Artificial Intelligence

Didier Lime

École Centrale de Nantes – LS2N

Last modification: May 15, 2024

Outline

Outline

Outline

Thinking and Acting Humanly

- Assuming humans are the embodiment of intelligence, an intelligent agent should maybe act and think **like humans**;
- This is useful in defining what an intelligent agent should be able to do;
- It also inspiring for finding ways to solve these tasks automatically.

The Turing Test

- The "imitation game" proposed by Alan Turing in 1950;
- A machine has a 5 minutes long written conversation with a human interrogator;
- It passes the test if the interrogator mistakes it for a person 30% of the time;
- In the **total** Turing test, the interrogator can test the visual perception of the machine and pass objects through a hatch.
- No machine has ever passed the test (though some did in restricted settings).

Acting Rationally

- Alternatively, we can consider a machine is intelligent if it thinks and acts rationally;
- It makes correct inferences;
- It acts so as to always produce the best outcome (up to some measure of utility);
- This a pragmatic approach that does not enforce a predefined form of intelligence;
- It is more amenable to mathematical analysis.
- John Von Neumann said (cited by E. T. Jaynes):

  > *You insist that there is something a machine cannot do. If you will tell me precisely what it is that a machine cannot do, then I can always make a machine which will do just that!*

Strong and Weak AI

- Some philosophers make a difference between **weak AI** and **strong AI**;
- In weak AI, machines can act *as if* they were intelligent;
- In strong AI, machines *actually think*;
- The difference is mostly **irrelevant** to AI researchers.
- Turing's response:

  > *Instead of arguing continually over this point, it is usual to have the polite convention that everyone thinks.*

Artificial Intelligence as a Field of Research

- Artificial intelligence is a relatively recent field of research (1950's);
- It regroups lots of (very) different subfields;
- It draws upon philosophy, mathematics, economics, neuroscience, psychology, computer science, control theory, linguistics, etc.

Outline

Turing's "Computing Machinery and Intelligence" article

- Published in 1950;
- Puts the objective on making "intelligent" **digital machines**;
- Introduced many of the key concepts of AI:
  - the Turing test;
  - machine learning;
  - genetic algorithms;
  - reinforcement learning.

The Dartmouth Conference

- "A 2 month, 10 man study of artificial intelligence (...) during the summer of 1956 at Dartmouth College in Hanover, New Hampshire"
- Included Marvin Minsky, Claude Shannon, John McCarthy and many others.
- A basic theorem prover "Logic Theorist" was introduced at the occasion.
- The founding event of AI:
  - emphasises duplicating human faculties;
  - sets it apart from others research fields.

Expert Systems

- Initial successes (simple geometry prover, self-taught checkers program, LISP, perceptrons, etc.) until the late 1960's
- Disappointment and cut of funding from 1966 to 1973
- The first big industrial successes of AI is expert systems (1970s-1980s):
  - logical inference (if - then rules);
  - extensively use domain-specific knowledge;
  - Prolog;
  - dozens of installed systems in big companies.

Deep Blue

- 1997: Chess world-champion Gary Kasparov lost to IBM's Deep Blue in a 6-game match;
- Program working on dedicated hardware;
- Evaluated 200 millions moves per seconds;
- Nowadays purely software chess programs routinely perform at this level.

Autonomous Cars

- A very old problem (1920's, remote control);
- 1990's: First successes: semi-autonomous driving
- 2000's: Fully autonomous driving in deserts
- 2010's: Fully autonomous driving in cities (still with many limitations)
- Now a major topic for all the car industry.

(Image) Classification

- Programs now perform classification tasks at or above human-level;
- Examples include ImageNet Large-Scale Visual Recognition challenge;
- Error improved from 25% in 2011 to 16% in 2012 with deep neural network based approaches;
- Now at less than 5%

Alpha Go

- 2015: first victory over a Go professional without handicap;
- 2017: first victory over a top-rated player;
- Used deep neural networks for move and position evaluation;
- Used Monte Carlo tree search variants for move tree exploration.

ChatGPT

- 2017: *Transformers*: « Attention is all you need »;
- 2022: ChatGPT;
- Basé sur un *Large Language Model* : GPT3.5 puis GPT4;
- GPT = *Generative Pre-trained Transformer*;
- Architecture de décodeur avec couches d'attention;
- GPT3.5: 175 miliards de paramètres; GPT4 ?
- Apprentissage supervisé + apprentissage par renforcement.

Outline

Structure of a Rational Artificial Agent

- An agent has **sensors** and **actuators** to operate with its **environment**;
- It has some internal **memory**;
- Within the memory, possibly a **model** of the environment;
- It can use all this to find out:
  - how the environment evolves by itself;
  - the outcomes of the agents' actions;
  - the "best" action.
- A **performance evaluation** module;
- A **learning** module.

The Notion of Environment

- The environment can be many things:
  - The real-world;
  - A mathematical construct;
  - Computer programs, etc.;
- It has several key properties (in relation with the capabilities of the agent):
  - Observability;
  - Uncertainty;
  - Presence of other agents;
  - Static or dynamic;
  - Discrete or continuous;
  - Known or not.

Taking Decisions

- To find out the best action, the agent can:
  - Use only the current state (based on the sensor inputs) and:
    - a database of good decisions;
    - or a learned representation of good decisions.
  - Use a model of the environment and plan optimally over several actions;
  - Ideally a mix of the two (e. g. AlphaZero).
- These involve tradeoffs between computational power/memory and accuracy;
- The complexity of the models (for learning or planning) is part of this tradeoff.

Learning

- There are many opportunities for an agent to **learn**:

Learning

- There are many opportunities for an agent to **learn**:
  - Model (transitions, probabilities, rewards, objectives);

Learning

- There are many opportunities for an agent to **learn**:
  - Model (transitions, probabilities, rewards, objectives);
  - Utility functions (states or actions);

Learning

- There are many opportunities for an agent to **learn**:
  - Model (transitions, probabilities, rewards, objectives);
  - Utility functions (states or actions);
  - Strategies.

Learning

- There are many opportunities for an agent to **learn**:
    - Model (transitions, probabilities, rewards, objectives);
    - Utility functions (states or actions);
    - Strategies.
- Different types of **feedback** are possible to help:

Learning

- There are many opportunities for an agent to **learn**:
  - Model (transitions, probabilities, rewards, objectives);
  - Utility functions (states or actions);
  - Strategies.
- Different types of **feedback** are possible to help:
  - Unsupervised learning: try to find patterns in data (e.g. clustering);

Learning

- There are many opportunities for an agent to **learn**:
  - Model (transitions, probabilities, rewards, objectives);
  - Utility functions (states or actions);
  - Strategies.
- Different types of **feedback** are possible to help:
  - Unsupervised learning: try to find patterns in data (e.g. clustering);
  - Reinforcement learning: use rewards and punishments;

Learning

- There are many opportunities for an agent to **learn**:
  - Model (transitions, probabilities, rewards, objectives);
  - Utility functions (states or actions);
  - Strategies.
- Different types of **feedback** are possible to help:
  - Unsupervised learning: try to find patterns in data (e.g. clustering);
  - Reinforcement learning: use rewards and punishments;
  - Supervised learning: use positive and negative examples.

Outline

Introduction

Optimal Strategies in Deterministic Environments
- The Simple Case
- Path Finding
- Optimal Decisions

Non-Deterministic Environments

Uncertain (Probabilistic) Environments

Accounting for other Agents

Supervised Learning

Conclusion

Outline

Modelling successive states

- The agent should be able to evaluate the **outcomes** of its actions;
  - We need to model the different **states** the system can be in;
  - And to model how we **move** from one state to another;
- A basic model for that is the **directed graph**.

Example: A Sliding Puzzle

1	2	3
4		5
7	8	6

Modelling successive states

- The agent should be able to evaluate the **outcomes** of its actions;
  - We need to model the different **states** the system can be in;
  - And to model how we **move** from one state to another;
- A basic model for that is the **directed graph**.

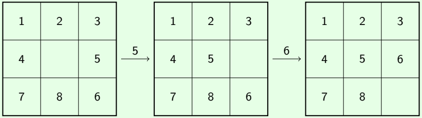
Example: A Sliding Puzzle

$$\begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 4 & & 5 \\ \hline 7 & 8 & 6 \\ \hline \end{array} \xrightarrow{5} \begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 4 & 5 & \\ \hline 7 & 8 & 6 \\ \hline \end{array}$$

Modelling successive states

- The agent should be able to evaluate the **outcomes** of its actions;
  - We need to model the different **states** the system can be in;
  - And to model how we **move** from one state to another;
- A basic model for that is the **directed graph**.

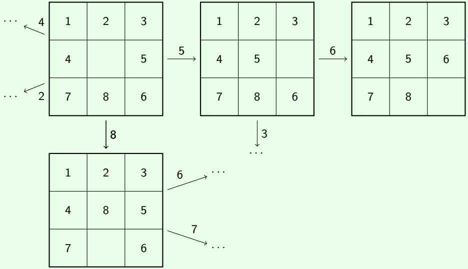
Example: A Sliding Puzzle



Modelling successive states

- The agent should be able to evaluate the **outcomes** of its actions;
  - We need to model the different **states** the system can be in;
  - And to model how we **move** from one state to another;
- A basic model for that is the **directed graph**.

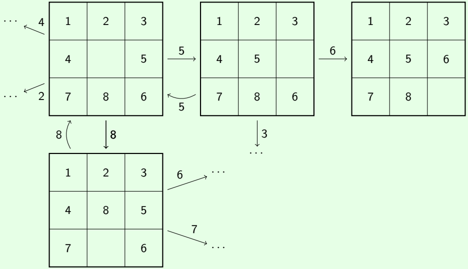
Example: A Sliding Puzzle



Modelling successive states

- The agent should be able to evaluate the **outcomes** of its actions;
  - We need to model the different **states** the system can be in;
  - And to model how we **move** from one state to another;
- A basic model for that is the **directed graph**.

Example: A Sliding Puzzle



(Labeled) Directed Graphs

Definition (Directed Graph)

A **directed graph** is a pair (V, E) where:

- V is a set of **vertices**;
- $E \subseteq V \times V$ is a set of **edges**.

- There is an edge from v to v', and we write $v \to v'$, when $(v, v') \in E$;
- We can label nodes with a function $V \to \Lambda$ for some label set Λ;
- We can label edges with a function $E \to \Sigma$ for some label set Σ.

(Labeled) Directed Graphs: Exercise

Exercise

A farmer, a wolf, a goat, and a cabbage are on one side of a river that they must cross. There is a boat that can be operated only by the farmer, and with only two places. When the farmer is not there, the wolf will eat the goat, and the goat will eat the cabbage.

1. Draw a directed graph modelling this problem. You should not expand the nodes in which the “game” is won or lost;
2. Give a strategy for the farmer allowing everyone to safely cross the river.

Continuous Environments

- For some systems the environment is intrinsically **continuous**;
- We can use the previous approach by **discretising** the continuous variables:
- For instance: replace a continuous variable $x \in [0, 30]$ by a discrete variable with three different discrete values corresponding to: $0 \leq x < 10$, $10 \leq x < 20$ and $20 \leq x \leq 30$.
- Another possibility is to use more expressive formalisms: e.g. **timed automata** or **hybrid automata**.

Achieving a (distant) goal

- A **goal** for some agent, is to make the system reach some beneficial state of the system;
- To achieve this we could move **at random** but:
  - This can take a very long (even infinite!) time;
  - The agent might first reach forbidden states.
- We need to **plan** ahead and find a **strategy** to get the goal in the **model** of the system.

Strategies

Definition (Strategy)

A **strategy** is a function $f : V^* \to V$ such that for all $\sigma \in V^*, (\mathrm{last}(\sigma), f(\sigma)) \in E$.

V^* is the set of finite sequences of elements of V.
For a sequence σ, $\mathrm{last}(\sigma)$ is the last element of σ.

Definition (Positional Strategy)

A **strategy** f is **positional** (or memoryless) if: $\forall \sigma, \sigma', \mathrm{last}(\sigma) = \mathrm{last}(\sigma')$ implies $f(\sigma) = f(\sigma')$.

We can thus define positional strategies as functions $V \to V$.

Outcome

- When the agent applies a strategy, it produces sequences of vertices, the **outcome**, that represent the successive results of each action.
- In the case when the environment is fully observable, there is only one agent, and all actions are deterministic, the outcome of a strategy is a **single path**:

Definition (Outcome)

The **outcome** $\mathrm{Outcome}(v, f)$ of strategy f from vertex v is the set of vertex sequences inductively defined by:

- $v \in \mathrm{Outcome}(v, f)$;
- if $\sigma \in \mathrm{Outcome}(v, f)$ then $\sigma.v' \in \mathrm{Outcome}(v, f)$ iff $v' = f(\sigma)$.

- We will extend these definitions as we relax those hypotheses later on.

Objectives

Definition (Objective)

An **objective** is a set of vertex sequences.

Definition (Maximal sequence)

A vertex sequence is **maximal** within a sequence set X if it is either **infinite** or it is **not a prefix** of some sequence in X.

When $X = V^*$, we just say the sequence is maximal.
We denote by $\mathrm{MaxOutcome}(v, f)$ the subset of sequences of $\mathrm{Outcome}(v, f)$ that are maximal within $\mathrm{Outcome}(v, f)$

Definition (Reachability Objective)

Given a target vertex g, the reachability objective $\mathrm{Reach}(g)$ is the set of maximal sequences σ such that $g \in \sigma$.

Winning Strategies, Winning Vertices

Definition (Winning Strategy)

A strategy f is **winning** from some vertex v for some objective X if $\mathrm{MaxOutcome}(v, f) \subseteq X$.

Definition (Winning Vertex)

A vertex v is **winning** if there exists a winning strategy from v.

Outline

Introduction

Optimal Strategies in Deterministic Environments
- The Simple Case
- Path Finding
- Optimal Decisions

Non-Deterministic Environments

Uncertain (Probabilistic) Environments

Accounting for other Agents

Supervised Learning

Conclusion

Path finding

- Under the hypothesis of a fully observable, monoagent, and deterministic environment, with a reachability objective:
  - if there is a winning strategy there is a **positional** winning strategy;
  - the outcome of a strategy is a **single path**.
- Finding a winning strategy to reach some vertex therefore amounts to find this path;
- The strategy then consists in each vertex of the path in **going to the next one**.

Spanning Tree Walk

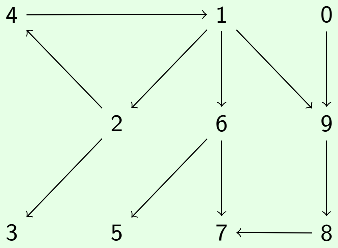
```
input:  s0     // source state
        t      // target state
output: r      // is t reachable from s0?
        pred   // state predecessors table

∀s', pred[s'] ← ⊥ // No predecessor initially
r ← false
P ← ∅ // The set of visited vertices
W ← {s0} // The set of open vertices (waiting to be explored)
while W ≠ ∅ and not r:
    s ← next(W) // take and remove the next state from W
    if s = t:
        r ← true
    else :
        P ← P ∪ {s}
        foreach successor s' of s:
            if s' ∉ P and s' ∉ W:
                pred[s'] ← s // build the path
                W ← W ∪ {s'}
```

Breadth-first Search (BFS)

- If W is a **queue**, then the order of exploration if **breadth-first**.

Example

Look for a path from 1 to 0 (impossible):



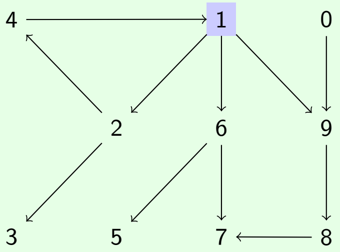
W = [1]

Breadth-first Search (BFS)

- If W is a **queue**, then the order of exploration if **breadth-first**.

Example

Look for a path from 1 to 0 (impossible):



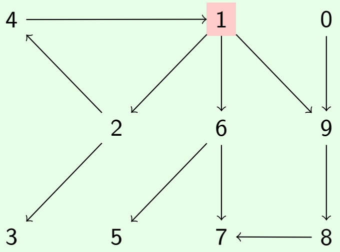
$$W = []$$

Breadth-first Search (BFS)

- If W is a **queue**, then the order of exploration if **breadth-first**.

Example

Look for a path from 1 to 0 (impossible):



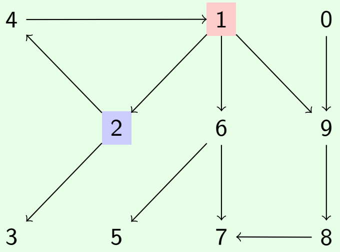
W = [2,6,9]

Breadth-first Search (BFS)

- If W is a **queue**, then the order of exploration if **breadth-first**.

Example

Look for a path from 1 to 0 (impossible):



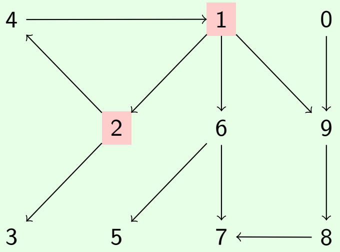
W = [6,9]

Breadth-first Search (BFS)

- If W is a **queue**, then the order of exploration if **breadth-first**.

Example

Look for a path from 1 to 0 (impossible):



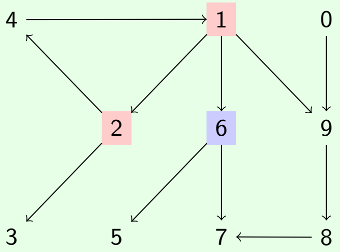
W = [6,9,4,3]

Breadth-first Search (BFS)

- If W is a **queue**, then the order of exploration if **breadth-first**.

Example

Look for a path from 1 to 0 (impossible):



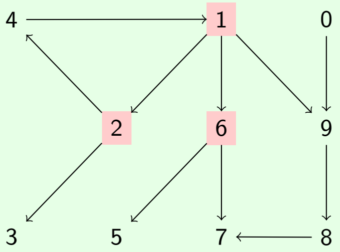
W = [9,4,3]

Breadth-first Search (BFS)

- If W is a **queue**, then the order of exploration if **breadth-first**.

Example

Look for a path from 1 to 0 (impossible):



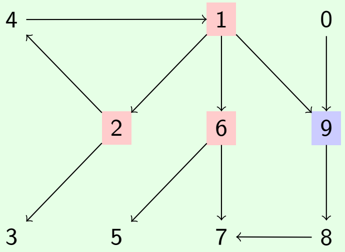
W = [9,4,3,5,7]

Breadth-first Search (BFS)

- If W is a **queue**, then the order of exploration if **breadth-first**.

Example

Look for a path from 1 to 0 (impossible):



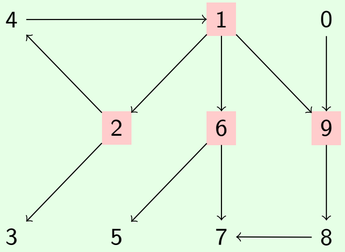
W = [4,3,5,7]

Breadth-first Search (BFS)

- If W is a **queue**, then the order of exploration if **breadth-first**.

Example

Look for a path from 1 to 0 (impossible):



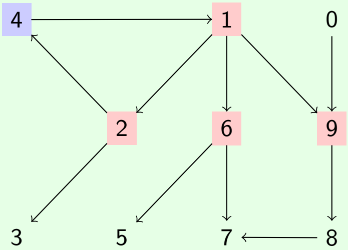
W = [4,3,5,7,8]

Breadth-first Search (BFS)

- If W is a **queue**, then the order of exploration if **breadth-first**.

Example

Look for a path from 1 to 0 (impossible):



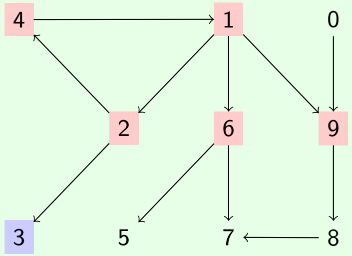
W = [3,5,7,8]

Breadth-first Search (BFS)

- If W is a **queue**, then the order of exploration if **breadth-first**.

Example

Look for a path from 1 to 0 (impossible):



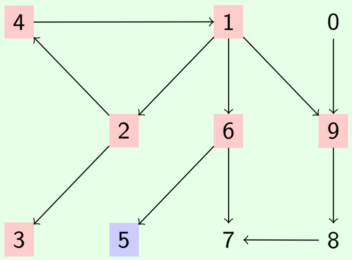
W = [5,7,8]

Breadth-first Search (BFS)

- If W is a **queue**, then the order of exploration if **breadth-first**.

Example

Look for a path from 1 to 0 (impossible):



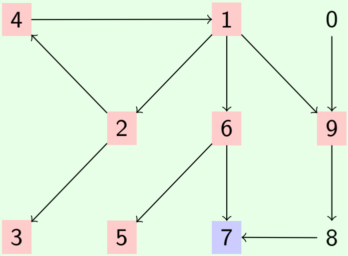
W = [7,8]

Breadth-first Search (BFS)

- If W is a **queue**, then the order of exploration if **breadth-first**.

Example

Look for a path from 1 to 0 (impossible):



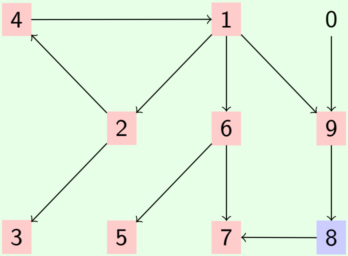
W = [8]

Breadth-first Search (BFS)

- If W is a **queue**, then the order of exploration if **breadth-first**.

Example

Look for a path from 1 to 0 (impossible):



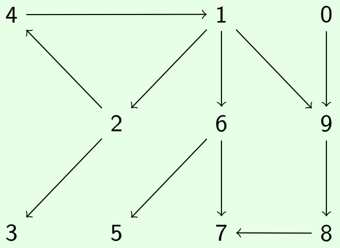
W = []

Depth-first Search (DFS)

- If W is a **stack**, then the order of exploration if **depth-first**.

Example

Look for a path from 1 to 0 (impossible):



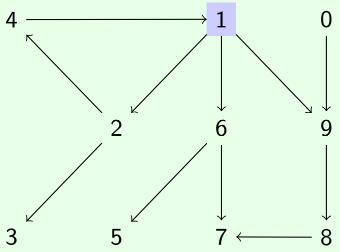
W = [1]

Depth-first Search (DFS)

- If W is a **stack**, then the order of exploration if **depth-first**.

Example

Look for a path from 1 to 0 (impossible):



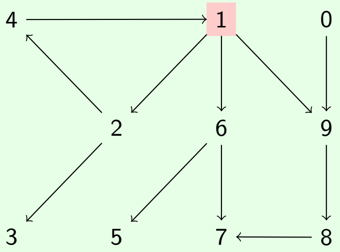
W = []

Depth-first Search (DFS)

- If W is a **stack**, then the order of exploration if **depth-first**.

Example

Look for a path from 1 to 0 (impossible):



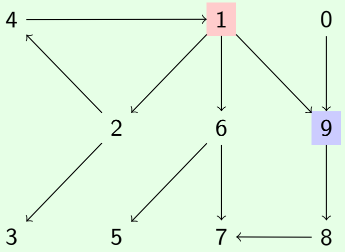
W = [9,6,2]

Depth-first Search (DFS)

- If W is a **stack**, then the order of exploration if **depth-first**.

Example

Look for a path from 1 to 0 (impossible):



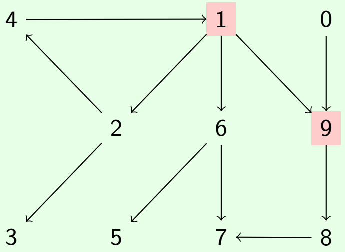
W = [6,2]

Depth-first Search (DFS)

- If W is a **stack**, then the order of exploration if **depth-first**.

Example

Look for a path from 1 to 0 (impossible):



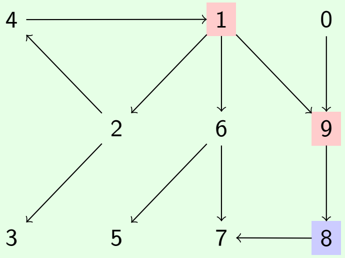
W = [8,6,2]

Depth-first Search (DFS)

- If W is a **stack**, then the order of exploration if **depth-first**.

Example

Look for a path from 1 to 0 (impossible):



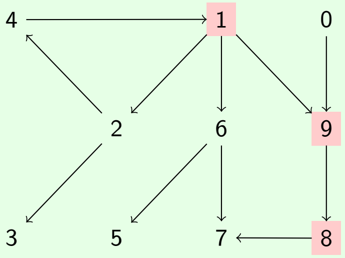
W = [6,2]

Depth-first Search (DFS)

- If W is a **stack**, then the order of exploration if **depth-first**.

Example

Look for a path from 1 to 0 (impossible):



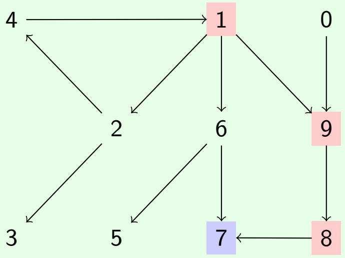
W = [7,6,2]

Depth-first Search (DFS)

- If W is a **stack**, then the order of exploration if **depth-first**.

Example

Look for a path from 1 to 0 (impossible):



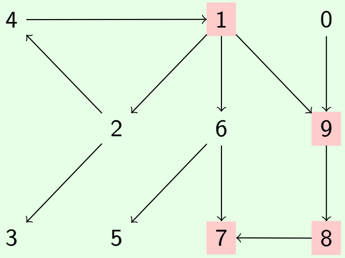
W = [6,2]

Depth-first Search (DFS)

- If W is a **stack**, then the order of exploration if **depth-first**.

Example

Look for a path from 1 to 0 (impossible):



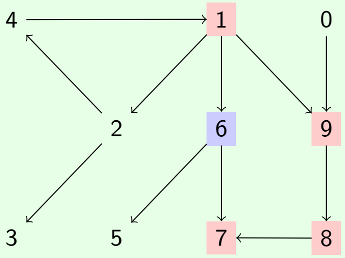
W = [6,2]

Depth-first Search (DFS)

- If W is a **stack**, then the order of exploration if **depth-first**.

Example

Look for a path from 1 to 0 (impossible):



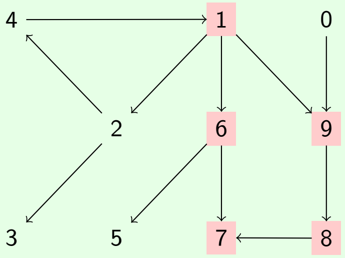
W = [2]

Depth-first Search (DFS)

- If W is a **stack**, then the order of exploration if **depth-first**.

Example

Look for a path from 1 to 0 (impossible):



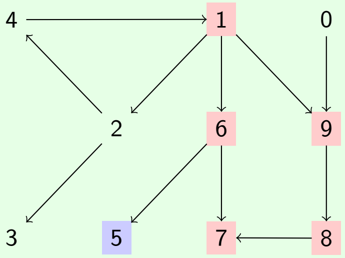
W = [5,2]

Depth-first Search (DFS)

- If W is a **stack**, then the order of exploration if **depth-first**.

Example

Look for a path from 1 to 0 (impossible):



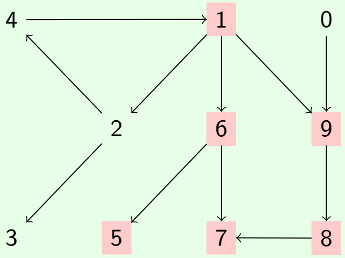
W = [2]

Depth-first Search (DFS)

- If W is a **stack**, then the order of exploration if **depth-first**.

Example

Look for a path from 1 to 0 (impossible):



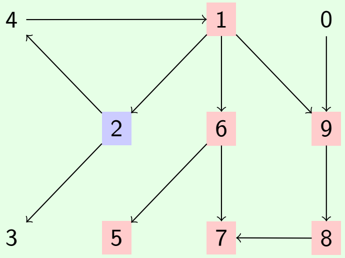
W = [2]

Depth-first Search (DFS)

- If W is a **stack**, then the order of exploration if **depth-first**.

Example

Look for a path from 1 to 0 (impossible):



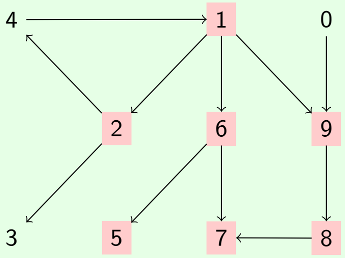
W = []

Depth-first Search (DFS)

- If W is a **stack**, then the order of exploration if **depth-first**.

Example

Look for a path from 1 to 0 (impossible):



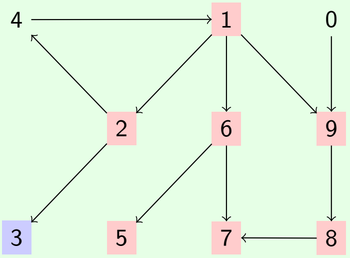
W = [3,4]

Depth-first Search (DFS)

- If W is a **stack**, then the order of exploration if **depth-first**.

Example

Look for a path from 1 to 0 (impossible):



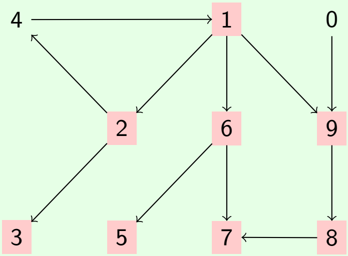
W = [4]

Depth-first Search (DFS)

- If W is a **stack**, then the order of exploration if **depth-first**.

Example

Look for a path from 1 to 0 (impossible):



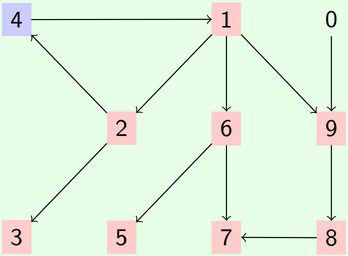
W = [4]

Depth-first Search (DFS)

- If W is a **stack**, then the order of exploration if **depth-first**.

Example

Look for a path from 1 to 0 (impossible):



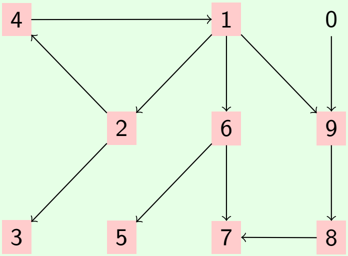
W = []

Depth-first Search (DFS)

- If W is a **stack**, then the order of exploration if **depth-first**.

Example

Look for a path from 1 to 0 (impossible):



W = []

BFS Vs DFS

- BFS gives the **shortest** path (wrt. number of edges);
- DFS can easily find loops;
- DFS explores in a more **local** manner;
- DFS can be made more **memory-efficient** (at the cost of exploring **redundant** paths).

Forgetful DFS

```
function fDFS:
    input:  s     // source state
            t     // target state
            path  // Current path
    output: r     // is t reachable from s?
            path' // Path to goal

    path' ← path : s // concatenate s
    if s = t:
        r ← true
    else:
        r ← false
        foreach successor s' of s and while not r:
            if s' ∉ path':
                (r, path'') ← fDFS(s', t, path')
                if r:
                    path' ← path''
```

Forgetful DFS

Exemple

$$\begin{tikzcd}
1 \arrow[r] \arrow[dr] & 2 \arrow[r] & 4 \arrow[loop right] \\
& 3 \arrow[u] & \\
5 \arrow[uu] & &
\end{tikzcd}$$

How many nodes are visited in the worst case by BFS, DFS and "forgetful DFS", before we can conclude that 5 is not reachable?

Completeness

A search algorithm is **complete** *if whenever there exists a path to the target vertex, the algorithm always finds one.*

Exercise

Are BFS and DFS (normal and forgetful) complete?

Completeness

A search algorithm is **complete** *if whenever there exists a path to the target vertex, the algorithm always finds one.*

Exercise

Are BFS and DFS (normal and forgetful) complete? What about infinite graphs?

Iterative Deepening

- Even in finite graphs, DFS can go astray for a while if unlucky;
- To alleviate these problems, we can arbitrarily **bound** the length of the paths explored;
- If we find a path, we are done;
- Otherwise, **increase** the bound and **retry**.
- Works well because in a tree there are much more **leaves** than **internal nodes**.

Exercise

Exercise

Two person move *simultaneously* on a map in a turn-wise manner. Each turn they both move to one the neighboring city. The new turn starts when both have reached their destination. We want a (global) strategy to make them meet.

- Can we model this as a graph path-finding problem?
- If the map is completely connected, is there always a solution?
- Find a map for which one of the two persons must visit a city twice before meeting the other.

Outline

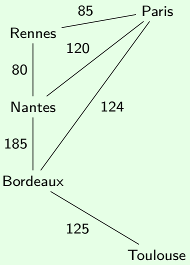
Costs

- BFS gives the **shortest** path to the goal;
- But all actions might not have the same **cost**;
- Cost may represent time, money, energy, etc.
- We may be interested in the **minimum cost** path to the goal:
  - extend the model;
  - modify the algorithms.

Weighted (Directed) Graphs

- we add an integer **weight** (or cost) to each edge with a function $\omega : E \rightarrow \mathbb{N}_{>0}$;
- the case where weights can be **non-positive** leads to more complex algorithms (e.g. Bellman-Ford).

Example: Some train travel durations in the West



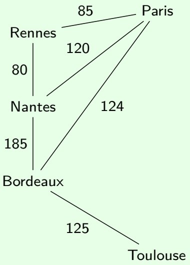
What is the best option to go from Rennes to Bordeaux?

Dijkstra's Algorithm

```
input:  s0    // source state
        t     // target state
output: r     // is t reachable from s0?
        pred  // state predecessors table
        cost  // state cost table

∀s', pred[s'] ← ⊥ // No predecessor initially
cost[s0] ← 0
∀s' ≠ s0, cost[s'] ← +∞ // Initial costs
r ← false
W ← {s0} // W is a priority queue
while W ≠ ∅ and not r:
    s ← extract_min(W, cost) // take and remove the mincost state from W
    if s = t:
        r ← true
    else :
        foreach successor s' of s:
            if cost[s] + ω(s, s') < cost(s'):
                cost[s'] ← cost[s] + ω(s, s')
                pred[s'] ← s // build the path
                W ← W ∪ {s'}
```

Dijkstra's Algorithm: Example

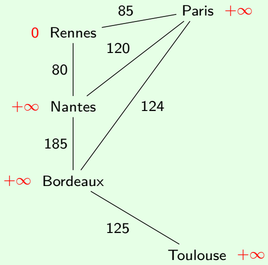
Example: Some train travel durations in the West



What is the best option to go from Rennes to Bordeaux?

Dijkstra's Algorithm: Example

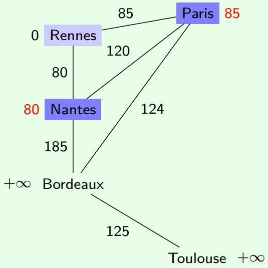
Example: Some train travel durations in the West



What is the best option to go from Rennes to Bordeaux?

Dijkstra's Algorithm: Example

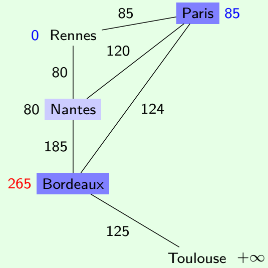
Example: Some train travel durations in the West



What is the best option to go from Rennes to Bordeaux?

Dijkstra's Algorithm: Example

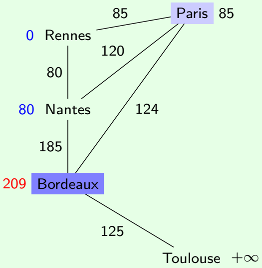
Example: Some train travel durations in the West



What is the best option to go from Rennes to Bordeaux?

Dijkstra's Algorithm: Example

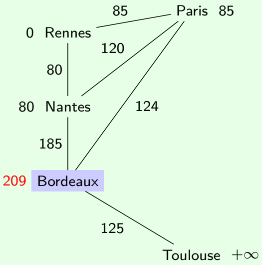
Example: Some train travel durations in the West



What is the best option to go from Rennes to Bordeaux?

Dijkstra's Algorithm: Example

Example: Some train travel durations in the West



What is the best option to go from Rennes to Bordeaux?

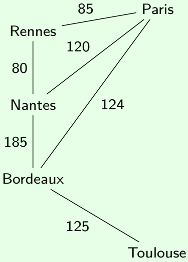
A^*

- Dijkstra's algorithm explores first the vertex n with the minimum **path-cost** $g(n)$ but knows nothing of what remains to do (**uninformed search**);
- Suppose now we have a **heuristic** $h(n)$ that estimates the cost from n to the target;
- We assume $h(x) = 0$ for the target vertex;
- We explore nodes by selecting the one with the minimum value for

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

A^*: Example

Example: Some train travel durations in the West



Rennes	45
Nantes	33
Paris	60
Toulouse	25
Bordeaux	0

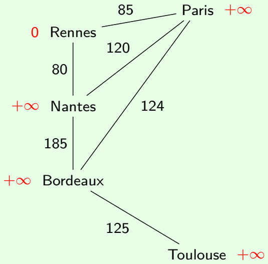
Table: Straight flight times (min) to Bordeaux at 500 km/h

What is the best option to go from Rennes to Bordeaux?

Use straight flight times as heuristic

A^*: Example

Example: Some train travel durations in the West



Rennes	45
Nantes	33
Paris	60
Toulouse	25
Bordeaux	0

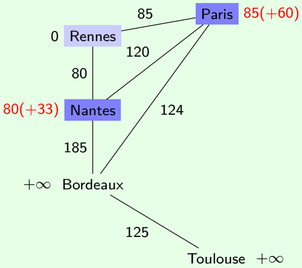
Table: Straight flight times (min) to Bordeaux at 500 km/h

What is the best option to go from Rennes to Bordeaux?

Use straight flight times as heuristic

A^*: Example

Example: Some train travel durations in the West



Rennes	45
Nantes	33
Paris	60
Toulouse	25
Bordeaux	0

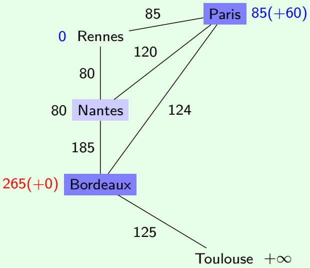
Table: Straight flight times (min) to Bordeaux at 500 km/h

What is the best option to go from Rennes to Bordeaux?

Use straight flight times as heuristic

A^*: Example

Example: Some train travel durations in the West



Rennes	45
Nantes	33
Paris	60
Toulouse	25
Bordeaux	0

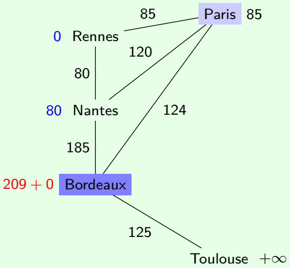
Table: Straight flight times (min) to Bordeaux at 500 km/h

What is the best option to go from Rennes to Bordeaux?

Use straight flight times as heuristic

A^*: Example

Example: Some train travel durations in the West



Rennes	45
Nantes	33
Paris	60
Toulouse	25
Bordeaux	0

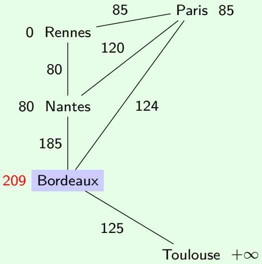
Table: Straight flight times (min) to Bordeaux at 500 km/h

What is the best option to go from Rennes to Bordeaux?

Use straight flight times as heuristic

A^*: Example

Example: Some train travel durations in the West



Rennes	45
Nantes	33
Paris	60
Toulouse	25
Bordeaux	0

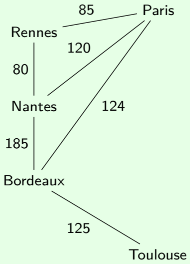
Table: Straight flight times (min) to Bordeaux at 500 km/h

What is the best option to go from Rennes to Bordeaux?

Use straight flight times as heuristic

A^*: Example

Example: Some train travel durations in the West



Rennes	45
Nantes	185
Paris	124
Toulouse	125
Bordeaux	0

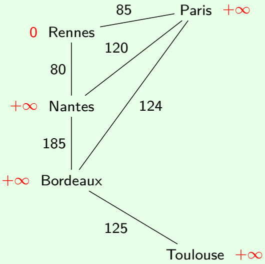
Table: Actual duration when direct connection or straight flight times (min) to Bordeaux at 500 km/h

What is the best option to go from Rennes to Bordeaux?

Use actual direct times when available or straight flight times otherwise as heuristic

A^*: Example

Example: Some train travel durations in the West



Rennes	45
Nantes	185
Paris	124
Toulouse	125
Bordeaux	0

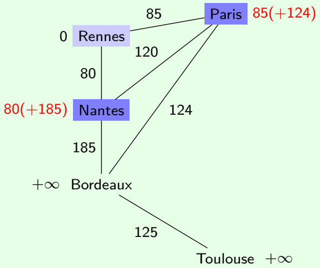
Table: Actual duration when direct connection or straight flight times (min) to Bordeaux at 500 km/h

What is the best option to go from Rennes to Bordeaux?

Use actual direct times when available or straight flight times otherwise as heuristic

A^*: Example

Example: Some train travel durations in the West



Rennes	45
Nantes	185
Paris	124
Toulouse	125
Bordeaux	0

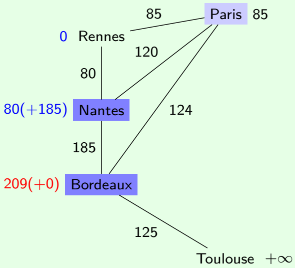
Table: Actual duration when direct connection or straight flight times (min) to Bordeaux at 500 km/h

What is the best option to go from Rennes to Bordeaux?

Use actual direct times when available or straight flight times otherwise as heuristic

A^*: Example

Example: Some train travel durations in the West



Rennes	45
Nantes	185
Paris	124
Toulouse	125
Bordeaux	0

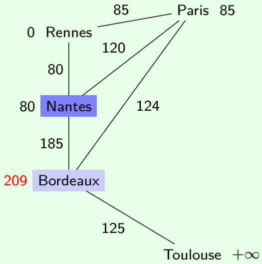
Table: Actual duration when direct connection or straight flight times (min) to Bordeaux at 500 km/h

What is the best option to go from Rennes to Bordeaux?

Use actual direct times when available or straight flight times otherwise as heuristic

A^*: Example

Example: Some train travel durations in the West



Rennes	45
Nantes	185
Paris	124
Toulouse	125
Bordeaux	0

Table: Actual duration when direct connection or straight flight times (min) to Bordeaux at 500 km/h

What is the best option to go from Rennes to Bordeaux?

Use actual direct times when available or straight flight times otherwise as heuristic

A^*: About heursitics

- The heuristic is a **function** that takes a **state** as input;
- Its output is an **estimate** of the cost from that state to the goal;
- Its result has the **same unit** as the weight on edges;
- By adding the cost from the initial state s_0 to state s and the heuristic at s, we get an estimate of the **total cost** of going through s to reach the goal;
- Finding a (good) heuristic is part of solving the minimum cost problem with A^*.

A^*: Optimality and "good" heuristics

- A^* is guaranteed to find an **optimal** path if h is **admissible**:
  An A^* heuristic is admissible if it never overestimates the cost to the goal.
- A^* is guaranteed never to visit a vertex twice if h is **consistent**:
  An A^* heuristic h is consistent if:
  $\forall (x, y) \in E, h(x) \leq \omega(x, y) + h(y)$.
- We can obtain **admissible** heuristics by solving **relaxed** versions versions of the problem, i.e., by removing some constraints.

Exercise

1. Were the previous two heuristics admissible and consistent?
2. Prove that if h is consistent then it is admissible;
3. Give an admissible heuristic that is not consistent.

Exercises

Exercise

Bridge crossing (Levmore & Cook, 1981). We quote the version of (Rote, 2002): "Four people begin on the same side of a bridge. You must help them across tothe other side. It is night. There is one flashlight. A maximum of two people can cross at a time. Any party who crosses, either one or two people, must have the flashlight to see. The flashlight must be walked back and forth, it cannot be thrown, etc. Each person walks at a different speed. A pair must walk together at the rate of the slower person's pace, based on this information: Person 1 takes $t_1 = 1$ min to cross, and the other persons take $t_2 = 2$ min, $t_3 = 5$ min, and $t_4 = 10$ min to cross, respectively."

We want to find the fastest strategy to cross.

1. Can you intuitively find a strategy that wins in less than 20 min?
2. Give an admissible heuristic for this problem;
3. Find the fastest strategy using A^* and expanding only the relevant part of the weighted graph.

Exercises

Exercise

Knowing the straight line travel times $d(i,j)$ between cities i and j, what is an admissible heuristic for minimizing the time to meet in the previous two persons simultaneously moving (at the same speed) on a map problem? What if we want to minimise the sum of the travel times of both persons?

Exercise

Greedy Best-first Search. Suppose that instead of using $f = g + h$ to select the vertex to explore we use only h. Show that this does not always lead to an optimal path in finite graphs and that it might not be complete in infinite graphs.

Outline

Outline

Introduction

Optimal Strategies in Deterministic Environments

Non-Deterministic Environments

- (Optimal) Strategies in Non-deterministic Environments
- Partially Observable Non-deterministic Environments

Uncertain (Probabilistic) Environments

Accounting for other Agents

Supervised Learning

Conclusion

Non-determinism

- We have supposed that performing an action always has the **same outcome**;
- We now relax this assumption: performing an action may lead **non-deterministically** to (a finite number of) **different states**;
- We have no information on what state will be reached but we can **observe** it.

Non-determinism: Example

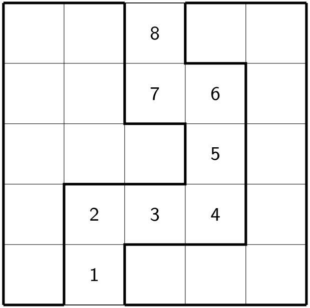
Exemple

A robot goes through a given tiled turning corridor while carrying a big pile of important stuff.

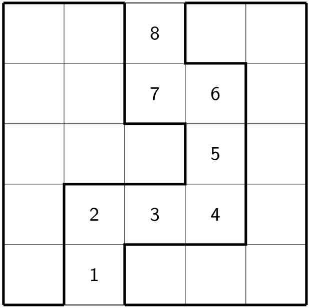
- When it turns left the pile **may**:
  - get left-unbalanced if it was balanced;
  - fall down if it was already left-unbalanced;
  - get balanced again, if it was right-unbalanced.
- The situation is symmetric when turning right.
- The robot moves the next tile in 1s or 2s if unbalanced.
- The robot can always stop and balance the pile again (4s).

Find a strategy to cross the corridor as fast as possible without the pile falling down.

Non-determinism: Example

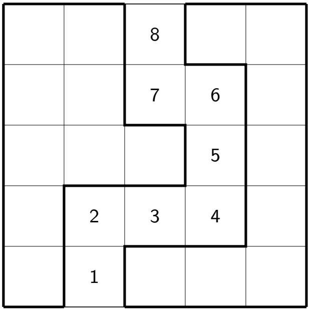


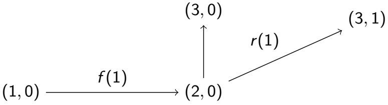
Non-determinism: Example



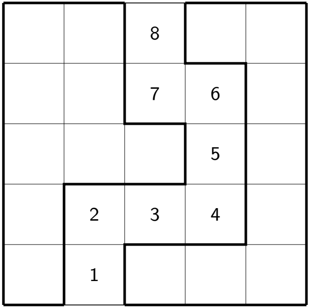
$$(1,0) \xrightarrow{f(1)} (2,0)$$

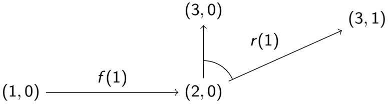
Non-determinism: Example



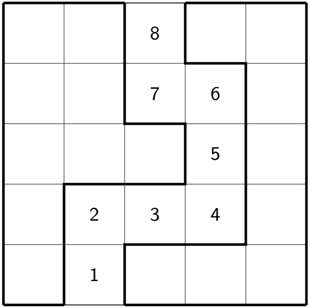


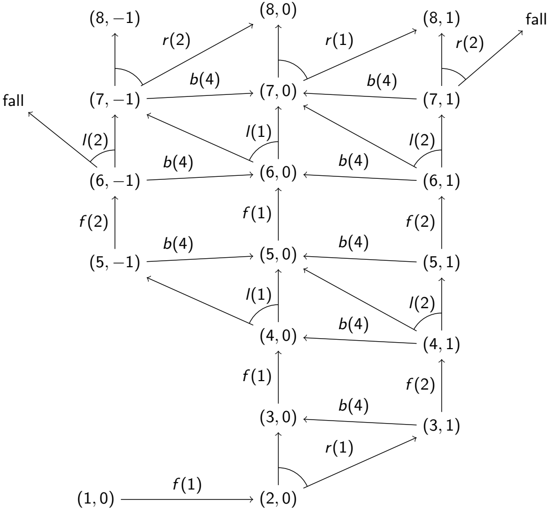
Non-deterministism: Example





Non-determinism: Example





Hypergraphs

- We use **hyperedges** to model non-deterministic outcomes:

Definition (Hypergraph)

A (directed) *hypergraph* is a pair (V, E) where V is a set of vertices and $E \subseteq V \times 2^V \setminus \{\emptyset\}$ a set of **hyperedges**.

2^V denotes the **powerset** of V, i.e, the set of all subsets of V

- An hyperedge $(v, \{v_1, \ldots, v_n\})$ models a unique action from v that can lead non-deterministically to any of the vertices $v_1, \ldots, v_n$.
- Hypergraphs are also called an **AND-OR** graphs;
- We can define **weighted** hypergraphs as before.

Non-determinism

- Playing some action means choosing an hyperedge:

Definition (Strategy)

A **strategy** is a function $f : V^* \to 2^V$ such that for all $\sigma \in V^*, (\mathrm{last}(\sigma), f(\sigma)) \in E$.

- It may have **several outcomes** and **all** of them must be winning:

Definition (Outcome)

The **outcome** $\mathrm{Outcome}(v, f)$ of strategy f from vertex v is the set of vertex sequences inductively defined by:

- $v \in \mathrm{Outcome}(v, f)$;
- if $\sigma \in \mathrm{Outcome}(v, f)$ then $\sigma.v' \in \mathrm{Outcome}(v, f)$ iff $v' \in f(\sigma)$.

- The outcome is not a single path anymore but a **tree** allowing to deal with all **contingencies**.

Solving Non-determinism

```
input:  s0     // source state
        G      // target state set
output: r      // is G reachable from s0?
        strat  // strategy

∀s′, strat[s′] ← ⊥ // No strategy initially
r ← false
∀s′ ∈ G, R[s′] ← true, ∀s′ ∉ G, R[s′] ← false
W ← G // The set of open vertices (waiting to be explored)
while W ≠ ∅ and not R[s0]:
    s′ ← next(W) // take and remove the next state from W
    foreach s such that not R[s] and ∃(s, K) ∈ E with s′ ∈ K:
        R[s] ← ⋁_{(s,K′)∈E} ( ⋀_{s′′∈K′} R[s′′])
        if R[s]:
            W ← W ∪ {s}
            strat[s] ← K′, s.t. ⋀_{s′′∈K′} R[s′′]

r ← R[s0]
```

- $\vee$ is a notation for the **logical or** between two booleans: e.g. true $\vee$ false $=$ true;
- $\wedge$ is a notation for the **logical and** between two booleans: e.g. true $\wedge$ false $=$ false;
- $\bigvee_{b \in B} b$, with $B = \{b_1, b_2, \ldots, b_n\}$ and each b_i a boolean variable is the **logical or** between all the b_i: $\bigvee_{b \in B} b = b_1 \vee b_2 \vee \cdots \vee b_n$
- Similarly, $\bigwedge_{b \in B} = b_1 \wedge b_2 \wedge \cdots \wedge b_n$ is the **logical and** between all the b_i.

Non-determinism: Exercise

Exercise: Moving Robot

Apply the previous algorithm to find a strategy for the robot to safely reach tile 8 (disregarding costs).

Optimal Decisions in Non-deterministic Environments

- To deal with costs, we can extend Dijkstra's and A* algorithms (A*LD, AO*);
- For instance Knuth extension of Dijkstra's algorithm:
  - initially goals have cost 0 and others $+\infty$;
  - always visit the vertex with the smallest cost first.
  - move backwards (hence start with goals);
  - compute costs instead of booleans, using max instead of $\wedge$ and min instead of $\vee$.

Exercise

Apply this algorithm to find an *optimal* strategy for the robot.

- We can also use other functions than max (e.g. $+$).

