COMP3411

WEEK 1 — Intro

▼ Wk 1 — History of AI

1950 — Alan Turing's paper on the imitation game and child machine

Believed computers (in 50yrs) would be able to imitate humans to a high degree (avg interrogator will not have more than a 70% chance of correctly distinguishing a human from a machine playing the imitation game)

Suggested programming machines to simulate a child's mind, then educating them to simulate the adult's mind instead of programming an adult's mind

1956 — Dartmouth Conference

Considered the founding event of AI

- Research project to assess the state and future of AI
- Challenge was to demonstrate that every aspect of learning and human intelligence could be simulated by a machine

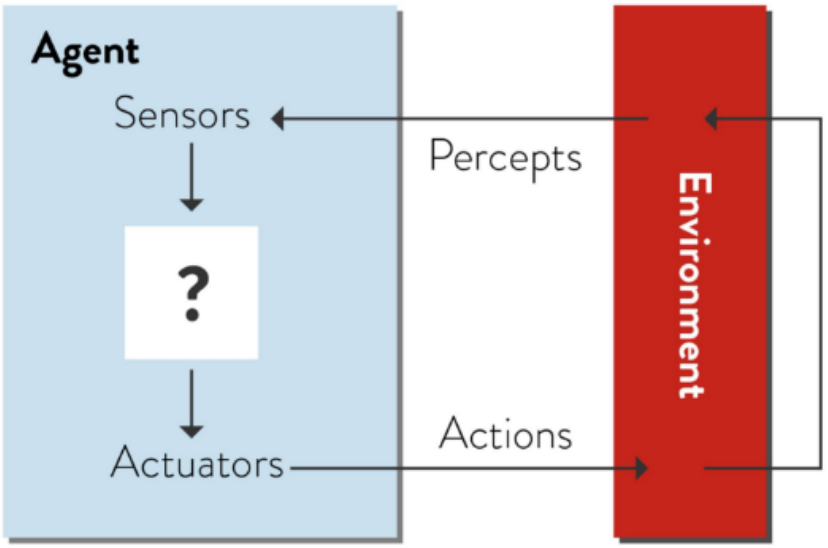
1969 — Shakey Robot

First mobile robot with sufficient artificial intelligence to navigate through rooms by itself

- Was able to perceive and reason about its surroundings
- Could perform tasks that required planning and route-finding

▼ Wk 1 — Agents

Through agents, we aim to create autonomous systems that can exhibit complex behaviours in dynamic environments



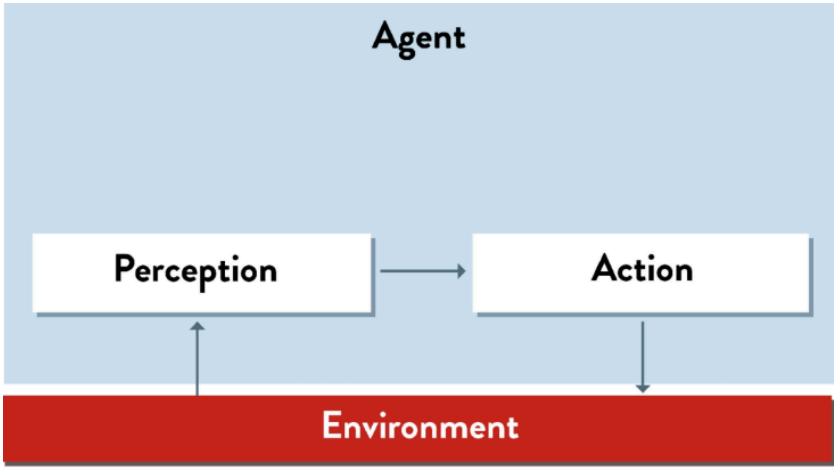
1. Agent perceives the current state of its environment through sensors
2. Agent processes perceived information to determine an action to take
3. Agent controls actuators ato take that action and change the state of the environment
4. Loop to step 1

AI Reasoning

- The world model must be represented in a way that makes reasoning easy
- Reasoning (problem solving and planning) in AI almost always involves some kind of search amongst possible solutions

▼ Reactive Agents

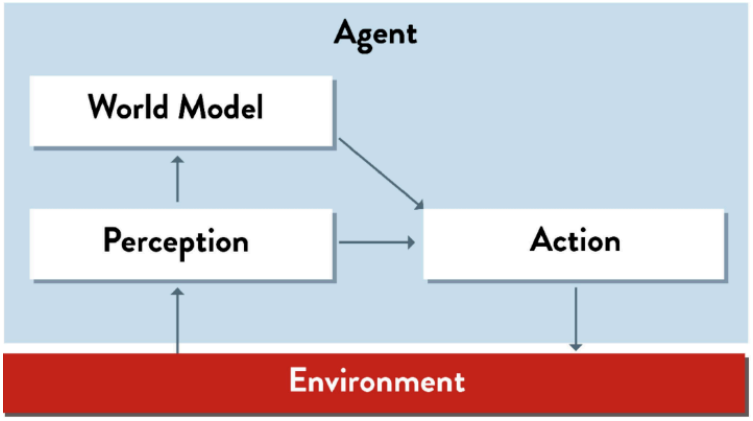
Reactive Agents react only to the raw perceived information



- Uses a policy (set of condition-action rules that are simple to apply) to decide how to react to the perceived information (if this do that, else do that)
- Limited in that they have no memory or "state" → sort of like greedy algo
  - Cannot base decisions on previous observations
  - May repeat the same sequence of actions over and over in infinite loop, which may be escapable through an element of randomness in its actions
  - Must be at the current state to see the next state, often preventing most optimal reactions
- Aka simple reflex agents

▼ Model-Based Agents

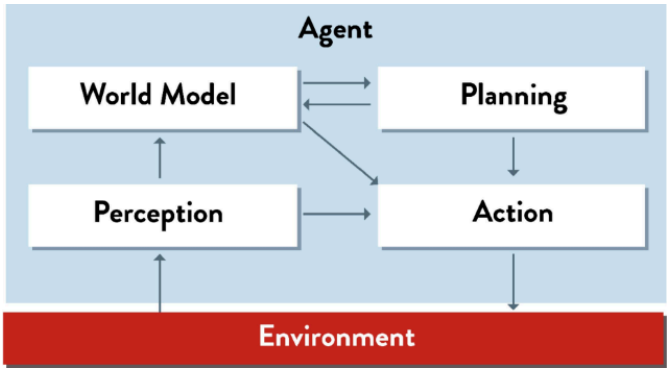
Model-based agents handle partial observability by maintaining a model that keeps track of the part of the environment (world) that it cannot currently see



- Maintains an internal state (World Model) that depends on previously perceived information, remembering some of the unobserved aspects of the current perceived state
- The world model allows it to handle partial visibility → react to the currently unobservable portion of the environment
- Perceives the current state of the world, sees how the world has evolved (effects of its actions), then chooses an action in the same way as a reflex agent (policy)
- Limited in that they only look into the past history, and cannot plan
  - Will perform poorly when tasks require searching several moves ahead, many individual steps, or logical reasoning to achieve goals

▼ Planning Agents

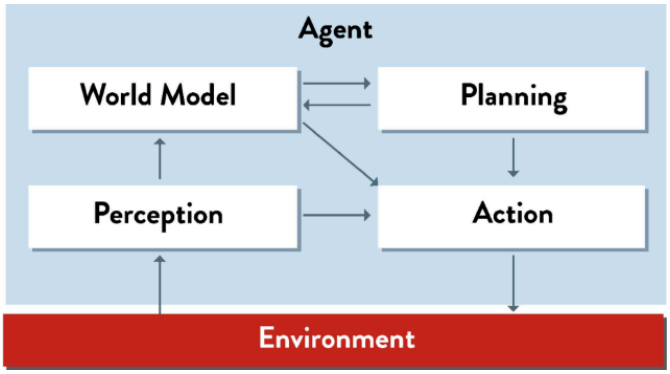
Decision-making involves considerations of the future impact of actions on the current state, and choosing the best action for its goals



- More flexible since the knowledge supporting its decisions is represented explicitly and modifiable; agent's behaviour can easily be changed
- Slower to react since it has to plan/"think"
    1. Perceive the current state of the world and "understand" the effects of its actions by comparing it with the World Model
    2. Use this understanding to predict what is likely to happen if it does a certain action,
    3. Pick the action where the predicted outcome is most optimal with respect to its goals
- Note that, to find and pick the globally optimally next action, the agent may need to search through different possible actions and sequences of actions
- Aka goal-based or teleological agent

▼ Utility-Based Agents

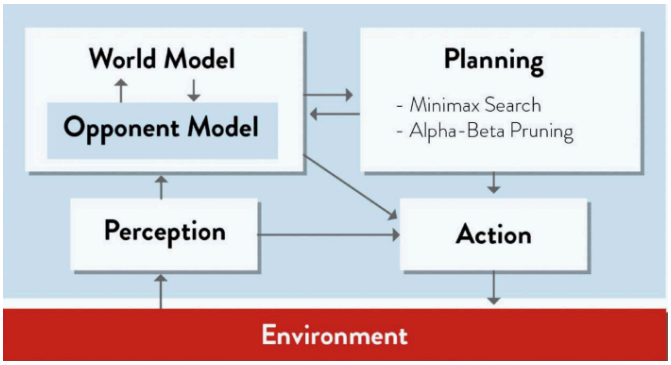
Decision-making involves considerations of the future impact of actions on the current state, and choosing the action that will make it most "happy" wrt its utility



- Does the same stuff as a planning agent, except it uses some Utility to evaluate how "happy" it will be in the predicted outcome of actions/sequences of actions, picking the the predicted state for which utility happiness is maximal
- "Happy" refers to a state's satisfaction, measured by some Utility

▼ Game-Playing Agents

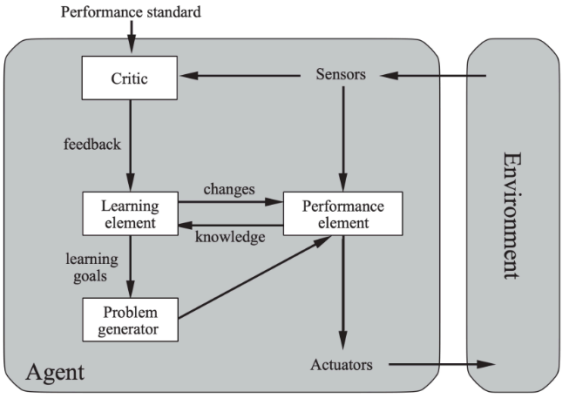
Decision-making involves considering what the globally optimal next action is with regards to opponent(s) predicted moves in competitive scenarios



- Agent also tries to predict how the Opponent will act by maintaining an Opponent Model
- Planning will involve using strategies and techniques to plan its actions
  - Minimax strategy chooses the action that minimises opponents' chances of winning, and maximises the agent's
  - Alpha-Beta Pruning technique skips bad actions when considering individual or sequences of actions to evaluate, optimising the planning

▼ Learning Agents

Learning Agents improve their performance over time by learning from their experiences/history



- Performance element is responsible for choosing what action to do based on the precepts and some strategy
- Critic is responsible for monitoring the performance element by comparing its actions/results to a fixed performance standard, and gives feedback to the learning element
- Learning Element is responsible for improving the performance element (by modifying its strategy) based on the feedback from the critic

    - Note that the Learning Element is not actually a separate module, just a conceptual simplification of the critic and performance element interacting
- Problem Generator is responsible for generating new tasks that the agent can learn, and thereby train/improve itself through

Learning is critical as

- It is impractical if possible to design all aspects of a system by hand
- It is impractical to adapt an agent to new situations by reprogramming it every time

All agents can incorporate learning to improve their performance

▼ Wk 1 — Knowledge Representation

Knowledge often depends and changes on the context

Knowledge Representation refers to how knowledge is structured for reasoning by machines

- A knowledge base is an explicit set of sentences about some domain expressed in a suitable formal representation language
- Sentences express facts or non-facts, and their structure depends on the formal representation language
- All knowledge-based agents have a knowledge base at its core

Fundamental Questions:

- How do we write down knowledge about a domain/problem? → Formal Language → Propositional and First-Order Logic
- How do we automate reasoning to deduce new facts or ensure consistency of a knowledge base?

▼ Representation

▼ Feature-Based Representations

- Knowledge is represented in terms of numerical or categorical features/attributes of objects
    - Abstract real world entities are represented as sets of features/attributes (objects)
- Objects are collections of features that can be together used to form a vector which can in turn be used to classify or infer some property about an object

    - Feature-Based representation contains relational information through the relations between features within objects
- Low to medium level of abstraction
- Easy to reason about since we have objects with clearly defined structures and attributes, allowing reasoning to be straightforward and systematic
- Easy to make generalisations by analysing patterns in features across different instances in the base
- Is memory efficient
- Suitable for machine learning and statistical inference, but not for low-level perception (processing sensory information like sight and sound)

▼ Iconic Representations

- Knwoledge is represented/encoded using numerical (statistical) data structured to mirror the relevant aspects of the real world
    - e.g. faceID works by encoding your face as a matrix of numbers
    - Maintains the visual or spatial form of the objext
- Knowledge is represented through pixels (icons) in a such a way that the structure of the pixels mirrors/captures the relevant aspects of the real world
    - Sort of like reducing a picture to a free-body diagram in physics
- Medium to High level of abstraction
- Requires a lot of memory, but is fast
- Does not generalise well since each instance in the base is closely tied to the real-world features it perceived, and hence very specific
- Designed to match sensory patterns, so suitable for pattern recognition, vision and spatial reasoning, but difficult to perform inferences beyond pattern-response

▼ Symbolic Representations

- Knowledge is represented through a set of symbols (e.g. words, phrases) and logic that together represent objects, concepts and relationships or rules in a formal way
- High level of abstraction
- Suitable for rule-based reasoning and logical inference (complex reasoning), but not low-level perception since this representation is rooted in logic and formal structure

▼ Rule-Based Systems

A production rule has the form “if <condition> then <conclusion>”

- Production rules can often be written to closely resemble natural language
- Capitalisation indicates that something is a variable replaceable by a constant through pattern matching
- Executing a rule may generate a new derived fact
- We say there is a dependency between two rules since the conclusion of one can satisfy the condition of another

Uncertainty in Rules may arise from

- Uncertain evidence → we cannot be certain that the condition is true based on the evidence
- Bayesian Inference: uncertain links between evidence and conclusion
- Fuzzy Logic: vague rule

▼ **Simple Rules Example**

```
rule r1_1
if the employer of Person is acme
then the salary of Person becomes large.

// condition is first line, conclusion is second

// Person has capital 'P', so it is a variable
// replaceable by a constant like felix_lin

// uncertain evidence → if we are not certain that someone works for acme

// fuzzy logic → what exactly is considered large?

// bayesian inference → it is only likely that an acme employee
// has a large salary, not certain (100%)

rule r1_2
if the salary of Person is large
or the job_satisfaction of Person is true
then the contentment of Person becomes true

// there is a dependency between r1_1 and r1_2

fact f1_1
```

```
the employer of felix_lin is acme

derived fact f1_2
the salary of felix_lin is large
```

Inference Networks are structures that represent how rules and facts are connected, and how information flows through these connections to draw conclusions

- Interdependencies among rules define a network
- An inference network shows which facts can be logically combined (and, or, not) to form new facts or conclusions

Rules that make up an inference network can be used to link cause and effect

- “if <cause> then <effect>”

Deduction is a form of reasoning

- goes from general to specific → general rules to specific deduced facts → deduce the effect given some courses
- cause + rule ⇒ effect

Abduction → infer the most likely explanation for a set of obsercations

- Reasoning from effect to cause
- effect + rule ⇒ cause

Induction

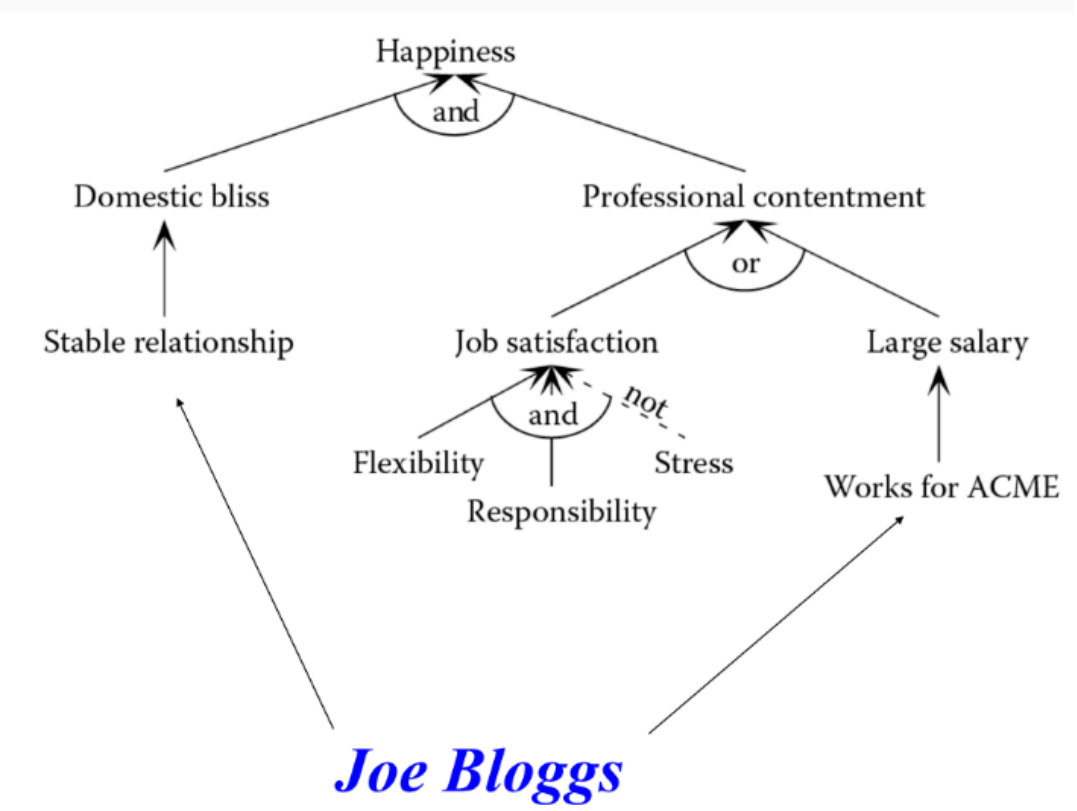
- Reasoning from specific instances to general rules
- cause + effect ⇒ rule

▼ **Inference Network and Deduction, Abduction and Induction Example**

Suppose we added the following rules

```
if Person is in a stable relationship
then Person has domestic bliss.

if Person has a flexible job with responsibilities and no stress
then the job satisfaction of Person is true
```

Then we have the below inference network



Now suppose joe_bloggs is in a stable relationship, and working for acme.

