AI Klausur

GENERAL

Notions of ‚AI'

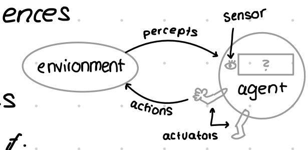
- Strong AI = Systems thinking / acting like humans
- Weak AI = Systems thinking / acting rationally => Intelligent Agents
- ‚Thinking' rationally = Laws of thought / Logic, normative rules of derivation
- ‚Acting' rationally = maximise goal achievement based on available info

→ computational limits make perfect rationality unachievable
  → we seek rational agent with best performance

AI ≠ machine Learning (ML is part of AI)

INTELLIGENT AGENTS

- Situated agents: humans, robots, thermostat, ...
  - sensors: for perceiving world → produce perception sequences
  - actuators: for acting
  - agent function: $f: P^* \rightarrow A$ percept histories → actions
  - agent program runs on physical architecture to produce f.



  Agent = Architecture + Programm
    Architecture = Programming device + sensors + actuators
    Programm = gets sensor data → returns actions for actuators

- Rationality: defined by... Performance measure (criteria of success) +
  prior knowledge of env. + percept sequence (history) = agent actions

  > *For each possible percept history, select an action that is expected to maximize its performance measure, given the evidence by the percept history and whatever built-in knowledge the agent has.*

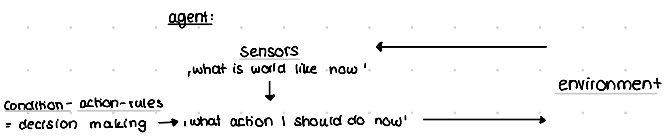
- Rational agent: does the ‚right' thing based on information / knowl.
  = exploration of world, learning, autonomy
  ≠ perfect, omniscient, clairvoyant

- Agent characteristics: PEAS
  Performance (efficiency, safety, ...)
  Environment (to consider)
  Actuators (I can use)
  Sensors (input)

- Environment types
  - fully vs. partially observable
  - deterministic vs. stochastic
  - episodic vs. sequential
  - static vs. dynamic
  - discrete vs. continuous
  - known vs. unknown
  - single-agent vs. multi-agent

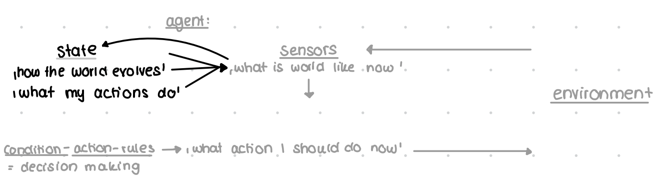
- Agent types : 4 types , hierarchy by capabilities

1. Simple reflex agent



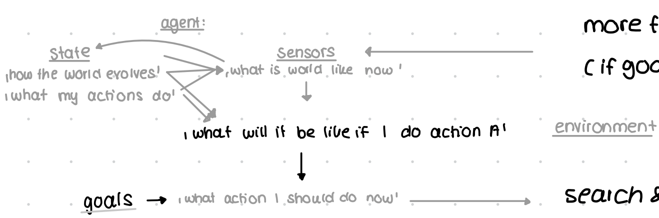
no memory
no sequences of percepts
looping possible

2. model-based reflex agent with state



memory
update world state
reason about unobservable parts
goals only implicit

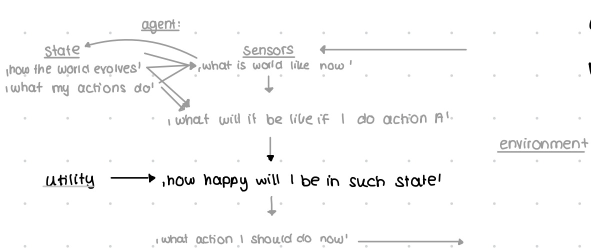
3. goal-based agent



explicit goals, world, actions & effects
more flexible better maintainable
(if goals change don't need to change rules)

search & planning of actions to achieve goal

4. utility based agent



access goals with utility function
resolve conflicting goals
only <u>expected</u> utility

PROBLEM SOLVING & SEARCH

- Search Problem Definition
  1. Initial state
  2. Successor function = set of action state pairs
  3. goal test (implicit: x has... , explicit: =x)
  4. past cost (additive) , $c(x,a,y)$ step cost from x to y with action a

  A solution is a sequence of actions leading from the initial state to the goal state
  State space must be abstracted (easier than real problem)

- Basic search algorithms: offline = world doesn't change
  1. Tree-like search:
     generate successors of already explored states (expanding states)
     maintain list of nodes available for expansion (frontier)

```
function TREE-SEARCH (problem) returns a solution or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        f the node contains a goal state then return the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier
```

for repeated states
linear problem might turn
into exponential one!

  2. Graph-search:
     besides frontier maintain explored set

```
function GRAPH-SEARCH (problem) returns a solution or failure
    initialize the frontier using the initial state of problem
    initialize the explored set to be empty
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        add the node to the explored set
        expand the chosen node, adding the resulting nodes to the frontier
        only if not in the frontier or explored set
```

