CS188: Artificial Intelligence Notes

1 Search and Planning

1.1 Search Problems

(a) A **rational agent** is an entity that has goals and performs a series of actions that yield the best/optimal expected outcome given these goals.

(b) A **reflex agent** simply selects an action based solely on the current state of the world.

(c) A **planning agent** maintains a model of the world and uses this model to simulate performing various actions before selecting the best one.

(d) A **world state** contains all information about a given state, whereas a **search state** contains only the information necessary for planning. A search problem is defined by four things:

  i. **State space:** The set of all possible states in a given world.

  ii. **Successor function:** A function that takes in a state and an action and computes the cost of that action as well as the successor state.

  iii. **Start state:** The state in which an agent exists initially.

  iv. **Goal test:** Boolean-valued function for if a state is the goal state.

(e) **State Space size:** Determine the state representation, then use the fundamental counting principle. For example, on a Pacman $M \times N$ grid, the state (x, y, food pellet grid) has size $MN2^{MN}$.

1.2 Search Algorithms

(a) A **state space graph** is constructed with nodes representing states, with directed edges to its successors. Each state appears exactly once. **Search trees**, on the other hand, have no restriction on the number of times a state appears.

(b) The general algorithm for graph search is as follows:

```
function GRAPH-SEARCH(problem, fringe, strategy) return a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe, strategy)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            for child-node in EXPAND(STATE[node], problem) do
                fringe ← INSERT(child-node, fringe)
            end
    end
```

(c) Note that a more memory efficient variant checks if a node is in the closed set before adding it to the fringe. Tree search is identical except that it lacks a closed set.

(d) We say that a search algorithm is **complete** if the algorithm is guaranteed to find a solution if one exists, and **optimal** if the algorithm returns the least cost path to the goal.

(e) Below, let b be the branching factor of the search tree and m be the maximum depth of the tree.

(f) **Depth First Search**

  i. The fringe is implemented as a stack.

  ii. Only Graph-Search DFS is *complete* (could get stuck in cycles in Tree Search). Not optimal.

  iii. Time complexity is $O(b^m)$. Space complexity is $O(bm)$.

(g) **Breadth First Search**

  i. The fringe is implemented as a queue.

  ii. BFS is always *complete*. Optimality guaranteed only if all edge weights are equal.

  iii. Time and space complexity is $O(b^s)$, where s is the depth of the solution.

(h) **Uniform Cost Search (Dijkstra)**

  i. The fringe is implemented as a priority queue, ordered by lowest cumulative distance traveled (*computed backward cost*).

  ii. A more memory efficient variant updates the priority of a node on the fringe rather than adding another.

  iii. UCS is complete and optimal.

  iv. Let C^* be the cost of the optimal path, and ϵ be the minimal edge weight between any two nodes in the state space graph. Then an upper bound on time and space complexity is $O(b^{C^*/\epsilon})$.

(i) **Greedy Search**

  i. The fringe is implemented as a priority queue, ordered by lowest hueristic value.

  ii. It is neither complete nor optimal.

(j) **A***

  i. The fringe is implemented as a priority queue, ordered by lowest computed backward cost *plus* estimated forward cost (a heuristic).

ii. Both complete and optimal, given an appropriate heuristic (see below).

(k) **Heuristics**

i. A heuristic is a function $h(n)$ that takes in a state and returns a numeric estimate of the cost to the goal state. They are typically solutions to relaxed problems (ex: Manhattan distance, Euclidean distance).

ii. A heuristic is **admissible** if the value returned by the heuristic is neither negative nor greater than the true cost of reaching the goal from the given state.

iii. A heuristic is **consistent** if for all edges $A \rightarrow B$, $h(A) - h(B) \leq \text{cost}(A, B)$. Consistency ensures that the first time a node is expanded, the cost to that state is optimal.

iv. Any valid heuristic must have that $h(G) = 0$.

v. Consistency *implies* admissibility.

vi. A* Tree search is guaranteed complete and optimal for any *admissible* heuristic. A* graph search is guaranteed complete and optimal for any *consistent* heuristic. Note that a simple fix for admissible but inconsistent heuristics is to allow a state to be expanded again if the cost to that state is lower than what it was last time it was expanded.

1.3 Constraint Satisfaction Problems

Before, the path to the goal was most important. Now, only the goal itself is important.

(a) **CSP**s are a special subset of search problems. A state is defined by assignments to **variables** X_i. The set of values a variable X_i can take on is the **domain** D. (We will only consider discrete variables with finite domains). The goal test is a set of constraints specifying allowable combinations of values for subsets of variables.

(b) A **unary** constraint is a constraint on a single variable (e.g., $X \geq 2$) while a **binary** constraint relates two variables (e.g., $X \geq Y$). In a binary constraint graph, each node is a variable and each edge is a constraint relating two variables. A non-binary CSP has constraints relating multiple variables (e.g., the standard constraints for Sudoku).

(c) **Backtracking Search** is DFS with two extra features:

i. Variables are ordered.

ii. Consider only values which do not conflict with previous constraints; incremental goal test.

