# TDT4171 - Artificial Intelligence Methods

Martin Bjerke

June 1, 2018

# Contents



In the following summary of the syllabus of TDT4171 in 2018, text that has an additional margin is only mentioned in the syllabus, not in the lectures. The sections and equations have the same numbering as in the book.

# 1 Book

## 13 Quantifying Uncertainty

Agent encounters uncertainty through partial obersvability, non-determinism or both. To counter this problem-solving agents and logic agents tracks the world sate and stores it as a *belief state*. The *qualification problem* arises when an agent tries to store all possibilities(even implausible ones) in its belief stat, making it impossible to infer its next action. Therefore *rational decision* takes into consideration both the relative importance of an action and its likelihood.

Using inference to make an action fails on the following ground:

*Laziness* failure to enumerate exceptions, qualifications etc

*Theoretical ignorance* no theory for the domain

*Practical ignorance* not possible to run all tests

Probabilistic assertions can summarise these effects, and an agent can store its *degrees of belief* that something will happen and use *probability theory* to chose the best action. The combination of probability theory and *utility theory*(inference) is called *decision theory* and all of them can be used to choose an action.

Probability gives an assertion about how the world looks(there is a 50% chance of rain today versus it is raining today), and the set of all possible worlds(denoted $\Omega$) is called the sample space. A fully specified probability model gives a probability, $P(\omega)$, of each possible world, and by the axioms of probability each world has a probability between 0 and 1 and together the probability must sum to 1:

$$0 \leq P(\omega) \leq 1 \text{ for every } \omega \text{ and } \sum_{\omega \in \Omega} P(\omega) = 1 \tag{13.1}$$

The probability of *propositions*(also called probabilistic assertions or queries, or events) are the sum of the probability of the worlds where the proposition holds:

$$\text{For any proposition } \phi,\ P(\phi) = \sum_{\omega \in \phi} P(\omega) \tag{13.2}$$

There are different "types" of probability:

**Prior/unconditional probabilities:** Independent probabilities

**Posterior/conditional probabilities:** Dependent probabilities, the probability of rain, given that its already cloudy. Denoted $P(a \mid b)$.

**Probability distributions:** Outputs a probability given a value, most common in settings where the sample space is continuous, like temperature.

**Joint probability distributions:** Normalised probability distributions

As more knowledge is gained, conditions arises, like you might not know you have a cavity, but when you get toothache, you might consider it, $P(Cavity|Toothache)$. Note, this means you ONLY know that you have a toothache, not anything more relevant information. The conditional probability can be calculated using unconditional probabilities:

$$P(a \mid b) = \frac{P(a \wedge b)}{P(b)} \text{ or usually presented as the } \textit{product rule} P(a \wedge b) = P(a \mid b)P(b) \tag{13.3}$$

Some notation, *random variables* are denoted with uppercase names and each variable has a *domain* of possible values, like $Die = \{1, 2, 3, 4, 5, 6\}$, boolean variables has two values $\{true, false\}$ and values with names are denoted with lowercase names. When a bold $\boldsymbol{P}$ is used it the probability of all the possible values of the variable, $\boldsymbol{P}(Weather) = \langle 0.6, 0.1, 0.29, 0.01 \rangle$[1] and this is also called a *probability distribution*. With discrete variables the probability distribution can be represented as a *joint probability distribution* as shown below. Going back to the possible worlds talk about above, if each random variable of a world is given a specific value this represents one specific world. If the joint probability distribution contains all possible random variables, then it is *full joint probability distribution*(FJPD).

| | toothache | | ¬ toothache | |
|---|---|---|---|---|
| | catch | ¬catch | catch | ¬catch |
| cavity | .108 | .012 | .072 | .008 |
| ¬cavity | .016 | .064 | .144 | .576 |

Using this table each possible value of each variable can be calculated using probabilistic inference: ex. $P(toothache) = .108 + .012 + .016 + 0.64 = .2$ and $P(cavity \vee toothache) = .108 + .012 + .016 + .64 + .072 + .008 = .28$.

Together with the product rule (Eq. **??**) and the *inclusion-exclusion principle* (Eq. 13.4) we get the *Kolmogorov's axioms*. These build up the rest of probability theory.

$$P(a \vee b) = P(a) + P(b) - P(a \wedge b) \tag{13.4}$$

Summing up all the entries of a FJPD of a specific value, i.e $P(cavity) = .108 + .012 + .072 + .008 = 0.2$, is called *marginalisation* and can be represented as the following general rule, with the variables $\boldsymbol{Y}$ and $\boldsymbol{Z}$:

$$\boldsymbol{P}(Y) = \sum_{z \in \boldsymbol{Z}} \boldsymbol{P}(Y, z) \tag{13.5}$$

Above this is partially done with $Cavity$, $\boldsymbol{P}(Cavity) = \sum_{z \in \{Catch, Toothache\}} \boldsymbol{P}(Cavity, z)$. From Eq. **??** we get the conditioning rule:

$$\boldsymbol{P}(\boldsymbol{Y}) = \sum_{z} \boldsymbol{P}(\boldsymbol{Y} \mid z)P(z) \tag{13.8}$$

Going back to Eq. 13.3 and if you calculate both $P(cavity \mid toothache) = 0.6$ and $P(\neg cavity \mid thoothache) = 0.4$ these add up to 1, as they should. Also notice that the $\frac{1}{P(toothache)}$ is constant, and therefore we view it as a *normalisation* constant of the distribution of $\boldsymbol{P}(Cavity \mid toothache)$ and is therefore swapped with a normalisation variable $\alpha$, which is set so that a distribution $\boldsymbol{P}(A)$

[1]Meaning here that $P(Weather = sunny) = 0.6$, $P(Weather = rain) = 0.1$, $P(Weather = cloudy) = 0.29$, and $P(Weather = snow) = 0.01$

has a total probability of 1. This is done since $\frac{1}{P(toothache)}$ is just a normalisation, so we don't actually need to know it. From this we extract a general inference procedure:

Let $X$ be all the query variables, $E$ be the evidence variables and $e$ be the specific values and let $Y$ be the remaining hidden variables. This gives the following formula for summing the joint entries:

$$\boldsymbol{P}(X \mid e) = \alpha \boldsymbol{P}(X, e) = \alpha \sum_{y} \boldsymbol{P}(X, e, y) \tag{13.9}$$

In mote cases, the use of conditional independence reduces the size of the representation of the joint distribution from exponential in n to linear. For example n coin flips, if they where dependent there would be $2^n$ representations in the joint distribution, but since they are independent, only $n$ is needed.

Identifying independence, two variables are independent iff:
$P(X \mid Y) = P(X)$ equivalently $P(Y \mid X) = P(Y)$

Bayes rule:

$$\begin{aligned}
\boldsymbol{P}(Y \mid X) &= \frac{\boldsymbol{P}(X \mid Y)\boldsymbol{P}(Y)}{\boldsymbol{P}(X)} \\
&\text{or with some background evidence } e \\
\boldsymbol{P}(Y \mid X, e) &= \frac{\boldsymbol{P}(X \mid Y, e)\boldsymbol{P}(Y, e)}{\boldsymbol{P}(X, e)} \\
&\text{or by using the normalisation trick from before} \\
\boldsymbol{P}(X \mid e) &= \alpha \langle P(e \mid x)P(x), P(e \mid \neg x)P(\neg x) \rangle \\
&= \boldsymbol{P}(X \mid Y) = \alpha \boldsymbol{P}(Y \mid X)\boldsymbol{P}(X)
\end{aligned} \tag{13.13}$$

Variables are said to be conditionally independent if they have the same *cause*, but different *effects*, where the effects are independent, but they are both dependent on the cause. I.e. Toothache and catch are both an effect of having a cavity, but having a toothache is independent of the toothpick to catch and vise versa; $\boldsymbol{P}(toothache \wedge catch \mid Cavity) = \boldsymbol{P}(toothache \mid Cavity)\boldsymbol{P}(catch \mid Cavity)$. Conditional independence is represented in the following equation:

$$\begin{aligned}
&\boldsymbol{P}(X, Y \mid Z) = \boldsymbol{P}(X \mid Z)\boldsymbol{P}(Y \mid Z) \\
&\text{since Y and X are independent, the following is also true} \\
&\boldsymbol{P}(X \mid Y, Z) = \boldsymbol{P}(X \mid Z) \\
&\boldsymbol{P}(Y \mid X, Z) = \boldsymbol{P}(Y \mid Z)
\end{aligned} \tag{13.19}$$

Using conditional independence to "shorten" a probability distribution is called a *naive Bayes model*, this is because it used in cases where the effect variables are not actually conditionally independent given the cause variable.

