

Introduction to Artificial Intelligence

Course syllabus, Fall 2024

Prof. Gilles Louppe
g.louppe@uliege.be

Us

This course is given by

- Theoretical lectures: Gilles Louppe
- Exercise sessions: Gilles Louppe, Gérôme Andry, and student instructors
- Programming projects: Arnaud Delaunoy

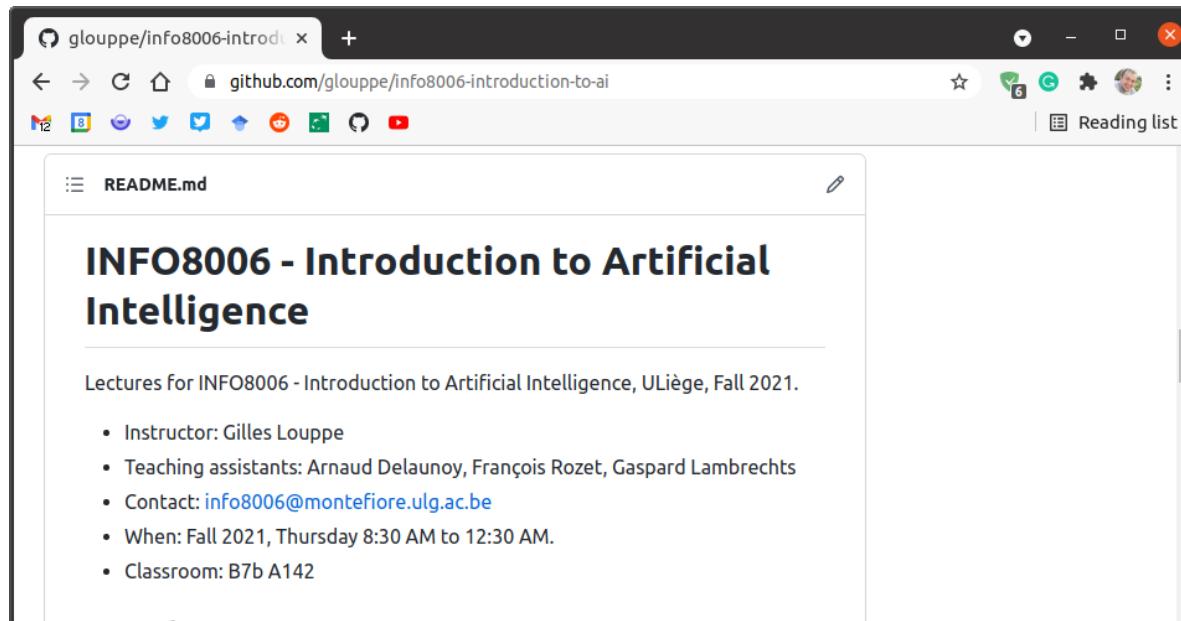
Feel free to contact us at info8006@montefiore.ulg.ac.be or on [Discord](#) for help.



Materials

The schedule and slides are available at github.com/glouppe/info8006-introduction-to-ai.

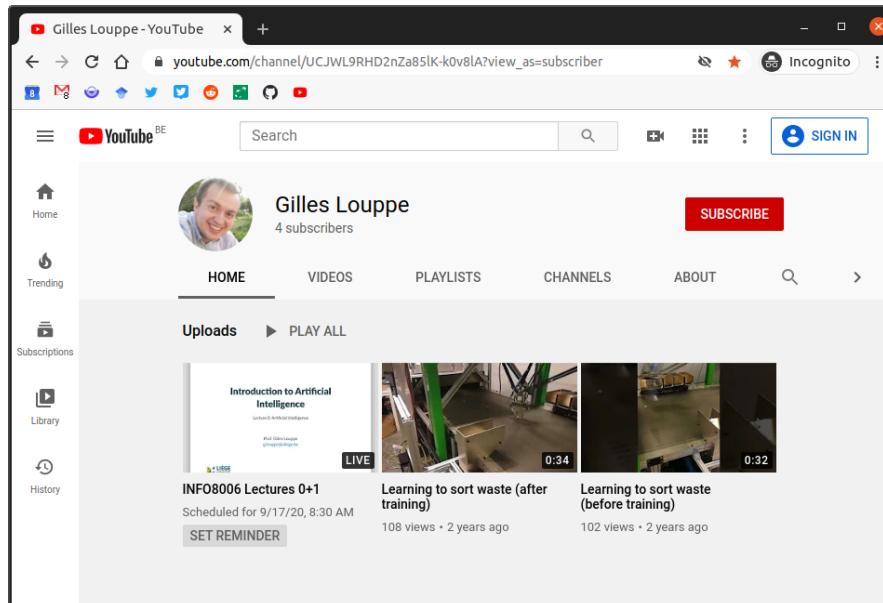
- In HTML and in PDFs.
- Minor updates up to the day before the lesson.



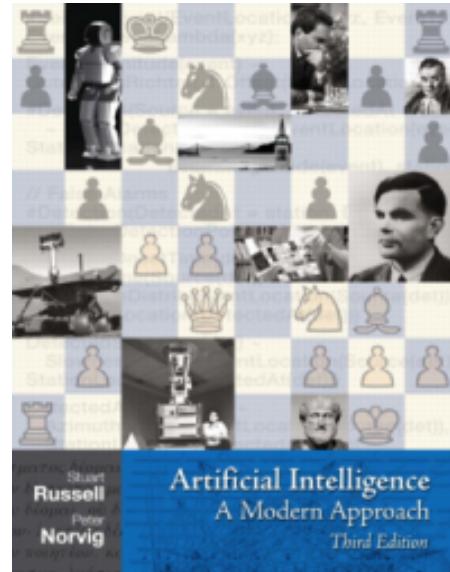
Videos

Videos from Fall 2020 are available at https://youtube.com/playlist?list=PLLqXZ_E-UXlybvRU7vgaYMTbxZdT73ZFD. They are not up-to-date with the current course, but they may still be useful.

Lectures will also be recorded and made available on [MyUnicast](#).



Textbook



The core content of this course is based on the following textbook:

Stuart Russel, Peter Norvig. "Artificial Intelligence: A Modern Approach", Third Edition, Global Edition.

This textbook is **recommended**. It covers both the theory and the exercises.

CS188

- Some lessons, exercises, and various other materials are partially adapted from [CS188 Introduction to Artificial Intelligence](#), from UC Berkeley.
- Cartoons that you will see in those slides were all originally made for CS188.

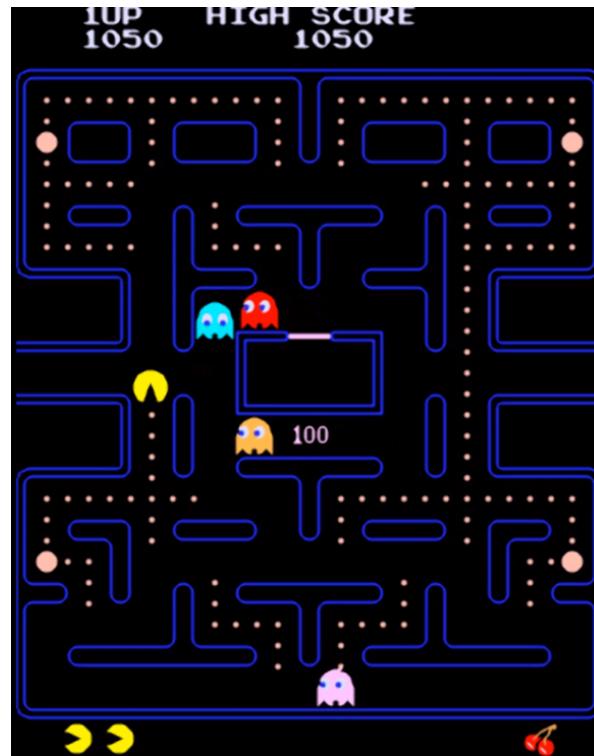


Exercise sessions

- Exercise sessions are held every week after the lecture.
- You will work on exercises by yourself or in small groups, with our help.
 - Your active participation is expected.
 - The exercises are designed to help you understand the materials and prepare for the exam.
 - Use this time to get answers to your questions.
- Solutions are provided for all exercises.

Programming projects

Implement an intelligent agent for playing [Pacman](#). The project will be divided into three parts, with increasing levels of difficulty.



Evaluation

- Written exam (60%)
- Programming projects (40%)
 - Project 0: 0% (tutorial of September 26)
 - Project 1: 20%
 - Project 2: 20%
 - Programming projects are **mandatory** for presenting the exam.

Evaluation of the projects

- Projects are evaluated **automatically** on Gradescope.
- Preliminary tests are provided to help you debug your code.
- However, the final grade will be based on additional tests that are not provided.
 - It is part of the learning experience to design your own tests.
 - It is not because the public tests pass that your code is correct.
- The efficiency of your code will also be taken into account, beyond its correctness.

Honor code

You may consult papers, books, online references, or publicly available implementations for ideas that you may want to adapt and incorporate into your projects, so long as you clearly cite your sources in your code and your writeup. *However, under no circumstances, may you base your project on someone else's implementation.* In particular, the use of large language models (e.g., ChatGPT, Github Copilot) is forbidden*.

Plagiarism is checked and sanctioned by a grade of 0. Cases of plagiarism will all be reported to the Faculty office.

*: Unless you build and train your own :-)

Introduction to Artificial Intelligence

Lecture 0: Artificial Intelligence

Prof. Gilles Louppe
g.louppe@uliege.be

ChatGPT PLUS

Help me pick
a gift for my dad who loves fishing

Brainstorm edge cases
for a function with birthdate as input, horoscope as ou...

Make up a story
about Sharky, a tooth-brushing shark superhero

Create a personal webpage for me
after asking me three questions

 Send a message



10:27 🏠 ⏱ 0.00 KB/S VOLTE 86 % 🔋

X



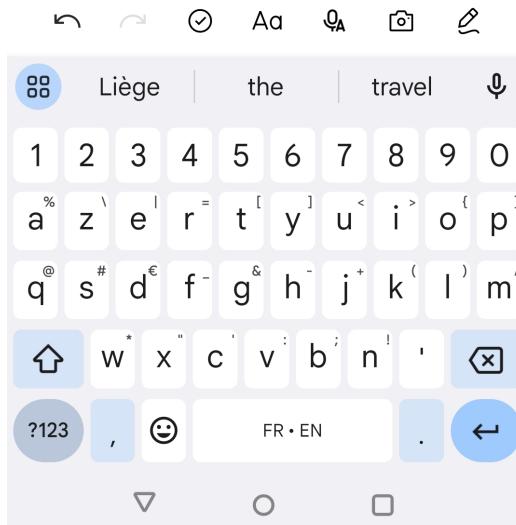
⋮

08/09, 10:26 | 45

So tell me, what would you recommend for a
1-day trip to |

One simple idea:

Guess the next word



In the 1960s, Armstrong ____

In the 1960s, Armstrong performed ____

In the 1960s, Armstrong performed a moonwalk ____

In the 1960s, Armstrong performed a moonwalk on the ____

In the 1960s, Armstrong performed a moonwalk on the lunar ____

In the 1960s, Armstrong performed a moonwalk on the lunar surface and said ____

This explains why large language models ...

- invent things and cannot cite sources;
- never produce the same answers;
- cannot count, compute, or reason*;
- can hardly correct their own mistakes once they have been made.

*: At least not with a vanilla transformer and a greedy decoding strategy.



Rock, Paper, Scissors with GPT-4o



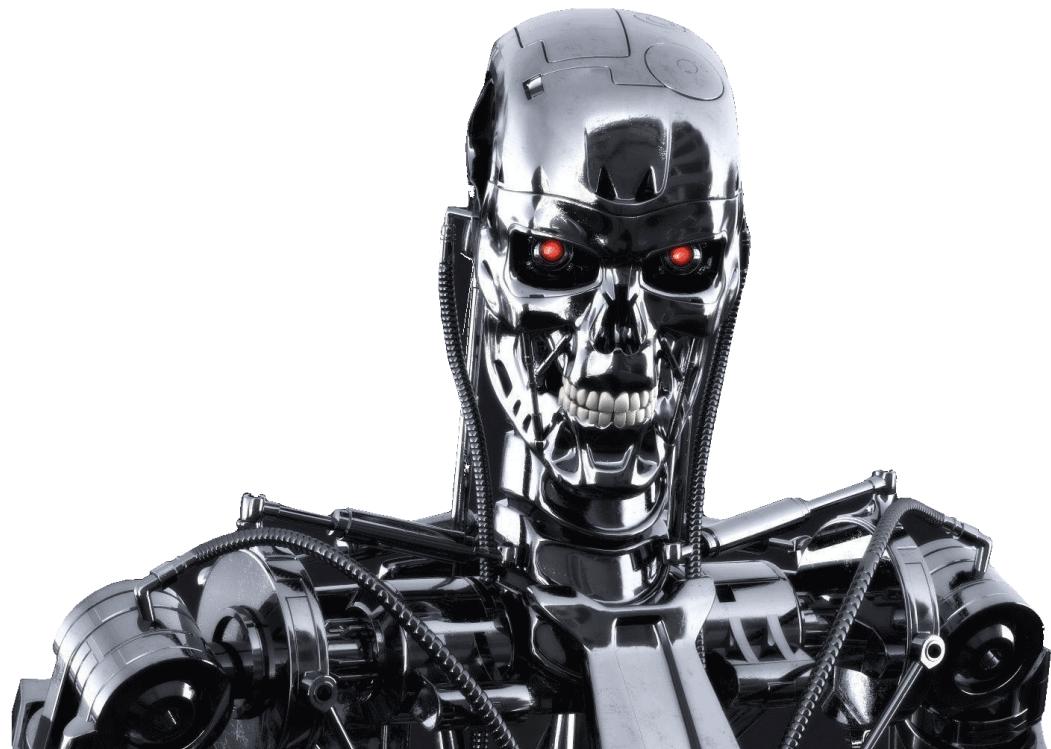
Later bekij...
Later



Delen
Share

Not just text, but also images and sounds.

Artificial Intelligence



"With artificial intelligence we are summoning the demon" -- Elon Musk, 2014.



"We're really closer to a smart washing machine than Terminator" -- Fei-Fei Li,
Director of Stanford AI Lab, 2017.

Rencontre avec Yann Le Cun, directeur de la recherche en AI chez...

DAILYMOTION PRO



2:39



Yann LeCun, 2018.



AI 'godfather' quits Google over dangers of ...



Later bekij...



Delen

'AI could
be smarter
than us'

BBC NEWS



Geoffrey Hinton, 2023.

IMAGINATION
IN ACTION

Yann LeCun | Imagination In Action | Davos ...



Later bekij...
[Later bekijken](#)

Delen
[Delen](#)



Yann LeCun, 2023.

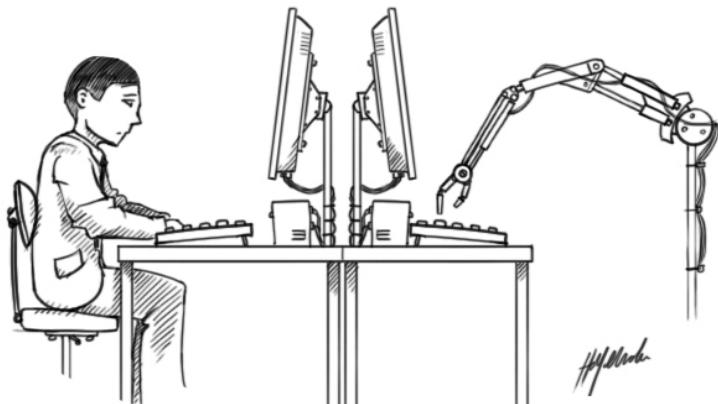
A definition of AI?



"Artificial intelligence is the science of making machines do things that would require intelligence if done by men." -- Marvin Minsky, 1968.

The Turing test

A computer passes the **Turing test** (aka the Imitation Game) if a human operator, after posing some written questions, cannot tell whether the written responses come from a person or from a computer.



*Can machines think?
(Alan Turing, 1950)*

An agent would not pass the Turing test without the following **requirements**:

- natural language processing
- knowledge representation
- automated reasoning
- machine learning
- computer vision (total Turing test)
- robotics (total Turing test)

Despite being proposed almost 70 years ago, the Turing test is **still relevant** today.

The Turing test tends to focus on **human-like errors**, **linguistic tricks**, etc.

However, it seems more important to study the **principles** underlying intelligence than to replicate an exemplar.



Aeronautics is not defined as the field of making machines
that fly so exactly like pigeons that they can fool even other pigeons.

A modern definition of AI

An 'AI system' is a machine-based system that is designed to operate with varying levels of autonomy and that may exhibit adaptiveness after deployment, and that, for explicit or implicit objectives, infers, from the input it receives, how to generate outputs such as predictions, content, recommendations, or decisions that can influence physical or virtual environments. -- European AI Act, Article 3, 2024.

A short history of AI

1940-1950: Early days

- 1943: McCulloch and Pitts: Boolean circuit model of the brain.
- 1950: Turing's "Computing machinery and intelligence".

1950-1970: Excitement and expectations

- 1950s: Early AI programs, including Samuel's checkers program, Newell and Simon's Logic Theorist and Gelernter's Geometry Engine.
- 1956: Dartmouth meeting: "Artificial Intelligence" adopted.
- 1958: Rosenblatt invents the perceptron.
- 1965: Robinson's complete algorithm for logical reasoning.
- 1966-1974: AI discovers computational complexity.



The Dartmouth workshop (1956)

The study is to proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it.



The Thinking Machine (Artificial Intelligenc...



Later bekij...



Delen



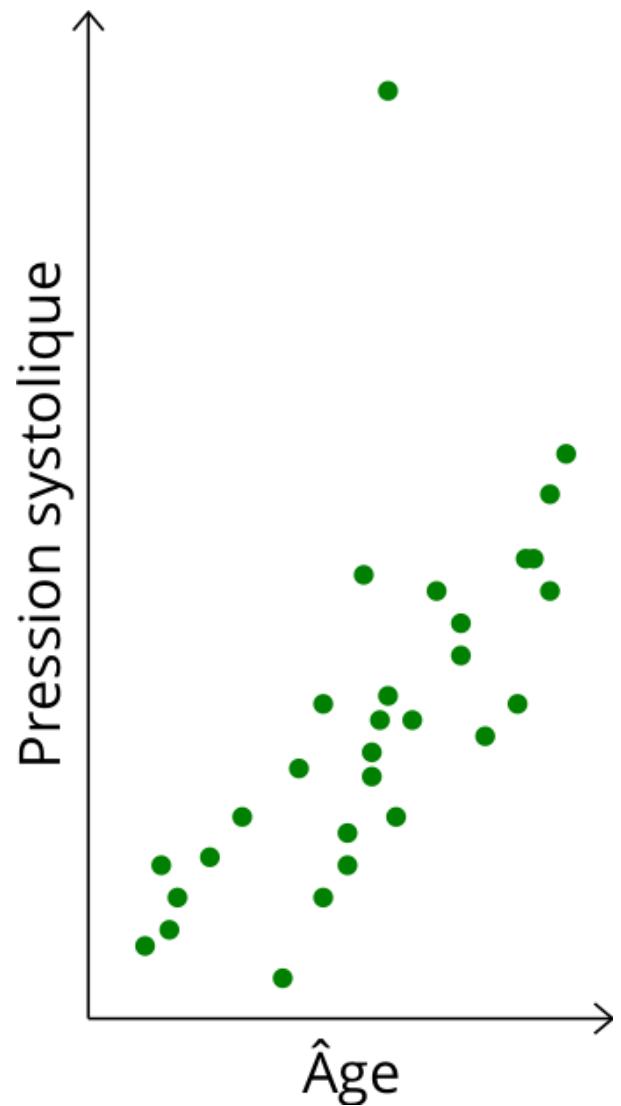
1970-1990: Knowledge-based approaches

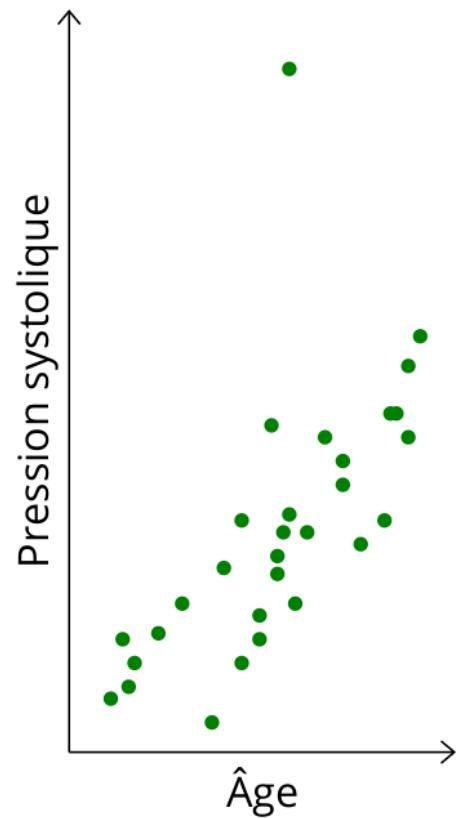
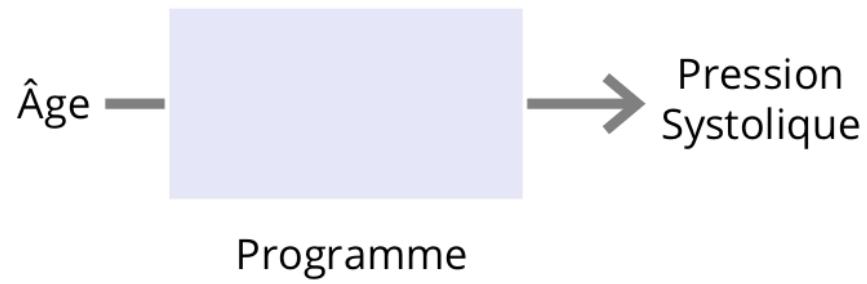
- 1969: Neural network research almost disappears after Minsky and Papert's book (1st AI winter).
- 1969-1979: Early development of knowledge-based systems.
- 1980-1988: Expert systems industrial boom.
- 1988-1993: Expert systems industry busts (2nd AI winter).

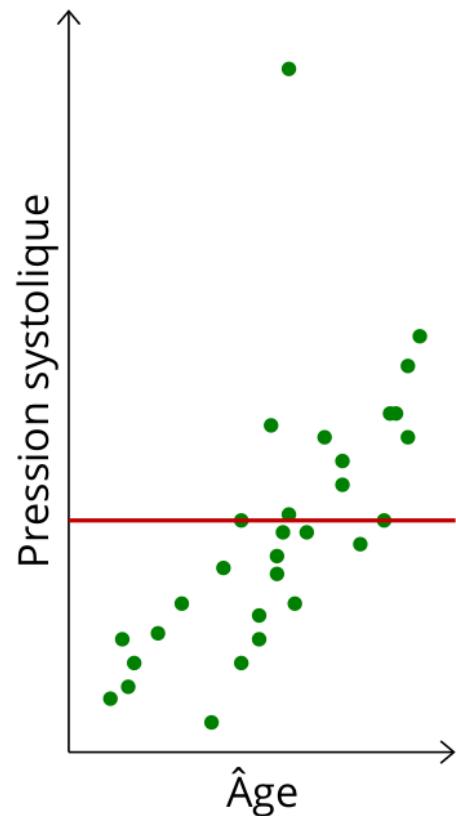
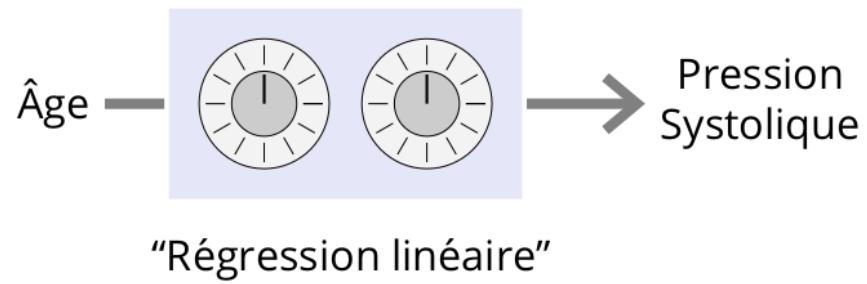
1990-Present: Statistical approaches

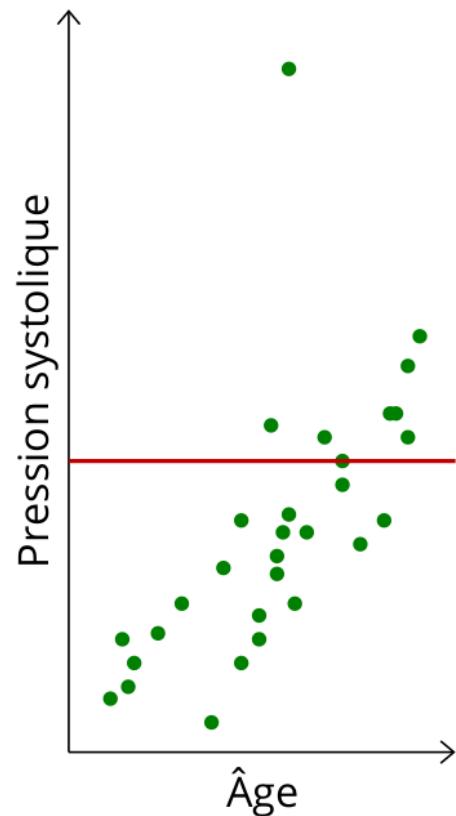
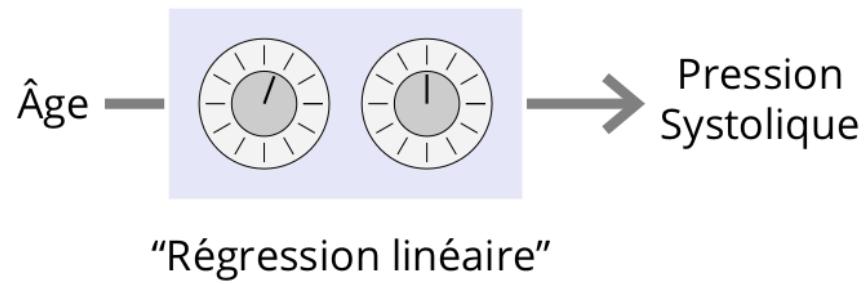
- 1985-1995: The return of neural networks.
- 1988-: Resurgence of probability, focus on uncertainty, general increase in technical depth.
- 1995-2010: New fade of neural networks.
- 2000-: Availability of very large datasets.
- 2010-: Availability of fast commodity hardware (GPUs).
- 2012-: Resurgence of neural networks with deep learning approaches.
- 2017: Attention is all you need (transformers).
- 2022: ChatGPT released to the public.

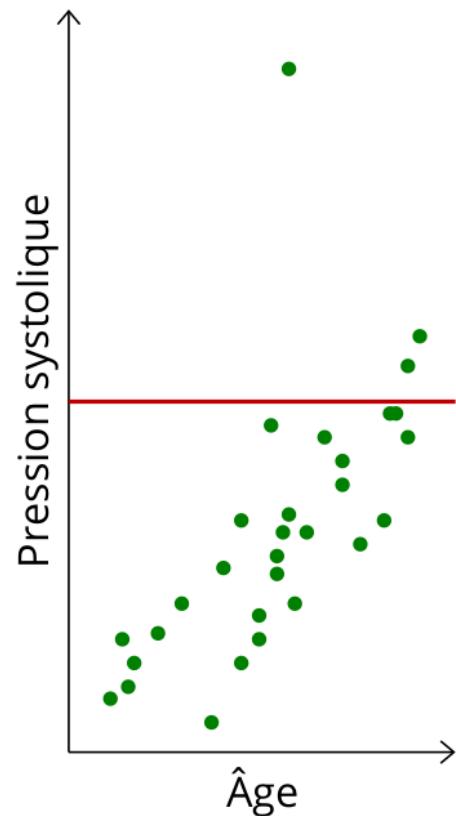
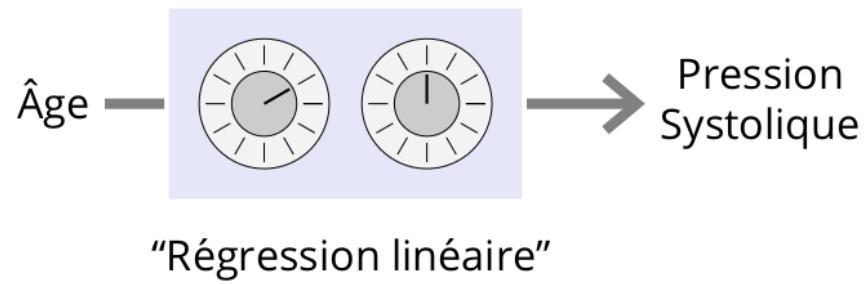
The deep learning revolution

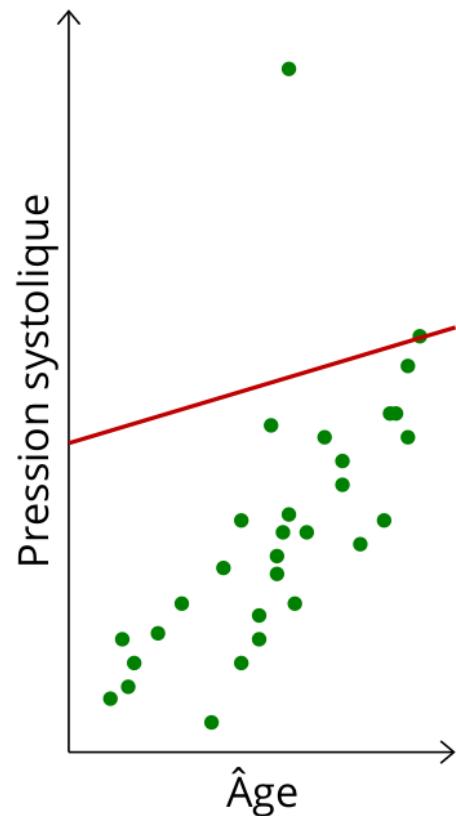
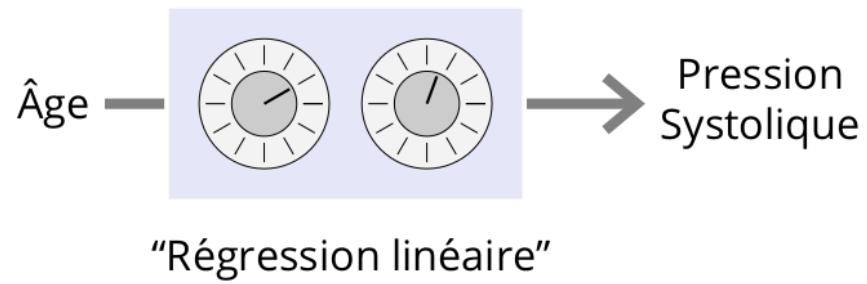


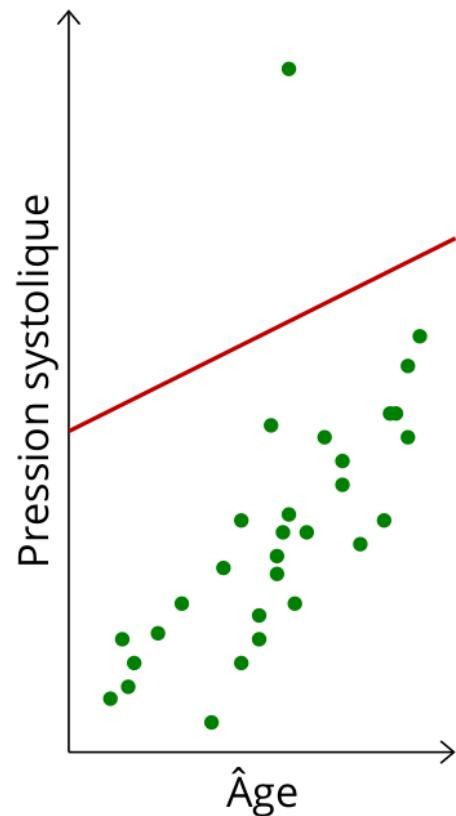
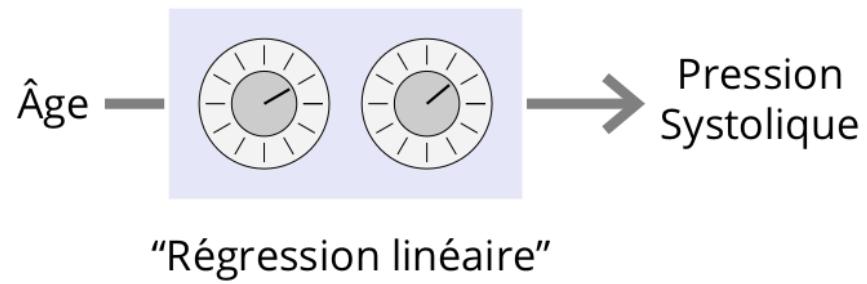


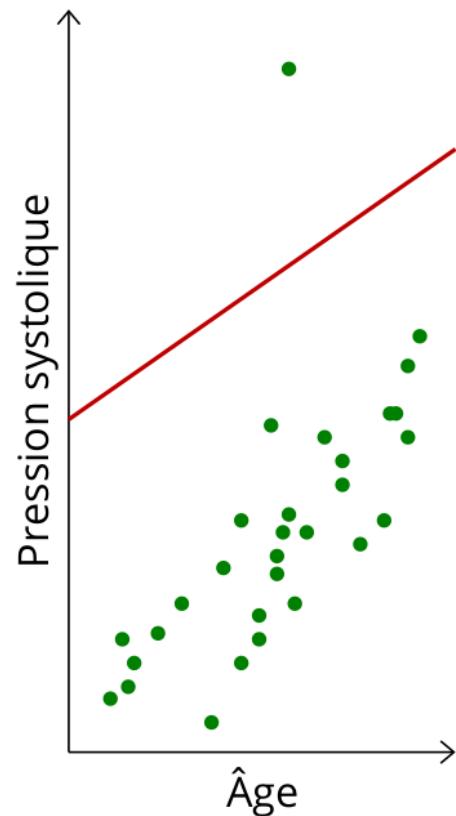
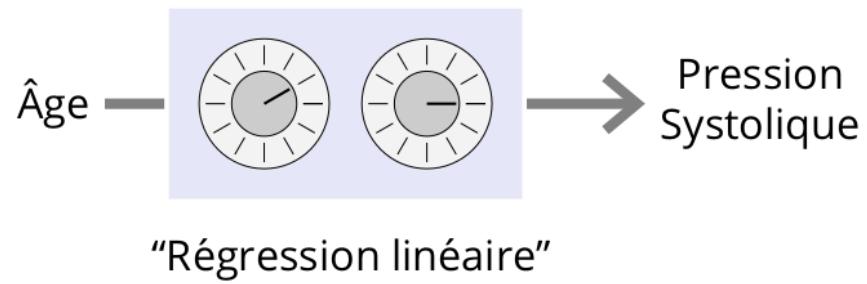


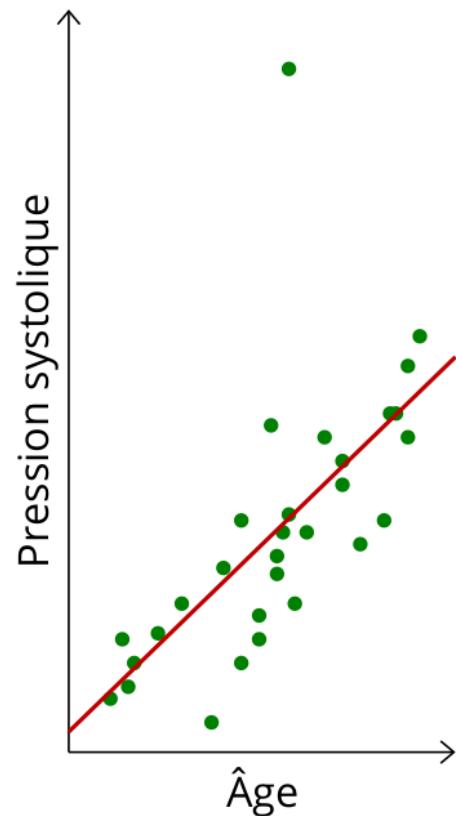
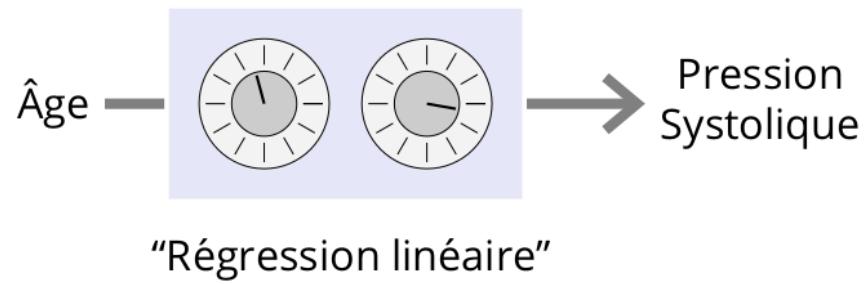






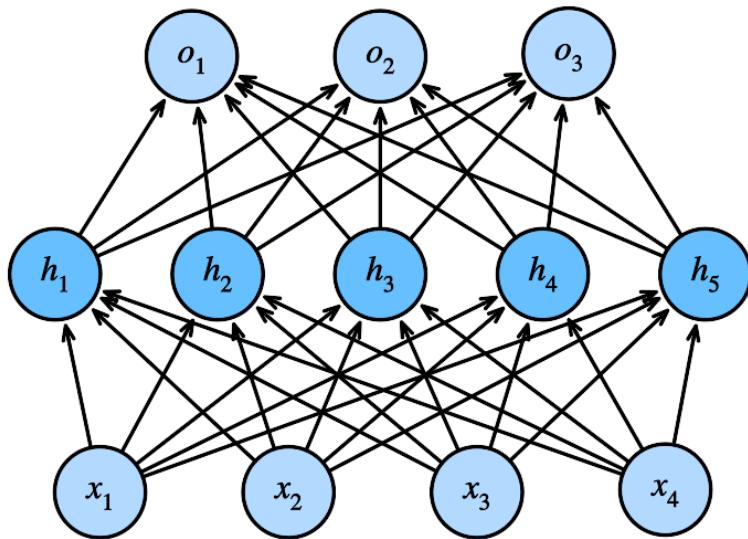




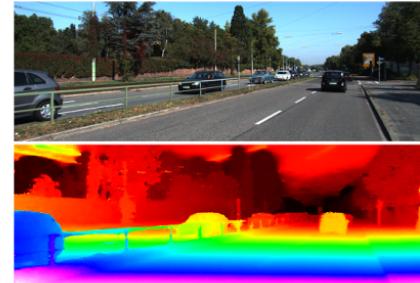
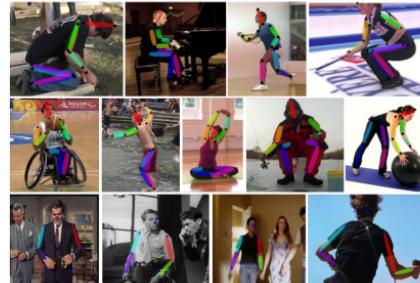
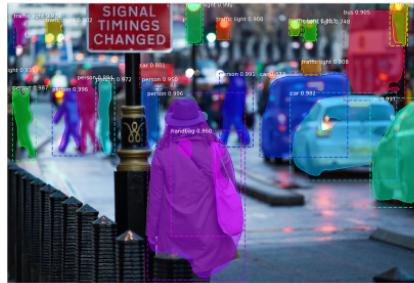


Deep learning **scales up** the statistical and machine learning approaches by

- using larger models known as neural networks,
- training on larger datasets,
- using more compute resources.



Specialized neural networks can be trained achieve super-human performance on many complex tasks that were previously thought to be out of reach for machines.

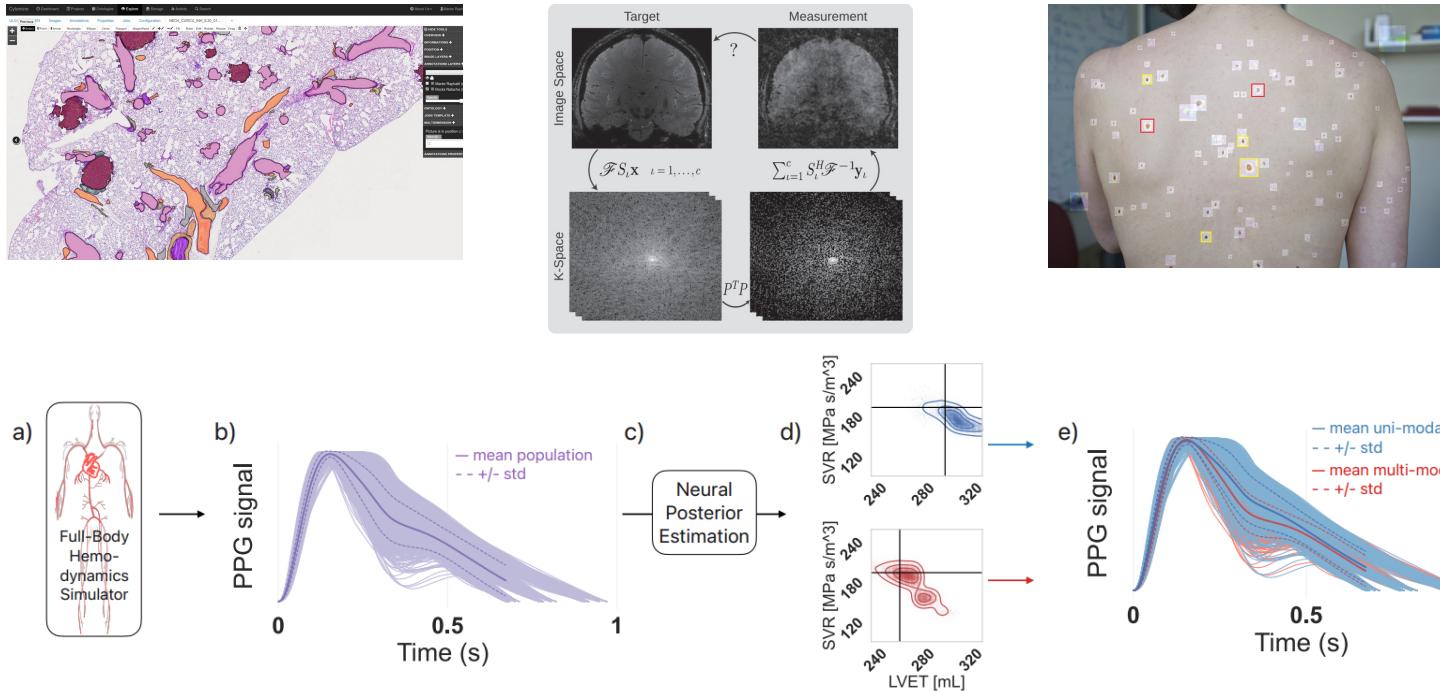


- I: Jane went to the hallway.
I: Mary walked to the bathroom.
I: Sandra went to the garden.
I: Daniel went back to the garden.
I: Sandra took the milk there.
Q: Where is the milk?
A: garden

(Top) Scene understanding, pose estimation, geometric reasoning.

(Bottom) Planning, Image captioning, Question answering.

Neural networks form **primitives** that can be transferred to many domains.



(Top) Analysis of histological slides, denoising of MRI images, nevus detection.

(Bottom) Whole-body hemodynamics reconstruction from PPG signals.

The breakthrough

Attention Is All You Need

Ashish Vaswani*
Google Brain
avaswani@google.com

Noam Shazeer*
Google Brain
noam@google.com

Niki Parmar*
Google Research
nikip@google.com

Jakob Uszkoreit*
Google Research
usz@google.com

Llion Jones*
Google Research
llion@google.com

Aidan N. Gomez* †
University of Toronto
aidan@cs.toronto.edu

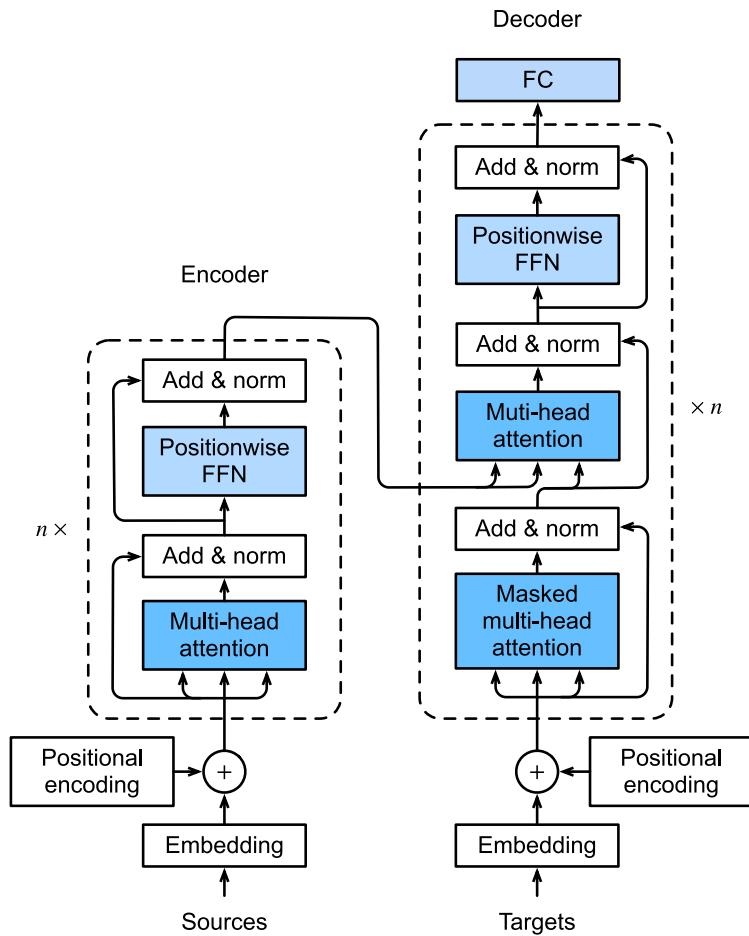
Lukasz Kaiser*
Google Brain
lukasz.kaiser@google.com

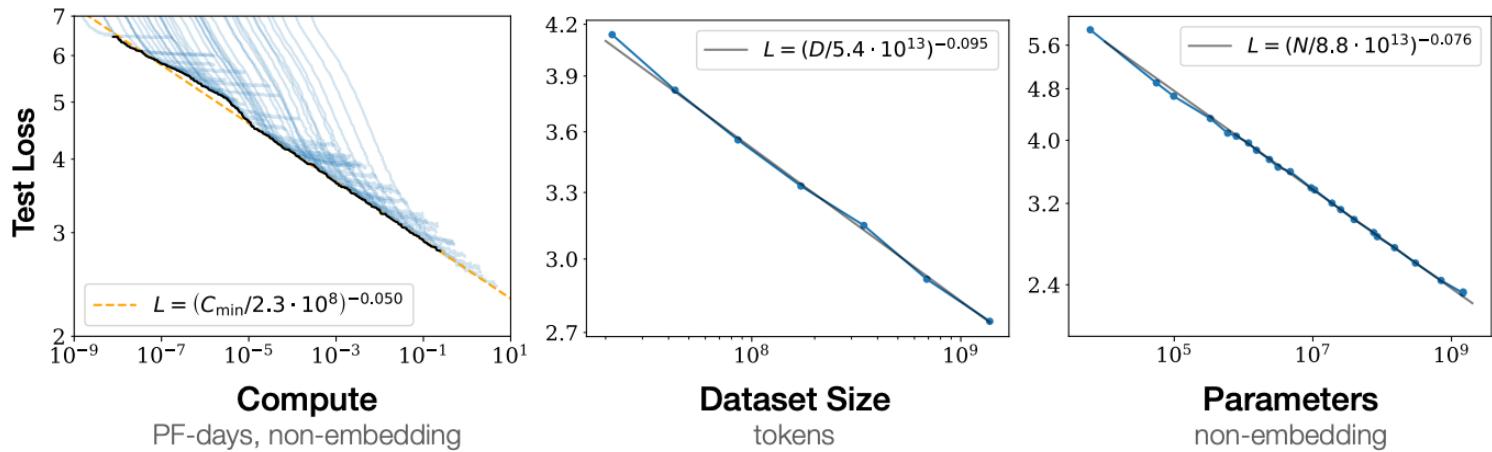
Illia Polosukhin* ‡
illia.polosukhin@gmail.com

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

Vaswani et al., 2017.





A brutal simplicity:

- The more data, the better the model.
- The more parameters, the better the model.
- The more compute, the better the model.

Scaling up further to gigantic models, datasets, and compute resources keeps pushing the boundaries of what is possible, **with no sign of slowing down**.



Claude 3.5 Sonnet as a writing partner



Later bekij...

Delen

The screenshot shows a conversational interface with two main sections. On the left, a user message is displayed:

AI: *...knowing how their families' histories intertwine and potentially influence their current relationship and work together. It adds depth to the story by suggesting multiple avenues for backstory and character development.*

User: Would you like me to explain any part of the genogram in more detail or explore how these family connections might influence the story?

AI: Fantastic. I'm curious how best to showcase the arthropod <-> human interaction. Could you ideate on a sample of the translation?

User: Certainly! Let's explore a sample of how the arthropod-human interaction and translation might work between Claw'd and Dr. Wave. I'll create a brief dialogue that demonstrates this unique communication system.

AI: Click to open document

AI: [Performs a series of rapid claw clicks, antennae vibrations, and releases a pattern of small bubbles]

JustAI (Translation Interface): Analyzing crab communication patterns... Processing...

AI: Translated output: "Hey doc,

On the right, a modal window titled "Sample Dialogue: Claw'd and Dr. Wave's Translated Interaction" is shown:

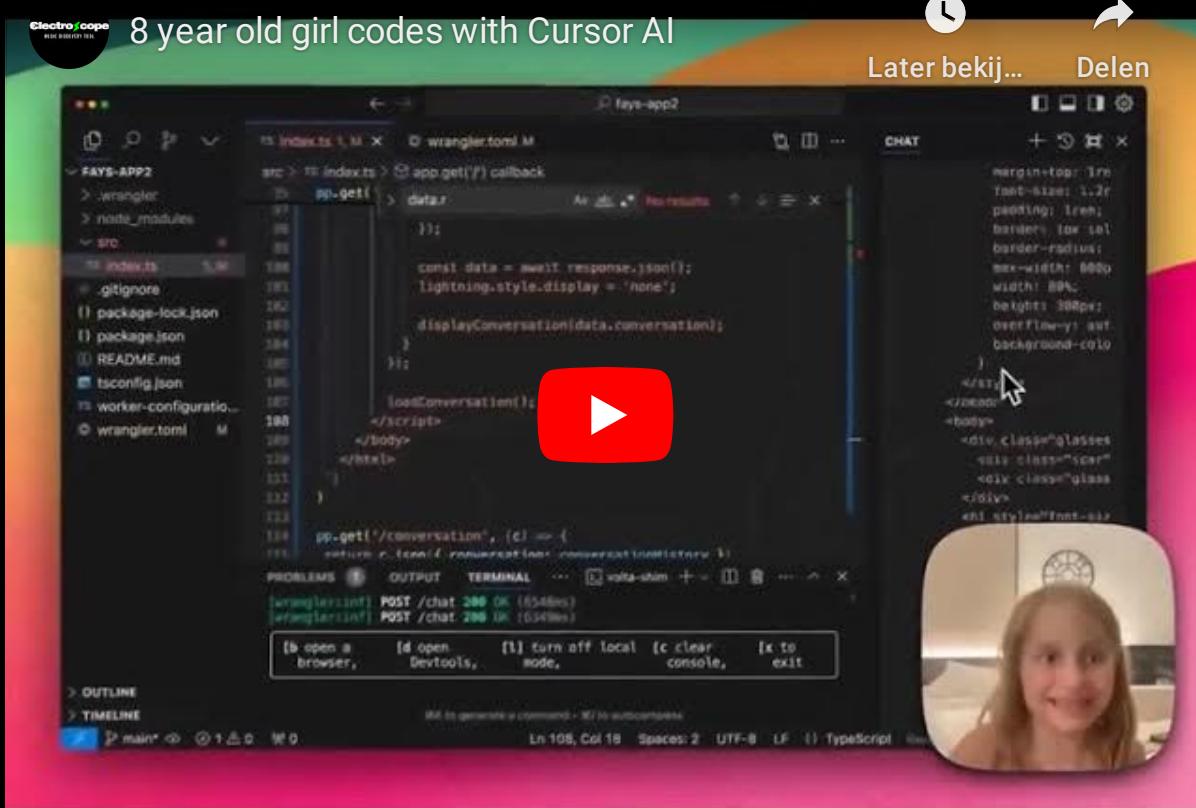
Setting: An underwater research station. Dr. Wave is analyzing data on a holographic display. Claw'd approaches, his movements captured by an array of advanced sensors.

Claw'd: [Performs a series of rapid claw clicks, antennae vibrations, and releases a pattern of small bubbles]

JustAI (Translation Interface): Analyzing crab communication patterns... Processing...

AI: Translated output: "Hey doc,

Conversational AI assistants (Anthropic, 2024)



Code assistants (Cursor, 2024)



Sense, Solve, and Go: The Magic of the Wa...



Later bekij...



Delen



Autonomous cars (Waymo, 2022)



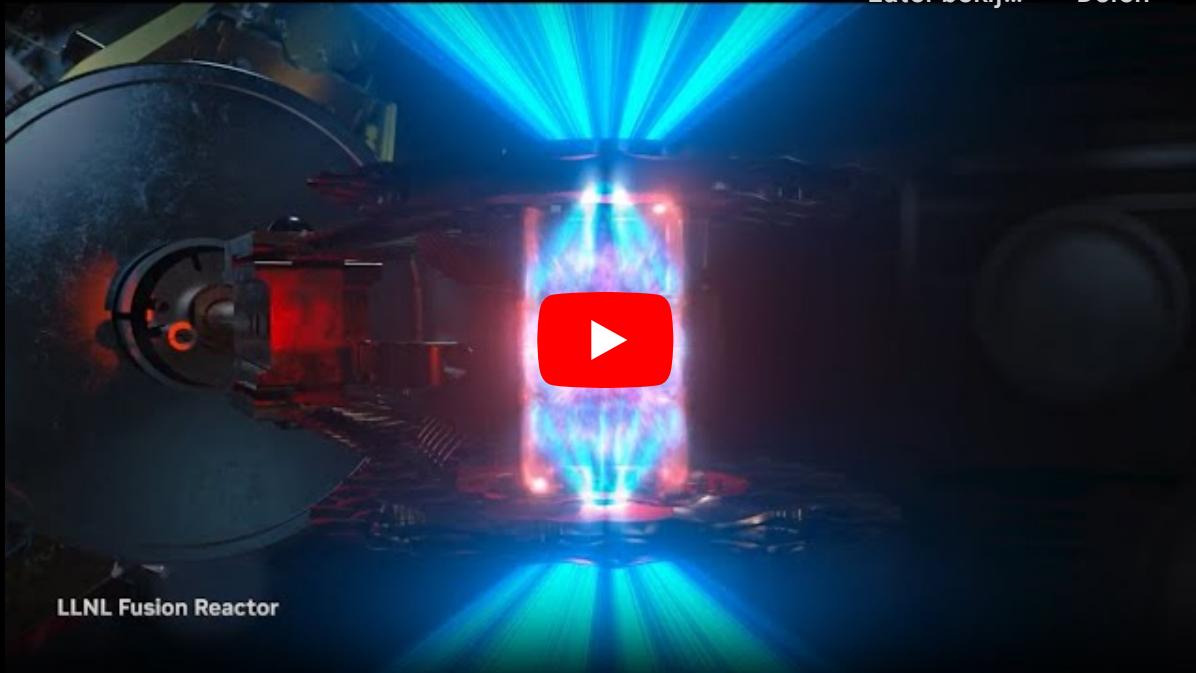
Powering the Future of Clean Energy | I AM ...



Later bekij...
Later bekijken



Delen



Powering the future of clean energy (NVIDIA, 2023)



Camels, Code & Lab Coats: How AI Is Advancing Medicine



Later bekijk...



Delen



How AI is advancing medicine (Google, 2018)

Deep learning can also **solve problems that no one could solve before**.

AlphaFold: From a sequence of amino acids to a 3D structure

nature

Explore content ▾ About the journal ▾ Publish with us ▾

nature > articles > article

Article | [Open access](#) | Published: 15 July 2021

Highly accurate protein structure prediction with AlphaFold

John Jumper , Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishabh Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michał Zieliński, ... Demis Hassabis 

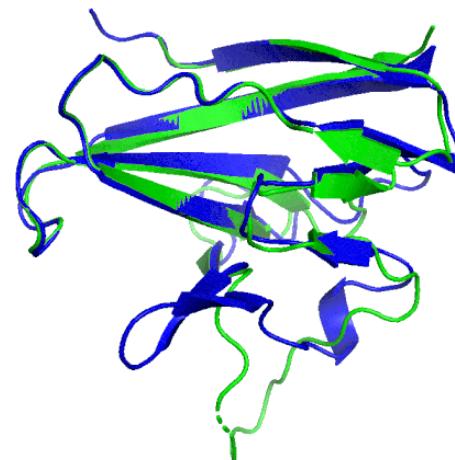
+ Show authors

[Nature](#) 596, 583–589 (2021) | [Cite this article](#)

1.42m Accesses | 12k Citations | 3493 Altmetric | [Metrics](#)

Abstract

Proteins are essential to life, and understanding their structure can facilitate a mechanistic understanding of their function. Through an enormous experimental effort^{1,2,3,4}, the structures of around 100,000 unique proteins have been determined⁵, but this represents a small fraction of the billions of known protein sequences^{6,7}. Structural coverage is bottlenecked by the months to years of painstaking effort required to determine a single protein structure. Accurate computational approaches are needed to address this gap and to enable large-scale structural bioinformatics. Predicting the three-dimensional structure that a protein will adopt based solely on its amino acid sequence—the structure prediction component of the ‘protein folding problem’⁸—has been an important open research problem for more than 50 years⁹. Despite recent progress^{10,11,12,13,14}, existing methods fall far short of atomic accuracy, especially when no homologous structure is available. Here we provide the





AlphaFold: The making of a scientific break...



Later bekij...



Delen



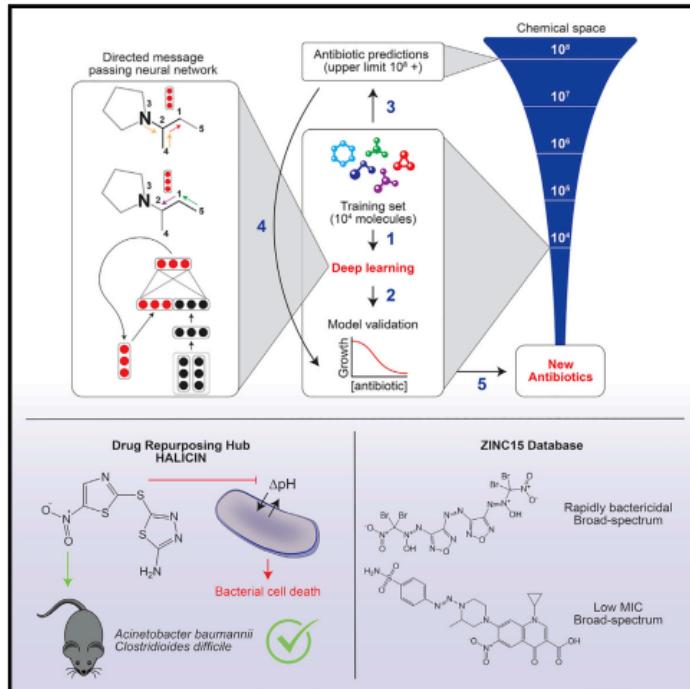
AI for Science (Deepmind, AlphaFold, 2020)

Drug discovery with graph neural networks

Cell

A Deep Learning Approach to Antibiotic Discovery

Graphical Abstract



Authors

Jonathan M. Stokes, Kevin Yang,
Kyle Swanson, ..., Tommi S. Jaakkola,
Regina Barzilay, James J. Collins

Correspondence

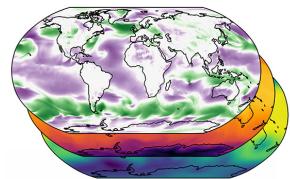
regina@csail.mit.edu (R.B.),
jimjc@mit.edu (J.J.C.)

In Brief

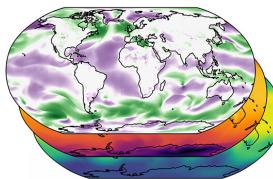
A trained deep neural network predicts antibiotic activity in molecules that are structurally different from known antibiotics, among which Halicin exhibits efficacy against broad-spectrum bacterial infections in mice.

GraphCast: fast and accurate weather forecasts

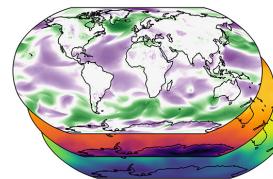
a) Input weather state



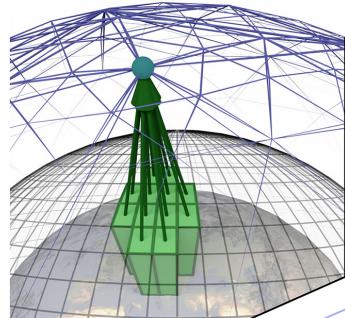
b) Predict the next state



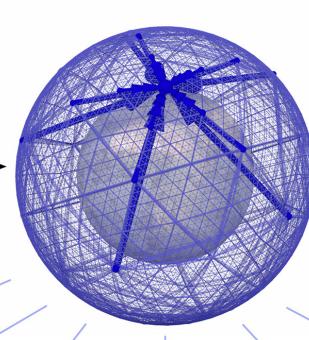
c) Roll out a forecast



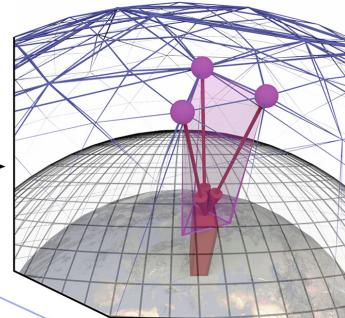
d) Encoder



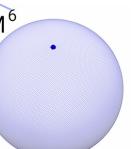
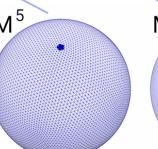
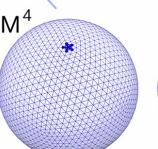
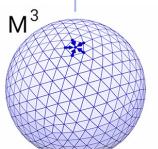
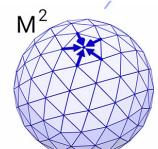
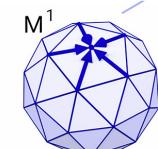
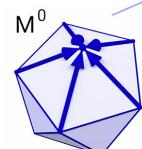
e) Processor



f) Decoder



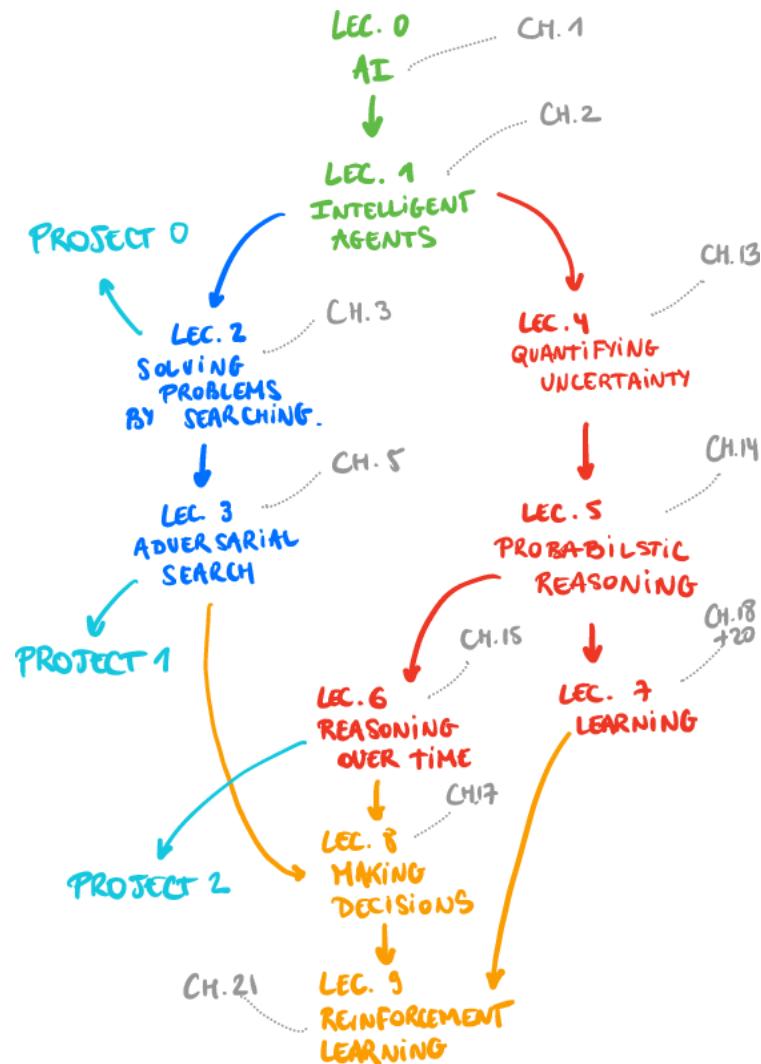
g) Simultaneous multi-mesh message-passing



INFO8006 Introduction to AI

Course outline

- Lecture 0: Artificial intelligence
- Lecture 1: Intelligent agents
- Lecture 2: Solving problems by searching
- Lecture 3: Adversarial search
- Lecture 4: Quantifying uncertainty
- Lecture 5: Probabilistic reasoning
- Lecture 6: Reasoning over time
- Lecture 7: Machine learning and neural networks
- Lecture 8: Making decisions
- Lecture 9: Reinforcement learning



My mission

By the end of this course, you will have built autonomous agents that efficiently make decisions in fully informed, partially observable and adversarial settings. Your agents will draw inferences in uncertain and unknown environments and optimize actions for arbitrary reward structures.

The models and algorithms you will learn in this course apply to a wide variety of artificial intelligence problems and will serve as the foundation for further study in any application area (from engineering and science, to business and medicine) you choose to pursue.

Goals and philosophy

General

- Understand the landscape of artificial intelligence.
- Be able to write from scratch, debug and run (some) AI algorithms.

Well-established and state-of-the-art algorithms

- Good old-fashioned AI: well-established algorithms for intelligent agents and their mathematical foundations.
- Introduction to materials new from research (\leq 5 years old).
- Understand some of the open questions and challenges in the field.

Practical

- Fun and challenging course projects.

Going further

This course is designed as an introduction to the many other courses available at ULiège and (broadly) related to AI, including:

- INFO8006: Introduction to Artificial Intelligence ← **you are there**
- DATS0001: Foundations of Data Science
- ELEN0062: Introduction to Machine Learning
- INFO8010: Deep Learning
- INFO8004: Advanced Machine Learning
- INFO9023: Machine Learning Systems Design
- INFO8003: Optimal decision making for complex problems
- INFO948: Introduction to Intelligent Robotics
- INFO9014: Knowledge representation and reasoning
- ELEN0016: Computer vision

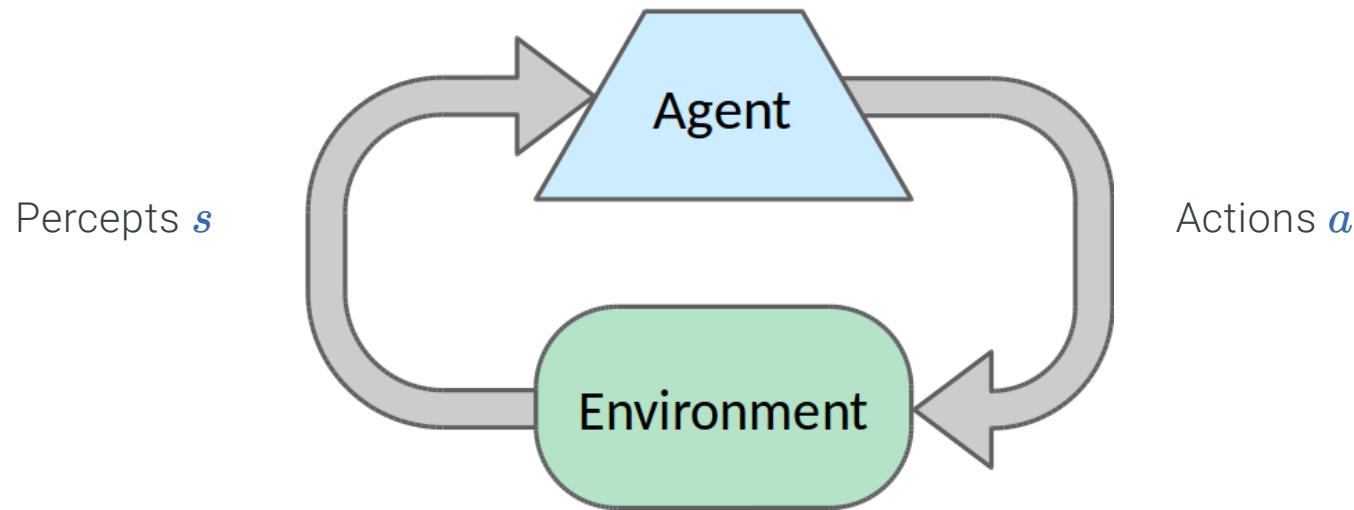
Introduction to Artificial Intelligence

Lecture 1: Intelligent agents

Prof. Gilles Louppe
g.louppe@uliege.be

Intelligent agents

Agents and environments



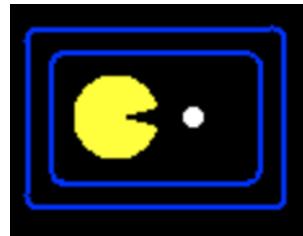
Agents

- An **agent** is an entity that **perceives** its environment through sensors and take **actions** through actuators.
- The agent behavior is described by its **policy**, a function

$$\pi : \mathcal{P}^* \rightarrow \mathcal{A}$$

that maps percept sequences to actions.

Simplified Pacman world



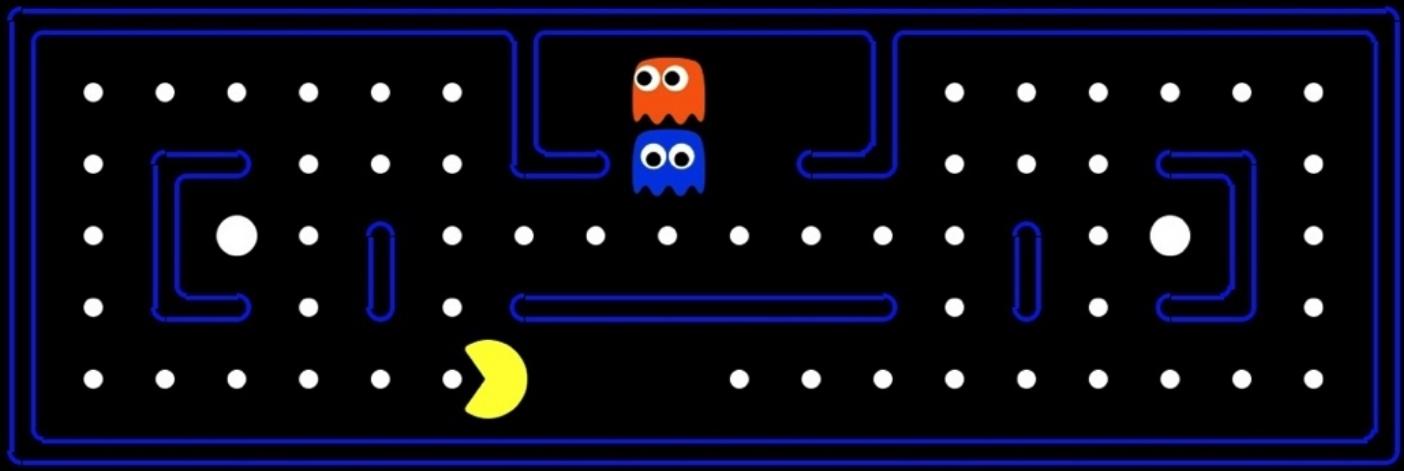
Let us consider a 2-cell world with a Pacman agent.

- Percepts: location and content, e.g. **(left cell, no food)**
- Actions: **go left, go right, eat, do nothing**

Pacman agent

The policy of a Pacman agent is a function that maps percept sequences to actions. It can be implemented as a table.

Percept sequence	Action
(left cell, no food)	go right
(left cell, food)	eat
(right cell, no food)	go left
(left cell, food)	eat
(left cell, no food), (left cell, no food)	go right
(left cell, no food), (left cell, food)	eat
(...)	(...)



SCORE: 18

What about the actual Pacman?

The optimal policy?

What is the optimal agent policy?

How to even formulate the goal of Pacman?

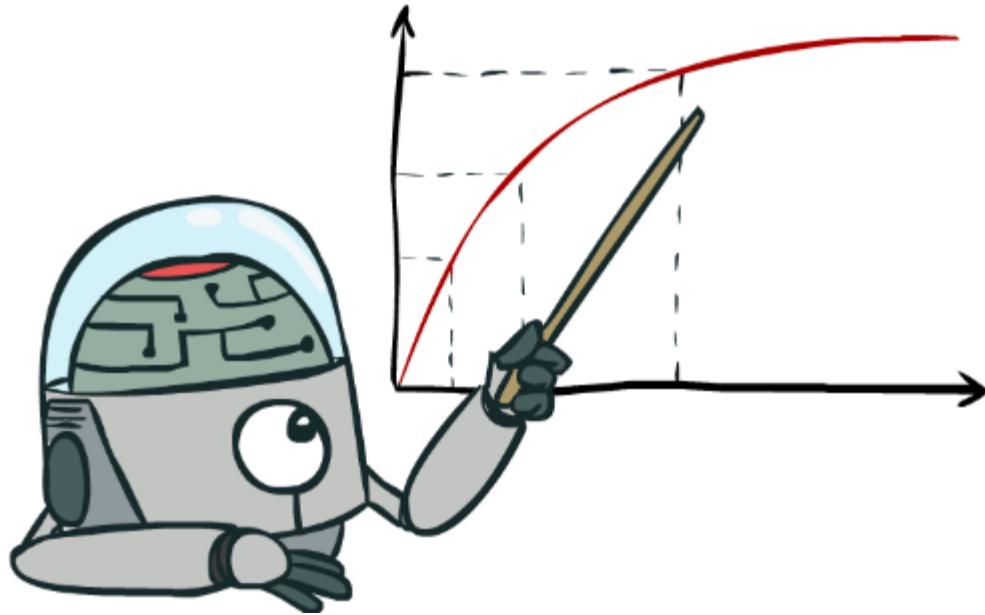
- 1 point per food dot collected up to time t ?
- 1 point per food dot collected up to time t , minus one per move?
- penalize when too many food dots are left not collected?

Can it be implemented in a **small** and **efficient** agent program?

Rational agents

- A performance measure evaluates a sequence of environment states caused by the agent's behavior.
- A rational agent is an agent that chooses whichever action that **maximizes** the **expected** value of the performance measure, given the percept sequence to date.

Rationality only concerns **what** decisions are made (not the thought process behind them, human-like or not).



In this course, Artificial intelligence = **Maximizing expected performance**

- Rationality \neq omniscience
 - percepts may not supply all relevant information.
- Rationality \neq clairvoyance
 - action outcomes may not be as expected.
- Hence, rational \neq successful.
- However, rationality leads to exploration, learning and autonomy.

Performance, environment, actuators, sensors

The characteristics of the performance measure, environment, action space and percepts dictate approaches for selecting rational actions. They are summarized as the **task environment**.

Example 1: a chess-playing agent

- performance measure: win, draw, lose, ...
- environment: chess board, opponent, ...
- actuators: move pieces, ...
- sensors: board state, opponent moves, ...

Example 2: a self-driving car

- performance measure: safety, destination, legality, comfort, ...
- environment: streets, highways, traffic, pedestrians, weather, ...
- actuators: steering, accelerator, brake, horn, speaker, display, ...
- sensors: video, accelerometers, gauges, engine sensors, GPS, ...

Example 3: a medical diagnosis system

- performance measure: patient health, cost, time, ...
- environment: patient, hospital, medical records, ...
- actuators: diagnosis, treatment, referral, ...
- sensors: medical records, lab results, ...

Environment types

Fully observable vs. partially observable

Whether the agent sensors give access to the complete state of the environment, at each point in time.

Deterministic vs. stochastic

Whether the next state of the environment is completely determined by the current state and the action executed by the agent.

Episodic vs. sequential

Whether the agent's experience is divided into atomic independent episodes.

Static vs. dynamic

Whether the environment can change, or the performance measure can change with time.

Discrete vs. continuous

Whether the state of the environment, the time, the percepts or the actions are continuous.

Single agent vs. multi-agent

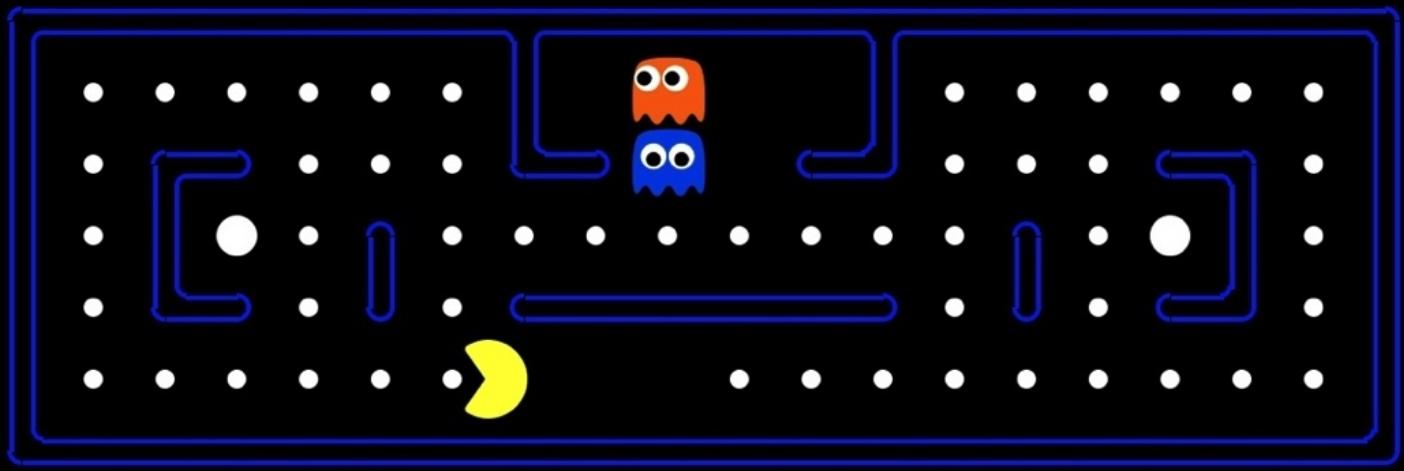
Whether the environment include several agents that may interact which each other.

Known vs unknown

Reflects the agent's state of knowledge of the "law of physics" of the environment.

Are the following task environments fully observable? deterministic? episodic? static? discrete? single agents? Known?

- Crossword puzzle
- Chess, with a clock
- Poker
- Backgammon
- Taxi driving
- Medical diagnosis
- Image analysis
- Part-picking robot
- Refinery controller
- The real world



SCORE: 18

What about Pacman?

Agent programs

Our goal is to design an **agent program** that implements the agent policy.

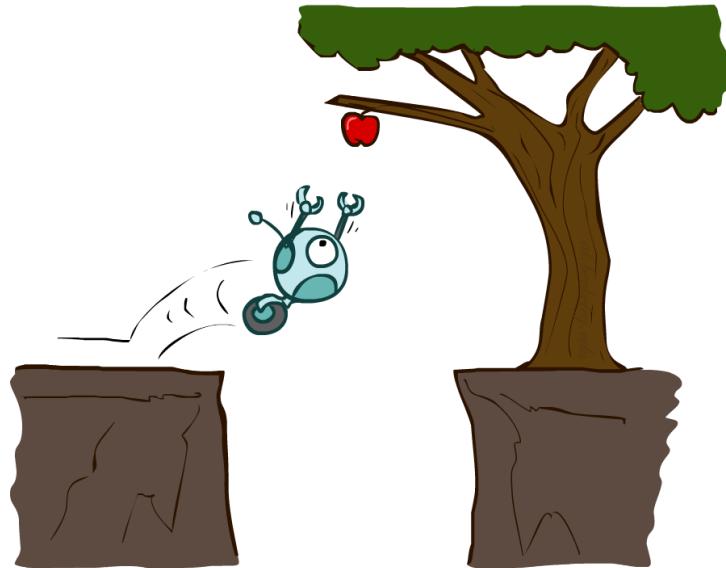
Agent programs can be designed and implemented in many ways:

- with tables
- with rules
- with search algorithms
- with learning algorithms

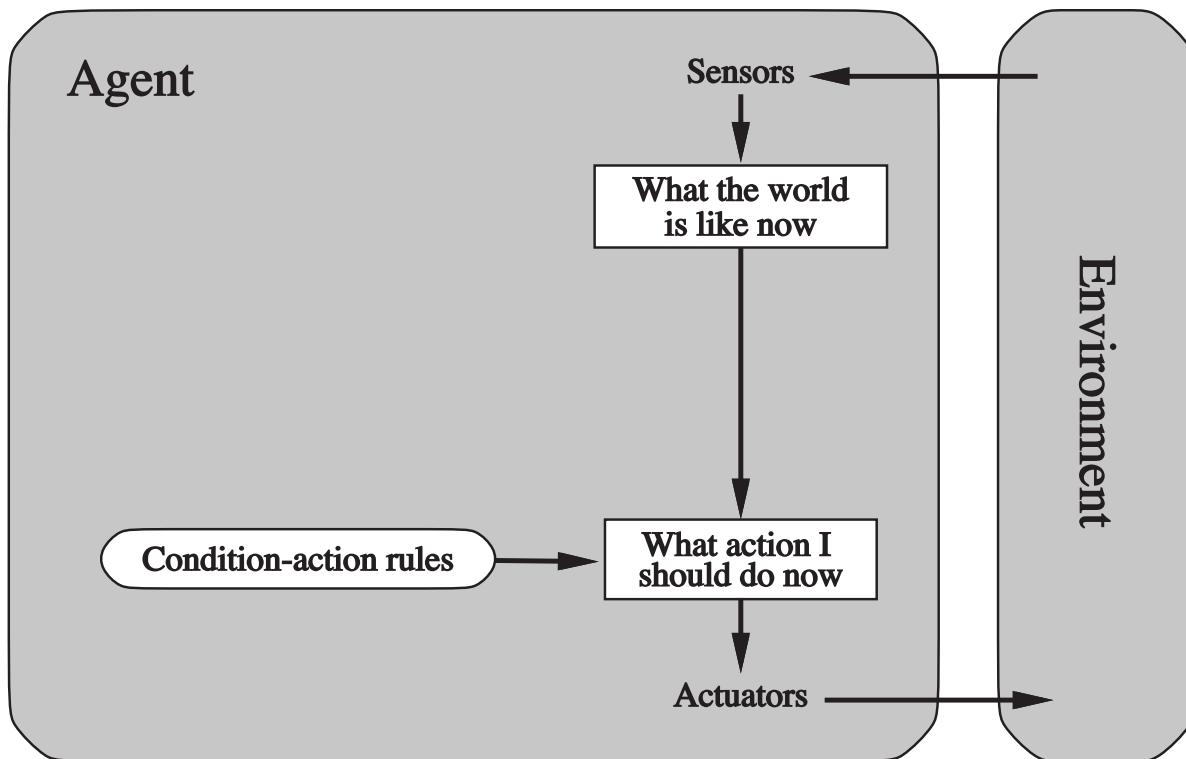
Reflex agents

Reflex agents ...