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
  return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
      if value is consistent with assignment given CONSTRAINTS[csp] then
          add {var = value} to assignment
          result ← RECURSIVE-BACKTRACKING(assignment, csp)
          if result ≠ failure then return result
          remove {var = value} from assignment
  return failure
```

(d) **Optimization 1: Filtering**

Forward Checking: After a value is assigned to X_i, prune the domains of unassigned variables that share a constraint with X_i that would violate the constraint if assigned. In this way, we return failures earlier, when a variable runs out of possible values. But it doesn't provide early detection for all failures.

Arc Consistency: An arc $X \rightarrow Y$ is consistent iff for every x (possible value) in X there is some y (possible value) in Y which could be assigned without violating a constraint. If not for some x, delete x from X (the tail). Then re-enqueue all arcs ending at X that had already been dequeued. We run the following algorithm after every assignment:

```
function AC-3( csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables {X_1, X_2, ..., X_n}
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do                          tail      head
    (X_i, X_j) ← REMOVE-FIRST(queue)
    if REMOVE-INCONSISTENT-VALUES(X_i, X_j) then
      for each X_k in NEIGHBORS[X_i] do
        add (X_k, X_i) to queue
---------------------------------------------------------------------
function REMOVE-INCONSISTENT-VALUES( X_i, X_j) returns true iff succeeds
  removed ← false
  for each x in DOMAIN[X_i] do
    if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint X_i ↔ X_j
      then delete x from DOMAIN[X_i]; removed ← true
  return removed
```

AC-3 is a stronger version of forward checking and finds failure earlier. There are n^2 arcs (where n is the number of variables) which each require a d^2 checking operation (where d is the size of the domain). Each arc can be put back on the queue at most d times, so run time $O(n^2 d^3)$. After enforcing arc consistency, there can be one solution left, many solutions left, or no solutions left (and not know it).

k-consistency: For each k nodes, any consistent assignment to $k-1$ can be extended to the k^{th} node. (Arc consistency is 2-consistency). Higher quality filtering, but very costly.

(e) **Optimization 2: Ordering**

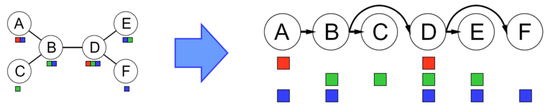
MRV: Select the variable with the minimum remaining values in its domain next. This is "fail-fast" ordering.

LCV: After picking a variable, order values by least constraining value (that is, from highest total number of values remaining across all variables, to lowest). Requires arc consistency checking.

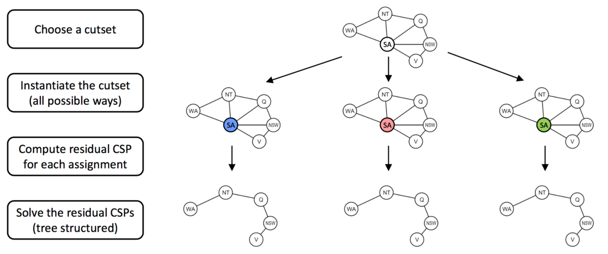
(f) **Optimization 3: Structure**

A general CSP worst case run time is $O(d^n)$. But if the constraint graph has no loops (i.e., it is a tree), then the CSP can be solved in $O(nd^2)$ time.

The algorithm for a **tree-structured CSP** is as follows: Choose any variable to be the root variable and order variables so that parents precede children. We remove backward (starting with the right-most node, enforce arc-consistency for the arc from the parent to the node, then do the same for the second-to-right-most node and its parent, and so on). Then we assign forward (start at the left-most node picking any value remaining in its domain, then go to the next variable picking any value in its domain that is consistent with its parent, and continue left to right). We can then find the solution without backtracking in one pass.



For nearly tree-structured CSPs, we can use cutset conditioning.


Choose a cutset
Instantiate the cutset (all possible ways)
Compute residual CSP for each assignment
Solve the residual CSPs (tree structured)
WA
NT
Q
SA
NSW
V

First find a cutset (i.e. a set of variables such that if removed, the remaining constraint graph forms a tree), then loop over all instantiations of the cutset variables, and for each instantiation call the tree-structured CSP solver to verify if for that instantiation a solution exists. A cutset of size c gives runtime $O(d^c(n-c)d^2)$.

(g) **Iterative Improvement**

Take an assignment with unsatisfied constraints. While not solved, randomly select any conflicted variable and choose a value that violates the fewest constraints.

This simple strategy (called min-conflicts) is often effective in practice. Can solve n-queens in almost constant time for arbitrary n.

(h) Min-conflicts is an example of local search (as opposed to tree search) because there is no fringe of unexplored alternatives, but only one option that is continuously made better. Faster and more memory efficient, but not complete or optimal.

(i) Another local search is hill climbing: starting anywhere, repeatedly move to the best neighboring state until no neighbors are better than current. The problem is that it may get stuck in a local maximum.

(j) Simulated annealing allows "downhill" moves when no neighbors are better than current, but make them rarer as time goes on. If we decrease the "temperature" slowly enough, we are guaranteed to converge to optimal solution.

1.4 Adverserial Search

(a) Adversarial search returns a strategy, or **policy**, (as opposed to a plan) which simply recommends the best possible move given some configuration.

(b) General games involve agents with independent utilities (values placed on outcomes), while **zero-sum games** involve agents with opposite utilities where one agent acts to maximize while the other acts to minimize.

(c) A **game tree** is a directed graph whose nodes are positions in a game and whose edges are actions. The levels, or **plies**, of a game tree alternate between players.

(d) A state's value is defined as the best possible outcome (utility) an agent can achieve from that state. Only terminal state values are known. In adverserial game trees, the **minimax** value of a node is the best achievable utility against an optimal adversary. If the opponent plays suboptimally, we are guaranteed a better value than minimax. This value is computed mutually recursively: the value of a max agent node is the max of the values of the min agent child nodes, and vice-versa. Same space and time complexity as DFS.

(e) Complete game trees are realistically impossible to search, so we forget optimality and replace terminal utilities with an evaluation function at a given depth. Such functions are typically weighted linear sums of features.

(f) **Alpha beta pruning** is an efficiency improvement to the minimax algorithm that maintains two values, α and β, which represent the maximum and minimum score that the maximizing agent and minimizing agent is assured of, respectively. Initially $\alpha = -\infty$ and $\beta = \infty$. It can happen that when choosing a certain branch of a certain node, $(\beta \le \alpha)$. If this is the case, the parent node will not choose this node so the other branches of the node do not have to be explored.