# 14 Bayesian networks

Bayesian networks are graphical notations for conditional independence assertions, giving a compact specification of full joint distributions. The syntax is a set of nodes(one per variable) that makes out a DAG and a conditional distribution for each node given its parents, $\boldsymbol{P}(X_i \mid Parents(X_i))$

All conditional distribution represented as a conditional probability table(CPT) gives the distribution over $X_i$ for each combination of parent values.

Example: A CPT for boolean $X_i$ with k boolean parents has $2^k$ rows for the combinations of parent values. Since $X_i$ is boolean, $P(X_i = true) = 1 - P(X_i = false)$, so each row requires only one representative, so if each variable has maximum k parents, the complete network requires $O(n \cdot 2^k)$ numbers.

### Representing a full joint distribution and constructing a Bayesian network

A generic entry in a joint distribution is just a conjunction of assignment of values to all variables, $P(X_1 = x_1 \wedge X_2 = x_2 \wedge ... \wedge X_n = x_n)$ which is written in shorthand as $P(x_1, x_2, ..., x_n)$. Since $x_i$ is just dependent of its parents this value is given by:

$$P(x_1, x_2, ..., x_n) = \prod_{i=0}^{n} P(x_i \mid parents(X_i)) \tag{14.1}$$

Example: Let B and E be parents of A, and J and M be children of A,

$$P(j \wedge m \wedge a \wedge \neg b \wedge \neg e) = P(j \mid a)P(m \mid a)P(a \mid \neg b, \neg e)P(\neg b)P(\neg e) \tag{14.2}$$

The chain rule can be extracted from this using the product rule:

$$\begin{aligned} P(x_1, ..., x_n) &= P(x_n \mid x_{n-1}, ..., x_1)P(x_{n-1}, ..., x_1) \\ &= P(x_n \mid x_{n-1}, ..., x_1)P(x_{n-1} \mid x_{n-2}, ..., x_1)P(x_{n-2}, ..., x_1) \\ &= ... = \prod_{i=1}^{n} P(x_i \mid x_{i-1}, ..., x_1) \end{aligned} \tag{14.3}$$

Using the chain rule in Eq. 14.1 and generalising:

$$\boldsymbol{P}(X_i \mid X_i, ..., X_1) = \boldsymbol{P}(x_i \mid Parents(X_i)) \tag{14.4}$$

Local semantics: each node is conditionally independent of its non descendants given its parents. Markov blanket: Each node is conditionally independent of all other nodes then its parents, children and children's parents.

Construction of a Bayesian network:

1. Choose an ordering of variables $X_1, ..., X_n$
2. for $i = 1$ to $n$
   (a) add $X_i$ to the network
   (b) select parents from $X_1, ..., X_{i-1}$ such that $P(X_i \mid Parents(X_i)) = P(X_i \mid X_1, ..., X_{i-1})$

Step 1 - Decide what to model

Step 2 - Define variables

Step 3 - Define the graphical structure that connects the variables(The qualitative part)

Step 4 - Fix parameters to specify each $P(x_i \mid pa(x_i))$(The quantitative part)

Step 5 - Verification

There are other approaches, other then probability, to handle uncertainty:

- Qualitative reasoning: I.e. default reasoning: A conclusion is sound as long as there are no better conclusions to draw.
- Rule-based: Add a "fudge factor" to each rule to accommodate uncertainty. There are 3 desirable properties from rule-based:
  - *Locality*; Given the rule $A \implies B$, B can be concluded by only looking at A, not the other evidence.
  - *Detachment*; Once a proof for a proposition is found it can be detached from its justification.
  - *Truth-functionality*; Truth of sentences can be computed by the truth of the components. This can't be done in probability without independence.

  There have been attempts to add uncertainty to RB by assigning a belief to the rules.
- Dempster-Shafer theory: Compute the probability that the evidence supports the proposition(use a *belief function*).
- Fuzzy logic: is fuzzy

# 15 Probabilistic reasoning over time

To handle the coming dynamic models they are viewed discretely using *time slices*, containing sets of observable and unobservable random variables. For notation $\boldsymbol{X}_t$ denotes the set of state variables at time $t$ which are unobservable and $\boldsymbol{E}_t$ denotes the set of observable evidence variables, so the observations at time $t$ are $\boldsymbol{E}_t = e_t$. The notation a:b denotes (inclusive) the interval from a to b, so $\boldsymbol{X}_{a:b}$ are the set of variables $\boldsymbol{X}_a, ..., \boldsymbol{X}_b$. The examples used are based on the umbrella world of Chapter 15. The notations of *transition model* is how the world evolves and *sensor model* is how the evidence variables get their values.

The *transition model* specifies the probability distribution over the latest state variables, given the previous states, $\boldsymbol{P}(\boldsymbol{X}_t \mid \boldsymbol{X}_{0:t-1})$. Using this increases the number of previous states need over time. This is solved by making a *Markov assumption*, the next states only depend on a finite set of previous states. This is called a *Markov process* or *Markov chain*, the simplest form being the *first-order Markov process* where the current state only depends on the previous one. In other words, a future state is only dependent on the previous state and conditionally independent of the rest:

$$\boldsymbol{P}(\boldsymbol{X}_t \mid \boldsymbol{X}_{0:t-1}) = \boldsymbol{P}(\boldsymbol{X}_t \mid \boldsymbol{X}_{t-1}) \tag{15.1}$$

Having bounded the number of transition models using a Markov assumption, we want to bound the possible number of different transition models that arises. We want a *stationary process*, where the rules of changes to the world states does not change. I.e in the umbrella world $\boldsymbol{P}(R_t \mid R_{t-1})$ is the same for all $t$.

Next is bounding the sensor model. This is done by making a *sensor Markov assumption* that works the same way as a Markov assumption, limiting the number of past evidence values needed and giving us the following sensor model(sometimes called the *observation model*):

$$\boldsymbol{P}(\boldsymbol{E}_t \mid \boldsymbol{X}_{0:t}, \boldsymbol{E}_{0:t-1}) = \boldsymbol{P}(\boldsymbol{E}_t \mid \boldsymbol{X}_t) \tag{15.2}$$

Lastly we need a prior probability distribution at time $t = 0, \boldsymbol{P}(\boldsymbol{X}_0)$. Given these assumptions and equations we have the complete joint distribution for any t:

$$\boldsymbol{P}(\boldsymbol{X}_{0:t}, \boldsymbol{E}_{1:t}) = \boldsymbol{P}(\boldsymbol{X}_0) \prod_{i=1}^{t} \boldsymbol{P}(\boldsymbol{X}_i \mid \boldsymbol{X}_{i-1}) \boldsymbol{P}(\boldsymbol{E}_i \mid \boldsymbol{X}_i) \tag{15.3}$$

To use these models, the following interference tasks can be used:

- *Filtering*: Calculate the posterior distribution, aka the belief state, given the evidence observed so up until a time slice $t$; $\boldsymbol{P}(\boldsymbol{X}_t \mid \boldsymbol{e}_{1:t})$. A close related calculation can be done to get $P(\boldsymbol{e}_{1:t})$
- *Prediction*: Calculate the probability of a future belief state at time slice $k$, where $k > 0$, given the present evidence values; $\boldsymbol{P}(\boldsymbol{X}_{t+k} \mid \boldsymbol{e}_{1:t})$.
- *Smoothing*: Calculate the belief state of a previous time slice $k$, given the observed evidence up until $t$, where $0 < k < t$; $\boldsymbol{P}(\boldsymbol{X}_k \mid \boldsymbol{e}_{1:t})$.
- *Most likely explanation*: Calculate the most likely belief states that would give the observed evidence; $argmax_{x_{1:t}} P(x_{1:t} \mid e_{1:t})$

Together with these tasks learning is also possible. Both the transition model and the sensor model can be learned through observation.

