# Artificial Intelligence Techniques - Lecture Notes

Notes to aid, not to replace. I'm simply a human, thus the risk you embrace.

Zohar Cochavi

# Introduction

Not much needs to be said regarding the introduction of probabilistic machine learning. However, one rule deserves special attention and a quick reminder is therefore in place.

## Bayes' Rule

In short, Bayes' rule allows us to 'update' our current belief, i.e. the probability of a state, $S$ given some possible actions, $a_0, \dots, a_i$, $P(S|a_0, \dots, a_i)$, using observations. That is, by 'flipping' the dependence, actions given the next state instead of the next state given actions, we can relate actions and states.

Let me clarify by first providing Bayes' rule:

$$P(S|a) = \frac{P(a|S)P(S)}{P(a)} \tag{1}$$

Instead of using $a$ for action, we can also use $o$ for observation. When considering a classification example, this makes more sense. In that case, we would like to 'learn' how certain attributes relate to classes (the states in the previous example). This is done by making 'observations' in the form of features and, since we already have the label, this results in *an observation given some label*. Using Bayes' theorem (see eq. 1), we can thus determine the *label given some features*.

The example of actions and states applies when considering an **agent** acting in an unfamiliar environment. Here, we consider $P(S)$ our **prior** belief, or the belief before (prior to) the observation, and 'update' the belief by considering our belief given some observations, $P(S|o)$.

This implies somehow that $P(o|S)$ is easier to determine than its inverse. [...]

# Imitation Learning - Inverse Reinforcement Learning

- Instead of learning everything ourselves, we want to use some expert examples and improve from there. This is called **imitation learning** or **behavioral cloning**.
- One technique for applying such learning is **inverse reinforcement learning** in which the model attempts to find the rewards associated with certain actions.

Table 1: Various methods for imitation learning and their different attributes{#tbl:imitation_learning}

| | Direct Policy Learning | Reward Learning | Access to Environment | Interactive Demonstrator | Pre-collected demonstrations |
|---|---|---|---|---|---|
| Behavioral cloning (BC) | Yes | No | No | No | Yes |
| Direct policy learning (interactive IL) | Yes | No | Yes | Yes | Optional |
| Inverse Reinforcement Learning (IRL) | No | Yes | Yes | No | Yes |
| Preference-based RL | No | Yes | Yes | Yes | No |

- The formal goal is as follows: find a reward function $R$ for which the expert-provided policy is optimal.
- This, however, is a rather 'vague' solution for which some heuristics will make developing a general solution a little easier:
  1. We prefer solutions where the distance (the difference in value is maximal) to other policies: $\max_R(\pi_R^* - \pi_R)$
  2. We prefer smaller rewards: $\min(R)$ or $\max(-R)$.

# Bayesian Networks and Inference

Core to the idea of an intelligent agent is that the agent somehow manages to solve a task for which the solution is not explicitly embedded into the agent. If we told the agent explicitly what to do, it would simply be a front for some deterministic algorithm. The lack of determinism therefore implies the need for the agent to handle uncertainty. Therefore, we need to be able to formally represent uncertainty.

More formally, an issue arises when an agent attempts to determine whether it will be able to complete a task. It might guarantee a certain outcome, *given* that a whole host of events do, or do not, take place. This just kicks the can down the road, as we then have to guarantee whether the dependencies for the result are also satisfied. This leads to something called the **qualification problem**.

Combining some fundamental intuition about probability and the issue of the qualification problem, we can look at events as a chain of probabilistic dependencies. One way of structuring such dependencies is in something called a **Bayesian Network**.

## Representing Knowledge in an Uncertain Domain

A Bayesian Network is a data structure that encapsulates the full joint probability distribution in a concise manner. Meaning it is nothing more than a representation of a (possibly very complex) function. Let's take the example of a patient having a cavity in their tooth, $P(Cavity)$ which might induce a toothache and can be observed with a 'catch' (see fig. ).

```
graph TD;
    Cavity-->Toothache;
    Cavity-->Catch;
    Weather;
```

There, we see how one random variable might influence two other random variables. Furthermore, there is no need for all the nodes to be connected, a forest is a valid instance of a Bayesian network (the weather is independent of the cavity).

There is another assumption hidden in this graph, namely that $Toothache$ and $Catch$ are **conditionally independent** given $Cavity$. Conditional independence is formalized by the following relation.

$$P(Catch \land Toothache|Cavity) = P(Catch|Cavity) \land P(Toothache|Cavity)$$

A formal specification of a Bayesian Network (BN) can be characterized as follows:

1. Each node corresponds to a random variable, which may be discrete or continuous.
2. A set of directed links or arrows connects pairs of nodes. If there is an arrow from node $X$ to node $Y$, $X$ is said to be a parent of $Y$. The graph has no directed cycles (and hence is a directed acyclic graph, or DAG).
3. Each node $X_i$ has a conditional probability distribution $P(X_i|Parents(X_i))$ that quantifies the effect of the parents on the node.

A more involved example would be the following, where an *Alarm* might be triggered by *Burglary* or an *Earthquake* which then proceeds to call John, $JohnCalls$, or Mary, $MaryCalls$.

```
graph TD;
    Burglary ---> Alarm;
    Earthquake ---> Alarm;
    Alarm ---> JohnCalls;
    Alarm ---> MaryCalls;
```

The **conditional probability table** (CPT) of *Alarm* can be found in tbl. 2. Since we are dealing with binary conditions (either $A$ happens or it doesn't), $P(\not{A})$ is implicit. In other cases, the sum of the probabilities in one row should equal 1 to be considered valid probabilities.

