ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

DR. CRISTIAN AXENIE

Technische Hochschule Ingolstadt

Acknowledgement

The course provides an introduction to theory and application of neuronal (deep) networks, neuromorphic systems, fuzzy control techniques, support-vector machines, evolutionary and genetic algorithms for optimization, immunological computation and artificial immune systems, reinforcement learning, distributed agent-based learning and on-line streaming machine learning. It is mainly based on the Computational Intelligence Course taught by Cristian Axenie and Prof. Jorg Conradt at TU Munich. The materials shall only be used within the class and not distributed outside.

Contents

Lectures

1. Introduction to Artificial Intelligence and Machine Learning

What is Artificial Intelligence?

According to Wikipedia:

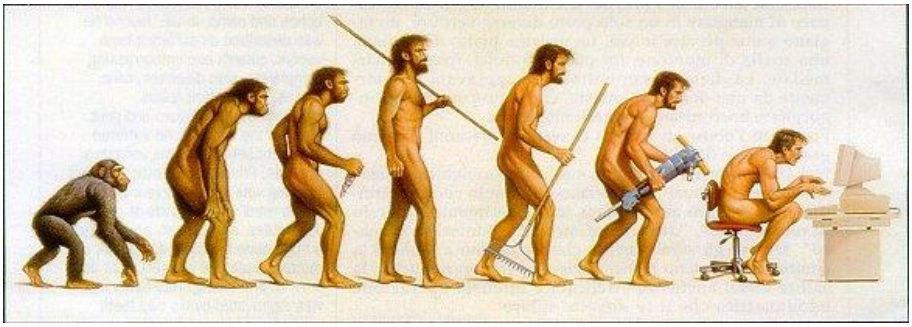
i.e. usually refers to the ability of a computer to learn a specific task from data or experimental observation. But generally, computational intelligence is a set of nature-inspired computational methodologies and approaches to address complex real-world problems to which mathematical or traditional modeling cannot offer explicit solutions.

But what is intelligence?

Chess-playing	GO playing
60's vision of intelligence: machine beating humans today achieved through brute force search and sorting on even phone CPU	the ultimate game mastered by humans, exponential solution search space 2016 a PC running artificial neural networks beats the best human player
	

But why is it computational?

Transferring principles from biology and its solutions to problems into computer systems.



In a formal way, how can we describe intelligence?

From a computational point of view, intelligence can be described as the superposition of all processing steps applied to the input a system receives to compute an output.

In computation theory this will be equivalent to a Turing machine: a mathematical model of a hypothetical computing machine which can use a predefined set of rules to determine a result from a set of input variables.

This formalism has also generated a test for machine intelligence, the Turing Test which states that: a machine is intelligent is able to exhibit intelligent behavior similar to that of a human.

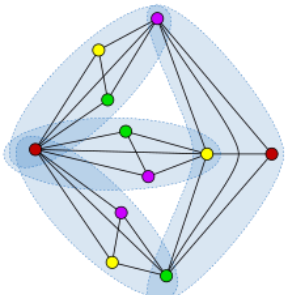
The focus of the class and brief summary

During the class we will discuss methods of processing input to extract meaning, typically in biologically inspired ways and emphasize differences from traditional computing.

Sorting

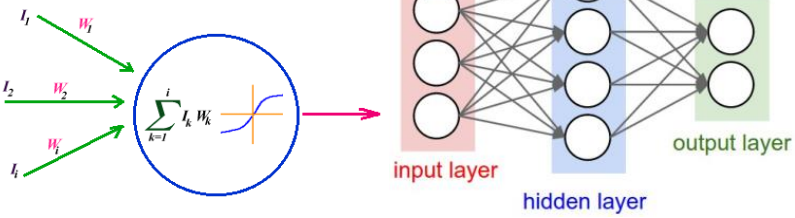
- Different strategies for sorting
- Advantages and disadvantages on large datasets

Searching in graphs

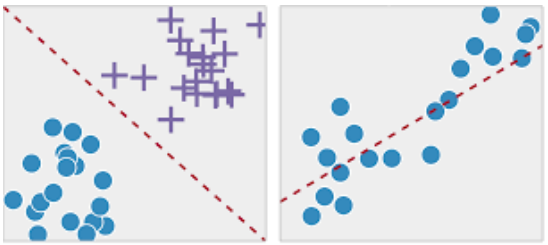


- Graph traversal algorithms
- Dynamic Programming

Neural networks

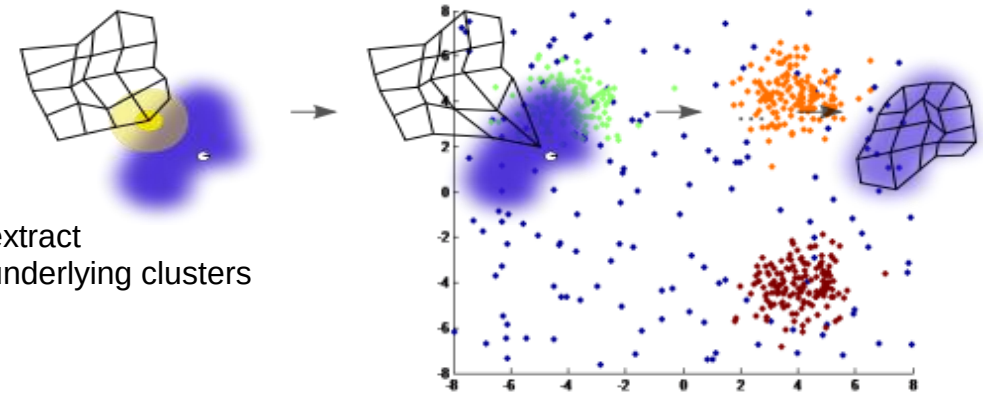


- Architecture
  - Single neuron processing
  - Multi-layer neural networks
- Tasks
  - Classification
  - Regression (function approximation)
- Learning (supervised)
  - The system has the correct (expected) answer and can improve its estimate and take decisions



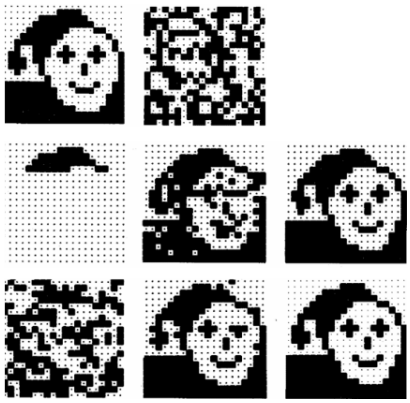
Neural networks

- Learning (unsupervised)
  - The system doesn't have the correct (expected) answer, yet such a system can:
    - learn the underlying structure of the data



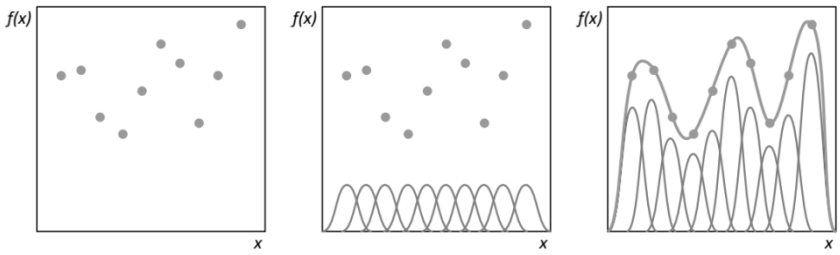
- extract underlying clusters

- reconstruct distorted patterns after previously learning them



Radial Basis Functions

Radial basis functions can take complex forms (e.g. Gaussian - real valued functions whose output depends on the distance from a particular point) and usually used for function approximation.

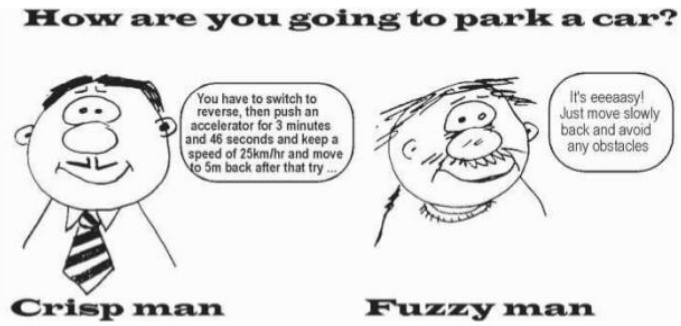


Fuzzy systems

How does human perception work?

The characteristics of human's answer will be:

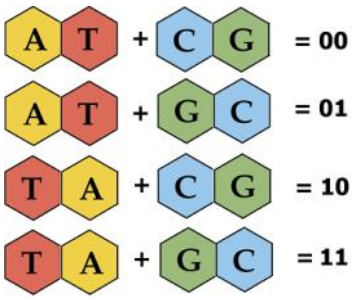
- Imprecise / vague
- Involving modifier/hedge of linguistic term (quite, fairly, too, very, etc.)
- Implies uncertainty



Fuzzy logic is based on uncertain reasoning using linguistic terms and rules based on human-like reasoning: IF x THEN y.

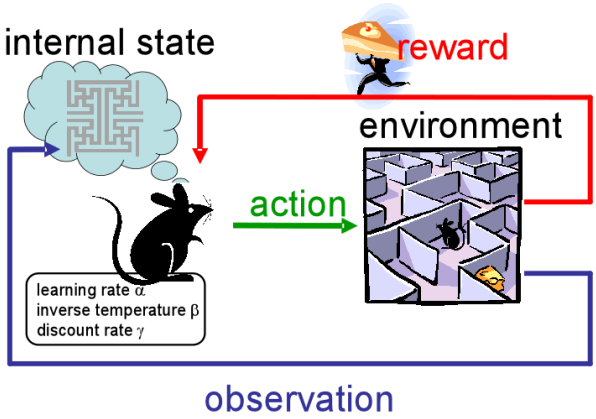
Evolutionary algorithms

Mimic biological evolution and its mechanisms for combining genetic material towards survival of the fittest. Used in optimization problems for which little is known about the underlying function.



Reinforcement Learning

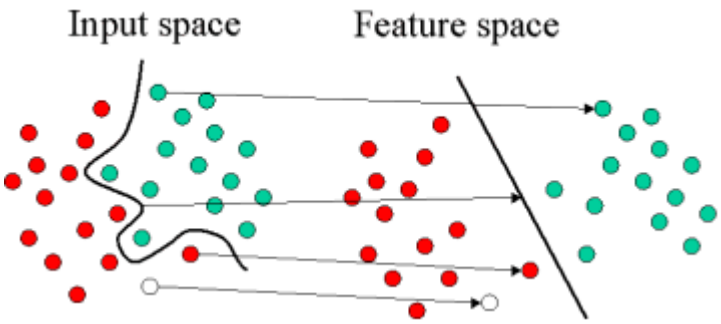
Studies how agents ought to take actions in an environment so as to maximize some notion of cumulative reward.



Statistical Learning

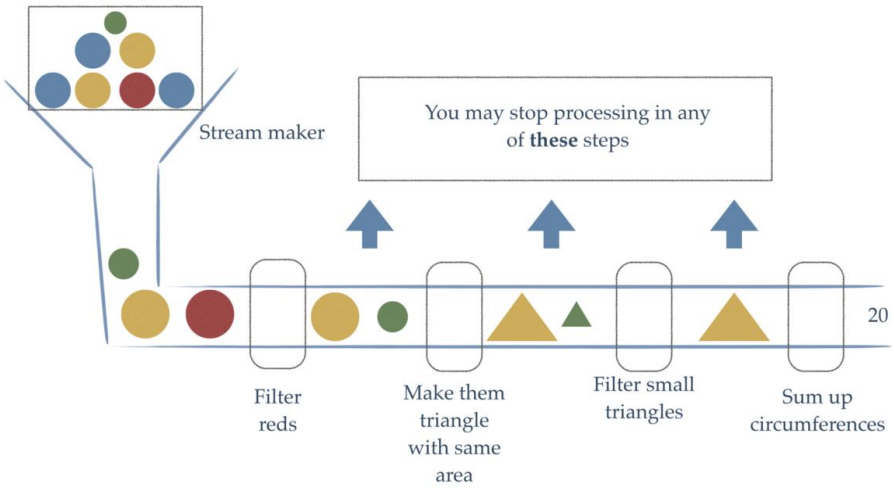
Framework for machine learning drawing from the fields of statistics and functional analysis dealing with the problem of finding a predictive function based on data.

Assumes collecting large amounts of data to recognize complex patterns, e.g. classification using Support Vector Machines.



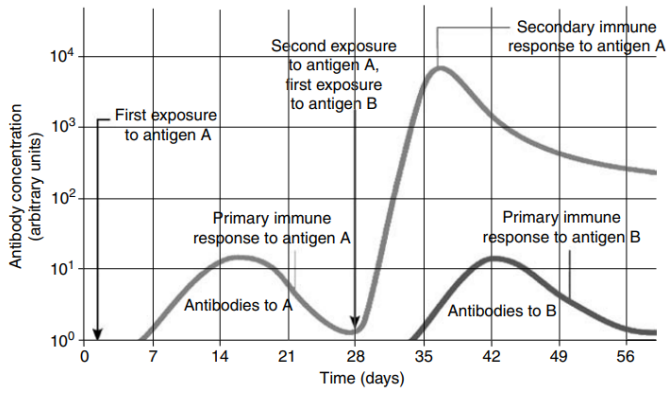
Online/Streaming Machine Learning

Stream processing paradigm simplifies parallel software and hardware by restricting the parallel computation that can be performed.
Given a **sequence of data (a stream)**, **a series of operations (functions)** is **applied** to **each element** in the stream, in a **declarative** way, we specify what we want to achieve and not how.



Immunological Computation

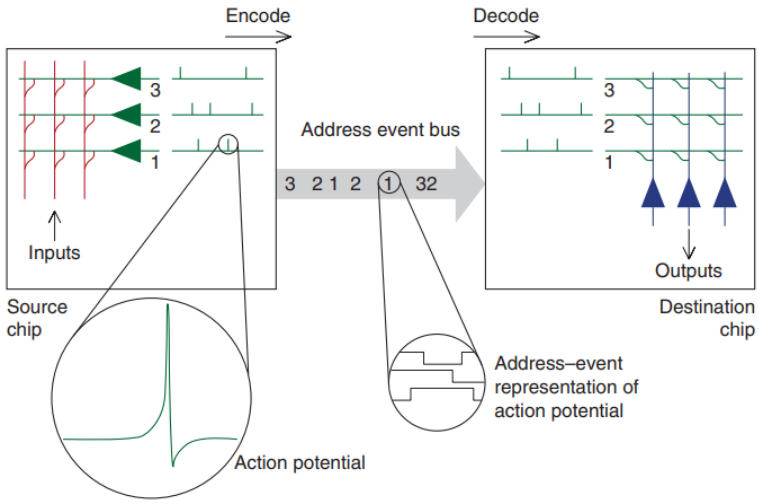
Artificial Immune Systems (AIS) is a diverse **area of research** that attempts to **bridge** the divide between **immunology** and **engineering** and are developed through the application of techniques such as **mathematical and computational modeling of immunology**, abstraction from those models into **algorithm** (and **system**) **design** and **implementation** in the context of engineering. AIS has become known as an **area of computer science and engineering** that uses **immune system metaphors** for the creation of novel solutions to problems.



Neuromorphic Computing

Neuromorphic engineering is concerned with the design and fabrication of artificial neural systems whose architecture and design principles are based on those of biological nervous systems. Neuromorphic systems of neurons and synapses can be implemented in the electronic medium CMOS (complementary metal oxide semiconductor) using hybrid analog/digital VLSI (very large-scale integrated) technology.

Asynchronous communication scheme between two chips (i.e. artificial neurons) using the address–event representation (AER).



Acknowledgement

The course provides an introduction to theory and application of neuronal (deep) networks, neuromorphic systems, fuzzy control techniques, support-vector machines, evolutionary and genetic algorithms for optimization, immunological computation and artificial immune systems, reinforcement learning, distributed agent-based learning and on-line streaming machine learning. It is mainly based on the Computational Intelligence Course taught by Cristian Axenie and Prof. Jorg Conradt at TU Munich. The materials shall only be used within the class and not distributed outside.

Contents

Lectures

1. Introduction to Artificial Intelligence and Machine Learning

What is Artificial Intelligence?

According to Wikipedia:

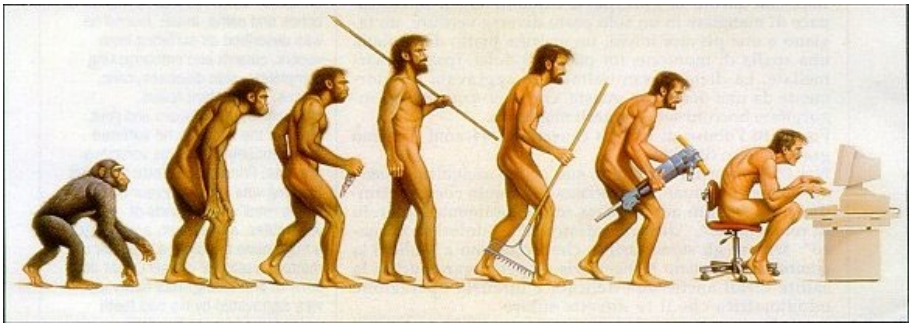
i.e. usually refers to the ability of a computer to learn a specific task from data or experimental observation. But generally, computational intelligence is a set of nature-inspired computational methodologies and approaches to address complex real-world problems to which mathematical or traditional modeling cannot offer explicit solutions.

But what is intelligence?

Chess-playing	GO playing
60's vision of intelligence: machine beating humans today achieved through brute force search and sorting on even phone CPU	the ultimate game mastered by humans, exponential solution search space 2016 a PC running artificial neural networks beats the best human player
	

But why is it computational?

Transferring principles from biology and its solutions to problems into computer systems.



In a formal way, how can we describe intelligence?

From a computational point of view, intelligence can be described as the superposition of all processing steps applied to the input a system receives to compute an output.

In computation theory this will be equivalent to a Turing machine: a mathematical model of a hypothetical computing machine which can use a predefined set of rules to determine a result from a set of input variables.

This formalism has also generated a test for machine intelligence, the Turing Test which states that: a machine is intelligent is able to exhibit intelligent behavior similar to that of a human.

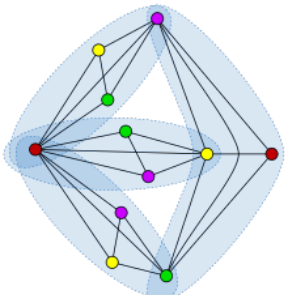
The focus of the class and brief summary

During the class we will discuss methods of processing input to extract meaning, typically in biologically inspired ways and emphasize differences from traditional computing.

Sorting

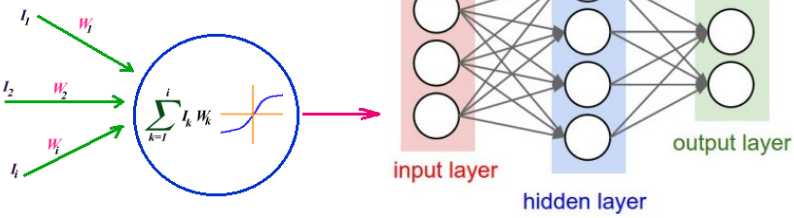
- Different strategies for sorting
- Advantages and disadvantages on large datasets

Searching in graphs

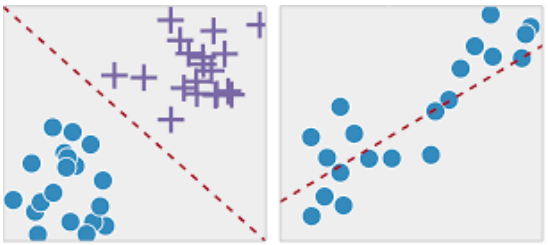


- Graph traversal algorithms
- Dynamic Programming

Neural networks

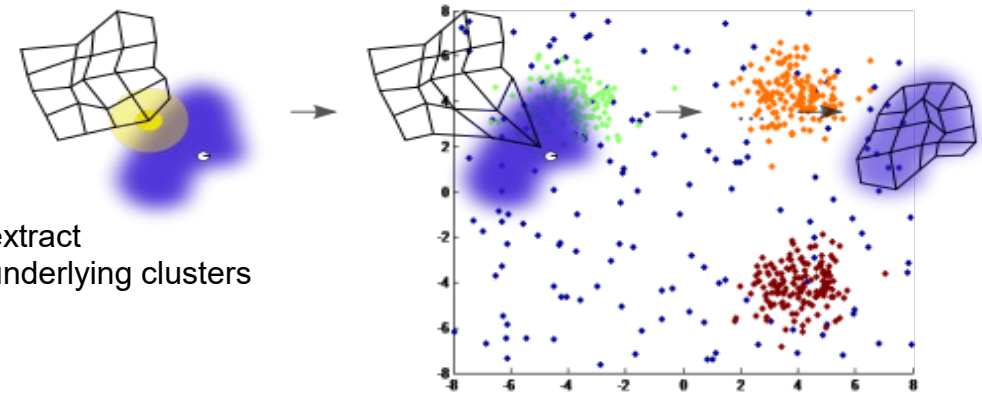


- Architecture
  - Single neuron processing
  - Multi-layer neural networks
- Tasks
  - Classification
  - Regression (function approximation)
- Learning (supervised)
  - The system has the correct (expected) answer and can improve its estimate and take decisions



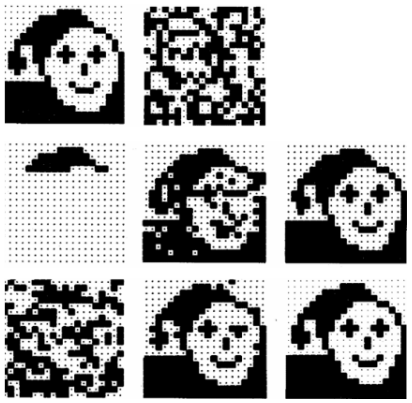
Neural networks

- Learning (unsupervised)
  - The system doesn't have the correct (expected) answer, yet such a system can:
    - learn the underlying structure of the data



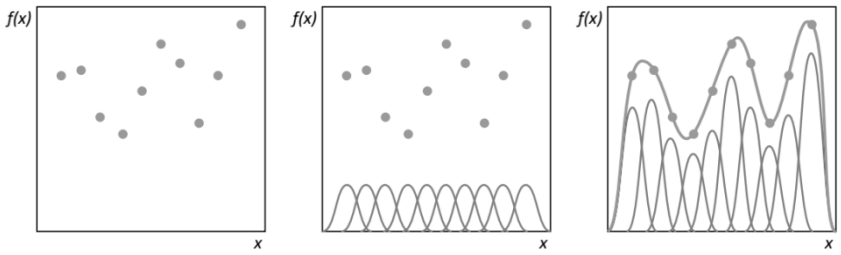
- extract underlying clusters

- reconstruct distorted patterns after previously learning them



Radial Basis Functions

Radial basis functions can take complex forms (e.g. Gaussian - real valued functions whose output depends on the distance from a particular point) and usually used for function approximation.

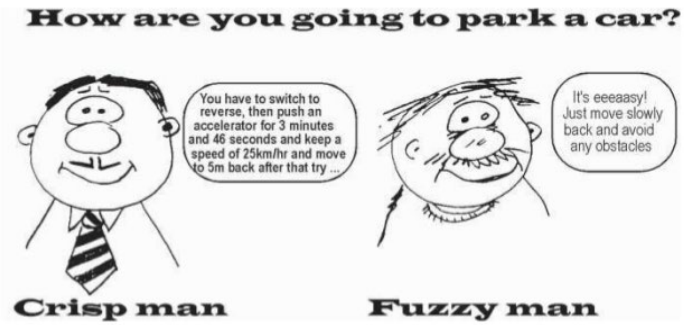


Fuzzy systems

How does human perception work?

The characteristics of human's answer will be:

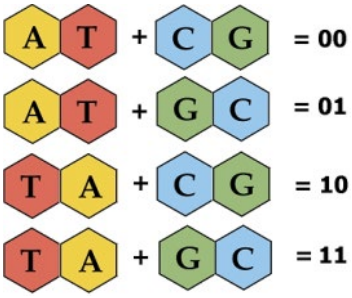
- Imprecise / vague
- Involving modifier/hedge of linguistic term (quite, fairly, too, very, etc.)
- Implies uncertainty



Fuzzy logic is based on uncertain reasoning using linguistic terms and rules based on human-like reasoning: IF x THEN y.

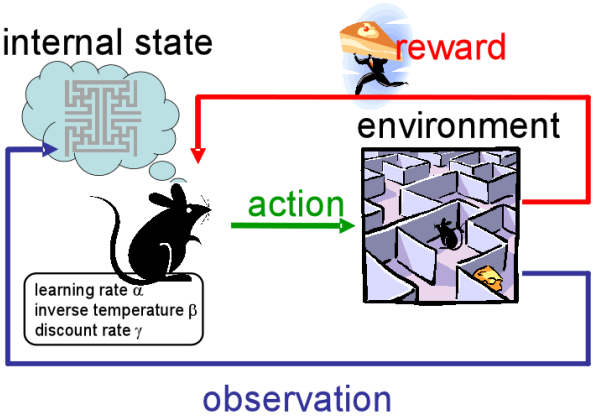
Evolutionary algorithms

Mimic biological evolution and its mechanisms for combining genetic material towards survival of the fittest. Used in optimization problems for which little is known about the underlying function.



Reinforcement Learning

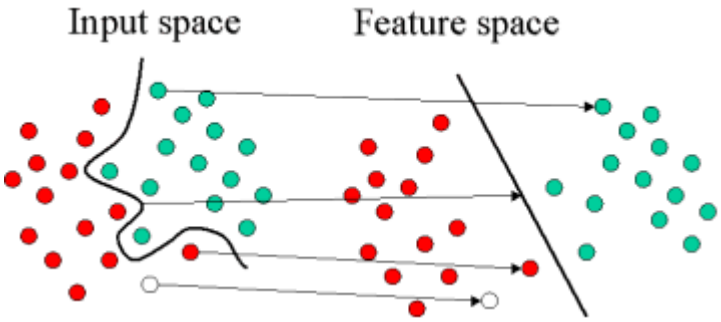
Studies how agents ought to take actions in an environment so as to maximize some notion of cumulative reward.



Statistical Learning

Framework for machine learning drawing from the fields of statistics and functional analysis dealing with the problem of finding a predictive function based on data.

Assumes collecting large amounts of data to recognize complex patterns, e.g. classification using Support Vector Machines.

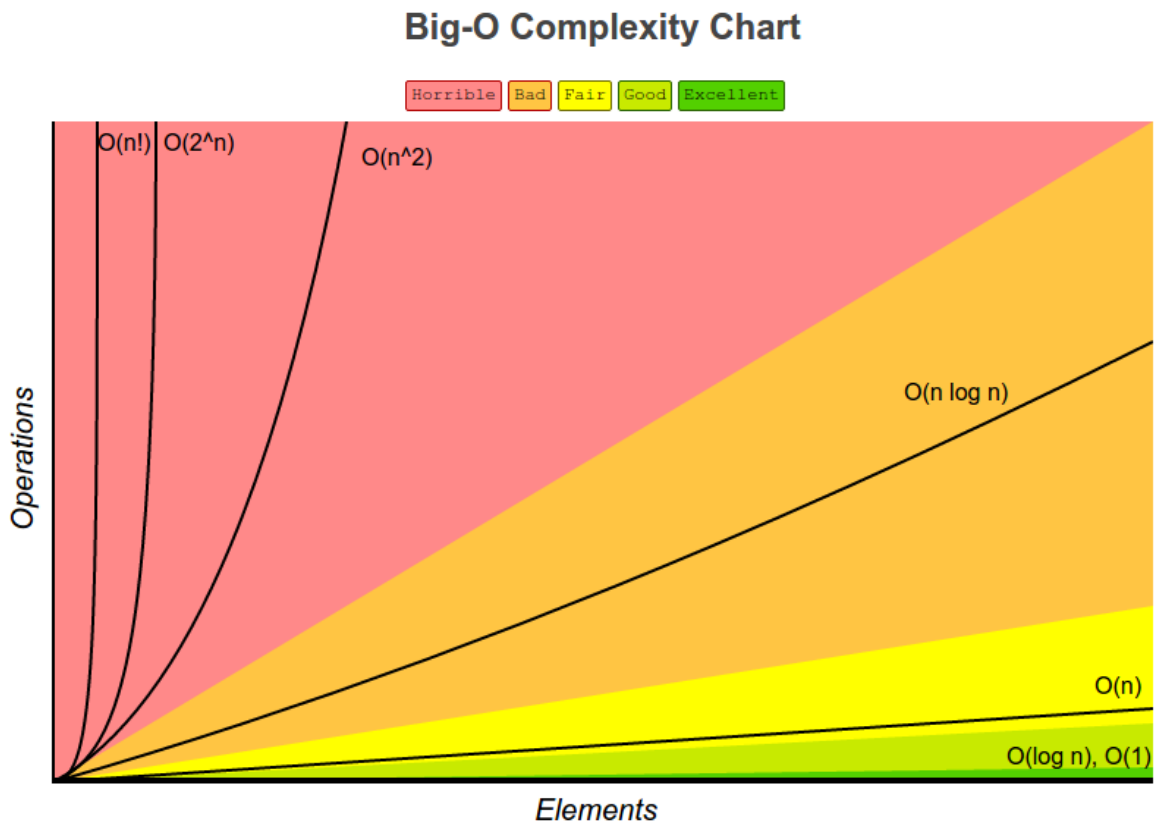


Contents

Lectures

2. Traditional computation

A sorting algorithm is an algorithm that organizes elements of a sequence in a certain order. Since the early days of computing, the sorting problem has been one of the main battlefields for researchers. The basic metric usually used in the analysis of algorithms is depicted in the following figure.



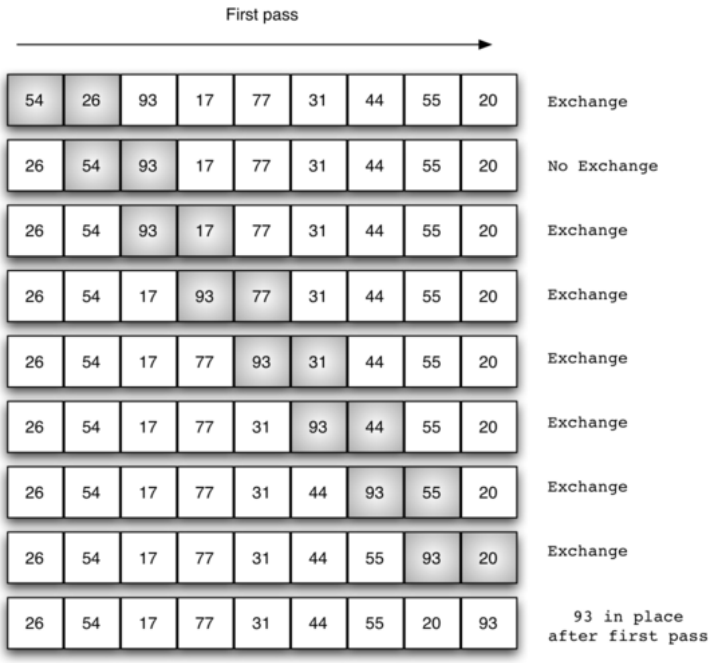
The reason behind this is not only the need of solving a very common task but also the challenge of solving a complex problem in the most efficient way (in terms of memory usage and time).

2.1 Sorting algorithms

Bubble sort

The bubble sort makes multiple passes through a list. It compares adjacent items and exchanges those that are out of order. Each pass through the list places the next largest value in its proper place. In essence, each item “bubbles” up to the location where it belongs.

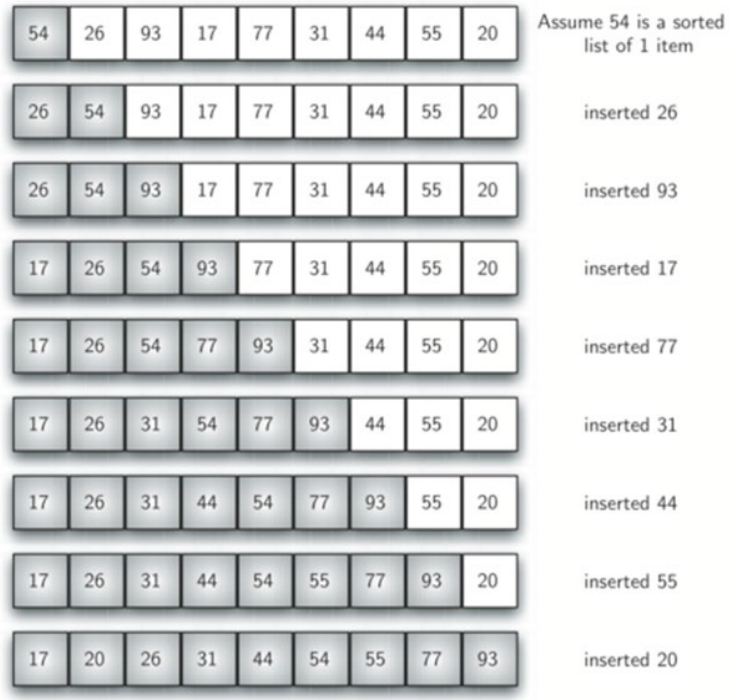
The basic process in depicted in the following figure. The shaded items are being compared to see if they are out of order. It is important to note that once the largest value in the list is part of a pair, it will continually be moved along until the pass is complete.



Insertion sort

Insertion sort always maintains a sorted sublist in the lower positions of the list. Each new item is then “inserted” back into the previous sublist such that the sorted sublist is one item larger.

The basic process in depicted in the following figure. The shaded items represent the ordered sublists as the algorithm makes each pass.

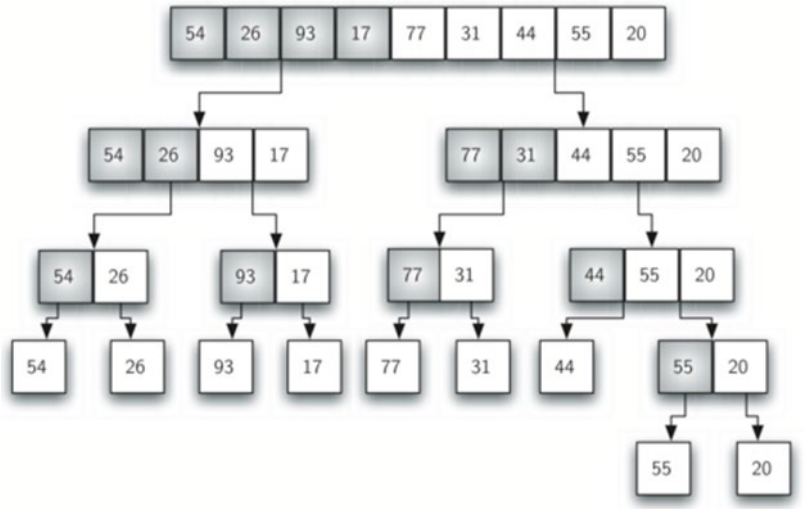


Merge sort

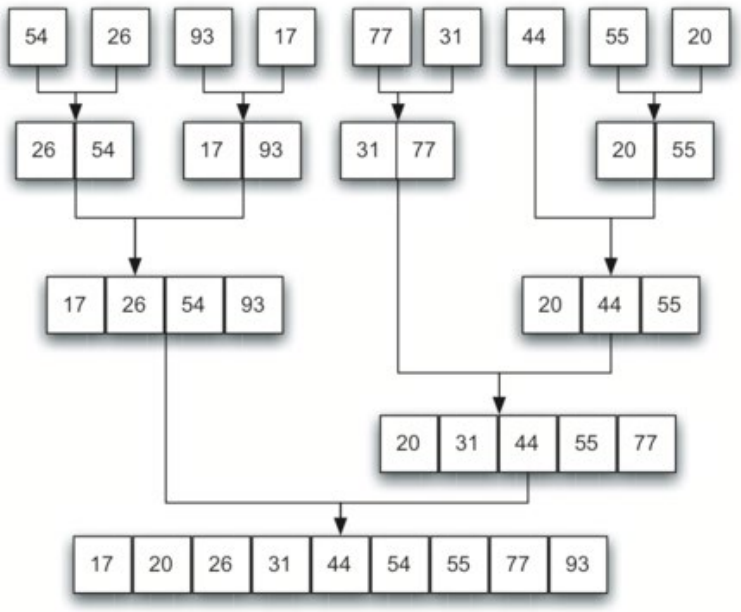
It is based on recursion and implements the Divide and Conquer strategy as a way to improve the performance of sorting algorithms.

Merge sort is a recursive algorithm that continually splits a list in half. If the list is empty or has one item, it is sorted by definition (the base case). If the list has more than one item, we split the list and recursively invoke a merge sort on both halves. Once the two halves are sorted, the fundamental operation, called a merge, is performed. Merging is the process of taking two smaller sorted lists and combining them together into a single, sorted, new list. The basic process in depicted in the following figure.

Divide



Conquer



Comparison among sorting algorithms in terms of complexity

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Timsort	O(n)	O(n log(n))	O(n log(n))	O(n)
Heapsort	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Tree Sort	O(n log(n))	O(n log(n))	O(n^2)	O(n)
Shell Sort	O(n log(n))	O(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)
Counting Sort	O(n+k)	O(n+k)	O(n+k)	O(k)
Cubesort	O(n)	O(n log(n))	O(n log(n))	O(n)

2.2 Graph search algorithms

Graphs and their formal definition

Graphs can be used to represent many interesting things about our world, including systems of roads, airline flights from city to city, how the Internet is connected, or even the sequence of classes you must take to complete a major in computer science.

In this section we will formally define a graph and its components.

Vertex

A vertex (also called a “node”) is a fundamental part of a graph. It can have a name, which we will call the “key.” A vertex may also have additional information. We will call this additional information the “payload.”

Edge

An edge (also called an “arc”) is another fundamental part of a graph. An edge connects two vertices to show that there is a relationship between them. Edges may be one-way or two-way. If the edges in a graph are all one-way, we say that the graph is a directed graph, or a digraph. The class prerequisites graph shown above is clearly a digraph since you must take some classes before others.

Weight

Edges may be weighted to show that there is a cost to go from one vertex to another. For example in a graph of roads that connect one city to another, the weight on the edge might represent the distance between the two cities.

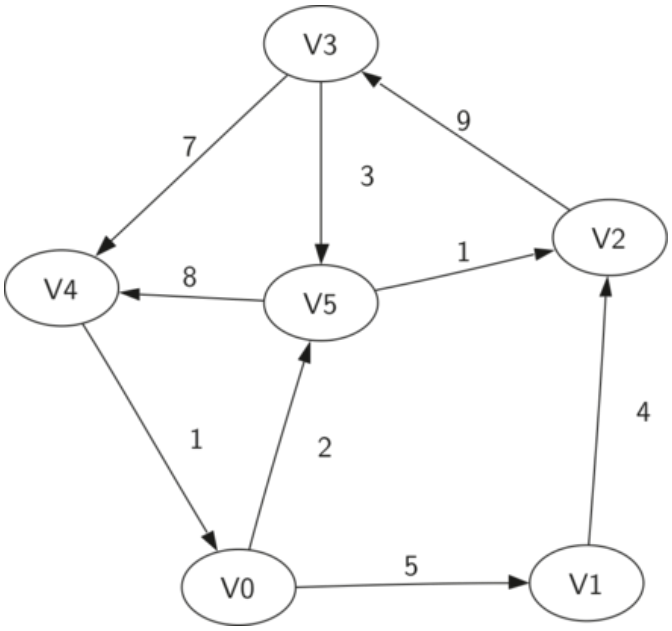
Path

A path in a graph is a sequence of vertices that are connected by edges.

Cycle

A cycle in a directed graph is a path that starts and ends at the same vertex.

With those definitions in hand we can formally define a graph. A graph can be represented by G where G = (V, E). For the graph G, V is a set of vertices and E is a set of edges. Each edge is a tuple (v, w) where w, v ∈ V. We can add a third component to the edge tuple to represent a weight. A subgraph s is a set of edges e and vertices v such that e ⊂ E and v ⊂ V. A sample graph is shown in the following figure.



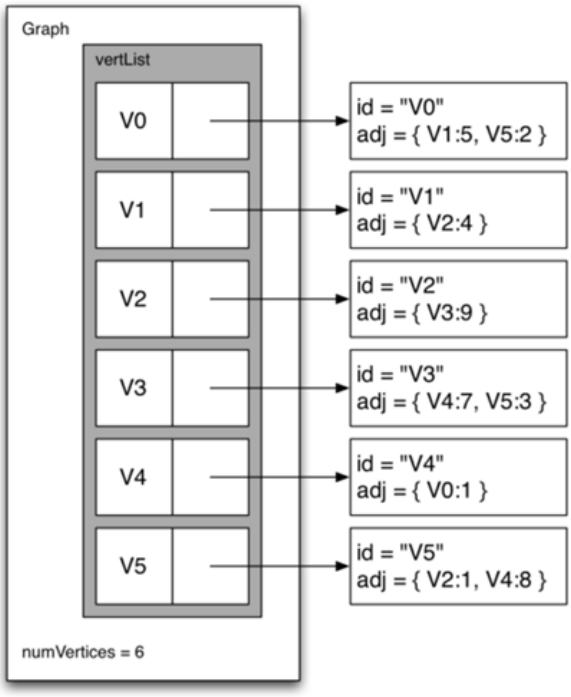
Representing graphs: Adjacency matrix

One of the easiest ways to implement a graph is to use a two-dimensional matrix. In this matrix implementation, each of the rows and columns represent a vertex in the graph. The value that is stored in the cell at the intersection of row v and column w indicates if there is an edge from vertex v to vertex w. When two vertices are connected by an edge, we say that they are adjacent. The adjacency matrix for the previously introduced graph is depicted in the following diagram.

	V0	V1	V2	V3	V4	V5
V0		5				2
V1			4			
V2				9		
V3					7	3
V4	1					
V5			1		8	

Representing graphs: Adjacency list

A more space-efficient way to implement a sparsely connected graph is to use an adjacency list. In an adjacency list implementation we keep a master list of all the vertices in the Graph object and then each vertex object in the graph maintains a list of the other vertices that it is connected to. The adjacency list for the previously introduced graph is shown in the following diagram.



The advantage of the adjacency list implementation is that it allows us to compactly represent a sparse graph. The adjacency list also allows us to easily find all the links that are directly connected to a particular vertex.

