

UDACITY

Artificial Intelligence

Contents

PART I: FUNDAMENTALS OF AI	2
Welcome.....	2
Problem solving	2
Probability in AI	4
Probabilistic inference.....	5
Machine learning.....	7
Unsupervised learning.....	11
Representation with Logic.....	16
Planning.....	17
Planning under Uncertainty	19
Reinforcement learning.....	21
Hidden Markov Models and Filters	25
Summarizing Table	27
PART II: APPLICATIONS OF AI	30
Games.....	30
Game Theory	32
Advanced Planning	34
Computer Vision.....	35
Robotics.....	39
Natural Language Processing	41
Q&A	47

Skill: 2/3

Time: 4 months

Prerequisites: probability theory and linear algebra.

Instructors: Peter Norvig (Google) and Sebastian Thrun (Stanford, Google).

Summarized by Ido Greenberg, 2016.

PART I: FUNDAMENTALS OF AI

Welcome

1. Applications: finance, robotics, games, medicine, web...
2. “Intelligent Agent” interacts with “Environment”
3. Input is received through sensors, output sent through activators
4. Problems have the following characteristics:
 - a. Fully/partially observable
 - b. deterministic/stochastic
 - c. discrete/continuous
 - d. benign/adversarial (without or with opponent)

Problem solving

Problem definition

1. A problem is defined by:
 - a. A set of states S
 - b. Initial state s_0
 - c. Actions allowed for every state $\{a_i^s\}_{s \in S}$
 - d. Result function $r: S \times A \rightarrow S$
 - e. Goal test function $g: S \rightarrow \{T, F\}$
 - f. Step cost function $c: S \times A \times S \rightarrow R$
 - g. Path cost function $C(s_1, a_1, s_2, a_2, \dots, s_n) = \sum c(s_i, a_i, s_{i+1})$
2. Graph zones: explored / **frontier** / unexplored
3. Problem implementation:
 - a. The explored states can be represented by a set (hash table or tree), to allow membership test (has the state been explored yet?).
 - b. The frontier states can be represented by a priority queue (which is the state from which to spread next?). It should be represented as a set as well (to allow membership test).
 - c. The explored paths can be represented by a list of nodes, each containing 4 fields: final state; action led to that final state; total cost; and pointer to the same path without the final state (“parent”).

Graph search algorithms

Search algorithm	Description	Optimal (finds the best path)	Complete	Frontier size (derives memory usage)
Breadth-first	Next vertex = fewer steps from origin	Yes: searching until all frontier is more expensive than the solution found	Yes: if the best path is of length n , then there's finite number of paths shorter than n , then the optimal one will be found.	Number of vertices of distance n from the origin (may be $O(e^n)$)
Cheapest-first	Next vertex = smaller aggregate cost from origin		The course says yes, though I can design an infinite tree with costs $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \dots$, where the correct path is of cost 1, and will not be found.	
A*	Next vertex = smaller [aggregate cost from origin + estimated distance from destination]	Yes if distance estimation is never pessimistic (otherwise we may give up on beneficial states); better estimation => shorter (more efficient) search	Yes , same as Breadth-first	
Depth-first	Next vertex = go deeper if possible, otherwise go back and try another direction	No: searching until finding any solution	No: infinite search tree may prevent arriving to the correct path. However, if the tree is finite and the destination is very deep in the tree, then DFS may reach it faster than BFS.	All the current path is kept – $O(n)$

1. A* distance **admissible heuristic**: the optimistic heuristic is intended to save time by preventing looking at irrelevant states and paths (e.g. those which get us too far from the destination). Such heuristics may be generated by omitting constraints and solving a simplified optimization problem, yielding optimistic solutions.
2. Problem solving by graph search is possible when the states domain is:
 - a. Fully observable
 - b. Known
 - c. Discrete
 - d. Deterministic
 - e. Static

Probability in AI

Bayes rule

1. Bayes rule along with all the professional terms:

Handwritten Bayes Rule formula with annotations:

$$\underbrace{P(A|B)}_{\text{POSTERIOR}} = \frac{\underbrace{P(B|A)}_{\text{LIKELIHOOD}} \cdot \underbrace{P(A)}_{\text{PRIOR}}}{\underbrace{P(B)}_{\text{MARGINAL LIKELIHOOD}}}$$

Below the denominator, the formula for marginal likelihood is written:

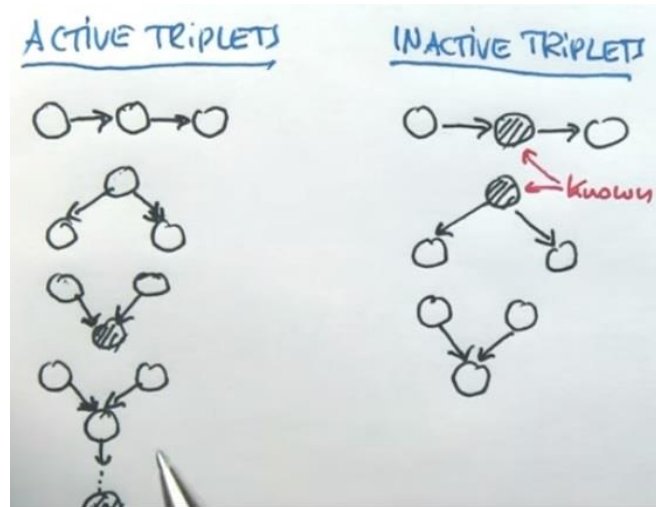
$$P(B) = \sum_a P(B|A=a) P(A=a)$$

Below this sum, the text "TOTAL PROBABILITY" is written.

2. Note that the denominator can be computed by the total probability formula.
3. The denominator may also be deduced from simple normalization, if we have both $P(B|A)P(A)$ and $P(B|\neg A)P(\neg A)$.

Bayesian Network

1. **Bayesian network:** probabilistic graphical model representing random variables with conditional dependencies.
2. The distribution of a variable is determined by the values of its parent nodes.
3. Given n k -nary parents (i.e. with k possible values each), a node would be defined by k^n distributions – one for each combination of the parents' values. The joint distribution of N k -nary variables is determined by $\sum_{i=1}^N (k-1)k^{n_i}$ **parameters**, where x_i has n_i parents; while there are 2^N states.
4. In the example of the car failure analysis, there are ~65K possible combinations of binary variables. Bayesian network reduced the number of specified probabilities to 47, so these networks can compactly keep a lot of information.
5. **D-separation:** two nodes may be either dependent or independent, depending on which other values of nodes in the network are known. The dependence can be tested by terms of nodes triplets, where *active triplets* "pass" the dependence from the first node to the last one, as demonstrated in the following chart:



6. In other words, dependence exists for any of the following:
- There's direct distribution influence.
 - There's indirect distribution influence through unknown variables.
 - There's common unknown parent node influencing both.
 - There's common known child node influenced by both.

Probabilistic inference

- Note: the following inferences and sampling methods make sense in variant cases when the sampled population is available in different ways. For example:
 - Sometimes we control the independent variables and sometimes not.
 - Sometimes we can measure only the individual probability of a variable Y and sometimes only its conditional probabilities wrt different X 's, etc.

Unfortunately, that's hardly explained or demonstrated in the lecture.

2. Enumeration:

- Estimate the probability of $Y = y$ by summation over the probabilities of all the states that satisfy this equality, i.e. $P(Y = y) = \sum_{i_1, \dots, i_n} P(\{X_j = x_{i_j}\}_j \text{ and } Y = y)$, where x_{i_j} is the j 'th possible value of the i 'th variable (assuming it's discrete).
- Such enumeration requires summation over all 2^n states of n binary variables, for example. To speed up enumeration, we can exploit the direct dependencies between the variables. For example, if X_i determines the distribution of X_{i+1} (and the dist' of X_1 is known), then we can calculate the distribution of X_i by induction, using only $2n$ calculations rather than 2^n . This is called **Variables Elimination**, since in each step we practically unite X_i and X_{i+1} into one variable.

3. Approximate inference:

- Sampling:** a distribution can be approximated by just sampling it (simulate the process and measure all variables, like in Monte-Carlo simulation).

- b. Sampling allows estimation of either the complete joint probability distribution $P(X,Y)$ or an individual variable distribution $P(Y)$, making the method *consistent*.
- c. It also allows approximating a conditional distribution given other variable value $P(Y|X = x)$, by using only the samples satisfying $X = x$. This is called **Rejection Sampling**, since we reject irrelevant samples.

4. **Likelihood weighting:**

- a. Rejection sampling is inefficient when most samples are irrelevant (e.g. if we tried to measure the weather given that today a war begins, then we would reject years of samples and keep only a few relevant days).
- b. Instead, we can sample only relevant samples (e.g. only the days when a war was opened) and keep all of them.
- c. If we use this method to compute both $P(Y|X = x)$ and $P(Y|X \neq x)$, then we can approximate $P(Y)$ consistently by $P(Y) = P(X = x) \cdot P(Y|X = x) + P(X \neq x) \cdot P(Y|X \neq x)$. This is called likelihood weighting, since we assign weight to each conditional probability distribution.

5. **Gibbs sampling:**

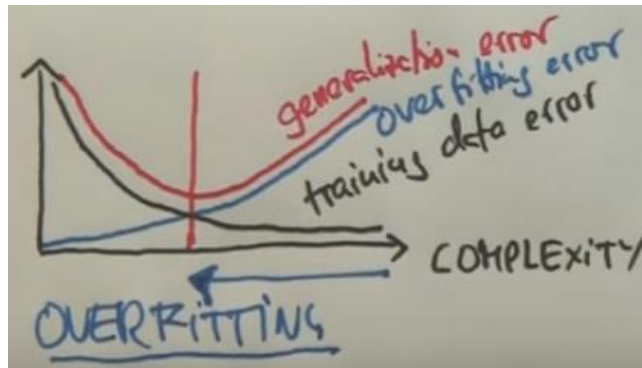
- a. Sampling method intended to be more efficient when desired values of conditioning variables are rare (i.e. for measuring $P(Y|X = x)$ where $P(X = x)$ is small). The idea [Berkman] is to find one such sample, then look for more samples in the same local parameters range, assuming there is higher probability to find $X=x$ in this parametric neighborhood, and assuming the sampling mechanism allows us to control where we look (for example if one of the variables is a location in the world to measure, we can sample the locations in which $X=x$ is more probable). However, this is not explained so well in the lecture.
- b. Gibbs sampling is usually implemented by **Markov Chain Monte Carlo (MCMC)**.
- c. We begin with a certain sample of all the variables. Then, for each **generation**, we sample (once? multiple times?) one of the variables given the current values of the other variables, then change the value of this variable and go on to the next generation. This way, in each generation we change (exactly? at most?) one value among the variables.
- d. Although the samples clearly depend on each other (each one is almost identical to the former), this sampling method turns out to be consistent, i.e. to asymptotically yield the correct complete joint distribution.