```
def value(state, a=MinInt, b=MaxInt):
    if state.isTerminal():
        return state.score
    elif state.agent.isMaxAgent():
        return maxValue(state, a, b)
    else:
        return minValue(state, a, b)

def maxValue(state, a, b):
    v = MinInt
    for successor in state.getSuccessors():
        v = max(v, value(successor, a, b))
        if v >= b:
            return v
        a = max(a, v)
    return v

def minValue(state, a, b):
    v = MaxInt
    for successor in state.getSuccessors():
        v = min(v, value(successor, a, b))
        if v <= a:
            return v
        b = min(b, v)
    return v
```

Although intermediate values may be wrong, the pruning has no effect on the minimax value computed for the root. This improvement reduces the minimax algorithm's runtime to $O(b^{d/2})$ (when the ordering of child nodes is optimal), effectively doubling the depth the search can go to with the same amount of computation. Optimal ordering is achieved when the best move for a particular agent is it's first child.

(g) In the traditional **minimax** algorithm, the levels of the game tree alternate from max to min until the depth limit of the tree has been reached. In an **expectimax** tree, the "chance" nodes are interleaved with the max and min nodes. Instead of taking the max or min of the utility values of their children, chance nodes take a weighted average, with the weight being the probability that that child is reached. These nodes can represent randomness like dice rolls, or an agent that chooses actions randomly.

(h) Pruning is not possible with expectimax, since there are no guarantees with node values.

(i) There's room for robust variation in node layering, allowing for game trees that are modified expectimax/minimax hybrids for any zero-sum game.

(j) Non-zero sum games may involve involve multiple maximixing agents, each trying to maximize their component of a utility tuple. Can give rise to competition or cooperation dynamically.

1.5 Utility

(a) **Utilities** are functions that take in outcomes and return scores that reflect the preferences of the agent. The principle of maximum expected utility is that a *rational* agent should chose the action that maximizes its expected utility, given its knowledge. In artificial intelligence, we hard-wire utilities to let behaviors emerge.

(b) For the minimax algorithm, the scale of the evaluation function doesn't matter: only the ordering of the children matters. States are insensitive to *monotonic transformations*, or any transformation that preserves ordering (such as squaring).

(c) In contrast, for expectimax, the magnitude of a state must be meaningful, so states are sensitive to monotonic transformations.

(d) An agent must have preferences among prizes $A, B, \cdots$ and lotteries $L = [p, A; (1-p)B]$.

- Preference: $A \succ B$
- Indifference: $A \sim B$
- Prefers or indifferent to: $A \succeq B$

(e) There exist constraints on preferences to be considered rational. Rational preferences imply behavior describable as maximization of expected utility. The axioms of rationality:

  i. $(A \succ B) \vee (B \succ A) \vee (A \sim B)$
  ii. $(A \succ B) \wedge (B \succ C) \implies (A \succ C)$
  iii. $A \succ B \succ C \implies \exists p\ [p, A; 1-p, C] \sim B$
  iv. $A \sim B \implies [p, A;\ 1-p, C] \sim [p, B;\ 1-p, C]$
  v. $A \succ B \implies (p \geq q \iff [p, A;\ 1-p, B] \succeq [q, A;\ 1-q, B]$

(f) The principle of MEU says that for any rational preferences, there exists a real-valued function U consistent with these preferences such that

- $U(A) \geq U(B) \iff A \succeq B$
- $U([p_1, S_1; \cdots; p_n S_n]) = \sum_i p_i U(S_i)$

We see that a utility function is risk-seeking if it is concave up; risk-averse if it is concave down; and risk-neutral if the slope is constant.

1.6 Markov Decision Processes

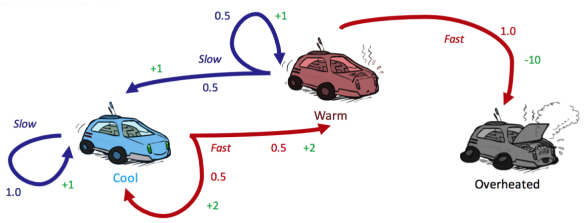
(a) Now we dive deep into non-deterministic search, which we were introduced to by expectimax. An MDP consists of

  i. **States,** including a start state and possibly one or more terminal states.
  ii. Set of **actions**.

iii. A discount factor γ.

iv. **Transition function** $T(s,a,s')$ that returns the probability of ending up in s' by taking action a from state s.

v. **Reward function** $R(s,a,s')$ that returns the reward (positive or negative) for ending up in s' by taking action a from state s.



(b) MDPs are "Markovian" in the sense that the probability of arriving in a state s' at time $t+1$ depends only on the state s and action a taken at time t.

(c) Uncertainty in MDPs is modeled by **q-states**, denoted by the tuple (s,a), which are like expectimax chance nodes: they represent an action having been taken but has yet to be resolved into a successor state.

(d) The goal of MDPs is to *maximize utility*, which here is the sum of rewards. To stop MDPs from obtaining infinite reward, we enforce finite horizons or discount factors.

(e) **Finite horizons** gives an agent a fixed number of timesteps to acrue as much reward as possible.

(f) **Discount factors** model exponential decay of rewards over time. The utility is given by $\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1})$. When $|\gamma| < 1$, this sum is finite valued, in that it must be $\le \frac{R_{max}}{1-\gamma}$.

(g) Solving a MDP amounts to finding an optimal policy $\pi^*(s)$, a function that maps each state s to an action that, if taken, will yield the maximum expected utility.

(h) Denote the optimal value of a state s as $V^*(s)$, which is the expected utility an optimal agent receives starting from s. Denote the optimal value of a q-state (s,a) as $Q^*(s,a)$, which is the expected utility an optimal agent receives starting in s and taking action a. Then

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

and

$$Q^*(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

From this, we get the Bellman equation:

$$V^*(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

This provides a framework for testing optimality. If we can determine a value $V(s)$ for each state s such that the Bellman equation holds true for each state, then $V(s) = V^*(s)$.

