

# Machine Learning

## Artificial Intelligence

# What is ML?

## Definition

- Arthur Samuel (1959)
  - “field of study that gives computers the ability to learn without being explicitly programmed”
- Herbert Simon (1970)
  - “Learning is any process by which a system improves performance from experience.”
- Tom Mitchell (1998)
  - “a well-posed learning problem is defined as follows: He says that a computer program is set to learn from an experience E with respect to some task T and some performance measure P if its performance on T as measured by P improves with experience E”
- Ethem Alpaydin (2010)
  - Programming computers to optimize a performance criterion using example data or past experience.

## Necessity

- Better computational systems
  - Too difficult or too expensive to be constructed manually
    - Systems that automatically adapt
      - Spam filters
    - Systems that discover information in large database → data mining
      - Financial analysis
      - Text/image analyses
- Understanding the biological systems

# How to design a ML?

- Improve of task **T**
  - Establish the goal (what has to be learned) – *objective function* – and its *representation*
  - Select a learning algorithm to perform the inference of the goal based on experience
- Respect a performance metric **P**
  - Evaluation of the algorithm's performances
- Based on experience **E**
  - Select an experience database

## Examples:

- T: playing checkers
- P: percent of winning games
- E: playing the game
- T: handwritten recognition
- P: percent of correct recognized words
- E: database of images with different words
- T: separate the spams
- P: percent of correct classified emails
- E: databases with annotated emails

# Objective Function

## What is the function that must be learned?

Ex.: checkers game → a function that:

Selects the next move

Evaluates a move

In order to identify the best next move

## Representation of objective function

Different representations:

- Table
- Symbolic rules
- Numeric functions
- Probabilistic functions

There is a trade-off between

- How expressive is meaning of the representation
- Easy of learning

Objective function computation

- Polynomial time
- Non-polynomial time

Ex. for checkers game

A linear combination of `#white_pieces`, `#black_pieces`,  
`#white_compromised_pieces`,  
`#black_compromised_pieces`

# The Learning Algorithm

## Selection

- According to the training data
- Induce the hypothesis definition that
  - Match the data
  - Generalize the unseen data
- Main principle
  - Error minimisation (cost function – loss function)

## Evaluation

### Experimental

- By comparing different methods on different data (cross-validation)
- Collect data based on performances
  - Accuracy, training time, testing time
- Statistical analyse of the differences

### Theoretic

- Mathematical analyse of algorithms and theorem proving
  - Computational complexity
  - Ability to match the training data
  - Complexity of the most relevant sample for learning

# Comparison between 2 algorithms

## Performance measures

- Parameters of a statistic series
- Proportion (percent) computed for a statistical series (ex. Accuracy)

## Comparing based on confidence intervals

- For a problem and 2 solving algorithms with performances  $p_1$  and  $p_2$
- Confidence intervals  
 $I_1 = [p_1 - \Delta_1, p_1 + \Delta_1]$  and  $I_2 = [p_2 - \Delta_2, p_2 + \Delta_2]$
- If  $I_1 \cap I_2 = \emptyset \rightarrow$  algorithm 1 works better than algorithm 2 (for the given problem)
- if  $I_1 \cap I_2 \neq \emptyset \rightarrow$  impossible to decide

Confidence interval for the mean (average)

- For a statistical series of  $n$  data, with computed mean  $m$  and dispersion  $\sigma$ , determine the confidence interval of the mean  $\mu$
- $P(-z \leq (m-\mu)/(\sigma/\sqrt{n}) \leq z) = 1 - \alpha \rightarrow \mu \in [m - z\sigma/\sqrt{n}, m + z\sigma/\sqrt{n}]$
- $P = 95\% \rightarrow z = 1.96$

Confidence interval for accuracy

- For an accuracy  $p$  computed for  $n$  data, determine the confidence interval of accuracy
- $p \in [p - z(p(1-p)/n)^{1/2}, p + z(p(1-p)/n)^{1/2}]$
- $P = 95\% \rightarrow z = 1.96$

$P=1-\alpha$	$z$
99.9%	3.3
99.0%	2.577
98.5%	2.43
97.5%	2.243
95.0%	1.96
90.0%	1.645
85.0%	1.439
75.0%	1.151

# Training database

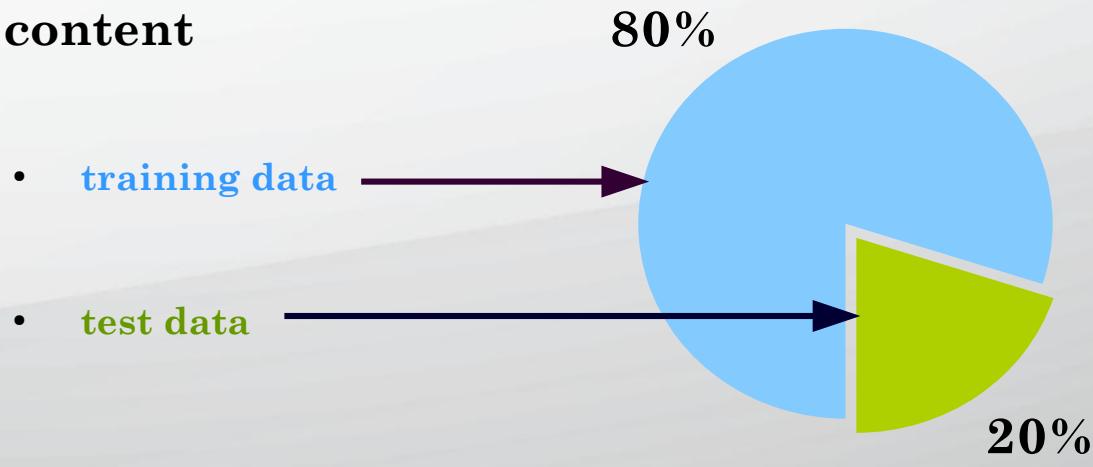
## choose the training database based on:

- direct experience
  - pairs (in, out) that are useful for the objective function
  - eg. checkers game → board game annotated by correct or incorrect move
- indirect experience
  - useful feedback (unlike i/o pairs) for the objective function
  - eg. checkers game → sequences of moves and the final score of the game

## data sources

- random generated examples
  - positive and negative examples
- positive examples collected by a learner
- real examples

## content



## characteristics

- independent data
  - otherwise → collective learning
- training and testing data must respect the same distribution law
  - otherwise → *transfer learning/inductive transfer*
    - vehicle recognition → truck recognition
    - text analyses
    - spam filters

# Training database

## characteristics extracted (attributes) from raw data

- quantitative characteristics → nominal or rational scale
  - continuous values → weight
  - discrete values → # of computers
  - range values → event times
- qualitative characteristics
  - nominal → colour
  - ordinal → sound intensity (low, medium, high)
- structured
  - trees – root is a generalisation of children (vehicle → car, bus, tractor, truck)

## data transformation

- standardisation → numerical attributes
  - remove the scale effect (different scale and units)
  - raw values are transformed in z scores
- $Z_{ij} = (x_{ij} - \mu_j)/\sigma_j$ , where  $x_{ij}$  – value of  $j^{th}$  attribute of  $i^{th}$  instance,  $\mu_j$  ( $\sigma_j$ ) is the mean (standard deviation) of  $j^{th}$  attribute for all instances
- selection of some attributes

# ML classification – goal oriented

## Intelligent systems for prediction

Aim: predict the output for a new input based on a previously learned model

Eg. predicting sales of a product for a time in the future based on price, calendar month, region, average income

## Intelligent systems for regression

Aim: estimation of the (uni or multi variable) function's shape based on a previously learned model

Eg.: estimate the function that models the edge of a surface

## Intelligent systems for classification

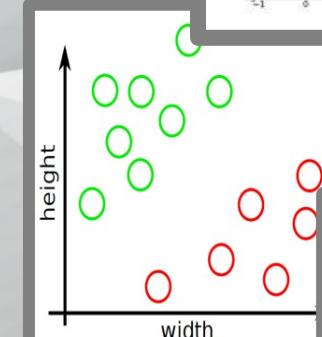
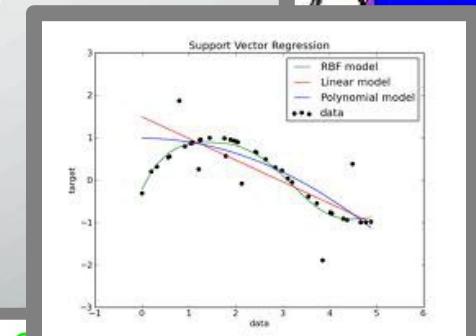
Aim: classify an object into one or more – known or unknown - categories based on their characteristics

Eg.: diagnostic systems for cancer: malign or benign or normal

## Intelligent systems for planning

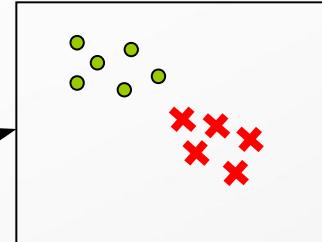
Aim: generate a sequence of optimal actions for performing a task

Eg.: planning the moves of a robot from a position to a source of energy



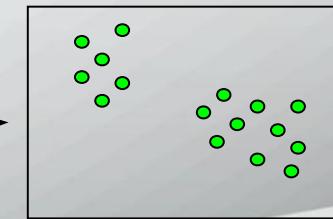
# ML classification – based on learning experience

- Intelligent systems with supervised learning



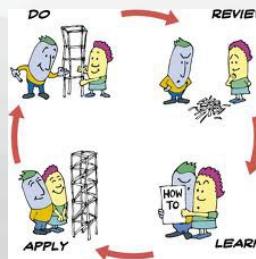
data is labeled

- Intelligent systems with unsupervised learning

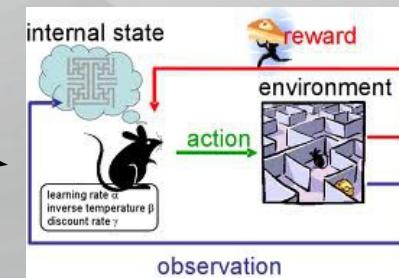


data is not labeled

- Intelligent systems with active learning



- Intelligent systems with reinforcement learning



# Supervised learning

## Aim

to provide a correct output for a new input

## Definition

**for a given data set (examples, instances, cases)**

- training data – pairs ( $attribute\_data_i, output_i$ ), where
  - $i = 1, N$  ( $N$  = number of training data pairs)
  - $attribute\_data_i = (atr_{i1}, atr_{i2}, \dots, atr_{im})$ ,  $m$  – number of attributes (characteristics, properties) for one data
  - $output_i$ 
    - a category from a predefined given set with  $k$  elements (number of classes) → classification problem
    - a real number → regression problem
- teste data – a set ( $attribute\_data_i$ ),  $i = 1, n$  ( $n$  = number of test pairs).

## determine

- a function (unknown) that maps the input attributes to the output on the training data set
- the output (class/value) associated with the test data (new) using the detected function

## Other names

classification (regression), inductive learning

## Process → 2 steps

- **Training**

learning the classification model – using an algorithm

- **Testing**

test the model with new data (not seen in the training step)

## Feature

DB is labeled (for learning and sometimes also for testing)

## Suitable problem types

- regression, classification

# Supervised learning learning quality

We consider a performance measure for the algorithm that is evaluated during both steps, training and testing, separately.

ex. accuracy (Acc = number of samples correctly classified / total number of samples)

## Evaluation Methods

for a large dataset	for a small dataset	for a very small dataset
disjunctive sets for training and testing	cross validation with $h$ equal sub sets of the dataset	leave one out cross validation

Difficulties: **over-fitting** – good performance on training data but very poor on testing data

## Performance measures:

- statistical, efficiency, robustness, scalability, interpretability, compactness, ...

# Statistical metrics

## Classification Accuracy

- the ratio of number of correct predictions to the total number of input samples

$$Acc = \frac{TP + TN}{N}$$

– works well only if there are equal number of samples belonging to each class.

## Confusion Matrix

describes the complete performance of the model.

		Reality	
		positiv class	negativ class
computed results	positiv class	True positiv (TP)	False positiv (FP)
	negativ class	False negative (FN)	True negative (TN)

## Precision

- the number of correct positive results divided by the number of positive results predicted by the classifier.

$$Prec = \frac{TP}{TP + FP}$$

## Recall

- the number of correct positive results divided by the number of all relevant samples

$$Recall = \frac{TP}{TP + FN}$$

## F1 Score

- is the Harmonic Mean between precision and recall.
- the range for F1 Score is [0, 1]

$$F1 = 2 * \frac{1}{\frac{1}{Prec} + \frac{1}{Recall}}$$

- how precise your classifier is (how many instances it classifies correctly),
- how robust it is (it does not miss a significant number of instances).

## Mean Squared Error

- is the average of the squares of the differences between the original values and the predicted values.

$$MeanSquaredError = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

- it is easier to compute the gradient

# Unsupervised learning

## Aim

to detect a model or a internal util structure of the data

## Definition

for a given data set (examples, instances, cases)

- training data – a set  $\{attribute\_data_i \mid i=1, N\}$  where
  - $N$  = number of training data pairs
  - $attribute\_data_i = (atr_{i1}, atr_{i2}, \dots, atr_{im})$ ,  $m$  – number of attributes (characteristics, properties) for one data
- teste data – a set  $(attribute\_data_j), i=1, n$  ( $n$  = number of test pairs).

## determine

- an unknown function that groups the training data in several classes
  - the number of classes  $k$  can be predefined or unknown
  - the data from one class are similar
- the output is the class the new input data (new) using the detected function

## Examples of distances

consider two points  $p$  and  $q$  from  $R^m$

- Euclidean  $d(p, q) = \sqrt{\sum_{j=1}^m (p_j - q_j)^2}$
- Manhattan  $d(p, q) = \sum_{j=1}^m |p_j - q_j|$
- Mahalanobis – measures the distance between a point  $p$  and a distribution  $Q$
- Internal product  $d(p, q) = \sum_{j=1}^m p_j q_j$
- Cosine  $d(p, q) = \sum_{j=1}^m p_j q_j / (\sqrt{\sum_{j=1}^m p_j^2} \sqrt{\sum_{j=1}^m q_j^2})$
- Hamming – number of differences between  $p$  and  $q$
- Levenshtein – the minimum number of transformations necessary to change  $p$  in  $q$

## Distance vs. Similarity

Distance → min

Similarity → max

# Unsupervised learning

## Other names

- clustering

## Process → 2 steps

- Training
  - learning (determine), using an algorithm, the existing clusters
- Testing
  - test the model with new data (not seen in the training step)

## Feature

- DB is not labeled (for learning and sometimes also for testing)

## Learning quality

## Suitable problem types

- Identifying some classes (clusters)
  - Genome analysis
  - Image processing
  - Social network analysis
  - Market segmentation
  - Astronomic data analysis
  - Computer clustering
- Dimensional reduction
- Identifying data properties (causes, explanations, etc.)
- Modeling data density

### Internal criteria – high similarity within a cluster and reduced one between clusters

- distance inside the cluster
  - distance between the clusters
  - Davies-Bouldin index
  - Dunn Index
- External criteria – using benchmarks made from already grouped data sets
    - Comparison with known data – almost impossible in practice
    - Precision
    - Recall
    - F1-measure

# Active learning

The algorithm can receive supplementary information during the learning step in order to improve it's performance.

Example: what is the subset of the training data that will facilitate the learning process.

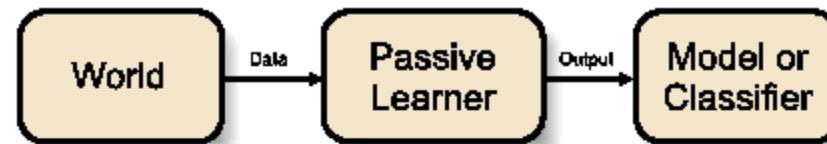


Figure 1.1: General schema for a passive learner.



Figure 1.2: General schema for an active learner.

# Reinforcement learning

## Aim

Learning, over a period of time, a course of action (behavior) that maximizes long-term rewards (earnings)

## Characteristic

Interaction with the environment (actions → rewards)  
Decision sequence

## Problems' type

Ex. Training a Dog (Good and Bad Dog)

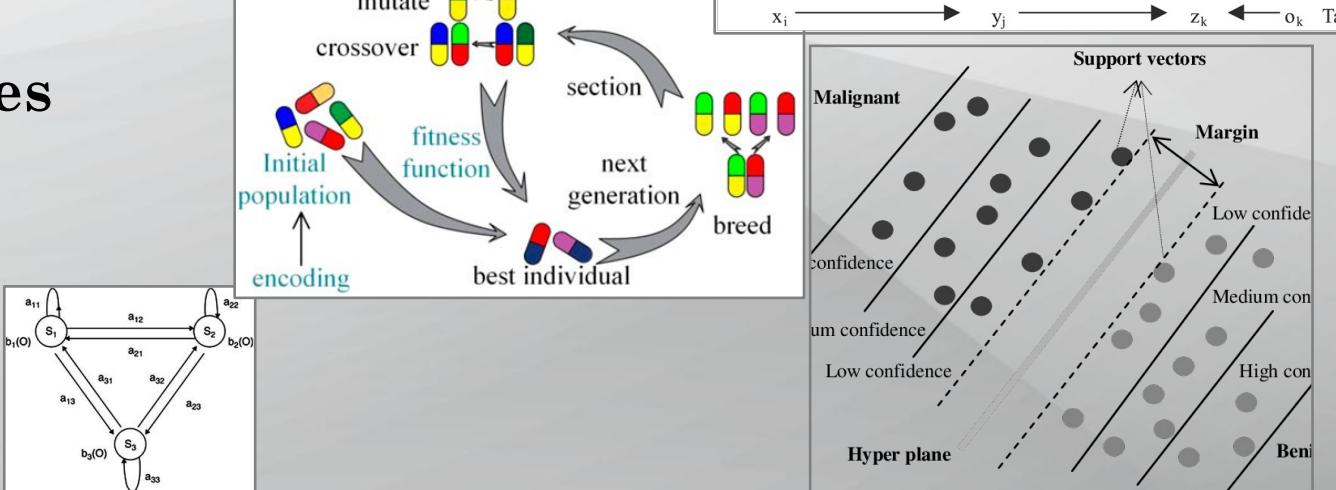
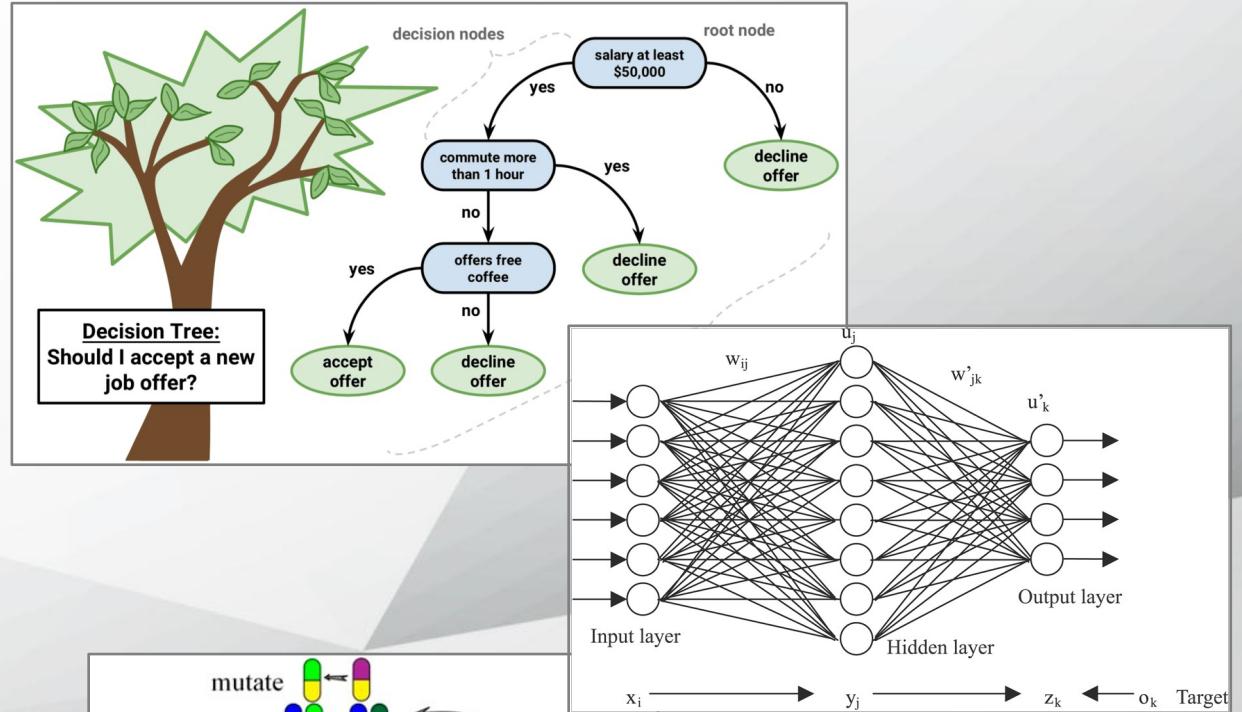
## Supervised learning

Decision → consequence (malignant or benign cancer)

# Intelligent Systems

Based on algorithm

- Decision trees
- Artificial Neural Networks
- Evolutionary algorithms
- Support Vector Machines
- Hidden Markov Models



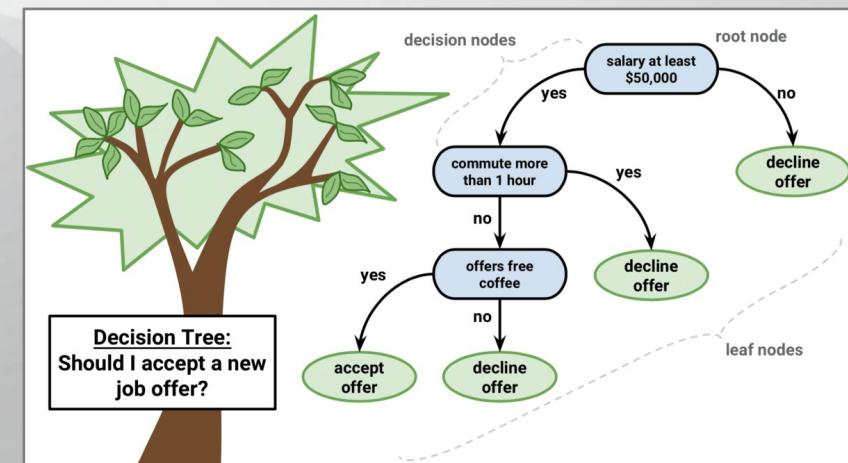
# Decision Trees (DT)

## Aim

- divide a collection of articles in smaller sets by successively applying some decision rules → asking more questions  
each question is addressed based on the answer of the previous question
- elements are characterized by non-metric information

## Definition

- Decision tree – a special graph (bi-color and oriented tree)
  - Contains three node types:
    - Decision nodes → possibilities of decider (a test on an attribute of item that must be classified)
    - Hazard nodes → random events outside the control of decider (exam results, therapy consequences)
    - Result nodes → final states that have a utility or a label
  - Decision and hazard nodes alternate on the tree levels
  - Result nodes → leaf (terminal nodes)
  - (oriented) edges of the tree consequences of decisions (can be probabilistic)
- Each internal node corresponds to an attribute
- Each branch under a node (attribute) corresponds to the value of that attribute
- Each leaf corresponds to a class



# Problems solved with DT

## Properties

- problem's instances are represented by a fixed number of attributes, each attribute having a finite number of values;
- objective function takes discrete values;
- DT represents a dis-junction of more conjunctions, each conjunction being “attribute  $a_i$  has value  $v_j$ ”;
- training data could contain errors;
- training data could be incomplete;
  - Some data have not all attributes.

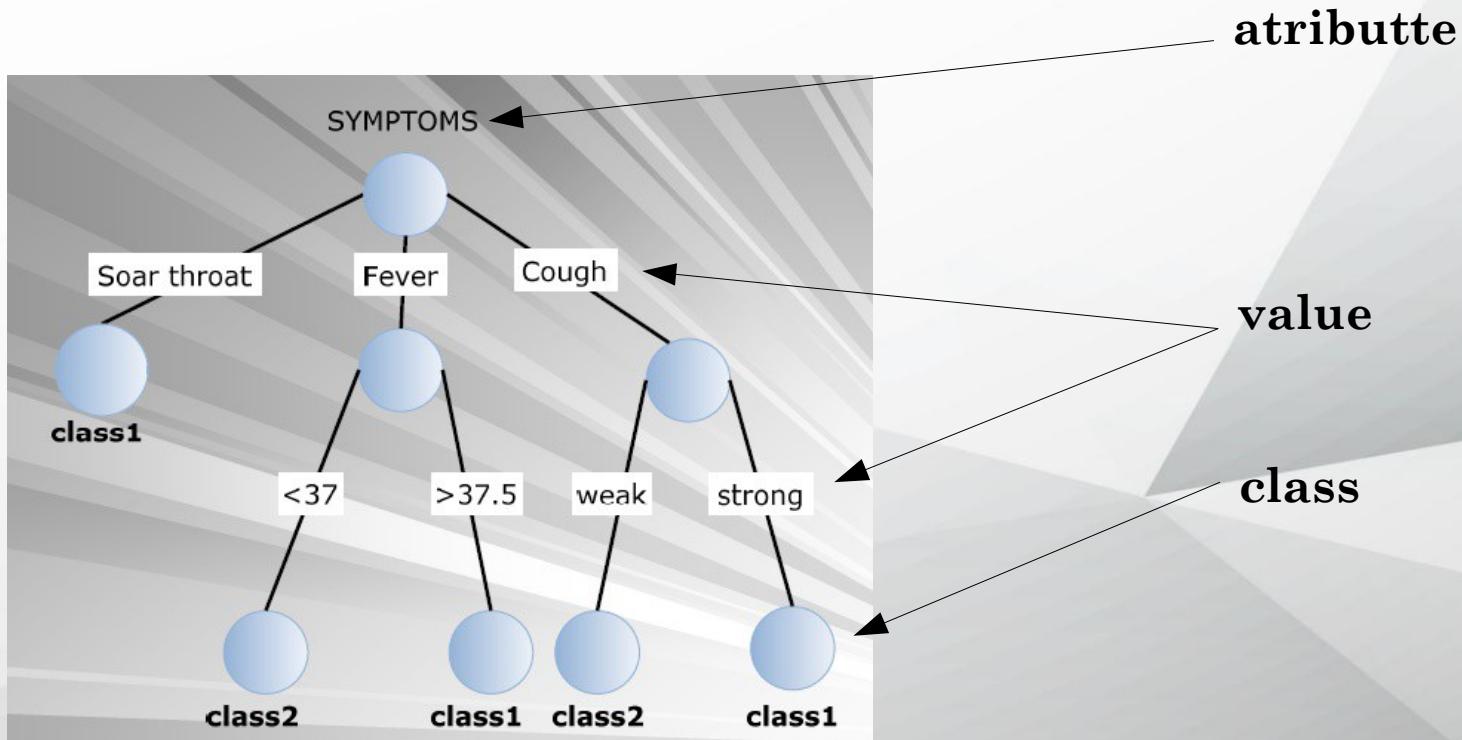
## Classification problem

- Binary classifications
  - Instances are  $[(attribute_{ij}, value_{ij}), class_i], i=1,2,\dots,n, j=1,2,\dots,m$ , and  $class_i$  taking 2 values;
- Multi-class (k-class)
  - Instances are  $[(attribute_{ij}, value_{ij}), class_i], i=1,2,\dots,n, j=1,2,\dots,m$ , and  $class_i$  taking  $k$  values;

## Regression problems

- instead to label each node by the label of a class, each node has associated a real value or a function that depends on the inputs of that node;
- input space is split in decision regions by parallel cuttings to  $Ox$  and  $Oy$ ;
- discrete outputs are transformed in continuous functions;
- quality of problem solving:
  - Prediction error (square or absolute)

# Examples of DT



# Example of train data for DT

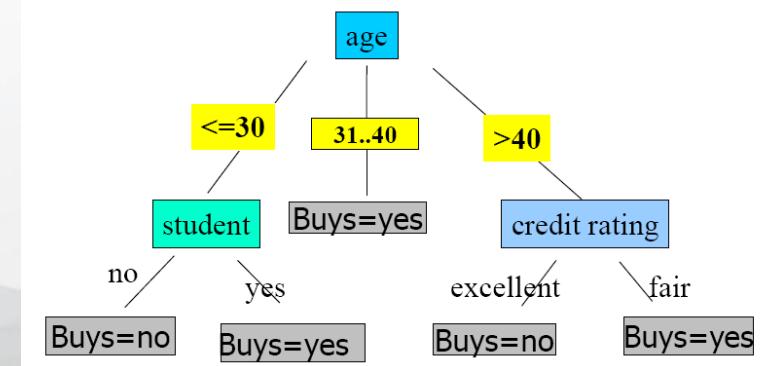
## Credits

Approved or not

ID	Age	Has_Job	Own_House	Credit_Rating	Class
1	young	false	false	fair	No
2	young	false	false	good	No
3	young	true	false	good	Yes
4	young	true	true	fair	Yes
5	young	false	false	fair	No
6	middle	false	false	fair	No
7	middle	false	false	good	No
8	middle	true	true	good	Yes
9	middle	false	true	excellent	Yes
10	middle	false	true	excellent	Yes
11	old	false	true	excellent	Yes
12	old	false	true	good	Yes
13	old	true	false	good	Yes
14	old	true	false	excellent	Yes
15	old	false	false	fair	No

# Examples of DT

rec	Age	Income	Student	Credit_rating	Buys_computer(CLASS)
r1	<=30	High	No	Fair	No
r2	<=30	High	No	Excellent	No
r3	31...40	High	No	Fair	Yes
r4	>40	Medium	No	Fair	Yes
r5	>40	Low	Yes	Fair	Yes
r6	>40	Low	Yes	Excellent	No
r7	31...40	Low	Yes	Excellent	Yes
r8	<=30	Medium	No	Fair	No
r9	<=30	Low	Yes	Fair	Yes
r10	>40	Medium	Yes	Fair	Yes
r11	<=30	Medium	Yes	Excellent	Yes
r12	31...40	Medium	No	Excellent	Yes
r13	31...40	High	Yes	Fair	Yes
r14	>40	Medium	No	Excellent	No



# Process – DT

## **Tree construction (induction)**

Based on training data

Works bottom-up or top-down (splitting)

## **Using the tree as a problem solver**

All decisions performed along a path  
from the root to a leaf form a rule

Rules from DT are used for labeling  
new data

## **Pruning**

Identify and move/eliminate branches  
that reflect noise or exceptions

# Tree construction

## Split the training data into subsets based on the characteristics of data

a node – question related to a property

branches of a node – possible answers to the question of the node

initially, all examples are located in the root

- an attribute gives the root label and its values give the branches

on next levels, examples are partitioned based on their attributes → order of attributes

- for each node, an attribute is (recursively) chosen – its values → branches

splitting – greedy decision making

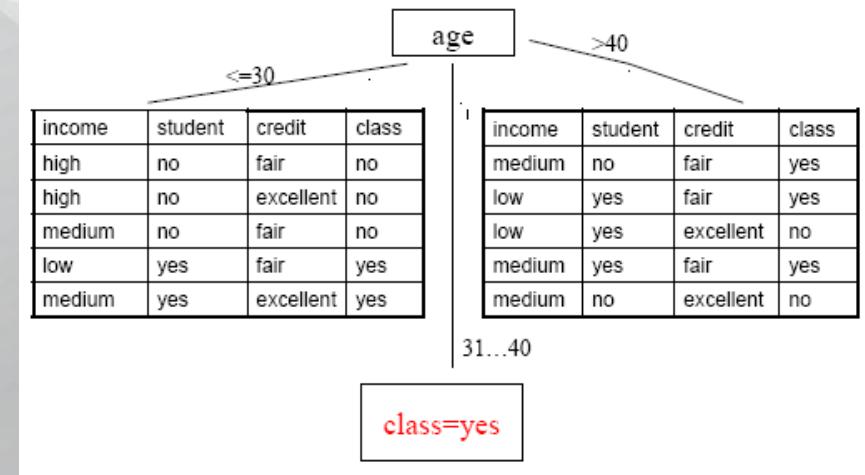
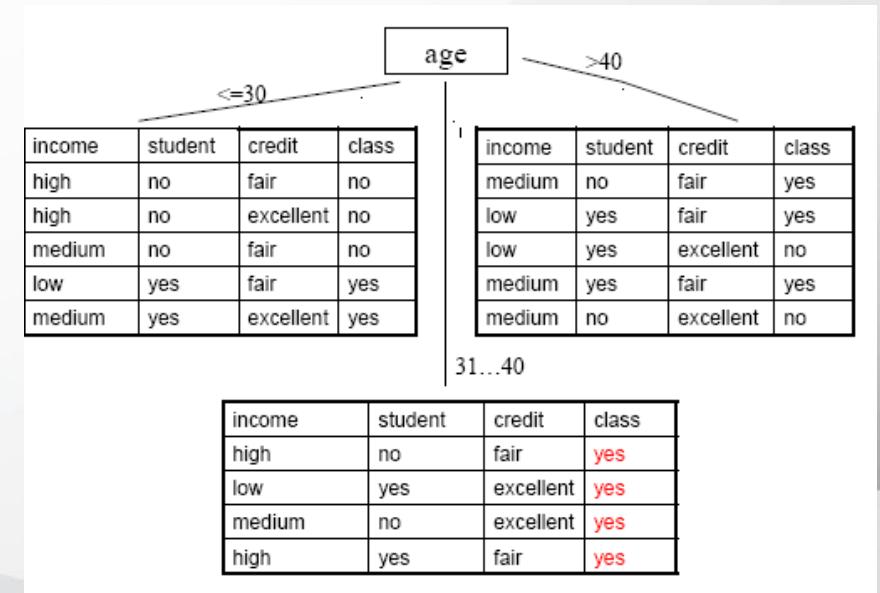
## Iterative process

stop conditions:

- all examples from a node belong to the same class → node is a leaf and is labeled by  $class_i$
- there are no left examples → node becomes a leaf and is labeled by the majority class of training data
- there are no attributes

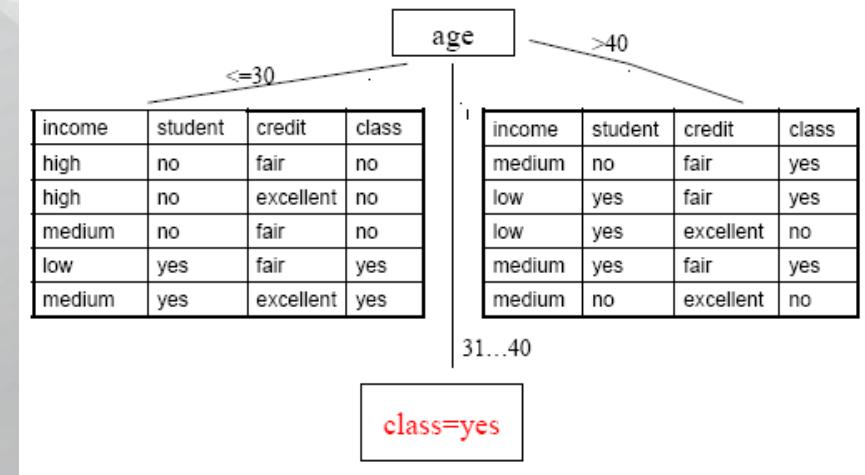
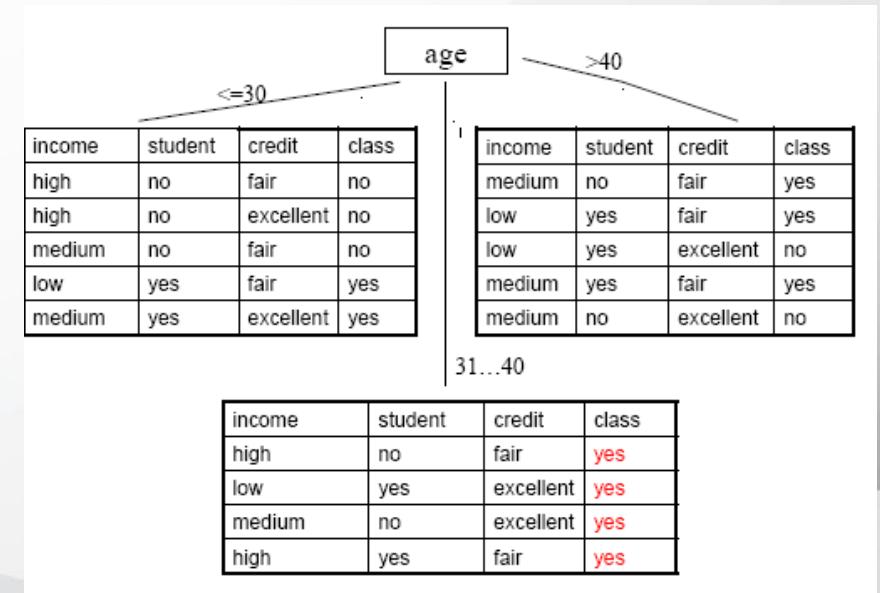
# Example – tree construction

rec	Age	Income	Student	Credit_rating	Buys_computer(CLASS)
r1	<=30	High	No	Fair	No
r2	<=30	High	No	Excellent	No
r3	31...40	High	No	Fair	Yes
r4	>40	Medium	No	Fair	Yes
r5	>40	Low	Yes	Fair	Yes
r6	>40	Low	Yes	Excellent	No
r7	31...40	Low	Yes	Excellent	Yes
r8	<=30	Medium	No	Fair	No
r9	<=30	Low	Yes	Fair	Yes
r10	>40	Medium	Yes	Fair	Yes
r11	<=30	Medium	Yes	Excellent	Yes
r12	31...40	Medium	No	Excellent	Yes
r13	31...40	High	Yes	Fair	Yes
r14	>40	Medium	No	Excellent	No



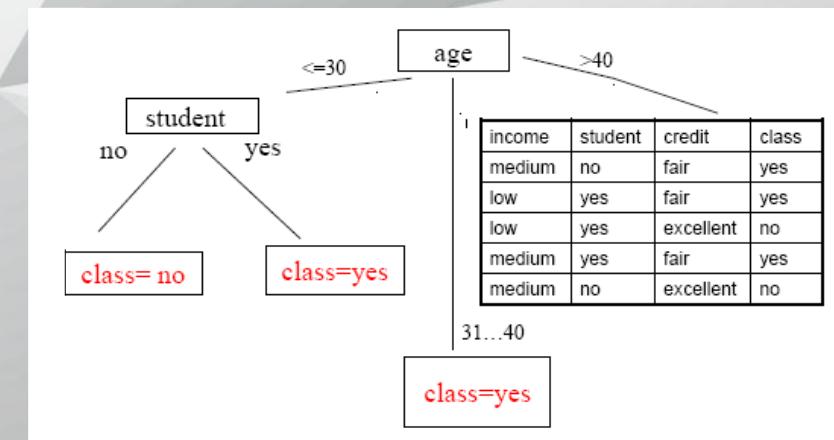
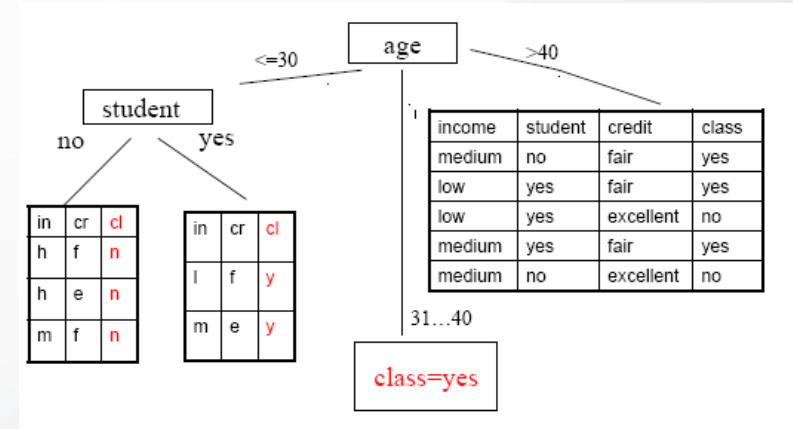
# Example – tree construction

rec	Age	Income	Student	Credit_rating	Buys_computer(CLASS)
r1	<=30	High	No	Fair	No
r2	<=30	High	No	Excellent	No
r3	31...40	High	No	Fair	Yes
r4	>40	Medium	No	Fair	Yes
r5	>40	Low	Yes	Fair	Yes
r6	>40	Low	Yes	Excellent	No
r7	31...40	Low	Yes	Excellent	Yes
r8	<=30	Medium	No	Fair	No
r9	<=30	Low	Yes	Fair	Yes
r10	>40	Medium	Yes	Fair	Yes
r11	<=30	Medium	Yes	Excellent	Yes
r12	31...40	Medium	No	Excellent	Yes
r13	31...40	High	Yes	Fair	Yes
r14	>40	Medium	No	Excellent	No



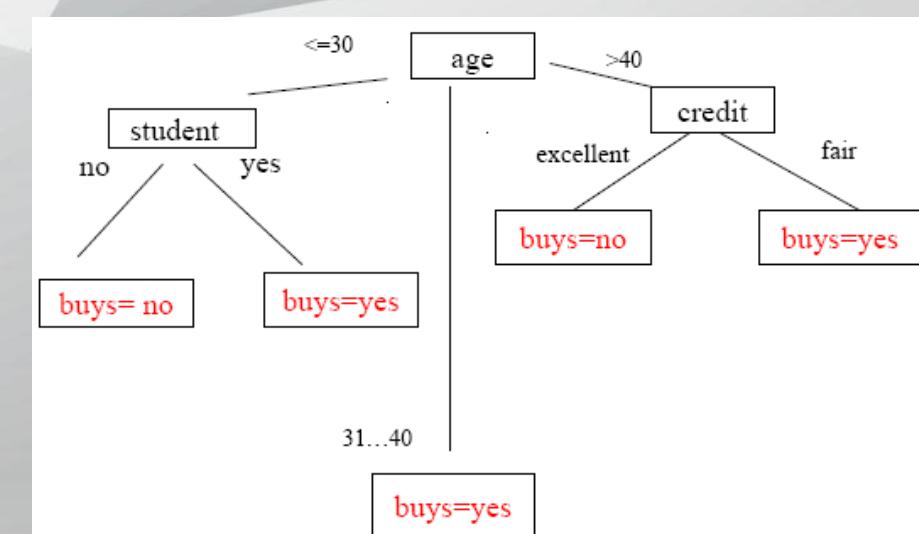
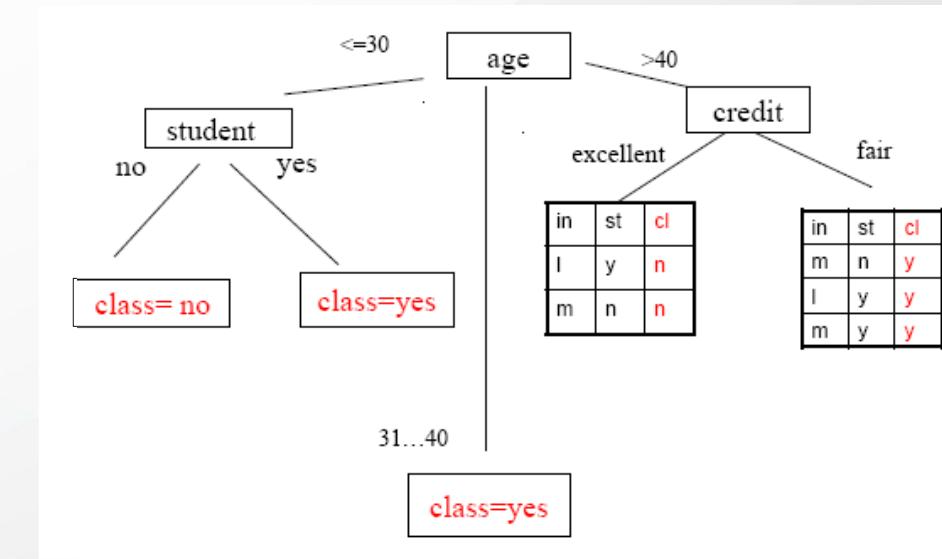
# Example – tree construction

rec	Age	Income	Student	Credit_rating	Buy Computer (CLASS)
r1	<=30	High	No	Fair	No
r2	<=30	High	No	Excellent	No
r3	31...40	High	No	Fair	Yes
r4	>40	Medium	No	Fair	Yes
r5	>40	Low	Yes	Fair	Yes
r6	>40	Low	Yes	Excellent	No
r7	31...40	Low	Yes	Excellent	Yes
r8	<=30	Medium	No	Fair	No
r9	<=30	Low	Yes	Fair	Yes
r10	>40	Medium	Yes	Fair	Yes
r11	<=30	Medium	Yes	Excellent	Yes
r12	31...40	Medium	No	Excellent	Yes
r13	31...40	High	Yes	Fair	Yes
r14	>40	Medium	No	Excellent	No



# Example – tree construction

rec	Age	Income	Student	Credit_rating	Buy Computer (CLASS)
r1	<=30	High	No	Fair	No
r2	<=30	High	No	Excellent	No
r3	31...40	High	No	Fair	Yes
r4	>40	Medium	No	Fair	Yes
r5	>40	Low	Yes	Fair	Yes
r6	>40	Low	Yes	Excellent	No
r7	31...40	Low	Yes	Excellent	Yes
r8	<=30	Medium	No	Fair	No
r9	<=30	Low	Yes	Fair	Yes
r10	>40	Medium	Yes	Fair	Yes
r11	<=30	Medium	Yes	Excellent	Yes
r12	31...40	Medium	No	Excellent	Yes
r13	31...40	High	Yes	Fair	Yes
r14	>40	Medium	No	Excellent	No



# ID3/C4.5 algorithm

```
generate(D, A){    //D – a partitioning of training data, A – list of attributes
    create a new node N
    if examples from D belong to a single class C then
        node N becomes a leaf and is labeled by C
        return node N
    else
        if A=∅ then
            node N becomes a leaf and is labeled by majority class of D
            return node N
        else
            separation_attribute = AttributeSelection(D, A)
            label node N by separation_attribute
            for all possible values vj of separation_attribute
                let Dj – set of examples from D that have separation_attribute=vj
                if Dj =∅ then
                    add a leaf (to node N) labeled by majority class of D
                else
                    add a node (to node N) return by generate(Dj, A–separation_attribute)
            return node N
}
```

Greedy, recursive, top-down, divide-and-conquer

AttributeSelection(D,A)

selects the attribute that corresponds to a node (root or internal node)

- Random
- Attribute with the fewest/most values
- Based on a pre-established order:
- Information gain
  - Gain rate
  - Distance between partitions created by the attribute

# Information gain

An impurity measure

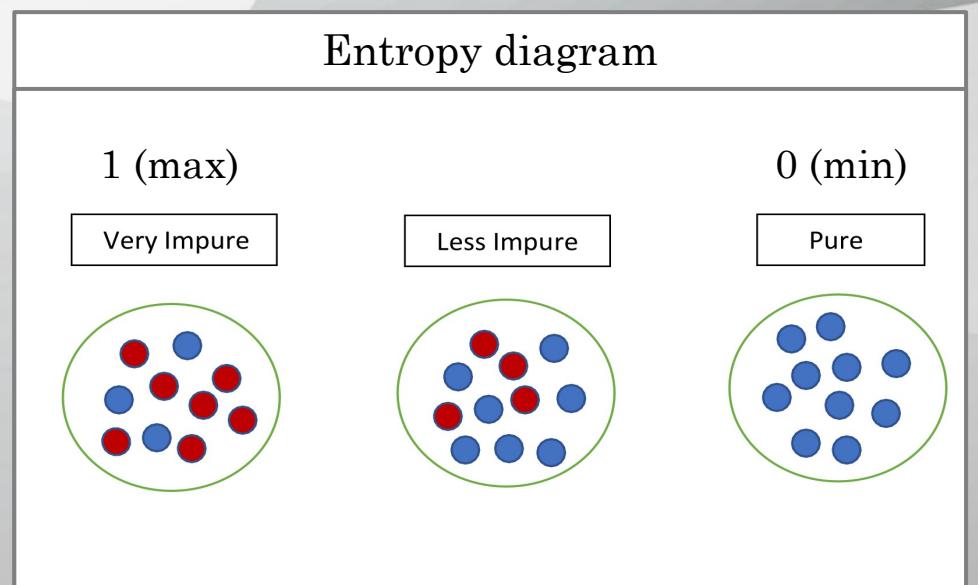
- 0 (minim) – if all examples belong to the same class
- 1 (maxim) – if examples are uniform distributed over classes

Based on data entropy

- Expected number of bits required by coding the class of an element from data
- Binary classification (2 classes):  $E(S) = -p_+ \log_2 p_+ - p_- \log_2 p_-$ , where
  - $p_+$  - proportion of positive examples in dataset S
  - $p_-$  - proportion of negative examples in dataset S
- Multi-class classification:  $E(S) = \sum_{i=1, 2, \dots, k} -p_i \log_2 p_i$ 
  - data entropy related to target attribute (output attribute), where
    - $p_i$  – proportion of examples from class  $i$  in dataset S

Information gain of an attribute

- How the elimination of attribute  $a$  reduces the dataset's entropy
- $Gain(S, a) = E(S) - \sum_{v \in \text{values}(a)} |S_v| / |S| E(S_v)$
- $\sum_{v \in \text{values}(a)} |S_v| / |S| E(S_v)$  – expected information



# Example – Information gain

	a1	a2	a3	Class
d1	big	red	circle	class 1
d2	small	red	square	class 2
d3	small	red	circle	class 1
d4	big	blue	circle	class 2

$$S = \{d_1, d_2, d_3, d_4\} \Rightarrow p_+ = \frac{2}{4}, p_- = \frac{2}{4} \Rightarrow E(S) = -p_+ \log_2 p_+ - p_- \log_2 p_- = 1$$

$$S_{v=\text{big}} = \{d_1, d_4\} \Rightarrow p_+ = \frac{1}{2}, p_- = \frac{1}{2} \Rightarrow E(S_{v=\text{big}}) = 1$$

$$S_{v=\text{small}} = \{d_2, d_3\} \Rightarrow p_+ = \frac{1}{2}, p_- = \frac{1}{2} \Rightarrow E(S_{v=\text{small}}) = 1$$

$$S_{v=\text{red}} = \{d_1, d_2, d_3\} \Rightarrow p_+ = \frac{2}{3}, p_{\text{minus}} = \frac{1}{3} \Rightarrow E(S_{v=\text{red}}) = 0.923$$

$$S_{v=\text{blue}} = \{d_4\} \Rightarrow p_+ = 0, p_- = 1 \Rightarrow E(S_{v=\text{blue}}) = 0$$

$$S_{v=\text{circle}} = \{d_1, d_3, d_4\} \Rightarrow p_+ = \frac{2}{3}, p_{\text{minus}} = \frac{1}{3} \Rightarrow E(S_{v=\text{circle}}) = 0.923$$

$$S_{v=\text{square}} = \{d_4\} \Rightarrow p_+ = 0, p_- = 1 \Rightarrow E(S_{v=\text{square}}) = 0$$

Formula for information gain

$$\text{Gain}(S, a) = E(S) - \sum_{v \in \text{values}(a)} \frac{|S_v|}{|S|} E(S_v)$$

$$\rightarrow \text{Gain}(S, a_1) = 1 - \left( \frac{|S_{v=\text{big}}|}{|S|} E(S_{v=\text{big}}) + \frac{|S_{v=\text{small}}|}{|S|} E(S_{v=\text{small}}) \right) = 1 - \left( \frac{2}{4} 1 + \frac{2}{4} 1 \right) = 0$$

$$\rightarrow \text{Gain}(S, a_2) = 1 - \left( \frac{|S_{v=\text{red}}|}{|S|} E(S_{v=\text{red}}) + \frac{|S_{v=\text{blue}}|}{|S|} E(S_{v=\text{blue}}) \right) = 1 - \left( \frac{3}{4} 0.923 + \frac{1}{4} 0 \right) = 0.307 \quad \star$$

$$\rightarrow \text{Gain}(S, a_3) = 1 - \left( \frac{|S_{v=\text{circle}}|}{|S|} E(S_{v=\text{circle}}) + \frac{|S_{v=\text{square}}|}{|S|} E(S_{v=\text{square}}) \right) = 1 - \left( \frac{3}{4} 0.923 + \frac{1}{4} 0 \right) = 0.307 \quad \star$$

# Gain rate

Penalises an attribute by integrating a new term that depends on spreading degree and on uniformity degree of separation – *Split information*

## ***Split information***

- entropy related to possible values of attribute  $a$
- describes the potential worth of splitting a branch from a node

$$\text{SplitInformation}(S, a) = - \sum_{v=\text{value}(a)} \frac{|S_v|}{|S|} \log_2 \frac{|S_v|}{|S|}$$

where  $S_v$  is the proportion of examples from dataset  $S$  that have attribute  $a$  with value  $v$

***Information gain ratio*** is the ratio between the *information gain* and the *split information* value:

$$IGT(S, a) = \frac{\text{gain}(S, a)}{\text{SplitInformation}(S, a)}$$

aims to reduce a bias towards multi-valued attributes!