- choose an action based on current percept (and maybe memory);
- may have memory or model of the world's current state;
- do not consider the future consequences of their actions.

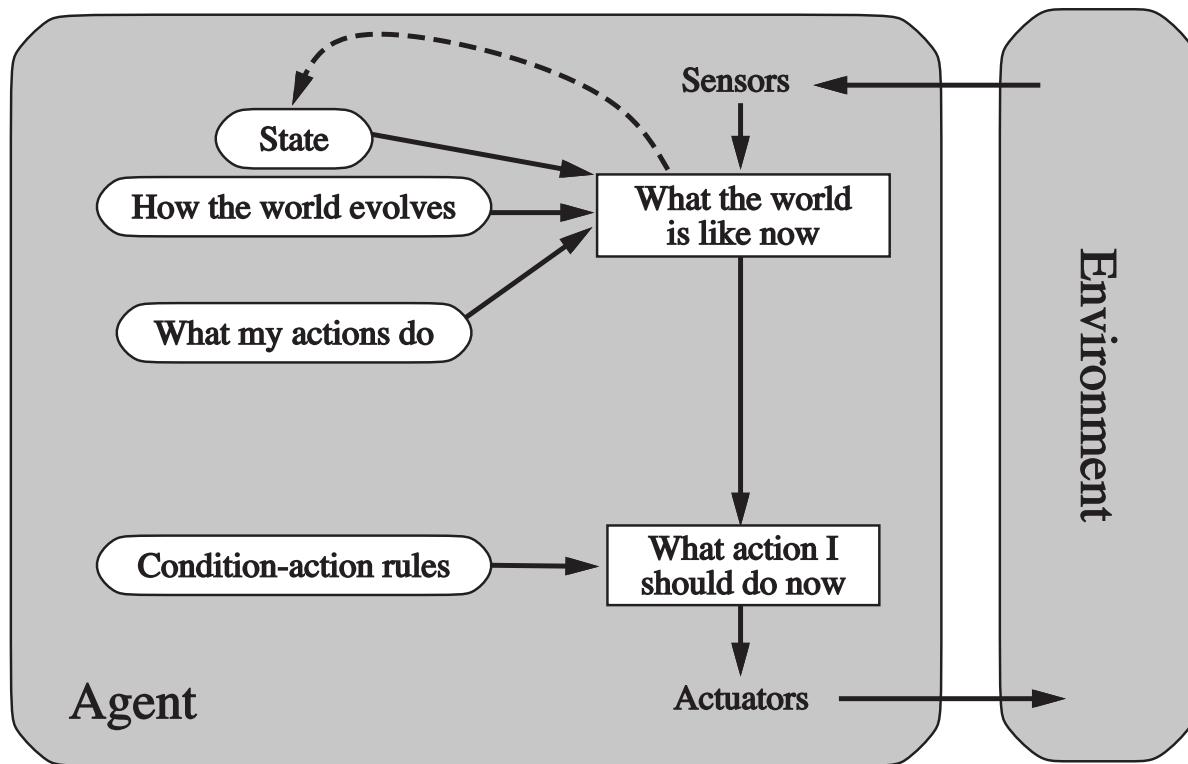


Simple reflex agents



- Simple reflex agents select actions on the basis of the current percept, ignoring the rest of the percept history.
- They implement condition-action rules that match the current percept to an action. Rules provide a way to compress the function table.
- They can only work in a Markovian environment, that is if the correct decision can be made on the basis of only the current percept. In other words, if the environment is fully observable.

Model-based reflex agents

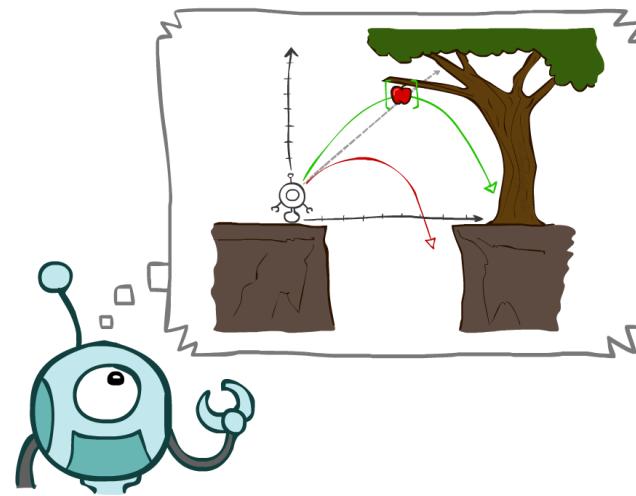


- Model-based agents handle partial observability of the environment by keeping track of the part of the world they cannot see now.
- The internal state of model-based agents is updated on the basis of a model which determines:
 - how the environment evolves independently of the agent;
 - how the agent actions affect the world.

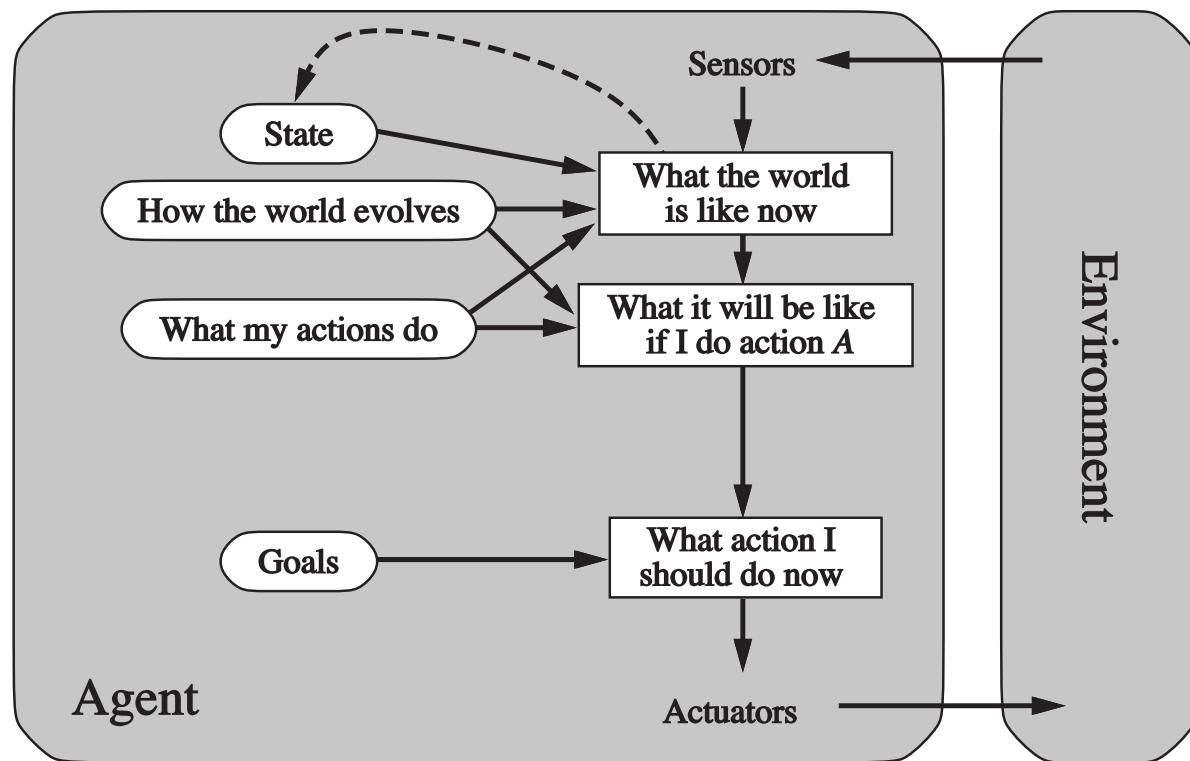
Planning agents

Planning agents ...

- ask "what if?";
- make decisions based on (hypothesized) consequences of actions;
- must have a model of how the world evolves in response to actions;
- must formulate a goal.

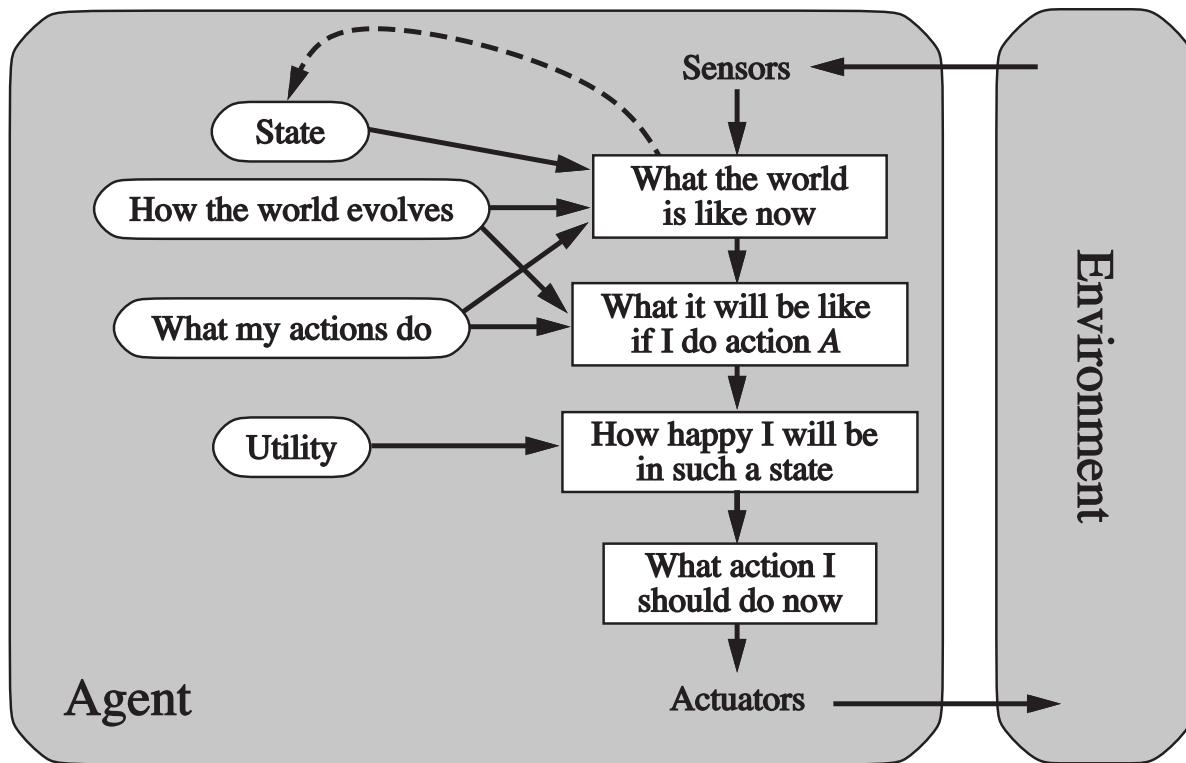


Goal-based agents



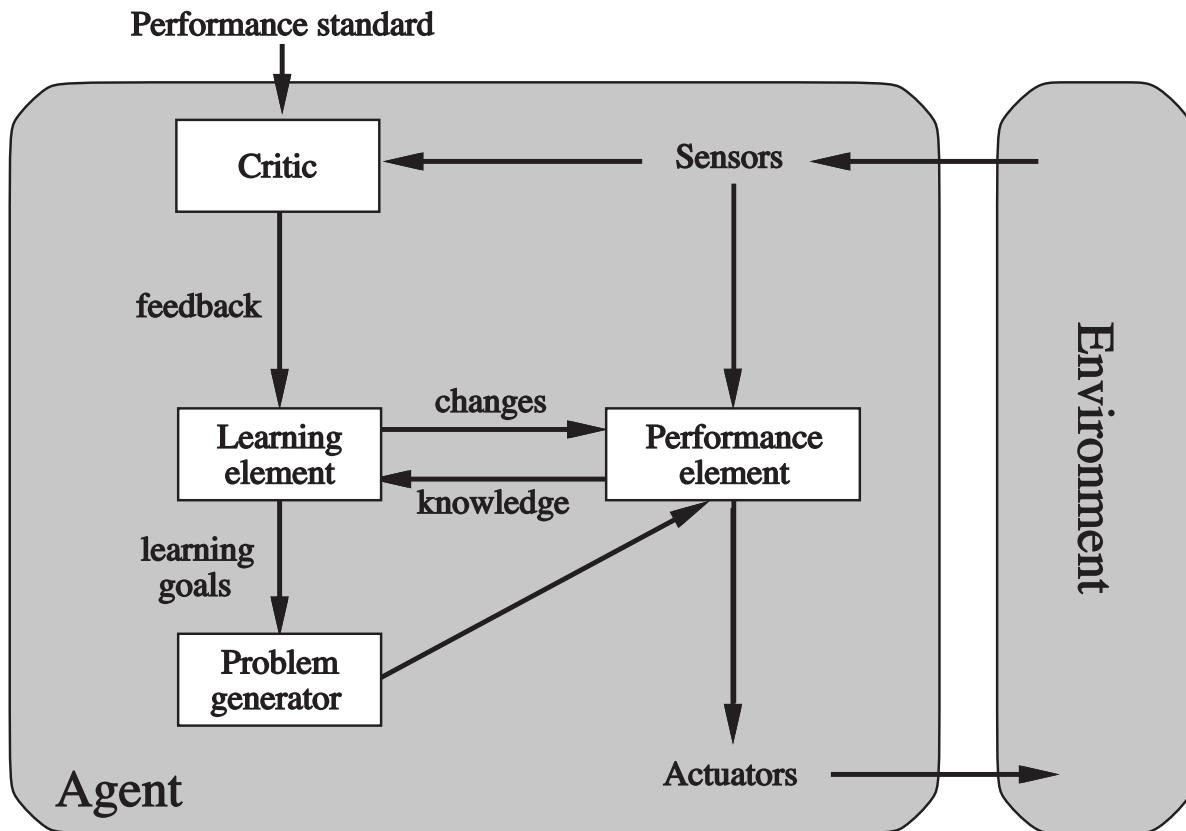
- Decision process:
 1. generate possible sequences of actions
 2. predict the resulting states
 3. assess **goals** in each.
- A **goal-based agent** chooses an action that will achieve the goal.
 - More general than rules. Goals are rarely explicit in condition-action rules.
 - Finding action sequences that achieve goals is difficult. **Search** and **planning** are two strategies.

Utility-based agents



- Goals are often not enough to generate high-quality behavior. Goals only provide binary assessment of performance.
- A utility function scores any given sequence of environment states.
 - The utility function is an internalization of the performance measure.
- A rational utility-based agent chooses an action that maximizes the expected utility of its outcomes.

Learning agents



- Learning agents are capable of self-improvement. They can become more competent than their initial knowledge alone might allow.
- They can make changes to any of the knowledge components by:
 - learning how the world evolves;
 - learning what are the consequences of actions;
 - learning the utility of actions through rewards.

A learning autonomous car

- Performance element:
 - The current system for selecting actions and driving.
- The **critic** observes the world and passes information to the **learning element**.
 - E.g., the car makes a quick left turn across three lanes of traffic. The critic observes shocking language from the other drivers and informs bad action.
 - The learning element tries to modify the performance element to avoid reproducing this situation in the future.
- The **problem generator** identifies certain areas of behavior in need of improvement and suggest experiments.
 - E.g., trying out the brakes on different surfaces in different weather conditions.

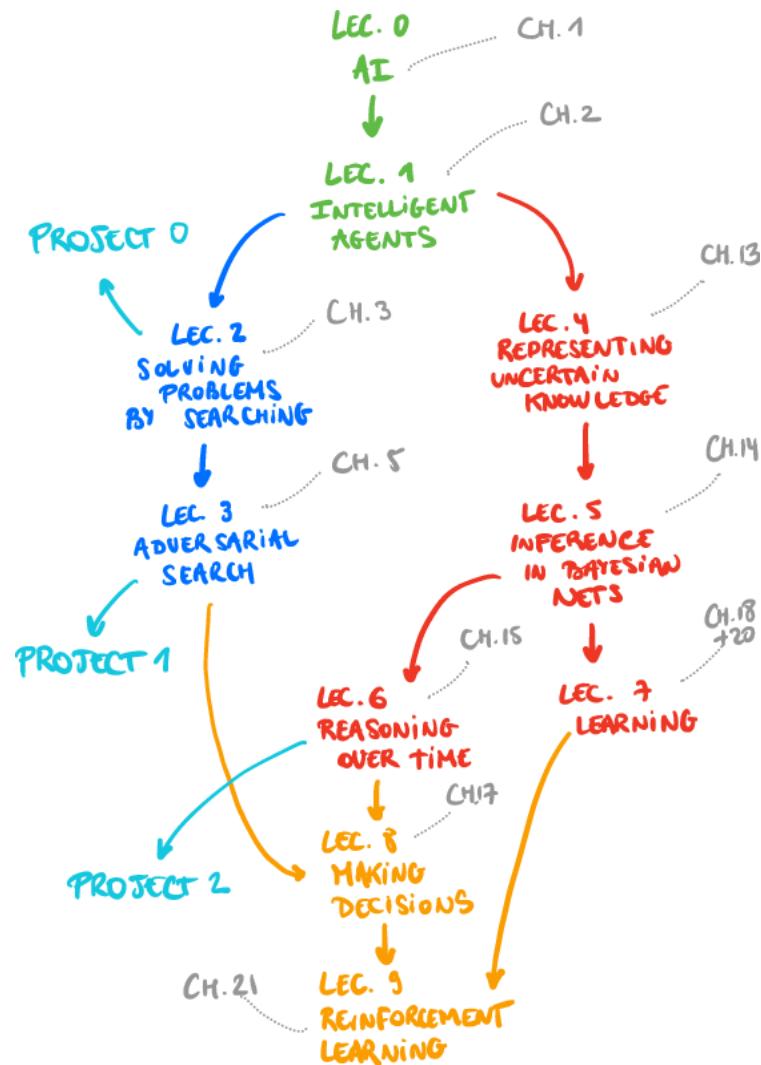
Summary

- An **agent** is an entity that perceives and acts in an environment.
- The **performance measure** evaluates the agent's behavior. **Rational agents** act so as to maximize the expected value of the performance measure.
- **Task environments** includes performance measure, environment, actuators and sensors. They can vary along several significant dimensions.
- The **agent program** effectively implements the agent function. Their designs are dictated by the task environment.
- **Simple reflex agents** respond directly to percepts, whereas **model-based reflex agents** maintain internal state to track the world. **Goal-based agents** act to achieve goals while **utility-based agents** try to maximize their expected performance.
- All agents can improve their performance through **learning**.

Introduction to Artificial Intelligence

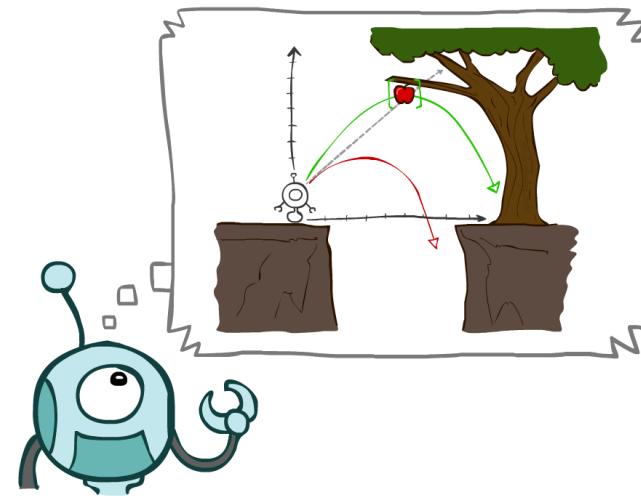
Lecture 2: Solving problems by searching

Prof. Gilles Louppe
g.louppe@uliege.be



Today

- Planning agents
- Search problems
- Uninformed search methods
 - Depth-first search
 - Breadth-first search
 - Uniform-cost search
- Informed search methods
 - A*
 - Heuristics

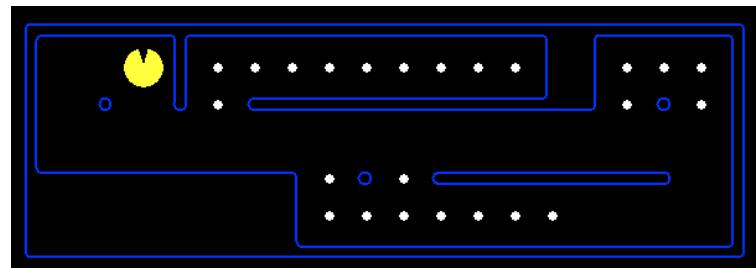
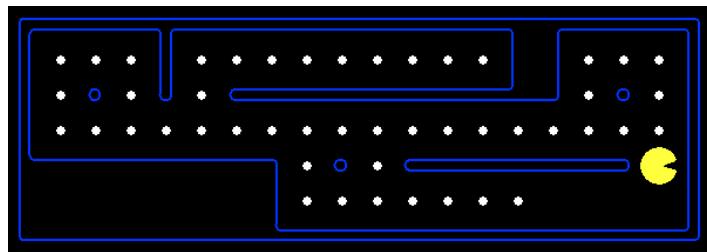


Planning agents

Reflex agents

Reflex agents

- select actions on the basis of the current percept;
- may have a model of the world current state;
- do not consider the future consequences of their actions;
- consider only **how the world is now**.



For example, a simple reflex agent based on condition-action rules could move to a dot if there is one in its neighborhood. No planning is involved to take this decision.

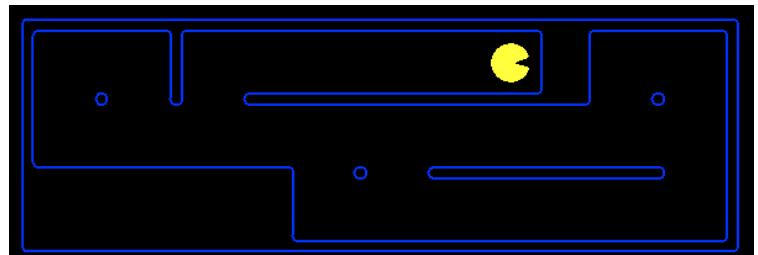
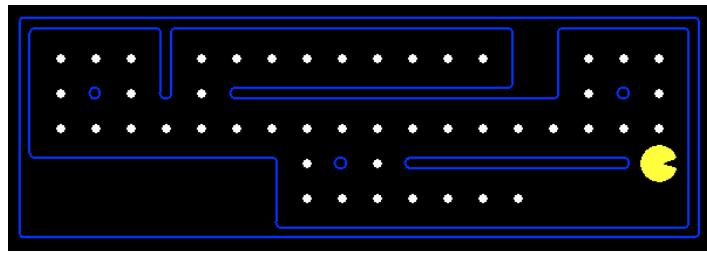
Problem-solving agents

Assumptions:

- Single-agent, observable, deterministic and known environment.

Problem-solving agents

- take decisions based on (hypothesized) consequences of actions, by considering **how the world could be**;
- must have a model of how the world evolves in response to actions;
- formulate a goal, explicitly.



A planning agent looks for sequences of actions to eat all the dots.

```

function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
    state, some description of the current world state
    goal, a goal, initially null
    problem, a problem formulation

  state  $\leftarrow$  UPDATE-STATE(state, percept)
  if seq is empty then
    goal  $\leftarrow$  FORMULATE-GOAL(state)
    problem  $\leftarrow$  FORMULATE-PROBLEM(state, goal)
    seq  $\leftarrow$  SEARCH(problem)
    if seq = failure then return a null action
    action  $\leftarrow$  FIRST(seq)
    seq  $\leftarrow$  REST(seq)
  return action

```

Figure 3.1 A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

Offline vs. Online solving

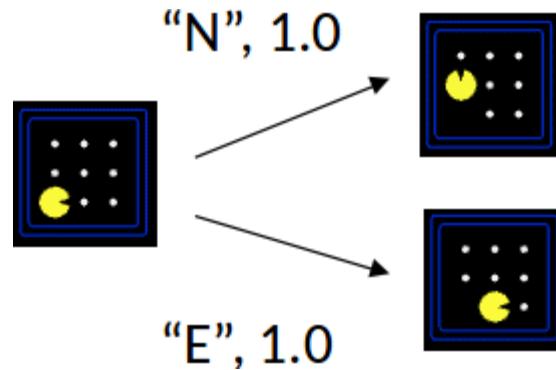
- Problem-solving agents are **offline**. The solution is executed "eyes closed", ignoring the percepts.
- **Online** problem solving involves acting without complete knowledge. In this case, the sequence of actions might be recomputed at each step.

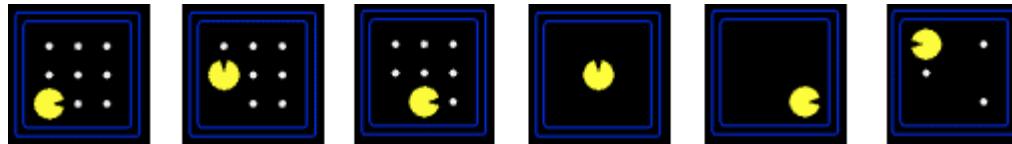
Search problems

Search problems

A **search problem** consists of the following components:

- A representation of the **states** of the agent and its environment.
- The **initial state** of the agent.
- A description of the **actions** available to the agent given a state s , denoted **actions(s)**.
- A **transition model** that returns the state $s' = \text{result}(s, a)$ that results from doing action a in state s .
 - We say that s' is a **successor** of s if there is an acceptable action from s to s' .





- Together, the initial state, the actions and the transition model define the **state space** of the problem, i.e. the set of all states reachable from the initial state by any sequence of action.
 - The state space forms a directed graph:
 - nodes = states
 - links = actions
 - A path is a sequence of states connected by actions.
- A **goal test** which determines whether the solution of the problem is achieved in state s .
- A **path cost** that assigns a numeric value to each path.
 - In this course, we will also assume that the path cost corresponds to a sum of positive **step costs** $c(s, a, s')$ associated to the action a in s leading to s' .

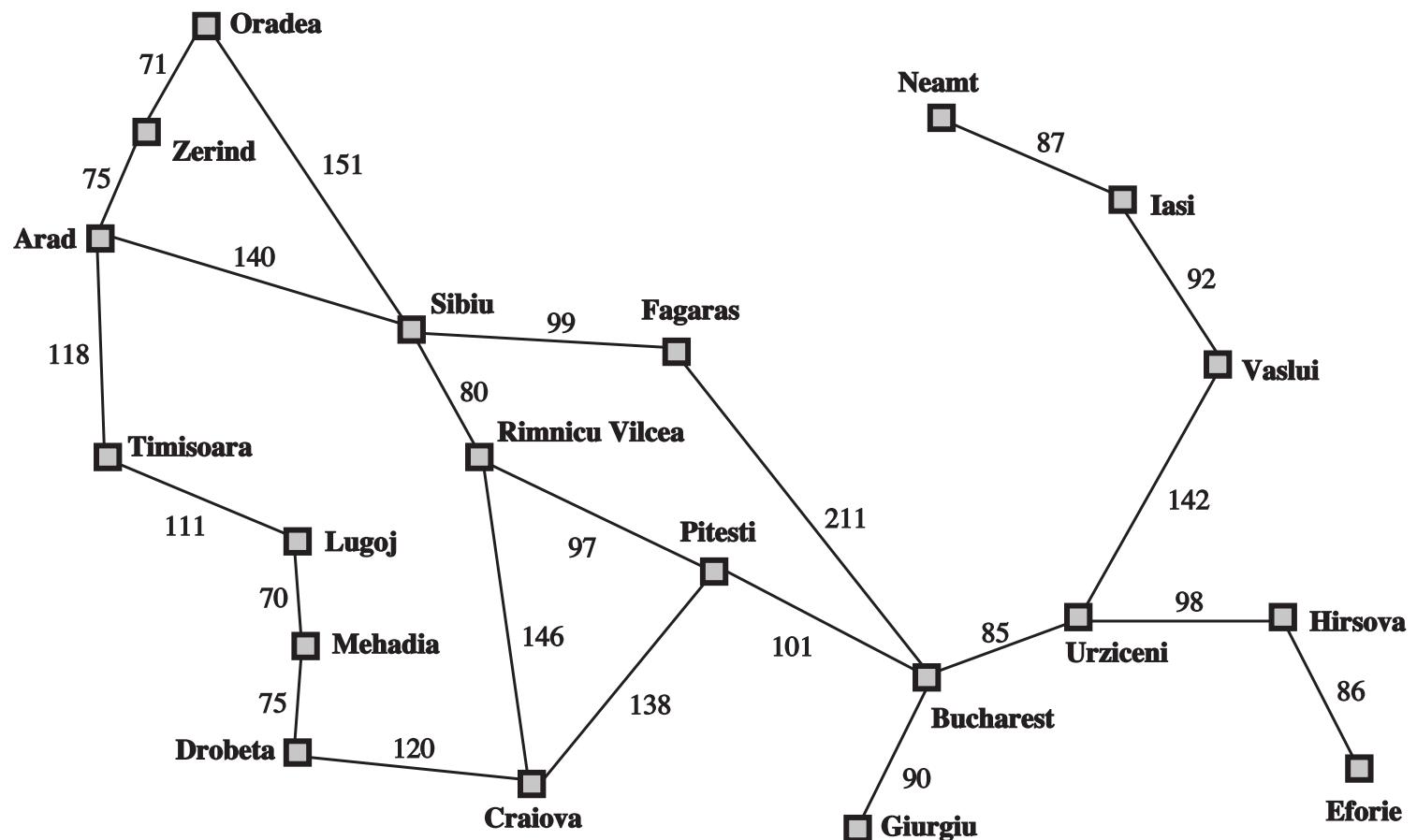
A **solution** to a problem is an action sequence that leads from the initial state to a goal state.

- A solution quality is measured by the path cost function.
- An **optimal solution** has the lowest path cost among all solutions.

Exercise

What if the environment is partially observable? non-deterministic?

Example: Traveling in Romania



How to go from Arad to Bucharest?

- Initial state = the city we start in.
 - $s_0 = \text{in(Arad)}$
- Actions = Going from the current city to the cities that are directly connected to it.
 - $\text{actions}(s_0) = \{\text{go(Sibiu)}, \text{go(Timisoara)}, \text{go(Zerind)}\}$
- Transition model = The city we arrive in after driving to it.
 - $\text{result}(\text{in}(Arad), \text{go}(Zerind)) = \text{in}(Zerind)$
- Goal test: whether we are in Bucharest.
 - $s \in \{\text{in(Bucharest)}\}$
- Step cost: distances between cities.

Selecting a state space

The real world is absurdly **complex**.

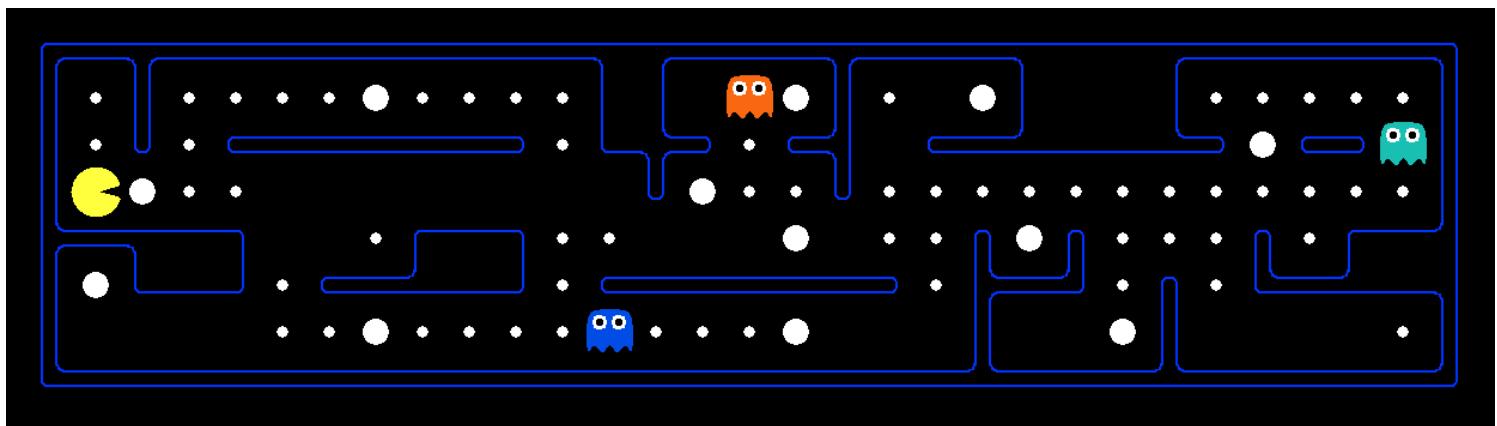
- The **world state** includes every last detail of the environment.
- A **search state** keeps only the details needed for planning.



Search problems are **models**.

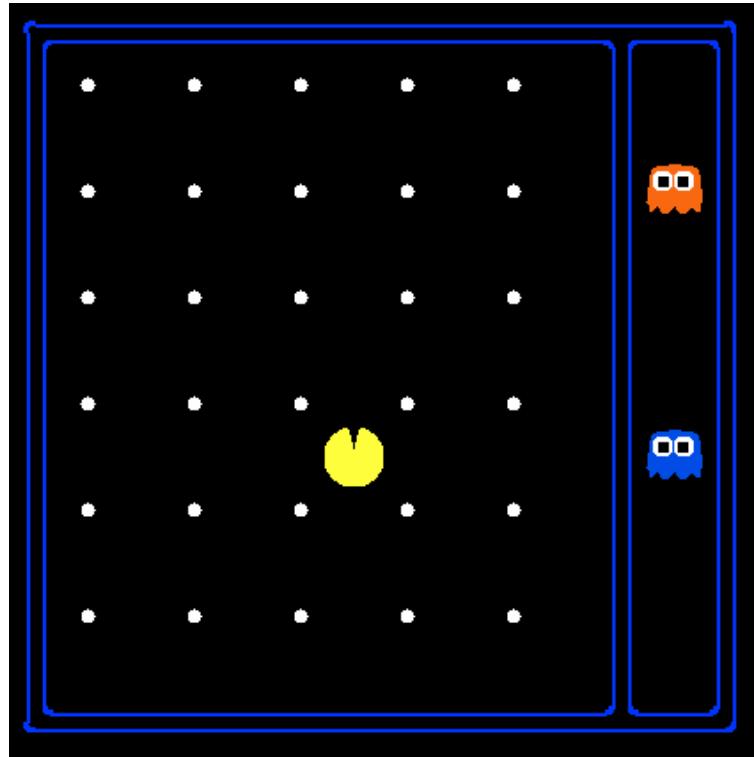
Example: eat-all-dots

- States: $\{(x, y), \text{dot booleans}\}$
- Actions: NSEW
- Transition: update location and possibly a dot boolean
- Goal test: dots all false



State space size

- World state:
 - Agent positions: 120
 - Found count: 30
 - Ghost positions: 12
 - Agent facing: NSEW
- How many?
 - World states?
 - $120 \times 2^{30} \times 12^2 \times 4$
 - States for eat-all-dots?
 - 120×2^{30}

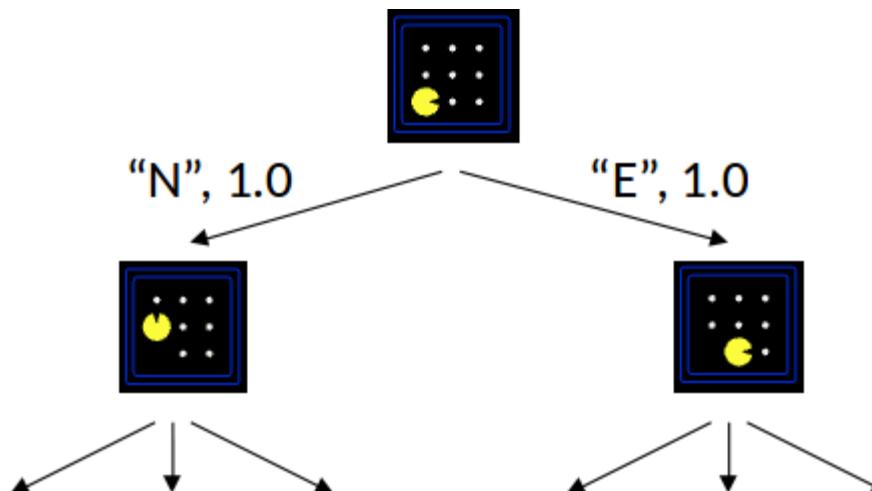


Search trees

The set of acceptable sequences starting at the initial state form a **search tree**.

- Nodes correspond to states in the state space, where the initial state is the root node.
- Branches correspond to applicable actions, with child nodes corresponding to successors.

For most problems, we can never actually build the whole tree. Yet we want to find some optimal branch!