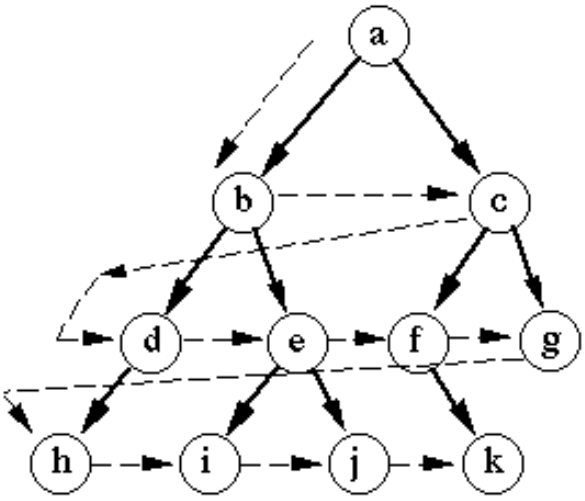
Algorithms on graphs

Graphs can be used to model many types of relations and processes in physical, biological, social and information systems. Many practical problems can be represented by graphs. One interesting problem is graphs search, or graph traversal. It refers to the process of visiting (checking and/or updating) each vertex in a graph. Such traversals are classified by the order in which the vertices are visited.

Breadth First Search (BFS)

Breadth first search (BFS) is one of the easiest algorithms for searching a graph. Given a graph G and a starting vertex s, a breadth first search proceeds by exploring edges in the graph to find all the vertices in G for which there is a path from s. The remarkable thing about a breadth first search is that it finds all the vertices that are a distance k from s before it finds any vertices that are a distance k+1. One good way to visualize what the breadth first search algorithm does is to imagine that it is building a tree, one level of the tree at a time. A breadth first search adds all children of the starting vertex before it begins to discover any of the grandchildren.

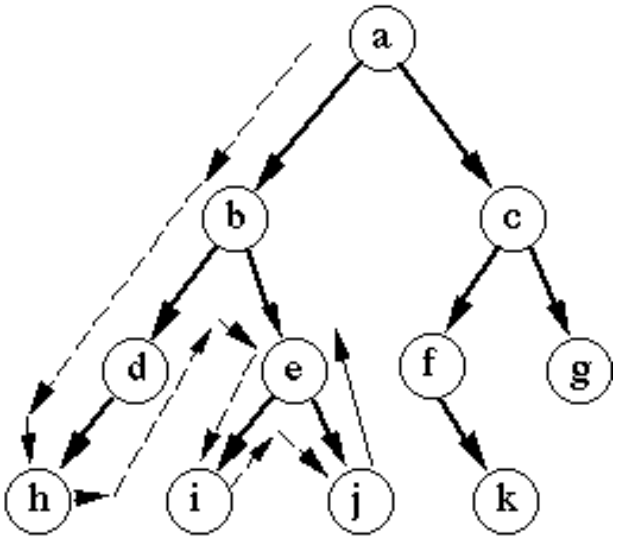
To keep track of its progress, BFS colors each of the vertices white, gray, or black. All the vertices are initialized to white when they are constructed. A white vertex is an undiscovered vertex. When a vertex is initially discovered it is colored gray, and when BFS has completely explored a vertex it is colored black. This means that once a vertex is colored black, it has no white vertices adjacent to it. A gray node, on the other hand, may have some white vertices adjacent to it, indicating that there are still additional vertices to explore. The basic process in depicted in the following figure.



Depth First Search (DFS)

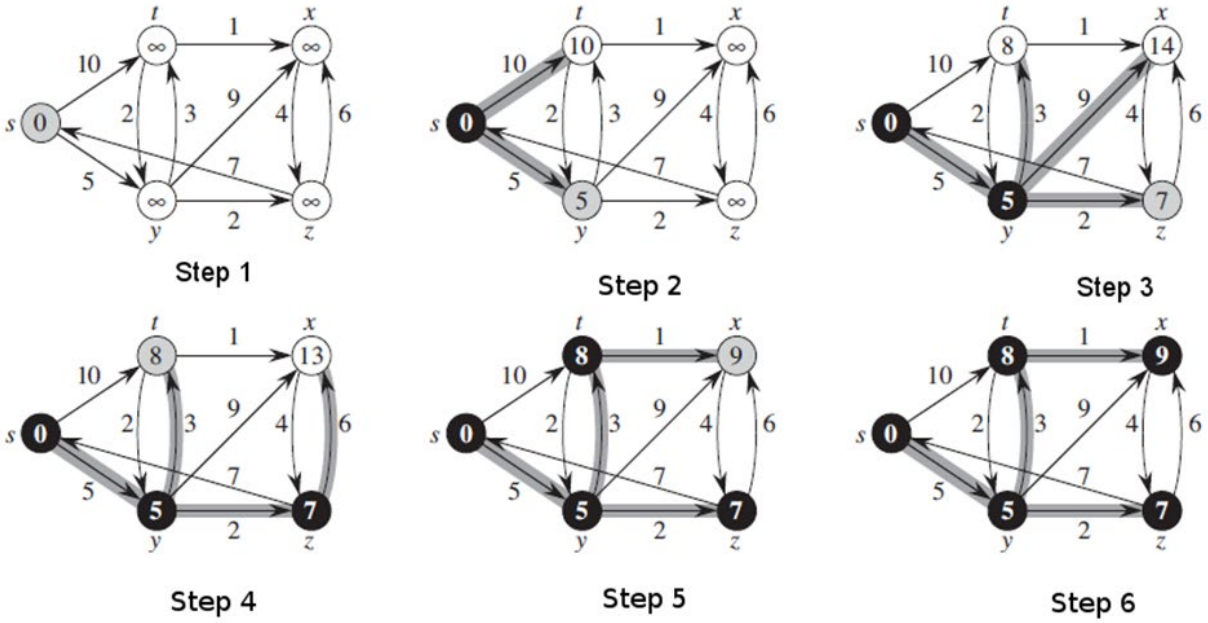
The goal of DFS is to search as deeply as possible, connecting as many nodes in the graph as possible and branching where necessary. As with the breadth first search our depth first search makes use of predecessor links to construct the tree. The

difference is that the DFS introduces the vertices at the beginning of the queue instead of the end. The basic process in depicted in the following figure.



Dijkstra's Algorithm

Dijkstra's algorithm is an iterative algorithm that provides us with the shortest path from one particular starting vertex to all other vertices in the graph. This algorithm extends the previously introduced approaches by introduces a cost on each edge. The basic process in depicted in the following figure.



A algorithm*

A* (pronounced "A - star") is one of the most popular methods for finding the shortest path between two locations in a mapped area. A* was developed in 1968 to combine

heuristic approaches like Best-First-Search (BFS) and formal approaches like Dijsktra's algorithm.

It uses a cost function, the sum of the path cost and a heuristic, such as Cartesian distance to goal. This algorithm will guide the search and will not explore the entire solution space as Dijkstra’s.

Contents

Lectures

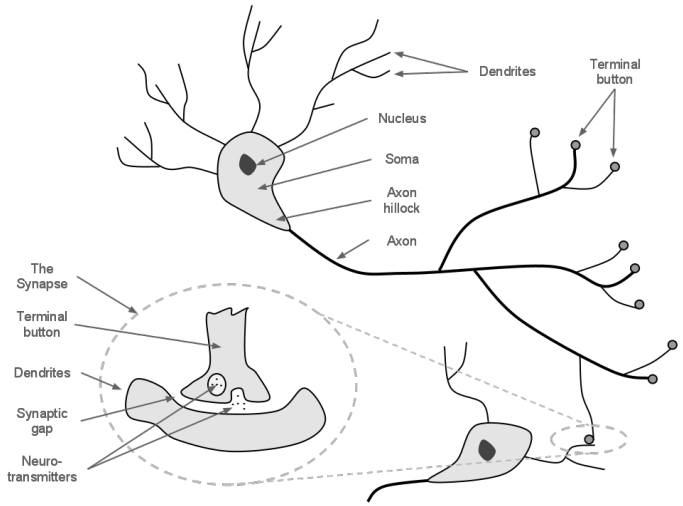
3. Supervised neural computation

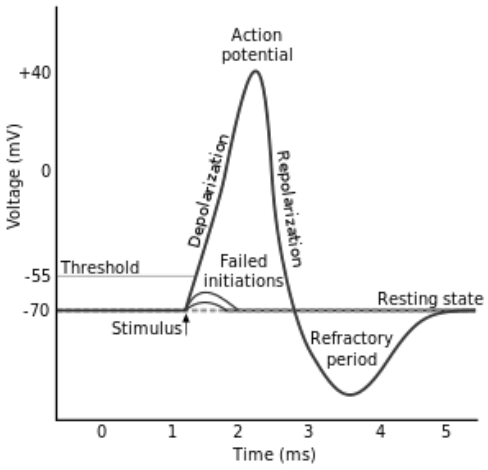
3.1 Biological neurons vs. artificial neurons

Neural networks have a remarkable ability to **extract meaning** from **complex**, **imprecise**, and often **noisy** data. They are able to **learn patterns and trends** governing the data which are typically not visible to humans. This is due to their ability to **generalize** and to **respond** to **unexpected** inputs/patterns.

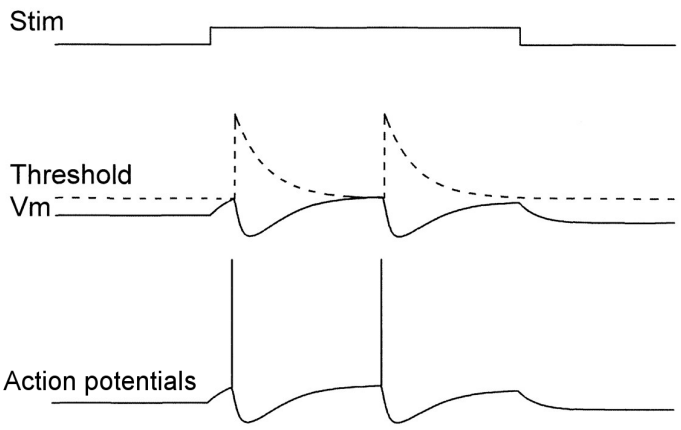
The brain is a highly complex, nonlinear, and parallel computer (information-processing system). It has the capability to organize its structural constituents, known as neurons, so as to perform certain computations (e.g., pattern recognition, perception, and motor control) many times more efficient than the fastest digital computer in existence today. On short timescales, one can conceive of a **single neuron** as a **computational device** that **maps** inputs at its synapses into a sequence of action potentials or spikes.

A **biological neuron** and its **physiological** properties (e.g. membrane voltage)





What type of **computation** happens inside a cell?

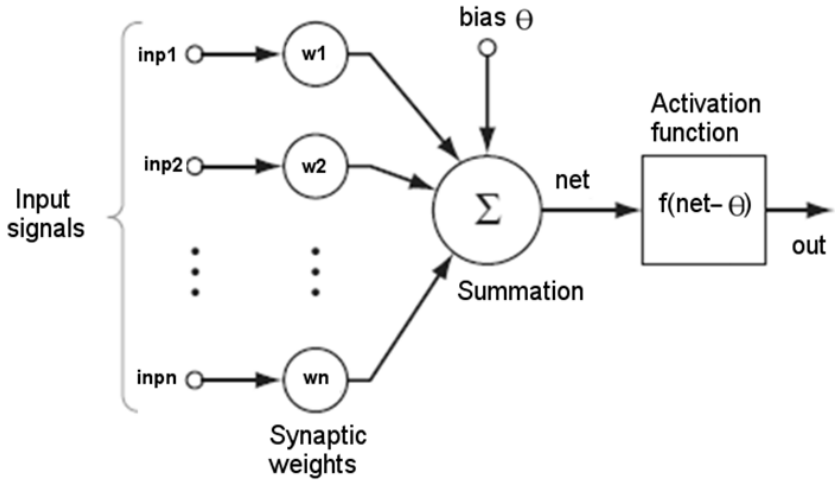


Input signal (i.e. stimulus)

Cell membrane voltage changes

Output (i.e. spike / action potential)

An **artificial neuron** emulates the process of **collecting input signals** (i.e. **pre-synaptic spikes**) and providing an **output** after reaching a threshold (i.e. **post-synaptic spike**).



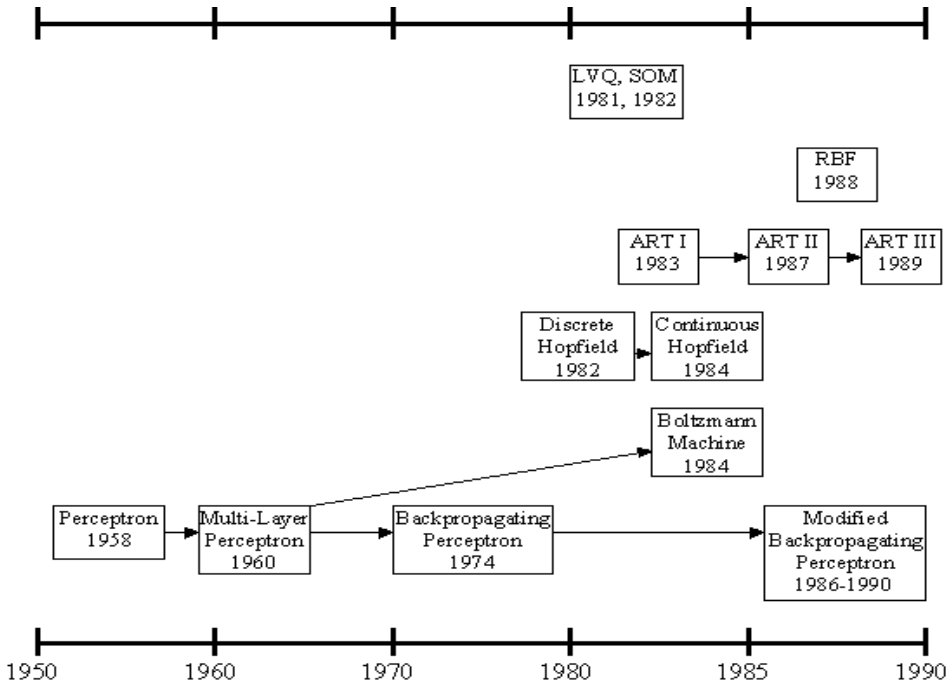
The **net input** of the neuron is given by

$$net = \sum_{i=1}^{n} p_i w_i = p_1 w_1 + p_2 w_2 + \cdots + p_n w_n$$

whereas the output of the neuron is computed as

$$out = f(net - \theta)$$

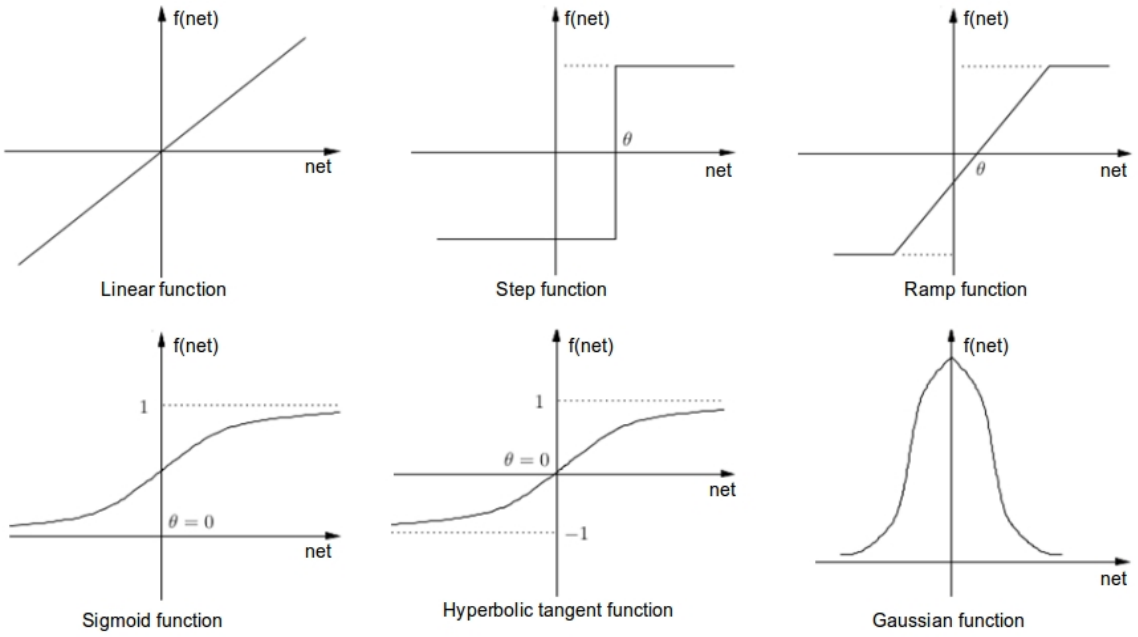
Although the history of the artificial neural networks stems from the 1940s, the decade of the first electronic computer, the first significant step took place in **1958**, when **Rosenblatt** introduced the first concrete neural model, the **perceptron**.



An artificial neuron performs a (**nonlinear**) **mapping** from **input to output**, typically from a **high-dimensional input** space to a **low-dimensional (often one-dimensional) output** space.

Mathematically, the computation described by an artificial neuron describes the calculation of a **weighted sum of its inputs** and the application of an **activation** (**squashing**) **function** that determines the output of the neuron.

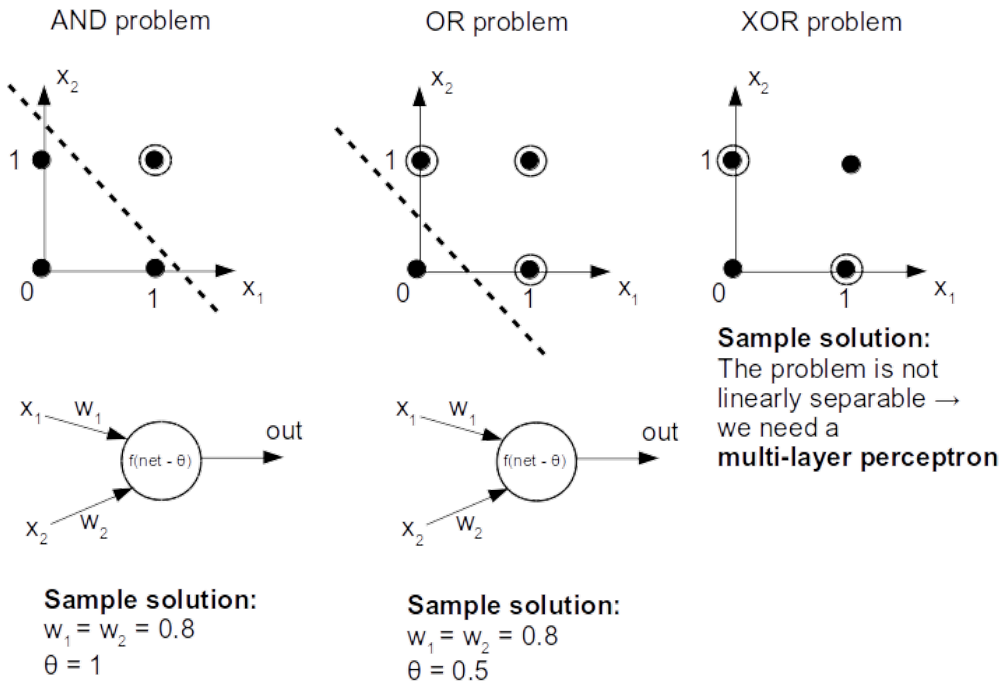
In most cases the **activation functions** are **monotonically increasing functions** with different effects on the output of the neuron.



What can an **"artificial neuron" compute**?

A **two-input perceptron** with **one neuron** which has a "step" activation function is capable to **separate / classify input patterns** (e.g. in the simplest case group inputs in classes, <0 or ≥0). Such a **classification** implies a **decision / separation boundary**, which is determined by the input vectors for which the net input is zero.

Considering **linearly separable** classes a single neuron perceptron can implement problems emulating logic gates, such as **AND** and **OR functions.**



In order to construct **linear decision boundaries** that explicitly try to separate the data into different classes as well as possible, we need **to train the system** by **updating the weights** (**both for inputs and the bias**). In other words, the system needs to find a separating **hyperplane** by **minimizing the distance** of misclassified points to the **decision boundary**.

3.2 Learning in artificial neurons

The training process is based on the **Gradient Descent Learning Rule**, which assumes **minimizing an error function** of the **mismatch between the target and the actual output** of the neuron.

The actual **error metric** is given by

$$E(t) = \sum_{p=1}^{P_T} \left(t_p(t) - out_p(t)\right)^2,$$

where t_p is **the target** value for pattern p, and out_p is the currently computed **neuron output** for input pattern p from the training set. The error metric is **squared,** such that all errors are positive and large errors are stronger penalized compared to small errors.

For every single training pattern, **weight update** uses this rule to follow the negative Gradient in weight space; i.e. ultimately go to the position in weight space with smallest output error.

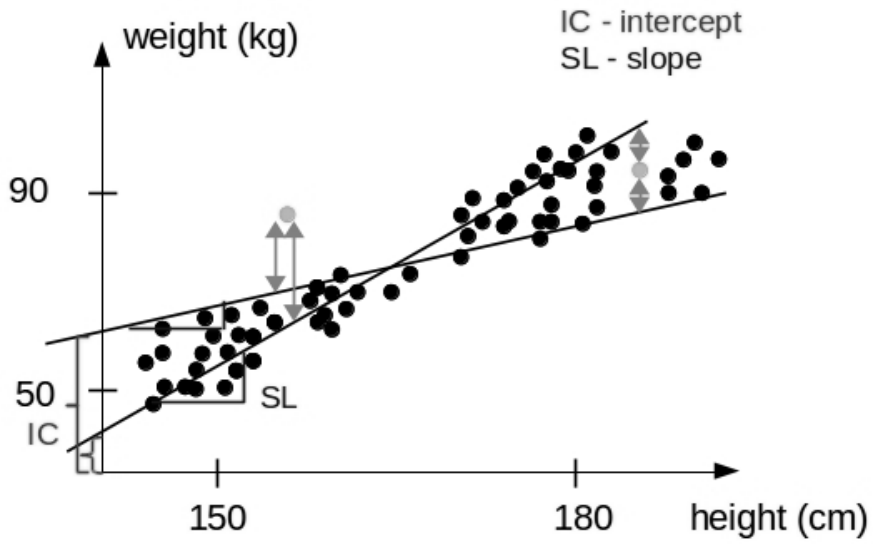
$$w_i(t) = w_i(t-1) + \Delta w_i(t)$$

$$\Delta w_i(t) = \eta\left(\frac{-\partial E(t)}{\partial w_i(t)}\right)$$

If the classes are **linearly separable**, the **algorithm converges** to a separating hyperplane in a **finite number of steps**.

What does **learning** mean? Basically, a **learning process** assumes fitting a model to data. In the context of neural computation, the **model** is the **neuron** or the **neural network**, and the fitting process assumes the **update of weights**.

In order to understand this basic process, we provide a simple example that uses the dependency between the weight and the height of a group of people as training data. The neural system should be able to extract such a mapping from the data.

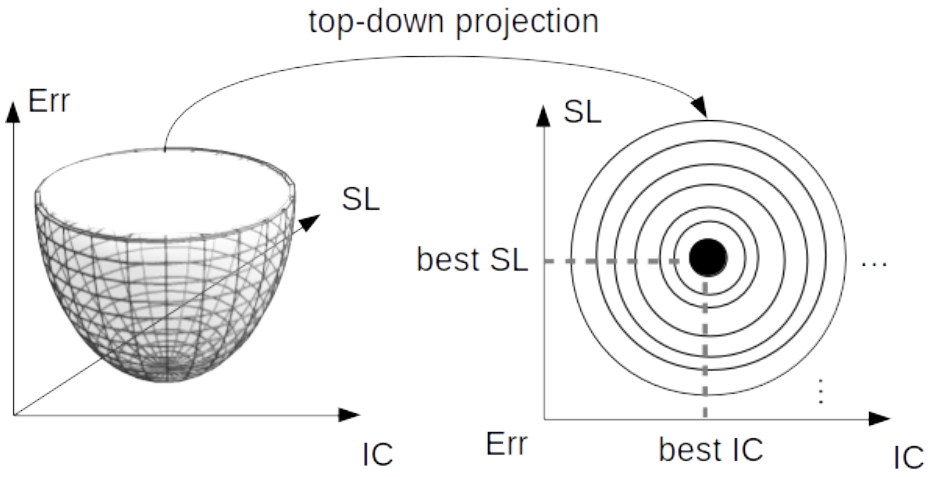


How do future (currently unknown) data points fit? What is the mapping between the weight and the height? There are several possible “models” (multiple lines) that can fit the data; each characterized by a slope (SL) and an intercept (IC).

In these terms we need to define **a metric** that decides **how good** or **how bad our particular choice of the model is**. A typical **error function** is the sum squared error between the true value and the output (as explained above):

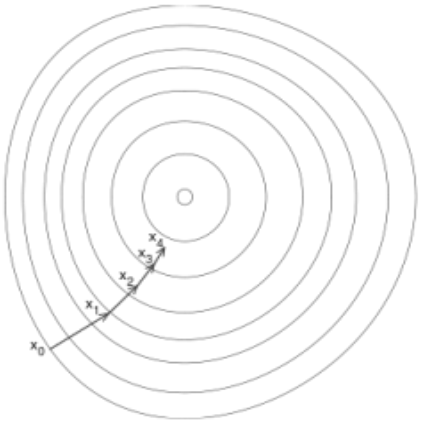
$$Err = \sum_{points} \left(true_p - out_p\right)^2$$

Finding the “best” **combination of SL and IC minimizes the “error”** of the model.



How do we achieve this? In order to guide the search for the suitable parameters Gradient Descent is used:

1. Pick **random initial values** for IC/SL (e.g. x_0)
2. **Calculate the gradient** with respect to each model parameter (i.e. IC,SL)
3. **Update the parameters** in the **direction** of the negative **gradient**
4. **Repeat** 2 and 3 until **convergence** (e.g. $x_{1...4}$)



How does the process of gradient descent relate to neurons?

As previously shown a neuron integrates the available input, inp_i, weighting each contribution, w_i.

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

For the previous example, we assume only one quantity as input (e.g. height) and predict the other quantity (e.g. weight) as output. The "linear neuron" adapts its two "internal weights" (w_1 connected to input height; and w_2 to bias input), such that w_1 and w_2 become the unknown value IC and SL that characterize the line which approximates the data.

This simple example can be formalized to a **learning rule in a single neuron**.

We define the error signal for the entire dataset, E, and compute the error for each single training example, E_p, using the current set of weights w_i:

$$E = \sum_{points} \left(true_p - out_p\right)^2 \rightarrow Ep = \left(true_p - out_p\right)^2$$

Given the error signal, we compute the gradient (derivative) with respect to the input weights of a linear neuron (i.e. here for simplicity the **activation function is linear**)

$$\Delta w_i(t) = \eta G_i(t), \qquad \text{with} \quad G_i(t) = \frac{\partial E_p(t)}{\partial w_i(t)},$$

which we can rewrite as (chain rule)

$$G_i(t) = \frac{\partial E_p(t)}{\partial w_i(t)} = \frac{\partial E_p(t)}{\partial out_p(t)} \cdot \frac{\partial out_p(t)}{\partial w_i(t)} \text{with } \; out_p(t) = \sum_{i=1}^{n} p_i(t) \cdot w_i\,(t)$$

Analyzing both factors individually yields

$$(1)\ \frac{\partial E_p(t)}{\partial out_p(t)} = \frac{\partial \left(true_p - out_p\right)^2}{\partial out_p(t)} = 2 \cdot \left(true_p - out_p\right) \cdot (-1) = -2 \cdot \left(true_p - out_p\right)$$

and

$$(2)\ \frac{\partial out_p(t)}{\partial w_i(t)} = \frac{\partial \sum_{j=1} p_j(t) \cdot w_j(t)}{\partial w_i(t)} = p_i \qquad \text{(only for j=i the derivative exists)}$$

Combined, the **gradient in the direction of the weight w_i** is given by

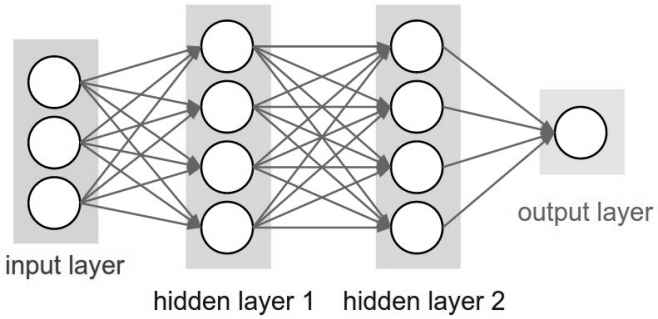
$$G_i(t) = \frac{\partial E_p(t)}{\partial w_i(t)} = \frac{\partial E_p(t)}{\partial out_p(t)} \cdot \frac{\partial out_p(t)}{\partial w_i(t)} = -2 \cdot \left(true_p - out_p\right) \cdot \in p_i$$

and we can use this to **adapt weights**, such that the **error value** is **minimized**.

This approach holds also for neurons with **non-linear activation functions**, as long as this activation function is **differentiable**.

3.3 From single neurons to neural networks

So far we introduced a simple neuron that can adapt to available data; if and only if the distribution of data matches its activation function (linear in the example above). The advanced goal is to **learn "arbitrary" data**, not just such that happens to fit the given neuron transfer function. This we achieve by combining multiple neurons in a **neural network** using feed forward **connectivity** between the neuron layers. In such a structure each **neuron represents some aspect of the data** and the neurons higher up in the **hierarchy** combine these.



The manner in which the neurons of a neural network are structured is intimately linked with the **learning algorithm** used to train the network. In a **layered neural network**, the neurons are organized in the form of layers. The simplest form of a layered network has an input layer of (external) source nodes that projects directly to an output layer of neurons (computational nodes), but not vice versa. This network is strictly **feedforward**! More complex feedforward neural networks additionally contain one or more hidden layers, whose computational nodes are correspondingly called **hidden neurons** or **hidden units**; the term “hidden” refers to the fact that this part of the neural network is not seen directly from either the input or the output of the network. Another class of network structures are **recurrent neural** networks with at least one **feedback loop**; we will analyze them when introducing unsupervised learning algorithms (Chapter 4).

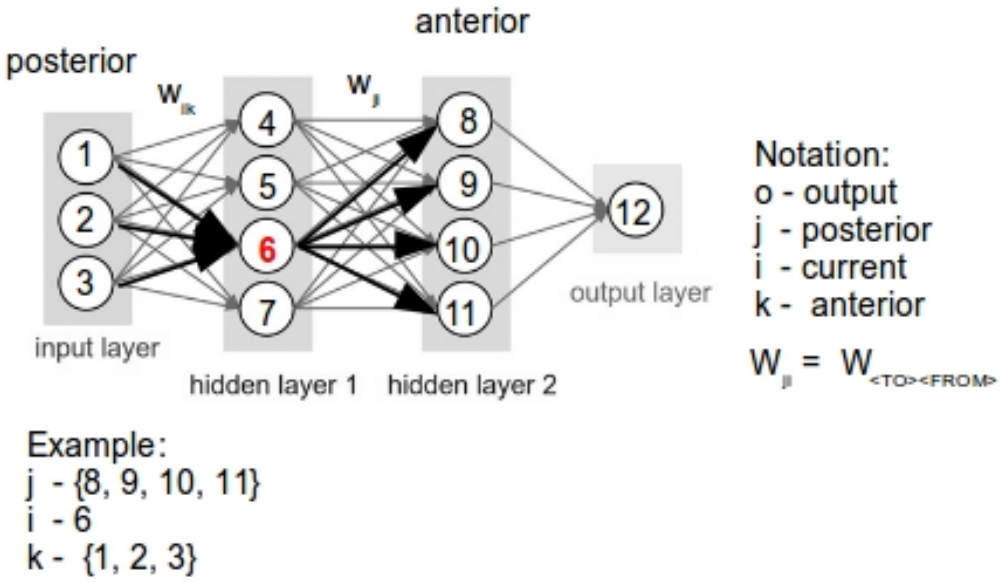
3.4 Learning in neural networks:

Error Backpropagation in Multi-Layer Neural Networks

We have shown a method to train weights for a single neuron in chapter 3.2 which uses the desired output for weight updates. This method does not directly work in neuronal networks, as we don’t know what neurons in the hidden layers should do. We do know the network’s desired final output (i.e. the training data output value); but it is completely unclear what neurons inside the network should compute to finally reach that output. Therefore we cannot use the learning rule from 3.2, but instead backward propagate the learning error from output towards input, depending on activity of neurons. This idea is called “error backpropagation algorithm”.

The **error backpropagation algorithm** was originally introduced in the 1970s, but its importance wasn't fully appreciated until a famous paper published 1986 by David Rumelhart, Geoffrey Hinton, and Ronald Williams. The backpropagation algorithm searches the **minimum of the error function** in **weight space**, using **gradient descent**. The particular combination of weights which minimizes the error function is considered to be the solution for learning a representation of data. Since this method requires computation of the gradient of the error function at each iteration step, we must **guarantee continuity and differentiability** of the error function.

The goal of backpropagation is to compute the partial derivatives of the cost function with respect to any weights in the network. In order to introduce the formalism of backpropagation we introduce the following **notations**:



1. Definitions

(1) The error signal for a certain unit i at training time *t* is given by:

$$\delta_i(t) = \frac{-\partial E_i(t)}{\partial net_i(t)}$$

where the net input to neuron *i* is

$$net_i(t) = \sum_{k \in A} w_{ik}(t) \cdot out_k(t)$$

(2) The weight change for weight w_{ik} is given by

$$\Delta w_{ik}(t) = \frac{-\partial E_i(t)}{\partial w_{ik}(t)}$$

2. Understanding the gradient for weight change

Starting from the weight change at the neuron level we can infer the representation of the update in terms of the output of the previous layer and the error at the current neuron:

$$\Delta w_{ik}(t) = \frac{-\partial E_i(t)}{\partial w_{ik}(t)} = \frac{-\partial E_i(t)}{\partial net_i(t)} \cdot \frac{\partial net_i(t)}{\partial w_{ik}(t)} = \delta_i(t) out_k(t)$$

with the error signal for node *i* computed as

$\frac{-\partial E_i(t)}{\partial net_i(t)} = \delta_i(t)$ (by definition (1))

$\frac{\partial net_i(t)}{\partial w_{ik}(t)} = \frac{\partial \sum_{l \in Ai} w_{il}(t) \cdot out_l(t)}{\partial w_{ik}(t)} = out_k(t)$ (non-zero only for l=k)

3. Forward activation of the network

In this phase the "input" is applied to the bottom layer, and we compute all neurons' outputs layer by layer towards the target output:

$$out_i(t) = f_i\big(net_i(t)\big) = f_i\left(\sum_{k \in A} w_{ik}(t) \cdot out_k(t)\right)$$

4. Calculating the error of the output neuron

For the final output the dataset contains a desired value, $target_o$, hence we can compute the error signal at the network output, out_o, (similar to chapter 3.2):

$$\delta_o(t) = \frac{-\partial E_o(t)}{\partial net_o(t)} = 2 \cdot (target_o - out_o)$$

Note that for simplicity we assume a linear output unit (without loss of generality).

5. Propagating the error back through the network

After computing the error at the network output we propagate the error signal back through the network (hence the name of the learning mechanism).

$$\delta_i(t) = \frac{-\partial E_i(t)}{\partial net_i(t)}$$

Applying the Chain Rule:

$$\delta_i(t) = \frac{\partial E_i(t)}{\partial net_i(t)} = \sum_{j \in P} \frac{-\partial E_i(t)}{\partial net_j(t)} \cdot \frac{\partial net_j(t)}{\partial out_i(t)} \cdot \frac{\partial out_i(t)}{\partial net_i(t)}$$

Where

(1) $\frac{-\partial E_i(t)}{\partial net_j(t)} = \delta_j(t)$ (by definition (1))

(2) $\frac{\partial net_j(t)}{\partial out_i(t)} = \frac{\partial \sum_{m \in P} w_{jm}(t) \cdot out_m(t)}{\partial out_i(t)} = w_{ji}$ (only nonzero for m=i)

(3) $\frac{\partial out_i(t)}{\partial net_i(t)} = \frac{\partial f(net_i(t))}{\partial net_i(t)} = f'\big(net_i(t)\big)$

Combining those three equations we compute the error signal for a neuron *i* in the network:

$$\delta_i(t) = \sum_{j \in P} \delta_j(t) \cdot w_{ji}(t) \cdot f'\big(net_i(t)\big)$$

and given the activation function ***f*** is independent of the ***j**th* node, we can rewrite the error:

$$\delta_i(t) = f'\big(net_i(t)\big) \cdot \sum_{j \in P} \delta_j(t) \cdot w_{ji}(t)$$

⇨ We observe that the **error for neuron *i*** (δ_i) only depends on the known error of neurons in **"higher" levels** j of the network hierarchy (δ_j).

6. Computing the weight update for weight anterior to posterior (k → i)

Computing the weight update using the weight increment and the error signal

$\Delta w_{ik}(t) = \delta_i(t) \cdot out_k(t)$ $\quad\quad$ $\delta_i(t) = f'\big(net_i(t)\big) \cdot \sum_{j \in P} \delta_j(t) \cdot w_{ji}(t)$

The final **weight update** is

$$\Delta w_{ik}(t) = f'\big(net_i(t)\big) \cdot \sum_{j \in P} \delta_j(t) \cdot w_{ji}(t) \cdot f\big(net_k(t)\big)$$

or

$$\Delta w_{ik}(t) = f'\big(net_i(t)\big) \cdot \sum_{j \in P} \delta_j(t) \cdot w_{ji}(t) \cdot out_k(t)$$

This update rule allows training of feedforward multilayer neural networks, for tasks such as **regression** (function approximation) and **classification**.

3.5 Supervised learning: tips and tricks

The main problem in using artificial neural networks is **parameter tuning**, because there is **no definite and explicit method to select optimal values** for the network parameters. In this section we will discuss **design choices** regarding data **pre-processing**, weight **initialization**, and **error functions**.

1. Weight initialization

   A possible option is to set all the initial weights to zero (or any other constant). This is a fatal mistake, because if every neuron in the network computes the same output, they will also all learn based on identical gradients during error-backpropagation and undergo the exact same parameter updates. In other words, there is no source of asymmetry between neurons if their weights are initialized the same; they all act as they are a single (highly redundant) neuron.

   A common method to break symmetry early on is to **initialize all neurons' weights to small random numbers**. Thereby, neurons all behave uniquely, so they will compute distinct updates and develop into diverse contributors of the full network.

   Use small weights, as small weights are likely to get the net input into the steep region of the neurons' transfer functions. The gradient will initially be large, so neurons quickly differentiate.

   Tip: initialize the weight with small random numbers, e.g. -0.001…0.001

2. Input/Output normalization

   Normalization refers to normalizing each of the data dimensions so that they are all on approximately similar scale. There are two common ways of achieving this normalization. One is to divide each dimension by its standard deviation, once it has been zero-centered. Another form of this preprocessing normalizes each dimension so that the min and max along the dimension is -1 and 1 respectively.

   Tip: use a linear (final) output neuron transfer, and normalize all input data to [-1 .. +1].

3. Small weight updates

   Weight updates are controlled by a parameter, $\eta(t)$, **the learning rate**, which determines how fast the weight change will take place. At every update only a single data point is processed; hence a "full update" will match this point well but neglect others. The learning rate allows small updates towards consecutive examples, which improves overall network behavior. The learning rate should typically be very small.

   Tip: use a constant small learning rate, e.g. $\eta(t)=0.001$

4. Avoiding local minima in weight space

   In training neural networks (as in any local gradient based method) the training might get stuck in local minima, instead of finding a global minimum. A technique called **Simulated Annealing** might help to overcome local minima but regularly

perturbing the current set of weight. This perturbation (“shaking”) shall initially be large and decay with training success, so that initially the network likely “jumps” out of local minima; but later likely stays within a found solution.

Tip: use small occasional random perturbation (“shaking”) of weights to escape local minima.

5. Network size

Number of Input channels and output channels given by problem data set (this we cannot decide, but it is given by the problem to be solved).

The important **metric** for the design of neural networks are the number of **neurons**, or more precise the number of free parameters. How do we decide on what architecture to use when faced with a practical problem? How many layers? How many neurons per layer? First, note that as we increase the size and number of layers in a neural network, the **capacity of the network** increases. That is, the space of **representable functions** grows, since the neurons can collaborate to express many **different functions**. But there is no theory yet to tell the designer how many hidden units are needed to approximate any given function.

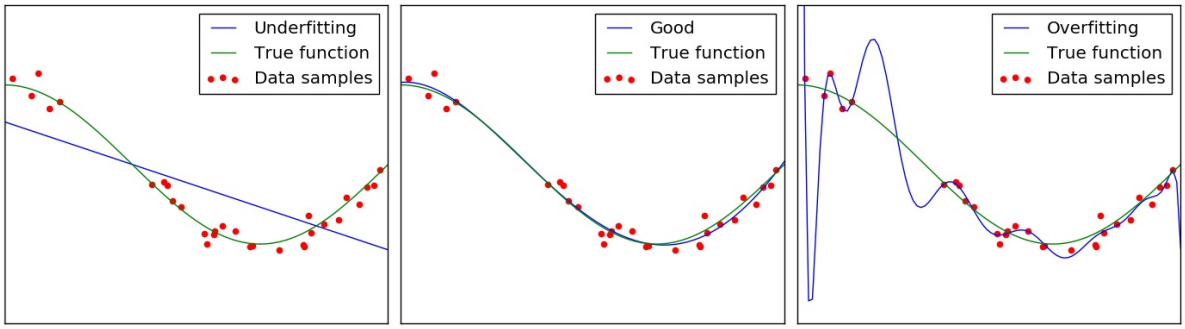
Some sources and articles offer **"rules of thumb"** for choosing size and topology of a neural network:

- "A rule of thumb is for the size of this [hidden] layer to be somewhere between the input layer size ... and the output layer size ..." (Blum, 1992, p. 60).
- "you will never require more than twice the number of hidden units as you have inputs" in an MLP with one hidden layer (Swingler, 1996, p. 53)
- "How large should the hidden layer be? One rule of thumb is that it should never be more than twice as large as the input layer." (Berry and Linoff, 1997, p. 323)

Tip: use small “fan-out” after input layer (a few more neurons than input signals; at most 2x) and slowly reduce neurons to required output size.