### Filtering/forward message

To calculate the belief state at time slice $t + 1$, we can use the following equation;

$$\begin{aligned}
\boldsymbol{P}(\boldsymbol{X}_{t+1} \mid \boldsymbol{e}_{1:t+1}) &= \boldsymbol{P}(\boldsymbol{X}_{t+1} \mid \boldsymbol{e}_{1:t}, \boldsymbol{e}_{t+1}) && \text{(dividing up the evidence)} \\
&= \alpha \boldsymbol{P}(\boldsymbol{e}_{t+1} \mid \boldsymbol{X}_{t+1}, \boldsymbol{e}_{1:t}) \boldsymbol{P}(\boldsymbol{X}_{t+1} \mid \boldsymbol{e}_{1:t}) && \text{(using Bayes rule)} \\
&= \alpha \boldsymbol{P}(\boldsymbol{e}_{t+1} \mid \boldsymbol{X}_{t+1}) \boldsymbol{P}(\boldsymbol{X}_{t+1} \mid \boldsymbol{e}_{1:t}) && \text{(using sensor Markov assumption)}
\end{aligned} \tag{15.4}$$

The terms are as follows; $\alpha$, the normalisation constant; $\boldsymbol{P}(\boldsymbol{e}_{t+1} \mid \boldsymbol{X}_{t+1})$, the sensor model is used to update the belief state; $\boldsymbol{P}(\boldsymbol{X}_{t+1} \mid \boldsymbol{e}_{1:t})$, a prediction of the next belief state. This last prediction is based on the current belief state $\boldsymbol{X}_t$ and this is used integrated in the following way:

$$\begin{aligned}
\boldsymbol{P}(\boldsymbol{X}_{t+1} \mid \boldsymbol{e}_{1:t+1}) &= \alpha \boldsymbol{P}(\boldsymbol{e}_{t+1} \mid \boldsymbol{X}_{t+1}) \sum_{\boldsymbol{x}_t} \boldsymbol{P}(\boldsymbol{X}_{t+1} \mid \boldsymbol{x}_t, \boldsymbol{e}_{1:t}) P(\boldsymbol{x}_t \mid \boldsymbol{e}_{1:t}) \\
&= \alpha \boldsymbol{P}(\boldsymbol{e}_{t+1} \mid \boldsymbol{X}_{t+1}) \sum_{\boldsymbol{x}_t} \boldsymbol{P}(\boldsymbol{X}_{t+1} \mid \boldsymbol{x}_t) P(\boldsymbol{x}_t \mid \boldsymbol{e}_{1:t})
\end{aligned} \quad \begin{aligned} &\text{(predicting } \boldsymbol{X}_{t+1} \text{ using } \boldsymbol{X}_t) \\ &\text{(using Markov assumption)} \end{aligned} \tag{15.5}$$

Notice that $\boldsymbol{P}(\boldsymbol{X}_{t+1} \mid \boldsymbol{x}_t)$ comes from the transition model. Further $\boldsymbol{P}(\boldsymbol{x}_t \mid \boldsymbol{e}_{1:t})$ can be viewed as a *forward message*(as this is calculated at each time step using the previous state) and Eq. 15.5 can be in shorthand as:

$$\boldsymbol{f}_{1:t+1} = \alpha FORWARD(\boldsymbol{f}_{1:t}, \boldsymbol{e}_{1:t+1})$$

where $\boldsymbol{f}_{1:0} = \boldsymbol{P}(\boldsymbol{X}_0)$

### Prediction

This is done by using Eq. 15.5 using just the evidence observed so far, $\boldsymbol{e}_{1:t}$ and the previous predicted state, $\boldsymbol{X}_{t+k}$:

$$\boldsymbol{P}(\boldsymbol{X}_{t+k+1} \mid \boldsymbol{e}_{1:t}) = \sum_{\boldsymbol{x}_{t+k}} \boldsymbol{P}(\boldsymbol{X}_{t+k+1} \mid \boldsymbol{x}_{t+k}) P(\boldsymbol{x}_{t+k} \mid \boldsymbol{e}_{1:t}) \tag{15.6}$$

These predictions will converge to a fixed sate called the *stationary distribution* after a number of time steps called the *mixing time*, dooming any attempt to predicting to far into the future.

### Likelihood

This is useful when reviewing the assumed belief states calculated given the evidence and is done using the *likelihood message*, $\boldsymbol{l}_{1:t}(\boldsymbol{X}_t) = \boldsymbol{P}(\boldsymbol{X}_t, \boldsymbol{e}_{1:t})$, this in turn can be inferred into the following:

$$\begin{gathered} \boldsymbol{l}_{1:t+1} = FORWARD(\boldsymbol{l}_{1:t}, \boldsymbol{e}_{1:t+1}) \\ \text{and from } \boldsymbol{l}_{1:t} \text{ the actual likelihood can be calculated} \\ L_{1:t} = P(\boldsymbol{e}_{1:t}) = \sum_{\boldsymbol{x}_t} \boldsymbol{l}_{1:t}(\boldsymbol{x}_t) \end{gathered} \tag{15.7}$$

### Smoothing

Computing the belief state of a previous state, $\boldsymbol{P}(\boldsymbol{X}_k \mid \boldsymbol{e}_{1:t})$ at time $0 \leq k \leq t$, using evidence up to the present, $t$, can be done by splitting the computation into two:

$$\begin{aligned} \boldsymbol{P}(\boldsymbol{X}_k \mid \boldsymbol{e}_{1:t}) &= \boldsymbol{P}(\boldsymbol{X}_k \mid \boldsymbol{e}_{1:k}, \boldsymbol{e}_{k+1:t}) && \text{split the evidence} \\ &= \alpha \boldsymbol{P}(\boldsymbol{X}_k \mid \boldsymbol{e}_{1:k}) \boldsymbol{P}(\boldsymbol{e}_{k+1:t} \mid \boldsymbol{X}_k, \boldsymbol{e}_{1:k}) && \text{using Baye's rule} \\ &= \alpha \boldsymbol{P}(\boldsymbol{X}_k \mid \boldsymbol{e}_{1:k}) \boldsymbol{P}(\boldsymbol{e}_{k+1:t} \mid \boldsymbol{X}_k) && \text{using Eq. 15.2} \\ &= \alpha \boldsymbol{f}_{1:k} \times \boldsymbol{b}_{k+1:t} && \text{element wise multiplication} \end{aligned} \tag{15.8}$$

Where $\boldsymbol{b}_{k+1:t} = \boldsymbol{P}(\boldsymbol{e}_{k+1:t} \mid \boldsymbol{X}_k)$ is the backward message that is the recursive analogy to the forward message that is calculated backwards from $t$:

$$\begin{aligned} \boldsymbol{P}(\boldsymbol{e}_{k+1:t} \mid \boldsymbol{X}_k) &= \sum_{\boldsymbol{x}_k+1} \boldsymbol{P}(\boldsymbol{e}_{k+1:t} \mid \boldsymbol{x}_{k+1}, \boldsymbol{X}_k) \boldsymbol{P}(\boldsymbol{x}_{k+1} \mid \boldsymbol{X}_k) \\ &= \sum_{\boldsymbol{x}_k+1} \boldsymbol{P}(\boldsymbol{e}_{k+1:t} \mid \boldsymbol{x}_{k+1}) \boldsymbol{P}(\boldsymbol{x}_{k+1} \mid \boldsymbol{X}_k) && \text{conditioning on } \boldsymbol{X}_{k+1} \\ &= \sum_{\boldsymbol{x}_k+1} \boldsymbol{P}(\boldsymbol{e}_{k+1}, \boldsymbol{e}_{k+2:t} \mid \boldsymbol{x}_{k+1}) \boldsymbol{P}(\boldsymbol{x}_{k+1} \mid \boldsymbol{X}_k) && \text{conditional independence} \\ &= \sum_{\boldsymbol{x}_k+1} \boldsymbol{P}(\boldsymbol{e}_{k+1} \mid \boldsymbol{x}_{k+1}) \boldsymbol{P}(\boldsymbol{e}_{k+2:t} \mid \boldsymbol{x}_{k+1}) \boldsymbol{P}(\boldsymbol{x}_{k+1} \mid \boldsymbol{X}_k) \end{aligned} \tag{15.9}$$

Notice that the first and third products are from the models, and the second is the recursive call.