optimization: reached set: all <u>states</u> with generated nodes

- Implementation
  - A node (Datastructure) consists of:
    state, parent, children, depth, path cost $g(x)$
  - Frontier is a queue

```
function NODE(problem, parent, action) returns a node
  return a node with
    STATE = problem.RESULT(parent.STATE, action),
    PARENT = parent, ACTION = action,
    PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

python etc.: **yield** NODE(STATE = ..., PARENT = ..., ACTION=..., PATH-COST=...)

- Search strategies:
  - uninformed search = basic algorithms, only info in problem definition
  - informed search = information about solution cost (heuristic)
  - local search = 'history-less', one step changes

- Search strategy evaluations:
  - completeness = does it always find solution if one exists?
  - optimality = does it find least-cost solution?
  - time complexity = number of nodes generated / expanded
  - space complexity = max. number of nodes in memory

  For time & space:
  - b = maximum branching factor
  - d = depth of least cost solution
  - m = maximum depth of state space (∞ if loop)

  A tree has b^m nodes: level 0: 1, level 1: m level 2: m*m level 3: m^3

  The root has level 0!  m = 2!

UNINFORMED SEARCHES

- Breadth-First-Search (BFS): expand shallowest unexpanded node
  Frontier is FIFO, successors go at end, reached set for no loops
  Goal test at generation time before putting child into frontier
  → complete if b is finite, optimal if step cost = 1,
  time & space $O(b^d)$ if goal test gen. time, $O(b^{d+1})$ if exp. time

- Uniform-cost-search: takes past cost into account
  ≥ Best-First-Search: expand least-cost unexpanded node
  Frontier is priority queue sorted by evaluation function f = path cost
  Goal test at expansion time → stop at optimal, not <u>any</u> solution
  → complete if step cost ≥ ϵ (lower bound), optimal: yes $O(b^{1+\lfloor C^*/\epsilon \rfloor})$
  time & space: n with $g(n) \leq C^*$ C^* = cost of optimal solution, if cost = 1 ⇔ BFS

- Depth First Search (DFS): expand deepest unexpanded node
  Frontier is LIFO queue, put successors at front
  → complete: no in infinite spaces / loops, optimal: no, stop at first not best sol
  ↳ tree like version. Keep explored set → complete in finite spaces
  time: $O(b^m)$. if solutions are dense may be faster then BFS, space: $O(bm)$

variant: only keep 1 successor node at a time & backtrack store only 1 branch

- Depth-limited search (DLS)
  DFS with depth limit ℓ → report cutoff at limit

- Iterative deepening search (IDS): Increase limit ℓ iterative
  → complete: yes, optimal: if step cost = 1; mildly more expensive th. BFS
  time: $O(b^d)$, space: $O(bd)$

- Bidirectional search (BDS): needs invertible actions
  go forward from initial & backward from goal state → meet at $d/2$
  ⇒ 2 queues (BFS or UCS) + check whether exp. node in other set
  → complete: if step cost ≥ ϵ, optimal: yes, both if b is finite
  time & space: $O(b^{d/2})$

Criterion	BFS	UCS	DFS	DLS	IDS	BDS
Complete?	Yes$^{\alpha}$	Yes$^{\alpha,\beta}$	No	No	Yes$^{\alpha}$	Yes$^{\alpha,\delta}$
Optimal?	Yes$^{\gamma}$	Yes	No	No	Yes$^{\gamma}$	Yes$^{\alpha,\delta}$
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon\rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon\rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

α if b is finite
β if step costs $\geq \epsilon$ for positive ϵ
γ optimal if step costs are all identical
δ if both directions are BFS or UCS

All only complete if b finite

→ time can be overcome with technical improvements,
  memory is crucial ⇒ IDS with linear space!
→ Graph search can be exponentially more efficient than tree-like-S

- Klausurfrage: Is IDS optimal if costs increase with depths?
  = monoton ansteigende Kosten ⇒ ja! (laut Wikipedia)

INFORMED / HEURISTIC SEARCHES

= using problem-/domain specific knowledge
= evaluation function $f(n)$ estimating real-life / optimal function $f^*(n)$

→ heuristic function $h(n)$ estimates minimal cost from state n to goal
  $h(goal) = 0$ $h(n)$ gets smaller to goal   computing $h(n)$ has low cost
- $h(n)$ admissable: for every node: $h(n) \geq 0$ $h(goal) = 0$
  $h(n) \leq h^*(n)$, $h^*(n)$ = true costs ⇒ $h(n)$ is lower bound = optimistic
- $h(n)$ consistent: for every node n & successor n' & cost $c(n, \overset{\text{action}}{a}, n')$
  $h(n) \leq c(n, a, n') + h'(n)$ → then $f(n) = \underset{\text{spent cost}}{g(n)} + h(n)$ is non-decreasing
- how to get admissable heuristics? derive from subproblem or
  derive from exact resolution cost of relaxed version of problem

consistent: $f(n') = g(n') + h(n') = g(n) + c(n,a,n') + h(n') \geq g(n) + h(n) = f(n) \Leftrightarrow f(n') \geq f(n)$

- For admissable heuristics: better the nearer to real cost
  - h_1 & h_2 adm. → h_2 dominates h_1 if $h_2(n) \geq h_1(n)$ for every node n
    => if h_2 dominates h_1, it's better for search
  - h_1 & h_2 adm → $h(n) = \max(h_1(n), h_2(n))$ is adm. & dominates h_1 & h_2

- Greedy Search: uses $f(n) = h(n)$, only local info, not already spent costs
  Expand node with smallest f-value
  → complete: yes, if with loop checks, optimal: no, but okay
  time & space: $O(b^m)$ bad!

- A*-search: uses $f(n) = g(n) + h(n)$ → avoid expanding already expensive paths
  $g(n)$: path cost so far from start to n
  $h(n)$: estimated costs from n to goal
  $f(n)$: estimated total path cost through n to goal
  → if $h(n)$ is admissable, A*-search (tree search) is optimal
  ↳ A* (graph search) can discard optimal solutions even if $h(n)$ is adm.

  find node via path that is not optimal first → already in frontier/reached set

  → if $h(n)$ is consistent, A*-search (graph search) is optimal
  ⇒ A* search expands nodes in order of increasing f-values:
  gradually add „f-contours": contour f_i has all nodes with $f < f_i$, where $f_i < f_{i+1}$
  A* search expands all nodes with $f(n) < C^*$, some nodes with $f(n) = C^*$, no nodes with $f(n) > C^*$, fewest nodes safely if $h(n)$ consistent
  → complete: yes, unless infinite number of nodes with $f(n) \leq f(G)$, optimal: yes
  time: exponential in $\epsilon \times d$ with relative error $\epsilon = \frac{h(n_0) - h^*(n_0)}{h^*(n_0)}$ $h^*(n_0) = C^*$
  space: $O(b^d)$ exponential (stores every expanded node)

- Memory bound search: remedies of A* exp memory consumption
  - Avoid duplication in reached, frontier
  - Iterative deepening A* (IDA*) ≈ IDS
    → with limit cost C → next limit: smallest value $f(n) > C$ ... = expanding f-contours
    → no storage of nodes beyond optimal cost: space $O(d)$ linear
  - Recursive First-Best Search (RFBS)
    f-limit = value of best alternative path than current node n
    → if $f(n) > f$-limit: unwind back to alternative path
    & change f-values of nodes to best f-value of their children (sub-tree)
    → optimal if $h(n)$ admissable but more time for regenerating paths in exchange for saving space!

SEARCH IN COMPLEX ENVIRONMENTS

→ no model of domain / info of heuristics available → generate info locally

LOCAL SEARCH ⇒ suitable for online environment

Path not relevant, only goal state, state space = set of complete configurations

⇒ find optimal configuration (e.g. Traveling Salesmen Problem (TSP))

- Hill climbing (Gradient Descent / Ascent):

  try out highest-valued neighbour / successor for better solution

  → problem: might get stuck on local maxima / flat maxima / ridges (sequences of local maxima)

  ⇒ remedies: random-restart ⇒ complete if done often enough, side moves, stochastic neighbor selection

- Simulated annealing:

  allow intermediate bad solutions / moves → escape local maxima gradually

  start with t = 1, decrease t gradually „cooling of"

  choose random successor of current

  if successor is better than current, choose successor

  if successor is worse, choose it with a probability depending on t

  The higher t, the more accept bad moves

  once t = 0 return current

  → will find optimal solution if cooling-down slow enough

  ⇒ ‚slow enough' can be worse than exhaustive search!

- local beam search

  keep k states in memory & choose top k of all their successors

  → searches finding good states recruit other searches

  → problem: might all end up on same local hill

  → remedy: choose k successors randomly, but biased towards good ones ↳ ~ natural selection

  → problem: states get too close → remedy: use stochastic variant

- Genetic algorithms

  = stochastic local beam search + successors from combining

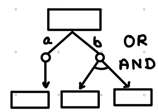
1. Select parents weighted by fitness (quality)
2. Crossover: reproduce child
3. With small probability mutate child

→ do so until an individual is fit enough or time is up

=> requires states (individuals) encoded as strings / programs / ...
=> crossover can produce solutions distant from parents
=> crossover only helps if substrings are meaningful = components / blocks
else no more helpful than shuffling randomly

CONTINUOUS STATE SPACES e.g. minimizing a function $f(x)$
Approaches:
- Constraint optimization: optimize objective function f
  convex optimation: linear programming
- Discretization: turn continuous space into finitely many successors
  empirical gradient search: steepest ascent hill-climbing
- Gradient method: analytical approach: find optimum $f'(x)=0$

SEARCH WITH NONDETERMINISTIC ACTIONS = follow-up states not determined
belief-state = set of states the agent is possibly in
→ solution must take belief states into account



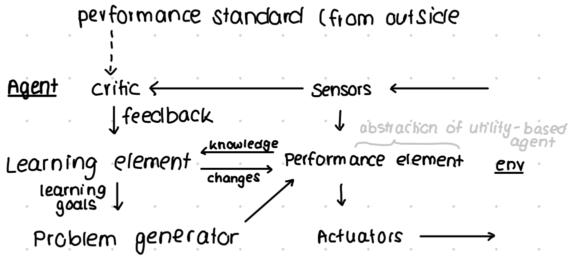
- And-Or-Tree:
  OR-Nodes = choice of action in some state
  AND-Nodes = possible action outcomes
  => solution is a subtree that has a goal state at each leaf
  → terminates in finite spaces, can use BFS, UCS, A* with adm heuristic
- cyclic solutions: if loops at leaf must consider worst case
  → while loop with chance of escape, use labels
  → not applicable if hidden variables prevent loop exit

ONLINE SEARCH: in dynamic world = environment changes
lack of information, action effects unclear → search as you go
Action(s) = set of actions doable in state s learn RESULT (s, a)
=> reach goal from initial state, local expansion s → s', no jumping
→ dead ends: backtrack, consider goal reachable from any space
- Implementation via DFS suitable: From state try all actions (save 'untried'),
  until goal or dead-end → backtrack (save 'unbacktracked' for each state)
- Performance measuring: competitive ratio: cost of path traveled / optimal path
  characteristics: size of state space
- online local search by hill climbing: no random restart, but random walk
  = randomly picking actions → in finite spaces complete, but exponential time

LEARNING FROM EXAMPLES

Learning = modifying agent's decision mechanism to improve performance
→ for unknown environments, when programming is not possible

- Learning agent architecture:
  - Performance element: select external actions
  - Critic: performance / result assesment
    → view past outcomes
  - Learning element: make improvements
  - Problem generator: suggest actions for new experiences



E.g. taxi: PE = driving, Critic: customer feedback, LE: break softly, Problem gen: try breaks on rain

- Learning element (LE) makes changes on knowledge elements
  Design depends on: type of performance element, functional component, representation of funct. comp., feedback-type

Performance element	Functional component	Representation	Feedback
Alpha-beta search	Evaluation function	Weighted linear function	Win/loss
Logical agent	Transition model	Successor-state axioms	Outcome
Utility-based agent	Transition model	Dynamic Bayes net	Outcome
Simple reflex agent	Percept-action function	Neural net	Right action

- Representation of states of the world:
  - atomic = monolithic states (blackbox) with labels → search algorithms
  - factored = with attributes with values → planning, CSP
  - structured = dependency / relationships betw. attributes → Logic, bayesian network

- Learning modes:
  - unsupervised: no feedback → clustering, concept formation
  - supervised: correct answer for each instance ('label')
    → requires teacher / labelling   reward ≠ correct answer
  → in practice: semi-supervised = mix of labelled & unlabelled
  - Reinforcement: occasional reward / punishment ≈ payoff

* classification methods (s.b.)
  - support vector machines: maximum margin separators
  - k-nearest neighbour: the larger k, the less overfitting

INDUCTIVE LEARNING = Learn functions from examples

f is the target function, an example is pair $(x, f(x))$ = expected (input, output)

Given an finite training set $(x_1, y_1) \dots (x_n, y_n)$ of examples y_i = ground truth

find a function h that approximates f: $h \approx f$ h = hypothesis

→ if $h = f$, the learning problem is realizable

- types of outputs:
  - classification = one of finitely many values * (s.a.)
  - regression = a number

- assumptions (highly simplified):
  → no prior knowledge, deterministic observable environment, given examples (selection of new hard!), agent wants to learn f

- Inductive learning method:
  Construct / adjust h to agree with f on training set (= h consistent)
  → curve fitting linear / quadratic / polynomial / complicated
  < degree k for k examples
  ⇒ 'Ockham's razor': maximize simplicity under consistency

- complex vs. simple hypotheses:
  - semantic: bias-variance trade-off
    bias = deviation from expected output over different training sets
    variance = change in h by change in training set
  v↑ → complex h: fit data well ⇒ overfitting = too much adjusted to particular data
  b↑ → simpler h: generalize better ⇒ underfitting = no pattern found in data
  - computational: issue = computational complexity $h^* = \text{argmax}_{h \in H} P(data|h) \cdot P(h)$
    trade-off expressiveness h space ↔ finding a good h in it

- measuring learning performance: Hume's Problem: $h = f$?
  - use computational / statistical theorems
  - try h on new test set from same distribution as training set
    → Learning curve: x = % of correct y = num of examples (training set size)
    ↳ depends on: realizability of f (e.g. missing attributes, h space to restrictive)
    (only linear...)
    redundant expressiveness (loads of irrelevant attributes)

DECISION TREES = A possible representation of h

uses attributes = factored representation

input vector of values of attributes → outputs decision

tree = sequence of tests on attributes (nodes), for each value 1 child-node

→ each leaf gives decision

- boolean classification would be only values $\top$/yes, $\bot$/no

⇒ Decision trees can express <u>any</u> function

⇒ There is a trivial consistent decision tree for any training set: 1 path to leaf per ex

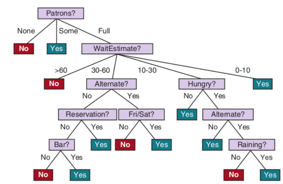
⇒ aim: generalize = find small flat consistent dt

⇒ recursively choose most ‚significant' attribute

→ if no more examples/attr. before classif., return plurality value

→ if attr. empty: nondeterministic domain, hidden attributes

The "true" tree for deciding whether to wait:



- Choosing attributes: good attributes split examples into classes
- Greedy approach: pick attribute with most classifications
- Information gain: not only max. classif., consider entropy

  The more uncertain the outcome (= the higher the entropy), the more info an attribute contains when classified. For $\frac{1}{2} \leftrightarrow \frac{1}{2}$ max = 1 bit, $0{,}1 \leftrightarrow 0{,}99$ low

- Entropy $h(\langle p_1, \dots, p_n \rangle) = \sum_{i=1}^{n} -p_i \cdot \log_2(p_i)$

  for $n=2$ (boolean) $h(\langle p_1, p_2 \rangle) = B(p_1) = -p_1 \cdot \log_2(p_1) - (1-p_1) \cdot \log_2(1-p_1)$

- Remainder $Rem(A) = \sum_k \frac{p_k + n_k}{p+n} \cdot B\left(\frac{p_k}{p_k + n_k}\right)$ bits to classify attr.
- Gain$(A) = B(p/(n+p)) - Rem(A)$ If $Rem(A) = 0$ → all examples classified

  Bits to classify example set = entropy - Rem = how much entropy stays

  Gain = a lot of unsureness, but few stays

- Broadenings: missing values, continuous att → use split points, c. output → function

ALTERNATIVE HYPOTHESES

For n attr. there exist 2^{2^n} non-equiv. decision trees, for conj.: 3^n

more expressive h-space = target f better expressed, num of h ↑, BUT worse expressions

- Decision Lists = cascaded if statements:

  if conj. h_1 then O_1 (yes/no) elif conj h_2 then O_2 ...

  ⇒ can express all boolean functions

  k-decision list: each conj. max k literals → PAC learnable

= how many examples do we need to learn f?

- PAC (Probably approximately correct) learnability
  seriously wrong h will be identified with high prob. after small num of examples
  ⇔ a h with high num of examples is probably approx. correct
  ↳ Error rate error(h): average error of h → h approx correct if error(h) < ϵ
  ↳ Probability P(bad h agrees with N examples) $\leq (1-\epsilon)^N$
  ⇔ $N \geq \frac{1}{\epsilon}(\ln \frac{1}{\alpha} + \ln|H|)$ examples are sufficient (for k-DL: $|H| = O(n \cdot \log_2(n^k))$)

- model selection: choose h
  - Overfitting: h may consider irrelevant attributes (less likely with more examp) more likely with more attr.
    ⇒ remedy: decision tree pruning: split over relev. Attr. using info gain / statistic
  - k-fold-cross-validation = Testing Training + Validation + Test set
    use part of indep. & identically distibuted data for test & training
    1. split examples E into k equal sized sets $F_1, \dots, F_k$
    2. do k rounds of learning: use F_i as validation set, $E \setminus F_i$ as training
    3. average scores

⇒ construct model h by varying params, using a learner & cross-validation
= learning-algorithm
keep validation error low: Overfitting starts when model capacity gets close to interpolation point (= almost goes through all test points)
approximation interpolation ⇒ probably overfitting

LINEAR REGRESSION → learning continuous valued functions
$h \approx f$ minimize loss error on examples $h_{a,b}(x) = a \cdot x + b$
square loss of h on (x,y): $Loss(h; (x,y)) = (y - h(x))^2$
→ finding a & b easy, may be computed by gradient search

- univariate linear regression: 1 input, 1 output variable
  $h_w(x) = w_0 + w_1 \cdot x$
  $Loss(h_w, (x,y)) = (y - h_w(x))^2$ $\quad Loss(h_w) = \sum_{i=1}^N (y_i - h_w(x_i))^2$
  ⇒ find optimal $w^* = \text{argmin}_w(Loss(h_w))$ → $\partial Loss / \partial w_i = \Delta Loss(h_w) = 0$
  ↳ solution: $w_0 = \cdots$ Formel $\quad w_1 = \cdots$ Formel

instead of Formel

- Gradient descent: compute optimal values w^* incrementally
  = vary parameter w_i minimizing loss $\quad Loss(w)$ quadr., $\partial loss / \partial w_i$ linear
  update rules:

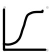
$$w_0 \leftarrow w_0 + \alpha(y - h_{\mathbf{w}}(x)) \qquad w_0 \leftarrow w_0 + \alpha \sum_{j=1}^N (y_j - h_{\mathbf{w}}(x_j))$$
$$w_1 \leftarrow w_1 + \alpha(y - h_{\mathbf{w}}(x)) \cdot x \qquad w_1 \leftarrow w_1 + \alpha \sum_{j=1}^N (y_j - h_{\mathbf{w}}(x_j)) \cdot x_j$$

→ as long as $Loss \neq 0$ / not converged, update w

- multi-variable linear regression:
  $f(x)$, $x = x_0, \dots, x_n$ $x_0 = 1$ fixed $h_w(x) = \sum_{j=0}^{N} w_j \cdot x_j = w \cdot x$
