Artificial Intelligence

Billy Fang

Fall 2013

The following is a collection of notes on artificial intelligence; they are heavily based on Russell and Norvig's text [1].

Contents

1 Search **2**
- 1.1 Uninformed search strategies . . . 2
- 1.2 Informed (heuristic) search strategies . . . 3
- 1.3 Heuristic functions . . . 4
- 1.4 Local search and optimization . . . 4
- 1.5 Adversarial search and games . . . 5

2 Logical agents **6**
- 2.1 Propositional logic . . . 6
- 2.2 Propositional theorem proving . . . 7
- 2.3 Resolution . . . 8
- 2.4 Effective propositional model checking . . . 9
- 2.5 Local search . . . 10

3 Reasoning with uncertainty **10**
- 3.1 Bayesian networks . . . 11
- 3.2 Exact inference in Bayesian networks . . . 12
- 3.3 Approximate inference in Bayesian networks . . . 12

4 Uncertainty over time **13**
- 4.1 Inference in temporal models . . . 14
- 4.2 Hidden Markov Models . . . 16
- 4.3 Kalman filters . . . 16
- 4.4 Dynamic Bayesian networks . . . 17

5 Decisions under uncertainty **18**
- 5.1 Value iteration . . . 19
- 5.2 Proving correctness . . . 19
- 5.3 Policy iteration . . . 20
- 5.4 Partially observable MDPs . . . 20

6 Learning **21**
- 6.1 Supervised learning . . . 21
- 6.2 The theory of learning . . . 21
- 6.3 Decision trees . . . 22
- 6.4 Linear classification . . . 23
- 6.5 Support vector machines . . . 24
- 6.6 Ensemble models . . . 25

6.7 Unsupervised learning . . . 26
6.8 Learning in Bayesian networks . . . 26
6.9 Learning hidden Markov models . . . 27
6.10 Reinforcement learning in MDPs . . . 27

1 Search

Intelligent agents **perceive** and **act** on an **environment** to achieve a **goal**. Their actions have **costs**. We can formulate/model a problem using using a **state space**, where actions are transitions between states. A **path** is a sequence of actions, and a **path cost** is the sum of the costs of the actions on the path.

In tree search, we pick a node of the frontier, expand it, and add its successors to the frontier. A drawback of tree search is that a state may be visited multiple times. This motivates keeping an explored set. In graph search, the frontier separates the explored set from the unexplored set.

An algorithm is **complete** if it is guaranteed to find a solution when one exists. An algorithm is **optimal** if the solution it returns has the lowest possible path cost. We should also consider **time complexity** and **space complexity**.

1.1 Uninformed search strategies

Uninformed search (or **blind search**) means that the strategies can only generate successors and check if a state is a goal state.

Breadth-first search expands nodes based on proximity to the root node. In tree search, it expands each level exhaustively, while in graph search it expands by widening radius. This is done by storing the frontier as a queue. It is complete if the branching factor is finite. It is optimal if the path cost is a nondecreasing function of the depth of the node (for example, all actions having the same cost). The number of nodes generated (time complexity) for a b-ary tree when the solution is at depth d is $b + \cdots + b^d = O(b^d)$ if the algorithm applies the goal test on a node when generated. If the goal test is applied when selected for expansion, then the time complexity would be $O(b^{d+1})$. There will be $O(b^{d-1})$ nodes in the explored set and $O(b^d)$ nodes in the frontier, so the space complexity is $O(b^d)$. The time and space complexity of BFS are very bad, although in practice the space complexity is a bigger problem.

Uniform-cost search expands node with the lowest path cost $g(n)$, implemented by storing the frontier as a priority queue. The goal test must be applied when a node is selected for expansion rather than when it is first generated because (a) the first goal node generated may be on a suboptimal path, and (b) there may be a better path found to a node currently on the frontier. UCS is clearly optimal. Completeness follows as well, provided that the costs are $> \epsilon$ (possible to get stuck in an infinite sequence of zero-cost actions). Let C^* be the cost of the optimal solution, and assume every action has cost $> \epsilon$. The time and space complexities are $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ which can be much greater than b^d, since UCS may get bogged down in low-cost useless steps before looking at large-cost useful steps. When path costs are equal, UCS is almost identical to BFS.

Depth-first search expands the last node added to the frontier (stack or a priority queue with depth as a priority). In graph search, DFS is complete in finite spaces; in tree search, DFS is not complete. In all cases, it is nonoptimal. In graph search, the time complexity is bounded by the size of the state space (possibly infinite). In tree search, it could generate all $O(b^m)$ nodes of the search tree (where m is the maximum depth, possibly infinite), which can be much larger than the size of the state space. Note that m can be larger than d, the depth of the best solution. DFS's advantage is in saving space; it needs only store a single path along with unexpanded sibling nodes along the path. Thus the space complexity is $O(bm)$ for tree search. In graph search, the explored set still takes up space, so it possibly may need to remember everything: $O(b^m)$.

Depth-limited search treats nodes some given limit ℓ as having no successors. It is incomplete if $\ell < d$; it is nonoptimal in general (but is optimal when $d = \ell$). Its time complexity is $b + b^2 + \cdots + b^\ell = O(b^\ell)$, and its space complexity is $O(b\ell)$ for tree search and $O(b^\ell)$ for graph search.

Iterative deepening depth-first search performs DLS for $\ell = 0, 1, 2, \ldots$ in that order, until a goal is found. Its space complexity is $O(bd)$ for tree search and $O(b^d)$ for graph search. It is complete when

the branching factor is finite and optimal when the path cost is a nondecreasing function of the depth of the node. It may seem wasteful to go over the same nodes in the top level every time, but it is not so bad because most of the nodes are at the bottom level. The bottom level is generated once, the next-to-bottom level is generated twice, and so on, so the time complexity is $(d+1) + db + \cdots 2b^{d-1} + b^d = O(b^d)$.

Bidirectional search requires reversible actions, and checks whether the frontiers of the forward and backward searches meet. The motivation is that $b^{d/2} + b^{d/2}$ is much smaller than b^d.

Criterion	BFS	UCS	DFS	DLS	IDDFS	Bidirectional
Complete	Y	Y*	N	N	Y*	Y*
Optimal	Y*	Y	N	N	Y*	Y*
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	state space or $O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$ or $O(b^m)$	$O(b\ell)$ or $O(b^\ell)$	$O(bd)$ or $O(b^d)$	$O(b^{d/2})$

1.2 Informed (heuristic) search strategies

In **informed search**, we use problem-specific knowledge beyond the definition of the problem itself to find solutions more efficiently. The general approach is **best-first search**, which is an instance of tree search or graph search in which an **evaluation function** $f(n)$ determines which node is selected for expansion. The evaluation function acts as a cost estimate, so the node with the lowest evaluation is expanded first. This is identical to UCS, except f is used instead of g.

Most algorithms include a **heuristic function** $h(n)$ as a component when defining f. The heuristic function is the estimated cost of the cheapest path from n to a goal state.

Greedy best-first search is the case where $f(n) := h(n)$. Due to the greediness, it is not optimal. It is incomplete (except for graph search in finite spaces).

A* search uses the evaluation function $f(n) := g(n) + h(n)$.

A heuristic is **admissible** if it never overestimates the cost to reach the goal. As a result, $f(n) = g(n) + h(n)$ never overestimates the true cost of a solution going through the current path to n. We impose the conditions that h must be nonnegative and $h(n) = 0$ if n is a goal node.

A heuristic is **consistent** or **monotonic** if $h(n) \leq c(n, a, n') + h(n')$. A consistent heuristic is admissible.

Theorem 1.1. *Tree search A* is optimal if $h(n)$ is admissible.*

Proof. Suppose n is a goal state, and suppose for sake of contradiction that n' is another goal state. Somewhere along the path to n', there must have been a node n'' that was not expanded. Noting that $h(n) = h(n') = 0$ because they are goal nodes, we have

$$g(n) = f(n) \overset{\text{A*}}{\leq} f(n'') = g(n'') + h(n'') \overset{\text{admiss.}}{\leq} g(n'') + c(n'' \to n') = g(n').$$

□

The proof in the case of graph search is a result of the following lemmas.

Lemma 1.2. *If $h(n)$ is consistent, then the values of $f(n)$ along any path are nondecreasing.*

Proof. $f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$ □

Lemma 1.3. *Whenever graph search A* selects a node n for expansion, the optimal path to that node has been found.*

Proof. If not, then there would have to be another frontier node n' on the true optimal path to n, by the graph separation property. Since f is nondecreasing along the subpath $n' \to n$ of the optimal path, it must be that $f(n') \leq f(n' \to n) < f(n)$ so that n' should have been chosen by A* at this point rather than n, a contradiction. □

Theorem 1.4. *Graph search A* is optimal if $h(n)$ is consistent.*

Proof. For graph search, A* expands nodes in nondecreasing order of $f(n)$; thus, the first goal node chosen for expansion will have path cost $g(n) = f(n)$ less than any later goal node. □

We may draw **contours** for f in the state space. If $C*$ is the cost of the optimal solution path, then A* expands all nodes with $f(n) < C*$, some nodes with $f(n) = C*$, and no nodes with $f(n) > C*$. Completeness thus requires that there be finitely many nodes with cost at most $C*$, which is true if $b < \infty$ and all step costs are $> \epsilon$.

A* is **optimally efficient** for any given consistent heuristic: no other optimal algorithm is guaranteed to expand fewer nodes than A* except possibly through tie-breaking among nodes with $f(n) = C*$. This is becuase any algorithm that does not expand all nodes with $f(n) < C*$ runs the risk of missing the optimal solution.

In practice, the number of states within the goal contour can be exponential in the length of the solution; memory runs out. One solution is **iterative-deepening A***, where the cutoff value is the smallest f-cost of any node that exceeded the cutoff on the previous iteration. Another is **recursive best-first search**, where the second-best alternative is stored and invoked when the current value of f exceeds some limit. **SMA***expands the best leaf until memory is full, and then drops the worst leaf node. It also backs up the value of the forgotten node in its parent, so that despite not knowing which way to go from the parent, we still have an idea of how worthwhile it is to go anywhere from n.

1.3 Heuristic functions

A heuristic h_2 **dominates** h_1 if $h_2(n) \geq h_1(n)$.

Proposition 1.5. *If h_2 dominates h_1, then A* using h_2 will never expand more nodes than A* using h_1.*

Proof. Every node with $h(n) < C * -g(n)$ will be expanded. So any node expanded by h_2 will be expanded by h_1. □

It is generally better to use a heuristic with higher values.