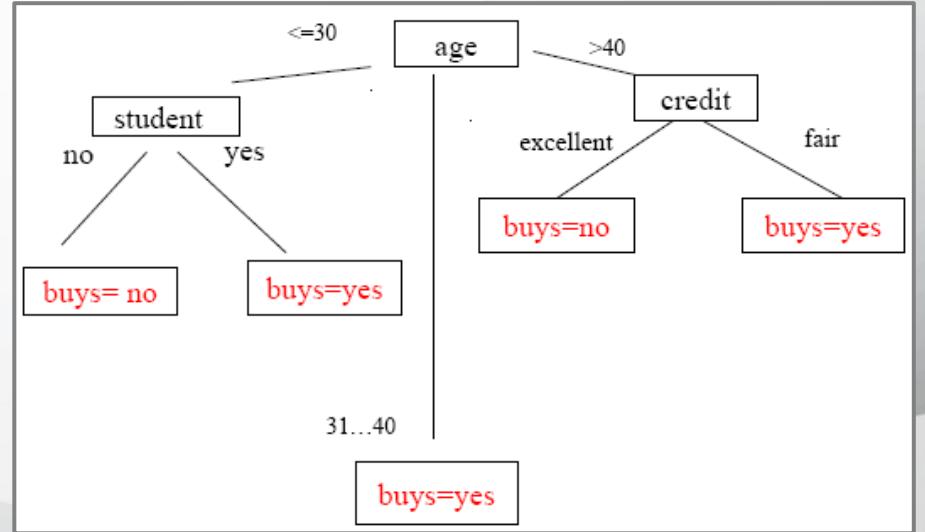
# Using a tree as a solver

Extract the rules from the constructed tree

- IF  $age = “<=30”$  AND  $student = “no”$  THEN  $buys\_computer = “no”$
- IF  $age = “<=30”$  AND  $student = “yes”$  THEN  $buys\_computer = “yes”$
- IF  $age = “31\dots 40”$  THEN  $buys\_computer = “yes”$
- IF  $age = “>40”$  AND  $credit\_rating = “excellent”$  THEN  $buys\_computer = “no”$
- IF  $age = “>40”$  AND  $credit\_rating = “fair”$  THEN  $buys\_computer = “yes”$

Use the rules for classifying the test data (new data)

- for  $x$  (a data without class) – rules can be written as predicates
- IF  $age(x, <=30)$  AND  $student(x, no)$  THEN  $buys\_computer(x, no)$
- IF  $age(x, <=30)$  AND  $student(x, yes)$  THEN  $buys\_computer(x, yes)$



## Difficulties

- *Underfitting* – DT constructed on training data is too simple → large classification error during training and testing
- *Overfitting* – DT constructed on training data match the training data, but it cannot generalize new data

## Solutions

- Pruning – remove some branches (not useful, redundant) → smaller tree
- Cross-validation

# Pruning a tree

## *pre-pruning*

Increasing the tree is stopped during construction by stopping the division of nodes that become leaf labeled by majority class of examples from that node

## *post-pruning*

After the DT is constructed, eliminate the branches of some nodes that become leaf → classification error reduces (on testing data)

### Motivation – simplifying the tree

- After the DT is constructed, classification rules are extracted in order to represent the knowledge as *if-then* rules (easy to understand)
- A rule is created by traversing the DT from root to a leaf
- Each pair (*attribute, value*)  $\leftrightarrow$  (*node, edge*) – is a conjunction in the premise of the rule (*if part*), except the last node of the path that is a leaf and represents the consequence (*output, then part*) of the rule

# Conclusion

## Advantages

- Easy to understand and interpret
- Can use nominal or categorized data
- Decision logic can be easily followed (rules are visible)
- Works better with large data

## Disadvantages

- Instability due to some changes in the training data
- Complexity due to the representation
- Difficult to use
- The DT construction is expensive
- The DT construction requires a lot of information

Tool example:

WEKA J48



# **THANK YOU !**

# **Artificial Neural Networks**

Artificial Intelligence

# Intelligent Systems - ANNs

## Aim example

binary classification for any input data  
(discrete or continuous)

- data can be separated by:
  - ◆ a line  $\rightarrow ax + by + c = 0$  (if  $m = 2$ )
  - ◆ a plan  $\rightarrow ax + by + cz + d = 0$  (if  $m = 3$ )
  - ◆ a hyperplan  $\rightarrow \sum a_i x_i + b = 0$  (if  $m > 3$ )
- How do we identify the optimal values of  $a, b, c, d, a_i$ ?
  - ◆ Artificial Neural Networks (ANNs)
  - ◆ Support Vector Machines (SVMs)

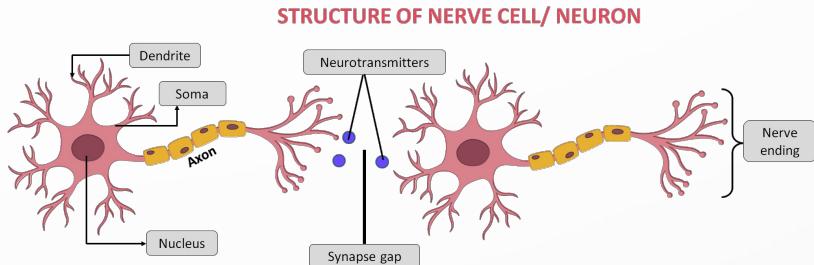
## Why ANN?

some tasks can be easily done by humans, but they are difficult to be encoded as algorithms

- Shape recognition
  - Old friends
  - Handwritten
  - Voice
- Rational processes
  - Car driving
  - Piano playing
  - Basketball playing
  - Swimming

such tasks are too difficult to be formalized and done by a rational process

# The learning process (brain)



## Human brain

- ~10.000.000.000 of neurons connected through synapses

## A neuron

- has a body (soma), an axon, and more dendrites
- can be in a given state
  - Active state – if the input information is over a given stimulation threshold
  - Passive state – otherwise

## Synapse

- link between the axon of a neuron and the dendrites of other neurons
- take part to information exchange between neurons
- 5.000 connections/neuron (average)
- during a life, new connections can appear

## Brain → Neural network

- complex system, non-linear and parallel that processes information
- information is stored and processed by the entire network, not only by a part of network

## Models for information processing:

- learning
- storing
- memorizing

## Learning

- a basic characteristic
- useful connections become permanent (others are eliminated)

## Memory

### *Short time memory*

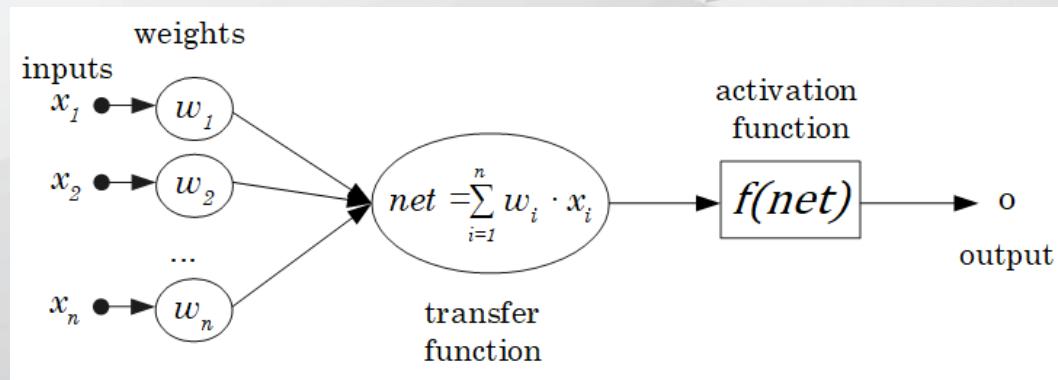
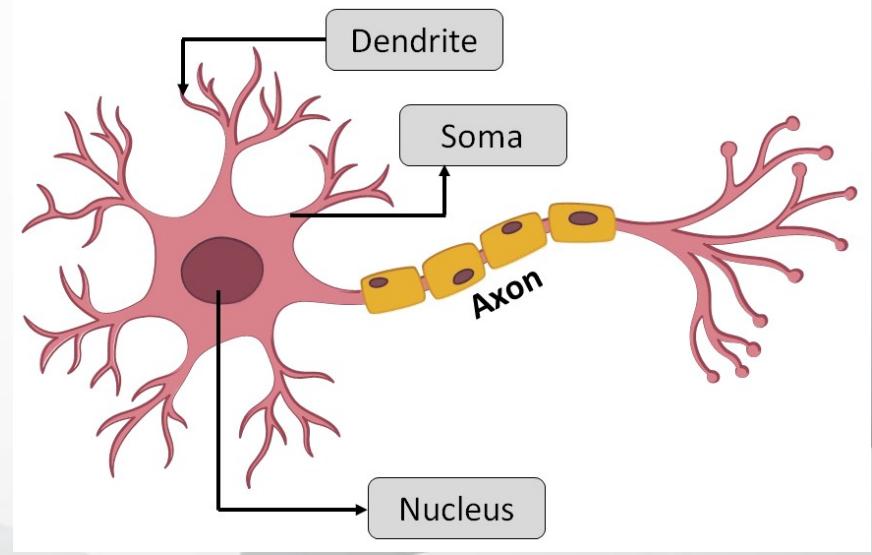
- immediately → 30 sec.
- working memory

### *Long term memory*

- capacity
- increasing along life
- limited → learning a poetry strophe by strophe
- influenced by emotional states

# Biology versus artificial

BNN	ANN
Soma	Node
Dendrite	Input
Axon	Output
Activation	Processing
Synapse	Weighted connection

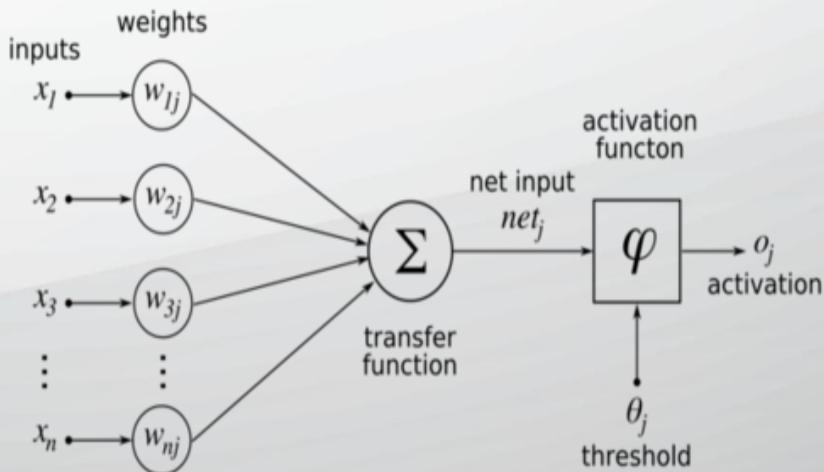


# The perceptron

## Learning Objectives

By the end of this section you should be able to answer there questions:

1. what is a perceptron?
2. what activation function has a perceptron?
3. what problems can (and can't) be solved by a perceptron?
4. how is trained a perceptron?



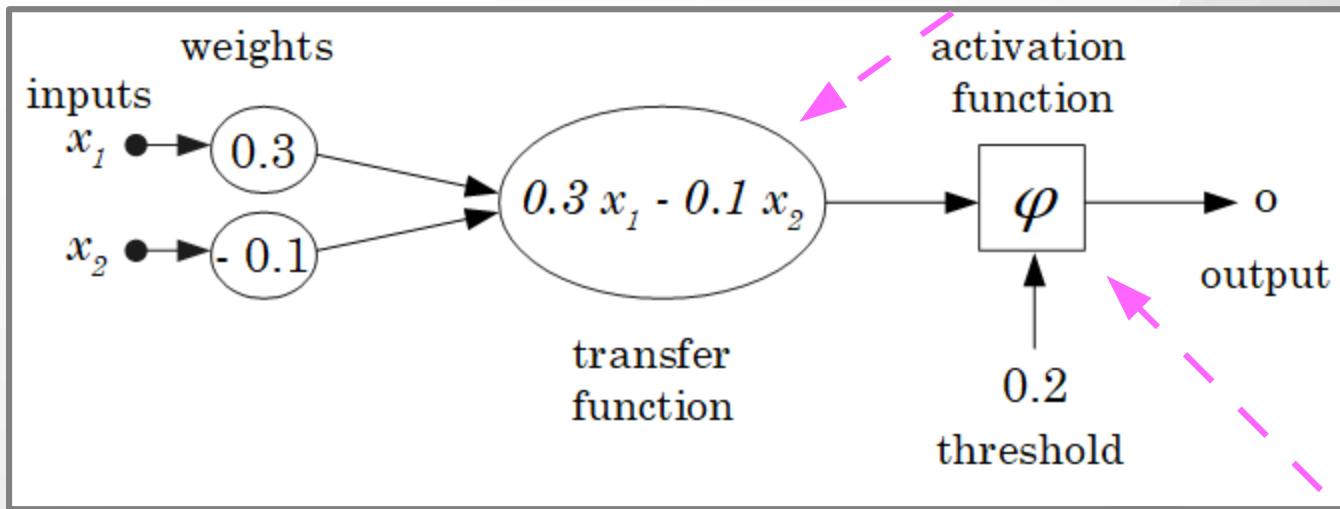
## It is the first model of a neuron

- invented by Frank Rosenblatt (1928 – 1971) an American psychologist
- the original perceptron was designed to take a number of binary inputs, and produce one binary output (0 or 1)
- uses different **weights** to represent the importance of each **input**, and that the sum of the values should be greater than a **threshold value** before making a **decision** like true or false (0 or 1)
- the original algorithm:
  1. Set a threshold value
  2. Multiply all inputs with its weights
  3. Sum all the results
  4. Activate the output

# Perceptron – example

Consider the parameters:  $w_1=0.3$     $w_2=-0.1$     $\theta=0.2$

$$net = \sum_{i=1}^n w_i * x_i$$



For the input:  $x=(x_1, x_2)=(1, 0)$

We get the output:  $\varphi(0.3 \times 1 - 0.1 \times 0) = \varphi(0.3) = 1$

**Activation function**

$$\varphi(net) = \begin{cases} 0, & \text{if } net < \theta \\ 1, & \text{if } net \geq \theta \end{cases}$$

threshold function

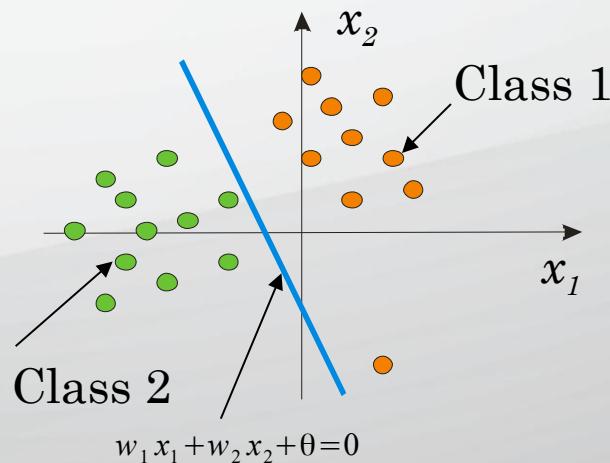
# Solving capacity

## Observe!

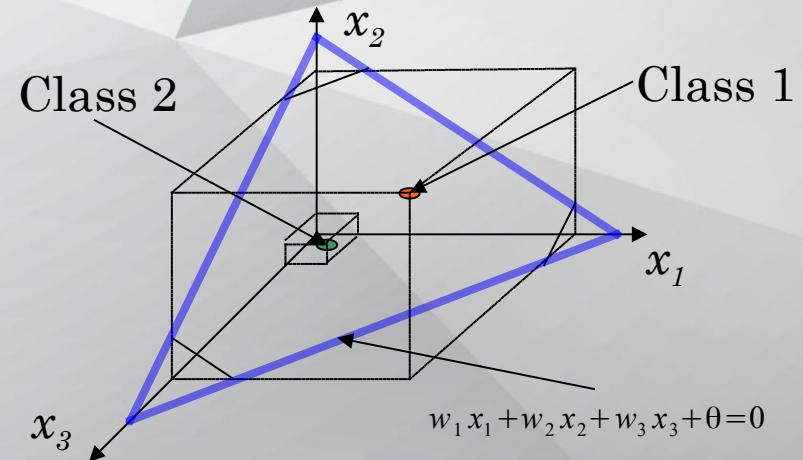
From the transfer function we get

$$y = \sum_{i=1}^n w_i x_i \quad - \text{equation of a hyperplane.}$$

If we compose the transfer function with the threshold function (the activation) we **linear separate** the space  $\mathbf{R}^n$  with this hyperplane in two regions.



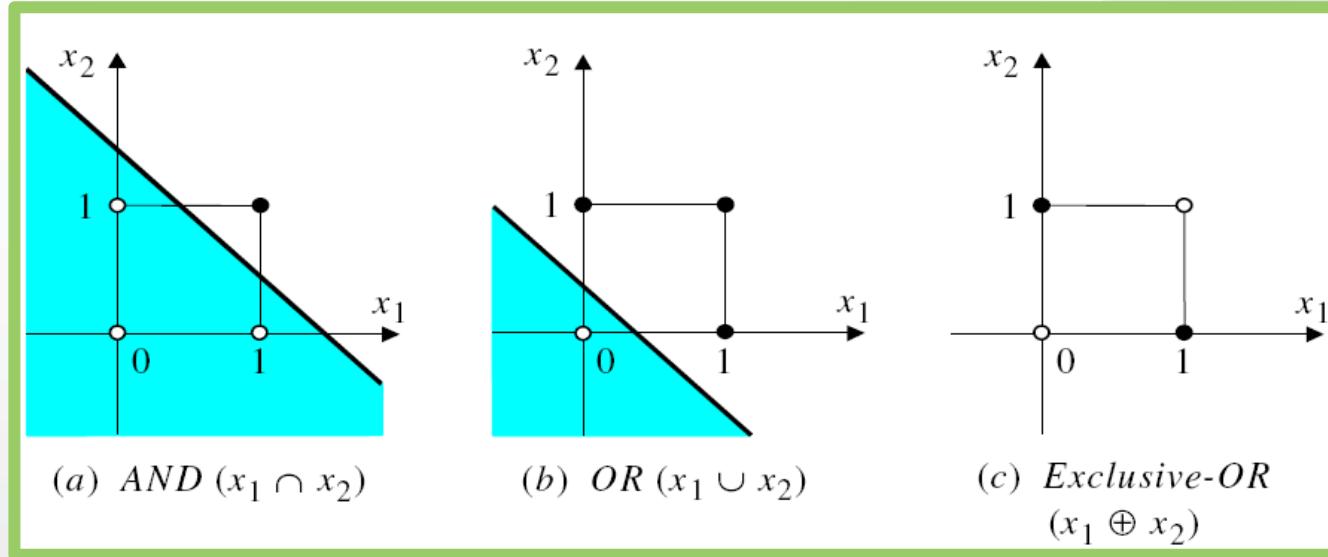
Binary classification with  $m=2$  input attributes



Binary classification with  $m=3$  input attributes

# Perceptron – limits

Non-linear separable data can not be classified.



A perceptron can learn And and OR operations,  
but it can not learn XOR operation (it is not linear separable)

Solutions

Neuron with continuous threshold  
More neurons

# Neuron training

## Perceptron's rule → perceptron's algorithm

1. Start by some random weights
2. Determine the quality of the model created for these weights for a **single input data**
3. Re-compute the weights based on the model's quality
4. Repeat (from step 2) until a maximum quality is obtained

## Delta's rule → algorithm of gradient descent

1. Start by some random weights
2. Determine the quality of the model created for these weights for **all input data**
3. Re-compute the weights based on the model's quality
4. Repeat (from step 2) until a maximum quality is obtained

Similar to perceptron's rule, but the model's quality is established based on all data (all training data).

# The perceptron – learning

Aim:

To identify the optimal weights ( $w_1, w_2, \dots, w_m$ ) by minimising the errors.

Training data set of  $n$  data

$$\{((x_1^d, x_2^d, \dots, x_m^d), t^d) : d = 1, 2, \dots, n\}$$

with  $m$  – the number of attributes.

The error is the difference between the real output  $y$  and the output  $o$  computed by the perceptron for the input  $(x_1, x_2, \dots, x_m)$ .

Perceptron's algorithm:

Based on error minimisation associated to an instance of train data

Modify the weights based on error associated to an instance of train data

# Perceptron's learning alg.

```
function training_perceptron( $\theta, \eta, m, n, \{(x_1^d, x_2^d, \dots, x_m^d), t^d\} : d=1, 2, \dots, n\})$ 
     $w_i = \text{random}(-1, 1)$ , where  $i=1, 2, \dots, m$  # initialise random the perceptron's weights

    do repeat
        for  $d = 1$  to  $n$  do
            activate the neuron and determine the output  $o^d = \varphi\left(\sum_{i=1}^n w_i * x_i^d\right)$ 
            compute the error  $e_d = t^d - o^d$ 
            determine the weight modification  $\Delta w_i = \eta e_d x_i^d$ 
            modify the weights  $w_i = w_i + \Delta w_i$ 
        until there are no incorrect classified examples

    return  $(w_1, w_2, \dots, w_m)$ 

end function
```

# Solving *logic AND* problem

AND	0 (False)	1 (True)
0 (False)	0	0
1 (True)	0	1

threshold

$$\theta=0.2$$

learning rate

$$\eta=0.1$$

Epoch	Inputs		Desired output $Y_d$	Initial weights		Actual output $Y$	Error $e$	Final weights	
	$x_1$	$x_2$		$w_1$	$w_2$			$w_1$	$w_2$
1	0	0	0	0.3	-0.1	0	0	0.3	-0.1
	0	1	0	0.3	-0.1	0	0	0.3	-0.1
	1	0	0	0.3	-0.1	1	-1	0.2	-0.1
	1	1	1	0.2	-0.1	0	1	0.3	0.0
2	0	0	0	0.3	0.0	0	0	0.3	0.0
	0	1	0	0.3	0.0	0	0	0.3	0.0
	1	0	0	0.3	0.0	1	-1	0.2	0.0
	1	1	1	0.2	0.0	1	0	0.2	0.0
3	0	0	0	0.2	0.0	0	0	0.2	0.0
	0	1	0	0.2	0.0	0	0	0.2	0.0
	1	0	0	0.2	0.0	1	-1	0.1	0.0
	1	1	1	0.1	0.0	0	1	0.2	0.1
4	0	0	0	0.2	0.1	0	0	0.2	0.1
	0	1	0	0.2	0.1	0	0	0.2	0.1
	1	0	0	0.2	0.1	1	-1	0.1	0.1
	1	1	1	0.1	0.1	1	0	0.1	0.1
5	0	0	0	0.1	0.1	0	0	0.1	0.1
	0	1	0	0.1	0.1	0	0	0.1	0.1
	1	0	0	0.1	0.1	0	0	0.1	0.1
	1	1	1	0.1	0.1	1	0	0.1	0.1

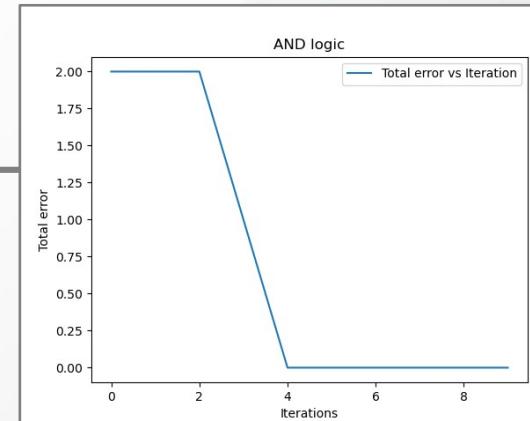
# Perceptron use

```
from numpy import random, dot, array
import matplotlib as mpl

def threshold(x):
    if x < 0.2 :
        return 0
    return 1

class Perceptron:
    def __init__(self, noInputs, activationFunction, learningRate):
        self.noInputs = noInputs
        self.weights = random.rand(self.noInputs)
        self.activationFunction = activationFunction
        self.output = 0
        self.learningRate = learningRate
        self.errorVariation = []
    def fire(self, inputs):
        self.output = self.activationFunction(
            inner(array(inputs), self.weights))
        return self.output

    def trainPerceptronRule(self, inputs, output):
        totalError = 0
        for i in range(len(inputs)):
            error = output[i] - self.fire(inputs[i])
            delta = self.learningRate*error*array(inputs[i])
            self.weights += delta
            totalError += error**2
        self.errorVariation.append(totalError)
```



```
def test1():
    # AND logic
    p = Perceptron(2, threshold, 0.1)
    x = [[1,1],[1,0],[0,1],[0,0]]
    t = [1,0,0,0]
    noIterations = 10
    for i in range(noIterations):
        p.trainPerceptronRule(x,t)
    print(p.weights)
    for j in range(len(x)):
        print(x[j], p.fire(x[j]))

    mpl.pyplot.plot(range(noIterations),
                    p.errorVariation, label = 'Total error
                                         vs Iteration')
    mpl.pyplot.xlabel('Iterations')
    mpl.pyplot.ylabel('Total error')
    mpl.pyplot.legend()
    mpl.pyplot.title('AND logic')
    mpl.pyplot.show()
```

# Artificial Neural Networks

## Learning Objectives

By the end of this section you should be able to answer there questions:

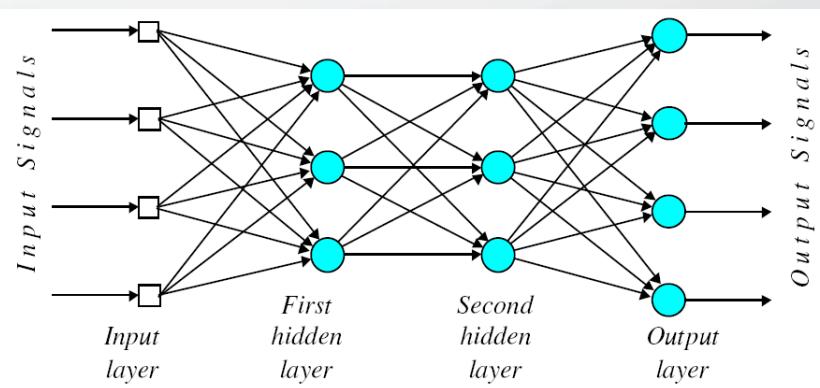
1. What is a *feed forward artificial neural network*?
2. How is organized an ANN?
3. What activation functions are used in ANNs?
4. How passes the signal trough an ANN?
5. How to train an ANN?

## A structure similar to a biological neural network

- 1943, neurophysiologist Warren McCulloch and mathematician Walter Pitts
- the original ANN was built with electrical circuits
- A set of nodes (units, neurons, processing elements) located in a graph with more layers

In a *feed forward network* the information moves in only one direction, **forward**, from the input nodes, through the hidden nodes (if any) to the output nodes.

– is the most simple type of ANN.



# Artificial Neural Networks

## Nodes

- Have inputs and outputs
- Perform a simple computing through an activation function
- Connected by weighted links
  - Links between nodes give the network structure
  - Links influence the performed computations

## Layers

- Input layer
  - Contains  $m$  nodes ( $m$  – the number of attributes of a data)
- Output layer
  - Contains  $r$  nodes ( $r$  – the number of outputs)
- Intermediate layers
  - Different structures
  - Different sizes

### Example of a feed forward network structure:

**6:4:10:2** → input layer with 6 units (artificial neurons), two hidden layers with 4 and 10 units respectively and an output layer with 2 units.

this structure can be used for a problem with 6 attributes and 2 outputs

each node from the first layer has one input and one output, each unit from the first hidden layer has 6 weights and one output, each unit from the second hidden layer has 4 weights and one output, each node from the output layer has 10 weights and one output.

# Artificial Neural Networks

## Learning process

A training data set of  $n$  data

$$\{((x_1^d, x_2^d, \dots, x_m^d), (t_1^d, t_2^d, \dots, t_r^d)) : d = 1, 2, \dots, n\}$$

where  $m$  – number of attributes and  $r$  – number of outputs

Form an ANN with  $m$  input nodes,  $r$  output nodes and an internal structure

- Some hidden layers, each layer having a given number of nodes
- With weighted connections between every 2 nodes of consecutive layers

Determine the optimal weights by minimising the error

- Difference between the real output  $y$  and the output computed by the network

# Problems solved by an ANN

- Problem data can be represented by pairs (attribute-value)
- Objective function can be:
  - Single or multi-criteria
  - Discrete or continuous (real values)
- Training data can be noised
- A large training time

# Neuron processing

Information is transmitted to the neuron → compute the weighted sum of inputs

$$net = \sum_{i=1}^n w_i * x_i$$

Neuron processes the information → by using an activation function

$$o = f(net)$$

- Constant function
- Step function
- Slope function
- Linear function
- Sigmoid function
- Gaussian function

# Activation function

## Constant function

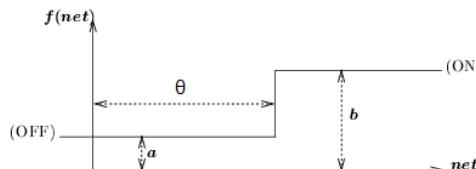
$$f(\text{net}) = \text{const.}$$



## Step function

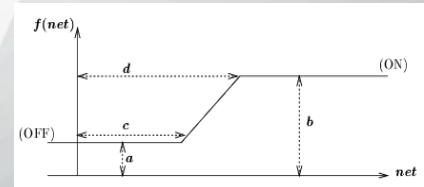
$\theta$  - threshold

$$f(\text{net}) = \begin{cases} a, & \text{if } \text{net} < \theta \\ b, & \text{if } \text{net} \geq \theta \end{cases}$$



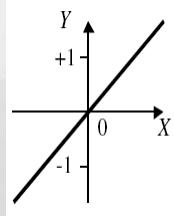
## Slope function

$$f(\text{net}) = \begin{cases} a, & \text{if } \text{net} \leq c \\ b, & \text{if } \text{net} \geq d \\ a + \frac{(\text{net} - c)(b - a)}{d - c}, & \text{otherwise} \end{cases}$$



## Linear function

$$f(\text{net}) = a * \text{net} + b$$

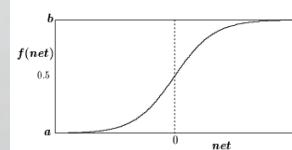


## Sigmoid function

$$f(\text{net}) = z + \frac{1}{1 + \exp(-x \cdot \text{net} + y)}$$

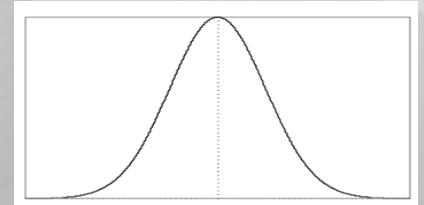
$$f(\text{net}) = \tanh(x \cdot \text{net} - y) + z$$

$$\text{where } \tanh(u) = \frac{e^u - e^{-u}}{e^u + e^{-u}}$$



## Gaussian function

$$f(\text{net}) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{1}{2}\left(\frac{\text{net} - \mu}{\sigma}\right)^2\right]$$



**Attention!** Also some other functions are in use – we will see this later!

# Gradient descent

- based on the error associated to the entire set of train data
- modify the weights in the direction of the steepest slope of error reduction  $E(\mathbf{w})$  for the entire set of train data

$$E(\mathbf{w}) = \frac{1}{2} \sum_{d=1}^n (t^d - o^d)^2$$

- How to determine the steepest slope? → derive  $E$  based on  $w$  (establish the gradient of error  $E$ )

$$\nabla E(\mathbf{w}) = \left( \frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \dots, \frac{\partial E}{\partial w_m} \right)$$

- Error's gradient is computed based on the neuron's activation function (that function must be differentiable → continuous)

➤ linear function       $f(\text{net}) = \sum_{i=1}^m w_i x_i^d$

➤ sigmoid function       $f(\text{net}) = \frac{1}{1 + e^{-\mathbf{w}\mathbf{x}}} = \frac{1}{1 + e^{-\sum_{i=1}^m w_i x_i^d}}$

- How are modified the weights ?       $\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$ , where  $i = 1, 2, \dots, m$

# Compute the error's gradient

- linear function

$$\begin{aligned}
 f(\text{net}) &= \sum_{i=1}^m w_i x_i^d \\
 \frac{\partial E}{\partial w_i} &= \frac{\partial \frac{1}{2} \sum_{d=1}^n (t^d - o^d)^2}{\partial w_i} = \frac{1}{2} \sum_{d=1}^n \frac{\partial (t^d - o^d)^2}{\partial w_i} = \frac{1}{2} \sum_{d=1}^n 2(t^d - o^d) \frac{\partial (t^d - \mathbf{w}\mathbf{x}^d)}{\partial w_i} \\
 \frac{\partial E}{\partial w_i} &= \sum_{d=1}^n (t^d - o^d) \frac{\partial (t^d - w_1 x_1^d - w_2 x_2^d - \dots - w_m x_m^d)}{\partial w_i} = \sum_{d=1}^n (t^d - o^d)(-x_i^d) \\
 \Delta w_i &= -\eta \frac{\partial E}{\partial w_i} = \eta \sum_{d=1}^n (t^d - o^d) x_i^d
 \end{aligned}$$

- sigmoid function

$$\begin{aligned}
 f(\text{net}) &= \frac{1}{1 + e^{-\mathbf{w}\mathbf{x}}} = \frac{1}{1 + e^{-\sum_{i=1}^m w_i x_i^d}} & y = s(z) = \frac{1}{1 + e^{-z}} \Rightarrow \frac{\partial s(z)}{\partial z} = s(z)(1 - s(z)) \\
 \frac{\partial E}{\partial w_i} &= \frac{\partial \frac{1}{2} \sum_{d=1}^n (t^d - o^d)^2}{\partial w_i} = \frac{1}{2} \sum_{d=1}^n \frac{\partial (t^d - o^d)^2}{\partial w_i} = \frac{1}{2} \sum_{d=1}^n 2(t^d - o^d) \frac{\partial (t^d - \text{sig}(\mathbf{w}\mathbf{x}^d))}{\partial w_i} = \sum_{d=1}^n (t^d - o^d)(1 - o^d)o^d(-x_i^d) \\
 \Delta w_i &= -\eta \frac{\partial E}{\partial w_i} = \eta \sum_{d=1}^n (t^d - o^d)(1 - o^d)o^d x_i^d
 \end{aligned}$$

# Simple and stochastic GDA

## Simple GDA

Initialisation of network weights

$$w_i = \text{random}(a, b), \text{ where } i = 1, 2, \dots, m$$

**While** not stop condition

$$\Delta w_i = 0, \text{ where } i = 1, 2, \dots, m$$

**For each** train example  $(\mathbf{x}_d, t_d)$ , where  $d = 1, 2, \dots, n$

Activate the neuron and determine the output  $o_d$

$$\text{Linear activation} \rightarrow o_d = \mathbf{w} \cdot \mathbf{x}_d$$

$$\text{Sigmoid activation} \rightarrow o_d = \text{sig}(\mathbf{w} \cdot \mathbf{x}_d)$$

**For each** weight  $w_i$ , where  $i = 1, 2, \dots, m$

Determine the weight modification

$$\Delta w_i = \Delta w_i - \eta \frac{\partial E}{\partial w_i}$$

where  $\eta$  - learning rate

**For each** weight  $w_i$ , where  $i = 1, 2, \dots, m$

$$\text{Modify the weights } w_i \quad w_i = w_i + \Delta w_i$$

**EndWhile**

## Stochastic GDA

Initialisation of network weights

$$w_i = \text{random}(a, b), \text{ where } i = 1, 2, \dots, m$$

**While** not stop condition

$$\Delta w_i = 0, \text{ unde } i = 1, 2, \dots, m$$

**For** a random sample subset from the training data set  
 $(\mathbf{x}_{d_i}, t_{d_i})$ , where  $d_i \in \{1, 2, \dots, n\}$

Activate the neuron and determine the output  $o_{d_i}$

$$\text{Linear activation} \rightarrow o_{d_i} = \mathbf{w} \cdot \mathbf{x}_{d_i}$$

$$\text{Sigmoid activation} \rightarrow o_{d_i} = \text{sig}(\mathbf{w} \cdot \mathbf{x}_{d_i})$$

**For each** weight  $w_i$ , where  $i = 1, 2, \dots, m$

Determine the weight modification

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

where  $\eta$  - learning rate

**For each** weight  $w_i$ , where  $i = 1, 2, \dots, m$

$$\text{Modify the weights } w_i \quad w_i = w_i + \Delta w_i$$

**EndWhile**

# Neuron learning

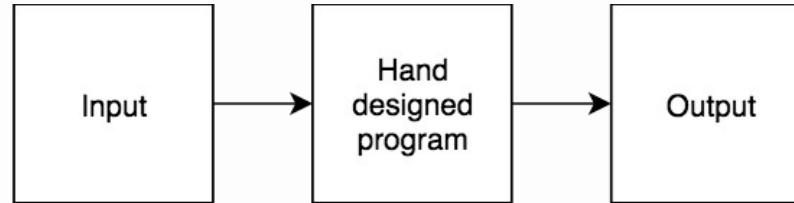
Differences	Perceptron's algorithm	Gradient descent algorithm (delta rule)
What does $o^d$ represent?	$o^d = \text{sign}(\mathbf{w}\mathbf{x}^d)$	$o^d = \mathbf{w}\mathbf{x}^d$ or $o^d = \text{sig}(\mathbf{w}\mathbf{x}^d)$
Convergence	After a finite # of steps (until the perfect separation)	Asymptotic (to minimum error)
Solved problems	With linear separable data	Any data (linear separable or non-linear)
Neuron's output	Discrete and with threshold	Continuous and without threshold

# **THANK YOU !**

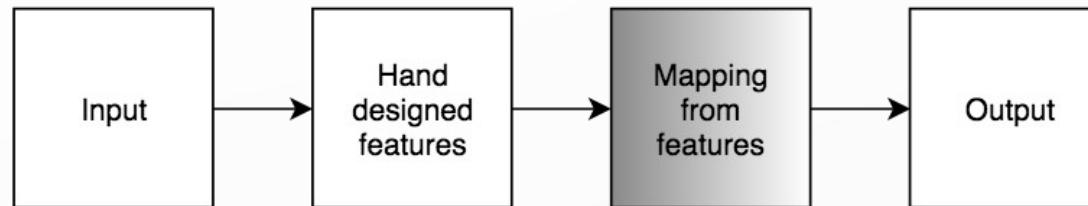
# Artificial Neural Networks

# Learning multiple components

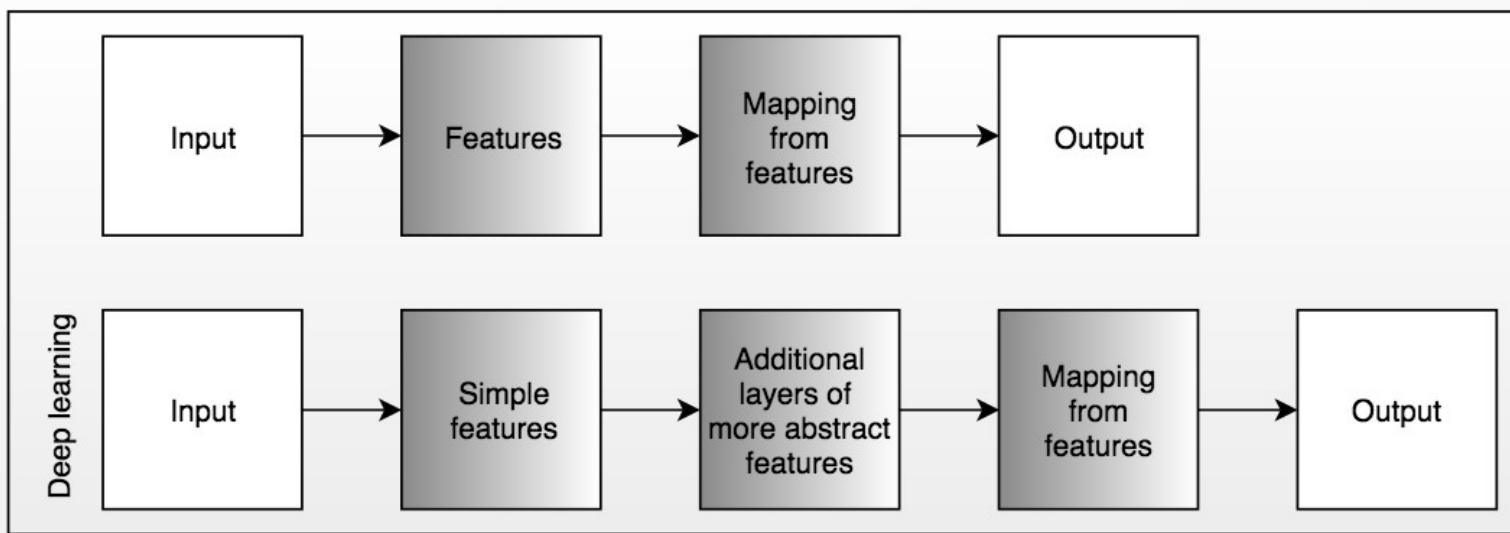
Rule-based systems



Classic machine learning

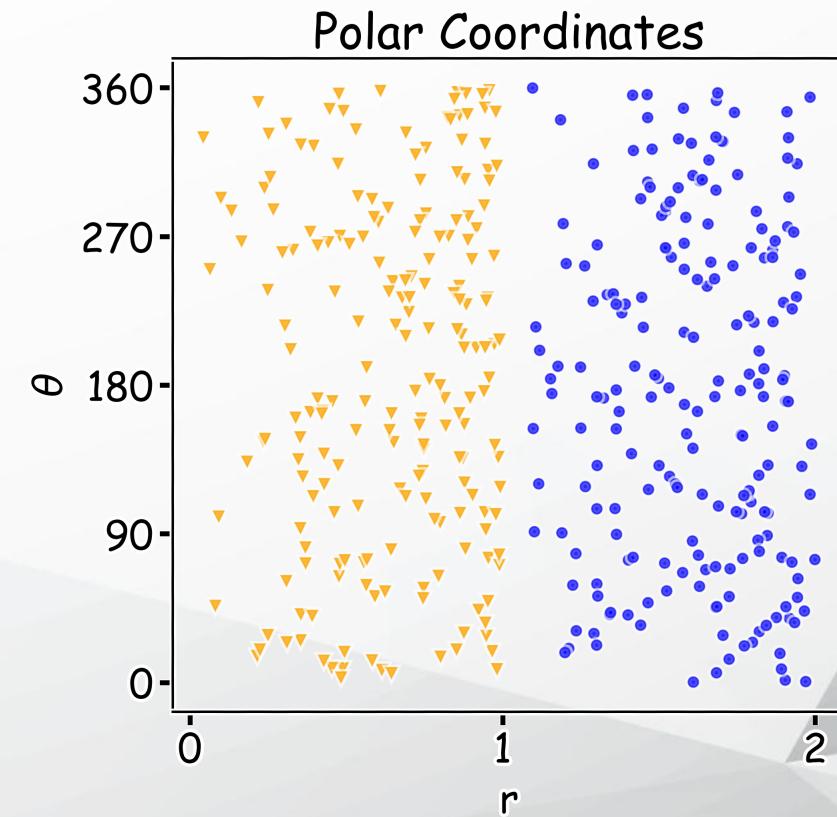
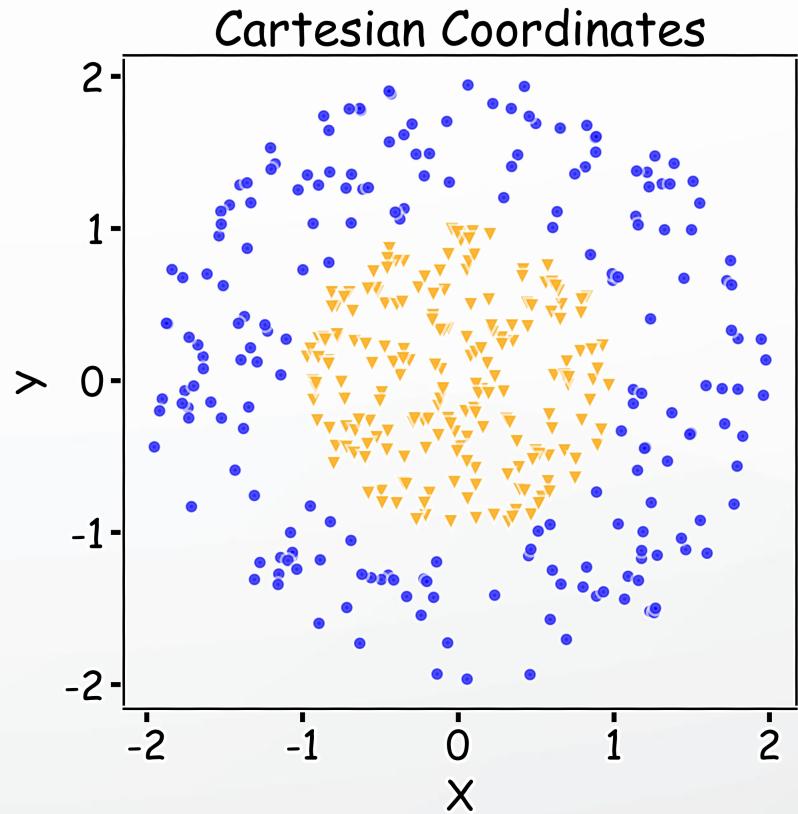


Representation learning

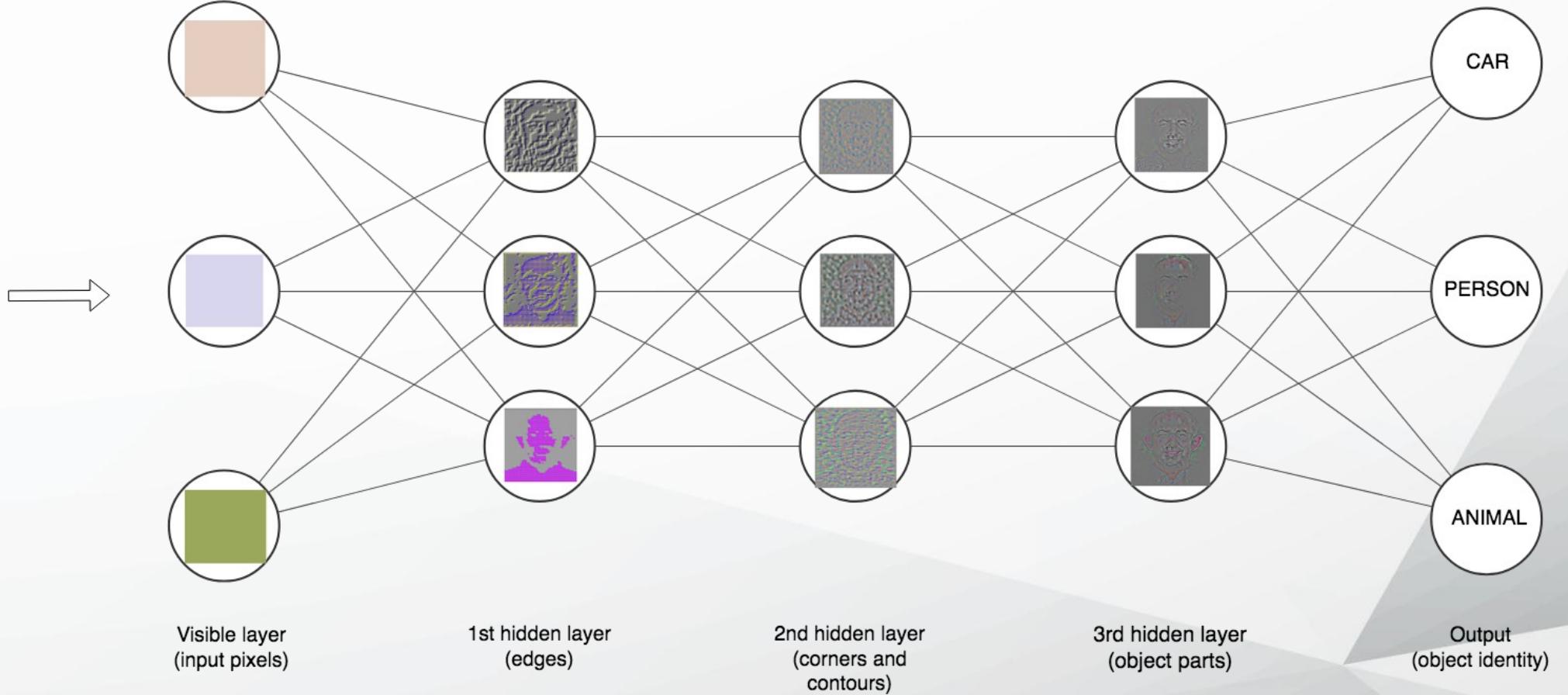


# Representation

- First - Representation matters!



# Depth – repeated compositions

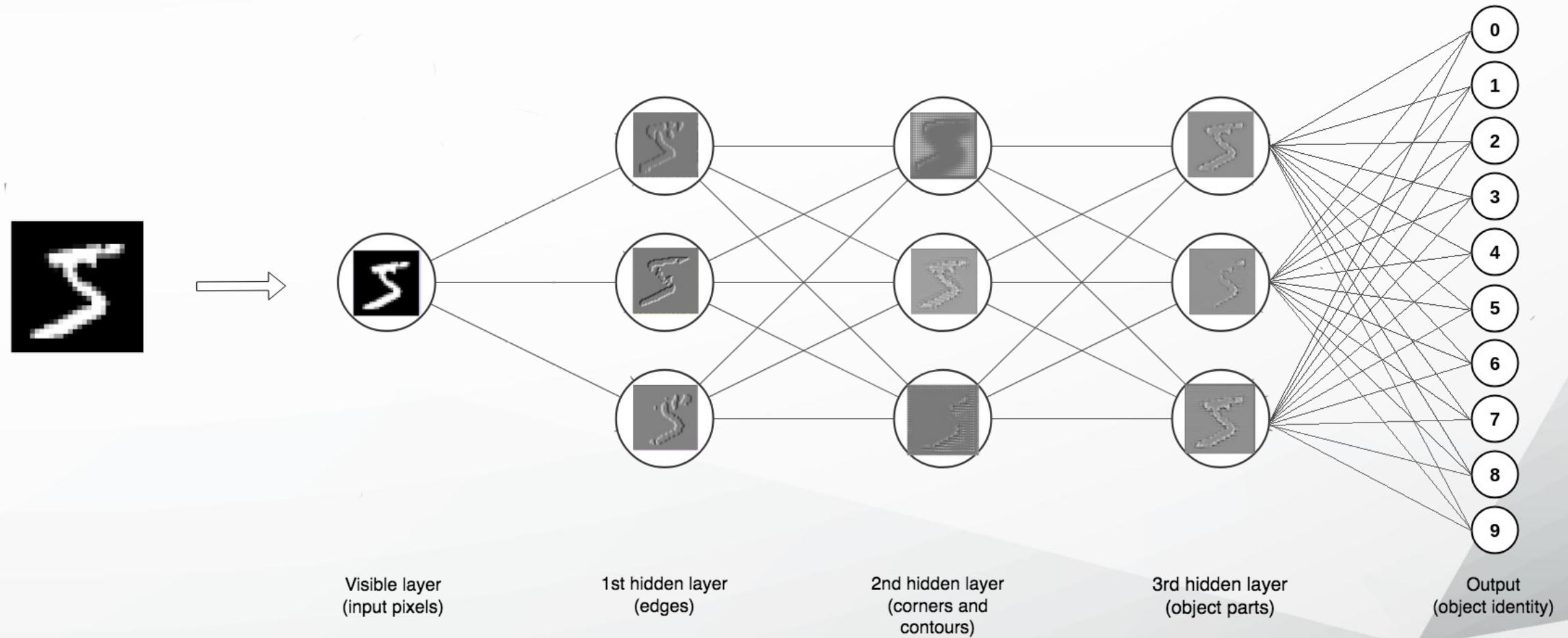


# Depth – repeated compositions

MNIST – dataset of hand written digits



# Depth – repeated compositions



# Common types of Neural Network Architectures

- Feedforward Neural networks
- Convolutional Neural Networks
- Recurrent Neural Networks
- Long Short-term Memory networks
- Autoencoders
- Generative Adversarial Networks

# Feed-forward Neural Networks

- Simplest form of ANN:
  - the perceptrons are arranged in layers
    - the first layer is taking the inputs
    - the last layer is producing the outputs
    - between them there are hidden layers
- The data flow goes in one direction:
  - each perceptron is connected with every perceptron on the next layer
  - there is no connection between the perceptrons in the same layer

# Features of feed-forward ANNs

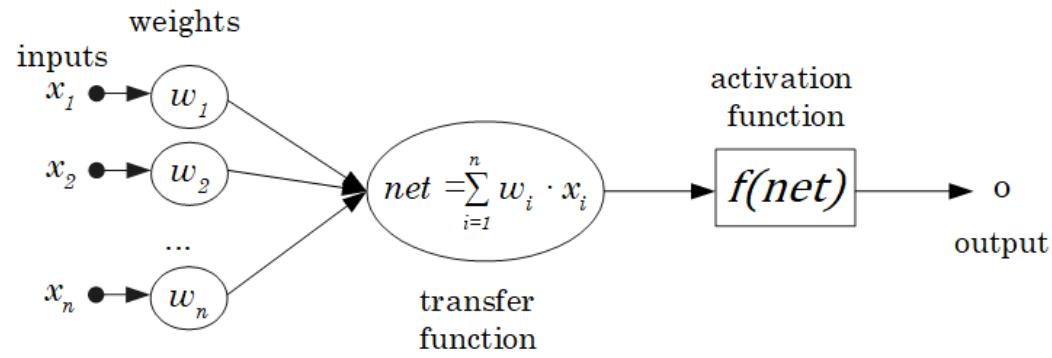
- recall that a single perceptron can classify points into two regions that are linearly separable

Consider a more complex network

- with more perceptrons that are independent of each other in the hidden layer, the points are classified in more pairs of linearly separable regions, each of it having a unique line separating the region.
- by varying the number of nodes in the hidden layer, the number of layers, and the number of input and output nodes, one can classification of points in arbitrary dimension into an arbitrary number of groups
- hence feed-forward networks are commonly used for classification.

