CS 188
Lecture Notes

Soklynin Nou
Fall 2025

Table of Contents

Introduction

Lecture 1

8/28/25

1. Background

What is **AI**?

There are many interpretations of **Artificial Intelligence**, which depends on the goal of the model. Are we try to make machines that:

- Behave like humans
- Act like humans
- Think rationally
- Act rationally

These are just some of the general interpretation of what an AI should be. The perspectives above highlights how some goals when making AI are results based or method based.

1.1. Rational Decisions

We use the term **Rational** in a very specific way. **Rationality** Maximally achieve a predetermined goal. We express this in terms of utility of the outcome, how much is the outcome worth. With added certainty, we are technically maximizing the **Expected Utility**.

1.2. Designing Ration Agents

An **Agent** is an entity that perceives and acts. A **Rational Agent** selects actions that maximize its (expected) utility based on its environment. An abstract of this is that the agent is in an environment, where it perceives the actions of its surroundings and takes actions accordingly.

- Skill-based Perspective
  - Can the model perform tasks
- Embodiment Perspective
  - Be able to take action in the world. Build the body and go from there
- Psychometrics Perspective
  - Instead of Measuring specific abilities, measure it on a broad range of tasks
- Human-Compatible Perspective
  - Maximize human utility. But since human utility is not well defined, it has inherent uncertainty

1.2.1. Reflex Agents

The simplest type of agent is the reflex agent, which chooses action based on current percept. It could have some memory of past events but doesn't take into account future details.

1.2.2. Planning Agents

These agents take into account the future consequences of their actions based on a formulated goal.

Uninformed Search

Lecture 2

9/02/25

2. Search Problems

A **Search Problem** consists of:

- **State space:** all possible states
- **Successor Function:** What is the next state after taking an action in a given state
- **Start State:**What is the initial state
- **Goal Test:** What is the machine trying to achieve

We would then need a solution to solve the search problem. A **Solution** is a sequence of actions starting from the Start State to the Goal State.
In Search Problems, we want to abstract away unnecessary details to capture just enough information.

2.1. State Space

A **State Space** consists of a **World State**, every single detail of the environment, and a **Search State**, only details needed for planning.

2.2. State Space Graphs

This is a mathematical way of representing search problems. The nodes are the possible states and the arcs/edges represent the successors.

A **Search Tree** is a tree where the root node is the start state, the edges are the actions, and the children are successors of the parent given the edge action.
Every search algorithm should be:

- **Complete:** Guaranteed to find a solution if one exists
- **Optimal:** Guaranteed to find the least solution

There is also the time and space complexity that comes with every algorithm.

2.2.1. Depth-First Search

This strategy is to explore the the deepest nodes first. It uses LIFO stack, which means we keep pushing all child of the node, pop the top node and iteratively run the algorithm. Terminates when the stack is empty.

```
def dfs:
  place root node on stack
  while stack is not empty
    pop the stack
    ignore if node is visited
    append node to result and set as visited
    for every children of node:
      if the node is not visited
        set node as visited
        place node in stack
  return result
```

2.2.2. Breadth-First Search

Expand across by breadth instead of depth. This is implemented using a FIFO queue. we start with the root node, place all children in a queue, dequeue the first children, repeat.

```
def bfs:
  place root node in queue
  while queue is not empty
    dequeue a node
    ignore if node is visited
    append node to result and set as visited
    for every children of node:
      if the node is not visited
        set node as visited
        place node in queue
  return result
```

2.3. Uniform Cost Search

Expands the cheapest node first using a priority queue. This is similar to Breadth-First Search where each depth is the cost of the node. This means we can represent the tree with cost contours, or layer with path of equal cost.

A* Search and Heuristics

Lecture 3

9/04/25

3. Informed Search

3.1. Search Heuristic

A **Search Heuristic** is a function that estimates how close a state is to the goal. We can use this to set our costs where a higher output of the heuristic function means the edge leads to a state further from the goal and a smaller output means the states is closer to the goal.

3.2. Greedy Search

This algorithm expands to the closest node first. This assumes that the solution is in the sub-tree of the immediate lowest cost edge

3.3. A* Search

In **A* Search**, we keep track of 3 values:

- **G value/Backwards Cost**: The cost to reach any given node.
- **H value/Forward Cost**: The distance from the goal
- **F value**: The sum of the backward and forward cost.

Terminates when we dequeue the goal

3.4. Admissible Heuristic

We want our heuristic to be **Optimistic** rather than pessimistic. An admissible/optimistic heuristic is:

$$0 \leq h(n) \leq h^*(n) \tag{1}$$

This means that our heuristic should underestimate the true cost of each node to the goal. With this, we can prove the optimality of the A^* algorithm.

Let A is the optimal solution and B is the suboptimal solution. Assume that A^* chooses B instead of A. An ancestor n of A must have been explored since we reached B. We know that:

$$\begin{aligned} &f(n) \leq g(A) && \text{Admissibility heuristic} \\ &f(A) = g(A) && \text{Since A is the goal, h(A)} = 0 \\ &f(A) < f(A) && \text{By definition} \\ &f(n) \leq f(A) \leq f(B) \end{aligned} \tag{2}$$

This means that node n should expand before node B, leading to a contradiction.

3.5. Creating Heuristics

Often, admissible heuristics are solutions to relaxed problems. **Relaxed Problems** are a version of the problem without the constraints, i.e. Pac-man without the walls.

In general, heuristics are defined in a semi lattice. This means that a heuristic dominants another if:

$$\forall n : h_{a(n)} \geq h_{c(n)} \tag{3}$$

This forms a semi-lattice where $h(n) = \max\Big(h_{a(n)}, h_{c(n)}\Big)$. The bottom of the semi-lattice is the zero heuristic, where $h(n) = 0$ and the top is the exact heuristic.

Additionally, our heuristic must have consistency. **Consistency** means the edge/arc cost must be less than or equal to the actual cost of the edge/arc:

$$h(A) - h(C) \leq \text{cost(A to C)} \tag{4}$$

A consequence of this is that the f value along the path never increases. Additionally, consistency is a sufficient condition for admissibility

Constraint Satisfaction Problems I

Lecture 4

9/09/25

4. Background

In search problems, we are making a number of assumptions about the world: a single agent, deterministic actions, fully observed states, discrete state spaces. etc. So we have to come up with the notion of **Identification Problems**, which is a problem of setting assignments to variables. This problem only focuses on the goal, not the path.

4.1. Constraint Satisfaction Problems

Constraint Satisfaction Problems are subsets of search problems where a states are variables $\mathbb{X}_i$ with values from a domain $\mathbb{D}$. Our goal is to set constraints specifying allowable combinations of values for subsets of variables. Additionally, our constraint can be **Implicit**, which relates multiple variables to each other, or **Explicit**, which defines a set of possible values for that variable. The solution are assignments that satisfies all the constraints.

4.1.1. Binary Constraint Satisfaction Problems

A **Binary Constraint Satisfaction Problem** are a subset of CSPs where each constraint relates at most two variables. This can be turned into a graph with edges representing constraints. If constraints relates to more than two variable, we can add a dummy node to represent the constraint with edges connecting the variable it relates to.

4.2. Solving Constraint Satisfaction Problems

A naive approach to solving CSPs is to formulate it as a search problem. But, the problem with this is that search problems cannot check constraints of the states nor is it efficient since we have to go down to the leaf. One fix for checking constraints is using backtracking search.

4.2.1. Backtracking Search

Backtracking Search is an extra computation on top of normal searches that checks constraints as you go. This requires extra computation on each node but removes the chance of going down an invalid path.

```
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
```

Backtracking requires us to consider parameters of assignments. Some parameters are:

- **Ordering:** what variable should be assigned next and in what order should the values be tried
- **Filtering:** how to detect if a solution is invalid early.
- **Structure:** can we take advantage of a problem's structure.

4.2.1.1. Filtering

The general idea of filtering is to keep track of domains for unassigned variables and cross off bad options. we do this by **Forward Checking**, crossing off values that violate a constraint when added to the existing assignment. Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures.

4.2.1.2. Arc Consistency

Arc Consistency handles the detection of invalid solutions. A arc $X \to Y$ is consistent iff:

$$\forall x \in X, \exists y \in Y : Y = y \wedge X = x \text{ is valid} \tag{5}$$

The algorithm of this is we select each possible value of X and see if there is an assignment to Y. If not, remove that value from X. This means we check the head and remove the tail. Even after enforcing arc consistency, there could be one solution, multiple solutions, or no solutions. **Important:** If X loses a value, neighbors of X need to be rechecked!

4.2.1.3. Ordering

Minimum remaining values: Choose the variable with the fewest legal left values in its domain.
Least Constraining Value: Choose the values that remove the least amount of other values

Constraint Satisfaction Problems II

Lecture 5

9/11/25

4.2.2. K-Consistency

We have seen before 1-consistency, where a constraint only applies to one variable, 2-consistency, where any pair of variables must be consistent. We can expand this notion to **K-Consistency**, which says that:

$$\forall x \in \{X_1, ..., X_{k-1}\}, \exists x' \in X_k : \text{all k variables are consistent} \tag{6}$$

4.2.3. Structure

We can exploit the structure of the problem to more efficiently solve the problem. A case is to use divide and conquer to tackle the subproblems, assuming they are independent.

Theorem: If the constraint graph has no loops, the CSP can be solved in $O(nd^2)$.