One way to get heuristic functions is to **relax** the problem. (Manhattan distance, straight-line distance).

Given admissible heuristics $h_1, \ldots, h_m$, the composite heuristic $h(n) := \max\{h_1(n), \ldots, h_m(n)\}$ dominates all of them and is also admissible.

Another way is to use the solution cost of a **subproblem** (e.g. in 8-puzzle, getting 1,2,3 into place). Then we can use **pattern databases** to store these solution costs for every possible instance of the subproblem, and combine it with other pattern databases.

1.4 Local search and optimization

In some cases, we are not concerned with the path to the solution. **Local search** algorithms use a single current node, rather than multiple paths, and move to neighbors. Although they are not systematic, their key advantages are lower memory and the ability to find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable. They are also useful to solve pure **optimization problems** which finds the best state according to an **objective function**; these problems often do not fit the standard search model. For example, reproductive fitness could be an objective function for evolution, but there is no goal test nor path cost.

We consider the **state-space landscape**, a plot of the objective function vs states. Our aim is to find a **global maximum or minimum**. A **complete** local search algorithm always finds a goal if one exists, while an **optimal** one always finds the global min or max.

The **hill-climbing** or **steepest-ascent** version simply moves in the direction of the best neighbor, forgetting the past and not looking any farther into the future. It stops upon reaching a peak, where no neighbors have higher value. It gets stuck at **local maxima** and **plateaux**, and has difficulty navigating **ridges** (a sequence of local maxima not directly connected with each other). If we allow **sideways moves**, we may be able to exit a plateau, but run the risk of staying in an infinite loop; to combat this, we may put a limit on the number of sideways moves. **Stochastic hill climbing** chooses a neighbor at random (or with probability weighted with the steepness). **First-choice hill climbing** is an example; it generates successors randomly and chooses the first one that is better than the current state. **Random-restart hill**

climbing performs hill climbing from different randomly chosen initial states. If each hill-climbing search has a probability p of success, then the expected number of restarts required is $1/p$.

Simulated annealing can be compared to the task of getting a ping-pong ball into the lowest crevice in a bumpy surface. (Here, we switch to **gradient descent**, i.e., minimizing cost instead of maximizing the objective function.) If we let the ball roll, it will stop at a local minimum, but if we shake the surface, it can pop out of the local minimum, but we must be careful to shake too hard and dislodge it from the global minimum. This is implemented by choosing a random neighbor, and accepting it if it improves the situation, or accepting it with some probability if it worsens the situation.

Local beam search keeps track of k states rather than just one; at each point it chooses the k best successors. It is not the same as k random restarts because it aggregates information from all k nodes; it can gravitate the search to a certain part of the space. Local beam search can suffer if the k states converge on the same region; a variant called **stochastic beam search** alleviates the problem by making the probability of choosing a givn successor as an increasing function of its value.

A **genetic algorithm** is a variant of stochastic beam search in which the successor states are generated by combining two parent states rather than by modifying a single state. A **fitness** function rates the goodness of a state. Given a randomly selected population of k states, two states are chosen (with probability corresponding to their fitness) to be parents. They create a child state that is a combination of the parents in some sense. Finally, the child state is given a mutation with small probability.

1.5 Adversarial search and games

We consider deterministic, turn-taking, two-player, **zero-sum games** of perfect information, like chess. A **utility function** defines the final numeric value for a game that ends in a terminal state. A zero-sum game is such that the total payoff to all players is the same (not necessarily zero) for any instance of the game. We can represent the sequences of moves as a search tree.

We take the role of MAX playing against MIN. The **minimax value** of a node is the utility of being in the state, assuming that both players play optimally from there until the end of the game. The minimax value of a terminal state is its utility. Otherwise, it is the maximum (resp. minimum) minimax value of all possible moves that MAX (resp. MIN) could make.

$$\text{Minimax}(s) = \begin{cases} \text{Utility}(s) & \text{if } s \text{ is a terminal node} \\ \max_{a \in \text{Actions}(s)} \text{Minimax}(\text{Result}(s,a)) & \text{if it is } \textsc{max}\text{'s turn'} \\ \min_{a \in \text{Actions}(s)} \text{Minimax}(\text{Result}(s,a)) & \text{if it is } \textsc{min}\text{'s turn'} \end{cases}$$

The **minimax algorithm** computes the best decision recursively, backing up the minimax values of each state in the tree.

To generalize to games with more than two players, the utility can be instead a k-tuple, with each player trying to maximize a certain component of the tuple.

Alpha-beta pruning is a way to avoid searching the entire tree. At all nodes, we keep track of the interval that its utility lies in. At first, we initialize the intervals to $[-\infty, +\infty]$, but with more information, we can update. For example, if a MIN node sees that one of its children has value 3, then we can update the range of the MIN node to $[-\infty, 3]$. Armed with these intervals, we can avoid searching parts of the tree if we know that it is hopeless. α is the value of best (highest) choice at any choice point along the path for MAX, while β is the value of the best (lowest) choice along the path for MIN. We want large α and small β to prune more.

The effectiveness of this pruning depends highly on the order in which the states are examined. This can reduce the time complexity from $O(b^m)$ (minimax) to $O(b^{m/2})$ (alpha-beta).

It may be useful to store the utility of previously seen positions: **transposition table**.

Instead of calculating the utility exactly, we can use a heuristic **evaluation function** instead to estimate the utility. We determine a **cut off** point for the search, at which we call the evaluation function instead of calculating utility. This may be problematic, as something very bad can happen just beyond the cutoff. We should apply the evaluation function only to positions that are **quiescent** (unlikely to exhibit wild swings in the near future).

Algorithm 1 Alpha-beta search

```
function ALPHA-BETA(state)
    v ← Max-value(state, −∞, +∞)
    return action with value v
```

```
function MAX-VALUE(state, α, β)
    if terminal state then
        return utility
    v ← −∞
    for all action a do
        v ← max(v, Min-value(Result(s, a), α, β))
        if v ≥ β then return v
        α ← max(α, v)
```

```
function MAX-VALUE(state, α, β)
    if terminal state then
        return utility
    v ← +∞
    for all action a do
        v ← min(v, Max-value(Result(s, a), α, β))
        if v ≤ α then return v
        β ← max(β, v)
```

Another problem is the **horizon effect**, in which an opponent's move causes serious damage and is inevitable, but can be temporarily avoided. This is problematic because the algorithm could push the bad result beyond the cutoff point.

One way to mitigate the horizon effect is the **singular extension**, a move that is clearly better than all other moves in a given position. If found before, the singular move is saved, and when the search reaches the depth limit, it considers the singular move if it is legal.

In games such as chess, it is very useful to use **lookup tables** for the opening and the endgame instead of search.

2 Logical agents

A **knowledge base** is a set of **sentences**. The sentences are expressed according to the **syntax** of the representation language, which specifies all the sentences that are [defined to be] well-formed. A logic must define the **semantics** (meaning) of sentences. The semantics defines the **truth** of each sentence with respect to each **model** (possible world). If a sentence α is true in model m, we say m satisfies α. We write $\alpha \models \beta$ if sentence α entails sentence β, that is, $\alpha \models \beta$ if and only if in ever ymodel in which α is true, β is also true.

Logical inference is the process of deriving conclusions from some knowledge base. One way is **model checking**, which checks all possible models. We write $KB \vdash_i \alpha$ if an inference algorithm i can derive α from KB. An inference algorithm is **sound** if it derives only entailed sentences. An inference algorithm is **complete** if it can derive any sentence that is entailed.

2.1 Propositional logic

We now present a powerful logic called **propositional logic**. The **atomic sentences** consist of a single **proposition symbol**, which stands for a proposition that can be true or false. **Complex sentences** are constructed from simpler sentences, using parentheses and **logical connectives**, such are

- $\neg$ (not). A **literal** is either an atomic sentence (**positive literal**) or a negated atomic sentence (**negative literal**).

- $\wedge$ (and). A sentence whose main connective is $\wedge$ is called a **conjunction**.
- $\vee$ (or). A sentence whose main connective is $\vee$ is a **disjunction**.
- $\Rightarrow$ (implies).
- $\Leftrightarrow$ (if and only if).

We now specify its semantics, which must specify how to compute the truth value of any sentence, given a model. Defining $True$ to be true in every model, and $False$ to be true in every model, we can express the semantics in a **truth table**.

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
F	F	T	F	F	T	T
F	T	T	F	T	T	F
T	F	F	F	T	F	F
T	T	F	T	T	T	T

2.2 Propositional theorem proving

One simple inference procedure is to check all assignments to the propositional symbols. This is not desirable because the time complexity is $O(2^n)$ where n isthe number of symbols.

Two sentences α and β are logically equivalent (denoted $\alpha \equiv \beta$) if they are true in the same set of models. Moreover, $\alpha \equiv \beta$ if and only if $\alpha \models \beta$ and $\beta \models \alpha$.

We list some equivalences.

$$\alpha \Rightarrow \beta \equiv \neg\beta \Rightarrow \neg\alpha \quad \text{(contraposition)}$$

$$\alpha \Rightarrow \beta \equiv \neg\alpha \vee \beta \quad \text{(implication elimination)}$$

$$\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta \quad \text{(De Morgan)}$$

$$\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta \quad \text{(De Morgan)}$$

A sentence is **valid** if it is true in all models; such a sentence is called a **tautology**.

Theorem 2.1 (Deduction theorem). *$\alpha \models \beta$ if and only if the sentence $(\alpha \Rightarrow \beta)$ is valid.*

Proof. Simply check all four possible models (truth tables). □

A sentence is **satisfiable** if it is true in some model. The problem of determining the satisfiability of sentences in propositional logic (**SAT**) is NP-complete.

Theorem 2.2. *α is valid if and only if $\neg\alpha$ is unsatisfiable.*

Theorem 2.3 (Proof by contradiction). *$\alpha \models \beta$ if and only if the sentence $(\alpha \wedge \neg\beta)$ is unsatisfiable.*

We now discuss **inference rules**.

$$\frac{\alpha \Rightarrow \beta, \alpha}{\beta} \quad \text{(Modus Ponens)}$$

$$\frac{\alpha \wedge \beta}{\alpha} \quad \text{(And-Elimination)}$$

$$\frac{\alpha, \beta}{\alpha \vee \beta} \quad \text{(Reverse And-Elimination)}$$

Any of the logical equivalences can also be used as inference rules.