Suppose joe_bloggs is in a stable relationship and working for acme, then we can infer that he is happy

- He works for acme so he has a large salary, and therefore has professional contenment
- He is in a stable relationship so he has domestic bliss
- He has professional contentment and domestic bliss, so he is happy

Suppose joe_bloggs is happy, then we can infer that he has domestic bliss and professional contenment, and that he is in a stable relationship

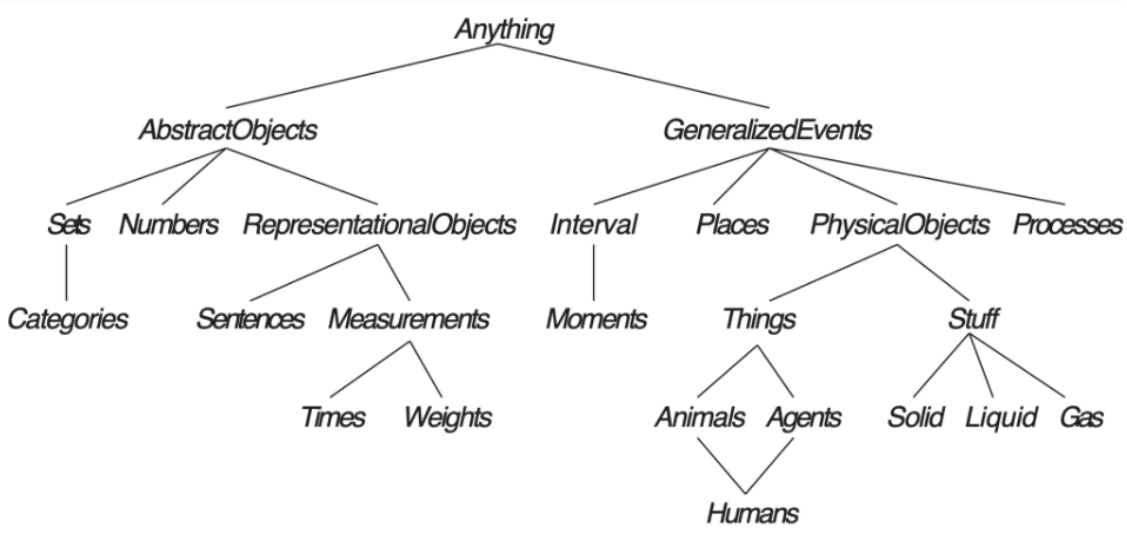
- He is happy, so he must have domestic bliss and professional contentment
- He has domestic bliss, so he must be in a stable relationship
- We cannot infer if he has job satisfaction (and hence anything about his job) or if he has a large salary

Suppose we did not yet have rule r1_1, but had that all employees of acme had large salaries, then we could induct rule r1_1

We must assume that only the facts in the knowledge base or derived from rules are true, and that everything else is false

- It is more accurate to say that “a proof fails” or “a proof succeeds” instead of saying “it’s false” or “it’s true”

▼ Ontological Engineering



Approach to structure and organise knowledge by defining an ontology (hierarchy of categories) that represents knowledge (concepts and relationships) in a specific domain

- higher up categories are more abstract
  - Child categories are specialisations of their parents
  - Parent categories are generalisations of their children
  - Specialisations do not have to be disjoint → humans are animals and agents
- Approach to structure and organise knowledge by defining an ontology that represents concepts and relationships in a specific domain
- We can define categories without defining their specialisations (children) until later if ever
- Organising objects into categories is vital for knowledge representation, as much of the reasoning takes place at the category level since we can make predictions based on the object’s classification instead of the actual instance of the object → we can reasonably predict that an object classified as a car can be driven

- Categories organise and simplify the knowledge base through inheritance
    - If we have the inheritance Food → Fruit → Apple, and all instances of Food are edible, then we can infer that all Fruits including Apples must be edible

Categorising Objects

1. We infer the presence of objects from perceptual input
2. We infer the category membership from the perceived properties of the objects
3. We use information about the category to make predictions about the objects

E.g. If we perceive something that is green, ovoid in shape, has red flesh with black seeds, 30cm in diameter and is in the fruit aisle, we infer that it is a watermelon

- Walks like a duck, talks like a duck → its a duck

Semantic networks

- We have Facts, Objects, Attributes, and Relationships among instances of objects and classes of objects
- Relationships and attributes can be transient (changeable), static (fixed), or default
- Attributes and relationships can be represented as a network known as an associative or semantic network
- Efficient algorithms for inferring properties of objects based on category membership can be used with semantic networks

▼ **Semantic Network Example**

We have the following set of statements, where instance objects are underlined, classes are **bold**, and relationships are coloured

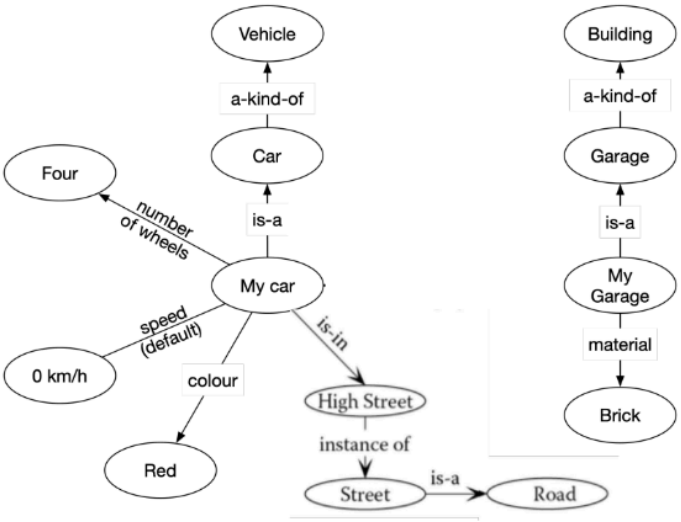
- My car is a **car**
- A car is a kind of **vehicle**
- A car has four wheels
- A car's speed is 0 mph
- My car is red
- My car is in my garage
- My garage is a **garage**
- A garage is a **building**

- My garage is made from brick
- My car is in High Street
- High Street is a **street**
- A **street** is a **road**

is in is a transient relationship as it can change

- By default, my car is in my garage, but later my car is in High Street

Then we have the following semantic network



subclass relationships are shown by a-kind-of (ako) arrow, where the arrowhead is on the super class

instance relationships are shown by an is-a arrow where the arrowhead is on the class

Description logics

- Formal language for constructing and combining category definitions
- Allow for efficient algorithms for deciding subset and superset relationships between categories
- Propositional Logic and First-Order Logic are common description logic formal languages

▼ Propositional and First-Order Logic

Declarative approach to building an agent:

1. Tell the system what it needs to know, then it can ask itself what it needs to do
2. Answers to its own questions should follow from the knowledge base and its inference mechanism

- We need formal language to formally specify how to answer questions and structure the knowledge base

Formal Language

- Natural language is ambiguous, making it difficult to interpret the meaning of phrases/sentences, and also to define and compute inferences
  - "That guy is tall" → what is considered tall?
  - "The boy saw a girl with a telescope" → did the boy see the girl through a telescope, or did he see a girl holding a telescope?
- Symbolic logic is a syntactically unambiguous language
- Propositional and First-Order Logic are common formal languages used for knowledge bases

Propositions are entities (facts or non-facts) that can be true or false

- We use single letters to represent basic propositions → e.g. *P*: Socrates is bald
- We combine propositions into more complex sentences using logical operators and propositional connectives

¬	negation	$\neg P$	"not P"
∧	conjunction	$P \wedge Q$	"P and Q"
∨	disjunction	$P \vee Q$	"P or Q"
→	implication	$P \rightarrow Q$	"If P then Q"
↔	bi-implication	$P \leftrightarrow Q$	"P if and only if Q"

- **NOTE**: reasoning is independent of proposition definitions, and Propositional Logic cannot express relations and ontologies

First-Order Logic can express knowledge about objects, properties and relationships

- Constants refer to specific objects → *a, b, ..., Mary(objects)*
- Variables represent general objects that can take different values → *x, y, ...*

- Functions map objects to other objects → *f()*, *mother_of()*, *sine()* → *mother_of(x)* returns the object that is the mother of *x*
- Predicates represent properties and/or relationships between objects
  - *Person(x), Tall(x)* represents that *x* is a tall person
  - *Mother(x, y)* represents that *x* is the mother of *y*
- Quantifiers specify the scope of variables a sentence is applicable to
  - $\forall$ means for all → $\forall$*x Person(x)* $\rightarrow$ *Mortal(x)* means "for all objects *x*, if *x* is a Person, *x* is mortal → all people are mortal
  - $\exists$ means there exists → $\exists$*x Salary(x) > 10* means "there exists an object *x* whose salary is greater than 10

First-Order Logic Language

- Terms refer to objects → constants, variables and functions are applied to terms
- Atomic formulae are predicates applied to tuples of terms → e.g. likes(Mary, mother_of(Mary))
- Quantified formulae → atomic formulae with a quantifier
  - $\forall x\ likes(x, a)$ → occurrences of x in the formulae are bound by the quantifier $\forall$
  - $\exists x\ likes(x, mother\_of(y))$ → occurrences of x in the formulae are bound by quantifier $\exists$, and y is 'free' (can be any term)

First-Order Logic can also state facts about categories, relating objects to categories or quantify members

- $\in$ in → $B_1 \in Basketballs$ → means B_1 is a basketball
- $\subset$ subclass relation → $Basketballs \subset Balls$ → Basketballs is a subclass of Balls
- We can also say things about common properties of members of a category, and recognise members of a category through their properties
  - $\forall x(x \in Basketballs \Rightarrow Spherical(x))$ → all basketballs are spherical
  - $Orange(x) \wedge Round(x) \wedge Diameter(x) = 24cm \wedge x \in Balls \Rightarrow x \in Basketballs$ → if x is orange, round 24cm in diameter and is a ball, it is a basketball

WEEK 2 — Search

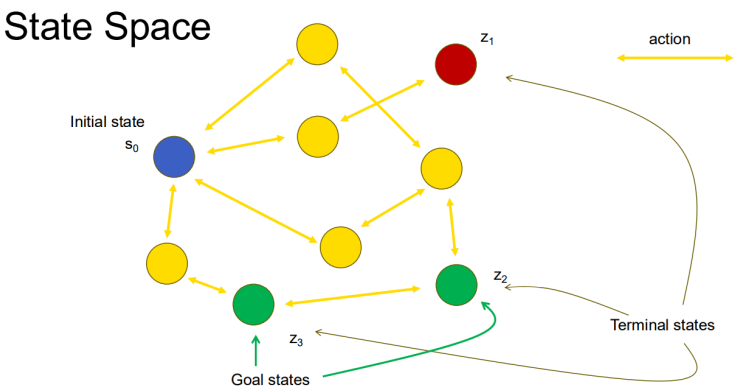
▼ Search

Agents must use search to find a sequence of actions that achieves its goals when no single action will do

- Search is needed for tasks like motion planning, navigation, speech and natural language, machine learning, game playing and any tasks that need some sort of planning

For AI to work, we have to be able to convert real world problems into machine-understandable ones

- Machines need to use precise algorithms that allow them to explore vast possibilities systematically and efficiently
- We describe a problem using a state, action, and transition function that maps a state (and any other information needed) and an action to a new state

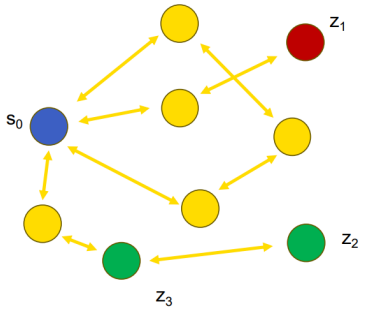


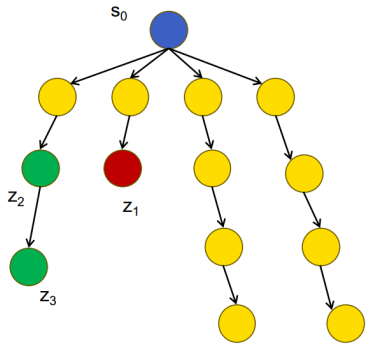
We represent all possible states of a problem in a state space, which is often visualised as a graph

- Each node contains a state and other related information (e.g. legal actions from the state
- Edges between nodes represent an action that can transition one node to another
- note that the graph
- Searching in a state space is essentially finding a path from the initial state to a goal state
    - We also have terminal states → states in which your agent 'dies' or terminates upon reaching → i.e. if there are no more actions to be taken
    - Often terminal states are not explicit states, but states that fulfil some criterion

- We can have multiple goal states

We sometimes represent the state space using a tree where each root to leaf path represents a terminating path in the state space

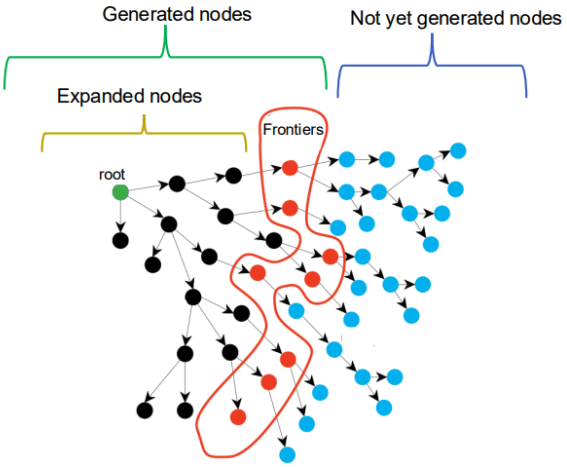




Problem Solving by Graph Searching

- **Expanded nodes**: green and black
- **Frontiers**: red
- Generated: green, black and red
- Not yet generated nodes: blue

Search strategy differ in the way they expand the frontier



However, for many problems, it is impractical or impossible to generate the whole graph representation of the state space

- We do not search all possible paths, but gradually build possible paths by following possible actions starting with the initial state → search algorithms
- To solve problems using search, we generally:
  1. Determine the states, actions and transition function
  2. Convert their definitions into a machine readable format
  3. Choose a search strategy according to the problem

Terminology and Notation for Search Algorithms

- We say a node is generated when it is created and inserted into the queue
- We say a node is expanded when we remove it from the queue and generate nodes from it
- We say an algorithm is complete if it is guaranteed to find a solution in finite time, given that the solution exists
- We say an algorithm is optimal if it is guaranteed to return the optimal solution
- b is the branching factor (degree of each node)
- d is the depth of the shallowest solution
- m is the maximum depth of the search tree
- k is the depth limit

▼ Uninformed Search Algorithms

NOTE: The algo examples are very generic and serve more to show the structure and mechanisms than to be a boilerplate

Uninformed search algorithms are those that follow a strict algorithm without any complex heuristics to eliminate non-solutions from the search space

- BFS, DFS
- Depth-limited search
- Iterative depth-deepening search (IDDFS)
- Bidirectional search
- Uniform cost search (UCS)

	Time	Space	Complete	Optimal
BFS	$O(b^d)$	$O(b^d)$	Yes if b is finite	Yes if all actions cost the same
DFS	$O(b^m)$	$O(bm)$	No	No
Uniform-Cost	$O(b^{C^*/\epsilon})$	$O(b^{C^*/\epsilon})$	Yes if b is finite and steps cost $\geq \epsilon > 0$	Yes
DLS	$O(b^k)$	$O(bk)$	Yes if goal is within the depth bound k	No
IDDFS	$O(b^d)$	$O(bd)$	Yes if b is finite	Yes if all actions cost the same

Breadth-First Search

▼ Pseudocode

```
from collections import deque

def breadth_first_search(start, goal):
    frontier = deque([start])  # Queue for BFS
    explored = set()

    while frontier:
        node = frontier.popleft()  # Get the first node in the queue

        if isGoal(node, goal):
            return node  # Return solution node

        explored.add(node)

        for child in expand(node):
            if child not in explored and child not in frontier:
                child.parent = node  # Set parent to reconstruct path
                frontier.append(child)

    return None  # No solution found
```

- Works same as in 2521 with slight differences
  - We say a node is generated when it is created and inserted into the queue
  - We say a node is expanded when we remove it from the queue and generate nodes from it
- If a problem has breadth b (every node has out degree b) and a solution at depth d, it has $O(b^d)$ time and space complexity (assuming other operations are cost less) → BFS must generate and store all nodes from level 0 to d, and each level b times more nodes than the previous
- Optimal if all actions cost the same → unweighted graph
- Main problem with BFS is space complexity growing exponentially

Depth-First Search

▼ Pseudocode

```
def depth_first_search_recursive(node, goal, explored=None):
    if explored is None:
        explored = set()  # Initialize explored set at the first call

    if isGoal(node, goal):  # Goal test
        return node  # Return solution node

    explored.add(node)  # Mark the node as explored

    for child in expand(node):  # Expand the current node
        if child not in explored:
            child.parent = node  # Set parent to reconstruct the path
            result = depth_first_search_recursive(child, goal, explored)  # Recursive call
            if result:  # If a solution is found in recursion, propagate it upward
                return result

    return None  # No solution found in this branch
```

- Only stores a single path from root to current node, and any remaining unexpanded siblings of nodes in the path → memory efficient
- Memory efficient as nodes can be removed from memory once they have been expanded, and all of their descendants have been fully explored → O(bm)
- Still requires cycle checking (check that node is not already in path before visiting

Depth-Limited Search → DLS

▼ Pseudocode

```
def depth_limited_search(node, goal, limit):
    if isGoal(node, goal):
        return node  # Return solution node

    if limit == 0:
        return None  # Depth limit reached

    for child in expand(node):
        child.parent = node  # Set parent to reconstruct path
        result = depth_limited_search(child, goal, limit - 1)
        if result:
            return result

    return None  # No solution found within depth limit
```

- DFS with a predetermined depth limit l → nodes on level/distance l from the initial-state/node are treated as if they have no successors → treated as leaf nodes
- Optimal/useful when we know a solution isn't deep (doesn't take a lot of actions) → if a solution is deeper than expected, we will miss it
- Can return the solution, a cutoff or a failure → failure = no solution, cutoff = no solution within the given depth
- No need for cycle checking, since infinite loops will be caught by the cutoff
- Avoids searching really deep or infinitely deep paths that are clearly not optimal//solution-yielding

Iterative Depth-Deepening Search

▼ Pseudocode

```
def iterative_deepening_search(start, goal):
    depth = 0
    while True:
        result = depth_limited_search(start, goal, depth)
        if result:
            return result  # Return solution node
        depth += 1  # Increase depth limit incrementally
```

- Attempts all possible depth bounds until a solution is found
- Benefits of DLS and BFS → low memory, optimal and complete → the depth of the solution will be the minimum-depth solution
  - $O(b^d)$ time
  - $O(bd)$ space
- Generally preferred when we have a large search space and the depth of the solution is unknown

Bidirectional Search

▼ Pseudocode

```
from collections import deque

def bidirectional_search(start, goal):
    frontier_start = deque([start])
    frontier_goal = deque([goal])
    explored_start = set()
    explored_goal = set()
```

```
    while frontier_start and frontier_goal:
        # Expand forward search
        solution = expand_bidirectional(frontier_start, explored_start, explored_goal)
        if solution:
            return solution

        # Expand backward search
        solution = expand_bidirectional(frontier_goal, explored_goal, explored_start)
        if solution:
            return solution

    return None  # No solution found

def expand_bidirectional(frontier, explored, other_explored):
    node = frontier.popleft()
    explored.add(node)

    for child in expand(node):
        if child in other_explored:
            return child  # Return meeting node as solution node
        if child not in explored and child not in frontier:
            child.parent = node
            frontier.append(child)

    return None
```

- Simultaneously searches from the initial node and goal node until there is an overlapping path → fast → $O(b^{d/2})$
    - Using hash table to check if a node appears in the other path in O(1) time
- Must store all visited (expanded) states of at least one of the searches so that we can check if a node visited by the other search has been visited by this search→ $O(b^{d/2})$
- Optimal and guarantees the solution path is the shortest path if BFS is used to search from initial and goal nodes
- Rarely applicable to a problem
- Typically uses BFS

Uniform Cost Search

▼ Pseudocode

```
import heapq

def uniform_cost_search(start, goal):
    frontier = []  # Priority queue (min-heap)
    heapq.heappush(frontier, (0, start))  # Add start node with cost 0
    explored = set()

    while frontier:
        cost, node = heapq.heappop(frontier)  # Pop node with lowest cost

        if isGoal(node, goal):
            return node  # Return solution node

        explored.add(node)

        for child in expand(node):
            new_cost = cost + cost_function(node, child)
            if child not in explored:
                child.parent = node  # Set parent to reconstruct path
                heapq.heappush(frontier, (new_cost, child))

    return None  # No solution found
```

- Aka lowest-cost first search
- Sometimes transitioning between nodes has a cost → weighted graphs
- Guaranteed to find minimum cost path in search spaces where all edges have positive costs
- Like BFS, but expands the lowest-cost path, not the closest one → uses a priority queue sorted by cost instead of a Queue
    - Reduces to BFS when all possible actions have the same cost
- Will complete if b is finite and transition $cost \geq \epsilon$, with $\epsilon > 0$
- $O(b^{C^*/\epsilon})$ time complexity, where C^* is the cost of the optimal solution → cost per step is C^*/ϵ
- $O(b^{C^*/\epsilon}) = O(b^d)$ if all step costs are equal

▼ Informed Search Algorithms (and heuristics)

NOTE: The algo examples are very generic and serve more to show the structure and mechanisms than to be a boilerplate

Informed (Heuristic) Search Algorithms are those that use heuristics to make educated guesses that guide the search towards the goal → focus on the path most likely to lead to the goal state

- Uses domain knowledge with their heuristic to search in the direction of where they predict the goal will be → will use problem-specific knowledge
- Implemented using a Priority Queue to store frontier nodes ordered by their evaluated 'likeliness' of leading to the goal state
- Greedy Best-First Search
- A* Search

Selection in Path Building → Heuristics

- Informed search algos use an evaluation function f(n) to evaluate the likeliness of node n leading to the goal state when selecting the node to extend a path by
- They use a Priority Queue to store frontier nodes ordered by f(n)
- A heuristic refers to a rule used to decide which option is best
  - The heuristic must underestimate the actual costs of getting to the goal from the current node
  - h(n) evaluates and returns the heuristic priority of the node n, returning 0 if n is the goal
  - The heuristic function can be found by simplifying the calculation of true cost
- The efficiency of an informed search algo largely depends on how good the heuristic is

Heuristic admissibility

- A heuristic h is said to be admissible if $\forall n\ h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost from n to the goal
  - h is admissible if f(n) never overestimates the cost of using node n
- Admissible heuristics are by nature optimistic, since they think the cost of solving the problem is less than it actually is
- Admissible heuristics can often be derived from the exact solution cost of a simplified or 'relaxed' version of the problem (problem with some constraints weakened or removed)