6. Overtraining/overfitting

The final and often most critical issue in developing a neural network is **generalization**: how well will the network make predictions for cases that are not shown in the training set? Artificial neural networks can suffer from either **underfitting** or **overfitting**.



A network that is not sufficiently complex can fail to fully detect the underlying signal in a data set, leading to underfitting. A network that is too complex may fit the noise (beyond fitting the signal), which causes overfitting. Overfitting is especially dangerous because it can easily cause terrible predictions will within the range of the training data (see how the prediction differs e.g. for -0.44 input).

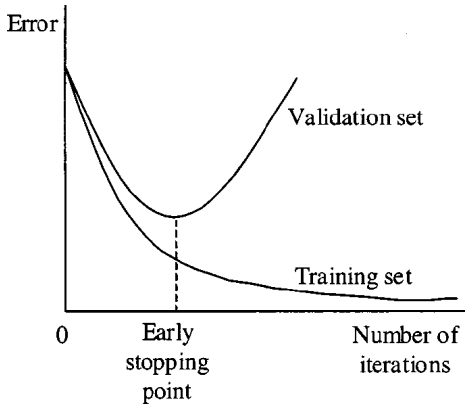
The best way to avoid overfitting is to use **large amounts of training data**. Given a fixed amount of training data, there are several approaches to avoiding underfitting and overfitting, and hence improve generalization: model selection, jittering, weight decay, Bayesian learning, combining networks, and – most commonly used - **Early stopping**.

Early stopping

While training on data, the network seems to get better and better, i.e., the error on the training set decreases. The network learns to represent every single data point as good as possible, which ultimately results in a lookup table. We would like to find the time of training when the generalization ends and learning of individual data points begins.

For this, all available data is **divided into two subsets, where 70% of all data samples are used for training, and 30% of all data samples are used for independent testing**. The first subset – the training set –is used for computing the gradient and updating the network weights and biases as before. The second subset is the testing set, which is **never used to train/update any weights**. This testing set is only used to compute remaining error of the network performance given the current training state. Note that we can compute the error, as we know input patter and desired output for the test data samples. The error on the testing set (called “validation error”) is monitored during the training process.

The validation error normally decreases during the initial phase of training, as does the training set error. However, when the network begins to overfit the data, the error on the validation set typically begins to rise. At this point in time the network achieves best generalization abilities and learning needs to stop (“early stopping”). Any further training will only lead to fitting individual data-point (see overfitting above), which reduces generalization abilities.



Contents

Lectures

4. Unsupervised neural computation

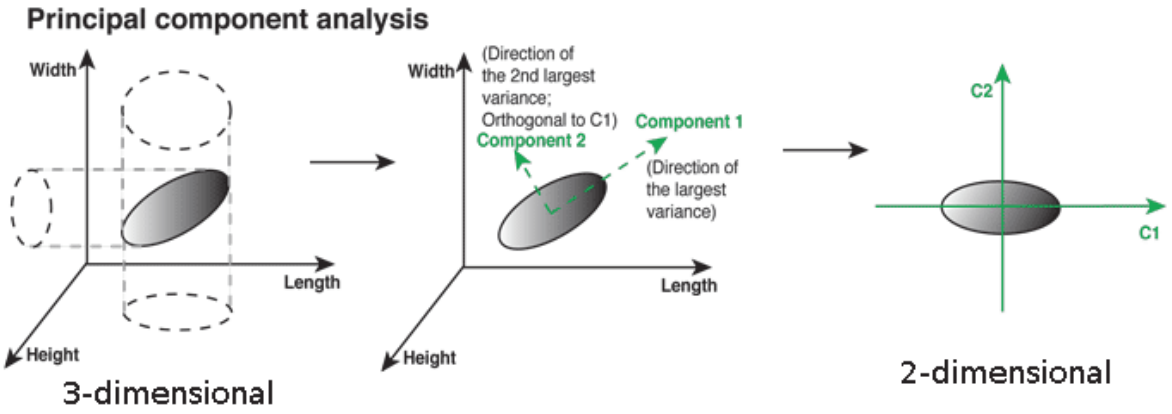
4.1. Introduction to unsupervised learning

Just as there are different ways in which we ourselves learn from our own surrounding environments, so it is with neural networks. In a broad sense, we may categorize the learning processes through which neural networks function as follows: **learning with a teacher** and **learning without a teacher.** These different forms of learning as performed on neural networks parallel those of human learning.

Learning with a teacher is also referred to as **supervised learning**. In conceptual terms, we may think of the teacher as having knowledge of the environment, with that knowledge being represented by a set of input - output examples.

Unsupervised learning does not require target vectors for the outputs. Without input-output training pairs as external teachers, unsupervised learning is **self-organized** to produce consistent output vectors by **modifying weights**. That is to say, there are **no labelled examples** of the function to be learned by the network.

For a specific task-independent measure, once the **network** has become tuned to the **statistical regularities** of the input data, the network develops the ability to **discover internal structure** for **encoding features** of the input or **compress the input data,** and thereby to **create new classes** automatically.



In many problems, such as **data compression** or **dimensionality reduction**, the measured data vectors are **high-dimensional** but we may have reason to believe that the data lie near a **lower-dimensional manifold**. Learning a **suitable low-dimensional manifold from high-dimensional data** is essentially the same as learning this **underlying source**. **Dimensionality reduction** can also be seen as the process of **deriving a set of degrees of freedom** which can be used to **reproduce** most of the **variability** of a data set.

Principal components analysis (PCA) is a classical method that provides a sequence of best **linear approximations to a given high-dimensional observation**. It is one of the most popular techniques for dimensionality reduction. However, its

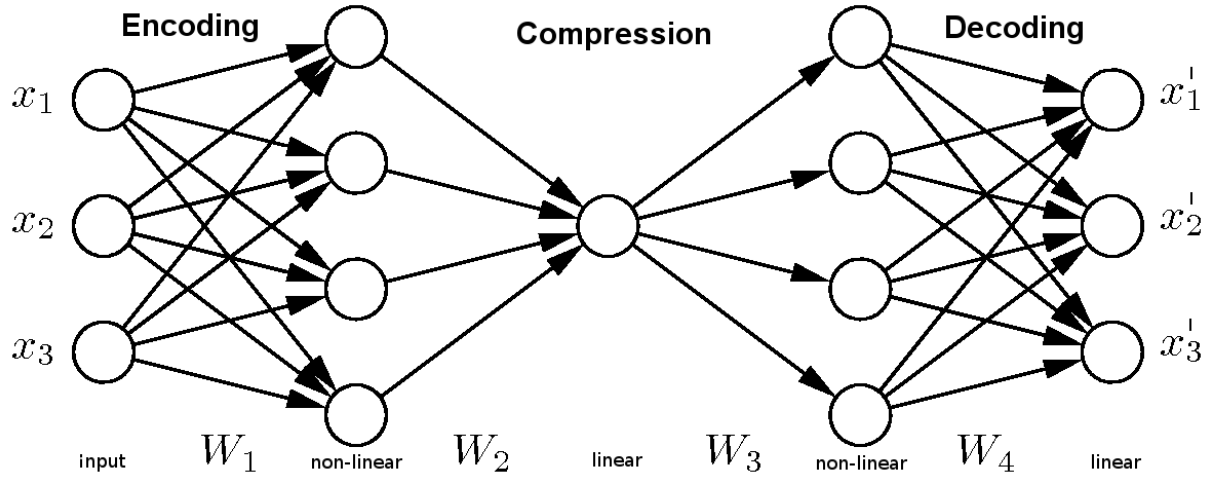
effectiveness is limited by its global linearity. Unfortunately, in dealing with large datasets, PCA can become unmanageable in computational terms.

Is there a way to overcome this computational limitation? Can PCA be realized in a neural network using an unsupervised learning algorithm?

Nonlinear principal component analysis (NLPCA) is commonly seen as a nonlinear generalization of standard principal component analysis (PCA). It generalizes the principal components from straight lines to curves (nonlinear).

Thus, the subspace in the original data space which is described by all **nonlinear components** is also curved. Nonlinear PCA can be achieved by using a **neural network** with an **auto-associative architecture** also known as autoencoder.

Such **auto-associative neural network** is a **multi-layer perceptron** that performs an identity mapping, meaning that the output of the network is required to be identical to the input. However, in the middle of the network is a layer that works as a bottleneck in which a **reduction of the dimension of the data** is enforced. This bottleneck-layer provides the desired component values.



In such a network the inputs (i.e. x_1, x_2, x_3) are identical to the desired outputs (i.e. x_1', x_2', x_3'). The network implements a "**mapping to itself**". In such a structure if the number of hidden units is small the network needs to find and **"efficient" representation**.

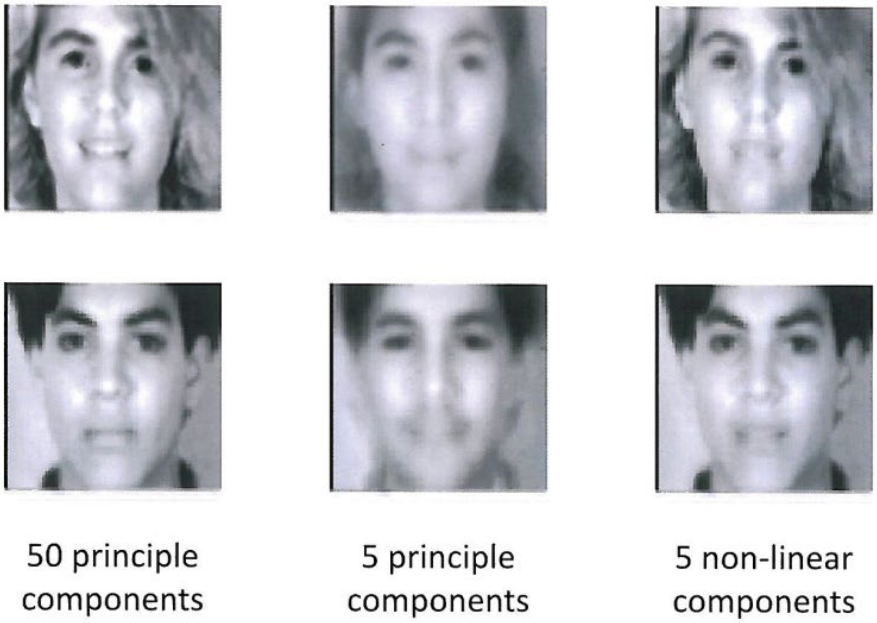
If the neuron in the **center of the network is constrained** to a **linear transfer function** it will find the **direction of the first principle component** (i.e. the direction in which the **data shows the largest variance)**.

Such neural networks can be used in information processing fields such as **pattern recognition** and **data compression,** such as face recognition or speech recognition. For example, consider a set of images produced by the rotation of a face through different angles. Clearly only one degree of freedom is being altered, and thus the images lie along a continuous one dimensional curve through image space.

In another example, for facial recognition, it has been proven that using a locally linear algorithm for nonlinear dimension reduction in an auto-associative network, one can get more precise recognition.

Example Faces

Non-linear Dimension Reduction, Nanda Kambhatla and Todd K. Leen, NIPS 1993



50 principle components | 5 principle components | 5 non-linear components

120 faces in total
original image is 64x64 pixels of 8bit/pixel gray level

4.2. Radial Basis Functions

In solving a nonlinearly separable pattern-classification problem, there is usually practical benefit to be gained by mapping the input space into a new space of high enough dimension. Basically, a nonlinear mapping is used to transform a nonlinearly separable classification problem into a linearly separable one with high probability. The Radial Basis Functions (RBF) technique consists in selecting such a mapping function, F:

$$f(x) = \sum_{i=1}^{N} w_i \, \phi(\|x - c_i\|),$$

where $\{\phi(\|x - x_i\|) | \, i = 1,2, \dots, N\}$ is the set of *N* arbitrary (generally nonlinear) functions known as radial-basis functions, c_i is the *i-th* center, and $\|\,.\,\|$ denotes a distance metric, usually the Euclidian distance. Typical choices for radial basis functions are:

- Spline functions,

$$\phi(x) = x^2 log x$$

- Gaussian functions,

$$\phi(x) = e^{\frac{-x^2}{\beta}}$$

- Multi-quadratic functions,

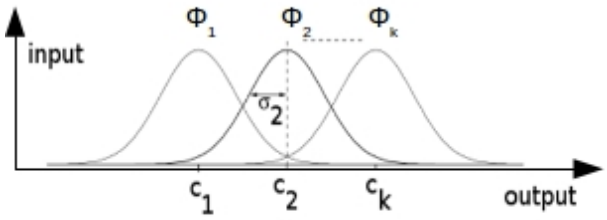
$$\phi(x) = \sqrt{(x^2 + \beta^2)}$$

- Inverse multi-quadratic function,

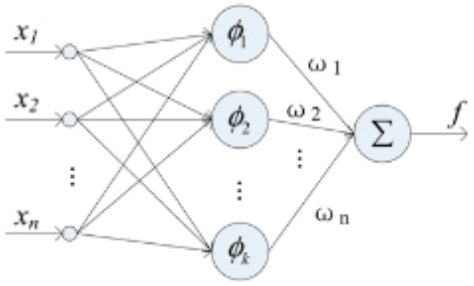
$$\phi(x) = \frac{1}{\sqrt{(x^2 + \beta^2)}}$$

It has been proved that a RBF network (RBN) can approximate well any arbitrarily continuous function if a sufficient number of radial-basis function units are given (the network structure is large enough), and the network parameters are carefully chosen. RBN also has the best approximation property in the sense of having the minimum distance from any given function under approximation.

Radial-basis functions (RBF)



Radial-Basis Functions Network (RBN)



In a simplified perspective, a RBN is basically approximating the input data through a linear combination of the *k* Gaussians Φ_k with centers c_k. The Gaussian "activation" is determined, as previously mentioned, by the Euclidian distance of an input point to the Gaussian center. This process, allows the projection of the input in a higher dimension where classification / prediction is easier.

There are two main problems related to parameterizing such learning systems: 1) determining the parameters for the RBFs (i.e. c_k , σ_k); 2) determining the weights of the network, w_k (i.e. can be performed using Backpropagation).

For the first problem, at design time, a narrow variance Gaussian per data sample is considered. Subsequently, the number of Gaussian is reduced to allow interpolation of the input space. But, how well is data represented by a single Gaussian? This

question opens a new interpretation of the Gaussian hidden units of RBNs. In a neurobiological context, the Gaussians correspond to sensory receptive fields. Such a receptive field is defined as “the region of a sensory field from which an adequate sensory stimulus will elicit a response”. In RBN terminology, the receptive field of a hidden unit is that region of the input layer of source nodes from which an adequate pattern will elicit a response.

This definition applies equally well to multilayer perceptrons (MLPs) and RBNs. Summarizing the main aspects we can analyze comparatively MLPs and RBFs:

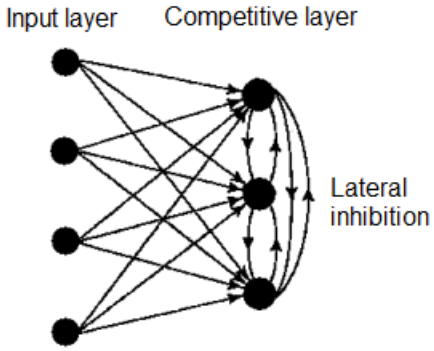
	RBFs	**MLPs**
Hidden units	$f(\|\|x - c_i\|\|)$ Decreasing with increasing distance (“localized)	$f(\sum_{i=1}^{N} w_i \ inp_i)$ Usually nonlinear, monotonically increasing
Output	Only a few active contributors	Many contributors Most “hidden” neurons are active Problems with local minima
Network topology	Simple 3-layer structure	Many structures possible dependent on problem
Training	2-stage process: 1. Finding Gaussian params 2. Training weights	All weights are simultaneously adapted through backpropagation

4.3. Vector Quantization

In neurobiology, during neural growth, synapses are strengthened or weakened, in a process usually modelled as a competition for resources. In such a learning process, there is a **competition** between the neurons to fire. More precisely, neurons compete with each other (in accordance with a learning rule) for the “opportunity” to respond to features contained in the input data. In its simplest form, such behaviour describes a “**winner-takes-all**” strategy. In such a strategy, the neuron with the greatest total input “wins” the competition and turns on; all the other neurons in the network then switch off. The aim of such learning mechanisms is to **cluster the data**.

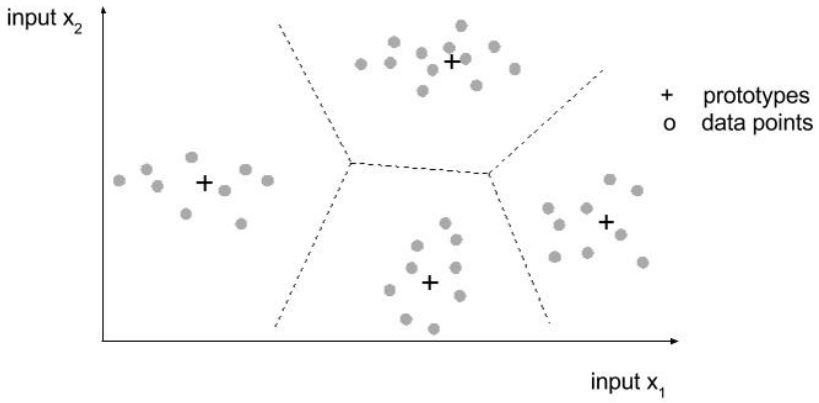
In a typical scenario, such behavior can be implemented with a neural network that consists of **two layers**—an input layer and a **competitive layer with lateral**

inhibition. The input layer receives the available data. The competitive layer consists of neurons that compete with each other.

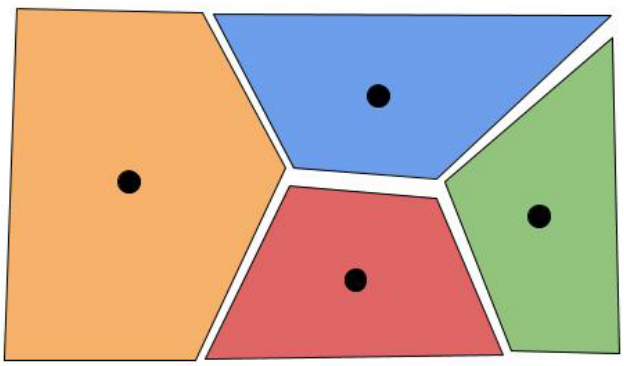


The basic mechanism of competitive learning is to find a **winning unit** and **update its weights** to make it more likely to win in future if a similar input will be given to the network.

Vector quantization (VQ) is a form of competitive learning. Such an algorithm is able to discover structure in the input data. Generally speaking, vector quantization is a form of **lossy data compression**—lossy in the sense that some information contained in the input data is lost as a result of the compression.



An input data point belongs to a certain class if its position (in the 2D space) is closest to the class prototype, fulfilling the **Voronoi partitioning** (i.e. partitioning of a plane into regions based on distance to points in a specific subset of the plane.



Algorithm:

#1 Choose the number of clusters, *M*

#2 Initialize the prototypes $w_1, w_2, \ldots, w_n$ (hint: pick random input samples but distributed evenly in the input space)

#3 Repeat until "good enough"

#4 Randomly pick an input *x*

#5 Determine the winning prototype node *k* such that

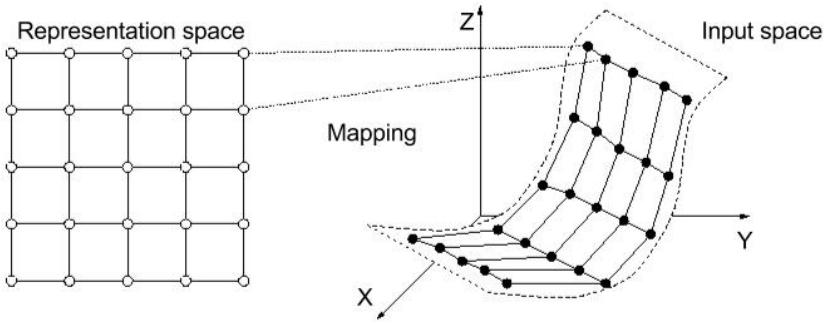
$$|w_k - x| \leq |w_i - x| \text{ for all nodes } i$$

#6 Update the winning prototype weights

$w_k(t+1) = w_k(t) + \eta(x - w_k(t))$, where η is the learning rate.

4.4. Kohonen's Self-Organizing-Maps

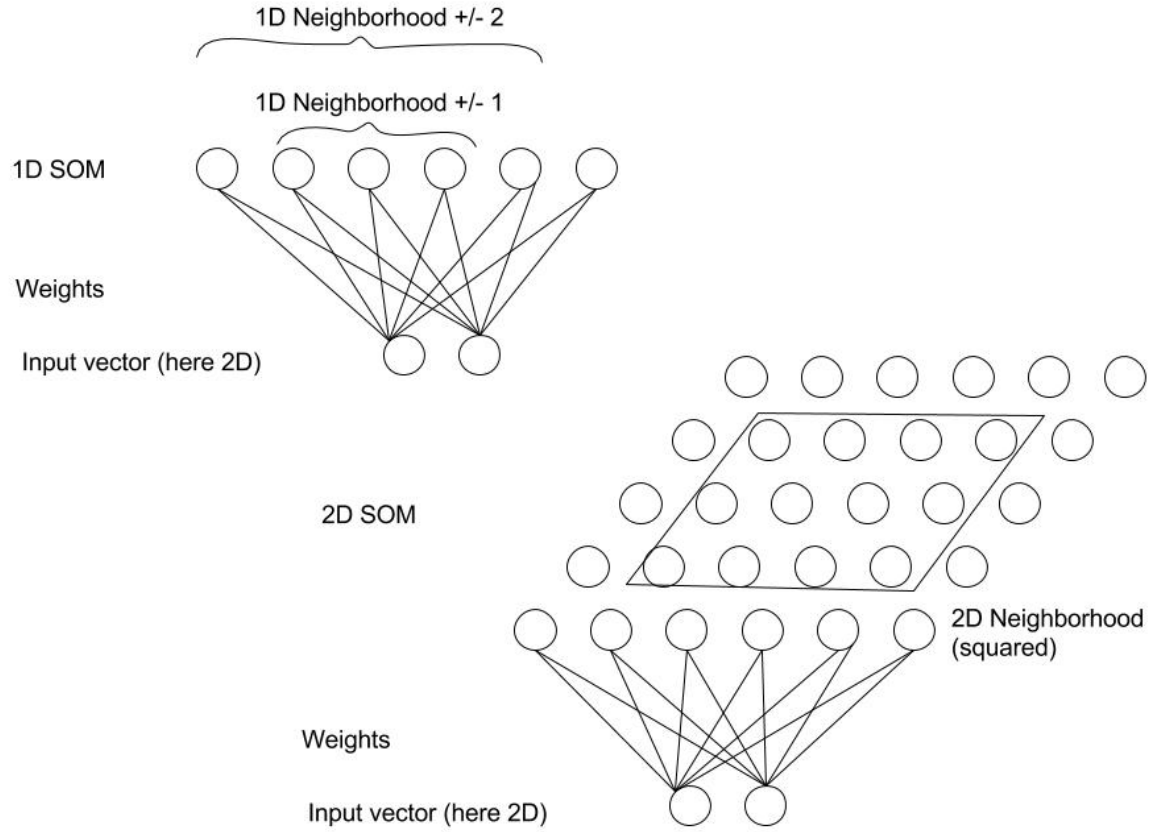
Kohonen's self-organizing map (**SOM**) is one of the most **popular unsupervised neural network** models. Developed for an associative memory model, it is an unsupervised learning algorithm with a simple structure and computational form, and is motivated by the **retina-cortex mapping**. The SOM can provide **topologically preserved mapping** from input to output spaces, such that "nearby" sensory stimuli are represented in "nearby" regions.



External stimuli are received by various sensors or **receptive fields** (for example, visual-, auditory-, motor-, or somato-sensory), coded or abstracted by the living neural networks, and projected through axons onto the cerebral cortex, often to distinct parts of the cortex. In other words, the different areas of the cortex (cortical maps) often correspond to different sensory inputs, though some brain functions require collective responses.

These networks are based on **competitive learning**; the output neurons of the network compete among themselves to be activated or fired, with the result that only one output neuron, or one neuron per group, is on at any one time. An output neuron that wins the competition is called **a winner-takes-all** neuron, or simply best matching unit.

In a self-organizing map, the neurons are placed at the nodes of a **lattice** that is usually **one** or **two dimensional**.



The neurons become **selectively tuned** to various input patterns (stimuli) or classes of input patterns in the course of **the competitive learning** process. The locations of the neurons so tuned (i.e., the winning neurons) become ordered with respect to each other in such a way that a meaningful coordinate system for different input features is created over the lattice. A self-organizing map is therefore characterized by the formation of a **topographic map** of the input patterns, in which the spatial locations (i.e., coordinates) of the neurons in the lattice are indicative of intrinsic **statistical features** contained in the input patterns.

The SOM is an optimal VQ when the **neighbourhood** eventually shrinks to just the winner, as it will satisfy the two necessary **conditions for VQ** (Voronoi partition and centroid condition). The use of the neighbourhood function makes the SOM superior to common VQs in two main respects. Firstly, the SOM is better at overcoming the under- or over-utilization and local minima problem. The second is that the SOM will produce a map with some ordering among the code vectors, and this gives the map an ability to **tolerate noise** in the input or **retrieval patterns**.

Finally, once the network has become tuned to the **statistical regularities** of the input data, the **network develops the ability to form internal representations** for encoding features of the input and thereby to create new classes automatically.

Algorithm:

#1 Initialize all weights w_{ij} and define the neighborhood function $\phi(i, j)$

#2 Select input *x* and determine the winning unit *i* such that

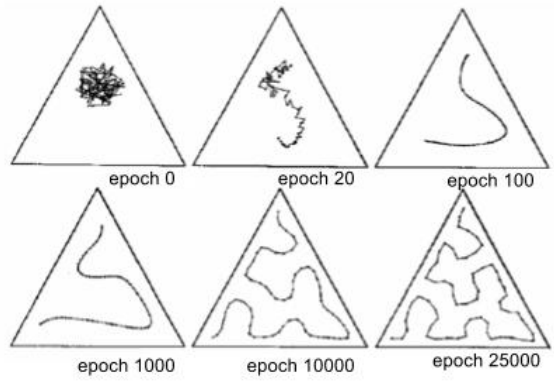
$$|x - w_k| \leq |x - w_j| \text{ for all nodes } j \neq i$$

#3 Update weights for all units *j* given the winner unit *i*

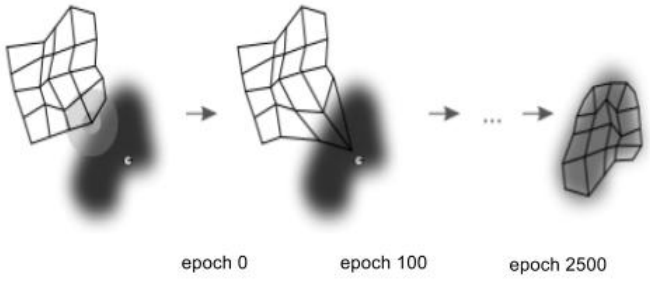
$w_j(t+1) = w_j(t) + \eta\phi(j, i)(x - w_j(t))$, where η is the learning rate

#4 Repeat from step #2 until convergence is reached and update η and ϕ

In a practical example such a 1D SOM network is able to cover uniformly a 2D input space preserving the topology. This behavior holds also for higher dimensional output spaces.



1D SOM Network training



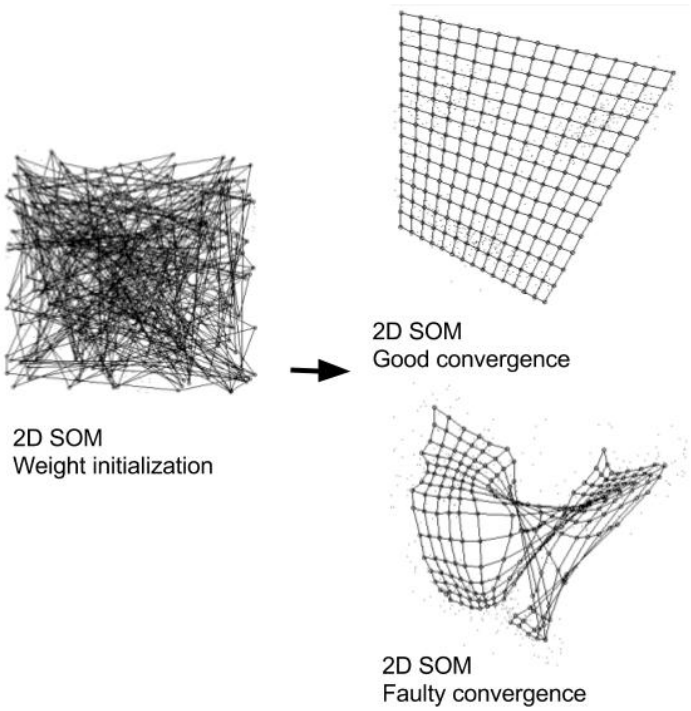
2D SOM Network training

Tips and tricks:

- Choosing a neighborhood function ϕ
  - Typically functions which decrease influence with distance (e.g. concentric squares, hexagons, and other polygonal shapes as well as Gaussian functions)
  - Neighborhood has to be large at the beginning (e.g. initial radius initialized as half size of the net) and decrease in time to 1 (convergence)
- Learning rate η
  - Function that decreases in time (e.g. inverse time, exponential, linear)
- Number of training steps
  - For good accuracy the number of learning steps has to be high enough, e.g. 500 times the number of SOM neurons

Typical problems:

- Mapping into 2D – weights in the middle of the 2D lattice do not get updated similarly to the ones at the extremities and a “knot” appears



- Mapping into a too-low dimension (e.g. 3D cube into 2D – a dimension is lost)

The term self-organizing map signifies a class of mappings defined by error-theoretic considerations. In practice they result in certain unsupervised, competitive learning processes, computed by simple-looking SOM algorithms. Many industries have found the SOM-based software tools useful. The most important property of the SOM, orderliness of the input-output mapping, can be utilized for many tasks: reduction of the amount of training data, speeding up learning nonlinear interpolation and extrapolation, generalization, and effective compression of information for its transmission.

4.5. Hopfield Networks

Biological substrate

Donald Hebb hypothesized in 1949 how neurons are connected with each other in the brain: “When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A’s efficiency, as one of the cells firing B, is increased.”, and postulated a new learning mechanism, **Hebbian learning**. In other words **neural networks stores and retrieves associations**, which are learned as synaptic connection. In **Hebbian learning**, both **presynaptic** and **postsynaptic**

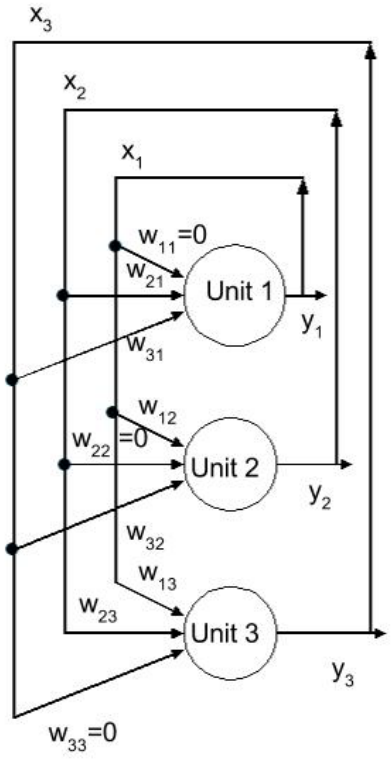
neurons are involved. Human memory thus works in an **associative** or **content-addressable** way.

Technical perspective

The basic Hopfield network consists of a set of neurons and a corresponding set of **unit-time delays**, forming a **multiple-loop feedback** system. The model is a **recurrent neural network** with fully interconnected neurons. The number of **feedback** loops is equal to the number of neurons. Basically, the output of each neuron is fed back, via a **unit-time delay** element, to each of the other neurons in the network. Such a structure allows the network to **recognise** any of the **learned patterns** by exposure to only partial or even some **corrupted information** about that pattern, i.e., it eventually settles down and returns the closest pattern or the best guess.

In the Hopfield model it is assumed that the individual units preserve their individual states until they are selected for a new update. The selection is made randomly. A Hopfield network consists of n totally coupled units, that is, each unit is connected to all other units except itself. The network is symmetric because the weight w_{ij} for the connection between unit *i* and unit *j* is equal to the weight w_{ji} of the connection from unit *j* to unit *i*. This can be interpreted as meaning that there is a single bidirectional connection between both units. The absence of a connection from each unit to itself avoids a permanent feedback of its own state value. Below three neurons *i = 1, 2, 3* with values $x_i = \pm 1$ have connectivity w_{ij}; any update has input xi and output y_i:

.



Update rule:

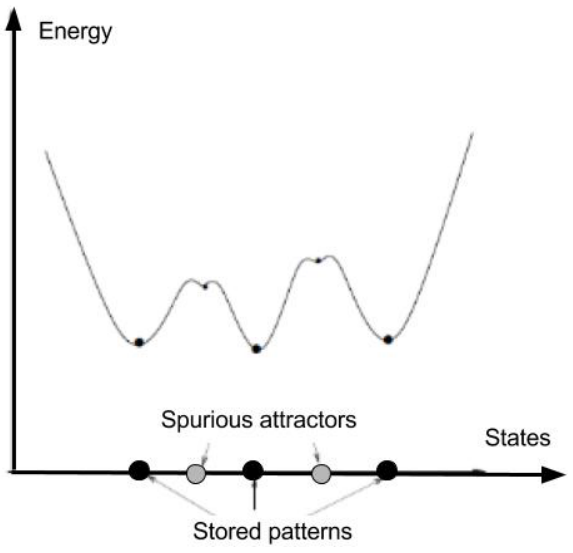
Assume N neurons $= 1, \cdots, N$ with values $x_i = \pm 1$
The update rule is for the node i is given by:

$$\text{If } h_i \geq 0 \text{ then } 1 \leftarrow x_i \text{ otherwise } -1 \leftarrow x_i$$

where $h_i = \sum_{j=1}^{N} w_{ij} x_j + b_i$ is called the **field** at i, with $b_i \in \mathbb{R}$ a bias.
Thus, $x_i \leftarrow \text{sgn}(h_i)$, where $\text{sgn}(r) = 1$ if $r \geq 0$, and $\text{sgn}(r) = -1$ if $r < 0$.
We put $b_i = 0$ for simplicity as it makes no difference to training the network with random patterns.
We therefore assume $h_i = \sum_{j=1}^{N} w_{ij} x_j$.
There are now two ways to update the nodes:
Asynchronously: At each point in time, update one node chosen randomly or according to some rule.
Synchronously: Every time, update all nodes together.

The Hopfield network may be operated in a **continuous** mode or a **discrete** mode, depending on the used **neuron model**.

In the application of the Hopfield network as **a content-addressable memory**, we know a priori the **fixed points (attractors)** of the network in that they correspond to the patterns to be stored. However, the synaptic weights of the network that produce the desired fixed points are unknown, and the problem is how to determine them. The primary function of a content-addressable memory is to retrieve a pattern (item) stored in memory in response to the presentation of an incomplete or noisy version of that pattern.



One way in which such properties may be used to implement a computational task is by way of the concept of **energy minimization**. **Hopfield networks** are an example of such an approach. Hopfield networks have an **energy function** which decreases or is unchanged with asynchronous updating. For a given state $x \in \{-1, 1\}^N$ of the network and for any set of connection weights w_{ij} with $w_{ij} = w_{ji}$ and $w_{ii} = 0$, let

$$E = -\frac{1}{2}\sum_{i,j=1}^{N} w_{ij} x_i y_i$$

Practical aspects

- How many random patterns can we store in a Hopfield network with N nodes? In other words, given N, what is an upper bound for p, the number of stored patterns, such that the crosstalk term remains small enough with high probability?
    - A long and sophisticated analysis of the stochastic Hopfield network shows that if p/N > 0.138, small errors can pile up in updating and the memory becomes useless.
- For small enough p, the stored patterns become attractors of the dynamical system given by the synchronous updating rule. However, we also have other, so-called spurious states.
    - If we start at a state close to any of these spurious attractors then we will converge to them. However, they will have a small basin of attraction.

Contents

Lectures

5. Deep Neural Learning

Formally speaking, **Deep Learning** allows **computational models** that are composed of **multiple processing layers** to **learn representations** of **data** with multiple levels of abstraction. These methods have dramatically improved the state-of-the-art in **speech recognition**, **visual object recognition**, **object detection** and many other domains such as drug discovery and genomics. **Deep Learning discovers** intricate **structure** in large data sets by using the **backpropagation** algorithm to indicate how a machine should change its internal parameters that are used to compute the **representation in each layer** from the representation in the previous layer. **Deep convolutional nets** have brought about **breakthroughs** in **processing images**, **video**, **speech** and **audio**, whereas **recurrent nets** have shone light on sequential data such as **text** and **speech**.

Deep learning solves the central problem in representation learning by introducing **representations** that are expressed in terms of other, simpler representations. Deep learning enables the computer to build **complex concepts** out of **simpler concepts**. Figure 1 shows how a deep learning system can **represent the concept** of an **image** of a person by combining **simpler concepts**, such as **corners** and **contours**, which are in turn defined in terms of edges.

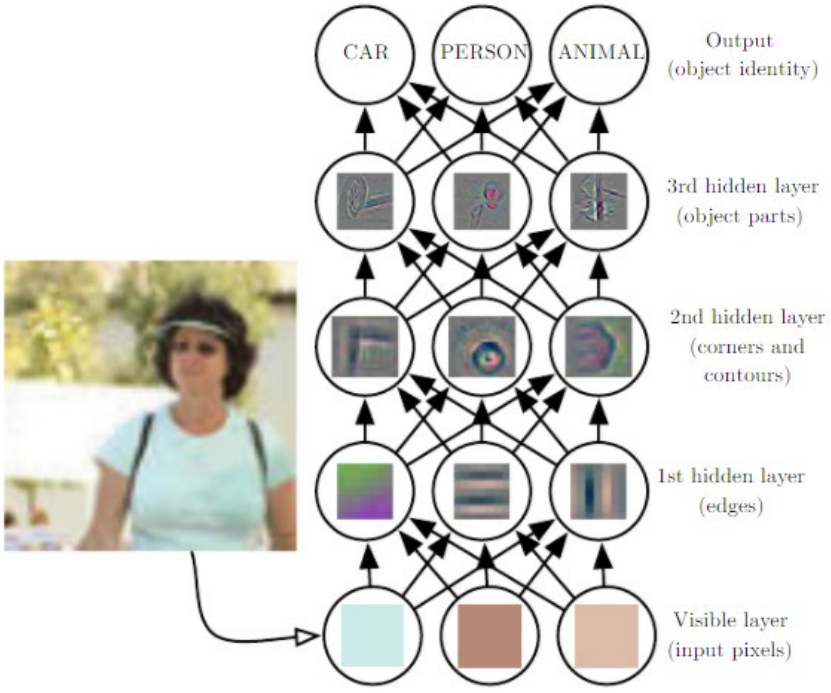


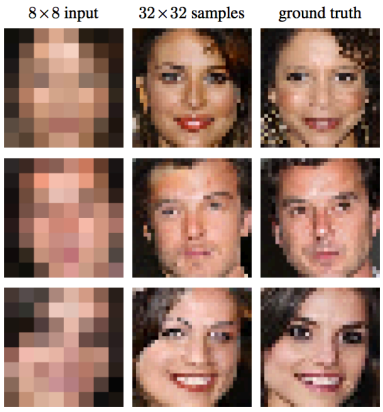
Figure 1 A Deep Learning model for image recognition

The quintessential example of a deep learning model is the **feedforward deep network**, or **multilayer perceptron (MLP)**, that we already studied in the previous chapters. A multilayer perceptron is just a **mathematical function mapping** some set of **input values** to **output values**. The function is formed by composing many simpler functions. We can think of each application of a different **mathematical function** as providing a **new representation of the input**. The idea of **learning the right representation** for the data provides one perspective

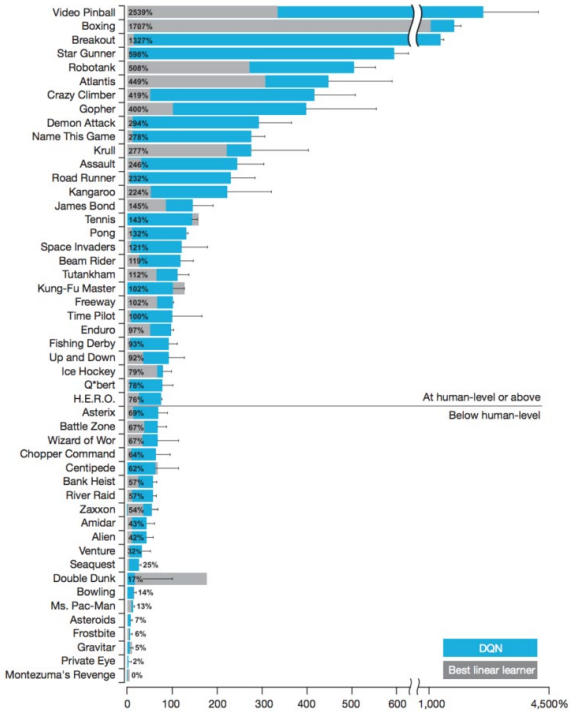
on deep learning. Another perspective on deep learning is that **depth enables learning a multistep** computer **program**.