(i) **Value Iteration** computes optimal values of states. It is a dynamic programming algorithm that uses an iteratively longer time limit to compute time-limited values until convergence. It works as follows:

i. $\forall s$, initialize $V_0(s) = 0$.

ii. $\forall s$, repeat until values converge:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_k(s')]$$

Note that value iteration will actually converge to the same vector of values from *any* intitial values. For all s, we can interpret the value of $V_k(s)$ as the optimal value of s with a time limit of k time steps. It is exactly what a depth k expectimax search returns. The runtime of this algorithm is $O(S^2A)$ for each iteration where S and A are the number of states and actions, respectively. For an MDP terminating after k timesteps, expectimax will be $O((SA)^k)$ while value iteration will be $O(SA^2k)$.

(j) **Policy extraction** allows us to extract the optimal policy once optimal values for states have been found. It is, $\forall s$:

$$\begin{aligned}\pi^*(s) &= \arg\max_a Q^*(s,a) \\ &= \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]\end{aligned}$$

Clearly, it's better for policy extraction to have the optimal q-values of states. Storing only each $V^*(s)$ means that we must recompute all necessary q-values with the Bellman equation, equivalent to performing a depth-1 expectimax.

(k) **Policy Iteration** uses iterative convergence to compute an optimal policy. It tends to outperform value iteration, by virtue of the fact that policies usually converge must faster than the values of states. It works as follows:

i. Define an arbitrary initial policy $\pi_0(s)$; that is, fix a single action for each state.

ii. Evaluate the current policy with **policy evaluation**: for a policy π, compute $V^\pi(s)$ for all s, where $V^\pi(s)$ is the expected utility received starting in s following policy π:

$$V^\pi(s) = \sum_{s'} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V^\pi(s')]$$

Note that without the max operator, this becomes a linear system of S equations that we can solve. We could also just use the update rule as in value iteration, but this is slower (runtime is $O(S^2)$ for each update).

iii. Use **policy improvement** to generate a better policy. It runs policy extraction on the (non-optimal!) values obtained from the policy evaluation. $\forall s$,

$$\pi_{i+1}(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^{\pi_i}(s')]$$

iv. Repeat (ii) and (iii) until $\pi_{i+1} = \pi$, at which point the policy has converged and we conclude we have reached $\pi^*(s)$.

(l) Both value iteration and policy iteration are guaranteed to converge for any finite MDP with discount factor $0 < \gamma < 1$.

1.7 Reinforcement Learning

(a) We previously solved MDPs through offline planning; agents had full knowledge of transition and reward functions, so it was possible to precompute optimal actions without ever taking actions.

(b) In online planning, an agent must explore and receive feedback, which it uses to estimate an optimal policy before using this policy for exploitation (reward maximization). This is called **reinforcement learning.**

(c) At each timestep in online planning, an agent starts in s, takes action a, ends up in s', and recieved a reward r. The tuple (s, a, s', r) is a **sample**; a string of samples is an **episode.**

(d) In **model-based learning**, an agent generates an approximation of the transition function $\hat{T}(s,a,s')$ by normalizing the counts it has collected: divide the count for (s,a,s') by the count for (s,a). Whenever we see fit, we can end the agent's training and switch to exploitation by running value or policy iteration on the obtained $\hat{T}$ and $\hat{R}$ values. The idea is simple but the memory overhead can be costly.

(e) Now we consider **model-free learning**. Direct evaluation and temporal difference learning are examples of **passive reinforcement learning**, in which an agent does policy evaluation. (We cannot approximate optimal values directly because the max operator doesn't allow for weighted averaging). Q-learning is an example of **active reinforcement learning**, in which an agent iteratively updates its policy while learning until determining the optimal policy.

(f) **Direct evaluation** fixes some policy π and has the agent experience several episodes, keeping count of total utility gained *starting from* each state in the episode, and the number times it visited each state. It computes the estimated value of any state s by dividing the total utility obtained from all episodes involving s by the number of times s was visited. This method is unnecessarily slow to converge.

(g) **Temporal difference learning** uses the idea of learning from every experience, rather than just at the end of an episode. We initialize all states s to $V^\pi(s) = 0$. At each timestep, an agent takes $\pi(s)$ to get to s' and receives $R(s,\pi(s),s')$. We obtain a **sample value**:

$$\textit{sample} = R(s,\pi(s),s') + \gamma V^\pi(s')$$

and incorporate into the existing model for $V^\pi(s)$ with an **exponential moving average**:

$$V^\pi(s) \leftarrow (1-\alpha)V^\pi(s) + \alpha \cdot \textit{sample}$$

where α is the **learning rate**. It's typical to start out with $\alpha = 1$ and slowly shrink it towards 0. The model for $V^\pi(s)$ only improves, so we see that older (thus less accurate) samples are given exponentially less weight. This method learns at every timestep, and converges to true state values much faster with fewer episodes than direct evaluation.

(h) **Q-learning** learns the q-values of states directly. It approximates **q-value iteration**, which obeys the following update rule:

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')]$$

Q-learning acquires q-value samples at every transition:

$$\textit{sample} = R(s,a,s') + \gamma \max_{a'} Q(s',a')$$

and incorporates them into an exponential moving average:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha \cdot \textit{sample}$$

These updates mimic Bellman updates eventually. Whereas TD learning and direct evaluation learned the values of states under a specific policy, Q-learning learns optimal q-values for every state even when taking suboptimal actions.