Heuristic Dominance

- Given two admissible heuristics h1 and h2, if $\forall n\ h2(n) \geq h1(n)$, then we say that h2 dominates h1

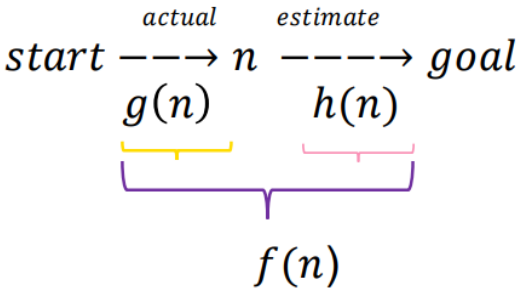
    - h2 is better for search → if h2 is admissible, and greater than h1, then we have that $\forall n\ h^*(n) \geq h2(n) \geq h1(n)$ → h2 is more accurate
    - Also, using h2 will expand fewer nodes
- Sometimes we can combine heuristics in various ways to produce a better dominant heuristic
    - A common example is simply taking the max of two heuristics

Greedy Best First Search → f(n) = h(n)

▼ Pseudocode

```
import heapq

def greedy_best_first_search(start, goal):
    frontier = []  # Priority queue (min-heap)
    heapq.heappush(frontier, (heuristic(start, goal), start))  # Push start node with he
    explored = set()

    while frontier:
        _, node = heapq.heappop(frontier)  # Pop node with the lowest heuristic value

        if isGoal(node, goal):
            return node  # Return solution node

        explored.add(node)

        for child in expand(node):
            if child not in explored:
                child.parent = node  # Set parent to reconstruct path
                heapq.heappush(frontier, (heuristic(child, goal), child))

    return None  # No solution found
```

- Similar to DFS, but we always pick next node according to h(n) → always pick the node that seems closest to the goal
- Not complete → may get stuck in loops or fail to explore some paths
- Not guaranteed to find the optimal path → does not consider the path cost
- $O(b^m)$ time and space complexity, where m is the maximum depth in the search space
- A good heuristic can however reduce time and memory costs substantially



A* Search → f(n) = actual cost of current path + h(n) = g(n) + h(n)

▼ Pseudocode

```
import heapq

def a_star_search(start, goal):
    frontier = []  # Priority queue (min-heap)
    heapq.heappush(frontier, (0, start))  # Push start node with f(n) = 0
    cost_so_far = {start: 0}  # Tracks g(n), cost from start to node

    while frontier:
        _, node = heapq.heappop(frontier)  # Pop node with lowest f(n)

        if isGoal(node, goal):
            return node  # Return solution node

        for child in expand(node):
            new_cost = cost_so_far[node] + cost_function(node, child)  # Calculate g(n)
            if child not in cost_so_far or new_cost < cost_so_far[child]:
                cost_so_far[child] = new_cost  # Update cost
                priority = new_cost + heuristic(child, goal)  # Calculate f(n)
                child.parent = node  # Set parent to reconstruct path
                heapq.heappush(frontier, (priority, child))

    return None  # No solution found
```

- Advantages of Uniform-Cost and Greedy Best-First Search → considers both the cost of the path so far, and the estimated cost to the goal
  - Uniform-Cost aspects minimise h(n)
  - Greedy aspects minimise h(n)
- f(n) hence represents the estimated total cost of the cheapest path from the initial node to n to the goal node → the total cost of the solution extending the current path with n

- Complete unless there are infinitely many nodes n with $f(n) \leq$ cost of solution
  - Complete if b is finite, and h is admissible
- Optimal if h is admissible → if h never overestimates the cost of using node n
- A* is efficient, but may consume lots of memory as all generated nodes must be stored

Iterative Deepening A* Search

▼ Pseudocode

```
def iterative_deepening_a_star(start, goal):
    # Initial threshold is the heuristic value of the start node
    threshold = heuristic(start, goal)

    while True:
        result, new_threshold = search(start, goal, 0, threshold)
        if result:  # Goal found
            return result  # Return solution node
        if new_threshold == float('inf'):  # No more nodes to explore
            return None
        threshold = new_threshold  # Increase threshold for the next iteration

# Depth-limited A* search with a threshold
def search(node, goal, g_cost, threshold):
    f_cost = g_cost + heuristic(node, goal)  # Calculate f(n) = g(n) + h(n)

    if f_cost > threshold:  # If f(n) exceeds the threshold, return the new threshold
        return None, f_cost

    if isGoal(node, goal):
        return node, threshold  # Goal found, return solution node

    minimum = float('inf')  # Track the smallest f(n) that exceeds the threshold

    for child in expand(node):
        child.parent = node  # Set parent to reconstruct path
        new_g_cost = g_cost + cost_function(node, child)
        result, temp_threshold = search(child, goal, new_g_cost, threshold)

        if result:  # Goal found
            return result, temp_threshold

        if temp_threshold < minimum:
```

```
            minimum = temp_threshold

    return None, minimum  # Return the smallest f(n) that exceeded the threshold
```

- A low-memory variant of A* that performs a series of DFSs first, but cuts off each search when f(n) exceeds the current threshold → intialise with f(initial node)
- Lower memory since it doesn't have to store the entire frontier → at the cost of performance since each iteration has some repeated work from the previous iteration

WEEK 3 — Neural Networks

5612

▼ Fundamentals

Neural Networks are ML programs/models that mimic how biological neurons identify phenomena and draw conclusions to make decisions

- Neurocomputing refers to a non-algorithmic paradigm for information processing → focuses on learning, adapting, and distributed parallel processing
    - Neural Networks learn from data by adjusting their internal parameters (weights and biases) to increase accuracy when they are exposed to new data
    - Inputs are distributed over many interconnected artificial neurons (nodes), and multiple neurons process multiple inputs simultaneously in parallel

Classifiers predict the discrete labels for entities

- Determines the likely class an entity is a member of → e.g. classifies characters from an image into English, Chinese, Greek, etc.
- Approximate a discriminant function that decides the class membership of an input, and this function is adapted through the learning process

$$w_{ij} = w_{ij} - \alpha \frac{\partial Loss}{\partial w_{ij}}$$

Regressors predict continuous (numeric) values from input data → e.g. predicting the price of a house given its area, number of rooms, and neighbouring houses' prices

- Approximate the generating (unknown) function that represents the relationship between input and output variables

- Neural Network learns to map input variables to a continuous output variable, aiming to increase accuracy (minimise difference between predicted and actual values)

Artificial Neurons are automata characterised by an internal state, input signals, a bias and an activation/transfer function

1. Neuron receives inputs from other neurons or external sources, and each input has an associated weight
2. Calculates the net input → weighted sum (input * input_weight) + bias
3. Activation function takes the net input and determines the output (may involve using some threshold)
4. Output is passed to the next layer

- The activation function introduces non-linearity into the system, allowing ANNs to solve complex problems
- The activation level of an artificial neuron refers to the output value
- We have input edges going to a neuron, and output edges going from a neuron, and all edges have weights
- Weights represent the significance of an input, and can be positive or negative → will change as the ANN learns through training

McCulloch-Pitts model of artificial neurons

- Inputs and output are binary (0 or 1), and the neuron has a threshold value θ
- Inputs can be excitatory or inhibitory → amplify or suppress the activity of downstream neurons
    - Excitatory have weight 1, and inhibitory have weight -1 → excitatory adds 1 to the weighted sum, inhibitory subtracts 1 from it if the input is 1
- Transfer function: weighted sum
- Activation function: typically if weighted sum ≥ θ, output 1, else output 0
    - `sum > threshold ? 1 : 0;`
- Note that it cannot handle more complex tasks

Learning in Neural Networks

- Neurons or a neural net adjusts connections (weights) to obtain the intended output, or output that meets a certain criteria
- Hebbian Learning (1949) → when neuron A persistently activates neuron B, the connection between the two neurons becomes stronger

    - The weight of inputs from A to B increases so as to increase likelihood of activation since we know ‘know’ from experience that B will likely activate if A does
    - Associative memories → increasing weights to strengthen connections makes it easier for the activation of B to be determined when A is activated
- Back-propagation involves passing inputs forwards through the ANN, then calculating the error (difference between produced and target output) and passing it back through the ANN to adjust the weights and increase accuracy using optimisation algorithms

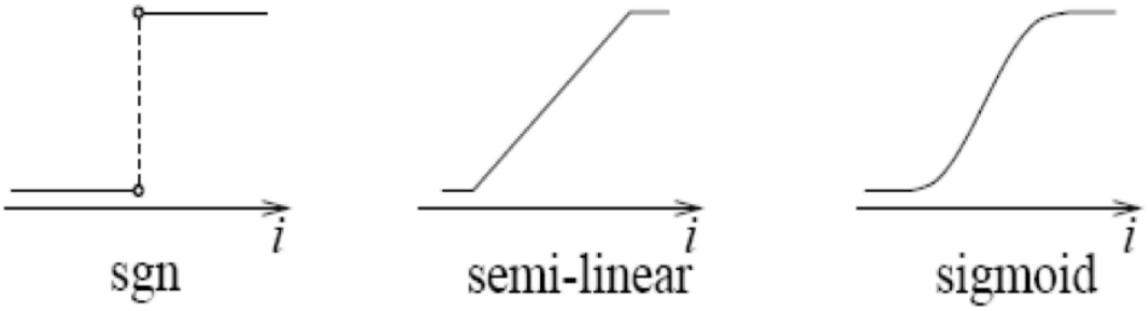
Artificial Neural Networks (ANNs) are computational models utilising architectures and functions that loosely model the brain

- We characterise ANNs through their:
    - Number of neurons
    - Interconnection architecture
    - Weight values
    - Activation and transfer functions
- ANNs consist of layers of interconnected artificial neurons (nodes) that function as processing units
    - Input layer receives input data from external sources
    - Hidden layer processes inputs by applying weights and activation functions between nodes
    - Output layer produces the final prediction or classification based on the data received at the input layer and processed by the hidden layer
- Forward-propagation → inputs are fed into the network and passed forward through transfer and activation functions to produce the output
- Neurons work in parallel to accomplish a global task for the ANN

ANNs work by learning, or by simulation or recognition

- Learning involves adapting the weights, architecture and transfer and activation functions in an ANN through back-propagation through input data
    - We feed data sets into the ANN, and use some correction method after each set of inputs to train it
- Reinforcement learning uses a reward and penalty system to adjust
- Supervised learning involves giving the ANN labeled data where each set of inputs is paired with the correct output

    - Aim is to have the ANN learn a mapping from inputs to outputs
- Unsupervised learning involves giving the ANN unlabeled data where the correct output for each set of inputs are not known
    - ANN must find patterns of relationships in the data by itself, and learn to group data into clusters/categories based on some criteria or similarity



Four main activation functions $g(s)$

- Step function → usually used for classification tasks
- Signum (sgn) → returns -1, 0 or 1 to represent the 'sign' and 'side' of given inputs
- Semi-linear/ReLu → Has a linear component that is derivable, and hence the activation function can be derived and used efficiently with back-propagation
- Sigmoid → $f(x) = \frac{1}{1+e^{-x}}$ → Useful for converting numbers to probabilities → "probably a Whale Shark, maybe a Whale Shark, unlikely to be a Whale Shark"
- Note that the derivative of certain activation functions can be reduced to a formula involving the function
    - Sigmoid

$$g(s) = z$$
$$g'(s) = z(1-z)$$

    - Tanh
    -

▼ Single and Multi Layer Perceptrons

Single-Layer Perceptron (SLP)

- Can perform simple linear classification tasks
    - Generally only applicable for linearly separable classes

    - Most efficient and sensible when we have two classes → classifies input entity by separating two categories with a straight line (hyperplane or decision boundary)
    - If we have multiple linearly separable classes, we usually train multiple binary SLP classifiers and use their results together
- ANN has an input and output layer, but no hidden layers → inputs represent the features, output represents the predicted class of the inputs
    - Weights of edges from input to output nodes control/correlate to how much influence each input has on the classification
    - We don't really consider the input layer as a real layer since there are no artificial neuron type computations in that layer → the single layer is the output layer
- If we have a function g_i that returns the confidence of it belonging to class s_i for each s_i, the classification rule for an entity u is $u \in s_i$ iff $g_i(u) > g_j(u)$ where $i \neq j$, and for all i, j pairs
- If we have a binary classification, we can use a two layer ANN with a step function (`wsum > theta ? 1 : -1`) for the sole output node
- Learning rule involves starting with small random weights, and adjusting them based on the error for sets of input data
- Perceptron convergence theorem → for any data set that is linearly separable, the perceptron learning rule is guaranteed to find a solution in a finite number of iterations

Multi-Layer Perceptron (MLP)

- Most of the time we are talking about MLPs
- Artificial Neurons in the network are organised into successive layers that each take input from the layer before it, and pass outputs to the layer after it
    - No internal connections within layers → no nodes in the same layer are connected to each other
    - Feedforward network → nodes can only transfer data/signals forwards → no cycles
    - At least one hidden layer
- Structure
    - At least one hidden layer
    - Nodes are fully connected → every node in one layer is connected to every node in the next layer
    - Nodes use non-linear sigmoid activation functions

    - Nodes may have an associated bias term that is added to the weighted sum of inputs to shift the output of the activation function
        - Changing the bias term will shift the activation function horizontally
        - Bias term allows data (in weighted sum form) to be uninfluenced by the origin, ensuring nodes can model data when inputs are zeroes or require an offset → can model data properly
        - Note: if all inputs are 0 the activation function may never fire (produce non-zero outputs) without the addition of some constant bias term
- Can solve non-linearly separable problems and act as universal approximator (can approximate any continuous function to a desired level of accuracy) given enough nodes

Back-propagation optimisation algorithm → Gradient Descent

- The error E is calculated using a Loss Function → e.g mse

$$E = \frac{1}{N}\sum(target - predicted)^2$$

    - Where d is the desired output (target), y is the actual output produced, and N is the number of samples
- We then update the weight w of an edge using the formula

$$w = w - \alpha\frac{\partial E}{\partial w}$$

    - Where α is the learning rate → constant between 0 and 1
    - We gradually change the weight in the direction of the gradient →
- Since we are using (partial) derivatives, back-propagation only works with functions that are continuous (smooth when graphed) → replace discontinuous step functions with continuous ones like sigmoid or tanh

▼ Neural Network Design

Steps to Design a Neural Network

1. Exhaustive Analysis of the System
2. Pre-processing
3. Design the Neural Model
4. Training
5. Generalisation

Exhaustive Analysis of the System involves understanding the problem and system requirements in depth

- Identify inputs and outputs
- Consider other models
- Consider whether we have correct and sufficient data to train a neural model

Pre-Processing involves increasing the quality of the data for training

- Neural networks depend on data for training, and so the quality and quantity of data matters greatly
  - Quality → the degree to which the available data represents the function being approximated → how relevant and accurate the data is for the classification or regression task
- Data must be cleaned → remove or handle missing, noisy or irrelevant data points, and address corrupt or inconsistent data to prevent bias in the model
- Data needs to be normalised so that data for different features can be compared and considered fairly
  - Each node in the input layer represents a feature, and we want features to have the same influence over the output, so we need to normalise them
  - Suppose an input is height, and another is num pets, we don't want height to dominate the output, so we scale both down to a common range
  - It is necessary to perform the corresponding denormalisation at the output stage
- Data may sometimes need to be transformed to make it suitable for NNs → e.g. picture of a face to matrices for facial recognition

Designing the Neural Model involves determining the architecture and parameters (weights and biases) of the NN

- Choosing the number of hidden layers, and nodes per layer
  - General rule of thumb is that the number of connections (weights) between nodes should be less than the number of samples in the data set divided by 10
  - The number of weights in an MLP with Ni input nodes, Nh nodes in hidden layers, and No output nodes is given by $(Ni + 1) * Nh + (Nh + 1) * No$
    - +1 to account for each node's bias
  - If we had 3 inputs, 4 nodes in each hidden layer, and 2 outputs, we have 26 weights, and so we should have at least 260 samples in the dataset to train the network and calibrate weights

- Choosing activation functions for nodes, and a loss function and optimisation algorithm for the overall NN
- What to initialise weights to

Training involves determining how to calculate error and back-propagate

- Bias arises in NNs when the model is too simplistic to capture underlying data patterns
  - Carefully and selectively increase the number of nodes in hidden layers, or the number of hidden layers to increase the complexity of the model
  - Ensure there is sufficient training data
- Overparameterisation occurs when the NN has too many connections/parameters/weights relative to the amount of training data available
  - Model may become too flexible and learn noise instead of general patterns from the training data
  - Selectively prune the number of nodes in the hidden layer, and hidden layers
  - Increase the size of the training data
- Overfitting occurs when the model performs well on training data, but poorly on new data
  - Learns specific patterns in the training data, but fails to generalise
  - Use a training set and a test set, and stop training early when the test validation error starts to increase while the training error decreases
  - Selectively prune the number of nodes in the hidden layer, and hidden layers
- Cross-validation involves splitting the dataset in two, and training the model on one part and testing it on the other to ensure that it can generalise, repeating multiple times
  - Some methods may involve training multiple models and selecting the one with the least error on the test set

Generalisation refers to how well the NN can perform with unseen data → how well its learnt patterns generalise on other data

- testing generalisation often involves a smaller third set of data to be tested on
- The generalisation set should also be representative of the phenomenon being modelled by the training and test sets

▼ Neural Network Architecture

Two main network structures

- Feed-Forward Network → explained before
  - Nodes receive input upstream and deliver output downstream → only goes in the forward direction → no loops → processes data in a single pass
  - Nodes have no internal state besides weights and biases
- Recurrent Network
  - ‘time steps’ → the point at which one element of a sequence is fed into the network
  - Feeds outputs back into its own inputs → ‘recursion’ → processes data across multiple ‘time steps’ instead of in one pass
  - Nodes maintain a hidden state (short term memory) which is updated at each time step based on the current input and previous hidden state → feedback loop → allows learning from past inputs
  - Activation levels of the NN form a dynamic system → can reach a stable state, exhibit oscillations or chaotic behaviour
  - Response of an RNN to an input depends on its hidden states

Deep Learning Architectures

- NN models with multiple layers that form a hierarchical model are known as deep learning models
- Convolutional NNs are specialised for vision tasks
- RNNs are used for time series → sequential data processing

Week 4 — Reinforcement Learning

▼ Basics

Reinforcement Learning (RL) involves learning by interacting with the environment

- Agent must be able to sense the environment’s state and perform actions to affect these states
- Bases decisions on a reward → feedback (signal) providing information about how good or bad the action was in terms of achieving the agent’s goals
  - Rewards can be positive, negative or zero
  - Rewards are how we communicate what we want to achieve to the agent, NOT how we want it achieved
  - If we set rewards to communicate how we want it achieved, the agent will learn to maximise subgoals → e.g. taking opponent’s pieces instead of winning the

game in chess

- Agent must select actions based on the rewards received and continually refine its decisions over time to maximise cumulative rewards
  - Goal is for the agent to learn a policy that maximises cumulative rewards over time, NOT just immediate rewards
  - Training samples are presented one at a time, and the reward after performing an action on a sample influences the action performed for the next sample
- **NOTE**: Most agents observe their influence on the environment, but reinforcement learning is specifically concerned with learning a good policy through trial and error
  - Some agents can directly employ RL methods

Policy refers to the agent's strategy for choosing actions at each state

- Policy defines the agent's behaviour → informs the agent on how to act in a particular situation
- Can be deterministic (maps state to action), or stochastic (maps state to action chosen from a probability distribution over actions)
- Policy is learned and updated as the agent gains experience from the environment

Reward Function evaluates the reward of an action

- Maps each perceived state (or state-action pair) to a number representing the reward
- Reward Function can be stochastic → chosen 'randomly' from a probability distribution

Value Function provides a measure of the long-term desirability of states

- Estimates how good it is for an agent to be in a particular state, taking into account the expected future rewards
- Helps the agent distinguish between short-term rewards and long-term success
- Agent inputs the result of its Value Function into its policy, which determines its next action

RL Agents may also have a model of the environment

- Model-based reinforcement learning → uses a model of the environment to represent the agent's understanding of how the environment works
- Model is used in planning → to predict resultant states and rewards of actions

In simpler non-associative tasks there is no direct relationship between actions and states

- In RL problems, the environment has multiple states the agent can be in → agent must learn which actions to take in different states to maximise its long-term rewards
- If the agent's actions affect both the next state and reward, we have a full RL problem where the agent needs to choose actions based on the current state, and also anticipate how its actions will affect future states and rewards

▼ Exploitation vs Exploration

Exploitation is the process of exploiting current knowledge (what it has learnt) to greedily select the action that will maximise the immediate reward

Exploration is the process of trying new actions to gather more information about the environment

