CS188: Artificial Intelligence
UC Berkeley

Kelvin Lee

October 15, 2020

These are course notes for UC Berkeley's Fall 2020 CS188 Artificial Intelligence course instructed by Professor Anca Dragan.

Contents

§1 Introduction

§1.1 Agents

The central problem in AI is the creation of a rational **agent**, an entity that has goals and tries to perform a series of **actions** that yield the optimal expected outcome. Rational agents exist in an **environment**. An environment and the agents reside within it create a **world**.

- **Reflex agent**: doesn't consider the consequences of its actions, but selects actions based solely on the current state of the world
- **Planning agents:** maintains a model of the world and uses this model to simulate performing various actions.

§1.2 State Spaces and Search Problems

A **search problem** requires four things:

- **state space**: Set of all possible states that are possible in the given world
- **successor function**: A function that takes in a state and an action and computes the cost of performing that action as well as the **successor state**, the state the world would be in if the given agent performed that action
- **start state**: The state in which an agent exists initially
- **goal test**: A function that takes a state as input, and determines whether it is a goal state

The order in which states are considered is determined by a predetermined **strategy**.

Difference between a **world state** and a **search state**: A world state contains all information about a give state, whereas a serach state contains only the information about the world that's necessary for planning(for space efficiency reasons).

Example 1.1

Consider the game of Pacman. We can have two types of search problem: pathing and eat-all-dots.

	Pathing	**Eat-all-dots**
States	(x, y) locations	(x, y) location, dot booleans
Actions	North, South, East, West	North, South, East, West
Successor	Update location only	Update location and booleans
Goal test	Is (x, y) =END?	Are all dot booleans false?

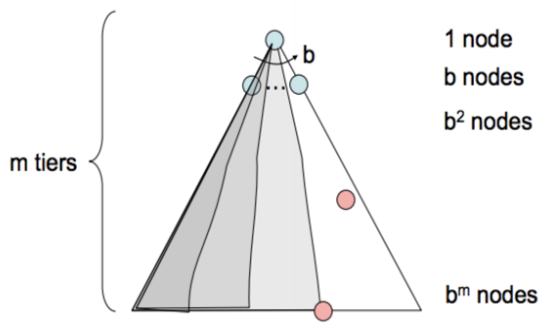
In this case, we see that the states for pathing contain less information than that for eat-all-dots.

§1.3 Searching strategies

When considering strategies for reaching the goal state from the starting state in a search tree, we should consider the following:

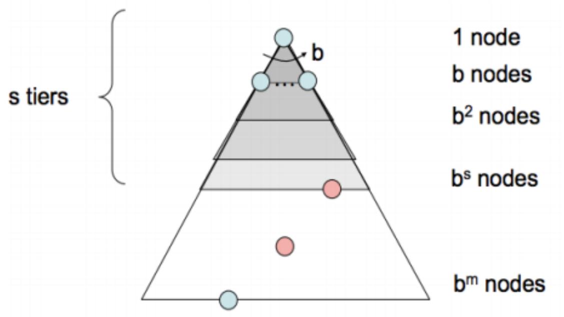
- **Completeness:** if a solution to the search problem exists, it the strategy guaranteed to find it given infinite computational resources?
- **Optimality:** is the strategy guaranteed to find the lowest cost path to a goal state?
- **Branching factor b:** the increase in the number of nodes on the fringe each time a fringe node is dequeued and replaced with its children is $O(b)$. At depth k in the search tree, there exists $O(b^k)$ nodes.
- The maximum depth m.
- The depth of the shallowest solution s.

§1.4 Depth-First Search



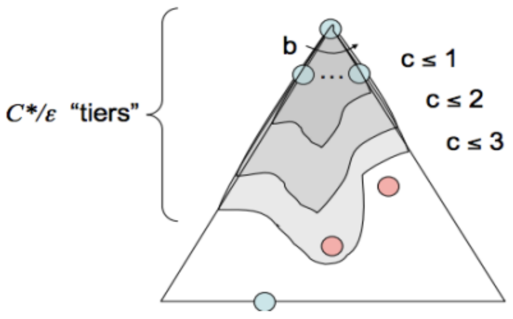
- **Description:** always selects the deepest fringe node from the start node for expansion.
- **Fringe representation:** removing the deepest node and replacing it on the fringe with its children necessarily means the children are now the new deepest nodes - their depth is one greater than the depth of the previous deepest node. This implies that to implement DFS, we require a structure that always gives the most recently added objects highest priority. A last-in, first-out (LIFO) stack does exactly this, and is what is traditionally used to represent the fringe when implementing DFS.
- **Completeness:** DFS is **not complete**. May get stuck in a cycle forever.
- **Optimality:** finds the "leftmost" solution in the search tree without regard for path costs, and so is **not optimal**.
- **Time Complexity:** May end up exploring the entire search tree. Hence, given a tree with maximum depth m, the runtime of DFS is $O(b^m)$.
- **Space Complexity:** In the worst case, DFS maintains b nodes at each of m depth levels on the fringe. This is a simple consequence of the fact that once b children of some parent are enqueued, the nature of DFS allows only one of the subtrees of any of these children to be explored at any given point in time. Hence, the space complexity of BFS is $O(bm)$.

§1.5 Breadth-First Search



- **Description:** BFS is a strategy for exploration that always selects the shallowest fringe node from the start node for expansion.
- **Fringe representation:** If we want to visit shallower nodes before deeper nodes, we must visit nodes in their order of insertion. Hence, we desire a structure that outputs the oldest enqueued object to represent our fringe. For this, BFS uses a first-in, first-out (FIFO) queue, which does exactly this.
- **Completeness:** BFS is **complete**. If a solution exists, then the depth of the shallowest node s must be finite, so BFS must eventually search this depth.
- **Optimality:** BFS is generally **not optimal** because it simply does not take costs into consideration when determining which node to replace on the fringe. The special case where BFS is guaranteed to be **optimal is if *all edge costs are equivalent***, because this reduces BFS to a special case of uniform cost search, which is discussed below.
- **Time Complexity:** We must search $1 + b + b^2 + \ldots + b^s$ nodes in the worst case, since we go through all nodes at every depth from 1 to s. Hence, the time complexity is $O\left(b^s\right)$
- **Space Complexity:** The fringe, in the worst case, contains all the nodes in the level corresponding to the shallowest solution. Since the shallowest solution is located at depth s, there are $O\left(b^s\right)$ nodes at this depth.

§1.6 Uniform Cost Search



- **Description:** Uniform cost search (UCS), our last strategy, is a strategy for exploration that always selects the lowest cost fringe node from the start node for expansion.
- **Fringe representation:** To represent the fringe for UCS, the choice is usually a heap-based priority queue, where the weight for a given enqueued node v is the path cost from the start node to v, or the backward cost of v. Intuitively, a priority queue constructed in this manner simply reshuffles itself to maintain the desired ordering by path cost as we remove the current minimum cost path and replace it with its children.
- **Completeness:** Uniform cost search is **complete**. If a goal state exists, it must have some finite length shortest path; hence, UCS must eventually find this shortest length path.
- **Optimality:** UCS is **optimal** if we assume **all edge costs are nonnegative**. By construction, since we explore nodes in order of increasing path cost, we're guaranteed to find the lowest-cost path to a goal state. The strategy employed in Uniform cost Search is identical to that of Dijkstra's algorithm, and the chief difference is that UCS terminates upon finding a solution state instead of finding the shortest path to all states. Note that having negative edge costs in our graph can make nodes on a path have decreasing length, ruining our guarantee of optimality. (See Bellman-Ford algorithm for a slower algorithm that handles this possibility)
- **Time Complexity:** Let us define the optimal path cost as C^* and the minimal cost between two nodes in the state space graph as ε. Then, we must roughly explore all nodes at depths ranging from 1 to C^*/ε, leading to an runtime of $O\left(b^{C^*/\varepsilon}\right)$
- **Space Complexity:** Roughly, the fringe will contain all nodes at the level of the cheapest solution, so the space complexity of UCS is estimated as $O\left(b^{C^*/\varepsilon}\right)$

§1.7 Greedy Search

- **Description:** Greedy search is a strategy for exploration that always selects the fringe node with the lowest heuristic value for expansion, which corresponds to the state it believes is nearest to a goal.
- **Fringe representation:** Greedy search operates identically to UCS, with a priority queue fringe representation. The difference is that instead of using computed

backward cost (the sum of edge weights in the path to the state) to assign priority, greedy search uses estimated forward cost in the form of heuristic values.

- **Completeness and Optimality:** Greedy search is **not complete** and **not optimal**, particularly in cases where a very bad heuristic function is selected. It generally acts fairly unpredictably from scenario to scenario, and can range from going straight to a goal state to acting like a badly-guided DFS and exploring all the wrong areas.