Tree search algorithms

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
  fringe  $\leftarrow$  INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node  $\leftarrow$  REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE(node)) then return node
    fringe  $\leftarrow$  INSERTALL(EXPAND(node, problem), fringe)
```

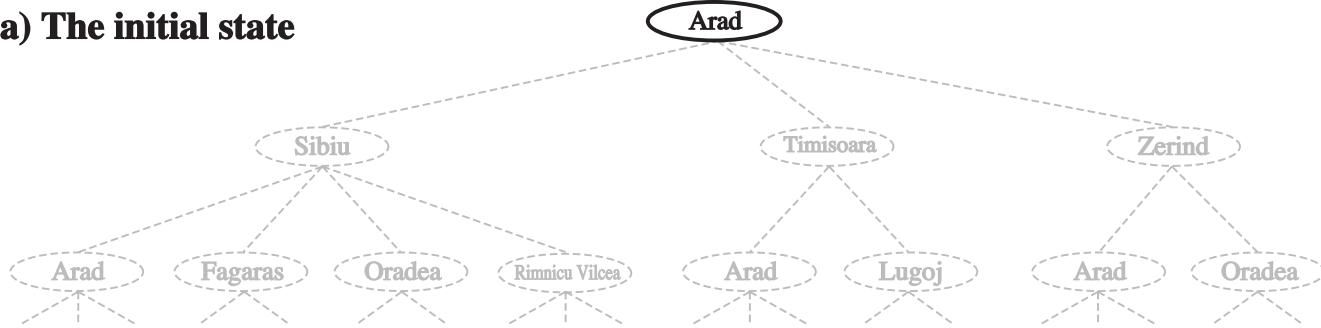
Important ideas

- Fringe (or frontier) of partial plans under consideration
- Expansion
- Exploration

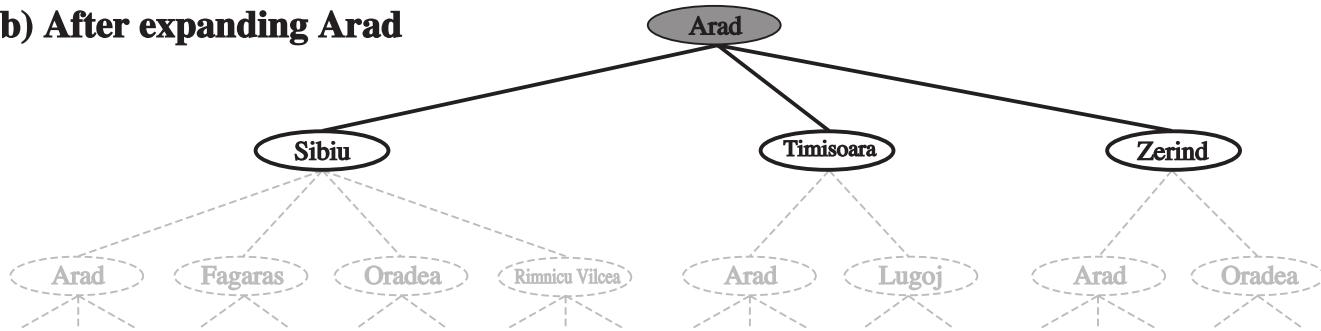
Exercise

Which fringe nodes to explore? How to expand as few nodes as possible, while achieving the goal?

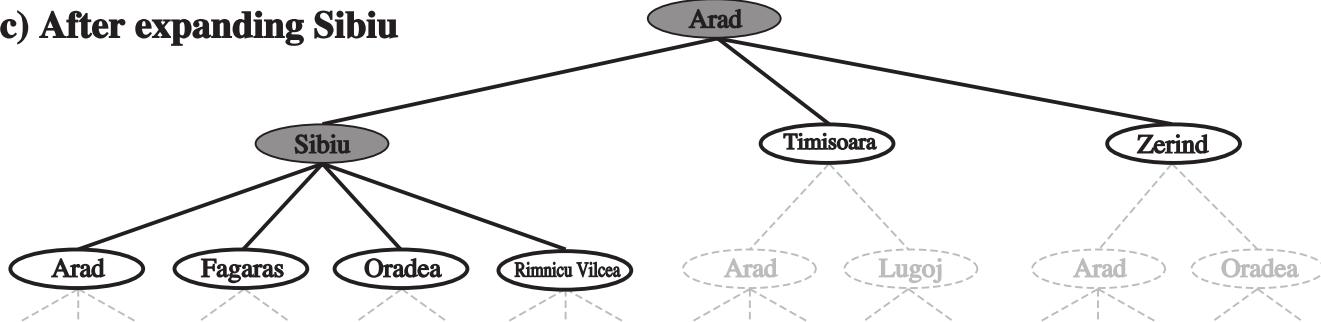
(a) The initial state



(b) After expanding Arad



(c) After expanding Sibiu



Uninformed search strategies

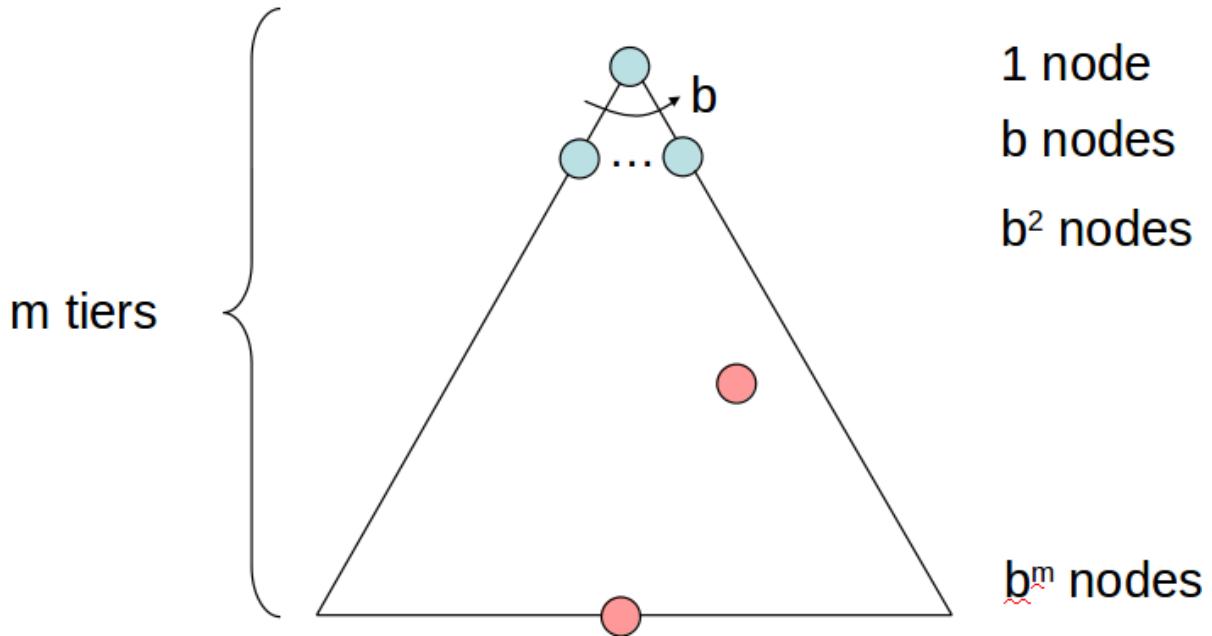
Uninformed search strategies use only the information available in the problem definition. They do not know whether a state looks more promising than some other.

Strategies

- Depth-first search
- Breadth-first search
- Uniform-cost search
- Iterative deepening

Properties of search strategies

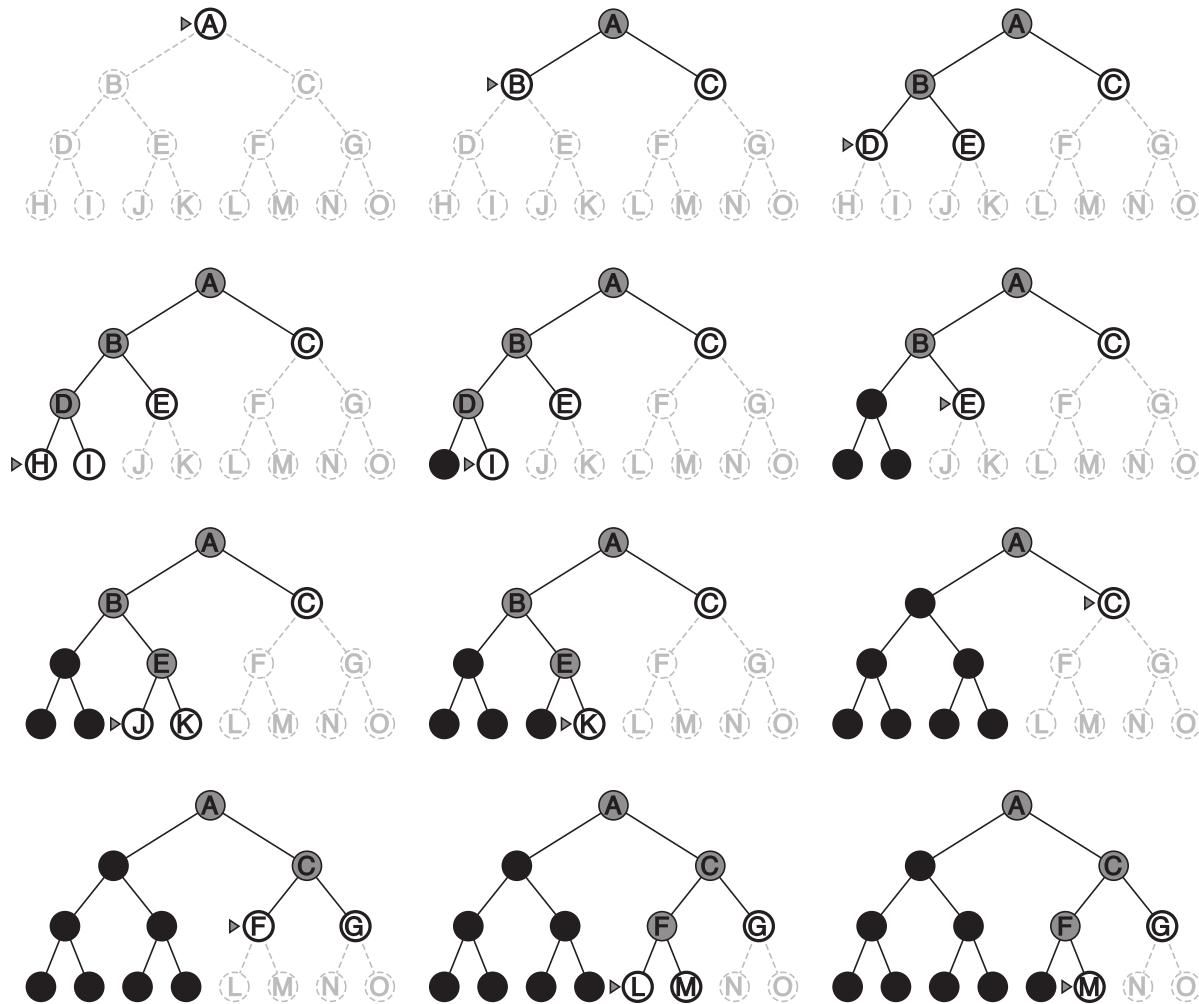
- A strategy is defined by picking the **order of expansion**.
- Strategies are evaluated along the following dimensions:
 - **Completeness**: does it always find a solution if one exists?
 - **Optimality**: does it always find the least-cost solution?
 - **Time complexity**: how long does it take to find a solution?
 - **Space complexity**: how much memory is needed to perform the search?
- Time and complexity are measured in terms of
 - ***b***: maximum branching factor of the search tree
 - ***d***: depth of the least-cost solution
 - the depth of ***S*** is defined as the number of actions from the initial state to ***S***.
 - ***m***: maximum length of any path in the state space (may be ∞)

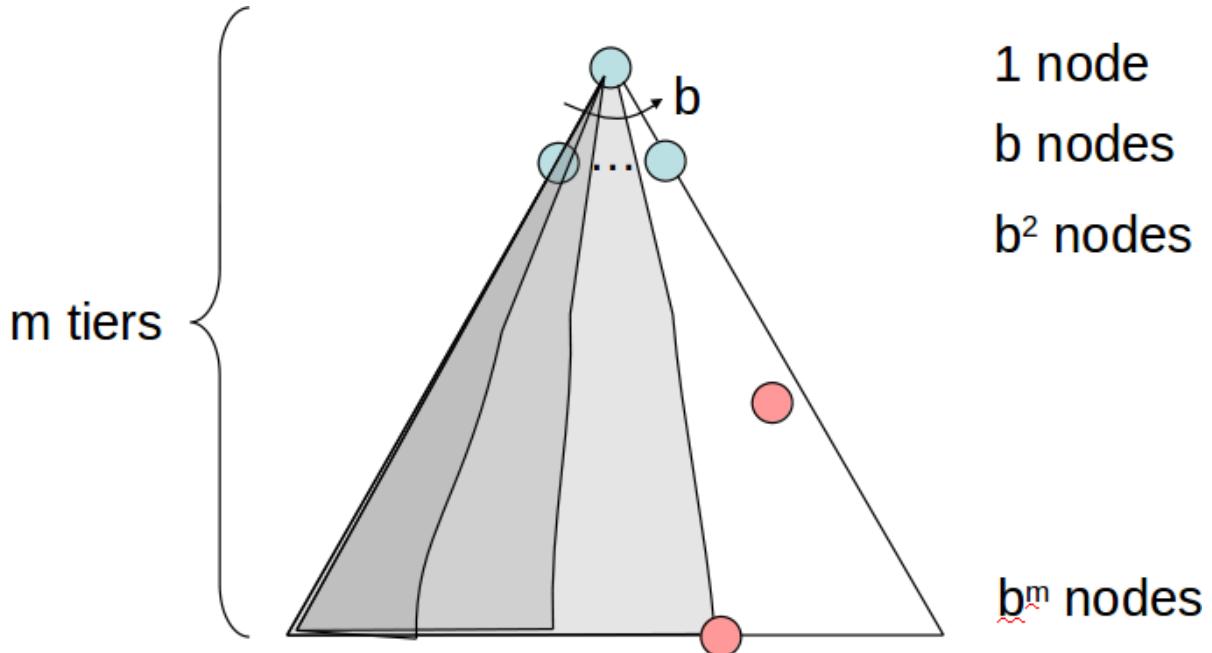


Depth-first search



- **Strategy:** expand the deepest node in the fringe.
- **Implementation:** fringe is a **LIFO stack**.

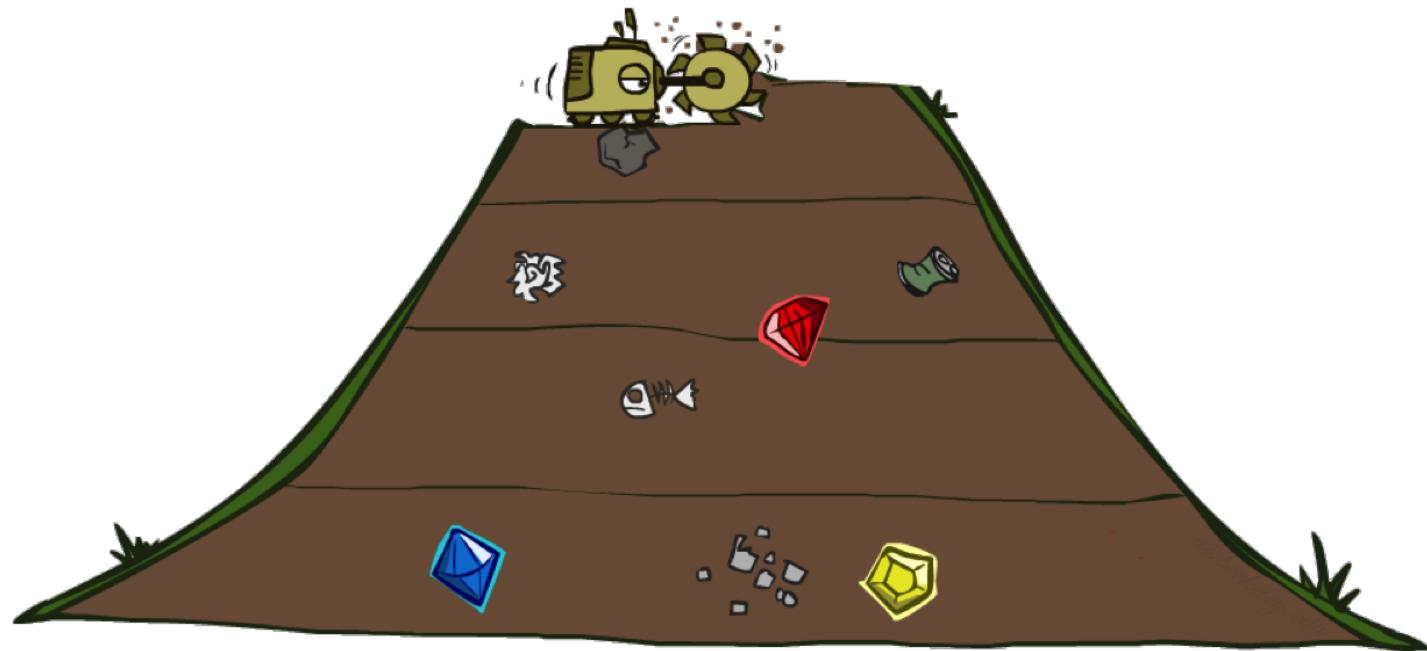




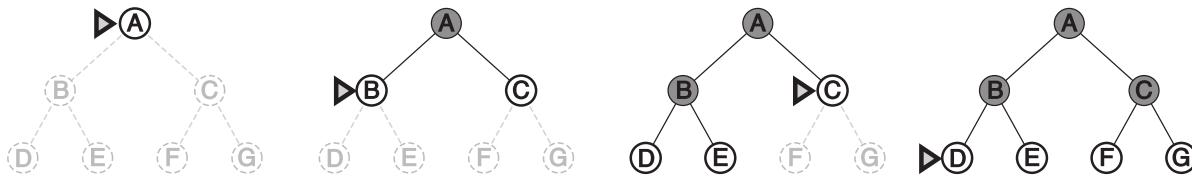
Properties of DFS

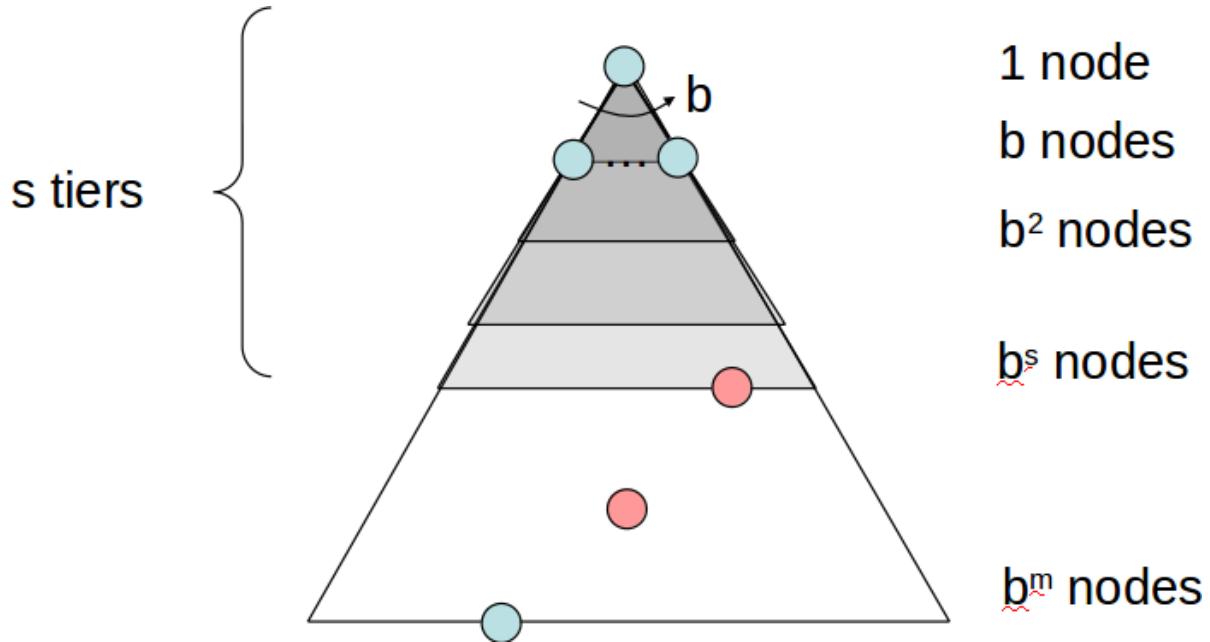
- Completeness:
 - m could be infinite, so only if we prevent cycles (more on this later).
- Optimality:
 - No, DFS finds the leftmost solution, regardless of depth or cost.
- Time complexity:
 - May generate the whole tree (or a good part of it, regardless of d). Therefore $O(b^m)$, which might be much greater than the size of the state space!
- Space complexity:
 - Only store siblings on path to root, therefore $O(bm)$.
 - When all the descendants of a node have been visited, the node can be removed from memory.

Breadth-first search



- **Strategy:** expand the shallowest node in the fringe.
- **Implementation:** fringe is a **FIFO queue**.





Properties of BFS

- **Completeness:**
 - If the shallowest goal node is at some finite depth d , BFS will eventually find it after generating all shallower nodes (provided b is finite).
- **Optimality:**
 - The shallowest goal is not necessarily the optimal one.
 - BFS is optimal only if the path cost is a non-decreasing function of the depth of the node.
- **Time complexity:**
 - If the solution is at depth d , then the total number of nodes generated before finding this node is $b + b^2 + b^3 + \dots + b^d = O(b^d)$
- **Space complexity:**
 - The number of nodes to maintain in memory is the size of the fringe, which will be the largest at the last tier. That is $O(b^d)$

(demo)

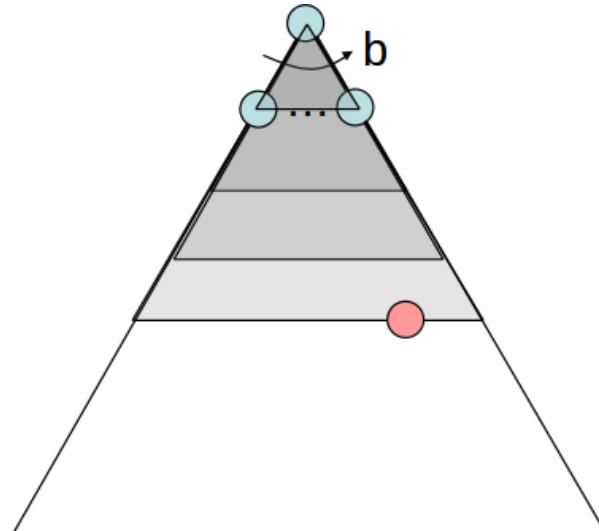
Iterative deepening

Idea: get DFS's space advantages with BFS's time/shallow solution advantages.

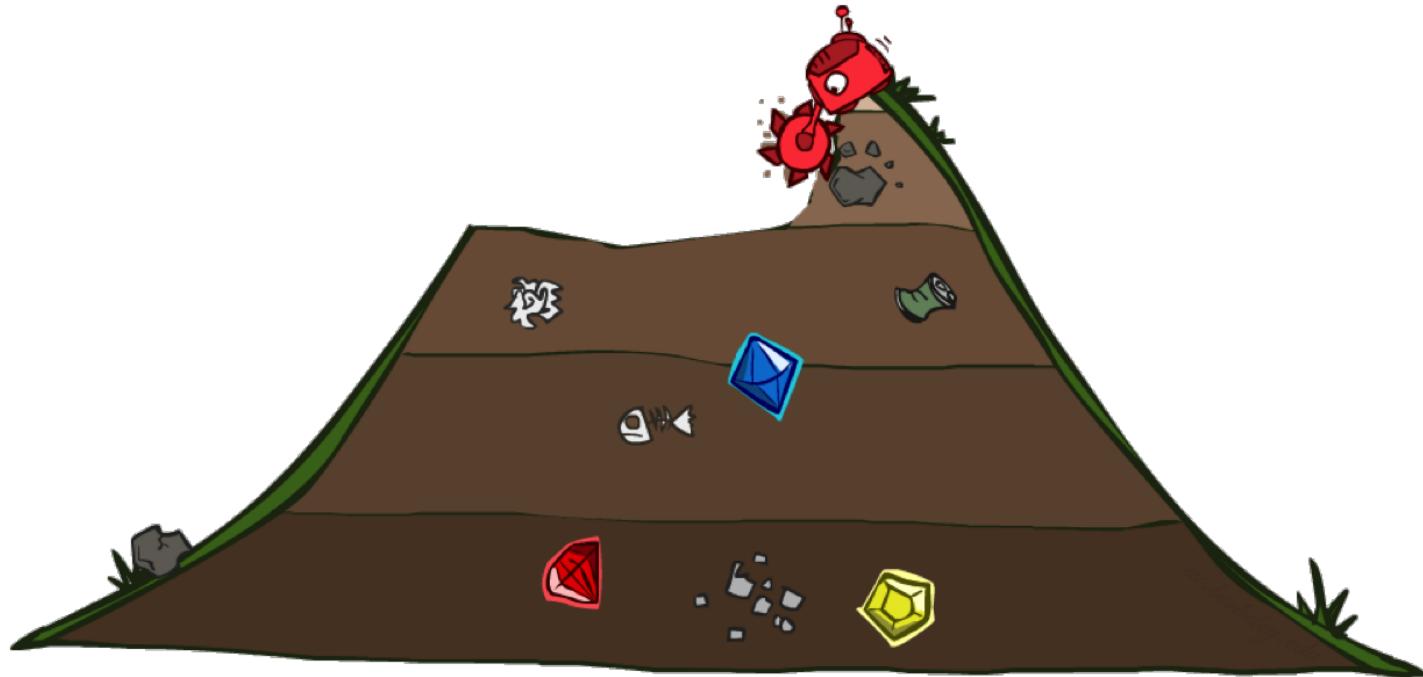
- Run DFS with depth limit 1.
- If no solution, run DFS with depth limit 2.
- If no solution, run DFS with depth limit 3.
 - ...

Exercise

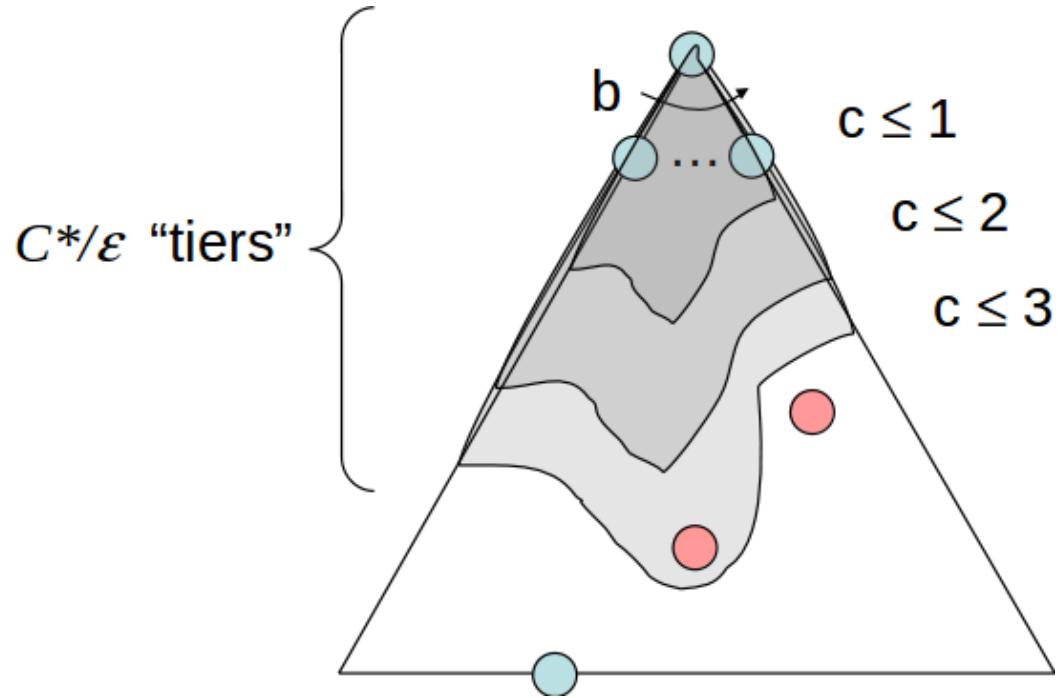
- What are the properties of iterative deepening?
- Isn't this process wastefully redundant?



Uniform-cost search



- **Strategy**: expand the cheapest node in the fringe.
- **Implementation**: fringe is a **priority queue**, using the cumulative cost $g(n)$ from the initial state to node n as priority.

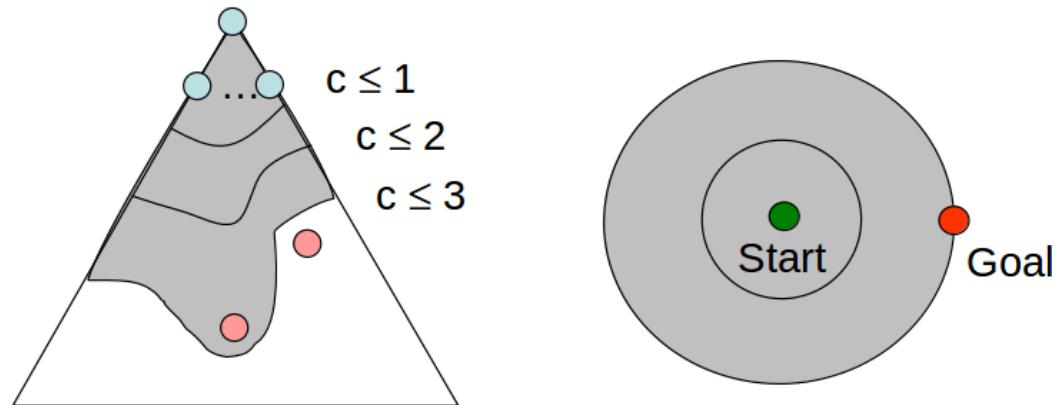


Properties of UCS

- Completeness:
 - Yes, if step cost are all such that $c(s, a, s') \geq \epsilon > 0$. (Why?)
- Optimality:
 - Yes, since UCS expands nodes in order of their optimal path cost.
- Time complexity:
 - Assume C^* is the cost of the optimal solution and that step costs are all $\geq \epsilon$.
 - The "effective depth" is then roughly C^*/ϵ .
 - The worst-case time complexity is $O(b^{C^*/\epsilon})$.
- Space complexity:
 - The number of nodes to maintain is the size of the fringe, so as many as in the last tier $O(b^{C^*/\epsilon})$.

(demo)

Informed search strategies



One of the **issues of UCS** is that it explores the state space in **every direction**, without exploiting information about the (plausible) location of the goal node.

Informed search strategies aim to solve this problem by expanding nodes in the fringe in decreasing order of **desirability**.

- Greedy search
- A*

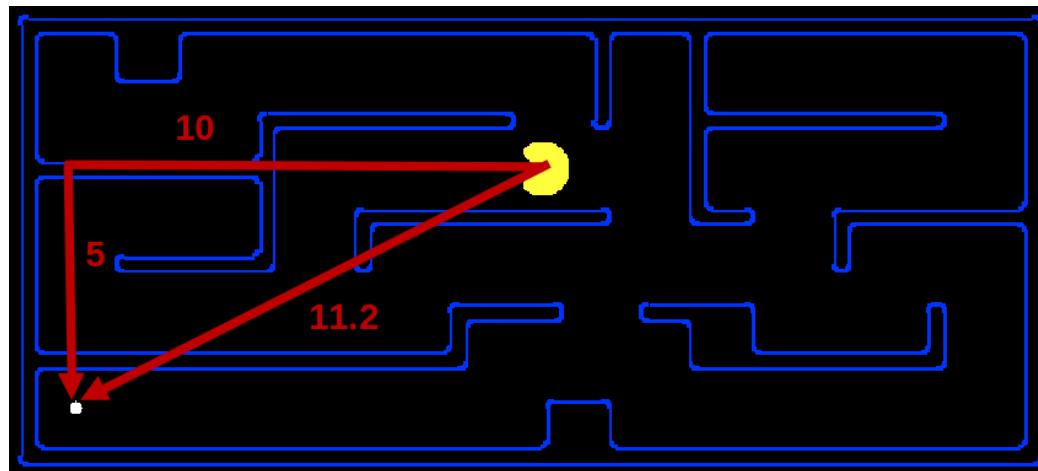
Greedy search



Heuristics

A **heuristic** (or evaluation) function $h(n)$ is:

- a function that **estimates** the cost of the cheapest path from node n to a goal state;
 - $h(n) \geq 0$ for all nodes n
 - $h(n) = 0$ for a goal state.
- is designed for a **particular** search problem.



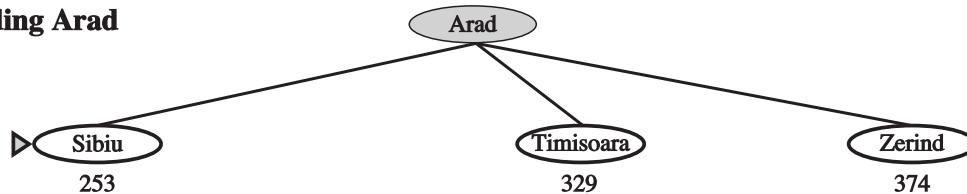
Greedy search

- **Strategy:** expand the node n in the fringe for which $h(n)$ is the lowest.
- **Implementation:** fringe is a priority queue, using $h(n)$ as priority.

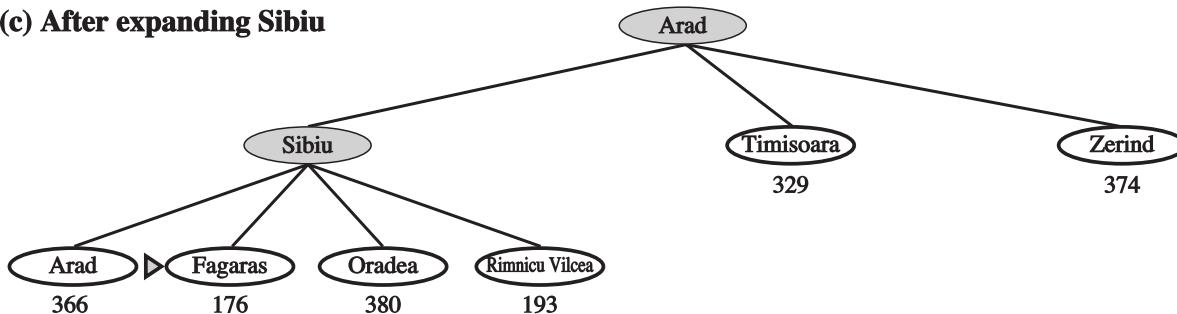
(a) The initial state



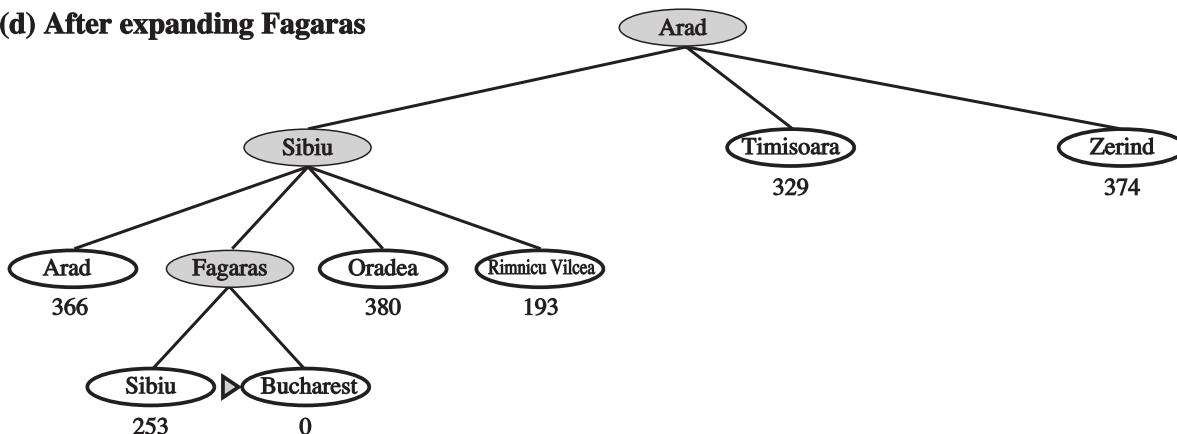
(b) After expanding Arad



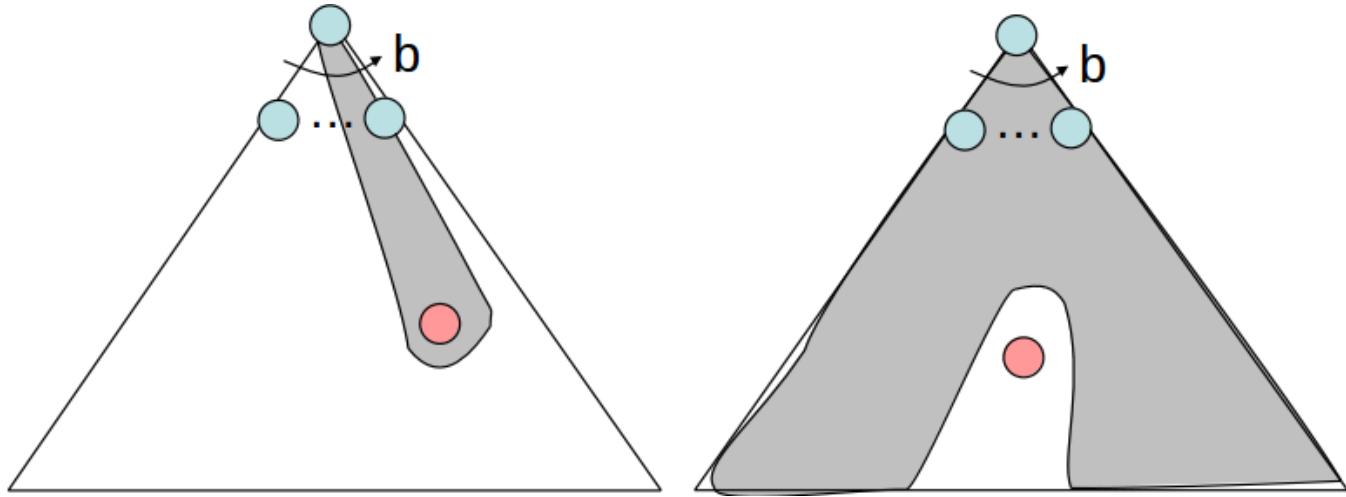
(c) After expanding Sibiu



(d) After expanding Fagaras



$h(n)$ = straight line distance to Bucharest.



At best, greedy search takes you straight to the goal.

At worst, it is like a badly-guided BFS.

Properties of greedy search

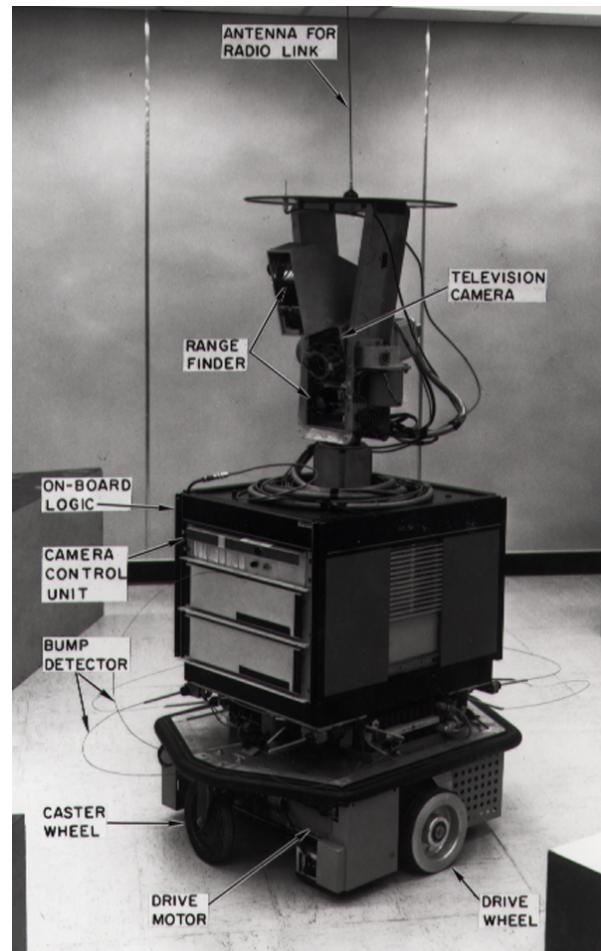
- Completeness:
 - No, unless we prevent cycles (more on this later).
- Optimality:
 - No, e.g. the path via Sibiu and Fagaras is 32km longer than the path through Rimnicu Vilcea and Pitesti.
- Time complexity:
 - $O(b^m)$, unless we have a good heuristic function.
- Space complexity:
 - $O(b^m)$, unless we have a good heuristic function.

A*



Shakey the Robot

- A* was first proposed in 1968 to improve robot planning.
- Goal was to navigate through a room with obstacles.



A*

- Uniform-cost orders by path cost, or backward cost $g(n)$
- Greedy orders by goal proximity, or forward cost $h(n)$
- A* combines the two algorithms and orders by the sum

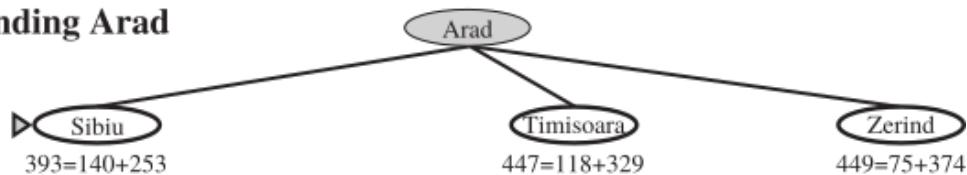
$$f(n) = g(n) + h(n)$$

- $f(n)$ is the estimated cost of cheapest solution through n .

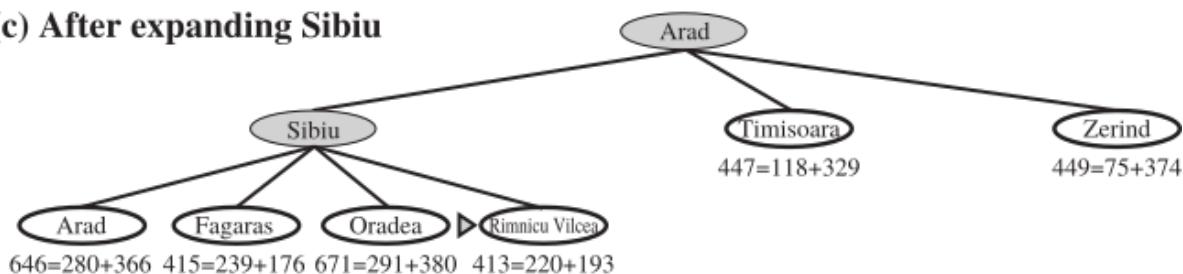
(a) The initial state



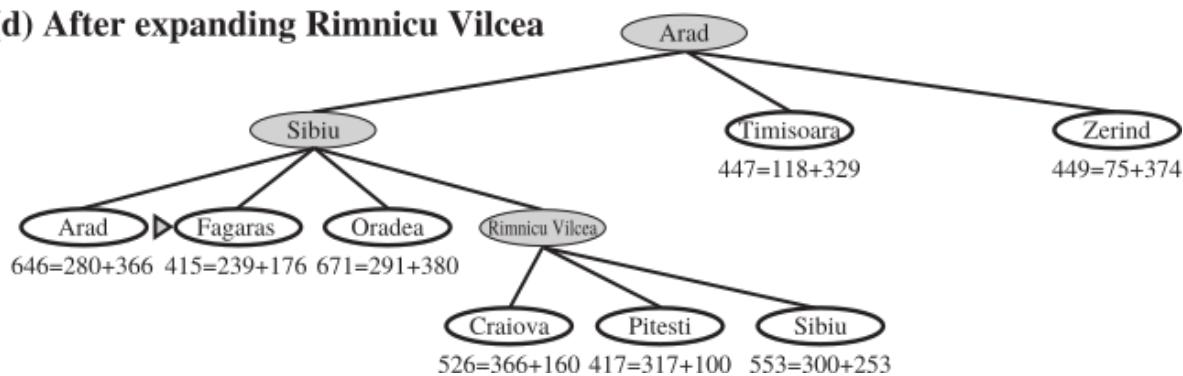
(b) After expanding Arad



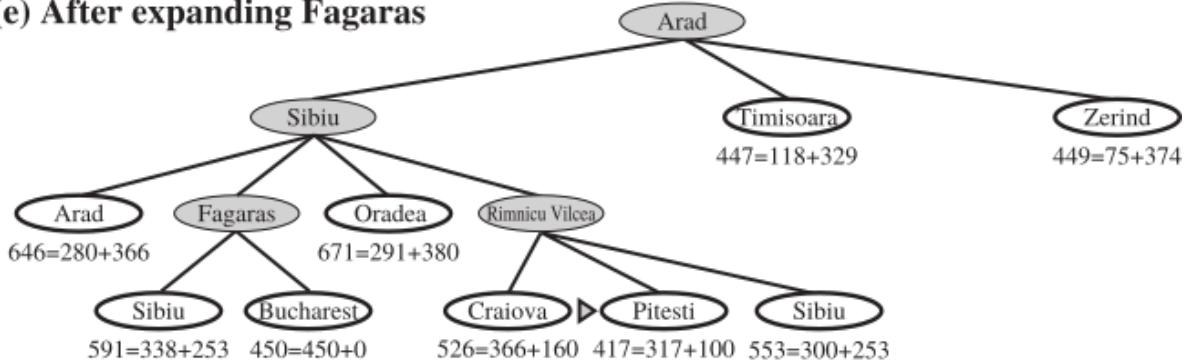
(c) After expanding Sibiu



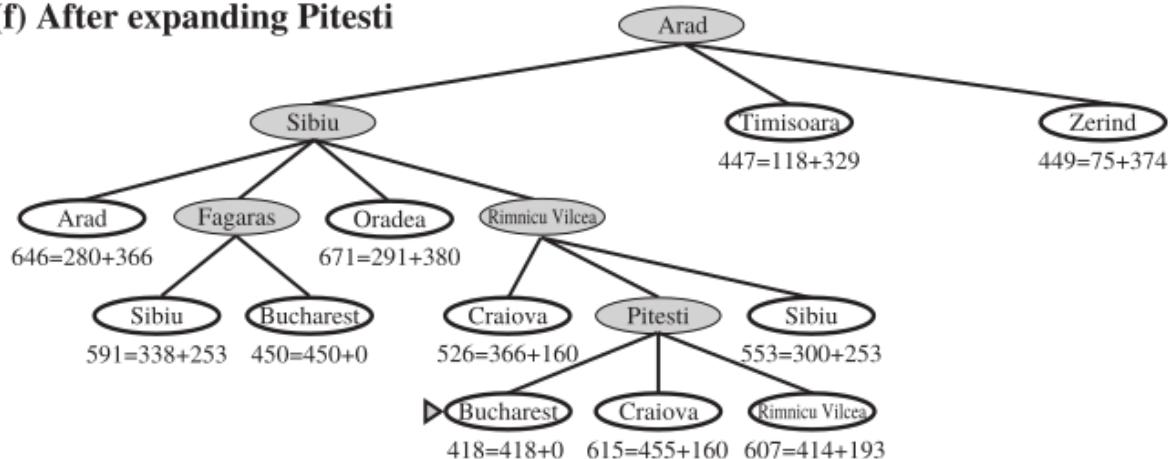
(d) After expanding Rimnicu Vilcea



(e) After expanding Fagaras



(f) After expanding Pitesti



Exercise

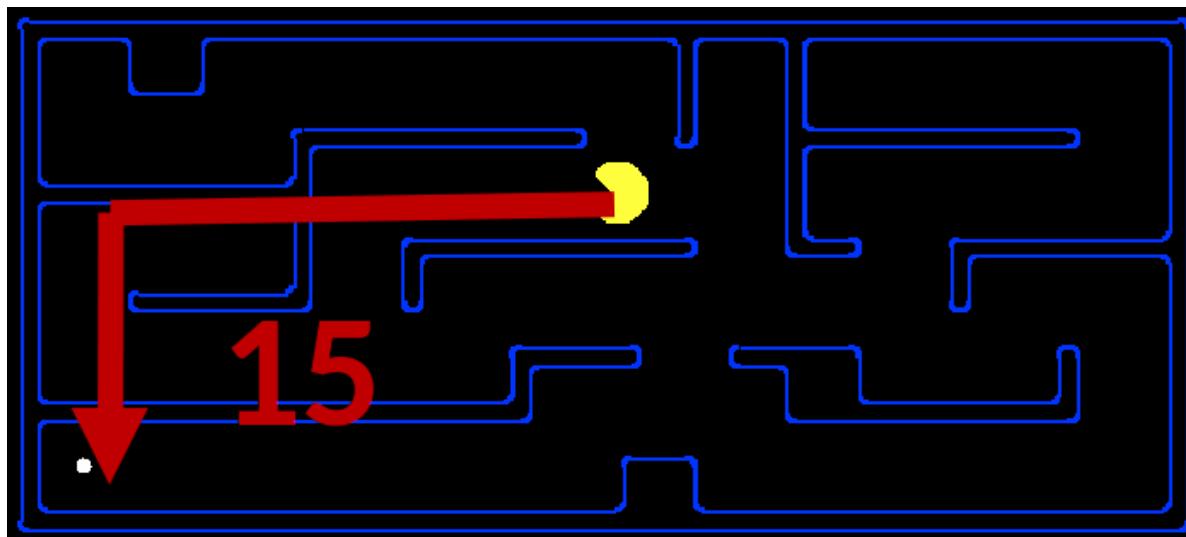
Why doesn't A* stop at step (e), since Bucharest is in the fringe?

Admissible heuristics

A heuristic h is admissible if

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

where $h^*(n)$ is the true cost to a nearest goal.



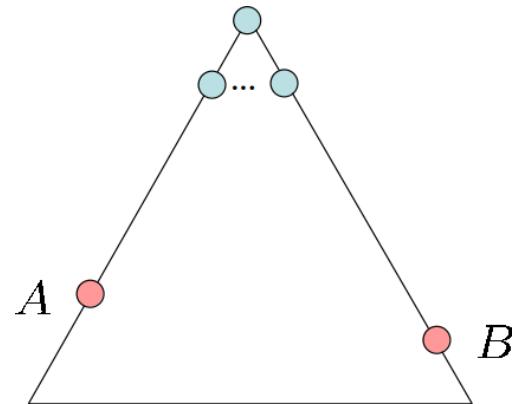
The Manhattan distance is admissible

Optimality of A*

Assumptions:

- A is an optimal goal node
- B is a suboptimal goal node
- h is admissible

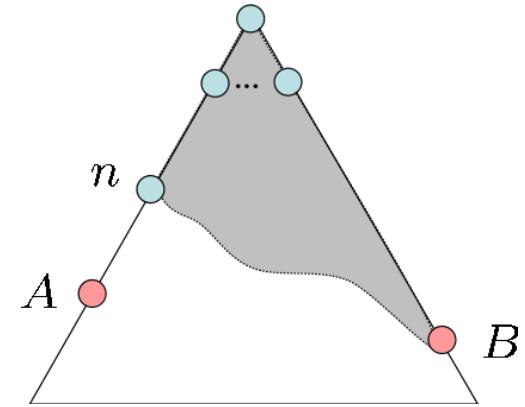
Claim: A will exit the fringe before B .



Proof

Assume B is on the fringe. Some ancestor n of A is on the fringe too.

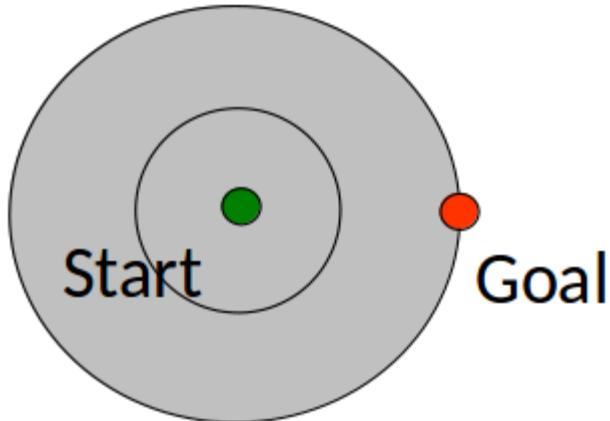
- $f(n) \leq f(A)$
 - $f(n) = g(n) + h(n)$ (by definition)
 - $f(n) \leq g(A)$ (admissibility of h)
 - $f(A) = g(A) + h(A) = g(A)$ ($h = 0$ at a goal)
- $f(A) < f(B)$
 - $g(A) < g(B)$ (B is suboptimal)
 - $f(A) < f(B)$ ($h = 0$ at a goal)
- Therefore, n expands before B .
 - since $f(n) \leq f(A) < f(B)$



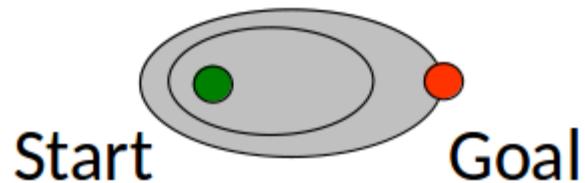
Similarly, all ancestors of A expand before B , including A . Therefore A^* is optimal.

A* contours

- Assume f -costs are non-decreasing along any path.
- We can define **contour levels t** in the state space, that include all nodes n for which $f(n) \leq t$.



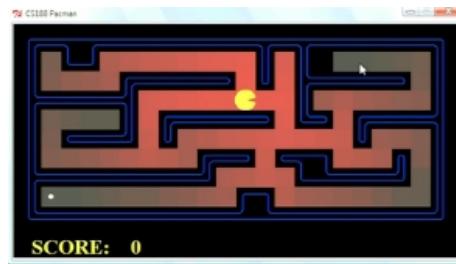
For UCS ($h(n) = 0$ for all n), bands are circular around the start.



For A* with accurate heuristics, bands stretch towards the goal.



Greedy search



UCS



A*

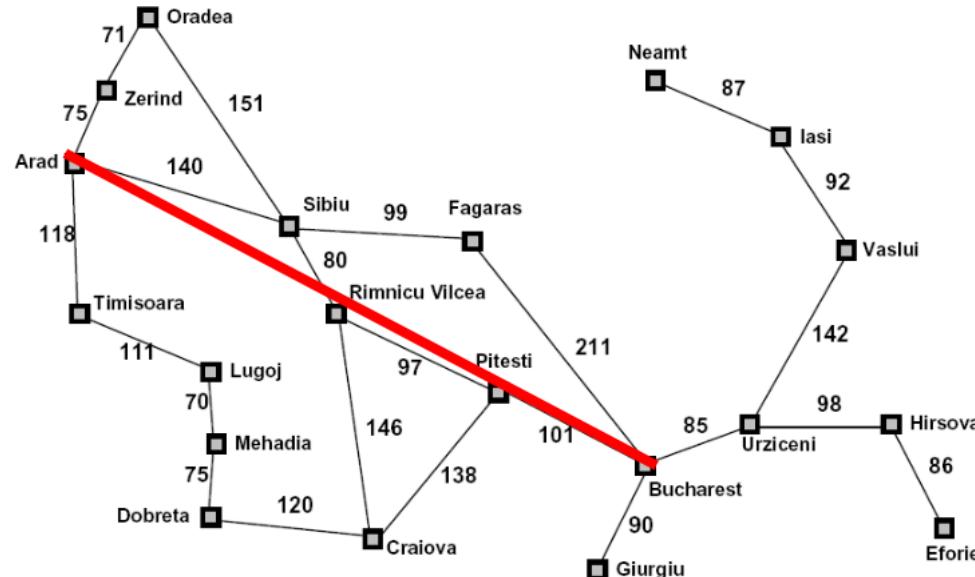
(demo)

Creating admissible heuristics

Most of the work in solving hard search problems optimally is in finding admissible heuristics.

Admissible heuristics can be derived from the exact solutions to [relaxed problems](#), where new actions are available.

366



Dominance

- If h_1 and h_2 are both admissible and if $h_2(n) \geq h_1(n)$ for all n , then h_2 dominates h_1 and is better for search.
- Given any admissible heuristics h_a and h_b ,

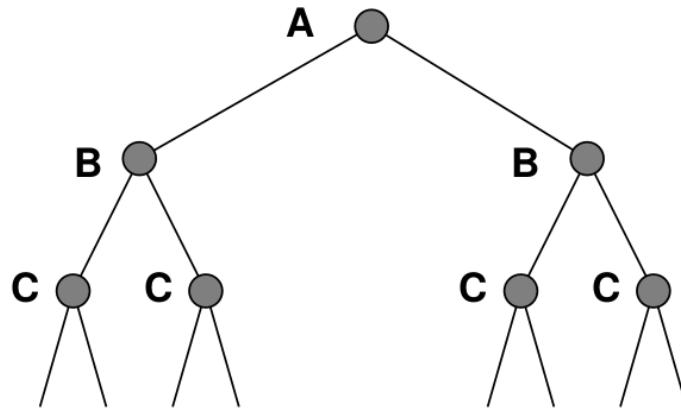
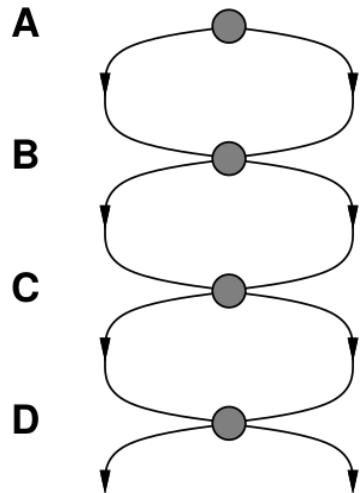
$$h(n) = \max(h_a(n), h_b(n))$$

is also admissible and dominates h_a and h_b .

Learning heuristics from experience

- Assuming an **episodic** environment, an agent can **learn** good heuristics by playing the game many times.
- Each optimal solution s^* provides **training examples** from which $h(n)$ can be learned.
- Each example consists of a state n from the solution path and the actual cost $g(s^*)$ of the solution from that point.
- The mapping $n \rightarrow g(s^*)$ can be learned with **supervised learning** algorithms.
 - Linear models, Neural networks, etc.

Graph search



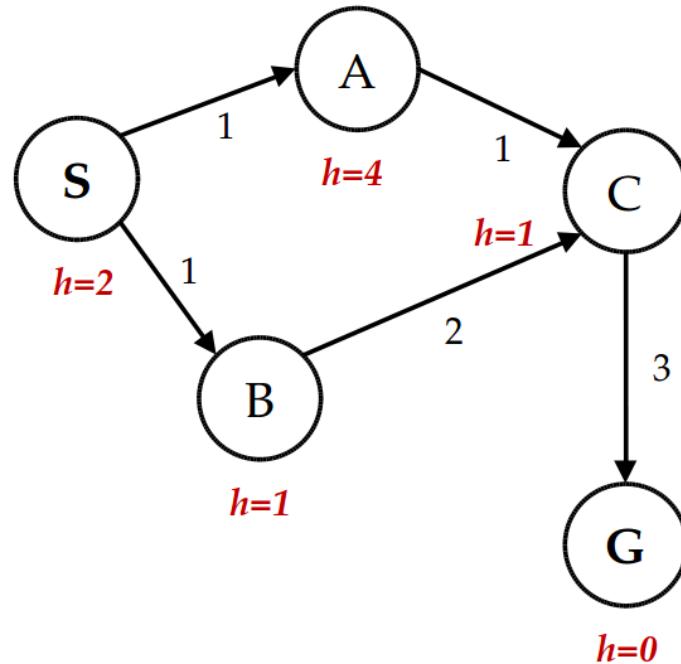
The failure to detect **repeated states** can turn a linear problem into an exponential one. It can also lead to non-terminating searches.

Redundant paths and cycles can be avoided by **keeping track** of the states that have been **explored**. This amounts to grow a tree directly on the state-space graph.

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

A* graph-search gone wrong?

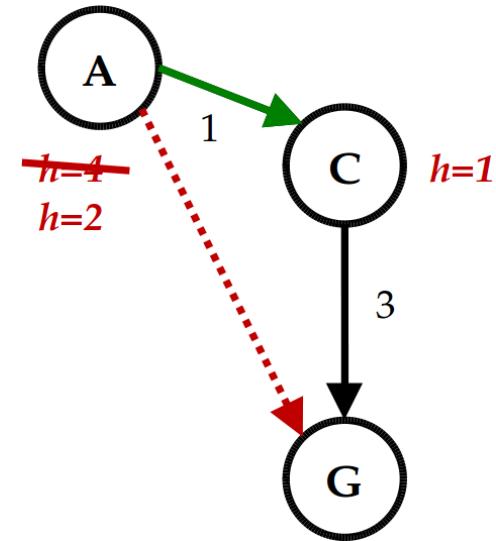
- We start at S and G is a goal state.
- Which path does graph search find?



Consistent heuristics

A heuristic h is consistent if for every n and every successor n' generated by any action a ,

$$h(n) \leq c(n, a, n') + h(n').$$



Consequences of consistent heuristics:

- $f(n)$ is non-decreasing along any path.
- $h(n)$ is admissible.
- With a consistent heuristic, graph-search A* is optimal.

Recap example: Super Mario



- Task environment?
 - performance measure, environment, actuators, sensors?
- Type of environment?
- Search problem?
 - initial state, actions, transition model, goal test, path cost?
- Good heuristic?



A* in action

Summary

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored.
- Variety of uninformed search strategies (DFS, BFS, UCS, Iterative deepening).
- Heuristic functions estimate costs of shortest paths. Good heuristic can dramatically reduce search cost.
- Greedy best-first search expands lowest h , which shows to be incomplete and not always optimal.
- A* search expands lowest $f = g + h$. This strategy is complete and optimal.
- Graph search can be exponentially more efficient than tree search.

Introduction to Artificial Intelligence

Lecture 3: Games and Adversarial search

Prof. Gilles Louppe
g.louppe@uliege.be



Nash Equilibrium| HD | With Subtitles|



Later bekij...
...



Delen



Today

- How to act rationally in a **multi-agent** environment?
- How to anticipate and respond to the **arbitrary behavior** of other agents?
- Adversarial search
 - Minimax
 - $\alpha - \beta$ pruning
 - H-Minimax
 - Expectiminimax
 - Monte Carlo Tree Search
- Modeling assumptions
- State-of-the-art agents.

Minimax

Games

- A **game** is a multi-agent environment where agents may have either **conflicting** or **common** interests.
- Opponents may act **arbitrarily**, even if we assume a deterministic fully observable environment.
 - The solution to a game is a **strategy** specifying a move for every possible opponent reply.
 - This is different from search where a solution is a **fixed sequence**.
- Time is often **limited**.

Types of games

- Deterministic or stochastic?
- Perfect or imperfect information?
- Two or more players?

Formal definition

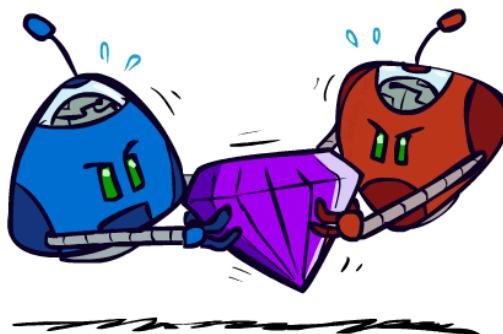
A **game** is formally defined as a kind of search problem with the following components:

- A representation of the **states** of the agents and their environment.
- The **initial state** s_0 of the game.
- A function $\text{player}(s)$ that defines which **player** $p \in \{1, \dots, N\}$ has the move in state s .
- A description of the legal **actions** (or **moves**) available to a state s , denoted $\text{actions}(s)$.
- A **transition model** that returns the state $s' = \text{result}(s, a)$ that results from doing action a in state s .
- A **terminal test** which determines whether the game is over.

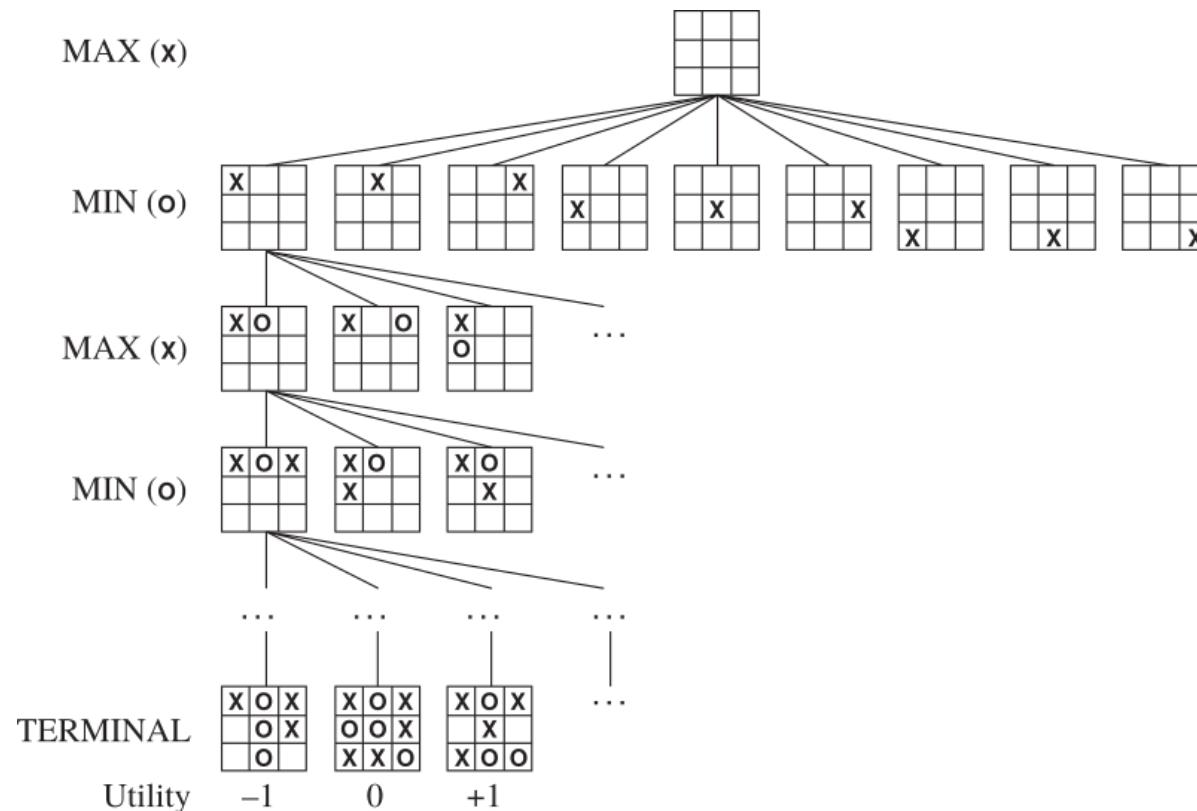
- A utility function $\text{utility}(s, p)$ (or payoff) that defines the final numeric value for a game that ends in s for a player p .
 - E.g., $1, 0$ or $\frac{1}{2}$ if the outcome is win, loss or draw.
- Together, the initial state, the $\text{actions}(s)$ function and the $\text{result}(s, a)$ function define the game tree.
 - Nodes are game states.
 - Edges are actions.

Zero-sum games

- In a **zero-sum** game, the total payoff to all players is **constant** for all games.
 - e.g., in chess: $0 + 1, 1 + 0$ or $\frac{1}{2} + \frac{1}{2}$.
- For two-player games, agents share the **same utility** function, but one wants to **maximize** it while the other wants to **minimize** it.
 - MAX maximizes the game's **utility** function.
 - MIN minimizes the game's **utility** function.
- **Strict competition.**
 - If one wins, the other loses, and vice-versa.



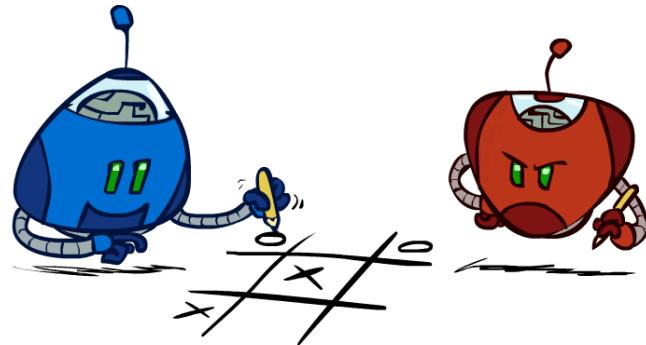
Tic-Tac-Toe game tree



What is an optimal strategy (or perfect play)? How do we find it?

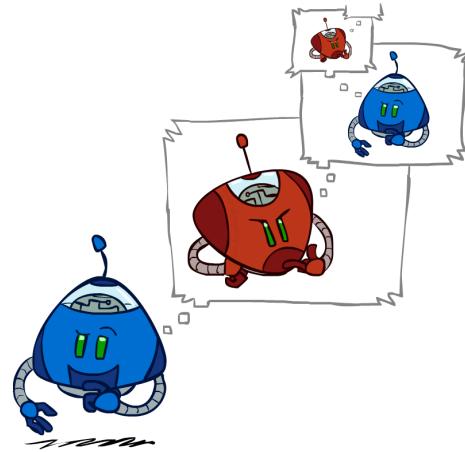
Assumptions

- We assume a deterministic, turn-taking, two-player zero-sum game with perfect information.
 - e.g., Tic-Tac-Toe, Chess, Checkers, Go, etc.
- We will call our two players **MAX** and **MIN**. **MAX** moves first.



Adversarial search

- In a search problem, the optimal solution is a sequence of actions leading to a goal state.
 - i.e., a terminal state where MAX wins.
- In a game, the opponent (MIN) may react **arbitrarily** to a move.
- Therefore, a player (MAX) must define a contingent **strategy** which specifies
 - its moves in the initial state,
 - its moves in the states resulting from every possible response by MIN,
 - its moves in the states resulting from every possible response by MIN in those states, ...



Minimax

The **minimax value** $\text{minimax}(s)$ is the largest achievable payoff (for MAX) from state s , assuming an **optimal adversary** (MIN).

$\text{MINIMAX}(s) =$

$$\begin{cases} \text{UTILITY}(s) & \text{if } \text{TERMINAL-TEST}(s) \\ \max_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if } \text{PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if } \text{PLAYER}(s) = \text{MIN} \end{cases}$$

The **optimal** next move (for MAX) is to take the action that maximizes the minimax value in the resulting state.

- Assuming that MIN is an optimal adversary that maximizes the **worst-case outcome** for MAX.
- This is equivalent to not making an assumption about the strength of the opponent.

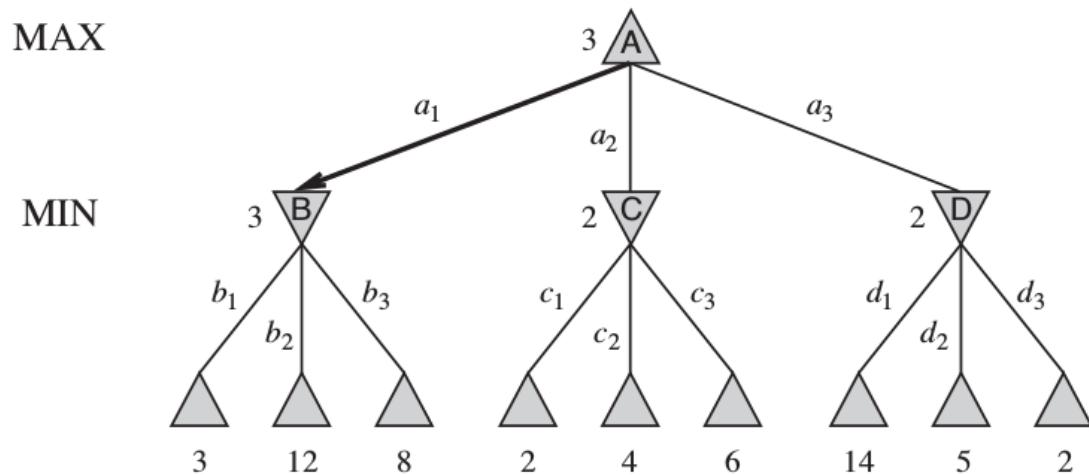


Figure 5.2 A two-ply game tree. The \triangle nodes are “MAX nodes,” in which it is MAX’s turn to move, and the ∇ nodes are “MIN nodes.” The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX’s best move at the root is a_1 , because it leads to the state with the highest minimax value, and MIN’s best reply is b_1 , because it leads to the state with the lowest minimax value.

Properties of Minimax

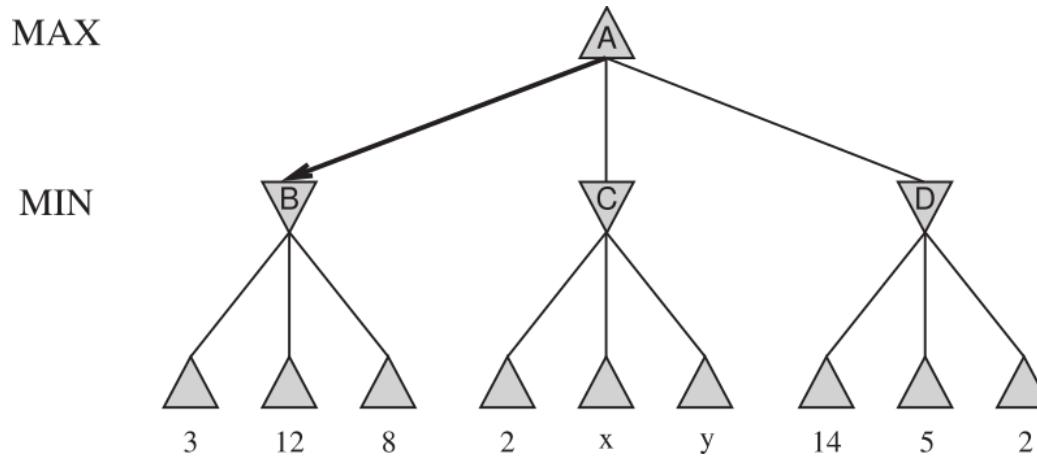
- Completeness:
 - Yes, if tree is finite.
- Optimality:
 - Yes, if MIN is an optimal opponent.
 - What if MIN is suboptimal?
 - Show that MAX will do even better.
 - What if MIN is suboptimal and predictable?
 - Other strategies might do better than Minimax. However they may do worse on an optimal opponent.

Minimax efficiency

- Assume $\text{minimax}(s)$ is implemented using its recursive definition.
- How efficient is minimax?
 - Time complexity: same as DFS, i.e., $O(b^m)$.
 - Space complexity:
 - $O(bm)$, if all actions are generated at once, or
 - $O(m)$, if actions are generated one at a time.

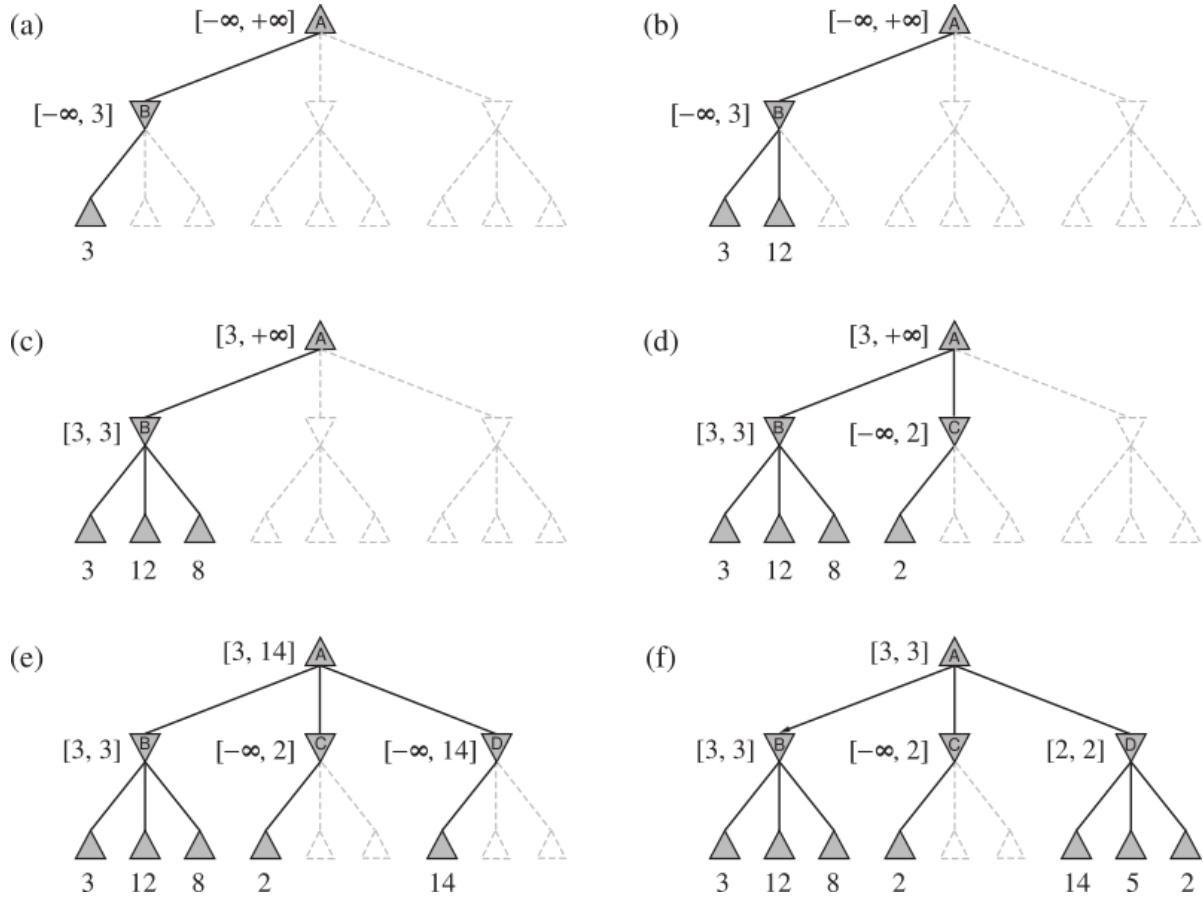
Do we need to explore the whole game tree?

Pruning



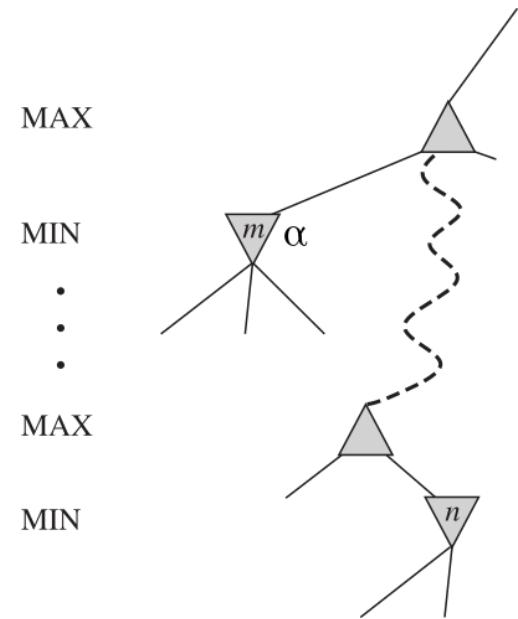
$$\begin{aligned}\text{MINIMAX}(\text{root}) &= \max(\min(3, 12, 8), \min(2, x, y), \min(14, 5, 2)) \\ &= \max(3, \min(2, x, y), 2) \\ &= \max(3, z, 2) \quad \text{where } z = \min(2, x, y) \leq 2 \\ &= 3.\end{aligned}$$

Therefore, it is possible to compute the **correct** minimax decision **without** looking at every node in the tree.



We want to compute $v = \text{minimax}(n)$, for $\text{player}(n)=\text{MIN}$.

- We loop over n 's children.
- The minimax values are being computed one at a time and v is updated iteratively.
- Let α be the best value (i.e., the highest) at any choice point along the path for MAX.
- If v becomes lower than α , then n will never be reached in actual play.
- Therefore, we can stop iterating over the remaining n 's other children.



Similarly, β is defined as the best value (i.e., lowest) at any choice point along the path for MIN. We can halt the expansion of a MAX node as soon as v becomes larger than β .

α - β pruning

- Updates the values of α and β as the path is expanded.
- Prune the remaining branches (i.e., terminate the recursive calls) as soon as the value of the current node is known to be worse than the current α or β value for MAX or MIN, respectively.

α - β search

function ALPHA-BETA-SEARCH(*state*) **returns** an action

$v \leftarrow \text{MAX-VALUE}(\textit{state}, -\infty, +\infty)$

return the *action* in ACTIONS(*state*) with value v

function MAX-VALUE(*state*, α , β) **returns** a utility value

if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)

$v \leftarrow -\infty$

for each *a* **in** ACTIONS(*state*) **do**

$v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$

if $v \geq \beta$ **then return** v

$\alpha \leftarrow \text{MAX}(\alpha, v)$

return v

function MIN-VALUE(*state*, α , β) **returns** a utility value

if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)

$v \leftarrow +\infty$

for each *a* **in** ACTIONS(*state*) **do**

$v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$

if $v \leq \alpha$ **then return** v

$\beta \leftarrow \text{MIN}(\beta, v)$

return v

Properties of α - β search

- Pruning has **no effect** on the minimax values. Therefore, **completeness** and **optimality** are preserved from Minimax.
- Time complexity:
 - The effectiveness depends on the order in which the states are examined.
 - If states could be examined in **perfect order**, then $\alpha - \beta$ search examines only $O(b^{m/2})$ nodes to pick the best move, vs. $O(b^m)$ for minimax.
 - $\alpha - \beta$ can solve a tree twice as deep as minimax can in the same amount of time.
 - Equivalent to an effective branching factor \sqrt{b} .
- Space complexity: $O(m)$, as for Minimax.

Game tree size



Chess:

- $b \approx 35$ (approximate average branching factor)
- $d \approx 100$ (depth of a game tree for typical games)
- $b^d \approx 35^{100} \approx 10^{154}$.
- For $\alpha - \beta$ search and perfect ordering, we get $b^{d/2} \approx 35^{50} = 10^{77}$.

Finding the exact solution with Minimax remains **intractable**.

Transposition table

- Repeated states occur frequently because of **transpositions**: distinct permutations of the move sequence end in a same position.
- Similarly to the `closed` set in Graph-Search (Lecture 2), it is worth storing the evaluation of a state such that further occurrences of the state do not have to be recomputed.

What data structure should be used to efficiently store and look-up values of positions?

Imperfect real-time decisions

- Under **time constraints**, searching for the exact solution is not feasible in most realistic games.
- Solution: cut the search earlier.
 - Replace the **utility(s)** function with a heuristic **evaluation function eval(s)** that estimates the state utility.
 - Replace the terminal test by a **cutoff test** that decides when to stop expanding a state.

H-MINIMAX(s, d) =

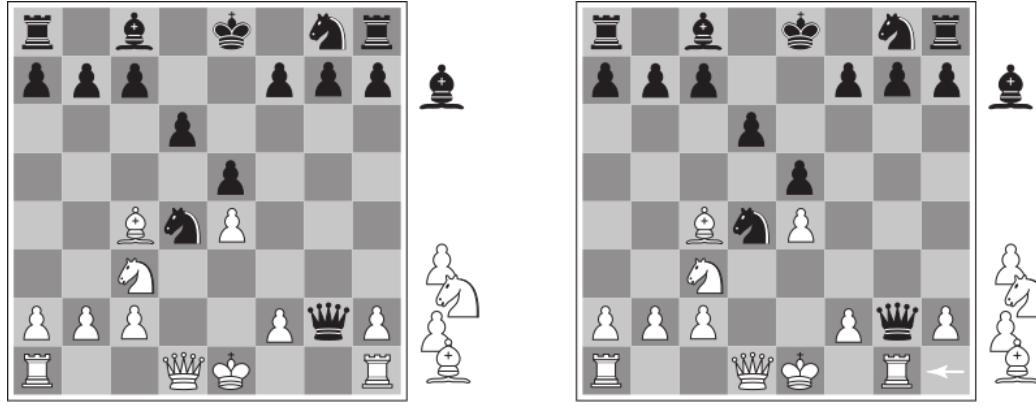
$$\begin{cases} \text{EVAL}(s) & \text{if CUTOFF-TEST}(s, d) \\ \max_{a \in \text{Actions}(s)} \text{H-MINIMAX}(\text{RESULT}(s, a), d + 1) & \text{if } \text{PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{H-MINIMAX}(\text{RESULT}(s, a), d + 1) & \text{if } \text{PLAYER}(s) = \text{MIN}. \end{cases}$$

Can $\alpha - \beta$ search be adapted to implement H-Minimax?

Evaluation functions

- An evaluation function $\text{eval}(s)$ returns an **estimate** of the expected utility of the game from a given position s .
- The computation **must be short** (that is the whole point to search faster).
- Ideally, the evaluation should **order** states in the same way as in Minimax.
 - The evaluation values may be different from the true minimax values, as long as order is preserved.
- In non-terminal states, the evaluation function should be strongly **correlated** with the actual chances of winning.

Quiescence



(a) White to move

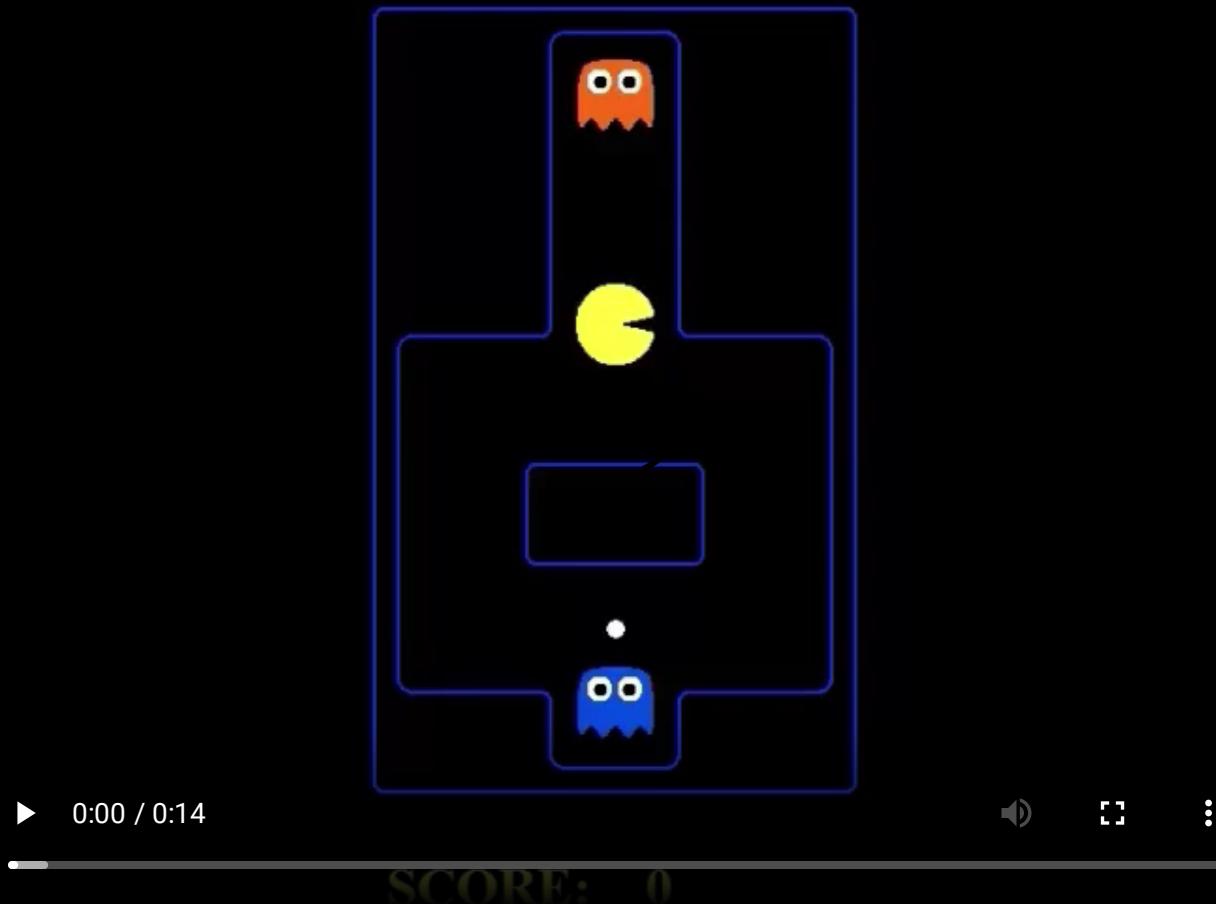
(b) White to move

- These states only differ in the position of the rook at lower right.
- However, Black has advantage in (a), but not in (b).
- If the search stops in (b), Black will not see that White's next move is to capture its Queen, gaining advantage.
- Cutoff should only be applied to positions that are **quiescent**.
 - i.e., states that are unlikely to exhibit wild swings in value in the near future.

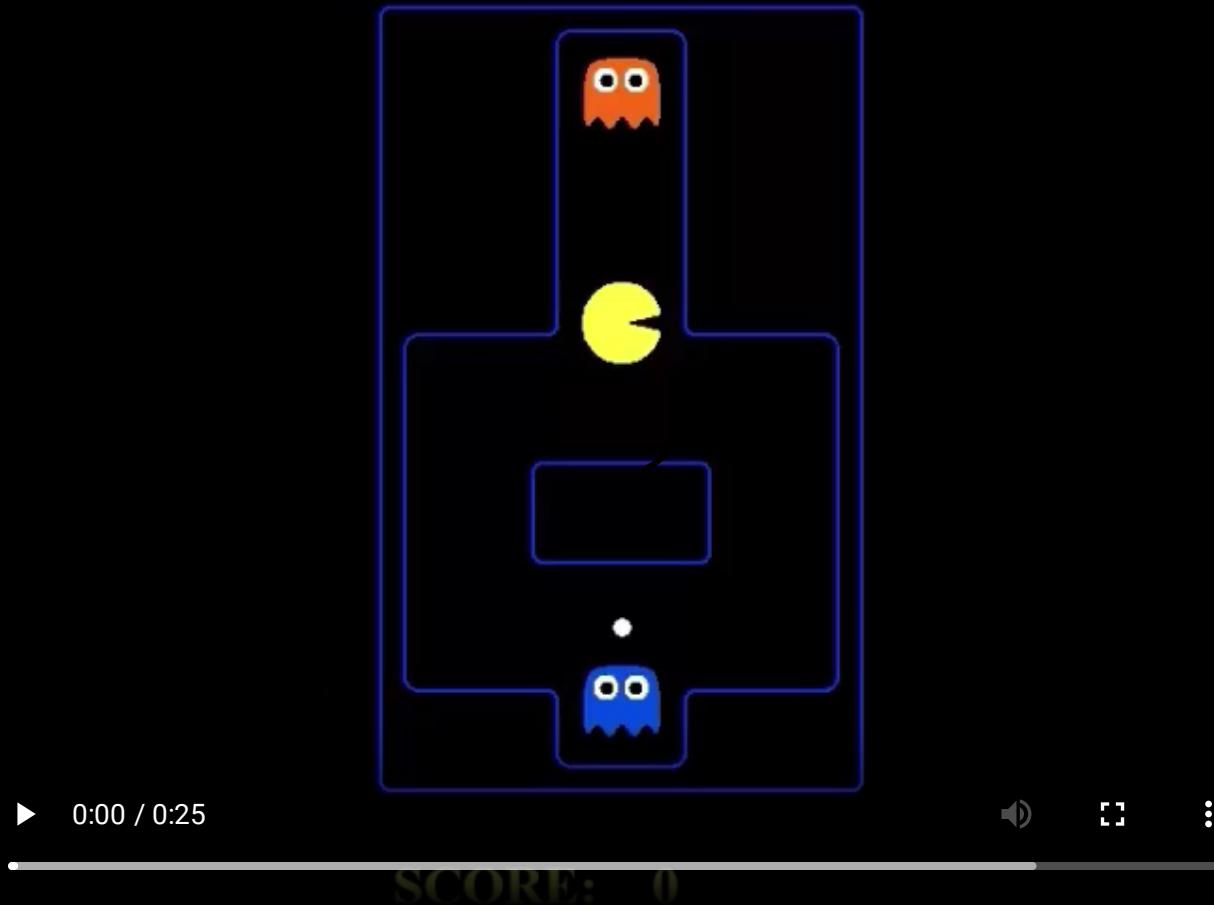
The horizon effect

Evaluations functions are **always imperfect**.

- If not looked deep enough, **bad moves** may appear as **good moves** (as estimated by the evaluation function) because their consequences are hidden beyond the search horizon.
 - and vice-versa!
- Often, the deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters.



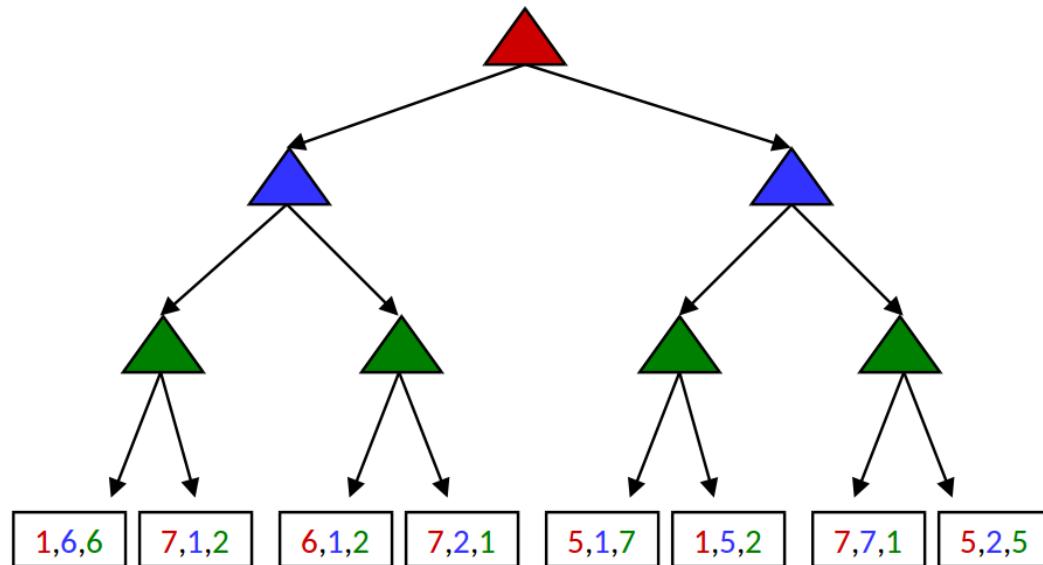
Cutoff at depth 2, evaluation = the closer to the dot, the better.



Cutoff at depth 10, evaluation = the closer to the dot, the better.

Multi-agent games

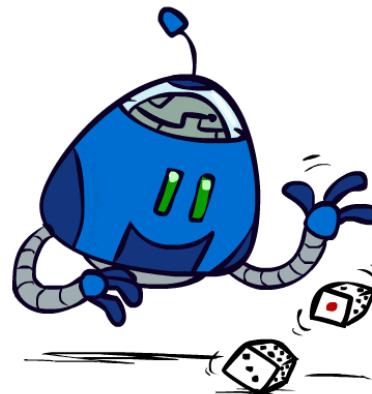
- What if the game is not zero-sum, or has **multiple players**?
- Generalization of Minimax:
 - Terminal states are labeled with utility **tuples** (1 value per player).
 - Intermediate states are also labeled with utility tuples.
 - Each player maximizes its own component.
 - May give rise to cooperation and competition dynamically.



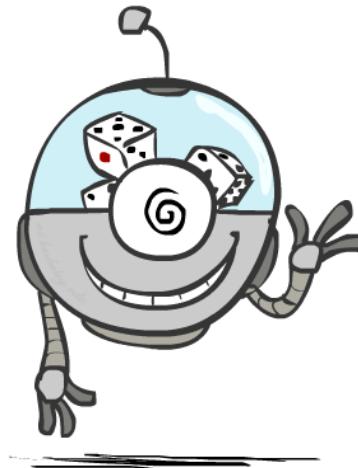
Stochastic games

Stochastic games

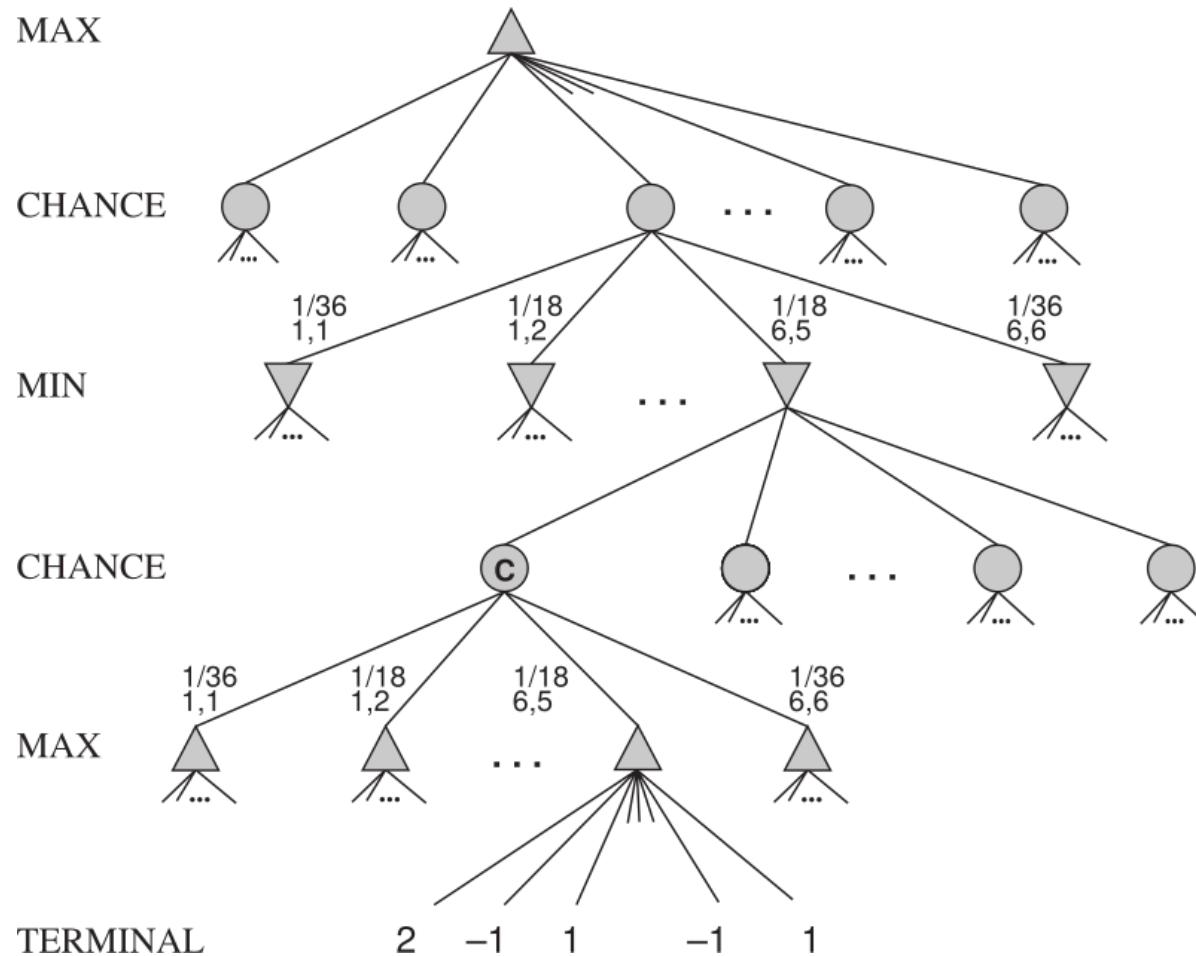
- In real life, many unpredictable external events can put us into unforeseen situations.
- Games that mirror this unpredictability are called **stochastic games**. They include a random element, such as:
 - explicit randomness: rolling a dice;
 - actions may fail: when moving a robot, wheels might slip.



- In a game tree, this random element can be **modeled** with **chance nodes** that map a state-action pair to the set of possible outcomes, along with their respective **probability**.
- This is equivalent to considering the environment as an extra **random agent** player that moves after each of the other players.



Stochastic game tree



Expectimax

- Because of the uncertainty in the action outcomes, states no longer have a **definite minimax** value.
- However, we can calculate the **expected** value of a state under optimal play by the opponent.
 - i.e., the average over all possible outcomes of the chance nodes.
 - **minimax** values correspond instead to the worst-case outcome.

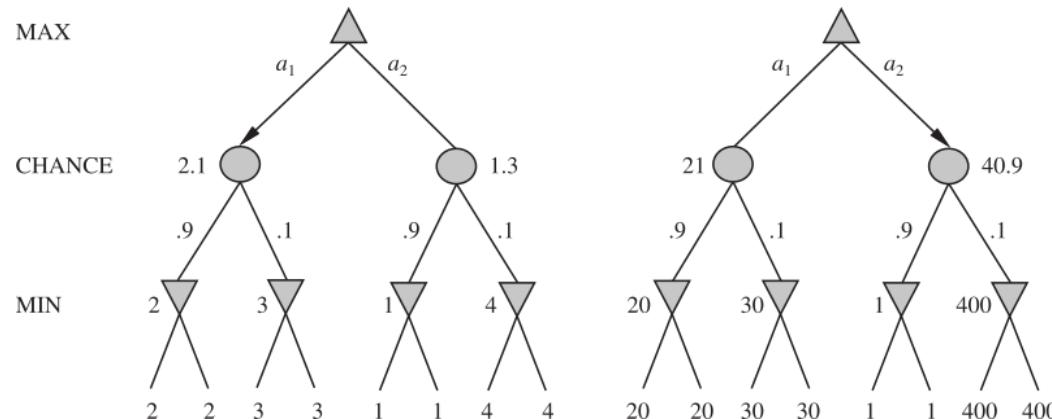
EXPECTIMINIMAX(s) =

$$\begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_a \text{EXPECTIMINIMAX}(\text{RESULT}(s, a)) & \text{if } \text{PLAYER}(s) = \text{MAX} \\ \min_a \text{EXPECTIMINIMAX}(\text{RESULT}(s, a)) & \text{if } \text{PLAYER}(s) = \text{MIN} \\ \sum_r P(r) \text{EXPECTIMINIMAX}(\text{RESULT}(s, r)) & \text{if } \text{PLAYER}(s) = \text{CHANCE} \end{cases}$$

Does taking the rational move mean the agent will be successful?

Evaluation functions

- As for $\text{minimax}(n)$, the value of $\text{expectiminimax}(n)$ may be approximated by stopping the recursion early and using an evaluation function.
- However, to obtain correct move, the evaluation function should be a **positive linear transformation** of the expected utility of the state.
 - It is not enough for the evaluation function to just be order-preserving.
- If we assume bounds on the utility function, $\alpha - \beta$ search can be adapted to stochastic games.



An order-preserving transformation on leaf values changes the best move.

Monte Carlo Tree Search

Random playout evaluation

- To evaluate a state, have the algorithm play **against itself** using **random moves**, thousands of times.
- The sequence of random moves is called a **random playout**.
- Use the proportion of wins as the state evaluation.
- This strategy does **not require domain knowledge!**
 - The game engine is all that is needed.

Monte Carlo Tree Search

The focus of MCTS is the analysis of the most promising moves, as incrementally evaluated with random playouts.

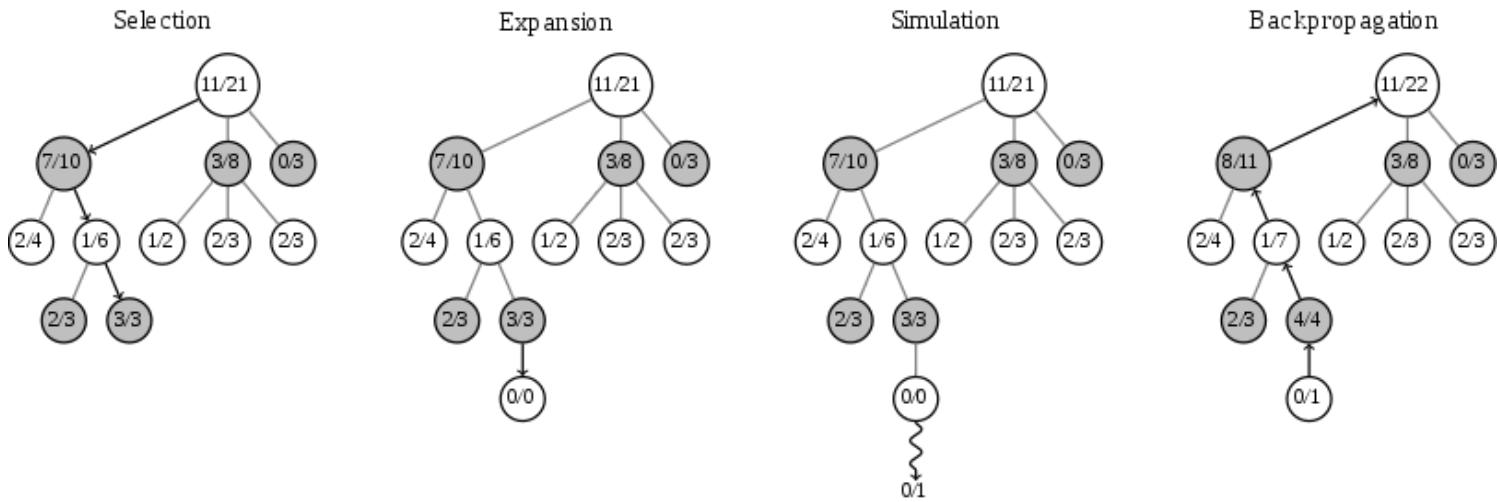
Each node n in the current search tree maintains two values:

- the number of wins $Q(n, p)$ of player p for all playouts that passed through n ;
- the number $N(n)$ of times n has been visited.

The algorithm searches the game tree as follows:

1. **Selection**: start from root, select successive child nodes down to a node n that is not fully expanded.
2. **Expansion**: unless n is a terminal state, create a new child node n' .
3. **Simulation**: play a random playout from n' .
4. **Backpropagation**: use the result of the playout to update information in the nodes on the path from n' to the root.

Repeat 1-4 for as long the time budget allows. Pick the best next direct move.



Exploration and exploitation

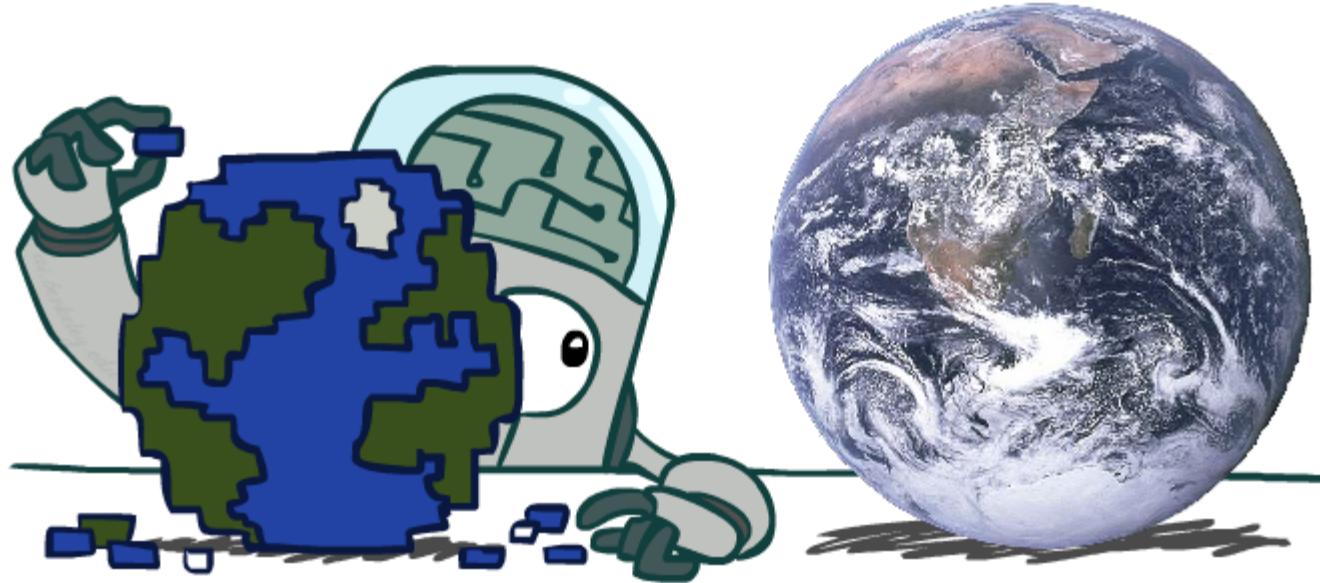
Given a limited budget of random playouts, the efficiency of MCTS critically depends on the choice of the nodes that are selected at step 1.

During the traversal of the branch in the selection step, the UCB1 policy picks the child node n' of n that maximizes

$$\frac{Q(n', p)}{N(n')} + c \sqrt{\frac{\log N(n)}{N(n')}}.$$

- The first term encourages the **exploitation** of higher-reward nodes.
- The second term encourages the **exploration** of less-visited nodes.
- The constant $c > 0$ controls the trade-off between exploitation and exploration.

Modeling assumptions

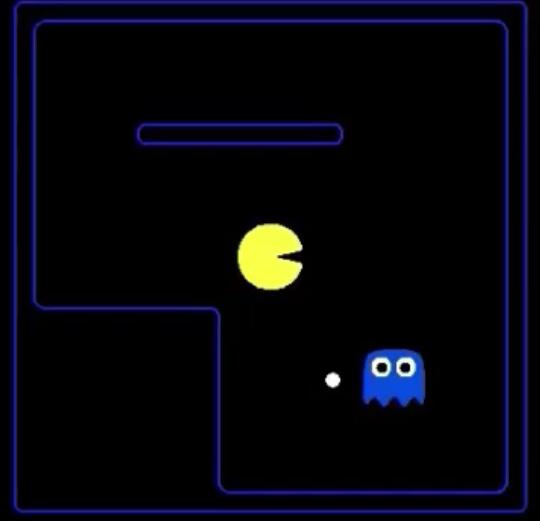


What if our assumptions are incorrect?

Setup

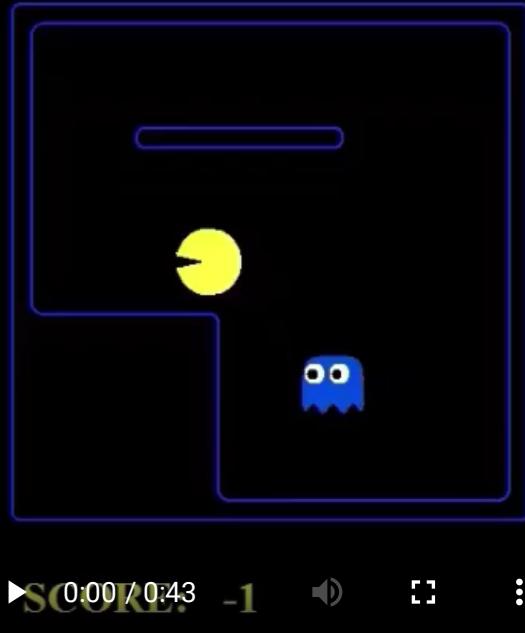
- P_1 : Pacman uses depth 4 search with an evaluation function that avoids trouble, while assuming that the ghost follows P_2 .
- P_2 : Ghost uses depth 2 search with an evaluation function that seeks Pacman, while assuming that Pacman follows P_1 .
- P_3 : Pacman uses depth 4 search with an evaluation function that avoids trouble, while assuming that the ghost follows P_4
- P_4 : Ghost makes random moves.





▶ SCORE: 000 / 0:09 0 🔍 ⏸ ⋮

Minimax Pacman (P_1) vs. Adversarial ghost (P_2)



Minimax Pacman (P_1) vs. Random ghost (P_4)



Expectiminimax Pacman (P_3) vs. Random ghost (P_4)



Expectiminimax Pacman (P_3) vs. Adversarial ghost (P_2)

State-of-the-art game programs

Checkers

1951

First computer player by Christopher Strachey.

1994

The computer program [Chinook](#) ends the 40-year-reign of human champion Marion Tinsley.

- Library of opening moves from grandmasters;
- A deep search algorithm;
- A good move evaluation function (based on a linear model);
- A database for all positions with eight pieces or fewer.

2007

Checkers is solved. A weak solution is computationally proven.

- The number of involved calculations was 10^{14} , over a period of 18 years.
- A draw is always guaranteed provided neither player makes a mistake.

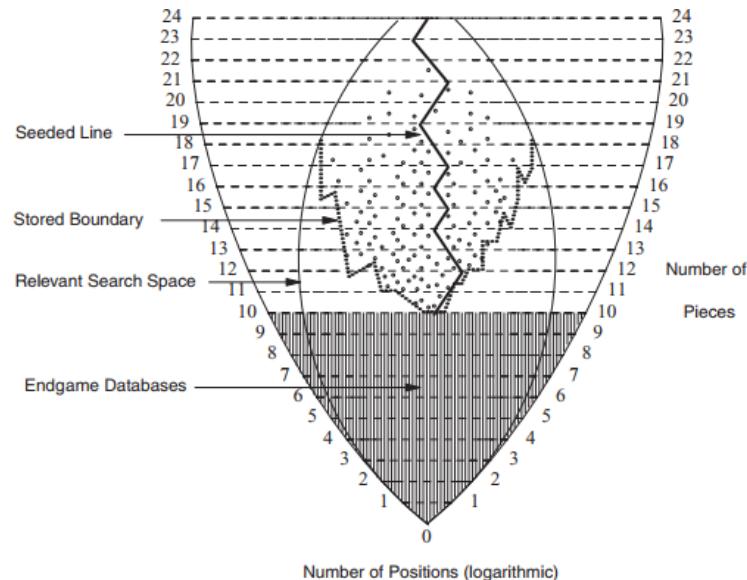


Fig. 2. Forward and backward search. The number of pieces on the board are plotted (vertically) versus the logarithm of the number of positions (Table 1). The shaded area shows the endgame database part of the proof—i.e., all positions with ≤ 10 pieces. The inner oval area shows that only a portion of the search space is relevant to the proof. Positions may be irrelevant because they are unreachable or are not required for the proof. The small open circles indicate positions with more than 10 pieces for which a value has been proven by a solver. The dotted line shows the boundary between the top of the proof tree that the manager sees (and stores on disk) and the parts that are computed by the solvers (and are not saved in order to reduce disk storage needs). The solid seeded line shows a “best” sequence of moves.

Chess

1997

- Deep Blue defeats human champion Gary Kasparov.
 - 200000000 position evaluations per second.
 - Very sophisticated evaluation function.
 - Undisclosed methods for extending some lines of search up to 40 plies.
- Modern programs (e.g., Stockfish or AlphaZero) are better, if less historic.
- Chess remains unsolved due to the complexity of the game.



Go

For long, Go was considered as the Holy Grail of AI due to the size of its game tree.

- On a 19x19, the number of legal positions is $\pm 2 \times 10^{170}$.
- This results in $\pm 10^{800}$ games, considering a length of 400 or less.



2010-2014

Using Monte Carlo tree search and machine learning, computer players reach low dan levels.

2015-2017

Google Deepmind invents AlphaGo.

- 2015: AlphaGo beat Fan Hui, the European Go Champion.
- 2016: AlphaGo beat Lee Sedol (4-1), a 9-dan grandmaster.
- 2017: AlphaGo beat Ke Jie, 1st world human player.

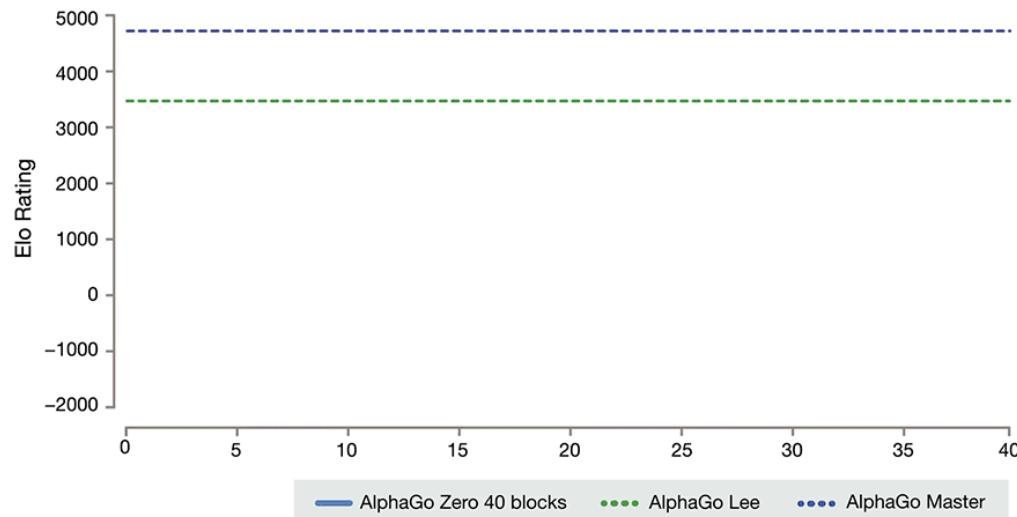
AlphaGo combines Monte Carlo tree search and deep learning with extensive training, both from human and computer play.



Press coverage for the victory of AlphaGo against Lee Sedol.

2017

AlphaGo Zero combines [Monte Carlo tree search](#) and [deep learning](#) with extensive training, with self-play only



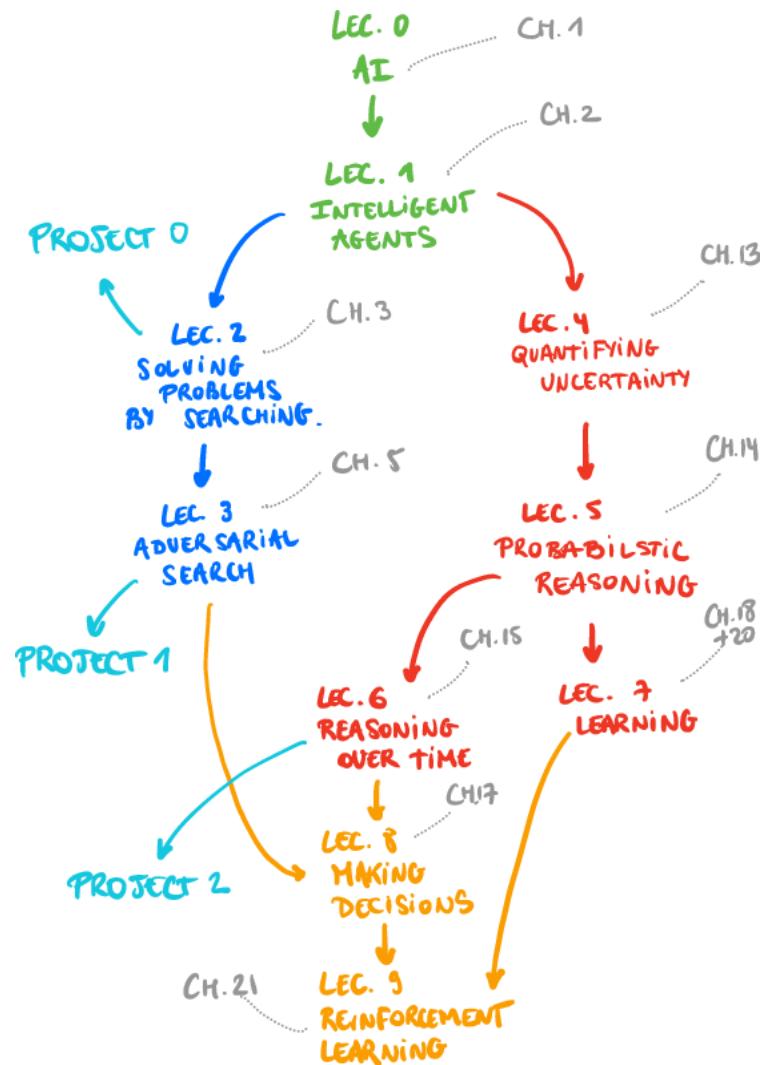
Summary

- Multi-player games are variants of search problems.
- The difficulty is to account for the fact that the opponent may act arbitrarily.
 - The optimal solution is a **strategy**, and not a fixed sequence of actions.
- **Minimax** is an optimal algorithm for deterministic, turn-taking, two-player zero-sum game with perfect information.
 - Due to practical time constraints, exploring the whole game tree is often **infeasible**.
 - Approximations can be achieved with heuristics, reducing computing times.
 - Minimax can be adapted to stochastic games.
 - Minimax can be adapted to games with more than 2 players.
- Optimal behavior is **relative** and depends on the assumptions we make about the world.

Introduction to Artificial Intelligence

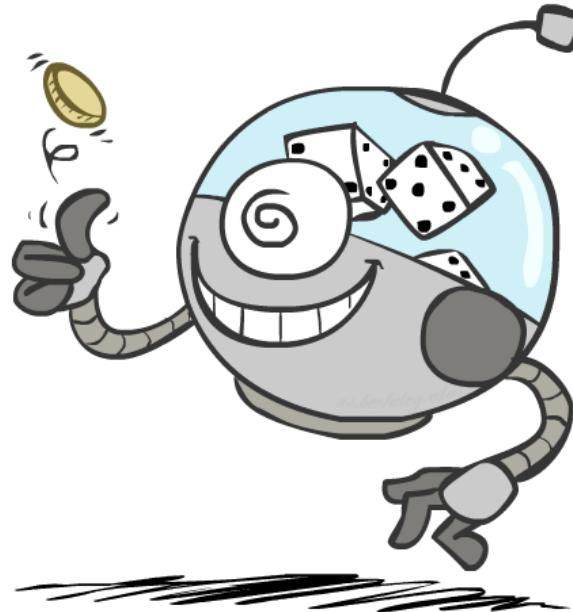
Lecture 4: Quantifying uncertainty

Prof. Gilles Louppe