# Artificial neural network – structure

## a node's structure



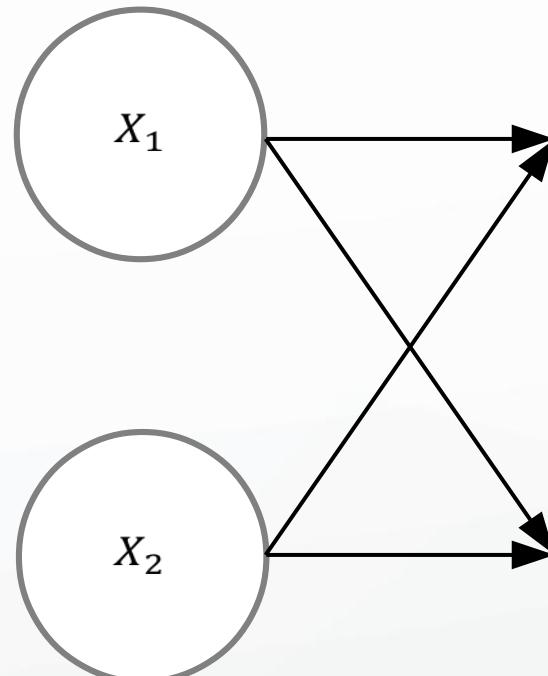
$$\mathbf{X} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

$$\mathbf{W} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix}$$

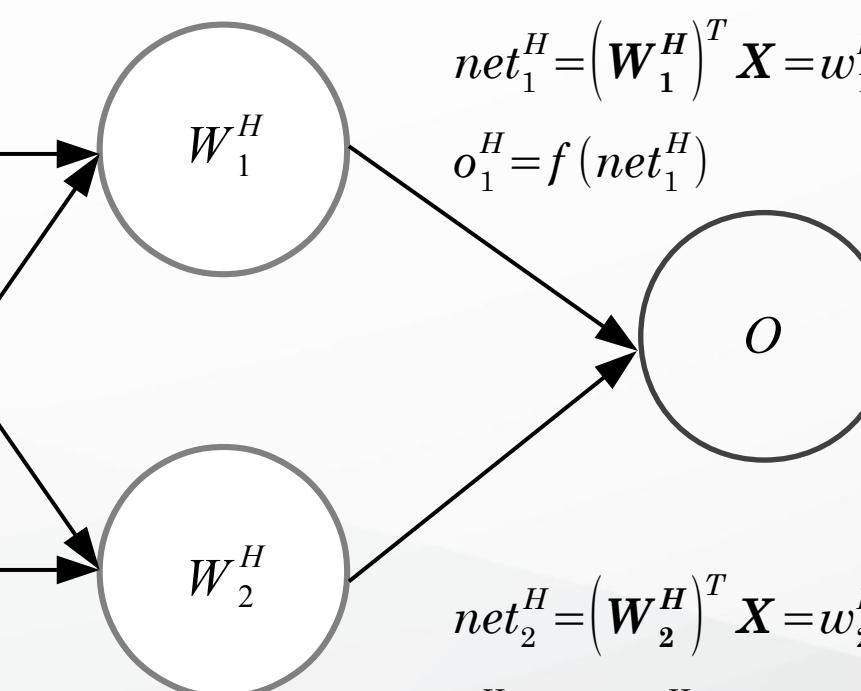
$$\mathbf{o} = f(\mathbf{X}^T \mathbf{W})$$

# Artificial neural network – structure

Input layer



Hidden layer



Output layer

$$\begin{aligned}net_1^H &= (\mathbf{W}_1^H)^T \mathbf{X} = w_{11}^H x_1 + w_{12}^H x_2 \\o_1^H &= f(net_1^H) \\net_2^H &= (\mathbf{W}_2^H)^T \mathbf{X} = w_{21}^H x_1 + w_{22}^H x_2 \\o_2^H &= f(net_2^H)\end{aligned}$$

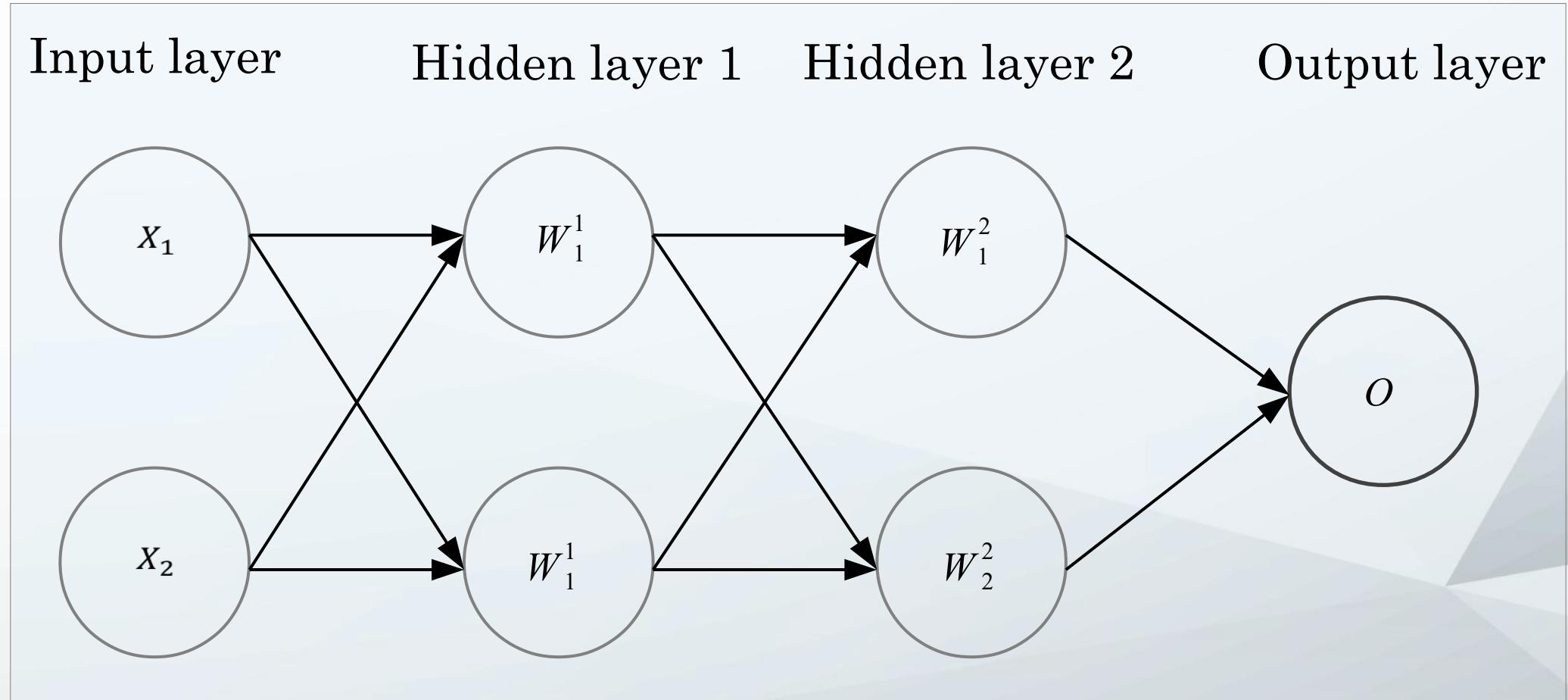
**Output function**

$O = g(o_1^H, o_2^H) \longrightarrow Error = E(O, t)$

**Loss function**

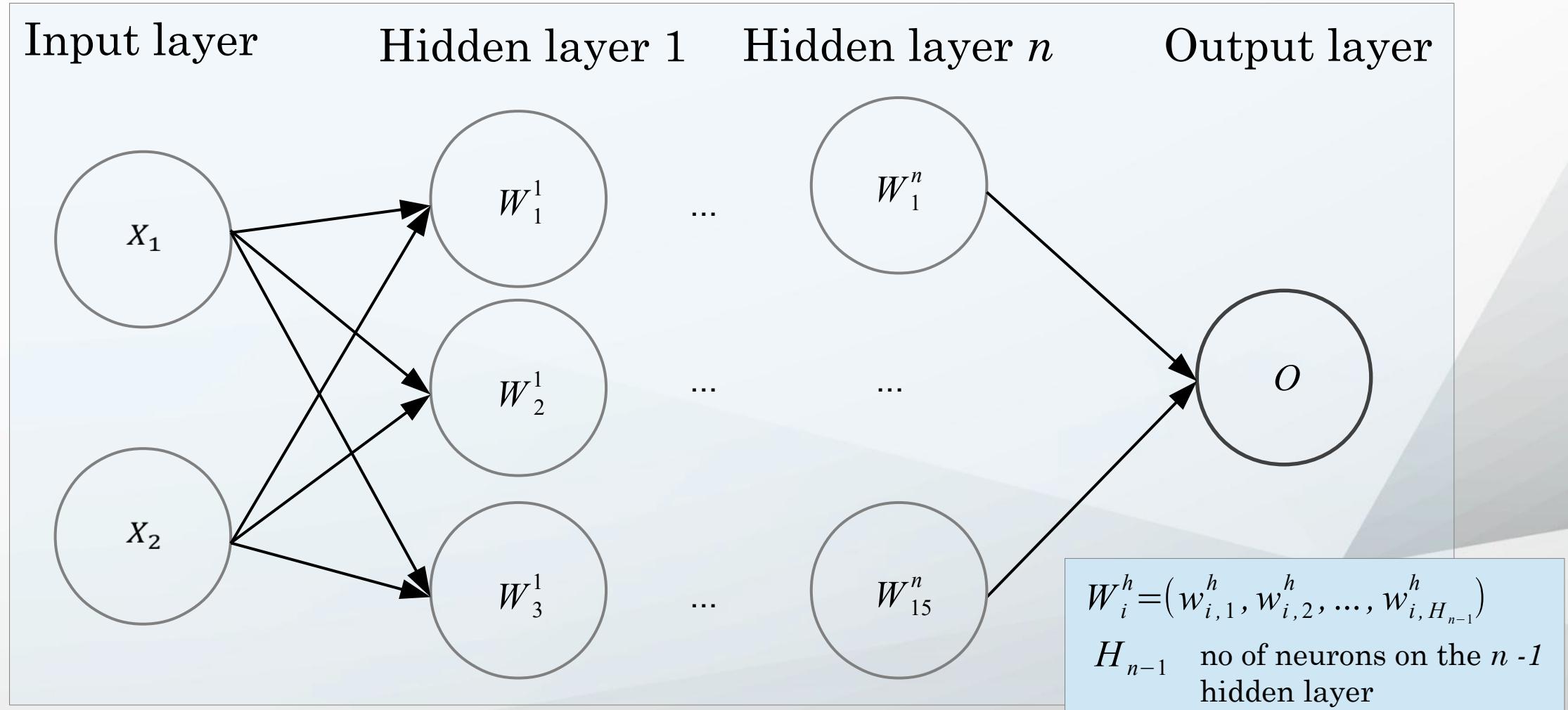
# Artificial neural network – structure

Example of a **2 : 2 : 2 : 1** architecture with 2 hidden layers



# Artificial neural network – structure

Example of a **2:3: ... :15:1** architecture with  $n$  hidden layers



Design choices for feed-forward ANNs

**Activation function**

**Loss function**

**Output units**

**Architecture**

# Linearity versus non-linearity

- Linear models:
  - Can be fit efficiently (via convex optimization)
  - Limited model capacity

- Alternative:

$$f(\mathbf{x}) = \mathbf{W}^T \phi(\mathbf{x})$$

- Where  $\phi(\mathbf{x})$  is a *non-linear transform*

# Activation functions for ANNs

The activation function should:

- Provide **non-linearity**
- Ensure **gradients remain large** through hidden unit

Common choices are

- Sigmoid
- Relu, leaky ReLU, Generalized ReLU, MaxOut
- Softplus
- Tanh
- Swish

# Traditional ANNs (like feed-forward)

- Manually engineer
  - Domain specific, enormous human effort
- Generic transform
  - Maps to a higher-dimensional space
  - Kernel methods: e.g. RBF kernels
  - Over fitting: does not generalize well to test set
  - Cannot encode enough prior information

# Deep Learning

- Directly learn  $\phi$

$$f(\mathbf{x}, \eta) = W^T \phi(\mathbf{x}, \eta)$$

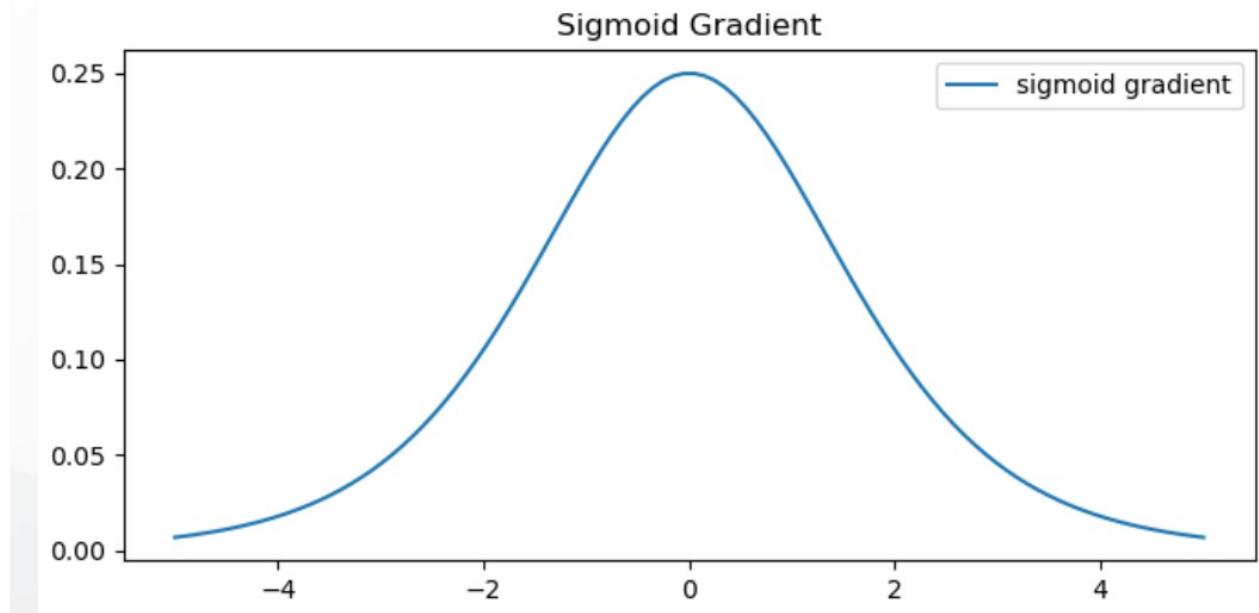
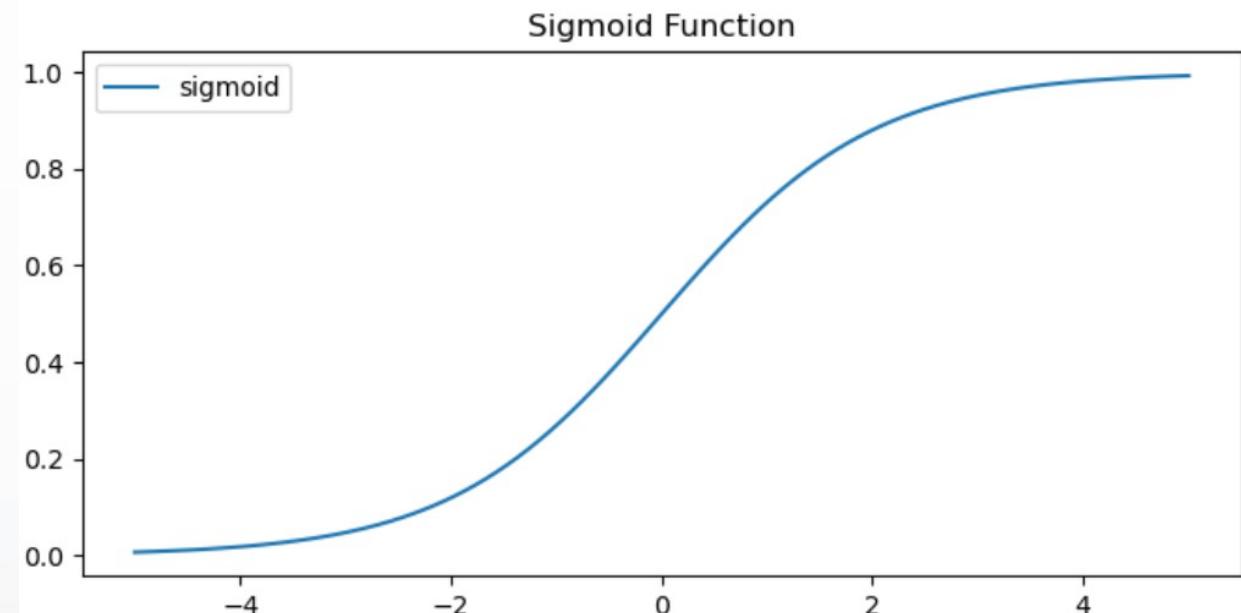
- $\phi(\mathbf{x}, \eta)$  is an automatically-learned **representation** of  $x$
- For **deep networks**,  $\phi$  is the function learned by the **hidden layers** of the network
- $\eta$  are the learned weights

Non-convex optimization

- Can encode prior beliefs, generalizes well

# Sigmoid (aka Logistic)

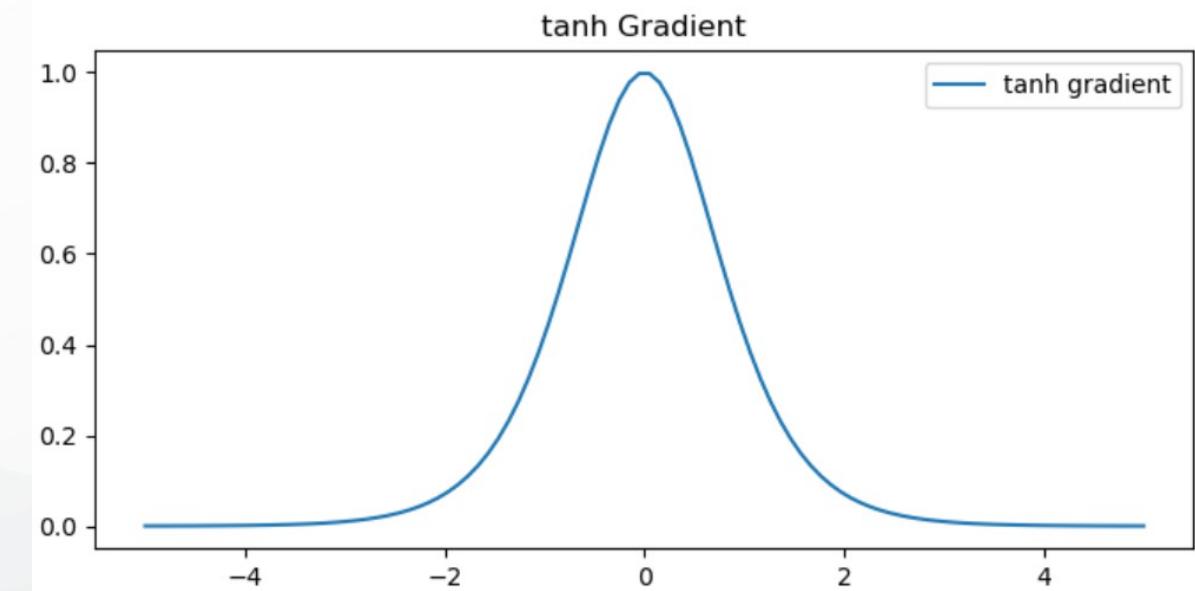
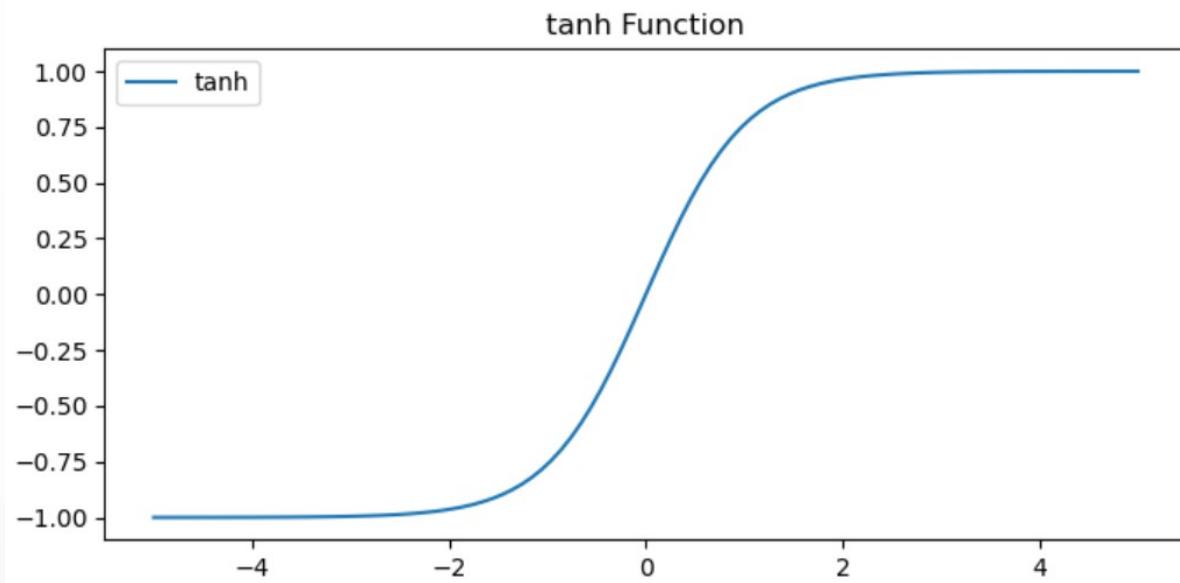
$$y = \frac{1}{1 + e^{-x}}$$



Derivative is **zero** for much of the domain. This leads to “vanishing gradients” in backpropagation.

# Hyperbolic tangent (aka tanh)

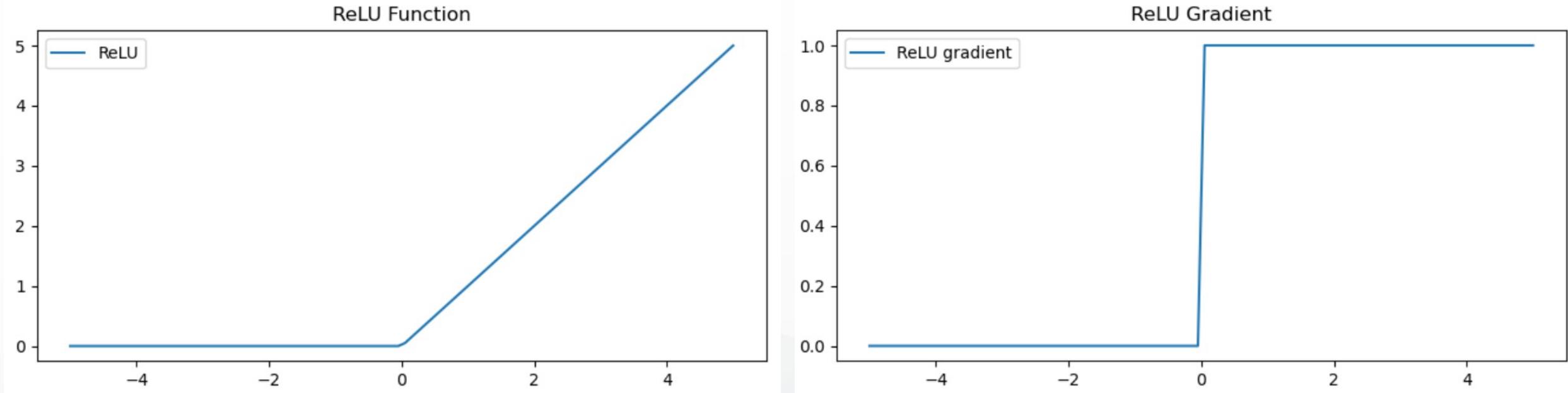
$$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



Derivative is also **zero** for much of the domain.

# Rectified Linear Unit (ReLU)

$$y = \max(0, x)$$



**Two major advantages:**

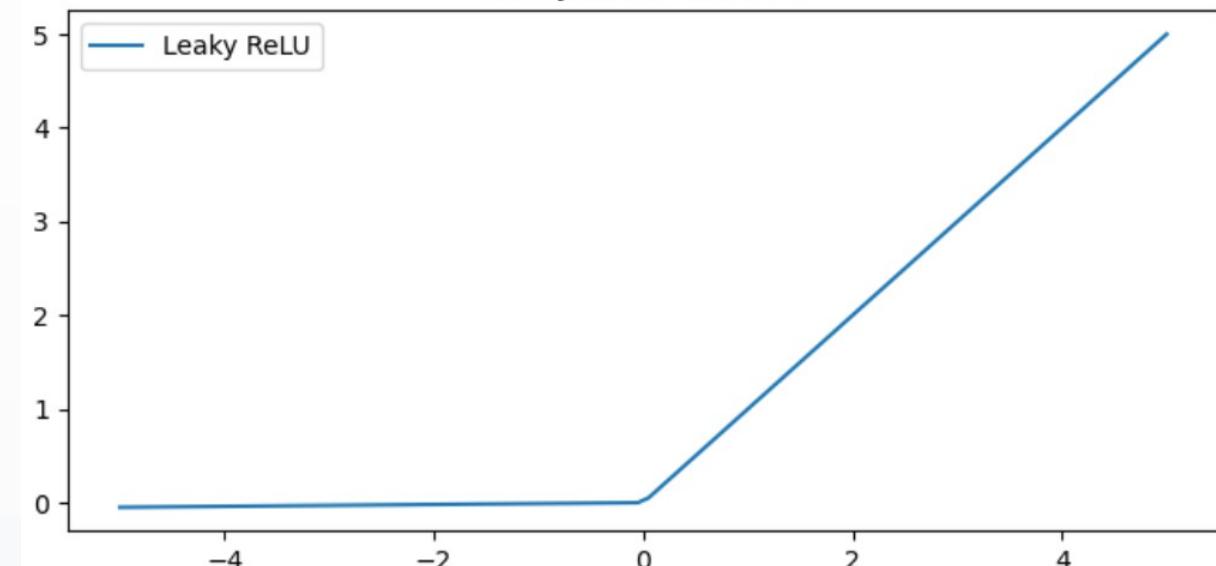
1. No vanishing gradient when  $x > 0$
2. Provides sparsity (regularization) since  $y = 0$  when  $x < 0$

# Leaky ReLU

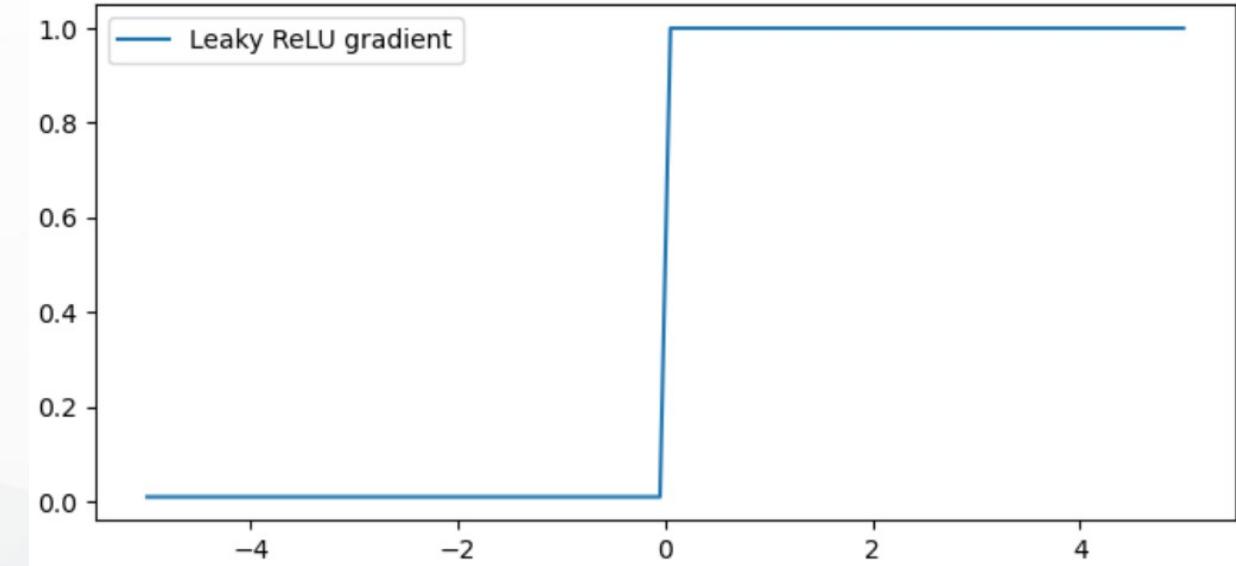
$$y = \max(0, x) + \min(\alpha x, 0)$$

$\alpha$  is a small positive value

Leaky ReLU Function



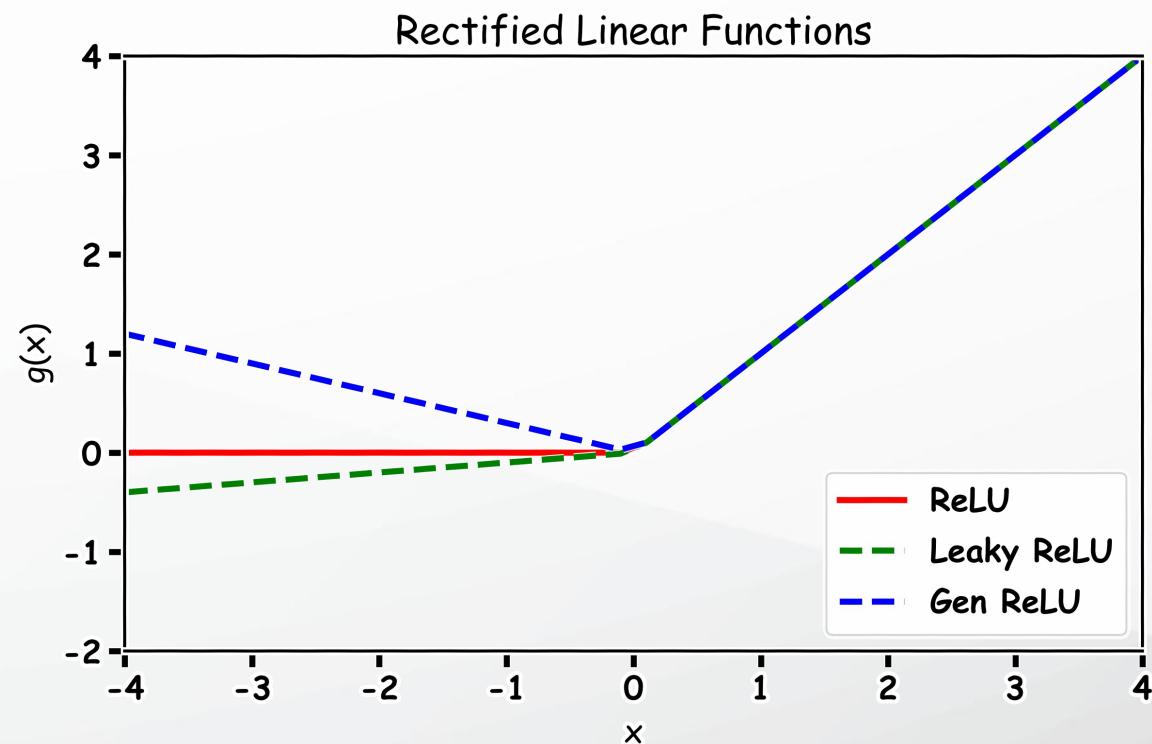
Leaky ReLU Gradient



- Tries to fix “dying ReLU” problem: derivative is non-zero everywhere.
- Some people report success with this form of activation function, but the results are not always consistent

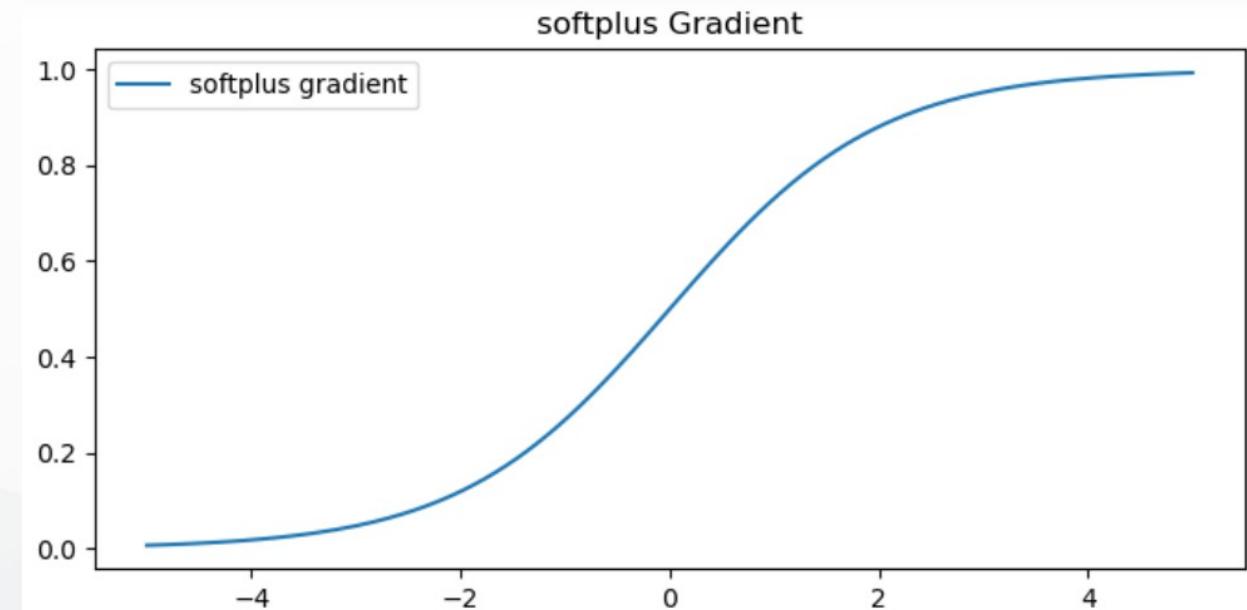
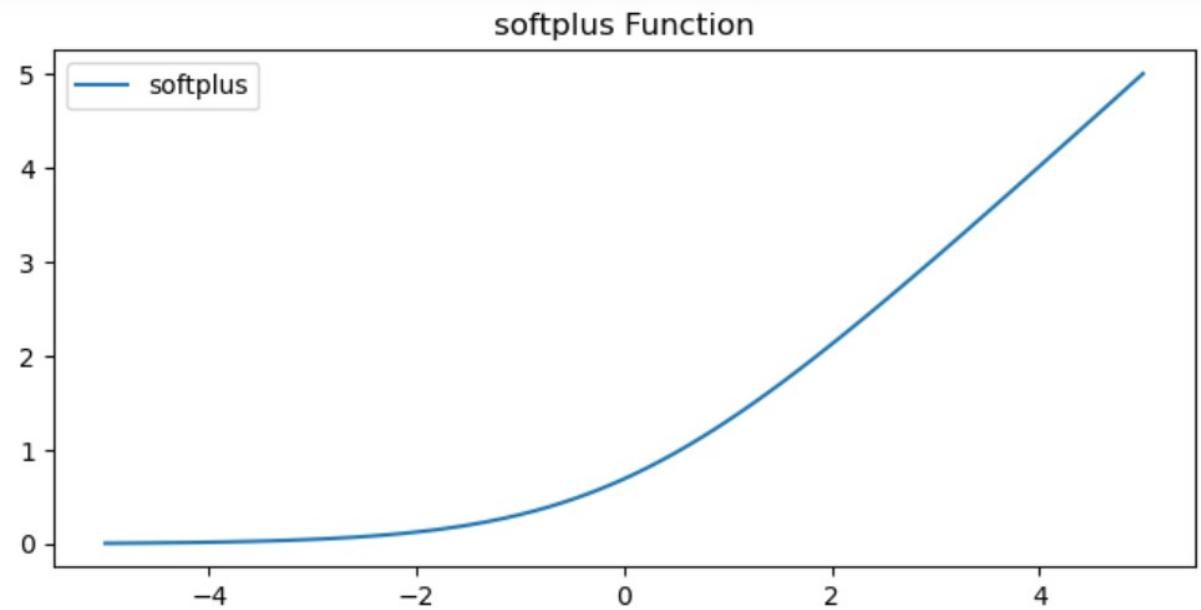
# Generalized ReLU

$$y = \max(0, x) + \alpha \min(x, 0)$$



# Softplus function

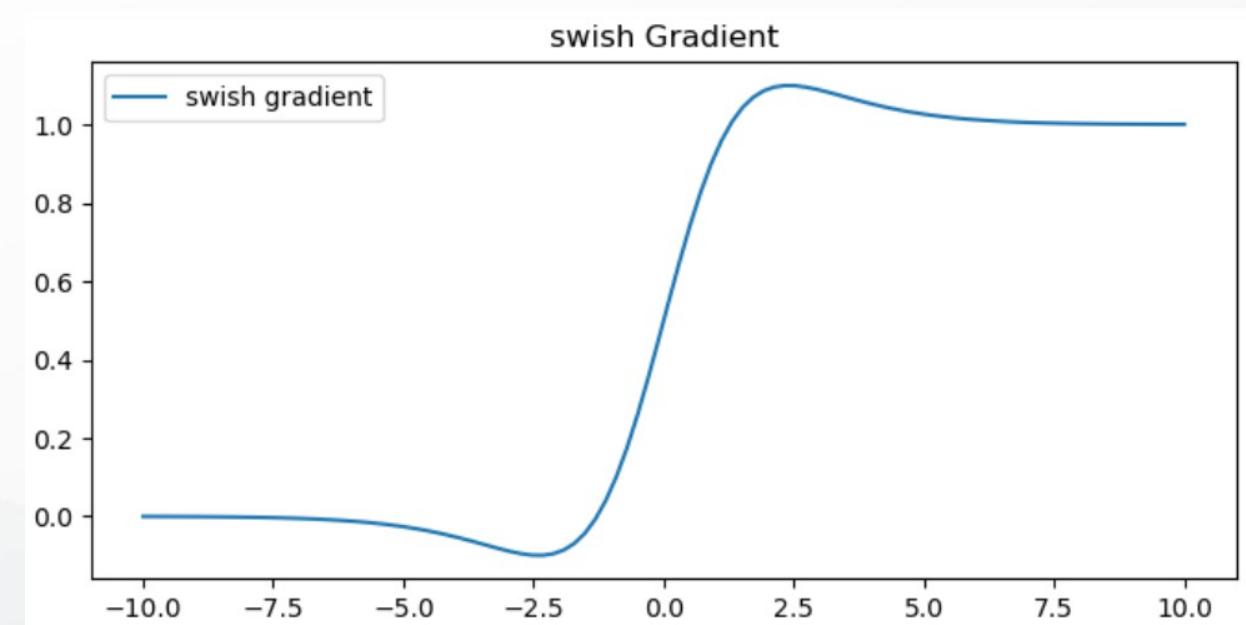
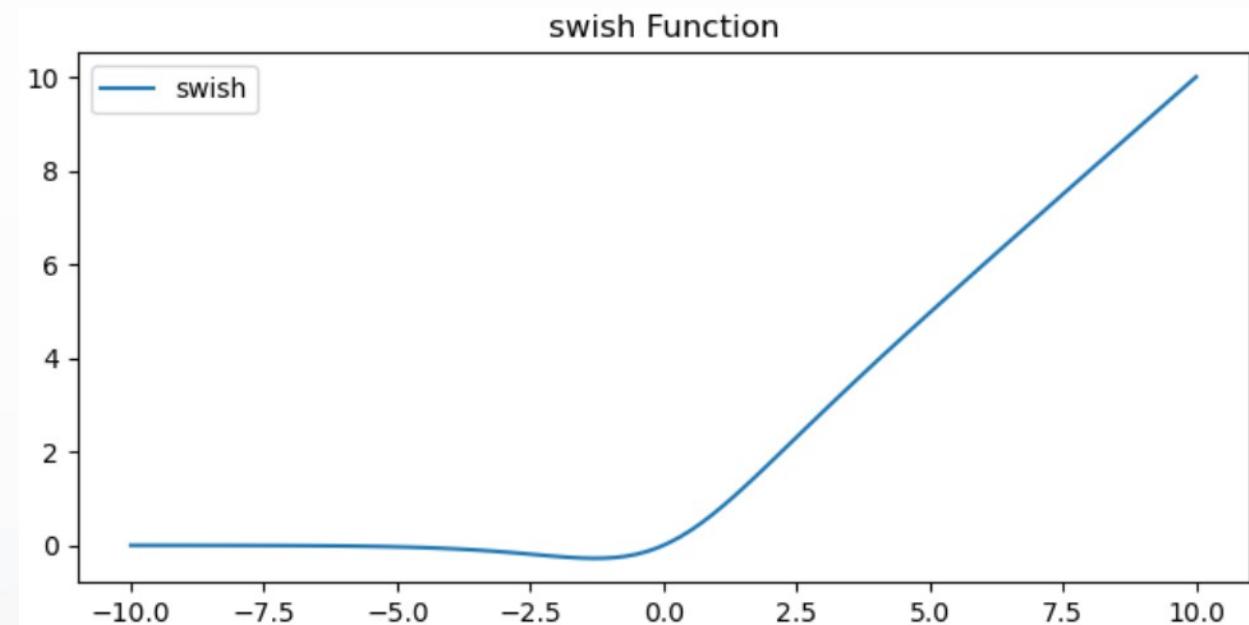
$$y = \log(1 + e^x)$$



Outputs produced by sigmoid and tanh functions have upper and lower limits whereas softplus function produces outputs in scale of  $(0, +\infty)$ .

# Swish function

$$y = x \operatorname{sigmoid}(x)$$



# Loss functions for ANNs

- Likelihood for a given point
- Assume independence – likelihood for all measurements
- Maximize the likelihood, or equivalently maximize the log-likelihood
- Turn all this into a loss function

# Common loss functions for ANNs

Mean Absolute Error Loss

Mean Squared Error Loss

Negative Log-Likelihood Loss

Cross-Entropy Loss

Hinge Embedding Loss

...

# Mean Absolute Error Loss (MAE)

also called L1 Loss

- computes the average of the sum of absolute differences between actual values and predicted values.

$$\text{loss}(x, y) = |x - y|$$

- $x$  represents the actual value and  $y$  the predicted value.

used for:

- regression problems
  - especially when the distribution of the target variable has outliers, such as small or big values that are a great distance from the mean value.

# Mean Squared Error Loss (MSE)

also called L2 Loss

- computes the average of the squared differences between actual values and predicted values

$$\text{loss}(x, y) = (x - y)^2$$

- $x$  represents the actual value and  $y$  the predicted value.

MSE is the default loss function for most regression problems.

# Cross-Entropy Loss

Is the difference between two probability distributions for a provided set of occurrences or random variables.

In the discrete setting, given two probability distributions  $p$  and  $q$ , their cross-entropy is defined as

$$H(p, q) = - \sum_{x \in X} p(x) \log(q(x))$$

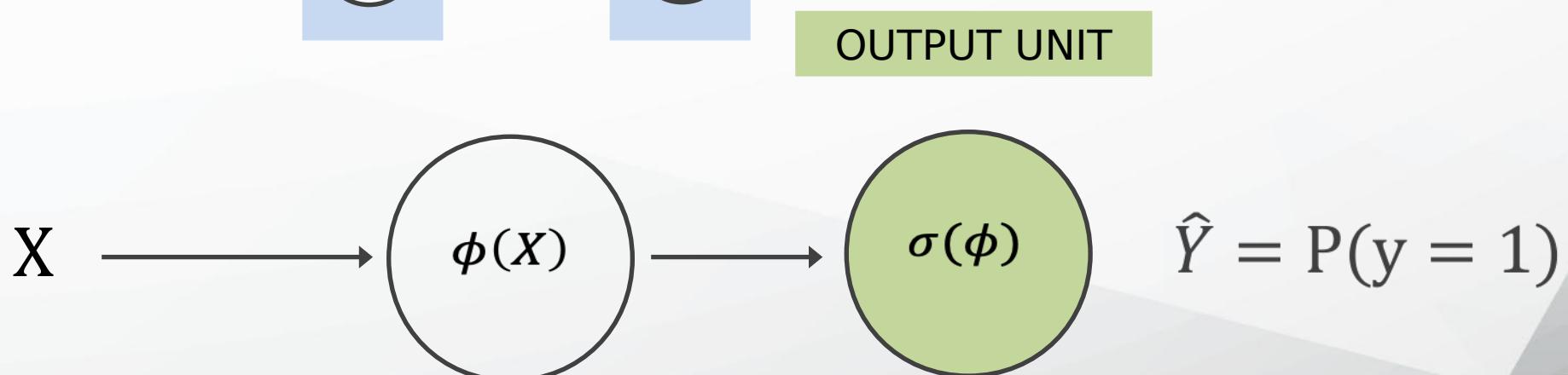
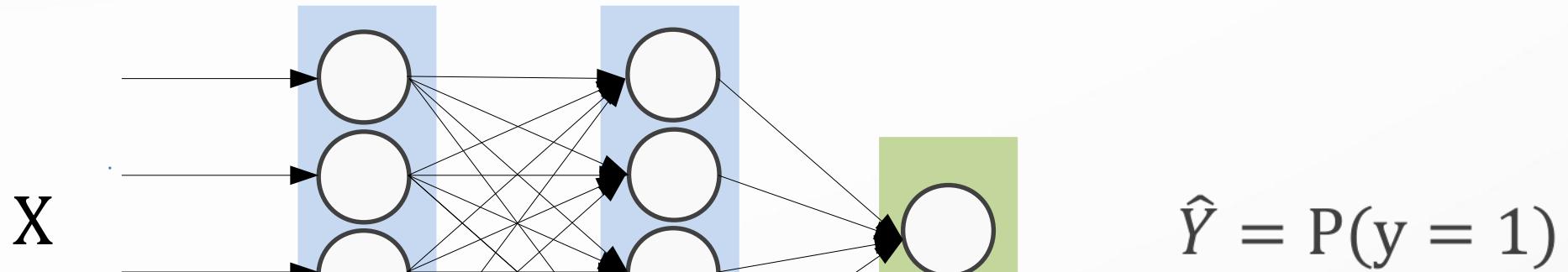
To compute the loss we apply the cross-entropy operator between the desired output and to the real output of our model after we used the softmax operator on the output.

Used in classifications

# Output functions

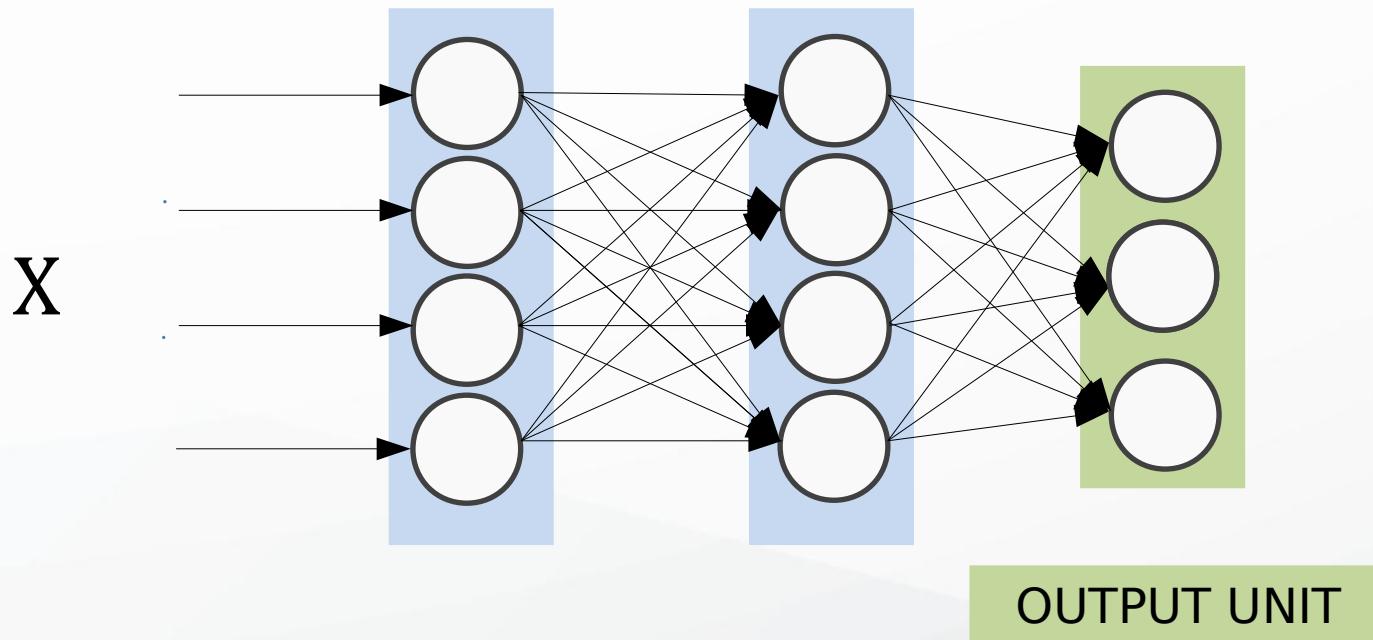
Output Type	Output Distribution	Output layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary Cross Entropy
Discrete	Multinoulli	Softmax	Cross Entropy
Continuous	Gaussian	Linear	MSE
Continuous	Arbitrary	-	GANS

# Output unit for binary classification



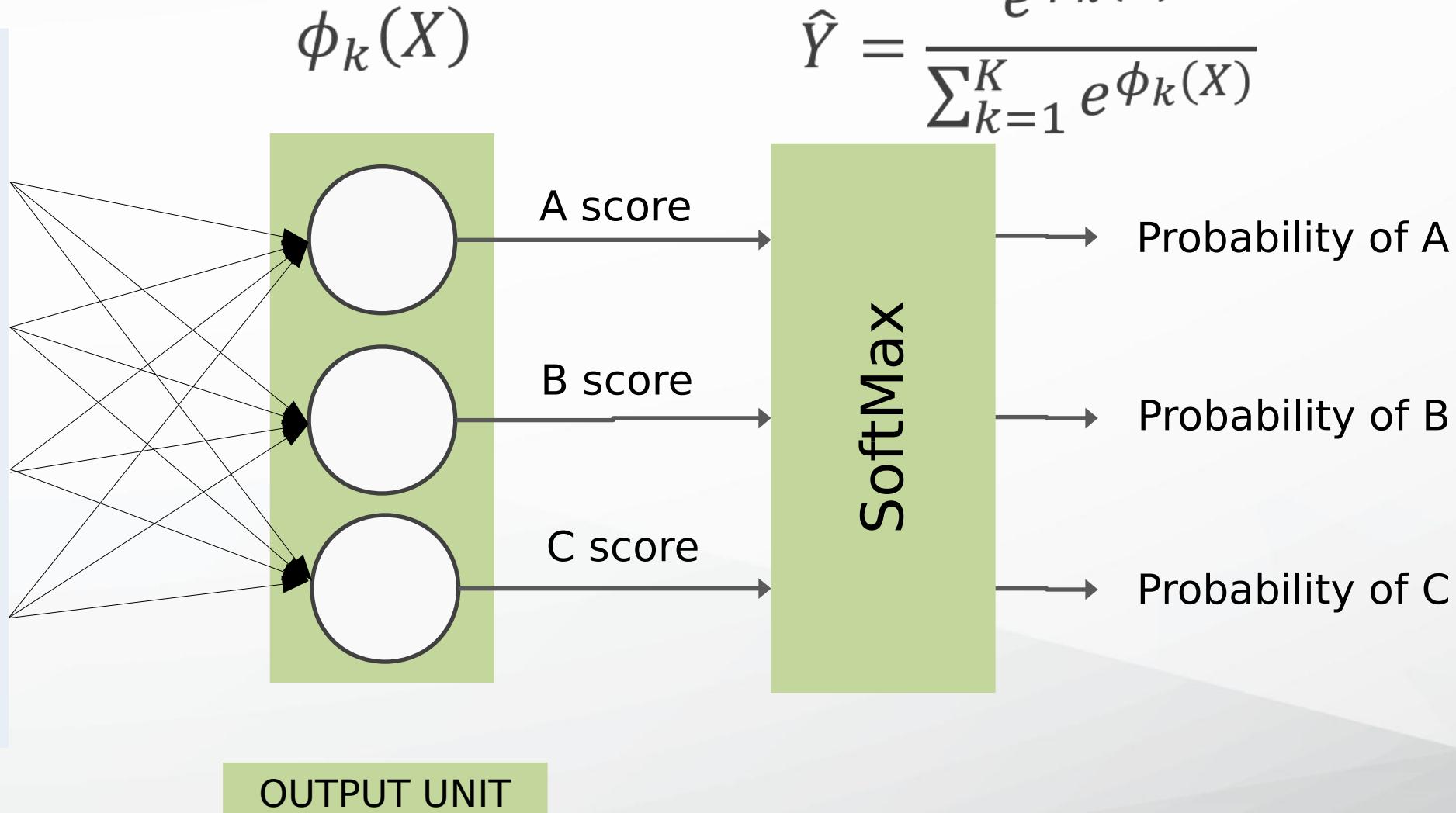
$$X \Rightarrow \phi(X) \Rightarrow P(y = 1) = \frac{1}{1 + e^{-\phi(X)}}$$

# Output unit for multi class classification



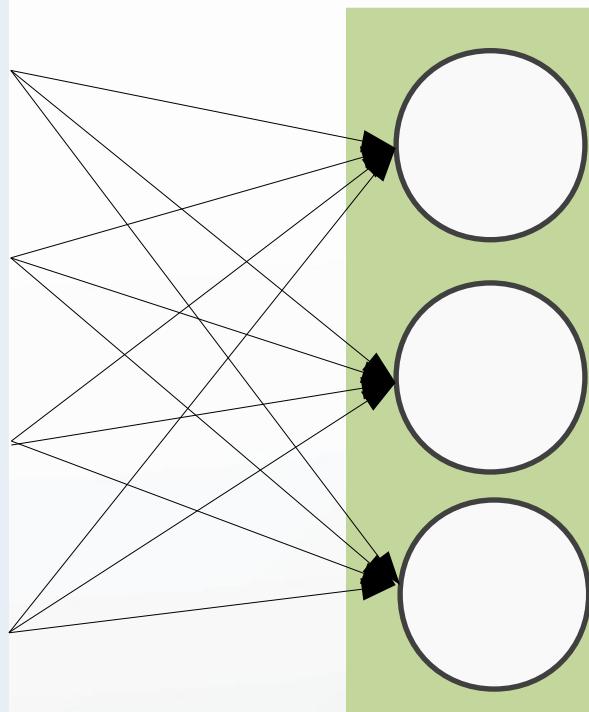
# Softmax

rest of the network



# Softmax

rest of the network



$$\phi_k(X)$$

$$\hat{Y} = \frac{e^{\phi_k(X)}}{\sum_{k=1}^K e^{\phi_k(X)}}$$

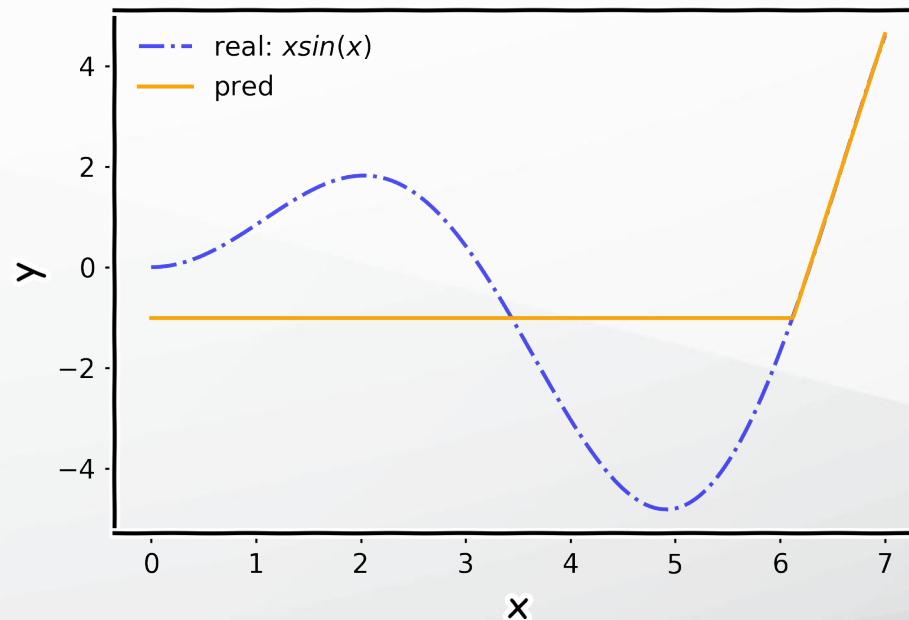
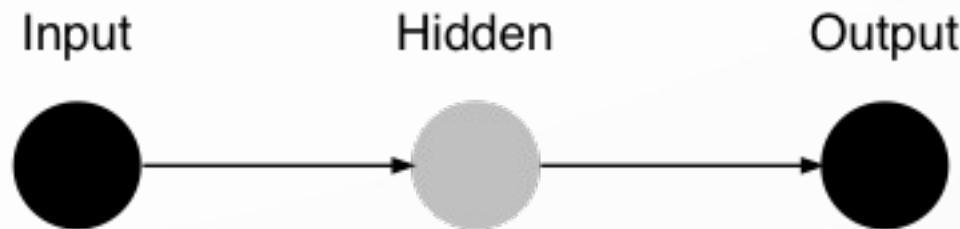
→ Probability of A

→ Probability of B

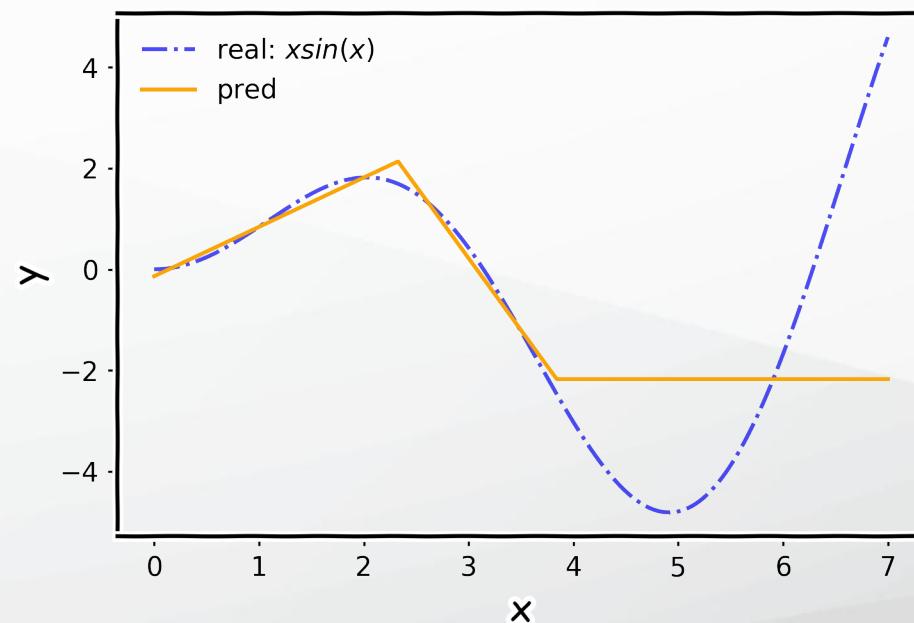
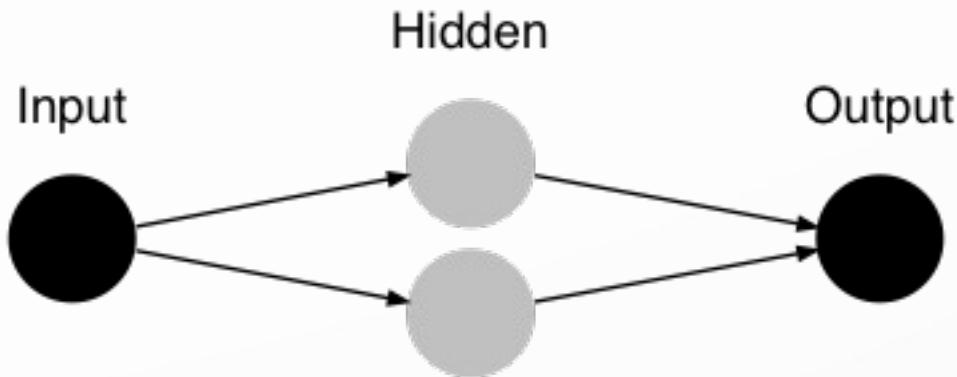
→ Probability of C