§1.8 A* Search

- **Description:** A* search is a strategy for exploration that always selects the fringe node with the lowest estimated total cost for expansion, where total cost is the entire cost from the start node to the goal node.
- **Fringe representation:** Just like greedy search and UCS, A^* search also uses a priority queue to represent its fringe. Again, the only difference is the method of priority selection. A^* combines the total backward cost (sum of edge weights in the path to the state) used by UCS with the estimated forward cost (heuristic value) used by greedy search by adding these two values, effectively yielding an estimated total cost from start to goal. Given that we want to minimize the total cost from start to goal, this is an excellent choice.
- **Completeness and Optimality:** A$*$ search is both complete and optimal, given an appropriate heuristic (which we'll cover in a minute). It's a combination of the good from all the other search strategies we've covered so far, incorporating the generally high speed of greedy search with the optimality and completeness of UCS!

§1.9 Admissibility and Consistency

Main idea: estimated heuristic costs $\leq$ actual costs.

Definition 1.2 (Admissibility)

For a heuristic h to be **admissible**, the following must be true:

$$\forall n, 0 \leq h(n) \leq h^*(n),$$

in other words, heuristic cost $\leq$ actual cost to goal.

Definition 1.3 (Consistency)

For a heuristic h to be **consistent**, we h underestimate the cost between all nodes:

$$h(A) - h(C) \leq \text{cost}(A \text{ to } C)$$

Consequences of consistency:

- The f value along a path never decreases

$$h(A) \leq \text{cost}(A \text{ to } C) + h(C)$$

- A^* graph search is optimal

Remark 1.4. Consistency implies admissibility, but admissibility does not imply consistency. Hence, **consistency** is a stronger condition.

Theorem 1.5

For a given search problem, if the **admissibility** constraint is satisfied by a heuristic function h using A^* **tree search** with h on that search problem will yield an optimal solution.

Proof. To be added. □

Theorem 1.6

For a given search problem, if the **consistency** constraint is satisfied by a heuristic function h using A^* **graph search** with h on that search problem will yield an optimal solution.

Proof. To be added. □

§2 Constraint Satisfaction Problems

Basic Ideas:

- A special subset of search problems.
- State is defined by variables X_i with values from a domain D (sometimes D depends on i).
- Goal test is a set of constraints specifying allowable combinations of values for subsets of variables.
- Consists of variables, domains, and constraints (unary, binary, higher-order).

§2.1 Backtracking Search

Backtracking search is the basic uninformed algorithm for solving CSPs.
Basic Ideas:

- One variable at a time.
- Check constraints as you go.
- Backtracking = DFS + above two improvements.

Backtracking can be inefficient, we can improve it by considering ordering and filtering.

§2.2 Filtering

- **Filtering:** keeps track of domains for unassigned variables and crosses off bad options.
- **Forward checking:** crosses off values that violate a constraint when added to the existing assignment, but doesn't provide early detection for all failures.

§2.3 Arc Consistency

- An arc $X \to Y$ is **consistent** iff for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint.
- Forward checking basically enforces consistency of arcs pointing to each new assignment.
- If X loses a value, neighbors of X need to be rechecked.
- Arc consistency detects failure earlier than forward checking.
- Can be run as a preprocessor or after each assignment.
- We delete values from the tail.
- The runtime for arc consistency is $O(n^2d^3)$.
- **Limitations:** Can have one solution left, multiple solutions left, or no solutions left (and not know it).
- Arc consistency still runs inside a backtracking search.

§2.3.1 K-Consistency

- Increasing degrees of consistency.
- 1-Consistency (Node Consistency): Each single node's domain has a value which meets that node's unary constraints.
- 2-Consistency (Arc Consistency): For each pair of nodes, any consistent assignment to one can be extended to the other.
- K-Consistency: For each k nodes, any consistent assignment to $k-1$ can be extended to the kth node.
- Higher k more expensive to compute.
- Arc consistency is the case when $k = 2$.

§2.4 Ordering

§2.4.1 Minimum Remaining Value (MRV)

- Choose the variable with the fewest legal left values in its domain.
- "Fail-fast" ordering.
- The most constrained variable is most likely to run out of possible values and result in backtracking if left unassigned, so it's best to assign a value as soon as possible.

§2.4.2 Least Constraining Value (LCV)

- Choose the **least constraining value**, i.e, the one that rules out the fewest values in the remaining variables.

§2.5 Local Search

- **Local search** is another widely used algorithm for solving CSP problems.
- Works by **iterative improvement:**
  - start with some random assignment to values then iteratively select a random conflicted variable and reassign its value to the ones that violates the fewest cnostraints until no more constraint violations exist (**min-conflicts heuristic**)
- In short, it improves a single option until you can't make it better (no fringe).
- Successor function: local changes.
- Much faster and more memory efficient (but incomplete and suboptimal).

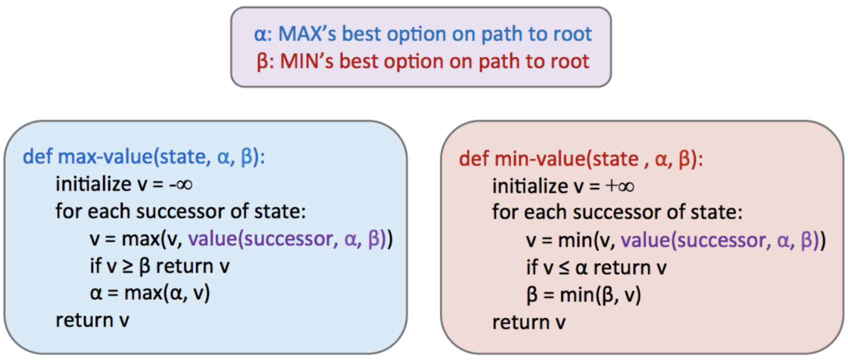
§3 Game Trees

- Zero-sum games (Opposite utilities)
- General games (Independent utilities)

§3.1 Alpha-Beta Pruning

- Has no effect on minimax value computed for the root.
- Values of intermediate nodes might be wrong
  - Important: children of the root may have the wrong value
- Good child ordering improves effectiveness of pruning
- With "perfect ordering":
  - Time complexity drops to $O(b^{m/2})$
  - Doubles solvable depth!

Implementation:



α: MAX's best option on path to root
β: MIN's best option on path to root

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

§3.2 Evaluation Function

Definition 3.1 (Evaluation Function)
Evaluation functions are functions that take in a state and output an estimate of the true minimax value of that node.

- Evaluation functions are widely employed in **depth-limited minimax**
- Because evaluation functions can only yield estimates of the values of non-terminal utilities, this *removes the guarantee of optimal play when running minimax.*

The most common design for an evaluation function is a linear combination of features.

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

Each $f_i(s)$ corresponds to a feature extracted from the input state s, and each feature is assigned a corresponding weight w_i.

Example 3.2

In a game of checkers we might construct an evaluation function with 4 features: number of agent pawns, number of agent kings, number of opponent pawns, and number of opponent kings.

- We select appropriate weights based loosely on their importance.
- It makes most sense to select positive weights for our agent's pawns/kings and negative weights for our opponents pawns/kings.
- The features corresponding to our agent's/opponent's kings deserve weights with greater magnitude than the features concerning pawns.

Below is a possible evaluation function that conforms to the features and weights we've just brainstormed:

$$Eval(s) = 2 \cdot kings(s) + pawns(s) - 2 \cdot opp\_kings(s) - opp\_pawns(s)$$

Keep in mind that the evaluation function **yields higher scores for better positions as frequently as possible**.

§3.3 Expectimax

We've now seen how minimax works and how running full minimax allows us to respond optimally against an optimal opponent. However, minimax has some natural constraints on the situations to which it can respond.

- Because minimax believes it is responding to an optimal opponent, it's often **overly pessimistic** in situations where optimal responses to an agent's actions are **not guaranteed**.
- Such situations include scenarios with inherent randomness such as card or dice games or unpredictable opponents that move randomly or suboptimally.

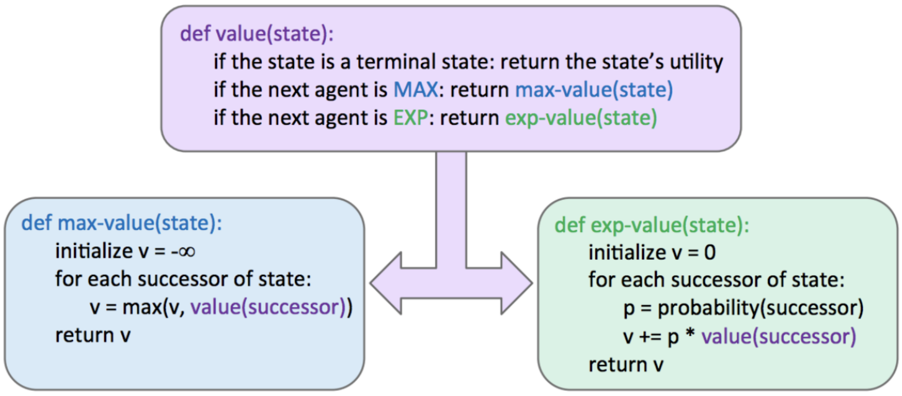
This randomness can be represented through a generalization of minimax known as **expectimax**.

- Expectimax introduces **chance nodes** into the game tree, which instead of considering the worst case scenario as minimizer nodes do, considers the **average case**.