Outline

Partial Observability

- The agent often **perceives** the environment through a **limited number** of **sensors**;
- It might however have a more complete internal representation (model) of the environment;
- It then uses its percepts to **estimate** its current state and act accordingly;
- The evolution of the system is perceived through an **observation function** mapping states to a finite set of **observations** $\mathcal{O}$.

Partially Observable Non-deterministic Environments

- We model the environment with an hypergraph with edges labeled by actions from a finite set Σ. We also call Σ the labelling function.

Partially Observable Non-deterministic Environments

- We model the environment with an hypergraph with edges labeled by actions from a finite set Σ. We also call Σ the labelling function.
- The observation function has type $V \to \mathcal{O}$;

Partially Observable Non-deterministic Environments

- We model the environment with an hypergraph with edges labeled by actions from a finite set Σ. We also call Σ the labelling function.
- The observation function has type $V \to \mathcal{O}$;
- Let $\mathcal{R}$ be the set of **runs**, *i.e.*, sequences $v_1 a_1 v_2 a_2 \dots v_n$, with $v_i \in V$ and $a_i \in \Sigma$:

Definition (Strategy)

A **strategy** is a function $f : \mathcal{O}(\mathcal{R}) \to \Sigma$ such that for all $\sigma, \sigma' \in V^*$ such that $\mathcal{O}(\sigma) = \mathcal{O}(\sigma')$, there is a an hyperedge labeled by $f(\sigma)$ from $\mathrm{last}(\sigma)$ iff there is one from $\mathrm{last}(\sigma')$.

Definition (Outcome)

The **outcome** $\mathrm{Outcome}(v, f)$ of strategy f from vertex v is the set of runs inductively defined by:

- $v \in \mathrm{Outcome}(v, f)$;
- if $\sigma \in \mathrm{Outcome}(v, f)$ then $\sigma.v' \in \mathrm{Outcome}(v, f)$ iff $v' \in f(\mathcal{O}(\sigma))$.

Finding a Strategy

Exercise

Prove that we may need a non-positional strategy to win in such a setting.

Finding a Strategy

Exercise

Prove that we may need a non-positional strategy to win in such a setting.

- At each step we need to do some **state estimation** (aka **filtering**);

Finding a Strategy

Exercise

Prove that we may need a non-positional strategy to win in such a setting.

- At each step we need to do some **state estimation** (aka **filtering**);
- We build **information sets/belief states** gathering all vertices that we could be in given the observations seen.

Finding a Strategy

Exercise

Prove that we may need a non-positional strategy to win in such a setting.

- At each step we need to do some **state estimation** (aka **filtering**);
- We build **information sets/belief states** gathering all vertices that we could be in given the observations seen.
- We then reduce the problem of finding a strategy to a fully observable graph problem.

Building Belief States

- We build **sets of vertices**;

Building Belief States

- We build **sets of vertices**;
- Initially we have only one vertex: $\{v_0\}$

Building Belief States

- We build **sets of vertices**;
- Initially we have only one vertex: $\{v_0\}$
- Now suppose we have some belief state V'

Building Belief States

- We build **sets of vertices**;
- Initially we have only one vertex: $\{v_0\}$
- Now suppose we have some belief state V'
- Suppose we do some action a (possible from all vertices in V'):

Building Belief States

- We build **sets of vertices**;
- Initially we have only one vertex: $\{v_0\}$
- Now suppose we have some belief state V'
- Suppose we do some action a (possible from all vertices in V'):
  1. **Predict** in which states we can now be:

$$V'_a = \bigcup_{v \in V'} \Big(\bigcup_{(v,X) \in E \text{ s.t. } \Sigma((v,X))=a} X \Big)$$

Building Belief States

- We build **sets of vertices**;
- Initially we have only one vertex: $\{v_0\}$
- Now suppose we have some belief state V'
- Suppose we do some action a (possible from all vertices in V'):
  1. **Predict** in which states we can now be:

$$V'_a = \bigcup_{v \in V'} \Big(\bigcup_{(v,X) \in E \text{ s.t. } \Sigma((v,X))=a} X \Big)$$

  2. **Filter** using the new observation o:

$$V'_{a,o} = \{v \mid v \in V'_a \text{ and } \mathcal{O}(v) = o\}$$

Building Belief States

- We build **sets of vertices**;
- Initially we have only one vertex: $\{v_0\}$
- Now suppose we have some belief state V'
- Suppose we do some action a (possible from all vertices in V'):
  1. **Predict** in which states we can now be:
  $$V'_a = \bigcup_{v \in V'} \Big(\bigcup_{(v,X) \in E \text{ s.t. } \Sigma((v,X))=a} X \Big)$$
  2. **Filter** using the new observation o:
  $$V'_{a,o} = \{v \mid v \in V'_a \text{ and } \mathcal{O}(v) = o\}$$
- We build a **finite** belief hypergraph by considering:

Building Belief States

- We build **sets of vertices**;
- Initially we have only one vertex: $\{v_0\}$
- Now suppose we have some belief state V'
- Suppose we do some action a (possible from all vertices in V'):
  1. **Predict** in which states we can now be:
  $$V'_a = \bigcup_{v \in V'} \Big(\bigcup_{(v,X) \in E \text{ s.t. } \Sigma((v,X))=a} X \Big)$$
  2. **Filter** using the new observation o:
  $$V'_{a,o} = \{v \mid v \in V'_a \text{ and } \mathcal{O}(v) = o\}$$

- We build a **finite** belief hypergraph by considering:
  - Belief states as vertices;

Building Belief States

- We build **sets of vertices**;
- Initially we have only one vertex: $\{v_0\}$
- Now suppose we have some belief state V'
- Suppose we do some action a (possible from all vertices in V'):
  1. **Predict** in which states we can now be:

$$V'_a = \bigcup_{v \in V'} \left(\bigcup_{(v,X) \in E \text{ s.t. } \Sigma((v,X))=a} X \right)$$

  2. **Filter** using the new observation o:

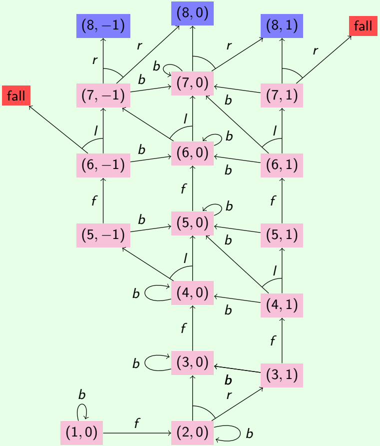
$$V'_{a,o} = \{v \mid v \in V'_a \text{ and } \mathcal{O}(v) = o\}$$

- We build a **finite** belief hypergraph by considering:
  - Belief states as vertices;
  - There is an hyperedge labeled by a between V' and all the $V'_{a,o}$ for all observations o that can occur.

Example

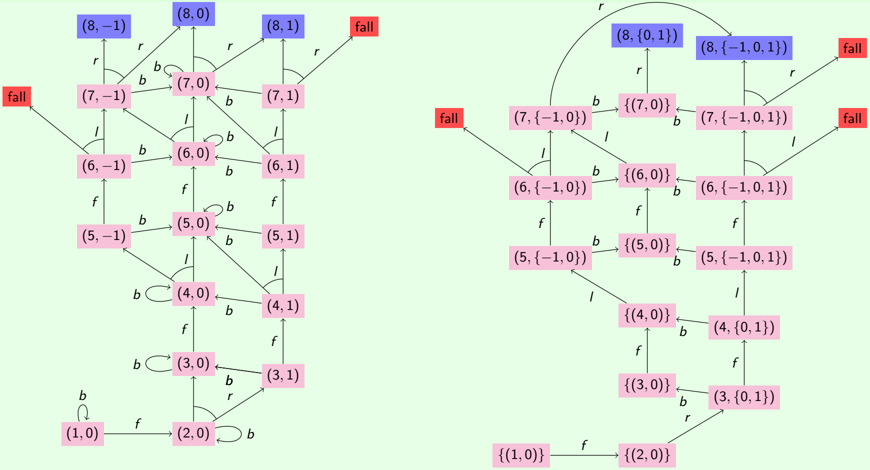
Back to the balancing robot

- We ignore costs;
- We add explicit b loops on balanced states;
- Suppose the pile is so high the robot cannot see where it is and whether the pile is unbalanced or not;
- We have only three observations:
  - we are in 8,
  - we are not in 8 and all is well,
  - the pile has fallen down.



Example

Back to the balancing robot



Building the Strategy

- Suppose we had an original reachability objective defined by a set of vertices G;

Building the Strategy

- Suppose we had an original reachability objective defined by a set of vertices G;
- Let the reachability objective R on the belief hypergraph be defined by the set of subsets of G;

Building the Strategy

- Suppose we had an original reachability objective defined by a set of vertices G;
- Let the reachability objective R on the belief hypergraph be defined by the set of subsets of G;
- Let F be the (positional) winning strategy for R on the belief hypergraph;

Building the Strategy

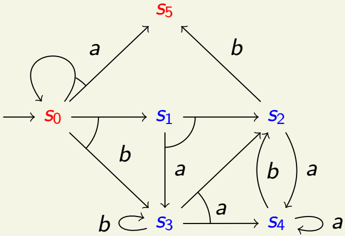
- Suppose we had an original reachability objective defined by a set of vertices G;
- Let the reachability objective R on the belief hypergraph be defined by the set of subsets of G;
- Let F be the (positional) winning strategy for R on the belief hypergraph;
- For each run on observations $\omega = o_1 a_1 \ldots o_n$, there is by construction a **unique** vertex sequence σ_ω in the belief hypergraph;

Building the Strategy

- Suppose we had an original reachability objective defined by a set of vertices G;
- Let the reachability objective R on the belief hypergraph be defined by the set of subsets of G;
- Let F be the (positional) winning strategy for R on the belief hypergraph;
- For each run on observations $\omega = o_1 a_1 \dots o_n$, there is by construction a **unique** vertex sequence σ_ω in the belief hypergraph;
- The winning strategy is therefore $f(\omega) = F(\text{last}(\sigma_\omega))$.

Exercises

Exercise



Does there exist a winning strategy to go from s_0 to s_5? *Explain by building the corresponding **belief graph**.*

Exercises

Back to the balancing robot

1. What is the minimum number of balancing moves necessary in a winning strategy?
2. What is the minimal size of the memory necessary for a winning strategy? (number of past pairs (observation, action) memorised)
3. Give (a minimal number of) observations to add to have a memoryless winning strategy.

Exercises

The blind bartender (Martin Gardner, 1979)

4 glasses are places on the corners of a square tray. Some are upright, some are upside-down. There are two players: a blindfolded bartender (B), and an antagonist (A). The bartender faces one side of the tray, cannot observe the board and wins if the glasses are all up or all down. At each round, B announces which glasses (1 or 2) to turn, then A turns the tray by a multiple of 90 degrees, and finally A turns the glasses at the positions originally announced by B (or equivalently B turns them with boxing gloves on).

1. Using symmetries, down to how many different configurations can we reduce the problem?
2. Using symmetries, how many different moves can we distinguish on those configurations?
3. Model this problem using an hypergraph with partial observability;
4. Can the bartender win? If so give a winning strategy.

Outline

Quantifying Uncertainty

- We often have some information on the relative possibility of the occurrence of uncertain outcomes;
- This can be modeled using **probabilities**;
- A rational agent then chooses actions maximizing its **utility**.

Utility of Morgenstern and Von Neumann (1944)

- We can model the **preferences** of agents with relations:
  $A \preceq B$: the agent weakly prefers B over A

Utility of Morgenstern and Von Neumann (1944)

- We can model the **preferences** of agents with relations:
  $A \preceq B$: the agent weakly prefers B over A
- Outcomes such as A may be probabilistic (lotteries): $A = pB + (1-p)C$;

Utility of Morgenstern and Von Neumann (1944)

- We can model the **preferences** of agents with relations:
  $A \preceq B$: the agent weakly prefers B over A
- Outcomes such as A may be probabilistic (lotteries): $A = pB + (1-p)C$;
- We assume $\preceq$ has good properties (completeness, transitivity, continuity, independence, decomposability);

Utility of Morgenstern and Von Neumann (1944)

- We can model the **preferences** of agents with relations:
  $A \preceq B$: the agent weakly prefers B over A
- Outcomes such as A may be probabilistic (lotteries): $A = pB + (1-p)C$;
- We assume $\preceq$ has good properties (completeness, transitivity, continuity, independence, decomposability);
- Then we can give a utility value to each outcome (even complex lotteries):

Utility of Morgenstern and Von Neumann (1944)

- We can model the **preferences** of agents with relations:
  $A \preceq B$: the agent weakly prefers B over A
- Outcomes such as A may be probabilistic (lotteries): $A = pB + (1 - p)C$;
- We assume $\preceq$ has good properties (completeness, transitivity, continuity, independence, decomposability);
- Then we can give a utility value to each outcome (even complex lotteries):

Theorem (Von Neumann & Morgenstern, 1944)

Given a preference relation with the good properties above, there exists a real-valued function U such that:

Utility of Morgenstern and Von Neumann (1944)

- We can model the **preferences** of agents with relations: $A \preceq B$: the agent weakly prefers B over A
- Outcomes such as A may be probabilistic (lotteries): $A = pB + (1-p)C$;
- We assume $\preceq$ has good properties (completeness, transitivity, continuity, independence, decomposability);
- Then we can give a utility value to each outcome (even complex lotteries):

Theorem (Von Neumann & Morgenstern, 1944)

Given a preference relation with the good properties above, there exists a real-valued function U such that:

- $U(A) \leq U(B)$ *iff* $A \preceq B$*;*

Utility of Morgenstern and Von Neumann (1944)

- We can model the **preferences** of agents with relations:
  $A \preceq B$: the agent weakly prefers B over A
- Outcomes such as A may be probabilistic (lotteries): $A = pB + (1-p)C$;
- We assume $\preceq$ has good properties (completeness, transitivity, continuity, independence, decomposability);
- Then we can give a utility value to each outcome (even complex lotteries):

Theorem (Von Neumann & Morgenstern, 1944)

Given a preference relation with the good properties above, there exists a real-valued function U such that:

- $U(A) \leq U(B)$ *iff* $A \preceq B$*;*
- $U(p_1 R_1 + \cdots + p_n R_n) = p_1 U(R_1) + \cdots + p_n U(R_n)$.

Utility of Morgenstern and Von Neumann (1944)

- We can model the **preferences** of agents with relations:
  $A \preceq B$: the agent weakly prefers B over A
- Outcomes such as A may be probabilistic (lotteries): $A = pB + (1-p)C$;
- We assume $\preceq$ has good properties (completeness, transitivity, continuity, independence, decomposability);
- Then we can give a utility value to each outcome (even complex lotteries):

Theorem (Von Neumann & Morgenstern, 1944)

Given a preference relation with the good properties above, there exists a real-valued function U such that:

- $U(A) \leq U(B)$ *iff* $A \preceq B$;
- $U(p_1 R_1 + \cdots + p_n R_n) = p_1 U(R_1) + \cdots + p_n U(R_n)$.

- That value is defined up to any **affine transformation**: $U'(R) = aU(R) + b$ also works.

Utility of Morgenstern and Von Neumann (1944)

- We can model the **preferences** of agents with relations:
  $A \preceq B$: the agent weakly prefers B over A
- Outcomes such as A may be probabilistic (lotteries): $A = pB + (1-p)C$;
- We assume $\preceq$ has good properties (completeness, transitivity, continuity, independence, decomposability);
- Then we can give a utility value to each outcome (even complex lotteries):

Theorem (Von Neumann & Morgenstern, 1944)

Given a preference relation with the good properties above, there exists a real-valued function U such that:

- $U(A) \leq U(B)$ *iff* $A \preceq B$;
- $U(p_1 R_1 + \cdots + p_n R_n) = p_1 U(R_1) + \cdots + p_n U(R_n)$.

- That value is defined up to any **affine transformation**: $U'(R) = aU(R) + b$ also works.
- A rational agent can decide between outcomes by choosing the one that **maximises utility** (principle of maximal expected utility, or MEU).

Exercise: Allais' Paradox

- Choose between the following two lotteries:

A :

Gain	$0M€$	$1M€$	$5M€$
Prob.	0	1	0

B :

Gain	$0M€$	$1M€$	$5M€$
Prob.	0.01	0.89	0.1

- and between those two:

C :

Gain	$0M€$	$1M€$	$5M€$
Prob.	0.89	0.11	0

D :

Gain	$0M€$	$1M€$	$5M€$
Prob.	0.9	0	0.1

Exercise: Allais' Paradox

- Choose between the following two lotteries:

A :

Gain	$0M€$	$1M€$	$5M€$
Prob.	0	1	0

B :

Gain	$0M€$	$1M€$	$5M€$
Prob.	0.01	0.89	0.1

- and between those two:

C :

Gain	$0M€$	$1M€$	$5M€$
Prob.	0.89	0.11	0

D :

Gain	$0M€$	$1M€$	$5M€$
Prob.	0.9	0	0.1

Exercise

Prove that $B \preceq A$ and $C \preceq D$ cannot be represented by a Von Neumann and Morgenstern utility function. You may want to start by assuming the utility of 0€ is 0, the utility of 5M€ is 1, and the utility of 1M€ is somewhere in between.

Outline

Markov Decision Processes

- We now consider **sequences** of decisions;

Markov Decision Processes

- We now consider **sequences** of decisions;
- We extend hypergraphs to introduce probabilities for possible outcomes:

Definition (Markov Decision Process (MDP))

An MDP is a tuple $(S, s_0, A, P, R, \gamma)$ where:

- S is a finite set of states;
- s_0 is the initial state;
- A is a finite set of actions;
- $P : S \times A \to \text{Dist}(S)$ is the transition function;
- $R : S \times A \times S \to \mathbb{R}$ is the reward function;

Markov Decision Processes

Back to the balancing robot

- the pile has 10% chance getting balanced/unbalanced when turning;
- We put reward 1 on states $(8, *)$;
- We put reward -0.1 on all other states.

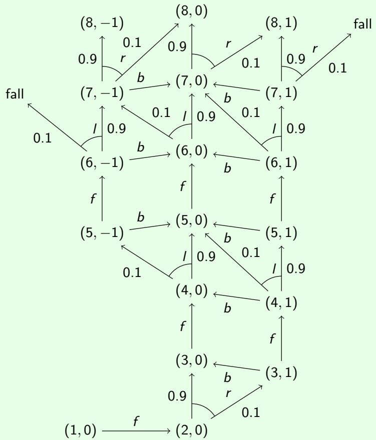
In hyperarcs with a single target, the target has probability one

Markov Decision Processes

Back to the balancing robot

- the pile has 10% chance getting balanced/unbalanced when turning;
- We put reward 1 on states $(8, *)$;
- We put reward -0.1 on all other states.

In hyperarcs with a single target, the target has probability one



Markov Decision Processes

Exercise

Which of the following games can be modelled as MDPs?

- Roulette;
- Russian roulette;
- Blackjack
- Backgammon;
- Slot machines (one-armed bandit).

Sequential Decisions: Strategies

Definition (Strategy in an MDP)

A strategy (or policy, or scheduler) in an MDP is a function $\pi : S^* \to A$.

Sequential Decisions: Strategies

Definition (Strategy in an MDP)

A strategy (or policy, or scheduler) in an MDP is a function $\pi : S^* \to A$.

- Memoryless strategies are **not sufficient** in general for MDPs to maximise the probability of reaching some state:

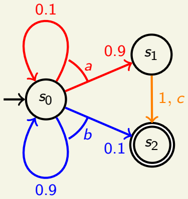
Sequential Decisions: Strategies

Definition (Strategy in an MDP)

A strategy (or policy, or scheduler) in an MDP is a function $\pi : S^* \to A$.

- Memoryless strategies are **not sufficient** in general for MDPs to maximise the probability of reaching some state:

Exercise



If only two successive moves are allowed (finite horizon), what is the optimal strategy to reach s_2?

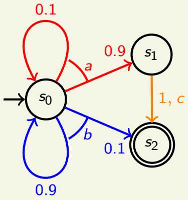
Sequential Decisions: Strategies

Definition (Strategy in an MDP)

A strategy (or policy, or scheduler) in an MDP is a function $\pi : S^* \to A$.

- Memoryless strategies are **not sufficient** in general for MDPs to maximise the probability of reaching some state:

Exercise



If only two successive moves are allowed (finite horizon), what is the optimal strategy to reach s_2?

- They are if we consider an **infinite horizon**.

Winning and Rewards

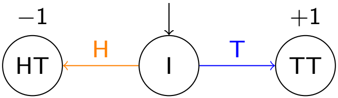
- The notion of winning strategy is not as useful as before:
- We have no definite **goal state**;
- We are more interested in **maximising the rewards** obtained.

Example: Let's play matching pennies

Matching pennies

Matching pennies is a two-player game. Let S and D be the two players. Initially, each player has a penny. They independently and secretly choose heads or tails and reveal their choices simultaneously. If both faces match, S takes both coins, otherwise D takes them.

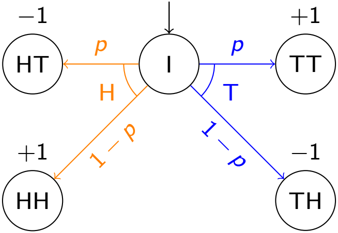
Consider the point of view of S. Assume we know that D always plays tails. What is the best strategy and how much is the associated gain?



$$
\begin{aligned}
&U(HT) = -1, U(TT) = 1,\\
&U(H) = U(HT) = -1\\
&U(T) = U(TT) = 1\\
&U(I) = 0 + \max\big(U(H), U(T)\big) = 1
\end{aligned}
$$

Example: Let's play matching pennies

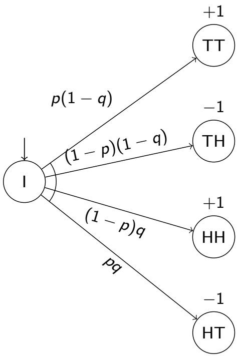
- Assume now we know that D always plays tails with probability p. What is the best **deterministic** strategy?



$$U(HT) = -1, U(HH) = 1, U(TT) = 1, U(TH) = -1$$
$$U(H) = pU(HT) + (1-p)U(HH)$$
$$U(T) = pU(TT) + (1-p)U(TH)$$
$$U(I) = 0 + \max\left(U(H), U(T)\right) = \max\left(1-2p, 2p-1\right)$$

Example: Let's play matching pennies

- Assume still we know that D always play tails with probability p. What is the best **probabilistic** strategy?
- Assume we play heads with probability q.



$$
\begin{aligned}
U(I) &= p(1-q) + q(1-p) - pq - (1-p)(1-q) \\
&= 2(p+q-2pq) - 1 \\
\frac{\partial U(I)}{\partial q} &= 2(1-2p)
\end{aligned}
$$

- when $p < 0.5$, $U(I)$ increases with q so $q = 1$ is the optimal.
- when $p > 0.5$, $U(I)$ decreases with q so $q = 0$ is the optimal.
- when $p = 0.5$, $U(I)$ is constant, so any strategy is fine.

Theorem

For any (finite) MDP, there exists a ***deterministic*** *strategy that maximizes the expected gain over an infinite horizon.*

Example: Let's play matching pennies

- Assume D's strategy is to play tails with probability 0.2 and S's strategy is to always play heads.
- Now if D knows these two strategies (a.k.a. the **strategy profile**), she would certainly want to change her strategy.

Definition (Nash equilibrium)

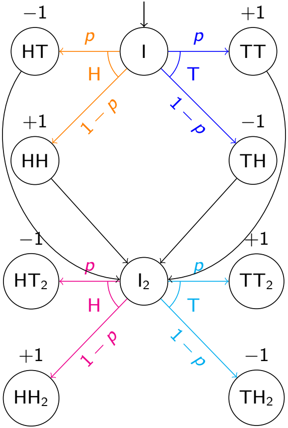
A strategy profile is a **Nash equilibrium** if none of the players has an incentive to unilaterally change their strategy.

Exercise

Does there exist a Nash equilibrium in *matching pennies*?

Example: Let's play matching pennies

- Assume D's strategy is to play tails with probability p and we play 2 rounds. What is the best strategy?



$$
\begin{aligned}
U(I_2) &= \max\left(1-2p, 2p-1\right)\\
U(HT) &= U(TH) = -1 + U(I_2)\\
U(TT) &= U(HH) = 1 + U(I_2)\\
U(I) &= \max\big(p(-1+U(I_2)) + (1-p)(1+U(I_2)),\\
&\qquad\quad p(1+U(I_2)) + (1-p)(-1+U(I_2))\big)\\
&= \max\left(U(I_2) + (1-2p), U(I_2) + (2p-1)\right)\\
&= 2\max\left(1-2p, 2p-1\right)
\end{aligned}
$$

Utility of Sequences of States

Suppose that we have chosen rewards to match preferences between states. Consider the length of sequences:

- length 1 (one state): the utility is clearly the reward of that state.

Utility of Sequences of States

Suppose that we have chosen rewards to match preferences between states. Consider the length of sequences:

- length 1 (one state): the utility is clearly the reward of that state.
- length $>$1: by induction, we get the immediate reward of the first state plus (a fraction of) the utility of the sequence starting from the second state. With $\gamma \in [0, 1]$:

$$U(s_0 s_1 \ldots s_n) = U(s_0) + \gamma U(s_1 \ldots s_n)$$

Utility of Sequences of States

Suppose that we have chosen rewards to match preferences between states. Consider the length of sequences:

- length 1 (one state): the utility is clearly the reward of that state.
- length >1: by induction, we get the immediate reward of the first state plus (a fraction of) the utility of the sequence starting from the second state. With $\gamma \in [0, 1]$:

$$U(s_0 s_1 \dots s_n) = U(s_0) + \gamma U(s_1 \dots s_n)$$

- This implies:

Utility of Sequences of States

Suppose that we have chosen rewards to match preferences between states. Consider the length of sequences:

- length 1 (one state): the utility is clearly the reward of that state.
- length >1: by induction, we get the immediate reward of the first state plus (a fraction of) the utility of the sequence starting from the second state. With $\gamma \in [0, 1]$:

$$U(s_0 s_1 \ldots s_n) = U(s_0) + \gamma U(s_1 \ldots s_n)$$

- This implies:
  - Choosing states with greater rewards increases utility;

Utility of Sequences of States

Suppose that we have chosen rewards to match preferences between states. Consider the length of sequences:

- length 1 (one state): the utility is clearly the reward of that state.
- length >1: by induction, we get the immediate reward of the first state plus (a fraction of) the utility of the sequence starting from the second state. With $\gamma \in [0, 1]$:

$$U(s_0 s_1 \ldots s_n) = U(s_0) + \gamma U(s_1 \ldots s_n)$$

- This implies:
    - Choosing states with greater rewards increases utility;
    - $\gamma < 1$ implies getting a reward sooner increases utility.

Utility of Sequences of States

- Infinite sequences:
  - if $\gamma < 1$, the utility is finite (we assume that here) and bounded by:

$$\sum_{i=0}^{\infty} \gamma^i R_{\max} = \frac{R_{\max}}{1-\gamma}$$

  - otherwise, one can also use averages on the number of states.

Utility of States

- Given a strategy π, we have a (discrete time) Markov chain: it is purely probabilistic;

Utility of States

- Given a strategy π, we have a (discrete time) Markov chain: it is purely probabilistic;
- From some state s, the states that will be visited are random variables: S_t is the state visited after t transitions (*at time t*).

Utility of States

- Given a strategy π, we have a (discrete time) Markov chain: it is purely probabilistic;
- From some state s, the states that will be visited are random variables: S_t is the state visited after t transitions (*at time t*).
- The utility of a state, for a given strategy, is the expected utility of the sequence we obtain:

  To simplify the notation, we assume the rewards depend only on the state

$$U^{\pi}(s) = E\Big[\sum_{t=0}^{\infty} \gamma^{t} R(S_t)\Big]$$

Computing an Optimal Strategy: Value Iteration

- An **optimal** strategy π^* from *s* is one that maximises the utility of *s*;

Computing an Optimal Strategy: Value Iteration

- An **optimal** strategy π^* from s is one that maximises the utility of s;
- π^* then always goes to the state with maximal utility;

Computing an Optimal Strategy: Value Iteration

- An **optimal** strategy π^* from s is one that maximises the utility of s;
- π^* then always goes to the state with maximal utility;
- **Bellman equation** for MDP:

$$U^{\pi^*}(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} P(s'|a, s) U^{\pi^*}(s')$$

Computing an Optimal Strategy: Value Iteration

- An **optimal** strategy π^* from s is one that maximises the utility of s;
- π^* then always goes to the state with maximal utility;
- **Bellman equation** for MDP:

$$U^{\pi^*}(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} P(s'|a, s) U^{\pi^*}(s')$$

- These equations are non-linear but we can solve them by iteration (a.k.a. **dynamic programming**) up to an error ϵ:

Computing an Optimal Strategy: Value Iteration

- An **optimal** strategy π^* from s is one that maximises the utility of s;
- π^* then always goes to the state with maximal utility;
- **Bellman equation** for MDP:

$$U^{\pi^*}(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} P(s'|a,s) U^{\pi^*}(s')$$

- These equations are non-linear but we can solve them by iteration (a.k.a. **dynamic programming**) up to an error ϵ:

Value Iteration Algorithm

Computing an Optimal Strategy: Value Iteration

- An **optimal** strategy π^* from s is one that maximises the utility of s;
- π^* then always goes to the state with maximal utility;
- **Bellman equation** for MDP:

$$U^{\pi^*}(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} P(s'|a,s) U^{\pi^*}(s')$$

- These equations are non-linear but we can solve them by iteration (a.k.a. **dynamic programming**) up to an error ϵ:

Value Iteration Algorithm

- $\forall s, U_0(s) = 0$;

Computing an Optimal Strategy: Value Iteration

- An **optimal** strategy π^* from s is one that maximises the utility of s;
- π^* then always goes to the state with maximal utility;
- **Bellman equation** for MDP:

$$U^{\pi^*}(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} P(s'|a,s) U^{\pi^*}(s')$$

- These equations are non-linear but we can solve them by iteration (a.k.a. **dynamic programming**) up to an error ϵ:

Value Iteration Algorithm

- $\forall s, U_0(s) = 0$;
- $\forall s, U_{i+1}(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} P(s'|a,s) U_i(s')$;

Computing an Optimal Strategy: Value Iteration

- An **optimal** strategy π^* from s is one that maximises the utility of s;
- π^* then always goes to the state with maximal utility;
- **Bellman equation** for MDP:

$$U^{\pi^*}(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} P(s'|a,s) U^{\pi^*}(s')$$

- These equations are non-linear but we can solve them by iteration (a.k.a. **dynamic programming**) up to an error ϵ:

Value Iteration Algorithm

- $\forall s, U_0(s) = 0$;
- $\forall s, U_{i+1}(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} P(s'|a,s) U_i(s')$;
- Until we have $\max_s |U_{i+1}(s) - U_i(s)| < \frac{\epsilon(1-\gamma)}{\gamma}$.

Convergence of Value Iteration

- Let B be the **Bellman update**, i.e., the function s.t. $U_{i+1} = B(U_i)$;
- B is a **contraction** by a factor of γ for the **max norm** $||U|| = \max |U(s)|$:

$$||B(U) - B(U')|| \leq \gamma ||U - U'||$$

- So, if $\gamma < 1$, it has a **unique** fixed point U^* (i.e. such that $B(U^*) = U^*$);
- In particular, for U^*, and any i:

$$||B(U_i) - U^*|| \leq \gamma ||U_i - U^*||$$

Convergence of Value Iteration

- Since utilities are bounded by $\frac{R_{\max}}{1-\gamma}$, the initial error is:

$$||U_0 - U^*|| = ||U^*|| \leq \frac{R_{\max}}{1-\gamma}$$

Convergence of Value Iteration

- Since utilities are bounded by $\frac{R_{\max}}{1-\gamma}$, the initial error is:

$$||U_0 - U^*|| = ||U^*|| \leq \frac{R_{\max}}{1-\gamma}$$

- After N iterations:

$$||U_n - U^*|| \leq \gamma^n ||U_0 - U^*|| \leq \gamma^n \frac{R_{\max}}{1-\gamma}$$

Convergence of Value Iteration

- Since utilities are bounded by $\frac{R_{\max}}{1-\gamma}$, the initial error is:

$$||U_0 - U^*|| = ||U^*|| \leq \frac{R_{\max}}{1-\gamma}$$

- After N iterations:

$$||U_n - U^*|| \leq \gamma^n ||U_0 - U^*|| \leq \gamma^n \frac{R_{\max}}{1-\gamma}$$

- So for the error to be less than ϵ we need n iterations with:

$$n = \left\lceil \frac{\log\left(\frac{R_{\max}}{\epsilon(1-\gamma)}\right)}{\log(\frac{1}{\gamma})} \right\rceil$$

Convergence of Value Iteration

- Since utilities are bounded by $\frac{R_{\max}}{1-\gamma}$, the initial error is:

$$||U_0 - U^*|| = ||U^*|| \leq \frac{R_{\max}}{1-\gamma}$$

- After N iterations:

$$||U_n - U^*|| \leq \gamma^n ||U_0 - U^*|| \leq \gamma^n \frac{R_{\max}}{1-\gamma}$$

- So for the error to be less than ϵ we need n iterations with:

$$n = \left\lceil \frac{\log\left(\frac{R_{\max}}{\epsilon(1-\gamma)}\right)}{\log(\frac{1}{\gamma})} \right\rceil$$

- We can also prove that if $||U_{i+1} - U_i|| < \epsilon\frac{1-\gamma}{\gamma}$ then $||U_{i+1} - U^*|| < \epsilon$.

Convergence of Value Iteration: Policy Loss

- Let π_i be the strategy defined using U_i.

Convergence of Value Iteration: Policy Loss

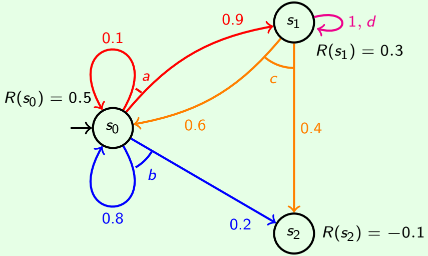
- Let π_i be the strategy defined using U_i.
- It might not be optimal, even after the utilities have been computed up to ϵ.

Convergence of Value Iteration: Policy Loss

- Let π_i be the strategy defined using U_i.
- It might not be optimal, even after the utilities have been computed up to ϵ.
- But if $||U_i - U^*|| < \epsilon$, we can prove that $||U^{\pi_i} - U^*|| < \epsilon\frac{\gamma}{1-\gamma}$

Value Iteration: Example

Exemple



Assume $\gamma = 1$

	s_0	s_1	s_2
U_0	0	0	0
U_1	$0.5 + \gamma \max(0.1 * 0 + 0.9 * 0, 0.8 * 0 + 0.2 * 0) = \mathbf{0.5}$	$0.3 + \gamma \max(0.6 * 0 + 0.4 * 0, 1 * 0) = \mathbf{0.3}$	-0.1
U_2	$0.5 + \gamma \max(0.1 * 0.5 + 0.9 * 0.3, \mathbf{0.8} * \mathbf{0.5} - \mathbf{0.2} * \mathbf{0.1}) = \mathbf{0.88}$	$0.3 + \gamma \max(0.6 * 0.5 - 0.4 * 0.1, \mathbf{1} * \mathbf{0.3}) = \mathbf{0.6}$	-0.1
U_3	$0.5 + \gamma \max(0.1 * 0.88 + 0.9 * 0.6, \mathbf{0.8} * \mathbf{0.88} - \mathbf{0.2} * \mathbf{0.1}) = \mathbf{1.184}$	$0.3 + \gamma \max(0.6 * 0.88 - 0.4 * 0.1, \mathbf{1} * \mathbf{0.6}) = \mathbf{0.9}$	-0.1
U_4	...	...	...

Policy Iteration

- In practice, the **policy** becomes optimal **before** the algorithm has converged.

Policy Iteration

- In practice, the **policy** becomes optimal **before** the algorithm has converged.
- An alternative approach is then **policy iteration**:

Policy Iteration

- In practice, the **policy** becomes optimal **before** the algorithm has converged.
- An alternative approach is then **policy iteration**:

Policy Iteration Algorithm

Policy Iteration

- In practice, the **policy** becomes optimal **before** the algorithm has converged.
- An alternative approach is then **policy iteration**:

Policy Iteration Algorithm

- fix an arbitrary strategy π';

Policy Iteration

- In practice, the **policy** becomes optimal **before** the algorithm has converged.
- An alternative approach is then **policy iteration**:

Policy Iteration Algorithm

- fix an arbitrary strategy π';
- do:

Policy Iteration

- In practice, the **policy** becomes optimal **before** the algorithm has converged.
- An alternative approach is then **policy iteration**:

Policy Iteration Algorithm

- fix an arbitrary strategy π';
- do:
    - $\pi \leftarrow \pi'$

Policy Iteration

- In practice, the **policy** becomes optimal **before** the algorithm has converged.
- An alternative approach is then **policy iteration**:

Policy Iteration Algorithm

- fix an arbitrary strategy π';
- do:
  - $\pi \leftarrow \pi'$
  - compute the corresponding utilities U^π

Policy Iteration

- In practice, the **policy** becomes optimal **before** the algorithm has converged.
- An alternative approach is then **policy iteration**:

Policy Iteration Algorithm

- fix an arbitrary strategy π';
- do:
  - $\pi \leftarrow \pi'$
  - compute the corresponding utilities U^π
  - compute the MEU strategy π' maximising U^π with one-step look-ahead.

Policy Iteration

- In practice, the **policy** becomes optimal **before** the algorithm has converged.
- An alternative approach is then **policy iteration**:

Policy Iteration Algorithm

- fix an arbitrary strategy π';
- do:
  - $\pi \leftarrow \pi'$
  - compute the corresponding utilities U^π
  - compute the MEU strategy π' maximising U^π with one-step look-ahead.
- repeat until $\pi' = \pi$

Policy Iteration

- In practice, the **policy** becomes optimal **before** the algorithm has converged.
- An alternative approach is then **policy iteration**:

Policy Iteration Algorithm

- fix an arbitrary strategy π';
- do:
  - $\pi \leftarrow \pi'$
  - compute the corresponding utilities U^π
  - compute the MEU strategy π' maximising U^π with one-step look-ahead.
- repeat until $\pi' = \pi$

- Since the strategy is fixed, computing utilities is simpler: there is no max;

Policy Iteration

- In practice, the **policy** becomes optimal **before** the algorithm has converged.
- An alternative approach is then **policy iteration**:

Policy Iteration Algorithm

- fix an arbitrary strategy π';
- do:
  - $\pi \leftarrow \pi'$
  - compute the corresponding utilities U^π
  - compute the MEU strategy π' maximising U^π with one-step look-ahead.
- repeat until $\pi' = \pi$

- Since the strategy is fixed, computing utilities is simpler: there is no max;
- This can be done in $\mathcal{O}(n^3)$ with standard linear algebra techniques.

Modified Policy Iteration

- Computing the utilities can also be done by (simplified) **value iteration**;

Modified Policy Iteration

- Computing the utilities can also be done by (simplified) **value iteration**;
- We can use a with a fixed number of iterations (we do not need exact utility values).

Modified Policy Iteration

- Computing the utilities can also be done by (simplified) **value iteration**;
- We can use a with a fixed number of iterations (we do not need exact utility values).
- It is also possible to only update the utility for subsets of the state set: e.g. or states likely to be reached, or predecessors of those that have been modified in the previous iteration.

Outline

Learning MDPs with Reinforcement Learning

- We want to learn an optimal strategy on an unknown MDP from observations;

Learning MDPs with Reinforcement Learning

- We want to learn an optimal strategy on an unknown MDP from observations;
- We can learn the MDP itself;

Learning MDPs with Reinforcement Learning

- We want to learn an optimal strategy on an unknown MDP from observations;
- We can learn the MDP itself;
- For some techniques, we can learn only states and possible actions;

Learning MDPs with Reinforcement Learning

- We want to learn an optimal strategy on an unknown MDP from observations;
- We can learn the MDP itself;
- For some techniques, we can learn only states and possible actions;
- In both cases, we will compute an (approximate) optimal policy using the idea of (generalised) **policy iteration**;

Learning MDPs with Reinforcement Learning

- We want to learn an optimal strategy on an unknown MDP from observations;
- We can learn the MDP itself;
- For some techniques, we can learn only states and possible actions;
- In both cases, we will compute an (approximate) optimal policy using the idea of (generalised) **policy iteration**;
- Starting from a random policy π_0, we repeat the following until $\pi_{n+1} = \pi_n$:

Learning MDPs with Reinforcement Learning

- We want to learn an optimal strategy on an unknown MDP from observations;
- We can learn the MDP itself;
- For some techniques, we can learn only states and possible actions;
- In both cases, we will compute an (approximate) optimal policy using the idea of (generalised) **policy iteration**;
- Starting from a random policy π_0, we repeat the following until $\pi_{n+1} = \pi_n$:
  1. Evaluate (possibly partially) the utilities for π_n;

Learning MDPs with Reinforcement Learning

- We want to learn an optimal strategy on an unknown MDP from observations;
- We can learn the MDP itself;
- For some techniques, we can learn only states and possible actions;
- In both cases, we will compute an (approximate) optimal policy using the idea of (generalised) **policy iteration**;
- Starting from a random policy π_0, we repeat the following until $\pi_{n+1} = \pi_n$:
  1. Evaluate (possibly partially) the utilities for π_n;
  2. Update π_0 using the new utilities and the MEU principle, giving π_{n+1}.

Learning States and Transition Probabilities

- We assume a policy π is fixed such that for any new state we discover we know which action to play;

Learning States and Transition Probabilities

- We assume a policy π is fixed such that for any new state we discover we know which action to play;
- We therefore learn a **Markov chain**;

Learning States and Transition Probabilities

- We assume a policy π is fixed such that for any new state we discover we know which action to play;
- We therefore learn a **Markov chain**;
- We simulate / execute the system and record **states** and **rewards** as we discover them;

Learning States and Transition Probabilities

- We assume a policy π is fixed such that for any new state we discover we know which action to play;
- We therefore learn a **Markov chain**;
- We simulate / execute the system and record **states** and **rewards** as we discover them;
- We need only **estimate** the transition probabilities;

Learning States and Transition Probabilities

- We assume a policy π is fixed such that for any new state we discover we know which action to play;
- We therefore learn a **Markov chain**;
- We simulate / execute the system and record **states** and **rewards** as we discover them;
- We need only **estimate** the transition probabilities;
- Once we have the model, we can learn utilities for π using **value iteration**.

Estimating Transition Probabilities

$$
\begin{tikzcd}
0 \arrow[r, "\theta"] \arrow[dr, "1-\theta"'] & 1 \\
& 2
\end{tikzcd}
$$

- We observe the numbers of times we go from one state to another and use **maximum a priori estimation** with a uniform prior (i.e., **maximum likelihood** estimation).

Estimating Transition Probabilities

$$\begin{tikzcd} 0 \arrow[r, "\theta"] \arrow[dr, "1-\theta"'] & 1 \\ & 2 \end{tikzcd}$$

- We observe the numbers of times we go from one state to another and use **maximum a priori estimation** with a uniform prior (i.e., **maximum likelihood** estimation).
- Suppose we have observed $n_1 > 0$ times $0 \to 1$ and $n_2 > 0$ times $0 \to 2$ (and denote by D that data);

Estimating Transition Probabilities

$$\begin{tikzcd} 0 \arrow[r, "\theta"] \arrow[dr, "1-\theta"'] & 1 \\ & 2 \end{tikzcd}$$

- We observe the numbers of times we go from one state to another and use **maximum a priori estimation** with a uniform prior (i.e., **maximum likelihood** estimation).
- Suppose we have observed $n_1 > 0$ times $0 \rightarrow 1$ and $n_2 > 0$ times $0 \rightarrow 2$ (and denote by D that data);
- We want the value of p that maximises $P(\theta = p|D)$;

Estimating Transition Probabilities

$$
\begin{tikzcd}
0 \arrow[r, "\theta"] \arrow[dr, "1-\theta"'] & 1 \\
& 2
\end{tikzcd}
$$

- We observe the numbers of times we go from one state to another and use **maximum a priori estimation** with a uniform prior (i.e., **maximum likelihood** estimation).
- Suppose we have observed $n_1 > 0$ times $0 \rightarrow 1$ and $n_2 > 0$ times $0 \rightarrow 2$ (and denote by D that data);
- We want the value of p that maximises $P(\theta = p|D)$;
- By Bayes' rule, this is maximising: $\frac{P(D|\theta=p)P(\theta=p)}{P(D)}$;

Estimating Transition Probabilities

$$\begin{tikzcd} 0 \arrow[r, "\theta"] \arrow[dr, "1-\theta"'] & 1 \\ & 2 \end{tikzcd}$$

- We observe the numbers of times we go from one state to another and use **maximum a priori estimation** with a uniform prior (i.e., **maximum likelihood** estimation).
- Suppose we have observed $n_1 > 0$ times $0 \to 1$ and $n_2 > 0$ times $0 \to 2$ (and denote by D that data);
- We want the value of p that maximises $P(\theta = p|D)$;
- By Bayes' rule, this is maximising: $\frac{P(D|\theta=p)P(\theta=p)}{P(D)}$;
- $P(D)$ is the sum of all $P(D|\theta = p)$ for all p and is **independent of p**;

Estimating Transition Probabilities

$$\begin{tikzcd} 0 \arrow[r, "\theta"] \arrow[dr, "1-\theta"'] & 1 \\ & 2 \end{tikzcd}$$

- We observe the numbers of times we go from one state to another and use **maximum a priori estimation** with a uniform prior (i.e., **maximum likelihood** estimation).
- Suppose we have observed $n_1 > 0$ times $0 \to 1$ and $n_2 > 0$ times $0 \to 2$ (and denote by D that data);
- We want the value of p that maximises $P(\theta = p|D)$;
- By Bayes' rule, this is maximising: $\frac{P(D|\theta=p)P(\theta=p)}{P(D)}$;
- $P(D)$ is the sum of all $P(D|\theta = p)$ for all p and is **independent of p**;
- We suppose that **a priori** all values of p are **equally likely**;

Estimating Transition Probabilities

$$\begin{tikzcd} 0 \arrow[r, "\theta"] \arrow[dr, "1-\theta"'] & 1 \\ & 2 \end{tikzcd}$$

- We observe the numbers of times we go from one state to another and use **maximum a priori estimation** with a uniform prior (i.e., **maximum likelihood** estimation).
- Suppose we have observed $n_1 > 0$ times $0 \rightarrow 1$ and $n_2 > 0$ times $0 \rightarrow 2$ (and denote by D that data);
- We want the value of p that maximises $P(\theta = p|D)$;
- By Bayes' rule, this is maximising: $\frac{P(D|\theta=p)P(\theta=p)}{P(D)}$;
- $P(D)$ is the sum of all $P(D|\theta = p)$ for all p and is **independent of p**;
- We suppose that **a priori** all values of p are **equally likely**;
- We therefore need to maximise $P(D|\theta = p)$.

Estimating Transition Probabilities

- We have $P(D|\theta = p) = p^{n_1}(1-p)^{n_2}$;

Estimating Transition Probabilities

- We have $P(D|\theta = p) = p^{n_1}(1-p)^{n_2}$;
- Since the log function is increasing we can equivalently maximise

$$\log\big(P(D|\theta = p)\big) = n_1 \log(p) + n_2 \log(1-p)$$

Estimating Transition Probabilities

- We have $P(D|\theta = p) = p^{n_1}(1-p)^{n_2}$;
- Since the log function is increasing we can equivalently maximise

$$\log\big(P(D|\theta = p)\big) = n_1 \log(p) + n_2 \log(1-p)$$

- The derivative is:

$$\frac{n_1}{p} - \frac{n_2}{1-p}$$

Estimating Transition Probabilities

- We have $P(D|\theta = p) = p^{n_1}(1-p)^{n_2}$;
- Since the log function is increasing we can equivalently maximise

$$\log\big(P(D|\theta = p)\big) = n_1 \log(p) + n_2 \log(1-p)$$

- The derivative is:

$$\frac{n_1}{p} - \frac{n_2}{1-p}$$

- It is zero when:

$$(1-p)n_1 - pn_2 = 0$$

Estimating Transition Probabilities

- We have $P(D|\theta = p) = p^{n_1}(1-p)^{n_2}$;
- Since the log function is increasing we can equivalently maximise

$$\log\big(P(D|\theta = p)\big) = n_1 \log(p) + n_2 \log(1-p)$$

- The derivative is:

$$\frac{n_1}{p} - \frac{n_2}{1-p}$$

- It is zero when:

$$(1-p)n_1 - pn_2 = 0$$

- and finally:

$$\boxed{p = \frac{n_1}{n_1+n_2}}$$

Learning Utilities: Monte-Carlo

- We assume the strategy is fixed;

Learning Utilities: Monte-Carlo

- We assume the strategy is fixed;
- We simulate / execute the system and want to learn the utility of the states we discover;

Learning Utilities: Monte-Carlo

- We assume the strategy is fixed;
- We simulate / execute the system and want to learn the utility of the states we discover;
- Recall the utility of a state is:

$$U^\pi(s) = E\Big[\sum_{t=0}^{\infty} \gamma^t R(S_t)\Big]$$

Learning Utilities: Monte-Carlo

- We assume the strategy is fixed;
- We simulate / execute the system and want to learn the utility of the states we discover;
- Recall the utility of a state is:

$$U^\pi(s) = E\big[\sum_{t=0}^{\infty} \gamma^t R(S_t)\big]$$

- We need to estimate the expectation:

Learning Utilities: Monte-Carlo

- We assume the strategy is fixed;
- We simulate / execute the system and want to learn the utility of the states we discover;
- Recall the utility of a state is:

$$U^\pi(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t)\right]$$

- We need to estimate the expectation:
  - Consider n executions and the n corresponding families of random variables $(S_t^i)_{t \geq 0}$, $i \in [1..n]$;

Learning Utilities: Monte-Carlo

- We assume the strategy is fixed;
- We simulate / execute the system and want to learn the utility of the states we discover;
- Recall the utility of a state is:

$$U^\pi(s) = E\big[\sum_{t=0}^{\infty} \gamma^t R(S_t)\big]$$

- We need to estimate the expectation:
  - Consider n executions and the n corresponding families of random variables $(S_t^i)_{t\geq 0}$, $i \in [1..n]$;
  - By the **law of large numbers**, the average (on i) of the $\sum_{t=0}^{\infty} \gamma^t R(S_t^i)$ converges to the utility;

Learning Utilities: Monte-Carlo

- We assume the strategy is fixed;
- We simulate / execute the system and want to learn the utility of the states we discover;
- Recall the utility of a state is:

$$U^\pi(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t)\right]$$

- We need to estimate the expectation:
  - Consider n executions and the n corresponding families of random variables $(S_t^i)_{t\geq 0}$, $i \in [1..n]$;
  - By the **law of large numbers**, the average (on i) of the $\sum_{t=0}^{\infty} \gamma^t R(S_t^i)$ converges to the utility;
- We need to approximate the infinite sum:

Learning Utilities: Monte-Carlo

- We assume the strategy is fixed;
- We simulate / execute the system and want to learn the utility of the states we discover;
- Recall the utility of a state is:

$$U^\pi(s) = E\Big[\sum_{t=0}^{\infty} \gamma^t R(S_t)\Big]$$

- We need to estimate the expectation:
  - Consider n executions and the n corresponding families of random variables $(S_t^i)_{t\geq 0}$, $i \in [1..n]$;
  - By the **law of large numbers**, the average (on i) of the $\sum_{t=0}^{\infty} \gamma^t R(S_t^i)$ converges to the utility;
- We need to approximate the infinite sum:
  - For all k, $\sum_{t=0}^{k} \gamma^t R(S_t^i)$ is an approximation of the true utility of the execution;

Learning Utilities: Monte-Carlo

- We assume the strategy is fixed;
- We simulate / execute the system and want to learn the utility of the states we discover;
- Recall the utility of a state is:

$$U^{\pi}(s) = E\Big[\sum_{t=0}^{\infty} \gamma^t R(S_t)\Big]$$

- We need to estimate the expectation:
  - Consider n executions and the n corresponding families of random variables $(S_t^i)_{t \geq 0}$, $i \in [1..n]$;
  - By the **law of large numbers**, the average (on i) of the $\sum_{t=0}^{\infty} \gamma^t R(S_t^i)$ converges to the utility;
- We need to approximate the infinite sum:
  - For all k, $\sum_{t=0}^{k} \gamma^t R(S_t^i)$ is an approximation of the true utility of the execution;
  - Given $\epsilon > 0$, if $k \geq \frac{\log\left(\frac{\epsilon(1-\gamma)}{R_{\max}}\right)}{\log(\gamma)}$, then $|\sum_{t=0}^{\infty} \gamma^t R(S_t^i) - \sum_{t=0}^{k} \gamma^t R(S_t^i)| < \epsilon$;

Learning Utilities: Monte-Carlo

- We assume the strategy is fixed;
- We simulate / execute the system and want to learn the utility of the states we discover;
- Recall the utility of a state is:

$$U^\pi(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t)\right]$$

- We need to estimate the expectation:
  - Consider n executions and the n corresponding families of random variables $(S_t^i)_{t\geq 0}$, $i \in [1..n]$;
  - By the **law of large numbers**, the average (on i) of the $\sum_{t=0}^{\infty} \gamma^t R(S_t^i)$ converges to the utility;
- We need to approximate the infinite sum:
  - For all k, $\sum_{t=0}^{k} \gamma^t R(S_t^i)$ is an approximation of the true utility of the execution;
  - Given $\epsilon > 0$, if $k \geq \frac{\log\left(\frac{\epsilon(1-\gamma)}{R_{\max}}\right)}{\log(\gamma)}$, then $|\sum_{t=0}^{\infty} \gamma^t R(S_t^i) - \sum_{t=0}^{k} \gamma^t R(S_t^i)| < \epsilon$;
  - And the average is also precise up to ϵ.

