1 Artificial Intelligence

- An **agent** is an entity that can perceive and act. This course is about designing rational agents.
- Rational behavior: doing the right thing.
- Environment Types: Fully observable; Deterministic; Episodic; Static, Discrete; Single-agent. The counter part: partially observable; stochastic; sequential; dynamic; continuous; multi-agent.
- An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.
- Being rational means maximizing your **expected utility**. And a better title for this course would be **Computational Rationality**.
- **Rational:** maximally achieving pre-defined goals.
- **Rationality:** only concerns what decisions are made (not the thought process behind them)
- **Utility:** Goals are expressed in the terms of the utility of outcomes. And CS188 thinks that being rational means maximizing your expected utility.
- **Automation:** Applied AI involves many kinds of automation. Scheduling, route planning, medical diagnosis, web search engines, spam classifiers, automated help desks, fraud detection, product recommendations. "Did any one of these remind you of subtopics and projects in Machine Learning?"
- **Agent:** An agent is an entity that perceives and acts. A rational agent selects actions that maximize its *expected* utility.
- Making decision, reasoning under Uncertainty, and their applications.

2 Problem Solving

- A search problem consists of
  - a state space
  - a successor function (namely **update function** in data mining algorithm series, more namely **recursion** in bullshit technology.)
  - a start state (**initial value**), goal test (**terminating value**) and path cost function (**we say weights in Graph Theory**)
  - Does any one of the above reminds you of **recursion**?
- Problems are often modelled as a state space, a set of states that a problem can be in. The set of states forms a graph where two states are **connected** if there is an operation that can be performed to transform the first state into the second.
- A solution is a sequence of actions (a plan) which transforms the start state to a goal state.
- **State space graph**: A mathematical representation of a search problem.
- **Search Trees**
  - This is a "what-if" tree of plans and outcomes
  - For most problems, we can never actually build the whole tree
- **General Tree Search** Frontier; Expansion; Exploration Strategy.
- **States vs. Nodes** Nodes in state space graphs are problem states; Nodes in search trees are plans. The same problem state may be achieved by multiple search tree nodes.
- **Graph Search** Graph Search still produces a search tree; Graph search is almost always better than tree search.
- DFS graph search needs to store "explored set", which is $O(b^m)$. However, **DFS is optimal** when the search tree is finite, all action costs are identical and all solutions have the same length. However limitating this may sound, there is an important class of problems that satisfies these conditions: the CSPs (constraint satisfaction problems). Maybe all the examples you thought about fall in this (rather common) category.
- The breadth first search and iterative deepening are conceptually and computationally the same. The only difference is the "space" (we call them **memory**) would be partially saved by iterative deepening search.
- **Heuristics** estimate of how close a node is to a goal; Designed for a particular search problem.
- **A star search** Uniform-cost orders by **path cost**, or backward cost $g(n)$; Best-first orders by **distance** to goal, or forward cost $h(n)$. A* Search orders by the sum: $f(n) = g(n) + h(n)$. The distance is an estimated one.
- When A* terminates its search, it has, by definition, found a path whose actual cost is lower than the estimated cost of any path through any node on the frontier. But since those estimates are optimistic, A* can safely ignore those nodes.
- In general, most natural admissible heuristics tend to be consistent, especially if from relaxed problems.
- Types of agents: reflex agents, planning agents (optimal vs. complete planning).

2.1 Uninformed Search

- **State Space:** A state space is also an abstraction of the world. A successor function **models** how your world works (namely, evolves and response to your actions).
- **Search Problems:** They are just models, aka, no more than models in the mathematical sense.
- **World State:** Includes every last detail of the environment.

- **Search State:** Keeps only the details needed for planning (namely, abstraction). Because only with abstraction can we solve problems smoothly.
- **Search Trees:** For most problems, we can never actually build the whole tree. (So we **ignore** most of the tree.)
- **Complete:** Guaranteed to find a solution if one exists? **Optimal:** Guaranteed to find the least cost path?
- **DFS vs BFS:** When will one outperform the other?
- **Uniform Cost Search:** Expand a cheapest node first. Thus fringe is a **priority queue**. (priority, cumulative cost, namely, add them up!) Therefore it's complete and optimal! But it explores options in every "direction". And this algorithm shows **no information** about goal location.
- Search operates over **models** (namely, abstractions) of the world. Planning is all "in simulation", therefore your search is only as good as your model is.

2.2 Informed Search

- **Informed Search:** Inject information of where the goal might be. Key idea: Heuristics.
- **Successor Function:** If I do this, what will happen, in my model.
- **Search Heuristics:** Something tells you that whether you are getting close to the goal, or not. It's a function that *estimates* how close a state is to a goal. It's designed for a particular search. Examples: Manhattan distance, euclidean distance. (They are not perfect, but they are *at least* something.)
- **Greedy Search:** A common case: best-first takes you straight to the (wrong) goal.
- **A* Search:** Revised. Combine both UCS and Greedy, namely, tortoise and rabbit. Uniform-cost orders by path cost, or backward cost. Greedy orders by goal proximity, or forward cost.
- **A* Search:** Stop when you **dequeue** a goal from the fringe. Lesson: We need estimates to be less than actual costs.
- **Admissibility:** Admissible (optimistic) heuristics slow down bad plans but never out-weight true costs. Inadmissible is just a fancy name for, **pessimistic**, it traps good plans on the fringe.
- A heuristic **h** is **admissible** (optimistic) if:

$$0 \leq h(n) \leq h^*(n) \tag{1}$$

  where $h^*(n)$ is a true cost to a nearest goal. Thus coming up with admissible heuristics is most of what's involved in using A^* in practice. A^* is not problem specific, but your heuristic is.
- **Crating Admissible Heuristics:** Most of the work in solving hard search problems optimally is in coming up with admissible heuristics. Often, admissible heuristics are solutions to **relaxed problems**, where new actions are available.
- **Graph Search:** For tree search, if it fails to detect repeated states can cause exponentially more work. Idea: **never expand** a state twice.
- Important: (in python's idea) store the closed set as a set, not a list. In Lisp's concept, make it a hash table (it is verified, just use hash table in Lisp).
- **Consistency of Heuristics:** real cost should be larger or equal than cost implied by heuristic. (Namely, please be **Conservative**, aka guess "smally" rather than "biggerly".) Implication: f value along a path never decreases.
- **Optimality:** For tree search, requires heuristic admissible; for graph search, requires consistent. And consistency implies admissibility.
- **Heuristics:** The design of this number (function) is key, often use **relaxed problems**.

2.3 Constraint Satisfaction Problems

- **Search:** a single agent, deterministic actions, fully observed state, discrete state space.
- **Planning:** a sequence of actions. The **path** to the goal is the important thing.
- **Identification:** assignments to variables. The goal itself is important, not the path.
- **CSP:** a special subset of search problems. State is defined by **variable** X_i with values from a domain D (sometimes D depends on i). Goal test is a set of constraints specifying allowable combinations of values for subsets of variables.
- CSP allows useful general-purpose algorithms with more power than standard search algorithms. (Namely, add more "rules", walk through (traverse) less paths.)
- **CSP Varieties:** Discrete variables; continuous variables.
- **Varieties of Constraints:** Unary; Binary; Higher-order constraints. Or Preferences (soft constraints).
- **Backtrack Search:** The basic uninformed algorithm for solving CSPs. Namely, recursion. One variable at a time; check constraints as you go (Online shit? Incremental goal test). So backtracking is equal to DFS add variable ordering and add fail-on-violation.
- **Improve Backtracking:** Ordering; Filtering; Structure.
- **Filtering:** Keep track of domains for unassigned variables and cross off bad options. Namely, build a mathematical filter. Namely, ask (cond, else) when doing forward checking.
- **Forward Checking:** Enforcing consistency of arcs pointing to each new assignment.
- **Arc Consistency:** It still runs inside a backtrack search.
- **Ordering:** Minimum Remaining Values. Variable ordering, **always** choose the variable with the **fewest** legal left values in its domain, given a choice of variables.

- What the hell is CSP? Variables; Domains; Constraints—Implicit, Explicit, Unary/Binary/N-ary. Goals: find any solution; find all; find best, etc.
- **K-Consistency:** For each k nodes, any consistent assignment to $k-1$ can be extended to the k^{th} node.
- Suppose a graph of n variables can be broken into subproblems of only c variables. Example: $n = 80, d = 2, c = 20$. But this "crap" is somehow impractical.
- **Tree-Structured CSPs:** Theorem, if the constraint graph has no loops, the CSP can be solved in $O(nd^2)$ time. For general CSPs, worst case is $O(d^n)$. This also applies to probabilistic reasoning: an example of the relation between syntactic restrictions and the complexity of reasoning.
- **Nearly Tree-Structured CSPs:** Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a **tree**.
- *Sorry this is in ai class, everything is hard.—CS188*
- **Tree Decomposition:** Create a tree-structured graph of mega-variables. Each mega-variables encodes part of the original CSP.
- CSPs are a special kind of search problems where states are partial assignments and goal test is defined by constraints. The basic solution is backtrack search.
- **Local Search:** (yet another fancy name of EM algorithm.) It improves a single option until you can't make it better. (**No fringe!**)
- Generally local search is much faster and more memory efficient. But it is also **incomplete and suboptimal**.
- **Hill Climbing:** Simple general idea—Start wherever, repeat: move to the best neighboring state; if no neighbors better than current, quit.
- **Simulated Annealing:** Idea, escape local maxima by allowing downhill moves.
- The more downhill steps you need to escape a local optimum, the less likely you are to ever make them all in a row. Therefore people think hard about ridge operators which let you jump around the space in better ways.
- **Genetic Algorithms:** It uses a natural selection metaphor—keep best N hypotheses at each step based on a fitness function; Also have pairwise crossover operators, with optional mutation to give variety.

2.4 Adversarial Search

- **Meaning:** How to decide how to act, when there is an adversary in "your world (model, abstraction, etc.)".
- Monte Carlo methods are just a fancy name for **randomized** methods.
- **Pacman:** Behavior from **Computation**.
- **Axes:** Deterministic or stochastic? One, two or more players? Zero sum? Perfect information (can you see the state)?
- For this course, we want algorithms for calculating a **strategy (policy)** which recommends a **move** from each state.
- Different from search: we do not **control** our opponent. We need to give out **policies**.
- One possible formalization is: States, Players, Actions, Transition Function $S \times A \to S$, Terminal Test $S \to \{t, f\}$, Terminal Utilities $S \times P \to R$.
- Players usually take turns; Actions may depend on player/state; Terminal utilities tells us how much it's worth to **each of the players**.
- **Zero-Sum Games:** Let us think of a single value that one maximizes and the other minimizes.
- **General Games:** Agents have independent utilities. Cooperation, indifference, competition, and more are all possible.
- **Value** of a state: The **best** achievable outcome (utility) from that state.
- **Minimax Values:** States Under Opponent's Control: $V(s') = min\ V(s)$ States Under Agent's Control: **Maximize** out of all possible "worst" results your **opponent** offers. In choosing universities and advisors, pick out the "tallest" guy from the "small man".
- In other words, life is much much worse when there is an (or more than one) adversary. I want the "global maximum", but the adversary just **won't let it happen**.
- **Minimax Search:** A state-space search tree; players alternate turns; compute each node's **minimax value**, namely the best achievable utility against a rational (optimal) adversary.
- Ask this question to yourself: do we **really** need to explore the whole tree?
- **Resource Limits:** In realistic games, cannot search to leaves. Solution: **Depth-Limited Search**. Search only to a limited depth in the tree, and replace terminal utilities with an evaluation function for non-terminal positions.
- **Depth Matters:** An important example of the tradeoff between complexity of features and complexity of computation.
- **Evaluation Functions:** In practice, typically weighted linear sum of features.
- **Game Tree Pruning:** Look at the trees that do not have to be minimized.
- **Alpha-Beta Pruning:** Key idea it symmetric. To sum up, it's already **bad enough** that it won't be played. α MAX's best option on the path to root. β MIN's best option on path to root. Tip: You have to be right for the children of the route. Therefore good child ordering improves effectiveness of pruning.