(i) To ensure convergence, α must be decreased to 0 over time. In RL, we cannot *guarantee* optimal value convergence unless we allow for infinite exploration.

(j) Q-learning isn't practically efficient when we must store data for millions of q-states. **Approximate Q-learning** learns about general experiences and extrapolates to many similar situations. The key is to represent states (and q-states) as **feature vectors** which encode important properties of the state, such as distance to closest pellet, distance to closest ghost, number of ghosts, etc. Then we can write state or q-state values as linear value functions:

$$V(s) = \vec{w} \cdot \vec{f}(s)$$
$$Q(s,a) = \vec{w} \cdot \vec{f}(s,a)$$

where $\vec{w} = \begin{bmatrix} w_1 & w_2 & \cdots & w_n \end{bmatrix}$ are weights and $\vec{f}(s,a) = \begin{bmatrix} f_1(s,a) & f_2(s,a) & \cdots & f_n(s,a) \end{bmatrix}^T$ are the features. Approximate Q-learning recieves a sample and computes a difference:

$$\mathit{difference} = [R(s,a,s') + \gamma \max_{a'} Q(s',a')] - Q(s,a)$$

then performs the following update:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot \mathit{difference}$$
$$w_i \leftarrow w_i + \alpha \cdot \mathit{difference} \cdot f_i(s,a)$$

Thus, we adjust the weight of the active features according to how large a role they played in an incorrect model prediction.

(k) There is a tradeoff between **exploration and exploitation**. Agents following an **ϵ-greedy** policy define some probability $0 \le \epsilon \le 1$ and explore randomly with probability ϵ and exploit with probability $(1-\epsilon)$. To find a balance is tricky, and requires fine-tuning.

(l) **Exploration functions** eliminate the need for manually setting exploration time. Agents choose actions given by an exploration function, which often takes the form

$$f(s,a) = Q(s,a) + \frac{k}{N(s,a)}$$

where $N(s,a)$ denotes the number of times (s,a) has been visited, and k is some fixed amount. Exploration is automatically encoded by the exploration function, since the term $\frac{k}{N(s,a)}$ gives a bonus to infrequently taken actions. It uses a modified q-value update:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[R(s,a,s') + \max_{a'} f(s',a')]$$

As time goes on, this bonus decreases to 0, so $f(s,a)$ regresses to $Q(s,a)$ and we are back to regular q-learning.

2 Probabilistic Inference

2.1 Probability

(a) **Random variables**, denoted by capital letters, represent aspects of the world that are uncertain. For random variables with binary domains, we sometimes say $\{+x, -x\}$.

(b) A **probability distribution** is a table of value, probability pairs for each value a random variable can take on, such that the sum of is 1.

(c) A **joint distribution** over a set of random variables $X_1, X_2, \cdots, X_n$ specifies a probability to each assignment $P(x_1, x_2, \cdots, x_n)$ for each $x_i \in X_i$ such that the sum is 1.

(d) Marginal distributions are sub-tables which eliminate variables; **marginalization** is the process of summing over a variable to eliminate it.

(e) Conditional probability:

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

(f) A **conditional distribution** is a probability distribution over some variables given fixed values of others. Can be obtained from a joint distribution by selecting the assignments that match the evidence and normalizing (dividing every entry by sum over entries) the selection.

(g) **Inference by enumeration:** We have a set of evidence variables (variables for which we know the values of), a query variable, and hidden (unknown) variables. First we select all entries consistent with the evidence. Then we sum out hidden variables to get a joint distribution with only the query and evidence variables. Lastly normalize.

(h) **Chain rule:** We can write any joint distribution as a product of conditional distributions:

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i | x_{i-1}, \ldots, x_1)$$

(i) **Bayes' Rule:**

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

(j) Two variables X and Y are **independent** ($X \perp\!\!\!\perp Y$) iff $\forall x,y \ P(x,y) = P(x)P(y)$, or equivalently $P(x|y) = P(x)$. Given a joint distribution, we can test for independence by marginalizing, then seeing if the joint distribution formed as if X and Y were independent is the same as the original joint distribution.

(k) Two variables X and Y are **conditionally independent** given Z ($X \perp\!\!\!\perp Y|Z$ and $Y \perp\!\!\!\perp X|Z$) iff $\forall x,y,z \ P(x,y|z) = P(x|z)P(y|z)$ or equivalently $P(x|y,z) = P(x|z)$. Conditional independence allows us to simplify terms in the chain rule product for a joint distribution.

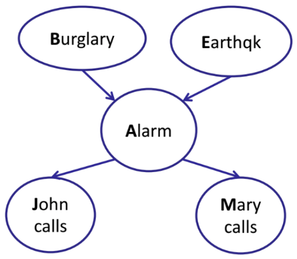
2.2 Bayes' Nets

(a) Bayes' Nets are a technique of implicitly encoding complex joint distributions as simple local interactions. They allow us to perform probabilistic inference. They are directed, acyclic graphs with variables as nodes and edges indicating direct influence between variables.

(b) To recover the probability of an assignment:

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

(c) Bayes' Nets need not reflect causal patterns. The fewer the arrows, the more conditional independence assumptions; it is only mandatory that a Bayes Net does not make *additional* independence assumptions.



(d) **Reasoning about Independence**

We know each variable is conditionally independent of its non-descendants, given its parents. We can go further with *d-separation*: Given a particular query, ask a question about the independence of two variables. For example, for the query $P(A|B)$ we ask "are A and B independent"?, or for the query $P(A|B, C, D)$ we ask "are A and B conditionally independent given C and D"?

i. Draw the ancestral graph (i.e., the graph of only the variables in question and their ancestors- parents, parent's parents, etc).

ii. For every pair of variables with a common child in the ancestral graph, draw an undirected edge between them.

iii. Replace all directed edges with undirected edges.

iv. Delete the given variables (if any) and their edges. Then:

- If the variables are disconnected (a path does not exist, or one or both variables were removed because they were given) in this graph, they are guaranteed to be independent.
