# Survivor's Manual for CIS 4210/5210

Jeffrey Saito (Hitman7128)

Last Update: December 21, 2024

This course investigates algorithms to implement resource-limited knowledge-based agents which sense and act in the world. Topics include, search, machine learning, probabilistic reasoning, natural language processing, knowledge representation and logic. After a brief introduction to the language, programming assignments will be in Python.

— *Description of CIS 4210/5210 in course catalog*

**Exams in CIS 4210/5210 are closed book but allow one double sided paper as a cheat sheet.**

## Contents

**1 General Test Strategies (addressing both tests)** **1**

**2 Midterm** **2**

2.1 Module 1 - Rational Agents + Python Knowledge . . . 2
2.2 Module 2 - Uninformed Search . . . 3
2.3 Module 3 - Informed Search . . . 5
2.4 Module 4 - Games and Adversarial Search . . . 7
2.5 Module 5 - Constraint Satisfaction Problems . . . 8
2.6 Module 6 - Logical Agents . . . 9

**3 Final** **11**

3.1 Module 7 - Markov Decision Processes . . . 11
3.2 Module 8 - Reinforcement Learning . . . 13
3.3 Module 9 - Probabilities and Language Models . . . 16
3.4 Module 10 - Bayes' Nets . . . 18
3.5 Module 11 - Naive Bayes and Perceptrons . . . 20
3.6 Module 12 - Neural Networks . . . 22
3.7 Module 13 - Natural Language Processing . . . 25

## §1 General Test Strategies (addressing both tests)

- Many topics in the course follow the theme of "easy to boil down to a slogan, but when you get into details, it gets messy very quickly." So I will try to cover the main topics as best as I can and separate most of the noise from the crux of the modules. This isn't an exhaustive list of content, as the textbook covers many of the important details and more obscure subtopics.

- Questions on both tests are very all-or-nothing. There are "select all that apply questions," but if you select one that's wrong or don't select one that's right, you don't get any partial credit.

- That being said, you will need to know many of the details, especially for "select all that apply," so you can truly distinguish which ones are correct and which ones are not. However, that is not possible to effectively achieve without knowing the topics and important machinery well enough.

- Don't underestimate actually practicing algorithm executions! (like the search algorithms) Make some examples for yourself that are feasible to calculate by hand, to get used to the process. Remember, if you make a mistake on the test, you forfeit the points for that question.
- In the first test, there were a fair number of "application" type questions, where you had to apply the knowledge you used rather than reciting it. However, the second test was more skewed towards reciting specific knowledge.
- The first test's cheat sheet should ideally have all the relevant search algorithms, as well as their properties. If you think the execution of any of the algorithms is tricky (like alpha-beta pruning), write an example of an execution on there.
- The second test's cheat sheet should have the information on active paths/triples, as it would be impractical to memorize that. Like the first test, if you're not particularly familiar with an algorithm's execution (like updating weights in a perceptron), write an example down.
- Other than that, it's fairly up to you what else you put on there, but ideally, it should be personalized to your weak topics and should be well-organized.

## §2 Midterm

Modules 1 to 6 were covered.

**Statistics from Fall 2024 iteration** (out of 52 points):

- Mean: 36.15
- Median: 36.27
- Max: 50.27
- Upper Quartile: 41.45
- Lower Quartile: 31.61

### §2.1 Module 1 - Rational Agents + Python Knowledge

#### Python Knowledge

- Python is Strong and Dynamic typed:
  - Strong Typing: The compiler will throw an error if you try do an operation on a type that doesn't work with said operation
  - Dynamic Typing: The same variable can have its datatype changed
- You can use negative indexing of lists to access rightmost elements, i.e. $x[-1]$ accesses the rightmost element of a list $x$.
- The line $y = x[:]$ creates a shallow copy of a list $x$, meaning $x$'s pointers/references within the array are copied to $y$.
- You can get a sublist of a list $x$ from elements in positions $a$ to $b$ (inclusive) via $y = x[a : b]$. If you wanted every $n^{th}$ element in the sublist from positions $a$ to $b$, you would write $y = x[a : b : n]$.
- Some operations you can do on lists are append (add an element to the end), insert at a specific position, get the index of the first occurrence of an element, count how many times an element shows up, and removing an element. You can also reverse and sort an array (with the sorting function being customizable).
- Tuples are a sequence with ordered positions, just like lists, but are immutable (meaning you cannot change it at all after creation).

- A dictionary is like a hash map, where you can insert key-value pairs. Later, if you want to access a value associated with the key, you input the key and then it returns the associated value.
- Dictionary keys can be just about anything as long as you don't have two of the same key and the datatype of the keys is immutable. This means a dictionary key cannot be a list (use a tuple instead) or another dictionary. However, a dictionary *value* can be just about anything, including a list or even another dictionary.
- You can return multiple values in a function, in which the return type is a tuple.
- List comprehension is a shorter way of creating a list using a for loop (or even a double for loop). It's sort of in the reverse order as a for loop: you have the value you want to add to the list and then the actual for loop after.
- Iterator - an object that you can iterator over (like with a for loop) and is more efficient to iterate over than a standard for loop
- Generator - A function that it adds objects to an iterator (with the yield keyword), instead of having to append the objects to a list and returning it

### Agents

- We want an agent to think "rationally," which means doing the right thing and being able to maximize the reward with the information it has. Sometimes, there will be obstacles for the agent, which will be seen in later modules.
- The environment may also have random elements out of the agent's control. In that case, rationality involves maximizes the agent's expected reward.
- A task environment for an agent can be defined with the acronym PEAS:
  - Performance Measure - again, a rational agent wants to maximize this
  - Environment - Basically, the surroundings of the agent. Can be different positions, weather, obstacles, etc. It's very broad what can be included here
  - Actuators - the tools the agent can use to change/interact with the environment
  - Sensors - the tools the agent has to acquire information about the environment
- There are different categories for the environment, some make it more difficult for the agent than others:
  - **Fully Observable vs. Partially observable:** Can the agent see everything that is occurring in the environment and all the information it needs to help decide its actions?
  - **Deterministic vs. Stochastic:** When the agent makes a move with a specific intention, is the outcome always the same? Or is there randomness involved like a slot machine?
  - **Episodic vs. Sequential:** Does one move not impact the choices it may have later on, or does it?
  - **Static vs. Dynamic:** Does the environment not change while the agent is deliberating, or does it?
  - **Discrete vs. Continuous:** Is the selection of choices and environment states discrete (meaning a finite number or countably infinite), or are there an uncountable number of choices like a point along the real line?
  - **Single Agent vs. Multi Agent:** Is the agent by itself or does it have to worry about other agents who may be an opponent/ally?

## §2.2 Module 2 - Uninformed Search

### Introduction

- A search problem, in simplest terms, involves finding a way to get from start to finish, generally with as little cost as possible.

- We will work with deterministic search problems for now, meaning if we execute the same sequence of actions from a particular state, the result is always the same.
- The formal necessities for such a problem are: the set of states, a starting state, a set of actions, a transition model (that tells us where we end up in if we're at a state $s$ and choose some action $a$), a path cost function, and goal state(s).
- Many search algorithms we will see will use a **frontier**, which is essentially a queue (sometimes with priority to certain nodes) that indicates which nodes are being processed at the moment.
- We can handle a search problem in one of two ways:
  - Tree Search: This is when we don't keep track of which nodes we have visited already. We will save space, but we could run into an infinite loop if we have cycles.
  - Graph Search: This is when we *do* keep track of which nodes we have visited already.
- We also have attributes we want to examine for each of the search algorithms:
  - Complete - Does it always get us a solution or could get it stuck?
  - Optimality - Does it get us the minimum cost solution?
  - Time complexity
  - Space complexity
- Also, define the following:
  - $b$ - branching factor (how many neighbors max does any node have)
  - $d$ - depth (shortest number of edges necessary to reach a goal node)
  - $m$ - maximum length path within the state space (could be infinite)

## BFS

- You may remember this back from CIS 1210: we insert nodes into the queue FIFO (first-in-first-out). This means the next node we want to process is at the front and successors get added to the back. We stop when we notice a successor of some node we just processed is a goal state.
- Analysis:
  - Complete: Yes (for finite $b$)
  - Optimal: Yes (if the cost is the same on every edge; otherwise, no)
  - Time Complexity: $O(b^d)$ (because we have an exponential number of nodes per layer with factor $b$, the number of neighbors)
  - Space Complexity: $O(b^d)$
- The exponential factor of space is certainly undesirable and will get worse quickly (as $b$ or $d$ increases).

