intelligent systems

Famke Nouwens

Lecture 1+2 – Intelligent agents

There is no commonly accepted definition of intelligence, but there are considered different forms, i.e. social, emotional, senso-motoric etc.

Indirect goal of AI: computational precision of the everyday notion of intelligence.

The Turing test was designed to determine the intelligence of AI: "a computer program is intelligent *if* it answers questions so that its responses are indistinguishable from a humans.

Motivations behind and requirements on AI:

- Visionary: build artefacts that produce intelligent behaviour in manner of humans
- Pragmatic: build artefacts that show behaviour comparable to human intelligent behaviour

There are also considered 2 types of AI:

- Weak: acts as if intelligent
- Strong: actually thinks

Types of AI:

- Knowledge based AI: since 1956. Its guiding model is an individual human and its guiding assumptions are that intelligence is knowledge representation and processing. Hypothesis: symbol system – the ability to produce and manipulate symbols is a necessary and sufficient condition for intelligence. This is a top-down design of intelligence: start with high-level concepts at the knowledge level and break them down into smaller, programmable units. The major problem is symbol grounding: how does it all relate to the real world? How do words get their meaning?
- Behaviour-based AI: since 1985. Its guiding model is an individual human and animal and its guiding assumptions are that intelligence is built upon elementary behavioural activities and that senso-motoric coupling is essential. Hypothesis: physical grounding – rooting of symbols in the real world (in which the artefact acts) is a necessary condition for intelligence where no rooting → no meaning → no intelligent behaviour. This is a bottom-up design of intelligence: creating basic elements and allowing a system to evolve to best suit the environment.
- Connectionism (neural networks): has been around 3 times. Its guiding model is the human brain, its guiding assumptions is that processing of information through very simple but many interconnected units that interact at allow signal-processing level. The key characteristics are: parallel, distributed and sub-symbolic information processing.
- Distributed AI: since 1980. Its guiding model is a group of humans (human society) and its guiding assumptions are that to act together is characteristic to intelligent beings ("no intelligence without interaction") and that interacting units operate on the knowledge level (rather than signal level as in connectionism). Its key issues are communication, coordination, cooperation, organization etc. Why would we deal with distributed intelligence?
    - Some problems can only be solved on the basis of high-level interaction among intelligent entities.
    - Parallelism, scalability, robustness
    - Close relationship among intelligence and interaction
    - Intuitively clear approach to complex applications

Agents are "systems at level 1". There is no commonly accepted definition of agent, since they are applied differently by different people and the definitions are often based on intuitive understanding. Emerging standard view: an agent is a computational entity that is situated (part of the environment) and that is capable of flexible, autonomous activity – action and interaction – in order to meet its design objectives. Some characteristics of agency that are sometimes claimed to be essential are: rationality, mobility, adaptivity and introspections. However, a minimally intelligent agent is:

- Pro-active: takes the initiative to satisfy their (delegated) goals
- Reactive: perceives and responds timely to the environment

- Socially able: capable of interacting with other agents and humans, which includes cooperation and negotiation.

Difficulties with agents and intelligent systems are that separately goal-directed systems and reactive systems can be simple but combining both in a good balance can be very complex. The agent/system must react to new situations while focusing on getting things done.

To allow for qualitatively different system perspectives and to have different levels of abstraction, there is an agent vs object concept which is complementary, rather than mutually exclusive. Both encapsulate:
- Identity: who
- State: what
- Passive behaviour: how, if invoked

But agents additionally encapsulate active behaviour: when, why, with whom, whether at all. There is a gradual transition from agent to object.

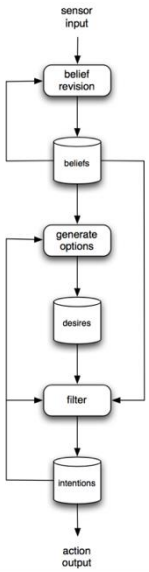
	Monolithic	Modular	OO	AO
Unit behaviour	Nonmodular	Modular	Modular	Modular
Unit state	External	External	Internal	Internal
Unit invocation	External	External	External	Internal

Expert system – interact with user to collect facts and help with a decision process.

Agent environments:
- Accessible vs inaccessible: which level of information does the agent get about its environment? Is it complete? Accurate? Up to date?
- Deterministic vs non-deterministic: is there a guaranteed effect for actions? Is there uncertainty about the next state? → sufficiently complex determinism is just as bad as non-determinism.
- Episodic vs non-episodic: independent stages of behaviour where the shorter the episode the easier. Only the current (or recent) percept is relevant, and one does not relate to the other.
- Static vs dynamic: is the influence of the agent the only cause for change?
- Discrete vs continuous: is there a fixed number of actions & perceptions?

Different agent architectures:
- Logic-based: fits with symbolic or knowledge-based AI view where intelligent behaviour is a result of a symbolic representation of the environment combined with logical deduction or theorem proving. Components:
  - Theory of agency ρ: describes how intelligent agents behave in an executable way
  - Belief database Δ: information the agent has about the environment
  - $\Delta \vdash_\rho \varphi$ means that φ can be derived from Δ using the rules of ρ. It basically has implied actions, allowed actions and no action. ("if you can prove this then do this").
  - Problems:
    - Symbol grounding: coupling perception with symbolic facts
    - Reasoning takes (too much) time
    - Very hard to build a sufficiently complete model of a complex environment
  - But logic-based is not dead as symbol grounding is starting to work (deep learning for vision), hardware is getting ridiculously fast and logical policies can be learned.
- Planning: all about using an environmental model to determine the best action to take. The theory of planning has been well-developed (success stories: MCTS).
- Reactive: behavioural, situated agents. Intelligence emerges from the interaction between simple behaviours and the environment but cannot be disembodied (result from only thinking/reasoning about things).
  - Subsumption architecture: decision making is established through a set of behaviours, where each behaviour accomplishes some task (i.e. FSM). There are no complex symbolic

representations and no symbolic reasoning: situation → action. There is a subsumption hierarchy for when multiple behaviours choose conflicting actions (e.g. autonomous cars).
    - Advantages: simplicity, computational tracktability, robustness against failure and can be quite elegant.
    - Problems:
        - Only local information is used, no model of environment
        - Short term view is inherent
        - Emergence can be hard to predict
        - Dynamics of interactions can become too complex to design
  - Reinforcement learning in Markov Decision Processes:
    - Learns a reactive policy
    - Environment is assumed to be Markovian
    - Global optimality through local decisions are made possible through a value-function (value = sum of discounted future rewards).
- Belief/desire/intention: use practical reasoning.
  1. Deliberation: what are the goals we want to achieve?
  2. Means-end reasoning: how are we going to achieve these goals?
  - Intentions:
    - Drive means-end reasoning, lead to actions
    - Constrain future deliberation, restrict reasoning
    - Persist until achieved, believed to be unachievable or purpose is gone
    - Influence beliefs for future reasoning
  - This results in a trade-off: accomplish intentions through direct action and reconsider current intentions.
    - Bold agent: never stops to reconsider
    - Cautious agent: constantly stops to reconsider
  - If the world/environment changes a lot, it's better to be cautious.
  - Components are:
    - Current beliefs (info about environment)
    - BR function: updates beliefs according to perceptions
    - GO function: generates available options/desires
    - Current options/desires (must be consistent)
    - Filter function: deliberation or intention revision process
    - Current intentions (agent's focus)
    - Action selection: translates intentions into action
  - Usually this implies representing the intentions as a stack or hierarchy again, to make action selection and prioritization possible.



