CS50's Introduction to Artificial Intelligence with Python

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

Brian Yu (https://brianyu.me)
brian@cs.harvard.edu

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.reddit.com/user/davidjmalan)
(https://www.threads.net/@davidjmalan) (https://twitter.com/davidjmalan)

Lecture 0

Artificial Intelligence

Artificial Intelligence (AI) covers a range of techniques that appear as sentient behavior by the computer. For example, AI is used to recognize faces in photographs on your social media, beat the World's Champion in chess, and process your speech when you speak to Siri or Alexa on your phone.

In this course, we will explore some of the ideas that make AI possible:

0. **Search**

Finding a solution to a problem, like a navigator app that finds the best route from your origin to the destination, or like playing a game and figuring out the next move.

1. **Knowledge**

Representing information and drawing inferences from it.

2. **Uncertainty**

Dealing with uncertain events using probability.

3. **Optimization**

Finding not only a correct way to solve a problem, but a better—or the best—way to solve it.

4. **Learning**

Improving performance based on access to data and experience. For example, your email is able to distinguish spam from non-spam mail based on past experience.

5. **Neural Networks**

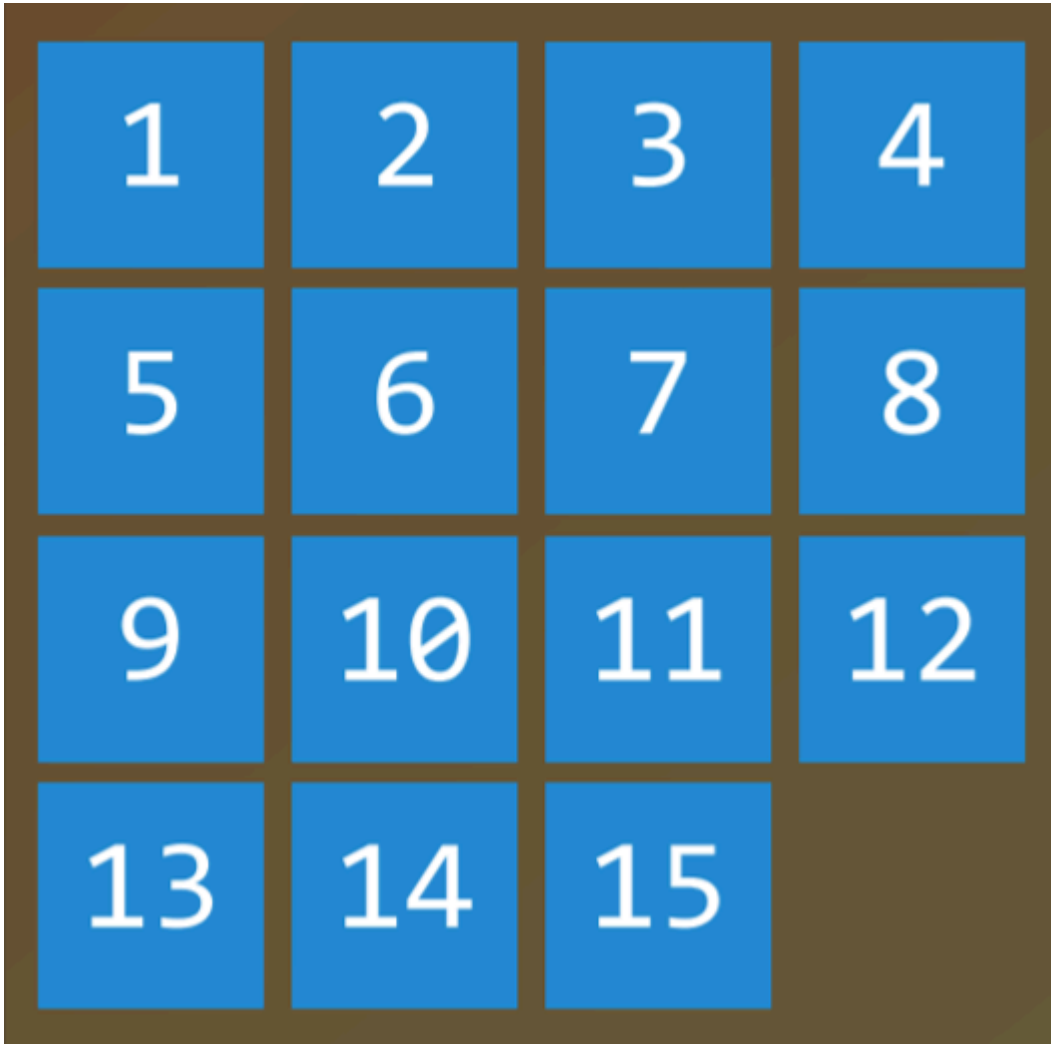
A program structure inspired by the human brain that is able to perform tasks effectively.

6. **Language**

Processing natural language, which is produced and understood by humans.

Search

Search problems involve an agent that is given an initial state and a goal state, and it returns a solution of how to get from the former to the latter. A navigator app uses a typical search process, where the agent (the thinking part of the program) receives as input your current location and your desired destination, and, based on a search algorithm, returns a suggested path. However, there are many other forms of search problems, like puzzles or mazes.



Finding a solution to a 15 puzzle would require the use of a search algorithm.

- **Agent**

  An entity that perceives its environment and acts upon that environment. In a navigator app, for example, the agent would be a representation of a car that needs to decide on which actions to take to arrive at the destination.

- **State**

A configuration of an agent in its environment. For example, in a 15 puzzle (https://en.wikipedia.org/wiki/15_puzzle), a state is any one way that all the numbers are arranged on the board.

- **Initial State**

  The state from which the search algorithm starts. In a navigator app, that would be the current location.

- **Actions**

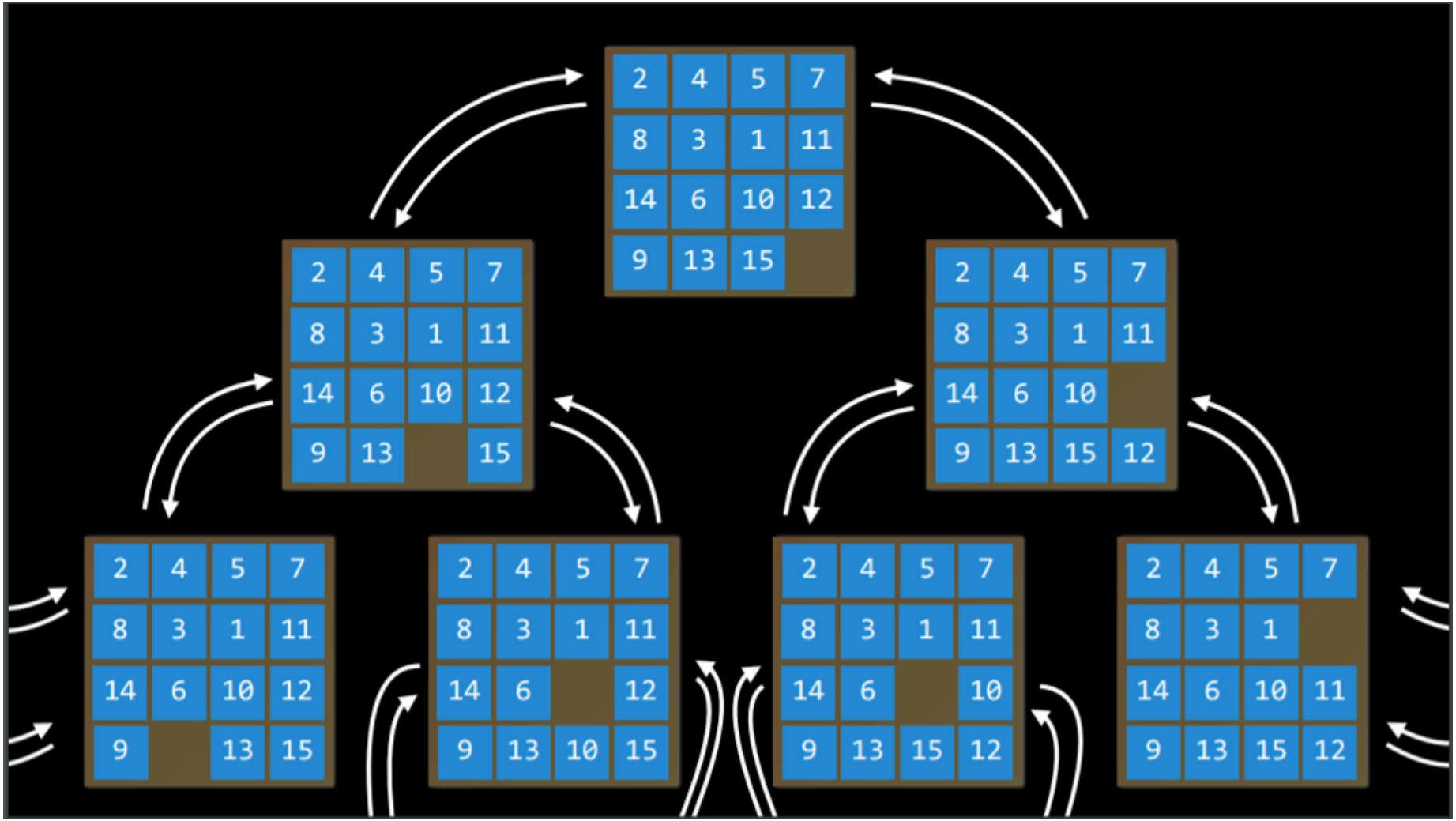
  Choices that can be made in a state. More precisely, actions can be defined as a function. Upon receiving state `s` as input, `Actions(s)` returns as output the set of actions that can be executed in state `s`. For example, in a *15 puzzle*, the actions of a given state are the ways you can slide squares in the current configuration (4 if the empty square is in the middle, 3 if next to a side, 2 if in the corner).

- **Transition Model**

  A description of what state results from performing any applicable action in any state. More precisely, the transition model can be defined as a function. Upon receiving state `s` and action `a` as input, `Results(s, a)` returns the state resulting from performing action `a` in state `s`. For example, given a certain configuration of a *15 puzzle* (state `s`), moving a square in any direction (action `a`) will bring to a new configuration of the puzzle (the new state).

- **State Space**

  The set of all states reachable from the initial state by any sequence of actions. For example, in a 15 puzzle, the state space consists of all the 16!/2 configurations on the board that can be reached from any initial state. The state space can be visualized as a directed graph with states, represented as nodes, and actions, represented as arrows between nodes.



- **Goal Test**

  The condition that determines whether a given state is a goal state. For example, in a navigator app, the goal test would be whether the current location of the agent (the representation of the car) is at the destination. If it is — problem solved. If it's not — we continue searching.

- **Path Cost**

  A numerical cost associated with a given path. For example, a navigator app does not simply bring you to your goal; it does so while minimizing the path cost, finding the fastest way possible for you to get to your goal state.

Solving Search Problems

- **Solution**

  A sequence of actions that leads from the initial state to the goal state.

  - **Optimal Solution**

    A solution that has the lowest path cost among all solutions.

In a search process, data is often stored in a ***node***, a data structure that contains the following data:

- A *state*
- Its *parent node*, through which the current node was generated
- The *action* that was applied to the state of the parent to get to the current node
- The *path cost* from the initial state to this node

Nodes contain information that makes them very useful for the purposes of search algorithms. They contain a *state*, which can be checked using the *goal test* to see if it is the final state. If it is, the node's *path cost* can be compared to other nodes' *path costs*, which allows choosing the *optimal solution*. Once the node is chosen, by virtue of storing the *parent node* and the *action* that led from the *parent* to the current node, it is possible to trace back every step of the way from the *initial state* to this node, and this sequence of actions is the *solution*.

However, *nodes* are simply a data structure — they don't search, they hold information. To actually search, we use the **frontier**, the mechanism that "manages" the *nodes*. The *frontier* starts by containing an initial state and an empty set of explored items, and then repeats the following actions until a solution is reached:

Repeat:

1. If the frontier is empty,
   - *Stop*. There is no solution to the problem.
2. Remove a node from the frontier. This is the node that will be considered.

3. If the node contains the goal state,
    - Return the solution. *Stop*.

    Else,

```
* Expand the node (find all the new nodes that could be reached from this node), and add resulting nodes to the
* Add the current node to the explored set.
```

Depth-First Search

In the previous description of the *frontier*, one thing went unmentioned. At stage 1 in the pseudocode above, which node should be removed? This choice has implications on the quality of the solution and how fast it is achieved. There are multiple ways to go about the question of which nodes should be considered first, two of which can be represented by the data structures of **stack** (in *depth-first* search) and **queue** (in *breadth-first search*; and here is a cute cartoon demonstration (https://www.youtube.com/watch?v=2wM6_PuBIxY) of the difference between the two).

We start with the *depth-first* search (*DFS*) approach.

A *depth-first* search algorithm exhausts each one direction before trying another direction. In these cases, the frontier is managed as a *stack* data structure. The catchphrase you need to remember here is "*last-in first-out*." After nodes are being added to the frontier, the first node to remove and consider is the last one to be added. This results in a search algorithm that goes as deep as possible in the first direction that gets in its way while leaving all other directions for later.

(An example from outside lecture: Take a situation where you are looking for your keys. In a *depth-first* search approach, if you choose to start with searching in your pants, you'd first go through every single pocket, emptying each pocket and going through the contents carefully. You will stop searching in your pants and start searching elsewhere only once you will have completely exhausted the search in every single pocket of your pants.)

- Pros:
    - At best, this algorithm is the fastest. If it "lucks out" and always chooses the right path to the solution (by chance), then *depth-first* search takes the least possible time to get to a solution.

- Cons:
  - It is possible that the found solution is not optimal.
  - At worst, this algorithm will explore every possible path before finding the solution, thus taking the longest possible time before reaching the solution.

Code example:

```
    # Define the function that removes a node from the frontier and returns it.
    def remove(self):
          # Terminate the search if the frontier is empty, because this means that there is no solution.
        if self.empty():
            raise Exception("empty frontier")
        else:
                  # Save the last item in the list (which is the newest node added)
            node = self.frontier[-1]
            # Save all the items on the list besides the last node (i.e. removing the last node)
            self.frontier = self.frontier[:-1]
            return node
```

Breadth-First Search

The opposite of *depth-first* search would be *breadth-first* search (*BFS*).

A *breadth-first* search algorithm will follow multiple directions at the same time, taking one step in each possible direction before taking the second step in each direction. In this case, the frontier is managed as a *queue* data structure. The catchphrase you need to remember here is "*first-in first-out*." In this case, all the new nodes add up in line, and nodes are being considered based on which one was added first (first come first served!). This results in a search algorithm that takes one step in each possible direction before taking a second step in any one direction.

(An example from outside lecture: suppose you are in a situation where you are looking for your keys. In this case, if you start with your pants, you will look in your right pocket. After this, instead of looking at your left pocket, you will take a look in one drawer. Then on the table. And so on, in every location you can think of. Only after you will have exhausted all the locations will you go back to your pants and search in the next pocket.)

- Pros:
  - This algorithm is guaranteed to find the optimal solution.
- Cons:
  - This algorithm is almost guaranteed to take longer than the minimal time to run.
  - At worst, this algorithm takes the longest possible time to run.

Code example:

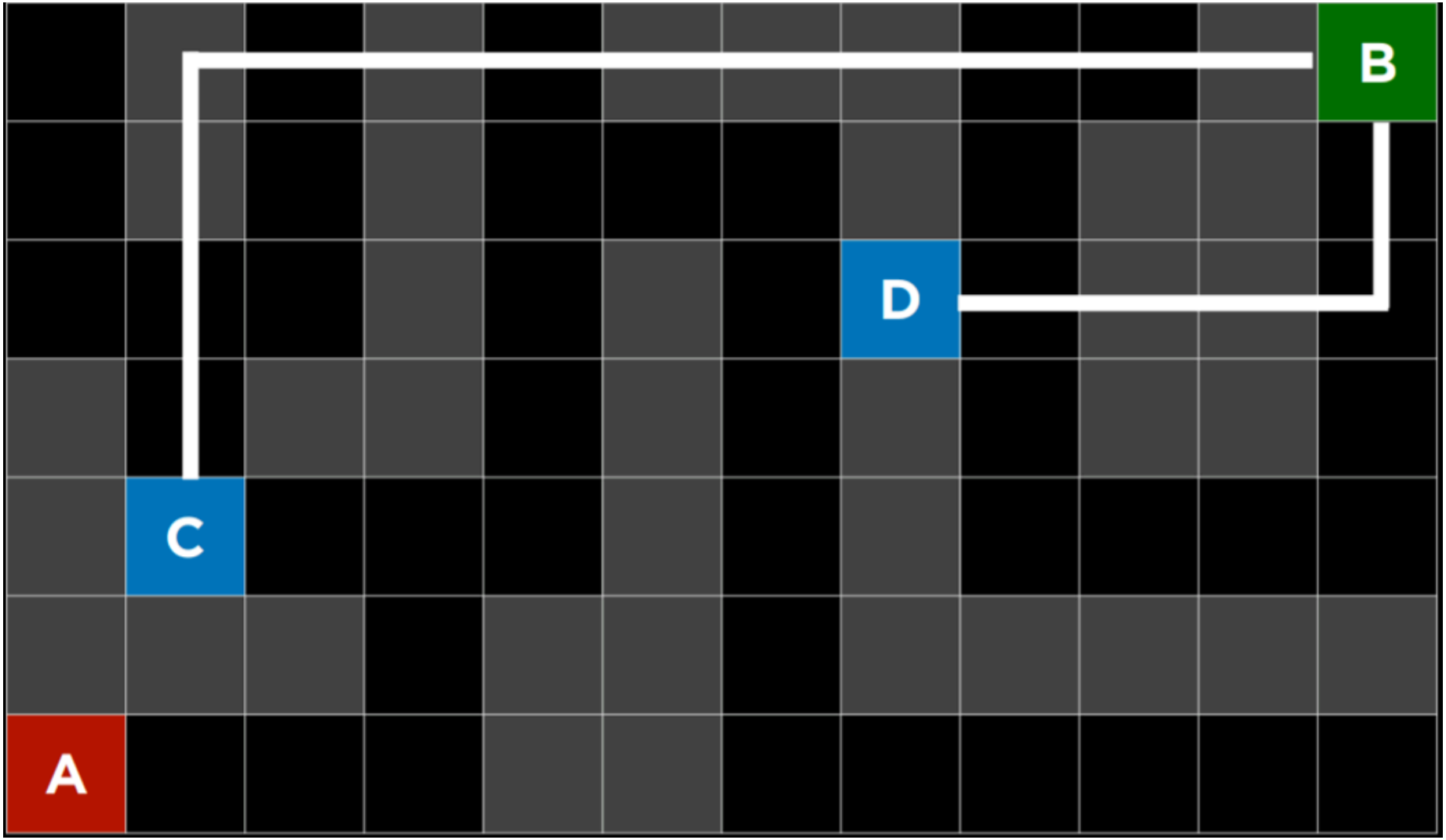
```
    # Define the function that removes a node from the frontier and returns it.
    def remove(self):
          # Terminate the search if the frontier is empty, because this means that there is no solution.
        if self.empty():
            raise Exception("empty frontier")
        else:
            # Save the oldest item on the list (which was the first one to be added)
            node = self.frontier[0]
            # Save all the items on the list besides the first one (i.e. removing the first node)
            self.frontier = self.frontier[1:]
            return node
```

Greedy Best-First Search

Breadth-first and depth-first are both **uninformed** search algorithms. That is, these algorithms do not utilize any knowledge about the problem that they did not acquire through their own exploration. However, most often is the case that some knowledge about the problem is, in fact, available. For example, when a human maze-solver enters a junction, the human can see which way goes in the general direction of the solution and which way does not. AI can do the same. A type of algorithm that considers additional knowledge to try to improve its performance is called an **informed** search algorithm.

Greedy best-first search expands the node that is the closest to the goal, as determined by a **heuristic function** *h(n)*. As its name suggests, the function estimates how close to the goal the next node is, but it can be mistaken. The efficiency of the *greedy best-first* algorithm depends on how good the heuristic function is. For example, in a maze, an algorithm can use a heuristic function that relies on the **Manhattan distance** between the possible nodes and the end of the maze. The *Manhattan distance* ignores walls and counts

how many steps up, down, or to the sides it would take to get from one location to the goal location. This is an easy estimation that can be derived based on the (x, y) coordinates of the current location and the goal location.



Manhattan Distance

However, it is important to emphasize that, as with any heuristic, it can go wrong and lead the algorithm down a slower path than it would have gone otherwise. It is possible that an *uninformed* search algorithm will provide a better solution faster, but it is less likely to do so than an *informed* algorithm.

A* Search

A development of the *greedy best-first* algorithm, *A* search* considers not only *h(n)*, the estimated cost from the current location to the goal, but also *g(n)*, the cost that was accrued until the current location. By combining both these values, the algorithm has a more accurate way of determining the cost of the solution and optimizing its choices on the go. The algorithm keeps track of (*cost of path until now + estimated cost to the goal*), and once it exceeds the estimated cost of some previous option, the algorithm will ditch the current path and go back to the previous option, thus preventing itself from going down a long, inefficient path that *h(n)* erroneously marked as best.

Yet again, since this algorithm, too, relies on a heuristic, it is as good as the heuristic that it employs. It is possible that in some situations it will be less efficient than *greedy best-first* search or even the *uninformed* algorithms. For *A* search* to be optimal, the heuristic function, *h(n)*, should be:

1. *Admissible*, or never *overestimating* the true cost, and
2. *Consistent*, which means that the estimated path cost to the goal of a new node in addition to the cost of transitioning to it from the previous node is greater or equal to the estimated path cost to the goal of the previous node. To put it in an equation form, *h(n)* is consistent if for every node *n* and successor node *n'* with step cost *c*, $h(n) \leq h(n') + c$.

Adversarial Search

Whereas, previously, we have discussed algorithms that need to find an answer to a question, in **adversarial search** the algorithm faces an opponent that tries to achieve the opposite goal. Often, AI that uses adversarial search is encountered in games, such as tic tac toe.

Minimax

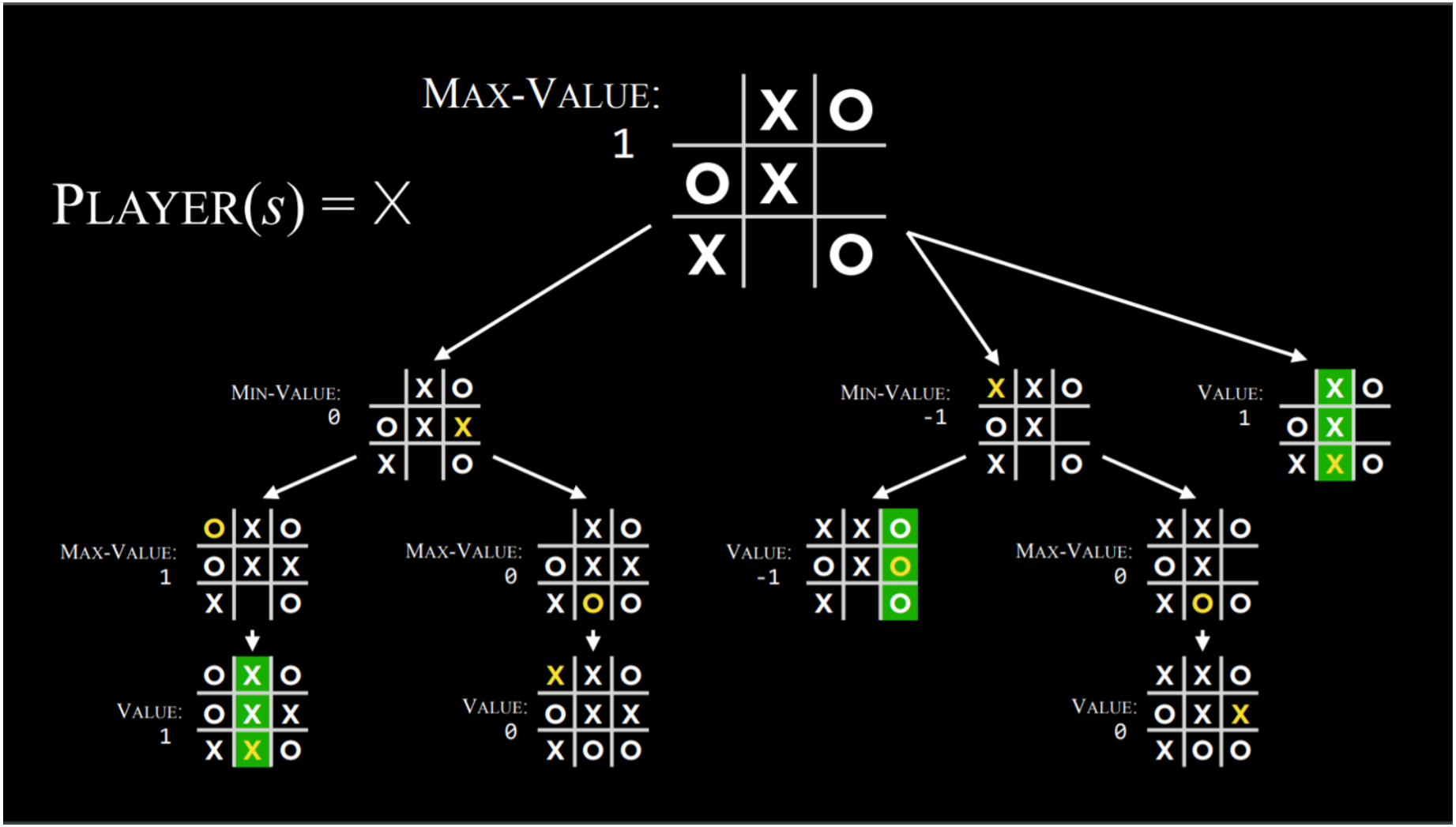
A type of algorithm in adversarial search, **Minimax** represents winning conditions as (-1) for one side and (+1) for the other side. Further actions will be driven by these conditions, with the minimizing side trying to get the lowest score, and the maximizer trying to get the highest score.

Representing a Tic-Tac-Toe AI:

- S_0: Initial state (in our case, an empty 3X3 board)
- *Players(s)*: a function that, given a state *s*, returns which player's turn it is (X or O).
- *Actions(s)*: a function that, given a state *s*, return all the legal moves in this state (what spots are free on the board).
- *Result(s, a)*: a function that, given a state *s* and action *a*, returns a new state. This is the board that resulted from performing the action *a* on state *s* (making a move in the game).
- *Terminal(s)*: a function that, given a state *s*, checks whether this is the last step in the game, i.e. if someone won or there is a tie. Returns *True* if the game has ended, *False* otherwise.
- *Utility(s)*: a function that, given a terminal state *s*, returns the utility value of the state: -1, 0, or 1.

How the algorithm works:

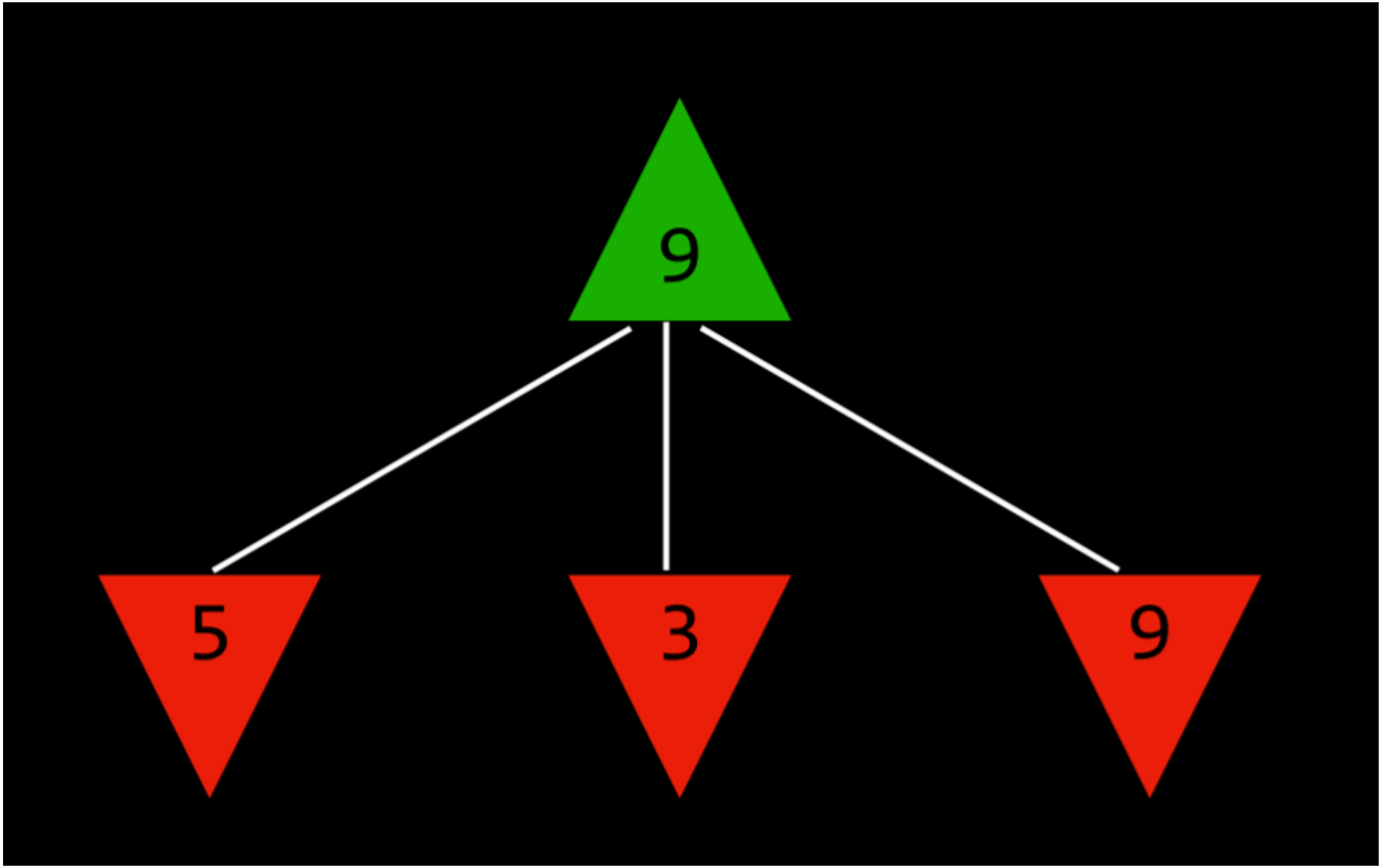
Recursively, the algorithm simulates all possible games that can take place beginning at the current state and until a terminal state is reached. Each terminal state is valued as either (-1), 0, or (+1).



Minimax Algorithm in Tic Tac Toe

Knowing based on the state whose turn it is, the algorithm can know whether the current player, when playing optimally, will pick the action that leads to a state with a lower or a higher value. This way, alternating between minimizing and maximizing, the algorithm creates values for the state that would result from each possible action. To give a more concrete example, we can imagine that the maximizing player asks at every turn: "if I take this action, a new state will result. If the minimizing player plays optimally, what action can that player take to bring to the lowest value?" However, to answer this question, the maximizing player has to ask: "To know what the minimizing player will do, I need to simulate the same process in the minimizer's mind: the minimizing player will try to ask: 'if I

take this action, what action can the maximizing player take to bring to the highest value?'" This is a recursive process, and it could be hard to wrap your head around it; looking at the pseudo code below can help. Eventually, through this recursive reasoning process, the maximizing player generates values for each state that could result from all the possible actions at the current state. After having these values, the maximizing player chooses the highest one.



The Maximizer Considers the Possible Values of Future States.

To put it in pseudocode, the Minimax algorithm works the following way:

- Given a state *s*
  - The maximizing player picks action *a* in *Actions(s)* that produces the highest value of *Min-Value(Result(s, a))*.
  - The minimizing player picks action *a* in *Actions(s)* that produces the lowest value of *Max-Value(Result(s, a))*.
- Function *Max-Value(state)*
  - *v* = -∞
  - if *Terminal(state)*:

    return *Utility(state)*
  - for *action* in *Actions(state)*:

    v = Max(v, Min-Value(Result(state, action)))

    return *v*
- Function *Min-Value(state)*:
  - *v* = ∞
  - if *Terminal(state)*:

    return *Utility(state)*
  - for *action* in *Actions(state)*:

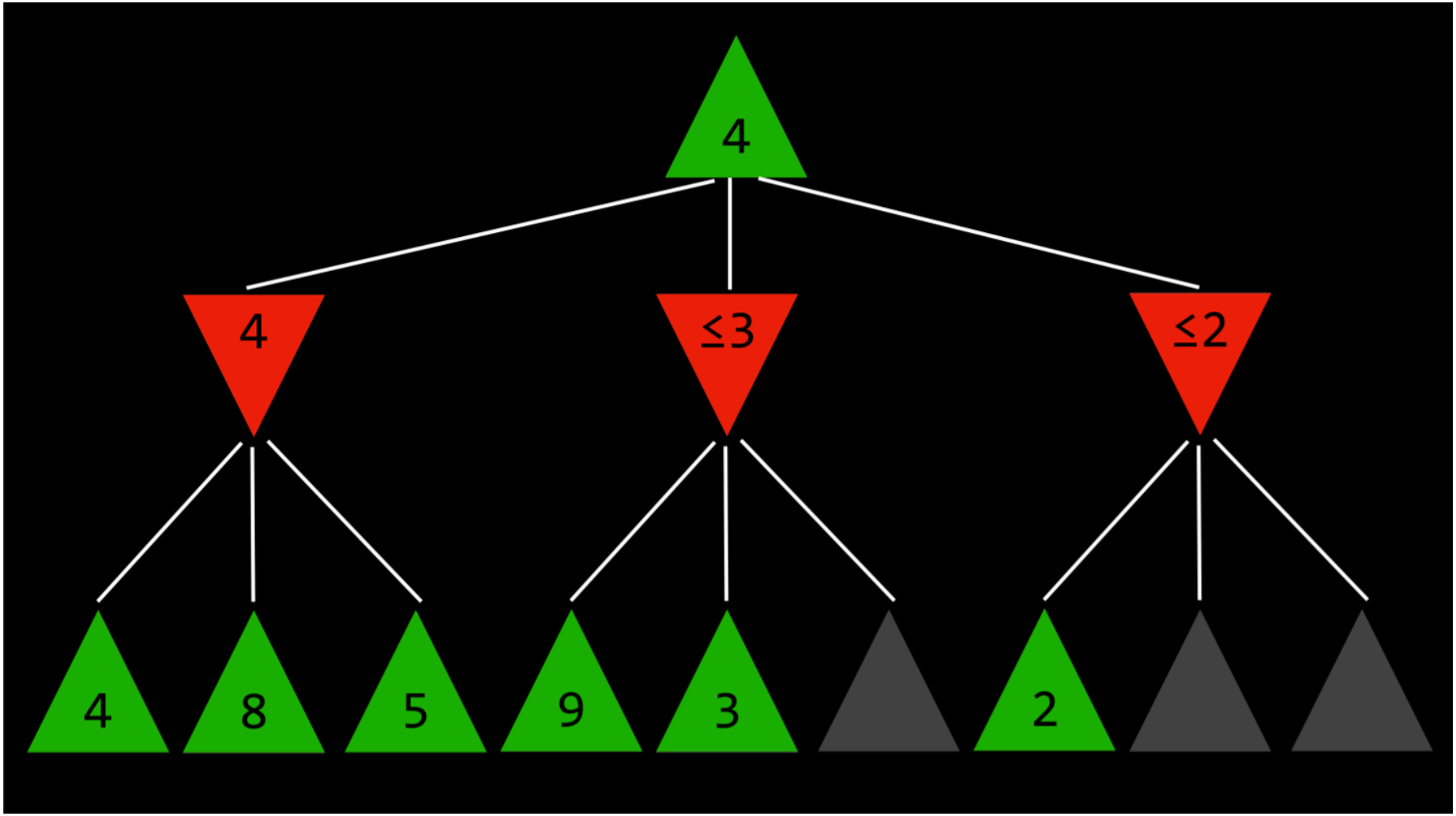
    v = Min(v, Max-Value(Result(state, action)))

    return *v*

Alpha-Beta Pruning

A way to optimize *Minimax*, **Alpha-Beta Pruning** skips some of the recursive computations that are decidedly unfavorable. After establishing the value of one action, if there is initial evidence that the following action can bring the opponent to get to a better score than the already established action, there is no need to further investigate this action because it will decidedly be less favorable than the previously established one.

This is most easily shown with an example: a maximizing player knows that, at the next step, the minimizing player will try to achieve the lowest score. Suppose the maximizing player has three possible actions, and the first one is valued at 4. Then the player starts generating the value for the next action. To do this, the player generates the values of the minimizer's actions if the current player makes this action, knowing that the minimizer will choose the lowest one. However, before finishing the computation for all the possible actions of the minimizer, the player sees that one of the options has a value of three. This means that there is no reason to keep on exploring the other possible actions for the minimizing player. The value of the not-yet-valued action doesn't matter, be it 10 or (-10). If the value is 10, the minimizer will choose the lowest option, 3, which is already worse than the preestablished 4. If the not-yet-valued action would turn out to be (-10), the minimizer will this option, (-10), which is even more unfavorable to the maximizer. Therefore, computing additional possible actions for the minimizer at this point is irrelevant to the maximizer, because the maximizing player already has an unequivocally better choice whose value is 4.



Depth-Limited Minimax

There is a total of 255,168 possible Tic Tac Toe games, and 10^{29000} possible games in Chess. The minimax algorithm, as presented so far, requires generating all hypothetical games from a certain point to the terminal condition. While computing all the Tic-Tac-Toe games doesn't pose a challenge for a modern computer, doing so with chess is currently impossible.

Depth-limited Minimax considers only a pre-defined number of moves before it stops, without ever getting to a terminal state. However, this doesn't allow for getting a precise value for each action, since the end of the hypothetical games has not been reached.

To deal with this problem, *Depth-limited Minimax* relies on an **evaluation function** that estimates the expected utility of the game from a given state, or, in other words, assigns values to states. For example, in a chess game, a utility function would take as input a current configuration of the board, try to assess its expected utility (based on what pieces each player has and their locations on the board), and then return a positive or a negative value that represents how favorable the board is for one player versus the other. These values can be used to decide on the right action, and the better the evaluation function, the better the Minimax algorithm that relies on it.

CS50's Introduction to Artificial Intelligence with Python

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

Brian Yu (https://brianyu.me)
brian@cs.harvard.edu

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://www.reddit.com/user/davidjmalan) (https://www.threads.net/@davidjmalan)
(https://twitter.com/davidjmalan)

Lecture 1

Knowledge

Humans reason based on existing knowledge and draw conclusions. The concept of representing knowledge and drawing conclusions from it is also used in AI, and in this lecture we will explore how we can achieve this behavior.

Knowledge-Based Agents

These are agents that reason by operating on internal representations of knowledge.

What does "reasoning based on knowledge to draw a conclusion" mean?

Let's start answering this with a Harry Potter example. Consider the following sentences:

1. If it didn't rain, Harry visited Hagrid today.
2. Harry visited Hagrid or Dumbledore today, but not both.
3. Harry visited Dumbledore today.

Based on these three sentences, we can answer the question "did it rain today?", even though none of the individual sentences tells us anything about whether it is raining today. Here is how we can

go about it: looking at sentence 3, we know that Harry visited Dumbledore. Looking at sentence 2, we know that Harry visited either Dumbledore or Hagrid, and thus we can conclude

4. Harry did not visit Hagrid.

Now, looking at sentence 1, we understand that if it didn't rain, Harry would have visited Hagrid. However, knowing sentence 4, we know that this is not the case. Therefore, we can conclude

5. It rained today.

To come to this conclusion, we used logic, and today's lecture explores how AI can use logic to reach to new conclusions based on existing information.

Sentence

A sentence is an assertion about the world in a knowledge representation language. A sentence is how AI stores knowledge and uses it to infer new information.

Propositional Logic

Propositional logic is based on propositions, statements about the world that can be either true or false, as in sentences 1-5 above.

Propositional Symbols

Propositional symbols are most often letters (P, Q, R) that are used to represent a proposition.

Logical Connectives

Logical connectives are logical symbols that connect propositional symbols in order to reason in a more complex way about the world.

- **Not (¬)** inverses the truth value of the proposition. So, for example, if P: "It is raining," then ¬P: "It is not raining".

  Truth tables are used to compare all possible truth assignments to propositions. This tool will help us better understand the truth values of propositions when connected with different logical connectives. For example, below is our first truth table:

P	¬P
false	true
true	false

- **And (∧)** connects two different propositions. When these two proposition, P and Q, are connected by ∧, the resulting proposition P ∧ Q is true only in the case that both P and Q are true.

P	**Q**	**P ∧ Q**
false	false	false
false	true	false
true	false	false
true	true	true

- **Or (∨)** is true as as long as either of its arguments is true. This means that for P ∨ Q to be true, at least one of P or Q has to be true.

P	**Q**	**P ∨ Q**
false	false	false
false	true	true
true	false	true
true	true	true

It is worthwhile to mention that there are two types of Or: an inclusive Or and an exclusive Or. In an exclusive Or, P ∨ Q is false if P ∧ Q is true. That is, an exclusive Or requires only one of its arguments to be true and not both. An inclusive Or is true if any of P, Q, or P ∧ Q is true. In the case of Or (∨), the intention is an inclusive Or.

> **A couple of side notes not mentioned in lecture**:
>
> - Sometimes an example helps understand inclusive versus exclusive Or. Inclusive Or: "in order to eat dessert, you have to clean your room or mow the lawn." In this case, if you do both chores, you will still get the cookies. Exclusive Or: "For dessert, you can have either cookies or ice cream." In this case, you can't have both.
> - If you are curious, the exclusive Or is often shortened to XOR and a common symbol for it is ⊕).

- **Implication (→)** represents a structure of "if P then Q." For example, if P: "It is raining" and Q: "I'm indoors", then P → Q means "If it is raining, then I'm indoors." In the case of P implies Q (P

→ Q), P is called the **antecedent** and Q is called the *consequent*.

When the **antecedent** is true, the whole implication is true in the case that the **consequent** is true (that makes sense: if it is raining and I'm indoors, then the sentence "if it is raining, then I'm indoors" is true). When the **antecedent** is true, the implication is false if the **consequent** is false (if I'm outside while it is raining, then the sentence "If it is raining, then I'm indoors" is false). However, when the **antecedent** is false, the implication is always true, regardless of the **consequent**. This can sometimes be a confusing concept. Logically, we can't learn anything from an implication (P → Q) if the **antecedent** (P) is false. Looking at our example, if it is not raining, the implication doesn't say anything about whether I'm indoors or not. I could be an indoors type and never walk outside, even when it is not raining, or I could be an outdoors type and be outside all the time when it is not raining. When the antecedent is false, we say that the implication is *trivially* true.

P	**Q**	**P → Q**
false	false	true
false	true	true
true	false	false
true	true	true

- **Biconditional (↔)** is an implication that goes both directions. You can read it as "if and only if." P ↔ Q is the same as P → Q and Q → P taken together. For example, if P: "It is raining." and Q: "I'm indoors," then P ↔ Q means that "If it is raining, then I'm indoors," and "if I'm indoors, then it is raining." This means that we can infer more than we could with a simple implication. If P is false, then Q is also false; if it is not raining, we know that I'm also not indoors.

P	**Q**	**P ↔ Q**
false	false	true
false	true	false
true	false	false
true	true	true

Model

The model is an assignment of a truth value to every proposition. To reiterate, propositions are statements about the world that can be either true or false. However, knowledge about the world is represented in the truth values of these propositions. The model is the truth-value assignment that provides information about the world.

For example, if P: "It is raining." and Q: "It is Tuesday.", a model could be the following truth-value assignment: {P = True, Q = False}. This model means that it is raining, but it is not Tuesday. However, there are more possible models in this situation (for example, {P = True, Q = True}, where it is both raining an a Tuesday). In fact, the number of possible models is 2 to the power of the number of propositions. In this case, we had 2 propositions, so 2^2=4 possible models.

Knowledge Base (KB)

The knowledge base is a set of sentences known by a knowledge-based agent. This is knowledge that the AI is provided about the world in the form of propositional logic sentences that can be used to make additional inferences about the world.

Entailment ($\models$)

If $\alpha \models \beta$ (α entails β), then in any world where α is true, β is true, too.

For example, if α: "It is a Tuesday in January" and β: "It is January," then we know that $\alpha \models \beta$. If it is true that it is a Tuesday in January, we also know that it is January. Entailment is different from implication. Implication is a logical connective between two propositions. Entailment, on the other hand, is a relation that means that if all the information in α is true, then all the information in β is true.

Inference

Inference is the process of deriving new sentences from old ones.

For instance, in the Harry Potter example earlier, sentences 4 and 5 were inferred from sentences 1, 2, and 3.

There are multiple ways to infer new knowledge based on existing knowledge. First, we will consider the **Model Checking** algorithm.

- To determine if KB $\models \alpha$ (in other words, answering the question: "can we conclude that α is true based on our knowledge base")
  - Enumerate all possible models.
  - If in every model where KB is true, α is true as well, then KB entails α (KB $\models \alpha$).

Consider the following example:

P: It is a Tuesday. Q: It is raining. R: Harry will go for a run. KB: $(P \wedge \neg Q) \rightarrow R$ (in words, P and not Q imply R) P (P is true) ¬Q (Q is false) Query: R (We want to know whether R is true or false; Does KB $\models$ R?)