## DFS

- Another algorithm back from CIS 1210: we go LIFO (last-in-first-out). This means we process the node at the end of the frontier first. This essentially allows us to keep on exploring until we hit a dead end and then we have to backtrack.
- Analysis:
  - Complete: No (we could get stuck going down an infinite path that doesn't have a goal node on it)
  - Optimal: No (DFS can't tell if there is a shortcut along the path)
  - Time Complexity: $O(b^m)$
  - Space Complexity: $O(bm)$ (because we don't branch out like BFS does, it's linear)
- DFS is better if space is more restricted. However, BFS is better if you could have infinite paths and $d$ is small.

### Depth Limited Search

- This tries to resolve the fact that DFS could get stuck in an infinite path.
- We set a fixed number $\ell$ in advance and run DFS, but we force it to never go further than depth $\ell$ and then begin backtracking from there.
- Analysis:
  - Complete: No (if $\ell < d$), Yes otherwise
  - Optimal: No (if $\ell > d$), Yes if $\ell = d$
  - Time Complexity: $O(b^\ell)$
  - Space Complexity: $O(b\ell)$
- So this requires knowledge of $d$ to be effective, which we won't always know.

### Iterative Deepening DFS

- This tries to get the best of both worlds of BFS and DFS.
- We take the Depth Limited Search idea and keep increasing $\ell$ until we find a goal state.
- Analysis:
  - Complete: Yes
  - Optimal: Yes (because it will detect the goal exactly in the iteration where $\ell = d$ and no earlier)
  - Time Complexity: $O(b^d)$
  - Space Complexity: $O(bd)$

### Uniform Cost Search (Dijkstra's)

- If you remember Dijkstra's from CIS 1210, this is another name for it (CCB hates the name "Uniform Cost Search" btw).
- As a reminder, suppose we have a source $s$ and some other node $n$. Then, we have a priority queue where the nodes are ordered by the lowest cost we found thus far to get from $s$ to $n$.
- If we find better paths to $n$ while $n$ is still on the queue, we update its value on the priority queue, so it moves forward in the queue.
- We process nodes on the frontier when it gets to the front and once a node $n$ leaves the queue, it never enters back and we know we found the shortest path to $n$.
- However, unlike BFS, we don't test if we're at a goal state when we examine successors of a node but rather when we remove a node from the queue and notice it's a goal state. This is because if we don't let the node reach the front, we miss out on possible better paths that could be updated on the node before it reaches the front.
- Uniform Cost Search is complete and optimal, so long as the edge weights are nonnegative.

A con with these algorithms thus far is that we see all branching paths as *too* equally good and can easily waste time going off in the blatantly wrong direction. In Module 3, you'll see ways around that.

## §2.3 Module 3 - Informed Search

### Introduction

- So we need a way to not go blatantly in the wrong direction. We need some information that can help us navigate in the right general direction. Introducing...
- **Heuristic functions** - a function that takes in a node $n$ as an input and returns an estimate for the remaining distance to the goal from $n$.

## Greedy Best First Search

- A search algorithm where we rely solely on a heuristic function $h$ for our choices. In particular, when at a node $u$, we always choose the neighbor $v$ of $u$ with the lowest heuristic value. Repeat until we reach a goal node.
- This is not an optimal search algorithm. The locally best choices do not always lead to the overall optimal solution.
- This is not complete either. The heuristic can easily lead us towards a dead end and get us stuck in there.
- So this algorithm is okay if you just desperately want a solution without regards to how optimal it is and with no guarantees that it will even return one.

## Heuristic Brainstorming

- Let's get some intuition on heuristics before we tackle the star of the show.
- **Admissible Heuristic:** A heuristic $h$ that is always nonnegative and never overestimates the true distance from any node $n$ to the goal. This means that we're never overly pessimistic about ignoring what-would-be ideal paths to the goal.
- **Consistent Heuristic:** A heuristic $h$ that satisfies a "triangle inequality": where for all nodes $u$ and successors $v$ of $u$, we have
$$h(u) \leq \text{cost}(u \text{ to } v) + h(v).$$
This essentially means the heuristic is not too optimistic to the point where it sends us on detours detracting us from the optimal path.
- One key property is that any consistent heuristic is also admissible, but not all admissible heuristics are consistent.
- An easy way to create a heuristic that is admissible is to temporarily ignore restrictions until it becomes something that you can easily compute. This is because removing restrictions opens up lower cost "solutions" that may not actually work but are a good enough to never overestimate the true cost.

## A* Search

- We choose a heuristic function $h$. Also, for a node $n$, we define $g(n)$ to be the true distance from the start to $n$. Then, we define $f(n) = g(n) + h(n)$. We have a priority queue using $f(n)$ as the key. Then, it's the same old from there: we always choose to expand the node $u$ at the front of the frontier and then add its successors $v$ with key $f(v)$ (or decrease the key if $v$ is already in the frontier).
- If the heuristic is inadmissible, you cannot guarantee optimality or completeness.
- What about when the heuristic is admissible? Well, it's complicated. First, we define two approaches to the **graph** search for A*:
  - Open Set: We allow nodes to be revisited and decrease keys within the frontier if we find better paths to a node already in the frontier.
  - Closed Set: We never revisit nodes, not even to decrease keys.
- Then, depending our approach for A* (Tree/Graph):
  - **Tree search** is optimal as long as the heuristic is admissible (doesn't have to be consistent)
  - **Graph search** with an open set is also optimal as long as the heuristic is admissible
  - **Graph search** with a closed set is tricky: if the heuristic is admissible but not consistent, then you can NOT guarantee optimality. However, you can if the heuristic is consistent.

## §2.4 Module 4 - Games and Adversarial Search

### Introduction

- First, we specify what type of "games" we will be analyzing:
  - Multiagent: There is more than one player
  - Zero-sum: There players are actively competing against each other to maximize their own score (which involves trying to screw with their opponents as much as possible)
  - Finite number of possible states
  - Fully observable: Nothing is hidden from either player
  - Deterministic
- We also need to define different parts of the game, namely: the initial state, the set of legal moves, a condition for when the game ends (otherwise, the terminal states), and the reward/utility function.
- The focus will be on 2-player games, with the players being named MAX and MIN. We will declare MAX to go first and MAX is attempting to maximize their own score, while MIN is attempting to screw over MAX as much as possible to minimize MAX's score (and thus, MIN maximizes their own score).

### Minimax

- Essentially, the game will be represented as a tree, where nodes represent different configurations of the environment. Edges from a node represent legal moves from the configuration, so the node's children correspond to the resulting configurations for each move.
- The layers in the tree alternate between MAX's turn and MIN's turn. In particular, the root node of the tree represents the start where MAX has the first move.
- We'll have a value function $f$ that measures some value of the state in whether it is in MAX's favor or against. The higher the $f$ value, the more favorable the state is for MAX. Base cases will be in terminal states.
- However, if the state space is so large that it's impractical to search through it entirely (like chess), we may limit the depth of the tree and estimate $f$ based on features available on the board (like how many pieces of each type we have left compared to the opponent).
- This algorithm is recursive in nature. When we're at a node $n$ for MAX's turn, it will recursively compute the values of all children nodes of $n$. Then, it will select the one with the highest $f$ value (since MAX is trying to maximize its score).
- On the other hand, when we're at a node $n$ for MIN's turn, it will still recursively compute the values of $n$'s children. However, it will select the one with the lowest $f$ value (to screw MAX over).
- The intuition is the whole zero sum game concept, where MAX wants to make the best choice but MIN wants MAX to have a worse score for MIN to have a better score for itself (since it's zero-sum).

### Alpha-Beta Pruning

- Oh boy, this topic is tricky. Someone did provide a helpful YouTube video so that the process is easier to visualize: `https://www.youtube.com/watch?v=l-hh51ncgDI`
- At a high level, Alpha-Beta pruning is a method of removing unwanted branches of the search tree that we know are not going to have any effect on the outcome.
- For each node $n$, we have two variables $\alpha$ and $\beta$. We have that $\alpha$ represents the best score MAX could do for itself using a path from the root including $n$. We have that $\beta$ represents the best score MIN could do for itself using a path from the root including $n$. By default, every node has $\alpha = -\infty$ and $\beta = \infty$.

- Our pruning condition is $\alpha \geq \beta$. If that condition ever occurs at a node, we prune (or cut off) the rest of the children edges attached to it.
- How do we manage $\alpha$ and $\beta$ as we traverse the tree? When we go down the tree, we always propagate the $\alpha$ and $\beta$ values from parent to child.
- However, when we're at node $n$ and as we recursively figure out the values $n$'s children, we update $\alpha$ or $\beta$. In particular, if we're at a MAX node and we find a child node with value $v$ s.t. $v > \alpha$, we update $\alpha$ to $v$ at $n$.
- If we're at a MIN node instead and we find a child node with value $v$ s.t. $v < \beta$, we update $\beta$ to $v$ at $n$.
- Here's some intuition behind alpha-beta pruning: Suppose we're at the root (which is MAX) and we find its first child node has value 3. So we have value 3 on the table. Then, when exploring the second child node $c$ (which is a MIN), it discovers a child of $c$ with value 2. So if MAX goes to $c$, MIN can send it to that child of value 2, which is worse than 3, so why bother any more with the other choices? The other choices won't change the fact that going to $c$ means a value of 2 (worse than 3) or even worse.