Table 2: Conditional probability table (CPT) of the *Alarm* event. Notice how dependence increases the size of the table according to $2^n$, where $n$ is the number of dependent variables. This is because we are dealing with Boolean conditions.

| B | E | P(A) |
|---|---|---|
| t | t | .95 |
| t | f | .94 |
| f | t | .29 |
| f | f | .001 |

## The Semantics of Bayesian Networks

As mentioned before, a BN is a representation of a joint distribution function. Another way to interpret it, however, is to view it as an encoding of a collection of conditional independence statements. Instead of focusing on how one is dependent on another, we think about which events are *not* dependent on another.

A BN can be described and used according to the following relation, which stems from the **chain rule**.

$$P(x_1, \dots, x_n) = \prod_{i=1}^{n} P(x_i | parents(X_i))$$

That is, the probability of a certain set of outcomes for the events described by the BN (i.e. the values of it's nodes) can be calculated by taking the product of the probabilities of all these events given their respective parent(s). The conditional probability used in the product series can then be described by the aforementioned CPT.

This means that, even though the BN is technically equivalent to the joint probability distribution, assuming the network is sparse (i.e. not all nodes are connected to all other nodes) it is more efficient in terms of computation. Consider the following: if a joint probability distribution of Boolean random variables scales according to $2^n$ where $n$ is the number of random variables, a BN scales according to $n2^k$ where $k$ is the average number of parents.

Formally, we can say that a node is conditionally independent of its **non-descendents** given its parents. Furthermore, we can say that a node is conditionally independent of the whole network given its parents, children, and children's parents, also called its **Markov Blanket**. The more connected a BN is, the bigger the average Markov Blanket of its nodes.

Instead of considering all combinations of events, we assume that some events do not influence one another. At the core of the BN lies this assumption, and one should therefore consider the balance between accuracy and computational complexity.

## Efficient Representation of Conditional Distributions

Assumptions can be made about the interaction of parents and how they relate to their children. We can assume logical or min/max statements, and from there construct the complete CPT. An example would be a *Fever* which could be caused by the *Flu*, a *Cold*, or *Malaria*. We then assume that these relate to each other in something called a **fuzzy OR** scenario, i.e. the relation tends to that of a logical OR. This means that we would only need, for example, the probability of someone to get a *Fever* from the *Cold* and the probability to get a *Fever* from the *Flu* to determine what the combined probability (see eq. 2)

$$\begin{gathered} P(Fever|Cold, Flu, \neg Malaria) = \\ (1 - P(\neg Fever|Cold, \neg Flu, \neg Malaria) \cdot P(\neg Fever|\neg Cold, Flu, \neg Malaria)) \end{gathered} \tag{2}$$

Instead of using a CPT, we can also employ 'normal' probability density functions (PDFs) to describe the probabilistic characteristics of a random variable. While we could use discretization to work with continious random variables, this often comes at a loss of accuracy and potentially unwieldy CPTs. There are some considerations to make with regard to **hybrid networks** where we find both discrete and continuous random variables. Most notably, when a continuous parent has a discrete child, we have to create some threshold function. The most interesting here is the so-called **logistic function** which is often used in neural networks for the same purpose.

## ⋈ Exact Inference

Now my friends, the time has come, to collect info and probably infer some. We know how BNs work and how to construct them. Now, we would like to use our model to infer information given some state, i.e. perform a query.

In this section, we will only discuss *exact* inference. First, we discuss the 'easy' way, and then apply some optimization on top of that in the form of memoization.

### Inference by Enumeration

Let's take the BN presented in fig. , and use the following query: $P(Burglary \mid JohnCalls = true, MaryCalls = true)$. Inference by enumeration simply takes all the possible contexts in which John might call, and where Mary might call and considers where this would have been caused by a burglary.

We can start solving our query using the following relation,

$$P(X \mid e) = \alpha P(X, e) = \alpha \sum_y P(X, e, y)$$

where $\alpha = \frac{1}{e}$.

Applying our query to the equation then gives,

$$P(B \mid j, m) = \alpha P(B, j, m) = \alpha \sum_e \sum_a P(B, j, m, e, a)$$

For the sake of simplicity, we only consider $B = true$, which can be rewritten to the following when taking our BN as the joint function (i.e. substitute $P(B, j, m, e, a$ with the characteristic of the BN).

$$P(b \mid j, m) = \alpha \sum_e \sum_a P(b)P(e)P(a \mid b, e)P(j \mid a)P(m \mid a)$$

Which can then be simplified as follows,

$$P(b \mid j, m) = \alpha P(b) \sum_{e} P(e) \sum_{a} P(a \mid b, e) P(j \mid a) P(m \mid a) \tag{3}$$

and then be solved by looking up the respective values in the CPTs.

## Variable Elimination

One can imagine the previous algorithm to be suboptimal. Considering the time complexity, we come to the conclusion that it runs in $\mathcal{O}(2^n)$. Realizing your algorithm runs in exponential time is never a nice experience.. But, it's a great improvement considering calculating the straight joint probability runs in $\mathcal{O}(n2^n)$.

One way in which we can improve the performance of the model, is by reusing past calculations (memoization, if you know, you know). This dynamic programming technique is called **variable elimination**. The idea is to solve the equation from the bottom up (that is, when viewed as a syntax tree, we start at the leaves and move up) so that we can re-use calculations we've made before.

[...]

# Maintaining a Belief over Time

## Introduction

- Changing the simple **sensor model** to a **transition model** by incorporating time steps, but no ability to which next state is the most likely (why? most probable is not most likely?).
- Different Types of models:
  - Hidden Markov Models
  - Kalman Filters
  - Dynamic Bayesian Networks
- Incorporating time can be important to, for example, take into account the rate of change (first derivative) when making decisions.