Machine learning

1. Buzz buzz buzz – machine learning does everything etc. autonomic car uses machine learning etc.
2. Goal definition:
 - a. **Supervised** – learn data telling us the desired output
 - b. **Unsupervised** – make sense in data without required output
 - c. **Reinforcement** – learn by feedback from environment (as in some genetic algorithms)

Supervised learning

1. Find function $f(x) = y$ that fits given data $\{x_i, y_i\}_i$, in a way that allows generalization of the prediction function f to new data as well.
2. **Occam's (Okham's) Razor**: choose the less complex hypothesis.
3. In practice, there's tradeoff between fit and complexity. The goal is to find the complexity minimizing the *generalization error*:



4. **Overfitting** may be very hazardous. For example:
 - a. 10 points $\{(x_i, y_i)\}$ around a line should be approximated by line rather than 10-degree polynomial.
 - b. Assume that we classify SPAM according to frequency of keywords (using **Naive Bayes** model which assumes independent occurrences), and that one arbitrary word has never occurred in spam messages yet. Then every new message containing this word will be classified as HAM (non-spam), which is clearly overfitting.
5. The general approach to overfitting prevention, is to **reduce the sensitivity of the model's parameters to the training data** – through either model design, data manipulation or Loss measuring.
 - a. In particular, complex models (many degrees of freedom) have higher tendency to overfitting, since it is more probable to have a combination of parameters that “happens” to fit the training data without solving the essential problem of finding a true pattern. That's the case in the 10-degree polynomial.
6. **Laplace smoothing**: when dividing data into classes and counting the occurrences of samples belonging to each class – add K to the count of every class.
 - a. In other words, we take weighted average of the data distribution with unite distribution.

- b. It prevents 0-counters which are dominant in calculations.
 - c. The fit error derived from this smoothness becomes smaller as the number of samples satisfies $N \gg K$.
- 7. **SPAM filtering:** the naive Bayes model described above practically doesn't work for intelligent spammers, so more advanced heuristics are used: known spamming IP, person previously contacted, similar emails for many addressee, capital letters, consistent links texts and URLs, using the name of the addressee...
- 8. **Digit recognition:**
 - a. From 16x16 images of hand-written digits to the corresponding digit.
 - b. Main challenge in input representation is high sensitivity of the image to shifting. It can be partially solved by convolution with smoothing function.
 - c. Naive Bayes is not so good here since it assumes independence between input entries (pixels), but we will use it here for studying.
- 9. **Overfitting prevention:**
 - a. Occam's razor implemented by Laplace smoothing with parameter K – see above. K , as hyper-parameters in general, can be chosen to minimize generalization error, by using **Cross Validation** data.
 - b. The convention is 80% train data (determine parameters by fitting), 10% CV data (tune hyper-parameters for small generalization error), and 10% test data (not involved in the learning process – used only once in the end, otherwise there may be hidden overfitting).
- 10. **Regression:**
 - a. Continuous output values, opposed to discrete labels of **classification**.
 - b. **Linear regression** is of the form $Z = f(X) = W_1 \cdot X + W_0$.
 - c. Goal: minimize the **Loss function** wrt data, which is the L_2 norm of the errors $Y - f(X)$. For linear regression, minimization gives (by $\frac{\partial L}{\partial W_0} = 0, \frac{\partial L}{\partial W_1} = 0$):
 - i. $W_0 = \frac{1}{M} \sum y_i - \frac{W_1}{M} \sum x_i$ (M is the number of training samples)
 - ii. $W_1 = \frac{M \sum x_i y_i - \sum x_i \sum y_i}{M \sum x_i^2 - (\sum x_i)^2}$
 - d. **Logistic regression:** $Z = \frac{1}{1 + e^{f(X)}} \in (0,1)$
- 11. **Regularization:** add **complexity penalty** to the loss function: $L = \text{Loss}(\text{data fit}) + \text{Loss}(\text{parameters})$. Usually It's just L_P norm of the parameters.
- 12. **Gradient descent:** start at some point in the parametric space, and do iterative small steps against the gradient of the loss ($\Theta := \Theta - r \frac{\partial L}{\partial \Theta}$), to gradually reduce the loss. Can numerically find local minimum. Finding the global minimum is tricky and can be studied in *Optimization Theory*.
- 13. **Perceptron:** implementation of linear classification, invented in the 40's. The perceptron is a **linear separator** used to separate 2 classes of samples.
 - a. It is computed by Gradient Descent wrt corresponding loss function, giving $w_i := w_i + r(y_j - f(x_j))$. It converges iff the data is linearly-separable, then it converges to a linear separator.
 - b. Online GD is typically used, i.e. every iteration uses only a batch of samples, possibly new ones.

- c. **Margin** of linear separator = its distance to the closest sample. Perceptron does not deal with maximizing the margin. Most popular tools to achieve maximum margin are **Support Vector Machines (SVM)** and **Boosting**, which are out of the scope of the course.
- d. Briefly, the trick is to add fictive new coordinates (**features**) to the data $x_{n+k} = f_k(x_1, \dots, x_n)$, allowing linear separation between the different classes. For example, in the classic 2 circles separation ($x_1^2 + x_2^2 < R^2$ vs. $x_1^2 + x_2^2 > R^2$), the 3rd coordinate can be $x_3 := \sqrt{x_1^2 + x_2^2}$. In SVMs, large new feature spaces can be generated by something named **Kernel**, which implicitly represent them without actually computing them.

14. **K-nearest neighbors (KNN)**: non-parametric machine learning method!

- a. Learning: just memorize all training data.
- b. Predicting: just find the K nearest neighbors, and choose the majority class label among them.
- c. Assumes local continuity – if many neighbors are of class Y, then so is the new sample.
- d. In all the parametric methods, the number of parameters is inherently independent on the size of the data. In non-parametric methods, the number of “parameters” can grow with the data. In KNN, the “parameters” are actually the whole training data.
- e. As in Laplacian smoothing (see above), K here is practically the smoothing parameter, or the **regularizer**. Higher K derives smoother classification borders, though it allows more outliers, i.e. training samples located in the zone of the wrong class (which may be either good or bad, depending on the problem).
- f. Problem I: large data sets make search long. Organizing data in **kDD-trees** allow logarithmic search.
- g. Problem II: too many input dimensions make the feature spaces too sparse, since to keep the density of the samples, their number should increase exponentially with the dimensions. Thus KNN is typically used only for input of few dimensions, and collapses for 20+ dimensions.

Supervised Learning Algorithms					
Goal	Method		Typical learning	Common regularization	Parametric
Regression	Linear regression		Minimize loss analytically	Complexity penalty in loss function	Yes
	Logistic regression		Minimize loss by GD		
Classification	Naive Bayes		Learn “atomic” probabilities empirically	Laplace smoothing	
	Linear separation	Perceptron	~GD (what do we minimize exactly?)		
	Maximizing margins	SVM	?		
		Boosting	?		
		K-nearest neighbors (KNN)		Memorize training data	Use many neighbors (K)

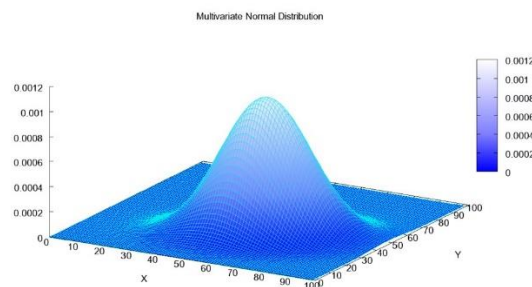
Unsupervised learning

1. “Find structure in data”. For example: data organized in 2 limited zones in R^2 (allowing **clustering**); or data organized along 1 line in R^2 (allowing **dimensionality reduction**).
2. Problem definition:
 - a. Input: iid data samples
 - b. Goal: estimate the density probability distribution of the data
3. *Blind signal separation*: advanced unsupervised learning task, and a special case of *factor analysis*. A signal consisting of the sum of 2 signals needs to be separated (e.g. 2 voices speaking together).
4. Learning new concepts: for example, identify object in Google street view and separate them to clusters (cars, trees, signs...). That’s an unsolved problem.
5. Unsupervised learning is very important since nowadays we can easily have tons of data, but it is difficult to get good labels of these data (required for supervised learning).
6. Very modern algorithms do both unsupervised and supervised learning (**self-supervised** or **semi-supervised**), e.g. by producing labels and applying them.

Clustering

1. **K-Means** clustering:
 - a. Goal: find the K points which best correspond to center of clusters of the data.
 - b. Algorithm: pick K random points in the space, then iteratively until convergence:
 - i. Associate every data sample to the currently closest data center, yielding temporary clustering of the data.
 - ii. Re-define the K data centers to be the centers (i.e. minimizers of L_2 distances) of the temporary clusters.
 - c. Converges to locally optimal clustering. Global optimization is NP-hard.
 - d. Problems:
 - i. Local minima may prevent correct optimization.
 - ii. High dimensionality derives too sparse feature space (as in KNN).
 - iii. Lack of mathematical basis (“you might not care..., but for the sake of this class, let’s just care about it”).
2. **Gaussian learning** (background for EM):
 - a. Multi-variate Gaussian:

$$f(x) = (2\pi)^{-N/2} |\Sigma|^{-1/2} \exp\left(-\frac{1}{2}(x - \mu)^T \Sigma^{-1}(x - \mu)\right)$$



- b. Gaussian learning: what is the best Gaussian fitting the data (i.e. maximizing the likelihood of the data)?
 - c. 1D: mean= μ :=average; variance= σ^2 :=average quadratic deviation (proven by log-likelihood maximization: $\frac{\partial \log(P(x_1 \dots x_M | \mu, \sigma))}{\partial \mu, \sigma} = 0$).
 - d. N-D: generalized formula...
3. **Expectation Maximization (EM)**:
- a. Generalization of K-means, based on actual probability distribution and probability theory basis.
 - b. The difference wrt K-means is that the assignments to classes are “soft” rather than “hard”, i.e. we find the relative correspondence to each class rather than choosing the most corresponding one.
 - c. We assume that the data consist of sum of multi-variate Gaussians (**Gaussian Mixture**), and find the correspondence of every sample to every Gaussian.
 - d. Initialization: pick K random Gaussians centers (μ_i), deviations (Σ_i) and sizes (or priors, π_i).
 - e. **E-step**: given the Gaussians parameters, compute the likelihood of every sample j to belong to every Gaussian i :

$$e_{ij} = \pi_i \frac{1}{(2\pi)^{N/2} |\Sigma_i|} \exp \left(-\frac{1}{2} (x_j - \mu_i)^T \Sigma_i^{-1} (x_j - \mu_i) \right)$$

(up to normalization over all j 's)

- f. **M-step**: given the samples and their correspondence to the Gaussians, find the best Gaussians parameters:

$$\pi_i = \frac{1}{M} \sum_j e_{ij}$$

$$\mu_i = \frac{1}{\sum_j e_{ij}} \sum_j e_{ij} x_j \quad (\text{weighted average of the samples } x_j)$$

$$\Sigma_i = \frac{1}{\sum_j e_{ij}} \sum_j e_{ij} (x_j - \mu_i)^T (x_j - \mu_i)$$

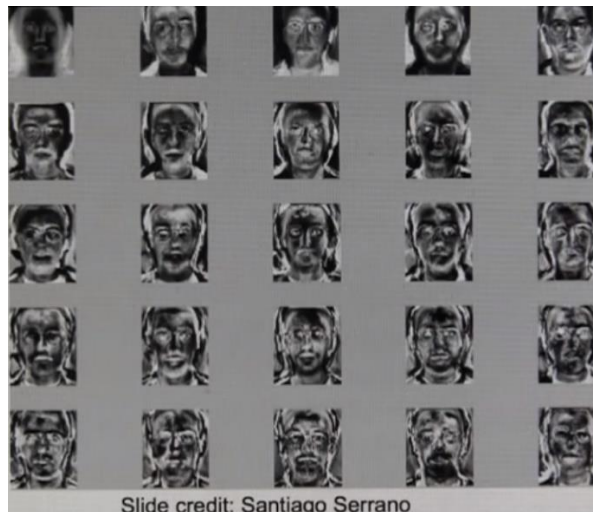
- g. EM converges to local maximum of the likelihood $P(x_j | \pi, \mu, \Sigma)$, or equivalently, to local minimum of the **negative log-likelihood** $-\sum_j \log P(x_j | \pi, \mu, \Sigma)$, which is more convenient for computations.
4. Choose K in EM:
- a. Usually we don't know the number of clusters in advance, thus we need to check that we don't have either missing clusters (i.e. some domain of samples is poorly covered) or spare clusters.
 - b. The popular method is to guess some K , then iteratively:
 - i. Run EM.