3 Uncertain Knowledge and Reasoning

3.1 Expectimax and Utilities

- Uncertain outcomes controlled by **chance**, not an adversary!
- Values should now reflect average-case (expectimax) outcomes, not worst-case (minimax) outcomes. (Explicit randomness, Unpredicable opponents, Actions can fail).
- **Expectimax search:** compute the average score under optimal play. Key idea: Calculate their **expected utilities**.
- For average-case expectimax reasoning, we need **magnitudes** to be meaningful. Not only order matters, magnitudes as well.
- As we get more evidence, probabilities may change.
- **Random variable** represents an event whose outcome is unknown.
- **Probability distribution** is an assignment of **weights** to outcomes. Note, in functional programming, we can also say that it's a **mapping** of weights to outcomes.
- The **expected value** of a function of a random variable is the **average**, weighted by the probability distribution over outcomes.
- Having a probabilistic belief about another agent's action **does not** mean that the agent is flipping any coins!
- Based on **how we think** the world works, what computation we should do. Are our opponents adversarial or by chance?
- **Minimax** generalization:
  - Terminals have utility tuples (namely, lists)
  - Node values are also utility tuples
  - Each player maximizes its own component
  - Can give rise to **cooperation** and **competition** dynamically.
- **Vampire Bunnies**
- Worst case reasoning only works to the extent that your model is sufficiently simple. Namely, **minimax** is just a **binary** "crap", though the world can be "simulated" based on asking **yes or nos**, it is still too young too simple, sometimes naive, naive! "Yo you may be hit by a METEOR!!"
- Tip: A rational agent should choose the action that maximizes its expected utility, **given its knowledge**.
- **Utilities** are functions from outcomes (states of the world) to real numbers that describe an **agent's preferences**.
- We hard-wire utilities and let behaviors emerge. The search procedure should do that for us. Behavior is complicated and **context** dependent.
- Utilities can also be regarded as a reflection of **uncertain outcomes**. But win or lose, you play it.
- An agent with **intransitive preferences** can be induced to give away all of its money. (Loop forever.)
- Utility scales: normalized utilities, micromorts, quality-adjusted life years.
- People would pay to reduce their risks.

3.2 Markov Decision Processes

- **MDP** A way of formalizing the idea of non-deterministic search, which is the search when your actions' outcomes are **uncertain**.
- **Noisy Movement** actions do not always go as planned.
- An MDP is defined by: a set of states, a set of actions, a transition function, a reward function, a start state, and maybe a terminal state. Therefore, one way to solve the MDPs is with expectimax search. Namely, it's yet another **fancy search**, but our **testing goal** has changed.
- "Markov" means action outcomes depend only on the current state (namely, to simplify the calculation, we **do need to** make some assumptions.) This is just like search, where the **successor function** could only depend on the current state (not the history). Well, if you do want to depend on the history, GIFF more powerful computers and be a master of, you named it, **statistics and matrix**.
- In deterministic single-agent search problems, we wanted an optimal plan, or sequence of actions, from start to a goal. (Actually same "greedy ideas" in non-deterministic problems, but in situations like this, we are forced to enjoy the **uncertainty**, which comes from, well, you named it, mother nature or rather, quantum mechanics.)
- An optimal **policy** is one that maximizes expected utility **if followed**. When we say if, you know we are talking about the fucking **uncertainty**.
- An **explicit** policy defines a reflex agent.
- **Reflex** an action that is performed as a response to a stimulus and without conscious thought.
- Expectimax did **not** compute entire policies, solutions, ideas, paths whatsoever. It computed the action for a single state only. But **IT WORKS**.
- What **Markov** did is just to remove the redundancy so that we can use **minimal** information to **render** the things down.
- **Your models are never going to be perfect.**
- Each MDP state projects an expectimax-like search tree.
- **Utilities of Sequences** Ask questions: more or less; now or later (mind the **discounting**).
- **Discounting** values of rewards (may and usually so) decay exponentially. It helps our algorithms **converge**!!!

- **Infinite Utilities** Finite horizon, discounting or absorbing state (like "overheated" for racing). In general we will have **discounts**.
- **Policy** Mapping from actions to states. **Utility** sum of (discounted) rewards.
- **Values of States** fundamental operation: compute the (**expectimax**) value of a state.
  - Expected utility under optimal action.
  - Average sum of (discounted) rewards.
  - This is **just** what expectimax computed.
- Recursive definition of values:
  - $V^*(s) = \max_a Q^*(s, a)$
  - The value of a state is the **max** over all the **actions** available to you.
  - $Q^*(s, a) = \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^*(s')]$
  - We are going to get a reward in the next time step, and a **future** reward. They are going to be **weighted** by the **relative probabilities** that come from our **transition function**. Mind the future shit, so we need to add a **discount** on the future, because it's not what we get NOW.
- States are repeated. Tree goes on forever. Note: deep parts of the tree eventually **does not** matter if $\gamma < 1$.
- **Time-Limited Values** $V_k(s)$ is the **optimal** value of s if the game ends in k more time steps. Equivalently, it's what a **depth-k** expectimax would give from s. (Namely, life is short, PLEASE do not loop/recur forever, please...)
- Because we are just truncating, we are just ending the game—we **do not** need an evaluation function. Because **IT JUST STOPS**.
- So it is a **trade-off** of how many states you have, how connected they are and how deeply you want to go into the tree.
- Basic idea: approximations get refined towards optimal values. (When we say refine, we can also say, you guessed it, **filtering**.)
- **Convergence**
  - If the tree has maximum depth M, then V_M holds the actual untruncated values. **I have done the EM search** and any further iteration/recursion will do nothing (Namely, their **reward**, return, gains, utilities are considered, regarded, or set 0, so it's "mathematically" dead, then we do **not** need to, you name it, do calculations **any more**).
  - If the discount is **less than** 1.
  - The last layer is at best all $R_{\max}$.
  - The last layer is at worst $R_{\min}$.
  - But everything is **discounted** by γ^k that far out.
  - So V_k and V_{k+1} are **at most** $\gamma^k \max |R|$ different.
  - So as k increases, the values converge because the **differences** are **smaller than smaller**. After all these "smalling" processes, it will die, or at least looks like "dead". That word means there is **no significant** further growth.

3.3 Markov Decision Processes Continued

- Oh! That's their probability of landing a s'.
- **Policy** A map (surely it's with Python and Lisp) of states to actions.
- Your transition function tells you what the **likelihoods** are.
- **Rewards** are instantaneous and **values** are accumulative. It's an very important distinction.
- **How to be optimal:** Take correct first action; Keep being optimal. It's just some kind of recursive procedures, or some "iterative dynamic programming". Holy Jesus Lord, **they are actually fuckingly the same thing, will you be able to understand**? This of course is in terms of **implementation**. And isn't this concept called "greedy" in algorithm book?
- Bellman equations **characterize** the optimal values; the **fuck yourself** algorithm "computes" them. Because of this, there is now a **real recursion**. It bottoms out at 0. And suddenly we have a notion of times-attached.
- Expectimax trees max over all actions to compute the **optimal** values.
- Turn recursive Bellman equations into **updates**. Namely, it's just the fucking value iteration.
- Without the maxes, the Bellman equations are just a **linear system**.
- $\pi^*(s) = argmax_a \Sigma T(s, a, s')[R(s, a, s') + \gamma V^*(s')]$ This is called **policy extraction**, since it gets the policy implied by the values. This is also called **reconstruct**.
- $\pi^*(s) = argmax_a Q(s, a)$ Computing actions from **Q-Values** is completely trivial. And it is much easier to select from q-values than values.
- What value iteration does, is essentially mimics Bellman Equation. Problem: It's slow—$O(S^2A)$ per iteration. The "max" at each state rarely changes. And more importantly, the **policy** often converges **long before** the values.
- **Policy Iteration (Or Recursion)** Evaluation: for fixed current policy π, find values with policy evaluation. Improvement: for fixed values, get a better policy by **argmax**.
- Another view (which is more practical) is thinking we are doing **value iteration** but on most rounds we just go with the **last action** that optimize the state, rather than considering them all.
- They differ only in whether we plug in a fixed policy or max over actions. Namely, these all look (and essentially are) the same.

- That wasn't planning! It was learning. There was an MDP, but you couldn't solve it with just **purely precomputation**. (It's all about the computation, but orders, namely, prior and posterior do matter.)

- Important ideas in reinforcement learning (in control theory bitches, they call the **crap negative/positive feedback loop**):

  - Exploration: you **have to** (as is with my situation, I have to learn, to prove to others, and to be strong) try unknown actions to get information.

  - Exploitation: eventually, you have to **use what you know**

  - Regret: even if you learn intelligently, you **make mistakes**.

  - **Sampling** because of chance, you have to try things **repeatedly**.

  - **Difficulty** learning can be much harder than solving a known MDP. (Comment: maybe that's why the Machine Learning SIGs are recruiting more and more, say, members/noobs?)

- Your **lack of knowledge** then triggers a much more difficult reasoning problem about **HOW** you should act. (Math buddies tells you WHAT. Programmers do the HOW, and a few true geniuses learn/are forced to ask WHY.) "Why OSX is more fancy? Why Windows has such a large market share, etc."

- **How you solve an MDP when you don't know which MDP you are solving?** Eggs first? Or Chicken first?

3.4 Probability

- You deal with integrals rather than summations.

- Size of distribution if n variables with domain sizes d?

- A probabilistic model is a **joint distribution** over a set of random variables.

- Marginal distributions are sub-tables which eliminate variables.

- **Sanity Check**.

- $P(a|b) = \frac{P(a,b)}{P(b)}$ "Hey! I know b is happening. Then you can, in this way calculate the **probability** of a is ALSO happening."

- **Conditional Distribution** is a consequence of our **Joint Distribution**. (Mother and Son relationship.)

- **Probabilistic Inference** compute a desired probability from other **known** probabilities.

- Inference by enumeration: select the entries **consistent** with the evidence; sum out H to get joint of Query and evidence; normalize.

3.5 Markov Models

- $P(y)P(x|y) = P(x,y)$ derived from conditional probabilities. We specify a marginal and a conditional, then we get a **joint distribution**.

- **The Chain Rule** for joint probability distribution: $P(x_1, x_2, \ldots, x_n) = \Pi_i P(x_1|x_1 \ldots x_{i-1})$

- $P(x|y) = \dfrac{P(y|x)}{P(y)}P(x)$ Both of them equal to the **joint probability**. And this fancy refactored formula is called (you guessed it) **Bayes' Rule**.

- Let us build one conditional from its **reverse**.

- What the heck is "given z", "given x" or "given y" in the probabilistic reasoning, or statistic crap? "They will bend to my command!" Or in a more functional flavor, **Realm of Racket**. In set theory, that's called "realm of z", "realm of x", etc.

- **Conditional Independence** is our most basic and robust form of knowledge about uncertain environments. X is conditionally independent of Y given Z **if and only if**:

$$\forall x, y, z : P(x, y|z) = P(x|z)P(y|z)$$

  or equivalently, **if and only if**:

$$\forall x, y, z : P(x|z, y) = P(x|z)$$

- **If I destroy you, what business is that of yours?**

- Why I write crap sentences like the one above? Do not you see in the second form that y is "destroyed"?

- Often, we want to **reason about a sequence** of observations.

- Markov Models **implied conditional independencies** Past variables independent of future variables **given the present**.