g.louppe@uliege.be



Today

- Random variables
- Probability distributions
- Inference
- Independence
- The Bayes' rule



Do not overlook this lecture!

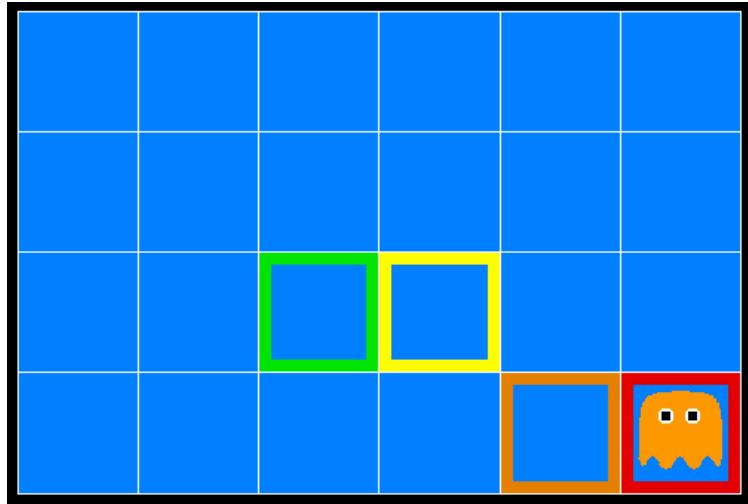
Quantifying uncertainty

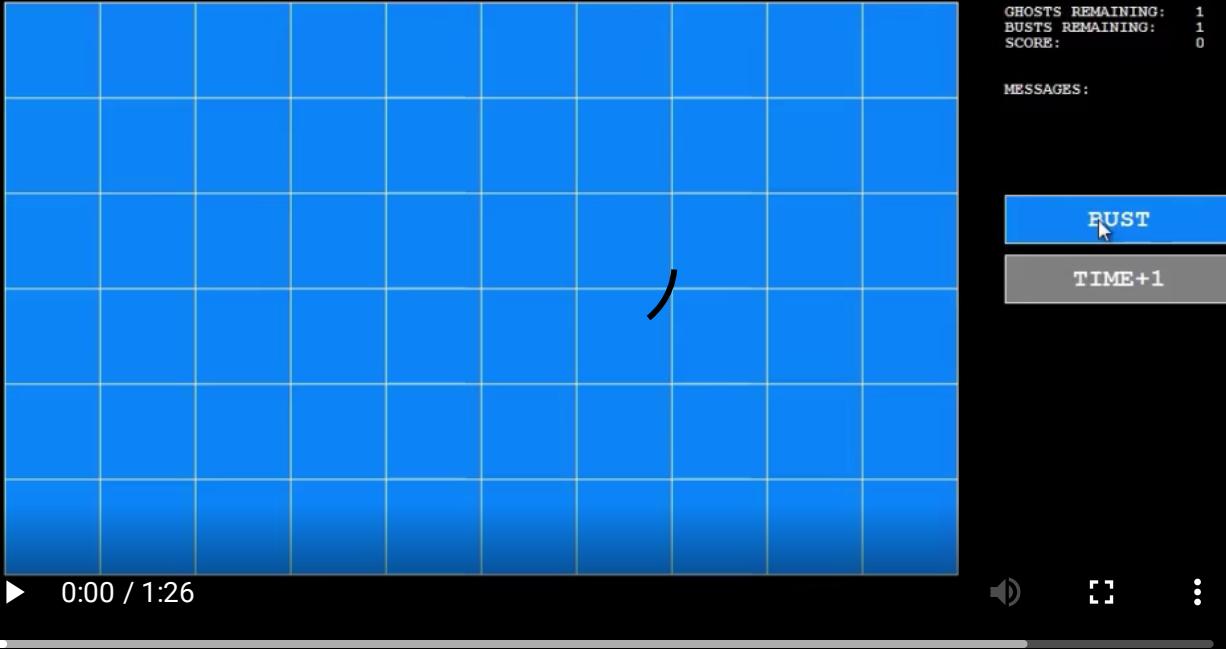
A ghost is **hidden** in the grid somewhere.

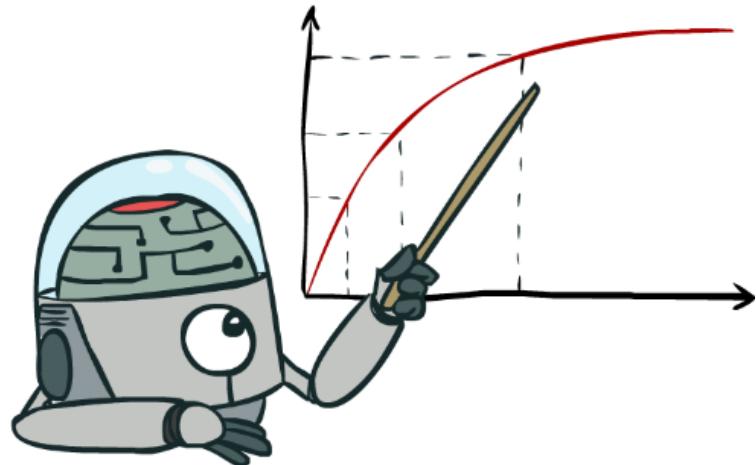
Sensor readings tell how close a square is to the ghost:

- On the ghost: red
- 1 or 2 away: orange
- 3 away: yellow
- 4+ away green

Sensors are **noisy**, but we know the probability values $P(\text{color}|\text{distance})$, for all colors and all distances.







Principle of maximum expected utility

An agent is rational if it chooses the action that yields the **highest expected utility**, averaged over all the possible outcomes of the action.

What does "expected" mean exactly?

Uncertainty

General setup:

- **Observed** variables or evidence: agent knows certain things about the state of the world (e.g., sensor readings).
- **Unobserved** variables: agent needs to reason about other aspects that are uncertain (e.g., where the ghost is).
- (Probabilistic) **model**: agent knows or believes something about how the observed variables relate to the unobserved variables.

Probabilistic reasoning provides a framework for managing our knowledge and beliefs.

Probabilistic assertions

Probabilistic assertions express the agent's inability to reach a definite decision regarding the truth of a proposition.

- Probability values **summarize** effects of
 - **ignorance** (theoretical, practical)
 - **laziness** (lack of time, resources)
- Probabilities relate propositions to one's own state of knowledge (or lack thereof).
 - e.g., $P(\text{ghost in cell } [3, 2]) = 0.02$

Frequentism vs. Bayesianism

What do probability values represent?

- The objectivist **frequentist** view is that probabilities are real aspects of the universe.
 - i.e., propensities of objects to behave in certain ways.
 - e.g., the fact that a fair coin comes up heads with probability **0.5** is a propensity of the coin itself.
- The subjectivist **Bayesian** view is that probabilities are a way of characterizing an agent's beliefs or uncertainty.
 - i.e., probabilities do not have external physical significance.
 - This is the interpretation of probabilities that we will use!

How shall we assign numerical values to beliefs?

Kolmogorov's axioms

Begin with a set Ω , the sample space.

$\omega \in \Omega$ is a sample point or possible world.

A probability space is a sample space equipped with a probability function, i.e. an assignment $P : \mathcal{P}(\Omega) \rightarrow \mathbb{R}$ such that:

- 1st axiom: $P(\omega) \in \mathbb{R}, 0 \leq P(\omega)$ for all $\omega \in \Omega$
- 2nd axiom: $P(\Omega) = 1$
- 3rd axiom: $P(\{\omega_1, \dots, \omega_n\}) = \sum_{i=1}^n P(\omega_i)$ for any set of samples

where $\mathcal{P}(\Omega)$ the power set of Ω .

Example

- Ω = the 6 possible rolls of a die.
- ω_i (for $i = 1, \dots, 6$) are the sample points, each corresponding to an outcome of the die.
- Assignment P for a fair die:

$$P(1) = P(2) = P(3) = P(4) = P(5) = P(6) = \frac{1}{6}$$

Random variables

- A **random variable** is a function $X : \Omega \rightarrow D_X$ from the sample space to some domain defining its outcomes.
 - e.g., $\text{Odd} : \Omega \rightarrow \{\text{true}, \text{false}\}$ such that $\text{Odd}(\omega) = (\omega \bmod 2 = 1)$.
- P induces a **probability distribution** for any random variable X .
 - $P(X = x_i) = \sum_{\{\omega : X(\omega) = x_i\}} P(\omega)$
 - e.g., $P(\text{Odd} = \text{true}) = P(1) + P(3) + P(5) = \frac{1}{2}$.
- An **event** E is a set of outcomes $\{(x_1, \dots, x_n), \dots\}$ of the variables X_1, \dots, X_n , such that

$$P(E) = \sum_{(x_1, \dots, x_n) \in E} P(X_1 = x_1, \dots, X_n = x_n).$$

Notations

- Random variables are written in upper roman letters: \mathbf{X} , \mathbf{Y} , etc.
- Realizations of a random variable are written in corresponding lower case letters. E.g., x_1 , x_2 , ..., x_n could be of outcomes of the random variable \mathbf{X} .
- The probability value of the realization x is written as $P(\mathbf{X} = x)$.
- When clear from context, this will be abbreviated as $P(x)$.
- The probability distribution of the (discrete) random variable \mathbf{X} is denoted as $\mathbf{P}(\mathbf{X})$. This corresponds e.g. to a vector of numbers, one for each of the probability values $P(\mathbf{X} = x_i)$ (and not to a single scalar value!).

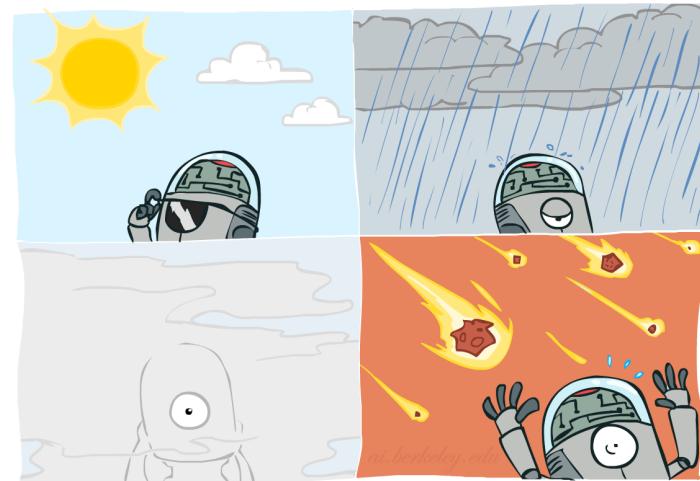
Probability distributions

For discrete variables, the **probability distribution** can be encoded by a discrete list of the probabilities of the outcomes, known as the **probability mass function**.

One can think of the probability distribution as a **table** that associates a probability value to each **outcome** of the variable.

P(W)

<i>W</i>	<i>P</i>
sun	0.6
rain	0.1
fog	0.3
meteor	0.0



Joint distributions

A **joint** probability distribution over a set of random variables X_1, \dots, X_n specifies the probability of each (combined) outcome:

$$P(X_1 = x_1, \dots, X_n = x_n) = \sum_{\{\omega : X_1(\omega) = x_1, \dots, X_n(\omega) = x_n\}} P(\omega)$$

$$\mathbf{P}(T, W)$$

T	W	P
hot	sun	0.4
hot	rain	0.1
cold	sun	0.2
cold	rain	0.3

Marginal distributions

The **marginal distribution** of a subset of a collection of random variables is the joint probability distribution of the variables contained in the subset.

$$\mathbf{P}(T, W)$$

T	W	P
hot	sun	0.4
hot	rain	0.1
cold	sun	0.2
cold	rain	0.3

$$\mathbf{P}(T)$$

T	P
hot	0.5
cold	0.5

$$\mathbf{P}(W)$$

W	P
sun	0.6
rain	0.4

$$P(t) = \sum_w P(t, w) \quad P(w) = \sum_t P(t, w)$$

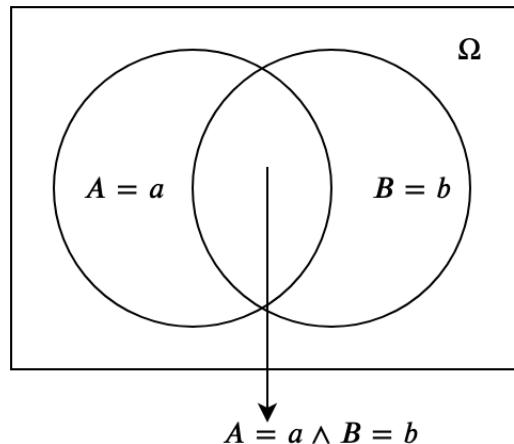
Intuitively, marginal distributions are sub-tables which eliminate variables.

Conditional distributions

The **conditional probability** of a realization a given the realization b is defined as the ratio of the probability of the joint realization a and b , and the probability of b :

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

Indeed, observing $B = b$ rules out all those possible worlds where $B \neq b$, leaving a set whose total probability is just $P(b)$. Within that set, the worlds for which $A = a$ satisfy $A = a \wedge B = b$ and constitute a fraction $P(a, b)/P(b)$.



Conditional distributions are probability distributions over some variables, given **fixed** values for others.

$$\mathbf{P}(T, W)$$

T	W	P
hot	sun	0.4
hot	rain	0.1
cold	sun	0.2
cold	rain	0.3

$$\mathbf{P}(W|T = \text{hot})$$

W	P
sun	0.8
rain	0.2

$$\mathbf{P}(W|T = \text{cold})$$

W	P
sun	0.4
rain	0.6

Probabilistic inference

Probabilistic **inference** is the problem of computing a desired probability from other known probabilities (e.g., conditional from joint).

- We generally compute conditional probabilities.
 - e.g., $P(\text{on time}|\text{no reported accidents}) = 0.9$
 - These represent the agent's **beliefs** given the evidence.
- Probabilities change with new evidence:
 - e.g., $P(\text{on time}|\text{no reported accidents, 5AM}) = 0.95$
 - e.g., $P(\text{on time}|\text{no reported accidents, rain}) = 0.8$
 - e.g., $P(\text{ghost in } [3, 2]|\text{red in } [3, 2]) = 0.99$
 - Observing new evidence causes beliefs to be updated.

General case

- Evidence variables: $E_1, \dots, E_k = e_1, \dots, e_k$
- Query variables: Q
- Hidden variables: H_1, \dots, H_r
- $(Q \cup E_1, \dots, E_k \cup H_1, \dots, H_r) =$ all variables X_1, \dots, X_n

Inference is the problem of computing $\mathbf{P}(Q|e_1, \dots, e_k)$.

Inference by enumeration

Start from the joint distribution $\mathbf{P}(Q, E_1, \dots, E_k, H_1, \dots, H_r)$.

1. Select the entries consistent with the evidence $E_1, \dots, E_k = e_1, \dots, e_k$.
2. Marginalize out the hidden variables to obtain the joint of the query and the evidence variables:

$$\mathbf{P}(Q, e_1, \dots, e_k) = \sum_{h_1, \dots, h_r} \mathbf{P}(Q, h_1, \dots, h_r, e_1, \dots, e_k).$$

3. Normalize:

$$Z = \sum_q P(q, e_1, \dots, e_k)$$
$$\mathbf{P}(Q|e_1, \dots, e_k) = \frac{1}{Z} \mathbf{P}(Q, e_1, \dots, e_k)$$

Example

- $\mathbf{P}(W)?$
- $\mathbf{P}(W|\text{winter})?$
- $\mathbf{P}(W|\text{winter, hot})?$

<i>S</i>	<i>T</i>	<i>W</i>	<i>P</i>
summer	hot	sun	0.3
summer	hot	rain	0.05
summer	cold	sun	0.1
summer	cold	rain	0.05
winter	hot	sun	0.1
winter	hot	rain	0.05
winter	cold	sun	0.15
winter	cold	rain	0.2

Complexity

- Inference by enumeration can be used to answer probabilistic queries for discrete variables (i.e., with a finite number of values).
- However, enumeration does not scale!
 - Assume a domain described by n variables taking at most d values.
 - Space complexity: $O(d^n)$
 - Time complexity: $O(d^n)$

Can we reduce the size of the representation of the joint distribution?

Product rule

$$P(a, b) = P(b)P(a|b)$$

Example

$\mathbf{P}(W)$

W	P
sun	0.8
rain	0.2

$\mathbf{P}(D|W)$

D	W	P
wet	sun	0.1
dry	sun	0.9
wet	rain	0.7
dry	rain	0.3

$\mathbf{P}(D, W)$

D	W	P
wet	sun	?
dry	sun	?
wet	rain	?
dry	rain	?

Chain rule

More generally, any joint distribution can always be written as an incremental product of conditional distributions:

$$P(x_1, x_2, x_3) = P(x_1)P(x_2|x_1)P(x_3|x_1, x_2)$$

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

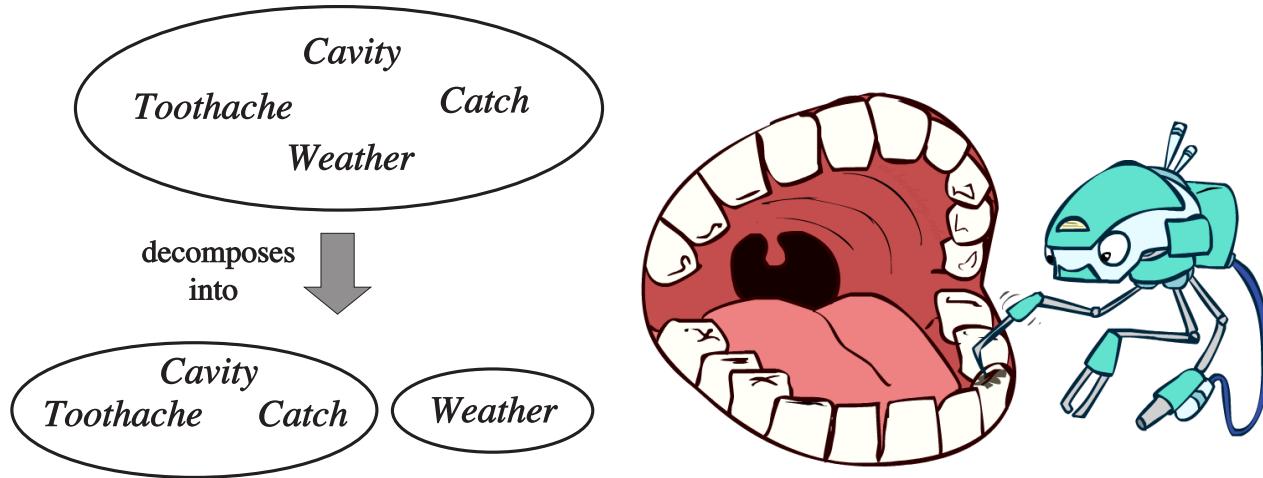
Independence

A and B are independent iff, for all $a \in D_A$ and $b \in D_B$,

- $P(a|b) = P(a)$, or
- $P(b|a) = P(b)$, or
- $P(a, b) = P(a)P(b)$

Independence is denoted as $A \perp B$.

Example 1



$$\begin{aligned} P(\text{toothache, catch, cavity, weather}) \\ = P(\text{toothache, catch, cavity})P(\text{weather}) \end{aligned}$$

The original 32-entry table reduces to one 8-entry and one 4-entry table (assuming 4 values for **Weather** and boolean values otherwise).

Example 2

For n independent coin flips, the joint distribution can be fully factored and represented as the product of n 1-entry tables.

- $2^n \rightarrow n$

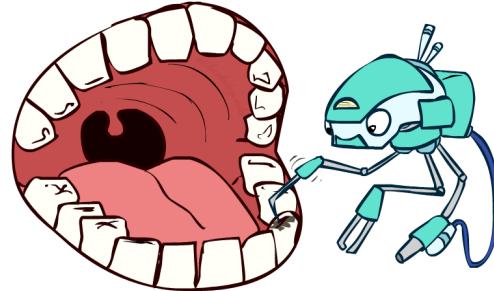
Conditional independence

A and B are conditionally independent given C iff, for all $a \in D_A, b \in D_B$ and $c \in D_C$,

- $P(a|b, c) = P(a|c)$, or
- $P(b|a, c) = P(b|c)$, or
- $P(a, b|c) = P(a|c)P(b|c)$

Conditional independence is denoted as $A \perp B|C$.

- Using the chain rule, the join distribution can be factored as a product of conditional distributions.
- Each conditional distribution may potentially be simplified by conditional independence.
- Conditional independence assertions allow probabilistic models to scale up.



Example 1

Assume three random variables **Toothache**, **Catch** and **Cavity**.

Catch is conditionally independent of **Toothache**, given **Cavity**. Therefore, we can write:

$$\begin{aligned} P(\text{toothache, catch, cavity}) &= P(\text{toothache}|\text{catch, cavity})P(\text{catch}|\text{cavity})P(\text{cavity}) \\ &= P(\text{toothache}|\text{cavity})P(\text{catch}|\text{cavity})P(\text{cavity}) \end{aligned}$$

In this case, the representation of the joint distribution reduces to $2 + 2 + 1$ independent numbers (instead of $2^n - 1$).

Example 2 (Naive Bayes)

More generally, from the product rule, we have

$$P(\text{cause}, \text{effect}_1, \dots, \text{effect}_n) = P(\text{effect}_1, \dots, \text{effect}_n | \text{cause})P(\text{cause})$$

Assuming pairwise conditional independence between the effects given the cause, it comes:

$$P(\text{cause}, \text{effect}_1, \dots, \text{effect}_n) = P(\text{cause}) \prod_i P(\text{effect}_i | \text{cause})$$

This probabilistic model is called a **naive Bayes** model.

- The complexity of this model is $O(n)$ instead of $O(2^n)$ without the conditional independence assumptions.
- Naive Bayes can work surprisingly well in practice, even when the assumptions are wrong.

Study the next slide. **Twice.**

The Bayes' rule

The product rule defines two ways to factor the joint distribution of two random variables.

$$P(a, b) = P(a|b)P(b) = P(b|a)P(a)$$

Therefore,

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



- $P(a)$ is the prior belief on a .
- $P(b)$ is the probability of the evidence b .
- $P(a|b)$ is the posterior belief on a , given the evidence b .
- $P(b|a)$ is the conditional probability of b given a . Depending on the context, this term is called the likelihood.

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



The Bayes' rule is the **foundation** of many AI systems.

Example 1: diagnostic probability from causal probability.

$$P(\text{cause}|\text{effect}) = \frac{P(\text{effect}|\text{cause})P(\text{cause})}{P(\text{effect})}$$

where

- $P(\text{effect}|\text{cause})$ quantifies the relationship in the **causal** direction.
- $P(\text{cause}|\text{effect})$ describes the **diagnostic** direction.

Let S =stiff neck and M =meningitis. Given $P(s|m) = 0.7$, $P(m) = 1/50000$, $P(s) = 0.01$, it comes

$$P(m|s) = \frac{P(s|m)P(m)}{P(s)} = \frac{0.7 \times 1/50000}{0.01} = 0.0014.$$

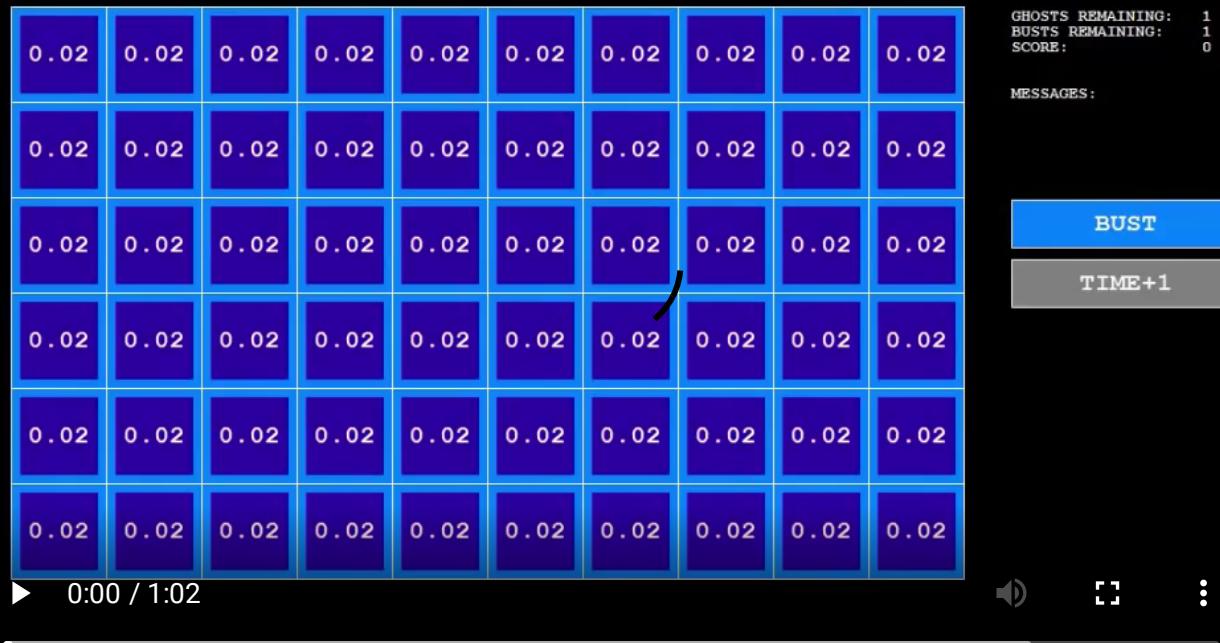
Example 2: Ghostbusters, revisited

- Let us assume a random variable G for the ghost location and a set of random variables $R_{i,j}$ for the individual readings.
- We start with a uniform prior distribution $\mathbf{P}(G)$ over ghost locations.
- We assume a sensor reading model $\mathbf{P}(R_{i,j}|G)$.
 - That is, we know what the sensors do.
 - $R_{i,j}$ = reading color measured at $[i, j]$
 - e.g., $P(R_{1,1} = \text{yellow}|G = [1, 1]) = 0.1$
 - Two readings are conditionally independent, given the ghost position.

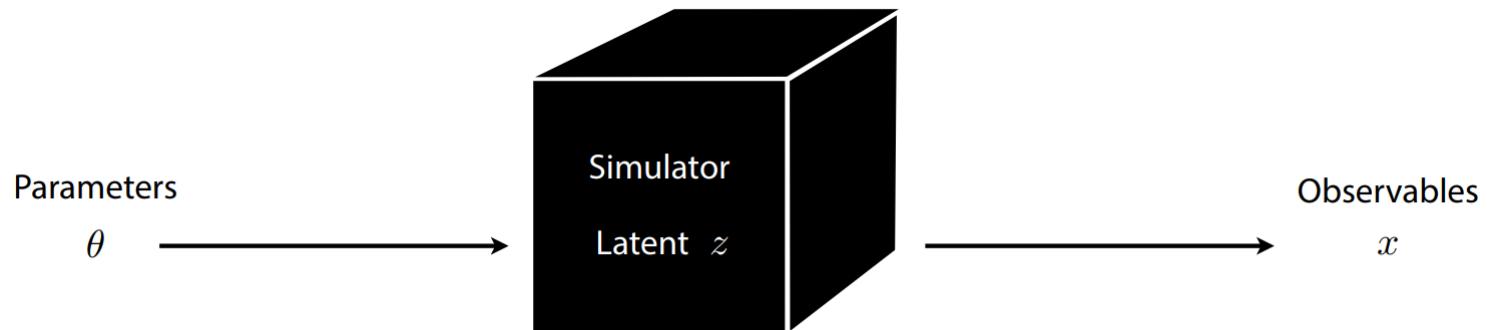
- We can calculate the posterior distribution $\mathbf{P}(G|R_{i,j})$ using Bayes' rule:

$$\mathbf{P}(G|R_{i,j}) = \frac{\mathbf{P}(R_{i,j}|G)\mathbf{P}(G)}{\mathbf{P}(R_{i,j})}.$$

- For the next reading $R_{i',j'}$, this posterior distribution becomes the prior distribution over ghost locations, which we update similarly.



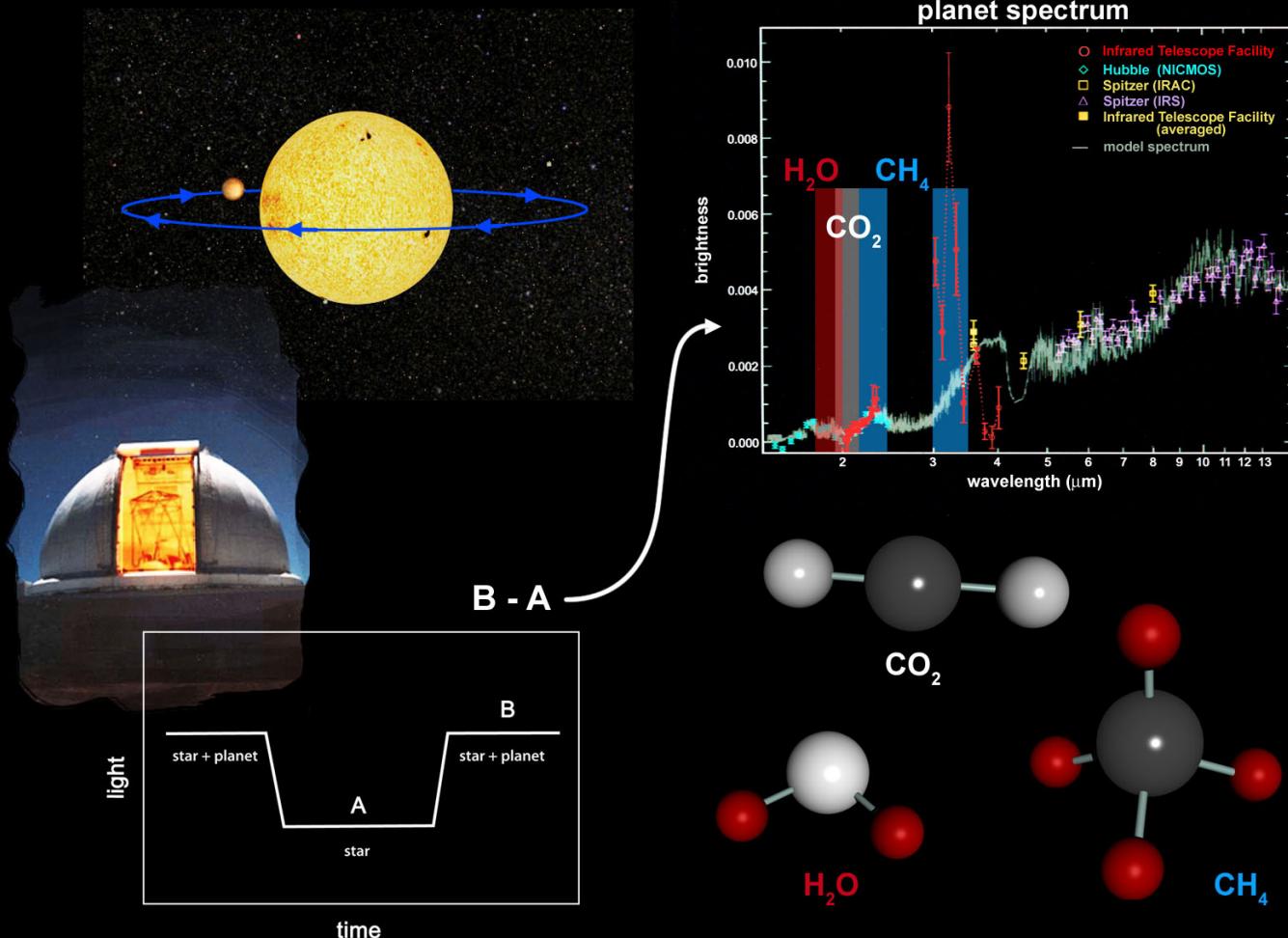
Example 3: AI for Science



Given some observation x and prior beliefs $p(\theta)$, science is about updating one's knowledge, which may be framed as computing

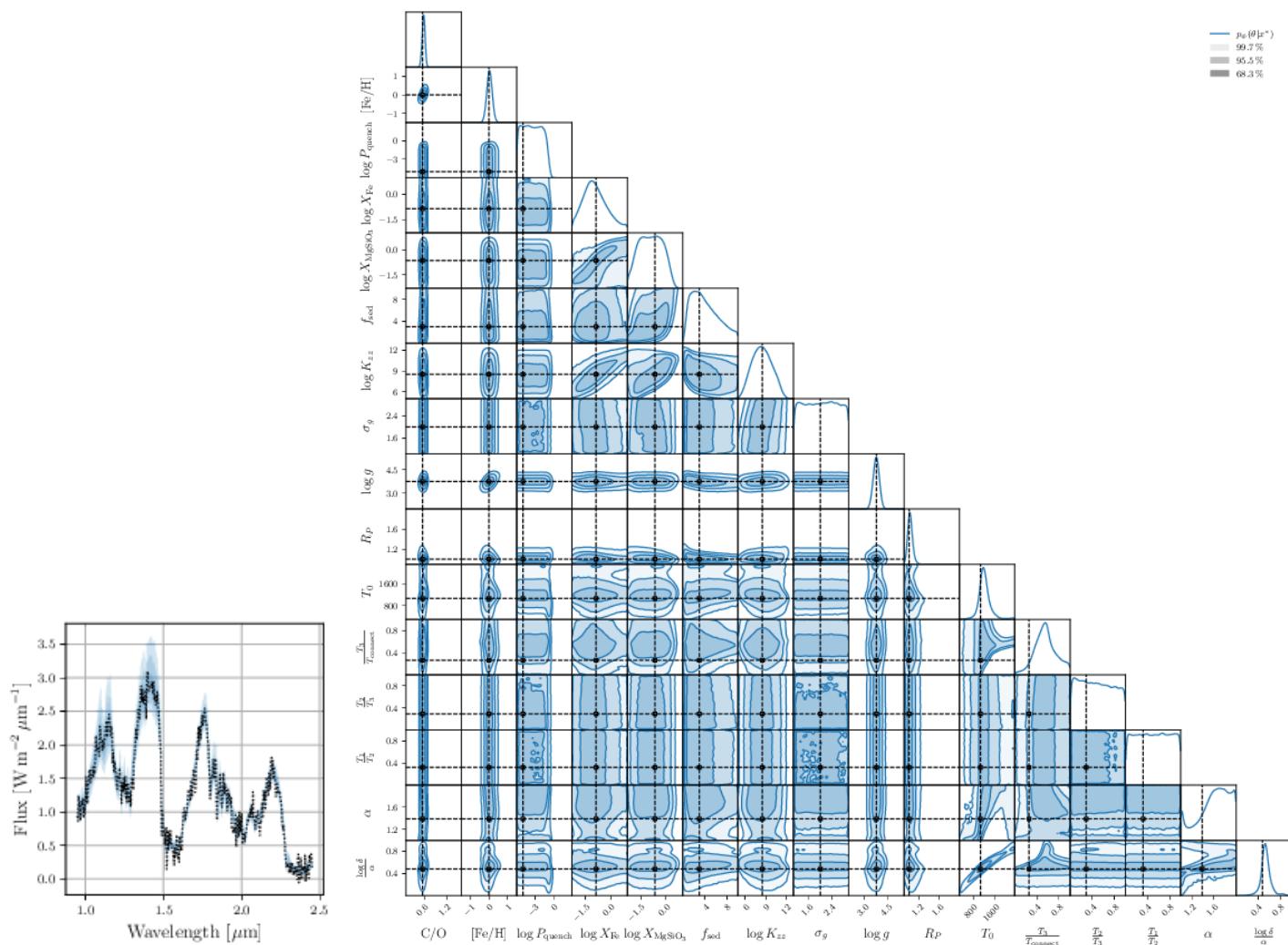
$$p(\theta|x) = \frac{p(x|\theta)p(\theta)}{p(x)}.$$

Exoplanet atmosphere characterization





$p_{\theta}(\theta | x^*)$
99.7%
95.5%
68.3%



Summary

- Uncertainty arises because of laziness and ignorance. It is **inescapable** in complex non-deterministic or partially observable environments.
- Probabilistic reasoning provides a framework for managing our knowledge and **beliefs**, with the Bayes' rule acting as the workhorse for inference.

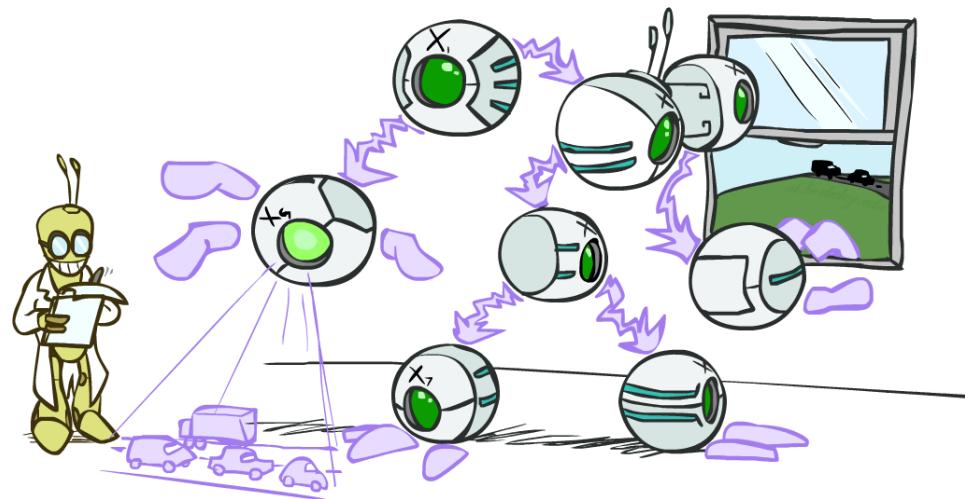
Introduction to Artificial Intelligence

Lecture 5: Probabilistic reasoning

Prof. Gilles Louppe
g.louppe@uliege.be

Today

- Bayesian networks
 - Semantics
 - Construction
 - Independence relations
- Inference
- Parameter learning



Representing uncertain knowledge

The explicit representation of the joint probability distribution grows exponentially with the number of variables.

Independence and conditional independence assumptions reduce the number of probabilities that need to be specified. They can be represented explicitly in the form of a Bayesian network.

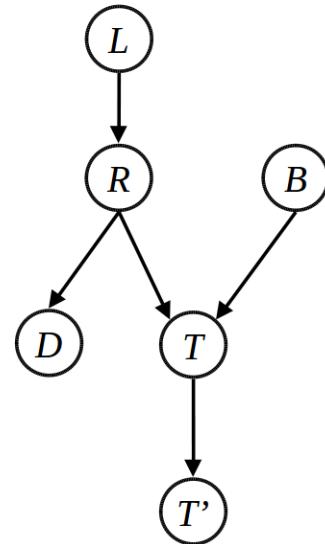
Bayesian networks

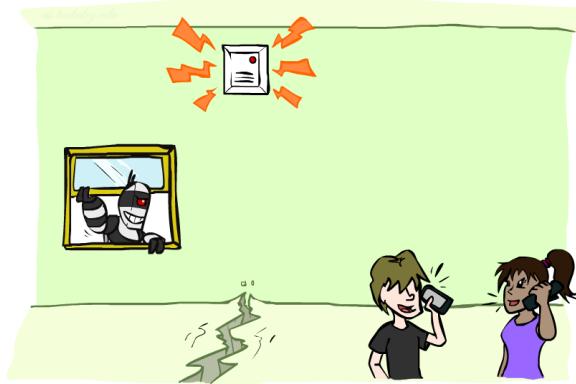
A Bayesian network is a **directed acyclic graph** where

- each **node** corresponds to a random variable;
 - observed or unobserved
 - discrete or continuous
- each **edge** is directed and indicates a direct probabilistic dependency between two variables;
- each node X_i is annotated with a **conditional probability distribution**

$$\mathbf{P}(X_i | \text{parents}(X_i))$$

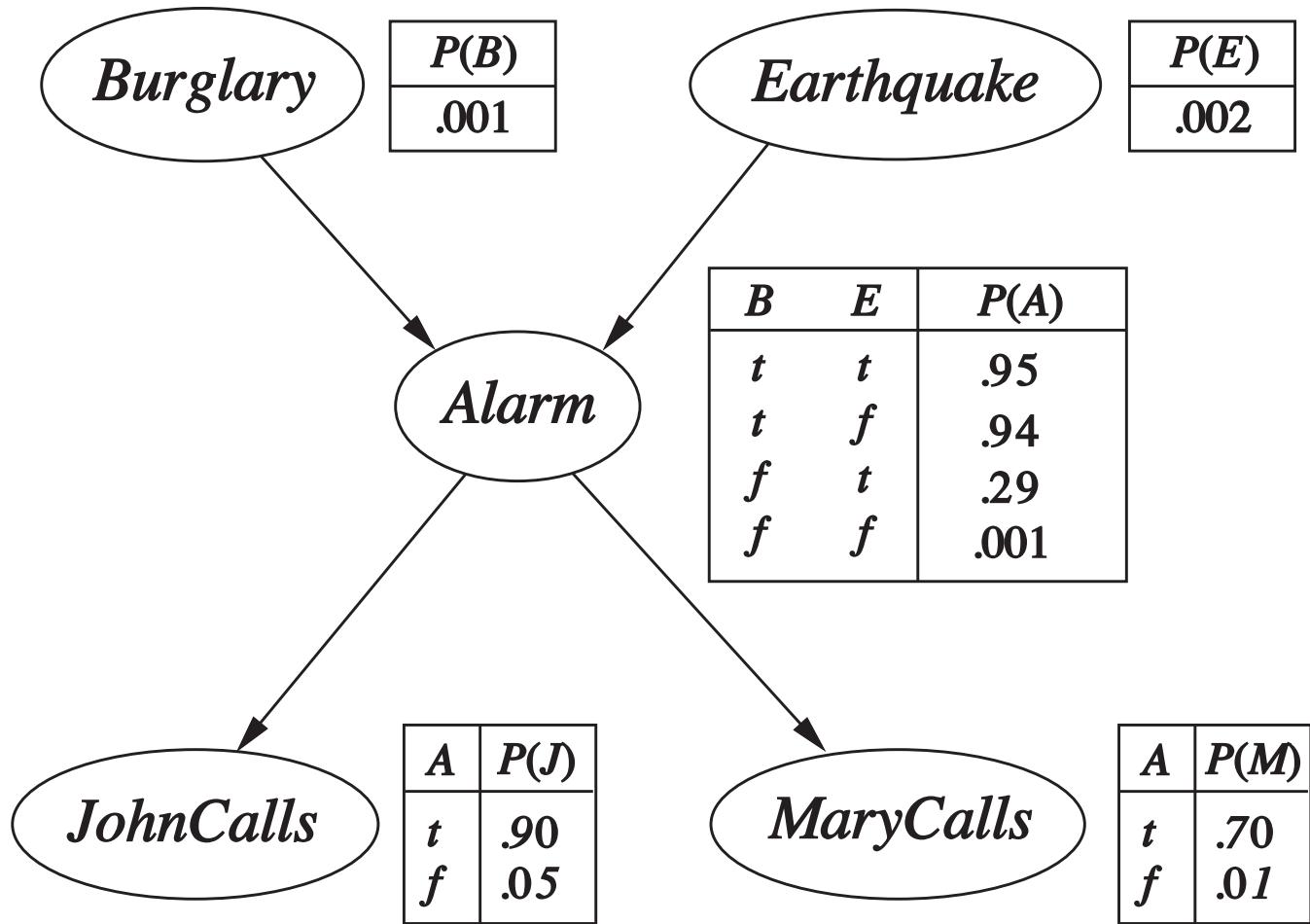
that defines the distribution of X_i given its parents in the network.





Example 1

- Variables: **Burglar**, **Earthquake**, **Alarm**, **JohnCalls**, **MaryCalls**.
- The network topology can be defined from domain knowledge:
 - A burglar can set the alarm off
 - An earthquake can set the alaram off
 - The alarm can cause Mary to call
 - The alarm can cause John to call



Semantics

A Bayesian network implicitly encodes the full joint distribution as a product of local distributions, that is

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

Proof:

- By the chain rule, $P(x_1, \dots, x_n) = \prod_{i=1}^n P(x_i | x_1, \dots, x_{i-1})$.
- Provided that we assume conditional independence of X_i with its predecessors in the ordering given the parents, and provided $\text{parents}(X_i) \subseteq \{X_1, \dots, X_{i-1}\}$, we have

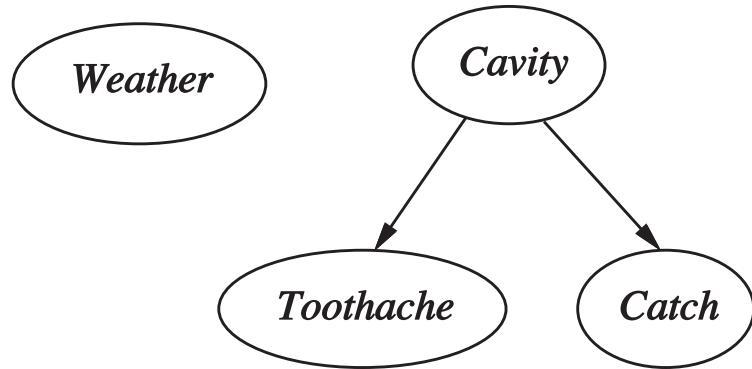
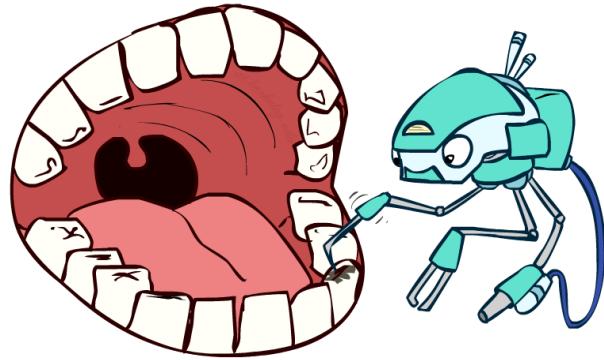
$$P(x_i | x_1, \dots, x_{i-1}) = P(x_i | \text{parents}(X_i)).$$

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

Example 1 (continued)

$$\begin{aligned}P(j, m, a, \neg b, \neg e) &= P(j|a)P(m|a)P(a|\neg b, \neg e)P(\neg b)P(\neg e) \\&= 0.9 \times 0.7 \times 0.001 \times 0.999 \times 0.998 \\&\approx 0.00063\end{aligned}$$

Example 2



The dentist's scenario can be modeled as a Bayesian network with four variables, as shown on the right.

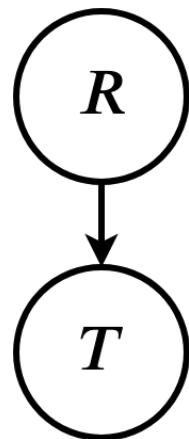
By construction, the topology of the network encodes conditional independence assertions. Each variable is independent of its non-descendants given its parents:

- **Weather** is independent of the other variables.
- **Toothache** and **Catch** are conditionally independent given **Cavity**.



Example 3

Edges may correspond to causal relations.



$$\mathbf{P}(R)$$

R	P
r	0.25
$\neg r$	0.75

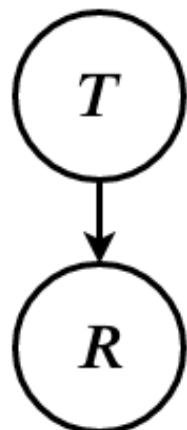
$$\mathbf{P}(T|R)$$

R	T	P
r	t	0.75
r	$\neg t$	0.25
$\neg r$	t	0.5
$\neg r$	$\neg t$	0.5



Example 3 (bis)

... but edges need not be causal!



$\mathbf{P}(T)$

T	P
t	9/16
$\neg t$	7/16

$\mathbf{P}(R|T)$

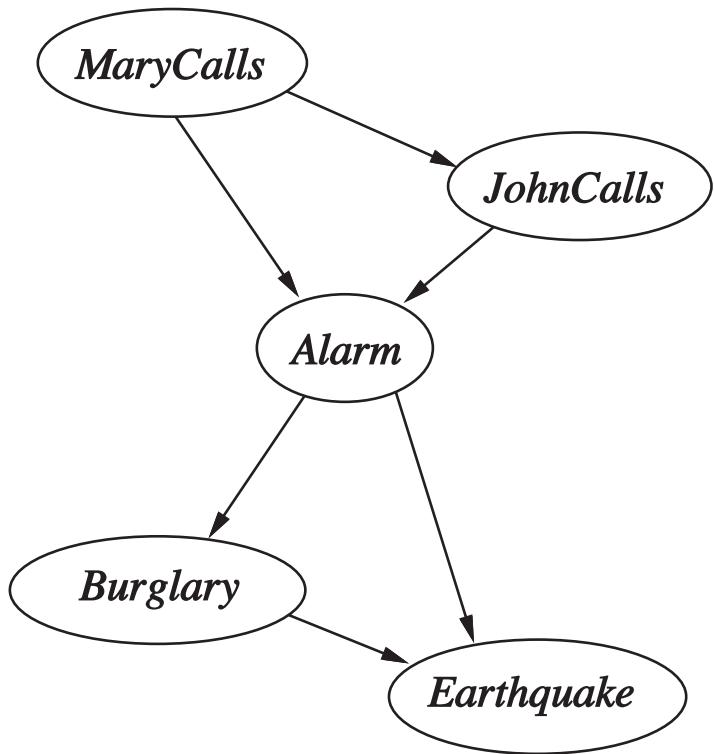
T	R	P
t	r	1/3
t	$\neg r$	2/3
$\neg t$	r	1/7
$\neg t$	$\neg r$	6/7

Construction

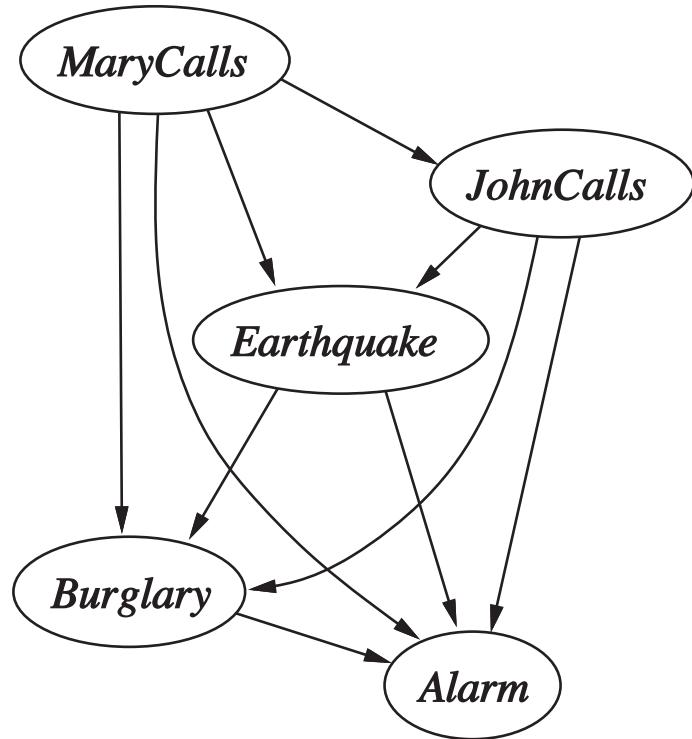
Bayesian networks can be constructed in any order, provided that the conditional independence assertions are respected.

Algorithm

1. Choose some **ordering** of the variables X_1, \dots, X_n .
2. For $i = 1$ to n :
 1. Add X_i to the network.
 2. Select a minimal set of parents from X_1, \dots, X_{i-1} such that $P(x_i|x_1, \dots, x_{i-1}) = P(x_i|\text{parents}(X_i))$.
 3. For each parent, insert a link from the parent to X_i .
 4. Write down the CPT.



(a)

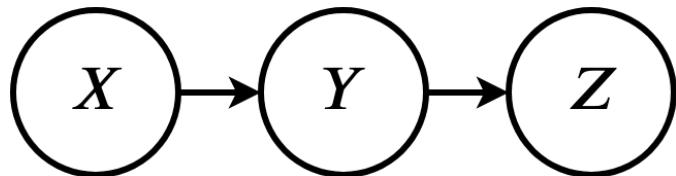


(b)

Do these networks represent the same distribution? Are they as compact?

Independence relations

Since the topology of a Bayesian network encodes conditional independence assertions, it can be used to answer questions about the independence of variables given some evidence.



Example: Are X and Z necessarily independent?

Cascades

Is X independent of Z ? No.

Counter-example:

- Low pressure causes rain causes traffic, high pressure causes no rain causes no traffic.
- In numbers:
 - $P(y|x) = 1$,
 - $P(z|y) = 1$,
 - $P(\neg y|\neg x) = 1$,
 - $P(\neg z|\neg y) = 1$



X : low pressure, Y : rain, Z : traffic.

$$P(x, y, z) = P(x)P(y|x)P(z|y)$$

Is X independent of Z , given Y ? Yes.

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

We say that the evidence along the cascade **blocks** the influence.



X : low pressure, Y : rain, Z : traffic.

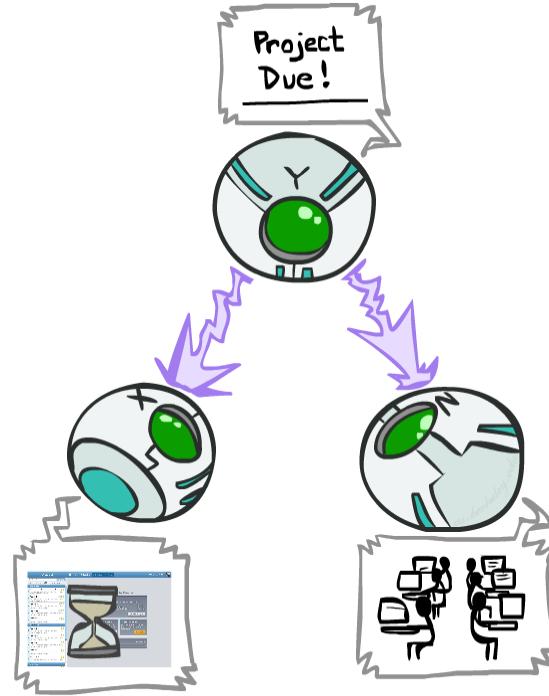
$$P(x,y,z) = P(x)P(y|x)P(z|y)$$

Common parent

Is X independent of Z ? No.

Counter-example:

- Project due causes both forums busy and lab full.
- In numbers:
 - $P(x|y) = 1$,
 - $P(\neg x|\neg y) = 1$,
 - $P(z|y) = 1$,
 - $P(\neg z|\neg y) = 1$



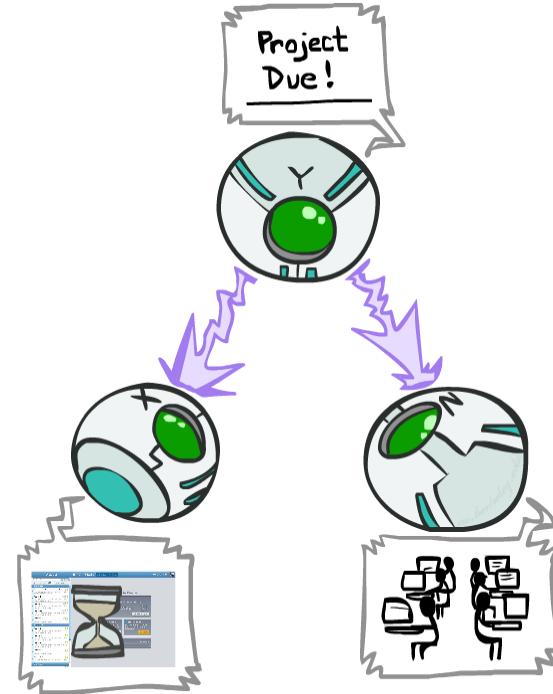
X : forum busy, Y : project due, Z : lab full.

$$P(x, y, z) = P(y)P(x|y)P(z|y)$$

Is X independent of Z , given Y ? Yes

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

Observing the parent blocks the influence between the children.



X : forum busy, Y : project due, Z : lab full.

$$P(x,y,z) = P(y)P(x|y)P(z|y)$$

v-structures

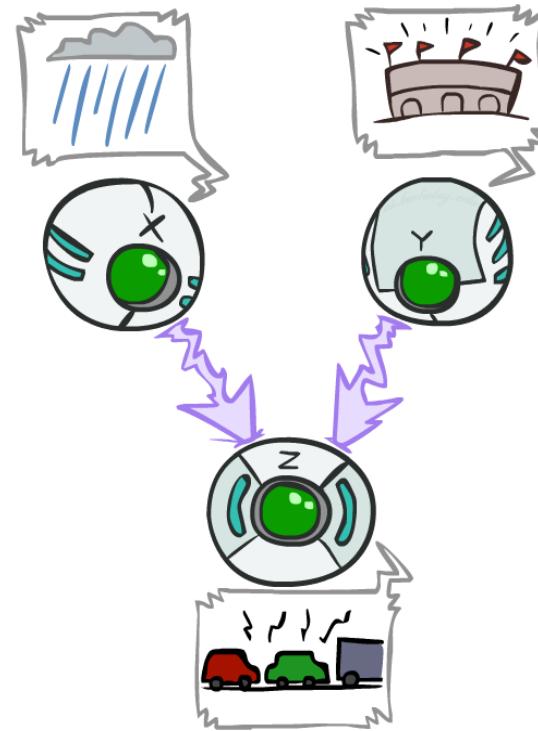
Are X and Y independent? Yes.

- The ballgame and the rain cause traffic, but they are not correlated.
- (Prove it!)

Are X and Y independent given Z ?

No!

- Seeing traffic puts the rain and the ballgame in competition as explanation.
- This is **backwards** from the previous cases. Observing a child node **activates** influence between parents.



X : rain, Y : ballgame, Z : traffic.

$$P(x, y, z) = P(x)P(y)P(z|x, y)$$

d-separation

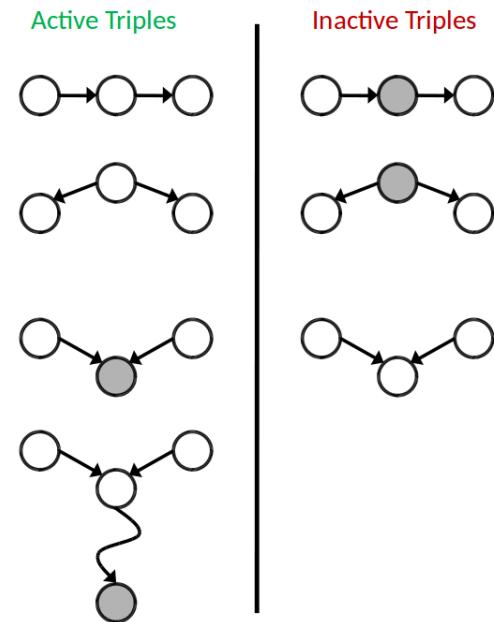
Let us assume a complete Bayesian network. Are X_i and X_j conditionally independent given evidence $Z_1 = z_1, \dots, Z_m = z_m$?

Consider all (undirected) paths from X_i to X_j :

- If one or more active path, then independence is not guaranteed.
- Otherwise (i.e., all paths are inactive), then independence is guaranteed.

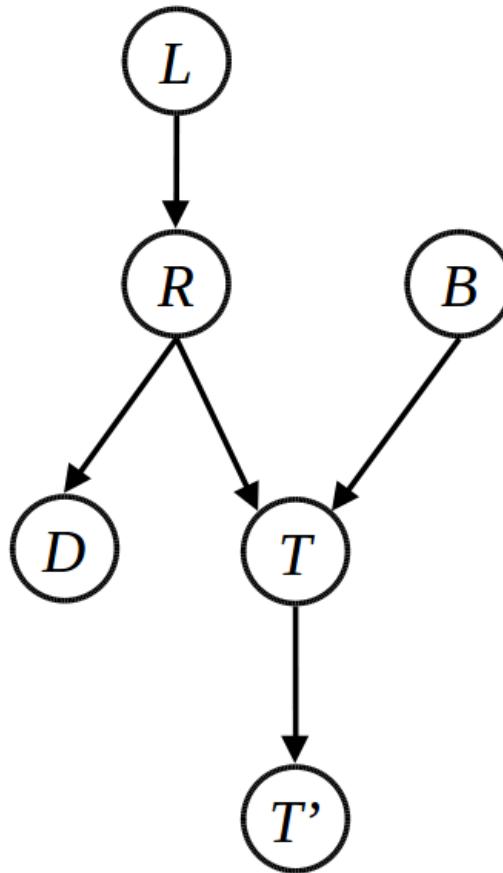
A path is **active** if each triple along the path is active:

- Cascade $A \rightarrow B \rightarrow C$ where B is unobserved (either direction).
- Common parent $A \leftarrow B \rightarrow C$ where B is unobserved.
- v-structure $A \rightarrow B \leftarrow C$ where B or one of its descendants is observed.



Example

- $L \perp T' | T?$
- $L \perp B?$
- $L \perp B | T?$
- $L \perp B | T'?$
- $L \perp B | T, R?$



Inference

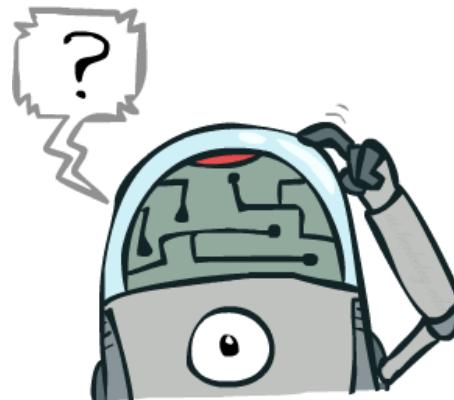
Inference is concerned with the problem **computing a marginal and/or a conditional probability distribution** from a joint probability distribution:

Simple queries: $\mathbf{P}(X_i|e)$

Conjunctive queries: $\mathbf{P}(X_i, X_j|e) = \mathbf{P}(X_i|e)\mathbf{P}(X_j|X_i, e)$

Most likely explanation: $\arg \max_q P(q|e)$

Optimal decisions: $\arg \max_a \mathbb{E}_{p(s'|s,a)} [V(s')]$



Inference by enumeration

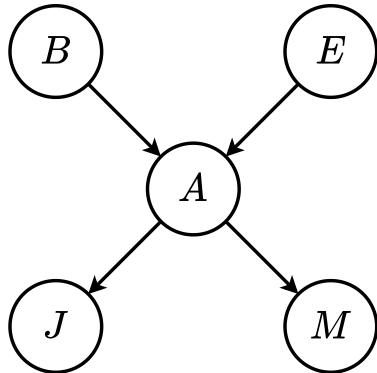
Start from the joint distribution $\mathbf{P}(Q, E_1, \dots, E_k, H_1, \dots, H_r)$.

1. Select the entries consistent with the evidence $E_1, \dots, E_k = e_1, \dots, e_k$.
2. Marginalize out the hidden variables to obtain the joint of the query and the evidence variables:

$$\mathbf{P}(Q, e_1, \dots, e_k) = \sum_{h_1, \dots, h_r} \mathbf{P}(Q, h_1, \dots, h_r, e_1, \dots, e_k).$$

3. Normalize:

$$Z = \sum_q P(q, e_1, \dots, e_k)$$
$$\mathbf{P}(Q|e_1, \dots, e_k) = \frac{1}{Z} \mathbf{P}(Q, e_1, \dots, e_k)$$

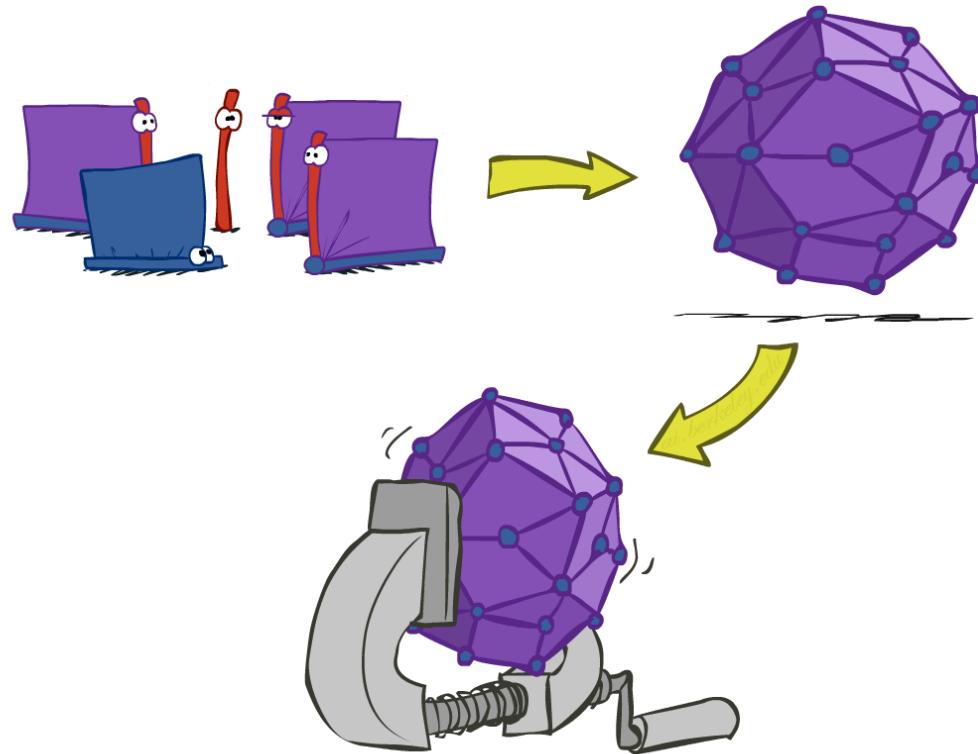


Consider the alarm network and the query $\mathbf{P}(B|j, m)$. We have

$$\begin{aligned}\mathbf{P}(B|j, m) &= \frac{1}{Z} \sum_e \sum_a \mathbf{P}(B, j, m, e, a) \\ &\propto \sum_e \sum_a \mathbf{P}(B, j, m, e, a).\end{aligned}$$

Using the Bayesian network, the full joint entries can be rewritten as the product of CPT entries

$$\mathbf{P}(B|j, m) \propto \sum_e \sum_a \mathbf{P}(B)P(e)\mathbf{P}(a|B, e)P(j|a)P(m|a).$$



Inference by enumeration is slow because the whole joint distribution is joined up before summing out the hidden variables.

Factors that do not depend on the variables in the summations can be factored out, which means that marginalization does not necessarily have to be done at the end, hence saving some computations.

For the alarm network, we have

$$\begin{aligned}\mathbf{P}(B|j, m) &\propto \sum_e \sum_a \mathbf{P}(B) P(e) \mathbf{P}(a|B, e) P(j|a) P(m|a) \\ &= \mathbf{P}(B) \sum_e P(e) \sum_a \mathbf{P}(a|B, e) P(j|a) P(m|a).\end{aligned}$$

function ENUMERATION-ASK(X, \mathbf{e}, bn) **returns** a distribution over X

inputs: X , the query variable
 \mathbf{e} , observed values for variables \mathbf{E}
 bn , a Bayes net with variables $\{X\} \cup \mathbf{E} \cup \mathbf{Y}$ /* $\mathbf{Y} = \text{hidden variables}$ */

$\mathbf{Q}(X) \leftarrow$ a distribution over X , initially empty

for each value x_i of X **do**

$\mathbf{Q}(x_i) \leftarrow$ ENUMERATE-ALL($bn.\text{VARS}, \mathbf{e}_{x_i}$)
 where \mathbf{e}_{x_i} is \mathbf{e} extended with $X = x_i$

return NORMALIZE($\mathbf{Q}(X)$)

function ENUMERATE-ALL($vars, \mathbf{e}$) **returns** a real number

if EMPTY?($vars$) **then return** 1.0

$Y \leftarrow \text{FIRST}(vars)$

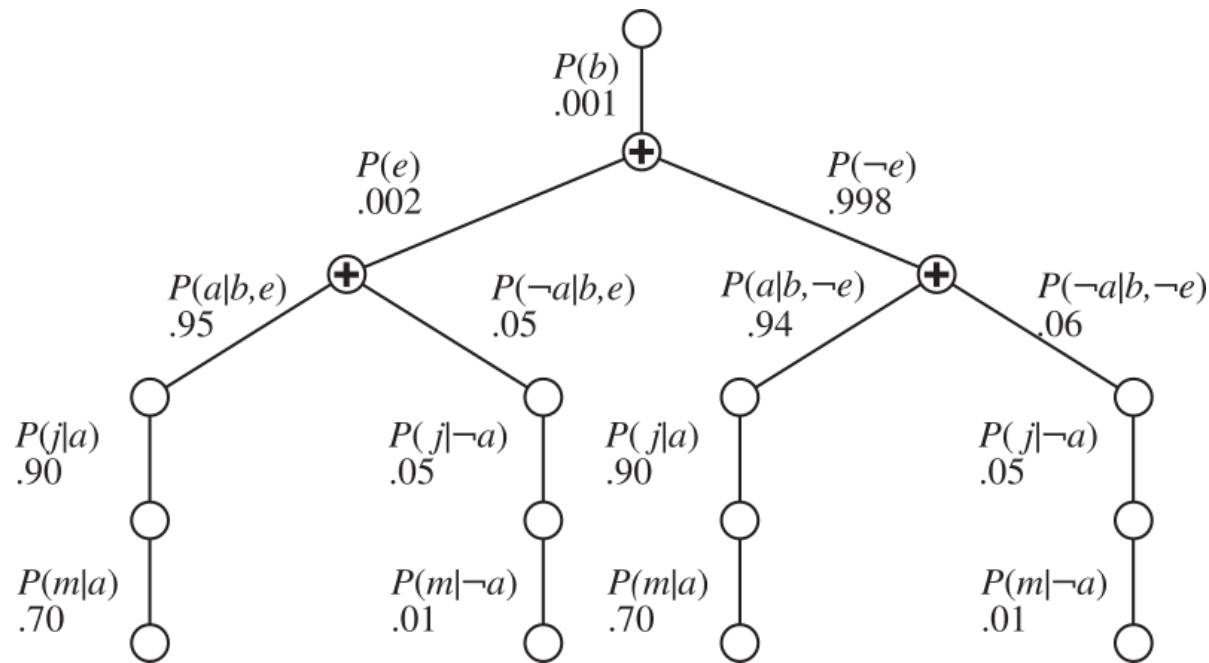
if Y has value y in \mathbf{e}

then return $P(y | parents(Y)) \times$ ENUMERATE-ALL(REST($vars$), \mathbf{e})

else return $\sum_y P(y | parents(Y)) \times$ ENUMERATE-ALL(REST($vars$), \mathbf{e}_y)
 where \mathbf{e}_y is \mathbf{e} extended with $Y = y$

Same complexity as DFS: $O(n)$ in space, $O(d^n)$ in time.

Evaluation tree for $P(b|j, m)$



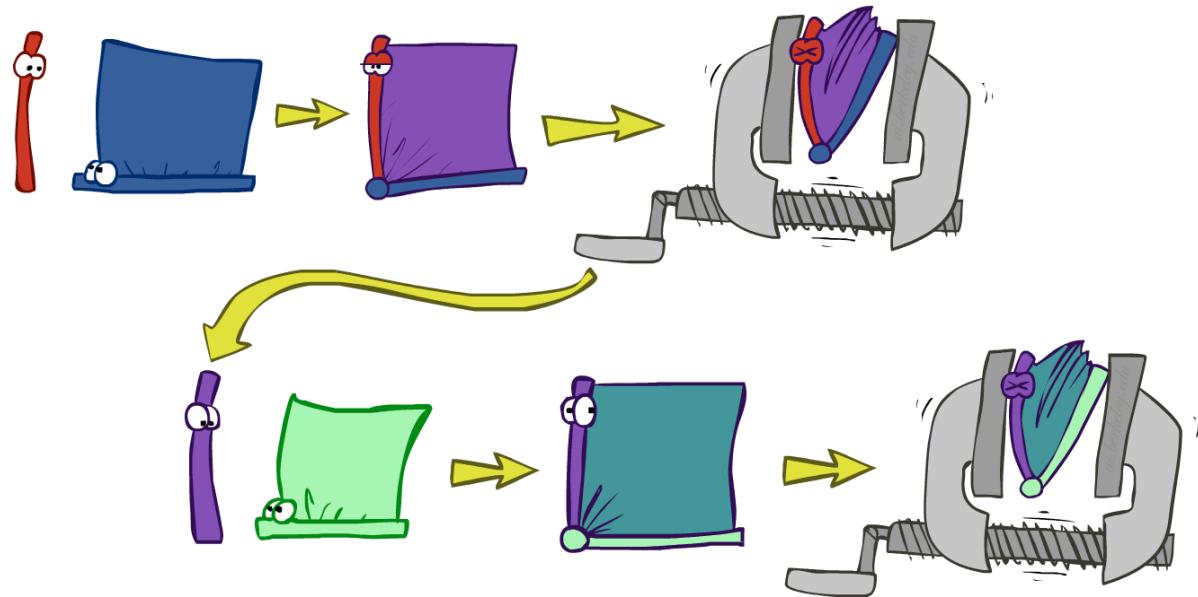
Despite the factoring, inference by enumeration is still **inefficient**. There are repeated computations!

- e.g., $P(j|a)P(m|a)$ is computed twice, once for e and once for $\neg e$.
- These can be avoided by storing **intermediate results**.

Inference by variable elimination

The **Variable Elimination** algorithm carries out summations right-to-left and stores intermediate factors to avoid recomputations. The algorithm interleaves:

- Joining sub-tables
- Eliminating hidden variables



Variable Elimination

Query: $\mathbf{P}(Q|e_1, \dots, e_k)$.

1. Start with the initial factors (the local CPTs, instantiated by the evidence).
2. While there are still hidden variables:
 1. Pick a hidden variable H
 2. Join all factors mentioning H
 3. Eliminate H
3. Join all remaining factors
4. Normalize

Factors

- Each factor \mathbf{f}_i is a multi-dimensional array indexed by the values of its argument variables. E.g.:

$$\mathbf{f}_4 = \mathbf{f}_4(A) = \begin{pmatrix} P(j|a) \\ P(j|\neg a) \end{pmatrix} = \begin{pmatrix} 0.90 \\ 0.05 \end{pmatrix}$$

$$\mathbf{f}_4(a) = 0.90$$

$$\mathbf{f}_4(\neg a) = 0.5$$

- Factors are initialized with the CPTs annotating the nodes of the Bayesian network, conditioned on the evidence.

Join

The pointwise product \times , or join, of two factors \mathbf{f}_1 and \mathbf{f}_2 yields a new factor \mathbf{f}_3 .

- Exactly like a database join!
- The variables of \mathbf{f}_3 are the union of the variables in \mathbf{f}_1 and \mathbf{f}_2 .
- The elements of \mathbf{f}_3 are given by the product of the corresponding elements in \mathbf{f}_1 and \mathbf{f}_2 .

A	B	$\mathbf{f}_1(A, B)$	B	C	$\mathbf{f}_2(B, C)$	A	B	C	$\mathbf{f}_3(A, B, C)$
T	T	.3	T	T	.2	T	T	T	.3 \times .2 = .06
T	F	.7	T	F	.8	T	T	F	.3 \times .8 = .24
F	T	.9	F	T	.6	T	F	T	.7 \times .6 = .42
F	F	.1	F	F	.4	T	F	F	.7 \times .4 = .28
						F	T	T	.9 \times .2 = .18
						F	T	F	.9 \times .8 = .72
						F	F	T	.1 \times .6 = .06
						F	F	F	.1 \times .4 = .04

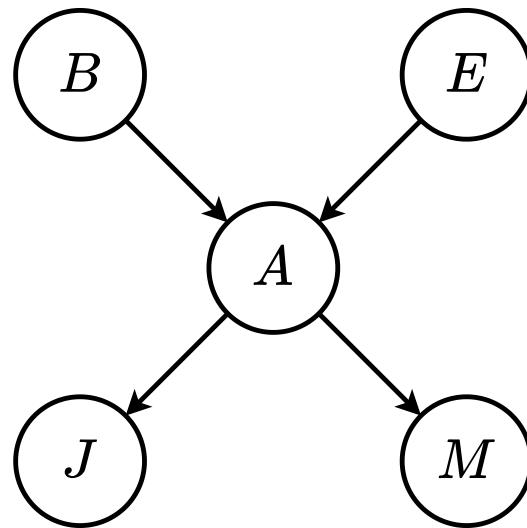
Figure 14.10 Illustrating pointwise multiplication: $\mathbf{f}_1(A, B) \times \mathbf{f}_2(B, C) = \mathbf{f}_3(A, B, C)$.

Elimination

Summing out, or eliminating, a variable from a factor is done by adding up the sub-arrays formed by fixing the variable to each of its values in turn.

For example, to sum out A from $\mathbf{f}_3(A, B, C)$, we write:

$$\begin{aligned}\mathbf{f}(B, C) &= \sum_a \mathbf{f}_3(a, B, C) = \mathbf{f}_3(a, B, C) + \mathbf{f}_3(\neg a, B, C) \\ &= \begin{pmatrix} 0.06 & 0.24 \\ 0.42 & 0.28 \end{pmatrix} + \begin{pmatrix} 0.18 & 0.72 \\ 0.06 & 0.04 \end{pmatrix} = \begin{pmatrix} 0.24 & 0.96 \\ 0.48 & 0.32 \end{pmatrix}\end{aligned}$$



Run the variable elimination algorithm for the query $\mathbf{P}(B|j, m)$.

Relevance

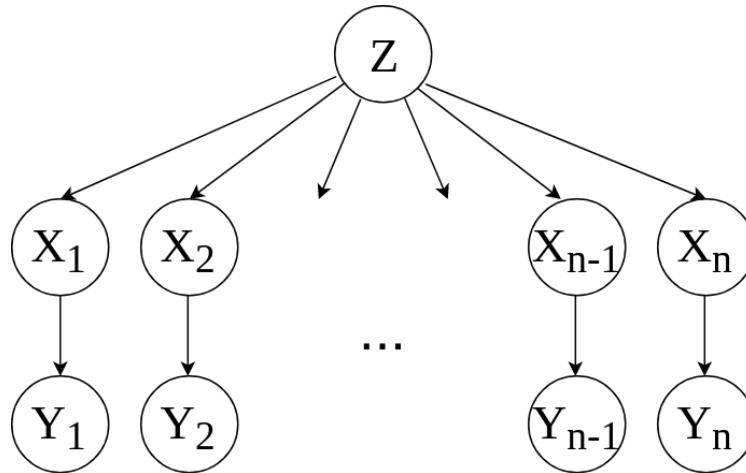
Consider the query $\mathbf{P}(J|b)$:

$$\mathbf{P}(J|b) \propto P(b) \sum_e P(e) \sum_a P(a|b, e) \mathbf{P}(J|a) \sum_m P(m|a)$$

- $\sum_m P(m|a) = 1$, therefore M is **irrelevant** for the query.
- In other words, $\mathbf{P}(J|b)$ remains unchanged if we remove M from the network.

Theorem. H is irrelevant for $\mathbf{P}(Q|e)$ unless $H \in \text{ancestors}(\{Q\} \cup E)$.

Complexity



Consider the query $\mathbf{P}(X_n | y_1, \dots, y_n)$.

Work through the two elimination orderings:

- Z, X_1, \dots, X_{n-1}
- X_1, \dots, X_{n-1}, Z

What is the size of the maximum factor generated for each of the orderings?

- Answer: 2^{n+1} vs. 2^2 (assuming boolean values)

The computational and space complexity of variable elimination is determined by the largest factor.

- The elimination [ordering](#) can greatly affect the size of the largest factor.
- The optimal ordering is **NP-hard** to find. There is no known polynomial-time algorithm to find it.

Approximate inference

Exact inference is **intractable** for most probabilistic models of practical interest.
(e.g., involving many variables, continuous and discrete, undirected cycles, etc).

We must resort to **approximate** inference algorithms:

- Sampling methods: produce answers by repeatedly generating random numbers from a distribution of interest.
- Variational methods: formulate inference as an optimization problem.
- Belief propagation methods: formulate inference as a message-passing algorithm.
- Machine learning methods: learn an approximation of the target distribution from training examples.

Parameter learning

When modeling a domain, we can choose a probabilistic model specified as a Bayesian network. However, specifying the individual probability values is often difficult.

A workaround is to use a **parameterized** family $\mathbf{P}(X|\theta)$ (sometimes also noted $\mathbf{P}_\theta(X)$) of models, and **estimate** the parameters θ from data.

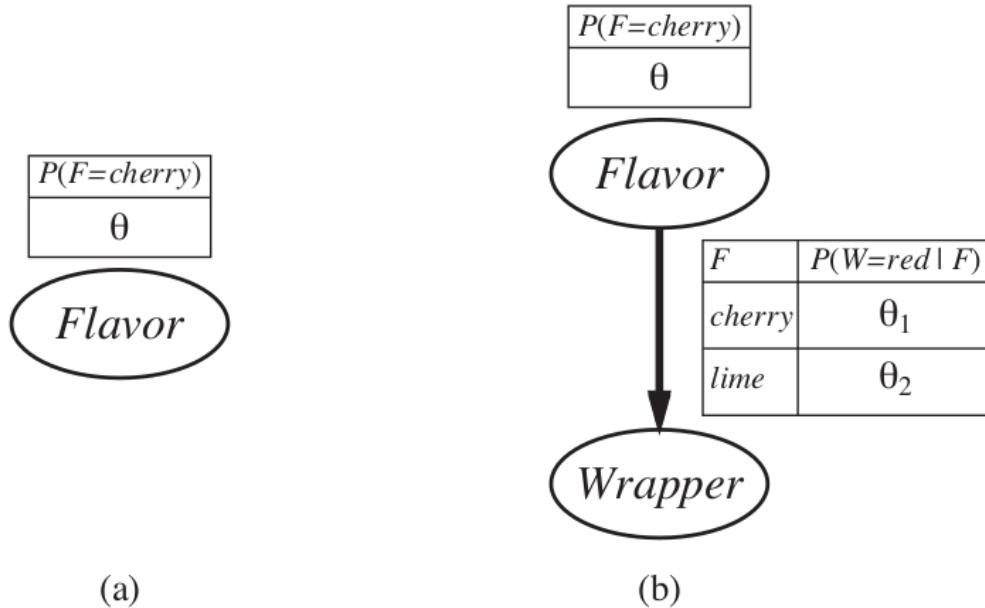


Figure 20.2 (a) Bayesian network model for the case of candies with an unknown proportion of cherries and limes. (b) Model for the case where the wrapper color depends (probabilistically) on the candy flavor.

Maximum likelihood estimation

Suppose we have a set of N i.i.d. observations $\mathbf{d} = \{x_1, \dots, x_N\}$.

The likelihood of the parameters θ is the probability of the data given the parameters

$$P(\mathbf{d}|\theta) = \prod_{j=1}^N P(x_j|\theta).$$

The maximum likelihood estimate (MLE) θ^* of the parameters is the value of θ that maximizes the likelihood

$$\theta^* = \arg \max_{\theta} P(\mathbf{d}|\theta).$$

In practice,

1. Write down the log-likelihood $L(\theta) = \log P(\mathbf{d}|\theta)$ of the parameters θ .
2. Write down the derivative $\frac{\partial L}{\partial \theta}$ of the log-likelihood of the parameters θ .
3. Find the parameter values θ^* such that the derivatives are zero (and check whether the Hessian is negative definite).

Case (a)

What is the fraction θ of cherry candies?

Suppose we unwrap N candies, and get c cherries and $l = N - c$ limes. These are i.i.d. observations, therefore

$$P(\mathbf{d}|\theta) = \prod_{j=1}^N P(x_j|\theta) = \theta^c(1-\theta)^l.$$

Maximize this w.r.t. θ , which is easier for the log-likelihood and leads to

$$\begin{aligned} L(\mathbf{d}|\theta) &= \log P(\mathbf{d}|\theta) = c \log \theta + l \log(1-\theta) \\ \frac{\partial L(\mathbf{d}|\theta)}{\partial \theta} &= \frac{c}{\theta} - \frac{l}{1-\theta} = 0. \end{aligned}$$

Hence $\theta = \frac{c}{N}$.

Case (b)

Red and green wrappers depend probabilistically on flavor. E.g., the likelihood for a cherry candy in green wrapper is

$$\begin{aligned} & P(\text{cherry, green} | \theta, \theta_1, \theta_2) \\ &= P(\text{cherry} | \theta, \theta_1, \theta_2) P(\text{green} | \text{cherry}, \theta, \theta_1, \theta_2) \\ &= \theta(1 - \theta_1). \end{aligned}$$

The likelihood for the parameters, given N candies, r_c red-wrapped cherries, g_c green-wrapped cherries, etc., is

$$\begin{aligned} P(\mathbf{d} | \theta, \theta_1, \theta_2) &= \theta^c (1 - \theta)^l \theta_1^{r_c} (1 - \theta_1)^{g_c} \theta_2^{r_l} (1 - \theta_2)^{g_l} \\ L &= c \log \theta + l \log(1 - \theta) + \\ & r_c \log \theta_1 + g_c \log(1 - \theta_1) + \\ & r_l \log \theta_2 + g_l \log(1 - \theta_2). \end{aligned}$$

The derivatives of \mathbf{L} yield

$$\frac{\partial \mathbf{L}}{\partial \theta} = \frac{c}{\theta} - \frac{l}{1-\theta} = 0 \Rightarrow \theta = \frac{c}{c+l}$$

$$\frac{\partial \mathbf{L}}{\partial \theta_1} = \frac{r_c}{\theta_1} - \frac{g_c}{1-\theta_1} = 0 \Rightarrow \theta_1 = \frac{r_c}{r_c+g_c}$$