- Action chosen may not lead to immediate rewards
- Helps agent discover potentially better actions that lead to higher rewards in the long run
- Agent deliberately chooses actions they are less certain of or have not tried frequently to explore different possibilities

Agents must balance exploitation and exploration to learn an optimal policy for maximising long-term rewards

- Too much exploitation can lock the agent in a greedy approach and miss globally optimal courses of action
- Too much exploration can mean the agent wastes time trying many suboptimal actions, not using the good strategies it already learnt → slow performance and learning

▼ Action-Value Estimation and Action Selection

Action-Value Estimation is the process of estimating how good it is for an agent to take a particular action in a given state

- $q_*(a)$ is the real/true action value → the expected value of action a
- $Q_t(a)$ is the estimated action value → the current estimate of the value for taking action a and time step t
- The simple estimation method is to take the average of rewards R_i for a given action a that has been taken K_a times

$$Q_t(a) = \frac{R_1 + R_2 + ... + R_{K_a}}{K_a}$$

- K_a is the number of times action a has been taken
- if $K_a = 0$, $Q_t(a)$ is defined with an arbitrary (initial) value (e.g. 0)
- if $K_a \to \infty$, $Q_t(a)$ converges to $q_*(a)$

Incremental Estimation

- Calculating $Q_t(a)$ through the simple estimation method can be very space and performance expensive → requires all past rewards for all actions to be stored
- Incremental Estimation Method involves storing only the estimate, and updating it for an action when we take that action

$$Q_{t+1}(a) = Q_t(a) + \frac{1}{K_a}(R_{t+1} - Q_t(a))$$

-

Greedy Action Selection Method

- Selects the action A_t^* with the highest estimated value

$$Q_t(A_t^*) = \max_a Q_t(a)$$

Epsilon-Greedy Action Selection Method

- Select the best action most of the time, and sometimes (with a small probability of ϵ) a random action (exploration)
- $Q_t(a)$ converges to $q*(a)$ with probability $q - \epsilon$

Softmax Action Selection Method

- More probabilistic approach → the probability of selecting an action is proportional to its value estimate

$$P(a) = \frac{e^{Q_t(a)/\tau}}{\sum_{i=1}^{n} e^{Q_t(i)/\tau}}$$

  - Temperature parameter τ controls the randomness of action selection
  - Large τ values make actions equally likely
  - Small τ values make actions with higher values more likely
- Calculate the probability of each possible action using the above formula, then add their probabilities until you get a cumulative probability greater than the randomly

generated number → add action probabilities in the same order the actions are defined/given

- Avoids cases where the worst actions are very very bad → makes the chance of selecting them very low
- The temperature parameter τ (tau) controls the randomness of action selection:
  - Lower τ makes the selection more deterministic (favoring higher-valued actions)
  - Higher τ makes the selection more uniform (closer to random)

▼ Agent-Environment Interface and Tasks

Agent-Environment Interface describes the interaction between agent and environment

- Agent is the learner and decision-maker
- Environment is everything external to the agent that the agent interacts with
- Reward R, set of states S, action A_t, actions available in a state $A(S_t)$, policy π_t where $\pi_t(a|s)$ is the probability of action a in state s
- RL methods detail how an agent updates its policy as a result of its experience

Episodic Tasks are those where the agent's experience is divided into distinct episodes

- Each episode has a clear beginning, end and finite duration, and the environment is reset to its initial state after each episode
- The (finite-horizon) return G_t in an episodic task is the total cumulative reward the agent collects during the episode, which it aims to maximise

Non-Episodic (or Continuing) Tasks are those where the agent interacts with the environment indefinitely

- No clear episodes, terminal states or natural resetting points
- Cannot define the return in terms of finite time steps since it is infinite
- We define the return as the discounted return, where future rewards are discounted by a discount factor $\gamma \in [0, 1]$ to reflect that immediate rewards are more valuable than distant future rewards
  - If $\gamma = 0$, the agent only cares about immediate rewards and ignores future ones (greedy)
  - If $\gamma \approx 1$, the agent values future rewards almost as much as immediate ones
- The discounted return starting from time step t is:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

- Discounted return allows the agent to balance short and long term rewards

Unified Notation is a standardised way of representing key elements of the RL framework

▼ Markov Property

The Markov Property is that a state should retain compact information about the past that is needed to make optimal decisions

- Agent should not be blamed for not knowing something it has never been shown, but should be blamed for forgetting relevant encountered information
- Markov Property ensures that the future only depends on the current state (and action), and not on past states → we don't need to look at the past sequence of states, just the compact summary of the past in the current state
- Markov Property cannot always be fully satisfied
- RL tasks with the Markov Property are called Markov Decision Processes (MDPs)
- MDP: <S, A, δ, r>
  - S is a finite set of states
  - A is a set of actions
  - δ is the transition function $S \times A \rightarrow S$
  - r is the reward function $S \times A \rightarrow \mathbb{R}$

▼ Value Functions and Temporal-Difference

Value functions estimate how good it is for the agent to be in a given state in terms of future reward or expected return

- $v_\pi(s)$ or $V^\pi(S)$ is the expected return when starting in state s and following policy π

$$v_\pi(s) = E_\pi[G_t | S_t = s]$$

- $q_\pi(s, a)$ or $Q^\pi(S, A)$ is the value of taking action a in state s under policy π

$$q_\pi(s, a) = E_\pi[G_t | S_t = s, A_t = a]$$

- Note that $E_\pi[X]$ is the expected value of X under policy π
- Value functions $v_\pi(s)$ and $q_\pi(s, a)$ can be estimated from experience
- We use parameterised function approximators to keep track of $v_\pi(s)$ and $q_\pi(s, a)$ if there are many states → it would otherwise be impractical

- We use * to denote maximal/optimal return/value → $v^*(s)$ and $q^*(s,a)$
- We simplify RL problems by assuming that actions are deterministic

$$V^*(s_k) = R + \gamma V^*(s_{k+1})$$

- optimal state-value function $V^* : S \to R$ ^^^
- s_k is the current state, s_{k+1} is the next state following the optimal policy

$$Q(s_k, a) = r(s_k, a) + \gamma * \max\left(Q(s_{k+1}, \_)\right)$$

$Q(s,a) = r(s,a) + \gamma V^*(s')$

- $s' = \delta(s,a)$ is the succeeding state, assuming the optimal policy
- Differs from state-value in that it takes an action as well
- The Q-value for an action a in a state s is the immediate reward $r(s,a)$ for the action + the discounted value of following the optimal policy after that action → the expected cumulative reward
- Agent uses Q-values to decide what actions to take in a given state → chooses the one with the maximum Q-value

Temporal Difference (TD) Predictions refer to the agent's ability to make predictions about the future based on its current state and actions

- TD Predictions are typically about the expected future rewards or returns
- TD Methods are used in RL to learn value functions → uses Monte Carlo and dynamic programming approaches
  - TD Prediction focuses on learning state-values
  - TD Control extends to learning action-values (how good it is to take a particular action in a given state)
- TD Methods update the state-value function incrementally after each step/action → Monte Carlo methods have to wait for the entire episode to end first
- TD(0) is the simplest TD method → it updates the value of a state immediately after observing a reward and the next state

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

  - The stuff in the square brackets [] is called the TD error
  - The learning rate α essentially how much we are updating it by each time
  - Recall that $\leftarrow$ means assignment update (equiv to `vst = vst + ...` in code)

    - Note that R_{t+1} is the reward for going from S_t to S_{t+1}
- TD approximations can be on-policy or off-policy
    - On-Policy methods learn the value of a policy by interacting with the environment using the same policy that the agent is currently following → greedily changes the current policy towards the optimal one
    - Off-Policy methods learn the value of an optimal policy while following a different one → allows the agent to improve the policy it is not currently following

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = ($
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

SARSA (State-Action-Reward-State-Action) is an on-policy TD control algorithm

- Updates the action-value function based on the current policy that the agent is following

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
    $S \leftarrow S'$;
  until S is terminal

Q-Learning is an off-policy TD control algorithm

- Learns the optimal action-value function regardless of the agent's current policy
- Agent can follow any policy, but the action-value function is updated based on the assumption that it is following the optimal policy
- One step Q-learning

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a') - Q(S_t, A_t)]$$

Actor-Critic Methods refer to a class of algorithms that combine the benefits of policy and value-based methods

- Actor-Critic methods approximate the policy, and the agent uses a parameterised policy function that maps states to actions
- typically on-policy
- Actor is responsible for selecting actions and structures the policy, representing it as a probability distribution over actions given the current state
- Critic must learn and critique the followed policy → estimates the value function and calculates the TD error, providing them as feedback to the actor to update the value function and policy

Week 5 — Optimisation Methods

▼ Optimisation Methods

Optimisation methods are critical in improving a model's performance

- They work to minimise or maximise an objective function (e.g. loss function) in predictions
- Help AI models learn the best parameters to achieve accurate results

Gradient Descent involves adjusting parameters in the opposite direction of the gradient to find a minimum or maximum of a function

- Very easy to implement and generally works well, but is relatively slow and inefficient, and may get stuck on local minimums instead of global ones, leading to suboptimal results
- Designed to effectively handle nonlinear machine learning problems (including NNs)
- Improved Variations of Gradient Descent
  - Momentum → adds a fraction of the previous movement (update) to the current movement → faster and smoother convergence, and escapes local minima, but the momentum parameter must be chosen carefully/tuned
  - Stochastic Gradient Descent → computes the gradient using only one (or a small subset) randomly selected sample in each iteration → faster, but higher variance (noise) in updates, causing oscillations and making it harder to converge precisely

    - Adaptive Moment Estimation (ADAM) → adapts individual learning rates for each parameter (weight) through momentum and variance scaling → faster convergence and robust to noisy data (less oscillations), but may struggle to converge to the exact optimal point, is sensitive to the learning rate, and is computationally expensive

▼ Complexity, Problem Class and Search Space

In AI, the complexity of an algorithm is a measure of the amount of resources it consumes

- Time and Space (memory and drive) are the main resources we care about

A P problem can be solved in polynomial time on a deterministic computer

- Can be solved quickly
- Are a subset of NP

An NP problem cannot be solved in polynomial time on a deterministic computer

- AKA intractable problems
- Can be verified quickly, but not solved

An NP-Complete problem is one that we don't know if they are tractable or not

- We do not have any polynomial algorithms for them, but some have exponential time complexity
- Are a subset of NP
- All NP-C problems are reducible to each other
- Extremely difficult to solve

We follow the hypothesis that P ≠ NP → otherwise it would all just be one class P

For NP-C problems, we have different approaches to finding solutions that are 'good' or 'close' enough

- Approximation Algorithms → quickly find solutions that fall within a certain range of error → sufficient in practical scenarios where exact solutions aren't critical → but not all NP-C problems have good approximation algos
- Heuristics and Metaheuristics → algorithms use some heuristic or metaheuristic method to try and find a solution → performs reasonably well in many cases, and are

generally fast → but they lack a measure or guarantee about the quality of the solution (how close it is to the optimal solution)

- Genetic Algorithms → algorithms produce and improve many possible solutions until evolving one that is possibly close to the optimum → no guarantee of the quality of the solution despite improving

Search Space refers to the set of all possible solutions for a given problem

- Represents all possible configurations/combinations of variables that could potentially solve the problem
- Feasible solutions are solutions within the search space that satisfy all constraints of the problem
- Constraints are the rules or conditions that must be satisfied for any solution to be considered valid
  - they restrict the set of possible solutions from the entire universe of solutions U to a smaller set of feasible solutions S
  - can be strong (must be satisfied) or weak (merely recommended)
- Solution Space/Feasible Region S is a subset of the search space containing all feasible solutions to the problem
- Objective Function is a function that evaluates how good or bad a solution is → goal of an optimisation problem is to maximise or minimise the value of the objective function

Optimisation Problem $O(S, f)$

- Find an optimal solution within the feasible Solution Space S, such that it maximises or minimises the objective function f
- x_0 is an optimal solution in a minimisation problem iff $f(x_0) \leq f(x) \quad \forall x \in S$ → use $\geq$ instead for maximisation

Neighbourhood Search Procedures are optimisation techniques used to solve complex problems by iteratively searching the neighbourhood of a current solution for a better one

- The set of all possible solutions that can be reached from the current solution by applying one movement/transformation defines its neighbourhood
- Not all movements/transformations lead to feasible solutions → we only consider those that maintain feasibility when searching

1. Generate an initial solution

2. Determine what movements/transformations can be made on the current solution to define its neighbourhood
3. Make a movement/transformation to move to a new solution
4. Evaluate the new solution's quality using the objective function
5. Repeat steps 2-4 until the stopping criterion is met
6. Return the best solution found

Neighbourhood Search Procedures can take different approaches with constraints

- Restrict exploration to the feasible region
    - infeasible solutions are not evaluated, saving computation time
    - guarantees that the final solution found is always feasible
    - May result in inefficiency, as optimal solutions near the boundaries of the feasible region can be harder to reach without exploring the infeasible
- Complete exploration of the solution space
    - may lead to more effective exploration of the search space, increasing
    - time is wasted evaluating infeasible solutions
    - no guarantee that the final solution found is feasible

We also have 3 strategies for restricted exploration of the feasible region

- Rejection strategy → any infeasible solutions generated during the search are ignored
- Repair strategy → if an infeasible solution is generated, a repair operator is applied to transform it into a feasible solution → often based on heuristics
- Prevention strategy → solution representation and movement/transformation operations are designed such that infeasible solutions are never generated → requires more design effort, and the approach is problem-specific

A common scheme for complete exploration of the solution space is to use penalty-based strategies

- A penalty function is added to the original unconstrained objective function

$$Min(f'(x)) = f(x) + w * P(x)$$

    - $P(x)$ is a penalty function measuring the extent to which the constraints are violated (0 if x is feasible)

- w is a weighting coefficient controlling the balance between minimising the original objective function and reducing constraint violations

Metaheuristics

Metaheuristics are strategies for solving optimisation problems through a 'guided' search of the solution space

- Solution representation must include all information necessary for their identification and evaluation
- Search involves generating a sequence of points in the space, where each point is obtained from the previous one through a series of movements/transformations
- Guides the search so that high-quality (often not exact) solutions can be found efficiently → provide guidelines for obtaining paths that yield high-quality solutions while also ensuring adequate efficiency → balance between exploration and exploitation

note:

- stopping criterion is typically when a set number of iterations is completed, or when improvement plateaus

Memoryless Metaheuristics do not use or maintain any explicit memory of past search information

- rely solely on the current solution and its neighbourhood to make decisions about the next movement/transformation
- focus on exploration by using randomised or stochastic search techniques

Simulated Annealing is a memoryless metaheuristic

- Used to find near-optimal solutions for combinatorial optimisation problems → in scenarios where the problem has many good solutions, but only a few global optimum
- Occasionally moves to a worse solution to escape local optima based on some chance (temperature)
- Becomes more selective (temperature cools/lessens) over time → algorithm focuses on refining the best solutions as it progresses → moves from exploration to exploitation
- Uses a cooling schedule to control how much the temperature cools by each iteration → cooling schedule is critical to balancing exploration and exploitation

Algorithm 1 Simulated annealing optimisation method.

Require: Input(T_0, α, N, T_f)

$T \leftarrow T_0$
$S_{act} \leftarrow$ generate initial solution
while $T \geq T_f$ **do**
  for cont $\leftarrow 1$ TO $N(T)$ **do**
    $S_{cond} \leftarrow$ Neighbour solution [from (S_{act})]
    $\delta \leftarrow f(S_{cond}) - f(S_{act})$
    if rand(0,1) $< e^{-\delta/T}$ **or** $\delta < 0$ **then**
      $S_{act} \leftarrow S_{cond}$
    end if
  end for
  $T \leftarrow \alpha(T)$
end while
return Best S_{act} visited

```
// begin with
// T_init and T_fin → the initial and final temperature
// alpha → the cooling rate in (0, 1)
// N the number of iterations to go for
sim_annealing_opt(T_init, alpha, N, T_fin) → Soln {
    let T = T_init
    // generate an initial solution
    let S_curr = gen_init_soln()
    // while current T greater than final T
    while T >= T_fin {
        // generate neighbour of current soln
        for _ in 0..N {
            let S_new = gen_neighbour_from(S_curr);
            let delta = f(S_new) - f(S_curr)
            // if random exploration chance, or if S_new is better
            if rand(0, 1) < e ^ (-delta/T) || delta < 0 {
                S_curr = S_new
            }
        }
        // cool the temperature
        T = alpha * T
    }
```

```
        return S_curr
}
```

- $T_0,\ T_f$ is the initial and final temperature → loop until current temp T is less than T_f
- S_{act} is the current solution
- $N(T)$ is the number of iterations (neighbour solutions) to evaluate for temperature T
- δ is the difference between the objective function evaluation of the current and neighbour solutions → if $\delta < 0$, the neighbour is better than the current

1. Generate an initial solution
2. Explore the current solution's neighbourhood
    a. Evaluate a neighbour
    b. accept (move to) the neighbour if it is better, or if it is worse but the exploration chance based on the temperature activates
    c. Repeat the a and b for all neighbours
3. Decrease the temperature according to some cooling schedule function
4. Repeat 2-3 until the current temperature is less than some set threshold (the final temperature)
5. Return the best solution visited

Memory-based Metaheuristics use past information or historical data to guide the search process

- certain aspects of the search, such as the best solutions found so far or promising regions in the space, are stored
- memory allows the algorithm to make informed decisions and adapt its search strategy

Algorithm 1 Tabu search optimisation method.

1: $s_0 \leftarrow$ generate initial solution
2: $s_{best} \leftarrow s_0$
3: tabuList $\leftarrow \{s_0\}$
4: **repeat**
5: $\quad \{s_1, s_2, ..., s_n\} \leftarrow$ generate neighbourhood from (s_{best})
6: $\quad s_{candidate} \leftarrow s_1$
7: $\quad$ **for** i $\leftarrow$ 2 TO n **do**
8: $\quad\quad \delta \leftarrow f(s_i) - f(s_{candidate})$
9: $\quad\quad$ **if** s_i is not in tabuList **and** $\delta < 0$ **then**
10: $\quad\quad\quad s_{candidate} \leftarrow s_i$
11: $\quad\quad$ **end if**
12: $\quad$ **end for**
13: $\quad s_{best} \leftarrow s_{candidate}$
14: $\quad$ Add $s_{candidate}$ to tabuList
15: **until** a termination criterion is satisfied
16: **return** s_{best}

Tabu Search is a memory-based metaheuristic

- Uses a tabu list to keep track of recently visited solutions to prevent cycling and encouraging exploration
  - tabu may have a tenure (size) that dictates how long a solution remains in the list
- We say a solution is tabu if it is in the list and does not fulfil some aspiration criterion
  - Aspiration criterion is a criteria that, if fulfilled, exempts a solution from being tabu → helps the algo avoid missing good solutions just because they are tabu
  - Example of aspiration criterion would be if the solution is better than the best known solution so far
- Keeps track of the best known solution

```
tabu_search(N, tenure) → Soln {
    let tabu_list = []
    let S_curr = gen_init_soln()
    tabu_list.push(S_curr)

    let S_best = S_curr

    // while stopping criterion is not met
    while !stopping_criterion() {
        let best_neighbour = None
        // generate and iterate over neighbours
        for nbour in gen_neighbours(S_curr) {
            // if neighbour is tabu and does not meet the
```

```
                // aspiration criteria, skip it
                if tabu_list.contains(nbour) && !asp_crit(nbour) {
                    continue
                // if neighbour is better than the best known neighbour
                } else if f(neighbour) < f(best_neighbour) {
                    best_neighbour = neighbour
                }
            }
            // move to the best non-tabu neighbour, even if it is worse
            // than the current soln
            S_curr = best_neighbour
            tabu_list.push(S_curr)

            if f(S_curr) < f(S_best) {
                S_best = S_curr
            }
            if len(tabu_list) >= tenure { // optional
                tabu_list.pop()
            }
        }

        return S_best
}
```

1. Generate an initial solution
2. Add the current solution to the tabu list, and update the best known solution if better
3. Explore the current solution's neighbourhood, evaluating each neighbour
4. Move to the best non-tabu neighbour (even if it is worse than the current solution)
5. Repeat 2-3 until a stopping criterion is met
6. Return the best solution

▼ Population-Based Methods

Population-Based methods are optimisation algorithms that work with a population of solutions rather than a single one

- Population is typically initialised randomly or by using heuristic techniques
- Solutions in the population are evaluated simultaneously, allowing the solution space to be explored more efficiently
    - the diversity of its population of solutions allows it to explore the space more broadly → helps avoid getting stuck in local optima and promotes a more comprehensive search
    - May require careful parameter tuning

    - Can be computationally demanding due to the population size and iterative nature
- Genetic algorithms fall under this category of methods