### Expectimax

- In the case that there is non-determinism or random events, we need to slightly modify our approach.
- In particular, we'll represent game states where a random event occurs as "chance nodes."
- To get the value of a chance node, we compute the expected value. In particular, we take into account the probabilities of the actions from that chance node and the values of the children nodes generated by the actions.

## §2.5 Module 5 - Constraint Satisfaction Problems

### Introduction

- We've solved problems with the intent of trying to get an optimal solution (particularly search algorithms and minimax). But how about just trying to get **a** solution in conditions where it may be elusive to find due to constraints in place? These are Constraint Satisfaction Problems.
- A CSP (Constraint Satisfaction Problem) consists of:
  - Variables $X_1, X_2, \cdots, X_n$
  - A domain for each variable (a set of values a variable can take on)
  - Some number of constraints (for example, $X_1 = X_2 + 1$ or $X_4 > X_3$)
- Our goal is to get a **consistent** assignment of variables, which means all the provided constraints are followed simultaneously under the assignment.

### Constraints and Local Consistencies

- Types of Constraints:
  - Unary - Involves a single variable (like $X_1 \neq 1$)
  - Binary - Involves 2 variables (like $X_1 = X_2$)
  - Higher-order - Involves 3 or more variables
  - Preference - an optional constraint to satisfy but we would be rewarded more if we did follow it
- To keep things simple, we should focus on enforcing the unary and binary constraints. In particular, this leads to "local consistencies," where we just focus on these two categories of constraints.
- Handling unary constraints (called node consistency) is easy, but handling binary constraints (called arc consistency) requires more care.

- We'll create a graph, where the variables are nodes. In particular, if there is a binary constraint between variables $A$ and $B$, we draw edges $A \to B$ and $B \to A$.
- Arc Consistency: Suppose variables $A$ and $B$ have a binary constraint. Then, we say variables $A$ and $B$ are consistent if for every possible value for $A$, we have some value for $B$ that doesn't violate the binary constraint.
- The arc consistency is a way of making progress because if there is a value of $A$ for which you can't find a matching $B$ (without violating the constraint), we might as well remove that value of $A$. Furthermore, since $A$ lost a value, that could affect other variables that are connected to $A$ (because a third variable $C$ may have a value that relied on that lost value of $A$ to have the constraint be satisfied). So we can quickly create a chain reaction of removing values on variables that we know won't work.
- These ideas naturally lead into an algorithm.

### AC-3

- First, we create the graph as described where nodes are variables. Also, for any variables $A$ and $B$ that have a binary constraint involving them, draw edges $A \to B$ and $B \to A$.
- We also initialize each variable's domain to be their entire provided domain (for now). We will soon try to carefully eliminate values from each variable's domain that don't end up working.
- Then, create a massive queue for all arcs (or edges) $(A, B)$ in the graph.
- We process edges in the queue as follows: we pop an element $(X, Y)$ from the queue. We check if $X$ is consistent with $Y$, we ignore $(X, Y)$ and move on to the next element in the queue.
- Otherwise, we remove all values of $X$ from its domain that are not consistent with $Y$. Since we ended up removing a value from the domain of $X$, we have to add all edges in the form $(Z, X)$ (as $Z$ ranges across all neighbors of $X$, besides $Y$). Remember, this is the fact that a value of $Z$ could've relied on the removed value(s) from $X$, so we need to check these arcs again for consistency.
- Basically, we just repeat this until the queue empties out.

The AC-3 algorithm can sometimes lead to a solution, but not always. If we get stuck, we may need to be more ambitious in trying to get a solution with the benefit of AC-3 limiting the possible values for each variable. In particular, we could use a DFS combined with guess and check so that we can try different values out from the more limited selection (to save time) and also have the ability to backtrack if we make a mistake.

## §2.6 Module 6 - Logical Agents

### Introduction

- Sometimes, the environment itself may not be observable. Remember in Module 1 about sensors? The agent can use its sensors to gather information about the environment and sometimes, make outright conclusions.
- It stores all its knowledge in a **knowledge base**. In there, we have two types of statements: **axioms**, which the agent assumes to be true by default, and **derived sentences**, which are statements that the agent concludes from the axioms and knowledge it has acquired.
- In the Wampa World example presented in class, we have a robot wandering around a grid. Depending on whether it picks up stench, a breeze, or some other percept at a tile in the grid, it will remember that in its knowledge base.
- The knowledge base will expand as more of the environment is explored. The robot then uses axioms such as a breeze tile implying there is a pit tile adjacent to it to derive more knowledge. Eventually, the robot can piece together the knowledge to conclude stuff like "This combination of percepts is only possible if a pit is at this tile!"

## Logic

- So as the robot acquires information, it can create **models**, which are possible configurations of the environment that are consistent with all the information the robot has picked up so far.
- Suppose we have statements $\alpha$ and $\beta$. Then, we say $\alpha$ entails $\beta$ (denoted as $\alpha \models \beta$) if, for every model where $\alpha$ is true, $\beta$ is also true. Essentially, $\beta$ is a more broad statement than $\alpha$ that we can obtain if we find $\alpha$ is true.
- Using entailment to discover these more broad statements is called **logical inference**. We can also design algorithms to find such inferences.
- A logical inference algorithm is **sound** if the entailed conclusions derived are always correctly entailed. The algorithm is **complete** if it manages to derive all entailed conclusions that are possible from the given knowledge base.
- If you remember Truth Tables from CIS 1600, you can show equivalence of logical statements. For example, DeMorgan's Laws state $\neg(\alpha \vee \beta) = (\neg\alpha \wedge \neg\beta)$ and $\neg(\alpha \wedge \beta) = (\neg\alpha \vee \neg\beta)$ for any statements $\alpha$ and $\beta$.
- In particular, using logical equivalences is very powerful in cleaning up our knowledge base, so it's not a disorganized mess of statements.
- There are some self-explanatory combinations of simplifications we can make. For example, if we have both $\alpha$ and $\alpha \implies \beta$ in our knowledge base, we can conclude $\beta$ (and then we no longer need the implication).
- However, we will see a way to clean up the knowledge base into a nicer form.

## CNF

- First, to describe CNF (conjunctive normal form), a **literal** is either a statement in its untouched form or the negation of a statement (either $\alpha$ or $\neg\alpha$ for some statement $\alpha$). Then, a clause is a series of literals joined with ORs (such as $(\alpha \vee \beta \vee \neg\gamma)$).
- Then, a CNF is a some number of clauses joined by ANDs ($\wedge$).
- We have a procedure to turn a knowledge base into CNF form, where we lay out our statements from the knowledge base and join them with ANDs:
  - If we have $\alpha \iff \beta$ in our knowledge base (for some statements $\alpha$ and $\beta$), change it to $(\alpha \implies \beta) \wedge (\beta \implies \alpha)$.
  - If we have $\alpha \implies \beta$, change it to $\neg\alpha \vee \beta$.
  - Move negations inside clauses (i.e. we don't want ¬(some clause)) so that they directly modify the statements inside the clauses. Then, remove double negations and use DeMorgan's laws (when necessary).
  - Then, use distributivity rules of $\vee$ over $\wedge$, which is given by

$$(\alpha \vee (\beta \wedge \gamma)) = ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))).$$

- Essentially, we use logical equivalences to slowly remove unwanted operations until we get just ORs and ANDs.

## Applications of CNF

- The CNF gives us a useful form for proving theorems, where we can be more ambitious about making assumptions and see if that leads us to anything useful.

- We can utilize a technique called **resolution** to eliminate literals that are opposites of each other. Suppose we had $a \lor b \lor c$ and $\neg b$ as clauses in our CNF. Well, since we know not $b$ is true, we can't satisfy the OR with $b$ being true, so we can eliminate $b$ from it to change it to $a \lor c$.
- Remember proof by contradiction? We can use that idea if we wanted to show the knowledge base entails some statement $\alpha$. We go about it by assuming $\neg\alpha$ is true and then add it to the knowledge base.
- From there, we can use the resolution technique to repeatedly eliminate variables from other clauses. If we end up with an empty clause, that means we have a contradiction (basically saying we cannot have the knowledge base satisfied if we assume $\neg\alpha$). However, if we don't end up with an empty clause, we can't conclude anything from the "proof."

Also, using the knowledge base, we can derive processes to try to reach a conclusion we might want to prove. We might want to work forwards from what we know until we reach a chain of implications that end in the desired conclusion being true. This is **forward chaining**. Or we can work backwards from the goal in **backward chaining**. These proof techniques are similar to how you may go about writing a proof in the theory CIS classes here at UPenn!