To answer the query using the Model Checking algorithm, we enumerate all possible models.

P	**Q**	**R**	**KB**
false	false	false	
false	false	true	
false	true	false	
false	true	true	
true	false	false	
true	false	true	
true	true	false	
true	true	true	

Then, we go through every model and check whether it is true given our Knowledge Base.

First, in our KB, we know that P is true. Thus, we can say that the KB is false in all models where P is not true.

P	**Q**	**R**	**KB**
false	false	false	false
false	false	true	false
false	true	false	false
false	true	true	false
true	false	false	
true	false	true	
true	true	false	

P	Q	R	KB
true	true	true	

Next, similarly, in our KB, we know that Q is false. Thus, we can say that the KB is false in all models where Q is true.

P	Q	R	KB
false	false	false	false
false	false	true	false
false	true	false	false
false	true	true	false
true	false	false	
true	false	true	
true	true	false	false
true	true	true	false

Finally, we are left with two models. In both, P is true and Q is false. In one model R is true and in the other R is false. Due to $(P \land \neg Q) \rightarrow R$ being in our KB, we know that in the case where P is true and Q is false, R must be true. Thus, we say that our KB is false for the model where R is false, and true for the model where R is true.

P	Q	R	KB
false	false	false	false
false	false	true	false
false	true	false	false
false	true	true	false
true	false	false	false
true	false	true	true

P	Q	R	KB
true	true	false	false
true	true	true	false

Looking at this table, there is only one model where our knowledge base is true. In this model, we see that R is also true. By our definition of entailment, if R is true in all models where the KB is true, then KB ⊨ R.

Next, let's look at how knowledge and logic can be represented as code.

```
from logic import *

# Create new classes, each having a name, or a symbol, representing each proposition.
rain = Symbol("rain")  # It is raining.
hagrid = Symbol("hagrid")  # Harry visited Hagrid
dumbledore = Symbol("dumbledore")  # Harry visited Dumbledore

# Save sentences into the KB
knowledge = And(  # Starting from the "And" logical connective, becasue each propositi

    Implication(Not(rain), hagrid),  # ¬(It is raining) → (Harry visited Hagrid)

    Or(hagrid, dumbledore),  # (Harry visited Hagrid) ∨ (Harry visited Dumbledore).

    Not(And(hagrid, dumbledore)),  # ¬(Harry visited Hagrid ∧ Harry visited Dumbledore

    dumbledore  # Harry visited Dumbledore. Note that while previous propositions cont
    )
```

To run the Model Checking algorithm, the following information is needed:

- Knowledge Base, which will be used to draw inferences
- A query, or the proposition that we are interested in whether it is entailed by the KB
- Symbols, a list of all the symbols (or atomic propositions) used (in our case, these are `rain`, `hagrid`, and `dumbledore`)
- Model, an assignment of truth and false values to symbols

The model checking algorithm looks as follows:

```
def check_all(knowledge, query, symbols, model):

    # If model has an assignment for each symbol
    # (The logic below might be a little confusing: we start with a list of symbols. T
    if not symbols:
```

```
        # If knowledge base is true in model, then query must also be true
        if knowledge.evaluate(model):
            return query.evaluate(model)
        return True
    else:

        # Choose one of the remaining unused symbols
        remaining = symbols.copy()
        p = remaining.pop()

        # Create a model where the symbol is true
        model_true = model.copy()
        model_true[p] = True

        # Create a model where the symbol is false
        model_false = model.copy()
        model_false[p] = False

        # Ensure entailment holds in both models
        return(check_all(knowledge, query, remaining, model_true) and check_all(knowle
```

Note that we are interested only in the models where the KB is true. If the KB is false, then the conditions that we know to be true are not occurring in these models, making them irrelevant to our case.

> **An example from outside lecture**: Let P: Harry plays seeker, Q: Oliver plays keeper, R: Gryffindor wins. Our KB specifies that P Q (P ∧ Q) → R. In other words, we know that P is true, i.e. Harry plays seeker, and that Q is true, i.e. Oliver plays keeper, and that if both P and Q are true, then R is true, too, meaning that Gryffindor wins the match. Now imagine a model where Harry played beater instead of seeker (thus, Harry did not play seeker, ¬P). Well, in this case, we don't care whether Gryffindor won (whether R is true or not), because we have the information in our KB that Harry played seeker and not beater. We are only interested in the models where, as in our case, P and Q are true.)

Further, the way the `check_all` function works is recursive. That is, it picks one symbol, creates two models, in one of which the symbol is true and in the other the symbol is false, and then calls itself again, now with two models that differ by the truth assignment of this symbol. The function will keep doing so until all symbols will have been assigned truth-values in the models, leaving the list `symbols` empty. Once it is empty (as identified by the line `if not symbols`), in each instance of the function (wherein each instance holds a different model), the function checks whether the KB is true given the model. If the KB is true in this model, the function checks whether the query is true, as described earlier.

Knowledge Engineering

Knowledge engineering is the process of figuring out how to represent propositions and logic in AI.

Let's practice knowledge engineering using the game Clue.

In the game, a murder was committed by a *person*, using a *tool* in a *location*. People, tools, and locations are represented by cards. One card of each category is picked at random and put in an envelope, and it is up to the participants to uncover whodunnit. Participants do so by uncovering cards and deducing from these clues what must be in the envelope. We will use the Model Checking algorithm from before to uncover the mystery. In our model, we mark as `True` items that we know are related to the murder and `False` otherwise.

For our purposes, suppose we have three people: Mustard, Plum, and Scarlet, three tools: knife, revolver, and wrench, and three locations: ballroom, kitchen, and library.

We can start creating our knowledge base by adding the rules of the game. We know for certain that one person is the murderer, that one tool was used, and that the murder happened in one location. This can be represented in propositional logic the following way:

(Mustard ∨ Plum ∨ Scarlet)

(knife ∨ revolver ∨ wrench)

(ballroom ∨ kitchen ∨ library)

The game starts with each player seeing one person, one tool, and one location, thus knowing that they are not related to the murder. Players do not share the information that the saw in these cards. Suppose our player gets the cards of Mustard, kitchen, and revolver. Thus, we know that these are not related to the murder and we can add to our KB

¬(Mustard)

¬(kitchen)

¬(revolver)

In other situations in the game, one can make a guess, suggesting one combination of person, tool and location. Suppose that the guess is that Scarlet used a wrench to commit the crime in the library. If this guess is wrong, then the following can be deduced and added to the KB:

(¬Scarlet ∨ ¬library ∨ ¬wrench)

Now, suppose someone shows us the Plum card. Thus, we can add

¬(Plum)

to our KB.

At this point, we can conclude that the murderer is Scarlet, since it has to be one of Mustard, Plum, and Scarlet, and we have evidence that the first two are not it.

Adding just one more piece of knowledge, for example, that it is not the ballroom, can give us more information. First, we update our KB

¬(ballroom)

And now, using multiple previous pieces of data, we can deduce that Scarlet committed the murder with a knife in the library. We can deduce that it's the library because it has to be either the ballroom, the kitchen, or the library, and the first two were proven to not be the locations. However, when someone guessed Scarlet, library, wrench, the guess was false. Thus, at least one of the elements in this statement has to be false. Since we know both Scarlet and library to be true, we know that the wrench is the false part here. Since one of the three instruments has to be true, and it's not the wrench nor the revolver, we can conclude that it is the knife.

Here is how the information would be added to the knowledge base in Python:

```
# Add the clues to the KB
knowledge = And(

    # Start with the game conditions: one item in each of the three categories has to
    Or(mustard, plum, scarlet),
    Or(ballroom, kitchen, library),
    Or(knife, revolver, wrench),

    # Add the information from the three initial cards we saw
    Not(mustard),
    Not(kitchen),
    Not(revolver),

    # Add the guess someone made that it is Scarlet, who used a wrench in the library
    Or(Not(scarlet), Not(library), Not(wrench)),

    # Add the cards that we were exposed to
    Not(plum),
    Not(ballroom)
)
```

We can look at other logic puzzles as well. Consider the following example: four different people, Gilderoy, Pomona, Minerva, and Horace, are assigned to four different houses, Gryffindor, Hufflepuff, Ravenclaw, and Slytherin. There is exactly one person in each house. Representing the puzzle's conditions in propositional logic is quite cumbersome. First, each of the possible assignments will have to be a proposition in itself: MinervaGryffindor, MinervaHufflepuff, MinervaRavenclaw, MinervaSlytherin, PomonaGryffindor… Second, to represent that each person belongs to a house, an Or statement is required with all the possible house assignments per person

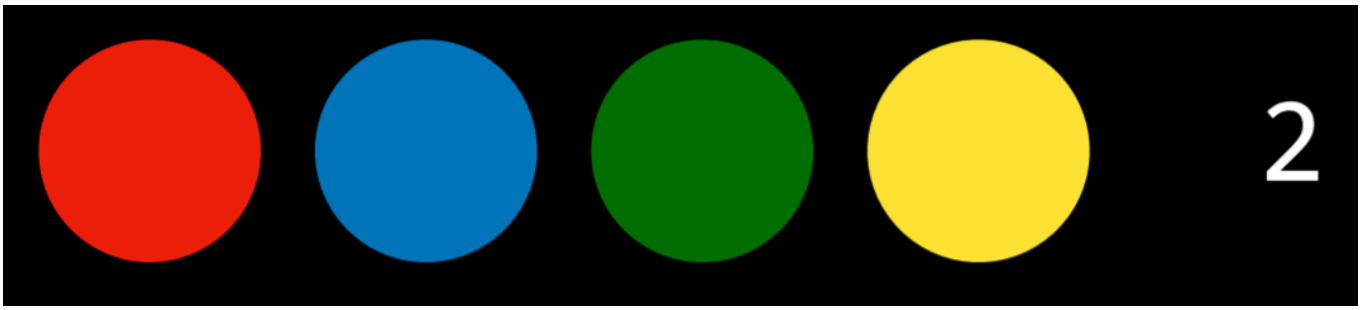
(MinervaGryffindor ∨ MinervaHufflepuff ∨ MinervaRavenclaw ∨ MinervaSlytherin), repeat for every person.

Then, to encode that if one person is assigned to one house, they are not assigned to the other houses, we will write

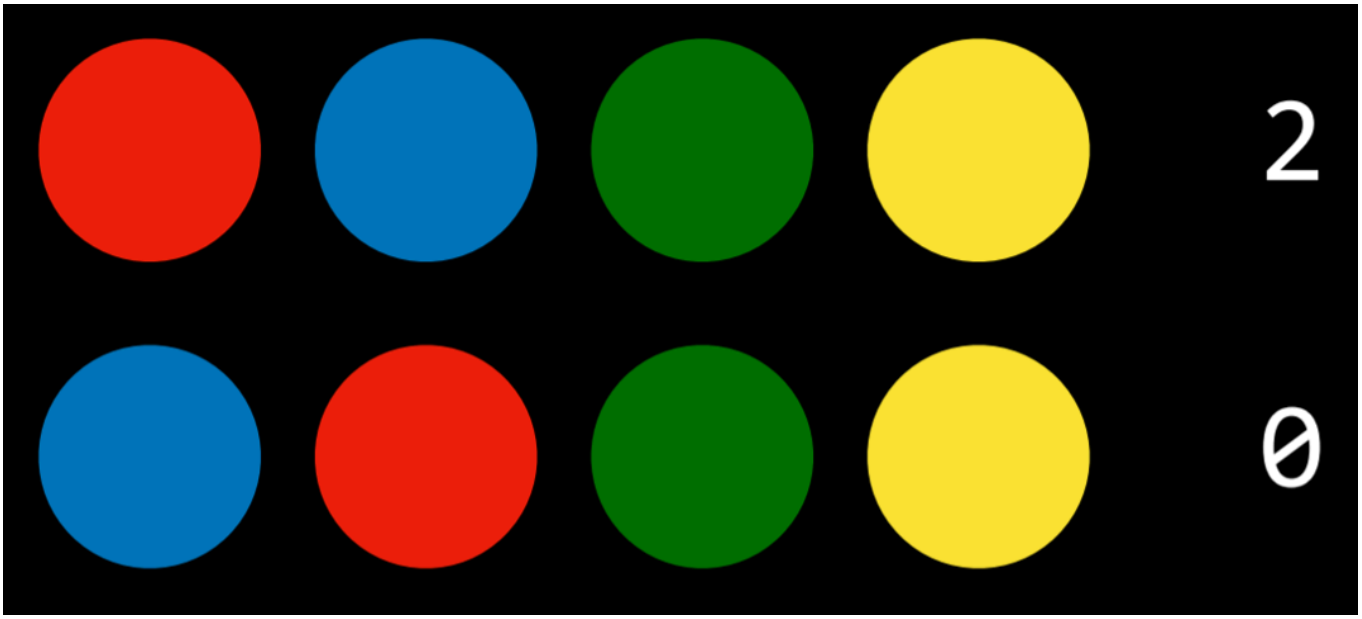
(MinervaGryffindor ⟶ ¬MinervaHufflepuff) ∧ (MinervaGryffindor ⟶ ¬MinervaRavenclaw) ∧ (MinervaGryffindor ⟶ ¬MinervaSlytherin) ∧ (MinervaHufflepuff ⟶ ¬MinervaGryffindor)…

and so on for all houses and all people. A solution to this inefficiency is offered in the section on first order logic. However, this type of riddle can still be solved with either type of logic, given enough cues.

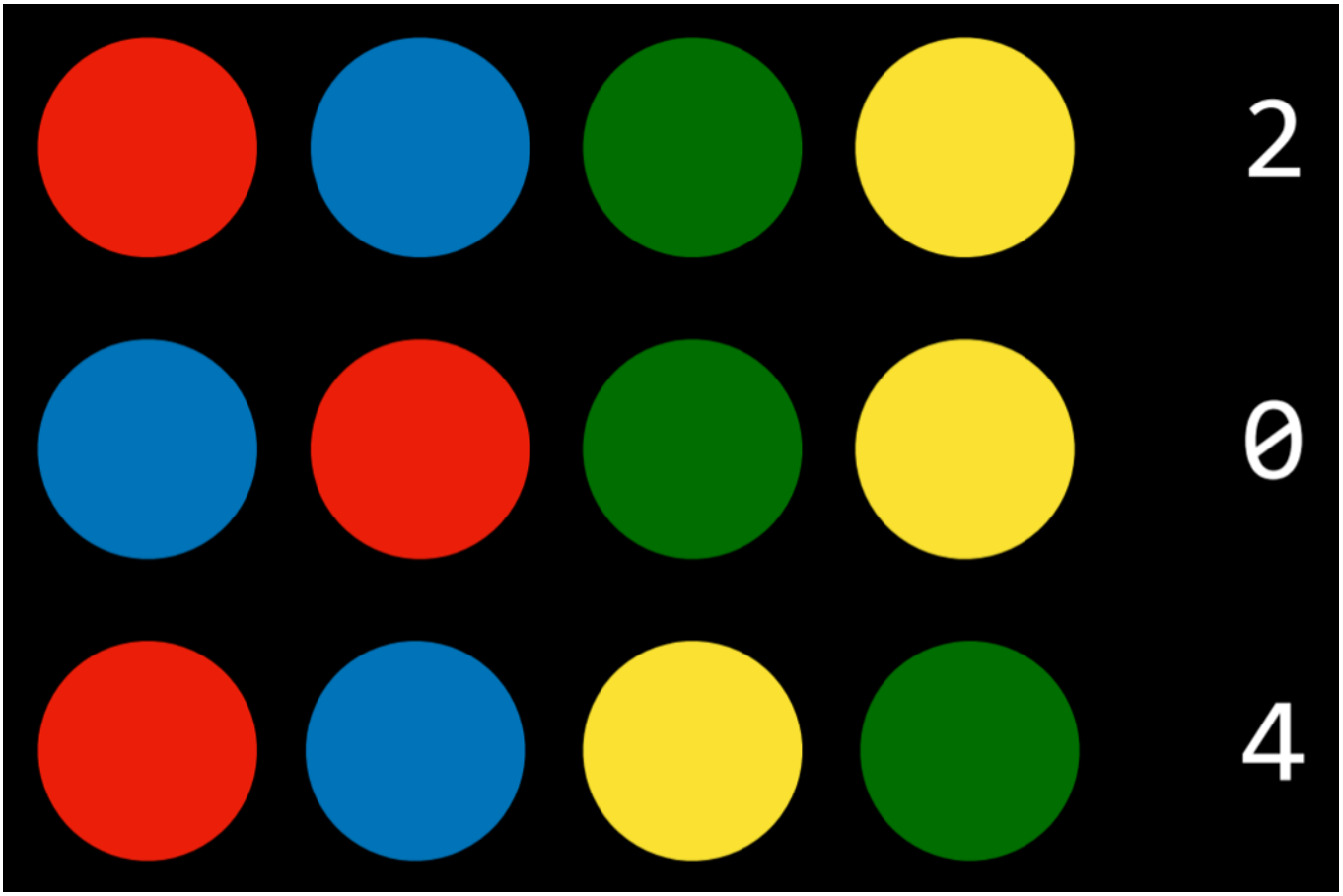
Another type of puzzle that can be solved using propositional logic is a Mastermind game. In this game, player one arranges colors in a certain order, and then player two has to guess this order. Each turn, player two makes a guess, and player one gives back a number, indicating how many colors player two got right. Let's simulate a game with four colors. Suppose player two suggests the following ordering:



Player one answers "two." Thus we know that some two of the colors are in the correct position, and the other two are in the wrong place. Based on this information, player two tries to switch the locations of two colors.



Now player one answers “zero.” Thus, player two knows that the switched colors were in the right location initially, which means the untouched two colors were in the wrong location. Player two switches them.



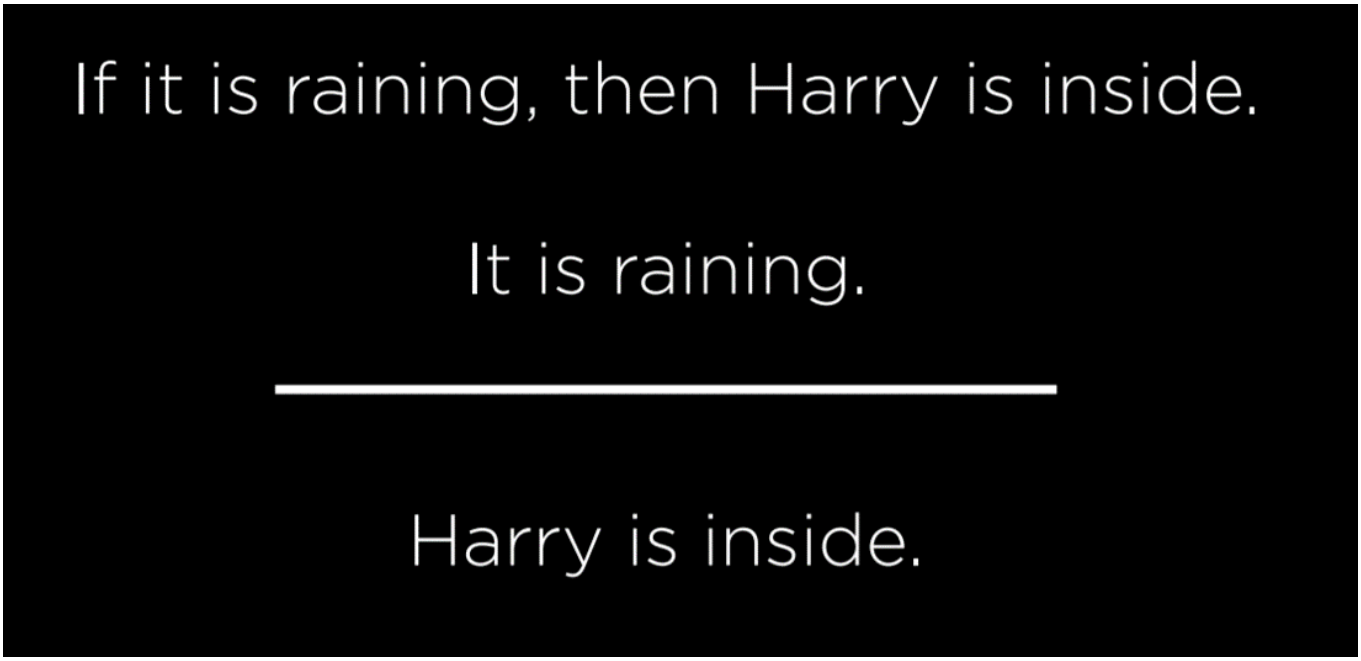
Player one says “four” and the game is over.

Representing this in propositional logic would require us to have (number of colors)^2 atomic propositions. So, in the case of four colors, we would have the propositions red0, red1, red2, red3, blue0… standing for color and position. The next step would be representing the rules of the game in propositional logic (that there is only one color in each position and no colors repeat) and adding them to the KB. The final step would be adding all the cues that we have to the KB. In our case, we would add that, in the first guess, two positions were wrong and two were right, and in the second guess, none was right. Using this knowledge, a Model Checking algorithm can give us the solution to the puzzle.

Inference Rules

Model Checking is not an efficient algorithm because it has to consider every possible model before giving the answer (a reminder: a query R is true if under all the models (truth assignments) where the KB is true, R is true as well). Inference rules allow us to generate new information based on existing knowledge without considering every possible model.

Inference rules are usually represented using a horizontal bar that separates the top part, the premise, from the bottom part, the conclusion. The premise is whatever knowledge we have, and the conclusion is what knowledge can be generated based on the premise.



In this example, our premise consists of the following propositions:

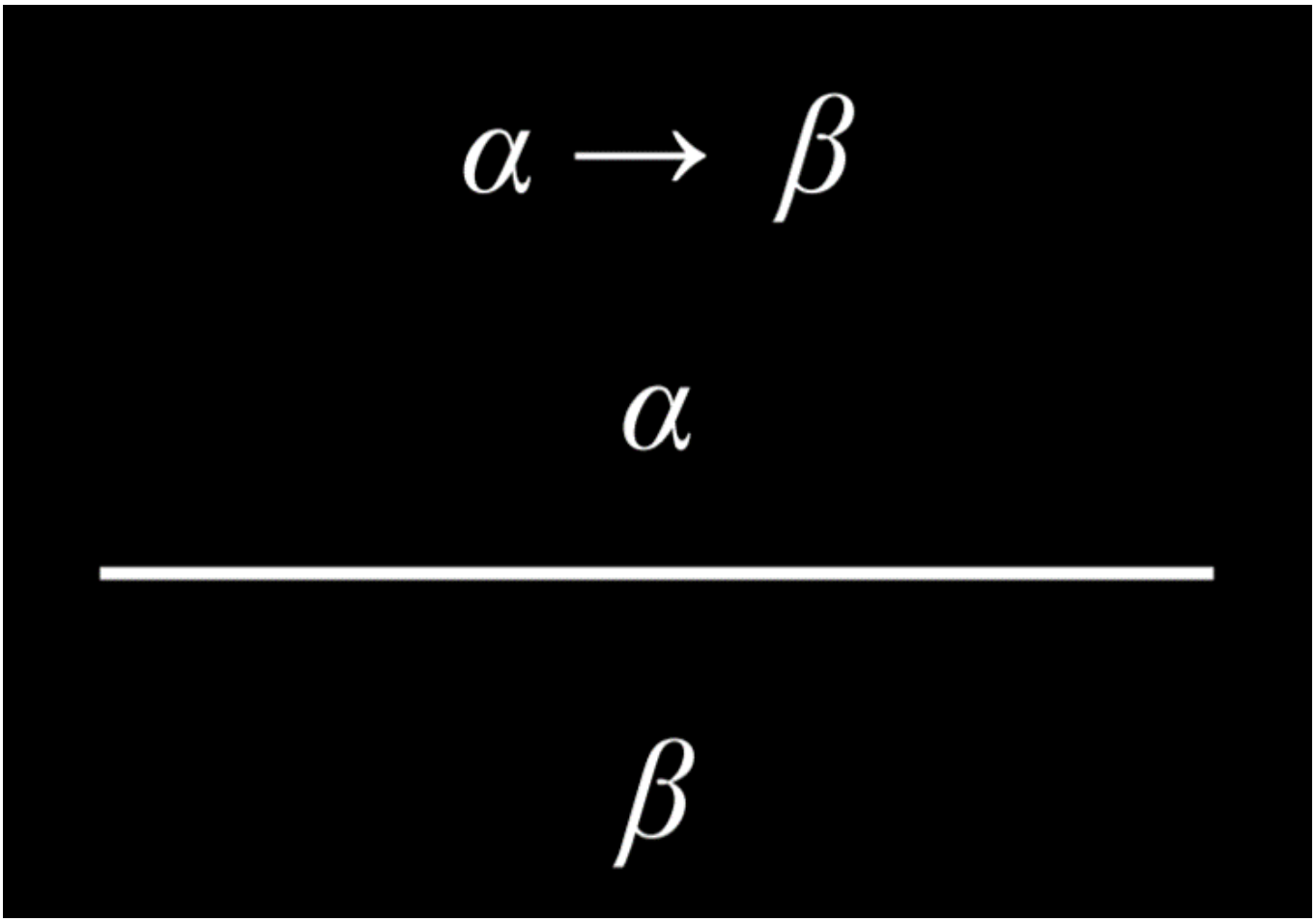
- If it is raining, then Harry is inside.
- It is raining.

Based on this, most reasonable humans can conclude that

- Harry is inside.

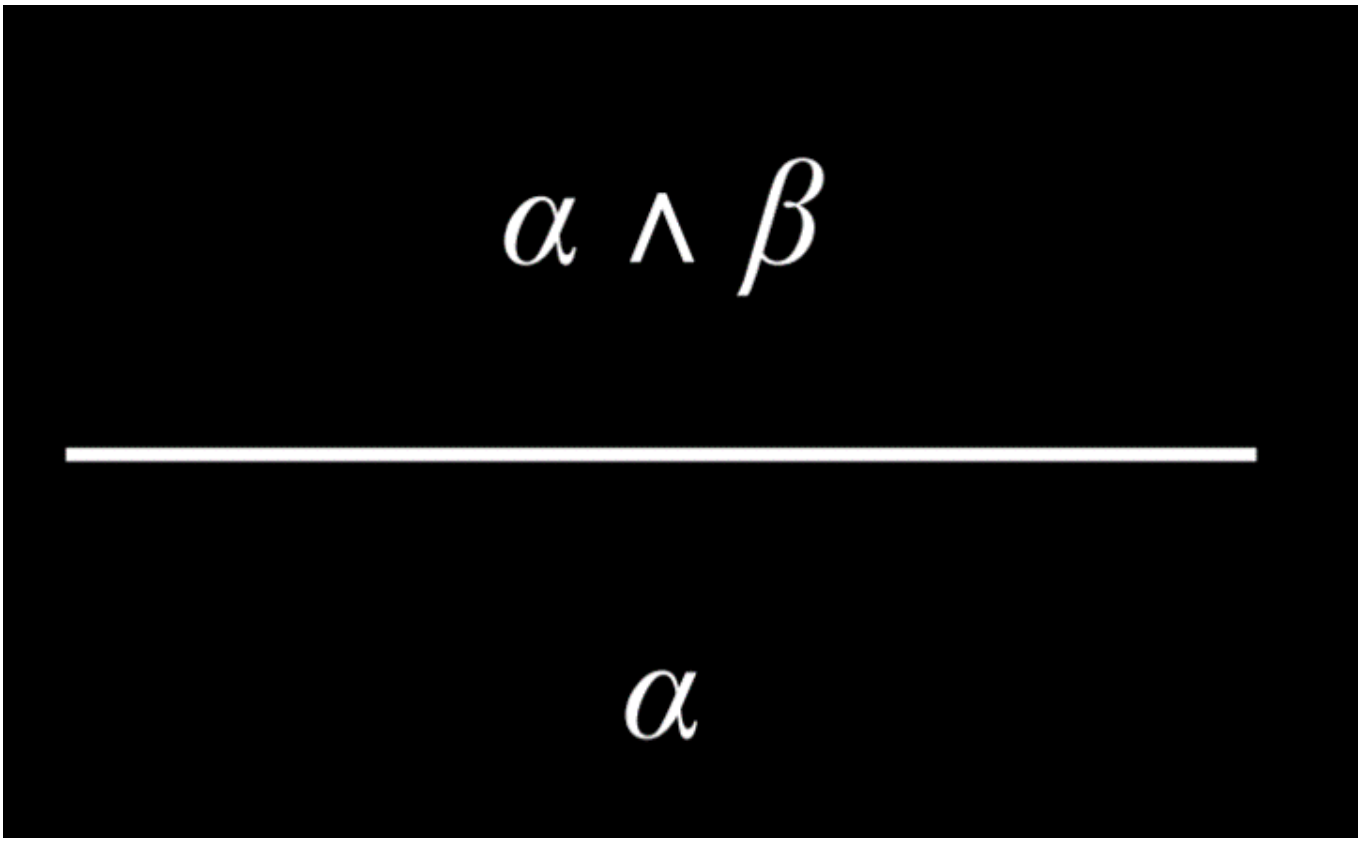
Modus Ponens

The type of inference rule we use in this example is Modus Ponens, which is a fancy way of saying that if we know an implication and its antecedent to be true, then the consequent is true as well.



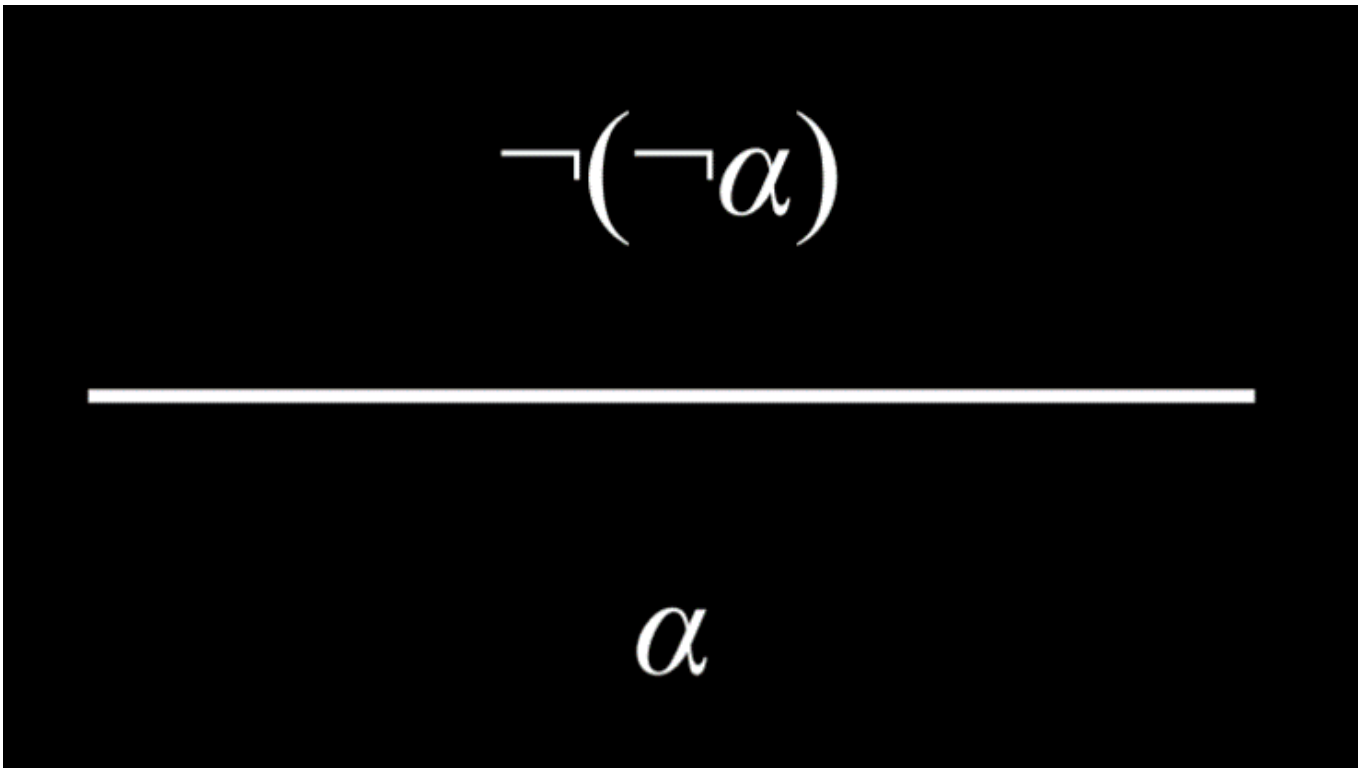
And Elimination

If an And proposition is true, then any one atomic proposition within it is true as well. For example, if we know that Harry is friends with Ron and Hermione, we can conclude that Harry is friends with Hermione.



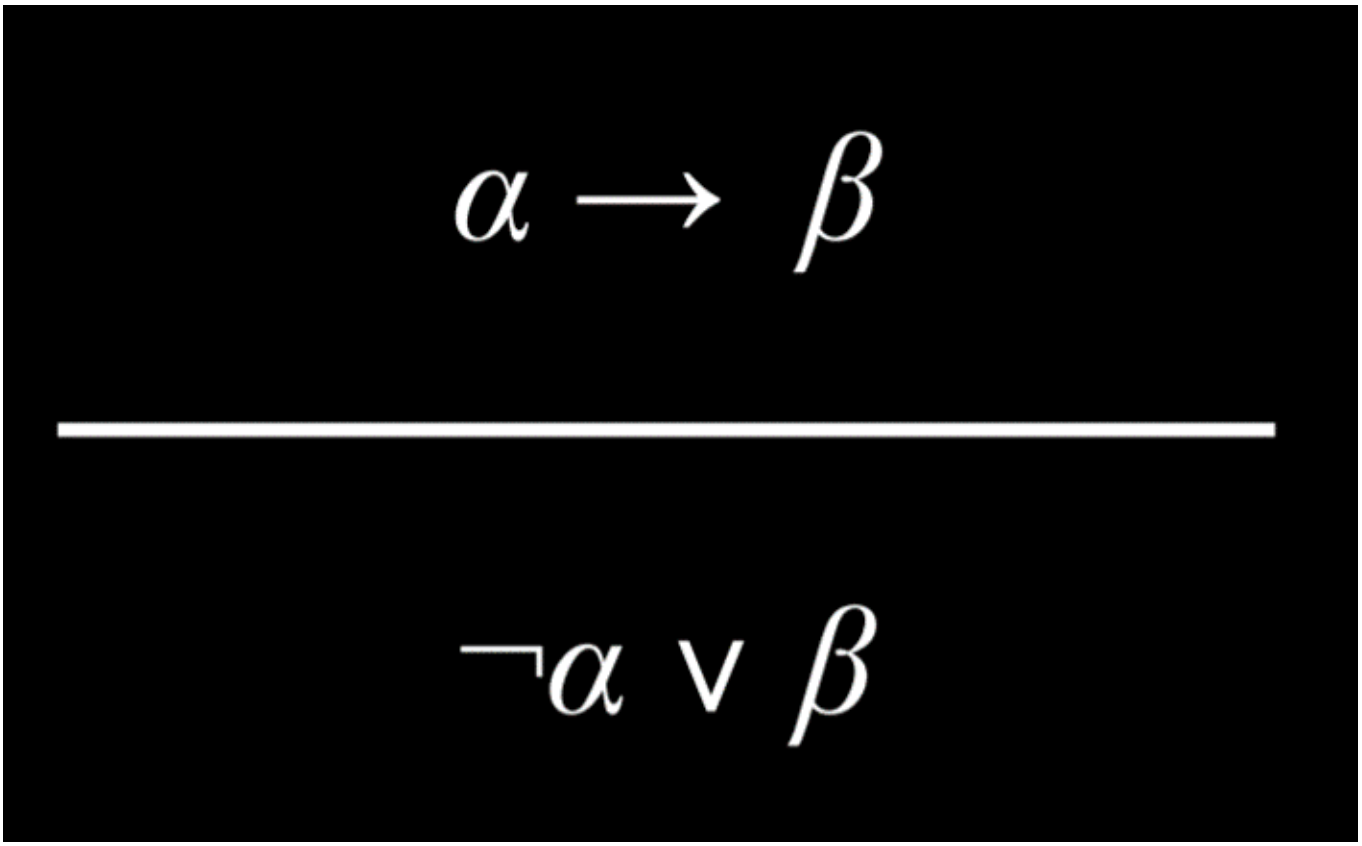
Double Negation Elimination

A proposition that is negated twice is true. For example, consider the proposition "It is not true that Harry did not pass the test". We can parse it the following way: "It is not true that (Harry did not pass the test)", or "¬(Harry did not pass the test)", and, finally "¬(¬(Harry passed the test))." The two negations cancel each other, marking the proposition "Harry passed the test" as true.



Implication Elimination

An implication is equivalent to an Or relation between the negated antecedent and the consequent. As an example, the proposition "If it is raining, Harry is inside" is equivalent to the proposition "(it is not raining) or (Harry is inside)."



This one can be a little confusing. However, consider the following truth table:

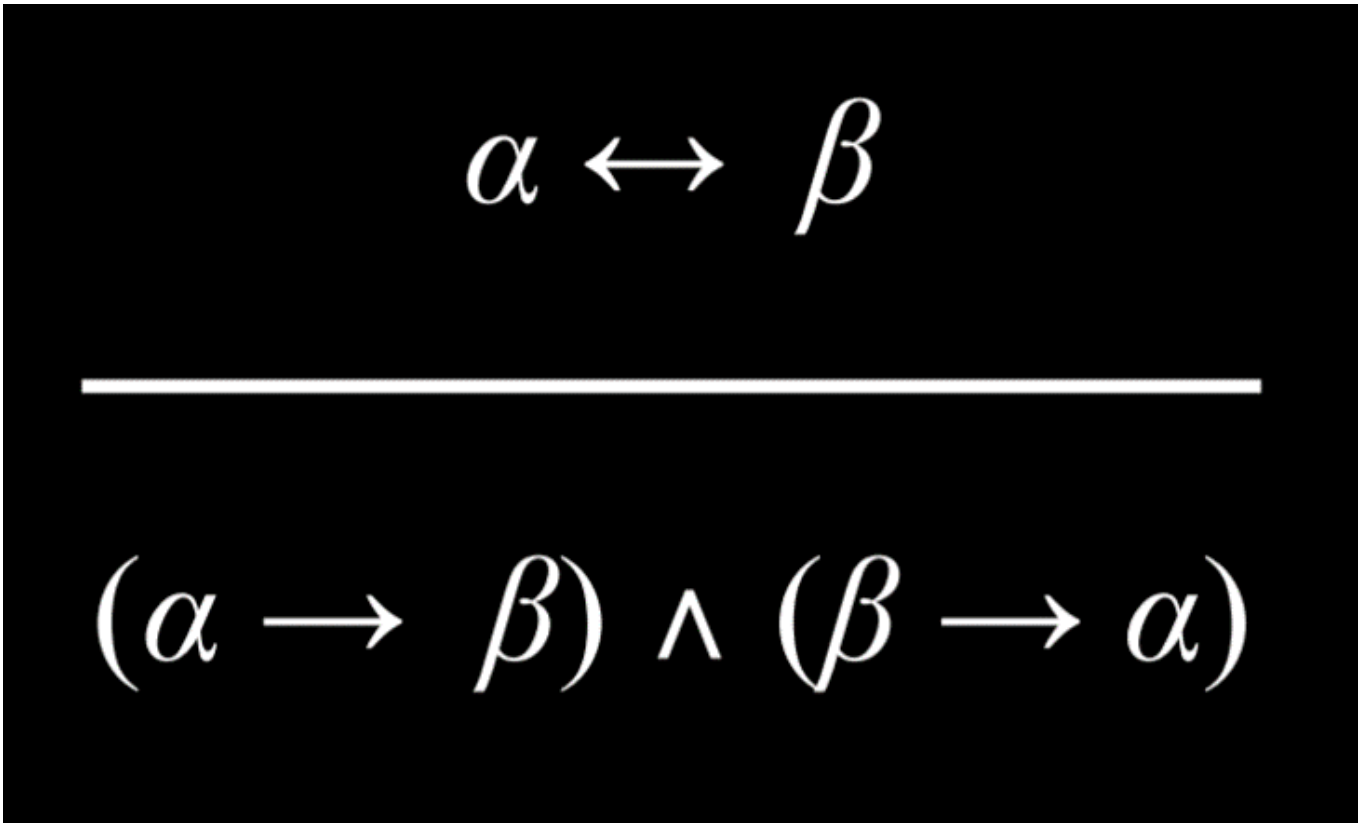
P	**Q**	**P → Q**	**¬P ∨ Q**
false	false	true	true
false	true	true	true
true	false	false	false
true	true	true	true

Since P → Q and ¬P ∨ Q have the same truth-value assignment, we know them to be equivalent logically. Another way to think about this is that an implication is true if either of two possible conditions is met: first, if the antecedent is false, the implication is trivially true (as discussed earlier, in the section on implication). This is represented by the negated antecedent P in ¬P ∨ Q, meaning that the proposition is always true if P is false. Second, the implication is true when the

antecedent is true only when the consequent is true as well. That is, if P and Q are both true, then ¬P ∨ Q is true. However, if P is true and Q is not, then ¬P ∨ Q is false.

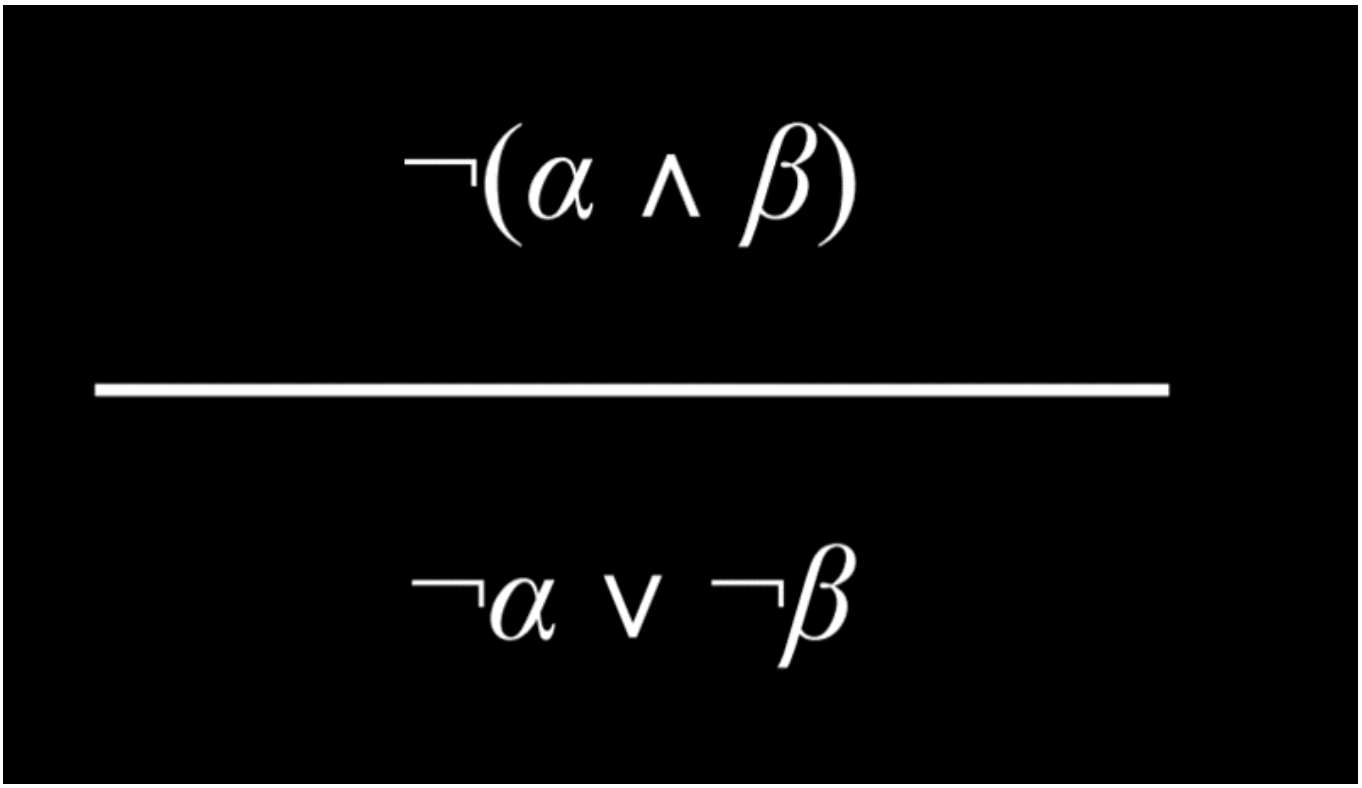
Biconditional Elimination

A biconditional proposition is equivalent to an implication and its inverse with an And connective. For example, "It is raining if and only if Harry is inside" is equivalent to ("If it is raining, Harry is inside" And "If Harry is inside, it is raining").