OUTPUT UNIT

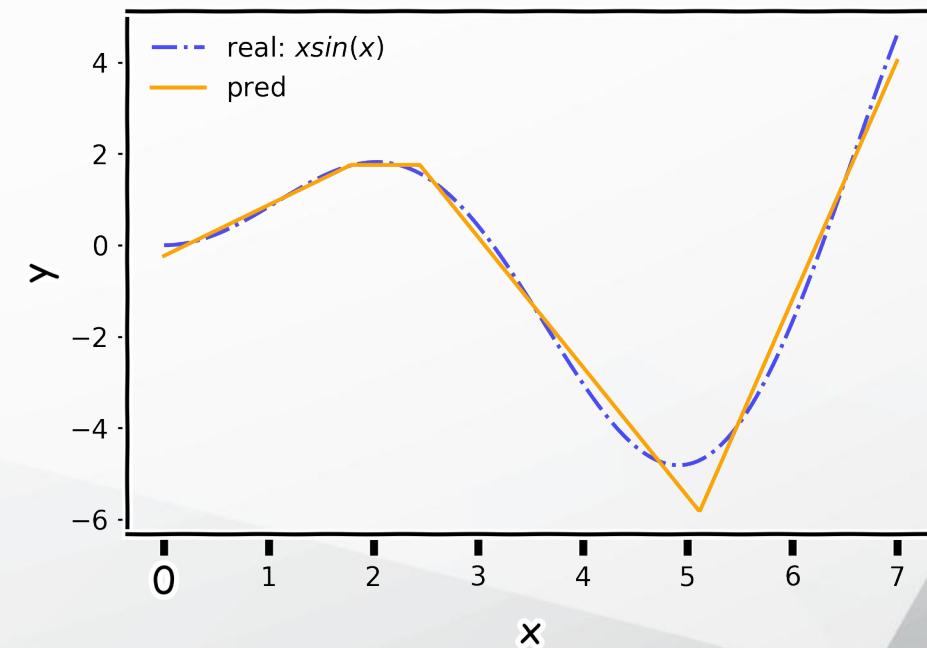
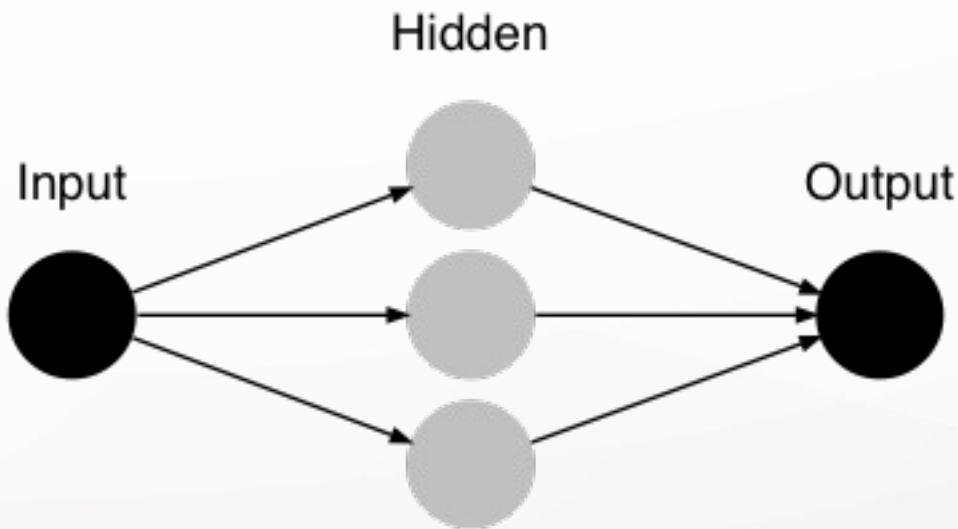
# Architecture – performance example



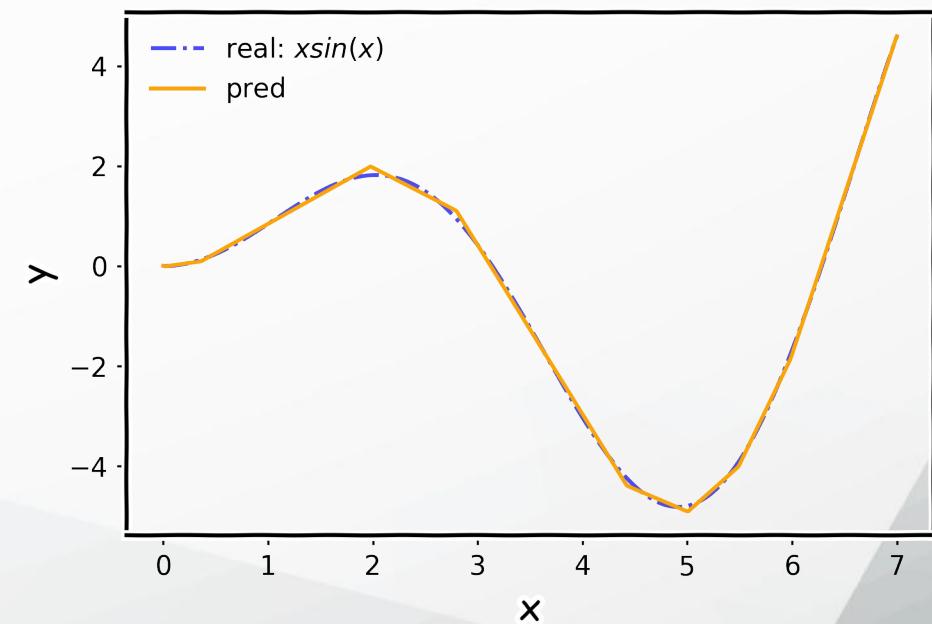
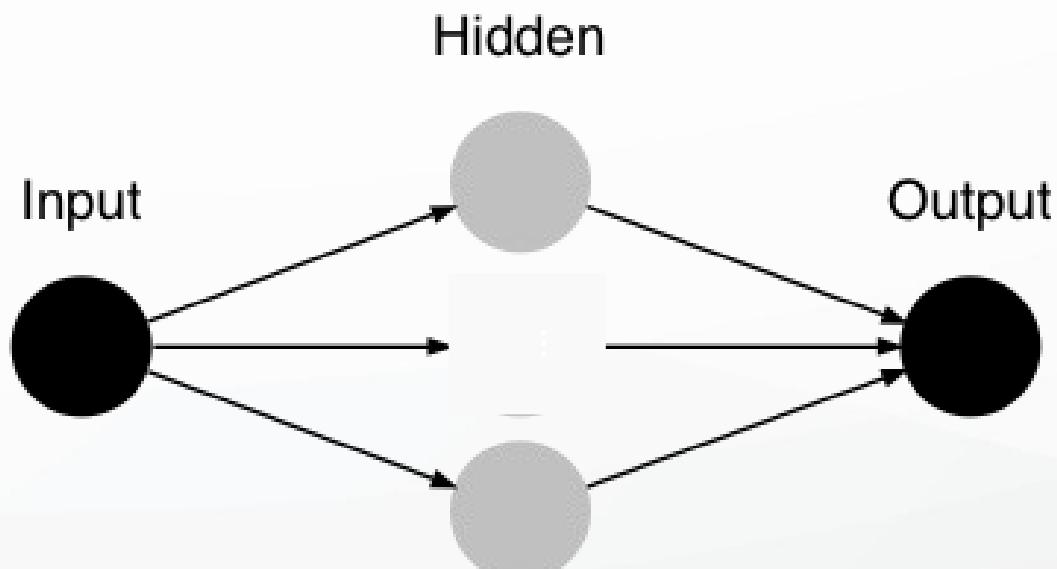
# Architecture – performance example



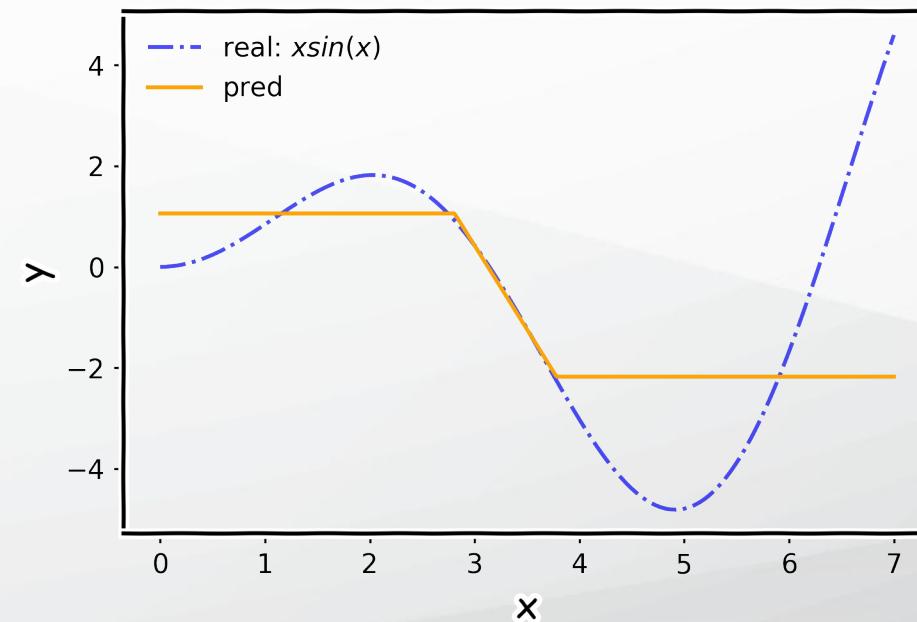
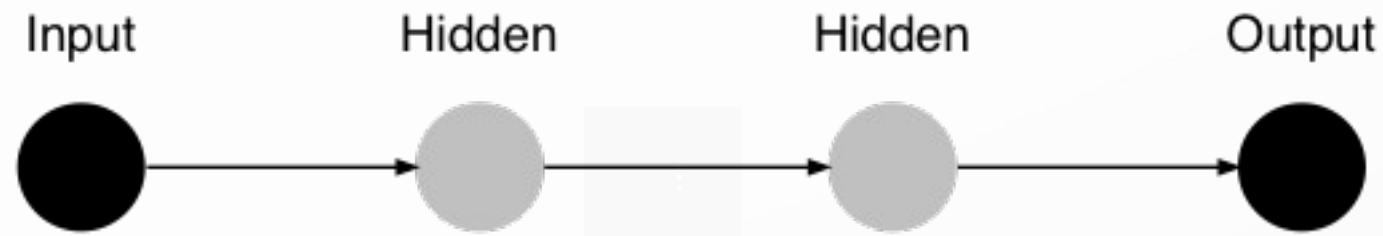
# Architecture – performance example



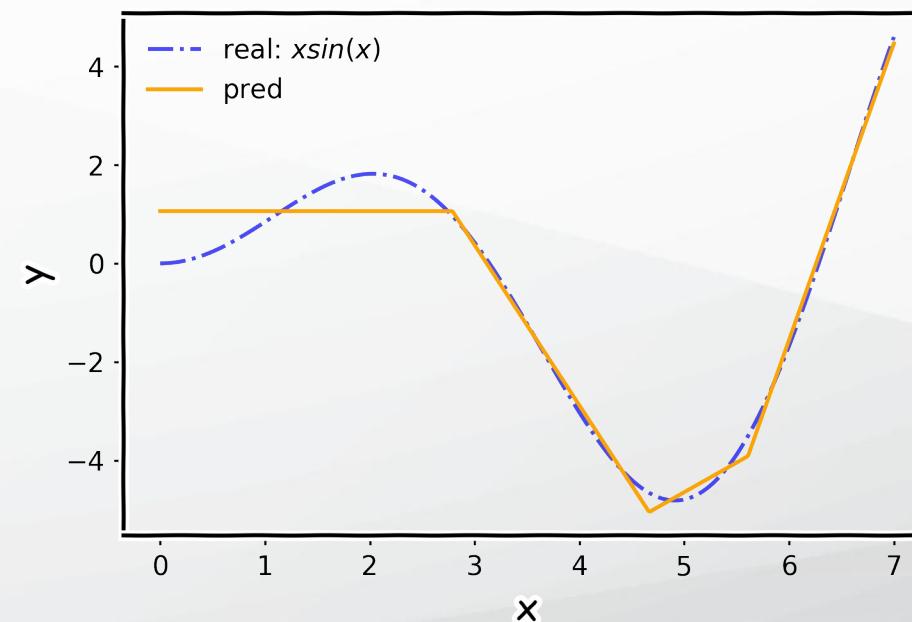
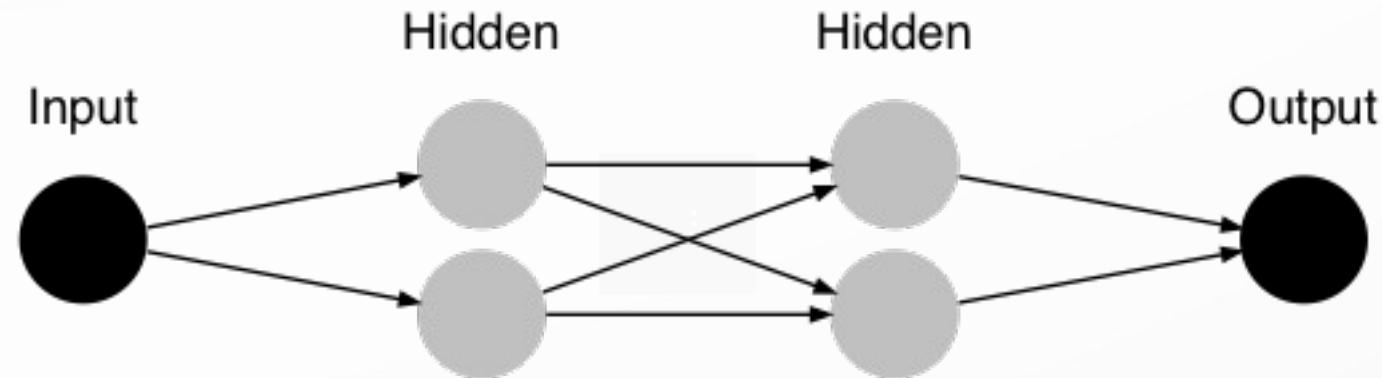
# Architecture – performance example



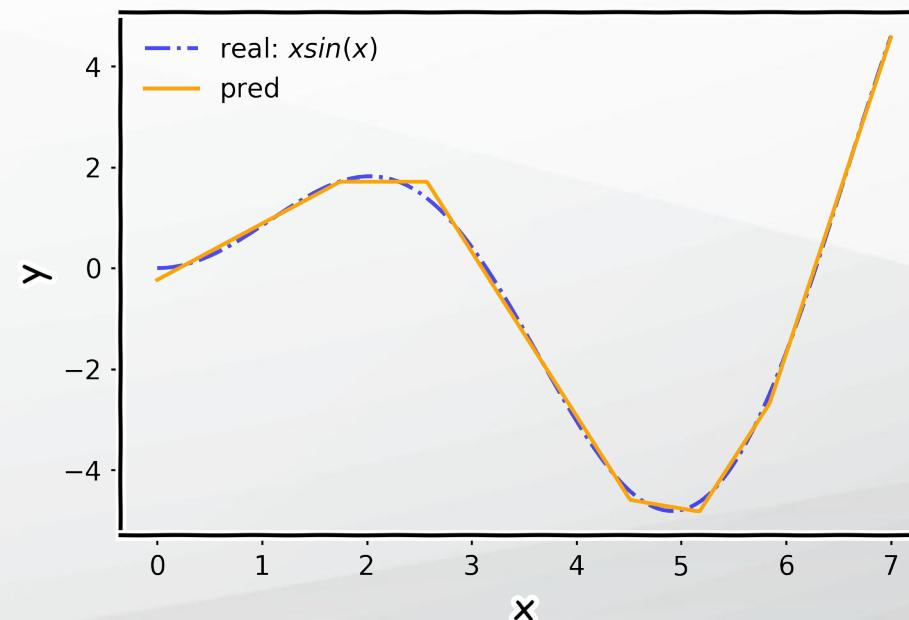
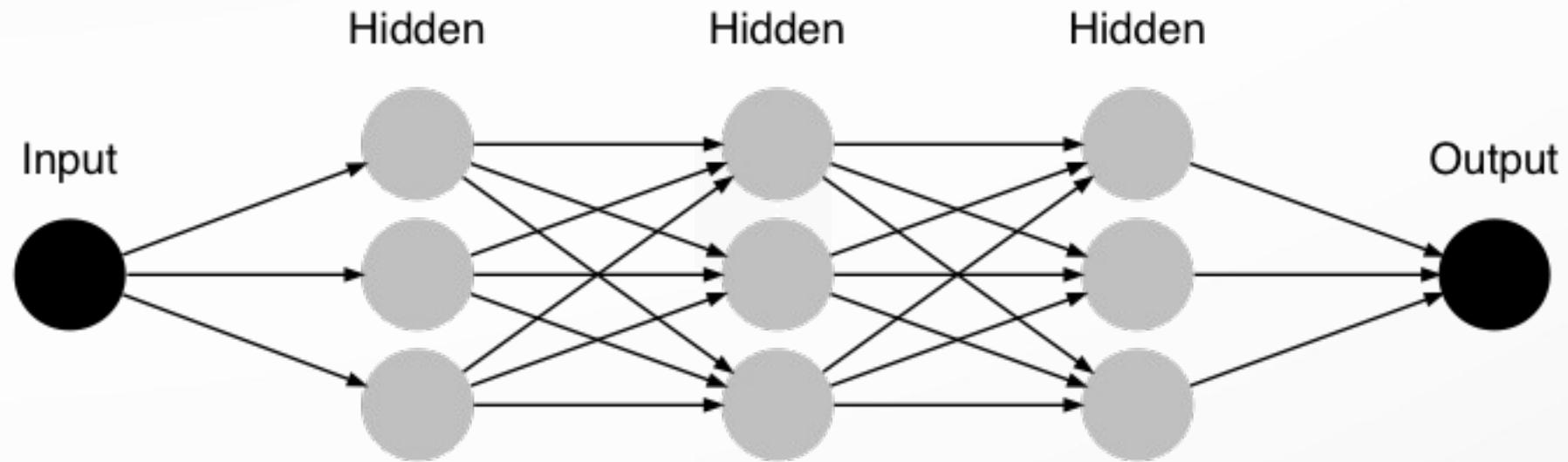
# Architecture – performance example



# Architecture – performance example



# Architecture – performance example



# Universal Approximation Theorem

**Think of a Neural Network as function approximation.**

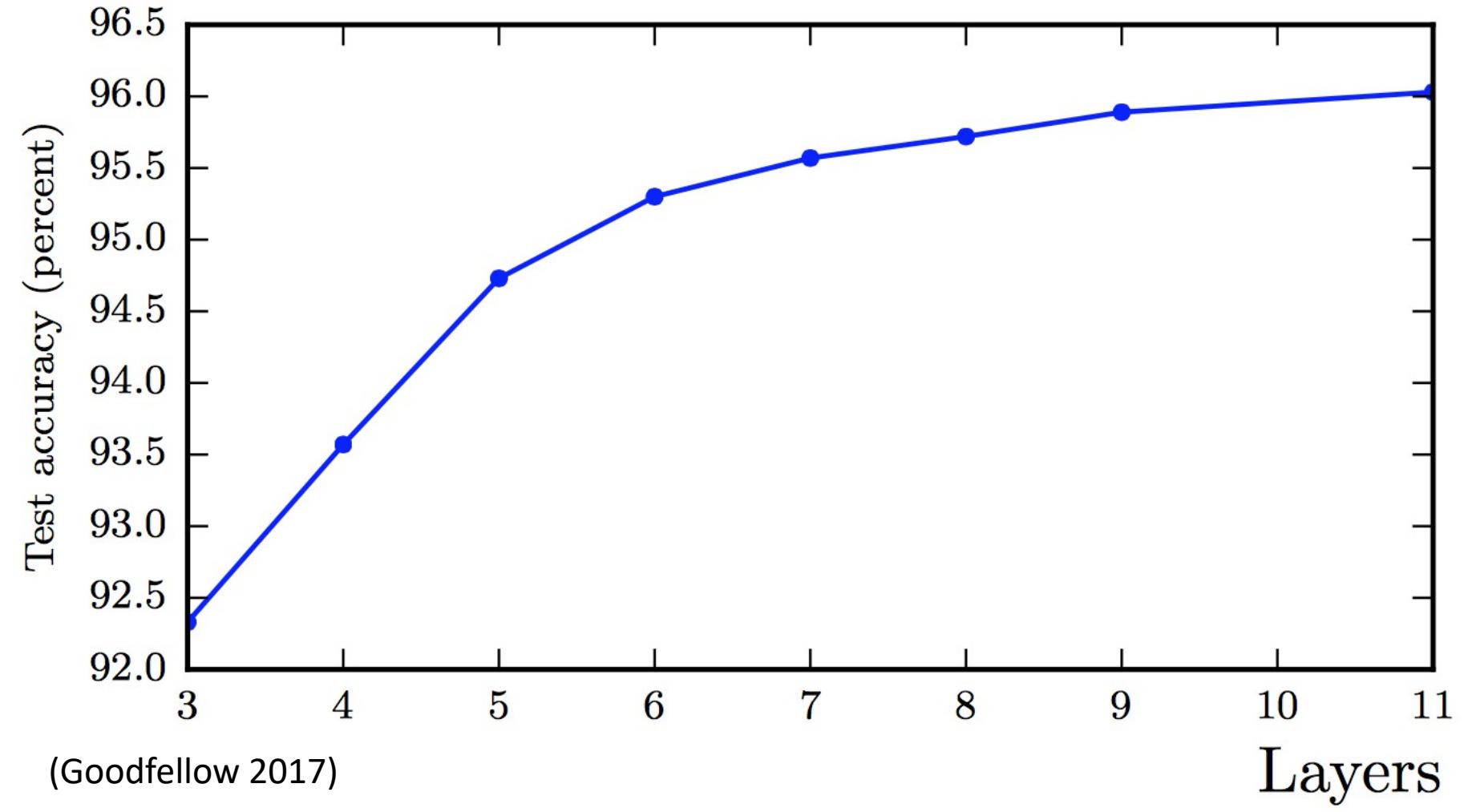
- NN:

One hidden layer is enough to *represent* an approximation of any function to an arbitrary degree of accuracy

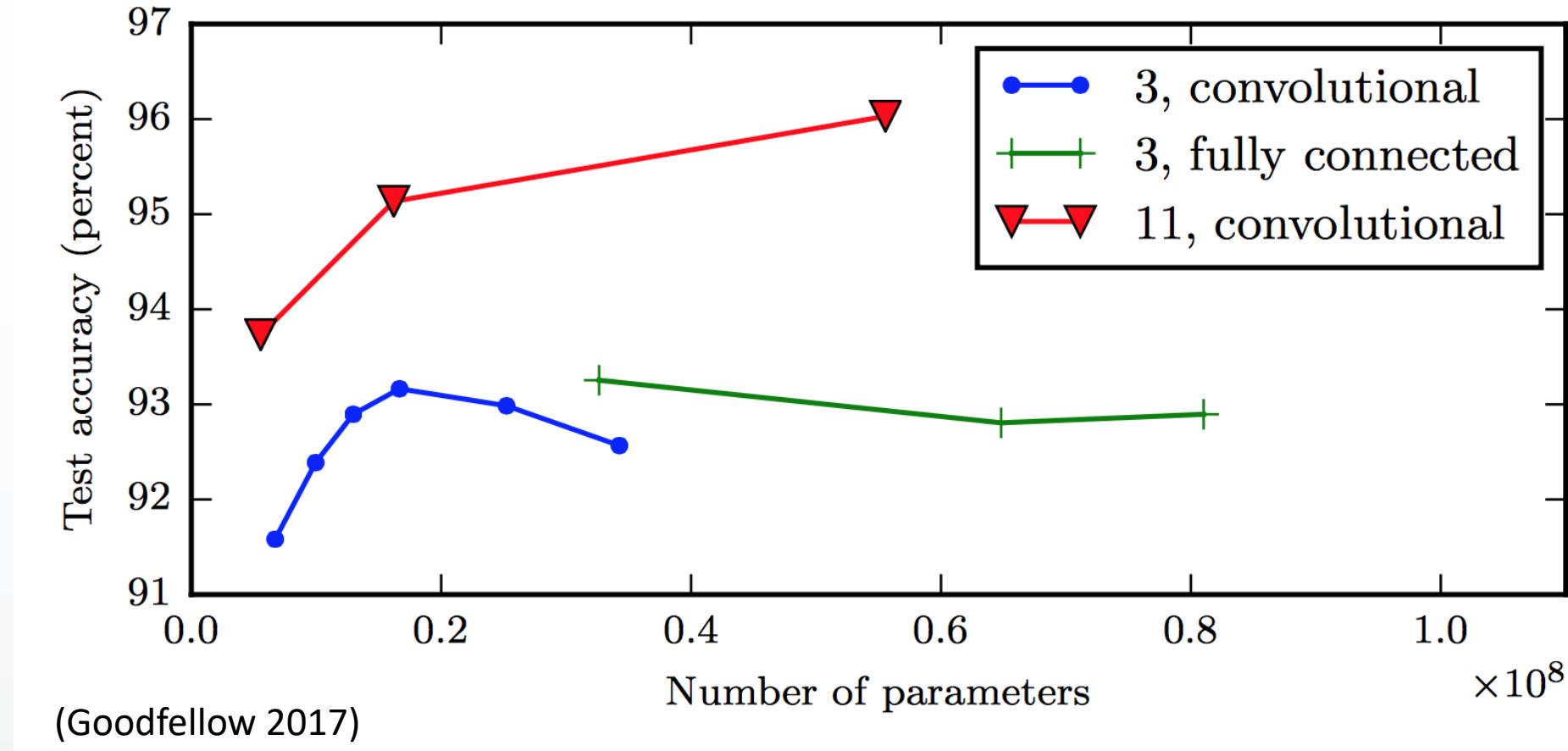
- So why deeper?

- Shallow net may need (exponentially) more width
  - Shallow net may overfit more

# Better generalization with depth



# Overfitting in shallow nets



The **11-layer net** generalizes better on the test set when controlling for number of parameters.

The 3-layer nets perform worse on the test set, even with similar number of total parameters.

# Training a feed-forward ANN

Initialisation of network weights  $w_{1,1}^1, w_{1,2}^1, w_{1,3}^1, \dots, w_{1,1}^n, \dots, w_{H_{n-1}, H_n}^n, w_{1,1}^O, w_{1,2}^O, \dots, w_{r, H_n}^O$

While not stop condition

For each train example  $(x^d, t^d)$ , where  $d = 1, 2, \dots, n$

Activate the network and determine the output  $o^d$ :

forward propagate the information and determine the output of each neuron

Modify the weights:

establish and backward propagate the error:

- establish the errors of neurons from the output layer  $e_r, r=1, \dots, m$
- backward propagate the errors in the entire network → distribute the errors on all connections of the network
- modify the weights

EndWhile

# Back-propagation of errors in an ANN

- One of the first algorithms for fine tuning of the weights in an ANN
- Very popular

## **Advantages:**

- A gradient descent method
- does not require normalization of input vectors (normalization could improve performance)

## **Limitations:**

- is not guaranteed to find the global minimum of the error function, but only a local minimum
- it has trouble crossing plateaus in the error function landscape.
- requires the derivatives of activation functions to be known at network design time.

# Computing the derivatives of error

We consider an error function for the model  $E(X, W, t)$

Compute the derivative of this function with respect to every weight  $w_{j,k}^l$  (the weight from layer  $l$ , between neuron  $j$  from layer  $l$  and neuron  $k$  from layer  $l-1$ ).

In general it is:

$$\frac{\partial E}{\partial w_{j,k}^l} = \frac{\partial E}{\partial o_j^l} \frac{\partial o_j^l}{\partial w_{j,k}^l} = \frac{\partial E}{\partial o_j^l} \frac{\partial o_j^l}{\partial \text{net}_j^l(X^l)} \frac{\partial \text{net}_j^l(X^l)}{\partial w_{j,k}^l}$$

Observe that in the last derivative we have a sum, but only one term depends on the weight so:

$$\frac{\partial \text{net}_j^l(X^l)}{\partial w_{j,k}^l} = \frac{\partial}{\partial w_{j,k}^l} \left( \sum_{q=1}^{n_{l-1}} x_q^l w_{j,q}^l \right) = \frac{\partial(x_k^l w_{j,k}^l)}{\partial w_{j,k}^l} = x_k^l = o_k^{l-1}$$

Here:  $X^l = (x_1^l, x_2^l, \dots, x_{n_l-1}^l)$  is the input for layer  $l$  (aka the output  $o^{l-1}$  from the previous layer)

# Computing the derivatives of error

For the first hidden layer we have a different situation: the input for this layer is the actual input in the network.

If we evaluate further the partial derivative we get:

$$\frac{\partial o_j^l}{\partial \text{net}_j^l(X^l)} = \frac{\partial \phi(\text{net}_j^l(X^l))}{\partial \text{net}_j^l(X^l)}$$

which is the partial derivative of the activation function  $\phi$  - hence the importance of the derivative

In practice we begin from the last layer because the partial derivatives for this layer are straight forward, and we move backward from layer to layer until we reach the first hidden one.

# Thank You!

# Deep learning

April 2023

Introduction

Tensors

Convolution

Kernels

Convolutional  
Network -  
structure

Convolutional  
Layer

Feature learning  
Training the  
Convolution Layer

Pooling Layer

# Outline

Introduction

Introduction

Tensors

Tensors

Convolution

Convolution

Kernels

Kernels

Convolutional Network - structure

Convolutional Network - structure

Convolutional Layer

Convolutional Layer

Feature learning

Feature learning  
Training the Convolution Layer

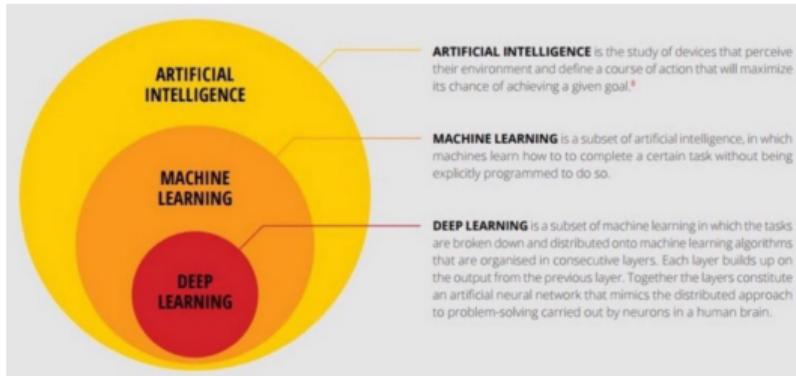
Training the Convolution Layer

Pooling Layer

Pooling Layer

# What is deep learning?

- ▶ a sub-field of machine learning dealing with algorithms inspired by the structure and function of the brain called artificial neural networks



- ▶ it mirrors the functioning of our brains
- ▶ similar to how nervous system structured where each neuron connected each other and passing information

Introduction

Tensors

Convolution

Kernels

Convolutional  
Network -  
structure

Convolutional  
Layer

Feature learning  
Training the  
Convolution Layer

Pooling Layer

# Deep Convolutional Neural Networks

(*ConvNets*) are deep artificial neural networks used:

- ▶ to classify images (e.g. name what they see)
- ▶ cluster them by similarity (photo search)
- ▶ perform object recognition within scenes

algorithms that can identify faces, individuals, street signs, tumors, perform optical character recognition (OCR).

Introduction

Tensors

Convolution

Kernels

Convolutional  
Network -  
structure

Convolutional  
Layer

Feature learning  
Training the  
Convolution Layer

Pooling Layer

# Deep Convolutional Neural Networks

- ▶ pre-processing required in a ConvNet is much lower as compared to other classification algorithms
- ▶ architecture of a ConvNet is analogous to that of the connectivity pattern of Neurons in the Human Brain - Visual Cortex
- ▶ a ConvNet captures the spatial and temporal dependencies in an image

Introduction

Tensors

Convolution

Kernels

Convolutional Network - structure

Convolutional Layer

Feature learning  
Training the Convolution Layer

Pooling Layer

# Tensors

Introduction

Tensors

Convolution

Kernels

Convolutional  
Network -  
structure

Convolutional  
Layer

Feature learning  
Training the  
Convolution Layer

Pooling Layer

An  $n^{th}$  - rank tensor in  $m$ -dimensional space is a mathematical object that has  $n$  indices and  $m^n$  components and obeys certain transformation rules.

- generalizations of scalars, vectors, and matrices to an arbitrary number of indices.
- used in physics such as elasticity, fluid mechanics, and general relativity.

# Notations for tensors

- ▶  $a_{ijk\dots}$ ,  $a^{ijk\dots}$ ,  $a_i^{jk\dots}$ , etc., may have an arbitrary number of indices
  - ▶ a tensor (rank  $r + s$ ) may be of mixed type  $(r, s)$ :  
 $r$  "contravariant" (upper) indices and  $s$  "covariant" (lower) indices.
- the positions of the slots in which contravariant and covariant indices are placed are significant!

$$a_{\mu\nu}^{\lambda} \neq a_{\mu}^{\nu\lambda}$$

Introduction

Tensors

Convolution

Kernels

Convolutional  
Network -  
structure

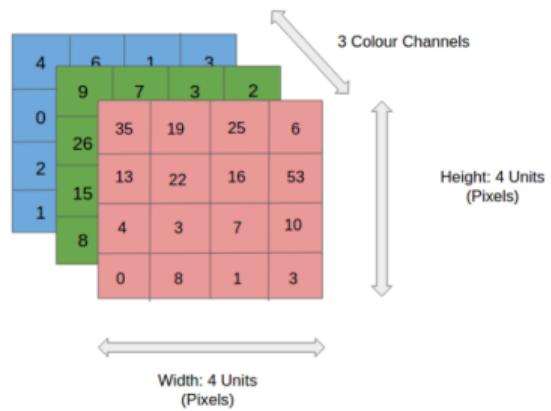
Convolutional  
Layer

Feature learning  
Training the  
Convolution Layer

Pooling Layer

# Images as tensors

ConvNets ingest and process images as tensors.



- reduce the images → form easy to process (without losing critical features)

Introduction  
Tensors  
Convolution  
Kernels  
Convolutional Network - structure  
Convolutional Layer  
Feature learning  
Training the Convolution Layer

Pooling Layer

# Convolution operation

an operation on two functions  $x(t)$  (**input**) and  $w(t)$  (**kernel**)  
of a real - valued argument

$$s(t) = \int x(a)w(t-a)da$$

notation:

$$s(t) = (x \circledast w)(t)$$

if  $t$  is discrete the integral turns into a sum

$$s(t) = (x \circledast w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

Introduction

Tensors

Convolution

Kernels

Convolutional  
Network -  
structure

Convolutional  
Layer

Feature learning  
Training the  
Convolution Layer

Pooling Layer

# Convolution operation

in practice we have two tensors:

- ▶ the input - a multidimensional array of data
  - ▶ the kernel - a multidimensional array of parameters  
(adapted by the learning algorithm)
- stored separately → that these functions are zero everywhere but in the finite set of points
  - we can implement the infinite summation as a summation over a finite number of array elements
  - convolutions can be over more than one axis at a time

$$S(i, j) = (I \circledast K)(i, j) = \sum_m \sum_n I(m, n)K(i - m, j - n)$$

Introduction

Tensors

Convolution

Kernels

Convolutional Network - structure

Convolutional Layer

Feature learning  
Training the Convolution Layer

Pooling Layer

# Using a kernel

we want apply a filter over the image

AIM: **extract the high-level features** (edges, color, gradient orientation)

Example: *Image Dimensions = 5 (Height) x 5 (Breadth) x 1 (Number of channels, eg. RGB)*

Kernel / filter:

$$K = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

Introduction

Tensors

Convolution

Kernels

Convolutional Network - structure

Convolutional Layer

Feature learning  
Training the Convolution Layer

Pooling Layer

# Using a kernel

Introduction

Tensors

Convolution

Kernels

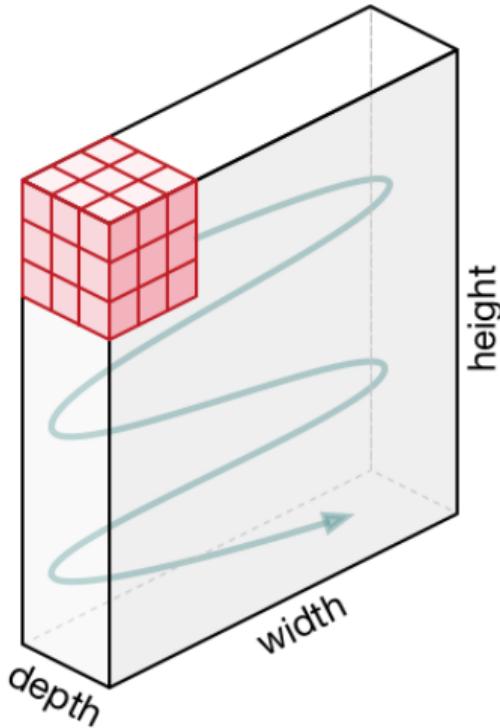
Convolutional  
Network -  
structure

Convolutional  
Layer

Feature learning  
Training the  
Convolution Layer

Pooling Layer

# Using a Kernel - movement of the Kernel



- ▶ kernel shifts
- ▶ every time performing a matrix multiplication operation between K and the portion P of the image over which the kernel is hovering

Introduction  
Tensors  
Convolution  
Kernels  
Convolutional Network - structure  
Convolutional Layer  
Feature learning  
Training the Convolution Layer  
Pooling Layer

# Using a kernel - 3 channels

Introduction

Tensors

Convolution

Kernels

Convolutional  
Network -  
structure

Convolutional  
Layer

Feature learning  
Training the  
Convolution Layer

Pooling Layer

# Layers of a convolutional network

Typical three stages:

- ▶ first: several convolutions
- ▶ second: a nonlinear activation function (ex: rectified linear activation function) - **detector stage**
- ▶ third: **pooling** function to modify the output

Examples: max pooling (Zhou and Chellappa, 1988 - maximum output within a rectangular neighborhood), the average of a rectangular neighborhood,  $L^2$  norm of a rectangular neighborhood, a weighted average based on the distance from the central pixel.

Introduction  
Tensors  
Convolution  
Kernels  
Convolutional Network - structure  
Convolutional Layer  
Feature learning  
Training the Convolution Layer  
Pooling Layer

# Training the network

- ▶ similar with ANN
- ▶ after computing the error, a gradient descent method is applied to all layers

Introduction

Tensors

Convolution

Kernels

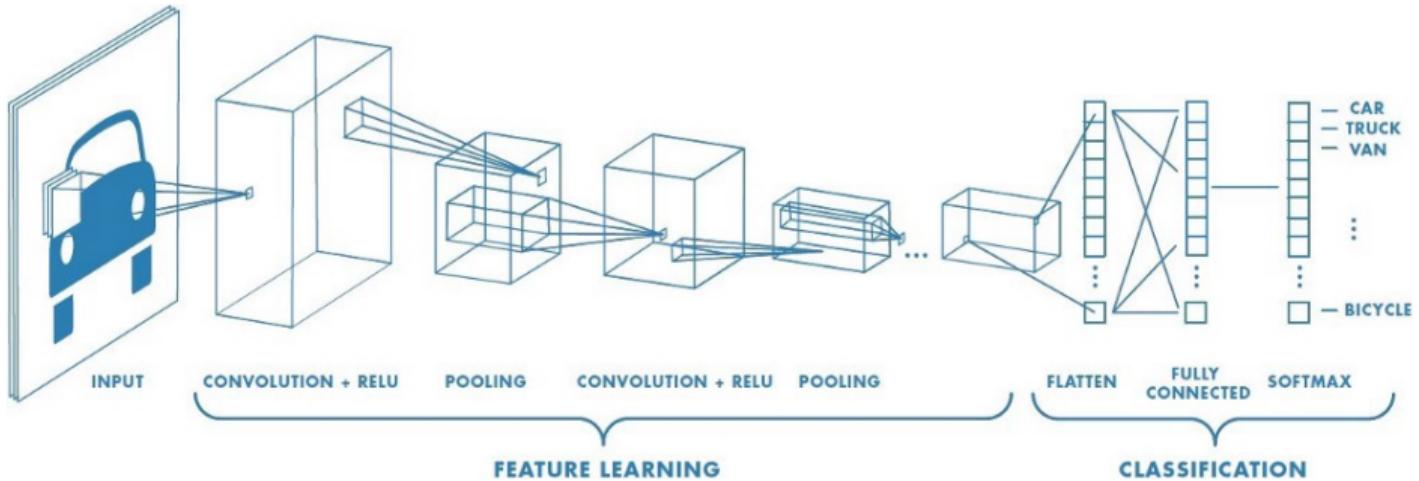
Convolutional  
Network -  
structure

Convolutional  
Layer

Feature learning  
Training the  
Convolution Layer

Pooling Layer

# Feature learning



# Feature learning

Applying this filter to an image will result in a feature map that only contains vertical lines. It is a vertical line detector.

1.0	0.0	-1.0
2.0	0.0	-2.0
1.0	0.0	-1.0

Table: A 3x3 element Sobel filter for detecting vertical lines

Dragging this filter systematically across pixel values in an image can only highlight vertical line pixels.

# Feature learning

-2.0	-2.0	-4.0	-2.0	-2.0
-1.0	-1.0	-2.0	-1.0	-1.0
0.0	0.0	0.0	0.0	0.0
1.0	1.0	2.0	1.0	1.0
2.0	2.0	4.0	2.0	2.0

Table: A horizontal line detector

Introduction

Tensors

Convolution

Kernels

Convolutional Network - structure

Convolutional Layer

Feature learning  
Training the Convolution Layer

Pooling Layer

- ▶ combining the filters' results (combining both feature maps, will result in all of the lines in an image being highlighted)
- ▶ a suite of tens or even hundreds of other small filters can be designed to detect other features in the image

# Feature learning

- ★ the values of the filter are weights to be learned during the training of the network
- training under gradient descent
- ▶ the network will learn what types of features to extract from the input
- ▶ the extracted features will be those that minimize the loss function (e. g. extract features that are the most useful for classifying images as dogs or cats)

Introduction

Tensors

Convolution

Kernels

Convolutional Network - structure

Convolutional Layer

Feature learning  
Training the Convolution Layer

Pooling Layer

# Gradient descent on Convolution Layer

- ▶ the input  $X = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \\ x_{3,1} & x_{3,2} & x_{3,3} \end{bmatrix}$
- ▶ the filter  $F = \begin{bmatrix} f_{1,1} & f_{1,2} \\ f_{2,1} & f_{2,2} \end{bmatrix}$
- ▶ the output is  $O = X \circledast F$
- ▶  $\frac{\partial E}{\partial O}$  is the gradient of loss from previous layer

Introduction

Tensors

Convolution

Kernels

Convolutional  
Network -  
structure

Convolutional  
Layer

Feature learning  
Training the  
Convolution Layer

Pooling Layer

# Gradient descent on Convolution Layer

[Introduction](#)[Tensors](#)[Convolution](#)[Kernels](#)[Convolutional Network - structure](#)[Convolutional Layer](#)[Feature learning  
Training the Convolution Layer](#)[Pooling Layer](#)

After we apply the convolution on  $X$  and  $F$  we have:

$$O_{1,1} = x_{1,1} * F_{1,1} + x_{1,2} * F_{1,2} + x_{2,1} * F_{2,1} + x_{2,2} * F_{2,2}$$

$$O_{1,2} = x_{1,2} * F_{1,1} + x_{1,3} * F_{1,2} + x_{2,2} * F_{2,1} + x_{2,3} * F_{2,2}$$

$$O_{2,1} = x_{2,1} * F_{1,1} + x_{2,2} * F_{1,2} + x_{3,1} * F_{2,1} + x_{3,2} * F_{2,2}$$

$$O_{2,2} = x_{2,2} * F_{1,1} + x_{2,3} * F_{1,2} + x_{3,2} * F_{2,1} + x_{3,3} * F_{2,2}$$

# Gradient descent on Convolution Layer

we compute the partial derivative of  $O$  with respect of  $F$

$$\frac{\partial O_{1,1}}{\partial F_{1,1}} = x_{1,1}, \quad \frac{\partial O_{1,1}}{\partial F_{1,2}} = x_{1,2}, \quad \frac{\partial O_{1,1}}{\partial F_{2,1}} = x_{2,1}, \quad \frac{\partial O_{1,1}}{\partial F_{2,2}} = x_{2,2}$$

similar we compute for  $O_{1,2}$ ,  $O_{2,1}$ , and  $O_{2,2}$

the gradient to update the filter will be

$$\frac{\partial E}{\partial F} = \frac{\partial E}{\partial O} * \frac{\partial O}{\partial F} \tag{1}$$

Introduction  
Tensors  
Convolution  
Kernels  
Convolutional Network - structure  
Convolutional Layer  
Feature learning  
Training the Convolution Layer  
Pooling Layer

# Gradient descent on Convolution Layer

if we expand Equation 1

$$\begin{aligned}\frac{\partial E}{\partial F_{1,1}} &= \frac{\partial E}{\partial O_{1,1}} * \frac{\partial O_{1,1}}{\partial F_{1,1}} + \frac{\partial E}{\partial O_{1,2}} * \frac{\partial O_{1,2}}{\partial F_{1,1}} \\ &\quad + \frac{\partial E}{\partial O_{2,1}} * \frac{\partial O_{2,1}}{\partial F_{1,1}} + \frac{\partial E}{\partial O_{2,2}} * \frac{\partial O_{2,2}}{\partial F_{1,1}} \\ &= \frac{\partial E}{\partial O_{1,1}} * x_{1,1} + \frac{\partial E}{\partial O_{1,2}} * x_{1,2} \\ &\quad + \frac{\partial E}{\partial O_{2,1}} * x_{2,1} + \frac{\partial E}{\partial O_{2,2}} * x_{2,2}\end{aligned}$$

Introduction

Tensors

Convolution

Kernels

Convolutional  
Network -  
structure

Convolutional  
Layer

Feature learning  
Training the  
Convolution Layer

Pooling Layer

# Gradient descent on Convolution Layer

[Introduction](#)[Tensors](#)[Convolution](#)[Kernels](#)[Convolutional Network - structure](#)[Convolutional Layer](#)[Feature learning  
Training the Convolution Layer](#)[Pooling Layer](#)

we observe that this is actually a convolution product between the input and the loss gradient

$$\begin{bmatrix} \frac{\partial E}{\partial F_{1,1}} & \frac{\partial E}{\partial F_{1,2}} \\ \frac{\partial E}{\partial F_{2,1}} & \frac{\partial E}{\partial F_{2,2}} \end{bmatrix} = X \circledast \begin{bmatrix} \frac{\partial E}{\partial O_{1,1}} & \frac{\partial E}{\partial O_{1,2}} \\ \frac{\partial E}{\partial O_{2,1}} & \frac{\partial E}{\partial O_{2,2}} \end{bmatrix} \quad (2)$$

# Gradient descent on Convolution Layer

Introduction

Tensors

Convolution

Kernels

Convolutional  
Network -  
structure

Convolutional  
Layer

Feature learning  
Training the  
Convolution Layer

Pooling Layer

similar we get for the derivative of  $E$  with respect of  $X$

$$\begin{bmatrix} \frac{\partial E}{\partial x_{1,1}} & \frac{\partial E}{\partial x_{1,2}} & \frac{\partial E}{\partial x_{1,3}} \\ \frac{\partial E}{\partial x_{2,1}} & \frac{\partial E}{\partial x_{2,2}} & \frac{\partial E}{\partial x_{2,3}} \\ \frac{\partial E}{\partial x_{3,1}} & \frac{\partial E}{\partial x_{3,2}} & \frac{\partial E}{\partial x_{3,3}} \end{bmatrix} = \begin{bmatrix} F_{2,2} & F_{2,1} \\ F_{1,2} & F_{1,1} \end{bmatrix} \circledast \begin{bmatrix} \frac{\partial E}{\partial O_{1,1}} & \frac{\partial E}{\partial O_{1,2}} \\ \frac{\partial E}{\partial O_{2,1}} & \frac{\partial E}{\partial O_{2,2}} \end{bmatrix}$$

observe that matrix  $F$  is flipped over 180° degrees in this formula.

# Pooling

helps to make the representation approximately invariant to small translations of the input.

- useful property if we care more about whether some feature is present than exactly where it is;
- essential for handling inputs of varying size;
- Some guidance as to which kinds of pooling one should use in various situations : Boureau et al., 2010.

Introduction

Tensors

Convolution

Kernels

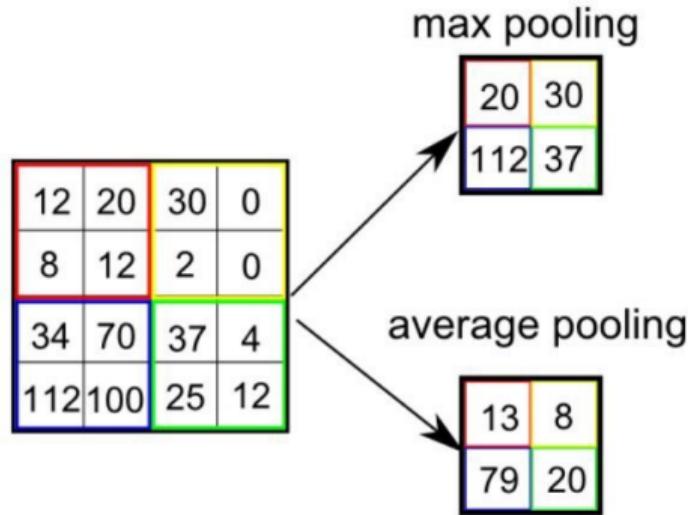
Convolutional Network - structure

Convolutional Layer

Feature learning  
Training the Convolution Layer

Pooling Layer

# Pooling



- ▶ Max Pooling - returns the maximum value from the portion of the image covered by the Kernel;
- ▶ Average Pooling returns the average of all the values from the portion of the image covered by the Kernel.

Introduction  
Tensors  
Convolution  
Kernels  
Convolutional Network - structure  
Convolutional Layer  
Feature learning  
Training the Convolution Layer  
Pooling Layer

# Training the Pooling layer

Introduction

Tensors

Convolution

Kernels

Convolutional  
Network -  
structure

Convolutional  
Layer

Feature learning  
Training the  
Convolution Layer

Pooling Layer

in an  $N \times N$  pooling block, backward propagation of error is reduced to a single value - value of the “winning unit”

# ARTIFICIAL INTELLIGENCE



**Solving search problems**

Uninformed search strategies

# Content

---

- ❑ Problems
- ❑ Problem solving
  - Steps of problem solving
- ❑ Solving problem by search
  - Steps of solving problem by search
  - Search strategies

# Problems

---



- Two problem types:
  - Solving in a deterministic manner
    - Computing the sinus of an angle or the square root of a number
  - Solving in a stochastic manner
    - Real-world problems → design of ABS
    - Involve the search of a solution → AI's methods

# Problems

## □ Tipology

### ■ Search/optimisation problems

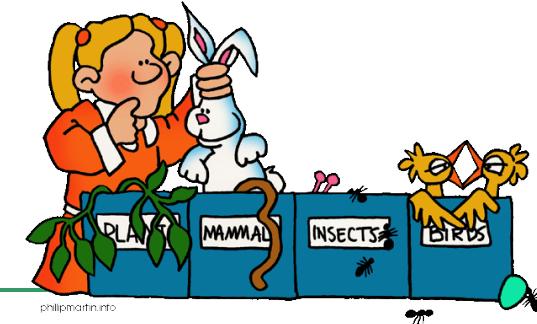
- Planning, satellite's design

### ■ Modeling problems

- Predictions, classifications

### ■ Simulation problems

- Game theory

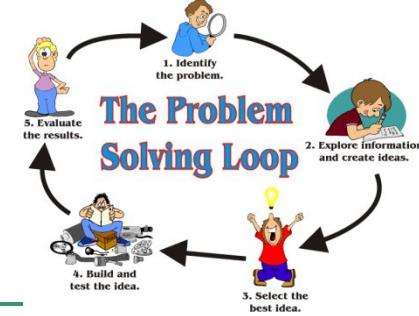




# Problem solving

---

- Identification of a solution
  - In computer science (AI) → search process
  - In engineering and mathematics → optimisation process
  
- How?
  - Representation of (partial) solutions → points in the search space
  - Design of a search operators → map a potential solution into another one



# Steps in problem solving

- Problem definition
- Problem analyses
- Selection of a solving technique
  - Search
  - Knowledge representation
  - Abstraction



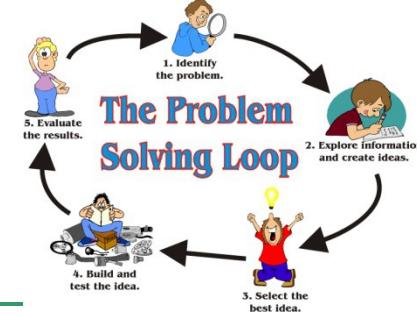
# Solving problems by search

---

- Based on some objectives
- Composed by actions that accomplish the objectives
  - Each action changes a state of the problem
- More actions that map the initial state of problem into a final state

# Steps in solving problems by search

## Problem definition

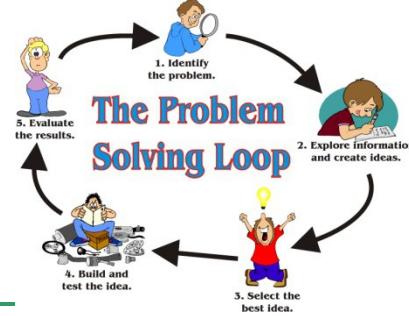


### □ Problem definition involves:

- A search space
  - All possible states
  - Representation
    - Explicit – construction of all possible states
    - Default – by using some data structures and some functions (operators)
- One or more initial state
- One or more final states
- One or more paths
  - More successive states
- A set of rules (actions)
  - Successor functions (operators) – next state after a given one
  - Cost functions that evaluate
    - How a state is mapped into another state
    - An entire path
  - Objective functions that check if a state is final or not

# Steps in solving problems by search

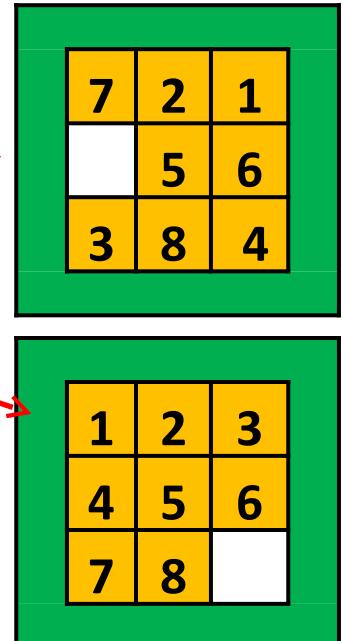
## Problem definition

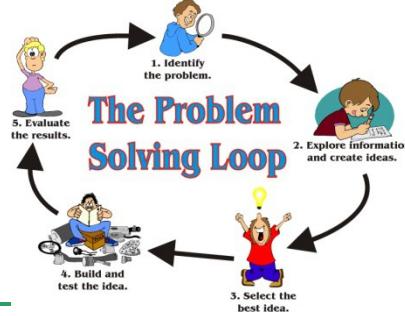


### □ Examples

#### ■ Puzzle game with 8 pieces

- State's space – different board configurations for a game with 8 pieces
- Initial state – a random configuration
- Final state – a configuration where all the pieces are sorted in a given manner
- Rules -> white moves
  - conditions: move inside the table
  - Transformations: the white space is moved up, down, to left or to right
- Solution - optimal sequence of white moves





# Steps in solving problems by search

## Problem definition

### □ Examples

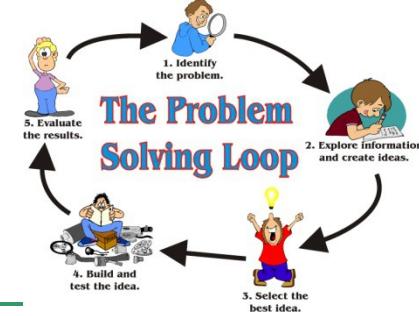
#### ■ Queen's problem

- State's space – different board configurations for a game with  $n$  queens
- Initial state – a configuration without queens
- Final state – a configuration  $n$  queens so that none of them can hit any other in one move
- Rules -> put a queen on the table
  - conditions: the queen is not hit by any other queen
  - Transformations: put a new queen in a free cell of the table
- Solution - optimal placement of queens

	a	b	c	d	e	f	g	h	
1	Q				Q				1
2				Q				Q	2
3					Q				3
4						Q			4
5		Q							5
6					Q				6
7	Q								7
8						Q			8
	a	b	c	d	e	f	g	h	

# Steps in solving problems by search

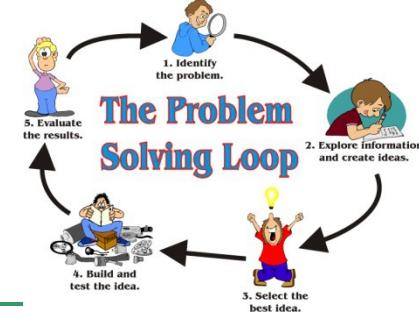
## Problem analyse



- The problem can be decomposed?
  - The sub-problems are independent or not?
- The possible state's space is predictable?
- We want a solution or an optimal solution?
- The solution is represented by a single state or by more successive states?
- We require some knowledge for limiting the search or for identifying the solution?
- The problem is conversational or solitary?
  - Human interaction is required for problem solving?

