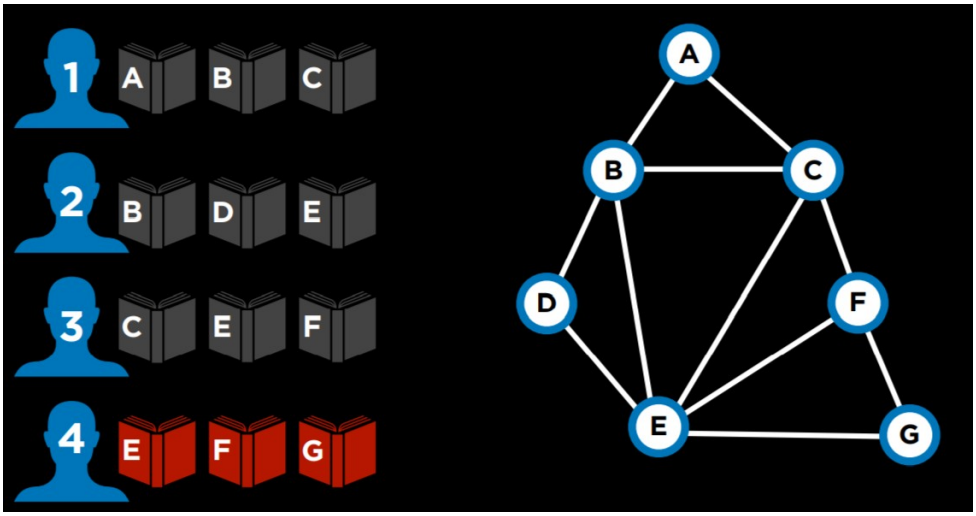
4.3. Constraint Satisfaction

Constraint Satisfaction problems are a class of problems where variables need to be assigned values while satisfying some conditions.

Constraint satisfaction problems have these components:



The example of the lecture, students and exams remember (in this case we use constraints that are represented as a graph. Each node on the graph is a course, and an edge is drawn between two courses if they can't be scheduled on the same day.):



And some more terms to work with constraint satisfaction problems:

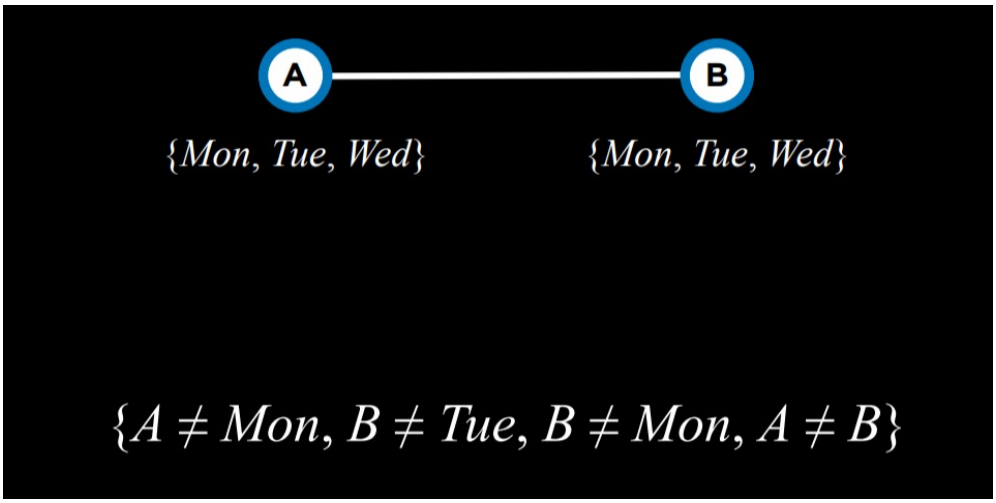
- A ***Hard Constraint*** is a constraint that must be satisfied in a correct solution.
- A ***Soft Constraint*** is a constraint that expresses which solution is preferred over others.
- A ***Unary Constraint*** is a constraint that involves only one variable. In our example, a unary constraint would be saying that course A can't have an exam on Monday {A ≠ Monday}.

•A ***Binary Constraint*** is a constraint that involves two variables. This is the type of constraint that we used in the example above, saying that some two courses can't have the same value {A ≠ B}.

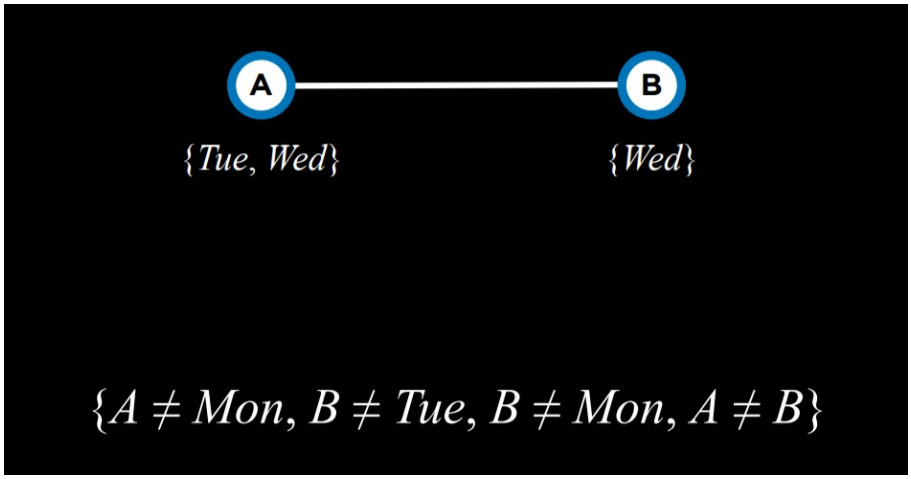
4.3.1 Node Consistency

Node consistency is when all the values in a variable's domain satisfy the variable's unary constraints.

For example:



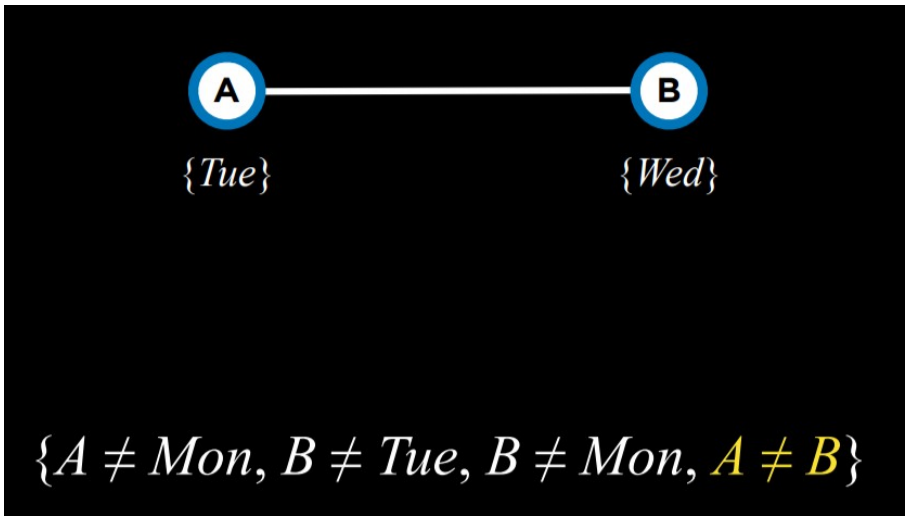
To achieve node consistency we remove all the unary constraints (not the last one because is a binary constraint):



4.3.2. Arc Consistency

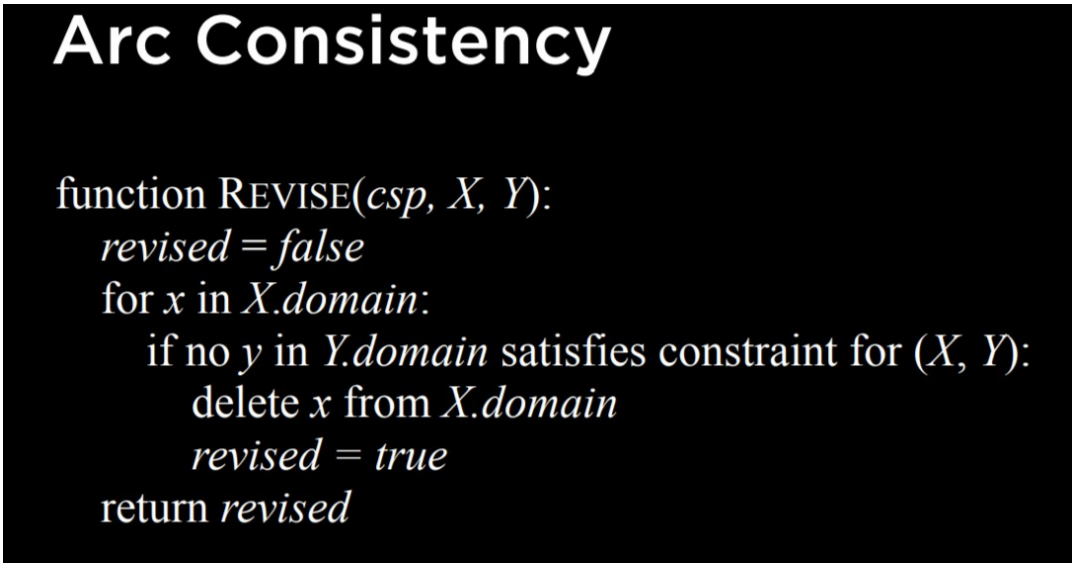
Arc consistency is when all the values in a variable's domain satisfy the variable's binary constraints (note that we are now using "arc" to refer to what we previously referred to as "edge"). In other words, to make X arc-consistent with respect to Y, remove elements from X's domain until every choice for X has a possible choice for Y.

Considering the example from before:

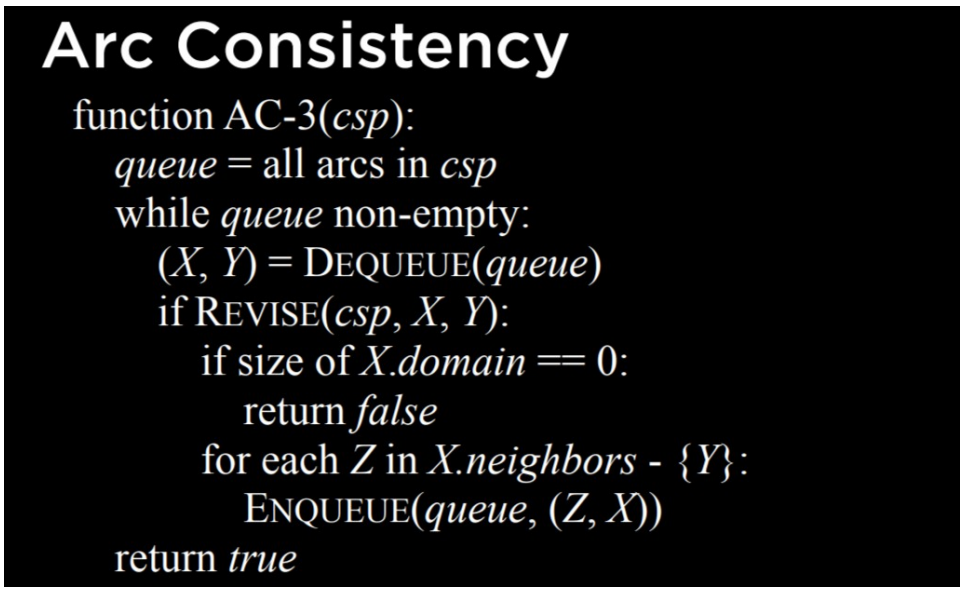


If A takes the value Tuesday, then B can take the value Wednesday. However, if A takes the value Wednesday (like it could in the previous image), then there is no value that B can take. Therefore, A is not arc-consistent with B. To change this, we can remove Wednesday from A's domain.

Here's a pseudocode function that would allow us to make a variable arc-consistent with respect to some other variable (note that csp stands for "constraint satisfaction problem"):

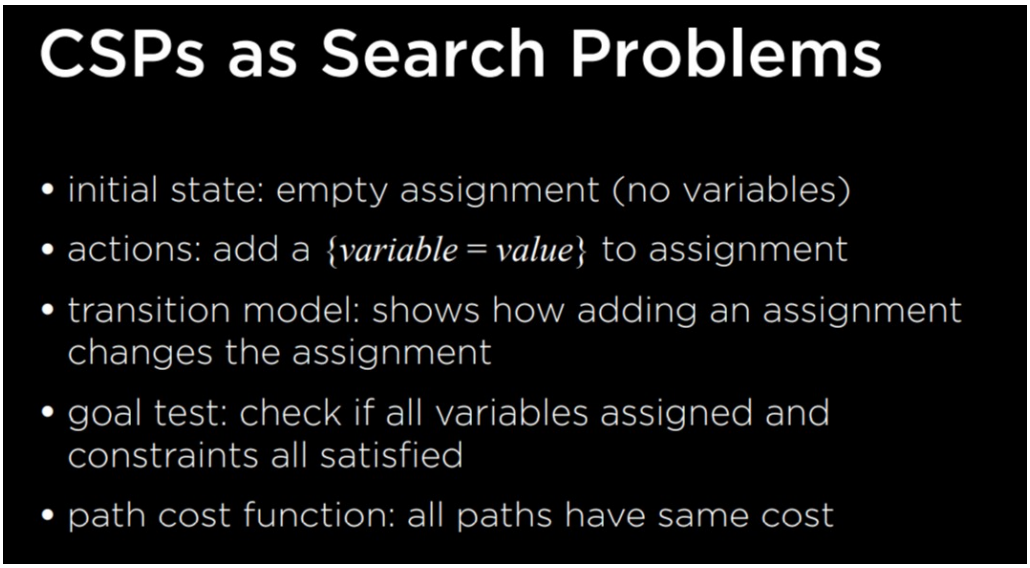


Often we are interested in making the whole problem arc-consistent and not just one variable with respect to another. In this case, we will use an algorithm called AC-3, which uses Revise:



While the algorithm for arc consistency can simplify the problem, it will not necessarily solve it, since it considers binary constraints only and not how multiple nodes might be interconnected. Our previous example, where each of 4 students is taking 3 courses, remains unchanged by running AC-3 on it.

A constraint satisfaction problem can be seen as a search problem:

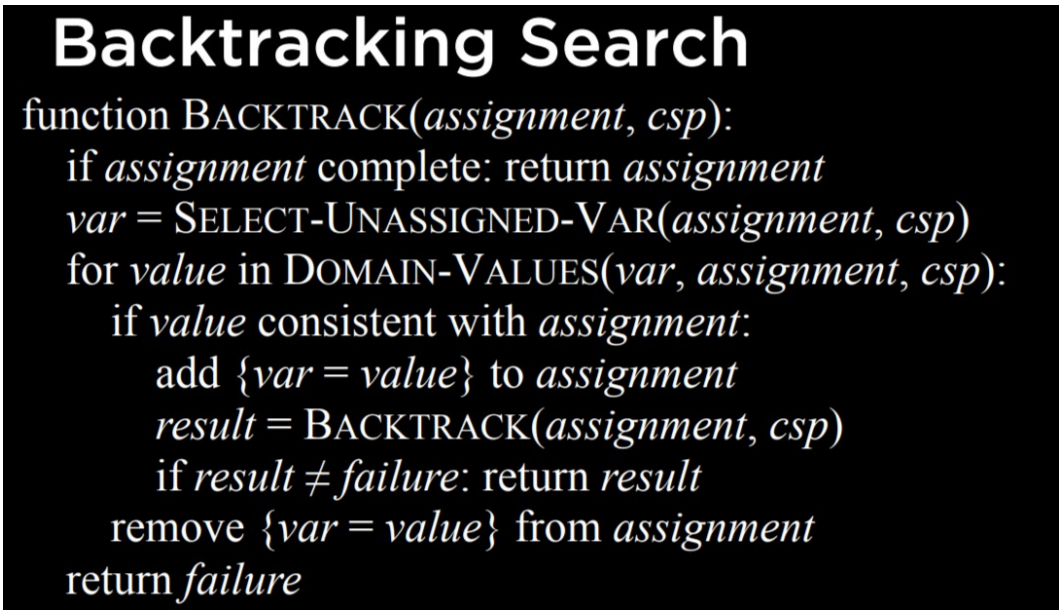


However, going about a constraint satisfaction problem naively, as a regular search problem, is massively inefficient.

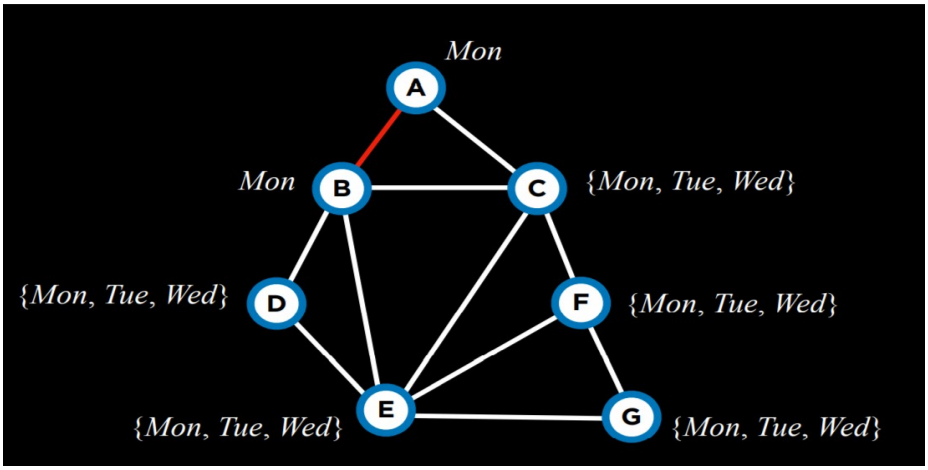
4.4. Backtracking Search

Backtracking search is a type of a search algorithm that takes into account the structure of a constraint satisfaction search problem. In general, it is a recursive function that attempts to continue assigning values as long as they satisfy the constraints. If constraints are violated, it tries a different assignment.

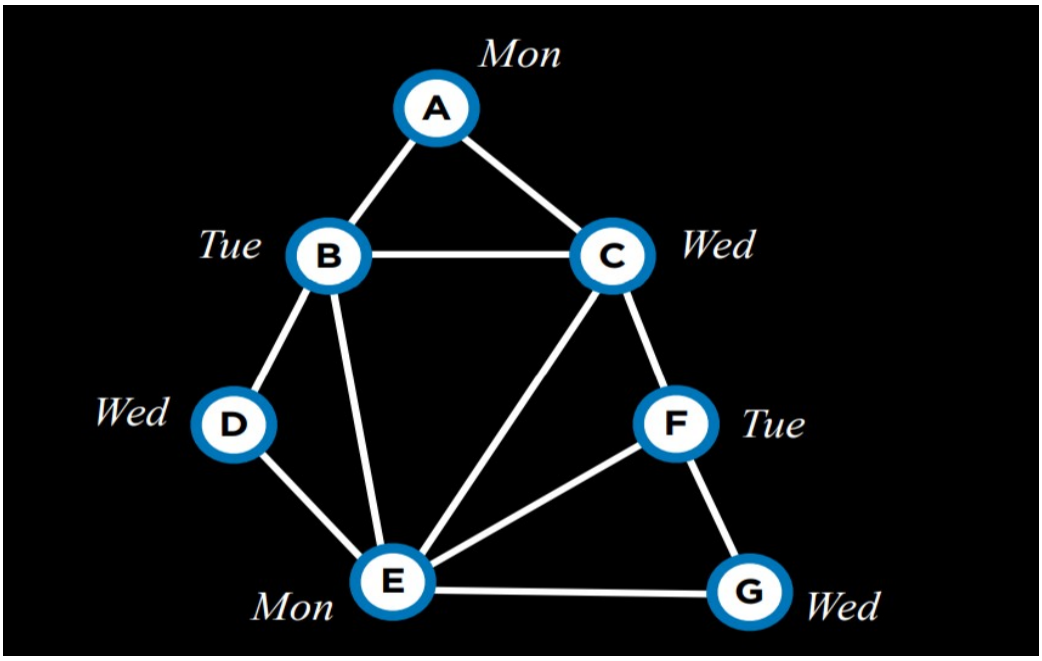
Here's the pseudocode of how it would look like (Naive backtracking search without any heuristics or inference):



If we see an example:



First we choose in the first node the value Monday (randomly) and then we try in the B node the value monday, then, using this assignment, we run the algorithm again, the algorithm will consider B, and assign Monday to it (bottom left), and it would return false, so the backtracking search algorithm goes back and tries another value, and this process repeats itself recursively until a solution is found or it determines that no solution is possible.



As mentioned in the previous pages we can also make inferences, Although backtracking search is more efficient than simple search, it still takes a lot of

computational power. Enforcing arc consistency, on the other hand, is less resource intensive. By interleaving backtracking search with inference (enforcing arc consistency), we can get at a more efficient algorithm.

This algorithm is called the ***Maintaining Arc-Consistency algorithm***. This algorithm will enforce arc-consistency after every new assignment of the backtracking search.