Another way of exploiting the structure is to turn the graph into a topological graph, where the directed edges are the arcs.

In general, if a CSP is tree-structured or close to tree-structured, we can run the tree-structured CSP algorithm on it to derive a solution in linear time. Similarly, if a CSP is close to tree-structured, we can

use cut-set conditioning to transform the CSP into one or more independent tree-structured CSPs and solve each of these separately.

4.3. Iterative Algorithms for CSPs

We start with a complete assignment of all the problems. Then, while it is not solved, we iteratively assign a different value to a randomly selected variable. We can use the minimum-conflict heuristic for this.

4.4. Local Search

Local Search is an algorithm where you start at an initial state and gradually improve by looking at the neighbors. We do this until we reach a local optimal solution. This is the same as **Gradient Decent**.

4.5. Genetic Algorithms

In **Genetic Algorithms**, we keep the best N solutions at each step based on a fitness function and perform pairwise crossover operators, with optional mutation to give variety in the hopes that the children is a better solution. This is analogous to natural selection.

Game Trees I

Lecture 6

9/16/25

5. Types of Games

5.1. Deterministic Games

A **Deterministic Game** is a game with no form of chance. It follows many possible structures, but it should generally have an initial state, actions from each states to another, a terminal test, and the terminal utility. The solution to these games is a policy of in each states.

5.2. Zero Sum Games

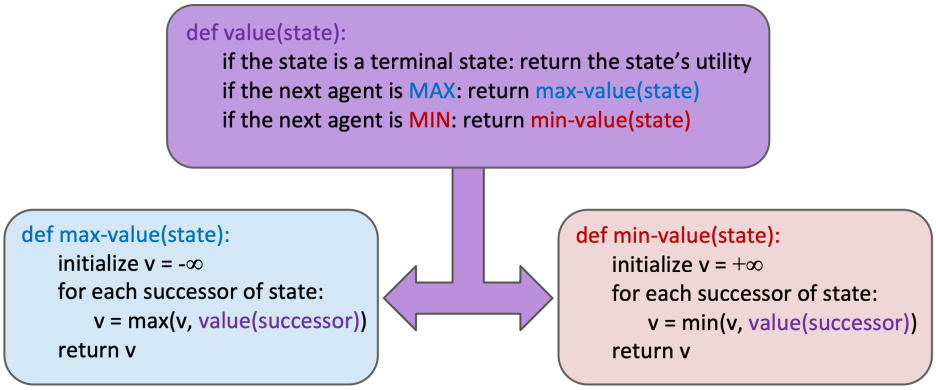
In a **Zero-Sum Game**, every Agents have opposite utilities, this lets us think of a single value that one maximizes and the other minimizes. This means that each agent is adversarial.

5.3. Adversarial Search

Because these types of games has two agents that are who are completely adversarial against each other, the agents must also consider the actions of its opponents, who considers the agents' actions, and so on. We assume the adversary plays optimally and is aware we are also playing optimally.

5.4. Minimax Values

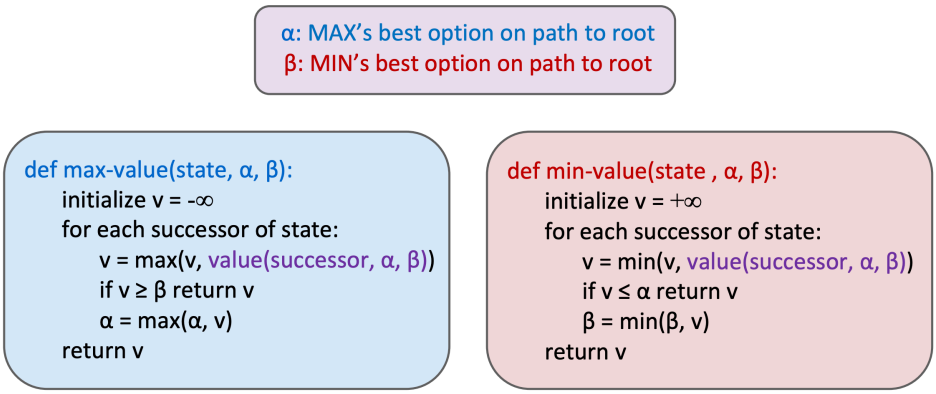
We can formulate actions of both agents, if they take turns, as nodes in a tree. We do this by alternating between layers of which one agent take action then the other does. Because each agent respectively minimize or maximize the utility, the value of the node as the min/max of the node's children.



5.5. Alpha-Beta Pruning

In a typical minimax tree, we are essentially running depth-first search. This is inefficients because it has to scan the entire tree, even if the sub-tree will be irrelevant. This is why we need **Alpha-Beta Pruning**.

Alpha-Beta Pruning is when we set a lower bound for each subsequent sibling sub-tree, letting us ignore the rest of the nodes if the upper-bound of that sub-tree is lower than the lower bound. In other words, if with finish a sub-tree and know that the adversary will a value x, and the first node of the next sub-tree produces a value $y < x$, we can ignore the rest of that sub-tree since we already have a value greater than its upper-bounded value. The upper-bound is the value which the adversary will pick in that sub-tree.



The pruning has no effect on the final value of the root, but the intermediate values of each of the node could be incorrect since we ignore a potentially lower value. This means that if we care about what action to take, rather than just the utility, alpha-beta pruning doesn't allow for action selection.

Game Trees II

Lecture 7

9/18/25

5.6. Resource Limits

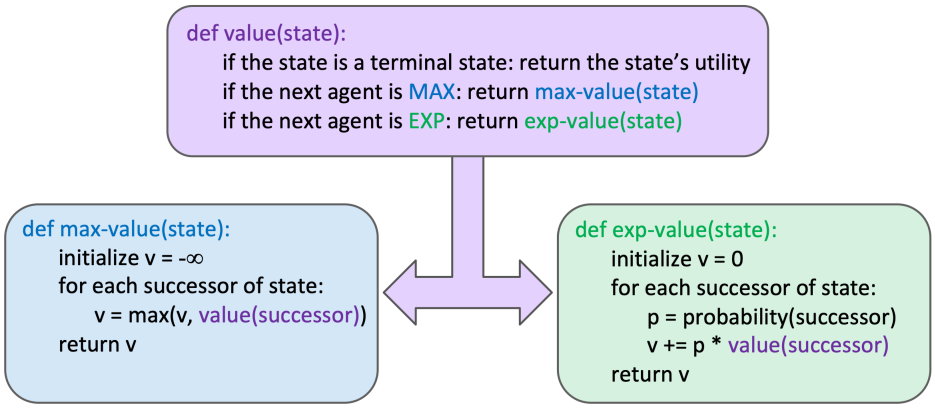
One problem with making a tree of possibility is the resource consumption and the limited space we have. To fix this, we implement **Depth-Limited Search**. In this search, we only search up to a certain depth of the tree. We then estimate the value of the nodes at that depth, since we are not at the terminal state. We would then propagate the values up to the root. This would remove the guarantee of optimal play.

One way to estimate the value of the node is to assign values on every action and state. We could then calculate the value function of the sequence of actions and states taken.

A potential problem is a replanning agent. This type of agent can loop at a tie-breaker. This will lead to the agent taking a series of looping actions and being stuck. Another reason for the being stuck is the evaluation function. A good evaluation function should minimize ties of sibling nodes.

5.7. Indeterministic Games (Expectimax)

So far, we assumed that the opponent is playing optimally. But, we need to handle the case in which they are not or when we are unsure of the opponent's strategy. This means that we have to introduce an idea of uncertainty to our node selection method. This is the idea of **Expectimax**. Expectimax is a search algorithm where the children node is selected based on specific probability. This allows us to represent the average value of the node.



In Expectimax, unlike Minimax, the magnitude of the values matter since we are taking the average of the nodes. This means we need to formulate a **Utility Function** which describes the preferences of the agent. The **Preferences** model how much a prize is valued over another.

5.8. Multi-Agent Utilities

In the case where there are more than two players, the game will have more complex behaviors, and no longer a zero-sum game. This is because we can model each player differently, whether they are indifferent of the other player's score, adversarial, or supporting.

Markov Decision Processes I

Lecture 8

9/23/25

6. Non-deterministic Search

Unlike deterministic search, **Non-deterministic Search** is use to model the real world since there are uncertainty in the world like failures or inherent chance like dice rolls in a game. We can use a **Markov Chain** to model this probabilistic model to better understand the framework.

6.1. Markov Decision Problem

A **Markov Decision Problem** is a search problem where the tree is modelled after a Markov Chain. MDPs are defined with:

- A set of states s ∈ S
- A set of actions a ∈ A
- A transition function $T(s, a, s')$
  - Probability that a from s leads to s', i.e., $P(s' | \ s, a)$, also called the model or the dynamics
- A reward function $R(s, a, s')$ or just $R(s)$ and $R(s')$
  - What reward you give for taking action a at state s and ending at state s'
- A start state and terminal state

Why do we use Markov? For Markov decision processes, "Markov" means action outcomes depend only on the current state. This simplifies the model since we don't have to consider the past of future.

In Deterministic Search, we have a plan that outlines the optimal path for the entire tree. This is in contrast to MDPs, which implements a policy π^*. A **Policy** outlines an action for every single state. Usually, policies cannot be implemented as a lookup table since they are often extremely large.