Genetic Algorithm Terminology

- Solutions are chromosomes/individuals → commonly represented as strings of bits or binary
- Parts of solutions are genes → commonly represented as bits
- Locus refers to the position of the gene, allele is the value
- Phenotype is the decoded solution (external appearance of the allele)
- Genotype is the encoded solution (internal structure)

Transformation in Genetic Algos involve crossover, mutation and selection

- Crossover involves taking two chromosomes and combining characteristics to generate 2 offspring
- Crossover rate p_c is the probability that the crossover operation is executed on a pair of chromosomes → critical to algorithm performance

$$crossover\_rate = \frac{n_{offspring}}{population\_size}$$

- Frequency of the crossover operation is determined by the crossover rate, which
- Mutation involves randomly changing a gene in a chromosome → contributes to exploration
- Mutation rate p_m is the probability that the mutation operation executes on a chromosome → controls the rate at which new genes are introduced to the population
- Selection involves choosing chromosomes from the current population and offspring to continue onto the next generation
- Selective pressure is the degree to which fitter chromosomes are favoured
    - High pressure → fitter chromosomes are favoured more → faster convergence since fittest chromosomes are propagated quickly → exploitation → can lead to premature convergence at a local optima
    - Low pressure → more randomness in selection, and less fit chromosomes have a higher chance of being selected → maintains diversity → exploration → but can be slow
    - Typically start low and gradually increase pressure

- **NOTE**: The below are selection strategies
- Uniform Selection → size stays the same for the entire algorithm → select exactly enough chromosomes from current and offspring to match the size
- Expanded Selection→ size of each generation is the size of the previous generation plus the number of offsprings produced
- Stochastic Sampling → probabilistic selection method introducing randomness to prevent certain chromosomes (especially very fit 'super chromosomes') from dominating the population rapidly → helps maintain diversity and avoids premature convergence
- Deterministic Sampling → deterministic selection method where we sort chromosomes by fitness and choose the best ones

Genetic Algorithms are a population-based, stochastic optimisation technique

- Idea is to have stochastic search technique that mimics natural selection to evolve better solutions to complex optimisation problems over time
- Models the process of evolution for a single chromosome (solution) as a sequence of changes in genes
- Used in problems of nonlinear and high-dimensional optimisation

NOTE: Steps may vary based on selection strategies used

1. Generate an initial population of random chromosomes
2. Evaluate the fitness of each chromosome in the current generation using the fitness function
3. Select chromosomes from the current generation based on their fitness (fitter individuals have a higher chance of being selected for reproduction)
4. For each pair of selected individuals, apply the crossover operator to generate new offspring per the crossover rate
5. Apply the mutation operator to the offspring, introducing small random changes to promote diversity per the mutation rate
6. Form the next generation (population) by using the chosen sampling methods (uniform vs expanded selection, stochastic vs deterministic sampling)
7. Repeat 2-6 until the stopping criterion is met
8. Return the best solution found during the search

▼ Assignment 1 Notes

`keras.models.Sequential(layers=None, trainable=True, name=None)`

- Creates and returns a `Sequential` model object → the NN model

- Usually we just use it without arguments → `keras.models.Sequential()`
- Can also pass in a `List` of layers

`Dense(units, activation=None)`

- Creates and returns a `Dense` layer object
  - `units` is the number of nodes in the layer
  - `activation` is the activation function to be used → linear by default → `g(x) = x`
  - `input_dim=` is a kwarg specifying the dimension of input
- nodes in a `Dense` layer object are fully connected to all nodes in the next layer (if exists)
- For the first, input layer of the model, we can specify `input_shape=(x,)` to say that inputs are 1D (e.g. List) and have `x` features, or `input_shape=(x, y, z)` to say that inputs are 2D having `x`, `y` and `z` features
  - For a 64×64 pixel colour image, we would use `input_shape(64, 64, 3)` → 64 pixels tall, 64 pixels wide, 3 (RGB) colour values
- e.g. `Dense(5, activation='tanh', input_dim=2)` creates a `Dense` layer object of 5 nodes that use tanh as an activation function, and takes inputs that have dimension 2 (`[x, w]`)

`Dropout(rate)`

- Creates an returns a `Dropout` layer object that randomly sets inputs to 0 with a frequency of `rate` at each step during training
- Inputs that aren't set to 0 are scaled up by `1 / (1 - rate)` so that the sum over all inputs is unchanged
- Helps prevent overfitting

`my_model.add(Dense(...))`, `my_model.pop()`

- `add` adds a `Dense` layer object to the stack of layers in `my_model`
  - Can actually be any type of `Layer`, not just `Dense`
- `pop` pops the layer at the top of the stack

`my_model.compile(optimizer="rmsprop", loss=None)`

- configures `my_model` for training, telling it to use the specified optimizer and loss functions
  - optimizer is typically `Adam` or `SGD` from `keras.optimizers`
  - we usually use `loss='mean_squared_error'`

- `SGD` and `Adam` require a learning rate to be specified → `SGD(learning_rate=0.01)`
- We can also specify a `List` or `Dictionary` of metrics to be evaluated during training/testing
  - metrics in the `List` / `Dictionary` can be the string name of a function in `keras.metrics` library, or an instance of the function
  - `metrics=['accuracy', 'mean_squared_error', 'precision', keras.metrics.BinaryAccuracy()]`
  - `my_model.metrics_names` will give the display labels for the metrics

`my_model.predict(x)`

- Generates a `NumPy` array of predictions for the input samples in `x`
- `x` can be a `NumPy` array or list of arrays, `tensor` or list of `tensors`, a `tf.data.Dataset`, or a `keras.utils.PyDataset` instance
  - We use a list of arrays or `tensors` if we have multiple inputs to run through the model

`my_model.evaluate(x, y)`

- `x` is the input data, `y` is the target data, and both have the same format as `x` for `predict(x)`
- Evaluates the performance of `my_model` over inputs `x` against the target outputs `y`
- Returns the loss value and any metrics of the model compiled
  - If multiple loss values, returns a list of them

`my_model.fit(x, y, epochs=1, validation_split=0.0)`

- Trains `my_model` , and returns a `History` object whose `.history` attribute is a record of loss and metric values from training
- `x` and `y` are the same above, and `epochs` is the number of times (epochs) we train the model → each epoch iterates over all of `x` and `y` once
- `validation_split` specifies what percent of the training data should be used as validation data instead → if `validation_data` is provided, `validation_split` will be ignored

$$w_i = w_i + \alpha(target - predicted) * input_i$$
$$b = b + \alpha(target - predicted)$$

Week 7 — Computer Vision

▼ Intro To Computer Vision

Computer Vision → field of AI focusing on how machines interpret and understand visual information from the world

- Rough goal is to automate tasks that need visual cognition like object (incl. character) detection and recognition, image classification, augmented reality
- Challenges to computer vision include variable illumination, complex and hard-to-describe objects, overlapping objects, shadows, etc.
- Currently we have made decent progress in specific computer vision problems like face recognition, but not much in general

Images are represented as a time-varying 2D matrix where each pixel (element) is an intensity value $I(x, y, t)$

- x and y are the horizontal and vertical spatial dimensions of the 2D image
- Colour vision involves three matrices (one for each RGB intensity value)
- Monochrome (grayscale) vision uses a single matrix
- Dynamic time-varying scenes (e.g. video) have a 3D matrix that varies with time t also → each 2D x by y layer is a frame in the video
- Iconic Model or Features can be obtained from the Image matrix, and the information that needs to be extracted depends on the task

Current Computer Vision essentially involves perspective projection, which is the process of mapping a 3D scene onto a 2D image plane

- Many-to-One transformation → points in the 3D world that align along the same viewing direction are projected onto the same point on the 2D image sensor → some information is loss → can't get exact depth or spatial arrangement from a single image alone
- Different scenes may produce identical 2D images
- Single images cannot be directly inverted to reconstruct the scene
- Ambient light levels can create noise → when we have bad lighting, we get random variations in pixel values, degrading the image quality (grainy or blurry) → affects performance of computer vision algorithms

When analysing objects in computer vision, certain features are critical to the extraction of meaningful information

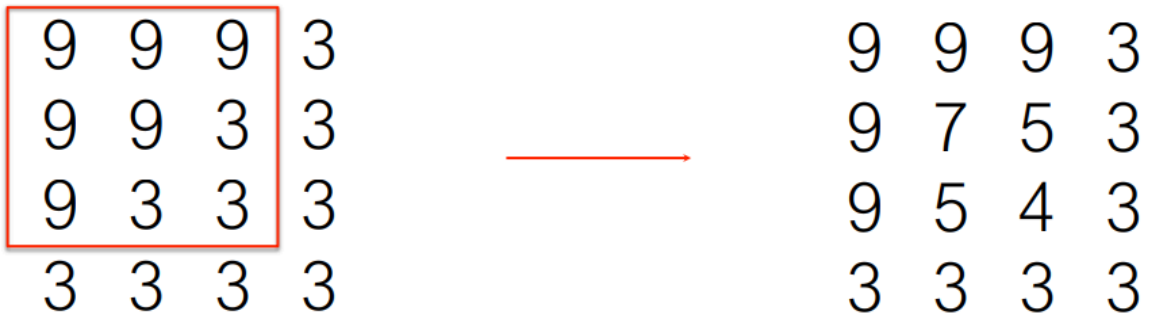
- Illumination (incident light) and reflectance (reflected light)
- Depth (distance from camera) and orientation (angle of the normal to the surface)
- Shading, colour, texture, etc.

Binary Vision involves converting an image into a binary image to simplify it, making it 'easier' for algorithms to process (detect objects or regions of interest)

- Binary Images are those that contain of only two pixel values: 1 (white) and 0 (black)
- Thresholding Process transforms the og image to a bin binary one by using a threshold brightness/intensity value

$$bin[x,y] = \begin{cases} 1 & \text{if } og[x,y] > threshold, \\ 0 & \text{otherwise} \end{cases}$$

▼ Image Processing



Averaging example with a 3×3 window using cropping to handle edge cases

- We use cropping so all pixels on the edges are skipped, and we only smooth the centre 4
- Supposing we start at the top left [0][0], we skip until [1][1] and apply smoothing, then we slide to [1][2] and smooth it, skip [1][3] and [2][0], and so on and so forth

Averaging is an image processing technique to improve the quality of an image

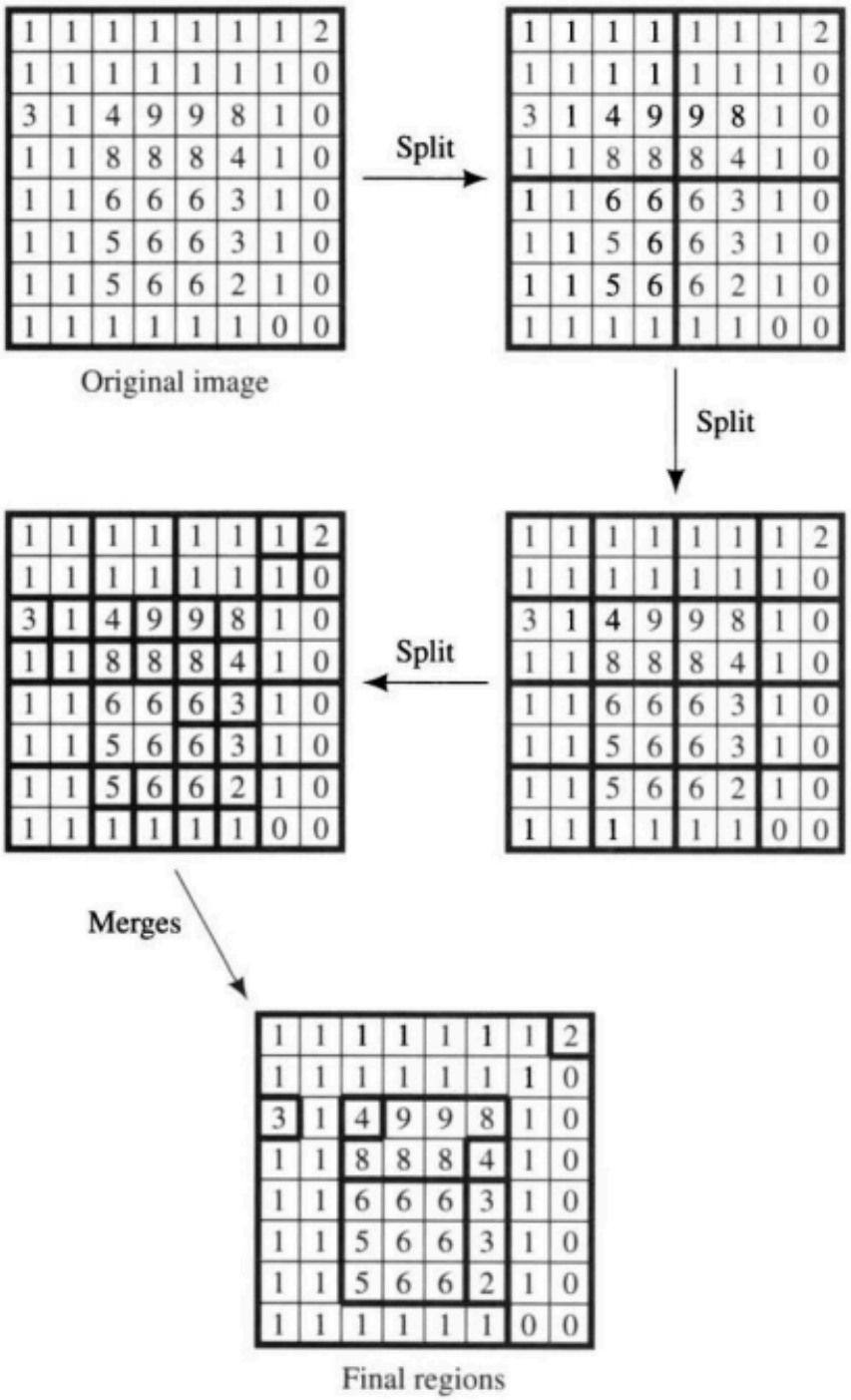
- Noise refers to random variations in pixel values, and may be caused by low lighting, sensor imperfections, transmission errors, etc.
- Averaging involves sliding a neighbourhood (window/kernel) over an image, pixel-by-pixel, and smoothing each pixel at the centre of the window
  - Smoothing involves replacing the value of the centre pixel with the average of pixels in the window
  - Reduces noise in an image → isolated dark or bright bits (thin lines, specks) are removed or 'blended' into the image

    - However, the image will get blurred, and sharp details and edges may be lost as a side effect
    - Window/kernel size impacts the effect of smoothing (and hence averaging) → larger windows blur more, but overly small windows won't do much to improve quality
- For edge pixels, we may pad the missing elements in the window with default values, or crop (skip smoothing) these pixels
- Due to the side effect, we should only use averaging if reducing noise is more important than preserving details

Edge Enhancement is an image processing technique to make edges more visible

1. Apply filters to detect edges → calculate difference between centre pixel and other pixels in the kernel to detect rapid changes in intensity (severe intensity transitions)
2. Pixel values along detected edges are increased/emphasised to make them stand out more clearly (highlight them)

- We extract edges to build a line drawing that we then analyse to identify objects → line drawing captures only the essential structure of objects and boundaries
- It is common to apply averaging and edge enhancement together



Split-and-Merge region finding method with $\varepsilon = 1$

Region Finding is an image processing technique used to group pixels into meaningful regions based on specific criteria

- Goal is to identify connected regions within an image that represent objects or areas of interest to facilitate further analysis of an image
- A region is said to be homogenous if there does not exist an intensity difference greater than some ε threshold within the region

- Split and merge method:
    1. Split each non-homogenous region in four
    2. Recurse on each of the 4 regions until all regions are homogenous (single pixel regions are homogenous)
    3. Iterate over all identified regions, merging adjacent ones if they are homogenous

▼ Scene Analysis

Scene Analysis is the process of extracting meaningful information from an image or video of a scene

- Scene refers to a visual environment containing multiple objects, regions, and/or activities
- Goal is to understand relationships between elements in the scene (captured through images) to make sense of the whole context
- Again, since scene to image is a many-to-one transformation, we lose spatial and depth information → often need multiple images to extract deeper meaningful information
- Extracted information is broadly categorised as specific and general
    - General Information includes things like camera location and orientation, illumination sources, and indoor/outdoor classification
    - Specific Information includes things like objects detected and their classes, an estimate of object depth, and recognised activities in dynamic scenes

Iconic Models are models that represent a scene (or parts of it)

- Iconic models create a structured model of key elements → helps form high-level understandings of a scene
- Scene is represented using maps or structural diagrams where objects are simplified into basic forms whilst still capturing spatial arrangements and relationships
- Highlights how objects are arranged relative to one another
- Divides model into regions of interest that each have some meaning within the scene → e.g. sky, building, road
- Representation may include shapes or outlines of objects, spatial relations between objects and maps of regions

Feature-Based Analysis involves extracting specific features from images for particular tasks

- Features may include points, edges, corners, textures, shapes, etc.

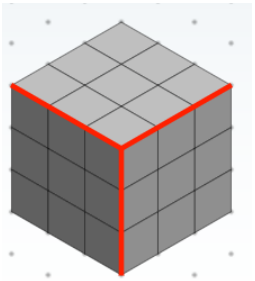
- Extracted features are often matched with models exhibiting those features to recognise objects or classify scenes

We need to be able to interpret lines and curves in images to understand the structure of scenes, and infer properties, relationships and intersections from the structure

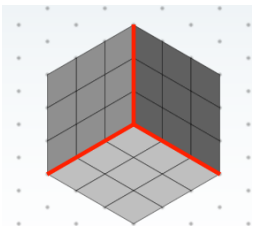
- We fit segments of rectilinear lines to straight edges, and curved ones to conic sections (ellipses, parabolas, hyperbolas) → allows accurate recognition of objects
- Lines should be postulated for rectilinear objects
  - Rectilinear objects are made of straight edges that can be reasonably assumed to exist to some extent
  - The 'missing' edges should be included, and line fitting algorithms will validate and refine them
- This fitting and postulating of lines produces line drawings → we associate properties with components of line drawings to infer the structure and relationships of surfaces

Types of Intersections

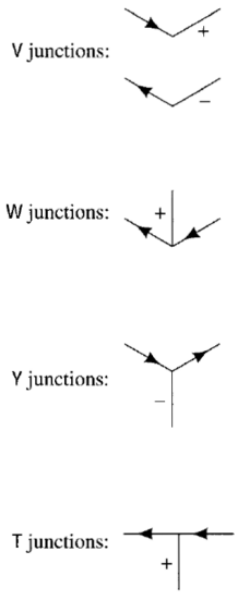
- Occlusion (→) → intersection of two planes, but one plane occludes (blocks) the other from view in the image
- Blade (+) → Intersection of planes at a convex edge where all are visible
- Fold (-) → intersection of planes at a concave edge where all are visible
- NOTE:
  - Convex edges → outer edges → they 'pop out'



  - Concave edges → inner edges → they 'cave in'

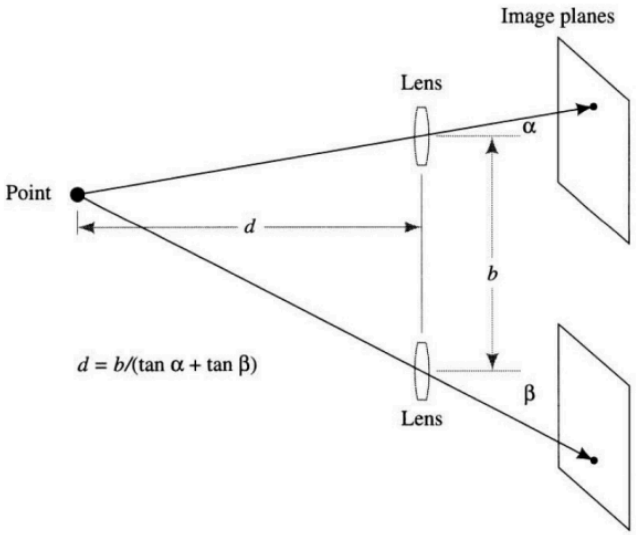


Types of Junctions



Model-Based Vision involves using pre-defined models of objects or scenes to recognise and interpret visual data

- Instead of analysing the image from scratch, we compare the input image with stored models to identify objects and analyse the scene
    - Extract features, match the scene to a model, interpret the scene
- Very useful when dealing with structured environments or known objects
- Models are predefined geometric or structural representations of an object, scene or shape → can be 3D CAD models, mathematical descriptions, templates or patterns, etc.