Over the last few years Deep Learning was applied to hundreds of problems, ranging from computer vision to natural language processing. In many cases Deep Learning outperformed previous work. Deep Learning is heavily used in both academia to study intelligence and in the industry in building intelligent systems to assist humans in various tasks:

- Computer vision and pattern recognition
    - Restore colors in B&W photos and videos
    - **Pixel restoration**
    - Real-time multi-person pose estimation
    - Describing photos
    - Changing gazes of people in photos
    - Real-time analysis of behaviors
    - Iterating photos to create new objects
    - Translation



- Computer games, robots & self-driving cars
    - Winning Atari Breakout
    - **Beating people in computer games**
    - Self-driving cars
    - Robotics



- Sound
    - Voice generation
    - Music composition
    - Restoring sound in videos
- Art
    - Transferring style from famous paintings
    - Automatically writing Wikipedia articles, math papers, computer code and even Shakespeare
    - Handwriting
- Computer hallucinations, predictions and other wild things
    - Predicting demographics and election results
    - Deep dreaming
    - AI invents and hacks its own crypto to avoid eavesdropping
    - Deep Learning networks creating Deep Learning networks
    - Predicting earthquakes

5.1 Fundamentals of Deep Networks

Revisiting the definition of deep learning, the facets that differentiate deep learning networks in general from “canonical” feed-forward multilayer networks are as follows:

- **More neurons** than previous networks
- More **complex ways of connecting layers**
- **Explosion** of **computing power** to train
- **Automatic feature extraction**

To further provide color to our definition of deep learning, here we define **the four major architectures of deep networks**:

- **Unsupervised Pre-trained** Networks
- **Convolutional** Neural Networks
- **Recurrent** Neural Networks
- **Recursive** Neural Networks

Starting from a basic MLP model, a **deep learning model** assumes **a high neuron count** that has risen over the years to express more complex models. **Layers** also have **evolved** from each layer being **fully connected in multilayer networks** to **locally connected patches** of neurons between layers in **Convolutional Neural Networks (CNNs)** and recurrent connections to the same neuron in **Recurrent Neural Networks** (in addition to the connections from the previous layer).

More connections means that the neural networks have **more parameters to optimize,** and this required the explosion in computing power that occurred over the past decades. All of these advances provided the foundation to build next-generation neural networks capable of **extracting features** for themselves in a **more intelligent** fashion. This allowed deep networks to **model more complex problem spaces** (e.g., image recognition advances) than previously possible, as we have seen in the multitude of previous examples. As industry demands are ever changing and ever reaching, the capabilities of neural networks have had to charge forward.

5.2 Common Architectural Principles of Deep Networks

Before we get into the specific architectures of the major deep networks, in the next section, we extend our understanding of the **core components**. First, we’ll reexamine the core components again as follows and extend their coverage for the purposes of **understanding deep networks**.

Parameters

As we also learned in previous chapters, parameters relate to the x parameter vector in the equation Ax = b in basic machine learning. **Parameters** in neural networks relate directly to the **weights on the connections** in the network. We take the dot product of the matrix A, and the parameter vector x to get our current output column vector b. The closer our outcome vector b is to the actual values in the training data, the better our model is. We use **methods of optimization** such as **gradient descent** to find good values for the parameter vector to minimize loss across our training dataset.

In **deep networks**, we still have a parameter vector representing the **connection** in the network model we're trying to **optimize**. The **biggest change** in **deep networks** with respect to parameters is **how the layers are connected** in the different architectures.

Layers

Layers are a **fundamental architectural unit** in deep networks. One can **customize a layer** by changing the type of **activation function** it uses (or subnetwork type). Moreover, one can use **combinations of layers** to achieve a goal (e.g., classification or regression). Finally, it is important to note that each type of layer requires different **hyperparameters** (specific to the architecture) to get a deep network to learn initially. Further **hyperparameter tuning** can then be beneficial through **reducing overfitting**.

Activation Functions

In deep network **activation functions** are used in specific architectures to **drive feature extraction**. The **higher-order features** learnt from the data in deep networks are a **nonlinear transform** applied to the output of the previous layer. This allows the network to **learn patterns** in the data within a **constrained space**.

Depending on the activation function one picks, one will find that some **objective functions** are more appropriate for different kinds of data (e.g., dense versus sparse).

Hidden layers are concerned with extracting progressively higher-order features from the raw data. Commonly used functions include: Sigmoid, Tanh, Hard tanh, Rectified Linear Unit (ReLU) (and its variants).

Output layers for regression are motivated by what type of answer we expect our model to output. If we want to output a single real-valued number from our model, we'll want to use a **linear** activation function.

Output layer for binary classification need a **sigmoid** output layer with a single neuron to return a real value in the range of 0.0 to 1.0 (excluding those values) for the single class. This real-valued output is typically interpreted as a **probability distribution**.

Output layer for multiclass classification one cares about the best score across these classes. It typically uses a **softmax** output layer with an **argmax** function to get the highest score of all the classes. The **softmax** output layer computes a **probability distribution** over all the classes.

Loss Functions

Loss functions quantify the **agreement** between the **predicted** output (or label) and the **ground truth** output. We use loss functions to **determine the penalty** for an **incorrect classification** of an input vector. Typically, when designing deep neural nets one can use one of the following loss functions: **Squared** loss, **Logistic** loss, **Hinge** loss, **Negative log** likelihood.

Optimization Algorithms

Training a model in deep learning involves **finding the best set** of values for the **parameter** vector of the model. One can think of deep learning as an **optimization** problem in which one **minimizes the loss function** with **respect** to the **parameters** of our **prediction** function (based on the model).

First-order optimization algorithms calculate the **Jacobian matrix**. The Jacobian has one partial derivative per parameter (to calculate partial derivatives, all other variables are momentarily treated as constants). The algorithm then takes one step in the direction specified by the Jacobian. **Second-order algorithms** calculate the **derivative of the Jacobian** (i.e., the derivative of a matrix of derivatives) by approximating the **Hessian**. Second-order methods take into account **interdependencies between parameters** when choosing how much to modify each parameter.

Gradient descent is a member of this **path-finding class of algorithms**. Variations of gradient descent exist, but at its core, it finds the **next step** in the right **direction** with respect to an **objective** at each iteration. Those steps **move** us toward a **global minimum** error or **maximum likelihood**.

Stochastic gradient descent (SGD) is machine learning's workhorse optimization algorithm. **SGD** trains **several orders of magnitude faster** than methods such as batch gradient decent, with no loss of model accuracy. The **strengths** of SGD are **easy implementation** and the **quick processing** of large datasets. You can **adjust SGD** by adapting the **learning rate** (e.g., Adagrad) or **using second-order information** (i.e., the Hessian). SGD is also a popular algorithm for training neural networks due to its **robustness** in the face of **noisy updates**, building **models** that **generalize well**.

Hyperparameters

A **hyperparameter** is any **configuration setting** of a deep net that is free to be chosen by the user that might **affect performance**.

Here we have parameters such as:

- **Layer size**
  - The **number of neurons in a given layer**. For the **input layer**, this will match up to the **number** of **features** in the **input vector**. For the output **layer**, this will either be a **single output neuron** or a number of neurons matching the number of classes we are trying to predict.
- **Magnitude** (**momentum**, **learning rate**)
  - Hyperparameters in the magnitude group involve the gradient, step size, and momentum. The **learning rate** in machine learning is **how fast we change the parameter vector** as we move through search space. If the learning rate becomes too high, we can move toward our goal faster but we might also take a step so large that we shoot right past the best answer to the problem, as well.
  - **Momentum** is a factor between 0.0 and 1.0 that is **applied** to the **change rate of the weights over time**. Typically, we see the value for momentum between 0.9 and 0.99.
- **Regularization**
  - Regularization is a measure taken against **overfitting**. **Overfitting** occurs when a model describes the training set but **cannot generalize well** over new inputs. Overfitted models have no predictive capacity for data that they haven't seen.
  - **Regularization** for hyperparameters helps modify the gradient so that it doesn't step in directions that lead it to overfit.
- **Activations** (and activation function families)
- **Weight initialization** strategy
- **Loss functions**
- **Settings for epochs** during training (**mini-batch** size)
  - With **mini-batching** we **send more than one input vector** (a group or batch of vectors) to be trained in the learning system. This allows us to use **hardware** and **resources** more **efficiently** at the computer-architecture level.
- **Normalization** scheme for input data

5.3 Deep Networks Building Blocks

Next, we'll take the concepts in the previous section (i.e. Parameters, Layers Activation functions, Loss functions, Optimization methods, Hyper-parameters) and build on them to better understand the **building block networks of deep networks**.

Inspired by networks of biological neurons, feed-forward networks are the simplest artificial neural networks. They are composed of **an input layer**, **one or many hidden layers**, and **an output layer**. In this section, we introduce networks that are considered **building blocks** of larger deep networks, such as **Restricted Boltzmann Machines (RBMs) and Autoencoders**.

Both RBMs and autoencoders are characterized by an **extra layer-wise step** for training. They are often used for the **pretraining phase** in other larger deep networks.

RBM

RBMs model probability and are great at **feature extraction**. They are feed-forward networks in which data is **fed** through them in **one direction** with two biases rather than one bias as in traditional backpropagation feed-forward networks. RBMs are used in deep learning for **feature extraction** and **dimensionality reduction**.

"Restricted Boltzmann Machines" are networks in which **connections between nodes** of the same layer are **prohibited** (e.g., there are no visible-visible or hidden-hidden connections along which signal passes).

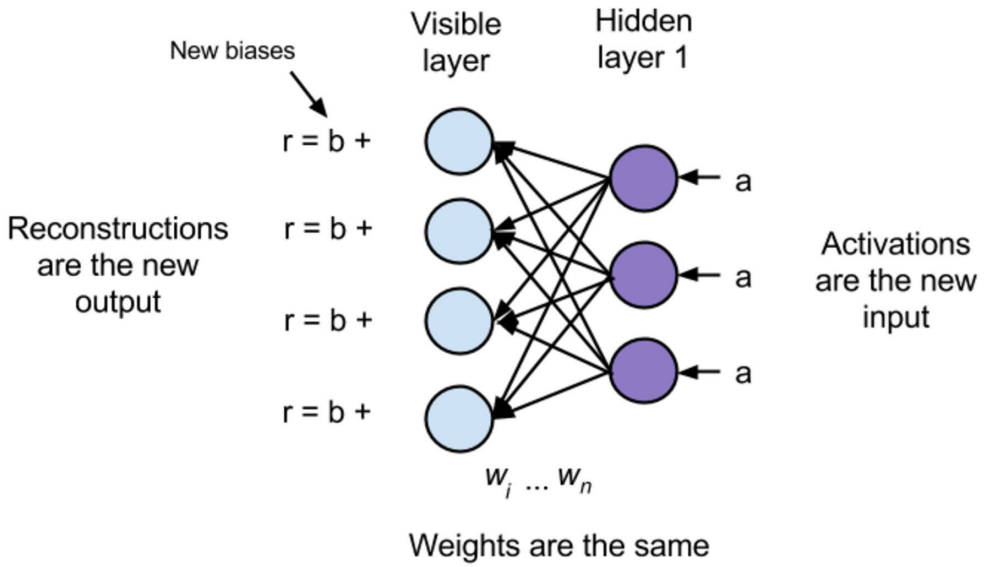


Figure 2 The Restricted Boltzmann Machine model

A standard RBM has a visible layer and a hidden layer, as shown in Figure 2. We can also see a graph of weights (connections) between the hidden and visible units in the figure. Think of these weights in the same way you think of weights in the classical neural network sense. With RBMs, every visible unit is connected to every hidden unit, yet no units from the same layer are connected. Each layer of an RBM can be imagined as a row of nodes. The nodes of the visible and hidden layers are connected by connections with associated weights.

The technique known as **pretraining** using RBMs means teaching it to **reconstruct** the **original data** from a limited sample of that data. That is, given a chin, a trained network could approximate (or "reconstruct") a face. **RBMs learn to reconstruct** the **input** dataset.

RBMs **calculate gradients** by using an algorithm called contrastive divergence. **Contrastive divergence** (**CD**) is the name of the algorithm used in sampling for the

layer-wise pretraining of a RBM. Also called CD-k, contrastive divergence **minimizes** the **Kullback-Leibler** (**KL**) **divergence** (the **delta between the real distribution of the data and the guess**) by sampling k steps of a Markov chain to compute a guess.

Autoencoders

Autoencoders are a variant of **feed-forward neural networks** that have an **extra bias** for calculating the error of **reconstructing** the **original input**. After training, **autoencoders** are then **used** as a **normal feed-forward** neural network for activations. This is an **unsupervised** form of **feature extraction** because the neural network uses only the original input for learning weights rather than backpropagation, which has labels.

We use **autoencoders** to **learn compressed representations** of datasets. Typically, we use them to **reduce a dataset's dimensionality**. The output of the autoencoder network is a **reconstruction** of the **input data** in the most **efficient form**.

Autoencoders share a strong **resemblance** with **multilayer perceptron neural networks** in that they have an **input layer**, **hidden layers** of neurons, and then an **output layer**. The **key difference** to note between a multilayer perceptron network diagram (from earlier chapters) and an autoencoder diagram is the **output layer** in an **autoencoder** has the **same number of units** as the **input layer** does, as depicted in Figure 3.

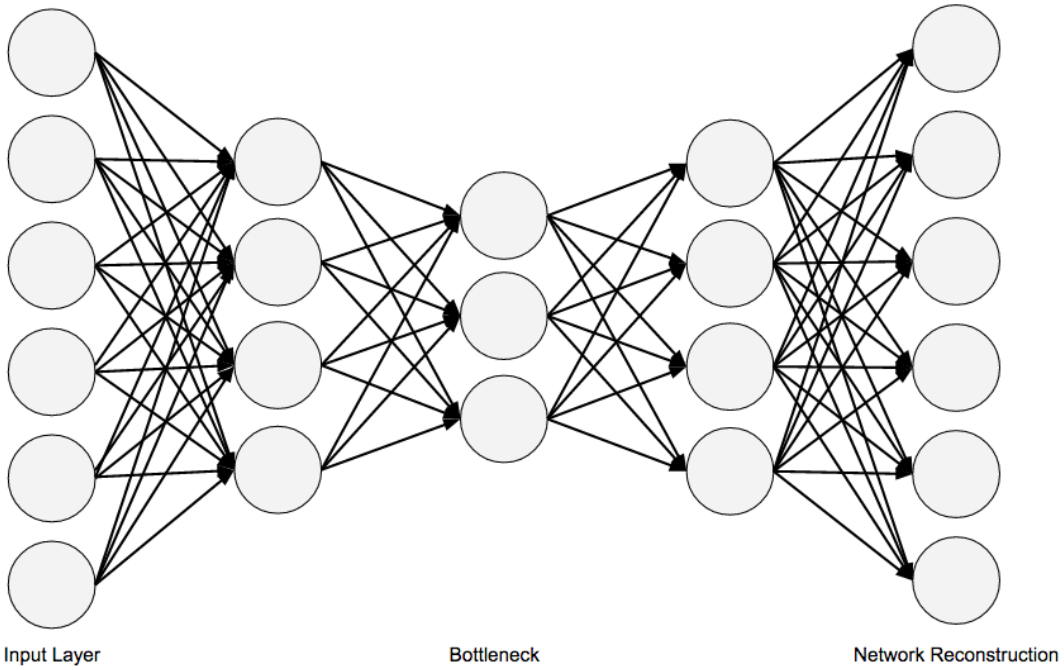


Figure 3 The Autoencoder model

Building a model to represent the input dataset might not sound useful on the surface. However, we're **less interested in the output itself** and more **interested** in the **difference between** the **input** and **output representations**. If we can train a neural network to learn data it commonly "sees," then this network can also let us know when it's "seeing" data that is unusual, or anomalous.

5.4 Major Deep Networks Architectures

From the class of **Unsupervised Pretrained Networks** we will analyze **Deep Belief Networks (DBNs)** and **Generative Adversarial Networks (GANs).**

Deep Belief Networks

DBNs are composed of layers of Restricted Boltzmann Machines (RBMs) for the pretrain phase and then a feed-forward network for the fine-tune phase. Figure 4 shows the network architecture of a DBN.

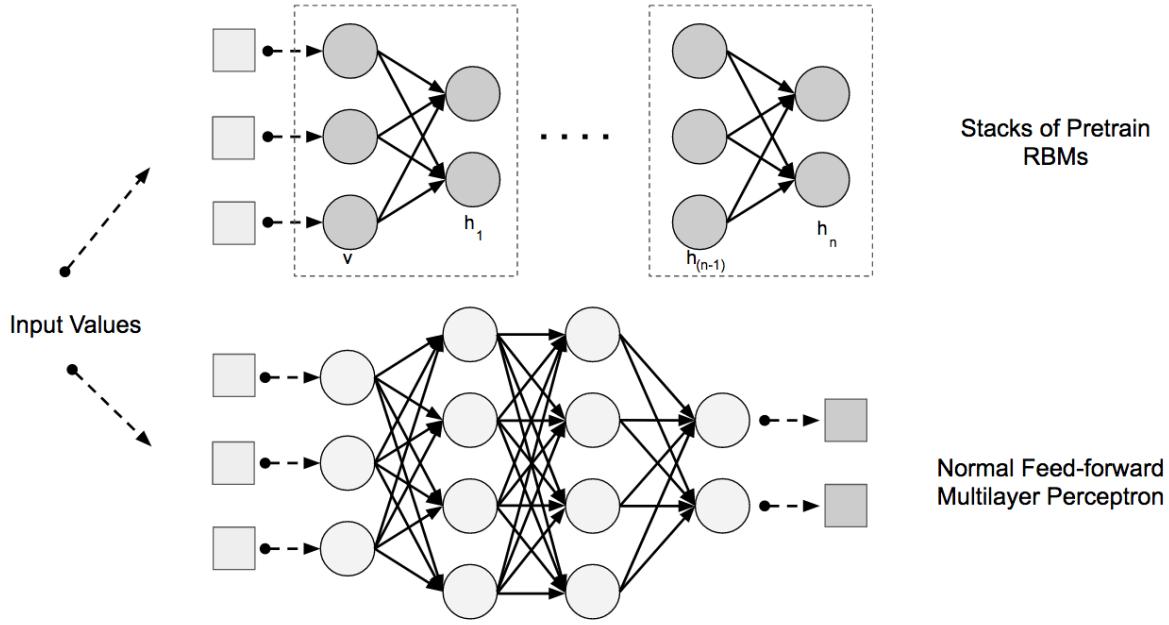


Figure 4 The DBN model

Such a model uses **RBMs to extract higher-level features** from the raw input vectors. To do that, we want to set the hidden unit states and weights such that when we show the RBM an input record and ask the RBM to **reconstruct** the record the record, it **generates something close** to the **original input** vector.

The fundamental purpose of RBMs in the context of deep learning and DBNs is to learn these **higher-level features** of a dataset in an **unsupervised** training fashion.

Generative Adversarial Networks

GANs have been shown to be quite adept at **synthesizing novel images** based on other training images. GANs are an example of a network that uses **unsupervised learning** to train two models in parallel.

A key aspect of GANs (and **generative models** in general) is how they use a **parameter count** that is significantly **smaller** than normal with respect to the amount of data on which we're training the network. The **network** is forced to **efficiently represent** the training **data**, making it more effective at generating data similar to the training data. A GAN is composed of a/some **discriminator network(s)** and a **generative network**.

The **discriminator networks** take images as input, and then **output a classification**. The gradient of the output of the discriminator network with respect to the synthetic input data indicates how to make small changes to the synthetic data to make it more realistic. The generative network in **GANs generates data** (or images) with a special kind of layer called a **deconvolutional layer**. During training, we use **backpropagation** to update the generating network's parameters to generate more realistic output images. The goal here is to update the generating network's parameters to the point at which the discriminating network is sufficiently "fooled" by the generating network because the output is so realistic as compared to the training data's real images.

Next we will analyze **Convolutional Neural Networks (CNN).**

The goal of a **CNN** is to **learn higher-order features** in the data via **convolutions**. They are well suited to **object recognition** with images and consistently top image classification competitions. They can identify faces, individuals, street signs, platypuses, and many other aspects of visual data.

CNNs overlap with text analysis via optical character recognition, but they are also useful when **analyzing words** as **discrete textual units**. They're also good at **analyzing sound**.

The **efficacy of CNNs** in image recognition is one of the main reasons why the world **recognizes the power of deep learning**. As Figure 5 illustrates, CNNs are good at building **position** and **rotation invariant features** from **raw image data**.

CNNs tend to be most **useful** when there is some **structure in the input data**. An example would be how **images** and **audio** data that have a specific set of **repeating patterns** and input values next to each other are **related spatially**. **Conversely**, the columnar data exported from a **relational database management system** (RDBMS) tends to **have no structural relationships** spatially. Columns next to one another just happen to be materialized that way in the database exported materialized view.

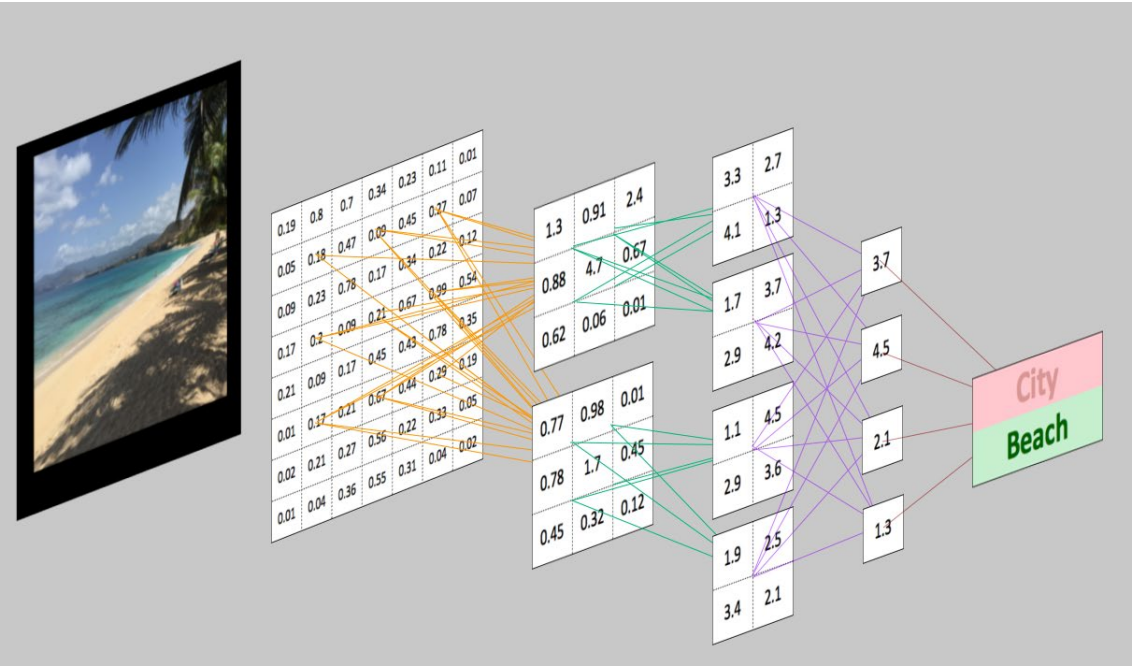


Figure 5 The CNN model for image recognition

CNNs transform the input data from the input layer through all connected layers into a set of class scores given by the output layer. There are many variations of the CNN architecture, but they are based on the pattern of layers, as demonstrated in Figure 6.

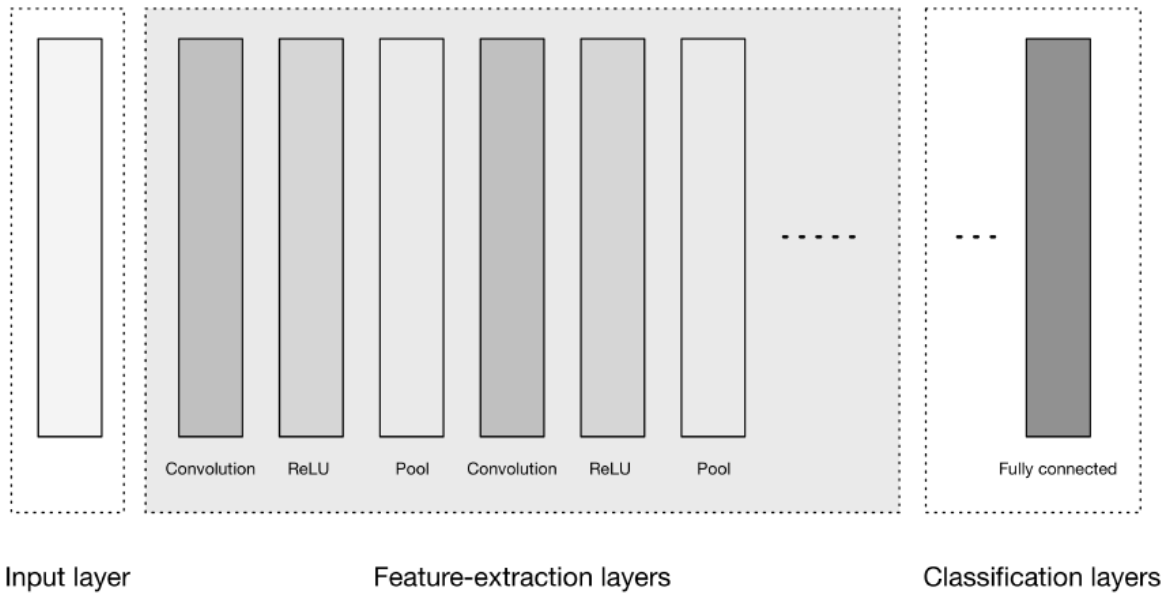


Figure 6 Generic CNN model

In the following paragraphs we look at the **individual layers** in **the CNN architecture** and emphasize the core principles of operation and the impact each layer has in the overall processing capabilities of the network.

Input Layers

Input layers are where we load and store the raw input data of the image for processing in the network. This input data specifies the width, height, and number of channels. Typically, the number of channels is three, for the RGB values for each pixel.

Convolutional Layers

Convolutional layers are considered the core building blocks of CNN architectures. The layer will compute a dot product between the region of the neurons in the input layer and the weights to which they are locally connected in the output layer. The resulting output generally has the same spatial dimensions (or smaller spatial dimensions) but sometimes increases the number of elements in the third dimension of the output (depth dimension).

A **convolution** is defined as a **mathematical operation** describing a rule for how to merge two sets of information. It is important in both physics and mathematics and defines a bridge between the space/time domain and the frequency domain through the use of Fourier transforms. It takes input, applies a convolution kernel, and gives us a feature map as output.

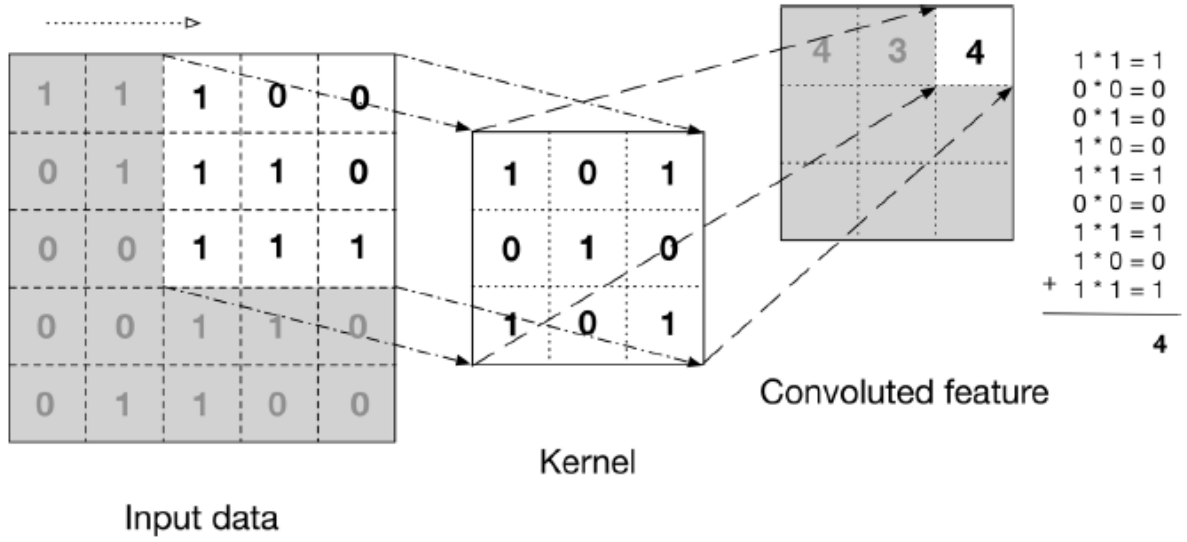


Figure 7 The convolution operation in CNNs

The **convolution operation**, shown in Figure 7, is known as the **feature detector of a CNN**. The **input** to a convolution can be **raw data** or a feature map output from another convolution. It is often **interpreted as a filter** in which the kernel filters input data for certain kinds of information; for example, an edge kernel lets pass through only information from the edge of an image.

Pooling Layers

Pooling layers are commonly inserted between successive convolutional layers. We want to follow convolutional layers with pooling layers to **progressively reduce** the

spatial size (width and height) of the data representation. Pooling layers **reduce** the **data representation progressively** over the network and help control overfitting. The pooling layer operates independently on every depth slice of the input.

Fully Connected Layers

We use this layer to **compute class scores** that we'll use as output of the network (e.g., the output layer at the end of the network). Fully connected layers have the normal parameters for the layer and hyperparameters. Fully connected layers **perform transformations** on the input data volume that are a function of the activations in the input volume and the parameters (weights and biases of the neurons).

In the next section we will look at **Recurrent Neural Networks.**

Recurrent Neural Networks are in the family of feed-forward neural networks. They are **different from other feed-forward networks** in their ability to **send information over time-steps**. Recurrent Neural Networks take each vector from a sequence of input vectors and model them one at a time. This allows the network to **retain state** while modeling each input vector across the window of input vectors. **Modeling the time dimension** is a hallmark of Recurrent Neural Networks.

Recurrent Neural Networks are a **superset of feed-forward neural networks** but they add the concept of **recurrent connections**. These connections (or recurrent edges) **span adjacent time-steps** (e.g., a previous time-step), giving the model the **concept of time**. The conventional connections do not contain cycles in recurrent neural networks. However, recurrent connections can form cycles including connections back to the original neurons themselves at future time-steps.

LSTM Networks

Long Short-Term Memory (LSTM) networks are the most commonly used variation of Recurrent Neural Networks. LSTM networks were introduced in 1997 by Hochreiter and Schmidhuber.

The **critical component of the LSTM** is the **memory cell and the gates** (including the forget gate, but also the input gate). The contents of the **memory cell** are **modulated** by the **input gates** and **forget gates**. Assuming that both of these gates are closed, the contents of the memory cell will remain unmodified between one time-step and the next. The **gating structure allows information to be retained** across many time-steps, and consequently also allows gradients to flow across many time-steps. This allows the **LSTM** model to **overcome** the **vanishing gradient** (i.e, gradients become too large or too small and make it difficult to model long-range dependencies in the structure of the input dataset) problem that occurs with most Recurrent Neural Network models.

The generic architecture of a LSTM block is depicted in Figure 8.

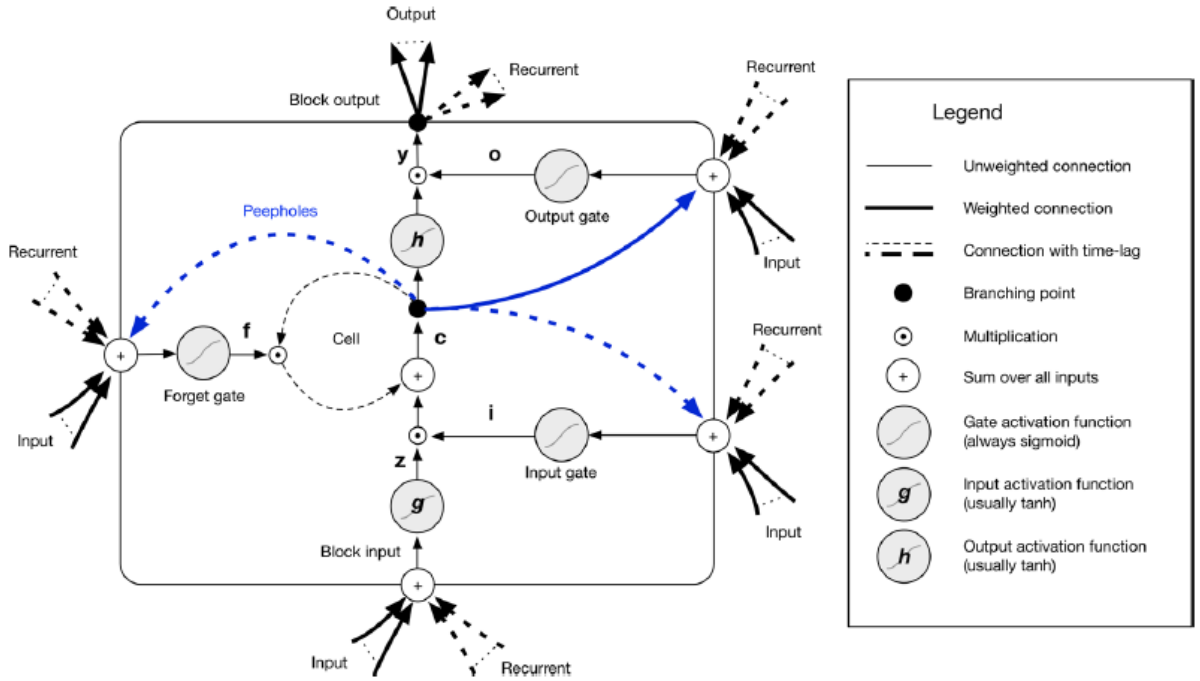


Figure 8 Generic architecture of LSTM block

LSTM layers

A **basic layer** accepts an input vector x (non-fixed) and gives output y. The output y is influenced by the input x and the history of all inputs. The layer is influenced by the **history** of **inputs** through the **recurrent connections**. The RNN has some internal state that is updated every time we input a vector to the layer. The **state** consists of a **single hidden vector**.

Training LSTM

LSTM networks use **supervised learning** to update the weights in the network. They train on one input vector at a time in a sequence of vectors. Vectors are real-valued and become sequences of activations of the input nodes. Every non-input unit computes its current activation at any given time-step. This **activation value** is **computed** as the **nonlinear function** of the **weighted sum** of the **activations** of all **units** from which it **receives connections**. For each input vector in the sequence of input, the **error is equal to the sum of the deviations** of all target signals from corresponding activations computed by the network.

Backpropagation through time (BPTT)

Recurrent Neural Network training can be computationally expensive. The traditional option is to use BPTT. BPTT is fundamentally the same as standard backpropagation: we apply the chain rule to work out the derivatives (gradients) based on the connection structure of the network. It's through time in the sense that some of those gradients/error signals will also flow backward from future time-steps to current time-steps, not just from the layer above (as occurs in standard backpropagation).

RNNs and CNNs are usually used together to exploit both the structure in the data and the temporal component in tasks such as labelling objects in images, as shown in Figure 9.

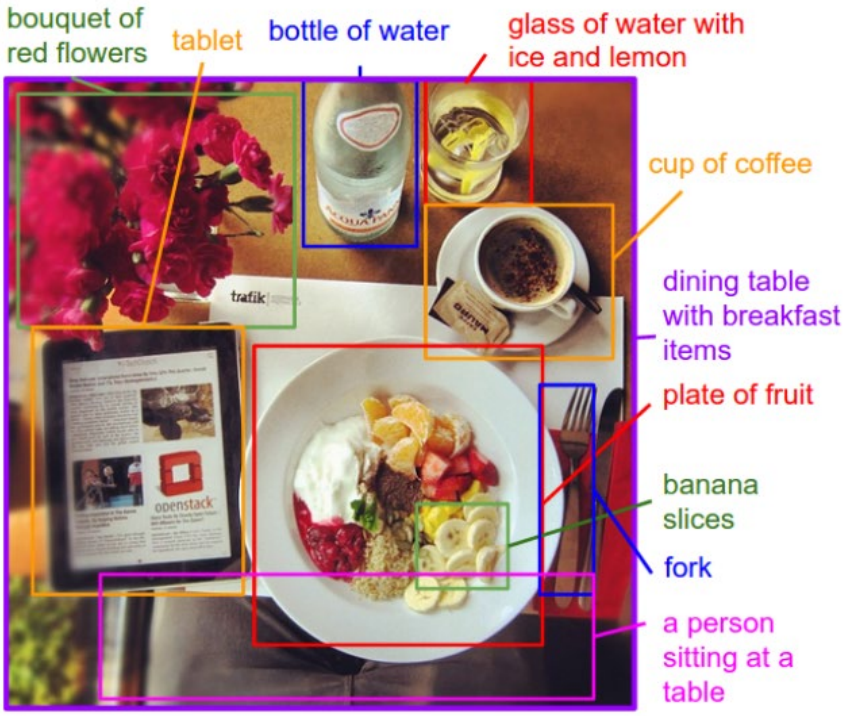


Figure 9 Labeling images with a blended CNN/Recurrent Neural Network

We conclude our analysis of deep neural processing architectures with the analysis of **Recursive Neural Networks.**

Recursive Neural Networks, like Recurrent Neural Networks, can **deal with variable length input**. The primary difference is that Recurrent Neural Networks have the ability to **model the hierarchical structures** in the **training dataset**. Images commonly have a scene composed of many objects. Deconstructing scenes is often a problem domain of interest yet is nontrivial. The **recursive nature** of this deconstruction challenges us to not only **identify the objects** in the scene, but also **how the objects relate** to form the scene.

A Recursive Neural Network architecture is **composed of a shared-weight matrix** and a **binary tree structure** that allows the recursive network to **learn varying sequences** of words or parts of an image. It is useful as a sentence and scene parser. Recursive Neural Networks use a variation of backpropagation called **backpropagation through structure (BPTS).** The **feed-forward pass happens bottom-up**, and **backpropagation** is **top-down**. Think of the objective as the top of the tree, whereas the inputs are the bottom.

Both Recursive and Recurrent Neural Networks share many of the same use cases. **Recurrent Neural Networks** are traditionally used in **Natural Language Processing**

NLP because of their ties to binary trees, contexts, and natural-language-based parsers. In the case of **Recursive Neural Networks**, it is a constraint that we use a **parser** that builds the **tree structure** (typically constituency parsing). Recursive Neural Networks can **recover both granular structure** and **higher-level hierarchical structure** in **datasets** such as **images** or **sentences**.

When to use deep learning?

You should use deep learning when...

- Simpler models (logistic regression) don't achieve the accuracy level your use case needs
- You have complex pattern matching in images, NLP, or audio to deal with
- You have high dimensionality data
- You have the dimension of time in your vectors (sequences)

When to stick with traditional machine learning?

You should use a traditional machine learning model when...

- You have high-quality, low-dimensional data; for example, columnar data from a database export
- You're not trying to find complex patterns in image data

In summary, **deep learning** is an **approach to machine learning** that has drawn heavily on our knowledge of the **human brain**, **statistics** and **applied math** as it developed over the past several decades. In recent years, deep learning has seen **tremendous growth** in its popularity and usefulness, largely as the result of more **powerful computers**, **larger datasets** and **techniques to train deeper networks**. The years ahead are full of challenges and opportunities to improve deep learning even further and to bring it to new frontiers.

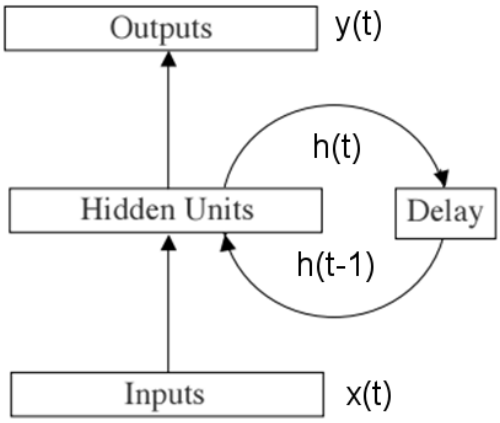
Contents

Lectures

6. Technical implementations of neural computation

6.1. Recurrent networks

The human brain is wired not only to **recognize individual instances** but to also to analyze entire **sequences of inputs**. These sequences are rich in information, have complex **time dependencies**, and can be of arbitrary length. For example, vision, motor control, speech, and comprehension require high-dimensional processing of their inputs, as they change over time. This is something that feed-forward networks are poor at modeling.



One promising solution to tackling the problem of learning sequences of information is the **recurrent neural network (RNN)**. Such networks are built on the same computational unit as the feed forward neural networks, but differ in the architecture of how these neurons are connected to one another. Feed forward neural networks were **organized in layers**, where information flowed **unidirectionally** from input units to output units. There were no undirected cycles in the connectivity patterns. Neurons in the brain do contain undirected cycles as well as connections within layers and similarly, in order to create more powerful computational systems such as those modeled by RNNs. The generic processing scheme behind a RNN is depicted in the following figure.

A **RNN can learn many behaviors / sequence** processing tasks that are not learnable by traditional machine learning methods. This supported the use of RNNs for technical applications: general computers which can learn algorithms to map input sequences to output sequences, with or without a teacher. They are computationally more **powerful** and **biologically more plausible** than other **adaptive** approaches such as Hidden Markov Models (no continuous internal states), feed-forward networks or Support Vector Machines (no internal states at all).

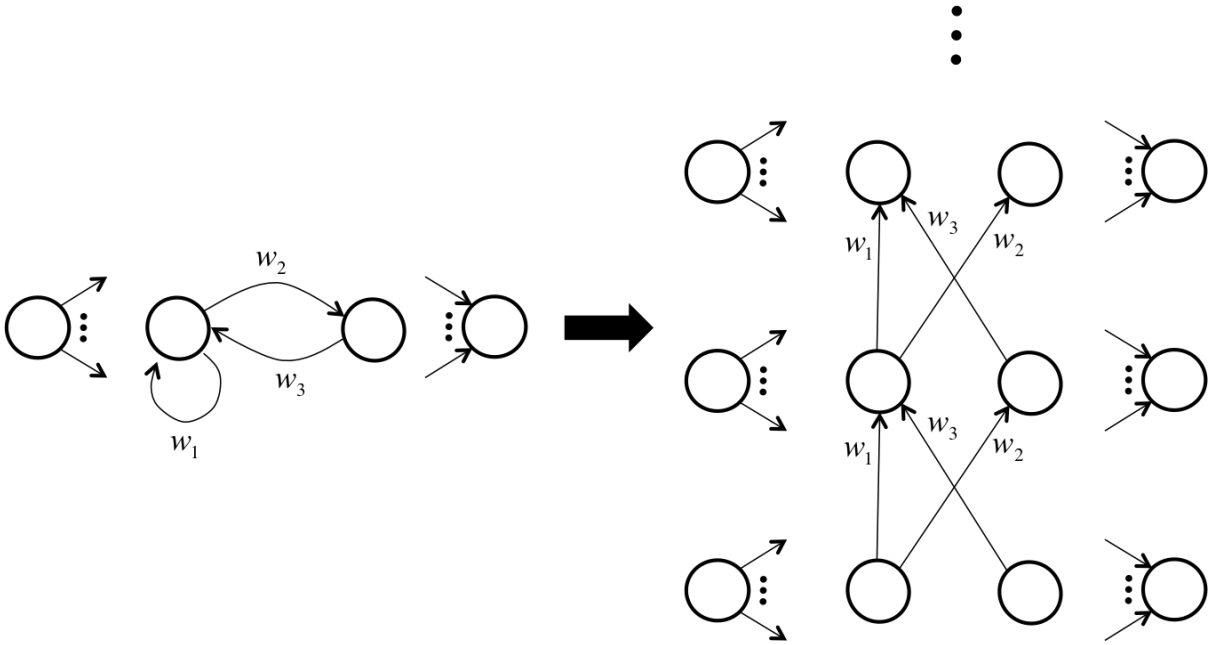
Training a RNN - Backpropagation through time

How do we train a RNN to achieve such a behavior? Specifically, how do we determine the connection weights? And how do we choose the initial activities of all of the hidden

units? An initial idea might be to use **backpropagation** directly, due to its successful use in feed forward neural networks.