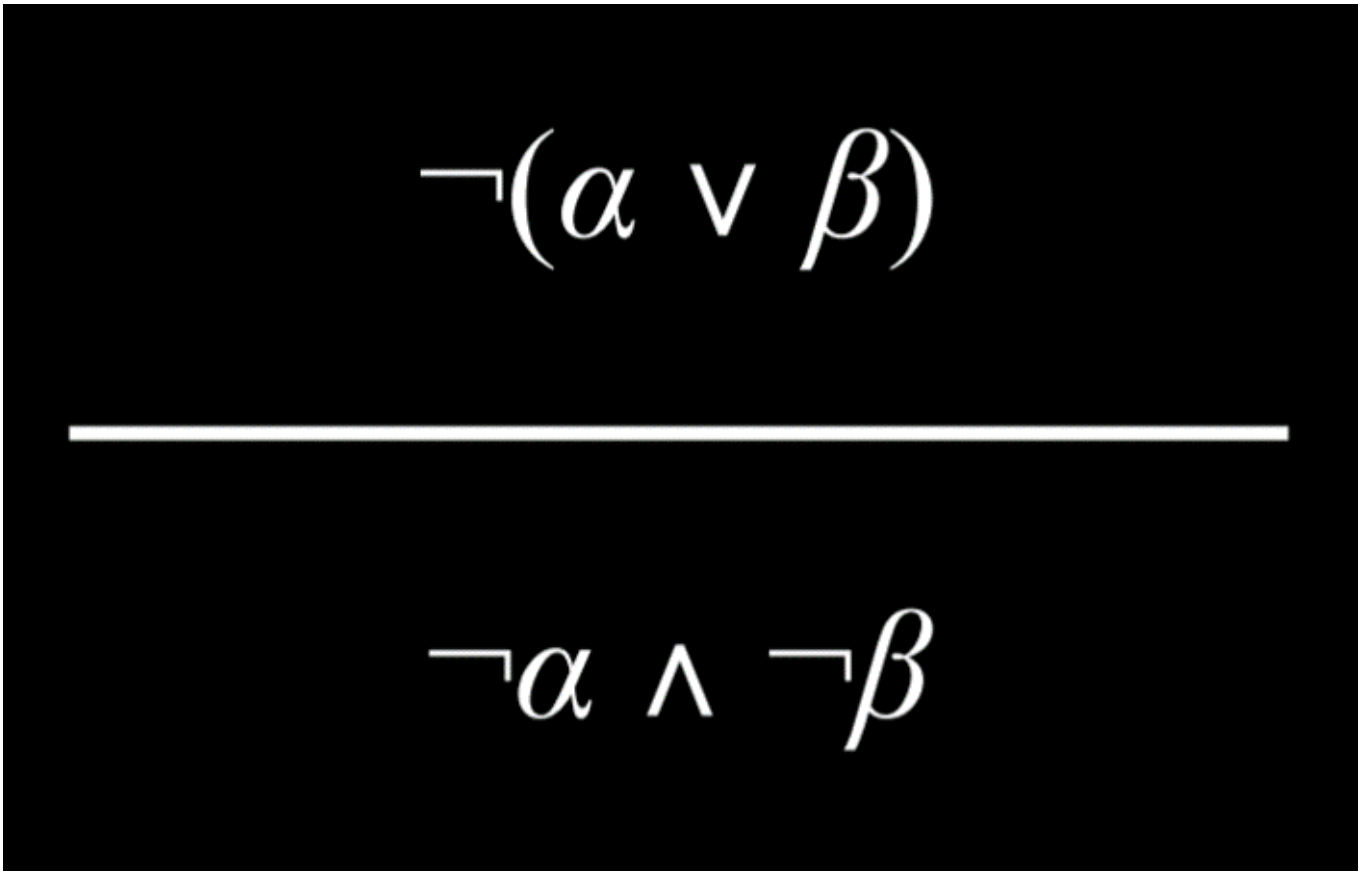


De Morgan's Law

It is possible to turn an And connective into an Or connective. Consider the following proposition: "It is not true that both Harry and Ron passed the test." From this, it is possible to conclude that "It is not true that Harry passed the test" Or "It is not true that Ron passed the test." That is, for the And proposition earlier to be true, at least one of the propositions in the Or propositions must be true.

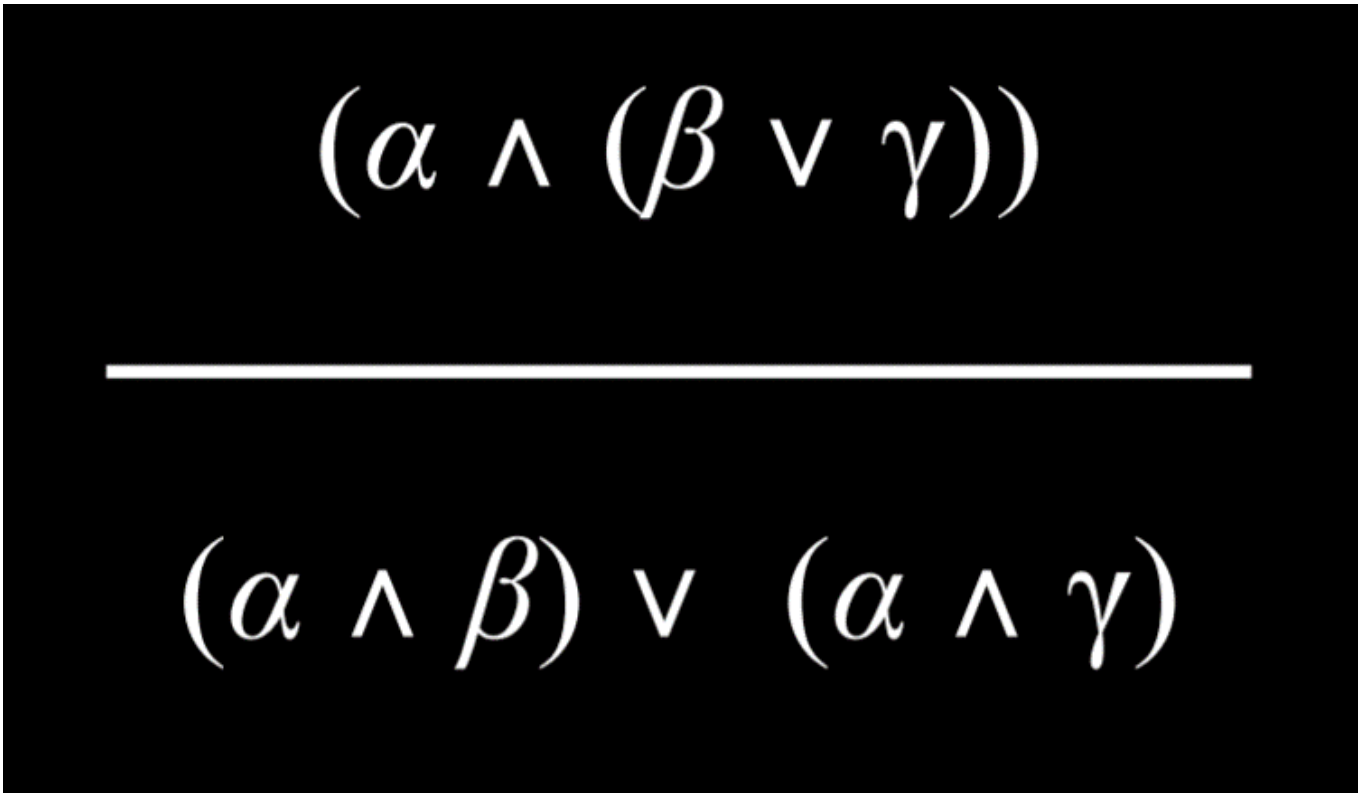


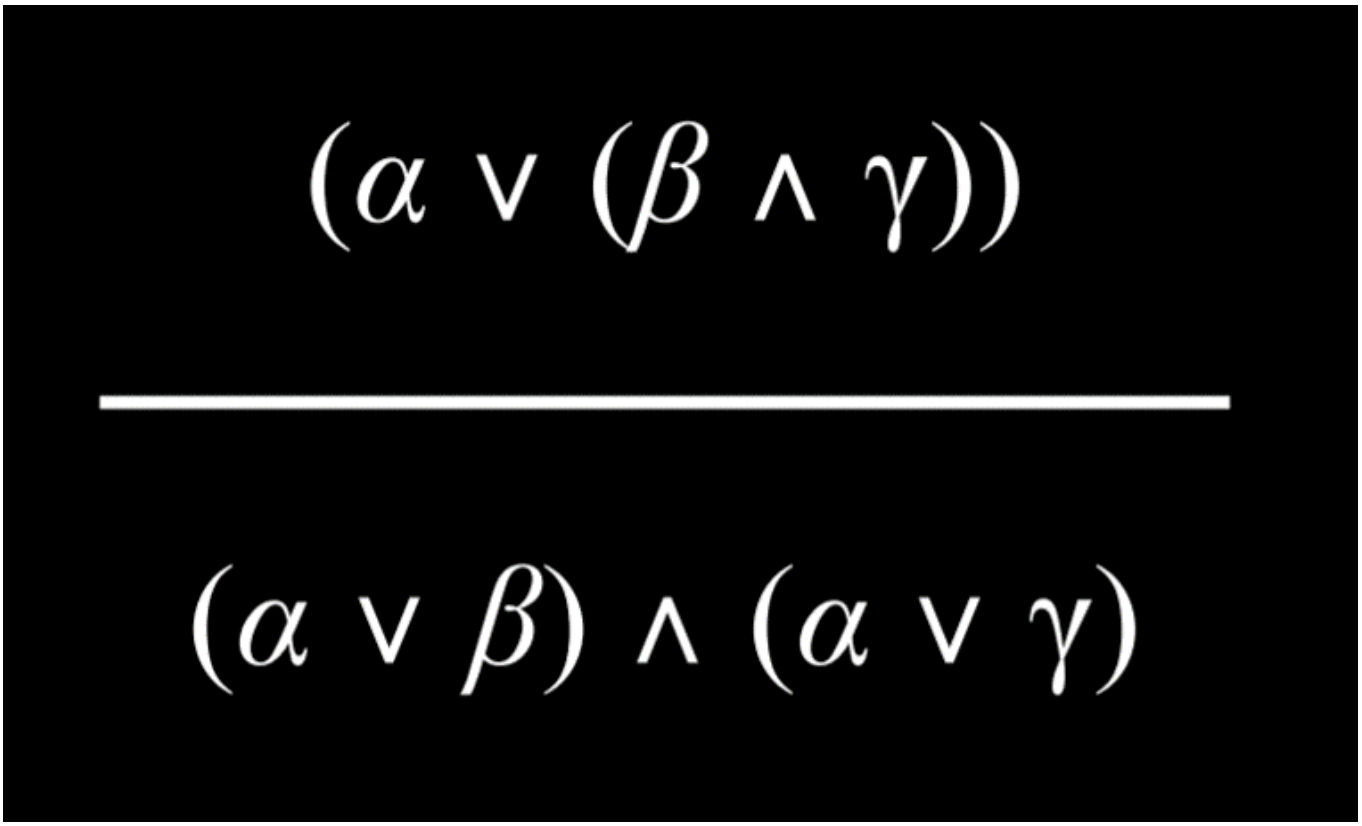
Similarly, it is possible to conclude the reverse. Consider the proposition “It is not true that Harry or Ron passed the test.” This can be rephrased as “Harry did not pass the test” And “Ron did not pass the test.”



Distributive Property

A proposition with two elements that are grouped with And or Or connectives can be distributed, or broken down into, smaller units consisting of And and Or.





Knowledge and Search Problems

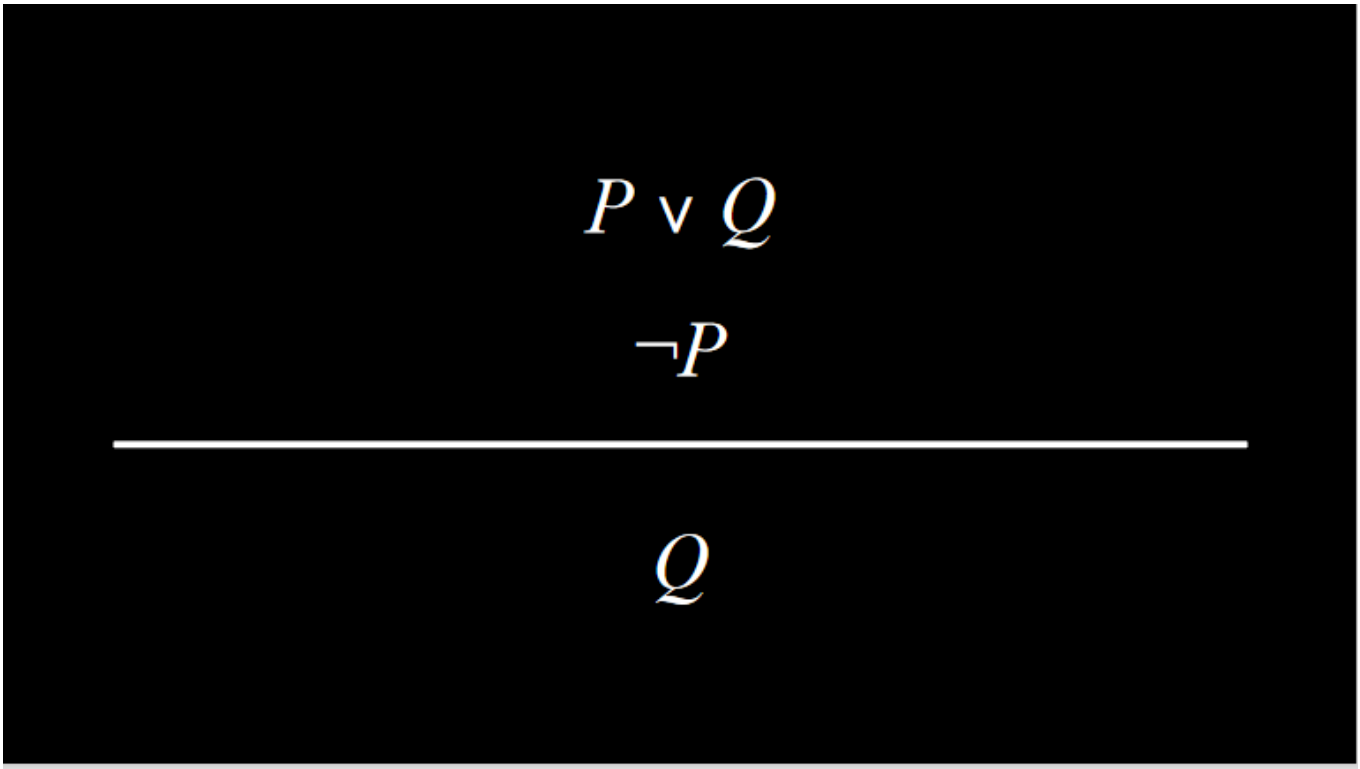
Inference can be viewed as a search problem with the following properties:

- Initial state: starting knowledge base
- Actions: inference rules
- Transition model: new knowledge base after inference
- Goal test: checking whether the statement that we are trying to prove is in the KB
- Path cost function: the number of steps in the proof

This shows just how versatile search algorithms are, allowing us to derive new information based on existing knowledge using inference rules.

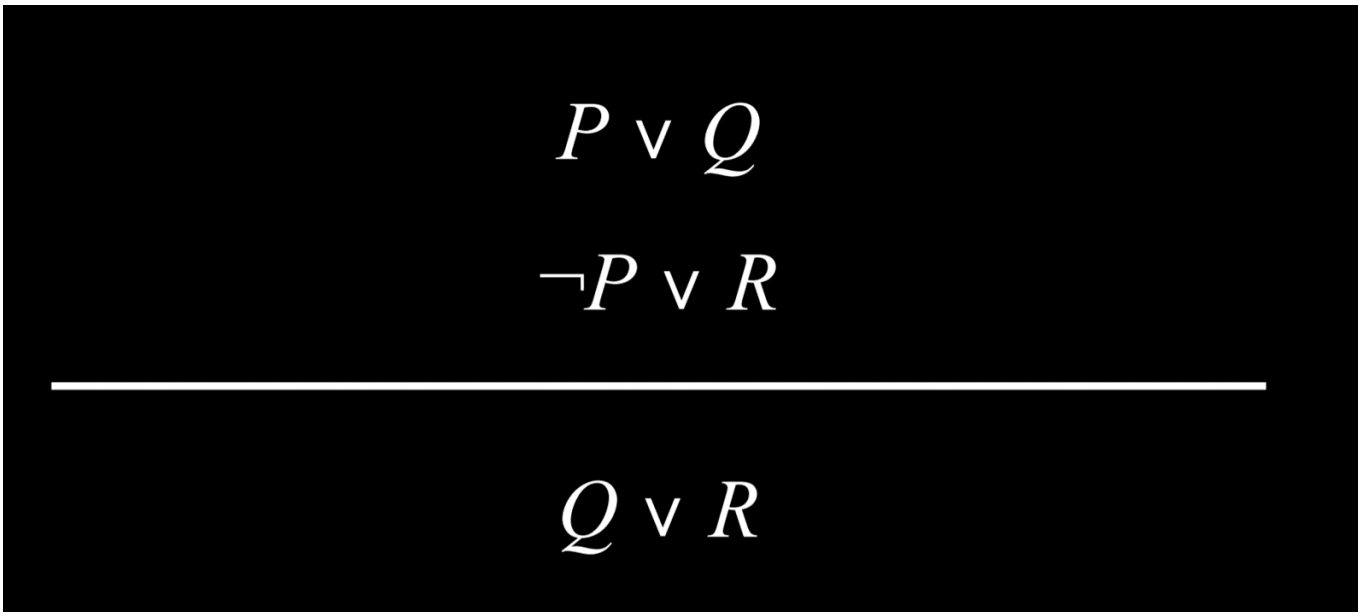
Resolution

Resolution is a powerful inference rule that states that if one of two atomic propositions in an Or proposition is false, the other has to be true. For example, given the proposition "Ron is in the Great Hall" Or "Hermione is in the library", in addition to the proposition "Ron is not in the Great Hall," we can conclude that "Hermione is in the library." More formally, we can define resolution the following way:



Resolution relies on **Complementary Literals**, two of the same atomic propositions where one is negated and the other is not, such as P and ¬P.

Resolution can be further generalized. Suppose that in addition to the proposition "Ron is in the Great Hall" Or "Hermione is in the library", we also know that "Ron is not in the Great Hall" Or "Harry is sleeping." We can infer from this, using resolution, that "Hermione is in the library" Or "Harry is sleeping." To put it in formal terms:



Complementary literals allow us to generate new sentences through inferences by resolution. Thus, inference algorithms locate complementary literals to generate new knowledge.

A **Clause** is a disjunction of literals (a propositional symbol or a negation of a propositional symbol, such as P, ¬P). A **disjunction** consists of propositions that are connected with an Or logical connective (P ∨ Q ∨ R). A **conjunction**, on the other hand, consists of propositions that are connected with an And logical connective (P ∧ Q ∧ R). Clauses allow us to convert any logical statement into a **Conjunctive Normal Form** (CNF), which is a conjunction of clauses, for example: (A ∨ B ∨ C) ∧ (D ∨ ¬E) ∧ (F ∨ G).

Steps in Conversion of Propositions to Conjunctive Normal Form

- Eliminate biconditionals
  - Turn (α ↔ β) into (α → β) ∧ (β → α).
- Eliminate implications
  - Turn (α → β) into ¬α ∨ β.
- Move negation inwards until only literals are being negated (and not clauses), using De Morgan's Laws.
  - Turn ¬(α ∧ β) into ¬α ∨ ¬β

Here's an example of converting (P ∨ Q) → R to Conjunctive Normal Form:

- (P ∨ Q) → R
- ¬(P ∨ Q) ∨ R /Eliminate implication
- (¬P ∧ ¬Q) ∨ R /De Morgan's Law
- (¬P ∨ R) ∧ (¬Q ∨ R) /Distributive Law

At this point, we can run an inference algorithm on the conjunctive normal form. Occasionally, through the process of inference by resolution, we might end up in cases where a clause contains the same literal twice. In these cases, a process called **factoring** is used, where the duplicate literal is removed. For example, (P ∨ Q ∨ S) ∧ (¬P ∨ R ∨ S) allow us to infer by resolution that (Q ∨ S ∨ R ∨ S). The duplicate S can be removed to give us (Q ∨ R ∨ S).

Resolving a literal and its negation, i.e. ¬P and P, gives the **empty clause** (). The empty clause is always false, and this makes sense because it is impossible that both P and ¬P are true. This fact is used by the resolution algorithm.

- To determine if KB ⊨ α:
  - Check: is (KB ∧ ¬α) a contradiction?
    - If so, then KB ⊨ α.
    - Otherwise, no entailment.

Proof by contradiction is a tool used often in computer science. If our knowledge base is true, and it contradicts ¬α, it means that ¬α is false, and, therefore, α must be true. More technically, the algorithm would perform the following actions:

- To determine if KB ⊨ α:
  - Convert (KB ∧ ¬α) to Conjunctive Normal Form.
  - Keep checking to see if we can use resolution to produce a new clause.
  - If we ever produce the empty clause (equivalent to False), congratulations! We have arrived at a contradiction, thus proving that KB ⊨ α.
  - However, if contradiction is not achieved and no more clauses can be inferred, there is no entailment.

Here is an example that illustrates how this algorithm might work:

- Does (A ∨ B) ∧ (¬B ∨ C) ∧ (¬C) entail A?
- First, to prove by contradiction, we assume that A is false. Thus, we arrive at (A ∨ B) ∧ (¬B ∨ C) ∧ (¬C) ∧ (¬A).
- Now, we can start generating new information. Since we know that C is false (¬C), the only way (¬B ∨ C) can be true is if B is false, too. Thus, we can add (¬B) to our KB.
- Next, since we know (¬B), the only way (A ∨ B) can be true is if A is true. Thus, we can add (A) to our KB.
- Now our KB has two complementary literals, (A) and (¬A). We resolve them, arriving at the empty set, (). The empty set is false by definition, so we have arrived at a contradiction.

First Order Logic

First order logic is another type of logic that allows us to express more complex ideas more succinctly than propositional logic. First order logic uses two types of symbols: *Constant Symbols* and *Predicate Symbols*. Constant symbols represent objects, while predicate symbols are like relations or functions that take an argument and return a true or false value.

For example, we return to the logic puzzle with different people and house assignments at Hogwarts. The constant symbols are people or houses, like Minerva, Pomona, Gryffindor, Hufflepuff, etc. The predicate symbols are properties that hold true or false of some constant symbols. For example, we can express the idea that Minerva is a person using the sentence Person(Minerva). Similarly, we can express the idea the Gryffindor is a house using the sentence House(Gryffindor). All the logical connectives work in first order logic the same way as before. For example, ¬House(Minerva) expresses the idea that Minerva is not a house. A predicate symbol can also take two or more arguments and express a relation between them. For example, BelongsTo expresses a relation between two arguments, the person and the house to which the person belongs. Thus, the idea that Minerva belongs to Gryffindor can be expressed as BelongsTo(Minerva, Gryffindor). First order logic allows having one symbol for each person and one symbol for each house. This is more succinct than propositional logic, where each person—house assignment would require a different symbol.

Universal Quantification

Quantification is a tool that can be used in first order logic to represent sentences without using a specific constant symbol. Universal quantification uses the symbol ∀ to express "for all." So, for example, the sentence ∀x. BelongsTo(x, Gryffindor) → ¬BelongsTo(x, Hufflepuff) expresses the idea that it is true for every symbol that if this symbol belongs to Gryffindor, it does not belong to Hufflepuff.

Existential Quantification

Existential quantification is an idea parallel to universal quantification. However, while universal quantification was used to create sentences that are true for all x, existential quantification is used to create sentences that are true for at least one x. It is expressed using the symbol ∃. For example, the sentence ∃x. House(x) ∧ BelongsTo(Minerva, x) means that there is at least one symbol that is both a house and that Minerva belongs to it. In other words, this expresses the idea that Minerva belongs to a house.

Existential and universal quantification can be used in the same sentence. For example, the sentence ∀x. Person(x) → (∃y. House(y) ∧ BelongsTo(x, y)) expresses the idea that if x is a person, then there is at least one house, y, to which this person belongs. In other words, this sentence means that every person belongs to a house.

There are other types of logic as well, and the commonality between them is that they all exist in pursuit of representing information. These are the systems we use to represent knowledge in our

AI.

CS50's Introduction to Artificial Intelligence with Python

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

Brian Yu (https://brianyu.me)
brian@cs.harvard.edu

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://www.reddit.com/user/davidjmalan) (https://www.threads.net/@davidjmalan)
(https://twitter.com/davidjmalan)

Lecture 2

Uncertainty

Last lecture, we discussed how AI can represent and derive new knowledge. However, often, in reality, the AI has only partial knowledge of the world, leaving space for uncertainty. Still, we would like our AI to make the best possible decision in these situations. For example, when predicting weather, the AI has information about the weather today, but there is no way to predict with 100% accuracy the weather tomorrow. Still, we can do better than chance, and today's lecture is about how we can create AI that makes optimal decisions given limited information and uncertainty.

Probability

Uncertainty can be represented as a number of events and the likelihood, or probability, of each of them happening.

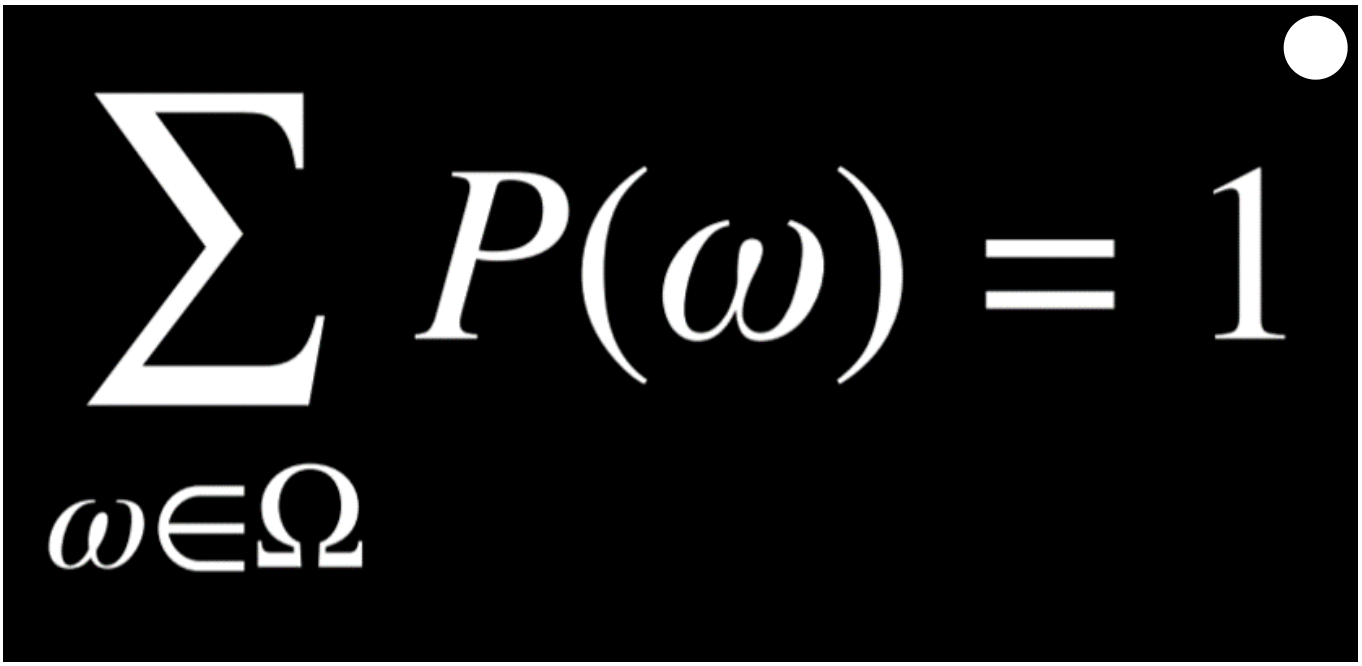
Possible Worlds

Every possible situation can be thought of as a world, represented by the lowercase Greek letter omega ω. For example, rolling a die can result in six possible worlds: a world where the die yields

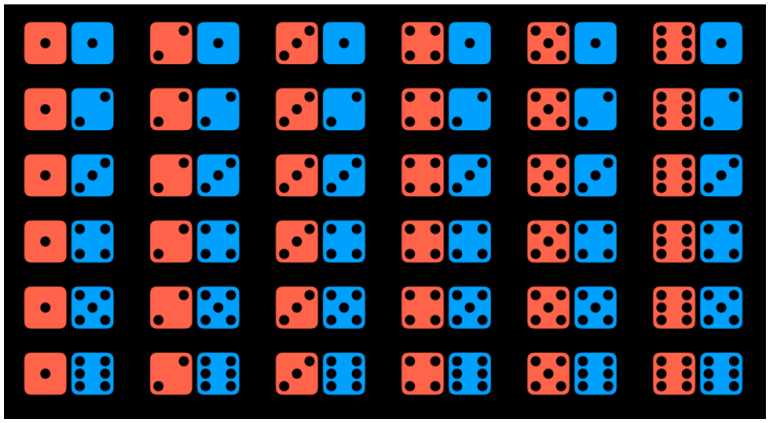
a 1, a world where the die yields a 2, and so on. To represent the probability of a certain world, we write P(ω).

Axioms in Probability

- $0 < P(\omega) < 1$: every value representing probability must range between 0 and 1.
  - Zero is an impossible event, like rolling a standard die and getting a 7.
  - One is an event that is certain to happen, like rolling a standard die and getting a value less than 10.
  - In general, the higher the value, the more likely the event is to happen.
- The probabilities of every possible event, when summed together, are equal to 1.



The probability of rolling a number *R* with a standard die can be represented as P(*R*). In our case, P(*R*) = 1/6, because there are six possible worlds (rolling any number from 1 through 6) and each is equally likely to happen. Now, consider the event of rolling two dice. Now, there are 36 possible events, which are, again, equally as likely.



However, what happens if we try to predict the sum of the two dice? In this case, we have only 11 possible values (the sum has to range from 2 to 12), and they do not occur equally as often.

To get the probability of an event, we divide the number of worlds in which it occurs by the number of total possible worlds. For example, there are 36 possible worlds when rolling two dice. Only in one of these worlds, when both dice yield a 6, do we get the sum of 12. Thus, P(*12*) = 1/36, or, in words, the probability of rolling two dice and getting two numbers whose sum is 12 is 1/36. What is P(*7*)? We count and see that the sum 7 occurs in 6 worlds. Thus, P(*7*) = 6/36 = 1/6.

Unconditional Probability

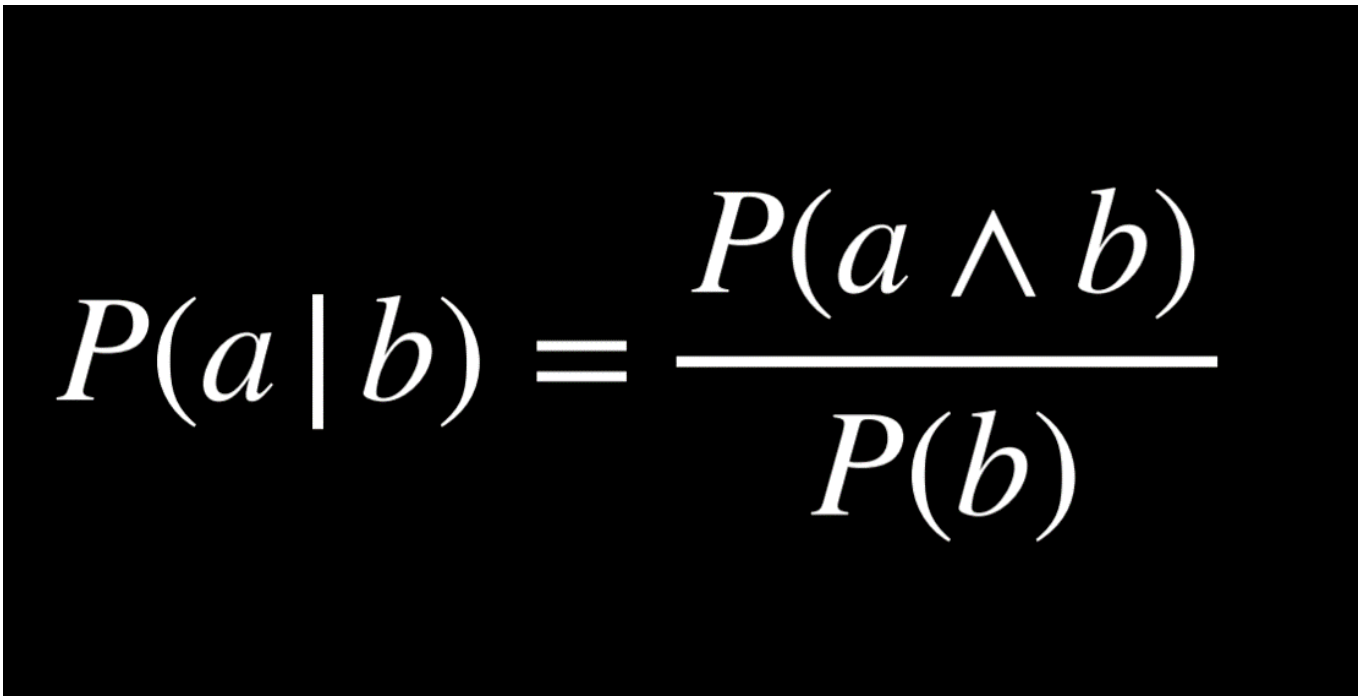
Unconditional probability is the degree of belief in a proposition in the absence of any other evidence. All the questions that we have asked so far were questions of unconditional probability, because the result of rolling a die is not dependent on previous events.

Conditional Probability

Conditional probability is the degree of belief in a proposition given some evidence that has already been revealed. As discussed in the introduction, AI can use partial information to make educated guesses about the future. To use this information, which affects the probability that the event occurs in the future, we rely on conditional probability.

Conditional probability is expressed using the following notation: P(*a* | *b*), meaning "the probability of event *a* occurring given that we know event *b* to have occurred," or, more succinctly, "the probability of *a* given *b*." Now we can ask questions like what is the probability of rain today given that it rained yesterday P(*rain today* | *rain yesterday*), or what is the probability of the patient having the disease given their test results P(*disease* | *test results*).

Mathematically, to compute the conditional probability of *a* given *b*, we use the following formula:

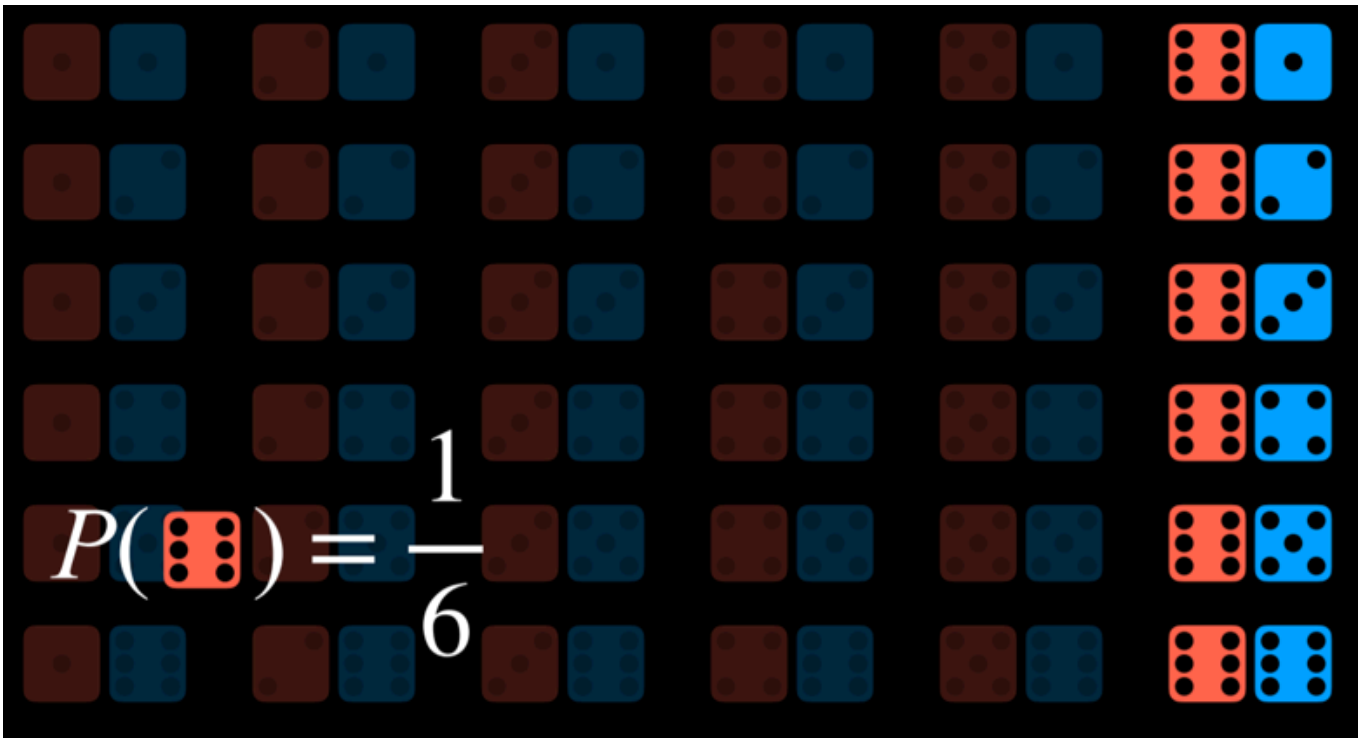


To put it in words, the probability that *a* given *b* is true is equal to the probability of *a* and *b* being true, divided by the probability of *b*. An intuitive way of reasoning about this is the thought "we are interested in the events where both *a* and *b* are true (the numerator), but only from the worlds where we know *b* to be true (the denominator)." Dividing by *b* restricts the possible worlds to the ones where *b* is true. The following are algebraically equivalent forms to the formula above:

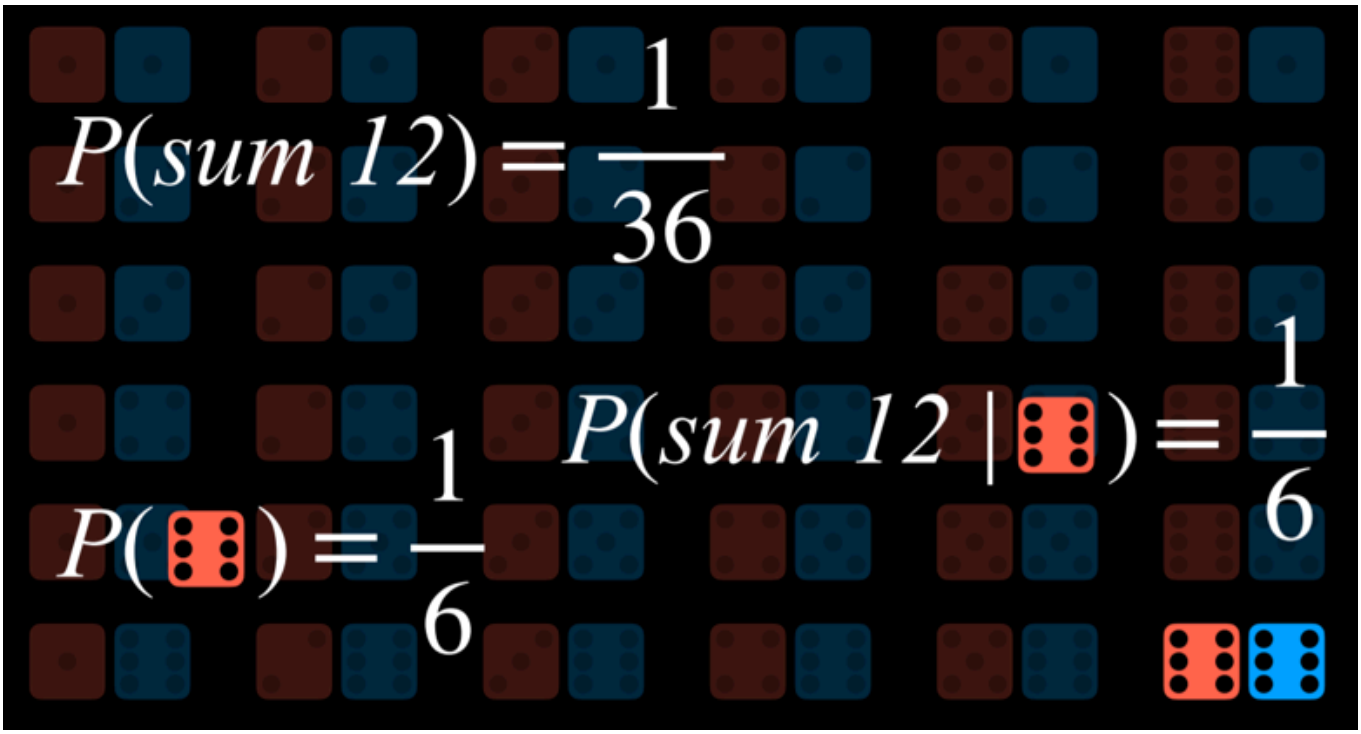
$$P(a \wedge b) = P(b)P(a \mid b)$$

$$P(a \wedge b) = P(a)P(b \mid a)$$

For example, consider P(*sum 12* | *roll six on one die*), or the probability of rolling two dice and getting a sum of twelve, given that we have already rolled one die and got a six. To calculate this, we first restrict our worlds to the ones where the value of the first die is six:



Now we ask how many times does the event *a* (the sum being 12) occur in the worlds that we restricted the question to (dividing by P(*b*), or the probability of the first die yielding 6).



Random Variables

A random variable is a variable in probability theory with a domain of possible values that it can take on. For example, to represent possible outcomes when rolling a die, we can define a random variable *Roll*, that can take on the values {*0, 1, 2, 3, 4, 5, 6*}. To represent the status of a flight, we can define a variable *Flight* that takes on the values {*on time, delayed, canceled*}.

Often, we are interested in the probability with which each value occurs. We represent this using a probability distribution. For example,

- P(*Flight* = *on time*) = 0.6
- P(*Flight* = *delayed*) = 0.3
- P(*Flight* = *canceled*) = 0.1

To interpret the probability distribution with words, this means that there is a 60% chance that the flight is on time, 30% chance that it is delayed, and 10% chance that it is canceled. Note that, as shown previously, the sum the probabilities of all possible outcomes is 1.

A probability distribution can be represented more succinctly as a vector. For example, **P**(*Flight*) = <*0.6, 0.3, 0.1*>. For this notation to be interpretable, the values have a set order (in our case, *on time, delayed, canceled*).

Independence

Independence is the knowledge that the occurrence of one event does not affect the probability of the other event. For example, when rolling two dice, the result of each die is independent from the other. Rolling a 4 with the first die does not influence the value of the second die that we roll. This is opposed to dependent events, like clouds in the morning and rain in the afternoon. If it is cloudy in the morning, it is more likely that it will rain in the morning, so these events are dependent.

Independence can be defined mathematically: events *a* and *b* are independent if and only if the probability of *a* and *b* is equal to the probability of *a* times the probability of *b*: $P(a \wedge b) = P(a)P(b)$.

Bayes' Rule

Bayes' rule is commonly used in probability theory to compute conditional probability. In words, Bayes' rule says that the probability of *b* given *a* is equal to the probability of *a* given *b*, times the probability of *b*, divided by the probability of *a*.

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

For example, we would like to compute the probability of it raining in the afternoon if there are clouds in the morning, or P(*rain* | *clouds*). We start with the following information:

- 80% of rainy afternoons start with cloudy mornings, or P(*clouds* | *rain*).
- 40% of days have cloudy mornings, or P(*clouds*).
- 10% of days have rainy afternoons, or P(*rain*).

Applying Bayes' rule, we compute (0.1)(0.8)/(0.4) = 0.2. That is, the probability that it rains in the afternoon given that it was cloudy in the morning is 20%.

Knowing P(*a* | *b*), in addition to P(*a*) and P(*b*), allows us to calculate P(*b* | *a*). This is helpful, because knowing the conditional probability of a visible effect given an unknown cause, P(*visible effect* | *unknown cause*), allows us to calculate the probability of the unknown cause given the visible effect, P(*unknown cause* | *visible effect*). For example, we can learn P(*medical test results* | *disease*) through medical trials, where we test people with the disease and see how often the test picks up

on that. Knowing this, we can calculate P(*disease* | *medical test results*), which is valuable diagnostic information.

Joint Probability

Joint probability is the likelihood of multiple events all occurring.

Let us consider the following example, concerning the probabilities of clouds in the morning and rain in the afternoon.

C = *cloud*	**C = ¬*cloud***
0.4	0.6

R = *rain*	**R = ¬*rain***
0.1	0.9

Looking at these data, we can't say whether clouds in the morning are related to the likelihood of rain in the afternoon. To be able to do so, we need to look at the joint probabilities of all the possible outcomes of the two variables. We can represent this in a table as follows:

	R = *rain*	**R = ¬*rain***
C = *cloud*	0.08	0.32
C = ¬*cloud*	0.02	0.58

Now we are able to know information about the co-occurrence of the events. For example, we know that the probability of a certain day having clouds in the morning and rain in the afternoon is 0.08. The probability of no clouds in the morning and no rain in the afternoon is 0.58.

Using joint probabilities, we can deduce conditional probability. For example, if we are interested in the probability distribution of clouds in the morning given rain in the afternoon. P(*C* | *rain*) = P(*C, rain*)/P(*rain*) (a side note: in probability, commas and ∧ are used interchangeably. Thus, P(*C, rain*) = P(*C* ∧ *rain*)). In words, we divide the joint probability of rain and clouds by the probability of rain.