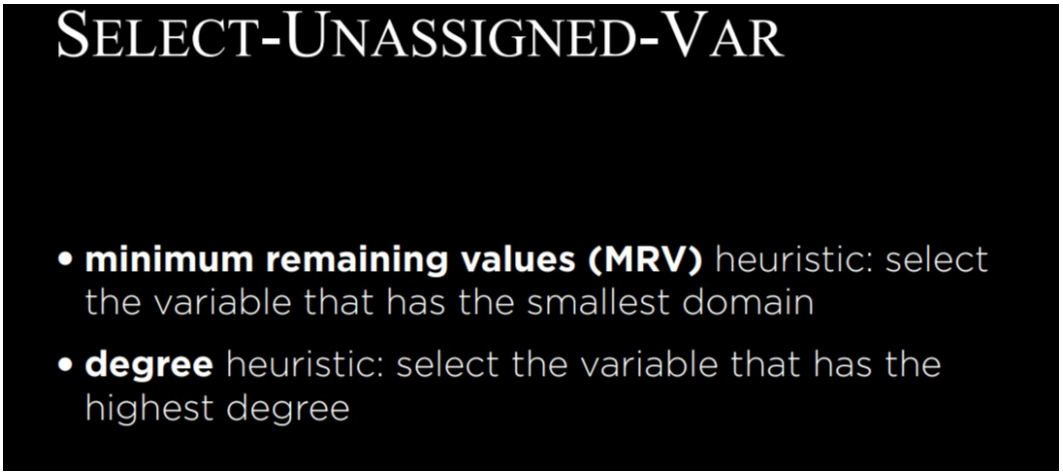
maintaining arc-consistency

When we make a new assignment to X, calls AC-3, starting with a queue of all arcs (Y, X) where Y is a neighbor of X

So here's how our backtracking search algorithm would look like with maintaining arc-consistency:

```
function BACKTRACK(assignment, csp):
    if assignment complete: return assignment
    var = SELECT-UNASSIGNED-VAR(assignment, csp)
    for value in DOMAIN-VALUES(var, assignment, csp):
        if value consistent with assignment:
            add {var = value} to assignment
            inferences = INFERENCE(assignment, csp)
            if inferences ≠ failure: add inferences to assignment
            result = BACKTRACK(assignment, csp)
            if result ≠ failure: return result
        remove {var = value} and inferences from assignment
    return failure
```

Now, we can talk a little bit more in depth about the ***SELECT-UNASSIGNED-VAR*** function. In the previous examples we were choosing these variables at random, but if we choose them with some criteria, by following certain heuristics we can actually make our algorithms more efficient:



A degree is how many arcs connect a node to other nodes.

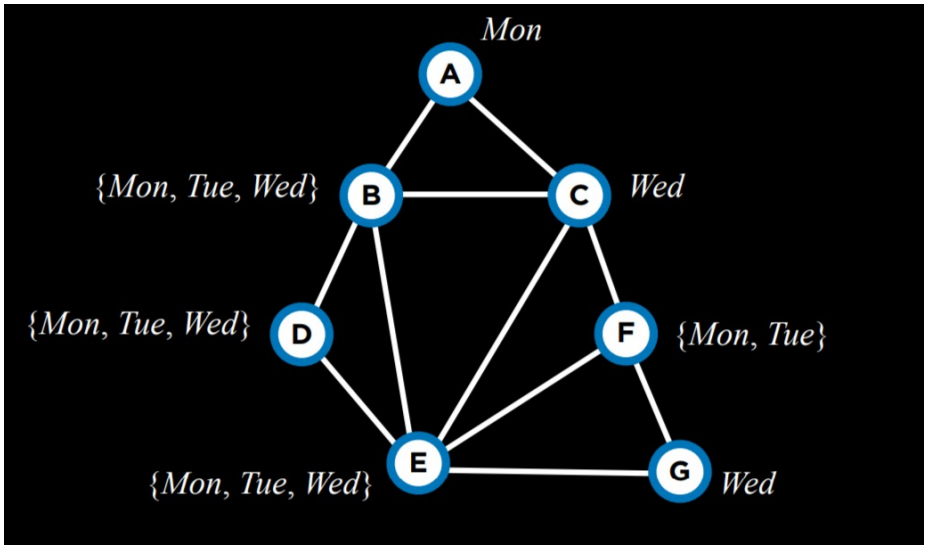
Another way to make the algorithm more efficient is employing yet another heuristic when we select a value from the domain of a variable:

```
function BACKTRACK(assignment, csp):
    if assignment complete: return assignment
    var = SELECT-UNASSIGNED-VAR(assignment, csp)
    for value in DOMAIN-VALUES(var, assignment, csp):
        if value consistent with assignment:
            add {var = value} to assignment
            inferences = INFERENCE(assignment, csp)
            if inferences ≠ failure: add inferences to assignment
            result = BACKTRACK(assignment, csp)
            if result ≠ failure: return result
        remove {var = value} and inferences from assignment
    return failure
```

DOMAIN-VALUES

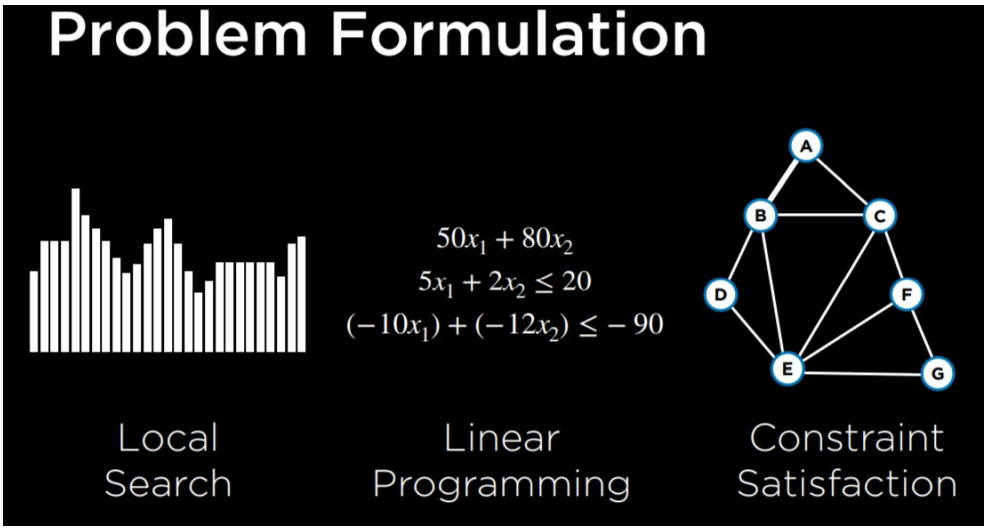
- **least-constraining values** heuristic: return variables in order by number of choices that are ruled out for neighboring variables
  - try least-constraining values first

The idea here is that, while in the degree heuristic we wanted to use the variable that is more likely to constrain other variables, here we want this variable to place the least constraints on other variables. That is, we want to locate what could be the largest potential source of trouble (the variable with the highest degree), and then render it the least troublesome that we can (assign the least constraining value to it).



For example, let's consider variable C. If we assign Tuesday to it, we will put a constraint on all of B, E, and F. However, if we choose Wednesday, we will put a constraint only on B and E. Therefore, it is probably better to go with Wednesday.

To summarize, optimization problems can be formulated in multiple ways:



4.5. Quiz

*(*Justification in txt)

✓ For which of the following will you always find the same solution, even if you re-run the algorithm multiple times? *1/1

Assume a problem where the goal is to minimize a cost function, and every state in the state space has a different cost.

- ○ Steepest-ascent hill-climbing, each time starting from a different starting state
- ◉ Steepest-ascent hill-climbing, each time starting from the same starting state ✓
- ○ Stochastic hill-climbing, each time starting from a different starting state
- ○ Stochastic hill-climbing, each time starting from the same starting state
- ○ Both steepest-ascent and stochastic hill climbing, so long as you always start from the same starting state
- ○ Both steepest-ascent and stochastic hill climbing, each time starting from a different starting state
- ○ No version of hill-climbing will guarantee the same solution every time

The following two questions will both ask you about the optimization problem described below.

A farmer is trying to plant two crops, Crop 1 and Crop 2, and wants to maximize his profits. The farmer will make $500 in profit from each acre of Crop 1 planted, and will make $400 in profit from each acre of Crop 2 planted.

However, the farmer needs to do all of his planting today, during the 12 hours between 7am and 7pm. Planting an acre of Crop 1 takes 3 hours, and planting an acre of Crop 2 takes 2 hours.

The farmer is also limited in terms of supplies: he has enough supplies to plant 10 acres of Crop 1 and enough supplies to plant 4 acres of Crop 2.

Assume the variable C1 represents the number of acres of Crop 1 to plant, and the variable C2 represents the number of acres of Crop 2 to plant.

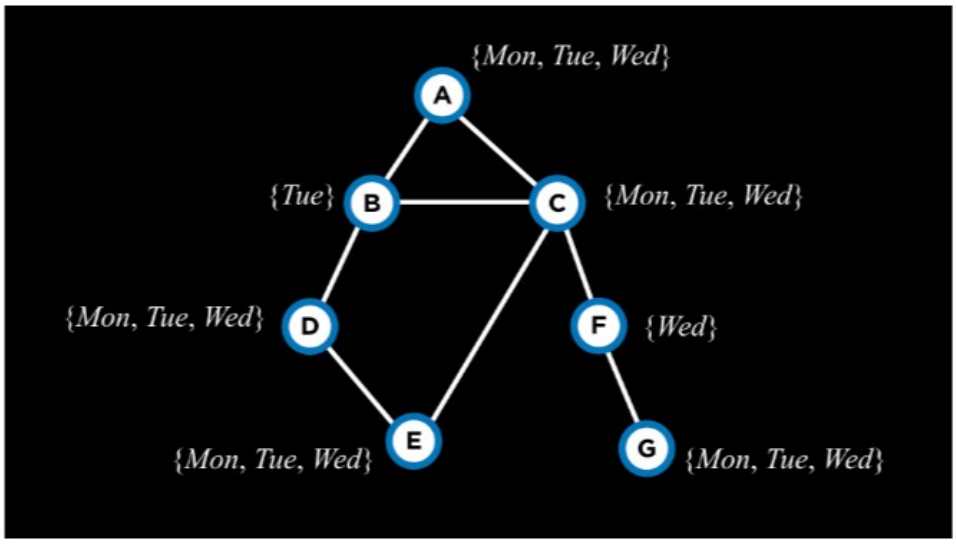
✓ What would be a valid objective function for this problem? * 1/1

- ○ 500 * 10 * C1 + 400 * 4 * C2
- ◉ 500 * C1 + 400 * C2 ✓
- ○ -3 * C1 - 2 * C2
- ○ C1 + C2
- ○ 10 * C1 + 4 * C2

✓ What are the constraints for this problem? * 1/1

- ◉ 3 * C1 + 2 * C2 <= 12, C1 <= 10, C2 <= 4 ✓
- ○ 3 * C1 + 2 * C2 <= 12, C1 + C2 <= 14
- ○ 3 * C1 <= 10, 2 * C2 <= 4
- ○ C1 + C2 <= 12, C1 + C2 <= 14

The following question will ask you about the below exam scheduling constraint satisfaction graph, where each node represents a course. Each course is associated with an initial domain of possible exam days (most courses could be on Monday, Tuesday, or Wednesday; a few are already restricted to just a single day). An edge between two nodes means that those two classes must have exams on different days.



✓ After enforcing arc consistency on this entire problem, what are the resulting domains for the variables C, D, and E? *1/1

- ○ C's domain is {Mon, Tue}, D's domain is {Wed}, E's domain is {Mon}
- ◉ C's domain is {Mon}, D's domain is {Mon, Wed}, E's domain is {Tue, Wed} ✓
- ○ C's domain is {Mon}, D's domain is {Tue}, E's domain is {Wed}
- ○ C's domain is {Mon, Tue, Wed}, D's domain is {Mon, Wed}, E's domain is {Mon, Tue, Wed}
- ○ C's domain is {Mon}, D's domain is {Mon, Wed}, E's domain is {Mon, Tue, Wed}
- ○ C's domain is {Mon}, D's domain is {Wed}, E's domain is {Tue}

5. Learning

In this chapter we will start with machine learning. Machine learning provides a computer with data, rather than explicit instructions. Using these data, the computer learns to recognize patterns and becomes able to execute tasks on its own.

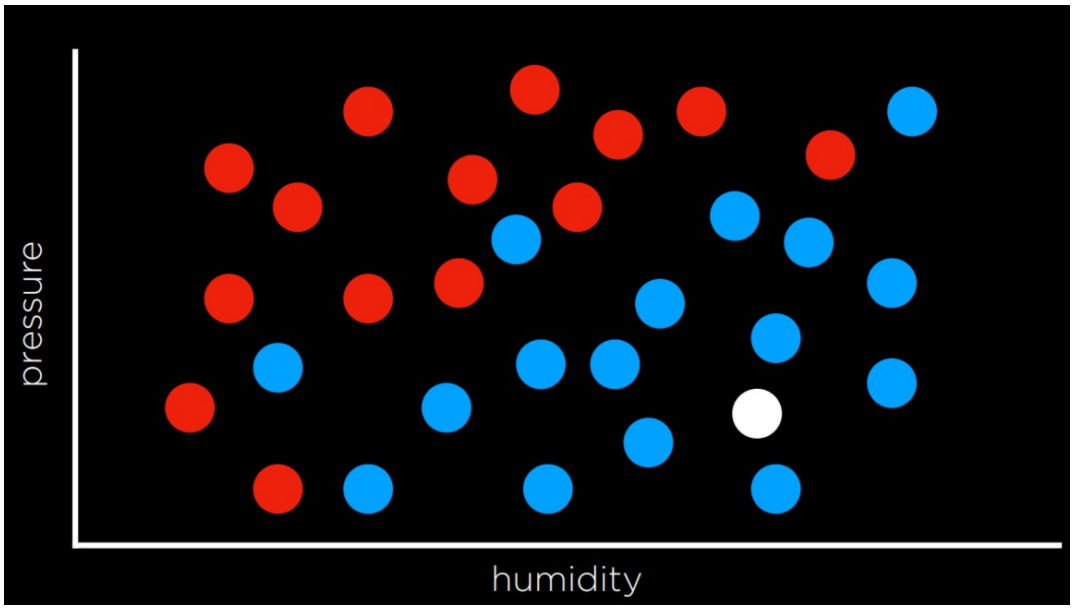
5.1. Supervised Learning

There are lots of tasks under supervised learning, for example:

5.1.1. Classification

Classification is the supervised learning task of learning a function mapping an input point to a discrete category.

This task can be formalized as follows. We observe nature, where a function *f(humidity, pressure)* maps the input to a discrete value, either Rain or No Rain. This function is hidden from us, and it is probably affected by many other variables that we don't have access to. Our goal is to create function *h(humidity, pressure)* that can approximate the behavior of function *f*. Such a task can be visualized by plotting days on the dimensions of humidity and rain (the input), coloring each data point in blue if it rained that day and in red if it didn't rain that day (the output). The white data point has only the input, and the computer needs to figure out the output.



5.1.1.1. Nearest-neighbor classification

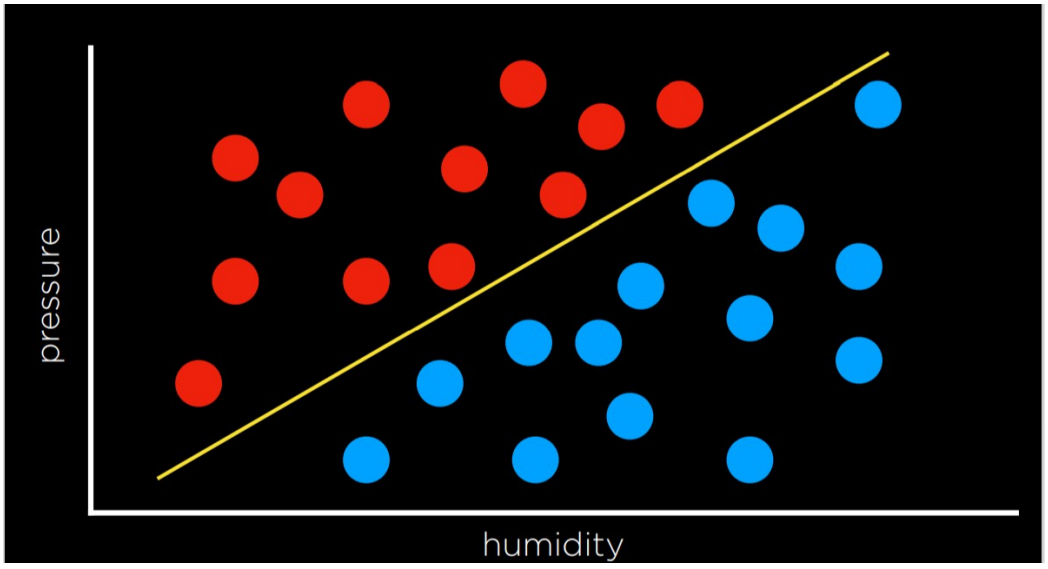
This a type of classification algorithm that, given an input, chooses the class of the nearest data point to that input. You can think about it in the context of the previous example.

One way to get around the limitations of nearest-neighbor classification is by using **k-nearest-neighbors classification**, where the dot is colored based on the most frequent color of the k nearest neighbors. It is up to the programmer to decide what k is. Using a 3-nearest neighbors classification, for example, the white dot above will be colored blue, which intuitively seems like a better decision.

A drawback of the k-nearest-neighbors classification is that, using a naive approach, the algorithm will have to measure the distance of every single point to the point in question, which is computationally expensive. This can be sped up by using data structures that enable finding neighbors more quickly or by pruning irrelevant observation.

5.1.1.2. Perceptron learning

Another way of going about a classification problem, is looking at the data as a whole and trying to create a decision boundary. In two-dimensional data, we can draw a line between the two types of observations. Every additional data point will be classified based on the side of the line on which it is plotted.



The drawback to this approach is that data are messy, and it is rare that one can draw a line and neatly divide the classes into two observations without any mistakes. Often, we will compromise, drawing a boundary that separates the observations correctly more often than not, but still occasionally misclassifies them.

The input (x1, x2) will be given to a hypothesis function $h(x_1, x_2)$, which will output its prediction of whether it is going to rain that day or not by checking on which side of the decision boundary the observation falls. Formally, the function will weight each of the inputs with an addition of a constant, ending in a linear equation of the following form:

x_1 = Humidity

x_2 = Pressure

$$h(x_1, x_2) = \begin{cases} \text{Rain if } w_0 + w_1x_1 + w_2x_2 \geq 0 \\ \text{No Rain otherwise} \end{cases}$$

The weights and values are represented by vectors. We produce a Weight Vector **w**: (w_0, w_1, w_2), and getting to the best weight vector is the goal of the machine learning algorithm. We also produce an Input Vector **x**: (1, x_1, x_2).

We take the dot product of the two vectors arriving at the expression above: $w_0 + w_1x_1 + w_2x_2$. The first value in the input vector is 1 because, when multiplied by the weight vector w_0, we want to keep it a constant.

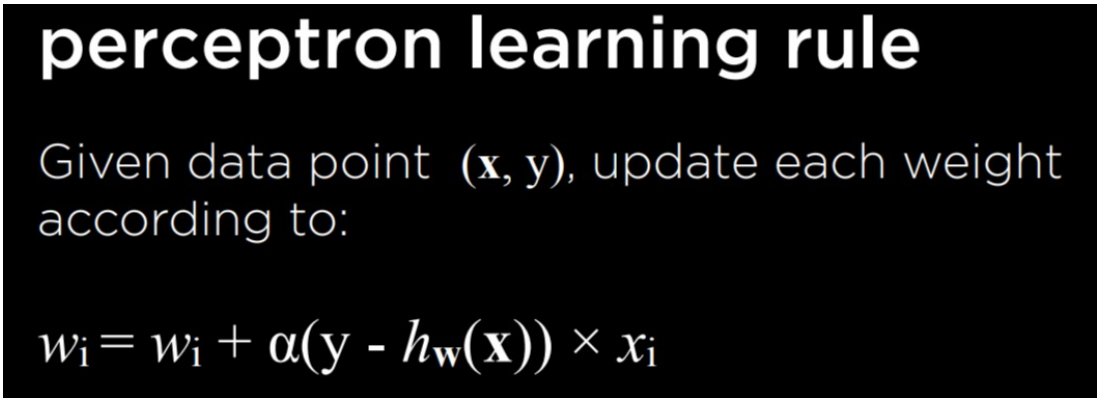
Weight Vector **w**: (w_0, w_1, w_2)

Input Vector **x**: $(1, x_1, x_2)$

$\mathbf{w} \cdot \mathbf{x}$: $w_0 + w_1x_1 + w_2x_2$

$$h_{\mathbf{w}}(\mathbf{x}) = \begin{cases} 1 \text{ if } \mathbf{w} \cdot \mathbf{x} \geq 0 \\ 0 \text{ otherwise} \end{cases}$$

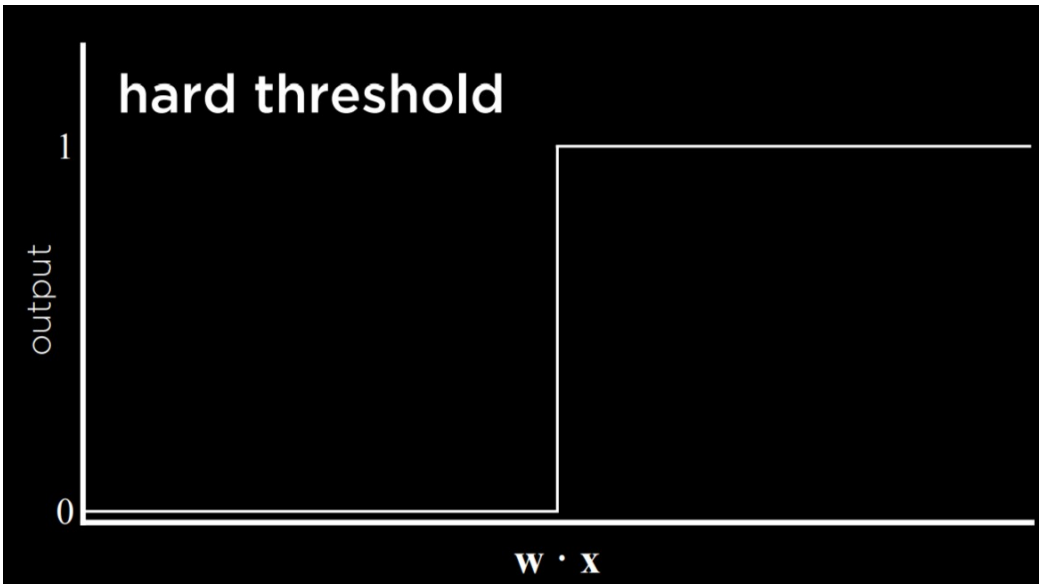
Since the goal of the algorithm is to find the best weight vector, when the algorithm encounters new data it updates the current weights. It does so using the **perceptron learning rule**:



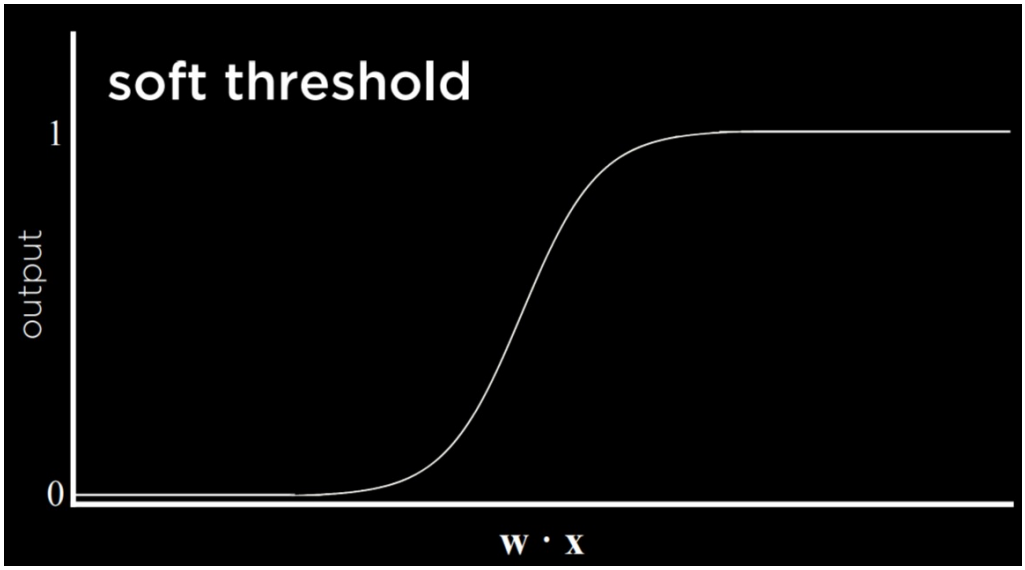
The important takeaway from this rule is that for each data point, we adjust the weights to make our function more accurate. Here, y stands for the observed value while the hypothesis function stands for the estimate. If they are identical, this whole term is equal to zero, and thus the weight is not changed.