Stereo Vision perceives depth by analysing differences between two or more images taken from slightly different viewpoints → kinda like triangulation

- Allows 3D structure of a scene to be estimated alongside distance of objects from camera
- The two images work similarly to how we use two eyes to perceive depth
- We can do this with just a single image if we know the object and the camera's position in the scene

▼ Cognitive Vision

Aim is to simulate human-like understanding of visual scenes by incorporating reasoning, learning and memory

- Perception is not just extracting information → it is interpreting the information while being aware of the context
- Computer Vision systems get 3D descriptions of the scene and assign labels to objects and/or actions that are provided to symbolic reasoning systems
  - Difficult to get labels (meaningful symbols) from visual data since there is no direct causal link between the moment (present) and context/history of the scene
  - Also difficult to anticipate future events or infer behaviour over time for similar reasons

Cognitive vision involves extracting features from visual data for real-time control

- Reasons about the detected object entities, the actions they are performing, how the actions evolve into meaningful events, alternative possibilities for what may happen

next

    - Tries to answer why, how and who on top of what and where
- Continuous feedback loop between prediction and exploration → predict, act, observe, adapt loop
    - Predicts what it expects to see and happen based on its understanding of the scene, and internal models and goals
    - Acts (based on reasoning/planning) to explore the environment, verifying predictions through perception and adjusting behaviour if necessary
- Makes cognitive vision more adaptive and capable of goal-directed behaviour
- Language serves as an essential mechanism for guiding V's attention and lessening what R must process
    - Synonymy (understanding synonyms) helps systems recognise multiple expressions referring to the same thing → greater flexibility in following instructions
    - Hypernymy (recognising is-a relationships) helps reason about objects even if the system only detects a general class of an. object
- Cognitive Vision utilises the contextual co-occurrence of objects to improve recognition and reasoning → narrows search space and guides attention towards expected objects

Interactions between Vision (V) and Reasoning (R) systems

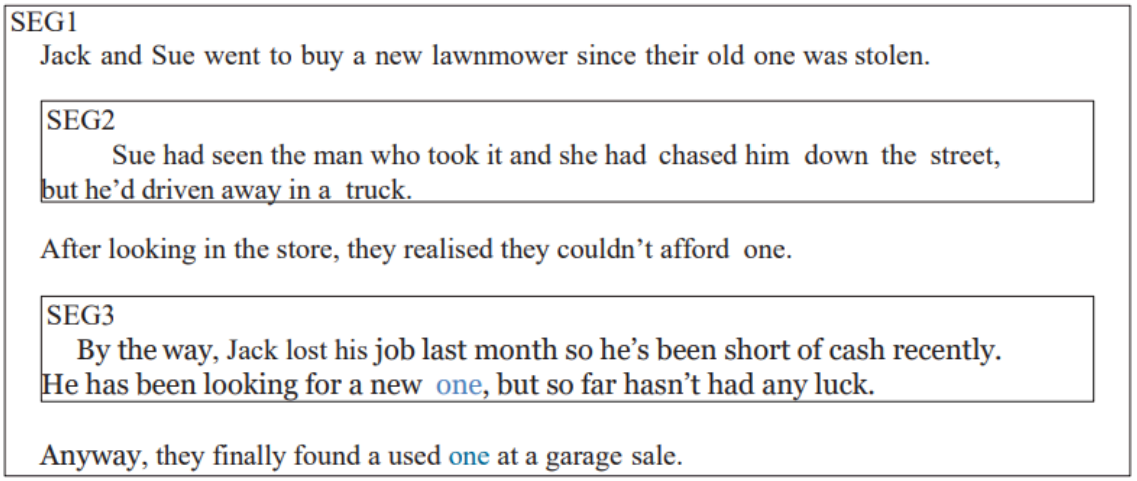
- Five interaction paths for V and R
    - $V \rightarrow R$ → visual data from V informs R → traditional perspective for computer vision
    - $R \rightarrow V$ → a reasoning task R prompts the system to look for something specific
    - $V \rightarrow R \rightarrow V$ → V informs R, and if R finds the result implausible or unclear, it asks V for more information → feedback loop
    - $R \rightarrow V \rightarrow R$ → R determines a requirement prompting V to collect specific data that is fed back to R for further analysis
    - $R \rightarrow VV...V$ → R imagines or envisions multiple possible outcomes or scenarios (Vs) → creates mental models or simulations to anticipate actions or events
- Interactions can happen at any stage
- Note that cognitive vision does not operate in isolation, but integrates multiple sensory modalities for holistic understandings of the environment and evaluations of actions

Week 8 — Language Processing

▼ Intro To Language Processing

Ambiguity makes it difficult to interpret the meaning of statements or expressions in natural language, and is thus the central challenge in NLP

- Much ambiguity arises from references like “one”, “it”, “others”
- We break down text into a hierarchical structure of ‘segments’ for processing
  - Different segments can play different roles that are factored into the NLP application’s understanding
  - Text may be broken into segments if there is a topic shift/change
  - Pronouns or references are tracked to which previous segment they refer to, so as to resolve ambiguity somehow???



- SEG1 is the main narrative
- SEG2 provides background information about why the main narrative is happening
- SEG3 provides contextual information to support reasoning throughout the main narrative

▼ Formal Languages and Grammar

Grammar rules are formal devices for defining sets of sequences of symbols

- Sequences may represent a statement in a programming language, or may be sentences in natural languages
- Formally, a grammar is a 4-tuple $G = (V, \Sigma, R, S)$

    - V is a finite set of non-terminal symbols → variables that define intermediate structures → expand into other non-terminal or terminal symbols according to R
    - Σ is a finite set of terminal symbols → basic units of the language → actual words of characters making up the final output of a sentence
    - R is the set of production rules → relations defined in $V \times (V \cup \Sigma)$ → each rule has the form $A \to \alpha$, where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$
    - S is the start symbol → the initial non-terminal from which the derivation process begins → goal is to derive a sequence containing only terminal symbols from S
- Production rules are written `<S> ::= a b` or `<S> ::= a<S>b`, taking a terminal symbol `S` and producing another sequence of terminal and non-terminal symbols
    - First rule says that non-terminal `S` can be rewritten as `a b`
    - Second rule says that non-terminal `S` can be rewritten as `a S b`

Note:

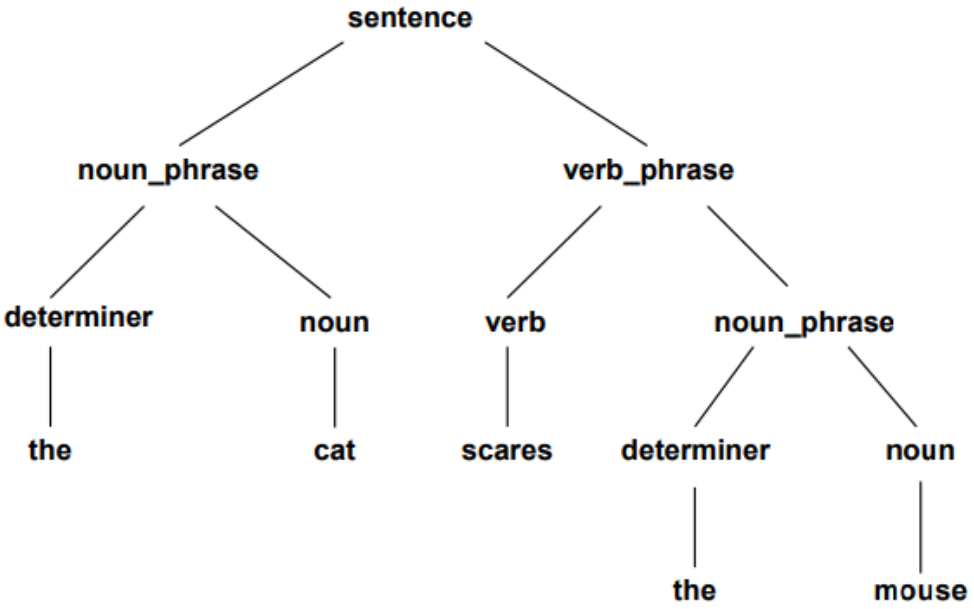
- $\alpha \in (V \cup \Sigma)^*$ means that α is a valid sequence of non-terminal and terminal symbols, including the empty sequence
- $(a)^*$ means 0 or more a
- $[a]$ means that a is optional

▼ Example

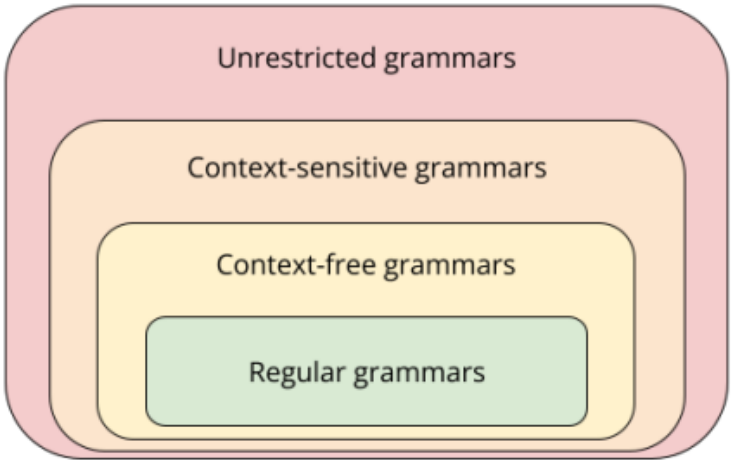
Suppose we have the following $V = \{sentence, determiner, noun, verb\}$, $\Sigma = \{[a], [the], [cat], [mouse], [scares], [hates]\}$, and $R =$

```
sentence --> noun_phrase, verb_phrase
noun_phrase --> determiner, noun
verb_phrase --> verb, noun_phrase

determiner --> [a]
determiner --> [the]

noun --> [cat]
noun --> [mouse]

verb --> [scares]
verb --> [hates]
```

For $S =$ "the cat scares the mouse", we can get the following parser tree



- This helps us understand how we may break a non-terminal state `S` into a sequence of terminal states
- Notes for parser trees:
  - Leaves are terminal symbols in the Grammar
  - Internal (not root or leaf) nodes are labelled by non-terminals
  - Parent-child relations are specified by the rules of the grammar → e.g. $verb \rightarrow [scares]$ so we have parent $verb$ and child $scares$



Chomsky's Hierarchy is a hierarchy of classes of grammatical formalisms

- Grammatical formalisms are classified by their generative capacity
- Unrestricted Grammars → both sides of the rule can have any number of terminal and non-terminal symbols → e.g. $A\ B\ C \rightarrow D\ E$ → can describe any computable language
- Context-Sensitive Grammars → RHS must contain at least as many symbols as the LHS → e.g. $A\ X\ B \rightarrow A\ Y\ B$ → in the context of a preceding A and following B, X can be rewritten as Y → can represent $a^n b^n c^n$
- Context-Free Grammars → LHS is a single non-terminal symbol → can handle $a^n b^n$ but not $a^n b^n c^n$ patterns since it uses one level of recursion that can track the number of as and bs, but not as and bs and cs
- Regular Grammars → LHS is a single non-terminal, RHS is a terminal optionally followed by a non-terminal symbol → cannot represent $a^n b^n$, most they can do is $a^* b^*$

▼ Regular Expressions

Regex is a formal language for specifying text strings

- `[]` matches any chars inside the brackets → `[nN]uke` matches `nuke` and `Nuke`
- `[A-Z], [a-z], [0-9]` matches any char within the range in the brack → can have `[a-zA-Z]` to match any alphabetic char → `[0-9] nukes` matches `2 nukes` , `3 nukes` , etc.
- `^` means negation if it is the first char in `[]` → e.g. `[^a-zA-Z]` means not alphabetic, but `[a-zA-Z^0-9]` matches a letter or '^' or a digit, and `a^b` matches "a^b" only
- `|` for disjunction → `don|key` matches `don` and `key`
- `?` optionally matches the previous char
- `*` matches 0 or more of the previous char
- `+` matches 1 or more of the previous char
- `.` matches any 1 char
- `\d` for digits, `\D` for no digits
- `\s` for whitespace `\S` for no whitespace
- `\w` for word (`[a-zA-Z0-9_]`), `\W` for non-word characters
- `{x}` means x of the preceding char → e.g. `a{4}` matches `aaaa` and `\d{4}` matches `1234` , `5678` , `8888` , etc.
- Outside of `[]` , `^` indicates the start and `$` indicates the end of a pattern to match
  - `.* ^abc$ .*` matches anything "abc" anything
  - `^[^a-zA-Z]` would match "123" of "123ABCabc"
  - `.$` would match "!" from "Holy moly!"

- `\` is the escape char → e.g. `\(brackets\)` matches "(brackets)"

Regex Errors can be type 1 or type 2

- Type 1 → false positives → matching strings that should not have been matched
- Type 2 → false negatives → not matching strings that should have been matched
- To reduce the error rate, we increase accuracy/precision (minimise false positives) and increase coverage/recall (minimise false negatives)

In Python and Unix we can do `s/regexp/pattern/` to replace matches of a `regexp` with `pattern`

- We call this substitution
- Applying `s/([0-9]+)/<\1>/` on "Hello 123 and 456" returns "Hello <123> and <456>"
  - `([0-9]+)` captures "123" and "456"
  - `\1` references the content captured by the first group `([0-9]+)`

Regex Capture Groups allow us to group parts of a pattern and extract or reuse matched content

- Groups are enclosed in parentheses, and the matched groups can be extracted, referenced later in the regex itself, or used for substitutions
- `\x` refers to the content captured by the xth capture group
- E.g. applying `my name is (.+)` on "my name is Felix" will capture "Felix"
- If we do not want a group to capture content, we put `?:` after the opening bracket `(` → e.g. `/(?:some|a few) (people|cats) like some \1/` will match "some cats like some cats", but not "some cats like some some" since `\1` references the first capture group, which is `(people|cats)` since `(?:some|a few)` is not a capture group

Word Counting in NLP can be tricky due to pauses and fragments

- Need to consider whether filled pauses like "uh", "um", etc. and fragments like "so-" in "so- sorry" count as words

Terms to understand in NLP

- A lemma is the base or dictionary form of a word → words that share the same base form and general meaning belong to the same lemma
- Wordforms are the actual surface form of a word in text
- "cat" and "cats" are different wordforms that belong to the same lemma

- Morphemes are the small meaningful units that make up words
  - stems are the core meaning-bearing units
  - Affixes are the parts that adhere to stems, often with grammatical functions
- A token refers to an individual occurrence of a word or symbol in a given text
  - Every word, punctuation mark or unit of text that is counted is a token
  - Different occurrences of a word are counted as separate tokens
- A type is the unique instance of a word or symbol in a given text
  - The types of a text represent its distinct vocabulary
- Corpora refers to a large collections (datasets) of texts or speech used for the training of NLP models

All NLP tasks require text normalisation

1. Tokenise (segment) words
2. Normalise word formats
3. Segment sentences

Tokenisation

- A simple way to tokenise is to split by whitespace
- The `tr` command is a Unix tool for space-based tokenisation → given a text file, it outputs the word tokens and their frequencies, ignoring all numbers and punctuations
- However, we can't just blindly remove punctuation (emails, urls, prices), and multi-word expressions should still be considered one word (e.g. "Hong Kong"
- A standard method is to use deterministic algorithms based on regexp to tokenise text

Word Normalisation involves putting words/tokens into a standard format

- E.g. either "U.S.A" or "USA", not both
- For applications like information retrieval or speech recognition, we can freely reduce all letters to lower-case and (maybe) only keep capitals of mid-sentence words to preserve proper nouns
- For sentiment analysis or machine translation, case is helpful and so should be preserved
- Lemmatisation involves representing all words as their lemma (shared base)

    - am, are, is and be are all replaced with be, and car, cars are replaced with car
    - “he is throwing stones at zombies” becomes “he be throw stone at zombie”
    - Lemmatisation is done by morphological parsing → “cats” becomes two morphemes (”cat” and “s”)
    - Stemming involves reducing terms to stems by crudely dropping affixes → Porter Stemming algo based on rewrite rules → e.g. “ATIONAL” to “ATE” (relational to relate) drop “ING” (doing to do), “SSES” to “SS” (grasses to grass)
    - The Porter Stemming algorithm is based on a series of rewrite rules run in series → e.g. replace “ATIONAL” with “ATE” (relational becomes relate), “ING” gets dropped (motoring to motor), “SSES” to “SS” (grasses to grass)

Sentence Segmentation can be very difficult due to the ambiguity of some punctuation

- Especially true with period ‘.’ → can be used for abbreviations like Dr. or Inc., decimal points, ellipses (...), etc. as well as to terminate a sentence
- Thus we commonly tokenise text then use a set of rules or ML to classify a period as either a part of a word or a sentence-boundary

▼ Minimum Edit Distance

The minimum edit distance between two strings X and Y is the minimum number of editing operations required to transform X into Y

- Operations are insertion, deletion and substitution
    - Insertion and Deletion cost 1
    - Substitution involves deleting then inserting, so it costs 2
- When we begin, we want to align X and Y such that the maximum possible number of character matches are there
- We call the sequence of edits required to transform X into Y the path from X to Y, and the cost of this path is the edit distance

Edit Path Finding involves starting from A and visiting adjacent states through each operation until we reach the goal state B

- Lots of distinct paths wind up at the same state → we can just keep track of the shortest path to each distinct state → Dynamic Programming
- Supposing `len(X) == n` and `len(Y) == m`, we let `D(i,j) = the distance between X[1..i] and Y[1..j]` (the edit distance of the first i and j chars of X and Y)
- Order of Computation → increasing i and increasing j
- Base Cases → `D(i, 0) = i` and `D(0,j) = j`
- Recursion →

$$D(i,j) = \min \begin{cases} D(i-1,j)+1 & (\text{deletion}) \\ D(i,j-1)+1 & (\text{insertion}) \\ D(i-1,j-1) + \begin{cases} 2 & \text{if } X(i) \neq Y(j) \\ 0 & \text{if } X(i) = Y(j) \end{cases} \end{cases}$$

- Final Result → `D(n, m)`

▼ Pseudocode

```
let X = some string;
let Y = other string;

let n = len(X);
let m = len(Y);

// nxm memo initialised to INT_MAX
let mut D = [[INT_MAX; m]; n];

for i in 0..n {
    for j in 0..m {
        let t1 = D{i - 1][j] + 1;
        let t2 = D[i][j - 1] + 1;
        let t3 = D[i - 1][j - 1] + if X[i] != Y[j] {
            2
        } else {
            0
        };
        D[i][j] = min(t1, min(t2, t3));
    }
}

println!("Edit distance between {X} and {Y} is {}", D[n-1][m-1]);
```

N	9	8	9	10	11	12	11	10	9	**8**
O	8	7	8	9	10	11	10	9	8	9
I	7	6	7	8	9	10	9	8	9	10
T	6	5	6	7	8	9	8	9	10	11
N	5	4	5	6	7	8	9	10	11	10
E	4	3	4	5	6	7	8	9	10	9
T	3	4	5	6	7	8	7	8	9	8
N	2	3	4	5	6	7	8	7	8	7
I	1	2	3	4	5	6	7	6	7	8
#	**0**	1	2	3	4	5	6	7	8	9
	#	E	X	E	C	U	T	I	O	N

▼ Natural Language Modelling → N-gram Models

Probabilistic Language Models assign probabilities to sequences of words to 'understand' how likely words are to appear together in some specific order

- Used for tasks like autocomplete, speech recognition, translation, spelling and grammar correction
- A language model (LM) is a model that computes either
  - The probability of the entire sequence $P(W) = P(w_1, w_2, w_3, w_4)$
  - The likely next word in a sequence $P(w_4|w_1, w_2, w_3)$
- While Grammars rely on rules for sentence structure, LMs learn patterns and probabilities from large datasets, making them more suited for real-world tasks

To compute $P(W)$ we use conditional probabilities

- Formula for probability of B given A is $P(B|A) = P(A, B)/P(A)$ → rearrange to get the probability of A then B → $P(A, B) = P(A) * P(B|A)$
- In general

$$P(w_1, w_2, ..., w_n) = P(w_1) * P(w_2|w_1) * P(w_3|w_2, w_1)... * P(w_n|w_1, ..., w_n - 1)$$

- It is impractical to count all sentence occurrences and calculate the probabilities of $P(B|A)$, so we use the Markov Assumption → assume the next words depends only on a fixed number k of previous words
  - $P(w_n|w_1, w_2, ..., w_{n-1}) \approx P(w_n|w_{n-(k-1)}, ..., w_{n-1})$
  - We then have that $P(w_1, w_2, ..., w_n) \approx \prod_{i=1}^{n} P(w_i|w_{i-k+1}, ..., w_{i-1})$