- **Stationary Distribution** The distribution we end up with is called stationary distribution P_∞ of the chain. It satisfies: $P_\infty(X) = P_{\infty+1}(X) = \sum_x P(X|x)P_\infty(x)$. It just says that this **function converges**. It converges! There is an END!

3.6 HMMs and Particle Filtering

- Often, we want to **reason about a sequence** of observations. Then we need to introduce **time** (or space) into our models.

- A **Markov Model** is a chain-structured BN.

- How we could mathematically show what is **intuitively** true here, in Markov models.

- "The distribution on the year 3000 turns out not to depend on the possibility of today's at all." It's just a **property** of transition function/matrix.

- When running Gibbs sampling long enough we get a sample from the desired distribution. The **long enough** does indeed implies **infinity**.

- I know two things, how the world changes in every time-step. And some kind of reading every step to help me. Namely, you observe outputs (effects) at each time step.
- "This is your assumptions about the world."
- HMMs have two important independence properties: Markov hidden process, future **depends** on past **via the present**. More importantly, current observation independent of **all else** given current state.
- The evidence variables are **conditionally independent** given the HIDDEN state.
- **Hidden State** Usually the thing you want to figure out. **Evidence Variable** The thing you observe.
- We all know about Bayes Nets. You write down the things you **know**, and **figure out** the things you don't know.
- Passage of Time, basic idea, beliefs get "pushed" through the **transitions**. IMHO, just two-step Bayes calculation, you have Evidence, then you reason about X_1. Since E_1 has no direct relation with X_2, the whole remaining processes can **only** done via $X_1 \to X_2$.
- Beliefs can be "reweighted" by likelihood of evidence.
- More particles (THEY ARE JUST samples), more accuracy.
- This is like prior sampling—samples' frequencies reflect the transition probabilities. **If enough** samples, **close** to exact values before and after.

3.7 Application of HMMs

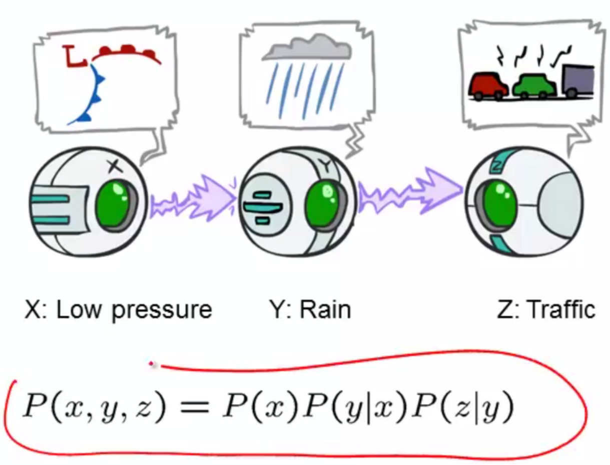
- As time passes, the distribution can **change**. It depends on the **model** we have for **how** time passes.
- **Dynamic Bayes Nets** We want to track multiple variables over time, using **multiple sources of evidence**.
- Repeat a fixed Bayes net structure at each time.
- Forward Algorithm (Sum) $f_t[x_t] = P(x_t, e_{1:t}) = P(e_t|x_t)\Sigma_{x_{t-1}} P(t_t|x_{t-1}) f_{t-1}[x_{t-1}]$
- Viterbi Algorithm (Max) $m_t[x_t] = \max_{x_1:t-1} P(x_{1:t-1}, x_t, e_{1:t})$
- It costs money to run, well, **computation**.
-

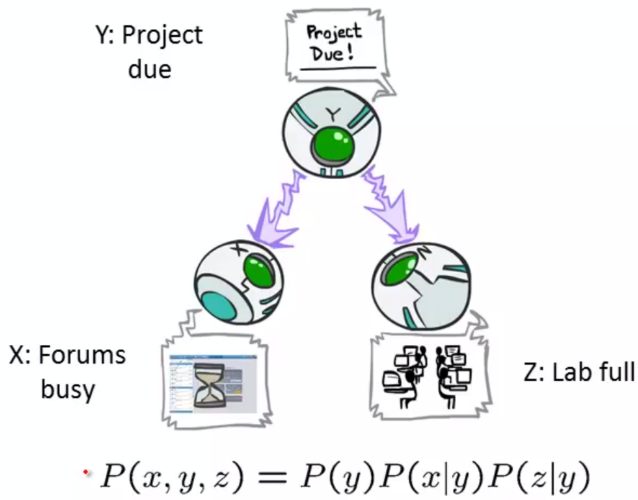
3.8 Bayes Nets: Representation

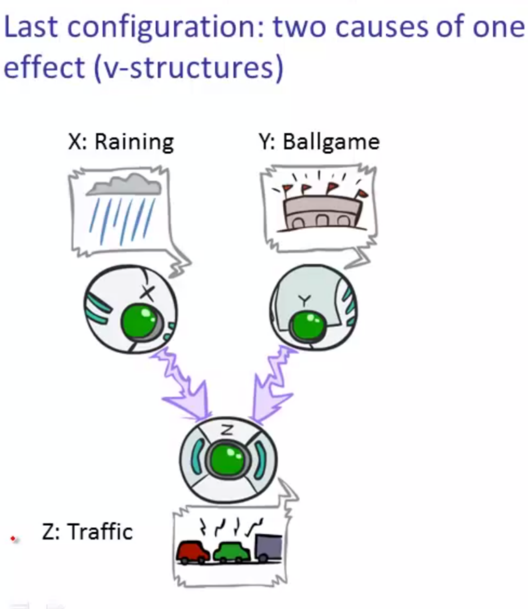
- HOW TO DEAL WITH UNCERTAINTY?
- **Models** describe how **a portion of** the world works.
- Models are always **abstractions and simplifications**.
- **Conditional Independence** is our most basic and robust form of knowledge about uncertain environments.
- Things appear in the exponent tend to **blow things up**. Like a "blow job"?
- Bayes' nets or graphical models help us express conditional independence assumptions.
- You apply the chain rule. You apply the conditional independence assumption, and it simplifies these conditional distributions, making them smaller.
- **Bayes' nets** is a technique for describing complex joint distributions (models) using simple, local distributions (conditional probabilities).
- Indeed, Bayes Networks is one kind of Graphical Models.
- **Arcs** indicates interactions, the "direct influence" between variables. Formally speaking, the arcs encode **conditional independence**.
- Bayes Net Semantics
  - A set of nodes, one per variable X
  - A directed, acyclic graph
  - A conditional distributions for **each node**
  - To sum up, it's Topology plus Local Conditional Probabilities
- $P(x_1, x_2, \ldots, x_n) = \Pi_i^n P(x_i|parents(X_i))$ All we need is this assumption.
- What do the arrows really mean? Topology may happen to encode causal structure; it really encodes, yet again, you named it, **conditional independence**.

3.9 Bayes Nets: Independence

- Why KSA is the best country in the world?
- Important for modeling: understand assumptions made when choosing a Bayes net graph
- $P(x, y, z) = P(x)P(y|x)P(z|y)$ Evidence along the chain "blocks" the influence.
- Observing the cause blocks influence between effects.
- The third case: it's backwards from the other cases. Observing an **effect** influence between possible causes.
- Any complex example can be broken into repetitions of three canonical cases.







- **Reachability** Shade evidence nodes, look for paths in the resulting graph
- Question: Are X and Y conditionally independent given evidence variable Z? **Yes**, if X and Y are "d-separated" by Z. How-to? Consider all undirected paths from X to Y, if there is **no active paths**, we get (conditional) independence. Namely, if ALL paths are inactive, then the (conditional) independence is guaranteed.
- All it takes to block a path is **a single INACTIVE** segment.
- A Bayes Net's joint distribution may have further (conditional) independence that is not detectable until you inspect its specific distribution.

3.10 Bayes Nets: Inference

- **Inference** Calculating some useful quantity from a joint probability distribution.
- Core of this part is to find **more effective ways** to do "inference by enumeration".
- **Variable Elimination** Interleave joining and marginalizing.
- Marginalization: Take a factor and sum out a variable. Aka it shrinks a factor to a smaller one; in Communication, they say it's a **projection** operation. (In math, they say let's do **differentiation**.)
- $A + \bar{A} = 1$ and $A \cdot \bar{A} = 0$
- All we are doing is exploiting **math equations/tricks** to? Improve **computational efficiency**. So that rolls back to my good old question—What if we have a computer with infinite **computation power**? What would we do with that?
- The computational and space complexity of variable elimination is determined by the largest factor.
- If we can answer $P(z)$ equals to 0 or not, we answered the 3-SAT problem has a solution. Hence **inference** in Bayes Net is **NP-hard**. **No known efficient** probabilistic inference in general.
- A polytree is a directed graph with **no undirected** cycles.
- Remember to pick the ordering of elimination that is EFFICIENT.

3.11 Bayes Nets: Sampling

- **Sampling** is just a way to do approximate inference. But it can also help you "know" an unknown distribution.
- How do you sample in an uniform way over $[0, 1)$.
- Sampling in Bayes Nets: prior, rejection, likelihood weighting, Gibbs.
- If you know the query you are going to answer, you can upgrade to **rejection sampling**.
- Problems with rejection sampling: if evidence is **unlikely**, rejects lots of samples; evidence not exploited as you sample.
- Key idea in likelihood weighting: if it is an evidence (evidence means true, in this method we **force it to be true** with a lower "class") you just fix it (FORCE TRUE), no sampling involved. But you correct ("compensate") for the fact that you didn't sample by "re-weighting" (namely, revolution. You are NOT happy, so you want to re-weight the society.) the sample.
- Problems with likelihood weighting. Evidence influences the choice of downstream variables, but not **upstream ones**.
- We would like to consider evidence when we sample every variable. "Why not use Gibbs sampling?"
- Gibbs sampling is a special case of more general methods called **Markov Chain Monte Carlo** methods. Namely, randomized methods.
- You may read about Monte Carlo methods! Well, they are just **sampling**.

- **Gibbs Sampling** procedure: keep track of a full instantiation $(x_1, x_2, x_3, \ldots, x_n)$. Start with an arbitrary instantiation with the evidence. Sample one variable at a time, conditioned on **all the rest**, but **keep evidence fixed**. Most important of all, keep repeating this for a long time.
- **Correct Distribution** $P(unknown|known)$.

3.12 Decision Diagrams and Value of Information

- OUR GOAL: Choose the action which maximizes the expected utility given the evidence.
- Σ Screw those fucking Σs you see ALMOST everywhere in books, slides, articles, journals. It's just a fancy name/notation for **Expectation** with some "domain-specific" distribution of probabilities. For **pdf**s, you can almost (not in everycase, but most case you can) **visualize** every one of them. (For high dimensional buddies, we do the selection/extraction/filtering.)
- Why do we need MEU? Because this course is named **Computational Rationality**.
- **Value of Information** Compute value of acquiring evidence. "But you are right that getting information equals to (they timidly saying corresponds to) observe the evidence in the Bayes Nets." See! That's how the academia operates, well, in general.
- $VPI(E'|e) = (\Sigma_{e'} MEU(e, e')P(e'|e)) - MEU(e)$ Just remember or visualize the forecast/umbrella case in your mind, then forget this mind-sucking equation.
- **In AI course, reasoning part** Information corresponds to the **observation (fix, given or whatsoever)** of a node in the decision network. If data is "noisy" that just means we **don't** observe the original variable (bad shooting of your, you named it, sperms), but **another variable** which is a noisy version of the original one. (More intuitively, you're WRONG.)
- **POMDPs** are MDPs over **belief states** b (the distribution over S).
- General solutions map belief functions to actions, JUST GO DYNAMIC PROGRAMMING/FUCKING RECURSION/MAP REDUCE:
  - Can divide regions of belief space (set of belief functions) into policy regions (gets complex quickly).
  - Can build approximate policies using discretization methods.
  - Can factor belief functions in various ways.