  ⇒ find optimal $w^* = \mathrm{argmin}_w(\mathrm{Loss}(h_w)) = (X^TX)^{-1}X$ $X = (x_{i,j})$
  update rule: $w_i \leftarrow w_i + \alpha \cdot \sum_j (y_j - h_{\mathbf{w}}(\mathbf{x}_j)) \cdot x_{j,i}$
  → Issue: Overfitting due to multiple attributes → add regulation term $\lambda \cdot$ complexity

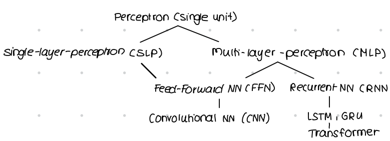
- Linear classifiers
  - hard treshhold: use linear function / seperators $h_w(x) = \begin{cases} 1 & \text{if } w \cdot x \geq 0 \\ 0 & \text{else} \end{cases}$

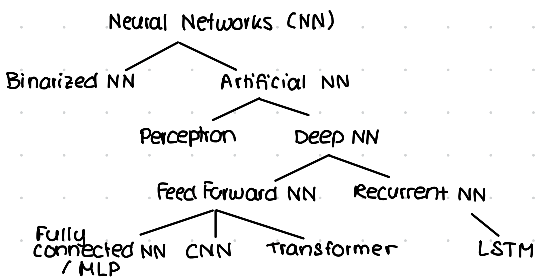
    → does not work with gradient descent ($\partial loss / \partial w_i$ is alway 0 or undef (at 0))
    not differentiable
    ↳ use update rule: $w_i \leftarrow w_i + \alpha(y - h_{\mathbf{w}}(\mathbf{x})) \cdot x_i$
  ⇒ for any linear seperable data set this rule converges to a consistent function = can learn any lin. sep. $f(x)$ from sufficient data
  ⇒ if not lin. sep.: converges for decaying α to min-err solution



  - logistic regression: softened treshhold
    sigmoid function $g(x) = 1 / (1 + e^{-w \cdot x})$
    → output $h_w(x)$ is probability for class membership
    update rule: $w_i \leftarrow w_i + \alpha(y - h_{\mathbf{w}}(\mathbf{x})) \cdot h_{\mathbf{w}}(\mathbf{x}) \cdot (1 - h_{\mathbf{w}}(\mathbf{x})) \cdot x_i$

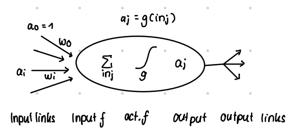
NEURAL NETWORKS



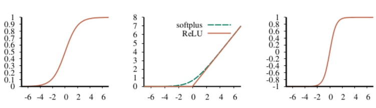


PERCEPTRONS

- Perceptron = 1 unit
  = linear sum of weighted inputs a_i: $in_j = \sum_i w_{ij} a_i$ a_0 fixed (→ w_0)
  → activation function decides whether perceptron is activated / value
  ⇒ 1 output $a_j = g(in_j)$



- Activation functions:



Sigmoid $= \left(\frac{1}{1+e^{-x}}\right)$ ReLu $= \max(0, x)$ tanh

Relu not fully differentiable
→ softplus = glatte Approx v. ReLu → derivate is sigmoid
→ changing w_0 shifts the treshhold function
→ evaluation discrete o. continuous

- Perceptron Learning: $h_w(x) = g(in)$ where $in = \sum_i w_i x_i = w \cdot x$

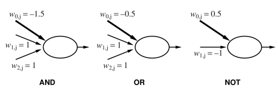
  Learn $f(x)$ by adjusting w to reduce error on training set (regression)

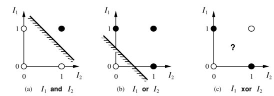
  $Err = y - h_w(x)$ square-loss $loss(w) = (y - h_w(x))^2$ → search for minimal loss with gradient descent

  perceptron learning rule (update rule):

  $w_i = w_i + \alpha \cdot Err \cdot g'(in) \cdot x_i$ α = learning rate (step size)

- Expressiveness of perceptrons:
  - single perceptron/neuron (with hard treshhold): complete basis of boolean functions if linear seperable: AND, OR, NOT, not XOR
  - network: arbitrary functions





$h_w(\mathbf{x}) = \sum_i w_i x_i \geq 0$ (i.e., $h_w(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} \geq 0$)

NEURAL NETWORKS (NNs)

- Definition: NN is a function $f(x)$ that maps $x = (x_1, \dots, x_n)$ of u^{in} to $y = (y_1, \dots, y_m)$ of u^{out} composed of units: input units u^{in}, output units u^{out}, processing (hidden) units

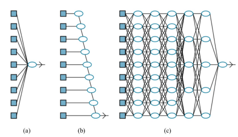
  + links/arcs from u_i to u_j with weight $w_{i,j}$

  $f(x) = h_w(x)$ where $w = (w_{i,j})$ is an $N \times N$ matrix

- Network structures

  a. perceptron (short paths) b. decision list (some long paths)

  c. deeper network (long paths)



- Feed-Forward Networks (FFN)

  = uni-directional = directed acyclic graphs (DAD)

  node in layer i is directly connected to nodes in layer i+1, i-1

  => No internal state! = simple reflex agent implementation

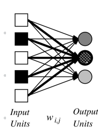
  => Types: Single-layer-/multi-layer-perceptrons

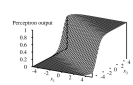
- Single-Layer-Perceptron (SLP): no hidden units, but multiple outputs

  → output units operate seperately = no shared weights

  → adjusting weights moves location/orientation/steepness of cliff

  → can view this as single perceptron: adjust w_{ij} to w_i

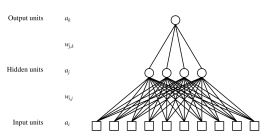




- Multi-Layer-Perceptron (MLP):

  Layers usally fully connected

  multiple outputs possible



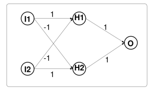
- Expressiveness of MLPs:
  - 1 layer (SLP) = all linear seperable functions
  - 2 layers = all continuous functions at arbitrary precision
    -> requires exponentially many hidden units ≠ realistic

- XOR with MLP
  hard treshhold $g(x)=1 \Leftrightarrow x \geq 1$

I1	0	0	1	1
I2	0	1	0	1
I1 xor I2	0	1	1	0

in/out

H1	0/0	-1/0	1/1	0/0
H2	0/0	1/1	-1/0	0/0
O	0/0	1/1	1/1	0/0



- Network learning = constructing weights for NNs
  Network structure fixed → adjust weights to learn function
- SLP: use perceptron learning rule for each output
- MLP: weights affect multiple outputs → push error back & adjust weights
  1. Compute loss of whole network = sum up all gradient losses of outputs
  2. Push back error by dividing it on contributing weights

Learning rules:

output — how much this contributes to error — error — how much this input changes output

Output Layer a_k: $w_{jk} \leftarrow w_{jk} + \alpha \cdot a_j \cdot \Delta_k$; $\Delta_k = Err_k \cdot g'(in_k)$

hidden layer a_j: $w_{ij} \leftarrow w_{ij} + \alpha \cdot a_i \cdot \Delta_j$; $\Delta_j = g'(in_j) \cdot \sum_k w_{jk} \cdot \Delta_k$

s.o. — input contribution — outputs to which it is connected — the outputs errors

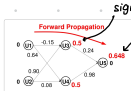
node u_j is responsible for fraction of error Δ_k at output layer

Examples:

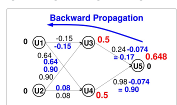
activation: sigmoid function

$$g(x) = \sigma(x) = 1/(1+e^{-x})$$
$$g'(x) = \sigma(x) \cdot (1-\sigma(x))$$

sigmoid function result



expected: $f(0,0)=0$



Calculation (numbers rounded):
- output unit a_k: $\Delta_k = a_k \cdot (1-a_k) \cdot (y_k - a_k)$
- hidden unit a_j: $\Delta_j = a_j \cdot (1-a_j) \cdot \sum_k (\Delta_k \cdot w_{j,k})$

$$\Delta_5 = Err_5 \cdot g'(in_5) = a_5 \cdot (1-a_5) \cdot (y_5 - a_5) = 0.648 \cdot (1-0.648) \cdot (0-0.648) = -0.148$$

$$\Delta_4 = g'(in_4) \cdot \sum_k w_{4,k}\Delta_k = a_4 \cdot (1-a_4) \cdot (w_{4,5} \cdot \Delta_5) = 0.5 \cdot (1-0.5) \cdot (0.98 \cdot -0.148) = -0.036$$

other Δ_j values: $\Delta_3 = -0.009$, $\Delta_2 = 0.011$, $\Delta_1 = -0.022$

Computation of the weight changes (learning rate $\alpha = 1$):

$$w_{4,5}: \quad \alpha \cdot a_4 \cdot \Delta_5 = 1 \cdot 0.5 \cdot (-0.148) = -0.074$$

Updated weight $w_{4,5} = 0.90$

Other weight changes: computations similar

- Applications: e.g. handwritten digit recognition: error nowadays 0,23%
- Aspects: ⊕ good for complex pattern recognition / unstructured input
  less need for determining relevant input factors
  ⊖ choice of NN hard, needs good training material, results not easy to understand

- Training issues & Solutions:
  - Overfitting → improve Generalization:
    - choosing right NN architecture: data: CNN → images, RNN → sequential data
      deeper networks better, adversial examples → robuste Modelle,
      (kleine Änderung in Eingaben, die NN output verändern)

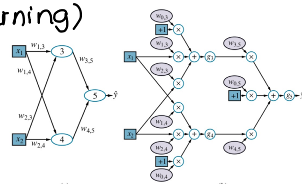
Anpassungen d. Parameter (Hyperparameter-Tuning)
NAS = Neural Architecture Search sucht gute NN-Strukturen
- weight decay = regularization: add penalty $\lambda \sum_{ij} w_{ij}^2$ to loss function
  → big values will be restricted (≠ Overfitting)
- dropout = at each training step randomly deactivate some neurons
  → NN not dependent on some neuron

- Slow convergence & local maxima: common to gradient descent
  - Stochastic gradient descent: uses minibatch = small set of examples from training set ⇒ faster → parallel computing possible
  - Batch normalization: rescale values at internal layers per minibatch
  → decrease learning rate & increase minibatch size over time
  ⇒ If loss surface is convex: will find global minimum guaranteed
  else no guaranty

- Exploding & vanishing gradients → in deeper / RNNs
  when using exp. act. f (softmax, sigmoid) & iterative recursive computations
  - Exploding: when weights $w > 1$ → using clipping possible
  - Vanishing: $w < 1$ → error signal extinguishes → NN degenerates
  use: - batch normalization, non-exp. act. f., LSTM for RNN (s.b.), RNs

- Residual Networks (RNs) against vanishing gradients
  = perturb, not replace previous' layer representation
  instead of $z_i = f(z_{i-1}) = g_i(w_i \cdot z_{i-1})$ use
  $z_i = g_{ri}(z_{i-1} + f(z_{i-1}))$ g_r = act. f of residual layer
  → info is by default propagated → no need to learn all the time

- Computational graphs
  = NN seen as a data flow graph / computation graph u_i (shows calculations)
  = a circuit with multiplication ×, addition + and activation gates g_i as nodes
  at layer i: weights $w^{(i)}$, activation function $g^{(i)}$
  e.g. $h_w(x) = g^{(2)}(w^{(2)} g^{(1)}(w^{(1)} x))$
  → computations forward (output) / backwards (weight learning) using automatic differentiation



⇒ make calculations in NNs easy / efficient
~ basis for NN-Software

- Input layer : Input nodes = Input $x = x_1, \dots, x_n$
  Types of data-input:
  - Factored data with attributes: boolean 1/0, integer numbers
  - Images: $X \times Y$ RGB image: array-like internal structures (tensors); adjacency matters
  - Categorical data: value range $v_1, \dots, v_d$ → One hot encoding: d input bits ⇒ v_i: $b^{(i)} = \underbrace{0,1,0,0\dots}_{\text{pos. } i}$

- Output Layer : encoding of output similar to input
  Loss function: - square loss
  - negative Likelihood: $w^* = \text{argmin}_w - \sum_{j=1}^{N} \log P_w(y_j \mid x_j)$
    → log convenient: sum instead of product; minimizing = maximizing probability
  - cross entropy loss $H(P, Q)$ = dissimilarity between distributions P & Q
    → $P :=$ true distribution P^*, $Q :=$ hypothesis $P_w(y \mid x)$ (y has to be interpretable as probability)