Our rule for determining values of nodes with expectimax is as follows:

$$\begin{aligned}
\forall \text{ agent-controlled states, } V(s) &= \max_{s' \in \text{successors}(s)} V(s') \\
\forall \text{ chance states, } V(s) &= \sum_{s' \in \text{successors}(s)} p(s' \mid s) V(s') \\
\forall \text{ terminal states, } V(s) &= \text{ known}
\end{aligned}$$

The pseudocode for expectimax is quite similar to minimax, with only a few small tweaks to account for expected utility instead of minimum utility, since we're replacing minimizing nodes with chance nodes:



```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is EXP: return exp-value(state)

def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v

def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```

§3.4 Minimax vs Expectimax

- **Minimax:** Used when our opponent(s) behaves optimally, and can be optimized using $\alpha-\beta$ pruning. Minimax provides more **conservative** actions than expectimax, and so tends to yield **favorable** results when the opponent is unknown as well.
- **Expectimax:** Used when we facing a suboptimal opponent(s), using a probability distribution over the moves we believe they will make to compute the expected value of states.

§3.5 Utilities

Rational agents must follow the **principle of maximum utility** - they must always select the action that maximizes their expected utility. However, obeying this principle only benefits agents that have **rational preferences**.
Let's now properly define the mathematical language of preferences:

- If an agent prefers receiving a prize A to receiving a prize B, this is written $A \succ B$
- If an agent is indifferent between receiving A or B, this is written as $A \sim B$
- A lottery is a situation with different prizes resulting with different probabilities. To denote lottery where A is received with probability p and B is received with probability $(1-p)$, we write

$$L = [p, A; (1-p), B]$$

In order for a set of preferences to be rational, they must follow the five **Axioms of Rationality**:

Theorem 3.3 (Axioms of Rationality)

Orderability:

$$(A \succ B) \vee (B \succ A) \vee (A \sim B)$$

A rational agent must either prefer one of A or B, or be indifferent between the two.

Transitivity:

$$(A \succ B) \wedge (B \succ C) \Rightarrow (A \succ C)$$

If a rational agent prefers A to B and B to C, then it prefers A to C.

Continuity:

$$A \succ B \succ C \Rightarrow \exists p [p, A; (1-p), C] \sim B$$

If a rational agent prefers A to B but B to C, then it's possible to construct a lottery L between A and C such that the agent is indifferent between L and B with appropriate selection of p.

Substitutability:

$$A \sim B \Rightarrow [p, A; (1-p), C] \sim [p, B; (1-p), C]$$

A rational agent indifferent between two prizes A and B is also indifferent between any two lotteries which only differ in substitutions of A for B or B for A.

Monotonicity:

$$A \succ B \Rightarrow (p \geq q) \Leftrightarrow [p, A; (1-p), B] \succeq [q, A; (1-q), B]$$

If a rational agent prefers A over B, then given a choice between lotteries involving only A and B the agent prefers the lottery assigning the highest probability to A

If all five axioms are satisfied by an agent, then it's guaranteed that the agent's behavior is describable as a **maximization of expected utility**, which implies that there exists a real-valued **utility function** U that when implemented will assign greater utilities to preferred prizes, and also that the utility of a lottery is the expected value of the utility of the prize resulting from the lottery.

These two statements can be summarized in two concise mathematical equivalences:

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

§4 Non-deterministic Search

- Want to account for another influencing factor - **the dynamics of world itself**
- The environment in which an agent is placed may subject the agent's actions to being **nondeterministic**, which means that there are multiple possible successor states that can result from an action taken in some state
- Such problems where the world poses a degree of uncertainty are known as **nondeterministic search** problems, and can be solved with models known as **Markov decision processes**, or MDPs

§4.1 Markov Decision Processes

Definition 4.1 (Markov Decision Processes)
A Markov Decision Process is defined by several properties:

- **A set of states** S
- **A set of actions** A
- **A start state**
- **A start state**
- **Possibly one or more terminal states**
- **Possibly a discount factor** γ
- **A transition function** $T(s, a, s')$
  it's a probability function which represents the probability that an agent taking an action $a \in A$ from a state $s \in S$ ends up in a state $s' \in S$
- **A reward function** $R(s, a, s')$
  Typically, MDPs are modeled with small "living" rewards at each step to reward an agent's survival, along with large rewards for arriving at a terminal state. Rewards may be positive or negative depending on whether or not they benefit the agent in question, and the agent's objective is naturally to acquire the maximum reward possible before arriving at some terminal state.

The movement of an agent through a MDP can thus be modeled as follows:

$$s_0 \overset{a_0}{\to} s_1 \xrightarrow{a_1} s_2 \overset{a_2}{\to} s_3 \xrightarrow{a_3} \dots$$

Knowing that an agent's goal is to maximize it's reward across all timesteps, we can correspondingly express this mathematically as a maximization of the following utility function:

$$U([s_0, a_0, s_1, a_1, s_2, \dots]) = R(s_0, a_0, s_1) + R(s_1, a_1, s_2) + R(s_2, a_2, s_3) + \dots$$

§4.2 Markovianess

Markov decision processes satisfy the **Markov property**, or **memoryless property**.

Definition 4.2 (Markov Property)
The future and the past are conditionally independent, given the present.
Mathematically,

$$P\left(S_{t+1}=s_{t+1} \mid S_t=s_t, A_t=a_t, \ldots, S_0=s_0\right)=P\left(S_{t+1}=s_{t+1} \mid S_t=s_t, A_t=a_t\right)$$

where each S_t and A_t denote the random variables representing our agent's state and actions respectively during time t.

In other words, given present state, knowing the past doesn't give us any more information about the future. These memoryless probabilities are encoded by the transition function: $\boxed{T\left(s, a, s'\right)=P\left(s' \mid s, a\right)}$

§4.3 The Bellman Equation

In order to talk about the Bellman equation for MDPs, we must first introduce two new mathematical quantities:

- The optimal value of a state $s, V^*(s)$ - the optimal value of s is the expected value of the utility an optimally-behaving agent that starts in s will receive, over the rest of the agent's lifetime.
- The optimal value of a q-state $(s, a), Q^*(s, a)$ - the optimal value of (s, a) is the expected value of the utility an agent receives after starting in s, taking a, and acting optimally henceforth.

Using these two new quantities and the other MDP quantities discussed earlier, the Bellman equation is defined as follows:

Definition 4.3 (Bellman Equation)

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

Let's also define the equation for the optimal value of a q-state (more commonly known as an optimal q-value):

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

Note that this second definition allows us to reexpress the Bellman equation as

Definition 4.4 (Bellman Equation 2.0)

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

The term $[R(s,a,s') + \gamma V^*(s')]$ represents the utility attained by acting optimally after arriving in state s' from q-state (s,a), so

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

is simply a weighted sum of utilities, with each utility weighted by its probability of occurrence.

Bellman's usage is as a condition for optimality. In other words, if we can somehow determine a value $V(s)$ for every state $s \in S$ such that the Bellman equation holds true for each of these states, we can conclude that these values are the optimal values for their respective states. Indeed, satisfying this condition implies $\forall s \in S, V(s) = V^*(s)$

§4.4 Value Iteration

Value iteration is a dynamic programming algorithm that uses an iteratively longer time limit to compute time-limited values until convergence (that is, until the V values are the same for each state as they were in the past iteration: $\forall s, V_{k+1}(s) = V_k(s)$). It operates as follows:

1. $\forall s \in S$, initialize $V_0(s) = 0$. This should be intuitive, since setting a time limit of 0 timesteps means no actions can be taken before termination, and so no rewards can be acquired.

2. Repeat the following update rule until convergence:

Definition 4.5 (Value Iteration)

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

Note that though the Bellman equation looks essentially identical in construction to the update rule above, they are not the same. The Bellman equation gives a condition for optimality, while the update rule gives a method to iteratively update values until convergence. When convergence is reached, the Bellman equation will hold for every state:

$$\forall s \in S, V_k(s) = V_{k+1}(s) = V^*(s).$$

§4.5 Policy Extraction

Recall that the ultimate goal in solving a MDP is to determine an optimal policy. This can be done once all optimal values for states are determined using **policy extraction**.

The intuition behind is very simple: if you're in a state s, you should take the action a which yields the maximum expected utility. Not surprisingly, a is the action which takes us to the q-state with maximum q-value, allowing for a formal definition of the optimal policy:

Definition 4.6

$$\forall s \in S, \pi^*(s) = \underset{a}{\operatorname{argmax}} Q^*(s,a) = \underset{a}{\operatorname{argmax}} \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma V^*(s')\right]$$

It's useful to keep in mind for performance reasons that it's better for policy extraction to have the optimal q-values of states, in which case a single argmax operation is all that is required to determine the optimal action from a state. Storing only each $V^*(s)$ means that we must recompute all necessary q-values with the Bellman equation before applying argmax, equivalent to performing a depth-1 expectimax.

§4.6 Policy Iteration

Cons for value iteration:

- must update the values of all $|S|$ states (where $|n|$ refers to the cardinality operator), each of which requires iteration over all $|A|$ actions as we compute the q-value for each action.
- The computation of each of these q-values, in turn, requires iteration over each of the $|S|$ states again, leading to a poor runtime of $O\left(|S|^2|A|\right)$.
- overcomputing since the policy as computed by policy extraction generally converges significantly faster than the values themselves.

Policy iteration operates as follows:

1. Define an initial policy. This can be arbitrary, but policy iteration will converge faster the closer the initial policy is to the eventual optimal policy.
2. Repeat the following until convergence:
   - Evaluate the current policy with **policy evaluation**. For a policy π, policy evaluation means computing $V^\pi(s)$ for all states s, where $V^\pi(s)$ is expected utility of starting in state s when following π :

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

     Define the policy at iteration i as π_i. since we are fixing a single action for each state, we no longer need the max operator which effectively leaves us with a system of $|S|$ equations generated by the above rule. Each $V^{\pi_i}(s)$ can then be computed by simply solving this system. Alternatively, we can also compute $V^{\pi_i}(s)$ by using the following update rule until convergence, just like in value iteration:

     $$V_{k+1}^{\pi_i}(s) \leftarrow \sum_{s'} T(s,\pi_i(s),s')\left[R(s,\pi_i(s),s') + \gamma V_k^{\pi_i}(s')\right]$$

     However, this second method is typically slower in practice.
   - Once we've evaluated the current policy, use **policy improvement** to generate a better policy. **Policy improvement** uses policy extraction on the values

of states generated by policy evaluation to generate this new and improved policy:

$$\pi_{i+1}(s) = \underset{a}{\operatorname{argmax}} \sum_{s'} T\left(s, a, s'\right)\left[R\left(s, a, s'\right) + \gamma V^{\pi_i}\left(s'\right)\right]$$

If $\pi_{i+1} = \pi_i$, the algorithm has converged, and we can conclude that $\pi_{i+1} = \pi_i = \pi^*$

§4.7 Summary

- **Value iteration**: Used for computing the **optimal values of states**, by iterative updates until convergence.
- **Policy evaluation**: Used for computing the values of states under a **specific policy**.
- **Policy extraction**: Used for determining a policy given some state value function. If the state values are optimal, this policy will be optimal. This method is **used after running value iteration**, to compute an **optimal policy** from the optimal state values; or as a subroutine in policy iteration, to compute the best policy for the currently estimated state values.
- **Policy iteration**: A technique that encapsulates both policy evaluation and policy extraction and is used for iterative convergence to an optimal policy. It **tends to outperform value iteration**, by virtue of the fact that policies usually converge much faster than the values of states.

§5 Reinforcement Learning

Basic idea:

- Receive feedback in the form of rewards
- Agent's utility is defined by the **reward function**
- Must learn to act so as to **maximize expected rewards**
- All learning is based on observed samples of outcomes
- Still assumes a MDP(Online/Offline)
- Still looking for a policy
- **New twist: don't know T or R**

§5.1 Model-Based Learning

- generates an approximation of the transition function $\hat{T}(s, a, s')$ by counting and normalization (to be interpreted as probabilities)
- generates a policy $\pi_{exploit}$ by running value or policy iteration with our current models for $\hat{T}$ and $\hat{R}$ and use $\pi_{exploit}$ for exploitation
- traverses the MDP taking actions seeking reward maximization rather than seeking learning
- can be **expensive** to maintain counts

Hence, we need **model-free learning** to avoid counts and the memory overhead required by model-based learning.

§5.2 Model-Free Learning

§5.2.1 Algorithms Categories:

- **Passive Reinforcement Learning:** agent is given a policy to follow and learns the value of states under that policy as it experiences episodes, similar to policy evaluation for MDPs except that T and R are unknown.
- **Active Reinforcement Learning:** agent uses the feedback received to iteratively update policy while learning until eventually determining the optimal policy after sufficient exploration.

§5.2.2 Algorithms:

- **Direct Evaluation** (passive)
- **Temporal Difference Learning** (passive)
- **Q-Learning** (active)
- **Approximate Q-Learning** (active)

§5.2.3 Direct Evaluation

Pros:

- easy to understand
- doesn't require any knowledge of T, R
- eventually computes the correct average values, using just sample transitions

Cons:

- wastes information about state connections
- each state must be learned separately, so it takes a long time to learn

§5.2.4 Temporal Difference Learning

Basic Ideas:

- **learning from every experience**, rather than simply keeping track of total rewards and number of times states are visited and learning at the end as direct evaluation does
- policy still fixed, still doing evaluation
- converges to learning true state values much faster with fewer episodes than direct evaluation
- move values toward value of whatever successor occurs: running average

Definition 5.1 (TD Learning)

$$\begin{aligned} &\text{Sample of V(s)}: && \text{sample } = R\left(s, \pi(s), s'\right) + \gamma V^{\pi}\left(s'\right) \\ &\text{Update to V(s)}: && V^{\pi}(s) \leftarrow (1-\alpha) V^{\pi}(s) + (\alpha) \text{ sample} \\ &\text{Same update:} && V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha\left(\text{sample} - V^{\pi}(s)\right) \end{aligned}$$

Cons:

- less stable and may converge to the wrong solution
- can't turn values into (new) policy

§5.2.5 Q-Learning

Major issue with direct evaluation and TD learning: optimal policy requires knowledge of the q-values of states.

To compute q-values from the values we have, we require a transition function and reward function as dictated by the Bellman equation.

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

Basic Idea:

- learn the q-values of states directly, bypassing the need to ever know any values, transition functions, or reward functions
- uses **Q-Value iteration**:

Definition 5.2 (Q-Value Iteration)

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

- derived the same way as TD learning, but by acquiring q-value samples:

Definition 5.3 (Q-Learning)

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

and incoporating them into an exponential moving average.

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

- converges to optimal policy even acting suboptimally, i.e. **off-policy learning**

Caveats:

- need to explore enough
- the learning rate α has to be small enough eventually but not decrease it too quickly
- in the limit, it doesn't matter what actions are selected
- impossible to visit all states and store all Q-values

§5.2.6 Approximate Q-Learning

Basic Ideas:

- **feature-based representation** of states, which represents each state as a vector known as a **feature vector**.
- write a q function (or value function) for any state using a few weights:

Definition 5.4 (Linear Value Functions)

$$V(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$
$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \ldots + w_n f_n(s,a)$$

Advantage: experience is summed up in a few powerful numbers
Disadvantage: states may share features but actually be very different in value!

- Q-learning with linear Q-functions:

Definition 5.5 (Approximate Q-Learning)

$$\begin{aligned} \text{transition } &= (s, a, r, s') \\ \text{difference } &= \left[r + \gamma \max_{a'} Q\left(s', a'\right)\right] - Q(s,a) \end{aligned}$$

$$Q(s,a) \leftarrow Q(s,a) + \alpha[\text{ difference }] \quad \textbf{Exact Q's}$$

$$w_i \leftarrow w_i + \alpha[\text{ difference }] f_i(s,a) \quad \textbf{Approximate Q's}$$

- **Intuitive interpretation:** Adjust weights of active features. E.g., if something unexpectedly bad happens, blame the features that were on: disprefer all states with that state's features Formal justification: online least squares

§5.3 Exploration and Exploitation

§5.3.1 ε-Greedy Policies

- acts randomly and explores with probability ε
- follows current established policy and exploit with probability $(1-\varepsilon)$
- large value for $\varepsilon \implies$ the agent will still behave mostly randomly
- a small value for $\varepsilon \implies$ the agent will explore infrequently $\implies$ learn the optimal policy very slowly
- hence, ε must be manually tuned and lowered over time

§5.3.2 Exploration Functions

- avoids the trouble of tuning ε manually
- use a modified q-value iteration update to give some preference to visiting less-visited states. The modified update is as follows:

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

- there exists some degree of flexibility in designing an exploration function, but a common choice is to use

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

  with k being some predetermined value, and $N(s,a)$ denoting the number of times q-state (s,a) has been visited.

§5.4 Summary

Reinforcement learning has an **underlying MDP**, and the goal of reinforcement learning is to solve this MDP by deriving an optimal policy. The difference is the **lack of knowledge of the transition function** T **and the reward function** R for the underlying MDP. As a result, agents must learn the optimal policy through online trial-by-error rather than pure offline computation. There are many ways to do this:

- **Model-based learning:** Runs computation to estimate the values of the transition function T and the reward function R and uses MDP-solving methods like value or policy iteration with these estimates.
- **Model-free learning:** Avoids estimation of T and R, instead using other methods to directly estimate the values or q-values of states.
  - **Direct evaluation:** follows a policy π and counts total rewards reaped from each state and the total number of times each state is visited. If enough samples are taken, this converges to the true values of states under π, albeit being ***slow and wasting information about the transitions between states***.
  - **Temporal difference learning:** follows a policy π and uses an ***exponential moving average*** with sampled values until convergence to the true values of states under π. TD learning and direct evaluation are examples of ***on-policy learning***, which learn the values for a specific policy before deciding whether that policy is suboptimal and needs to be updated.
  - **Q-Learning:** learns the optimal policy ***directly through trial and error*** with q-value iteration updates. This an example of **off-policy learning**, which learns an optimal policy ***even when taking suboptimal actions***.
  - **Approximate Q-Learning:** does the same thing as Q-learning but uses a ***feature-based representation*** for states to generalize learning.

§6 Probability

§6.1 Probabilistic Inference

Definition 6.1 (Probabilistic Inference)

Probabilistic inference is the task of deriving a desired probability from other known probabilities (e.g. conditional from joint).

- Given joint PDF, we can trivially perform compute any desired probablity distribution $P\left(Q_1 \ldots Q_k \mid e_1 \ldots e_k\right)$ using **inference by enumeration**, which includes three types of variables:
  1. **Query variables** Q_i: unknown values
  2. **Evidence variables** e_i: observed variables with known values
  3. **Hidden variables**: values present in the overall joint distribution but not in the distribution we are currently trying to compute.

  In this procedure, we collect all the rows consistent with the observed evidence variables, sum out all hidden variables, then normalize the table so that it is a probability distribution (i.e. values sum to 1).

Theorem 6.2 (Product Rule)

$$P(y)P(x \mid y) = P(x, y)$$

Theorem 6.3 (Chain Rule)

$$P\left(x_1, x_2, \ldots x_n\right) = \prod_i P\left(x_i \mid x_1 \ldots x_{i-1}\right)$$

Theorem 6.4 (Bayes' Rule)

$$P(x \mid y) = \frac{P(y \mid x)P(x)}{P(y)}$$

Theorem 6.5 (Conditional Independence)
X is **conditionally independent** of Y given Z

$$X \perp\!\!\!\perp Y \mid Z$$

if and only if:

$$\forall x, y, z : P(x, y \mid z) = P(x \mid z)P(y \mid z)$$

or, equivalently, if and only if

$$\forall x, y, z : P(x \mid z, y) = P(x \mid z)$$

§6.2 Bayes' Nets (Representation)

Definition 6.6 (Bayes' Nets)
Bayes' nets is a technique for describing complex joint distributions (models) using simple, local distributions (conditional probabilities)

Basic Ideas:

- a set of nodes, one per variable X
- directed, acyclic graph
- a conditional distribution (**CPT**) for each node
- each node is conditionally independent of all its ancestor nodes in the graph, given all of its parents
- given all CPTs for a graph, we can calculate the probability of a given assignment using chain rule:

$$P(x_1, x_2, \dots, x_n) = \prod_{i=1}^{n} P(x_i|\text{parents}(X_i))$$

§6.3 Bayes' Nets (Inference)

Definition 6.7 (Inference)
Inference is the process of calculating the joint PDF for some set of query variables based on some set of observed variables.

- can be solved by forming joint PDF and using inference by enumeration, which requires creation of and iteration over an exponentially large table
- alternate approach: eliminate variables one by one:
  1. join (multiply together) all factors involving X.
  2. Sum out X.

Definition 6.8 (factor)
A **factor** is an **unnormalized** probability.

§6.3.1 Variable Elimination

Basic Ideas:

- eliminate by joining and summing over the variables (marginalizing)
- NP-hard, but usually much faster than **inference by enumeration**
- order of elimination affects efficiency (eliminate leaf nodes first usually)

§6.4 Bayes' Nets (Sampling)

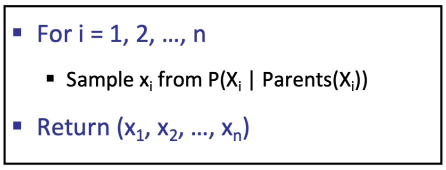
Basic Ideas:

- repeated simulation
- draw N samples from a sampling distribution S
- compute an approximate posterior probability
- show this converges to the true probability P

Why sample?

- Learning: get samples from an unknown distribution
- Inference: getting a sample is faster than computation (e.g. with variable elimination)

§6.4.1 Prior Sampling