The **problem with using backpropagation in RNNs** is that there are cyclical dependencies. In feed-forward networks, when we calculate the error derivatives with respect to the weights in one layer, we could express them completely in terms of the error derivatives from the layer above. In a recurrent neural network, we don't have this layering because the neurons do not form a directed acyclic graph. Trying to backpropagate through a RNN could force us to try to express an error derivative in terms of itself, which is not analytically tractable.

So how can we use backpropagation for RNNs, if at all? The answer lies in employing a transformation, where we convert our RNN into a new structure that's essentially a feed-forward neural network. This strategy is termed "**unrolling**" **the RNN through time.**



The process is simple, but it has a profound impact on our ability to analyze the neural network. We take the RNN's inputs, outputs, and hidden units and replicate it for every time step. These replications correspond to layers in our new feed forward neural network. We then connect hidden units as follows. If the original RNN has a connection of weight ω from neuron i to neuron j, in our feed forward neural network, we draw a connection of weight ω from neuron i in every layer t_k to neuron j in every layer t_{k+1}.

Thus, to train our RNN, we randomly initialize the weights, "unroll" it into a feed forward neural network, and backpropagate to determine the optimal weights. To determine the initializations for the hidden states at time t_0, we can treat the initial activities as parameters fed into the feed forward network at the lowest layer and backpropagate to determine their optimal values as well.

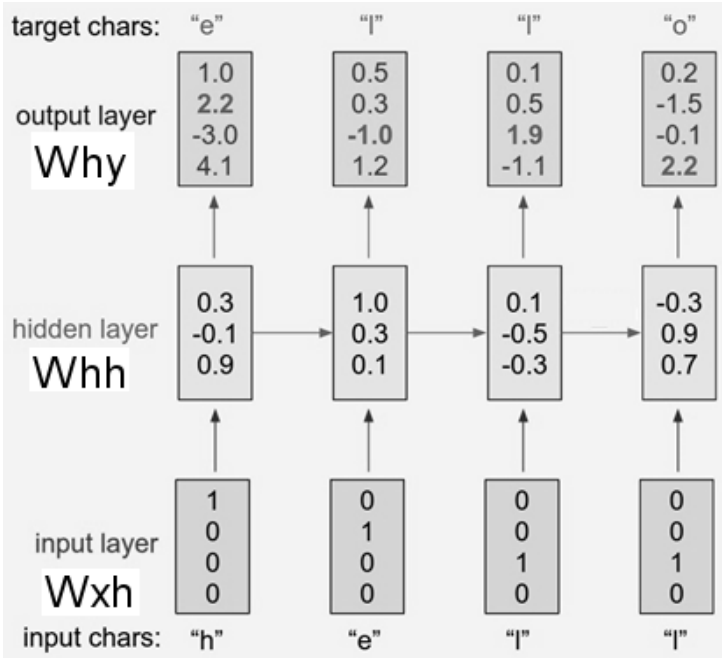
We run into a problem however, which is that after every batch of training examples we use, we need to modify the weights based on the error derivatives we calculated. In our feed-forward network, we have sets of connections that all correspond to the same connection in the original RNN. The error derivatives calculated with respect to their weights, however, are not guaranteed to be equal, which means we might be modifying them by different amounts.

We can get around this problem, by averaging (or summing) the error derivatives over all the connections that belong to the same set. This means that after each batch, we modify corresponding connections by the same amount, so if they were initialized to the same value, they will end up at the same value.

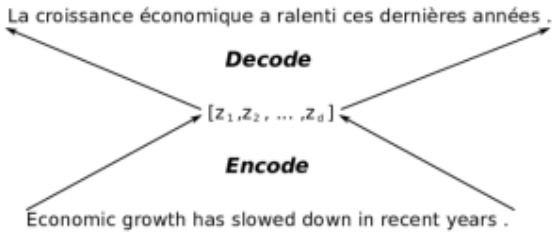
Typical applications of RNNs

RNNs have shown great success in many tasks, such as: language modeling and text generation, machine translation, speech recognition and image description generation. At the moment the most commonly used type of **RNNs** are **Long Short-Term Memory Networks (LSTMs)**.

With respect to **language modeling and text generation** given a sequence of words we want to predict the probability of each word given the previous words. Language models allow us to measure how likely a sentence. Such a metric is important for machine translation (since high-probability sentences are typically correct). A side-effect of being able to predict the next word is that we get a **generative model**, which allows us to generate new text by **sampling** from the output **probabilities**. In language modeling the input is typically a **sequence of words** (encoded as one-hot vectors for example), and our output is the **sequence of predicted words,** as shown in the following diagram where W_{hx}, W_{hh}, W_{hy}, are the input, hidden and output weights of the network.

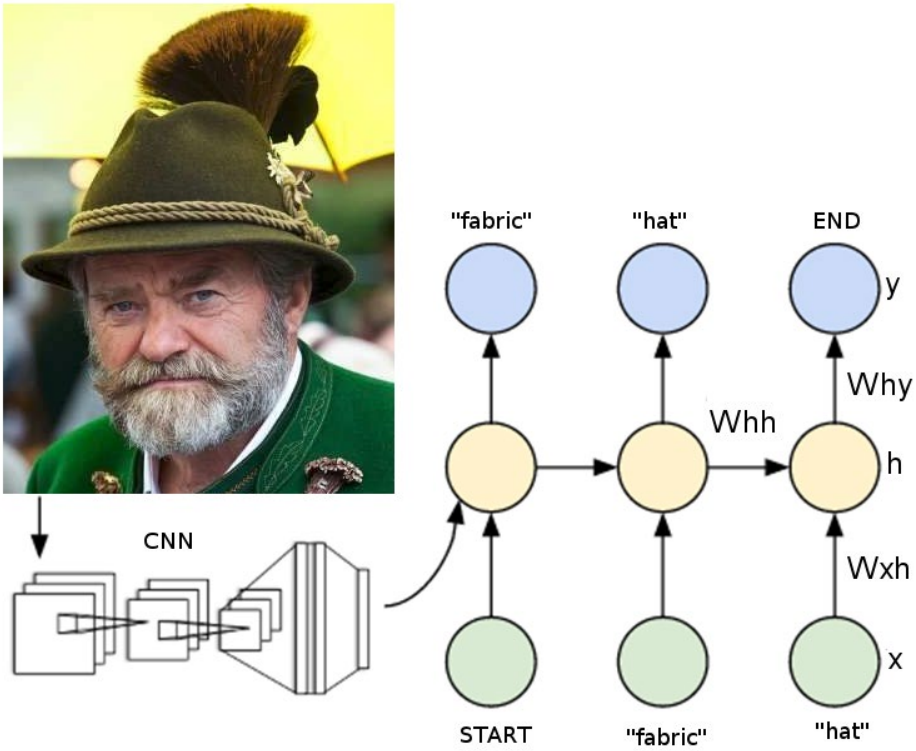


Machine translation is similar to language modeling in that the input is a sequence of words in a source language (e.g. German). We want to output a sequence of words in a target language (e.g. English). A key difference is that our output only starts after the system has seen the **complete input**, because the first word of the translated sentences may require **information captured** from the complete input sequence, as shown in the figure.



With respect to **speech recognition**, given an input sequence of acoustic signals from a sound wave, a RNN can predict a sequence of phonetic segments together with their probabilities.

Finally, together with **Convolutional Neural Networks (CNNs)**, RNNs have been used as part of a model **to generate descriptions for unlabeled images**. The combined model even aligns the generated words with features found in the images, as shown in the following figure.

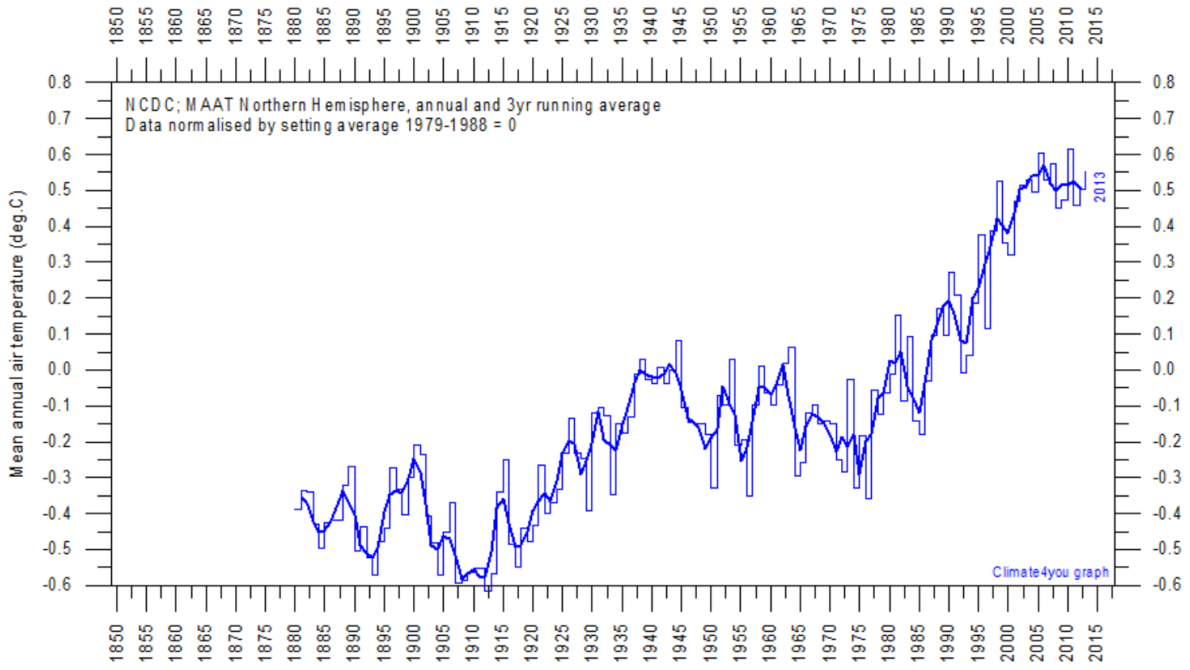


Another application where **RNNs** are successful is **time-series prediction**. Time series prediction problems are a difficult type of predictive modeling problem. Unlike

regression predictive modeling, time series also adds the complexity of sequence dependence among the input variables.

6.2 Time-series prediction

A time series is a **sequence of data** points with a **natural temporal order**, measured usually at uniform time intervals. Typical examples are financial markets (e.g. economic factors, financial indexes, exchange rate, spread), meteorology (e.g. weather variables, like temperature, pressure, and wind), biomedicine (e.g. physiological signals, heart-rate, patient temperature), web (e.g. clicks, logs) etc.



The analysis of time-series brings a deep understanding of the data. First of all, time-series analysis can realize: the **prediction** of the future based on the past, the **control** of the process producing the series, the **understanding** of the mechanism generating the series, and finally the **description** of the salient features of the series.

Forecasting a time series is possible since future depends on the past or analogously because there is a relationship between the future and the past. However this **relation is not deterministic** and can **hardly** be **explained** in an **analytical** form.

The selection of a proper model is extremely important as it reflects the **underlying structure of the series** and this fitted model in turn is used for future forecasting. A **time series model** is said to be **linear or non-linear** depending on whether the current value of the series is a linear or non-linear function of past observations.

In general models for time series data can have many forms and represent different stochastic processes. There are two widely used linear time series models in literature,

namely the **Autoregressive** (AR) and **Moving Average** (MA) models. Combining these two, the **Autoregressive Moving Average** (ARMA) and **Autoregressive Integrated Moving Average** (ARIMA) models have been proposed in literature. The **Autoregressive Fractionally Integrated Moving Average** (ARFIMA) model generalizes ARMA and ARIMA models.

An ARMA model is a combination of AR and MA models and is suitable for univariate time series modeling. In an AR model the future value of a variable is assumed to be a linear combination of *p* past observations and a random error together with a constant term. Mathematically the AR model can be expressed as:

$$y_t = c + \sum_{i=1}^{p} \phi_i y_{i-1} + \epsilon_t$$

Here y_t and ϵ_t are respectively the actual value and random error (or random shock) at time *t*, ϕ_i *(i = 1,2,..., p)* are model parameters and *c* is a constant. The integer constant *p* is the order of the model.

Just as an AR model regress against past values of the series, an MA model uses past errors as the explanatory variables. The MA model is given by:

$$y_t = \mu + \sum_{j=1}^{q} \theta_j \epsilon_{t-j} + \epsilon_t$$

Here *μ* is the mean of the series, θ_j *(j = 1,2,...,q)* are the model parameters and *q* is the order of the model.

Autoregressive and moving average models can be effectively combined together to form a general and useful class of time series models, known as the ARMA models. Mathematically an ARMA model is represented as:

$$y_t = c + \epsilon_t + \sum_{i=1}^{p} \phi_i y_{i-1} + \sum_{j=1}^{q} \theta_j \epsilon_{t-j}$$

Here the model orders *p, q* refer to *p* autoregressive and *q* moving average terms.

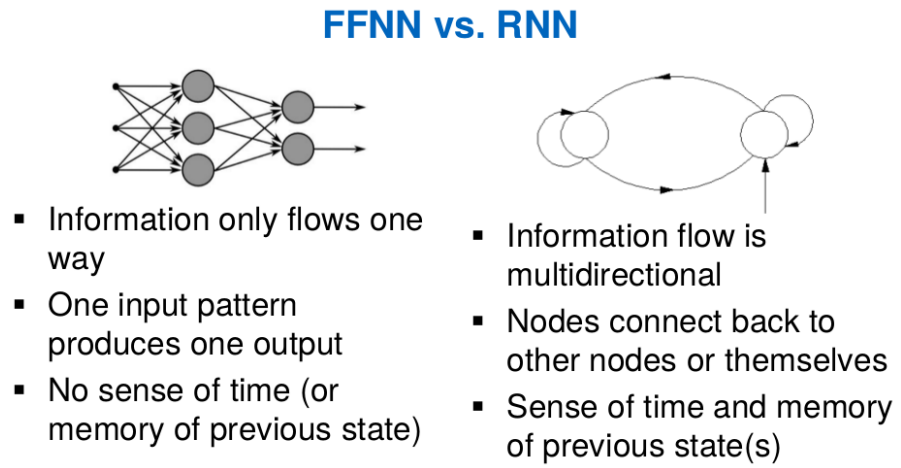
The ARMA models, described above can only be used for stationary time series data. However in practice many time series such as those related to socio-economic and business show non-stationary behavior. Time series, which contain trend and seasonal patterns, are also non-stationary in nature. Thus from application view point ARMA models are inadequate to properly describe non-stationary time series, which are frequently encountered in practice. For this reason the ARIMA model is proposed, which is a generalization of an ARMA model to include the case of non-stationarity as well.

Although linear models have drawn much attention due to their relative simplicity in understanding and implementation, many practical time series show non-linear patterns. Various nonlinear models have been suggested in literature: **Autoregressive Conditional Heteroskedasticity** (ARCH), **Non-linear Autoregressive** (NAR), and **Nonlinear Moving Average** (NMA).

Why using RNNs for prediction?

AR, MA and ARMA models are limited to prediction of linear system dynamics whereas RNNs can approximate nonlinear functions (i.e. ***Universal approximation theorem***). Moreover, RNNs can be applied without an extensive analysis of underlying assumptions and are useful when knowledge is difficult to specify but there is an abundance of examples (non-parametric modeling). Hence, time series prediction is based on the inference of future behavior from examples of past behavior.

Time-series prediction can be realized with various architectures of **neural networks**, both **feedforward (FFNN)** and **recurrent (RNN)**. We provide a comparative view on how such neural processing architectures can tackle the forecasting problem.

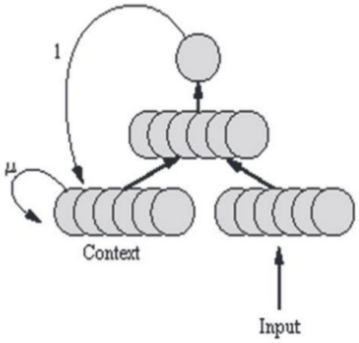


Providing an extension of the multilayer perceptron with context units simple RNN have a memory or sense of time. Such networks are useful for tasks that are dependent on a sequence of successive states. A RNN can predict the next item in a sequence from the current and preceding input and can be trained by back-propagation, as previously shown. There are 2 types of simple RNNs, **Jordan** and **Elman networks**, which differ through their internal connectivity (see figure).

Simple Recurrent Neural Networks (SRNN)

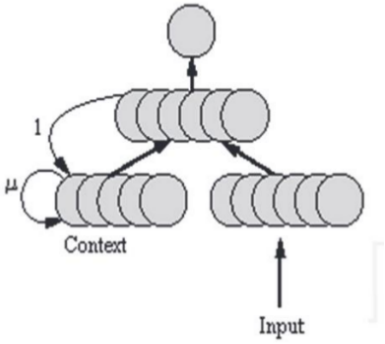
Two types of SRNN:

Jordan network



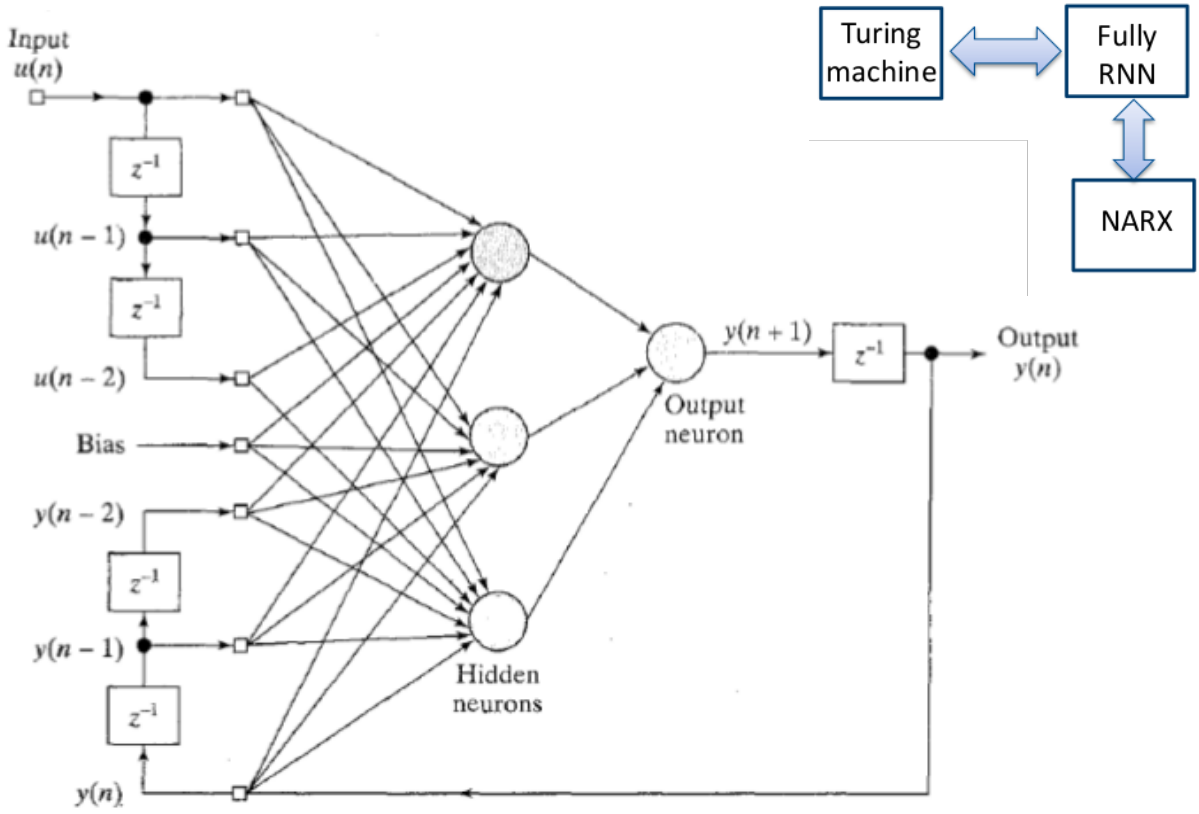
Feedback from the output to the context layer

Elman network



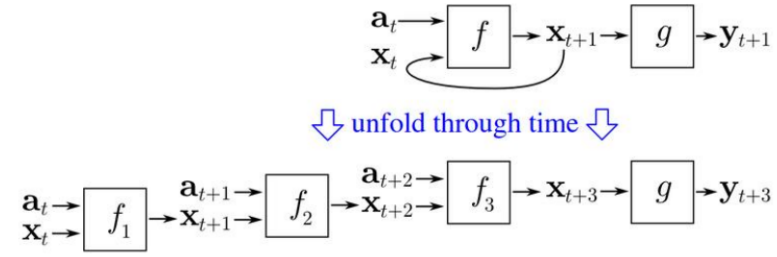
Feedback from the hidden to the context layer

In the following section we analyze the basics of **Backpropagation through time** as applied to **Nonlinear Auto Regressive with eXogenous inputs (NARX)** models. Such models are used to implement RNNs in real world scenarios. NARX relates the current value of a time series to the past values of: the time series and the driving exogenous series, as shown in the following diagram:



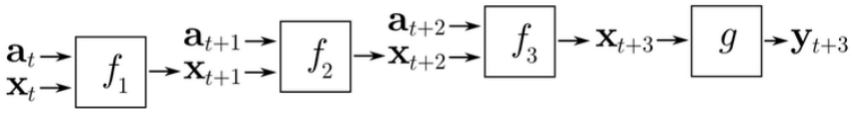
Applying **Backpropagation through time** to the **NARX** model we have.

- BPTT is a gradient-based technique for training RNN



- Training data: $\langle \mathbf{a}_0, \mathbf{y}_0 \rangle, \langle \mathbf{a}_1, \mathbf{y}_1 \rangle, \langle \mathbf{a}_2, \mathbf{y}_2 \rangle, \ldots, \langle \mathbf{a}_{n-1}, \mathbf{y}_{n-1} \rangle$

$$\langle \mathbf{x}_t, \mathbf{a}_t, \mathbf{a}_{t+1}, \mathbf{a}_{t+2}, \ldots, \mathbf{a}_{t+k-1}, \mathbf{y}_{t+k} \rangle$$



For each pattern presentation:

- Update the weights in each instance of f according to standard back-propagation;
- Assign to the weights the average of all weights across instances;
- Compute X_{t+1}.

Despite it provides a faster solution finding than general-purpose optimization techniques the method risks to fall in **local optima** or **vanishing gradient**. Finally, the computational power of such models was analyzed and described through two theorems and a corollary:

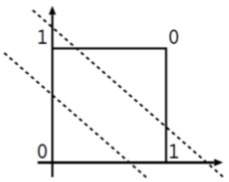
- Theorem I (Siegelmann and Sontag, 1991)
  - All Turing machines may be simulated by fully connected RNN built on neurons with sigmoid activation function.
- Theorem II (Siegelmann et al. 1997)
  - NARX network with one layer of hidden neurons with sigmoid activation function and a linear output neuron can simulate fully connected RNN with sigmoid activation function except for a linear slowdown.
- Corollary (Giles, 1996)
  - NARX networks with one hidden layer of neurons and sigmoid activation function and a linear output neuron are Turing equivalent.

In the next section we analyze some basic examples on applying such models for prediction.

Example 1: XOR time series

- Input: 10100001111010101.....

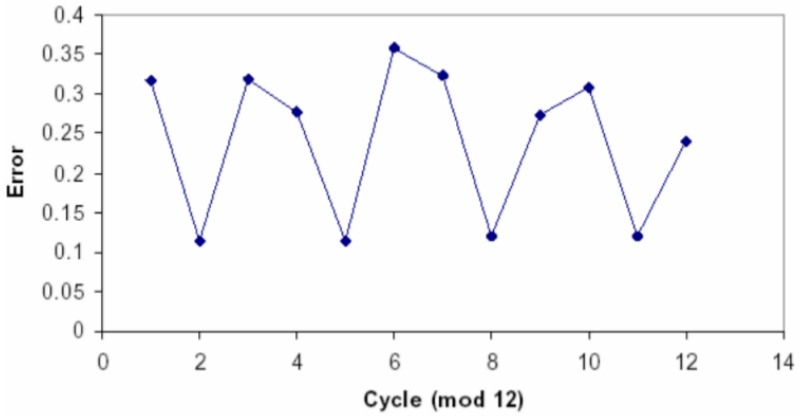
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0



- Bit 3 is XOR of bit 1 and 2,

bit 6 is XOR of 4 and 5, and so on

- A SRNN with 1 input, two context, two hidden and one output unit is trained on a sequence of 3000 bits.
- Input: 1 0 1 0 0 0 0 1 1 1 1 0 1 0 1. . .
- Output: 0 1 0 0 0 0 1 1 1 1 0 1 0 1 ?. .



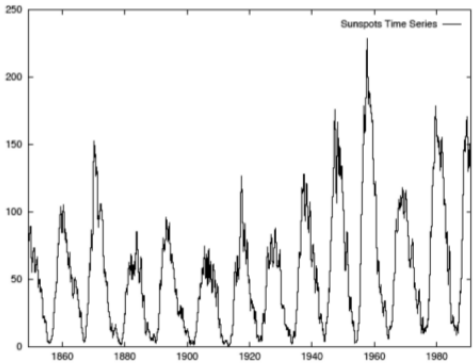
- Graph of root mean squared error over 12 consecutive inputs in sequential XOR task.
- Data points are averaged over 1200 trials.

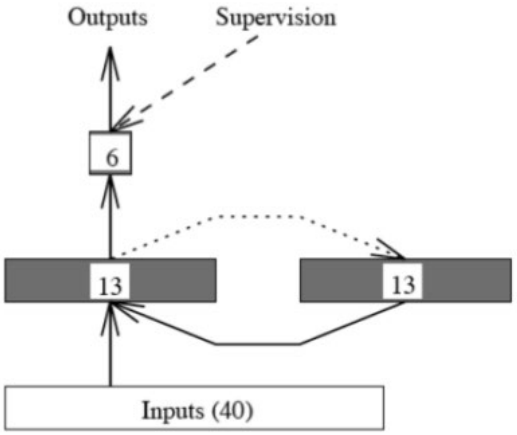
Example 2: predicting sunspot activity

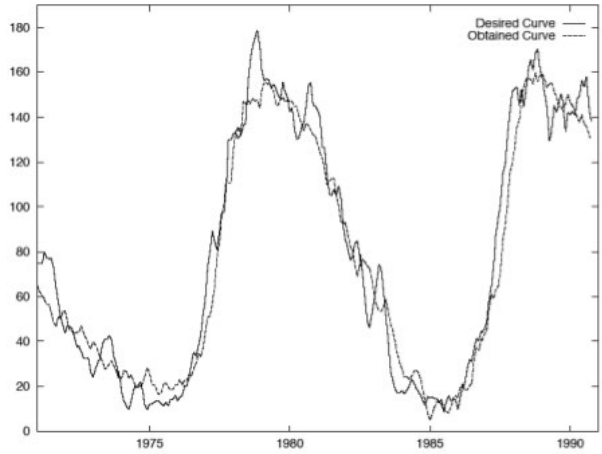
(Fessant, Bengio and Collobert)

- Problem: sunspots affect ionospheric propagation of radio waves
- Telecom companies want to predict sunspot activity six months in advance.
- Sunspots follow an 11 year cycle, varying from 9-14 years
- Goal: predicting IR5, a smoothed index of monthly solar activity

Monthly data of solar activity







Input: $\{x[t-40], \cdots, x[t-1]\}$

Output: $\{\hat{x}[t], \cdots, \hat{x}[t+5]\}$

Up to this point, we have studied about various stochastic and neural network methods for time series modeling and forecasting. Despite of their own strengths and weaknesses, these methods are quite successful in forecasting applications. Recently, a new statistical learning theory, the **Support Vector Machine (SVM)** has been receiving increasing attention for **classification** and **forecasting**.

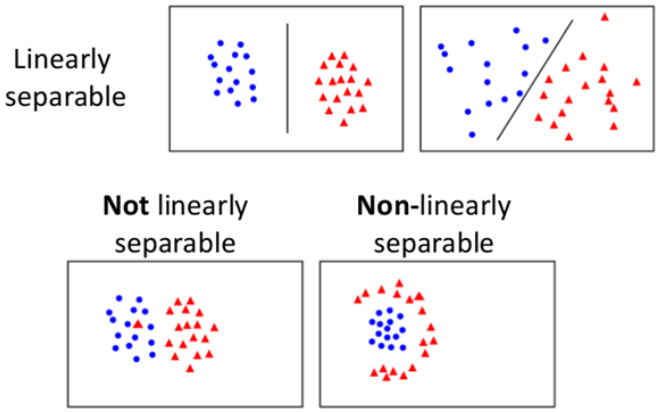
6.3. Support Vector Machines

Initially **SVMs** were designed to solve **pattern classification problems**, such as optimal character recognition, face identification and text classification, etc. But soon they found wide applications in other domains, such as **function approximation**, **regression** estimation and **time series prediction** problems.

The objective of SVM is to find a **decision rule** with good generalization ability through selecting some particular subset of training data, called **support vectors.** In this method, an optimal **separating hyperplane** is constructed, after **nonlinearly mapping the input space** into a **higher dimensional feature space**. Thus, the quality and complexity of SVM solution does not depend directly on the input space.

The **training process** is equivalent to **solving** a linearly constrained **quadratic programming** problem. So, contrary to other networks' training, the **SVM solution is always unique and globally optimal**. However a major **disadvantage of SVM** is that when the training size is large, it requires an **enormous amount of computation** which increases the **time complexity** of the solution.

The main idea of SVM when applied to **binary classification** problems is to find a canonical hyperplane which maximally separates the two given classes of training samples, as shown in the following diagram.

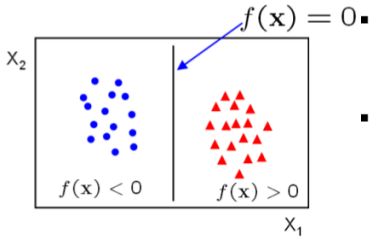


For this classification problem, we can define a linear classifier for both a 2D and 3D input space.

Linear classifier

$$f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$$

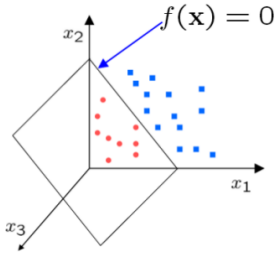
- in 2D the discriminant is a line
- **w** is the normal to the line and **b** the bias
- **w** is the weight vector
- **w** and **b** depend on the training data



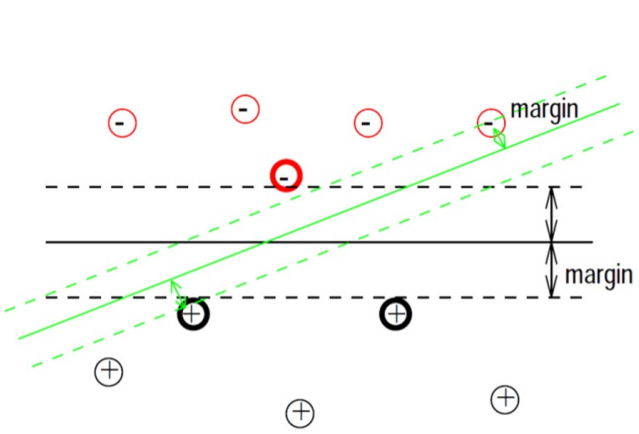
Linear classifier in a 3D space

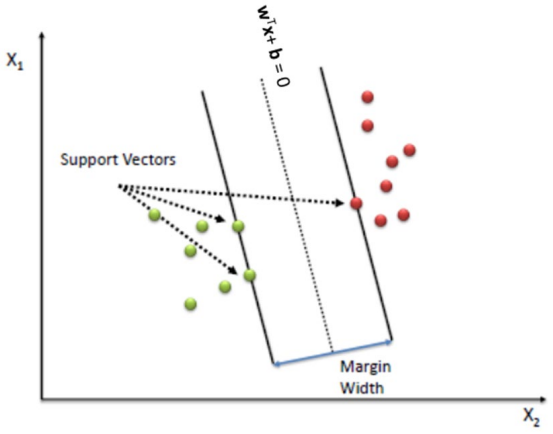
$$f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$$

- in 3D the discriminant is a plane
- in *n*D it is a hyperplane

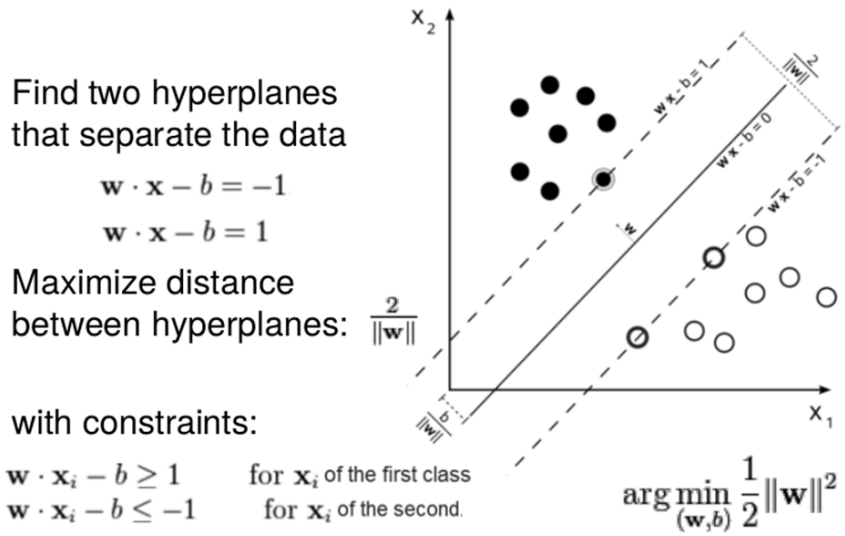


But, a question arises, which is the best (optimal) linear classifier? The solution is to select the most stable under perturbations of the inputs (i.e. the **maximum margin solution**), as shown in the next diagram depicting the margins and support vectors.

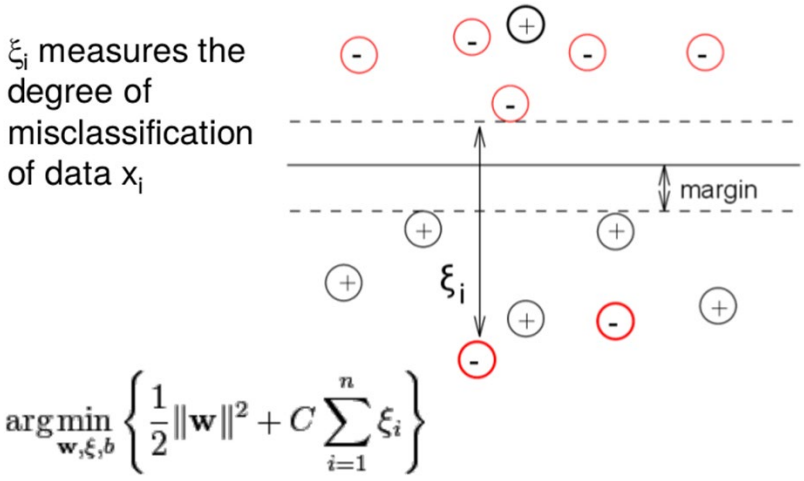




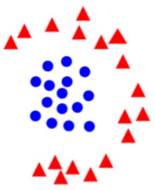
The margin maximization process assumes finding the hyperplanes separating the data which maximize a certain distance metric, as depicted in the next diagram.



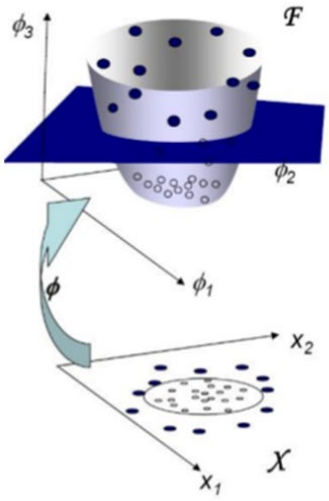
In the case there are outliers the problem is which is the best *w* with outliers? In this case we must make a tradeoff between the margin and the number of mistakes on the training data. The solution comes from calculating the soft margin as shown in the following diagram.



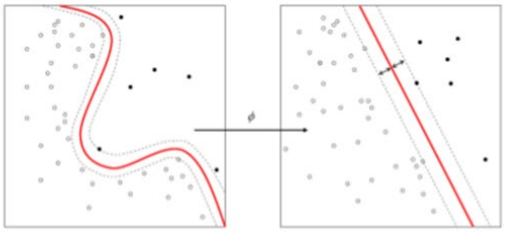
In SVM applications it is convenient to map the points of the input space to a **high dimensional feature space** through some **non-linear mapping** and the **optimal separating hyperplane i**s constructed in this new feature space. This method also **resolves the problem** where the training **points are not separable** by a linear decision boundary.



Because SVM uses an **appropriate transformation** the training data points can be made linearly separable in the feature space. The key idea is the **kernel trick**, briefly introduced in the following diagram.



- The choice of kernel functions is highly application-dependent
- it is the most important factor in SVM applications



A common choice for kernels is the Gaussian radial basis function but functions like polynomial or hyperbolic tangent are also used. Usually, in order to find the best kernel search and optimization techniques are employed, such as grid search, random search or Bayesian optimization.

The standard SVM formulation solves only binary classification problems, nevertheless combining several binary classifiers to construct a multi-class classifier is a usual technique. The typical **multi-class classification** models are **One-versus-all** (winner-takes-all strategy) and **One-versus-one** (max-wins voting strategy) and have empirically good performance, a solution that is global and unique and a simple geometric interpretation.

6.4. Liquid State Machines

The Liquid State Machine (LSM) had been proposed as **a computational model** that is more adequate for modeling **computations in cortical microcircuits** than traditional models, such as **Turing machines** or **attractor based models** in dynamical systems (i.e. RNNs). In contrast to these other models, the LSM is a model for **real-time computations on continuous streams of data**, both inputs and outputs of a LSM are streams of data in continuous time.

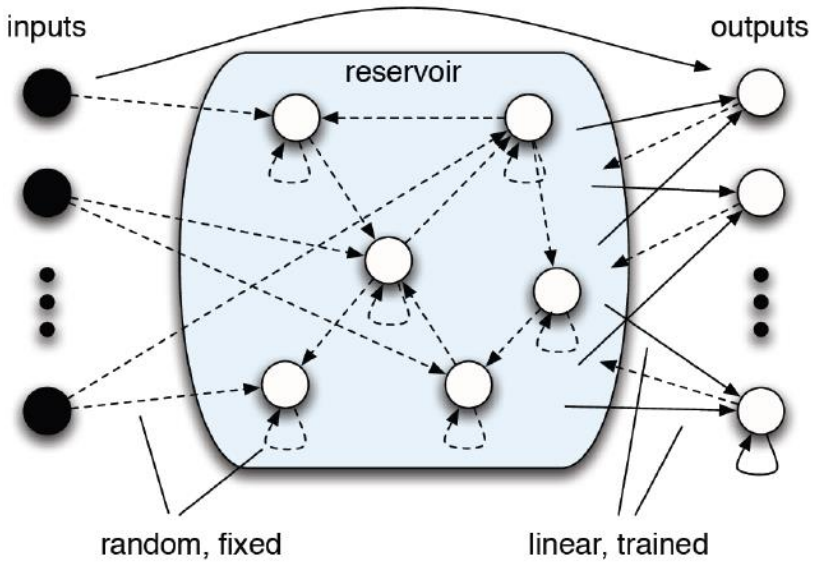
The basic idea of **LSM** is to use **a high dimensional dynamical system** and have the **inputs continuously perturb** it. If the dynamics are sufficiently complex, the LSM should **act as a set of filters** projecting the inputs into a **higher dimensional space**.

The LSM uses the **internal dynamics** of a **recurrent spiking neural network** to carry out computations on its input. The **internal state** serves as input for the so-called **readout function**. The liquid itself does not generate any output; it merely serves as a **'reservoir'** for the inputs. The readout then looks at the liquid state (the response of the liquid to a certain input), and computes the output of the LSM.

Given a **time series of input**, the LSM can **produce a time series** of behaviors as output. To get the desired behaviors, one will have to **adjust the weights** on the links between the core and the output.

The LSM come as an alternative to RNNs (Maass 2001, Jaeger 2001), which are more difficult to train than feed-forward neural networks and infinitesimally small changes to RNN parameters can lead to drastic discontinuous changes in its behavior. Moreover, gradient descent RNN training methods might: have slow convergence (computationally expensive); involve a critical selection of learning parameters (vanishing gradient problem); or fall in local minima.

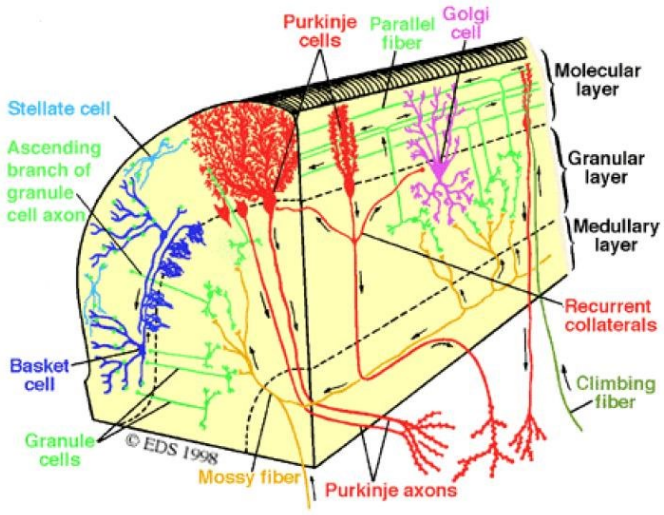
LSMs are an approach to a more **general class** of models called, **reservoir computing**. This approach to computation is represented by two types of models, **Echo state networks** and **Liquid state machines**. The basic working principle is depicted in the following diagram.