If we underestimated (calling No Rain while Rain was observed), then the value in the parentheses will be 1 and the weight will increase by the value of x_i scaled by α the learning coefficient. If we overestimated (calling Rain while No Rain was observed), then the value in the parentheses will be -1 and the weight will decrease by the value of x scaled by α. The higher α, the stronger the influence each new event has on the weight.

The result of this process is a threshold function that switches from 0 to 1 once the estimated value crosses some threshold:



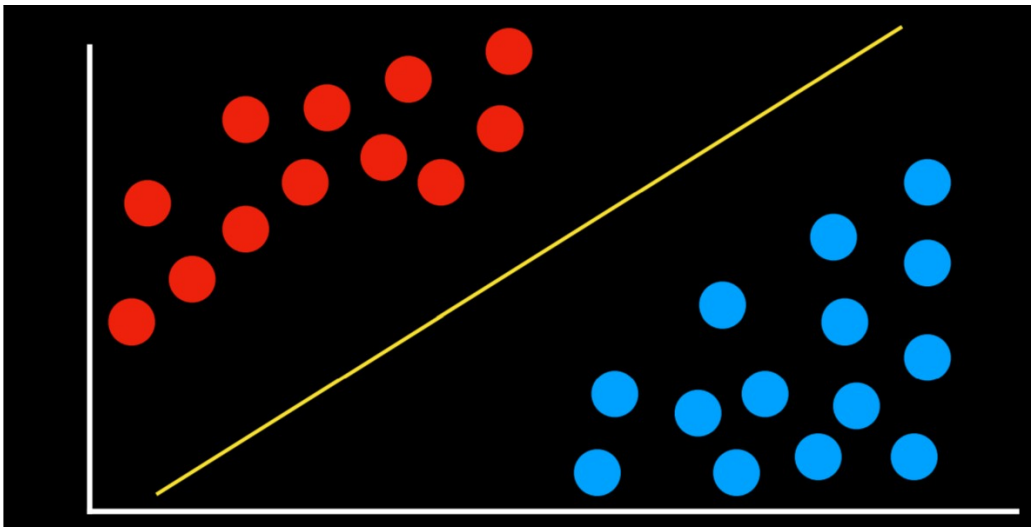
The problem with this type of function is that it is unable to express uncertainty. A way to go around this is by using a logistic function, which employs **a soft threshold.** A logistic function can yield a real number between 0 and 1, which will express confidence in the estimate.



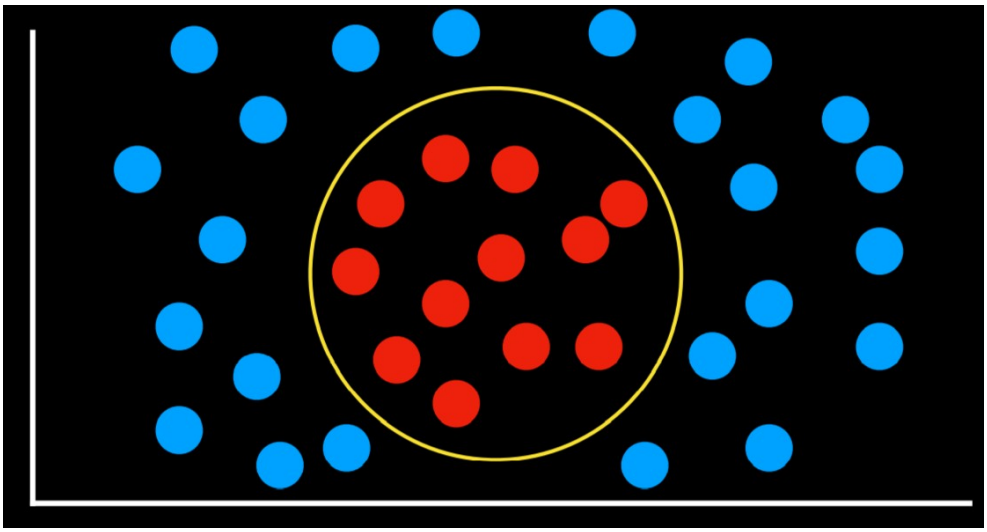
5.1.1.3. Support Vector Machines

This approach uses an additional vector (support vector) near the decision boundary to make the best decision when separating the data.

This type of boundary, which is as far as possible from the two groups it separates, is called the **Maximum Margin Separator**.

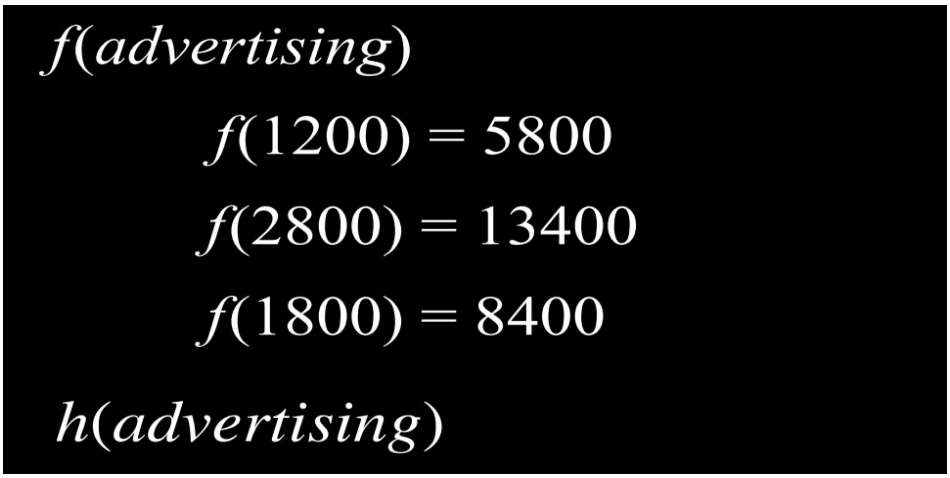


Another benefit of support vector machines is that they can represent decision boundaries with more than two dimensions, as well as non-linear decision boundaries, such as below.



5.1.2. Regression

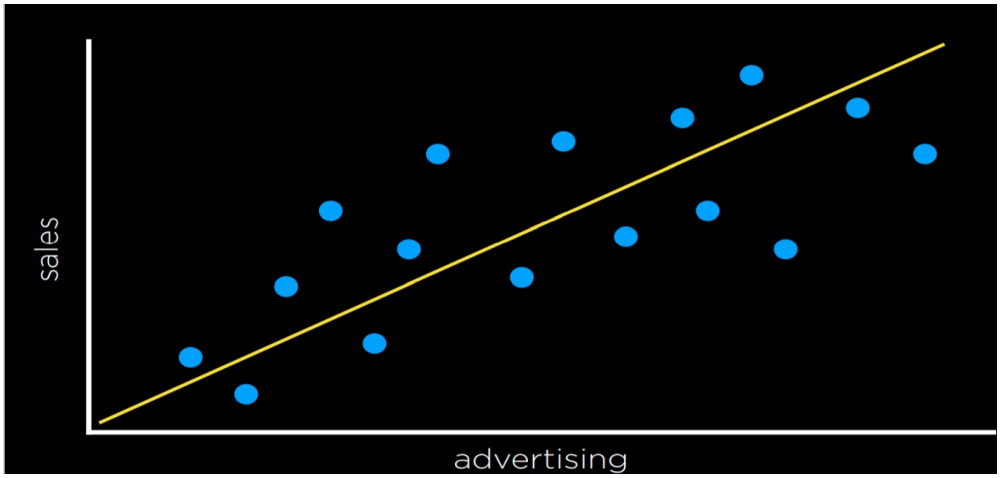
Regression is a supervised learning task of a function that maps an input point to a continuous value, some real number. This differs from classification in that classification problems map an input to discrete values (Rain or No Rain).



f(advertising) represents the observed income following some money that was spent in advertising (note that the function can take more than one input variable). These are the data that we start with.

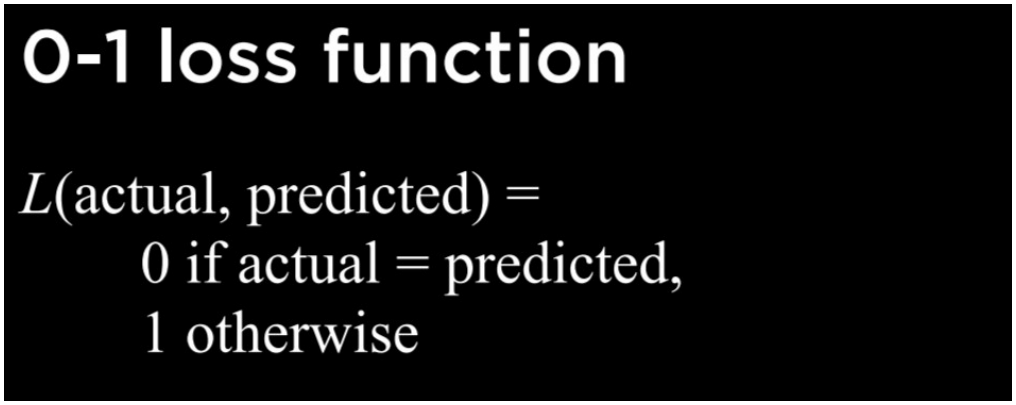
With this data, we want to come up with a hypothesis function ***h(advertising)*** that will try to approximate the behavior of function *f*.

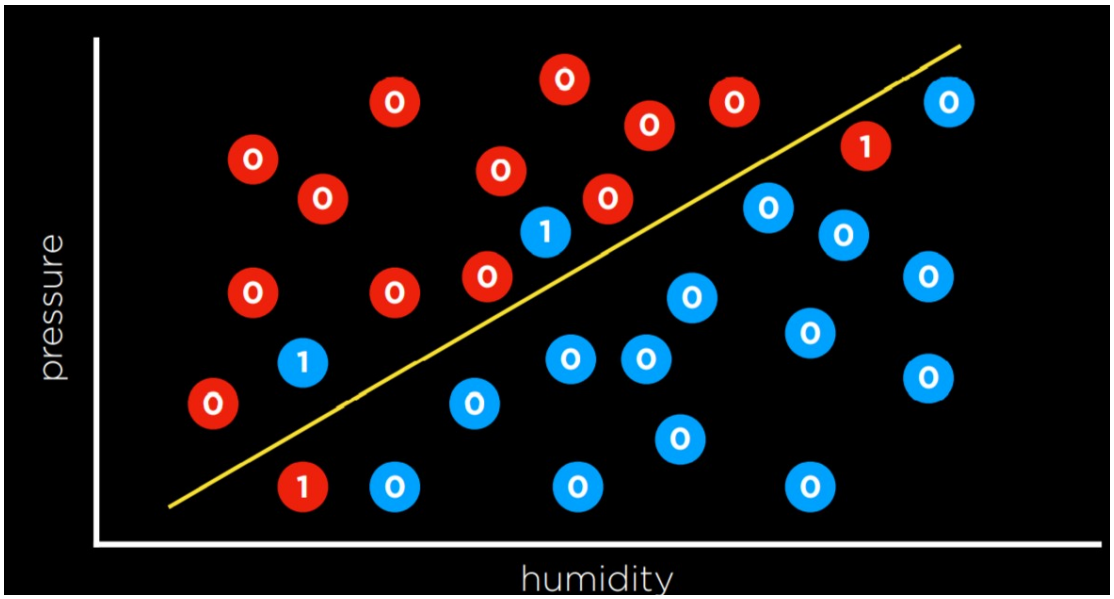
h will generate a line whose goal is not to separate between types of observations, but to predict, based on the input, what will be the value of the output.



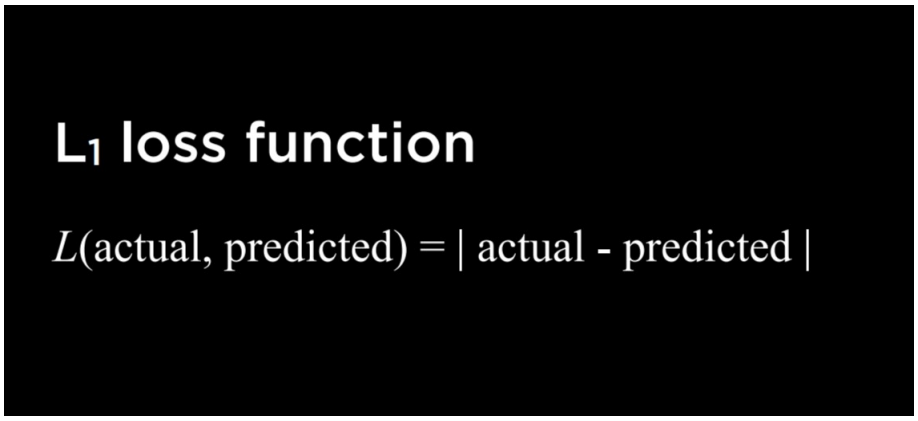
5.1.3. Loss Functions

Loss functions are a way to quantify the utility lost by any of the decision rules above. The less accurate the prediction, the larger the loss.

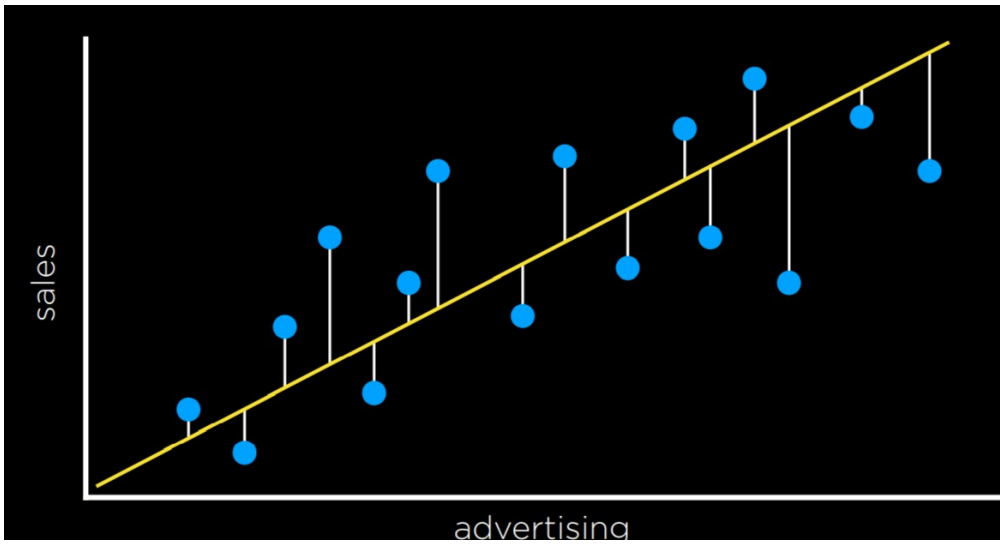




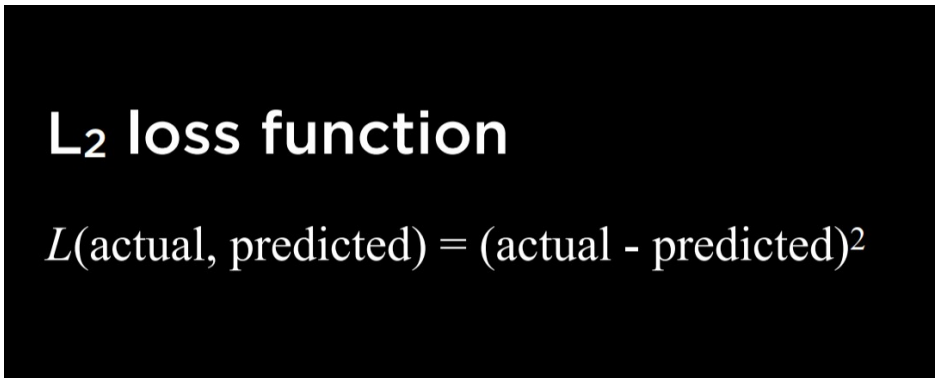
L_1 and L_2 loss functions can be used when predicting a continuous value. In this case, we are interested in quantifying for each prediction how much it differed from the observed value.



L_1 can be visualized by summing the distances from each observed point to the predicted point on the regression line:

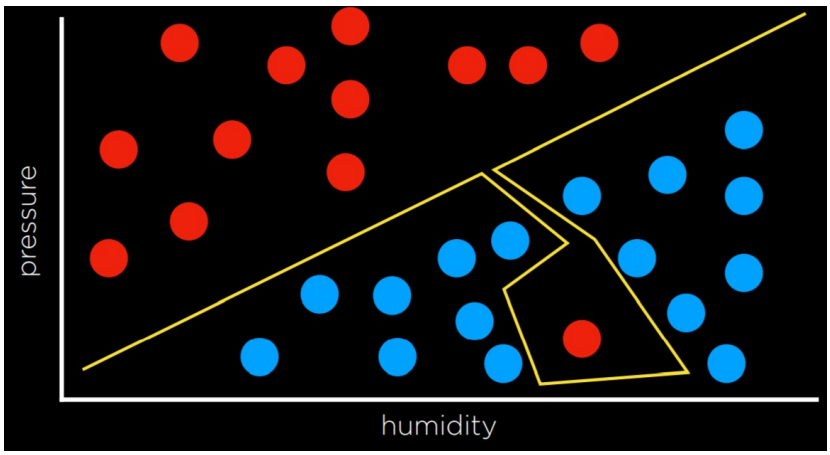


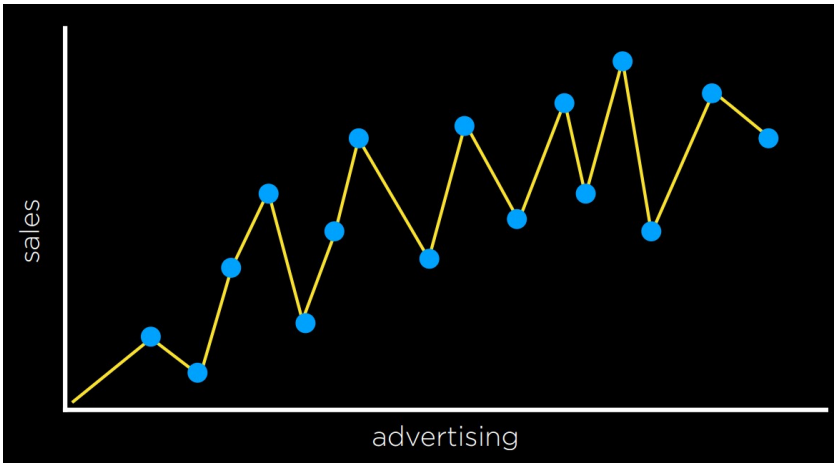
One can choose the loss function that serves their goals best. L_2 penalizes outliers more harshly than L_1 because it squares the the difference:



5.1.4. Overfitting

Overfitting is when a model fits the training data so well that it fails to generalize to other data sets. In this sense, loss functions are a double edged sword. In the two examples below, the loss function is minimized such that the loss is equal to 0. However, it is unlikely that it will fit new data well.





5.1.5. Regularization

Regularization is the process of penalizing hypotheses that are more complex to favor simpler, more general hypotheses. We use regularization to avoid overfitting.

In regularization, we estimate the cost of the hypothesis function h by adding up its loss and a measure of its complexity.

cost(h) = loss(h) + λcomplexity(h)

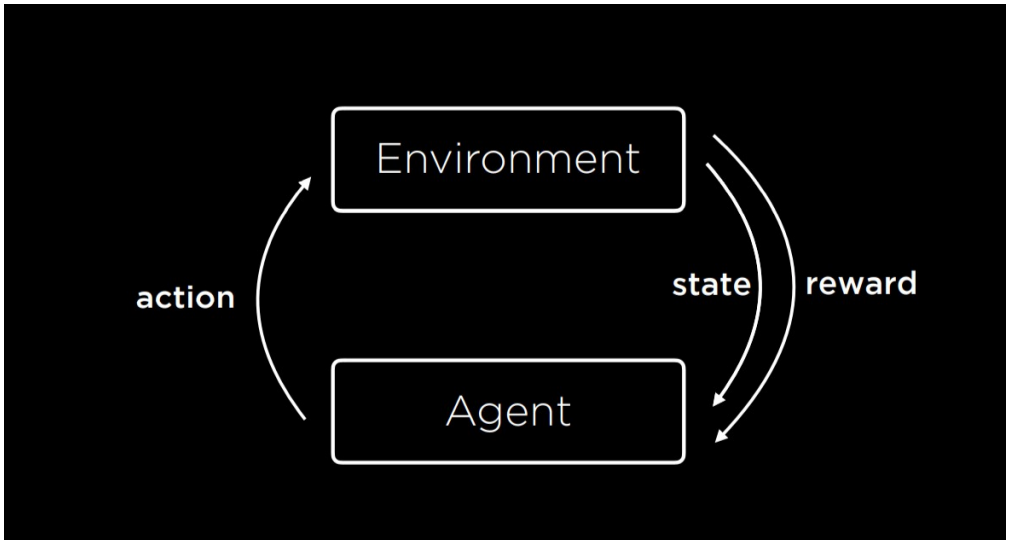
Lambda (λ) is a constant that we can use to modulate how strongly to penalize for complexity in our cost function. The higher λ is, the more costly complexity is.