There are some implementations of rewards we can do:

- **Living Reward:** Usually negative, the lower the reward, the faster the agent will want to exit.

MDP Search Tree: Each MDP state projects an Expectimax-like search tree

6.2. Utilities of Sequences

There are many configurations of when a reward is received. Sometimes we want to make the agent prefer an immediate reward rather than a reward later. To do this, we make the reward decay exponentially by adding a **Discount Factor**. What this means is that the utility of a reward is calculated by:

$$R(s, t) = \lambda^t R(s) \tag{7}$$

This means that the reward R is worth $\lambda^T R$ at time step t, where $\lambda \leq 1$.

If we assume stationary preferences:

$$[a_1, a_2, ...] \succ [b_1, b_2, ...] \iff [r, a_1, a_2, ...] \succ [r, b_1, b_2, ...] \tag{8}$$

then we can define utility as:

- Additive: $U[r_0, r_1, ...] = r_0 + r_1 + r_2 + ...$
- Discounted: $U[r_0, r_1, ...] = r_0 + \lambda r_1 + \lambda^2 r_1 + ...$

What if the game lasts forever? Well, we can implement a finite horizon to terminate the game at a fixed time step. What we can also do is to use a discount factor less then one, which will be bounded due to the geometric series.

6.3. Solving MDPs

To calculate the optimal utility at a given state, we need to recurse over the possible successor states. We define the optimal value of a given state s as:

$$V^*(s) = \max_a Q^*(s, a) \tag{9}$$

where $Q^*(s, a)$ is the optimal value of taking an action a at state s. We compute $Q^*(s, a)$ by:

$$Q^*(s, a) = \sum_{s'} T(s, a, s')[R(s, a, s') + \lambda V^*(s')] \tag{10}$$

this equation says that the optimal value of taking an action a at state s is the sum of the expected values of state s' and its possible successors.

Substituting Q^* into the first equation, we get:

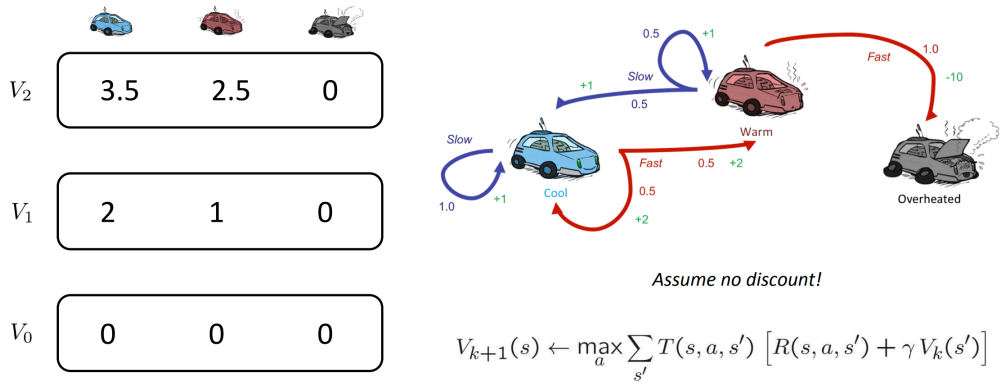
$$V^*(s) = \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \lambda V^*(s')] \tag{11}$$

We also need to implement **Time Limited Values**. This means we now have $V_k(s)$ to be the optimal value of s if the game ends in k more time steps. This allows us to bound our recursive tree to a depth of k. We now compute this by:

$$V_{k+1}(s) = \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \lambda V_k^*(s')] \tag{12}$$

where $V_0(s') = 0$ is our base case. The complexity of this algorithm is $O(S^2A)$.

Example:



Markov Decision Processes II

Lecture 9

9/25/25

6.4. Bellman Equations

Definition of "optimal utility" via Expectimax recurrence gives a simple one-step look ahead relationship amongst optimal utility values:

$$V^*(s) = \max_a \sum_{s'} \overbrace{T(s,a,s')}^{\text{Probabilty}} \underbrace{[R(s,a,s') + \gamma V^*(s')]}_{\text{Utility of taking action a in s to s'}} \tag{13}$$

6.5. Policy Method/Evaluation

One variance of policy evaluation is for a fixed policy:

$$V^\pi(s) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')] \tag{14}$$

notice how we are no longer maximizing over the possible actions since we have a fixed policy. This reduces the time complexity to: $O(S^2)$. This is also a linear system of equation.

6.6. Policy Extraction

What we can also do is to find what policy is optimal at each state:

$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')] \tag{15}$$

This is called **Policy Extraction**. Given the q-values, we can also compute it by:

$$\pi^*(s) = \arg\max_a Q^*(s,a) \tag{16}$$

this is more trivial since $Q^*(s,a)$ tells you the utility of each action.

6.7. Policy Iteration

Policy iteration is a method of finding the optimal policy. The problem with value iteration is that it is slow to converge values, bottle necking the policies. In Policy iteration, we alternate between two steps:

1. **Policy evaluation:** calculate utilities for some fixed policy (not optimal utilities!) until convergence

$$V_{k+1}^{\pi_i}(s) = \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma V_k^{\pi_i}(s')\right] \tag{17}$$

2. **Policy improvement:** update policy using one-step look-ahead with resulting converged (but not optimal!) utilities as future values

$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^{\pi_i}(s')] \tag{18}$$

This method converges much faster under some conditions.

In Policy Iteration, we run with a fixed policy and compute the utility of each state given that policy. We then find the best expected action for each room and change the policy accordingly. Repeat until the best expected action for each room is the policy we just ran.

Reinforcement Learning I

Lecture 10

9/30/25

7. Reinforcement Learning

In **Reinforcement Learning** we assume a MDP:

- A set of states $s \in S$
- A set of actions (per state) A
- A model $T(s, a, s')$
- A reward function $R(s, a, s')$

The difference is that we don't know the transition or reward function. Reinforcement learning takes into account the following concepts:

- **Exploration:** perform unknown actions to gather information
- **Exploitation:** Based on current information, perform the optimal action
- **Regret:** the loss between the best action and performed action
- **Sampling:** Perform action repeatedly to estimate it better
- **Difficulty:** learning can be much harder than solving a known MDP

7.1. Model-Based Learning

The idea behind **Model-based Learning** is that we make an approximate model based on what we know about the reward and transition functions. We then run the model as is.

1. **Step 1:** Take actions a_i at state s_i and estimate $T(s_i, a_i, s_i')$, get $R(s_i, a_i, s_i')$ when we reach s_i'
2. **Step 2:** Solve the MDP using the estimated reward and transition functions

7.2. Model Free Learning/Passive Reinforcement Learning

In **Passive Reinforcement Learning**, we are given a fixed policy to learn the reward and transition functions. We could do this by **Direct Evaluation**, where we average reward of running that policy at each initial state.

This is a simple algorithm that is easy to run but faces some problems. Some problems are that we might get unlucky when evaluating the value of a state and each state is learned separately.

7.2.1. Sample-Based Policy Evaluation

Since we don't know the transition function, we can take a sample of our past actions onto successor states to get an estimate of the function:

$$\begin{aligned} \text{sample}_i &= R(s, \pi(s), s_1') + \gamma V_k^\pi(s') \\ V_{k+1}^\pi(s) &= \frac{1}{n} \sum_i \text{sample}_i \end{aligned} \tag{19}$$

7.3. Temporal Difference Learning / Exponential Moving Average

The previous strategy we did was to run a certain amount of episodes, then extract information from the runs. **Temporal Difference Learning** updates $V(s)$ each time we experience transition (s, a, s', r):

$$V^\pi(s) = (1-\alpha)V^\pi(S) + \alpha \text{ sample} \tag{20}$$

What this means is that our estimate for state s is a weighted sum of our sample of s we just observed and the old estimate we had before, where the value of alpha is importance of the new value.

In **Exponential Moving Average**, we can compute the average of the states we see by running interpolation update:

$$\begin{aligned} \bar{x}_n &= (1-\alpha)\bar{x}_{n-1} + \alpha \bar{x}_n \\ &= \frac{x_n + (1-\alpha)x_{n-1} + (1-\alpha)^2 x_{n-2} + ...}{1 + (1-\alpha) + (1-\alpha)^2 + ...} \end{aligned} \tag{21}$$

This puts more emphasis on the most recent sample, with more distant sample decaying exponentially. α tends to be a small number and gets smaller over time so that the averages converge.

Temporal Difference Learning gives us a good way of estimating values of individual states, but it doesn't allow for policy extraction like other methods. To fix this, we should estimate the **Q-value**, which describes the values of an action and state pair, instead of just the value.

7.4. Active Reinforcement Learning

Unlike passive reinforcement learning, where a fixed policy is given to us, **Active Reinforcement Learning** makes us decide the next action by ourselves. Just like passive reinforcement learning. We are not given the reward nor the transition function

7.4.1. Q-Value Iteration

In **Q-Value Iteration**, we start we $Q_0(s, a) = 0$ as the base case and calculate:

$$Q_{k+1}(s, a) = \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma \max_{a'} \underbrace{Q_k(s', a')}_{V_k(s')} \right] \tag{22}$$

We learn $Q(s, a)$ values as we go:

- receive a sample (s, a, s', r)
- Consider your old estimate: Q(s, a)
- Consider your new sample estimate: sample $= R(s, a, s') + \gamma \max_a Q(s', a')$

Q-learning converges to optimal policy after enough iterations. This is called **Off-Policy learning**.