## Transition and Sensor Models

- **Markov assumption**: *"The current state depends on only a finite fixed number of previous states."* Any system or process that satisfies this assumption is called a **Markov Process** or **Markov Chain**.
- The number of dependent prior states, $n$, is integrated into the so-called order, a so-called $n$**th-order Markdown Process**.
- Formally, a first-order Markov process is denoted as:

$$P(X_t|X_{0:t-1}) = P(X_t|X_{t-1})$$

- Important is that the underlying laws of the system do not change, therefore, the distributions are stationary over $t$.
- Another important assumption is that of the **sensor Markov assumption** (see eq. 4). That is, the observable variable is only dependent on the hidden variable in the current time-step. Also called the observation model.

$$P(E_t|X_{0:t}, E_{0:t-1}) = P(E_t|X_t) \tag{4}$$

- The process has to initialize with a prior probability at $t = 0$, $P(X_0)$.
- Thus, we can denote the complete joint probability distribution for a **first-order Markov Process** (see eq. 5).

$$P(X_{0:t}, E_{1:t}) = P(X_0)\prod_{i=1}^{t} P(X_i|X_{i-1})P(E_i|X_i) \tag{5}$$

- The accuracy of such a model depends on how 'reasonable' the Markov assumption is, and how closely the chain of causality resembles the real world (the probability that someone brings an umbrella might also be dependent on the amount of sun, instead of only whether it's raining.).
- There are two ways to improve the accuracy of a model (which are re-formulations of one-another):
  1. Increase the order of the Markov process model.
  2. Increase the set of state variables.

## Inference in Temporal Models

- Various methods for inference exist which can complement each other to improve accuracy of the model besides just answering queries.
  1. **Filtering**: Informs our agent about the distribution of the current hidden state based on the observations made until now, $P(X_t|e_{1:t})$.
  2. **Prediction**: Determine the distribution of the next hidden state based on the observations made until now, $P(X_{t+1}|e_{1:t})$.
  3. **Smoothing**: Determine the distribution of a past hidden state, $0 \geq kt$, based on the observations made until now, $P(X_k|e_{1:t})$.
  4. **Most likely explanation**: Determine the most likely sequence of events based on the observations made until now, $P(x_{1:t}|e_{1:t})$.
- Another technique called **Learning**, is based on **smoothing** combined with the **EM** algorithm.

### Filtering And Prediction

- The process of filtering (see eq. 6) depends on prediction, (at least, based on the formulations mentioned above). ($\alpha$ is always a normalization constant).

$$P(X_{t+1}|e_{1:t+1}) = \alpha P(e_{t+1}|X_{t+1}) \sum_{x_t} P(X_{t+1}|x_t)P(x_t|e_{1:t}) \tag{6}$$

- They are two steps that are necessary for one another. Therefore, while they are separate queries, they could be considered two steps in the same process. To get $P(X_{t+1}|e_{1:t+1})$,
  1. Predict:
  $$P(X_{t+1}|e_{1:t}) = \sum_{x_{t-1}} P(X_t|x_{t-1})P(x_{t-1})$$
  2. Filter:
  $$P(X_{t+1}|e_{1:t+1}) = \alpha P(e_{t+1}|X_{t+1})P(X_{t+1}|e_{1:t})$$
- The initialization predict step could be considered as prediction with empty evidence (i.e. without evidence), $e_0$. In which case it reduces to the prior, $P(x_0|e_0) = P(x_0)$.

$$\begin{aligned}P(X_1|e_0) &= \sum_{x_0} P(X_1|x_0)P(x_0|e_0)\\ &= \sum_{x_0} P(X_1|x_0)P(x_0)\end{aligned}$$

### Smoothing

- With smoothing, we can essentially improve the model to take into account new observations, and can be described by eq. 7.

$$\begin{aligned}P(X_k|e_{1:t}) &= \alpha P(X_k|e_{1:k})P(e_{k+1:t}|X_k)\\ &= \alpha f_{1:k} \times b_{k+1:t}\end{aligned} \tag{7}$$

- The factors $f$ and $b$, implying forward and backward respectively, can then be described by eq. 6 and eq. 8 respectively.

$$\begin{aligned}P(e_{k+1:t}|X_k) &= \sum_{x_{k+1}} P(e_{k+1}|x_{k+1})P(e_{k+2:t}|x_{k+1})P(x_{k_1}|X_k)\\ \implies b_{k+1:t} &= Backward(b_{k+2:t}, e_{k+1})\end{aligned} \tag{8}$$

- While smoothing would take $O(t)$ for a single time-step, and thus $O(t^2)$ for all timesteps, by saving the intermediate results, we can smooth the whole sequence in $O(t)$ by applying the **forward-backward algorithm**.

### Most-likely Sequence

- Since the most-likely sequence is formed by the joint probability of all the hidden states, we cannot just apply smoothing and calculate the most likely hidden state based on that states' prior. I sort of get the idea why you can't just keep 'smoothing' over and over again and then use those states, but I can't quite put my finger on it.

$$\begin{aligned} &\max_{x_1 \dots x_t} P(x_1, \dots, x_t, X_{t+1} | e_{1:t+1}) \\ &= \alpha P(e_{t+1}|X_{t+1}) \max_{x_t} \left( P(X_{t+1}|x_t) \max_{x_1 \dots x_{t-1}} P(x_1, \dots, x_{t-1}, x_t | e_{1:t}) \right) \end{aligned} \tag{9}$$

- The algorithm to compute the most likely sequence is also called the **Viterbi algorithm** (see eq. 9).