One way to test whether we overfitted the model is with **Holdout Cross Validation**. In this technique, we split all the data in two: a **training set** and a **test set**. We run the learning algorithm on the training set, and then see how well it predicts the data in the test set. The idea here is that by testing on data that were not used in training, we can a measure how well the learning generalizes.The downside of holdout cross validation is that we don't get to train the model on half the data, since it is used for evaluation purposes.

A way to deal with this is using **k-Fold Cross-Validation**. In this process, we divide the data into k sets. We run the training k times, each time leaving out one dataset and using it as a test set. We end up with k different evaluations of our model, which we can average and get an estimate of how our model generalizes without losing any data.

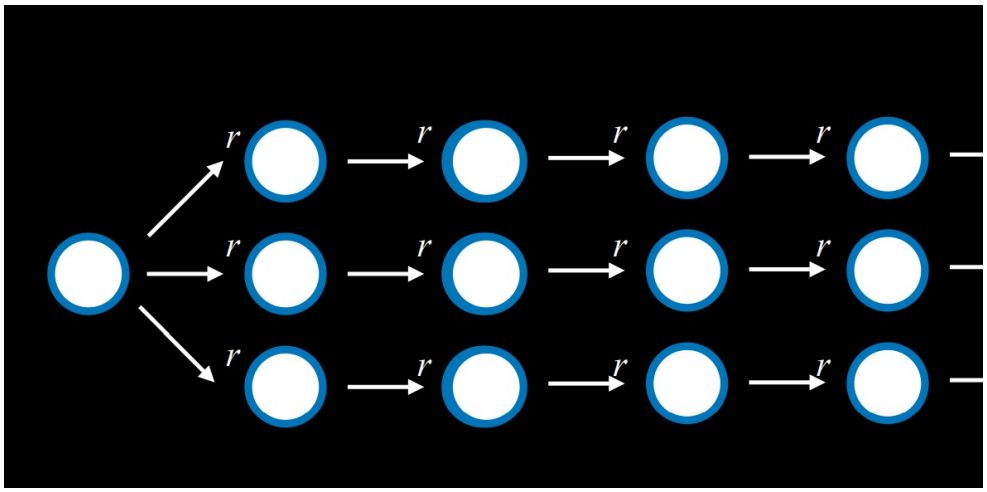
5.2. Reinforcement Learning

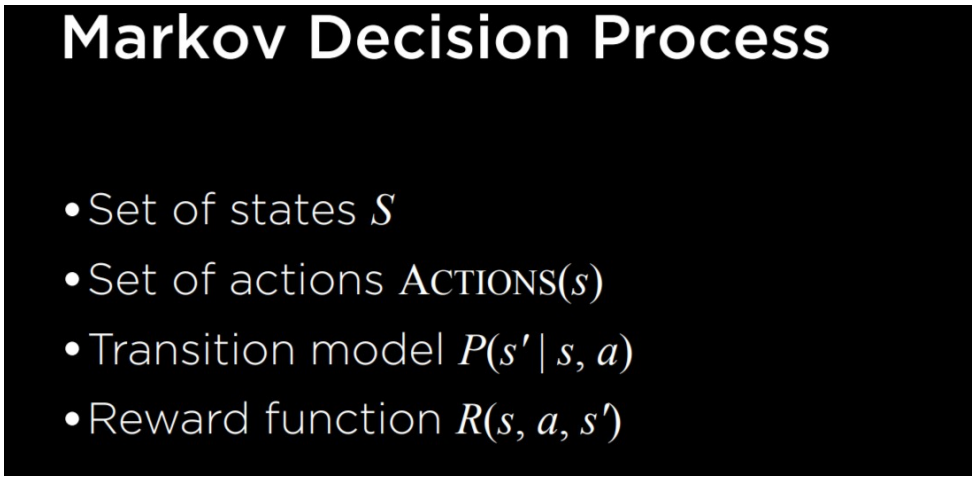
Reinforcement learning is another approach to machine learning, where after each action, the agent gets feedback in the form of reward or punishment (a positive or a negative numerical value).



5.2.1. Markov decision processes

Reinforcement learning can be viewed as a Markov decision process, Markov decision process is a Model for decision-making, representing states, actions, and their rewards.

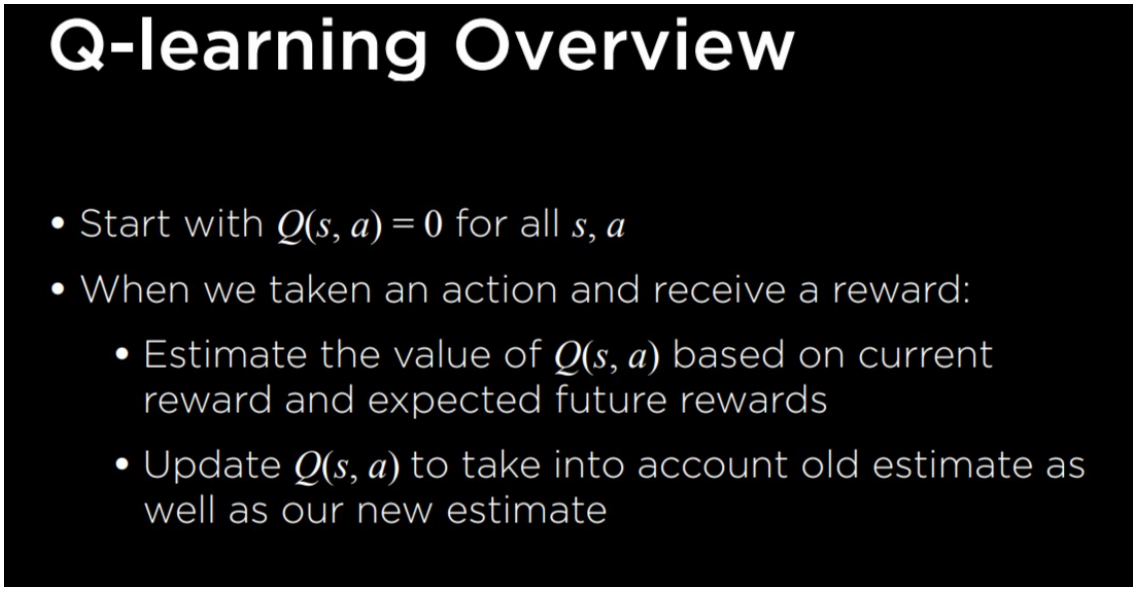


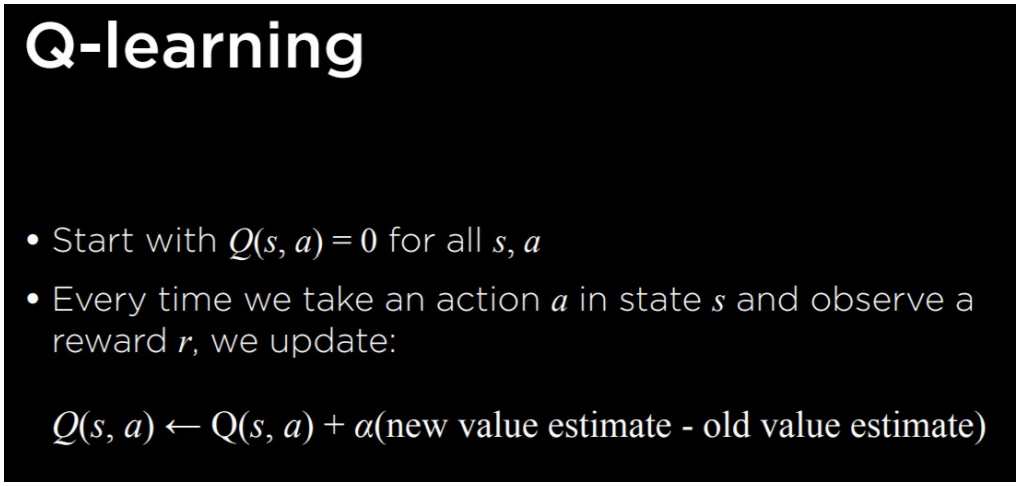


Where s is the original state, a is the action and s' is the new state that we are currently in.

5.2.2. Q-Learning

Q-Learning is one model of reinforcement learning, where a function **Q(s, a)** outputs an estimate of the value of taking action *a* in state *s*.





Alpha represents in effect, how much we value new information compared to how much we value old information, 0 would mean ignore all the new information, just keep this Q value the same. 1 that means replace the old information entirely with the new information. And somewhere in between, keep some sort of balance between these two values.

This algorithm can be expressed more formally as:

$$Q(s, a) \leftarrow Q(s, a) + \alpha((r + \gamma \max_{a'} Q(s', a')) - Q(s, a))$$

The new value estimate can be expressed as a sum of the reward (r) and the future reward estimate.

To get the future reward estimate, we consider the new state that we got after taking the last action, and add the estimate of the action in this new state that will bring to the highest reward.

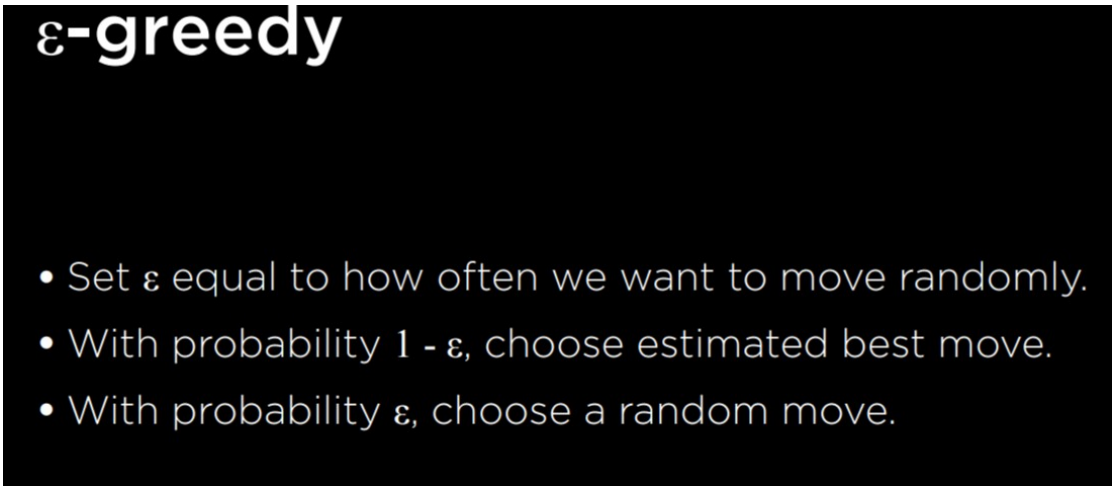
(In other words, the max part, It's going to be take the maximum across all possible actions I could take next and say, all right, of all of these possible actions I could take, which one is going to have the highest reward?)

This way, we estimate the utility of making action *a* in state *s* not only by the reward it received, but also by the expected utility of the next step. The value of the future reward estimate can sometimes appear with a coefficient gamma that controls how much future rewards are valued.

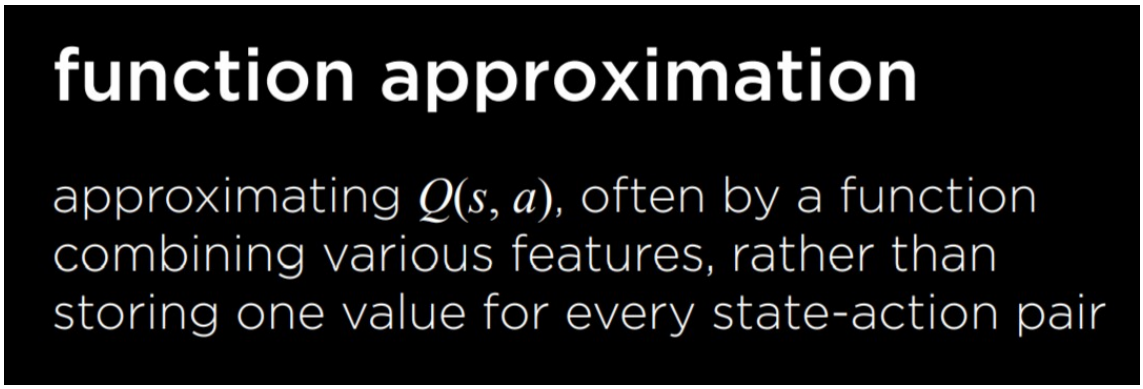
In the other hand a **Greedy Decision-Making algorithm** completely discounts the future estimated rewards, instead always choosing the action **a** in current state **s** that has the highest **Q(s, a)**. This brings us to discuss the **Explore vs. Exploit** tradeoff. A greedy algorithm always exploits, taking the actions that are already established to bring to good outcomes. However, it will always follow the same path to the solution, never finding a better path.

Exploration, on the other hand, means that the algorithm may use a previously unexplored route on its way to the target, allowing it to discover more efficient solutions along the way.

To implement the concept of exploration and exploitation, we can use the **ε (epsilon) greedy algorithm**.



Another way to train a reinforcement learning model is to give feedback not upon every move, but upon the end of the whole process. This approach becomes more computationally demanding when a game has multiple states and possible actions, such as chess. It is infeasible to generate an estimated value for every possible move in every possible state. In this case, we can use a **function approximation.**



Thus, the algorithm becomes able to recognize which moves are similar enough so that their estimated value should be similar as well, and use this heuristic in its decision making.

5.3. Unsupervised Learning

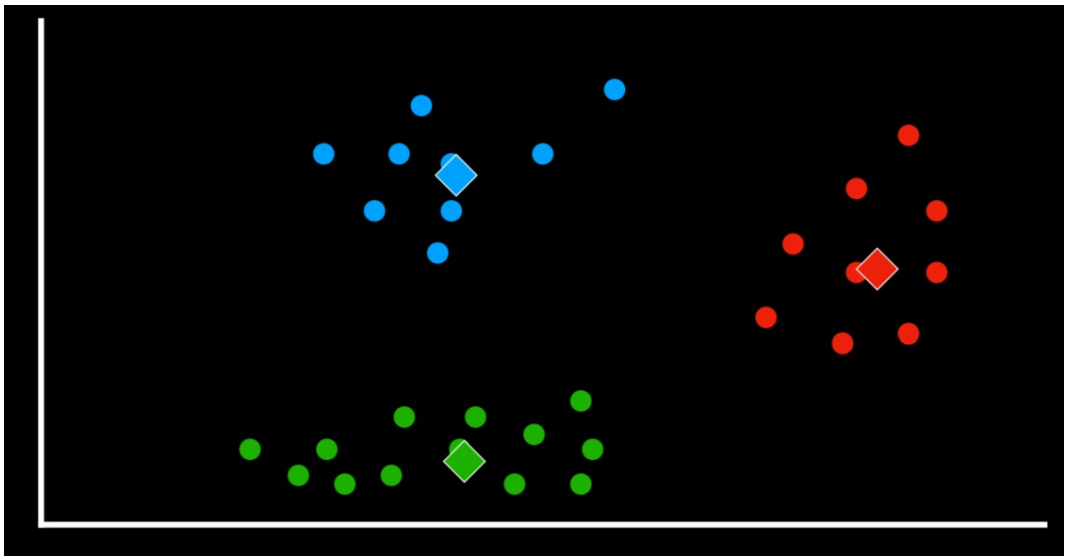
Given input data without any additional feedback, learn patterns.

5.3.1. Clustering

Clustering is an unsupervised learning task that takes the input data and organizes it into groups such that similar objects end up in the same group. This can be used, for example, in genetics research, when trying to find similar genes, or in image segmentation, when defining different parts of the image based on similarity between pixels.

5.3.1.1. k-means Clustering

Algorithm for clustering data based on repeatedly assigning points to clusters and updating those clusters' centers. It maps all data points in a space, and then randomly places k cluster centers in the space (it is up to the programmer to decide how many).



5.4. Quiz

✓ Categorize the following as supervised learning, reinforcement learning, unsupervised learning, or not machine learning: A social network's AI uses existing tagged photos of people to identify when those people appear in new photos. *1/1

- ○ Not an example of machine learning
- ◉ Supervised learning ✓
- ○ Reinforcement learning
- ○ Unsupervised learning

✓ Imagine a regression AI that makes the following predictions for the following 5 data points. What is the total L2 loss across all of these data points (i.e., the sum of all the individual L2 losses for each data point)? *1/1

For data point 1, the true output is 2 and the AI predicted 4. For data point 2, the true output is 4 and the AI predicted 5. For data point 3, the true output is 4 and the AI predicted 3. For data point 4, the true output is 5 and the AI predicted 2. For data point 5, the true output is 6 and the AI predicted 5.

- ○ 0
- ○ 4
- ○ 5
- ○ 8
- ◉ 16 ✓
- ○ 19
- ○ 21
- ○ 64

✓ If Hypothesis 1 has a lower L1 loss and a lower L2 loss than Hypothesis 2 on a set of training data, why might Hypothesis 2 still be a preferable hypothesis? *1/1

- ◉ Hypothesis 1 might be the result of overfitting. ✓
- ○ Hypothesis 1 might be the result of cross-validation.
- ○ Hypothesis 1 might be the result of loss.
- ○ Hypothesis 1 might be the result of regression.
- ○ Hypothesis 1 might be the result of regularization.

✓ In the ε-greedy approach to action selection in reinforcement learning, which of the following values of ε makes the approach identical to a purely greedy approach? *1/1

- (●) ε = 0 ✓
- ( ) ε = 0.25
- ( ) ε = 0.5
- ( ) ε = 0.75
- ( ) ε = 1

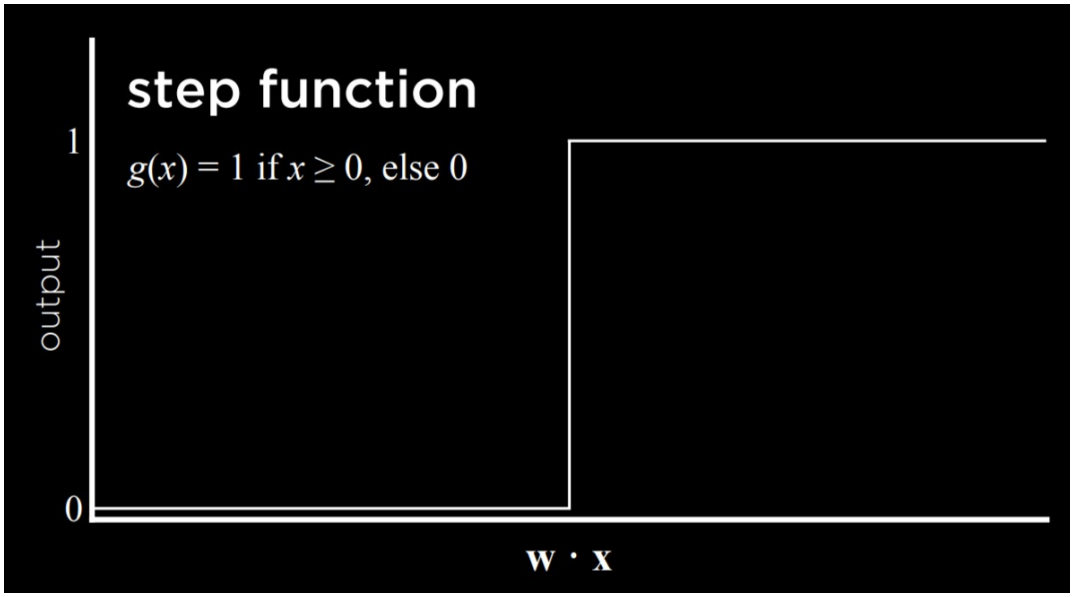
6. Neural Networks

An **Artificial Neural Network** is a mathematical model for learning inspired by biological neural networks. Artificial neural networks model mathematical functions that map inputs to outputs based on the structure and parameters of the network. In artificial neural networks, the structure of the network is shaped through training on data.

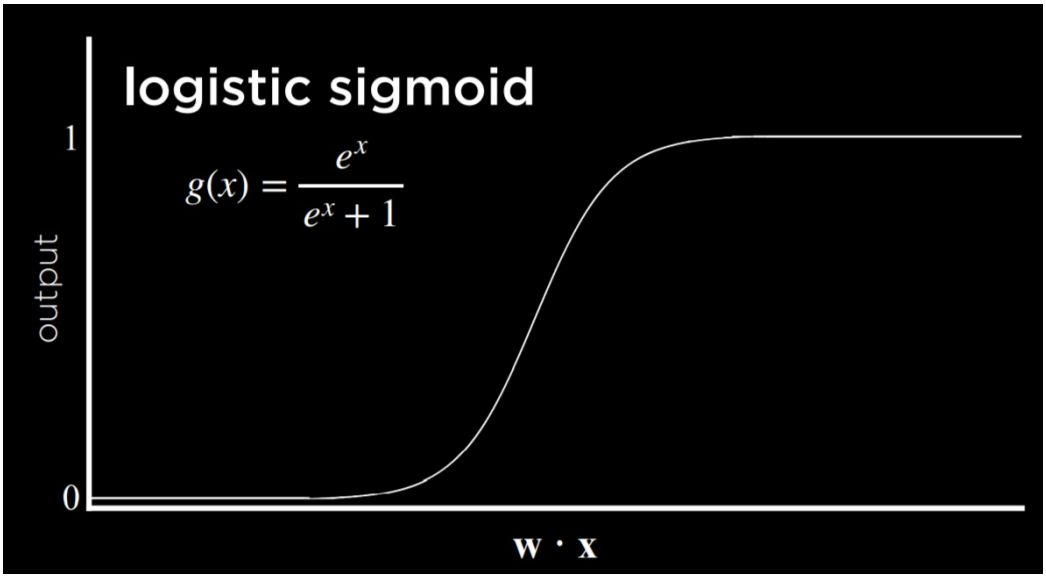
When implemented in AI, the parallel of each neuron is a **unit** that's connected to other units. For example, like in the last lecture, the AI might map two inputs, x_1 and x_2, to whether it is going to rain today or not.