- If the variables are connected in this graph, they are not guaranteed to be independent.

(e) Probabilistic inference, or calculating some useful quantity from a joint distribution, is NP-complete. Enumeration is exact but exponential complexity. Variable elimination is exact with worst-case exponential complexity. Sampling is approximate.

(f) **Inference by enumeration** in Bayes' Nets, following the same setup as described earlier, works as follows:

i. We track objects called factors. Initial factors are the conditional probability tables encoded by each node, with only the rows of known values (if there any).

ii. Join all factors. That is, for each variable, join all factors with that variable by point-wise multiplication. In a join operation, the unconditioned variables in the resulting CPT consist of the *set* of input unconditioned variables, and the conditioned variables in the resulting CPT consist of all input variables not in the unconditioned set. Finally join all remaining factors. Ex: To join $P(R)$ and $P(T|R)$, each entry in $P(R,T)$ is $P(r,t) = P(r)P(t|r)$, $\forall r, t$.

iii. Now we have the full relevant joint distribution. Marginalize each hidden variable by summing it out to shrink the factor. Lastly normalize.

(g) **Variable elimination** interleaves joining and marginalizing, which is usually much faster. It works by starting off with initial factors (instantiated by evidence), then while there are hidden variables: pick a hidden variable H, join all factors with H, then sum out H. Join all remaining factors and normalize.

(h) The computational complexity depends on the size (number of entries) in the largest factor. The order of variables to eliminate greatly affects this size; there does not always exist an order that produces small factors. In general, it is best to choose variables that appear in the fewest number of CPTs first.

(i) **Sampling** is faster than variable elimination, and eventually converges to the true probability.

(j) **Prior sampling:** A sample of a full assignment is obtained as follows: for each variable in the Bayes' Net in topological order, draw a random value from [0, 1). Map this value to the corresponding assignment from the variable's CPT given any previous assignments. Add this variable's assignment to the list of assignments. Once every variable has been assigned, the sample is complete.

We can answer queries from samples by computing the number of samples consistent with the query divided by the number of samples consistent with evidence (if any).

(k) If we know the query in advance, we can sample more efficiently:

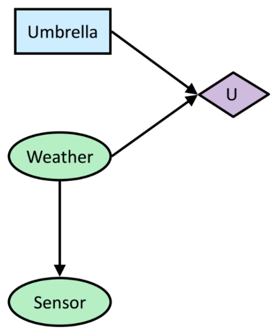
Rejection sampling stops computing a sample once an evidence variable has been assigned a value inconsistent with the query evidence. This method can still be inefficient when evidence variables are "downstream", or the evidence is unlikely.

Likelihood weighting solves these problems by forcing evidence variables to be consistent with the query. Each sample must also have a corresponding weight, which is the product of the probabilities of the evidence variables taking on their forced values given the other assignments. Then, to answer queries, we instead compute the sum of the weights of samples consistent with the query divided by the sum of the weights of samples consistent with the evidence. Weights obtained by likelihood weighting can be very small if upstream variables (before evidence) are unlikely given the evidence, making samples less useful.

Gibbs sampling effectively samples *all* variables conditioned on evidence. It works as follows: first generate an arbitrary sample consistent with evidence. To generate a new sample, choose a non-evidence variable X and resample X given all other assignments (can be done efficiently, as only CPTs with resampled variable need be considered). Repeating this infinitely many times yields samples with correct distribution.

2.3 Decision Networks and VPI

(a) Decision networks are generalizations of Bayes' Nets that allow us to choose actions which maximize expected utility given evidence.



(b) In addition to random variable nodes, there are also action nodes (rectangles) that represent decisions and cannot have parents, and utility nodes (diamonds) which give the utility of each possible action and variable value.

(c) To select an action, we instantiate all evidence. Then calculate the posterior probability (probability given the evidence) of all parents of utility node. Then calculate the expected utility for every action, and choose action with MEU.

(d) The **value of perfect information** is the *expected* gain in MEU if we knew the value of a variable:

$$VPI(E'|e) = \left(\sum_{e' \in E'} P(e'|e) MEU(e, e') \right) - MEU(e)$$

where E is a random variable and MEU is a function of evidence:

$$MEU(e) = \max_a EU(a)$$

In other words, *VPI(E)* represents the utility we we'd be willing to spend to get perfect information about E.

(e) VPI is always

(a) non-negative

(b) zero for any node that is conditionally independent given the evidence so far of all parents of the utility node

(c) non-additive; sometimes knowing the value of one variable lessens the value of knowing another variable

(d) order-independent:

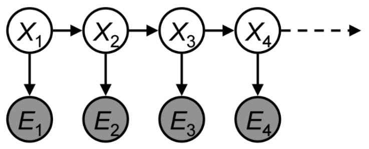
$$\begin{aligned} VPI(A, B) &= VPI(A) + VPI(A|B) \\ &= VPI(B) + VPI(B|A) \end{aligned}$$

2.4 Hidden Markov Models

(a) **Markov models** specify how a state X changes over time. The past is conditionally independent of the future given the present. Note that we have encountered these before in the context of MDP transitions; however, here there will not be choice of action.

(b) The only parameter is $P(X_t|X_{t-1})$, or the transition probabilities. The distribution at timestep infinity $P_\infty(X)$ is independent of the initial distribution and is called the *stationary distribution.* This is the idea behind Google's pagerank.

(c) A **hidden Markov model** is a Markov chain for which the state is only partially observable; at each timestep, we observe the effects of the underlying Markov chain. Ex: in speech recognition, acoustic signals are observations, specific words are states.