- ii. Using the cost function $-\sum_j \log P(x_j|\pi, \mu, \Sigma, K) + cluster_cost \cdot K$ (likelihood vs. cluster cost), check if there are unnecessary clusters (that without them the cost function becomes cheaper) and remove them.
 - iii. If there are samples which are poorly covered (i.e. have some middle ground likelihoods), then add new random classes centers near these locations.
- c. This method overcomes local minima which utilize the K clusters poorly, since it identifies unnecessary clusters and can remove and restart their locations randomly.

Clustering Algorithms (both intended for clusters concentrated around some centers)			
Algorithm	Convergence	Mathematical justification	Dealing with local minima
K-Means	Yes	-	-
Expectation Maximization (EM)		Locally maximizing likelihood	K estimation trick also partially overcomes local minima

Dimensionality reduction

1. Many data sets use more dimensions than needed. For example, faces in 50x50 images can be represented by much less than 2500 coordinates (we know, for example, that those coordinates allow representations of images of many other objects). In the example of faces, a compact representation would use **Eigen-faces** linear basis of around 12 basic faces, which are surprisingly enough to reconstruct a face.



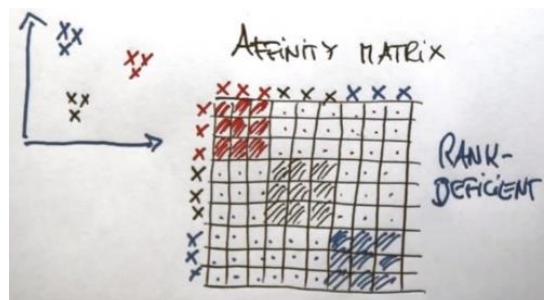
Slide credit: Santiago Serrano

2. This unit deals with *linear dimensionality reduction* only.
3. The main idea is to identify the directions of high variance, keep them only, and project the data onto them.
4. Technically, this is implemented as follows:
 - a. Fit Gaussian to the data
 - b. Compute the eigenvectors of the Gaussian (i.e. of Σ)

- c. Choose leading eigenvectors (by eigenvalues)
 - d. Project data onto the chosen eigenvectors space
- 5. Sounds like **PCA**, although the name is not mentioned.
- 6. **Scan** example:
 - a. Data are scans of people's bodies ("body formations space"). The scan finds the surface of the body.
 - b. The goal is to identify physiques (e.g. thick, tall etc.) and postures (e.g. standing, throwing etc.).
 - c. Linear eigenvector decomposition finds that low linear subspace can express much such information. For instance, 3 dimensions can express variant thickness, height, weight and gender.
 - d. The lecturer did it using a method named SCAPE (Shape Completion and Animation of People).
 - e. Applications:
 - i. Scanning completion (when partial or corrupted)
 - ii. Motion animation
 - iii. Bodies simulations
- 7. Non-linear dimensionality reduction methods:
 - a. **Piece-wise linear projection**: divide samples into clusters (e.g. by K-means), then project every cluster on its own linear subspace.
 - b. **Local linear embedding (LLE)** (sounds like the same as above)
 - c. **Isomap**
- 8. **Spectral clustering**:
 - a. When a cluster is not concentrated around some center (e.g. the samples lay on the edge of some shape), K-means and EM may fail finding the clusters.



- b. Spectral clustering uses **Affinity Matrix** to measure the similarity between every pair of points: $A \in R^{M \times M}$, A_{ij} decreases with $|x_i - x_j|$.



- c. The affinity matrix can be easily decomposed by PCA. Every dominant eigenvector represents a cluster, and every dominant entry in the eigenvector corresponds to a sample belonging to the cluster.

9. In summary:

- a. **Linear dimensionality reduction** can be implemented using the eigenvectors of the covariance matrix Σ of the Gaussian fit (**PCA?**).
- b. **Non-linear reduction** can be implemented using modern methods such as **LLE** and **Isomap**.
- c. **Clustering** which identifies **affinity** (continuity) of data samples can be implemented by **spectral clustering**.

Representation with Logic

The idea is to use the tools of logic (in particular *first order logic*) to model the world by compact representation.

Propositional logic

1. Binary variables (symbols) – either T or F (non probabilistic).
2. *Model* = assignment of Boolean values to the variables.
3. *Truth tables* – the basic rules of logic: definition of and/or/not/implies/equivalence.
4. Given axioms (sentences assumed to be true), every variable value is T, F or unknown (as said before, no probabilistic terms).
5. Sentences may be **valid** (true for every model), **satisfiable** (true for some model) or **unsatisfiable** (false for every model).
6. The main strength of propositional logic is to identify sentences which are logically valid or unsatisfiable, independently of the assumptions (model).
7. Limitations of propositional logic:
 - a. Uncertainty – no probabilistic terms
 - b. Objects – no objects, properties or relationships – only sentences (like in non-OOP)
 - c. Shortcuts – no compact tools to represent general sentences such as "all the kids are short" – we need to say it for every one by itself...

First Order Logic

1. Deals with limitations b & c of propositional logic (see above), by defining objects and general statements.
2. The advanced representation of objects is demonstrated in the table:

Theory	World representation	Beliefs
First Order logic	Structured representation: Relations, Objects (variables & constants), Functions	T/F/?
Propositional Logic	Factored representation: statements combining atomic representations by logic gates	
Probability Theory	Atomic representation: just different states which may either be satisfied or not	[0,1]

3. The shortcuts are allowed by the advanced syntax, including:
 - a. Atomic sentences (expressing relations between objects)
 - b. Operators combining sentences
 - c. Quantifiers (**all**, **exist**) allowing general phrases

Planning

Background: planning in stochastic and partially observable world

1. AI – creating an agent which performs actions according to the situation → planning is the core of that.
2. Problem solving (section 2, e.g. A*) is good for deterministic & fully observable problems:
 - a. It uses planning and executing, where the execution is "blind", i.e. doesn't get feedback from the environment and doesn't update the plan.
 - b. Nice experiment showed that people don't manage to walk in straight line without feedback from the environment.
3. Properties of real problems (some of them seem mathematically equivalent):
 - a. Stochastic – the results of an action may be non-deterministic by terms of the plan (e.g. drive straight somewhere should depend on the light in the traffic light, which is not part of the model of the world used for planning, thus may yield unexpected results).
 - b. Multi agent – other agents take actions as well.
 - c. Partial observability – for example a certain road may be closed and it may be announced in a sign before the road, i.e. not known in advanced.
 - d. Unknown – e.g. incomplete map or inaccurate GPS.
 - e. Hierarchical – the plan cannot include steps like "push the gas" but only high level instructions, so low level actions should be determined interactively.
4. This section:
 - a. Goal: problem solving – i.e. decide whether a problem is solvable and find a plan to solve it – in **stochastic** and **partially observable** world.
 - b. Limits (next section): no distinguish between **probable** and improbable situations.

Belief state space

1. States are replaced by **sets of states** (same idea as Non-Deterministic Automats). A set contains all the currently-possible states. The exact state is unknown in case of partial observability (e.g. local observability).
2. Goal: reach a subset of goal states.
3. We need to reduce the set of states accordingly.
4. Actions may have variant effects on states subset size:
 - a. Reduce the size if different input states are **mapped into the same output** states.
 - b. Reduce the size **through observations** which remove false states. Observations are represented by nodes (in addition to the standard decision/action nodes), and the agent does not control the output of these nodes.
 - c. Increase the size if the action is **stochastic**, i.e. one input state may have variant output states.
5. In stochastic world, sometimes no classic plan can solve the problem. For example, if we wish to get to the right, but every right move may fail and remain in the same place, then no sequence of right moves guarantees actually getting to the right. Thus new notion of plan is required.

6. **Infinite sequence** of actions can be defined by conditioned actions based on observations. In the last example, it can be "[(move right) while (location is left)]". This notion adds branches to every plan (in addition to the branches in the decision tree, which decide between plans).
7. A plan is *successful* iff all the leafs in the plan tree are goal states.
8. A solution is *bounded* in time iff its tree has no loops.
9. Compact representation of belief states (i.e. sets of states) can be used by descriptive variables. For example, if the world is defined by 4 binary variables, sometimes a set of 8 states (out of the possible 16) can be described just by the value of one of those variables.

Classical planning

1. Problem definition:
 - a. State space: K-Boolean (2^K states).
 - b. World state: complete assignment (every variable has value).
 - c. Belief state: complete or partial assignment, or even formulas of variables.
 - d. Actions: every action has *pre-condition* and *effect* in terms of relations of objects (as in logic).
2. Finding classical plan:
 - a. **Forward/Progression search**: start from initial state and search over possible actions.
 - b. **Backward/Regression search**: start from goal state and search backward over sufficient actions (e.g. if I want to buy a certain book, I won't go over all possible purchases, but rather look at the purchase which can end up with me having that certain book, i.e. Buy(this_certain_book)).
 - c. **Plan space search**: start from abstract plan like "[initial state \rightarrow goal state]", and refine the plan until it's complete. It's unclear in the lecture how to do that.
3. Nowadays, **forward search is the most popular** of the three, since modern search plans use **heuristics** to reduce the space of possible choices, and heuristics work more efficiently with concrete possible actions (rather than hypothetical actions and states as in regression and plan space search).
4. Heuristics can be achieved by simplified actions, ignoring pre-conditions or negative effects. Having the representation of actions, the heuristics can be defined automatically by the program.

Situation calculus

1. Represent planning in terms of First Order Logic, to allow advanced goals such as "move all cargo to this airport".
2. Since the problem is represented as a theorem in FOL needed to be proved, it **can be solved by a general theorems prover of FOL**.
3. Representations:
 - a. Actions and situations are represented by objects.
 - b. Pre-conditions are formulas of the form "Conditions \rightarrow Possible(certain_action)".

- c. Conventionally, an action is a function whose last argument is the input state.
- d. Initial state and goal are defined by sets of formulas.
- 4. **Successor state axiom** – defines the flow of states:
 - a. $\forall \text{action}, \text{state}: \text{Poss}(a, s) \rightarrow [\text{some fluent is true iff } (a \text{ made it true OR it was true in } s \text{ and } a \text{ didn't undo it})]$.