### Most likely sequence

Because of the *Markov assumptions* the transition between states are dependent on the previous state, therefore, if you change the question from which sequence is most likely to what path through the states that takes(see Fig 15.5 in book), the most likely state at time $t+1$ is dependent on the most likely state at $t$. This relationship can be expressed as:

$$\begin{aligned}&\max_{\boldsymbol{x}_1 \ldots \boldsymbol{x}_t} \boldsymbol{P}(\boldsymbol{x}_1, ..., \boldsymbol{x}_t, \boldsymbol{X}_{t+1} \mid \boldsymbol{e}_{1:t+1}) = \\ &\alpha \boldsymbol{P}(\boldsymbol{e}_{1:t+1} \mid \boldsymbol{X}_{t+1}) \max_{\boldsymbol{x}_t}(\boldsymbol{P}(\boldsymbol{X}_{t+1} \mid \boldsymbol{x}_t) \max_{\boldsymbol{x}_1 \ldots \boldsymbol{x}_{t-1}} P(\boldsymbol{x}_1, ..., \boldsymbol{x}_t \mid \boldsymbol{e}_{1:t}))\end{aligned} \tag{15.10}$$

The *Hidden Markov Model*(HMM) is a way to represent the world state as a single discrete random variable, where the values are the possibles states of the world. This will be the state variable $X_t$, which is a vector of the size $S$, where $S$ is the number of possible states of the world. The *transition model* $\boldsymbol{P}(X_t \mid X_{t-1})$ is written as a $S \times S$ matrix $\boldsymbol{T}$, where $\boldsymbol{T}_{ij} = P(X_t = j \mid X_{t-1} = i)$, the transition from state $i$ to state $j$. Next the sensor model is written as a diagonal matrix $\boldsymbol{O}$, where the diagonal $\boldsymbol{O}_i = P(e_t \mid X_t = i)$ and the rest are 0. The forward and backward messages then becomes:

$$\boldsymbol{f}_{1:t+1} = \alpha \boldsymbol{O}_{t+1} \boldsymbol{T}^T \boldsymbol{f}_{1:t} \tag{15.11}$$

$$\boldsymbol{b}_{k+1:t} = \boldsymbol{T} \boldsymbol{O}_{k+1} \boldsymbol{b}_{k+2:t} \tag{15.12}$$

In the real world there are often failures in the observable variables, like a sensor failing, both temporally and permanently. These event can be handled using *Dynamic Bayesian Networks*(DBN) which incorporates several different models:

- *Transient failure model*: This model handles the event of a temporary error in the observable variable by adding a small probability that the wrong value might be observed.
- *Persistent failure model*: This model handles the event when the the observable variable permanently return the wrong value. This is done by adding a new variable to the DBN that represents the state of the sensor of that observable variable.

## 16 Making simple decisions

When dealing with decisions we define $RESULT(a)$ as a random variable whose values are possible outcome states. The probability of outcome $s'$ of an action $a$ given evidence $\boldsymbol{e}$ is written as: $P(RESULT(a) = s' \mid a, \boldsymbol{e})$. An agents preference is captured by the *utility function*, $U(s)$, and the *expected utility* of an action given evidence, $EU(a \mid \boldsymbol{e})$, is just the average utility value of the possible outcomes:

$$EU(a \mid \boldsymbol{e}) = \sum_{s'} P(RESULT(a) = s' \mid a, \boldsymbol{e}) U(s') \tag{16.1}$$

The principle of *maximum expected utility*(MEU) says that a rational agent should choose the action that maximises the agent's expected utility; $action = \text{argmax}_a EU(a \mid \boldsymbol{e})$ Note that the utility score is not the same as the performance score, as the utility score is the expected value of an action, the performance score is the actual value of the action. For an agent to be *unbiased* the difference of expected value and predicted value must be 0.

Some notation on preference:

A $\prec$ B A is preferred over B

A $\sim$ B  A and B are equally preferred

A $\precsim$ B  A is preferred over B or they are equal

The probability of the outcomes of an action is represented as a lottery $L$ with each outcome state, $S_1, ..., S_n$, and their probabilities, $p_1, ..., p_n$; $L = [p_1, S_1; p_2, S_2; ...; p_n, S_n]$.

For a preference relation to be reasonable it must fulfil the following restraints:

- *Orderability:* Exactly one of the following must hold; $(A \prec B), (B \prec A), (A \sim B)$
  The sates must either be equal or one can order them.
- *Transitivity:* $(A \prec B) \wedge (B \prec C) \implies (A \prec C)$
- *Continuity:* $A \prec B \prec C \implies \exists p[p, A; 1-p, C] \sim B$
  There must be a way to "represent" the outcome of B, by using A and B.
- *Substitutability:* $A \sim B \implies [p, A; 1-p, C] \sim [p, B; 1-p, C]$
  Here $\sim$ can be swapped with $\prec$.
- *Monotonicity:* $A \prec B \implies (p > q \iff [p, A; 1-p, B] \prec [q, A; 1-q, B])$
  If A is preferred over B, then the lottery with the highest probability of getting A is preferred.
- *Decomposability:* $[p, A; 1-p, [q, B; 1-q, C]] \sim [p, A; (1-p)q, B; (1-p)(1-q), C]$
  Two consecutive lotteries can be turned into one.

Some utility function properties:

- $U(A) < U(B) \iff A \prec B$
- $U(A) = U(B) \iff A \sim B$
- The utility of a lottery is defined as:
  $U(L) = U([p_1, S_1; p_2, S_2; ...; p_n, S_n]) = \sum_i p_i U(S_i)$

As an agent might only be interested in ordering states, the actual utility value isn't necessary, these functions are called *value functions* or *ordinal utility functions*. *Preference elicitation* is used to discover a utility function and is done by presenting choices to an agent the using its observed preferences.

When an agent prefers to take risks for higher reward they are *risk-seeking*, the reverse is *risk-averse*, preferring safer bets. The value an agent accepts as guaranteed after a bet is the *certainty equivalent*, the difference between this end the *expected monetary value* (EMV) is the *insurance premium*.

*Normative theory* is the theory of how an agent should act, and *descriptive theory* is the theory of how the agent actually acts. Not all actions are completely rational as covered by the *certainty effect*, that a person usually chooses the guaranteed win. Something similar is covered by the *ambiguity aversion*, that people go for the known probabilities, instead of the unknown probabilities.

Problems where the utility is based on several variables are called *multiattribute utility theory*. Here the terms *strict dominance* means that a solution $S_1$ is better in every utility aspect than an solution $S_2$, and *stochastic dominance*, if two actions, $A_1, A_2$ leads to probability distributions $p_1(x), p_2(x)$ the $A_1$ stochastic dominates $A_2$ if: $\forall x \int_{-\infty}^{x} p_1(x')dx' \leq \int_{-\infty}^{x} p_2(x')dx'$.

*Preference independence* is when two variables are not dependent on the value of a third variable. *Mutual preferential independence* (MPI) says that each variable is equally important, so they don't

affect the other variables during a trade off. For attributes that are MPI the preferred behaviour can be described by maximising the function *additive value function*, $V(x_1, ..., x_n) = \sum_i V_i(x_i)$. The preference independence can be extended with *utility independence*(UI) to cover lotteries. A set of attributes is *mutually utility independent* (MUI) if each of its subsets is UI of the remaining attributes. MUI can be expressed using a *multiplicative utility function*.

Rational decisions can be done using decision networks, an extension of Bayesian networks. The network is constructed by the following node types:

- *Chance nodes:* same as for Bayesian networks, circles with variables
- *Decision nodes:* rectangles representing where the decision maker can take actions
- *Utility nodes:* diamonds representing the utility function.

These networks can be evaluated in the following way:

1. Set the evidence variables for the current state.
2. For each possible value of the decision node:
   (a) Set the decision node to that value.
   (b) Calculate the posterior probabilities for the parent nodes of the utility node, using a standard probabilistic inference algorithm
   (c) Calculate the resulting utility for the action.
3. Return the action with the highest utility.