6.1. Activation functions

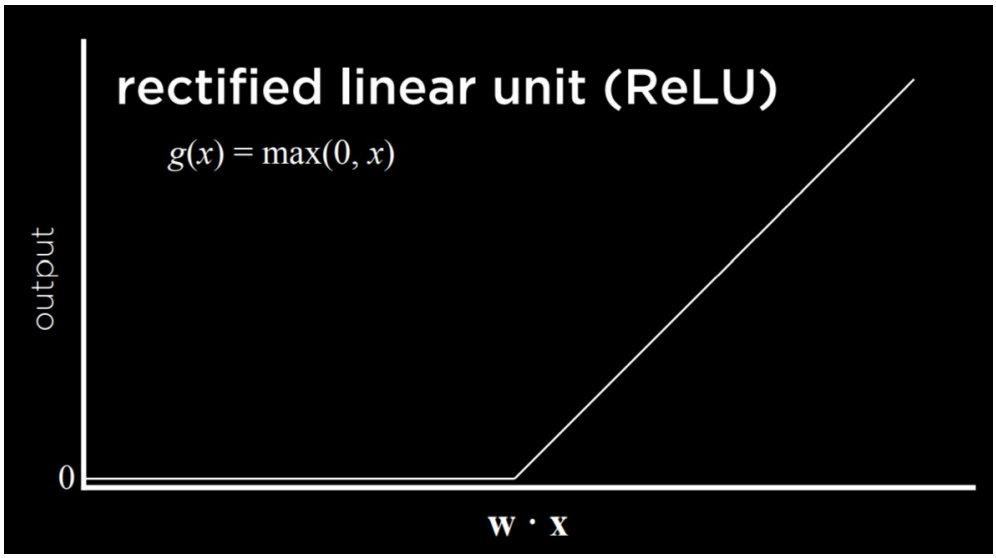
To use the hypothesis function to decide whether it rains or not, we need to create some sort of threshold based on the value it produces. One way to do this is with a step function, which gives 0 before a certain threshold is reached and 1 after the threshold is reached.



Another way to go about this is with a logistic function, which gives as output any real number from 0 to 1, thus expressing graded confidence in its judgment.

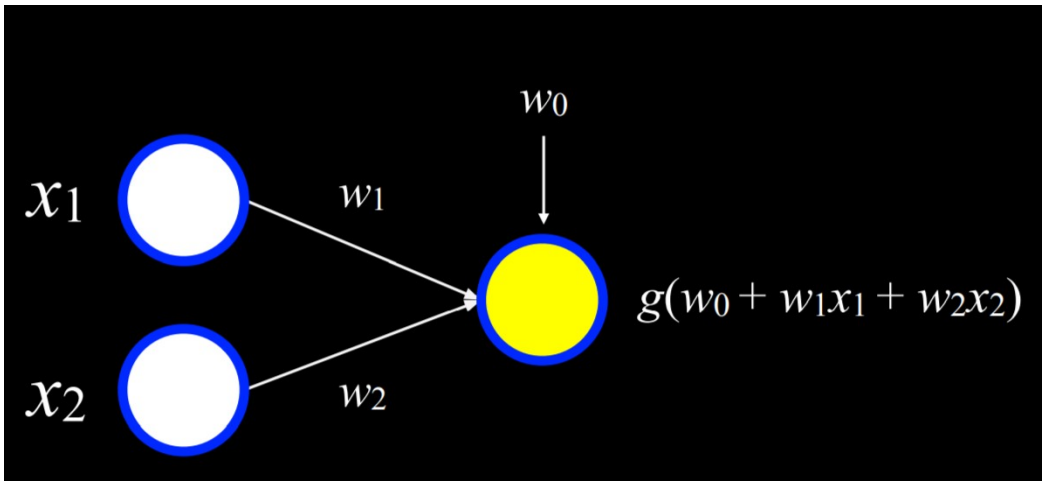


Another possible function is Rectified Linear Unit (ReLU), which allows the output to be any positive value. If the value is negative, ReLU sets it to 0.



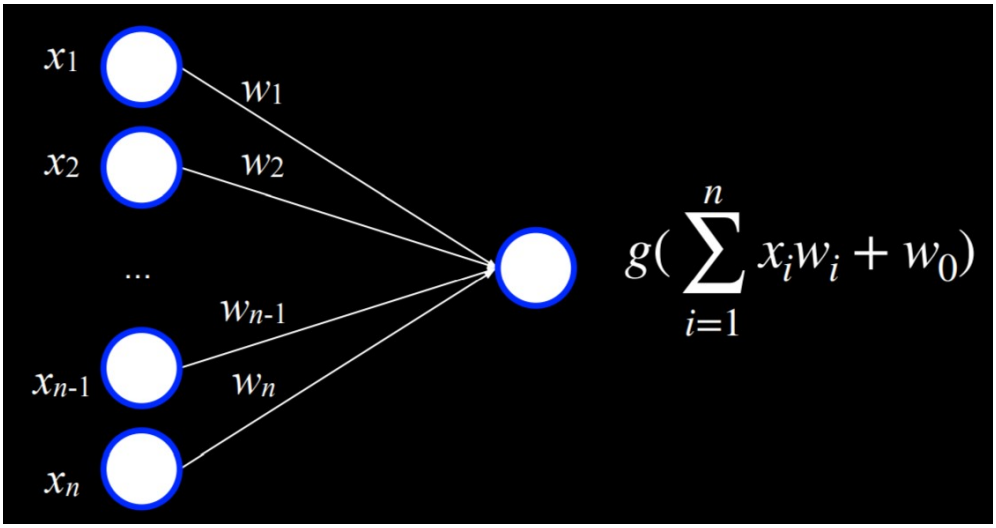
6.2. Neural network structure

A neural network (a simple one) can be thought as this graphical representation where a function sums up inputs to produce outputs:



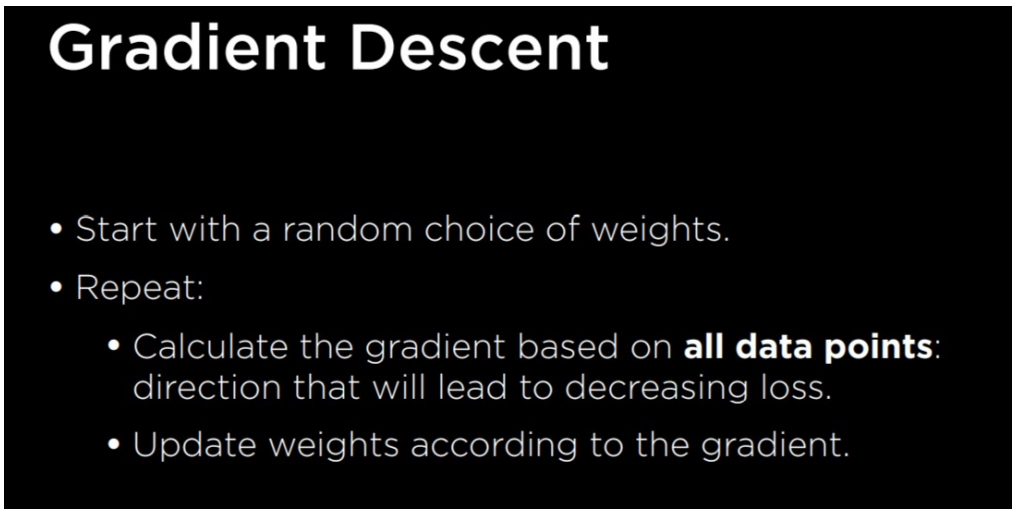
The units on the left are inputs and the unit on the right is the output. The inputs are connected to the output by a weighted edge. To make a decision, the output unit multiplies the inputs by their weights in addition to the bias (w_o), and the uses function *g* to determine the output.

For example we could test the functions logical OR and logical AND where the bias would be -1 and -2 respectively.

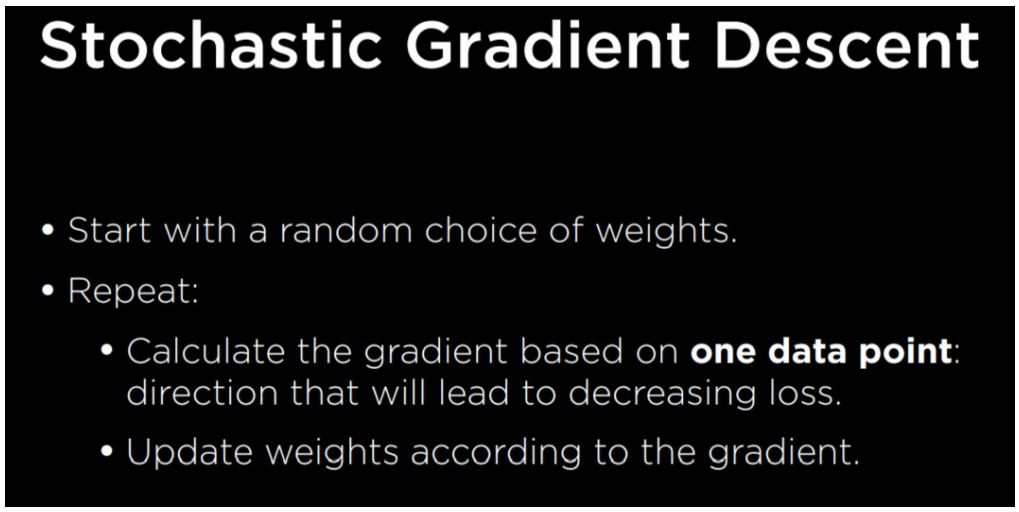


6.3. Gradient Descent

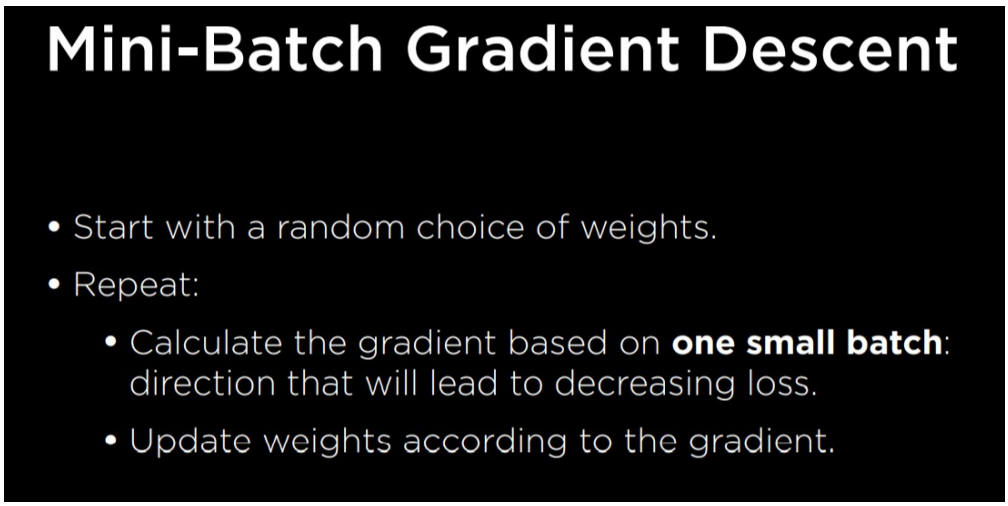
Gradient descent is an algorithm for minimizing loss when training neural networks. As was mentioned earlier, a neural network is capable of inferring knowledge about the structure of the network itself from the data. Whereas, so far, we defined the different weights, neural networks allow us to compute these weights based on the training data. To do this, we use the gradient descent algorithm, which works the following way:



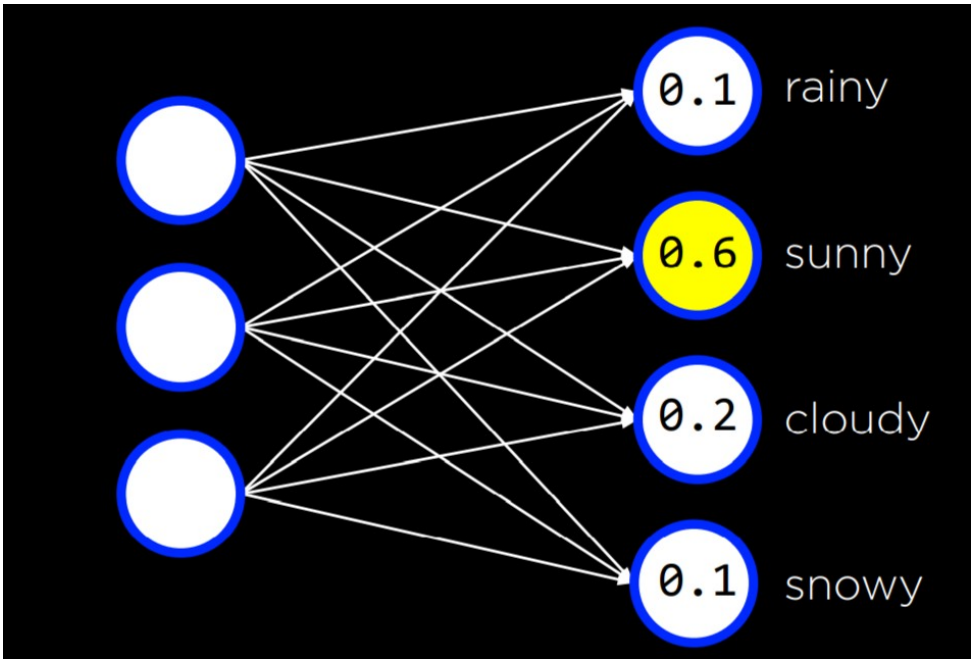
The problem with this kind of algorithms is that it requires to look at alll the data points multiple times which is computationally costly. To solve this we have alternatives like:



or:



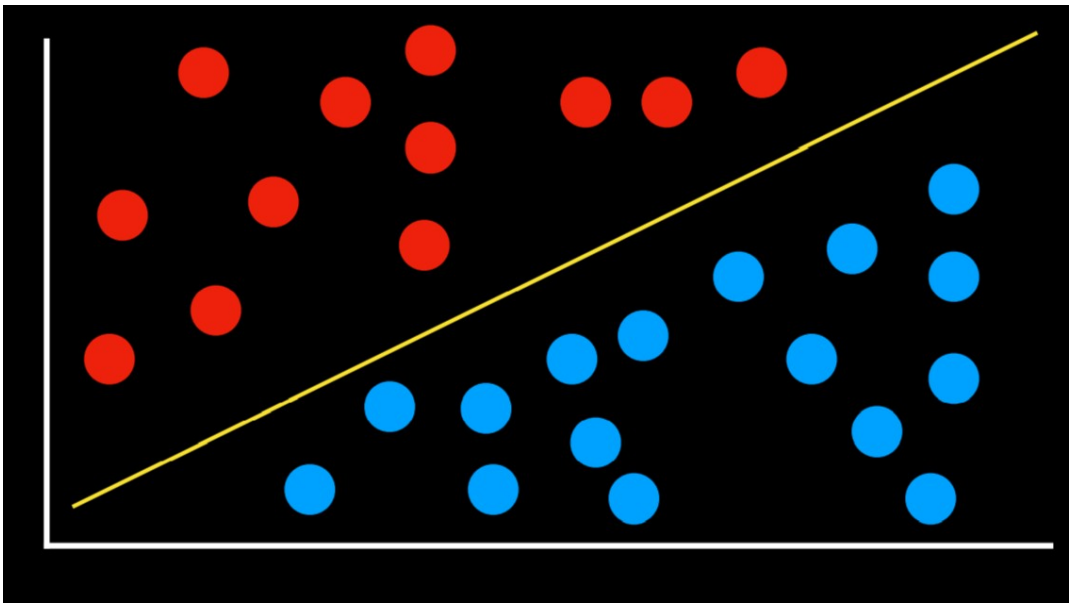
Using gradient descent, it is possible to find answers to many problems. For example, we might want to know more than "will it rain today?" We can use some inputs to generate probabilities for different kinds of weather, and then just choose the weather that is most probable:



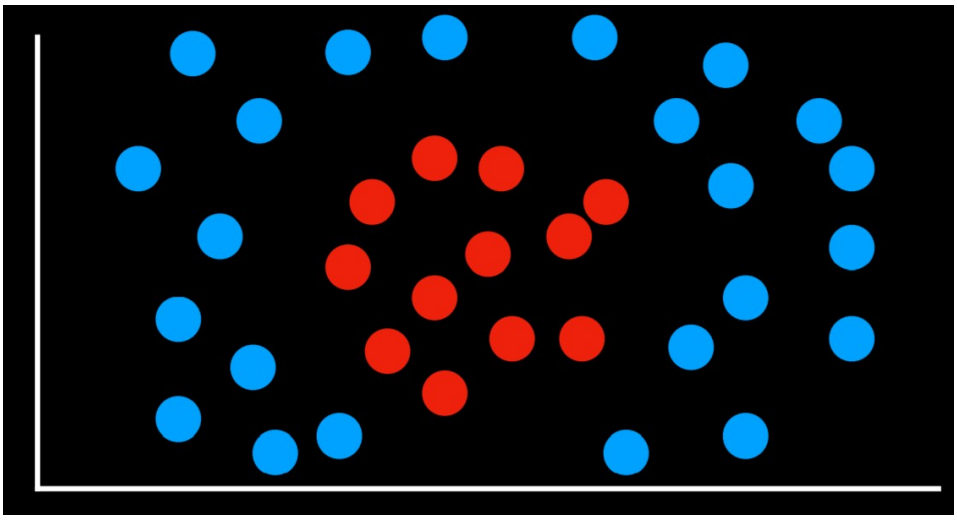
This can be done with any number of inputs and outputs, where each input is connected to each output, and where the outputs represent decisions that we can make.

Note that in this kind of neural networks the outputs are not connected. This means that each output and its associated weights from all the inputs can be be seen as an individual neural network and thus can be trained separately from the rest of the outputs.

So far, our neural networks relied on **perceptron** output units. These are units that are only capable of learning a linear decision boundary, using a straight line to separate data:



That is, based on a linear equation, the perceptron could classify an input to be one type or another. However, often, data are not linearly separable:



In this case, we turn to multilayer neural networks to model data non-linearly.

6.4. Multilayer Neural Networks

A multilayer neural network is an artificial neural network with an input layer, an output layer, and at least one **hidden layer**. While we provide inputs and outputs to train the model, we, the humans, don't provide any values to the units inside the hidden layers.

Each unit in the first hidden layer receives a weighted value from each of the units in the input layer, performs some action on it and outputs a value. Each of these values is weighted and further propagated to the next layer, repeating the process until the output layer is reached.

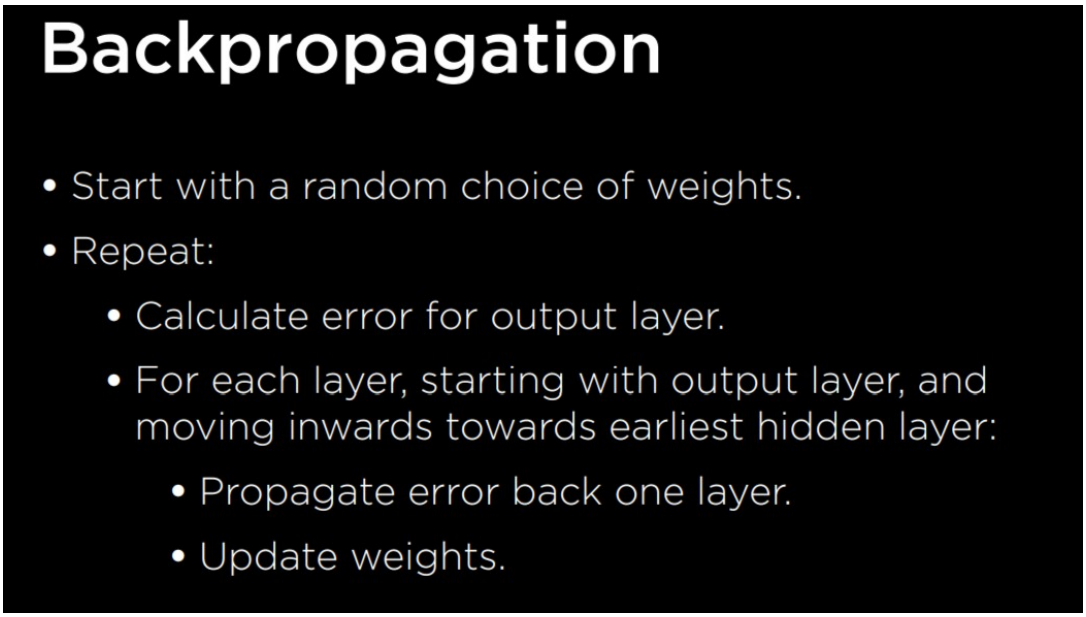
Through hidden layers, it is possible to model non-linear data.