$$\frac{\partial \mathbf{L}}{\partial \theta_2} = \frac{r_l}{\theta_2} - \frac{g_l}{1-\theta_2} = 0 \Rightarrow \theta_2 = \frac{r_l}{r_l+g_l}.$$

In case (a), if we unwrap 1 candy and get 1 cherry, what is the MLE? How confident are we in this estimate?

- With small datasets, maximum likelihood estimation can lead to overfitting.
- The MLE does not provide a measure of uncertainty about the parameters.

Bayesian parameter learning

We can treat parameter learning as a **Bayesian inference** problem:

- Make the parameters θ random variables and treat them as hidden variables.
- Specify a **prior** distribution $\mathbf{P}(\theta)$ over the parameters.
- Then, as data arrives, update our beliefs about the parameters to obtain the **posterior** distribution $\mathbf{P}(\theta|\mathbf{d})$.

How should Figure 20.2 (a) be updated?

Case (a)

What is the fraction θ of cherry candies?

We assume a Beta prior

$$P(\theta) = \text{Beta}(\theta|a, b) = \frac{1}{Z} \theta^{a-1} (1-\theta)^{b-1}$$

where Z is a normalization constant.

Then, observing a cherry candy yields the posterior

$$\begin{aligned} P(\theta|\text{cherry}) &\propto P(\text{cherry}|\theta)P(\theta) \\ &= \theta \text{Beta}(\theta|a, b) \\ &= \theta(1-\theta)^{b-1} \theta^{a-1} (1-\theta)^{b-1} \\ &= \theta^a (1-\theta)^{b-1} \\ &= \text{Beta}(\theta|a+1, b). \end{aligned}$$

Case (b)

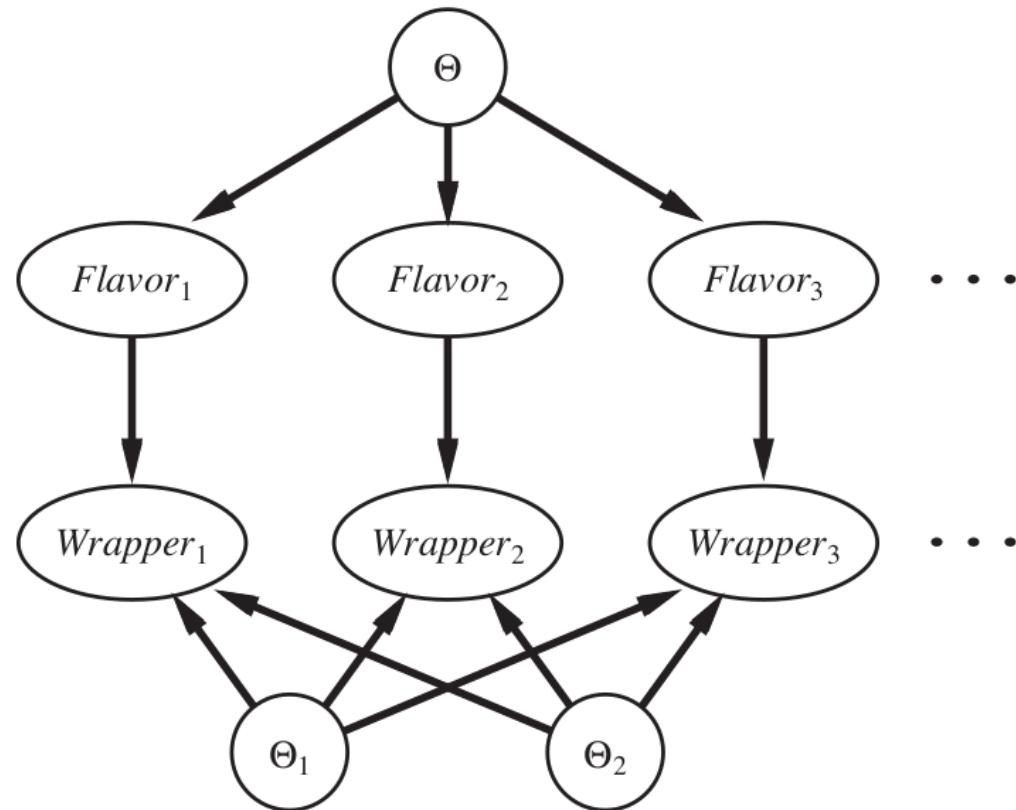


Figure 20.6 A Bayesian network that corresponds to a Bayesian learning process. Posterior distributions for the parameter variables Θ , Θ_1 , and Θ_2 can be inferred from their prior distributions and the evidence in the $Flavor_i$ and $Wrapper_i$ variables.

Maximum a posteriori estimation

When the posterior cannot be computed analytically, we can use **maximum a posteriori** (MAP) estimation, which consists in approximating the posterior with the point estimate θ^* that maximizes the posterior distribution, i.e.,

$$\theta^* = \arg \max_{\theta} P(\theta|\mathbf{d}) = \arg \max_{\theta} P(\mathbf{d}|\theta)P(\theta).$$

(demo)

Summary

- A Bayesian Network specifies a full joint distribution. BNs are often exponentially smaller than an explicitly enumerated joint distribution.
- The topology of a Bayesian network encodes conditional independence assumptions between random variables.
- Inference is the problem of computing a marginal and/or a conditional probability distribution from a joint probability distribution.
 - Exact inference is possible for simple Bayesian networks, but is intractable for most probabilistic models of practical interest.
 - Approximate inference algorithms are used in practice.
- Parameters of a Bayesian network can be learned from data using maximum likelihood estimation or Bayesian inference.

Introduction to Artificial Intelligence

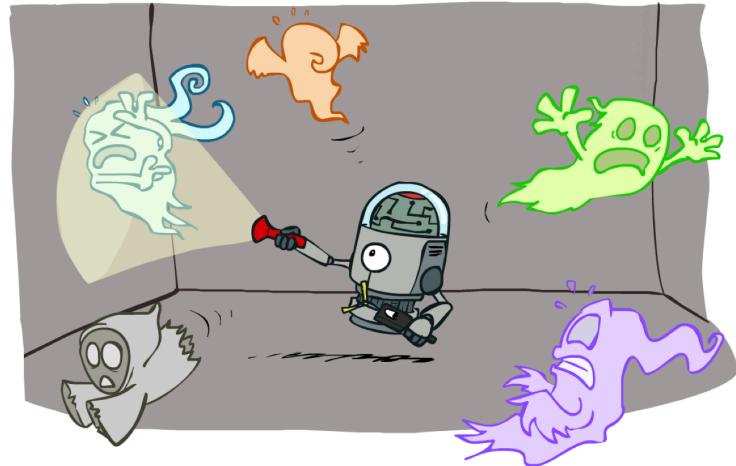
Lecture 6: Reasoning over time

Prof. Gilles Louppe
g.louppe@uliege.be

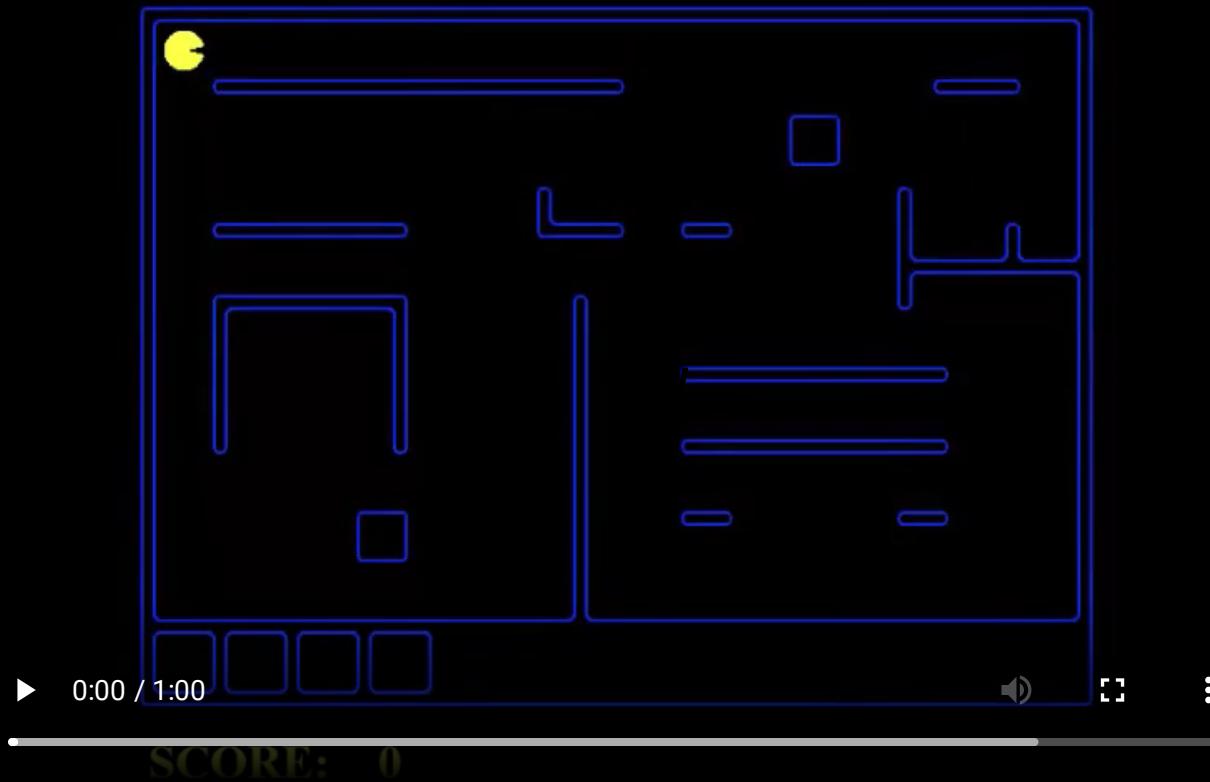
Today

Maintain a **belief state** about the world, and update it as time passes and evidence is collected.

- Markov models
 - Markov processes
 - Inference tasks
 - Hidden Markov models
- Filters
 - Kalman filter
 - Particle filter



Do not overlook this lecture!



Pacman revenge: How to make good use of the sonar readings?

Markov models

Modelling the passage of time

We will consider the world as a **discrete** series of time slices, each of which contains a set of random variables:

- \mathbf{X}_t denotes the set of **unobservable** state variables at time t .
- \mathbf{E}_t denotes the set of **observable** evidence variables at time t .

We specify

- a prior $\mathbf{P}(\mathbf{X}_0)$ that defines our initial belief state over hidden state variables,
- a transition model $\mathbf{P}(\mathbf{X}_t | \mathbf{X}_{0:t-1})$ (for $t > 0$) that defines the probability distribution over the latest state variables, given the previous (unobserved) values,
- a sensor model $\mathbf{P}(\mathbf{E}_t | \mathbf{X}_{0:t}, \mathbf{E}_{0:t-1})$ (for $t > 0$) that defines the probability distribution over the latest evidence variables, given all previous (observed and unobserved) values.

Markov processes

Markov assumption

The current state of the world depends only on its immediate previous state(s), i.e., \mathbf{X}_t depends on only a bounded subset of $\mathbf{X}_{0:t-1}$.

Random processes that satisfy this assumption are called **Markov processes** or **Markov chains**.

First-order Markov processes

Markov processes such that

$$\mathbf{P}(\mathbf{X}_t | \mathbf{X}_{0:t-1}) = \mathbf{P}(\mathbf{X}_t | \mathbf{X}_{t-1})$$

i.e., \mathbf{X}_t and $\mathbf{X}_{0:t-2}$ are conditionally independent given \mathbf{X}_{t-1} .



Sensor Markov assumption

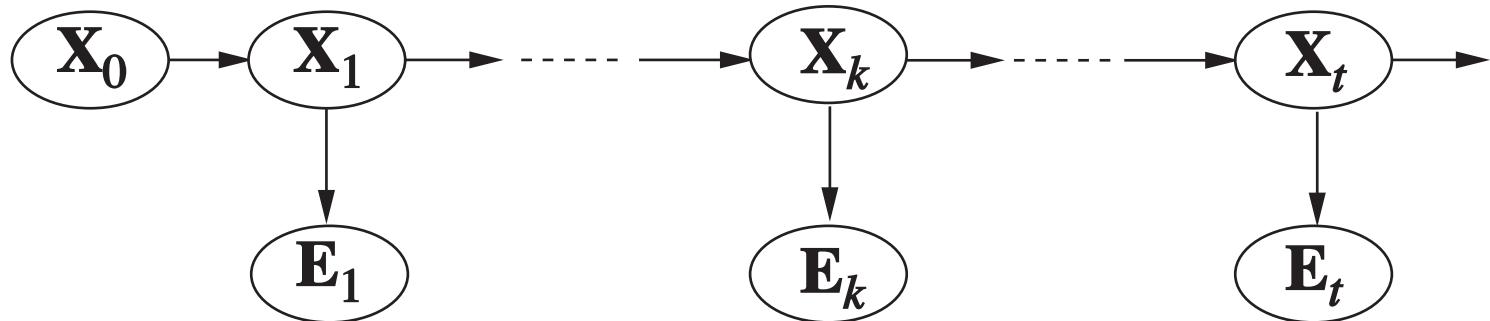
We make a (first-order) sensor Markov assumption

$$\mathbf{P}(\mathbf{E}_t | \mathbf{X}_{0:t}, \mathbf{E}_{0:t-1}) = \mathbf{P}(\mathbf{E}_t | \mathbf{X}_t).$$

Stationarity assumption

The transition and the sensor models are the same for all t (i.e., the laws of physics do not change with time).

Joint distribution

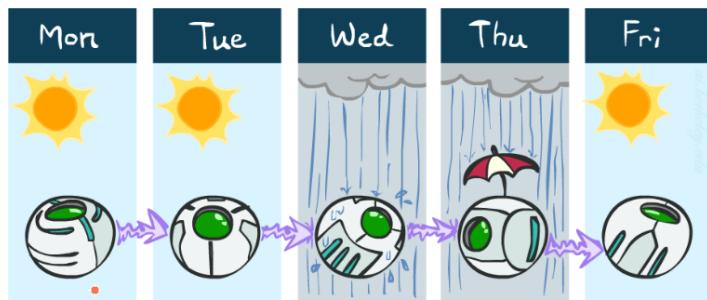
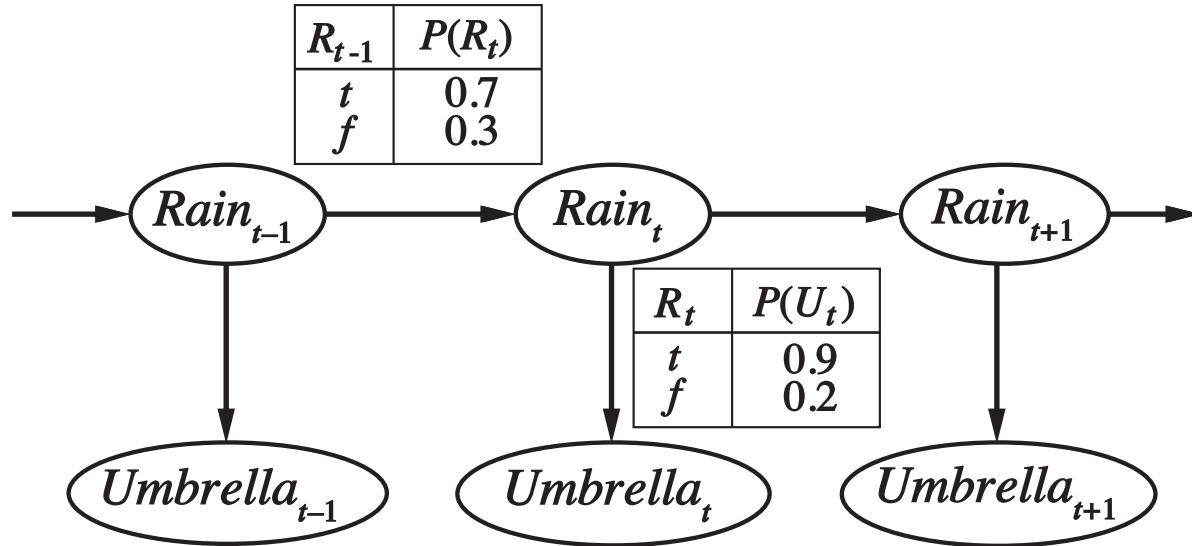


A Markov chain coupled with a sensor model can be represented as a [growable](#) Bayesian network, unrolled infinitely through time.

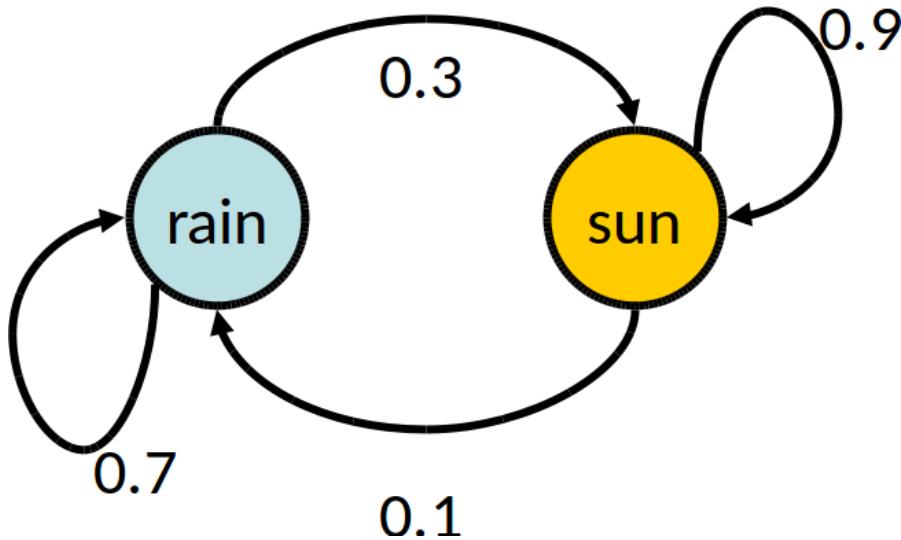
The [joint distribution](#) of all its variables up to t is

$$\mathbf{P}(\mathbf{X}_{0:t}, \mathbf{E}_{1:t}) = \mathbf{P}(\mathbf{X}_0) \prod_{i=1}^t \mathbf{P}(\mathbf{X}_i | \mathbf{X}_{i-1}) \mathbf{P}(\mathbf{E}_i | \mathbf{X}_i).$$

Example: Will you take your umbrella today?



- $P(Umbrella_t | Rain_t)$?
- $P(Rain_t | Umbrella_{0:t-1})$?
- $P(Rain_{t+2} | Rain_t)$?

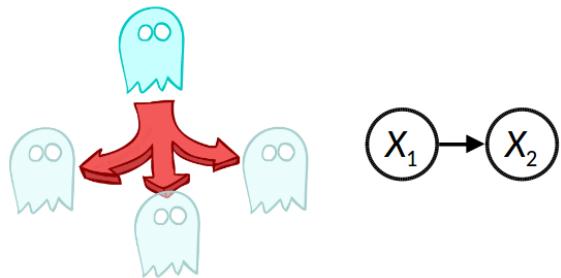


The transition model $\mathbf{P}(\text{Rain}_t | \text{Rain}_{t-1})$ can equivalently be represented by a state transition diagram.

Inference tasks

- Prediction: $\mathbf{P}(\mathbf{X}_{t+k} | \mathbf{e}_{1:t})$ for $k > 0$
 - Computing the posterior distribution over future states.
 - Used for evaluation of possible action sequences.
- Filtering: $\mathbf{P}(\mathbf{X}_t | \mathbf{e}_{1:t})$
 - Filtering is what a rational agent does to keep track of the current hidden state \mathbf{X}_t , its **belief state**, so that rational decisions can be made.
- Smoothing: $\mathbf{P}(\mathbf{X}_k | \mathbf{e}_{1:t})$ for $0 \leq k < t$
 - Computing the posterior distribution over past states.
 - Used for building better estimates, since it incorporates more evidence.
 - Essential for learning.
- Most likely explanation: $\arg \max_{\mathbf{x}_{1:t}} P(\mathbf{x}_{1:t} | \mathbf{e}_{1:t})$
 - Decoding with a noisy channel, speech recognition, etc.

Base cases



$$\begin{aligned}\mathbf{P}(\mathbf{X}_2) &= \sum_{\mathbf{x}_1} \mathbf{P}(\mathbf{X}_2, \mathbf{x}_1) \\ &= \sum_{\mathbf{x}_1} P(\mathbf{x}_1) \mathbf{P}(\mathbf{X}_2 | \mathbf{x}_1)\end{aligned}$$

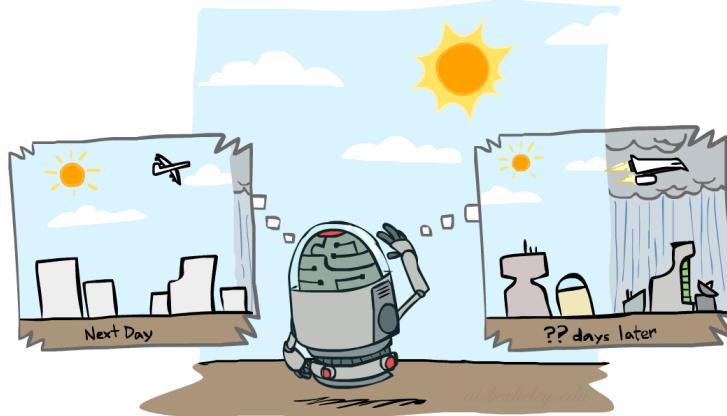
(Predict) Push $\mathbf{P}(\mathbf{X}_1)$ forward through the transition model.



$$\begin{aligned}\mathbf{P}(\mathbf{X}_1 | \mathbf{e}_1) &= \frac{\mathbf{P}(\mathbf{e}_1 | \mathbf{X}_1) \mathbf{P}(\mathbf{X}_1)}{P(\mathbf{e}_1)} \\ &\propto \mathbf{P}(\mathbf{e}_1 | \mathbf{X}_1) \mathbf{P}(\mathbf{X}_1)\end{aligned}$$

(Update) Update $\mathbf{P}(\mathbf{X}_1)$ with the evidence \mathbf{e}_1 , given the sensor model.

Prediction



To predict the future $\mathbf{P}(\mathbf{X}_{t+k} | \mathbf{e}_{1:t})$:

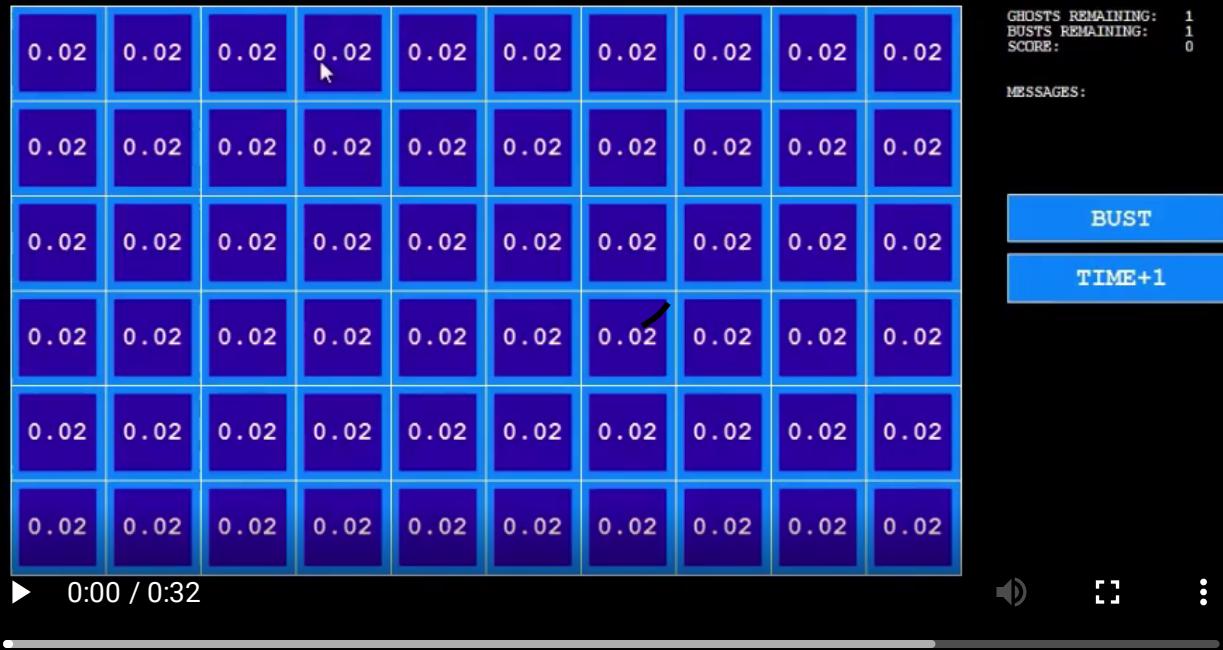
- Push the prior belief state $\mathbf{P}(\mathbf{X}_t | \mathbf{e}_{1:t})$ through the transition model:

$$\mathbf{P}(\mathbf{X}_{t+1} | \mathbf{e}_{1:t}) = \sum_{\mathbf{x}_t} \mathbf{P}(\mathbf{X}_{t+1} | \mathbf{x}_t) P(\mathbf{x}_t | \mathbf{e}_{1:t})$$

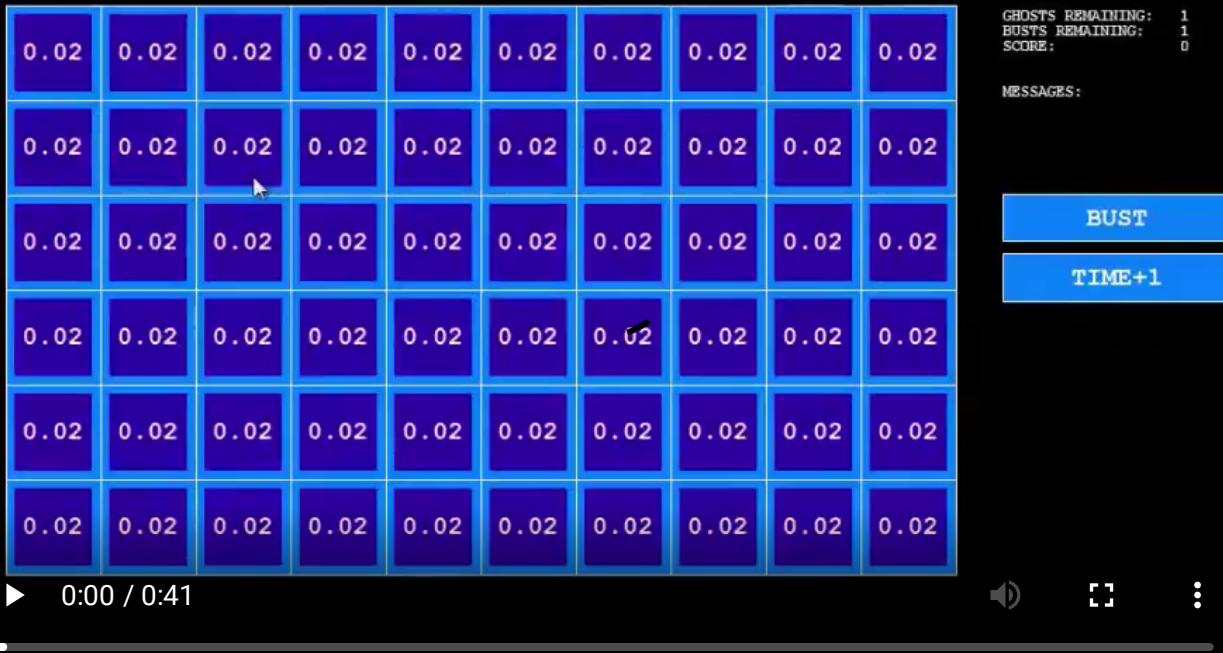
- Repeat up to $t + k$, using $\mathbf{P}(\mathbf{X}_{t+k-1} | \mathbf{e}_{1:t})$ to compute $\mathbf{P}(\mathbf{X}_{t+k} | \mathbf{e}_{1:t})$.



Random dynamics



Circular dynamics



Whirlpool dynamics

<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
<0.01	<0.01	1.00	<0.01	<0.01	<0.01
<0.01	<0.01	<0.01	<0.01	<0.01	<0.01

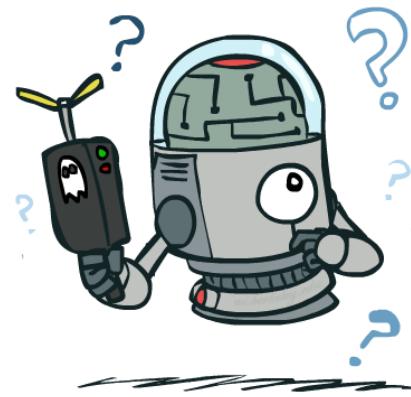
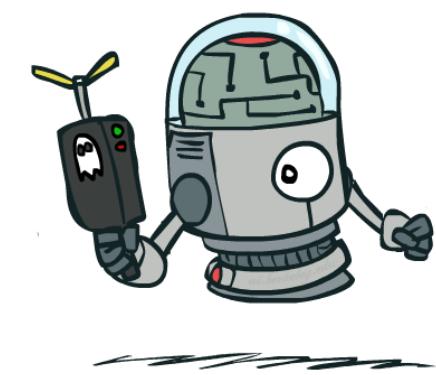
$T = 1$

<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
<0.01	<0.01	0.06	<0.01	<0.01	<0.01
<0.01	0.76	0.06	0.06	<0.01	<0.01
<0.01	<0.01	0.06	<0.01	<0.01	<0.01

$T = 2$

0.05	0.01	0.05	<0.01	<0.01	<0.01
0.02	0.14	0.11	0.35	<0.01	<0.01
0.07	0.03	0.05	<0.01	0.03	<0.01
0.03	0.03	<0.01	<0.01	<0.01	<0.01

$T = 5$



As time passes, uncertainty (usually) increases in the absence of new evidence.

Stationary distributions

What if $t \rightarrow \infty$?

- For most chains, the influence of the initial distribution gets lesser and lesser over time.
- Eventually, the distribution converges to a fixed point, called a **stationary distribution**.
- This distribution is such that

$$\mathbf{P}(\mathbf{X}_\infty) = \mathbf{P}(\mathbf{X}_{\infty+1}) = \sum_{\mathbf{x}_\infty} \mathbf{P}(\mathbf{X}_{\infty+1} | \mathbf{x}_\infty) P(\mathbf{x}_\infty).$$

\mathbf{X}_{t-1}	\mathbf{X}_t	P
sun	sun	0.9
sun	rain	0.1
rain	sun	0.3
rain	rain	0.7

Example

$$\begin{aligned}
 P(\mathbf{X}_\infty = \text{sun}) &= P(\mathbf{X}_{\infty+1} = \text{sun}) \\
 &= P(\mathbf{X}_{\infty+1} = \text{sun} | \mathbf{X}_\infty = \text{sun})P(\mathbf{X}_\infty = \text{sun}) \\
 &\quad + P(\mathbf{X}_{\infty+1} = \text{sun} | \mathbf{X}_\infty = \text{rain})P(\mathbf{X}_\infty = \text{rain}) \\
 &= 0.9P(\mathbf{X}_\infty = \text{sun}) + 0.3P(\mathbf{X}_\infty = \text{rain})
 \end{aligned}$$

Therefore, $P(\mathbf{X}_\infty = \text{sun}) = 3P(\mathbf{X}_\infty = \text{rain})$.

Which implies that $P(\mathbf{X}_\infty = \text{sun}) = \frac{3}{4}$ and $P(\mathbf{X}_\infty = \text{rain}) = \frac{1}{4}$.

Filtering

0.05	0.01	0.05	<0.01	<0.01	<0.01
0.02	0.14	0.11	0.35	<0.01	<0.01
0.07	0.03	0.05	<0.01	0.03	<0.01
0.03	0.03	<0.01	<0.01	<0.01	<0.01

Before observation

<0.01	<0.01	<0.01	<0.01	0.02	<0.01
<0.01	<0.01	<0.01	0.83	0.02	<0.01
<0.01	<0.01	0.11	<0.01	<0.01	<0.01
<0.01	<0.01	<0.01	<0.01	<0.01	<0.01

After observation

With new evidence, uncertainty decreases. Beliefs get reweighted. But how?

Bayes filter

An agent maintains a **belief state** estimate $\mathbf{P}(\mathbf{X}_t | \mathbf{e}_{1:t})$ and updates it as new evidences \mathbf{e}_{t+1} are collected.

This process can be implemented as a recursive Bayesian estimation procedure $\mathbf{P}(\mathbf{X}_{t+1} | \mathbf{e}_{1:t+1}) = f(\mathbf{e}_{t+1}, \mathbf{P}(\mathbf{X}_t | \mathbf{e}_{1:t}))$ that alternates between two steps:

- (Predict step): Project the current belief state forward from t to $t + 1$ through the transition model.
- (Update step): Update this new state using the evidence \mathbf{e}_{t+1} .

Formally, the Bayes filter is defined as

$$\begin{aligned}\mathbf{P}(\mathbf{X}_{t+1} | \mathbf{e}_{1:t+1}) &= \mathbf{P}(\mathbf{X}_{t+1} | \mathbf{e}_{1:t}, \mathbf{e}_{t+1}) \\ &\propto \mathbf{P}(\mathbf{e}_{t+1} | \mathbf{X}_{t+1}, \mathbf{e}_{1:t}) \mathbf{P}(\mathbf{X}_{t+1} | \mathbf{e}_{1:t}) \\ &\propto \mathbf{P}(\mathbf{e}_{t+1} | \mathbf{X}_{t+1}) \mathbf{P}(\mathbf{X}_{t+1} | \mathbf{e}_{1:t}) \\ &\propto \mathbf{P}(\mathbf{e}_{t+1} | \mathbf{X}_{t+1}) \sum_{\mathbf{x}_t} \mathbf{P}(\mathbf{X}_{t+1} | \mathbf{x}_t, \mathbf{e}_{1:t}) P(\mathbf{x}_t | \mathbf{e}_{1:t}) \\ &\propto \mathbf{P}(\mathbf{e}_{t+1} | \mathbf{X}_{t+1}) \sum_{\mathbf{x}_t} \mathbf{P}(\mathbf{X}_{t+1} | \mathbf{x}_t) P(\mathbf{x}_t | \mathbf{e}_{1:t})\end{aligned}$$

where

- the normalization constant

$$Z = P(\mathbf{e}_{t+1} | \mathbf{e}_{1:t}) = \sum_{\mathbf{x}_{t+1}} P(\mathbf{e}_{t+1} | \mathbf{x}_{t+1}) P(\mathbf{x}_{t+1} | \mathbf{e}_{1:t})$$

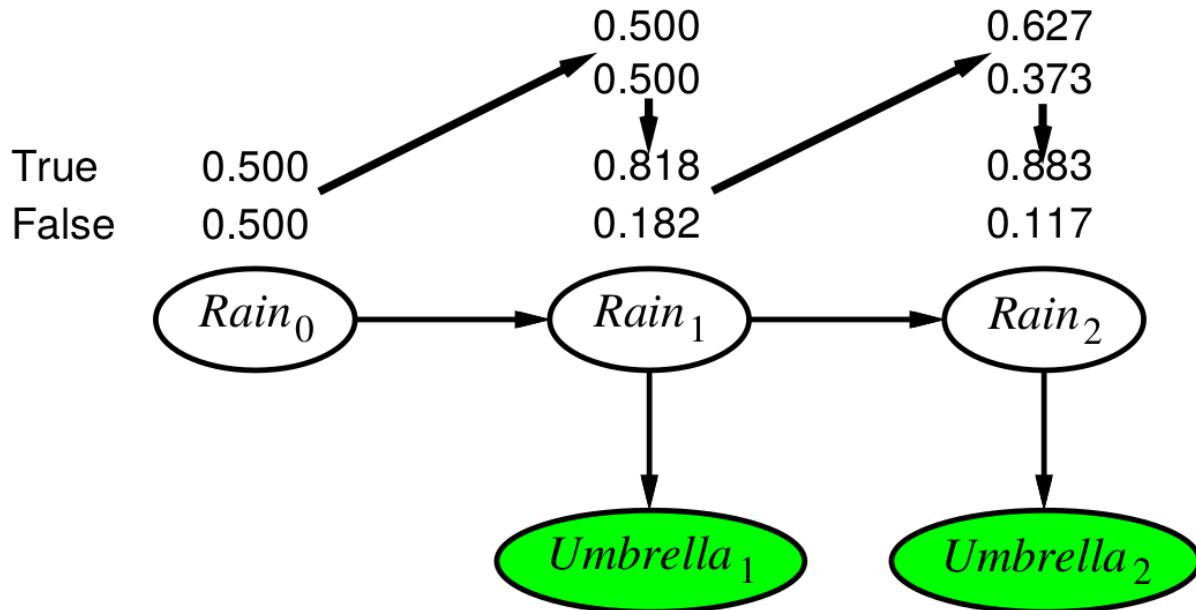
is used to make probabilities sum to 1;

- in the last expression, the first and second terms are given by the model while the third is obtained recursively.

We can think of $\mathbf{P}(\mathbf{X}_t | \mathbf{e}_{1:t})$ as a message $\mathbf{f}_{1:t}$ that is propagated **forward** along the sequence, modified by each transition and updated by each new observation.

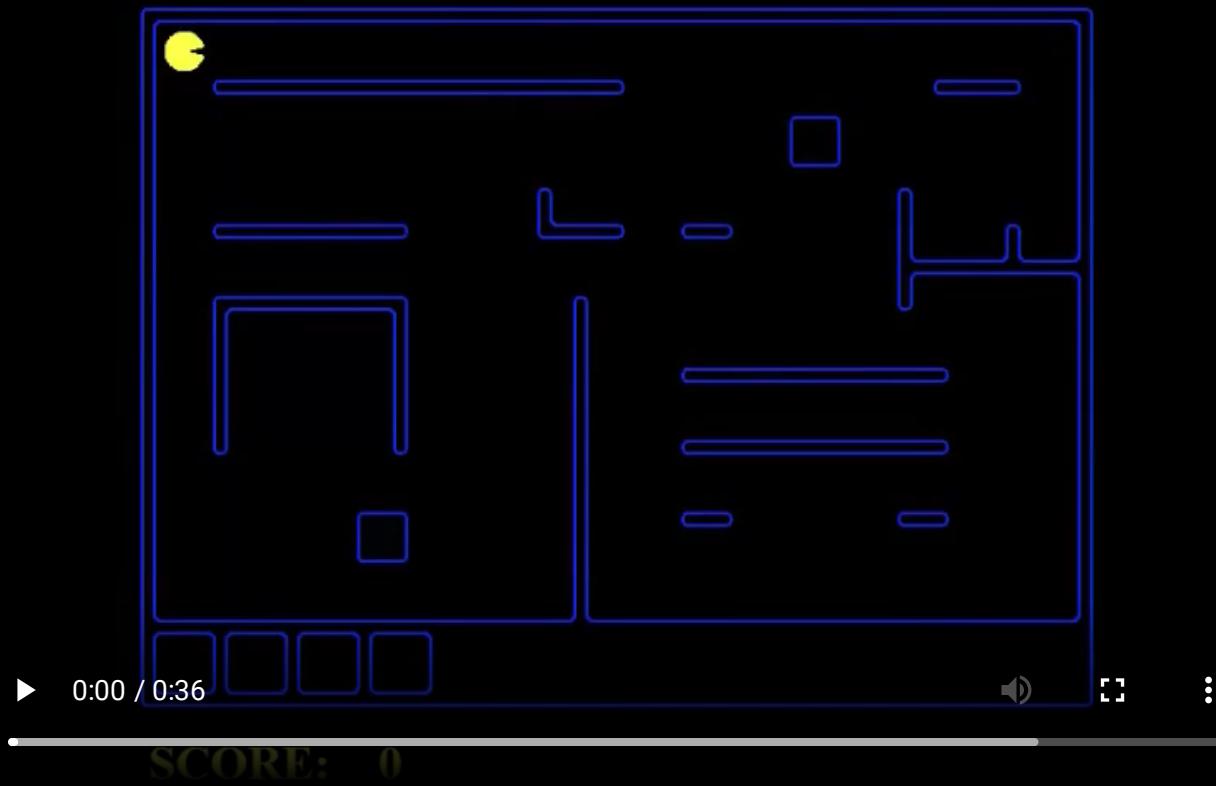
Thus, the process can be implemented as $\mathbf{f}_{1:t+1} \propto \text{forward}(\mathbf{f}_{1:t}, \mathbf{e}_{t+1})$. Its complexity is constant (in time and space) with t .

Example



R_{t-1}	$P(R_t)$
true	0.7
false	0.3

R_t	$P(U_t)$
true	0.9
false	0.2



Ghostbusters with a Bayes filter

Smoothing

We want to compute $\mathbf{P}(\mathbf{X}_k | \mathbf{e}_{1:t})$ for $0 \leq k < t$.

Dividing the evidence $\mathbf{e}_{1:t}$ into $\mathbf{e}_{1:k}$ and $\mathbf{e}_{k+1:t}$, we have

$$\begin{aligned}\mathbf{P}(\mathbf{X}_k | \mathbf{e}_{1:t}) &= \mathbf{P}(\mathbf{X}_k | \mathbf{e}_{1:k}, \mathbf{e}_{k+1:t}) \\ &\propto \mathbf{P}(\mathbf{X}_k | \mathbf{e}_{1:k}) \mathbf{P}(\mathbf{e}_{k+1:t} | \mathbf{X}_k, \mathbf{e}_{1:k}) \\ &\propto \mathbf{P}(\mathbf{X}_k | \mathbf{e}_{1:k}) \mathbf{P}(\mathbf{e}_{k+1:t} | \mathbf{X}_k).\end{aligned}$$

Let the **backward** message $\mathbf{b}_{k+1:t}$ correspond to $\mathbf{P}(\mathbf{e}_{k+1:t} | \mathbf{X}_k)$. Then,

$$\mathbf{P}(\mathbf{X}_k | \mathbf{e}_{1:t}) = \alpha \mathbf{f}_{1:k} \times \mathbf{b}_{k+1:t},$$

where \times is a pointwise multiplication of vectors.

This backward message can be computed using backwards recursion:

$$\begin{aligned}\mathbf{P}(\mathbf{e}_{k+1:t} | \mathbf{X}_k) &= \sum_{\mathbf{x}_{k+1}} \mathbf{P}(\mathbf{e}_{k+1:t} | \mathbf{X}_k, \mathbf{x}_{k+1}) \mathbf{P}(\mathbf{x}_{k+1} | \mathbf{X}_k) \\ &= \sum_{\mathbf{x}_{k+1}} P(\mathbf{e}_{k+1:t} | \mathbf{x}_{k+1}) \mathbf{P}(\mathbf{x}_{k+1} | \mathbf{X}_k) \\ &= \sum_{\mathbf{x}_{k+1}} P(\mathbf{e}_{k+1} | \mathbf{x}_{k+1}) P(\mathbf{e}_{k+2:t} | \mathbf{x}_{k+1}) \mathbf{P}(\mathbf{x}_{k+1} | \mathbf{X}_k).\end{aligned}$$

The first and last factors are given by the model. The second factor is obtained recursively. Therefore,

$$\mathbf{b}_{k+1:t} = \text{backward}(\mathbf{b}_{k+2:t}, \mathbf{e}_{k+1}).$$

Forward-backward algorithm

function FORWARD-BACKWARD(\mathbf{ev} , $prior$) **returns** a vector of probability distributions

inputs: \mathbf{ev} , a vector of evidence values for steps $1, \dots, t$

$prior$, the prior distribution on the initial state, $\mathbf{P}(\mathbf{X}_0)$

local variables: \mathbf{fv} , a vector of forward messages for steps $0, \dots, t$

\mathbf{b} , a representation of the backward message, initially all 1s

\mathbf{sv} , a vector of smoothed estimates for steps $1, \dots, t$

$\mathbf{fv}[0] \leftarrow prior$

for $i = 1$ **to** t **do**

$\mathbf{fv}[i] \leftarrow \text{FORWARD}(\mathbf{fv}[i - 1], \mathbf{ev}[i])$

for $i = t$ **downto** 1 **do**

$\mathbf{sv}[i] \leftarrow \text{NORMALIZE}(\mathbf{fv}[i] \times \mathbf{b})$

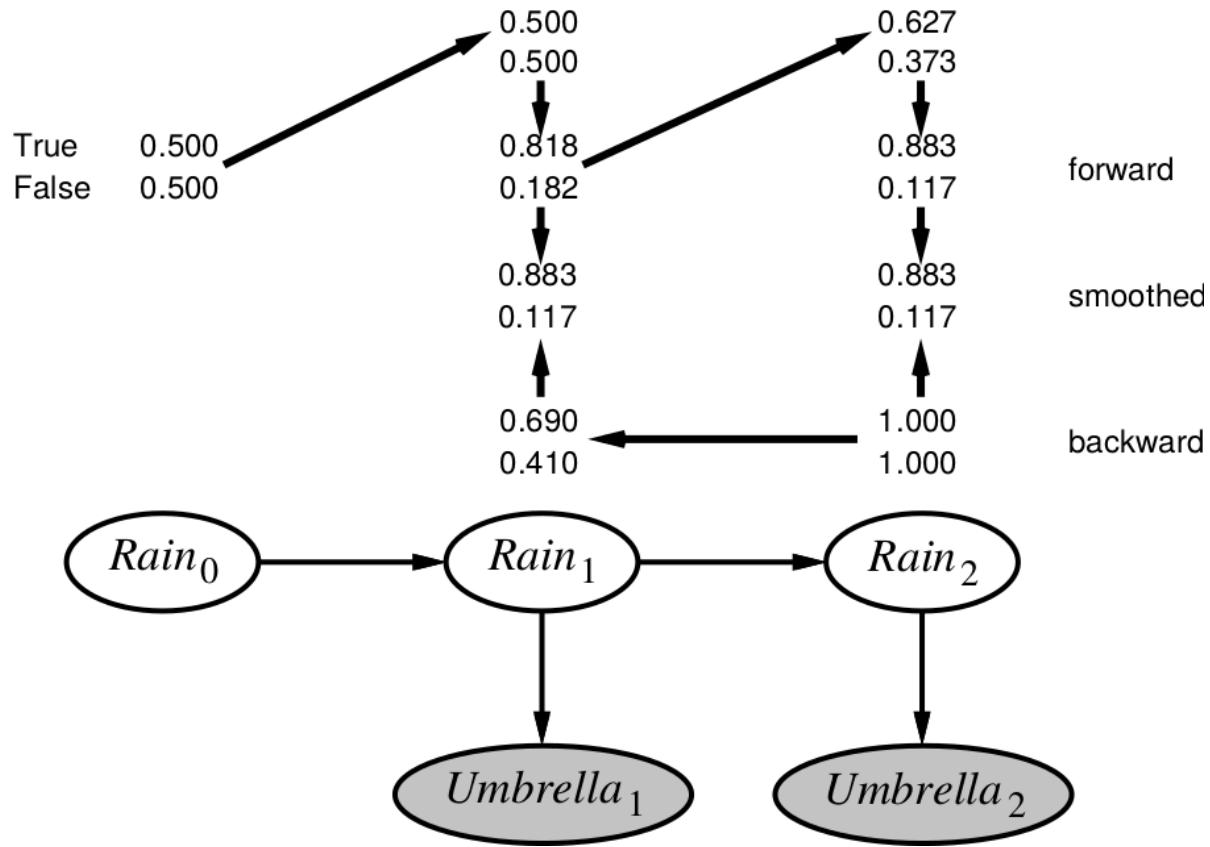
$\mathbf{b} \leftarrow \text{BACKWARD}(\mathbf{b}, \mathbf{ev}[i])$

return \mathbf{sv}

Complexity:

- Smoothing for a particular time step k takes: $O(t)$
- Smoothing a whole sequence (because of caching): $O(t)$

Example



Most likely explanation





Suppose that **[true, true, false, true, true]** is the umbrella sequence.

- What is the weather sequence that is the most likely to explain this?
- Among all 2^5 sequences, is there an (efficient) way to find the most likely one?

The most likely sequence **is not** the sequence of the most likely states!

The most likely path to each \mathbf{x}_{t+1} , is the most likely path to **some** \mathbf{x}_t plus one more step. Therefore,

$$\begin{aligned} & \max_{\mathbf{x}_{1:t}} \mathbf{P}(\mathbf{x}_{1:t}, \mathbf{X}_{t+1} | \mathbf{e}_{1:t+1}) \\ & \propto \mathbf{P}(\mathbf{e}_{t+1} | \mathbf{X}_{t+1}) \max_{\mathbf{x}_t} (\mathbf{P}(\mathbf{X}_{t+1} | \mathbf{x}_t) \max_{\mathbf{x}_{1:t-1}} \mathbf{P}(\mathbf{x}_{1:t-1}, \mathbf{x}_t | \mathbf{e}_{1:t})). \end{aligned}$$

This is identical to filtering, except that

- the forward message $\mathbf{f}_{1:t} = \mathbf{P}(\mathbf{X}_t | \mathbf{e}_{1:t})$ is replaced with

$$\mathbf{m}_{1:t} = \max_{\mathbf{x}_{1:t-1}} \mathbf{P}(\mathbf{x}_{1:t-1}, \mathbf{X}_t | \mathbf{e}_{1:t}),$$

where $\mathbf{m}_{1:t}(i)$ gives the probability of the most likely path to state i .

- The update has its sum replaced by max.

The resulting algorithm is called the **Viterbi algorithm**, which computes the most likely explanation as

$$\mathbf{m}_{1:t+1} \propto \mathbf{P}(\mathbf{e}_{t+1} | \mathbf{X}_{t+1}) \max_{\mathbf{x}_t} \mathbf{P}(\mathbf{X}_{t+1} | \mathbf{x}_t) \mathbf{m}_{1:t}.$$

Its complexity is linear in t , the length of the sequence.

Example

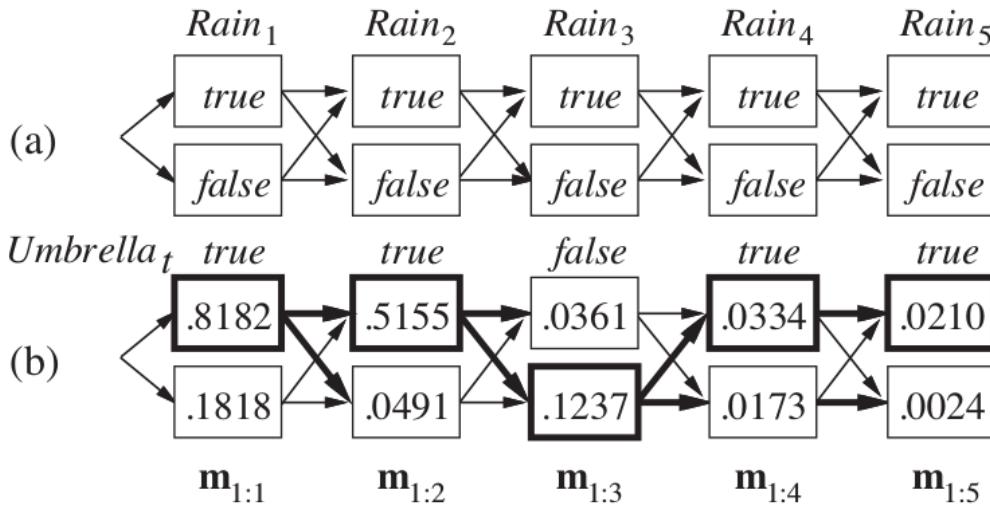


Figure 15.5 (a) Possible state sequences for $Rain_t$ can be viewed as paths through a graph of the possible states at each time step. (States are shown as rectangles to avoid confusion with nodes in a Bayes net.) (b) Operation of the Viterbi algorithm for the umbrella observation sequence [true, true, false, true, true]. For each t , we have shown the values of the message $\mathbf{m}_{1:t}$, which gives the probability of the best sequence reaching each state at time t . Also, for each state, the bold arrow leading into it indicates its best predecessor as measured by the product of the preceding sequence probability and the transition probability. Following the bold arrows back from the most likely state in $\mathbf{m}_{1:5}$ gives the most likely sequence.

Hidden Markov models

So far, we described Markov processes over arbitrary sets of state variables \mathbf{X}_t and evidence variables \mathbf{E}_t .

- A **hidden Markov model** (HMM) is a Markov process in which the state \mathbf{X}_t and the evidence \mathbf{E}_t are both **single discrete** random variables.
 - $\mathbf{X}_t = X_t$, with domain $D_{X_t} = \{1, \dots, S\}$
 - $\mathbf{E}_t = E_t$, with domain $D_{E_t} = \{1, \dots, R\}$
- This restricted structure allows for a reformulation of the forward-backward algorithm in terms of matrix-vector operations.

Note on terminology

Some authors instead divide Markov models into two classes, depending on the observability of the system state:

- Observable system state: Markov chains
- Partially-observable system state: Hidden Markov models.

We follow here instead the terminology of the textbook, as defined in the previous slide.

Simplified matrix algorithms

- The prior $\mathbf{P}(X_0)$ becomes a (normalized) column vector $\mathbf{f}_0 \in \mathbb{R}_+^S$.
- The transition model $\mathbf{P}(X_t|X_{t-1})$ becomes an $S \times S$ transition matrix \mathbf{T} , such that

$$\mathbf{T}_{ij} = P(X_t = j | X_{t-1} = i).$$

- The sensor model $\mathbf{P}(E_t|X_t)$ is defined as an $S \times R$ sensor matrix \mathbf{B} , such that

$$\mathbf{B}_{ij} = P(E_t = j | X_t = i).$$

- Let the observation matrix \mathbf{O}_t be a diagonal matrix whose elements corresponds to the column e_t of the sensor matrix \mathbf{B} .
- If we use column vectors to represent forward and backward messages, then we have

$$\mathbf{f}_{1:t+1} = \alpha \mathbf{O}_{t+1} \mathbf{T}^T \mathbf{f}_{1:t}$$

$$\mathbf{b}_{k+1:t} = \mathbf{T} \mathbf{O}_{k+1} \mathbf{b}_{k+2:t},$$

where $\mathbf{b}_{t+1:t}$ is an all-one vector of size S .

- Therefore the forward-backward algorithm needs time $O(S^2 t)$ and space $O(St)$.

Example

Suppose that [true, true, false, true, true] is the umbrella sequence.

$$\mathbf{f}_0 = \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}$$

$$\mathbf{T} = \begin{pmatrix} 0.7 & 0.3 \\ 0.3 & 0.7 \end{pmatrix}$$

$$\mathbf{B} = \begin{pmatrix} 0.9 & 0.1 \\ 0.2 & 0.8 \end{pmatrix}$$

$$\mathbf{O}_1 = \mathbf{O}_2 = \mathbf{O}_4 = \mathbf{O}_5 = \begin{pmatrix} 0.9 & 0.0 \\ 0.0 & 0.2 \end{pmatrix}$$

$$\mathbf{O}_3 = \begin{pmatrix} 0.1 & 0.0 \\ 0.0 & 0.8 \end{pmatrix}$$

See code/lecture6-forward-backward.ipynb for the execution.

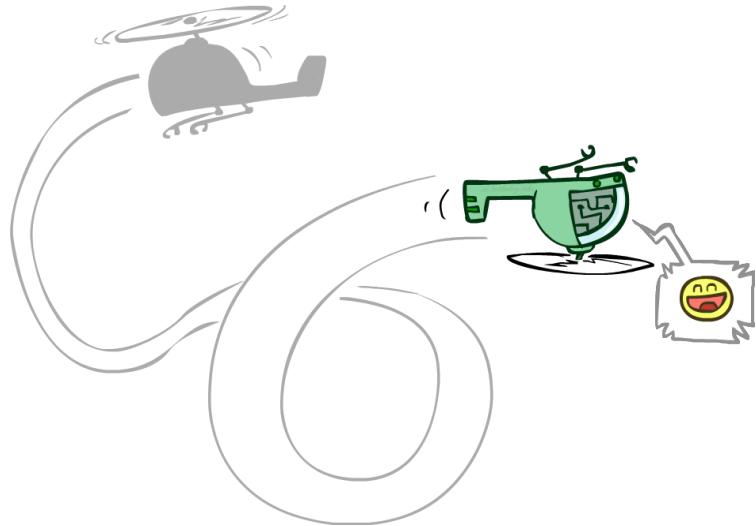
Stationary distribution

The stationary distribution \mathbf{f} of a HMM is a distribution such that

$$\mathbf{f} = \mathbf{T}^T \mathbf{f}.$$

Therefore, the stationary distribution corresponds to a (normalized) eigenvector of the transposed transition matrix with an eigenvalue of 1 .

Filters



Suppose we want to track the position and velocity of a robot from noisy observations collected over time.

Formally, we want to estimate **continuous** state variables such as

- the position \mathbf{X}_t of the robot at time t ,
- the velocity $\dot{\mathbf{X}}_t$ of the robot at time t .

We assume **discrete** time steps.

Continuous variables

Let $X : \Omega \rightarrow D_X$ be a random variable.

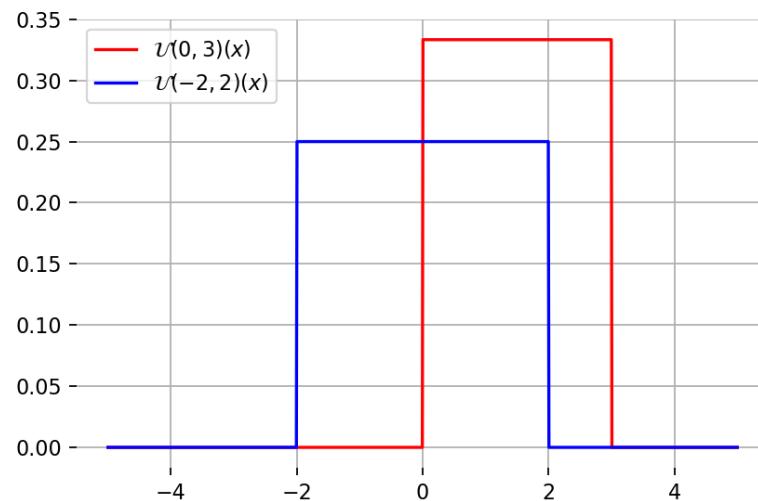
- When D_X is uncountably infinite (e.g., $D_X = \mathbb{R}$), X is called a **continuous random variable**.
- If X is absolutely continuous, its probability distribution is described by a **density function** p that assigns a probability to any interval $[a, b] \subseteq D_X$ such that

$$P(a < X \leq b) = \int_a^b p(x)dx,$$

where p is non-negative piecewise continuous and such that

$$\int_{D_X} p(x)dx = 1.$$

Uniform

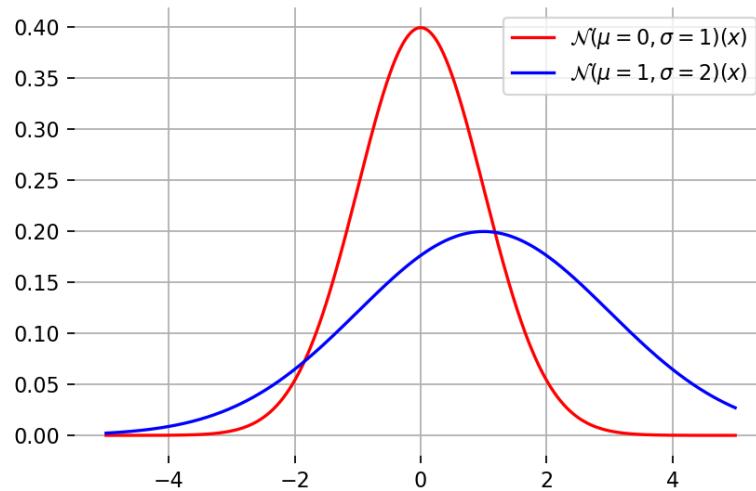


The uniform distribution $\mathcal{U}(a, b)$ is described by the density function

$$p(x) = \begin{cases} \frac{1}{b-a} & \text{if } x \in [a, b] \\ 0 & \text{otherwise} \end{cases}$$

where $a \in \mathbb{R}$ and $b \in \mathbb{R}$ are the bounds of its support.

Normal

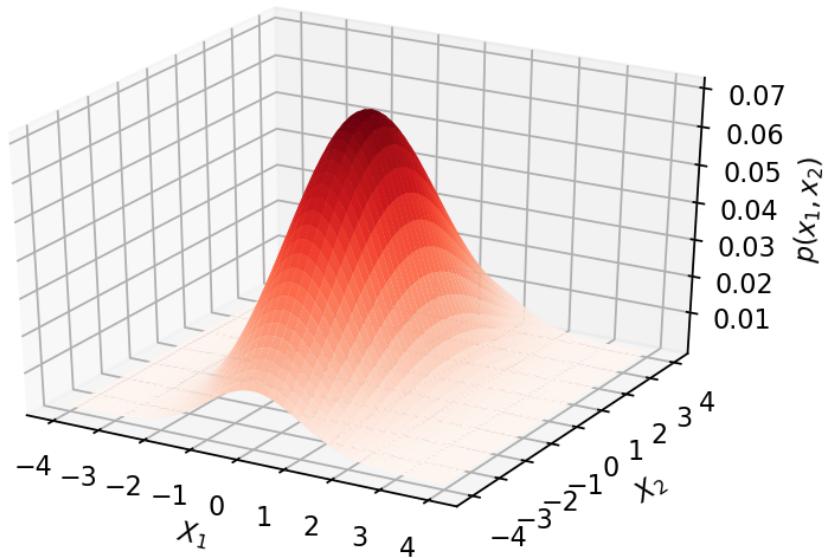


The normal (or Gaussian) distribution $\mathcal{N}(\mu, \sigma)$ is described by the density function

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

where $\mu \in \mathbb{R}$ and $\sigma \in \mathbb{R}^+$ are its mean and standard deviation parameters.

Multivariate normal



The multivariate normal distribution generalizes to n random variables. Its (joint) density function is defined as

$$p(\mathbf{x} = x_1, \dots, x_n) = \frac{1}{\sqrt{(2\pi)^n |\Sigma|}} \exp \left(-\frac{1}{2} (\mathbf{x} - \mathbf{m})^T \Sigma^{-1} (\mathbf{x} - \mathbf{m}) \right)$$

where $\mathbf{m} \in \mathbb{R}^n$ and $\Sigma \in \mathbb{R}^{n \times n}$ is positive semi-definite.

Cheat sheet for Gaussian models (Särkkä, 2013)

If \mathbf{x} and \mathbf{y} have the joint Gaussian distribution

$$p\left(\begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix}\right) = \mathcal{N}\left(\left(\begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} \middle| \begin{pmatrix} \mathbf{a} \\ \mathbf{b} \end{pmatrix}, \begin{pmatrix} \mathbf{A} & \mathbf{C} \\ \mathbf{C}^T & \mathbf{B} \end{pmatrix}\right)\right),$$

then the marginal and conditional distributions of \mathbf{x} and \mathbf{y} are given by

$$p(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\mathbf{a}, \mathbf{A})$$

$$p(\mathbf{y}) = \mathcal{N}(\mathbf{y}|\mathbf{b}, \mathbf{B})$$

$$p(\mathbf{x}|\mathbf{y}) = \mathcal{N}(\mathbf{x}|\mathbf{a} + \mathbf{C}\mathbf{B}^{-1}(\mathbf{y} - \mathbf{b}), \mathbf{A} - \mathbf{C}\mathbf{B}^{-1}\mathbf{C}^T)$$

$$p(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{y}|\mathbf{b} + \mathbf{C}^T\mathbf{A}^{-1}(\mathbf{x} - \mathbf{a}), \mathbf{B} - \mathbf{C}^T\mathbf{A}^{-1}\mathbf{C}).$$

If the random variables \mathbf{x} and \mathbf{y} have Gaussian probability distributions

$$\begin{aligned} p(\mathbf{x}) &= \mathcal{N}(\mathbf{x}|\mathbf{m}, \mathbf{P}) \\ p(\mathbf{y}|\mathbf{x}) &= \mathcal{N}(\mathbf{y}|\mathbf{Hx} + \mathbf{u}, \mathbf{R}), \end{aligned}$$

then the joint distribution of \mathbf{x} and \mathbf{y} is Gaussian with

$$p\left(\begin{pmatrix}\mathbf{x} \\ \mathbf{y}\end{pmatrix}\right) = \mathcal{N}\left(\left(\begin{pmatrix}\mathbf{x} \\ \mathbf{y}\end{pmatrix} \middle| \left(\begin{matrix}\mathbf{m} \\ \mathbf{Hm} + \mathbf{u}\end{matrix}\right), \begin{pmatrix}\mathbf{P} & \mathbf{PH}^T \\ \mathbf{HP} & \mathbf{HPH}^T + \mathbf{R}\end{pmatrix}\right)\right).$$

Continuous Bayes filter

The Bayes filter extends to **continuous** state and evidence variables \mathbf{X}_t and \mathbf{E}_t .

The summations are replaced with integrals and the probability mass functions with probability densities, giving the recursive Bayesian relation

$$p(\mathbf{x}_{t+1} | \mathbf{e}_{1:t+1}) \propto p(\mathbf{e}_{t+1} | \mathbf{x}_{t+1}) \int p(\mathbf{x}_{t+1} | \mathbf{x}_t) p(\mathbf{x}_t | \mathbf{e}_{1:t}) d\mathbf{x}_t,$$

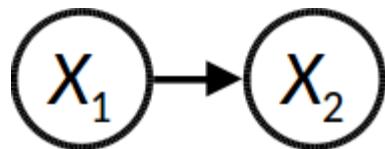
where the normalization constant is

$$Z = \int p(\mathbf{e}_{t+1} | \mathbf{x}_{t+1}) p(\mathbf{x}_{t+1} | \mathbf{e}_{1:t}) d\mathbf{x}_{t+1}.$$

Kalman filter

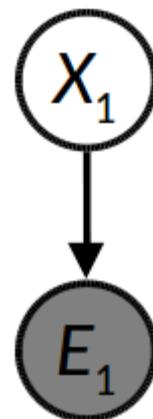
The **Kalman filter** is a special case of the Bayes filter, which assumes:

- Gaussian prior
- Linear Gaussian transition model
- Linear Gaussian sensor model



$$p(\mathbf{x}_{t+1} | \mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t+1} | \mathbf{A}\mathbf{x}_t + \mathbf{b}, \Sigma_{\mathbf{x}})$$

Transition model



$$p(\mathbf{e}_t | \mathbf{x}_t) = \mathcal{N}(\mathbf{e}_t | \mathbf{C}\mathbf{x}_t + \mathbf{d}, \Sigma_{\mathbf{e}})$$

Sensor model

Filtering Gaussian distributions

- *Prediction step:*

If the distribution $p(\mathbf{x}_t | \mathbf{e}_{1:t})$ is Gaussian and the transition model $p(\mathbf{x}_{t+1} | \mathbf{x}_t)$ is linear Gaussian, then the one-step predicted distribution given by

$$p(\mathbf{x}_{t+1} | \mathbf{e}_{1:t}) = \int p(\mathbf{x}_{t+1} | \mathbf{x}_t) p(\mathbf{x}_t | \mathbf{e}_{1:t}) d\mathbf{x}_t$$

is also a Gaussian distribution.

- *Update step:*

If the prediction $p(\mathbf{x}_{t+1} | \mathbf{e}_{1:t})$ is Gaussian and the sensor model $p(\mathbf{e}_{t+1} | \mathbf{x}_{t+1})$ is linear Gaussian, then after conditioning on new evidence, the updated distribution

$$p(\mathbf{x}_{t+1} | \mathbf{e}_{1:t+1}) \propto p(\mathbf{e}_{t+1} | \mathbf{x}_{t+1}) p(\mathbf{x}_{t+1} | \mathbf{e}_{1:t})$$

is also a Gaussian distribution.

Therefore, for the Kalman filter, $p(\mathbf{x}_t | \mathbf{e}_{1:t})$ is a multivariate Gaussian distribution $\mathcal{N}(\mathbf{x}_t | \boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t)$ for all t .

- Filtering reduces to the computation of the parameters $\boldsymbol{\mu}_t$ and $\boldsymbol{\Sigma}_t$.
- By contrast, for general (non-linear, non-Gaussian) processes, the description of the posterior grows **unboundedly** as $t \rightarrow \infty$.

1D example

Gaussian random walk:

- Gaussian prior:

$$p(x_0) = \mathcal{N}(x_0 | \mu_0, \sigma_0^2)$$

- The transition model adds random perturbations of constant variance:

$$p(x_{t+1} | x_t) = \mathcal{N}(x_{t+1} | x_t, \sigma_x^2)$$

- The sensor model yields measurements with Gaussian noise of constant variance:

$$p(e_t | x_t) = \mathcal{N}(e_t | x_t, \sigma_e^2)$$

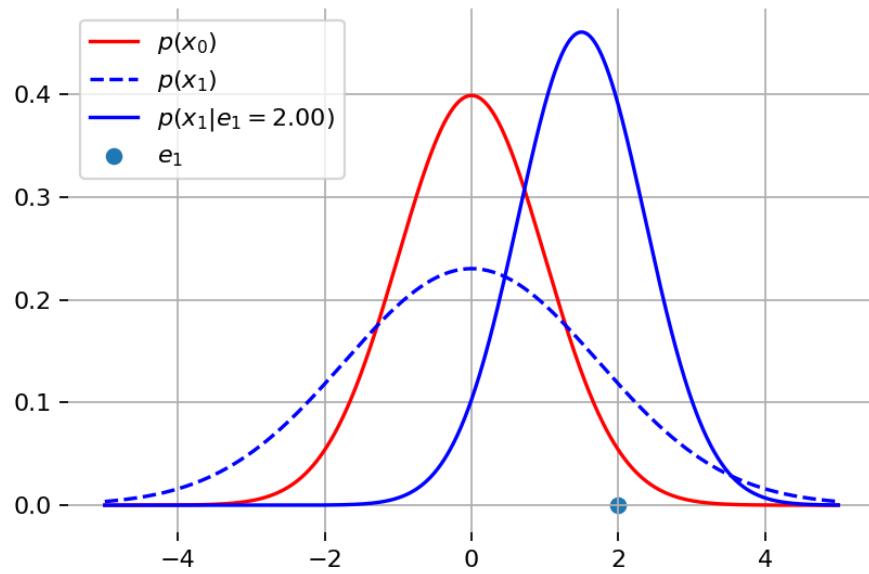
The one-step predicted distribution is given by

$$\begin{aligned} p(x_1) &= \int p(x_1|x_0)p(x_0)dx_0 \\ &\propto \int \exp\left(-\frac{1}{2}\frac{(x_1 - x_0)^2}{\sigma_x^2}\right) \exp\left(-\frac{1}{2}\frac{(x_0 - \mu_0)^2}{\sigma_0^2}\right) dx_0 \\ &\propto \int \exp\left(-\frac{1}{2}\frac{\sigma_0^2(x_1 - x_0)^2 + \sigma_x^2(x_0 - \mu_0)^2}{\sigma_0^2\sigma_x^2}\right) dx_0 \\ &\dots \text{ (simplify by completing the square)} \\ &\propto \exp\left(-\frac{1}{2}\frac{(x_1 - \mu_0)^2}{\sigma_0^2 + \sigma_x^2}\right) \\ &= \mathcal{N}(x_1|\mu_0, \sigma_0^2 + \sigma_x^2) \end{aligned}$$

Note that the same result can be obtained by using instead the Gaussian models identities.

For the update step, we need to condition on the observation at the first time step:

$$\begin{aligned}
 p(x_1|e_1) &\propto p(e_1|x_1)p(x_1) \\
 &\propto \exp\left(-\frac{1}{2}\frac{(e_1 - x_1)^2}{\sigma_e^2}\right) \exp\left(-\frac{1}{2}\frac{(x_1 - \mu_0)^2}{\sigma_0^2 + \sigma_x^2}\right) \\
 &\propto \exp\left(-\frac{1}{2} \frac{\left(x_1 - \frac{(\sigma_0^2 + \sigma_x^2)e_1 + \sigma_e^2\mu_0}{\sigma_0^2 + \sigma_x^2 + \sigma_e^2}\right)^2}{\frac{(\sigma_0^2 + \sigma_x^2)\sigma_e^2}{\sigma_0^2 + \sigma_x^2 + \sigma_e^2}}\right) \\
 &= \mathcal{N}\left(x_1 \middle| \frac{(\sigma_0^2 + \sigma_x^2)e_1 + \sigma_e^2\mu_0}{\sigma_0^2 + \sigma_x^2 + \sigma_e^2}, \frac{(\sigma_0^2 + \sigma_x^2)\sigma_e^2}{\sigma_0^2 + \sigma_x^2 + \sigma_e^2}\right)
 \end{aligned}$$



In summary, the update equations given a new evidence e_{t+1} are:

$$\mu_{t+1} = \frac{(\sigma_t^2 + \sigma_x^2)e_{t+1} + \sigma_e^2\mu_t}{\sigma_t^2 + \sigma_x^2 + \sigma_e^2}$$

$$\sigma_{t+1}^2 = \frac{(\sigma_t^2 + \sigma_x^2)\sigma_e^2}{\sigma_t^2 + \sigma_x^2 + \sigma_e^2}$$

General Kalman update

The same derivations generalize to multivariate normal distributions.

Assuming the transition and sensor models

$$\begin{aligned} p(\mathbf{x}_{t+1} | \mathbf{x}_t) &= \mathcal{N}(\mathbf{x}_{t+1} | \mathbf{F}\mathbf{x}_t, \boldsymbol{\Sigma}_x) \\ p(\mathbf{e}_t | \mathbf{x}_t) &= \mathcal{N}(\mathbf{e}_t | \mathbf{H}\mathbf{x}_t, \boldsymbol{\Sigma}_e), \end{aligned}$$

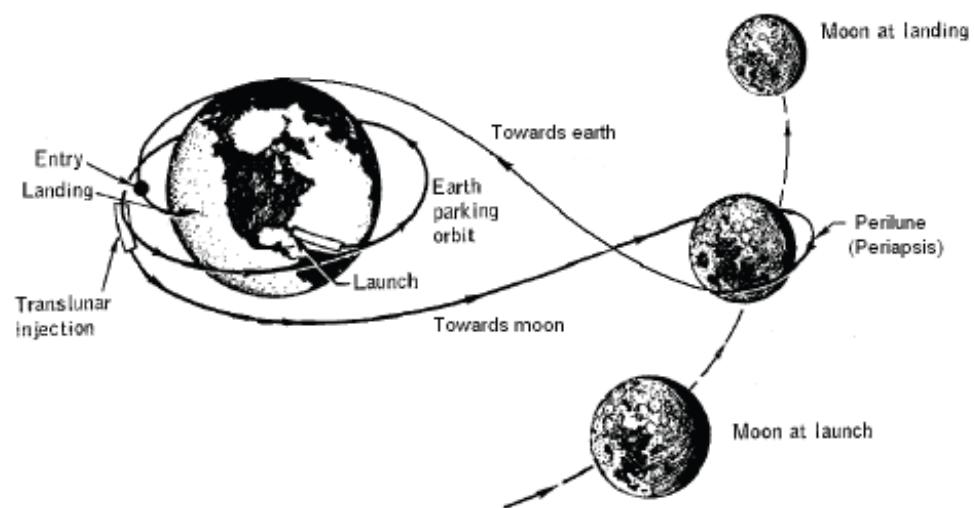
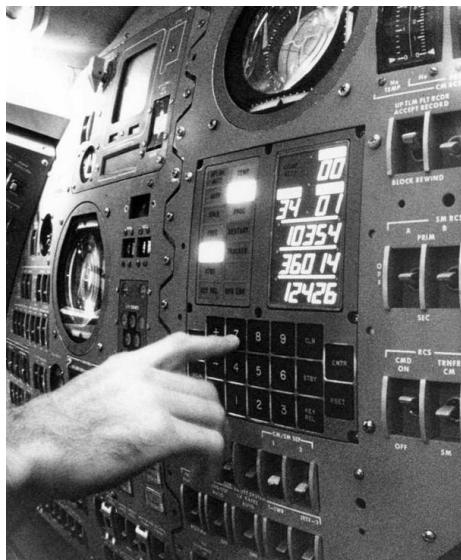
we arrive at the following general update equations:

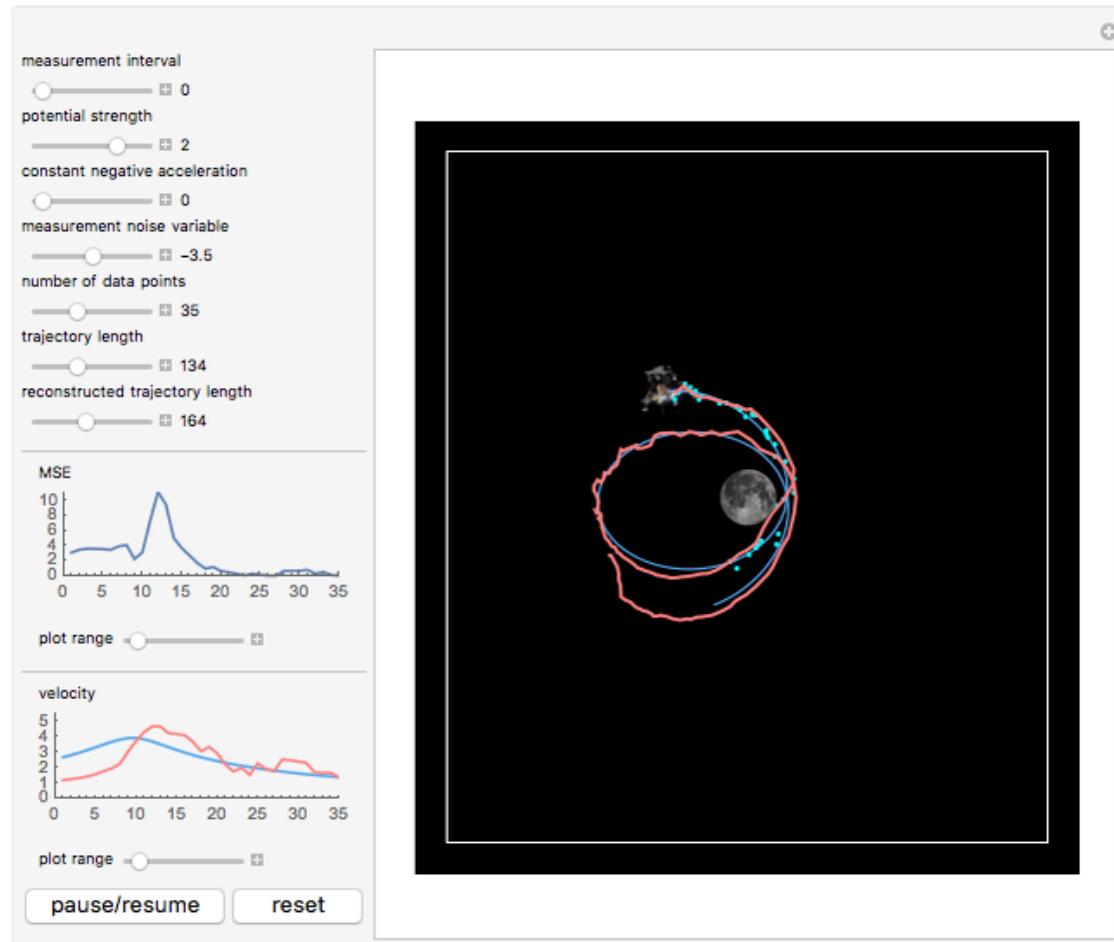
$$\begin{aligned} \mu_{t+1} &= \mathbf{F}\mu_t + \mathbf{K}_{t+1}(\mathbf{e}_{t+1} - \mathbf{H}\mathbf{F}\mu_t) \\ \boldsymbol{\Sigma}_{t+1} &= (\mathbf{I} - \mathbf{K}_{t+1}\mathbf{H})(\mathbf{F}\boldsymbol{\Sigma}_t\mathbf{F}^T + \boldsymbol{\Sigma}_x) \\ \mathbf{K}_{t+1} &= (\mathbf{F}\boldsymbol{\Sigma}_t\mathbf{F}^T + \boldsymbol{\Sigma}_x)\mathbf{H}^T(\mathbf{H}(\mathbf{F}\boldsymbol{\Sigma}_t\mathbf{F}^T + \boldsymbol{\Sigma}_x)\mathbf{H}^T + \boldsymbol{\Sigma}_e)^{-1} \end{aligned}$$

where \mathbf{K}_{t+1} is the Kalman gain matrix.

Apollo guidance computer

The Apollo Guidance Computer used a Kalman filter to estimate the position of the spacecraft. The Kalman filter was used to merge new data with past position measurements to produce an optimal position estimate of the spacecraft.





Demo: tracking an object in space using the Kalman Filter.

Data assimilation for weather forecasts

In weather forecasting, filtering is used to combine observations of the atmosphere with numerical models to estimate its current state. This is called **data assimilation**.

Then, the model is used to predict the future states of the atmosphere.



20 YEARS OF 4DVAR



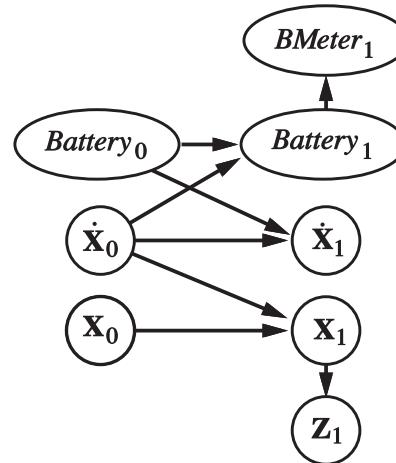
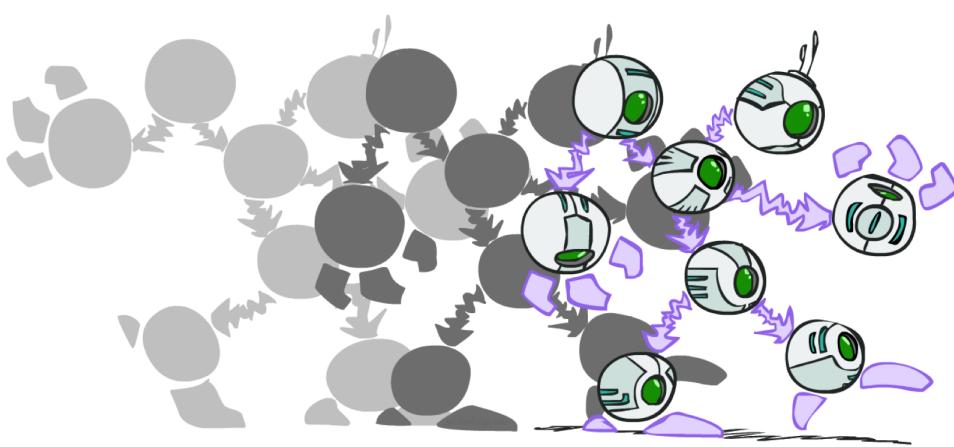
Watch later



Share



Dynamic Bayesian networks

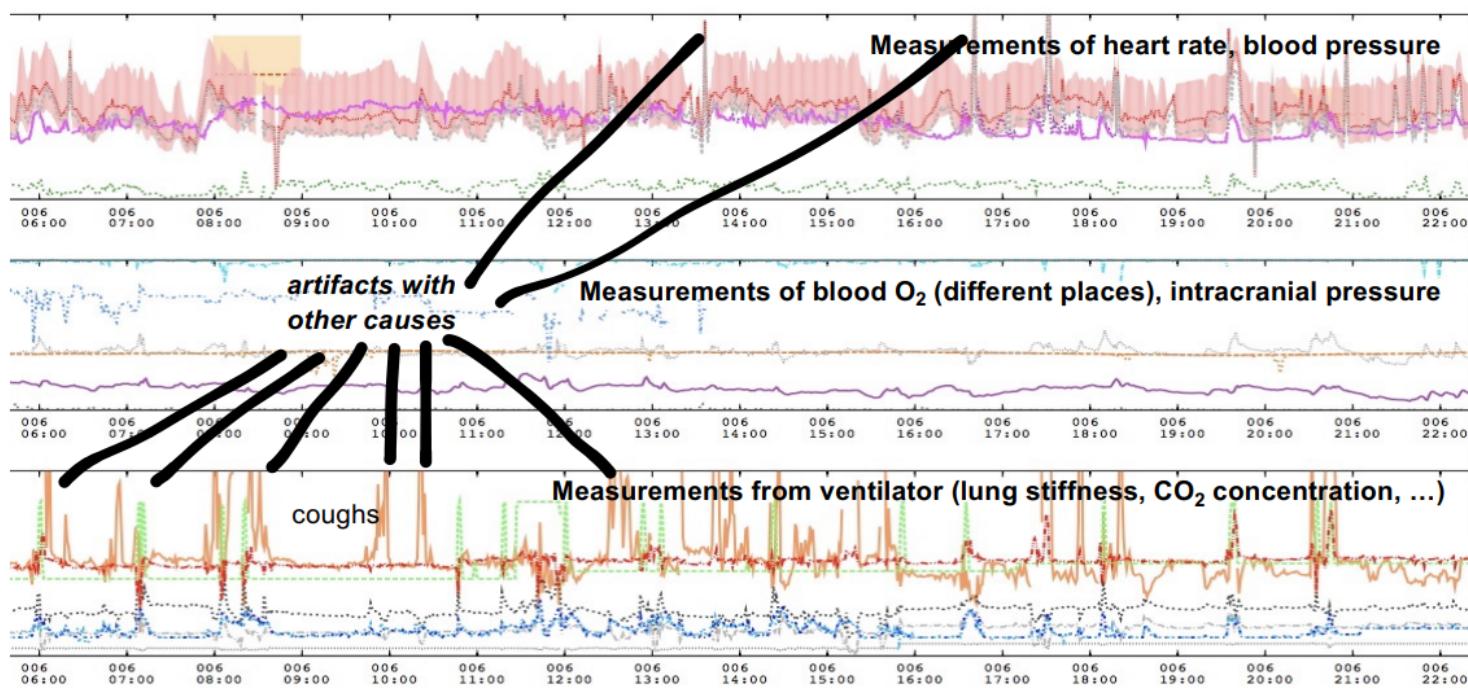


Dynamics Bayesian networks (DBNs) can be used for tracking multiple variables over time, using multiple sources of evidence. Idea:

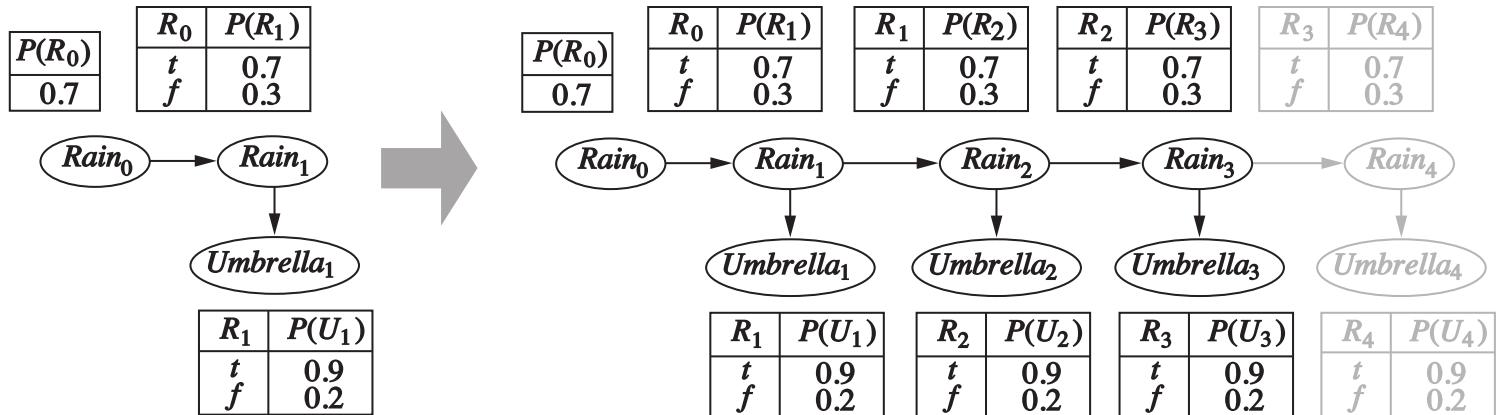
- Repeat a fixed Bayes net structure at each time t .
- Variables from time t condition on those from $t - 1$.

DBNs are a generalization of HMMs and of the Kalman filter.

Application: ICU monitoring



Exact inference



Unroll the network through time and run any exact inference algorithm (e.g., variable elimination)

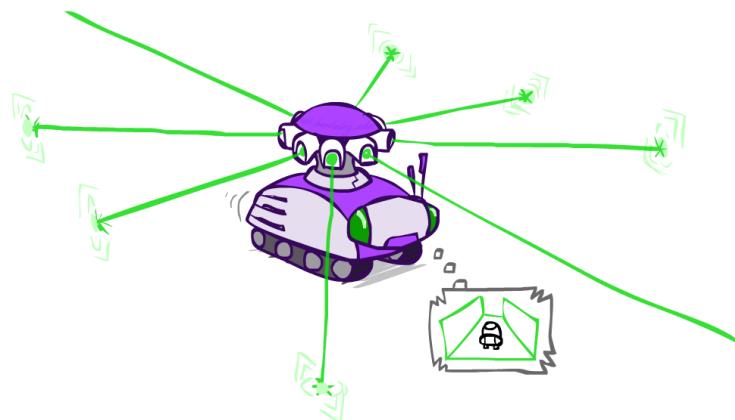
- Problem: inference cost for each update grows with t .
- Rollup filtering: add slice $t + 1$, sum out slice t using variable elimination.
 - Largest factor is $O(d^{n+k})$ and the total update cost per step is $O(nd^{n+k})$.
 - Better than HMMs, which is $O(d^{2n})$, but still **infeasible** for large numbers of variables.

Particle filter

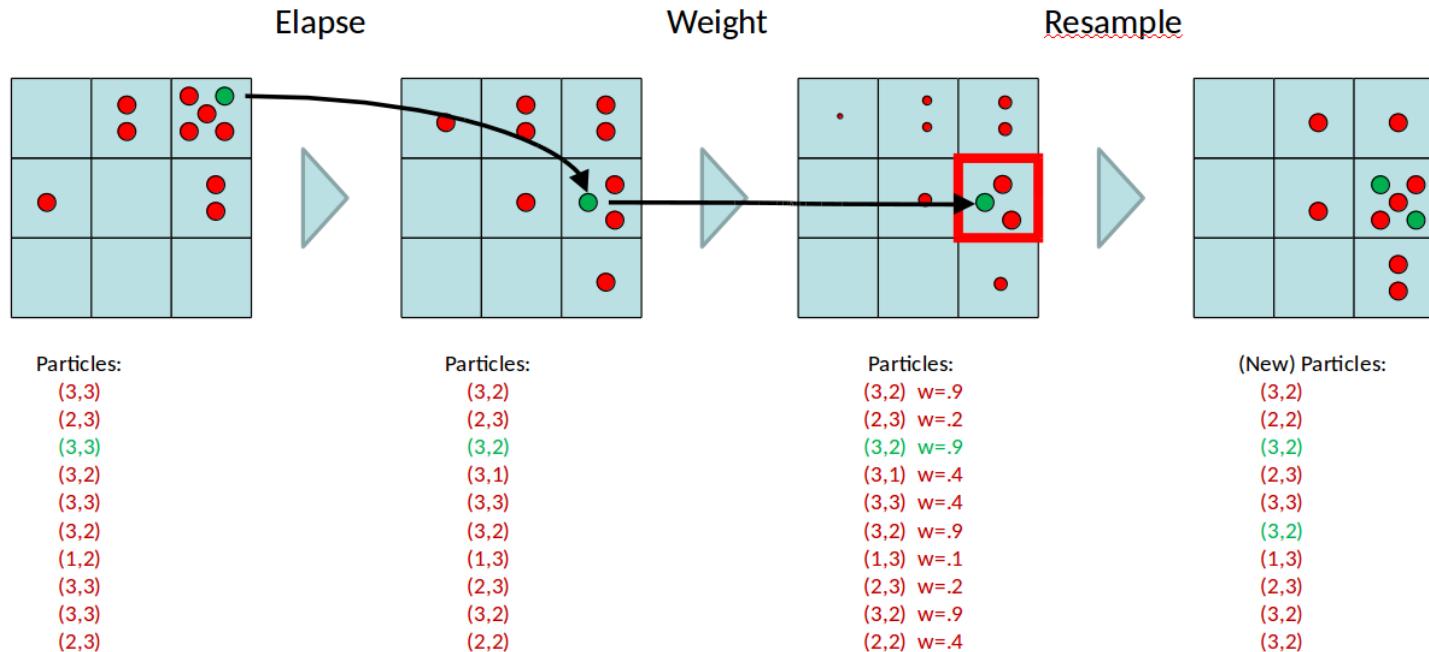
Basic idea:

- Maintain a finite population of samples, called **particles**.
 - The representation of our beliefs is a list of N particles.
- Ensure the particles track the high-likelihood regions of the state space.
- Throw away samples that have very low weight, according to the evidence.
- Replicate those that have high weight.

This scales to high dimensions!



Update cycle



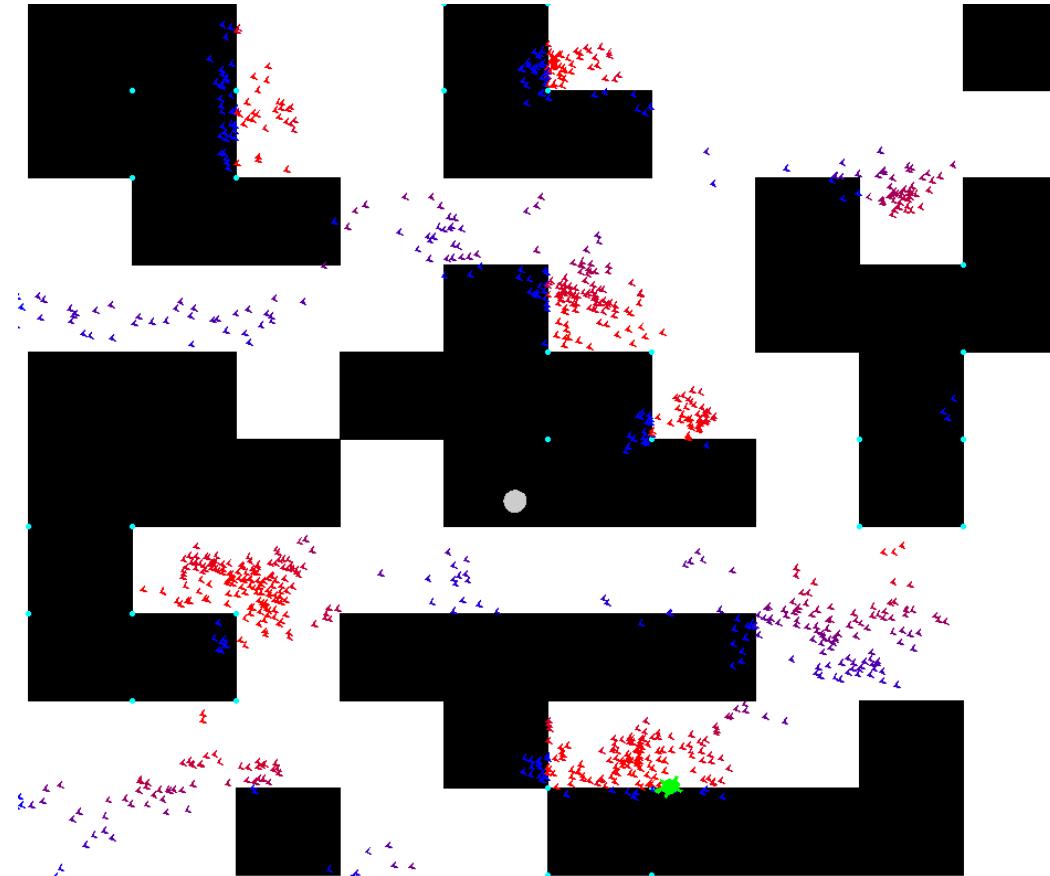
```