- Layered: when there is a need for both pro-active and reactive behaviour. The planning for goals depends on current conditions and it must respond to changes in environment.
  - Two types:
    - Vertically layered: one or two passes. In is perceptual input, output is action.
        - Example: (2-pass) InteRRaP, which has bottom-up activation and top-down execution. It has social interactions about others, every-day behaviour about itself and reactive behaviour about the environment.
    - Horizontally layered: Number of desired behaviours = number of layers, which might need a mediator function if actions contradict. Central control might create bottleneck if complex.
        - Example: Touringmachine, which is a symbolic representation of state of all entities and constructs plans to achieve the agent's objectives. It has

communication and priority between the three layers and follows situation action rules, like subsumption architecture.

Lecture 3 – Genetic Algorithms

Complexity of agents' behaviour comes from complexity of design. Machine learning can help generate complexity.

Evolutionary computation is a way to solve an optimization problem using a population of candidate solutions and the dynamics of genetic evolution (selection, reproduction and mutation).

- It can be stopped at any time and give the current best solution
- It is stochastic so running the same algorithm twice will not generate the same solution twice

Three parts to algorithm:

- Initial population construction
- In iteration: fill mating pool with selected individuals (for reproduction) according to a selection criterion (you can have repetitive individuals).
- Generate new population from mating pool

Randomly generated set of possible solutions is the start of the algorithm but must be sufficiently diverse and large and will act as the basis for better solutions. It evolves through each generation.

Genotype – the representation of an individual. Most simple form is string of bits but can also be a tree or graph. (set of genes)

Phenotype – the thing encoded by the genotype (physical expression of genes). The performance can be tested on the phenotype.

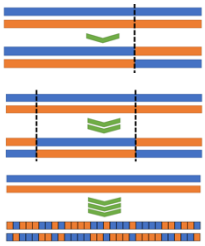
Fitness – the criterion to be optimized. Its complexity will limit the number of individuals that make up the population and limit the number of generations that can be computed. It's a way of selecting candidate individuals for both survival and reproduction (if not in the mating pool, you die off).

Fitness functions:

- Roulette wheel selection: higher change of being selected if you're fitter. Can be tuned with Boltzmann or Gibbs: $p_i = \frac{e^{f_i/T}}{\sum_{j=1}^{N} e^{f_i/T}}$ where higher T means more diverse (fit is less important).
- Tournament selection: from a selection of size *n* choose the top *x* fittest candidates and add them to the mating pool. Smaller tournaments add more diversity in the population → less likely to have very fit candidates in the tournament. Tuning happens with tournament size *k*.

Reproduction ways:

- Crossover: two parents share their genetic code and generate offspring.
  - Single point: cut at one point, take first part of parent 1, take second part of parent 2 for child 1 and the opposite for child 2
  - Double point: do the same as for single point, but then twice, generating two 'switch' points.
  - Uniform: for each gene in the genotype, flip a coin for which parent it gets the gene from.
  - If you don't have string-based format, there is a high probability of creating infeasible solutions (solution that is not diverse).

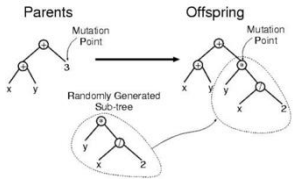


Mutation: random changes in the genotype. It helps to create more diverse population, by searching regions uncovered by the initial population.

- Bit-flipping mutation: randomly selected bit gets flipped OR the genome gets flipped completely.
- Real value mutation:
  - Randomly change values to boundary extreme values
  - Uniformly generate a random number between the boundaries

- Non-uniformly generate a number between a certain range that is deemed the best
- Gaussian noise addition: add a random number to the outcome/gene.

If you're not using strings but trees, an option is headless chicken crossover: generate random subtree without it really having a parent.



Schema theory offers insight to why good solutions survive and get combined to better solutions.

- Schema: way of defining the genes in a general form, where binary numbers indicate fixed value positions and * indicates the value does not matter.
- Order: the number of significant bits (fixed positions) in the schema
- Defining length: the longest distance between fixed positions in the schema.

Schemas can be used to reason about the disruptive properties of mutation and crossover.

- Surviving mutation: $p = (1 - p_m)^{o(s)}$, where p_m is the mutation probability of a single bit
  - Schemas with higher order have a *lower* probability of surviving mutation.
- Surviving crossover: $p = (1 - p_c \frac{d(s)}{n-1})$ where p_c is the crossover probability, $d(s)$ the defining length and n the number of bits.
  - Schemas with longer defining length have a *lower* probability of surviving crossover.

Thus most likely to survive if: low order and short defining length.

Building blocks theorem – if you want to have a well performing evolutionary algorithm, you have to identify and recombine building blocks of high fitness to build entire solutions, where the blocks take the form of schemas. Guidelines for designing your genotype:

1. Put related genes close together so that they can form schemas with low defining length (i.e. make them one gene ideally).
2. Make each gene as independent from others as possible

For designing the crossover: give the crossover operator a low probability of destroying schemas.

Crossover and mutation are meant to search the whole space and combine genes into new solutions. However, they can destroy the best solutions. Solution → elitism: copy the top *x* best solutions directly into the new generation.

Cooperative coevolution – if by design the building blocks are independent, then they can be searched independently. This is much faster and easier.

If all genes are independent in the population, we can just look at single gene and see how it evolved over the generations. And we can do this for each gene. Separable Natural Evolution Strategy (SNES): one Gaussian distribution for each gene, where the genotype sample from the population is generated by *sampling each gene separately*.

SNES: the means and standard deviations are updated according to a weighted sum of the genomes in the generated population scaled by utility (the gradient), where fit individuals have a stronger vote in the gradient. Mutation is covered by sampling each generation, its randomness ensuring diversity and preventing sub-optimal convergence.

Premature convergence (niche solution): if a fitness function is not convex but has multiple local optima in which the algorithm can get stuck in. This can happen if too many individuals become similar. Avoiding overproduction of similar individuals can be achieved with a niche penalty: adding a penalty to the fitness evaluation of any group with sufficient similarity (niche radius).

Problem with evolutionary algorithms is that they do not scale well with problem complexity. However, SNES and building block theorem assume that genes are independent → the problem can be divided into smaller subproblems. Not the case for intrinsically complex problems.

Fitness function requirements and limitations:

- Must be able to separate partial solutions from non-solutions
- Fitness value cannot be binary (solved vs not solved), but rather a percentage of success

Lecture 4 – Expert Systems

Expert Systems – computer systems that use knowledge and inference to solve problems in a specialized subject domain, that are difficult enough to require human expertise. Early expert systems were all chemistry and biology based, e.g. Dendral (heuristic program for explaining empirical data) and MYCIN (rule-based computer program for advising physicians regarding antimicrobial therapy selection).

Difference between intelligent system and expert system is that an expert system misses the real-world embedding.

Expert System	Decision Support System
Combines knowledge and/or data with reasoning	Process data with analytics
Make decisions by using user provided information	Helps users make decisions by providing the right information
Can provide additional information by explaining their decision to the user.	(Decision still lies with the human)

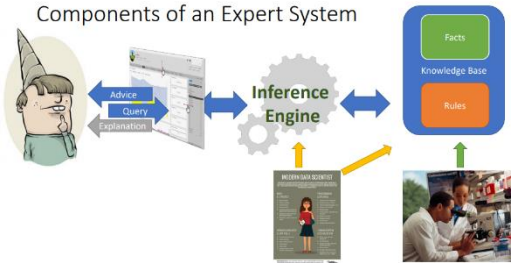
Usages of expert systems are:

- Medical or troubleshooting diagnosis
- Crime investigation
- Biological Classification
- Space Exploration

Expert systems are popular because they are proven and are provable technology: they can be checked for correctness.