5.4.1. Backpropagation

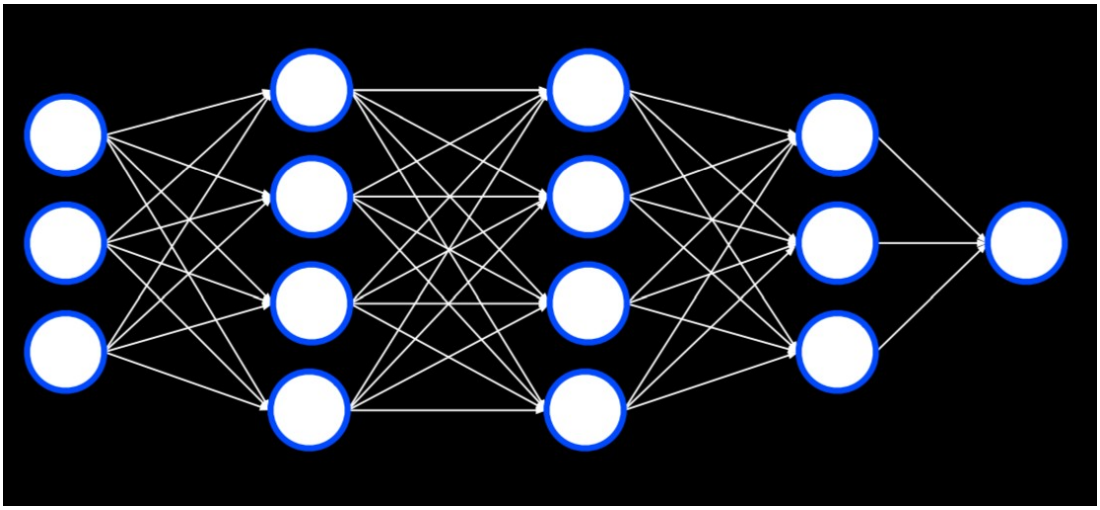
Backpropagation is the main algorithm used for training neural networks with hidden layers. It does so by starting with the errors in the output units, calculating the gradient descent for the weights of the previous layer, and repeating the process until the input layer is reached.

Note: (History: people thought if you know what the error or the losses on the output node, well, then based on what these weights are-- if one of these weights is higher than another-- you can calculate an estimate for how much the error from this node was due to this part of the hidden node, or this part of the hidden layer, or this part of the hidden layer, based on the values of these weights, in effect saying, that based on the error from the output, I can backpropagate the error and figure out an estimate for what the error is for each of these the hidden layer as well).

Pseudocode:

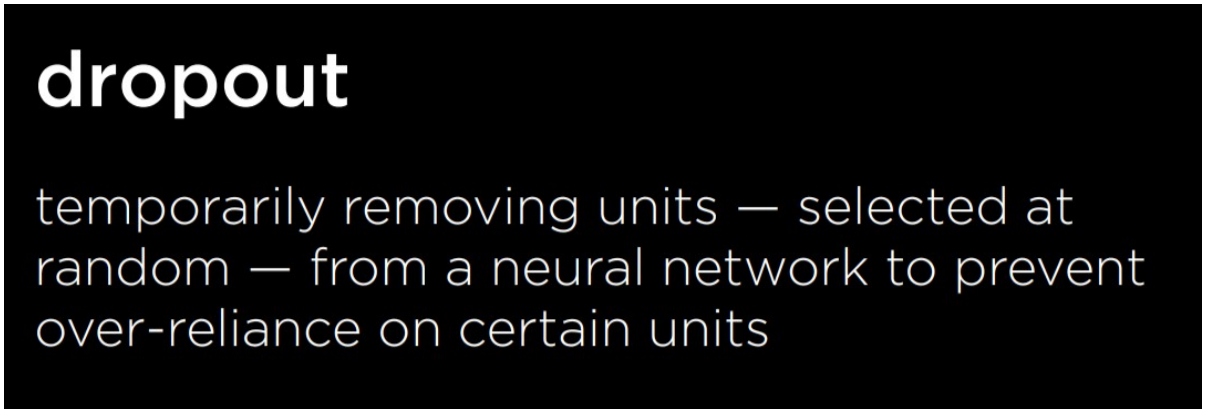


This can be extended to any number of hidden layers, creating **deep neural networks,** which are neural networks that have more than one hidden layer.



6.5. Overfitting

Overfitting also happens in neural networks, one way to prevent this is by using **dropout.**



6.6. Computer Vision

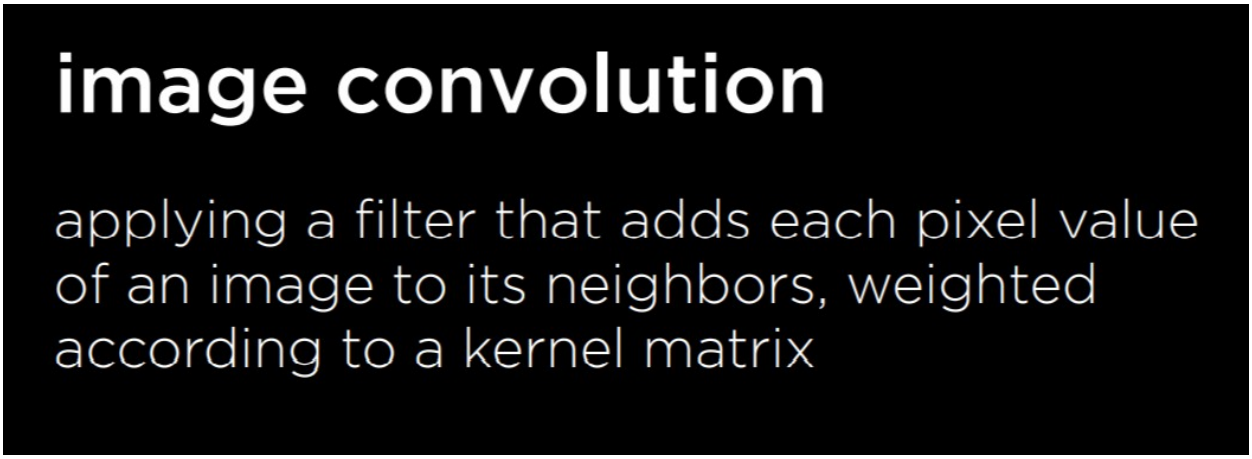
Computer vision encompasses the different computational methods for analyzing and understanding digital images, and it is often achieved using neural networks. For example, computer vision is used when social media employs face recognition to automatically tag people in pictures. Other examples are handwriting recognition and self-driving cars.

We can create a neural network where each color value in each pixel is an input, where we have some hidden layers, and the output is some number of units that tell us what it is that was shown in the image. However, there are a few drawbacks to this approach.

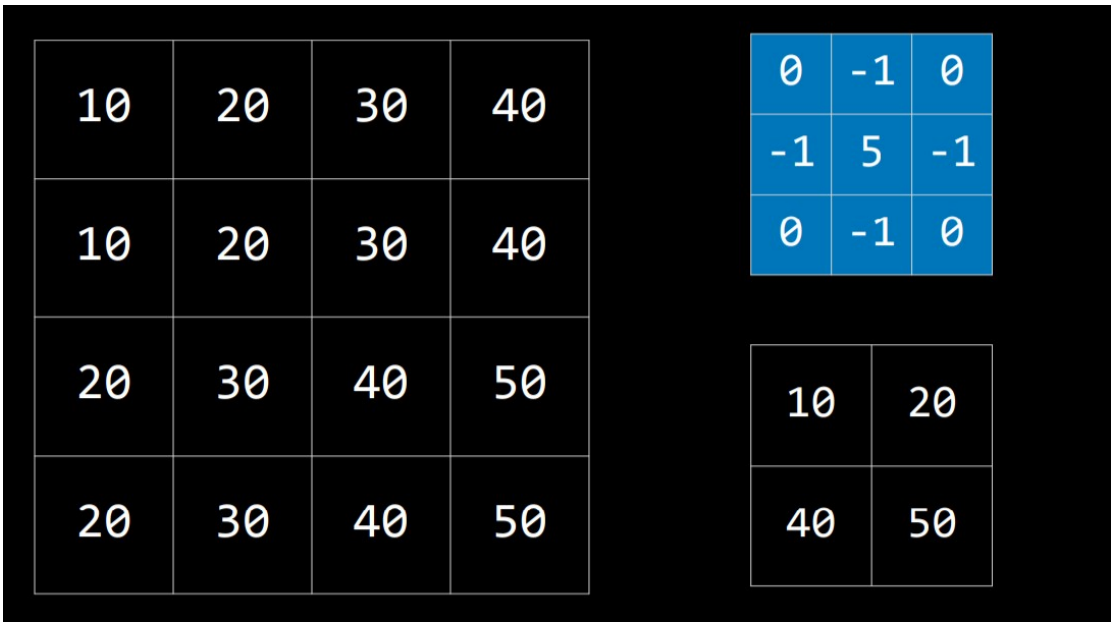
First, by breaking down the image into pixels and the values of their colors, we can't use the structure of the image as an aid, like humans do. That is, as humans, if we see a part of a face we know to expect to see the rest of the face, and this quickens computation.

We want to be able to use a similar advantage in our neural networks. Second, the sheer number of inputs is very big, which means that we will have to calculate a lot of weights.

6.6.1. Image Convolution



For example:



The kernel is the blue matrix, and the image is the big matrix on the left. The resulting filtered image is the small matrix on the bottom right. To filter the image with the kernel, we start with the pixel with value 20 in the top-left of the image (coordinates 1,1).

Then, we will multiply all the values around it by the corresponding value in the kernel and sum them up (10*0 + 20*(-1) + 30*0 + 10*(-1) + 20*5 + 30*(-1) + 20*0 + 30*(-1) + 40*0), producing the value 10.

Then we will do the same for the pixel on the right (30), the pixel below the first one (30), and the pixel to the right of this one (40). This produces a filtered image with the values we see on the bottom right.

Different kernels can achieve different tasks. For edge detection the following kernel is often used:

-1	-1	-1
-1	8	-1
-1	-1	-1

The idea here is that when the pixel is similar to all its neighbors, they should cancel each other, giving a value of 0. Therefore, the more similar the pixels, the darker the part of the image, and the more different they are the lighter it is. Applying this kernel to an image results in an image with pronounced edges.

Still, processing the image in a neural network is computationally expensive due to the number of pixels that serve as input to the neural network.

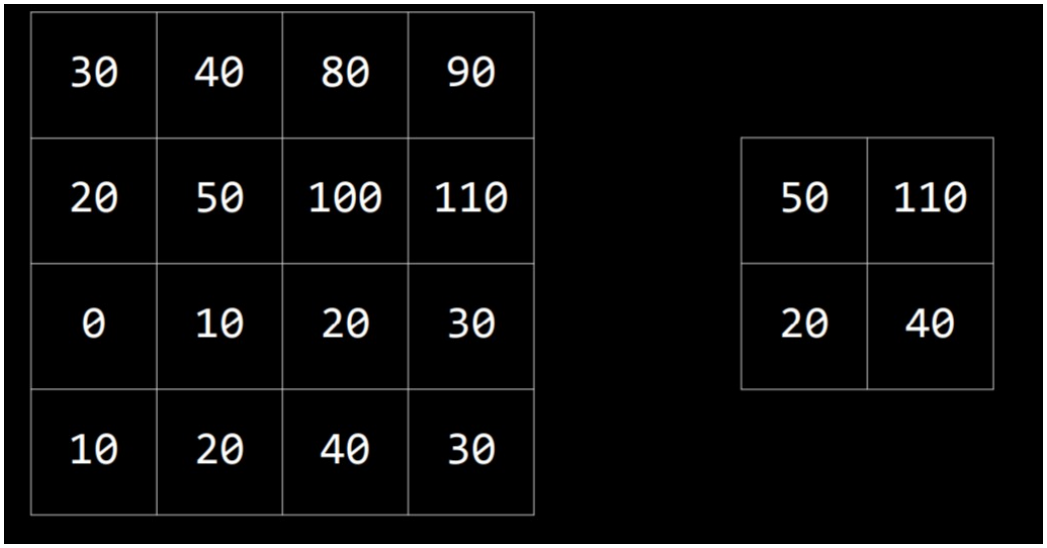
6.6.1.1. Pooling

Realistically speaking though, if you've got a really big image, that poses a couple of problems. One, it means a lot of input going into the neural network, but two, it also means that we really have to care about what's in each particular pixel, whereas realistically we often, if you're looking at an image, you really just care about whether there is a particular feature in some region of the image, and maybe you don't care about exactly which pixel it happens to be.

One way to approach this problem is using **pooling** where the size of the input is reduced by sampling from regions in the input. Pixels that are next to each other belong to the same area in the image, which means that they are likely to be similar.

Therefore, we can take one pixel to represent a whole area. And in particular, one of the most popular types of pooling is called **Max-Pooling**, in Max-Pooling the selected pixel is the one with the highest value of all others in the same region.

For example, if we divide the left square (below) into four 2X2 squares, by max-pooling from this input, we get the small square on the right:

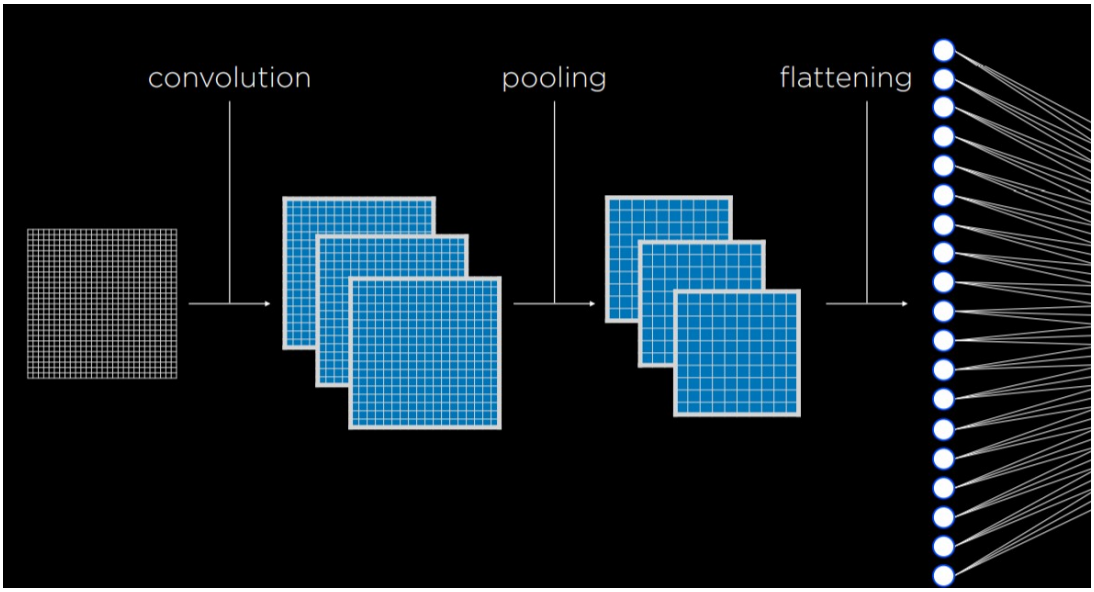


6.7. Convolutional Neural Networks

A convolutional neural network is a neural network that uses convolution, usually for analyzing images.

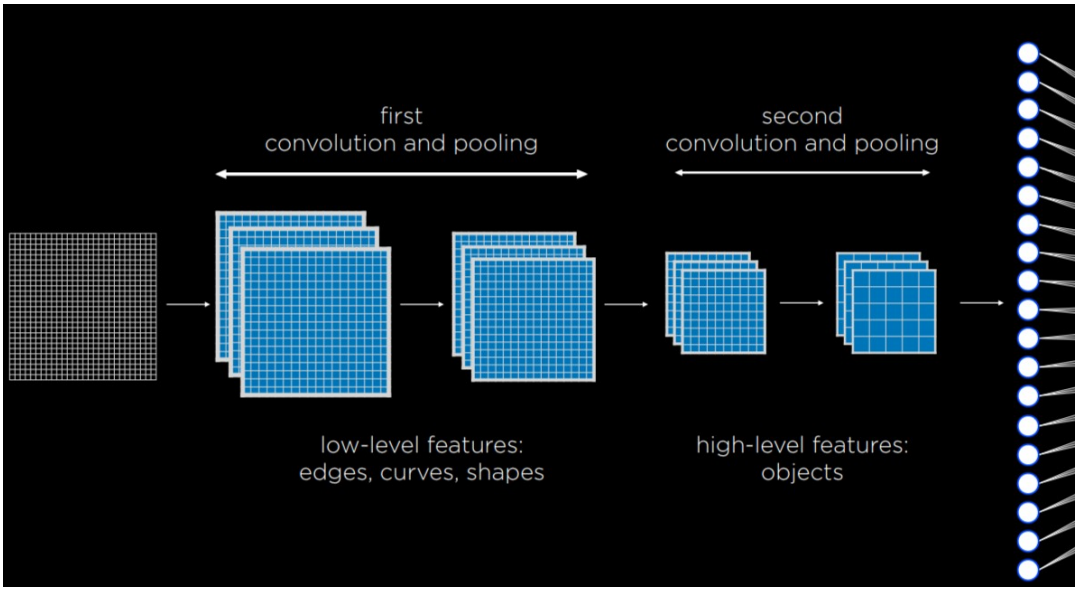
It starts by applying filters that can help distill some features of the image using different kernels. These filters can be improved in the same way as other weights in the neural network, by adjusting their kernels based on the error of the output.

Then, the resulting images are pooled, after which the pixels are fed to a traditional neural network as inputs (a process called **flattening**).



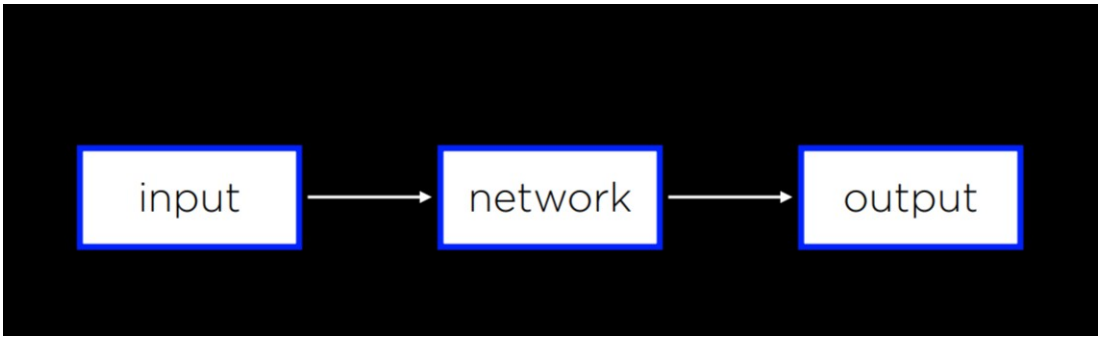
The convolution and pooling steps can be repeated multiple times to extract additional features and reduce the size of the input to the neural network. One of the benefits of these processes is that, by convoluting and pooling, the neural network becomes less sensitive to variation.

That is, if the same picture is taken from slightly different angles, the input for convolutional neural network will be similar, whereas, without convolution and pooling, the input from each image would be vastly different.

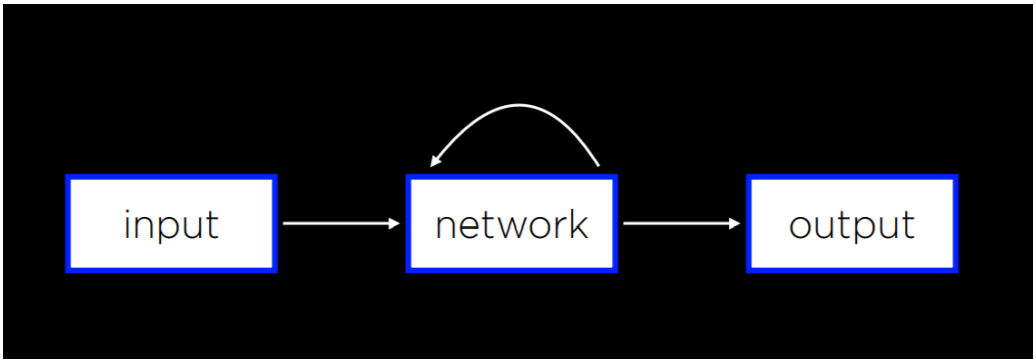


6.8. Recurrent Neural Networks

Feed-Forward Neural Networks are the type of neural networks that we have discussed so far, where input data is provided to the network, which eventually produces some output. A diagram of how feed-forward neural networks work can be seen below.

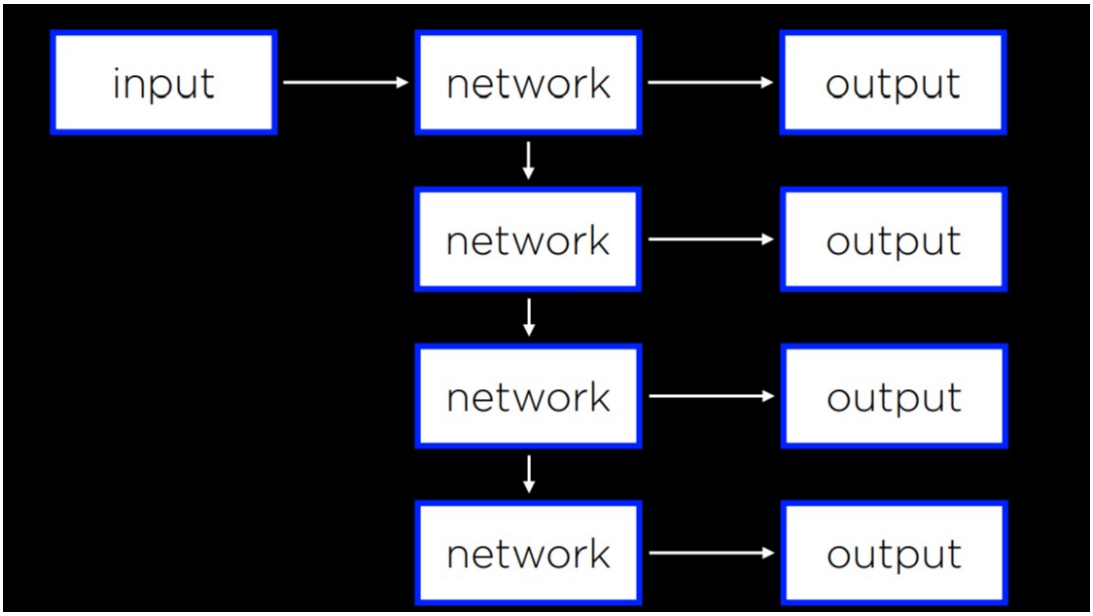


As opposed to that recurrent neural networks are neural networks that generates output that feeds back into its own inputs.