Theorem proving determines $KB \models \alpha$ using syntactic manipulation (inference). It is sound, but its completeness depends on which inference rules are used. Theorem proving can be more efficient than model checking because it can ignore irrelevant propositions, no matter how many of them there are.

This is essentially a search problem, where the goal is to find sentence α while deriving entailed sentences.

2.3 Resolution

An algorithm is **sound** if returning yes implies that $KB \models \alpha$. It is **complete** if it returns yes whenever $KB \models \alpha$. It turns out that **resolution** is enough to ensure completeness.

The resolution steps resolve **complementary literals**.

$$\frac{\ell_1 \vee \cdots \vee \ell_k, n}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k} \text{ where } m \text{ and } \ell_i \text{ are complementary literals} \qquad \text{(unit resolution)}$$

A small technical point: $A \vee A$ should be reduced to A.

A sentence is in **conjunctive normal form** if it is a conjunction of **clauses**, where each clause is a disjunction of literals.

We can convert any sentence of propositional logic to CNF.

1. Replace $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.
2. Replace $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$.
3. Use De Morgan's law to push $\neg$ inside parentheses.
4. Apply distributivity of $\vee$ over $\wedge$.

To show $KB \models \alpha$, we show that $KB \wedge \neg\alpha$ is unsatisfiable. We convert $KB \wedge \neg\alpha$ into CNF, and resolve complementary literals, adding the new clause to the set if not already there. The process continues until

- there are no new clauses that can be added, in which case $KB \not\models \alpha$, or
- two clauses resolve to yield the **empty clause** (contradiction), in which case $KB \models \alpha$.

The following theorem demonstrates the completeness of the resolution algorithm.

Theorem 2.4 (Ground resolution theorem)**.** *If a set of clauses is unsatisfiable, then the resolution closure of those clauses contains the empty clause.*

Proof. We define the **resolution closure** $RC(S)$ of a set of clauses $S := \{P_1, \ldots, P_k\}$ to be the set of clauses derivable by repeated applications of the resolution rule to clauses in S or their derivatives. We will show the contrapositive: if $\varnothing \notin RC(S)$, then S is satisfiable under the model defined in the following way:

- If a clause in $RC(S)$ contains the literal $\neg P_i$ and all its other literals are false under the assignment chosen for $P_1, \ldots, P_{i-1}$, then assign $false$ to P_i.
- Otherwise, assign $true$ to P_i.

This model satisfies S. Suppose otherwise, that at some [earliest] stage i, assigning P_i causes some clause C to become false. This means that after assigning $P_1, \ldots P_{i-1}$, the clause C is either $false \vee false \vee \cdots \vee false \vee P_i$ or $false \vee false \vee \cdots \vee false \vee \neg P_i$. If only one of these clauses is in the set, then the algorithm would have assigned the appropriate value to P_i to make the clause true, so it must be that both of these clauses are in the set. Since $RC(S)$ is closed under resolution, the resolvent of these two clauses would be in $RC(S)$ and would already have been falsified by the assignments $P_1, \ldots, P_{i-1}$, contradicting the claim the P_i was the first assignment to falsify a clause. □

A **definite clause** is a disjunction of literals of which exactly one is positive. A **Horn clause** is a disjunction of literals of which at most one is positive. All definite clauses are Horn clauses. **Goal clauses** (clauses with no positive literals) are also Horn clauses. Horn clauses are closed under resolution: the resolvent of two Horn clauses is also a Horn clause.

Definite clauses can be written as implications:

$$(\neg P_1 \vee \neg P_2 \vee \cdots \vee \neg P_{n-1} \vee P_n) \to \left(\left(\bigwedge_{i=1}^{n-1} P_i \right) \Rightarrow P_n \right)$$

A single positive literal is called a **fact**.

Inference with Horn clauses can be done through **forward chaining** and **backward chaining**. Deciding entailment with Horn clauses can be done in time linear in the size of the knowledge base.

Forward chaining determines if a single proposition symbol q, the query, is entailed by a knowledge base of definite clauses. It begins from known facts (positive literals) in the knowledge base. If all the premises of an implication are known, then its conclusion is added to the set of known facts. It can be depicted as a graph, where premises point to implications. Forward chaining is sound (each inference is Modus Ponens). It is also complete, in that every entailed atomic sentence will be derived: after no new inferences are possible, let all inferred symbols be $true$ and all others $false$. This model satisfies every definite clause in the original knowledge base. (Suppose otherwise, then some clause $a_1 \wedge \cdots \wedge a_k \Rightarrow b$ is false, meaning that the premise is true but b is false, contradicting the fact that there are no new possible inferences).

Backward chaining begins with the query, and goes backward until it can find a series of implications whose premises are known.

2.4 Effective propositional model checking

The **Davis-Putnam-Logemann-Loveland algorithm** takes sentences in CNF and performs a recursive depth-first enumeration of possible models.

- Early termination: The algorithm detects whether a sentence is true or false even with a partial model. A clause is true if any literal is true.
- Pure symbol heuristic: A **pure symbol** is a symbol that always appears with the same sign in all clauses. If there exists a model for the sentence, assigning the pure symbol to be $true$ if it is a positive literal (or $false$ if it is a negative literal) will never make a clause false.
- Unit clause heuristic: A **unit clause** has all literals assigned to false except one, which forces an assignment for the unassigned symbol. This in turn can force other assignments: **unit propagation**.

Algorithm 2 DPLL algorithm

function DPLL(clauses, symbols, model)
  if every clause in *clauses* is true in *model* **then**
    return $true$
  if some clause in *clauses* is false in *model* **then**
    return $false$
  $P, value \leftarrow$ Find-Pure-Symbol($symbols, clauses, model$)
  if P is non-null **then**
    return DPLL($clauses, symbols - P, model \cup \{P = value\}$)
  $P, value \leftarrow$ Find-Unit-Clause($symbols, clauses, model$)
  if P is non-null **then**
    return DPLL($clauses, symbols - P, model \cup \{P = value\}$)
  $P \leftarrow$ First($symbols$)
  $rest \leftarrow$ Rest($symbols$)
  return DPLL($clauses, rest, model \cup \{P = true\}$) **or** DPLL($clauses, rest, model \cup \{P = false\}$)

Some improvements:

- Component analysis: if clauses become separated into disjoint subsets after assigning some variables, it is more efficient to work on each component separately.
- Variable and value ordering: the **degree heuristic** suggests choosing the variable that appears more frequently over all remaining clauses.
- Intelligent backtracking: backing up to the relevant point of conflict can be a significant improvement. In **conflict clause learning**, conflicts are stored as clauses so that they will not be repeated later.

- Random restarts
- Clever indexing: for example, indexing things like "the set of clauses in which X_i appears as a positive literal."

In conflict clause learning, we often build an **implication graph** (similar to the one in forward chaining), and split it into a "conflict" side and a "reason" side. A **unique implication point** is on every path from the last assignment to the conflict. There are various methods of splitting the implciation graph.

- 1-UIP: cut just after the implication point closest to the conflict.
- Rel Sat, aka Last-UIP: cut right before the farthest implication point (the decision variable)
- First new clause: cut close to conflict clause (nothing to do with implication points)

Conflict clause learning exhibit **heavy-tail** phenomena: it is fast most of the time, but there is a non-negligible probability that it is very slow. This can be mitigated with random restarts.

There is also a **backdoor** phenomena: after setting a few certain variables, everything "falls into place," but the question is how to find these variables.

2.5 Local search

WalkSAT begins with a random model. At every iteration, it chooses an unsatisfied clause and picks a symbol in the clause to flip. It chooses between two methods:

- flip the symbol that minimizes the number of unsatisfied clauses in the result
- pick a symbol randomly

If WalkSAT does not find a solution, it may need more time. Thus, WalkSAT is most useful when we expect a solution to exist. However, it cannot prove unsatisfiability.

An **underconstrained** problem has many solutions in the space of assignments (few clauses relative to number of variables). On the other hand, an **overconstrained** problem has many clauses relative to the number of variables, and is likely to have no solutions. Letting m be the number of clauses, n be the number of variables, and k be the number of literals per clause. We denote the set of all such clauses as $CNF_k(m, n)$. For small m/n, the probability of satisfiability is close to 1, and for large m/n it is close to 0. The probability drops sharply at some point. The **satisfiability threshold conjecture** states that for every $k \geq 3$, there is a threshold ratio r_k such that as m goes to infinity, the probability that $CNF_k(m, rm)$ is satisfiable becomes 1 for all $r < r_k$, and 0 for all $r > r_k$. It is unproven.

3 Reasoning with uncertainty

We omit the discussion of the basics of probability.

Marginalization:

$$P(Y = y) = \sum_z P(Y = y, Z = z)$$

Conditioning:

$$P(Y = y) = \sum_z P(Y = y \mid Z = z)P(Z = z)$$

We see that $P(X = x \mid e) = \alpha P(X = x, e)$ where $\alpha = 1/P(e)$ and does not depend on the value of x. Thus, to find the probability distribution of $P(X|e)$, it suffices to normalize the vector for $P(X, e)$.

X and Y are **conditionally independent** given Z if

$$P(X, Y \mid Z) = P(X \mid Z)P(Y \mid Z).$$

Using absolute or conditional independence, we can decompose a large joint distribution into smaller distributions.

The **naive Bayes** model is a model where a single cause directly influences a number of effects, all of which are conditionally independent given the cause. Its joint distribution can be written

$$P(Cause, Effect_1, \ldots, Effect_n) = P(Cause) \prod_i P(Effect_i \mid Cause).$$

3.1 Bayesian networks

A **Bayesian network** is a directed graph. Each node corresponds to a random variable. 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. Each node X_i contains the conditional probability distribution $P(X_i \mid Parents(X_i))$.

The Bayes net represents a joint distribution as follows:

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i \mid parents(X_i)).$$

The **chain rule** is

$$P(x_1, \ldots, x_n) = P(x_n \mid x_{n-1}, \ldots, x_1) P(x_{n-1} \mid x_{n-2}, \ldots, x_1) \cdots P(x_2 \mid x_1) P(x_1) = \prod_{i=1}^{n} P(x_i \mid x_{i-1}, \ldots, x_1).$$

We see that the specification of the joint distribution is equivalent to the assertion that

$$P(X_i \mid X_{i-1}, \ldots, X_1) = P(X_i \mid Parents(X_i)) \tag{1}$$

provided that $Parents(X_i) \subseteq \{X_{i-1}, \ldots, X_1\}$. Thus, we require that each node is conditionally independent of its other predecessors in the node ordering, given its parents. The following construction satisfies this condition.