4 Learning

4.1 Reinforcement Learning 1

- Basic idea:
  - Receive feedback in the form of **rewards**
  - Agent's utility is defined by the **reward function**
  - Must (learn to) act so as to **maximize expected rewards**
  - Rule of thumb: All learning is based on what? **Observed samples of outcomes**. Namely, sensing the elephant. When you take an action, you see what happens. But you **DO NOT** see everything that might happen.
- New twist: don't know Transition function or reward function. Must actually **try actions and states** out to learn.
- Offline (MDPs). Online (Reinforcement Learning). (And then pretend that it is true.)
- **Model-Based Idea** Learn an approximate model based on experiences (namely, empirical crap). Solve for values **as if** the learned model **were** correct. (WHAT IF it is not correct? Then go fuck yourself.) This is, indeed how the state-of-the-art data mining, machine learning, and maybe cock sucking works.
- Model-Based Learning: Learn the empirical MDP model, then solve the **learned** MDP.
- $E[A] = \sum_a P(a) \cdot a$ This is 100% valid when we **know** $P(A)$. Without $P(A)$, we can **try to** collect samples $[a_1, a_2, \ldots, a_N]$.
  - For Model Based, we have $\hat{P}(a) = \dfrac{num(a)}{N}$ Eventually you learn the right model.
  - For Model Free, we just do $E[A] \simeq \dfrac{1}{N} \sum_i a_i$ Samples appear with the **right frequencies**.
- Passive Reinforcement Learning: Goal is to learn the **state values** by inputting/carrying out a fixed policy (namely, action/signal).
- We are missing something! But in the end, it's an average. It's an **average** of a bunch of things, each thing is a **one-step reward** plus a **discounted future** from **previous computation**. (Aren't math and natural language description sucking? Indeed they are.)
- **Sample-Based Policy Evaluation** Take samples of outcomes s' (by doing the ACTION!) and **average**.
- Big Idea: Learn from every experience. Namely: Update $V(s)$ each time we experience a transition (s, a, s', r). How? $V^\pi \leftarrow (1-\alpha)V^\pi(s) + (\alpha) * sample$ The fucking α is called learning rate.
- You can think of that as an **error**. $V^\pi(s) \leftarrow V^\pi(s) + \alpha(sample - V^\pi(s))$ Adjust your estimate in the direction of error by some small step size of α. In communication, they call the crap **Affine Projection** algorithm.
- Running interpolation update: $\bar{x}_n = (1-\alpha) \cdot \bar{x}_{n-1} + \alpha \cdot x_n$
- **Active Reinforcement Learning** learn the optimal policy/values
- But Q-Values are **more useful** (how the heck do you know that it is more useful? By trials and errors indeed.)

- $Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \max\limits_{a'} Q_k(s',a')]$ This crap is called sample-based Q-value recursion/iteration. Key idea: instead of looking at **two consecutive** values, let us look at **two consecutive** Q-values. (Namely, because they are "more useful". Practically, they are relatively easy to, you guessed it, COMPUTE.)

- Q-Learning converges to **optimal** policy, even if you are acting **suboptimally**! Caveats:

  - You have to explore enough.
  - You have to eventually make the learning rate small enough.
  - Basically, in the limit, it doesn't matter how you select actions. (Note: In the end, you will learn that cliff jumping is bad.)

4.2 Reinforcement Learning 2

- **Big idea** Compute all averages over T using sample outcomes, namely by trials and errors.

- The problem is, that I **do not** know the transition function.

- **Exploration** You try things, which may be disasters. Namely, you just do not know the **pits** until you try it.

- **Exploitation** You do the things which currently appear to be good.

- How to explore? Simplest, random actions (ϵ-greedy). Possible solution to problems: lower ϵ over time; use exploration functions.

- One sample exploration function is: take a value estimate u and a visit count n, returns an optimistic utility: $f(u,n) = u + k/n$. "You try things that are known to lead to things that are unknown."

- **Regret** is a measure of your total mistake cost: the difference (namely, do a comparison or subtraction or differentiation, whatsoever you like.) between your expected rewards, including youthful suboptimality, AND optimal (expected) rewards.

- Minimizing regret goes beyond learning to be optimal—it requires **optimally learning to be optimal**. (Comment: Regret, balabala, value iteration all such and such are fancy names given by **Computer Scientists** to ease the fear of dealing with TONS of mathematical equations. Namely, to concrete those fucking math abstractions—equations, inequations.) All in all, these are the ABSTRACTIONS (simulation, modeling, whatsoever) of our WORLD.

- Anyway, those models do work, under certain circumstances.

- Instead and In reality we want to **generalize**:

  - Learn about some small number of training states from experience (or part of the real states, namely, test states)
  - Generalize that experience to new, similar situations
  - This is a fundamental idea in **machine learning**, namely, build a MODEL or heuristic, and see if it works.

- You are going to learn faster if you do not have to **repeat** lessons in every **similar state**.

- **Features** are functions from states to real numbers (often 0/1) that capture **important** properties (namely, CORRECTLY describes) of the state.

- "Give me a transition so that I can learn."

- "You compute how wrong you were, and you try to make that number less."

- Adjust **weights** of active features.

- Q-learning priority: Get Q-values close (modeling). Action selection priority: get ordering of Q-values right (prediction). Solution: learn policies that maximize **rewards**, not the value that predict them. Namely, care more about PREDICTION. More namely, care more about, well, you named it, "FUTURE".

- Better methods exploit lookahead structure, sample wisely, change parameters...

4.3 Machine Learning: Naive Bayes

- **Up until now** how to use a model to make optimal decisions.

- **Machine Learning** how to acquire a model from data/experience. Learning parameters/structure/hidden concepts.

- Requirements: probabilities, BN graphs, clustering, etc.

- **Classification** given inputs x, predict labels (THAT'S JUST A FUCKING FANCY NAME for classes) y.

- **Model-Based Classification** Build a model where both labels and features are random variables. Instantiate any observed features. Then query for the distribution of the label conditioned on the features.

- **Inference for Naive Bayes** Compute posterior distribution over label variable Y (big Y is just a vector/set/domain/pussy of all possible elements it **the labels** could be, More mathematically $Y = (y_1, y_2, y_3, \ldots, y_n)$). Sum to get probability of evidence. And, according to the "formalization" of Bayes Rule, normalize your results.

- These probabilities are collectively called the **parameters** of the model (did any one of the words remind you of the good old days sucking CONTROL THEORY? If true, you HIT it.) And these **parameters** are denoted by θ (THAT'S JUST HOW ACADEMIA did it. You can name it America/China/Filippino or whatsoever.)

- Magical parameters often come from training data counts. In practice, they are the "trials and errors".

- **Important Concepts in Machine Learning** Data is just labeled instances, e.g. emails marked as spam/not spam. Features: attribute-value pairs which characterize each x (small x, the element of Set S).

- **Experimentation Cycle** Learn parameters on **training set**; Fine tune parameters on **held out** set. And last but not least, compute accuracy on **test set**. Very important: YOU ARE NEVER PERMITTED to **peek** at the test set. Or that would result in **overfitting**.

- If you are given the final exam, you could figure out what all those answers are but you haven't really learn the **general concepts**.

- Relative frequency parameters will **overfit** the training data. As an extreme case, imagine using the entire email as the only feature. So to generalize better, we need to **smooth** or **regularize** the estimates.

- Elicitation: ask a human.

- **Empirically** use the training data. This is the fucking learning process. Example: look at the empirical rate of that value. $P_{ML}(x) = \dfrac{count(x)}{total\ samples}$ And the likelihood of the data is: $L(x, \theta) = \Pi_i P_\theta(x_i)$.

- **Maximum Likelihood** Relative frequencies (as per above) are the maximum likelihood estimates.

$$\theta_{ML} = arg_\theta \max P(X|\theta)$$

  Another option is to consider the most likely parameter value given the data

$$\theta_{MAP} = arg_\theta \max P(\theta|X) = arg_\theta \max P(X|\theta)P(\theta)$$

- "Given the data, what is the most likely MODEL for our samples." Namely, we can try different **prior probabilities** and see which one works **better**.

- **Laplace's estimate** Pretend you saw every outcome **one more time** than you actually did.

- For real classification problems, **smoothing** is critical.

- We should be able to add these information sources as **new variables** in the Naive Bayes model!

4.4 Machine Learning: Perceptrons

- Perceptron has a way to kind of better tune-in of different types of features and **still get a (relatively) good result**.

- Naive Bayes models can incorporate a variety of features, but tend to do best in **homogeneous cases** (e.g. all features are word occurrences.)

- As you can see in Machine Learning, most of the time inputs are **feature values**. Each feature has a **weight**. Sum is the **activation**.

- Learning in this case: figure out the weight vector from **examples**.

- **Weights** in perceptron. For each training instance, classify with current weights; if correct, no change! if wrong, adjust the weight vector. "How to adjust the weight vector?" "STFU."

- **Multiclass** When the negative class has weight 0, we get binary class.

- For Multiclass Decision, the prediction with **highest score** wins.

- You do not stop at **one pass** until everything converges. Namely, try all possible stuff and see which one is **most likely**.

- **Mistake Bound** the maximum number of mistakes (binary case) related to the **margin** or **degree** of separability. $mistakes < \frac{k}{\delta^2}$

- **Noise** If the data isn't separable, weights might thrash.

- **Mediocre generalization** finds a "barely" separating solution.

- **Fixing the Perceptron** How to? Adjust the weight update to mitigate these effects.

- **Only support vectors** matter; other training samples are ignorable.

- Basically, SVMs are MIRA where you optimize over all samples at once. "It's all about the direction of w (namely the perceptron) here, not the magnitude."

- This algorithm takes multiple passes through data, but it is "often" (who the heck knows whether it's really often. Got the money. Got the publication. And got the girl.)

- Especially there are a wide variety of features that you have to incorporate and trade off between.

4.5 Machine Learning: Kernels and Clustering

- **Nearest-Neighbor Classification** the key issue is how to define **similarity**. Small k gives relevant neighbors, large k gives smoother functions.

- Parametric models: more data means better settings.

- Non-parametric models: Better in **the limit**, often worse in the non-limit.

- Essentially the distance and similarity are "almost equivalent".

- Many similarities based on **feature dot products**.

- A lot of work in Machine Learning, well, as have stated, is to **design the features** by/for yourself.

- As we evaluate the new training example, what we do is computing the **similarity** between $f(x_i)$ and $f(x)$ (so it should be a $N - N$ matrix) for kernel K. Once we have the kernels, we sum them up together with **appropriate weights**. That's our FUCKING FANCY NEW metric.

- **Kernelized Perceptron** If we had a black box (kernel) K that told us the dot product of two examples x and x', we will no longer need to ever take dot products. (FUCK THIS KERNEL TRICK, IT USES IF. When we use "if", we do say something. But most of the time, that's assumption.)

- The kernel is a **function** just as the very similar way that **feature is a function**.

- **Kernel Trick** With this trick, we can substitute **any** similarity function in place of the dot product. Keep in mind that **if** your kernel does not satisfy **certain technical requirements**, lots of proofs break. But in practice they "sometimes work".

- **Non-linear Separators** Data that is linearly separable works out great for linear decision rules. But what are we going **to do** if the dataset is just too hard? How about... mapping data to a higher-dimensional space?

- **Kernels** implicitly **map** original vectors to higher dimensional spaces.

- The training data is what you use to estimate the bulk of your parameters; the held-out data is to select which "hyper-parameter" to use.

- Can't you just add these features on your own? Yes, in principle, just compute them. But number of features can get large.

- Kernels let us compute with these features implicitly. That's why they exist.

- In reality there may be 100,000 features. Namely the x dimensional space.

- All those methods are very sensitive to what **similarity function** you use.