Reinforcement Learning II

Lecture 11

10/02/25

7.5. Exploration Function

The method of Exploration is to learn more about the environment around us. When doing this, we need to balance the over exploring, which may lead to repeated bad paths, and under exploring, which leads to not finding the optimal path.

The simplest way to do this is through random randoms, or **ε-greedy**. This method says that the agent acts randomly with probability ε, which is very small, and follows the current policy otherwise.

One key idea is to lower epsilon overtime since we would have explored most of the environment already.

The **Exploration Function** takes in a value estimate u and a visit count n, and returns an optimistic utility: $f(u, n) = u + \frac{k}{n}$. Intuitively, this is an upper bound on how go the path can be. This will turn the estimated value to:

$$Q(s, a) = \alpha R(s, a, s') + \gamma \max_{a'} f(Q(s', a), N(s', a)) \tag{23}$$

This will make the agent tend towards states that haven't been explored much.

7.6. Regret

Intuitively, **Regret** is the measure of the total mistake cost. This is the different between your expect reward and the optimal expected reward.

7.7. Approximate Q-Learning

In basic Q-learn, we have a table of all q-values. But, this is impractical in the real world since states are usually extremely large, which also means it will take a long time to visit enough states.

What we do instead is to generalize the Q-values of the environment. Rather than maintaining a list of value, we describe a state using a vector of features.

To compute this, we can do a weighted sum of the features:

$$\begin{aligned} V(s) &= \omega_1 f_1(s) + ... + \omega_n f_n(s) \\ Q(s, a) &= \omega_1 f_1(s, a) + ... + \omega_n f_n(s, a) \end{aligned} \tag{24}$$

Intuitively, we adjust the weights of each feature based on our prior knowledge and the current sample:

$$\omega_i = \omega_i + \alpha[\text{difference}] f_i(s, a) \tag{25}$$

7.8. Policy Search

In **Policy Search**, we start with an initial value function or Q-function and we adjust each function weight to see if the policy improves.

One problem with this is to know if the policy improve because we will have to run many samples.

Probability

Lecture 12

10/07/25

8. Uncertainty

Expanding on the concept of learning without given information, we use the concept of **Uncertainty** to add to the model of the world. In a general situation, we have:

- **Observed Variables:** Data the agent knows
- **Unobserved Variables:** Data the agent doesn't knows
- **Model:** Relating the observed to the unobserved variables

To do this, we use **Random Variables**, which is an aspect of the world that has uncertainty.

8.1. Probabilistic Model

In **Probability Distribution**, we associate each attribute of the environment with a non-zero probability. The reason why a probability of zero cannot be used is for computational reasons.

But, explicitly assignment probability to each outcome is impractical since if we have n variables, each with a sample space of d, the total size of the probability table is n^d.

A **Probabilistic Model** is a joint distribution over a set of random variables. This is similar to Constraint Satisfaction Problems, where the constraints are the probability and the domain is the sample space. To expand on this, we need to implement new concepts.

An **Event E** is a set of outcomes:

$$P(E) = \sum_{x_1,...,x_n \in E} P(x_1, ..., x_n) \tag{26}$$

Marginal Distribution is a sub-table that eliminate variables. For example, given the table for $P(T, W)$:

$$P(T) = \sum_s P(T = t, S = s) \tag{27}$$

Intuitively, we group variables and aggregate the variable we want eliminated.

A **Conditional Probability** is the probability of an outcome given what we know: We could then use this for a **Conditional Distribution**, the probabilities over known variable.

$$P(W \mid T = t) = \frac{P(W = w, T = t)}{\sum_W P(W = w_i, T = t)} \tag{28}$$

Each outcome is the joint probability of a given outcome over the sum of all probability over conditional. We then should **Normalize** it, which is the process of making the probability of all entries add up to 1.

8.2. Probabilistic Inference

Probabilistic Inference compute a desired probability from other known probabilities. To do this, we compute conditional probabilities since it can relate variables to each other.

8.2.1. Inference by Enumeration

Inference by Enumeration is a general framework of implementing inference where we have:

- **Evidence variable:** Known information
- **Query variable:** What we are trying to model
- **Hidden Variable:** unknown information

We then compute $P(Q \mid e_1, ..., e_k)$. We can do this use the following procedure:

1. Select the entries containing the evidence
2. Sum out the hidden using conditional distribution
3. Normalize the distribution

To run this procedure we need to apply concepts of probability:

- **Product Rule:** $P(x, y) = P(x|y)p(y)$
- **Chain Rule:** $P(x_1, ..., x_n) = \prod_i P(x_i|x_1, ..., x_{i-1})$
- **Bayes Rule:** $P(x|y) = \frac{P(y|x)P(x)}{P(y)}$

Bayes Nets: Representation

Lecture 13

10/09/25

8.2.2. Independence

We say two variables $X \perp\!\!\!\perp Y$ are independent if the outcome of one doesn't affect the other:

$$\begin{aligned} P(x, y) &= P(x)P(y) \\ P(x|y) &= P(x) \end{aligned} \tag{29}$$

We rarely have absolute independence, instead we might have **Conditional Independence**. In this instance, some variables might be independent if another variable is present. If we say that:

$$P(A^+|B^+, C^+) = P(A^+|C^+)P(A^-|B^+, C^-) = P(A^-|C^-) \tag{30}$$

then we can say that A is *conditionally independent* of B given C. Biconditional implication:

$$\begin{aligned} P(B|A, C) &= P(B|C) \\ P(B, A|C) &= P(B|C)P(A|C) \end{aligned} \tag{31}$$

This can be applicable when simplifying the chain rule since we can remove any variable that is conditionally independent with the random variable.

9. Bayes' Net

Bayes' Nets is a technique to compactly describe complex joint distributions using simple, local distributions. This is a graph where the nodes are the variables and the edges indicates conditional independence. Each node should have a probability of the node given its connect nodes.

This means that a Bayes' Net implicitly encodes a joint distribution:

$$P(x_1, x_2, ..., x_3) = \prod_{i=1}^{n} P(x_i|\text{parents}(X_i)) \tag{32}$$

This formula shows that we can simplify the chain rule of probability to only include the node's parents. This then implies that anything that is not a direct parent of the node is conditionally independent from the node.

Bayes Nets: Independence

Lecture 14

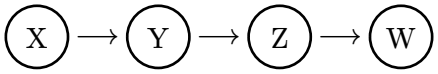
10/14/25

9.1. Size of Bayes' Net

Because each node much encode the probability given its parents, the size of the Bayes' Net scales exponentially with each new variable, specifically $N * D^K$ where K is the number of direct parent, N is the number of nodes, and D is the sample space.

9.2. Independence of Bayes' Net

Encoded in a Bayes' Net is the implication that any nodes not connected are conditionally independent of each other:



$$P(W|X, Y, Z) = P(W|Z)$$
$$P(Z|X, Y) = P(Z|Y) \tag{33}$$

We can then say that X and Z are conditionally independent given Y, as well as:

- $Z \perp\!\!\!\perp X|Y$
- $W \perp\!\!\!\perp Y|Z$
- $W \perp\!\!\!\perp X|Z$

9.3. D-Separation

In general, proving independence is very tedious since it requires analyzing all N nodes. We should then use the **D-Separation** algorithm to simplify the process.

To prove that $Z \perp\!\!\!\perp X|Y$, we ask ourselves if we know Y, does knowing X or Z affect the outcome:

$$P(z|x,y) = \frac{P(x,y,z)}{P(x,y)} = \frac{\overbrace{\cancel{P(x)P(y|x)}P(z|y)}^{\text{Bayes' Net Reconstitution Recipe}}}{\underbrace{\cancel{P(x)P(y|x)}}_{\text{Product Rule}}} = P(z|y) \tag{34}$$

We can intuitively think of it as if removing Y will remove all paths going from X to Z.
Example 1:

(35)

In the above example, X and Z are independent from each other but are not conditionally independent given Y.
Example 2:

(36)

In the above example, X and Z are not independent from each other but are conditionally independent given Y.

These two examples highlight how the direction of the arc is important to determine conditionally and unconditionally independence.

9.4. Active/Inactive Paths

The quickest way of checking for conditional and unconditional independence is to check for **Active/ Inactive Paths**. The common pattern for **Active Triples** are:

- Casual Chain: $A \to B \to C$, where B is unobserved
- Common Cause: $A \leftarrow B \to C$, where B is unobserved
- Common Effect: $A \to B \leftarrow C$, where B or one of its descendents are observed

If an inactive segment is on the path between two nodes, then the path is blocked. If there is no path form X to Z, then they are independent.

Bayes Nets: Inference

Lecture 15

10/16/25

9.5. Inference

Inference: Calculating some useful information from the joint probability distribution. This allows use to extract the information we want by taking the joint distribution of the probability distribution.

9.5.1. Inference by Enumeration

Inference by Enumeration is the algorithm that helps us find inferences. The steps are:

1. The entries consistent with the evidence

2. Sum out H to get joint of Query and evidence
3. Normalize

This is straight forward but slow because we join up the whole distribution, which is exponentially large, before summing up the hidden variable.