Learning Utilities: Monte-Carlo – Terminal States

- If a simulation ends up in a **terminal** state then it is finite;

Learning Utilities: Monte-Carlo – Terminal States

- If a simulation ends up in a **terminal** state then it is finite;
- All its **suffixes** are also finite and may be used to approximate the utility of the their starting state;

Learning Utilities: Monte-Carlo – Terminal States

- If a simulation ends up in a **terminal** state then it is finite;
- All its **suffixes** are also finite and may be used to approximate the utility of the their starting state;
- When reachability of terminal states is **guaranteed**, we therefore simulate until we reach a terminal state;

Learning Utilities: Monte-Carlo – Terminal States

- If a simulation ends up in a **terminal** state then it is finite;
- All its **suffixes** are also finite and may be used to approximate the utility of the their starting state;
- When reachability of terminal states is **guaranteed**, we therefore simulate until we reach a terminal state;
- We can then **update** the utility of **all** the states visited.

Learning Utilities: Q-functions

- Since the policy π is fixed, we actually learn the utility of doing the **prescribed** action $\pi(s)$ in state s;

Learning Utilities: Q-functions

- Since the policy π is fixed, we actually learn the utility of doing the **prescribed** action $\pi(s)$ in state s;
- We thus learn a function $Q(s, a)$ for $a = \pi(s)$;

Learning Utilities: Q-functions

- Since the policy π is fixed, we actually learn the utility of doing the **prescribed** action $\pi(s)$ in state s;
- We thus learn a function $Q(s, a)$ for $a = \pi(s)$;
- Q is called a ***Q*-function**;

Learning Utilities: Q-functions

- Since the policy π is fixed, we actually learn the utility of doing the **prescribed** action $\pi(s)$ in state s;
- We thus learn a function $Q(s, a)$ for $a = \pi(s)$;
- Q is called a ***Q*-function**;
- It includes the immediate reward and the discount factor;

Learning Utilities: Q-functions

- Since the policy π is fixed, we actually learn the utility of doing the **prescribed** action $\pi(s)$ in state s;
- We thus learn a function $Q(s, a)$ for $a = \pi(s)$;
- Q is called a ***Q*-function**;
- It includes the immediate reward and the discount factor;
- We have:

$$U(s) = \max_{a \in A} Q(s, a)$$

Learning Utilities: Temporal Differences

- Recall the Bellman equation for a fixed strategy π:

$$U^{\pi}(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi(s)) U^{\pi}(s')$$

Learning Utilities: Temporal Differences

- Recall the Bellman equation for a fixed strategy π:

$$U^{\pi}(s) = R(s) + \gamma \sum_{s'} P(s'|s,\pi(s))U^{\pi}(s')$$

- That is:

$$U^{\pi}(s) = E\left[R(s) + \gamma U^{\pi}(s')\right]$$

Learning Utilities: Temporal Differences

- Recall the Bellman equation for a fixed strategy π:

$$U^{\pi}(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi(s)) U^{\pi}(s')$$

- That is:

$$U^{\pi}(s) = E\Big[R(s) + \gamma U^{\pi}(s')\Big]$$

- As usual, we can estimate the expectation by an **average** over random experiments;

Learning Utilities: Temporal Differences

- Recall the Bellman equation for a fixed strategy π:

$$U^{\pi}(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi(s)) U^{\pi}(s')$$

- That is:

$$U^{\pi}(s) = E\left[R(s) + \gamma U^{\pi}(s')\right]$$

- As usual, we can estimate the expectation by an **average** over random experiments;
- While simulating/executing the system, for observed transition number n from s to s', we compute:

Learning Utilities: Temporal Differences

- Recall the Bellman equation for a fixed strategy π:

$$U^\pi(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi(s)) U^\pi(s')$$

- That is:

$$U^\pi(s) = E\Big[R(s) + \gamma U^\pi(s')\Big]$$

- As usual, we can estimate the expectation by an **average** over random experiments;
- While simulating/executing the system, for observed transition number n from s to s', we compute:
  - value of the n-th random experiment: $V_n^\pi(s) = R(s) + \gamma U_{n-1}^\pi(s')$;

Learning Utilities: Temporal Differences

- Recall the Bellman equation for a fixed strategy π:

$$U^\pi(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi(s)) U^\pi(s')$$

- That is:

$$U^\pi(s) = E\Big[R(s) + \gamma U^\pi(s')\Big]$$

- As usual, we can estimate the expectation by an **average** over random experiments;
- While simulating/executing the system, for observed transition number n from s to s', we compute:
  - value of the n-th random experiment: $V_n^\pi(s) = R(s) + \gamma U_{n-1}^\pi(s')$;
  - utility after n experiments: $U_n^\pi(s) = \frac{1}{n} \sum_{k=1}^{n} V_k^\pi(s)$.

Learning Utilities: Temporal Differences

- Note that:

$$U_n^\pi(s) = \frac{1}{n}\sum_{k=1}^{n} V_k^\pi(s) = \frac{1}{n}\left(V_n^\pi + \sum_{k=1}^{n-1} V_k^\pi(s)\right)$$

Learning Utilities: Temporal Differences

- Note that:

$$U_n^\pi(s) = \frac{1}{n}\sum_{k=1}^{n} V_k^\pi(s) = \frac{1}{n}\left(V_n^\pi + \sum_{k=1}^{n-1} V_k^\pi(s)\right)$$

- That is:

$$U_n^\pi(s) = \frac{1}{n}\left(V_n^\pi + (n-1)U_{n-1}^\pi(s)\right)$$

Learning Utilities: Temporal Differences

- Note that:

$$U_n^\pi(s) = \frac{1}{n}\sum_{k=1}^{n} V_k^\pi(s) = \frac{1}{n}\left(V_n^\pi + \sum_{k=1}^{n-1} V_k^\pi(s)\right)$$

- That is:

$$U_n^\pi(s) = \frac{1}{n}\left(V_n^\pi + (n-1)U_{n-1}^\pi(s)\right)$$

- Or:

$$U_n^\pi(s) = U_{n-1}^\pi(s) + \frac{1}{n}\left(V_n^\pi - U_{n-1}^\pi(s)\right)$$

Learning Utilities: Temporal Differences

- Note that:

$$U_n^\pi(s) = \frac{1}{n}\sum_{k=1}^{n} V_k^\pi(s) = \frac{1}{n}\left(V_n^\pi + \sum_{k=1}^{n-1} V_k^\pi(s)\right)$$

- That is:

$$U_n^\pi(s) = \frac{1}{n}\left(V_n^\pi + (n-1)U_{n-1}^\pi(s)\right)$$

- Or:

$$U_n^\pi(s) = U_{n-1}^\pi(s) + \frac{1}{n}\left(V_n^\pi - U_{n-1}^\pi(s)\right)$$

- Finally, with $\alpha_n = \frac{1}{n}$:

$$U_n^\pi(s) = U_{n-1}^\pi(s) + \alpha_n\left(R(s) - U_{n-1}^\pi(s) + \gamma U_{n-1}^\pi(s')\right)$$

Learning Utilities: Temporal Differences

- Note that:

$$U_n^\pi(s) = \frac{1}{n}\sum_{k=1}^{n} V_k^\pi(s) = \frac{1}{n}\left(V_n^\pi + \sum_{k=1}^{n-1} V_k^\pi(s)\right)$$

- That is:

$$U_n^\pi(s) = \frac{1}{n}\left(V_n^\pi + (n-1)U_{n-1}^\pi(s)\right)$$

- Or:

$$U_n^\pi(s) = U_{n-1}^\pi(s) + \frac{1}{n}\left(V_n^\pi - U_{n-1}^\pi(s)\right)$$

- Finally, with $\alpha_n = \frac{1}{n}$:

$$U_n^\pi(s) = U_{n-1}^\pi(s) + \alpha_n\left(R(s) - U_{n-1}^\pi(s) + \gamma U_{n-1}^\pi(s')\right)$$

- Or equivalently:

$$U_n^\pi(s) = (1-\alpha_n)U_{n-1}^\pi(s) + \alpha_n\left(R(s) + \gamma U_{n-1}^\pi(s')\right)$$

Learning Utilities: Temporal Differences

- The **TD algorithm** for learning utilities for policy π is then:

Learning Utilities: Temporal Differences

- The **TD algorithm** for learning utilities for policy π is then:

TD Algorithm

Learning Utilities: Temporal Differences

- The **TD algorithm** for learning utilities for policy π is then:

TD Algorithm

- Initialise arbitrarily $U_0^\pi(s)$, for all s;

Learning Utilities: Temporal Differences

- The **TD algorithm** for learning utilities for policy π is then:

TD Algorithm

- Initialise arbitrarily $U_0^\pi(s)$, for all s;
- Choose a decreasing learning rate sequence $\forall n, 0 < \alpha_n < 1$ with:

$$\sum_{t=1}^{\infty} \alpha(t) = \infty \text{ and } \sum_{t=1}^{\infty} \alpha(t)^2 < \infty$$

Learning Utilities: Temporal Differences

- The **TD algorithm** for learning utilities for policy π is then:

TD Algorithm

- Initialise arbitrarily $U_0^\pi(s)$, for all s;
- Choose a decreasing learning rate sequence $\forall n, 0 < \alpha_n < 1$ with:

$$\sum_{t=1}^{\infty} \alpha(t) = \infty \text{ and } \sum_{t=1}^{\infty} \alpha(t)^2 < \infty$$

- Simulate/execute the system according to π and at each step from s to s', compute:

$$U_n^\pi(s) = (1-\alpha_n)U_{n-1}^\pi(s) + \alpha_n \left(R(s) + \gamma U_{n-1}^\pi(s')\right)$$

Learning Utilities: Temporal Differences

- The **TD algorithm** for learning utilities for policy π is then:

TD Algorithm

- Initialise arbitrarily $U_0^\pi(s)$, for all s;
- Choose a decreasing learning rate sequence $\forall n, 0 < \alpha_n < 1$ with:

$$\sum_{t=1}^{\infty} \alpha(t) = \infty \text{ and } \sum_{t=1}^{\infty} \alpha(t)^2 < \infty$$

- Simulate/execute the system according to π and at each step from s to s', compute:

$$U_n^\pi(s) = (1 - \alpha_n) U_{n-1}^\pi(s) + \alpha_n \left(R(s) + \gamma U_{n-1}^\pi(s') \right)$$

- Then U_n^π converges towards U^π.

Learning Utilities: TD with non-stationarity

- In a more general setting, where **rewards may evolve** with time, we might want to give a **greater weight** to the **more recent** observations;

Learning Utilities: TD with non-stationarity

- In a more general setting, where **rewards may evolve** with time, we might want to give a **greater weight** to the **more recent** observations;
- Consider that α_n is a constant $\alpha < 1$:

$$U_n^\pi(s) = U_{n-1}^\pi(s) + \alpha\left(V_n^\pi - U_{n-1}^\pi(s)\right)$$

Learning Utilities: TD with non-stationarity

- In a more general setting, where **rewards may evolve** with time, we might want to give a **greater weight** to the **more recent** observations;
- Consider that α_n is a constant $\alpha < 1$:

$$U_n^\pi(s) = U_{n-1}^\pi(s) + \alpha\left(V_n^\pi - U_{n-1}^\pi(s)\right)$$

- Then:

$$U_n^\pi(s) = \alpha V_n^\pi + (1-\alpha)U_{n-1}^\pi(s)$$

Learning Utilities: TD with non-stationarity

- In a more general setting, where **rewards may evolve** with time, we might want to give a **greater weight** to the **more recent** observations;
- Consider that α_n is a constant $\alpha < 1$:

$$U_n^\pi(s) = U_{n-1}^\pi(s) + \alpha\left(V_n^\pi - U_{n-1}^\pi(s)\right)$$

- Then:

$$U_n^\pi(s) = \alpha V_n^\pi + (1-\alpha)U_{n-1}^\pi(s)$$

- Or:

$$U_n^\pi(s) = \alpha V_n^\pi + (1-\alpha)\big(\alpha V_{n-1}^\pi + (1-\alpha)U_{n-2}^\pi(s)\big)$$

Learning Utilities: TD with non-stationarity

- In a more general setting, where **rewards may evolve** with time, we might want to give a **greater weight** to the **more recent** observations;
- Consider that α_n is a constant $\alpha < 1$:

$$U_n^\pi(s) = U_{n-1}^\pi(s) + \alpha\left(V_n^\pi - U_{n-1}^\pi(s)\right)$$

- Then:

$$U_n^\pi(s) = \alpha V_n^\pi + (1-\alpha)U_{n-1}^\pi(s)$$

- Or:

$$U_n^\pi(s) = \alpha V_n^\pi + (1-\alpha)\big(\alpha V_{n-1}^\pi + (1-\alpha)U_{n-2}^\pi(s)\big)$$

- And so on to get:

$$U_n^\pi(s) = \alpha \sum_{k=1}^{n} (1-\alpha)^k V_k^\pi + (1-\alpha)^n U_0^\pi(s)$$

Learning Utilities: TD with non-stationarity

- In a more general setting, where **rewards may evolve** with time, we might want to give a **greater weight** to the **more recent** observations;
- Consider that α_n is a constant $\alpha < 1$:

$$U_n^\pi(s) = U_{n-1}^\pi(s) + \alpha\left(V_n^\pi - U_{n-1}^\pi(s)\right)$$

- Then:

$$U_n^\pi(s) = \alpha V_n^\pi + (1-\alpha)U_{n-1}^\pi(s)$$

- Or:

$$U_n^\pi(s) = \alpha V_n^\pi + (1-\alpha)\big(\alpha V_{n-1}^\pi + (1-\alpha)U_{n-2}^\pi(s)\big)$$

- And so on to get:

$$U_n^\pi(s) = \alpha\sum_{k=1}^{n}(1-\alpha)^k V_k^\pi + (1-\alpha)^n U_0^\pi(s)$$

- With:

$$\alpha\sum_{k=1}^{n}(1-\alpha)^k + (1-\alpha)^n = \alpha\frac{1-(1-\alpha)^n}{1-(1-\alpha)} + (1-\alpha)^n = 1$$

Learning Utilities: TD with non-stationarity

- In a more general setting, where **rewards may evolve** with time, we might want to give a **greater weight** to the **more recent** observations;
- Consider that α_n is a constant $\alpha < 1$:

$$U_n^\pi(s) = U_{n-1}^\pi(s) + \alpha\left(V_n^\pi - U_{n-1}^\pi(s)\right)$$

- Then:

$$U_n^\pi(s) = \alpha V_n^\pi + (1-\alpha)U_{n-1}^\pi(s)$$

- Or:

$$U_n^\pi(s) = \alpha V_n^\pi + (1-\alpha)\big(\alpha V_{n-1}^\pi + (1-\alpha)U_{n-2}^\pi(s)\big)$$

- And so on to get:

$$U_n^\pi(s) = \alpha\sum_{k=1}^{n}(1-\alpha)^k V_k^\pi + (1-\alpha)^n U_0^\pi(s)$$

- With:

$$\alpha\sum_{k=1}^{n}(1-\alpha)^k + (1-\alpha)^n = \alpha\frac{1-(1-\alpha)^n}{1-(1-\alpha)} + (1-\alpha)^n = 1$$

- We only have convergence in average over n but it works well in practice.

Learning Action-State Utilities: SARSA

- One of the particularities of TD learning is that it does **not** need a **model** (i.e. transition probabilities);

Learning Action-State Utilities: SARSA

- One of the particularities of TD learning is that it does **not** need a **model** (i.e. transition probabilities);
- Yet when we want to update the policy, we need to select the policy using the MEU principle;

Learning Action-State Utilities: SARSA

- One of the particularities of TD learning is that it does **not** need a **model** (i.e. transition probabilities);
- Yet when we want to update the policy, we need to select the policy using the MEU principle;
- To choose the best **expected** utility we need the transition probabilities;

Learning Action-State Utilities: SARSA

- One of the particularities of TD learning is that it does **not** need a **model** (i.e. transition probabilities);
- Yet when we want to update the policy, we need to select the policy using the MEU principle;
- To choose the best **expected** utility we need the transition probabilities;
- Instead we can directly use TD to compute state action utilities $Q(s, a)$;

Learning Action-State Utilities: SARSA

- One of the particularities of TD learning is that it does **not** need a **model** (i.e. transition probabilities);
- Yet when we want to update the policy, we need to select the policy using the MEU principle;
- To choose the best **expected** utility we need the transition probabilities;
- Instead we can directly use TD to compute state action utilities $Q(s, a)$;
- When moving from s to s' with action a, and if the next action selected from s' is a', we have:

$$\boxed{Q_n^\pi(s, a) = (1 - \alpha_n) Q_{n-1}^\pi(s, a) + \alpha_n \left(R(s) + \gamma Q_{n-1}^\pi(s', a')\right)}$$

Learning Action-State Utilities: SARSA

- One of the particularities of TD learning is that it does **not** need a **model** (i.e. transition probabilities);
- Yet when we want to update the policy, we need to select the policy using the MEU principle;
- To choose the best **expected** utility we need the transition probabilities;
- Instead we can directly use TD to compute state action utilities $Q(s, a)$;
- When moving from s to s' with action a, and if the next action selected from s' is a', we have:

$$\boxed{Q_n^\pi(s, a) = (1 - \alpha_n)Q_{n-1}^\pi(s, a) + \alpha_n \left(R(s) + \gamma Q_{n-1}^\pi(s', a')\right)}$$

- If s' is terminal then we take $Q_{n-1}^\pi(s', a') = 0$;

Learning Action-State Utilities: SARSA

- One of the particularities of TD learning is that it does **not** need a **model** (i.e. transition probabilities);
- Yet when we want to update the policy, we need to select the policy using the MEU principle;
- To choose the best **expected** utility we need the transition probabilities;
- Instead we can directly use TD to compute state action utilities $Q(s, a)$;
- When moving from s to s' with action a, and if the next action selected from s' is a', we have:

$$\boxed{Q^{\pi}_{n}(s, a) = (1 - \alpha_n)Q^{\pi}_{n-1}(s, a) + \alpha_n \left(R(s) + \gamma Q^{\pi}_{n-1}(s', a')\right)}$$

- If s' is terminal then we take $Q^{\pi}_{n-1}(s', a') = 0$;
- Updating the policy at s then consists only in choosing $a = \text{argmax}_b Q(s, b)$;

Learning Action-State Utilities: SARSA

- One of the particularities of TD learning is that it does **not** need a **model** (i.e. transition probabilities);
- Yet when we want to update the policy, we need to select the policy using the MEU principle;
- To choose the best **expected** utility we need the transition probabilities;
- Instead we can directly use TD to compute state action utilities $Q(s, a)$;
- When moving from s to s' with action a, and if the next action selected from s' is a', we have:

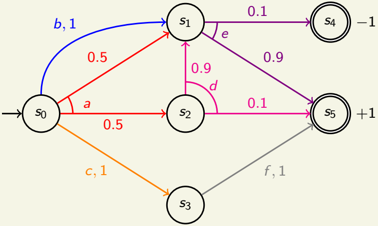
$$\boxed{Q^{\pi}_{n}(s, a) = (1 - \alpha_n)Q^{\pi}_{n-1}(s, a) + \alpha_n \left(R(s) + \gamma Q^{\pi}_{n-1}(s', a')\right)}$$

- If s' is terminal then we take $Q^{\pi}_{n-1}(s', a') = 0$;
- Updating the policy at s then consists only in choosing $a = \text{argmax}_b Q(s, b)$;
- This is the *State Action Reward State Action* (SARSA) algorithm.

Choice of policies

- When we know utilities for all states, we can use the MEU principle to improve the strategy, like in **policy iteration**;

Exercise

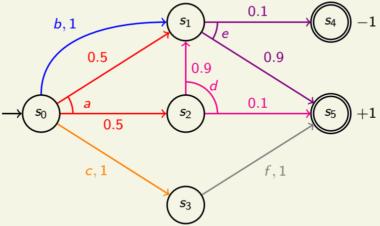


Assume the policy π is in s_0 do a (and the rest is forced).

Choice of policies

- When we know utilities for all states, we can use the MEU principle to improve the strategy, like in **policy iteration**;

Exercise



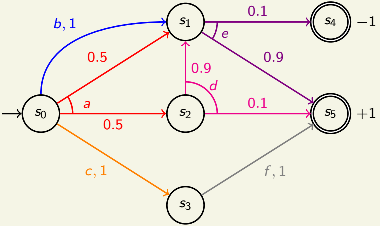
Assume the policy π is in s_0 do a (and the rest is forced).

1. How do the utilities for s_1, s_2, and s_3 computed with π fixed (using any of the previous methods, with utilities initially 0) compare?

Choice of policies

- When we know utilities for all states, we can use the MEU principle to improve the strategy, like in **policy iteration**;

Exercise



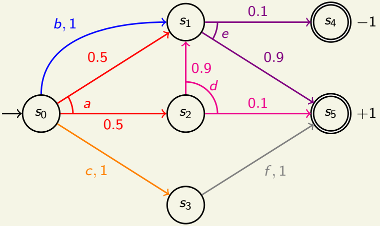
Assume the policy π is in s_0 do a (and the rest is forced).

1. How do the utilities for s_1, s_2, and s_3 computed with π fixed (using any of the previous methods, with utilities initially 0) compare?
2. What is the new MEU policy? How will utilities evolve then?

Choice of policies

- When we know utilities for all states, we can use the MEU principle to improve the strategy, like in **policy iteration**;

Exercise



Assume the policy π is in s_0 do a (and the rest is forced).

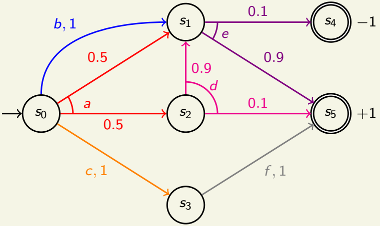
1. How do the utilities for s_1, s_2, and s_3 computed with π fixed (using any of the previous methods, with utilities initially 0) compare?
2. What is the new MEU policy? How will utilities evolve then?

- We cannot rely only on the utilities computed given π:

Choice of policies

- When we know utilities for all states, we can use the MEU principle to improve the strategy, like in **policy iteration**;

Exercise



Assume the policy π is in s_0 do a (and the rest is forced).

1. How do the utilities for s_1, s_2, and s_3 computed with π fixed (using any of the previous methods, with utilities initially 0) compare?
2. What is the new MEU policy? How will utilities evolve then?

- We cannot rely only on the utilities computed given π:
- We need to explore **beyond** π and be sure to visit **all states** often enough.

Exploration and Exploitation

- We might want to forget about π and choose actions randomly in all states;

Exploration and Exploitation

- We might want to forget about π and choose actions randomly in all states;
- But as our estimates get better, we would like to focus on the most promising parts;

Exploration and Exploitation

- We might want to forget about π and choose actions randomly in all states;
- But as our estimates get better, we would like to focus on the most promising parts;
- The idea is then to be **Greedy in the Limit of Infinite Exploration** (GLIE):

Exploration and Exploitation

- We might want to forget about π and choose actions randomly in all states;
- But as our estimates get better, we would like to focus on the most promising parts;
- The idea is then to be **Greedy in the Limit of Infinite Exploration** (GLIE):
  1. start with a fair amount of exploration;

Exploration and Exploitation

- We might want to forget about π and choose actions randomly in all states;
- But as our estimates get better, we would like to focus on the most promising parts;
- The idea is then to be **Greedy in the Limit of Infinite Exploration** (GLIE):
  1. start with a fair amount of exploration;
  2. explore less and less as we get more and more precise information we get.

Exploration and Exploitation

- We might want to forget about π and choose actions randomly in all states;
- But as our estimates get better, we would like to focus on the most promising parts;
- The idea is then to be **Greedy in the Limit of Infinite Exploration** (GLIE):
    1. start with a fair amount of exploration;
    2. explore less and less as we get more and more precise information we get.
- We can use a variety of such schemes, including UCB1, ϵ-greedy policies, and softmax-based policies.

Exploration and Exploitation: ϵ-greedy policies

- We consider **probabilistic** policies π, in which $\pi(s, a)$ is the probability of choosing action a when in state s;

Exploration and Exploitation: ϵ-greedy policies

- We consider **probabilistic** policies π, in which $\pi(s, a)$ is the probability of choosing action a when in state s;
- A (probabilistic) policy π is ϵ-greedy if for all states s:
  $|A(s)|$ is the number of actions enabled in state s

Exploration and Exploitation: ϵ-greedy policies

- We consider **probabilistic** policies π, in which $\pi(s, a)$ is the probability of choosing action a when in state s;
- A (probabilistic) policy π is ϵ-greedy if for all states s:
  $|A(s)|$ is the number of actions enabled in state s
  1. there exist an action a such that $\pi(s, a) = 1 - \epsilon + \frac{\epsilon}{|A(s)|}$

Exploration and Exploitation: ϵ-greedy policies

- We consider **probabilistic** policies π, in which $\pi(s, a)$ is the probability of choosing action a when in state s;
- A (probabilistic) policy π is ϵ-greedy if for all states s:
  $|A(s)|$ is the number of actions enabled in state s
  1. there exist an action a such that $\pi(s, a) = 1 - \epsilon + \frac{\epsilon}{|A(s)|}$
  2. for all actions $b \neq a$, $\pi(s, b) = \frac{\epsilon}{|A(s)|}$

Exploration and Exploitation: ϵ-greedy policies

- We consider **probabilistic** policies π, in which $\pi(s, a)$ is the probability of choosing action a when in state s;
- A (probabilistic) policy π is ϵ-greedy if for all states s:
  $|A(s)|$ is the number of actions enabled in state s
  1. there exist an action a such that $\pi(s, a) = 1 - \epsilon + \frac{\epsilon}{|A(s)|}$
  2. for all actions $b \neq a$, $\pi(s, b) = \frac{\epsilon}{|A(s)|}$
- When updating the policy using the MEU principle, we always transform it into an ϵ-greedy policy.

Exploration and Exploitation: softmax-based policies

- In the MEU principle we replace the max function by the **softmax** function;

Exploration and Exploitation: softmax-based policies

- In the MEU principle we replace the max function by the **softmax** function;
- This gives us a probabilistic policy, for $T > 0$:

$$\pi(s, a) = \frac{e^{\frac{\sum_{s'} P(s'|s,a)U(s')}{T}}}{\sum_b e^{\frac{\sum_{s'} P(s'|s,b)U(s')}{T}}}$$

Exploration and Exploitation: softmax-based policies

- In the MEU principle we replace the max function by the **softmax** function;
- This gives us a probabilistic policy, for $T > 0$:

$$\pi(s,a) = \frac{e^{\frac{\sum_{s'} P(s'|s,a)U(s')}{T}}}{\sum_b e^{\frac{\sum_{s'} P(s'|s,b)U(s')}{T}}}$$

- Or if we have computed state-action utilities $Q(s,a)$ instead of state utilities $U(s)$;

$$\pi(s,a) = \frac{e^{\frac{Q(s,a)}{T}}}{\sum_b e^{\frac{Q(s,b)}{T}}}$$

Exploration and Exploitation: softmax-based policies

- In the MEU principle we replace the max function by the **softmax** function;
- This gives us a probabilistic policy, for $T > 0$:

$$\pi(s, a) = \frac{e^{\frac{\sum_{s'} P(s'|s,a)U(s')}{T}}}{\sum_b e^{\frac{\sum_{s'} P(s'|s,b)U(s')}{T}}}$$

- Or if we have computed state-action utilities $Q(s, a)$ instead of state utilities $U(s)$;

$$\pi(s, a) = \frac{e^{\frac{Q(s,a)}{T}}}{\sum_b e^{\frac{Q(s,b)}{T}}}$$

- When T tends towards 0, the policy is nearly **deterministic**;

Exploration and Exploitation: softmax-based policies

- In the MEU principle we replace the max function by the **softmax** function;
- This gives us a probabilistic policy, for $T > 0$:

$$\pi(s,a) = \frac{e^{\frac{\sum_{s'} P(s'|s,a)U(s')}{T}}}{\sum_b e^{\frac{\sum_{s'} P(s'|s,b)U(s')}{T}}}$$

- Or if we have computed state-action utilities $Q(s,a)$ instead of state utilities $U(s)$;

$$\pi(s,a) = \frac{e^{\frac{Q(s,a)}{T}}}{\sum_b e^{\frac{Q(s,b)}{T}}}$$

- When T tends towards 0, the policy is nearly **deterministic**;
- When T tends towards $+\infty$, the policy is nearly **uniformly random**.

Off-policy learning

- The utility estimation schemes above do **on-policy** learning: evaluate the policy by playing it;
- To ensure exploration, we modify slightly the obtained MEU policies;
- We hope the result stays close to the original;
- It is sometimes desirable to completely separate the policies for exploration and exploitation;
- Learn a policy while exploring according to another: **off-policy** learning.

Off-policy Monte-Carlo: Importance sampling

- We want to estimate:

$$U^\pi(s) = E\Big[\sum_{t=0}^{\infty} \gamma^t R(S_t)\Big]$$

- This is the limit of the expectation on finite runs of length n when $n \to +\infty$:
  $P^\pi(\rho)$ is the probability of run ρ under policy π, and $R(\rho)$ its total reward

$$\sum_{|\rho|=n} P^\pi(\rho) R(\rho)$$

- Now assume we explore using a different policy π'. What we estimate with Monte-Carlo is:

$$\sum_{|\rho|=n} P^{\pi'}(\rho) R(\rho)$$

- So we need to correct this by multiplying the rewards by the importance-sampling ratios $\frac{P^\pi(\rho)}{P^{\pi'}(\rho)}$
- Note that the ratios depend only on the policies, not the MDP, as transition probabilities cancel out.

Off-policy Monte-Carlo: Importance sampling

- We can as before estimate the expectation with an average over runs obtained randomly: **ordinary importance sampling** (OIS)
- Or we can divide the sum of the rewards by the sum of the importance sampling ratios of the runs (instead of their number): **weighted importance sampling** (WIS)
- For any number of runs n:
  - OIS is unbiased: its expectation is the desired value
  - the variance of OIS is unbounded
  - WIS is biased but the bias goes to 0 with $n \to +\infty$;
  - the variance of OIS is bounded, and converges to 0 with $n \to +\infty$ when $R(\rho)$ is bounded.
- In practice, WIS is strongly preferred.

Off-policy TD: Q-learning

- In SARSA, the utility of the destination state is estimated according to the action that will be chosen there according to the current policy: $Q(s', a')$;
- But a' is not necessarily the best action to do in s';
- With the current knowledge this is $\text{argmax}_b Q(s', b)$;
- **Q-learning** uses this observation to compute the Q functions not for π but for the optimal strategy:

$$\boxed{Q_n^\pi(s, a) = (1 - \alpha_n) Q_{n-1}^\pi(s, a) + \alpha_n \left(R(s) + \gamma \max_b Q_{n-1}^\pi(s', b)\right)}$$

Conclusion

- With reinforcement learning, we can learn an MDP as we explore it;

Conclusion

- With reinforcement learning, we can learn an MDP as we explore it;
- We can also learn **without a model** (Q-learning);

Conclusion

- With reinforcement learning, we can learn an MDP as we explore it;
- We can also learn **without a model** (Q-learning);
- Models usually decrease the number of trials needed for a given accuracy;

Conclusion

- With reinforcement learning, we can learn an MDP as we explore it;
- We can also learn **without a model** (Q-learning);
- Models usually decrease the number of trials needed for a given accuracy;
- Models for approximating the utility function allow for better **generalisation** and a **smaller memory footprint**;

Conclusion

- With reinforcement learning, we can learn an MDP as we explore it;
- We can also learn **without a model** (Q-learning);
- Models usually decrease the number of trials needed for a given accuracy;
- Models for approximating the utility function allow for better **generalisation** and a **smaller memory footprint**;
- Other techniques exist, in particular **policy search** in which we optimise directly the policy function expressed as a differentiable parameterised function.

Outline

POMDPs

- We add to the MDP formalism:

POMDPs

- We add to the MDP formalism:
  - a finite number of observations $\mathcal{O}$ with a **Markov assumption**: they depend only on the **current** state;

POMDPs

- We add to the MDP formalism:
  - a finite number of observations $\mathcal{O}$ with a **Markov assumption**: they depend only on the **current** state;
  - an observation function $S \to \mathrm{Dist}(\mathcal{O})$ that we also call $\mathcal{O}$ assigning to each state s a distribution on the observations, expressing which is likely to be observed in all states:

POMDPs

- We add to the MDP formalism:
  - a finite number of observations $\mathcal{O}$ with a **Markov assumption**: they depend only on the **current** state;
  - an observation function $S \to \mathrm{Dist}(\mathcal{O})$ that we also call $\mathcal{O}$ assigning to each state s a distribution on the observations, expressing which is likely to be observed in all states:
  - This defines the probabilities $P(o|s)$;

POMDPs

- We add to the MDP formalism:
  - a finite number of observations $\mathcal{O}$ with a **Markov assumption**: they depend only on the **current** state;
  - an observation function $S \to \text{Dist}(\mathcal{O})$ that we also call $\mathcal{O}$ assigning to each state s a distribution on the observations, expressing which is likely to be observed in all states:
  - This defines the probabilities $P(o|s)$;
  - the observation could also depend on the action taken;

POMDPs

- We add to the MDP formalism:
  - a finite number of observations $\mathcal{O}$ with a **Markov assumption**: they depend only on the **current** state;
  - an observation function $S \to \text{Dist}(\mathcal{O})$ that we also call $\mathcal{O}$ assigning to each state s a distribution on the observations, expressing which is likely to be observed in all states:
  - This defines the probabilities $P(o|s)$;
  - the observation could also depend on the action taken;
- The resulting formalism is called **Partially Observable MDP** (POMDP);

POMDPs

- We add to the MDP formalism:
  - a finite number of observations $\mathcal{O}$ with a **Markov assumption**: they depend only on the **current** state;
  - an observation function $S \to \mathrm{Dist}(\mathcal{O})$ that we also call $\mathcal{O}$ assigning to each state s a distribution on the observations, expressing which is likely to be observed in all states:
  - This defines the probabilities $P(o|s)$;
  - the observation could also depend on the action taken;
- The resulting formalism is called **Partially Observable MDP** (POMDP);
- Finding optimal strategies in POMDPs is fairly complex;

POMDPs

- We add to the MDP formalism:
  - a finite number of observations $\mathcal{O}$ with a **Markov assumption**: they depend only on the **current** state;
  - an observation function $S \to \mathrm{Dist}(\mathcal{O})$ that we also call $\mathcal{O}$ assigning to each state s a distribution on the observations, expressing which is likely to be observed in all states:
  - This defines the probabilities $P(o|s)$;
  - the observation could also depend on the action taken;
- The resulting formalism is called **Partially Observable MDP** (POMDP);
- Finding optimal strategies in POMDPs is fairly complex;
- We have seen a first subclass with Partially Observable Hypergraphs (no probabilities);

POMDPs

- We add to the MDP formalism:
  - a finite number of observations $\mathcal{O}$ with a **Markov assumption**: they depend only on the **current** state;
  - an observation function $S \to \text{Dist}(\mathcal{O})$ that we also call $\mathcal{O}$ assigning to each state s a distribution on the observations, expressing which is likely to be observed in all states:
  - This defines the probabilities $P(o|s)$;
  - the observation could also depend on the action taken;
- The resulting formalism is called **Partially Observable MDP** (POMDP);
- Finding optimal strategies in POMDPs is fairly complex;
- We have seen a first subclass with Partially Observable Hypergraphs (no probabilities);
- We now focus on **Hidden Markov Chains** (no choice of actions).

Hidden Markov Chains

- We now assume a POMDP where only one action is possible at each time;

Hidden Markov Chains

- We now assume a POMDP where only one action is possible at each time;
- This is a finite **Markov chain** with partial observation: a **hidden Markov chain** (HMC);

Hidden Markov Chains

- We now assume a POMDP where only one action is possible at each time;
- This is a finite **Markov chain** with partial observation: a **hidden Markov chain** (HMC);
- Recall that we have for each observation $o \in \mathcal{O}$: $P(o|s)$ is the probability of observing o when in state s.

Interesting Problems

We are interested in different problems. Given some observations **up to now**:

- **Observation likelihood**: how likely are the observations made?

Interesting Problems

We are interested in different problems. Given some observations **up to now**:

- **Observation likelihood**: how likely are the observations made?
- **Filtering**: in what states are we likely to be now;

Interesting Problems

We are interested in different problems. Given some observations **up to now**:

- **Observation likelihood**: how likely are the observations made?
- **Filtering**: in what states are we likely to be now;
- **Prediction**: in what states are we likely to be in a few steps from now;

Interesting Problems

We are interested in different problems. Given some observations **up to now**:

- **Observation likelihood**: how likely are the observations made?
- **Filtering**: in what states are we likely to be now;
- **Prediction**: in what states are we likely to be in a few steps from now;
- **Smoothing**: in what states were we likely to be some steps before;

Interesting Problems

We are interested in different problems. Given some observations **up to now**:

- **Observation likelihood**: how likely are the observations made?
- **Filtering**: in what states are we likely to be now;
- **Prediction**: in what states are we likely to be in a few steps from now;
- **Smoothing**: in what states were we likely to be some steps before;
- **Most likely explanation**: What is the most likely sequence of states up to now?

Probability Notations

- A random variable X with values in a finite domain
  e.g. the value obtained after rolling a die.

Probability Notations

- A random variable X with values in a finite domain
  e.g. the value obtained after rolling a die.
- Probability that X has some value x: $P(X = x)$ or just $P(x)$ when non-ambiguous
  Unconditional or prior probability (or just prior).

Probability Notations

- A random variable X with values in a finite domain
  e.g. the value obtained after rolling a die.
- Probability that X has some value x: $P(X = x)$ or just $P(x)$ when non-ambiguous
  Unconditional or prior probability (or just prior).
- Probability that X has value x, knowing that Y has value y: $P(X = x|Y = y)$ or just $P(x|y)$
  Conditional or posterior probability (or just posterior).

Probability Notations

- A random variable X with values in a finite domain
  e.g. the value obtained after rolling a die.
- Probability that X has some value x: $P(X = x)$ or just $P(x)$ when non-ambiguous
  Unconditional or prior probability (or just prior).
- Probability that X has value x, knowing that Y has value y: $P(X = x|Y = y)$ or just $P(x|y)$
  Conditional or posterior probability (or just posterior).
- Probability that X has value x and Y has value y, knowing that Z has value z and W value w: $P(X = x, Y = y|Z = z, W = w)$ or just $P(x, y|z, w)$

Probability Notations: Probability vectors

- Probability vector of X: $\mathbb{P}(X)$
  The vector $(P(X = x_1), P(X = x_2), \ldots, P(x = x_n))$ where $x_1, \ldots, x_n$ are all the possible values of X.

Probability Notations: Probability vectors

- Probability vector of X: $\mathbb{P}(X)$
  The vector $(P(X = x_1), P(X = x_2), \ldots, P(x = x_n))$ where $x_1, \ldots, x_n$ are all the possible values of X.
- Conditional vector of X knowing Y: $\mathbb{P}(X|Y)$
  The matrix containing all pairs $P(X = x|Y = y)$ for all values x of X and y of Y

Probability Notations: Probability vectors

- Probability vector of X: $\mathbb{P}(X)$
  The vector $(P(X = x_1), P(X = x_2), \dots, P(x = x_n))$ where $x_1, \dots, x_n$ are all the possible values of X.
- Conditional vector of X knowing Y: $\mathbb{P}(X|Y)$
  The matrix containing all pairs $P(X = x|Y = y)$ for all values x of X and y of Y
- Joint vector of X and Y: $\mathbb{P}(X, Y)$
  The matrix containing all $P(x, y)$ for all values of x of X and y of Y

Probability Notations: Probability vectors

- Probability vector of X: $\mathbb{P}(X)$
  The vector $(P(X = x_1), P(X = x_2), \ldots, P(x = x_n))$ where $x_1, \ldots, x_n$ are all the possible values of X.
- Conditional vector of X knowing Y: $\mathbb{P}(X|Y)$
  The matrix containing all pairs $P(X = x|Y = y)$ for all values x of X and y of Y
- Joint vector of X and Y: $\mathbb{P}(X, Y)$
  The matrix containing all $P(x, y)$ for all values of x of X and y of Y
- We also denote $P(x)$ by $\mathbb{P}(x)$ for individual values x.

Probability Reminder: Simple Rules

- Product rule:

$$P(x, y) = P(x|y)P(y) \text{ and } \mathbb{P}(X, Y) = \mathbb{P}(X|Y)\mathbb{P}(Y)$$

For $\mathbb{P}$ the right handside is really the matrix containing $P(X = x|Y = y)P(Y = y)$ for all x and y

Probability Reminder: Simple Rules

- Product rule:

$$P(x, y) = P(x|y)P(y) \text{ and } \mathbb{P}(X, Y) = \mathbb{P}(X|Y)\mathbb{P}(Y)$$

For $\mathbb{P}$ the right handside is really the matrix containing $P(X = x|Y = y)P(Y = y)$ for all x and y

- Marginalisation (or summing out):

$$\mathbb{P}(Y) = \sum_{z \in Z} \mathbb{P}(Y, z)$$

$z \in Z$ denotes that z takes all the possible values of Z

Probability Reminder: Simple Rules

- Product rule:

$$P(x, y) = P(x|y)P(y) \text{ and } \mathbb{P}(X, Y) = \mathbb{P}(X|Y)\mathbb{P}(Y)$$

  For $\mathbb{P}$ the right handside is really the matrix containing $P(X = x|Y = y)P(Y = y)$ for all x and y

- Marginalisation (or summing out):

$$\mathbb{P}(Y) = \sum_{z \in Z} \mathbb{P}(Y, z)$$

  $z \in Z$ denotes that z takes all the possible values of Z

- Conditioning (obtained by combining the previous two equations):

$$\mathbb{P}(Y) = \sum_{z \in Z} \mathbb{P}(Y|z)P(z)$$

Probability Reminder: Independence

- Independence: X and Y are independent:

$$\forall y \in Y, \mathbb{P}(X|y) = \mathbb{P}(X) \text{ and } \mathbb{P}(X, y) = \mathbb{P}(X)P(y)$$

Probability Reminder: Independence

- Independence: X and Y are independent:

$$\forall y \in Y, \mathbb{P}(X|y) = \mathbb{P}(X) \text{ and } \mathbb{P}(X, y) = \mathbb{P}(X)P(y)$$

- Conditional Independence: X and Y are independent given z:

$$\forall y \in Y, \mathbb{P}(X, y|z) = \mathbb{P}(X|z)P(y|z)$$

Exercise

I flip a coin. If tails, then I grade all exams with 12 + roll of a fair dice and if heads, by 5 + dice.

1. Are the events “student X has grade 8” and “student Y has grade 6” independent?
2. Are they conditionally independent given the occurrence “the coin flip gave heads”?

Probability Reminder: Independence

- Independence: X and Y are independent:

$$\forall y \in Y, \mathbb{P}(X|y) = \mathbb{P}(X) \text{ and } \mathbb{P}(X, y) = \mathbb{P}(X)P(y)$$

- Conditional Independence: X and Y are independent given z:

$$\forall y \in Y, \mathbb{P}(X, y|z) = \mathbb{P}(X|z)P(y|z)$$

Exercise

I flip a coin. If tails, then I grade all exams with 12 + roll of a fair dice and if heads, by 5 + dice.

1. Are the events “student X has grade 8” and “student Y has grade 6” independent?
2. Are they conditionally independent given the occurrence “the coin flip gave heads”?

- Note that Markov assumptions are conditional independence assumptions.

Probabilistic Inference: Bayes' Rule

- Simple rule (direct from the product rule):

$$\mathbb{P}(Y|X) = \frac{\mathbb{P}(X|Y)\mathbb{P}(Y)}{\mathbb{P}(X)}$$

Probabilistic Inference: Bayes' Rule

- Simple rule (direct from the product rule):

$$\mathbb{P}(Y|X) = \frac{\mathbb{P}(X|Y)\mathbb{P}(Y)}{\mathbb{P}(X)}$$

- Conditionalised rule

$$\mathbb{P}(Y|X,e) = \frac{\mathbb{P}(X|Y,e)\mathbb{P}(Y|e)}{\mathbb{P}(X|e)}$$

Probabilistic Inference: Bayes' Rule

- Bayes' rule allows to do inference:

$$P(\text{hypothesis}|\text{evidence}) = \frac{P(\text{evidence}|\text{hypothesis})P(\text{hypothesis})}{P(\text{evidence})}$$

- Note that to compute this, we must either know $P(\text{evidence})$, or $P(\text{evidence}|\neg\text{hypothesis})$ to compute it by conditioning.

Probabilistic Inference: Bayes' Rule

- Bayes' rule allows to do inference:

$$P(\text{hypothesis}|\text{evidence}) = \frac{P(\text{evidence}|\text{hypothesis})P(\text{hypothesis})}{P(\text{evidence})}$$

- Note that to compute this, we must either know $P(\text{evidence})$, or $P(\text{evidence}|\neg\text{hypothesis})$ to compute it by conditioning.

Exercise

Pigritia (Pg) is a fairly common affliction that occurs with a 1/1000 rate. A person afflicted by Pg, usually also exhibits a *palmar hypertrichosis* syndrom (PHS).

- The rate of false positive is 5% (the patient may also have *lycanthropia* in 5% of the cases);
- The rate of false negative is 1% (rarely, a sick person who always sleeps with clenched fists may prevent the development of the hypertrichosis).

What is the probability that a person with PHS also has Pg?

Back to HMCs: Example

Attention estimation

- Students are attending an IA class with laptops open before them;
- They can be in two states: listening (l) and web surfing (w);
- Suppose all students are initially listening;
- The professor can witness three possible events: smiling (s), looking at the laptop screen (c), and looking at the professor or blackboard (p);
- When listening at some instant, students tend to continue listening at the next at 70% chance;
- When surfing at some instant, students tend to continue surfing at the next at 80% chance;
- When listening, students smile at 10% (jokes of the professor), look at their screen at 30% for additional info, or look up at 60%;
- When surfing, students smile at 30% (lolcats), look at their screen at 60% (social networking notification), or look up at 10% (am I busted?);

Computing the state-observation joint probabilities

- We denote by X_t the random variable giving the state at time t (after t steps);

Computing the state-observation joint probabilities

- We denote by X_t the random variable giving the state at time t (after t steps);
- Given some observations $o_1, \ldots, o_t$ up to time step t, we want to compute the **joint probabilities** of (hidden) states with those observations: $\mathbb{P}(X_t, o_1, \ldots, o_t)$
  We note $\mathbb{P}(X_t, o_{1:t})$ for short

Computing the state-observation joint probabilities

- We denote by X_t the random variable giving the state at time t (after t steps);
- Given some observations $o_1, \dots, o_t$ up to time step t, we want to compute the **joint probabilities** of (hidden) states with those observations: $\mathbb{P}(X_t, o_1, \dots, o_t)$
  We note $\mathbb{P}(X_t, o_{1:t})$ for short
- We could enumerate all the corresponding sequences and sum the probabilities (exponential number);

Computing the state-observation joint probabilities

- We denote by X_t the random variable giving the state at time t (after t steps);
- Given some observations $o_1, \ldots, o_t$ up to time step t, we want to compute the **joint probabilities** of (hidden) states with those observations: $\mathbb{P}(X_t, o_1, \ldots, o_t)$
  We note $\mathbb{P}(X_t, o_{1:t})$ for short
- We could enumerate all the corresponding sequences and sum the probabilities (exponential number);
- We can do better with an iterative algorithm called **forward algorithm**.

The forward algorithm

- We compute $\mathbb{P}(X_t, o_{1:t})$ by summing out X_{t-1} from $\mathbb{P}(X_t, X_{t-1}, o_{1:t})$:

$$\mathbb{P}(X_t, o_{1:t}) = \sum_{x_{t-1}} \mathbb{P}(X_t, x_{t-1}, o_{1:t})$$

The forward algorithm

- We compute $\mathbb{P}(X_t, o_{1:t})$ by summing out X_{t-1} from $\mathbb{P}(X_t, X_{t-1}, o_{1:t})$:

$$\mathbb{P}(X_t, o_{1:t}) = \sum_{x_{t-1}} \mathbb{P}(X_t, x_{t-1}, o_{1:t})$$

- By the product rule:

$$\mathbb{P}(X_t, o_{1:t}) = \sum_{x_{t-1}} \mathbb{P}(o_t | X_t, x_{t-1}, o_{1:t-1}) \mathbb{P}(X_t | x_{t-1}, o_{1:t-1}) P(x_{t-1}, o_{1:t-1})$$

The forward algorithm

- We compute $\mathbb{P}(X_t, o_{1:t})$ by summing out X_{t-1} from $\mathbb{P}(X_t, X_{t-1}, o_{1:t})$:

$$\mathbb{P}(X_t, o_{1:t}) = \sum_{x_{t-1}} \mathbb{P}(X_t, x_{t-1}, o_{1:t})$$

- By the product rule:

$$\mathbb{P}(X_t, o_{1:t}) = \sum_{x_{t-1}} \mathbb{P}(o_t | X_t, x_{t-1}, o_{1:t-1}) \mathbb{P}(X_t | x_{t-1}, o_{1:t-1}) P(x_{t-1}, o_{1:t-1})$$

- By the transition and sensor Markov assumptions:

$$\mathbb{P}(X_t, o_{1:t}) = \mathbb{P}(o_t | X_t) \sum_{x_{t-1}} \mathbb{P}(X_t | x_{t-1}) P(x_{t-1}, o_{1:t-1})$$

The forward algorithm

- We have expressed $\mathbb{P}(X_t, o_{1:t})$ in function of $\mathbb{P}(X_t, o_{1:t-1})$ and o_t;

The forward algorithm

- We have expressed $\mathbb{P}(X_t, o_{1:t})$ in function of $\mathbb{P}(X_t, o_{1:t-1})$ and o_t;
- We note:

$$\mathbb{P}(X_t, o_{1:t}) = \textsc{Forward}(\mathbb{P}(X_{t-1}, o_{1:t-1}), o_t)$$

The forward algorithm

- We have expressed $\mathbb{P}(X_t, o_{1:t})$ in function of $\mathbb{P}(X_t, o_{1:t-1})$ and o_t;
- We note:

$$\mathbb{P}(X_t, o_{1:t}) = \textsc{Forward}(\mathbb{P}(X_{t-1}, o_{1:t-1}), o_t)$$

- This gives an iterative algorithm initialised with $\mathbb{P}(X_0)$, the initial distribution on hidden states.

The forward algorithm: Matrix expression

- The transitions probabilities can be written as matrix T such that $T_{ij} = P(X_t = j|X_{t-1} = i)$;
  i is the row, j the column

The forward algorithm: Matrix expression

- The transitions probabilities can be written as matrix T such that $T_{ij} = P(X_t = j | X_{t-1} = i)$;
  i is the row, j the column
- For each observation, at time t we know which observation o_t was made;

The forward algorithm: Matrix expression

- The transitions probabilities can be written as matrix T such that $T_{ij} = P(X_t = j | X_{t-1} = i)$;
  i is the row, j the column
- For each observation, at time t we know which observation o_t was made;
- We write the probabilities of $P(o_t | x_i)$ on the i-th diagonal entry of a diagonal matrix O_t;

The forward algorithm: Matrix expression

- The transitions probabilities can be written as matrix T such that $T_{ij} = P(X_t = j | X_{t-1} = i)$;
  i is the row, j the column
- For each observation, at time t we know which observation o_t was made;
- We write the probabilities of $P(o_t | x_i)$ on the i-th diagonal entry of a diagonal matrix O_t;
- Let $\ell_{1:t} = P(X_t, o_{1:t})$. Then:

$$\boxed{\ell_{1:0} = \mathbb{P}(X_0) \text{ and } \ell_{1:t} = O_t T^{\mathsf{T}} \ell_{1:t-1}}$$

The forward algorithm: Exercise

Attention estimation

- Students are attending an IA class with laptops open before them;
- They can be in two states: listening (l) and web surfing (w);
- Suppose all students are initially listening;
- The professor can witness three possible events: smiling (s), looking at the laptop screen (c), and looking at the professor or blackboard (p);
- When listening at some instant, students tend to continue listening at the next at 70% chance;
- When surfing at some instant, students tend to continue surfing at the next at 80% chance;
- When listening, students smile at 10% (jokes of the professor), look at their screen at 30% for additional info, or look up at 60%;
- When surfing, students smile at 30% (lolcats), look at their screen at 60% (social networking notification), or look up at 10% (am I busted?);

Exercise

Write the matrices and compute the probabilities: $P(l_3, s_1, c_2, p_3)$ and $P(w_3, s_1, c_2, p_3)$.