- **Agglomerative Clustering** How should we define **closest** for clusters with **multiple elements**? There are many options, like closest pair, farthest pair, average of all pairs, Ward's method, etc. Different choices create different clustering behaviors.

-

4.6 Advanced Applications: NLP, Games and Cars

- **Web Search** Not exactly classification, but rather **ranking**.

- "How do you generalize perceptron to this new scenario?"

- **Natural Language Processing** Analyze and process human languages, broadly, robustly, accurately...

- Speech recognition, machine translation, information extraction, dialog interfaces, question answering.

- **Parsing** Given a sentence, find the best tree-search.

- **PCFGs** are a formal probabilistic model of trees. Try to remember the Abstract Syntax Trees.

- "When you don't have money, you can't pay researchers."

- **Levels of Transfer** Word, Phrases, Syntax, Semantics, Interlingua.

- "It's in general not a good thing to play a lot of computer games."

- Adversarial, Long Horizon, Partially Observable, Realtime, Huge Branching factor, Concurrent, Resource-rich.

-

4.7 Advanced Applications: Computer Vision and Robotics

- **Object Detection Approach** HOG plus SVM.

- Another approach is deep learning.

- **Hill Climbing Again** Start wherever, move to the best neighboring state, if no neighbors better than current, quit. We define the neighbors to be a small perturbations of w.

- **Auto-Encoder** $min_w \Sigma_{i=1}^{m} \Sigma_{j=1}^{n} (f_j^i - h_j(w, f^i))^2$

- **Low-Level** control problem: moving a foot into a new location. This is equivalent to search with successor function, then control the moving motors.

- **High-Level** control problem: where should we place the feet. "We" build the reward function $R(x) = w.f(s)$

- Demonstrate path across the "training terrain". Run apprenticeship to learn the reward function. Receive "testing terrain", the height map. And last but not least, find the optimal policy with respect to **the learned reward function** for crossing the testing terrain.

5 Step-by-Step Tutorials

5.1 DFS and BFS

- **DFS** always looks for things deepest in the tree first.

- Priority Queue: always go with the **highest priority**. High priority is **defined** to have **lowest** cost.

-

5.2 A Star Search

- Same as intuitive tree search, the lower cost a path has, **the higher priority it deserves**.

- For applications in graph search: when you are about to expand a node, you check whether you expand the node that entered the same state—if that's the case, you don't (and of course, there is **no need**) to expand the same node, you just take it out from the **already existing** priority queue.

- So you avoid re-expanding **plans** (namely, paths) that extend to the **same state**.

- To understand trees, you have to understand **recursion**. Similarly, to understand **recursion**, you have to do a **lot** of practice with TREES.
- **Consistency** as defined in A^* search. $heur(s) - heur(s') \leq cost(s, s')$.
- WHAT THE HELL IS THE TRUE NAME FOR HEURISTIC? **Guessing**. So to achieve **optimality**, you have to do some **proper "guessing"**. It means that only if you have a consistent heuristic, will you be able to get **optimal path** when doing the A^* search.

5.3 Alpha-Beta Pruning

- **Alpha** (my best) best already explored option along path to the root for maximizer.
- **Beta** (your best) best already explored option along path to the root for the **minimizer**
- Try to understand this sentence, then you will understand the **essence** of **min-max** tree. "6 is better than $+\infty$ so the minimizer would pick 6."

5.4 D-Separation

- The good old question, "Are X and Y conditionally independent given evidence vars Z?" Yes if X and Y are **separated** by Z. Consider all **undirected** paths from X to Y, if there is no active paths, we get **conditional independence**. This is equivalent to say, **all paths from X to Y** are **inactive**.
- All it takes to block a path is **a single inactive** segment.

5.5 Elimination of One Variable

- Well, this part is somehow short. I may add something else later.

5.6 Variable Elimination

- Integral, Summation or whatsoever.
- Ordering can affect the size of factors you generate along the way.
- In all those **elimination** processes, we do the **renormalization** in the last step. We may not know the details. Even if we do, doing it at last would reduce the **computational complexity**. In short, we do so either because we lack computational power or **we are lazy enough**.
-

5.7 Sampling in Bayes Nets

- **Sampling** from a Bayes Net. We can use **forward sampling**. It's also called **prior sampling**.
- Whenever we ask the answer for conditional probability distribution, the samples we care about are the ones that have **correct evidence instantiation**.
- If we know the **query** in advance, can we **sample more efficiently**.
- To "conclude" this part, let us rephrase the good old sentence. "All probabilities are equal. But some are more equal than others."
-

5.8 Gibbs Sampling

- It's just a many-time likelihood sampling with a "randomized" variable selection.
-

5.9 Maximum Likelihood

-

5.10 Laplace Smoothing

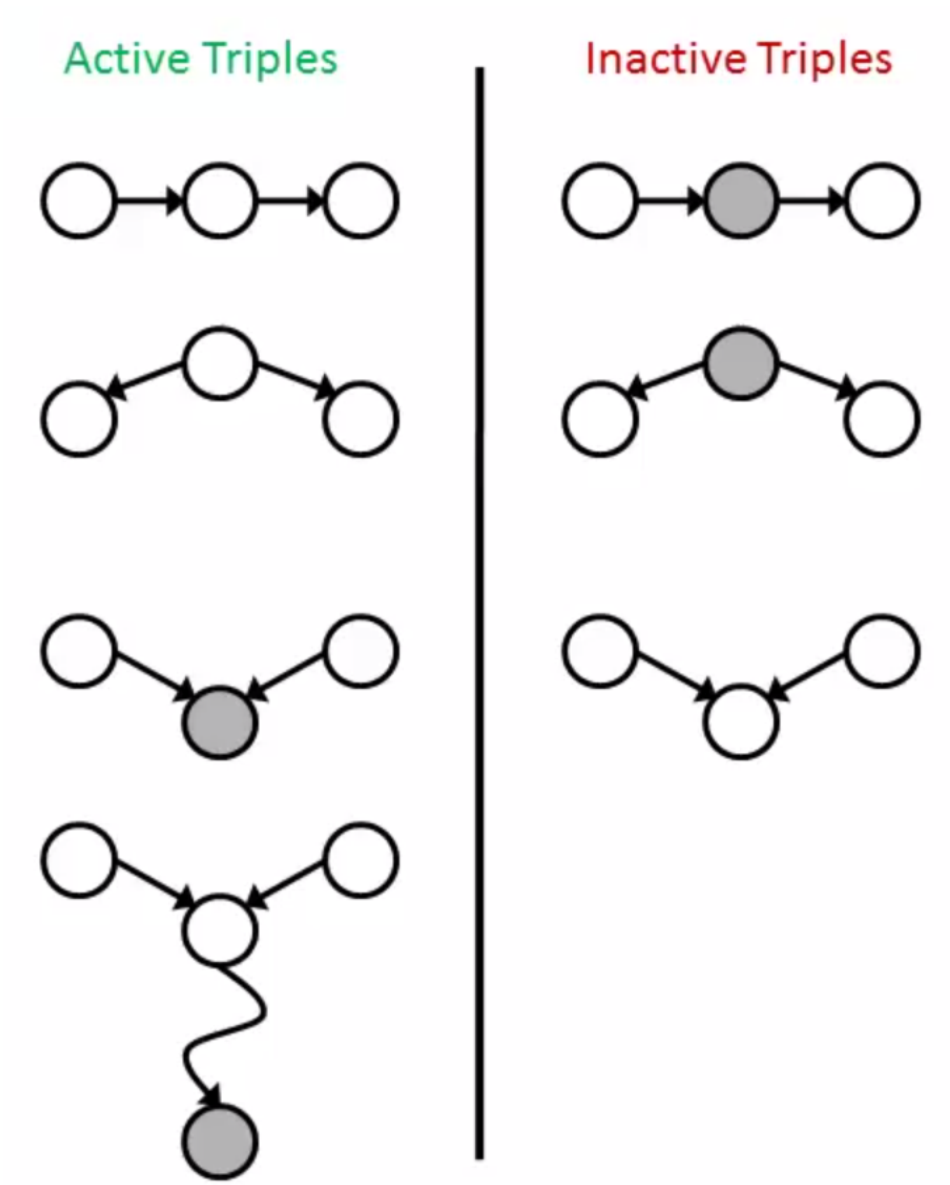
- **Laplace Smoothing** tends to draw your probability distribution closer to the uniform distribution.
- The larger you pick k, the closer you will be drawn into uniform distribution.
- Because of this kind (bunch) of smoothing, your estimated probabilities would be **non-zero**. That's very important because **Machine Learners** hate 0.
- "The core implementation of uniform distribution is, you guessed it, **communism**."
- Surely it is showing some kinds of **equality**, or rather **communism** among the data/samplings.

5.11 Perceptron Update

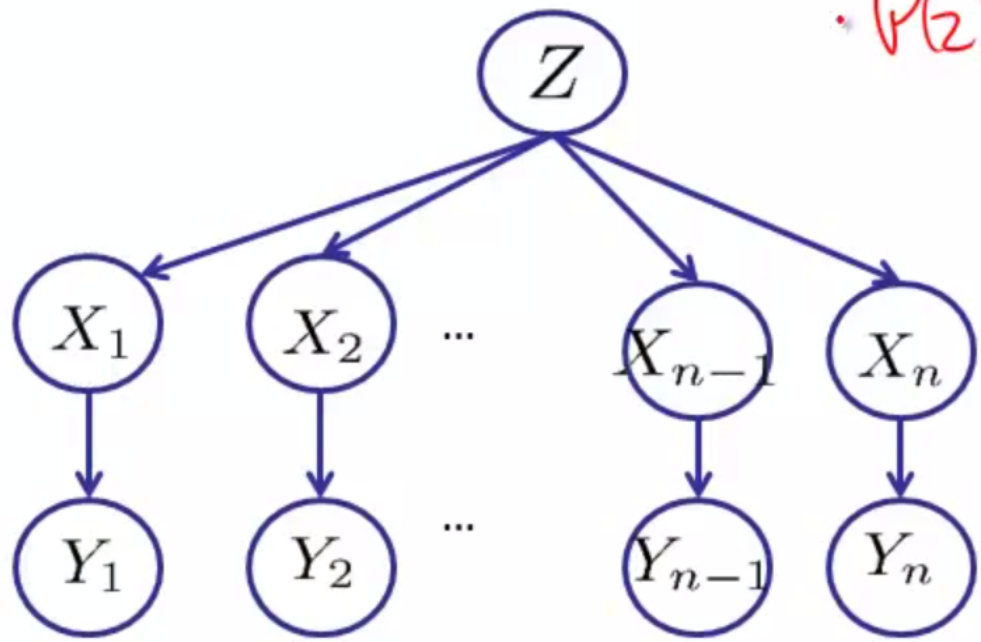
- The perceptron is a **linear** classifier and the way you train the **weights** of your perceptron is by iterating through the data.
-