Planning under Uncertainty

MDP

1. **Markov Decision Process** is planning tool for stochastic, fully observable environment.
2. The world's representation is a graph of states (nodes) and actions (edges) as in conventional planning (e.g. situation calculus), except that the transitions between states are not deterministic, but rather based on probabilistic transition matrix, i.e. every edge splits in the middle.
3. Problems with conventional planning (searching over decisions tree):
 - a. Large *branching factor*: Decisions trees grow very rapidly, especially when there are many possible actions outcomes (as in stochastic environment).
 - b. Trees may become too deep. In particular, loops (caused by uncertainty) make them infinitely deep.
 - c. The same states are visited many times in the tree, which is very inefficient representation, since usually we would like to do the same action in the same state.
4. **Policy**:
 - a. For every state, choose the best action among available ones.
 - b. Compact representation – every state appears only once.
 - c. For every state we define reward/cost achieved by getting to the state. The value of a plan π applied from a state s is $V^\pi(s) = E[\sum_t R_t | s_0 = s]$.
 - d. Expressing cost of time:
 - i. Add constant cost to every state, giving motivation to avoid visiting the state repeatedly.
 - ii. Add **discount factor** reducing the reward when loosing time, by defining the value as $E[\sum_t \gamma^t R_t]$.
5. **Value iteration**: how to find a policy with high value?
 - a. Define value for every state, derived from both its inherent value and the values of its neighborhood:
 - i. Start with the inherent values of the states $\{v(s) = R(s)\}$.
 - ii. Every iteration, re-define the value of every state, by assigning the best expected value that can be achieved in the following step (**back-up equation**):

$$v(s) = \left[\max_a \gamma \sum_{s'} P(s'|s, a) v(s') \right] + R(s)$$

- iii. (Exceptions are terminal states remaining with $v(s) = R(s)$.)
 - iv. Convergence is guaranteed.
 - b. Now the optimal policy is just the actions which maximize the expected values of the states (same as the back-up equation, but with argmax rather than max ...).
6. Summary:

MARKOV DECISION PROCESSES

FULLY OBSERVABLE $s, \dots, s_n, a_1, \dots, a_n$

STOCHASTIC $P(s' | a, s)$

REWARD $R(s)$

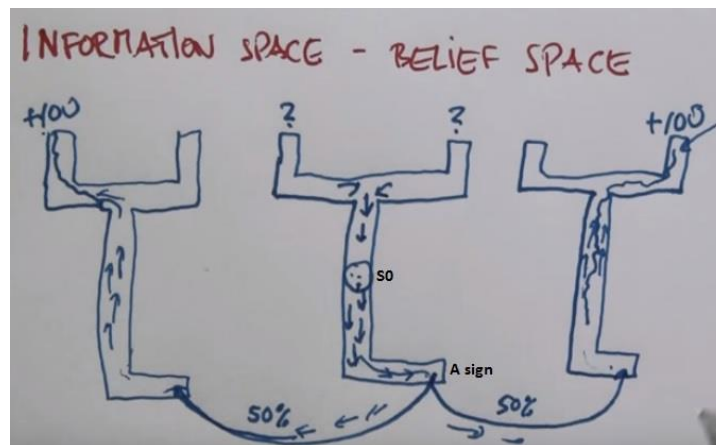
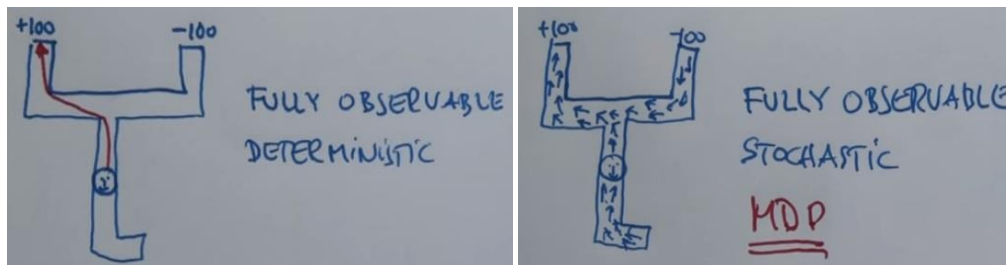
OBJECTIVE: $E \sum_t \gamma^t R^t \rightarrow \max$

VALUE ITERATION: $V(s) \quad Q(s, a)$

CONVERGES: $\pi = \text{argmax} \dots$

POMDP

1. MDP cannot express information gathering, since it represents fully observable world.
2. **Partially Observable Markov Decision Process**: represent partial observability by duplicating the whole states space according to available information. Gathering information transforms us to the relevant copy of the states space.
3. Demonstration: A* (fully observable & deterministic world), MDP (fully obs. & stochastic), and POMDP (partially obs. & stochastic).



Reinforcement learning

1. In the previous sections we tried to get to states with large reward. In this section we will see how to plan when we don't know the rewards of the states and the probabilities of transitions between them.
 - a. Quite similar to the challenge of POMOP. Maybe the difference is that here we also use machine learning (they said something about it before...).
2. **Reinforcement learning** was successful, for example, in playing backgammon (6-besh) and controlling helicopter remotely.
3. Reinforcement learning is a form of learning, additional to supervised ($\{x_i, y_i\}$) and unsupervised ($\{x_i\}$) learning. In this form of learning, we have sequences of states and actions ($\{s_i\}, \{a_i\}$), and rewards associated with certain states.
4. Goal: find the optimal policy, i.e. what is the optimal action in every state.
5. Types of agents of reinforcement learning:
 - a. **Utility-based agent**: given the transitions probabilities P , it learns the rewards $R(s)$ of the states and uses P and R together for planning (as before).
 - b. **Q-learning agent**: learns the utility function $Q(s, a)$ of performing certain actions on certain states, then uses Q without explicitly associate rewards $R(s)$ to the states.
 - c. **Reflex agent**: learns the optimal behavior on every state $\pi(s)$ and uses it even without predicting the utility.

	R ??	P ??	
agent	know	learn	use
Utility-based agent	P	$R \Rightarrow U$	U
Q-learning agent		$Q(s, a)$	Q
reflex agent		$\pi(s)$	π

6. **Passive RL vs. Active RL**:
 - a. Passive reinforcement learning learns the environment on-the-way, i.e. without affecting the policy of actions.
 - b. Active reinforcement learning learns the environment and changes the policy interactively – both to improve the utility and to allow additional exploration of the environment.
7. **Passive Temporal Difference learning**: a basic passive RL method, assuming the rewards of states are locally observable.
 - a. Basic idea: run some policy a lot of times, and every time update the utility of every state according to the utility of the one followed it.
 - b. Algorithm: use some arbitrary policy π and run it iteratively from an initial state. On every run, on every step of the policy:
 - i. If the output state s' is new: define its utility $U(s') := r'$.
 - ii. Increment the counter of visits at the input state $N(s)$. The *learning rate* will depend on N , since we wish to use smaller updates as we have more visits (hence more confidence) in the state.

- iii. Update the utility of the input state s according to its own reward r , the output state utility $U(s')$, and some learning rate $\alpha(N(s))$ (γ is discount factor as in previous section):

$$U(s) := U(s) + \alpha(N(s))[r + \gamma U(s') - U(s)]$$

- iv. Optional: update previous states as well (that changes the algorithm from **TD(0)** to **TD(1)**).

1. Meaning of the modification: in $s_1 \rightarrow s_2 \rightarrow \dots \rightarrow s_n$, $U(s_1)$ will give up on $U(s_2)$ in favor of $U(s_n)$, while summing all $r_1 \dots r_n$ on the way.
 2. TD(0) updates $U(s_1)$ with info about s_2 taken from all the data of s_2 (using $U(s_2)$ directly).
 3. TD(1) updates $U(s_1)$ with info about s_2 taken only from scenarios in which s_2 followed s_1 (replacing $U(s_2)$ by the actual future rewards in those scenarios). Hence TD(1) does not assume the Markov Property (s_2 is used for $U(s_1)$ merely in scenarios where $s_1 \rightarrow s_2$), but as a result it exploits much less data, and is sensitive to the paths of states that occurred in the specific scenarios that passed through s_1 .
- c. It is also possible to update K -steps back (rather than 1 step as in TD(0) or ∞ steps as in TD(1)). Usually a weighted mixture of K 's is used, where the weighting is implemented through a parameter $0 \leq \lambda \leq 1$, yielding **TD(λ)**.
 - d. This method performs not so good, mostly because it is passive:
 - i. The chosen policy affects both the sampled states and their estimated utilities.
 - ii. It may miss some states, not assigning any utility to them.
 - iii. It may visit states only few times because of low probability – yielding unreliable estimates.
 - iv. It takes a lot of time to converge at all.

8. **Greedy reinforcement learner**: active RL method.

- a. Same as *passive temporal difference*, but updates the policy π every one or several iterations.
- b. The policy is updated to be optimal in terms of MDP wrt the current utilities estimates.
- c. Greedy \rightarrow may converge to non-optimal policy.
 - i. Solution 1: try frequent random deviations from policy (as in genetic algorithms) – can work but converges very slowly.
 - ii. Solution 2 – **Exploration Agent**: force assignment of constant large utility $U(s) := R$ to states that were not explored enough yet ($N(s) < C$), so that the agent will tend to explore them. When $N(s) \geq C$, go back to assign the true estimator of $U(s)$.

9. **Q-learning**:

- a. The exploration agent can get estimation of the rewards of the states $r(s)$. However, unless we know the transitions probabilities $P(s'|s, a)$, we cannot derive the correct policy.

- b. Thus, Q-learning gives up on learning P and U in order to maximize $\sum_{s'} P(s'|s, a)U(s')$, and instead tries to maximize directly $\sum_{s'} Q(s, a)$.
- c. The algorithm is similar to the exploration agent, but with exploration over pairs (s, a) rather than states only, which may enlarge the exploration space significantly.
- d. The iterative utility estimation is generalized from TD (Temporal Difference):

$$Q(s, a) := Q(s, a) + \alpha[r(s) + \gamma Q(s', a') - Q(s, a)]$$

10. **Function Generalization** to large state spaces:

- a. The methods described in this section have to explore the states space, which is typically huge.
- b. The practical solution is:
 - i. Explore only some of the states.
 - ii. Identify conceptually similar states in order to generalize learning to un-explored states.
- c. To implement that, we have to represent the states space in advance by terms of relevant features rather than the “exact real state”.
 - i. Too many features \rightarrow too big states space + unimportant features prevent generalization between similar states.
 - ii. Too few features \rightarrow essentially different states are identified as similar.
- d. In terms of features $\{f_i\}$, the utility can be formed as $Q(s, a) = \sum_i w_i f_i$, where the goal of the learning process is to update the weights $\{w_i\}$ of the features.
- e. This method essentially forms the agent problem as supervised learning problem:
 - i. The input is the features array $f_1 \dots f_d$.
 - ii. The output is the utility $Q(s, a) = Q(\{f_i\})$.
 - iii. The “brain” model is $Q = \sum w_i f_i$.
 - iv. The learning is based on the empirical utilities of the explored states and actions.

11. In summary, we saw several methods, each one based on the former, and only the last one is practically scalable to effectively explore large states spaces:

Method	Description	Type	Exploration	Local minima overcome	Exploration space	Convergence
Passive Temporal Difference (TD)	Run policy iteratively & update utilities	Utility-based (<i>P</i> is required)	Passive	No	States of current policy only	Very fast
Greedy Reinforcement Learner	TD + Update policy by MDP on estimated utilities		Active		States of greedy policy	Fast
Random Reinforcement Learner	Greedy + Add random deviations from policy			Yes	All states	Slow
Exploration Agent	Greedy + Add deviations towards un-explored states					Fast
Q-learning	Exploration over states and actions			Q-learning	States & actions	?
Generalization	Q-learning + features-based utility	Q-learning + supervised	Features	?		