1. Determine the set of variables, and order them. Some orderings will be better than others.
2. For $i = 1, \ldots, n$, do the following
   - Choose from $X_1, \ldots, X_{i-1}$ a minimal set of parents for X_i, such that Equation (1) is satisfied.
   - Insert a link from each parent to X_i.
   - Write the conditional probability table $P(X_i \mid Parents(X_i))$.

A variable is conditionally independent of its non-descendants, given its parents. It is also conditionally independent of all other nodes in the network, given its parents, children, and children's parents (its **Markov blanket**).

- A path between two nodes X and Y is **blocked** given a set of evidence nodes $\{Z_i\}$
  - if the path goes through a **collider** node (the path contains a pair of arrows that collide head to head) and neither the collider nor any of its descendants are evidence nodes, or
  - if the path contains an evidence node that is not a collider.
- X and Y are said to be **d-separated** by evidence nodes $\{Z_i\}$ if there exists no unblocked path from X to Y. If X and Y are d-separated by $\{Z_i\}$, they are conditionally independent given $\{Z_i\}$.

3.2 Exact inference in Bayesian networks

A basic task is to calculate some conditional probability.

Inference by enumeration: product rule, marginalize, normalize

$$\begin{aligned} P(x \mid e_1, \dots, e_k) &= \alpha P(x, e_1, \dots, e_k) \\ &= \alpha \sum_{y_1} \sum_{y_2} \cdots \sum_{y_\ell} P(x, e_1, \dots, e_k, y_1, \dots, y_\ell) \\ &= \alpha \sum_{y_1} \sum_{y_2} \cdots \sum_{y_\ell} \prod P(X_i \mid Parents(X_i)) \end{aligned}$$

An improvement is **variable elimination**, which avoids redundant computation by storing some calculations.

If the Bayes net is a **polytree** (there is at most one undirected path between any two nodes in the network), then the time and space complexity of exact inference is linear in the size of the network. Otherwise, variable elimination can have exponential time and space complexity.

3.3 Approximate inference in Bayesian networks

A simple direct sampling algorithm (for absolute probability, not conditional) generates samples from the joint distribution specified by the network. In topological order, sample values for each variable according to the conditional probabilities of the network to arrive at a sample of all the variables. Repeat to estimate the query absolute probability.

If there are evidence variables, one method is to simply throw away any samples that do not match the evidence. This is **rejection sampling**. The drawback is that many samples will be thrown away. In **likelihood sampling**, we sample only the nonevidence variables, so that our samples will be consistent with the evidence. However, we weight the samples according to the probability that the evidence would occur.

The weight of a sample is calculated as follows:

- w is initialized to 1
- In topological order, for each X_i, do
  - If X_i is an evidence variable with value x_i, then $w \leftarrow wP(X_i = x_i \mid Parents(X_i))$.
  - Otherwise, sample a value for X_i from $P(X_i \mid Parents(X_i))$.

Algorithm 3 Likelihood weighting

```
function LIKELIHOOD-WEIGHTING(X, e⃗, bn, N)
    for j = 1, ..., N do
        x⃗, w ← Weighted-Sample(bn, e⃗)
        W⃗[x] ← W⃗[x] + w where x is the value of X in x⃗
    return Normalize(W⃗)
function WEIGHTED-SAMPLE(bn, e⃗)
    w ← 1
    x⃗ ← an event with n elements initialized from e⃗
    for all doX_i ∈ {X_1, ..., X_n}
        if X_i is an evidence variable with value x_i in e⃗ then
            w ← wP(X_i = x_i | Parents(X_i))
        elsex⃗[i] ← a random sample from P(X_i | Parents(X_i))
    return x⃗, w
```

Markov chain Monte Carlo algorithms generate each sample by making a random change to the preceding sample. We discuss a particular form called **Gibbs sampling**. It begins at an arbitrary state

that agrees with the evidence variables, and generates a next state by randomly sampling a value for a nonevidence variable X_i. This sampling is conditioned on the current values of the variables in the Markov blanket (parents, children, and children's parents) of X_i.

Let $q(\vec{x} \to \vec{x}')$ be the transition probability. This is a **Markov chain** on the state space. Defining $\pi_t(\vec{x})$ to be the probability that the system is in state x at time t, we have

$$\pi_{t+1}(\vec{x}') = \sum_{\vec{x}} \pi_t(\vec{x}) q(\vec{x} \to \vec{x}').$$

We say that the chain has reached is **stationary distribution** if $\pi_t = \pi_{t+1}$, and denote the distribution by π. Then

$$\pi(\vec{x}') = \sum_{\vec{x}} \pi(\vec{x}) q(\vec{x} \to \vec{x}')$$

for all $\vec{x}'$ If the transition probability distribution q is **ergodic** (every state is reachable from each other, and there are no strictly periodic cycles), then there is exactly one distribution π satisfying this equation for any given q.

We say that q is in **detailed balance** with $\pi(\vec{x})$ if

$$\pi(\vec{x}) q(\vec{x} \to \vec{x}') = \pi(\vec{x}') q(\vec{x}' \to \vec{x})$$

for all $\vec{x}, \vec{x}'$.

Detailed balance implies stationarity:

$$\sum_{\vec{x}} \pi(\vec{x}) q(\vec{x} \to \vec{x}') = \pi(\vec{x}') \sum_{\vec{x}} q(\vec{x}' \to \vec{x}) = \pi(\vec{x}').$$

We now show that Gibbs sampling satisfies detailed balance. Defining $\vec{X_i}^c$ to be all variables excluding X_i, we have

$$q_i(\vec{x} \to \vec{x}') = q_i((x_i, \vec{x_i}^c) \to (x_i, \vec{x_i}^c)) = P(x_i' \mid \vec{x_i}^c, \vec{e})$$

which implies

$$\begin{aligned}
\pi(\vec{x}) q_i(\vec{x} \to \vec{x}') &= P(\vec{x} \mid \vec{e}) P(x_i' \mid \vec{x_i}^c, \vec{e}) \\
&= P(x_i, \vec{x_i}^c \mid \vec{e}) P(x_i' \mid \vec{x_i}^c, \vec{e}) \\
&= P(x_i \mid \vec{x_i}^c, \vec{e}) P(\vec{x_i}^c \mid \vec{e}) P(x_i' \mid \vec{x_i}^c, \vec{e}) \\
&= P(x_i \mid \vec{x_i}^c, \vec{e}) P(x_i', \vec{x_i}^c \mid \vec{e}) \\
&= \pi(\vec{x}') q_i(\vec{x}' \to \vec{x})).
\end{aligned}$$

The middle steps done by performing the chain rule forward and backward.

We only have left to show how the sampling occurs. Using the fact that a variable is conditionally independent of all other variables given its Markov blanket, we have

$$P(x_i' \mid \vec{x_i}^c, \vec{e}) = P(x_i' \mid mb(X_i)) = \alpha P(x_i' \mid Parents(X_i)) \prod_{Y_j \in Children(X_i)} P(y_j \mid parents(Y_j)).$$

4 Uncertainty over time

Given a temporal random variable X_t, the **Markov assumption** is the condition that X_t depends only on a finite fixed number of previous states. In a **first-order Markov chain**, we have

$$P(X_t \mid X_{0:t-1}) = P(X_t \mid X_{t-1}).$$

We assume that the process is **stationary**, that $P(X_t \mid X_{t-1})$ is the same for all t. We also have the **sensor Markov assumption**:

$$P(E_t \mid X_{0:t}, E_{0:t-1}) = P(E_t \mid X_t).$$

Given an initial distribution $P(X_0)$, along with the transition and sensor models, we have a specification of the complete joint distribution over all the variables.

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

4.1 Inference in temporal models

We explore the following tasks:

- **Filtering**: computing the distribution over the most recent state (the **belief state**) given all evidence so far.
- **Prediction**: computing the distribution over a future state, given all evidence to so far.
- **Smoothing**: computing the distribution over a past state, given all evidence up to the present.
- **Most likely explanation**: given a sequence of observations, computing the sequence of states most likely to have generated those observations.
- **Learning**: learning the transition and sensor models from observations.

4.1.1 Filtering, prediction, and likelihood

$$\begin{aligned}
P(X_{t+1} \mid e_{1:t+1}) &= \alpha P(e_{t+1} \mid X_{t+1}, e_{1:t}) P(X_{t+1} \mid e_{1:t}) && \text{(Bayes's rule)} \\
&= \alpha P(e_{t+1} \mid X_{t+1}) P(X_{t+1} \mid e_{1:t}) \\
&= \alpha P(e_{t+1} \mid X_{t+1}) \sum_{x_t} P(X_{t+1} \mid x_t, e_{1:t}) P(x_t \mid e_{1:t}) \\
&= \alpha P(e_{t+1} \mid X_{t+1}) \sum_{x_t} P(X_{t+1} \mid x_t) P(x_t \mid e_{1:t})
\end{aligned}$$

We see that the filtering probability can be calculated recursively by propagating a forward message $f_{1:k} := P(X_k \mid e_{1:k})$ using the above relationship, rewritten as

$$f_{1:t+1} = \alpha \, \text{Forward}(f_{1:t}, e_{t+1}),$$

initialized with $f_{1:0} := P(X_0)$. The time and space for each update are both constant in t (but do depend on the size of the state space).

Prediction is very similar. The one step update is

$$P(X_{t+k+1} \mid e_{1:t}) = \sum_{x_{t+k}} P(X_{t+k+1} \mid x_{t+k}) P(x_{t+k} \mid e_{1:t}).$$

We can also calculate the **likelihood** of the evidence sequence using a similar equation.

$$P(X_t, e_{1:t}) = P(e_t \mid X_t) \sum_{x_{t-1}} P(X_t \mid x_{t-1}) P(x_{t-1} \mid e_{1:t-1})$$

Defining $\ell_{1:t}(X_t) = P(X_t, e_{1:t})$, we have the relationship

$$\ell_{1:t+1} := \text{Forward}(\ell_{1:t}, e_{t+1}),$$

and can calculate the likelihood by

$$P(e_{1:t}) = \sum_{x_t} P(x_t, e_{1:t}) = \sum_{x_t} \ell_{1:t}(x_t).$$

4.1.2 Smoothing

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

where $\times$ represents pointwise multiplication of vectors, and where the backward message is defined as $b_{k+1:t} := P(e_{k+1:t} \mid X_k)$.

$$\begin{aligned}
P(e_{k+1:t} \mid X_k) &= \sum_{X_{k+1}} P(e_{k+1:t} \mid X_k, x_{k+1}) P(x_{k+1} \mid X_k) \\
&= \sum_{x_{k+1}} P(e_{k+1} \mid x_{k+1})(x_{k+1} \mid X_k) \\
&= \sum_{x_{k+1}} P(e_{k+1}, e_{k+2:t} \mid x_{k+1}) P(x_{k+1} \mid X_k) \\
&= \sum_{x_{k+1}} P(e_{k+1} \mid x_{k+1}) P(e_{k+2:t} \mid x_{k+1}) P(x_{k+1} \mid X_k)
\end{aligned}$$

which gives the relationship

$$b_{k+1:t} = \text{Backward}(b_{k+2:t}, e_{k+1}).$$

Note that the backward message is initialized with $b_{t+1:t} = P(e_{t+1:t} \mid X_t) = P(\mid X_t) = \vec{1}$ where $\vec{1}$ is a vector of 1s.