- Unigram calculates the probability of a sequence as the product of the probabilities of each individual word → $P(W) = \prod_{i=1}^{n} P(w_i)$
  - $P(w_i)$ is estimated using the frequency of w_i in the corpus → $\frac{count(w_i)}{n}$
  - Cannot take context into account
- Bigram assumes probability of a word only depends on the previous one (first-order Markov rule) → $P(W) = P(w_1) \times \prod_{i=2}^{n} P(w_i|w_{i-1})$
  - $P(w_i|w_{i-1})$ is estimated from word pairs in the corpus → $\frac{count(w_{i-1}w_i)}{count(w_{i-1})}$
  - Cannot capture longer dependencies
  - If a bigram does not appear in the training corpus, we may get 0 probabilities for unseen word pairs
- We can extend to N-gram models → trigrams, 4-grams, 5-grams
- Generally, N-grams are insufficient models of language since language has long-distance dependencies that will exceed efficient window sizes (k), but they are still useful in many cases

▼ Tute Notes → some stuff on some different classifiers

`LinearSVC(C=1.0, max_iter=1000)`

- linear vector classifier
- `C` is the regularisation parameter

`KNeighborsClassifier(n)`

- Samples `n` neighbours and picks the one with the lowest distance
- Used often with unlabelled data

`AdaBoostClassifier()`

- Adaptive boosting → starts with a simple model, gets the result, builds on the model, repeat

Week 9 — Uncertain Reasoning

▼ Intro

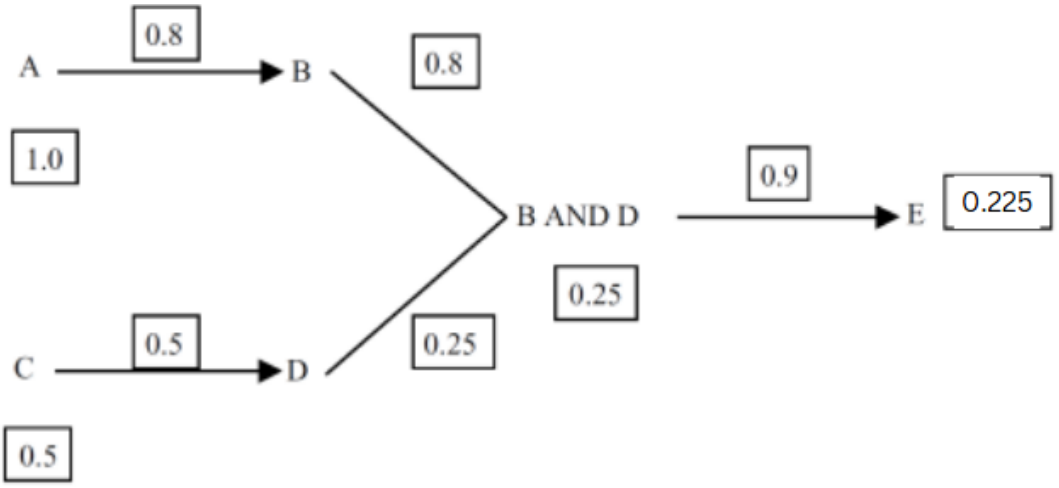
When reasoning with uncertainty, we often lack knowledge of precise data, and so base our reasoning off of default or common values instead

- The more knowledge we lack, the more guesses or defaults we use, and the lower the quality of our final reasoning becomes

- Uncertainty can be expressed both in the information (data) and reasoning process (model makes assumptions or uses approximations or probabilities)
- Systems designed to emulate human reasoning must generate and rank multiple potential solutions
    - Uncertainty might be expressed as confidence in which continuous values are allowed → probability that an output is correct
- AI systems that can handle uncertain reasoning can make robust decisions with imperfect or missing information
- NOTE: the conclusion of a rule in uncertain reasoning cannot be guaranteed → e.g. high blood pressure increases heart attack risk, but not all people with HBP have heart attacks
- Uncertain evidence → handle by using confidence factors

Confidence Factors are factors that quantify the certainty or belief in a particular piece of evidence → they represent the level of confidence we have in a piece of evidence

- When we may have uncertain evidence, we assign confidence levels to input data, so that the reasoning system can weigh how trustworthy the evidence is when making inferences
- Uncertainty can be in antecedents or rules → we assign confidence factors to both types
    - Uncertainty in antecedents (the 'if' part of 'if-then') mean we are uncertain about the conditions of a rule → can come from user-provided data, or from the deductions of other rules
    - Uncertainty in a rule refers to our confidence in a rule itself → we assign confidence factors to rules themselves
- If we have a rule "IF A THEN B", then we have that $CF(B) = CF(A) \times CF(\text{IF A THEN B})$
- NOTE: $CF(A \wedge B) = \min(CF(A), CF(B))$, and $CF(A||B) = CF(A) + CF(B) - (CF(A) \times CF(B))$
- Inference networks are sequences of relationships between facts and confidence factors



Bayesian Inference involves inferring information from data through probability when uncertain

- We use prior evidence with current information to calculate the probability of a hypothesis being true, and update this probability as more information becomes available

▼ Probabilistic Inference → Bayes' Rule

Reasoning is independent of the definitions of propositions

- Recall that propositions are entities (facts or non-facts) that can be true or false

Probability Theory serves as the formal language for representing and reasoning with uncertain knowledge

- Bayes Theorem allows us to compute conditional probabilities $P(A|B)$
- Widely used in ML → e.g. classification predictive modelling

Probabilistic Inference involves attaching a degree of belief to each primitive proposition/event

- If we have a proposition A, then we have $P(A)$ as the degree of belief in A being true
- Notation
  - $P(Weather = rainy) = 0.6$ means we believe the weather will be rainy with a 60% certainty
  - $P(Cavity = true) = 0.05$ means we believe the person has a cavity with a 5% certainty
- Axioms

    - $P(A) = 1.0$ means we are certain A is true, $P(A) = 0.0$ means we are certain A is not true (A is false) → $P(True) = 1.0$, $P(False) = 0.0$
    - $P(A \vee B) = P(A) + P(B) - P(A \wedge B)$
- Properties
    - $P(\tilde{A}) = 1 - P(A)$
    - $P(A) = P(A \wedge B) + P(A \wedge \tilde{B})$
    - $\sum P(A = a) = 1.0$ → probabilities of values a in sample space A is 1.0

Bird	Flier	Young	Probability
T	T	T	0.0
T	T	F	0.2
T	F	T	0.04
T	F	F	0.01
F	T	T	0.01
F	T	F	0.01
F	F	T	0.23
F	F	F	0.5

Joint Probability Distribution provides the probability of different combinations of outcomes for two or more variables → shows the joint probability of different combinations of events

- Joint Probability $P(A \wedge B) = P(A, B) = P(A \text{ and } B)$ is the probability of two events A and B occurring simultaneously
- Full Joint Probability Distribution shows the joint probability for all possible combinations of events
    - If we have n variables that can take k possible values, we have k^n rows in the table → i.e. if we have n Boolean variables, we have a table with 2^n rows
    - The sum of the probabilities is 1
    - If we have the full joint probability distribution, we can compute any probability in the domain → e.g. if we had the table for A, B, C and D, we can calculate P(A), P(B), ..., P(AB), P(AC), ..., P(ABC), P(ABD), ..., P(ABCD), etc. from it
    - Note that P(X) is the sum of the probabilities of all rows where X is true
    - The sum of the probabilities
- If a problem or statement requires us to compute the Full joint probability distribution, it is intractable as this can be very large

Conditional Probability is the probability of one event given the occurrence of another

$$P(A|B) = \frac{P(A \wedge B}{P(B)}$$

- Conditional probabilities accumulate evidence → critical for reasoning
- $P(A|B) = 1$ is equivalent to "IF B THEN A" → $P(A|B) = 0.9$ is "IF B THEN A" with 90% certainty
- Chain: $P(A, B, C, D) = P(A|B, C, D) \times P(B|C, D) \times P(C|D) \times P(D)$
- Product: $P(A, B) = P(A) \times P(B|A) = P(B) \times P(A|B)$
- Chain rule is just product rule, but all terms except a base probability is expanded

Bayes' Theorem

$$P(A|B) = \frac{P(B|A) \times P(A)}{P(B)}$$
$$= \frac{P(B|A) \times P(A)}{P(B|A) \times P(A) + P(B|\neg A) \times P(\neg A)}$$

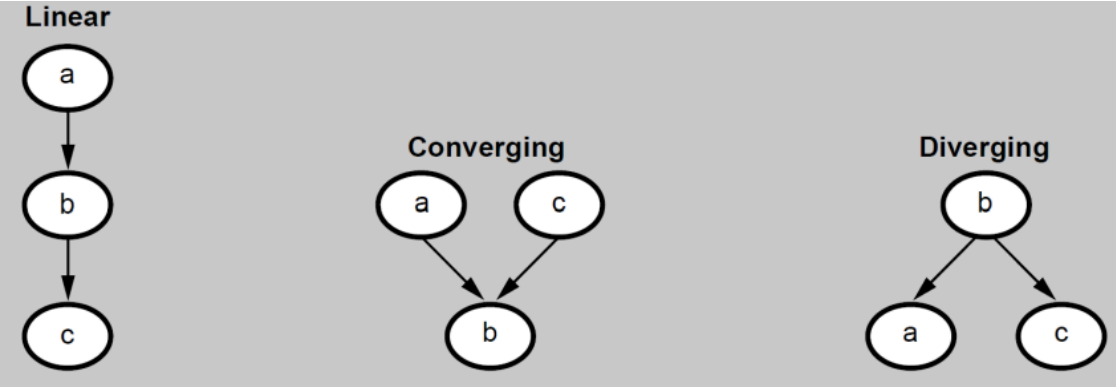
- $P(A|B)$ is posterior probability, $P(A)$ is prior probability, $P(B|A)$ is likelihood, and $P(B)$ is the evidence
- We often have situations where we have $P(B|A)$, but want $P(A|B)$ → Bayes' Theorem allows us to get $P(A|B)$ from $P(B|A)$

▼ Belief Networks → Bayesian Networks

Bayesian or belief Networks are space-efficient data structures (DAGs) for full joint probability distributions

- They represent causal relations → arcs from cause to immediate effects → include nodes for each variable, and an edge from A to B if A is a direct causal influence on B
  - Variables are bools
  - Certain conditional independence assumptions hold → these reduce the number of probabilities we need to compute for full joint probability distribution → if variables are independent, combinations with them don't matter so we can cut them out
  - Events A and B are only conditionally dependent if they are immediate neighbours (i.e. one is a parent/child of the other)
- Causes at the top, effects at the bottom → If we want to do predictive reasoning, we go top down, otherwise we go bottom-down if we want to do diagnostic reasoning

- For full joint probability distribution representation, we make prior probabilities root nodes, and conditional probabilities non-root nodes that consider all combinations from the direct predecessors
- Three types of topologies (connection types between nodes)



  - Linear → (b) will calculate $P(b|a)$ and $P(b|\neg a)$, (c) will calculate $P(c|b)$ and $P(c|\neg b)$
  - Converging → (b) will calculate $P(b|a,c)$, $P(b|a,\neg c)$, $P(b|\neg a,c)$, $P(\neg a,\neg b)$
  - Diverging → (a) will calculate $P(a|b)$ and $P(a|\neg b)$, (c) will calculate $P(c|b)$ and $P(c|\neg b)$

▼ Fuzzy Logic

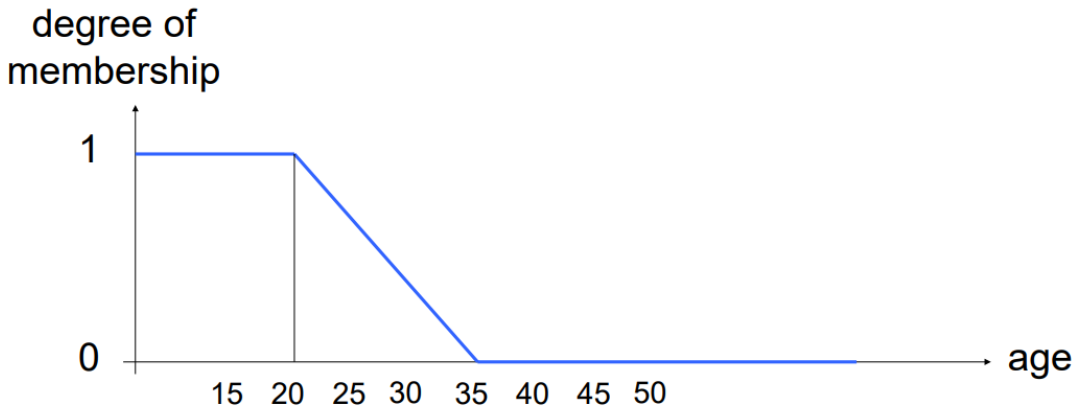
In fuzzy logic we use degrees of truth rather than the binary true or false logic

- Variables can belong to more than one category to varying degrees → e.g. temperature can belong to multiple categories at varying degrees → 30 degrees celsius is hot to a degree of 0.8, and warm to a degree of 0.4
- Each variable has a membership function that determines how true a statement is

Fuzziness differs from probabilistic uncertainty in that:

- Ambiguity is related to the degree to which events occur, regardless of the probability of their occurrence

- Fuzziness does not dissipate with an increase in the number of occurrences → it is about the subjectivity in the rating of events, not the probability that an event happens

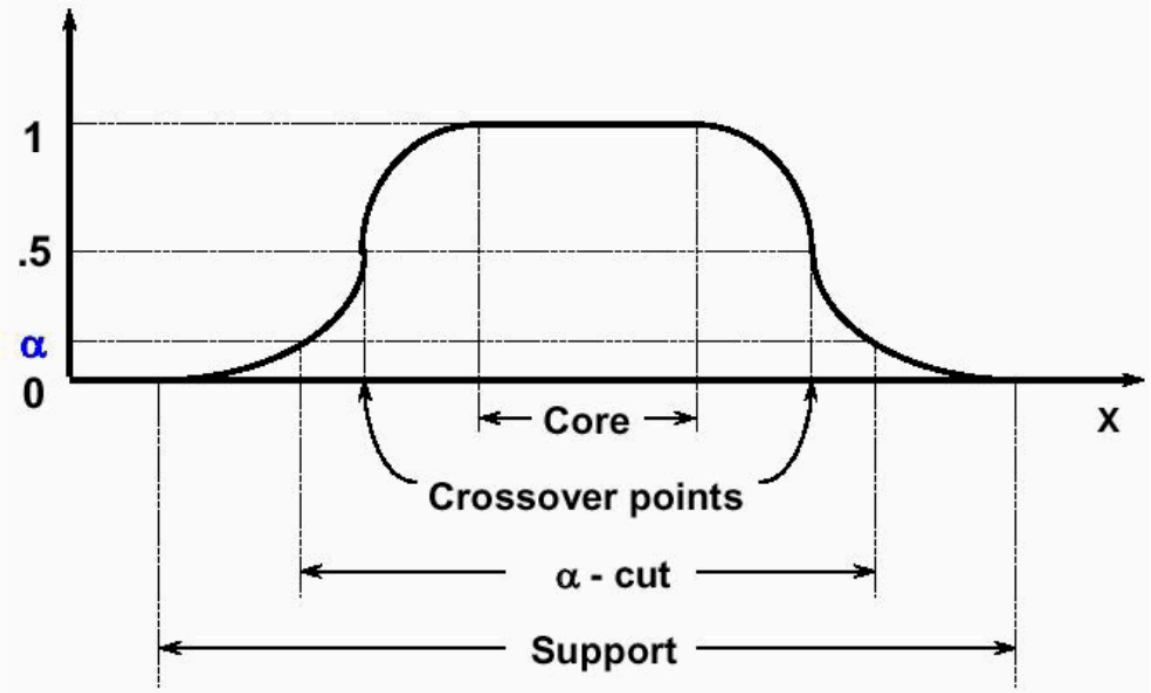


A = {1/10, 1/15, 1/20, 0.75/25, 0.25/30, 0/35}

A = {(1,10), (1,15), (1,20), (0.75,25), (0.25,30), (0,35)}

Fuzzy Sets relax the degree of membership restriction to within the [0,1] interval instead of having to be 0 or having to be 1 → can have partial members

- A fuzzy set in the universe U is characterised by membership function $A(x)$ that accepts values $x \in [0, 1]$
- The degree of membership of a variable x to a fuzzy set S is written $\mu_S(x)$
- Two ways of representing a fuzzy set S
  - $S = \{(\mu_S(x), x) | x \in U\}$ → the fuzzy set has tuples "(degree, var)"
  - $S = \{(\mu_S(x)/x) | x \in U\}$ → the fuzzy set has elements "(degree / var)" → note that / does NOT mean division
- Note that the plot of $A(x)$ can be a bump like _/_



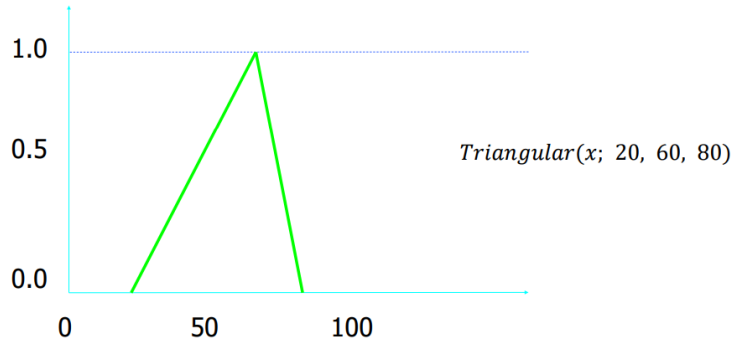
Parts of a fuzzy set

- The support of a fuzzy set is the set of all elements that have non-zero membership values → $Support(S) = \{x \in U | \mu_S(x) > 0\}$
  - If $|Support(S)| == 0$ it is an empty fuzzy set
  - If $|Support(S)| == 1$, it is a fuzzy singleton
- Crossover points are where the membership value is 0.5
- The core of a fuzzy set is the set of all elements that have membership values of 1 → $Core(S) = \{x \in U | \mu_S(x) == 1\}$
- The height of a fuzzy set is the highest membership value achieved by any point
- The centre value of a fuzzy set is a representative value indicating the centre of the set's mass → if the mean of all points with the maximum membership value is finite, the centre is the average of those values, otherwise it is the smallest value among those points
- The α-cut is the set containing all elements that have membership values greater than or equal to a select α
  - $S_\alpha = \{x \in U | \mu_S(x) \geq \alpha\}$
  - $S_{\alpha+} = \{x \in U | \mu_S(x) > \alpha\}$ for a strong α-cut
- If we have a fuzzy set S and two values $\alpha_1, \alpha_2 \in [0, 1]$ such that $\alpha_1 > \alpha_2$, then
  - $S_{\alpha_1} \subset S_{\alpha_2}$
  - $(S_{\alpha_1} \cap S_{\alpha_2}) = S_{\alpha_1}$

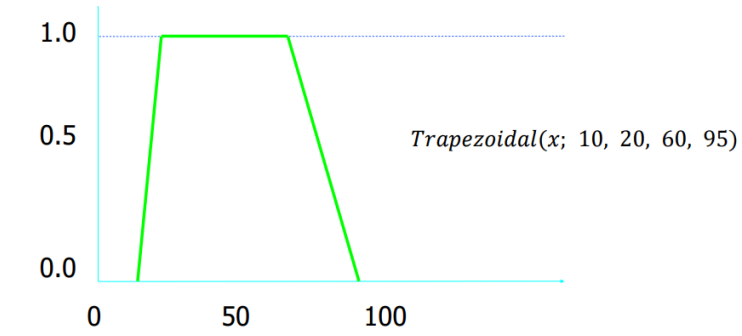
- $(S_{\alpha_1} \cup S_{\alpha_2}) = S_{\alpha_2}$
- And the same for the strong α-cuts

Membership functions are defined through fuzzy set notation, and can hence be discrete or continuous

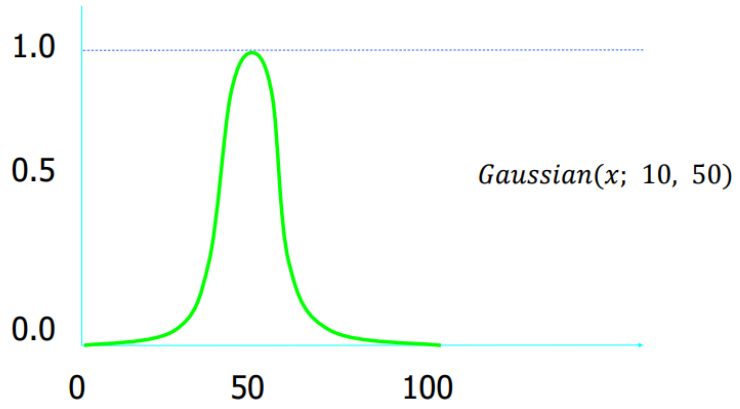
- We denote the membership function of x to set S as $f_S(x)$
  - $f_S(x) \in [0, 1]$
  - $f_S(x) == 1$ means x completely belongs to S, and $f_S(x) == 0$ means x does not belong in S
  - $f_{A \text{ or } B}(x) = \max(f_A(x), f_B(x))$
  - $f_{A \text{ and } B}(x) = \min(f_A(x), f_B(x))$
  - $f_{\text{nor } A}(x) = 1 - f_A(x)$
- Membership functions can be Triangular, Trapezoidal, Gaussian, Bell-Shaped, Sigmoid