Hidden Markov Models and Filters

HMM

1. A lot of buzz introduction – very effective in robotics, medical, finance, speech recognition, language...
2. **HMMs** are used for analysis and prediction of **time series**.
 - a. **Prediction** = what is the state going to be at time t ?
 - b. Analysis = **state estimation** = given certain measurements at times $1, \dots, t$, what is the (*internal* or *hidden*) state at time t ?
3. Time series can be represented by simple case of Bayes network – $\{s_i \rightarrow s_{i+1}\}$ – forming a **Markov Chain**. Every state emits a **measurement** $z_i(s_i)$, which is the only observable thing in the process (thus the state is *hidden*).
4. This model is also the core of probabilistic filters such as **Kalman Filters** and **Particle Filters**.
5. **Localization Problem** Example: A robot navigates a building, and wants to know its location given a map and range sensors. In this case, the time series is the location as function of time, and the measurements are the ranges measured to the walls. An advanced variant of the problem is not having the map, and building it from scratch on the fly.
6. Markov chain can be represented by **transitions matrix** $P_{ij} = P(s_{t+1} = j | s_t = i)$. A **stationary distribution** of such chain is the asymptotic distribution, and satisfies $\forall i: P(s = i) = P(P \cdot s = i)$.
 - a. **Ergodic** Markov chain = has unique stationary distribution = initial state knowledge fades over time. Rate of convergence to stationary distribution is called **mixing speed**.
 - b. Estimating the transitions matrix: it can be found empirically using maximum likelihood, i.e. by $P_{ij} := \frac{|\{s_t=i \text{ and } s_{t+1}=j\}|}{|\{s_t=i\}|}$, possibly using Laplacian smoothing (i.e. adding some uniform-distributed samples to prevent overfitting).
7. **State estimation**:
 - a. Our problem is hidden, i.e. we know only the measurements rather than the states. Thus, the empirical estimation has to be indirect, based on $P(z|s)$. The perdition has to be based on both $P(z|s)$ and $P(s_{t+1}|s_t)$.
 - b. The estimate of the current state given past measurements is essentially a convolution of the distributions of the states given the measurements $\{P(s_t|z_t)\}_t$, each normalized according to prior and transitions probabilities $\{P(s_t|s_{t'<t})\}_t$.

Particle Filters

1. **Particle filters** basic idea:
 - a. Represent the belief state using "particles" randomly assigned to various actual states.
 - i. Allows us to work with continuous spaces.
 - b. Every action changes the state of every particle accordingly.
 - c. Every measurement changes the probability of every particle to represent the correct state. Particles of very low probability can be omitted (*filtered*).

2. Sampling-based Implementation:

- a. Rather than keeping “actual particles” along with their probabilities, we keep the distribution of the current state. Every step, we *resample* the particles multiple times using this distribution, then use the samples to produce the distribution of the next step, based on the new measurements.
- b. In this implementation, computational resources are proportional to state likelihood, producing inherent computational efficacy.

3. Algorithm:

- a. Input:
 - i. $S = \{(x, w)\}$ initial particles with uniform importance weights
 - ii. U controls (transitions matrix?)
 - iii. Z new measurements
- b. Until convergence of particles distribution:
 - i. $S' = \phi$
 - ii. For $i = 1 \dots n$
 - 1. Sample a particle s_i from the distribution of S
 - 2. $x' \sim P(x'|U, s_i)$ predict next state of s_i
 - 3. $w' = P(z|x')$ assign importance weight according to z
 - 4. $S' = S' \cup \{(x', w')\}$ update new particles set
 - iii. $w' = w' / \text{sum}(w')$ weights normalization
 - iv. $S = S'$
- c. That easy to implement!

4. Problems:

- a. High-dimensional spaces require exponentially many particles to fill the space.
 - i. **Rao-Blackwellized particle filters** and others try to generalize for high dimensions.
- b. *Degenerate conditions* – too few particles, too deterministic environment, etc. – this is unclear. Probably some randomization is needed to guarantee effectiveness.

5. Advantages – which make particle filters VERY useful in many applications:


- a. Easy to implement.
- b. Manage computational resources efficiently.
- c. Can deal with complex posterior distributions containing many peaks.

Summarizing Table

Planning

Methods			Main Idea			Robustness: Conditions Allowed			Properties				Examples		Comments		
Category			Method	World Representation	Brain	Learning	Stochastic	Partially Observable	Continuous	Optimality	Completeness	Time	Memory	Demonstration	Applications		
Planning – agent in environment chooses actions in states	Searching – find a desired state	Graph search – discrete world	BFS (Breadth First Search)	Graph of states & actions	Expand close nodes	-	X	X	X	V	V	$O(n+m)$ (nodes & edges)	$O(\exp(h))$	tree search	path planning in roads map	I think I have example for incompleteness: tree with one branch of costs 1/2, 1/4, 1/8... and one branch of one cost 1. CFS & DFS would fail all the same...	
			CFS (Cost First Search)		Expand cheap nodes							$O(n+m)$ - but more efficient than BFS					
			DFS (Depth First Search)		Expand deep nodes					X	X (for infinite tree search)	MAY be faster if destination is deep and we are lucky...	$O(h)$ (h is height of tree)	deep & branching tree under memory constraints; or just when destination expected to be close to a leaf rather than to the root	usually staff dealing with connectivity components		
			A*		Expand heuristically cheap nodes					V (assuming optimistic heuristic)	V	$O(n+m)$ - more efficient as the heuristic is better	$O(\exp(h))$	tree search	path planning in roads map		
			Situation Calculus	FOL – objects, variables & formulas	General theorems prover					?	X	V	?	V	?	?	"move all cargo to certain airport" using well-defined actions
		World search – continuous world	Particle filters	Variables describing states; particles representing beliefs	Actions -> new observations -> assign probabilities to particles	Possible to use any Reinforcement learner while exploring	V	V	V	not defined w.r.t unknown environment	X (crowded particles required)	~ N_particles X N_steps	~ Number of particles (exponential w.r.t dimensionality)	Self localization in 2D/3D environment	quite versatile and not exactly search algorithm; managing computational resources on the fly according to updated probabilities		
	Rao-Blackwellized particle filters		?												generalization of particle filters to be scalable to high dimensions		
	Planning under uncertainty – find beneficial policy	Mostly observable world	MDP (Markov Decision Tree)	Graph of states & actions	Value iteration + greedy maximization	-	V	X	X	V	V (assuming finite states space...)	Going over nodes rather than paths - more efficient for loops and many branches	~ Number of nodes	maze solver with possible loops	driving in a well known environment		
			POMDP (Partially Observable MDP)	Graph of states & actions – duplicated by unobservability. Observations move us to another copy of the states space				little and defined (each unobservable node duplicates the states space exponentially)					~ Number of nodes X (possible options for unobservable nodes ^ number of unobservable nodes)	maze solver with few signs needed to be read	some unclear uses in wikipedia, e.g. conservation of Sumatran tigers		
	Reinforcement learning – learn beneficial policy (sà a)	Utility-based agent – learn utilities of states	TD (Temporal Differences)	Graph of states & actions (unknown rewards)	Any planning algorithm (the agent just finds the rewards). Transitions probabilities are needed to finally derive the policy from the rewards, e.g. by MDP.	Variant of value-iteration - but the rewards are observed while applying some policy (rather than known in advance)	V	V (unobservable rewards, observable transitions)	X	X (explored states and estimations are sensitive to chosen policy)		Very slow (passive -> policy is not intended to search)		learn utilities of ship navigation WITHOUT the authority to influence the navigation	playing backgammon, controlling a helicopter, etc.		after the first empirical iteration, why do we need to repeat the experiment, rather than just applying value iteration based on the observed inherent rewards? Note: transitions probabilities required for deriving policy from the rewards found
			Greedy learner			As in TD, but with updating the policy once a while according to the estimated rewards (active rather than passive)				Greedy (explored states and estimations are sensitive to policies)				learn utilities of ship navigation WITH the authority to influence the navigation			
			Exploration agent			As in Greedy learner, but with additional cost to explored states - forcing further exploration				Overcoming local minima	V (assuming the already-explored cost is large enough)						
		Q-learning agent – learn utilities of actions in states	Q-exploration agent	Pairs of states & actions	Just pick the max of U(s,a) given s	As in Exploration Agent, but measuring U(s,a) rather than R(s)	V (both rewards & transitions)	V	V	Overcoming local minima; sensitive to too simplified representations		Up to properties of states which are not represented		efficient representation of states space allows scalability	learn how to play packman (lots of states but many states are similar)		
			Function Generalization	Features characterizing the states efficiently													
		Reflex agent – learn utilities of policies	-														was not discussed except from the main idea - learn what to do in every state, even without predicting the utilities of the possible actions

Data Learning

Methods			Main Idea		Properties			Examples		Comments		
Category			Method	Model ("Brain")	Learning - minimize Loss	parameters	scalability	margins maximization	Demonstration	Applications	Comments	
Data learning – analyze data and generalize to unknown data	Supervised – learn to predict output from input (data={(x,y)})	Regression - output is continuous or at least ordered	Linear	$W1 \cdot x + W0$	analytic minimalization	$(n+1) \times m$		-				
			Logistic	$1/(1+\exp(W1 \cdot x + W0))$								
			Neural Networks	linear + pairwise nonlinear + composition	GD		~ as above \times number of layers					
		Classification - output is discrete	Probabilistic Inference	Bayesian Network	experimental sampling	for N k-nary vars with n_i parents each: $\sum_{i=1}^N (k-1) k^{n_i}$	The network structure allows reduced number of basic probabilities (as noted in the left, wrt k^N)	-		SPAM filtering	1. Naïve Bayes is a simplified case of disconnected net (i.e. independent variables). 2. The principles of D-separation and Variables Elimination allow computing and measuring certain dependencies in the net.	
			Perceptron	$\text{sign}(W1 \cdot x + W0)$	GD	$(n+1) \times m$		X				
			SVM	feature coordinates $x_{(n+k)}(x_1, \dots, x_n)$ + linear separation	?	?	implicit representation of features is somehow supposed to allow scalability	V?	define $x3=(x1^2+x2^2)$ to allow classification by radius			
			Boosting	?	?	?		V?				
			KNN	majority class among nearest neighbors	memorizing data	0	only for $< \sim 20$ dimensions - data density requires exponential increase	-				
			HMM	Hidden Markov Chain + measurements of its sequential variables	experimental sampling	n^2 transitions probabilities for n states		-		Analysis & prediction of time series in robotics, medicine, etc.		
	Unsupervised – learn structures in data and represent their distribution (data={x})	Clustering	K-means	K centers of clusters	randomize centers; then until convergence: {assign clusters; update centers}	K	data density requires exponential increase in dimensionality	-			Not strong enough mathematical basis.	
			EM	K distributions of clusters	As in K-means, but cluster assignments and distribution updates are probabilistic rather than deterministic	$K \times$ (parameters per distribution)					K can be determined dynamically, through: 1. detection and removal of clusters with low contribution to the log likelihood; 2. detection of "middle ground" likelihoods and addition of close centers.	
			Spectral Clustering	Affinity matrix (based on some function of similarity/distance between data points)	PCA of the affinity matrix (representing continuity of data points) + assignment of clusters according to dominant eigenvectors							
		Dimensionality reduction	PCA	Linear coordination transformation	Fit data by one multi-dimensional Gaussian, and choose coordinations according to its leading eigenvectors	only cutoff (determining number of dimensions)				Eigen-faces, Body scan properties	Was not named PCA in the lecture, but I'm quite sure this is it.	
			LLE	Piecewise (local) linear coordination transformation	Clustering (e.g. K-means) + linear projection (i.e. PCA) of each cluster	number of clusters AND dimensional cutoff for each cluster					The lecturer talked about Piecewise linear projection and Local Linear Embedding, which are probably the same one.	
			Isomap		?							Was not explained in the lecture, except for being non-linear.
			Spectral Clustering	Affinity matrix (based on some function of similarity/distance between data points)	PCA of the affinity matrix (representing continuity of data points) + assignment of clusters according to dominant eigenvectors							

Regularization of Supervised Learning

Regularization = overfitting prevention = reducing the sensitivity of the learnt parameters to the training data.