In the last equation, it is possible to view P(*rain*) as some constant by which P(*C, rain*) is multiplied. Thus, we can rewrite P(*C, rain*)/P(*rain*) = αP(*C, rain*), or α<0.08, 0.02>. Factoring out α leaves us with the proportions of the probabilities of the possible values of C given that there is rain in the

afternoon. Namely, if there is rain in the afternoon, the proportion of the probabilities of clouds in the morning and no clouds in the morning is 0.08:0.02. Note that 0.08 and 0.02 don't sum up to 1; however, since this is the probability distribution for the random variable C, we know that they should sum up to 1. Therefore, we need to normalize the values by computing α such that $\alpha 0.08 + \alpha 0.02 = 1$. Finally, we can say that P(*C* | *rain*) = <0.8, 0.2>.

Probability Rules

- **Negation**: P(¬*a*) = 1 - P(*a*). This stems from the fact that the sum of the probabilities of all the possible worlds is 1, and the complementary literals *a* and ¬*a* include all the possible worlds.
- **Inclusion-Exclusion**: P(*a* ∨ *b*) = P(*a*) + P(*b*) - P(*a* ∧ *b*). This can interpreted in the following way: the worlds in which *a* or *b* are true are equal to all the worlds where *a* is true, plus the worlds where *b* is true. However, in this case, some worlds are counted twice (the worlds where both *a* and *b* are true)). To get rid of this overlap, we subtract once the worlds where both *a* and *b* are true (since they were counted twice).

  > Here is an example from outside lecture that can elucidate this. Suppose I eat ice cream 80% of days and cookies 70% of days. If we're calculating the probability that today I eat ice cream or cookies P(*ice cream* ∨ *cookies*) without subtracting P(*ice cream* ∧ *cookies*), we erroneously end up with ~~0.7 + 0.8 = 1.5~~. This contradicts the axiom that probability ranges between 0 and 1. To correct for counting twice the days when I ate both ice cream and cookies, we need to subtract P(*ice cream* ∧ *cookies*) once.

- **Marginalization**: P(*a*) = P(*a, b*) + P(*a,* ¬*b*). The idea here is that *b* and ¬*b* are disjoint probabilities. That is, the probability of *b* and ¬*b* occurring at the same time is 0. We also know *b* and ¬*b* sum up to 1. Thus, when *a* happens, *b* can either happen or not. When we take the probability of both *a* and *b* happening in addition to the probability of *a* and ¬*b*, we end up with simply the probability of *a*.

Marginalization can be expressed for random variables the following way:

$$P(X = x_i) = \sum_j P(X = x_i, Y = y_j)$$

The left side of the equation means "The probability of random variable X having the value x_i." For example, for the variable C we mentioned earlier, the two possible values are *clouds in the morning* and *no clouds in the morning*. The right part of the equation is the idea of marginalization. P($X = x_i$)

is equal to the sum of all the joint probabilities of x_i and every single value of the random variable Y. For example, P(*C* = *cloud*) = P(*C* = *cloud, R* = *rain*) + P(*C* = *cloud, R* = ¬*rain*) = 0.08 + 0.32 = 0.4.

- **Conditioning**: P(*a*) = P(*a* | *b*)P(*b*) + P(*a* | ¬*b*)P(¬*b*). This is a similar idea to marginalization. The probability of event *a* occurring is equal to the probability of *a* given *b* times the probability of *b*, plus the probability of *a* given ¬*b* time the probability of ¬*b*.

$$P(X = x_i) = \sum_j P(X = x_i \mid Y = y_j) P(Y = y_j)$$

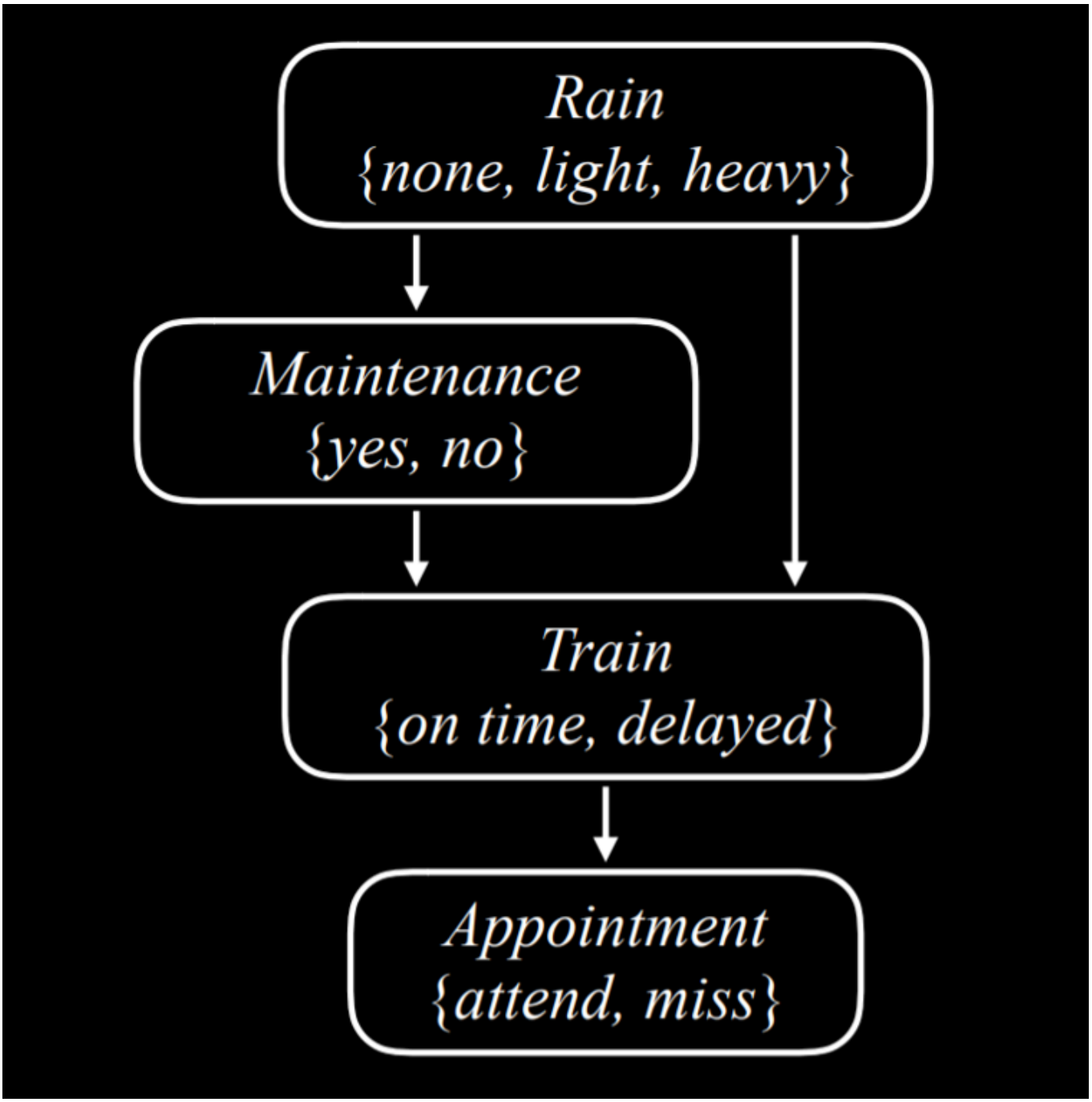
In this formula, the random variable X takes the value x_i with probability that is equal to the sum of the probabilities of x_i given each value of the random variable Y multiplied by the probability of variable Y taking that value. This makes sense if we remember that P(*a* | *b*) = P(*a, b*)/P(*b*). If we multiply this expression by P(*b*), we end up with P(*a, b*), and from here we do the same as we did with marginalization.

Bayesian Networks

A Bayesian network is a data structure that represents the dependencies among random variables. Bayesian networks have the following properties:

- They are directed graphs.
- Each node on the graph represent a random variable.
- An arrow from X to Y represents that X is a parent of Y. That is, the probability distribution of Y depends on the value of X.
- Each node X has probability distribution P(*X* | *Parents(X)*).

Let's consider an example of a Bayesian network that involves variables that affect whether we get to our appointment on time.



Let's describe this Bayesian network from the top down:

- Rain is the root node in this network. This means that its probability distribution is not reliant on any prior event. In our example, Rain is a random variable that can take the values {*none, light, heavy*} with the following probability distribution:

none	*light*	*heavy*
0.7	0.2	0.1

- Maintenance, in our example, encodes whether there is train track maintenance, taking the values {*yes, no*}. Rain is a parent node of Maintenance, which means that the probability

distribution of Maintenance is affected by Rain.

R	*yes*	*no*
none	0.4	0.6
light	0.2	0.8
heavy	0.1	0.9

- Train is the variable that encodes whether the train is on time or delayed, taking the values {*on time, delayed*}. Note that Train has arrows pointing to it from both Maintenance and Rain. This means that both are parents of Train, and their values affect the probability distribution of Train.

R	**M**	*on time*	*delayed*
none	yes	0.8	0.2
none	no	0.9	0.1
light	yes	0.6	0.4
light	no	0.7	0.3
heavy	yes	0.4	0.6
heavy	no	0.5	0.5

- Appointment is a random variable that represents whether we attend our appointment, taking the values {*attend, miss*}. Note that its only parent is Train. This point about Bayesian network is noteworthy: parents include only direct relations. It is true that maintenance affects whether the train is on time, and whether the train is on time affects whether we attend the appointment. However, in the end, what directly affects our chances of attending the appointment is whether the train came on time, and this is what is represented in the Bayesian network. For example, if the train came on time, it could be heavy rain and track maintenance, but that has no effect over whether we made it to our appointment.

T	*attend*	*miss*
on time	0.9	0.1
delayed	0.6	0.4

For example, if we want to find the probability of missing the meeting when the train was delayed on a day with no maintenance and light rain, or P(*light, no, delayed, miss*), we will compute the following: P(*light*)P(*no* | *light*)P(*delayed* | *light, no*)P(*miss* | *delayed*). The value of each of the individual probabilities can be found in the probability distributions above, and then these values are multiplied to produce P(*no, light, delayed, miss*).

Inference

At the last lecture, we looked at inference through entailment. This means that we could definitively conclude new information based on the information that we already had. We can also infer new information based on probabilities. While this does not allow us to know new information for certain, it allows us to figure out the probability distributions for some values. Inference has multiple properties.

- Query **X**: the variable for which we want to compute the probability distribution.
- Evidence variables **E**: one or more variables that have been observed for event **e**. For example, we might have observed that there is light rain, and this observation helps us compute the probability that the train is delayed.
- Hidden variables **Y**: variables that aren't the query and also haven't been observed. For example, standing at the train station, we can observe whether there is rain, but we can't know if there is maintenance on the track further down the road. Thus, Maintenance would be a hidden variable in this situation.
- The goal: calculate **P**(*X* | *e*). For example, compute the probability distribution of the Train variable (the query) based on the evidence **e** that we know there is light rain.

Let's take an example. We want to compute the probability distribution of the Appointment variable given the evidence that there is light rain and no track maintenance. That is, we know that there is light rain and no track maintenance, and we want to figure out what are the probabilities that we attend the appointment and that we miss the appointment, **P**(*Appointment* | *light, no*). from the joint probability section, we know that we can express the possible values of the Appointment random variable as a proportion, rewriting **P**(*Appointment* | *light, no*) as αP(*Appointment, light, no*). How can we calculate the probability distribution of Appointment if its parent is only the Train variable, and not Rain or Maintenance? Here, we will use marginalization. The value of **P**(*Appointment, light, no*) is equal to α[**P**(*Appointment, light, no, delayed*) + **P**(*Appointment, light, no, on time*)].

Inference by Enumeration

Inference by enumeration is a process of finding the probability distribution of variable X given observed evidence e and some hidden variables Y.

$$\mathbf{P}(X \mid e) = \alpha\, \mathbf{P}(X, e) = \alpha \sum_{y} \mathbf{P}(X, e, y)$$

In this equation, X stand for the query variable, e for the observed evidence, y for all the values of the hidden variables, and α normalizes the result such that we end up with probabilities that add up to 1. To explain the equation in words, it is saying that the probability distribution of X given e is equal to a normalized probability distribution of X and e. To get to this distribution, we sum the normalized probability of X, e, and y, where y takes each time a different value of the hidden variables Y.

Multiple libraries exist in Python to ease the process of probabilistic inference. We will take a look at the library *pomegranate* to see how the above data can be represented in code.

First, we create the nodes and provide a probability distribution for each one.

```
from pomegranate import *

# Rain node has no parents
rain = Node(DiscreteDistribution({
    "none": 0.7,
    "light": 0.2,
    "heavy": 0.1
}), name="rain")

# Track maintenance node is conditional on rain
maintenance = Node(ConditionalProbabilityTable([
    ["none", "yes", 0.4],
    ["none", "no", 0.6],
    ["light", "yes", 0.2],
    ["light", "no", 0.8],
    ["heavy", "yes", 0.1],
    ["heavy", "no", 0.9]
], [rain.distribution]), name="maintenance")

# Train node is conditional on rain and maintenance
train = Node(ConditionalProbabilityTable([
    ["none", "yes", "on time", 0.8],
    ["none", "yes", "delayed", 0.2],
    ["none", "no", "on time", 0.9],
    ["none", "no", "delayed", 0.1],
    ["light", "yes", "on time", 0.6],
    ["light", "yes", "delayed", 0.4],
    ["light", "no", "on time", 0.7],
    ["light", "no", "delayed", 0.3],
    ["heavy", "yes", "on time", 0.4],
    ["heavy", "yes", "delayed", 0.6],
    ["heavy", "no", "on time", 0.5],
```

```
    ["heavy", "no", "delayed", 0.5],
], [rain.distribution, maintenance.distribution]), name="train")

# Appointment node is conditional on train
appointment = Node(ConditionalProbabilityTable([
    ["on time", "attend", 0.9],
    ["on time", "miss", 0.1],
    ["delayed", "attend", 0.6],
    ["delayed", "miss", 0.4]
], [train.distribution]), name="appointment")
```

Second, we create the model by adding all the nodes and then describing which node is the parent of which other node by adding edges between them (recall that a Bayesian network is a directed graph, consisting of nodes with arrows between them).

```
# Create a Bayesian Network and add states
model = BayesianNetwork()
model.add_states(rain, maintenance, train, appointment)

# Add edges connecting nodes
model.add_edge(rain, maintenance)
model.add_edge(rain, train)
model.add_edge(maintenance, train)
model.add_edge(train, appointment)

# Finalize model
model.bake()
```

Now, to ask how probable a certain event is, we run the model with the values we are interested in. In this example, we want to ask what is the probability that there is no rain, no track maintenance, the train is on time, and we attend the meeting.

```
# Calculate probability for a given observation
probability = model.probability([["none", "no", "on time", "attend"]])

print(probability)
```

Otherwise, we could use the program to provide probability distributions for all variables given some observed evidence. In the following case, we know that the train was delayed. Given this information, we compute and print the probability distributions of the variables Rain, Maintenance, and Appointment.

```
# Calculate predictions based on the evidence that the train was delayed
predictions = model.predict_proba({
    "train": "delayed"
})

# Print predictions for each node
for node, prediction in zip(model.states, predictions):
```

```
    if isinstance(prediction, str):
        print(f"{node.name}: {prediction}")
    else:
        print(f"{node.name}")
        for value, probability in prediction.parameters[0].items():
            print(f"    {value}: {probability:.4f}")
```

The code above used inference by enumeration. However, this way of computing probability is inefficient, especially when there are many variables in the model. A different way to go about this would be abandoning **exact inference** in favor of **approximate inference**. Doing this, we lose some precision in the generated probabilities, but often this imprecision is negligible. Instead, we gain a scalable method of calculating probabilities.

Sampling

Sampling is one technique of approximate inference. In sampling, each variable is sampled for a value according to its probability distribution. We will start with an example from outside lecture, and then cover the example from lecture.

> To generate a distribution using sampling with a die, we can roll the die multiple times and record what value we got each time. Suppose we rolled the die 600 times. We count how many times we got 1, which is supposed to be roughly 100, and then repeat for the rest of the values, 2-6. Then, we divide each count by the total number of rolls. This will generate an approximate distribution of the values of rolling a die: on one hand, it is unlikely that we get the result that each value has a probability of 1/6 of occurring (which is the exact probability), but we will get a value that's close to it.

Here is an example from lecture: if we start with sampling the Rain variable, the value *none* will be generated with probability of 0.7, the value *light* will be generated with probability of 0.2, and the value *heavy* will be generated with probability of 0.1. Suppose that the sampled value we get is *none*. When we get to the Maintenance variable, we sample it, too, but only from the probability distribution where Rain is equal to *none*, because this is an already sampled result. We will continue to do so through all the nodes. Now we have one sample, and repeating this process multiple times generates a distribution. Now, if we want to answer a question, such as what is P(*Train = on time*), we can count the number of samples where the variable Train has the value *on time*, and divide the result by the total number of samples. This way, we have just generated an approximate probability for P(*Train = on time*).

We can also answer questions that involve conditional probability, such as P(*Rain = light* | *Train = on time*). In this case, we ignore all samples where the value of Train is not *on time*, and then proceed as before. We count how many samples have the variable Rain = *light* among those samples that have Train = *on time*, and then divide by the total number of samples where Train = *on time*.

In code, a sampling function can look like `generate_sample`:

```
import pomegranate

from collections import Counter

from model import model

def generate_sample():

    # Mapping of random variable name to sample generated
    sample = {}

    # Mapping of distribution to sample generated
    parents = {}

    # Loop over all states, assuming topological order
    for state in model.states:

        # If we have a non-root node, sample conditional on parents
        if isinstance(state.distribution, pomegranate.ConditionalProbabilityTable):
            sample[state.name] = state.distribution.sample(parent_values=parents)

        # Otherwise, just sample from the distribution alone
        else:
            sample[state.name] = state.distribution.sample()

        # Keep track of the sampled value in the parents mapping
        parents[state.distribution] = sample[state.name]

    # Return generated sample
    return sample
```

Now, to compute P(*Appointment* | *Train* = *delayed*), which is the probability distribution of the Appointment variable given that the train is delayed, we do the following:

```
# Rejection sampling
# Compute distribution of Appointment given that train is delayed
N = 10000
data = []

# Repeat sampling 10,000 times
for i in range(N):

    # Generate a sample based on the function that we defined earlier
    sample = generate_sample()

    # If, in this sample, the variable of Train has the value delayed, save the sample
    if sample["train"] == "delayed":
        data.append(sample["appointment"])

# Count how many times each value of the variable appeared. We can later normalize by
```

```
print(Counter(data))
```

Likelihood Weighting

In the sampling example above, we discarded the samples that did not match the evidence that we had. This is inefficient. One way to get around this is with likelihood weighting, using the following steps:

- Start by fixing the values for evidence variables.
- Sample the non-evidence variables using conditional probabilities in the Bayesian network.
- Weight each sample by its **likelihood**: the probability of all the evidence occurring.

For example, if we have the observation that the train was on time, we will start sampling as before. We sample a value of Rain given its probability distribution, then Maintenance, but when we get to Train - we always give it the observed value, in our case, *on time*. Then we proceed and sample Appointment based on its probability distribution given Train = *on time*. Now that this sample exists, we weight it by the conditional probability of the observed variable given its sampled parents. That is, if we sampled Rain and got *light*, and then we sampled Maintenance and got *yes*, then we will weight this sample by P(*Train = on time* | *light, yes*).

Markov Models

So far, we have looked at questions of probability given some information that we observed. In this kind of paradigm, the dimension of time is not represented in any way. However, many tasks do rely on the dimension of time, such as prediction. To represent the variable of time we will create a new variable, X, and change it based on the event of interest, such that X_t is the current event, X_{t+1} is the next event, and so on. To be able to predict events in the future, we will use Markov Models.

The Markov Assumption

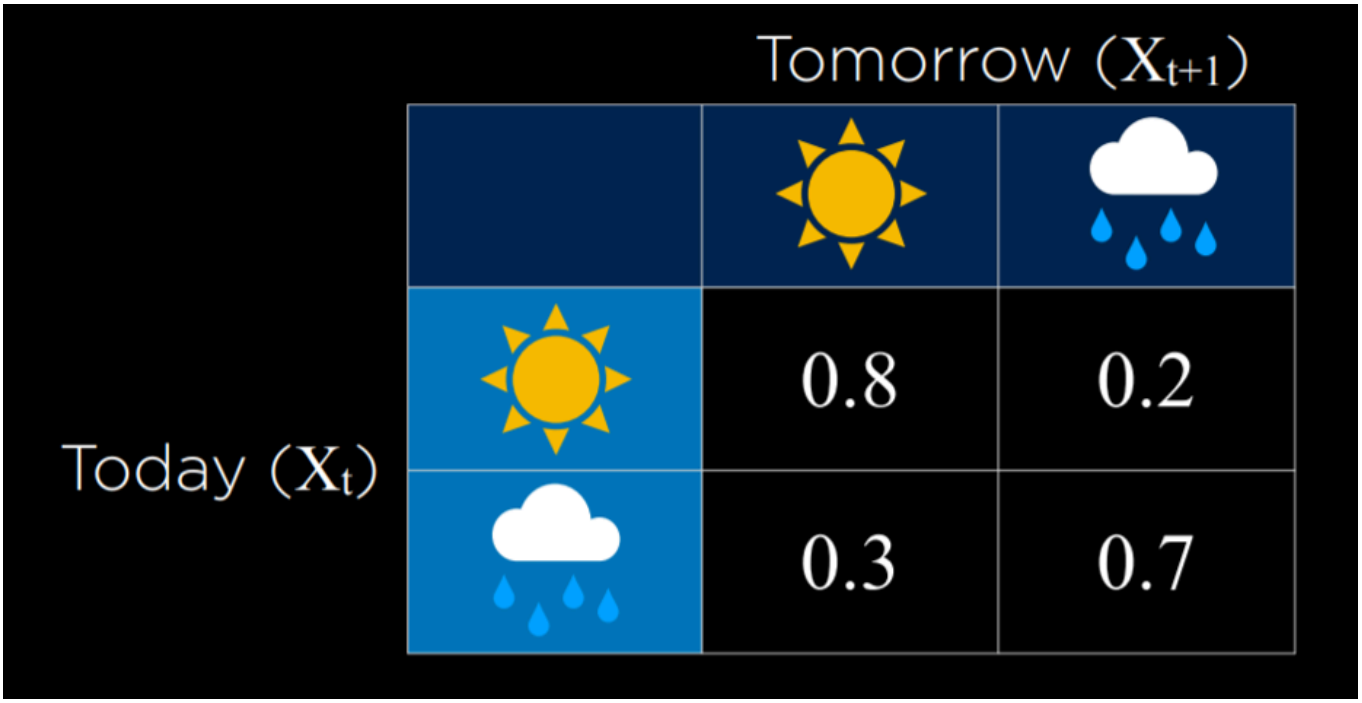
The Markov assumption is an assumption that the current state depends on only a finite fixed number of previous states. This is important to us. Think of the task of predicting weather. In theory, we could use all the data from the past year to predict tomorrow's weather. However, it is infeasible, both because of the computational power this would require and because there is probably no information about the conditional probability of tomorrow's weather based on the weather 365 days ago. Using the Markov assumption, we restrict our previous states (e.g. how many previous days we are going to consider when predicting tomorrow's weather), thereby making the task manageable. This means that we might get a more rough approximation of the probabilities of interest, but this is often good enough for our needs. Moreover, we can use a Markov model based

on the information of the one last event (e.g. predicting tomorrow's weather based on today's weather).

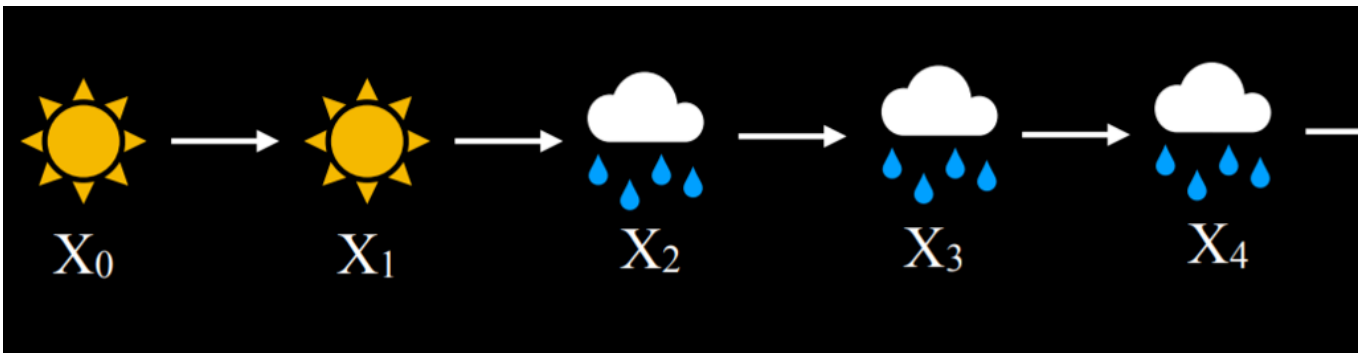
Markov Chain

A Markov chain is a sequence of random variables where the distribution of each variable follows the Markov assumption. That is, each event in the chain occurs based on the probability of the event before it.

To start constructing a Markov chain, we need a **transition model** that will specify the the probability distributions of the next event based on the possible values of the current event.



In this example, the probability of tomorrow being sunny based on today being sunny is 0.8. This is reasonable, because it is more likely than not that a sunny day will follow a sunny day. However, if it is rainy today, the probability of rain tomorrow is 0.7, since rainy days are more likely to follow each other. Using this transition model, it is possible to sample a Markov chain. Start with a day being either rainy or sunny, and then sample the next day based on the probability of it being sunny or rainy given the weather today. Then, condition the probability of the day after tomorrow based on tomorrow, and so on, resulting in a Markov chain:



Given this Markov chain, we can now answer questions such as "what is the probability of having four rainy days in a row?" Here is an example of how a Markov chain can be implemented in code:

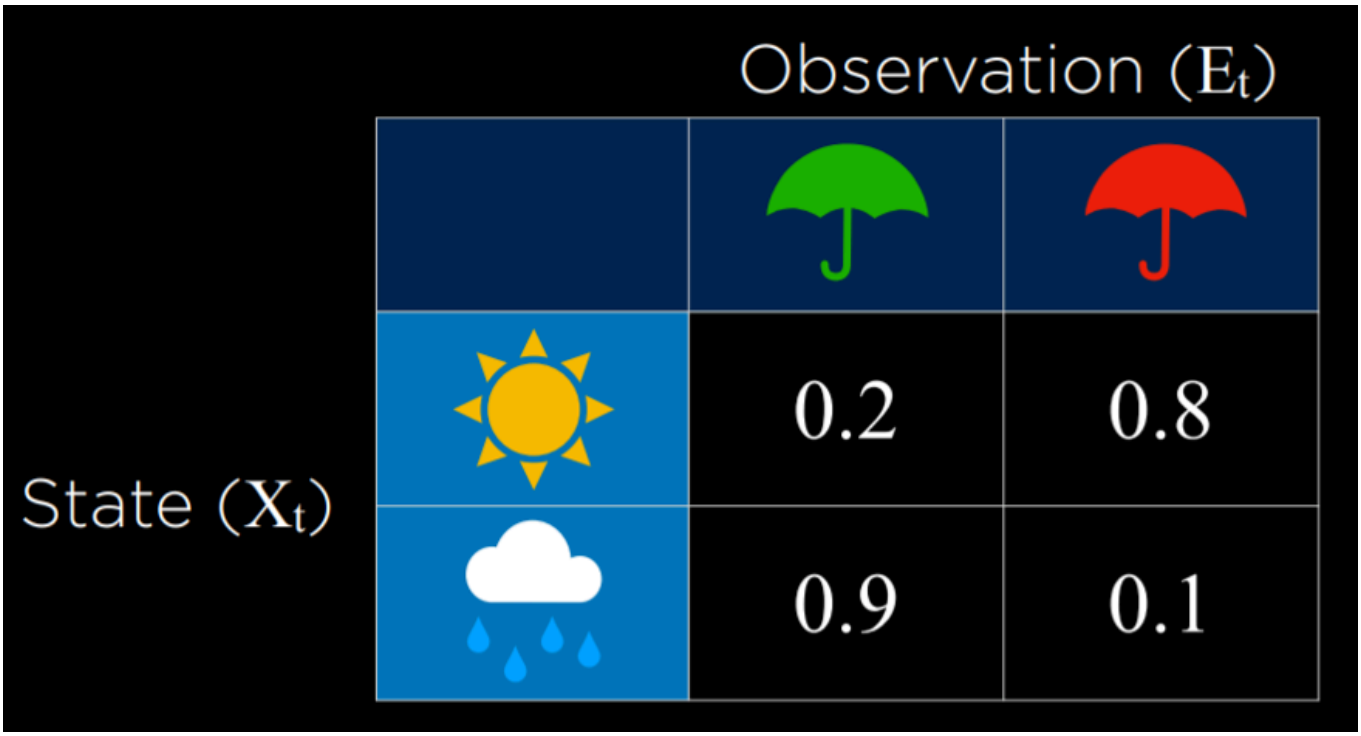
```
from pomegranate import *

# Define starting probabilities
start = DiscreteDistribution({
    "sun": 0.5,
    "rain": 0.5
})

# Define transition model
transitions = ConditionalProbabilityTable([
    ["sun", "sun", 0.8],
    ["sun", "rain", 0.2],
    ["rain", "sun", 0.3],
    ["rain", "rain", 0.7]
], [start])

# Create Markov chain
model = MarkovChain([start, transitions])

# Sample 50 states from chain
print(model.sample(50))
```

Hidden Markov Models

A hidden Markov model is a type of a Markov model for a system with hidden states that generate some observed event. This means that sometimes, the AI has some measurement of the world but no access to the precise state of the world. In these cases, the state of the world is called the **hidden state** and whatever data the AI has access to are the **observations**. Here are a few examples for this:

- For a robot exploring uncharted territory, the hidden state is its position, and the observation is the data recorded by the robot's sensors.
- In speech recognition, the hidden state is the words that were spoken, and the observation is the audio waveforms.
- When measuring user engagement on websites, the hidden state is how engaged the user is, and the observation is the website or app analytics.

For our discussion, we will use the following example. Our AI wants to infer the weather (the hidden state), but it only has access to an indoor camera that records how many people brought umbrellas with them. Here is our **sensor model** (also called **emission model**) that represents these probabilities:

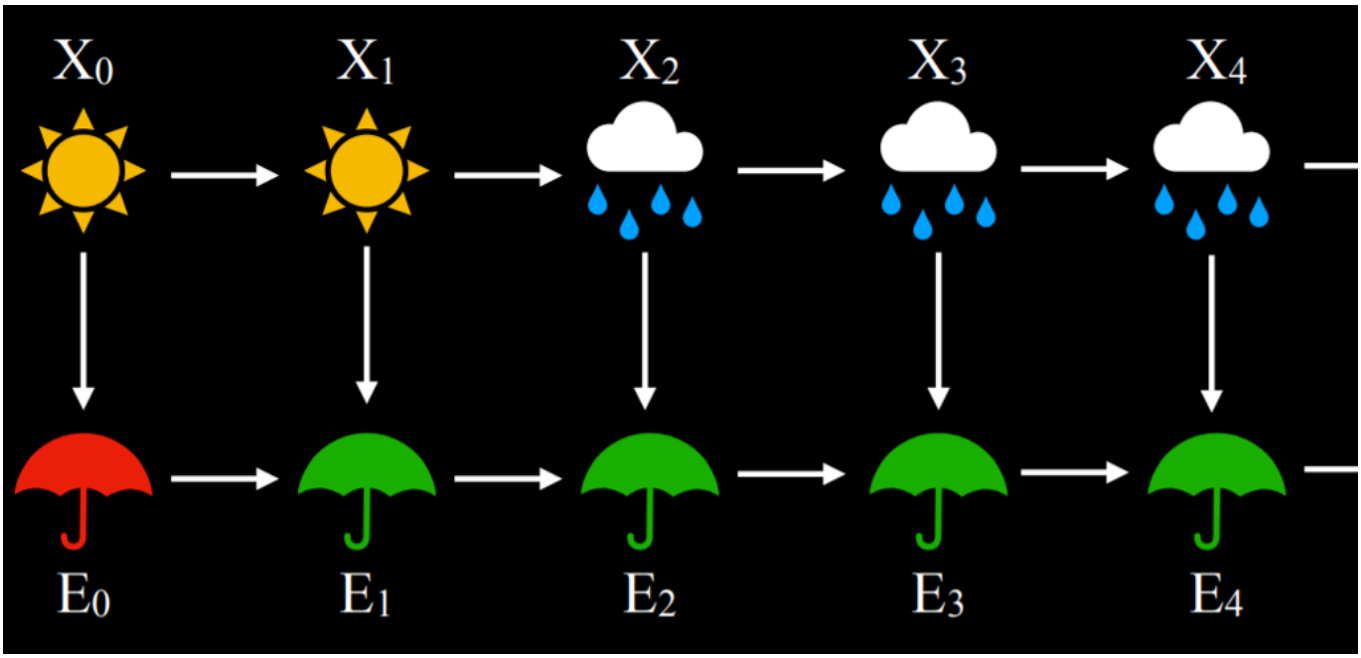


In this model, if it is sunny, it is most probable that people will not bring umbrellas to the building. If it is rainy, then it is very likely that people bring umbrellas to the building. By using the observation of whether people brought an umbrella or not, we can predict with reasonable likelihood what the weather is outside.

Sensor Markov Assumption

The assumption that the evidence variable depends only on the corresponding state. For example, for our models, we assume that whether people bring umbrellas to the office depends only on the weather. This is not necessarily reflective of the complete truth, because, for example, more conscientious, rain-averse people might take an umbrella with them everywhere even when it is sunny, and if we knew everyone's personalities it would add more data to the model. However, the sensor Markov assumption ignores these data, assuming that only the hidden state affects the observation.

A hidden Markov model can be represented in a Markov chain with two layers. The top layer, variable X, stands for the hidden state. The bottom layer, variable E, stands for the evidence, the observations that we have.



Based on hidden Markov models, multiple tasks can be achieved:

- Filtering: given observations from start until now, calculate the probability distribution for the current state. For example, given information on when people bring umbrellas form the start of time until today, we generate a probability distribution for whether it is raining today or not.
- Prediction: given observations from start until now, calculate the probability distribution for a future state.
- Smoothing: given observations from start until now, calculate the probability distribution for a past state. For example, calculating the probability of rain yesterday given that people brought umbrellas today.
- Most likely explanation: given observations from start until now, calculate most likely sequence of events.

The most likely explanation task can be used in processes such as voice recognition, where, based on multiple waveforms, the AI infers the most likely sequence of words or syllables that brought to these waveforms. Next is a Python implementation of a hidden Markov model that we will use for a most likely explanation task:

```
from pomegranate import *

# Observation model for each state
sun = DiscreteDistribution({
    "umbrella": 0.2,
    "no umbrella": 0.8
})

rain = DiscreteDistribution({
    "umbrella": 0.9,
```

```
    "no umbrella": 0.1
})

states = [sun, rain]

# Transition model
transitions = numpy.array(
    [[0.8, 0.2], # Tomorrow's predictions if today = sun
     [0.3, 0.7]] # Tomorrow's predictions if today = rain
)

# Starting probabilities
starts = numpy.array([0.5, 0.5])

# Create the model
model = HiddenMarkovModel.from_matrix(
    transitions, states, starts,
    state_names=["sun", "rain"]
)
model.bake()
```

Note that our model has both the sensor model and the transition model. We need both for the hidden Markov model. In the following code snippet, we see a sequence of observations of whether people brought umbrellas to the building or not, and based on this sequence we will run the model, which will generate and print the most likely explanation (i.e. the weather sequence that most likely brought to this pattern of observations):

```
from model import model

# Observed data
observations = [
    "umbrella",
    "umbrella",
    "no umbrella",
    "umbrella",
    "umbrella",
    "umbrella",
    "umbrella",
    "no umbrella",
    "no umbrella"
]

# Predict underlying states
predictions = model.predict(observations)
for prediction in predictions:
    print(model.states[prediction].name)
```

In this case, the output of the program will be rain, rain, sun, rain, rain, rain, rain, sun, sun. This output represents what is the most likely pattern of weather given our observations of people bringing or not bringing umbrellas to the building.

CS50's Introduction to Artificial Intelligence with Python

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

Brian Yu (https://brianyu.me)
brian@cs.harvard.edu

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://www.reddit.com/user/davidjmalan) (https://www.threads.net/@davidjmalan)
(https://twitter.com/davidjmalan)

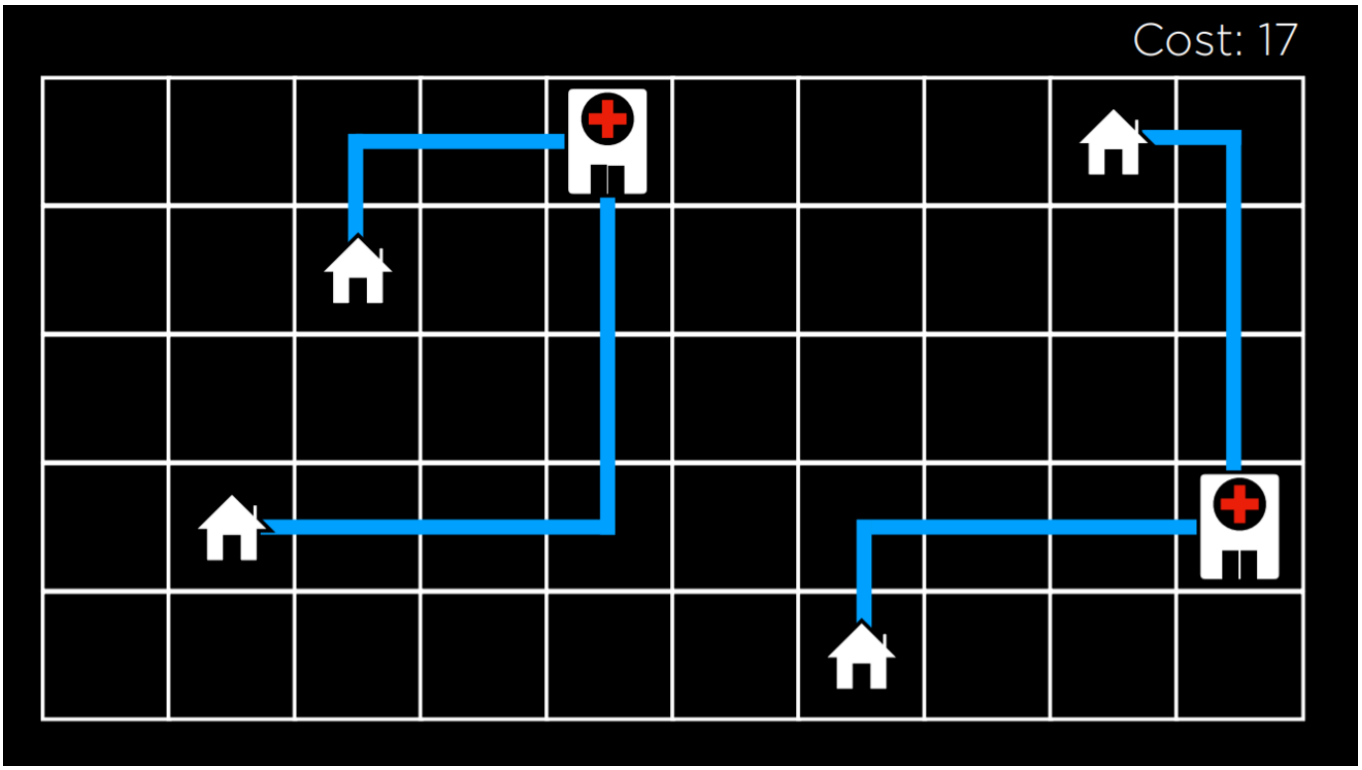
Lecture 3

Optimization

Optimization is choosing the best option from a set of possible options. We have already encountered problems where we tried to find the best possible option, such as in the minimax algorithm, and today we will learn about tools that we can use to solve an even broader range of problems.

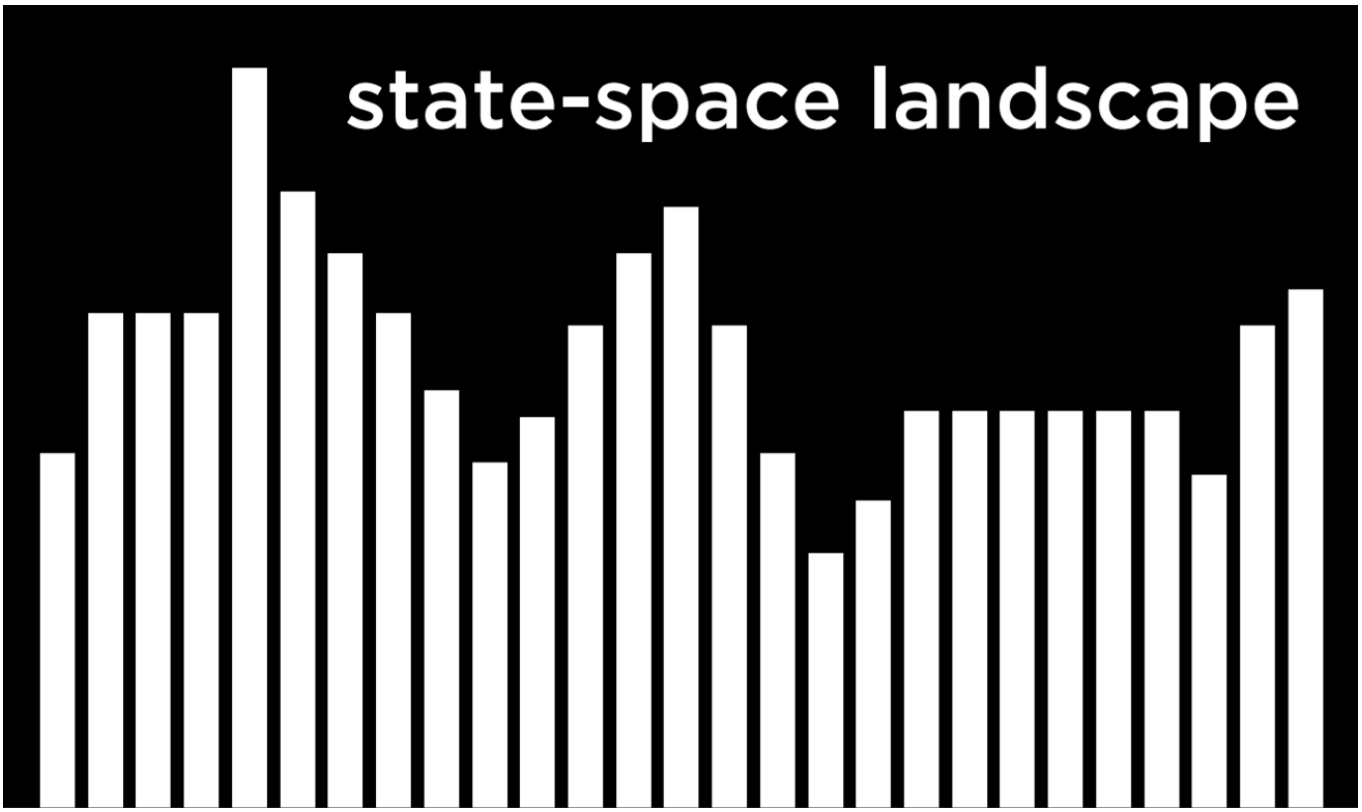
Local Search

Local search is a search algorithm that maintains a single node and searches by moving to a neighboring node. This type of algorithm is different from previous types of search that we saw. Whereas in maze solving, for example, we wanted to find the quickest way to the goal, local search is interested in finding the best answer to a question. Often, local search will bring to an answer that is not optimal but "good enough," conserving computational power. Consider the following example of a local search problem: we have four houses in set locations. We want to build two hospitals, such that we minimize the distance from each house to a hospital. This problem can be visualized as follows:



In this illustration, we are seeing a possible configuration of houses and hospitals. The distance between them is measured using Manhattan distance (number of moves up, down, and to the sides; discussed in more detail in lecture 0), and the sum of the distances from each house to the nearest hospital is 17. We call this the **cost**, because we try to minimize this distance. In this case, a state would be any one configuration of houses and hospitals.

Abstracting this concept, we can represent each configuration of houses and hospitals as the state-space landscape below. Each of the bars in the picture represents a value of a state, which in our example would be the cost of a certain configuration of houses and hospitals.



Going off of this visualization, we can define a few important terms for the rest of our discussion:

- An **Objective Function** is a function that we use to maximize the value of the solution.
- A **Cost Function** is a function that we use to minimize the cost of the solution (this is the function that we would use in our example with houses and hospitals. We want to minimize the distance from houses to hospitals).
- A **Current State** is the state that is currently being considered by the function.
- A **Neighbor State** is a state that the current state can transition to. In the one-dimensional state-space landscape above, a neighbor state is the state to either side of the current state. In our example, a neighbor state could be the state resulting from moving one of the hospitals to any direction by one step. Neighbor states are usually similar to the current state, and, therefore, their values are close to the value of the current state.

Note that the way local search algorithms work is by considering one node in a current state, and then moving the node to one of the current state's neighbors. This is unlike the minimax algorithm, for example, where every single state in the state space was considered recursively.