## Dynamic Bayesian Networks

- Dynamic Bayesian Networks (DBNs) are a generalization of the HMMs. Every HMM is a single-variable DBN; every discrete DBN is an HMM. **A DBN can have multiple 'sensors', while a HMM can have only one** (that is one observable).
- Exact inference in DBNs becomes intractable as their size grows. Thus, we use approximate inference. Specifically, **particle filtering**:

  First, a population of $N$ initial-state samples is created by sampling from the prior distribution $P(X_0)$. Then, the update cycle is repeated for each time step:

  1. Each sample is propagated forward by sampling the next state value $x_{t+1}$ given the current value $x_t$ for the sample, based on the transition model $P(X_{t+1}|x_t)$.
  2. Each sample is weighted by the likelihood it assigns to the new evidence, $P(e_{t+1}|x_{t+1})$
  3. The population is resampled to generate a new population of $N$ sample. Each new sample is selected from the current population; the probability that a particular sample is selected is proportional to its weight. The new samples are unweighted.

# Learning

## Why Learning

1. Some model might not be available. In this case, it can be built up over time by some other algorithm. An example would be to train a deep learning model; there is no *specific* version of the model available to the programmers, but we can adapt a general model to our specific purposes.
2. …
3. Not understanding the problem.

## What is learning

While there are many approaches to learning, some of which might be more useful/correct in certain contexts than others. Here, we focus on **learning as induction**. How to use induction to learn is then dependent on the technique that is used to implement some agent.

Thus, while were are not explicitly building a model, we provide information to some statistical model to then make decisions. It is therefore important to decided what kind of information we should provide the model for it to make the right decision. Not only that, we also have to quantify what is the right decision.

We can split the space of what is the 'right' decision into two philosophies, the **Idealistic** and **Pragmatic** approach. The idealistic attempts to stay true to the way the world works, making observations and then using statistical models to learn. The pragmatic approach, however, simply considers "whatever improves the performance".

While the idealistic perspective would have great accuracy, the tractability, however, is at odds. As the world is incredibly complex, the search space for hypothesis (often denoted as $H$) is also *huge*. Therefore, assumptions often need to be made, which brings most, if not any, model somewhere in between the two extremes.

## Formalizing Learning

- Learning in Bayesian Networks can take a couple of forms:
    1. **Bayesian Learning**: Is done by computing the posterior $P(H|o)$ which uses the prior $P(h)$ and can be used in a weighted variant (utility) to determine the best 'action', $V(a) = \sum_h P(H|d)u(a,h)$
        - Bayesian learning is optimal, but untractable with realistic scenarios as the hypothesis space is often incredibly large.
    2. **Maximum Likelihood**: The most likely hypothesis, $h_{ML}$ will be $h_{ML} = \max_h P(d|h)$. Then we can select and action according to $V(a) = u(a, h_{ML})$. It is, however, prone to overfitting.

- Instead of computing the default maximum likelihood, we often compute the maximum **log likelihood**. As the logarithm is a monotonically increasing function, it should preserve the maxima we are looking for.
- For example, the equation $P(d|h_\theta) = \theta^c \cdot (1-\theta)^l$, where $c$ is the number of cherry candies, $l$ the number of lime candies and $\theta$ the probability of encountering the former, can be solved in a more 'computationally friendly' manner by taking the log, $L(d|h_\theta) = c\log\theta + l\log(1-\theta)$. We can then find the maximum by taking the derivative and finding 0, $\frac{dL(d|h_\theta)}{d\theta} = 0$. This will result in $\theta = \frac{c}{c+l}$, not very surprising.

3. **Maximum A Posteriori (MAP)**: Which is just the **ML**, but also taking into consideration the prior of the hypothesis: $h_{MAP} = \max_h P(d|h)P(h)$.
   - Here, the prior (just like in Bayesian Learning) penalizes the hypothesis being too complex. We regard more complex hypotheses being less likely.

- To be clear, this is distinct from **Most Likely Sequence Estimation** because there, we attempt to determine what happened, but the model itself doesn't necessarily improve. We learn about hidden states, but nothing about the model. This is my current hypothesis as of lecture 5

## The EM Algorithm

- Before, we only considered learning when we had access to data about *all* the relevant variables. Now, we consider learning not only the observable data, but also hidden or **latent variables**.
- To solve such learning problems, one can use the **Expectation Maximization** algorithm, or, **EM Algorithm**.
- Take the problem of learning **mixture of Gaussians** (we will later see how this is relevant to latent variables). That is, we have a **mixture distrbution**, $P$, with $k$ **components**:

$$P(x) = \sum_k P(C=i)P(x|C=i)$$

- The problem then is, *determine the center, $\mu_i$, of each Gaussian and the covariance, $\Sigma_i$, of each component, $i$.*
- The EM algorithm will provide exactly this and works according to a two-step process.
  1. **E-step**: Compute probabilities $p_{ij} = P(C=i|x_j)$, the probability that datum $x_j$ was generated by component $i$. By Bayes' rule, we have $p_{ij} = \alpha P(x_j|C=i)P(C=i)$. The term $P(x_j|C=i)$ is just the probabiltiy at $x_j$ for the $i$th Gaussian. Define $n_i = \sum_j p_{ij}$, the effective number of data points currently assigned to component $i$.
  2. **M-step**: Compute the new mean, covariance, and component weights using the following steps in sequence:

$$\mu_i \leftarrow \sum_j p_i j x_j / n_i$$
$$\Sigma_i \leftarrow \sum_j p_i j (x_j - \mu_i)(x_j - \mu_i)^T / n_i$$
$$w_i \leftarrow n_i / N$$

# Quantification of Objectives and Reward Hacking

- AI can be viewed as a model that is optimized according to a function that reflects 'intelligent behavior. (Backpropagation, Reinforcement Learning, Genetic Algorithms, etc.)