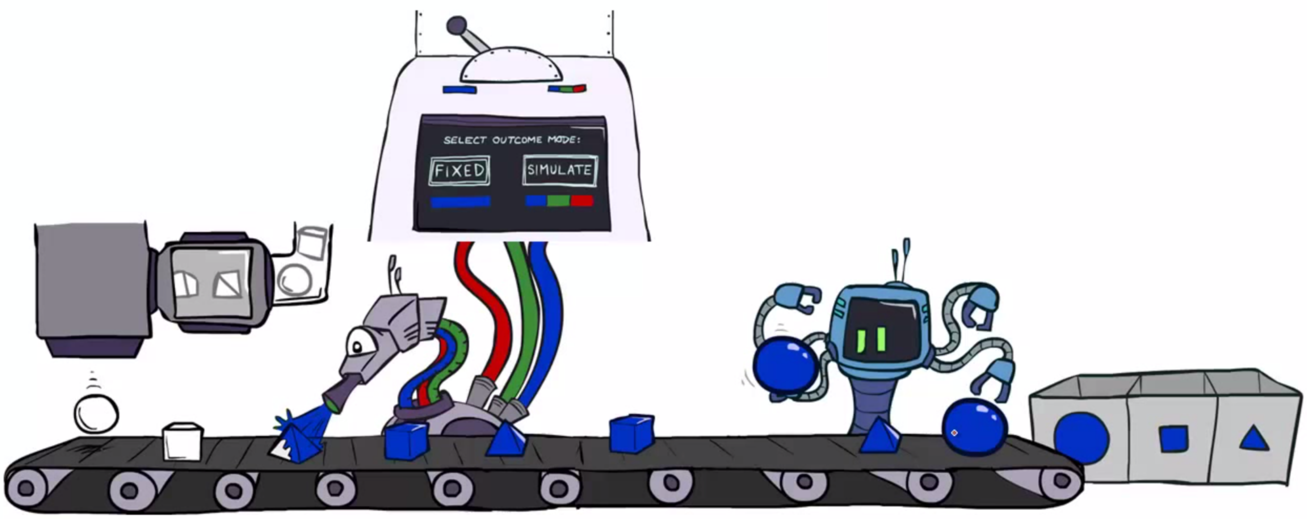
6 Appendix

6.1 Visualization

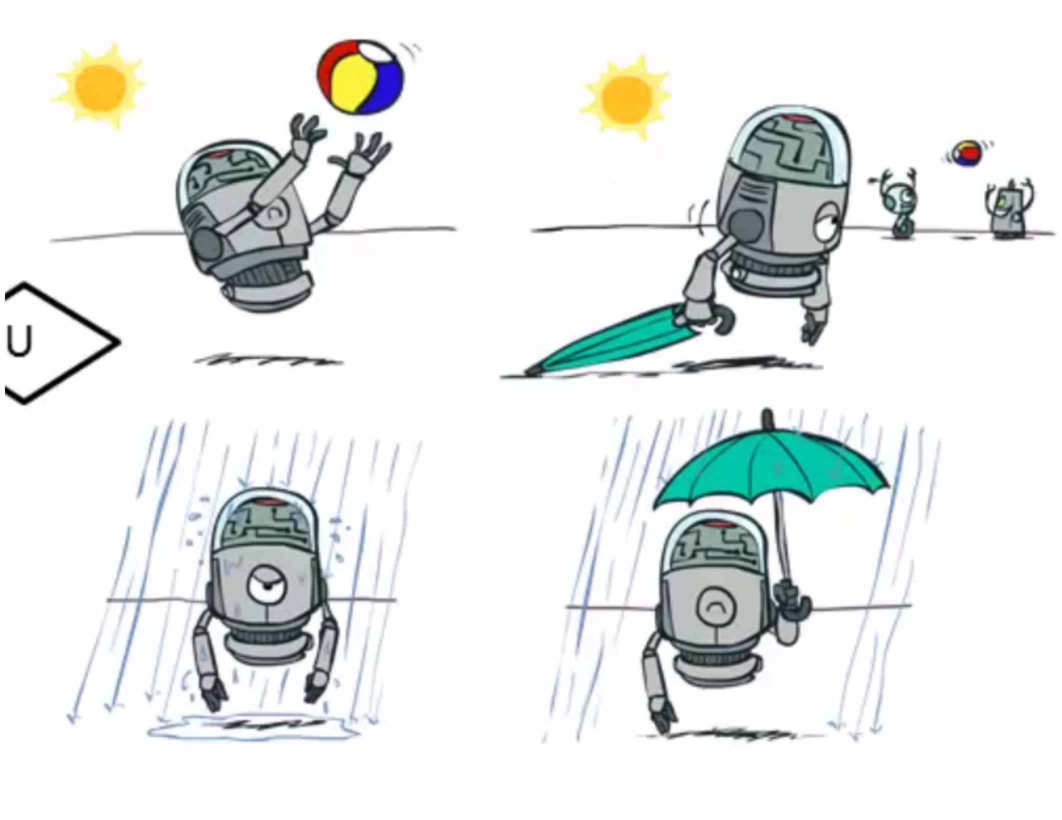


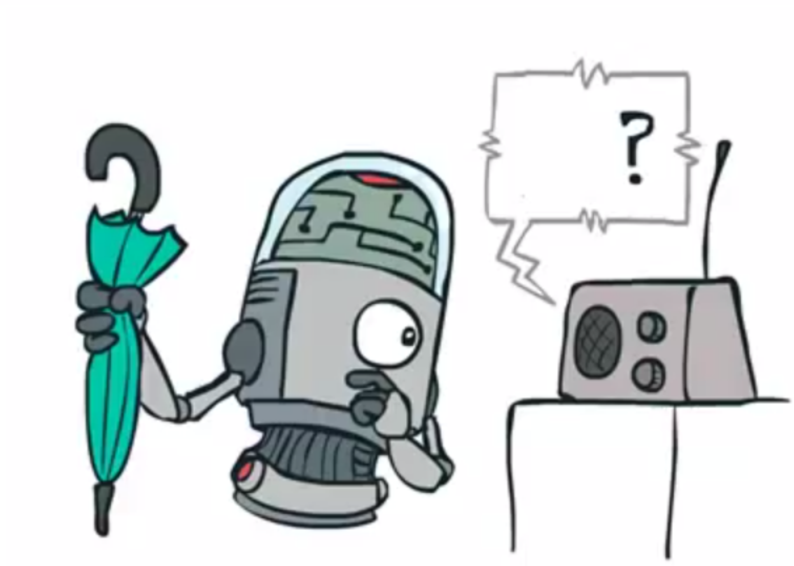
$\cdot P(z)\ P($

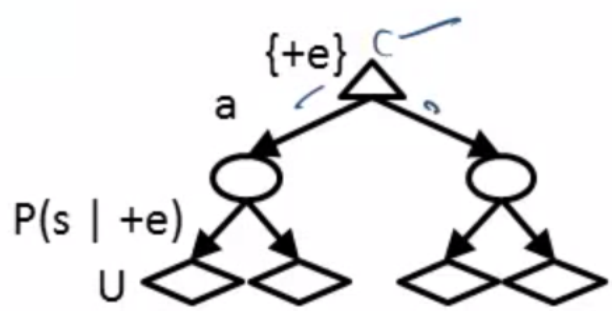


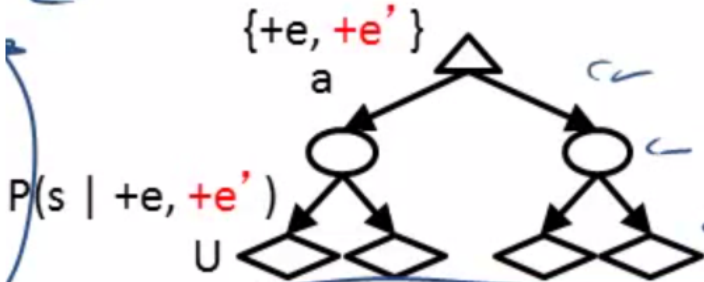


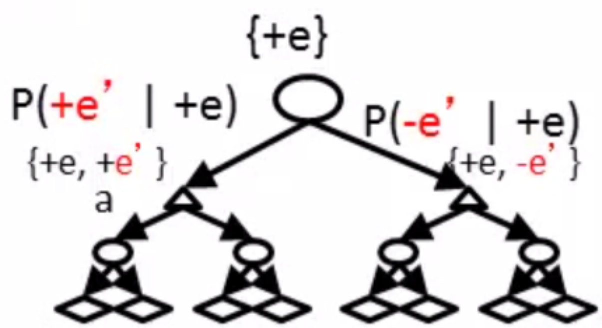
- Strategy
  - Run Dijkstra's algorithm to calculate shortest distances between all pairs of locations.
  - Features: Food pellets, heavy emphasis on power pellets, ghost distances.
  - Run depth 1 expectimax search with fine tuning of weights.

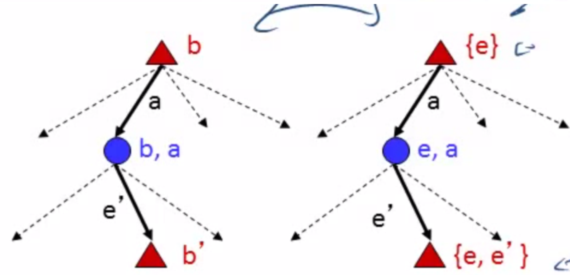

U

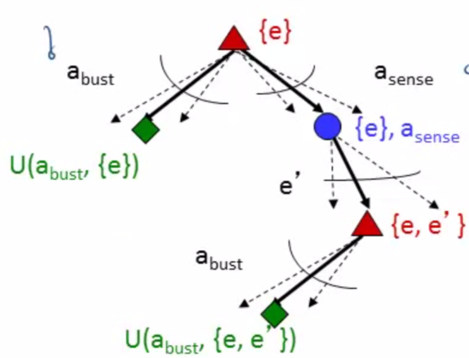

?


{+e}
a
P(s | +e)
U


{+e, +e' }
a
P(s | +e, +e')
U


{+e}
P(+e' | +e)
P(-e' | +e)
{+e, +e' }
{+e, -e' }
a

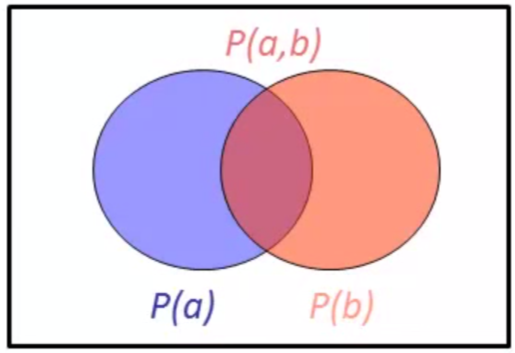

b
a
b, a
e'
b'
{e}
a
e, a
e'
{e, e' }


{e}
a_{bust}
a_{sense}
U(a_{bust}, {e})
{e}, a_{sense}
e'
{e, e' }
a_{bust}
U(a_{bust}, {e, e' })

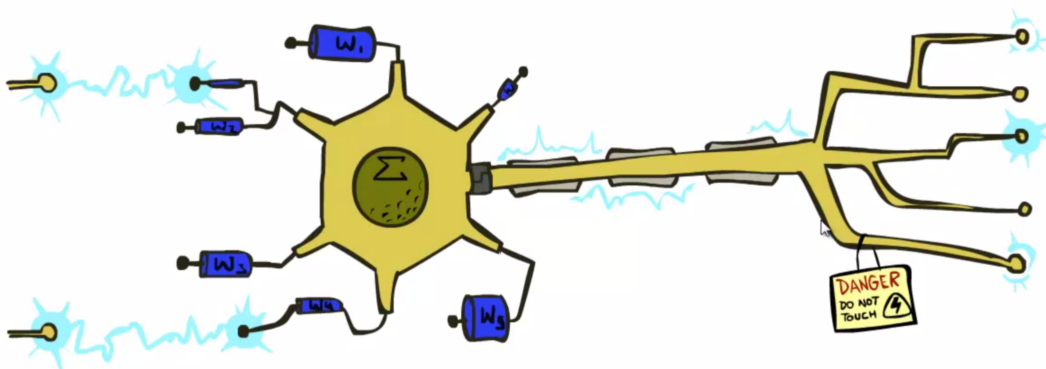
A simple relation between joint and conditional probabilities