function PARTICLE-FILTERING(e,  $N$ ,  $dbn$ ) returns a set of samples for the next time step
  inputs: e, the new incoming evidence
     $N$ , the number of samples to be maintained
     $dbn$ , a DBN with prior  $\mathbf{P}(\mathbf{X}_0)$ , transition model  $\mathbf{P}(\mathbf{X}_1|\mathbf{X}_0)$ , sensor model  $\mathbf{P}(\mathbf{E}_1|\mathbf{X}_1)$ 
  persistent:  $S$ , a vector of samples of size  $N$ , initially generated from  $\mathbf{P}(\mathbf{X}_0)$ 
  local variables:  $W$ , a vector of weights of size  $N$ 

  for  $i = 1$  to  $N$  do
     $S[i] \leftarrow$  sample from  $\mathbf{P}(\mathbf{X}_1 \mid \mathbf{X}_0 = S[i])$  /* step 1 */
     $W[i] \leftarrow \mathbf{P}(\mathbf{e} \mid \mathbf{X}_1 = S[i])$  /* step 2 */
     $S \leftarrow$  WEIGHTED-SAMPLE-WITH-REPLACEMENT( $N$ ,  $S$ ,  $W$ ) /* step 3 */
  return  $S$ 

```

Robot localization



(See demo)

Summary

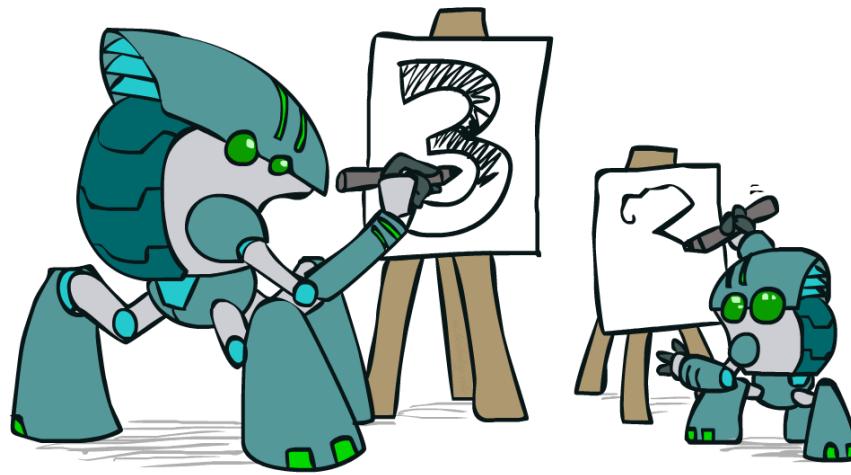
- Temporal models use state and sensor variables replicated over time.
 - Their purpose is to maintain a belief state as time passes and as more evidence is collected.
- The Markov and stationarity assumptions imply that we only need to specify
 - a transition model $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{X}_t)$,
 - a sensor model $\mathbf{P}(\mathbf{E}_t|\mathbf{X}_t)$.
- Inference tasks include filtering, prediction, smoothing and finding the most likely sequence.
- Filter algorithms are all based on the core of idea of
 - projecting the current belief state through the transition model,
 - updating the prediction according to the new evidence.

Introduction to Artificial Intelligence

Lecture 7: Machine learning and neural networks

Prof. Gilles Louppe
g.louppe@uliege.be

Today



Learning from data is a key component of artificial intelligence. In this lecture, we will introduce the principles of:

- Machine learning
- Neural networks

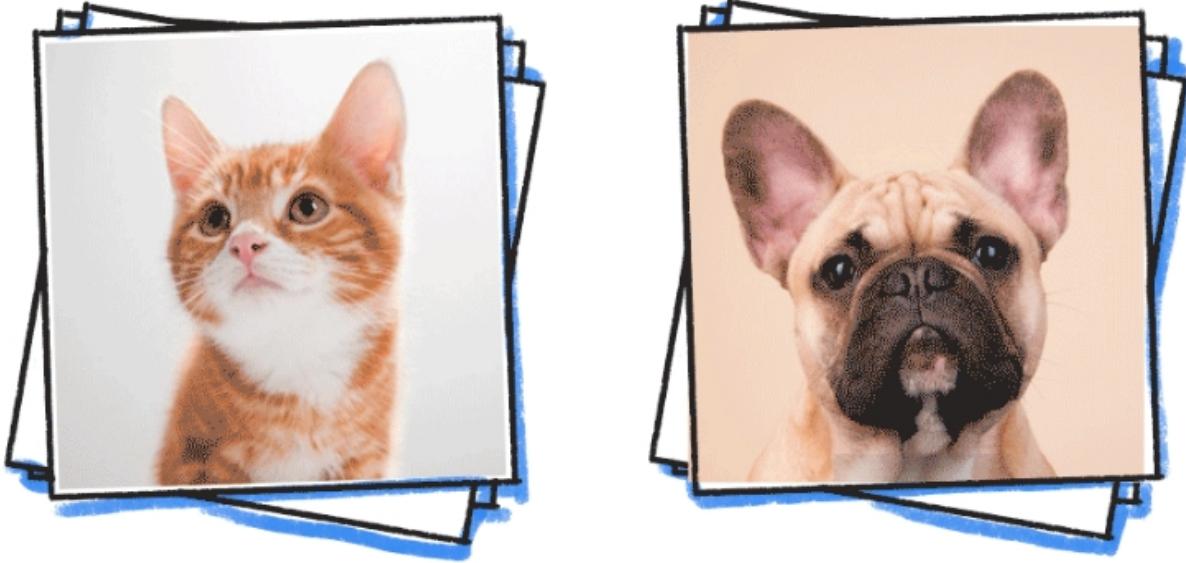
Learning agents

What if the environment is **unknown**?

- Learning provides an automated way to modify the agent's internal decision mechanisms to improve its own performance.
- It exposes the agent to reality rather than trying to hardcode reality into the agent's program.

More generally, learning is useful for any task where it is difficult to write a program that performs the task but easy to obtain examples of desired behavior.

Machine learning

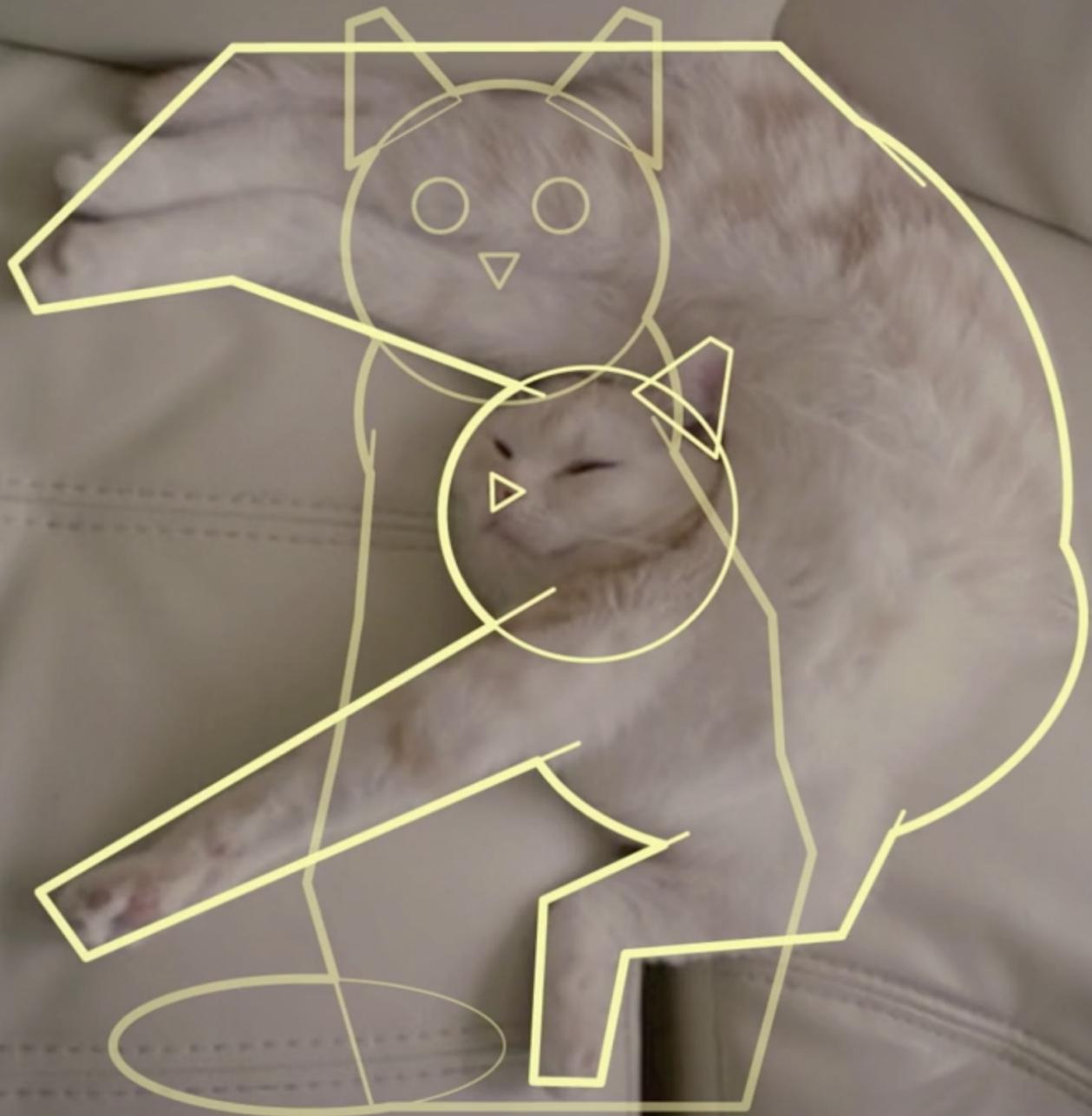


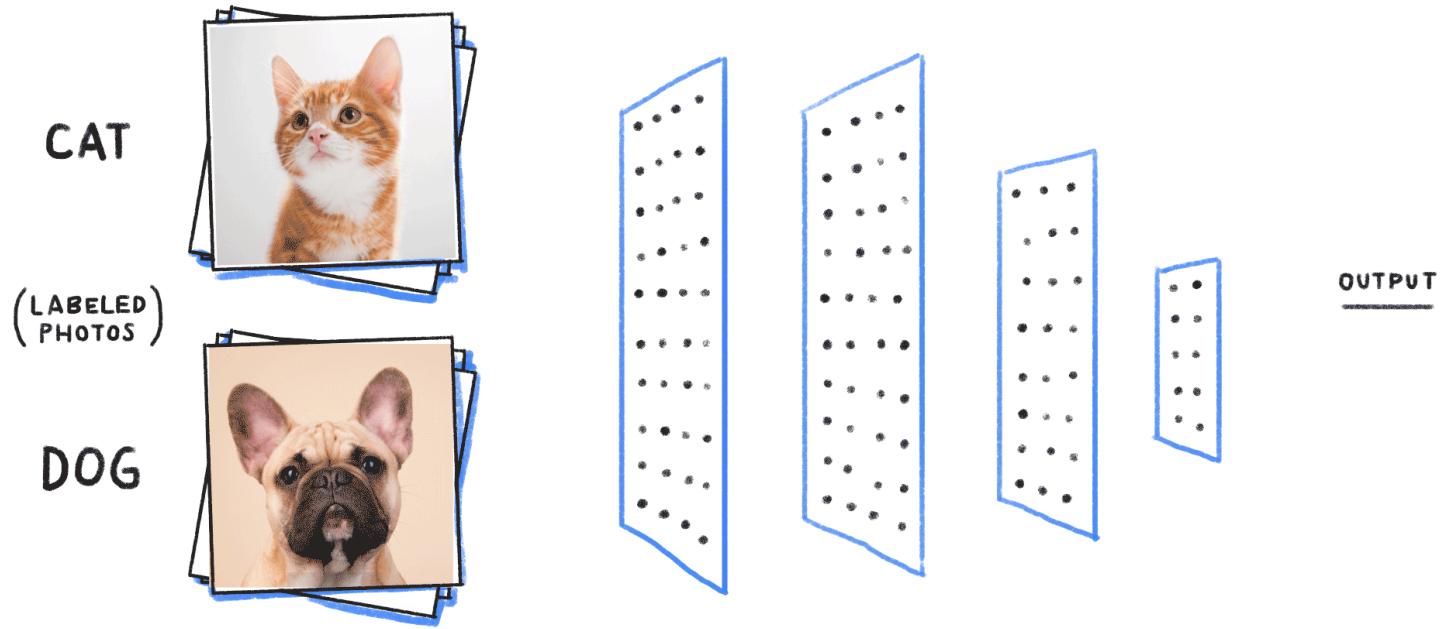
How would you write a computer program that recognizes cats from dogs?











The deep learning approach.

Problem statement

Let $\mathbf{d} \sim p(\mathbf{x}, y)$ be a dataset of N example input-output pairs

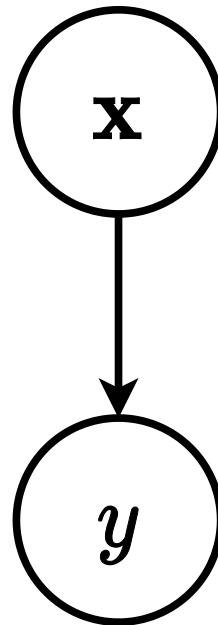
$$\mathbf{d} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_N, y_N)\},$$

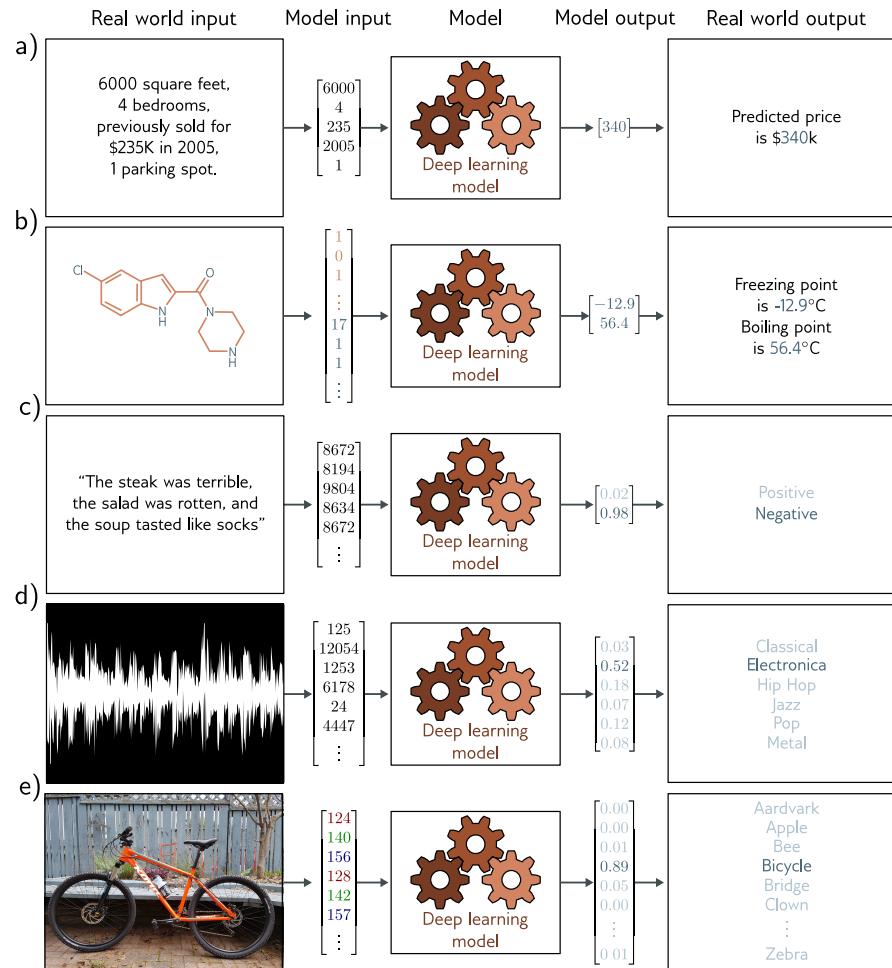
where $\mathbf{x}_i \in \mathbb{R}^d$ are d -dimensional vectors representing the input values and $y_i \in \mathcal{Y}$ are the corresponding output values.

From this data, we want to identify a probabilistic model

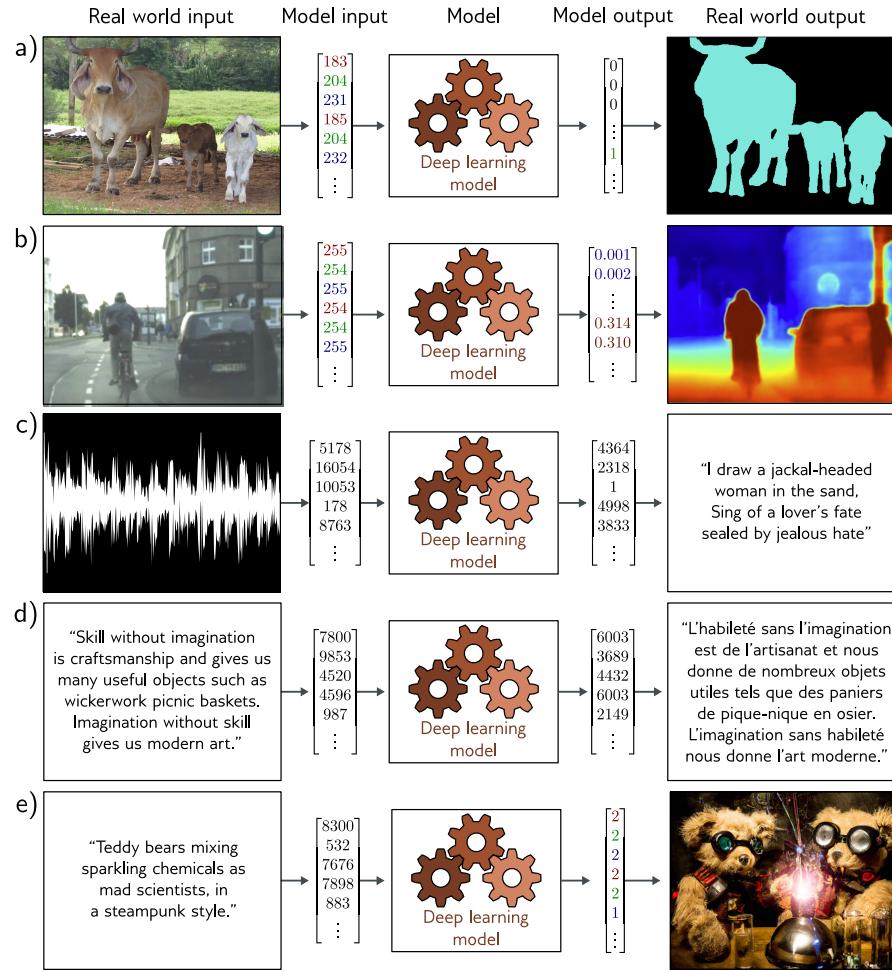
$$p_\theta(y|\mathbf{x})$$

that best explains the data.





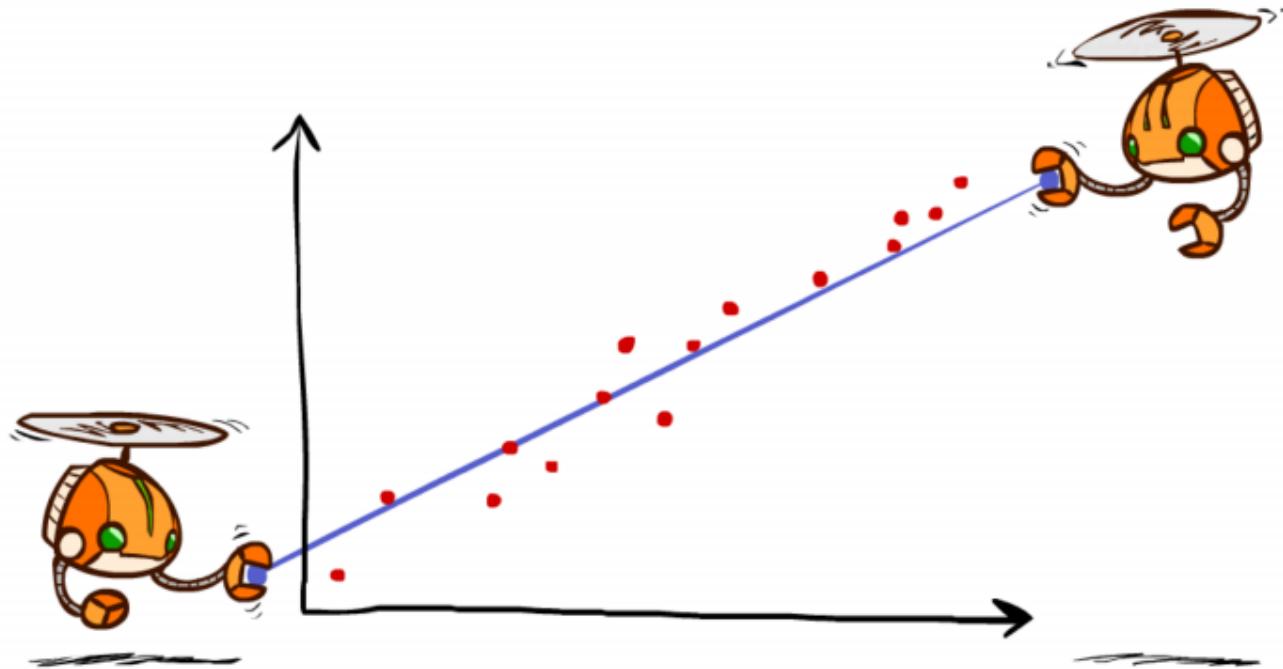
Regression ($y \in \mathbb{R}$) and classification ($y \in \{0, 1, \dots, C - 1\}$) problems.

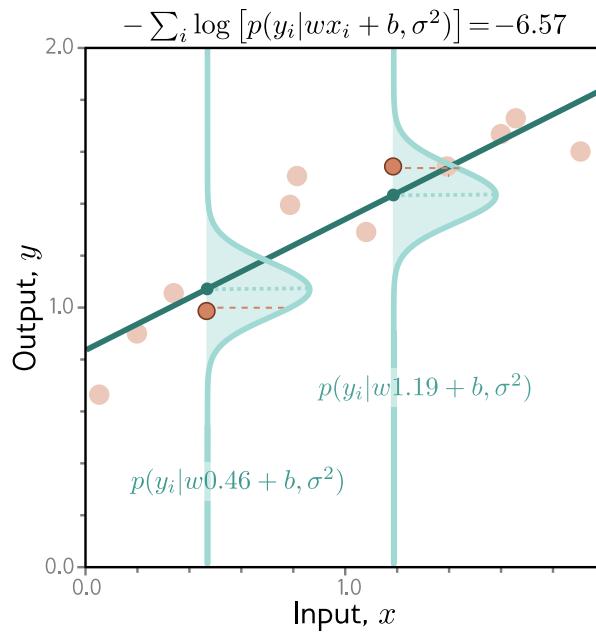
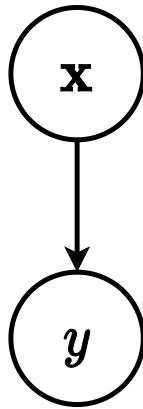


Supervised learning with structured outputs ($y \in \mathcal{Y}$).

Linear regression

Let us first assume that $y \in \mathbb{R}$.





Linear regression considers a parameterized linear Gaussian model for its parametric model of $p(y|\mathbf{x})$, that is

$$p(y|\mathbf{x}) = \mathcal{N}(y|\mathbf{w}^T \mathbf{x} + b, \sigma^2),$$

where \mathbf{w} and b are parameters to determine.

To learn the conditional distribution $p(y|\mathbf{x})$, we maximize

$$p(y|\mathbf{x}) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2} \frac{(y - (\mathbf{w}^T \mathbf{x} + b))^2}{\sigma^2}\right)$$

w.r.t. \mathbf{w} and b over the data $\mathbf{d} = \{(\mathbf{x}_j, y_j)\}$.

To learn the conditional distribution $p(y|\mathbf{x})$, we maximize

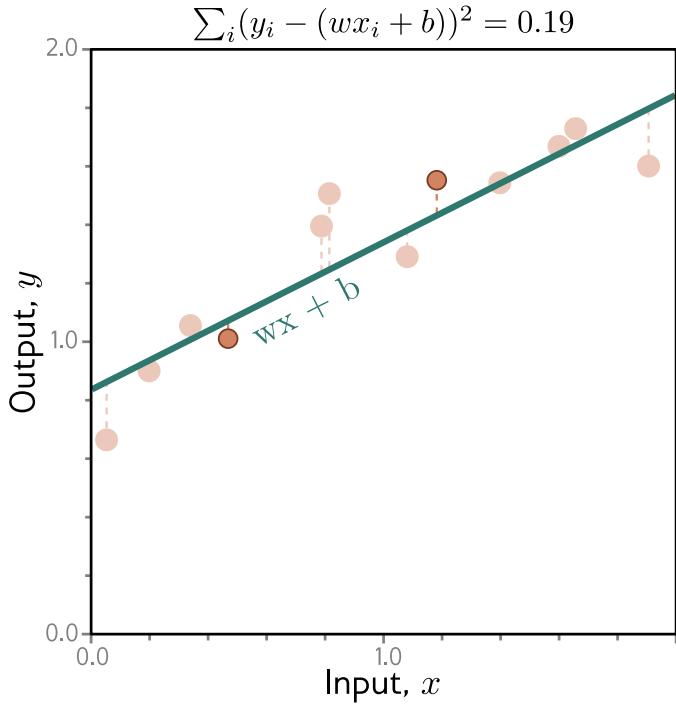
$$p(y|\mathbf{x}) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2} \frac{(y - (\mathbf{w}^T \mathbf{x} + b))^2}{\sigma^2}\right)$$

w.r.t. \mathbf{w} and b over the data $\mathbf{d} = \{(\mathbf{x}_j, y_j)\}$.

By constraining the derivatives of the log-likelihood to $\mathbf{0}$, we arrive to the problem of minimizing

$$\mathcal{L}(\mathbf{w}, b) = \sum_{j=1}^N (y_j - (\mathbf{w}^T \mathbf{x}_j + b))^2.$$

Therefore, minimizing the sum of squared errors corresponds to the MLE solution for a linear fit, assuming Gaussian noise of fixed variance.



Minimizing the negative log-likelihood of a linear Gaussian model reduces to minimizing the sum of squared residuals.

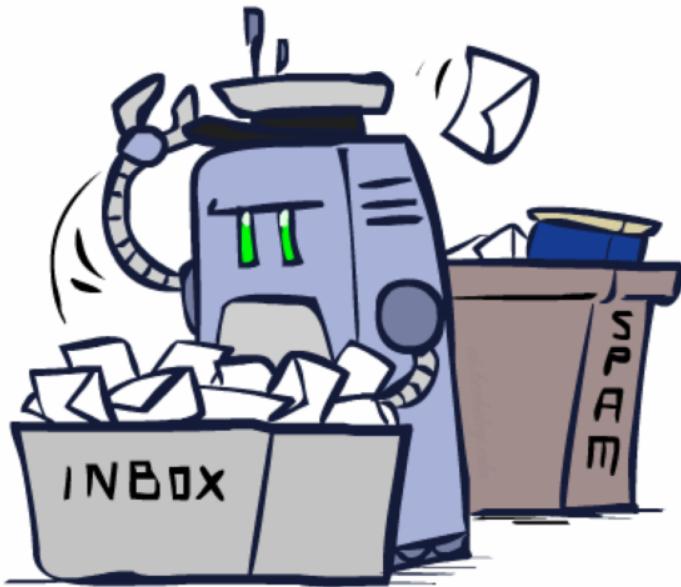
If we absorb the bias term b into the weight vector \mathbf{w} by adding a constant feature $x_0 = 1$ to the input vector \mathbf{x} , the solution \mathbf{w}^* is given analytically by

$$\mathbf{w}^* = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y},$$

where \mathbf{X} is the input matrix made of the stacked input vectors \mathbf{x}_j (including the constant feature) and \mathbf{y} is the output vector made of the output values y_j .

Logistic regression

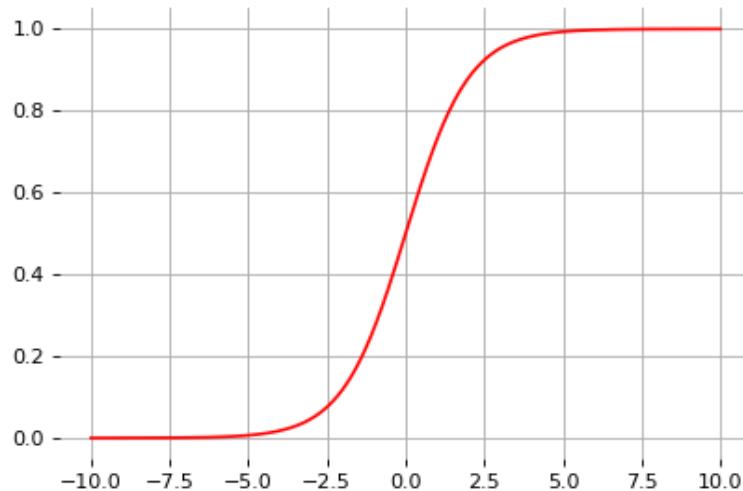
Let us now assume $y \in \{0, 1\}$.



Logistic regression models the conditional as

$$P(Y = 1|\mathbf{x}) = \sigma(\mathbf{w}^T \mathbf{x} + b),$$

where the sigmoid activation function $\sigma(x) = \frac{1}{1+\exp(-x)}$ looks like a soft heavyside:



Following the principle of maximum likelihood estimation, we have

$$\begin{aligned}
 & \arg \max_{\mathbf{w}, b} P(\mathbf{d} | \mathbf{w}, b) \\
 &= \arg \max_{\mathbf{w}, b} \prod_{\mathbf{x}_i, y_i \in \mathbf{d}} P(Y = y_i | \mathbf{x}_i, \mathbf{w}, b) \\
 &= \arg \max_{\mathbf{w}, b} \prod_{\mathbf{x}_i, y_i \in \mathbf{d}} \sigma(\mathbf{w}^T \mathbf{x}_i + b)^{y_i} (1 - \sigma(\mathbf{w}^T \mathbf{x}_i + b))^{1-y_i} \\
 &= \arg \min_{\mathbf{w}, b} \underbrace{\sum_{\mathbf{x}_i, y_i \in \mathbf{d}} -y_i \log \sigma(\mathbf{w}^T \mathbf{x}_i + b) - (1 - y_i) \log(1 - \sigma(\mathbf{w}^T \mathbf{x}_i + b))}_{\mathcal{L}(\mathbf{w}, b) = \sum_i \ell(y_i, \hat{y}(\mathbf{x}_i; \mathbf{w}, b))}
 \end{aligned}$$

This loss is an estimator of the **cross-entropy**

$$H(p, q) = \mathbb{E}_p[-\log q]$$

for $p = Y | \mathbf{x}_i$ and $q = \hat{Y} | \mathbf{x}_i$.

Unfortunately, there is no closed-form solution for the MLE of \mathbf{w} and b .

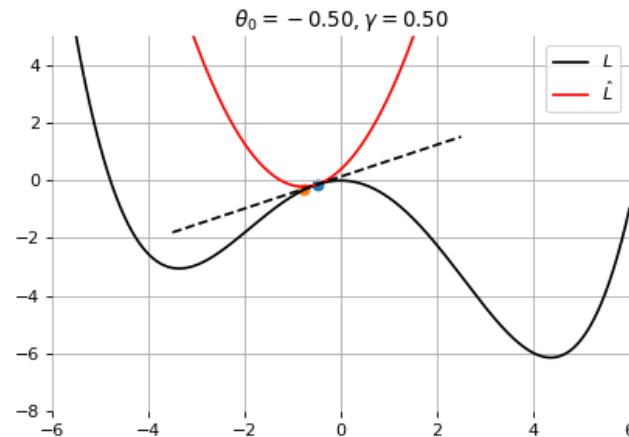
Gradient descent

Let $\mathcal{L}(\theta)$ denote a loss function defined over model parameters θ (e.g., \mathbf{w} and b).

To minimize $\mathcal{L}(\theta)$, gradient descent uses local linear information to iteratively move towards a (local) minimum.

For θ_0 , a first-order approximation around θ_0 can be defined as

$$\hat{\mathcal{L}}(\epsilon; \theta_0) = \mathcal{L}(\theta_0) + \epsilon^T \nabla_{\theta} \mathcal{L}(\theta_0) + \frac{1}{2\gamma} \|\epsilon\|^2.$$



A minimizer of the approximation $\hat{\mathcal{L}}(\epsilon; \theta_0)$ is given for

$$\begin{aligned}\nabla_\epsilon \hat{\mathcal{L}}(\epsilon; \theta_0) &= 0 \\ &= \nabla_\theta \mathcal{L}(\theta_0) + \frac{1}{\gamma} \epsilon,\end{aligned}$$

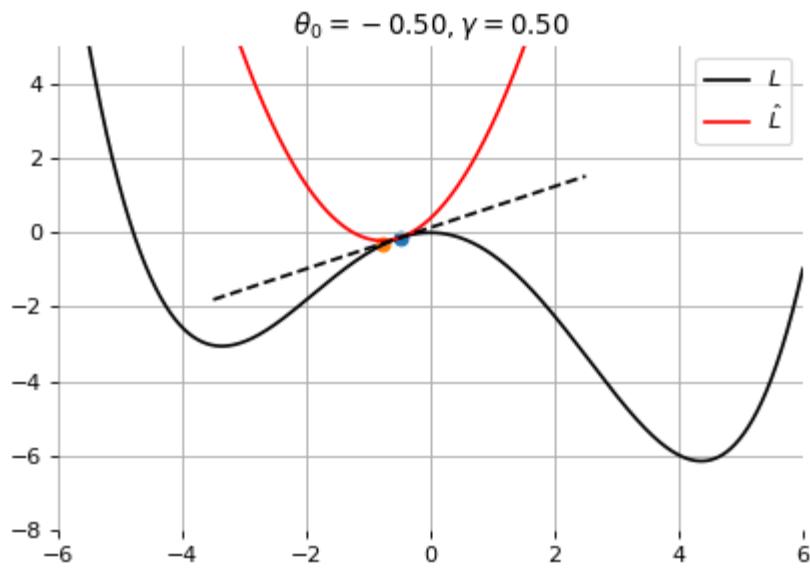
which results in the best improvement for the step $\epsilon = -\gamma \nabla_\theta \mathcal{L}(\theta_0)$.

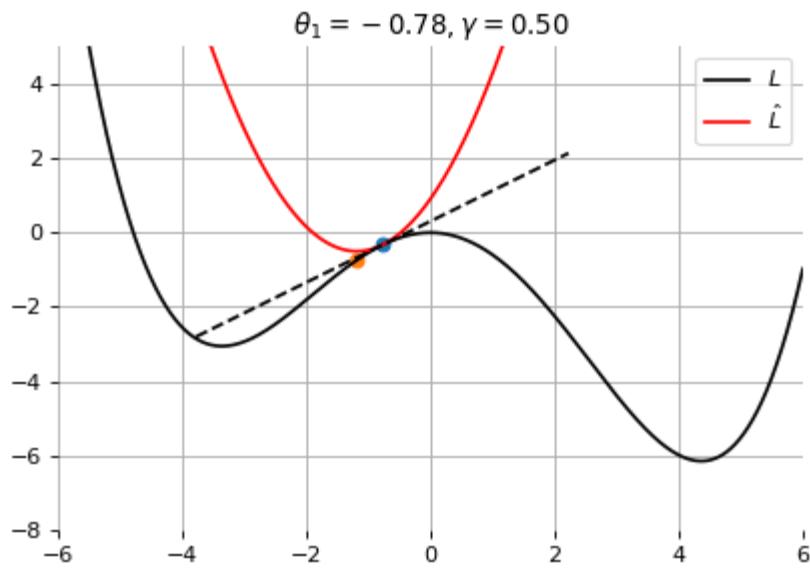
Therefore, model parameters can be updated iteratively using the update rule

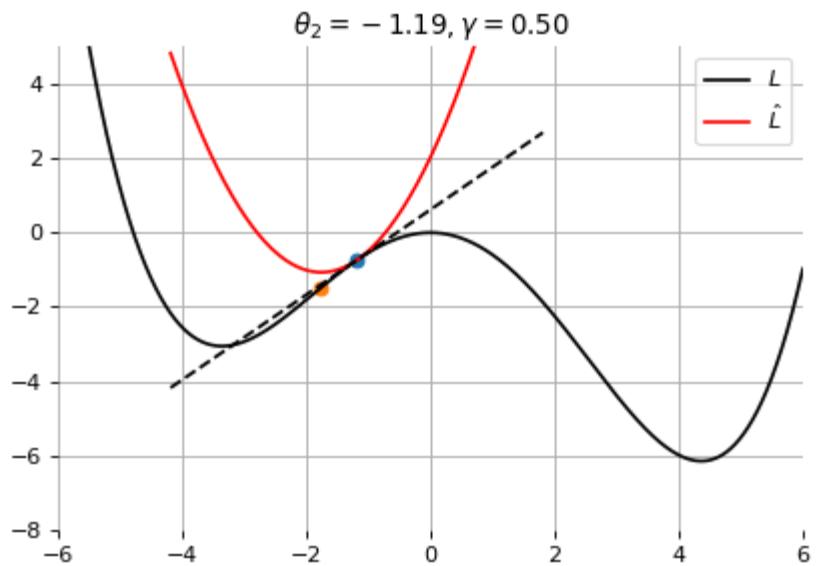
$$\theta_{t+1} = \theta_t - \gamma \nabla_\theta \mathcal{L}(\theta_t),$$

where

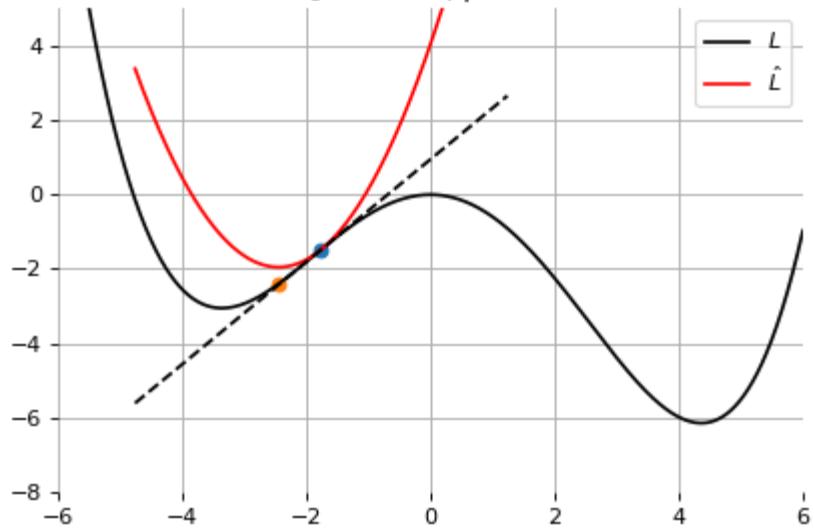
- θ_0 are the initial parameters of the model,
- γ is the learning rate.



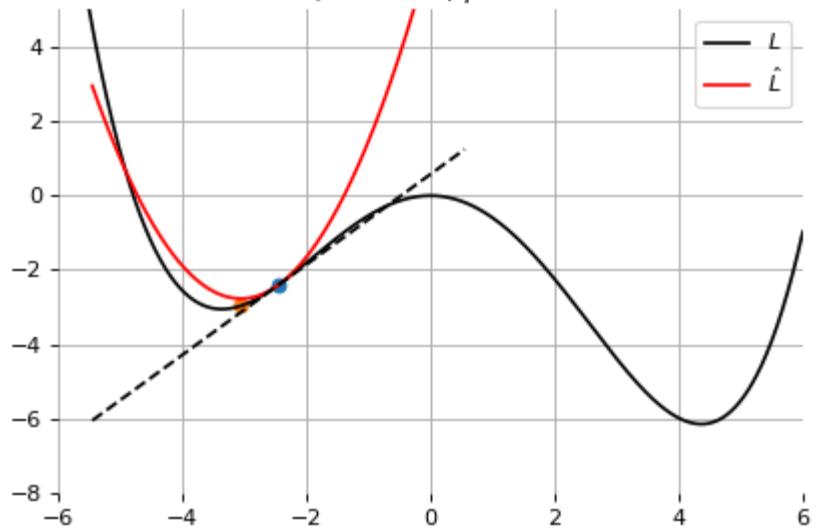


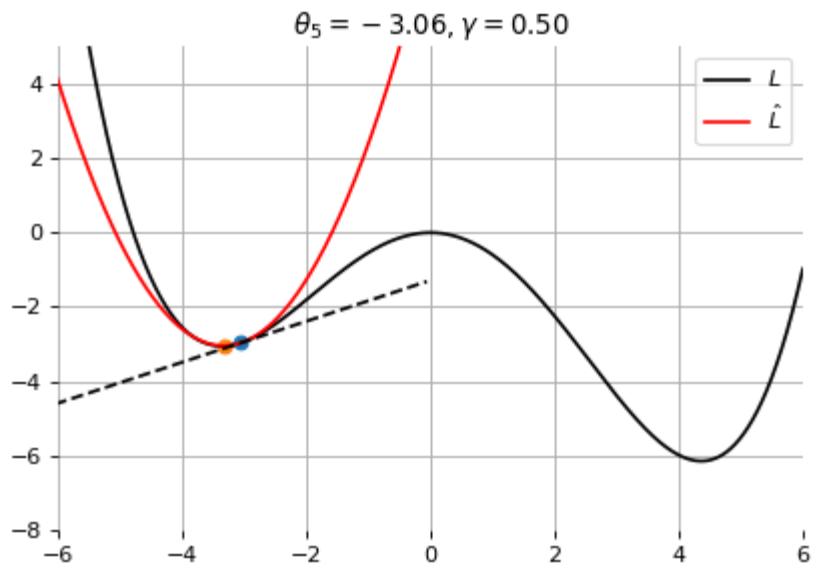


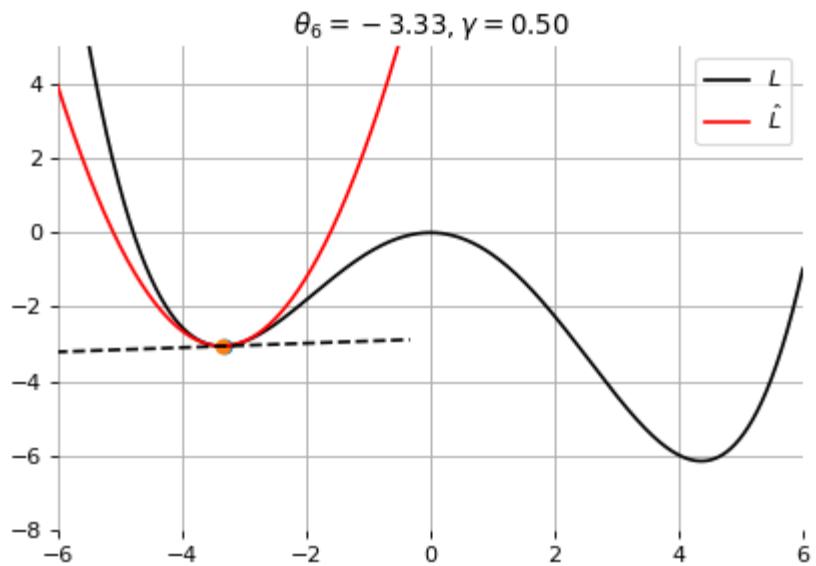
$$\theta_3 = -1.76, \gamma = 0.50$$



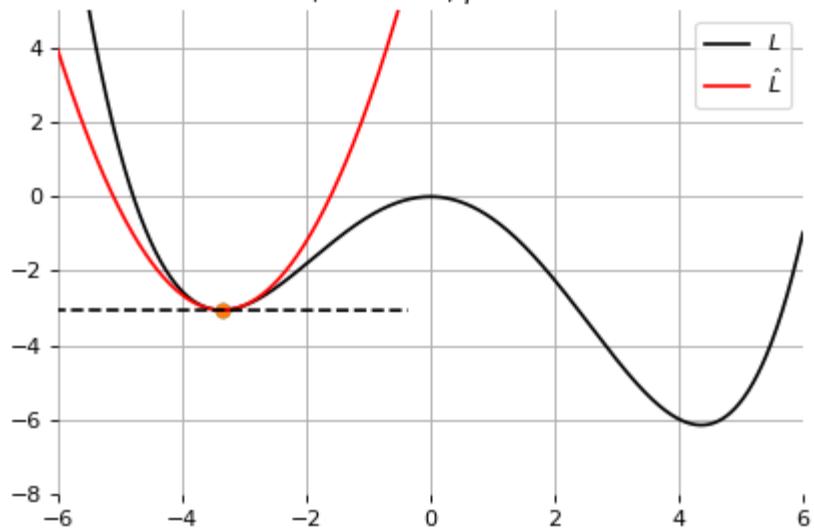
$$\theta_4 = -2.45, \gamma = 0.50$$







$$\theta_7 = -3.36, \gamma = 0.50$$

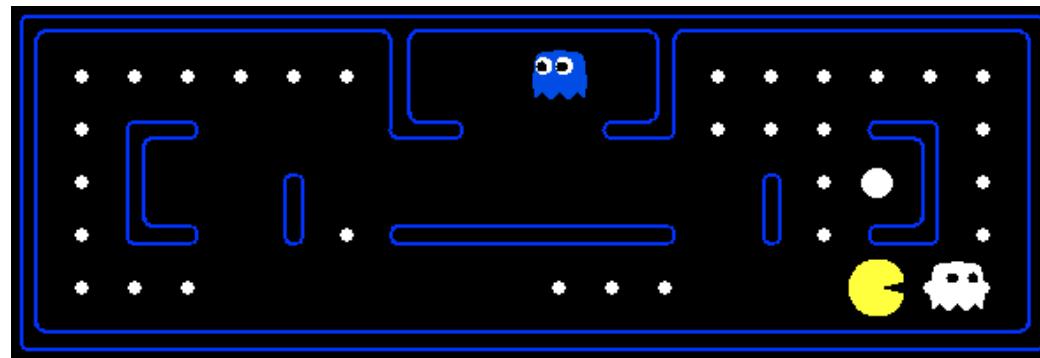


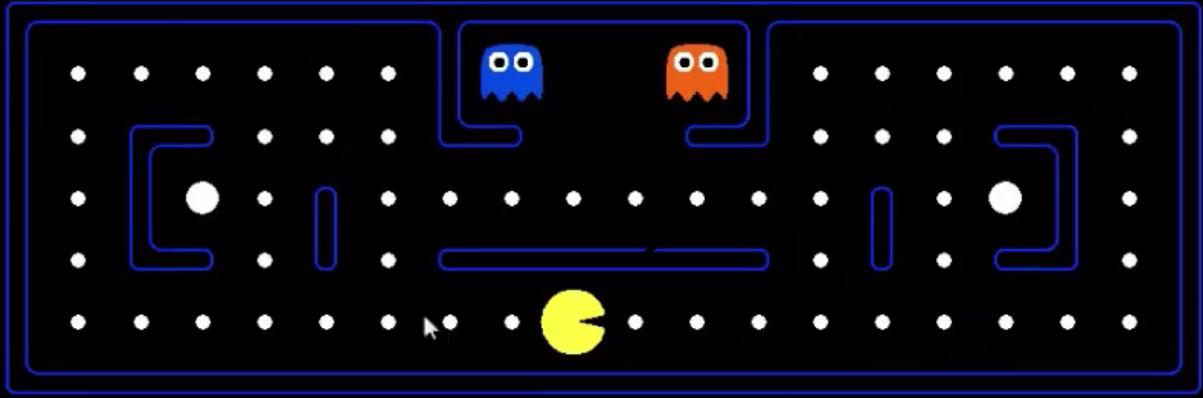
(Step-by-step code example)

Example: imitation learning in Pacman

Can we learn to play Pacman only from observations?

- Feature vectors $\mathbf{x} = g(\mathbf{s})$ are extracted from the game states \mathbf{s} . Output values \mathbf{y} corresponds to actions a .
- State-action pairs (\mathbf{x}, \mathbf{y}) are collected by observing an expert playing.
- We want to learn the actions that the expert would take in a given situation. That is, learn the mapping $f : \mathbb{R}^d \rightarrow \mathcal{A}$.
- This is a multiclass classification problem that can be solved by combining binary classifiers.



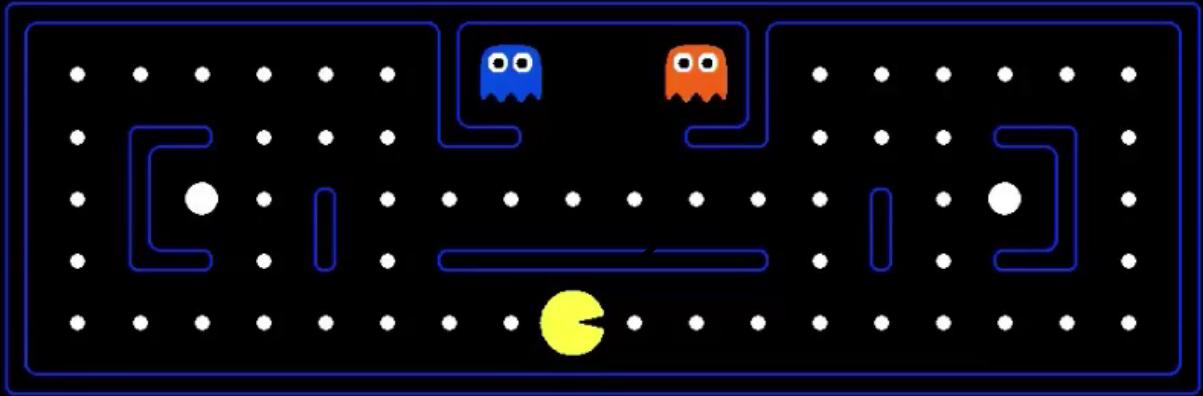


SCORE: 0

▶ 0:00 / 0:18



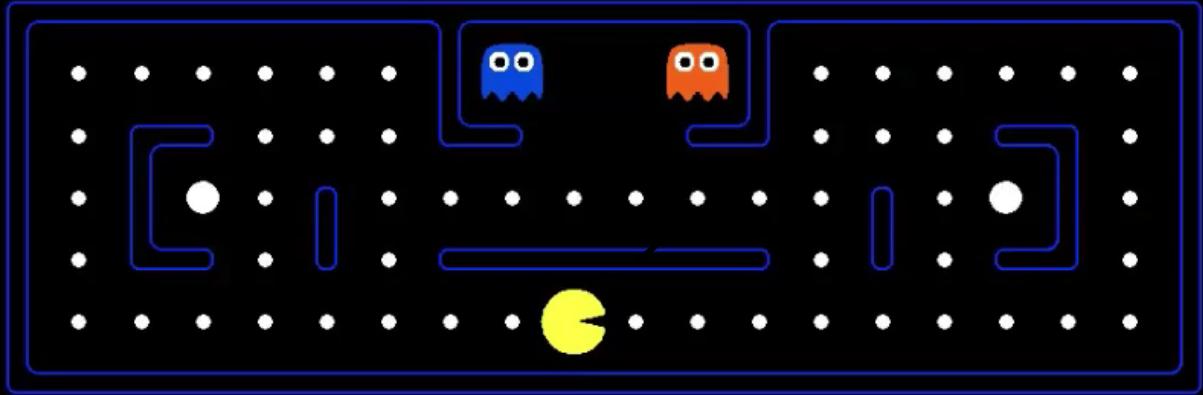
The agent observes a very good Minimax-based agent for two games and updates its weight vectors as data are collected.



SCORE: 0

► 0:00 / 0:18





After two training episodes, the ML-based agents plays.
No more Minimax!

Deep Learning

(a short introduction)

Shallow networks

A shallow network is a function

$$f : \mathbb{R}^{d_{\text{in}}} \rightarrow \mathbb{R}^{d_{\text{out}}}$$

that maps multi-dimensional inputs \mathbf{x} to multi-dimensional outputs \mathbf{y} through a hidden layer $\mathbf{h} = [h_0, h_1, \dots, h_{q-1}] \in \mathbb{R}^q$, such that

$$\begin{aligned} h_j &= \sigma \left(\sum_{i=0}^{d_{\text{in}}-1} w_{ji} x_i + b_j \right) \\ y_k &= \sum_{j=0}^{q-1} v_{kj} h_j + c_k, \end{aligned}$$

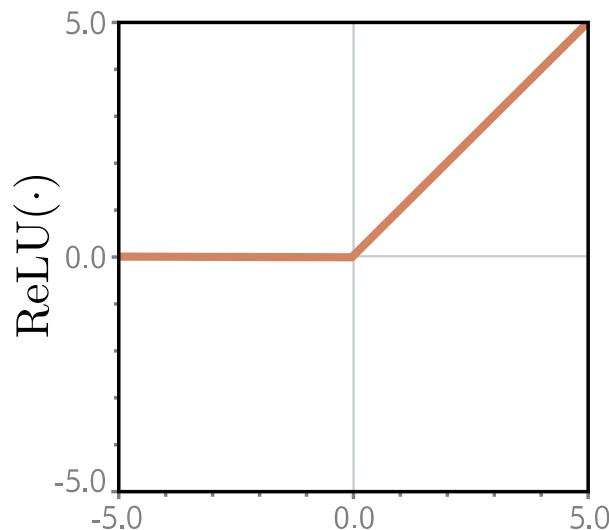
where w_{ji}, b_j, v_{kj} and c_k ($i = 0, \dots, d_{\text{in}} - 1, j = 0, \dots, q - 1, k = 0, \dots, d_{\text{out}} - 1$) are the model parameters and σ is an activation function.

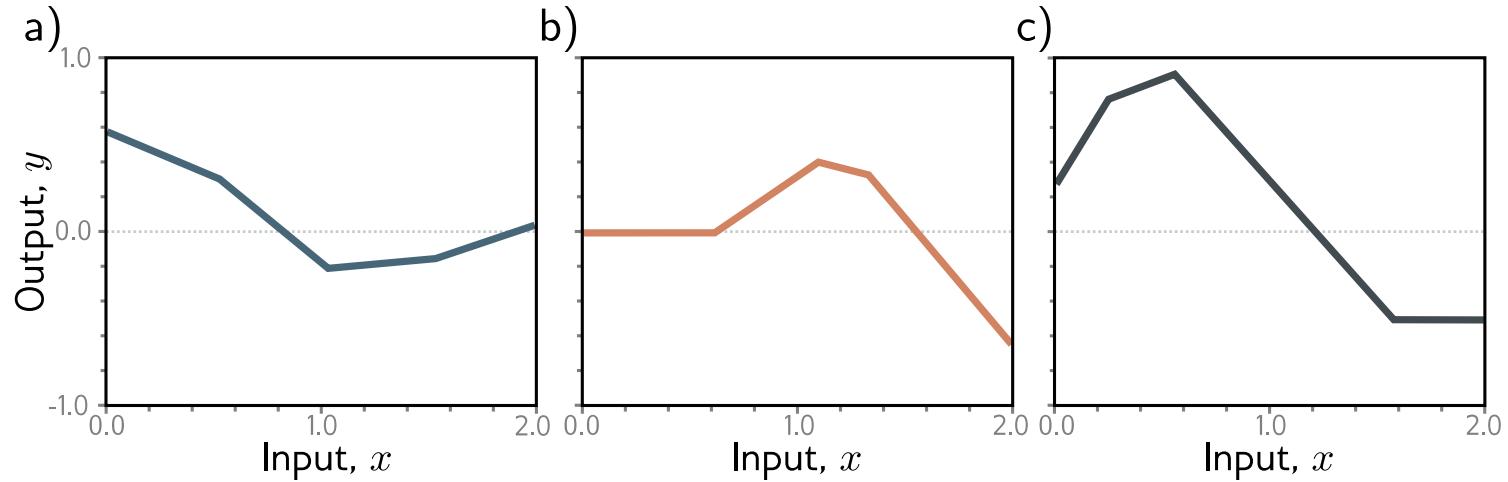
Single-input single-output networks

We first consider the case where $d_{\text{in}} = 1$ and $d_{\text{out}} = 1$ for the single-input single-output network

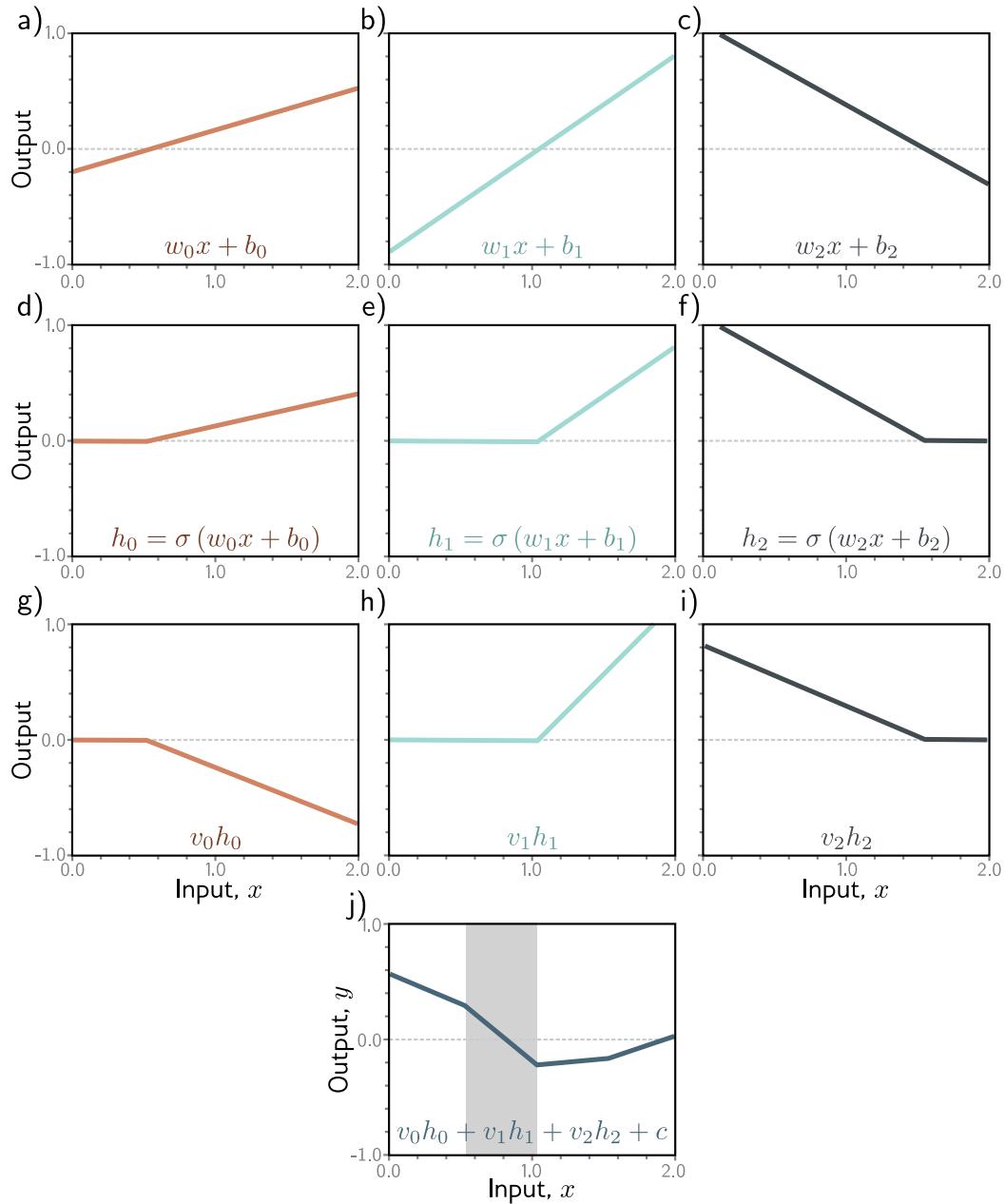
$$y = v_0 \sigma(w_0 x + b_0) + v_1 \sigma(w_1 x + b_1) + v_2 \sigma(w_2 x + b_2) + c$$

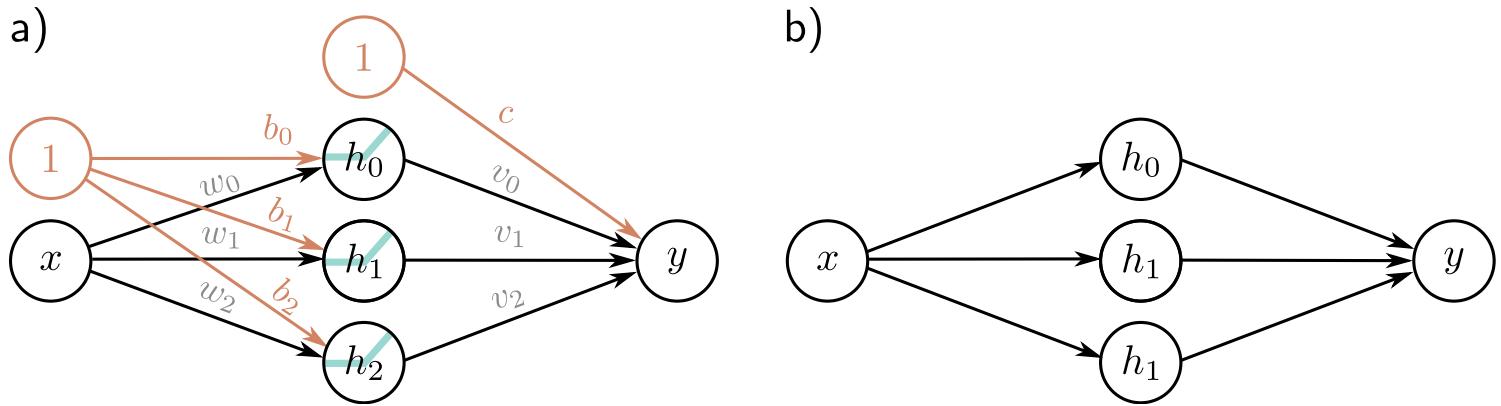
where $w_0, w_1, w_2, b_0, b_1, b_2, v_0, v_1, v_2$ and c are the model parameters and where the activation function σ is $\text{ReLU}(\cdot) = \max(0, \cdot)$.





This network defines a family of piecewise linear functions where the positions of the joints, the slopes and the heights of the functions are determined by the 10 parameters $w_0, w_1, w_2, b_0, b_1, b_2, v_0, v_1, v_2$ and c .



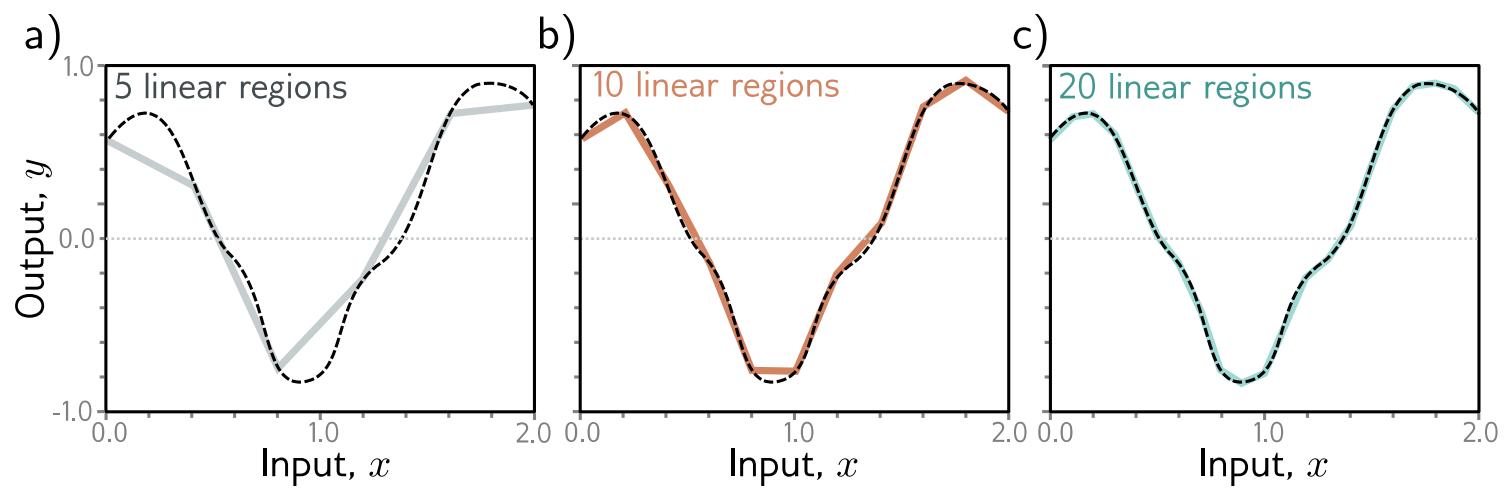


- a) The input x is on the left, the hidden units h_0 , h_1 and h_2 are in the middle, and the output y is on the right. Computation flows from left to right.
- b) More compact representation of the same network where we omit the bias terms, the weight labels and the activation functions.

Universal approximation theorem

The number q of hidden units h_j is a measure of the *capacity* of the shallow network. With **ReLU** activation functions, the hidden units define (up to) q joints in the input space, hence defining $q + 1$ linear regions in the output space.

The **universal approximation theorem** states that a single-hidden-layer network with a finite number of hidden units can approximate any continuous function on a compact subset of \mathbb{R}^d to arbitrary accuracy.



Multivariate outputs

To extend the network to multivariate outputs $\mathbf{y} = [y_0, y_1, \dots, y_{d_{\text{out}}-1}]$, we simply add more output units as linear combinations of the hidden units.

For example, a network with two output units y_0 and y_1 might have the following structure:

$$h_0 = \sigma(w_0x + b_0)$$

$$h_1 = \sigma(w_1x + b_1)$$

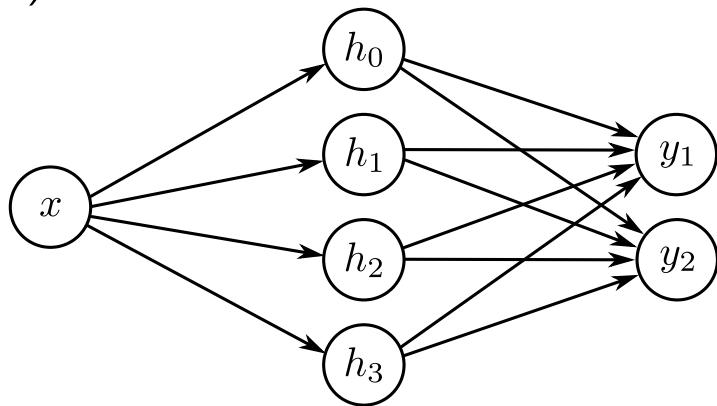
$$h_2 = \sigma(w_2x + b_2)$$

$$h_3 = \sigma(w_3x + b_3)$$

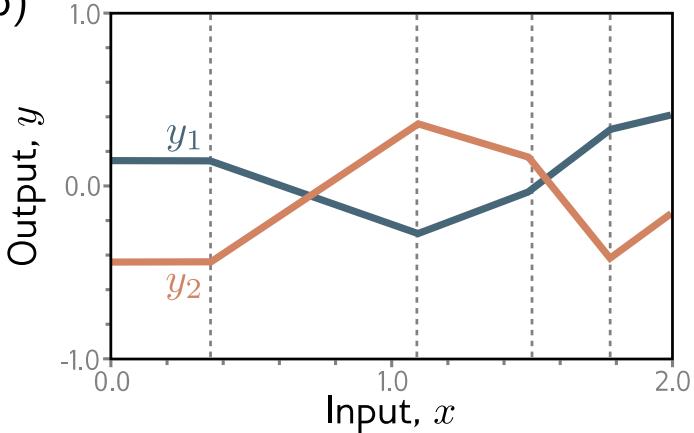
$$y_0 = v_{00}h_0 + v_{01}h_1 + v_{02}h_2 + v_{03}h_3 + c_0$$

$$y_1 = v_{10}h_0 + v_{11}h_1 + v_{12}h_2 + v_{13}h_3 + c_1$$

a)



b)



- a) With two output units, the network can model two functions of the input x .
- b) The four joints of these functions are constrained to be at the same positions, but the slopes and heights of the functions can vary independently.

Multivariate inputs

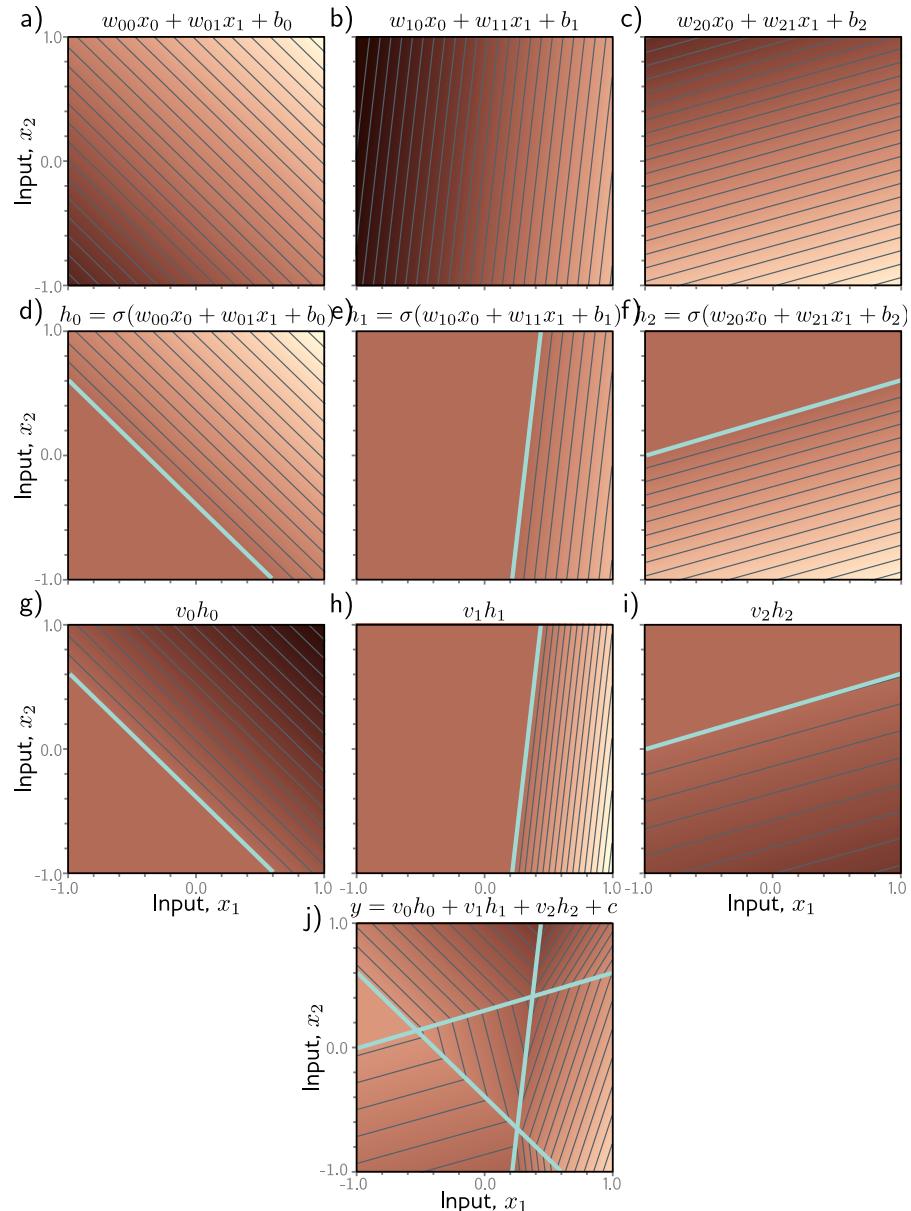
To extend the network to multivariate inputs $\mathbf{x} = [x_0, x_1, \dots, x_{d_{\text{in}}-1}]$, we extend the linear relations between the input and the hidden units.

For example, a network with two inputs $\mathbf{x} = [x_0, x_1]$ and a scalar output y might have three hidden units h_0, h_1 and h_2 defined as

$$\begin{aligned}h_0 &= \sigma(w_{00}x_0 + w_{01}x_1 + b_0) \\h_1 &= \sigma(w_{10}x_0 + w_{11}x_1 + b_1) \\h_2 &= \sigma(w_{20}x_0 + w_{21}x_1 + b_2).\end{aligned}$$

The hidden units are then combined to produce the output y as

$$y = v_0h_0 + v_1h_1 + v_2h_2 + c.$$



Deep networks

We first consider the composition of two shallow networks, where the output of the first network is fed as input to the second network as

$$h_0 = \sigma(w_0x + b_0)$$

$$h_1 = \sigma(w_1x + b_1)$$

$$h_2 = \sigma(w_2x + b_2)$$

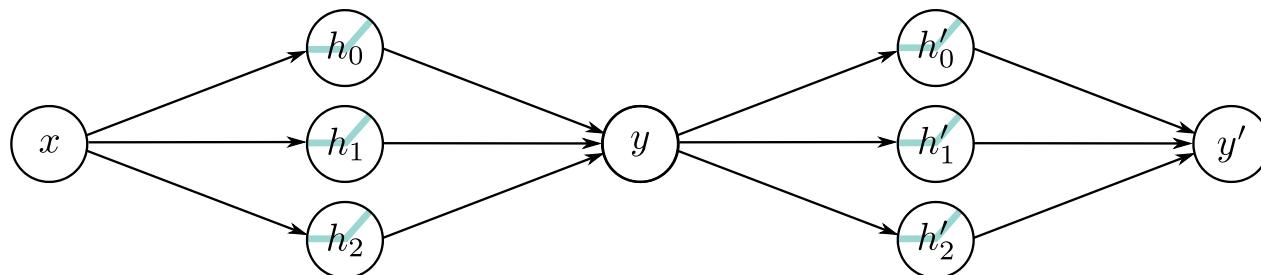
$$y = v_0h_0 + v_1h_1 + v_2h_2 + c$$

$$h'_0 = \sigma(w'_0y + b'_0)$$

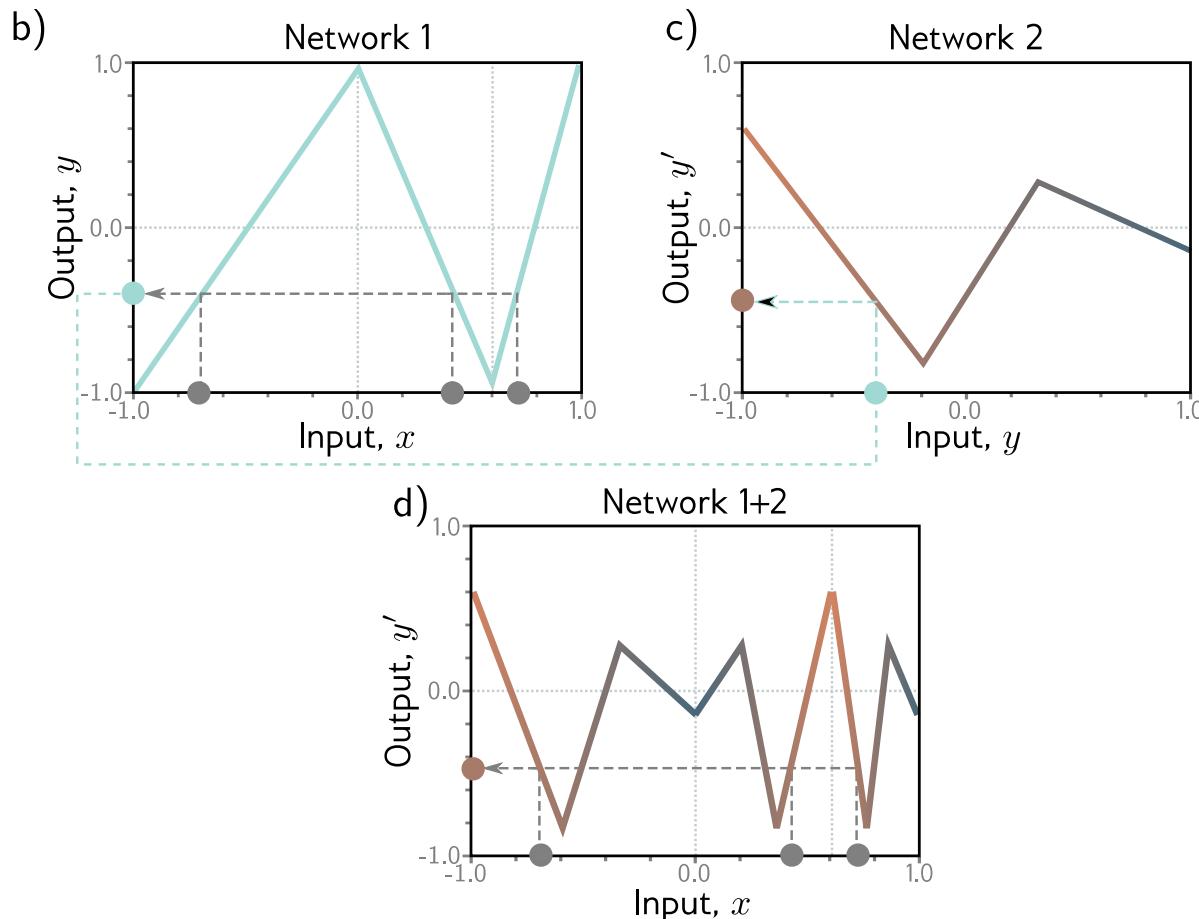
$$h'_1 = \sigma(w'_1y + b'_1)$$

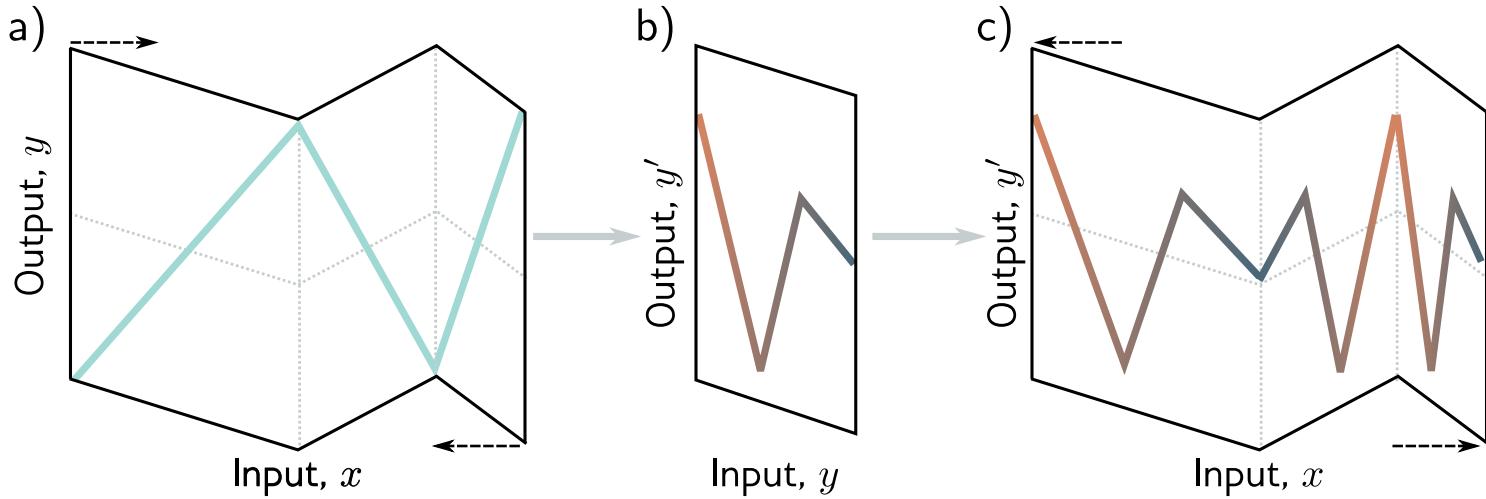
$$h'_2 = \sigma(w'_2y + b'_2)$$

$$y' = v'_0h'_0 + v'_1h'_1 + v'_2h'_2 + c'.$$



With **ReLU** activation functions, this network also describes a family of piecewise linear functions. However, each linear region defined by the hidden units of the first network is further divided by the hidden units of the second network.



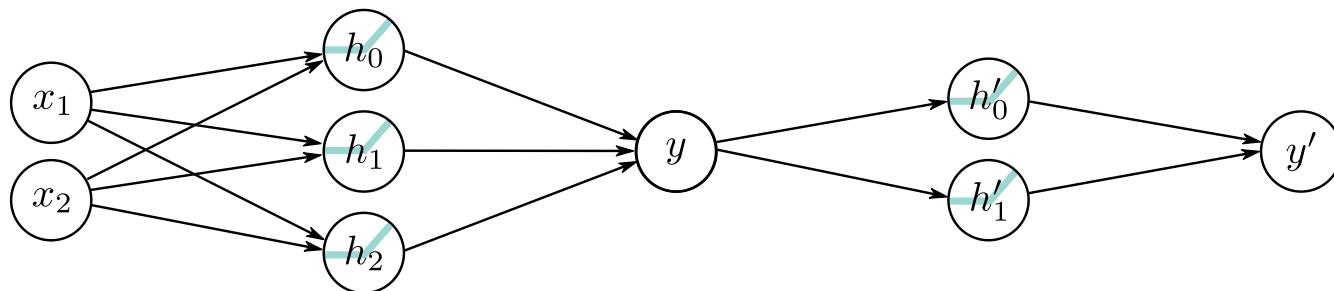


The folding interpretation of a deep network:

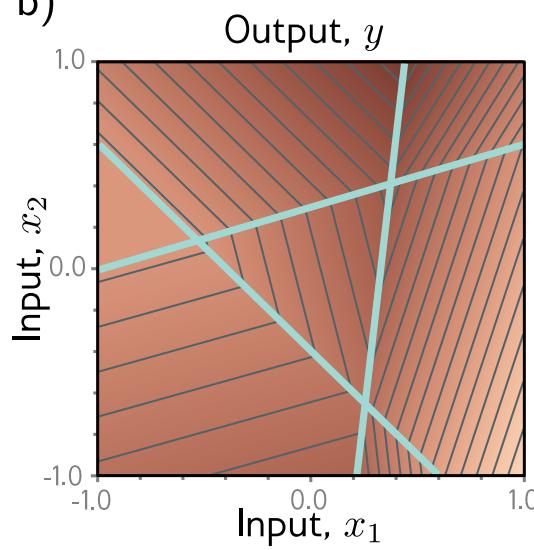
- The first network folds the input space back on itself.
- The second network applies its function to the folded space.
- The final output is revealed by unfolding the folded space.

Similarly, composing a multivariate shallow network with a shallow network further divides the input space into more linear regions.

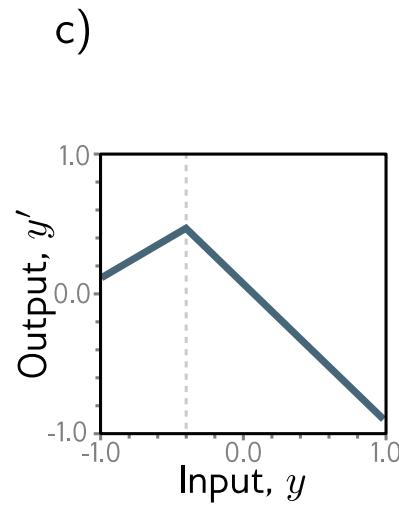
a)



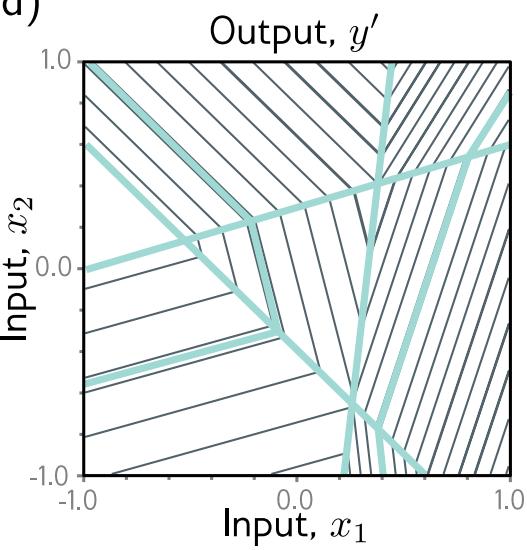
b)



c)



d)



From composing shallow networks to deep networks

Since the operation from $[h_0, h_1, h_2]$ to y is linear and the operation from y to $[h'_0, h'_1, h'_2]$ is also linear, their composition in series is linear.

It follows that the composition of the two shallow networks is a special case of a deep network with two hidden layers where the first layer is defined as

$$h_0 = \sigma(w_0x + b_0)$$

$$h_1 = \sigma(w_1x + b_1)$$

$$h_2 = \sigma(w_2x + b_2),$$

the second layer is defined as

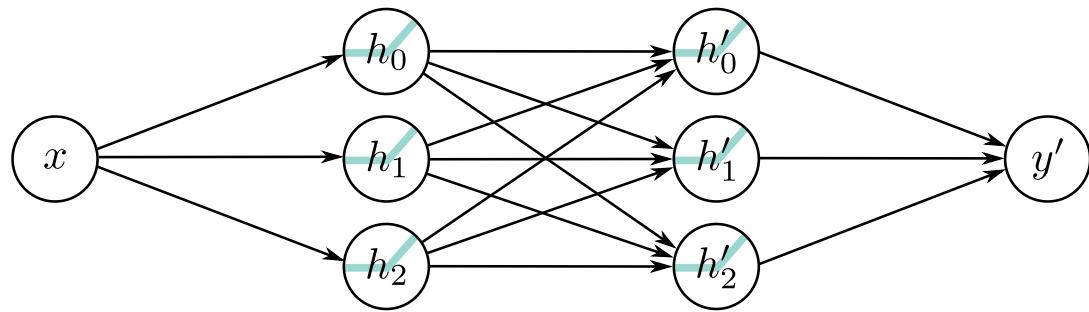
$$h'_0 = \sigma(w'_{00}h_0 + w'_{01}h_1 + w'_{02}h_2 + b'_0)$$

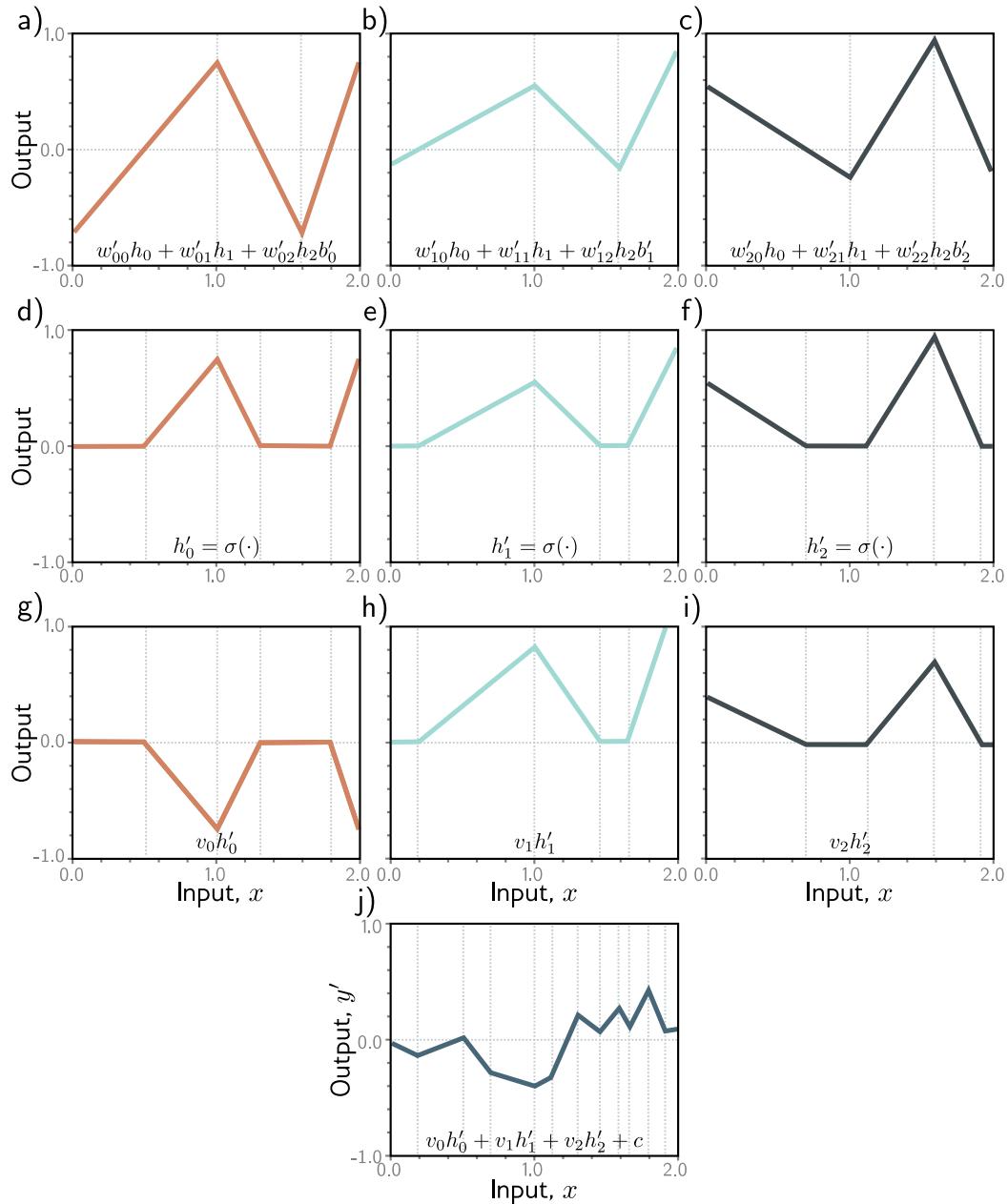
$$h'_1 = \sigma(w'_{10}h_0 + w'_{11}h_1 + w'_{12}h_2 + b'_1)$$

$$h'_2 = \sigma(w'_{20}h_0 + w'_{21}h_1 + w'_{22}h_2 + b'_2)$$

and the output is defined as

$$y = v_0h'_0 + v_1h'_1 + v_2h'_2 + c.$$





General formulation

The computation of a hidden layer can be written in matrix form as

$$\begin{aligned}\mathbf{h} &= \begin{bmatrix} h_0 \\ h_1 \\ \vdots \\ h_{q-1} \end{bmatrix} = \sigma \left(\begin{bmatrix} w_{00} & w_{01} & \cdots & w_{0(d_{\text{in}}-1)} \\ w_{10} & w_{11} & \cdots & w_{1(d_{\text{in}}-1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{(q-1)0} & w_{(q-1)1} & \cdots & w_{(q-1)(d_{\text{in}}-1)} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{d_{\text{in}}-1} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{q-1} \end{bmatrix} \right) \\ &= \sigma(\mathbf{W}^T \mathbf{x} + \mathbf{b})\end{aligned}$$

where $\mathbf{W} \in \mathbb{R}^{d_{\text{in}} \times q}$ is the weight matrix of the hidden layer and $\mathbf{b} \in \mathbb{R}^q$ is the bias vector of the hidden layer.

Hidden layers can be composed in series to form a deep network with L layers such that

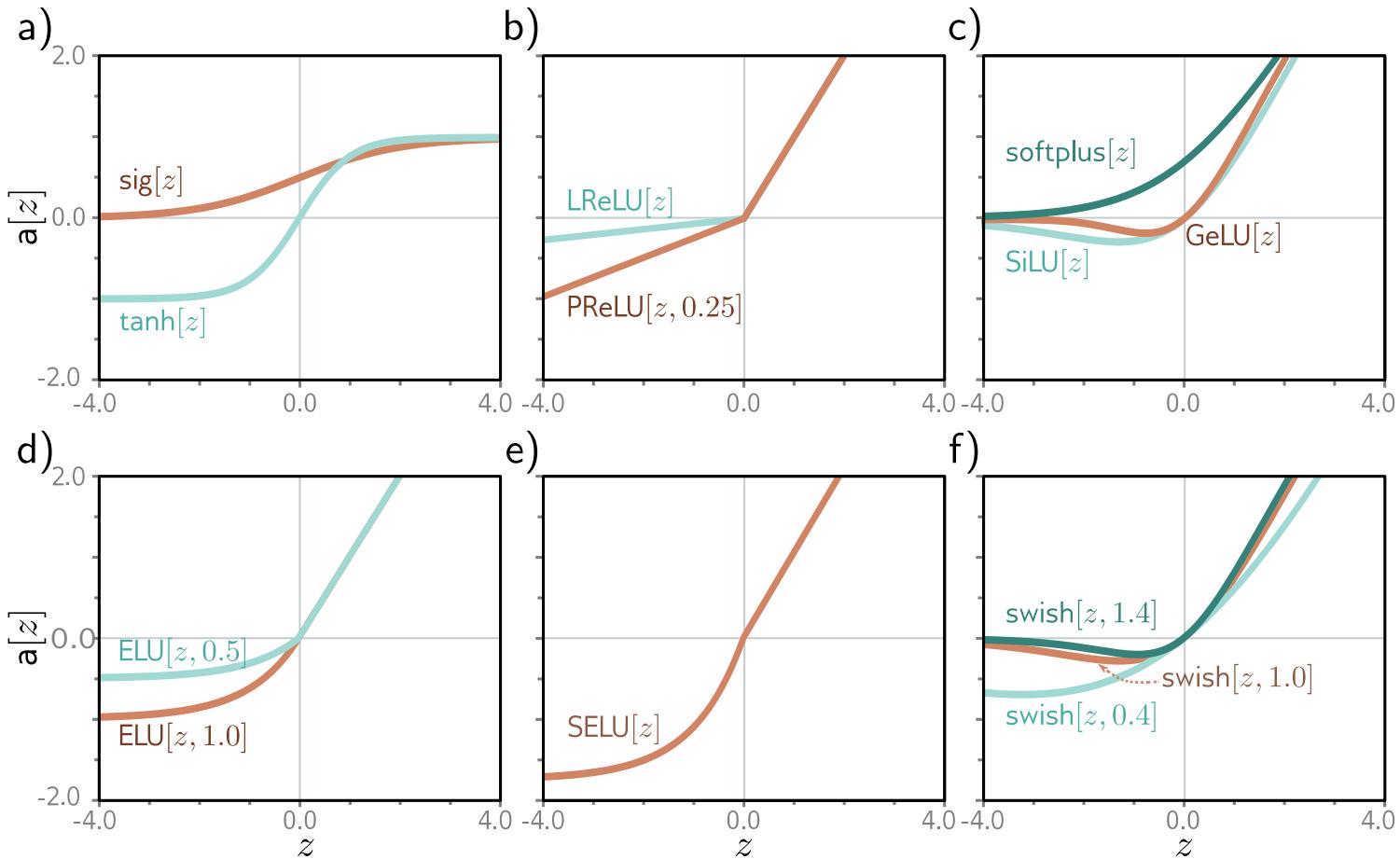
$$\begin{aligned}\mathbf{h}_0 &= \mathbf{x} \\ \mathbf{h}_1 &= \sigma(\mathbf{W}_1^T \mathbf{h}_0 + \mathbf{b}_1) \\ \mathbf{h}_2 &= \sigma(\mathbf{W}_2^T \mathbf{h}_1 + \mathbf{b}_2) \\ &\vdots \\ \mathbf{h}_L &= \sigma(\mathbf{W}_L^T \mathbf{h}_{L-1} + \mathbf{b}_L) \\ \mathbf{y} &= \mathbf{h}_L,\end{aligned}$$

where $\mathbf{W}_\ell \in \mathbb{R}^{q_{\ell-1} \times q_\ell}$ is the weight matrix of the ℓ -th layer, $\mathbf{b}_\ell \in \mathbb{R}^{q_\ell}$ is the bias vector of the ℓ -th layer, and $\mathbf{h}_\ell \in \mathbb{R}^{q_\ell}$ is the hidden vector of the ℓ -th layer.

This model is the feedforward neural network/fully connected network/multilayer perceptron (MLP).

Activation functions

The choice of the activation function σ is crucial for the expressiveness of the network and the optimization of the model parameters.



Output layers

- For regression, the width q of the last layer L is set to the dimensionality of the output d_{out} and the activation function is the identity $\sigma(\cdot) = \cdot$, which results in a vector $\mathbf{h}_L \in \mathbb{R}^{d_{\text{out}}}$.
- For binary classification, the width q of the last layer L is set to 1 and the activation function is the sigmoid $\sigma(\cdot) = \frac{1}{1+\exp(-\cdot)}$, which results in a single output $\mathbf{h}_L \in [0, 1]$ that models the probability $p(y = 1|\mathbf{x})$.
- For multi-class classification, the sigmoid activation σ in the last layer can be generalized to produce a vector $\mathbf{h}_L \in \Delta^C$ of probability estimates $p(y = i|\mathbf{x})$. This activation is the **Softmax** function, where its i -th output is defined as

$$\text{Softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_{j=1}^C \exp(z_j)},$$

for $i = 1, \dots, C$.

Loss functions

The parameters (e.g., \mathbf{W}_ℓ and \mathbf{b}_ℓ for each layer ℓ) of a deep network $f(\mathbf{x}; \theta)$ are learned by minimizing a loss function $\mathcal{L}(\theta)$ over a dataset $\mathbf{d} = \{(\mathbf{x}_j, \mathbf{y}_j)\}$ of input-output pairs.

The loss function is derived from the likelihood:

- For regression, assuming a Gaussian likelihood, the loss is the mean squared error $\mathcal{L}(\theta) = \frac{1}{N} \sum_{(\mathbf{x}_j, \mathbf{y}_j) \in \mathbf{d}} (\mathbf{y}_j - f(\mathbf{x}_j; \theta))^2$.
- For classification, assuming a categorical likelihood, the loss is the cross-entropy $\mathcal{L}(\theta) = -\frac{1}{N} \sum_{(\mathbf{x}_j, \mathbf{y}_j) \in \mathbf{d}} \sum_{i=1}^C y_{ij} \log f_i(\mathbf{x}_j; \theta)$.