Components of an expert system:

- Person that queries the interface and receives an advice and explanation from the interface
- Interface: *for example chat-boxes or a question list.*
- Inference engine
- Knowledge Base: built by experts combined with data scientists (or something related)



The KB and inference engine use rules (if-then statements) and facts (data about specific cases/instances) to inference. Different knowledge representations are description logics, ontologies or non-monotonic logics.

Probabilistic reasoning: when you have uncertainty and you want to model that uncertainty (as part of the model). *Example: facts are P(A=student) = 0.6, P(B=heads) = 0.5. Then reasoning: P(A|B) = x.*

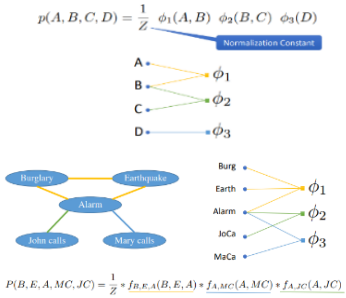
Bayes rule: $P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$. In general, there is too much data needed to estimate a target variable.

Example: assume some lab test for a disease has 98% chance of giving positive result if the disease is present, and 97% chance of giving a negative result if the disease is absent. Assume 0.8% of the population has this disease. Given a positive result, what is the probability that the disease is present?

$$P(disease|Pos) = \frac{P(POS|DIS) \cdot P(DIS)}{P(POS)} = \frac{0.98 \cdot 0.008}{(0.98 \cdot 0.0008 + 0.03 \cdot 0.992)}$$

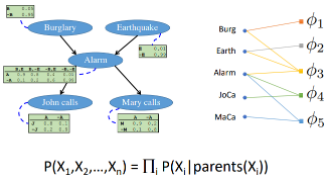
Naïve Bayes: approximation of Bayes' rule. It assumes that given the value of the class, all the attributes are independent (conditional independence). It's much more feasible than Bayes' rule, but makes extreme assumptions.

Therefore we have two extremes: either we guess the joint probability distribution (which would yield optimal classifier, but is infeasible in practice) or we use Naïve Bayes (which is much more feasible, but makes too strong assumptions). We want something in between.



Factor graphs: graphs that group random variables into conditionally independent 'cliques'. There also exists a bipartite factor graph (don't know if this is always the case).

Markov network: an undirected model that uses a non-directed graph where the cliques are fully connected subgraphs.



Bayesian network: a directed model that uses a directed graph and conditional probability tables of a class given its parents. As a factor graph:

Inference: in general, exact inference is NP-complete. But we can do for example marginalizing → determine the node that you want to have the probability for and them you sum over all values that the node could possibly have (include outside factors that influence the probability of the node).

Variable elimination: an exact inference method that determines $P_x(Y|Z)$ by getting rid of all random variables *x* in X\Y by multiplying all factors in which x appears and:

- Filling in the observed values of x if x is in Z
- Summing out over all possible values of x if x is not in Z.

Loopy Belief propagation: message passing algorithm with damped updates. 'If convergence' and marginal of X is the product of converged messages sent to X.

There are many formalisms for combining probabilities and logic:

- Statistical relational AI: StaRAI
- Distribution semantics: ProbLog
- Causal probabilities: CP-logic (making things a bit simpler).

Distribution semantics: uses probabilistic predicates and logical rules. Basically first-order logic (logic with variables) where ProbLog adds probabilities to logic rules ("probability that this predicate is true"). It assumes marginal independence between ground atoms. DeepProbLog: adding neural predicates, that produce probability distribution over classifications and that can incorporated as a special predicate.

In ProbLog, to determine if there is an edge: draw a random number 0<x<1 and check if it's below the probability of there being an edge. If yes, draw edge, if not, don't.

Problem: you can't just sum up the probabilities since they might overlap. We don't need to know the probability of the world anymore, because if we know all the possible paths, it's easy to compute. (?) We don't want to know the probability of using a certain edge, but rather if we can go from A to B. To do all this, we need to know the probabilities of an edge existing, which we often don't know.

Causal probabilities: joint probabilities can be hard to estimate by human experts, we're much better at prediction causal probabilities → CP-logic.

ML in expert systems:

- As a way to keep the rule base up to date by adapting or adding new rules to the knowledge base without expert input (Inductive Logic Programming).
- To fully automate the decision model building and forego the rule-based systems completely

Hot topic in expert systems: how to obtain the explanation of the model's answer → you need to be able to explain the recommendations it makes. This is relatively easy for the logical parts but will be harder for the statistical learning parts.

Advantages of expert systems:

- Consistency → same data gives the same decisions
- Memory → computers do not forget
- Logic → no sentimental or biased decisions

- Infinitely reproducible and does not get tired

Disadvantages of expert systems:
- Common sense is hard to program
- Creativity → difficult to find new solutions to problems
- Maintenance → knowledge base must be manually updated
- Not adaptable by itself → there is no response to changing conditions

Small summary: Non-probabilistic expert systems have historically been resolution-based (no semantics) and ontology-based and learning systems are on the rise

Lecture 5 – Multi Agent Coordination

The environment (its diversity, dynamics and predictability etc.) and cooperating entities (the number, their homogeneity, their goals etc.) influence how complex it will be to implement cooperative systems, as well as how the cooperation works (how frequent, what level, the patterns etc.). *Example: robots work in a warehouse → are there humans walking around? Are there markers on the floor? Etc.*

Coordination – managing dependencies between activities, or any decision that uses information concerning the existence/decisions/decision making strategies of others.

According to Malone & Crowston, there are 4 components of coordination:

Component of coordination	Associated coordination process
Goals	Identifying goals
Activities	Mapping goals to activities (goal decomposition)
Actors	Selecting actors and assigning activities to actors
Interdependencies	'Managing' interdependencies

Kinds of interdependencies:

Kinds of interdependence	Common object	Example of interdependence in manufacturing	Examples of coordination processes for managing interdependence
Prerequisite	Output of one activity is required by the next activity	Parts must be delivered in time to be used	Ordering activities, moving information from one activity to the next
Shared resource	Resource required by multiple activities	Two parts installed with a common tool	Allocating resources
Simultaneity	Time at which more than one activity must occur	Installing two matched parts at the same time	Synchronizing activities



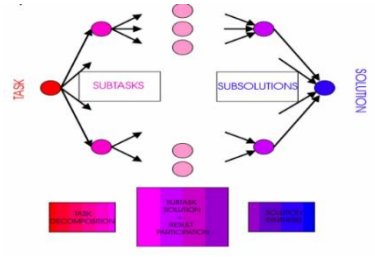
We coordinate instead of having central control because:
- Principle of Bounded Rationality (Simon):
  - The human mind's processing capacity is limited
  - The amount of information that can be processed by an individual is limited
  - The detail of control an individual may wield is limited
- Increasing complexity of computer applications
- Distributed AI perspective: to understand intelligence requires to deal with systems able to interact appropriately (?)
- Coordination is better than one central point that in case of failure would fuck the whole system up

Basic models and mechanisms for coordination:

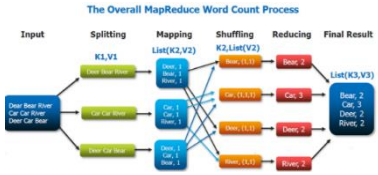
- Client-Server:
  - Client: sends requests for operations to another process
  - Server: receives requests for operations from another process

- Service: operational subject of request by some client
- Client and server can be dynamically played by processes, where a process may act both as a client and as a server, and a server may be a client of other servers.
- Pros:
  - Simple control structure
  - Simple synchronization
- Cons:
  - Server can be a bottleneck
  - Poor failure tolerance
  - Information used by servers may be out of date (communication delays)
- Can be fixed with server replication, but this requires additional coordination and synchronisation

- Task and Results Sharing: generic cooperation model.