9.5.2. Inference by Variable Elimination

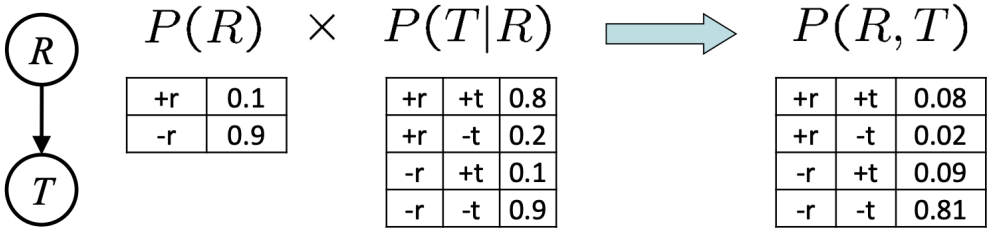
A better option is **Variable Elimination**, which is exponential in the worst case but is faster in general. This is the process of take a subset of pieces, multiplying them. This means that there will be intermediate distributions that could or could not be useful.

These sub-distributions are called **Factors**, a multi-dimensional array. These array are usually not displayed in the final result.

To perform Variable Elimination, we:

Join Factors:
First basic operation: joining factors

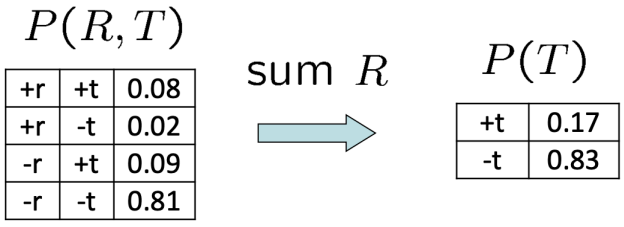
- Combining factors:
- Just like a database join
- Get all factors over the joining variable
- Build a new factor over the union of the variables involved



We apply the transformation of each condition from $P(R)$ to the corresponding value in $P(T \mid R)$

Eliminate:
Take a factor and sum out a variable

- Shrinks a factor to a smaller one
- A projection operation



This is equivalent to `df.groupby([all_col - r]).sum()`

9.5.2.1. Variable Elimination Ordering

Based on the implementation of Variable Elimination, the order in of the variables we eliminate through each iteration can effect the runtime of the algorithm. We want to chose factors that doesn't grow the result factor too much.

Bayes Nets: Sampling

Lecture 16

10/21/25

10. Bayes Net Sampling

An alternate approach for probabilistic reasoning is to implicitly calculate the probabilities for our query by simply counting samples. We can generate samples from the Bayes net and use those samples to estimate probabilities This will give us a good estimate of the conditional probability.

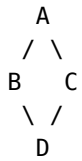
10.1. Prior Sampling

Generate samples from the full joint distribution defined by the Bayes net. In short, we walk down the Bayes Net and sample each node given fixed parents. It is good when we don't have evidence.

Algorithm:

1. Order variables in topological order.
2. For each variable, sample its value based on the values of its parents.
3. Repeat to generate many samples.

Example: Consider the following Bayes net:



To generate a sample:

1. Pick an outcome A from P(A).
2. Pick an outcome B from P(B | A).
3. Pick an outcome C from P(C | A).
4. Pick an outcome D from P(D | B, C).

10.2. Rejection Sampling

In rejection sampling, we generate samples from the prior distribution but only keep those that are consistent with the evidence.

Algorithm:

1. Generate a sample from the prior distribution.
2. If the sample is consistent with the evidence, keep it; otherwise, discard it.
3. Repeat to generate many samples.
4. Estimate probabilities based on the kept samples.

Example:
To estimate P(Burglary | JohnCalls = true):

1. Generate a sample from the prior distribution.
2. If JohnCalls = true in the sample, keep it; otherwise, discard it.
3. Count how many kept samples have Burglary = true and divide by the total number of kept samples to estimate the probability.

The main drawback of rejection sampling is that if the evidence is rare, many samples will be discarded, leading to inefficiency.

10.3. Likelihood Weighting

Likelihood weighting is a more efficient sampling method that incorporates evidence directly into the sampling process. Instead of discarding samples that don't match the evidence, we weight them based on how likely they are given the evidence.
Algorithm:

1. For each variable, if it is an evidence variable, set it to the evidence value and assign a weight based on its probability given its parents.
2. For non-evidence variables, sample their values based on their parents (Honest sampling).
3. Repeat to generate many weighted samples.
4. Estimate probabilities using the weights of the samples.

Example:
To estimate P(Burglary | JohnCalls = true):

1. For each sample, set JohnCalls = true and assign a weight based on P(JohnCalls = true | Parents).
2. Sample the other variables (Burglary, Earthquake, MaryCalls) based on their parents.
3. Use the weights of the samples to estimate the probability of Burglary given JohnCalls = true.

The disadvantage of this is that it works better if our query evidence is at the top.

10.4. Gibbs Sampling

Gibbs sampling is a Markov Chain Monte Carlo (MCMC) method that generates samples by iteratively updating the value of one variable at a time, conditioned on the current values of all other variables.
Algorithm:

1. Initialize all variables to some value.
2. For each variable, sample its value based on the current values of all other variables.
3. Repeat for many iterations to generate samples.
4. Estimate probabilities based on the samples.

Example:
To estimate P(Burglary | JohnCalls = true):

1. Initialize all variables (Burglary, Earthquake, MaryCalls) to some values.
2. For each variable, sample its value based on the current values of the other variables and the evidence (JohnCalls = true).
3. After many iterations, use the samples to estimate the probability of Burglary given JohnCalls = true.

Gibbs sampling is particularly useful when dealing with complex networks where direct sampling is difficult.

Decision Networks and VPI

Lecture 17

10/23/25

11. Decision Networks

A decision network (or influence diagram) is an extension of a Bayesian network that includes decision nodes and utility nodes to model decision-making under uncertainty.

- **Chance Nodes:** Represent random variables (like in Bayes nets).
- **Decision Nodes:** Represent choices available to the decision-maker.
- **Utility Nodes:** Represent the utility (or value) associated with outcomes.

Example:
Consider a decision network for a medical diagnosis scenario:

```
[Test Result]
     |
  [Disease] ----> [Treatment Decision] ----> [Utility]
```

- The "Test Result" node is a chance node representing the outcome of a medical test.
- The "Disease" node is a chance node representing whether the patient has a disease.
- The "Treatment Decision" node is a decision node representing the choice of treatment.
- The "Utility" node represents the utility associated with the treatment outcome (e.g., health improvement, side effects).

Inference in Decision Networks:
To make decisions using a decision network, we need to compute the expected utility of each decision option and choose the one with the highest expected utility.
Steps to Compute Expected Utility:

1. Compute expected utility by summing over all possible outcomes, weighted by their probabilities.
2. Choose the decision option with the highest expected utility.

Example Calculation:
To decide whether to treat a patient based on the test result:

1. Calculate the expected utility of treating the patient given positive and negative test results.
2. Calculate the expected utility of not treating the patient given positive and negative test results.
3. Compare the two expected utilities and choose the option with the higher value.

Advantages of Decision Networks:

- Provide a structured way to model complex decision-making scenarios.
- Allow for the incorporation of uncertainty and preferences in decision-making.
- Facilitate the analysis of trade-offs between different decision options.

Challenges:

- Computational complexity in large networks.
- Difficulty in accurately specifying probabilities and utilities.
- Sensitivity to changes in model parameters.

11.1. Value of Information

The value of information (VOI) is a concept used to quantify the benefit of obtaining additional information before making a decision. It helps decision-makers determine whether acquiring new information is worth the cost associated with obtaining it.

Definition:

The value of information is defined as the increase in expected utility that results from having access to additional information before making a decision.

Calculating VOI:

1. Compute the expected utility of the decision without the additional information.
2. Compute the expected utility of the decision with the additional information.
3. The value of information is the difference between the expected utility with the information and the expected utility without it.

Example:

Consider a scenario where a doctor must decide whether to treat a patient for a disease based on symptoms. The doctor can choose to order a diagnostic test that provides additional information about the patient's condition.

1. Without the test, the doctor estimates the expected utility of treating or not treating based on prior probabilities of the disease.
2. With the test, the doctor can update the probabilities based on the test results and compute the expected utility of each decision option.
3. The VOI is the difference in expected utility between the two scenarios.

Properties:

- VOI is always non-negative; having additional information cannot decrease expected utility.
- VOI is non-additive; the value of multiple pieces of information is not the sum of their values.
- VOI is order-independent; the order in which information is obtained does not affect its value.

Value of Imperfect Information:

In many cases, the information obtained may be imperfect or noisy. All we have to do is add another node to encode the noise.

12. Partially Observable Markov Decision Processes (POMDPs)

A Partially Observable Markov Decision Process (POMDP) is an extension of a Markov Decision Process (MDP) that accounts for situations where the agent does not have full observability of the environment's state. In a POMDP, the agent must make decisions based on incomplete and uncertain information about the current state.

Components of a POMDP:

- **States (S):** A finite set of states representing the possible configurations of the environment.
- **Actions (A):** A finite set of actions that the agent can take.
- **Observations (O):** A finite set of observations that the agent can receive about the environment.
- **Transition Model (T):** The probability of going from one state to another given an action: T(s, a, s').
- **Observation Model (O):** The probability of receiving an observation given the current state and action: O(s', a, o) = P(o | s', a).
- **Reward Function (R):** A function that defines the immediate reward received after taking an action in a state: R(s, a).