Hill Climbing

Hill climbing is one type of a local search algorithm. In this algorithm, the neighbor states are compared to the current state, and if any of them is better, we change the current node from the current state to that neighbor state. What qualifies as better is defined by whether we use an objective function, preferring a higher value, or a decreasing function, preferring a lower value.

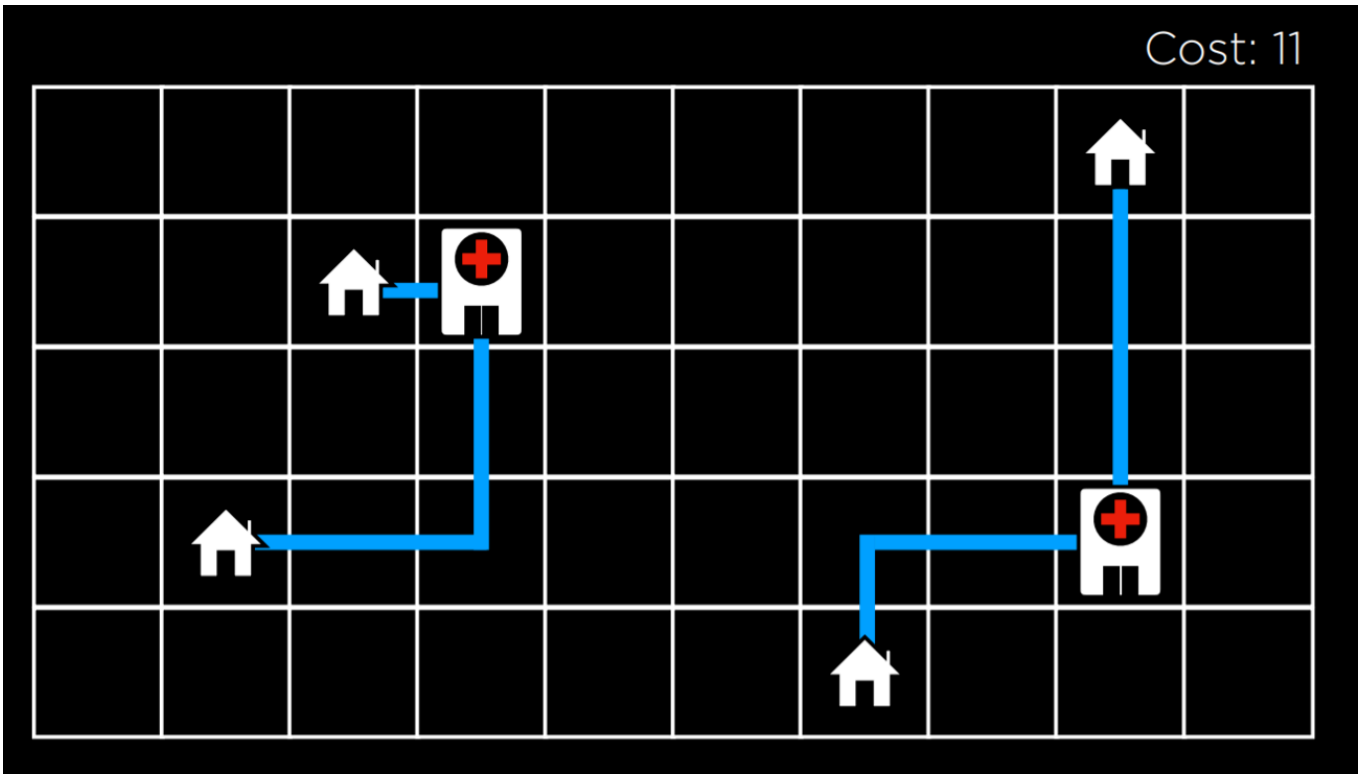
A hill climbing algorithm will look the following way in pseudocode:

function Hill-Climb(*problem*):

- *current* = initial state of *problem*
- repeat:
  - *neighbor* = best valued neighbor of *current*
  - if *neighbor* not better than *current*:
    - return *current*
  - *current* = *neighbor*

In this algorithm, we start with a current state. In some problems, we will know what the current state is, while, in others, we will have to start with selecting one randomly. Then, we repeat the following actions: we evaluate the neighbors, selecting the one with the best value. Then, we compare this neighbor's value to the current state's value. If the neighbor is better, we switch the current state to the neighbor state, and then repeat the process. The process ends when we compare the best neighbor to the current state, and the current state is better. Then, we return the current state.

Using the hill climbing algorithm, we can start to improve the locations that we assigned to the hospitals in our example. After a few transitions, we get to the following state:

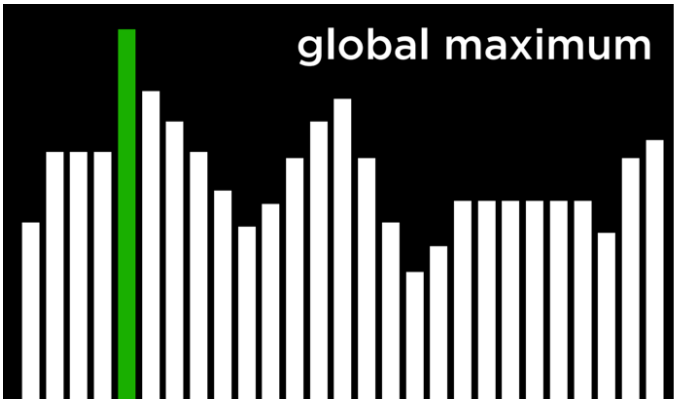


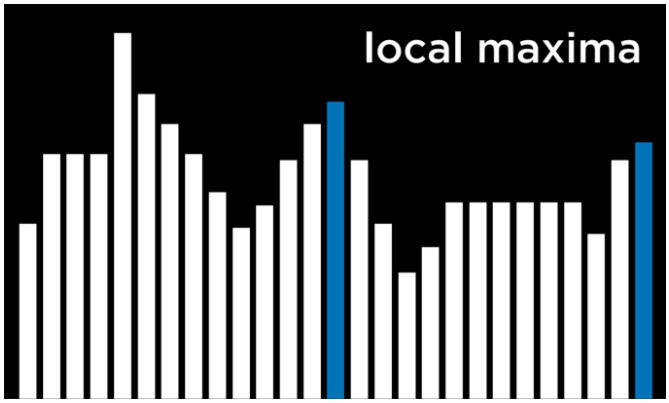
At this state, the cost is 11, which is an improvement over 17, the cost of the initial state. However, this is not the optimal state just yet. For example, moving the hospital on the left to be underneath the top left house would bring to a cost of 9, which is better than 11. However, this version of a hill

climbing algorithm can't get there, because all the neighbor states are at least as costly as the current state. In this sense, a hill climbing algorithm is short-sighted, often settling for solutions that are *better* than some others, but not necessarily the *best* of all possible solutions.

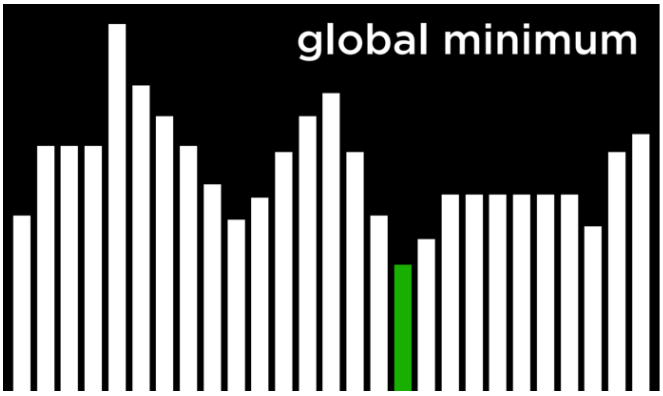
Local and Global Minima and Maxima

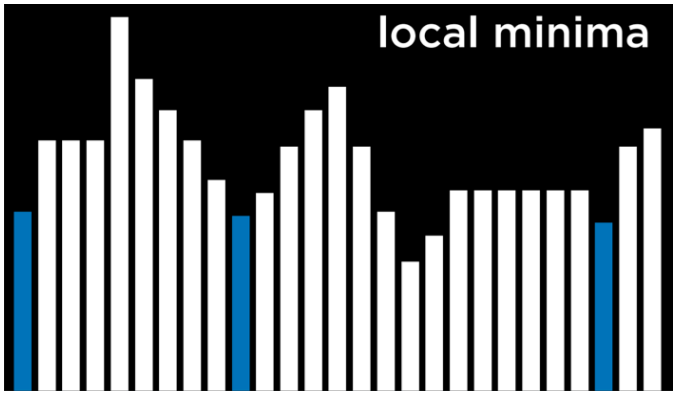
As mentioned above, a hill climbing algorithm can get stuck in local maxima or minima. A ***local* maximum** (plural: maxima) is a state that has a higher value than its *neighboring states*. As opposed to that, a ***global* maximum** is a state that has the highest value of *all states* in the state-space.



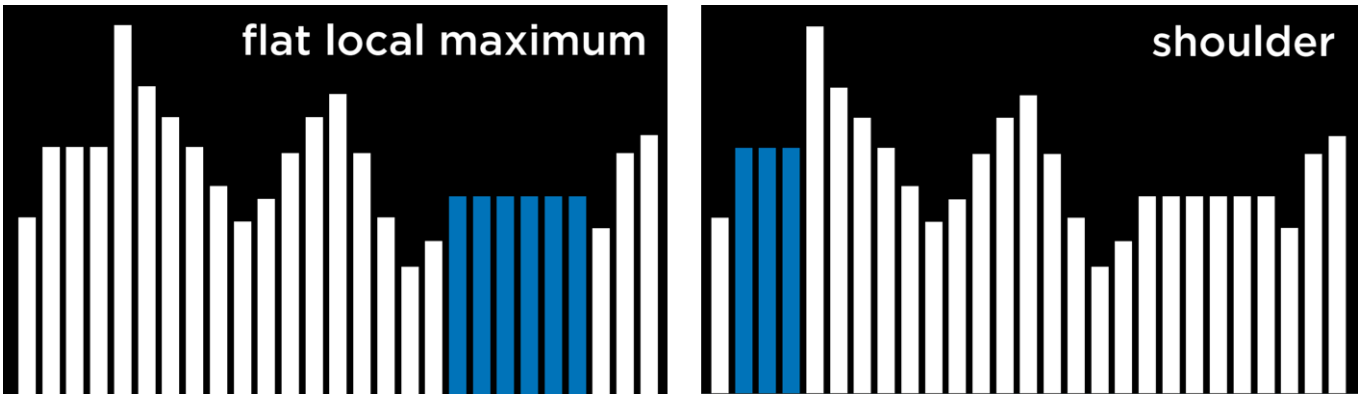


In contrast, a ***local* minimum** (plural: minima) is a state that has a lower value than its *neighboring states*. As opposed to that, a ***global* minimum** is a state that has the lowest value of *all states* in the state-space.





The problem with hill climbing algorithms is that they may end up in local minima and maxima. Once the algorithm reaches a point whose neighbors are worse, for the function's purpose, than the current state, the algorithm stops. Special types of local maxima and minima include the **flat local maximum/minimum**, where multiple states of equal value are adjacent, forming a plateau whose neighbors have a worse value, and the **shoulder**, where multiple states of equal value are adjacent and the neighbors of the plateau can be both better and worse. Starting from the middle of the plateau, the algorithm will not be able to advance in any direction.



Hill Climbing Variants

Due to the limitations of Hill Climbing, multiple variants have been thought of to overcome the problem of being stuck in local minima and maxima. What all variations of the algorithm have in common is that, no matter the strategy, each one still has the potential of ending up in local minima and maxima and no means to continue optimizing. The algorithms below are phrased such that a higher value is better, but they also apply to cost functions, where the goal is to minimize cost.

- **Steepest-ascent**: choose the highest-valued neighbor. This is the standard variation that we discussed above.
- **Stochastic**: choose randomly from higher-valued neighbors. Doing this, we choose to go to any direction that improves over our value. This makes sense if, for example, the highest-valued neighbor leads to a local maximum while another neighbor leads to a global maximum.
- **First-choice**: choose the first higher-valued neighbor.
- **Random-restart**: conduct hill climbing multiple times. Each time, start from a random state. Compare the maxima from every trial, and choose the highest amongst those.
- **Local Beam Search**: chooses the *k* highest-valued neighbors. This is unlike most local search algorithms in that it uses multiple nodes for the search, and not just one.

Although local search algorithms don't always give the best possible solution, they can often give a good enough solution in situations where considering every possible state is computationally infeasible.

Simulated Annealing

Although we have seen variants that can improve hill climbing, they all share the same fault: once the algorithm reaches a local maximum, it stops running. Simulated annealing allows the algorithm to "dislodge" itself if it gets stuck in a local maximum.

Annealing is the process of heating metal and allowing it to cool slowly, which serves to toughen the metal. This is used as a metaphor for the simulated annealing algorithm, which starts with a high temperature, being more likely to make random decisions, and, as the temperature decreases, it becomes less likely to make random decisions, becoming more "firm." This mechanism allows the algorithm to change its state to a neighbor that's worse than the current state, which is how it can escape from local maxima. The following is pseudocode for simulated annealing:

function Simulated-Annealing(*problem*, *max*):

- *current* = initial state of *problem*
- for *t* = 1 to *max*:
    - *T* = Temperature(*t*)
    - *neighbor* = random neighbor of *current*
    - ΔE = how much better *neighbor* is than *current*
    - if $\Delta E > 0$:
        - *current* = *neighbor*
    - with probability e^($\Delta E/T$) set *current* = *neighbor*
- return *current*

The algorithm takes as input a problem and *max*, the number of times it should repeat itself. For each iteration, *T* is set using a Temperature function. This function return a higher value in the early iterations (when *t* is low) and a lower value in later iterations (when *t* is high). Then, a random neighbor is selected, and ΔE is computed such that it quantifies how better the neighbor state is than the current state. If the neighbor state is better than the current state ($\Delta E > 0$), as before, we set our current state to be the neighbor state. However, when the neighbor state is worse ($\Delta E < 0$), we still might set our current state to be that neighbor state, and we do so with probability e^($\Delta E/T$). The idea here is that a more negative ΔE will result in lower probability of the neighbor state being chosen, and the higher the temperature *T* the higher the probability that the neighbor state will be chosen. This means that the worse the neighbor state, the less likely it is to be chosen, and the earlier the algorithm is in its process, the more likely it is to set a worse neighbor state as current state. The math behind this is as follows: e is a constant (around 2.72), and ΔE is negative (since this neighbor is worse than the current state). The more negative ΔE, the closer the resulting value to 0. The higher the temperature *T* is, the closer $\Delta E/T$ is to 0, making the probability closer to 1.

Traveling Salesman Problem

In the traveling salesman problem, the task is to connect all points while choosing the shortest possible distance. This is, for example, what delivery companies need to do: find the shortest route from the store to all the customers' houses and back.

In this case, a neighbor state might be seen as a state where two arrows swap places. Calculating every possible combination makes this problem computationally demanding (having just 10 points gives us 10!, or 3,628,800 possible routes). By using the simulated annealing algorithm, a good solution can be found for a lower computational cost.

Linear Programming

Linear programming is a family of problems that optimize a linear equation (an equation of the form $y = ax_1 + bx_2 + ...$).

Linear programming will have the following components:

- A cost function that we want to minimize: $c_1x_1 + c_2x_2 + ... + c_nx_n$. Here, each x_- is a variable and it is associated with some cost c_-.
- A constraint that's represented as a sum of variables that is either less than or equal to a value ($a_1x_1 + a_2x_2 + ... + a_nx_n \leq b$) or precisely equal to this value ($a_1x_1 + a_2x_2 + ... + a_nx_n = b$). In this case, x_- is a variable, and a_- is some resource associated with it, and b is how much resources we can dedicate to this problem.
- Individual bounds on variables (for example, that a variable can't be negative) of the form $l_i \leq x_i \leq u_i$.

Consider the following example:

- Two machines, X_1 and X_2. X_1 costs \$50/hour to run, X_2 costs \$80/hour to run. The goal is to minimize cost. This can be formalized as a cost function: $50x_1 + 80x_2$.
- X_1 requires 5 units of labor per hour. X_2 requires 2 units of labor per hour. Total of 20 units of labor to spend. This can be formalized as a constraint: $5x_1 + 2x_2 \leq 20$.
- X_1 produces 10 units of output per hour. X_2 produces 12 units of output per hour. Company needs 90 units of output. This is another constraint. Literally, it can be rewritten as $10x_1 + 12x_2 \geq 90$. However, constraints need to be of the form ($a_1x_1 + a_2x_2 + \ldots + a_nx_n \leq b$) or ($a_1x_1 + a_2x_2 + \ldots + a_nx_n = b$). Therefore, we multiply by (-1) to get to an equivalent equation of the desired form: $(-10x_1) + (-12x_2) \leq -90$.

An optimizing algorithm for linear programming requires background knowledge in geometry and linear algebra that we don't want to assume. Instead, we can use algorithms that already exist, such as Simplex and Interior-Point.

The following is a linear programming example that uses the scipy library in Python:

```
import scipy.optimize

# Objective Function: 50x_1 + 80x_2
# Constraint 1: 5x_1 + 2x_2 <= 20
# Constraint 2: -10x_1 + -12x_2 <= -90

result = scipy.optimize.linprog(
    [50, 80],  # Cost function: 50x_1 + 80x_2
    A_ub=[[5, 2], [-10, -12]],  # Coefficients for inequalities
    b_ub=[20, -90],  # Constraints for inequalities: 20 and -90
)

if result.success:
    print(f"X1: {round(result.x[0], 2)} hours")
    print(f"X2: {round(result.x[1], 2)} hours")
else:
    print("No solution")
```

Constraint Satisfaction

Constraint Satisfaction problems are a class of problems where variables need to be assigned values while satisfying some conditions.

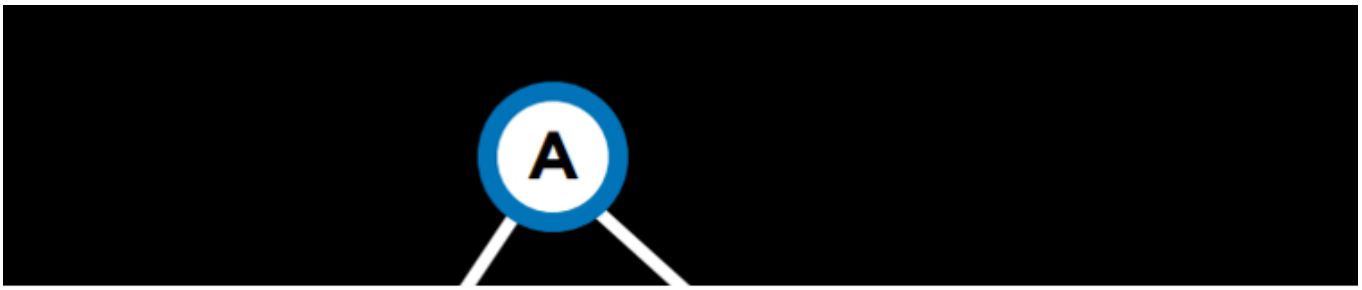
Constraints satisfaction problems have the following properties:

- Set of variables ($x_1, x_2, \ldots, x_n$)
- Set of domains for each variable $\{D_1, D_2, \ldots, D_n\}$
- Set of constraints C

Sudoku can be represented as a constraint satisfaction problem, where each empty square is a variable, the domain is the numbers 1-9, and the constraints are the squares that can't be equal to each other.

Consider another example. Each of students 1-4 is taking three courses from A, B, …, G. Each course needs to have an exam, and the possible days for exams are Monday, Tuesday, and Wednesday. However, the same student can't have two exams on the same day. In this case, the variables are the courses, the domain is the days, and the constraints are which courses can't be scheduled to have an exam on the same day because the same student is taking them. This can be visualized as follows:

This problem can be solved using constraints that are represented as a graph. Each node on the graph is a course, and an edge is drawn between two courses if they can't be scheduled on the same day. In this case, the graph will look this:



A few more terms worth knowing about constraint satisfaction problems:

- A **Hard Constraint** is a constraint that must be satisfied in a correct solution.
- A **Soft Constraint** is a constraint that expresses which solution is preferred over others.
- A **Unary Constraint** is a constraint that involves only one variable. In our example, a unary constraint would be saying that course A can't have an exam on Monday {$A \neq Monday$}.
- A **Binary Constraint** is a constraint that involves two variables. This is the type of constraint that we used in the example above, saying that some two courses can't have the same value {$A \neq B$}.

Node Consistency

Node consistency is when all the values in a variable's domain satisfy the variable's unary constraints.

For example, let's take two courses, A and B. The domain for each course is {*Monday, Tuesday, Wednesday*}, and the constraints are {$A \neq Mon, B \neq Tue, B \neq Mon, A \neq B$}. Now, neither A nor B is consistent, because the existing constraints prevent them from being able to take every value that's in their domain. However, if we remove Monday from A's domain, then it will have node consistency. To achieve node consistency in B, we will have to remove both Monday and Tuesday from its domain.

Arc Consistency

Arc consistency is when all the values in a variable's domain satisfy the variable's binary constraints (note that we are now using "arc" to refer to what we previously referred to as "edge"). In other words, to make X arc-consistent with respect to Y, remove elements from X's domain until every choice for X has a possible choice for Y.

Consider our previous example with the revised domains: A:{*Tuesday, Wednesday*} and B: {*Wednesday*}. For A to be arc-consistent with B, no matter what day A's exam gets scheduled (from its domain), B will still be able to schedule an exam. Is A arc-consistent with B? If A takes the value Tuesday, then B can take the value Wednesday. However, if A takes the value Wednesday, then there is no value that B can take (remember that one of the constraints is $A \neq B$). Therefore, A is not arc-consistent with B. To change this, we can remove Wednesday from A's domain. Then, any value that A takes (Tuesday being the only option) leaves a value for B to take (Wednesday). Now, A is arc-consistent with B. Let's look at an algorithm in pseudocode that makes a variable arc-consistent with respect to some other variable (note that csp stands for "constraint satisfaction problem").

function Revise(*csp, X, Y*):

- *revised = false*
- for *x* in *X.domain*:
    - if no *y* in *Y.domain* satisfies constraint for (*X,Y*):
        - delete *x* from *X.domain*
        - *revised* = true
- return *revised*

This algorithm starts with tracking whether any change was made to X's domain, using the variable *revised*. This will be useful in the next algorithm we examine. Then, the code repeats for every value in X's domain and sees if Y has a value that satisfies the constraints. If yes, then do nothing, if not, remove this value from X's domain.

Often we are interested in making the whole problem arc-consistent and not just one variable with respect to another. In this case, we will use an algorithm called AC-3, which uses Revise:

function AC-3(*csp*):

- *queue* = all arcs in *csp*
- while *queue* non-empty:
    - (*X, Y*) = Dequeue(*queue*)
    - if Revise(*csp, X, Y*):
        - if size of *X*.domain == 0:
            - return *false*
        - for each *Z* in *X*.neighbors - {*Y*}:
            - Enqueue(queue, (*Z,X*))
- return true

This algorithm adds all the arcs in the problem to a queue. Each time it considers an arc, it removes it from the queue. Then, it runs the Revise algorithm to see if this arc is consistent. If changes were made to make it consistent, further actions are needed. If the resulting domain of X is empty, it means that this constraint satisfaction problem is unsolvable (since there are no values that X can take that will allow Y to take any value given the constraints). If the problem is not deemed unsolvable in the previous step, then, since X's domain was changed, we need to see if all the arcs associated with X are still consistent. That is, we take all of X's neighbors except Y, and we add the arcs between them and X to the queue. However, if the Revise algorithm returns false, meaning that the domain wasn't changed, we simply continue considering the other arcs.

While the algorithm for arc consistency can simplify the problem, it will not necessarily solve it, since it considers binary constraints only and not how multiple nodes might be interconnected. Our previous example, where each of 4 students is taking 3 courses, remains unchanged by running AC-3 on it.

We have encountered search problems in our first lecture. A constraint satisfaction problem can be seen as a search problem:

- Initial state: empty assignment (all variables don't have any values assigned to them).
- Actions: add a {*variable* = *value*} to assignment; that is, give some variable a value.
- Transition model: shows how adding the assignment changes the assignment. There is not much depth to this: the transition model returns the state that includes the assignment following the latest action.
- Goal test: check if all variables are assigned a value and all constraints are satisfied.

- Path cost function: all paths have the same cost. As we mentioned earlier, as opposed to typical search problems, optimization problems care about the solution and not the route to the solution.

However, going about a constraint satisfaction problem naively, as a regular search problem, is massively inefficient. Instead, we can make use of the structure of a constraint satisfaction problem to solve it more efficiently.

Backtracking Search

Backtracking search is a type of a search algorithm that takes into account the structure of a constraint satisfaction search problem. In general, it is a recursive function that attempts to continue assigning values as long as they satisfy the constraints. If constraints are violated, it tries a different assignment. Let's look at the pseudocode for it:

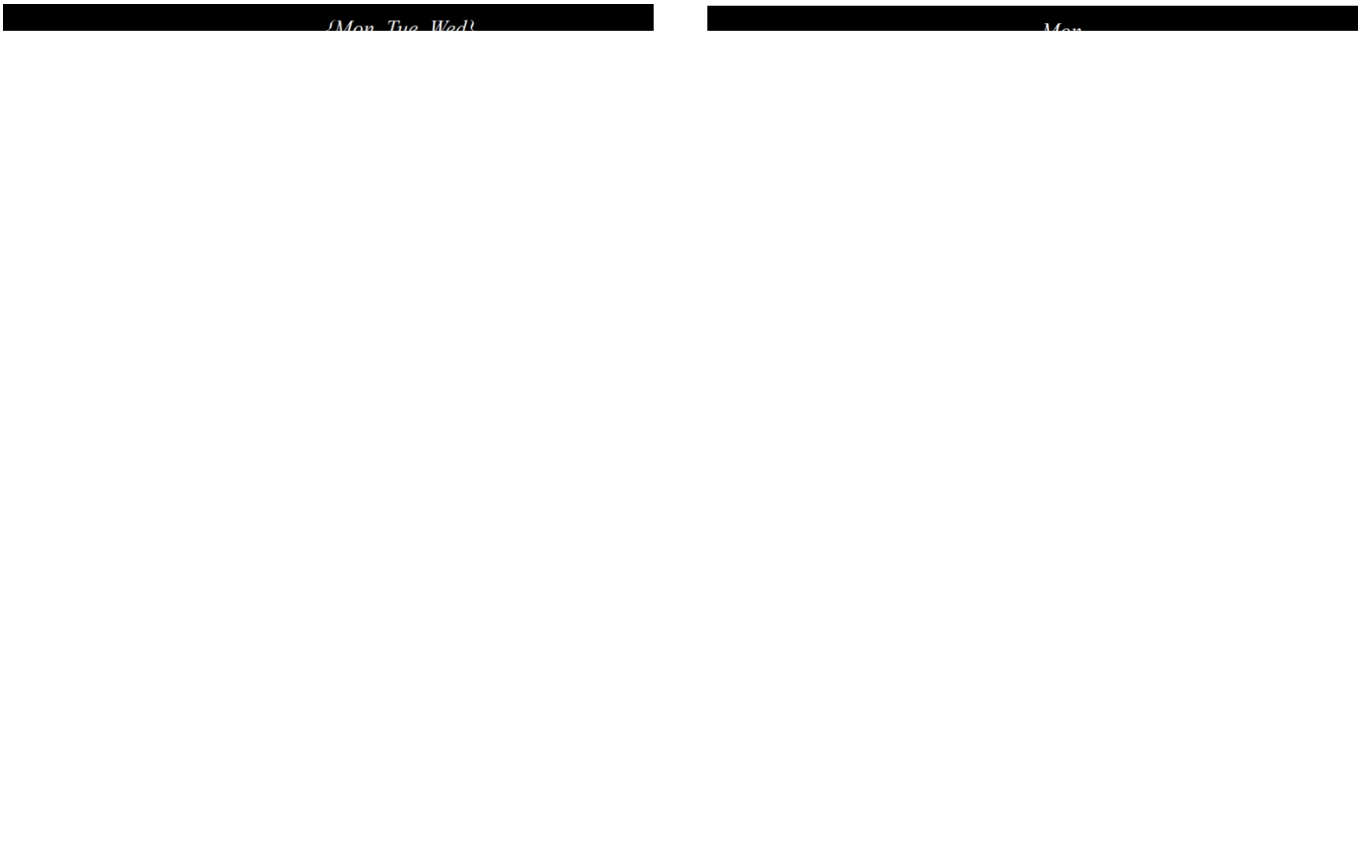
function Backtrack(*assignment, csp*):

- if *assignment* complete:
    - return *assignment*
- *var* = Select-Unassigned-Var(*assignment, csp*)
- for *value* in Domain-Values(*var, assignment, csp*):
    - if *value* consistent with *assignment*:
        - add {*var* = *value*} to *assignment*
        - *result* = Backtrack(*assignment, csp*)
        - if *result* $\neq$ *failure*:
            - return *result*
        - *remove* {*var* = *value*} from *assignment*
- return failure

In words, this algorithm starts with returning the current assignment if it is complete. This means that, if the algorithm is done, it will not perform any of the additional actions. Instead, it will just return the completed assignment. If the assignment is not complete, the algorithm selects any of the variables that do not have an assignment yet. Then, the algorithm tries to assign a value to the variable, and runs the Backtrack algorithm again on the resulting assignment (recursion). Then, it checks the resulting value. If it is not *failure*, it means that the assignment worked out, and it should return this assignment. If the resulting value is *failure*, then the latest assignment is removed, and a new possible value is attempted, repeating the same process. If all possible values in the domain returned *failure*, this means that we need to backtrack. That is, that the problem is

with some previous assignment. If this happens with the variable we start with, then it means that no solution satisfies the constraints.

Consider the following course of action:



We start with empty assignments (top left). Then, we choose the variable A, and assign to it some value, Monday (top right). Then, using this assignment, we run the algorithm again. Now that A already has an assignment, the algorithm will consider B, and assign Monday to it (bottom left). This assignment returns false, so instead of assigning a value to C given the previous assignment, the algorithm will try to assign a new value to B, Tuesday (bottom right). This new assignment satisfies the constraints, and a new variable will be considered next given this assignment. If, for example, assigning also Tuesday or Wednesday to B would bring to a failure, then the algorithm would backtrack and return to considering A, assigning another value to it, Tuesday. If also Tuesday and Wednesday return *failure*, then it means we have tried every possible assignment and the problem is unsolvable.

In the source code section, you can find an implementation from scratch of the backtrack algorithm. However, this algorithm is widely used, and, as such, multiple libraries already contain an implementation of it.

Inference

Although backtracking search is more efficient than simple search, it still takes a lot of computational power. Enforcing arc consistency, on the other hand, is less resource intensive. By

interleaving backtracking search with inference (enforcing arc consistency), we can get at a more efficient algorithm. This algorithm is called the **Maintaining Arc-Consistency** algorithm. This algorithm will enforce arc-consistency after every new assignment of the backtracking search. Specifically, after we make a new assignment to X, we will call the AC-3 algorithm and start it with a queue of all arcs (*Y,X*) where Y is a neighbor of X (and not a queue of all arcs in the problem). Following is a revised Backtrack algorithm that maintains arc-consistency, with the new additions in **bold**.

function Backtrack(*assignment, csp*):

- if *assignment* complete:
  - return *assignment*
- *var* = Select-Unassigned-Var(*assignment, csp*)
- for *value* in Domain-Values(*var, assignment, csp*):
  - if *value* consistent with *assignment*:
    - **add {*var = value*} to *assignment***
    - ***inferences* = Inference(*assignment, csp*)**
    - if *inferences* ≠ *failure*:
      - add *inferences* to *assignment*
    - *result* = Backtrack(*assignment, csp*)
    - if *result* ≠ *failure*:
      - return *result*
    - *remove* {*var = value*} **and *inferences*** from *assignment*
- return failure

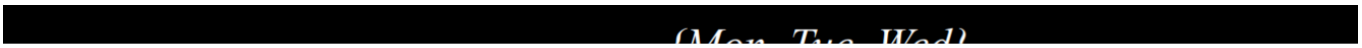
The Inference function runs the AC-3 algorithm as described. Its output is all the inferences that can be made through enforcing arc-consistency. Literally, these are the new assignments that can be deduced from the previous assignments and the structure of the constrain satisfaction problem.

There are additional ways to make the algorithm more efficient. So far, we selected an unassigned variable randomly. However, some choices are more likely to bring to a solution faster than others. This requires the use of heuristics. A heuristic is a rule of thumb, meaning that, more often than not, it will bring to a better result than following a naive approach, but it is not guaranteed to do so.

Minimum Remaining Values (MRV) is one such heuristic. The idea here is that if a variable's domain was constricted by inference, and now it has only one value left (or even if it's two values), then by making this assignment we will reduce the number of backtracks we might need to do later. That is, we will have to make this assignment sooner or later, since it's inferred from enforcing arc-consistency. If this assignment brings to failure, it is better to find out about it as soon as possible and not backtrack later.

For example, after having narrowed down the domains of variables given the current assignment, using the MRV heuristic, we will choose variable C next and assign the value Wednesday to it.

The **Degree** heuristic relies on the degrees of variables, where a degree is how many arcs connect a variable to other variables. By choosing the variable with the highest degree, with one assignment, we constrain multiple other variables, speeding the algorithm's process.



For example, all the variables above have domains of the same size. Thus, we should pick a domain with the highest degree, which would be variable E.

Both heuristics are not always applicable. For example, when multiple variables have the same least number of values in their domain, or when multiple variables have the same highest degree.

Another way to make the algorithm more efficient is employing yet another heuristic when we select a value from the domain of a variable. Here, we would like to use the **Least Constraining Values** heuristic, where we select the value that will constrain the least other variables. The idea here is that, while in the degree heuristic we wanted to use the variable that is more likely to constrain other variables, here we want this variable to place the least constraints on other variables. That is, we want to locate what could be the largest potential source of trouble (the variable with the highest degree), and then render it the least troublesome that we can (assign the least constraining value to it).

For example, let's consider variable C. If we assign Tuesday to it, we will put a constraint on all of B, E, and F. However, if we choose Wednesday, we will put a constraint only on B and E. Therefore, it is probably better to go with Wednesday.

To summarize, optimization problems can be formulated in multiple ways. Today we considered local search, linear programming, and constraint satisfaction.

CS50's Introduction to Artificial Intelligence with Python

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

Brian Yu (https://brianyu.me)
brian@cs.harvard.edu

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://www.reddit.com/user/davidjmalan) (https://www.threads.net/@davidjmalan)
(https://twitter.com/davidjmalan)

Lecture 4

Machine Learning

Machine learning provides a computer with data, rather than explicit instructions. Using these data, the computer learns to recognize patterns and becomes able to execute tasks on its own.

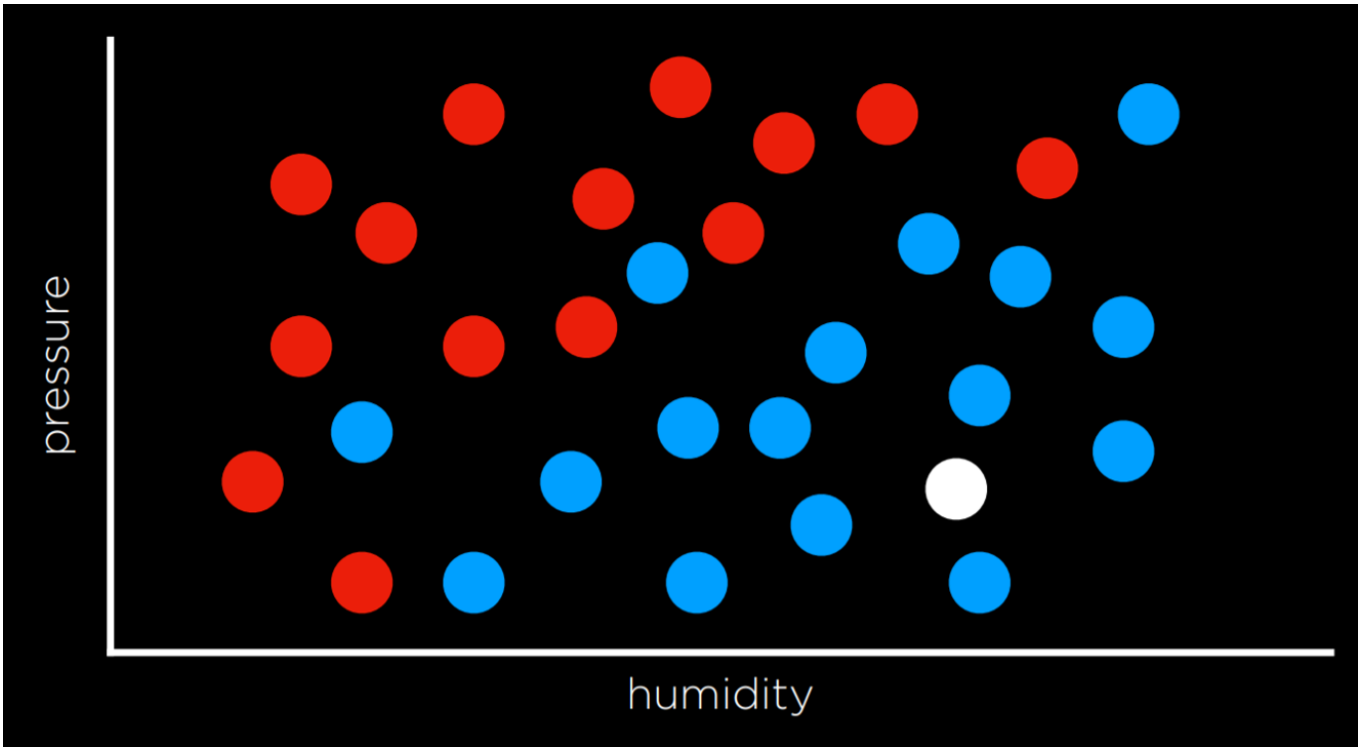
Supervised Learning

Supervised learning is a task where a computer learns a function that maps inputs to outputs based on a dataset of input-output pairs.

There are multiple tasks under supervised learning, and one of those is **Classification**. This is a task where the function maps an input to a discrete output. For example, given some information on humidity and air pressure for a particular day (input), the computer decides whether it will rain that day or not (output). The computer does this after training on a dataset with multiple days where humidity and air pressure are already mapped to whether it rained or not.

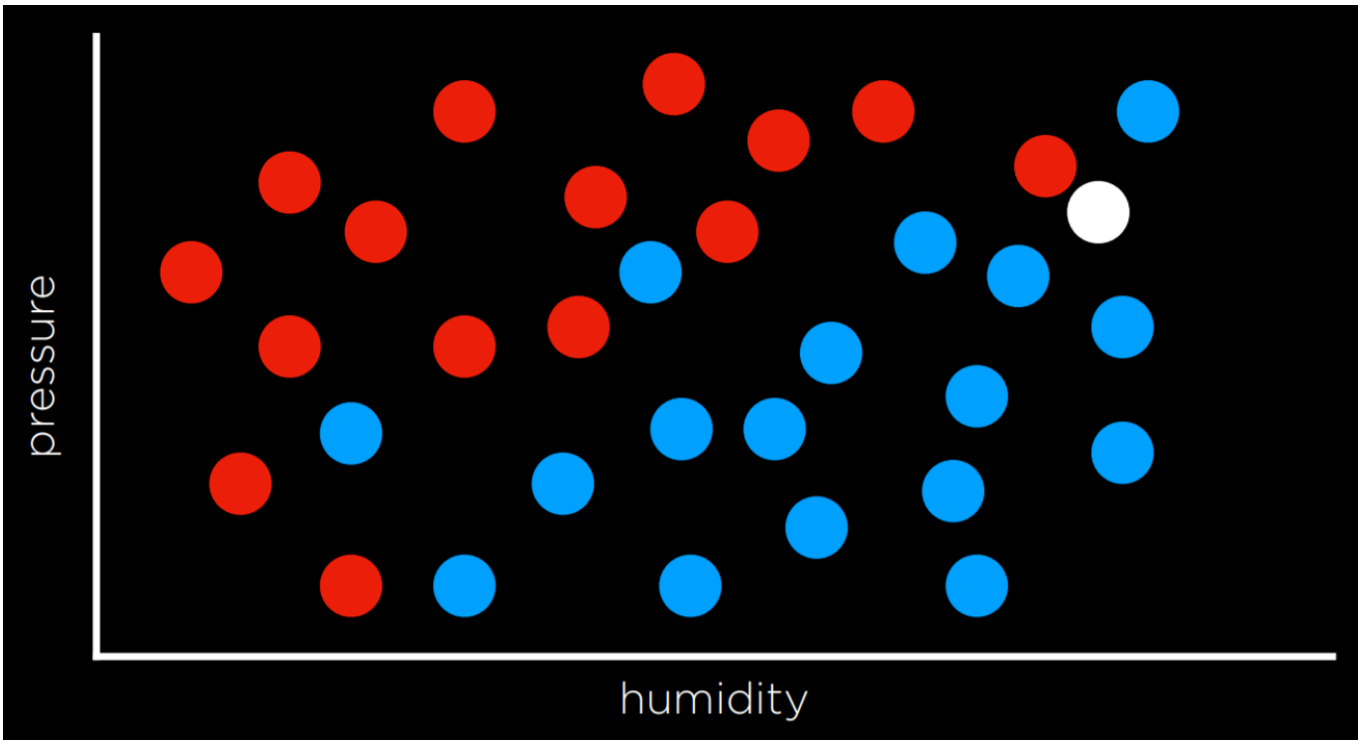
This task can be formalized as follows. We observe nature, where a function *f(humidity, pressure)* maps the input to a discrete value, either Rain or No Rain. This function is hidden from us, and it is

probably affected by many other variables that we don't have access to. Our goal is to create function *h(humidity, pressure)* that can approximate the behavior of function *f*. Such a task can be visualized by plotting days on the dimensions of humidity and rain (the input), coloring each data point in blue if it rained that day and in red if it didn't rain that day (the output). The white data point has only the input, and the computer needs to figure out the output.



Nearest-Neighbor Classification

One way of solving a task like the one described above is by assigning the variable in question the value of the closest observation. So, for example, the white dot on the graph above would be colored blue, because the nearest observed dot is blue as well. This might work well some times, but consider the graph below.



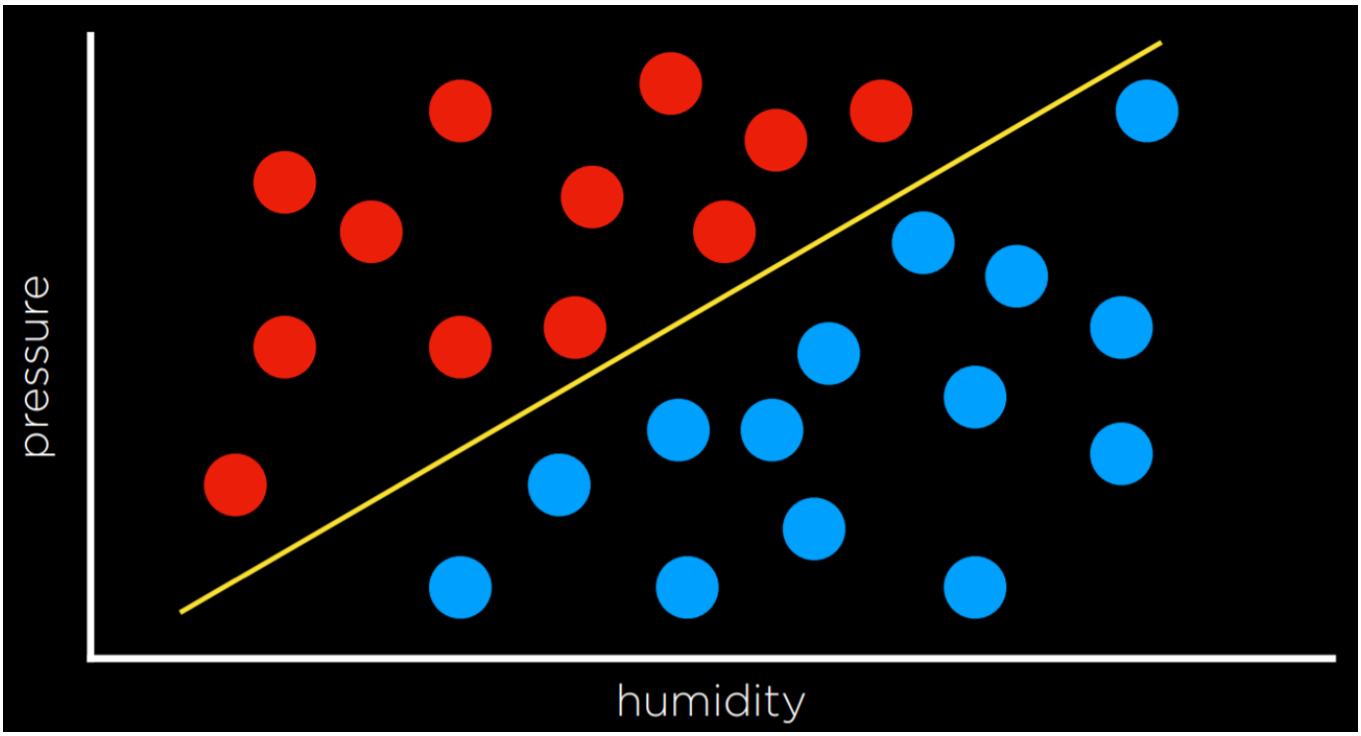
Following the same strategy, the white dot should be colored red, because the nearest observation to it is red as well. However, looking at the bigger picture, it looks like most of the other observations around it are blue, which might give us the intuition that blue is a better prediction in this case, even though the closest observation is red.