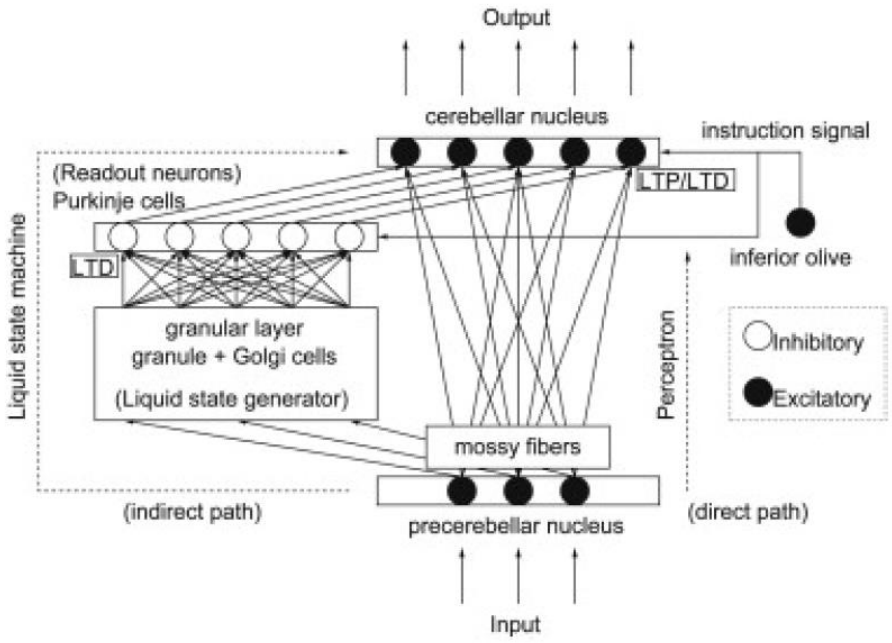


The **recurrently connected nodes** compute a **large variety of nonlinear functions** on the input. Given a **large enough variety** it is possible to obtain linear combinations (using the read out units) to perform arbitrary mathematical operations.

Reservoir computing and LSMs greatly facilitates the practical application of RNNs, such that in many tasks reservoir computing networks outperform classical fully trained RNNs.

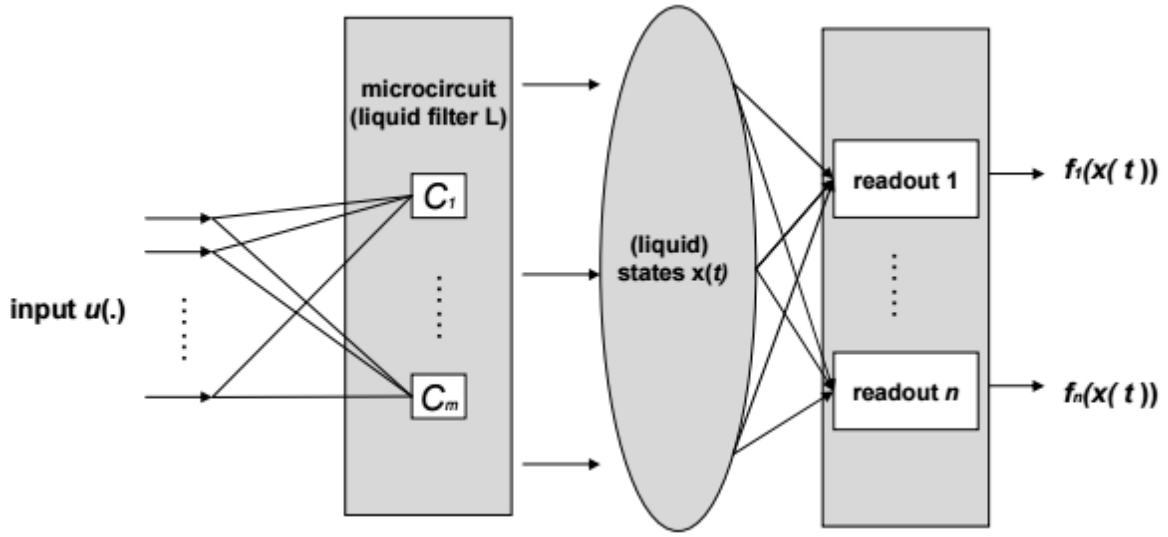
The underlying working principles of the LSM can be tracked back in neurobiological systems, namely looking at the cerebellum as a liquid state machine (Yamazaki and Tanaka 2005).





In this model, the **granular layer** represents the passage of time by generating long sequences of active **spiking populations** (**liquid state**) whereas the **Purkinje cells** stop firing at the timing instructed by climbing fiber signals from the inferior olive (**readout neurons – typically a trained FFNN**).

A **LSM comprises three parts**, an **input layer**, a **large randomly interconnected unit** which has the intermediate states transformed from input, and an **output layer**. As the name of the model hints, they use the microcircuit as a "**liquid filter**" that serves as an **unbiased fading memory** about current and preceding inputs to the circuit. Typically recurrent neural nets that **employ Leaky Integrate and Fire Neurons (LIF)** are used in these machines. The basic structure is depicted in the next diagram:



The liquid filter unit *L* serves like excitable medium core to pre-process the input *u* and transforms the input into liquid states *x*. Then the temporal features extracted are passed to the readout unit through a function *f* that maps the liquid state *x* at time *t* into the output.

Given the previous sections and analysis we performed on various neurally inspired computational architectures, we can enumerate the reasons why LSMs are attractive and an active area of research.

Advantages:

- Efficient in comparison with classical trained RNNs.
- Easier to learn dependencies requiring long-range memory.
- The same network can perform multiple computations on different time-scales.

Disadvantages:

- LSMs don't actually explain how the brain functions. At best they can replicate some parts of brain functionality.
- Inefficient from an implementation point of view in comparison with custom designed circuits.

Contents

Lectures

7. Reinforcement Learning

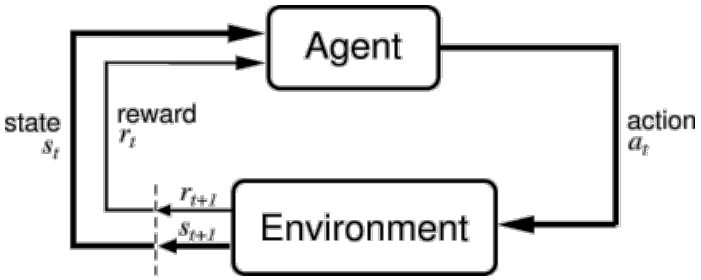
“If an action taken by a learning system is followed by a satisfactory state of affairs, then the tendency of the system to produce that particular action is strengthened or reinforced. Otherwise, the tendency of the system to produce that action is weakened”. This **generic definition** was given in 1991 by R. Sutton, the inventor of Reinforcement Learning (RL), to root such a novel learning mechanism in the framework of the already formalized adaptive optimal control.

Many non-linear control problems today cannot get solved by computers; not because of memory and CPU-time, but because the system designer does not know how to program “the correct things”, for example: inverted pendulum uprising, pole balancing, airplane stabilization and even board games.

RL has been seen as a **derivative of supervised learning** based on **trial-and-error (and reward)**. In contrast to supervised learning, there is **no direct teacher** to provide how much output error a particular action has produced. Instead, the output has been quantified into either **‘positive’ or ‘negative’ reward** corresponding to **closer** to or **further** from the **goal**.

7.1 Introduction to Reinforcement Learning

In the basic formulation RL models allow an agent to actively choose a decision policy based on explorations of the environment.



http://webdocs.cs.ualberta.ca/~sutton/book/ebook/node28.html

The main elements of a RL model are:

1. the **Environment**
2. the **Reward**
3. the **Policy**
4. the **Value function**

In brief, the **policy** shows how to choose a good action for a given state, the **value function** shows how good / valuable a state is, whereas the **reward** shows how much reward does being in a certain state bring. The RL **searches** a **mapping** from **state to action** by **trial-and-error**.

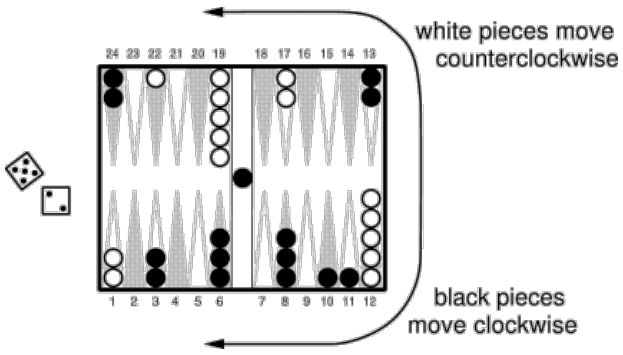
1. Environment

The environment must be **observable** through the **sensory readings** and **actions** must have an **impact** on the environment through the **motor commands**. In defining the environment some assumptions are made. The environment must be: **perfectly observable** (i.e. all states that potentially have an impact on the choice of actions can be observed); **deterministic** (i.e. same actions taken in same state lead to same results in repeated trials). These **constraints** are relaxed in more advanced algorithms, but for the current introduction we will consider them.

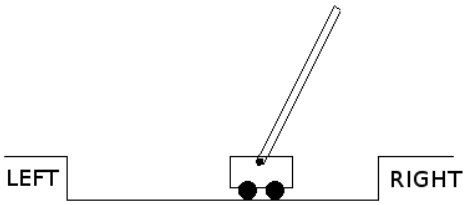
2. Reward

The **reward** *r(s),* or the **reinforcement signal**, depicts a mapping from state to action and is justified by the environment giving reward to the learning system in obvious situations (i.e. after running a board game or when the plane crashes). The RL agent tries to **maximize all expected future reward.** It is the job of the system designer to define a **reward function** correctly. There are at least three types of rewards:

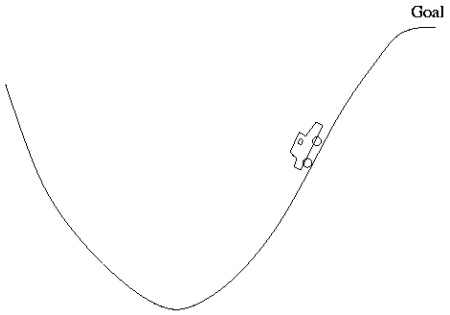
- Pure delayed reward
    - All reward is zero until the final state is reached and the sign of the final reward indicates the success or failure.
    - Examples:
        - Playing backgammon
            - States: configuration of pieces (high-dimensional state space);
            - Reward: a final “win” gives a reward of +1 whereas a final “loss” gives a reward of -1; there is no reward for intermediate actions.



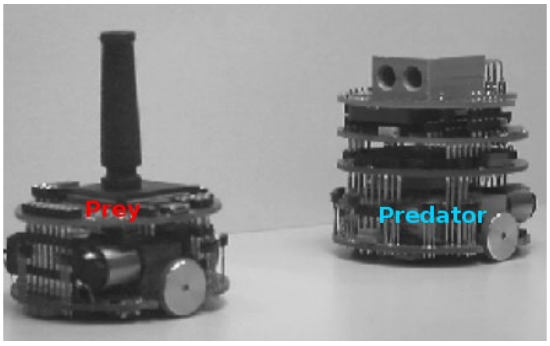
        - Cart balancing
            - RL should keep the pole upright by moving the cart left or right;
            - Reward: final rewards -1 if pole falls or cart touches left/right borders and 0 otherwise.



- Minimum time to goal reward
    - Example:
        - Get a car uphill
            - The car's engine is too weak to get the car up by itself. The RL system needs to find the concept of "momentum" to get up to the goal and minimize the time spent in the valley;
            - State space: position of the car, velocity of the car;
            - Action space: drive forward, drive backward, empty gear;
            - Reward: continuously -1 at every time step and 0 once the car reached the Goal.



- Multi-player games
    - Two or more agents work simultaneously to achieve potentially opposing goals:
        - Predator and prey scenario:
            - One robot is chasing another robot; so one agent tries to minimize the distance between the two, the other robot tries to maximize the distance;



- The predator has to select actions that provide the best reward given the unknown actions of the prey so that it is learning to maximize reward in a "worst case scenario".

3. Policy

The **policy** *π(s)* is responsible to map from state to action to be taken. This is what needs to be learned: how to choose a good action for a given state.

4. Value function

The **value function**, *V(s),* shows how valuable a state is. The value of a state is defined as the **sum of the expected future rewards** when **starting from a state *s* following a policy** *π(s).* An important observation is that, the value function can be represented as a neural network, an equation or as a look-up table.

There are two models for value functions:

- Finite horizon model

$$V(s) = \sum_{t=1}^{n} r\big(s(t)\big), \text{ where } s(t+1) = \pi(s(t))$$

- Infinite horizon model

$$V(s) = \sum_{t=0}^{\infty} \gamma^t r\big(s(t)\big), \text{ where } s(t+1) = \pi\big(s(t)\big), \gamma^t \in]0,1]$$

A good example is the **Markov Decision Processes** (MDP). MDPs assume that the complete state of the world is visible to the agent. This is clearly highly unrealistic (think of a robot in a room with enclosing walls: it cannot see the state of the world outside of the room). **POMDPs** (Partially Observable MDPs) model the information available to the agent by specifying a function from the hidden state to the observables, just as in an HMM. The goal now is to find a mapping from observations (not states) to actions.

How to find an algorithm that finds the best possible value function? Going from *v(s)* to find *π(s)* is simple, we could, for example, use simple hill-climbing and chose the action which maximizes *v(s+1).*

In many real-world cases, agents must deal with **environments that contain non-determinism**. Perhaps the **agent's actions can fail**, or its **sensors can be inaccurate**, or outside forces might change the environment. Deterministic environments are nice, because they let us apply search algorithms such as A* to find an optimal sequence of actions. However, many environments are not deterministic.

As we previously remarked MDPs are a way to model sequential decision making under uncertainty. In this framework one can formulate the total reward from a policy as the sum of the discounted expected utility of each state visited by that policy.

The optimal policy is the policy that maximizes this equation. In this assignment, we will look at three algorithms for discovering this policy. Several methods have been developed to tackle this problem.

Value iteration

The essential idea behind value iteration is that if we knew the true value of each state, our decision would be simple: always choose the action that maximizes expected utility. But we don't initially know a state's true value; we only know its immediate reward. But, for example, a state might have low initial reward but be on the path to a high-reward state.

Assuming we have a look-up table of values v(s) and we sweep through the table to update v_t:

$$V'_t = \max_{\pi(s)}(r(s_{t+1}) + \gamma V(s_{t+1}))$$

or "incrementally"

$$\Delta V'_t = \eta(\max_{\pi(s)}\big(r(s_{t+1}) + \gamma V(s_{t+1})\big) - V(s_t))$$

The **value iteration** algorithm is **guaranteed to find the best control policy** for any system by taking **long term reward** into account, though *r(s)*. But the system designer needs to know the **results of all actions** (i.e. through $\max_{\pi(s)}$) in a given state and it needs **very long time to converge**.

The **pseudo-code for the value iteration** algorithm:

- Initialize V(s) = rand() for all s ∋ target
- Initialize V(s) = target value for all s ∈ target
- Repeat
  - $\Delta = 0$
  - for all s ∈ S in random order
    - $v - V(s)$
    - *V(s)* = $\max_{\pi(s)}\big(r(s_{t+1}) + \gamma V(s_{t+1})\big)$
    - $\Delta = \max(\Delta, |v - V(s)|)$
  - end
- until $\Delta == 0$ *or* $\Delta \leq 0$

Continuous states, Residual Gradient Algorithm

A RL algorithm can be **guaranteed to converge for lookup tables,** yet **unstable** for function-approximation systems that have even a **small amount of generalization**. **Direct algorithms** can be fast but **unstable**, and **residual gradient algorithms** can be **stable** but **slow**. Direct algorithms attempt to make each state match its successors, but ignore the effects of generalization during learning. **Residual gradient algorithms** take into account the **effects**

of generalization, but attempt to make each state match both its successors and its predecessors.

Considering that the value function can be calculated as

$$V(s) = V^*(s) + e(s)$$

where $V^*(s)$ is the unknown "perfect" value of state *s* and *e(s)* the error, instead of a look-up table *V(s)* we can use a function approximator to compute *V(s)*. **Neural networks, as universal function approximators,** are a good candidate for such a task. We can rewrite the previous equation as a **function approximation** problem given the current set of weights w_t with

$$V^*(s_t) = V(s_t, w_t) = NeuralNet(s_t, w_t)$$

and the change in weights

$$\Delta w_t = -\eta(\max_{\pi(s)}\big(r(s_{t+1}) + \gamma V(s_{t+1})\big) - V(s_t))\frac{\partial V(s_t, w_t)}{\partial w_t}$$

where

$$\Delta w_t = \underbrace{-\eta(\max_{\pi(s)}\big(r(s_{t+1}) + \gamma V(s_{t+1})\big) - V(s_t))}_{produces\ an\ error\ for\ current\ V} \quad \underbrace{\frac{\partial V(s_t, w_t)}{\partial w_t}}_{the\ change\ of\ V\ w.r.t.the\ weights\ (see\ backprop)}$$

This allows "**interpolating**" between states and starting using the RL even when not all states have been explored. In this formulation the **learning rate η has to be very small otherwise oscillations** might occur because we are updating the weights at time *t,* but once the update has occurred, the "error" has changed because it depends on *w*. A solution would be to use a different error measure of doing small changes only.

Value iteration works fine, but it **has two weaknesses**: first, it can take a **long time to converge** in some situations, even when the underlying policy is not changing, and second, it's not actually doing what we really need. We actually **don't care what the value of each state is**; that's just a tool to help us find the optimal policy.
So why not just find that policy directly? We can do so by **modifying value iteration to iterate over policies**. We start with a random policy, compute each state's utility given that policy, and then select a new optimal policy. This technique is called **policy iteration.**

7.2 Q-Learning

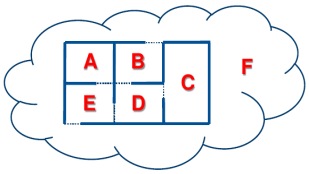
Value Iteration and **Policy Iteration** work well for determining an optimal policy, but they **assume** that the **agent has a great deal of domain knowledge**. Specifically, they assume that **the agent accurately knows the transition function** and the **reward for all states** in the environment. This is considerable amount information to which, in many cases, an agent may not have access to.

In order to handle such cases there is a way to learn this information. In essence, there is a **trade-off between learning time for a priori knowledge**. One way to do this is using **Q-learning**. This RL model is a form of **model-free learning**, meaning that an **agent does not need to have any model** of the environment; it only needs to know **what states exist** and **what actions are possible** in each state.

The core idea is to **map** the tuple **(state, action)** to **reward**. We define a function *Q(s,a)* as the reward for a given action in a state plus all future rewards along the optimal actions:

$$Q(s,a) = r(s,a) + \gamma \max Q(s',a)$$

This formulation allows the agent to "perform" only states s_t and s_{t+1}, not a large number of s_{t+1}.
We introduce the basic algorithm by looking at a simple example, Search and rescue robot planning. Suppose the robot operates in an environment with 5 rooms connected with certain doors, as shown in the following figure. The task for the robot, if placed initially in any room, is to find the best way to reach outside world (F) from that room without knowing the pattern?



1. Set parameter γ and reward matrix R
2. Initialize matrix Q as zero matrix
3. For each episode*:
   a. Select random initial state
   b. Do while not reach goal state:
      b.1. Select one among all possible actions for the current state
      b.2. Using this possible action, consider to go to the next state
      b.3. Get max Q-value of this next state based on all possible actions
      b.4. Compute using update-rule:
      Q(s,a) = r(s,a) + γ*max_Q(s',a')
      b.5. Set the next state as the current state

discount factor
all actions
next state

Note: γ closer to 0 → consider only immediate reward
γ closer to 1 → waiting for greater reward → introduce delay

Episode → one period of exploration until reaching the goal

An extension to Q-Learning for a continuous state space has also been developed. In this case we use a neural network for *Q(s,a)* and the change in *Q* is given by

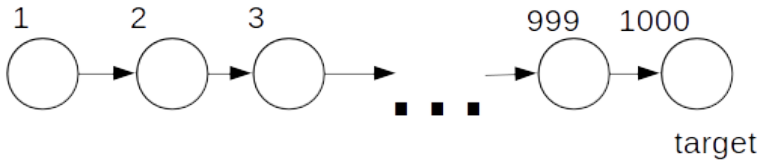
$$\Delta Q(s_t, a_t) = \big(r(s_t, a_t) + \gamma \max Q(s', a)\big) - \ Q(s_t, a_t))$$

and the change in weights

$$\Delta w_t = -\eta \Delta Q(s_t, a_t) \frac{\partial Q(s_t, a_t, w_t)}{\partial w_t}$$

Another RL algorithm used in practice is **Temporal Difference (TD) Learning**, which is a more general approach over Q-Learning. TD-Learning is an approach to learning how to **predict a quantity** that depends on future values of a given signal. It can be **used to learn both the value function and the Q-function**, whereas Q-learning is a specific TD-Learning algorithm used to learn the Q-function.

Assume we have a RL system in which a sequence of actions is deterministic.



In such a system, if we initialize values randomly as before, most updates of values are based on nothing but random values. “True” information is only slowly transported from “right to left”. Here it needs 1000 steps until “true” information arrives at state 1. We would prefer to update the values of a state *s* on something more than just value of *s+1*. If we expand the previous formulation we obtain:

$$Q^1(s_t, a_t) = r(s_t, a_t) + \gamma \max Q(s_{t+1}, a_{t+1})$$

$$Q^2(s_t, a_t) = r(s_t, a_t) + \gamma r(s_{t+1}, a_{t+1}) + \ \gamma^2 \max Q(s_{t+2}, a_{t+2})$$

$$Q^3(s_t, a_t) = r(s_t, a_t) + \gamma r(s_{t+1}, a_{t+1}) + \ \gamma^2 r(s_{t+2}, a_{t+2}) + \ \gamma^3 \max Q(s_{t+3}, a_{t+3})$$

$$\blacksquare \blacksquare \blacksquare$$

$$Q^n(s_t, a_t) = r(s_t, a_t) + \gamma r(s_{t+1}, a_{t+1}) + \cdots + \ \gamma^{n-1} r(s_{t+n-1}, a_{t+n-1}) + \ \gamma^n \max Q(s_{t+n}, a_{t+n})$$

But again, in a real-world system we cannot access future states, especially in a non-Markov system. In robotics, RL systems typically perform some actions and remember past rewards and states, and apply such updates retrospectively.
TD-Learning methods learn their estimates in part on the basis of other estimates. They learn a guess from a guess - they **bootstrap**.

TD-Learning methods are alternatives to Monte Carlo methods for solving the **prediction problem**. In both cases, the extension to the control problem is via the idea of **generalized policy iteration (GPI)** that we abstracted from **dynamic programming**. This is the idea that **policy and value functions** should **interact** in such a way that they both **move toward their optimal** values.

TD-Learning methods are naturally implemented in an **on-line**, fully **incremental fashion** compared to Monte Carlo methods. With **Monte Carlo** methods one must **wait until the end of an episode**, because only then is the return known, whereas with **TD methods** one need **wait only one time step**.

Contents

Lectures

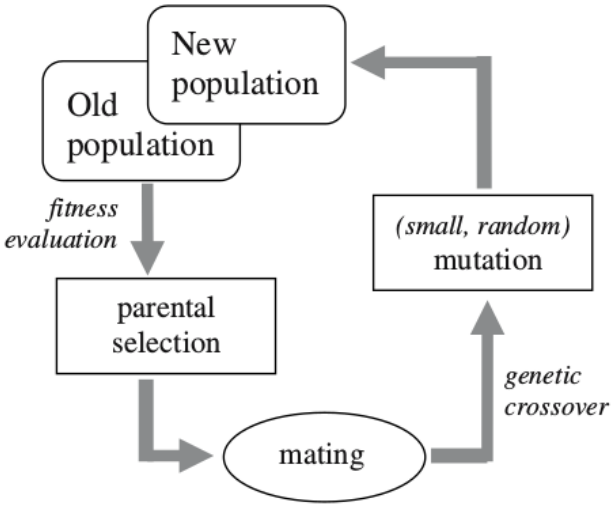
8. Evolutionary programming

Evolutionary programming is a method for simulating evolution that has been investigated for over 30 years. The inspiration for evolutionary methods goes back to the **1850s** and the work of **Charles Darwin** with his **theories of evolution**, **natural selection** and **'survival-of-the-fittest'**. This chapter tries to offer an introduction to **evolutionary programming**, and indicates its relationship to other methods of evolutionary computation, specifically genetic algorithms and evolution strategies.

The efforts within the field of evolutionary computation have generally followed three main lines of investigation: (1) **genetic algorithms**, (2) **evolution strategies**, or (3) **evolutionary programming**. These techniques are broadly similar and rely on a **population of competing solutions** which are subjected to **random alterations and compete** to be **retained as parents** of **successive reproduction epochs**.

The **differences** between the methods concern the **level in the hierarchy of evolution** being modeled: the **chromosome**, the **individual**, or the **species**.

Looking at the **chromosome level, Genetic Algorithms** model evolution as a succession of changing gene frequencies, where **competing solutions** are encoded as chromosomes in genes. The space of possible solutions is explored by **applying transformations to the solutions** as observed in the chromosomes of living organisms (i.e. cross-over, mutation, mating). In contrast, **evolutionary programming** models evolution as a **process of adaptive behavior** of species, rather than adaptive genetic processing.



8.1. Introduction to evolutionary computing

What is Evolutionary computation? In an attempt to answer this question, a generic response has been given: An **abstraction** from the **theory of biological evolution** that is used to create **optimization procedures** or **methodologies**, implemented in **computer software** that are used **to solve problems**. In evolutionary computation

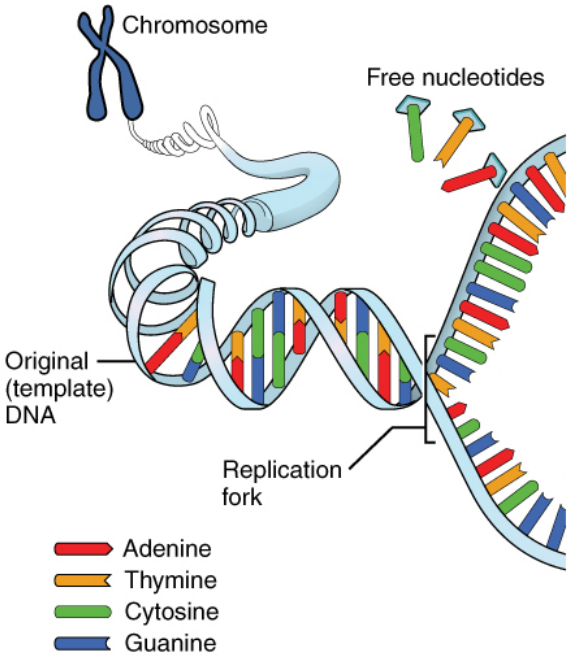
populations of solutions are evolved to exploit a continuous range of solution while at the same time maintaining a strong behavioral connection between offspring and their parents.

Evolutionary programming emerged as an **alternative approach to artificial intelligence**. Rather than **emulating human neural computation** or human **behaviors**, **evolutionary programming** was modeled as a **process that generates organisms** of **increasing intellect** over time. **Intelligence** was defined, in this context, as the **ability** of an organism to **achieve goals** in a **range of environments** and **evolution is seen as optimization.**

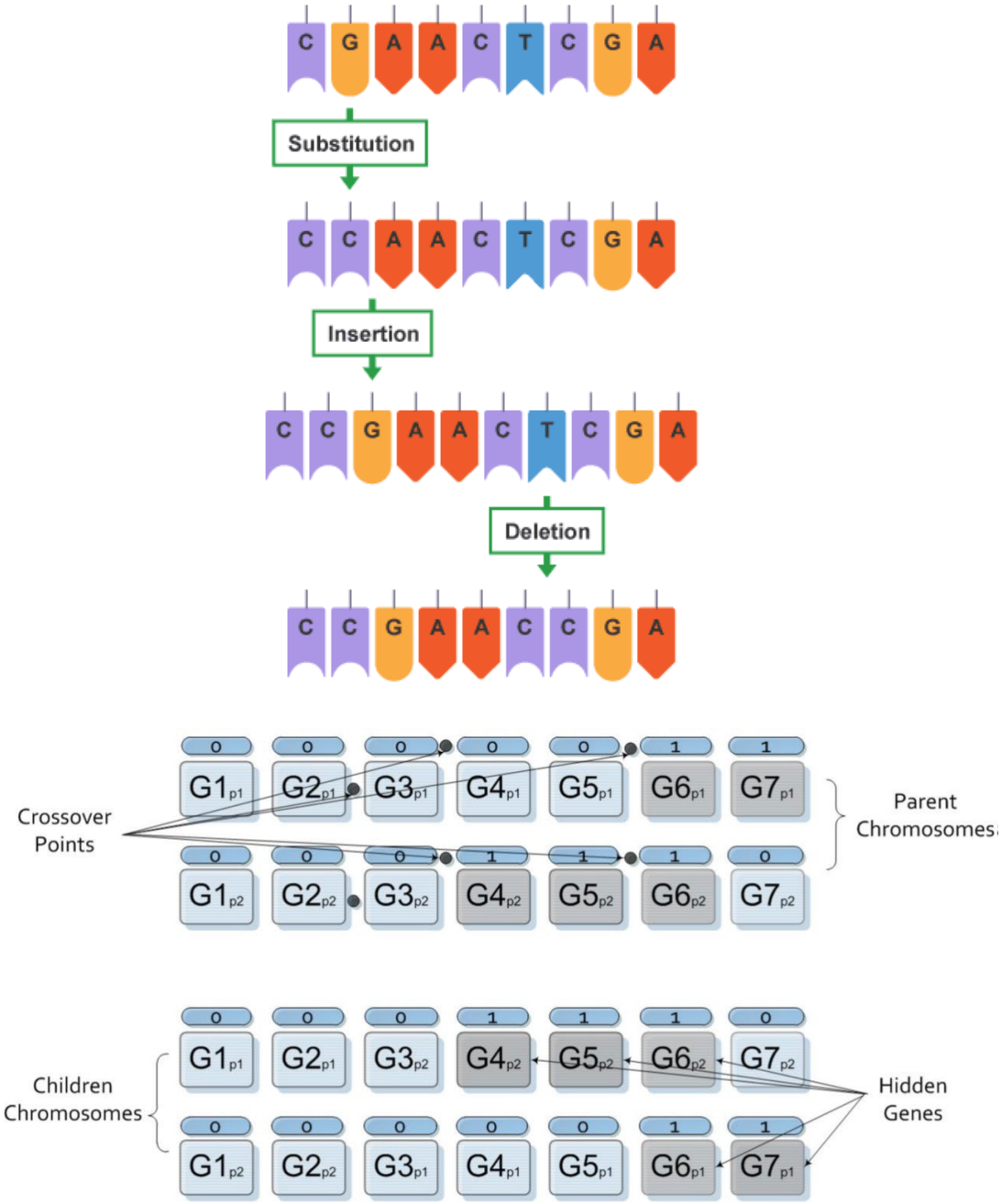
8.2 Genetic Algorithms

Genetic algorithms were formally introduced in the 1970s by John Holland at University of Michigan. Advances in computing hardware have made them attractive for various **types of optimization problems**. In particular, genetic algorithms work very well on **mixed (continuous and discrete), combinatorial problems**.

In the basic formulation, a **gene** is a part of the DNA sequence that **encodes information**. Humans roughly have 20500 genes. The DNA sequence is composed of double-helix of complementary nucleotides: Adenine (A), Cytosine (C), Guanine (G), Thymine (T) and Uracil (U). In normal spiral DNA the bases form pairs between the two strands: A with T and C with G.



The two individual strings can get completed by free nucleotides as, in principle, all information is still available. Bur some **modifications** can occur during reproduction **processes (transformations)** such as: **mutations (i.e. substitution**, **insertion**, **deletion)** or **cross-over**.



Such processes create "**modified blueprints**" of a biological system, which create a modified organism when transcribed (i.e. read out and build in software terms). Through "**survival of the fittest**" only those individuals that have **the largest success survive**, yet taking into account that this is not a goal-directed selection.

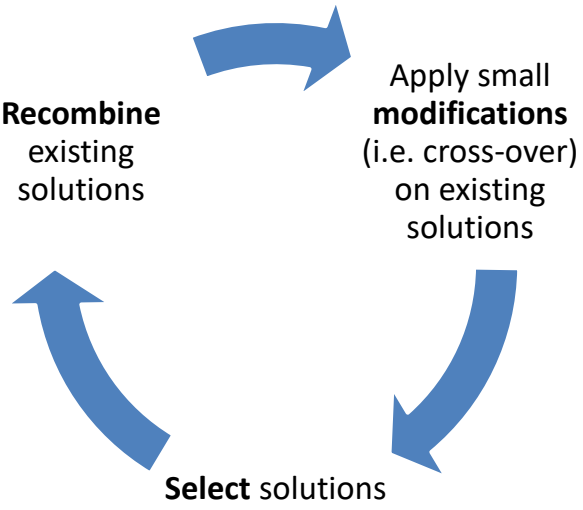
How can we use such processes in technology?

Searching a large state-space or **n-dimensional surface**, GAs may offer significant **benefits over more typical search of optimization techniques**, such as linear programming, heuristic, depth-first, breath-first, but they tend to be **computationally expensive**.

In another perspective, GAs are **adaptive heuristic search algorithms** based on the **evolutionary ideas** of **natural selection** and **genetics**. They represent an **intelligent exploitation of a random search** used to solve optimization problems.

Although **randomized**, GAs **are by no means random**, instead they **exploit historical information** to **direct the search** into the region of **better performance** within the **search space**.

We can see such an algorithm as a **3-step iterative process** accounting for a **stochastic exploration**:



GAs are **less susceptible to settle in local optima** than **gradient search methods**. Moreover, due to the fact that they can **optimize nonlinear, discontinuous functions**, there is **no need to formulate them** only for **differentiable functions**.

Although there are many variations of GAs the core principled algorithm is introduced in the following section.

- Initialize all possible solutions x_i in a population P
- Evaluate $f(x_i)$ and select "good" x_i in the metric of f
- Repeat

- for i=0 to k // k = number of children to produce
  - select $P_{1/2}$ parents out of P
  - generate x_i such through recombination of $P_{1/2}$ (**cross-over**)
  - **mutate** x_i
  - evaluate $f(x_i)$
  - add x_i to a new population P'
- end
- select new P out of P'
- forget P'

❖ until fitness "good enough"

In the next section we look at the **terminology** and **specific requirements** to design and implement a GA.

Encoding

A chromosome should in some way contain information about solution that it represents. A typical way of encoding is a binary string. Each **chromosome** is represented by a **binary string**. Each **bit (gene)** in the string can represent some **characteristics of the solution**. The **encoding depends** mainly **on the solved problem**. For example, one can encode directly integer or real numbers; sometimes it is useful to encode some permutations. In an ideal case **small changes in the representation** should result in **small changes in the state**, and vice versa. This is problematic in binary encoding but not in Gray encoding.

Decimal	Gray Code	Binary
0	0000	0000
1	0001	0001
2	0011	0010
3	0010	0011
4	0110	0100
5	0111	0101
6	0101	0110
7	0100	0111
8	1100	1000
9	1101	1001
10	1111	1010
11	1110	1011
12	1010	1100
13	1011	1101
14	1001	1110
15	1000	1111

Initialization

The **population size** depends on the nature of the problem. Typically, the initial population is **generated randomly**, allowing the span over the **entire search space**. To accelerate computation, sometimes, solutions may be "**seeded**" in areas where **optimal solutions are likely** to be found. The size of the initial population determines the **computational complexity** and the **exploration ability**.

Fitness function

It's necessary to be able to **evaluate how "good" a potential solution is** relative to other potential solutions. The **fitness function** is responsible for performing this **evaluation** and **returning**, typically, a positive integer number, or fitness value, that reflects how optimal the solution is. This is the most important design aspect of a GA.

Selection criteria

An important question is how to **select** parents for crossover. This can be done in many ways, but the main idea is to select the **better parents** in the hope that the **better parents** will **produce better offspring**. In GAs **elitism** is often used, but other methods like **proportional selection, tournament selection, and rank-based selection** are used. This means, that at least **one of a generation's best solution** is copied without changes to a new population, so the **best solution can survive** to the succeeding generation.

Typically we can look at this phenomenon from the point of view of a **selective pressure,** or how long will it need for the “best” organism to take over? We can differentiate a **low pressure** scenario, with **very long convergence time** but offering **good solutions** and a **high pressure** scenario, with **very short convergence** time but offering a **non-optimal solution**.

Reproduction / mutation operators

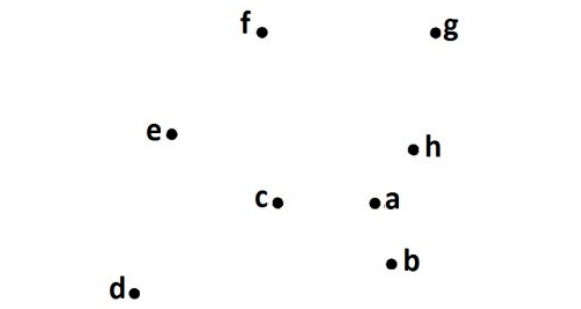
After selection an encoding, we can proceed to crossover operation. **Crossover operates on selected genes** from parent **chromosomes** and creates **new offspring**. The simplest way how to do that is to **choose randomly some crossover point** (s) and copy everything before this point from the first parent and then copy everything after the crossover point from the other parent.

After a crossover is performed, **mutation** takes place. **Mutation** is intended to **prevent falling of all solutions** in the population into a **local optimum** of the solved problem. Mutation operation randomly changes the offspring resulted from crossover.

Sample applications

Travelling Salesman Problem

Many problems do not require the optimization of a series of real valued parameters, but the discovery of an ideal ordered list.

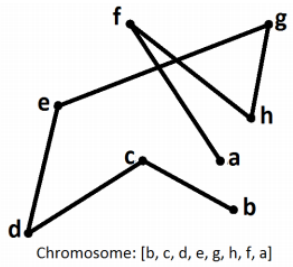


A TSP. What is the shortest tour that connects all eight cities?

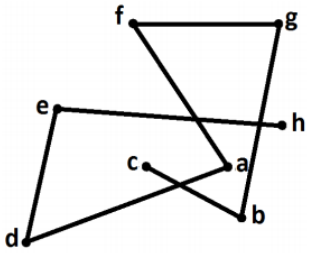
City	a	b	c	d	e	f	g	h
a	0	17	27	73	61	57	51	23
b	17	0	37	73	72	74	66	40
c	27	37	0	48	35	49	65	50
d	73	73	48	0	47	82	113	95
e	61	72	35	47	0	38	80	78
f	57	74	49	82	38	0	48	65
g	51	66	65	113	80	48	0	40
h	23	40	50	95	78	65	40	0

Distances in km between the cities .

We cannot simply mutate or crossover the chromosome.



Chromosome: [b, c, d, e, g, h, f, a]

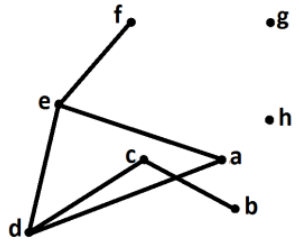


Chromosome: [c, b, g, f, a, d, e, h]

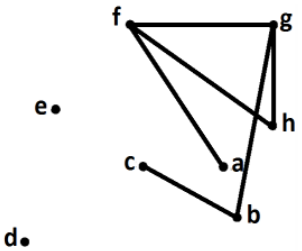
crossover point

[b, c, d, e, | g, h, f, a]

[c, b, g, f, | a, d, e, h]

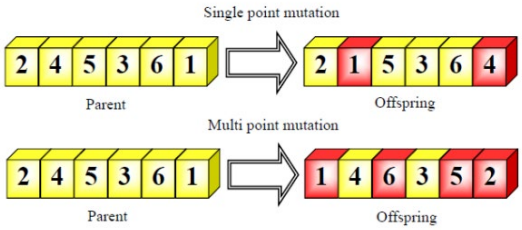


Chromosome: [b, c, d, e, a, d, e, f]



Chromosome: [c, b, g, f, g, h, f, a]

One solution would be literal permutation encoding with reorder mutation, implemented as shown in the following diagram.