## §3 Final

Modules 7 to 13 were covered (so it is **not** cumulative).

**Statistics from Fall 2024 iteration** (out of 34 points):

- Mean: 22.21
- Median: 22.85
- Max: 30.8
- Upper Quartile: 25.65
- Lower Quartile: 19.1

### §3.1 Module 7 - Markov Decision Processes

#### Introduction

- Up to this point, the environments have always been deterministic: if you start a particular state and execute the same set of actions, the outcome is always the same. However, from now on, we'll be dealing with non-deterministic (or stochastic) environments, where the same action doesn't necessarily produce the same outcome.
- We can no longer use a sequence of actions as a "solution," but rather a policy that tells us what our best move is in any given situation.
- This leads to a class of problems called "Markov Decision Processes" or MDPs. These are sequential, fully observable, and stochastic with additive rewards.
- To formally define an MDP, we need the following:
  - Set of states $S$
  - Set of actions $A$
  - Transition function $T(s, a, s')$, where $s, s' \in S$ and $a \in A$. It will tell us the probability that, given we're in state $s$ and we select action $a$, we end in state $s'$ after doing action $a$.
  - Reward function $R(s, a, s')$ that tells us what reward we get if we're at state $s$, do action $a$, and end up in state $s'$ (this reward could be negative)
  - Starting state $s_0 \in S$

- Again, the solution to an MDP takes the form a policy $\pi$, but we objectively want one that maximizes our expected reward gain.
- The rewards are additive, which can create problems. Depending on our rewards, the optimal policy can be undesirable. If the rewards are all positive and don't change in value as the agent makes moves, it can lead to a situation where the agent can goof around because there's no time incentive. If the rewards are all negative, it can lead to a situation where the agent makes a beeline to a negative terminal state (if a positive terminal state doesn't exist or is too far).
- There are 3 ways to create a "time incentive":
  - Create a finite horizon, meaning we hard-stop after $T$ moves, for some fixed number $T$.
  - Create a discounting factor $\gamma$, where $0 < \gamma < 1$. In particular, after the agent makes $t$ moves, each reward will only be worth $\gamma^t$ times its original value (akin to exponential decay).
  - Create an absorbing terminal state, meaning no what actions the agent will make, it will have to reach a terminal state after a finite number of moves.

## Value Iteration

- This is a way of getting the expected values of being at a particular state. Our search tree will be like expectimax. We will also implement $\gamma$ (the exponential decay factor of rewards).
- We define $V_k(s)$ to be the expected value of our reward sum given that we're acting optimally, we're in state $s$ and have $k$ moves left.
- Our base case is $V_0(s) = 0$ for every state $s \in S$. If we have no moves left, we have no rewards we can collect.
- Now, we will compute values of $V_{k+1}(s)$ recursively:

$$V_{k+1}(s) = \max_{a \in A} \left\{ \sum_{s' \in S} \left( T(s,a,s')[R(s,a,s') + \gamma \cdot V_k(s')] \right) \right\}$$

- This is called the Bellman equation. To interpret it, consider the expression inside the max, but arbitrarily fix the $a \in A$:
  - The sum is essentially an expected value over all states $s' \in S$ that we could transition to.
  - The $T(s,a,s')$ term is the probability we go to state $s'$ and then $R(s,a,s') + \gamma \cdot V_k(s')$ is our reward for going to state $s'$.
  - Namely, we reap the $R(s,a,s')$ immediately and then we're at a state where we're at $s'$ with $k$ moves left.
  - However, the exponential decay means the reward is hit with a factor of $\gamma$, so the future reward is $\gamma \cdot V_k(s')$.
  - So the expression inside the max is a way of saying the expected reward for choosing action $a$.
- But that's of course for a particularly fixed action $a \in A$ that we get to choose. We obviously want to choose the one maximizing our expected reward, so we choose the $a \in A$ that maximizes the quantity.
- Remember our time incentives? We could put a hard-cap on $k$ to signify a limit on the number of moves in order for the search tree to be finite. But it turns out, when $\gamma < 1$, the values of $V_k(s)$ (for each $s \in S$) converge as $k \to \infty$.
- When the search tree is finite, it's obvious why the values will converge. But when it's infinite and $\gamma < 1$, the intuition is that the sum of an infinite geometric series with factor $\gamma < 1$ always converges. So the idea is you can get an upper bound on your max reward by a constant. You don't need to understand the math, just the intuition. Point is, under either of those two conditions, *the $V$ values always converge*.

- The values converging naturally gets us the policy $\pi$ in which, for each state $s$, we just pick the action $a$ that leads to the best expected reward.
- However, value iteration is fairly slow with runtime $O(|S|^2|A|)$ per iteration ($|S|^2$ comes from an $(s, s')$ pair and $|A|$ comes from the choices for $a$).

### Policy Iteration

- The next idea is to iterate policies and keep improving them.
- Consider any fixed policy $\pi$ (its optimality doesn't matter). Define $V_k^\pi(s)$ to be the same as $V_k(s)$, but we stick with policy $\pi$ when iterating values.
- We'll reuse the Bellman equations from value iteration, but now we have

$$V_{k+1}^{\pi}(s) = \sum_{s'\in S}\big(T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma \cdot V_k(s')]\big).$$

  The idea is we now longer have a max, since our policy $\pi$ tells us which action to take at state $s$ (namely $\pi(s)$). This means it's more efficient per iteration! It will be $O(|S|^2)$ work per iteration, since we don't explore every action $a$.
- Using this idea, we can construct a sequence of policies $\pi_1$, $\pi_2$, $\cdots$.
- First, we start off with an arbitrary but fixed policy $\pi_1$. Then, if we want to go from policy $\pi_i$ to $\pi_{i+1}$, here's what we do:
  - Compute values of $V_k^{\pi_i}(s)$ for all integers $k \geq 1$ and $s \in S$ until we converge. We can define convergence as the values from iteration $k$ to $k+1$ barely changing at all, or something the lines of $|V_k^{\pi_i}(s) - V_{k+1}^{\pi_i}(s)| < \epsilon$ for some very small positive $\epsilon$ (say $\epsilon = 10^{-6}$).
  - Once we're satisfied with our $V$ values for $\pi_i$, we use them to get $\pi_{i+1}$. We do a one step look ahead:

$$\pi_{i+1}(s) = \operatorname*{argmax}_{a\in A}\left\{\sum_{s'\in S}\big(T(s,a,s')[R(s,a,s') + \gamma \cdot V^{\pi_i}(s')]\big)\right\}$$

    Basically at each state $s$, to determine $\pi_{i+1}(s)$, we choose the best action based on the converged $V$ values for $\pi_i$.
- It turns out no matter how bad our initial policy is, this policy iteration algorithm will always converge to the optimal one!
- Policy iteration will get you the optimal policy much faster than value iteration will, since the $V$ values will be improving ever so slightly, but the max action $a$ at a state $s$ will no longer change after enough iterations. So if you just want the optimal policy, use policy iteration. If you want the values, you have to use value iteration.
- Another idea is that since the maxes are gone under a fixed policy $\pi$, the Bellman equations are a linear system of equations.

## §3.2 Module 8 - Reinforcement Learning

### Introduction

- In MDPs, the environment was stochastic, but the transition probabilities and rewards were available to us. However, this will be a different story where we have neither of those to start with.
- Also, there will have to be a balance between trying new things and trying familiar things. Do we explore the possibility of a new thing having a better/worse reward? Or do we explore a familiar thing where we already have a good idea of what to expect?
- Key concepts:

  - Exploration - As stated, we need to try new actions out
  - Exploitation - However, we might want to stick to what we know is familiar and reap rewards that way
  - Regret - A way of looking back on hindsight and seeing how ideal our exploration methods were
  - Sampling - We will have to experience a state, action combination multiple times to better understand the probabilities, since we don't know them right away
  - Difficulty - Obviously, this type of problem is generally harder than an MDP

- One attempt at this will be "Model-Based Learning," where we guess the transition function $T$ and reward function $R$ based on prior experience. A naive way to go about this would be to determine $T(s,a,s')$ based on averages (like if we noticed for some fixed $(s,a)$ that we go to a state $t$ half the time, we guess $T(s,a,t) = 0.5$).

## Passive Reinforcement Learning

- This is a learning method, where we're given a fixed policy $\pi$. Then, whether we like the policy or not, we follow it precisely to determine the actions.
- Suppose we kept following this same policy over the course of many samples. This will get us estimates for the $V(s)$ values by taking a particular state $s$ and averaging the cumulative rewards from every time we visited $s$ in the samples.
- So suppose we visit $s$ 3 times over the course of all samples. We observe the first time, the rewards we got from $s$ to the end of that episode (where we first visit $s$) is 50. Then, suppose for the second and third times, we get 15 and 25, respectively. Then, our average for $V(s)$ is $\frac{50+15+25}{3} = 30$.
- The good news is, we don't need the values of $T$ and $R$ for this method.
- The bad news is, it's time-consuming. Also, our experience of each state is treated independently (nowhere in our average do we use the fact that $s$ might be connected to other states), so we're not utilizing the connections between states.