This is different from classification in that the output can be of varying length based on the properties of the image. While feed-forward neural networks are incapable of varying the number of outputs, recurrent neural networks are capable to do that due to their structure.

In the captioning task, a network would process the input to produce an output, and then continue processing from that point on, producing another output, and repeating as much as necessary.

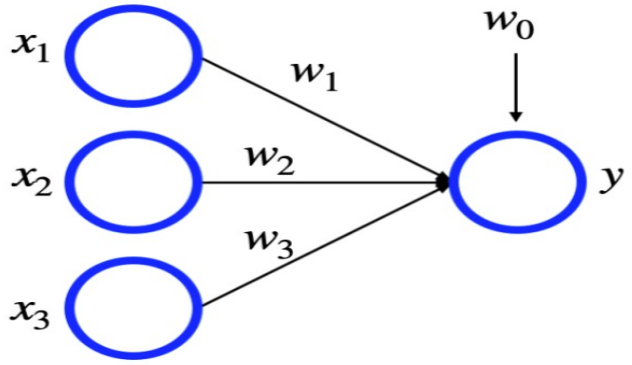


Recurrent neural networks are helpful in cases where the network deals with sequences and not a single individual object. Above, the neural network needed to produce a sequence of words. However, the same principle can be applied to analyzing video files, which consist of a sequence of images, or in translation tasks, where a sequence of inputs (words in the source language) is processed to produce a sequence of outputs (words in the target language).

6.9. Quiz

Detailed answers in this Week's source code.

The following question will ask you about the below neural network, where we set w0 = -5, w1 = 2, w2 = -1, and w3 = 3. x1, x2, and x3 represent input neurons, and y represents the output neuron.



✓ What value will this network compute for y given inputs x1 = 3, x2 = 2, and x3 = 4 if we use a step activation function? What if we use a ReLU activation function? *1/1

- ○ 0 for step activation function, 0 for ReLU activation function
- ○ 0 for step activation function, 1 for ReLU activation function
- ○ 1 for step activation function, 0 for ReLU activation function
- ○ 1 for step activation function, 1 for ReLU activation function
- ◉ 1 for step activation function, 11 for ReLU activation function ✓
- ○ 1 for step activation function, 16 for ReLU activation function
- ○ 11 for step activation function, 11 for ReLU activation function
- ○ 16 for step activation function, 16 for ReLU activation function

✓ How many total weights (including biases) will there be for a fully connected neural network with a single input layer with 3 units, a single hidden layer with 5 units, and a single output layer with 4 units? *1/1

- ○ 9
- ○ 12
- ○ 20
- ○ 35
- ○ 39
- ○ 40
- ◉ 44 ✓
- ○ 60
- ○ 69

✓ Consider a recurrent neural network that listens to a audio speech sample, and classifies it according to whose voice it is. What network architecture is the best fit for this problem? *1/1

- ○ One-to-many (single input, multiple outputs)
- ○ One-to-one (single input, single output)
- ◉ Many-to-one (multiple inputs, single output) ✓
- ○ Many-to-many (multiple inputs, multiple outputs)

The following question will ask you about a 4x4 grayscale image with the following pixel values.

2	4	6	8
16	14	12	10
18	20	22	24
32	30	28	26

✓ What would be the result of applying a 2x2 max-pool to the original image? *1/1

Answers are formatted as a matrix [[a, b], [c, d]] where [a, b] is the first row and [c, d] is the second row.

- (●) [[16, 12], [32, 28]] ✓
- ( ) [[16, 14], [32, 30]]
- ( ) [[22, 24], [32, 30]]
- ( ) [[14, 12], [30, 28]]
- ( ) [[16, 14], [22, 24]]
- ( ) [[16, 12], [32, 30]]

7. Language

Natural Language Processing: spans all tasks where the AI gets human language as input. The following are a few examples of such tasks:

- **automatic summarization:** where the AI is given text as input and it produces a summary of the text as output.
- **information extraction:** where the AI is given a corpus of text and the AI extracts data as output.
- **language identification:** where the AI is given text and returns the language of the text as output.
- **machine translation:** where the AI is given a text in the origin language and it outputs the translation in the target language.
- **named entity recognition:** where the AI is given text and it extracts the names of the entities in the text (for example, names of companies).
- **speech recognition:** where the AI is given speech and it produces the same words in text.
- **text classification:** where the AI is given text and it needs to classify it as some type of text.
- **word sense disambiguation:** where the AI needs to choose the right meaning of a word that has multiple meanings (e.g. bank means both a financial institution and the ground on the sides of a river).

7.1. Syntax and Semantics

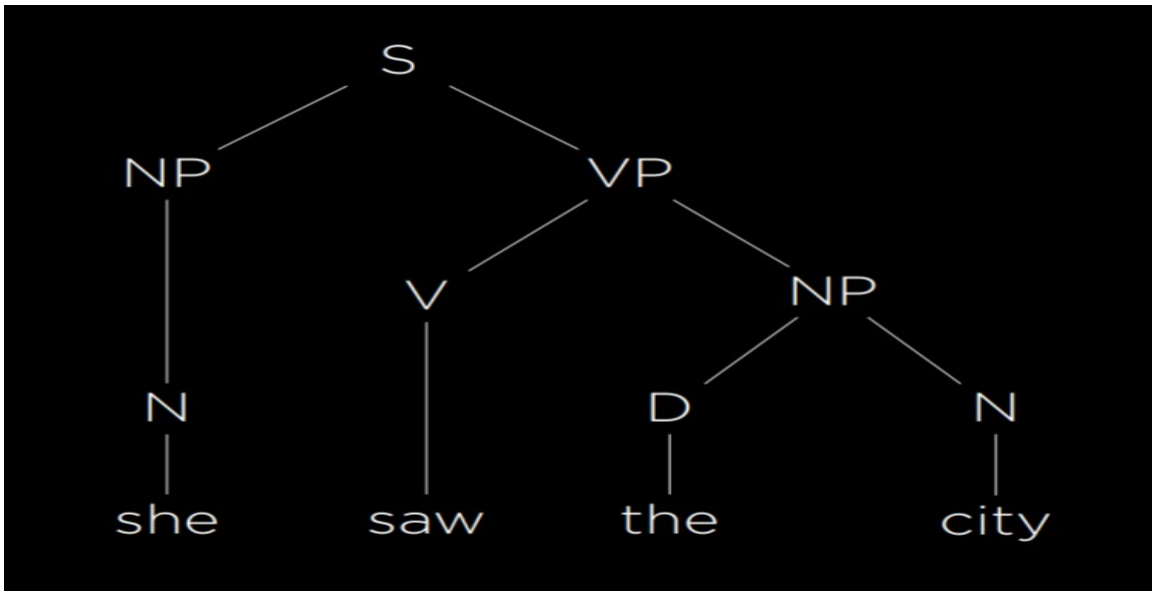
Syntax is sentence structure. To be able to parse human speech and produce it, the AI needs to command syntax. **Semantics** to be able to parse human speech and produce it, the AI needs to command semantics.

7.2. Context-Free Grammar

Formal Grammar is a system of rules for generating sentences in a language. In **Context-Free Grammar,** the text is abstracted from its meaning to represent the structure of the sentence using formal grammar. Let's consider the following example sentence:

She saw the city.

This is a simple grammatical sentence, and we would like to generate a syntax tree representing its structure (Remember from the lecture all the basic rules:



Using formal grammar, the AI is able to represent the structure of sentences. In the grammar we have described, there are enough rules to represent the simple sentence above. To represent more complex sentences, we will have to add more rules to our formal grammar.

7.3. n-grams

An n-gram is a sequence of n items from a sample of text. In a **character n-gram**, the items are characters, and in a **word n-gram** the items are words. A unigram, bigram, and trigram are sequences of one, two, and three items.

“How often have I said to you that when you have eliminated the impossible whatever remains, however improbable, must be the truth?”

In that sentence, the first three n-grams are “how often have,” “often have I,” and “have I said.”

n-grams are useful for text processing. While the AI hasn’t necessarily seen the whole sentence before, it sure has seen parts of it, like “have I said.” Since some words occur together more often then others, it is possible to also predict the next word with some probability.

For example, your smartphone suggests words to you based on a probability distribution derived from the last few words you typed. Thus, a helpful step in natural language processing is breaking the sentence into n-grams.

7.4. Tokenization

Tokenization is the task of splitting a sequence of characters into pieces (tokens). Tokens can be words as well as sentences, in which case the task is called **word tokenization** or **sentence tokenization**. We need tokenization to be able to look at *n*-grams, since those rely on sequences of tokens.

We start by splitting the text into words based on the space character. While this is a good start, this method is imperfect because we end up with words with punctuation, such as “remains,”.

So, for example, we can remove punctuation. However, then we face additional challenges, such as words with apostrophes (e.g. “o’clock”) and hyphens (e.g. “pearl-grey). Additionally, some punctuation is important for sentence structure, like periods. However, we need to be able to tell apart between a period at the end of the word “Mr.” and a period in the end of the sentence.

Dealing with these questions is the process of tokenization. In the end, once we have our tokens, we can start looking at *n*-grams.

7.5. Markov Models

Markov models are also useful to generate text. To do so, we train the model on a text, and then establish probabilities for every *n*-th token in an *n*-gram based on the *n* words preceding it. For example, using trigrams, after the Markov model has two words, it can choose a third one from a probability distribution based on the first two. Then, it can choose a fourth word from a probability distribution based on the second and third words.

Eventually, using Markov models, we are able to generate text that is often grammatical and sounding superficially similar to human language output. However, these sentences lack actual meaning and purpose.

7.6. Text Categorization

7.6.1. Bag-of-words Model

Bag-of-words is a model that represents text as an unordered collection of words. This model ignores syntax and considers only the meanings of the words in the sentence. This approach is helpful in some classification tasks, such as sentiment analysis (another classification task would be distinguishing regular email from spam email). Sentiment analysis can be used, for instance, in product reviews, categorizing reviews as positive or negative. Consider the following sentences:

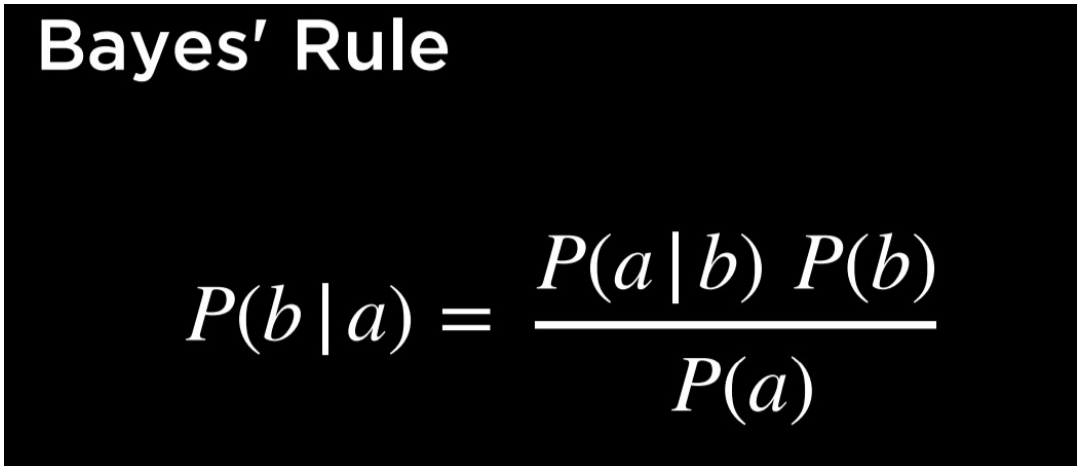
1. “My grandson loved it! So much fun!”
2. “Product broke after a few days.”
3. “One of the best games I’ve played in a long time.”
4. “Kind of cheap and flimsy, not worth it.”

Based only on the words in each sentence and ignoring the grammar, we can see that sentences 1 and 3 are positive (“loved,” “fun,” “best”) and sentences 2 and 4 are negative (“broke,” “cheap,” “flimsy”).

7.6.2. Naive Bayes

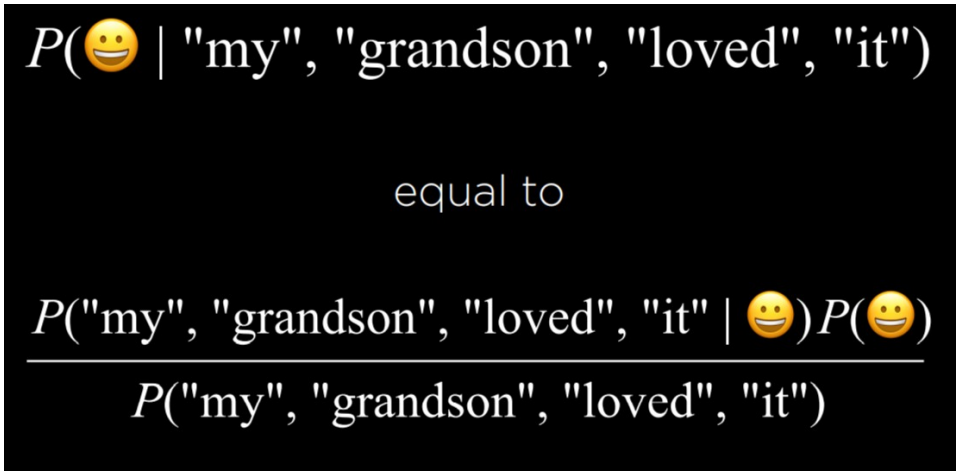
Naive Bayes is a technique that's can be used in sentiment analysis with the bag-of-words model. In sentiment analysis, we are asking "What is the probability that the sentence is positive/negative given the words in the sentence."

Answering this question requires computing conditional probability, and it is helpful to recall Bayes' rule from lecture 2:



Now, we would like to use this formula to find P(sentiment | text), or, for example, P(positive | "my grandson loved it"). We start by tokenizing the input, such that we end up with P(positive | "my", "grandson", "loved", "it").

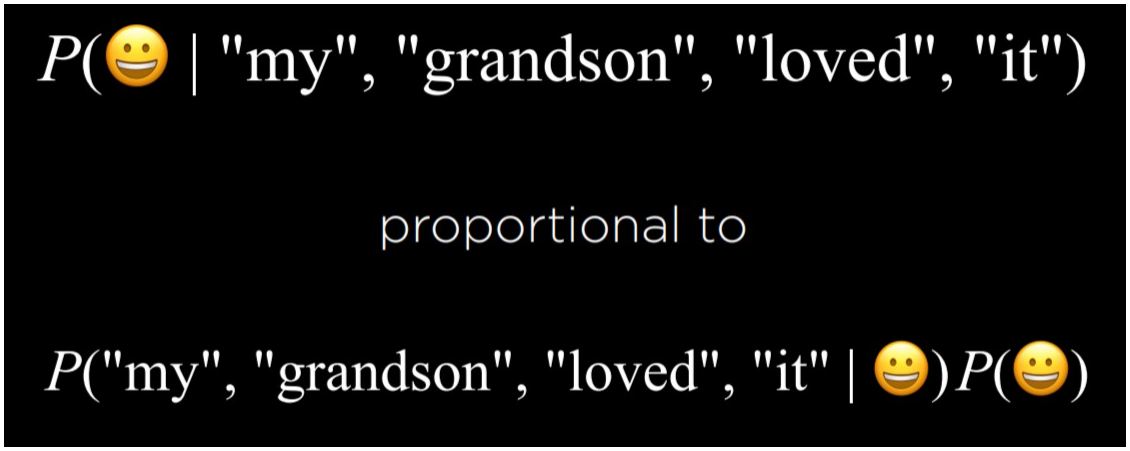
Applying Bayes' ruled directly, we get the following expression:



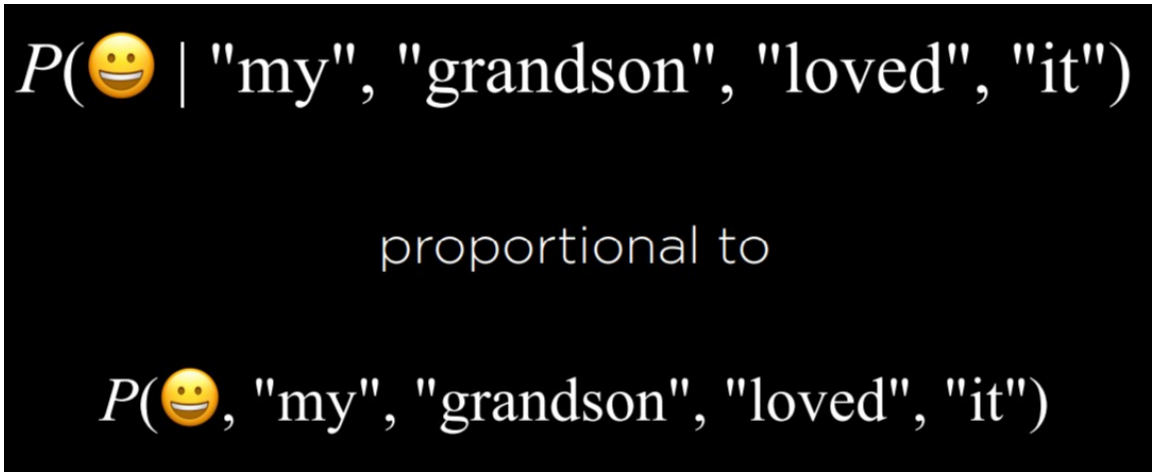
This complicated expression will give us the precise answer to P(positive | “my”, “grandson”, “loved”, “it”).

However, we can simplify the expression if we are willing to get an answer that’s not equal, but proportional to P(positive | “my”, “grandson”, “loved”, “it”). Later on, knowing that the probability distribution needs to sum up to 1, we can normalize the resulting value into an exact probability.

This means that we can simplify the expression above to the numerator only:

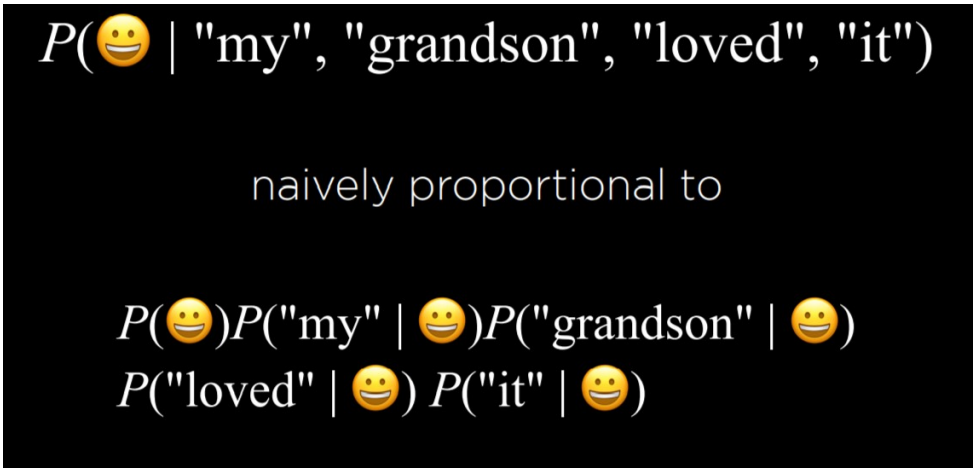


Again, we can simplify this expression based on the knowledge that a conditional probability of *a* given *b* is proportional to the joint probability of *a* and *b*. Thus, we get the following expression for our probability:



Calculating this joint probability, however, is complicated, because the probability of each word is conditioned on the probabilities of the words preceding it. It requires us to compute P(positive)*P(“my” | positive)*P(“grandson” | positive, “my”)*P(loved | positive, “my”, “grandson”)*P(“it” | positive, “my”, “grandson”, “loved”).

Here is where we use Bayes’ rules naively: we assume that the probability of each word is independent from other words. This is not true, but despite this imprecision, Naive Bayes’ produces a good sentiment estimate. Using this assumption, we end up with the following probability:



each one of these is easier to calculate, for example:

$$P(😀) = \frac{\text{number of positive samples}}{\text{number of total samples}}$$

$$P(\text{"loved"} \mid 😀) = \frac{\text{number of positive samples with "loved"}}{\text{number of positive samples}}$$

$$P(😕)P(\text{"my"} \mid 😕)P(\text{"grandson"} \mid 😕)$$
$$P(\text{"loved"} \mid 😕)\ P(\text{"it"} \mid 😕)$$

😀	😕
0.49	0.51

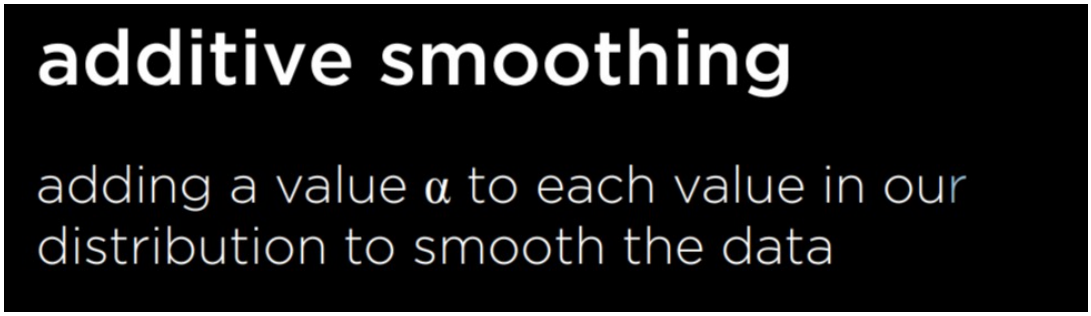
	😀	😕
my	0.30	0.20
grandson	0.01	0.02
loved	0.32	0.08
it	0.30	0.40

😀 `0.00014112`

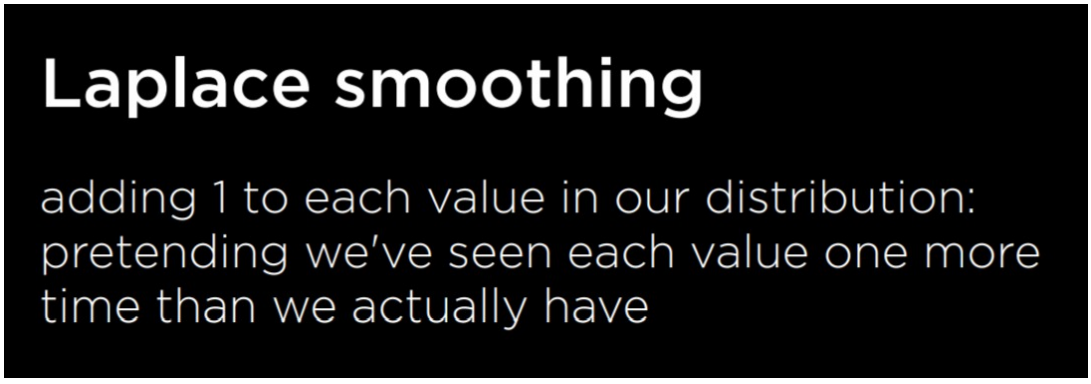
😕 `0.00006528`

At this point, they are in proportion to each other, but they don't tell us much in terms of probabilities. To get the probabilities, we need to normalize the values, arriving at P(positive) = 0.6837 and P(negative) = 0.3163. The strength of naive Bayes is that it is sensitive to words that occur more often in one type of sentence than in the other.