Another solution is partially matched crossover (PMX) described as:

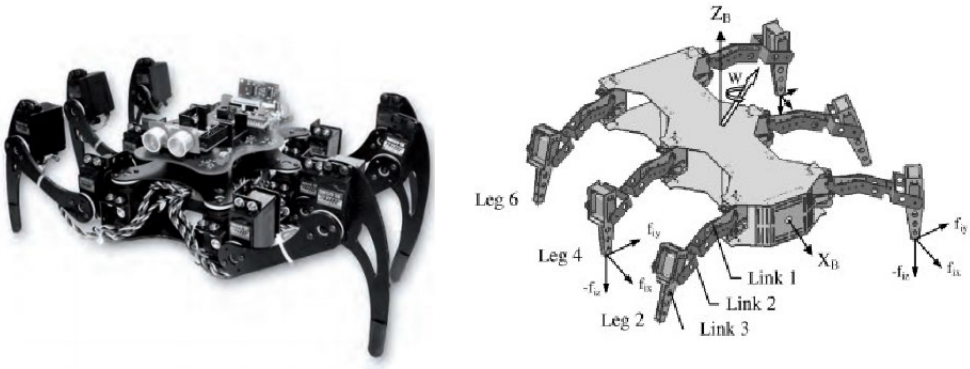
tour 1 = b c / d e g / h f a
tour 2 = c b / g f a / d e h

}

tour 1' = b c / g f a / h f a
tour 2' = c b / d e g / d e h

Note that cities that are visited twice in one tour are swapped with cities that are visited twice in the other tour. Only one representative (the one not in the matching section) of such cities is swapped. Thus **tour 1" = b c g f a h e d** and **tour 2" = c b d e g a f h.**

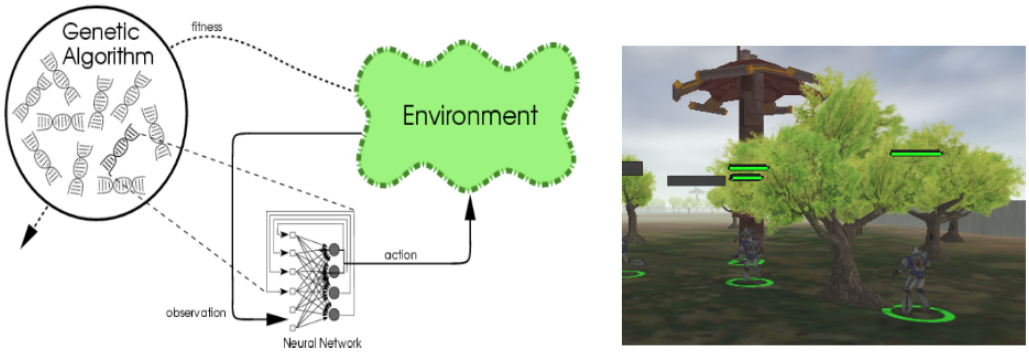
Robot walking learning



Sources: http://www.ti.bfh.ch/uploads/media/2009_59_1_genetic_algorithm.pdf
http://www.youtube.com/watch?v=KHV7fWvnn_0

Evolving computer game players

NERO - Neuro Evolving Robotic Operatives



Sources: http://nerogame.org/
https://code.google.com/p/opennero/

Contents

Lectures

9. Fuzzy Inference Systems

Humans are unconsciously able to address **complex**, **ambiguous**, and **uncertain problems** thanks to thinking. The **thought process** is possible because **humans** do not need the complete description of the problem since they have the capacity to **reason approximately.** Lotfi **Zadeh** proposed and developed in 1965 a theory of such approximate reasoning as an approach to **modeling uncertainty.**

This new theory provided “a **logical system** which aims at a formalization of approximate reasoning. In this case it is an **extension of many-valued logic**. However the agenda of this theory, **Fuzzy Logic (FL)** is different from that of the traditional many-valued logic. Such key concepts in FL as the concept of **linguistic variable**, **fuzzy if-then rule**, fuzzy **quantification** and **defuzzification**, **inference** and **interpolative reasoning**, among others, are not addressed in traditional systems.”

FL can be conceptualized as a **generalization of classical logic.** Modern fuzzy logic was developed to model those problems in which **imprecise data** must be used or in which the **rules of inference** are formulated in a **very general** way making use of **diffuse categories**.

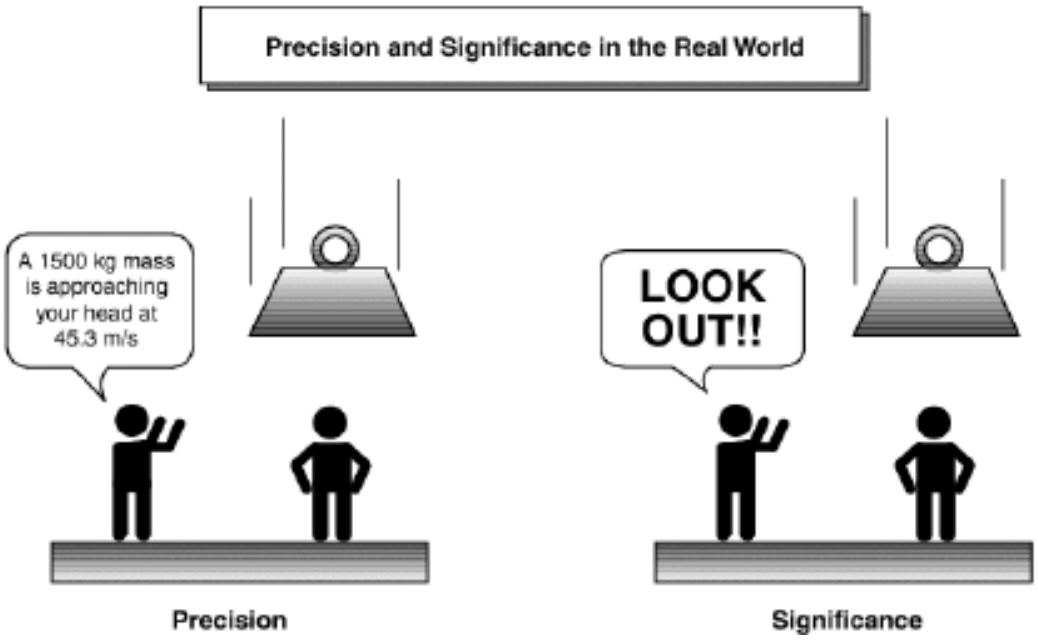
9.1 Introduction to Fuzzy Logic

In this section we will start with an intuitive introduction to fuzzy logic. In our everyday language we use a great deal of **vagueness and imprecision**, that can also be called **fuzziness**. We are concerned with how we can represent and manipulate inferences with this kind of information. Fuzzy sets provide a way that is very similar to the human

reasoning system. But how does human perception work?

If one asks “How comfortable is this room, is it warm or cold?”, most likely a human will answer “It‘s quite cold here”, “Fairly warm”, or “It is too hot for me”. So, the characteristics of human's answer will be: **imprecise** / **vague**, typically involving **modifier/hedge of linguistic terms** (quite, fairly, too, very, etc.), and implies **uncertainty**.

We all use **vague information and imprecision** to solve problems. Hence, our computational methods should be able to represent and manipulate fuzzy and statistical uncertainties. But how important is it to be exactly right when a rough answer will do? It pays off trading between significance and precision - something that humans have been managing for a very long time!



The imprecision in fuzzy models is generally quite high. However, when precision is apparent, fuzzy systems are less efficient than more precise algorithms in providing us with the best understanding of the system. In the following examples, we explain how many industries have taken advantage of the fuzzy theory.

The **Sendai Subway** 1000 series use fuzzy logic to control its speed, and this system accounts for the relative smoothness of the starts/stops when compared to other trains, and is 10% more energy efficient than human-controlled acceleration.

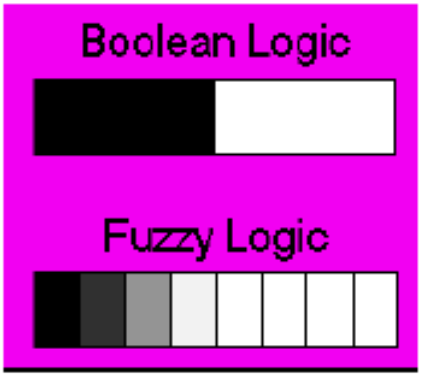
Canon developed an autofocusing camera (**EOS-5D**) using fuzzy control system.
It tracks the rate of change of lens movement during focusing, and controls its speed to prevent overshoot.

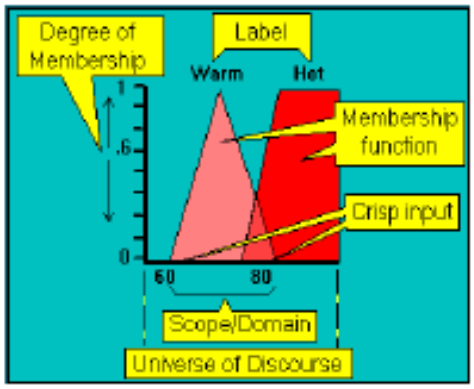
Samsung Fuzzy Logic washing machines (**WA78K1**) monitor varying conditions inside the machine and, accordingly, fuzzy logic controls the washing process, water intake, water temperature, wash time, rinse performance, and spin speed.

Audi TT DSG / S-Tronic gearbox with fuzzy logic automatic gear selection behavior.
Gear choice can be inferred from sensory readings which take into account the human factor (e.g. driving style)

Mathematical foundations of fuzzy logic rest in **fuzzy set theory**, which can be seen as a **generalization of classical set (crisp) theory**. Fuzziness is a **language concept**; its main strength is its **vagueness using symbols** and defining them.

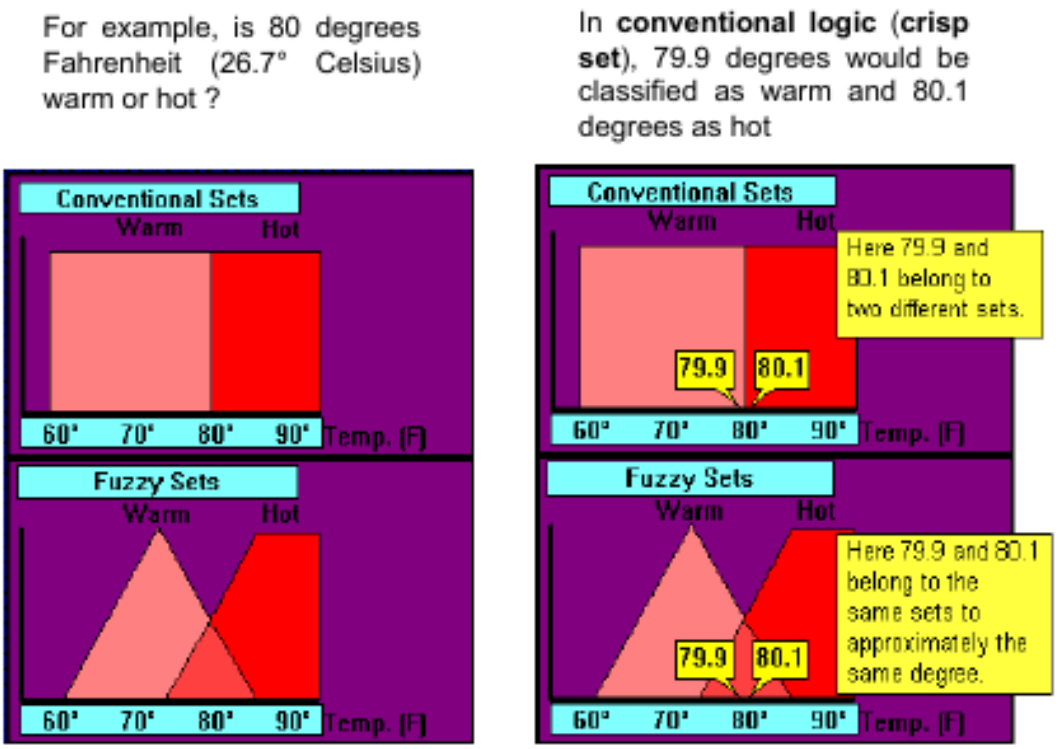
In 1965 Prof. Lotfi A. Zadeh introduced fuzzy sets, where many **degrees of membership** are allowed, and indicated with a number between 0 and 1. The point of departure for fuzzy sets is simply the generalization of the valuation set from the pair of numbers {0,1} to all the numbers in [0,1] as depicted in the following diagram.





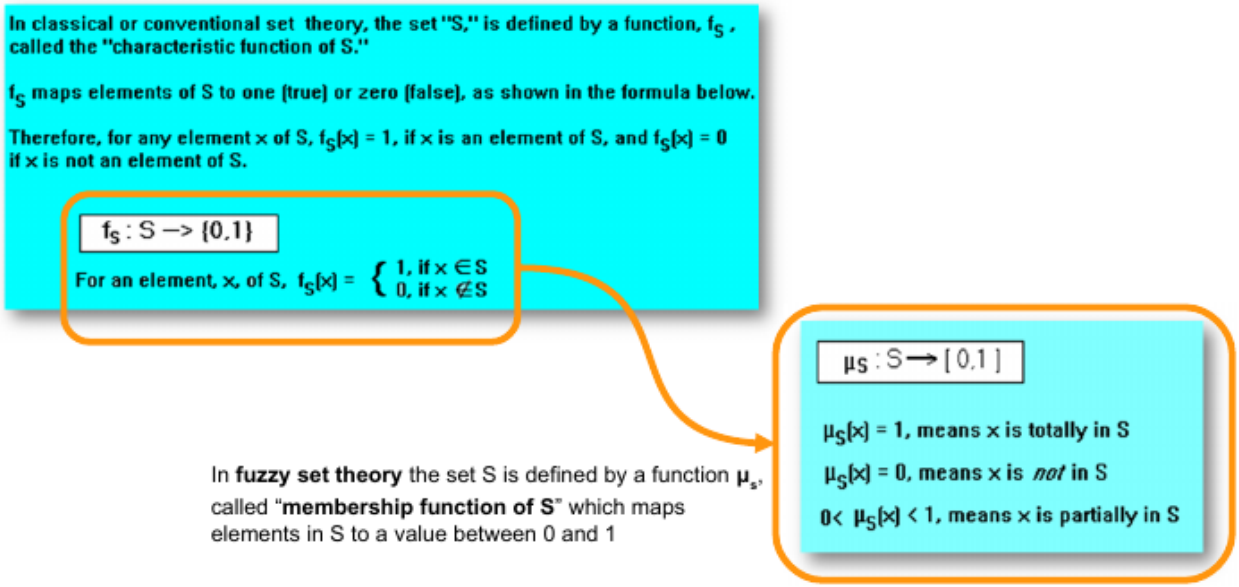
This is called a **membership function** and describes fuzzy sets. More precisely, membership functions are **mathematical tools for indicating flexible membership** to a set (**fuzzification**), modeling and quantifying the meaning of symbols. They can represent a **subjective notion** of a vague class, such as chairs in a room, size of

people, and performance among others. In a typical example one, of temperature measurement, can easily understand the generalization capabilities of fuzzy logic and the modeling and description of crisp physical quantities.



Fuzzy Logic formalism

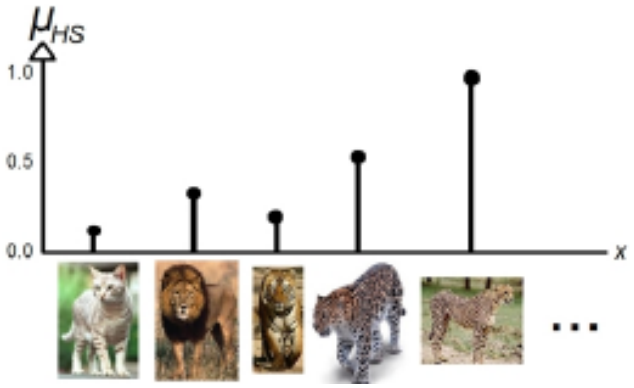
Formally, the core idea behind fuzzy sets and fuzzy representation of crisp quantities can be synthetically depicted in the following diagram.



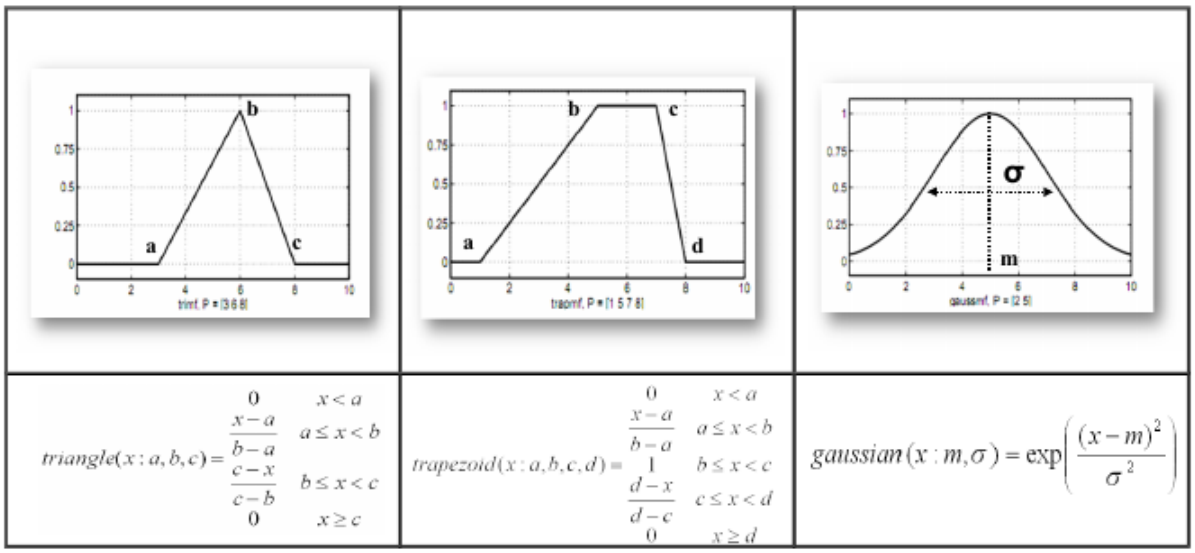
A **fuzzy set** A, corresponding to a **universe of discourse** X, can be represented by an ordered **set of pairs**:

$$A = \{(x, \mu_A(x)) \mid x \in X\}$$

A fuzzy set may be discrete or continuous, for example: Universe of discourse: cat species families, X = {“cat”, “lion”, “tiger”, “leopard”, “cheetah”} and the associated fuzzy set “HS” for animals with “high-speed”, HS = {(x, $\mu_{HS}(x)$) | x ϵ X} = {(cat, 0.1), (lion, 0.3), (tiger, 0.2), (leopard, 0.5), (cheetah, 0.9)}



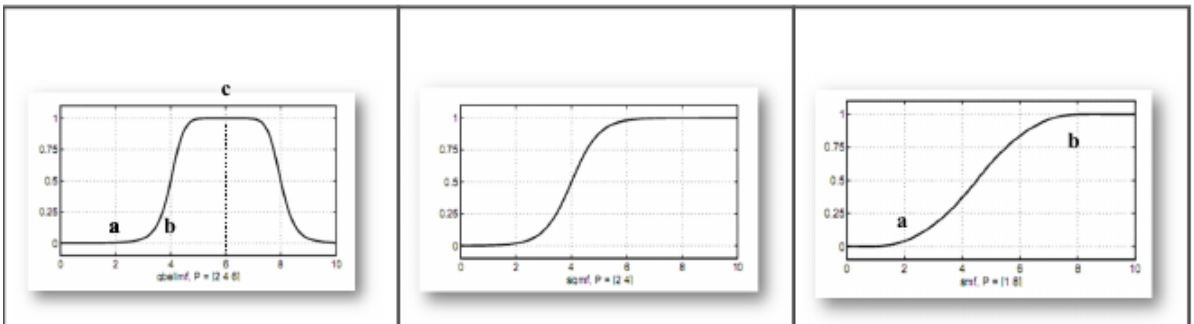
Membership functions give **numerical meaning to a fuzzy set** by mapping crisp inputs from a specified domain to membership degrees ranging [0,1] (**fuzzification**). They can have known analytical formulations: triangular, trapezoidal, Gaussian etc.



$$triangle(x:a,b,c)=\begin{cases}0 & x<a\\ \frac{x-a}{b-a} & a\le x<b\\ \frac{c-x}{c-b} & b\le x<c\\ 0 & x\ge c\end{cases}$$

$$trapezoid(x:a,b,c,d)=\begin{cases}0 & x<a\\ \frac{x-a}{b-a} & a\le x<b\\ 1 & b\le x<c\\ \frac{d-x}{d-c} & c\le x<d\\ 0 & x\ge d\end{cases}$$

$$gaussian(x:m,\sigma)=\exp\left(\frac{(x-m)^2}{\sigma^2}\right)$$



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

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

$$S(x:a,b)=\begin{cases}0 & x<a\\ 2\left(\frac{x-a}{b-a}\right)^2 & a\le x<\frac{a+b}{2}\\ -2\left(\frac{x-b}{b-a}\right)^2 & \frac{a+b}{2}\le x<b\\ 1 & x\ge b\end{cases}$$

The **support** of a fuzzy set A is the crisp set that contains all the elements of X that have nonzero membership degrees in A,

$$supp(A) = \{ x \in X \mid \mu_A(x) > 0 \}$$

The **boundary** is the crisp set that contains all the elements of X that have the membership degrees of $0 < \mu_A(x) < 1$ in A,

$$bnd(A) = \{ x \in X \mid 0 < \mu_A(x) < 1 \}$$

The **core** of a normal fuzzy set A is the crisp set that contains all the elements of X that have the membership grades of 1 in A,

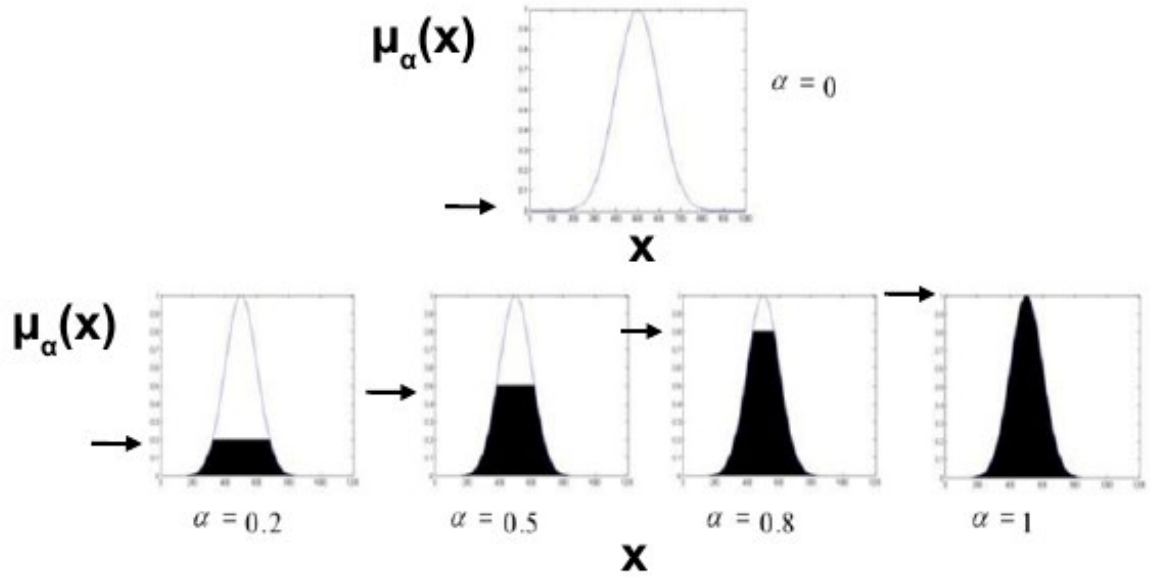
$$core(A) = \{ x \in X \mid \mu_A(x) = 1 \}$$

If the support of a normal fuzzy set consists of a single element x0 of X, which has the property:

supp(A) = core(A) = {x0} this set is called a **singleton** (see cat HS example).

Alpha cut set A_α is a crisp set of a fuzzy set A, where

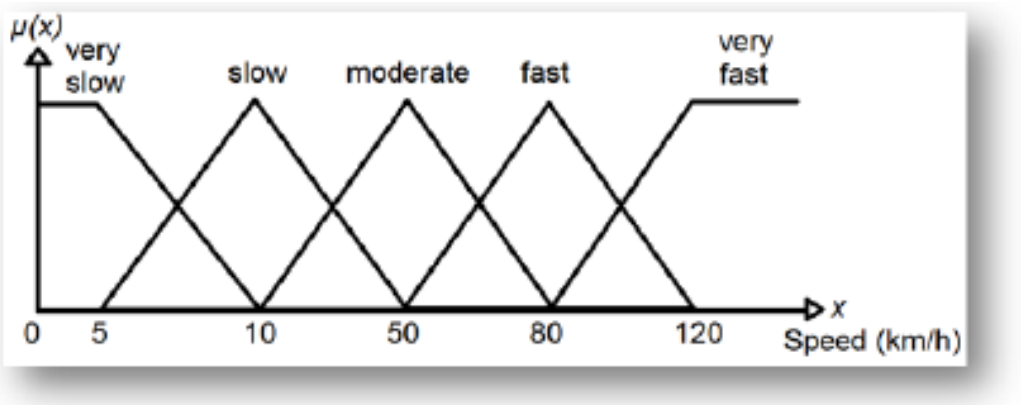
$$A_\alpha = \{ x \in X \mid \mu_A(x) \geq \alpha \}$$



Strong alpha cut set A_α is a crisp set of a fuzzy set A, where

$$A_\alpha = \{x \in X \mid \mu_A(x) > \alpha \}$$

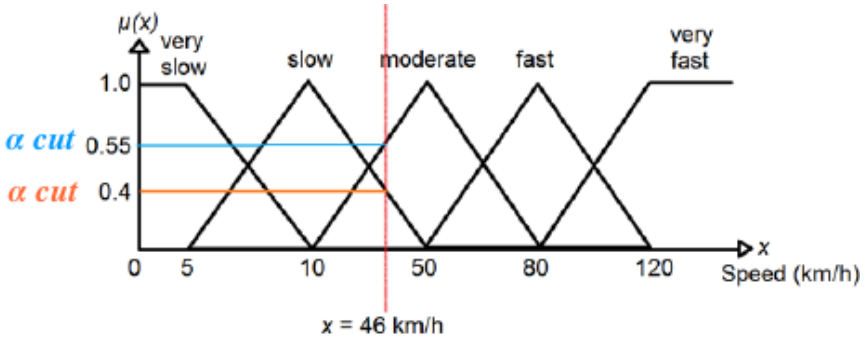
Linguistic variables have an identifier for a membership function which define the partitions (partitioning) of the universe of discourse. This is an application dependent setting. For example, a variable “speed”, which is going to be used as an input for a fuzzy control system, might be defined as: S = { very slow, slow, moderate, fast, very fast } and will partition the universe of discourse as shown in the following figure.



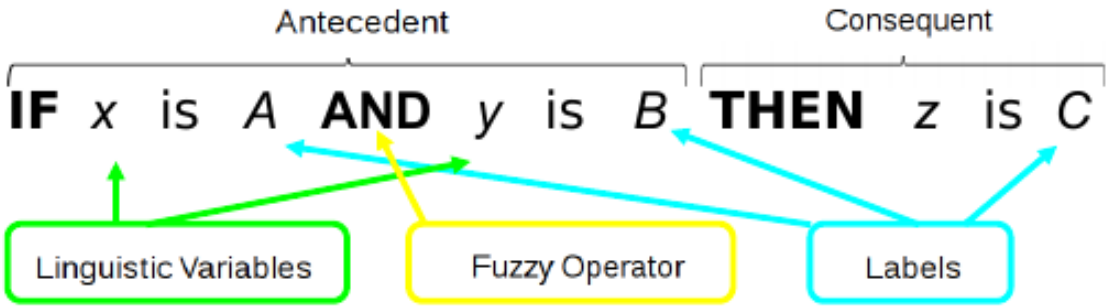
Linguistic variables are used in the **fuzzification** process to assign a membership degree for a certain crisp value for each label of a linguistic variable. In our previous example of speed partitioning:

Fuzzyfication

$\mu_{very\ slow}(x=46) = 0.0$
$\mu_{slow}(x=46) = 0.40$
$\mu_{moderate}(x=46) = 0.55$
$\mu_{fast}(x=46) = 0.0$
$\mu_{very\ fast}(x=46) = 0.0$



In order to **operate on the fuzzy linguistic variables** a fuzzy inference system uses **if-then rules**. In the most general form of fuzzy rule is using **first order logic** and can aggregate multiple variables and labels that describe physical quantities, as shown in the next figure.



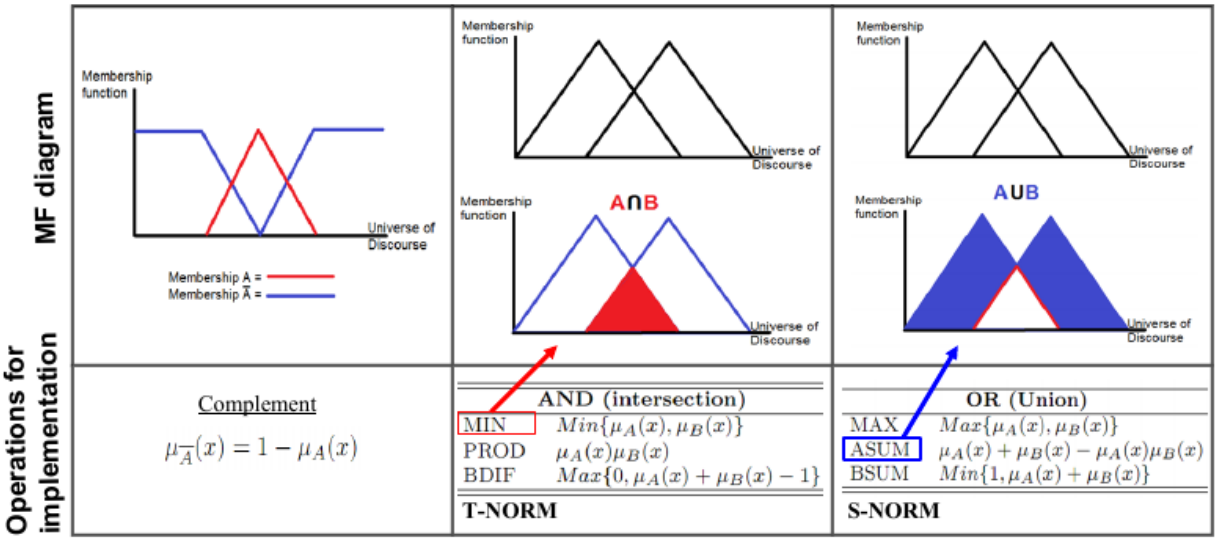
In general, if there are several **input variables x_1, x_2, …, x_n** with several logical connections (i.e. **AND, OR**), the fuzzy if-then clauses can be written as:

IF x_1 is A_{r1} **AND** x_2 is A_{r2} **OR** x_3 is A_{r3}... **AND** x_n is A_{rn} **THEN** y is B_r

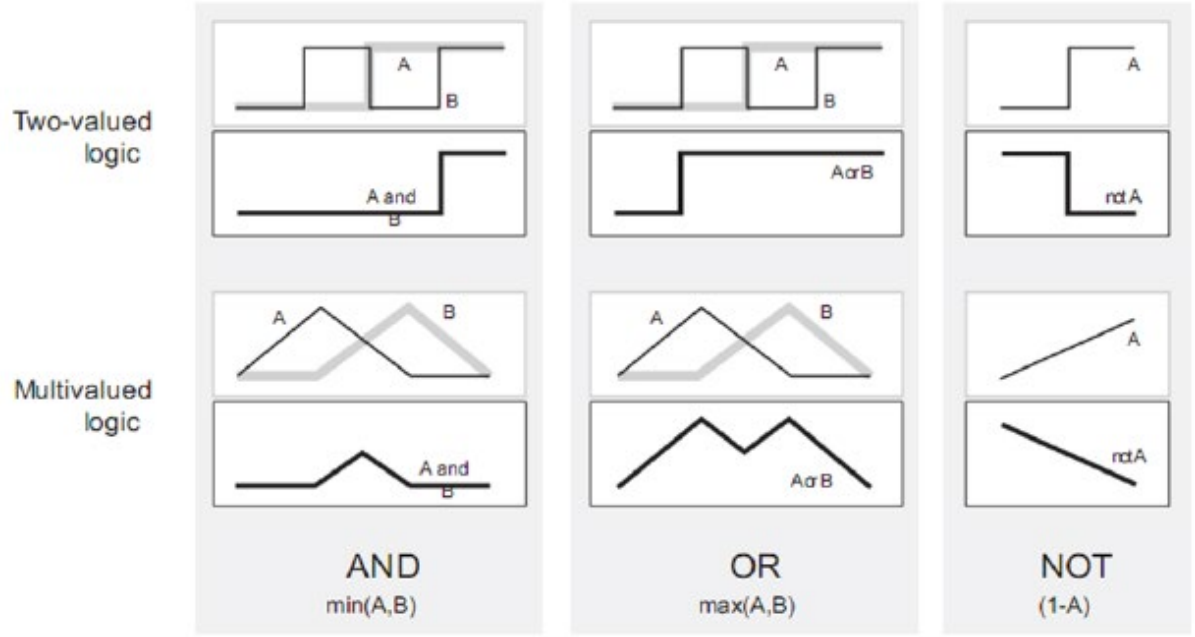
where $r = 1...m$ is the rule number

The typical operations with crisp sets and Boolean logic can be extended also to fuzzy sets by choosing an appropriate operator. There are multiple possibilities to implement

typical conjunction and disjunction operators used to link variables in the premise (i.e. antecedent) or the conclusions (i.e. consequent).

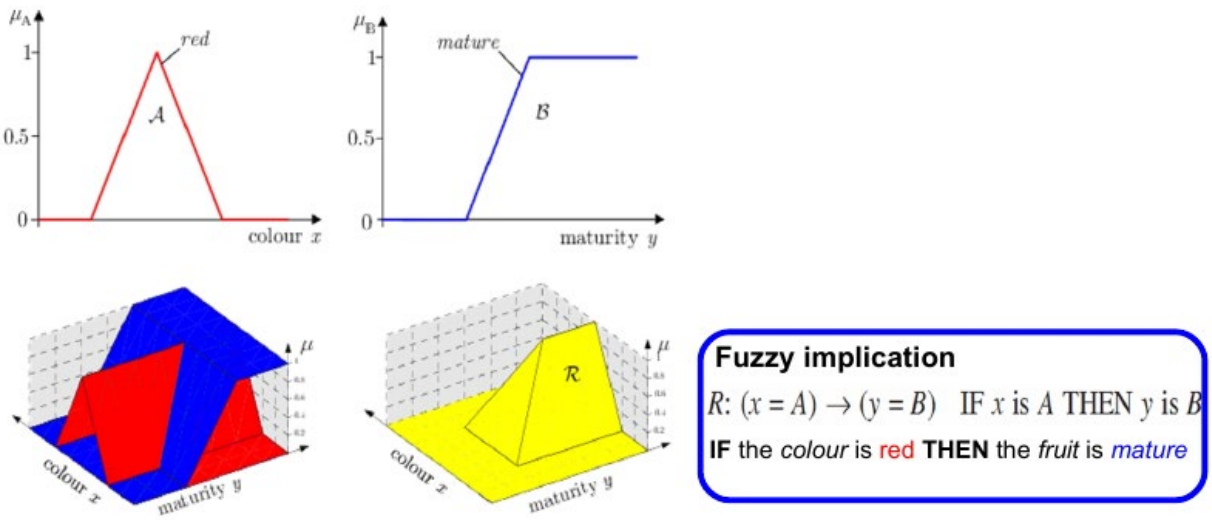


As previously mentioned fuzzy logic extends classical, bivalent logic, and its operators have a different effect, as shown in the next comparative diagram.



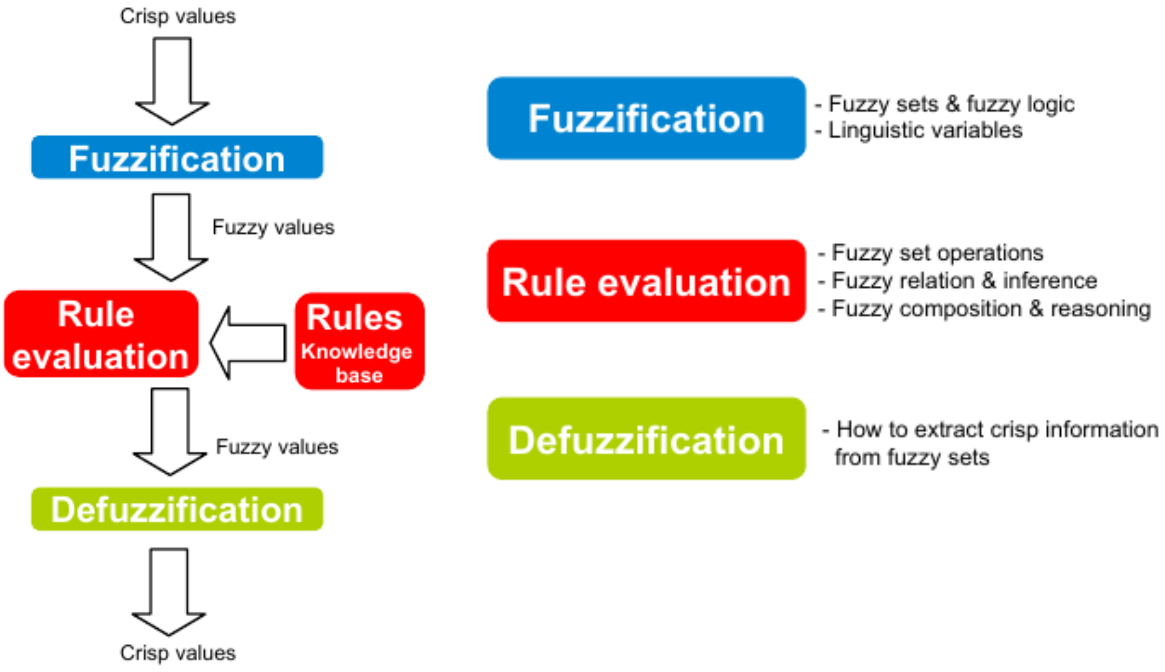
The basic formulation of fuzzy if-then rules assumes connecting the antecedent and the consequent using an operation called fuzzy implication or relations. Such operations are used to model **dependencies**, **correlations** or **connections** between **variables**, quantities or attributes. Moreover, they allow a **generalization** of the definition of fuzzy set from **2-D space to 3-D space**, describing the "degree of association" of the elements, basically a Cartesian product of two fuzzy sets.

In a basic example, let's consider the relationship between the color of a fruit, x, and the grade of maturity, y and the effect of the implication.



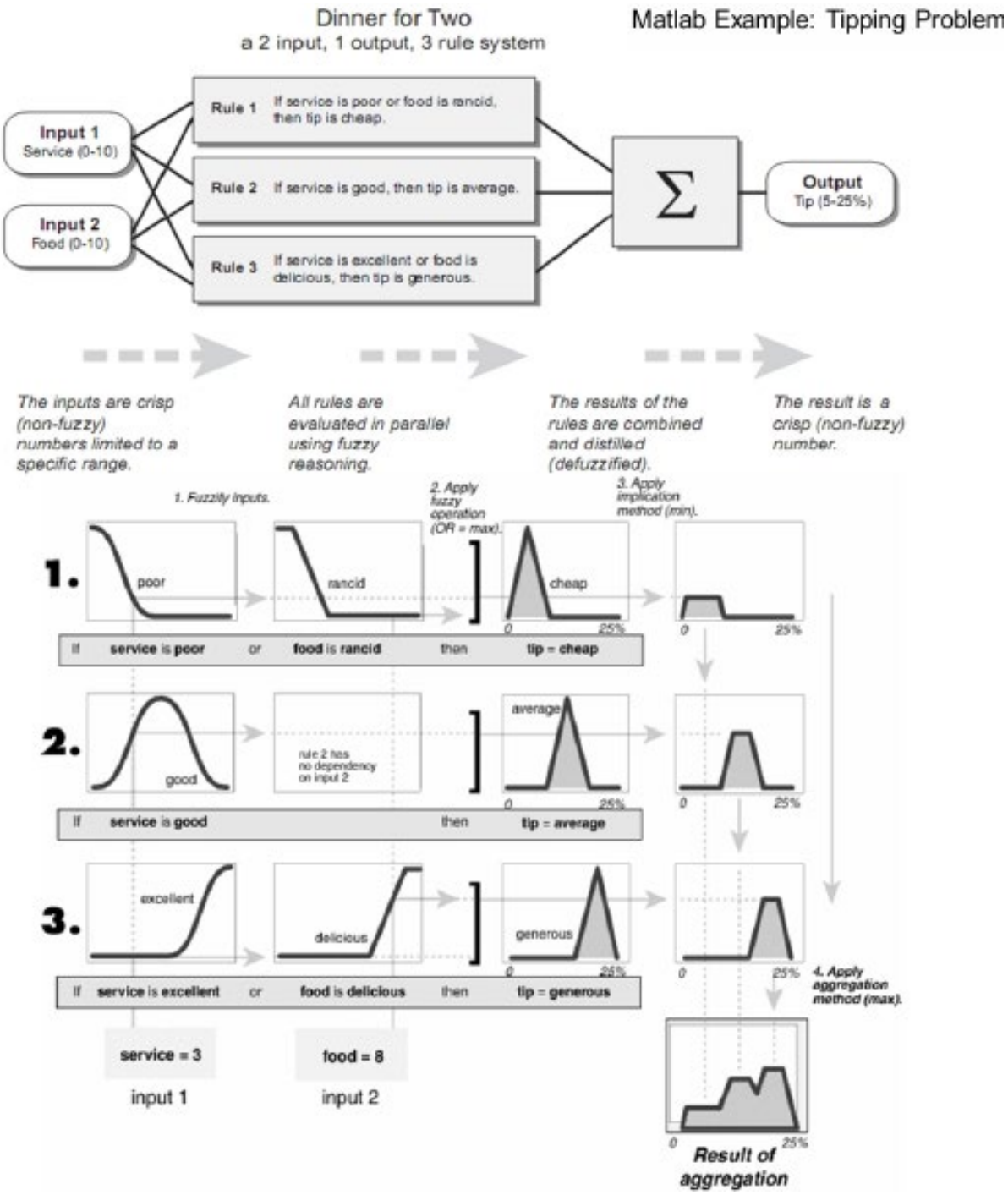
A fuzzy inference system is taking decisions are based on testing of all of the rules, by combining or **aggregating** them in an **inference process**. In fuzzy aggregation the outputs of each rule are combined into a single fuzzy set once for each output variable. In this process input is the list of output functions returned by the implication process for each rule whereas the output is one fuzzy set for each output variable.

The typical **processing pipeline** in a fuzzy system is depicted in the following diagram.

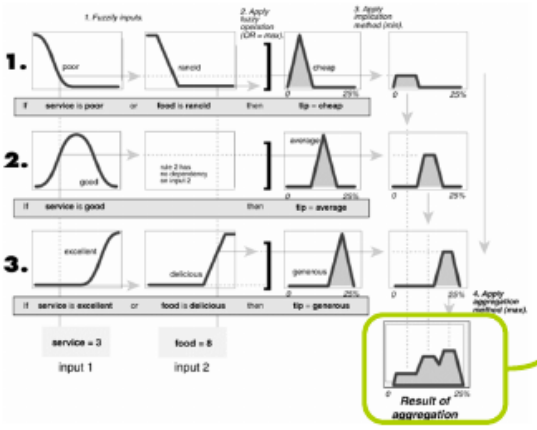


In order to understand the underlying mechanisms of a fuzzy inference system we describe the processing pipeline above for a given problem, namely the tipping problem (adapted from Matlab – Fuzzy toolbox Example).

In this problem we design a fuzzy inference system to compute the tip (output) given two measures of quality, namely service and food quality. For this limited example we choose 3 rules combining the input and output variables with the operators previously introduced.