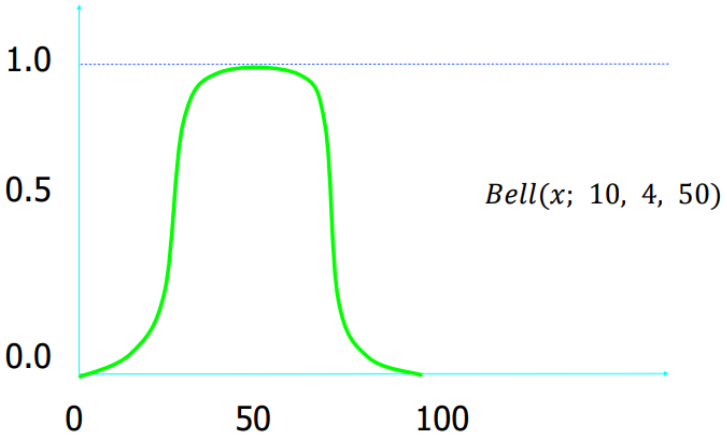
$$Triangular(x; a, b, c) = max\left(min\left(\frac{x-a}{b-a}, \frac{c-x}{c-b}\right), 0\right)$$



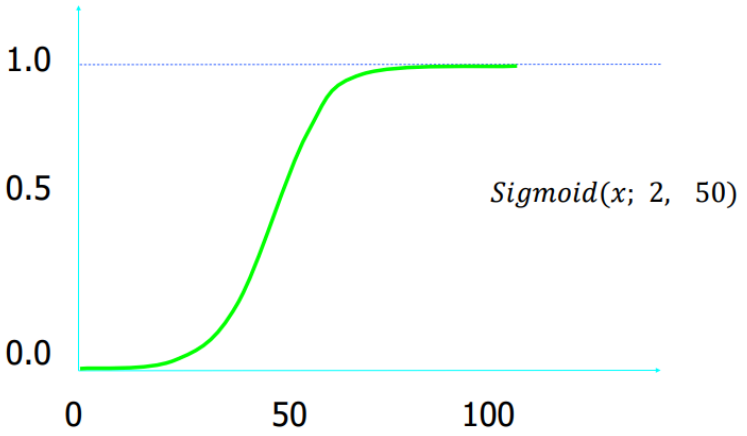
$$Trapezoidal(x; a, b, c, d) = max\left(min\left(\frac{x-a}{b-a}, 1, \frac{d-x}{d-c}\right), 0\right)$$



$$Gaussian(x; \sigma, c) = e^{-\left(\frac{x-c}{\sigma}\right)^2}$$



$$Bell(x; a, b, c) = \frac{1}{1 + \left(\frac{x - c}{a}\right)^{2b}}$$



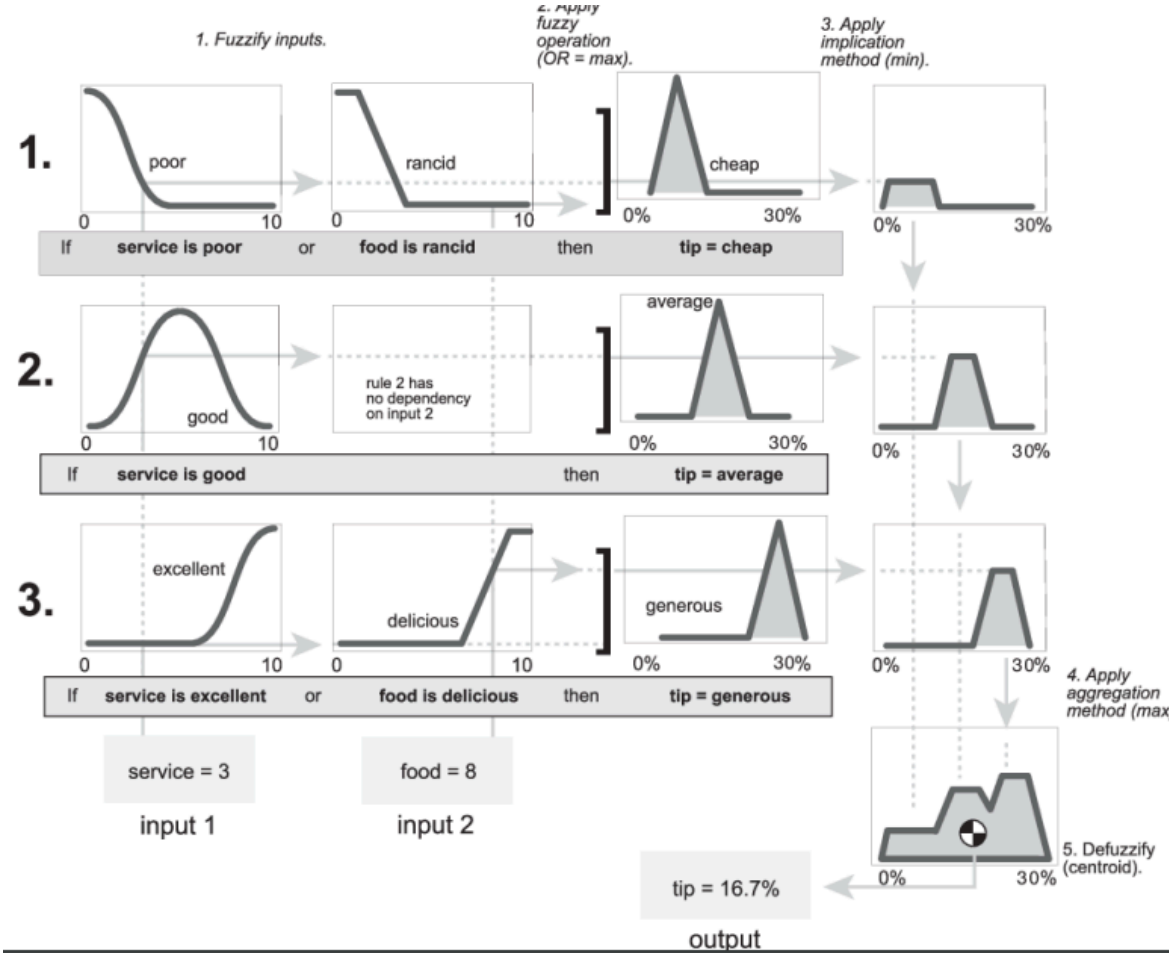
$$Sigmoid(x; a, c) = \frac{1}{1 + e^{-a(x-c)}}$$

Rules in Fuzzy sets are expressed in terms of linguistic variables → IF x is A THEN y is B

- We use words for logic operators like in python → “and”, “or”
- and → pick min probability
- or → pick max probability
- When aggregating the results of rules for a particular case, we take the maximum output for each membership → e.g. if we had the outputs speed.low = 0.5 and speed.low = 0.7, we take speed.low to be 0.7

Defuzzification is the process of converting a fuzzy set into a single crisp (clearly defined, binary) value that can be used in decision-making

- Many different defuzzification methods



Fuzzy Inference is the process of using fuzzy logic to transform and map crisp inputs to crisp outputs

- Fuzzify the inputs by using membership functions, then evaluate the rules on the fuzzified inputs, defuzzify the results and return them as the outputs
- In the example above, we map service=3 and food=8 to tip=16.7% → crisp to crisp even though the intermediates were fuzzy

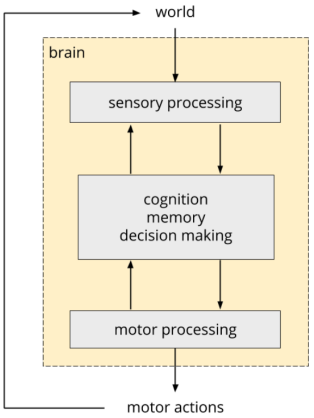
Human-Aligned Intelligent Robotics

▼ Intro

For intelligent assistive robots to successfully accomplish complex tasks, they must be able to constantly acquire new skills

- Robots can autonomously learn new tasks, but the time needed to do so can be significant

The brain-world interactive framework describes how the human brain interacts with the external world



- Our brain continually process what we sense, make decisions based on our knowledge, and process commands that execute actions interacting with the world, changing its state
- Changing the state changes what we sense, so we have a feedback loop

The 7 Levels of AI

- Simple Behaviours
  - Level 1 → Rule-based logic → basic systems that follow predefined rules
- Machine Learning
  - Level 2 → basic machine learning → simple AI that can learn patterns from data without complex understanding, and can make predictions

    - Level 3 → Pattern Recognition → advanced machine learning for identifying complex patterns → can find correlations or categories in data
    - Level 4 → Large language models → specialised AI models trained on extensive language data → can understand and generate human-like text → NLP
- Optimisation
    - Level 5 → static optimisation → AI models that optimise fixed goals without adapting over time → can find optimal solutions for a specific problem under set conditions
    - Level 6 → sequential decision problems → AI models that make a series of decisions over time, adjusting as it learns → can handle tasks involving planning and prediction
- Frontier of General (FoG) AI
    - Level 7 → Reasoning, creativity and judgement → advanced AI that truly understands, reasons about and generates ideas

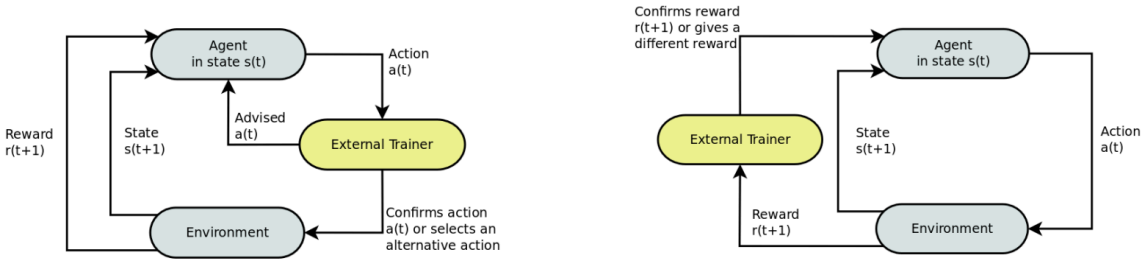
Simulation tools refer broadly to software used for simulating robots, environments and physical interactions

- Physics Engines are responsible for simulating physical interactions in the environment → e.g. gravity, collisions, forces
    - Computations can be in Joint Coordinates or Cartesian Coordinates
    - Computations in Joint Coordinates involves calculating movements based on the robot's joint angles and movements
    - Computations in Cartesian Coordinates involves calculating movements in terms of the robot's position in 3D space (x,y,z) → joints are treated as constraints that limit movement → more flexibility for simulating more general interactions/movements
- Simulation Systems combine physics engines with additional tools to create full simulations of robotic systems
    - Simulation Systems can be Platform-Specific or Generic
    - Platform-Specific Robot Simulation/Controls are designed to mimic and control specific types of robots → tailored to particular robots, and often include specialised control algorithms
    - Generic simulation systems are more flexible, and can be adapted to various types of robots and tasks → not tied to specific platforms, so they are suitable for testing a wide range of robots, designs and applications

▼ Human-in-the-loop Learning

Human-in-the-loop Learning is learning where a human is actively involved

- Human input is used in the training and decision-making process of an AI model
- Leverages human expertise to guide, improve and correct the model's learning → often results in generally better models than fully automated learning
- Sometimes an artificial agent takes the place of a human trainer and advises the learner-agent instead
- The amount and frequency of external input (feedback) can vary (periodic vs conditional vs continuous)
- Consistency of the feedback refers to how predictable and reliable the feedback provided is
    - Consistent feedback would always follow the same rules and provides the same clear guidance every time → easier to understand feedback and improve
    - Inconsistent feedback may not follow the same rules → i.e. feedback may or may not be given in the same scenario, and feedback differs
- If advice is assumed to be fully observable, the learner-agent assumes it has complete access to all the feedback given by the trainer, allowing it to rely on the feedback entirely → does not have to infer or guess missing information



A. Policy shaping

B. Reward shaping

Learning by Feedback can be done in one of two ways

- Policy Shaping → trainer can change the action taken/selected by the learner
- Reward Shaping → trainer can modify the proposed reward for the learner's action

Interactive Reinforcement Learning (IntRL) with a good teacher can notably speed up convergence

- A good teacher can guide agents with diverse behaviours → must be able to understand and accomodate different learner agents
    - Polymath agents are those that are versatile, flexible and able to adapt to a variety of tasks/strategies

    - Specialist agents are those that are highly skilled in a single specific area or approach
- To guide agents well, teachers must offer balanced, consistent feedback, while still supporting exploration

Persistent Rule-Based IntRL

- Introduces persistence to IntRL → introduces information retention and reuse
- Allows users to provide information/feedback in the form of rules rather than recommendations based on state-action
    - Rules represent known, best-practice actions or constraints for certain states in the environment
    - Rules are persistent → they remain active throughout the learning process instead of just at the start → continually influences the learner's decisions
    - Persistent influence helps learner stay aligned with certain critical behaviours or safety protocols → allows stabler/safer exploration in high-risk environments
- Probabilistic Policy Reuse (PPR) is a technique to allow learner's to decide when to reuse or ignore previously given feedback → the probability of using/reusing advice may decay
- Ripple-down rules (RDR) is an iterative technique for building and maintaining binary decision trees
- This stuff reduces the number of interactions between teacher and student

▼ Robotic Sensory Modalities — Multimodal Integration

Robot Vision is how robots 'see' the world

- RGB-D sensors capture colour and depth information about scenes
- Skeleton-Tracking is Computer Vision technique used to detect and track human body. joints and movements in real time → detected joints are connected to form a 'skeleton'
    - Useful for gesture recognition, and understanding body language

Robots can incorporate LLMs as part of their intelligence framework

- Allows them to handle language-based tasks → interpret instructions, communicate with humans in more natural ways, read

Multi-Modal Associative Architectures are AI systems designed to process and integrate information from multiple sensory modalities, and to associate these different types of information to understand complex environments

- data/information modalities include visual, audio, text, etc.
- Each modality is processed by its own NN or AI model
- An Association or Fusion layer combines information from different modalities
  - Early Fusion method combines data from multiple modalities at the beginning of the processing pipeline
  - Late Fusion method processes each modality separately, combining the output at the end
- Each modality outputs its own integrated label and confidence values upon processing
- The integrated label value λ^X represents the output for modality X
- The confidence value γ^Y represents the confidence that the label value is correct for modality Y
- The Multi-Modal Integration Model fuses these integrated label and confidence values to get the multi-modal label λ^I and confidence value γ^I
- The fusion formula can differ → i.e. argmax, ln, some other transformation algo

▼ Contextual Affordances

Affordances represent characteristics of the relation between an agent and an object

- `affordance := <object, action, effect>`
- `effect = f(object, action)`
- Objects have physical properties that suggest certain uses
- Context (location, time, current task) influences which affordances are relevant → environment and robot's current goal shape how it interprets and interacts with objects
- Contextual affordances consider the robot's state → robot combines the object's affordances with the context to decide what action to take
  - `effect = f(state, object, action)`
  - Helps understand what actions are possible with an object, and what actions with an object are appropriate or useful in the specific scenario/state
- Contextual affordances can be implemented in a learning system using a neural network or with an IntRL architecture to be integrated with robots

▼ Explainable Robotic Systems

Explainable Robotic Systems are those that are designed to make their actions, decisions and underlying processes understandable to humans

- Human users must be able to correctly understand robots in the environment
- We can provide feature-based or goal-driven explanations, but not technical stuff → no "I choose action left because it maximises future collected reward" since the average user may not understand what is meant by future collected reward

Algorithm 1 Memory-based explainable reinforcement learning approach with the on-policy method SARSA to compute the probability of success and the number of transitions to the goal state.

Initialize $Q(s, a)$, T_t, T_s, P_s, N_t
for each episode **do**
  Initialize $T_{List}[]$
  Choose an action using $a_t \leftarrow$ SELECTACTION(s_t)
  repeat
    Take action a_t
    Save state-action transition T_{List}.add(s, a)
    $T_t[s][a] \leftarrow T_t[s][a] + 1$
    Observe reward r_{t+1} and next state s_{t+1}
    Choose next action a_{t+1} using softmax action selection method
    $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$
    $s_t \leftarrow s_{t+1}$; $a_t \leftarrow a_{t+1}$
  until s is terminal (goal or aversive state)
  if s is goal state **then**
    for each s,a $\in T_{List}$ **do**
      $T_s[s][a] \leftarrow T_s[s][a] + 1$
    end for
  end if
  Compute $P_s \leftarrow T_s/T_t$
  Compute N_t for each s $\in T_{List}$ as pos(s, T_{List}) + 1
end for

Memory-Based Method for explainable AI (XAI)

- Non-experts will ask "why" or "why not" regarding what an AI model does
- Memory-Based Explainable Reinforcement Learning (MXRL)
  - Uses episodic memory to refer back to previous decisions and actions and answer "why" questions → episodic memory may be a chronological list of state-action pairs
  - We use MXRL to compute the P_s, the probability or rational of taking a certain action in a particular state, and N_t, the reason why an alternative actionw as not taken
- We use P_s and N_t, as it is much more understandable for non-experts than Q-values
  - P_s is the probability of success for taking specific actions in certain states
  - N_t is the number of transitions required to reach the goal state from a given state-action pair → how far the goal is if we take this action
- Notes for psuedocode
  - T_t is the total number of times each state-action pair is visited

- T_s is the total number of successful occurrences of each state-action pair → how many episodes we visit the state-action pair in end in success

Memory-Based in a Hierarchical Scenario

- Agent uses memory and hierarchical goals to navigate or solve tasks in a structured environment
- Hierarchical goals → goals are split into multiple tasks of different levels in the hierarchy → tasks may have subtasks or objectives that must be completed before progressing to the next level
- Agent assigns priorities to different goals based on the hierarchical structure
- E.g. need to first build the foundation, then raise the structure, before finally painting the building or sth

Algorithm 2 Explainable reinforcement learning approach to compute the probability of success using the learning-based approach.

Initialize $Q(s,a)$, $\mathbb{P}(s_t,a_t)$
for each episode **do**
  Initialize s_t
  Choose an action a_t from s_t
  repeat
    Take action a_t
    Observe reward r_{t+1} and next state s_{t+1}
    Choose next action a_{t+1} using softmax action selection method
    $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)]$
    $\mathbb{P}(s_t,a_t) \leftarrow \mathbb{P}(s_t,a_t) + \alpha[\varphi_{t+1} + \mathbb{P}(s_{t+1},a_{t+1}) - \mathbb{P}(s_t,a_t)]$
    $s_t \leftarrow s_{t+1}$; $a_t \leftarrow a_{t+1}$
  until s_t is terminal (goal or aversive state)
end for

Algorithm 3 Explainable reinforcement learning approach to compute the probability of success using the introspection-based approach.

Initialize $Q(s,a)$, $\hat{P}_s$
for each episode **do**
  Initialize s_t
  Choose an action a_t from s_t
  repeat
    Take action a_t
    Observe reward r_{t+1} and next state s_{t+1}
    Choose next action a_{t+1} using softmax action selection method
    $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)]$
    $s_t \leftarrow s_{t+1}$; $a_t \leftarrow a_{t+1}$
  until s_t is terminal (goal or aversive state)
  $\hat{P}_s \approx \left[(1-\sigma)\cdot\left(\frac{1}{2}\cdot log_{10}\frac{Q(s_t,a_t)}{R^T}+1\right)\right]_{\hat{P}_s\geq 0}^{\hat{P}_s\leq 1}$
end for

Learning and Introspection-Based Methods → goal-driven explanations

- Goal-driven explanations are explanations provided by an AI agent that specifically relate to achieving a target or goal → helps non-experts understand what the agent is doing, and why its doing it in relation to the goal
- Learning-Based (algo2) methods are approaches where an AI agent learns directly from interactions with the environment → RL algos are learning-based
  - $\mathbb{P}(s_t, a_t)$ represents the probability of success for taking action a_t in state s_t
  - $\mathbb{P}$ allows agents to provide explanations based on actual experiences
  - Relies on RL to iteratively improve Q-values and update probability of success for each state-action pair
- Introspection-Based (algo3) methods are approaches where an AI agent uses internal reasoning or self-assessment to learn?

    - Uses internal metrics (e.g. Q-values) and theoretical approximations to estimate the success probability
    - $\hat{P}_s$ represents the introspective estimated success probability → allows agents to explain that an action was believed to be likely based on its internal reasoning of potential outcomes
- Learning-based (algo2) relies on RL to iteratively improve the Q-values and update the probability of success for each state-action pair