# Steps in solving problems by search

## Selection of a solving technique

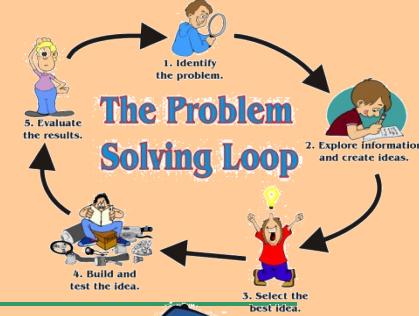


- Solving by moving rules (and control strategy) in the search space until we find a path from the initial state to the final state
- Solving by search
  - Examination of all possible states in order to identify
    - A path from the initial state to the final state
    - An optimal state
  - The search space = all possible states and the operators that maps the states



# Steps in solving problems by search

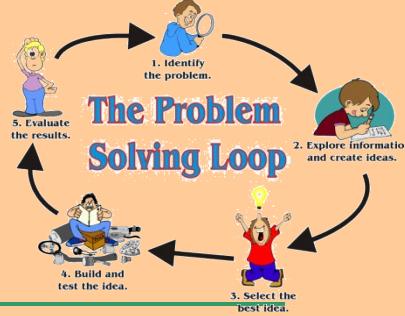
## Selection of a solving technique



### □ Solving by search

- More searching strategies → how we select one of them?
  - Computational complexity (temporal and spatial)
  - Completeness → the algorithms always ends and finds a solution (if it exists)
  - Optimal → the algorithms finds the optimal solution (the optimal cost of the path from the initial state to the final state)





# Steps in solving problems by search

## Selection of a solving technique

### □ Solving by search

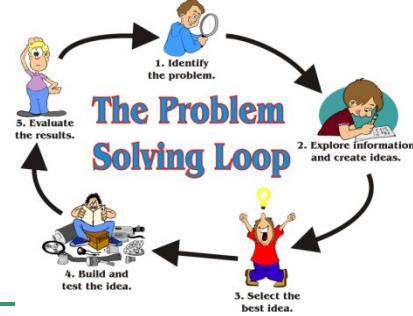
- More searching strategies → how we select one of them? → Computational complexity (temporal and spatial)
  - Strategy's performance depends on
    - Time for running
    - Memory for running
    - Size of input data
    - Computer's performance
    - Compiler's quality
  - Can be evaluated by complexity → computational efficiency
    - Spatial → required memory for solution identification
      - $S(n)$  – memory used by the best algorithms  $A$  that solves a decision problem  $f$  with  $n$  input data
    - Temporal → required time for solution identification
      - $T(n)$  – running time (number of steps) of the best algorithm  $A$  that solves a decision problem  $f$  with  $n$  input data

} Internal factors  
}

} External factors

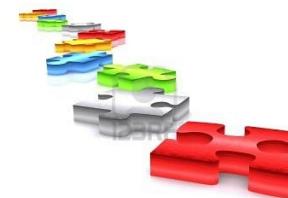
# Steps in solving problems by search

## Selection of a solving technique



- Problem solving by search can be performed by:

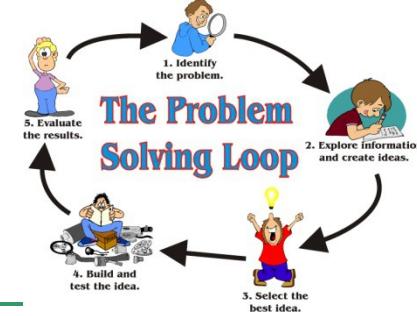
- Step by step construction of solution
- Optimal solution identification



www.shutterstock.com - 36774760

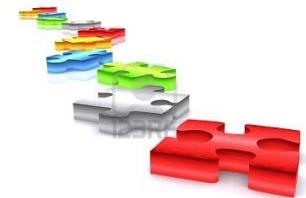
# Steps in solving problems by search

## Selection of a solving technique



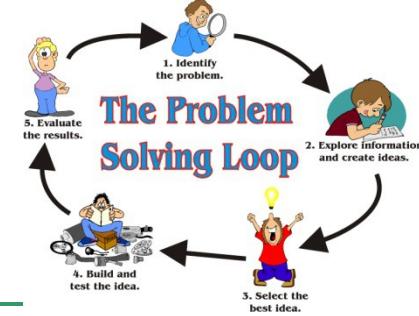
### □ Problem solving by search can be performed by:

- Step by step construction of solution
  - ▣ Problem's components
    - Initial state
    - Operators (successor functions)
    - Final state
    - Solution = a path (of optimal cost) from the initial state to the final state
  - ▣ Search space
    - All the states that can be obtained from the initial state (by using the operators)
    - A state = a component of solution
  - ▣ Example
    - Traveling Salesman Problem (TSP)
  - ▣ Algorithms
    - Main idea: start with a solution's component and adding new components until a complete solution is obtained
    - Recurrent → until a condition is satisfied
    - The search's history (path from initial state to the final state) is retained in LIFO/FIFO containers
  - ▣ Advantages
    - Do not require knowledge (intelligent information)



# Steps in solving problems by search

## Selection of a solving technique



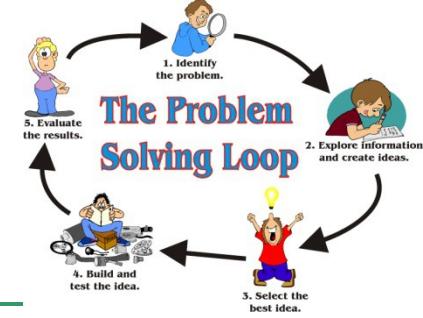
### □ Problem solving by search can be performed by:

- Optimal solution identification
  - ▣ Problem's components
    - Conditions (constraints) that must be satisfied by the solution
    - Evaluation function for a potential solution → optimum identification
  - ▣ Search space
    - All possible and complete solutions
    - State = a complete solution
  - ▣ Example
    - Queen's problem
  - ▣ Algorithms
    - Main idea: start with a state that doesn't respect some conditions and change it for eliminating these violations
    - Iterative → a single state is retained and the algorithm tries to improve it
    - The searches history is not retained
  - ▣ Advantages
    - Simple
    - Requires a small memory
    - Can find good solutions in (continuous) search spaces very large (where other algorithms can not be utilised)



# Steps in solving problems by search

## Selection of a solving technique



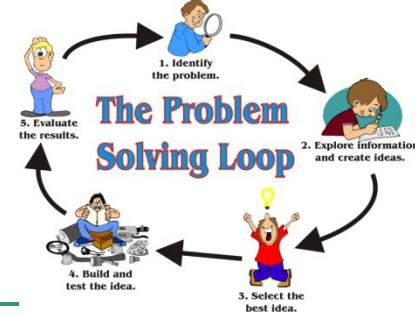
### □ Solving problem by search involves:

- Very complex algorithms (NP-complete problems)
- Search in an exponential space



# Steps in solving problems by search

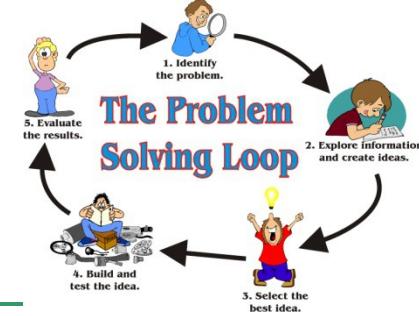
## Selection of a solving technique



- Topology of search strategies:
  - Solution **generation**
    - **Constructive** search
      - Solution is identified step by step
      - Ex. TSP
    - **Perturbative** search
      - A possible solution is modified in order to obtain another possible solution
      - Ex. SAT - Propositional Satisfaction Problem
  - Search space **navigation**
    - **Systematic** search
      - The entire search space is visited
        - Solution identification (if it exists) → complete algorithms
    - **Local** search
      - Moving from a point of the search space into a neighbor point → incomplete algorithms
      - A state can be visited more times
  - **Certain** items of the search
    - **Deterministic** search
      - Algorithms that exactly identify the solution
    - **Stochastic** search
      - Algorithms that approximate the solution
  - Search space **exploration**
    - **Sequential** search
    - **Parallel** search

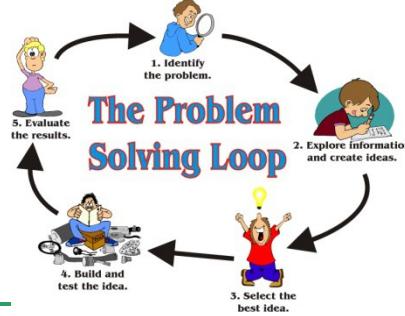
# Steps in solving problems by search

## Selection of a solving technique



Topology of search strategies:

- **Number of objectives**
  - **Single-objective** search
    - The solution must respect a single condition/constraint
  - **Multi-objective** search
    - The solution must respect more conditions/constraints
- **Number of solutions**
  - **single-modal search**
    - There is a single optimal solution
  - **multi-modal search**
    - There are more optimal solutions
- **Algorithm**
  - Search over a **finite number of steps**
  - **Iterative search**
    - The algorithms converge through the optimal solutions
  - **Heuristic search**
    - The algorithms provide an approximation of the solution
- Search **mechanism**
  - **traditional search**
  - **modern search**
- where the search takes **place**
  - **local search**
  - **global search**



# Steps in solving problems by search

## Selection of a solving technique

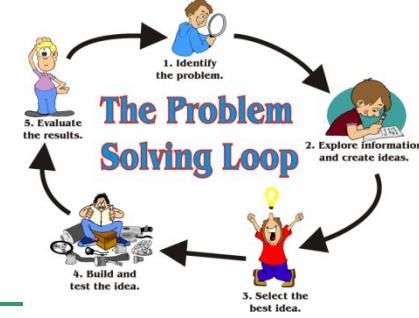
---

Topology of search strategies:

- Type **(linearity) of constraints**
  - **Linear search**
  - **non-linear search**
    - Clasical (deterministic)
      - Direct – based on evaluation of the objective function
      - Indirect – based on derivative (I and/or II) of the objective function
    - Enumeration-based
      - How solution is identified
        - Uninformed – the solution is the final state
        - Informed – deals with an evaluation function for a possible solution
      - Search space type
        - Complete – the space is finite (if solution exists, then it can be found)
        - Incomplete – the space is infinite
    - Stochastic search
      - Based on random numbers
- **Agents** involves in search
  - Search by **a single agent** → without obstacle for achieving the objectives
  - **Adversarial search** → the opponent comes with some uncertainty

# Steps in solving problems by search

## Selection of a solving technique

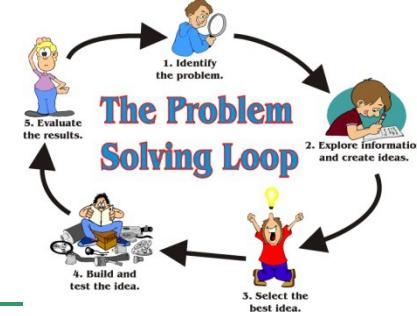


### Example

- Topology of search strategies
  - Solution generation
    - **Constructive search**
    - Perturbative search
  - Search space navigation
    - **Systematic search**
    - Local search
  - Certain items of the search
    - **Deterministic search**
    - Stochastic search
  - Search space exploration
    - **Sequential search**
    - Parallel search

# Steps in solving problems by search

## Selection of a solving technique

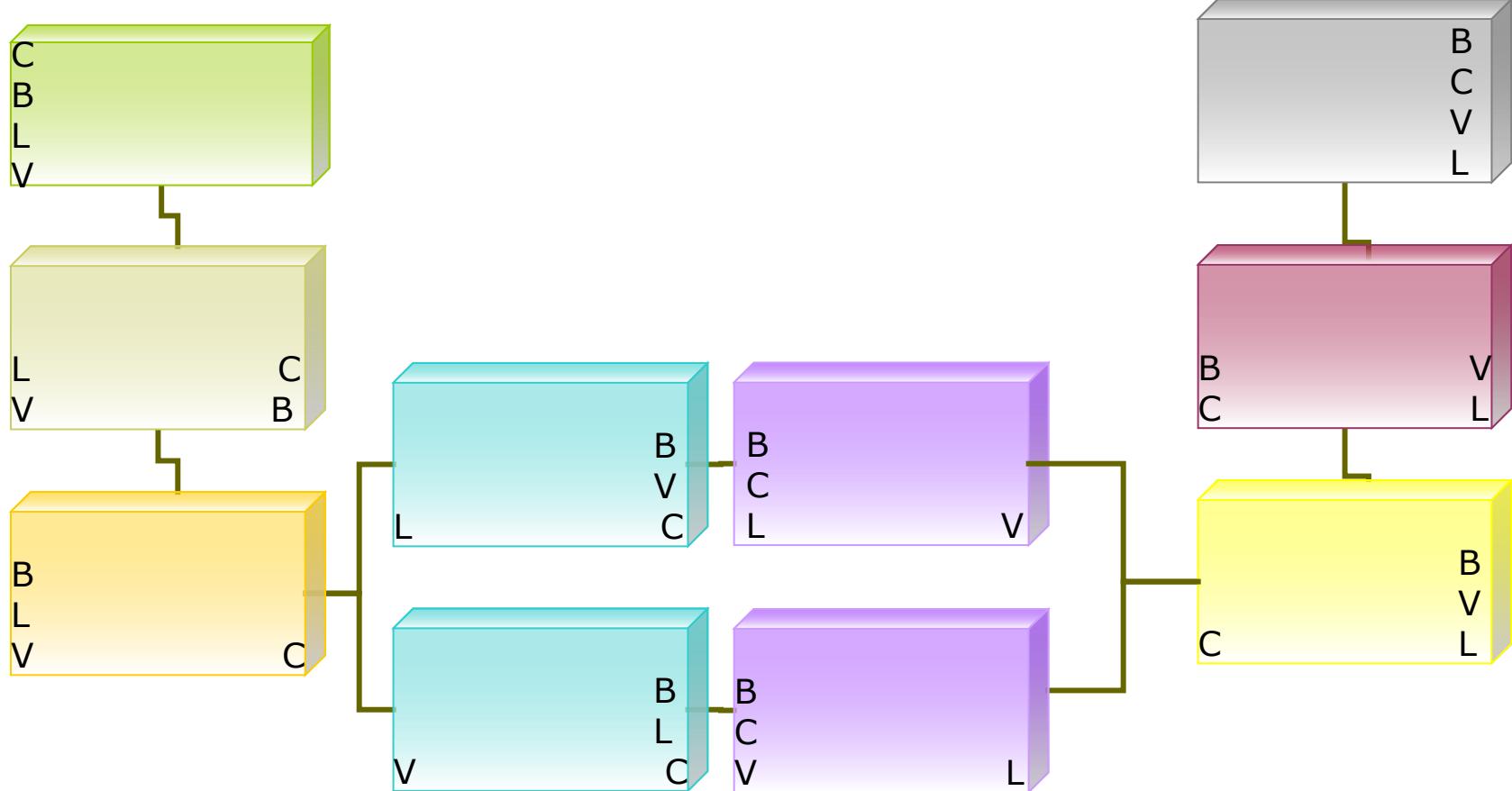
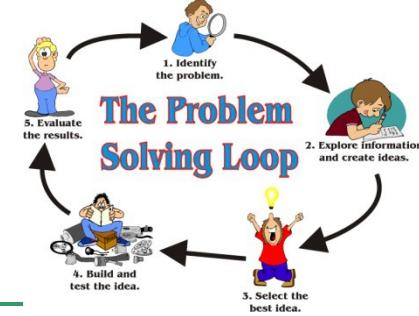


### Example

- Constructive, global, deterministic, sequential search
- Problem "capra, varza și lupul"
  - Input :
    - A goat, a cabbage and a wolf on a river-side
    - A boat with a boater
  - Output:
    - Move all the passengers on the other side of the river
    - Taking into account:
      - The boat has only 2 places
      - It is not possible to rest on the same side:
        - The goat and the cabbage
        - The wolf and the goat

# Steps in solving problems by search

## Selection of a solving technique



# Search strategies – Basic elements

---



- Abstract data types (ADTs)
  - ADT list → linear structure
  - ADT tree → hierachic structure
  - ADT graph → graph-based structure
  
- ADT
  - Domain and operations
  - Representation



# Uninformed search strategies (USS)

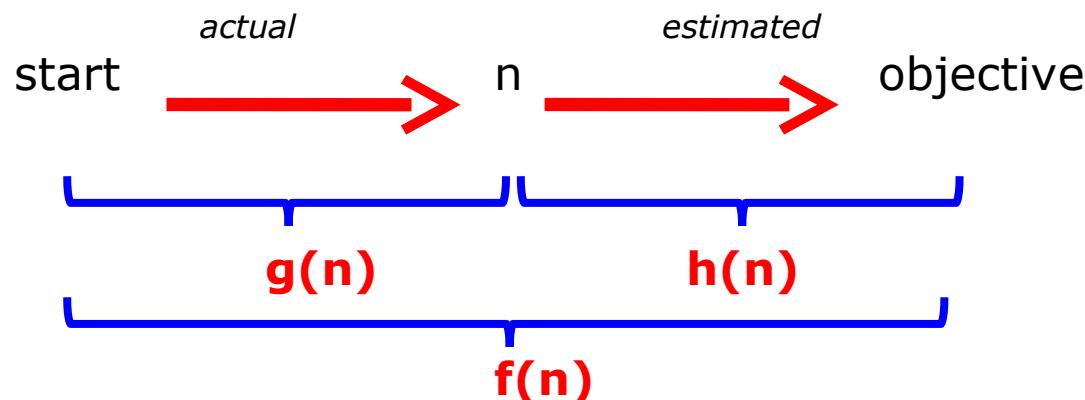
- Characteristics
  - Are NOT based on problem specific information
  - Are general
  - Blind strategies
  - Brute force methods
- Topology
  - Order of node exploration:
    - USS in linear structures
      - Linear search
      - Binary search
    - USS in non-linear structures
      - Breadth-first search
        - Uniform cost search (branch and bound)
      - Depth first search
        - Limited depth first search
        - iterative deepening depth-first search
      - Bidirectional search



# SS in tree-based structures

## □ Basic elements

- $f(n)$  – evaluation function for estimating the cost of a solution through node (state)  $n$
- $h(n)$  – evaluation function for estimating the cost of a solution path from node (state)  $n$  to the final node (state)
- $g(n)$  – evaluation function for estimating the cost of a solution path from the initial node (state) to node (state)  $n$
- $f(n) = g(n) + h(n)$





# USS in tree-based structures

## Breadth-first search – BFS

### □ Basic elements

- All the nodes of depth  $d$  are visited before all the nodes of depth  $d+1$
- All children of current node are added into a **FIFO** list (**queue**)

### □ Example

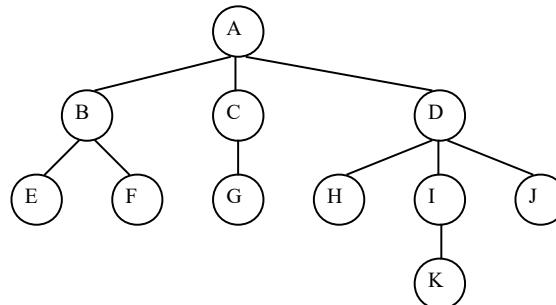
- Visiting order: A, B, C, D, E, F, G, H, I, J, K

### □ Algorithm

```

bool BFS(elem, list){
    found = false;
    visited = {};
    toVisit = {start};           //FIFO list
    while((toVisit != {}) && (!found)){
        node = pop(toVisit);
        visited = visited U {node};
        if (node == elem)
            found = true;
        else{
            aux = {};
            for all (unvisited) children of node do
                aux = aux U {child};
        }
        toVisit = toVisit U aux;
    } //while
    return found;
}

```



Vizitate deja	De vizitat
$\emptyset$	A
A	B, C, D
A, B	C, D, E, F
A, B, C	D, E, F, G
A, B, C, D	E, F, G, H, I, J
A, B, C, D, E	F, G, H, I, J
A, B, C, D, E, F	G, H, I, J
A, B, C, D, E, F, G	H, I, J
A, B, C, D, E, F, G, H	I, J
A, B, C, D, E, F, G, H, I	J, K
A, B, C, D, E, F, G, H, I, J	K
A, B, C, D, E, F, G, H, I, J, K	$\emptyset$

# USS in tree-based structures

## Breadth-first search – BFS



### □ Search analyse:

- Time complexity:
  - $b$  – ramification factor (number of children of a node)
  - $d$  – length (depth) of solution
  - $T(n) = 1 + b + b^2 + \dots + b^d \Rightarrow O(b^d)$
- Space complexity
  - $S(n) = T(n)$
- Completeness
  - If solution exists, then BFS finds it
- Optimality
  - No

### □ Advantages

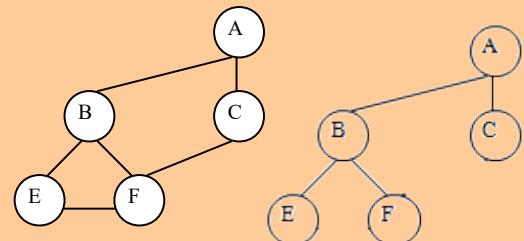
- Finds the shortest path to the objective node (the shallowest solution)

### □ Disadvantages

- Generate and retain a tree whose size exponentially increases (with depth of objective node)
- Exponential time and space complexity
- **Russel&Norving experiment**
- Works only for small search spaces

### □ Applications

- Identification of connex components in a graph
- Identification of the shortest path in a graph
- Optimisation in transport networks → **algorithm Ford-Fulkerson**
- Serialisation/deserialisation of a binary tree (vs. serialization in a sorted manner) allows efficiently reconstructing of the tree
- Collection copy (garbage collection) → **algorithm Cheney**



Vizitate deja	De vizitat
Φ	B
B	A, E, F
B, A	E, F, C
B, A, E	F, C
B, A, E, F	C
B, A, E, F, C	Φ



# USS in tree-based structures

## Uniform cost search – UCS

### Basic elements

- BFS +special expand procedure (based on the cost of links between nodes)
- All the nodes of depth  $d$  are visited before all the nodes of depth  $d+1$
- All children of current node are added into a **FIFO ordered** list
  - The nodes of minimum cost are firstly expanded
  - When a path to the final state is obtained, it became a candidate to the optimal solution
- *Branch and bound* algorithm

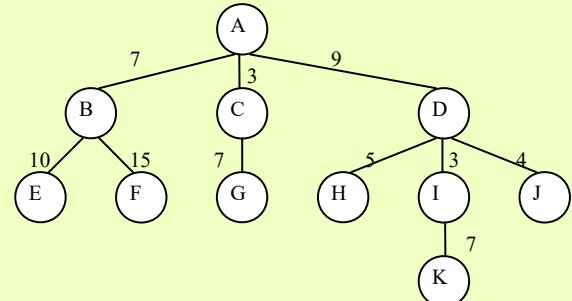
### Example

- Visiting order: A, C, B, D, G, E, F, I, H, J, K

### Algorithm

```

bool UCS(elem, list){
    found = false;
    visited = {};
    toVisit = {start}; //FIFO sorted list
    while((toVisit != {}) && (!found)){
        node = pop(toVisit);
        visited = visited U {node};
        if (node== elem)
            found = true;
        else
            aux = {};
            for all (unvisited) children of node do{
                aux = aux U {child};
            } // for
            toVisit = toVisit U aux;
            TotalCostSort(toVisit);
    } //while
    return found;
}
  
```



visited	toVisit
{}	A
{A}	C(3), B(7), D(9)
{A, C}	B(7), D(9), G(3+7)
{A, C, B}	D(9), G(10), E(7+10), F(7+15)
{A, C, B, D}	G(10), I(9+3), J(9+4), H(9+5), E(17), F(22)
{A, C, B, D, G}	I(12), J(13), H(14), E(17), F(22)
{A, C, B, D, G, I}	J(13), H(14), E(17), F(22), <b>K(9+3+7)</b>
{A, C, B, D, G, I, J}	H(14), E(17), F(22), <b>K(19)</b>
{A, C, B, D, G, I, J, H}	E(17), F(22), <b>K(19)</b>
{A, C, B, D, G, I, J, H, E}	F(22), <b>K(19)</b>
{A, C, B, D, G, I, J, H, E, F}	K(19)
{A, C, B, D, G, I, J, H, E, F, K}	{}



# USS in tree-based structures

## Uniform cost search – UCS

### □ Complexity analyse

- Time complexity
  - $b$  – ramification factor
  - $d$  - length (depth) of solution
  - $T(n) = 1 + b + b^2 + \dots + b^d \Rightarrow O(b^d)$
- Space complexity
  - $S(n) = T(n)$
- Completeness
  - yes – if solutions exists, then UCS finds it
- Optimality
  - Yes

### □ Advantages

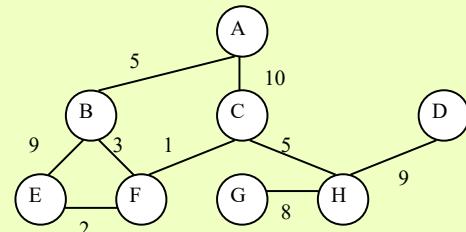
- Finding the minimum cost path to the objective node

### □ Disadvantages

- Exponential time and space complexity

### □ Applications

- Shortest path → Dijkstra algorithm



Vizitate deja	De vizitat
Φ	A(0)
A(0)	B(5), C(10)
A(0), B(5)	F(8), C(10), E(14)
A(0), B(5), F(8)	C(9), E(10)
A(0), B(5), F(8), C(9)	E(10), H(14)
A(0), B(5), F(8), C(9), E(10)	H(14)



# USS in tree-based structures

## depth-first search – DFS

### □ Basic elements

- Expand a child and depth search until
  - The final node is reached or
  - The node is a leaf
- Coming back in the most recent node that must be explored
- All the children of the current node are added in a **LIFO** list (**stack**)

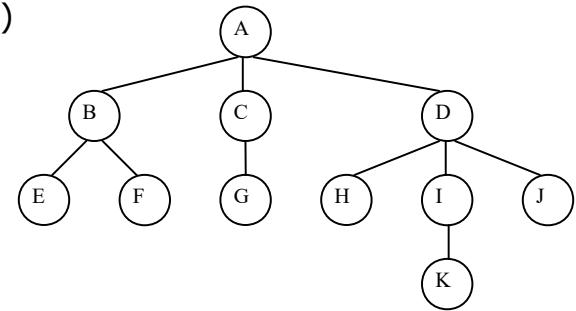
### □ Example

- Visiting order: A, B, E, F, C, G, D, H, I, K, J

### □ Algorithm

```

bool DFS(elem, list){
    found = false;
    visited = {};
    toVisit = {start};      //LIFO list
    while((toVisit != {}) && (!found)){
        node = pop(toVisit);
        visited = visited U {node};
        if (node== elem)
            found = true;
        else{
            aux = {};
            for all (unvisited) children of node do{
                aux = aux U {child};
            }
            toVisit = aux U toVisit;
        }
    } //while
    return found;
}
  
```



Vizitate deja	De vizitat
Φ	A
A	B, C, D
A, B	E, F, C, D
A, B, E	F, C, D
A, B, E, F	C, D
A, B, E, F, C	G, D
A, B, E, F, C, G	D
A, B, E, F, C, G, D	H, I, J
A, B, E, F, C, G, D, H	I, J
A, B, E, F, C, G, D, H, I	K, J
A, B, E, F, C, G, D, H, I, K	J
A, B, E, F, C, G, D, H, I, K, J	Φ

# USS in tree-based structures

## depth-first search – DFS



### Complexity analyse

- Time complexity
  - $b$  - ramification factor
  - $d_{max}$  – maximal length (depth) of explored tree
  - $T(n) = 1 + b + b^2 + \dots + b^{d_{max}} \Rightarrow O(b^{d_{max}})$
- Space complexity
  - $S(n) = b * d_{max}$
- Completeness
  - No → the algorithm does not end for infinite paths (there is no sufficient memory for all the nodes that are visited already)
- Optimality
  - No → depth search can find a longer path than the optimal one

### Advantages

- Finding the shortest path with minimal resources (recursive version)

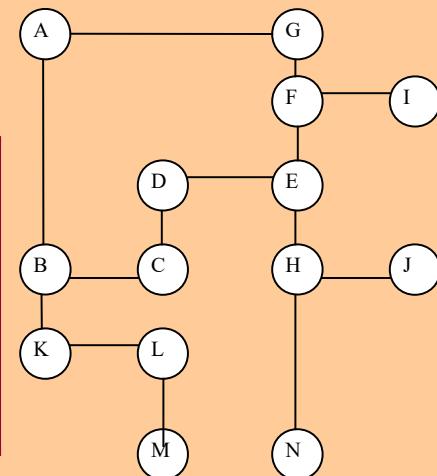
### Disadvantages

- Dead paths
  - Infinite cycles
  - Longer solution than the optimal one

### Applications

- Maze problem
- Identification of connex components
- Topological sorting
- Testing the graph planarity

A				G	
				F	I
		D		E	
B		C		H	J
K		L			
		M		N	



# USS in tree-based structures

## depth-first search – DFS



```

bool DFS_edges(elem, list){
    discovered = Φ;
    back = Φ;
    toDiscover = Φ;      //LIFO
    for (all neighbours of start) do
        toDiscover = toDiscover U {(start, neighbour)}
    found = false;
    visited = {start};
    while((toDiscover != Φ) && (!found)){
        edge = pop(toDiscover);
        if (edge.out !e visited){
            discovered = discovered U {edge};
            visited = visited U {edge.out}
            if (edge.out == end)
                found = true;
            else{
                aux = Φ;
                for all neighbours of edge.out do{
                    aux = aux U {(edge.out, neighbour)};
                }
                toDiscover = aux U toDiscover;
            }
        }
        else
            back = back U {edge}
    } //while
    return found;
}

```

Muchia	Muchii vizitare deja	Muchii de vizitat	înapoi	Noduri vizitate
	Φ	AB, AF	Φ	A
AB	AB	BC, BK, AF	Φ	A, B
BC	AB, BC	CD, BK, AF	Φ	A, B, C
CD	AB, BC, CD	DE, BK, AF	Φ	A, B, C, D
DE	AB, BC, CD, DE	EF, EH, BK, AF	Φ	A, B, C, D, E
EF	AB, BC, CD, DE, EF	FI, FG, EH, BK, AF	Φ	A, B, C, D, E, F
FI	AB, BC, CD, DE, EF, FI	FG, EH, BK, AF	Φ	A, B, C, D, E, F, I
FG	AB, BC, CD, DE, EF, FI, FG	GA, EH, BK, AF	Φ	A, B, C, D, E, F, I, G
GA	AB, BC, CD, DE, EF, FI, FG	EH, BK, AF	GA	A, B, C, D, E, F, I, G
EH	AB, BC, CD, DE, EF, FI, FG	HJ, HN, BK, AF	GA	A, B, C, D, E, F, I, G, H
HJ	AB, BC, CD, DE, EF, FI, FG, HJ	HN, BK, AF	GA	A, B, C, D, E, F, I, G, H, J
HN	AB, BC, CD, DE, EF, FI, FG, HI, HN	BK, AF	GA	A, B, C, D, E, F, I, G, H, N



# USS in tree-based structures depth-limited search – DLS

## □ Basic elements

- DFS + maximal depth that limits the search ( $d_{lim}$ )
- Solved the completeness problems of DFS

## □ Example

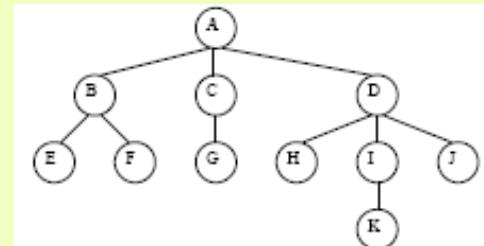
- $d_{lim} = 2$
- Visiting order: A, B, E, F, C, G, D, H, I, J

## □ Algorithm

```

bool DLS(elem, list, dlim) {
    found = false;
    visited = Φ;
    toVisit = {start}; //LIFO list
    while((toVisit != Φ) && (!found)) {
        node = pop(toVisit);
        visited = visited ∪ {node};
        if (node.depth <= dlim) {
            if (node == elem)
                found = true;
            else{
                aux = Φ;
                for all (unvisited) children of node do{
                    aux = aux ∪ {child};
                }
                toVisit = aux ∪ toVisit;
            } //if found
        } //if dlim
    } //while
    return found;
}

```



Vizitate deja	De vizitat
Φ	A
A	B, C, D
A, B	E, F, C, D
A, B, E	F, C, D
A, B, E, F	C, D
A, B, E, F, C	G, D
A, B, E, F, C, G	D
A, B, E, F, C, G, D	H, I, J
A, B, E, F, C, G, D, H	I, J
A, B, E, F, C, G, D, H, I	J
A, B, E, F, C, G, D, H, I, K, J	Φ



# USS in tree-based structures depth-limited search – DLS

## □ Complexity analyse

- Time complexity:
  - $b$  – ramification factor
  - $d_{lim}$  – limit of length (depth) allowed for the explored tree
  - $T(n) = 1 + b + b^2 + \dots + b^{d_{lim}} \Rightarrow O(b^{d_{lim}})$
- Space complexity
  - $S(n) = b * d_{lim}$
- Completeness
  - Yes, but  $\Leftrightarrow d_{lim} > d$ , where  $d$  = length (path) of optimal solution
- Optimality
  - No  $\rightarrow$  DLS can find a longer path than the optimal one

## □ Advantages

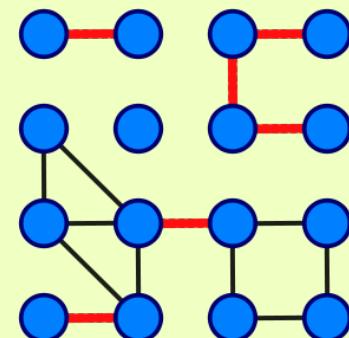
- Solves the completeness problems of DFS

## □ Disadvantages

- How to choose a good limit  $d_{lim}$ ?

## □ Applications

- Identification of bridges in a graph





# USS in tree-based structures

## iterative deepening depth search – IDDS

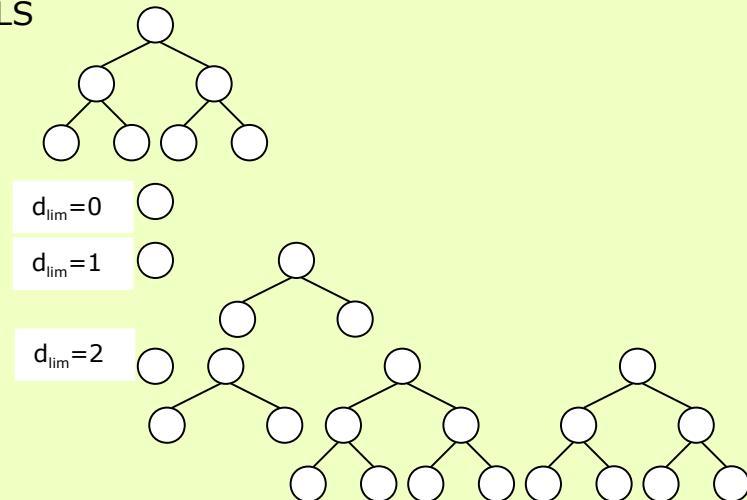
### □ Basic elements

- U DLS( $d_{lim}$ ), where  $d_{lim} = 1, 2, 3, \dots, d_{max}$
- Solves the identification of the optimal limit  $d_{lim}$  from DLS
- Usually, it works when:
  - The search space is large
  - The length (depth) of solution is known

### □ Example

### □ Algorithm

```
bool IDS(elem, list){  
    found = false;  
    dlim = 0;  
    while ((!found) && (dlim < dmax)) {  
        found = DLS(elem, list, dlim);  
        dlim++;  
    }  
    return found;  
}
```





# USS in tree-based structures

## iterative deepening depth search – IDDS

### □ Complexity analyse

- Time complexity:
  - $b^{d_{max}}$  nodes at depth  $d_{max}$  are expanded once =>  $1 * b^{d_{max}}$
  - $b^{d_{max}-1}$  nodes at depth  $d_{max}-1$  are expanded twice =>  $2 * (b^{d_{max}-1})$
  - ...
  - $b$  nodes at depth 1 are expanded  $d_{max}$  times =>  $d_{max} * b^1$
  - 1 node (the root) at depth 0 is expanded  $d_{max}+1$  times =>  $(d_{max}+1)*b^0$
- $$T(n) = \sum_{i=0}^{d_{max}} (i+1)b^{d_{max}-i} \Rightarrow O(b^{d_{max}})$$

- Space complexity
  - $S(n) = b * d_{max}$
- Completeness
  - yes
- Optimality
  - yes

### □ Advantages

- Requires linear memory
- The goal state is obtained by a minimal path
- Faster than BFS and DFS

### □ Disadvantages

- Requires to know the solution depth

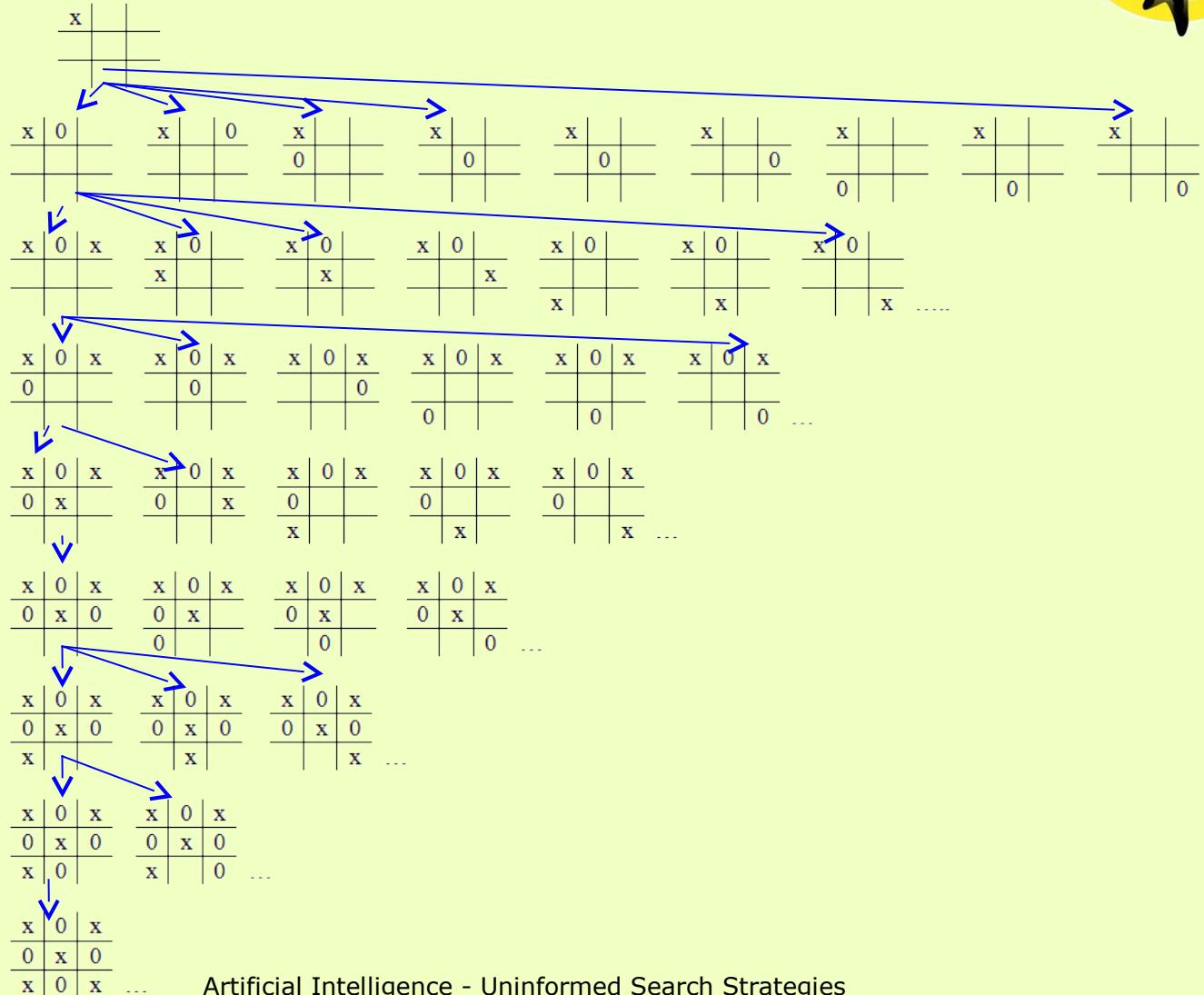
### □ Applications

- Tic tac toe game



# USS in tree-based structures

## iterative deepening depth search – IDDS





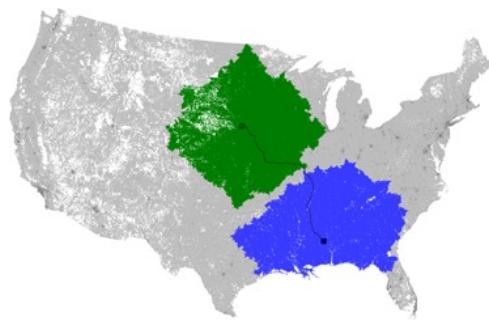
# USS in tree-based structures

## bi-directional search – BDS

### □ Basic elements

- 2 parallel search strategies
  - *forward*: from root to leaves
  - *backward*: from leaves to root
- that end when they meet
- any SS can be used in a direction
- Requires establishing:
  - the parents and the children of each node
  - the meeting point

### □ Example



### □ Algorithm

- Depend on the SS used



# USS in tree-based structures bi-directional search – BDS

## □ Complexity analyse

- Time complexity
  - $b$  – ramification factor
  - $d$  – solution length (depth)
  - $O(b^{d/2}) + O(b^{d/2}) \Rightarrow O(b^{d/2})$
- Space complexity
  - $S(n) = T(n)$
- Completeness
  - yes
- Optimality
  - yes

## □ Advantages

- Good time and space complexity

## □ Disadvantages

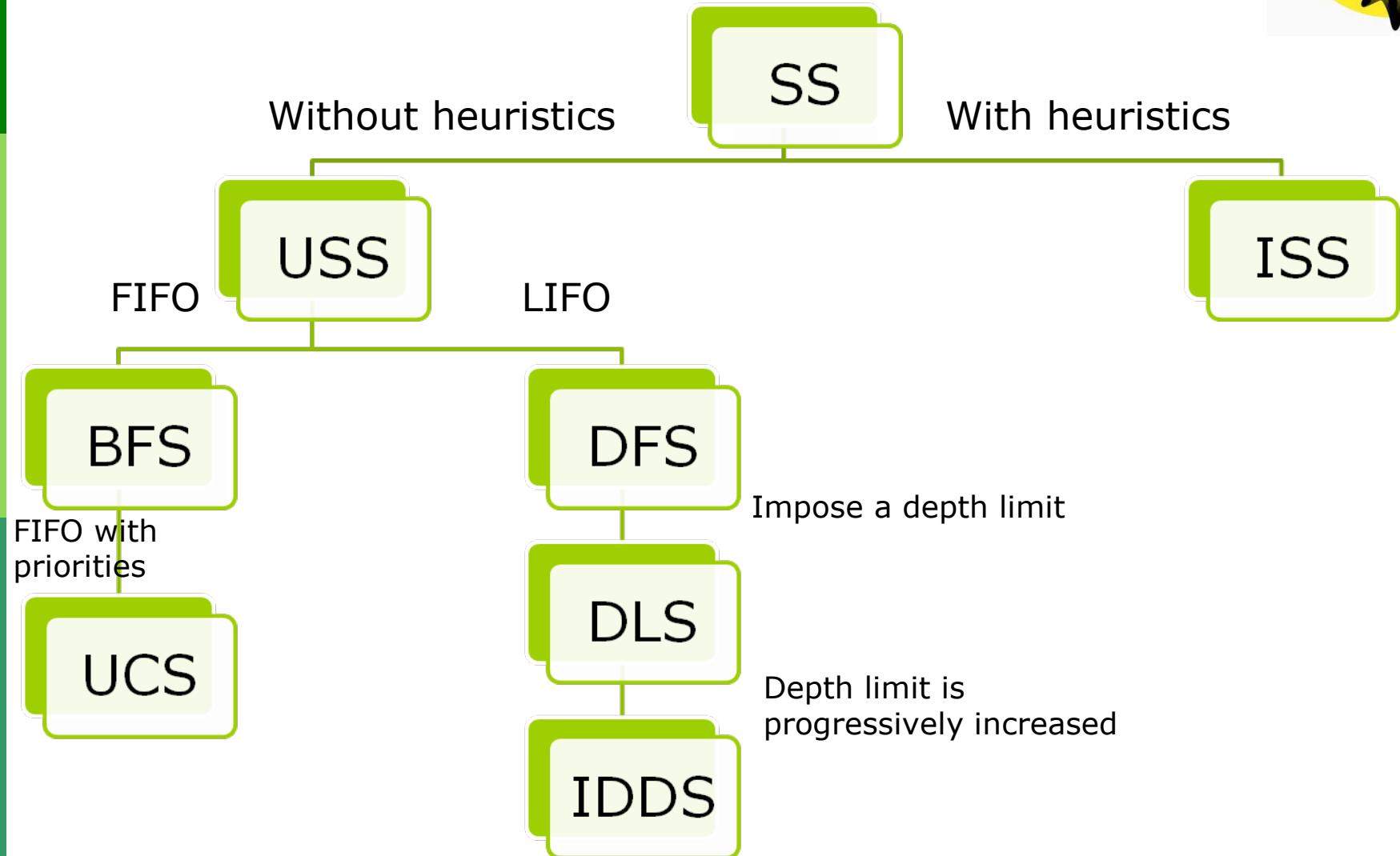
- Each state must be reversed
  - From had to tail
  - From tail to head
- Difficult to implement
- Identification of parents and children for all the nodes
- The final state must be known

## □ Applications

- Partitioning problem
- Shortest path



# USS in tree-based structures





# USS in tree-based structures

Comparison of performances

SS	Time complexity	Space complexity	Completeness	Optimality
BFS	$O(b^d)$	$O(b^d)$	Yes	Yes
UCS	$O(b^d)$	$O(b^d)$	Yes	Yes
DFS	$O(b^{d_{\max}})$	$O(b * d_{\max})$	No	No
DLS	$O(b^{d_{\text{lim}}})$	$O(b * d_{\text{lim}})$	Yes, if $d_{\text{lim}} > d$	No
IDS	$O(b^d)$	$O(b * d)$	Da	Yes
BDS	$O(b^{d/2})$	$O(b^{d/2})$	Yes	Yes

# ARTIFICIAL INTELLIGENCE



**Solving search problems**

Informed search strategies

Global and local

# Content

---

- Solving problem by search
  - Informed search strategies (ISS)
    - Global search strategies
      - Best first search
    - Local search strategies
      - Hill Climbing
      - Simulated Annealing
      - Tabu search



# Informed search strategies (ISS)

---

## ❑ Characteristics

- Based on specific information about the problem, trying to bound the search space by intelligent choosing the nodes to be explored
- An evaluation (heuristic) function sorts the nodes
- Specific to the problem

## ❑ Topology

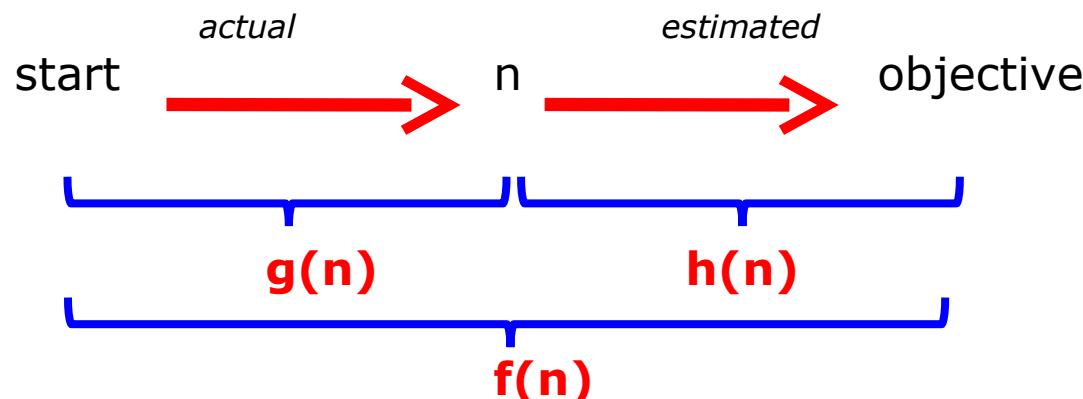
- Global search strategies
  - Best-first search
    - Greedy best-first search
    - A\* + versions of A\*
- Local search strategies
  - Tabu search
  - Hill climbing
  - Simulated annealing



# SS in tree-based structures

## □ Basic elements

- $f(n)$  – evaluation function for estimating the cost of a solution through node (state)  $n$
- $h(n)$  – evaluation function for estimating the cost of a solution path from node (state)  $n$  to the final node (state)
- $g(n)$  – evaluation function for estimating the cost of a solution path from the initial node (state) to node (state)  $n$
- $f(n) = g(n) + h(n)$





# ISS – Best first search

---

## □ Basic elements

- Best first search = first, the best element is processed
- Each state is evaluated by a function  $f$
- The best evaluated state is explored
- Example of a SS that depends on evaluation function
  - Uniform cost search (from USS)
    - $f = \text{path cost}$
  - ISSs use heuristic functions
- 2 possible BFS strategies
  - Expand the closest node to the objective state
  - Expand the best evaluated (best cost) node

## □ Example

- See next slides ☺



# ISS – Best first search

## □ Algorithm

```
bool BestFS(elem, list){  
    found = false;  
    visited =  $\Phi$ ;  
    toVisit = {start}; //FIFO sorted list (priority queue)  
    while((toVisit !=  $\Phi$ ) && (!found)){  
        if (toVisit ==  $\Phi$ )  
            return false  
        node = pop(toVisit);  
        visited = visited U {node};  
        if (node == elem)  
            found = true;  
        else  
            aux =  $\Phi$ ;  
        for all unvisited children of node do{  
            aux = aux U {child};  
        }  
        toVisit = toVisit U aux; //adding a node into the FIFO list based on its  
                           // evaluation (best one in the front of list)  
    } //while  
    return found;  
}
```



# ISS – best first search

## □ Complexity analyse

- Time complexity
  - $b$  – ramification factor
  - $d$  – maximal length (depth) of solution
  - $T(n) = 1 + b + b^2 + \dots + b^d \Rightarrow O(b^d)$
- Space complexity
  - $S(n) = T(n)$
- Completeness
  - No – infinite paths if the heuristic evaluates each node of the path as being the best selection
- Optimality
  - Possible - depends on heuristic

## □ Advantages

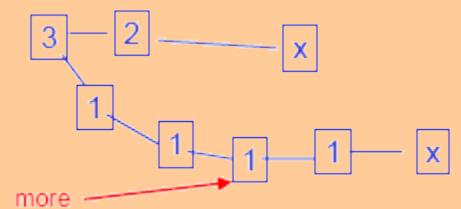
- Specific information helps the search
- Good speed to find the final state

## □ Disadvantages

- State evaluation → effort (computational, physic, etc)
- Some paths could seem to be good

## □ Applications

- *Web crawler (automatic indexer)*
- Games





# ISS – heuristic functions

---

- Etymology: *heuriskein* (*gr*)
  - *To find, to discover*
  - *Study of methods and rules of discovering and invention*
- Utility
  - Evaluation of the state potential (in the search space)
  - Estimation of path's cost from the current state to the final state
- Characteristics
  - Depends on the problem to be solved
  - New functions for new problems
  - A specific state is evaluated (instead of operators that map a state into another one)
  - Positive functions for each node  $n$ 
    - $h(n) \geq 0$  for all states  $n$
    - $h(n) = 0$  for final state
    - $h(n) = \infty$  for a state that starts a dead end



# ISS – heuristic functions

---

## □ Examples

- Missionary and cannibal problem
  - $h(n)$  – no of persons from initial river side
- 8-puzzle
  - $h(n)$  – no of pieces that are in wrong places
  - $h(n)$  – sum of Manhattan distance (of each piece relative to the final position)
- Travelling salesman problem
  - $h(n)$  – nearest neighbour ! ! !
- Pay a sum by using a minimal number of coins
  - $h(n)$  – choose the coin of best (large) value smaller than the sum to be paid



# ISS - Greedy

## □ Basic elements

- Evaluation function  $f(n) = h(n)$ 
  - Cost path estimation from the current state to the final one –  $h(n)$
  - cost minimization for the path that must be followed

## □ Example

- A,D,E,H

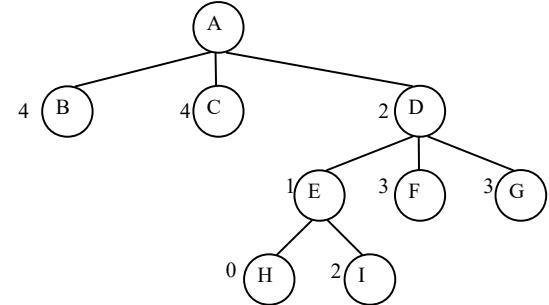
## □ Algorithm

```

bool BestFS(elem, list){
    found = false;
    visited = {};
    toVisit = {start};           //FIFO sorted list (priority queue
    while((toVisit != {}) && (!found)) {
        if (toVisit == {})
            return false
        node = pop(toVisit);
        visited = visited U {node};
        if (node == elem)
            found = true;
        else
            aux = {};
            for all unvisited children of node do{
                aux = aux U {child};
            }
        toVisit = toVisit U aux; //adding a node onto the FIFO list based on its evaluation  $h(n)$ 
                                //(best one in the front of list)

    } //while
    return found;
}

```



Vizitate deja	De vizitat
$\emptyset$	A
A	D, B, C
A, D	E, F, G, B, C
A, D, E	H, I, F, G, B, C
A, D, E, H	$\emptyset$



# ISS - Greedy

## □ Complexity analyse

- Time complexity → DFS
  - $b$  – ramification factor
  - $d^{\max}$  – maximal length (depth) of an explored tree
  - $T(n) = 1 + b + b^2 + \dots + b^{d^{\max}} \Rightarrow O(b^{d^{\max}})$
- Space complexity → BFS
  - $d$  - length (depth) of solution
  - $S(n) = 1 + b + b^2 + \dots + b^d \Rightarrow O(b^d)$
- Completeness
  - no
- Optimality
  - possible

## □ Advantages

- Quickly finds a solution (possible not-optimal), especially for small problems

## □ Disadvantages

- Sum of optimal local decisions ≠ global optimal decision
  - Ex. TSP

## □ Applications

- Planning problems
- Problem of partial sums
  - Coins
  - knapsack
- Puzzles
- Optimal paths in graphs

# ISS – A\*

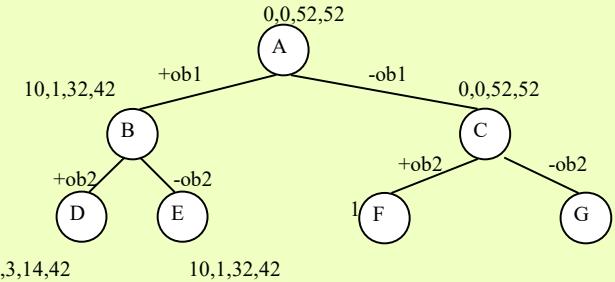


## □ Basic elements

- Combination of positive aspects from
  - Uniform cost search
    - Optimality and completeness
    - Sorted queues
  - Greedy search
    - Speed
    - Sorted based on evaluation
- Evaluation function  $f(n)$ 
  - Cost estimation of the path that passes through node  $n$   $f(n) = g(n) + h(n)$
  - $g(n)$  – cost function from the initial state to the current state  $n$
  - $h(n)$  – cost heuristic function from the current state to the final state
- Minimisation of the total cost for a path