```
For i = 1, 2, …, n
    Sample $x_i$ from $P(X_i \mid \text{Parents}(X_i))$
Return $(x_1, x_2, …, x_n)$
```

- This process generates samples with probability:

$$S_{PS}(x_1 \ldots x_n) = \prod_{i=1}^{n} P(x_i \mid \text{ Parents } (X_i)) = P(x_1 \ldots x_n)$$

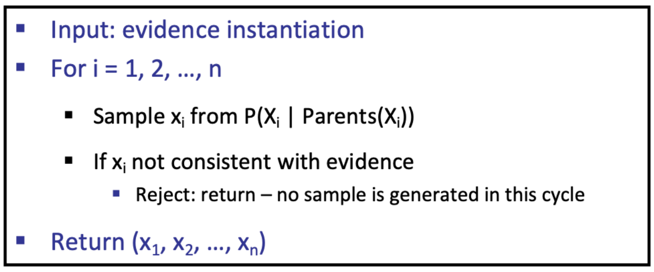
- Let the number of samples of an event be $N_{PS}(x_1 \ldots x_n)$. Then

$$\begin{aligned}\lim_{N\to\infty} \widehat{P}(x_1,\ldots,x_n) &= \lim_{N\to\infty} N_{PS}(x_1,\ldots,x_n)/N \\ &= S_{PS}(x_1,\ldots,x_n) \\ &= P(x_1 \ldots x_n)\end{aligned}$$

  i.e., the sampling procedure is ***consistent***.

Remark 6.9. However, this method may require the generation of a very large number of samples in order to perform analysis of unlikely scenarios.

§6.4.2 Rejection Sampling

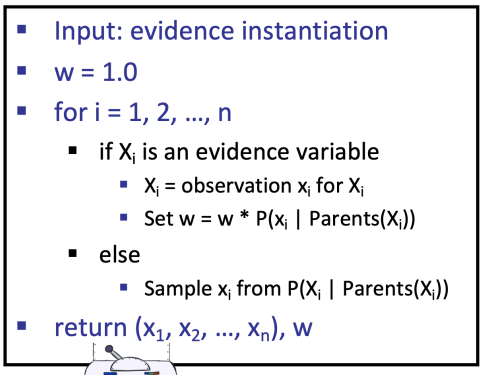


```
▪ Input: evidence instantiation
▪ For i = 1, 2, …, n
    ▪ Sample $x_i$ from $P(X_i \mid Parents(X_i))$
    ▪ If $x_i$ not consistent with evidence
        ▪ Reject: return – no sample is generated in this cycle
▪ Return $(x_1, x_2, …, x_n)$
```

- Modified prior sampling with sample rejections
- Reject samples that are inconsistent with evidence

Remark 6.10. If evidence is unlikely, we will end up rejecting lots of samples.

§6.4.3 Likelihood Sampling