1. Data manipulation – **Laplace smoothing**: "smooth" the train data by adding fictive uniformly-distributed samples – preventing overfitting caused by non-representative data.
2. **Cross validation**: measure generalization error rather than training error.
3. **Complexity penalty** in Loss function: allow only "cost-effective" influence of the data on the learnt parameters.
4. Design of **insensitive model**:
 - a. **Reduce degrees of freedom** (e.g. reduce number of parameters in NN or in polynomial interpolation).
 - b. **Avoid too local learning** (e.g. use large enough K in KNN).

PART II: APPLICATIONS OF AI

Games

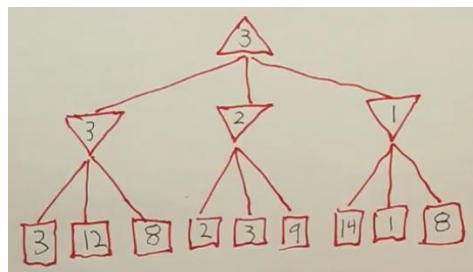
General

1. Games usually have strictly defined rules and environment, hence they are convenient for AI applications.
2. However, the methods learnt so far were not intended for adversarial tasks (against opponents).
3. Different games have different properties in terms of AI tasks. For example, wrt stochasticity, partial observability, unknown and adversary, we have:

St	P	O	U	A	
○	○	○	●		Chess, Go
●	●	○	●		Robot Soccer
○	●	○	●		Poker
●	●	●	●		Hide-and-go-seek
○	●	○	○		Cards Solitaire
○	●	○	○		Minesweeper

Chess modeling

4. In chess-like games, where we have **2 players** with **turns** and eventual **0-sum profit**, the game can be modeled by a tree, in which every node is either:
 - a. **"maximization node"** of player I
 - b. **"minimization node"** of player II (minimizing p1 profit = maximizing p2 profit)
 - c. **"value node"** defining the profit of the players directly
 - d. **"chance node"** representing stochasticity and evaluated by expectation
5. The values of the nodes and the strategies of the players can be derived by backward-induction (or recursively) from the value nodes, through the maximization & minimization nodes. For example:



6. In chess we have about $m=30$ moves per game, with about $b=40$ possibilities per move, yielding about $b^m = 40^{30} = 10^{48}$ nodes, which cannot be practically searched.
7. Approaches for efficient game tree search:
 - a. Reduce b:
 - i. Not all the nodes have to be scanned.
 - ii. Specifically, suppose P1 can guarantee value $v_1=3$ by action $a_1=1$ (see example above); and suppose that $a=2$ leads to a branch where P2 can

force value of 2. Then we don't have to scan further children of a_2 , since necessarily $v_2 \leq 2$, thus P1 won't choose a_2 .

- iii. No loss of correctness – we just skip unnecessary checks.
- iv. It is claimed that by efficient implementation we can have $\text{sqrt}(b)$ instead of b , i.e. $T = O\left(b^{\frac{m}{2}}\right)$.

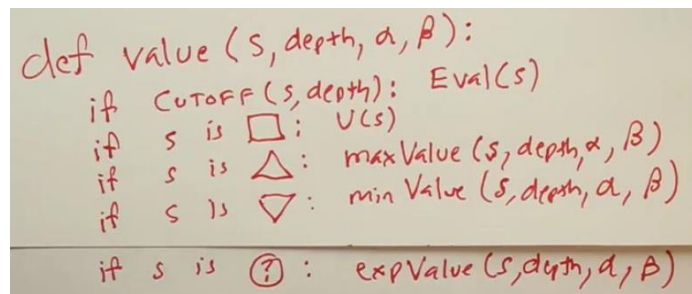
b. Reduce m:

- i. We can just cut the tree in a certain depth $\tilde{m} < m$.
- ii. The backwards-induced values are replaced by heuristic values \rightarrow only approximated solution.
- iii. The heuristic values can be assigned by either expert analysis or supervised learning (e.g. evaluating the states by “empirically, how probable is it to win from this situation?”).
- iv. **The horizon effect:** assume that we look m steps forward in the tree, and update our search & decision every step. After moving from s_1 to s_2 , we get new info and may “regret” and go back to s_1 (if the game allows that). In this situation, we will have an infinite loop due to the limit of the horizon.
- v. The evaluation of chance nodes is very sensitive to the heuristic. For example, when choosing between [50-50 lottery of 0 and 3] and [guaranteed value of 2], by expectation the last is preferred. However, by taking squares of the heuristic values, the first becomes better.

c. Graph representation:

- i. Not very clear. Clearly, graph may represent the problem more compactly since there are no repetitions and no memory (we don't care how we got to a state).
- ii. They say that we can build graphs for decisions in the common states of the beginning and the ending of a game, possibly based on empirical outcomes. For less common states in the middle of the game, some unclear approach (“killer move”) can be used. Yet the point of graph vs. tree is unclear.

8. Summary: the general approach for turn-taking games analysis is as follows:



```

def value(s, depth,  $\alpha$ ,  $\beta$ ):
  if CUTOFF(s, depth): Eval(s)
  if s is  $\square$ : V(s)
  if s is  $\triangle$ : maxVal(s, depth,  $\alpha$ ,  $\beta$ )
  if s is  $\nabla$ : minVal(s, depth,  $\alpha$ ,  $\beta$ )
  if s is  $\odot$ : expVal(s, depth,  $\alpha$ ,  $\beta$ )
  
```

The game-dependent analysis deals with determining the cutoff and evaluating the values heuristically.

Game Theory

1. Concepts of "good" strategies:
 - a. **Dominant** strategy (of a certain player): better than all other strategies, independently on all others' decisions.
 - b. **Equilibrium** (one strategy of each player): no one can benefit from changing only his strategy.
 - c. **Pareto Optimal** (one strategy of each player): there isn't other possible outcome which would be better for everyone.
2. Note that often a unique equilibrium is not Pareto Optimal, i.e. rational egocentric behavior yields bad results for everyone, whereas cooperation could improve the outcome for all the players.
3. **Mixed strategies:**
 - a. The rational behavior might be random.
 - b. The randomization can be exposed to the rivals, but the actual decision must be kept as secret.
4. Mixed strategies equilibrium problem can be solved algebraically and represented geometrically (minmax problem, simplexes, etc.) as taught in Game Theory class.
5. Being seen as irrational can be beneficial – e.g. by making threats more realistic.
6. Scalability: since the games tend to be very complicated, often we merge sets of states to have a simplified, approximated description of the game, which can be analyzed computationally (e.g. replace exact card value by "lower or higher than 10").
7. Game Theory pros & cons:
 - a. Deals with: Uncertainty, Partial Observability, Multi Agents, Stochastic outcomes.
 - b. Out of scope: unknown actions, continuous actions, irrational opponents, unknown utilities.
8. **Mechanism design**: determine the rules of a game to be beneficial for the players or the designer. Some considerations are external, such as making the game simpler by existence of dominant strategies for the participants.
9. **Second-price auction**:
 - a. Winner is the highest bid; price is the second highest bid.
 - b. The dominant strategy is to offer the value of the product $x_1 = v$:
 - i. Assume $x_1 > v$:
 1. If $v < x_1 < x_2$ then we win the bid but pay more than v and get **negative value** instead of 0 value.
 2. If $v < x_2 < x_1$ then we lose the bid and gets 0 value **anyway**.
 3. If $x_2 < v < x_1$ then we win and pay x_2 **anyway**.
 - ii. Assume $x_1 < v$:
 1. If $x_1 < v < x_2$ then we lose the game **anyway**.
 2. If $x_1 < x_2 < v$ then we **lose the game although we could win it**.
 3. If $x_2 < x_1 < v$ then we pay x_2 **anyway**.

- c. This is a ***Truth Revealing mechanism*** – the dominant strategy is to offer the true value!

Advanced Planning

Time

1. Problem definition:
 - a. Goal: complete a list of tasks in shortest time
 - b. Restrictions: every task requires previous tasks to be completed
2. Given that the start state starts at time 0, the “*time values*” of states can be defined recursively:
 - a. **Earliest Start Time (ES)**: earliest possible time to reach the state = the time needed to complete all previous tasks.
 - b. **Latest Start Time (LS)**: latest possible start time that allows completing all the tasks in the shortest time.

Resources

1. Resources can be represented in planning as variables. However, if we have n units of some resource (e.g. 10 apples), it is inefficient to represent them by n variables – it wastes both search time and memory. Thus we wish to represent the quantity of the resource directly.
2. To apply that, every action gets CONSUME property in addition to the CONDITION and EFFECT properties.

Hierarchical Planning

1. **Hierarchical Task Network (HTN)**: every step in the plan has sub-steps...
2. **Refinement planning**: add abstract actions in variant levels. Each such abstract action, named *refinement*, implements an action of some level using actions of a lower level. A refinement is defined by pre-condition and steps of low-actions. A high level task may have multiple refinements achieving variant outcomes. This abstraction allows effective planning.
3. Theorem: a HTN achieves a goal iff for every part (i.e. every abstract action), there exists a refinement that achieves the goal.
4. **Reachable states** for planning in HTN:
 - a. Assume that we know the possible outcomes of a high-level action.
 - b. We can apply the action and keep the possible outcomes (as in stochastic actions, but this time the uncertainty is derived from lack of decision).
 - c. Eventually, we can look for the intersection between the states we’ve reached and the goal states. If such intersection exists, we can choose the refinements of the actions accordingly in order to reach it.
5. When the reachable states of an abstract action are unknown, we can approximate them instead (e.g. use bounds on the possible outcomes of the actions – certainly possible outcomes as lower bound and maybe-possible outcomes as upper bound).
 - a. Goal states intersect lower bound = goal can be guaranteed
 - b. Goal states intersect upper bound = goal may be guaranteed
 - c. Goal states don’t intersect upper bound = goal can’t be guaranteed

Perception

1. Just add *percept* actions that are intended to sense the environment...

Computer Vision

Background

1. **Computer vision:** “making sense out of images or video”.
2. **Pin-hole camera:** simple camera with no lens – just letting the photons pass through a pin-hole. The image of an object is given at size $x = X \frac{f}{z}$, as we saw in class in 2004 (using equal triangles).
 - a. → parallel lines become non-parallel in image. As the lines get far from the camera, they converge into a common **vanishing point**.
3. Lens:
 - a. Pinhole camera is limited by the power of light passing through the pinhole, which must be very small in order to allow focused images. The small hole both reduces the amount of light and may even cause light diffraction (bending over the edges).
 - b. A lens focuses different light rays, hence it allows enlarged hole and much more light getting to the detectors.
 - c. **Lens equation:** $\frac{1}{f} = \frac{1}{z} - \frac{1}{z}$
4. Computer vision tasks:
 - a. **Classify objects**
 - b. **3D reconstructions**
 - c. **Motion analysis**