Belief State:
In a POMDP, the agent maintains a belief state, which is a probability distribution over all possible states. The belief state is updated based on the agent's actions and observations using Bayes' theorem. Put simply, the belief state represents the agent's knowledge about the environment at any given time.

Hidden Markov Models

Lecture 18

10/28/25

13. Markov Chains

A Markov chain is a mathematical system that undergoes transitions from one state to another on a state space. It is a random process that satisfies the Markov property, which states that the future state depends only on the current state and not on the sequence of events that preceded it.

Components of a Markov Chain:

- **States (S):** A finite set of states representing the possible configurations of the system.
- **Transition Probabilities (P):** A matrix that defines the probabilities of transitioning from one state to another.
- **Initial State Distribution (π):** A probability distribution over the states that defines the starting state of the system.

13.1. Mini-Forward Algorithm

The mini-forward algorithm is a simplified version of the forward algorithm used in Hidden Markov Models (HMMs) to compute the probability of a sequence of observations. It operates by iteratively updating the probabilities of being in each state at each time step based on the transition probabilities and observation likelihoods.

Algorithm:

1. Initialize the probabilities for the initial state distribution.
2. For each time step, update the probabilities for each state based on the previous time step's probabilities, transition probabilities, and observation likelihoods.
3. Normalize the probabilities at each time step to ensure they sum to 1.

Example:
Consider a simple HMM with two states (Rainy and Sunny) and two observations (Walk and Shop). The transition probabilities and observation likelihoods are defined as follows:

- Transition Probabilities:
  - P(Rainy | Rainy) = 0.7
  - P(Sunny | Rainy) = 0.3
  - P(Rainy | Sunny) = 0.4
  - P(Sunny | Sunny) = 0.6

- Observation Likelihoods:
  - P(Walk | Rainy) = 0.1
  - P(Shop | Rainy) = 0.9
  - P(Walk | Sunny) = 0.6
  - P(Shop | Sunny) = 0.4

To compute the probability of the observation sequence [Walk, Shop] using the mini-forward algorithm:

1. Initialize the probabilities for the initial state distribution (e.g., P(Rainy) = 0.5, P(Sunny) = 0.5).
2. For the first observation (Walk), update the probabilities for each state:
   - $P(\text{Rainy} \mid \text{Walk}) = P(\text{Walk} \mid \text{Rainy})(P(\text{Rainy} \mid \text{Rainy})P(\text{Rainy}) + P(\text{Rainy} \mid \text{Sunny})P(\text{Sunny}))$
   - $P(\text{Sunny} \mid \text{Walk}) = P(\text{Walk} \mid \text{Sunny})(P(\text{Sunny} \mid \text{Rainy})P(\text{Rainy}) + P(\text{Sunny} \mid \text{Sunny})P(\text{Sunny}))$
3. Normalize the probabilities.
4. For the second observation, repeat the update process using the first observation probabilities.
5. Normalize the probabilities again.

The final probabilities is the likelihood of being in each state after observing the sequence [Walk, Shop].

13.2. Stationary Distributions

A stationary distribution is a probability distribution over the states of a Markov chain that remains unchanged as the system evolves over time. In other words, if the Markov chain starts in the stationary distribution, it will remain in that distribution at all future time steps.

Finding Stationary Distributions:
To find the stationary distribution of a Markov chain, we need to solve the following equation:

$$\pi P = \pi \tag{37}$$

where π is the stationary distribution vector and P is the transition probability matrix. This equation states that the stationary distribution is an eigenvector of the transition matrix corresponding to the eigenvalue 1.

Example:
Consider a Markov chain with the following transition matrix:

	A	B
A	0.8	0.2
B	0.4	0.6

To find the stationary distribution π = [π(A), π(B)], we need to solve the equations:

$$\begin{aligned} \pi(A) &= 0.8\pi(A) + 0.4\pi(B) \\ \pi(B) &= 0.2\pi(A) + 0.6\pi(B) \\ \pi(A) + \pi(B) &= 1 \end{aligned} \tag{38}$$

Properties of Stationary Distributions:

- A Markov chain can have multiple stationary distributions if it is not irreducible or aperiodic.
- If a Markov chain is irreducible and aperiodic, it has a unique stationary distribution.
- The stationary distribution can be interpreted as the long-term behavior of the Markov chain.

13.3. Hidden Markov Models (HMMs)

A Hidden Markov Model (HMM) is a statistical model that represents a system with hidden states that generate observable outputs. HMMs are widely used in various applications, including speech recognition, natural language processing, and bioinformatics.

13.4. Filtering / Monitoring

Filtering, also known as monitoring, is the process of estimating the current hidden state of a system based on a sequence of observations. In an HMM, filtering involves computing the posterior distribution of the hidden state given the observed evidence up to the current time step.

13.4.1. Passage of Time Update

When time passes without any new observations, we need to update our beliefs about the hidden state based on the transition probabilities of the HMM. This is done using the following equation:

$$B(s') = \sum_s P(s' \mid s)B(s) \tag{39}$$

This says that the belief in state s' is the sum over all the probability of transitioning from s to s', for all s, times the belief in s.

Algorithm:

1. Initialize the belief state based on the initial state distribution.
2. For each time step, update the belief state using the transition probabilities and observation likelihoods.
3. Normalize the belief state to ensure it sums to 1.

Example:

Consider an HMM with two hidden states (Rainy and Sunny) and two observations (Walk and Shop). Given a sequence of observations [Walk, Shop], we want to estimate the current hidden state using filtering.

1. Initialize the belief state (e.g., P(Rainy) = 0.5, P(Sunny) = 0.5).
2. For the first observation (Walk), update the belief state:
   - $P(\text{Rainy} \mid \text{Walk}) = P(\text{Walk} \mid \text{Rainy})(P(\text{Rainy} \mid \text{Rainy})P(\text{Rainy}) + P(\text{Rainy} \mid \text{Sunny})P(\text{Sunny}))$
   - $P(\text{Sunny} \mid \text{Walk}) = P(\text{Walk} \mid \text{Sunny})(P(\text{Sunny} \mid \text{Rainy})P(\text{Rainy}) + P(\text{Sunny} \mid \text{Sunny})P(\text{Sunny}))$
3. Normalize the belief state.
4. For the second observation (Shop), repeat the update process using the first observation belief state.
5. Normalize the belief state again.

The final belief state represents the estimated probabilities of being in each hidden state after observing the sequence [Walk, Shop].

13.4.2. Observation Update

After we receive an observation, we need to update our beliefs based on how likely that observation is in each state. This is done using Bayes' theorem:

$$B(s) \propto P(e \mid s)B'(s) \tag{40}$$

where alpha is a normalization constant to ensure the beliefs sum to 1. This update adjusts our belief in each state based on the likelihood of the observed evidence given that state.

13.5. Forward Algorithm

The forward algorithm is a dynamic programming algorithm used in Hidden Markov Models (HMMs) to compute the probability of a sequence of observations. It operates by iteratively updating the probabilities of being in each hidden state at each time step based on the transition probabilities and observation likelihoods.

Algorithm:

1. Passage of Time: $p(x_2) = \sum_{x_1} P(x_1, x_2) = \sum_{x_1} p(x_1)P(x_2 \mid x_1)$
   - This says that the probability of getting to state x_2 is the sum over all the ways of getting to x_2 from any previous state x_1, times the probability of being in that previous state x_1.
2. Observe evidence: $p(x_2 \mid e_2) \propto P(e_2 \mid x_2)p(x_2)$
   - This says that the probability of being in state x_2 given evidence e_2 is proportional to the likelihood of observing e_2 given state x_2, times the prior probability of being in state x_2.

13.6. Online Belief Update

The online belief update is a method for updating the belief state of a Hidden Markov Model (HMM) as new observations are received. It combines the passage of time update and the observation update to maintain an accurate estimate of the hidden state over time.

Particle Filtering

Lecture 19

10/30/25

14. Filtering

Filtering, also known as monitoring, is the process of estimating the current state of a system based on a sequence of observations. In the context of Hidden Markov Models (HMMs), filtering involves computing the belief state, which is the probability distribution over the hidden states given all the observations up to the current time step.

1. **Elapse Time:** $P(x_t \mid \mid e_1 : t-1) = \sum_{x_{t-1}} P(x_{t-1} \mid e_{1:t-1})P(x_t \mid x_{t-1})$
2. **Observation:** $P(x_t \mid e_1 : t) \propto P(x_t \mid e_1 : t-1)P(e_t \mid x_t)$

14.1. Particle Filtering

Normal filtering can be computationally expensive, especially when the state space is large. Particle filtering is an approximate inference algorithm that uses a set of samples (particles) to represent the belief state. Each particle represents a possible state of the system, and the collection of particles approximates the probability distribution over the states.

Algorithm:

1. Initialize a set of N particles by sampling from the initial state distribution.
2. For each time step:
    - Elapse Time: For each particle, move it to a new state by sampling from the transition model.
    - Observation: Weight each particle based on the likelihood of the observed evidence given the particle's state. Down weight particles that are inconsistent with the evidence and up weight those that are consistent.
    - Resample: Draw N particles from the weighted set of particles, with replacement, to form the new set of particles.

14.1.1. Robot Localization Example

Consider a robot moving in a 1D world with three positions: Left, Center, and Right. The robot can move left or right with some uncertainty, and it can sense its position with some noise. We want to use particle filtering to estimate the robot's position based on its movements and sensor readings.