Observation likelihood

- We can compute $P(o_{1:t})$ by summing X_t out of $\mathbb{P}(X_t, o_{1:t})$:

$$P(o_{1:t}) = \sum_{x_t} P(x_t, o_{1:t})$$

Observation likelihood

- We can compute $P(o_{1:t})$ by summing X_t out of $\mathbb{P}(X_t, o_{1:t})$:

$$P(o_{1:t}) = \sum_{x_t} P(x_t, o_{1:t})$$

- Matricially:
  $\mathbf{1}$ is the vector containing only ones

$$\boxed{P(o_{1:t}) = \mathbf{1}^\mathsf{T} \ell_{1:t}}$$

Observation likelihood

- We can compute $P(o_{1:t})$ by summing X_t out of $\mathbb{P}(X_t, o_{1:t})$:

$$P(o_{1:t}) = \sum_{x_t} P(x_t, o_{1:t})$$

- Matricially:
  $\mathbf{1}$ is the vector containing only ones

$$\boxed{P(o_{1:t}) = \mathbf{1}^\top \ell_{1:t}}$$

Attention estimation

What are the likelihoods of the sequences (1) s, c, p, (2) p, p, p and (3) c, c, c?

Filtering

- We want to compute $\mathbb{P}(X_t|o_{1:t})$, for $t > 1$;

Filtering

- We want to compute $\mathbb{P}(X_t|o_{1:t})$, for $t > 1$;
- By the product rule:
$$\mathbb{P}(X_t|o_{1:t}) = \frac{\mathbb{P}(X_t, o_{1:t})}{P(o_{1:t})}$$

Filtering

- We want to compute $\mathbb{P}(X_t|o_{1:t})$, for $t > 1$;
- By the product rule:

$$\mathbb{P}(X_t|o_{1:t}) = \frac{\mathbb{P}(X_t, o_{1:t})}{P(o_{1:t})}$$

- That is:

$$\boxed{\mathbb{P}(X_t|o_{1:t}) = \frac{1}{\mathbf{1}^\mathsf{T}\ell_{1:t}}\ell_{1:t}}$$

Filtering

- We want to compute $\mathbb{P}(X_t|o_{1:t})$, for $t > 1$;
- By the product rule:

$$\mathbb{P}(X_t|o_{1:t}) = \frac{\mathbb{P}(X_t, o_{1:t})}{P(o_{1:t})}$$

- That is:

$$\boxed{\mathbb{P}(X_t|o_{1:t}) = \frac{1}{\mathbf{1}^\mathsf{T}\ell_{1:t}}\ell_{1:t}}$$

- We just need to **normalise** the vector $\ell_{1:t}$ so that it sums to 1.

Filtering

Exercise: Attention estimation

Compute for the previous problem the distribution formed by $P(l_3|s_1, c_2, p_3)$ and $P(w_3|s_1, c_2, p_3)$.

Filtering

Exercise: Attention estimation

Compute for the previous problem the distribution formed by $P(l_3|s_1, c_2, p_3)$ and $P(w_3|s_1, c_2, p_3)$.

Exercise: Localisation

A robot moves on a 5-tiles **circular** track with tiles painted black (b) or white (w). The robot only has a color sensor and is initially at an unknown position (with uniform probability). The robot moves at each step left with probability 0.3 and right, with probability 0.7:

Find the most likely position of the robot after observations w_1, b_2, w_3.

Prediction

- $\mathbb{P}(X_{t+1}|o_{1:t})$ is a one-step **prediction**:

Prediction

- $\mathbb{P}(X_{t+1}|o_{1:t})$ is a one-step **prediction**:
- By summing out X_t in $\mathbb{P}(X_{t+1}, X_t) = \mathbb{P}(X_{t+1}|X_t)\mathbb{P}(X_t)$:

$$\mathbb{P}(X_{t+1}|o_{1:t}) = \sum_{x_t} \mathbb{P}(X_{t+1}|x_t, o_{1:t})P(x_t|o_{1:t})$$

Prediction

- $\mathbb{P}(X_{t+1}|o_{1:t})$ is a one-step **prediction**:
- By summing out X_t in $\mathbb{P}(X_{t+1}, X_t) = \mathbb{P}(X_{t+1}|X_t)\mathbb{P}(X_t)$:

$$\mathbb{P}(X_{t+1}|o_{1:t}) = \sum_{x_t} \mathbb{P}(X_{t+1}|x_t, o_{1:t})P(x_t|o_{1:t})$$

- With the Markov assumption on transitions:

$$\mathbb{P}(X_{t+1}|o_{1:t}) = \sum_{x_{t-1}} \mathbb{P}(X_{t+1}|x_t)P(x_t|o_{1:t})$$

Prediction

- $\mathbb{P}(X_{t+1}|o_{1:t})$ is a one-step **prediction**:
- By summing out X_t in $\mathbb{P}(X_{t+1}, X_t) = \mathbb{P}(X_{t+1}|X_t)\mathbb{P}(X_t)$:

$$\mathbb{P}(X_{t+1}|o_{1:t}) = \sum_{x_t} \mathbb{P}(X_{t+1}|x_t, o_{1:t}) P(x_t|o_{1:t})$$

- With the Markov assumption on transitions:

$$\mathbb{P}(X_{t+1}|o_{1:t}) = \sum_{x_{t-1}} \mathbb{P}(X_{t+1}|x_t) P(x_t|o_{1:t})$$

- Matricially:

$$\mathbb{P}(X_{t+1}|o_{1:t}) = T^{\mathsf{T}} \mathbf{f}_{1:t}$$

Prediction

- $\mathbb{P}(X_{t+1}|o_{1:t})$ is a one-step **prediction**:
- By summing out X_t in $\mathbb{P}(X_{t+1}, X_t) = \mathbb{P}(X_{t+1}|X_t)\mathbb{P}(X_t)$:

$$\mathbb{P}(X_{t+1}|o_{1:t}) = \sum_{x_t} \mathbb{P}(X_{t+1}|x_t, o_{1:t})P(x_t|o_{1:t})$$

- With the Markov assumption on transitions:

$$\mathbb{P}(X_{t+1}|o_{1:t}) = \sum_{x_{t-1}} \mathbb{P}(X_{t+1}|x_t)P(x_t|o_{1:t})$$

- Matricially:

$$\mathbb{P}(X_{t+1}|o_{1:t}) = T^{\mathsf{T}}\mathbf{f}_{1:t}$$

- We could compute similarly n-step predictions.

$$\boxed{\mathbb{P}(X_{t+n}|o_{1:t}) = (T^{\mathsf{T}})^n\mathbf{f}_{1:t}}$$

Prediction: Exercise

Exercise: Attention estimation

After observing s_1, c_2, p_3 what is the probability that a student will be listening at time 4? What about times 13, 1003, and 1000003 ?

Smoothing

- We want to compute $\mathbb{P}(X_k|o_{1:t})$ with $k < t$;

Smoothing

- We want to compute $\mathbb{P}(X_k | o_{1:t})$ with $k < t$;
- Given some new evidence we refine a previous estimation.

Smoothing

- We want to compute $\mathbb{P}(X_k | o_{1:t})$ with $k < t$;
- Given some new evidence we refine a previous estimation.
- We try to make the "naive" estimation appear:

$$\mathbb{P}(X_k | o_{1:t}) = \mathbb{P}(X_k | o_{1:k}, o_{k+1:t})$$

Smoothing

- We want to compute $\mathbb{P}(X_k|o_{1:t})$ with $k < t$;
- Given some new evidence we refine a previous estimation.
- We try to make the "naive" estimation appear:

$$\mathbb{P}(X_k|o_{1:t}) = \mathbb{P}(X_k|o_{1:k}, o_{k+1:t})$$

- By Bayes' rule:

$$\mathbb{P}(X_k|o_{1:t}) = \frac{\mathbb{P}(X_k|o_{1:k})\mathbb{P}(o_{k+1:t}|X_k, o_{1:k})}{P(o_{k+1:t}|o_{1:k})}$$

Smoothing

- We want to compute $\mathbb{P}(X_k|o_{1:t})$ with $k < t$;
- Given some new evidence we refine a previous estimation.
- We try to make the "naive" estimation appear:

$$\mathbb{P}(X_k|o_{1:t}) = \mathbb{P}(X_k|o_{1:k}, o_{k+1:t})$$

- By Bayes' rule:

$$\mathbb{P}(X_k|o_{1:t}) = \frac{\mathbb{P}(X_k|o_{1:k})\mathbb{P}(o_{k+1:t}|X_k, o_{1:k})}{P(o_{k+1:t}|o_{1:k})}$$

- Given X_k, $o_{1:k}$ and $o_{k+1:t}$ are independent (sensor assumption):

$$\mathbb{P}(X_k|o_{1:t}) = \frac{1}{P(o_{k+1:t}|o_{1:k})}\mathbb{P}(X_k, o_{1:k})\mathbb{P}(o_{k+1:t}|X_k)$$

Smoothing

- We want to compute $\mathbb{P}(X_k|o_{1:t})$ with $k < t$;
- Given some new evidence we refine a previous estimation.
- We try to make the "naive" estimation appear:

$$\mathbb{P}(X_k|o_{1:t}) = \mathbb{P}(X_k|o_{1:k}, o_{k+1:t})$$

- By Bayes' rule:

$$\mathbb{P}(X_k|o_{1:t}) = \frac{\mathbb{P}(X_k|o_{1:k})\mathbb{P}(o_{k+1:t}|X_k, o_{1:k})}{P(o_{k+1:t}|o_{1:k})}$$

- Given X_k, $o_{1:k}$ and $o_{k+1:t}$ are independent (sensor assumption):

$$\mathbb{P}(X_k|o_{1:t}) = \frac{1}{P(o_{k+1:t}|o_{1:k})}\mathbb{P}(X_k, o_{1:k})\mathbb{P}(o_{k+1:t}|X_k)$$

- And finally:

$$\mathbb{P}(X_k|o_{1:t}) = \frac{1}{P(o_{1:t})}\mathbf{f}_{1:k}\mathbb{P}(o_{k+1:t}|X_k)$$

Smoothing

- We need to compute $\mathbb{P}(o_{k+1:t}|X_k)$;

Smoothing

- We need to compute $\mathbb{P}(o_{k+1:t}|X_k)$;
- We will do it iteratively from $\mathbb{P}(o_{k+2:t}|X_k)$ and o_{k+1};

Smoothing

- We need to compute $\mathbb{P}(o_{k+1:t}|X_k)$;
- We will do it iteratively from $\mathbb{P}(o_{k+2:t}|X_k)$ and o_{k+1};
- By conditioning on X_{k+1}:

$$\mathbb{P}(o_{k+1:t}|X_k) = \sum_{x_{k+1}} \mathbb{P}(o_{k+1:t}|X_k, x_{k+1})\mathbb{P}(x_{k+1}|X_k)$$

Smoothing

- We need to compute $\mathbb{P}(o_{k+1:t}|X_k)$;
- We will do it iteratively from $\mathbb{P}(o_{k+2:t}|X_k)$ and o_{k+1};
- By conditioning on X_{k+1}:

$$\mathbb{P}(o_{k+1:t}|X_k) = \sum_{x_{k+1}} \mathbb{P}(o_{k+1:t}|X_k, x_{k+1})\mathbb{P}(x_{k+1}|X_k)$$

- Given x_{k+1}, $o_{k+1:t}$ does not depend on X_k:

$$\mathbb{P}(o_{k+1:t}|X_k) = \sum_{x_{k+1}} P(o_{k+1}, o_{k+2:t}|x_{k+1})\mathbb{P}(x_{k+1}|X_k)$$

Smoothing

- We need to compute $\mathbb{P}(o_{k+1:t}|X_k)$;
- We will do it iteratively from $\mathbb{P}(o_{k+2:t}|X_k)$ and o_{k+1};
- By conditioning on X_{k+1}:

$$\mathbb{P}(o_{k+1:t}|X_k) = \sum_{x_{k+1}} \mathbb{P}(o_{k+1:t}|X_k, x_{k+1})\mathbb{P}(x_{k+1}|X_k)$$

- Given x_{k+1}, $o_{k+1:t}$ does not depend on X_k:

$$\mathbb{P}(o_{k+1:t}|X_k) = \sum_{x_{k+1}} P(o_{k+1}, o_{k+2:t}|x_{k+1})\mathbb{P}(x_{k+1}|X_k)$$

- By the product rule:

$$\mathbb{P}(o_{k+1:t}|X_k) = \sum_{x_{k+1}} P(o_{k+1}|o_{k+2:t}, x_{k+1})P(o_{k+2:t}|x_{k+1})\mathbb{P}(x_{k+1}|X_k)$$

Smoothing

- We need to compute $\mathbb{P}(o_{k+1:t}|X_k)$;
- We will do it iteratively from $\mathbb{P}(o_{k+2:t}|X_k)$ and o_{k+1};
- By conditioning on X_{k+1}:

$$\mathbb{P}(o_{k+1:t}|X_k) = \sum_{x_{k+1}} \mathbb{P}(o_{k+1:t}|X_k, x_{k+1})\mathbb{P}(x_{k+1}|X_k)$$

- Given x_{k+1}, $o_{k+1:t}$ does not depend on X_k:

$$\mathbb{P}(o_{k+1:t}|X_k) = \sum_{x_{k+1}} P(o_{k+1}, o_{k+2:t}|x_{k+1})\mathbb{P}(x_{k+1}|X_k)$$

- By the product rule:

$$\mathbb{P}(o_{k+1:t}|X_k) = \sum_{x_{k+1}} P(o_{k+1}|o_{k+2:t}, x_{k+1})P(o_{k+2:t}|x_{k+1})\mathbb{P}(x_{k+1}|X_k)$$

- Given x_{k+1}, o_{k+1} does not depend on $o_{k+2:t}$:

$$\mathbb{P}(o_{k+1:t}|X_k) = \sum_{x_{k+1}} P(o_{k+1}|x_{k+1})P(o_{k+2:t}|x_{k+1})\mathbb{P}(x_{k+1}|X_k)$$

Smoothing: Forward-Backward algorithm

- We have $\mathbb{P}(o_{k+1:t}|X_k)$ in function of $\mathbb{P}(o_{k+2:t}|X_k)$ and o_{k+1};

$$\mathbb{P}(o_{k+1:t}|X_k) = \sum_{x_{k+1}} P(o_{k+1}|x_{k+1})P(o_{k+2:t}|x_{k+1})\mathbb{P}(x_{k+1}|X_k)$$

Smoothing: Forward-Backward algorithm

- We have $\mathbb{P}(o_{k+1:t}|X_k)$ in function of $\mathbb{P}(o_{k+2:t}|X_k)$ and o_{k+1};

$$\mathbb{P}(o_{k+1:t}|X_k) = \sum_{x_{k+1}} P(o_{k+1}|x_{k+1})P(o_{k+2:t}|x_{k+1})\mathbb{P}(x_{k+1}|X_k)$$

- We write, by noting for $i > k$, $\mathbf{b}_{i:t} = \mathbb{P}(o_{i:t}|X_k)$:

$$\mathbf{b}_{i:t} = \textsc{Backward}(\mathbf{b}_{i+1:t}, o_i)$$

Smoothing: Forward-Backward algorithm

- We have $\mathbb{P}(o_{k+1:t}|X_k)$ in function of $\mathbb{P}(o_{k+2:t}|X_k)$ and o_{k+1};

$$\mathbb{P}(o_{k+1:t}|X_k) = \sum_{x_{k+1}} P(o_{k+1}|x_{k+1})P(o_{k+2:t}|x_{k+1})\mathbb{P}(x_{k+1}|X_k)$$

- We write, by noting for $i > k$, $\mathbf{b}_{i:t} = \mathbb{P}(o_{i:t}|X_k)$:

$$\mathbf{b}_{i:t} = \textsc{Backward}(\mathbf{b}_{i+1:t}, o_i)$$

- We initialise with $\mathbf{b}_{t+1:t} = \mathbf{1}$, the vector containing only ones
  The probability of observing an empty sequence is one in each state

Smoothing: Forward-Backward algorithm

- We have $\mathbb{P}(o_{k+1:t}|X_k)$ in function of $\mathbb{P}(o_{k+2:t}|X_k)$ and o_{k+1};

$$\mathbb{P}(o_{k+1:t}|X_k) = \sum_{x_{k+1}} P(o_{k+1}|x_{k+1})P(o_{k+2:t}|x_{k+1})\mathbb{P}(x_{k+1}|X_k)$$

- We write, by noting for $i > k$, $\mathbf{b}_{i:t} = \mathbb{P}(o_{i:t}|X_k)$:

$$\mathbf{b}_{i:t} = \textrm{B}\textsc{ackward}(\mathbf{b}_{i+1:t}, o_i)$$

- We initialise with $\mathbf{b}_{t+1:t} = \mathbf{1}$, the vector containing only ones
  The probability of observing an empty sequence is one in each state
- Matricially:

$$\boxed{\mathbf{b}_{i:t} = TO_i\mathbf{b}_{i+1:t}}$$

Smoothing

- Recall that for smoothing we have:

$$\mathbb{P}(X_k|o_{1:t}) = \frac{1}{P(o_{k+1:t}|o_{1:k})}\mathbb{P}(X_k|o_{1:k})\mathbb{P}(o_{k+1:t}|X_k)$$

- Then, with $\circ$ the Hadamard product, *i. e.*, pointwise:

$$\mathbb{P}(X_k|o_{1:t}) = \frac{1}{P(o_{1:t})}\mathbf{f}_{1:k} \circ \mathbf{b}_{k+1:t}$$

- And since $\mathbb{P}(X_k|o_{1:t})$ represents a distribution, we can just normalize the vector as before, and actually use $\ell_{1:k}$ instead of $\mathbf{f}_{1:k}$:

$$\boxed{\mathbb{P}(X_k|o_{1:t}) = \frac{1}{\mathbf{1}^\mathsf{T}\mathbf{s}_{1:k:t}}\mathbf{s}_{1:k:t} \text{ with } \mathbf{s}_{1:k:t} = \ell_{1:k} \circ \mathbf{b}_{k+1:t}}$$

Smoothing

- Recall that for smoothing we have:

$$\mathbb{P}(X_k|o_{1:t}) = \frac{1}{P(o_{k+1:t}|o_{1:k})}\mathbb{P}(X_k|o_{1:k})\mathbb{P}(o_{k+1:t}|X_k)$$

- Then, with $\circ$ the Hadamard product, *i. e.*, pointwise:

$$\mathbb{P}(X_k|o_{1:t}) = \frac{1}{P(o_{1:t})}\mathbf{f}_{1:k} \circ \mathbf{b}_{k+1:t}$$

- And since $\mathbb{P}(X_k|o_{1:t})$ represents a distribution, we can just normalize the vector as before, and actually use $\ell_{1:k}$ instead of $\mathbf{f}_{1:k}$:

$$\boxed{\mathbb{P}(X_k|o_{1:t}) = \frac{1}{\mathbf{1}^\mathsf{T}\mathbf{s}_{1:k:t}}\mathbf{s}_{1:k:t} \text{ with } \mathbf{s}_{1:k:t} = \ell_{1:k} \circ \mathbf{b}_{k+1:t}}$$

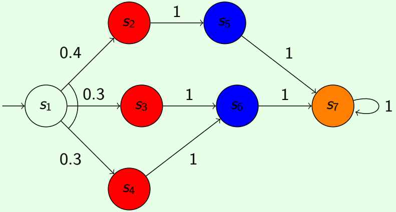
Exercise: Attention estimation & Localisation

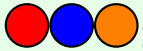
1. What is the smoothed estimate of a student listening at time 3 when observing s, c, p, s, c, p?
2. From where has the robot started in the B/W circular track, knowing we have observed w, b, w?

Smoothing: Exercise

Exercise: Sequence

Consider the following Hidden MC, in which the initial distribution is $[1, 0, 0, 0, 0, 0, 0]$:



Given the observation sequence :

1. What are the most likely states at times $3, 2$ and 1?
2. What is the most likely sequence producing the observation?

Most Likely Explanation – Decoding

- We want to compute the most probable sequence of hidden states given the observations:

Most Likely Explanation – Decoding

- We want to compute the most probable sequence of hidden states given the observations:
- That is the values of $x_1, \ldots, x_t$ that realise the following max:

$$\max_{x_1, \ldots, x_t} P(x_{1:t} | o_{1:t})$$

Most Likely Explanation – Decoding

- We want to compute the most probable sequence of hidden states given the observations:
- That is the values of $x_1, \ldots, x_t$ that realise the following max:

$$\max_{x_1,\ldots,x_t} P(x_{1:t} | o_{1:t})$$

- By the Markov property on transitions, if we fix x_t,the most probable sequence is made of:

Most Likely Explanation – Decoding

- We want to compute the most probable sequence of hidden states given the observations:
- That is the values of $x_1, \dots, x_t$ that realise the following max:

$$\max_{x_1,\dots,x_t} P(x_{1:t}|o_{1:t})$$

- By the Markov property on transitions, if we fix x_t,the most probable sequence is made of:
  1. a single transition from x_{t-1} to x_t;

Most Likely Explanation – Decoding

- We want to compute the most probable sequence of hidden states given the observations:
- That is the values of $x_1, \ldots, x_t$ that realise the following max:

$$\max_{x_1,\ldots,x_t} P(x_{1:t}|o_{1:t})$$

- By the Markov property on transitions, if we fix x_t,the most probable sequence is made of:
  1. a single transition from x_{t-1} to x_t;
  2. the most likely path to x_{t-1} given $o_{1:t-1}$
     o_t does not matter because x_t is fixed

Most Likely Explanation – Decoding

- We want to compute the most probable sequence of hidden states given the observations:
- That is the values of $x_1, \ldots, x_t$ that realise the following max:

$$\max_{x_1,\ldots,x_t} P(x_{1:t} | o_{1:t})$$

- By the Markov property on transitions, if we fix x_t,the most probable sequence is made of:
  1. a single transition from x_{t-1} to x_t;
  2. the most likely path to x_{t-1} given $o_{1:t-1}$
     o_t does not matter because x_t is fixed
- We can thus find an iterative algorithm to compute this.

Most Likely Explanation – Decoding

- We want to compute the most probable sequence of hidden states given the observations:
- That is the values of $x_1, \ldots, x_t$ that realise the following max:

$$\max_{x_1,\ldots,x_t} P(x_{1:t}|o_{1:t})$$

- By the Markov property on transitions, if we fix x_t,the most probable sequence is made of:
  1. a single transition from x_{t-1} to x_t;
  2. the most likely path to x_{t-1} given $o_{1:t-1}$
     o_t does not matter because x_t is fixed
- We can thus find an iterative algorithm to compute this.
- As before note that $P(x_{1:t}|o_{1:t})$ is just the normalisation of $P(x_{1:t}, o_{1:t})$ and the positive normalisation coefficient $P(o_{1:t})$ does not change the max.

Viterbi algorithm

- Noting $\mathbf{m}_{1:t+1} = \max_{x_{1:t}} \mathbb{P}(x_{1:t}, X_{t+1}, o_{1:t+1})$ and by the chain rule

$$\mathbf{m}_{1:t+1} = \max_{x_{1:t}} \mathbb{P}(o_{t+1}|x_{1:t}, X_{t+1}, o_{1:t})\mathbb{P}(X_{t+1}|x_{1:t}, o_{1:t})P(x_{1:t}, o_{1:t})$$

Viterbi algorithm

- Noting $\mathbf{m}_{1:t+1} = \max_{x_{1:t}} \mathbb{P}(x_{1:t}, X_{t+1}, o_{1:t+1})$ and by the chain rule

$$\mathbf{m}_{1:t+1} = \max_{x_{1:t}} \mathbb{P}(o_{t+1}|x_{1:t}, X_{t+1}, o_{1:t})\mathbb{P}(X_{t+1}|x_{1:t}, o_{1:t})P(x_{1:t}, o_{1:t})$$

- By the transition and sensor Markov assumptions:

$$\mathbf{m}_{1:t+1} = \max_{x_{1:t}} \mathbb{P}(o_{t+1}|X_{t+1})\mathbb{P}(X_{t+1}|x_t)P(x_{1:t}, o_{1:t})$$

Viterbi algorithm

- Noting $\mathbf{m}_{1:t+1} = \max_{x_{1:t}} \mathbb{P}(x_{1:t}, X_{t+1}, o_{1:t+1})$ and by the chain rule

$$\mathbf{m}_{1:t+1} = \max_{x_{1:t}} \mathbb{P}(o_{t+1}|x_{1:t}, X_{t+1}, o_{1:t})\mathbb{P}(X_{t+1}|x_{1:t}, o_{1:t})P(x_{1:t}, o_{1:t})$$

- By the transition and sensor Markov assumptions:

$$\mathbf{m}_{1:t+1} = \max_{x_{1:t}} \mathbb{P}(o_{t+1}|X_{t+1})\mathbb{P}(X_{t+1}|x_t)P(x_{1:t}, o_{1:t})$$

- Separating the max:

$$\mathbf{m}_{1:t+1} = \mathbb{P}(o_{t+1}|X_{t+1}) \max_{x_t} \left(\mathbb{P}(X_{t+1}|x_t) \max_{x_{1:t-1}} P(x_{1:t-1}, x_t, o_{1:t})\right)$$

Viterbi algorithm

- Noting $\mathbf{m}_{1:t+1} = \max_{x_{1:t}} \mathbb{P}(x_{1:t}, X_{t+1}, o_{1:t+1})$ and by the chain rule

$$\mathbf{m}_{1:t+1} = \max_{x_{1:t}} \mathbb{P}(o_{t+1}|x_{1:t}, X_{t+1}, o_{1:t})\mathbb{P}(X_{t+1}|x_{1:t}, o_{1:t})P(x_{1:t}, o_{1:t})$$

- By the transition and sensor Markov assumptions:

$$\mathbf{m}_{1:t+1} = \max_{x_{1:t}} \mathbb{P}(o_{t+1}|X_{t+1})\mathbb{P}(X_{t+1}|x_t)P(x_{1:t}, o_{1:t})$$

- Separating the max:

$$\mathbf{m}_{1:t+1} = \mathbb{P}(o_{t+1}|X_{t+1}) \max_{x_t} \left(\mathbb{P}(X_{t+1}|x_t) \max_{x_{1:t-1}} P(x_{1:t-1}, x_t, o_{1:t})\right)$$

- The value that we are looking for is the biggest coefficient in $\mathbf{m}_{1:t+1}$;

Viterbi algorithm

- Noting $\mathbf{m}_{1:t+1} = \max_{x_{1:t}} \mathbb{P}(x_{1:t}, X_{t+1}, o_{1:t+1})$ and by the chain rule

$$\mathbf{m}_{1:t+1} = \max_{x_{1:t}} \mathbb{P}(o_{t+1}|x_{1:t}, X_{t+1}, o_{1:t})\mathbb{P}(X_{t+1}|x_{1:t}, o_{1:t})P(x_{1:t}, o_{1:t})$$

- By the transition and sensor Markov assumptions:

$$\mathbf{m}_{1:t+1} = \max_{x_{1:t}} \mathbb{P}(o_{t+1}|X_{t+1})\mathbb{P}(X_{t+1}|x_t)P(x_{1:t}, o_{1:t})$$

- Separating the max:

$$\mathbf{m}_{1:t+1} = \mathbb{P}(o_{t+1}|X_{t+1}) \max_{x_t} \left(\mathbb{P}(X_{t+1}|x_t) \max_{x_{1:t-1}} P(x_{1:t-1}, x_t, o_{1:t})\right)$$

- The value that we are looking for is the biggest coefficient in $\mathbf{m}_{1:t+1}$;
- The corresponding transition is from the value of x_t that realises the max in $\mathbf{m}_{1:t+1}$ to that one.

Viterbi algorithm

- We have $\mathbf{m}_{1:t+1}$ in function of $\mathbf{m}_{1:t}$ and o_{t+1}:

$$\mathbf{m}_{1:t+1} = \textsc{Viterbi}(\mathbf{m}_{1:t+1}, o_{t+1})$$

Viterbi algorithm

- We have $\mathbf{m}_{1:t+1}$ in function of $\mathbf{m}_{1:t}$ and o_{t+1}:

$$\mathbf{m}_{1:t+1} = \textsc{Viterbi}(\mathbf{m}_{1:t+1}, o_{t+1})$$

- Matricially (probably does a lot of useless computations):
  $\textsc{eye}$ gives a diagonal matrix from a vector

$$\mathbf{m}_{1:t+1} = \max_{\text{by line}} O_{t+1} T^{\mathsf{T}} \textsc{eye}(\mathbf{m}_{\mathbf{1:t}})$$

Viterbi algorithm

- We have $\mathbf{m}_{1:t+1}$ in function of $\mathbf{m}_{1:t}$ and o_{t+1}:

$$\mathbf{m}_{1:t+1} = \textsc{Viterbi}(\mathbf{m}_{1:t+1}, o_{t+1})$$

- Matricially (probably does a lot of useless computations):
  EYE gives a diagonal matrix from a vector

$$\mathbf{m}_{1:t+1} = \max_{\text{by line}} O_{t+1} T^{\mathsf{T}} \textsc{eye}(\mathbf{m}_{\mathbf{1:t}})$$

- Remember which state realises the max at each step to get the actual sequence.

Viterbi algorithm

input: T, O // transition and observation matrices: $O_{ij} = P(o_j|s_i)$

S_1 // initial distribution on states

X // vector of observations: obs at time i is $o_{X[i]}$

output: Y // most likely sequence: hidden state at time i is $s_{Y[i]}$

$n \leftarrow length(X)$ // number of observations

$\forall j, m[j] \leftarrow O_{jX[1]} S_1[j]$ // initial distribution of states given first observation

$\forall i, j, pred[i, j] \leftarrow \perp$ // best predecessor of state j after observation i

for i in $[2..n]$:

  foreach state s_j:

    $m'[j] \leftarrow \max_k \{O_{jX[i]} T_{kj} m[k]\}$

    $pred[i, j] \leftarrow \text{argmax}_k \{O_{jX[i]} T_{kj} m[k]\}$

  $m \leftarrow m'$

// Build the path

$Y[n] \leftarrow \text{argmax}_k m[k]$

for i in $\{n - 1, \ldots, 1\}$:

  $Y[i] \leftarrow pred[i + 1, Y[i + 1]]$

Most Likely Explanation – Decoding

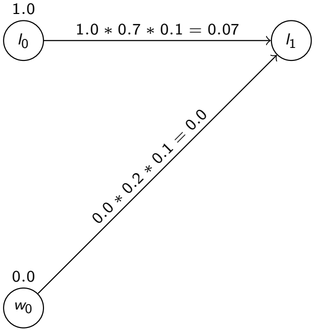
Attention estimation

What is the most likely explanation of the sequence s_1, c_2, p_3?

Most Likely Explanation – Decoding

Attention estimation

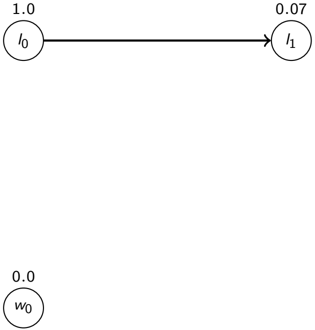
What is the most likely explanation of the sequence s_1, c_2, p_3?



Most Likely Explanation – Decoding

Attention estimation

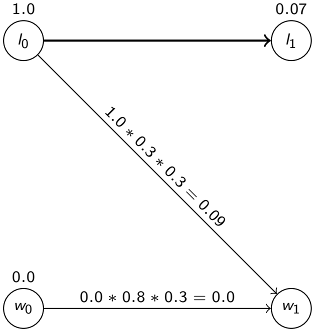
What is the most likely explanation of the sequence s_1, c_2, p_3?



Most Likely Explanation – Decoding

Attention estimation

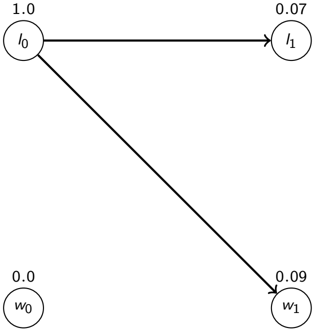
What is the most likely explanation of the sequence s_1, c_2, p_3?



Most Likely Explanation – Decoding

Attention estimation

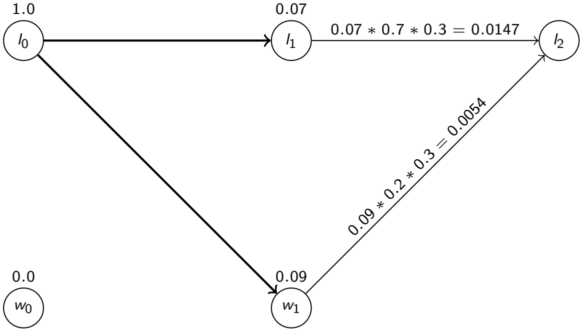
What is the most likely explanation of the sequence s_1, c_2, p_3?



Most Likely Explanation – Decoding

Attention estimation

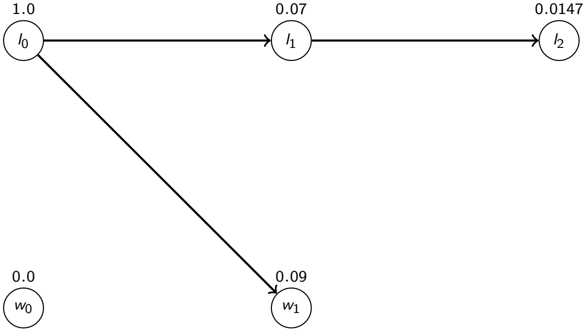
What is the most likely explanation of the sequence s_1, c_2, p_3?



Most Likely Explanation – Decoding

Attention estimation

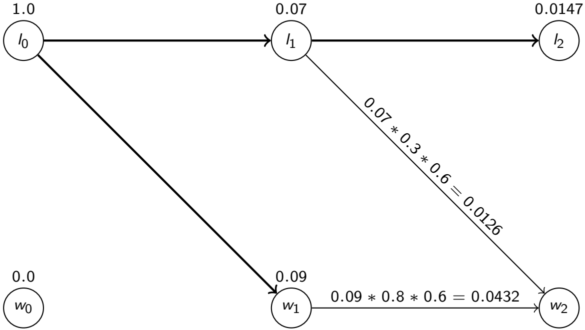
What is the most likely explanation of the sequence s_1, c_2, p_3?



Most Likely Explanation – Decoding

Attention estimation

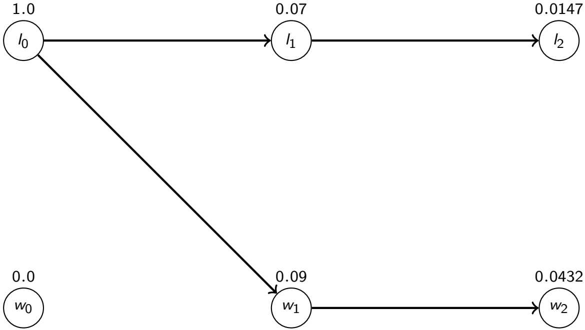
What is the most likely explanation of the sequence s_1, c_2, p_3?



Most Likely Explanation – Decoding

Attention estimation

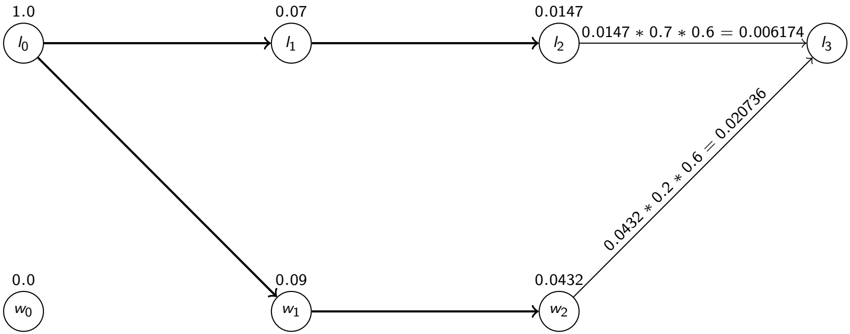
What is the most likely explanation of the sequence s_1, c_2, p_3?



Most Likely Explanation – Decoding

Attention estimation

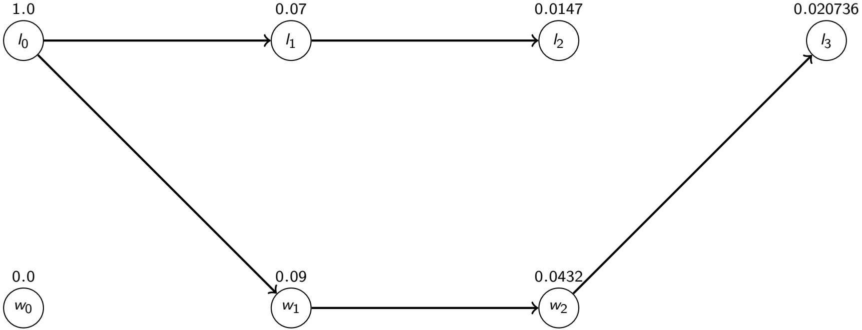
What is the most likely explanation of the sequence s_1, c_2, p_3?



Most Likely Explanation – Decoding

Attention estimation

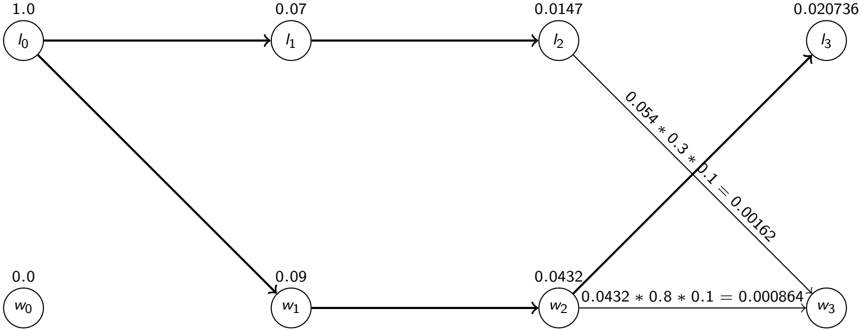
What is the most likely explanation of the sequence s_1, c_2, p_3?



Most Likely Explanation – Decoding

Attention estimation

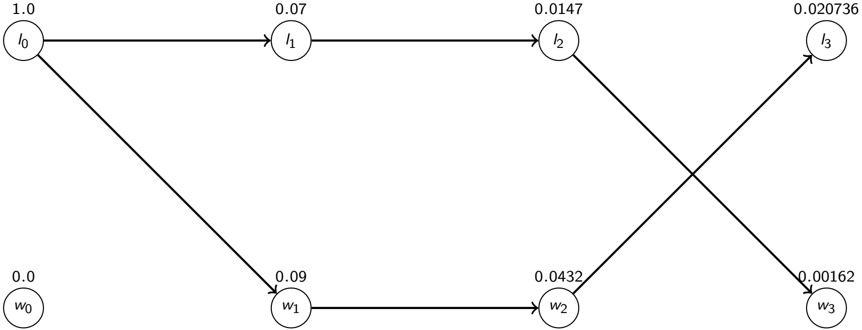
What is the most likely explanation of the sequence s_1, c_2, p_3?



Most Likely Explanation – Decoding

Attention estimation

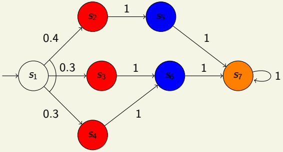
What is the most likely explanation of the sequence s_1, c_2, p_3?



Most Likely Explanation – Decoding

Exercise

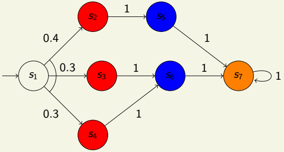
Apply Viterbi algorithm to find the correct explanation of  in:



Most Likely Explanation – Decoding

Exercise

Apply Viterbi algorithm to find the correct explanation of  in:



Small keyboard decoding

Given the frequencies of single letters and digrams (sequences of two letters), and considering that when hitting a key on the keyboard there is a 5% chance of hitting one the surrounding keys instead (uniformly distributed among those keys), write an HMC permitting to find the most likely intended key sequence given a sequence with typing errors.

Learning HMCs: The EM algorithm

- From sequences of observations, we can learn a corresponding HMC;
- The hidden states and observations must be known in advance: we learn the coefficients of the transition and observation matrices, and the initial distribution on hidden states;
- We use the Baum-Welch algorithm, a special case of the **Expectation-Maximisation** (EM) algorithm.

Learning HMCs: The EM algorithm

- Let $\theta = (T, O, X_0)$ be the parameters we want to learn;
- Given an observation sequence $o_{1:t}$, we want to maximise its probability (likelihood);
- That is we search:
$$\theta^* = \text{argmax}_\theta P_\theta(o_{1:t})$$
- Since we do not known the corresponding sequence $x_{1:t}$ of hidden states, this is:
$$\theta^* = \text{argmax}_\theta \sum_{x_{1:t}} P_\theta(o_{1:t}, x_{1:t})$$
- This requires enumerating all the possible values of the sequence $x_{1:t}$ however, which is intractable;
- Instead we **maximise** the **expectation** (for random sequences $x_{1:t}$, given $o_{1:t}$) of the (log-)likelihood, and proceed iteratively:
$$\begin{cases} \theta^0 \text{ chosen arbitrarily} \\ \theta^{n+1} = \text{argmax}_{\theta^n} \sum_{x_{1:t}} \log\left(P_{\theta^n}(o_{1:t}, x_{1:t})\right) P_{\theta^n}(x_{1:t}|o_{1:t}) \end{cases}$$
- This converges to a local optimum.

Learning HMCs: The Baum-Welch algorithm

- In the special case of HMCs, we can prove that θ^{n+1} can be computed using:
  - $\gamma_i(k) = P_{\theta^n}(x_k = s_i | o_{1:t})$.
    This is just **smoothing**.
  - $\zeta_{ij}(k) = P_{\theta^n}(x_k = s_i, x_{k+1} = s_j | o_{1:t})$.
    This is the normalisation of matrix $\mathbf{f}_k \circ (\mathbf{b}_{k+1} \circ O_{:t+1})^\mathsf{T} \circ T$
    $O_{:t+1}$ is the $t+1$-th column of O
- And:
  - $X_0^{n+1} = \gamma(1)$
  - $T_{ij}^{n+1} = \frac{\sum_{k=1}^{t-1} \zeta_{ij}(k)}{\sum_{k=1}^{t} \gamma_i(k)}$: the expected total number of transitions between s_i and s_j divided by the expected total number of transitions from state s_i;
  - $O_{ij}^{n+1} = \frac{\sum_{k=1, o_k=j}^{t} \gamma_i(k)}{\sum_{k=1}^{t} \gamma_i(k)}$: the expected total number of times we observed o_j when in state s_i divided by the expected total number of times we were in state s_i.

Conclusion

- If the matrices T and O are invertible, we can do some (space) optimisations to the backward computation;

Conclusion

- If the matrices T and O are invertible, we can do some (space) optimisations to the backward computation;
- The setting of HMCs can be generalised into **Dynamic Bayesian Networks** (DBNs);

Conclusion

- If the matrices T and O are invertible, we can do some (space) optimisations to the backward computation;
- The setting of HMCs can be generalised into **Dynamic Bayesian Networks** (DBNs);
- Another instance of DBNs using (linear) gaussian distributions is the **Kalman filter**;

Conclusion

- If the matrices T and O are invertible, we can do some (space) optimisations to the backward computation;
- The setting of HMCs can be generalised into **Dynamic Bayesian Networks** (DBNs);
- Another instance of DBNs using (linear) gaussian distributions is the **Kalman filter**;
- HMCs/ DBNs can be learnt using the **Expectation-Maximisation** (EM) algorithm.

Outline

Multiagent systems

- In many systems, there are several agents **acting concurrently** on the environment;

Multiagent systems

- In many systems, there are several agents **acting concurrently** on the environment;
- Recall that an agent tries to reach some objective or maximise some performance;

Multiagent systems

- In many systems, there are several agents **acting concurrently** on the environment;
- Recall that an agent tries to reach some objective or maximise some performance;
- Each agent must take into account the actions of others in its strategies;

Multiagent systems

- In many systems, there are several agents **acting concurrently** on the environment;
- Recall that an agent tries to reach some objective or maximise some performance;
- Each agent must take into account the actions of others in its strategies;
- We (continue to) model this as games on graphs: there are now several players.

Sequential games

- We consider 2-player zero-sum games (the rewards for each player are opposite);

Sequential games

- We consider 2-player zero-sum games (the rewards for each player are opposite);
- We extend MDPs by **partitioning** the set of actions A into $A_1 \cup A_2$, with $A_1 \cap A_2 = \emptyset$;

Sequential games

- We consider 2-player zero-sum games (the rewards for each player are opposite);
- We extend MDPs by **partitioning** the set of actions A into $A_1 \cup A_2$, with $A_1 \cap A_2 = \emptyset$;
- A strategy for Player i is now to choose actions from A_i;

Sequential games

- We consider 2-player zero-sum games (the rewards for each player are opposite);
- We extend MDPs by **partitioning** the set of actions A into $A_1 \cup A_2$, with $A_1 \cap A_2 = \emptyset$;
- A strategy for Player i is now to choose actions from A_i;
- Given strategies π_1 and π_2 for both players, the **outcome** is non-deterministic:

Sequential games

- We consider 2-player zero-sum games (the rewards for each player are opposite);
- We extend MDPs by **partitioning** the set of actions A into $A_1 \cup A_2$, with $A_1 \cap A_2 = \emptyset$;
- A strategy for Player i is now to choose actions from A_i;
- Given strategies π_1 and π_2 for both players, the **outcome** is non-deterministic:

Definition (Outcome)

The **outcome** $\mathrm{Outcome}(s, \pi_1, \pi_2)$ of strategies π_1 and π_2 from state s is the state sequence inductively defined by:

Sequential games

- We consider 2-player zero-sum games (the rewards for each player are opposite);
- We extend MDPs by **partitioning** the set of actions A into $A_1 \cup A_2$, with $A_1 \cap A_2 = \emptyset$;
- A strategy for Player i is now to choose actions from A_i;
- Given strategies π_1 and π_2 for both players, the **outcome** is non-deterministic:

Definition (Outcome)

The **outcome** $\mathrm{Outcome}(s, \pi_1, \pi_2)$ of strategies π_1 and π_2 from state s is the state sequence inductively defined by:

- $s \in \mathrm{Outcome}(s, \pi_1, \pi_2)$;

Sequential games

- We consider 2-player zero-sum games (the rewards for each player are opposite);
- We extend MDPs by **partitioning** the set of actions A into $A_1 \cup A_2$, with $A_1 \cap A_2 = \emptyset$;
- A strategy for Player i is now to choose actions from A_i;
- Given strategies π_1 and π_2 for both players, the **outcome** is non-deterministic:

Definition (Outcome)

The **outcome** $\text{Outcome}(s, \pi_1, \pi_2)$ of strategies π_1 and π_2 from state s is the state sequence inductively defined by:

- $s \in \text{Outcome}(s, \pi_1, \pi_2)$;
- if $\sigma \in \text{Outcome}(s, \pi_1, \pi_2)$ then $\sigma.s' \in \text{Outcome}(s, \pi_1, \pi_2)$ iff there exist an action a and a state s' such that $s \xrightarrow{a} s'$ and $a \in \pi_1(\sigma)$ or $a \in \pi_2(\sigma)$.

Utility of States

- Since the game is zero-sum we assume that if Player 1 tries to **maximise** the rewards, then Player 2 tries to **minimise** them;
  We could also give a $-R$ reward to Player 2 for each R reward of Player 1 and have both players try to maximise their rewards

Utility of States

- Since the game is zero-sum we assume that if Player 1 tries to **maximise** the rewards, then Player 2 tries to **minimise** them;
  We could also give a $-R$ reward to Player 2 for each R reward of Player 1 and have both players try to maximise their rewards
- Given strategies π_1 and π_2 for both players, the utility of state s for Player 1 is then:

$$U^{\pi_1,\pi_2}(s) = R(s) + \gamma \min\Big(\sum_{s' \in S} P(s'|\pi_1(s), s) U^{\pi_1,\pi_2}(s'), \sum_{s' \in S} P(s'|\pi_2(s), s) U^{\pi_1,\pi_2}(s')\Big)$$

Utility of States

- Since the game is zero-sum we assume that if Player 1 tries to **maximise** the rewards, then Player 2 tries to **minimise** them;
  We could also give a $-R$ reward to Player 2 for each R reward of Player 1 and have both players try to maximise their rewards
- Given strategies π_1 and π_2 for both players, the utility of state s for Player 1 is then:

$$U^{\pi_1,\pi_2}(s) = R(s) + \gamma \min\Big(\sum_{s' \in S} P(s'|\pi_1(s), s)U^{\pi_1,\pi_2}(s'), \sum_{s' \in S} P(s'|\pi_2(s), s)U^{\pi_1,\pi_2}(s')\Big)$$

- And therefore the optimal strategies π_1^* and π_2^* satisfy:

$$U^{\pi_1^*,\pi_2^*}(s) = R(s) + \gamma \min\Big(\max_{a \in A_1} \sum_{s' \in S} P(s'|a, s)U^{\pi_1^*,\pi_2^*}(s'), \min_{a \in A_2} \sum_{s' \in S} P(s'|a, s)U^{\pi_1^*,\pi_2^*}(s')\Big)$$

Utility of States

- Since the game is zero-sum we assume that if Player 1 tries to **maximise** the rewards, then Player 2 tries to **minimise** them;
  We could also give a $-R$ reward to Player 2 for each R reward of Player 1 and have both players try to maximise their rewards
- Given strategies π_1 and π_2 for both players, the utility of state s for Player 1 is then:

$$U^{\pi_1,\pi_2}(s) = R(s) + \gamma \min\Big(\sum_{s' \in S} P(s'|\pi_1(s), s)U^{\pi_1,\pi_2}(s'), \sum_{s' \in S} P(s'|\pi_2(s), s)U^{\pi_1,\pi_2}(s')\Big)$$

- And therefore the optimal strategies π_1^* and π_2^* satisfy:

$$U^{\pi_1^*,\pi_2^*}(s) = R(s) + \gamma \min\Big(\max_{a \in A_1} \sum_{s' \in S} P(s'|a, s)U^{\pi_1^*,\pi_2^*}(s'), \min_{a \in A_2} \sum_{s' \in S} P(s'|a, s)U^{\pi_1^*,\pi_2^*}(s')\Big)$$

- From the previous equation, we can easily adapt **value iteration** and **policy iteration** to find optimal strategies.

Turn-based Games

- A game played on a graph is **turn-based** if for any state s all the outgoing actions belong to a single player
  We do not require alternation

Turn-based Games

- A game played on a graph is **turn-based** if for any state s all the outgoing actions belong to a single player
  We do not require alternation
- We then say that s itself belongs to that player;

Turn-based Games

- A game played on a graph is **turn-based** if for any state s all the outgoing actions belong to a single player
  We do not require alternation
- We then say that s itself belongs to that player;
- This allows to simplify the previous equation:

Turn-based Games

- A game played on a graph is **turn-based** if for any state s all the outgoing actions belong to a single player
  We do not require alternation
- We then say that s itself belongs to that player;
- This allows to simplify the previous equation:
  - if s belongs to Player 1:

$$U^{\pi_1^*,\pi_2^*}(s) = R(s) + \gamma \max_{a \in A_1} \sum_{s' \in S} P(s'|a,s) U^{\pi_1^*,\pi_2^*}(s')$$

Turn-based Games

- A game played on a graph is **turn-based** if for any state s all the outgoing actions belong to a single player
  We do not require alternation
- We then say that s itself belongs to that player;
- This allows to simplify the previous equation:
  - if s belongs to Player 1:

$$U^{\pi_1^*,\pi_2^*}(s) = R(s) + \gamma \max_{a \in A_1} \sum_{s' \in S} P(s'|a,s) U^{\pi_1^*,\pi_2^*}(s')$$

  - if s belongs to Player 2:

$$U^{\pi_1^*,\pi_2^*}(s) = R(s) + \gamma \min_{a \in A_2} \sum_{s' \in S} P(s'|a,s) U^{\pi_1^*,\pi_2^*}(s')$$

Turn-based Games

Exercise

1. Prove that, in a turn-based game on a regular graph (with no probabilities, and no non-determinism), finding a strategy (if its exists) for Player 1 to reach some state whatever Player 2 does is equivalent to finding a strategy in an “and-or” graph (aka hypergraph).
2. Same question for general zero-sum 2-player reachability games on (regular) graphs (i.e., removing the turn-based assumption).
3. Deduce from 2 an algorithm to solve (find if winning and extract the strategy) directly general zero-sum 2-player reachability games on (regular) graphs.