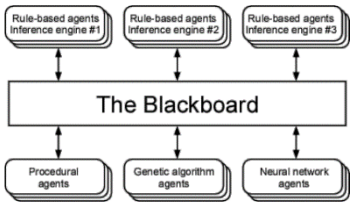


  - An example is MapReduce, that breaks an input down by splitting it, mapping part of the input in the splits, shuffling the mapped input and finally reducing it to the final result.



  - Subtask relations:
    - Disjoint: A and B not touching
    - Overlapping: A and B partly the same
    - Subsuming: B fully in A
    - Identical: A = B
  - Pros:
    - Each subproblem can be solved with less knowledge
    - Each subproblem requires less resources
    - Parallelism and robustness
    - Use of multiple sources of knowledge and skills
    - Mutual support through exchange of pre-results
  - Cons:
    - Each advantage requires design efforts on its own
    - Connection problem: who is responsible for which part of the process?
    - Lots of questions: who knows what, what are the costs, what are strategies, what is level of task decomposition?

- Blackboard System: shared system where agents have read and/or write privileges on a board where tasks can be posted. It requires common memory with read/write control.



  - Characteristics:
    - Every participant reads from and writes on a common memory area
    - Participants may read/write independently or in a coordinated way
    - Address of sender needs not be known
    - Participants themselves decide on information announcement and information search and evaluation.
  - Pros:
    - Suited for open applications
    - Supports variability in expertise
    - Flexible with respect to data structure
    - Regionalization/structuring is possible
    - Suitable for event/data driven applications
    - Supports incremental generation of solutions
  - Cons:
    - Data structure needs to be fixed

        - Can be chaotic
        - Central point
    - It is used in a topic wise approach by ROS: Robot Operating System, where nodes can push messages on a topic through a specific channel, and nodes can subscribe to the channels of interest.
- Control Net Protocol: very simple negotiation protocol. How it works:
    - A network of nodes (cooperating units acting as managers and contractors) where managers announce tasks to be done and contractors bid for the right to carry out a task. The contractor with the best bid is selected by the announcing manager.
    - A task announcement has:
        - Eligibility specification (minimal requirements for contractor)
        - Task abstraction (short description)
        - Bid specification (structure and contents)
        - Expiration date
    - Pros:
        - Flexible and distributed control
        - Dynamic roles (agent can act as manager and/or contractor)
        - Middle ground between client-server and blackboard
    - Cons:
        - Many design issues: what tasks should be announced, who should receive a specific announcement, why should a contractor bid etc.
- Functionally Accurate Cooperation (FA/C)**:** guideline for cooperation when the individuals' local knowledge is incomplete, uncertain and inconsistent.
    - Functionally correct: acceptable and reasonable from a local point of view
    - Cooperation: iterative refinement, transformation of local into global correctness
    - Requirements, involved parties have to be able to:
        - Measure and evaluate functional correctness
        - Detect inconsistencies between its tentative partial results and those received from others
        - Integrate into its local database those portions of partial results that are consistent with its own results
        - Revise and extend its tentative partial results on the basis of the newly integrated data

These were simple models of cooperation, where no advantage in task sharing comes automatically.

Advanced models and mechanisms for coordination:

- Auctioning:
    - English Auction: first-price open-cry. One variant is open-exit: no re-entry after declaring to exit the auction (you can't bid again if other people start bidding).
        - Bidder is free to raise his bid for predefined amount
        - Auction ends when no bidder is willing to raise anymore
        - Highest bidder wins at the price of his bid
    - First-price sealed-bid auction:
        - Each bidder submits one bid without knowing the others' bids
        - Highest bidder wins and pays amount of his bid
    - Vickrey auction: second-price sealed-bid auction.
        - Each bidder submits one bid without knowing the others' bids
        - Highest bidder wins but at the price of the second-highest bid
    - Dutch auction: descending.
        - Seller continuously lowers the price
        - Auction ends when one of the bidders takes the item at the current price
    - Characteristics:

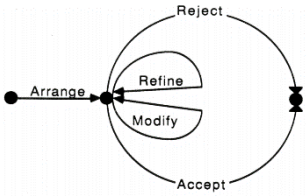
        - Valuation of goods/services:
            - Private value auction: each bidder's value of the good depend only on their own preference (i.e. art auction)
            - Common-value auction: each bidder's value entirely depends on other agent's values (i.e. fresh food auction)
            - Correlated value auction: bidder's value depends partly on own preferences and partly on others' values.
        - Risk attitude:
            - Risk-averse bidder: prefers to get the good even if they pay slightly more than private value
            - Risk-averse auctioneer: prefers to sell the good even if at a lower price
            - Risk-neutral: neither risk-averse nor prepared to take a risk
    - Auction theory: there are 3 expected outcomes.
        - Expected outcome I: English/Dutch/Vickrey/First-price-sealed-bid produce the same expected revenue to the auctioneer in private value auctions where the values are independently distributed and bidders are risk-neutral
        - Expected outcome II: among risk-averse bidders, Dutch and first-price-sealed-bid give higher expected revenue to the auctioneer, because the bidder can insure himself by bidding more than would be offered by a risk-neutral bidder.
        - Expected outcome III: Risk-averse auctioneers achieve higher expected revenue via Vickrey or English than via Dutch or first-price-sealed-bid.
        - Issues:
            - Dishonest auctioneers
            - Bidder collusion
            - Counterspeculation: getting actual information by talking to people/agents
- Voting systems:
    - Plurality Protocols: most simple; simultaneous voting happens on multiple alternatives and the highest number of votes wins.
        - Problem: irrelevant alternatives can split up the majority group (pizza toppings)
    - Binary Protocols: alternatives are voted on pairwise and the winner stays to the compared to another alternative. Surviving alternative is final winner.
        - Problems: irrelevant alternatives can change outcome and the order of pairings is crucial
    - Borda Protocol: each voter generates his own preference list over available alternatives (give a ranking). The alternative that has the highest total ranking becomes the choice. *Example: if you have 3 options, A, B & C, if A is first, A gets 3 points, if B is second-ranked, B gets 2 points etc. Sum all points = winner.*
        - Problem again is with irrelevant alternatives

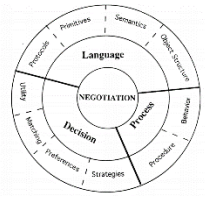
Influence of irrelevant alternatives:

Agent	Preferences
1	$a \succ b \succ c \succ d$
2	$b \succ c \succ d \succ a$
3	$c \succ d \succ a \succ b$
4	$a \succ b \succ c \succ d$
5	$b \succ c \succ d \succ a$
6	$c \succ d \succ a \succ b$
7	$a \succ b \succ c \succ d$
Borda count	c wins with 20, b has 19, a has 18, d loses with 13
Borda count with d removed	a wins with 15, b has 14, c loses with 13

Advanced models and mechanisms for coordination continued:

- Negotiation: exchange of information in multiple rounds for the purpose of coming to an agreement. Simple vs actual:





- Joint Planning: instead of individual plans make a coordinated plan.
  - General issues:
    - Key design questions: what does coordinated plan mean? How can we detect un-coordinatedness? How to generate coordinated plans?
    - General taxonomy of planning: single vs multi-component approaches
    - Relationships among plans:
      - Positive: equality (same effects), subsumption (effects of A cover effects of B) and favourableness (minor modification of A reduces efforts for carrying out B)
      - Negative: resource conflicts and incompatibility of activities and states
  - Partial Global Planning (Durfee '88): general coordination schema, with no assumptions about sub-problems, expertise or resources. Basic idea: each involved party can represent and reason about the actions and interactions of other involved parties and how they affect local activity.
    - Partial Global Plans (PGP) specify how different parts of a whole plan achieve more global states. Components:
      - Objective: why PGP exists, including its goal
      - Plan-activity map: what the parties are doing, major current plan steps (including costs and expected results)
      - Solution construction graph: information about how the parties should interact, what results should be exchanged and when to exchange them.
      - Status: bookkeeping information (pointers to relevant information received from other parties).
    - Key limitations: local actions may be executed without joint agreement and plan coordination is based on a relatively simple level of abstraction (e.g. no distinction between short and long term plans).
- Commitments and Conventions:
  - Commitments: first-class modelling abstraction.
    - Types:
      - Psychological commitment: to oneself
      - Social commitment: to others, performing actions or preventing
      - Joint commitment: with others, joint activities
      - Pre-commitment: decision to get involved in the future
      - Levelled commitment: can cancel, under penalty
    - Operations:
      - Create: initiate commitment
      - Discharge: success case, commitment was satisfied
      - Cancel: failure case, revokes the commitment
      - Release: neutral elimination of the commitment
      - Delegate: shift role of debtor to other agent (he does it now)
      - Assign: shift role of creditor to other agent (do it for him now)

- Conventions: description of circumstances under which an agent should (or is allowed to) reconsider its commitments. This because between making the commitment and carrying it out the world can change, and the agent must be able to respond to changes.
  - Challenges:
    - Decreases the reliability if commitments are abandoned too often
    - Must find balance between constantly and never reconsidering commitments
  - Social convention: description of how to behave with respect to other community members when commitments alter
- Central Hypothesis of Commitments & Conventions: Jennings '93, "All coordination mechanisms can ultimately be reduced to joint commitments and their associated social conventions." Thus coordination = commitments + conventions (+ social conventions + local reasoning).

Lecture 6 – Multi-Agent Reinforcement Learning (MARL)

Side-note: I barely understood anything from this lecture and lecture 7, so this is mostly notes I took during class or literally the slides.

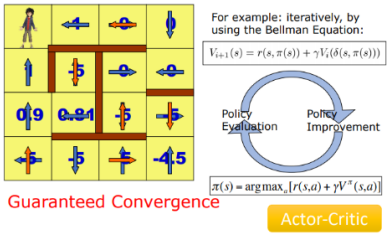
We want multi-agent systems because we don't want to have a single point of weakness that would break the system in case of failure. However, multi-agent systems are very complex, highly dynamic, and non-deterministic, where actions of one agent influence the world of others, many tasks require cooperation and there is a need for individual level of adaptation. This is difficult to achieve without learning; hence we have MARL.

Reinforcement learning: learning what to do using situation to action mapping and policies, where it learns from interactions, not examples. There are delayed rewards, no direct feedback, so that there is a need to explore and exploit.

Policy: function that translates from a state, action pair to a probability (probability of executing that action in that state) or function that translates from state to action (?). Tells the agent how to behave.

Single agent problem: given a set of states S and a set of actions A find a policy such that maximizes expected accumulated future rewards. $V^{\pi}(s_t) \equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i}$
γ is the discount factor, it's between 0 and 1. The sum V is finite. It's also a Markov Decision Process (?). This is only because the transition function has enough information required to predict and define the state.

Probabilistic policies mix up the policies.



Bellman equation: comes from the definition that we try to optimize. It tells you how you can compute the value of a state based on the previous value of a previous state under a given policy.
Example: policy iteration. Squares are states, arrows are actions. Discount factor is chosen to be 0.9. The values are calculated for the policy and based on this the policy is improved. We then calculate the values again and improve the policy again. We repeat this until guaranteed convergences. Actor critic: focus lies on the actor?

Value iteration: do policy improvement after only updating one state. This is a combination of the Bellman equation (not according to current policy) while looking for actions that can do this. It's a critic-algorithm that no longer does explicit iteration of policy. (?)

For a given state, select the action that maximizes the whole expression. δ(s,a) is s' → new state. Look for that action that maximizes the sum of rewards.

State-action values are Q-values. Q-value = immediate reward + discount factor × Q-value of the best action that you can do in that state.

Q-learning: translates delayed reward into something immediate using Q values. Algorithm for deterministic worlds:

1: Initialise Q *randomly;*
2: Initialize s;
3: **while (true) do**
4: Choose an action a;
5: Execute a and observe r and s';
6: $Q(s,a) := r + \gamma \max_{a'} Q(s',a')$;
7: $s := s'$;
8: ***end while***

In a non-deterministic environment, the training rule does not converge. We must take history into account. *Danger idea: lottery ticket never giving reward until once you win 10 euros → shouldn't buy tickets your entire life just because you won something once.*

To choose an action:

- Epsilon-greedy: roll a dice, if below exploration rate → pick a random action, else select the action with the highest Q-value. Epsilon can decay over time.
- Boltzmann Exploration: select each action a with probability: $P(a|s) = \frac{e^{Q(s,a)/T}}{\sum_{a'} e^{Q(s,a')/T}}$ (where T usually decays exponentially)

Temporal difference value: what your experience is telling you (what you expect from environment).

Simple strategies to prevent from having to visit all options (state-action pairs) infinite many times.

Multi-Agent Systems:

- Exponential state space S growth with the number of agents
- All actions jointly influence state transitions and rewards: space of possible actions and joint-action outcomes grow exponentially with number of agents, and rewards can be shared or individual.

If agents are cooperating, there are 3 types of rewards:

- Full system reward: if successful, give reward
- Local reward: direct local feedback to learn quicker
- Difference reward: each agent gets information about its impact (based on rewards with the agent's and without the agent's action)

If the agents are competing:

- Each agent has its own reward function
- Deterministic policies can be exploited: need to learn mixed policies. If an agent handles according to its best policy, it will get stuck in that policy → it becomes predictable.

Best response policy: policy that given the policy of other agents is optimal → given the strategies of my opponents, I know that this is my best strategy.
Nash equilibrium: collection of strategies such that no player can improve by changing its own policy. Each player's strategy is a best response.
Pareto optimality: strategy for which you cannot improve the situation of one, while keeping the situation of all other equal or better.

Game theory models strategic interactions as games, where normal form games model the simplest interaction.

Symmetric game – reward structure looks exactly the same, whether you are the column player or the row player, there is no difference. I.e. prisoner's dilemma (not stable).
Asymmetric is when the values differ in the cells.

Zero-sum game – add the rewards of the two players in any of the outcomes, the sum is 0. Best strategy: random.

In non-stationary environments, the action outcomes are jointly determined by all agents: all agents learn simultaneously, and an agent's action doesn't influence the state according to a fixed probability distribution.

Q-learning in normal form games: independent learner → other agents are ignored; agent behaves as during single agent learning and interactions with other agents are considered noise in a stochastic environment. However, resulting partial observability means convergence guarantees are lost.

People have tried to get learning algorithms that converge (behave predictably) and behave rationally (converge to best-response strategy). Early attempts:

- Learning Automata: policy iteration in stateless environments, originally assume a finite action set. Like Q-learning, uses immediate reward to update its action selection probabilities.