Both the forward and backward recursions take a constant (w.r.t. t) amount of time per step, so the time complexity of smoothing is $O(t)$. If we want to smooth the whole sequence, we could run smoothing once for each time step: $O(t^2)$. A more efficient method is to record the results of the forward filtering over the whole sequence, and then compute the smoothed estimate while computing the backward messages from t to 1. This is the **forward-backward algorithm**.

4.1.3 Most likely sequence

Faulty algorithm: compute distributions by smoothing at each time step, and taking the most likely state. The problem is that these distributions are over single time steps, whereas we are considering the most likely sequence, for which we need to consider joint probabilities over all time steps.

The intuition for the correct algorithm (the **Viterbi algorithm**) is to find a recursive relationship between most likely paths to x_{t+1} and most likely paths to x_t.

First, note that

$$\max_{x_{0:t+1}} P(x_{0:t+1} e_{1:t+1}) = \max_{x_{t+1}} \max_{x_{0:t-1}} P(x_{0:t}, e_{1:t}).$$

The inner maximum can be calculated recursively.

$$\begin{aligned}
\max_{x_{0:t}} P(x_{0:t+1} \mid e_{1:t+1}) &= \alpha \max_{x_{0:t}} P(e_{t+1} \mid x_{t+1}) P(x_{t+1} \mid x_t) P(x_{0:t}, e_{1:t}) \\
&= \alpha \max_{x_t} \left[P(e_{t+1} \mid x_{t+1}) P(x_{t+1} \mid x_t) \max_{x_{0:t-1}} P(x_{0:t}, e_{1:t}) \right]
\end{aligned}$$

with the base case $\max_{x_{0:-1}} P(x_{0:0} \mid e_{1:0}) := P(x_0)$.

The relationship is very similar to the forward propagation function.

4.2 Hidden Markov Models

In the particular case when the state of a process is a single discrete random variable, we call the process a **hidden Markov model** we may represent transition probabilities as matrices and apply some small improvements.

One application is robot localization: the observations are the robot's observations, and the hidden state is its actual location.

4.3 Kalman filters

Gaussian distribution:

$$p(x) = \mathcal{N}(\mu, \sigma^2)(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

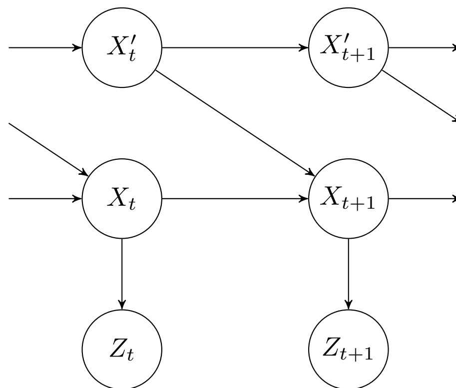
Multivariate Gaussian distribution:

$$p(\vec{x}) = \mathcal{N}(\vec{\mu}, \Sigma)(x) = \frac{1}{\sqrt{(2\pi)^k \det \Sigma}} \exp\left(\frac{1}{2}(\vec{x}-\vec{\mu})^T \Sigma^{-1} (\vec{x}-\vec{\mu})\right)$$

Consider a temporal variable $(X_t, Y_t, Z_t) \in \mathbb{R}^3$. We use **linear Gaussian** distributions: the next state is a linear function of the current state, plus some Gaussian noise. Considering the x-coordinate X_t, with a time interval Δ, and assuming constant velocity, the update is $X_{t+\Delta} = X_t + X_t'\Delta$. Adding Gaussian noise gives a linear Gaussian transition model:

$$P(X_{t+\Delta} = x_{t+\Delta} \mid X_t = x_t, X_t' = x_t') = \mathcal{N}(x_t + x_t'\Delta, \sigma^2)(x_{t+\Delta}).$$

The following is the Bayesian network associated with such a system.



To generalize to multiple dimensions, we use the **multivariate Gaussian** distribution.

The Gaussian distribution has nice properties:

- If (X, Y) is bivariate Gaussian, then X is Gaussian.
- If (X, Y) is Gaussian, then $X \mid Y$ is Gaussian.
- If $X \mid Y$ is Gaussian and Y is Gaussian, then (X, Y) is Gaussian.
- If X and Y are Gaussian, then $aX + bY$ is Gaussian.
- If $P(X_t \mid e_{1:t})$ is Gaussian, and the transition model $P(X_{t+1} \mid x_t)$ is linear Gaussian, then the one-step predicted distribution

  $$P(X_{t+1} \mid e_{1:t}) = \int_{x_t} P(X_{t+1} \mid x_t) P(x_t \mid e_{1:t}) \, dx_t$$

  is also Gaussian.

- If the prediction $P(X_{t+1} \mid e_{1:t})$ is Gaussian and the sensor model $P(e_{t+1} \mid X_{t+1})$ is linear Gaussian, then after conditioning on the new evidence, the updated distribution

$$P(X_{t+1} \mid e_{1:t+1}) = \alpha P(e_{t+1} \mid X_{t+1}) P(X_{t+1} \mid e_{1:t})$$

is also Gaussian.

4.4 Dynamic Bayesian networks

A **dynamic Bayesian network** is a Bayesian network that represents a temporal probability model as we discussed in Section 4.1. Every HMM can be represented as a DBN with a single state variable and a single evidence variable. Moreover, every discrete-variable DBN can be represented as an HMM by combining all the state variables into a single state variable whose values are all possible tuples of values of the original individual state variables. The advantage of retaining the DBN structure is to take advantage of sparseness in the temporal probability model. For example, if a DBN has 20 boolean state variables, each of which as 3 parents, then there are 20×2^3 probabilities, compared with the corresponding HMM which has $(2^{20})^2$ probabilities.

We showed that a Kalman filter model can be represented in a DBN with continuous variables and linear Gaussian conditional distributions, but not every DBN can be represented by a Kalman filter model because DBNs can model arbitrary distributions, while state distributions in Kalman filters must be univariate Gaussian.

4.4.1 Exact inference in DBNs

Since DBNs are Bayesian networks, we can simply **unroll** (construct the Bayes net) the DBN for as many time slices as necessary, and then apply the inference algorithms for Bayes nets. However, the time and space requirements per update would grow with $O(t)$. It is smarter to use **variable elimination** to achieve constant time and space per filtering update.

4.4.2 Approximate inference in DBNs

We could apply likelihood weighting to the unrolled DBN, but this would also suffer from increasing time and space requirements per update because the algorithm runs through the entire network each time. A key innovation is that we can use the samples themselves as an approximate representation of the current state distribution.

Another problem with likelihood weighting is that the algorithm suffers if the evidence variables are "downstream" from the variables being sampled, because the samples are generated without influence form the evidence. In a DBN, none of the state variables have any evidence variables among its ancestors, so although the weight of each sample will depend on the evidence, the actual set of samples generated will be completely independent of the evidence, which makes for poor performance. The second key innovation is to focus the set of samples on the high-probability regions of the state space, by throwing away samples of very low weight, while replicating those with high weight.

These algorithms are called **particle filtering**. A population of N initial-state samples is created by sampling from the prior distribution $P(X_0)$. Then for each time step, we perform the following.

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 $P(X_{t+1} \mid x_t)$.

2. Each sample is weighted by the likelihood it assigns to the new evidence $P(e_{t+1} \mid x_{t+1})$.

3. The population is resampled to generate a new population of N samples. Each new sample is selected from the current population; the probability that a particular sample is slected is proportional to its weight. The new samples are unweighted.

5 Decisions under uncertainty

A **Markov decision process** is a sequential decision problem for a fully observable, stochastic environment with a Markovian transition model and additive rewards. The outcome of an action is nondeterministic. A **policy** π specifies an action for every state. An optimal policy π^* is one that yields the highest utility.

MDP	Search	HMM
states	states	states
actions	actions	no actions
rewards	goal states	—
Markov transitions	Markov transitions	Markov transitions
nondeterministic actions	deterministic actions	—
observable states	observable states	unobservable states

For an infinite time limit, an optimal policy is **stationary**: there is no reason to behave differently in the same state at different times. Similarly, an agent's preferences between state sequences are also **stationary**: $[s_0, s_1, s_2, \ldots] \leq [s_0, s_1', s_2', \ldots] \iff [s_1, s_2, \ldots] \leq [s_1', s_2', \ldots]$.

We define the **utility** of a state sequence using **discounted rewards**, where the **discount** γ is between 0 and 1, by

$$U([s_0, s_1, \ldots]) := R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots = \sum_{t=0}^{\infty} \gamma^t R(s_t).$$

The discount factor is a measure of how much consideration is given to future rewards. With $\gamma < 1$ and a bounded reward function, the utility of infinite sequences is finite, which avoids the problem of comparing infinite utilities when $\gamma = 1$.

The expected utility of executing policy π at state s is

$$U^\pi(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t) \,\middle|\, \pi, S_0 = s\right]$$

where $S_1, S_2, \ldots$ are the random variables for the states reached by executing the policy.

We define $\pi_s^* := \operatorname{argmax}_\pi U^\pi(s)$, the optimal policy when s is the starting state. A consequence of using discounted utilities with infinite horizons is that the optimal policy is independent of the starting state. The intuition is that if π_a^* and π_b^* both reach c, there's no reason for them to disagree on where to go next. Thus, we may simply write π^*. We write $U^* := U^{\pi^*}$ to denote the expected sum of discounted rewards when executing an optimal policy.

With knowledge of U^*, we can recover π^*. The optimal policy is to choose the action that maximizes the expected utility of the subsequent state:

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

Conversely, if π^* is known, we can find U^* from the definition:

$$U^*(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t) \,\middle|\, \pi^*, S_0 = s\right]$$

Theorem 5.1 (Bellman equations)**.** *If U^* is an optimal policy, then*

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

for all s.