## Temporal Difference Learning

- Instead of just taking the average of our samples to get estimates for $V$ values, we can weight the average! This is called Temporal Difference Learning.
- If we still assume we use a fixed policy $\pi$, suppose we're at $s$. Then, we'll have the agent do its thing from there and collect rewards until the end of the episode, so we get a sample of the total reward $x = R(s, \pi(s), s') + \lambda \cdot V^\pi(s')$. Then, we update

$$V^\pi(s) \leftarrow (1-\alpha)\cdot V^\pi(s) + \alpha \cdot x$$

  for some $\alpha$ between 0 and 1. The idea is we incorporate the new data into our estimate for $V^\pi(s)$, but we can decide how much it should be incorporated.

- The $\alpha$ variable is called the "learning rate." The higher it is, the more the recent samples will contribute to the weighted average. We can lower $\alpha$ once we decide we have enough samples and that future samples won't be as impactful on the weighted average.
- A boon with this weighted average is that older samples get buried under exponential decay, so wrongful first impressions will not matter once we have enough samples.
- Unfortunately, values themselves don't get us a best policy: remember how we get the best policy? We choose the action $a$ from state $s$ that gets us the best expected future reward (since the stochastic environment means we can't always get from $s$ to the state with the best $V$ value). So we define

$$Q(s,a) = \sum_{s' \in S} \big(T(s,a,s')[R(s,a,s') + \gamma \cdot V(s')]\big)$$

which is essentially the expected future reward from being in state $s$ and choosing action $a$. These $Q$ values will be more important for iteration than $V$ values.

## Active Reinforcement Learning

- Here, we'll be more ambitious about choosing our actions for exploration vs exploitation. This is unlike passive reinforcement learning, where a policy entirely determined how we learn about the unfamiliar environment.
- First, we want to learn about $Q$ values rather than $V$ values. This will be done through "$Q$-learning."
- We borrow the idea from iterating $V$ values back in MDPs to create an equation for iterating $Q$-values. We have $Q_k(s,a)$ be the expected future reward from being in state $s$ and choosing action $a$ but we have $k$ moves left. Obviously, $Q_0(s,a) = 0$ always.
- Then,
$$Q_{k+1}(s,a) \leftarrow \sum_{s' \in S} \Big( T(s,a,s')[R(s,a,s') + \gamma \cdot \max_{a' \in A}(Q_k(s',a'))] \Big).$$
Very similar to the Bellman equation for $V$, where to take the expected value over the states $s'$ that we could end up in after. However, the $V$ term at the end is replaced with a max over $Q_k(s',a')$. Why? Because that term initially symbolized the later rewards now that we have $k$ moves left. But we want to write things in terms of $Q$, so when we have $k$ moves left, we're at some $s'$ and we want to choose the action $a'$ from there that gets us the best future rewards.
- We can also sample $Q$ values into a weighted average with $\alpha$ just like we did for Temporal Difference Learning.
- Now, to implement exploration vs exploitation and urge us to try an unexplored $s,a$ combination.
- The easiest way is an $\epsilon$-greedy Policy, where with some small probability $\epsilon$, we choose our action randomly and then all other times, we act according to the policy. However, this won't be helpful when we've explored most of the world.
- Another way is an exploration function $f(u,n) = u + \frac{k}{n}$, where $u$ is a $Q$-value, $k$ is a constant, and $n$ is the number of times we visited a state. The $\frac{k}{n}$ is an additional bonus to the $Q$-value that is a huge bonus when $n$ is small (meaning we haven't tried a specific $(s,a)$ pair before) but gets smaller as $n$ increases. By tacking on the $\frac{k}{n}$ to a $Q$-value estimate, we can be urged to take more unfamiliar actions, since in a policy, we take the $a$ with the largest $Q(s,a)$ value.
- We have the notion of regret, which as alluded to in the introduction, is a way of looking back on hindsight.

## Approximate Q-Learning

- Sometimes, there may be *too* many states like in Pac-Man that finding all $Q$ values is impractical! This is what Approximate Q-Learning is for.
- Like in Adverserial Search, we can approximate how a $Q$ value is with features (like in Pac-Man, how far away you are from a ghost) and take a weighted sum:
$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \cdots + w_n f_n(s,a),$$
where the $w_i$ are weights and $f_i(s,a)$ are features.
- We may not know what weights to assign to the features, but we can figure that out as we get more samples.
- Suppose we get a sample for $Q(s,a)$ and say its value is $t$. We compare that new sample to the current value of $Q(s,a)$ by defining $\Delta = t - Q(s,a)$.

- Then, we want to update weight $w_i$ by adding $\alpha \cdot \Delta \cdot f_i(s,a)$ to it (where we use the $f_i(s,a)$ value obtained from our sample).
- The idea is, if $\Delta$ is big, that means the new sample was much better than our current value for $Q(s,a)$, so *something* had to go correct. In particular, the features $f_i(s,a)$ that had higher values for that sample have their weights increased more (and even more so if $\Delta$ was big). This is a way of saying "Thanks to these features being active at the time for this wonderful sample, these are valuable features to have in general, so the weights should be more favorable/positive to them."
- The opposite is true when $\Delta$ is negative, where something went wrong and we want to blame the features that were active at that time, so we know states that repeat those features are most likely bad.
- Unfortunately, Approximate Q-Learning doesn't always converge.

## §3.3 Module 9 - Probabilities and Language Models

### Introduction

- This should all familiar to you if you've taken STAT 4300, but I will try to cover all the ideas concisely.
- Important Concepts:
  - Observed Variables - Variables that we know the values of
  - Unobserved Variables - Variables that we don't know the values of. Often our goal is to use information about the observed variables to guess what the unobserved variables are.
  - Model - A way of listing out the possible outcomes and their associated probabilities.
- Random Variable - a variable whose value depends on random events.
- Given a fixed set of random variables, the best way to describe a "world" is some combination of values for each random variable.
- Each world $w$ has probability $P(w)$ between 0 and 1 (inclusive). The sum of $P(w)$ over all possible worlds $w$ is 1. These are the axioms of probability.
- Sometimes, we may be given a "Probability Distribution" of a random variable $X$, which indicates which values $X$ can equal and how likely each value of $X$ is. We also have "Joint Distribution" for when we have multiple random variables and want to see how likely a combination of values is for the random variables.
- An event $E$ is merely a set of outcomes that follow a condition (the event that a dice rolls an even number, for example).
- If we are given multiple random variables, but only care about the values of 1 particular random variable $X$, we can create a "Marginal Distribution." Group together worlds with the same value of $X$ and sum the probabilities in each group together to get the distribution for $X$.
- Remember the conditional probability formula $P(a \mid b) = \frac{P(a,b)}{P(b)}$? (commas are often use between events to mean "and") We can create a conditional distribution for two random variables $X$ and $Y$, where we find $P(X = x \mid Y = y)$ for every $x$ that is a possible value of $X$ and every $y$ that is a possible value of $Y$.
- Conditional probability is the most important topic here, since we make guesses based on information we have. For example, if we see a puddle on the sidewalk in the morning, we can make a guess that it most likely rained last night (even if we don't know for sure whether it rained last night).
- We can rearrange that conditional probability formula to get the "Product Rule": $P(y) \cdot P(x \mid y) = P(x,y)$.

- We can also repeatedly apply the Product Rule for multiple events to get the "Chain Rule":

  $$P(x_1, x_2, \cdots, x_n) = P(x_1) \cdot P(x_2 \mid x_1) \cdot P(x_3 \mid x_1, x_2) \cdot \ldots \cdot P(x_n \mid x_1, x_2, \cdots, x_{n-1}).$$

  It means we can calculate a joint probability by ensuring each event occurred in succession: we first ensure that $x_1$ occurred and then given that $x_1$ occurred, we ensure $x_2$ occurred. In general, we ensure the next event occurs given that all the previous ones occurred.

- There's also the Bayes' rule, regarded to be the single most important probability rule for AI. We have $P(x, y) = P(x \mid y)P(y) = P(y \mid x)P(x)$, which implies

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

  It's important because we can switch the conditional and often times, one conditional is easier than the other.

## Probabilistic Language Models

- The idea is that if we're given a couple words of context, a probabilistic language model can guess the next word (or see how likely a particular sentence is).