```
▪ Input: evidence instantiation
▪ w = 1.0
▪ for i = 1, 2, …, n
    ▪ if $X_i$ is an evidence variable
        ▪ $X_i$ = observation $x_i$ for $X_i$
        ▪ Set w = w * $P(x_i \mid Parents(X_i))$
    ▪ else
        ▪ Sample $x_i$ from $P(X_i \mid Parents(X_i))$
▪ return $(x_1, x_2, …, x_n)$, w
```

- Set all variables equal to the evidence in our query
- Weight by probability of evidence given parents to make sampling distribution consistent
- Sampling distribution if z sampled and e fixed evidence

$$S_{WS}(\mathrm{z}, \mathrm{e}) = \prod_{i=1}^{l} P\left(z_i \mid \text{ Parents } \left(Z_i\right)\right)$$

with weights

$$w(\mathrm{z}, \mathrm{e}) = \prod_{i=1}^{m} P\left(e_i \mid \text{ Parents } \left(E_i\right)\right)$$

Together, weighted sampling distribution is consistent

$$\begin{aligned} S_{\mathrm{WS}}(z, e) \cdot w(z, e) &= \prod_{i=1}^{l} P\left(z_i \mid \text{Parents}\left(z_i\right)\right) \prod_{i=1}^{m} P\left(e_i \mid \text{Parents}\left(e_i\right)\right) \\ &= P(\mathbf{z}, \mathbf{e}) \end{aligned}$$

- Take evidence into account
- Most samples will reflect the state of the world suggested by the evidence

Remark 6.11. Evidence influences the choice of **downstream** variables, but not upstream ones. Weights obtained in likelihood weighting can sometimes be very small. Sum of weights over all samples is indicative of how many "effective" samples were obtained, so we want high weight.

§6.4.4 Gibbs Sampling

- 1. Fix evidence $R = +r$
  2. Initialize other variables randomly
  3. Repeat:
     Choose a non-evidence variable X and resample X from

$$P(X|\text{all other variables})$$

- Later samples will eventually **converge** to the correct distribution
- Both upstream and downstream variables condition on evidence.

§6.5 D-Separation

Basic Ideas:

- Two nodes A and B are guaranteed to be independent according to the if there exist no **active** paths between them
- If there exists **at least one** active path, no independence guarantees can be made

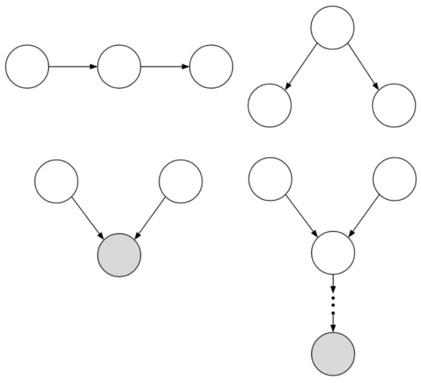


Figure 8: Active triples

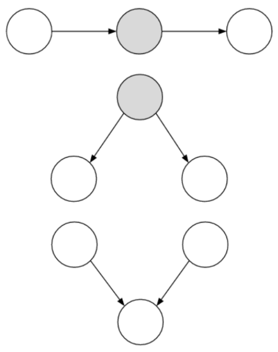


Figure 9: Inactive triples

§6.6 Summary

Bayes' Nets is a powerful representation of joint probability distributions. Its topological structure encodes independence and conditional independence relationships that can be used to model arbitrary distributions to perform inference and sampling.

§7 Markov Models

§7.1 Hidden Markov Models

Basic Ideas

- Markov Models with observations as hidden variables
- Contains state variables and evidence variables
- Represented compactly with initial distribution, transition model, and sensor model

Definition 7.1 (Belief Distribution)
The **belief distribution** at time t with all evidence $E_1, \ldots, E_t$ observed up to date is defined as

$$B(X_t) = P(X_t \mid e_1, \ldots, e_t)$$

Similarly, we denote $B'(X_t)$ as:

$$B'(X_t) = P(X_t \mid e_1, \ldots, e_{t-1})$$

Notation 7.2 ($e_{1:t}$). e_i represents evidence observed at timestep i,

$$e_{1:t} = e_1, \ldots, e_t$$

Definition 7.3 (Stationary Distribution)
Stationary distribution is the distribution that remains unchanged as time progresses, i.e.,

$$P(X_{t+1}) = P(X_t)$$

§7.1.1 Mini-Forward Algorithm

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

§7.1.2 The Forward Algorithm

Time Elapse Update

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

which can be simplified to

$$\boxed{B'(X_{t+1}) = \sum_{x_t} P(X_{t+1} \mid x_t) B(x_t)}$$

- As time passes, uncertainty accumulates

Observation Update

- Assume we have current belief $P(X|\text{previous evidence})$:

$$B'\left(X_{t+1}\right)=P\left(X_{t+1} \mid e_{1:t}\right)$$

Then, after evidence comes in:

$$\begin{aligned} P\left(X_{t+1} \mid e_{1:t+1}\right) &= P\left(X_{t+1}, e_{t+1} \mid e_{1:t}\right) / P\left(e_{t+1} \mid e_{1:t}\right) \\ &\propto_{X_{t+1}} P\left(X_{t+1}, e_{t+1} \mid e_{1:t}\right) \\ &= P\left(e_{t+1} \mid e_{1:t}, X_{t+1}\right) P\left(X_{t+1} \mid e_{1:t}\right) \\ &= P\left(e_{t+1} \mid X_{t+1}\right) P\left(X_{t+1} \mid e_{1:t}\right) \end{aligned}$$

which can be simplified to

$$\boxed{B\left(X_{t+1}\right) \propto_{X_{t+1}} P\left(e_{t+1} \mid X_{t+1}\right) B'\left(X_{t+1}\right)}$$

- Beliefs reweighted by likelihood of evidence
- Have to renormalize
- Uncertainty decreases as we get more observations

$$\boxed{B(X) \propto P(e \mid X) B'(X)}$$

§7.1.3 Viterbi Algorithm

- Solves for the most likely sequence of hidden states given the observed variables so far
  - Compute probability of best path to (state, time) tuple given evidence observed so far
  - Find terminal state on path with highest probability, traverse backward to return path
- Want to find maximum likelihood estimate of the sequence of hidden states:

$$\boxed{\arg \max_{x_1,\ldots,x_N} P\left(x_{1:N}, e_{1:N}\right)}$$

- Dynamic programming:
  - Define $m_t\left[x_t\right]=\max_{x_{1:t-1}} P\left(x_{1:t}, e_{1:t}\right)$, the maximum probability of a path starting at any x_0 and the evidence seen so far to a given x_t at time t.

$$\begin{aligned} m_t\left[x_t\right] &= \max_{x_{1:t-1}} P\left(e_t \mid x_t\right) P\left(x_t \mid x_{t-1}\right) P\left(x_{1:t-1}, e_{1:t-1}\right) \\ &= P\left(e_t \mid x_t\right) \max_{x_{t-1}} P\left(x_t \mid x_{t-1}\right) \max_{x_{1:t-2}} P\left(x_{1:t-1}, e_{1:t-1}\right) \\ &= P\left(e_t \mid x_t\right) \max_{x_{t-1}} P\left(x_t \mid x_{t-1}\right) m_{t-1}\left[x_{t-1}\right] \end{aligned}$$

Theorem 7.4 (Viterbi Algorithm)

$$m_t\left[x_t\right]=\max_{x_{1:t-1}} P\left(x_{1:t}, e_{1:t}\right)=P\left(e_t \mid x_t\right) \max_{x_{t-1}} P\left(x_t \mid x_{t-1}\right) m_{t-1}\left[x_{t-1}\right]$$

§7.2 Particle Filtering

Basic Ideas:

- simulate motion of a set of particles through a state graph to approximate belief distribution
- stores a list of particle positions instead of a full probability table

§7.2.1 Particle Filtering Simulation

- Particle initialization by sampling particles randomly, uniformly, or from some initial distribution
- Similar to forward algorithm, with a time elapse update followed by an observation update at each timestep:
  - **Time Elapse Update:** Update the value of each particle according to transition model $\Pr\left(T_{i+1} \mid t_i\right)$.
  - **Observation Update:** Use sensor model $P\left(F_i \mid T_i\right)$ to weight a particle in state t_i with sensor reading f_i, assign a weight of $P\left(f_i \mid t_i\right)$. The algorithm is as follows:
    1. Calculate the weights of all particles as described above.
    2. Calculate the total weight for each state.
    3. If the sum of all weights across all states is 0 , reinitialize all particles.
    4. Else, normalize the distribution of total weights over states and resample your list of particles from this distribution.

Remark 7.5. Note the similarity of the observation update to likelihood weighting, both downweight samples based on evidence.

§7.3 Summary

- **Markov models:** encode time-dependent random variables that possess the Markov property. We can compute a belief distribution at any timestep of our choice for a Markov model using probabilistic inference with the mini-forward algorithm.
- **Hidden Markov Models:** Markov models with the additional property that new evidence which can affect our belief distribution can be observed at each timestep. To compute the belief distribution at any given timestep with Hidden Markov Models, we use the forward algorithm.

Remark 7.6. Sometimes, running exact inference on these models can be too computationally expensive, in which case particle filtering is a better method to approximate inference.

§8 Decision Networks

Basic Ideas:

- Bayes nets with nodes for utility and actions
- Model effect of various actions on utilities based on an overarching graphical probabilistic model
- **Chance nodes(ovals):** each outcome in a chance node has an associated probability, which can be determined by running inference on the underlying Bayes' net it belongs to
- **Action nodes(rectangles):** nodes representing a choice between any of a number of actions which we have the power to choose from
- **Utility nodes(diamonds):** output a utility based on the values taken on by their parents
- **Expected utility** of taking an action $A = a$ given evidence $E = e$ and n chance nodes:
$$EU(a \mid e) = \sum_{x_1,\ldots,x_n} P(x_1,\ldots,x_n \mid e) U(a, x_1,\ldots,x_n)$$
where each x_i represents a value that the i^{th} chance node can take on

Definition 8.1 (Maximum Expected Utility (MEU))
Maximum Expected Utility is the expected utility of the action that has the highest expected utility:
$$\boldsymbol{MEU}(\boldsymbol{E} = \boldsymbol{e}) = \max_a \boldsymbol{EU}(A = a \mid \boldsymbol{E} = e)$$

§8.1 Value of Perfect Information

Definition 8.2 (Value of Perfect Information (VPI))
Value of Perfect Information mathematically quantifies the amount an agent's maximum expected utility is expected to increase if it observes some new evidence.

$$MEU(e) = \max_a \sum_s P(s \mid e) U(s,a)$$
$$MEU(e,e') = \max_a \sum_s P(s \mid e,e') U(s,a)$$
$$MEU(e,E') = \sum_{e'} P(e' \mid e) MEU(e,e')$$