Turn-based Games

Subtraction game

1. Model the subtraction game (simpler variant of the game of Nim), where starting from 6, each player subtracts a number between 1 and 3. The result must be strictly positive and the first player unable to move loses (and the other wins). We start with Player 1. Apply the previous algorithm to confirm that Player 1 has a winning strategy.
2. Idem when both players decide concurrently how much to subtract and we subtract the sum. If the result is non-positive then Player 1 wins if it is even (or zero) and Player 2 if it is odd.

Dealing with Big State Spaces

- Value iteration eventually propagates the rewards all along all the paths between nodes;

Dealing with Big State Spaces

- Value iteration eventually propagates the rewards all along all the paths between nodes;
- In many applications the corresponding MDPs can be **huge**: Backgammon 10^{20}, Chess 10^{40}, Go 10^{170}, Flying a helicopter $10^{?}$, ...

Dealing with Big State Spaces

- Value iteration eventually propagates the rewards all along all the paths between nodes;
- In many applications the corresponding MDPs can be **huge**: Backgammon 10^{20}, Chess 10^{40}, Go 10^{170}, Flying a helicopter $10^{?}$, ...
- There are two natural ideas to cope with the complexity:

Dealing with Big State Spaces

- Value iteration eventually propagates the rewards all along all the paths between nodes;
- In many applications the corresponding MDPs can be **huge**: Backgammon 10^{20}, Chess 10^{40}, Go 10^{170}, Flying a helicopter $10^{?}$, ...
- There are two natural ideas to cope with the complexity:
  - arbitrarily **limit the length** of paths;

Dealing with Big State Spaces

- Value iteration eventually propagates the rewards all along all the paths between nodes;
- In many applications the corresponding MDPs can be **huge**: Backgammon 10^{20}, Chess 10^{40}, Go 10^{170}, Flying a helicopter $10^{?}$, ...
- There are two natural ideas to cope with the complexity:
  - arbitrarily **limit the length** of paths;
  - only **explore some** of the paths.

Outline

Limiting the depth

- We want to evaluate the utility of state s by considering only path of **maximal length** L from s;

Limiting the depth

- We want to evaluate the utility of state s by considering only path of **maximal length** L from s;
- We thus obtain a **tree** rooted at s;

Limiting the depth

- We want to evaluate the utility of state s by considering only path of **maximal length** L from s;
- We thus obtain a **tree** rooted at s;
- We cannot use the successors to get the utility of leaves so we need an **approximator function** $\tilde{U}$:

Limiting the depth

- We want to evaluate the utility of state s by considering only path of **maximal length** L from s;
- We thus obtain a **tree** rooted at s;
- We cannot use the successors to get the utility of leaves so we need an **approximator function** $\tilde{U}$:
  - Obtained by expert knowledge;

Limiting the depth

- We want to evaluate the utility of state s by considering only path of **maximal length** L from s;
- We thus obtain a **tree** rooted at s;
- We cannot use the successors to get the utility of leaves so we need an **approximator function** $\tilde{U}$:
  - Obtained by expert knowledge;
  - Or by learning (e.g. reinforcement learning).

Expectiminimax and Minimax

- Given a state s, the maximum-depth L, and utility approximator (evaluation function) $\tilde{U}$, **Expectiminimax** consists in:

Expectiminimax and Minimax

- Given a state s, the maximum-depth L, and utility approximator (evaluation function) $\tilde{U}$, **Expectiminimax** consists in:
  1. building the tree rooted at s of paths from s with length less than L;

Expectiminimax and Minimax

- Given a state s, the maximum-depth L, and utility approximator (evaluation function) $\tilde{U}$, **Expectiminimax** consists in:
  1. building the tree rooted at s of paths from s with length less than L;
  2. assigning utility $\tilde{U}(s')$ to all leaves s';

Expectiminimax and Minimax

- Given a state s, the maximum-depth L, and utility approximator (evaluation function) $\tilde{U}$, **Expectiminimax** consists in:
  1. building the tree rooted at s of paths from s with length less than L;
  2. assigning utility $\tilde{U}(s')$ to all leaves s';
  3. computing $U(s)$ using value iteration on the tree.

Expectiminimax and Minimax

- Given a state s, the maximum-depth L, and utility approximator (evaluation function) $\tilde{U}$, **Expectiminimax** consists in:
  1. building the tree rooted at s of paths from s with length less than L;
  2. assigning utility $\tilde{U}(s')$ to all leaves s';
  3. computing $U(s)$ using value iteration on the tree.
- The value of $U(s)$ computed should be more precise than $\tilde{U}(s)$ thanks to the L-steps **look-ahead**;

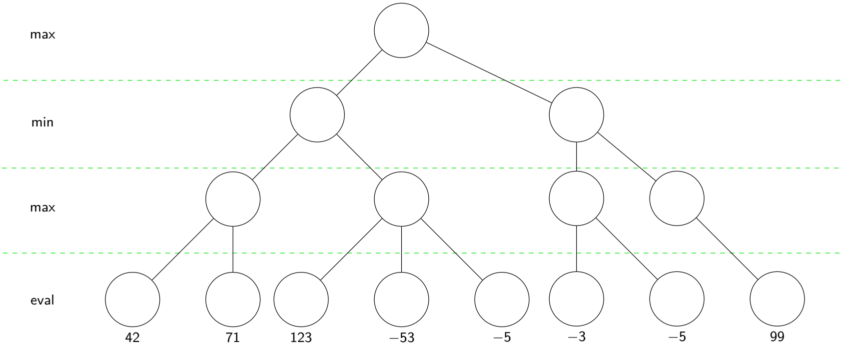
Expectiminimax and Minimax

- Given a state s, the maximum-depth L, and utility approximator (evaluation function) $\tilde{U}$, **Expectiminimax** consists in:
  1. building the tree rooted at s of paths from s with length less than L;
  2. assigning utility $\tilde{U}(s')$ to all leaves s';
  3. computing $U(s)$ using value iteration on the tree.
- The value of $U(s)$ computed should be more precise than $\tilde{U}(s)$ thanks to the L-steps **look-ahead**;
- If there are no probabilities then EXPECTIMINIMAX is called **Minimax** (historically introduced before).

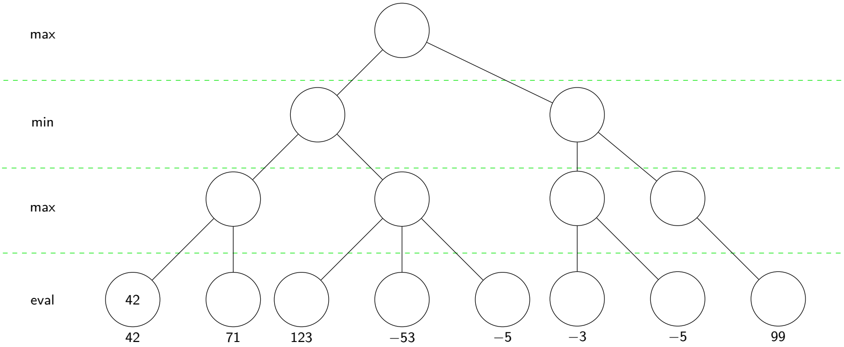
Minimax (turn-based)

```
input:  s          // source state
        maximizer  // player to play: true if Player 1 (maximiser)
        d          // depth (distance to L)
output: U          // utility of s

if d = 0 or s is terminal: // Leaf of the tree
    U ← evaluate(s)
else:
    if maximizer: // Player 1 is playing
        v ← −∞
        foreach child s′ of s:
            v ← max(v, MINIMAX(s′, not maximizer, d − 1))
    else: // Player 2 is playing
        v ← +∞
        foreach child s′ of s:
            v ← min(v, MINIMAX(s′, not maximizer, d − 1))
    U ← v
```

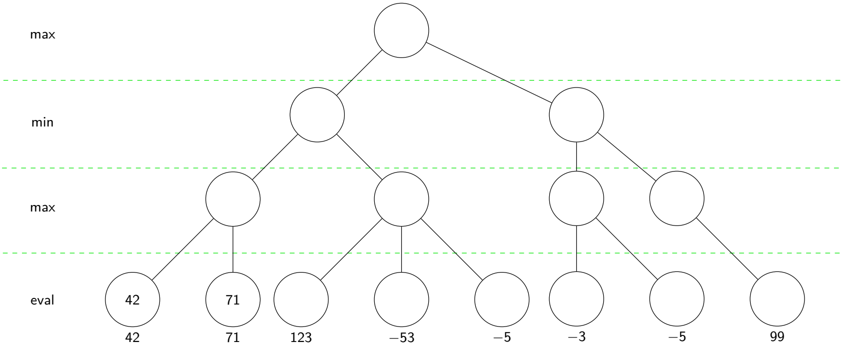
Minimax (turn-based)



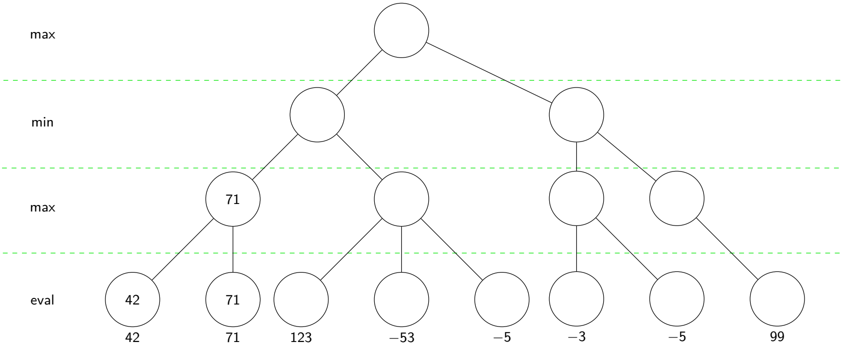
Minimax (turn-based)



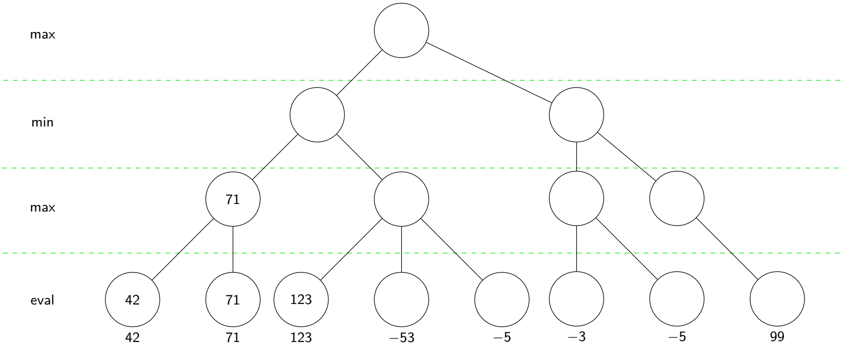
Minimax (turn-based)



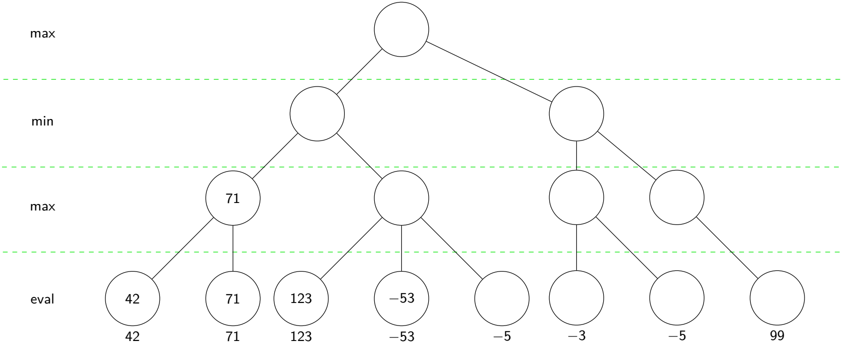
Minimax (turn-based)



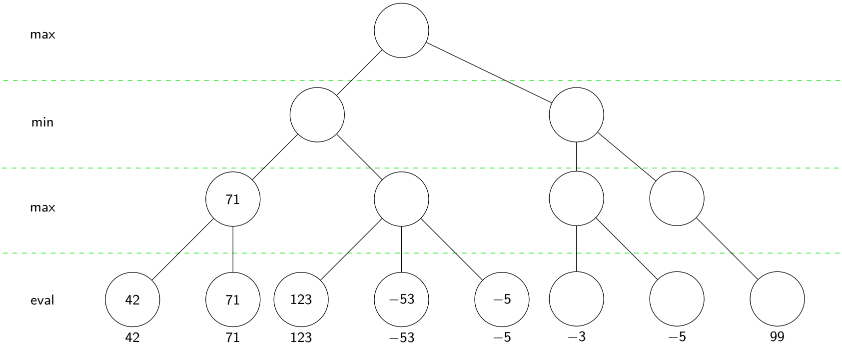
Minimax (turn-based)



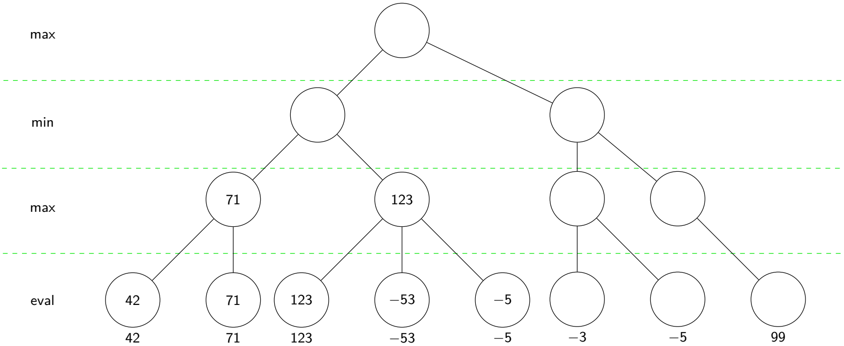
Minimax (turn-based)



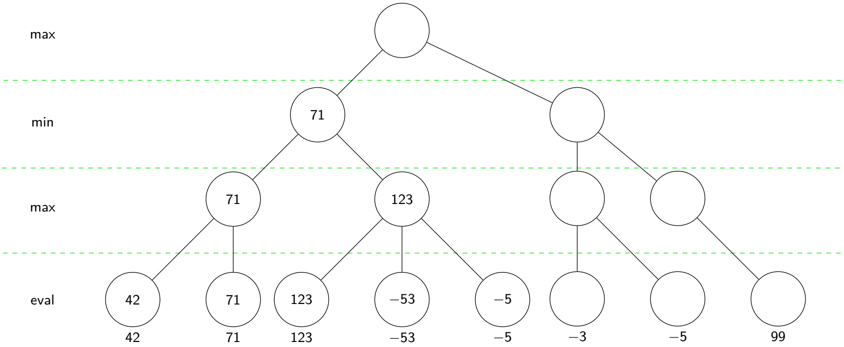
Minimax (turn-based)



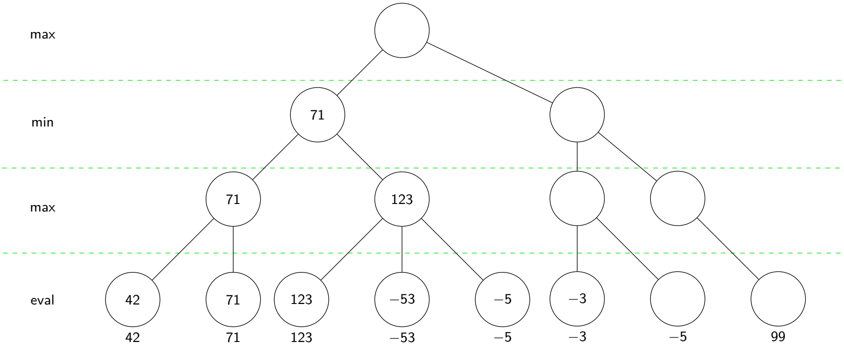
Minimax (turn-based)



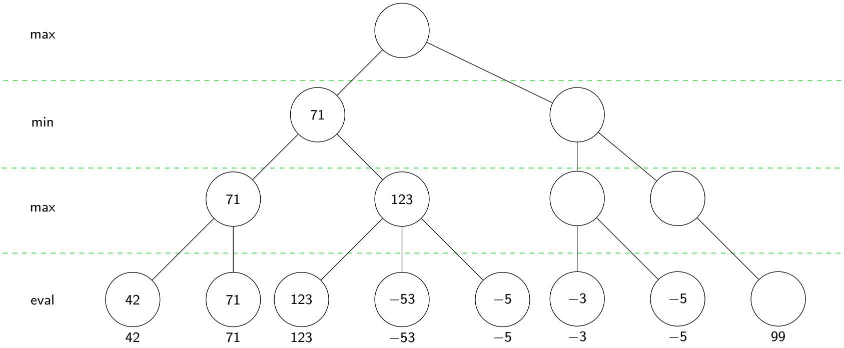
Minimax (turn-based)



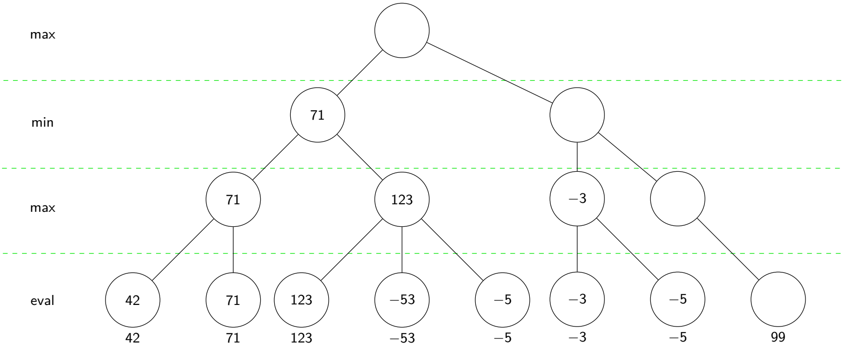
Minimax (turn-based)



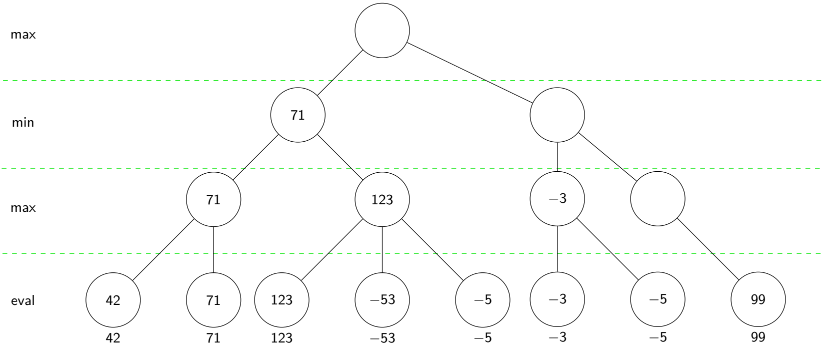
Minimax (turn-based)



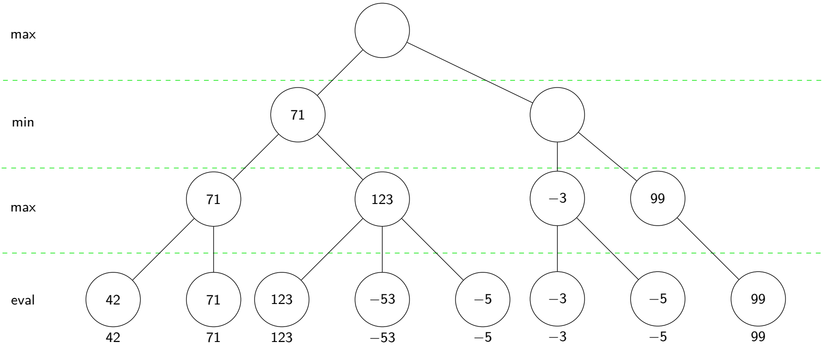
Minimax (turn-based)



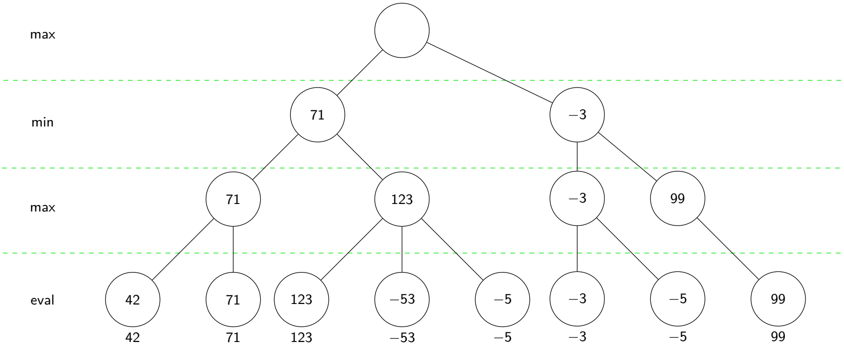
Minimax (turn-based)



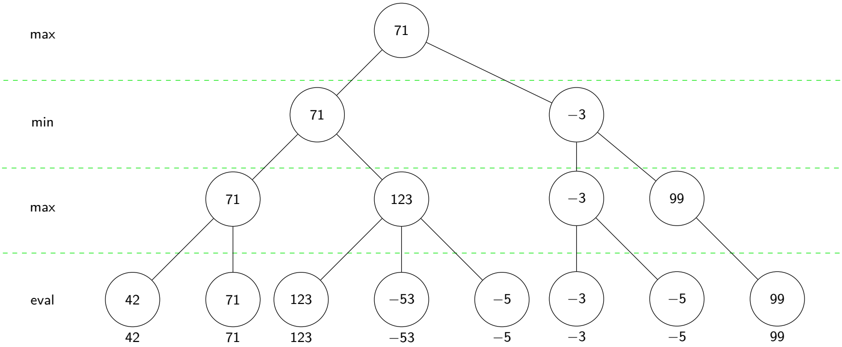
Minimax (turn-based)



Minimax (turn-based)



Minimax (turn-based)



Minimax: Choosing the best move

- It is possible to modify the previous algorithm to also return the best move;
- Or, more flexible, we can select the best move wiht another function, play_minimax :

```
input:  s          // source state, supposed non terminal
        maximizer  // joueur : true for maximiser, false for minimiser
        d          // depth (distance to L)
output: m          // best move

   if maximizer: // Player 1 plays
        m ← argmax_{s -a-> s'} MINIMAX(s', not maximizer, d − 1)
   else : // Player 2 moves
        m ← argmin_{s -a-> s'} MINIMAX(s', not maximizer, d − 1)
```

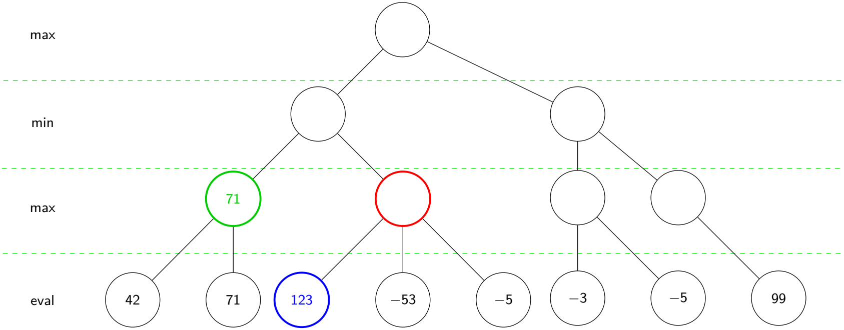
- We could also choose using a probability distribution on possible moves generated from the MINIMAX scores.

Negamax

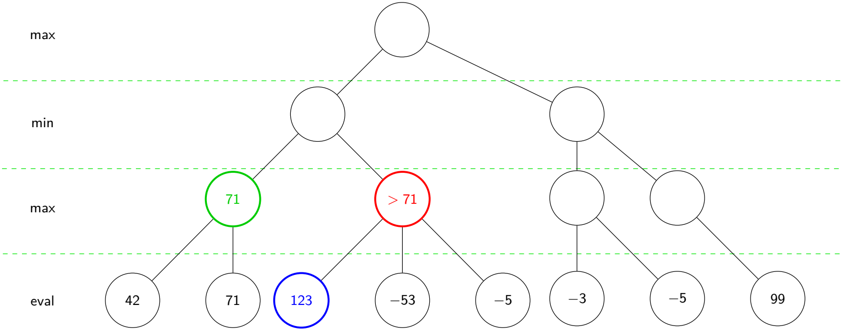
- We can have Player 2 maximise too by using the fact that $\min(a, b) = -\max(-a, -b)$

```
input:  s // source state
        p // player to play: +1 for Player 1, -1 for Player 2
        d // depth (distance to L)
output: U // utility of s

if d = 0 or s is terminal: // Leaf of the tree
    U ← evaluate(s)
else:
    v ← −∞
    foreach child s′ of s:
        v ← max(v, p ∗ NEGAMAX(s′, −p, d − 1))
    U ← p ∗ v
```

Alpha-beta Pruning

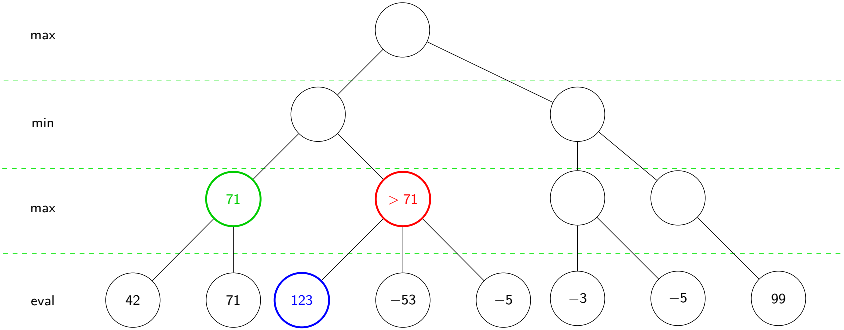


Alpha-beta Pruning



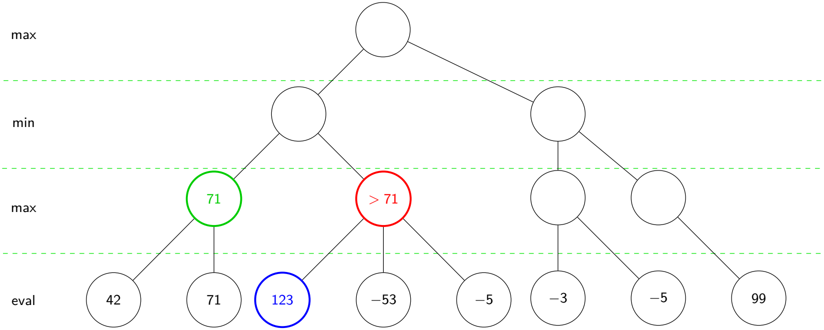
- After exploring , we know the value of is **at least** 123;
- Since the parent of is minimising, will never be chosen over ;

Alpha-beta Pruning



- After exploring , we know the value of is **at least** 123;
- Since the parent of is minimising, will never be chosen over ;
- So it is **futile** to continue explore its successors: β **cut-off**;

Alpha-beta Pruning

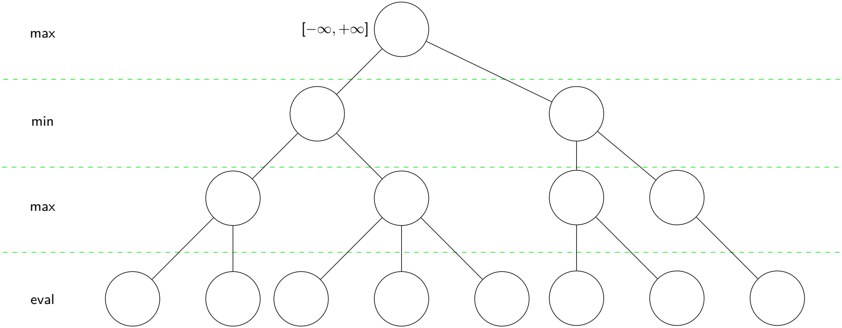


- After exploring ◯, we know the value of ◯ is **at least** 123;
- Since the parent of ◯ is minimising, ◯ will never be chosen over ◯;
- So it is **futile** to continue explore its successors: β **cut-off**;
- The same situation may arise with a child of a max node with a utility provably worse than another one: α **cut-off.**

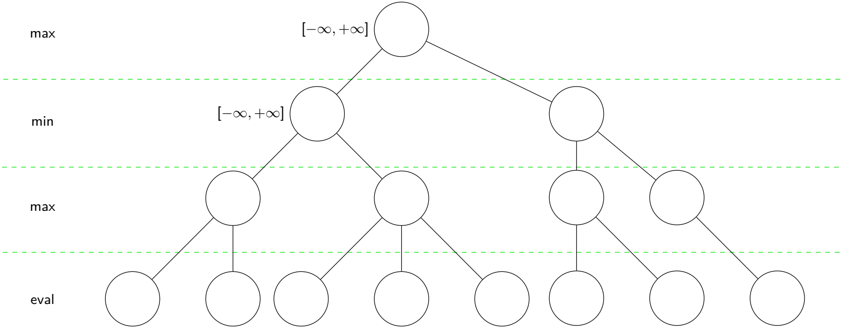
AlphaBeta

```
input:  s          // source state
        maximizer  // player to play: true if Player 1 (maximiser)
        d          // depth (distance to L)
        α          // utility lower-bound (start with −∞)
        β          // utility upper-bound (start with +∞)
output: U          // utility of s

if d = 0 or s is terminal: // Leaf of the tree
    U ← evaluate(s)
else:
    if maximizer: // Player 1 is playing
        v ← −∞
        foreach child s′ of s while α ≤ β:
            v ← max(v, AlphaBeta(s′, not maximizer, d − 1, α, β))
            α ← max(α, v)
    else: // Player 2 is playing
        v ← +∞
        foreach child s′ of s while α ≤ β:
            v ← min(v, AlphaBeta(s′, not maximizer, d − 1, α, β))
            β ← min(β, v)
    U ← v
```

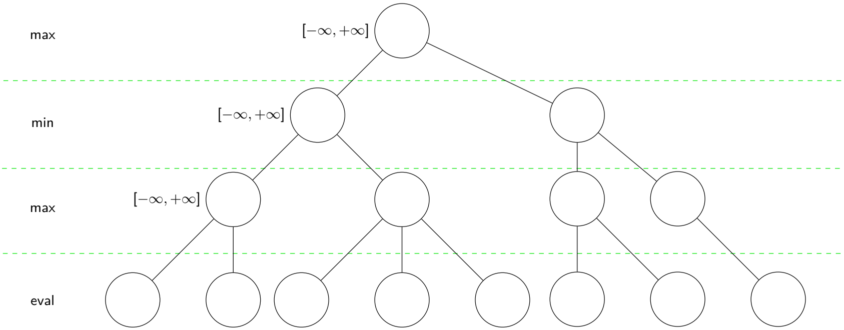
Alpha-beta Pruning



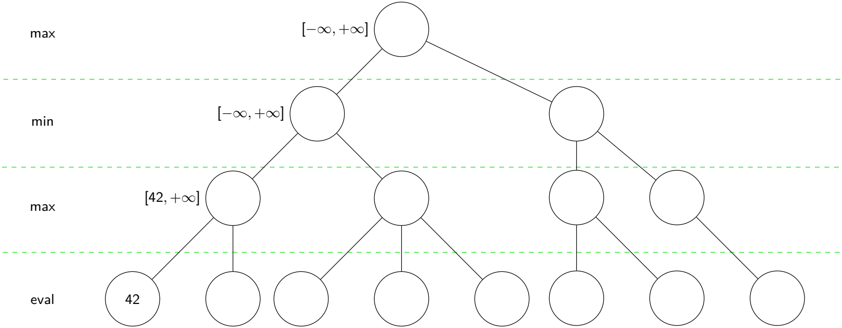
Alpha-beta Pruning



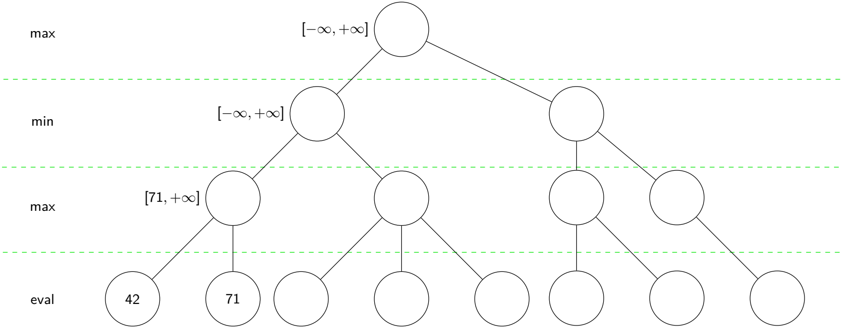
Alpha-beta Pruning



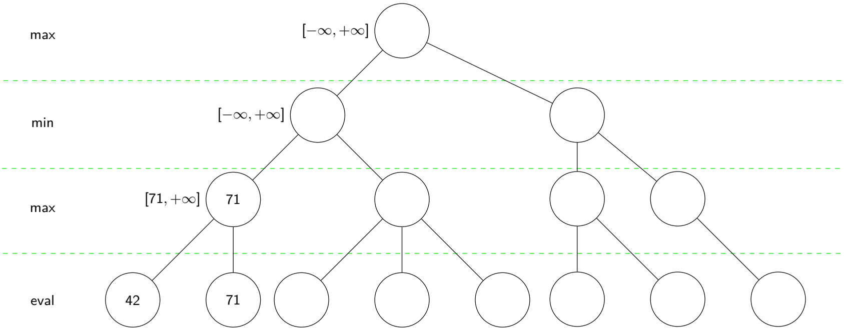
Alpha-beta Pruning



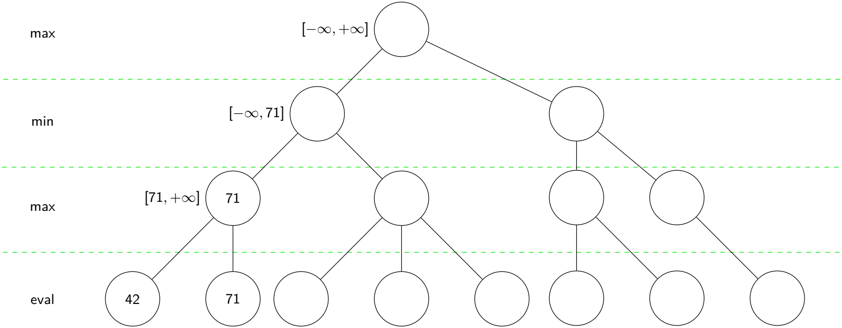
Alpha-beta Pruning



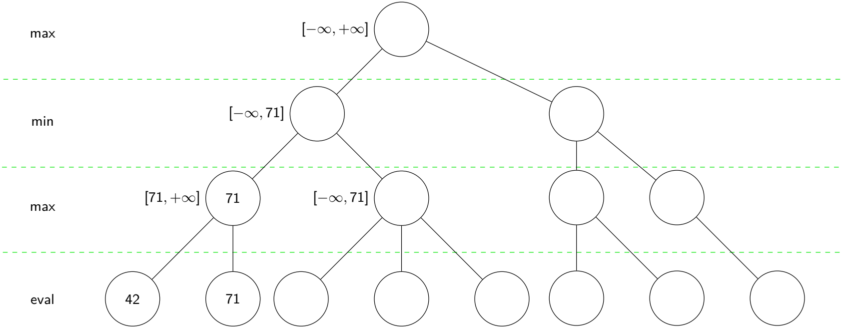
Alpha-beta Pruning



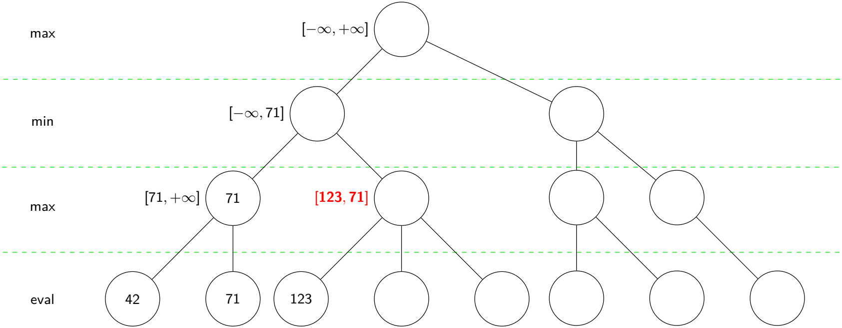
Alpha-beta Pruning



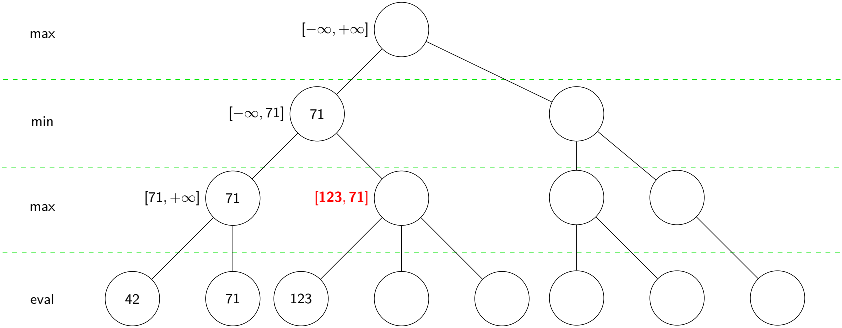
Alpha-beta Pruning



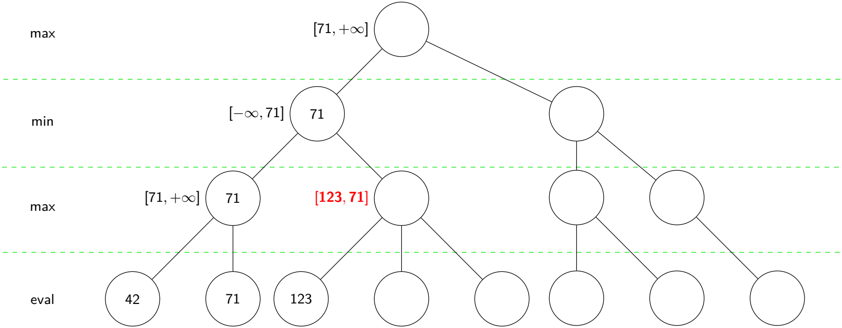
Alpha-beta Pruning



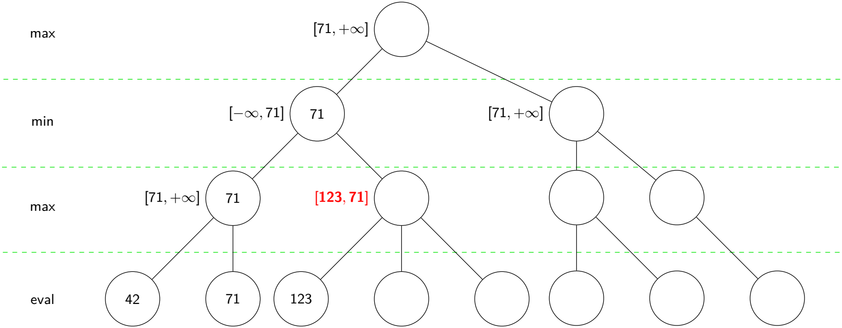
Alpha-beta Pruning



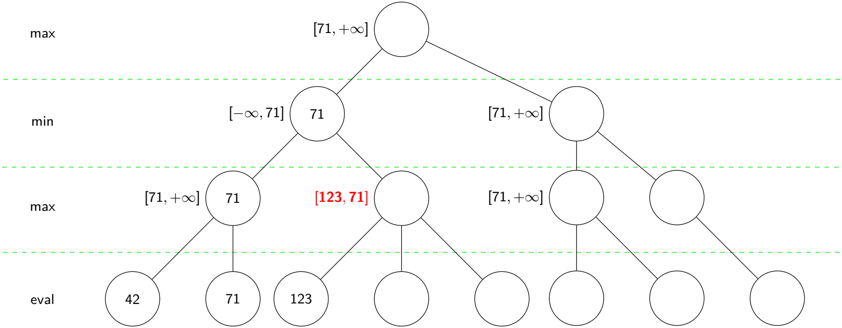
Alpha-beta Pruning



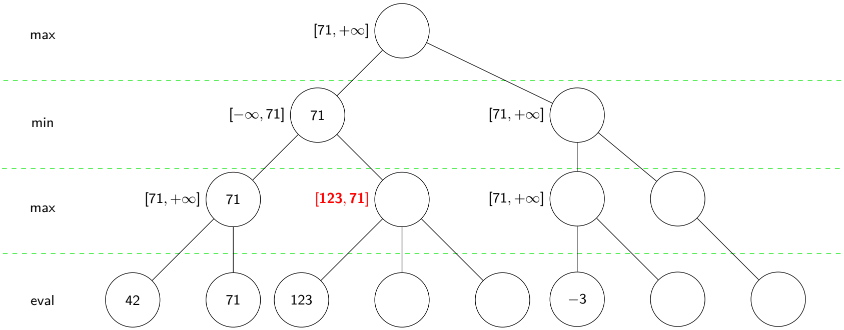
Alpha-beta Pruning



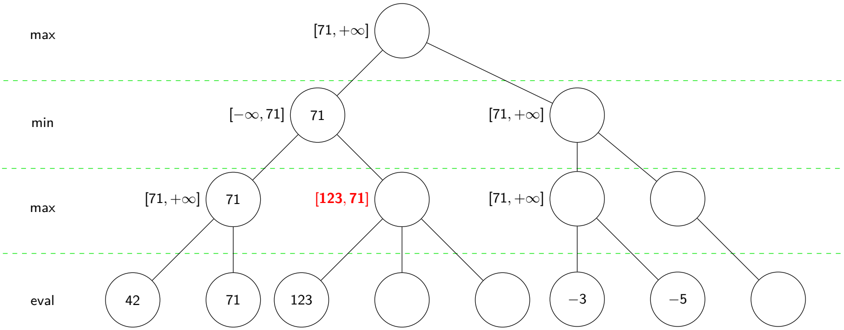
Alpha-beta Pruning



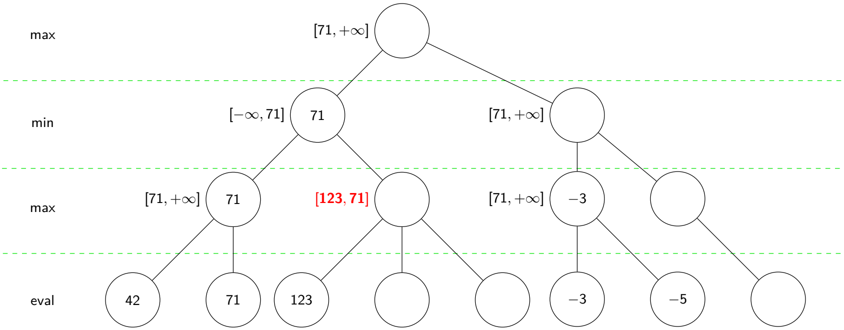
Alpha-beta Pruning



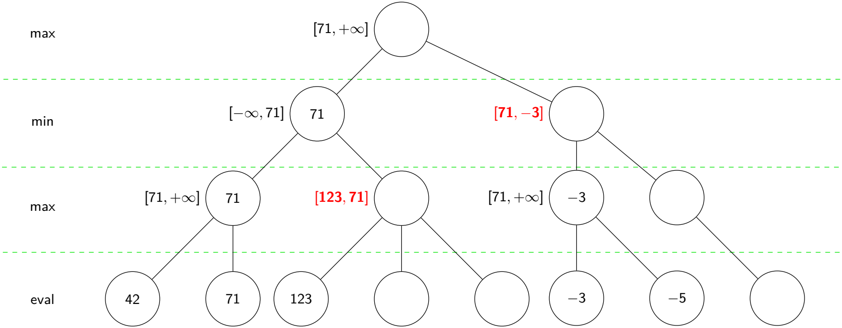
Alpha-beta Pruning



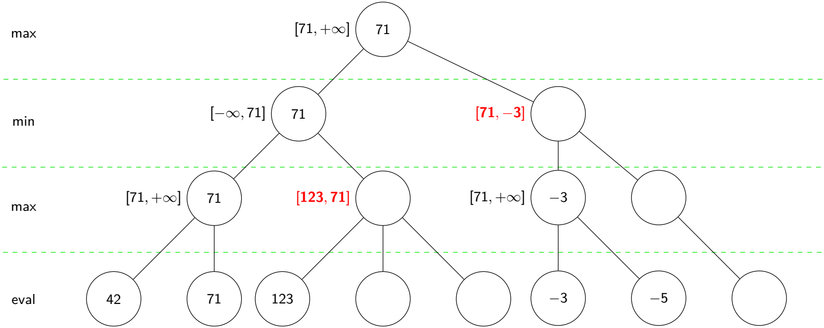
Alpha-beta Pruning



Alpha-beta Pruning



Alpha-beta Pruning



Alpha-beta Pruning

Exercise

Apply again the ALPHABETA algorithm on the previous example but consider that we always start with right-most child.

Alpha-beta Pruning

Exercise

Apply again the ALPHABETA algorithm on the previous example but consider that we always start with right-most child.

- With alpha-beta pruning, we can explore exponentially less nodes;
- But move ordering is crucial: the **best moves** for each player should be tried **first**;
- Finding a good move ordering can be done by shallower searches, notably using **iterative deepening**.

Outline

Monte-Carlo Tree Search (MCTS)

- We now consider exploring only part of the paths;

Monte-Carlo Tree Search (MCTS)

- We now consider exploring only part of the paths;
- A first idea is to **randomly sample** the set of paths starting after each direct move;

Monte-Carlo Tree Search (MCTS)

- We now consider exploring only part of the paths;
- A first idea is to **randomly sample** the set of paths starting after each direct move;
- Then we **execute** each run obtained and compute its reward;

Monte-Carlo Tree Search (MCTS)

- We now consider exploring only part of the paths;
- A first idea is to **randomly sample** the set of paths starting after each direct move;
- Then we **execute** each run obtained and compute its reward;
- Finally, we apply EXPECTIMINIMAX on the resulting tree;

Monte-Carlo Tree Search (MCTS)

- We now consider exploring only part of the paths;
- A first idea is to **randomly sample** the set of paths starting after each direct move;
- Then we **execute** each run obtained and compute its reward;
- Finally, we apply EXPECTIMINIMAX on the resulting tree;
- We need therefore those paths to be **finite**:

Monte-Carlo Tree Search (MCTS)

- We now consider exploring only part of the paths;
- A first idea is to **randomly sample** the set of paths starting after each direct move;
- Then we **execute** each run obtained and compute its reward;
- Finally, we apply EXPECTIMINIMAX on the resulting tree;
- We need therefore those paths to be **finite**:
    1. either all paths are finite by construction;

Monte-Carlo Tree Search (MCTS)

- We now consider exploring only part of the paths;
- A first idea is to **randomly sample** the set of paths starting after each direct move;
- Then we **execute** each run obtained and compute its reward;
- Finally, we apply EXPECTIMINIMAX on the resulting tree;
- We need therefore those paths to be **finite**:
  1. either all paths are finite by construction;
  2. or we explore to limited depth as before.

Monte-Carlo Tree Search (MCTS)

- We now consider exploring only part of the paths;
- A first idea is to **randomly sample** the set of paths starting after each direct move;
- Then we **execute** each run obtained and compute its reward;
- Finally, we apply EXPECTIMINIMAX on the resulting tree;
- We need therefore those paths to be **finite**:
  1. either all paths are finite by construction;
  2. or we explore to limited depth as before.
- The length of the paths is less limiting though;

Monte-Carlo Tree Search (MCTS)

- We now consider exploring only part of the paths;
- A first idea is to **randomly sample** the set of paths starting after each direct move;
- Then we **execute** each run obtained and compute its reward;
- Finally, we apply EXPECTIMINIMAX on the resulting tree;
- We need therefore those paths to be **finite**:
  1. either all paths are finite by construction;
  2. or we explore to limited depth as before.
- The length of the paths is less limiting though;
- The average reward converges to the true utility for the source state, but very slowly;

Monte-Carlo Tree Search (MCTS)

- We now consider exploring only part of the paths;
- A first idea is to **randomly sample** the set of paths starting after each direct move;
- Then we **execute** each run obtained and compute its reward;
- Finally, we apply EXPECTIMINIMAX on the resulting tree;
- We need therefore those paths to be **finite**:
  1. either all paths are finite by construction;
  2. or we explore to limited depth as before.
- The length of the paths is less limiting though;
- The average reward converges to the true utility for the source state, but very slowly;
- We explore lots of paths that would never be chosen by a **rational strategy**.

Monte-Carlo Tree Search (MCTS)

- Another idea is to combine this with building the tree **incrementally** to direct the search to the most promising moves;

Monte-Carlo Tree Search (MCTS)

- Another idea is to combine this with building the tree **incrementally** to direct the search to the most promising moves;
- Each time we add a node to the tree, we perform a random **playout** from that node;

Monte-Carlo Tree Search (MCTS)

- Another idea is to combine this with building the tree **incrementally** to direct the search to the most promising moves;
- Each time we add a node to the tree, we perform a random **playout** from that node;
- We **back-propagate** its reward all the way up to the root;

Monte-Carlo Tree Search (MCTS)

- Another idea is to combine this with building the tree **incrementally** to direct the search to the most promising moves;
- Each time we add a node to the tree, we perform a random **playout** from that node;
- We **back-propagate** its reward all the way up to the root;
- We maintain running statistics of the average reward, and number of playouts tried;

Monte-Carlo Tree Search (MCTS)

- Another idea is to combine this with building the tree **incrementally** to direct the search to the most promising moves;
- Each time we add a node to the tree, we perform a random **playout** from that node;
- We **back-propagate** its reward all the way up to the root;
- We maintain running statistics of the average reward, and number of playouts tried;
- The main challenge is, at each step, to choose between:

Monte-Carlo Tree Search (MCTS)

- Another idea is to combine this with building the tree **incrementally** to direct the search to the most promising moves;
- Each time we add a node to the tree, we perform a random **playout** from that node;
- We **back-propagate** its reward all the way up to the root;
- We maintain running statistics of the average reward, and number of playouts tried;
- The main challenge is, at each step, to choose between:
  - **Exploration**: Start exploring a new branch;

Monte-Carlo Tree Search (MCTS)

- Another idea is to combine this with building the tree **incrementally** to direct the search to the most promising moves;
- Each time we add a node to the tree, we perform a random **playout** from that node;
- We **back-propagate** its reward all the way up to the root;
- We maintain running statistics of the average reward, and number of playouts tried;
- The main challenge is, at each step, to choose between:
  - **Exploration**: Start exploring a new branch;
  - **Exploitation**: Extend an existing promising branch.

Multi-armed Bandits and UCB1

- Given n slot machines with different **unknown** expected gains;

Multi-armed Bandits and UCB1

- Given n slot machines with different **unknown** expected gains;
- If you win with one machine, should you continue to play with it, or try another one that could be even better?

Multi-armed Bandits and UCB1

- Given n slot machines with different **unknown** expected gains;
- If you win with one machine, should you continue to play with it, or try another one that could be even better?
- One possible strategy is **UCB1** (for *Upper Confidence Bound*):
  Play machine j that maximises $\bar{r}_j + \sqrt{\frac{2\ln(n)}{n_j}}$ where:

Multi-armed Bandits and UCB1

- Given n slot machines with different **unknown** expected gains;
- If you win with one machine, should you continue to play with it, or try another one that could be even better?
- One possible strategy is **UCB1** (for *Upper Confidence Bound*):
  Play machine j that maximises $\bar{r}_j + \sqrt{\frac{2\ln(n)}{n_j}}$ where:
  - $\bar{r}_j$ is the average reward on machine j up to now;

Multi-armed Bandits and UCB1

- Given n slot machines with different **unknown** expected gains;
- If you win with one machine, should you continue to play with it, or try another one that could be even better?
- One possible strategy is **UCB1** (for *Upper Confidence Bound*):
  Play machine j that maximises $\bar{r}_j + \sqrt{\frac{2\ln(n)}{n_j}}$ where:
  - $\bar{r}_j$ is the average reward on machine j up to now;
  - n is the total number of times we played;

Multi-armed Bandits and UCB1

- Given n slot machines with different **unknown** expected gains;
- If you win with one machine, should you continue to play with it, or try another one that could be even better?
- One possible strategy is **UCB1** (for *Upper Confidence Bound*):
  Play machine j that maximises $\bar{r}_j + \sqrt{\frac{2\ln(n)}{n_j}}$ where:
  - $\bar{r}_j$ is the average reward on machine j up to now;
  - n is the total number of times we played;
  - n_j is the number of times machine j was played.