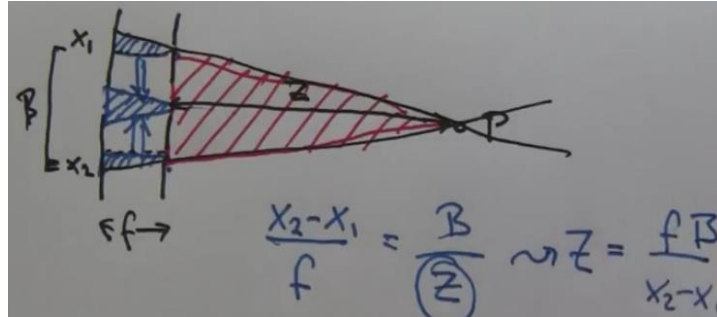
2D Image Analysis

1. Objects recognition: recognition is essential for all the tasks mentioned above. A recognition algorithm is required to be **invariant** to:
 - a. Scale
 - b. Illumination
 - c. Rotation
 - d. Deformation
 - e. Occlusion
 - f. View point (a difficult one – the object may change a lot!)
2. Extracting **Features** = telling things about the image.
3. **Linear Filter** = convolution = sum (or diff) of pixels, defined by a **kernel**, i.e. a **mask**.
 - a. It can be used, for example, to identify edges of a certain direction: $I_x = I \otimes [-1, 1]$ is a vertical lines detector (such lines will have large values after the convolution).
 - b. **Gradient Image** – $E = \sqrt{I_x^2 + I_y^2}$ – can be used to identify edges generally! (note that the filter is not linear anymore)
 - c. **Canny edge detector** – a more advanced filter to detect edges. It mainly applies gradient filter, then removes pixels which are not the highest in their environment – to get thinner lines.
 - d. Other kernels:

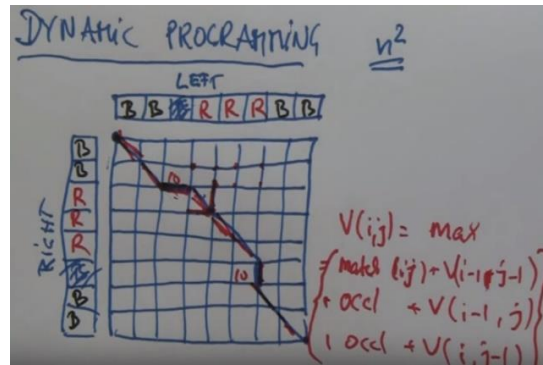
- i. *Sobel, Prewitt*: symmetric numerical approximations for “derivation” of the image – $I \otimes \begin{bmatrix} 1 & 0 & -1 \\ d & 0 & -d \\ 1 & 0 & -1 \end{bmatrix}$ – for better edges.
- ii. *Kirsh* kernel: something unexplained and unclear.
- iii. The lecturer urges us to create our own kernels.
- e. **Gaussian kernel** blurs the image, which has 2 uses:
 - i. Averaging before **downsampling** (to avoid aliasing).
 - ii. **Denoising** by smoothing.
- f. Two masks can be merged in advance to $I \otimes f \otimes g = I \otimes (f \otimes g)$ (since convolution is associative).
- g. **Harris Corner Detector**:
 - i. Corner is a useful feature since it is **local** (opposed to edge).
 - ii. In an image of Cartesian features (either horizontal or vertical), the area of a corner will be characterized by lots of large gradients – both horizontal and vertical (in opposed to edge which would have large gradients only in one direction). Hence a corner is detected by $\sum I_x^2 \gg 1 \wedge \sum I_y^2 \gg 1$.
 - iii. In the general (non-Cartesian) case, the coordinates are conceptually aligned using eigenvectors. Thus, **a corner is detected by 2 large eigenvalues of the matrix** $\begin{bmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix}$.
- h. Modern feature detectors:
 - i. Usually extension of Harris corner detector.
 - ii. Localizable (local features such as corners).
 - iii. Unique signatures – identify the feature with invariance to lighting, orientation, etc.
 - iv. Popular feature extractors:
 - 1. Histogram of Oriented Gradients (**HOG**)
 - 2. Scale Invariant Feature Transform (**SIFT**)

3D Vision – Deriving the Depth from Stereo

1. A main part of the gap between a 2D image and the 3D object it represents, is the “depth” of the image.
2. Given 1 camera, the depth can only be deduced if the size of at least one object is known, using proportions wrt the focus length.
3. **Stereo**: Given 2 cameras, the depth can be deduced from the shift between the images:



- a. (unless the image content is invariant in the direction of the line between the cameras)
4. **Correspondence** (data association): necessary for measuring the shifts between the images.
 - a. All shifts are 1D, hence if we see an object in the first camera, we can search it along a known 1D line in the second one.
 - b. Note that wrong correspondence yields wrong depth estimation, hence reducing the search domain is very beneficial.
 - c. Correspondence can be based on either **patches matching** (by SSD – Sum of Square Differences – minimization) or **features matching** (see features analysis above).
 - d. Assigning the estimated depth of every pixel can yield a **Disparity map**, in which high values represent close distance.
5. Correspondence – context and alignment:
 - a. When the detectors are far from each other, the occlusions derived from the depth may be different, thus the two images may be different – certain pixels will be missing in each image.
 - b. To align the images and cancel the effect of the missing pixels, the algorithm has to check the possibility of dropping pixels. **A cost is defined for miss-matched pixels and for dropped pixels**, and the algorithm minimizes the total cost to align and match the images under possible occlusions.
6. Alignment using dynamic programming:
 - a. Since alignment is based on dropping certain pixels, the possible combinations of dropping are exponential in the size of the image.
 - b. “Dynamic programming” aligns the images in $O(n_{\text{pixels}}^2)$.
 - c. The problem is defined as finding the best path in a corresponding map, where diagonal move represents pixels comparison, horizontal move represents occlusion in one image, and vertical move represents occlusion in the other image (costs of moves are defined accordingly):



- d. The best path is computed using MDP (see corresponding section). The value iteration method can be applied in this case from the origin and forwards, defining the cumulative cost of each node (rather than backwards from the destination as demonstrated before).
- e. Note that due to the structure of the abstract map, and since the “motion” within it is deterministic, only one iteration is needed to assign the values to the nodes. The generalization to 2D image patches may not be trivial, but in the 1D case demonstrated above, it is clear that we get $O(n^2)$.
7. Correspondence – additional challenges:
 - a. An object may have different locations wrt the background, if it is far from the background.
 - b. Rounded occluding objects have different occlusion edges from different POVs.
 - c. Reflections of light appear in different locations, depending on the POV.
8. Stereo vision can be improved by creating clear features through **structured light** illumination – e.g. by stripes illumination, or by laser texture. This is relevant mainly for mapping small objects in controlled environment.
9. A modern alternative approach to Stereo is just using **Laser Radar**.

Structure from Motion

10. Dynamic camera allows mapping vast areas using one camera, based on stereo of sequential images. This is called **Structure from Motion**.
11. The challenge is that the location of the camera may be unknown or inaccurate, thus the distance between the cameras poses is unknown; and the camera takes different images in different orientations (rather than only from different locations on the same plane as before).
12. To use stereo approach and compute the depth, we have to locate the camera poses and the objects simultaneously. This requires optimizing images correspondence wrt the locations, orientations and translations simultaneously, which is a complex non-linear optimization problem (can be solved for example by GD).
13. The problem has $6n_{poses} + 3n_{objs}$ unknowns (camera locations and translations, and objects locations); and $2n_{poses}n_{objs}$ constraints (x,y coordinates for every object in every image). Thus $6n_{poses} + 3n_{objs} \leq 2n_{poses}n_{objs}$ is required to solve the problem.
14. Note that problem is invariant to global shifts and rotations, as well as to scaling (i.e. stretching the whole system). Thus, there will always remain $3+3+1=7$ degrees of freedom in the reconstruction.

Robotics

1. Robotics: An agent senses the environment through sensors and performs actions.
2. The guy started his robotics career by leading his class to win the DARPA challenge of crossing a desert using an autonomous vehicle.
3. In the urban challenge, the vehicle managed to localize itself within a map and detect other cars and obstacles using particle filters and histogram filters.

Perception

4. **Perception** is the part of sensing and understanding the situation. Here we'll mainly talk about finding and predicting the location.
5. **Kinematic state** dimensionality of a robot (1st order approximation):
 - a. Car on a plane: x, y , rotation in plane \rightarrow 3 dimensions
 - b. Free-flying helicopter in the air: x, y, z , 3 Euler angles \rightarrow 6 dimensions
6. **Dynamic state** dimensionality:
 - a. Car on a plane: x, y , rotation in plane, v_{forward} , yaw velocity (=turn speed); no speed to the side \rightarrow 5 dimensions
 - b. Free-flying helicopter in the air: x, y, z , 3 Euler angles + differentiation of every such variable \rightarrow 12 dimensions
7. GPS can give the location of a car with ~5m error. In the guy's car, additional particle filter improved the localization to ~10cm error.
8. **Monte Carlo Localization:**
 - a. Basic car model: 2 connected wheels that can move in the same speed (to go forward) or in different speeds (to turn). Kinematic state is 3D and dynamic state is 5D, as explained above.
 - b. In the deterministic case, the location can be predicted every $\Delta t \approx 0.1s$ by:

$$x' = x + v \cdot \Delta t \cdot \cos\theta$$

$$y' = y + v \cdot \Delta t \cdot \sin\theta$$

$$\theta' = \theta + \omega \cdot \Delta t$$

- c. This can be directly applied to particle filter localization, given appropriate map and sensors. Reminder: particle filter performs iteratively: measure environment, weight particles, sample and predict.
- d. New particles are created by the fact that the actions are stochastic, hence small errors are added to the prediction step.

Planning

9. Planning is about choosing the next actions given a certain situation.
10. A driving path can be planned by the tools described in previous lessons... the ones demonstrated are partitioning the space into discrete states, and find the best path using value-iteration in MDP, or A* (which is useful when we have to use heuristics for unmapped roads, e.g. for passing obstacles from aside).

11. Continuous world vs. discrete state-space:

- a. The gap between those 2 is a significant problem. For example, a car can't take immediate turns (i.e. infinitely sharp turns), hence the path found by A* in discrete grid cells space cannot be applied to the actual car.
- b. **Hybrid A*** memorizes the exact kinematic state (x,y,theta) within a cell. This one is determined by the location prediction ODEs (which must have better numerical resolution than the grid cells). The transitions between cells are no longer straight forward, and smooth turns can be achieved by micro-turning within a cell. This is actually a kind of hierarchical planning – grid cell for path search, and exact location for driving implementation.