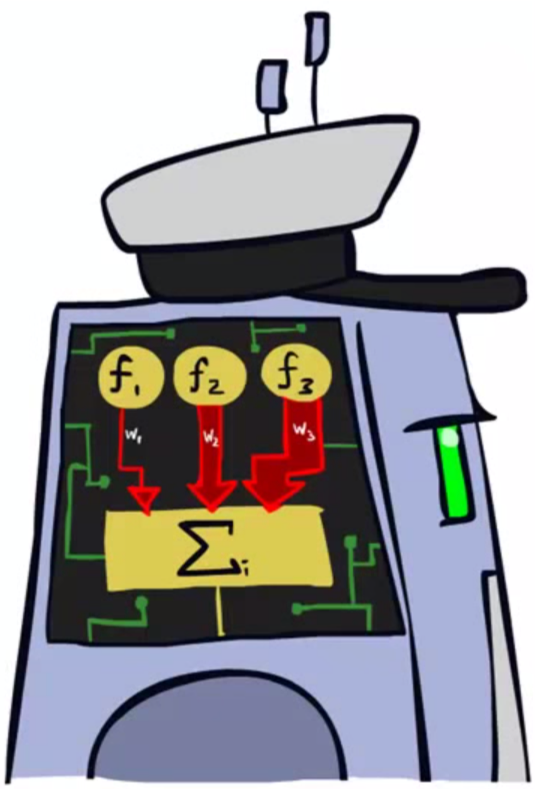
- In fact, this is taken as the *definition* of a conditional probability