Multi-armed Bandits and UCB1

- Given n slot machines with different **unknown** expected gains;
- If you win with one machine, should you continue to play with it, or try another one that could be even better?
- One possible strategy is **UCB1** (for *Upper Confidence Bound*):
  Play machine j that maximises $\bar{r}_j + \sqrt{\frac{2\ln(n)}{n_j}}$ where:
  - $\bar{r}_j$ is the average reward on machine j up to now;
  - n is the total number of times we played;
  - n_j is the number of times machine j was played.
- $\sqrt{\frac{2\ln(n)}{n_j}}$ is an **upper bound** of the size of the **confidence interval** for the average reward, such that the true expected reward has very high probability to fall inside;

Multi-armed Bandits and UCB1

- Given n slot machines with different **unknown** expected gains;
- If you win with one machine, should you continue to play with it, or try another one that could be even better?
- One possible strategy is **UCB1** (for *Upper Confidence Bound*):
  Play machine j that maximises $\bar{r}_j + \sqrt{\frac{2\ln(n)}{n_j}}$ where:
  - $\bar{r}_j$ is the average reward on machine j up to now;
  - n is the total number of times we played;
  - n_j is the number of times machine j was played.
- $\sqrt{\frac{2\ln(n)}{n_j}}$ is an **upper bound** of the size of the **confidence interval** for the average reward, such that the true expected reward has very high probability to fall inside;
- The less a machine is played the higher this value.

Upper Confidence on Trees (UCT)

We adapt UCB1 to our tree search as follows.

- At the top level:

Upper Confidence on Trees (UCT)

We adapt UCB1 to our tree search as follows.

- At the top level:
  1. Perform as many simulation **rounds** as desired (see below);

Upper Confidence on Trees (UCT)

We adapt UCB1 to our tree search as follows.

- At the top level:
  1. Perform as many simulation **rounds** as desired (see below);
  2. Select the move leading to the node j with the highest $p * r_j$, with $p \in \{-1, 1\}$ being the player to play.

Upper Confidence on Trees (UCT)

We adapt UCB1 to our tree search as follows.

- At the top level:
  1. Perform as many simulation **rounds** as desired (see below);
  2. Select the move leading to the node j with the highest $p * r_j$, with $p \in \{-1, 1\}$ being the player to play.
- For each round, starting at the root, and given a node:

Upper Confidence on Trees (UCT)

We adapt UCB1 to our tree search as follows.

- At the top level:
  1. Perform as many simulation **rounds** as desired (see below);
  2. Select the move leading to the node j with the highest $p * r_j$, with $p \in \{-1, 1\}$ being the player to play.
- For each round, starting at the root, and given a node:
  1. if a direct successor has never been explored ($n_j = 0$):

Upper Confidence on Trees (UCT)

We adapt UCB1 to our tree search as follows.

- At the top level:
  1. Perform as many simulation **rounds** as desired (see below);
  2. Select the move leading to the node j with the highest $p * r_j$, with $p \in \{-1, 1\}$ being the player to play.
- For each round, starting at the root, and given a node:
  1. if a direct successor has never been explored ($n_j = 0$):
     - Add it as a node to the tree;

Upper Confidence on Trees (UCT)

We adapt UCB1 to our tree search as follows.

- At the top level:
  1. Perform as many simulation **rounds** as desired (see below);
  2. Select the move leading to the node j with the highest $p * r_j$, with $p \in \{-1, 1\}$ being the player to play.
- For each round, starting at the root, and given a node:
  1. if a direct successor has never been explored ($n_j = 0$):
     - Add it as a node to the tree;
     - Perform a random playout from this node;

Upper Confidence on Trees (UCT)

We adapt UCB1 to our tree search as follows.

- At the top level:
  1. Perform as many simulation **rounds** as desired (see below);
  2. Select the move leading to the node j with the highest $p * r_j$, with $p \in \{-1, 1\}$ being the player to play.
- For each round, starting at the root, and given a node:
  1. if a direct successor has never been explored ($n_j = 0$):
     - Add it as a node to the tree;
     - Perform a random playout from this node;
     - Backpropagate the reward all the way up to the root, by adding it in all the nodes encountered;

Upper Confidence on Trees (UCT)

We adapt UCB1 to our tree search as follows.

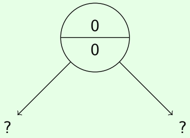
- At the top level:
  1. Perform as many simulation **rounds** as desired (see below);
  2. Select the move leading to the node j with the highest $p * r_j$, with $p \in \{-1, 1\}$ being the player to play.
- For each round, starting at the root, and given a node:
  1. if a direct successor has never been explored ($n_j = 0$):
     - Add it as a node to the tree;
     - Perform a random playout from this node;
     - Backpropagate the reward all the way up to the root, by adding it in all the nodes encountered;
  2. otherwise **recursively** go down into the tree by selecting the child using UCB1:

     maximise $p * \left(\frac{r_j}{n_j} + p * \sqrt{\frac{2\ln(n)}{n_j}}\right)$.

     n_j is the number of times we have performed a playout from node j and n is the sum of the n_j for all the nodes j we compare.

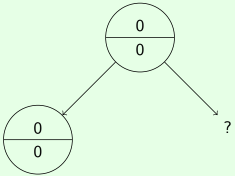
Upper Confidence on Trees (UCT)

Exemple



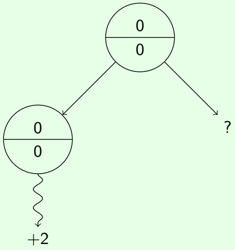
Upper Confidence on Trees (UCT)

Exemple



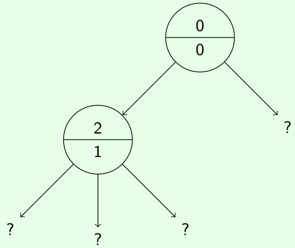
Upper Confidence on Trees (UCT)

Exemple



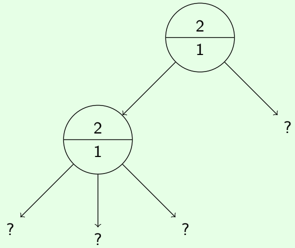
Upper Confidence on Trees (UCT)

Exemple



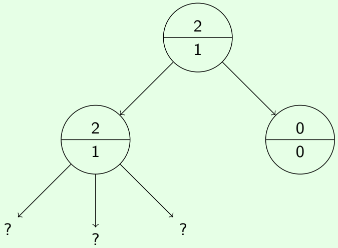
Upper Confidence on Trees (UCT)

Exemple



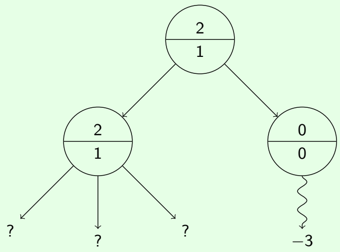
Upper Confidence on Trees (UCT)

Exemple



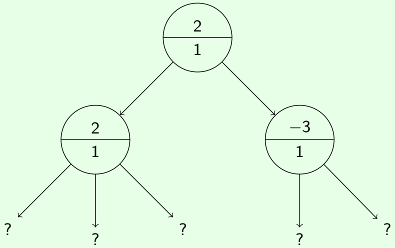
Upper Confidence on Trees (UCT)

Exemple



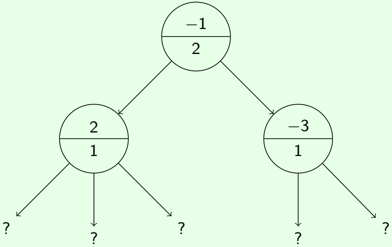
Upper Confidence on Trees (UCT)

Exemple



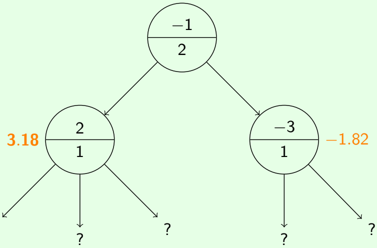
Upper Confidence on Trees (UCT)

Exemple



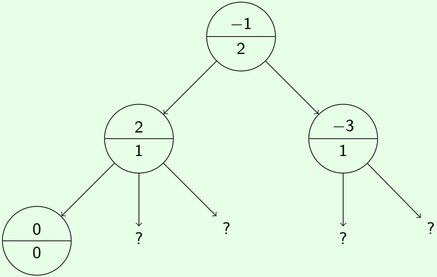
Upper Confidence on Trees (UCT)

Exemple



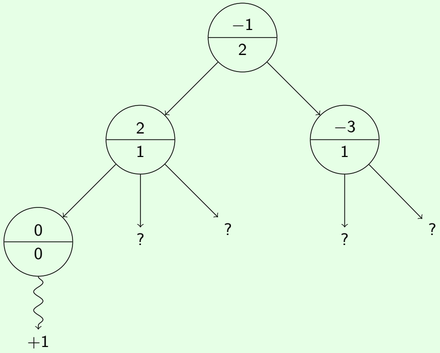
Upper Confidence on Trees (UCT)

Exemple



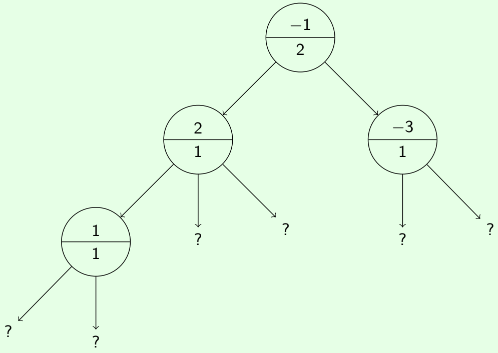
Upper Confidence on Trees (UCT)

Exemple



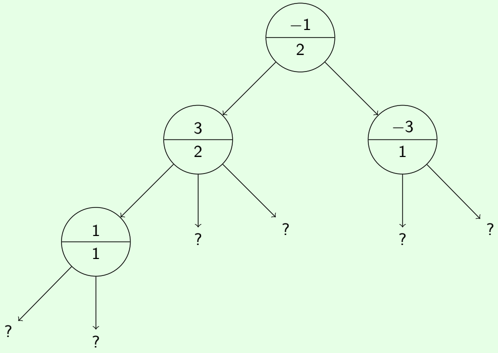
Upper Confidence on Trees (UCT)

Exemple



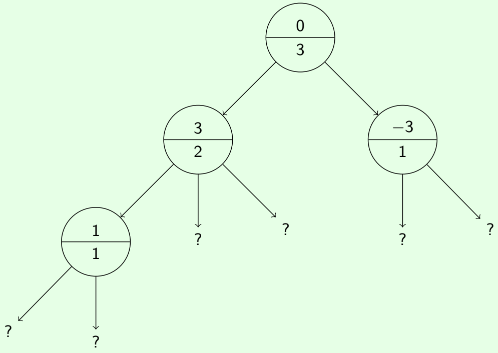
Upper Confidence on Trees (UCT)

Exemple



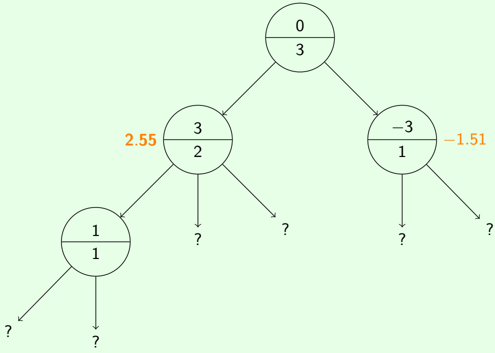
Upper Confidence on Trees (UCT)

Exemple



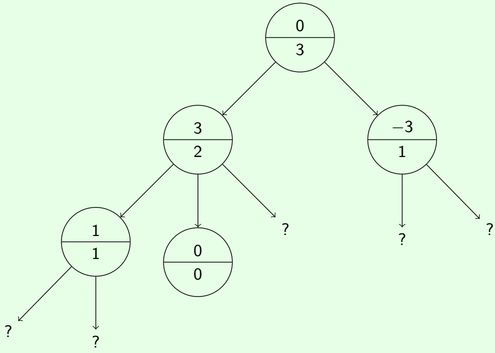
Upper Confidence on Trees (UCT)

Exemple



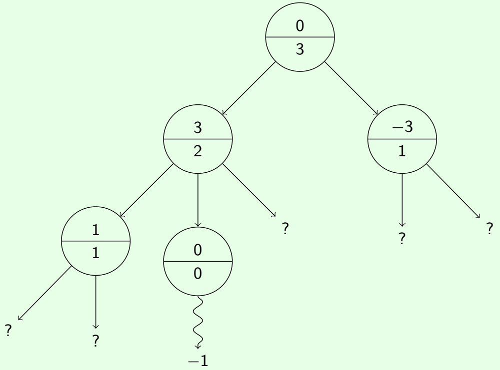
Upper Confidence on Trees (UCT)

Exemple



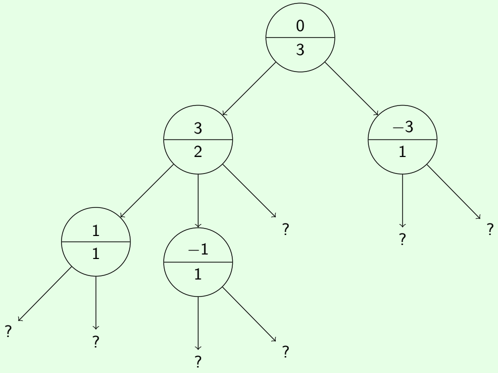
Upper Confidence on Trees (UCT)

Exemple



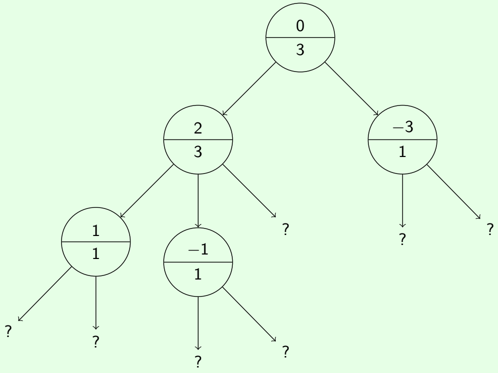
Upper Confidence on Trees (UCT)

Exemple



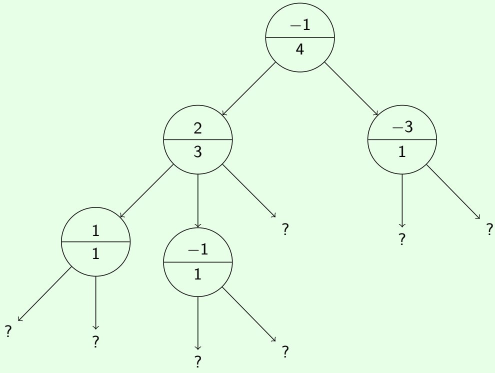
Upper Confidence on Trees (UCT)

Exemple



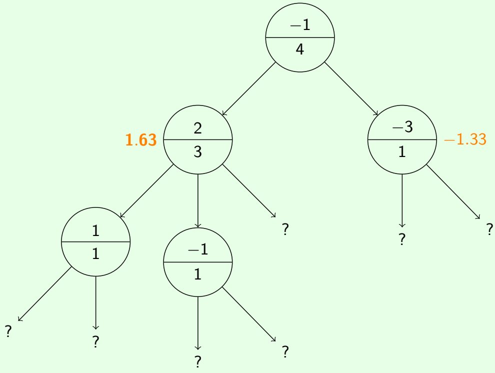
Upper Confidence on Trees (UCT)

Exemple



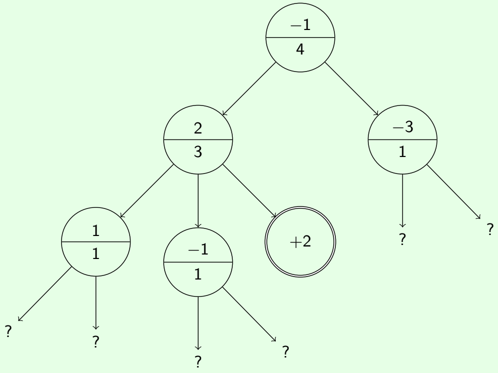
Upper Confidence on Trees (UCT)

Exemple



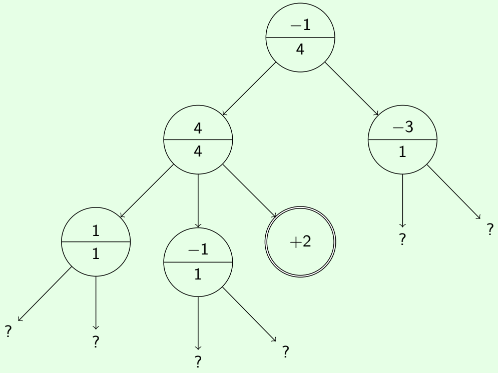
Upper Confidence on Trees (UCT)

Exemple



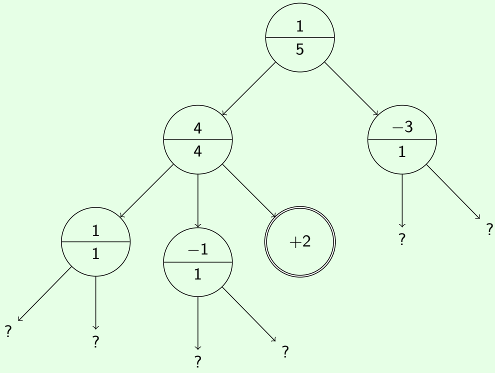
Upper Confidence on Trees (UCT)

Exemple



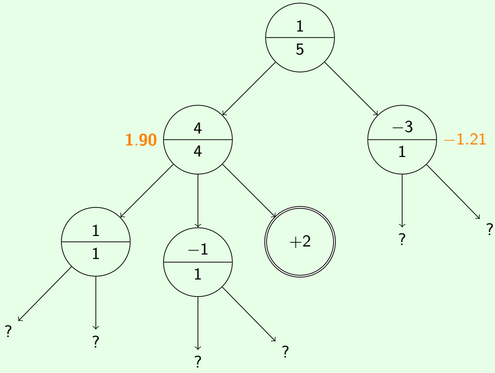
Upper Confidence on Trees (UCT)

Exemple



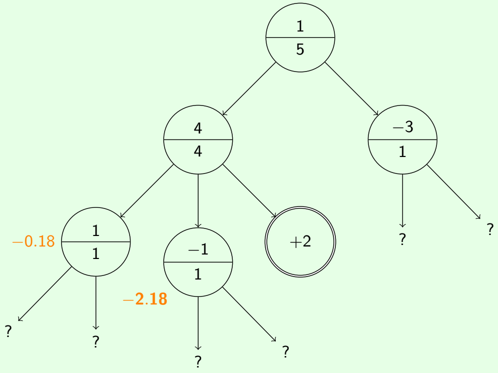
Upper Confidence on Trees (UCT)

Exemple



Upper Confidence on Trees (UCT)

Exemple



UCT

- At the limit, UCT **converges** to MINIMAX;
- The convergence is faster if we sample with a bias towards the **best moves** of each player;
- What a good move candidate might be from a given position can be **learnt**;
- UCT can be combined with depth-limiting;
- AlphaGo uses MCTS with a depth-limit and artificial neural networks for static position evaluation and finding potential good moves.

Outline

Supervised Learning

- Given **input-output pairs** $(\vec{x_1}, y_1), \ldots, (\vec{x_n}, y_n)$, generated by an **unknown** function f $(\forall i, y_i = f(\vec{x_i}))$, **supervised** (or inductive) learning consists in finding a function h that **approximates** f;

Supervised Learning

- Given **input-output pairs** $(\vec{x_1}, y_1), \ldots, (\vec{x_n}, y_n)$, generated by an **unknown** function f $(\forall i, y_i = f(\vec{x_i}))$, **supervised** (or inductive) learning consists in finding a function h that **approximates** f;
- Each $\vec{x_i}$ is a vector giving values to some finite set of **attributes**;

Supervised Learning

- Given **input-output pairs** $(\vec{x_1}, y_1), \ldots, (\vec{x_n}, y_n)$, generated by an **unknown** function f $(\forall i, y_i = f(\vec{x_i}))$, **supervised** (or inductive) learning consists in finding a function h that **approximates** f;
- Each $\vec{x_i}$ is a vector giving values to some finite set of **attributes**;
- **Classification** is supervised learning when the output of f is **finite**;

Supervised Learning

- Given **input-output pairs** $(\vec{x_1}, y_1), \ldots, (\vec{x_n}, y_n)$, generated by an **unknown** function f $(\forall i, y_i = f(\vec{x_i}))$, **supervised** (or inductive) learning consists in finding a function h that **approximates** f;
- Each $\vec{x_i}$ is a vector giving values to some finite set of **attributes**;
- **Classification** is supervised learning when the output of f is **finite**;
- Otherwise, it is **regression**.

Supervised Learning: Example

Software update decision

Consider the problem of deciding whether to apply now a software update or not. The attributes are:

1. **sec**: Is it a security update? (yes / no);
2. **reboot**: Does it require a reboot? (yes / no);
3. **in use**: Does it affect something I currently use? (yes / no);
4. **problems**: Does it update a software with which I currently have problems? (no / small problems / big problems);
5. **uptime**: Current duration without any reboot;
6. **crit**: Urgency / criticality of the work I am currently doing. (low / medium / high);
7. **size**: Size / duration of the update. (small / big);
8. **tod**: Time of the day (work / break);

Supervised Learning: Example

Example Data

in_use	reboot	problems	uptime	tod	crit	size	decision
no	no	big	0	work	high	small	yes
no	no	no	30	break	med	small	no
no	no	small	62	work	low	big	yes
no	no	big	21	work	low	big	yes
no	no	big	221	work	low	big	no
no	yes	small	234	work	med	small	no
no	yes	small	1234	break	med	small	yes
yes	yes	big	42	break	low	small	yes
yes	no	no	78	break	high	big	no
yes	no	small	444	break	low	big	yes
yes	yes	small	1024	work	high	big	no
yes	yes	small	10	work	med	small	yes
yes	no	small	24	work	med	big	no

Hypothesis space

- Searching our approximation h over the set of all functions is not feasible;
- We choose a priori an **hypothesis space** (or model):
  - Decision trees;
  - Hyperplanes;
  - Artificial neural networks;
  - ...
- **Hypotheses** h are learnt within an hypothesis space by minimising some **loss function**.

Hypothesis space

- A supervised learning problem is **realisable** if f belongs to the hypothesis space.

Hypothesis space

- A supervised learning problem is **realisable** if f belongs to the hypothesis space.
- But the more complex the hypothesis space:

Hypothesis space

- A supervised learning problem is **realisable** if f belongs to the hypothesis space.
- But the more complex the hypothesis space:
  - The more complex learning can be (e.g. more coefficients to find);

Hypothesis space

- A supervised learning problem is **realisable** if f belongs to the hypothesis space.
- But the more complex the hypothesis space:
  - The more complex learning can be (e.g. more coefficients to find);
  - The higher the risk of **overfitting**: with a polynomial of degree n, any n points can be fitted but the probability of good generalisation is low;

Hypothesis space

- A supervised learning problem is **realisable** if f belongs to the hypothesis space.
- But the more complex the hypothesis space:
  - The more complex learning can be (e.g. more coefficients to find);
  - The higher the risk of **overfitting**: with a polynomial of degree n, any n points can be fitted but the probability of good generalisation is low;
  - The more complex are the **computations** using the hypothesis for exploitation.

Hypothesis space

- A supervised learning problem is **realisable** if f belongs to the hypothesis space.
- But the more complex the hypothesis space:
  - The more complex learning can be (e.g. more coefficients to find);
  - The higher the risk of **overfitting**: with a polynomial of degree n, any n points can be fitted but the probability of good generalisation is low;
  - The more complex are the **computations** using the hypothesis for exploitation.
- **Ockham's razor** dictates to prefer simplicity.

Hyperparameters

- Within a given hypothesis space, we may need to further fix the value of **hyperparameters**:
  - degree of a polynomial model;
  - number of layers in a neural network;
  - number of neurons per layer;
  - ...
- Hyperparameters can also be learnt.

Generalisation

- The hypothesis should be **consistent**: agree (as much as possible) with the given data;

Generalisation

- The hypothesis should be **consistent**: agree (as much as possible) with the given data;
- But it should also **generalise** to other data;

Generalisation

- The hypothesis should be **consistent**: agree (as much as possible) with the given data;
- But it should also **generalise** to other data;
- To assess this, the given data is divided into **training** data and **test** data:

Generalisation

- The hypothesis should be **consistent**: agree (as much as possible) with the given data;
- But it should also **generalise** to other data;
- To assess this, the given data is divided into **training** data and **test** data:
  - Optimise / learn based on training data;

Generalisation

- The hypothesis should be **consistent**: agree (as much as possible) with the given data;
- But it should also **generalise** to other data;
- To assess this, the given data is divided into **training** data and **test** data:
  - Optimise / learn based on training data;
  - Test the performance using test data.

Generalisation

- The hypothesis should be **consistent**: agree (as much as possible) with the given data;
- But it should also **generalise** to other data;
- To assess this, the given data is divided into **training** data and **test** data:
  - Optimise / learn based on training data;
  - Test the performance using test data.
- Test data should never be used in the selection of **hyperparameters**, lest it introduces a bias:

Generalisation

- The hypothesis should be **consistent**: agree (as much as possible) with the given data;
- But it should also **generalise** to other data;
- To assess this, the given data is divided into **training** data and **test** data:
  - Optimise / learn based on training data;
  - Test the performance using test data.
- Test data should never be used in the selection of **hyperparameters**, lest it introduces a bias:
  - Further extract a **validation** data from the training data;

Generalisation

- The hypothesis should be **consistent**: agree (as much as possible) with the given data;
- But it should also **generalise** to other data;
- To assess this, the given data is divided into **training** data and **test** data:
  - Optimise / learn based on training data;
  - Test the performance using test data.
- Test data should never be used in the selection of **hyperparameters**, lest it introduces a bias:
  - Further extract a **validation** data from the training data;
  - Optimise the hyperparameters by testing with the validation set;

Generalisation

- The hypothesis should be **consistent**: agree (as much as possible) with the given data;
- But it should also **generalise** to other data;
- To assess this, the given data is divided into **training** data and **test** data:
  - Optimise / learn based on training data;
  - Test the performance using test data.
- Test data should never be used in the selection of **hyperparameters**, lest it introduces a bias:
  - Further extract a **validation** data from the training data;
  - Optimise the hyperparameters by testing with the validation set;
  - Fix the hyperparameter values and merge the validation data back in the training data;

Generalisation

- The hypothesis should be **consistent**: agree (as much as possible) with the given data;
- But it should also **generalise** to other data;
- To assess this, the given data is divided into **training** data and **test** data:
  - Optimise / learn based on training data;
  - Test the performance using test data.
- Test data should never be used in the selection of **hyperparameters**, lest it introduces a bias:
  - Further extract a **validation** data from the training data;
  - Optimise the hyperparameters by testing with the validation set;
  - Fix the hyperparameter values and merge the validation data back in the training data;
  - Learn the final model using all the training data. Evaluate using the test data.

k-fold cross-validation

1. Extract a test set;
2. Partition the rest of the data into k subsets S_i of equal size;
3. For each i, use S_i as validation set, and all the other sets as a single training set;
4. Choose hyperparameters that minimise the **average** error rate over the k rounds of training-evaluation;
5. Assess generalisation of the model using the test set.

Failure of the learning process

- The obtained classifier might not be the exact function because of:
  1. unrealisability / underfitting;
  2. overfitting;
  3. variance in the data;
  4. noise in the data;
  5. computational intractability of the hypothesis space.

Loss functions

- Some errors in classification are worse than others;

Loss functions

- Some errors in classification are worse than others;
- For regression, we will almost surely never have the exact output value with the classifier;

Loss functions

- Some errors in classification are worse than others;
- For regression, we will almost surely never have the exact output value with the classifier;
- **Loss** is an alternative to error rates based on the **utilities** of outputs:
  $\vec{x}$ is the input, y the true output, $\hat{y} = h(\vec{x})$ the output of the classifier h for x

$$L(\vec{x}, y, \hat{y}) = U(y, \vec{x}) - U(\hat{y}, \vec{x})$$

Loss functions

- Some errors in classification are worse than others;
- For regression, we will almost surely never have the exact output value with the classifier;
- **Loss** is an alternative to error rates based on the **utilities** of outputs:
  $\vec{x}$ is the input, y the true output, $\hat{y} = h(\vec{x})$ the output of the classifier h for x

$$L(\vec{x}, y, \hat{y}) = U(y, \vec{x}) - U(\hat{y}, \vec{x})$$

- Common loss functions include:

Loss functions

- Some errors in classification are worse than others;
- For regression, we will almost surely never have the exact output value with the classifier;
- **Loss** is an alternative to error rates based on the **utilities** of outputs:
  $\vec{x}$ is the input, y the true output, $\hat{y} = h(\vec{x})$ the output of the classifier h for x

$$L(\vec{x}, y, \hat{y}) = U(y, \vec{x}) - U(\hat{y}, \vec{x})$$

- Common loss functions include:
  - $L_1(y, \hat{y}) = |y - \hat{y}|$ for regression;

Loss functions

- Some errors in classification are worse than others;
- For regression, we will almost surely never have the exact output value with the classifier;
- **Loss** is an alternative to error rates based on the **utilities** of outputs:
  $\vec{x}$ is the input, y the true output, $\hat{y} = h(\vec{x})$ the output of the classifier h for x

$$L(\vec{x}, y, \hat{y}) = U(y, \vec{x}) - U(\hat{y}, \vec{x})$$

- Common loss functions include:
  - $L_1(y, \hat{y}) = |y - \hat{y}|$ for regression;
  - $L_2(y, \hat{y}) = (y - \hat{y})^2$ for regression;

Loss functions

- Some errors in classification are worse than others;
- For regression, we will almost surely never have the exact output value with the classifier;
- **Loss** is an alternative to error rates based on the **utilities** of outputs:
  $\vec{x}$ is the input, y the true output, $\hat{y} = h(\vec{x})$ the output of the classifier h for x

$$L(\vec{x}, y, \hat{y}) = U(y, \vec{x}) - U(\hat{y}, \vec{x})$$

- Common loss functions include:
  - $L_1(y, \hat{y}) = |y - \hat{y}|$ for regression;
  - $L_2(y, \hat{y}) = (y - \hat{y})^2$ for regression;
  - $L_{0/1}(y, \hat{y}) = 0$ if $y = \hat{y}$ else 1 for classification.

Loss functions

- Instead of minimising error rates we will minimise expected loss;

Loss functions

- Instead of minimising error rates we will minimise expected loss;
- But since we do not know the associated probabilities, we minimise **empirical loss** on the example set E:

$$\hat{L}_E(h) = \frac{1}{N} \sum_{(\vec{x},y)\in E} L(y, h(\vec{x}))$$

Regularisation

- Loss functions can be used to favor simple hypotheses;

Regularisation

- Loss functions can be used to favor simple hypotheses;
- We add a term $\lambda \text{Reg(h)}$ to the loss function, with $\lambda > 0$;

Regularisation

- Loss functions can be used to favor simple hypotheses;
- We add a term $\lambda\text{Reg}(h)$ to the loss function, with $\lambda > 0$;
- Reg(h) is a **regularisation** function;

Regularisation

- Loss functions can be used to favor simple hypotheses;
- We add a term λReg(h) to the loss function, with $\lambda > 0$;
- Reg(h) is a **regularisation** function;
- It is bigger when h is more complex: e.g. sum of the squares of the coefficients in a polynomial;

Regularisation

- Loss functions can be used to favor simple hypotheses;
- We add a term $\lambda\mathrm{Reg(h)}$ to the loss function, with $\lambda > 0$;
- Reg(h) is a **regularisation** function;
- It is bigger when h is more complex: e.g. sum of the squares of the coefficients in a polynomial;
- We then try to find the best value for λ as another **hyperparameter**.

Ensemble Learning

- Instead of producing one hypothesis, we might want to produce k and **combine** their predictions;

Ensemble Learning

- Instead of producing one hypothesis, we might want to produce k and **combine** their predictions;
- If their errors are jointly independent, we can decrease the error a lot by, e.g., a majority vote for classification;

Ensemble Learning

- Instead of producing one hypothesis, we might want to produce k and **combine** their predictions;
- If their errors are jointly independent, we can decrease the error a lot by, e.g., a majority vote for classification;
- It can also be used to combine simpler hypothesis with an "and": e.g. combine several hyperplane separators to obtain a convex polyhedron.

Ensemble Learning: Boosting

- **Boosting** consists in reusing the examples in the training data that are misclassified by the produced hypothesis;

Ensemble Learning: Boosting

- **Boosting** consists in reusing the examples in the training data that are misclassified by the produced hypothesis;
- They are then duplicated (or given a higher weight) to augment the original training set;

Ensemble Learning: Boosting

- **Boosting** consists in reusing the examples in the training data that are misclassified by the produced hypothesis;
- They are then duplicated (or given a higher weight) to augment the original training set;
- The new training set is used to obtain a new hypothesis, etc.

Ensemble Learning: Bagging

- **Bootstrap aggregating** (Bagging) consists in generating different training sets by sampling from the original training set;

Ensemble Learning: Bagging

- **Bootstrap aggregating** (Bagging) consists in generating different training sets by sampling from the original training set;
- Sampling is done uniformly, and with replacement;

Ensemble Learning: Bagging

- **Bootstrap aggregating** (Bagging) consists in generating different training sets by sampling from the original training set;
- Sampling is done uniformly, and with replacement;
- Each of those new training sets (called **bootstrap samples**) are used to produce an hypothesis.

Ensemble Learning: Random subspace

- The **random subspace** method consists in producing many hypotheses using only a subset of the input features for each;

Ensemble Learning: Random subspace

- The **random subspace** method consists in producing many hypotheses using only a subset of the input features for each;
- These subsets are determined randomly for each hypothesis;

Ensemble Learning: Random subspace

- The **random subspace** method consists in producing many hypotheses using only a subset of the input features for each;
- These subsets are determined randomly for each hypothesis;
- The number of features selected can also be random, but is often the same across hypotheses.

Outline

Decision Trees

- Suppose the unknown function works on a **finite** number of **features** (or **attributes**);

Decision Trees

- Suppose the unknown function works on a **finite** number of **features** (or **attributes**);
- We choose trees for hypotheses that test one attribute at each node;

Decision Trees

- Suppose the unknown function works on a **finite** number of **features** (or **attributes**);
- We choose trees for hypotheses that test one attribute at each node;
- tests are for a **finite** number of values and lead to different nodes depending on the value of the attribute;

Decision Trees

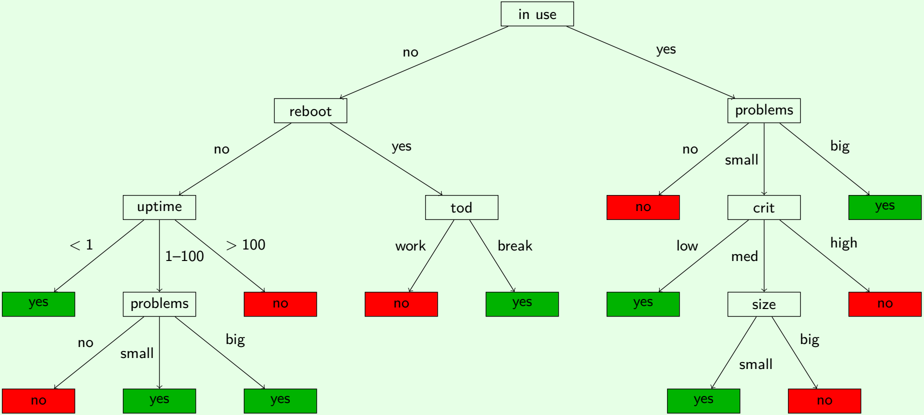
- Suppose the unknown function works on a **finite** number of **features** (or **attributes**);
- We choose trees for hypotheses that test one attribute at each node;
- tests are for a **finite** number of values and lead to different nodes depending on the value of the attribute;
- **Leaves** contain the final output value for the set of attributes (or a function giving the value);

Decision Trees

- Suppose the unknown function works on a **finite** number of **features** (or **attributes**);
- We choose trees for hypotheses that test one attribute at each node;
- tests are for a **finite** number of values and lead to different nodes depending on the value of the attribute;
- **Leaves** contain the final output value for the set of attributes (or a function giving the value);
- We focus on the case where the output is finite (classification), and the possible value ranges of the attributes are given.

Decision Trees

Software update decision



Decision Trees

- Decision trees are fairly **expressive**;

Decision Trees

- Decision trees are fairly **expressive**;
- For instance, they can model exactly **all boolean functions**;

Decision Trees

- Decision trees are fairly **expressive**;
- For instance, they can model exactly **all boolean functions**;
- For boolean functions, they can be "folded" into **binary decision diagrams** (BDDs), a very efficient data structure for symbolic handling of such functions.

Decision Trees

- Decision trees are fairly **expressive**;
- For instance, they can model exactly **all boolean functions**;
- For boolean functions, they can be "folded" into **binary decision diagrams** (BDDs), a very efficient data structure for symbolic handling of such functions.

Exercise

1. Write a decision tree for the **majority function** over three boolean variables (say yes iff two or more variables say yes).
2. How does the size of the tree grow with the (odd) number of variables?

Learning Decision Trees

- Ideally, we want, from a set of examples (values of attributes + corresponding value of the unknown function), to find the **smallest** decision tree representing it.

Learning Decision Trees

- Ideally, we want, from a set of examples (values of attributes + corresponding value of the unknown function), to find the **smallest** decision tree representing it.
- This is not possible in general using brute force (e.g. 2^{2^n} trees for a boolean function on n boolean attributes);

Learning Decision Trees

- Ideally, we want, from a set of examples (values of attributes + corresponding value of the unknown function), to find the **smallest** decision tree representing it.
- This is not possible in general using brute force (e.g. 2^{2^n} trees for a boolean function on n boolean attributes);
- We rely on **heuristics**, giving small (but not the smallest) trees.

Learning Decision Trees: the Greedy Algorithm (ID3)

- Testing an attribute with k values separates the data in k subsets:

in_use	reboot	problems	uptime	tod	crit	size	decision
no	no	big	0	work	high	small	yes
no	no	no	30	break	med	small	no
no	no	small	62	work	low	big	yes
no	no	big	21	work	low	big	yes
no	no	big	221	work	low	big	no
no	yes	small	234	work	med	small	no
no	yes	small	1234	break	med	small	yes
yes	yes	big	42	break	low	small	yes
yes	no	no	78	break	high	big	no
yes	no	small	444	break	low	big	yes
yes	yes	small	1024	work	high	big	no
yes	yes	small	10	work	med	small	yes
yes	no	small	24	work	med	big	no

in_use = no

in_use	reboot	problems	uptime	tod	crit	size	decision
no	no	big	0	work	high	small	yes
no	no	no	30	break	med	small	no
no	no	small	62	work	low	big	yes
no	no	big	21	work	low	big	yes
no	no	big	221	work	low	big	no
no	yes	small	234	work	med	small	no
no	yes	small	1234	break	med	small	yes

in_use = yes

in_use	reboot	problems	uptime	tod	crit	size	decision
yes	yes	big	42	break	low	small	yes
yes	no	no	78	break	high	big	no
yes	no	small	444	break	low	big	yes
yes	yes	small	1024	work	high	big	no
yes	yes	small	10	work	med	small	yes
yes	no	small	24	work	med	big	no

- If a subset contains only yes outputs or only no outputs we can select that output as the result.

Learning Decision Trees: the Greedy Algorithm (ID3)

- The main idea is to always test the **most discriminating** attributes first;

Learning Decision Trees: the Greedy Algorithm (ID3)

- The main idea is to always test the **most discriminating** attributes first;
- Try to **directly answer** (e.g. yes /no) for as many examples as possible;

Learning Decision Trees: the Greedy Algorithm (ID3)

- The main idea is to always test the **most discriminating** attributes first;
- Try to **directly answer** (e.g. yes /no) for as many examples as possible;
- For examples for which we have no direct answer we call the algorithm **recursively**, with less examples to fit, and one less attribute;

Learning Decision Trees: the Greedy Algorithm (ID3)

- The main idea is to always test the **most discriminating** attributes first;
- Try to **directly answer** (e.g. yes /no) for as many examples as possible;
- For examples for which we have no direct answer we call the algorithm **recursively**, with less examples to fit, and one less attribute;
- If we have no more attributes, we return the **plurality value** of the remaining examples (the answer that is the most frequent)
  this means that there are hidden attributes to which we do not have access; or the data is inconsistent (e.g. because of noise)

Learning Decision Trees: the Greedy Algorithm (ID3)

- The main idea is to always test the **most discriminating** attributes first;
- Try to **directly answer** (e.g. yes /no) for as many examples as possible;
- For examples for which we have no direct answer we call the algorithm **recursively**, with less examples to fit, and one less attribute;
- If we have no more attributes, we return the **plurality value** of the remaining examples (the answer that is the most frequent)
  this means that there are hidden attributes to which we do not have access; or the data is inconsistent (e.g. because of noise)
- If we have no more examples, then we return the plurality value of the **parent**
  this means that none of the examples were matching the combination of attributes on that branch

Learning Decision Trees: the Greedy Algorithm (ID3)

- The main idea is to always test the **most discriminating** attributes first;
- Try to **directly answer** (e.g. yes /no) for as many examples as possible;
- For examples for which we have no direct answer we call the algorithm **recursively**, with less examples to fit, and one less attribute;
- If we have no more attributes, we return the **plurality value** of the remaining examples (the answer that is the most frequent)
  this means that there are hidden attributes to which we do not have access; or the data is inconsistent (e.g. because of noise)
- If we have no more examples, then we return the plurality value of the **parent**
  this means that none of the examples were matching the combination of attributes on that branch
- To find the most discriminating attributes, we rely on **information theory**.

Finding Important Attributes: Entropy

- The **entropy** of a random variable V, with possible values v_k, is:

$$H(V) = -\sum_k P(v_k) \log_2 \left(P(v_k)\right)$$

Finding Important Attributes: Entropy

- The **entropy** of a random variable V, with possible values v_k, is:

$$H(V) = -\sum_k P(v_k) \log_2 \left(P(v_k)\right)$$

- It gives a measure of the **uncertainty** on V as the **number of bits** of information obtained from a value of V;

Exercise

What is the entropy of:

Finding Important Attributes: Entropy

- The **entropy** of a random variable V, with possible values v_k, is:

$$H(V) = -\sum_k P(v_k) \log_2 \left(P(v_k)\right)$$

- It gives a measure of the **uncertainty** on V as the **number of bits** of information obtained from a value of V;

Exercise

What is the entropy of:

- A random variable with only one value;

Finding Important Attributes: Entropy

- The **entropy** of a random variable V, with possible values v_k, is:

$$H(V) = -\sum_k P(v_k) \log_2 \left(P(v_k)\right)$$

- It gives a measure of the **uncertainty** on V as the **number of bits** of information obtained from a value of V;

Exercise

What is the entropy of:

- A random variable with only one value;
- A fair coin flip;

Finding Important Attributes: Entropy

- The **entropy** of a random variable V, with possible values v_k, is:

$$H(V) = -\sum_k P(v_k) \log_2 \left(P(v_k)\right)$$

- It gives a measure of the **uncertainty** on V as the **number of bits** of information obtained from a value of V;

Exercise

What is the entropy of:

- A random variable with only one value;
- A fair coin flip;
- A biased coin (with a heads probability of 0.8) flip;

Finding Important Attributes: Entropy

- The **entropy** of a random variable V, with possible values v_k, is:

$$H(V) = -\sum_k P(v_k)\log_2\left(P(v_k)\right)$$

- It gives a measure of the **uncertainty** on V as the **number of bits** of information obtained from a value of V;

Exercise

What is the entropy of:

- A random variable with only one value;
- A fair coin flip;
- A biased coin (with a heads probability of 0.8) flip;
- A 4-sided fair die.

Finding Important Attributes: Entropy

- The set of examples can be seen as a random variable with:

Finding Important Attributes: Entropy

- The set of examples can be seen as a random variable with:
  - values corresponding to the outputs;

Finding Important Attributes: Entropy

- The set of examples can be seen as a random variable with:
  - values corresponding to the outputs;
  - probabilities given by the frequencies of outputs.

Finding Important Attributes: Entropy

- The set of examples can be seen as a random variable with:
  - values corresponding to the outputs;
  - probabilities given by the frequencies of outputs.
- To simplify, we now consider only two possible output values;

Finding Important Attributes: Entropy

- The set of examples can be seen as a random variable with:
  - values corresponding to the outputs;
  - probabilities given by the frequencies of outputs.
- To simplify, we now consider only two possible output values;
- If we have p examples for which the output is "yes" and n for which it is "no", then we can consider the output as a random variable that is true with probability $\frac{p}{p+n}$;

Finding Important Attributes: Entropy

- The set of examples can be seen as a random variable with:
  - values corresponding to the outputs;
  - probabilities given by the frequencies of outputs.
- To simplify, we now consider only two possible output values;
- If we have p examples for which the output is “yes” and n for which it is “no”, then we can consider the output as a random variable that is true with probability $\frac{p}{p+n}$;
- For a random boolean variable that is true with probability q, the entropy is:

$$B(q) = -\big(q\log_2(q) + (1-q)\log_2(1-q)\big)$$

Finding Important Attributes: Entropy

- Testing an attribute A that has d values divides the examples into d subsets $E_1, \ldots, E_d$;

Finding Important Attributes: Entropy

- Testing an attribute A that has d values divides the examples into d subsets $E_1, \ldots, E_d$;
- Each E_k contains p_k examples with output "yes" and n_k with output "no";

Finding Important Attributes: Entropy

- Testing an attribute A that has d values divides the examples into d subsets $E_1, \ldots, E_d$;
- Each E_k contains p_k examples with output "yes" and n_k with output "no";
- The corresponding entropy is $B\left(\frac{p_k}{p_k+n_k}\right)$.

Finding Important Attributes: Entropy

- Testing an attribute A that has d values divides the examples into d subsets $E_1, \ldots, E_d$;
- Each E_k contains p_k examples with output "yes" and n_k with output "no";
- The corresponding entropy is $B\left(\frac{p_k}{p_k+n_k}\right)$.
- The probability of having subset E_k (i.e. that A has the k^{th} value) is $\frac{p_k+n_k}{p+n}$;

Finding Important Attributes: Entropy

- Testing an attribute A that has d values divides the examples into d subsets $E_1, \ldots, E_d$;
- Each E_k contains p_k examples with output "yes" and n_k with output "no";
- The corresponding entropy is $B\left(\frac{p_k}{p_k+n_k}\right)$.
- The probability of having subset E_k (i.e. that A has the k^{th} value) is $\frac{p_k+n_k}{p+n}$;
- So the **expected entropy** after choosing A is

$$R(A) = \sum_{k=1}^{d} \frac{p_k + n_k}{p + n} B\left(\frac{p_k}{p_k + n_k}\right)$$

Finding Important Attributes: Entropy

- Testing an attribute A that has d values divides the examples into d subsets $E_1, \ldots, E_d$;
- Each E_k contains p_k examples with output "yes" and n_k with output "no";
- The corresponding entropy is $B\left(\frac{p_k}{p_k+n_k}\right)$.
- The probability of having subset E_k (i.e. that A has the k^{th} value) is $\frac{p_k+n_k}{p+n}$;
- So the **expected entropy** after choosing A is

$$R(A) = \sum_{k=1}^{d} \frac{p_k + n_k}{p + n} B\left(\frac{p_k}{p_k + n_k}\right)$$

- Since we want to be as certain as possible of the value of the output after choosing A, we want to **minimise** this value.

Finding Important Attributes: Information Gain (Ratio)

- Minimising $R(A)$ corresponds to **maximising** the **information gain** of A, defined as:

$$IG(A) = B\left(\frac{p}{p+n}\right) - R(A)$$

Finding Important Attributes: Information Gain (Ratio)

- Minimising $R(A)$ corresponds to **maximising** the **information gain** of A, defined as:

$$IG(A) = B\left(\frac{p}{p+n}\right) - R(A)$$

- The information gain is not suitable when an attribute may have many different values (e.g. student number);

Finding Important Attributes: Information Gain (Ratio)

- Minimising $R(A)$ corresponds to **maximising** the **information gain** of A, defined as:

$$IG(A) = B\left(\frac{p}{p+n}\right) - R(A)$$

- The information gain is not suitable when an attribute may have many different values (e.g. student number);
- The **intrinsic value** of an attribute is (an estimation of) the entropy of the random variable that gives a value for this attribute:

$$IV(A) = -\sum_{k=1}^{d} \frac{n_k + p_k}{n+p} \log_2\left(\frac{n_k + p_k}{n+p}\right)$$

Finding Important Attributes: Information Gain (Ratio)

- Minimising $R(A)$ corresponds to **maximising** the **information gain** of A, defined as:

$$IG(A) = B\left(\frac{p}{p+n}\right) - R(A)$$

- The information gain is not suitable when an attribute may have many different values (e.g. student number);
- The **intrinsic value** of an attribute is (an estimation of) the entropy of the random variable that gives a value for this attribute:

$$IV(A) = -\sum_{k=1}^{d} \frac{n_k + p_k}{n+p} \log_2\left(\frac{n_k + p_k}{n+p}\right)$$

- The **information gain ratio** corrects the bias of information gain:

$$IGR(A) = \frac{IG(A)}{IV(A)}$$

Finding Important Attributes: Information Gain (Ratio)

Exercise

problems	size	decision
big	small	yes
no	small	no
small	big	yes
big	big	yes
big	big	no
small	small	no
small	small	yes
big	small	yes
no	big	no
small	big	yes
small	big	no
small	small	yes
small	big	no

Compute the information gain ratio for attributes *size* and *problems*.

Generalisation and Overfitting

- The greedy algorithm might infer patterns from noise in the data;

Generalisation and Overfitting

- The greedy algorithm might infer patterns from noise in the data;
- For example, a useless attribute might by chance come up more frequently with a given value;

Generalisation and Overfitting

- The greedy algorithm might infer patterns from noise in the data;
- For example, a useless attribute might by chance come up more frequently with a given value;
- The algorithm will **overfit** by creating a node for this attribute;

Generalisation and Overfitting

- The greedy algorithm might infer patterns from noise in the data;
- For example, a useless attribute might by chance come up more frequently with a given value;
- The algorithm will **overfit** by creating a node for this attribute;
- This impairs generalisation by adding an unnecessary constraint;

Generalisation and Overfitting

- The greedy algorithm might infer patterns from noise in the data;
- For example, a useless attribute might by chance come up more frequently with a given value;
- The algorithm will **overfit** by creating a node for this attribute;
- This impairs generalisation by adding an unnecessary constraint;
- The more training data, the less probable is this problem;

Generalisation and Overfitting

- The greedy algorithm might infer patterns from noise in the data;
- For example, a useless attribute might by chance come up more frequently with a given value;
- The algorithm will **overfit** by creating a node for this attribute;
- This impairs generalisation by adding an unnecessary constraint;
- The more training data, the less probable is this problem;
- Decision tree **pruning** is a technique to address this problem.

Decision tree χ^2 pruning

- Once the tree is built, we perform for some attribute A, with **only leaves** children, a pruning test;

Decision tree χ^2 pruning

- Once the tree is built, we perform for some attribute A, with **only leaves** children, a pruning test;
- If the node is pruned, we replace it by a leaf with the plurality value;

Decision tree χ^2 pruning

- Once the tree is built, we perform for some attribute A, with **only leaves** children, a pruning test;
- If the node is pruned, we replace it by a leaf with the plurality value;
- We then proceed to see if its parent now has only leaves children and can be pruned;

Decision tree χ^2 pruning

- Once the tree is built, we perform for some attribute A, with **only leaves** children, a pruning test;
- If the node is pruned, we replace it by a leaf with the plurality value;
- We then proceed to see if its parent now has only leaves children and can be pruned;
- A common pruning test is a **statistical** test between two hypotheses:

Decision tree χ^2 pruning

- Once the tree is built, we perform for some attribute A, with **only leaves** children, a pruning test;
- If the node is pruned, we replace it by a leaf with the plurality value;
- We then proceed to see if its parent now has only leaves children and can be pruned;
- A common pruning test is a **statistical** test between two hypotheses:
  1. **null hypothesis**: there is no pattern on this attribute;

Decision tree χ^2 pruning

- Once the tree is built, we perform for some attribute A, with **only leaves** children, a pruning test;
- If the node is pruned, we replace it by a leaf with the plurality value;
- We then proceed to see if its parent now has only leaves children and can be pruned;
- A common pruning test is a **statistical** test between two hypotheses:
  1. **null hypothesis**: there is no pattern on this attribute;
  2. there is indeed a pattern on this attribute.