1. Initialize a set of particles representing the robot's possible positions.
2. For each time step:
    - Elapse Time: Move each particle based on the robot's movement model (e.g., if the robot moves right, shift particles to the right with some probability).
    - Observation: Weight each particle based on the sensor reading (e.g., if the sensor indicates the robot is at Center, give higher weights to particles at Center).
    - Resample: Draw a new set of particles based on the weights to form the updated belief state.

The final set of particles represents the estimated distribution of the robot's position after considering its movements and sensor readings.

ML I: Naive Bayes

Lecture 20

11/04/25

15. Classification

In machine learning, classification is a supervised learning task where the goal is to predict the categorical label of an input based on its features.

15.1. Naive Bayes Classifier

The Naive Bayes classifier is a simple yet effective probabilistic classifier based on Bayes' theorem with the "naive" assumption of feature independence.

The model computes the posterior probability of each class given the input features and assigns the class with the highest probability. The formula for the Naive Bayes classifier is given by:

$$P(Y, F_1, ..., F_n) = P(Y) \prod_{i=1}^{n} P(F_i \mid Y) \tag{41}$$

Where:

- Y is the class label.
- $F_1, ..., F_n$ are the features.
- $P(Y)$ is the prior probability of class Y.

15.2. Inference for Naive Bayes

To classify a new instance with features $f_1, ..., f_n$, we compute the posterior probability for each class y:

$$P(Y = y \mid F_1 = f_1, ..., F_n = f_n) \propto P(Y = y) \prod_{i=1}^{n} P(F_i = f_i \mid Y = y) \tag{42}$$

15.3. General Naive Bayes

What we neeed to use Naive Bayes is:

1. Inference method:
    - Compute $P(Y = y)$ for each class y.
    - Compute $P(F_i = f_i \mid Y = y)$ for each feature F_i and class y.
2. Estimates of local conditional probabilisty tables:
    - $P(Y)$: Estimated from the training data as the frequency of each class.
    - $P(F_i \mid Y)$: Estimated from the training data as the frequency of each feature value given each class.

Naive Bayes assumes that all features are conditionally independent given the class label.
One detail is that we take the logarithm of the probabilities to avoid numerical underflow and convert products into sums:

$$\log P(Y = y \mid F_1 = f_1, ..., F_n = f_n) = \log P(Y = y) + \sum_{i=1}^{n} \log P(F_i = f_i \mid Y = y) + C \tag{43}$$

Where C is a constant that does not depend on y.

15.4. Traing and Testing

Empirical Risk Minimization (ERM) is a principle in statistical learning theory that aims to find a hypothesis that minimizes the empirical risk, which is the average loss over the training data. But we might overfit the training data if we only focus on minimizing the empirical risk.

To evaluate the performance of a classifier, we typically split the dataset into training, held out, and testing sets. The training set is used to train the model, the held out set is used for model selection and hyperparameter tuning, and the testing set is used to evaluate the final performance of the model.

Overfitting occurs when a model learns the training data too well, including its noise and outliers, leading to poor generalization to new, unseen data.

15.5. Parameter Estimation

Parameter estimation for the Naive Bayes classifier involves estimating the probabilities $P(Y)$ and $P(F_i \mid Y)$ from the training data. There are two common methods for parameter estimation: Maximum Likelihood Estimation (MLE) and Maximum A Posteriori (MAP) estimation.

15.5.1. Maximum Likelihood Estimation (MLE)

Maximum Likelihood Estimation (MLE) is a method for estimating the parameters of a statistical model by maximizing the likelihood function, which measures how well the model explains the observed data.

$$\theta_{\text{MLE}} = \arg\max_{\theta} P(\mathbb{X} \mid \theta) = \arg\max_{\theta} \prod_{i=1}^{n} P(x_i \mid \theta) \tag{44}$$

15.5.2. Maximum A Posteriori (MAP) Estimation

Maximum A Posteriori (MAP) estimation is a method for estimating the parameters of a statistical model by maximizing the posterior distribution, which combines the likelihood of the observed data with a prior distribution over the parameters.

$$\theta_{\text{MAP}} = \arg\max_{\theta} P(\theta \mid \mathbb{X}) \tag{45}$$

15.6. Laplace Smoothing

One common issue in parameter estimation is the zero-frequency problem, where a feature value does not appear in the training data for a particular class, leading to a probability of zero. To address this, we can use smoothing techniques such as Laplace smoothing, which adds a small constant to all counts to ensure that no probability is zero. This makes each estimates less extreme, which makes it more uniform.

15.7. Confidence

In addition to predicting the class label, the Naive Bayes classifier can also provide a measure of confidence in its predictions by computing the posterior probabilities for each class. The confidence can be interpreted as the probability that the predicted class is correct given the input features.

$$\text{confidence} = \max_{y} P(y \mid x) \tag{46}$$

ML II: Perceptrons

Lecture 21

11/06/25

16. Linear Classifier

A linear classifier makes its predictions based on a linear combination of the input features:

$$y = w_0 + w_1 x_1 + w_2 x_2 + ... + w_n x_n \tag{47}$$

Where:

- w_0 is the bias term.
- $w_1, w_2, ..., w_n$ are the weights associated with each feature.
- $x_1, x_2, ..., x_n$ are the input features.
- y is the output score.

16.1. Decision Rule

The decision rule for a linear classifier is typically based on the sign of the output score y:

- If $y \geq 0$, predict class 1.
- If $y < 0$, predict class 0.

We can visualize the decision boundary as a hyperplane in the feature space.

16.2. Weight Updates / Perceptrons

The perceptron algorithm is an online learning algorithm that updates the weights based on the prediction error for each training example. The update rule is designed to adjust the weights in a way that reduces future misclassifications.

Properties of the perceptron learning rule:

- **Separability:** The perceptron is separable if some parameters correctly classify all training data.
- **Convergence:** The perceptron algorithm converges to a solution if the data is linearly separable.
- **Mistake Bound:** The maximum number of mistakes relates to the margin or degree of separability.

Some problems with the perceptron algorithm:

- **Noise:** if the data is not separable, the perceptron may oscillate and never converge.
- **Margin:** perceptrons doesn't maximize the margin between classes, leading to poor generalization.

16.2.1. Binary Perceptrons

The binary perceptron algorithm updates the weights based on the prediction error:

- For each training example (f, y^*):
  - Classify with current weights to get prediction y.
  - If $y^* \neq y$ (misclassification):
    - Update weights: $w_i = w_i + y^* \cdot f$

16.2.2. Multiclass Perceptrons

For multiclass classification, the perceptron algorithm updates weights for each class:

- For each training example (f, y^*):
  - Classify with current weights to get predicted class y.
  - If $y^* \neq y$ (misclassification):
    - Update weights for true class: $w_{y^*} = w_{y^*} + f$
    - Update weights for predicted class: $w_y = w_y - f$

16.3. Logistic Regression

When the data is not linearly separable, the perceptron algorithm may fail to converge. To address this, we can use a Probabilistic Decision model such as Logistic Regression:

$$P(Y = 1 \mid X) = \sigma(z) = \frac{1}{1 + e^{-z}} \tag{48}$$

When interpreting scores to probabilities, we use the logistic (sigmoid) function:

$$z_1, z_2, z_3 \to \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3}}, \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3}}, \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3}} \tag{49}$$

ML III: Neural Networks

Lecture 22

11/13/25

17. Optimization

In general, optimization refers to the process of finding the best solution from a set of feasible solutions. In mathematical terms, it involves maximizing or minimizing an objective function subject to certain constraints.

17.1. Gradient Descent/Ascent

1-D optimization focuses on finding the maximum or minimum of a function of a single variable. Gradient descent and ascent are popular techniques for this purpose. Gradient descent and ascent is an iterative method that updates the variable in the direction proportional to the gradient:

$$\begin{aligned} x_{\{n+1\}} &= x_n - \alpha \cdot f'(x_n) \quad \text{for minimization} \\ x_{\{n+1\}} &= x_n + \alpha \cdot f'(x_n) \quad \text{for maximization} \end{aligned} \tag{50}$$

where α is the learning rate.

18. Deep Neural Networks

Deep Neural Networks (DNNs) are a class of artificial neural networks with multiple layers between the input and output layers. A deep neural network can also learn the features automatically from the data, eliminating the need for manual feature engineering.

In DNNs, each layer consists of multiple neurons that apply a linear transformation followed by a non-linear activation function. Repeated application of these transformations allows the network to get the set of complex features from the input data. To get the neuron at layer k, we can use the following equation:

$$z_i^k = g\left(\sum_j W_{i,j}^{k-1,k} \cdot z_j^{k-1}\right) \tag{51}$$

where z_i^k is the output of the i^{th} neuron at layer k, $W_{i,j}^{k-1,k}$ is the weight connecting the j^{th} neuron in layer $k-1$ to the i^{th} neuron in layer k, and g is the activation function.

18.1. Training DNNs

Training DNNs involves adjusting the weights of the network to minimize a loss function that measures the difference between the predicted output and the actual output. This is typically done using backpropagation combined with optimization algorithms like gradient descent.

Universal Function Approximation Theorem: A two layer neural network with a sufficient number of neurons can approximate any continuous function to any desired accuracy.

In general, we would we neural networks on complex data where it is difficult to manually engineer features. Examples include image recognition, natural language processing, and speech recognition.