In information value theory the *value of perfect information*(VPI) is expressed as the following:

$$VPI_{\boldsymbol{e}}(E_j) = (\sum_k P(E_j = e_{jk} \mid \boldsymbol{e}) EU(\alpha_{e_{jk}} \mid \boldsymbol{e}, E_j = e_{jk})) - EU(\alpha \mid \boldsymbol{e})$$

Where $Ej$ is the variable we don't know but want to find out, $\boldsymbol{e}$ is the observations so far, $e_{jk}$ is the new evidence and $\alpha$ is the value of the current best action.

Lastly in decision-theoretic expert systems *decision analysis* studies the application of decision theory on actual decision problems. There are usually talk about two roles, the *decision maker* that states preferences about outcomes, and the *decision analyst* that enumerates the possible actions and outcomes and elicits preferences from the decision maker to determine the best action. A decision-theoretic expert system is created in the following way:

1. *Create a causal model:* determine all possible variables, utility functions and actions.
2. *Simplify to a qualitative decision model:* Remove variables that does not affect the choice of action.
3. *Assign probabilities*
4. *Assign utilities:* These can be enumerated from the probabilities
5. *Verify and refine the model:* Using a *gold standard*(validation set)
6. *Perform sensitivity analysis:* If small changes results in big changes in the chosen action, the system might not be ready.

# 17 Making complex decisions

Focuses on *sequential decision problems* where the utility function depends on a sequence of decisions. First we look at the environment as *fully observable*, that is has a *transition model*, $P(s' \mid a, s)$ and that the transitions are *Markovian*. For each state an agent receives a reward $R(s)$ depending on that state and the utility function is just the sum of these rewards for the *environment history*. This is called a *Markov decision problem*(MDP).

When adding uncertainty to actions, i.e. there is a probability to take a right instead of a left, a sequence of actions does not guarantee success, therefore only the optimal action at each state is stored, $\pi(s)$, called a *policy*. The *optimal policy* for a state is the policy that gives the highest expected utility. As the performance of an agent is based on the states it visited this becomes a story in *multiattribute utility theory*. There is also the properties of *finite* or *infinite horizon*, where the agent either must reach a terminate state after $N$ states, or has no such constriction, on wards the an infinite horizon is assumed. If the optimal policy in a state, $\pi^*(s)$, changes over time it is *nonstationary*, and if it does not it is *stationary*. If an agent is stationary for preference it has the same preference-order for $[s_0, s_1, s_2, ...]$ and $[s'_0, s'_1, s'_2, ...]$, as for $[s_1, s_2, ...]$ and $[s'_1, s'_2, ...]$ iff $s_0 = s'_0$.

If a system is stationary the following utility function can be assigned:

1. *Additive rewards:* $U_h([s_0, s_1, s_2, ...]) = \sum_{i=0} R(s_i)$
2. *Discounted rewards:* $U_h([s_0, s_1, s_2, ...]) = \sum_{i=0} \gamma^i R(s_i)$
   Here $0 < \gamma \leq 1$ is the *discount factor*, this also gives an interest rate of $(1/\gamma) - 1$.

When assuming the infinite horizon these functions become tricky as they both go on to plus/minus infinity and we then have to order infinities. There are three solutions to this:

1. *Discounted rewards:* If $\gamma < 1$ this is infinite geometric series, which is finite:
   $U_h([s_0, s_1, s_2, ...]) = \sum_{i=0}^{\infty} \gamma^i R(s_i) \leq \sum_{i=0}^{\infty} \gamma^i R_{\max} = R_{\max}/(1-\gamma)$
2. *Proper policy:* This requires there to be a terminate state, so infinite series are never encountered.
3. *Average reward:* Averaging the rewards gives a way to handles the infinities.

Note, discounted rewards are the best.

Let the random variable $S_t$ denote the state the agent reaches at time step $t$ when using policy $\pi$. the expected utility then becomes:

$$U^{\pi}(s) = E[\sum_{t=0}^{\infty} \gamma^t R(S_t)] \tag{17.2}$$

Next let $\pi_s^*$ denote the optimal policy of state $s$, and it is chosen by:

$$\pi_s^* = \text{argmax}_{\pi} U^{\pi}(s) \tag{17.3}$$

And the best action can be calculated by:

$$\pi^*(s) = \text{argmax}_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U(s') \tag{17.4}$$

To calculate the optimal utility one of two algorithms can be used; *value iteration* or *policy iteration*. The value iteration algorithm is based on the *Bellman equation*, these also give unique solutions:

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s' \in NEIGH(s)} P(s' \mid s, a) U(s') \tag{17.5}$$

These equations are non linear and must therefore be solved using non-regular methods, one is iterative where each state value is updated for each iteration by the *Bellman update*:

$$U_{i+1} \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s' \in NEIGH(s)} P(s' \mid s, a) U_i(s') \tag{17.6}$$

By showing that the Bellman update is a *contraction* we guarantee that the value iteration converges. There are two properties of a contraction function; it has only one fixed point and for all inputs to the function, the output is closer to the fixed point. By viewing the Bellman update as a operator $B$ and $U_i, U_i'$ as vectors of utilities at iteration $i$, $U$ as the true utilities and the max norm of a vector $\|U\| = \max_s \mid U(s) \mid$ it can be shown that value iteration converges:

$$\begin{array}{rl} \left\|BU_i - BU_i'\right\| \leq \gamma \left\|U_i - U_I'\right\| & B \text{ is an contraction of factor } \gamma \\ \|BU_i - U\| \leq \gamma \|U_i - U\| & \text{using that } BU = U \text{ it shows that for each iteration} \\ & \text{the error is reduced by at least } \gamma \end{array} \tag{17.7}$$

From this it is also possible to calculate number of iterations needed to reach an error of size $\epsilon$; $N = \lceil \log(2R_{\max}/\epsilon(1-\gamma))/log(1/\gamma) \rceil$
Through contraction it is also shown that if the update is small then the error is also small; $\|U_{i+1} - U_i\| < \epsilon(1-\gamma)/\gamma \implies \|U_{i+1} - U\| < \epsilon$
Further the *policy loss* is connected to error by the following, where $U^{\pi_i}$ is the policy if $\pi_i$ was executed: $\|U_i - U\| < \epsilon \implies \|U^{\pi_i} - U\| < 2\epsilon\gamma/(1-\gamma)$

Next we look at the *policy iteration* that contains two steps:

1. *Policy evaluation:* given a policy $\pi_i$, calculate the utility of each state if $\pi_i$ was executed,$U_i = U^{\pi_i}$.

2. *Policy improvement:* Calculate a new MEU policy $\pi_{i+1}$, using one-step look-ahead based on $U_i$.

To improve on value iteration, the max operation is removed to make the algorithm linear by using $\pi_i$ instead of finding the best action, changing the Bellman equation to: $U_i(s) = R(s) + \gamma \sum_{s'} P(s' \mid s, \pi_i(s)) U_i(s')$, and the Bellman update to: $U_{i+1}(s) \leftarrow R(s) + \gamma \sum_{s'} P(s' \mid s, \pi_i(s)) U_i(s')$. This algorithm is called the *modified policy iteration*.

Next we look at *partially observable MDPs*(POMDP), which uses the same elements as a MDP, but extends it with a *senor model*, $P(e \mid s)$. Given the old belief state $b(s)$, an action $a$ and perceived evidence $e$ the next belief state becomes the following: $b'(s') = \alpha P(e \mid s') \sum_s P(s' \mid s, a) b(s)$ or $b' = FORWARD(b, a, e)$

# 18 Learning from examples

*Learning* is when an agent improves its performance by making observations of the environment. The improvements and techniques used for improvements are dependent on the following factors:

- Which components are improved; these can be for i.e. A mapping from the current state to actions, infer relevant properties from the precept sequence...
- What prior knowledge the agent has
- What representation is used, in todays machine learning this usually is a *factored representation*(feature vectors).
- What feedback is available. This is the difference between *unsupervised learning*(no feedback), *reinforcement learning*(gets rewards or punishments) and *supervised learning*(mapping input to output, so it knows the answer from the beginning). *Semi-supervised learning* is having a partially labelled training set.