    - Each agent has a vector with number of actions dimensions that stores action preference values that are updated each time-step through a function (see slide [34])
    - α and β are reward and penalty parameters, where β can be bigger than α.
        - $\alpha = \beta \rightarrow$ linear reward penalty
        - $\beta = 0 \rightarrow$ linear reward inaction
        - $\alpha \gg \beta \rightarrow$ linear reward-ε-penalty
        - $\alpha = 1, \beta = 0 \rightarrow$ cross-learning
    - A learning automata concerns itself with state information; for each state it uses a different learning automata and uses the average reward obtained since the last visit in the update.
- Cross learning – trying to describe human learning behaviour. X for action i (instead of $\pi(a_i)$). Capital R for reward (instead of r). To make someone unlearn a bad habit, there needs to be very bad, immediate 'punishment'. Second part: always smaller than first part, because probability is [0,1]. 'doing something makes you do it again'. Doing one action lowers the probability of doing another action, no matter the reward.
- Regret minimization: stateless approach, expandable through a network of learners. Assumes hindsight knowledge of the best possible payoff, calculates a loss for taking action a_j instead of the best action*: $l_j = R^* - R_j$
    - Polynomial Weights algorithm: compute a loss for taking an action. Tries to minimize the total loss. You don't get a probability from first formula, so you must normalize to get probabilities and obtain a legal policy. (see slide [39])
- Joint Action Learning: opponent modelling (action-selection-count for all other agents) where you learn Q-values for joint actions (slide [40]). This is to have a less naïve algorithm.
    - Coordination becomes possible in cooperative MAS
    - Requires much more statistics to be kept
    - Assumes that observing the actions of all agents is possible (difficult in complex tasks)
    - Might need to know about policy update mechanisms to make correct predictions
    - Probabilities can change: model how other agents are going to change their action probabilities

Lecture 7 – Multi-Agent Reinforcement Learning (MARL)

Markov Games: game that depends on state variables to summarize the history of the game. Value iteration in Markov Games (multidimensional): every agent gets its own reward and the value of a state is equal to maximum q-value over all actions in that state. T = transition probability. O is other players' action. V(s') is value of follow-up state. (Slide [46])

Playing optimally with direct feedback uses transposed vectors x and y to represent the policies of the column and row players, where policies are probability distributions. *For example, X_2 is the probability of playing column 2*. Expected payoff is the sum of all probabilities of choosing that cell (probs of choosing row i times probs of choosing col j) times the reward of that cell. Use Q-values to estimate $y^T x$. If the col-player chooses a fixed *x*, the row-player should play *y* such that min $y^T Ax$.

Minimax Q-learning → slides [49-53].

Nash Q-learning: general sum games where coordination or competitiveness is possible. Requires maintaining its own Q-values and the Q-values of other agents. Updates are based on other agents acting rationally (choosing NE actions).

- Nash Q-values: optimal Q-values received in a Nash equilibrium. They're defined on state-joint action tuples and are the expected sum of discounted rewards when all agents follow a specified Nash equilibrium strategies from the next time step onwards.
- It's a set of policies where each policy cannot be changed individually to better the position of the agent. $NashQ^T_j(s')$: T = time, j = specific agent, s' is certain state. Encodes what is supposed to come: estimated value from state s' onward when all agents play the NE.
- Properties:

- Agents must observe *all* rewards, not just their own
- Computing a NE for each agent each step seems like a lot of work
- Convergence guarantees are limited (have strong restrictions)
- Performs quite well, but is computationally intensive

Friend-or-foe Q-learning: Littman. Stronger convergence guarantees than Nash Q-learning. Distinguishes between competitive and cooperative games, works for cooperative or zero-sum games only.

There are many more multi-agent Q-learning algorithms, but it's messy to make sense of it all via experiments because of algorithm choices, parameter choices, reward scales, who plays whom etc.

Evolutionary Game Theory: dynamic of MAL. It is based on gradients. Ways to find the consequences of policy changes:

- Cross learning:
  - We're going to idealize it.
  - Normally you pick one action and then the choice of that action influences how the probability of that action changes and how you should update the probabilities of the other actions. Ideally we would like to be able to do all of the updates for all of the actions at each step with the correct probability.
  - If we would execute action i, this is how that execution would influence that action (how it would update) and how it would update when executing other actions (at the same time). Want to act as if we try all actions at every time step. Calculates the expected change. Slides [63-68]

$$E[\Delta x_i] = \overbrace{x_i[f_i - f_i x_i]}^{\text{update to this action}} + \overbrace{\sum_{j \neq i} x_j[-f_j x_i]}^{\text{update to other actions}}$$
$$= x_i\left[f_i - \textstyle\sum_j x_j f_j\right].$$
with $f_i = E[R_i]$ the expected reward for action i

- Regret Minimization Dynamics: the gradient is basically cross learning divided by a formula that modulates the learning rate according to best action's update.
- Boltzmann Exploration Dynamics: uses bellman equation, but there is no state involved. Time is the most important parameter, since we want to study how it changes over time for a certain action. (?) Slides [71-73]

The policy determines the update frequency, and an action value is only updated when selected for execution. This pollutes the gradient → the agents have no clue where to go (you mess with the gradient that you are trying to use).

$$E\left[\dot{Q}_i\right] = x_i(t) \cdot \alpha\left(E[r_i(t)] + \gamma \max_j Q_j(t) - Q_i(t)\right)$$

To get the gradient that we want, we adjust α with a division factor. This is called frequency adjusted Q-learning and gives:

$$Q_i(t+1) \leftarrow Q_i(t) + \frac{1}{x_i(t)}\alpha\left(r_i(t) + \gamma \max_j Q_j(t) - Q_i(t)\right)$$

However, because of this, if the probability of choosing action i is very small (x_i is small), the Q-value may become very large. This can be solved by multiplying the division factor with β and taking the minimum of that and 1:

$$Q_i(t+1) \leftarrow Q_i(t) + \min\left(\frac{\beta}{x_i(t)}, 1\right)\alpha\left(r_i(t) + \gamma \max_j Q_j(t) - Q_i(t)\right)$$

To choose β use trial-and-error. If beta is 0, it does not learn. If beta is 1, it is regular Q-learning.

If we have $x_i \geq \beta$ we have theoretical FAQ learning with learning rate $\alpha\beta$. Else we have idealized Q-learning. Remark on α: people who came up with this assume α is fixed. But a general α does not consider how often Q(s, a) value has been updated so far. In SARL we use: $\alpha = \frac{1}{visits(s,\ a)}$, which makes the update action dependent and relative to the selection probability.

Gradient Ascent Learning: used for describing learning algorithms, but we know what we want it to be. Say that V translates a policy into the policy-value (what it is worth to follow policy *x*), then we want to update *x* according:

$$\Delta x_i(t) \leftarrow \alpha \frac{\partial V(x(t))}{\partial x_i(t)}$$

Policy Hill Climbing: two learning rates α and δ (if δ = 1 it's like Q-learning). Idea: in a loop, select with exploration and execute an action *a* from state *s* and observe *r* and *s'*. Calculate Q(s, a) and update π (while constraining it to a legal probability distribution).

Win or Learn Fast (WoLF): policy hill climbing approach with adjustable learning rate δ. It actually uses two learning rates for policy adjustment, a small δ_w for when it's winning and a larger δ_l for when it's losing and thus needs to learn fast. It compares the expected value of the current policy to the average policy. If the current policy is doing better, choose a small learning rate δ_w. If the current policy is doing worse, use the large learning rate δ_l.