Proof.

$$
\begin{aligned}
U^*(s) &= E\left[\sum_{t=0}^{\infty}\gamma^t R(S_t)\middle|\pi^*, S_0 = s\right] \\
&= R(s) + E\left[\sum_{t=1}^{\infty}\gamma^t R(S_t)\middle|\pi^*, S_0 = s\right] \\
&= R(s) + \gamma E\left[\sum_{t=0}^{\infty}\gamma^t R(S_{t+1})\middle|\pi^*, S_0 = s\right] \\
&= R(s) + \gamma E\left[E\left[\sum_{t=0}^{\infty}\gamma^t R(S_{t+1})\middle|\pi^*, S_0 = s, S_1 = s'\right]\middle|\pi^*, S_0 = s\right] \\
&= R(s) + \gamma + \sum_{s'} P(s' \mid s, \pi^*(s)) E\left[\sum_{t=0}^{\infty}\gamma^t R(S_{t+1})\middle|\pi^*, S_0 = s, S_1 = s'\right] \\
&= R(s) + \gamma + \sum_{s'} P(s' \mid s, \pi^*(s)) U^*(s') \\
&= R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U^*(s').
\end{aligned}
$$

□

5.1 Value iteration

The **value iteration algorithm** begins with arbitrary initial values for the utilities, and performs the **Bellman update**:

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

This is iterated until $||U_{i+1} - U_i||_\infty := \max_{s \in S}|U_{i+1}(s) - U_i(s)| < \frac{1-\gamma}{\gamma}\epsilon$, which guarantees that $||U_{i+1} - U^*|| < \epsilon$.

5.2 Proving correctness

The Bellman update B is a **contraction**. More precisely,

$$||BU - BV||_\infty < \gamma ||U - V||_\infty$$

for $\gamma < 1$ and any U, V. It is clear that a contraction has a unique fixed point, and that the function moves the argument closer to the fixed point (since $BU^* = U^*$), so repeated applications of a contraction always reach the fixed point in the limit.

Assuming this is true, we can clearly see that the error in U_i is reduced by a factor of γ at each iteration:

$$||BU_i - U^*||_\infty \leq \gamma ||U_i - U^*||_\infty.$$

Since the utilities are bounded by $R_{\max}/(1-\gamma)$, implying that the maximum initial error is $||U_0 - U^*||_\infty \leq 2R_{\max}/(1-\gamma)$, we have that the error at the Nth iteration is $\gamma^N 2R_{\max}/(1-\gamma)$. If we want this to be less than ϵ, we can choose N sufficiently large.

Moreover, if $||U_{i+1} - U_i||_\infty < \frac{1-\gamma}{\gamma}\epsilon$, then $||U_{i+1} - U^*||_\infty < \epsilon$:

$$\begin{aligned}
\frac{1-\gamma}{\gamma}\epsilon &> ||U_{i+1} - U_i||_\infty \\
&\geq |||U_{i+1} - U^*||_\infty - ||U_i - U^*||_\infty| \\
&= ||U_i - U^*||_\infty - ||U_{i+1} - U^*||_\infty \\
&\geq \frac{1}{\gamma}||U_{i+1} - U^*||_\infty - ||U_{i+1} - U^*||_\infty \\
&= \frac{1-\gamma}{\gamma}||U_{i+1} - U^*||_\infty
\end{aligned}$$

What we are really concerned with is the performance of the policy based on one-step look-ahead using by U_i. The **policy loss** $||U^{\pi_i} - U^*||$ is the most the agent can lose by executing π_i instead of π^*. It turns out that if $||U_i - U^*||_\infty < \epsilon$, then $||U^{\pi_i} - U^*||_\infty < 2\epsilon\frac{\gamma}{1-\gamma}$. (See homework W5.)

5.3 Policy iteration

The **policy iteration algorithm** alternates between the following two steps, starting at some initial policy π_0.

- **Policy evaluation**: given a policy π_i, calculate $U_i := U^{\pi_i}$.
- **Policy improvement**: calculate a new policy π_{i+1} using one-step look-ahead based on U_i:

$$\pi_{i+1}(s) \leftarrow \text{argmax}_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U_i(s')$$

The algorithm terminates when the utilities stop changing.

To do policy evaluation, the actions are fixed by the policy, so we need only solve the system of simplified Bellman equations:

$$U_i(s) = R(s) + \gamma \sum_{s'} P(s' \mid s, \pi_i(s)) U_i(s').$$

Solving the system is now possible because it is linear, due to the absence of the max operator.

Instead of calculating the utilities exactly, performing a few steps of simplified value iteration to get an estimate of the utilities works fairly well:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \sum_{s'} P(s' \mid s, \pi_i(s)) U_i(s')$$

This is **modified policy iteration**.

In **asynchronous policy iteration**, we only update the policy for a subset of states at each iteration. This is useful in focusing locally, if we are unlikely to visit certain parts of the state space.

5.4 Partially observable MDPs

In a POMDP, the environment is only partially observable: the agent does not know what state it is in. We consider the **belief state** b, a probability distribution over all possible states the agent may be in. To calculate its current belief state as the conditional probability distribution over the actual states given the sequence of percepts and actions so far, we perform **filtering**:

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

which can be written analogously as $b' = \text{Forward}(b, a, e)$.

The probability of perceiving e given that a was performed in belief state b is given by summing over all the actual states s' that the agent might reach

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

Then the probability of reaching b' from b given action a is

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

where $P(b' \mid e, a, b) = \mathbf{1}_{b' = \text{Forward}(b,a,e)}$.

We can also write the expected reward for the actual state the agent might be in as

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

Armed with $P(b' \mid b, a)$ and $\rho(b)$ we can define an observable MDP. It turns out that the optimal policy for this MDP is also an optimal policy for the original POMDP.

6 Learning

In **unsupervised learning**, the agent learns patterns in the input without feedback. For example, **clustering** detects potentially useful clusters of input examples. In **reinforcement learning**, the agent learns from rewards or punishments. In **supervised learning** (learning by example), the agent observes some example input-output pairs and learns a function.

6.1 Supervised learning

Given a **training set** of input-output pairs, we want to find a function h that approximates the unknown function f that generated the set. When the output y is on a finite set of values, the learning problem is called **classification** (the output will sometimes be referred to as a **label**). If it is a number, the problem is **regression**.

We approximate f with a function h chosen from a **hypothesis space** $\mathcal{H}$. A **consistent** hypothesis agrees with the training set. A learning problem is **realizable** if $f \in \mathcal{H}$.

Supervised learning can choose the hypothesis $h^* := \text{argmax}_{h \in \mathcal{H}} P(h \mid \mathit{data})$ that is most likely given the data. But by Bayes's rule,

$$h^* = \text{argmax}_{h \in \mathcal{H}} P(\mathit{data} \mid h) P(h).$$

$P(h)$ is low if $\mathcal{H}$ is too large; in this way, we discourage complex hypotheses by giving them a low probability.

There is a tradeoff between complex hypotheses that fit the training data well and simpler hypotheses that generalize better. Moreover, there is a tradeoff between the expressiveness of a hypothesis space and the complexity of finding a good hypothesis within that space.

6.2 The theory of learning

Any hypothesis that is seriously wrong will be ousted with high probability after a certain small number of examples because it will make an incorrect prediction. Any hypothesis that is consistent with a sufficiently large number of training examples is **probably approximately correct (PAC)**.

We assume that future examples are drawn independently from the same fixed distribution as past examples. Moreover, we assume that f is deterministic and in $\mathcal{H}$.

The training error and generalization error are, respectively,

$$\text{error}_T(h) := \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}_{h(x^{(i)}) \neq f(x^{(i)})}$$

$$\text{error}_G(h) := E\left[\mathbf{1}_{h(x) \neq f(x)}\right]$$

A hypothesis is called **approximately correct** if $\text{error}_G(h) \leq \epsilon$; it is "close" to the true function f.

Theorem 6.1. *If h is consistent on N examples, then with probability $\geq 1 - \delta$,*

$$\text{error}_G(h_A) \leq \frac{1}{N}(\ln|\mathcal{H}| + \ln(1/\delta)).$$

Proof. Choosing some small ϵ, we can define $\mathcal{H}_{\text{bad}} := \{h \in \mathcal{H} : \text{error}_G(h) \geq \epsilon\}$. We calculate the probability that a bad hypothesis $h_b \in \mathcal{H}_{\text{bad}}$ is consistent with the N examples. Since $\text{error}_G(h_b) \geq \epsilon$, we know $P(h_b(x) = f(x)) \leq 1 - \epsilon$. By the independence assumption, the probability that h_b agrees on N examples is bounded by $(1-\epsilon)^N$. Noting that $\mathcal{H}_{\text{bad}} \subseteq \mathcal{H}$, we have

$$P(\mathcal{H}_{\text{bad}} \text{ contains a consistent hypothesis}) \leq |\mathcal{H}_{\text{bad}}|(1-\epsilon)^N \leq |\mathcal{H}|(1-\epsilon)^N.$$

Because we would like $|\mathcal{H}|(1-\epsilon)^N \leq \delta$, we have

$$\epsilon \geq \frac{1}{N}\left(\ln(1/\delta) + \ln|\mathcal{H}|\right)$$

where we use the fact that $1 - \epsilon \leq e^{-\epsilon}$ (for all $x \in \mathbb{R}$, $e^x - x - 1 \geq 0$). Thus for any ϵ satisfying the above inequality, we have that the probability that $\mathcal{H}_{\text{bad}}$ contains a consistent hypothesis is $\leq \delta$. The most useful one is $\epsilon = \frac{1}{N}\left(\ln(1/\delta) + \ln|\mathcal{H}|\right)$, which proves the theorem. □

For inconsistent hypotheses, we have the following theorem.

Theorem 6.2. *If h is a hypothesis learned on a training set of N examples, then with probability $\geq 1 - \delta$,*

$$\text{error}_G(h) \leq \text{error}_T(h) + \sqrt{\frac{1}{N}\left(\ln|\mathcal{H}| \ln \frac{2N}{\ln|\mathcal{H}|} + \ln(1/\delta)\right)}$$

For infinite hypothesis spaces (where $VC(\mathcal{H})$ is the dimension of $\mathcal{H}$),

Theorem 6.3. *If h is a hypothesis learned on a training set of N examples, then with probability $\geq 1 - \delta$,*

$$\text{error}_G(h) \leq \text{error}_T(h) + \sqrt{\frac{1}{N}\left(VC(\mathcal{H}) \ln \frac{2N}{VC(\mathcal{H})} + \ln(4/\delta)\right)}$$

$VC(\mathcal{H})$ represents the largest number of points that a classifier can classify perfectly, for any assignment of labels to the points. For example, a linear separator in a two-dimensional space has $VC(\mathcal{H}) = 3$; more generally, a n-dimensional hyperplane has $VC(\mathcal{H}) = n + 1$.