## □ Example

- Knapsack problem – capacity  $W$ ,  $n$  objects  $(o_1, o_2, \dots, o_n)$  each of them having a profit  $p_i$ ,  $i=1,2,\dots,n$ 
  - Solution: for  $W = 5 \rightarrow o_1$  and  $o_3$
- $g(n) = \sum p_i$ , for selected objects  $o_i$
- $h(n) = \sum p_j$ , for not-selected objects and  $\sum w_j \leq W - \sum w_i$
- Fetched node is a tuple  $(p, w, p^*, f)$ , where:
  - $p$  – profit of selected objects (function  $g(n)$ )
  - $w$  – weight of selected objects
  - $p^*$  – maximal profit that can be obtained starting from the current state and taking into account the available space in the knapsack (function  $h(n)$ )



	$o_1$	$o_2$	$o_3$	$o_4$
$p_i$	10	18	32	14
$w_i$	1	2	4	3



# ISS – A\*

## Algorithm

```
bool BestFS(elem, list){  
    found = false;  
    visited = Φ;  
    toVisit = {start}; //FIFO sorted list (priority queue  
    while((toVisit != Φ) && (!found)) {  
        if (toVisit == Φ)  
            return false  
        node = pop(toVisit);  
        visited = visited U {node};  
        if (node == elem)  
            found = true;  
        else  
            aux = Φ;  
        for all unvisited children of node do{  
            aux = aux U {child};  
        }  
        toVisit = toVisit U aux; //adding a node onto the FIFO list  
                           // based on its evaluation  $f(n) = g(n) + h(n)$   
                           // (best one in the front of list)  
    } //while  
    return found;  
}
```

# ISS – A\*



## □ Complexity analyse

- Time complexity
  - $b$  – ramification factor
  - $d^{\max}$  – maximal length (depth) of an explored tree
  - $T(n) = 1 + b + b^2 + \dots + b^{d^{\max}} \Rightarrow O(b^{d^{\max}})$
- Space complexity
  - $d$  - length (depth) of solution
  - $T(n) = 1 + b + b^2 + \dots + b^d \Rightarrow O(b^d)$
- Completeness
  - Yes
- Optimality
  - yes

## □ Advantages

- Expands the fewest nodes of the tree

## □ Disadvantages

- Large amount of memory

## □ Applications

- Planning problems
- Problems of partial sums
  - Knapsack problem
  - Coin's problem
- Puzzles
- Optimal paths in graphs

# ISS – A\*

---



## Versions

- iterative deepening A\* (IDA\*)
- memory-bounded A\* (MA\*)
- simplified memory bounded A\* (SMA\*)
- recursive best-first search (RBFS)
- dynamic A\* (DA\*)
- real time A\*
- hierarchical A\*



# Solving problem by search

## Methods

- Step-by-step construction of solution
- Identification of a possible optimal solution
  - Algorithms

- Until now → **systematic** exploration of the search space
  - Eg. A\* →  $10^{100}$  states ≈ 500 binary variables
- Real-world problems can have 10 000 – 100 000 variables → require new algorithms that **locally** explore the search space
- Main idea:
  - Start with a state that does not respect some conditions and
  - Change the state for eliminating these violations
    - The search moves into a neighbourhood of the current solution
  - Such that the search will advance through the optimal state
- Iterative algorithms
  - Only a state is retained
  - Try to improve this state
- Intelligent version of brute force algorithm
- Search past is not retained

```
bool LS(elem, list){  
    found = false;  
    crtState = initState  
    while ((!found) && timeLimitIsNotExceeded) {  
        toVisit = neighbours(crtState)  
        if (best(toVisit) is better than crtState)  
            crtState = best(toVisit)  
        if (crtState == elem)  
            found = true;  
    } //while  
    return found;  
}
```



# Solving problem by search

## Methods

- Step-by-step construction of solution
- Identification of a possible optimal solution
  - Advantages
    - Simple implementation
    - Less memory
    - can find good solution in large (continuous) search spaces where other systematic algorithms can not be applied
  - Is useful when
    - Can be generated reasonable complete solutions
    - Can be selected a good starting point
    - Exist operators for solution changing
    - Exists a progress measure (for evaluating how the search advances)
    - Exists an evaluation function for a possible solution





# Local search strategies (LSS)

## □ Typology

- Simple local search – a single neighbour state is retained
  - Hill Climbing -> chooses the best neighbour
  - Simulated Annealing -> probabilistically chooses the best neighbour
  - Tabu search -> retains the recent visited solutions
- Beam local search – more states (population) are retained
  - Evolutionary Algorithms
  - Particle swarm optimisation
  - Ant colony optimisation



# Local search strategies

## □ Simple local search

### ■ Special elements:

- Solution representation
- Evaluation of a possible solution
- Neighbourhood of a solution
  - How a neighbour solution is defined/generated
  - How neighbour solutions are identified:
    - Randomly
    - Systematically
- How a possible solution is accepted
  - First neighbour of the current solution better than the current solution
  - Best neighbour of the current solution better than the current solution
  - Best neighbour of the current solution weaker than the current solution
  - A random neighbour of the current solution

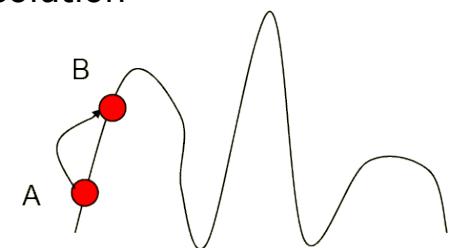
Depends on problem

# Local search strategies – Hill climbing (HC)



## □ Basic elements

- Climbing a foggy mountain by an amnesiac hiker :D
- Continuous moving to better values (larger → mountain climbing)
- Search advances to improved states until an optimal one is identified
- How a possible solution is accepted
  - Best neighbour of the current solution better than the current solution
- 
- Improvement by
  - Maximisation of state's quality → *steepest ascent HC*
  - Minimisation of state's quality → *gradient descent HC*
- HC ≠ *steepest ascent/gradient descent* (SA/GD)
  - HC optimises  $f(x)$  with  $x \in \mathbb{R}^n$  by changing an element of  $x$
  - SA/GD optimises  $f(x)$  with  $x \in \mathbb{R}^n$  by changing all the elements of  $x$

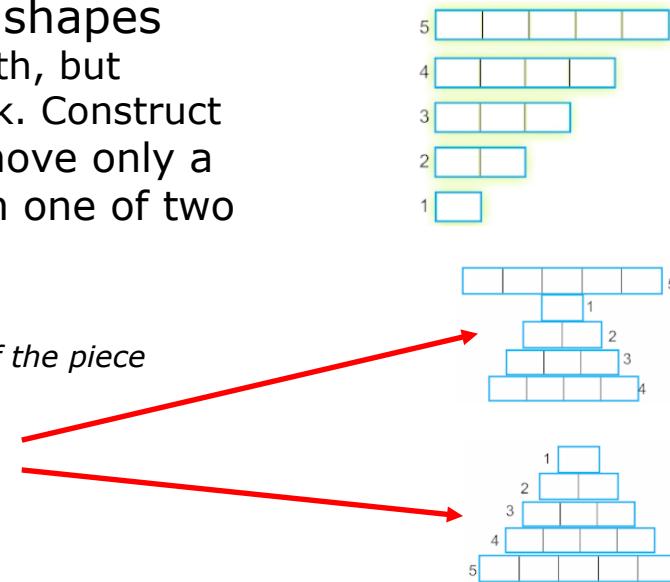


# Local search strategies – Hill climbing (HC)



## Example

- Construct tours from different geometrical shapes
  - We have  $n$  rectangular pieces (of the same width, but different lengths) that are overlapped in a stack. Construct a stable tour of all pieces such that at each move only a piece is moved from the top of the stack (on one of two supplementary stacks).
  - Solution representation
    - State  $x$  – vector of  $n$  pairs  $(i,j)$ , where  $i$  is the index of the piece ( $i=1,2,\dots,n$ ) and  $j$  is the index of the stack ( $j=1,2,3$ )
    - Initial state – vector of the initial tour
    - Final state – vector of the final tour
  - State evaluation
    - $f_1 = \#$  of correctly located pieces  $\rightarrow$  maximisation
      - Conformably tot the final tour –  $f_1 = n$
    - $f_2 = \#$  of wrongly located pieces  $\rightarrow$  minimisation
      - Conformably tot the final tour –  $f_2 = 0$
    - $f = f_1 - f_2 \rightarrow$  maximization
  - Neighbourhood
    - Possible moves
      - Move a piece  $i$  from stack  $j_1$  on stack  $j_2$
  - How a possible solution is accepted
    - Best neighbour of the current solution better than the current solution



# Local search strategies – Hill climbing (HC)

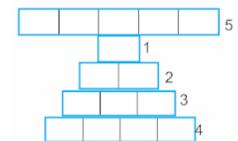


## □ Example

### ■ Iteration 1

#### □ current state = initial state:

- $x = s_1 = ((5,1), (1,1), (2,1), (3,1), (4,1))$
- Pieces 1, 2 and 3 are correctly located
- Pieces 4 and 5 are wrongly located
- $f(s_1) = 3 - 2 = 1$



#### □ $x^* = x$

#### □ Neighbours of current state $x$ – a single one → piece 5 moves on stack 2 →

- $s_2 = ((1,1), (2,1), (3,1), (4,1), (5,2))$
- $f(s_2) = 4 - 1 = 3 > f(x) \rightarrow x = s_2$



# Local search strategies – Hill climbing (HC)



## Example

### Iteration 2

- Current state  $x = ((1,1), (2,1), (3,1), (4,1), (5,2))$ 
  - $f(x) = 3$
- Neighbours of the current state – 2 neighbours:
  - Piece 1 moves on stack 2  $\rightarrow s_3 = ((2,1), (3,1), (4,1), (1,2), (5,2)) \rightarrow f(s_3) = 3-2=1 < f(x)$



- Piece 1 moves on stack 3  $\rightarrow s_4 = ((2,1), (3,1), (4,1), (5,2), (1,3)) \rightarrow f(s_4) = 3-2=1 < f(x)$
- There is no neighbour better than  $x \rightarrow$  stop
- $x^* = x = ((1,1), (2,1), (3,1), (4,1), (5,2))$
- But  $x^*$  is a local optimum just (not a global one)



# Local search strategies – Hill climbing (HC)



## □ Example

- Construct tours from different geometrical shapes - other solution
  - State evaluation
    - $f_1$  = sum of stack's height whose all pieces are correctly located (final tour  $f_1 = 10$ ) → maximisation
    - $f_2$  = sum of stack's height whose pieces are wrongly located (final tour  $f_2=0$ ) → minimisation
    - $f = f_1 - f_2 \rightarrow$  maximisation
  - Neighbourhood
    - Possible moves
      - Move a piece  $i$  from stack  $j_1$  on stack  $j_2$

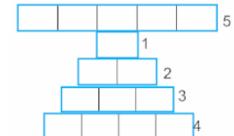
# Local search strategies – Hill climbing (HC)



## □ Example

### ■ Iteration 1

- Current state  $x$  = initial state  $s_1 = ((5,1), (1,1), (2,1), (3,1), (4,1))$ 
  - All pieces are wrongly located  $\rightarrow f_1 = 0, f_2 = 3+2 + 1 + 0 + 4 = 10$
  - $f(s_1) = 0 - 10 = -10$



- $x^* = x$

- Neighbours of current state  $x$  – a single one  $\rightarrow$  piece 5 is moved on stack 2  $\rightarrow s_2 = ((1,1), (2,1), (3,1), (4,1), (5,2))$



- $f(s_2) = 0 - (3+2+1+0) = -6 > f(x) \rightarrow x = s_2$

# Local search strategies – Hill climbing (HC)



## Example

### Iteration 2

- Current state  $x = ((1,1), (2,1), (3,1), (4,1), (5,2))$ 
  - $f(x) = -6$
- Neighbours of the current state – two neighbours:
  - Piece 1 is moved on stack 2  $\rightarrow s_3 = ((2,1), (3,1), (4,1), (1,2), (5,2)) \rightarrow f(s_3) = 0 - (0+2+3+0) = -5 > f(x)$ A diagram of a 5x5 grid of slots. Stacks 1 through 4 are each 3 units high, while stack 5 is 2 units high. Stack 1 is at position (1,1), stack 2 at (2,1), stack 3 at (3,1), stack 4 at (4,1), and stack 5 at (5,2). The stacks are represented by colored rectangles.  
$$f(s_3) = 0 - (0+2+3+0) = -5 > f(x)$$
  - Piece 1 is moved 3  $\rightarrow s_4 = ((2,1), (3,1), (4,1), (5,2), (1,3)) \rightarrow f(s_4) = 0 - (1+2+1) = -4 > f(x)$ A diagram of a 5x5 grid of slots. Stacks 1 through 4 are each 3 units high, while stack 5 is 2 units high. Stack 1 is at position (1,1), stack 2 at (2,1), stack 3 at (3,1), stack 4 at (4,1), and stack 5 at (5,2). In this variation, stack 1 has moved to position (1,3).  
$$f(s_4) = 0 - (1+2+1) = -4 > f(x)$$
- Best neighbour of  $x$  is  $s_4 \rightarrow x = s_4$

### Iteration 3

- ...

### Local optima are avoided

# Local search strategies – Hill climbing (HC)



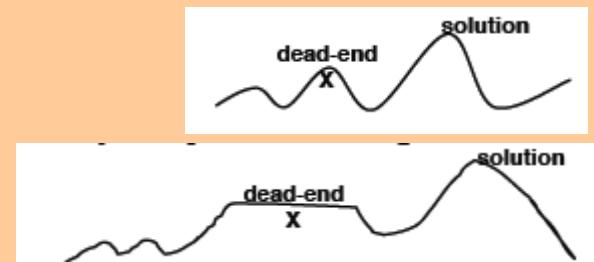
## □ Algorithm

```
Bool HC(S) {  
    x = s1      //initial state  
    x*=x        // best solution (found until now)  
    k = 0        // # of iterations  
    while (not termination criteria) {  
        k = k + 1  
        generate all neighbours of x (N)  
        Choose the best solution s from N  
        if f(s) is better than f(x) then x = s  
        else stop  
    } //while  
    x* = x  
    return x*;  
}
```

# Local search strategies – Hill climbing (HC)



- **Search analyse**
  - Convergence to local optima
- **Advantages**
  - Simple implementation -> solution approximation (when the real solution is difficult or impossible to find)
    - Eg. TSP with many towns
  - Does not require memory (does not come back into the previous state)
- **Disadvantages**
  - Evaluation function is difficult to be approximated
  - If a large number of moves are executed, the algorithm is inefficient
  - If a large number of moves are executed, the algorithm can block
    - In a local optimum
    - On a plateau – evaluation is constant
    - On a peak – a skip of more steps can help the search
- **Applications**
  - Cannibal's problem
  - 8-puzzle, 15-puzzle
  - TSP
  - Queen's problem



# Local search strategies – Hill climbing (HC)

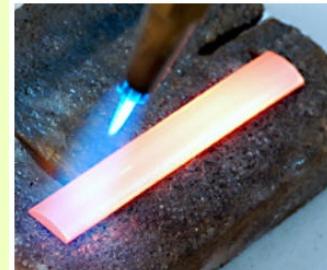
---



## ❑ Versions

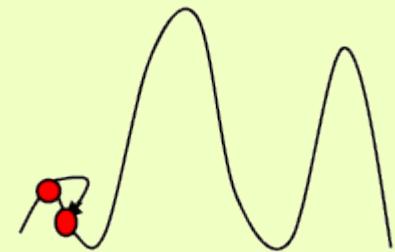
- Stochastic HC
  - ❑ The next state is randomly selected
- First-choice HC
  - ❑ Randomly generation of successors until a new one is identified
- Random-restart HC → *beam local search*
  - ❑ Restart the search from a randomly initial state when the search does not advance

# Local search strategies – Simulated Annealing

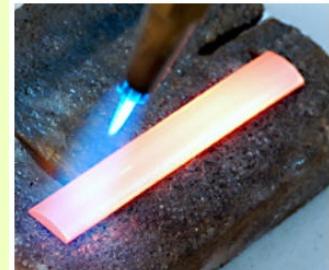


## □ Basic elements

- Inspired by physical process modelling
  - Metropolis et al. 1953, Kirkpatrick et al. 1982;
- Successors of the current state are randomly selected
  - If a successor is better than the current state
    - It becomes the new current state
    - Otherwise, it is retained by a given probability
- Weak moves are allowed with a given probability  $p$ 
  - Escape from local optima
- Probability  $p = e^{-\Delta E/T}$ 
  - Depends on difference (energy)  $\Delta E$
  - Is modelled by a temperature parameter  $T$
- The frequency of weak moves and their size gradually decrease when  $T$  is decreasing
  - $T = 0 \rightarrow$  hill climbing
  - $T \rightarrow \infty \rightarrow$  weak moves are frequently performed
- Optimal solution is identified only if the temperature slowly decreases
- How a possible solution is accepted
  - A random neighbour of the current solution better than the current solution or
  - Probabilistic, a random neighbour of the current solution weaker than the current solution

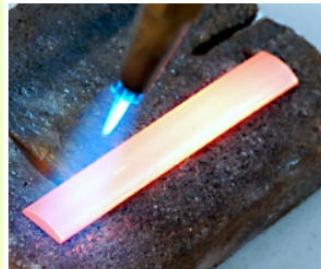


# Local search strategies – Simulated Annealing



## □ Example – 8-queen's problem

- Statement
  - Put 8 queens on a chessboard such there are no attacks between queens
- Solution representation
  - State  $x$  – permutation of  $n$  elements  $x = (x_1, x_2, \dots, x_n)$ , where  $x_i$  – line where the queen of column  $j$  is located
    - There are no attacks on lines or on columns
    - It is possible to find diagonal attacks
  - Initial state – a random permutation
  - Final state – a permutation without attacks
- Evaluation function for a state
  - $F$  – sum of attacked queens by each queen  $\rightarrow$  minimisation
- Neighbourhood
  - Possible moves
    - Move a queen from a line to a new line (swap 2 elements from permutation)
- How a possible solution is accepted
  - A random neighbour of the current solution
    - better than the current solution or
    - Weaker than the current solution – by a probability  $P(\Delta E) = e^{-\frac{\Delta E}{T}}$ , where
      - $\Delta E$  – energy (evaluation) difference of two states
      - $T$  – temperature,  $T(k) = 100/k$ , where  $k$  is the iteration number



# Local search strategies – Simulated Annealing

## Example – 8-queen's problem

### Iteration 1 ( $k = 1$ )

- Current state  $x$  = initial state

$$s_1 = (8, 5, 3, 1, 6, 7, 2, 4)$$

- $f(s_1) = 1+1 = 2$

- $x^* = x$

- $T = 100/1 = 100$

- A neighbour of current state  $x \rightarrow$  queen of line 5 is swapped by queen of line 7

$\rightarrow s_2 = (8, 7, 3, 1, 6, 5, 2, 4)$

- $f(s_2) = 1+1+1=3 > f(x)$

- $\Delta E = f(s_2) - f(s_1) = 1$

- $P(\Delta E) = e^{-1/100}$

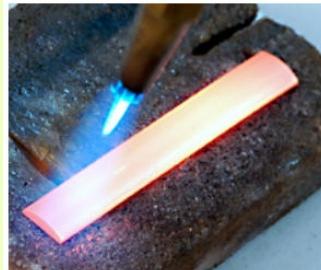
- $r = \text{random}(0,1)$

- $\text{if } r < P(\Delta E) \rightarrow x = s_2$

	a	b	c	d	e	f	g	h	
1	Q				Q				1
2					Q				2
3	Q								3
4									4
5	Q								5
6						Q			6
7					Q				7
8	Q								8
	a	b	c	d	e	f	g	h	

	a	b	c	d	e	f	g	h	
1	Q			Q					1
2			Q						2
3				Q					3
4					Q				4
5					Q				5
6						Q			6
7		Q							7
8			Q						8
	a	b	c	d	e	f	g	h	

# Local search strategies – Simulated Annealing



## □ Algorithm

```
bool SA(S){  
    x = s1           //initial state  
    x*=x           // best solution found until a given moment  
    k = 0           // iteration number  
    while (not termination criteria){  
        k = k + 1  
        generate a neighbour s of x  
        if f(s) is better than f(x) then x = s  
        else  
            pick a random number r (in (0,1) range)  
            if r < P( $\Delta E$ ) then x = s  
    } //while  
    x* = x  
    return x*;  
}
```

### □ Stop conditions

- The solution is found
- A number of iterations is reached
- The frozen temperature ( $T=0$ ) is hit

### □ How a small probability is chosen?

- $p = 0.1$
- $P$  decreases along the iterations
- $P$  decreases along the iterations and while the “error”  $|f(s) - f(x)|$  is increasing
  - $p = \exp(-|f(s) - f(x)|/T)$
  - Where  $T$  –temperature (that increases)
    - For a large  $T$  almost any neighbour is accepted
    - For a small  $T$ , only neighbours better than  $s$  are accepted
  - If the error is large, then the probability is small

# Local search strategies – Simulated Annealing



## □ Search analyse

- Convergence (complete, optimal) through global optima is slowly

## □ Advantages

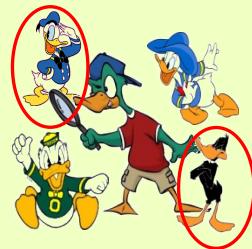
- Statistic-based algorithm → it is able to identified the optimal solution, but it requires many iterations
- Easy to implement
- Generally, if find a good (global) solution
- Can solve complex problems (with noise and many constraints)

## □ Disadvantages

- Slowly algorithm – convergence to solution takes a long time
  - Trade-off between the solution's quality and the time required to find it
- Depends on some parameters (temperature)
- The provided optimal solution could be local or global
- The solution's quality depends on the precision of variables involved in the algorithm

## □ Applications

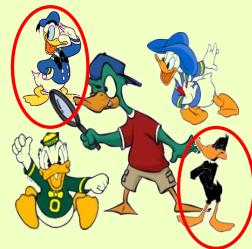
- Combinatorial optimisation problems → knapsack problem
- Design problems → digital circuits design
- Planning problems → production planning, tennis game planning



# Local search strategies – Tabu search

## □ Basic elements

- “Tabu” → things that cannot be touched because they are sacred
- Proposed in 1970 by F. Glover
- Main idea
  - starts with a state that violates some constraints and
  - Performs changes for eliminating them (the search moves into the best neighbour solution of the current solution) in order to identify the optimal solution
  - Retains
    - Current state
    - Visited states and performed moves (limited list of states that must be avoided)
  - How a possible solution is accepted
    - Best neighbour of the current solution better than the current solution and non-visited until that moment
- 2 important elements
  - Tabu moves (T) – determined by a non-Markov process that uses information obtained during last generations of search process
  - Tabu conditions – linear inequalities or logical links that depend on current solution
    - Influence the selection of tabu moves



# Local search strategies – Tabu search

## □ Example

### ■ Statement

- Pay a sum  $S$  by using  $n$  coins of values  $v_i$  as many as possible (each coin has  $b_i$  copies)

### ■ Solution representation

- State  $x$  – vector of  $n$  integers  $x = (x_1, x_2, \dots, x_n)$  with  $x_i \in \{0, 1, 2, \dots, b_i\}$
- Initial state – randomly

### ■ State evaluation

- $f_1 = S$  – total value of selected coins → minimisation
  - If the total value of coins  $> S$  → penalisation (eg. 500 units)
- $f_2 = \text{number of selected coins}$  → maximisation
- $f = f_1 - f_2 \rightarrow \text{minimisation}$

### ■ neighbourhood

- Possible moves
  - Including in the sum of  $j$  copies of coin  $i$  ( $\text{plus}_{i,j}$ )
  - Eliminating from the sum of  $j$  copies of coin  $i$  ( $\text{minus}_{i,j}$ )
- Tabu list retains performed moves of an iteration
  - - move = the added/eliminated coin



# Local search strategies – Tabu search

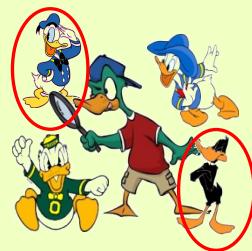
## Example

- $S = 500$ , penalisation= 500,  $n = 7$

<b><math>S=500</math></b>	<b><math>m_1</math></b>	<b><math>m_2</math></b>	<b><math>m_3</math></b>	<b><math>m_4</math></b>	<b><math>m_5</math></b>	<b><math>m_6</math></b>	<b><math>m_7</math></b>
$v_i$	10	50	15	20	100	35	5
$b_i$	5	2	6	5	5	3	10

<b>Stare curentă</b>	<b>Val. f</b>	<b>Listă tabu</b>	<b>Stări vecine</b>	<b>Mutări</b>	<b>Val. f</b>
2 0 1 0 0 2 1	384	$\emptyset$	2 0 1 3 0 2 1	plus <sub>4,3</sub>	321
			2 0 1 0 0 3 1	plus <sub>6,1</sub>	348
			0 0 1 0 0 2 1	minus <sub>1,2</sub>	406
2 0 1 3 0 2 1	321	plus <sub>4,3</sub>	2 0 1 3 5 2 1	plus <sub>5,5</sub>	316
			2 0 1 1 0 2 1	minus <sub>4,2</sub>	363
			2 1 1 3 0 2 1	plus <sub>2,1</sub>	270
2 1 1 3 0 2 1	270	plus <sub>4,3</sub> plus <sub>2,1</sub>	...		

- Final solution: 4 1 5 4 1 3 10 ( $f = -28$ )



# Local search strategies – Tabu search

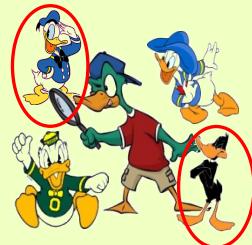
## □ Example

- $S = 500$ , penalisation = 500,  $n = 7$

<b><math>S=50</math></b>	$m_1$	$m_2$	$m_3$	$m_4$	$m_5$	$m_6$	$m_7$
$v_i$	10	50	15	20	100	35	5
$b_i$	5	2	6	5	4	3	10

Stare curentă	Val. f	Listă tabu	Stări vecine	Mutări	Val. f
2 0 1 0 0 2 1	384	$\emptyset$	1 0 1 4 0 2 1	minus <sub>1,1</sub> , plus <sub>4,4</sub>	311
			2 0 4 0 1 2 1	plus <sub>3,3</sub> , minus <sub>5,1</sub>	235
			2 0 1 0 4 2 6	plus <sub>5,4</sub> , plus <sub>7,5</sub>	450
2 0 4 0 1 2 1	235	plus <sub>3,3</sub> , minus <sub>5,1</sub>	2 0 5 0 5 2 1	plus <sub>3,1</sub> , plus <sub>5,4</sub>	315
			5 0 4 0 4 2 1	plus <sub>1,3</sub> , plus <sub>5,3</sub>	399
			2 2 4 0 5 2 1	plus <sub>2,2</sub> , plus <sub>5,4</sub>	739
2 0 4 0 1 2 1	235	plus <sub>3,3</sub> , minus <sub>5,1</sub>	...		

- Final solution: 4 1 5 4 1 3 10 ( $f = -28$ )

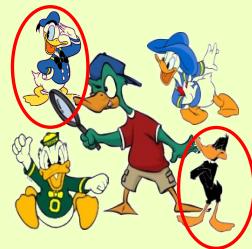


# Local search strategies – Tabu search

## □ Algorithm

```
bool TS(S){  
    Select x∈S    //S - search space  
    x*=x          //best solution until a moment  
    k = 0          //iteration number  
    T = ∅          //list of tabu moves  
    while (not termination criteria){  
        k = k + 1  
        generate a subset of solutions in the neighbourhood N-T of x  
        choose the best solution s from N-T and set x=s.  
        if f(x)<f(x*) then x*=x  
        update T with moves of generating x  
    } //while  
    return x*;  
}
```

- Stop conditions
  - Fix number of iterations
  - A given number of iterations without improvements
  - Sufficient proximity to the solution (if it is known)
  - Depletion unvisited elements of a neighbourhood



# Local search strategies – Tabu search

---

## □ **Search analyse**

- Quickly convergence to global optima

## □ **Advantages**

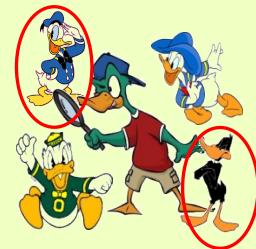
- The algorithm is general and can be easily implemented
- Quickly algorithm (can find in a short time the optimal solution)

## □ **Disadvantages**

- Identify the neighbours in continuous search spaces
- Large number of iterations
- Global optima identification is not guaranteed

# Local search strategies – Tabu search

---



## □ Applications

- Determination of tridimensional structure of proteins in amino acid sequences
- Traffic optimisation in communication networks
- Planning in production systems
- Network design in optical telecommunication
- Automatic routing of vehicles
- Graph problems (partitioning)
- Planning in audit systems



# Review

---

- ISS best first search
  - The best evaluated nodes are firstly expanded
  - Greedy ISS
    - Minimisation of the cost from the current state to the final state –  $h(n)$
    - Search time < USS
    - Non-complete
    - Non-optimal
  - A\* ISS
    - Minimisation of the cost from the initial state to the current state –  $g(n)$  and of the cost from the current state to the final state –  $h(n)$
    - Avoid to re-visit a state
    - Without supra-estimation of  $h(n)$
    - Large time and space complexity → depends on used heuristic
    - Complete
    - Optimal



# Review

## □ Local SS

### ■ Iterative algorithms

- Work with a possible solution → optimal solution
- Can block in local optima

	<b>Nex state selection</b>	<b>Acceptance criteria</b>	<b>Convergence</b>
HC	Best neighbor	Neighbor is better than current state	Local or global optima
SA	Random neighbor	Neighbor is better than current state or neighbor is weaker than current state (probabilistic acceptance)	Global optima (slowly)
TS	Best non-visited neighbor	Neighbor is better than current state	Global optima (quickly)

# ARTIFICIAL INTELLIGENCE



**Solving search problems**  
Evolutionary Algorithms

# Nature-inspired search

---

Best method for solving a problem

- Human brain
  - Has created the wheel, car, town, etc.
- Mechanism of evolution
  - Has created the human brain

Simulation of nature

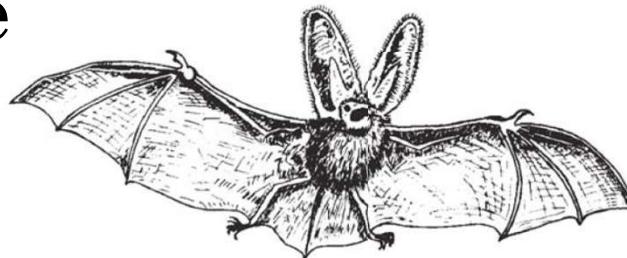
- By machines' help → the artificial neural networks simulate the brain
  - Flying vehicles, DNA computers, membrane-based computers
- By algorithms' help
  - Evolutionary algorithms simulate the evolution of nature
  - Particle Swarm Optimisation simulates the collective and social behaviour
  - Ant Colony Optimisation

# Basic elements

---

## ■ Simulation of nature

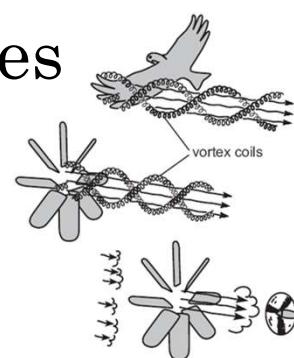
- Fly of bats
- Leonardo da Vinci – sketch of a flying machine



- Flies of birds and planes



- Flies of birds and wind-turbines



# Basic elements

---

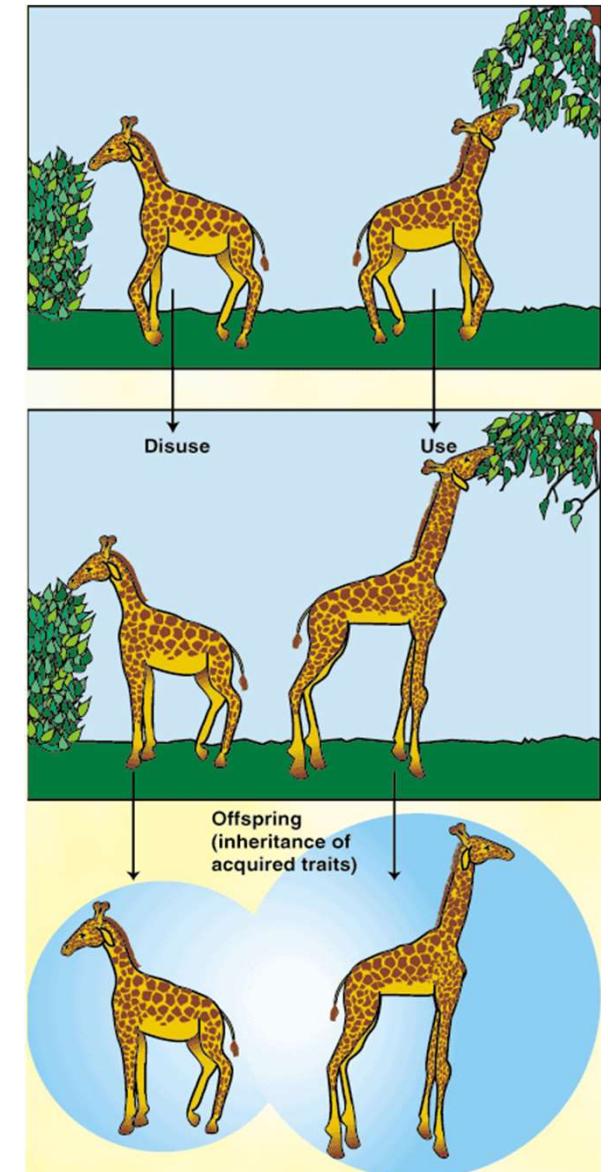
## Main characteristics of EAs

- Iterative and parallel processes
- Based on random search
- Bio-inspired – involve mechanisms as:
  - Natural selection
  - Reproduction
  - Recombination
  - Mutation

# Basic elements

## Historical points

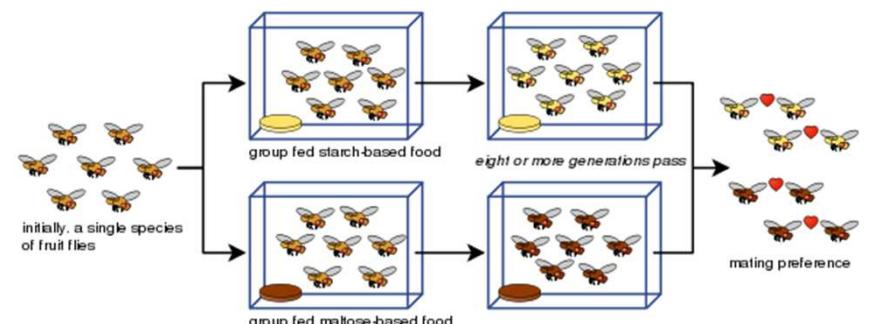
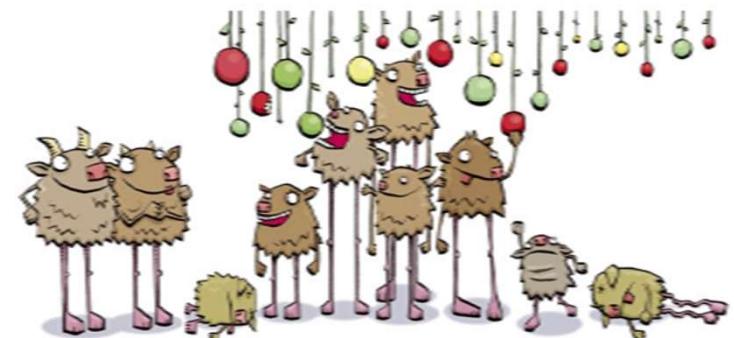
- Jean Baptise de Lamark (1744-1829)
  - Has proposed in 1809 an explanation for origin of species in the book *Zoological Philosophy*:
    - Needs of an organism determine the evolving characteristics
    - Useful characteristics could be transferred to offspring
  - *use and disuse law*



# Basic elements

## Historical points

- Charles Darwin (1807-1882)
  - In the book *Origin of Species* he proved that all the organisms have evolved based on:
    - Variation
      - Overproduction of offspring
    - Natural selection
      - Competition (generation of constant size)
      - Fitness survival
    - Reproduction
    - Occurrence of new species



# Basic elements

---

## Historical points

### Modern theory of evolution

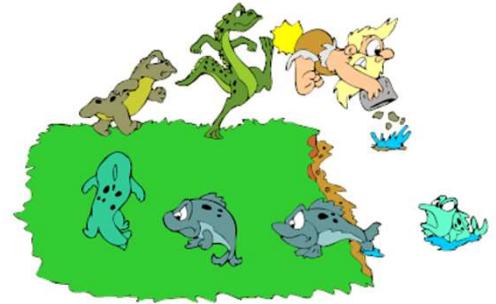
- Darwin's theory is improved by mechanism of genetic inheritance
- Genetic variance is produced by
  - Mutation and
  - Sexual recombination
- L. Fogel 1962 (San Diego, CA) → *Evolutionary Programming (EP)*
- J. Holland 1962 (Ann Arbor, MI) → *Genetic Algorithms (GAs)*
- I. Rechenberg & H.-P. Schwefel 1965 (Berlin, Germany) → *Evolution Strategies (ESs)*
- J. Koza 1989 (Palo Alto, CA) → *Genetic Programming (GP)*

# Basic elements

---

## Evolutionary metaphor

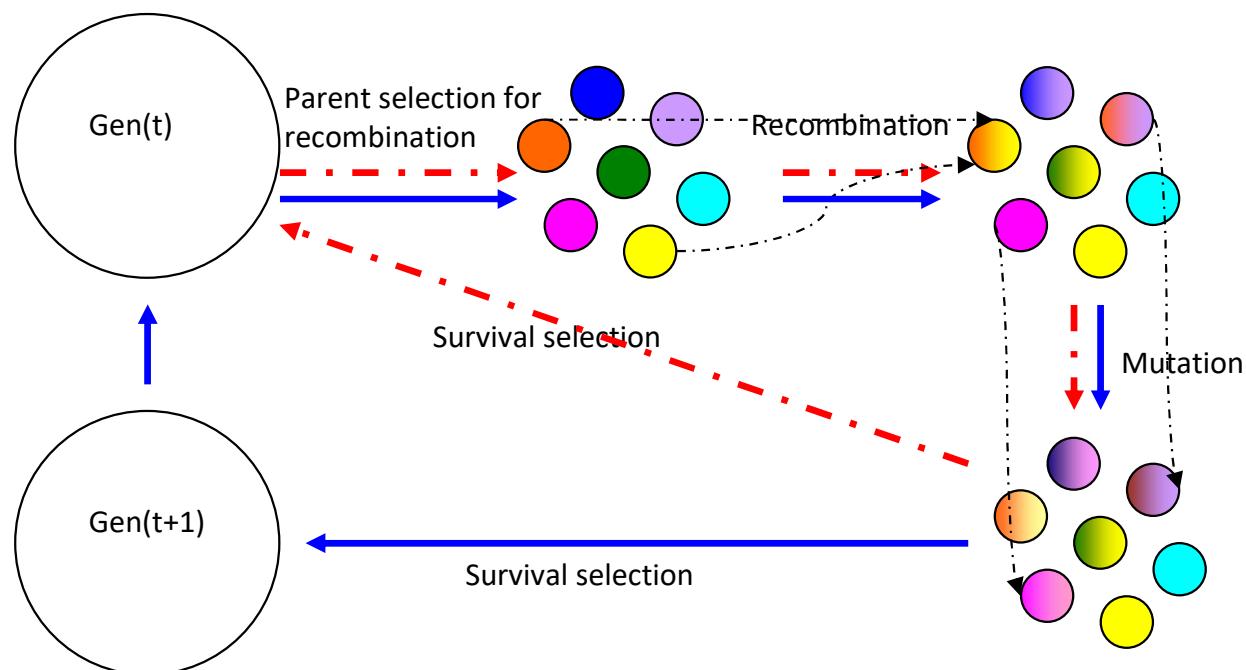
Natural evolution	↔	Problem solving
Individual	↔	Possible solution
Population	↔	Set of possible solutions
Chromosome	↔	Coding of a possible solution
Gene	↔	Part of coding
Fitness	↔	Quality
Crossover and Mutation	↔	Search operators
Environment	↔	Problem



# Algorithm

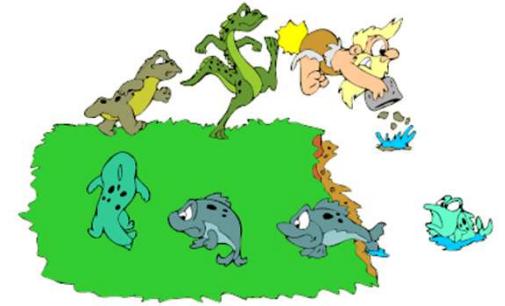
## General sketch of an EA

- Generational →
- Steady-state →



# Algorithm

---



## ▣ Design

- Chromosome representation
- Population model
- Fitness function
- Genetic operators
  - ▣ Selection
  - ▣ Mutation
  - ▣ Crossover
- Stop condition

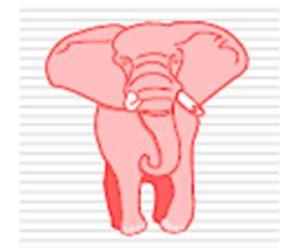
# Representation

---

- 2 levels of each possible solution

- External level → phenotype

- Individual – original object in the context of the problem
  - The possible solutions are evaluated here
  - Ant, knapsack, elephant, towns, ...



- Internal level → genotype

- Chromosome – code associated to an object

- Composed by genes, located in loci (fix positions) and having some values (alleles)

- The possible solutions are searched here

- One-dimensional vector (with numbers, bits, characters), matrix, ...

a d c a a c b

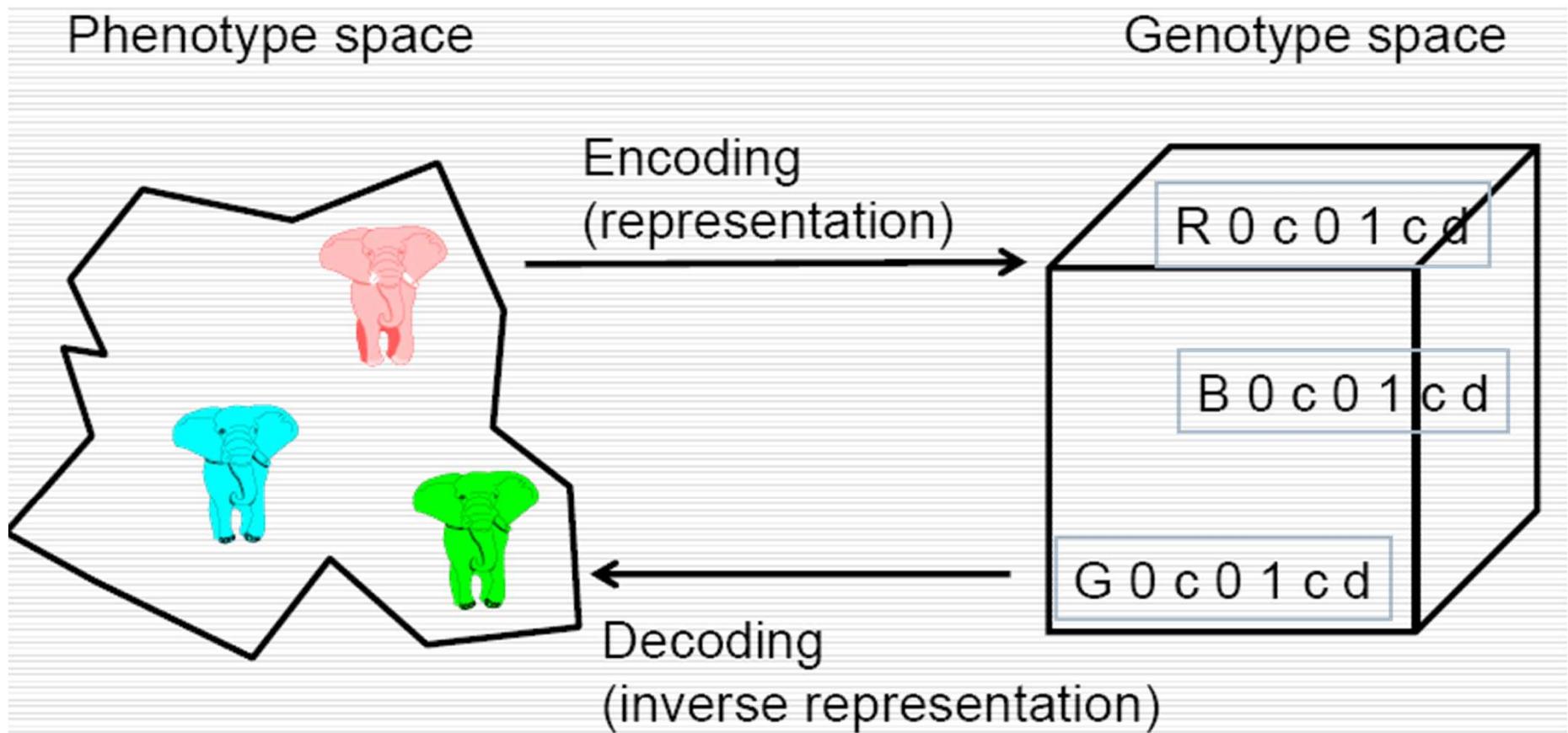
A horizontal row of green characters 'a', 'd', 'c', 'a', 'a', 'c', 'b' is enclosed in an orange rectangular border. The characters are aligned horizontally and centered within the border.

# Representation

---

■ Representation must be representative for:

- Problem
- Fitness function and
- Genetic operators



# Representation

---

## Linear

- Discrete

- Binary → knapsack problem

- Not-binary

- Integers

- Random → image processing

- Permutation → travelling salesman problem (TSP)

- Class-based → map colouring problem

- Continuous (real) → function optimization

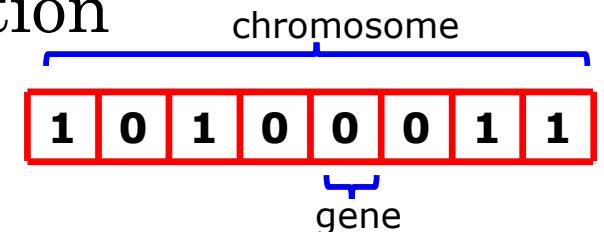
Tree-based → regression problems

# Representation

---

## Linear discrete and binary representation

- Genotype  
Bit-strings



# Representation

---

- Linear discrete and binary representation
  - Genotype
    - Bit-strings
  - Phenotype
    - Boolean elements
      - Ex. Knapsack problem – selected objects for the bag

Genotype	Phenotype
1   0   1   0   0   0   1   1	$ob_1 + ob_3 + ob_7 + ob_8$

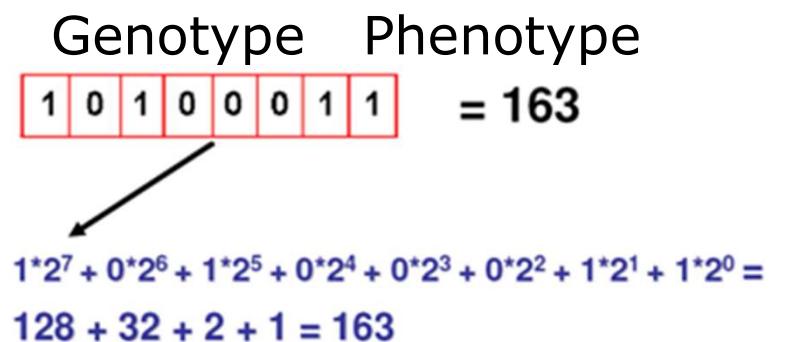
Objects 1, 3, 7 and 8 are selected

# Representation

---

- Linear discrete and binary representation
  - Genotype
    - Bit-strings
  - Phenotype
    - Boolean elements
      - Ex. Knapsack problem – selected objects for the bag

- Integers



# Representation

---

- Linear discrete and binary representation
  - Genotype
    - Bit-strings
  - Phenotype
    - Boolean elements
      - Ex. Knapsack problem – selected objects for the bag
- Integers
  - Real numbers from a range  
(ex. [2.5, 20.5])

Genotype	Phenotype								
<table border="1" style="border-collapse: collapse; width: 100%;"><tr><td style="width: 12.5%;">1</td><td style="width: 12.5%;">0</td><td style="width: 12.5%;">1</td><td style="width: 12.5%;">0</td><td style="width: 12.5%;">0</td><td style="width: 12.5%;">0</td><td style="width: 12.5%;">1</td><td style="width: 12.5%;">1</td></tr></table>	1	0	1	0	0	0	1	1	$= 13.9609$
1	0	1	0	0	0	1	1		

$x = 2.5 + \frac{163}{256} (20.5 - 2.5) = 13.9609$

# Representation

---

Transformation of real values from binary representation

- Let be  $z \in [x,y] \subseteq \mathbb{R}$  represented as  $\{a_1, \dots, a_L\} \in \{0, 1\}^L$
- Function  $[x,y] \rightarrow \{0,1\}^L$  must be inversely (a phenotype corresponds to a genotype)
- Function  $\Gamma: \{0,1\}^L \rightarrow [x,y]$  defines the representation

$$\Gamma(a_1, \dots, a_L) = x + \frac{y-x}{2^L - 1} \cdot \left( \sum_{j=0}^{L-1} a_{L-j} \cdot 2^j \right) \in [x, y]$$

- Remarks
  - $2^L$  values can be represented
  - L – maximum precision of solution
  - For a better precision → long chromosomes → slowly evolution

# Representation

---

Linear discrete non-binary integer random representation

- Genotype

- Vector of integers from a given range

- Phenotype

- Utility of numbers in the problem

- Ex. Pay a sum  $S$  by using different  $n$  coins

- Genotype → vector of  $n$  integers from range  $[0, S/\text{value of current coin}]$

- Phenotype → how many coins of each type must be considered

# Representation

---

Linear discrete non-binary integer permutation representation

- Genotype
  - Permutation of  $n$  elements ( $n$  – number of genes)
- Phenotype
  - Utility of permutation in problem
- Ex. Traveling Salesman Problem
  - Genotype → permutation of  $n$  elements
  - Phenotype → visiting order of towns (each town has associated a number from  $\{1,2,\dots,n\}$ )

# Representation

---

Linear discrete non-binary integer class-based representation

- Similarly to integer one, but labels are used instead numbers
- Genotype
  - Vector of labels from a given set
- Phenotype
  - Labels' meaning
- Ex. Map colouring problem
  - Genotype → vector of  $n$  colours ( $n$  – number of countries)
  - Phenotype → what colour has to be used for each country

# Representation

---

## Linear continuous (real) representation

- Genotype

Vector of real numbers

- Phenotype

Number meaning

- Ex. Function optimisation  $f:R^n \rightarrow R$

Genotype → more real numbers  $X=[x_1, x_2, \dots, x_n]$ ,  $x_i \in R$

Phenotype → values of function f arguments

# Representation

---

- Tree-based representation

- Genotype

- Trees than encode S-expressions
    - Internal nodes □functions (F)
      - Mathematical
        - Arithmetic operators
        - Boolean operators
      - Statements
        - Of a given programming language
        - Of other language type

- Leaf → terminals (T)

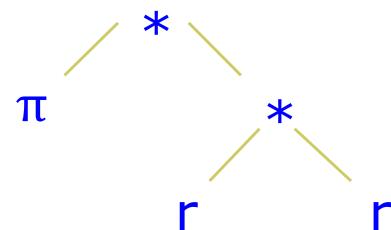
- Real or Boolean values, constants or variables
    - Sub-programs

- Phenotype

- Meaning of S-expressions

- Ex. Computing the circle area

$$\pi * r^2 \quad \longrightarrow$$



# Population

---

## Population – concept

- Aim

- Keeps a collection of possible solutions (candidate solutions)

- Repetitions are allowed

- Is entirely utilised during selection for recombination

- Properties

- (usually) fixed dimension  $\mu$

- Diversity

- Number of different fitnesses/phenotypes/genotypes

- Remarks

- Represents the basic unit that evolves

- The entire population evolves, not only the individuals!