  Multi-class-classification: output is vector of numbers
  → Softmax-layer turns these into probability distribution e.g. $(5, 0, 2) \rightarrow (0.97, 0.01, 0.29)$
  sigmoid = P für 2 Klassen e^x accentuates differences in output
  Regression output:
  Linear output layer = no activation function, just interpret number as gaussian prediction → = minimizing squared error = linear regression

- Hidden Layer: computed values in layers = diff. representations of x
  → complex transformation decomposed into simple learnable transformations
  → intermediate representations might be meaningful e.g. edges ⇒ faces
  typically ReLu & Softplus (≠ vanishing gradients), earlier sigmoid, tanh
  ⇒ deeper & narrower NNs better than shallow & wide ones

- Gradient computation:
  - usual: $\partial Loss / \partial w_{ij} = -2 a_i \Delta_j$ update $w_{ij} \leftarrow w_{ij} + \alpha \cdot a_i \cdot \Delta_j$
  - in practice: stochastic gradient descent using minibatches
    compute derivates $\partial Loss \; \partial h \; \partial g_h$ and backpropagate along nodes
    → at node $h = w$ $\partial Loss / \partial w$ is computed ↳ linear in n, but large memory requirement

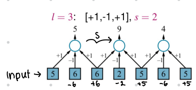
DEEP LEARNING

→ large data sets available crucial, efficient hardware (GPU, chips)
- Choosing NN Architecture: distinction not absolute! e.g. DeepL = CNN
  - CNNs ⇾ Computer Vision: feature extractors along spatial grid
  - RNNs ⇾ Natural language processing: update rules in streams of sequential data
  - Transformers ⇾ everything

- Convolutional NNs (CNNs) & Computer Vision (CV)
  - → CV: image classification, recognition, formation, ...
- — Encoding of images: vector of pixels too big, but image data has spatial invariance → encode small regions e.g. 3x3, 5x5
- ⇒ initial CNN-layer = spacial local connections (regions), not fully connected
  - ⇒ each neuron has a receptive field of e.g. 3x3, 5x5, ...
- Input image becomes „tensor" = array of any dimension, e.g. vector, matrix, ...
  - → keeps adjacency, allows to describe operations on local regions
  - → tensor operations e.g. pooling, convolution, matrix multiplication
    - ↳ described in computation graphs where each operation a node
  - → hardware: GPUs, TPUs (tensor processing units) → parallel processing
- — Kernel / Filter = pattern of weights replicated across local regions e.g. 3x3 matrix
  - → in CNN: multiple (d) kernels
- — Convolution $k * r$ = applying kernel k to region r

- CNN architecture:
  - input layer → convolution layers / pooling layers
  - → residual layer (to resolve vanishing gradients)
  - → output: fully connected layer (with softmax) to classify

- Convolution / Cross-correlation:
  - = applying kernels to pixels of the image $z = k \cdot x$ $\quad z_i = \sum_{j=1}^{l} k_j \cdot x_{i+i-(l+1)/s}$
  - x = input vector $\quad k$ = vector kernel of size l $\quad s$ = stride (step size)
- — 1 layer: (10 input)

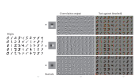


As matrix multipl.:

$$\begin{pmatrix} +1 & -1 & +1 & 0 & 0 & 0 & 0 \\ 0 & 0 & +1 & -1 & +1 & 0 & 0 \\ 0 & 0 & 0 & 0 & +1 & -1 & +1 \end{pmatrix} \begin{pmatrix} 5 \\ 6 \\ 6 \\ 2 \\ 5 \\ 6 \\ 5 \end{pmatrix} = \begin{pmatrix} 5 \\ 9 \\ 4 \end{pmatrix}$$

kernel in each row

- — multiple layers: increasing receptive field (input affecting) of neurons
  - → For layer 1: kernel size l, layer $m > 1$: $m \cdot (l-1) + 1$ for $s = 1$, else $\sim O(lm^s)$
  - → add padding so that hidden layers have same size at input
- — Example:



kernel with 3 values: light (1), gray (1), dark (-1)
→ colored pixels below treshhold: red, above treshhold: green

- Pooling layers = summarize adjacent units from preceding layer
  - → reduce size of matrix to help avoid overfitting, no activation function g
  - — average pooling: coarsening / downsampling image by factor l if $l = s$
  - — max pooling: feature distinction e.g. $l=2$ $s=1$:

5	9	4
6	2	3
7	1	1

⇒

9	9
7	3

max

5,5	4,5
4	1,8

avg

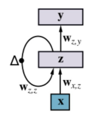
don't think we need this ↴

- Weird example: 256×256 RGB image, minibatch size 64
  → input is 4-dimensional tensor of size 256×256×3×64
  Apply 96 kernels of size 5×5×3 with stride $S=2$ in x-&y-direction:
  → output tensor = feature map of 96 channels: 128 × 128 × 96 × 64

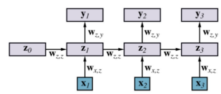
- FFN can only handle fixed-length input sequences. What if there's more? ↴

- Recurrent NNS (RNNS) & Natural Language Processing (NLP)
  RNNS = directed cycles / loops in networks: output of some state = input of other
  → 'Delay' of input (no instant self loop) ≈ internal state = short term memory
  update process for those 'hidden states': $z_t = f_w(z_{t-1}, x_i)$
- markov assumption: In hidden state z_t all info about previous states

- Training of RNNS: Input layer x, hidden layer z, output layer y
  $z_t = f_w(z_{t-1}, x_t) = g_z(w_{z,z} \cdot z_{t-1} + w_{xz} \cdot x_t) \equiv g_z(in_{z,t})$

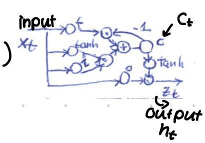


  ⇒ calculated hypothesis output: $\hat{y}_t = g_y(w_{zy} \cdot z_t) \equiv g_y(in_{y,t})^*$
  w_{xz}, w_{zz}, w_{yz} are weighted matrices shared across time points
  ⇒ this can be unrolled to Feed-Forward-NNS:



  ⇒ calculate gradients as before:
  gradient recursive in time: $\partial z_t / \partial w_{zz}$ from $\partial z_{t-1} / \partial w_{zz}$ (backpropagate through time)
  → gradients at final time T might suffer from vanishing ($w_{zz} < 1$) = short term memory ↴
  or exploding ($w_{zz} > 1$) gradients

- Long Short-Term Memory (LSTM) → Longer memory than RNN
  - values stored in memory cells C
  Per iteration: Input vector x_t, last hidden state h_{t-1}, last memory state C_{t-1}
  - forget gate g: carry over or forget C_{t-1} (reset to zero)
  - input gate i: update (add new info from vector to C_t)
  - output gate o: transfer to new hidden state h_t



  → store context, handle time lags of unknown duration
  → well suited for NLP, time-series forecasting
  → more training effort than FFN, less parallelization

- Natural language processing (NLP)

→ Issue: context is sequential data ⇒ use RNNs

- Input representation of word sequences:
  (one hot encoding: no semantic connections, n-grams: too big)
  word-embeddings: words/tokens represented by autom. learned vectors
  → similar words are mapped close in vector space => analogies

- Using FFNs: e.g. for Part-of-Speech-(POS)-Tagging:
  = analyzing words = predict 'tag' (noun, verb, ...) of word
  only in a window size e.g. in context of prev. & next 2 words
  1. words embedded in v×d matrix (v words, d latent variables)
     → each word a d-size vector ↳ learned features of word
  2. position of word important: embeddings multiplied by different parts of first hidden layer
  3. Softmax output layer: tag-probabilities
  4. Weights are learned by gradient descent

- Using RNNs: since context important; FFNs not sufficient
  1. word s_i embedded as vector x_i
  2. hidden layer z_t passed on as input to next step z_{t+1}
     → can use context info of a bounded number $z_{t'}$, $t'<t$ (still limited context)
  3. output y_i is softmax distribution over possible next word s_i

  Training RNNs for NLP: compute difference between observed output & actual data and backpropagate in time

- LSTM for longer-term input memory

- RNNs for text-generation:
  1. give input x_t, produce output y_t = softmax probability for next word
  2. Take 1 word from y_t as output for t and use it as x_{t+1}
  3. Repeat step 2 choosing randomly from y_t to generate varying outputs

- Classification with RNNs: needs labelled data & look-ahead $t+1$ (not only look back $t-1$)
  → bidirectional RNNs: concatenate right-to-left & left-to-right model
     ⇒ hidden z_t is a concatenation of vectors of both models

→ use e.g. for POS-tagging, document-classification (sentiment analysis) pos./neg.

⇒ since for every word a hidden state z_t (context) is generated, these need to be aggregated to 1 single output

- use last hidden state ⇝ biased
- use average pooling of input z_ts before FF-layer

- Sequence-to-sequence-models: not sentence → tag/word but → sentence
  Process whole sentence first:
  1 RNN for source sentence S, 1 RNN for target sentence T
  1. Run RNN_S, make its final hidden stage (= context, relations, meaning) the first hidden state of RNN_T
  2. Run RNN_T as text-generation RNN, feed output t as input for $t+1$
     ↳ choice of O_t learned e.g. via greedy / beam search

⇒ issue: nearby context bias, fixed context limit (dim. of z_t), slow (sequ. ≠ parall.)

↓ = concatenation = increasing num. of weights

- Attention = using all hidden vectors from RNNs & per word s_i pay attention only to for s_i relevant parts
  ⇒ make a 'context-based summarization' of sentence S into vector c_t, feed c_t concatenated with RNN_S output for x_t to RNN_T
  ⇒ c_t has attention scores $a_{t,t'}$ between target state t (output pos) and t'th word (input pos)

- Attention component:
  - weights not directly 'learned' but calculated by function
  - attention is entirely learned automatically (latent)

- Transformer: uses self-attention: each word attends to each other word (including itself)
  ↳ asymmetric, can be calculated simultaneously
  ↳ naive realization as dot product biased to self-attention

⇒ Project input into 3 representations:

q_t = query vector: attended from (~target)

k_t = key vector: attended to (~source)

v_t = value vector: generated context (~k)

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