(Step-by-step code example)

MLPs on images?

The MLP architecture is appropriate for tabular data, but not for images.

- Each pixel of an image is a feature, leading to a high-dimensional input vector.
- Each hidden unit is connected to all input units, leading to a high-dimensional weight matrix.

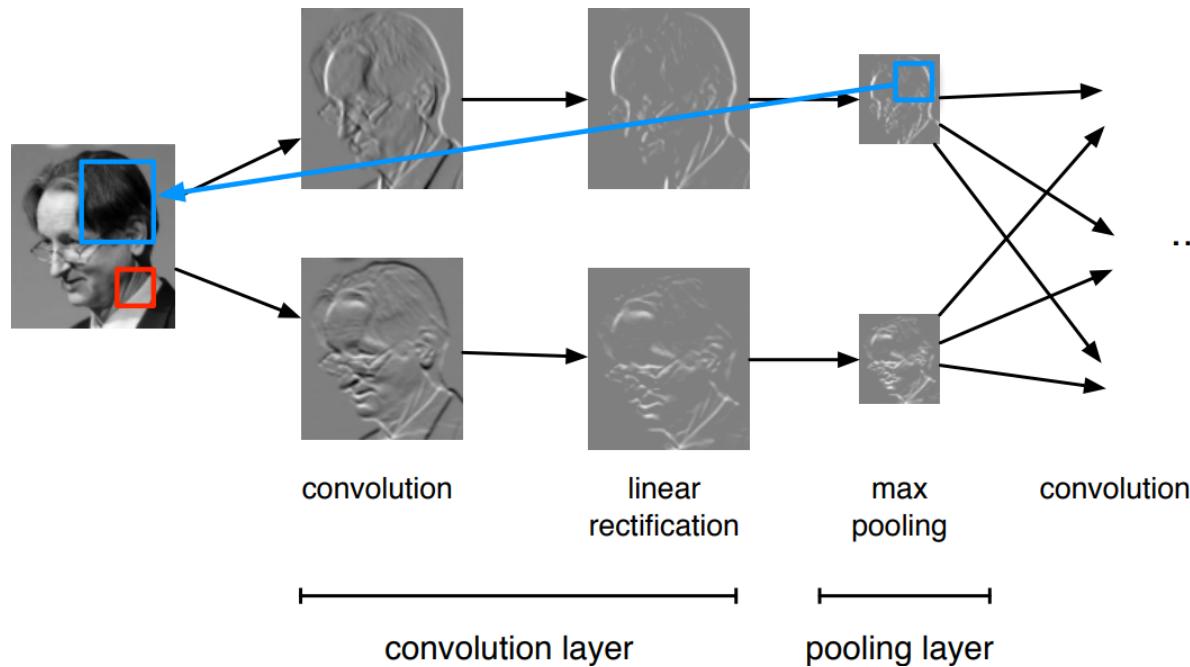
We want to design a neural architecture such that:

- in the earliest layers, the network responds similarly to similar patches of the image, regardless of their location;
- the earliest layers focus on local regions of the image, without regard for the contents of the image in distant regions;
- in the later layers, the network combines the information from the earlier layers to focus on larger and larger regions of the image, eventually combining all the information from the image to classify the image into a category.

Convolutional networks

Convolutional neural networks extend fully connected architectures with

- convolutional layers acting as local feature detectors;
- pooling layers acting as spatial down-samplers.



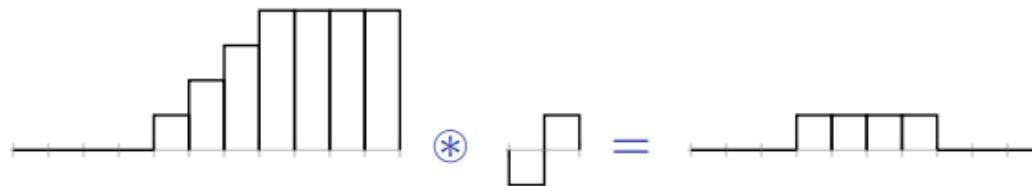
1d convolution

For the one-dimensional input $\mathbf{x} \in \mathbb{R}^W$ and the convolutional kernel $\mathbf{u} \in \mathbb{R}^w$, the discrete convolution $\mathbf{x} \circledast \mathbf{u}$ is a vector of size $W - w + 1$ such that

$$(\mathbf{x} \circledast \mathbf{u})[i] = \sum_{m=0}^{w-1} \mathbf{x}_{m+i} \mathbf{u}_m.$$

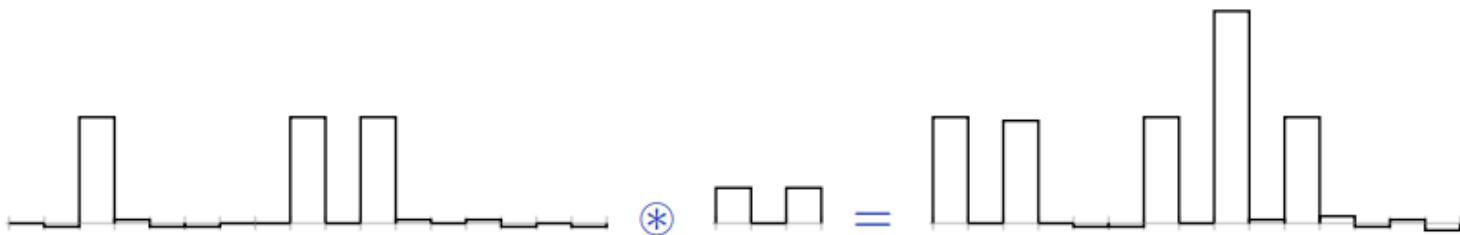
Convolutions can implement differential operators:

$$(0, 0, 0, 0, 1, 2, 3, 4, 4, 4, 4) \circledast (-1, 1) = (0, 0, 0, 1, 1, 1, 1, 0, 0, 0)$$



or crude template matchers:

$$(0, 0, 3, 0, 0, 0, 0, 0, 3, 0, 3, 0, 0, 0) \circledast (1, 0, 1) = (3, 0, 3, 0, 0, 0, 3, 0, 6, 0, 3, 0)$$

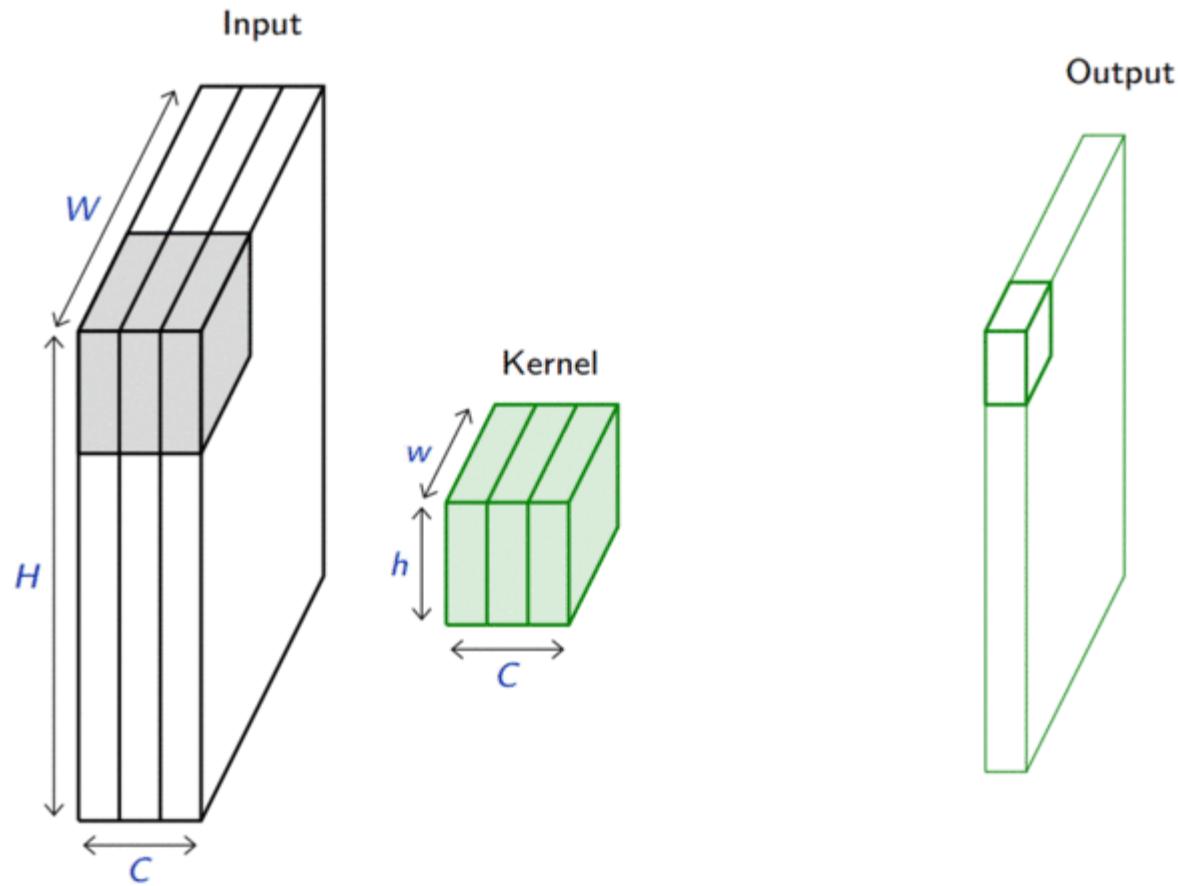


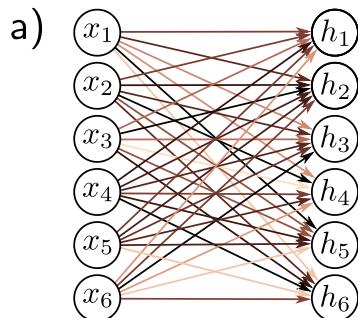
Convolutional layers

A convolutional layer is defined by a set of K kernels \mathbf{u} of size $c \times h \times w$, where h and w are the height and width of the kernel, and c is the number of channels of the input.

Assuming as input a 3D tensor $\mathbf{x} \in \mathbb{R}^{C \times H \times W}$, the output of the convolutional layer is a set of K feature maps of size $H' \times W'$, where $H' = H - h + 1$ and $W' = W - w + 1$. Each feature map \mathbf{o} is the result of convolving the input with a kernel, that is

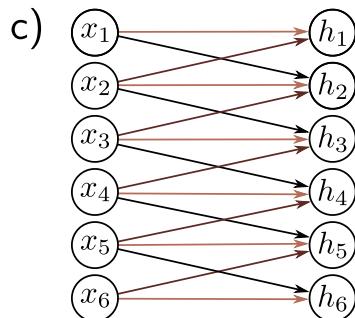
$$\mathbf{o}_{j,i} = (\mathbf{x} \circledast \mathbf{u})[j, i] = \sum_{c=0}^{C-1} \sum_{n=0}^{h-1} \sum_{m=0}^{w-1} \mathbf{x}_{c,n+j,m+i} \mathbf{u}_{c,n,m}$$





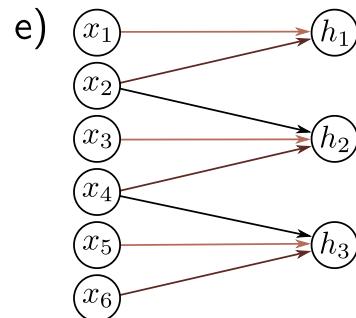
b)

	x_1	x_2	x_3	x_4	x_5	x_6
h_1	#	#	#	#	#	#
h_2	#	#			#	#
h_3	#	#	#	#	#	
h_4	#			#	#	#
h_5		#	#	#	#	#
h_6	#	#	#		#	#



d)

	x_1	x_2	x_3	x_4	x_5	x_6
h_1	#	#				
h_2		#	#			
h_3			#	#		
h_4				#	#	
h_5					#	#
h_6						#



f)

	x_1	x_2	x_3	x_4	x_5	x_6
h_1	#	#				
h_2		#	#		#	#
h_3						#

Convolutional layers (c-f) are a special case of fully connected layers (a-b) where hidden units are connected to local regions of the input through shared weights (the kernels).

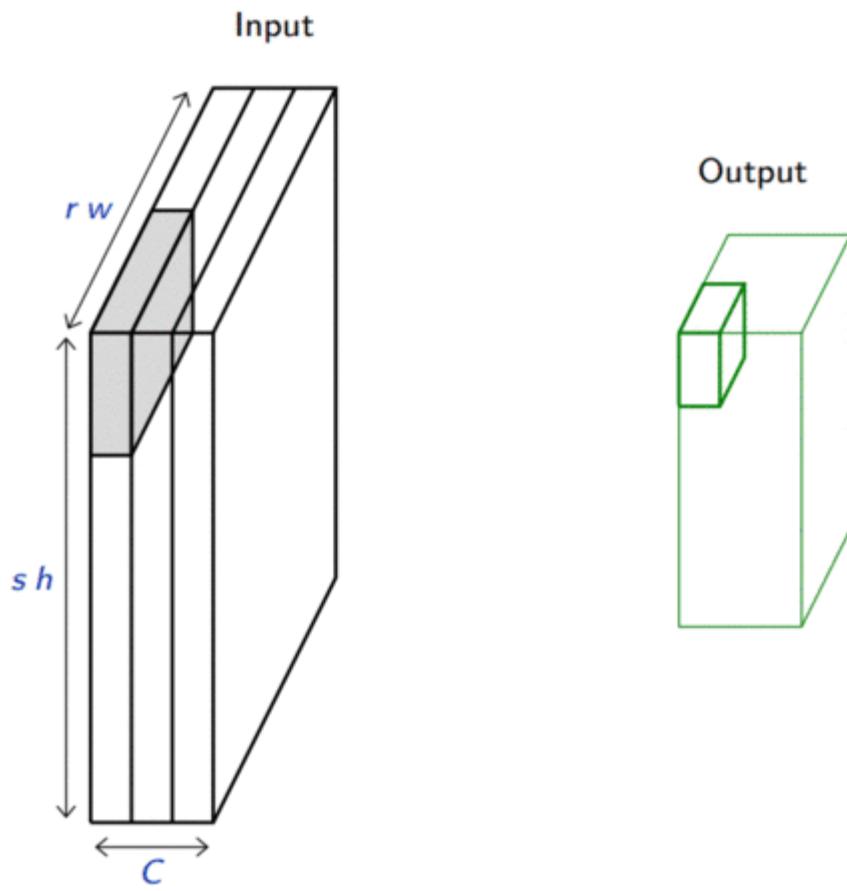
- The connectivity allows the network to learn local patterns in the input.
- Weight sharing allows the network to learn the same patterns at different locations in the input.

Pooling layers

Pooling layers are used to progressively reduce the spatial size of the representation, hence capturing longer-range dependencies between features.

Considering a pooling area of size $h \times w$ and a 3D input tensor $\mathbf{x} \in \mathbb{R}^{C \times (rh) \times (sw)}$, max-pooling produces a tensor $\mathbf{o} \in \mathbb{R}^{C \times r \times s}$ such that

$$\mathbf{o}_{c,j,i} = \max_{n < h, m < w} \mathbf{x}_{c,rj+n,si+m}.$$



(Step-by-step code example)

Recurrent networks

When the input is a sequence $\mathbf{x}_{1:T}$, the feedforward network can be made recurrent by computing a sequence $\mathbf{h}_{1:T}$ of hidden states, where \mathbf{h}_t is a function of both \mathbf{x}_t and the previous hidden states in the sequence.

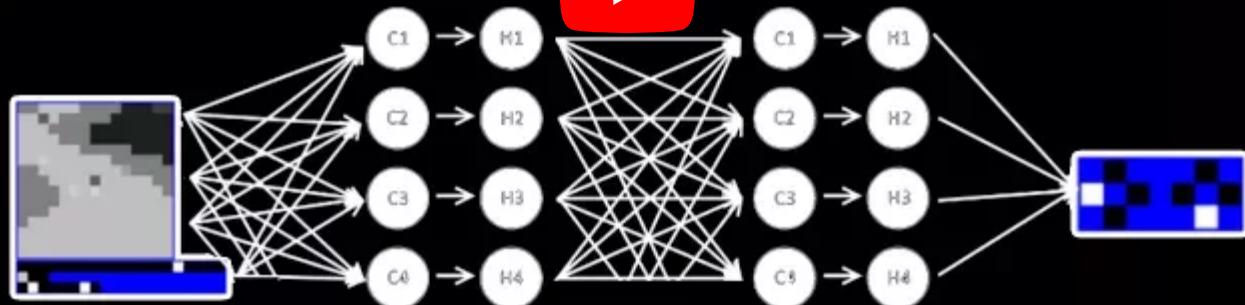
For example,

$$\mathbf{h}_t = \sigma(\mathbf{W}_{xh}^T \mathbf{x}_t + \mathbf{W}_{hh}^T \mathbf{h}_{t-1} + \mathbf{b}),$$

where \mathbf{h}_{t-1} is the previous hidden state in the sequence.

Notice how this is similar to filtering and dynamic decision networks:

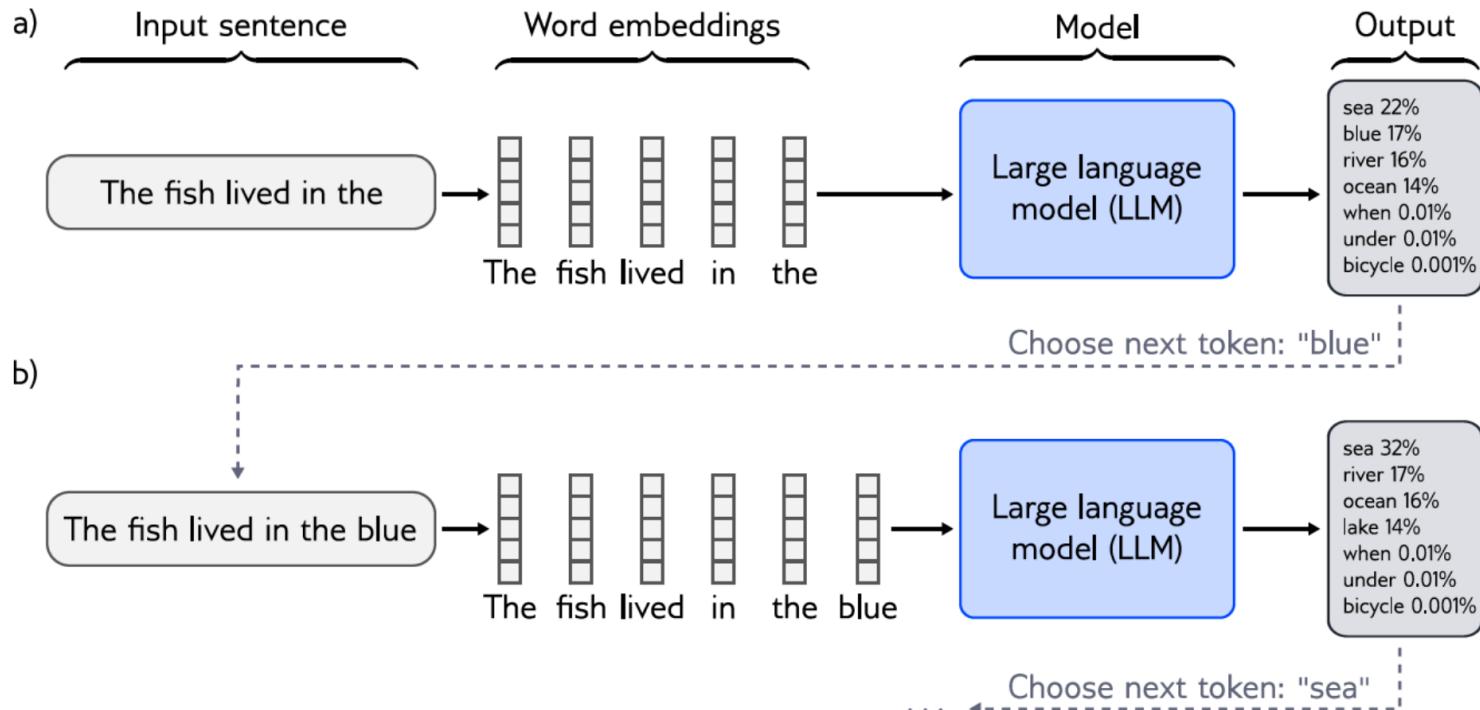
- \mathbf{h}_t can be viewed as some current belief state;
- $\mathbf{x}_{1:T}$ is a sequence of observations;
- \mathbf{h}_{t+1} is computed from the current belief state \mathbf{h}_t and the latest evidence \mathbf{x}_t through some fixed computation (in this case a neural network, instead of being inferred from the assumed dynamics).
- \mathbf{h}_t can also be used to decide on some action, through another network f such that $a_t = f(\mathbf{h}_t; \theta)$.



A recurrent network playing Mario Kart.

Transformers

Transformers are deep neural networks at the core of large-scale language models.

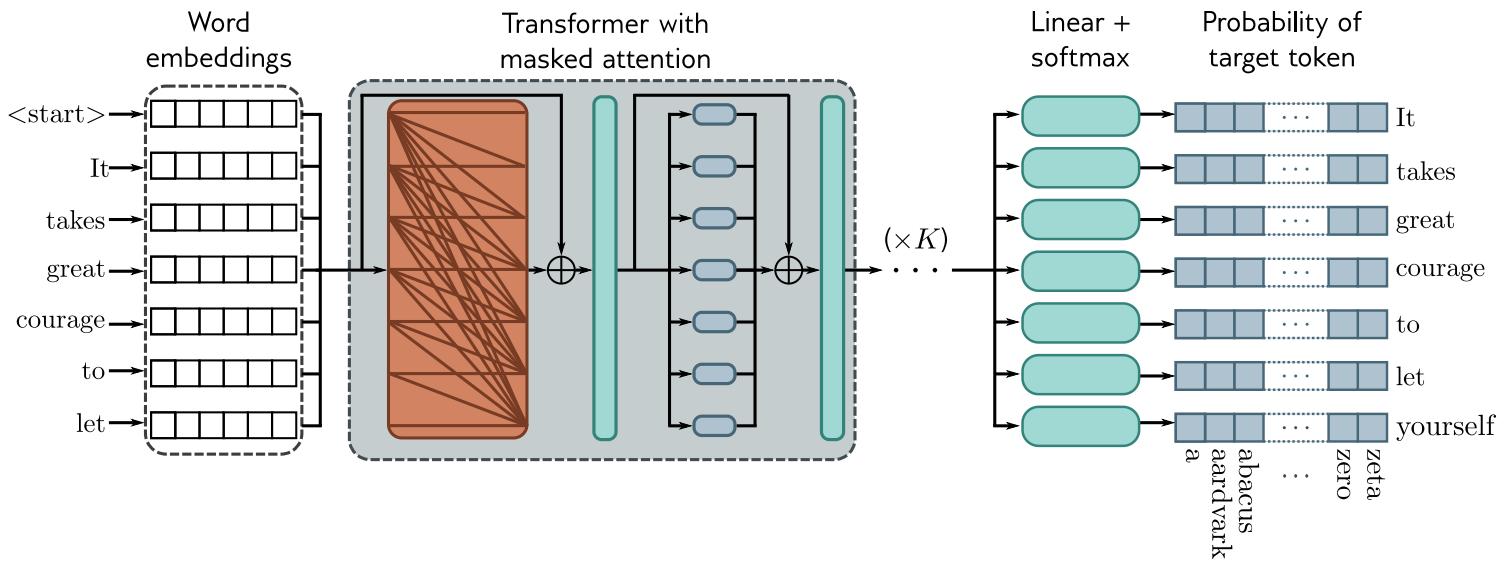


For language modeling, transformers define an **autoregressive model** that predicts the next word in a sequence given the previous words.

Formally,

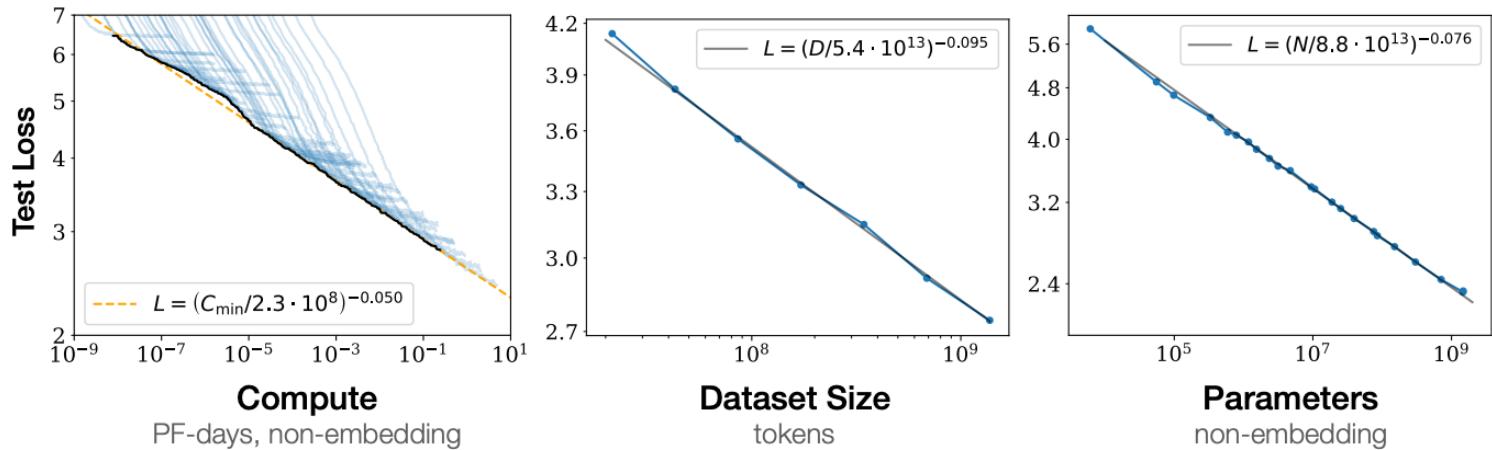
$$p(w_{1:t}) = p(w_1) \prod_{t=2}^T p(w_t | w_{1:t-1}),$$

where w_t is the next word in the sequence and $w_{1:t-1}$ are the previous words.



The decoder-only transformer is a stack of K transformer blocks that process the input sequence in parallel using (masked) self-attention.

The output of the last block is used to predict the next word in the sequence, as in a regular classifier.



Scaling laws

- The more data, the better the model.
- The more parameters, the better the model.
- The more compute, the better the model.

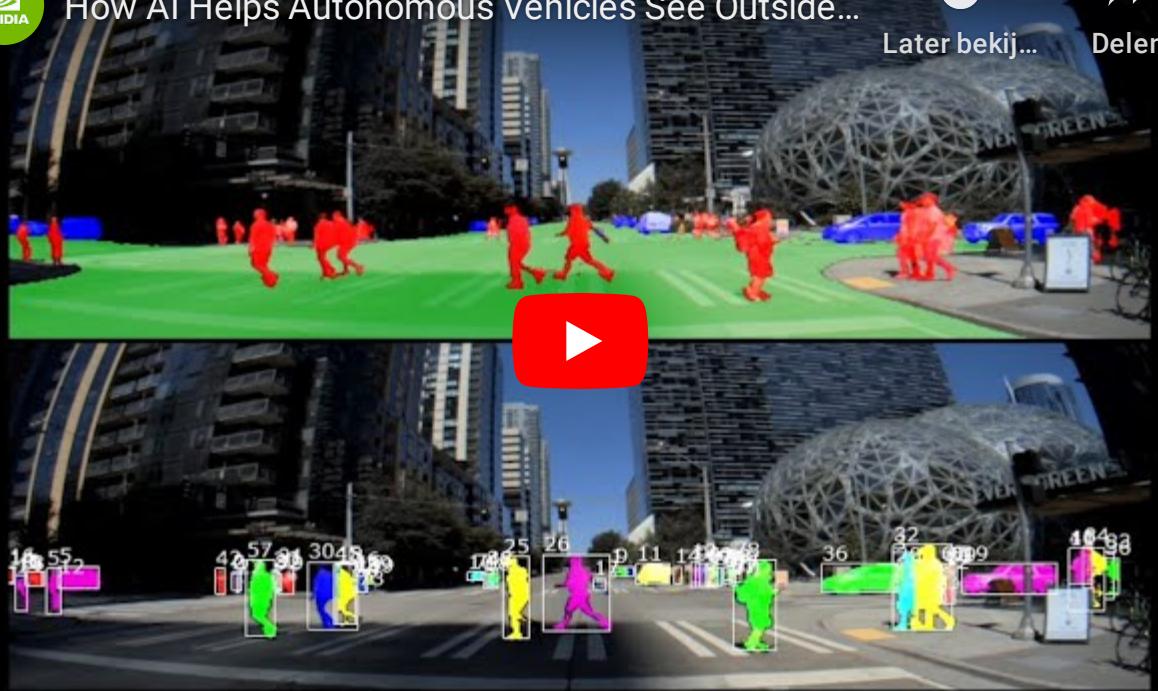
AI beyond Pacman



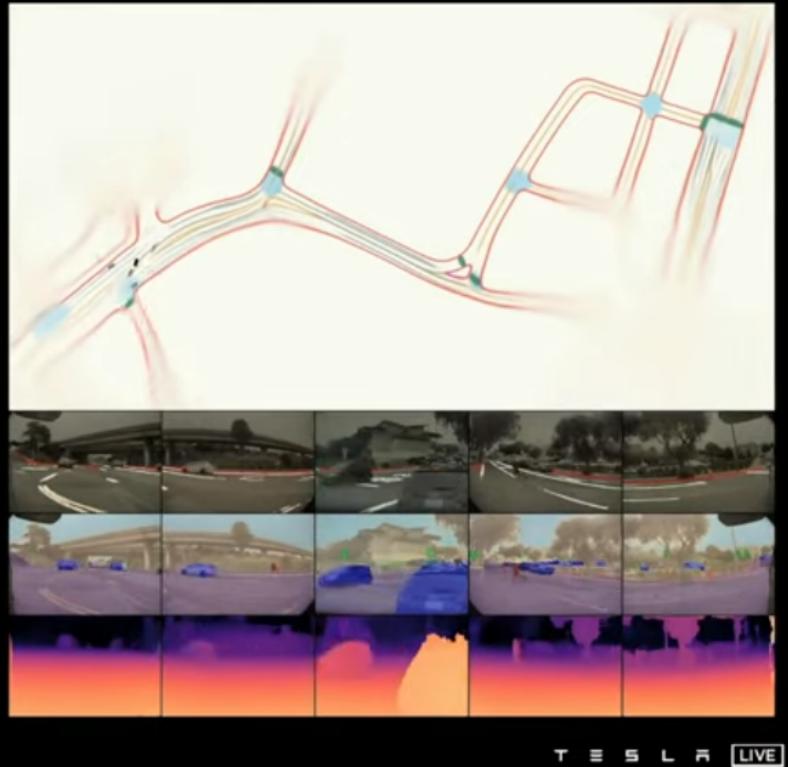
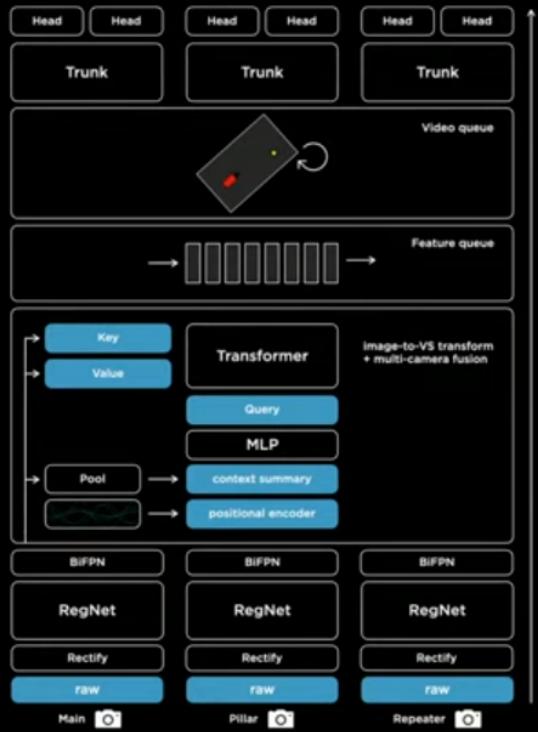
How AI Helps Autonomous Vehicles See Outside...



Later bekij...
Delen



How AI Helps Autonomous Vehicles See Outside the Box
(See also other episodes from NVIDIA DRIVE Labs)



Hydranet (Tesla, 2021)



Camels, Code & Lab Coats: How AI Is Advancing Medicine



Later bekijken



Delen



How machine learning is advancing medicine (Google, 2018)

Summary

- Deep learning is a powerful tool for learning from data.
- Neural networks are composed of layers of neurons that are connected to each other.
- The weights of the connections are learned by minimizing a loss function.
- Convolutional networks are used for image processing.
- Transformers are used for language processing.



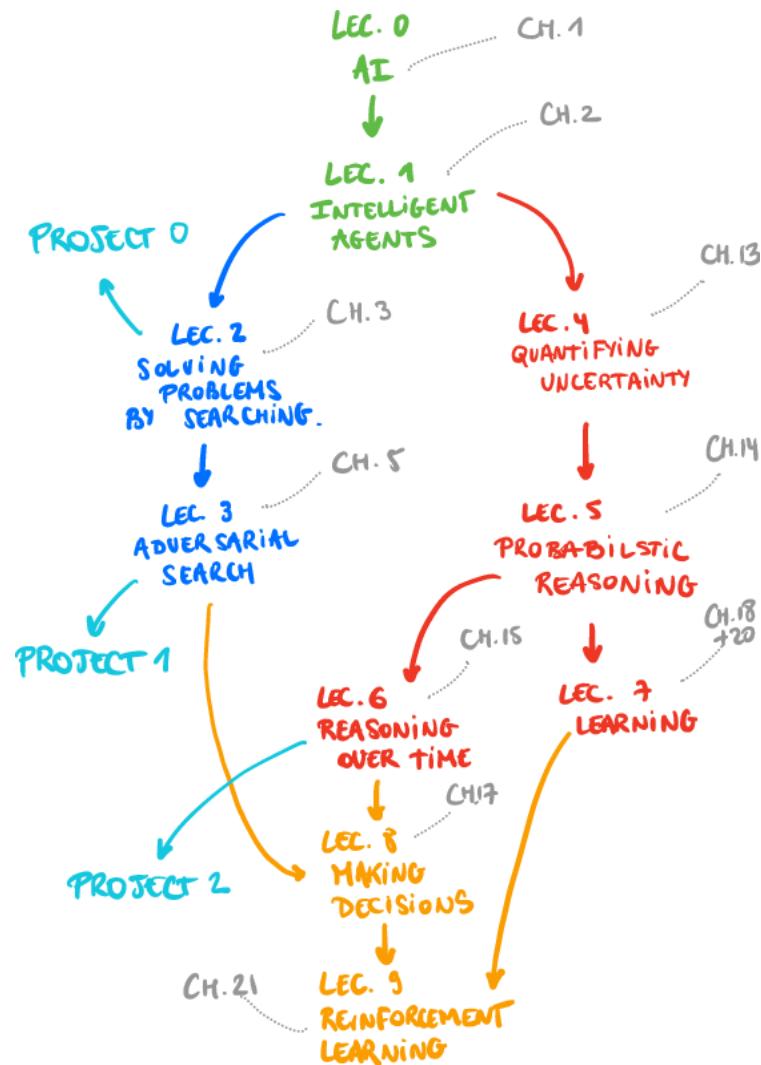
For the last forty years we have programmed computers; for the next forty years we will train them.

Chris Bishop, 2020.

Introduction to Artificial Intelligence

Lecture 8: Making decisions

Prof. Gilles Louppe
g.louppe@uliege.be



Today



Reasoning under uncertainty and **taking decisions**:

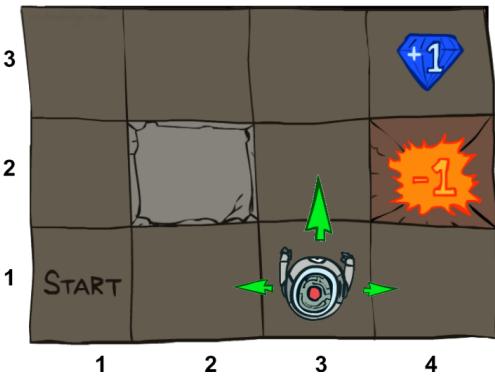
- Markov decision processes
 - MDPs
 - Bellman equation
 - Value iteration
 - Policy iteration
- Partially observable Markov decision processes

Grid world

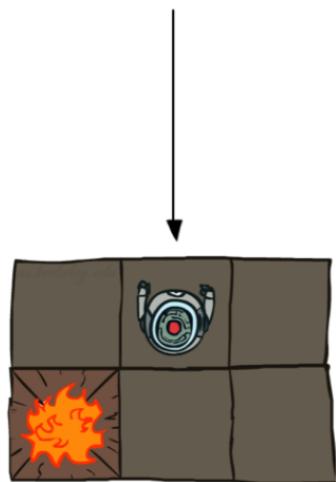
Assume our agent lives in a 3×4 grid environment.

- Noisy movements: actions do not always go as planned.
 - Each action achieves the intended effect with probability **0.8**.
 - The rest of the time, with probability **0.2**, the action moves the agent at right angles to the intended direction.
 - If there is a wall in the direction the agent would have been taken, the agent stays put.
- The agent receives rewards at each time step.
 - Small 'living' reward each step (can be negative).
 - Big rewards come at the end (good or bad).

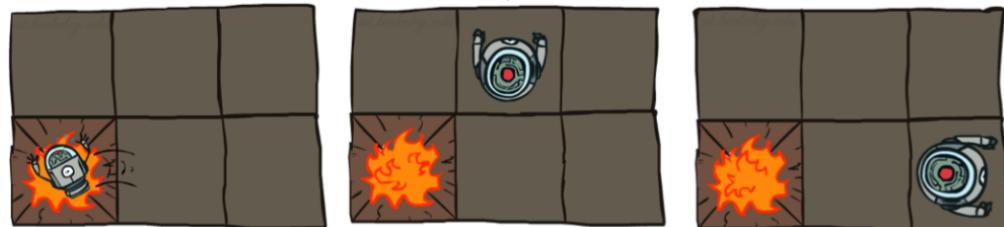
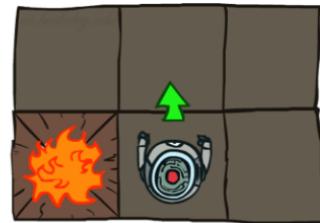
Goal: maximize sum of rewards.



Deterministic
actions



Stochastic actions

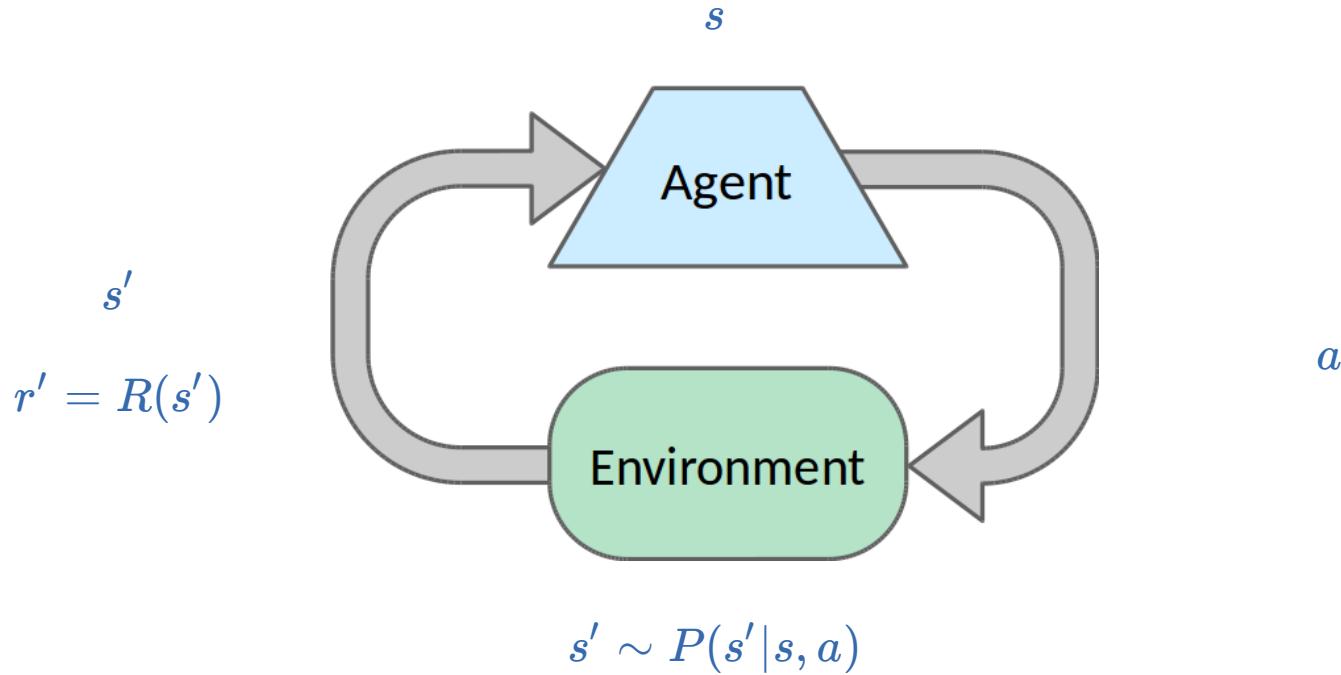


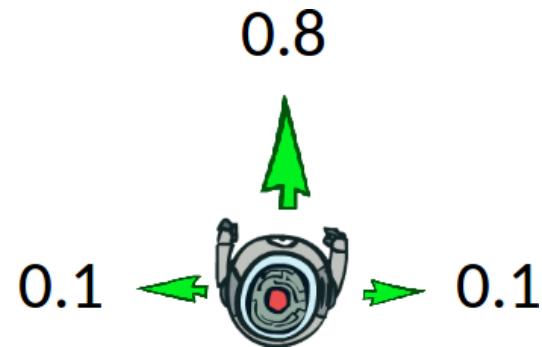
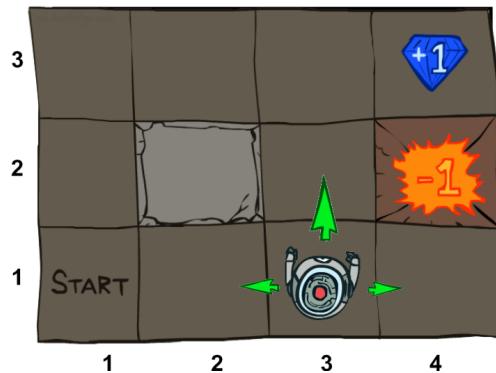
Markov decision processes

Markov decision processes

A **Markov decision process** (MDP) is a tuple $(\mathcal{S}, \mathcal{A}, P, R)$ such that:

- \mathcal{S} is a set of states s ;
- \mathcal{A} is a set of actions a ;
- P is a (stationary) transition model such that $P(s'|s, a)$ denotes the probability of reaching state s' if action a is done in state s ;
- R is a reward function that maps immediate (finite) reward values $R(s)$ obtained in states s .





Example

- \mathcal{S} : locations (i, j) on the grid.
- \mathcal{A} : [Up, Down, Right, Left].
- Transition model: $P(s'|s, a)$
- Reward:

$$R(s) = \begin{cases} -0.3 & \text{for non-terminal states} \\ \pm 1 & \text{for terminal states} \end{cases}$$

What is Markovian about MDPs?

Given the present state, the future and the past are independent:

$$P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \dots, s_0) = P(s_{t+1}|s_t, a_t)$$

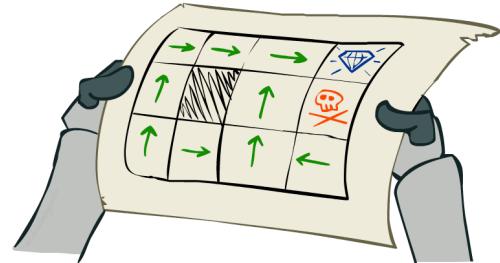
This is similar to search problems, where the successor function could only depend on the current state.



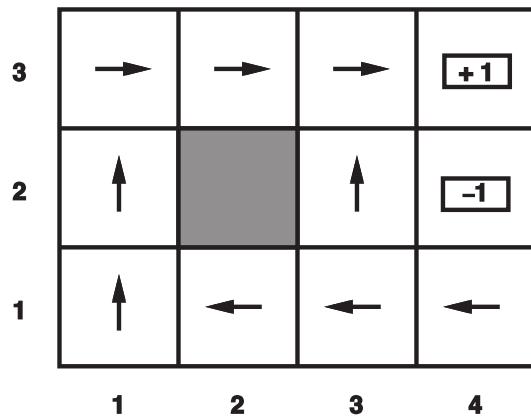
Andrey Markov

Policies

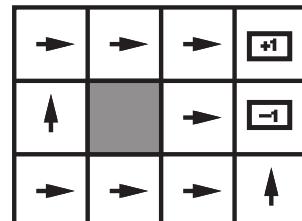
- In deterministic single-agent search problems, our goal was to find an optimal plan, or **sequence** of actions, from start to goal.
- For MDPs, we want to find an optimal **policy** $\pi^* : \mathcal{S} \rightarrow \mathcal{A}$.
 - A policy π maps states to actions.
 - An optimal policy is one that maximizes the expected utility, e.g. the expected sum of rewards.
 - An explicit policy defines a reflex agent.
- Expectiminimax did not compute entire policies, but only some action for a single state.



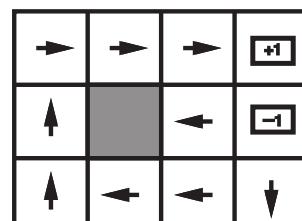
Optimal policy when
 $R(s) = -0.3$ for all non-terminal states s .



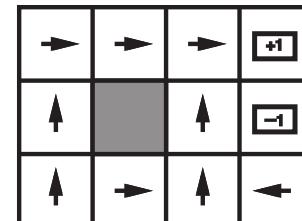
(a)



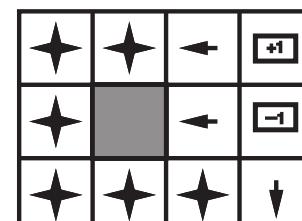
$$R(s) < -1.6284$$



$$-0.4278 < R(s) < -0.0850$$



$$-0.4278 < R(s) < -0.0850$$



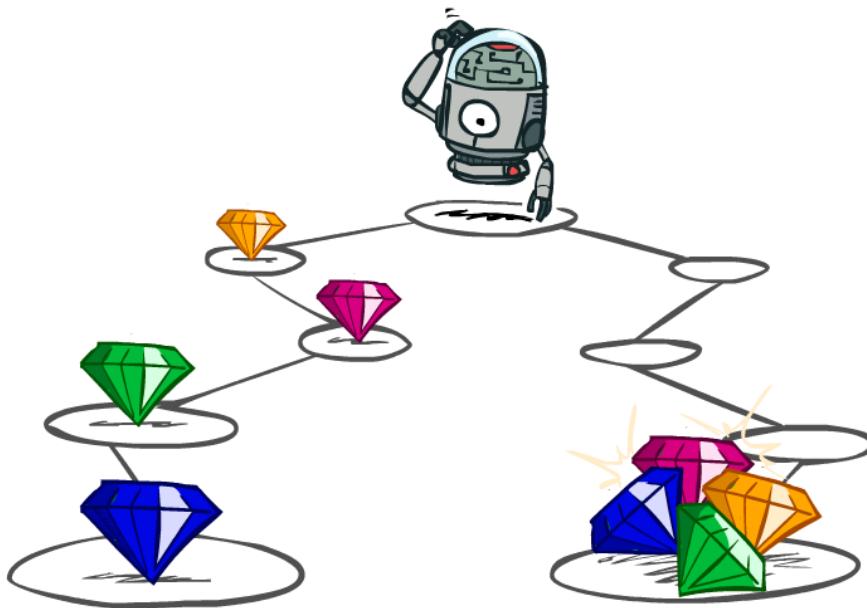
$$-0.0221 < R(s) < 0$$

(b)

(a) Optimal policy when $R(s) = -0.04$ for all non-terminal states s . (b) Optimal policies for four different ranges of $R(s)$.

Depending on $R(s)$, the balance between risk and reward changes from risk-taking to very conservative.

Utilities over time



What preferences should an agent have over state or reward sequences?

- More or less? $[2, 3, 4]$ or $[1, 2, 2]$?
- Now or later? $[1, 0, 0]$ or $[0, 0, 1]$?

Theorem

If we assume **stationary** preferences over reward sequences, i.e. such that

$$[r_0, r_1, r_2, \dots] \succ [r_0, r'_1, r'_2, \dots] \Rightarrow [r_1, r_2, \dots] \succ [r'_1, r'_2, \dots],$$

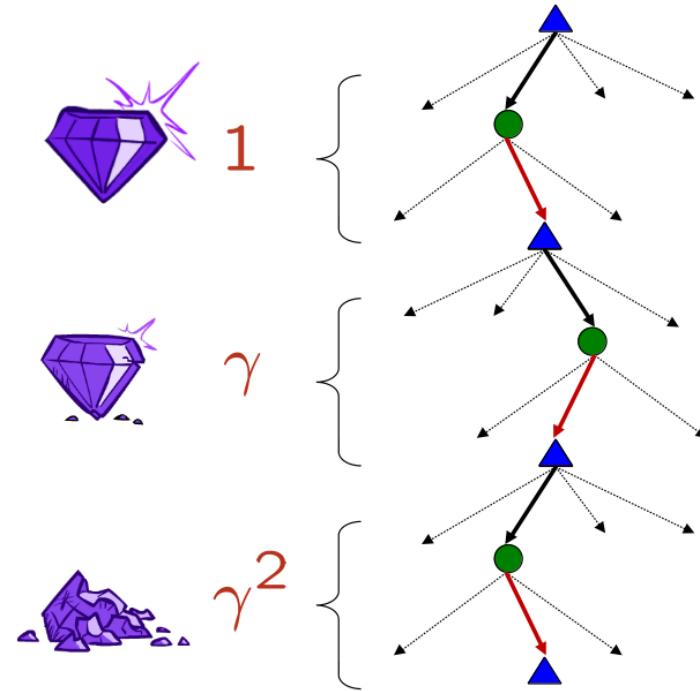
then there are only two coherent ways to assign utilities to sequences:

Additive utility: $V([r_0, r_1, r_2, \dots]) = r_0 + r_1 + r_2 + \dots$

Discounted utility: $V([r_0, r_1, r_2, \dots]) = r_0 + \gamma r_1 + \gamma^2 r_2 + \dots$
 $(0 < \gamma < 1)$

Discounting

- Each time we transition to the next state, we multiply in the discount once.
- Why discount?
 - Sooner rewards probably do have higher utility than later rewards.
 - Will help our algorithms converge.



Example: discount $\gamma = 0.5$

- $V([1, 2, 3]) = 1 + 0.5 \times 2 + 0.25 \times 3$
- $V([1, 2, 3]) < V([3, 2, 1])$

Infinite sequences

What if the agent lives forever? Do we get infinite rewards? Comparing reward sequences with $+\infty$ utility is problematic.

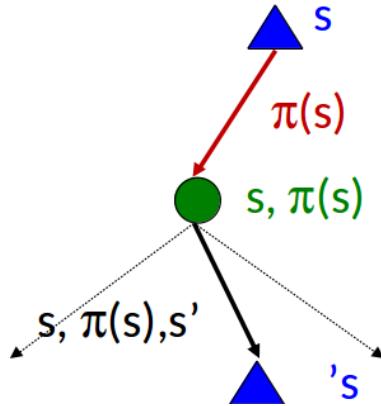
Solutions:

- Finite horizon: (similar to depth-limited search)
 - Terminate episodes after a fixed number of steps $\textcolor{blue}{T}$.
 - Results in non-stationary policies (π depends on time left).
- Discounting (with $0 < \gamma < 1$ and rewards bounded by $\pm R_{\max}$):

$$V([r_0, r_1, \dots, r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \leq \frac{R_{\max}}{1 - \gamma}$$

Smaller γ results in a shorter horizon.

- Absorbing state: guarantee that for every policy, a terminal state will eventually be reached.



Policy evaluation

The expected utility obtained by executing π starting in s is given by

$$V^\pi(s) = \mathbb{E} \left[\sum_{t=0}^{\infty} \gamma^t R(s_t) \right] \Bigg|_{s_0=s}$$

where the expectation is with respect to the probability distribution over state sequences determined by s and π .

Optimal policies

Among all policies the agent could execute, the **optimal policy** is the policy π_s^* that maximizes the expected utility:

$$\pi_s^* = \arg \max_{\pi} V^\pi(s)$$

Because of discounted utilities, the optimal policy is **independent** of the starting state s (see later). Therefore we simply write π^* .

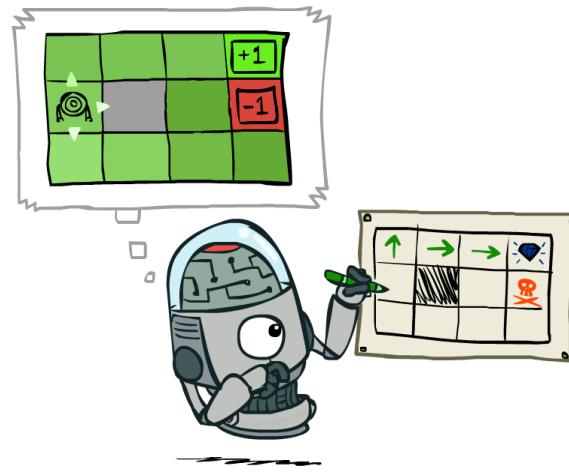
Values of states

The utility, or value, $V(s)$ of a state is now simply defined as $V^{\pi^*}(s)$.

- That is, the expected (discounted) reward if the agent executes an optimal policy starting from s .
- Notice that $R(s)$ and $V(s)$ are quite different quantities:
 - $R(s)$ is the short term reward for having reached s .
 - $V(s)$ is the long term total reward from s onward.

3	0.812	0.868	0.918
2	0.762		0.660
1	0.705	0.655	0.611
	1	2	3
			4

Utilities of the states in Grid World, calculated with $\gamma = 1$ and $R(s) = -0.04$ for non-terminal states.



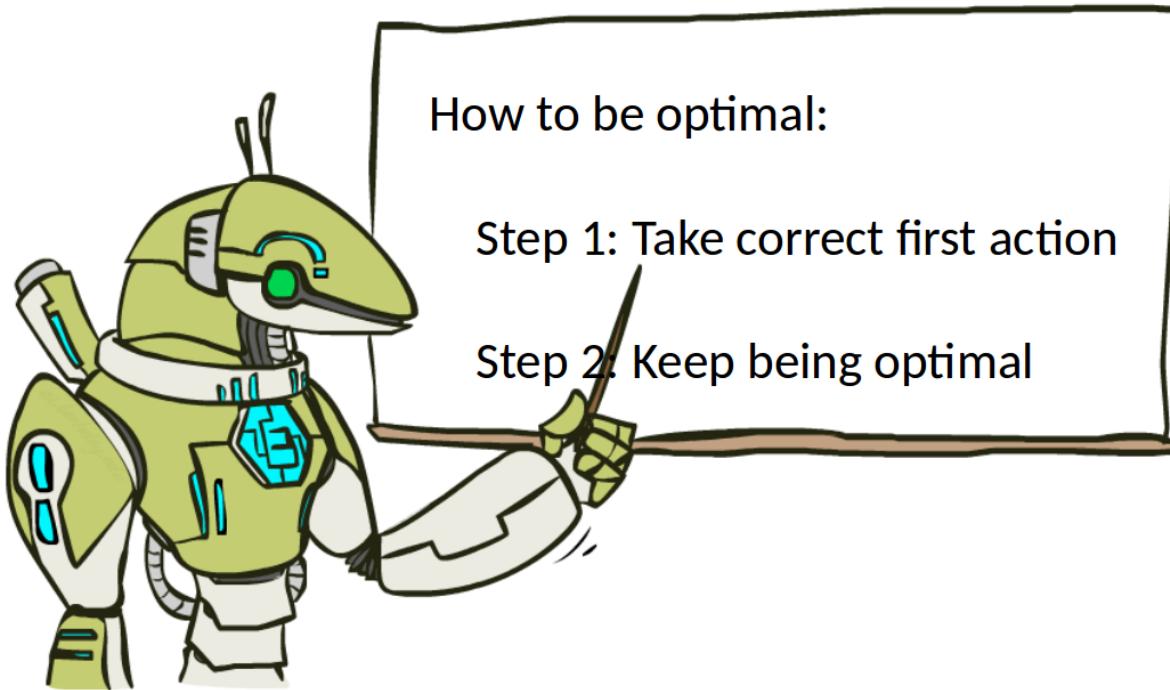
Policy extraction

Using the principle of maximum expected utility, the optimal action maximizes the expected utility of the subsequent state. That is,

$$\pi^*(s) = \arg \max_a \sum_{s'} P(s'|s, a)V(s').$$

Therefore, we can extract the optimal policy provided we can estimate the utilities of states.

$$\pi^*(s) = \arg \max_a \sum_{s'} P(s'|s, a)V(s')$$



The Bellman equation

The utility of a state is the immediate reward for that state, plus the expected discounted utility of the next state, assuming that the agent chooses the optimal action:

$$V(s) = R(s) + \gamma \max_a \sum_{s'} P(s'|s, a)V(s').$$

- These equations are called the **Bellman equations**. They form a system of $n = |\mathcal{S}|$ non-linear equations with as many unknowns.
- The utilities of states, defined as the expected utility of subsequent state sequences, are solutions of the set of Bellman equations.

Example

$$\begin{aligned} V(1, 1) = & -0.04 + \gamma \max[0.8V(1, 2) + 0.1V(2, 1) + 0.1V(1, 1), \\ & 0.9V(1, 1) + 0.1V(1, 2), \\ & 0.9V(1, 1) + 0.1V(2, 1), \\ & 0.8V(2, 1) + 0.1V(1, 2) + 0.1V(1, 1)] \end{aligned}$$

Value iteration

Because of the **max** operator, the Bellman equations are non-linear and solving the system is problematic.

The **value iteration** algorithm provides a fixed-point iteration procedure for computing the state utilities $V(\mathbf{s})$:

- Let $V_i(\mathbf{s})$ be the estimated utility value for \mathbf{s} at the i -th iteration step.
- The **Bellman update** consists in updating simultaneously all the estimates to make them **locally consistent** with the Bellman equation:

$$V_{i+1}(\mathbf{s}) := R(\mathbf{s}) + \gamma \max_a \sum_{\mathbf{s}'} P(\mathbf{s}'|\mathbf{s}, a) V_i(\mathbf{s}')$$

- Repeat until convergence.

function VALUE-ITERATION(mdp, ϵ) **returns** a utility function

inputs: mdp , an MDP with states S , actions $A(s)$, transition model $P(s' | s, a)$, rewards $R(s)$, discount γ

ϵ , the maximum error allowed in the utility of any state

local variables: U, U' , vectors of utilities for states in S , initially zero

δ , the maximum change in the utility of any state in an iteration

repeat

$U \leftarrow U'; \delta \leftarrow 0$

for each state s **in** S **do**

$U'[s] \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' | s, a) U[s']$

if $|U'[s] - U[s]| > \delta$ **then** $\delta \leftarrow |U'[s] - U[s]|$

until $\delta < \epsilon(1 - \gamma)/\gamma$

return U

(Step-by-step code example)

Convergence

Let V_i and V_{i+1} be successive approximations to the true utility V .

Theorem. For any two approximations V_i and V'_i ,

$$\|V_{i+1} - V'_{i+1}\|_\infty \leq \gamma \|V_i - V'_i\|_\infty.$$

- That is, the Bellman update is a contraction by a factor γ on the space of utility vector.
- Therefore, any two approximations must get closer to each other, and in particular any approximation must get closer to the true V .

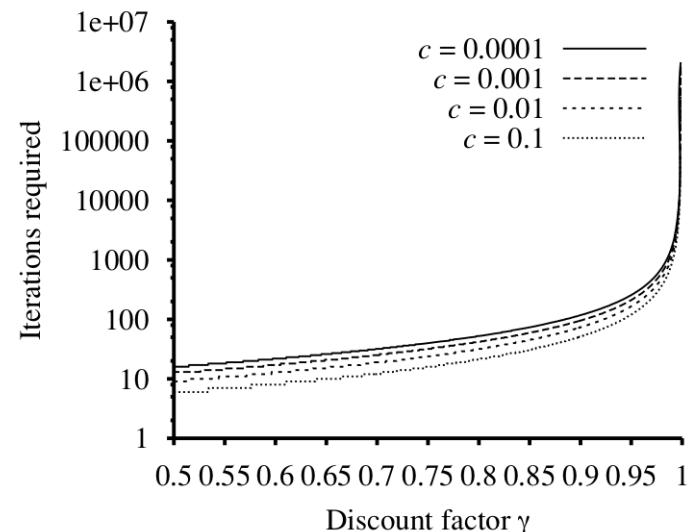
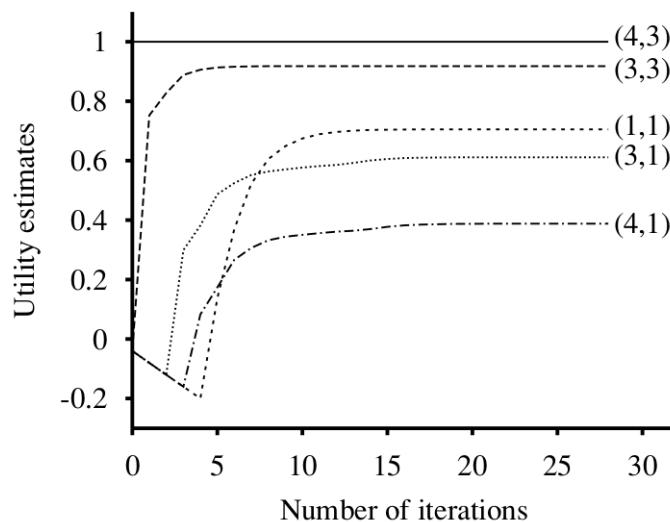
⇒ Value iteration always converges to a unique solution of the Bellman equations whenever $\gamma < 1$.

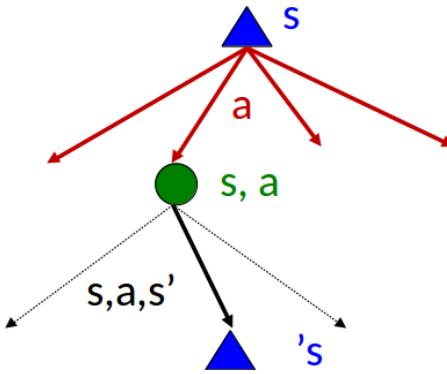
Performance

Since $\|V_{i+1} - V\|_\infty \leq \gamma \|V_i - V\|_\infty$, the error is reduced by a factor of at least γ at each iteration.

Therefore, value iteration converges exponentially fast:

- The maximum initial error is $\|V_0 - V\|_\infty \leq 2R_{\max}/(1 - \gamma)$.
- To reach an error of at most ϵ after N iterations, we require
$$\gamma^N 2R_{\max}/(1 - \gamma) \leq \epsilon.$$





Problems with value iteration

Value iteration repeats the Bellman updates:

$$V_{i+1}(s) = R(s) + \gamma \max_a \sum_{s'} P(s'|s, a) V_i(s')$$

- Problem 1: it is slow – $O(|\mathcal{S}|^2 |\mathcal{A}|)$ per iteration.
- Problem 2: the **max** at each state rarely changes.
- Problem 3: the policy π_i extracted from the estimate V_i might be optimal even if V_i is inaccurate!

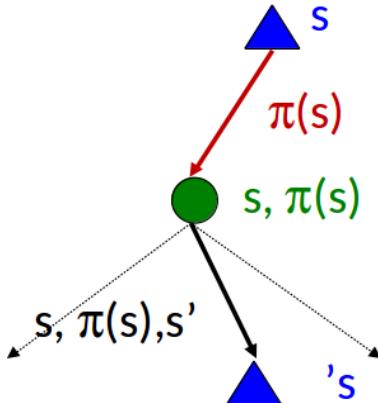
Policy iteration

The **policy iteration** algorithm instead directly computes the policy (instead of state values). It alternates the following two steps:

- Policy evaluation: given π_i , calculate $V_i = V^{\pi_i}$, i.e. the utility of each state if π_i is executed.
- Policy improvement: calculate a new policy π_{i+1} using one-step look-ahead based on V_i :

$$\pi_{i+1}(s) = \arg \max_a \sum_{s'} P(s'|s, a) V_i(s')$$

This algorithm is still optimal, and might converge (much) faster under some conditions.



Policy evaluation

At the i -th iteration we have a simplified version of the Bellman equations that relate the utility of s to the utilities of its neighbors:

$$V_i(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi_i(s)) V_i(s')$$

These equations are now **linear** because the **max** operator has been removed.

- for n states, we have n equations with n unknowns;
- this can be solved exactly in $O(n^3)$ by standard linear algebra methods.

In some cases $O(n^3)$ is too prohibitive. Fortunately, it is not necessary to perform exact policy evaluation. An approximate solution is sufficient.

One way is to run k iterations of simplified Bellman updates:

$$V_{i+1}(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi_i(s)) V_i(s')$$

This hybrid algorithm is called **modified policy iteration**.