# Population

---

## Population - initialisation

- Uniformly distributed in the search space (if it is possible)

### Binary strings

- Randomly generation of 0 and 1 with a 0.5 probability (fifty-fifth)

Arrays of real numbers uniformly generated (in a given range)

### Permutations

- Generation of identical permutation and making some changes

# Population

---

## Population - initialisation

- Uniformly distributed in the search space (if it is possible)

### Trees

- *Full* method – complete trees
  - Nodes of depth  $d < D_{\max}$  are randomly initialised by a function from function set  $F$
  - nodes of depth  $d = D_{\max}$  are randomly initialised by a terminal from the terminal set  $T$
- *Grow* method – incomplete trees
  - Nodes of depth  $d < D_{\max}$  are randomly initialised by an element from  $F \cup T$
  - nodes of depth  $d = D_{\max}$  are randomly initialised by a terminal from the terminal set  $T$
- *Ramped half and half* method
  - $\frac{1}{2}$  of population is initialised by *Full* methods
  - $\frac{1}{2}$  of population is initialised by *Grow* methods
  - By using different depths

# Population

---

## Population model:

- Generational EA

- Each generation creates  $\mu$  offspring

- Each individual survives a generation only

- Set of parents is totally replaced by set of offspring

- Steady-state EA

- Each generation creates a single offspring

- A single parent (the worst one) is replaced by the offspring

## Generation Gap

- Proportion of replaced population
- $1 = \mu/\mu$ , for generational model
- $1/\mu$ , for steady-state model

# Fitness function

---

## Aim

- Reflects the adaptation to environment
- Quality function or objective function
- Associates a value to each candidate solution
  - Consequences over selection → the more different values, the better

## Properties

- Costly stage
  - Unchanged individuals could not be re-evaluated

## Typology:

- Number of objectives
  - One-objective
  - Multi-objective → Pareto fronts
- Optimisation direction
  - Maximisation
  - Minimisation
- Degree of precision
  - Deterministic
  - Heuristic

# Fitness function

---

## Examples

- Knapsack problem
  - Representation → linear, discrete and binary
  - Fitness →  $\text{abs}(\text{knapsack's capacity} - \text{weight of selected objects}) \rightarrow \min$
- Problem of paying sum  $s$  by using different coins
  - Representation → linear, discrete and integer
  - Fitness →  $\text{abs}(\text{sum to be paid} - \text{sum of selected coins}) \rightarrow \min$
- TSP
  - Representation → linear, discrete, integer, permutation
  - Fitness → cost of path → min
- Numerical function optimization
  - Representation → linear, continuous, real
  - Fitness → value of function → min/max
- Computing the circle's area
  - Representation → tree-based
  - Fitness → sum of square errors (difference between the real value and the computed value for a given set of examples) → min



# Selection

---

## ❑ Aim:

- Gives more reproduction/survival chances to better individuals
  - ▣ Weaker individuals have chances also because they could contain useful genetic material
- Orients the population to improve its quality

## ❑ Properties

- Works at population-level
- Is based on fitness only (is independent to representation)
- Helps to escape from local optima (because its stochastic nature)



# Selection



- Aim
  - Parent selection (from current generation) for reproduction
  - Survival selection (from parents and offspring) for next generation
- Winner strategy
  - Deterministic – the best wins
  - Stochastic – the best has more chances to win
- Mechanism
  - Selection for recombination
    - Proportional selection (based on fitness)
    - Rank-based selection
    - Tournament selection
  - Survival selection
    - Age-based selection
    - Fitness-based selection



# Recombination selection

## Proportional selection (fitness-based selection) - PS

### Main idea

- Roulette algorithm for entire population
- Estimation of the copies # of an individual (selection pressure)

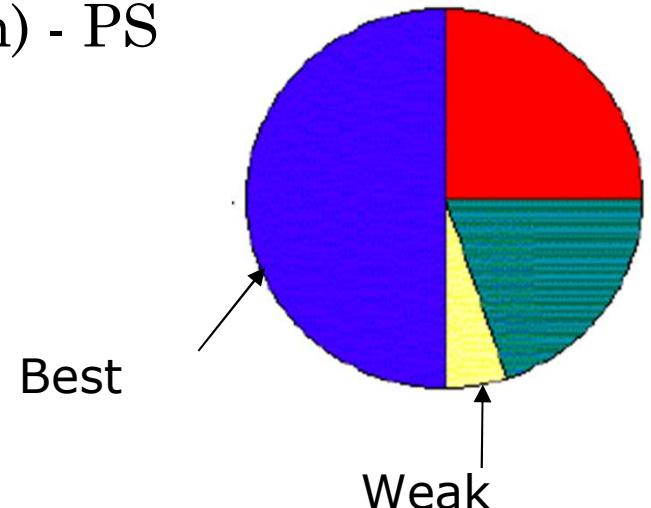
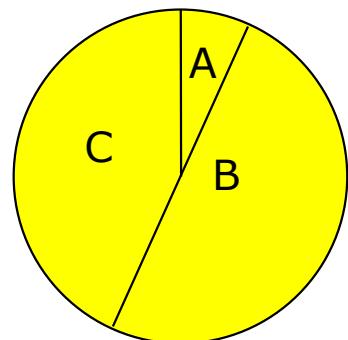
$$E(n_i) = \mu \frac{f(i)}{\langle f \rangle}, \text{ where:}$$

- $\mu$  = size of population,
- $f(i)$  = fitness of individual  $i$ ,
- $\langle f \rangle$  = mean fitness of population

### Better individuals

- Have more space on roulette
- Have more chances to be selected

### Ex. A population of $\mu = 3$ individuals



	$f(i)$	$P_{selPS}(i)$
A	1	$1/10=0.1$
B	5	$5/10=0.5$
C	4	$4/10=0.4$
Sum	10	1



# Recombination selection

## Proportional selection (fitness-based selection) – PS

### Advantages

- Simple algorithm

### Disadvantages

- Premature convergence

- Best chromosomes predispose to dominate the population

- Low selection pressure when fitness functions are very similar (at the end of a run)

- Real results are different to theoretical probabilistic distribution

- Works at the entire population level

### Solutions

- Fitness scaling

- Windowing

- $f(i) = f(i) - \beta^t$ , where  $\beta$  is a parameter that depends on evolution history
      - eg.  $\beta$  is the fitness of the weakest individual of current population (the  $t^{\text{th}}$  generation)

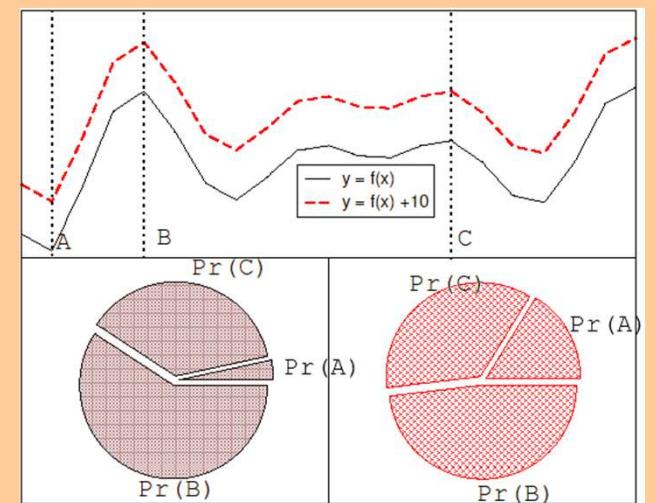
- Sigma scaling (Goldberg type)

- $f(i) = \max\{f(i) - (\langle f \rangle - c * \sigma_f), 0.0\}$ , where:
      - $C$  – a constant (usually, 2)
      - $\langle f \rangle$  – average fitness of population
      - $\sigma_f$  – standard deviation of population fitness

- Normalisation

- Starts by absolute (initial) fitnesses
    - Standardize these fitnesses such as the fitnesses:
      - Belong to  $[0,1]$
      - Best fitness is the smallest one (equal to 0)
      - Sum of them is 1

- Another selection mechanism





# Selection for recombination

## ❑ Ranking selection – RS

### ■ Main idea

- ❑ Sort the entire population based on fitness
  - Increases the algorithm complexity, but it is negligible related to the fitness evaluation
- ❑ Each individual receives a rank
  - Computes the selection probabilities based on these ranks
    - Best individual has rank  $\mu$
    - Worst individual has rank 1
- ❑ Tries to solve the problems of proportional selection by using relative fitness (instead of absolute fitness)



# Selection for recombination

## Ranking selection - RS

### Ranking procedures

Linear (LR)  $P_{lin\_rank}(i) = \frac{2-s}{\mu} + \frac{2i(s-1)}{\mu(\mu-1)}$

- $s$  – selection pressure
  - Measures the advantages of the best individual
  - $1.0 < s \leq 2.0$
  - In the generational algorithm  $s$  represents the copies number of an individual
- Ex. For a population of  $\mu=3$  individuals

	$f(i)$	$P_{selPS}(i)$	Rank	$P_{selLR}(i)$ for $s=2$	$P_{selRL}(i)$ for $s=1.5$
A	1	$1/10=0.1$	1	0.33	0.33
B	5	$5/10=0.5$	3	1.00	0.33
C	4	$4/10=0.4$	2	0.67	0.33
Sum	10	1			

Exponential (ER)  $P_{exp\_rank}(i) = \frac{1-e^{-i}}{c}$

- Best individual can have more than 2 copies
- $C$  – normalisation factor
  - Depends on the population size ( $\mu$ )
  - Must be choose such as the sum of selection probabilities to be 1



# Selection for recombination

---

- Ranking selection - RS
  - Advantages
    - Keep the selection pressure constant
  - Disadvantages
    - Works with the entire population
  - Solutions
    - Another selection procedure

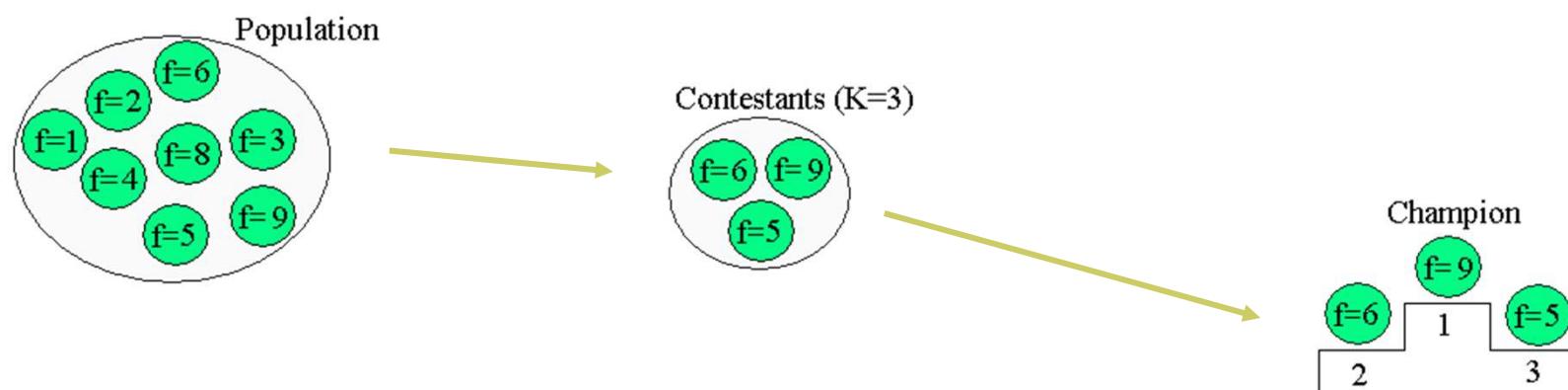


# Selection for recombination

## ■ Tournament selection

### ■ Main idea

- Chooses  $k$  individuals → sample of  $k$  individuals ( $k$  – tournament size)
- Selects the best individual of the sample
- Probability of sample selection depends on
  - Rank of individual
  - Sample size ( $k$ )
    - The larger  $k$  is, the greater selection pressure is
- Choosing manner – with replacement (steady-state model) or without replacement
  - Selection without replacement increases the selection pressure
- For  $k = 2$  the time required by the best individual to dominate the population is the same to that from linear ranking selection with  $s = 2 * p$ ,  $p$  – selection probability of the best individual from population





# Selection for recombination

---

## ?

### Tournament selection

#### ■ Advantages

- Does not work with the entire population
- Easy to implement
- Easy to control the selection pressure by using parameter k

#### ■ Disadvantages

- The real results of this selection are different to theoretical distribution (similarly to roulette selection)

# Survival selection



- Survival selection (selection for replacement)
  - Based on age
    - Eliminates the oldest individuals
  - Based on fitness
    - Proportional selection
    - Ranking selection
    - Tournament selection
    - Elitism
      - Keep the best individuals from a generation to the next one (if the offspring are weaker than parents, then keep the parents)
    - GENITOR (replaces the worst individual)
      - Elimination of the worst  $\lambda$  individuals



# Variation operators

---

## ② Aim :

- Generation of new possible solutions

## ③ Properties

- Works at individual level
- Is based on individual representation (fitness independent)
- Helps the exploration and exploitation of the search space
- Must produce valid individuals

## ④ Typology

- Arity criterion
  - ② Arity 1 → mutation operators
  - ② Arity > 1 → recombination/crossover operators



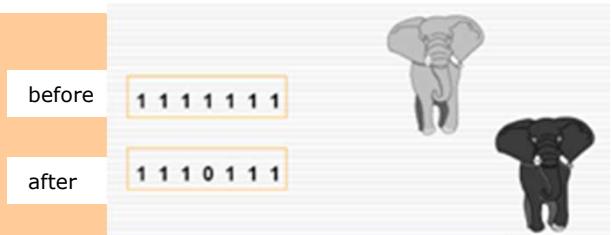
# Mutation

## ¶ Aim

- Reintroduces in population the lost genetic material
- Unary search operator (continuous space)
- Introduces the diversity in population (discrete space)

## ¶ Properties

- Works at genotype level
- Based on random elements
- Responsible to the exploration of promising regions of the search space
- Responsible to escape from local optima
- Must introduce small and stochastic changes for an individual
- Size of mutation must be controllable
- Can probabilistic take place (by a given probability  $p_m$ ) at the gene level



# Mutation



- Binary representation
  - ▣ Strong mutation – bit-flipping
  - ▣ Weak mutation
- Integer representation
  - ▣ Random resetting
  - ▣ Creep mutation
- Permutation representation
  - ▣ Insertion mutation
  - ▣ Swap mutation
  - ▣ Inverse mutation
  - ▣ scramble mutation
  - ▣ K-opt mutation
- Real representation
  - ▣ Uniform mutation
  - ▣ Non-uniform mutation
    - Gaussian mutation
    - Cauchy mutation
    - Laplace mutation
- Tree-based representation → future lecture
  - ▣ Grow mutation
  - ▣ Shrink mutation
  - ▣ Switch mutation
  - ▣ Cycle mutation
  - ▣ Koza mutation
  - ▣ Mutation for numerical terminals



# Mutation (binary representation)

- A chromosome  $c = (g_1, g_2, \dots, g_L)$  becomes  $c' = (g'_1, g'_2, \dots, g'_L)$ , where  $g_i, g'_i \in \{0, 1\}$ , for  $i = 1, 2, \dots, L$ .
- Strong mutation – *bit flipping*
  - Main idea
    - Changes by probability  $p_m$  (mutation rate) all the genes in their complement
      - $1 \rightarrow 0$
      - $0 \rightarrow 1$
  - Ex. A chromosome of  $L = 8$  genes,  $p_m = 0.1$





# Mutation (binary representation)

- ❑ A chromosome  $c = (g_1, g_2, \dots, g_L)$  becomes  $c' = (g'_1, g'_2, \dots, g'_L)$ , where  $g_i, g'_i \in \{0,1\}$ , for  $i = 1, 2, \dots, L$

- Weak mutation

- Main idea

- Changes by probability  $p_m$  (mutation rate) some of the genes in 0 or 1

- $1 \rightarrow 0/1$
      - $0 \rightarrow 1/0$

- Eg. A chromosome of  $L = 8$  genes,  $p_m = 0.1$





# Mutation (integer representation)

- ❑ A chromosome  $c = (g_1, g_2, \dots, g_L)$  becomes  $c' = (g'_1, g'_2, \dots, g'_L)$ , where  $g_i, g'_i \in \{val_1, val_2, \dots, val_k\}$  for  $i = 1, 2, \dots, L$ .

- ❑ Random resetting mutation

- Main idea

- ❑ The value of a gene is changed (by probability  $p_m$ ) into another value (from the definition domain)





# Mutation (integer representation)

- ❑ A chromosome  $c = (g_1, g_2, \dots, g_L)$  becomes  $c' = (g'_1, g'_2, \dots, g'_L)$ , where  $g_i, g'_i \in \{val_1, val_2, \dots, val_k\}$ , for  $i = 1, 2, \dots, L$ .

## ❑ Creep mutation

- Main idea
  - ❑ The value of a gene is changed (by probability  $p_m$ ) by adding a positive/negative value
    - New value follows a 0 symmetric distribution
    - The performed change is very small





# Mutation (permutation representation)

- A chromosome  $c = (g_1, g_2, \dots, g_L)$  with  $g_i \neq g_j$  for all  $i \neq j$  becomes  
 $c' = (g'_1, g'_2, \dots, g'_L)$ , where  $g_i, g'_i \in \{val_1, val_2, \dots, val_L\}$ ,  
for  $i = 1, 2, \dots, L$  s. a.  $g'_i \neq g'_j$  for all  $i \neq j$ .

## □ Swap mutation

### ■ Main idea

- Randomly choose 2 genes and swap their values





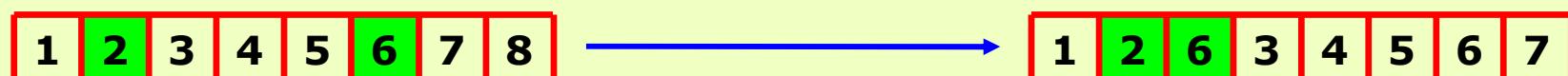
# Mutation (permutation representation)

- A chromosome  $c = (g_1, g_2, \dots, g_L)$  with  $g_i \neq g_j$  for all  $i \neq j$  becomes  $c' = (g_1', g_2', \dots, g_L')$ , where  $g_i, g_i' \in \{val_1, val_2, \dots, val_L\}$ , for  $i = 1, 2, \dots, L$  s. a.  $g_i' \neq g_j'$  for all  $i \neq j$ .

## □ Insertion mutation

### ■ Main idea

- Randomly choose 2 genes  $g_i$  and  $g_j$  with  $j > i$
- Insert gene  $g_j$  after gene  $g_i$  s.a.  $g_i' = g_i, g_{i+1}' = g_j, g_{k+2}' = g_{k+1}$ , for  $k = i, i+1, i+2, \dots$





# Mutation (permutation representation)

- A chromosome  $c = (g_1, g_2, \dots, g_L)$  with  $g_i \neq g_j$  for all  $i \neq j$  becomes  $c' = (g'_1, g'_2, \dots, g'_L)$ , where  $g_i, g'_i \in \{val_1, val_2, \dots, val_L\}$ , for  $i = 1, 2, \dots, L$  s.a.  $g'_i \neq g'_j$  for all  $i \neq j$ .
- Inversion mutation
  - Main idea
    - Randomly choose 2 genes and inverse the order of genes between them (sub-string of genes)





# Mutation (permutation representation)

- A chromosome  $c = (g_1, g_2, \dots, g_L)$  with  $g_i \neq g_j$  for all  $i \neq j$  becomes  $c' = (g_1', g_2', \dots, g_L')$ , where  $g_i, g_i' \in \{val_1, val_2, \dots, val_L\}$ , for  $i = 1, 2, \dots, L$  s.a.  $g_i' \neq g_j'$  for all  $i \neq j$ .

- *scramble mutation*

- Main idea

- Randomly choose a (continuous or discontinuous) sub-array of genes and re-organise that genes





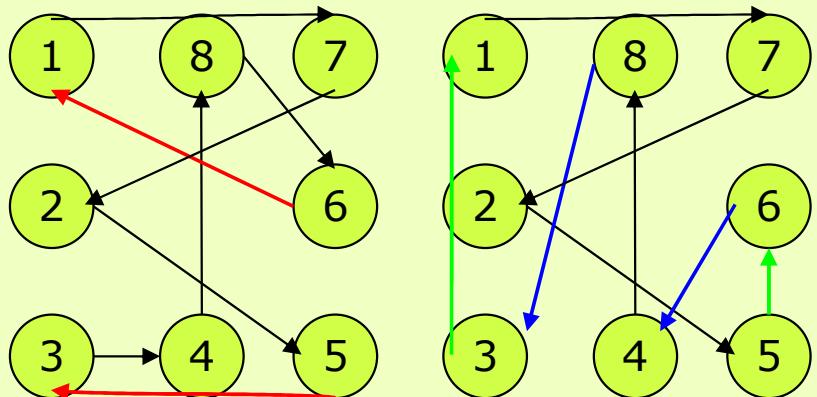
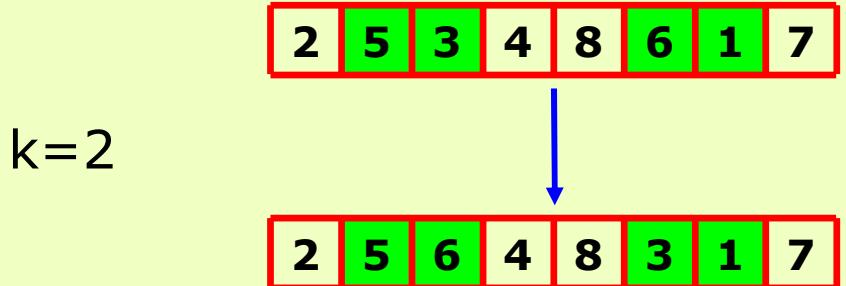
# Mutation (permutation representation)

- A chromosome  $c = (g_1, g_2, \dots, g_L)$  with  $g_i \neq g_j$  for all  $i \neq j$  becomes  $c' = (g'_1, g'_2, \dots, g'_L)$ , where  $g_i, g'_i \in \{val_1, val_2, \dots, val_L\}$ ,  
for  $i = 1, 2, \dots, L$  s.a.  $g'_i \neq g'_j$  for all  $i \neq j$ .

## □ K-opt mutation

### ■ Main idea

- Choose 2 disjoint sub-strings of length k
- Interchange 2 elements of these sub-strings





# Mutation (real representation)

- A chromosome  $c = (g_1, g_2, \dots, g_L)$  becomes  $c' = (g'_1, g'_2, \dots, g'_L)$ , where  $g_i, g'_i \in [a_i, b_i]$ , for  $i=1, 2, \dots, L$ .

## □ Uniform mutation

- Main idea
  - $g'_i$  is changed by probability  $p_m$  into a new value that is randomly uniform generated in  $[a_i, b_i]$  range



# Mutation (real representation)

- ❑ A chromosome  $c = (g_1, g_2, \dots, g_L)$  becomes  $c' = (g'_1, g'_2, \dots, g'_L)$ , where  $g_i, g'_i \in [a_i, b_i]$ , for  $i = 1, 2, \dots, L$ .

## ❑ Non-uniform mutation

### ■ Main idea

- ❑ The value of a gene is probabilistically ( $p_m$ ) changed by adding a positive/negative value
  - The added value belongs to a distribution of type
    - $N(\mu, \sigma)$  (Gaussian) with  $\mu = 0$
    - Cauchy ( $x_0, \gamma$ )
    - Laplace ( $\mu, b$ )
  - And it is re-introduced in  $[a_i, b_i]$  range (if it is necessary) – *clamping*

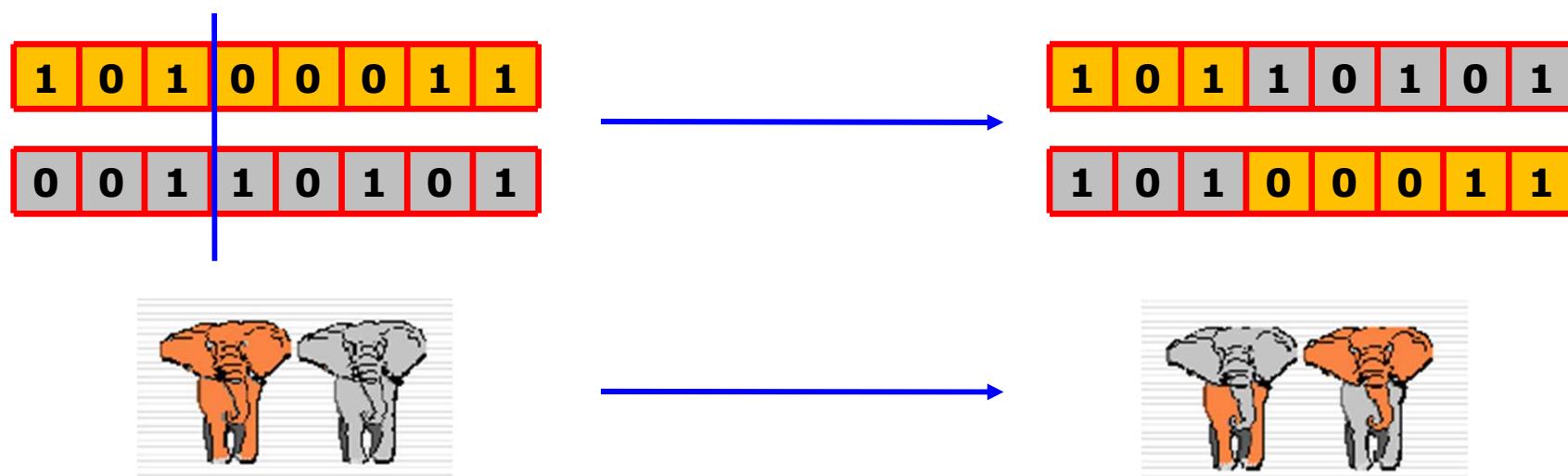
# Recombination



- Aim
  - Mix the parents' information

## Properties

- The offspring has to inherit something from both parents
  - Selection of mixed information is randomly performed
- Operator for exploitation of already discovered possible solutions
- The offspring can be better, the same or weaker than their parents
- Its effects are reducing while the search converges



# Recombination



- Types
  - Binary and integer representation
    - With cutting points
    - Uniforme
  - Permutation representation
    - Order crossover (version 1 and version 2)
    - Partially Mapped Crossover
    - Cycle crossover
    - Edge-based crossover
  - Real representation
    - Discrete
    - Arithmetic
      - Singular
      - Simple
      - Complete
    - Geometric
    - Shuffle crossover
    - Simulated binary crossover
  - Tree-based representation
    - Sub-tree based crossover → future lecture

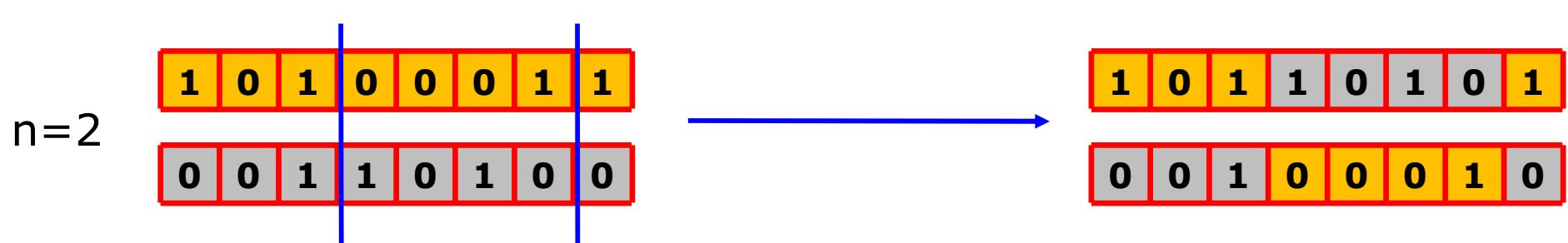


# Recombination (binary and integer representation)

- From 2 parent chromosomes
  - $p_1 = (g_1^1, g_2^1, \dots, g_L^1)$  and  $p_2 = (g_1^2, g_2^2, \dots, g_L^2)$
- 2 offspring are obtained
  - $c_1 = (g_1', g_2', \dots, g_L')$  and  $c_2 = (g_1'', g_2'', \dots, g_L'')$ ,
  - where  $g_i^1, g_i^2, g_i', g_i'' \in \{0, 1\} / \{val_1, val_2, \dots, val_k\}$ , for  $i = 1, 2, \dots, L$

## □ N-cutting point crossover

- Main idea
  - Choose  $n$  cutting-points ( $n < L$ )
  - Cut the parents through these points
  - Put together the resulted parts, by alternating the parents





# Recombination (binary and integer representation)

## □ N cutting point crossover

### ■ Properties

- Average of values encoded by parents = average of values encoded by offspring
  - Eg binary representation on 4 bits of integer numbers – XO with  $n = 1$  after second bit
    - $p_1 = (1, 0, 1, 0), p_2 = (1, 1, 0, 1)$
    - $c_1 = (1, 0, 0, 1), c_2 = (1, 1, 1, 0)$
    - $val(p_1) = 10, val(p_2) = 13 \rightarrow (val(p_1) + val(p_2)) / 2 = 23 / 2 = 11.5$
    - $val(c_1) = 9, val(c_2) = 14 \rightarrow (val(c_1) + val(c_2)) / 2 = 23 / 2 = 11.5$
  - Eg. Binary representation on 4 bits for knapsack problem ( $K=10$ , 4 items of weight and value: (2,7), (1,8), (3,1), (2,3))
    - $p_1 = (1, 0, 1, 0), p_2 = (1, 1, 0, 1)$
    - $c_1 = (1, 0, 0, 1), c_2 = (1, 1, 1, 0)$
    - $val(p_1) = 8, val(p_2) = 18 \rightarrow (val(p_1) + val(p_2)) / 2 = 26 / 2 = 13$
    - $val(c_1) = 10, val(c_2) = 16 \rightarrow (val(c_1) + val(c_2)) / 2 = 26 / 2 = 13$

- Probability of  $\beta \approx 1$  is the largest one

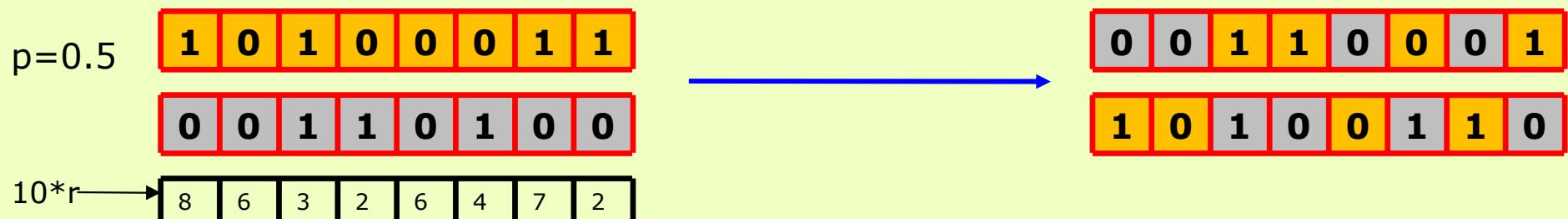
$$\beta = \left| \frac{val(d_1) - val(d_2)}{val(p_1) - val(p_2)} \right|$$

- Contracting crossover  $\beta < 1$ 
  - Offspring values are between parent values
- Expanding crossover  $\beta > 1$ 
  - Parent values are between offspring values
- Stationary crossover  $\beta = 1$ 
  - Offspring values are equal to parent values



# Recombination (binary and integer representation)

- From 2 parent chromosomes
  - $p_1 = (g_1^1, g_2^1, \dots, g_L^1)$  and  $p_2 = (g_1^2, g_2^2, \dots, g_L^2)$
- 2 offspring are obtained
  - $c_1 = (g_1', g_2', \dots, g_L')$  and  $c_2 = (g_1'', g_2'', \dots, g_L'')$ ,
  - where  $g_i^1, g_i^2, g_i', g_i'' \in \{0, 1\} / \{val_1, val_2, \dots, val_k\}$ , for  $i = 1, 2, \dots, L$
- Uniform crossover
  - Main idea
    - Each gene of an offspring comes from a randomly and uniform selected parent:
      - For each gene a uniform random number  $r$  is generated
      - If  $r < probability p$  (usually,  $p=0.5$ ),  $c_1$  will inherit that gene from  $p_1$  and  $c_2$  from  $p_2$ ,
      - otherwise,  $c_1$  will inherit  $p_2$  and  $c_2$  will inherit  $p_1$

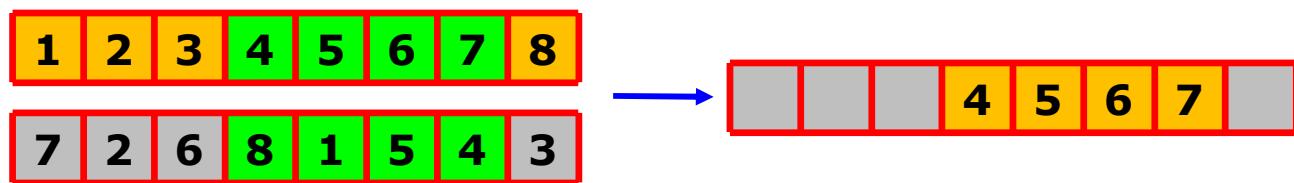
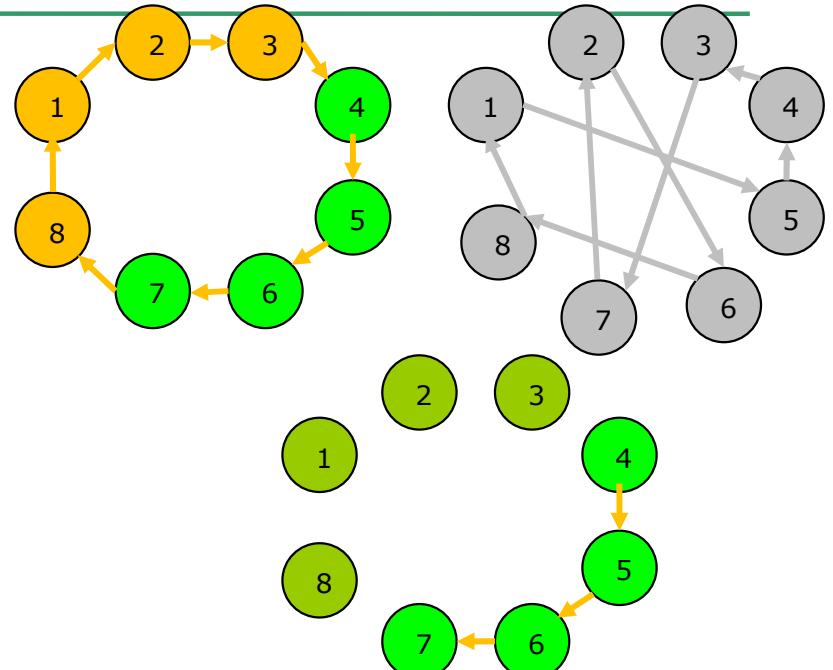




# Recombination (permutation representation)

- From 2 parent chromosomes
  - $p_1 = (g_1^1, g_2^1, \dots, g_L^1)$  and  $p_2 = (g_1^2, g_2^2, \dots, g_L^2)$
- 2 offspring are obtained
  - $c_1 = (g_1', g_2', \dots, g_L')$  and  $c_2 = (g_1'', g_2'', \dots, g_L'')$ ,
  - Where  $g_i^1, g_i^2, g_i', g_i'' \in [1, L] \cap \mathbb{Z}$ , for  $i = 1, 2, \dots, L$ .

- Order crossover
  - Main idea
    - Offspring keep the order of genes from parents
    - Choose a substring of genes from the parent  $p_1$
    - Copy the substring from  $p_1$  into offspring  $d_1$  (on corresponding positions)

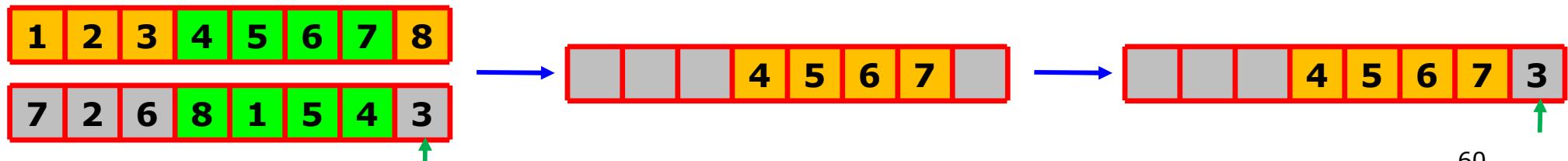
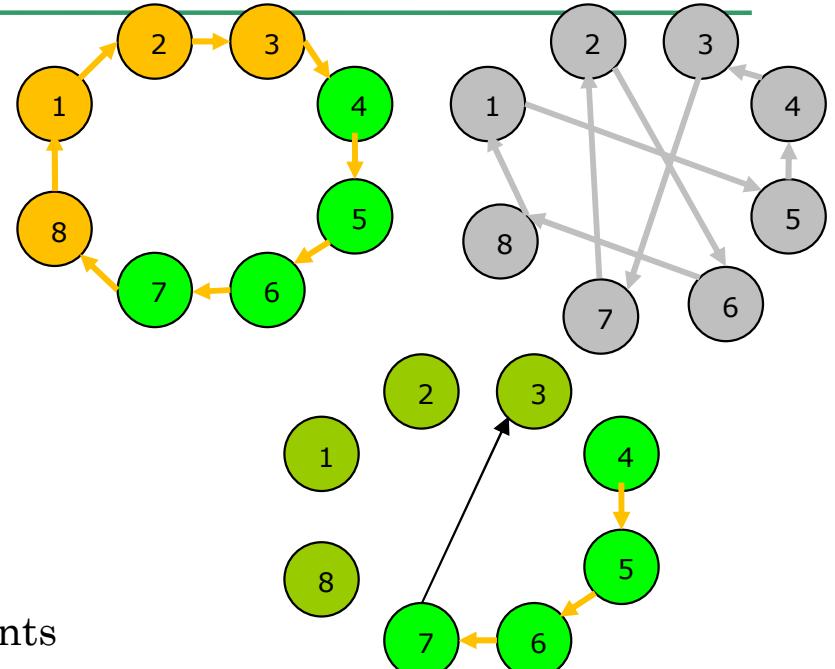




# Recombination (permutation representation)

- From 2 parent chromosomes
  - $p_1 = (g_1^1, g_2^1, \dots, g_L^1)$  and  $p_2 = (g_1^2, g_2^2, \dots, g_L^2)$
- 2 offspring are obtained
  - $c_1 = (g_1', g_2', \dots, g_L')$  and  $c_2 = (g_1'', g_2'', \dots, g_L'')$ ,
  - Where  $g_i^1, g_i^2, g_i', g_i'' \in [1, L] \cap \mathbb{Z}$ , for  $i = 1, 2, \dots, L$ .

- Order crossover
  - Main idea
    - Offspring keep the order of genes from parents
    - Choose a substring of genes from the parent  $p_1$
    - Copy the substring from  $p_1$  into offspring  $d_1$  (on corresponding positions)
    - Copy the genes of  $p_2$  in offspring  $d_1$ :
      - Starting with the first position after sub-string
      - Respecting gene's order from  $p_2$  and
      - Re-loading the genes from start (if the end of chromosome is reached)

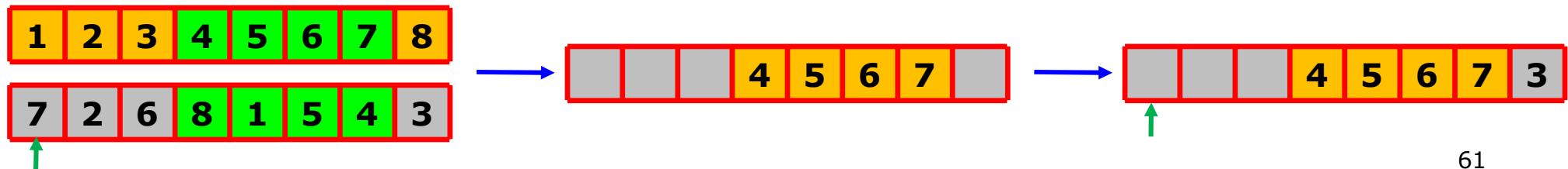
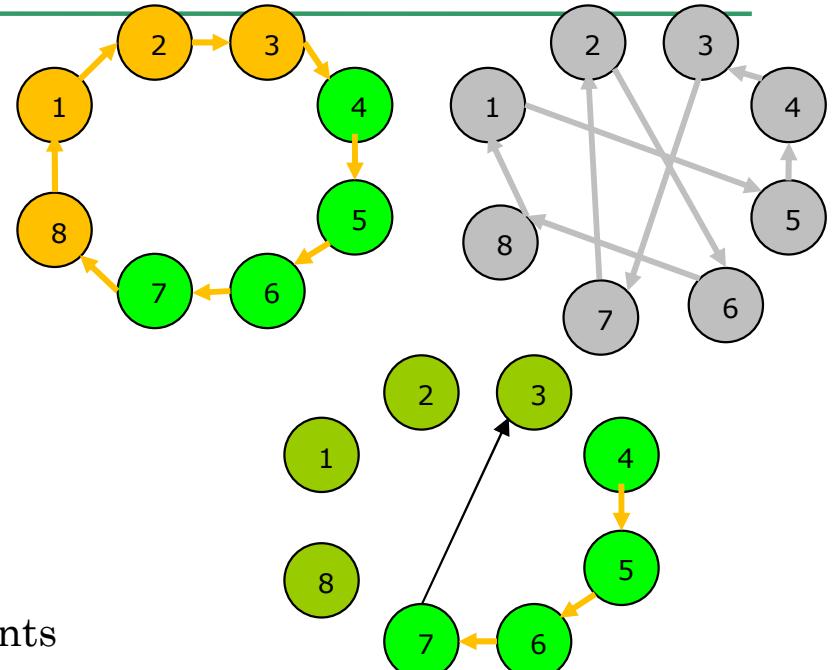




# Recombination (permutation representation)

- From 2 parent chromosomes
  - $p_1 = (g_1^1, g_2^1, \dots, g_L^1)$  and  $p_2 = (g_1^2, g_2^2, \dots, g_L^2)$
- 2 offspring are obtained
  - $c_1 = (g_1', g_2', \dots, g_L')$  and  $c_2 = (g_1'', g_2'', \dots, g_L'')$ ,
  - Where  $g_i^1, g_i^2, g_i', g_i'' \in [1, L] \cap \mathbb{Z}$ , for  $i = 1, 2, \dots, L$ .

- Order crossover
  - Main idea
    - Offspring keep the order of genes from parents
    - Choose a substring of genes from the parent  $p_1$
    - Copy the substring from  $p_1$  into offspring  $d_1$  (on corresponding positions)
    - Copy the genes of  $p_2$  in offspring  $d_1$ :
      - Starting with the first position after sub-string
      - Respecting gene's order from  $p_2$  and
      - Re-loading the genes from start (if the end of chromosome is reached)

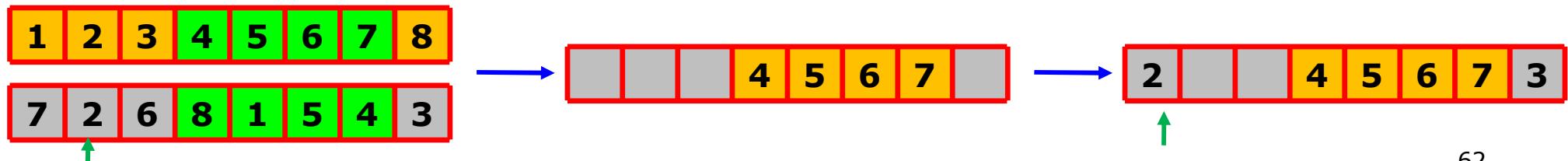
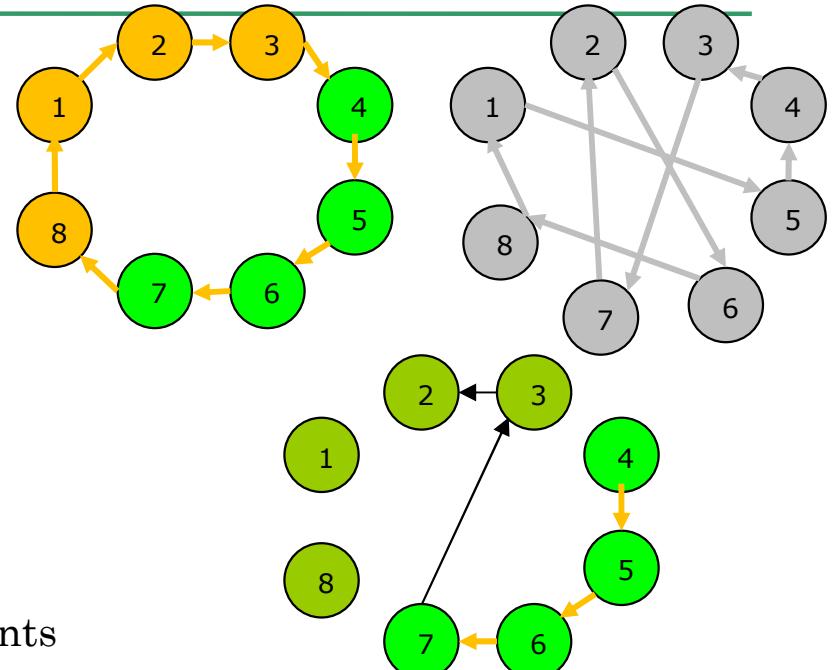




# Recombination (permutation representation)

- From 2 parent chromosomes
  - $p_1 = (g_1^1, g_2^1, \dots, g_L^1)$  and  $p_2 = (g_1^2, g_2^2, \dots, g_L^2)$
- 2 offspring are obtained
  - $c_1 = (g_1', g_2', \dots, g_L')$  and  $c_2 = (g_1'', g_2'', \dots, g_L'')$ ,
  - Where  $g_i^1, g_i^2, g_i', g_i'' \in [1, L] \cap \mathbb{Z}$ , for  $i = 1, 2, \dots, L$ .

- Order crossover
  - Main idea
    - Offspring keep the order of genes from parents
    - Choose a substring of genes from the parent  $p_1$
    - Copy the substring from  $p_1$  into offspring  $d_1$  (on corresponding positions)
    - Copy the genes of  $p_2$  in offspring  $d_1$ :
      - Starting with the first position after sub-string
      - Respecting gene's order from  $p_2$  and
      - Re-loading the genes from start (if the end of chromosome is reached)

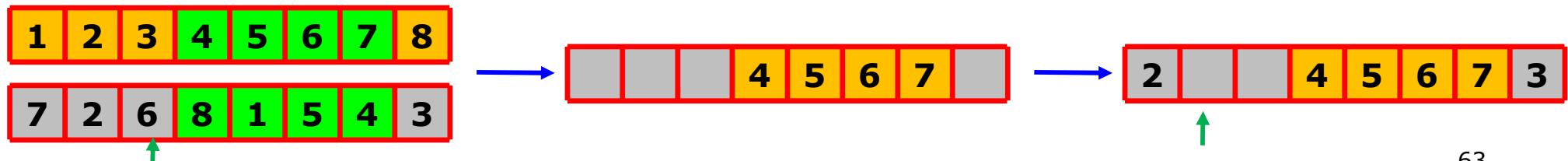
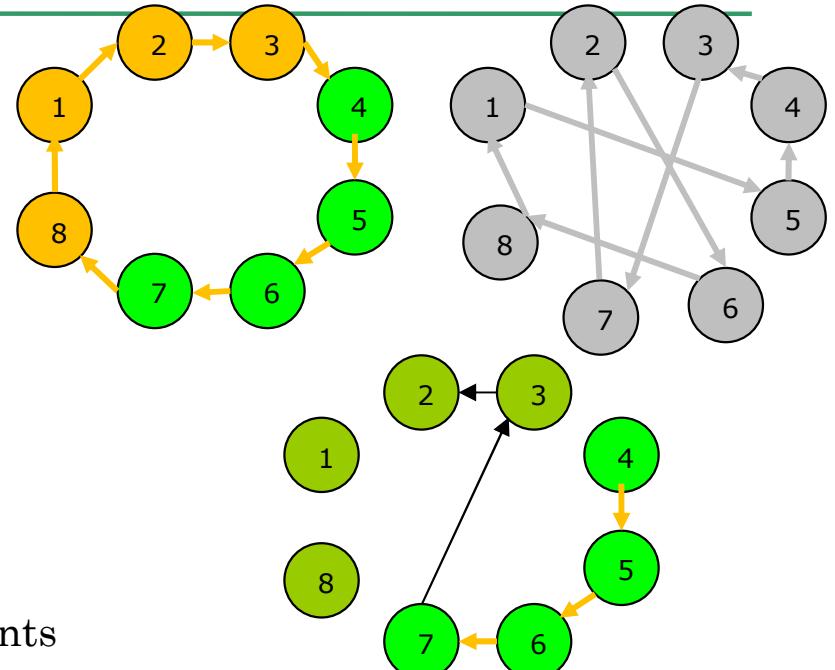




# Recombination (permutation representation)

- From 2 parent chromosomes
  - $p_1 = (g_1^1, g_2^1, \dots, g_L^1)$  and  $p_2 = (g_1^2, g_2^2, \dots, g_L^2)$
- 2 offspring are obtained
  - $c_1 = (g_1', g_2', \dots, g_L')$  and  $c_2 = (g_1'', g_2'', \dots, g_L'')$ ,
  - Where  $g_i^1, g_i^2, g_i', g_i'' \in [1, L] \cap \mathbb{Z}$ , for  $i = 1, 2, \dots, L$ .

- Order crossover
  - Main idea
    - Offspring keep the order of genes from parents
    - Choose a substring of genes from the parent  $p_1$
    - Copy the substring from  $p_1$  into offspring  $d_1$  (on corresponding positions)
    - Copy the genes of  $p_2$  in offspring  $d_1$ :
      - Starting with the first position after sub-string
      - Respecting gene's order from  $p_2$  and
      - Re-loading the genes from start (if the end of chromosome is reached)

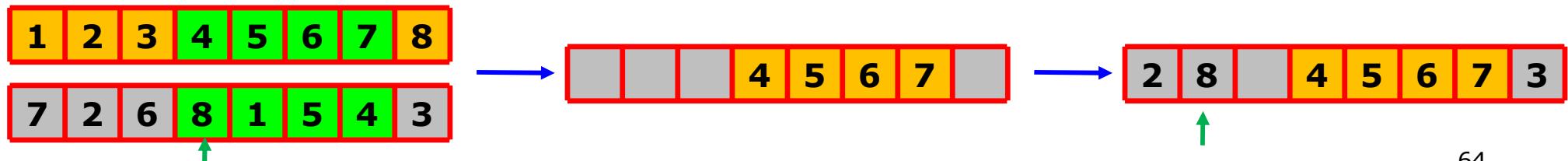
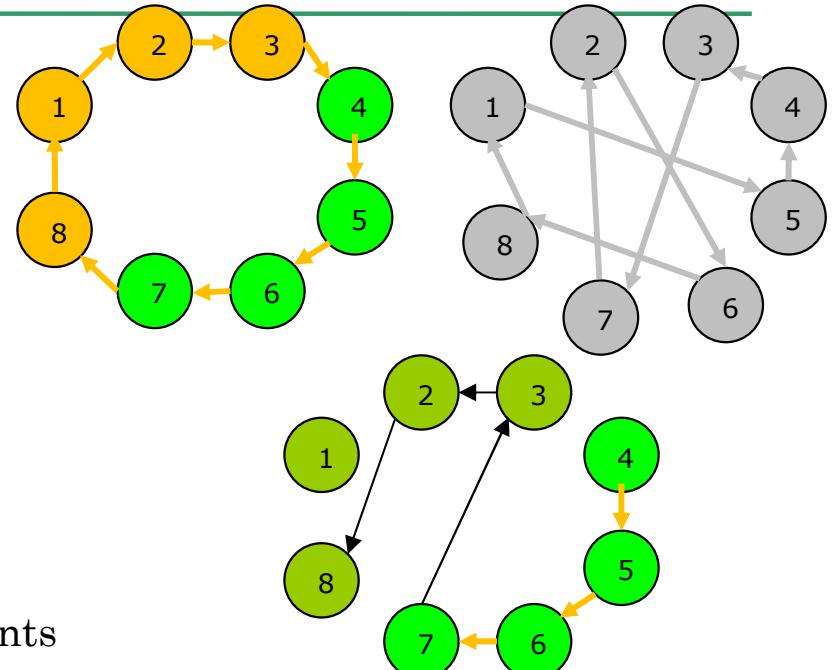




# Recombination (permutation representation)

- From 2 parent chromosomes
  - $p_1 = (g_1^1, g_2^1, \dots, g_L^1)$  and  $p_2 = (g_1^2, g_2^2, \dots, g_L^2)$
- 2 offspring are obtained
  - $c_1 = (g_1', g_2', \dots, g_L')$  and  $c_2 = (g_1'', g_2'', \dots, g_L'')$ ,
  - Where  $g_i^1, g_i^2, g_i', g_i'' \in [1, L] \cap \mathbb{Z}$ , for  $i = 1, 2, \dots, L$ .

- Order crossover
  - Main idea
    - Offspring keep the order of genes from parents
    - Choose a substring of genes from the parent  $p_1$
    - Copy the substring from  $p_1$  into offspring  $d_1$  (on corresponding positions)
    - Copy the genes of  $p_2$  in offspring  $d_1$ :
      - Starting with the first position after sub-string
      - Respecting gene's order from  $p_2$  and
      - Re-loading the genes from start (if the end of chromosome is reached)

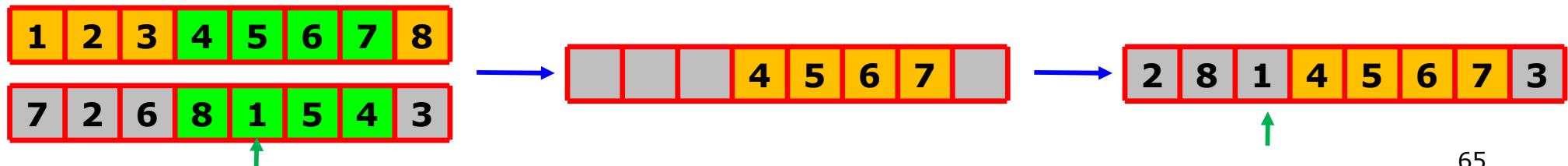
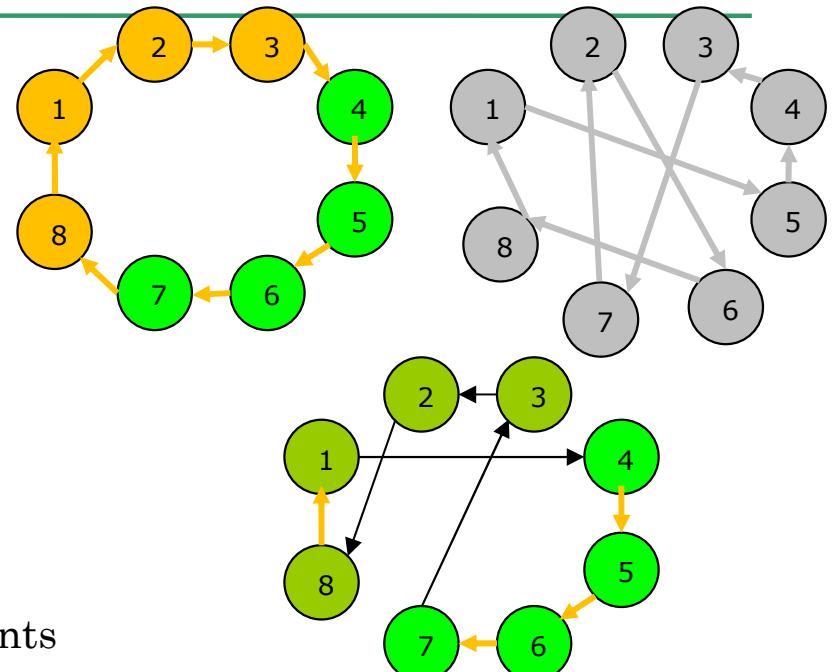




# Recombination (permutation representation)

- From 2 parent chromosomes
  - $p_1 = (g_1^1, g_2^1, \dots, g_L^1)$  and  $p_2 = (g_1^2, g_2^2, \dots, g_L^2)$
- 2 offspring are obtained
  - $c_1 = (g_1', g_2', \dots, g_L')$  and  $c_2 = (g_1'', g_2'', \dots, g_L'')$ ,
  - Where  $g_i^1, g_i^2, g_i', g_i'' \in [1, L] \cap \mathbb{Z}$ , for  $i = 1, 2, \dots, L$ .

- Order crossover
  - Main idea
    - Offspring keep the order of genes from parents
    - Choose a substring of genes from the parent  $p_1$
    - Copy the substring from  $p_1$  into offspring  $d_1$  (on corresponding positions)
    - Copy the genes of  $p_2$  in offspring  $d_1$ :
      - Starting with the first position after sub-string
      - Respecting gene's order from  $p_2$  and
      - Re-loading the genes from start (if the end of chromosome is reached)

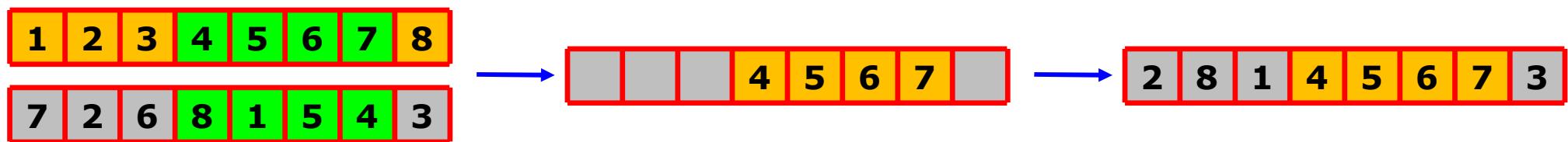
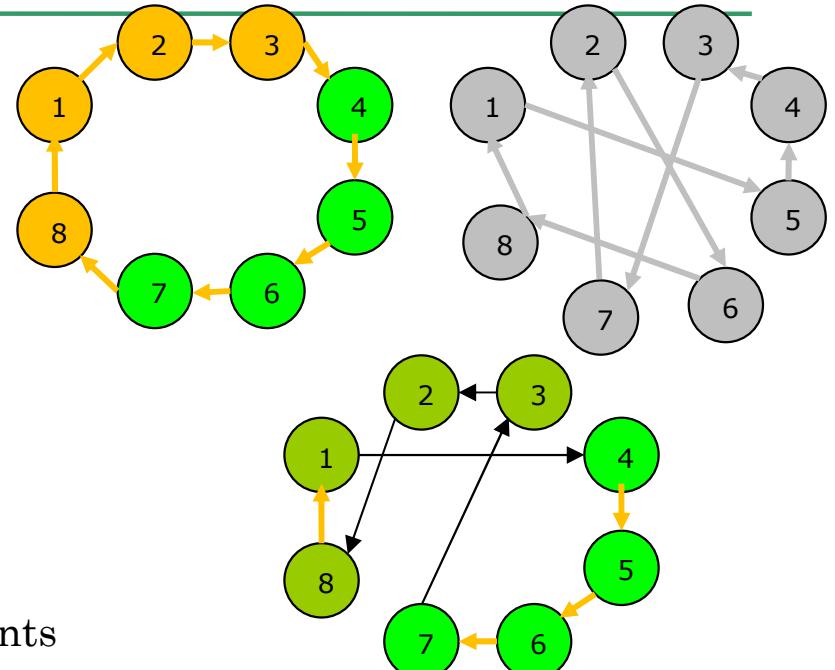




# Recombination (permutation representation)

- From 2 parent chromosomes
  - $p_1 = (g_1^1, g_2^1, \dots, g_L^1)$  and  $p_2 = (g_1^2, g_2^2, \dots, g_L^2)$
- 2 offspring are obtained
  - $c_1 = (g_1', g_2', \dots, g_L')$  and  $c_2 = (g_1'', g_2'', \dots, g_L'')$ ,
  - Where  $g_i^1, g_i^2, g_i', g_i'' \in [1, L] \cap \mathbb{Z}$ , for  $i = 1, 2, \dots, L$ .