$$\boxed{VPI(E' \mid e) = MEU(e,E') - MEU(e)}$$

Properties:

- **Nonnegativity:**
$$\forall E', e : VPI(E'|e) \geq 0$$
Observing new information $\Longrightarrow$ more informed decision, so MEU can only increase (or stay the same if the information is irrelevant

- **Nonadditivity:**

  $$\text{VPI}\left(E_j, E_k \mid e\right) \neq \text{VPI}\left(E_j \mid e\right) + \text{VPI}\left(E_k \mid e\right)$$

  VPI of observing two new evidence variables is equivalent to observing one, incorporating it into our current evidence, then observing the other. This is encapsulated by the order-independence property of VPI, described more below.

- **Order-independence:**

  $$\text{VPI}\left(E_j, E_k \mid e\right) = \text{VPI}\left(E_j \mid e\right) + \text{VPI}\left(E_k \mid e, E_j\right) = \text{VPI}\left(E_k \mid e\right) + \text{VPI}\left(E_j \mid e, E_k\right)$$

  Observing multiple new evidences yields the same gain in maximum expected utility regardless of the order of observation.

§9 Machine Learning

Algorithms:

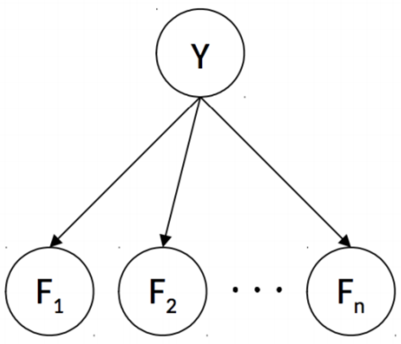
- Supervised learning algorithms
- Unsupervised learning algorithms

Dataset types:

- **Training data**: used to generate a model mapping inputs to outputs
- **Validation data** (hold-out/development data): used to measure model's performance by making predictions on inputs and generating an accuracy score
- **Test set**: portion of unseen data and is the equivalent of a "final exam" to gauge performance on real-world data
- **Hyperparameters**: model-specific values

§9.1 Naïve Bayes

- A a specific type of model for solving **classification problems**
- Assume all features are **independent** effects of the label



$$P(Y, F_1 \dots F_n) = P(Y) \prod_i P(F_i \mid Y)$$

$$\begin{aligned}
\text{prediction}(f_1, \cdots f_n) &= \underset{y}{\operatorname{argmax}} P(Y = y \mid F_1 = f_1, \dots F_N = f_n) \\
&= \underset{y}{\operatorname{argmax}} P(Y = y, F_1 = f_1, \dots F_N = f_n) \\
&= \underset{y}{\operatorname{argmax}} P(Y = y) \prod_{i=1}^{n} P(F_i = f_i \mid Y = y)
\end{aligned}$$

§9.2 Parameter Estimation

- Estimating the distribution of a random variable parametrized by unknown θ
- Learn the most likely value of θ given sample

§9.2.1 Maximum Likelihood Estimation

- Maximize the likelihood of the data
- Each identically distributed sample x_i is conditionally independent of the others given θ (**i.i.d**)

Definition 9.1 (Likelihood function)

$$\mathscr{L}(\boldsymbol{\theta}) = P_\theta(x_1, \ldots, x_N)$$

Since the samples x_i are i.i.d., we rewrite the above as:

$$\mathscr{L}(\theta) = \prod_{i=1}^{N} P_\theta(x_i)$$

- Relative frequencies are the maximum likelihood estimates

$$\begin{aligned} \theta_{ML} &= \arg\max_\theta P(\mathbf{X} \mid \theta) \\ &= \arg\max_\theta \prod_i P_\theta(X_i) \end{aligned} \longrightarrow P_{\mathrm{ML}}(x) = \frac{\mathrm{count}(x)}{\text{total samples}}$$

- The maximum likelihood estimate for θ is a value that satisfies

$$\frac{\partial}{\partial\theta}\mathscr{L}(\theta_{ML}) = 0$$

§9.3 Laplace Smoothing

- Mitigates the problem of **overfitting** (doesn't generalize well to previously unseen data)
- Laplace smoothing with **strength** k assumes having seen k extra of each outcome
- For a given sample the MLE for an outcome x that can take on $|X|$ different values from a sample of size N is

$$P_{MLE}(x) = \frac{\mathrm{count}(x)}{N}$$

then the Laplace estimate with strength k is

Theorem 9.2 (Laplace Smoothing)
General:

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

Conditional:

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

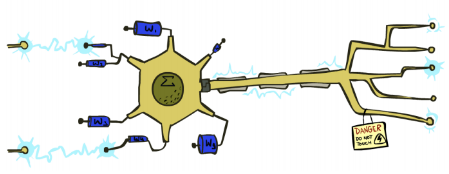
- Special cases ($k = 0, k = \infty$):

$$P_{LAP,0}(x) = P_{MLE}(x)$$

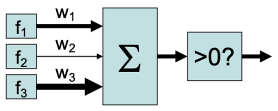
$$P_{LAP,\infty}(x) = \frac{1}{|X|}$$

- k is an hyperparameter typically determined by trial-and-error

§9.4 Perceptron



- Inputs are feature values
- Each feature has a weight
- Sum is the activation



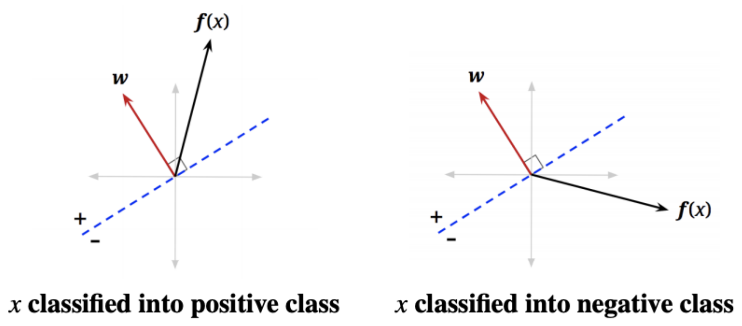
$$activation_w(x) = \sum_i w_i \cdot f_i(x) = \mathbf{w} \cdot \mathbf{f}(\mathbf{x})$$

$$\mathbf{w} \cdot \mathbf{f}(x) = \|\mathbf{w}\| \|\mathbf{f}(\mathbf{x})\| \cos(\theta)$$

$$\text{classify}(x) = \begin{cases} + & \text{if } \cos(\theta) > 0 \\ - & \text{if } \cos(\theta) < 0 \end{cases}$$

$$\text{classify}(x) = \begin{cases} + & \text{if } \theta < \frac{\pi}{2} \quad \text{(i.e. when } \theta \text{ is less than } 90^\circ, \text{ or acute)} \\ - & \text{if } \theta > \frac{\pi}{2} \quad \text{(i.e. when } \theta \text{ is greater than } 90^\circ, \text{ or obtuse)} \end{cases}$$

- **Decision boundary**(blue dotted line) orthogonal to $\mathbf{w}$.


f(x)
w
+
-
x classified into positive class
w
f(x)
+
-
x classified into negative class

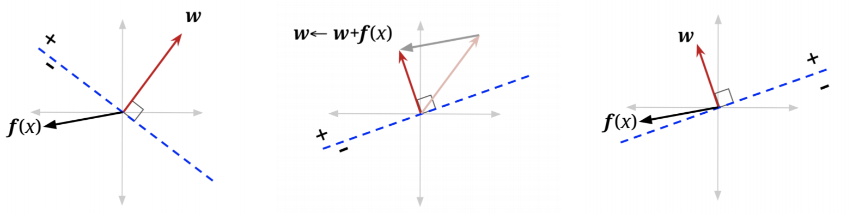
§9.4.1 Binary Decision Rule

Weight Updates:

- Start with $\mathbf{w} = \mathbf{0}$
- For each training instance:
  Classify by dot product:

$$y = \begin{cases} +1 & \text{if } \mathbf{w} \cdot \mathbf{f}(\mathbf{x}) \geq 0 \\ -1 & \text{if } \mathbf{w} \cdot \mathbf{f}(\mathbf{x}) < 0 \end{cases}$$

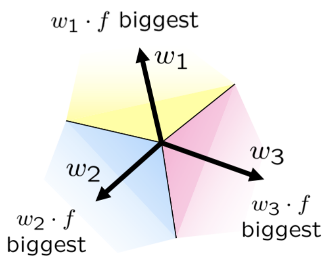
- If correct $(y = y^*)$, no change.
- If wrong: (add if $y^* = 1$, subtract if $y^* = -1$)



Mis-classifying x with old w **Updating w** **Updated classification of x**

$$\boxed{\mathbf{w} = \mathbf{w} \pm y^* \cdot \mathbf{f}}$$

§9.4.2 Multiclass Decision Rule



- A weight vector $\mathbf{w}_y$ for each class
- Activation of a class y:

$$\mathbf{w}_y \cdot \mathbf{f}(\mathbf{x})$$

- Prediction highest score(activation) wins

$$y = \arg\max_y \mathbf{w}_y \cdot \mathbf{f}(\mathbf{x})$$

Weight Updates:

- Start with all weights $= 0$
- Pick up training examples one by one
- Predict with current weights

$$y = \arg\max_y \mathbf{w}_y \cdot \mathbf{f}(\mathbf{x})$$

- If correct, no change
- If wrong: lower score of wrong answer, raise score of right answer

$$\mathbf{w}_y = \mathbf{w}_y - \mathbf{f}(\mathbf{x})$$
$$\mathbf{w}_{y^*} = \mathbf{w}_{y^*} + \mathbf{f}(\mathbf{x})$$

§9.4.3 Properties of Perceptrons

- **Separability:** true if some parameters get the training set perfectly correct
- **Convergence:** if the training is separable, perceptron will eventually converge (binary case)
- **Mistake Bound:** the maximum number of mistakes (binary case) related to the margin or degree of separability

$$\text{mistakes} \ < \frac{k}{\delta^2}$$

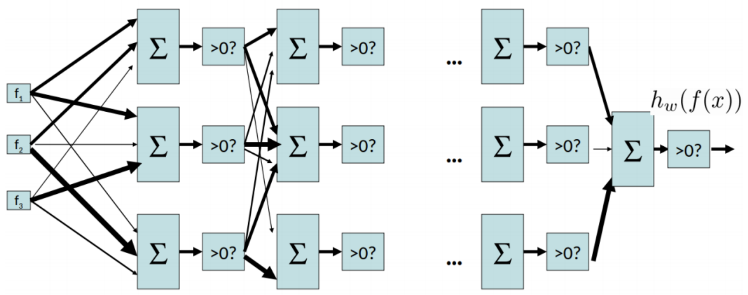
§9.4.4 Problems with Perceptrons

- **Noise:** if the data isn't separable, weights might thrash
  - Averaging weight vectors over time can help (averaged perceptron)
- **Mediocre generalization:** finds a **"barely"** separating solution
- **Overtraining:** test / held-out accuracy usually rises, then falls

§10 Neural Networks

§10.1 Motivation

§10.1.1 Multi-layer Perceptron



- Can express a much wider set of functions by increasing complexity
- Goal: select the best set of weights to parameterize the network

Theorem 10.1 (Universal Approximation Theorem)
A two-layer neural network with a sufficient number of neurons can approximate any continuous function to any desired accuracy.

§10.1.2 Accuracy

Binary Case:

- The accuracy of the binary perceptron after making m predictions can be expressed as:

$$l^{acc}(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m}\left(\operatorname{sgn}\left(\boldsymbol{w}\cdot\mathbf{f}\left(x^{(i)}\right)\right) == y^{(i)}\right)$$

  where sgn(x) represents the indicator function, which evaluates to 0 when x is negative, and 1 when x is positive

Multiclass Case:

- Want more expressive output than a binary label
- It becomes useful to produce a probability for each of the N classes
- To do so, transition from storing a weight vector to storing a weight vector for each class j
- Estimate probabilities with the **softmax function** $\sigma(x)$, which defines the probability of classifying $x^{(i)}$ to class j as:

Definition 10.2 (Softmax Function $\sigma(x)$)

$$\sigma\left(x^{(i)}\right)_j = \frac{e^{f\left(x^{(i)}\right)^T w_j}}{\sum_{k=1}^{N} e^{f\left(x^{(i)}\right)^T w_k}} = P\left(y^{(i)} = j \mid x^{(i)}\right)$$

- Likelihood for weights:

$$\ell(\boldsymbol{w}) = \prod_{i=1}^{m} P\left(y^{(i)} \mid x^{(i)}; \boldsymbol{w}\right)$$

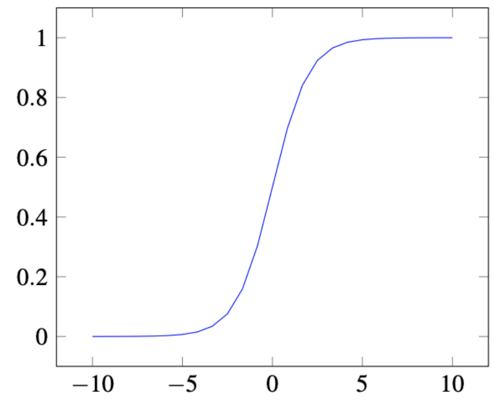
- Want to find the set of weights that maximizes this quantity
- Identical to finding the maximum of the log-likelihood expression (monotonic):

$$\ell\ell(\boldsymbol{w}) = \log \prod_{i=1}^{m} P\left(y^{(i)} \mid x^{(i)}; \boldsymbol{w}\right) = \sum_{i=1}^{m} \log P\left(y^{(i)} \mid x^{(i)}; \boldsymbol{w}\right)$$

§10.1.3 Multi-layer Feedforward Neural Network

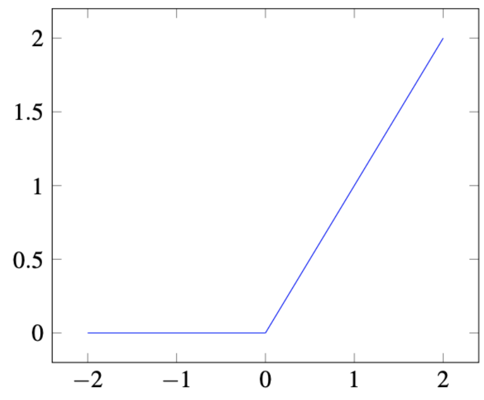
- Non-linearities make network more expressive
- Common activation functions:
  - **Sigmoid function**

Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$



  - **Rectified Linear Unit** (ReLU)

ReLU: $f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$



§10.1.4 Gradient Ascent/Descent

- Improve accuracy by optimizing weights via maximizing log-likelihood function