ML IV: Applications & Decision Trees

Lecture 23

11/18/25

19. Foramlizing Learning

19.1. Inductive Learning

In inductive learning, we observe examples and attempt to generalize from them. We do this by creating a function that maps inputs to outputs based on the observed data. The goal is to find a hypothesis that accurately predicts the output for new, unseen inputs. We can formalize this process as follows:

- Target Function $g : X \to Y$
- Hypothesis Space $H = \{h_1, h_2, ..., h_n\}$ where each $h_i : X \to Y$
- Training Data $D = \{(x_1, y_1), (x_2, y_2), ..., (x_m, y_m)\}$ where $y_i = g(x_i)$

Find $h \in H$ such that $h(x_i) = y_i$ for all $(x_i, y_i) \in D$ and h generalizes well to unseen data.

19.2. Bias and Variance

When evaluating a learning algorithm, we often consider two key concepts: bias and variance.

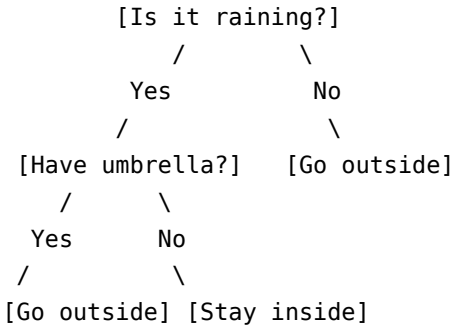
- **Bias** How well the model can spot the pattern. High bias can lead to underfitting, where the model fails to capture the underlying patterns in the data.
- **Variance** refers to the error introduced by the model's sensitivity to small fluctuations in the training data. High variance can lead to overfitting.

To lower variance, we can opeerationalize simplicity by:

- Reduce number of features / Reducing the complexity of the hypothesis space H
- Regularization: Adding a penalty term to the loss function to discourage complex models

20. Decision Trees

Decision trees are a popular method for both classification and regression tasks. They work by recursively partitioning the input space into regions based on feature values, leading to a tree structure where each internal node represents a decision based on a feature, and each leaf node represents a predicted output.
Example:



20.1. Decision Trees vs Perceptrons

Decision trees and perceptrons are both used for classification tasks, but they have different strengths and weaknesses.

- Decision Trees:
  - Can handle both categorical and numerical data
  - Can model non-linear decision boundaries
  - Prone to overfitting if not pruned properly
- Perceptrons:
  - Primarily used for binary classification
  - Can only model linear decision boundaries
  - Less interpretable than decision trees

Algorithm:

We grow the tree recursively by selecting the feature that splits the data at each node into the most homogeneous subsets:

```
function build_tree(data, features):
    if all examples have the same label:
        return leaf node with that label
    if features is empty:
        return leaf node with majority label
    best_feature = select_best_feature(data, features)
    tree = create_node(best_feature)
    for each value v of best_feature:
        subset = filter data where best_feature == v
        child_node = build_tree(subset, features - {best_feature})
        add child_node to tree
    return tree
```

20.2. Entropy and Information

To select the best feature for splitting the data, we can use the concept of entropy and information gain. Information says that the more uncertain about the outcome, the more information is gained when that uncertainty is reduced. Entropy is the expected amount of information needed to classify a randomly drawn example:

$$\text{Entropy}(S) = -\sum p_i * \log_2(p_i) \tag{52}$$

Information Gain is the reduction in entropy after a dataset is split on an attribute. Using this measure, we can select the feature that maximizes information gain at each node in the decision tree.

20.3. P-Chance

P-Chance is the probability to recieve the information gain by chance. We use P-chance to determine if our model is overfitting the data.

ML V: Transformers

Lecture 24

11/20/25

21. Large Language Models

Large Language Models are neural networks trained on vast amounts of text data to understand and generate human-like language. They utilize architectures such as Transformers to capture the context and semantics of language effectively.

21.1. Text Tokenization

Text tokenization is the process of breaking down text into smaller units called tokens, which can be words, subwords, or characters. This step is crucial for preparing text data for input into language models.

21.2. Word Embeddings

Word embeddings are dense vector representations of words that capture their meanings and relationships. We can interpret each word as a vector in space. The embedding vectors are learned during training and help models understand semantic similarities between words.

21.3. Self-Attention

Self-attention is a mechanism that allows models to weigh the importance of different words in a sentence when making predictions. It enables the model to focus on relevant parts of the input sequence, improving its ability to understand context. For example, in the sentence "The cat sat on the mat," self-attention helps the model recognize that "cat" and "sat" are closely related.

Implementation:
Self-attention is implemented using three main components: Query (Q), Key (K), and Value (V) matrices. The attention scores are computed as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{53}$$

A key is a representation of the word to be compared against, a query is the representation of the current word, and a value is the actual information to be aggregated.

21.4. Multi-Head Attention

Multi-head attention extends the self-attention mechanism by using multiple attention heads to capture different aspects of the input. Each head learns to focus on different parts of the sequence, allowing the model to gather diverse information. The outputs from all heads are concatenated and linearly transformed to produce the final output.

22. Transformer Architecture

The Transformer architecture is a deep learning model designed for handling sequential data, particularly in natural language processing tasks. It consists of an encoder and a decoder, both built using layers of multi-head attention and feed-forward neural networks. The process involves the following steps:

1. Input text is tokenized and converted into embeddings.
2. The transformer block processes the embeddings through multiple layers of multi-head attention and feed-forward networks.
3. The output is un-embedded and un-tokenized to generate predictions, such as the next word in a sequence.

23. Unsupervised / Self-Supervised Learning

The basic idea is to use large amounts of unlabeled text data to train language models. The model learns to predict the next word in a sentence or fill in missing words, enabling it to understand language patterns without explicit labels.

23.1. Pre-training and Fine-tuning

Pre-training involves training a language model on a large corpus of text to learn general language representations. Fine-tuning is the subsequent process of adapting the pre-trained model to specific tasks, such as sentiment analysis or question answering, using smaller labeled datasets.

24. Multi-Modal Models

Multi-modal models are designed to process and understand multiple types of data, such as text, images, and audio. These models can integrate information from different modalities to perform tasks like image captioning or visual question answering.

Applications: AI for Healthcare

Lecture 25

11/25/25

25. Discriminative AI for Healthcare

A **Discriminative** AI model is a narrow, task-specific model that is designed to perform a specific function, such as classifying images or predicting outcomes based on input data.

25.1. Applications

In many cases, computer vision models are used to find patterns in medical images that can help with diagnosis, prognosis, and treatment planning. Some common applications include:

- Cardiac Ultrasound
- 4D Flow MRI
- Cardiac MRI

In clinical notes, natural language processing (NLP) techniques are used to extract relevant information from unstructured text data.

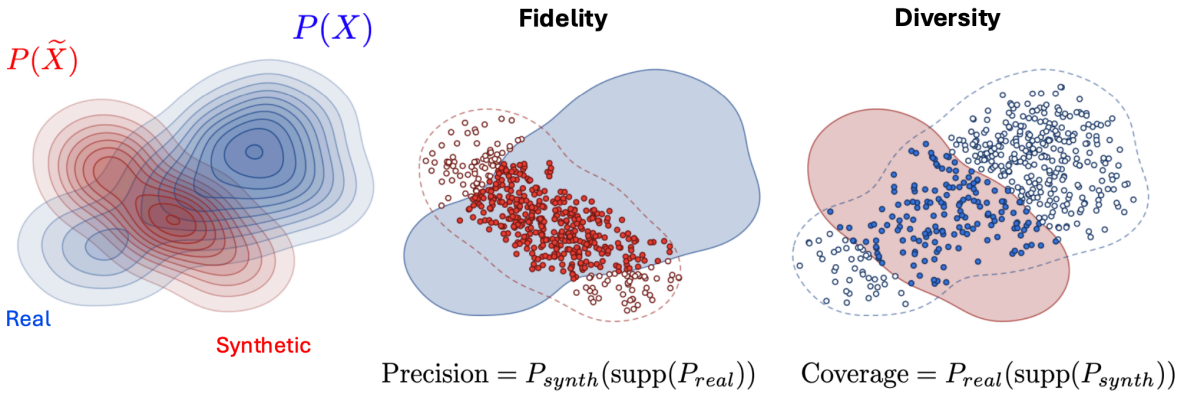
A **Benchmark Dataset** is a standardized dataset that is used to evaluate and compare the performance of different AI models.

25.2. Generative AI Models

Generative AI models are designed to create new content, such as images, text, or audio, based on patterns learned from training data. It learns the underlying distribution of the data and can generate new samples that are similar to the training data.

25.3. Fidelity and Diversity

- **Fidelity** refers to how closely the generated content resembles real-world data. High-fidelity models produce outputs that are realistic and indistinguishable from real data.
- **Diversity** refers to the variety of outputs that a generative model can produce. A diverse model can generate a wide range of different samples, rather than producing similar or repetitive outputs.



25.4. Human Evaluation of Generative Models

Evaluating generative models often involves human judgment to assess the quality and relevance of the generated content. This can include:

- **Visual Turing Test**: Human evaluators are asked to distinguish between real and generated images.
- **Clinical Relevance Assessment**: Experts evaluate whether the generated medical data is clinically useful and accurate.
- **Diversity Assessment**: Evaluators assess the variety of outputs produced by the model.