Lecture 8 – Deep Learning for Agents

If agents want to use high level reasoning, they have to have some kind of representation of the state that they're in. Symbol grounding problem: problem of constructing or a view of the environment. What is an object? What does it mean for something to be an object?

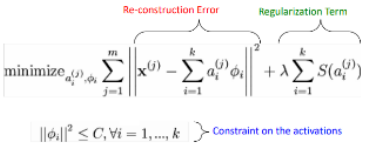
Assumption Q-learning: table-based Q-value representation, a single table entry for every possible state-action pair.

Redundant bases: a way to store information. *For example: streets represent a structure in a world. Information of longitude and latitude doesn't take the structure of the world*. We describe a picture by noting general shapes instead of noting the colours of each pixel.

Random images for us are images with random objects in it, instead of random colour for each pixel.

Good features: very expressive features that don't occur in a large percentages of the data.
Sparse feature: feature that doesn't occur very often, but if it does, it is very informative.

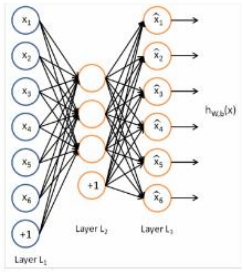
Sparse Coding: search for these features in an automatic way, modelled as an optimization problem. X is original data, meaning it is the data in its original form. Sum of *a*: how the sparse coding algorithm is going to represent the same data using the weights a_i and the features ϕ_i (bases vectors, features that are sparse but very informative). We want to try find bases function phi that allow the approximation of the original data x with weights that most of the time are 0.



How to find these? Optimization problem: first try to minimize the sum of the reconstructions and the second term is regularization defined on the weights *a* (low cost for features *a* where a lot of them are 0). This regularization term is often called L1 norm. The cost of this norm goes up with the size of *a* (linearly). We must add the constraint that phi is small so that all the a_i's are close to 0.

Bases vectors phi and their activations a_i make it easier to do certain predictions instead of starting from pixel-based representations (images).

Simplest form of deep-learning: auto-encoders. Blue is input and the entire network's goal is to reproduce that input in the final layer (i.e. reproduce the same image). Less nodes in the middle layer: forces the network to represent the same information that is high-dimensional in the input with only few dimensions in the middle. (better make sure the features represented by those fewer nodes are very informative). Then the output nodes need to learn what to be/do if it matches something in one of the fewer nodes.



This bottleneck forces the network to learn informative features. However, design question: how little nodes do we need?

Sparse-autoencoders: force a node to only be active in 5% of the cases (whatever data samples we have). This forces sparsity.
Variational autoencoder: force bottleneck in the middle by adding noise (not going to go into detail on this).

Why do we use autoencoders? To force sparsity, but we can also use them for classification, regression and/or predictions. The bottleneck is restricted to what it can and cannot represent. It cannot represent

noise. It will try to represent the picture with the noise, but will fill, and ideally represents the picture without noise:

- Imputation: prediction of missing values. (trained on pictures of people faces, what happens if there's a picture with sunglasses?)
- Inpainting: removing bars or something from a picture.

Autoencoders: can be repeated/stacked. Can learn new features that might even be better than first layer of autoencoder (higher-level features). With this, we can delete part of the network/model to build a prediction model that use higher-level features. *Example*: *using multiple autoencoders can represent the following features when having images: first pixels, then edges, then object parts and lastly objects.*

Convolutional filters: a filter is just a matrix of numbers which then gets applied to a picture over and over again. New pixel value in middle cell = First value in the filter * first value in the image + second value in the filter * second value in the image + …. Examples:

$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$ • Edge detection

$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$ • Sharpen

$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ • Blur

Face detection works by creating a filter of a face that gets applied on an image. The whole image will turn black except for the part that matches the face in the filter → bright spot there where there is a face.

Convolutional networks:

- Input layer
- Second layer are filter neurons of local parts of the input (*for example, face detection, where an image is split into 4 parts: 4 corners = 4 neurons*). Weights are repeated for every neuron, so they all detect the same thing. If one weight gets updated, then all weights for shared nodes get updated.
- Maxpooling layer: takes the maximum value of the previous nodes for which it is linked. This allows it to become a global filter (face found in the total picture).
- Softmax*:* translates the number of features into a probability distribution over a number of classes that you want to predict.

Go is easily represented in states: matrix with 0 (nothing), 1 (white) and 2 (black). Actions are basically method calls with a position: x and y coordinate. Policy: best action to play from a certain state. Deep learning is very popular now because of computational power and available data.

Simplest AI is planning & search: try all possible actions from a certain state. Build a search tree, where depth is the number of moves you look ahead. Best move to play is the move that leads to the highest number of wins (if the game would be played to the end). However, the problem is that your search tree would be way too big. So we reduce the size of the search tree: limit the width or limit the depth by learning from either human or own experience.

Two types of machine learning:

- Classification: look at search and say 'some of these moves are not played very often by experienced players' so they're probably not very good moves so we can eliminate these moves (if the probability is too low).
- Regression**:** predict the number of wins a position is expected to generate. Can be a probability or count.

A computer is kind of like a human looking at something it has never seen before: e.g. detecting structures in a full go board. A computer just sees numbers, instead of parts.

Saliency map: where would humans say is the information in the picture/data.

Learning the parameters in a NN is done by taking a random initial value for the parameters and then adjusting the weights to minimize the cost function (for an autoencoder it's going to be the inaccuracies in the reconstruction that make up the cost). Once the cost function is defined, the NN is going to use examples to calculate the cost and the gradient of the cost to find a global or local minimum → Finding the minimum of a function can be done by following the gradient: the local value of the derivative.

Now the NN has a specific structure, where each layer's activation is a function of the activations of the previous layer and the same goes for that layer. Thus the gradient is computed by the chain rule = product of derivatives. However, multiplying numbers is very unstable if they are not 1 (will make the gradient explode). If some of them or only one of these is 0, the whole gradient is 0. And this gradient is what is going to carry information across the network, so if this is 0, a lot of nodes are not going to know what to change.

Early networks used a nonlinear (logistic or tanh) functions for every node in the network, where the derivative was always smaller than ¼ (max derivative at x =0), which was a huge problem.

For nonlinear behaviour, people started using Rectified Linear Units: a function that has gradient 0 for a large part (if the node is not contributing, it should not change), and gradient 1 for if the node is contributing.

For Go, before deep learning, we didn't know how to represent the state/game. First model had accuracy 57% using supervised learning: classification.

$$\Delta\sigma \propto \frac{\partial \log p_\sigma(a|s)}{\partial \sigma}$$

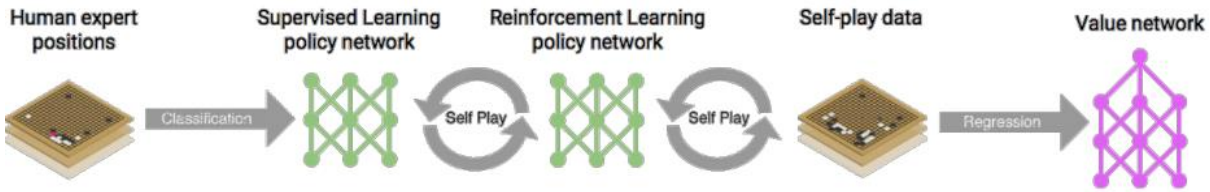
To improve the model, they let the model play itself: actions that lead to a win are made more likely, and actions that lead to a loss are made less likely. To prevent a mess, they always fixed one player → didn't change its policy (avoid messy part of MARL). 1.1 vs 1.2, 1.2 vs 1.3, 1.3 vs 1.7 etc. They did this for a week, until the last version won 80% of the games vs the first one. This was reinforcement learning policy, using a policy gradient algorithm, where policy is represented by a neural net. Z = 1 if game is won, z = -1 if game is lost.