$\sqrt{d_k}$ improves numerical stability

d = dimension, usually 512

$d_k = d_q = d_v$

1.

X				W_q		W_K		W_V	
Ich	0,9	0	0,2	0,1	0,7	1,3	0,6	1,3	0,6
bin	0,9	0,1	0,3	0,3	3,5	4,2	2,1	4,2	2,1
eine	0,2	0,3	0,6	2,2	0,4	1,1	0,4	1,1	0,4
Studentin	0,2	0,4	0,9						

Input — learned weight-matrices, usually $W_K = W_V$, here d=2

Q		K		V	
0,53	0,71	1,35	0,62	1,35	0,62
0,78	1,10	1,88	0,87	1,88	0,87
1,43	1,43	2,17	0,99	2,17	0,99
2,12	1,90	2,92	1,32	2,92	1,32

$Q = X \cdot W_q$

$K = X \cdot W_K$

$V = X \cdot W_V$

↳ softmax:

	Ich	bin	eine	Studentin
Ich	0,15	0,21	0,25	0,39
bin	0,11	0,18	0,24	0,46
eine	0,06	0,13	0,20	0,60
Studentin	0,03	0,08	0,15	0,73

attention-percentages

↳ human-readable

multiplied with V:

Ich	2,28	1,04
bin	2,37	1,08
eine	2,54	1,15
Studentin	2,68	1,21

contextualized Attention(Q, K, V)

- Transformer-Architecture:
  uses multi-head attention & positional encoding
  based solely on attention mechanisms, no recurrence / convolution
- Transf. = Encoder + Decoder — GPT = Generative Pre-Trained Transf.
  but there are E-only (BERT = understanding) & D-only (ChatGPT = generating) models
- Transformer layer: ~ 6+ layers in practice
  1. Self-attention: for every word generate attention, using hidden layers
  2. Simple Feed-Forward-Layer: for each word-vector seperately (same weights, ReLu)
  3. Residual connections: add inputs of each layer to output to avoid vanishing gradients
+ Position embedding: Model learns position vector for each word
  because self-att global → give first layer word emb + pos emb.
- Transformer for Translation: Enc. + Decoder-(only left-to-right + 2nd module attending to encoder output)

- Transformer & Generative AI (GAI): wide range
  - Large Language Models (LLMs): trillion params, passed turing test
  - Vision Transformers (ViT) e.g. BERT: global view on images via patches
  → Limitations: data amount, comp. resources, biases, ...

• Unsupervised learning:
  supervised = high test accuracy, but lots of labeled data needed
  unsupervised = only unlabeled data
  → learn new representation (features) or generative model
    e.g. a probability distribution $P_w(x, z)$ with latent variables z ← some feature in image e.g. glasses
    ↳ change z to generate new samples   ↳ integrating z gives $P_w(x)$
- Generative Adversarial Networks (GANs) → implicit model — generates samples but no readable probabilities
  1. generator maps values from x to z to produce samples
  2. discriminator classifies whether real / generated
  → Application: improve robustness of NNs, deepfakes

- Reinforcement Learning (RL)
  - Traditional: maximize reward (model-based: utility, model-free: policies, ...)   (= learn outcomes; = learn how to act)
  - Deep RL: DNNs as function approximators
  - RLHF (RL from human feedback): Actor, Reward, Critic, reference, PPO-algorithm

- DL ⊕ easy development, capabilities on unstructured data, parallelism
  ⊖ choice of param, data needed, implicit knowledge, difficult to predict
  challenges: combine symbolic (↗ Logic) & unsymbolic approaches (→ AI)

CONSTRAINT SATISFACTION PROBLEMS

↳ states = black-box

Standard search problems: problem specific routines: succ. f., heuristic f, goal test

CSP: general purpose algorithms using standard structured / simple representation

→ take advantage of state structure

- a state = defined by variables with values from an associated domain
- goal test = set of constraints of allowable combinations of values for variables

⇒ a simple formal representation language

- CSP Definition:
  - finite set $\mathcal{V}$ of variables, each with associated non-empty domain
  - finite set C of constraints (or for $C(v_i)$ just values)
    - → a constraint between variables v_i, v_j is a subset of tuples $D_i \times D_j$
    - → limits the values a variable can take, unary, binary, ..., n-ary
  - a state of a CSP is an assignment of values to some/all variables
  - ⇒ An assignment that does not violate any constraints is consistent / legal
  - ⇒ An assignment is complete iff it assigns every variable

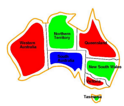
⇒ A solution to a CSP is a complete and consistent assignment

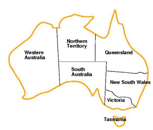
- A constrained optimization problem also maximises an objective function

- Example: map colouring as CSP:

Consider the task of colouring a map of Australia with the colours red, green, and blue such that no neighbouring region has the same colour.

There are many possible solutions, e.g.,
$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green\}$

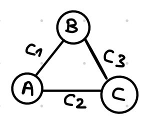




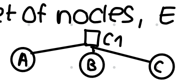
We can formulate this problem as the following CSP:
- *Variables:* $\mathcal{V} = \{WA, NT, Q, NSW, V, SA, T\}$
- *Domains:* $D_i = \{red, green, blue\}$, $i \in \mathcal{V}$
- *Constraints:* adjacent regions must have different colors
  - e.g., the allowable combinations of WA and NT are
    $$C(WA, NT) = \{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\},$$
  - or simply written as $WA \neq NT$ (if the language allows this).

- Constraint graphs:

algorithms can use graph structure



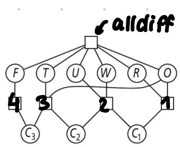
  - For binary constraints: nodes = vars, edges = constraints
  - For higher-order constraints: pair (X, E) X = set of nodes, E = hyperedges



- Cryptoarithmetic puzzles = example of higher-order-constraints

Example:

$$\begin{array}{r} T\ W\ O \\ +\ T\ W\ O \\ \hline F\ O\ U\ R \end{array}$$



- This is formulated as the following CSP:
  - Variables: $F, T, U, W, R, O, C_1, C_2, C_3$
  - Domains: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  - Constraints:
    - $Alldiff(F, T, U, W, R, O)$;
    - addition constraints:
      1 $O + O = R + 10 \cdot C_1$,
      2 $C_1 + W + W = U + 10 \cdot C_2$,
      3 $C_2 + T + T = O + 10 \cdot C_3$,
      4 $C_3 = F$.
- A solution for this CSP is, e.g., $938 + 938 = 1876$.

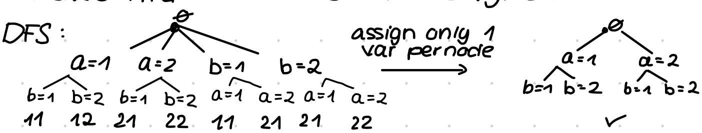
- Types of CSP: what kind of domains & constraints
  - n discrete variables with finite domains size d → $O(d^n)$ possible assignments
  - Boolean CSP: e.g. 3SAT → np-complete & exp!, but in practice often faster
  - discrete variables with infinite domains (e.g. $\mathbb{Z}$)
    → need constraint language instead of just enumerating tuples
    ⇒ there are solution alg. for linear constraints; non-linear constr. undecidable
  - continuous domains: real world problems → linear c. solvable in polyn. time

- CSP as standard search problem: uses incremental formulation of CSP
  - Initial state: empty assignment ∅
  - States: values assigned so far
  - successor function: assign unassigned var with non-conflicting value (= consistent)
  - goal-test: is assignment complete?

⇒ Since this is the same for all CSPs, standard search algorithms can be used
→ need only consider 1 variable at a time, since var ass. is commutative
e.g. Depth-First-Search generates $n!d^n$ leaves for d^n assignements

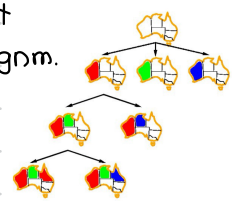
$V = \{a, b\}$ $D = \{1, 2\}$: DFS:

because every var assign. yields a new state
→ per node try all successor states



→ same end states more than once! = unnessesary

- Backtracking = DFS for CSP with single-variable assignement
  ⇒ assign 1 var at a time & if path fails jump back to last assignm.
  use general purpose algorithms to improve performance:
  → what var to assign next? which value? implications on other var?



- minimum-remaining-values-heuristic (MRV): (≠ first assignm.)
  choose variable with fewest legal values; if for some x = 0 report failure
- Degree-heuristic: choose variable with largest num of constraints to <u>unass</u>. var
- Least-constraining-value-heuristic: for choosing value, not variable
  choose value that rules out fewest choices for neighbouring variables
- Forward-checking: mightier → considers constraints before a var is chosen
  reduce search space: when X is assigned remove every inconsistent value from by constraint connected Y → if D_Y empty report failure (does not detect all failures)
- Arc-consistency: even mightier → uses constrain propagation:
  Arc X→Y in constraint-graph is consistent iff for every value $x \in D_X$ there is some allowed value $y \in D_Y$ of Y. Else delete value. Same for Y→X

Constraint propagation = Propagating implications of constraints between vars

- Further techniques:
  - Intelligent backtracking: when failure jump back to set of vars that caused failure (conflict set), to most recent var in that set
  - Local search algorithms very effective for CSP
  - Structures of graph e.g. subgraphs can be taken into account

KNOWLEDGE REPRESENTATION

Knowledge and reasoning crucial for dealing with partially observable env
→ knowl. based agent can combine general knowledge with current percepts to infer hidden aspects of the current state prior to selecting actions
⇒ represent implicit knowl. in suitable datastr. / algorithms for computation

- Logic = formal structures & rules
- Ontology = defines kinds of objects
- Computer Science

Agents often combine: (now also subsymbolic ≈ neurosymbolic approaches)
- Declarative approaches: express knowledge explicit, seperated from processing
  → flexibility, changes incorporated easily (modularity)
- Procedural approaches: knowl. manifested implicit in execution of operations
  → minimizing role of expl. repr., more efficient systems

- Knowledge based agents
  - Components: - knowledge base = set of sentences in formal language
    - methods to add new sentences & query: TELL & ASK
  - Agent program:

```
function KB-AGENT(percept) returns an action
    static: KB, a knowledge base
            t, a counter, initially 0, indicating time

    TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
    action ← ASK(KB, MAKE-ACTION-QUERY(t))
    TELL(KB, MAKE-ACTION-SENTENCE(action, t))
    t ← t + 1
    return action
```

1. TELL knowledge base what it perceives
2. ASK kb what action to perform
3. Record choice with TELL & perform action

→ details of interference mechanisms inside TELL & ASK

LOGIC = formal languages, represent info & draw conclusions
- Syntax define sentences, semantics meaning
- Entailment means one thing follows from another: $KB \models \alpha$ iff α true whenever KB true

  KB: premiss, α: conclusion, KBs are sets of sentences = „theories" $m(KB) \subseteq m(\alpha)$

  Entailment: semantics → models, inference: syntax → derivations

- models: Interpretation I where $I(\alpha) = true$, $M(\alpha)$ = set of all models of α
- equivalence $\alpha \equiv \beta$ iff $\alpha \models \beta$ and $\beta \models \alpha$
- valid = sentence always true, satisfiable = at least 1 model, unsat = no model
  → α is valid if $\neg\alpha$ is unsat, $KB \models \alpha$ if $KB \cup \{\neg\alpha\}$ is unsat
- inference (syntactical relation): $KB \vdash \alpha$ iff there exists a proof system
  = axioms + inference rules → derivation from KB / of α over elements of KB
- soundness: $KB \vdash \alpha \Rightarrow KB \models \alpha$, completeness: $KB \models \alpha \Rightarrow KB \vdash \alpha$

- Propositional logic: connectives $\neg \vee \wedge \rightarrow \Leftrightarrow$
  ⇒ truth tables, logical equivalences

$(\alpha \wedge \beta) \equiv (\beta \wedge \alpha)$	commutativity of $\wedge$
$(\alpha \vee \beta) \equiv (\beta \vee \alpha)$	commutativity of $\vee$
$((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma))$	associativity of $\wedge$
$((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma))$	associativity of $\vee$
$\neg(\neg\alpha) \equiv \alpha$	double-negation elimination
$(\alpha \Rightarrow \beta) \equiv (\neg\beta \Rightarrow \neg\alpha)$	contraposition
$(\alpha \Rightarrow \beta) \equiv (\neg\alpha \vee \beta)$	implication elimination
$(\alpha \Leftrightarrow \beta) \equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha))$	biconditional elimination
$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta)$	De Morgan
$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$	De Morgan
$(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$	distributivity of $\wedge$ over $\vee$
$(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$	distributivity of $\vee$ over $\wedge$

- First order logic
  constants, predicates, functions, variables,
  connectives, equality, quantifiers $\exists, \forall$
  atomic sentences = $p(\cdots)$ = truth value
  terms = function / constant / var = object
  → sentences are true w.r.t. a domain & an interpretation: $\mathcal{M} = (D, e)$
  domain $D^{\mathcal{M}}$ = objects, Interpr: constants $c^{\mathcal{M}}$, predicates $p^{\mathcal{M}} = \{\cdots\}$, funct. $f^{\mathcal{M}} = (\cdots)$
- „all S are P": $\forall x (S(x) \Rightarrow P(x))$ „some S are P": $\exists x (S(x) \wedge P(x))$

- Theorem proving: without models
  - Inference rules: $\frac{\alpha \Rightarrow \beta \quad \alpha}{\beta}$ $\frac{\alpha \wedge \beta}{\alpha}$ $\frac{\alpha \wedge \beta}{\beta}$ $\frac{a \rightarrow b}{\neg b \rightarrow \neg a}$ Demorgan $\cdots$
  - monotonicity: $KB \vdash \alpha$ and $KB \subseteq KB'$ then $KB' \vdash \alpha$ (additional knowledge can't invalidate inferences)
  - If $KB \cup \{\alpha\} \vdash \beta$ and $KB \cup \{\alpha\} \vdash \neg\beta$ then $KB \vdash \neg\alpha$
  - Resolution: on formulas in CNF = conj. of clauses (disj. of literals)
    $\frac{\cdots a \cdots \quad \cdots \neg a \cdots}{\cdots}$ $\frac{P \vee Q \quad \neg Q}{P}$ to show $KB \vdash \alpha$
    show $KB \cup \{\neg\alpha\}$ unsat
    → unsat if we derive empty clause $\square$
  - Conversion to CNF: 1. Eliminate $a \Leftrightarrow b$ by $(a \Rightarrow b) \wedge (b \Rightarrow a)$
    2. Eliminate $a \Rightarrow b$ by $\neg a \vee b$
    3. move $\neg$ inwards
    4. Distribute $\wedge \vee$ to CNF (Demorgans Laws)

- Aspects of knowledge representation:

- Ontological engineering: how to represent facts about the world => e.g. FOL
  → create representations of actions / time / physical objects ...
- General concept: upper ontology: graphs with general concept on top

- Organization of objects into categories = classifications
  → via predicates „Ball(a)" or functions: inheritance → taxonomy hierarchy
- categories disjoint if they have no members in common
- exhaustive decomposition: all subcategories together constitute categorys
- partition = disjoint exhaustive decomposition
- Physical composition: objects part of other objects partOf(x,y)
  composite objects define part but no particular structure bunchOf(...)
- Substances & Objects: categories vs. individual objects
  things = count nouns, stuff = massnoun
  → intrinsic properties belong to substance = stay same under subdivision
  extrinsic properties to objects (weight, length, ...)
  ⇒ physical objects belong to both categories = coextensive

PLANNING

= coming up with sequence of actions that achieve some goal
=> reasonig about results of actions either via FOL: $t \rightarrow t+1$
or using states: state $\xrightarrow{\text{action}}$ result state

- Problems with states:
  - frame problem: how to represent things that stay unchanged
  - ramification problem: representation of implicit effects
  - qualification problem: required preconditions ("qualifications") ensuring that an action succeeds

- Search vs. planning:
  Applying standard search algorithms for large real-world planning problems yield to enourmos search spaces due to irrelevant actions
  + finding good heuristic function difficult
  + can't take advantage of problem decomposition (subproblems)

- Planning environments:
  fully observable, deterministic, finite, static, discrete
  → expressive enough for good description, restrictive enough for efficient algor.
  ⇒ Standard Syntax: Planning Domain Definition Language = PDDL
  ⇒ Basis of most languages within PDDL: STRIPS

- STRIPS:
  - States: Decompose world into logical conditions = conj. of pos. literals
    → Instanciated state must be variable-free (ground) & function-free (finite domain)
    e.g. At(me, Lake) ✓   At(x,y) ✗   president (USA) ✗
  → closed world assumption: everything not mentioned is assumed false
  - Goals: state with conjunction of positive literals (= partially specified)
  - Actions: Precondition + effect; action schemata with variables = parameters
    ⇒ concrete action instanciates variables with constants
    Action Schemata: 1. name & parameter list e.g. Fly(p, from, to)
    2. precondition = conj. of function-free pos. literals — what must be true before
    3. effect = conj. of f-free pos. & neg literals — how state changes
  → Semantics: action is applicable if state satisfies preconds, else no effect
    result state s': add pos. literals from effect, delete literals where effect ¬p
    → every literal not mentioned in effect stays unchanged ⇒ <u>frame problem</u>
  - solution = action sequence when executed in initial state results in a state that satisfies goal
    bzw. ≈ partially ordered sets (≠ sequence) that respect order (s.b.)

- Action description language: E.g. Action(Fly(p: plane, Airport: to) ...

STRIPS	ADL
Only positive literals in states: *Rich* ∧ *InJail*	Positive and negative literals in states: ¬*Poor* ∧ ¬*Free*
Closed-World Assumption: Unmentioned literals are false	Open-World Assumption Unmentioned literals are unknown
Effect $P \wedge \neg Q$ means add P and delete Q	Effect $P \wedge \neg Q$ means add P and $\neg Q$ and delete $\neg P$ and Q
Only ground atoms in goals: *Rich* ∧ *InJail*	Quantified variables in goals: $\exists x\ (At(P_1, x) \wedge At(P_2, x))$ is the goal of having P_1 and P_2 in the same place
Goals are conjunctions: *Rich* ∧ *Famous*	Goals allow conjunction and disjunction: ¬*Poor* ∧ (*Famous* ∨ *Smart*)
Effects are conjunctions	Conditional effects are allowed: *when* $P : E$ means E is an effect only if P is satisfied
No support for equality	Equality is built in
No support for types	Variables can have types, as in $(p : Plane)$

has = and ≠
has typing

⇒ both: ramifications not naturally represented: implicit effects as explicit effects
no addressing of qualification problem (only finite prec, not every possibility)

- STRIPS example:

$Init(At(C_1, SFO) \land At(C_2, JFK) \land At(P_1, SFO) \land At(P_2, JFK) \land Cargo(C_1) \land Cargo(C_2) \land Plane(P_1) \land Plane(P_2) \land Airport(SFO) \land Airport(JFK))$

$Goal(At(C_1, JFK) \land At(C_2, SFO))$

$Action(Load(c, p, a),$
 PRECOND: $At(c, a) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a),$
 EFFECT: $\neg At(c, a) \land In(c, p))$

$Action(Unload(c, p, a),$
 PRECOND: $In(c, p) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a),$
 EFFECT: $At(c, a) \land \neg In(c, p))$

$Action(Fly(p, from, to),$
 PRECOND: $At(p, from) \land Plane(p) \land Airport(from) \land Airport(to),$
 EFFECT: $\neg At(p, from) \land At(p, to))$

➤ The following plan is a solution to the problem:
$[Load(C_1, P_1, SFO), Fly(P_1, SFO, JFK), Unload(C_1, P_1, JFK), Load(C_2, P_2, JFK), Fly(P_2, JFK, SFO), Unload(C_2, P_2, SFO)].$

Init	$At(C_1, SFO) \land At(C_2, JFK) \land At(P_1, SFO) \land At(P_2, JFK) \land Cargo(C_1) \land Cargo(C_2) \land Plane(P_1) \land Plane(P_2) \land Airport(SFO) \land Airport(JFK))$
$\Downarrow$	$Action(Load(C_1, P_1, SFO),$ PRECOND: $At(C_1, SFO) \land At(P_1, SFO) \land Cargo(C_1) \land Plane(P_1) \land Airport(SFO),$ EFFECT: $\neg At(C_1, SFO) \land In(C_1, P_1))$
s_1	$In(C_1, P_1) \land At(C_2, JFK) \land At(P_1, SFO) \land At(P_2, JFK) \land Cargo(C_1) \land Cargo(C_2) \land Plane(P_1) \land Plane(P_2) \land Airport(SFO) \land Airport(JFK))$
$\Downarrow$	$Action(Fly(P_1, SFO, JFK),$ PRECOND: $At(P_1, SFO) \land Plane(P_1) \land Airport(SFO) \land Airport(JFK),$ EFFECT: $\neg At(P_1, SFO) \land At(P_1, JFK))$
s_2	$In(C_1, P_1) \land At(C_2, JFK) \land At(P_1, JFK) \land At(P_2, JFK) \land Cargo(C_1) \land Cargo(C_2) \land Plane(P_1) \land Plane(P_2) \land Airport(SFO) \land Airport(JFK))$
$\Downarrow$	$Action(Unload(C_1, P_1, JFK),$ PRECOND: $In(C_1, P_1) \land At(P_1, JFK) \land Cargo(C_1) \land Plane(P_1) \land Airport(JFK),$ EFFECT: $At(C_1, JFK) \land \neg In(C_1, P_1))$
s_3	$At(C_1, JFK) \land At(C_2, JFK) \land At(P_1, JFK) \land At(P_2, JFK) \land Cargo(C_1) \land Cargo(C_2) \land Plane(P_1) \land Plane(P_2) \land Airport(SFO) \land Airport(JFK))$

- Planning with state-space-search: how to find plans
 forward sss: initial state → goal = progression planning
 backward sss: goal → initial state = regression planning

– Progression planning: initial state → consider action until reaching goal
 - initial state = initial state of problem
 - each state = set of pos. ground literals, literals not appearing = false
 - actions are applicable if precond satisfied, successor: add pos, delete neg. literals
 - goal test checks whether state satisfies goal
 → step cost typically 1
 → absence of function symbols: state space finite
 → any complete search algorithm (e.g. A*-search) yields complete planning alg.

– Regression planning: consider only relevant actions that achieve conjunct of goal & don't undo desired literals = ‚consistent' actions
 Process: Given goal G let A be a relevant & consistent action
 → predecessor: Delete pos. effects that appear in A from G
 Add precond literals from A (unless already there)
 ⇒ any standard search alg. can be used
 Example:

➤ Consider the cargo problem with 20 pieces of cargo, having the goa

$$At(C_1, B) \land At(C_2, B) \land \ldots \land At(C_{20}, B).$$

➤ Seeking actions having, e.g., the first conjunct as effect, we find $Unload(C_1, p, B)$ as relevant.

- This action will work only if its preconditions are satisfied.
 $\Longrightarrow$ any predecessor state must include the preconditions $In(C_1, p) \land At(p, B)$.
- Moreover, the subgoal $At(C_1, B)$ should not be true in the predecessor state.
 $\Longrightarrow$ The predecessor state description is

$$In(C_1, p) \land At(p, B) \land At(C_2, B) \land \ldots \land At(C_{20}, B).$$

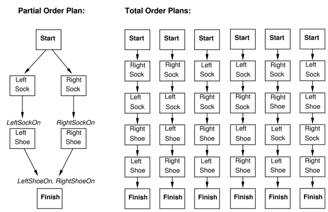
⇒ Forward & Backward SSS are totally ordered plans
 = strict sequences of actions → don't take adv. of pr. decomposition