```

function POLICY-ITERATION(mdp) returns a policy
  inputs: mdp, an MDP with states  $S$ , actions  $A(s)$ , transition model  $P(s' | s, a)$ 
  local variables:  $U$ , a vector of utilities for states in  $S$ , initially zero
     $\pi$ , a policy vector indexed by state, initially random

  repeat
     $U \leftarrow \text{POLICY-EVALUATION}(\pi, U, mdp)$ 
    unchanged?  $\leftarrow$  true
    for each state  $s$  in  $S$  do
      if  $\max_{a \in A(s)} \sum_{s'} P(s' | s, a) U[s'] > \sum_{s'} P(s' | s, \pi[s]) U[s']$  then do
         $\pi[s] \leftarrow \operatorname{argmax}_{a \in A(s)} \sum_{s'} P(s' | s, a) U[s']$ 
      unchanged?  $\leftarrow$  false
    until unchanged?
  return  $\pi$ 

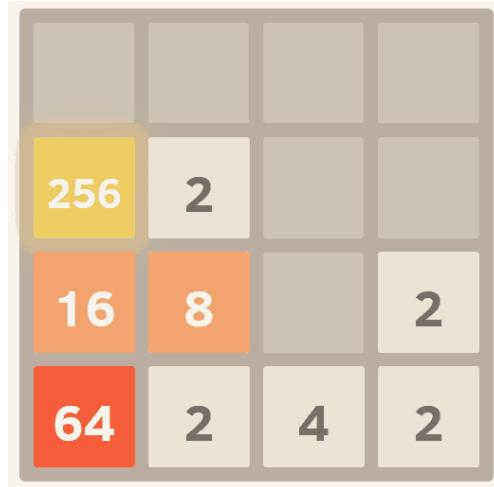
```

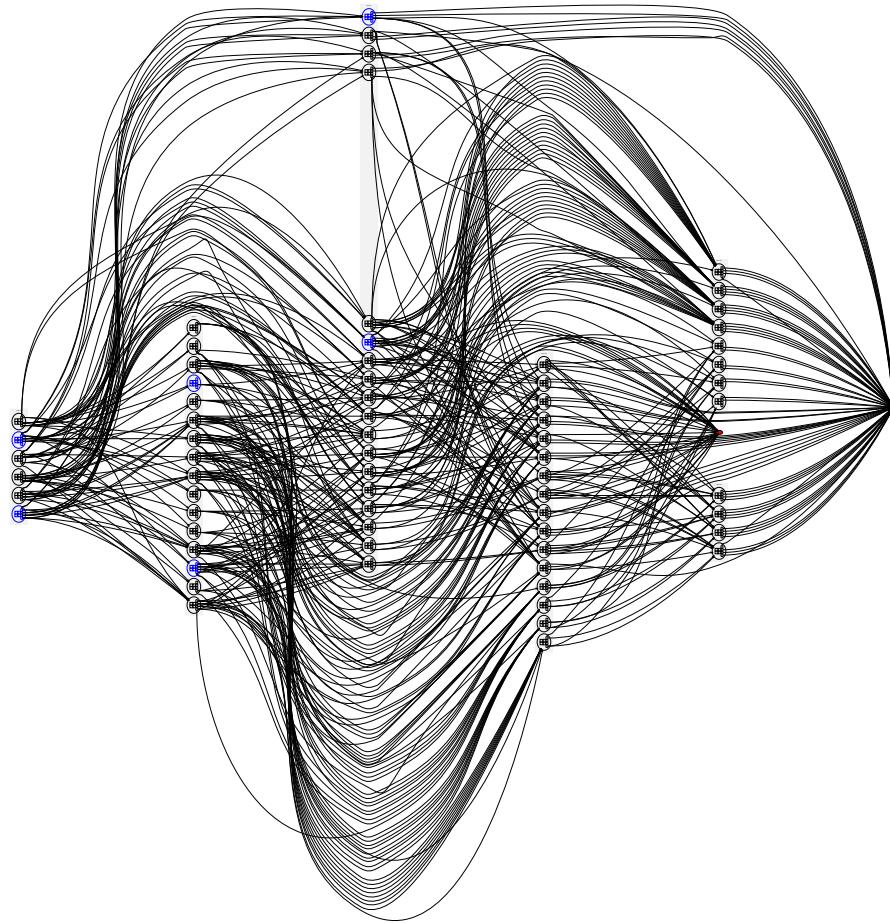
(Step-by-step code example)

Recap example: 2048

The game 2048 is a Markov decision process!

- \mathcal{S} : all possible configurations of the board (huge!)
- \mathcal{A} : swiping left, right, up or down.
- $P(s'|s, a)$: encodes the game's dynamic
 - collapse matching tiles
 - place a random tile on the board
- $R(s) = 1$ if s is a winning state, and 0 otherwise.





The transition model for a 2×2 board and a winning state at 8.

Optimal play for a 3×3 grid and a winning state at **1024**.

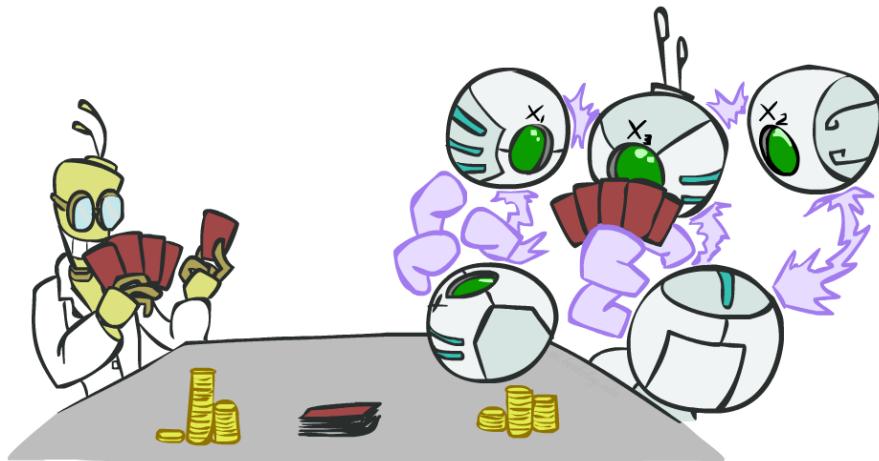
See jdlm.info: The Mathematics of 2048.

Partially observable Markov decision processes

POMDPs

What if the environment is only **partially observable**?

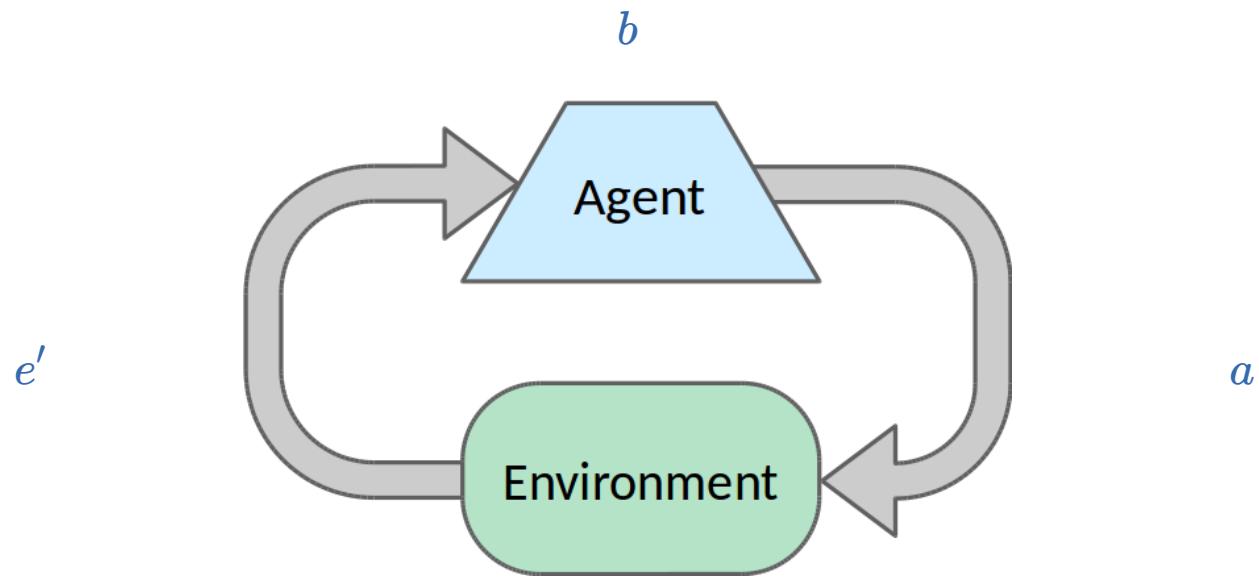
- The agent does not know in which state s it is in.
 - Therefore, it cannot evaluate the reward $R(s)$ associated to the unknown state.
 - Also, it makes no sense to talk about a policy $\pi(s)$.
- Instead, the agent collects percepts e through a sensor model $P(e|s)$, from which it can reason about the unknown state s .



We will assume that the agent maintains a belief state b .

- b represents a probability distribution $\mathbf{P}(S)$ of the current agent's beliefs over its state;
- $b(s)$ denotes the probability $P(S = s)$ under the current belief state;
- the belief state b is updated as evidence e are collected.

This is filtering!



$$s' \sim P(s'|s, a)$$

$$e' \sim P(e'|s')$$

Belief MDP

Theorem (Astrom, 1965). The optimal action depends only on the agent's current belief state.

- The optimal policy can be described by a mapping $\pi^*(b)$ from beliefs to actions.
- It does not depend on the actual state the agent is in.

In other words, POMDPs can be reduced to an MDP in belief-state space, provided we can define a transition model $P(b'|b, a)$ and a reward function ρ over belief states.

If b was the previous belief state and the agent does action a and perceives e , then the new belief state over S' is given by

$$b' = \alpha \mathbf{P}(e|S') \sum_s \mathbf{P}(S'|s, a) b(s) = \alpha \text{forward}(b, a, e).$$

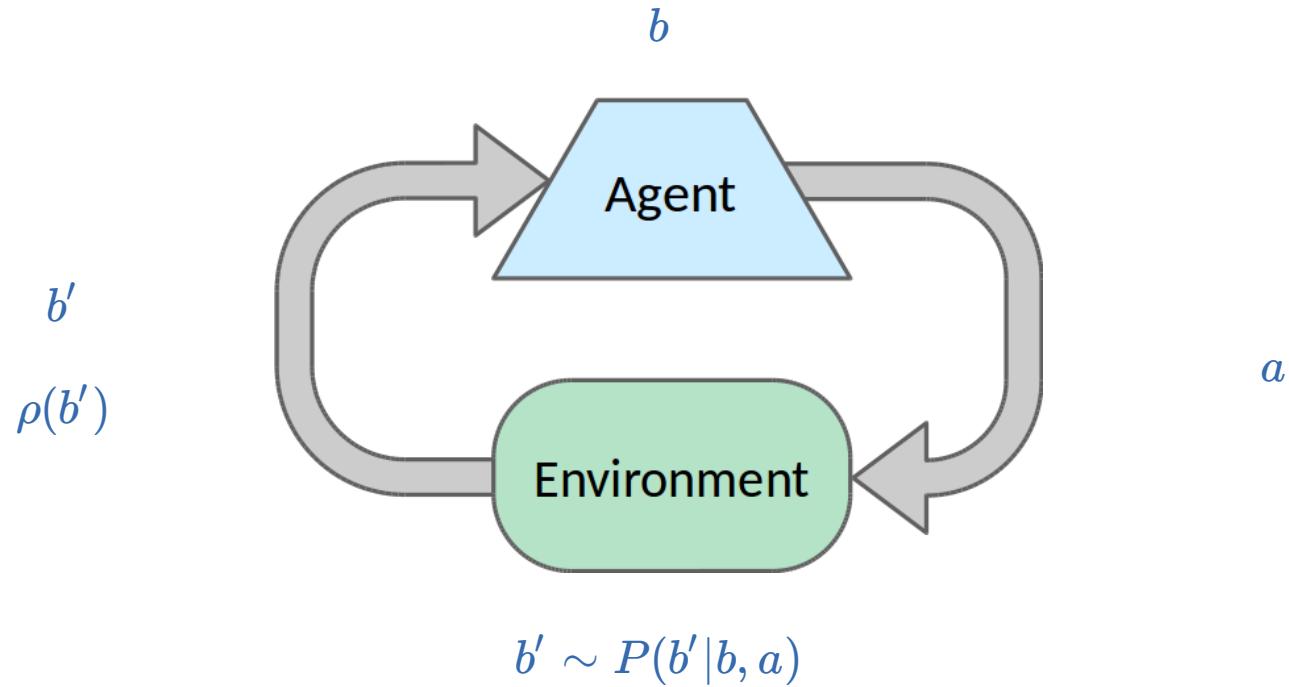
Therefore,

$$\begin{aligned} P(b'|b, a) &= \sum_e P(b', e|b, a) \\ &= \sum_e P(b'|b, a, e) P(e|b, a) \\ &= \sum_e P(b'|b, a, e) \sum_{s'} P(e|b, a, s') P(s'|b, a) \\ &= \sum_e P(b'|b, a, e) \sum_{s'} P(e|s') \sum_s P(s'|s, a) b(s) \end{aligned}$$

where $P(b'|b, a, e) = 1$ if $b' = \text{forward}(b, a, e)$ and 0 otherwise.

We can also define a reward function for belief states as the expected reward for the actual state the agent might be in:

$$\rho(b) = \sum_s b(s)R(s)$$



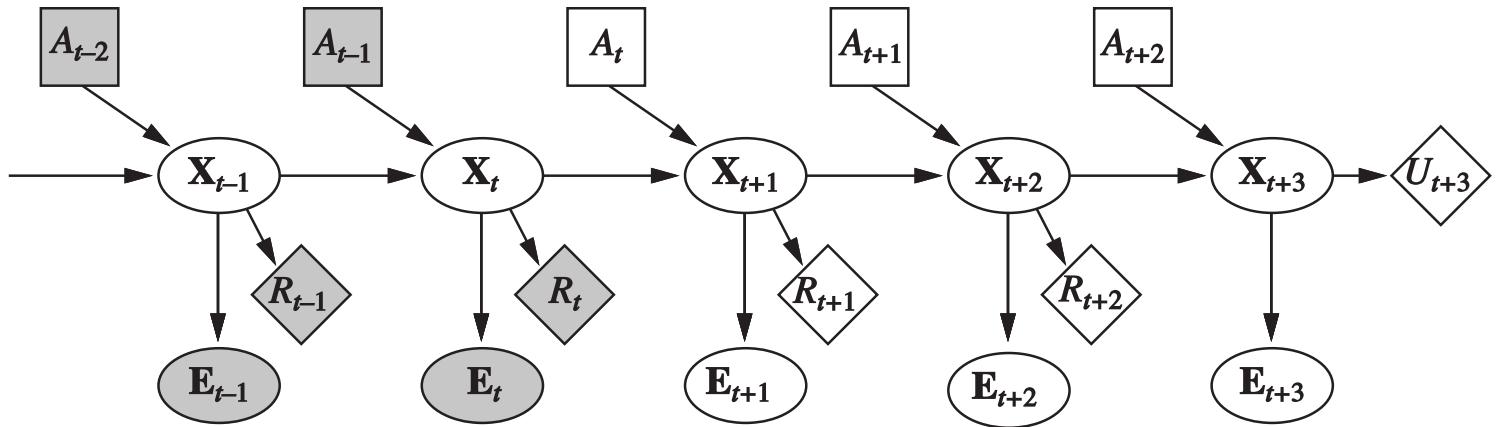
Although we have reduced POMDPs to MDPs, the Belief MDP we obtain has a **continuous** (and usually high-dimensional) state space.

- None of the algorithms described earlier directly apply.
- In fact, solving POMDPs remains a difficult problem for which there is no known efficient exact algorithm.
- Yet, Nature is a POMDP.

Online agents

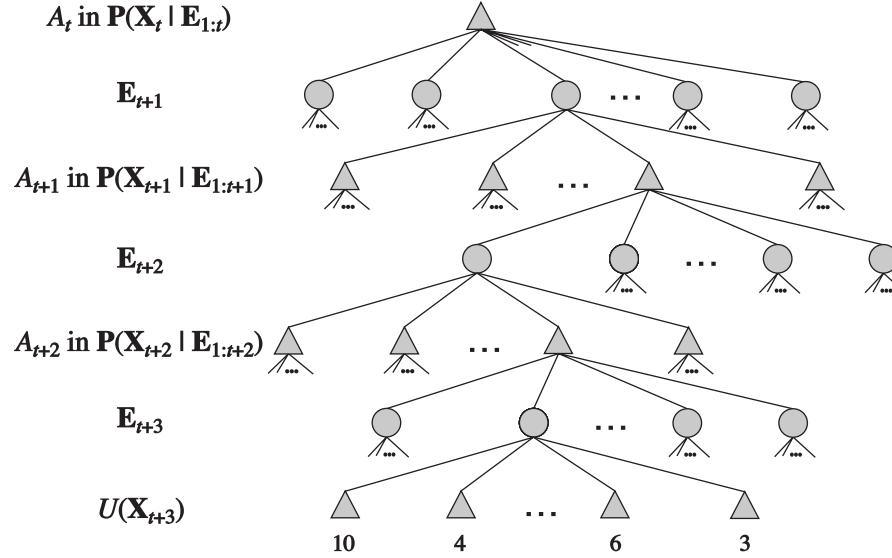
While it is difficult to directly derive π^* , a decision-theoretic agent can be constructed for POMDPs:

- The transition and sensor models are represented by a [dynamic Bayesian network](#);
- The dynamic Bayesian network is extended with decision (A) and utility (R) and U nodes to form a dynamic decision network;
- A [filtering algorithm](#) is used to incorporate each new percept and action and to update the belief state representation;
- Decisions are made by projecting forward possible action sequences and choosing (approximately) the best one, in a manner similar to a truncated [Expectiminimax](#).



At time t , the agent must decide what to do.

- Shaded nodes represent variables with known values.
- The network is unrolled for a finite horizon.
- It includes nodes for the reward of \mathbf{X}_{t+1} and \mathbf{X}_{t+2} , but the (estimated) utility of \mathbf{X}_{t+3} .



Part of the look-ahead solution of the previous decision network:

- Each triangular node is a belief state in which the agent makes a decision.
 - The belief state at each node can be computed by applying a filtering algorithm to the sequence of percepts and actions leading to it.
- The round nodes correspond to choices by the environment.

A decision can be extracted from the search tree by backing up the (estimated) utility values from the leaves, taking the average at the chance nodes and taking the maximum at the decision nodes.

Summary

- Sequential decision problems in uncertain environments, called MDPs, are defined by transition model and a reward function.
- The utility of a state sequence is the sum of all the rewards over the sequence, possibly discounted over time.
 - The solution of an MDP is a policy that associates a decision with every state that the agent might reach.
 - An optimal policy maximizes the utility of the state sequence encountered when it is executed.
- Value iteration and policy iteration can both be used for solving MDPs.
- POMDPs are much more difficult than MDPs. However, a decision-theoretic agent can be constructed for those environments.

Introduction to Artificial Intelligence

Lecture 9: Reinforcement Learning

Prof. Gilles Louppe
g.louppe@uliege.be

Today

How to make decisions under uncertainty, **while learning** about the environment?

- Reinforcement learning (RL)
- Passive RL
 - Model-based estimation
 - Model-free estimation
 - Direct utility estimation
 - Temporal-difference learning
- Active RL
 - Model-based learning
 - Q-Learning
 - Generalizing across states



LEC. 8

Known MDP: Offline Solution

Goal	Technique
Compute V^* , Q^* , π^*	Value / policy iteration
Evaluate a fixed policy π	Policy evaluation

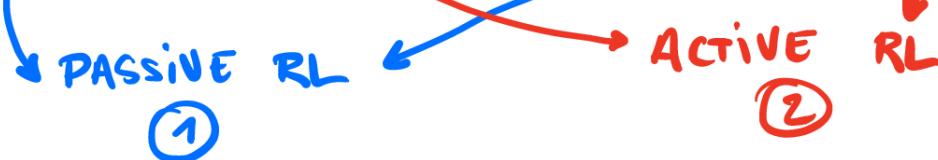
LEC. 9

Unknown MDP: Model-Based

Goal	*use features to generalize	Technique
Compute V^* , Q^* , π^*		VI/PI on approx. MDP
Evaluate a fixed policy π		PE on approx. MDP

Unknown MDP: Model-Free

Goal	*use features to generalize	Technique
Compute V^* , Q^* , π^*		Q-learning
Evaluate a fixed policy π		Value Learning



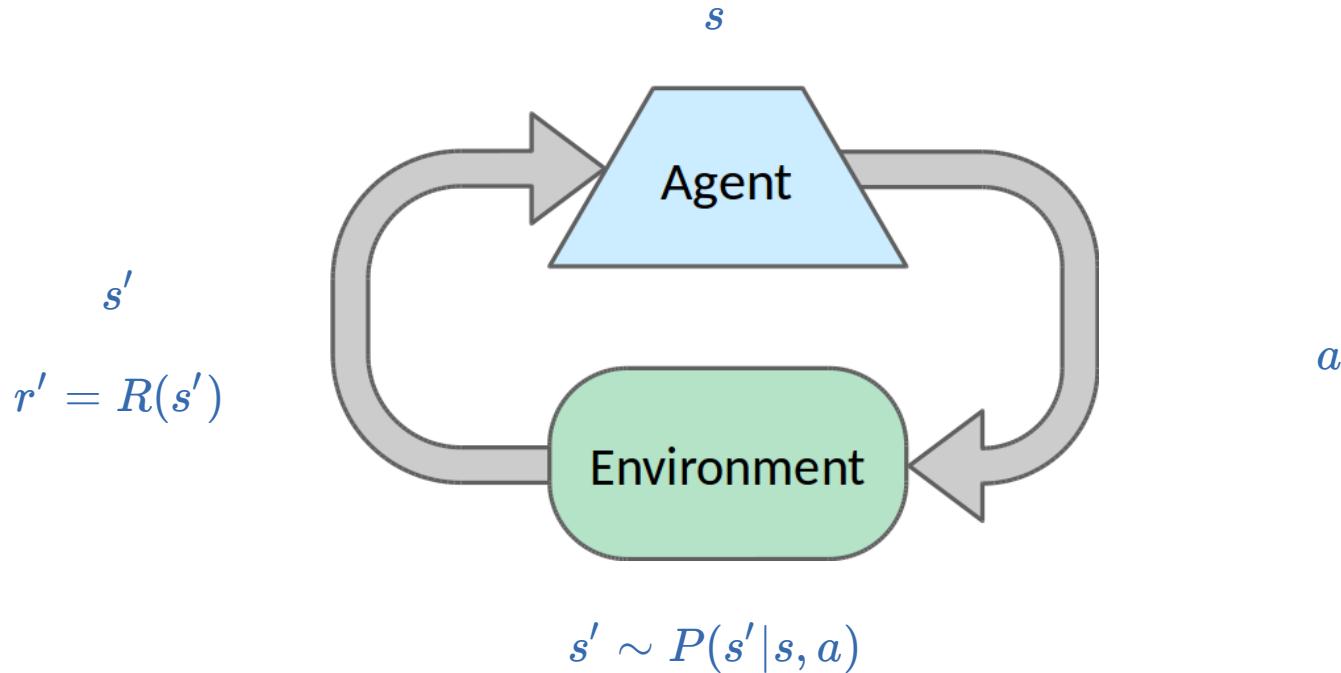
MDPs

A short recap.

MDPs

A **Markov decision process** (MDP) is a tuple $(\mathcal{S}, \mathcal{A}, P, R)$ such that:

- \mathcal{S} is a set of states s ;
- \mathcal{A} is a set of actions a ;
- P is a (stationary) transition model such that $P(s'|s, a)$ denotes the probability of reaching state s' if action a is done in state s ;
- R is a reward function that maps immediate (finite) reward values $R(s)$ obtained in states s .
- ($0 < \gamma \leq 1$ is the discount factor.)



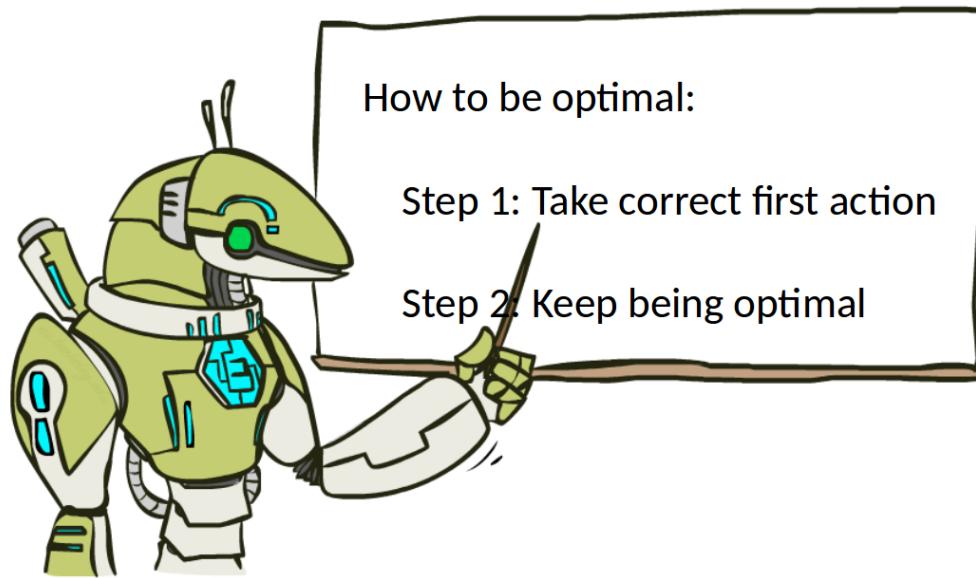
Remarks

- Although MDPs generalize to continuous state-action spaces, we assume in this lecture that both \mathcal{S} and \mathcal{A} are discrete and finite.
- The formalism we use to define MDPs is not unique. A quite well-established and equivalent variant is to define the reward function with respect to a transition (s, a, s') , i.e. $R(s, a, s')$. This results in new (but equivalent) formulations of the algorithms covered in Lecture 8.

The Bellman equation

The utility of a state is the immediate reward for that state, plus the expected discounted utility of the next state, assuming that the agent chooses the optimal action:

$$V(s) = R(s) + \gamma \max_a \sum_{s'} P(s'|s, a)V(s').$$



Value iteration

The **value iteration** algorithm provides a fixed-point iteration procedure for computing the state utilities $V(s)$:

- Let $V_i(s)$ be the estimated utility value for s at the i -th iteration step.
- The **Bellman update** consists in updating simultaneously all the estimates to make them **locally consistent** with the Bellman equation:

$$V_{i+1}(s) = R(s) + \gamma \max_a \sum_{s'} P(s'|s, a) V_i(s').$$

- Repeat until convergence.

Policy iteration

The **policy iteration** algorithm directly computes the policy (instead of state values). It alternates the following two steps:

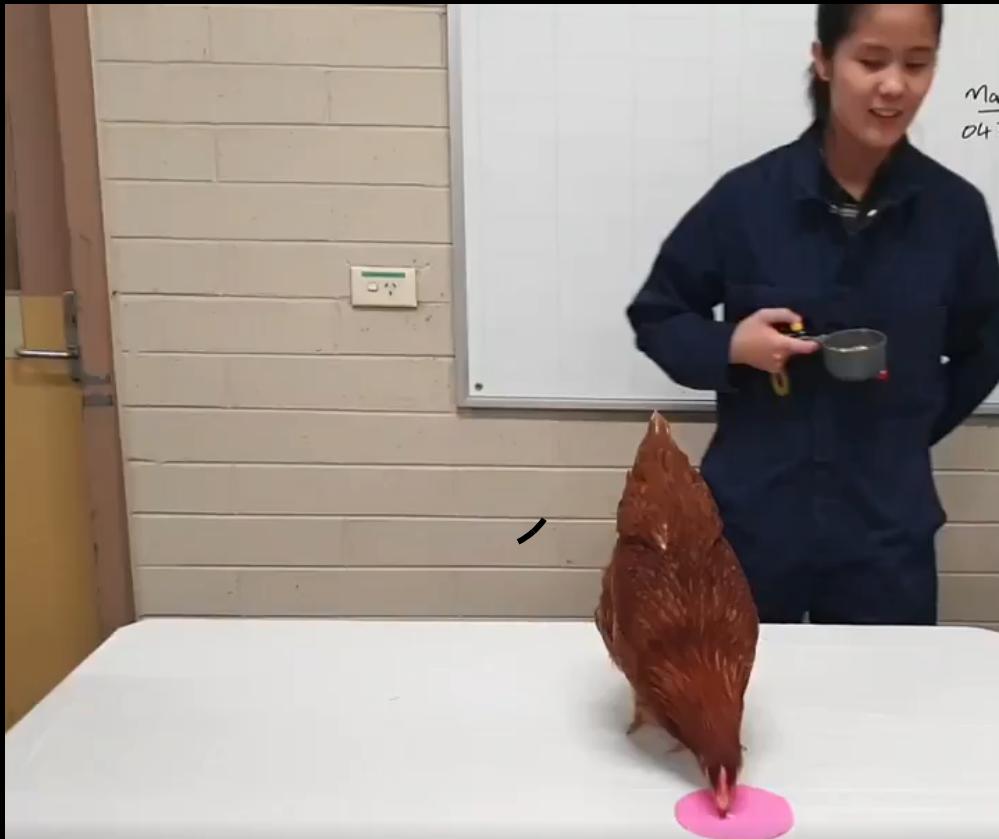
- Policy evaluation: given π_i , calculate $V_i = V^{\pi_i}$, i.e. the utility of each state if π_i is executed:

$$V_i(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi_i(s))V_i(s').$$

- Policy improvement: calculate a new policy π_{i+1} using one-step look-ahead based on V_i :

$$\pi_{i+1}(s) = \arg \max_a \sum_{s'} P(s'|s, a)V_i(s').$$

Reinforcement learning



▶ 0:00 / 0:40





▶ 0:00 / 0:27



What just happened?

- This wasn't planning, it was reinforcement learning!
- There was an MDP, but the chicken couldn't solve it with just computation.
- The chicken needed to actually act to figure it out.

Important ideas in reinforcement learning that came up

- Exploration: you have to try unknown actions to get information.
- Exploitation: eventually, you have to use what you know.
- Regret: even if you learn intelligently, you make mistakes.
- Sampling: because of chance, you have to try things repeatedly.
- Difficult: learning can be much harder than solving a known MDP.

Reinforcement learning

We still assume a Markov decision process $(\mathcal{S}, \mathcal{A}, P, R)$ such that:

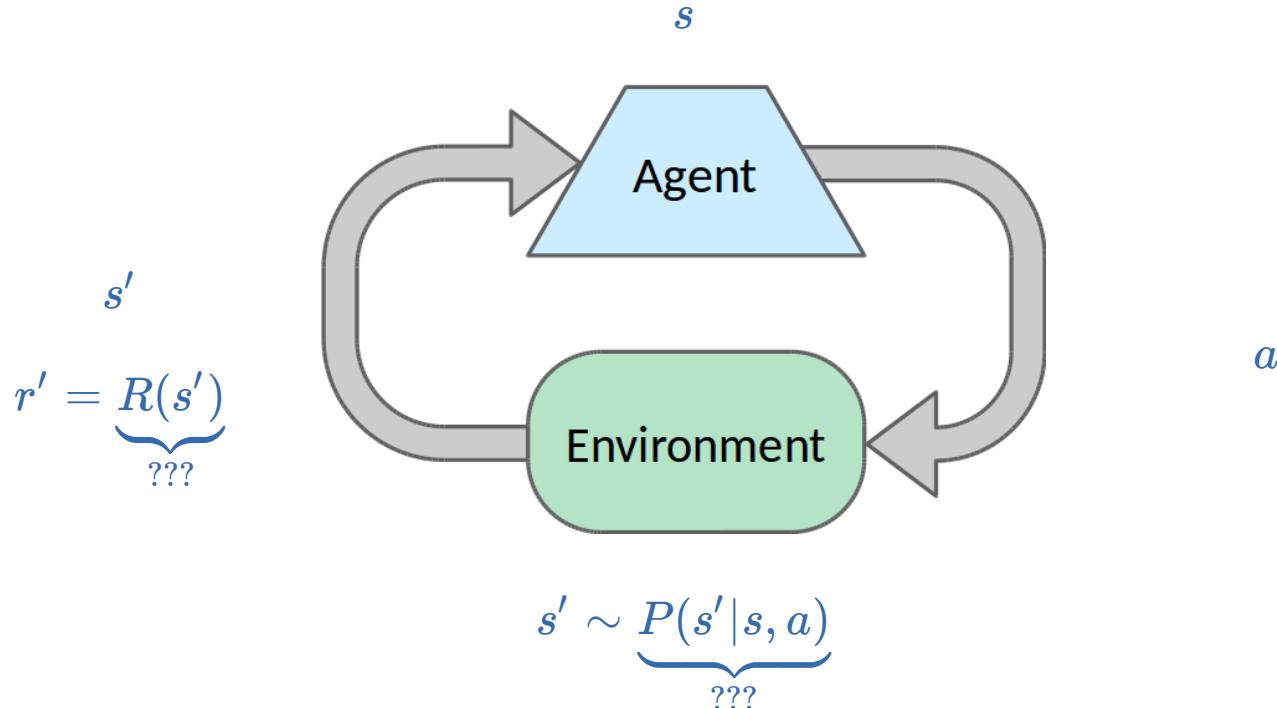
- \mathcal{S} is a set of states s ;
- \mathcal{A} is a set of actions a ;
- P is a (stationary) transition model such that $P(s'|s, a)$ denotes the probability of reaching state s' if action a is done in state s ;
- R is a reward function that maps immediate (finite) reward values $R(s)$ obtained in states s .

Our goal is find the optimal policy $\pi^*(s)$.

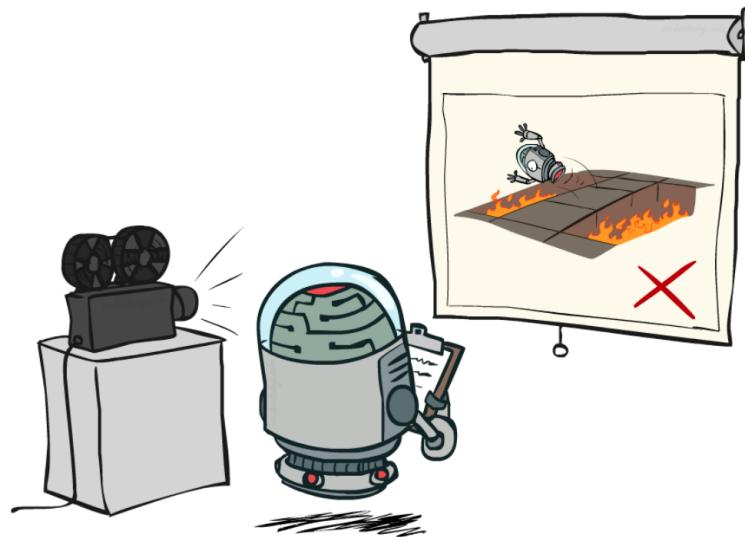
New twist

The transition model $P(s'|s, a)$ and the reward function $R(s)$ are unknown.

- We do not know which states are good nor what actions do!
- We must observe or interact with the environment in order to jointly learn these dynamics and act upon them.

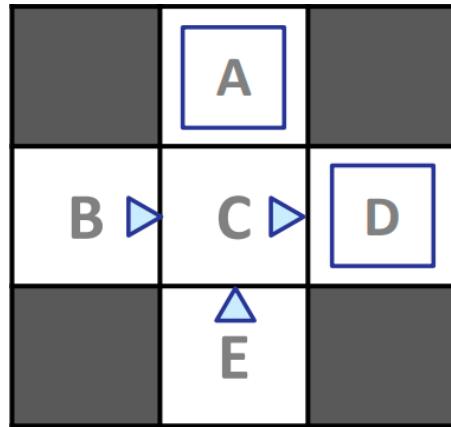


Passive RL



Goal: policy evaluation

- The agent's policy π is fixed.
- Its goal is to learn the utilities $V^\pi(s)$.
- The learner has no choice about what actions to take. It just executes the policy and learns from experience.



The agent executes a set of **trials** (or episodes) in the environment using policy π . Trial trajectories $(s, r, a, s'), (s', r', a', s'')$, ... might look like this:

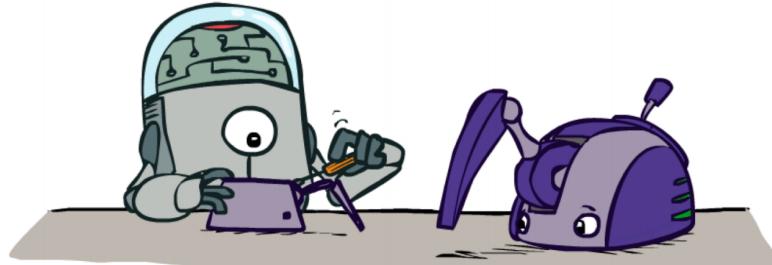
- Trial 1: $(B, -1, \text{east}, C), (C, -1, \text{east}, D), (D, +10, \text{exit}, \perp)$
- Trial 2: $(B, -1, \text{east}, C), (C, -1, \text{east}, D), (D, +10, \text{exit}, \perp)$
- Trial 3: $(E, -1, \text{north}, C), (C, -1, \text{east}, D), (D, +10, \text{exit}, \perp)$
- Trial 4: $(E, -1, \text{north}, C), (C, -1, \text{east}, A), (A, -10, \text{exit}, \perp)$

Model-based estimation

A **model-based** agent estimates approximate transition and reward models \hat{P} and \hat{R} based on experiences and then evaluates the resulting empirical MDP.

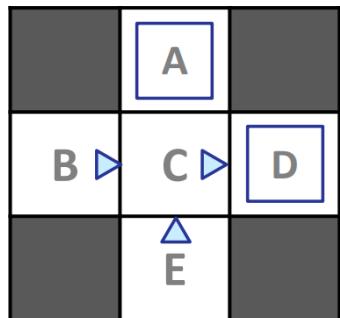
- Step 1: Learn an empirical MDP.
 - Estimate $\hat{P}(s'|s, a)$ from empirical samples (s, a, s') or with supervised learning.
 - Discover each $\hat{R}(s)$ for each s .
- Step 2: Evaluate π using \hat{P} and \hat{R} , e.g. as

$$V(s) = \hat{R}(s) + \gamma \sum_{s'} \hat{P}(s'|s, \pi(s))V(s').$$



Example

Policy π :



Trajectories:

- $(B, -1, \text{east}, C), (C, -1, \text{east}, D), (D, +10, \text{exit}, \perp)$
- $(B, -1, \text{east}, C), (C, -1, \text{east}, D), (D, +10, \text{exit}, \perp)$
- $(E, -1, \text{north}, C), (C, -1, \text{east}, D), (D, +10, \text{exit}, \perp)$
- $(E, -1, \text{north}, C), (C, -1, \text{east}, A), (A, -10, \text{exit}, \perp)$

Learned transition model \hat{P} :

$$\begin{aligned}\hat{P}(C|B, \text{east}) &= 1 \\ \hat{P}(D|C, \text{east}) &= 0.75 \\ \hat{P}(A|C, \text{east}) &= 0.25 \\ (...) \end{aligned}$$

Learned reward \hat{R} :

$$\begin{aligned}\hat{R}(B) &= -1 \\ \hat{R}(C) &= -1 \\ \hat{R}(D) &= +10 \\ (...) \end{aligned}$$

Model-free estimation

Can we learn V^π in a **model-free** fashion, without explicitly modeling the environment, i.e. without learning \hat{P} and \hat{R} ?

Direct utility estimation

(a.k.a. Monte Carlo evaluation)

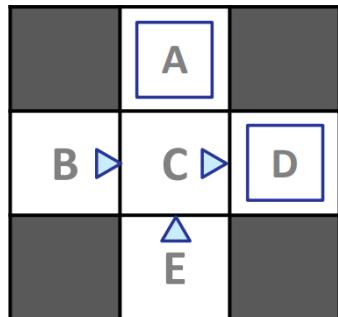
- The utility $V^\pi(s)$ of state s is the expected total reward from the state onward (called the expected **reward-to-go**)

$$V^\pi(s) = \mathbb{E} \left[\sum_{t=0}^{\infty} \gamma^t R(s_t) \right] \Big|_{s_0=s}$$

- Each trial provides a **sample** of this quantity for each state visited.
- Therefore, at the end of each sequence, one can update a sample average $\hat{V}^\pi(s)$ by:
 - computing the observed reward-to-go for each state;
 - updating the estimated utility for that state, by keeping a running average.
- In the limit of infinitely many trials, the sample average will converge to the true expectation.

Example ($\gamma = 1$)

Policy π :



Trajectories:

- $(B, -1, \text{east}, C), (C, -1, \text{east}, D), (D, +10, \text{exit}, \perp)$
- $(B, -1, \text{east}, C), (C, -1, \text{east}, D), (D, +10, \text{exit}, \perp)$
- $(E, -1, \text{north}, C), (C, -1, \text{east}, D), (D, +10, \text{exit}, \perp)$
- $(E, -1, \text{north}, C), (C, -1, \text{east}, A), (A, -10, \text{exit}, \perp)$

Output values

$\hat{V}^\pi(s)$:

	A	-10	
+8	C	+4	+10
B		D	
	E	-2	

If both B and E go to C under π ,
how can their values be different?

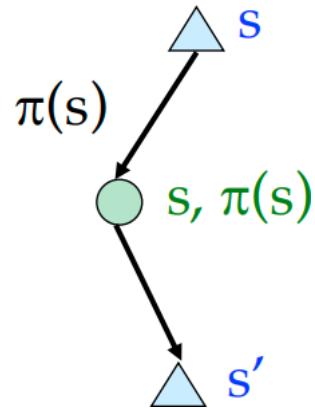
Unfortunately, direct utility estimation misses the fact that the state values $V^\pi(s)$ are not independent, since they obey the Bellman equations for a fixed policy:

$$V^\pi(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi(s)) V^\pi(s').$$

Therefore, direct utility estimation misses opportunities for learning and takes a long time to learn.

Temporal-difference learning

Temporal-difference (TD) learning consists in updating $V^\pi(s)$ each time the agent experiences a transition $(s, r = R(s), a = \pi(s), s')$.



When a transition from s to s' occurs, the temporal-difference update steers $V^\pi(s)$ to better agree with the Bellman equations for a fixed policy, i.e.

$$V^\pi(s) \leftarrow V^\pi(s) + \underbrace{\alpha(r + \gamma V^\pi(s') - V^\pi(s))}_{\text{temporal difference error}}$$

where α is the learning rate parameter.

Alternatively, the TD-update can be viewed as a single gradient descent step on the squared error between the target $r + \gamma V^\pi(s')$ and the prediction $V^\pi(s)$.
(More later.)

Exponential moving average

The TD-update can equivalently be expressed as the exponential moving average

$$V^\pi(s) \leftarrow (1 - \alpha)V^\pi(s) + \alpha(r + \gamma V^\pi(s')).$$

Intuitively,

- this makes recent samples more important;
- this forgets about the past (distant past values were wrong anyway).

Example ($\gamma = 1, \alpha = 0.5$)

	A	0	
B	0	C	D
E	0		
	A	0	
B	-0.5	C	0
E	0		

Transition: $(B, -1, \text{east}, C)$

TD-update:

$$\begin{aligned}V^\pi(B) &\leftarrow V^\pi(B) + \alpha(R(B) + \gamma V^\pi(C) - V^\pi(B)) \\&\leftarrow 0 + 0.5(-1 + 0 - 0) \\&\leftarrow -0.5\end{aligned}$$

		0	
	A		
-0.5	B	C	D
		•	
		0	
	E		

		0	
	A		
-0.5	B	C	D
		•	
		3.5	
		8	•
		0	
	E		

Transition: $(C, -1, \text{east}, D)$

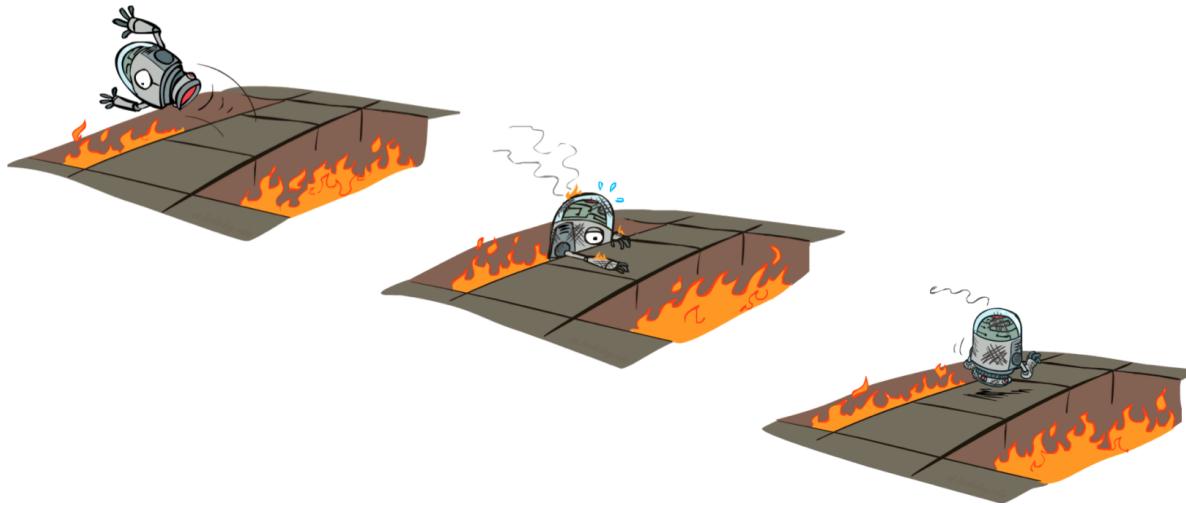
TD-update:

$$\begin{aligned}
 V^\pi(C) &\leftarrow V^\pi(C) + \alpha(R(C) + \gamma V^\pi(D) - V^\pi(C)) \\
 &\leftarrow 0 + 0.5(-1 + 8 - 0) \\
 &\leftarrow 3.5
 \end{aligned}$$

Convergence

- Notice that the TD-update involves only the observed successor s' , whereas the actual Bellman equations for a fixed policy involves all possible next states. Nevertheless, the **average** value of $V^\pi(s)$ will converge to the correct value.
- If we change α from a fixed parameter to a function that decreases as the number of times a state has been visited increases, then $V^\pi(s)$ will itself converge to the correct value.

Active RL



Goal: learn an optimal policy

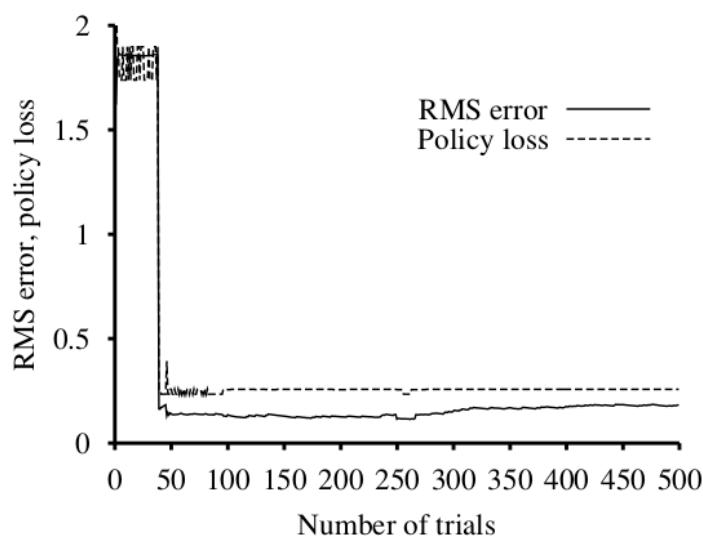
- The agent's policy is not fixed anymore.
- Its goal is to learn the optimal policy π^* or the state values $V(s)$.
- The learner makes choices!
- Fundamental trade-off: exploration vs. exploitation.

Model-based learning

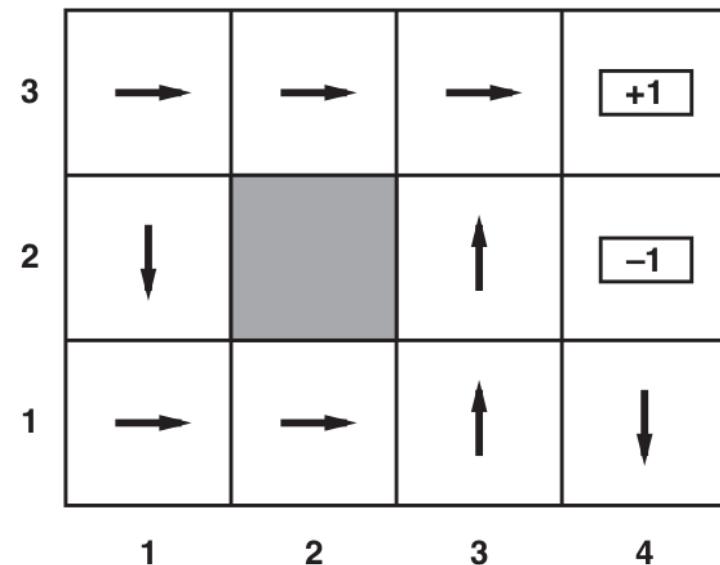
The passive model-based agent can be made active by instead finding the optimal policy π^* for the empirical MDP.

For example, having obtained a utility function V that is optimal for the learned model (e.g., with Value Iteration), the optimal action by one-step look-ahead to maximize the expected utility is

$$\pi^*(s) = \arg \max_a \sum_{s'} \hat{P}(s'|s, a)V(s').$$



(a)



(b)

Figure 21.6 Performance of a greedy ADP agent that executes the action recommended by the optimal policy for the learned model. (a) RMS error in the utility estimates averaged over the nine nonterminal squares. (b) The suboptimal policy to which the greedy agent converges in this particular sequence of trials.

The agent **does not** learn the true utilities or the true optimal policy!

The resulting policy is **greedy** and **suboptimal**:

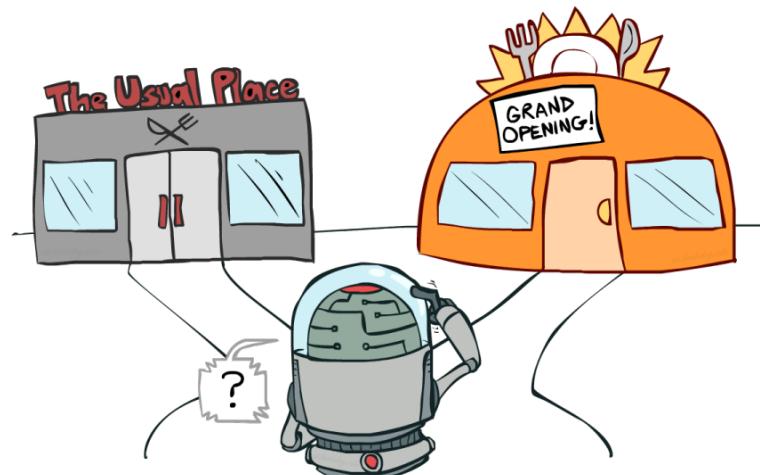
- The learned transition and reward models \hat{P} and \hat{R} are not the same as the true environment.
- Therefore, what is optimal in the learned model can be suboptimal in the true environment.

Exploration

Actions do more than provide rewards according to the current learned model. They also contribute to learning the true environment.

This is the **exploitation-exploration** trade-off:

- Exploitation: follow actions that maximize the rewards, under the current learned model;
- Exploration: follow actions to explore and learn about the true environment.



How to explore?

Simplest approach for forcing exploration: random actions (ϵ -greedy).

- With a (small) probability ϵ , act randomly.
- With a (large) probability $(1 - \epsilon)$, follow the current policy.

ϵ -greedy does eventually explore the space, but keeps trashing around once learning is done.

When to explore?

Better idea: explore areas whose badness is not (yet) established, then stop exploring.

Formally, let $V^+(s)$ denote an optimistic estimate of the utility of state s and let $N(s, a)$ be the number of times actions a has been tried in s .

For Value Iteration, the update equation becomes

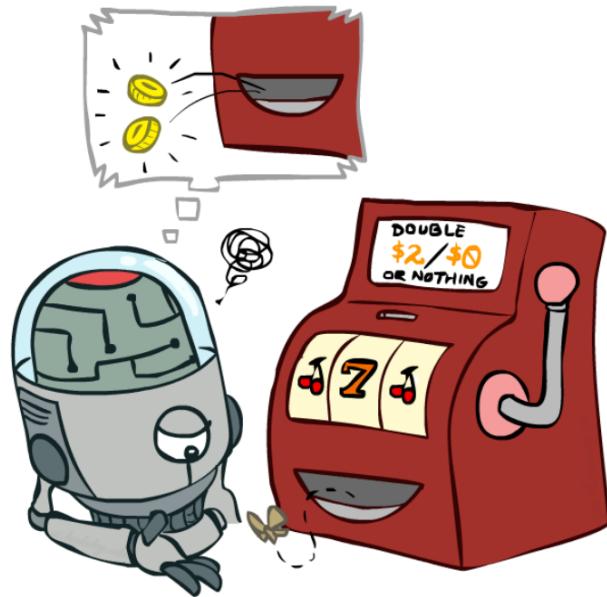
$$V_{i+1}^+(s) = R(s) + \gamma \max_a f\left(\sum_{s'} P(s'|s, a) V_i^+(s'), N(s, a)\right),$$

where $f(v, n)$ is called the exploration function.

The function $f(v, n)$ should be increasing in v and decreasing in n . A simple choice is $f(v, n) = v + K/n$.

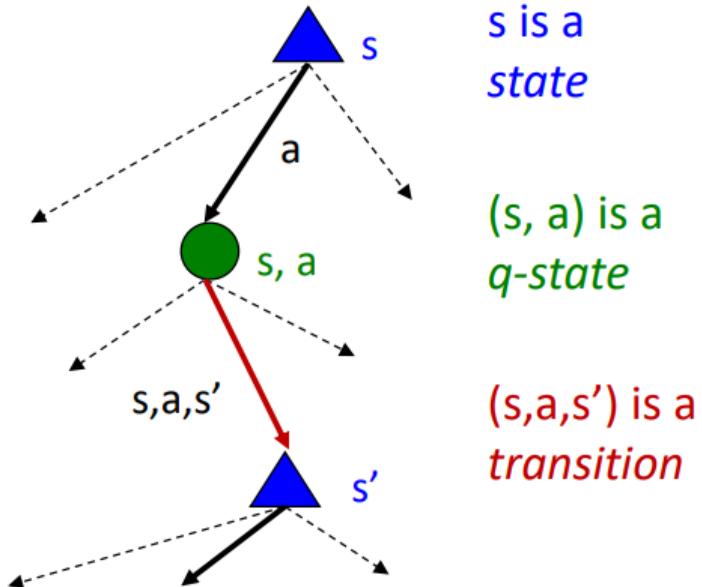
Model-free learning

Although temporal difference learning provides a way to estimate V^π in a model-free fashion, we would still have to learn a model $P(s'|s, a)$ to choose an action based on a one-step look-ahead.



Détour: Q-values

- The state-value $V(s)$ of the state s is the expected utility starting in s and acting optimally.
- The state-action-value $Q(s, a)$ of the q-state (s, a) is the expected utility starting out having taken action a from s and thereafter acting optimally.



Optimal policy

The optimal policy $\pi^*(s)$ can be defined in terms of either $V(s)$ or $Q(s, a)$:

$$\begin{aligned}\pi^*(s) &= \arg \max_a \sum_{s'} P(s'|s, a)V(s') \\ &= \arg \max_a Q(s, a)\end{aligned}$$

Bellman equations for Q

Since $V(s) = \max_a Q(s, a)$, the Q-values $Q(s, a)$ are recursively defined as

$$\begin{aligned} Q(s, a) &= R(s) + \gamma \sum_{s'} P(s'|s, a) V(s') \\ &= R(s) + \gamma \sum_{s'} P(s'|s, a) \max_{a'} Q(s', a'). \end{aligned}$$

As for value iteration, the last equation can be used as an update equation for a fixed-point iteration procedure that calculates the Q-values $Q(s, a)$. However, it still requires knowing $P(s'|s, a)$!

Q-Learning

The state-action-values $Q(s, a)$ can be learned in a model-free fashion using a temporal-difference method known as **Q-Learning**.

Q-Learning consists in updating $Q(s, a)$ each time the agent experiences a transition $(s, r = R(s), a, s')$.

The update equation for TD Q-Learning is

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

Since $\pi^*(s) = \arg \max_a Q(s, a)$, a TD agent that learns Q-values does not need a model of the form $P(s'|s, a)$, neither for learning nor for action selection!

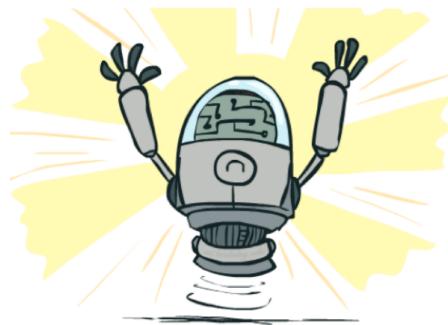
```

function Q-LEARNING-AGENT(percept) returns an action
  inputs: percept, a percept indicating the current state  $s'$  and reward signal  $r'$ 
  persistent:  $Q$ , a table of action values indexed by state and action, initially zero
     $N_{sa}$ , a table of frequencies for state-action pairs, initially zero
     $s, a, r$ , the previous state, action, and reward, initially null

  if TERMINAL?( $s$ ) then  $Q[s, \text{None}] \leftarrow r'$ 
  if  $s$  is not null then
    increment  $N_{sa}[s, a]$ 
     $Q[s, a] \leftarrow Q[s, a] + \alpha(N_{sa}[s, a])(r + \gamma \max_{a'} Q[s', a'] - Q[s, a])$ 
     $s, a, r \leftarrow s', \text{argmax}_{a'} f(Q[s', a'], N_{sa}[s', a']), r'$ 
  return  $a$ 

```

Figure 21.8 An exploratory Q-learning agent. It is an active learner that learns the value $Q(s, a)$ of each action in each situation. It uses the same exploration function f as the exploratory ADP agent, but avoids having to learn the transition model because the Q-value of a state can be related directly to those of its neighbors.



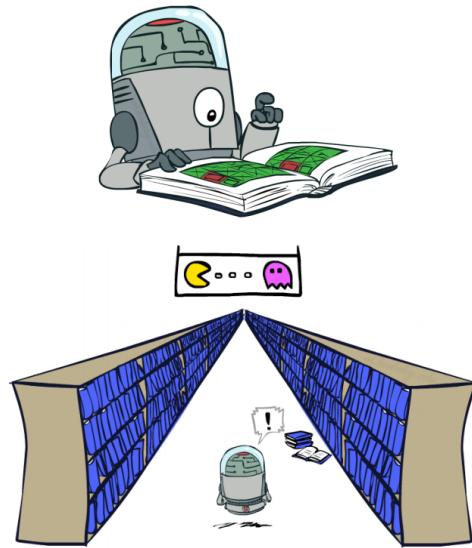
Convergence

Q-Learning converges to an optimal policy, even when acting suboptimally.

- This is called off-policy learning.
- Technical caveats:
 - You have to explore enough.
 - The learning rate must eventually become small enough.
 - ... but it shouldn't decrease too quickly.

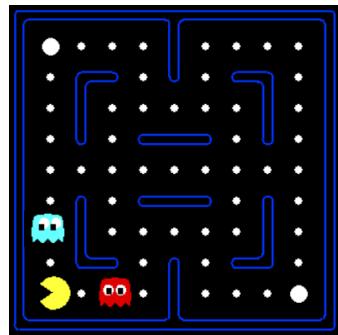
Generalizing across states

- Basic Q-Learning keeps a table for all Q-values $Q(s, a)$.
- In realistic situations, we cannot possibly learn about every single state!
 - Too many states to visit them all in training.
 - Too many states to hold the Q-table in memory.
- We want to generalize:
 - Learn about some small number of training states from experience.
 - Generalize that experience to new, similar situations.
 - This is supervised [machine learning](#) again!



Example: Pacman

(a)



(b)



(c)



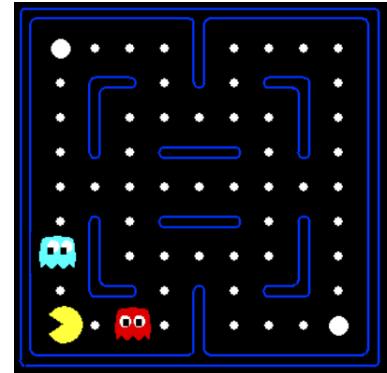
If we discover by experience that (a) is bad, then in naive Q-Learning, we know nothing about (b) nor (c)!

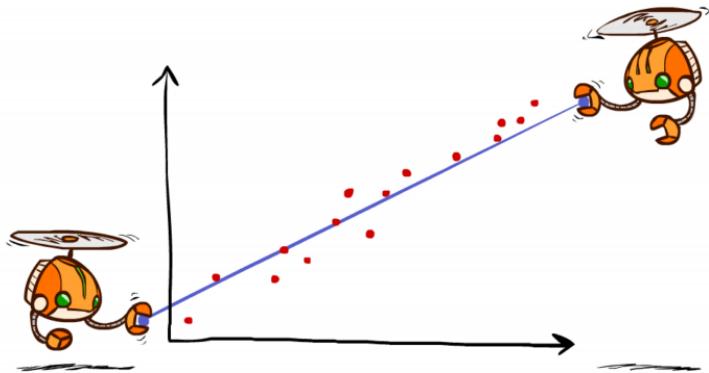
Feature-based representations

Solution: describe a state s using a vector

$$\mathbf{x} = [f_1(s), \dots, f_d(s)] \in \mathbb{R}^d$$
 of features.

- Features are functions f_k from states to real numbers that capture important properties of the state.
- Example features:
 - Distance to closest ghost
 - Distance to closest dot
 - Number of ghosts
 - ...
- Can similarly describe a q-state (s, a) with features $f_k(s, a)$.





Approximate Q-Learning

Using a feature-based representation, the Q-table can now be replaced with a function approximator, such as a linear model:

$$Q(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \dots + w_d f_d(s, a).$$

Upon the transition (s, r, a, s') , the update becomes

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

for all w_k .

In linear regression, imagine we had only one point \mathbf{x} with features $[f_1, \dots, f_d]$. Then,

$$\begin{aligned}\ell(\mathbf{w}) &= \frac{1}{2} \left(y - \sum_k w_k f_k \right)^2 \\ \frac{\partial \ell}{\partial w_k} &= - \left(y - \sum_k w_k f_k \right) f_k \\ w_k &\leftarrow w_k + \alpha \left(y - \sum_k w_k f_k \right) f_k,\end{aligned}$$

hence the Q-update

$$w_k \leftarrow w_k + \alpha \left(\underbrace{r + \gamma \max_{a'} Q(s', a')}_{\text{target } y} - \underbrace{Q(s, a)}_{\text{prediction}} \right) f_k(s, a).$$

DQN

Similarly, the Q-table can be replaced with a neural network as function approximator, resulting in the [DQN](#) algorithm.

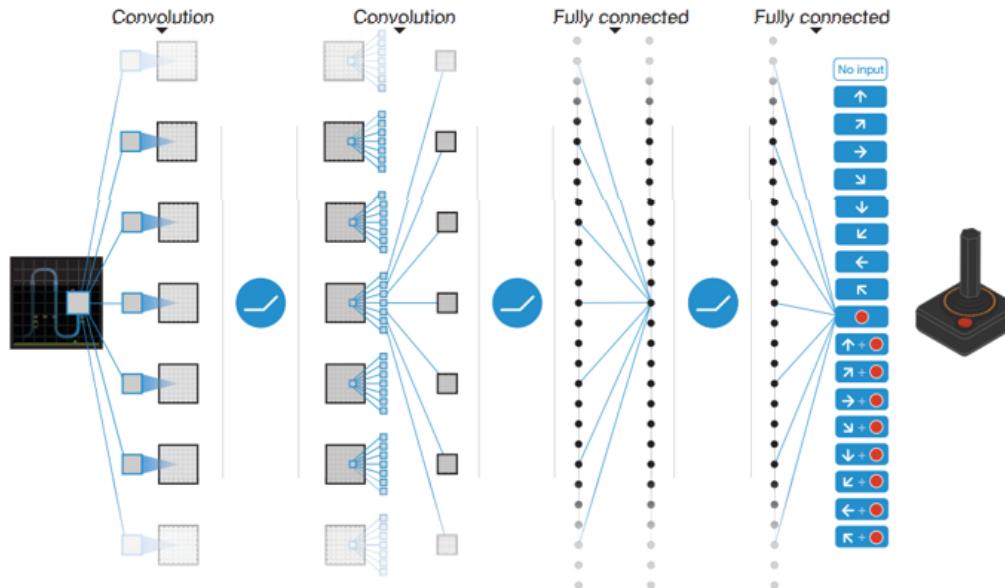


Figure 1 | Schematic illustration of the convolutional neural network. The details of the architecture are explained in the Methods. The input to the neural network consists of an $84 \times 84 \times 4$ image produced by the preprocessing map ϕ , followed by three convolutional layers (note: snaking blue line

symbolizes sliding of each filter across input image) and two fully connected layers with a single output for each valid action. Each hidden layer is followed by a rectifier nonlinearity (that is, $\max(0, x)$).

DQN Network Architecture

(demo)

Applications

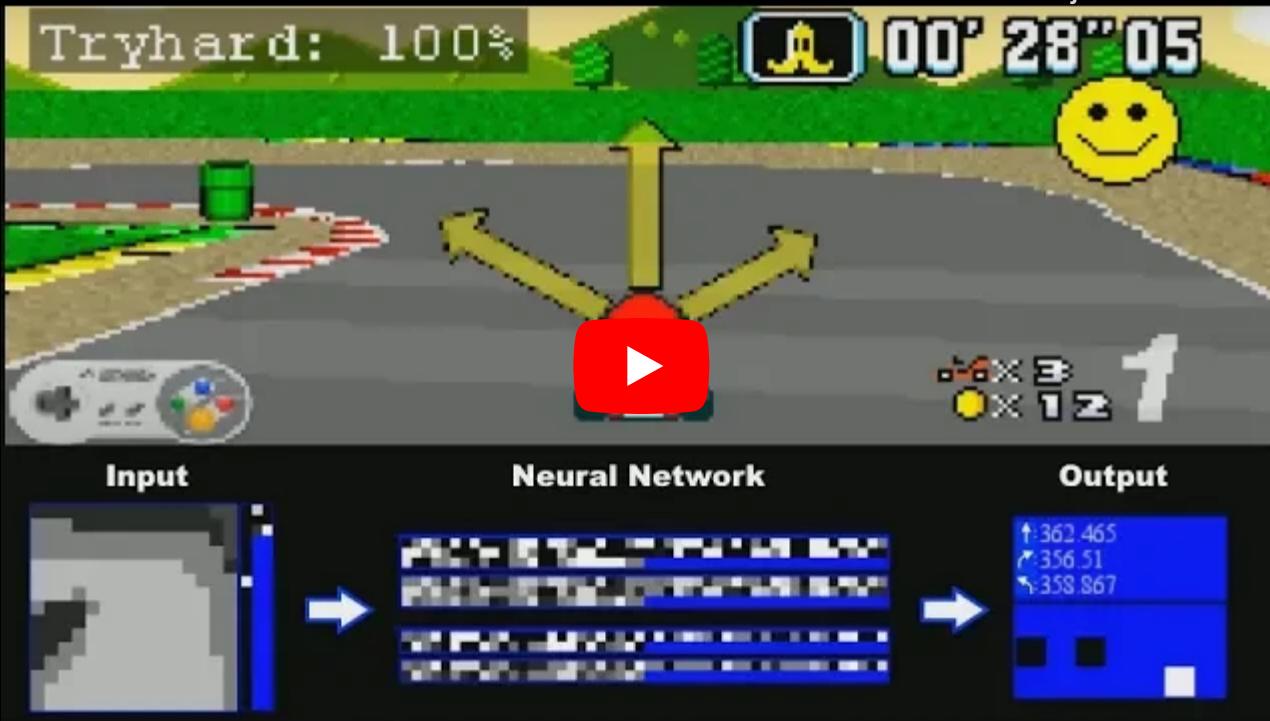


MarlQ -- Q-Learning Neural Network for Mario Ka...

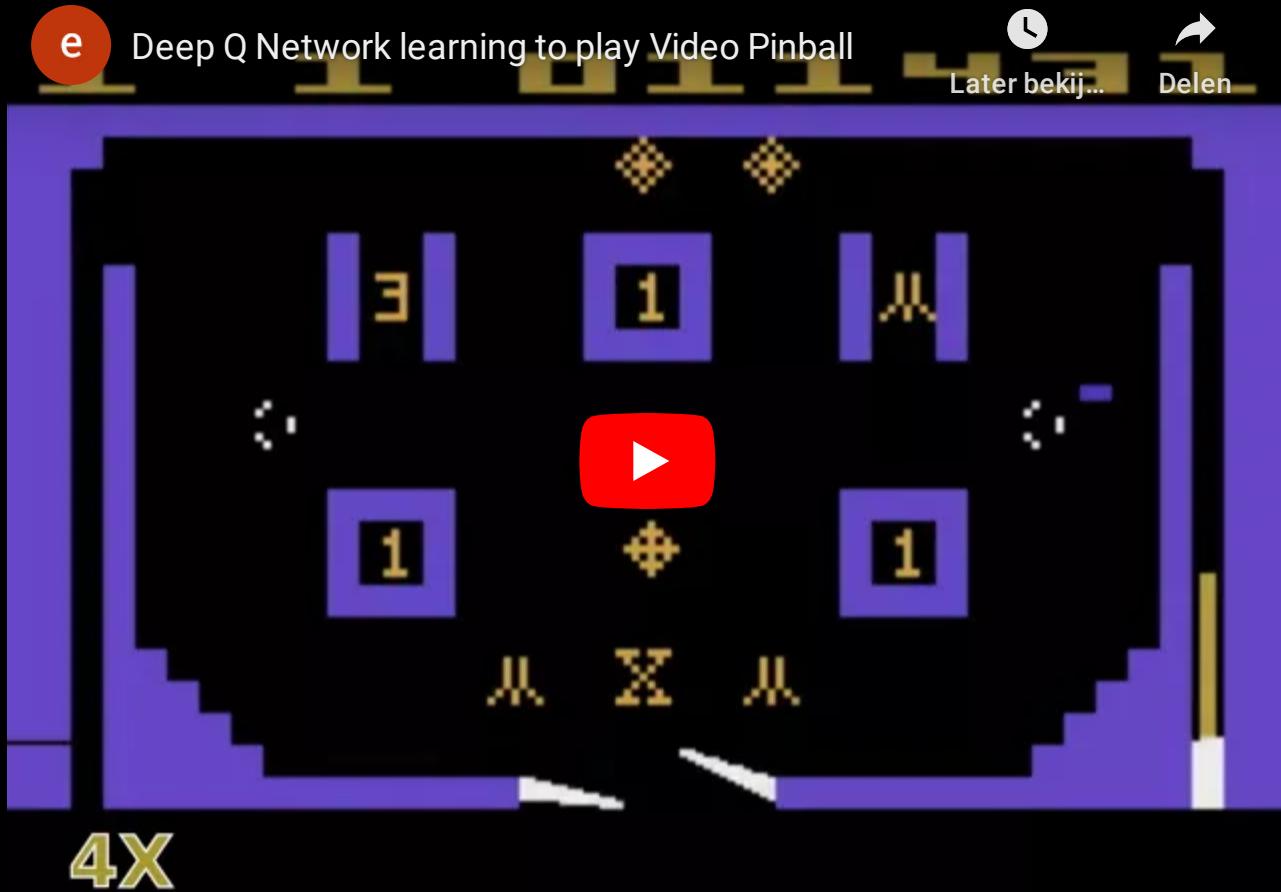


Later bekij...

Delen



MarlQ



Playing Atari Games (Pinball)



QT-Opt: Scalable Deep Reinforcement Learning f...

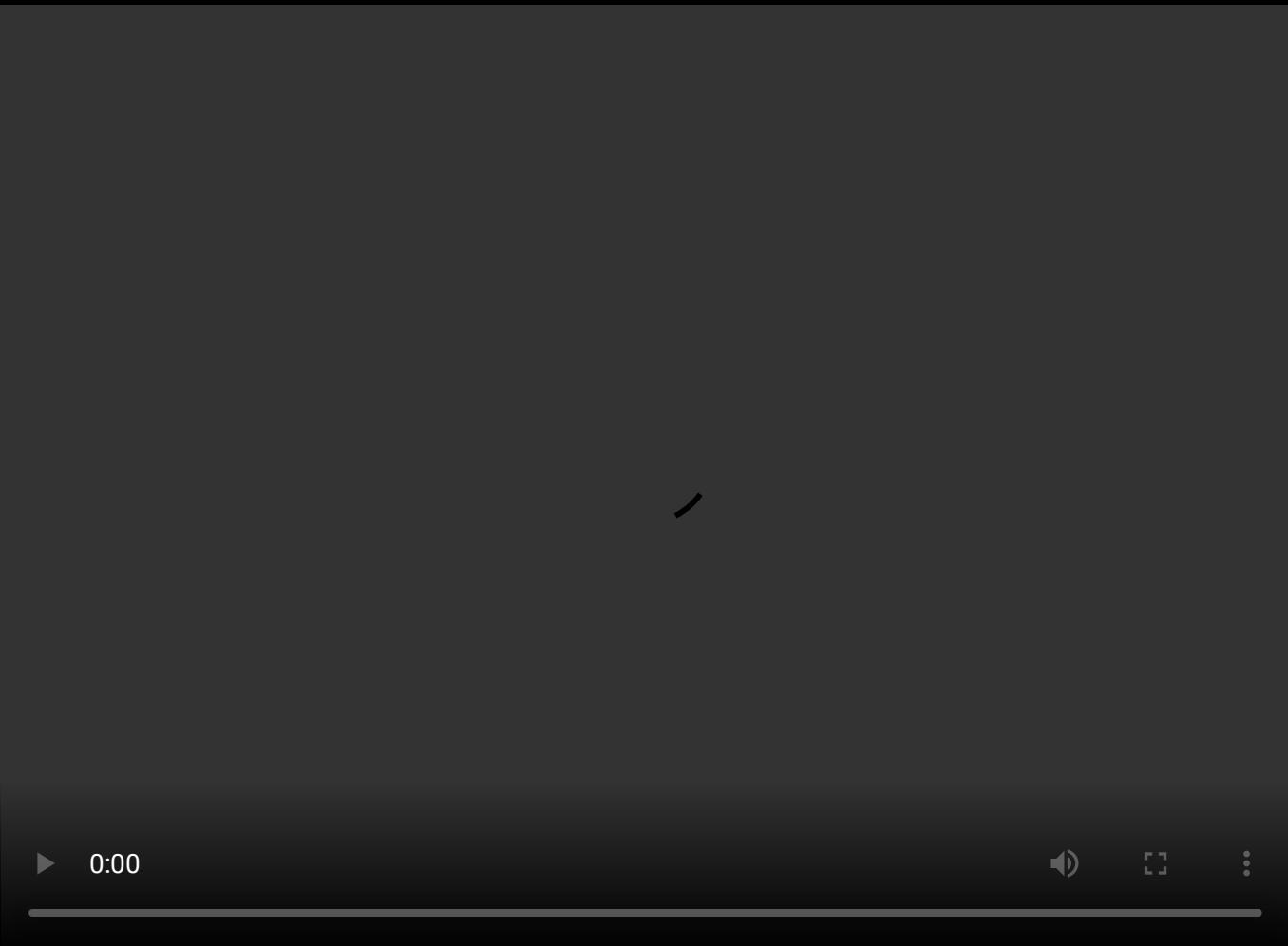


Later bekij...

Delen



Robotic manipulation (I)



Robotic manipulation (II)

Summary

Known MDP: Offline Solution *LEC. 8*

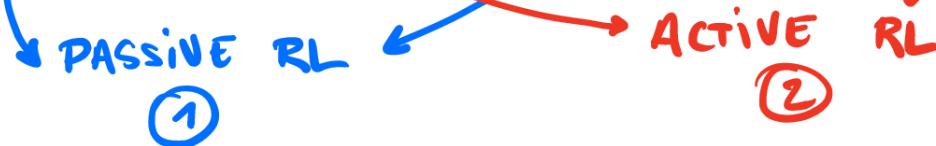
Goal	Technique
Compute V^* , Q^* , π^*	Value / policy iteration
Evaluate a fixed policy π	Policy evaluation

Unknown MDP: Model-Based *LEC. 9*

Goal	*use features to generalize	Technique
Compute V^* , Q^* , π^*		VI/PI on approx. MDP
Evaluate a fixed policy π		PE on approx. MDP

Unknown MDP: Model-Free *LEC. 9*

Goal	*use features to generalize	Technique
Compute V^* , Q^* , π^*		Q-learning
Evaluate a fixed policy π		Value Learning



My mission ✓

By the end of this course, you will have built autonomous agents that efficiently make decisions in fully informed, partially observable and adversarial settings. Your agents will draw inferences in uncertain and unknown environments and optimize actions for arbitrary reward structures.