$$\Delta\sigma \propto \frac{\partial \log p_\sigma(a|s)}{\partial \sigma} z$$

Limiting the depth: use the same architecture but add an additional regression layer to predict the win probability for that state: 0 prediction means bad position, 1 predictions means excellent position from which you'll definitely win. The gradient is driven by how much my current estimate of who's going to win differ from the observation of who actually won the game. Z is used to make the step along the gradient larger or smaller by seeing how much it deviates from the current estimate made by the network: supervised learning regression.

$$\Delta\theta \propto \frac{\partial v_\theta(s)}{\partial \theta}(z - v_\theta(s))$$

Information flow:



Monte Carlo Tree Search: a way to search the search tree intelligently. Four phases: selection, expansion, simulation, backpropagation. Selection: which moves would be most likely to be played (which part of the subtree to be investigated further). Expansion: need to evaluate the 'end' node. So we play out the rest of the game 'randomly' → simulation. We do multiple simulations to estimate how likely the game is won from that position. Backpropagation: combine search and predictions already made and adjust the likelihood of winning of all moves above the examined node.

Alpha Go Zero: no more pre-training, only self-play. No more multiple networks: 1 network for both action-probability and state-value, and no more simulations during MCTS, only state evaluation according to neural network.

- Self-play: small local search from state, at each endpoint use network to predict what the outcome is going to be (no more rollouts=random simulations), but action-selection at each node in the small search tree can use network action probabilities. But policy from which the actual action to be sampled was played, came from observations of all predicted outcomes at the bottom of the local search tree.
- MCTS: Q = prediction of the network, U = exploration term
- Instead of random simulation, they use the value of the network to predict the value of the state that you would end up in by selecting that action. And backpropagate this.

- Network learns both probabilities for action from a board position if you're playing to win, and what the win probability is for player white/black given that board position.
- Loss function uses just 3 terms: $l = (z - v)^2 - \boldsymbol{\pi}^{\mathrm{T}} \log \boldsymbol{p} + c\|\theta\|^2$
    - Mean squared error: how much is my prediction deviating from my observation
    - Reinforcement learning: log probability. Tune my policy according to what I played and how I succeeded, if I won, decrease probabilities. If you won, increase probabilities of choosing those actions.
    - Regularization term: makes sure network is forced to find good features that represent the go-board game really well. i.e. not all nodes should be active all the time.

AlphaGo Zero is very impressive (it found moves that are not played by human players, but actually perform better), but what's still missing:

- Go is deterministic, fully observable and discrete
- Perfect simulator of environment available
- Each game is relatively short
- Evaluation of outcomes is clear

So now the new objective is to build an AI for StarCraft: continuous action space, non-Markovian environment.

- Believe states: this is the state that the agent believes the world to be in. With this you kind of get rid of the fact that you need to explicitly model the whole history of how you ended up in that state.
- Recurrent neural networks: they have memory and thus circumvent the fact that immediate state is not sufficient.
- Attention model: which part of the modelled history should be paid attention to for predicting which action to execute.

Relation reinforcement learning: in different situations, objects are/act differently. In alphastar: looking at activations of nodes in the deep network over time, and if a node would be active in a certain sequence of levels of activation, that would often mean that there is some kind of power behind this so maybe some kind of behaviour of a complex object in the world.

Lecture 9 – CERN

I didn't understand much from this lecture.

Lecture 10 – Pathways To Human Level AI

What is next in the future? How far can we go in AI?

We can fake intelligence with a simulation of intelligence.

Artificial General Intelligence: something that will always fool you to think it is human →something that can talk back as a human, something that can read your mood etc.

The probability that we are living in a simulation now: $f_{sim} = \frac{f_p f_i \overline{N_i}}{(f_p f_i \overline{N_i}) + 1}$ where:

- f_p is the probability that a society can reach a level that allows them to run a simulation of a planet sized population
- f_i is percentage of populations that reach that capacity and are interested in simulation a planet-sized simulation
- N is the actual number of simulations that could be run.

Reconstruction of evolution: if it was possible to run a simulation of a planet sized population, then maybe we can also simulate evolution. Evolution also gave rise to intelligent systems.

- Arguments for:
    - Computational power is just rising and rising and we can expect it to keep doing so
    - Blind, undirected evolution gave rise to intelligence one. That's proof it could work
    - Surely then, a well-engineered evolutionary process can get there much faster?

- Arguments against:
    - Selection bias: just because we observe it happening once, we cannot generalize it to happen every time
    - Artificial intelligence might be very different than human intelligence. Natural evolution gave rise to natural intelligence, computation intelligence could be completely foreign. *E.g. how birds fly vs how planes fly.*
    - Computation power might not rise high enough → could be almost unreachable

Whole Brain Emulation: 1-to-1 modelling of the human brain in steps.

1. Take a particular brain
2. Scan its structure in detail
3. Construct a software model of it faithful to the original
4. Run on appropriate hardware where it will behave in essentially the same way as the original brain

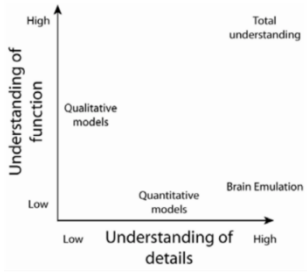
Premise behind this:

- Church-Turing thesis: "A Turing machine can emulate any other Turing machine". A machine is called Turing Complete if it can emulate a Turing Machine. Thus, all Turing Machines are Turing Complete.
- Physical Church-Turing thesis**:** "Every physically computable function can be computed by a Turing machine" (Turing machines are universal). Note that the function might be infeasible, but at the very least, if a Turing Machine had infinite time and infinite memory, it could do it.

Levels of detail of emulation:

- Brain emulation: software that models the states and functional dynamics of a brain at a relatively fine-grained level of detail
- Mind emulation: brain emulator that is detailed and correct enough to produce the phenomenological effects of a mind (e.g. self-awareness)
- Person emulation: mind emulation that emulates one particular mind

Non-organicism assumption: total understanding of the brain is not needed, just understanding the component parts and their functional interactions*. simple example: Compilers do not understand software and merely perform syntactic operations that transform human source code into machine executable code.*



Thus a whole brain emulation pipeline might (without any understanding) mechanically convert a physical system (a brain) into a software system (a simulation).

- Understanding of function: why is it doing certain things?
- Understanding of detail: how it works.

Scale separation: There exists a cut-off point beyond which we do not actually need to emulate to get the functionality we want. *Example: Modeling the movement of planets, doesn't require a weather model of Jupiter (or even further, of the Amazon rainforest)*.

The feasibility is based on the existence of a cut-off point, where you don't need to go into further levels of detail in the emulation to reach sufficient levels of accuracy. However, it is uncertain that such a point exists for the brain.

Image processing: noise removal, interpolation etc.
Scan interpretation: identification of cell types, connection tracing, estimation of parameters etc.

Computational complexity is highly correlated with how well the neural simulation model explains how a neuron works.

Seed-AI: bootstrapping through learning. Turing (1950) said: "Instead of trying to produce a programme to simulate the adult mind, why not rather try to produce one which simulates the child's? If this were then subjected to an appropriate course of education one would obtain the adult brain."

Conclusion: it’s difficult to make predictions, especially about the future.