- Partial-Order-Planning (POP) → deal with sub problems indep.
  = place actions into plan without (for all) specifying which one comes first

Example:

Init()
Goal(*RightShoeOn* ∧ *LeftShoeOn*)
Action(*RightShoe*, PRECOND : *RightSockOn*, EFFECT : *RightShoeOn*)
Action(*RightSock*, EFFECT : *RightSockOn*)
Action(*LeftShoe*, PRECOND : *LeftSockOn*, EFFECT : *LeftShoeOn*)
Action(*LeftSock*, EFFECT : *LeftSockOn*)

→ Actions can be combined independently



⇒ implement search for plan in POP-space: as instance of a search probl.
start with empty plan, consider ways of refining plan until complete
→ the actions are actions <u>on plans</u>: adding a step, imposing order, ...

- POP-algorithm components:
  - set of actions: elements for making up plan
    → empty plan: just start & finish <u>actions</u>
    start = no preconds, effect = all initial literals
    finish = no effects, precond = all goal literals
  - Ordering constraints: $A \prec B$ „A before B" = pair of actions
    → not immediately, just at some point, no cycles
  - causal links: $A \xrightarrow{p} B$ „A achieves p for B"
    p is effect of A & precond for B
    → plan may not add actions that conflict with causal link:
    if effect is ¬p and action can come after A, before B (ordering)
  - open preconditions: that are not satisfied yet

⇒ planners reduce set of open preconditions to empty set
⇒ a consistent plan has no cycles in ordering c. & no conflicts with c.l.
⇒ a solution is a consistent plan with no open preconditions

⇒ every linearisation of a partial-order solution is a total order sol
⇒ ‚executing plan' for POP = repeatedly choosing possible next actions

Example:

For instance, the final plan in the shoe-and-sock example has the following components (omitting the ordering constraints that put every other action after *Start* and before *Finish*):

Actions: {*RightSock*, *RightShoe*, *LeftSock*, *LeftShoe*, *Start*, *Finish*}

Orderings: {*RightSock* ≺ *RightShoe*, *LeftSock* ≺ *LeftShoe*}

Links: {$RightSock \xrightarrow{RightSockOn} RightShoe$, $LeftSock \xrightarrow{LeftSockOn} LeftShoe$, $RightShoe \xrightarrow{RightShoeOn} Finish$, $LeftShoe \xrightarrow{LeftShoeOn} Finish$}

Open preconditions: {}

- POP-algorithm:
  - Initial plan: Start & Finish, Start $\prec$ Finish, no causal links
    all preconds of Finish = Open preconditions
  - Successor function: pick one open precond p on any action B
    → generate succ for every consistent way of choosing action A that achieves p → add $A \xrightarrow{p} B$ and $A \prec B$ to plan (+ Start $\prec$ A & A $\prec$ Finish if action A is new)
    → resolve conflicts between new action/causal links:
    if action C conflicts with $A \xrightarrow{p} B$ add $C \prec A$ or $B \prec C$
    & add succ states for both if they result in consistent plan
  - goal-test: whether plan is solution = no open preconditions
    (planners only generate consistent plans, no need to check)

- Planning as satisfiability = translate planning problem into Prop. Formula → models of F are plans of problem

DECISION THEORY

deals with choosing among actions based on desirability of their undetermined outcomes

- Decision-theoretic agent:
  - combines utility-theory with probability-theory
  - → makes rational decisions in context of uncertainty & conflicting goals
  - → continious measure of outcome quality ⇔ goal-based: binary (goal ⇔ non-goal)
  - ⇒ preferences = utility function $U(s)$ = numbers for desirability of state

  - environments assumed episodic: not depending on previous actions
  - nondeterministic, partially observable environments
  - Result(a) = possible outcome states for action a
    $P(Result(a) = s' \mid a, e)$ Probability of outcome s' of a under observation e

- Expected utility (EU) of action a given evidence e = average utility of outcomes weighted by their probability

$$EU(a \mid e) = \sum_{s'} P(RESULT(a) = s' \mid a, e) \cdot U(s')$$

- Principle of maximum expected utility: agent will choose action = $\mathrm{argmax}_a\, EU(a \mid e)$

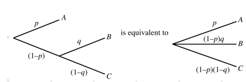
- Preferences:
  - $A \succ B$: Agent prefers A over B
  - $A \sim B$: Agent is indifferent between A and B
  - $A \succsim B$: Agent prefers A over B or is indifferent

- Lottery: Set of outcomes of an action can be seen as a lottery where the action is the ticket

  → Lottery $L = [p_1, S_1; p_2, S_2; \dots; p_n, S_n]$
  possible outcomes S_i that occur with probability p_i
  S_i is either an atomic state or another lottery

- Axioms of utility theory: conditions for reasonable preference relations

⇒ if these are violated, agent would behave irrational (e.g. loop-example)
- Orderability: Agent must decide for 2 Lotterys A & B whether $\succ, \sim, \succsim$
- Transitivity: $(A \succ B) \wedge (B \succ C) \Rightarrow (A \succ C)$
- Continuity: If lottery B is between A & C in preference, with some probability p agent will be indifferent between B for sure and a lottery with p for A and 1-p for C : $A \prec B \prec C : \exists p : [p; A, 1-p; C] \sim B$
- Substitutability: If $A \sim B$ then agent is indifferent between 2 more complex lotteries that differ in $A \Leftrightarrow B$: $A \sim B \Rightarrow [p, A; 1-p, C] \sim [p, B; 1-p, C]$
  same for $\succ$
- monotonicity: If $A \succ B$: agent prefers lottery with higher probability for A
  $A \succ B \Rightarrow (p > q \Leftrightarrow [p, A; 1-p, B] \succ [q, A; 1-q, B]$
- Decomposability: compound lotteries can be reduced to simpler ones:
  $[p, A; 1-p, [q, B; 1-q, C]] \sim [p, A; (1-p)q, B, (1-p)(1-q), C]$



- Existence of utility function:
  From preferences that satisfy axioms: $U(A) > U(B) \Leftrightarrow A \succ B$ $U(A) = U(B) \Leftrightarrow A \sim B$

- Expected utility of a lottery: sum of utilities * probabilities
  $U([p_1, S_1; \dots; p_n, S_n] = \sum_i p_i \cdot U(S_i)$

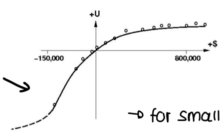
- determined by experiments & observation
  agent chooses same
  Utility function $U(S)$: determined to linear transf: $U(S) \sim a \cdot U(S) + b$
  → behaviour of agent unchanged by applying any monotonic fr. to $U(S)$ e.g. √
  ⇒ only ordinal function = ranking, actual values do not matter

- Utility scales => for measurement: no absolute scales
  → Fix utility of best prize $U(S_b) = u_\top = 1$ and worst $U(S_w) = u_\perp = 0$
  e.g. ‚1 micromort' = one in a million chance of death
  QALY = 1 year of perfect health

- Utility of money
  - monotonic preference for more money if other things equal
  - different for lotteries involving money

  - Expected monetary value (EMV) = Σ prob. * mon. outcome
  - S_n = State of possessing n Dollars, expected utilities $U(S_n)$
    → Expected utility EU = Σ prob * $U(S_k)$
    ⇒ BUT $U(S_n)$ depends on current financial status:
    Utility of money proportional to logarithm of amount

debt: reversal!



← richer: U linear over high amounts

S-shaped curve

→ for small changes in wealth: curve almost linear

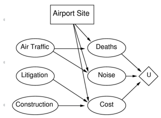
- Risks:
  $U(S_{EMV(L)})$ = being handed EMV(L) for 100% sure > U(L)
  → people are risk-averse: sure thing with payoff ≻ gamble with higher payoff
  → in large debt: risk seeking behaviour
  ⇒ agent will accept the certainty equivalent of a lottery over that lottery:
  e.g. 400$ for sure over 50% 1000$ = 400 > EMV(500) ⇒ 100$ insurance premium
  → if the agent has a linear curve it is ‚risk-neutral'

- Human judgement & irrationality: ↱ don't coincide
  Decision theory: how agents should act ⇔ Description: how humans act
- Certainty-Effect (Allais paradox): people are attracted to gain that is certain
  e.g. 100% certain 30$ ≻ 80% 40$, but not 25% 30$ ≻ 20% 40$
  → due to computational burden, mistrust in probabilities, emotions
- Abiguity aversion (Ellsberg paradox): elect unown probability rather than unknown

- Decision networks (Influence Diagrams) (extension of Bayesian networks)

→ info about agents current state, possible actions, results, utility

3 kind of nodes:



- Chance nodes (ovals): random variables with probabilities
- Decision nodes (rectangles): points with choices of actions
- Utility nodes (diamonds): utility function, parents influence outcome

- Evaluating decision networks:
  1. Set evidence values for current state
  2. For each possible action (= value of decision node):
     calculate probabilities of chance nodes that influence utility
     calculate utility for the action
  3. Choose action with highest utility

- Decision analysis: decision maker states preferences between outcomes
  decision analyst: enumerates actions + outcomes + prefs → best action

- Creating a decision network
  1. Create causal model
  2. Simplify to qualitative model
  3. Assign probabilities
  4. Assign utilities
  5. Verify system against gold-standard = correct input-output-pairs
  6. Sensitivity analysis (how sensitive decision to changes in p & u)

- The value of Information → when not all info available

→ what info to aquire? ⇒ value of observation derives from potential to affect the agents physical action

→ difference of in expected value before & after information

- Example:

A simple example:

- An oil company plans to buy one of n indistinguishable blocks of ocean-drilling rights.
- One of the blocks contains oil worth C dollars, while all other are worthless.
- The price for each block is C/n Dollars.
- If the company is *risk neutral*, then it is indifferent between buying a block and not buying one.
- Now assume that the company can buy information (results of a survey) that says definitively whether block 3 contains oil or not.
- How much should the company be willing to pay for this information?

To answer this question, we examine what the company would do if it had the information:

- With probability $1/n$, the survey will indicate oil in block 3.
  - In this case, the company will buy block 3 for C/n dollars and make a profit of $C - C/n = (n-1)C/n$ dollars.
- With probability $(n-1)/n$, the survey will show that block 3 contains no oil, hence the company will buy a different one.
  - Now, the probability of finding oil in one of the other blocks changes from $1/n$ to $1/(n-1)$, so the expected profit is $\frac{C}{(n-1)} - \frac{C}{n} = \frac{C}{n(n-1)}$ Dollars.
- Then, the resulting expected profit, given the survey information is

$$\frac{1}{n} \cdot \frac{(n-1)C}{n} + \frac{n-1}{n} \cdot \frac{C}{n(n-1)} = \frac{C}{n}.$$

- The company should be willing to pay up to C/n Dollars
  $\Longrightarrow$ the information is worth as much as the block itself!

- Value of perfect information (VPI): evidence e_j about variable E_j

best current action α: $EU(\alpha|e) = \max \sum_{s'} P(Result(a) = s' | a, e) \cdot U(s')$

after info e_j: $EU(\alpha|e, e_j) = \max \sum_{s'} P(Result(a) = s' | a, e, e_j) \cdot U(s')$

→ since E_j unknown calculate value of obtaining e_j: all possible e_{jk}:

$$VPI_e(E_j) = \sum_k P(E_j = e_j) \cdot EU(\alpha_{e_{jk}} | e, E_j = e_{jk}) - EU(\alpha | e)$$

⇒ VPI is non-negative, non-additive, order-independent $VPI(E_j, E_k) = VPI(E_k, E_j)$

PHILOSOPHICAL FOUNDATIONS OF AI

- Weak AI hypothesis: machines only <u>act</u> as if they were thinking
  - Turing Test: imitation game: machine & human → fool interrogator
    → GPT4 made 70% of people think it's human
- → Objections to intelligence of machines:
  - Argument of Disability: „A machine can't do..." ← but some things better!
  - mathematical Objection: some math. questions unanswerable by formal systems
    e.g. Gödels incompleteness → self-reference ← only for finite models
    → human understanding goes beyond proof: consciousness ≠ computation
  - Argument of Informality: human behaviour too complex to be captured by set of rules (qualification problem)
    + no biological body that perceives world (embodied cognition)
- Strong AI hypothesis: machines are thinking by simulating thinking
  ⇒ Argument of consciousness: emotions, being aware of mental state
    ↳ but we have no evidence over internal mental state of humans
- mind-Body-Problem:
  - dualist theory: mind & body 2 seperate realms: physical ⇔ consciousn.
  - monoist theory (physicalism): mental states = physical states
  - functionalism: mental state = intermediate repr. betw. in- & output
    → 2 systems with isomorphic causal processes would have same mental state
  - biological naturalism: mental states = higher level features caused by low level processes in neurons = properties of neurons

- Ethics & risks: AI might take over world (when acting irrationally)
  ⇒ watch 2001: A space odyssey