## Fitness and Quantification

- This 'fitness' has to be adjusted based on the problem at hand. For example,
  - Sum of squared errors for regression.
  - *Cross-entropy* for classification (i.e. difference between two probability distributions).
- Because we want to compute how our 'current' model compares to previous ones, we want the function to be quantifiable. That is, quantifiable in both input and output. (not quite the right use of quantifiable)
- Quantifying is hard because you embed assumptions about the problem you are trying to solve in the data you supply to the algorithm, and *the algorithm is only as good as the data you supply it with.*

## Reward Hacking

- **Reward Hacking**, or the **Inner Alignment** problem:

  The objective function admits some clever "easy" solution that formally maximizes it but perverts the spirit of the designer's intent (i.e. the objective function can be "gamed")

- **Goodhart's law** puts this behavior into perspective and reminds us that this is similar to human behavior:

  Any observed statistical regularity will tend to collapse once pressure is placed upon it for control purposes.

  When a measure becomes a target, is ceases to be a good measure.

- Several reasons could exist for an agent not displaying the expected/desired behavior that are associated with reward hacking:
  1. The objective function is wrong.
  2. The objective function is not properly evaluated (implementation or other practical issue besides the mathematical basis).
  3. The objective is 'correct', but the agent could not learn the 'correct behavior'.
- All these concepts might be responsible for agents acting out. It's important to consider that *someone* has to then take responsibility for these problems.

## Meaningful Human Control over AI

*Humans* not computers and their algorithms should ultimately remain in control of, and thus be *morally responsible* for relevant decisions.

- One approach to ensure humans can still bear responsibility to these systems, is to consider the **tracking** and **tracing** conditions:

  **Tracking**: The human-AI system (consisting of human agents, AI agents, infrastructures, etc.) should be able to track the relevant moral reasons of the relevant human agent(s) and be responsive to these reasons.

  **Tracing**: The system's behavior should be traceable to at least one human in the system design history or use context who can appreciate the capabilities of the system and their own role as morally responsible.

- Tracking and tracing can be further quantified into the following properties:
  1. The system has an explicit moral operational design domain (moral-ODD) and adheres to its boundaries.
  2. Humans and AI agents have appropriate and mutually compatible representations of each other and the task context.
  3. The agents' responsibility should be commensurate with their ability and authority to control the system.

# Utilities and Decision Theory

- Decision theory, in its simplest form, deals with choosing among actions based on the desirability of their immediate outcomes.
- In essence, an agent could work according to the principle of **maximum expected utility**. The expected utility is given according to eq. 10. And the agent would then take the action, $a$, which would result in the highest expected utility given some evidence $\mathbf{e}$.

$$\sum_s P(R(a) = s|a, \mathbf{e})U(s) \tag{10}$$

- In some sense, this is the most desirable any agent could do and could therefore be considered the foundation of AI. Even though the relation looks simple, it is essentially intractable for complex problems. The conditional probability requires a complete causal model of the environment, and $U$ requires perfect knowledge of the implications of one's actions which requires searching and planning.

## Utility Theory and Preferences

- To be able to make a decision, one needs to determine which outcome is better than another. We have to formalize the notion of **preferences**.
- Here, we use the notion of a **lottery** to indicate some undeterminable outcome based on a set of outcomes and their probabilities.
- We require the preferences to obey some constrains:
  - **Orderability**: Exactly one of $(A \succ B), (B \succ A), or (A \sim B)$ holds.
  - **Transitivity**: $(A \succ B) \wedge (B \succ C) \implies (A \succ C)$
  - **Continuity**: An agent should be indifferent to an outcome that is the average of two, or one that equals the average: $A \succ B \succ C \implies \exists_p : [p, A; 1-p, C] \sim B$
  - **Substutability**: I'm not quite sure what this is supposed to mean other than: Indefferences and preferences hold regardless of the complexity of a lottery: $A \sim B \implies [p, A; 1-p, C] \sim [p, B; 1-p, C]$.
  - **Monotonicity**: If a certain event has a higher probability of a preferred outcome, then the agent should choose that event: $A \succ B \implies (p > q \iff [p, A; 1-p, B] \succ [q, A; 1-q, B])$
  - **Decomposability**: Compound lotteries can be reduced to simpler ones. Also called the **no fun in gambling rule** as it shows that two consecutive lotteries can be compressed into a single equivalent lottery. The book contains a large mathematical relation, but I think the condition makes plenty of sense without it.
- Assembling our set of preferences, we can, from it determine a utility function. At least, we can approximate our known preferences with a utility function.

## Utility Functions

- Utilty functons are, for the most part, normalized. That is, there is a worst, and best-case scenario based on the values 0 and 1 respectively.
- If we consider a utility function $U(s)$ where $s$ is a particular state, then we can determine the expected utility by eq. 10 which can then be used to determine the action with the highest expected utility (see eq. 11). Which I absolutely refuse to call the *rational way to choose the best action*

$$a^* = max_a \mathbb{E}[U(a|\mathbf{e})] \tag{11}$$

- Here, a problem is that our estimation will be biased. We will not be able to perfectly model the underlying utility function and the error in each 'episode'[1] will therefore be non-zero. Still, if we assume this to be zero, i.e. **nonbiased**, the error will have some distribution based on the action taken.
- As we are 'optimistic' and take the best possible action, there is a tendency to pick samples with higher utility. Therefore, the error will slowly increase as the number of 'episodes' pass. This is called the **optimizer's curse**.
- The optimizer's curse can be avoided by using an explicit probability model for the approximated expected utility given the true expected utility: $P(\mathbb{E}U|\widehat{\mathbb{E}U})$.