```
▪ init  $w$
▪ for iter = 1, 2, …
      $w \leftarrow w + \alpha * \nabla g(w)$
```

- To maximize log-likelihood function, the **gradient vector** gives the local direction of steepest ascent (or descent if reverse the vector)

$$\nabla_w \ell\ell(\mathbf{w}) = \left[\frac{\partial \ell\ell(\mathbf{w})}{\partial \mathbf{w}_1}, \dots, \frac{\partial \ell\ell(\mathbf{w})}{\partial \mathbf{w}_n}\right]$$

Definition 10.3 (Gradient Ascent)
Gradient ascent is a greedy algorithm that calculates this gradient for the current weight values, then updates the parameters along the direction of the gradient, scaled by a step size, α, the **learning rate**.

- Algorithm:
  Initialize weights $\boldsymbol{w}$ For $i = 0, 1, 2, \dots$

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \nabla_w \ell\ell(\boldsymbol{w})$$

- If instead minimizing a function f, the update should subtract the scaled gradient

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} f(\mathbf{w})$$

  which gives the **gradient descent** algorithm

- Crude rule of thumb: update changes $\mathbf{w}$ about $0.1 - 1\%$

§10.2 Optimization

- Goal: maximize

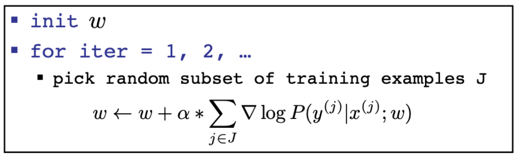
$$\ell\ell(\mathbf{w}) = \log \prod_{i=1}^{m} P\left(y^{(i)} \mid x^{(i)}; \mathbf{w}\right) = \sum_{i=1}^{m} \log P\left(y^{(i)} \mid x^{(i)}; \mathbf{w}\right)$$

- Alternative: **Batch Gradient Ascent**