Finding a function/rule from input-output pairs is called *inductive learning*. *Analytic* or *deductive learning* is to deduct new rules that entails from the previous known rules.

A *training set* is a set of input-output pairs, and a system tries to give a mapping(function) from the input to the output, this mapping is called the *hypotheses*, $h$. The hypotheses is tested using a *testing set*. A hypotheses *generalises* well if it scores high on the testing set. When the output, $y$, is an element of a finite set its a *classification*, while if $y$ is a number its called a *regression*. The set of possible hypotheses is called the hypotheses space, $\mathbb{H}$. If $h$ agrees with all the data points it's called *consistent*. The problem of choosing between several consistent hypotheses is called the *Ockham's razor*. The *true function*, $f$, is the correct mapping from input to output and a problem is *realisable* if the $f \in \mathbb{H}$. There is a trade-off between the expressiveness of $\mathbb{H}$ and the complexity of finding a good $h$.

A *decision tree* is a representation of a multiattribute function that returns a single output. The output variable is called the *goal predicate* and its values are the leaf nodes of the tree. The tree learning algorithm is greedy, as it chooses the most important attribute first, and solves the problem recursively by removing that attribute from the problem and the examples it can classify with just that attribute. For this recursion to work there are 4 cases to handle:

1. If the remaining examples are all of the same classification, we are done.
2. If there are several classifications of the examples, choose the best attribute to split the examples on.
3. If there is no examples left we have found an unknown combination of attributes the only thing that can be done is to return the *plurality* classification of all the examples(the most common output value).
4. If there are no attributes left, but examples of different classifications the only thing that can be done is to return the *plurality* classification of the remaining examples.

*Entropy* is the measure of uncertainty of a random variable, and is defined as: $H(V) = \sum_k P(v_k) \log_2 \frac{1}{P(v_k)} = -\sum_k P(v_k) \log_2 P(v_k)$
This is generalised to Boolean variables with a probability of $q$ of being true: $B(q) = -(q \log_2 q + (1-q) \log_2 (1-q))$
If a training set has p positive examples and n negative, the entropy of the goal attribute is: $H(Goal) = B(\frac{p}{p+n})$. A attribute $A$ with $d$ distinct values divides the training set $E$ into $d$ subsets. Each subset $E_k$ has $p_k$ and $n_k$ pos/neg examples so that we need $B(p_k/(p_k + n_k))$ extra bits of information to answer a query. A randomly chosen example has the $k$th value for the attribute with a probability of $(p_k + n_k)/(p + n)$ so the remaining entropy after testing on attribute A is:

$Remainder(A) = \sum_{k=0}^{d} \frac{p_k+n_k}{p+n} B(\frac{p_k}{p_k+n_k})$. The *information gained* from testing on $A$ is the reduction in entropy: $Gain(A) = B(\frac{p}{n+k}) - Remainder(A)$

*Overfitting* is when the learner deduce rules that are just accidentally in the data set, like if the learner only has two examples of a die being thrown and it becomes a 6, and the thrower also has a white t-shirt, the learner might say that each time a die is thrown by someone in a white shirts it will be a 6. To avoid overfitting in decision tree learning *decision tree pruning* is used, iterate over the nodes above the leaf nodes and see if they are irrelevant(there is only noise in the leaf nodes) it is turned into a leaf node. This discovery is done by a *significance test* where we assume the *null hypotheses* that there are no underlying pattern in the data and we calculate a distribution that then either verifies or rejects the hypotheses, in decision tree algortihms this is usually done using $\chi^2$-pruning. A combination of $\chi^2$-pruning and information gain is called *early stopping*.

To extend the applicability of decision trees some of the following issues must be addressed:

- *Missing data:* Not having values for every attribute in every example
- *Multivalued attributes:* If an attribute gives a classification for several examples it will have great information gain, but it might not be usefull(splitting on time for example). This is solved by using *gain ratio*.
- *Continuous and integer-valued input attributes:* Find the best place for a *split point*(where to split on height to get the biggest information gain).
- *Continouos-valued ouput attributes:* If the output should be continuous the tree becomes a *regression tree* and the leaf node is usually a function.

When choosing the best hypotheses the *stationary assumption* is made, each example is independent of the previous one and each example has the same probability distribution of its attributes. These examples are called *independent and identically distributed*(I.I.D). The *error rate* is defined as the proportion of mistakes it makes, i.e. $h(x) = y', y \neq y'$ for an $(x, y)$ example. Splitting examples into training and test set is called *holdout cross-validation*, but this reserves some of the examples for testing, and reduces the number of examples for training. A solution is *k-fold cross-validation*; Split the data set into $k$ sets, perform $k$ rounds of training with $1/k$ of the subset as testing data. If $n = k$ this is called *leave-one-out cross-validation*(LOOCV). Another problem is *peeking* at the test data, usually done by tweaking the settings of the system based on error rates of the test set, this is avoided by using a *validation set* that is not handled like this.

There are two ways to improve the choosing of the best hypotheses; improving *model selection*, defining a good hypotheses space; or *optimisation* finding the best hypotheses. An algorithm for this is a *wrapper* that tests learners based on a variable, *size*, which corresponds to the complexity of the hypotheses space, and returns the best learner after looking at every size of complexity in the hypotheses space.

Next we want to maximise the expected utility of the learner, not just minimise the error rate, this is usually done by minimising *loss function*, $L(x, y, y') = U(f(x) = y) - U(h(x) = y')$. The *generalisation loss* of $h$ with respect to $L$ is using the set of all examples $\epsilon$: $GenLoss_L(h) = \sum_{(x,y)\in\epsilon} L(y, h(x))P(x, y)$
and the best hypotheses is defined as: $h^* = \text{argmin}_h GenLoss_L(h)$
but since $P(x, y)$ is unknown this loss is estimated with *empirical loss* on a set of examples, $E$: $EmpLoss_{L,E}(h) = \frac{1}{N} \sum_{(x,y)\in E} L(y, h(x))$
and the estimated best hypotheses $h' = \text{argmin}_h EmpLoss_{L,E}(h)$. $h'$ and $h^*$ differ for 4 reasons:

- *Unreliability* , $f \notin \mathbb{H}$

- *Variance*, different sets of examples generates different learners
- *Noise*, $f$ is non deterministic, $f(x) = y \wedge f(x) = z$(also called *noisy*)
- *Computational complexity*, if $\mathbb{H}$ is big or complex it might be to hard to search it all.

*Regularisation* is the process of penalising complex hypotheses. Another way to "decomplexify" $\mathbb{H}$ is by *feature selection*, where some features are ignored.

### Neural networks

A NN is a network of nodes connected by links. A link between the nodes $i$ and $j$ propagates the *activation* $a_i$ from $i$ to $j$, it also has a *weight* $w_{ij}$ which determines the strength of the activation from $i$ to $j$. To active, node $j$ sums its inputs: $in_j = \sum w_{ij} a_i$
and then applies its *activation function*, $g$: $a_j = g(in_j)$. A network where the links only go one way is called *feed-forward network* where each node has its inputs from the upstream nodes and sends the activation to the downstream nodes. A *recurrent network* sends its activation's back trough the network. Using several *hidden layers* a NN can do *nonlinear regression*. Learning is done by *back-propagation*, which changes the weights in the network based on a learning rate, $\alpha$, and that weights "fault" for loss:

$$w_{jk} \leftarrow w_{jk} + \alpha a_j \delta_k \tag{18.11}$$

where $\delta_k = Err_k g'(in_k)$ and $Err_k$ is the $k$th component of the error vector $\vec{y} - \vec{h}_w$ NNs can be overfitted and as cross-validation can be used to avoid peeking. *Optimal brain damage* can also be used in NN to remove unused links.

Models where the number of input variables are bounded are called *parametric models*, the opposite are called *nonparametric models*. Examples of these are *instance-based learning or memory-based learning* where the hypotheses uses previous examples to infer a solution to the next problem. One example is the $k - nearestneighbors$, where the "closest" looking examples are used to classify/find a solution to a new example. The distance here is typically measured in *Minkowski distance*, or $L^p$-norm: $L^p(\vec{x}_j, \vec{x}_q) = (\sum_i \mid x_{ji} - x_{qi} \mid^p)^{1/p}$. To avoid one dimension of the attribute vector to dominate, the components are *normalised*.