## The Value of Information

- Becuase, generally, not all information is available to an agent, it has to 'ask questions' to its environment to gain more information.
- Therefore, one might wonder how to determine the best question to ask. One approach is consider the **value of perfect information** (see eq. 12).

$$VPI_{\mathrm{e}}(E_j) = \left( \sum_k P(E_j = e_{jk}|\mathbf{e}) \; EU(\alpha_{e_{jk}}|\mathbf{e}, E_j = e_{jk}) \right) - EU(\alpha|\mathbf{e}) \tag{12}$$

- In plain language, the formula (see eq. 12) works as follows: "if we know our expected utility at certain moment in time given some evidence, $\mathbf{e}$, what would some (possible) evidence(s), $E_j$, on average contribute to the expected utility?".

[1]With episode, I mean successive estimations. They use the word episodic in the book, and I think using it this way makes sense. The book uses the variable $k$ to express this.

# Planning Under Transition Uncertainty

- As we have seen, and will see, Markov models can be categorized according to the presence of hidden states, and whether the transitions can be chosen or not (see tbl. 3).

Table 3: Classification of different Markov models

| *Markov Models* | **Chosen transitions** | **No chosen transitions** |
|---|---|---|
| **No hidden states** | Markov Chain | Markov Decision Process (MDP) |
| **Hidden states** | Hidden Markov Model (HMM) | Partially Observable Markov Decision Process (MDP) |

## Sequential Decision Problems - Markov Decision Problems

- **Markov Decision Process (MPD)**: A fully observable, stochastic environment with a Markovian transition model, $P(s'|s,a)$ (only the current state has some relation with the next state), and additive rewards, according to a function $R(s)$.
- A set of actions might not provide a good solution as the problem is *stochastic*. The agent might take a wrong turn (note that the transition model is probabilistic), thus we give it a **policy**, $\pi$, that takes a state, $s$, and returns the next state: $s' = \pi(s)$.
- The optimal solution to an MPD is a policy, $\pi^*$, that returns the highest expected utility (remember, we cannot guarantee a sequence of events since they are stochastic).
- It's important to realize that this policy is learned and the quality of the policy (i.e. how well would it work in a real scenario) heavily depends on the quality of the reward function $R$.

### Utilities over Time

- An important consideration is whether there is a limited, finite, or infinite number of steps in which we can perform our (continued) task, i.e. a **finite-** or **infinite-horizon**. Consider the following.
- When studying the day before an exam, it is probably not very rewarding to spend a lot of time exactly what you're doing. You're better off learning past exams by hard until you fall asleep. If you would have kept up with the homework and studied incrementally, you can take your time to understand exactly what you're doing and the course will feel more rewarding (if you actually care about the subject, that is).
- If you would have an infinite amount of time, it does not matter what you do since you always have enough time to do anything.

- Finite horizon optimal solutions are **non-stationary**, while optimal solutions to infinite horizon problems are **stationary**.
- Of course, we could choose to simply add the utilities of the states in the sequence. Logically, however, it also makes sense to give lower reward to states further in the future. If we assume **stationary preferences**, our preferences stay the same over time, this implies exactly the two states above:
    1. **Additive Rewards**: $U_h([s_0, s_1, ...]) = R(s_0) + R(s_1) + ...$
    2. **Discounted Rewards**: $U_h([s_0, s_1, ...]) = \gamma^0 R(s_0) + \gamma^1 R(s_1) + ...$
- Discounted rewards therefore encapsulate additive rewards since they are equivalent for $\gamma = 1$.
- There are some considerations to be made when considering infinite horizons without terminal states, as it seems the reward could approach infinity. To this, there are three solutions:
    1. *Use discounted rewards.* Rewards are bound when they are discounted according to the following relation: $U(\mathbf{s}) = R_{max}/(1-\gamma)$, where $R_{max}$ is the maximum reward.
    2. *The assumption sucks and you have terminal states.* This, however, only holds if your agent is guaranteed to reach a terminal state at some point in time.
    3. *Take the average reward over a length of time.*

## Optimal Policies and the Utilities of States

- Now, for actually determining the optimal policy, we use an algorithm called **value iteration**.
- To explain this, we should first consider the **Bellman equation** (see eq. 13)

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a)U(s') \tag{13}$$

- This rephrases the utility of a policy from *summation of rewards* to the *utility of the current state plus the probabilistically weighted utility of the neighbor states.*
- To then discover the utilities of all $n$ states, we guess their initial probability and apply the **Bellman update** (see eq. 14) to iteratively approach the correct answer (see fig. 1).

$$U_{i+1}(s) \leftarrow \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a)U_i(s') \tag{14}$$

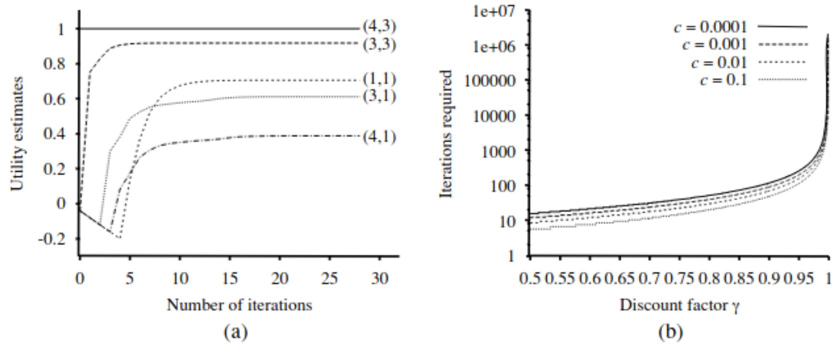


Figure 1: (a) Graph showing the evolution of the utilities of selected states using value iteration. (b) The number of value iterations $k$ required to guarantee an error of at most $\epsilon = c \cdot R_{max}$, for different values of c, as a function of the discount factor .