One problem that we can run into is that some words may never appear in a certain type of sentence. Suppose none of the positive sentences in our sample had the word "grandson." Then, P("grandson" | positive) = 0, and when computing the probability of the sentence being positive we will get 0. However, this is not the case in reality (not all sentences mentioning grandsons are negative). One way to go about this problem is with **Additive Smoothing:**



One type of additive smoothing is the **Laplace Smoothing:**



7.7. Information Retrieval

7.7.1. tf-idf

Information retrieval is the task of finding relevant documents in response to a user query. To achieve this task, we use **topic modeling** to discover the topics for a set of documents. How can the AI go about extracting the topics of documents? One way to do so is by looking at **term frequency**, which is simply counting how many times a term appears in a document. The idea behind this is that key terms and important ideas are likely to repeat.

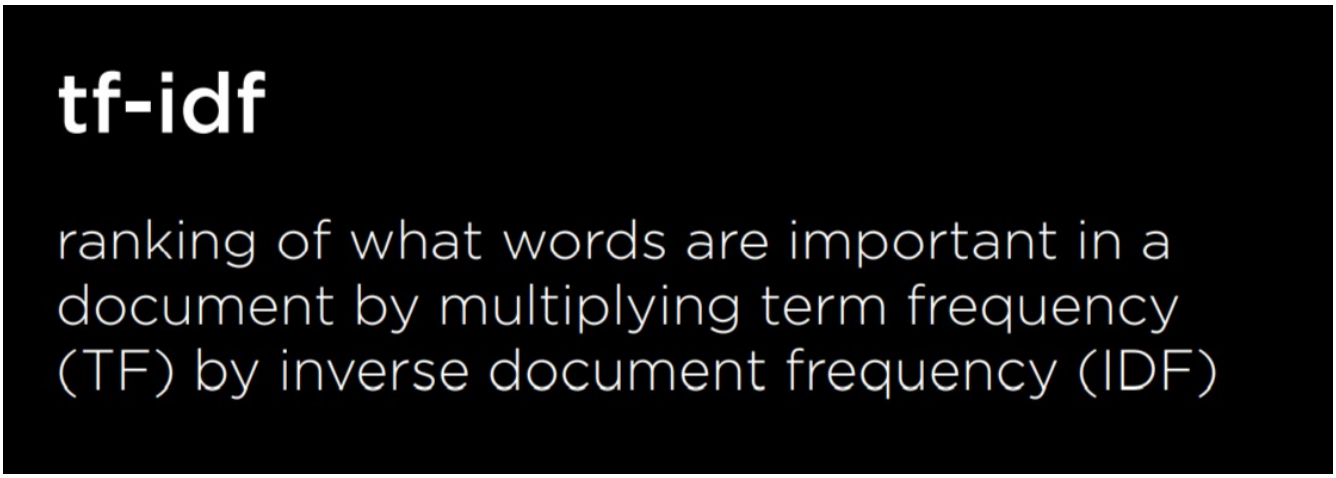
When you ran a counting algorithm naively, the top-5 most common words are likely to be just **function words,** words that have little meaning on their own, but are used to grammatically connect other words (am, by, do, is, which, with, yet…), the words that we are likely after are **content words**, words that carry meaning independently (such as: “crime,” “brothers,” “demons,” “gentle,” “meek,” etc).

Another problem that you may run into are irrelevant words, for example the sherlock holmes case, when you ran a counting algorithm excluding function words it’s very probable that a lot of words like “Holmes” or “Sherlock” are going to appear, which are not what we’re looking for.

This brings us to the idea of **Inverse Document Frequency:**

To calculate it we use the equation:

This is where the name of the library comes from: tf-idf stand for Term Frequency — Inverse Document Frequency:



7.8. Information Extraction

Information Extraction is the task of extracting knowledge from documents. So far, treating text with the bag-of-words approach was helpful when we wanted the AI to perform simple tasks, such as recognizing sentiment in a sentence as positive or negative, or retrieving the key words in a document.

However, for a task like information extraction, where we need the AI to understand the connections between different words, the bag-of-words approach won't get us very far.

A possible task of information extraction can take the form of giving a document to the AI as input and getting a list of companies and the years when they were founded as output. One way we can go about this is providing the AI with a template, such as "When {company} was founded in {year}." This will not get us perfect results, because not all information about companies and their years of foundation are written in precisely this format.

Still, if the dataset is large enough, we will definitely come across sentences of precisely this form, which will allow the AI to extract this knowledge. A more advanced approach would be to give the AI an abstracted example like "Facebook, 2004," and let it develop its own model to extract the data. By going through enough data, the AI will be able to infer possible templates for extracting information similar to the example.

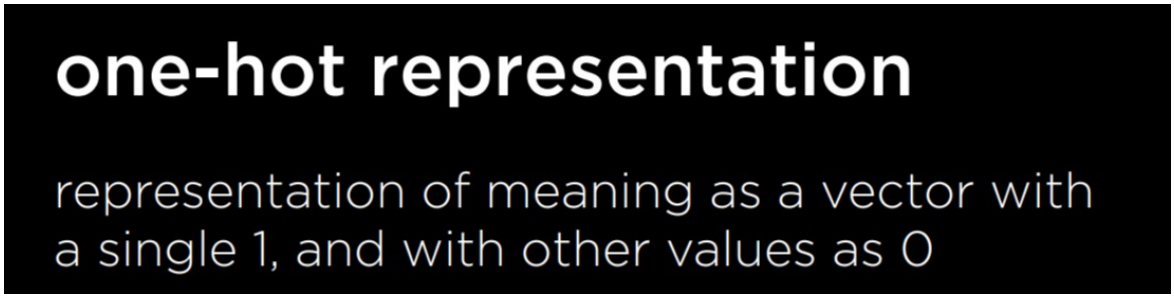
Using templates, even if self-generated by the AI, is helpful in tasks like Information Extraction. However, if we want the AI to be able to produce text like a human, the AI needs to be able to understand not just templates, but how all words in the language relate to each other in their meanings.

7.9. Word Net

Wordnet is a database similar to a dictionary, where words are given definitions as well as broader categories. For example, the word “house” will yield multiple definitions, one of which is “a dwelling that serves as living quarters for one or more families.” This definition is paired with the categories “building” and “dwelling.” Having access to these broader categories will allow us to make connections between words that are semantically similar.

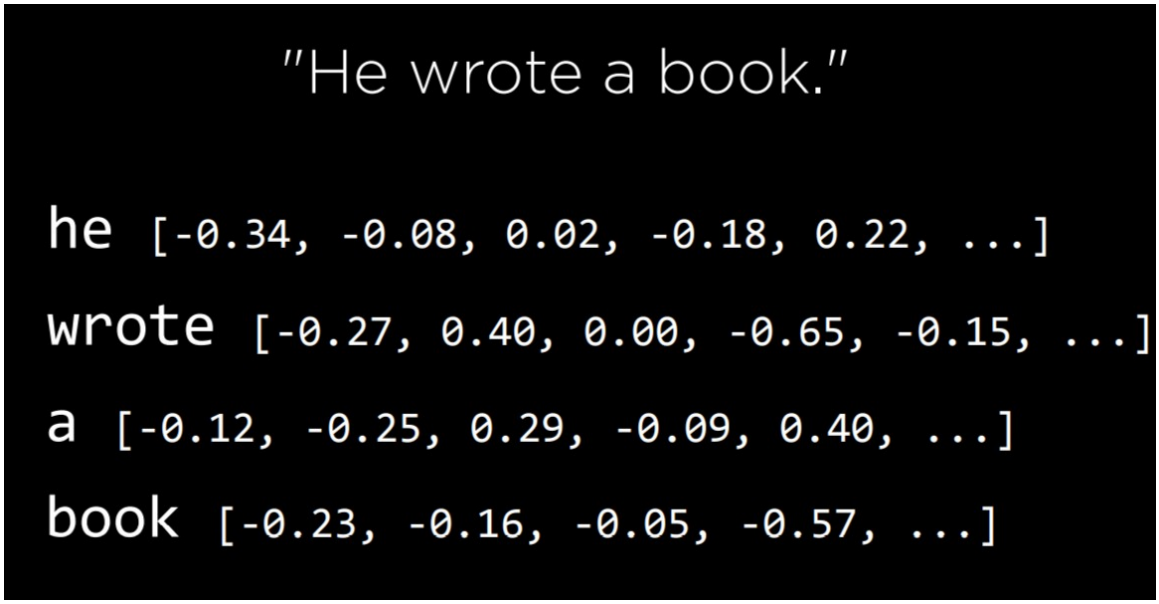
7.10. Word Representation

We want to represent word meanings in our AI. As we’ve seen before, it is convenient to provide input to the AI in the form of numbers. One way to go about this is by using **One-Hot Representation:**



There are clear limitations to this approach however, because if we have a dictionary with for example, 50000 words, we will end up with 50000 vectors of length 50000. This is incredibly inefficient. Another problem in this kind of representation is that we are unable to represent similarity between words like “wrote” and “authored.”

Instead, we turn to the idea of **Distributed Representation:**



Each vector has a limited number of values (much less than 50,000), This allows us to generate unique values for each word while using smaller vectors. Additionally, now we are able to represent similarity between words by how different the values in their vectors are.

“You shall know a word by the company it keeps” is an idea by J. R. Firth, an English linguist. Following this idea, we can come to define words by their adjacent words. For example, there are limited words that we can use to complete the sentence “for ___ he ate.” These words are probably words like “breakfast,” “lunch,” and “dinner.”

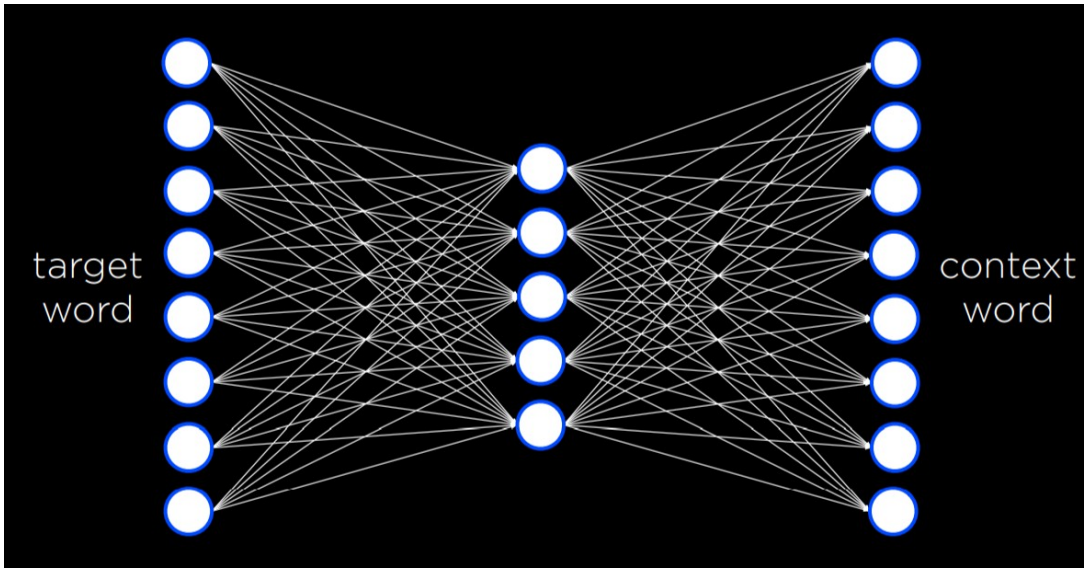
This brings us to the conclusion that by considering the environment in which a certain word tends to appear, we can infer the meaning of the word.

7.11. Word2vec

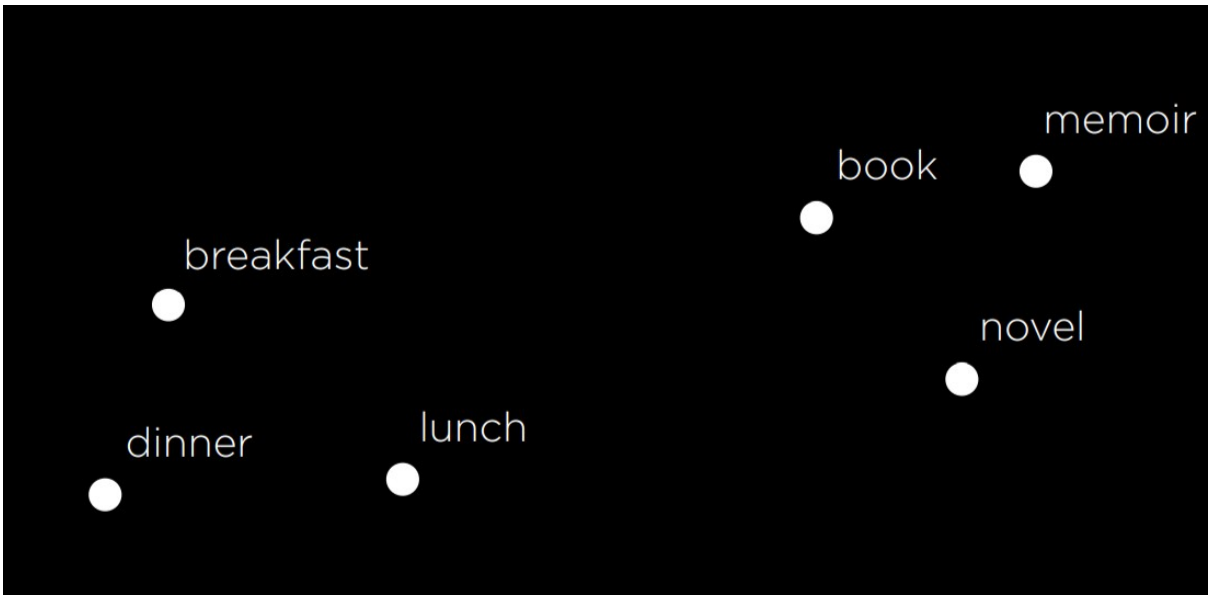
word2vec is an algorithm for generating distributed representations of words. It does so by **Skip-Gram Architecture:**

In this architecture, the neural network has an input unit for every target word. A smaller, single hidden layer (e.g. 50 or 100 units, though this number is flexible) will generate values that represent the distributed representations of words. Every unit in this hidden layer is connected to every unit in the input layer.

The output layer will generate words that are likely to appear in a similar context as the target words. Similar to what we saw in last lecture, this network needs to be trained with a training dataset using the backpropagation algorithm.

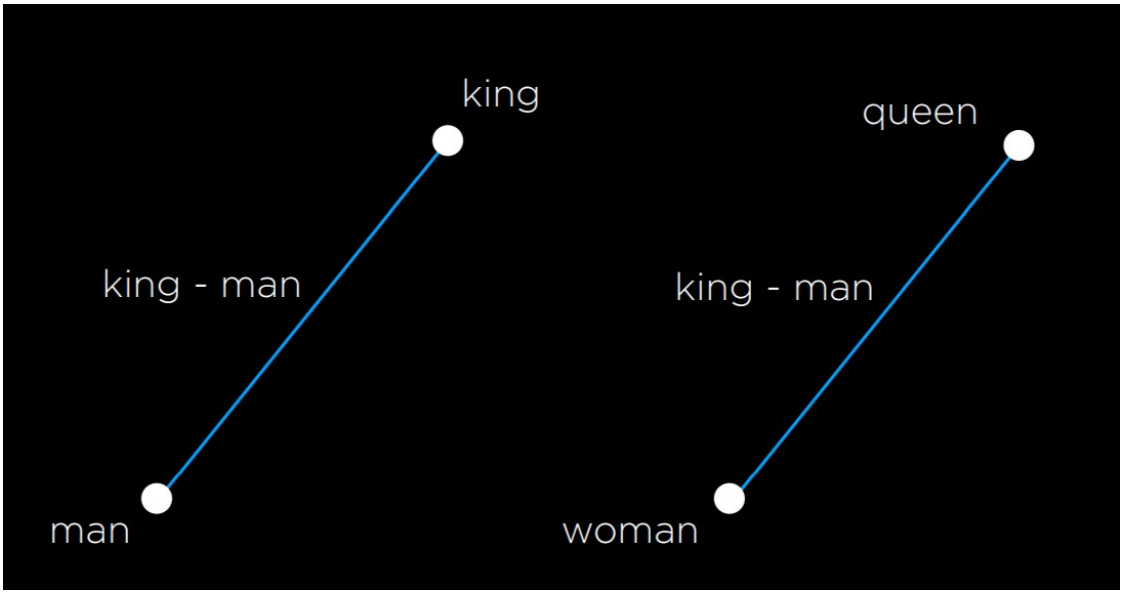


So this algorithm will represent each word as an array of values, these values doesn't really mean much, but by assigning close vector values to words related, the algorithm actually does something graphically similar to this:



We can also compute the difference between words based on how different their vectors are. For example, the difference between *king* and *man* is similar to the difference between *queen* and *woman*.

That is, if we add the difference between *king* and *man* to the vector for *woman*, the closest word to the resulting vector is *queen*!



Similarly, if we add the difference between *ramen* and *japan* to *america*, we get *burritos*. By using neural networks and distributed representations for words, we get our AI to understand semantic similarities between words in the language, bringing us one step closer to AIs that can understand and produce human language.

7.12. Quiz

The following question will ask you about the below context-free grammar, where S is the start symbol.

```
S -> NP V
NP -> N | A NP
A -> "small" | "white"
N -> "cats" | "trees"
V -> "climb" | "run"
```

The following question will also ask you about the following four sentences.
Sentence 1: Cats run.
Sentence 2: Cats climb trees.
Sentence 3: Small cats run.
Sentence 4: Small white cats climb.

✓ Of the four sentences above, which sentences can be derived from the above context-free grammar? *1/1

- ○ Only Sentence 1
- ○ Only Sentence 1 and Sentence 2
- ○ Only Sentence 1 and Sentence 3
- ○ Only Sentence 1 and Sentence 4
- ○ Only Sentence 1, Sentence 2, and Sentence 3
- ○ Only Sentence 1, Sentence 2, and Sentence 4
- ◉ Only Sentence 1, Sentence 3, and Sentence 4 ✓
- ○ All four sentences
- ○ None of the four sentences

The following question will ask you about a corpus with the following documents.
Document 1: a a b c
Document 2: a c c c d e f
Document 3: a c d d d
Document 4: a d f

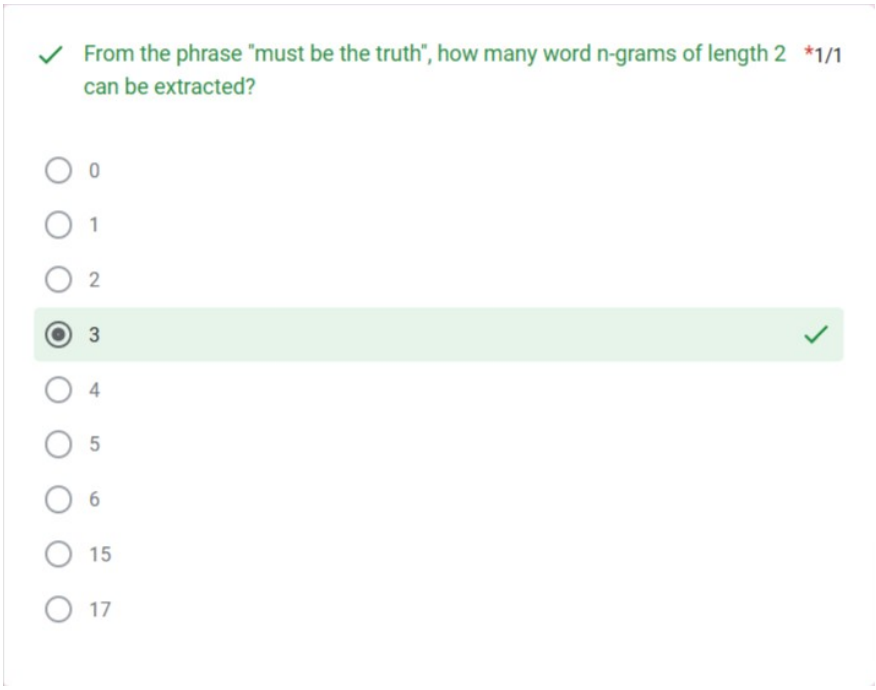
✓ What is the tf-idf value for "d" in Document 3? * 1/1

Round answers to two decimal places. Use the natural logarithm (log base e) when taking a logarithm.

- ○ 0.00
- ○ 0.57
- ○ 0.69
- ◉ 0.86 ✓
- ○ 2.07
- ○ 3.47
- ○ 6.00

✓ Why is "smoothing" useful when applying Naive Bayes? * 1/1

- ○ Smoothing allows Naive Bayes to be less "naive" by not assuming that evidence is conditionally independent.
- ○ Smoothing allows Naive Bayes to turn a conditional probability of evidence given a category into a probability of a category given evidence.
- ◉ Smoothing allows Naive Bayes to better handle cases where evidence has never appeared for a particular category. ✓
- ○ Smoothing allows Naive Bayes to better handle cases where there are many categories to classify between, instead of just two.



END INTRODUCTION