## 19 Reinforcement learning

TODO: Mangler chain rule i chap 13, conditionaly independence and conditionaly dependence

## 20 Philosophical Foundations

## 21 AI: The Present and Future

# 2 CBR paper

## 1 Background

CBR is a problem solving paradigm that uses specific knowledge of previous experiences on new problems(cases). It is a incremental, sustained learning method, as it can add each solved problem to its case base, adapting to new problems and solutions. Because it learns and the machine learning community is the main contributors to the research forward, CBR is seen as a sub-field of machine learning. The paper states lots of empirical evidence that humans uses CBR in their reasoning. Examples of how humans use CBR can be:

- Physician: reuses diagnosis and treatments on patient that has the same type of symptoms as a previous patient.
- Financial consultant: Decides to give a loan to a new customer based on a similar previous customer.

Note that CBR does not just use the same solution on the same type of problem, it uses a solution to a problem that it remembers working.

CBR requires methods to; extract relevant knowledge, integrate each case into the knowledge base/structure and index the cases for later use. One issue with CBR, if cognitive plausibility is a guiding principle, is that it should also incorporate other types of knowledge as well, not only from previous cases.

## 2 History

A lot of people have worked with CBR-systems through the years, noting the CREEK system and integration framework from NTNU Trondheim developed by Agnar Aamodt and sintef, because NTNU and sintef.

## 3 Fundamentals of case-based reasoning methods

There are 5 main differences in CBR methods; 1. how they id the current case, 2. how they find past similar cases, 3. how they use those cases to propose a solution to the current case, 4. how they evaluate the proposed solution and 5. how they learn from that solution. Further more there are 5 main types of CBR:

- Exemplar-based reasoning classifies a new case based on the most similar previous case. Exemplar is just another word for case. It does not change the solution of the previous case in any way.

- Instance-based reasoning is a highly syntactic specialisation of exemplar-based reasoning. It uses simple representations(usually a feature-vector) and focuses on automated learning(no human interaction while learning). This is a non-generalisation approach and is basicly very strict exemplar-based reasoning. Learning OR/AND examples.
- Memory-based reasoning collects cases into a huge memory and reasons through searching and accessing this memory. Therefore has a big focus on organising and accessing this memory. The utilization of parallel processing techniques is a characteristic of these methods.
- Case-based reasoning covers the most typical methods. Cases are usually assumed to have a certain degree of richness and complexity in information. These methods are capable of modifying/adapting solutions and utilises general background knowledge. They also borrows theories from cognitive psychology.
- Analogy-based reasoning solves new cases based on previous cases from another problem domain. It is a sub-field concerning identification and utilization of cross-domain analogies. The mapping problem is a central problem in this field: How to map the solution from an analogue(called the source or base) to the present problem(called the target).

All CBR system implements the following four core function, called the CBR cycle:

- Retrive the most simialar case/cases
- Reuse the information/knowledge of that case
- Revise the proposed solution
- Retain parts of the experience likely to be usefull in the future

> Together with the CBR cycle, a CBR system can be analysed in a task oriented view where the system is viewed as an agent that has goals and the means to achieve them. A system can be described from three different perspectives; Tasks, methods and domain knowledge models.

There are 5 problem areas that a CBR methods must have coherent solutions for to be called a CBR:

- Knowledge representation
- Retrieval method
- Reuse method
- Revise method
- Retain method

# 4 Representation of cases/Knowledge representation

CBR uses past cases to solve new ones, so representing past cases(knowledge base) in a way that makes searching it both effective and time efficient, is key. It is also important to store new cases and solutions. From this the main representation problem in CBR becomes what to store in a case, how the cases content should be structured, how all the cases should be organised/indexed for effective retrival and reuse, and how general domain knowledge can be integrated into this structure. The paper reviews two models; The dynamic memory model of Schank and Kolodner, and the category-exemplar model of Porter and Bareiss.

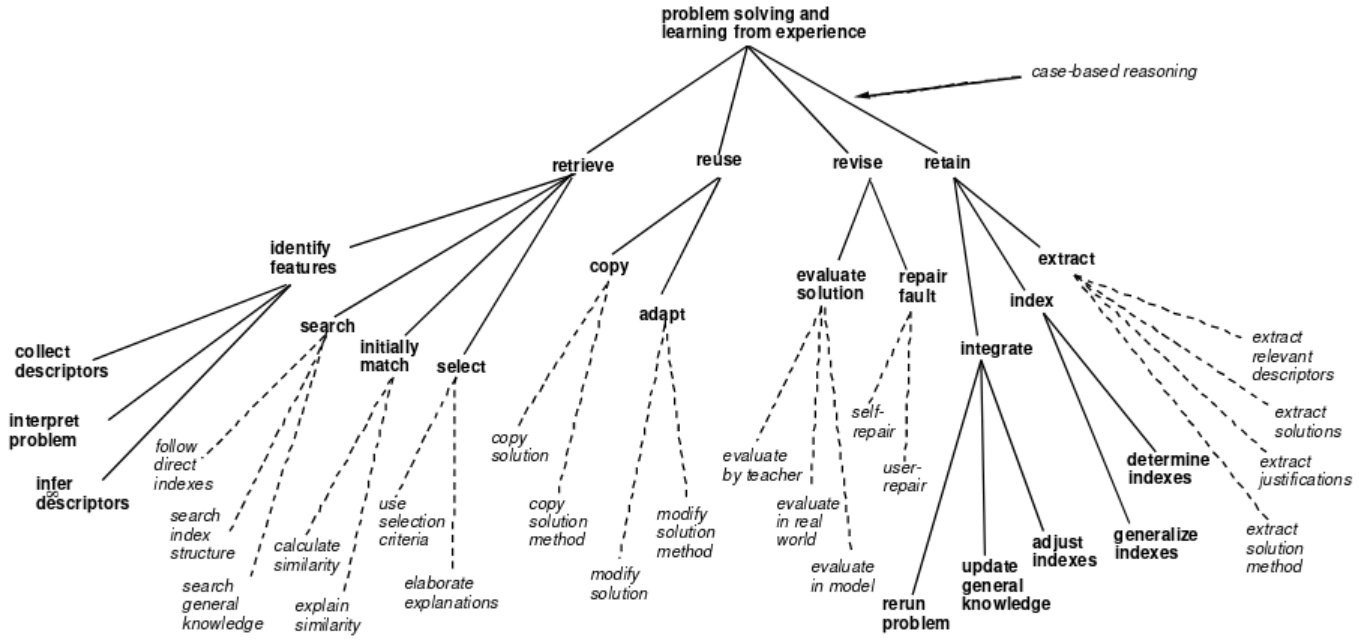


**Figure 2. A task-method decomposition of CBR**

## 4.1 The Dynamic Memory Model

The dynamic memory model is a hierarchical structure of episodic memory organisation packets. It organises specific cases with simialar properties under a general structure(a generalized episode - GE), this episode contains three different types of objects: Norms, cases, and indices. The norms are the common features of all the cases of a GE and the indices are the distinct features for each case. And an index works like index do work, an index name and index value and points to either a GE or a specific case. Searching in the tree is done by "pushing down" the case through the network structure. The model is dynamic in the sense that when a new case is to be stored, it is again "pushed down" through the network and if it ends up with the same index as another a new GE is created. It is claimed that this model is suitable for learning both generalized and case specific knowledge and that it is a model of human reasoning and learning.