## Partially Observable MDPs

- Instead of assuming al the information is available to us (i.e. we can directly observe which state we're in), we consider when this is not the case. Thus, we have **Partially Observable MDP**.
- Here, we add a **sensor model**, $P(e|s)$ similar to the HMM, and a **belief state**, $b(s)$. Then we can determine the next belief state, $b'$ as follows,

$$b'(s') = \alpha\, P(e \mid s') \sum_{s} P(s' \mid s, a)\, b(s)$$

  The fundamental insight required to understand POMDPs is this: *the optimal action depends only on the agent's current belief state.* That is, the optimal policy can be described by a mapping $\pi^*(b)$ from belief states to actions.

- Since we do not know the current state, it would also no make sense to base the optimal policy on the true current state.
- In conclusion, it means that an update on each time step would look as follows:
  1. Given the current belief state, $b$, execute the action $a = \pi^*(b)$.
  2. Receive the perceived evidence $e$.
  3. Set the current belief state to Forward$(b, a, e)$.
  4. Go back to *1*.
- Now, the thing is that we don't know what evidence we will actually receive. Therefore, we should estimate what the possible evidence might be. Or, at least, have a method of determining how reliable our update method might be.
- Let's first return to the probability of finding evidence, $e$, with what we do know,

$$\begin{aligned} P(e|a,b) &= \sum_{s'} P(e|a,s',b)P(s'|a,b) \\ &= \sum_{s'} P(e|s')P(s'|a,b) \\ &= \sum_{s'} P(e|s') \sum_{s} P(s'|s,a)b(s) \end{aligned}$$

- Then, we can determine the probability of finding that particular next belief, $b'$, from the Forward function.

$$\begin{aligned} P(b' \mid b,a) = P(b' \mid a,b) &= \sum_{e} P(b' \mid e,a,b)P(e \mid a,b) \\ &= \sum_{e} P(b' \mid e,a,b) \sum_{s'} P(e \mid s') \sum_{s} P(s' \mid s,a)b(s) \end{aligned} \tag{15}$$

- Furthermore, we can define a reward function for beliefs:

$$\rho(b) = \sum_{s} b(s)R(s) \tag{16}$$

- In conclusion, if we consider eq. 15 the **belief transition** and eq. 16 the **belief reward**, we have essentially reduced the POMDP to a normal MDP.

## Value Iteration for POMDPs

- Value iteration can be described by the following function:

$$\alpha_p(s) = R(s) + \gamma \left( \sum_{s'} P(s' \mid s, a) \sum_{e} P(e \mid s') \alpha_{p,e}(s') \right) \tag{17}$$

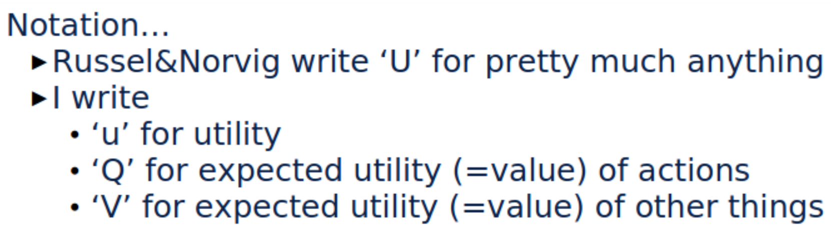
## Monte Carlo Tree Search

- An important observation is that up until one we have tried to determine the optimal policy performing any action, a so-called **offline planning** strategy.
- **Monte Carlo Tree Search (MCTS)** is an example of an **online planning** strategy and attempts to determine the best action to take by considering the transition function inferred by the action, $R(a) = s$, as a tree and stochastically exploring it.
- Long story short, MCTS works according to the following algorithm starting from the root of the tree, i.e. the current state:
  1. **Selection:** Select a child node using a selection strategy until a node with unexplored children is reached or a terminal state is encountered.
  2. **Expansion:** If the selected node has unexplored children (i.e., possible moves from the current state), expand the tree by adding one or more child nodes corresponding to these unexplored moves.
  3. **Simulation (Rollout):** Simulate a random or heuristic playout using a **rollout policy** from the newly added node (or from the selected node if it was already a terminal state) until a terminal state is reached. This is essentially a random sampling phase and represents a possible outcome of the game from the current position.
  4. **Backpropagation:** Update the statistics of the nodes along the path from the newly expanded or simulated node to the root. This typically involves updating the visit count and the total accumulated reward (or score) of the nodes.
  5. **Repeat:** Repeat steps 1 to 4 for a fixed number of iterations or until a time limit is reached.
  6. **Action Selection:** After the specified number of iterations, or when a computational budget is exhausted, choose the best move from the root node based on the node visit counts or other relevant statistics. This is usually the child node with the highest visit count, indicating it has been explored the most and has shown promising results.
- The rollout policy has an incredible effect on the performance of the MCTS algorithm. One could consider a MCTS algorithm to be essentially a *policy improvement operator*. That is, you give it a policy, and MCTS makes it better by applying additional search.
- Pros and cons:
  - (+) Rapidly zooms in on promising regions
  - (+) Can be used to improve policies
  - (+) Basis of many successful applications
  - (−) Needle in the hay-stack problems
  - (−) Problems with high branching factor

# Reinforcement Learning

The general idea of reinforcement learning comes down to the fact that we don't know exactly what the agent should do, but we do know when it has done something right. While we don't know *how* to win a chess game, we do know the goal is to win. Therefore, **Reinforcement Learning (RL)** concerns itself with learning based on **rewards**.