One way to get around the limitations of nearest-neighbor classification is by using **k-nearest-neighbors classification**, where the dot is colored based on the most frequent color of the *k* nearest neighbors. It is up to the programmer to decide what *k* is. Using a 3-nearest neighbors classification, for example, the white dot above will be colored blue, which intuitively seems like a better decision.

A drawback of the k-nearest-neighbors classification is that, using a naive approach, the algorithm will have to measure the distance of every single point to the point in question, which is computationally expensive. This can be sped up by using data structures that enable finding neighbors more quickly or by pruning irrelevant observation.

Perceptron Learning

Another way of going about a classification problem, as opposed to the nearest-neighbor strategy, is looking at the data as a whole and trying to create a decision boundary. In two-dimensional data, we can draw a line between the two types of observations. Every additional data point will be classified based on the side of the line on which it is plotted.



The drawback to this approach is that data are messy, and it is rare that one can draw a line and neatly divide the classes into two observations without any mistakes. Often, we will compromise, drawing a boundary that separates the observations correctly more often than not, but still occasionally misclassifies them.

In this case, the input of

- x_1 = Humidity
- x_2 = Pressure

will be given to a hypothesis function $h(x_1, x_2)$, which will output its prediction of whether it is going to rain that day or not. It will do so by checking on which side of the decision boundary the observation falls. Formally, the function will weight each of the inputs with an addition of a constant, ending in a linear equation of the following form:

- Rain $w_0 + w_1x_1 + w_2x_2 \geq 0$
- No Rain otherwise

Often, the output variable will be coded as 1 and 0, where if the equation yields more than 0, the output is 1 (Rain), and 0 otherwise (No Rain).

The weights and values are represented by vectors, which are sequences of numbers (which can be stored in lists or tuples in Python). We produce a Weight Vector **w**: (w_0, w_1, w_2), and getting to the best weight vector is the goal of the machine learning algorithm. We also produce an Input Vector **x**: $(1, x_1, x_2)$.

We take the dot product of the two vectors. That is, we multiply each value in one vector by the corresponding value in the second vector, arriving at the expression above: $w_0 + w_1x_1 + w_2x_2$. The first value in the input vector is 1 because, when multiplied by the weight vector w_0, we want to keep it a constant.

Thus, we can represent our hypothesis function the following way:

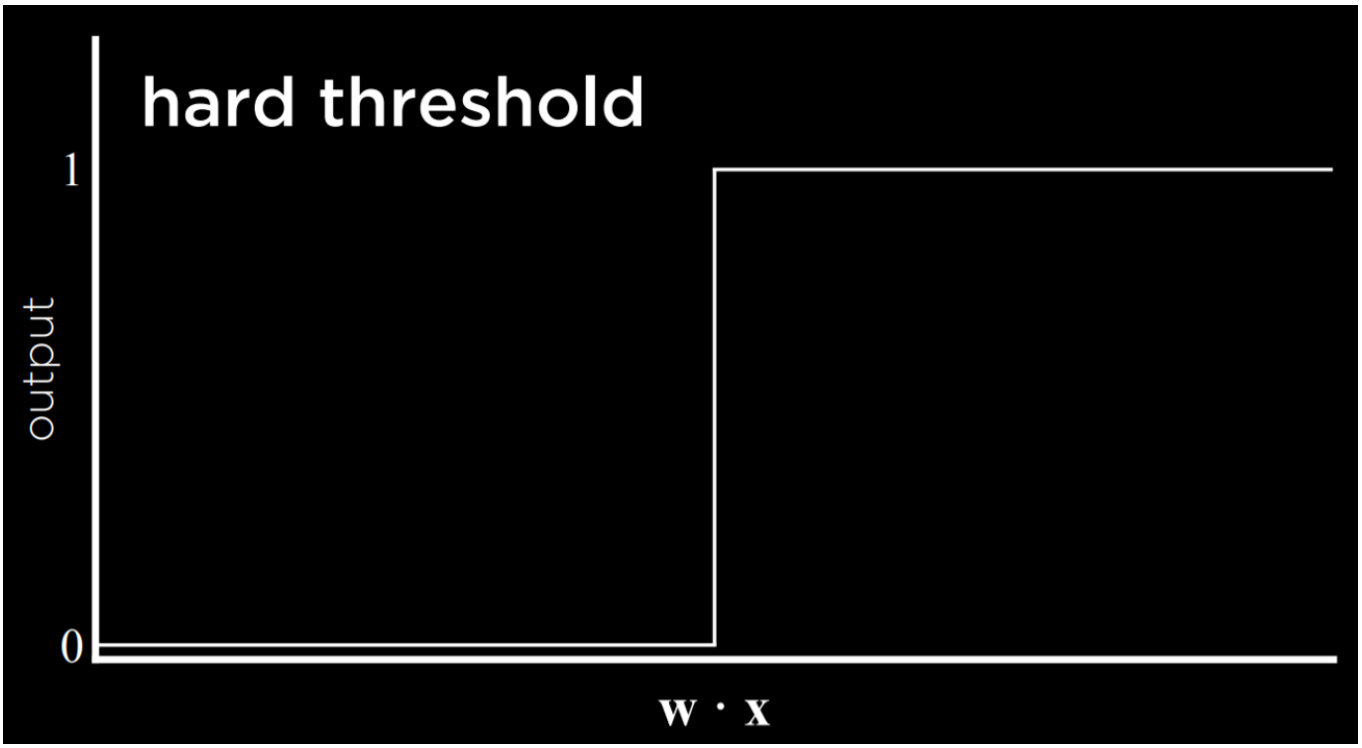
$$h_{\mathbf{w}}(\mathbf{x}) = \begin{matrix} 1 \text{ if } \mathbf{w} \cdot \mathbf{x} \geq 0 \\ 0 \text{ otherwise} \end{matrix}$$

Since the goal of the algorithm is to find the best weight vector, when the algorithm encounters new data it updates the current weights. It does so using the **perceptron learning rule**:

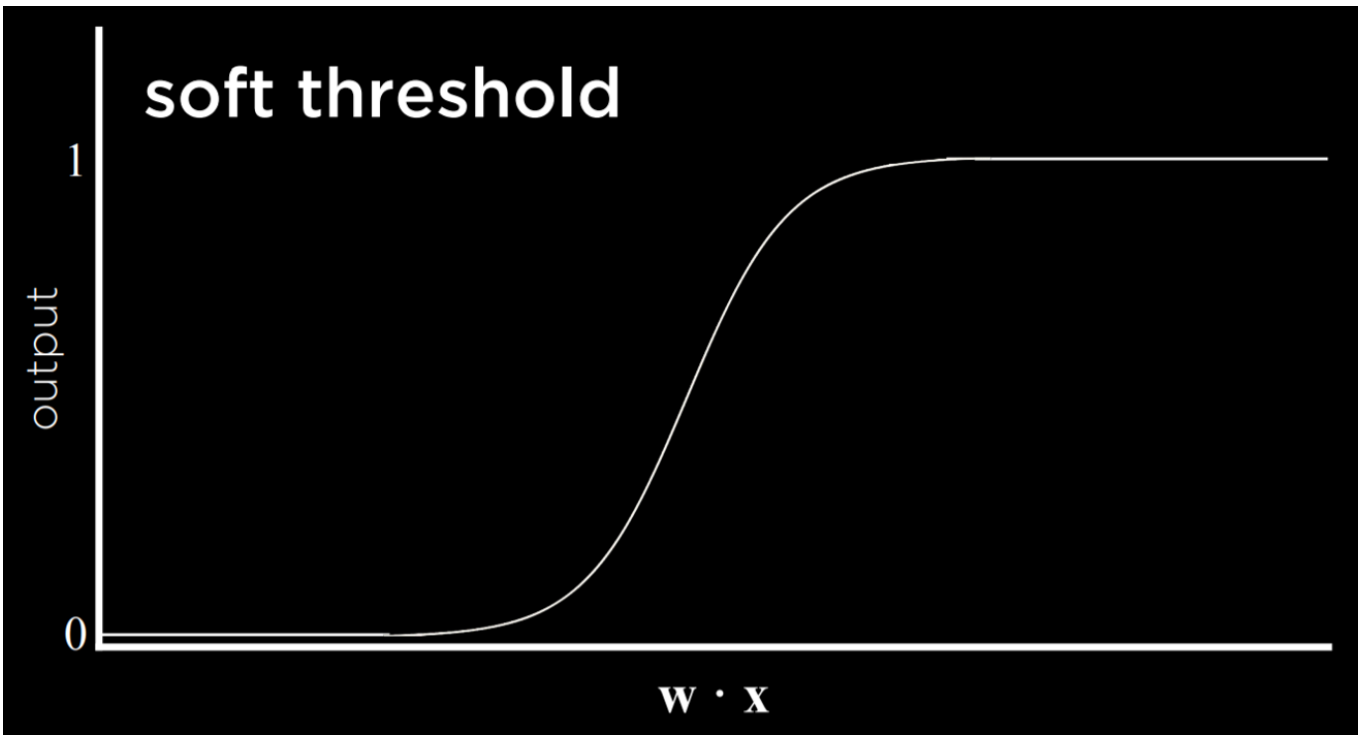
$$w_i = w_i + \alpha(y - h_{\mathbf{w}}(\mathbf{x})) \times x_i$$

The important takeaway from this rule is that for each data point, we adjust the weights to make our function more accurate. The details, which are not as critical to our point, are that each weight is set to be equal to itself plus some value value in parentheses. Here, y stands for the observed value while the hypothesis function stands for the estimate. If they are identical, this whole term is equal to zero, and thus the weight is not changed. If we underestimated (calling No Rain while Rain was observed), then the value in the parentheses will be 1 and the weight will increase by the value of x_i scaled by α the learning coefficient. If we overestimated (calling Rain while No Rain was observed), then the value in the parentheses will be -1 and the weight will decrease by the value of x scaled by α. The higher α, the stronger the influence each new event has on the weight.

The result of this process is a threshold function that switches from 0 to 1 once the estimated value crosses some threshold.

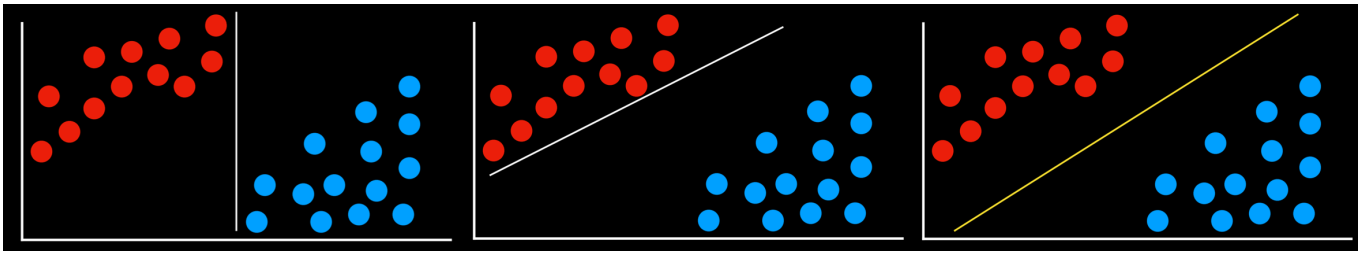


The problem with this type of function is that it is unable to express uncertainty, since it can only be equal to 0 or to 1. It employs a **hard threshold**. A way to go around this is by using a logistic function, which employs a **soft threshold**. A logistic function can yield a real number between 0 and 1, which will express confidence in the estimate. The closer the value to 1, the more likely it is to rain.



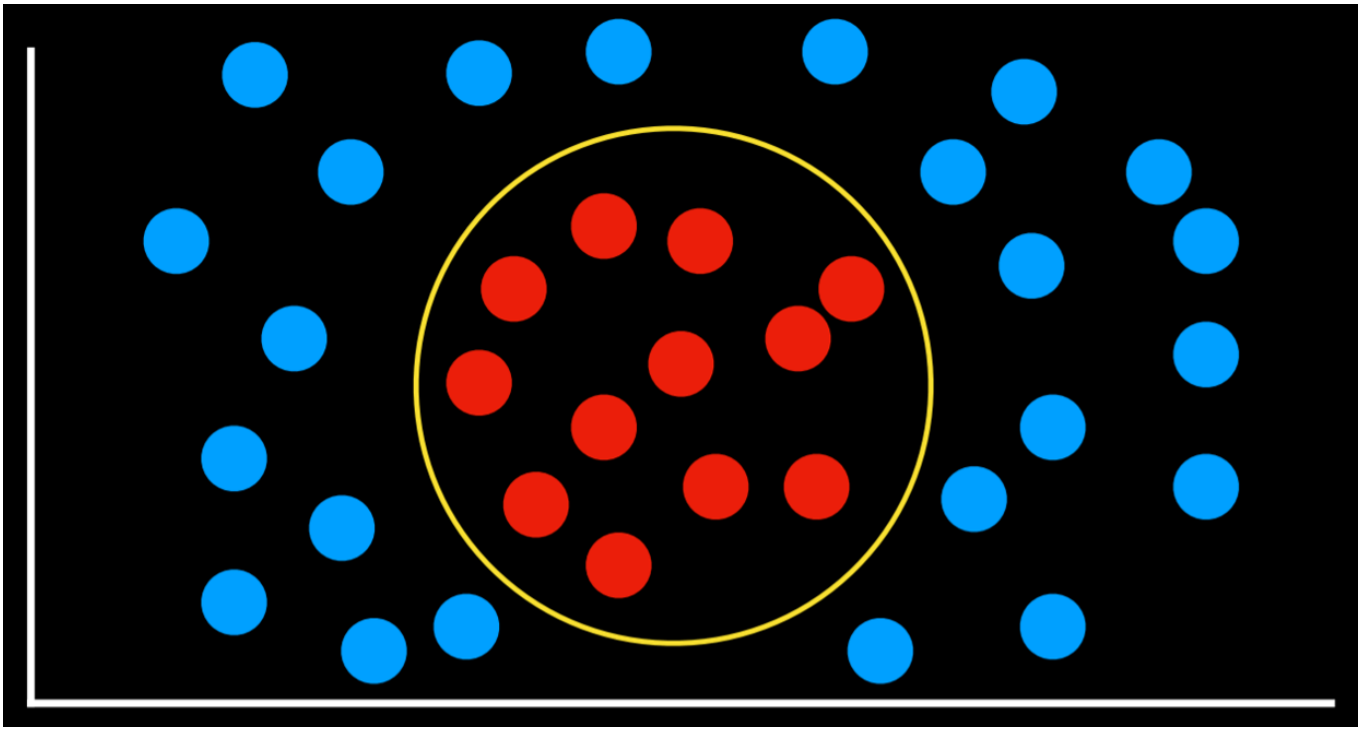
Support Vector Machines

In addition to nearest-neighbor and linear regression, another approach to classification is the Support Vector Machine. This approach uses an additional vector (support vector) near the decision boundary to make the best decision when separating the data. Consider the example below.



All the decision boundaries work in that they separate the data without any mistakes. However, are they equally as good? The two leftmost decision boundaries are very close to some of the observations. This means that a new data point that differs only slightly from one group can be wrongly classified as the other. As opposed to that, the rightmost decision boundary keeps the most distance from each of the groups, thus giving the most leeway for variation within it. This type of boundary, which is as far as possible from the two groups it separates, is called the **Maximum Margin Separator**.

Another benefit of support vector machines is that they can represent decision boundaries with more than two dimensions, as well as non-linear decision boundaries, such as below.

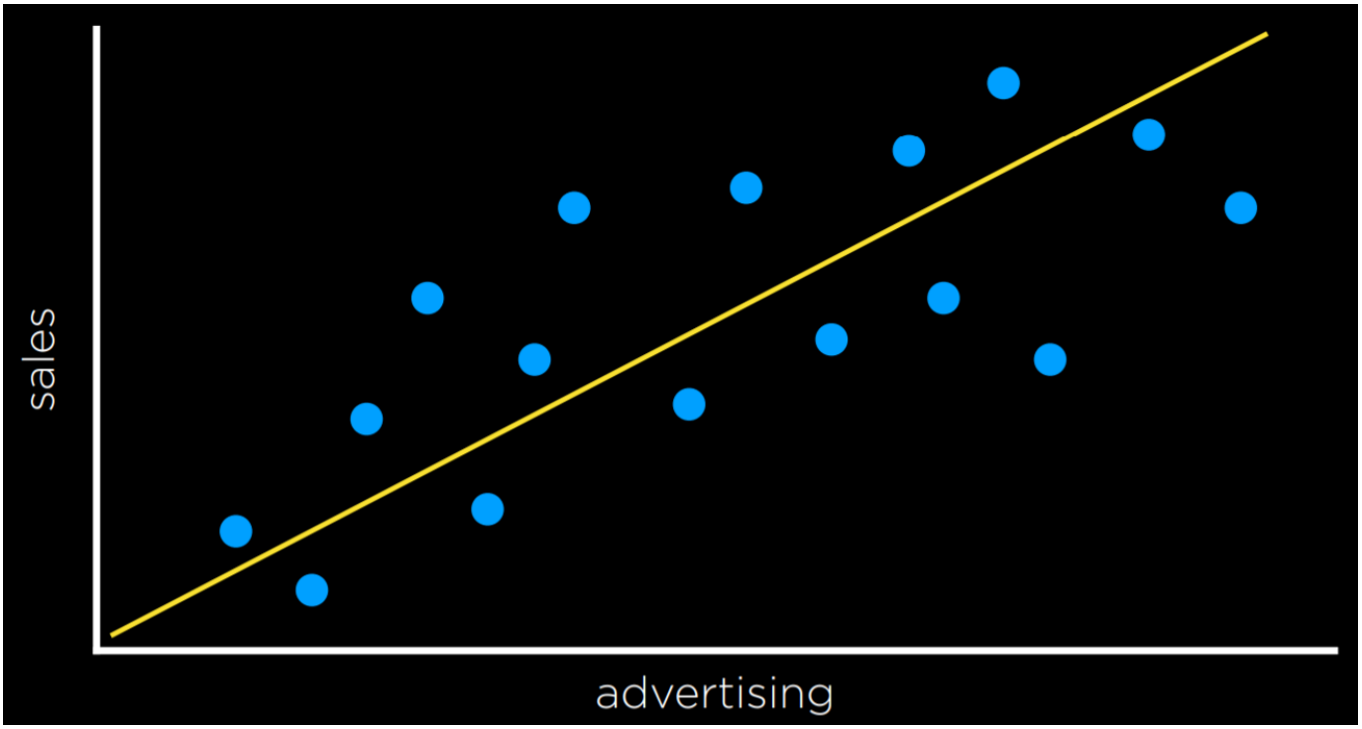


To summarize, there are multiple ways to go about classification problems, with no one being always better than the other. Each has their drawbacks and might prove more useful than others in specific situations.

Regression

Regression is a supervised learning task of a function that maps an input point to a continuous value, some real number. This differs from classification in that classification problems map an input to discrete values (Rain or No Rain).

For example, a company might use regression to answer the question of how money spent advertising predicts money earned in sales. In this case, an observed function *f(advertising)* represents the observed income following some money that was spent in advertising (note that the function can take more than one input variable). These are the data that we start with. With this data, we want to come up with a hypothesis function *h(advertising)* that will try to approximate the behavior of function *f*. *h* will generate a line whose goal is not to separate between types of observations, but to predict, based on the input, what will be the value of the output.



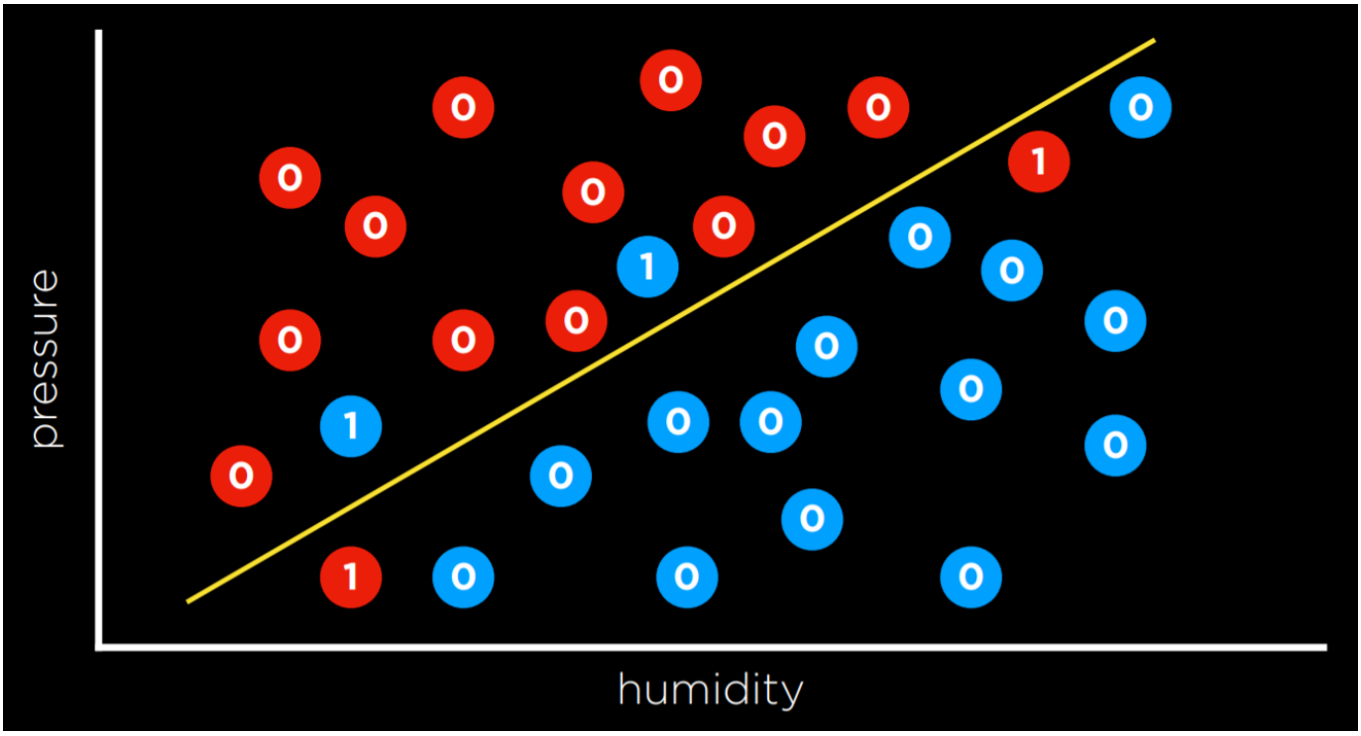
Loss Functions

Loss functions are a way to quantify the utility lost by any of the decision rules above. The less accurate the prediction, the larger the loss.

For classification problems, we can use a **0-1 Loss Function**.

- *L*(actual, predicted):
  - 0 if actual = predicted
  - 1 otherwise

In words, this function gains value when the prediction isn't correct and doesn't gain value when it is correct (i.e. when the observed and predicted values match).

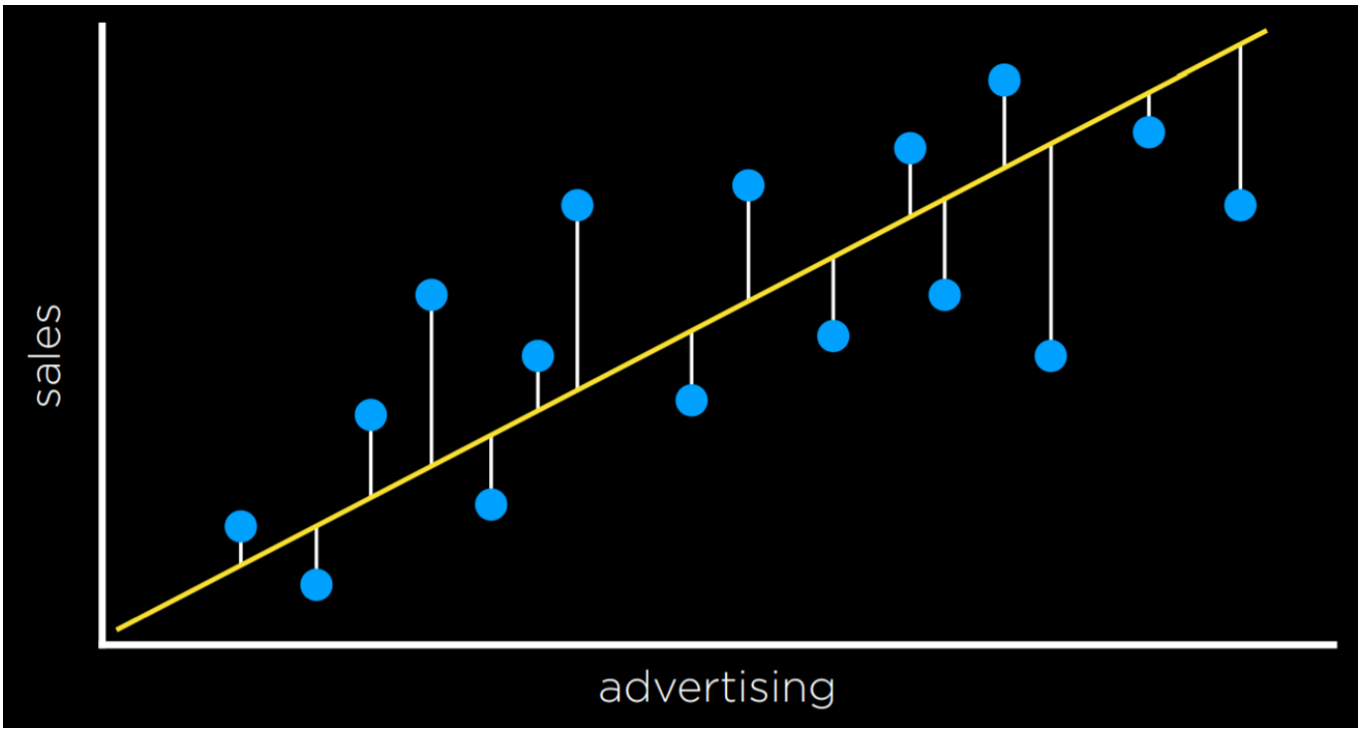


In the example above, the days that are valued at 0 are the ones where we predicted the weather correctly (rainy days are below the line and not rainy days are above the line). However, days when it didn't rain below the line and days when it did rain above it are the ones that we failed to predict. We give each one the value of 1 and sum them up to get an empirical estimate of how lossy our decision boundary is.

L_1 and L_2 loss functions can be used when predicting a continuous value. In this case, we are interested in quantifying for each prediction *how much* it differed from the observed value. We do this by taking either the absolute value or the squared value of the observed value minus the predicted value (i.e. how far the prediction was from the observed value).

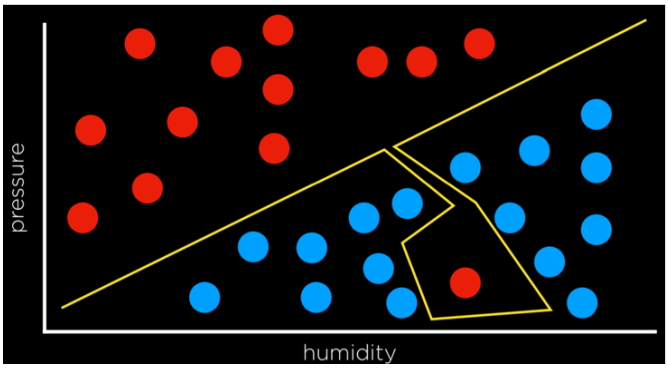
- L_1: L(actual, predicted) = |actual - predicted|
- L_2: L(actual, predicted) = (actual - predicted)^2

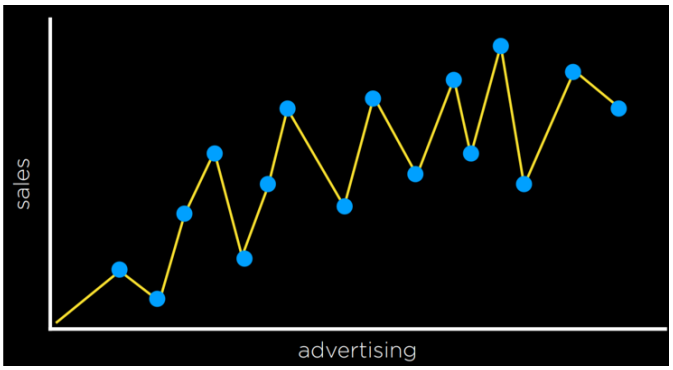
One can choose the loss function that serves their goals best. L_2 penalizes outliers more harshly than L_1 because it squares the the difference. L_1 can be visualized by summing the distances from each observed point to the predicted point on the regression line:



Overfitting

Overfitting is when a model fits the training data so well that it fails to generalize to other data sets. In this sense, loss functions are a double edged sword. In the two examples below, the loss function is minimized such that the loss is equal to 0. However, it is unlikely that it will fit new data well.





For example, in the left graph, a dot next to the red one at the bottom of the screen is likely to be Rain (blue). However, with the overfitted model, it will be classified as No Rain (red).

Regularization

Regularization is the process of penalizing hypotheses that are more complex to favor simpler, more general hypotheses. We use regularization to avoid overfitting.

In regularization, we estimate the cost of the hypothesis function h by adding up its loss and a measure of its complexity.

cost(h) = *loss*(h) + λ*complexity*(h)

Lambda (λ) is a constant that we can use to modulate how strongly to penalize for complexity in our cost function. The higher λ is, the more costly complexity is.

One way to test whether we overfitted the model is with **Holdout Cross Validation**. In this technique, we split all the data in two: a **training set** and a **test set**. We run the learning algorithm on the training set, and then see how well it predicts the data in the test set. The idea here is that by testing on data that were not used in training, we can a measure how well the learning generalizes.

The downside of holdout cross validation is that we don't get to train the model on half the data, since it is used for evaluation purposes. A way to deal with this is using ***k*-Fold Cross-Validation**. In this process, we divide the data into k sets. We run the training k times, each time leaving out one dataset and using it as a test set. We end up with k different evaluations of our model, which we can average and get an estimate of how our model generalizes without losing any data.

scikit-learn

As often is the case with Python, there are multiple libraries that allow us to conveniently use machine learning algorithms. One of such libraries is scikit-learn.

As an example, we are going to use a CSV (https://en.wikipedia.org/wiki/Comma-separated_values) dataset of counterfeit banknotes.

```
variance,skewness,curtosis,entropy,class
-0.89569,3.0025,-3.6067,-3.4457,1
3.4769,-0.15314,2.53,2.4495,0
3.9102,6.065,-2.4534,-0.68234,0
0.60731,3.9544,-4.772,-4.4853,1
2.3718,7.4908,0.015989,-1.7414,0
-2.2153,11.9625,0.078538,-7.7853,0
3.9433,2.5017,1.5215,0.903,0
3.931,1.8541,-0.023425,1.2314,0
3.9719,1.0367,0.75973,1.0013,0
0.55298,-3.4619,1.7048,1.1008,1
0.26877,4.987,-5.1508,-6.3913,1
```

The four left columns are data that we can use to predict whether a note is genuine or counterfeit, which is external data provided by a human, coded as 0 and 1. Now we can train our model on this data set and see if we can predict whether new banknotes are genuine or not.

```
import csv
import random

from sklearn import svm
from sklearn.linear_model import Perceptron
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier

# model = KNeighborsClassifier(n_neighbors=1)
# model = svm.SVC()
model = Perceptron()
```

Note that after importing the libraries, we can choose which model to use. The rest of the code will stay the same. SVC stands for Support Vector Classifier (which we know as support vector machine). The KNeighborsClassifier uses the k-neighbors strategy, and requires as input the number of neighbors it should consider.

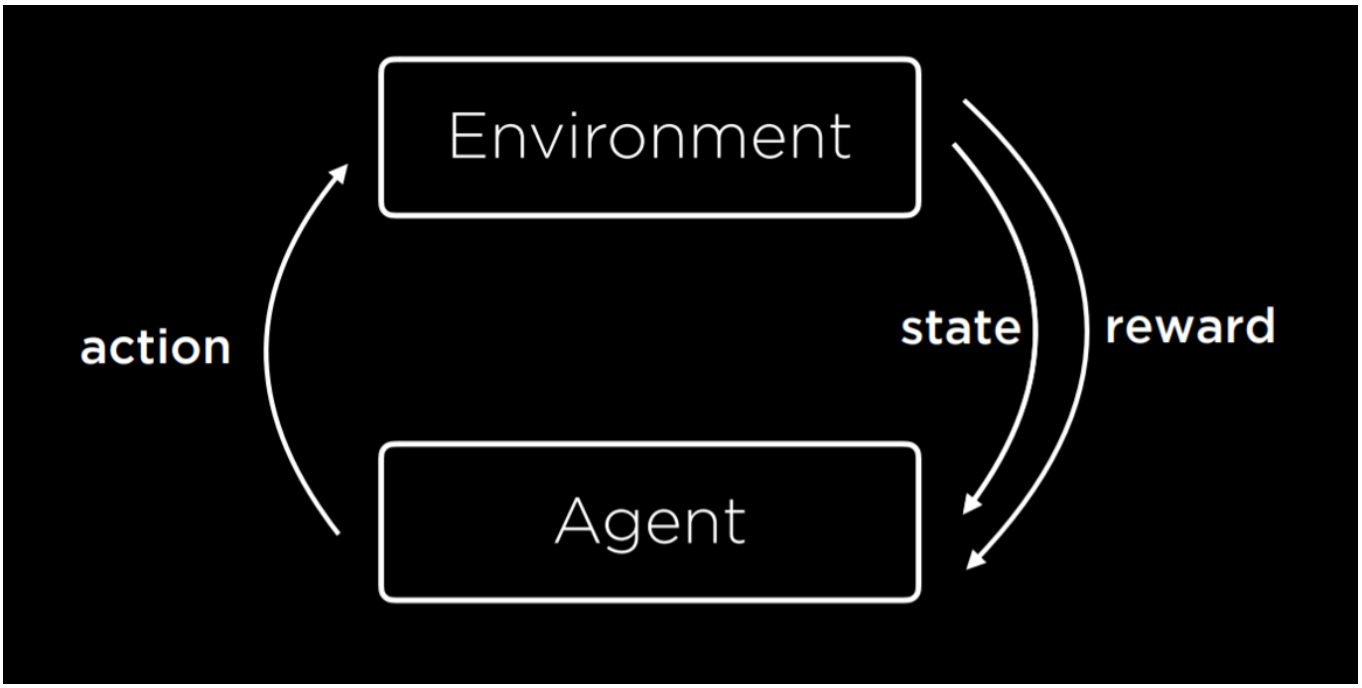
```
# Read data in from file
with open("banknotes.csv") as f:
    reader = csv.reader(f)
    next(reader)

    data = []
    for row in reader:
        data.append({
            "evidence": [float(cell) for cell in row[:4]],
            "label": "Authentic" if row[4] == "0" else "Counterfeit"
        })

# Separate data into training and testing groups
holdout = int(0.40 * len(data))
random.shuffle(data)
testing = data[:holdout]
training = data[holdout:]

# Train model on training set
X_training = [row["evidence"] for row in training]
y_training = [row["label"] for row in training]
model.fit(X_training, y_training)

# Make predictions on the testing set
X_testing = [row["evidence"] for row in testing]
y_testing = [row["label"] for row in testing]
predictions = model.predict(X_testing)

# Compute how well we performed
correct = 0
incorrect = 0
```

```
total = 0
for actual, predicted in zip(y_testing, predictions):
    total += 1
    if actual == predicted:
        correct += 1
    else:
        incorrect += 1

# Print results
print(f"Results for model {type(model).__name__}")
print(f"Correct: {correct}")
print(f"Incorrect: {incorrect}")
print(f"Accuracy: {100 * correct / total:.2f}%")
```

This manual version of running the algorithm can be found in the source code for this lecture under banknotes0.py. Since the algorithm is used often in a similar way, scikit-learn contains additional functions that make the code even more succinct and easy to use, and this version can be found under banknotes1.py.

Reinforcement Learning

Reinforcement learning is another approach to machine learning, where after each action, the agent gets feedback in the form of reward or punishment (a positive or a negative numerical value).



The learning process starts by the environment providing a state to the agent. Then, the agent performs an action on the state. Based on this action, the environment will return a state and a reward to the agent, where the reward can be positive, making the behavior more likely in the future, or negative (i.e. punishment), making the behavior less likely in the future.

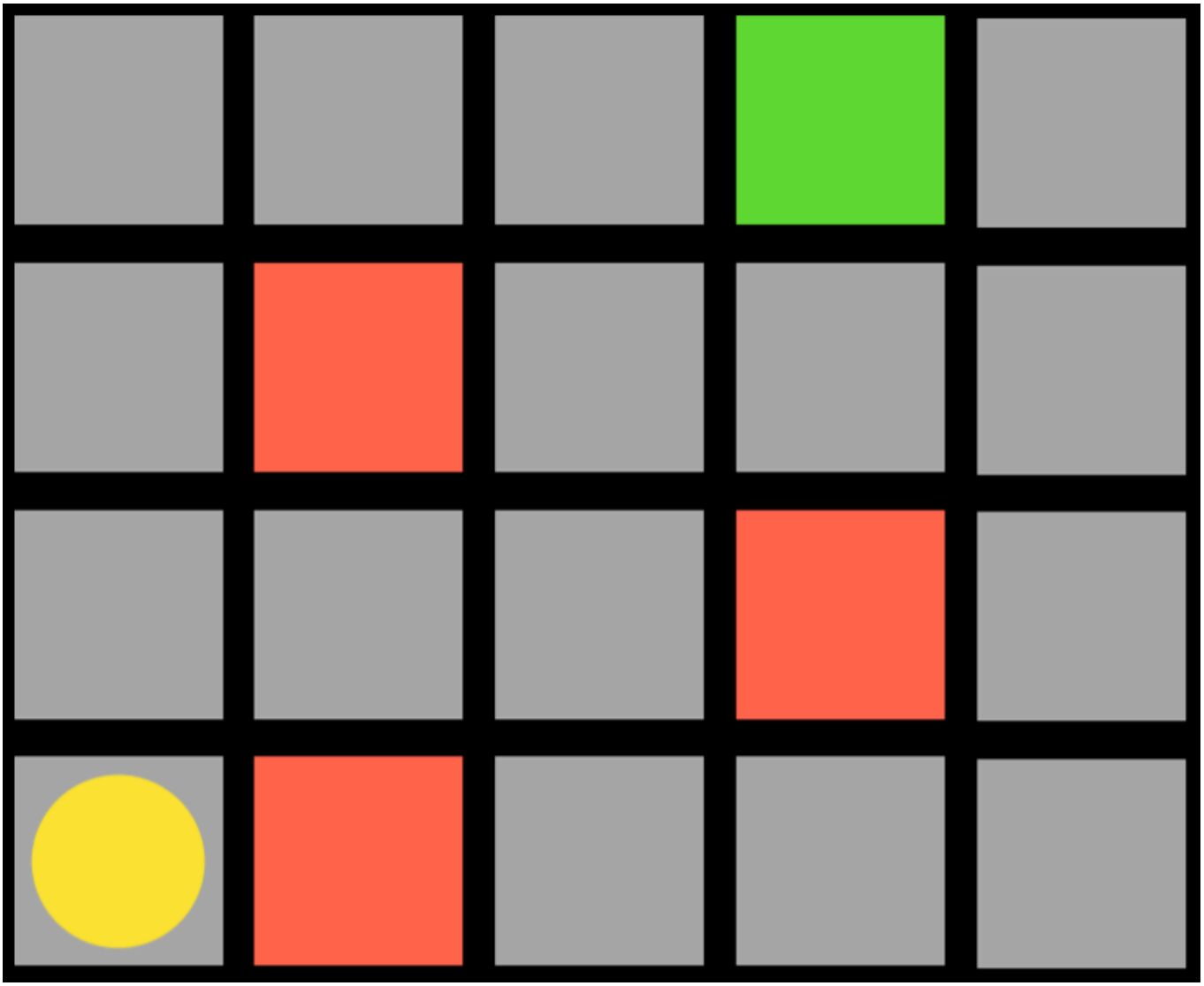
This type of algorithm can be used to train walking robots, for example, where each step returns a positive number (reward) and each fall a negative number (punishment).

Markov Decision Processes

Reinforcement learning can be viewed as a Markov decision process, having the following properties:

- Set of states S
- Set of actions $Actions(S)$
- Transition model $P(s' \mid s, a)$
- Reward function $R(s, a, s')$

For example, consider the following task:



The agent is the yellow circle, and it needs to get to the green square while avoiding the red squares. Every single square in the task is a state. Moving up, down, or to the sides is an action. The

transition model gives us the new state after performing an action, and the reward function is what kind of feedback the agent gets. For example, if the agent chooses to go right, it will step on a red square and get negative feedback. This means that the agent will learn that, when in the state of being in the bottom-left square, it should avoid going right. This way, the agent will start exploring the space, learning which state-action pairs it should avoid. The algorithm can be probabilistic, choosing to take different actions in different states based on some probability that's being increased or decreased based on reward. When the agent reaches the green square, it will get a positive reward, learning that it is favorable to take the action it took in the previous state.

Q-Learning

Q-Learning is one model of reinforcement learning, where a function ***Q(s, a)*** outputs an estimate of the value of taking action *a* in state *s*.

The model starts with all estimated values equal to 0 (***Q(s,a)* = 0** for all *s, a*). When an action is taken and a reward is received, the function does two things: 1) it estimates the value of ***Q(s, a)*** based on current reward and expected future rewards, and 2) updates ***Q(s, a)*** to take into account both the old estimate and the new estimate. This gives us an algorithm that is capable of improving upon its past knowledge without starting from scratch.

$Q(s, a) \leftarrow Q(s, a) + \alpha(\text{new value estimate} - Q(s, a))$

The updated value of ***Q(s, a)*** is equal to the previous value of ***Q(s, a)*** in addition to some updating value. This value is determined as the difference between the new value and the old value, multiplied by α, a learning coefficient. When $\alpha = 1$ the new estimate simply overwrites the old one. When $\alpha = 0$, the estimated value is never updated. By raising and lowering α, we can determine how fast previous knowledge is being updated by new estimates.

The new value estimate can be expressed as a sum of the reward (r) and the future reward estimate. To get the future reward estimate, we consider the new state that we got after taking the last action, and add the estimate of the action in this new state that will bring to the highest reward. This way, we estimate the utility of making action *a* in state *s* not only by the reward it received, but also by the expected utility of the next step. The value of the future reward estimate can sometimes appear with a coefficient gamma that controls how much future rewards are valued. We end up with the following equation:

$$Q(s, a) \leftarrow Q(s, a) + \alpha((r + \gamma \max_{a'} Q(s', a')) - Q(s, a))$$

A **Greedy Decision-Making** algorithm completely discounts the future estimated rewards, instead always choosing the action ***a*** in current state ***s*** that has the highest ***Q(s, a)***.

This brings us to discuss the **Explore vs. Exploit** tradeoff. A greedy algorithm always exploits, taking the actions that are already established to bring to good outcomes. However, it will always follow the same path to the solution, never finding a better path. Exploration, on the other hand, means that the algorithm may use a previously unexplored route on its way to the target, allowing it to discover more efficient solutions along the way. For example, if you listen to the same songs every single time, you know you will enjoy them, but you will never get to know new songs that you might like even more!

To implement the concept of exploration and exploitation, we can use the **ε (epsilon) greedy** algorithm. In this type of algorithm, we set ε equal to how often we want to move randomly. With probability 1-ε, the algorithm chooses the best move (exploitation). With probability ε, the algorithm chooses a random move (exploration).

Another way to train a reinforcement learning model is to give feedback not upon every move, but upon the end of the whole process. For example, consider a game of Nim. In this game, different numbers of objects are distributed between piles. Each player takes any number of objects from any one single pile, and the player who takes the last object looses. In such a game, an untrained AI will play randomly, and it will be easy to win against it. To train the AI, it will start from playing a game randomly, and in the end get a reward of 1 for winning and -1 for losing. When it is trained on 10,000 games, for example, it is already smart enough to be hard to win against it.

This approach becomes more computationally demanding when a game has multiple states and possible actions, such as chess. It is infeasible to generate an estimated value for every possible move in every possible state. In this case, we can use a **function approximation**, which allows us to approximate $Q(s, a)$ using various other features, rather than storing one value for each state-action pair. Thus, the algorithm becomes able to recognize which moves are similar enough so that their estimated value should be similar as well, and use this heuristic in its decision making.

Unsupervised Learning

In all the cases we saw before, as in supervised learning, we had data with labels that the algorithm could learn from. For example, when we trained an algorithm to recognize counterfeit notes, each banknote had four variables with different values (the input data) and whether it is counterfeit or not (the label). In unsupervised learning, only the input data is present and the AI learns patterns in these data.