6.3 Decision trees

A **decision tree** takes in a vector of inputs and outputs a label by performing a sequence of tests (a path from root to leaf).

The decision tree algorithm is recursive.

- If the remaining examples are all positive or all negative, then we make the current node a leaf and assign the appropriate label.
- If there are some positive and some negative examples, then choose the best attribute to split them.
- If there are no examples left, we return a value calculated from the plurality of all the examples used in the node's parent.
- If there are no attributes left, but both positive and negative examples, then some examples have the same description but different classifications. The best we can do is return the plurality of the remaining examples.

Algorithm 4 Decision Tree algorithm

```
function DECISION-TREE(examples, attributes, parentExamples)
    if examples is empty then
        return Plurality-Value(parentExamples)
    else if all examples have the same classification then
        return the label
    else if attributes is empty then
        return Plurality-Value(examples)
    else
        A ← argmax_{a∈attributes} Importance(a, examples)
        tree ← a new decision tree with root A
        for all value v_k of A do
            exs ← {e : e ∈ examples, e.A = v_k}
            subtree ← Decision-Tree(exs, attributes − A, examples)
            add a branch to tree with test A = v_k and subtree subtree
        return tree
```

To choose the best node, we introduce the notion of **entropy**. The entropy of a Boolean random variable that is true with probability q is

$$B(q) = -q \log_2 q + (1-q)\log_2(1-q).$$

Note that $B(0) = B(1) = 0$ and that B reaches its maximum at $B(0.5) = 1$.

If an example set has p positive and n negative examples, then the entropy of the whole set is $B(p/(p+n))$. The **information gain** (reduction of entropy) on a split that we seek to maximize is thus

$$B\left(\frac{p}{p+n}\right) - \sum_{k=1}^{d} \frac{p_k + n_k}{p+n} B\left(\frac{p_k}{p_k+n_k}\right).$$

This method of splitting is flawed in that it tends to split on attributes with many values. Alternative splitting rules that correct this flaw are **gain ratio** and **GINI index**.

For continuous attributes, we can sort by value, and then find the best threshold for a split.

Unfortunately, decision trees tend to overfit. One method of pruning is **early stopping**, which does not split when it does not look worthwhile. However, it is hard to tell how good a split is without looking further downward; some attributes work better together than alone. Another alternative is **post-pruning**, which takes the full tree and prunes back by eliminating splits that do not seem to be statistically valid.

6.4 Linear classification

If $\vec{x} \in \mathbb{R}^d$, then a linear classifier would be something like

$$h(\vec{x}) = g(\vec{w} \cdot \vec{x}) = \mathbf{1}_{\vec{w}\cdot\vec{x}\geq\theta} = \text{sign}(\vec{w} \cdot \vec{x} - \theta).$$

In this case, g is a **threshold function**. Note that we interpret the output to be in $\{0, 1\}$.

A **perceptron** takes in a vector $\vec{x}$ and uses such a function to output some y. In the simple **perceptron algorithm**, we initialize $\vec{w}$ and θ to some values, and perform the following for each example $(\vec{x}, y)$.

1. Calculate the estimate $\hat{y} \leftarrow \text{sign}(\vec{w}_t \cdot \vec{x} - \theta_t)$.

2. Update weight $\vec{w}$. If $\hat{y} = y$, don't change. Otherwise, move the weight slightly to avoid making the same mistake next time:
$$\vec{w}_{t+1} \leftarrow \vec{w}_t + \alpha(y - \hat{y})\vec{x}$$
for some learning rate α.

3. Update threshold if $\hat{y} \neq y$.

$$\theta_{t+1} \leftarrow \theta_t - \alpha(y - \hat{y})$$

Threshold functions are cumbersome because they are not differentiable. We choose to use the logistic function $g(z) = (1+e^{-z})^{-1}$ instead, which forms a soft boundary from 0 to 1. Note that $g'(z) = g(z)(1-g(z))$.

We seek to minimize the squared loss, summed over all examples: $\sum_{j=1}^{N}(y_j - h(x_j))^2$. For this ,we do **gradient descent** (the direction of steepest descent is $\vec{w} - \alpha \nabla Loss(\vec{w})$). For a single example $\vec{x}, y$), the derivation is

$$\begin{aligned}
\frac{\partial}{\partial w_i} Loss(\vec{w}) &= \frac{\partial}{\partial w_i}(y - h_{\vec{w}}(\vec{x}))^2 \\
&= 2(y - h_{\vec{w}}(\vec{x}))\frac{\partial}{\partial w_i}(y - h_{\vec{w}}(\vec{x})) \\
&= -2(y - h_{\vec{x}}(\vec{x}))g'(\vec{w} \cdot \vec{x})\frac{\partial}{\partial w_i}\vec{w} \cdot \vec{x} \\
&= -2(y - h_{\vec{w}}(\vec{x}))g'(\vec{w} \cdot \vec{x})x_i \\
&= -2(y - h_{\vec{w}}(\vec{x}))h_{\vec{w}}(\vec{x})(1 - h_{\vec{w}}(\vec{x}))x_i
\end{aligned}$$

Thus, the weight update for minimizing loss is

$$\vec{w} \leftarrow \vec{w} + \alpha(y - h_{\vec{w}}(\vec{x}))h_{\vec{w}}(\vec{x})(1 - h_{\vec{w}}(\vec{x}))\vec{x}$$

We can express the “and,” “or,” and “not” functions using perceptrons, but not “xor” because it is not linearly separable.

A **neural network** consists of a network of perceptrons feeding inputs and outputs to each other. There is an input layer that receives the true inputs, and an output layer that gives the final outputs. We use the **back-propagation algorithm**, which propagates the error from the output layer to the hidden layers when updating the weights. The weight update is similar to the one above.

6.5 Support vector machines

Support vector machines construct a **maximum margin separator** which helps with generalization. Using the **kernel trick**, they can embed the data into a higher dimensional space, which makes the linear separator more powerful.

We assume $y \in \{-1, 1\}$.

The solution depends only on a small subset of the examples: the **support vectors**, which lie closest to the separator. We want to find the separator that separates the examples correctly and has the largest margin.

Letting $\delta := \min_i \left| \frac{\vec{w} \cdot \vec{x}^{(i)} + b}{\sqrt{\vec{w} \cdot \vec{w}}} \right| = \min_i \frac{y^{(i)}(\vec{w} \cdot \vec{x}^{(i)} + b)}{\sqrt{\vec{w} \cdot \vec{w}}}$, we seek

$$\text{argmax}_{\vec{w},b}\, \delta \text{ such that } \frac{y^{(i)}(\vec{w} \cdot \vec{x}^{(i)} + b)}{\sqrt{\vec{w} \cdot \vec{w}}} \geq \delta, \forall i.$$

After some manipulations, we may write the problem as

$$\text{argmin}_{\vec{w},b}\, \frac{1}{2}\vec{w} \cdot \vec{w} \text{ such that } y^{(i)}(\vec{w} \cdot \vec{x}^{(i)} + b) \geq 1, \forall i.$$

Our hypothesis is

$$h(\vec{x}) = \text{sign}(\vec{w} \cdot \vec{x} + b).$$

To allow some errors, we allow **soft margin separation**, which penalizes errors based on their distance from the correct side.

$$\text{argmin}_{\vec{w},b,\xi_i}\, \frac{1}{2}\vec{w} \cdot \vec{w} + C\sum_i \xi_i \text{ such that } y^{(i)}(\vec{w} \cdot \vec{x}^{(i)} + b) \geq 1 - \xi_i, \xi_i \geq 0, \forall i.$$

C controls the tradeoff between margin size and training error, while $\sum_i \xi_i$ is an upper bound on the number of training errors.

This turns out to be equivalent to a dual optimization problem:

$$\text{argmax}_\alpha \sum_j \alpha_j - \frac{1}{2} \sum_j \sum_k \alpha_j \alpha_k y^{(j)} y^{(k)} (\vec{x}^{(j)} \cdot \vec{x}^{(k)}) \text{ such that } \sum_{j=1}^n y^{(j)} \alpha_j = 0, 0 \leq \alpha_j \leq C, \forall j$$

If $\{\alpha_j\}$ is the solution to the dual problem, then $\vec{w} = \sum_j \alpha_j y^{(j)} \vec{x}^{(j)}$ is the solution of the primal problem. Then our hypothesis will be

$$h(\vec{x}) = \text{sign}\left(\sum_j \alpha_j y^{(j)} (\vec{x} \cdot \vec{x}^{(i)}) + b\right).$$

An interesting property of the dual space is that $\vec{x}$ appears only as part of a dot product.

When examples are not linearly separable, we employ the **kernel trick**. We find a function Φ that maps $\vec{x}$ to a vector of feature values. For example, a feature space could be $\Phi(\vec{x}) = (x_1^2, x_2^2, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1x_2, 1)$, in which case the kernel function would be $K(\vec{x}, \vec{x}') := \Phi(\vec{x}) \cdot \Phi(\vec{x}') = (\vec{x} \cdot \vec{x}' + 1)^2$. In the dual optimization problem, we simply replace $\vec{x} \cdot \vec{x}'$ with $K(\vec{x}, \vec{x}')$.

6.6 Ensemble models

In **bootstrap aggregating (bagging)**, we take the original training set of size n, and sample n examples from it with replacement to get a "new" training set. We train a base hypothesis on each of the **bootstrap samples** and take the majority vote. Bagging works best on unstable learning algorithms, where small changes in the training set result in large changes in the hypothesis. Neural networks, perceptrons, and decision trees are unstable, while SVMs are stable (depend only on support vectors).

Boosting implements the notion of having hypotheses correct each other. It is primarily used with **weak hypotheses** (learning hypotheses that do slightly better than random guessing). It uses a **weighted example set** and shifts focus on examples that were misclassified. We define the weighted error, based on weight distribution D, by

$$\text{error}_D(h) = \sum_{i:h(x^{(i)}) \neq y^{(i)}} D(i).$$

We assume $y \in \{-1, 1\}$.