- Let's suppose we have a phrase with words $w_1$, $w_2$, $\cdots$, $w_n$ in that order. We can utilize the chain rule:

  $$P(w_1, w_2, \cdots, w_n) = \prod_{i=1}^{n} P(w_i \mid w_1, w_2, \cdots, w_{i-1}).$$

  Note that the conditional probabilities complicated, especially $i$ gets large and we have to consider the intersection of a bunch of events.

- We can remedy the complication by using the "Markov assumption", which is that the probability of the next word doesn't depend on the entire history of words before it but rather a couple words before it.

- We can use "$k$-grams", which look at $k-1$ prior words of history, instead of all prior words in order to approximate the probability. If we set $k = 3$ for example, then the next word only depends on the two prior words and then our probability is

  $$P(w_1, w_2, \cdots, w_n) \approx \prod_{i=1}^{n} P(w_i \mid w_{i-2}, w_{i-1}).$$

  Much simpler to consider!

- Now to actually compute these conditioned probabilities given a sample. Suppose we want to compute $P(w_i \mid w_{i-k+1}, \ldots, w_{i-1})$ (a $k$-gram). Using the conditional probability formula, basically, we should have the success instances on the numerator and the number of instances we're conditioning on in the denominator. So the numerator should be the number of times we actually have $w_{i-k}, \cdots, w_i$ in the sample. The denominator should be the number of instances of $w_{i-k}, \ldots, w_{i-1}$ in the sample.

- You do have to be careful with the probabilities: one problem is that certain sentences or combinations of words may not appear in our training data, even if they form plausible sentences. So the denominator could turn out to be 0. That would make the conditional probability moot and undefined in that case. We'll see a way to get around this problem later.

- We often calculate probabilities in log space to avoid underflow.

- For a sentence $w_1, w_2, \cdots, w_n$, we can define its perplexity to be

  $$\left(\frac{1}{P(w_1, w_2, \cdots, w_n)}\right)^{1/n}.$$

  This is a metric that tells us how likely we are to correctly predict the next word. In particular, the lower perplexity, the better.

## §3.4 Module 10 - Bayes' Nets

### Introduction

- Bayes' Nets, at a high level, are a way of seeing how random variables interact, so it's easier to make predictions on unobserved values through conditional probability.
- Independence can be a blessing and a curse. It's a blessing in that it reduces the number of rows in the probability table, since we can consider certain variables separately. But it's a curse, since if two variables are independent, one variable's value doesn't help us at all on making inferences about the other variable's value.
- Total independence is also fairly rare, so we introduce a new notion of independence, called "Conditional Independence."
- We say $X$ is independent of $Y$ given $Z$ (notated as $X \perp\!\!\!\perp Y \mid Z$) if $P(x, y \mid z) = P(x \mid z) \cdot P(y \mid z)$ for all $x, y, z$ that $X, Y, Z$ (respectively) can equal.
- The idea with conditional independence is that once the value of $Z$ is known, any information on what $X$ is won't change our perception on what $Y$ is and vice versa. For example, observing there's a snowstorm will lead us to believe that we will get a snow day and that it will be freezing outside. But once we know there's a snow storm, we already have an idea of the probability of a snow day and the probability it's freezing outside. Seeing that there's a snow day will make us go "Yeah, that's not going make me think it's any more likely to be freezing out because it's no surprise already."

### Creating the Bayes' Net

- Our goal with a Bayes' Net is to get joint probabilities built from simpler interactions between variables.
- It will be a directed graph of random variables where there's an edge $A \to B$ (for random variables $A$ and $B$) if the value of $B$ depends on the value of $A$.
- In essence, the probability table for a random variable $X$ will only depend on $X$'s parents (and not *every* other random variable in the graph).
- We can use the edges to model causation like "The weather causes traffic." While it can be useful to think about, it's not mathematically necessary.
- We can also use it for correlation: imagine we weren't able to observe the weather directly but we could observe our roof dripping. Then, we could have the roof dripping variable point to the random variables that the weather variable. This would make up for the lack of information on the weather variable.
- For simplification, we could assume every variable $X$ is conditionally independent of all other nodes except $X$'s parents.
- Suppose every variable was binary (which means it has exactly 2 possible values) and we had $N$ of them in our net. Normally, the probability table would be $2^N$ rows. But we can do better than that.
- If any node in the graph had $k$ parents at max, then the biggest a probability table would be for that node is $2^{k+1}$ rows. Accounting for all $N$ nodes, we would have $O(N \cdot 2^{k+1})$ rows, which is a lot better.

### Conditional Independence in a Bayes' Net

- We however want to think about conditional independence more formally. In particular, we'll start simple and examine small configurations that will lead to a generalization. There are 3 basic configurations of 3 random variables $X$, $Y$, and $Z$ interacting with each other:
- **Causal Chains:** We have $X \to Y \to Z$. We can figure out that $X$ and $Z$ are not independent. The intuition is how $X$ "influences" $Y$ (by the edge), which in turn influences $Z$. However, $X$ is independent of $Z$ given $Y$. The intuition is if we know $Y$, then the influence chain between $X$ and $Z$ is broken with the value of $Y$ being known.

- **Common Causes:** We have $Y \to X$ and $Y \to Z$. Like Causal Chains, $X$ and $Z$ are not independent, but $X$ is independent of $Z$ given $Y$. The idea here this time is, if we observe $X$, we can most likely surmise something about $Y$, since $Y$ points into $X$. Then, we can use that to guess something about $Z$. So knowing something about $X$ will influence what we think about $Z$. But once we know what $Y$ is, we already have perceptions on $X$ and $Z$. Then, the surprise factor of guessing $Z$ based on $X$ goes out the window.

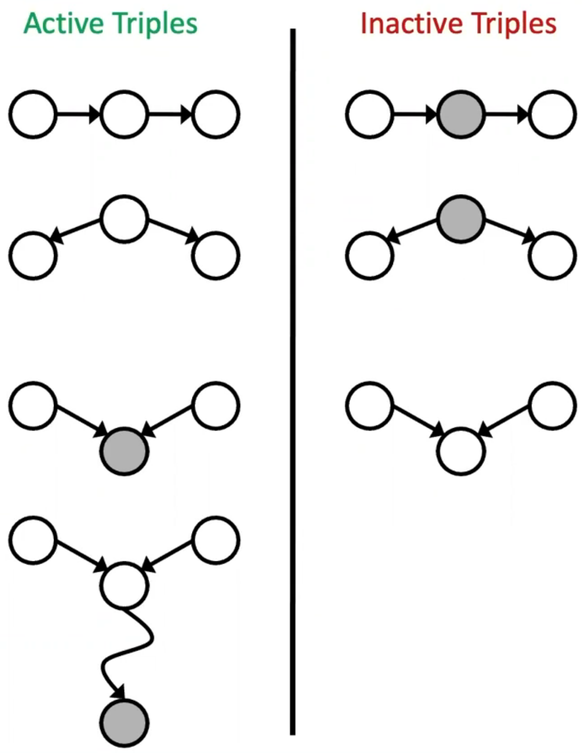
- **Common Effect:** We have $X \to Z$ and $Y \to Z$. This one is the opposite of the last 2. We have $X$ and $Y$ being independent, but $X$ and $Y$ are not independent given $Z$. The idea is, starting out, $X$ and $Y$ don't interact with each other. However, once we see something about $Z$, we go "Alright, which of $X$ or $Y$ caused it?" If we see $X$ didn't cause it, we are more inclined to believe $Y$ caused it (compared to if we see $X$ caused it). Yes, there's the chance both caused it, but observing $Z$ immediately puts $X$ and $Y$ under suspicion on which caused it.

- From these 3 simple interactions, we can extend this to more intricate connections between variables in the net.

## D-Separation

- This is algorithm for determining whether two variables are independent, given some variables we have observed so far.

- Suppose we want to determine if $A$ and $B$ are independent, given what we have so far. We look at *undirected* paths from $A$ to $B$ (so that means we can go backwards on the edges). First, we introduce the notion of active triples and inactive triples:



  Shaded nodes represent observed variables. Also, when determining whether a triple is active or not, we generally only care about the middle node in the triple and not the nodes on the ends.

- For a path to be **active**, every triple on the path has to be active. If there's even one inactive triple on it, it's inactive.

- If we have even one path from $A$ to $B$ that is active, we can't guarantee independence from $A$ to $B$. However, if all such paths are inactive, then $A$ and $B$ are independent.

- To explain the definitions easier, the nodes at the ends of **inactive** triples are independent (remember for example that in the causal chain case, if we observe the middle variable, the other two variables become independent). The opposite is for active variables where we can't conclude the nodes at the ends of the triple are independent.
- So if we have a sequence of nodes where we can't necessarily conclude independence about the ends, that's essentially what an active path is.
- However, in the case of an inactive path, we have independence between 2 variables at some point in the path (due to an inactive triple) and thus creating a rift. When all such paths from $A$ to $B$ have a rift (meaning only inactive paths from $A$ to $B$), then they have to be independent.