An HMM is defined by transitions $P(X_t|X_{t-1})$, an initial distribution $P(X_1)$, and emissions $P(E_t|X_t)$.

(d) An HMM is the simplest **Dynamic Bayes' Net**, or a Bayesian network which relates variables to each other over adjacent time steps.

(e) *Filtering* or *monitoring* is the task of tracking the belief distribution $B_t(X) = P(X_t|e_1, \dots, e_t)$, i.e., the belief in the distribution of X at time t. We start with $B_1(X)$, which is usually uniform, and update as we get observations.
Online belief update, also called the **forward algorithm**, is a dynamic programming algorithm to solve this:

(a) We have $B_t(X) = P(X_t|e_{1\to t})$.

(b) One timestep occurs. We have

$$B'_{t+1} = P(X_{t+1}|e_{1\to t}) = \sum_{x_t} P(X_{t+1}|x_t)B_t(x)$$

(prior beliefs in distribution times transition probabilities).

(c) Evidence comes in. We have

$$B_{t+1} = P(X_{t+1}|e_{1\to t+1}) \propto_{X_{t+1}} P(e_{t+1}|X_{t+1})B'_{t+1}(X)$$

(beliefs are reweighted by emissions).

(f) The **Viterbi algorithm** is a dynamic programming algorithm that will compute the most-likely explanation, or corresponding sequence of states, given a sequence of observations. Let $m_t(x_t)$ denote the probability of the most likely sequence of states given the evidence. Then

$$m_t(x_t) = P(e_t|x_t) \max_{x_{t-1}} P(x_t|x_{t-1})m_{t-1}(x_{t-1})$$

Return the actual sequence by maintaining pointers. Note that the update is identical to that of the Forward Algorithm, except with a max operator instead of a sum.

(g) **Particle filtering** is an approximate method to find $B_t(X)$. In practice, the domain of X can be too large to store in memory (such as in robot locationing), so it is more efficient to store a list of samples and their corresponding values of X, called particles. Then, $P(X = x)$ is approximated by the number of particles with value x.

i. Each particle is moved by sampling its next location from the transition probabilities.

ii. After observing evidence e, assign particles of value x a weight of $P(e|x)$.

iii. Re-sample particles from the weighted sample distribution; this is equivalent to normalizing the distribution.

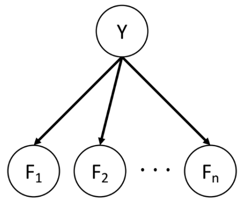
3 Overview of Classification

3.1 Classification with Naive Bayes

(a) For the most part, we have discussed using models to solve problems and make optimal decisions. Machine learning is the process of acquiring models from data.

(b) Classification is a supervised machine learning task to identify the category to which an input belongs. *Features* are the attributes used to make the decision. For example, in spam filtering, features may include words in text, text patterns, whether the sender is a contact, etc.

(c) A **Naive Bayes** model is a simple classifier:



All features are assumed to be independent of all others, given the class. This is unrealistic, but often works well in practice.

(d) We have $P(Y|F_1, \dots, F_n) \propto P(Y)\prod_{i=1}^{n} P(F_i|Y)$. There are $n \times |F| \times |Y|$ **parameters**, which is linear in n. The number of parameters is constant in n for spam filtering, since each feature CPT is the same. In contrast, in optical character recognition, Y takes on values in $[0, 9]$ and each feature node encodes the probability of a particular pixel being illuminated given the character. A common trick to reduce the number of parameters is to write one parameter as the complement of the sum of other parameters, when possible.

(e) Given the parameters ($P(Y)$ and $P(F_i|Y)$), classificiation is done by computing $P(Y|F_1, \dots, F_n)$ (or the log) and returning the value of Y with the highest probability.

(f) Where do we get parameters? **Maximum likelihood estimation (MLE)** finds the parameter values that maximize the likelihood of the observations given the parameters. In this case,

$$P_{ML}(x) = \frac{\text{count}(x)}{N}$$

For example, MLE in spam filtering would set $P(\text{spam}, w_i = w)$ to the number of times w appears in a spam email divided by the total number of words in all spam emails.

(g) The problem with using relative frequencies for parameters is that of *overfitting*; we cannot assign unseen events 0 probability. To generalize better, we need to *smooth* the estimates.

(h) In **Laplace Smoothing**, we pretend we saw each sample k more times. That is, we set

$$P_{LAP,k}(x) = \frac{\text{count}(x) + k}{N + k|X|}$$

k is a hyperparameter and can be interpreted as the strength of the prior belief. For conditionals, this is

$$P_{LAP,k}(x|y) = \frac{\text{count}(x,y) + k}{\text{count}(y) + k|X|}$$

(i) The general process in machine learning involves learning parameters from training data; tuning hyperparameters on hold-out data (i.e., choosing the values that maximize accuracy on hold-out data); then computing accuracy from the test data.

3.2 Classification with Perceptrons

(a) A **Perceptron** is a linear error-driven classifier. Inputs are feature values, and each feature is given a corresponding weight. The activation is the weighted sum:

$$\text{activation}_{\vec{w}} = \sum_i w_i f_i(x) = \vec{w} \cdot f(\vec{x})$$

If the activation is positive, output +1, otherwise -1 (corresponding to two different classes).

(b) If we want to take into account that one class is inherently more likely, we can create a constant offset by setting the first entry in the weight vector to a bias value, and setting the first entry in the feature vector to be 1 always.

(c) Learning the weights occurs as follows:

  (a) Initialize $\vec{w} = \vec{0}$.

  (b) For each training sample x with feature $f(\vec{x})$: If correct, no change. Otherwise, if true class was +1, update $\vec{w}_{new} = \vec{w}_{old} + \vec{f}$. If true class was -1, update $\vec{w}_{new} = \vec{w}_{old} - \vec{f}$.

(d) For multiple classes, simply keep a weight vector for each class and output the class with the highest activation. Learning works the same way, except subtract $f(\vec{x})$ from wrong answer and add $f(\vec{x})$ to the right answer.

(e) *Separability* is the property that there exists some set of parameters to get the training data completely right (i.e., there exists a separating hyperplane). If the training data is linearly separable, then the perceptron *will* eventually converge (binary case); otherwise, it will fail.