## 4.2 Category & Exemplar Model

The idea behind this model is that real world/natural concepts should be defined extensionally and that different features are assigned different importance' to a case's membership to a category. The case memory is embedded in a network structure of categories, semantic relations, cases and index pointers. Similar to the dynamic model a index pointer may point to either a category or a case. The indices are one of three types: feature links(pointing from features to categories), case links(pointing from categories to cases/categories) and difference links(pointing between similar cases). The cases of a category is stored according to how "typical" of that category they are(their prototypicality). Case matching(searching) is done by entering the new case's features and returning the most prototypical cases of the matching category. Similar to searching, a new case is stored by finding a matching case and creating the appropriate feature

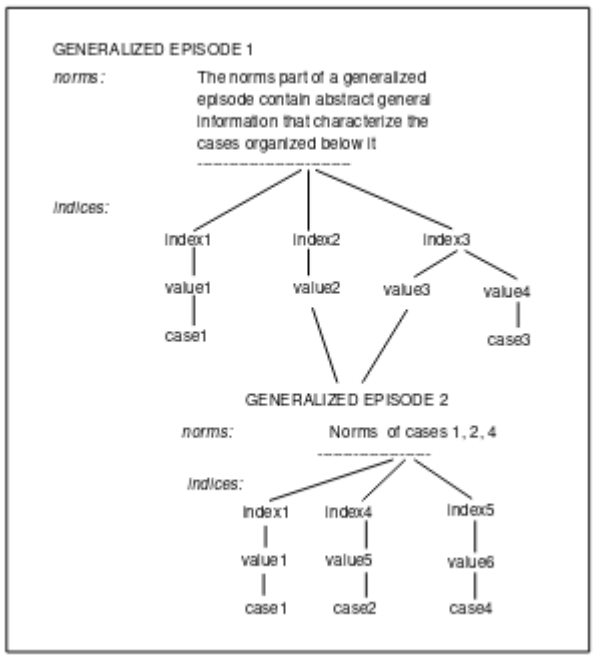


Figure 2.1: Structure of the dynamic model. It constructs a tree where each node is a GE with norms that are further seperated into indexes. Each index-value pair points from one GE to either a new GE or a case.

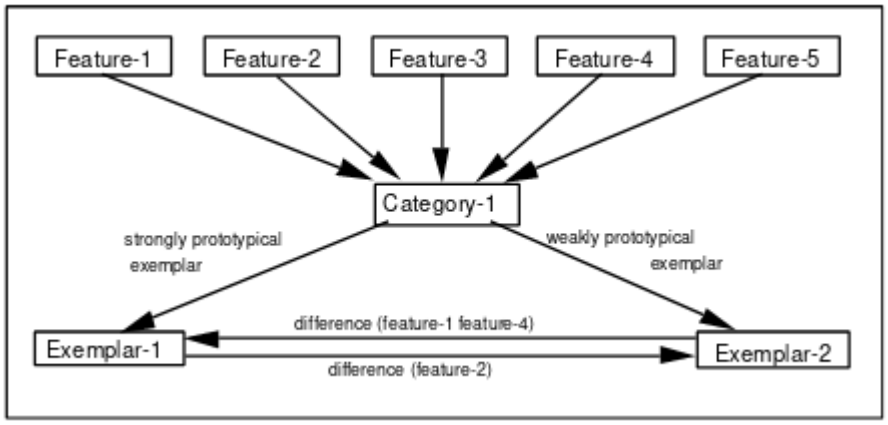


Figure 2.2: Structure of the Category & Exemplar model.

indices. If the case is sufficiently similar to an already existing case, it might not be retained at all, or merged with the matching case.

# 5 Case retrival

The Retrive method takes a problem description and returns a best matching previous case. It is split into 4 subtasks;

- Identify Features: Identify a new case's features/problem descriptors for use to match similar previous cases.
- Initial Match: take the features found in the previous task and return a set of similar cases.
- Search: A more elaborate search to reduces the number of cases found initially.
- Select: Chooses the best solution based on the cases found in previous task.

## 6 Case Reuse

The reuse of the retrieved case solution in the context of the new case focuses on two aspects: (a) the differences among the past and the current case, and (b) what part of retrieved case can be transferred to the new case. The possible subtasks of Reuse are Copy and Adapt;

1. Copy: Just reuse the solution of a previous case, not considering (a) and (b)
2. Adapt: There are two ways to reuse past cases with adaption
    - Transformational reuse: Reuse and adapt the past case solution
    - Derivational reuse: Reuse and adapt the past *method* that constructed the past solution

## 7 Case Revision

When a solution from the reuse phase is not correct, an opportunity to learn from the failure arises. This is seperated into two tasks;

- Evaluate solution: See the results of applying the solution to the environment.
- Repair fault: Detect the errors of the current solution and retrieve/generate explanations for the errors. Use these to modify(repair) the solution and retain usefull knowledge if possible. Then repeat the revision untill the solution is valid.

## 8 Case Retainment - Learning

This phase is triggered by the outcome of the case revision phase and involves selecting which information to retain, in what form it is retained, how to index the case, and how to intgrate the case into the memory structure.

- Extraction: Incorperate the new case into the memory or generalize the chosen previous case by retaining information from the current case. This information being gather from the Reuse or Revision phase, as the solution method, not just the concrete solution.
- Index: The choice of indexing method influences how the cases are structured(its search space). This is a knowledge acquisition problem and should be handled in the modeling step[1]. For examples see Sections 4.1 and 4.2

[1] As in the system modelling step

# 3 Deep learning

A deep learning system is a neural network. Neurons[1] are set up in several layers, where a number of inputs are sent in to the first layer, and the last layer gives an output.

Its the best, deep learning(DL) systems have beaten records in image recognition, speech recognition, predicting the activity of potential drug molecules, analysing particle accelerator data, reconstructing brain circuits, and predicting the effects of mutations in non-coding DNA on gene expression and disease, and more.

## 1 Supervised learning

Here the system is trained using a data set of inputs and a class. It takes each input and sends it through the system, and uses an *error function*, also called *objective function* or *loss function*, to calculate the error(or distance) between the desired output and the actual output. This error is then used to modify the weights of the network, this in effect nudges the network to be more accurate when given a "similar" problem. This weight adjustment is done by calculating each weights gradient, which is a number representing that weights impact on the error/objective function. When classifing, usually the weighted sum of the outputs are used to find which category the input belongs to using thresholds.

From early on linear(also called shallow) classifieres(networks with few neurons and layer, i.e. the perceptron) could only classify simple linearly seperable problem spaces[2]. By using deeper(several layer) networks this problem can be solved.

## 2 Backpropagation

This is the process where the gradient(derivate) of the error function is calculated with respect to each weight. Given an example from the training set of inputs and target, backpropagation is done by first sending the inputs through the network, this is the *feed forward* step, this gives an output from the network that is compared with the target of the example. Then, working backwards from the output layer the gradient is calculated for each weight in the layer and the weight is updated. More formally the steps are:

1. Take inputs(the feature vector) $\vec{x} = \langle x_1, x_2, ..x_d \rangle$ and the target $y^*$.

[1] A neuron is basicly a function(called the activation function) that takes inputs from the previous nodes in the network, toghether with weights and possible biases and the functions output(aka the neurons output) is sent to the next layer of neurons.

[2] A linearly seperable problem is a problem where, if the input(feature vector) and the target(correct classification of that input) is plotted on a graph(this plot is the search space/problem space) the different classes can be seperated by lines.

2. Feed the inputs through the network, using the weighted sum of the outputs of the previous layer as inputs to each activation function of the next layer untill the prediction(output of the network) is calculated, $y$.

3. Calculate the error $E = y - y^*$

4. backpropagate the error:

   (a) TODO: Add backprop algorithm

A system using the algorithm presented above is called a feedforward neural network and are used to map fixed-sized inputs, to fixed-sized outputs, like categorizing an image. The nodes/neurons not in the input or output layers are called hidden nodes/neurons and are responsible for distorting the input in a way that makes the categories linearly seperable by the last layer.

# 3 Convolutional neural networks

CNN are an extention to regular neural networks that work well with multi dimentional inputs(images as 2d or 3d arrays) and can be constructed to be multidimentional(input, hidden and output layers have both a height and a width). In a convolutional layer the input of each node in the layer isn't connected to all the nodes of the previous layer, instead it takes inputs from a group of the nodes of the previous layer. There are also pooling layers, that work the same way, but instead of using an activation fuction like ReLU, it uses statistical functions, like output the max of the input values, or the average.

By using groups of inputs, the network can recognise shapes in the input, independent of location. For example, if the CNN can recognise a dog, it can recognise it anywhere in the picture, not just in the place where the dog was placed in the training set. Because of this, and the efficient use of GPUs, since 2012 CNN has taken over for older, classical computer-vision systems.

# 4 Distributed representations and language processing

# 5 Recurrent neural networks