Natural Language Processing

1. Interests:
 - a. **Philosophical**: natural language is the way we think, and what we believe that separates us from other animals.
 - b. **Communication**: controlling machines using natural language is intuitive and convenient.
 - c. **Learning**: understanding natural language allows machines to learn huge amounts of available data.
2. Two popular language models:
 - a. Probabilistic, word-based, learned: based on the empirical probability of appearance of sequences of words $P(w_1, w_2, \dots)$.
 - b. Logical, trees/categories-based, hand-coded: based on abstract structures representing valid syntax (noun \rightarrow verb etc.).
 - c. The separation is traditional, but there can be probabilistic and structural models of the language.
3. **Naïve Bayes** model: every words is assumed to be independent on the context (**unigram** model – every word stands alone), which yields the naïve probabilistic model $P(\{w_i\}) = \prod_i p(w_i)$, that deals with "bags of words" rather than sequences.
4. A general probabilistic model will satisfies $P(\{w_i\}) = \prod_i p(w_i|w_1, \dots, w_{i-1})$, assuming that the probability of a word depends on the previous words in the sentence.
 - a. **Markov assumption** localizes the problem, assuming dependence on the last k words only: $P(\{w_i\}) = \prod_i p(w_i|w_{i-k}, \dots, w_{i-1})$.
 - i. $k = 0$ gives the naïve Bayes model.
 - ii. For $k = 1$ it is called **bigram** model, since the words kind of come in pairs. $k = 2$ gives **trigram** model, and in general we have **N-gram models** ($N = k + 1$).
 - iii. Higher k yields more complex and accurate model of the language.
 - b. **Stationarity assumption**: if I keep talking, it doesn't matter whether it's the 10th or the 20th sentence/word, thus $p(w_i|w_{i-k}, \dots, w_{i-1}) = p(w_j|w_{j-k}, \dots, w_{j-1})$. This assumption is usually used for simplicity even when there are only few sentences or words in each sentence.
 - c. **Smoothing**: as demonstrated before, to reduce the sensitivity to the empirical data available for learning, the learned distribution can be smoothed (e.g. by averaging it with uniform distribution, as in **Laplace smoothing**).
5. Word-based models applications:

- Classification (e.g. spam)
- Clustering (e.g. news stories)
- Input correction (spelling, segmentation)
- Sentiment analysis (e.g. product reviews)
- Information retrieval (e.g. web search)
- Question answering (e.g. IBM's Watson)
- Machine translation (e.g. Chinese to English)
- Speech recognition (e.g. Apple's Siri)

6. Variations of the probabilistic model may include the syntactic role of a word (noun, verb, etc.) or use basic units of letters or phrases (e.g. "New-York City") rather than words.
7. Letter-based model is effective for learning new words, detection of valid words, and classification of phrases to different languages/classes. Note that storing all probabilities of triplets of letters requires $\sim 30^3 \approx 3 \cdot 10^4$ triplets rather than $\sim 10^{6^3} = 10^{18}$ triplets of words.
 - a. In practical **language identification**, one usually uses both letter-based model and several discriminative words – which are common in one language and invalid in the others.
 - b. Note that for classification, one can use both word-based and letter-based features, and examine their efficacies using machine-learning technics.
 - c. In addition to machine learning, a nice trick for classification of phrases is based on compression technics (e.g. Hoffman compression): a typical text will be more efficiently compressed when it contains homogeneous language. Thus, compressed sentence will be smaller after being concatenated to text of the same language rather than to other language. This can be implemented as follows:

EN	DE	AZ
Hello world! This is a file full of English words ...	Hallo Welt! Dies ist eine Datei voll von deutschen Worte ...	Salam Dünya! Bu fayl Azərbaycan tam sözlər ...

NEW
This is a new piece of text to be classified.


```
(echo `cat new EN | gzip | wc -c` EN; \
echo `cat new DE | gzip | wc -c` DE; \
echo `cat new AZ | gzip | wc -c` AZ) \
| sort -n | head -1
```

This essentially detects common phrases in the language which appear in the new sentence. It demonstrates the close relations between compression and processing, through information theory and **entropy** of expressions.

Segmentation

8. **Segmentation** is the action of separating text into token representing words.

9. In written English it is trivially done by spaces. That's not the case in spoken English or even written Chinese (seriously? no spaces?) or words within a URL.
10. Goal: find $S^* = \operatorname{argmax}_{seg} P(w_1, \dots, w_n)$ over all possible segmentations of a sequence of characters.
11. Simplification – Naïve Bayes model:
 - a. The joint distribution is simpler to calculate: $P = \prod P(w_i)$.
 - b. Given n characters, there are 2^{n-1} possible segmentations. Independence between words allows us to avoid examining every single segmentation.
12. The naive segmentation works well in most cases, and can be implemented recursively as follows (some smoothing should be added inside $Pwords()$):

```
def splits(characters, longest=12):
    "All ways to split characters into a first word and remainder."
    return [(characters[:i], characters[i:])]
            for i in range(1, 1+min(longest, len(characters)))

def Pwords(words):
    "Probability of a sequence of words."
    return product(words, key=Pw)

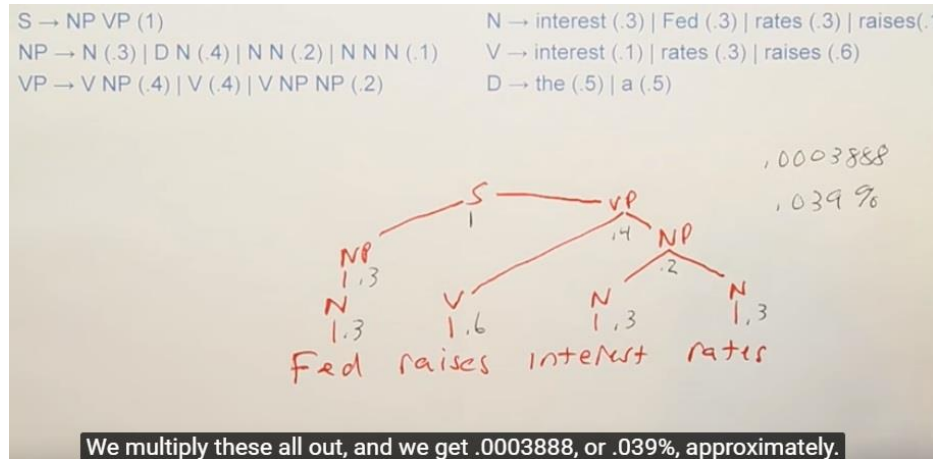
@memo
def segment(text):
    "Best segmentation of text into words, by probability."
    if text == "": return [ ]
    candidates = [[first]+segment(rest) for first,rest in splits(text)]
    return max(candidates, key=Pwords)
```

13. Problems with the simplified segmentation algorithm:
 - a. Ambiguity: "base rate sought to" vs. "base rates ought to".
 - b. Naivety: $P("in") * P("significant") > P("insignificant")$, although $P("in significant") \ll P("insignificant")$, so the naïve model is wrong here.
 - c. Smoothing: "g in or mouse go" was chosen rather than "ginormous ego". This can be prevented by larger database, but also by smarter smoothing, that allows the word 'ginormous' without observing it before. Such smart smoothing may use, for example, letter model that identifies the common ending 'ous'.

Spelling

14. **Spelling** is finding the correct version of a word among possible corrections.
15. Goal: given a word w , find $c^* = \operatorname{argmax}_c P(c|w) \sim \operatorname{argmax}_c P(w|c)P(c)$ (up to a constant denominator).
16. The prior probability $P(c)$ of the correct word can be determined as before, possibly naively from text database.
17. The probability $P(w|c)$ of a certain spelling mistake in a word is more difficult to learn due to lack of data.

4. Writing CFG turns out to be extremely difficult. A natural language L is typically very complex, and a simple grammar language L_G usually either doesn't cover the whole language or allows too many invalid sentences. One can give up on some of the valid sentences and then adding them to the grammar using exceptions, but this is really complex.
5. **Probabilistic Context Free Grammar (PCFG):**
 - a. Just associate a probability with each substruct decomposition (e.g. $P(NP \rightarrow D, N)$ wrt $P(NP \rightarrow N)$), to achieve probability estimation of the complete sentence.
 - b. Rules & implementation example:



- c. The probabilities should be learnt from real data of sentences trees. Such trees data is not naturally available (opposed to word counting which is straight forward from available texts). However, in the 90s, due to the importance of the field, some organizations created such databases manually. One of them is **The Penn Tree Bank** of Pennsylvania University.
 - d. PCFG advantages:
 - i. Reduce sensitivity to the language derived from the logical grammar – invalid sentences might be created, but they would be associated with low probabilities.
 - ii. Dealing with ambiguity (which interpretation has larger probability?).
6. **Lexicalized Probabilistic Context Free Grammar (LPCFG):**
 - a. PCFG defines probabilities for both syntactic structures (e.g. $NP \rightarrow N, N$) and single words (e.g. $N \rightarrow \text{watermelon}$).
 - b. We would like to define probabilities for relations between words. For example, in “I saw the man with a telescope”, we wish to solve the ambiguity and decide whether “with a telescope” refers to the man or to seeing. This cannot be estimated by structural analysis, but must examine the relations between the words themselves, and associate them with probabilities.
 - c. Note that structural analysis IS needed to understand which words relate to each other.
 - d. Specifically in the example, we wish to compare the following:
 - i. $P(NP \rightarrow NP, PP \mid NP = \text{man}, PP = \text{with telescope})$

- ii. $P(VP \rightarrow V, NP, PP \mid V=\text{saw}, NP=\text{man}, PP=\text{with telescope})$
 - e. Since the exact phrases “saw”, “the man” and “with a telescope” are very specific, it is difficult to learn their relationships from data. Thus, it is essential to use **back-off models** that generalize those terms (e.g. “the man” may belong to “persons” category).
 - f. All of that is done by Lexicalized Grammar.
7. **Parsing** – sentence \rightarrow tree:
- a. Search approach can be applied – either bottom-up or top-down.
 - b. Bottom-up: for every word note the possible interpretations (N,V,D), and then try to connect them into phrases, until a complete sentence is achieved.
 - c. Top-down: start from the complete sentence node, and try to decompose it into sub-phrases according to the grammar rules, until the decomposition fits the sentence.
 - d. The hierarchical structure of the tree allows recursive parsing, in which every phrase can be decomposed independently on the other phrases.

Machine Translation

1. Possible approaches:
 - a. Translate words.
 - b. Translate phrases – few words together.
 - c. Translate syntax tree – given structure of sentence in L1, what is the probable structure of the translated sentence in L2?
 - d. Translate semantics and meaning – understand the general meaning of the sentence, e.g. “person does something”, and keep that meaning in the translation.
2. The hierarchy of the approaches above is known as **Valcroix’s Pyramid**. The higher approaches in the hierarchy must rely on the lower ones in order to complete the translation task and return a concrete sentence.
3. A corresponding probabilistic model may take into account the following probabilities:
 - a. Segmentation – picking words together into phrases.
 - b. Translation – of L1 phrases to L2 phrases.
 - c. Distortion/syntax – how phrases tend to move within the sentence in such translations, or how the syntax tends to change between the languages.
 - d. Result – is the translated sentence valid?

Summary: Language Models

Model	Probabilistic	Structures Complexity	Semantics	Atomic Unit
Naïve Bayes	V	“Bags of words”	X	word/letter
Markovian		Sequences	partially	
CFG	X	Syntax	X	word
PCFG	V		V	
LPCFG				

Q&A

1. Only watched sporadically (about 2 hours of Q&A aggregately!).
2. Suggested software:
 - a. Mahout – machine learning toolkit, useful for both applications and independent building of ML algorithms.
 - b. MATLAB – strong at quick processing and visualization.
3. Out of scope: Genetic algorithms, Neural Networks, Fuzzy algorithms, Fuzzy logic.
4. They said something about connecting excellent students to some good employee, but I think it's only for the paid version of the course. Not sure.