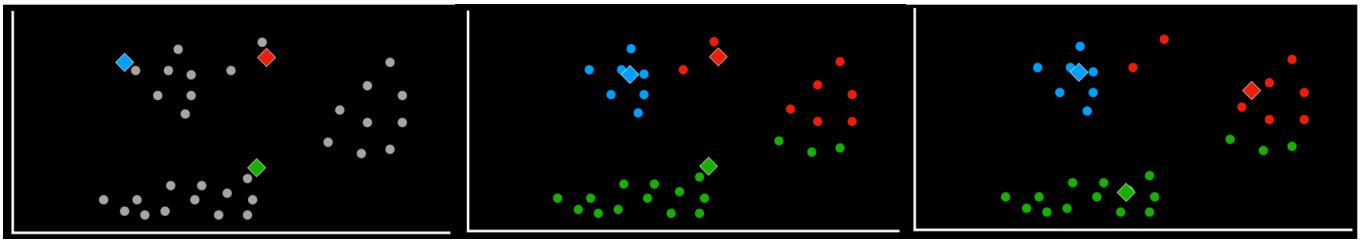
Clustering

Clustering is an unsupervised learning task that takes the input data and organizes it into groups such that similar objects end up in the same group. This can be used, for example, in genetics

research, when trying to find similar genes, or in image segmentation, when defining different parts of the image based on similarity between pixels.

k-means Clustering

k-means Clustering is an algorithm to perform a clustering task. It maps all data points in a space, and then randomly places k cluster centers in the space (it is up to the programmer to decide how many; this is the starting state we see on the left). Each cluster center is simply a point in the space. Then, each cluster gets assigned all the points that are closest to its center than to any other center (this is the middle picture). Then, in an iterative process, the cluster center moves to the middle of all these points (the state on the right), and then points are reassigned again to the clusters whose centers are now closest to them. When, after repeating the process, each point remains in the same cluster it was before, we have reached an equilibrium and the algorithm is over, leaving us with points divided between clusters.



CS50's Introduction to Artificial Intelligence with Python

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

Brian Yu (https://brianyu.me)
brian@cs.harvard.edu

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://www.reddit.com/user/davidjmalan) (https://www.threads.net/@davidjmalan)
(https://twitter.com/davidjmalan)

Lecture 5

Neural Networks

AI neural networks are inspired by neuroscience. In the brain, neurons are cells that are connected to each other, forming networks. Each neuron is capable of both receiving and sending electrical signals. Once the electrical input that a neuron receives crosses some threshold, the neuron activates, thus sending its electrical signal forward.

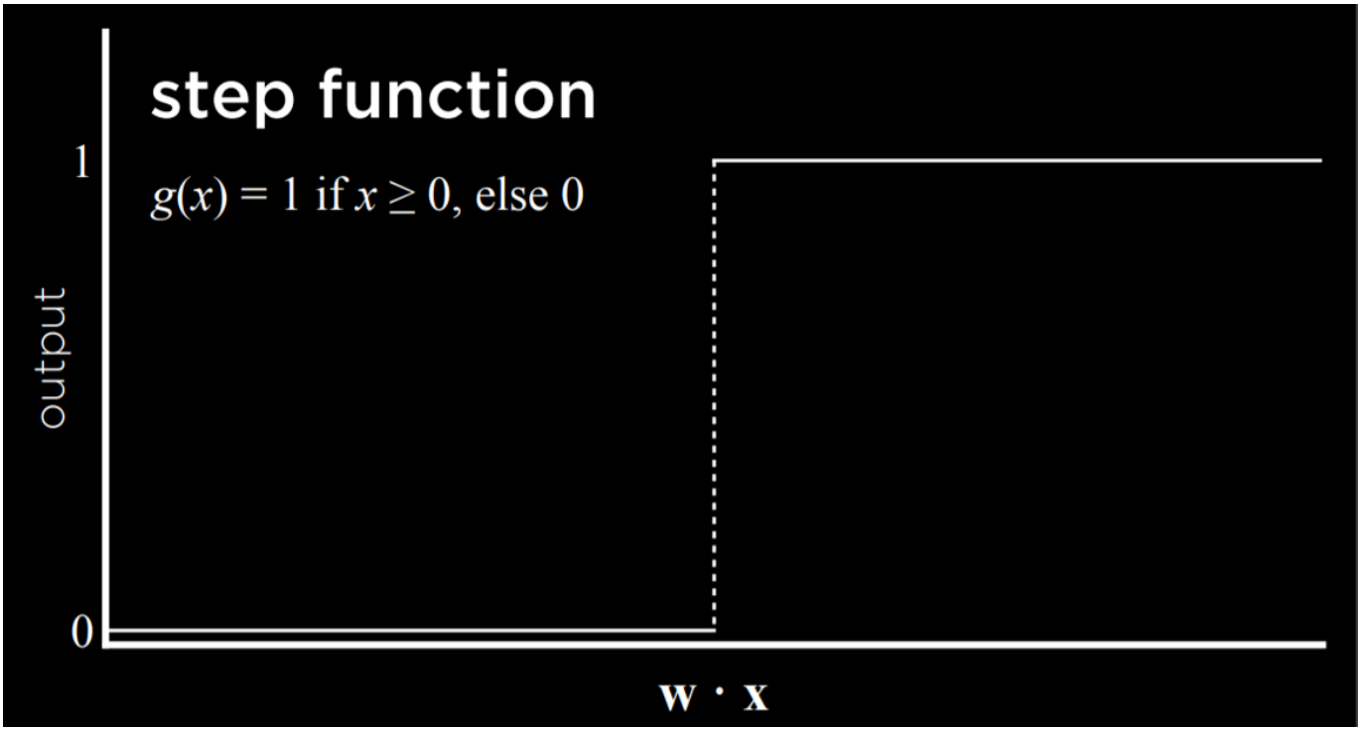
An **Artificial Neural Network** is a mathematical model for learning inspired by biological neural networks. Artificial neural networks model mathematical functions that map inputs to outputs based on the structure and parameters of the network. In artificial neural networks, the structure of the network is shaped through training on data.

When implemented in AI, the parallel of each neuron is a **unit** that's connected to other units. For example, like in the last lecture, the AI might map two inputs, x_1 and x_2, to whether it is going to rain today or not. Last lecture, we suggested the following form for this hypothesis function: $h(x_1, x_2) = w_0 + w_1x_1 + w_2x_2$, where w_1 and w_2 are weights that modify the inputs, and w_0 is a constant, also called **bias**, modifying the value of the whole expression.

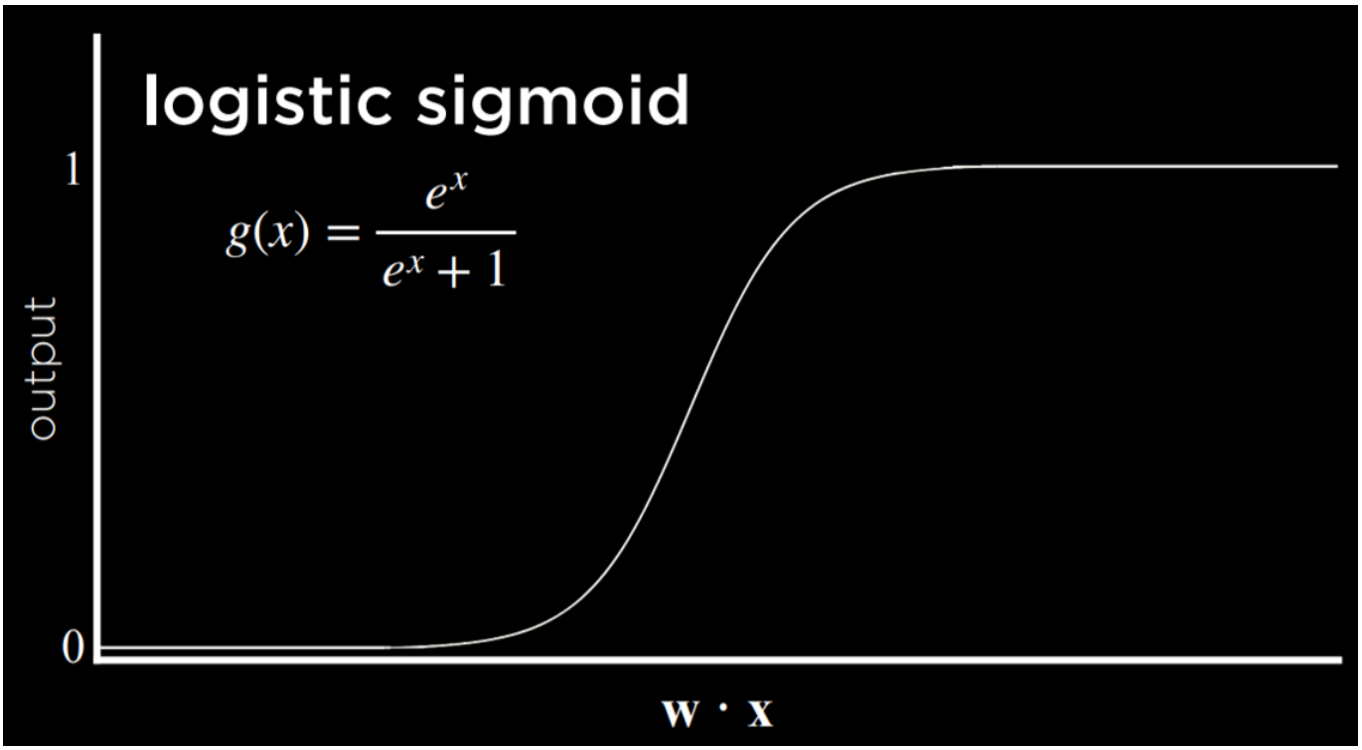
Activation Functions

To use the hypothesis function to decide whether it rains or not, we need to create some sort of threshold based on the value it produces.

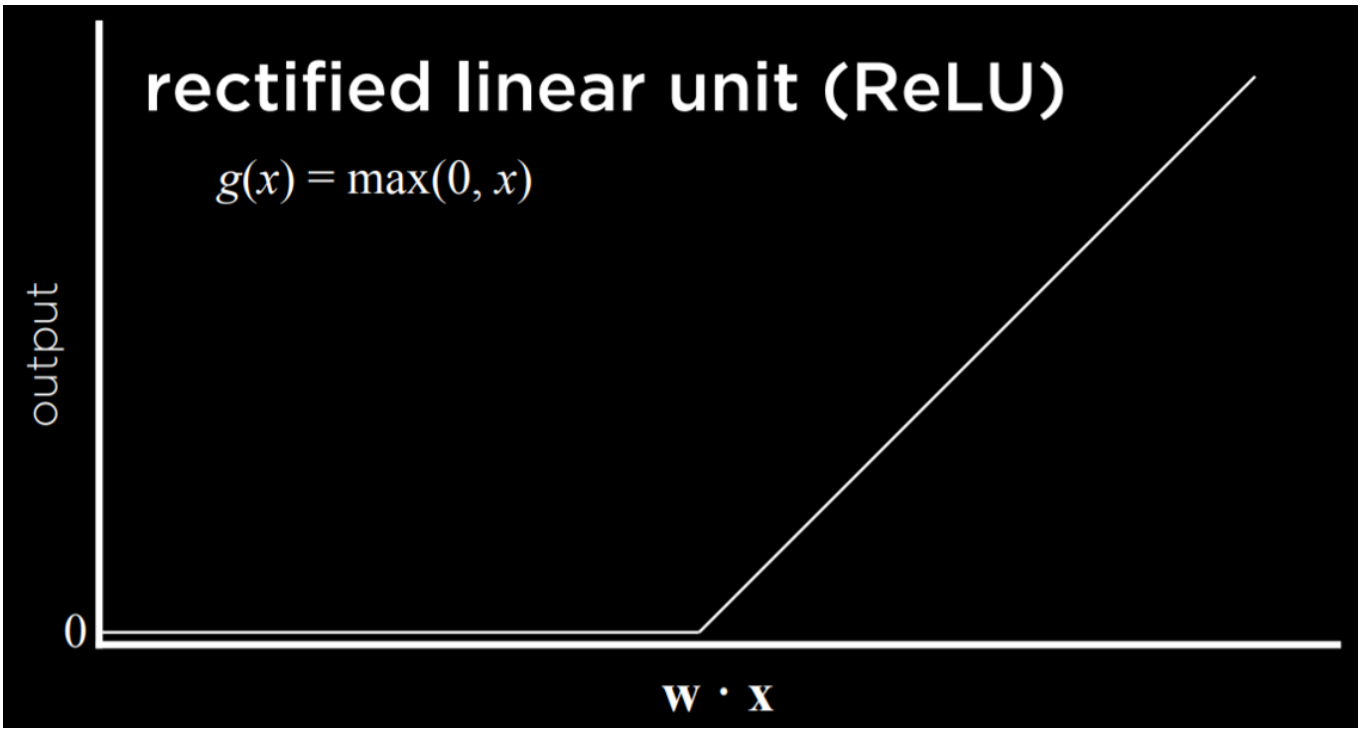
One way to do this is with a step function, which gives 0 before a certain threshold is reached and 1 after the threshold is reached.



Another way to go about this is with a logistic function, which gives as output any real number from 0 to 1, thus expressing graded confidence in its judgment.



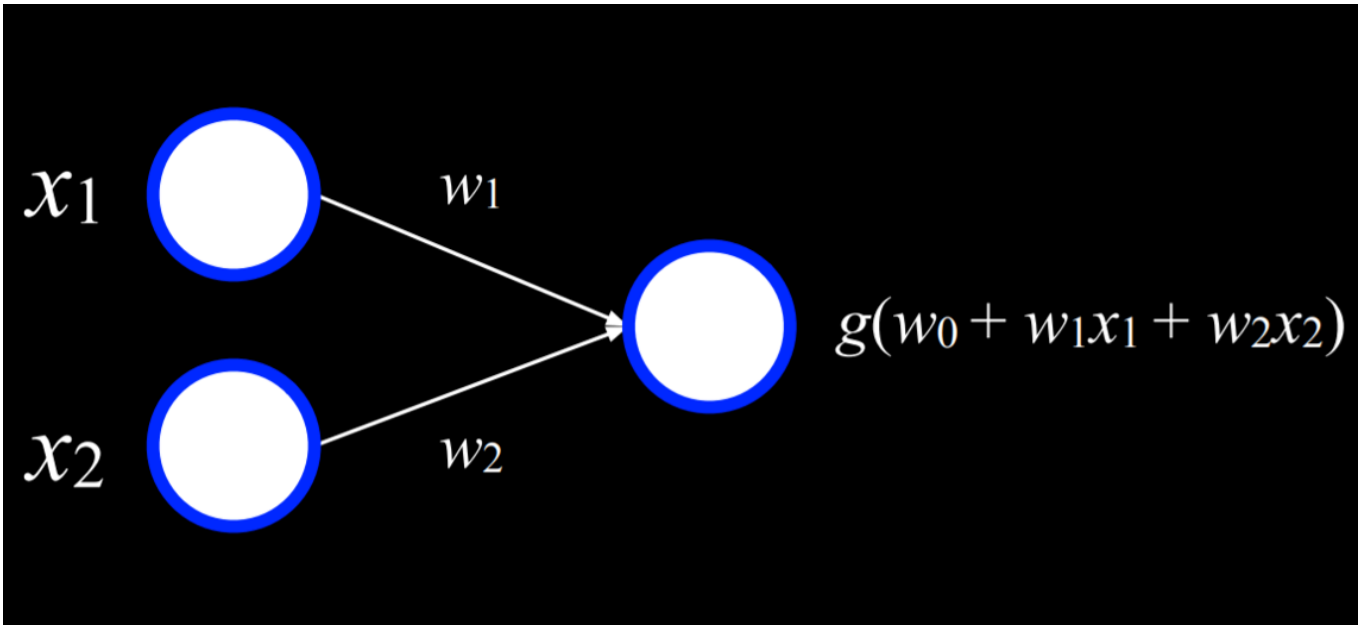
Another possible function is Rectified Linear Unit (ReLU), which allows the output to be any positive value. If the value is negative, ReLU sets it to 0.



Whichever function we choose to use, we learned last lecture that the inputs are modified by weights in addition to the bias, and the sum of those is passed to an activation function. This stays true for simple neural networks.

Neural Network Structure

A neural network can be thought of as a representation of the idea above, where a function sums up inputs to produce an output.

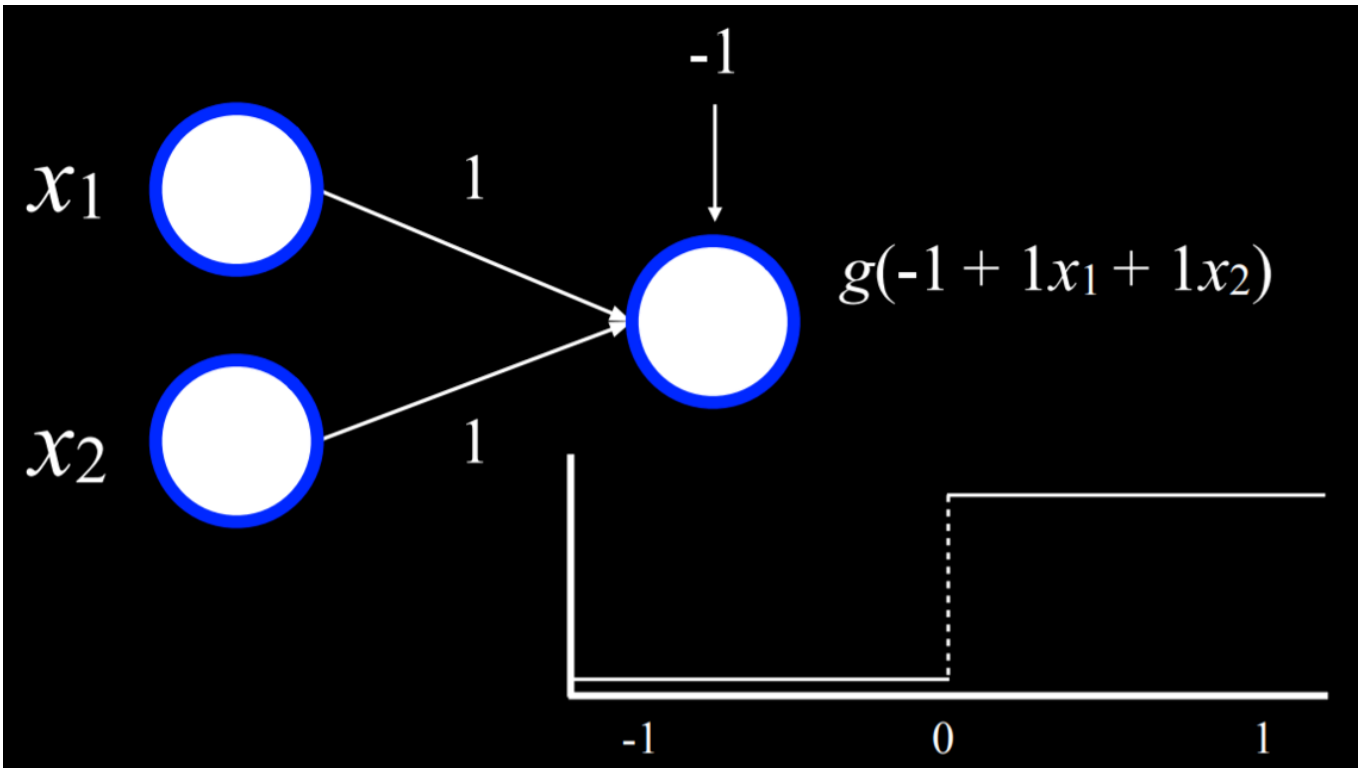


The two white units on the left are the input and the unit on the right is an output. The inputs are connected to the output by a weighted edge. To make a decision, the output unit multiplies the inputs by their weights in addition to the bias (w_0), and the uses function g to determine the output.

For example, an Or logical connective can be represented as a function f with the following truth table:

x	y	$f(x, y)$
0	0	0
0	1	1
1	0	1
1	1	1

We can visualize this function as a neural network. x_1 is one input unit, and x_2 is another input unit. They are connected to the output unit by an edge with a weight of 1. The output unit then uses function $g(-1 + 1x_1 + 2x_2)$ with a threshold of 0 to output either 0 or 1 (false or true).



For example, in the case where $x_1 = x_2 = 0$, the sum is (-1). This is below the threshold, so the function g will output 0. However, if either or both of x_1 or x_2 are equal to 1, then the sum of all inputs will be either 0 or 1. Both are at or above the threshold, so the function will output 1.

A similar process can be repeated with the And function (where the bias will be (-2)). Moreover, inputs and outputs don't have to be distinct. A similar process can be used to take humidity and air pressure as input, and produce the probability of rain as output. Or, in a different example, inputs can be money spent on advertising and the month when it was spent to get the output of expected revenue from sales. This can be extended to any number of inputs by multiplying each input $x_1 \dots x_n$ by weight $w_1 \dots w_n$, summing up the resulting values and adding a bias w_0.

Gradient Descent

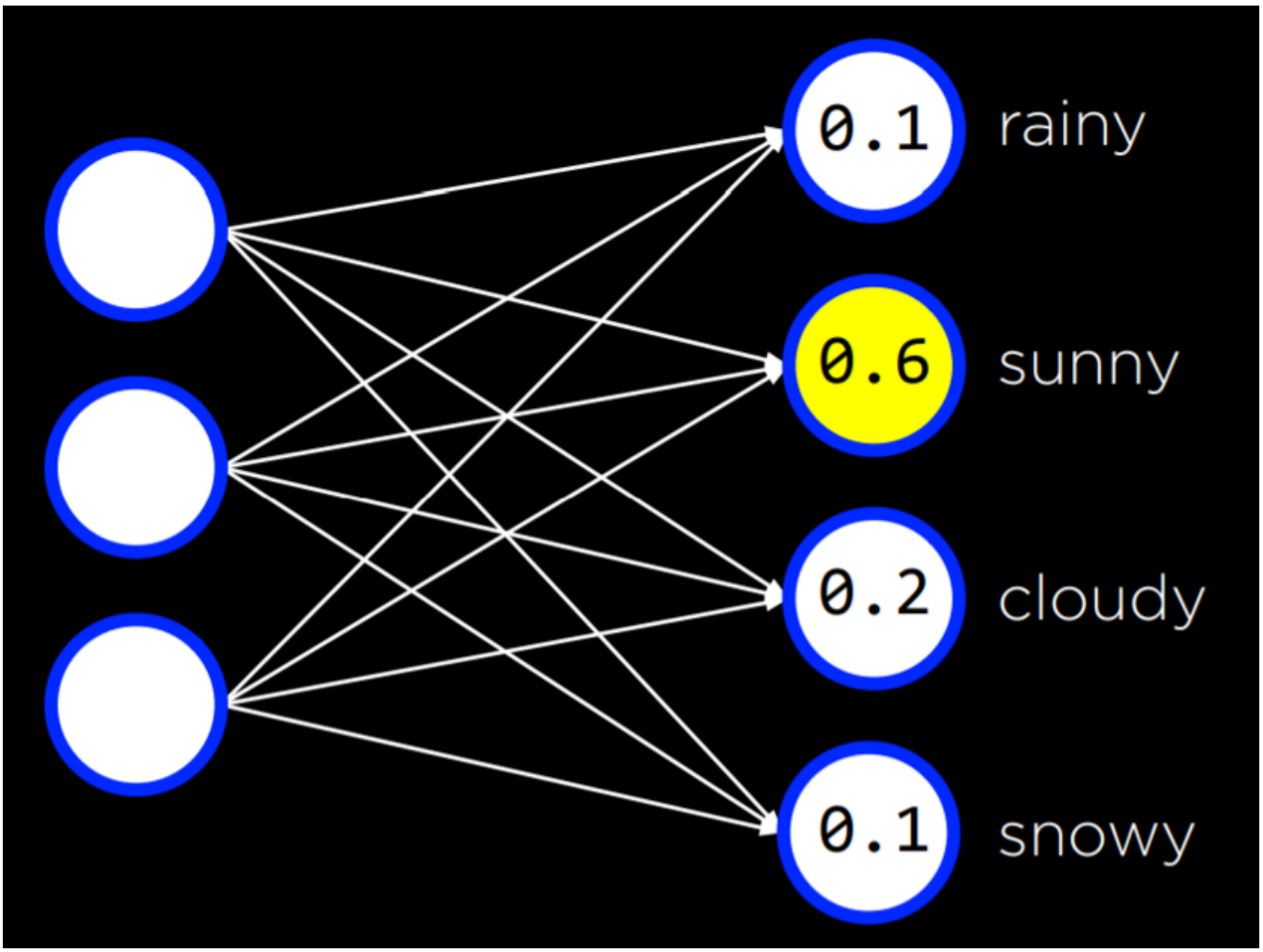
Gradient descent is an algorithm for minimizing loss when training neural networks. As was mentioned earlier, a neural network is capable of inferring knowledge about the structure of the network itself from the data. Whereas, so far, we defined the different weights, neural networks allow us to compute these weights based on the training data. To do this, we use the gradient descent algorithm, which works the following way:

- Start with a random choice of weights. This is our naive starting place, where we don't know how much we should weight each input.
- Repeat:
  - Calculate the gradient based on all data points that will lead to decreasing loss. Ultimately, the gradient is a vector (a sequence of numbers).

- Update weights according to the gradient.

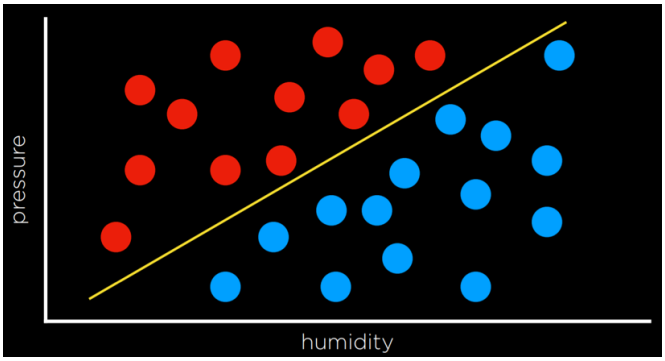
The problem with this kind of algorithm is that it requires to calculate the gradient based on *all data points*, which is computationally costly. There are a multiple ways to minimize this cost. For example, in **Stochastic Gradient Descent**, the gradient is calculated based on one point chosen at random. This kind of gradient can be quite inaccurate, leading to the **Mini-Batch Gradient Descent** algorithm, which computes the gradient based on on a few points selected at random, thus finding a compromise between computation cost and accuracy. As often is the case, none of these solutions is perfect, and different solutions might be employed in different situations.

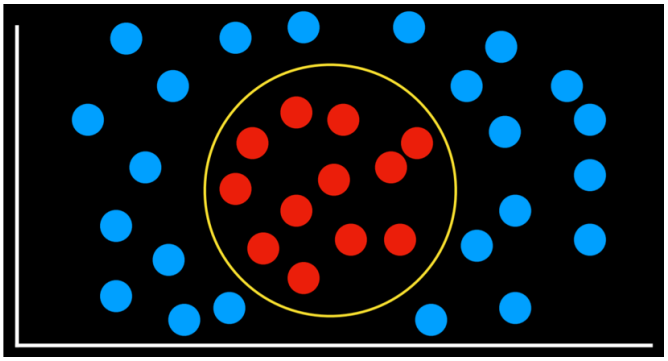
Using gradient descent, it is possible to find answers to many problems. For example, we might want to know more than "will it rain today?" We can use some inputs to generate probabilities for different kinds of weather, and then just choose the weather that is most probable.



This can be done with any number of inputs and outputs, where each input is connected to each output, and where the outputs represent decisions that we can make. Note that in this kind of neural networks the outputs are not connected. This means that each output and its associated weights from all the inputs can be be seen as an individual neural network and thus can be trained separately from the rest of the outputs.

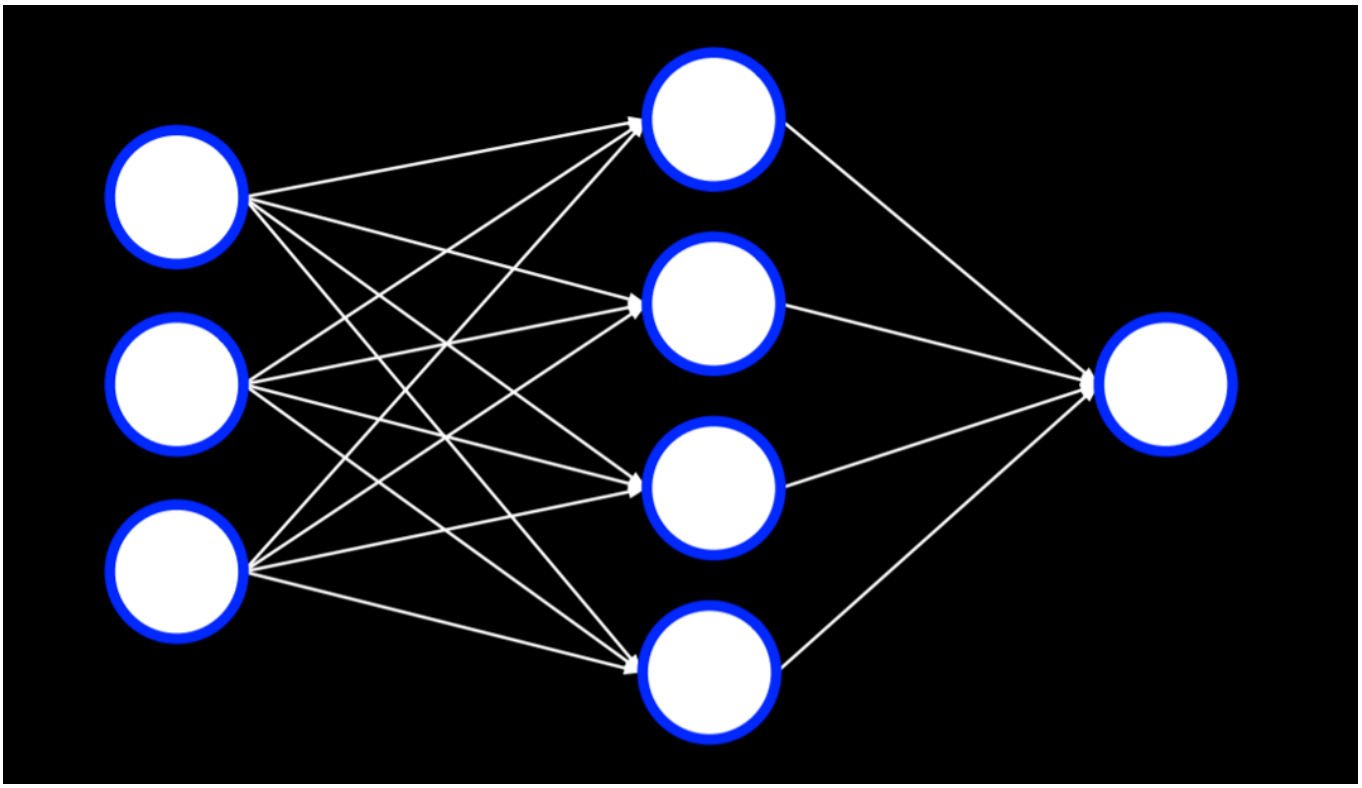
So far, our neural networks relied on **perceptron** output units. These are units that are only capable of learning a linear decision boundary, using a straight line to separate data. That is, based on a linear equation, the perceptron could classify an input to be one type or another (e.g. left picture). However, often, data are not linearly separable (e.g. right picture). In this case, we turn to multilayer neural networks to model data non-linearly.





Multilayer Neural Networks

A multilayer neural network is an artificial neural network with an input layer, an output layer, and at least one **hidden** layer. While we provide inputs and outputs to train the model, we, the humans, don't provide any values to the units inside the hidden layers. Each unit in the first hidden layer receives a weighted value from each of the units in the input layer, performs some action on it and outputs a value. Each of these values is weighted and further propagated to the next layer, repeating the process until the output layer is reached. Through hidden layers, it is possible to model non-linear data.

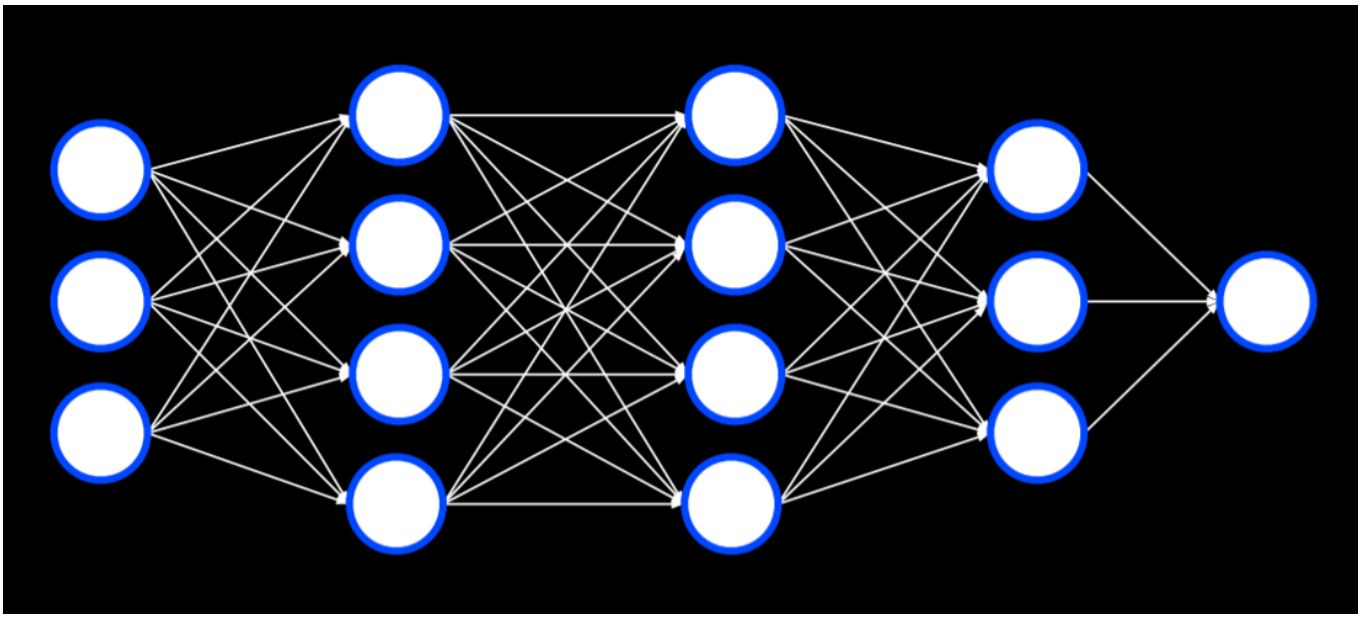


Backpropagation

Backpropagation is the main algorithm used for training neural networks with hidden layers. It does so by starting with the errors in the output units, calculating the gradient descent for the weights of the previous layer, and repeating the process until the input layer is reached. In pseudocode, we can describe the algorithm as follows:

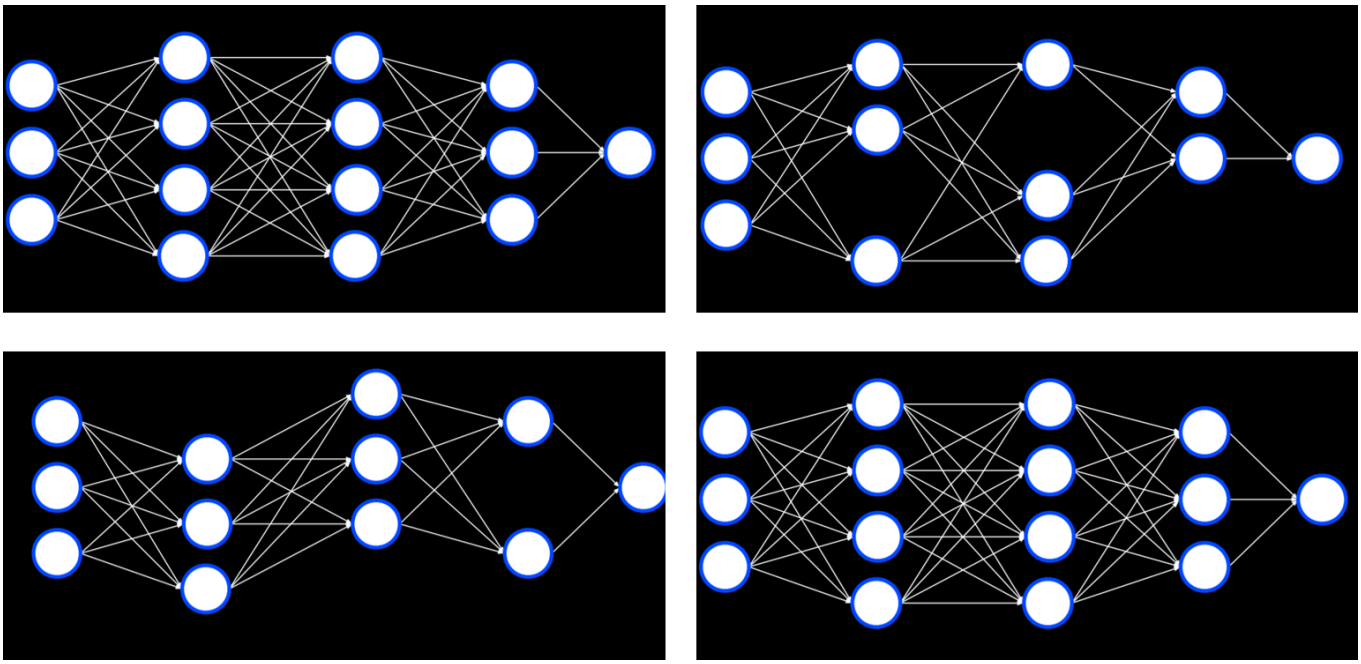
- Calculate error for output layer
- For each layer, starting with output layer and moving inwards towards earliest hidden layer:
  - Propagate error back one layer. In other words, the current layer that's being considered sends the errors to the preceding layer.
  - Update weights.

This can be extended to any number of hidden layers, creating **deep neural networks**, which are neural networks that have more than one hidden layer.



Overfitting

Overfitting is the danger of modeling the training data too closely, thus failing to generalize to new data. One way to combat overfitting is by **dropout**. In this technique, we temporarily remove units that we select at random during the learning phase. This way, we try to prevent over-reliance on any one unit in the network. Throughout training, the neural network will assume different forms, each time dropping some other units and then using them again:



Note that after the training is finished, the whole neural network will be used again.

TensorFlow

Like often is the case in python, multiple libraries already have an implementation for neural networks using the backpropagation algorithm, and TensorFlow is one such library. You are welcome to experiment with TensorFlow neural networks in this web application (http://playground.tensorflow.org/), which lets you define different properties of the network and run it, visualizing the output. We will now turn to an example of how we can use TensorFlow to perform the task we discussed last lecture: distinguishing counterfeit notes from genuine notes.

```
import csv
import tensorflow as tf
from sklearn.model_selection import train_test_split
```

We import TensorFlow and call it tf (to make the code shorter).

```
# Read data in from file
with open("banknotes.csv") as f:
    reader = csv.reader(f)
    next(reader)

    data = []
    for row in reader:
        data.append({
            "evidence": [float(cell) for cell in row[:4]],
            "label": 1 if row[4] == "0" else 0
        })

# Separate data into training and testing groups
```

```
evidence = [row["evidence"] for row in data]
labels = [row["label"] for row in data]
X_training, X_testing, y_training, y_testing = train_test_split(
    evidence, labels, test_size=0.4
)
```

We provide the CSV data to the model. Our work is often required in making the data fit the format that the library requires. The difficult part of actually coding the model is already implemented for us.

```
# Create a neural network
model = tf.keras.models.Sequential()
```

Keras is an api that different machine learning algorithms access. A sequential model is one where layers follow each other (like the ones we have seen so far).

```
# Add a hidden layer with 8 units, with ReLU activation
model.add(tf.keras.layers.Dense(8, input_shape=(4,), activation="relu"))
```

A dense layer is one where each node in the current layer is connected to all the nodes from the previous layer. In generating our hidden layers we create 8 dense layers, each having 4 input neurons, using the ReLU activation function mentioned above.

```
# Add output layer with 1 unit, with sigmoid activation
model.add(tf.keras.layers.Dense(1, activation="sigmoid"))
```

In our output layer, we want to create one dense layer that uses a sigmoid activation function, an activation function where the output is a value between 0 and 1.

```
# Train neural network
model.compile(
    optimizer="adam",
    loss="binary_crossentropy",
    metrics=["accuracy"]
)
model.fit(X_training, y_training, epochs=20)

# Evaluate how well model performs
model.evaluate(X_testing, y_testing, verbose=2)
```

Finally, we compile the model, specifying which algorithm should optimize it, what type of loss function we use, and how we want to measure its success (in our case, we are interested in the accuracy of the output). Finally, we fit the model on the training data with 20 repetitions (epochs), and then evaluate it on the testing data.

Computer Vision

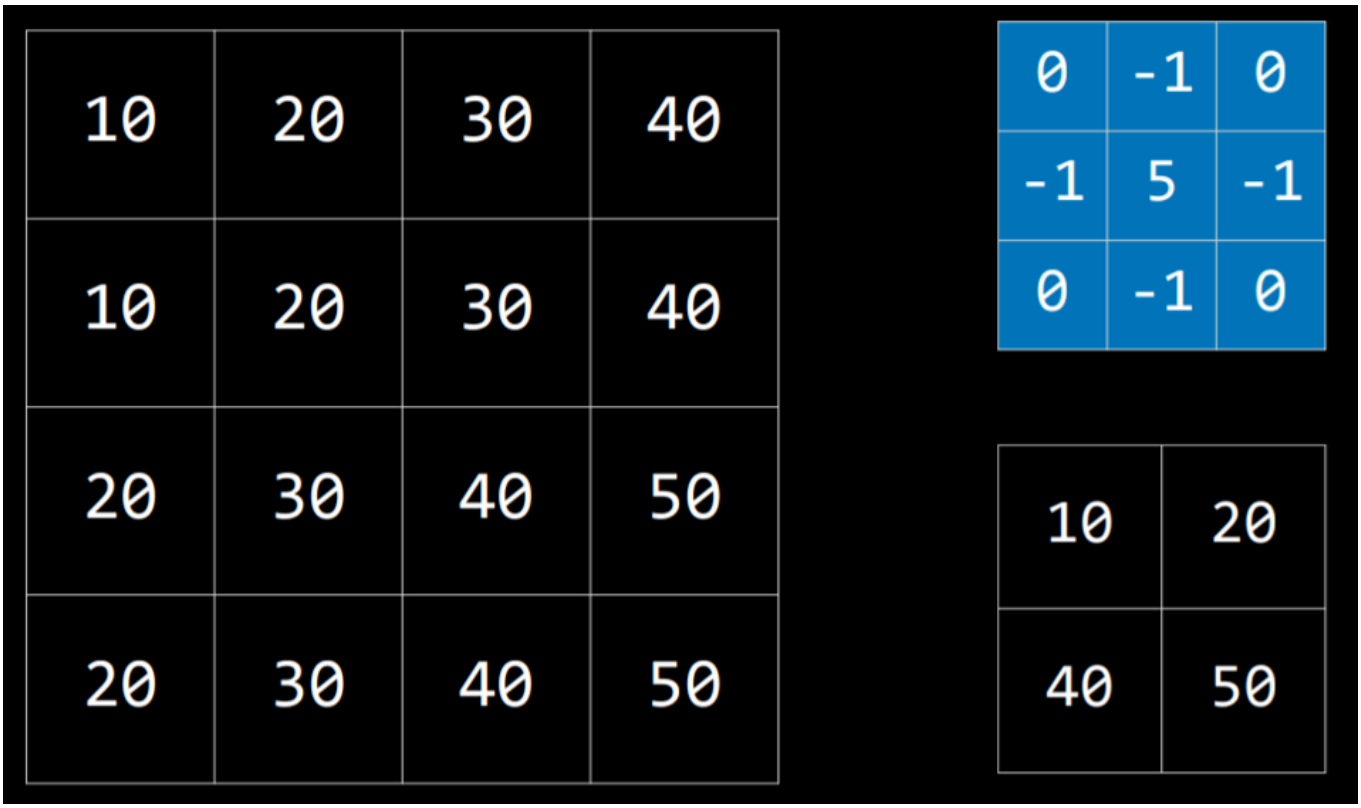
Computer vision encompasses the different computational methods for analyzing and understanding digital images, and it is often achieved using neural networks. For example, computer vision is used when social media employs face recognition to automatically tag people in pictures. Other examples are handwriting recognition and self-driving cars.

Images consist of pixels, and pixels are represented by three values that range from 0 to 255, one for red, one for green and one for blue. These values are often referred to with the acronym RGB. We can use this to create a neural network where each color value in each pixel is an input, where we have some hidden layers, and the output is some number of units that tell us what it is that was shown in the image. However, there are a few drawbacks to this approach. First, by breaking down the image into pixels and the values of their colors, we can't use the structure of the image as an aid. That is, as humans, if we see a part of a face we know to expect to see the rest of the face, and this quickens computation. We want to be able to use a similar advantage in our neural networks. Second, the sheer number of inputs is very big, which means that we will have to calculate a lot of weights.

Image Convolution

Image convolution is applying a filter that adds each pixel value of an image to its neighbors, weighted according to a kernel matrix. Doing so alters the image and can help the neural network process it.