(f) The standard update method finds *a* solution (assuming linearly separable data), but not necessarily the optimal solution. **MIRA** finds the *minimum* correcting update. Suppose we saw sample x, predicted it was class y, but it was really class y^*. Then we want new weight vectors W that minimize the change, but still fix the mistake:

$$\min_W \frac{1}{2} \sum_i ||\vec{w_i}^{new} - \vec{w_i}^{old}||^2$$
$$s.t. \quad \vec{w_{y^*}}^{new} \cdot f(\vec{x}) \geq \vec{w_y}^{new} \cdot f(\vec{x}) + 1$$

Note that only two weight vectors in W actually change, namely

$$\vec{w_{y^*}}^{new} = \vec{w_{y^*}}^{old} + \tau f(\vec{x})$$
$$\vec{w_y}^{new} = \vec{w_y}^{old} - \tau f(\vec{x})$$

satisfies these equations. In practice, we cap the value of τ with some hyperparameter C to account for noisy data. Note that with the original perceptron, the update might not be enough to fix the classification error; hence τ can be larger or smaller than 1.

(g) **SVMs** (Support vector machines) find the separator with the maximum margin, which is equivalent to minimizing the magnitudes of the weight vectors. We want to find new weight vectors W of minimum magnitude, but optimize over all samples:

$$\min_W \frac{1}{2} \sum_i ||\vec{w_i}||^2$$
$$s.t. \quad \forall i, y : \vec{w_{y^*}} \cdot f(\vec{x_i}) \geq \vec{w_y} \cdot f(\vec{x_i}) + 1$$

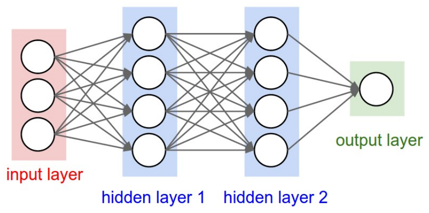
SVMs update weights based on the entire training set (batch), while MIRA updates based on a single training instance at a time (online).

(h) The perceptron algorithm with these modifications, with multiple passes through the data, is often more accurate than the Naive Bayes classifier.

(i) SVMs can perform a non-linear classification by implicitly mapping inputs into high-dimensional feature spaces where they are separable.

3.3 Deep Learning

(a) Manual feature design is difficult, requiring domain-specific expertise. **Deep learning** is the process of using multilayer **neural networks** for feature extraction. Each successive layer uses the output from the previous layer as input, forming a hierarchy from low-level to high-level features.



(b) The inputs are preliminary features of the data; hidden layers will learn high-level abstractions. The goal is to learn the *weights* of the connections between neurons so as to maximize accuracy on training data.

(c) A *loss function* quantifies the agreement between the predicted labels and the true labels.

(d) The activation function of the linear perceptron (+1 for activation, -1 otherwise) is poor for deep networks, as it is difficult to measure progress and optimize. An example of a nonlinear activation function in deep learning is **softmax**, where the activation of each class is the probability of the class y being j:

$$P(y = j | f(\vec{x})) = \frac{e^{\vec{w}_j \cdot f(\vec{x})}}{\sum_i e^{\vec{w}_i \cdot f(\vec{x})}}$$

Note that in the binary case, $\vec{w}_1 = -\vec{w_{-1}}$. Softmax is typically used only on the output node due to its normalization denominator.
Define the loss function $l(w)$ (of the entire neural network) as follows:

$$l(w) = \prod_{i=1}^{m} P(y = y_i | f(\vec{x}_i))$$

In training, we wish to learn weights that maximize this function. This is equivalent to maximizing the log of the expression:

$$ll(w) = \sum_{i=1}^{m} \log P(y = y_i | f(\vec{x}_i))$$

By convention, we minimize $-ll(w)$. This can be done by **gradient descent**.

(e) Gradient descent uses the following weight update in learning:

$$w \leftarrow w - \alpha * \nabla g(w)$$

where $g(w) = ll(w)$ and α is the learning rate (which must be carefully chosen and decreased over time). That is, we move weights against the direction of the gradient in order to move to a local minimum in the loss function. There is also sometimes a momentum parameter that adds a fraction of the previous weight update to the current one, which serves to speed up convergence to the minimum by damping oscillations.

(f) Back-propagation is an efficient method that computes the *exact* gradient using the chain rule to iteratively compute gradients at each layer. A brief overview of how back-propagation on a neural net is performed:

i. We have given inputs (from training data).

ii. Write out the output of the network as a computation graph, naming each connection $(a, b, c, \ldots)$ between neurons.

iii. In the forward pass, write the values of each successive computation node (neuron) until reaching the output node y.

iv. In the backward pass, use the expression for the parent node and the chain rule (as well as values of the node) to compute $\frac{\partial y}{\partial c}$ for each connection c until arriving back at the inputs. This yields a gradient vector containing the partial derivative of the output with respect to each weight in the network.

(g) Random initialization of weights outperforms constant weight initialization, since backpropagation will yield identical gradients for all neurons within each layer, effectively yielding duplicated neurons.

(h) Stochastic gradient descent calculates the gradient on a *mini-batch* of data points rather than the entire data set. In practice, this is both faster and achieves equally good results.

(i) Several activation functions (e.g. tanh, sigmoid, ReLU) are possible, with ReLU ($\max(0, x)$) being the most common choice for intermediate layers.

(j) Larger neural networks always work better than smaller networks, but they are more prone to overfitting. Dropout is an example of *regularization* that attempts to prevent overfitting. Some number of neurons are randomly omitted with probability $1 - p$ from the hidden layers during training in order to break rare dependencies that can occur in training data. The entire neural network is used during test time with outgoing weights scaled by p.

(k) *Convolutional Neural Networks* are used in image processing. They consist of multiple layers of receptive fields which process portions of the input image. The outputs of these collections are then tiled so that their input regions overlap. Tiling allows ConvNets to tolerate translation of the input image.