- We will specifically consider 3 designs:
  1. A **utility-base** agent that learns a utility function based on the states and from there behaves to maximize its utility. This seems similar to the utility of a state from the MDP chapter.
  2. A **Q-Learning** agent that learns an **action-utility**, or **Q-function**, giving the expected utility of taking an action in a certain state.
  3. A **reflex agent** that learns a policy that maps directly from states to actions. In MDPs from the previous chapter this was the first introduced technique I think
- Generally, there are two variants: **passive** learning, where a policy is fixed and an agent learns the utilities of the states; and **active** where is learns the utilities as well as the policy.
- In all cases, the primary problem is for the agent to **explore** as much of the domain. The agent needs to have had an adequate number of experiences to take sufficiently informed actions. That is, an agent should refrain from **beunen**.



Notation...
- Russel&Norvig write 'U' for pretty much anything
- I write
  - 'u' for utility
  - 'Q' for expected utility (=value) of actions
  - 'V' for expected utility (=value) of other things

Figure 2: I guess this might be important for the exam

- There is also a distinction to be made between **model-free** and **model-based** reinforcement learning.
  - **Model-free** learning captures learning methods which do not take into account the underlying transition and reward functions, $P$ and $R$ resp, of the underlying MDP.
  - **Model-based** RL attempts to populate these functions with learned values or distributions.

## Model-Free Reinforcement Learning

### Passive Learning

- The most simple example is that of a passive, model-free agent in a state-based representation where are the environment is fully observable.

- The policy is fixed, so we will determine the utility of the policy.
- The big difference being a lack of a **transition model**, $P(s'\bar{s}, a)$, and **reward function**, $R(s)$.
- The first method is **direct utility estimation** (aka **Monte Carlo prediction**) which basically boils down to executing the policy many times starting from a new state each time. From this, we get an expected reward that represents the utiltiy of the state.
- It is **unbiased**, meaning it will converge to the right solution, but has high **variance**, meaning it will take a lot of samples to get there.
- A problem with this is that is doesn't take into account that the utility of the current state is dependent on that of the previous according to the bellman equation.
- An alternative is to use **temporal-difference** learning which expresses the utility of the 'current' state into that in a difference between it and successive states. Formally, it uses the following relation to update the utility of the current state, given the current policy.

$$U^{\pi}(s) \leftarrow U^{\pi}(s) + \alpha(R(s) + \gamma U^{\pi}(s') - U^{\pi}(s)) \tag{18}$$

  Here, $\alpha$ is the **learning rate**.
- It could be considered a form of **stochastic gradient descent (SGD)**, but not quite. Not exactly sure why not quite. It's not very clear from the slides.
- It's important to note that this method is **biased** due to **bootstrapping**, but has **lower variance** than the previous method. The **bootstrapping** comes from the assumption in eq. 18 that this holds even for non-final $U^{\pi}(s)$.

## Active Learning

- We know how to learn $U^{\pi}$, but we really want to know how to learn $U^{*}$. Therefore, we have to learn a policy.
- A very simple approach would be to adapt eq. 18 to use $U(s, a)$ instead:

$$U^{\pi}(s, a) \leftarrow U^{\pi}(s, a) + \alpha(R(s, a) + \gamma \max_{a'} U^{\pi}(s', a') - U^{\pi}(s, a)) \tag{19}$$

  The expected utility of a state-action pair is also often denoted as $Q$. Therefore, this approach is also called **Q-Learning**. Note that we are taking $a'$ with that returns the maximum utility.
- **State-action-reward-state-action (SARSA)** is an alternative to the prior where we don't take the max action. We then essentially determine the expected utility of a policy. We can then try to optimize such that $\pi \rightarrow \pi^{*}$.

## Scaling Up

- Up until now, we have solely worked in a state-action based system which is essentially represented by a lookup table. Performing the previous algorithms on large state-spaces, however, quickly becomes impractical.
- An alternative to such a tabular approach would be **function approximation**. That is, use any sort of function to approximate the model. One such example would be to use a neural network.
- It's not 'true' gradient descent so only a few of the nice properties associated with it remain (at least for linear functions):
  - TD learning: **converges** to bounded approximation.
  - SARSA: might **chatter** (cycle around the optimal solution) but not diverge.

  - Q-learning: can **diverge**.

- The last property can be attributed to the **deadly triad of RL**:
  1. bootstrapping,
  2. off-policy learning, and
  3. function approximation

## Policy Search

- **Policy search** is very simple: just keep changing the policy until it improves, then stop. An example would be the following:

$$\pi(s) = \max_a Q_\theta(s, a)$$

  The idea is then that policy search would adjust the parameters, $\theta$, such that the policy improves.

- The policy could be improved by determining the **policy gradient** from the **policy value** and applying a hill-climbing algorithm.
- This could be combined with a value function that evaluates the current policy, in something called an **actor-critic method**. This reduces high variance otherwise associated with policy gradient methods.

# Model-Based Reinforcement Learning