Let's consider the following example:



The kernel is the blue matrix, and the image is the big matrix on the left. The resulting filtered image is the small matrix on the bottom right. To filter the image with the kernel, we start with the pixel with value 20 in the top-left of the image (coordinates 1,1). Then, we will multiply all the values around it by the corresponding value in the kernel and sum them up (10*0 + 20*(-1) + 30*0 + 10*(-1) + 20*5 + 30*(-1) + 20*0 + 30*(-1) + 40*0), producing the value 10. Then we will do the same for the pixel on the right (30), the pixel below the first one (30), and the pixel to the right of this one (40). This produces a filtered image with the values we see on the bottom right.

Different kernels can achieve different tasks. For edge detection, the following kernel is often used:

The idea here is that when the pixel is similar to all its neighbors, they should cancel each other, giving a value of 0. Therefore, the more similar the pixels, the darker the part of the image, and the more different they are the lighter it is. Applying this kernel to an image (left) results in an image with pronounced edges (right):

Let's consider an implementation of image convolution. We are using the PIL library (stands for Python Imaging Library) that can do most of the hard work for us.

```
import math
import sys

from PIL import Image, ImageFilter

# Ensure correct usage
if len(sys.argv) != 2:
    sys.exit("Usage: python filter.py filename")

# Open image
image = Image.open(sys.argv[1]).convert("RGB")

# Filter image according to edge detection kernel
filtered = image.filter(ImageFilter.Kernel(
    size=(3, 3),
    kernel=[-1, -1, -1, -1, 8, -1, -1, -1, -1],
    scale=1
))

# Show resulting image
filtered.show()
```

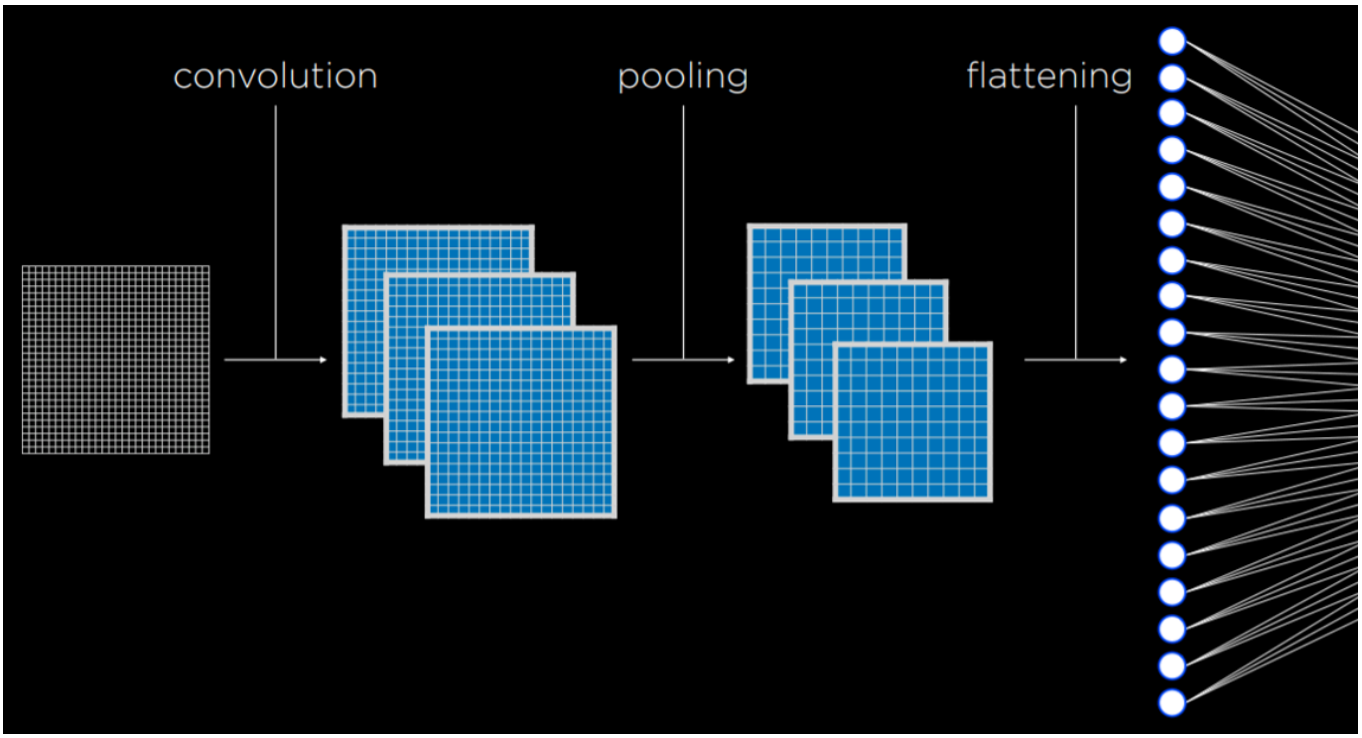
Still, processing the image in a neural network is computationally expensive due to the number of pixels that serve as input to the neural network. Another way to go about this is **Pooling**, where the size of the input is reduced by sampling from regions in the input. Pixels that are next to each other belong to the same area in the image, which means that they are likely to be similar. Therefore, we can take one pixel to represent a whole area. One way of doing this is with **Max-Pooling**, where the selected pixel is the one with the highest value of all others in the same region. For example, if we divide the left square (below) into four 2X2 squares, by max-pooling from this input, we get the small square on the right.

30	40	80	90
20	50	100	110
0	10	20	30
10	20	40	30

50	110
20	40

Convolutional Neural Networks

A convolutional neural network is a neural network that uses convolution, usually for analyzing images. It starts by applying filters that can help distill some features of the image using different kernels. These filters can be improved in the same way as other weights in the neural network, by adjusting their kernels based on the error of the output. Then, the resulting images are pooled, after which the pixels are fed to a traditional neural network as inputs (a process called **flattening**).



The convolution and pooling steps can be repeated multiple times to extract additional features and reduce the size of the input to the neural network. One of the benefits of these processes is that, by convoluting and pooling, the neural network becomes less sensitive to variation. That is, if the same picture is taken from slightly different angles, the input for convolutional neural network will be similar, whereas, without convolution and pooling, the input from each image would be vastly different.

In code, a convolutional neural network doesn't differ by much from a traditional neural network. TensorFlow offers datasets to test our models on. We will be using MNIST, which contains pictures of black and white handwritten digits. We will train our convolutional neural network to recognize digits.

```
import sys
import tensorflow as tf

# Use MNIST handwriting dataset
mnist = tf.keras.datasets.mnist

# Prepare data for training
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
y_train = tf.keras.utils.to_categorical(y_train)
y_test = tf.keras.utils.to_categorical(y_test)
x_train = x_train.reshape(
    x_train.shape[0], x_train.shape[1], x_train.shape[2], 1
)
x_test = x_test.reshape(
    x_test.shape[0], x_test.shape[1], x_test.shape[2], 1
)

# Create a convolutional neural network
model = tf.keras.models.Sequential([

    # Convolutional layer. Learn 32 filters using a 3x3 kernel
    tf.keras.layers.Conv2D(
        32, (3, 3), activation="relu", input_shape=(28, 28, 1)
    ),

    # Max-pooling layer, using 2x2 pool size
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),

    # Flatten units
    tf.keras.layers.Flatten(),

    # Add a hidden layer with dropout
    tf.keras.layers.Dense(128, activation="relu"),
    tf.keras.layers.Dropout(0.5),

    # Add an output layer with output units for all 10 digits
    tf.keras.layers.Dense(10, activation="softmax")
])
```

```
# Train neural network
model.compile(
    optimizer="adam",
    loss="categorical_crossentropy",
    metrics=["accuracy"]
)
model.fit(x_train, y_train, epochs=10)

# Evaluate neural network performance
model.evaluate(x_test,  y_test, verbose=2)
```

Since the model takes time to train, we can save the already trained model to use it later.

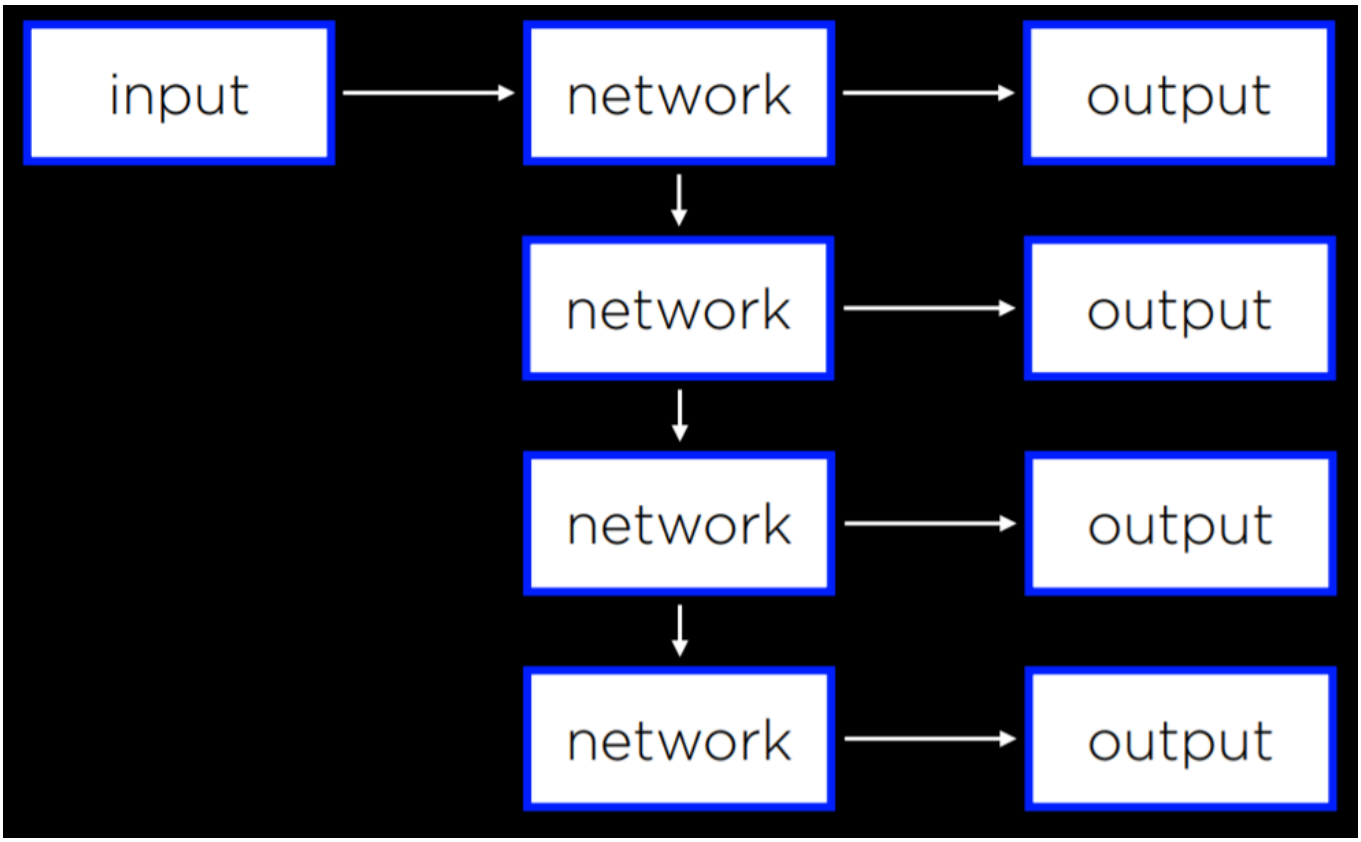
```
# Save model to file
if len(sys.argv) == 2:
    filename = sys.argv[1]
    model.save(filename)
    print(f"Model saved to {filename}.")
```

Now, if we run a program that receives hand-drawn digits as input, it will be able to classify and output the digit using the model. For an implementation of such a program, refer to recognition.py in the source code for this lecture.

Recurrent Neural Networks

Feed-Forward Neural Networks are the type of neural networks that we have discussed so far, where input data is provided to the network, which eventually produces some output. A diagram of how feed-forward neural networks work can be seen below.

As opposed to that, **Recurrent Neural Networks** consist of a non-linear structure, where the network uses its own output as input. For example, Microsoft's captionbot (https://www.captionbot.ai) is capable of describing the content of an image with words in a sentence. This is different from classification in that the output can be of varying length based on the properties of the image. While feed-forward neural networks are incapable of varying the number of outputs, recurrent neural networks are capable to do that due to their structure. In the captioning task, a network would process the input to produce an output, and then continue processing from that point on, producing another output, and repeating as much as necessary.



Recurrent neural networks are helpful in cases where the network deals with sequences and not a single individual object. Above, the neural network needed to produce a sequence of words. However, the same principle can be applied to analyzing video files, which consist of a sequence of images, or in translation tasks, where a sequence of inputs (words in the source language) is processed to produce a sequence of outputs (words in the target language).

CS50's Introduction to Artificial Intelligence with Python

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

Brian Yu (https://brianyu.me)
brian@cs.harvard.edu

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://www.reddit.com/user/davidjmalan) (https://www.threads.net/@davidjmalan)
(https://twitter.com/davidjmalan)

Lecture 6

These notes reflect the new version of Lecture 6, released on 14 August 2023. If you watched the prior version of the lecture and wish to see its notes, **click here**.

Language

So far in the course, we needed to shape tasks and data such that an AI will be able to process them. Today, we will look at how an AI can be constructed to process human language.

Natural Language Processing spans all tasks where the AI gets human language as input. The following are a few examples of such tasks:

- automatic summarization, where the AI is given text as input and it produces a summary of the text as output.
- information extraction, where the AI is given a corpus of text and the AI extracts data as output.

- language identification, where the AI is given text and returns the language of the text as output.
- machine translation, where the AI is given a text in the origin language and it outputs the translation in the target language.
- named entity recognition, where the AI is given text and it extracts the names of the entities in the text (for example, names of companies).
- speech recognition, where the AI is given speech and it produces the same words in text.
- text classification, where the AI is given text and it needs to classify it as some type of text.
- word sense disambiguation, where the AI needs to choose the right meaning of a word that has multiple meanings (e.g. bank means both a financial institution and the ground on the sides of a river).

Syntax and Semantics

Syntax is sentence structure. As native speakers of some human language, we don't struggle with producing grammatical sentences and flagging non-grammatical sentences as wrong. For example, the sentence "Just before nine o'clock Sherlock Holmes stepped briskly into the room" is grammatical, whereas the sentence "Just before Sherlock Holmes nine o'clock stepped briskly the room" is non-grammatical. Syntax can be grammatical and ambiguous at the same time, as in "I saw the man with the telescope." Did I see (the man with the telescope) or did I see (the man), doing so by looking through the telescope? To be able to parse human speech and produce it, the AI needs to command syntax.

Semantics is the meaning of words or sentences. While the sentence "Just before nine o'clock Sherlock Holmes stepped briskly into the room" is syntactically different from "Sherlock Holmes stepped briskly into the room just before nine o'clock," their content is effectively identical. Similarly, although the sentence "A few minutes before nine, Sherlock Holmes walked quickly into the room" uses different words from the previous sentences, it still carries a very similar meaning. Moreover, a sentence can be perfectly grammatical while being completely nonsensical, as in Chomsky's example, "Colorless green ideas sleep furiously." To be able to parse human speech and produce it, the AI needs to command semantics.

Context-Free Grammar

Formal Grammar is a system of rules for generating sentences in a language. In **Context-Free Grammar**, the text is abstracted from its meaning to represent the structure of the sentence using formal grammar. Let's consider the following example sentence:

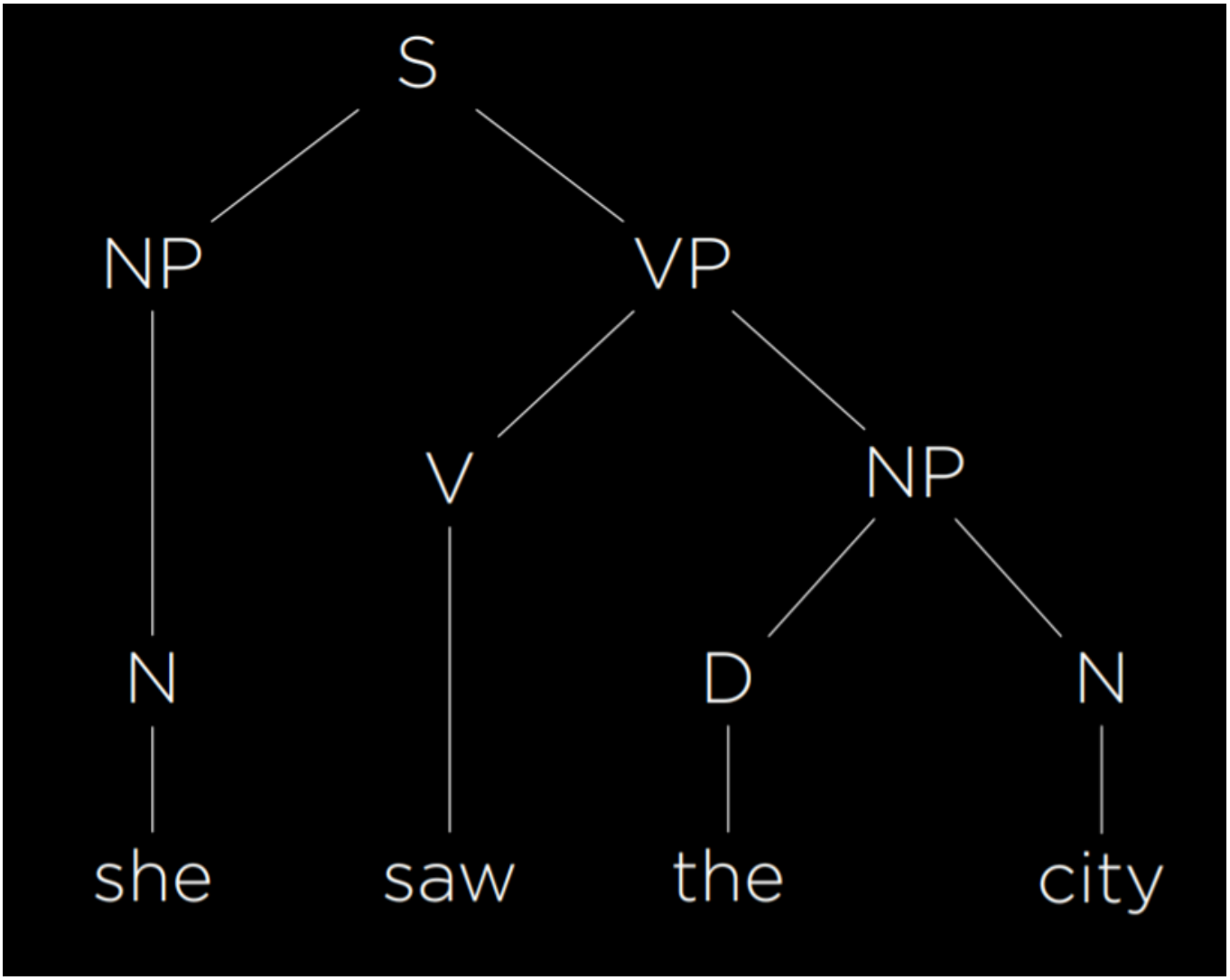
- She saw the city.

This is a simple grammatical sentence, and we would like to generate a syntax tree representing its structure.

We start by assigning each word its part of speech. *She* and *city* are nouns, which we will mark as N. *Saw* is a verb, which we will mark as V. *The* is a determiner, marking the following noun as definite or indefinite, and we will mark it as D. Now, the above sentence can be rewritten as

- N V D N

So far, we have abstracted each word from its semantic meaning to its part of speech. However, words in a sentence are connected to each other, and to understand the sentence we must understand how they connect. A noun phrase (NP) is a group of words that connect to a noun. For example, the word *she* is a noun phrase in this sentence. In addition, the words *the city* also form a noun phrase, consisting of a determiner and a noun. A verb phrase (VP) is a group of words that connect to a verb. The word *saw* is a verb phrase in itself. However, the words *saw the city* also make a verb phrase. In this case, it is a verb phrase consisting of a verb and a noun phrase, which in turn consists of a determiner and a noun. Finally, the whole sentence (S) can be represented as follows:



Using formal grammar, the AI is able to represent the structure of sentences. In the grammar we have described, there are enough rules to represent the simple sentence above. To represent more complex sentences, we will have to add more rules to our formal grammar.

nltk

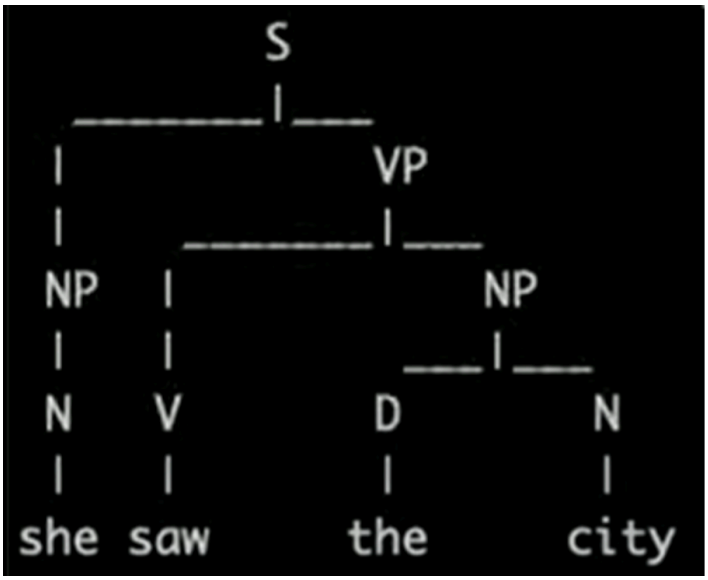
As is often the case in Python, multiple libraries have been written to implement the idea above. nltk (Natural Language Toolkit) is one such library. To analyze the sentence from above, we will provide the algorithm with rules for the grammar:

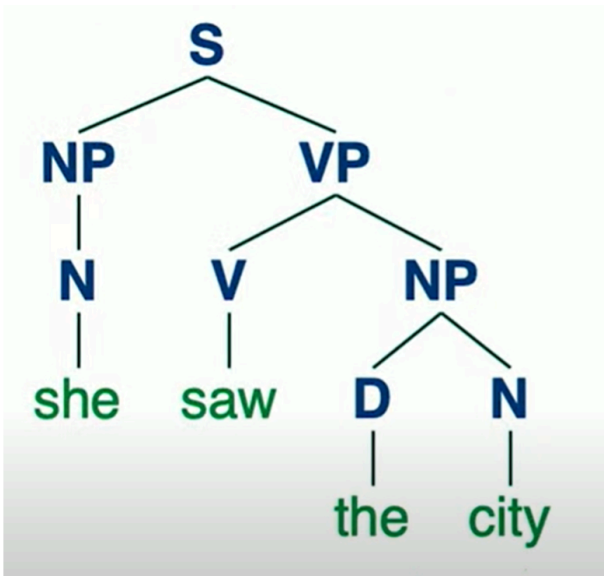
```
import nltk

grammar = nltk.CFG.fromstring("""
    S -> NP VP

    NP -> D N | N
    VP -> V | V NP

    D -> "the" | "a"
    N -> "she" | "city" | "car"
    V -> "saw" | "walked"
""")

parser = nltk.ChartParser(grammar)
```

Similar to what we did above, we define what possible components could be included in others. A sentence can include a noun phrase and a verb phrase, while the phrases themselves can consist of other phrases, nouns, verbs, etc., and, finally, each part of speech spans some words in the language.

```
sentence = input("Sentence: ").split()
try:
    for tree in parser.parse(sentence):
        tree.pretty_print()
        tree.draw()
except ValueError:
    print("No parse tree possible.")
```

After giving the algorithm an input sentence split into a list of words, the function prints the resulting syntactic tree (pretty_print) and also generates a graphic representation (draw).





n-grams

An *n*-gram is a sequence of *n* items from a sample of text. In a **character *n*-gram**, the items are characters, and in a **word *n*-gram** the items are words. A *unigram*, *bigram*, and *trigram* are sequences of one, two, and three items. In the following sentence, the first three *n*-grams are "how often have," "often have I," and "have I said."

"How often have I said to you that when you have eliminated the impossible whatever remains, however improbable, must be the truth?"

n-grams are useful for text processing. While the AI hasn't necessarily seen the whole sentence before, it sure has seen parts of it, like "have I said." Since some words occur together more often then others, it is possible to also predict the next word with some probability. For example, your smartphone suggests words to you based on a probability distribution derived from the last few words you typed. Thus, a helpful step in natural language processing is breaking the sentence into n-grams.

Tokenization

Tokenization is the task of splitting a sequence of characters into pieces (tokens). Tokens can be words as well as sentences, in which case the task is called **word tokenization** or **sentence tokenization**. We need tokenization to be able to look at *n*-grams, since those rely on sequences of tokens. We start by splitting the text into words based on the space character. While this is a good start, this method is imperfect because we end up with words with punctuation, such as "remains,". So, for example, we can remove punctuation. However, then we face additional challenges, such as words with apostrophes (e.g. "o'clock") and hyphens (e.g. "pearl-grey). Additionally, some punctuation is important for sentence structure, like periods. However, we need to be able to tell apart between

a period at the end of the word “Mr.” and a period in the end of the sentence. Dealing with these questions is the process of tokenization. In the end, once we have our tokens, we can start looking at *n*-grams.

Markov Models

As discussed in previous lectures, Markov models consist of nodes, the value of each of which has a probability distribution based on a finite number of previous nodes. Markov models can be used to generate text. To do so, we train the model on a text, and then establish probabilities for every *n*-th token in an *n*-gram based on the *n* words preceding it. For example, using trigrams, after the Markov model has two words, it can choose a third one from a probability distribution based on the first two. Then, it can choose a fourth word from a probability distribution based on the second and third words. To see an implementation of such a model using nltk, refer to generator.py in the source code, where our model learns to generate Shakespeare-sounding sentences. Eventually, using Markov models, we are able to generate text that is often grammatical and sounding superficially similar to human language output. However, these sentences lack actual meaning and purpose.

Bag-of-Words Model

Bag-of-words is a model that represents text as an unordered collection of words. This model ignores syntax and considers only the meanings of the words in the sentence. This approach is helpful in some classification tasks, such as sentiment analysis (another classification task would be distinguishing regular email from spam email). Sentiment analysis can be used, for instance, in product reviews, categorizing reviews as positive or negative. Consider the following sentences:

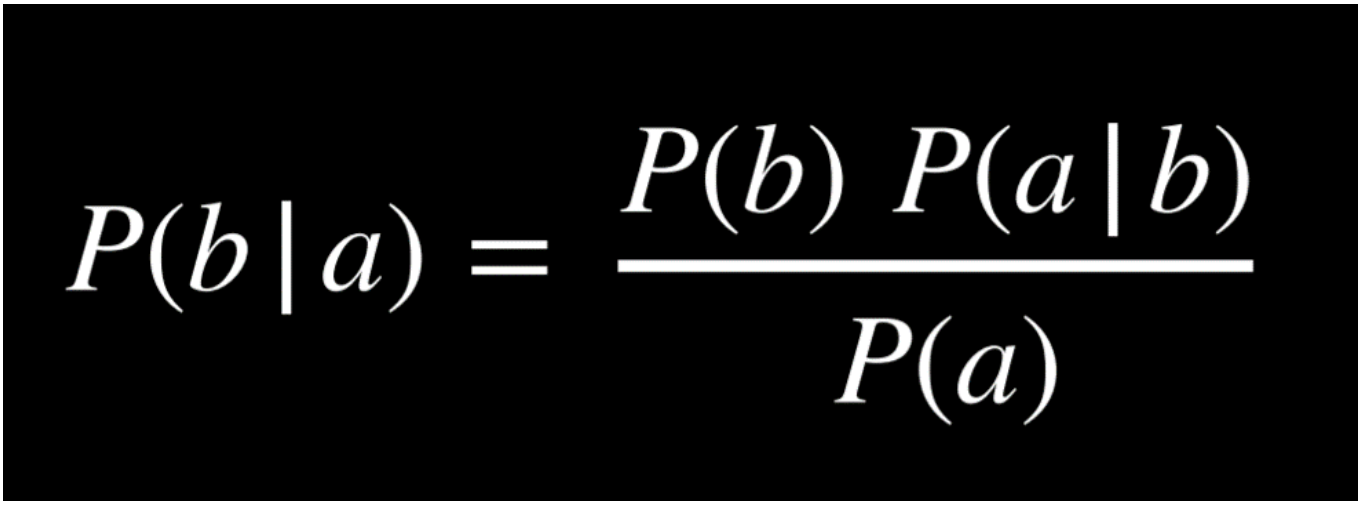
1. “My grandson loved it! So much fun!”
2. “Product broke after a few days.”
3. “One of the best games I’ve played in a long time.”
4. “Kind of cheap and flimsy, not worth it.”

Based only on the words in each sentence and ignoring the grammar, we can see that sentences 1 and 3 are positive (“loved,” “fun,” “best”) and sentences 2 and 4 are negative (“broke,” “cheap,” “flimsy”).

Naive Bayes

Naive Bayes is a technique that’s can be used in sentiment analysis with the bag-of-words model. In sentiment analysis, we are asking “What is the probability that the sentence is positive/negative

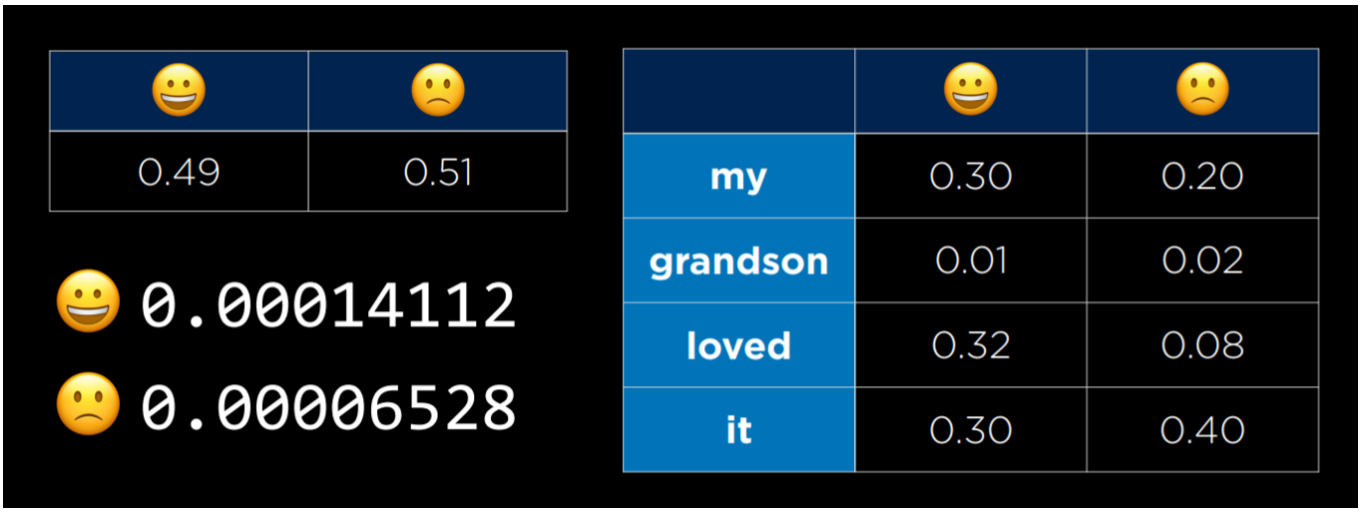
given the words in the sentence." Answering this question requires computing conditional probability, and it is helpful to recall Bayes' rule from lecture 2:



Now, we would like to use this formula to find P(sentiment | text), or, for example, P(positive | "my grandson loved it"). We start by tokenizing the input, such that we end up with P(positive | "my", "grandson", "loved", "it"). Applying Bayes' ruled directly, we get the following expression: P("my", "grandson", "loved", "it" | positive)*P(positive)/P("my", "grandson", "loved", "it"). This complicated expression will give us the precise answer to P(positive | "my", "grandson", "loved", "it").

However, we can simplify the expression if we are willing to get an answer that's not equal, but proportional to P(positive | "my", "grandson", "loved", "it"). Later on, knowing that the probability distribution needs to sum up to 1, we can normalize the resulting value into an exact probability. This means that we can simplify the expression above to the numerator only: P("my", "grandson", "loved", "it" | positive)*P(positive). Again, we can simplify this expression based on the knowledge that a conditional probability of *a* given *b* is proportional to the joint probability of *a* and *b*. Thus, we get the following expression for our probability: P(positive, "my", "grandson", "loved", "it")*P(positive). Calculating this joint probability, however, is complicated, because the probability of each word is conditioned on the probabilities of the words preceding it. It requires us to compute P(positive)*P("my" | positive)*P("grandson" | positive, "my")*P(loved | positive, "my", "grandson")*P("it" | positive, "my", "grandson", "loved").

Here is where we use Bayes' rules naively: we assume that the probability of each word is independent from other words. This is not true, but despite this imprecision, Naive Bayes' produces a good sentiment estimate. Using this assumption, we end up with the following probability: P(positive)*P("my" | positive)*P("grandson" | positive)*P("loved" | positive)*P("it" | positive), which is not that difficult to calculate. P(positive) = the number of all positive samples divided by the number of total samples. P("loved" | positive) is equal to the number of positive samples with the word "loved" divided by the number of positive samples. Let's consider the example below, with smiling and frowning emojies substituting the words "positive" and "negative":



On the right we are seeing a table with the conditional probabilities of each word on the left occurring in a sentence given that the sentence is positive or negative. In the small table on the left we are seeing the probability of a positive or a negative sentence. On the bottom left we are seeing the resulting probabilities following the computation. At this point, they are in proportion to each other, but they don't tell us much in terms of probabilities. To get the probabilities, we need to normalize the values, arriving at P(positive) = 0.6837 and P(negative) = 0.3163. The strength of naive Bayes is that it is sensitive to words that occur more often in one type of sentence than in the other. In our case, the word "loved" occurs much more often in positive sentences, which makes the whole sentence more likely to be positive than negative. To see an implementation of sentiment assessment using Naive Bayes with the nltk library, refer to sentiment.py.

One problem that we can run into is that some words may never appear in a certain type of sentence. Suppose none of the positive sentences in our sample had the word "grandson." Then, P("grandson" | positive) = 0, and when computing the probability of the sentence being positive we will get 0. However, this is not the case in reality (not all sentences mentioning grandsons are negative). One way to go about this problem is with **Additive Smoothing**, where we add a value α to each value in our distribution to smooth the data. This way, even if a certain value is 0, by adding α to it we won't be multiplying the whole probability for a positive or negative sentence by 0. A specific type of additive smoothing, **Laplace Smoothing** adds 1 to each value in our distribution, pretending that all values have been observed at least once.

Word Representation

We want to represent word meanings in our AI. As we've seen before, it is convenient to provide input to the AI in the form of numbers. One way to go about this is by using **One-Hot Representation**, where each word is represented with a vector that consists of as many values as we have words. Except for a single value in the vector that is equal to 1, all other values are equal to 0. How we can differentiate words is by which of the values is 1, ending up with a unique vector per word. For example, the sentence "He wrote a book" can be represented as four vectors:

- [1, 0, 0, 0] (he)
- [0, 1, 0, 0] (wrote)
- [0, 0, 1, 0] (a)
- [0, 0, 0, 1] (book)

However, while this representation works in a world with four words, if we want to represent words from a dictionary, when we can have 50,000 words, we will end up with 50,000 vectors of length 50,000. This is incredibly inefficient. Another problem in this kind of representation is that we are unable to represent similarity between words like "wrote" and "authored." Instead, we turn to the idea of **Distributed Representation**, where meaning is distributed across multiple values in a vector. With distributed representation, each vector has a limited number of values (much less than 50,000), taking the following form:

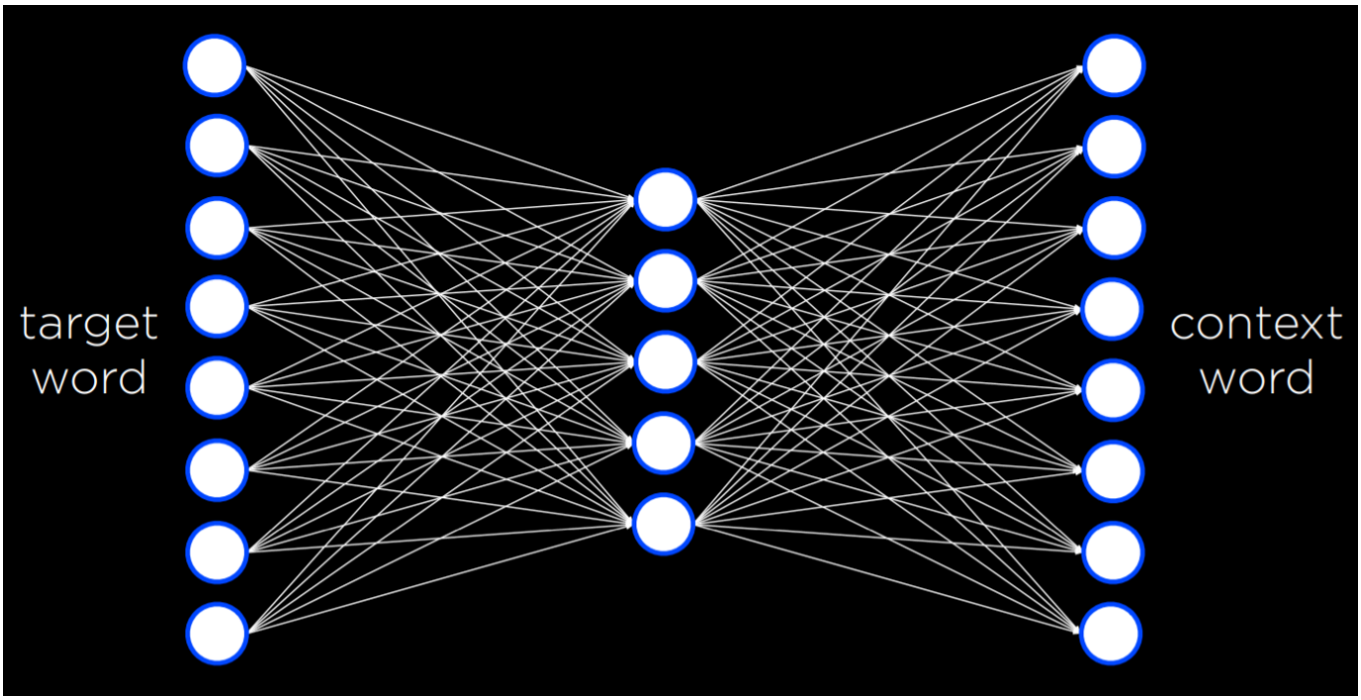
- [-0.34, -0.08, 0.02, -0.18, ...] (he)
- [-0.27, 0.40, 0.00, -0.65, ...] (wrote)
- [-0.12, -0.25, 0.29, -0.09, ...] (a)
- [-0.23, -0.16, -0.05, -0.57, ...] (book)

This allows us to generate unique values for each word while using smaller vectors. Additionally, now we are able to represent similarity between words by how different the values in their vectors are.

"You shall know a word by the company it keeps" is an idea by J. R. Firth, an English linguist. Following this idea, we can come to define words by their adjacent words. For example, there are limited words that we can use to complete the sentence "for ___ he ate." These words are probably words like "breakfast," "lunch," and "dinner." This brings us to the conclusion that by considering the environment in which a certain word tends to appear, we can infer the meaning of the word.

word2vec

word2vec is an algorithm for generating distributed representations of words. It does so by **Skip-Gram Architecture**, which is a neural network architecture for predicting context given a target word. In this architecture, the neural network has an input unit for every target word. A smaller, single hidden layer (e.g. 50 or 100 units, though this number is flexible) will generate values that represent the distributed representations of words. Every unit in this hidden layer is connected to every unit in the input layer. The output layer will generate words that are likely to appear in a similar context as the target words. Similar to what we saw in last lecture, this network needs to be trained with a training dataset using the backpropagation algorithm.



This neural network turns out to be quite powerful. In the end, of the process, every word ends up being just a vector, or a sequence of numbers. For example,

book: [-0.226776 -0.155999 -0.048995 -0.569774 0.053220 0.124401 -0.091108 -0.606255 -0.114630 0.473384 0.061061 0.551323 -0.245151 -0.014248 -0.210003 0.316162 0.340426 0.232053 0.386477 -0.025104 -0.024492 0.342590 0.205586 -0.554390 -0.037832 -0.212766 -0.048781 -0.088652 0.042722 0.000270 0.356324 0.212374 -0.188433 0.196112 -0.223294 -0.014591 0.067874 -0.448922 -0.290960 -0.036474 -0.148416 0.448422 0.016454 0.071613 -0.078306 0.035400 0.330418 0.293890 0.202701 0.555509 0.447660 -0.361554 -0.266283 -0.134947 0.105315 0.131263 0.548085 -0.195238 0.062958 -0.011117 -0.226676 0.050336 -0.295650 -0.201271 0.014450 0.026845 0.403077 -0.221277 -0.236224 0.213415 -0.163396 -0.218948 -0.242459 -0.346984 0.282615 0.014165 -0.342011 0.370489 -0.372362 0.102479 0.547047 0.020831 -0.202521 -0.180814 0.035923 -0.296322 -0.062603 0.232734 0.191323 0.251916 0.150993 -0.024009 0.129037 -0.033097 0.029713 0.125488 -0.018356 -0.226277 0.437586 0.004913]

By themselves, these numbers don't mean much. But by finding which other words in the corpus have the most similar vectors, we can run a function that will generate the words that are the most similar to the word *book*. In the case of this network it will be: book, books, essay, memoir, essays, novella, anthology, blurb, autobiography, audiobook. This is not bad for a computer! Through a bunch of numbers that don't carry any specific meaning themselves, the AI is able to generate words that really are very similar to *book* not in letters or sounds, but in meaning! We can also compute the difference between words based on how different their vectors are. For example, the difference between *king* and *man* is similar to the difference between *queen* and *woman*. That is, if we add the difference between *king* and *man* to the vector for *woman*, the closest word to the resulting vector is *queen*! Similarly, if we add the difference between *ramen* and *japan* to *america*,

we get *burritos*. By using neural networks and distributed representations for words, we get our AI to understand semantic similarities between words in the language, bringing us one step closer to AIs that can understand and produce human language.

Neural Networks

Recall that a **neural network** takes some input, passes it to the network, and creates some output. By providing the network with training data, it can do more and more of an accurate job of translating the input into an output. Commonly, machine translation uses neural networks. In practice, when we are translating words, we want to translate a sentence or paragraph. Since a sentence is a fixed size, we run into the problem of translating a sequence to another sequence where sizes are not fixed. If you have ever had a conversation with an AI chatbot, it needs to understand a sequence of words and generate an appropriate sequence as output.

Recurrent neural networks can re-run the neural network multiple times, keeping track of a state that holds all relevant information. Input is taken into the network, creating a hidden state. Passing a second input into the encoder, along with the first hidden state, produces a new hidden state. This process is repeated until an end token is passed. Then, a decoding state begins, creating hidden state after hidden state until we get the final word and another end token. Some problems, however, arise. One problem in the encoder stage where all the information from the input stage must be stored in one final state. For large sequences, it's very challenging to store all that information into a single state value. It would be useful to somehow combine all the hidden states. Another problem is that some of the hidden states in the input sequence are more important than others. Could there be some way to know what states (or words) are more important than others?

Attention

Attention refers to the neural network's ability to decide what values are more important than others. In the sentence "What is the capital of Massachusetts," attention allows the neural network to decide what values it will pay attention to at each stage of generating the output sentence. Running such a calculation, the neural network will show that when generating the final word of the answer, "capital" and "Massachusetts" are the most important to pay attention to. By taking the attention score, multiplying them by the hidden state values generated by the network, and adding them up, the neural network will create a final context vector that the decoder can use to calculate the final word. A challenge that arises in calculations such as these is that recurrent neural networks require sequential training of word after word. This takes a lot of time. With the growth of large language models, they take longer and longer to train. A desire for parallelism has steadily grown as larger and larger datasets need to be trained. Hence, a new architecture has been introduced.

Transformers

Transformers is a new type of training architecture whereby each input word is passed through a neural network simultaneously. An input word goes into the neural network and is captured as an encoded representation. Because all words are fed into the neural network at the same time, word order could easily be lost. Accordingly, **position encoding** is added to the inputs. The neural network, therefore, will use both the word and the position of the word in the encoded representation. Additionally, a **self-attention** step is added to help define the context of the word being inputted. In fact, neural networks will often use multiple self-attention steps such that they can further understand the context. This process is repeated multiple times for each of the words in the sequence. What results are encoded representations that will be useful when it's time to decode the information.

In the decoding step, the previous output word and its positional encoding are given to multiple self-attention steps and the neural network. Additionally, multiple attention steps are fed the encoded representation from the encoding process and provided to the neural network. Hence, words are able to pay attention to each other. Further, parallel processing is possible, and the calculations are fast and accurate.

Summing Up

We have looked at artificial intelligence in a wide array of contexts. We looked at search problems in how AI can look for solutions. We looked at how AI represents knowledge and create knowledge. We looked at uncertainty when it does not know things for sure. We looked at optimization, maximizing and minimizing function. We looked at machine learning, finding patterns by looking at training data. We learned about neural networks and how they use weights to go from input to output. Today, we looked at language itself and how we can get the computer to understand our language. We have just scratched the surface of this process. We truly hope you enjoyed this journey with us. This was Introduction to Artificial Intelligence with Python.