Decision tree χ^2 pruning

- Once the tree is built, we perform for some attribute A, with **only leaves** children, a pruning test;
- If the node is pruned, we replace it by a leaf with the plurality value;
- We then proceed to see if its parent now has only leaves children and can be pruned;
- A common pruning test is a **statistical** test between two hypotheses:
  1. **null hypothesis**: there is no pattern on this attribute;
  2. there is indeed a pattern on this attribute.
- We use the classic framework of the χ^2 **statistical test**.

Decision tree χ^2 pruning

- The null hypothesis means here that: the probability of an example e_i having output true and that of e having value x_k for A are independent:

$$P(e = \text{true}, A = x_k) = P(e = \text{true})P(A = x_k)$$

Decision tree χ^2 pruning

- The null hypothesis means here that: the probability of an example e_i having output true and that of e having value x_k for A are independent:

$$P(e = \text{true}, A = x_k) = P(e = \text{true})P(A = x_k)$$

- We can **estimate** these probabilities from the training set:

Decision tree χ^2 pruning

- The null hypothesis means here that: the probability of an example e_i having output true and that of e having value x_k for A are independent:

$$P(e = \text{true}, A = x_k) = P(e = \text{true})P(A = x_k)$$

- We can **estimate** these probabilities from the training set:
  1. for $P(e = \text{true})$: let $\hat{q} = \frac{p}{n+p}$;

Decision tree χ^2 pruning

- The null hypothesis means here that: the probability of an example e_i having output true and that of e having value x_k for A are independent:

$$P(e = \text{true}, A = x_k) = P(e = \text{true})P(A = x_k)$$

- We can **estimate** these probabilities from the training set:
  1. for $P(e = \text{true})$: let $\hat{q} = \frac{p}{n+p}$;
  2. for $P(A = x_k)$: let $\hat{x}_k = \frac{n_k+p_k}{n+p}$.

Decision tree χ^2 pruning

- The null hypothesis means here that: the probability of an example e_i having output true and that of e having value x_k for A are independent:

$$P(e = \text{true}, A = x_k) = P(e = \text{true})P(A = x_k)$$

- We can **estimate** these probabilities from the training set:
  1. for $P(e = \text{true})$: let $\hat{q} = \frac{p}{n+p}$;
  2. for $P(A = x_k)$: let $\hat{x}_k = \frac{n_k+p_k}{n+p}$.
  3. for $P(e = \text{true}, A = x_k)$: let $\hat{\pi}_k$ be the estimate, we should have $\hat{\pi}_k = \hat{q}\hat{x}_k = p\frac{n_k+p_k}{(n+p)^2}$.

Decision tree χ^2 pruning

- The null hypothesis means here that: the probability of an example e_i having output true and that of e having value x_k for A are independent:

$$P(e = \text{true}, A = x_k) = P(e = \text{true})P(A = x_k)$$

- We can **estimate** these probabilities from the training set:
  1. for $P(e = \text{true})$: let $\hat{q} = \frac{p}{n+p}$;
  2. for $P(A = x_k)$: let $\hat{x}_k = \frac{n_k+p_k}{n+p}$.
  3. for $P(e = \text{true}, A = x_k)$: let $\hat{\pi}_k$ be the estimate, we should have $\hat{\pi}_k = \hat{q}\hat{x}_k = p\frac{n_k+p_k}{(n+p)^2}$.
- Finally, we use the following indicator, to measure the difference between the observed number p_k of examples with attribute value x_k and positive output and the number predicted by the theory $(n+p)P(e = \text{true}, A = x_k)$ estimated as $\hat{p}_k = (n+p)\hat{\pi}_k = p\frac{n_k+p_k}{n+p}$ (and similarly for negative examples: $\hat{n}_k$):

$$\chi^2_{obs} = \sum_k \frac{(p_k - \hat{p}_k)^2}{\hat{p}_k} + \frac{(n_k - \hat{n}_k)^2}{\hat{n}_k}$$

Decision tree χ^2 pruning

- Under the null hypothesis χ^2_{obs} should be distributed according to the χ^2 distribution with $d-1$ degrees of freedom;

Decision tree χ^2 pruning

- Under the null hypothesis χ^2_{obs} should be distributed according to the χ^2 distribution with $d-1$ degrees of freedom;
- By some table lookup (or standard statistical computation), we now can find for any α the value $\chi^2_{d-1,\alpha}$ such that $P(\chi^2_{obs} \geq \chi^2_{d-1,\alpha}) = \alpha$.

Decision tree χ^2 pruning

- Under the null hypothesis χ^2_{obs} should be distributed according to the χ^2 distribution with $d - 1$ degrees of freedom;
- By some table lookup (or standard statistical computation), we now can find for any α the value $\chi^2_{d-1,\alpha}$ such that $P(\chi^2_{obs} \geq \chi^2_{d-1,\alpha}) = \alpha$.
- Hence we **accept** the null hypothesis with confidence α if $\chi^2_{obs} < \chi^2_{d-1,\alpha}$;

Decision tree χ^2 pruning

- Under the null hypothesis χ^2_{obs} should be distributed according to the χ^2 distribution with $d-1$ degrees of freedom;
- By some table lookup (or standard statistical computation), we now can find for any α the value $\chi^2_{d-1,\alpha}$ such that $P(\chi^2_{obs} \geq \chi^2_{d-1,\alpha}) = \alpha$.
- Hence we **accept** the null hypothesis with confidence α if $\chi^2_{obs} < \chi^2_{d-1,\alpha}$;
- In that case, the attribute is useless and the node can be removed.

Decision tree χ^2 pruning

- Under the null hypothesis χ^2_{obs} should be distributed according to the χ^2 distribution with $d-1$ degrees of freedom;
- By some table lookup (or standard statistical computation), we now can find for any α the value $\chi^2_{d-1,\alpha}$ such that $P(\chi^2_{obs} \geq \chi^2_{d-1,\alpha}) = \alpha$.
- Hence we **accept** the null hypothesis with confidence α if $\chi^2_{obs} < \chi^2_{d-1,\alpha}$;
- In that case, the attribute is useless and the node can be removed.

Exercise

Suppose we want to learn the XOR function on two binary inputs x_1 and x_2. Assume that, as a training set we have exactly *a* examples for each combination of the inputs.

1. Compute the resulting decision tree, using the greedy algorithm (without pruning);
2. Can any of the bottom-most non-leaf nodes be pruned?
3. What does the pruning test give for the root of the tree?

Random Forests

- **Random forests** are an ensemble learning method;

Random Forests

- **Random forests** are an ensemble learning method;
- They consists in using **bagging** and the **random subspace** method for decision trees;

Random Forests

- **Random forests** are an ensemble learning method;
- They consists in using **bagging** and the **random subspace** method for decision trees;
- This usually greatly improves the accuracy of decision trees, while remaining quite fast and fairly explainable.

Decision Trees: Conclusion

- Decision trees are a very popular learning model (esp. for medical applications);

Decision Trees: Conclusion

- Decision trees are a very popular learning model (esp. for medical applications);
- They are **fast** to train and use;

Decision Trees: Conclusion

- Decision trees are a very popular learning model (esp. for medical applications);
- They are **fast** to train and use;
- The result is **explainable**;

Decision Trees: Conclusion

- Decision trees are a very popular learning model (esp. for medical applications);
- They are **fast** to train and use;
- The result is **explainable**;
- They can be combined in random forests for increased **accuracy**;

Decision Trees: Conclusion

- Decision trees are a very popular learning model (esp. for medical applications);
- They are **fast** to train and use;
- The result is **explainable**;
- They can be combined in random forests for increased **accuracy**;
- There are additional techniques to:

Decision Trees: Conclusion

- Decision trees are a very popular learning model (esp. for medical applications);
- They are **fast** to train and use;
- The result is **explainable**;
- They can be combined in random forests for increased **accuracy**;
- There are additional techniques to:
  - automatically split numerical attributes into ranges;

Decision Trees: Conclusion

- Decision trees are a very popular learning model (esp. for medical applications);
- They are **fast** to train and use;
- The result is **explainable**;
- They can be combined in random forests for increased **accuracy**;
- There are additional techniques to:
  - automatically split numerical attributes into ranges;
  - handling missing data.

Decision Trees: Conclusion

- Decision trees are a very popular learning model (esp. for medical applications);
- They are **fast** to train and use;
- The result is **explainable**;
- They can be combined in random forests for increased **accuracy**;
- There are additional techniques to:
  - automatically split numerical attributes into ranges;
  - handling missing data.
- We can use linear regression in the leaves to obtain **regression trees**.

Outline

Linear Regression

- We now consider the hypothesis space of **linear functions**:

$$y = w_0 + w_1 x_1 + \cdots + w_n x_n$$

Linear Regression

- We now consider the hypothesis space of **linear functions**:

$$y = w_0 + w_1 x_1 + \cdots + w_n x_n$$

- For n attributes we then have $n + 1$ parameters;

Linear Regression

- We now consider the hypothesis space of **linear functions**:

$$y = w_0 + w_1 x_1 + \cdots + w_n x_n$$

- For n attributes we then have $n+1$ parameters;
- In the **regression** problem we want to find the straight line $h_{\vec{w}}$ that approximates best the set of examples;

Linear Regression

- We now consider the hypothesis space of **linear functions**:

$$y = w_0 + w_1 x_1 + \cdots + w_n x_n$$

- For n attributes we then have $n+1$ parameters;
- In the **regression** problem we want to find the straight line $h_{\vec{w}}$ that approximates best the set of examples;
- Gauss tells us that if the noise on data is normally distributed, then this means minimising the **euclidean distance** between the actual output and the prediction;

Linear Regression

- We now consider the hypothesis space of **linear functions**:

$$y = w_0 + w_1 x_1 + \cdots + w_n x_n$$

- For n attributes we then have $n + 1$ parameters;
- In the **regression** problem we want to find the straight line $h_{\vec{w}}$ that approximates best the set of examples;
- Gauss tells us that if the noise on data is normally distributed, then this means minimising the **euclidean distance** between the actual output and the prediction;
- This means minimising the L_2 loss over the N given examples:

$$Loss(h_{\vec{w}}) = \frac{1}{2}\sum_{k=1}^{N} L_2(y_k, h_{\vec{w}}(\vec{x}_k)) = \frac{1}{2}\sum_{k=1}^{N}(y_k - h_{\vec{w}}(\vec{x}_k))^2$$

Linear Regression

- We now consider the hypothesis space of **linear functions**:

$$y = w_0 + w_1 x_1 + \cdots + w_n x_n$$

- For n attributes we then have $n+1$ parameters;
- In the **regression** problem we want to find the straight line $h_{\vec{w}}$ that approximates best the set of examples;
- Gauss tells us that if the noise on data is normally distributed, then this means minimising the **euclidean distance** between the actual output and the prediction;
- This means minimising the L_2 loss over the N given examples:

$$Loss(h_{\vec{w}}) = \frac{1}{2}\sum_{k=1}^{N} L_2(y_k, h_{\vec{w}}(\vec{x}_k)) = \frac{1}{2}\sum_{k=1}^{N}(y_k - h_{\vec{w}}(\vec{x}_k))^2$$

- We can find the global minimum (the function is convex) by writing that all **partial derivatives are zero**.

Linear Regression: Overfitting and Regularisation

- In dimension > 2, some attributes may exhibit spurious relations in the example set that will be captured by learning (**overfitting**) ;

Linear Regression: Overfitting and Regularisation

- In dimension > 2, some attributes may exhibit spurious relations in the example set that will be captured by learning (**overfitting**) ;
- To mitigate this we use **regularisation** on the loss function;

Linear Regression: Overfitting and Regularisation

- In dimension > 2, some attributes may exhibit spurious relations in the example set that will be captured by learning (**overfitting**) ;
- To mitigate this we use **regularisation** on the loss function;
- We consider regularisation terms of the form:

$$L_q = \sum_{i=0}^{n} |w_i|^q$$

Linear Regression: Overfitting and Regularisation

- In dimension > 2, some attributes may exhibit spurious relations in the example set that will be captured by learning (**overfitting**) ;
- To mitigate this we use **regularisation** on the loss function;
- We consider regularisation terms of the form:

$$L_q = \sum_{i=0}^{n} |w_i|^q$$

- Two interesting subcases are $q = 1$ and $q = 2$;

Linear Regression: Overfitting and Regularisation

- In dimension > 2, some attributes may exhibit spurious relations in the example set that will be captured by learning (**overfitting**) ;
- To mitigate this we use **regularisation** on the loss function;
- We consider regularisation terms of the form:

$$L_q = \sum_{i=0}^{n} |w_i|^q$$

- Two interesting subcases are $q = 1$ and $q = 2$;
- L_1 tends to produce **sparse** hypotheses by setting weights to 0;

Linear Regression: Overfitting and Regularisation

- In dimension > 2, some attributes may exhibit spurious relations in the example set that will be captured by learning (**overfitting**) ;
- To mitigate this we use **regularisation** on the loss function;
- We consider regularisation terms of the form:

$$L_q = \sum_{i=0}^{n} |w_i|^q$$

- Two interesting subcases are $q = 1$ and $q = 2$;
- L_1 tends to produce **sparse** hypotheses by setting weights to 0;
- But it may be less appropriate than L_2 if the choice of the features is a bit arbitrary (e.g. coordinates in a rotationally invariant problem).

Gradient Descent

- With L_1 regularisation the loss function is not **differentiable** anymore;

Gradient Descent

- With L_1 regularisation the loss function is not **differentiable** anymore;
- Instead of computing the solution in closed-form, we can use iterative methods such as **gradient descent**:

Gradient Descent

- With L_1 regularisation the loss function is not **differentiable** anymore;
- Instead of computing the solution in closed-form, we can use iterative methods such as **gradient descent**:
  1. choose a gradient step (called here **learning rate**) $\alpha > 0$;

Gradient Descent

- With L_1 regularisation the loss function is not **differentiable** anymore;
- Instead of computing the solution in closed-form, we can use iterative methods such as **gradient descent**:
  1. choose a gradient step (called here **learning rate**) $\alpha > 0$;
  2. start from an arbitrary point in the parameters space;

Gradient Descent

- With L_1 regularisation the loss function is not **differentiable** anymore;
- Instead of computing the solution in closed-form, we can use iterative methods such as **gradient descent**:
  1. choose a gradient step (called here **learning rate**) $\alpha > 0$;
  2. start from an arbitrary point in the parameters space;
  3. repeat until convergence: find the local gradient $\frac{\partial Loss}{\partial w_i}$ and apply:

$$w_i \leftarrow w_i - \alpha \frac{\partial Loss}{\partial w_i}(\vec{w})$$

Exercise

Instantiate this update rule for a single data point $\vec{x}$ at a point $\vec{w}$, without any regularisation.

Gradient Descent

- Since *Loss* is a big sum on a batch of examples, this is called **batch gradient descent**;

Gradient Descent

- Since *Loss* is a big sum on a batch of examples, this is called **batch gradient descent**;
- Convergence is guaranteed provided that α is small enough, but can be very slow;

Gradient Descent

- Since *Loss* is a big sum on a batch of examples, this is called **batch gradient descent**;
- Convergence is guaranteed provided that α is small enough, but can be very slow;
- We can also **randomly** select one of the examples at each iteration: **stochastic gradient descent**;

Gradient Descent

- Since *Loss* is a big sum on a batch of examples, this is called **batch gradient descent**;
- Convergence is guaranteed provided that α is small enough, but can be very slow;
- We can also **randomly** select one of the examples at each iteration: **stochastic gradient descent**;
- It is usually faster but convergence is not guaranteed, unless we make α decrease at each iteration with

$$\sum_{t=1}^{\infty} \alpha(t) = \infty \text{ and } \sum_{t=1}^{\infty} \alpha(t)^2 < \infty$$

Linear Classification

- We now want to do **classification**: separate all points into two categories 1 and 0;

Linear Classification

- We now want to do **classification**: separate all points into two categories 1 and 0;
- We want the separation to be an **hyperplane** (i.e. defined by a linear equation)

Linear Classification

- We now want to do **classification**: separate all points into two categories 1 and 0;
- We want the separation to be an **hyperplane** (i.e. defined by a linear equation)
- Learning consists in finding $\vec{w}$ such that $\sum_i w_i x_i + w_0 \geq 0$ iff the answer for $\vec{x}$ is 1.

Linear Classification

- We now want to do **classification**: separate all points into two categories 1 and 0;
- We want the separation to be an **hyperplane** (i.e. defined by a linear equation)
- Learning consists in finding $\vec{w}$ such that $\sum_i w_i x_i + w_0 \geq 0$ iff the answer for $\vec{x}$ is 1.
- So an hypothesis has the form $h_{\vec{w}}(\vec{x}) = T(\vec{w} \cdot \vec{x})$, where:

Linear Classification

- We now want to do **classification**: separate all points into two categories 1 and 0;
- We want the separation to be an **hyperplane** (i.e. defined by a linear equation)
- Learning consists in finding $\vec{w}$ such that $\sum_i w_i x_i + w_0 \geq 0$ iff the answer for $\vec{x}$ is 1.
- So an hypothesis has the form $h_{\vec{w}}(\vec{x}) = T(\vec{w} \cdot \vec{x})$, where:
  - $\cdot$ is the dot product;

Linear Classification

- We now want to do **classification**: separate all points into two categories 1 and 0;
- We want the separation to be an **hyperplane** (i.e. defined by a linear equation)
- Learning consists in finding $\vec{w}$ such that $\sum_i w_i x_i + w_0 \geq 0$ iff the answer for $\vec{x}$ is 1.
- So an hypothesis has the form $h_{\vec{w}}(\vec{x}) = T(\vec{w} \cdot \vec{x})$, where:
  - $\cdot$ is the dot product;
  - T is a **threshold function** such $T(z) = 1$ if $z \geq 0$ and $T(z) = 0$ otherwise;

Linear Classification

- We now want to do **classification**: separate all points into two categories 1 and 0;
- We want the separation to be an **hyperplane** (i.e. defined by a linear equation)
- Learning consists in finding $\vec{w}$ such that $\sum_i w_i x_i + w_0 \geq 0$ iff the answer for $\vec{x}$ is 1.
- So an hypothesis has the form $h_{\vec{w}}(\vec{x}) = T(\vec{w} \cdot \vec{x})$, where:
  - $\cdot$ is the dot product;
  - T is a **threshold function** such $T(z) = 1$ if $z \geq 0$ and $T(z) = 0$ otherwise;
  - We assume a dummy coordinate x_0 that is always equal to 1.

Linear Classification: Hard Threshold

- For $h_{\vec{w}}(\vec{x}) = T(\vec{w} \cdot \vec{x})$, the loss function is not differentiable, and the gradient, almost always 0.

Linear Classification: Hard Threshold

- For $h_{\vec{w}}(\vec{x}) = T(\vec{w} \cdot \vec{x})$, the loss function is not differentiable, and the gradient, almost always 0.
- For any example $(\vec{x}_k, y_k)$, intuitively:

Linear Classification: Hard Threshold

- For $h_{\vec{w}}(\vec{x}) = T(\vec{w} \cdot \vec{x})$, the loss function is not differentiable, and the gradient, almost always 0.
- For any example $(\vec{x}_k, y_k)$, intuitively:
  1. if $y = h_{\vec{w}}(\vec{x})$, the weights are fine;

Linear Classification: Hard Threshold

- For $h_{\vec{w}}(\vec{x}) = T(\vec{w} \cdot \vec{x})$, the loss function is not differentiable, and the gradient, almost always 0.
- For any example $(\vec{x}_k, y_k)$, intuitively:
  1. if $y = h_{\vec{w}}(\vec{x})$, the weights are fine;
  2. if y is 1 and $h_{\vec{w}}(\vec{x})$ is 0, we want $h_{\vec{w}}(\vec{x})$ to be bigger, i.e., w_i to be bigger if x_i is positive and smaller otherwise;

Linear Classification: Hard Threshold

- For $h_{\vec{w}}(\vec{x}) = T(\vec{w} \cdot \vec{x})$, the loss function is not differentiable, and the gradient, almost always 0.
- For any example $(\vec{x}_k, y_k)$, intuitively:
  1. if $y = h_{\vec{w}}(\vec{x})$, the weights are fine;
  2. if y is 1 and $h_{\vec{w}}(\vec{x})$ is 0, we want $h_{\vec{w}}(\vec{x})$ to be bigger, i.e., w_i to be bigger if x_i is positive and smaller otherwise;
  3. if y is 0 and $h_{\vec{w}}(\vec{x})$ is 1, we want $h_{\vec{w}}(\vec{x})$ to be smaller.

Linear Classification: Hard Threshold

- For $h_{\vec{w}}(\vec{x}) = T(\vec{w} \cdot \vec{x})$, the loss function is not differentiable, and the gradient, almost always 0.
- For any example $(\vec{x}_k, y_k)$, intuitively:
  1. if $y = h_{\vec{w}}(\vec{x})$, the weights are fine;
  2. if y is 1 and $h_{\vec{w}}(\vec{x})$ is 0, we want $h_{\vec{w}}(\vec{x})$ to be bigger, i.e., w_i to be bigger if x_i is positive and smaller otherwise;
  3. if y is 0 and $h_{\vec{w}}(\vec{x})$ is 1, we want $h_{\vec{w}}(\vec{x})$ to be smaller.
- The **perceptron learning rule** does this:

$$w_i \leftarrow w_i + \alpha(y - h_{\vec{w}}(\vec{x}))x_i$$

Linear Classification: Hard Threshold

- For $h_{\vec{w}}(\vec{x}) = T(\vec{w} \cdot \vec{x})$, the loss function is not differentiable, and the gradient, almost always 0.
- For any example $(\vec{x}_k, y_k)$, intuitively:
  1. if $y = h_{\vec{w}}(\vec{x})$, the weights are fine;
  2. if y is 1 and $h_{\vec{w}}(\vec{x})$ is 0, we want $h_{\vec{w}}(\vec{x})$ to be bigger, i.e., w_i to be bigger if x_i is positive and smaller otherwise;
  3. if y is 0 and $h_{\vec{w}}(\vec{x})$ is 1, we want $h_{\vec{w}}(\vec{x})$ to be smaller.
- The **perceptron learning rule** does this:

$$w_i \leftarrow w_i + \alpha(y - h_{\vec{w}}(\vec{x}))x_i$$

- **Convergence** is guaranteed if the data is **linearly separable**;

Linear Classification: Hard Threshold

- For $h_{\vec{w}}(\vec{x}) = T(\vec{w} \cdot \vec{x})$, the loss function is not differentiable, and the gradient, almost always 0.
- For any example $(\vec{x}_k, y_k)$, intuitively:
  1. if $y = h_{\vec{w}}(\vec{x})$, the weights are fine;
  2. if y is 1 and $h_{\vec{w}}(\vec{x})$ is 0, we want $h_{\vec{w}}(\vec{x})$ to be bigger, i.e., w_i to be bigger if x_i is positive and smaller otherwise;
  3. if y is 0 and $h_{\vec{w}}(\vec{x})$ is 1, we want $h_{\vec{w}}(\vec{x})$ to be smaller.
- The **perceptron learning rule** does this:

$$w_i \leftarrow w_i + \alpha(y - h_{\vec{w}}(\vec{x}))x_i$$

- **Convergence** is guaranteed if the data is **linearly separable**;
- Otherwise we need to make α decrease over iterations (as for stochastic gradient descent).

Linear Classification: Logistic Threshold

- Instead of a hard threshold we can use other **similarly shaped** functions;

Linear Classification: Logistic Threshold

- Instead of a hard threshold we can use other **similarly shaped** functions;
- The **logistic** function (a sigmoid) is a good one:

$$\mathcal{L}(z) = \frac{1}{1 + e^{-z}}$$

Linear Classification: Logistic Threshold

- Instead of a hard threshold we can use other **similarly shaped** functions;
- The **logistic** function (a sigmoid) is a good one:

$$\mathcal{L}(z) = \frac{1}{1 + e^{-z}}$$

- It has values between 0 and 1, so we decide by rounding to the nearest value.

Linear Classification: Logistic Threshold

- Instead of a hard threshold we can use other **similarly shaped** functions;
- The **logistic** function (a sigmoid) is a good one:

$$\mathcal{L}(z) = \frac{1}{1 + e^{-z}}$$

- It has values between 0 and 1, so we decide by rounding to the nearest value.
- It is differentiable;

Linear Classification: Logistic Threshold

- Instead of a hard threshold we can use other **similarly shaped** functions;
- The **logistic** function (a sigmoid) is a good one:

$$\mathcal{L}(z) = \frac{1}{1 + e^{-z}}$$

- It has values between 0 and 1, so we decide by rounding to the nearest value.
- It is differentiable;
- Our model is then:

$$h_{\vec{w}}(\vec{x}) = \mathcal{L}(\vec{w} \cdot \vec{x})$$

Linear Classification: Logistic Threshold

- Instead of a hard threshold we can use other **similarly shaped** functions;
- The **logistic** function (a sigmoid) is a good one:

$$\mathcal{L}(z) = \frac{1}{1 + e^{-z}}$$

- It has values between 0 and 1, so we decide by rounding to the nearest value.
- It is differentiable;
- Our model is then:

$$h_{\vec{w}}(\vec{x}) = \mathcal{L}(\vec{w} \cdot \vec{x})$$

- We can then compute the gradient update for the **least square** (L_2) loss:
  Using $\mathcal{L}'(z) = \mathcal{L}(z)(1 - \mathcal{L}(z))$

$$w_i \leftarrow w_i + \alpha(y - h_{\vec{w}}(\vec{x}))h_{\vec{w}}(\vec{x})(1 - h_{\vec{w}}(\vec{x}))x_i$$

Linear Classification: Cross-entropy loss

- Instead of minimising the L_2 loss function, we can take advantage of the fact that expected answers are 0 or 1;
- The **cross-entropy** loss function is:

$$L(\vec{w}) = -\frac{1}{n}\sum_{i=1}^{n}\Big(y_i \ln\left(h_{\vec{w}}(\vec{x_i})\right) + (1-y_i)\ln\left(1-h_{\vec{w}}(\vec{x_i})\right)\Big)$$

- When $y=1$, for one example $(\vec{x},y)$, we have $L(\vec{w}) = \ln(h_{\vec{w}}(\vec{x}))$, decreasing to 0 when $h_{\vec{w}}(\vec{x})$ goes to 1 and symetrically for $y=0$.
- Also, still for one example $(\vec{x},y)$:

$$\frac{\partial L(\vec{w})}{\partial w_0} = (h_{\vec{w}}(\vec{x}) - y)$$

- And for all $i \neq 0$:

$$\frac{\partial L(\vec{w})}{\partial w_i} = (h_{\vec{w}}(\vec{x}) - y)x_i$$

Linear Classification: Logistic regression

- **Logistic regression** is the (binary) **classification** algorithm obtained by:
  - minimising the cross-entropy loss;
  - on a linear model with a logistic threshold:

$$h_{\vec{w}}(\vec{x}) = \frac{1}{1 + e^{-\vec{w}\cdot\vec{x}}}$$

- Minimisation is typically done using gradient descent:

$$\forall i, w_i \leftarrow w_i + \alpha(y - h_{\vec{w}}(\vec{x}))x_i, \text{ with } x_0 = 1$$

Linear Classification: Exercise

Learning boolean functions

1. Use the perceptron learning rule, with $\alpha = 1$, and starting from $w_0 = 0, w_1 = 0, w_2 = 1$, to compute a linear classifier for the binary OR logical function. You will successively apply each of the following examples, one at a time:

x_1	x_2	$x_1 \vee x_2$
0	0	0
0	1	1
1	0	1
1	1	1
0	0	0

2. Could we learn the majority function similarly? If yes, what weights would define the separation hyperplane?
3. Could we learn the binary XOR similarly?

Linear Classification: Conclusion

- Linear classification is often the **fastest** classification technique;

Linear Classification: Conclusion

- Linear classification is often the **fastest** classification technique;
- Many interesting problems are linearly separable;

Linear Classification: Conclusion

- Linear classification is often the **fastest** classification technique;
- Many interesting problems are linearly separable;
- Instead of computing any hyperplane, we can try to compute the one with the **biggest margin** to the nearest examples: this is the principle of **Support Vector Machines** (SVM), one of the most efficient classification techniques.

Linear Classification: Conclusion

- Linear classification is often the **fastest** classification technique;
- Many interesting problems are linearly separable;
- Instead of computing any hyperplane, we can try to compute the one with the **biggest margin** to the nearest examples: this is the principle of **Support Vector Machines** (SVM), one of the most efficient classification techniques.
- To deal with non-linearity, there are two main ideas:

Linear Classification: Conclusion

- Linear classification is often the **fastest** classification technique;
- Many interesting problems are linearly separable;
- Instead of computing any hyperplane, we can try to compute the one with the **biggest margin** to the nearest examples: this is the principle of **Support Vector Machines** (SVM), one of the most efficient classification techniques.
- To deal with non-linearity, there are two main ideas:
  - The **kernel trick**: consider the data in a higher dimension space. Under suitable conditions, this representation need not be explicitly computed. This is in particular applicable to SVM.

Linear Classification: Conclusion

- Linear classification is often the **fastest** classification technique;
- Many interesting problems are linearly separable;
- Instead of computing any hyperplane, we can try to compute the one with the **biggest margin** to the nearest examples: this is the principle of **Support Vector Machines** (SVM), one of the most efficient classification techniques.
- To deal with non-linearity, there are two main ideas:
  - The **kernel trick**: consider the data in a higher dimension space. Under suitable conditions, this representation need not be explicitly computed. This is in particular applicable to SVM.
  - **Combine** several linear classifiers.

Outline

Artificial Neural Networks (ANN)

- ANN have been proposed by Rosenblatt in 1958, inspired by McCulloch and Pitts in 1943;

Artificial Neural Networks (ANN)

- ANN have been proposed by Rosenblatt in 1958, inspired by McCulloch and Pitts in 1943;
- They are inspired by the **neurons** in the brain;

Artificial Neural Networks (ANN)

- ANN have been proposed by Rosenblatt in 1958, inspired by McCulloch and Pitts in 1943;
- They are inspired by the **neurons** in the brain;
- Artificial neurons have weighted inputs, and threshold-like **activation function** on their sum that produces an output;

Artificial Neural Networks (ANN)

- ANN have been proposed by Rosenblatt in 1958, inspired by McCulloch and Pitts in 1943;
- They are inspired by the **neurons** in the brain;
- Artificial neurons have weighted inputs, and threshold-like **activation function** on their sum that produces an output;
- Each artificial neuron essentially implements a **linear classifier**;

Artificial Neural Networks (ANN)

- ANN have been proposed by Rosenblatt in 1958, inspired by McCulloch and Pitts in 1943;
- They are inspired by the **neurons** in the brain;
- Artificial neurons have weighted inputs, and threshold-like **activation function** on their sum that produces an output;
- Each artificial neuron essentially implements a **linear classifier**;
- An **artificial neural network** connects many artificial neurons together.

Perceptrons

- A linear classifier (esp. with hard threshold) is called a **perceptron**;

Perceptrons

- A linear classifier (esp. with hard threshold) is called a **perceptron**;
- And the corresponding neuron is a perceptron unit;

Perceptrons

- A linear classifier (esp. with hard threshold) is called a **perceptron**;
- And the corresponding neuron is a perceptron unit;
- **Perceptron networks** are ANN with a single layer of neurons: outputs depend directly on inputs;

Perceptrons

- A linear classifier (esp. with hard threshold) is called a **perceptron**;
- And the corresponding neuron is a perceptron unit;
- **Perceptron networks** are ANN with a single layer of neurons: outputs depend directly on inputs;
- They correspond to a collection of unrelated linear classifiers on the same inputs.

Perceptrons

- A linear classifier (esp. with hard threshold) is called a **perceptron**;
- And the corresponding neuron is a perceptron unit;
- **Perceptron networks** are ANN with a single layer of neurons: outputs depend directly on inputs;
- They correspond to a collection of unrelated linear classifiers on the same inputs.
- Since OR and NOT are linearly separable, all boolean functions are representable by chaining several perceptron units together.

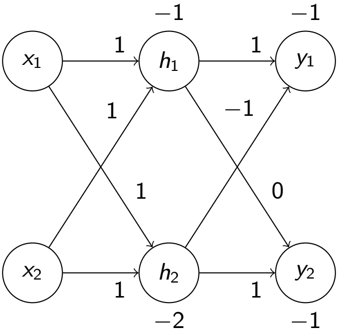
A simple ANN for the addition

Exercise

We consider two boolean inputs $x1$ and $x2$. Using hard threshold functions,

1. Give the weights for a linear classifier computing $x1$ AND $x2$;
2. Give the weights for a linear classifier computing $x1$ AND NOT $x2$;
3. Using the output of OR and AND as inputs to AND NOT, give a 2-layer neural network computing $x1$ XOR $x2$;
4. Add a neuron to compute an addition with carry.

A simple ANN for the addition



Artificial Neural Networks (ANN): Structure

- The arrangement of neurons can be arbitrary;

Artificial Neural Networks (ANN): Structure

- The arrangement of neurons can be arbitrary;
- Two basic structures:

Artificial Neural Networks (ANN): Structure

- The arrangement of neurons can be arbitrary;
- Two basic structures:
  - **recurrent** networks with loops in the structure; They give rise to **complex**, possibly unstable, dynamic systems.

Artificial Neural Networks (ANN): Structure

- The arrangement of neurons can be arbitrary;
- Two basic structures:
  - **recurrent** networks with loops in the structure; They give rise to **complex**, possibly unstable, dynamic systems.
  - **feed-forward** networks where the underlying graph is **acyclic**.

Artificial Neural Networks (ANN): Structure

- The arrangement of neurons can be arbitrary;
- Two basic structures:
  - **recurrent** networks with loops in the structure; They give rise to **complex**, possibly unstable, dynamic systems.
  - **feed-forward** networks where the underlying graph is **acyclic**.
- Classic feed-forward structures use successive **layers** of neurons;

Artificial Neural Networks (ANN): Structure

- The arrangement of neurons can be arbitrary;
- Two basic structures:
  - **recurrent** networks with loops in the structure; They give rise to **complex**, possibly unstable, dynamic systems.
  - **feed-forward** networks where the underlying graph is **acyclic**.
- Classic feed-forward structures use successive **layers** of neurons;
- Neurons in the same layer have no connection between them;

Artificial Neural Networks (ANN): Structure

- The arrangement of neurons can be arbitrary;
- Two basic structures:
  - **recurrent** networks with loops in the structure; They give rise to **complex**, possibly unstable, dynamic systems.
  - **feed-forward** networks where the underlying graph is **acyclic**.
- Classic feed-forward structures use successive **layers** of neurons;
- Neurons in the same layer have no connection between them;
- The inputs can themselves be seen as a layer of neurons with no input, and constant output;

Artificial Neural Networks (ANN): Structure

- The arrangement of neurons can be arbitrary;
- Two basic structures:
  - **recurrent** networks with loops in the structure; They give rise to **complex**, possibly unstable, dynamic systems.
  - **feed-forward** networks where the underlying graph is **acyclic**.
- Classic feed-forward structures use successive **layers** of neurons;
- Neurons in the same layer have no connection between them;
- The inputs can themselves be seen as a layer of neurons with no input, and constant output;
- Neurons with outputs fed into other neurons are called **hidden** and are often arranged in **hidden layers**.

Activation functions

- Many activation functions are possible: they are all mostly threshold-like;

Activation functions

- Many activation functions are possible: they are all mostly threshold-like;
    - hard threshold;

Activation functions

- Many activation functions are possible: they are all mostly threshold-like;
  - hard threshold;
  - logistic;

Activation functions

- Many activation functions are possible: they are all mostly threshold-like;
  - hard threshold;
  - logistic;
  - hyperbolic tangent: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$;

Activation functions

- Many activation functions are possible: they are all mostly threshold-like;
  - hard threshold;
  - logistic;
  - hyperbolic tangent: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$;
  - identity;

Activation functions

- Many activation functions are possible: they are all mostly threshold-like;
  - hard threshold;
  - logistic;
  - hyperbolic tangent: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$;
  - identity;
  - rectified linear unit (ReLU): $ReLU(x) = max(0, x)$;

Activation functions

- Many activation functions are possible: they are all mostly threshold-like;
  - hard threshold;
  - logistic;
  - hyperbolic tangent: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$;
  - identity;
  - rectified linear unit (ReLU): $ReLU(x) = max(0, x)$;
  - etc.

Activation functions

- Many activation functions are possible: they are all mostly threshold-like;
  - hard threshold;
  - logistic;
  - hyperbolic tangent: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$;
  - identity;
  - rectified linear unit (ReLU): $ReLU(x) = max(0, x)$;
  - etc.
- logistic and tanh are most widely-used in shallow networks;

Activation functions

- Many activation functions are possible: they are all mostly threshold-like;
  - hard threshold;
  - logistic;
  - hyperbolic tangent: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$;
  - identity;
  - rectified linear unit (ReLU): $ReLU(x) = max(0, x)$;
  - etc.
- logistic and tanh are most widely-used in shallow networks;
- ReLU are widely used in deep networks.

Multilayer ANN: Forward Response Computation

- Let g be the **activation function**;

Multilayer ANN: Forward Response Computation

- Let g be the **activation function**;
- We consider L fully interconnected layers (we can always put a constant 0 weight if needed)
  So the k-th input to any neuron in layer ℓ is the k-th neuron is layer $\ell - 1$

Multilayer ANN: Forward Response Computation

- Let g be the **activation function**;
- We consider L fully interconnected layers (we can always put a constant 0 weight if needed)

  So the k-th input to any neuron in layer ℓ is the k-th neuron is layer $\ell - 1$
- We denote by w_{jk}^{ℓ} the weight of the k-th input to the j-th neuron in layer ℓ

Multilayer ANN: Forward Response Computation

- Let g be the **activation function**;
- We consider L fully interconnected layers (we can always put a constant 0 weight if needed)
  So the k-th input to any neuron in layer ℓ is the k-th neuron is layer $\ell - 1$
- We denote by w^{ℓ}_{jk} the weight of the k-th input to the j-th neuron in layer ℓ
- We denote by b^{ℓ}_{j} the **bias** of the j-th neuron in layer ℓ;

Multilayer ANN: Forward Response Computation

- Let g be the **activation function**;
- We consider L fully interconnected layers (we can always put a constant 0 weight if needed)
  So the k-th input to any neuron in layer ℓ is the k-th neuron is layer $\ell - 1$
- We denote by w_{jk}^{ℓ} the weight of the k-th input to the j-th neuron in layer ℓ
- We denote by b_j^{ℓ} the **bias** of the j-th neuron in layer ℓ;
- z_j^{ℓ} denotes the weighted input of j-th neuron in layer ℓ and a_j^{ℓ} its output (or activation):

$$z_j^{\ell} = \sum_k w_{jk}^{\ell} a_k^{\ell-1} + b_j^{\ell} \text{ and } a_j^{\ell} = g(z_j^{\ell})$$

Forward Response: Matrix Form

- $z^\ell = [z_j^\ell]_j$ and $a^\ell = [a_j^\ell]_j$ are the column vectors of (resp.) weighted input and activation of layer ℓ;
- $w^\ell = [w_{jk}^\ell]_{jk}$ is the matrix of weights between layer ℓ and $\ell - 1$;
- $b^\ell = [b_j^\ell]_j$ is the column vector of biases of layer ℓ;
- a^0 is the column vector of **inputs**.
- Then, for $\ell \geq 1$:

$$\boxed{z^\ell = w^\ell a^{\ell-1} + b^\ell \text{ and } a^\ell = g(z^\ell)}$$

- The application of g is pointwise.

Forward Response: Matrix Form

- $z^\ell = [z_j^\ell]_j$ and $a^\ell = [a_j^\ell]_j$ are the column vectors of (resp.) weighted input and activation of layer ℓ;
- $w^\ell = [w_{jk}^\ell]_{jk}$ is the matrix of weights between layer ℓ and $\ell - 1$;
- $b^\ell = [b_j^\ell]_j$ is the column vector of biases of layer ℓ;
- a^0 is the column vector of **inputs**.
- Then, for $\ell \geq 1$:

$$\boxed{z^\ell = w^\ell a^{\ell-1} + b^\ell \text{ and } a^\ell = g(z^\ell)}$$

- The application of g is pointwise.

Exercise

1. Write the weight and bias matrices for the previous addition network;
2. Check on a few inputs that the result is the one expected.

Learning in Multilayer ANN

- When computing the answer of the network for a particular input, we go forward;
- Learning means (iteratively) updating the weights to optimise some loss function:
    1. Compute the answer of the network up to the output layer;
    2. Compute the error at the output layer;
    3. **Back-propagate** the error down to the first hidden layer;
    4. Compute the gradient wrt. each weight based on the error;
    5. Update the weights based on the gradient.

Output Layer Error

- We assume the loss function is **additive** across the component of the vector of outputs of the network: this is e.g. the case for the L_2 loss:

$$L_2(\vec{w}, \vec{b}) = \frac{1}{2}||\vec{y} - \vec{h}_{\vec{w},\vec{b}}(\vec{x})||_2^2 = \frac{1}{2}\sum_k (y_k - a_k^L)^2$$

Output Layer Error

- We assume the loss function is **additive** across the component of the vector of outputs of the network: this is e.g. the case for the L_2 loss:

$$L_2(\vec{w}, \vec{b}) = \frac{1}{2}||\vec{y} - \vec{h}_{\vec{w},\vec{b}}(\vec{x})||_2^2 = \frac{1}{2}\sum_k (y_k - a_k^L)^2$$

- We can then compute the gradients for all the outputs separately;

Output Layer Error

- We assume the loss function is **additive** across the component of the vector of outputs of the network: this is e.g. the case for the L_2 loss:

$$L_2(\vec{w}, \vec{b}) = \frac{1}{2}||\vec{y} - \vec{h}_{\vec{w},\vec{b}}(\vec{x})||_2^2 = \frac{1}{2}\sum_k (y_k - a_k^L)^2$$

- We can then compute the gradients for all the outputs separately;
- The **error** on the j-th output neuron is $\frac{\partial Loss}{\partial a_j^L}$. For the L_2 loss:

$$\frac{\partial L_2}{\partial a_j^L} = (a_j - y_j)$$

Output Layer Error

- We assume the loss function is **additive** across the component of the vector of outputs of the network: this is e.g. the case for the L_2 loss:

$$L_2(\vec{w}, \vec{b}) = \frac{1}{2}||\vec{y} - \vec{h}_{\vec{w},\vec{b}}(\vec{x})||_2^2 = \frac{1}{2}\sum_k (y_k - a_k^L)^2$$

- We can then compute the gradients for all the outputs separately;
- The **error** on the j-th output neuron is $\frac{\partial Loss}{\partial a_j^L}$. For the L_2 loss:

$$\frac{\partial L_2}{\partial a_j^L} = (a_j - y_j)$$

- It will be more convenient to consider the **modified error**:

$$\Delta_j^L = \frac{\partial Loss}{\partial z_j^L} = g'(z_j^L)\frac{\partial Loss}{\partial a_j^L}$$

Updating the Weights

- For the output layer, at some point $(\vec{w}, \vec{b})$:

$$\frac{\partial Loss}{\partial w_{jk}^L} = \frac{\partial Loss}{\partial z_j^L}\frac{\partial z_j^L}{\partial w_{jk}^L} = \Delta_j^L \frac{\partial\left(\sum_i w_{ji}^L a_i^{L-1} + b_i^L\right)}{\partial w_{jk}^L} = \Delta_j^L a_k^{L-1}$$

Updating the Weights

- For the output layer, at some point $(\vec{w}, \vec{b})$:

$$\frac{\partial Loss}{\partial w_{jk}^L} = \frac{\partial Loss}{\partial z_j^L}\frac{\partial z_j^L}{\partial w_{jk}^L} = \Delta_j^L \frac{\partial\left(\sum_i w_{ji}^L a_i^{L-1} + b_i^L\right)}{\partial w_{jk}^L} = \Delta_j^L a_k^{L-1}$$

- And for the biases:

$$\frac{\partial Loss}{\partial b_j^L} = \frac{\partial Loss}{\partial z_j^L}\frac{\partial z_j^L}{\partial b_j^L} = \Delta_j^L \frac{\partial\left(\sum_i w_{ji}^L a_i^{L-1} + b_i^L\right)}{\partial b_j^L} = \Delta_j^L$$

Updating the Weights

- For the output layer, at some point $(\vec{w}, \vec{b})$:

$$\frac{\partial Loss}{\partial w_{jk}^L} = \frac{\partial Loss}{\partial z_j^L}\frac{\partial z_j^L}{\partial w_{jk}^L} = \Delta_j^L \frac{\partial\left(\sum_i w_{ji}^L a_i^{L-1} + b_i^L\right)}{\partial w_{jk}^L} = \Delta_j^L a_k^{L-1}$$

- And for the biases:

$$\frac{\partial Loss}{\partial b_j^L} = \frac{\partial Loss}{\partial z_j^L}\frac{\partial z_j^L}{\partial b_j^L} = \Delta_j^L \frac{\partial\left(\sum_i w_{ji}^L a_i^{L-1} + b_i^L\right)}{\partial b_j^L} = \Delta_j^L$$

- We define similarly the (modified) error at neuron k in any (non input) layer ℓ as:

$$\Delta_k^\ell = \frac{\partial Loss}{\partial z_k^\ell}$$

Updating the Weights

- For the output layer, at some point $(\vec{w}, \vec{b})$:

$$\frac{\partial Loss}{\partial w_{jk}^L} = \frac{\partial Loss}{\partial z_j^L}\frac{\partial z_j^L}{\partial w_{jk}^L} = \Delta_j^L \frac{\partial\left(\sum_i w_{ji}^L a_i^{L-1} + b_i^L\right)}{\partial w_{jk}^L} = \Delta_j^L a_k^{L-1}$$

- And for the biases:

$$\frac{\partial Loss}{\partial b_j^L} = \frac{\partial Loss}{\partial z_j^L}\frac{\partial z_j^L}{\partial b_j^L} = \Delta_j^L \frac{\partial\left(\sum_i w_{ji}^L a_i^{L-1} + b_i^L\right)}{\partial b_j^L} = \Delta_j^L$$

- We define similarly the (modified) error at neuron k in any (non input) layer ℓ as:

$$\Delta_k^\ell = \frac{\partial Loss}{\partial z_k^\ell}$$

- Then the previous computation holds and:

$$\frac{\partial Loss}{\partial w_{jk}^\ell} = \Delta_j^\ell a_k^{\ell-1} \text{ and } \frac{\partial Loss}{\partial b_j^\ell} = \Delta_j^\ell$$

Backpropagation

- We compute Δ_k^ℓ by **backpropagating** the output layer error to layer ℓ.

$$\Delta_k^\ell = \frac{\partial Loss}{\partial z_k^\ell}$$

Backpropagation

- We compute Δ_k^ℓ by **backpropagating** the output layer error to layer ℓ.

$$\Delta_k^\ell = \frac{\partial Loss}{\partial z_k^\ell}$$

- *Loss* is a function of all $z_j^{\ell+1}$, each of which is a function of z_k^ℓ:

$$\Delta_k^\ell = \sum_j \frac{\partial Loss}{\partial z_j^{\ell+1}} \frac{\partial z_j^{\ell+1}}{\partial z_k^\ell} = \sum_j \Delta_j^{\ell+1} \frac{\partial \left(\sum_i w_{ji}^{\ell+1} g(z_i^\ell) + b_j^{\ell+1} \right)}{\partial z_k^\ell}$$

Backpropagation

- We compute Δ_k^ℓ by **backpropagating** the output layer error to layer ℓ.

$$\Delta_k^\ell = \frac{\partial Loss}{\partial z_k^\ell}$$

- *Loss* is a function of all $z_j^{\ell+1}$, each of which is a function of z_k^ℓ:

$$\Delta_k^\ell = \sum_j \frac{\partial Loss}{\partial z_j^{\ell+1}} \frac{\partial z_j^{\ell+1}}{\partial z_k^\ell} = \sum_j \Delta_j^{\ell+1} \frac{\partial \left(\sum_i w_{ji}^{\ell+1} g(z_i^\ell) + b_j^{\ell+1} \right)}{\partial z_k^\ell}$$

- And finally:

$$\Delta_k^\ell = \sum_j g'(z_k^\ell) w_{jk}^{\ell+1} \Delta_j^{\ell+1}$$

Backpropagation: Matrix Form

- As for computing forward the response, back-propagation can be done efficiently using matrix operations:

Backpropagation: Matrix Form

- As for computing forward the response, back-propagation can be done efficiently using matrix operations:
    - We need all weighted input vectors z^ℓ and activation vectors a^ℓ computed with the response of the network;

Backpropagation: Matrix Form

- As for computing forward the response, back-propagation can be done efficiently using matrix operations:
  - We need all weighted input vectors z^ℓ and activation vectors a^ℓ computed with the response of the network;
  - To compute the output layer error vector ($\circ$ is the Hadamard product, $\nabla_a Loss$ is the vector of $\frac{\partial Loss}{\partial a_j^L}$'s):

$$\boxed{\Delta^L = \nabla_a Loss \circ g'(z^L)}$$

Backpropagation: Matrix Form

- As for computing forward the response, back-propagation can be done efficiently using matrix operations:
  - We need all weighted input vectors z^ℓ and activation vectors a^ℓ computed with the response of the network;
  - To compute the output layer error vector ($\circ$ is the Hadamard product, $\nabla_a Loss$ is the vector of $\frac{\partial Loss}{\partial a_j^L}$'s):

$$\boxed{\Delta^L = \nabla_a Loss \circ g'(z^L)}$$

  - To backpropagate the error:

$$\boxed{\Delta^{\ell-1} = (w^\ell)^\mathsf{T} \Delta^\ell \circ g'(z^{\ell-1})}$$

Backpropagation: Matrix Form

- As for computing forward the response, back-propagation can be done efficiently using matrix operations:
  - We need all weighted input vectors z^ℓ and activation vectors a^ℓ computed with the response of the network;
  - To compute the output layer error vector ($\circ$ is the Hadamard product, $\nabla_a Loss$ is the vector of $\frac{\partial Loss}{\partial a_j^L}$'s):

$$\boxed{\Delta^L = \nabla_a Loss \circ g'(z^L)}$$

  - To backpropagate the error:

$$\boxed{\Delta^{\ell-1} = (w^\ell)^\mathsf{T} \Delta^\ell \circ g'(z^{\ell-1})}$$

  - To update the weights:

$$w^\ell \leftarrow w^\ell - \alpha \Delta^\ell (a^{\ell-1})^\mathsf{T}$$

Backpropagation: Matrix Form

- As for computing forward the response, back-propagation can be done efficiently using matrix operations:
  - We need all weighted input vectors z^ℓ and activation vectors a^ℓ computed with the response of the network;
  - To compute the output layer error vector ($\circ$ is the Hadamard product, $\nabla_a Loss$ is the vector of $\frac{\partial Loss}{\partial a_j^L}$'s):

$$\boxed{\Delta^L = \nabla_a Loss \circ g'(z^L)}$$

  - To backpropagate the error:

$$\boxed{\Delta^{\ell-1} = (w^\ell)^\mathsf{T} \Delta^\ell \circ g'(z^{\ell-1})}$$

  - To update the weights:

$$w^\ell \leftarrow w^\ell - \alpha \Delta^\ell (a^{\ell-1})^\mathsf{T}$$

  - To update the biases:

$$b^\ell \leftarrow b^\ell - \alpha \Delta^\ell$$

Backpropagation: Exercise

Exercise

For the previous addition network, but with logistic activation functions, and starting from null biases and matrices

$$w^1 = \begin{bmatrix} 0.9 & 1.1 \\ 0.8 & 0.9 \end{bmatrix} \text{ and } w^2 = \begin{bmatrix} 1.2 & -1.1 \\ 0 & 0.9 \end{bmatrix}$$

compute the modified errors and the new values of the weights and biases for input $[1\ 1]^T$ and expected answer $[0\ 1]^T$. Use $\alpha = 0.1$ and an L_2 loss function.