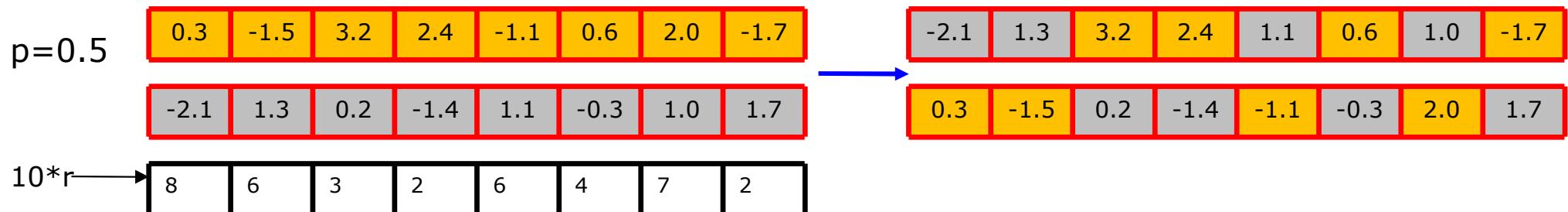
- Order crossover
  - Main idea
    - Offspring keep the order of genes from parents
    - Choose a substring of genes from the parent  $p_1$
    - Copy the substring from  $p_1$  into offspring  $d_1$  (on corresponding positions)
    - Copy the genes of  $p_2$  in offspring  $d_1$ :
      - Starting with the first position after sub-string
      - Respecting gene's order from  $p_2$  and
      - Re-loading the genes from start (if the end of chromosome is reached)





# Recombination (real representation)

- From 2 parent chromosomes
  - $p_1 = (g_1^1, g_2^1, \dots, g_L^1)$  and  $p_2 = (g_1^2, g_2^2, \dots, g_L^2)$
- 2 offspring are obtained
  - $c_1 = (g_1', g_2', \dots, g_L')$  and  $c_2 = (g_1'', g_2'', \dots, g_L'')$ ,
  - Where  $g_i^1, g_i^2, g_i', g_i'' \in [a_i, b_i]$ , for  $i = 1, 2, \dots, L$
- Discrete crossover
  - Main idea
    - Each gene offspring is taken (by the same probability,  $p = 0.5$ ) from one of the parents
    - Similarly to uniform crossover for binary/integer representation
    - The absolute values of genes are not changed (no new information is created)





# Recombination (real representation)

- From 2 parent chromosomes
  - $p_1 = (g_1^1, g_2^1, \dots, g_L^1)$  and  $p_2 = (g_1^2, g_2^2, \dots, g_L^2)$
- 2 offspring are obtained
  - $c_1 = (g_1', g_2', \dots, g_L')$  and  $c_2 = (g_1'', g_2'', \dots, g_L'')$ ,
  - Where  $g_i^1, g_i^2, g_i', g_i'' \in [a_i, b_i]$ , for  $i = 1, 2, \dots, L$
- Arithmetic crossover
  - Main idea
    - Create offspring between parents → arithmetic crossover
      - $z_i = a x_i + (1 - a) y_i$  where  $0 \leq a \leq 1$ .
    - Parameter  $a$  can be:
      - Constant → uniform arithmetic crossover
      - Variable → eg. Depends on the age of population
      - Random → generated for each new XO that is performed
    - New values of a gene can appear
  - Types:
    - Singular arithmetic crossover
    - Simple arithmetic crossover
    - Complete arithmetic crossover



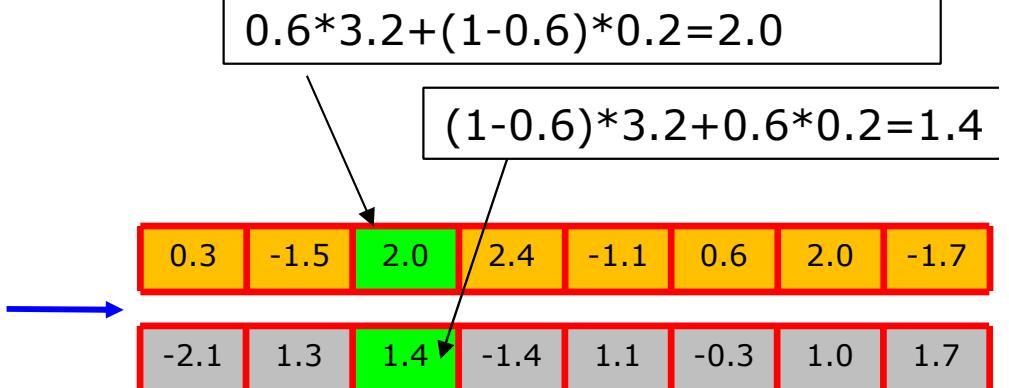
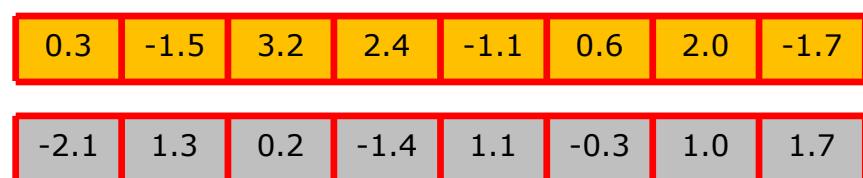
# Recombination (real representation)

- From 2 parent chromosomes
  - $p_1 = (g_1^1, g_2^1, \dots, g_L^1)$  and  $p_2 = (g_1^2, g_2^2, \dots, g_L^2)$
- 2 offspring are obtained
  - $c_1 = (g_1', g_2', \dots, g_L')$  and  $c_2 = (g_1'', g_2'', \dots, g_L'')$ ,
  - Where  $g_i^1, g_i^2, g_i', g_i'' \in [a_i, b_i]$ , for  $i = 1, 2, \dots, L$
  
- Singular arithmetic crossover
  - Choose one gene from two parents (of the same position  $k$ ) and combine them
    - $g_k' = a g_k^1 + (1 - a) g_k^2$
    - $g_k'' = (1 - a) g_k^1 + a g_k^2$
  - The rest of genes are unchanged
    - $g_i' = g_i^1$
    - $g_i'' = g_i^2$ , for  $i = 1, 2, \dots, L$  and  $i \neq k$

$$[a, b] = [-2.5, +3]$$

$$k = 3$$

$$\alpha = 0.6$$





# Recombination (real representation)

- From 2 parent chromosomes
  - $p_1 = (g_1^1, g_2^1, \dots, g_L^1)$  and  $p_2 = (g_1^2, g_2^2, \dots, g_L^2)$
- 2 offspring are obtained
  - $c_1 = (g_1', g_2', \dots, g_L')$  and  $c_2 = (g_1'', g_2'', \dots, g_L'')$ ,
  - Where  $g_i^1, g_i^2, g_i', g_i'' \in [a_i, b_i]$ , for  $i = 1, 2, \dots, L$
  
- Simple arithmetic crossover
  - Select a position  $k$  and combine all the genes after that position
    - $g_i' = a g_i^1 + (1 - a) g_i^2$
    - $g_i'' = (1 - a) g_i^1 + a g_i^2$ , for  $i = k, k + 1, \dots, L$
  - Genes from positions  $< k$  rest unchanged
    - $g_i' = g_i^1$
    - $g_i'' = g_i^2$ , for  $i = 1, 2, \dots, k - 1$

$$0.6 * 0.6 + (1 - 0.6) * (-0.3) = 0.24$$

$$(1 - 0.6) * 0.6 + 0.6 * (-0.3) = 0.06$$

$$[a, b] = [-2.5, +3]$$

$$k = 6$$

$$\alpha = 0.6$$





# Recombination (real representation)

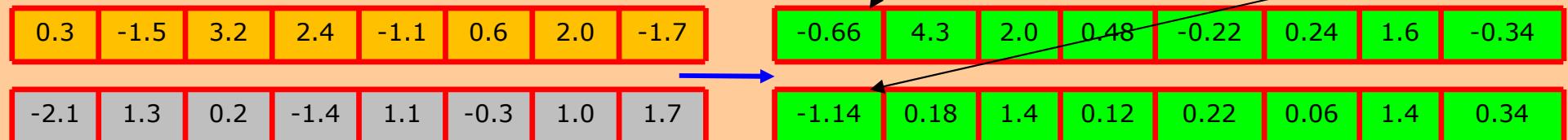
- From 2 parent chromosomes
  - $p_1 = (g_1^1, g_2^1, \dots, g_L^1)$  and  $p_2 = (g_1^2, g_2^2, \dots, g_L^2)$
- 2 offspring are obtained
  - $c_1 = (g_1', g_2', \dots, g_L')$  and  $c_2 = (g_1'', g_2'', \dots, g_L'')$ ,
  - Where  $g_i^1, g_i^2, g_i', g_i'' \in [a_i, b_i]$ , for  $i = 1, 2, \dots, L$
- Complete arithmetic crossover

- All of the genes are combined
  - $g_i' = a g_i^1 + (1 - a) g_i^2$
  - $g_i'' = (1 - a) g_i^1 + a g_i^2$ , for  $i = 1, 2, \dots, L$

$$0.6 * 0.3 + (1 - 0.6) * (-2.1) = -0.66$$

$$[a, b] = [-2.5, +3]$$
$$\alpha = 0.6$$

$$(1 - 0.6) * 0.3 + 0.6 * (-2.1) = -1.14$$

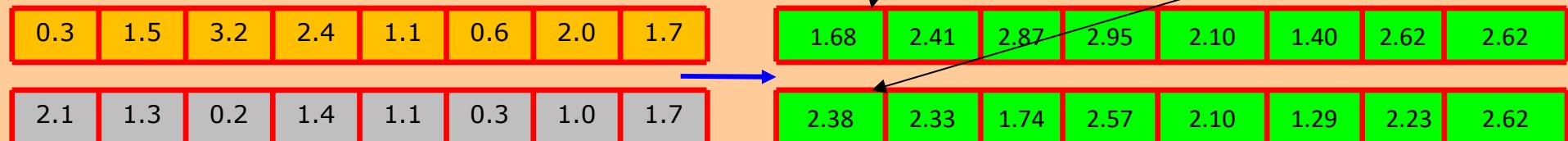




# Recombination (real representation)

- From 2 parent chromosomes
    - $p_1 = (g_1^1, g_2^1, \dots, g_L^1)$  and  $p_2 = (g_1^2, g_2^2, \dots, g_L^2)$
  - 2 offspring are obtained
    - $c_1 = (g_1', g_2', \dots, g_L')$  and  $c_2 = (g_1'', g_2'', \dots, g_L'')$ ,
    - Where  $g_i^1, g_i^2, g_i', g_i'' \in [a_i, b_i]$ , for  $i = 1, 2, \dots, L$
- 
- Geometric crossover
    - Main idea
      - Each gene of an offspring represents the product between parent's genes, each of them by a given exponent  $\omega$  and  $1-\omega$ , respectively (where  $\omega$  is a real positive number  $\leq 1$ )
      - $g_i' = (g_i^1)^\omega (g_i^2)^{1-\omega}$
      - $g_i'' = (g_i^1)^{1-\omega} (g_i^2)^\omega$

$$[a, b] = [-2.5, +3]$$
$$\omega = 0.7$$





# Recombination (real representation)

- From 2 parent chromosomes
  - $p_1 = (g_1^1, g_2^1, \dots, g_L^1)$  and  $p_2 = (g_1^2, g_2^2, \dots, g_L^2)$
- 2 offspring are obtained
  - $c_1 = (g_1', g_2', \dots, g_L')$  and  $c_2 = (g_1'', g_2'', \dots, g_L'')$ ,
  - Where  $g_i^1, g_i^2, g_i', g_i'' \in [a_i, b_i]$ , for  $i = 1, 2, \dots, L$
- Blend crossover – BLX
  - Main idea
    - A single offspring is created
    - Offspring's genes are randomly generated from  $[Min_i - I * a, Max_i + I * a]$  range, where:
      - $Min_i = \min\{g_i^1, g_i^2\}$ ,  $Max_i = \max\{g_i^1, g_i^2\}$
      - $I = Max - Min$ ,  $a$  – parameter from  $[0, 1]$

$$[a, b] = [-2.5, +3]$$

$$a = 0.7$$

0.3	1.5	3.2	2.4	1.1	0.6	2.0	1.7
2.1	1.3	0.2	1.4	1.1	0.3	1.0	1.7



1.25	1.45	-1.11	2.37	1.10	0.11	0.70	1.70
------	------	-------	------	------	------	------	------

Min	0.3	1.3	0.2	1.4	1.1	0.3	1.0	1.7
Max	2.1	1.5	3.2	2.4	1.1	0.6	2.0	1.7
I	0.8	0.2	3.0	1.0	0	0.3	1.0	0.0

Min-Ia	-0.26	1.16	-1.90	0.70	1.10	0.09	0.30	1.70
Max+Ia	2.66	1.50	3.20	2.40	1.10	0.60	2.00	1.70

# Recombination or mutation?

---

## □ Intense debates

### ■ Questions:

- Which is the best operator?
- Which is the most necessary operator?
- Which is the most important operator?

### ■ Answers:

- Depend on problem, but,
- In general, is better to use both operators
- Each of them having another role (purpose).
- EAs with mutation only are possible, but EAs with crossover only are not possible

## □ Search aspects:

- Exploration → discovering promising regions in the search space (accumulating useful information about the problem)
- Exploitation → optimising in a promising region of the search space (by using the existent information)
- Cooperation and competition mut exist between these 2 aspects

## □ Recombination

- Exploitation operator → performs a large jump into a region somewhere between the regions associated to parents
  - Effects of exploitation decrease while AE is converging
- Binary/n-ary operator that can combine information from 2/more parents
- Operator that does not change the frequency of values from chromosome at the population level

## □ Mutation

- Exploration operator → performs small random diversions, remaining in a neighbourhood of parent
  - Local optima escape
- Operator that can introduce new genetic information
- Operator that change the frequency of values from chromosome at the population level



# Stop condition

---

- Choosing a stop condition
  - An optimal solution was found
  - The physical resources were ended
    - A given number of fitness evaluation has been performed
  - The user resources (time, patience) were ended
    - Several generation without improvements have been born



# Evaluation

---

- Performance evaluation of an EA
  - After more runs
    - Statistical measures are computed
      - Average of solutions
      - Median of solutions
      - Best solution
      - Worst solution
      - Standard deviation of solutions – for comparisons
    - The number of independent runs must be large enough

# EAs

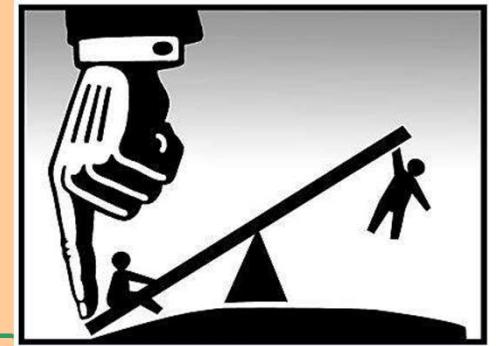
---



## Analyse of complexity

- The most costly part → fitness evaluation

# EAs

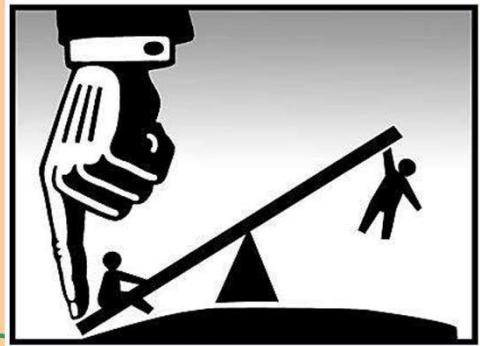


## Advantages

- AEs have a general sketch for all the problems
  - Only
    - representation
    - fitness function
  - are changed
- AEs are able to give better results than classical optimisation methods because
  - They do not require linearization
  - They are not based on some presumptions
  - They do not ignore some possible solutions
- AEs are able to explore more possible solutions than human can

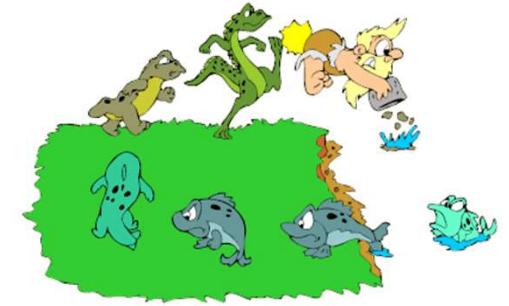
# AES

---



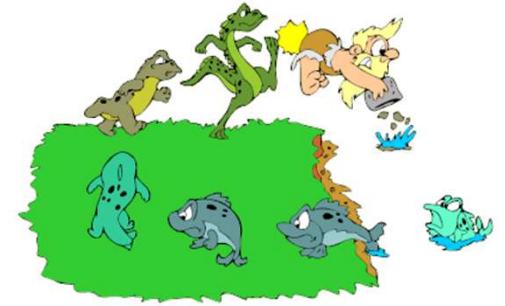
## Disadvantages

- Large running time



## Applications

- Vehicle design
  - Material composition
  - Vehicle shape
- Engineering design
  - Structural and organisational optimisation of constructions (buildings, robots, satellites, turbines)
- Robotics
  - Design and components optimisation
- Hardware evolution
  - Digital circuits optimisation
- Telecommunication optimisation
- Cross-word game generation
- Biometric inventions (inspired by natural architectures)
- Traffic and transportation routing
- PC games
- Cryptography
- Genetics
- Chemical analysis of kinematics
- Financial and marketing strategies



## Types

- Evolutionary strategies
- Evolutionary programming
- Genetic algorithms
- Genetic programming

---

**Thank you for your attention!**

# ARTIFICIAL INTELLIGENCE



**Particle Swarm Optimization  
Ant Colony Optimization**

# Local search

---

- Simple local search – a single neighbour state is retained
  - Hill climbing → selects the best neighbour
  - Simulated annealing → selects probabilistically the best neighbour
  - Tabu search → retains the list of visited solutions
- Beam local search – more states are retained (a population of states)
  - Evolutionary algorithms
  - Particle swarm optimisation
  - Ant colony optimization

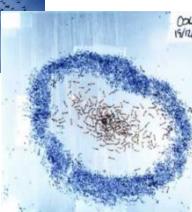
# Nature-inspired algorithms

---

## □ Swarm intelligence (collective intelligence)

- A group of individuals that interact in order to achieve some objectives by collective adaptation to a global or local environment
- A computational metaphor inspired by

- Birds' flying (V shape)
- Ants that are searching food
- Bees' swarms that are constructing their nest
- Schools of fish



- Because:

- Control is distributed among more individuals
- Individuals locally communicate
- System behaviour transcends the individual behaviour
- System is robust and can adapt to environment changes



- Social insects (2% of total)

- Ants
  - 50% of social insects
  - 1 ant has ~ 1mg → total weight of ants ~ total weight of humans
  - Live for over 100 millions of years (humans live for over 50 000 years)
- Termites
- Bees

# Nature-inspired algorithms

---

- Swarm (Group)
  - More individuals, apparently un-organised, that are moving in order to form a group, but each individual seems to move in a particular direction
  - Inside the group can appear some social processes
  - The collection is able to do complex tasks
    - Without a guide or an external control
    - Without a central coordination
  - The collection can have performances better than the independent individuals
- Collective adaptation → self-organisation
  - Set of dynamic mechanisms that generates a global behaviour as a result of interaction among individual components
  - Rules that specify this interaction are executed based on local information only, without global references
  - Global behaviour is an emergent property of the system (and not one external imposed)

# PSO

---

- ❖ Theoretical aspects
- ❖ Algorithm
- ❖ Example
- ❖ Properties
- ❖ Applications

# PSO – theoretical aspects

---

- Proposed by
  - ◆ Kennedy and Eberhart in 1995
  - ◆ Inspired by social behaviour of bird swarms and school of fish
- Search is
  - ◆ **Cooperative**, guided by the relative quality of individuals
- Search operators
  - ◆ A kind of mutation

# PSO – theoretical aspects

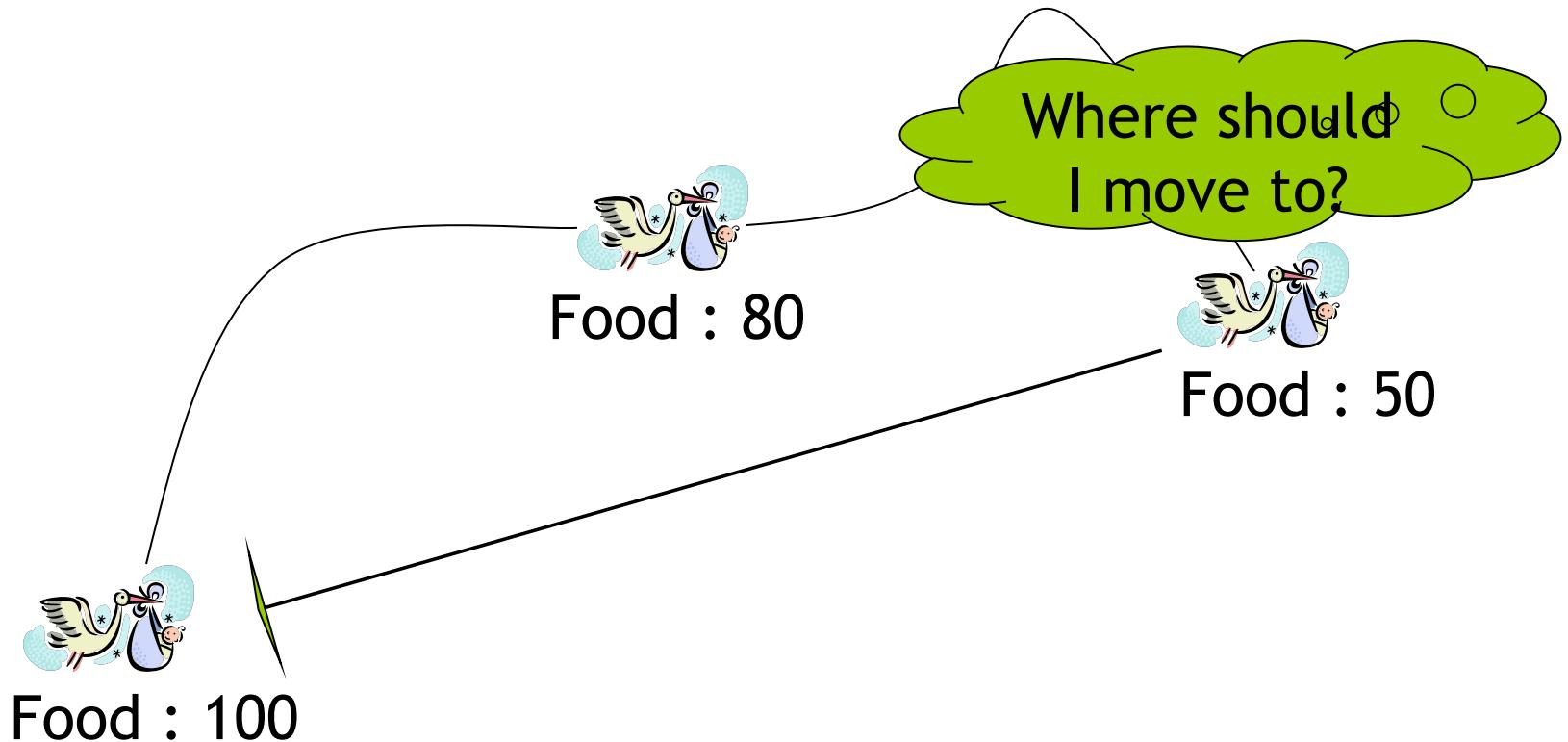
---

## □ Special elements

- Optimisation method based on:
  - Populations ( $\approx$  EAs) of particles ( $\approx$  chromosomes) that search the optimal solution
  - Cooperation (instead of concurrence, like in EAs case)
- Each particle
  - moves (in the search space) and has a velocity (velocity  $\approx$  movement, because the time is discrete)
  - Retains the place where it has obtained the best results
  - Has associated a neighbourhood of particles
- Particles cooperate
  - Exchange information among them (regarding the discovering performed in the places already visited)
  - Each particle knows the fitness of neighbours such as it can use the position of the best neighbour for adjusting its velocity

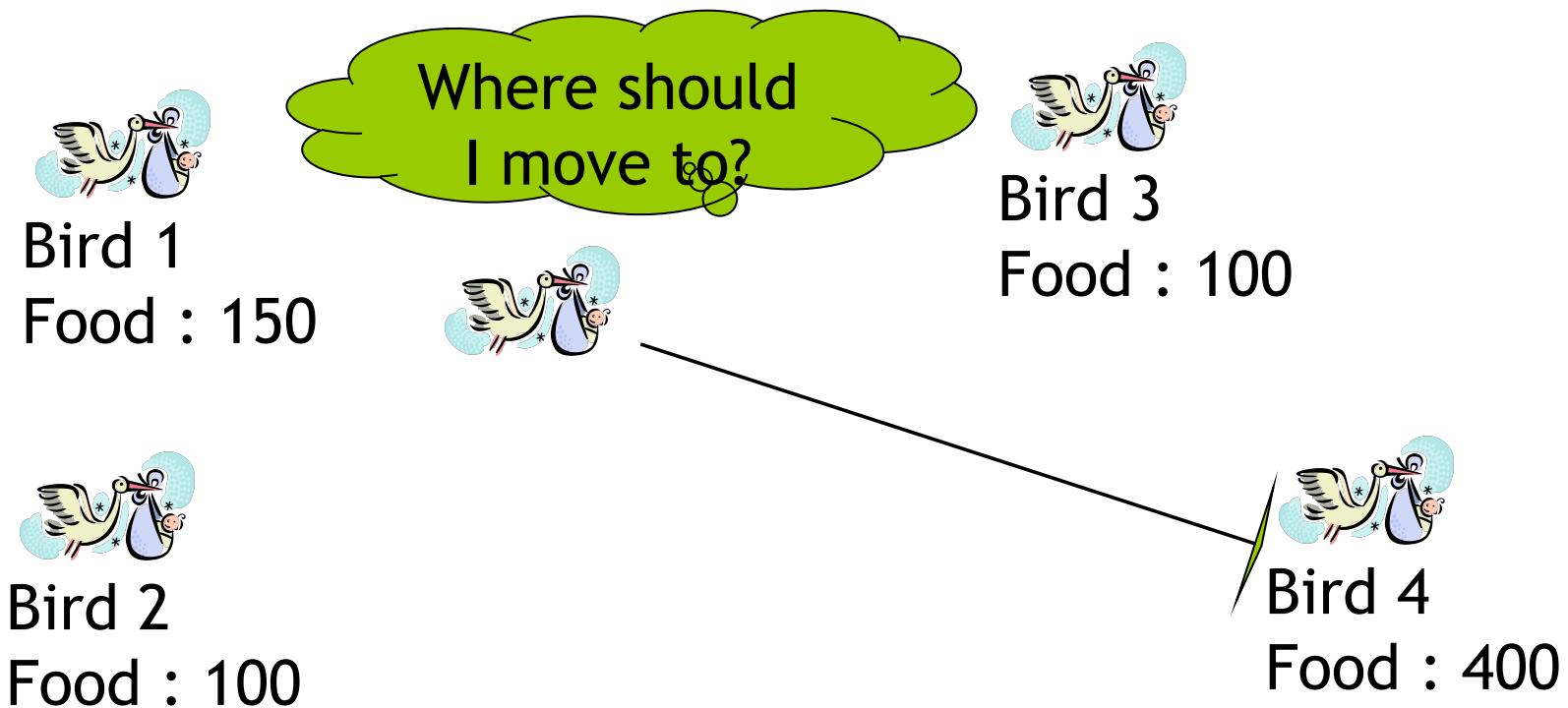
# PSO – theoretical aspects

Main idea: cognitive behaviour → an individual remembers past knowledge (has memory)



# PSO – theoretical aspects

Main idea: social behaviour → an individual relies on the knowledge of other members of the group



# PSO – algorithm

---

## □ General sketch

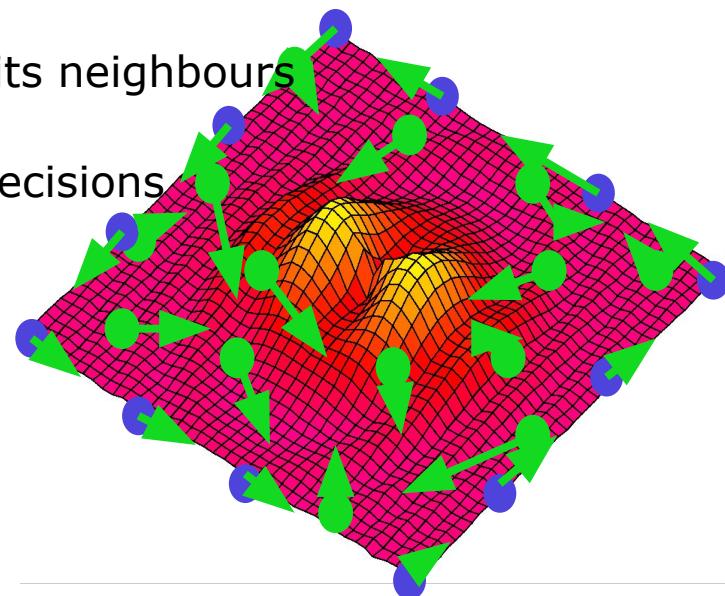
1. Creation of the initial population of particles
  1. Random positions
  2. Null/random velocities
2. Evaluation of particles
3. For each particle
  - Update the memory
    - Identify the best particle of the swarm ( $g_{Best}$ ) / of the current neighbourhood ( $I_{Best}$ )
    - Identify the best position (with the best fitness) reached until now –  $p_{Best}$
  - Update the velocity
  - Update the position
5. If the stop conditions are not satisfied, go back to step 2; otherwise STOP.

# PSO – algorithm

---

## 1. Creation of the initial population of particles

- Each particle has associated
  - A position – possible solution of the problem
  - A velocity – changes a position into another position
  - A quality function (fitness)
- Each particle has to:
  - Interact (exchange information) with its neighbours
  - Memorise a previous position
  - Use the information in order to take decisions
- Initialisation of particles
  - Random positions
  - Null/random velocities



# PSO – algorithm

---

## 2. Evaluation of particles

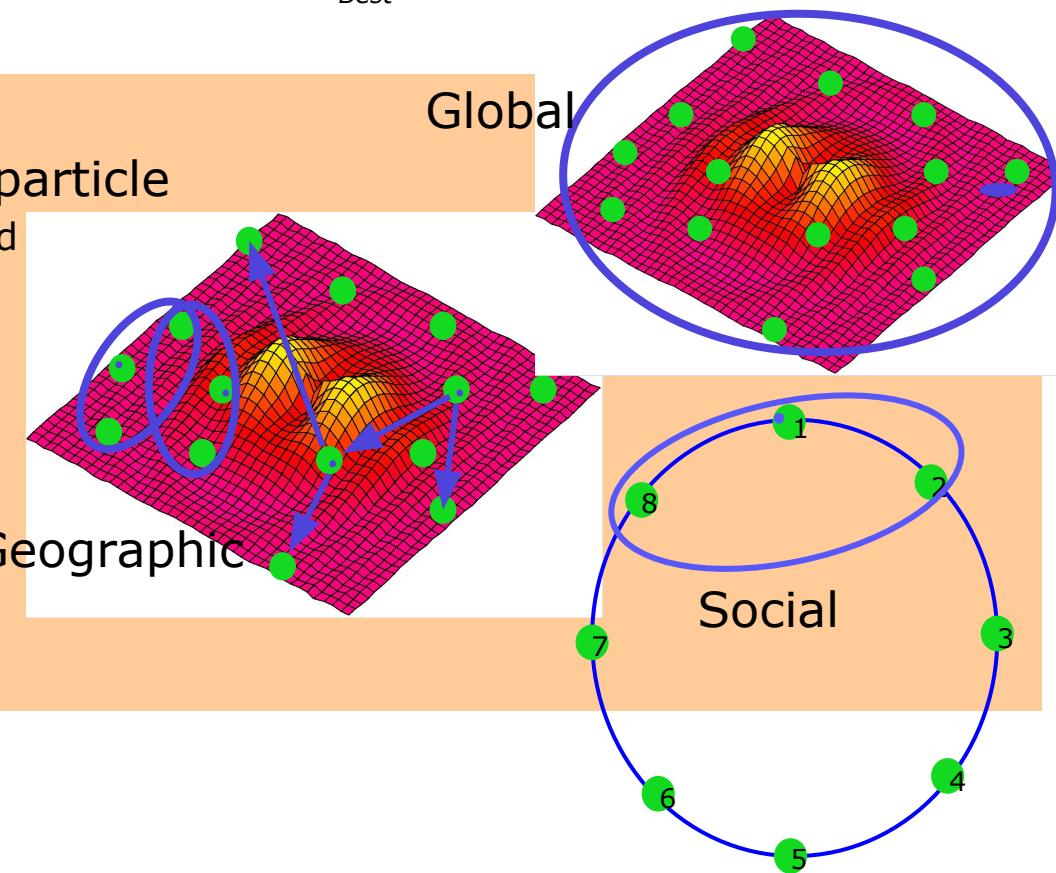
- Depends on problem

# PSO – algorithm

## 3. For each particle $p_i$

- Update the memory
  - Identify the best particle of the swarm ( $g_{Best}$ ) / of the current neighbourhood ( $I_{Best}$ )

- Neighbourhood for a particle
  - Neighbourhood's spread
    - Global
    - Local
  - Neighbourhood's type
    - Geographic
    - Social
    - Circular



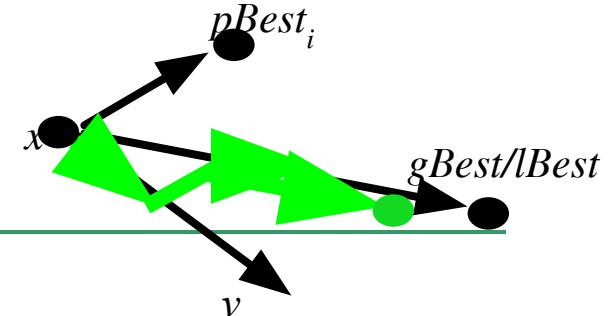
# PSO – algorithm

---

## 3. For each particle $p_i$

- Update the memory
  - Identify the best particle of the swarm ( $g_{Best}$ ) / of the current neighbourhood ( $I_{Best}$ )
  - Identify the best position (with the best fitness) reached until now –  $p_{Best}$

# PSO – algorithm



3. For each particle  $p_i$ ,
  - Update the velocity  $\mathbf{v}_i$  and position  $\mathbf{x}_i = (x_{i1}, x_{i2}, \dots, x_{iD})$  (on each dimension)
    - $v_{id} = w * v_{id} + c_1 * rand() * (p_{Best\_d} - x_{id}) + c_2 * rand() * (g_{Best\_d} - x_{id})$
    - $x_{id} = x_{id} + v_{id}$
    - where:
      - $i=1,N$  ( $N$  – total number of particles/swarm size);
      - $d = 1,D$
      - $w$  – inertia coefficient (Shi, Eberhart)
        - $w * v_{id}$  inertial factor → forces the particle to move in the same direction until now (*audacious*)
        - Balance the search between global exploration (large  $w$ ) and local exploration (small  $w$ )
        - Can be constant or descending (while the swarm is getting old)
      - $c_1$  - cognitive learning coefficient
        - $c_1 * rand() * (p_{Best\_d} - x_{id})$  – cognitive factor → forces the particle to move towards its best position (*conservation*)
      - $c_2$  - social learning coefficient
        - $c_2 * rand() * (g_{Best\_d} - x_{id})$  – social factor → forces the particle to move towards the best neighbour (*follower*)
        - $c_1$  and  $c_2$  can be equal or different ( $c_1 > c_2 \text{ si } c_1 + c_2 < 4$  – Carlisle, 2001)
    - Each component of velocity vector must belong to a given range  $[-v_{max}, v_{max}]$  in order to keep the particles inside the search space

# PSO – properties

---

- PSO principles:
  - Proximity – the group has to perform computing in space and time
  - Quality – the group has to be able of answering to the quality of environment
  - Stability – the group has not to change its behaviour at each environment change
  - Adaptability – the group has to be able of changing its behaviour when the cost on change is not prohibit
- Differences from EAs:
  - There is no recombination operator – information exchange takes place based on particle's experience and based on the best neighbour (not based on the parents selected based on quality only)
  - Position update ~ mutation
  - Selection is not utilized – survival is not based on quality (fitness)
- PSO versions
  - PSO binary and discrete
  - PSO with more social learning coefficients
  - PSO with heterogeneous particles
  - Hierarchical PSO

# PSO – properties

---

- PSO discrete (binary)
  - PSO version for a discrete search space
  - Position of a particle
    - Possible solution of the problem → binary string
    - Changes based on the velocity of particle
  - Velocity of a particle
    - Element from a continuous space
    - Changes based on standard PSO principles
    - Can be viewed as changing probability of the corresponding bit from the particle's position

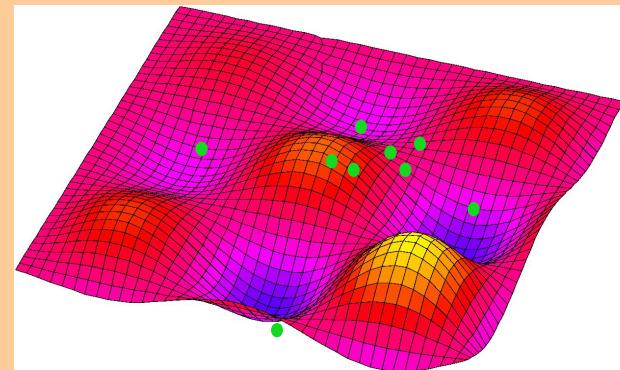
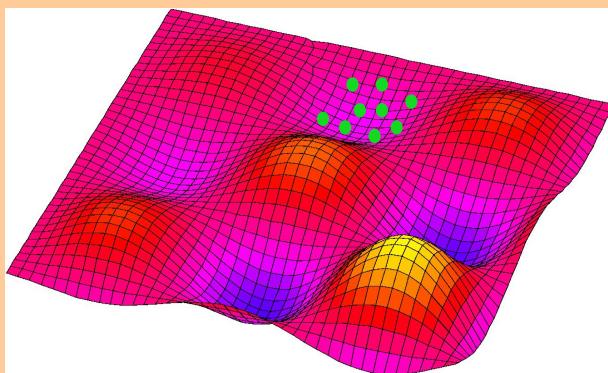
$$x_{ij} = \begin{cases} 1, & \text{if } \tau < s(v_{ij}) \\ 0, & \text{otherwise} \end{cases}, \text{ where } s(v_{ij}) = \frac{1}{1 + e^{-v_{ij}}}$$

# PSO – properties

---

## □ Risks

- Particles has the trend to group in the same place
  - To rapid convergence and the impossibility to escape from local optima
  - Solution:
    - Re-initialisation of some particles



- Particles move through infeasible regions

# PSO – properties

---

## □ Analysis of PSO algorithm

- Dynamic behaviour of the swarm can be analysed by 2 index:

- Dispersion index

- Measures the spreading degree of particle around the best particle of the swarm
    - Average of absolute distances (on each dimension) between each particle and the best particle of the swarm
    - Explains the cover degree (small or spread) of the search space

- Velocity index

- Measures the moving velocity of the swarm into a iteration
    - Average of absolute velocities
    - Explain how the swarm moves (aggressive or slow)

# PSO – applications

---

- Control and design of antenna
- Biological, medical and pharmaceutical applications
  - Analysis of tremor in Parkinson's disease
  - Cancer Classification
  - Prediction of protein structure
- Network communication
- Combinatorial optimisation
- Financial optimisation
- Image and video analyse
- Robotics
- Planning
- Network security, intrusion detection, cryptography
- Signal processing

# ACO

---

- ❖ Theoretical aspects
- ❖ Algorithm
- ❖ Example
- ❖ Properties
- ❖ Applications

# ACO – theoretical aspects

---

- Proposed
  - By Colorni and Dorigo in 1991 for solving discrete optimisation problems – TSP – as a comparison for EAs
  - Inspired by social behaviour of ants that search a path from their nest and food
  - Why ants?
    - Colony system (from several ants to millions of ants)
    - Labor division
    - Social behaviour is very complex
- Search
  - **Cooperative**, guided by the **relative** quality of individuals
- Search operators
  - Constructive ones, adding elements in solution

# ACO – theoretical aspects

---

## ❖ Special elements

- The optimisation problem must be transformed into a problem of identifying the optimal path in an oriented graph
- Ants construct the solution by walking through the graph and put pheromones on some edges
- Optimisation method based on
  - Ant colonies ( $\approx$ EAs) that search the optimal solution
  - Cooperation (instead of concurrence like in EAs)
- Each ant:
  - Moves (in the search space) and put some pheromones on its path
  - Memorises the path
  - Selects the path based on
    - The existing pheromones on that path
    - Heuristic information associated to that path
  - Cooperates with other ants through the pheromone trail (that corresponds to a path) that
    - Depends on the solution quality
    - Evaporates while the time is passing

# ACO – theoretical aspects

---

- Natural ants
  - An ant colony start to search some food

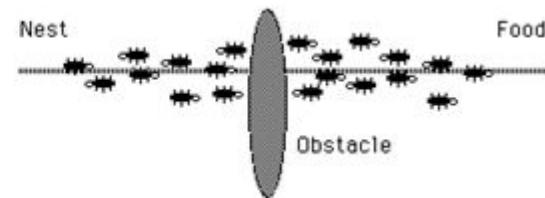


# ACO – theoretical aspects

---

## □ Natural ants

- An ant colony start to search some food
- At a moment, an obstacle appears

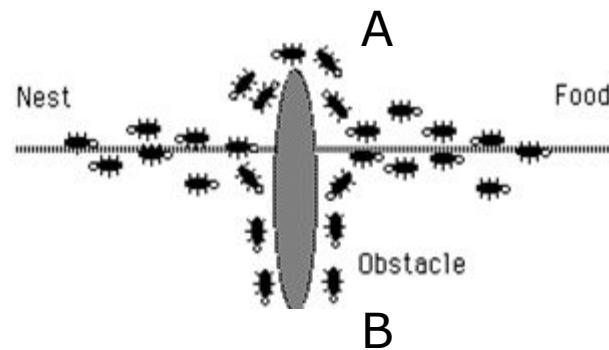


# ACO – theoretical aspects

---

## □ Natural ants

- An ant colony start to search some food
- At a moment, an obstacle appears
- The ants will surround the obstacle either on path A or path B

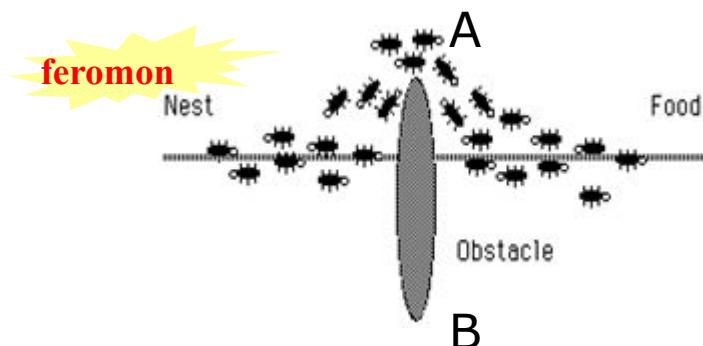


# ACO – theoretical aspects

---

## □ Natural ants

- An ant colony start to search some food
- At a moment, an obstacle appears
- The ants will surround the obstacle either on path A or path B
- Because the path A is shorter, the ants of this path will performed more rounds and, therefore, will put more pheromones
- Pheromone concentration will quickly increase on path A (relative to path B) such as the ants from path B will re-oriented to path A
- Because the ants do not follow path B and because the pheromone trail evaporates, the trail of ants from path B will disappear
- Therefore, the ants will take the shortest path (path A)



# ACO – theoretical aspects

---

- Artificial ants look like natural ants
  - Walk from their nest towards food
  - Discover the shortest path based on pheromone trail
    - Each ant performed random moves
    - Each ant put some pheromone on its path
    - Each ant detects the path of “boss ant” and tends to follow it
    - Increasing the pheromone of a path will determine to increase the probability to follow that path by more ants
- But they have some improvements:
  - Has memory
    - Retains performed moves → has a proper state (retaining the history of decisions)
    - Can come back to their nest (based on pheromone trail)
  - Are not completely blind – can appreciate the quality of their neighbour space
  - Perform move in a discrete space
  - Put pheromone based on the identified solution, also

# ACO – theoretical aspects

---

- Pheromone trail plays the role of
  - A collective, dynamic and distributed memory
  - A repository of the most recent ants' experiences of searching food
  
- Ants can indirectly communicate and can influence each-other
  - By changing the chemical repository
  - In order to identify the shortest path from nest to food

# ACO – algorithm

---

- While iteration < maximum # of iterations
  - 1. Initialisation
  - 2. While # of steps required to identify the optimal solution is not performed
    - For each ant of the colony
      - Increase the partial solution by an element (ant moves one step)
      - Change locally the pheromone trail based on the last element added in solution
  - 3. Change the pheromone trail on the paths traversed by
    - all ants/the best ant
  - 4. Return the solution identified by the best ant

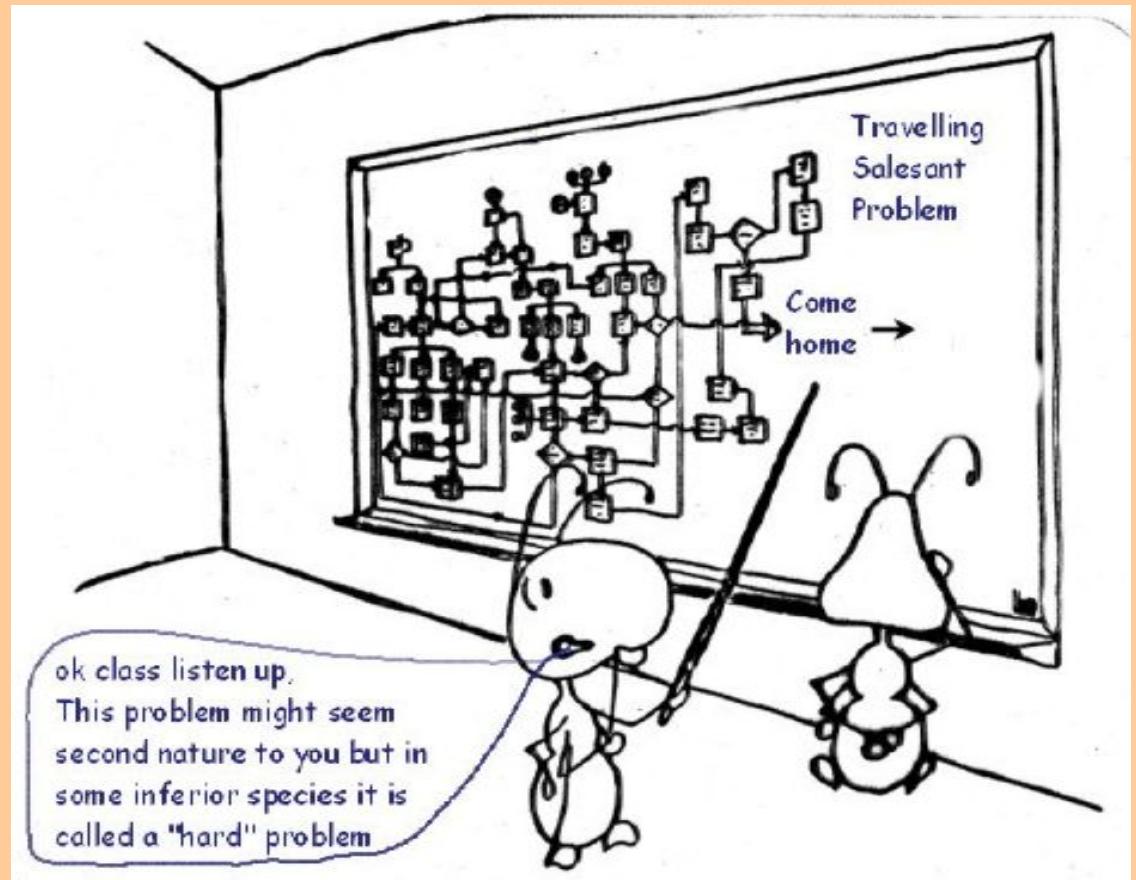
# ACO – algorithm

---

- 3 versions – differences:
  - Rules for transforming a state into another state (moving rules for ants)
  - Moment when the ants deposit pheromones
    - While the solution is constructed
    - At the end of solution's construction
  - Which ant deposits pheromones
    - All the ants
    - The best ant only
- Versions :
  - Ant system (AS)
    - **All** the ants deposit pheromones **after** a solution is **complete** constructed (global collective update)
  - MaxMin Ant System (MMAS)  $\approx$  AS, but
    - The **best** ant only deposits pheromones **after** a solution is **complete** constructed (global update of the leader)
    - Deposited pheromones is **limited** to a given range
  - Ant Colony System (ACO)  $\approx$  AS, but
    - **All** the ants deposit pheromones at **each step** of solution construction (collective local update)
    - The **best** ant only deposits pheromone after the solution is complete constructed (global update of the leader)

# ACO – example

- Travelling salesman problem - TSP
  - Finds the shortest path that visits only once all the n given cities.



# ACO – example

---

## 1. Initialization:

- t := 0 (time)
- For each edge (i,j) 2 elements are initialised:
  - $\tau_{ij}^{(t)} = c$  (intensity of pheromone trail on edge (I,j) at time t)
  - $\Delta\tau_{ij} = 0$  (quantity of pheromone deposited on edge (i,j) by all the ants)
- m ants are randomly places in n city-nodes ( $m \leq n$ )
- Each ant updates its memory (list of visited cities)
  - Adds in the list the starting city

# ACO – example for TSP

---

1. While # of steps required to identify the optimal solution is not performed (# of steps = n)

- For each ant of the colony

- Increase the partial solution by an element (ant moves one step)

- Each ant k (from city i) selects the next city j:

$$j = \begin{cases} \arg \max_{l \in \text{permis}_k} \{\tau_{il}\}^\alpha [\eta_{il}]^\beta \}, & \text{if } q \leq q_0 \\ J, & \text{otherwise} \end{cases}$$

Random proportional rule

- where:

- q – random uniform number from [0,1]
- $q_0$  – parameter,  $0 \leq q_0 \leq 1$  ( $q_0 = 0 \rightarrow$  AS/MMAS, otherwise ACO)
- J is a city selected by probability

$$p_{ij}^k(t) = \begin{cases} \frac{[\tau_{ij}^{(t)}]^\alpha [\eta_{ij}]^\beta}{\sum_{s=allowed_k(t)} [\tau_{is}^{(t)}]^\alpha [\eta_{is}]^\beta}, & j - allowed \\ 0, & otherwise \end{cases}$$

Pseudo-random proportional rule

- where:

- $p_{ij}^k$  – probability of transition of ant k from city i to city j
- $\eta_{ij} = \frac{1}{d_{ij}}$  – visibility from city I towards city j (attractive choice of edge (i,j))
- $allowed_k$  – cities that can be visited by ant k at time t
- $\alpha$  – controls the trail importance (how many ants have visited that edge)
- $\beta$  – controls the visibility importance (how close is the next city)

# ACO – example for TSP

---

1. While # of steps required to identify the optimal solution is not performed
  - For each ant of the colony
    - Increase the partial solution by an element (ant moves one step)
    - Change locally the pheromone trail based on the last element added in solution

$$\tau_{ij}^{(t+1)} = (1 - \varphi) \tau_{ij}^{(t)} + \varphi * \tau_0$$

- where:
  - $\varphi$  – pheromone degradation coefficient;  $\varphi \in [0, 1]$ ; for  $\varphi = 0 \rightarrow$  AS/MMAS, otherwise ACO
  - $\tau_0$  – initial value of pheromone
  - $(i,j)$  – last edge visited by ant

# ACO – example for TSP

---

3. Change the pheromone trail from
  - **Paths covered by all ants (AS)**
    - For each edge
      - Compute the unit quantity of pheromones put by the  $k^{\text{th}}$  ant on edge  $(i,j)$ 
        - $\Delta\tau_{ij}^k = \begin{cases} Q & \text{if the } k^{\text{th}} \text{ ant used the edge } (i,j) \\ L_k & \\ 0 & \end{cases}$
        - $Q$  – quantity of pheromone deposited by an ant.
        - $L_k$  – length (cost) of tour performed by the  $k^{\text{th}}$  ant
      - Compute the total quantity of pheromone from edge  $(ij)$      $\Delta\tau_{ij} = \sum_{k=1}^m \Delta\tau_{ij}^k$
      - Compute the intensity of pheromone trail as a sum of old pheromone evaporation and the new deposited pheromone
        - $\tau_{ij}^{(t+n)} = (1 - \rho) * \tau_{ij}^{(t)} + \Delta\tau_{ij}$
        - Where  $\rho$  ( $0 < \rho < 1$ ) – evaporation coefficient of pheromone trail from a complete tour to another complete tour

# ACO – example for TSP

---

3. Change the pheromone trail from
  - **The best path (ACO)**
  - **The best path of the best ant (MMAS)**
  - For each edge of the best path
    - Compute the unit quantity of pheromone deposited by the best ant on edge  $(ij)$ 
      - $\Delta\tau_{ij} = \frac{1}{L_{best}}$
      - $L_{best}$  – length (cost) of the best path
        - Of current iteration
        - Over all executed iteration (until that time)
    - Compute the intensity of pheromone trail as sum of old pheromone evaporation and the new deposited pheromone
      - Where  $\rho$  ( $0 < \rho < 1$ ) – – evaporation coefficient of pheromone trail from a complete tour to another complete tour
      - $\tau_{min}$  și  $\tau_{max}$  – limits (inferior and superior) of pheromone;
        - For  $\tau_{min} = -\infty$  and  $\tau_{max} = +\infty \rightarrow$  ACO, otherwise MMAS

$$\tau_{ij}^{(t+n)} = [(1-\rho) * \tau_{ij}^{(t)} + \rho * \Delta\tau_{ij}^{best}]_{\tau_{min}}^{\tau_{max}}$$

# ACO – properties

---

- Properties
  - Iterative algorithm
  - Algorithm that progressively constructs the solution based on
    - Heuristic information
    - Pheromone trail
  - Stochastic algorithm
- Advantages
  - Run continuous and real-time adaptive change input
    - Ex. for TSP the graph can be dynamically changed
  - Positive feedback helps to quickly discovering of solution
  - Distributed computing avoids premature convergence
  - Greedy heuristic helps to identify an acceptable solution from the first stages of search
  - Collective interaction of individuals
- Disadvantages
  - Slowly convergence vs other heuristic search
  - For TSP instances with more than 75 cities it finds weak solutions
  - In AS there is no central process to guide the search towards good solutions

# ACO – applications

---

- Optimal paths in graphs
  - Ex. Traveling Salesman Problem
- Problems of quadratic assignments
- Problems of network optimisation
- Transport problems



# Review

---

## □ PSO

- Beam local search
- Possible solutions → particles that have:
  - A position in the search space
  - A velocity
- Cooperative and perturbative search based on:
  - Position of the best particle of the swarm
  - Best position of particle (particle has memory)

## □ ACO

- Beam local search
- Possible solutions → ants that have:
  - Memory – retain steps of solution construction
  - Smell – take decisions based on pheromones deposited by other ants (social, collective, collaborative behaviour)
- Cooperative and constructive search

---

**Thank you!**