```
▪ init $w$
▪ for iter = 1, 2, …
    $w \leftarrow w + \alpha * \sum_i \nabla \log P(y^{(i)}|x^{(i)}; w)$
```

- At each iteration, use all data points to compute gradients for the parameters $\mathbf{w}$, update the parameters, and repeat until convergence (local maximum)
- Slow due to large datasets, thus rarely used
- Instead, use **mini-batching**.

Definition 10.4 (Mini-batching)
Mini-batching rotates through randomly sampled batches of k data points at a time, compute gradients of the loss function using the selected batch (sum over the k datapoints in the batch, instead of all m datapoints in the training set)



```
▪ init $w$
▪ for iter = 1, 2, …
    ▪ pick random subset of training examples J
        $w \leftarrow w + \alpha * \sum_{j \in J} \nabla \log P(y^{(j)} | x^{(j)}; w)$
```

- Quicker computation of each gradient update
- The limit where batch size $k = 1$ is known as **stochastic gradient ascent(SGA)**

Remark 10.5. Neural networks are powerful (and universal!) **function approximators**, but can be difficult to design and train. The following are some examples of ongoing research in deep learning focusing on neural network design:

1. **Network Architectures:** designing a network (choosing activation functions, number of layers, etc.) that's a good fit for a particular problem
2. **Learning Algorithms:** how to find parameters that achieve a low value of the loss function, a difficult problem since gradient descent is a greedy algorithm and neural nets can have many local optima
3. **Generalization and Transfer Learning:** since neural nets have many parameters, it's often easy to overfit training data - how do you guarantee that they also have low loss on testing data you haven't seen before?

§10.3 Backpropagation

- Used to efficiently calculate the gradients for each parameter in a neural network

Theorem 10.6 (Chain Rule (Calculus))

$$\frac{\partial f}{\partial t_i} = \frac{\partial f}{x_1} \cdot \frac{\partial x_1}{\partial t_i} + \frac{\partial f}{x_2} \cdot \frac{\partial x_2}{\partial t_i} + \ldots + \frac{\partial f}{x_n} \cdot \frac{\partial x_n}{\partial t_i}$$

- To compute the gradient of a given node t_i with respect to the output z, take a sum of children (t_i) terms
- The goal during backpropagation is to determine the gradient of output with respect to each of the inputs