Algorithm 5 AdaBoost

---

function ADABOOST($S = \{(\vec{x}^{(i)}, y^{(i)})\}_{i \in \{1,\ldots,n\}}$)
 $D_1(i) \leftarrow \frac{1}{n}$ for all i
 for $t = 1, \ldots, T$ **do**
  $h_t \leftarrow$ Weak-Learner(S, D_t)
  $\epsilon_t = \text{error}_{D_t}(h_t)$
  $\alpha_t \leftarrow \frac{1}{2} \ln \frac{1-\epsilon_t}{\epsilon_t}$
  for $i = 1, \ldots, n$ **do**
   $D_{t+1}(i) \leftarrow \frac{1}{Z_t} D_t(i) \exp(-\alpha_t y^{(i)} h_t(x^{(i)}))$
 return $H(\vec{x}) := \text{sign}(\alpha_1 h_1(\vec{x}) + \cdots + \alpha_T h_T(\vec{x}))$

The **edge** of h_t is $\gamma_t := \frac{1}{2} - \epsilon_t > 0$.

Theorem 6.4. *If H is returned by AdaBoost on training set S, then the training error of H decreases exponentially with the number of rounds:*

$$\text{error}_S(H) \leq \exp\left(-2 \sum_{t=1}^T \gamma_t^2\right).$$

Proof intuition. If $\vec{x}^{(i)}$ is misclassified by H, then it is misclassified by the weighted majority of the h_ts. The weight of $\vec{x}^{(i)}$ must have increased many times, so $D_T(i)$ must be large. But since $\sum_j D_T(j) = 1$, there cannot be too many examples with large weight, i.e., there are few mistakes. □

Boosting is often used with decision trees or decision stumps.

6.7 Unsupervised learning

In **unsupervised learning** there are no examples from which to learn.

Clustering attempts to partition the data into groups that make sense. A particular instance is **hierarchical agglomerative clustering**, which begins with each example in its own cluster, and merges the closest clusters iteratively until there is only one cluster.

There are several notions of distance between two clusters:

- Single link:
$$\min_{x^{(i)} \in C_1, x^{(j)} \in C_2} \text{Dist}(x^{(i)}, x^{(j)})$$
- Average link:
$$\frac{1}{|C_1||C_2|} \sum_{x^{(i)} \in C_1} \sum_{x^{(j)} \in C_2} \text{Dist}(x^{(i)}, x^{(j)})$$
- Complete link:
$$\max_{x^{(i)} \in C_1, x^{(j)} \in C_2} \text{Dist}(x^{(i)}, x^{(j)})$$

The Dist function can be Euclidean distance, Manhattan distance, etc.

After performing the iterations, we have a **dendrogram** that shows the order of the merges of the clusters. One must make a decision as to where to cut the dendrogram to obtain a partition.

In **K-means clustering**, we look for K good **cluster centers** so that we may assign examples to be in the cluster with the closest center. The algorithm begins with random centers, and then alternates between calculating the clusters with those centers, recalculating the centers of these clusters, and repeating until the clusters do not change anymore. Letting c_k be the center of cluster C_k, we would like to to minimize

$$\min_{C_k, c_k} \sum_k \sum_{x^{(i)} \in C_k} ||x^{(i)} - c_k||_2^2.$$

The algorithm is guaranteed to converge/stop, but it may converge to a local minimum. Some improvements include choosing good initial cluster centers (heuristic: choose initial centers to be far away from each other), and multiple restarts.

6.8 Learning in Bayesian networks

Suppose we have the structure of a Bayes net, for which we do not know the true parameters (probabilities) θ. We can try to learn them by generating a dataset of examples/samples from the Bayes net, and estimating the parameters. We use the notion of **maximum likelihood**

The probability of seeing the dataset given the true parameters is

$$P(Data \mid \theta) = P(x^{(1)}, \ldots, x^{(n)} \mid \theta) = \prod_{i=1}^{n} P(x^{(i)} \mid \theta)$$

where the last term can be further decomposed using the Bayes net structure. We are looking for the parameters that maximize the likelihood of seeing the dataset.

$$\theta^{ML} := \text{argmax}_\theta \, P(Data \mid \theta)$$

For general Bayes nets, the estimated probability is

$$\hat{P}^{ML}(X_i = x \mid Parents(X_i) = (x_j, \dots, x_k)) = \frac{\#\{examples : X_i = x, Parents(X_i) = (x_j, \dots, x_k)\}}{\#\{examples : Parents(X_i) = (x_j, \dots, x_k)\}}$$

What if some of the variables were **hidden** or **latent**? If we knew θ, we could calculate the conditional probabilities and fill in the table.

The **hard EM (expectation-maximization) algorithm** begins with a random θ, fills in the dataset by calculating the conditional probabilities and sampling from them, and then re-estimates θ from the new dataset. K-means clustering is an example of a hard EM algorithm. "Hard" here means that when we sample, we choose a single value for each variable. In contrast, the **soft EM algorithm** does not sample and make a choice, but instead updates θ using the probabilities. This mitigates a problem in hard EM, where information (the conditional probability) is lost after sampling.

These algorithms maximize

$$E\left[\prod_{i=1}^{n} P(x^{(i)} \mid \theta) \middle| D, \theta\right]$$

6.9 Learning hidden Markov models

We have the transition probabilities $T_{ss'}$, the observation model O_{se}, and the initial probabilit π_s. If the states X_t were observable, we could just use the usual estimations via sampling. However, since they are not observable, we must use EM. We can use smoothing to help calculate.

The **Baum-Welch algorithm** is an instance of EM.

- E step: calculate, for every s, s', t, i,

$$P(X_{t+1} = s', X_t = s \mid e^i_{1:T}, T, O, \pi)$$
$$P(X_t = s \mid e^i_{1:T}, T, O, \pi)$$

- M step: update T, O π.

6.10 Reinforcement learning in MDPs

In **reinforcement learning** in MDPs, the agent learns how to act based on the rewards he receives.

In **model-based** reinforcement learning, we estimate $P(s' \mid s, a)$ and $R(s)$, and apply MDP algorithms to find the optimal policy. In **model-free** reinforcement learning, we learn how to act without explicit estimation.

If we assume that actions are taken according to some policy π (**passive reinforcement learning**, then we can measure the goodness of the policy.

$$U^\pi(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t) \middle| \pi, S_0 = s\right]$$

This is similar to policy iteration, but the probabilities and rewards are unknown.

In **batch** model-based learning, we estimate the transition probability as

$$\hat{P}(s' \mid s, \pi(s)) \leftarrow \frac{\text{transitions } s \to s'}{\text{transitions from } s}$$

We can be a little smarter by updating the average instead of recalculating: **incremental** model-based learning:

$$\hat{P}(s' \mid s_t, \pi(s_t)) \leftarrow \frac{1}{N(s_t)}(\hat{P}(s' \mid s_t, \pi(s_t))(N(s_t) - 1) + \mathbf{1}_{s'=s_{t+1}})$$

Using this estimated probability, we can estimate $\hat{U}$ using policy evaluation.

In model-free learning, we can estimate

$$U^\pi(s) = R(s) + \gamma \sum_{s'} P(s' \mid s, \pi(s)) U^\pi(s') = R(s) + \gamma E[U^\pi(s') \mid s, \pi(s)] = E\left[R(s) + \gamma U^\pi(s') | s, \pi(s)\right]$$

using

$$\frac{1}{N(s)} \left(\sum_{s_t = s} r_t + \gamma U^\pi(s_{t+1}) \right)$$

The **temporal difference** learning update is **incremental model-free learning**:

$$\hat{U}^\pi(s_t) \leftarrow \frac{1}{N(s_t)} \Big((N(s_t) - 1)\hat{U}^\pi(s_t) + r_t + \gamma \hat{U}^\pi(s_{t+1}) \Big) = \hat{U}^\pi(s_t) + \frac{1}{N(s_t)} \underbrace{(r_t + \gamma \hat{U}^\pi(s_{t+1}) - \hat{U}^\pi(s_t))}_{\text{temporal difference}}$$

In both cases, $\hat{U}^\pi$ converges to the true U^π. Convergence is faster in model-based learning, but time/space requirements are much lower in the model-free approach. Model-based and model-free learning have time complexity cubic and linear, respectively, in the number of states, and have time complexity cubic and constant, respectively, in the number of states.

In **active reinforcement learning** the agent chooses his actions. In model-based learning, we can estimate the transition probability in the same way as above, replacing $\pi(s)$ with a_t. However, for model-free learning, we are stuck because in the Bellman equation, the expectation is inside the max function.

$$U^*(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U^*(s') = R(s) + \gamma \max_{a \in A(s)} E[U^*(s') \mid s, a]$$

We introduce the **action-value function** or **quality function** $Q(s, a)$ which is the expected value of executing a at state s:

$$Q^\pi(s, a) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t) \middle| S_0 = s, \text{perform action } a, \text{ then follow policy } \pi \right]$$

It satisfies

$$U^*(s) = \max_a Q^*(s, a).$$

It has a Bellman-like equation, but the expectation is outside the max, so we may apply the temporal difference trick from before.

$$Q^*(s, a) = R(s) + \gamma \sum_{s'} P(s' \mid s, a) \max_{a'} Q^*(s', a') = E[R(s) + \gamma \max_{a'} Q^*(s', a') \mid s, a]$$

This is called **Q-learning**.

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \frac{1}{N(s_t, a_t)} \underbrace{(r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}, a') - \hat{Q}(s_t, a_t))}_{\text{temporal difference}}$$

How do we choose the action? If we choose $a_t = \operatorname{argmax}_a \hat{Q}(s_t, a)$, it may not converge to the optimal policy, and it might settle with an average policy, not exploring other states/actions. If we choose a random action each time, it will converge to the optimal policy, but will waste a lot of time on bad actions, and can reap a large negative reward. We need a tradeoff between exploration and exploitation: multi-armed bandit problems.

We can exploit most of the time and explore once in a while, or we may assign higher utility to underexplored state/action pairs. If all state/action pairs are explored often enough, then Q-learning is guaranteed to converge to the optimal policy.

So far, the algorithms ignore the the structure of the states. In **function approximation**, we describe each state (or state/action pair) using a set of features, and then approximate U (or Q) using an **evaluation function** of the features. A popular evaluation function are neural networks. This reduces the model and makes learning faster. It also generalizes to unseen states, saving the need for explorations.

References

[1] Stuart Russell and Peter Norvig. **Artifical Intelligence: A Modern Approach.** Pearson Education, New Jersey, 3rd edition, 2010.