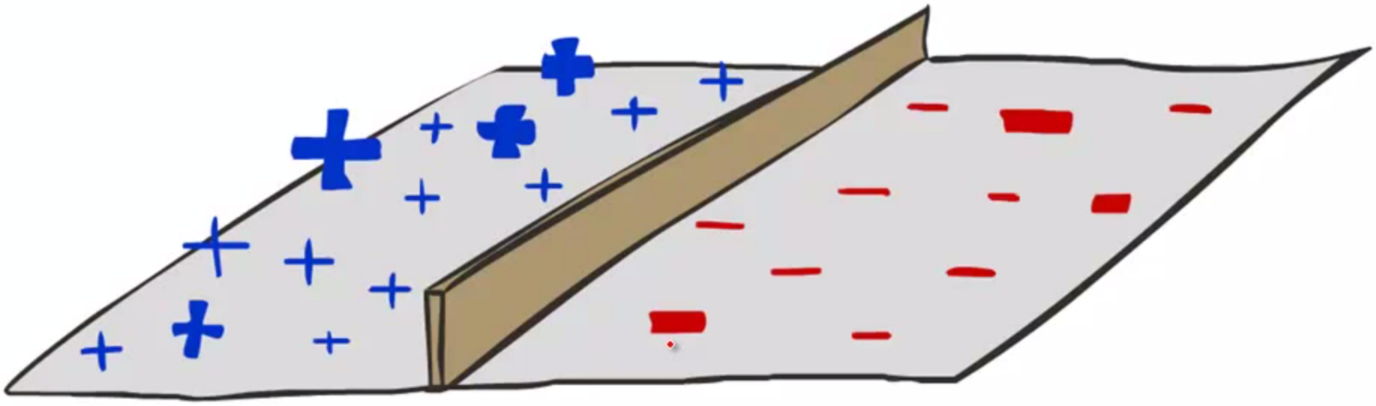
$$P(a|b) = \frac{P(a,b)}{P(b)}$$

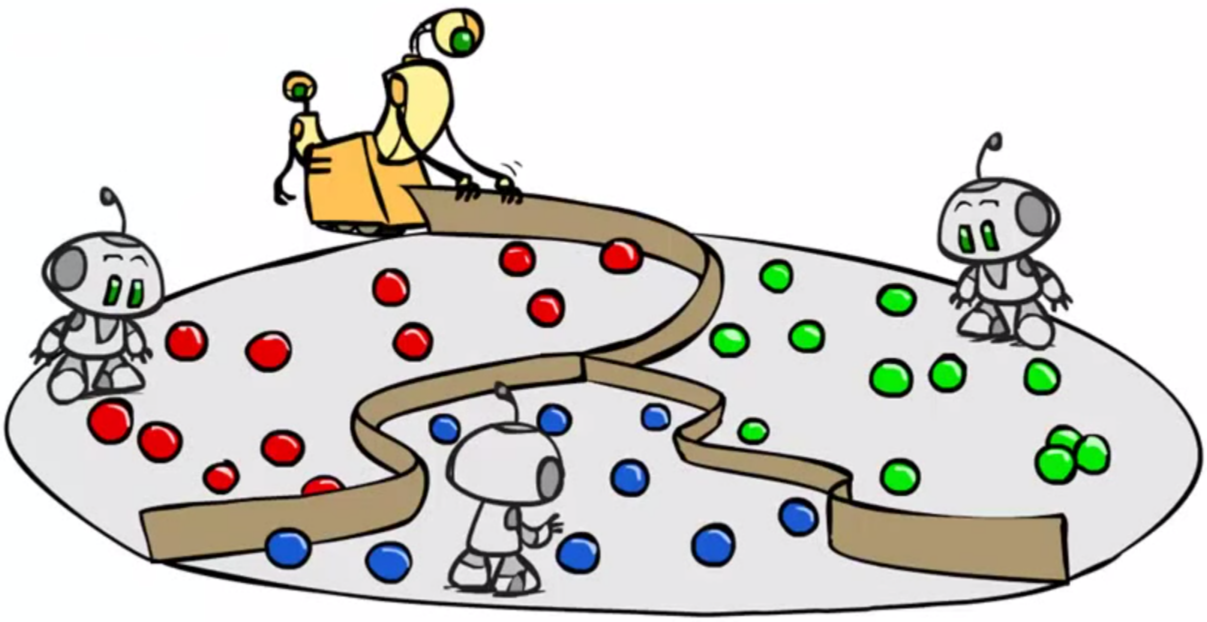


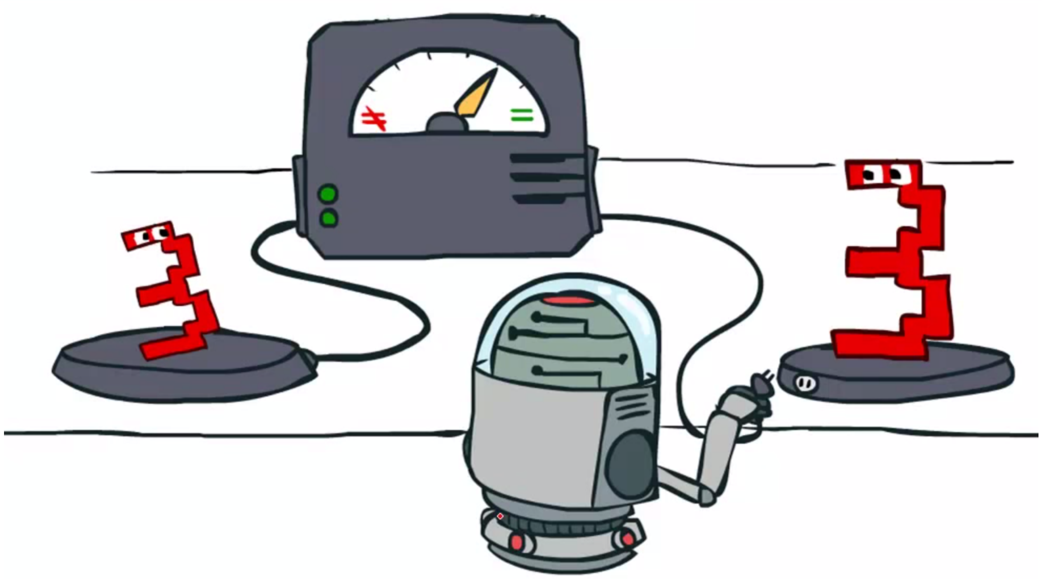
Perceptrons

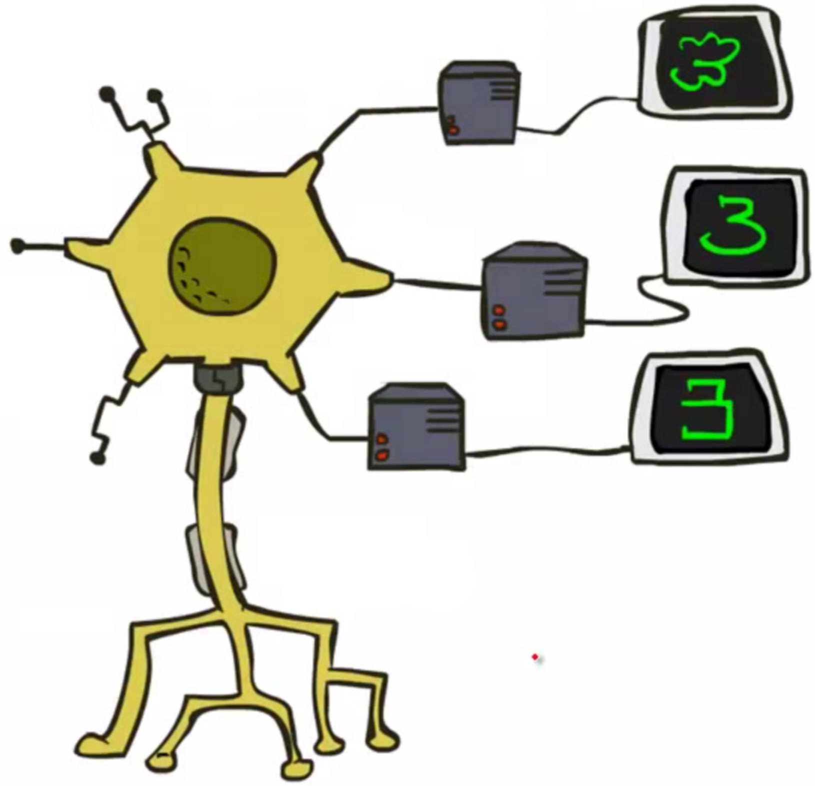


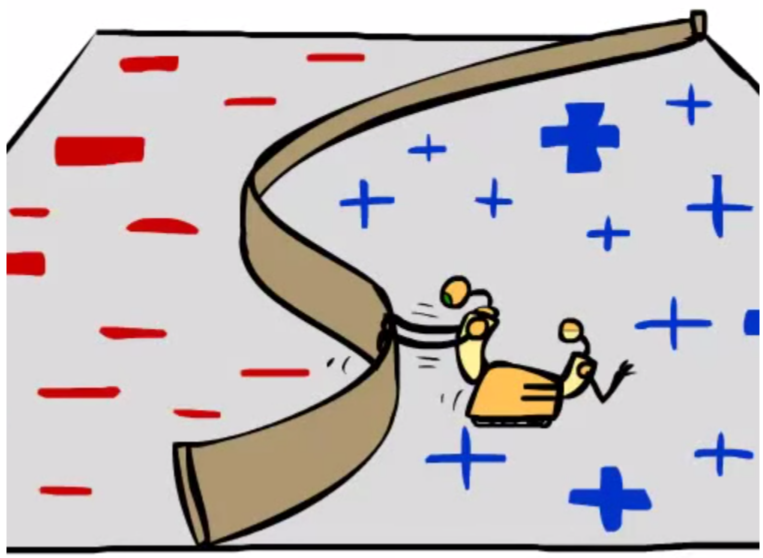


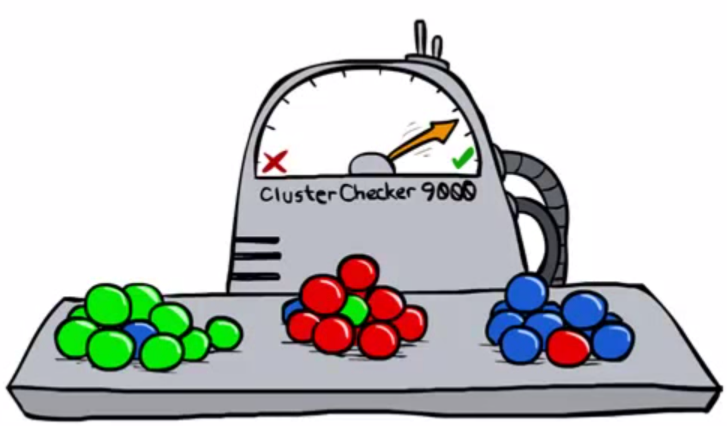


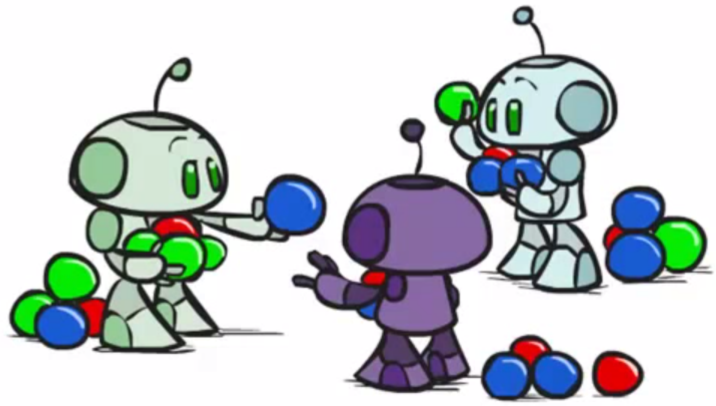


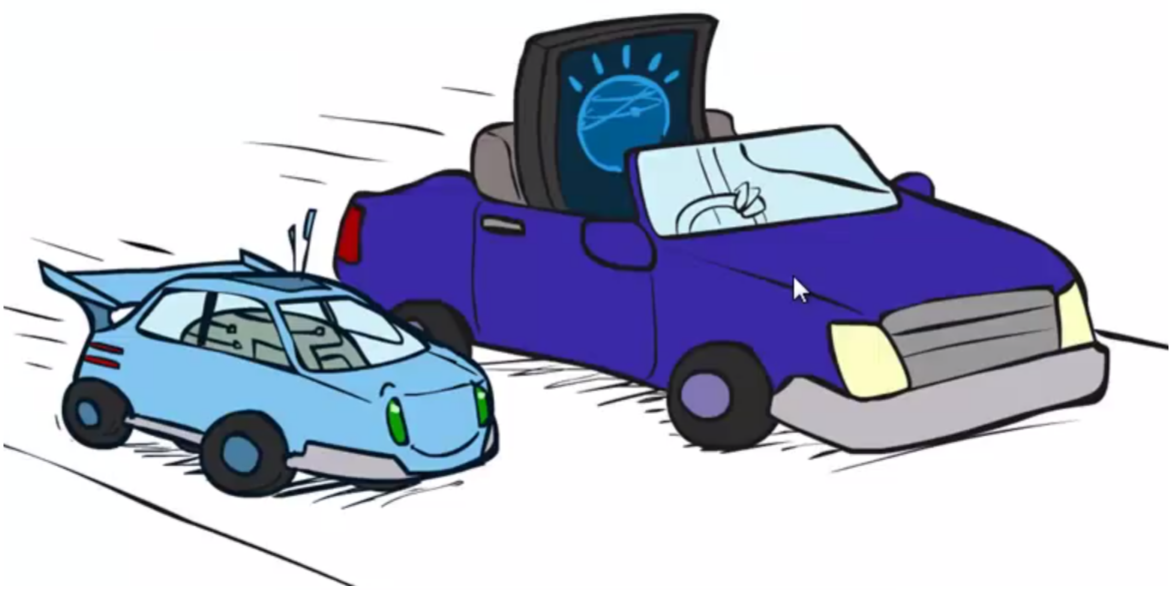




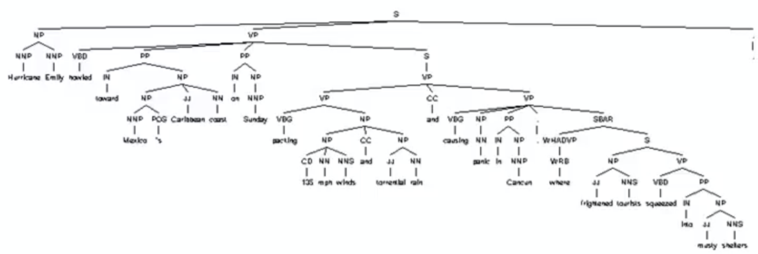



ClusterChecker 9000

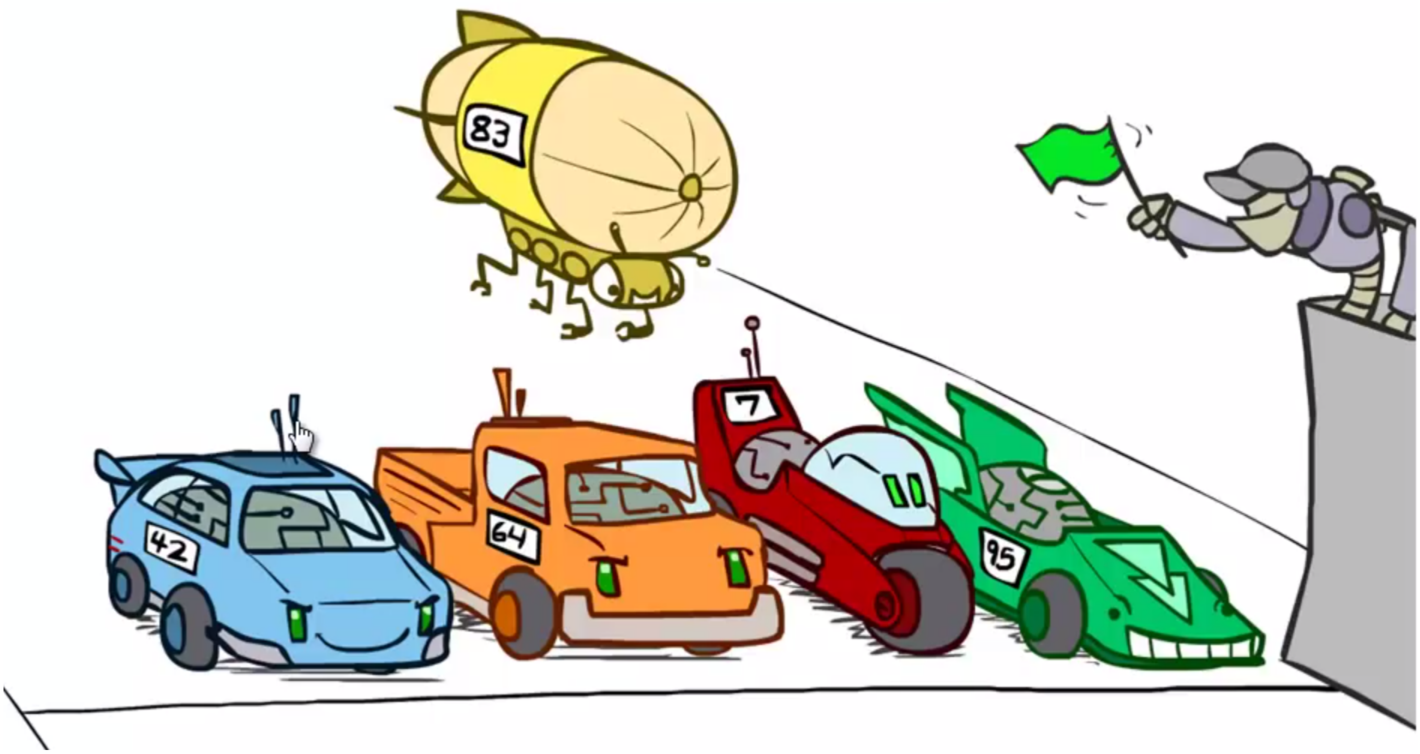


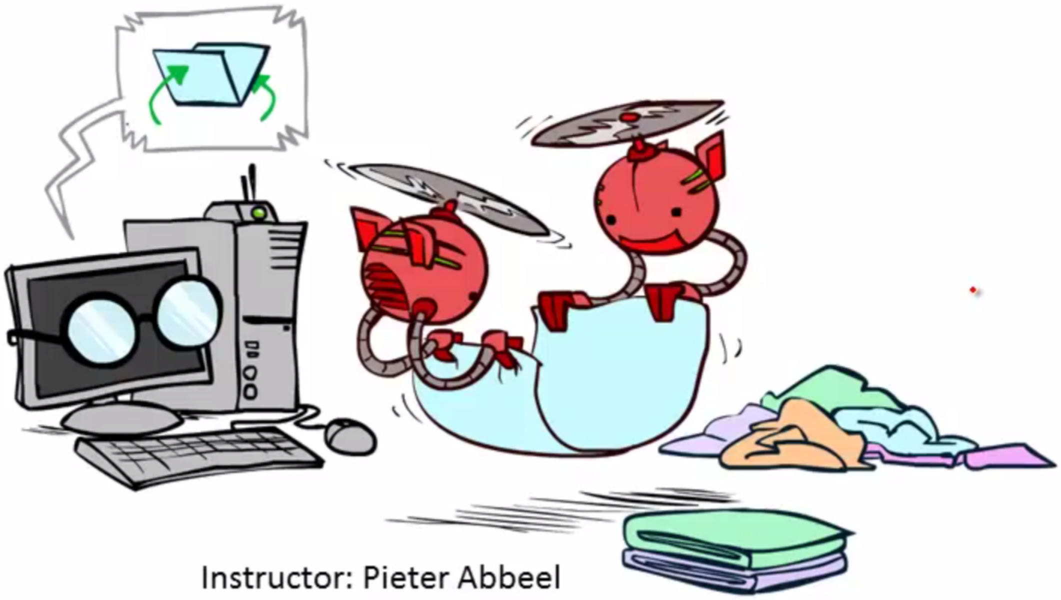


Hershey bars protest

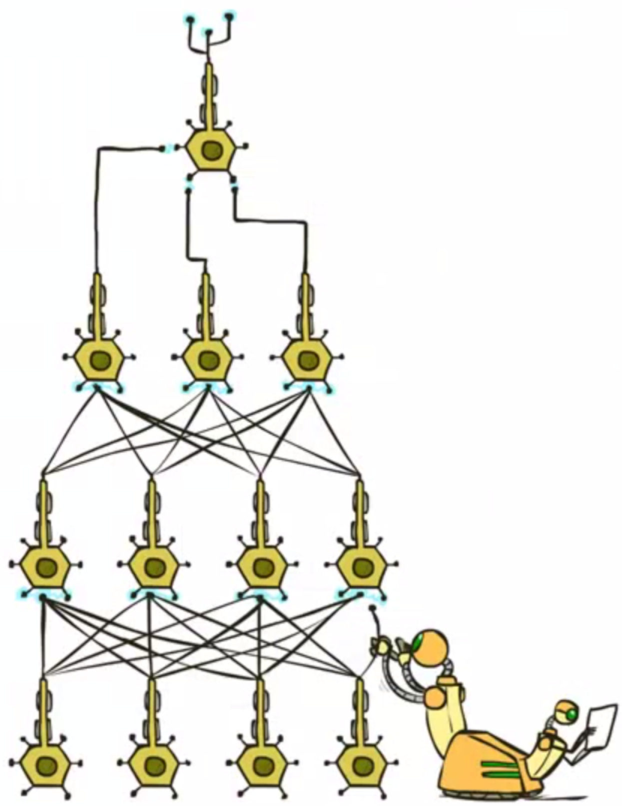


Hurricane Emily howled toward Mexico 's Caribbean coast on Sunday packing 135 mph winds and torrential rain and causing panic in Cancun, where frightened tourists squeezed into musty shelters.


83
42
64
7
95



Instructor: Pieter Abbeel



6.2 Related Topics

- In probability theory, a probability density function (pdf), or density of a continuous random variable, is a function that describes the relative likelihood for this random variable to take on a given value.

-