Stochastic Gradient Descent and Minibatches

- As in the linear case, we can use backpropagation to do **gradient descent**;

Stochastic Gradient Descent and Minibatches

- As in the linear case, we can use backpropagation to do **gradient descent**;
- **Stochastic Gradient Descent** (SGD) is the preferred using **epochs** of training:

Stochastic Gradient Descent and Minibatches

- As in the linear case, we can use backpropagation to do **gradient descent**;
- **Stochastic Gradient Descent** (SGD) is the preferred using **epochs** of training:
    - select a random subset X of the examples (called **minibatch**) and substract it from the training set;

Stochastic Gradient Descent and Minibatches

- As in the linear case, we can use backpropagation to do **gradient descent**;
- **Stochastic Gradient Descent** (SGD) is the preferred using **epochs** of training:
  - select a random subset X of the examples (called **minibatch**) and substract it from the training set;
  - train the network using X to estimate the gradient (average over X);

Stochastic Gradient Descent and Minibatches

- As in the linear case, we can use backpropagation to do **gradient descent**;
- **Stochastic Gradient Descent** (SGD) is the preferred using **epochs** of training:
  - select a random subset X of the examples (called **minibatch**) and substract it from the training set;
  - train the network using X to estimate the gradient (average over X);
  - repeat until there are no more examples left.

Stochastic Gradient Descent and Minibatches

- As in the linear case, we can use backpropagation to do **gradient descent**;
- **Stochastic Gradient Descent** (SGD) is the preferred using **epochs** of training:
  - select a random subset X of the examples (called **minibatch**) and substract it from the training set;
  - train the network using X to estimate the gradient (average over X);
  - repeat until there are no more examples left.
- Once an epoch is over, we can start another one, and so one until convergence.

Stochastic Gradient Descent and Minibatches

- Suppose that a^ℓ is now a **matrix** in which each column is the activation at layer ℓ for one example in the random subset (idem for z^ℓ, etc.)

Stochastic Gradient Descent and Minibatches

- Suppose that a^ℓ is now a **matrix** in which each column is the activation at layer ℓ for one example in the random subset (idem for z^ℓ, etc.)
- Then all previous formulas still hold, except for weight update;

Stochastic Gradient Descent and Minibatches

- Suppose that a^ℓ is now a **matrix** in which each column is the activation at layer ℓ for one example in the random subset (idem for z^ℓ, etc.)
- Then all previous formulas still hold, except for weight update;
- To update the weights we average the gradient over the m examples in the subset ($\Delta^{\ell,x}$ is the x-th column of Δ^ℓ, etc.):

$$w^\ell \leftarrow w^\ell - \alpha \times \frac{1}{m} \sum_x \Delta^{\ell,x} (a^{\ell-1,x})^\mathsf{T}$$

Stochastic Gradient Descent and Minibatches

- Suppose that a^ℓ is now a **matrix** in which each column is the activation at layer ℓ for one example in the random subset (idem for z^ℓ, etc.)
- Then all previous formulas still hold, except for weight update;
- To update the weights we average the gradient over the m examples in the subset ($\Delta^{\ell,x}$ is the x-th column of Δ^ℓ, etc.):

$$w^\ell \leftarrow w^\ell - \alpha \times \frac{1}{m} \sum_x \Delta^{\ell,x} (a^{\ell-1,x})^{\mathsf{T}}$$

- Similarly for biases:

$$b^\ell \leftarrow b^\ell - \alpha \times \frac{1}{m} \sum_x \Delta^{\ell,x}$$

Stochastic Gradient Descent and Minibatches

- Suppose that a^ℓ is now a **matrix** in which each column is the activation at layer ℓ for one example in the random subset (idem for z^ℓ, etc.)
- Then all previous formulas still hold, except for weight update;
- To update the weights we average the gradient over the m examples in the subset ($\Delta^{\ell,x}$ is the x-th column of Δ^ℓ, etc.):

$$w^\ell \leftarrow w^\ell - \alpha \times \frac{1}{m} \sum_x \Delta^{\ell,x} (a^{\ell-1,x})^\mathsf{T}$$

- Similarly for biases:

$$b^\ell \leftarrow b^\ell - \alpha \times \frac{1}{m} \sum_x \Delta^{\ell,x}$$

- Or simply:

$$\boxed{w^\ell \leftarrow w^\ell - \tfrac{\alpha}{m} \Delta^\ell (a^{\ell-1})^\mathsf{T}}$$

Stochastic Gradient Descent and Minibatches

- Suppose that a^ℓ is now a **matrix** in which each column is the activation at layer ℓ for one example in the random subset (idem for z^ℓ, etc.)
- Then all previous formulas still hold, except for weight update;
- To update the weights we average the gradient over the m examples in the subset ($\Delta^{\ell,x}$ is the x-th column of Δ^ℓ, etc.):

$$w^\ell \leftarrow w^\ell - \alpha \times \frac{1}{m} \sum_x \Delta^{\ell,x} (a^{\ell-1,x})^\mathsf{T}$$

- Similarly for biases:

$$b^\ell \leftarrow b^\ell - \alpha \times \frac{1}{m} \sum_x \Delta^{\ell,x}$$

- Or simply:

$$\boxed{w^\ell \leftarrow w^\ell - \tfrac{\alpha}{m} \Delta^\ell (a^{\ell-1})^\mathsf{T}}$$

- And:

$$\boxed{b^\ell \leftarrow b^\ell - \tfrac{\alpha}{m} \Delta^\ell \vec{1}}$$

Neuron Saturation

- Since we start with random weights, it is possible that weighted inputs to neurons are very big or very small at some point;

Neuron Saturation

- Since we start with random weights, it is possible that weighted inputs to neurons are very big or very small at some point;
- This means close to the “stable” regions of a threshold-like function;

Neuron Saturation

- Since we start with random weights, it is possible that weighted inputs to neurons are very big or very small at some point;
- This means close to the “stable” regions of a threshold-like function;
- This implies that in such a case the derivative is **almost zero**;

Neuron Saturation

- Since we start with random weights, it is possible that weighted inputs to neurons are very big or very small at some point;
- This means close to the “stable” regions of a threshold-like function;
- This implies that in such a case the derivative is **almost zero**;
- For instance, the widely-used logistic function (sigmoid) satisfies:

$$\mathcal{L}'(z) = \mathcal{L}(z)(1 - \mathcal{L}(z))$$

Neuron Saturation

- Since we start with random weights, it is possible that weighted inputs to neurons are very big or very small at some point;
- This means close to the "stable" regions of a threshold-like function;
- This implies that in such a case the derivative is **almost zero**;
- For instance, the widely-used logistic function (sigmoid) satisfies:

$$\mathcal{L}'(z) = \mathcal{L}(z)(1 - \mathcal{L}(z))$$

- This means that the modified error and the **gradient** will be **very small**
  Even if the output is not at all correct!

Neuron Saturation and Weight Initialisation

- Weights should (in hidden layers) should not be initialised all to zero, in order to break symmetry;

Neuron Saturation and Weight Initialisation

- Weights should (in hidden layers) should not be initialised all to zero, in order to break symmetry;
- An easy way to do is to draw them from **independent gaussian** distributions, with mean 0 and standard deviation 1;

Neuron Saturation and Weight Initialisation

- Weights should (in hidden layers) should not be initialised all to zero, in order to break symmetry;
- An easy way to do is to draw them from **independent gaussian** distributions, with mean 0 and standard deviation 1;
- If a neuron has many inputs, the probability that it starts saturated is not negligible with this scheme;

Neuron Saturation and Weight Initialisation

- Weights should (in hidden layers) should not be initialised all to zero, in order to break symmetry;
- An easy way to do is to draw them from **independent gaussian** distributions, with mean 0 and standard deviation 1;
- If a neuron has many inputs, the probability that it starts saturated is not negligible with this scheme;
- So a standard deviation of $\frac{1}{\sqrt{n}}$ is better, with n the number of inputs.

Cross-entropy and the Logistic Function

- For an output layer with **logistic** neurons, we can design *Loss* to eliminate saturation;

Cross-entropy and the Logistic Function

- For an output layer with **logistic** neurons, we can design *Loss* to eliminate saturation;
- Recall that:

$$\Delta_j^L = \mathcal{L}'(z_j^L)\frac{\partial Loss}{\partial a_j^L}$$

Cross-entropy and the Logistic Function

- For an output layer with **logistic** neurons, we can design *Loss* to eliminate saturation;
- Recall that:

$$\Delta_j^L = \mathcal{L}'(z_j^L)\frac{\partial Loss}{\partial a_j^L}$$

- We would thus like (for a single output):

$$\frac{\partial Loss_j}{\partial a_j^L} = \frac{a_j^L - y_j}{\mathcal{L}'(z_j^L)} = \frac{a_j^L - y_j}{\mathcal{L}(z_j^L)(1 - \mathcal{L}(z_j^L))} = \frac{a_j^L - y_j}{a_j^L(1 - a_j^L)}$$

Cross-entropy and the Logistic Function

- For an output layer with **logistic** neurons, we can design *Loss* to eliminate saturation;
- Recall that:

$$\Delta_j^L = \mathcal{L}'(z_j^L)\frac{\partial Loss}{\partial a_j^L}$$

- We would thus like (for a single output):

$$\frac{\partial Loss_j}{\partial a_j^L} = \frac{a_j^L - y_j}{\mathcal{L}'(z_j^L)} = \frac{a_j^L - y_j}{\mathcal{L}(z_j^L)(1-\mathcal{L}(z_j^L))} = \frac{a_j^L - y_j}{a_j^L(1-a_j^L)}$$

- Integrating wrt. a_j^L (up to a constant):

$$Loss_j = -y_j \log(a_j^L) - (1-y_j)\log(1-a_j^L)$$

Cross-entropy and the Logistic Function

- For an output layer with **logistic** neurons, we can design *Loss* to eliminate saturation;
- Recall that:

$$\Delta_j^L = \mathcal{L}'(z_j^L)\frac{\partial Loss}{\partial a_j^L}$$

- We would thus like (for a single output):

$$\frac{\partial Loss_j}{\partial a_j^L} = \frac{a_j^L - y_j}{\mathcal{L}'(z_j^L)} = \frac{a_j^L - y_j}{\mathcal{L}(z_j^L)(1-\mathcal{L}(z_j^L))} = \frac{a_j^L - y_j}{a_j^L(1-a_j^L)}$$

- Integrating wrt. a_j^L (up to a constant):

$$Loss_j = -y_j \log(a_j^L) - (1-y_j)\log(1-a_j^L)$$

- And over all outputs:

$$Loss = -\sum_j y_j \log(a_j^L) + (1-y_j)\log(1-a_j^L)$$

Cross-entropy and the Logistic Function

- This loss function is the **cross-entropy** loss function;

Cross-entropy and the Logistic Function

- This loss function is the **cross-entropy** loss function;
- It is always **non-negative**;

Cross-entropy and the Logistic Function

- This loss function is the **cross-entropy** loss function;
- It is always **non-negative**;
- It is **close to zero** when a^L is close to y
  Recall that y and a^L are vectors of numbers between 0 and 1;

Cross-entropy and the Logistic Function

- This loss function is the **cross-entropy** loss function;
- It is always **non-negative**;
- It is **close to zero** when a^L is close to y
  Recall that y and a^L are vectors of numbers between 0 and 1;
- By construction:

$$\boxed{\Delta^L = a^L - y}$$

Cross-entropy and the Logistic Function

- This loss function is the **cross-entropy** loss function;
- It is always **non-negative**;
- It is **close to zero** when a^L is close to y
  Recall that y and a^L are vectors of numbers between 0 and 1;
- By construction:

$$\boxed{\Delta^L = a^L - y}$$

- It is generally a better choice than L_2 loss if the the output layer has logistic neurons.

The Softmax Function

- Suppose we would like the output layer to give a **probability distribution**;

The Softmax Function

- Suppose we would like the output layer to give a **probability distribution**;
- For a binary outcome, we can use a logistic output neuron as before;

The Softmax Function

- Suppose we would like the output layer to give a **probability distribution**;
- For a binary outcome, we can use a logistic output neuron as before;
- Otherwise, we use several neurons, each representing an outcome and its activation representing the probability of that outcome;

The Softmax Function

- Suppose we would like the output layer to give a **probability distribution**;
- For a binary outcome, we can use a logistic output neuron as before;
- Otherwise, we use several neurons, each representing an outcome and its activation representing the probability of that outcome;
- We can then use the **softmax** activation function:

$$\mathcal{S}(\vec{z})_i = \frac{e^{z_i}}{\sum_k e^{z_k}}$$

The Softmax Function

- Suppose we would like the output layer to give a **probability distribution**;
- For a binary outcome, we can use a logistic output neuron as before;
- Otherwise, we use several neurons, each representing an outcome and its activation representing the probability of that outcome;
- We can then use the **softmax** activation function:

$$\mathcal{S}(\vec{z})_i = \frac{e^{z_i}}{\sum_k e^{z_k}}$$

- It takes into account the weighted input of **all** the neurons in the layer:

The Softmax Function

- Suppose we would like the output layer to give a **probability distribution**;
- For a binary outcome, we can use a logistic output neuron as before;
- Otherwise, we use several neurons, each representing an outcome and its activation representing the probability of that outcome;
- We can then use the **softmax** activation function:

$$\mathcal{S}(\vec{z})_i = \frac{e^{z_i}}{\sum_k e^{z_k}}$$

- It takes into account the weighted input of **all** the neurons in the layer:
- It has values between 0 and 1 and the sum of values across all the output neurons is 1.

The Softmax Function

- We can again use **cross-entropy** with softmax (in a slightly different way):

$$Loss = -\sum_i y_i \log(a_i^L)$$

The Softmax Function

- We can again use **cross-entropy** with softmax (in a slightly different way):

$$Loss = -\sum_{i} y_i \log(a_i^L)$$

- Now $\vec{y}$ is a vector with a 1 for the good answer, and 0 everywhere else;

The Softmax Function

- We can again use **cross-entropy** with softmax (in a slightly different way):

$$Loss = -\sum_i y_i \log(a_i^L)$$

- Now $\vec{y}$ is a vector with a 1 for the good answer, and 0 everywhere else;
- The sum is over the different output neurons;

The Softmax Function

- We can again use **cross-entropy** with softmax (in a slightly different way):

$$Loss = -\sum_i y_i \log(a_i^L)$$

- Now $\vec{y}$ is a vector with a 1 for the good answer, and 0 everywhere else;
- The sum is over the different output neurons;
- Let us compute the gradients: $Loss$ is a function of all a_j^L and all of them are functions of w_{jk}^L:

$$\frac{\partial Loss}{\partial w_{jk}^L} = -\frac{\partial Loss}{\partial a_i^L}\frac{\partial a_i^L}{\partial w_{jk}^L} = -\sum_i \frac{y_i}{a_i^L}\frac{\partial a_i^L}{\partial z_j^L}\frac{\partial z_j^L}{\partial w_{jk}^L} = -\sum_i \frac{y_i}{a_i^L}\frac{\partial a_i^L}{\partial z_j^L}a_k^{L-1}$$

The Softmax Function

- We can again use **cross-entropy** with softmax (in a slightly different way):

$$Loss = -\sum_i y_i \log(a_i^L)$$

- Now $\vec{y}$ is a vector with a 1 for the good answer, and 0 everywhere else;
- The sum is over the different output neurons;
- Let us compute the gradients: $Loss$ is a function of all a_j^L and all of them are functions of w_{jk}^L:

$$\frac{\partial Loss}{\partial w_{jk}^L} = -\frac{\partial Loss}{\partial a_i^L}\frac{\partial a_i^L}{\partial w_{jk}^L} = -\sum_i \frac{y_i}{a_i^L}\frac{\partial a_i^L}{\partial z_j^L}\frac{\partial z_j^L}{\partial w_{jk}^L} = -\sum_i \frac{y_i}{a_i^L}\frac{\partial a_i^L}{\partial z_j^L} a_k^{L-1}$$

- And for biases:

$$\frac{\partial Loss}{\partial b_j^L} = -\frac{\partial Loss}{\partial a_i^L}\frac{\partial a_i^L}{\partial b_j^L} = -\sum_i \frac{y_i}{a_i^L}\frac{\partial a_i^L}{\partial z_j^L}\frac{\partial z_j^L}{\partial b_j^L} = -\sum_i \frac{y_i}{a_i^L}\frac{\partial a_i^L}{\partial z_j^L}$$

The Softmax Function

- The softmax function is similar to the logistic function. The i-th component of its partial derivative wrt. to the j-th component of the input is:

$$\left(\frac{\partial \mathcal{S}(\vec{z})}{\partial z_j}\right)_i = \mathcal{S}(\vec{z})_i(1 - \mathcal{S}(\vec{z})_j) \text{ if } i = j \text{ and } -\mathcal{S}(\vec{z})_i\mathcal{S}(\vec{z})_j \text{ otherwise.}$$

The Softmax Function

- The softmax function is similar to the logistic function. The i-th component of its partial derivative wrt. to the j-th component of the input is:

$$\left(\frac{\partial \mathcal{S}(\vec{z})}{\partial z_j}\right)_i = \mathcal{S}(\vec{z})_i(1 - \mathcal{S}(\vec{z})_j) \text{ if } i = j \text{ and } -\mathcal{S}(\vec{z})_i\mathcal{S}(\vec{z})_j \text{ otherwise.}$$

- And finally, since $a_i^L = \mathcal{S}(\vec{z}^L)_i$, with δ_{ij} the Kronecker symbol:

$$\frac{\partial Loss}{\partial w_{jk}^L} = a_k^{L-1} \sum_i y_i(a_j^L - \delta_{ij})$$

The Softmax Function

- The softmax function is similar to the logistic function. The i-th component of its partial derivative wrt. to the j-th component of the input is:

$$\left(\frac{\partial \mathcal{S}(\vec{z})}{\partial z_j}\right)_i = \mathcal{S}(\vec{z})_i(1 - \mathcal{S}(\vec{z})_j) \text{ if } i = j \text{ and } -\mathcal{S}(\vec{z})_i\mathcal{S}(\vec{z})_j \text{ otherwise.}$$

- And finally, since $a_i^L = \mathcal{S}(\vec{z}^L)_i$, with δ_{ij} the Kronecker symbol:

$$\frac{\partial Loss}{\partial w_{jk}^L} = a_k^{L-1} \sum_i y_i(a_j^L - \delta_{ij})$$

- $\vec{y}$ has 0's everywhere but in the component representing the correct answer (which is 1), so:

$$\frac{\partial Loss}{\partial w_{jk}^L} = a_k^{L-1}(a_j^L - y_j) \text{ and } \frac{\partial Loss}{\partial b_j^L} = a_j^L - y_j$$

The Softmax Function

- The softmax function is similar to the logistic function. The i-th component of its partial derivative wrt. to the j-th component of the input is:

$$\left(\frac{\partial \mathcal{S}(\vec{z})}{\partial z_j}\right)_i = \mathcal{S}(\vec{z})_i(1 - \mathcal{S}(\vec{z})_j) \text{ if } i = j \text{ and } -\mathcal{S}(\vec{z})_i\mathcal{S}(\vec{z})_j \text{ otherwise.}$$

- And finally, since $a_i^L = \mathcal{S}(\vec{z}^L)_i$, with δ_{ij} the Kronecker symbol:

$$\frac{\partial Loss}{\partial w_{jk}^L} = a_k^{L-1} \sum_i y_i(a_j^L - \delta_{ij})$$

- $\vec{y}$ has 0's everywhere but in the component representing the correct answer (which is 1), so:

$$\frac{\partial Loss}{\partial w_{jk}^L} = a_k^{L-1}(a_j^L - y_j) \text{ and } \frac{\partial Loss}{\partial b_j^L} = a_j^L - y_j$$

- Independent from the derivative of the activation function also.

Regularisation in ANN

- As before, it is often helpful to use **regularisation** to limit **overfitting**;

Regularisation in ANN

- As before, it is often helpful to use **regularisation** to limit **overfitting**;
- This is done through the loss function, as in the linear case, using L_q regularisation;

Regularisation in ANN

- As before, it is often helpful to use **regularisation** to limit **overfitting**;
- This is done through the loss function, as in the linear case, using L_q regularisation;
- We can also use combinations: $L_1 + L_2$ is known as **elastic net regularisation**;

Regularisation in ANN

- As before, it is often helpful to use **regularisation** to limit **overfitting**;
- This is done through the loss function, as in the linear case, using L_q regularisation;
- We can also use combinations: $L_1 + L_2$ is known as **elastic net regularisation**;
- This can also be done by acting directly on the weights: dropout regularisation;

Regularisation in ANN

- As before, it is often helpful to use **regularisation** to limit **overfitting**;
- This is done through the loss function, as in the linear case, using L_q regularisation;
- We can also use combinations: $L_1 + L_2$ is known as **elastic net regularisation**;
- This can also be done by acting directly on the weights: dropout regularisation;
- As with all other methods, **increasing the size** of the training set, possibly artificially, reduces overfitting.

Regularisation in ANN: L_2 Regularisation

- L_2 regularisation: add, for $\frac{\lambda\alpha}{n} \in [0,1]$, the term ($n$ is the total number of examples):

$$R_2(\vec{w}) = \frac{\lambda}{2n}\Big(\sum_{j,k,\ell}(w_{jk}^{\ell})^2 + \sum_{j,\ell}(b_j^{\ell})^2\Big)$$

Regularisation in ANN: L_2 Regularisation

- L_2 regularisation: add, for $\frac{\lambda\alpha}{n} \in [0,1]$, the term ($n$ is the total number of examples):

$$R_2(\vec{w}) = \frac{\lambda}{2n}\Big(\sum_{j,k,\ell}(w_{jk}^{\ell})^2 + \sum_{j,\ell}(b_j^{\ell})^2\Big)$$

- We must add the derivative of the regularisation to compute the gradient:

$$\frac{\partial Loss}{\partial w_{jk}} = \frac{\partial NormalLoss}{\partial w_{jk}} + \frac{\partial R_2}{\partial w_{jk}} = \frac{\partial NormalLoss}{\partial w_{jk}} + \frac{\lambda}{n} w_{jk}$$

$$\frac{\partial Loss}{\partial b_j} = \frac{\partial NormalLoss}{\partial b_j} + \frac{\partial R_2}{\partial b_j} = \frac{\partial NormalLoss}{\partial b_j} + \frac{\lambda}{n} b_j$$

Regularisation in ANN: L_2 Regularisation

- L_2 regularisation: add, for $\frac{\lambda\alpha}{n} \in [0,1]$, the term ($n$ is the total number of examples):

$$R_2(\vec{w}) = \frac{\lambda}{2n}\Big(\sum_{j,k,\ell}(w^\ell_{jk})^2 + \sum_{j,\ell}(b^\ell_j)^2\Big)$$

- We must add the derivative of the regularisation to compute the gradient:

$$\frac{\partial Loss}{\partial w_{jk}} = \frac{\partial NormalLoss}{\partial w_{jk}} + \frac{\partial R_2}{\partial w_{jk}} = \frac{\partial NormalLoss}{\partial w_{jk}} + \frac{\lambda}{n} w_{jk}$$

$$\frac{\partial Loss}{\partial b_j} = \frac{\partial NormalLoss}{\partial b_j} + \frac{\partial R_2}{\partial b_j} = \frac{\partial NormalLoss}{\partial b_j} + \frac{\lambda}{n} b_j$$

- The update rules become:

$$\boxed{w_{jk} \leftarrow (1 - \tfrac{\alpha\lambda}{n}) w_{jk} - \alpha \tfrac{\partial NormalLoss}{\partial w_{jk}}}$$

$$\boxed{b_j \leftarrow (1 - \tfrac{\alpha\lambda}{n}) b_j - \alpha \tfrac{\partial NormalLoss}{\partial b_j}}$$

Regularisation in ANN: L_1 Regularisation

- L_1 regularisation works similarly to L_2;

Regularisation in ANN: L_1 Regularisation

- L_1 regularisation works similarly to L_2;
- Add to the cost:

$$R_1(\vec{w}) = \frac{\lambda}{n}\Big(\sum_{j,k,\ell} |w_{jk}^{\ell}| + \sum_{j,\ell} |b_j^{\ell}|\Big)$$

Regularisation in ANN: L_1 Regularisation

- L_1 regularisation works similarly to L_2;
- Add to the cost:

$$R_1(\vec{w}) = \frac{\lambda}{n}\Big(\sum_{j,k,\ell}|w_{jk}^{\ell}| + \sum_{j,\ell}|b_j^{\ell}|\Big)$$

- The update rules become:

$$\boxed{w_{jk} \leftarrow w_{jk} - \frac{\alpha\lambda}{n} sign(w_{jk}) - \alpha\frac{\partial NormalLoss}{\partial w_{jk}}}$$

$$\boxed{b_j \leftarrow b_j - \frac{\alpha\lambda}{n} sign(b_j) - \alpha\frac{\partial NormalLoss}{\partial b_j}}$$

Regularisation in ANN: L_1 Regularisation

- L_1 regularisation works similarly to L_2;
- Add to the cost:

$$R_1(\vec{w}) = \frac{\lambda}{n}\Big(\sum_{j,k,\ell} |w_{jk}^{\ell}| + \sum_{j,\ell} |b_j^{\ell}|\Big)$$

- The update rules become:

$$\boxed{w_{jk} \leftarrow w_{jk} - \frac{\alpha\lambda}{n} sign(w_{jk}) - \alpha \frac{\partial NormalLoss}{\partial w_{jk}}}$$

$$\boxed{b_j \leftarrow b_j - \frac{\alpha\lambda}{n} sign(b_j) - \alpha \frac{\partial NormalLoss}{\partial b_j}}$$

- R_1 is not differentiable wrt. w_{jk} or b_j when that parameter is 0;

Regularisation in ANN: L_1 Regularisation

- L_1 regularisation works similarly to L_2;
- Add to the cost:

$$R_1(\vec{w}) = \frac{\lambda}{n}\left(\sum_{j,k,\ell}|w^{\ell}_{jk}| + \sum_{j,\ell}|b^{\ell}_j|\right)$$

- The update rules become:

$$\boxed{w_{jk} \leftarrow w_{jk} - \frac{\alpha\lambda}{n} sign(w_{jk}) - \alpha\frac{\partial NormalLoss}{\partial w_{jk}}}$$

$$\boxed{b_j \leftarrow b_j - \frac{\alpha\lambda}{n} sign(b_j) - \alpha\frac{\partial NormalLoss}{\partial b_j}}$$

- R_1 is not differentiable wrt. w_{jk} or b_j when that parameter is 0;
- We can ignore this case by considering $sign(0) = 0$.

Regularisation in ANN: Dropout

- At the start of each batch of the SGD, **delete** each hidden neuron with probability p, and train normally;

Regularisation in ANN: Dropout

- At the start of each batch of the SGD, **delete** each hidden neuron with probability p, and train normally;
- At then end of the training multiply all weights of the n hidden neurons by p;

Regularisation in ANN: Dropout

- At the start of each batch of the SGD, **delete** each hidden neuron with probability p, and train normally;
- At then end of the training multiply all weights of the n hidden neurons by p;
- This simulates the training of 2^n nets at once, with an expected value of their results;

Regularisation in ANN: Dropout

- At the start of each batch of the SGD, **delete** each hidden neuron with probability p, and train normally;
- At then end of the training multiply all weights of the n hidden neurons by p;
- This simulates the training of 2^n nets at once, with an expected value of their results;
- It reduces **overfitting**;

Regularisation in ANN: Dropout

- At the start of each batch of the SGD, **delete** each hidden neuron with probability p, and train normally;
- At then end of the training multiply all weights of the n hidden neurons by p;
- This simulates the training of 2^n nets at once, with an expected value of their results;
- It reduces **overfitting**;
- And it makes each training stage faster (less neurons).

Deep Neural Networks and Deep Learning

- A **deep** neural network is one with (much) more than one hidden layer;

Deep Neural Networks and Deep Learning

- A **deep** neural network is one with (much) more than one hidden layer;
- **Deep learning** is learning with deep networks;

Deep Neural Networks and Deep Learning

- A **deep** neural network is one with (much) more than one hidden layer;
- **Deep learning** is learning with deep networks;
- Its main problem is the propagation of gradient along the many layers is **unstable**.

Exploding and Vanishing Gradients

- Suppose we have a network with n hidden layers, each with one neuron:
  Indices denote the layer

$$\Delta_1 = g'(z_1)w_1\Delta_2 = g'(z_1)w_1 g'(z_2)w_2 \cdots g'(z_n)w_n\Delta_{n+1}$$

Exploding and Vanishing Gradients

- Suppose we have a network with n hidden layers, each with one neuron:
  Indices denote the layer

$$\Delta_1 = g'(z_1)w_1\Delta_2 = g'(z_1)w_1 g'(z_2)w_2 \cdots g'(z_n)w_n\Delta_{n+1}$$

- Vanishing gradient: if $g'(z_i)w_i$'s are all less than 1 then this goes exponentially fast to 0 with n;

Exploding and Vanishing Gradients

- Suppose we have a network with n hidden layers, each with one neuron:
  Indices denote the layer

$$\Delta_1 = g'(z_1)w_1\Delta_2 = g'(z_1)w_1g'(z_2)w_2 \cdots g'(z_n)w_n\Delta_{n+1}$$

- Vanishing gradient: if $g'(z_i)w_i$'s are all less than 1 then this goes exponentially fast to 0 with n;
- Exploding gradient: If they are all greater than one then it goes to $+\infty$ exponentially fast with n.

Exploding and Vanishing Gradients

- Suppose we have a network with n hidden layers, each with one neuron:
  Indices denote the layer

$$\Delta_1 = g'(z_1)w_1\Delta_2 = g'(z_1)w_1g'(z_2)w_2 \cdots g'(z_n)w_n\Delta_{n+1}$$

- Vanishing gradient: if $g'(z_i)w_i$'s are all less than 1 then this goes exponentially fast to 0 with n;
- Exploding gradient: If they are all greater than one then it goes to $+\infty$ exponentially fast with n.
- To circumvent this problem:

Exploding and Vanishing Gradients

- Suppose we have a network with n hidden layers, each with one neuron:
  Indices denote the layer

$$\Delta_1 = g'(z_1)w_1\Delta_2 = g'(z_1)w_1 g'(z_2)w_2 \cdots g'(z_n)w_n\Delta_{n+1}$$

- Vanishing gradient: if $g'(z_i)w_i$'s are all less than 1 then this goes exponentially fast to 0 with n;
- Exploding gradient: If they are all greater than one then it goes to $+\infty$ exponentially fast with n.
- To circumvent this problem:
  - Regularisation;

Exploding and Vanishing Gradients

- Suppose we have a network with n hidden layers, each with one neuron:
  Indices denote the layer

$$\Delta_1 = g'(z_1)w_1\Delta_2 = g'(z_1)w_1g'(z_2)w_2 \cdots g'(z_n)w_n\Delta_{n+1}$$

- Vanishing gradient: if $g'(z_i)w_i$'s are all less than 1 then this goes exponentially fast to 0 with n;
- Exploding gradient: If they are all greater than one then it goes to $+\infty$ exponentially fast with n.
- To circumvent this problem:
  - Regularisation;
  - Adapted activation functions: e.g. **ReLU**;

Exploding and Vanishing Gradients

- Suppose we have a network with n hidden layers, each with one neuron:
  Indices denote the layer

$$\Delta_1 = g'(z_1)w_1\Delta_2 = g'(z_1)w_1g'(z_2)w_2\cdots g'(z_n)w_n\Delta_{n+1}$$

- Vanishing gradient: if $g'(z_i)w_i$'s are all less than 1 then this goes exponentially fast to 0 with n;
- Exploding gradient: If they are all greater than one then it goes to $+\infty$ exponentially fast with n.
- To circumvent this problem:
  - Regularisation;
  - Adapted activation functions: e.g. **ReLU**;
  - Specific architectures: e.g. **convolutional** networks.

Convolutional Neural Networks

- **Convolutional neural networks** are designed to work with spatially organised inputs, typically images;

Convolutional Neural Networks

- **Convolutional neural networks** are designed to work with spatially organised inputs, typically images;
- They use four types of hidden layers:

Convolutional Neural Networks

- **Convolutional neural networks** are designed to work with spatially organised inputs, typically images;
- They use four types of hidden layers:
  - Convolutional layers;

Convolutional Neural Networks

- **Convolutional neural networks** are designed to work with spatially organised inputs, typically images;
- They use four types of hidden layers:
  - Convolutional layers;
  - Pooling layers;

Convolutional Neural Networks

- **Convolutional neural networks** are designed to work with spatially organised inputs, typically images;
- They use four types of hidden layers:
  - Convolutional layers;
  - Pooling layers;
  - ReLU layers (apply ReLU to their input);

Convolutional Neural Networks

- **Convolutional neural networks** are designed to work with spatially organised inputs, typically images;
- They use four types of hidden layers:
  - Convolutional layers;
  - Pooling layers;
  - ReLU layers (apply ReLU to their input);
  - Fully connected layers.

Convolutional Neural Networks

- **Convolutional neural networks** are designed to work with spatially organised inputs, typically images;
- They use four types of hidden layers:
  - Convolutional layers;
  - Pooling layers;
  - ReLU layers (apply ReLU to their input);
  - Fully connected layers.
- Fully connected layers work as seen before.

Convolutional Layers

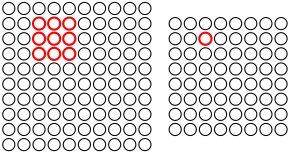
- A **convolutional layer** links each hidden neuron to a subset (**receptive field**) of the previous layer;

Convolutional Layers

- A **convolutional layer** links each hidden neuron to a subset (**receptive field**) of the previous layer;
- Typically in a 2D image, where inputs are e.g. pixel intensity, this subset is a square window;

Convolutional Layers

- A **convolutional layer** links each hidden neuron to a subset (**receptive field**) of the previous layer;
- Typically in a 2D image, where inputs are e.g. pixel intensity, this subset is a square window;
- As the square window moves on the image, with a given **stride**, we obtain as many convolution neurons:



Convolutional Layers

- Each neuron in the receptive field is linked to the corresponding convolutional neuron with a weight;

Convolutional Layers

- Each neuron in the receptive field is linked to the corresponding convolutional neuron with a weight;
- The convolutional neuron also has a bias (constant term) as usual;

Convolutional Layers

- Each neuron in the receptive field is linked to the corresponding convolutional neuron with a weight;
- The convolutional neuron also has a bias (constant term) as usual;
- These parameters are **shared** between the neurons in the convolutional layer.

Convolutional Layers

- Each neuron in the receptive field is linked to the corresponding convolutional neuron with a weight;
- The convolutional neuron also has a bias (constant term) as usual;
- These parameters are **shared** between the neurons in the convolutional layer.
- Those neurons detect the **same features**;

Convolutional Layers

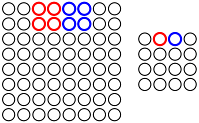
- Each neuron in the receptive field is linked to the corresponding convolutional neuron with a weight;
- The convolutional neuron also has a bias (constant term) as usual;
- These parameters are **shared** between the neurons in the convolutional layer.
- Those neurons detect the **same features**;
- Several such **feature maps** can be put in parallel, each with its own shared weights.

Pooling Layers

- A pooling layer comes after a convolutional layer;

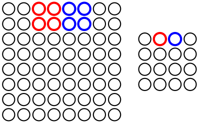
Pooling Layers

- A pooling layer comes after a convolutional layer;
- Neurons in that layer aggregate the information from the feature maps;



Pooling Layers

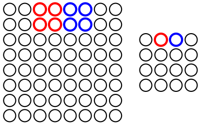
- A pooling layer comes after a convolutional layer;
- Neurons in that layer aggregate the information from the feature maps;



- It performs a **fixed operation** on the activation of neurons in a given subregion of feature maps;

Pooling Layers

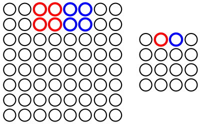
- A pooling layer comes after a convolutional layer;
- Neurons in that layer aggregate the information from the feature maps;



- It performs a **fixed operation** on the activation of neurons in a given subregion of feature maps;
- A single pooling neuron is linked to all the neurons corresponding to the subregion in **all the feature maps**.

Pooling Layers

- A pooling layer comes after a convolutional layer;
- Neurons in that layer aggregate the information from the feature maps;



- It performs a **fixed operation** on the activation of neurons in a given subregion of feature maps;
- A single pooling neuron is linked to all the neurons corresponding to the subregion in **all the feature maps**.
- Commmon aggregation operations are **max** and **average**.

Convolutional Neural Networks

- The complete typical architecture of a convolutional neural network is:

Convolutional Neural Networks

- The complete typical architecture of a convolutional neural network is:
    1. One or several convolutional layers each with a pooling layer (feature detection);

Convolutional Neural Networks

- The complete typical architecture of a convolutional neural network is:
  1. One or several convolutional layers each with a pooling layer (feature detection);
  2. One or several fully interconnected layers (the high-level reasoning);

Convolutional Neural Networks

- The complete typical architecture of a convolutional neural network is:
  1. One or several convolutional layers each with a pooling layer (feature detection);
  2. One or several fully interconnected layers (the high-level reasoning);
  3. An output layer with an associated loss function (as usual).

Convolutional Neural Networks

- The complete typical architecture of a convolutional neural network is:
  1. One or several convolutional layers each with a pooling layer (feature detection);
  2. One or several fully interconnected layers (the high-level reasoning);
  3. An output layer with an associated loss function (as usual).
- Convolutional layers have relatively **few parameters**, which helps with overfitting and speed.

Convolutional Neural Networks

- The complete typical architecture of a convolutional neural network is:
  1. One or several convolutional layers each with a pooling layer (feature detection);
  2. One or several fully interconnected layers (the high-level reasoning);
  3. An output layer with an associated loss function (as usual).
- Convolutional layers have relatively **few parameters**, which helps with overfitting and speed.
- Pooling layers have no modifiable parameters and can be implemented as part of the convolutional layers to which they correspond.

Artificial Neural Networks: Conclusion

- ANN are a **powerful** machine-learning technique;

Artificial Neural Networks: Conclusion

- ANN are a **powerful** machine-learning technique;
- They are very **flexible** and can have excellent **accuracy**;

Artificial Neural Networks: Conclusion

- ANN are a **powerful** machine-learning technique;
- They are very **flexible** and can have excellent **accuracy**;
- They also have weaknesses:

Artificial Neural Networks: Conclusion

- ANN are a **powerful** machine-learning technique;
- They are very **flexible** and can have excellent **accuracy**;
- They also have weaknesses:
  - They are slower than many other techniques and therefore resource-hungry;

Artificial Neural Networks: Conclusion

- ANN are a **powerful** machine-learning technique;
- They are very **flexible** and can have excellent **accuracy**;
- They also have weaknesses:
    - They are slower than many other techniques and therefore resource-hungry;
    - We lack much theory to formally understand what is happening;

Artificial Neural Networks: Conclusion

- ANN are a **powerful** machine-learning technique;
- They are very **flexible** and can have excellent **accuracy**;
- They also have weaknesses:
  - They are slower than many other techniques and therefore resource-hungry;
  - We lack much theory to formally understand what is happening;
  - They often must be used as black-boxes with results not explainable;

Artificial Neural Networks: Conclusion

- ANN are a **powerful** machine-learning technique;
- They are very **flexible** and can have excellent **accuracy**;
- They also have weaknesses:
  - They are slower than many other techniques and therefore resource-hungry;
  - We lack much theory to formally understand what is happening;
  - They often must be used as black-boxes with results not explainable;
  - They can require lots of tweaking to get good results, and therefore expert knowledge;

Artificial Neural Networks: Conclusion

- ANN are a **powerful** machine-learning technique;
- They are very **flexible** and can have excellent **accuracy**;
- They also have weaknesses:
  - They are slower than many other techniques and therefore resource-hungry;
  - We lack much theory to formally understand what is happening;
  - They often must be used as black-boxes with results not explainable;
  - They can require lots of tweaking to get good results, and therefore expert knowledge;
  - Can be fooled by specially innocuous-looking constructed inputs.

Artificial Neural Networks: Conclusion

- ANN are a **powerful** machine-learning technique;
- They are very **flexible** and can have excellent **accuracy**;
- They also have weaknesses:
  - They are slower than many other techniques and therefore resource-hungry;
  - We lack much theory to formally understand what is happening;
  - They often must be used as black-boxes with results not explainable;
  - They can require lots of tweaking to get good results, and therefore expert knowledge;
  - Can be fooled by specially innocuous-looking constructed inputs.
- Current trend is to use very deep networks ($>$100 layers) with:

Artificial Neural Networks: Conclusion

- ANN are a **powerful** machine-learning technique;
- They are very **flexible** and can have excellent **accuracy**;
- They also have weaknesses:
  - They are slower than many other techniques and therefore resource-hungry;
  - We lack much theory to formally understand what is happening;
  - They often must be used as black-boxes with results not explainable;
  - They can require lots of tweaking to get good results, and therefore expert knowledge;
  - Can be fooled by specially innocuous-looking constructed inputs.
- Current trend is to use very deep networks ($>$100 layers) with:
  - batch normalisation layers: learn a normalisation of the output of the previous layer

Artificial Neural Networks: Conclusion

- ANN are a **powerful** machine-learning technique;
- They are very **flexible** and can have excellent **accuracy**;
- They also have weaknesses:
  - They are slower than many other techniques and therefore resource-hungry;
  - We lack much theory to formally understand what is happening;
  - They often must be used as black-boxes with results not explainable;
  - They can require lots of tweaking to get good results, and therefore expert knowledge;
  - Can be fooled by specially innocuous-looking constructed inputs.
- Current trend is to use very deep networks ($>$100 layers) with:
  - batch normalisation layers: learn a normalisation of the output of the previous layer
  - residual networks: replicate lots of time a pattern that adds the input of a layer to the output of the next one.

Outline

Learning a Model for Utility

- For better **generalisation** and memory usage, we can learn utility as an approximated more "concise" function of states instead of a big table;

Learning a Model for Utility

- For better **generalisation** and memory usage, we can learn utility as an approximated more “concise” function of states instead of a big table;
- Models for the function can be linear, decision trees, artificial neural networks, etc.

Learning a Model for Utility

- For better **generalisation** and memory usage, we can learn utility as an approximated more “concise” function of states instead of a big table;
- Models for the function can be linear, decision trees, artificial neural networks, etc.
- The basic approach is:

Learning a Model for Utility

- For better **generalisation** and memory usage, we can learn utility as an approximated more “concise” function of states instead of a big table;
- Models for the function can be linear, decision trees, artificial neural networks, etc.
- The basic approach is:
  - Perform n trials directed by the current approximation;

Learning a Model for Utility

- For better **generalisation** and memory usage, we can learn utility as an approximated more "concise" function of states instead of a big table;
- Models for the function can be linear, decision trees, artificial neural networks, etc.
- The basic approach is:
  - Perform n trials directed by the current approximation;
  - Record the predictions (the model) and targets (observations) for all visited states;

Learning a Model for Utility

- For better **generalisation** and memory usage, we can learn utility as an approximated more "concise" function of states instead of a big table;
- Models for the function can be linear, decision trees, artificial neural networks, etc.
- The basic approach is:
  - Perform n trials directed by the current approximation;
  - Record the predictions (the model) and targets (observations) for all visited states;
  - Learn the results using supervised learning;

Learning a Model for Utility

- For better **generalisation** and memory usage, we can learn utility as an approximated more "concise" function of states instead of a big table;
- Models for the function can be linear, decision trees, artificial neural networks, etc.
- The basic approach is:
  - Perform n trials directed by the current approximation;
  - Record the predictions (the model) and targets (observations) for all visited states;
  - Learn the results using supervised learning;
  - Repeat.

Learning a Model for Utility

- We can discard some of the history;

Learning a Model for Utility

- We can discard some of the history;
- The target can be observed total rewards (Monte-carlo), or the right handside of Q-learning, etc.

Learning a Model for Utility

- We can discard some of the history;
- The target can be observed total rewards (Monte-carlo), or the right handside of Q-learning, etc.
- Models that can learn incrementally can be more efficient (neural networks, linear functions, etc.)

Learning a Model for Utility: Linear functions

- For linear functions, we can learn at every step:

Learning a Model for Utility: Linear functions

- For linear functions, we can learn at every step:
- Assume $U_{\vec{\theta}}(s) = \theta_0 + a_1(s)\theta_1 + a_2(s)\theta_2 + \cdots + a_n(s)\theta_n$.

Learning a Model for Utility: Linear functions

- For linear functions, we can learn at every step:
- Assume $U_{\vec{\theta}}(s) = \theta_0 + a_1(s)\theta_1 + a_2(s)\theta_2 + \cdots + a_n(s)\theta_n$.
- For any trial, the error we get can be defined as (where $u(s)$ is the obtained total reward for the trial)

$$E(s) = \frac{1}{2}(U_{\vec{\theta}}(s) - u(s))^2$$

Learning a Model for Utility: Linear functions

- For linear functions, we can learn at every step:
- Assume $U_{\vec{\theta}}(s) = \theta_0 + a_1(s)\theta_1 + a_2(s)\theta_2 + \cdots + a_n(s)\theta_n$.
- For any trial, the error we get can be defined as (where $u(s)$ is the obtained total reward for the trial)

$$E(s) = \frac{1}{2}(U_{\vec{\theta}}(s) - u(s))^2$$

- For all i, the gradient for θ_i is $\frac{\partial E(s)}{\partial \theta_i} = (U_{\vec{\theta}}(s) - u(s))\frac{\partial U_{\vec{\theta}}(s)}{\partial \theta_i}$;

Learning a Model for Utility: Linear functions

- For linear functions, we can learn at every step:
- Assume $U_{\vec{\theta}}(s) = \theta_0 + a_1(s)\theta_1 + a_2(s)\theta_2 + \cdots + a_n(s)\theta_n$.
- For any trial, the error we get can be defined as (where $u(s)$ is the obtained total reward for the trial)

$$E(s) = \frac{1}{2}(U_{\vec{\theta}}(s) - u(s))^2$$

- For all i, the gradient for θ_i is $\frac{\partial E(s)}{\partial \theta_i} = (U_{\vec{\theta}}(s) - u(s))\frac{\partial U_{\vec{\theta}}(s)}{\partial \theta_i}$;
- Using gradient descent:

Learning a Model for Utility: Linear functions

- For linear functions, we can learn at every step:
- Assume $U_{\vec{\theta}}(s) = \theta_0 + a_1(s)\theta_1 + a_2(s)\theta_2 + \cdots + a_n(s)\theta_n$.
- For any trial, the error we get can be defined as (where $u(s)$ is the obtained total reward for the trial)

$$E(s) = \frac{1}{2}(U_{\vec{\theta}}(s) - u(s))^2$$

- For all i, the gradient for θ_i is $\frac{\partial E(s)}{\partial \theta_i} = (U_{\vec{\theta}}(s) - u(s))\frac{\partial U_{\vec{\theta}}(s)}{\partial \theta_i}$;
- Using gradient descent:
    - $\theta_0 \leftarrow \theta_0 + \alpha(u(s) - U_{\vec{\theta}}(s))$;

Learning a Model for Utility: Linear functions

- For linear functions, we can learn at every step:
- Assume $U_{\vec{\theta}}(s) = \theta_0 + a_1(s)\theta_1 + a_2(s)\theta_2 + \cdots + a_n(s)\theta_n$.
- For any trial, the error we get can be defined as (where $u(s)$ is the obtained total reward for the trial)
$$E(s) = \frac{1}{2}(U_{\vec{\theta}}(s) - u(s))^2$$
- For all i, the gradient for θ_i is $\frac{\partial E(s)}{\partial \theta_i} = (U_{\vec{\theta}}(s) - u(s))\frac{\partial U_{\vec{\theta}}(s)}{\partial \theta_i}$;
- Using gradient descent:
  - $\theta_0 \leftarrow \theta_0 + \alpha(u(s) - U_{\vec{\theta}}(s))$;
  - for all $i \neq 0$, $\theta_i \leftarrow \theta_i + \alpha(u(s) - U_{\vec{\theta}}(s))a_i(s)$.

Learning a Model for Utility: Linear functions

- For linear functions, we can learn at every step:
- Assume $U_{\vec{\theta}}(s) = \theta_0 + a_1(s)\theta_1 + a_2(s)\theta_2 + \cdots + a_n(s)\theta_n$.
- For any trial, the error we get can be defined as (where $u(s)$ is the obtained total reward for the trial)

$$E(s) = \frac{1}{2}\left(U_{\vec{\theta}}(s) - u(s)\right)^2$$

- For all i, the gradient for θ_i is $\frac{\partial E(s)}{\partial \theta_i} = \left(U_{\vec{\theta}}(s) - u(s)\right)\frac{\partial U_{\vec{\theta}}(s)}{\partial \theta_i}$;
- Using gradient descent:
  - $\theta_0 \leftarrow \theta_0 + \alpha(u(s) - U_{\vec{\theta}}(s))$;
  - for all $i \neq 0$, $\theta_i \leftarrow \theta_i + \alpha(u(s) - U_{\vec{\theta}}(s))a_i(s)$.
- This update is done after each trial;

Learning a Model for Utility: Linear functions

- For linear functions, we can learn at every step:
- Assume $U_{\vec{\theta}}(s) = \theta_0 + a_1(s)\theta_1 + a_2(s)\theta_2 + \cdots + a_n(s)\theta_n$.
- For any trial, the error we get can be defined as (where $u(s)$ is the obtained total reward for the trial)

$$E(s) = \frac{1}{2}(U_{\vec{\theta}}(s) - u(s))^2$$

- For all i, the gradient for θ_i is $\frac{\partial E(s)}{\partial \theta_i} = (U_{\vec{\theta}}(s) - u(s))\frac{\partial U_{\vec{\theta}}(s)}{\partial \theta_i}$;
- Using gradient descent:
  - $\theta_0 \leftarrow \theta_0 + \alpha(u(s) - U_{\vec{\theta}}(s))$;
  - for all $i \neq 0$, $\theta_i \leftarrow \theta_i + \alpha(u(s) - U_{\vec{\theta}}(s))a_i(s)$.
- This update is done after each trial;
- We can do the same for TD learning (or Q-learning). After each step from s to s':

$$\theta_i \leftarrow \theta_i + \alpha(R(s) + \gamma U_{\vec{\theta}}(s') - U_{\vec{\theta}}(s))\frac{\partial U_{\vec{\theta}}(s)}{\partial \theta_i}$$

Learning a Model for Utility: Linear functions

- For linear functions, we can learn at every step:
- Assume $U_{\vec{\theta}}(s) = \theta_0 + a_1(s)\theta_1 + a_2(s)\theta_2 + \cdots + a_n(s)\theta_n$.
- For any trial, the error we get can be defined as (where $u(s)$ is the obtained total reward for the trial)

$$E(s) = \frac{1}{2}(U_{\vec{\theta}}(s) - u(s))^2$$

- For all i, the gradient for θ_i is $\frac{\partial E(s)}{\partial \theta_i} = (U_{\vec{\theta}}(s) - u(s))\frac{\partial U_{\vec{\theta}}(s)}{\partial \theta_i}$;
- Using gradient descent:
  - $\theta_0 \leftarrow \theta_0 + \alpha(u(s) - U_{\vec{\theta}}(s))$;
  - for all $i \neq 0$, $\theta_i \leftarrow \theta_i + \alpha(u(s) - U_{\vec{\theta}}(s))a_i(s)$.
- This update is done after each trial;
- We can do the same for TD learning (or Q-learning). After each step from s to s':

$$\theta_i \leftarrow \theta_i + \alpha(R(s) + \gamma U_{\vec{\theta}}(s') - U_{\vec{\theta}}(s))\frac{\partial U_{\vec{\theta}}(s)}{\partial \theta_i}$$

- Except for linear models with a fixed policy, convergence is **not theoretically guaranteed**.

Outline

Conclusion

- Rather than designing **intelligent** agents, it is easier to design **rational** agents;
- In order to do so artificial intelligence borrows heavily to mathematics, economics, computer science, control theory, signal processing, etc.
- Rational agents should **learn** from their environment and **plan** their actions to **optimise** some objective function;
- Relying on **models** for the environment usually makes this more effective;
- Useful models must cope with uncertainty, partial observation, non-determinism, and account for other agents.

Going further

Concepts we have not addressed include:

- Represent and reason on accumulated knowledge using:
  - logic;
  - Bayesian networks;
- More on supervised learning:
  - SVMs;
  - Non parameterised models (k-means...);
  - Bayesian learning.
- Unsupervised learning (clustering...)

References

- Stuart Russel and Peter Norvig, *Artificial Intelligence: A Modern Approach (3rd edition)*, Prentice-Hall, 2010.
- Richard S. Sutton and Andrew G. Barto, *Reinforcement Learning: An introduction*, MIT Press, 1998.
- The many Wikipedia pages on the topics addressed here: `http://www.wikipedia.org`
- `http://neuralnetworksanddeeplearning.com/`
- Saul X. Levmore and Elizabeth Early Cook. *Super Strategies for Puzzles and Games*. Doubleday, Garden City, NY, 1981.
- Günter Rote. *Crossing the bridge at night*. Bulletin of the European Association for Theoretical Computer Science. Vol. 78. pp. 241–246. 2002.