## §3.5 Module 11 - Naive Bayes and Perceptrons

### Introduction

- Machine Learning is a topic where we want to train a model with data so that it can understands different classifications. Then, with its knowledge, it should be able to classify future data through pattern recognition.
- Just like when we used features before, we want to use it again here as attributes that will help set classifications apart from each other.
- Then, we can assign features weights and have a weighted function that will predict the classification of an input (given the features it has).

### Naive Bayes

- A simple approach to Machine Learning: we make an important assumption that all features are independent of each other and only dependent on the label (another way of saying "classification").
- So if we have features $F_1$, $F_2$, $\cdots$, $F_n$ and the label is given by the random variable $Y$, we can write

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

- Like in Bayes' Nets, the number of rows in our probability table turns from an exponential in $n$ to a linear number of rows in $n$.
- As you can see, we require the probability information on $P(Y)$ and the conditionals in order to make calculations possible. Where do we get that information? We need some training data of course, where the labels for each piece of data are manually assigned.
- In fact, we have 3 types of data:
  - **Training Data:** As the name implies, the machine uses this data to understand the features that may lead to a particular label for a piece of data. Probabilities can be extracted from the training data.
  - **Held-Out Data:** This is the next phase where hyperparameters (which are essentially more important parameters that effect how the machine goes about learning data) are fine-tuned. The machine may understand probabilities from what it has seen so far, but it may need adjustments and practice before it is implemented.
  - **Test Data:** These are pieces of data that the machine has never seen before and is the phase where the machine has to "go out there and do its job" with the knowledge it has acquired.
- One problem is "Overfitting." The machine may not be able to understand nuance if presented a situation it has never seen before. For example, it may tasked with digit recognition and given data that obviously looks like a 3 but with a stray pixel in an unfamiliar spot. Because of that stray pixel alone, the machine may mistake it for another digit if it had never seen that pixel in that spot for a 3.

- Essentially, assigning 0 probabilities to unseen events is bad because those events could come up in the test set.
- **Laplace Smoothing** - This gets around unseen events have a 0 probability by taking every outcome and increasing the number of times we saw it by 1. We could also increase the number of times we saw each event by $k$ (for some $k$ fixed in advance). This can also be used to smooth conditional probabilities to get around when the denominator is normally 0 (and thus, leaving the conditional probability undefined).
- The issue is Laplace Smoothing doesn't work well in computing $P(X \mid Y)$ if $|X|$ or $|Y|$ is large, since the denominator gets increased substantially from the smoothing.
- **Linear Interpolation** - Another way of getting estimates for conditional probabilities (while getting around 0 probabilities), where we set

  $$P_{LIN}(x \mid y) = \alpha \cdot \hat{P}(x \mid y) + (1 - \alpha) \cdot \hat{P}(x)$$

  for some constant $\alpha$ (and where $\hat{P}$ represents the actually probability observed from the training data). This gets around when $x$ hasn't occurred given $y$, where we can fall back a bit on $x$ occurring in general.
- How do we determine the values of $k$ and/or $\alpha$ though? Remember the Held-Out Data? Yeah, $k$ and $\alpha$ are hyperparameters that can be fine-tuned when working on the Held-Out Data. That way, the machine can be more accurate before we actually use it on Test Data.
- We may also have to add more features if enough errors occur (and particularly, if they occur for the same reason).

## Perceptrons

- A second way of Machine Learning is through Perceptrons. The main learning mechanism will be learning from its mistakes and adjusting as it notices such mistakes.
- Again, we will need weighted features, as well as a threshold. The idea is that the weighted features create a scoring system for an input and if the input scores high enough, it gets given some label. But if it scores too low, it gets given the other label.
- If we have a vector of weights $(w_1, \cdots, w_n)$ assigned to $n$ features and an input $(x_1, \cdots, x_n)$ (with values corresponding to each feature). We can compute the "score" or dot-product as $\sum_{j=1}^{n} w_j x_j$ (noted as $w \cdot x$).
- The "threshold" can be noted as a bias term $b = -$threshold. We can think of this as: we get a "Pass" label if $w \cdot x + b \geq 0$ and a "Fail" label otherwise.
- In our training data, the labels are provided to us and we may make mistakes if our weights are set incorrectly. Feature values for the same input are fixed, so we will have to adjust our weights.
- We want the weights to form the best line/hyperplane that separates the two distinct labels. The bias allows us to shift the boundary line/hyperplane.
- Now, let's actually run through our training data to adjust the weights:
  - Start with $w = 0$ (the 0 vector).
  - Take the next piece of data $x$. Compute $w \cdot x + b$ and we will predict it to have the "Pass" label if the value is $\geq 0$ and the "Fail" label otherwise.
  - Now, actually check the label on $x$, which is given to us, since this is training data. If we were right on our prediction, we don't change anything.
  - Otherwise, if we were wrong on our prediction, we make a shift in $w$. In particular, we add $x$ to $w$ if $x$'s label in actuality was "Pass", but subtract $x$ to $w$ if $x$'s label in actuality was "Fail."
  - This is like Approximate Q-Learning where if something went wrong with some data $x$, we blame the features that were active in it.

- We can extend this idea to multiple labels by having a weight vector with each individual label. Instead of a threshold for determining how we assign labels, we compute $x$'s dot product with each label's weight vector and see which one gives us the highest dot product.
- This time, if we get a label wrong, we decrease the weights on the wrong answer and increase the weights on the right answer.
- If it's actually possible to separate the data linearly (that means we can separate it with a line/hyperplane), our process will eventually converge. If it's not linearly separable (through different types of data being clustered too close together), then it gets tricky.
- One big problem is that if we can't separate the data linearly, the weights could thrash around as we're trying to improve the line. But since we can never actually separate the data, it's a bunch of unstable changes.
- Margin-infused relaxed algorithm - To prevent wild swings of the weights, we can (by a lengthy process) compute a $\tau$ that depends on all the example data we have. Then, instead of adding $\pm x$ to $w$, we add $\pm \tau x$ to $w$, so we can fix the mistake but without such a drastic change to $w$. This also helps the line/hyperplane converge faster.
- Support Vector Machines - A way of further optimizing the separator. At a high level, it takes into account all the data we have and then adjusts the separator so that it more evenly divides the labels. In particular, no data point should be too close to the separator. This makes it easier classify outliers who may end up being close to the separator.
- Kernel Trick - When we add extra dimension to the data or transform it in some way so that it becomes linearly separable.
- Overall, perceptrons don't use probability unlike Bayes Nets. You may have to iterate through the data multiple times, but they are more accurate than Bayes Nets.

## §3.6 Module 12 - Neural Networks

### Introduction

- We take the perceptron idea from the last module and assemble multiple of them into one big unit called a "Neural Network." These are more complicated but are able to achieve more complex and intricate classifications that a single perceptron can't do on its own.
- Many concepts from perceptrons carry over like training the machine on some data, as well as defining features and learning weights.
- However, there are clear differences, particularly in how we classify data and how we correct errors.

### Logistic Regression

- One big difference between perceptrons and neural networks is classifying data. For an individual perceptron, it makes a guess yes or no. But we want to be more specific here: we want the machine to specify the probability that it thinks a piece of data is a particular label.
- Since we're considering probability and we don't want such drastic changes, we use a sigmoid function given by $\sigma(z) = \frac{1}{1+e^{-z}}$. In particular, if we compute $z = w \cdot x + b$ (the dot product plus the bias), we say our desired probability is $\sigma(z)$.
- We can assign a probability threshold like we did with perceptrons, like we need at least 0.5 probability for us to guess the label is True.
- The idea this time with using a probability is that it tells us nuance. For example, if a piece of data is so blatantly one particular label, the machine should output a high probability for that label. Basically, we want our probability estimate to be as close to correct as possible (including nuance on how "blatant" the label is).

## Cross-Entropy Loss

- This is a way of measuring the error (in other words, how far off the machine's output is from the actual labels). Obviously, we want to minimize the error for accuracy, and we need to cook up a function to measure it.
- Suppose for simplicity sake that the only possible labels are 0 and 1. If we have some observed variable(s) $x$ and $y$ is the correct label, we want to maximize

$$p(y \mid x) = \hat{y}^y (1 - \hat{y})^{1-y},$$

  where $\hat{y}$ is our prediction for the label.
- The important observation is when $y = 1$, $p(y \mid x) = \hat{y}$ and when $y = 0$, $p(y \mid x) = 1 - \hat{y}$.
- We want $\hat{y}$ to be as close to 1 as possible when $y = 1$, but we want $1 - \hat{y}$ to be as close to 1 as possible when $y = 0$ (to increase the accuracy on our prediction).
- The way of calculating Cross-Entropy Loss $-\log(p(y \mid x))$. Take my word on it that maximizing $p(y \mid x)$ (as we wish) is the same as minimizing the negative log of it. Since we talked about minimizing our loss earlier, it's easier to convert the quantity into something we want to minimize rather than maximize. We denote $L_{CE}(\hat{y}, y) = -\log(p(y \mid x))$.
- Since we use linear regression for our prediction (which is $\hat{y}$), we can plug in $\hat{y} = \sigma(w \cdot x + b)$, but that will make the general formula more complicated.
- The main idea is the cross-entropy loss will be higher the more inaccurate the prediction was (and vice versa). We can of course take the average of $L_{CE}$ over all the training data in order to measure the loss.

## Stochastic Gradient Descent

- So now we have a way of measuring loss, but now, we want to figure out how to actually minimize it! This will be done via an algorithm called **Stochastic Gradient Descent**, where we figure out what direction to adjust the weights and keep doing so until we reach a minimum.
- The analogy is that when we're skiing, if we want to get to the bottom (a minimum), we follow the slope of the mountain that is pointing downwards until we get stuck at the bottom of a crater.
- You may have remembered from calculus that derivatives are a way of measuring rate of change, which will help in determining which way is "down."
- One thing about perceptrons is that dot products that end up right near the threshold become very sensitive to change: if we nudge the value just a little bit, it may cross the threshold and then completely change the guess. This sudden, discontinuous "jump" in the value is not something that works well with derivatives. *This is why we appreciate the smoothness of the probability function.*
- There is a difference though between a **local minimum** and **global minimum**. A local minimum is where the function attains a minimum in relation to the points around it but is not necessarily the overall minimum of the function. The global minimum *is* defined to be the overall minimum.
- Take my word for this, but the neat thing is, the loss function is convex, which means there's only 1 minimum. That way if we're "skiing down" and get stuck in a crater, we don't have to worry if that's a local minimum that isn't the global minimum.
- So now if we imagine our loss function $f$ (that takes in a single weight $w$ and outputs its loss), the derivative is denoted as $\frac{d}{dw}[f(w)]$. Basically we can start with an initial weight $W_1$ and iterate new weights given by the formula

$$W_{t+1} = W_t - \eta \cdot \frac{df}{dw}[f(W_t)],$$

where $\eta$ is a learning rate (or a step size). Basically, we take a weight $W_t$ and see how the loss function changes at around that point in $f$ (which is why we take the derivative). The derivative points "upward" and since we want go down, we use a minus sign. So to get from $W_t$ to $W_{t+1}$, we nudge in the direction downward with $\eta$ controlling how much in the "downward" direction we go.

- We can decrease $\eta$ as the iterations pass, so that we make smaller, more precise steps to get to the very bottom.
- We can extend this to when we have multiple weights at once with multivariable calculus. We can start with a weight vector $\theta_1$. To iterate, we first compute $\nabla_t$, which will be the vector of partial derivatives of $f$ at $\theta_t$. (Basically, this vector shows how much each individual weight changes in $f$ around $\theta_t$) Then, we have
$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_t.$$
- We could train the weights by doing one random example at a time and update the weights accordingly. Or we can take a batch and compute the gradient over that.
- Once again, we could have overfitting, so we should stop once our accuracy is good enough on the Held-Out Data.
- We could also employ **Regularization**. This is when we penalize weights that are too heavy because an extreme example in our data set can have too drastic of an influence on a weight.
- Of course, if we have multiple classifications, we can use the softmax function instead of the sigmoid function.

## Constructing the Neural Network

- So now we can begin thinking about the actual network works. We'll have layers of neurons (similar to perceptrons) between the input and output. In particular, the original input serves as the input to the first layer. Then, any layer serves as the input to the next layer until we reach the output.
- Again, it is advantageous to use a neuron relying on a smooth function (like the sigmoid function) rather than a perceptron. The smoothness makes it compatible with derivatives.
- The trickiness is that we have to train neural networks not just for one layer but all of them. We can reuse the loss function in Cross-Entropy Loss and our optimization algorithm/strategy in Gradient Descent.
- We'll use a tactic called **Backpropagation** to address errors. In particular, finding an errors involves having to examine connections and work backwards from the source of the fault.
- Also, to generalize Cross-Entropy Loss with multiple labels $y_1, y_2, \cdots, y_C$, we can have
$$L_{CE}(\hat{y}, y) = -\sum_{i=1}^{C} y_i \log \hat{y}_i,$$
so that if $y_j = 1$, it becomes $L_{CE}(\hat{y}, y) = -\log \hat{y}_i$.
- When computing gradients now, we'll have to propagate change through multiple layers that feed into each other. This can be remedied using the **Chain Rule**, which is how we handle derivatives of nested functions (which is a way of saying, a sequence of functions where the output of one function feeds into the next one).
- Here's the high level idea: to compute the gradients and what function feeds into what, we can use a graph. We construct it by seeing what calculations are necessary to get from the start, pass through the layers, and get to the end. Then, we work backwards, using the chain rule along the way, to actually compute the gradients (the effect on the output based on changing a particular input).

## §3.7 Module 13 - Natural Language Processing

### Introduction

- NLP is the field of AI where we try to get the machine to emulate how a human communicates with language.
- Neural Language Models take substantially more training than $n$-grams, but are much more powerful. In particular, they can manage longer word histories and don't rely on smoothing to account for unseen combinations of words.

### Word Vectors

- Words are represented as vectors, which we'll see later helps encode similarity between words.
- Once again, we use the idea from $n$-grams where we have some context as input and then we get the probability distribution for the next word. In particular, softmax will help us get the probability because of its compatibility with multiple inputs.
- We use a corpus (a term for a massive text sample) for training. In particular, we use $k$-grams as training data (with the $(k+1)^{th}$ word being the label). We still use Cross-Entropy Loss for measuring error.
- However, we need to get around the fact we might not observe certain words. This can be done by assigning vectors/probabilities to words that haven't appeared. The vector for that word should be similar to other related words, so that it can be used to fill in a gap when generating a sentence later.
- So these word vectors get fed into the neural network as inputs to predict the next word. But how do we come up with the vectors in the first place?

### The Action

- There is a branch of linguistics called **Lexical Semantics** that studies not only word meaning but also relationships between words (like synonyms, antonyms, connotations, etc).
- Set theory can blend into this branch with the notion of hypernyms and hyponyms. In particular, a hypernym describes a category more broad than another like "fish" being a hypernym of "salmon." A hyponym is the other way around, describing something more specific. So "salmon" or "tuna" is a hyponym of "fish."
- People have manually created massive databases of grouping together relationships between words.
- We are using vectors to represent words because vectors support operations that will help us encode/understand relationships between words.
- Even if words aren't have outright synonyms of each other, they may have related concepts like "price" and "expensive". If two words appear in identical environments, that's a good indicator we can group them together as similar.
- The idea is analogous to how humans learn words: you can use the context and seeing it multiple times to deduce word meanings without having to look it up.
- One application of Machine Learning on the words is by using a bunch of documents for training data. Then, you can create a matrix containing rows as words and columns as document numbers, recording how many times a specific word showed up in a document.
  - By comparing columns, we can see how similar two documents are, which is useful in plagiarism detection (meaning simply copying someone else's document and swapping out words for synonyms will not fool the detector).
  - By comparing rows, we can draw similarities in the meanings of words.

- The vector properties help us encode analogies like "the word $a$ is to $a^*$ like $b$ is to $b^*$" (where we're given $a$, $a^*$, and $b$, but we have to deduce $b^*$). Essentially, the vector $b^* - b$ should be the same as $a^* - a$, so you can set $b^* = b - a + a^*$.
- This leads to applications like grouping comparative words together that describe the same concept (short, shorter, shortest).
- Also, these language models can be trained on older books to study how language evolved over time. For example, the word "literally" has its traditional meaning of "exactly," but has recently been used for extreme exaggeration.
- This model can also be used for applications like showing a word's 2024 definition had a (unexpected) similar meaning to some other word's 1900s definition.

### Issues

- As it turns out, it's not useful to get the distance between word vectors to see how similar they are because magnitudes of the vectors will mess that up. Instead, we want the angle between them (a smaller angle means more similarity).
- Another way to acquire information on words is to create a term by term matrix: terms will be paired up with other terms based on how often they appear near one another. But the problem is how features of language like adjectives and extra phrases can get in the way with distance between words.
- Instead, we can use dependency patterns, which break down a sentence into parts of speech and seeing which describe each other.
- Focusing solely on frequency isn't the best idea because it's too skewed towards the most frequent words like "the". We could mitigate this by dividing the frequency by the number of documents it appeared in to more penalize words that show up way too often. In particular, we can use TF-IDF, which is a metric to help us see if a particular word is uniquely important to a document.