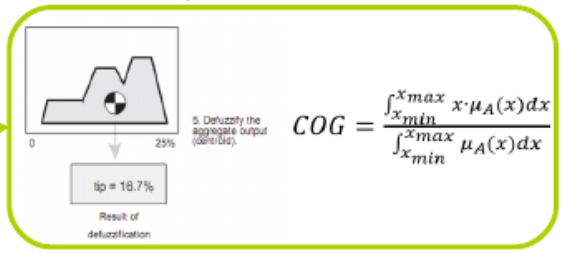


Once the fuzzy inference system has fired all the rules given the current values of the input variables the fuzzy value of the output must be brought back to its crisp representation. The process is called **defuzzification.**



Center of Gravity or **Centroid**

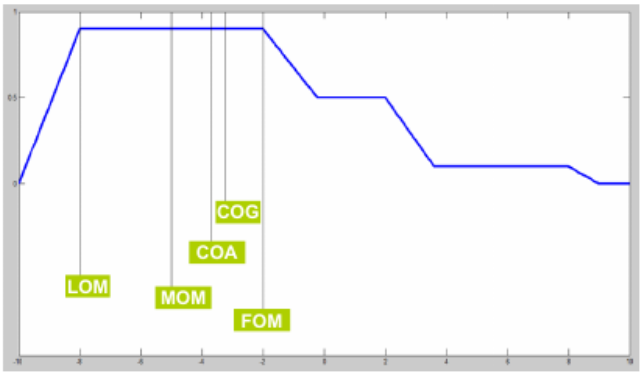
- the most popular
- finds the point where a vertical line would slice the aggregate set into two equal masses.



$$COG = \frac{\int_{x_{min}}^{x_{max}} x \cdot \mu_A(x)dx}{\int_{x_{min}}^{x_{max}} \mu_A(x)dx}$$

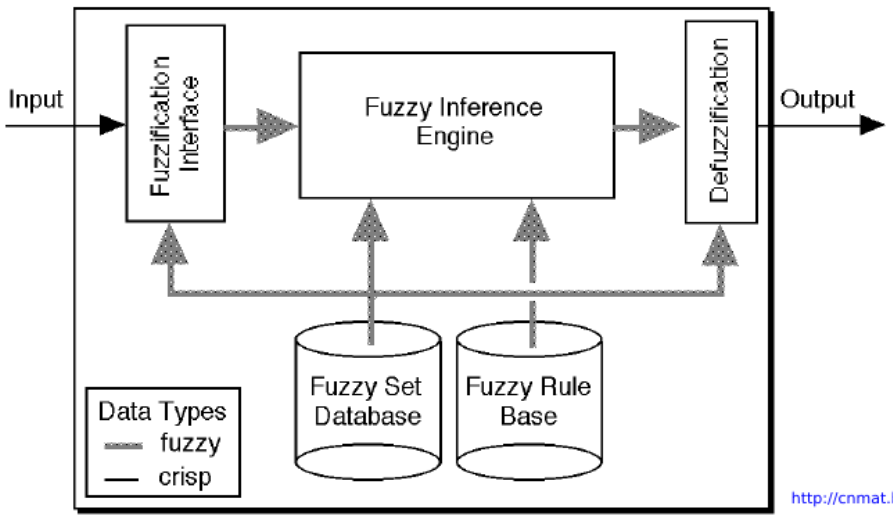
This process is performed using the membership function of the output variable and various mechanisms to defuzzify were developed.

- **Maxima methods and derivatives**
  - Selection of an element from the core of a fuzzy set as defuzzification value. Main advantage is simplicity.
    - Ex: **First of Maxima (FOM), Last of Maxima (LOM), Mean of Maxima (MOM)**
- **Distribution methods and derivatives**
  - Conversion of the membership function into a probability distribution, and computation of the expected value. Main advantage is continuity property.
    - Ex: **Center of Gravity (COG)**
- **Area methods**
  - The defuzzification value divides the area under the membership function in two (more or less) equal parts.
    - Ex: **Center of Area (COA)**



In the next section we will look at how to design and implement a fuzzy inference system for control applications, but before let's take an overview on existing types of fuzzy inference systems.

General structure of a **Fuzzy Infrence System (FIS)**

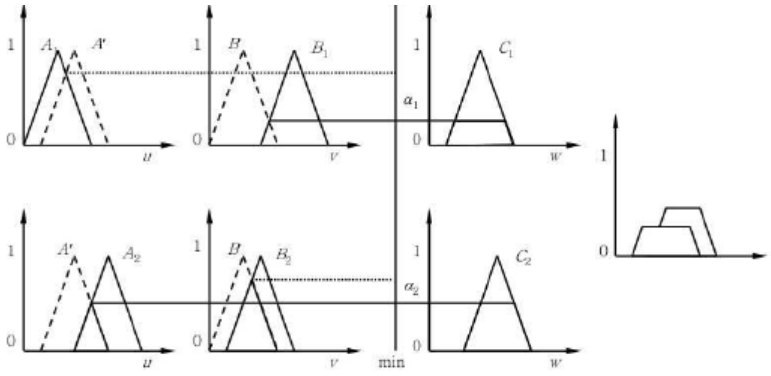


Source: http://cnmat.berkeley.edu/

Fuzzy inference systems differ through the operators implementing the process of implication, aggregation and / or defuzzification. Various types of FIS were developed: Mamdani FIS, Larsen FIS, Tsukamoto FIS and TSK (Takagi – Sugeno – Kang) FIS.

Mamdani FIS

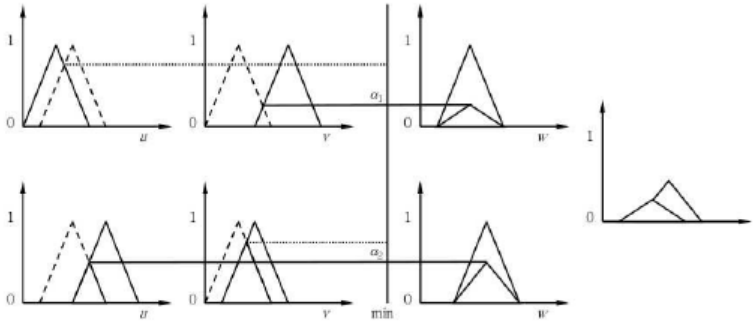
- the most used FIS
- 1975 – FLC to control steam engine and boiler using rules from experienced human operators
- minimum operator for a fuzzy implication
- max operator for the composition



Source:
Kevin M. Passino and S. Yurkovich, Fuzzy Control Addison Wesley , Menlo Park, CA, 1998

Larsen FIS

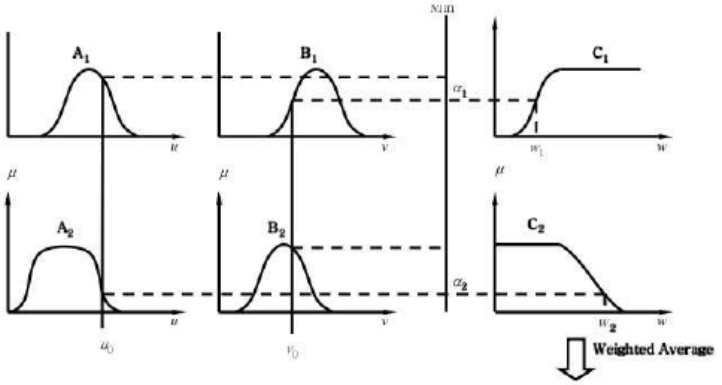
- min operator used for fuzzy implication
- max operator for the composition



Source:
Kevin M. Passino and S. Yurkovich, Fuzzy Control Addison Wesley , Menlo Park, CA, 1998

Tsukamoto FIS

- the consequent part : fuzzy set with a monotonic membership function
- the aggregated result : weighted sum
- no defuzzification needed

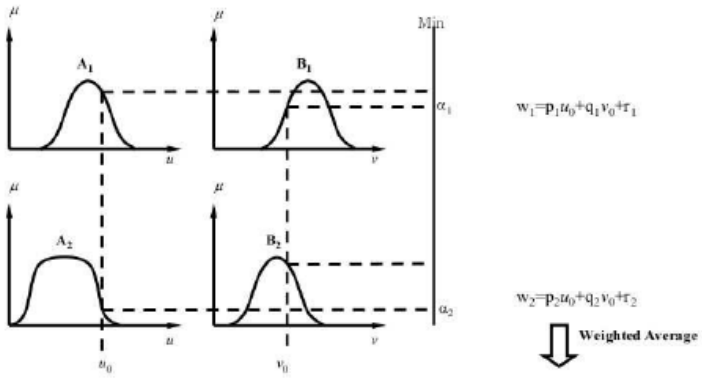


$$w_0 = \frac{\alpha_1 w_1 + \alpha_2 w_2}{\alpha_1 + \alpha_2}$$

Source:
Kevin M. Passino and S. Yurkovich, Fuzzy Control Addison Wesley , Menlo Park, CA, 1998

TSK (Takagi Sugeno Kang)

- the consequent part is given as a function of input variables
- The aggregated result : weighted average using the matching degree
- No deffuzification method needed → output is linear or constant, w = ax + by + c



Source:
Kevin M. Passino and S. Yurkovich, Fuzzy Control Addison Wesley , Menlo Park, CA, 1998

9.2 Fuzzy control systems

Classic control is based on a **detailed I/O function** which maps each high-resolution quantization interval of the input domain into a high-resolution quantization interval of the output domain. Finding a **mathematical expression** for this detailed mapping relationship may be **difficult**, if not impossible, in many applications.

Fuzzy control is based on an I/O function that maps each very low-resolution quantization interval of the input domain into a very low-low resolution quantization interval of the output domain. As there are **fuzzy quantization intervals** covering the input and output domains the **mapping relationship** can be very easily expressed using the **"if-then" formalism**. The overlapping of these fuzzy domains and their linear membership functions will eventually allow to achieve a rather high-resolution **I/O function between crisp input and output variables**.

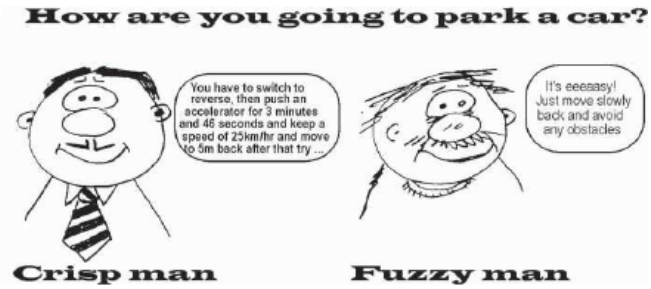
Motivation: classic vs. fuzzy control

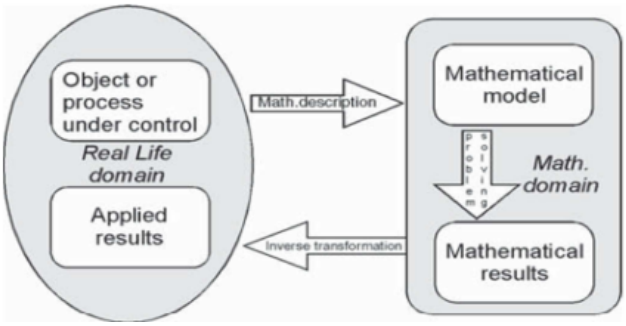
System nonlinearity
- use restrictive linear models
- nonlinear models
  - computationally intensive
  - stability issues

System and measurement uncertainty
- lack of certainty in the model
- structured / unstructured

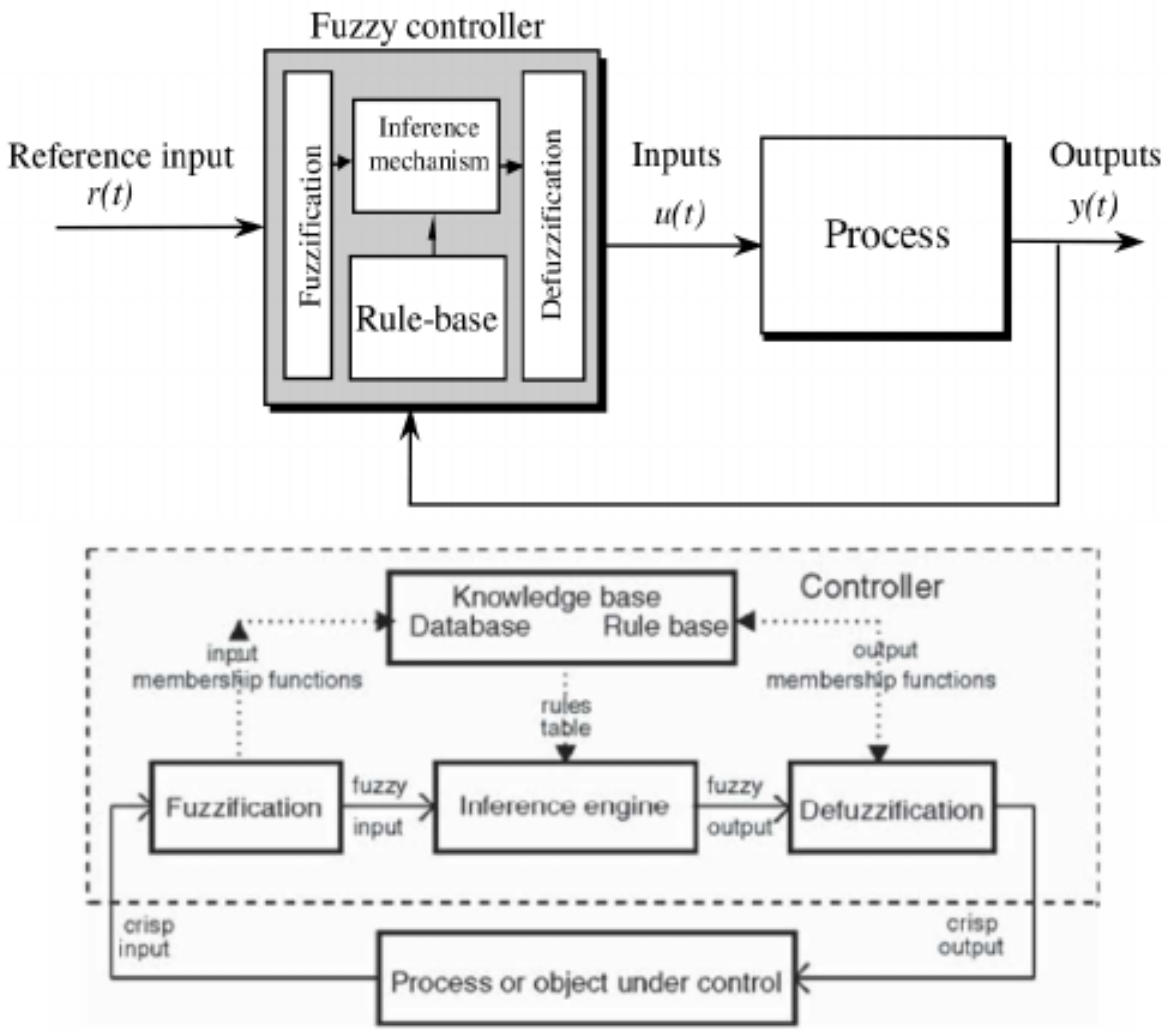
Multivariables, multiloops and environment constraints

Temporal behaviour
- time varying parameters





A block diagram of a **fuzzy control system** is shown in the following diagram.



Source:
Kevin M. Passino and S. Yurkovich, Fuzzy Control, Addison Wesley , Menlo Park, CA, 1998

The **fuzzy controller** is composed of the following four elements:

- **A rule-base** (a set of If-Then rules), which contains a fuzzy logic quantification of the expert's linguistic description of how to achieve good control.
- An inference mechanism (also called an "**inference engine**" or "fuzzy inference" module), which emulates the expert's decision making in interpreting and applying knowledge about how best to control the plant.
- A **fuzzification interface**, which converts controller inputs into information that the inference mechanism can easily use to activate and apply rules.
- A **defuzzification interface**, which converts the conclusions of the inference mechanism into actual inputs for the process.

In the next section we will focus on the design of fuzzy logic controllers through some worked examples of either benchmark control systems or autonomous navigation for cars.

Designing a fuzzy controller: worked examples

Notations:

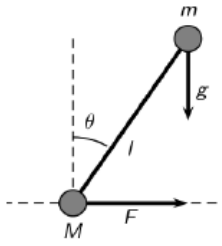
- input variables $\xi_1, \ldots, \xi_n$, control variable η
- measurements: used to determine actual value of η
- η may specify change of η
- assumption: ξ_i, $1 \leq i \leq n$ is value of X_i, $\eta \in Y$
- solution: **control function** φ

$$\varphi : X_1 \times \ldots \times X_n \to Y$$
$$(x_1, \ldots, x_n) \mapsto y$$

Mamdani FLC for inverted pendulum problem (classical benchmark)

Problem setup:

- balance an upright standing pole by moving its foot
- lower end of pole can be moved unrestrained along horizontal axis



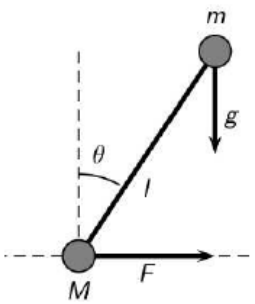
- mass m at foot and mass M at head
- influence of mass of shaft itself is negligible
- determine force F (control variable) that is necessary to balance pole standing upright

$\Rightarrow$ measurement of following output variables

1. angle θ of pole in relation to vertical axis
2. change of angle, *i.e.* triangular velocity $\dot{\theta} = \frac{d\theta}{dt}$

both should be zero

Setup:



- angle $\theta \in X_1 = [-90\,^\circ, 90\,^\circ]$
- theoretically, every angle velocity $\dot{\theta}$ possible
- extreme $\dot{\theta}$ are artificially achievable
- assume $-45\,^\circ/\mathrm{s} \leq \dot{\theta} \leq 45\,^\circ/\mathrm{s}$ holds *i.e.* $\dot{\theta} \in X_2 = [-45\,^\circ/\mathrm{s}, 45\,^\circ/\mathrm{s}]$
- absolute value of force $|F| \leq 10\,\mathrm{N}$

$\Rightarrow$ define $F \in Y = [-10\,\mathrm{N}, 10\,\mathrm{N}]$

differential equation of cartpole problem:

$$(M+m)\sin^2\theta \cdot l \cdot \ddot{\theta} + m \cdot l \cdot \sin\theta\cos\theta \cdot \dot{\theta}^2 - (M+m) \cdot g \cdot \sin\theta = -F \cdot \cos\theta$$

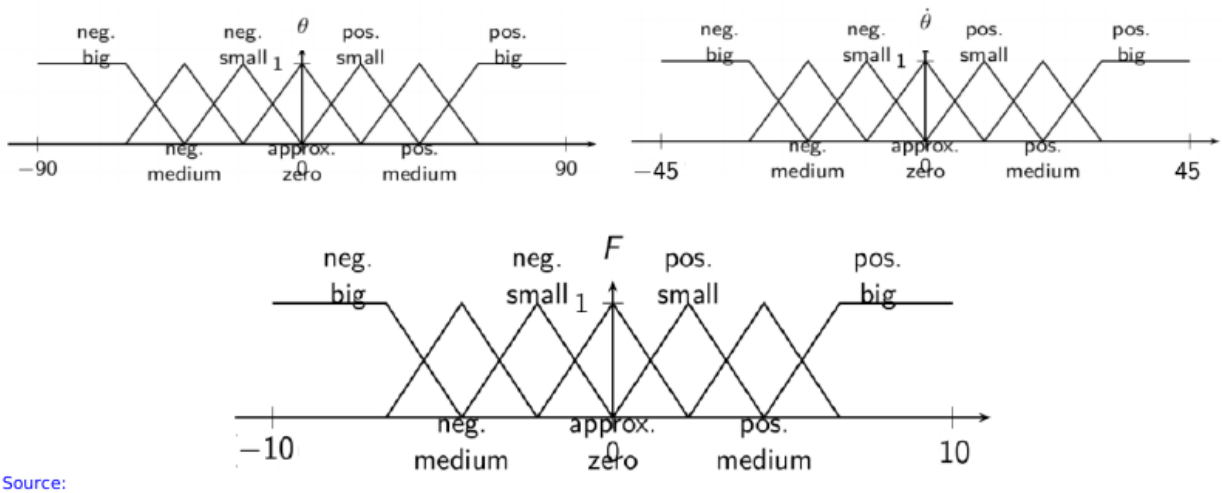
Fuzzy setup:

- X_1 partitioned into 7 fuzzy sets
- support of fuzzy sets: intervals with length $\frac{1}{4}$ of whole range X_1
- similar fuzzy partitions for X_2 and Y
- **next step:** specify rules

$$\text{if } \xi_1 \text{ is } A^{(1)} \text{ and } \ldots \text{ and } \xi_n \text{ is } A^{(n)} \text{ then } \eta \text{ is } B$$

$A^{(1)}, \ldots, A^{(n)}$ and B represent linguistic terms corresponding to $\mu^{(1)}, \ldots, \mu^{(n)}$ and μ according to $X_1, \ldots, X_n$ and Y

- rule base consists of k rules



Source:
Michels, K., Klawonn, F., Kruse, R., and Nürnberger, A. (2006), Fuzzy Control: Fundamentals, Stability and Design of Fuzzy Controllers

$\dot{\theta}$ \ θ	nb	nm	ns	az	ps	pm	pb
nb			ps	pb			
nm				pm			
ns	nm		ns	ps			
az	nb	nm	ns	az	ps	pm	pb
ps				ns	ps		pm
pm				nm			
pb				nb	ns		

- 19 rules for cartpole problem, *e.g.*

If θ is *approximately zero* and $\dot{\theta}$ is *negative medium* then F is *positive medium*.

- measurement $(x_1, \ldots, x_n) \in X_1 \times \ldots \times X_n$ is forwarded to decision logic
- consider rule

$$\text{if } \xi_1 \text{ is } A^{(1)} \text{ and } \ldots \text{ and } \xi_n \text{ is } A^{(n)} \text{ then } \eta \text{ is } B$$

- decision logic computes degree to $\xi_1, \ldots, \xi_n$ fulfills premise of rule
- for $1 \leq \nu \leq n$ value $\mu^{(\nu)}(x_\nu)$ is calculated
- combine values conjunctively by $\alpha = \min\left\{\mu^{(1)}, \ldots, \mu^{(n)}\right\}$
- for each rule R_r of the k rules:

$$\alpha_r = \min\left\{\mu^{(1)}_{i_{1,r}}(x_1), \ldots, \mu^{(n)}_{i_{n,r}}(x_n)\right\}$$

  - output of R_r = fuzzy set of output values

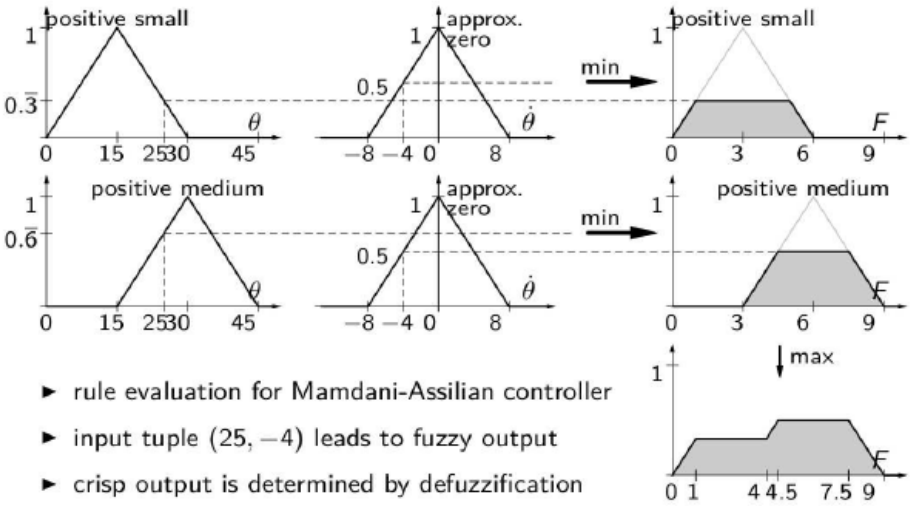
$\Rightarrow$ "cutting off" fuzzy set μ_{i_r} associated with conclusion of R_r at α_r

$\Rightarrow$ for input $(x_1, \ldots, x_n)$, R_r implies fuzzy set

$$\mu^{\text{output}(R_r)}_{x_1,\ldots,x_n} : Y \to [0,1]$$
$$y \mapsto \min\left\{\mu^{(1)}_{i_{1,r}}(x_1), \ldots, \mu^{(n)}_{i_{n,r}}(x_n), \mu_{i_r}(y)\right\}$$

- if $\mu^{(1)}_{i_{1,r}}(x_1) = \ldots = \mu^{(n)}_{i_{n,r}}(x_n) = 1$, then $\mu^{\text{output}(R_r)}_{x_1,\ldots,x_n} = \mu_{i_r}$
- if for all $\nu \in \{1, \ldots, n\}$, $\mu^{(\nu)}_{i_{1,r}}(x_\nu) = 0$, then $\mu^{\text{output}(R_r)}_{x_1,\ldots,x_n} = 0$

Rule evaluation:



Combination of rules:

- decision logic combines fuzzy sets from all rules
- **maximum** leads to output fuzzy set

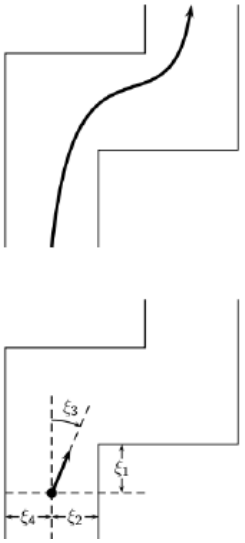
$$\mu^{\text{output}}_{x_1,\ldots,x_n} : Y \to [0,1]$$
$$y \mapsto \max_{1\le r\le k} \left\{ \min \left\{ \mu^{(1)}_{i_{1,r}}(x_1), \ldots, \mu^{(n)}_{i_{n,r}}(x_n), \mu_{i_r}(y) \right\} \right\}$$

$$\Rightarrow \quad \mu^{\text{output}}_{x_1,\ldots,x_n} \text{ is passed to defuzzification interface}$$

In the second worked example we analyze the design of a TSK fuzzy controller. Now we don't consider the defuzzification step anymore.

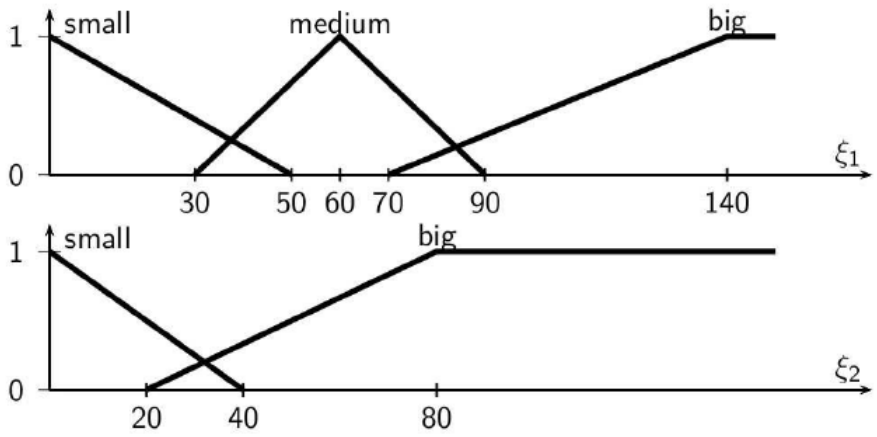
TSK FLC for Car steering assistance

System description:



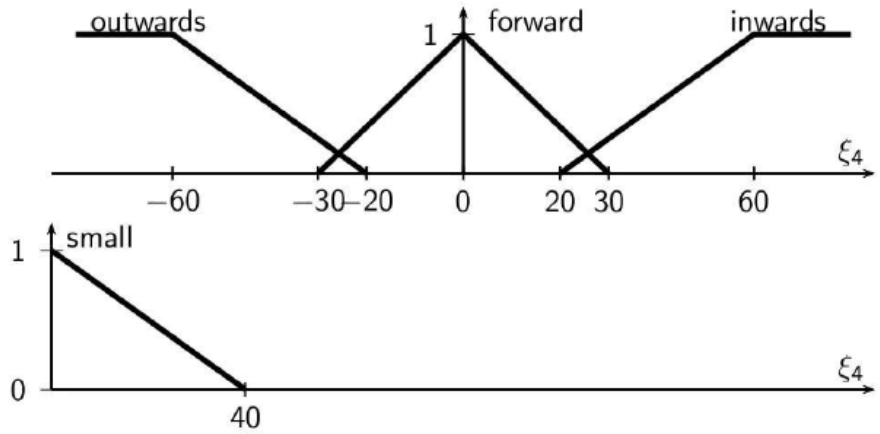
- pass bend with car at constant speed
- measured inputs:
  - ξ_1 : distance of car to beginning of bend
  - ξ_2 : distance of car to inner barrier
  - ξ_3 : direction (angle) of car
  - ξ_4 : distance of car to outer barrier
- η = rotation speed of steering wheel
- $X_1 = [0\,\text{cm}, 150\,\text{cm}]$, $X_2 = [0\,\text{cm}, 150\,\text{cm}]$
- $X_3 = [-90°, 90°]$, $X_4 = [0\,\text{cm}, 150\,\text{cm}]$

Fuzzy partitions for variables:



Source:
Michels, K., Klawonn, F., Kruse, R., and Nürnberger, A. (2006), Fuzzy Control: Fundamentals, Stability and Design of Fuzzy Controllers

Fuzzy partitions for variables:



Form of rules

$$R_r : \textbf{if } \xi_1 \text{ is } A \text{ and } \xi_2 \text{ is } B \text{ and } \xi_3 \text{ is } C \text{ and } \xi_4 \text{ is } D$$
$$\textbf{then } \eta = p_0^{(A,B,C,D)} + p_1^{(A,B,C,D)} \cdot \xi_1 + p_2^{(A,B,C,D)} \cdot \xi_2 + p_3^{(A,B,C,D)} \cdot \xi_3 + p_4^{(A,B,C,D)} \cdot \xi_4$$

$A \in \{small,\ medium,\ big\}$

$B \in \{small,\ big\}$

$C \in \{outwards,\ forward,\ inwards\}$

$D \in \{small\}$

$p_0^{(A,B,C,D)}, \ldots, p_4^{(A,B,C,D)} \in \mathbb{R}$

rule	ξ_1	ξ_2	ξ_3	ξ_4	p_0	p_1	p_2	p_3	p_4
R_1	-	-	outwards	small	3.000	0.000	0.000	−0.045	−0.004
R_2	-	-	forward	small	3.000	0.000	0.000	−0.030	−0.090
R_3	small	small	outwards	-	3.000	−0.041	0.004	0.000	0.000
R_4	small	small	forward	-	0.303	−0.026	0.061	−0.050	0.000
R_5	small	small	inwards	-	0.000	−0.025	0.070	−0.075	0.000
R_6	small	big	outwards	-	3.000	−0.066	0.000	−0.034	0.000
R_7	small	big	forward	-	2.990	−0.017	0.000	−0.021	0.000
R_8	small	big	inwards	-	1.500	0.025	0.000	−0.050	0.000
R_9	medium	small	outwards	-	3.000	−0.017	0.005	−0.036	0.000
R_{10}	medium	small	forward	-	0.053	−0.038	0.080	−0.034	0.000
R_{11}	medium	small	inwards	-	−1.220	−0.016	0.047	−0.018	0.000
R_{12}	medium	big	outwards	-	3.000	−0.027	0.000	−0.044	0.000
R_{13}	medium	big	forward	-	7.000	−0.049	0.000	−0.041	0.000
R_{14}	medium	big	inwards	-	4.000	−0.025	0.000	−0.100	0.000
R_{15}	big	small	outwards	-	0.370	0.000	0.000	−0.007	0.000
R_{16}	big	small	forward	-	−0.900	0.000	0.034	−0.030	0.000
R_{17}	big	small	inwards	-	−1.500	0.000	0.005	−0.100	0.000
R_{18}	big	big	outwards	-	1.000	0.000	0.000	−0.013	0.000
R_{19}	big	big	forward	-	0.000	0.000	0.000	−0.006	0.000
R_{20}	big	big	inwards	-	0.000	0.000	0.000	−0.010	0.000

Sample calculations:

- assume car is $10\,\mathrm{cm}$ away from beginning of bend ($\xi_1 = 10$)
- distance of car to inner barrier: $30\,\mathrm{cm}$ ($\xi_2 = 30$)
- distance of car to outer barrier: $50\,\mathrm{cm}$ ($\xi_4 = 50$)
- direction of car: "forward" ($\xi_3 = 0$)

$\Rightarrow$ according to all rules $R_1, \ldots, R_{20}$:
only premises of R_4 and R_7 have value $\neq 0$

Membership degrees to control car:

	small	medium	big
$\xi_1 = 10$	0.8	0	0

	small	big
$\xi_2 = 30$	0.25	0.167

	outwards	forward	inwards
$\xi_3 = 0$	0	1	0

	small
$\xi_4 = 50$	0

Sample output:

- for premise of R_4 and R_7, $\alpha_4 = 1/4$ and $\alpha_7 = 1/6$, resp.
- rules weights $W_4 = \frac{1/4}{1/4+1/6} = 3/5$ for R_4 and $W_5 = 2/5$ for R_7
- R_4 yields

$$\begin{aligned}\eta_4 &= 0.303 - 0.026 \cdot 10 + 0.061 \cdot 30 - 0.050 \cdot 0 + 0.000 \cdot 50 \\ &= 1.873\end{aligned}$$

- R_7 yields

$$\begin{aligned}\eta_4 &= 2.990 - 0.017 \cdot 10 + 0.000 \cdot 30 - 0.021 \cdot 0 + 0.000 \cdot 50 \\ &= 2.820\end{aligned}$$

- value for control variable:

$$\eta = 3/5 \cdot 1.873 + 2/5 \cdot 2.820 = 2.2518$$

Contents

Lectures

10. Online distributed streaming machine learning

During the day when you are reading this, more data will be produced than the amount of information contained in all printed material in the world. The Internet Data Center estimated the growth of data to be of a factor of 300 between 2005 and 2020, expecting to raise from 130 Exabytes to 20,000 Exabytes.

Big Data is one of the most popular terms nowadays, but Big Data is not only about the **volume**. Much of the data is **received in real time** and is most **valuable at the time of arrival**. For example, we want to detect shares market trends as soon as possible; a service operator wants to detect failures from logs within a seconds; and a news site may want to train their model to show users content which they are interesting in as shown extensively in Figure 1.

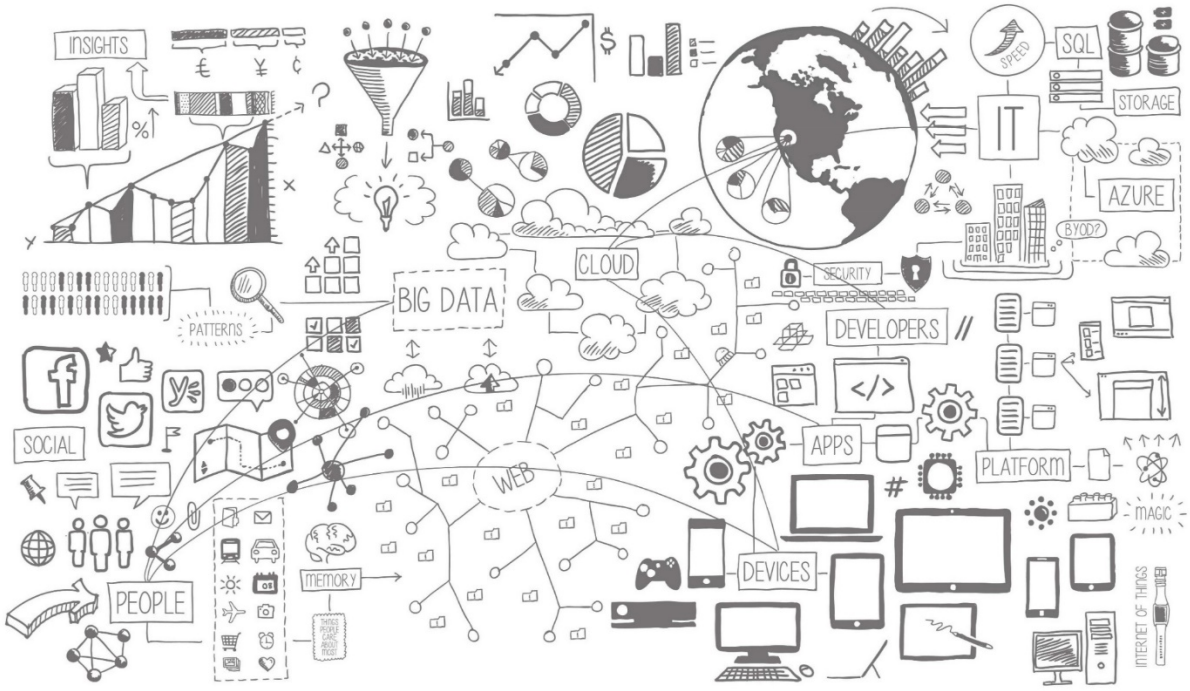


Figure 1 Big Data Landscape

Big data are often characterized like **Volume + Velocity + Variety**. **Volume** describes the quantity of data, it is the size of the data which determines the value and potential of the data. **Variety** is the next aspect of Big Data. It describes range of data types and sources. The term **'velocity'** in the context refers to the speed of generation of data or how fast the data is generated and processed to meet the demands. This is shown in Figure 2.

10.1 Real Time Big Data Analytics

We are going to focus on **stream processing** or sometimes referenced as fast data, where velocity is the key. The demand for stream processing is increasing. Immense

amounts of data have to be processed fast from a **rapidly growing set of disparate data sources**. This pushes the limits of traditional data processing infrastructures. These **stream-based applications** include **trading**, **social networks**, **Internet of Things**, system monitoring, and many other examples.

Figure 2 Velocity in Big Data

10.2 Stream Processing Engines

Stream processing paradigm simplifies parallel software and hardware by restricting the parallel computation that can be performed.
Given a **sequence of data (a stream)**, **a series of operations (functions)** is **applied** to **each element** in the stream, in a **declarative** way, we specify what we want to achieve and not how, as shown in Figure 3.

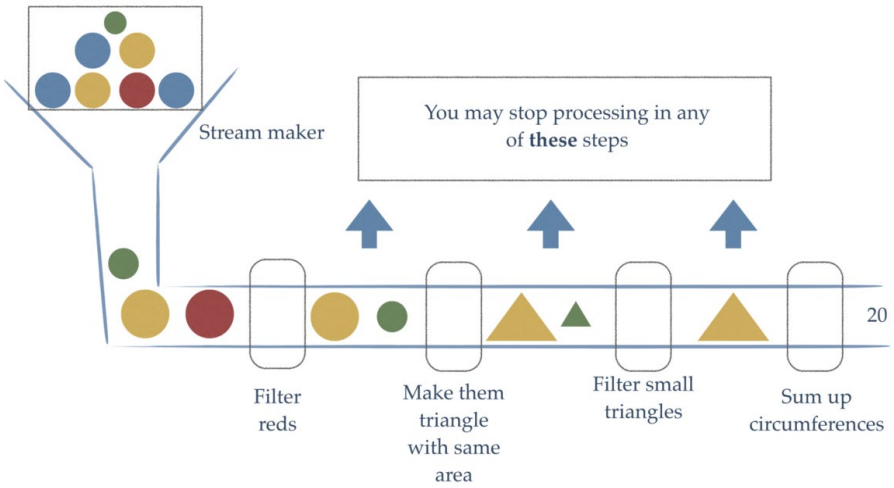


Figure 3 Stream processing paradigm

In general, **stream processing systems** support a large class of applications in which data are generated from **multiple sources** and are pushed **asynchronously** to servers which are responsible for processing. Therefore, stream processing applications are usually deployed as **continuous jobs** that run from the time of their submission until their cancellation.

Many applications in several domains such as telecommunications, network security and large scale sensor networks require online processing of continuous data flows. They produce **very high loads** that requires **aggregating the processing capacity of many nodes**.

Rather than **processing stored data like** in traditional database systems, **stream processing engines process data on-the-fly**. This is due to the amount of input that discourages persistent storage and the requirement of providing prompt results. Queries of streaming application are generally **continuous** and **stateful**. Once a query is registered, it starts **processing events** and only stops when the system terminates or the query is deregistered from the system. **Queries** typically maintain state such as **aggregates** of **windows** or **local variables**. **Query** state is kept on the same node that executes the query.

At the most basic level, shown in Figure4, a **stream processor,** such as **Flink,** is made up of:

- **Data source**: Incoming data that stream engine processes
- **Transformations**: The processing step, when the stream processor modifies incoming data
- **Data sink**: Where the processing engine sends data after processing

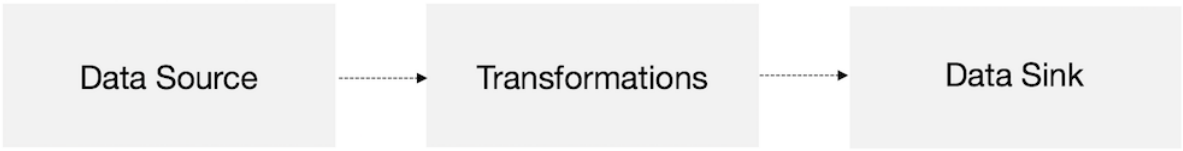


Figure 4 Stream processing paradigm

Aggregating events (e.g., counts, sums) works differently on streams because it is impossible to count all (**unbounded**).

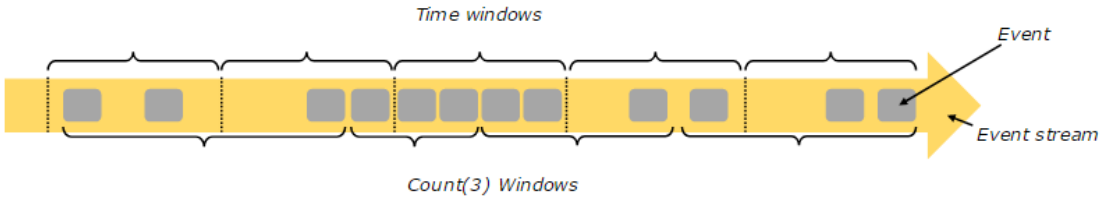


Figure 5 Stream aggregations

Stream processing and **windowing** makes it easy to compute **accurate results** over streams where events arrive **out of order** and where events may arrive **delayed**.

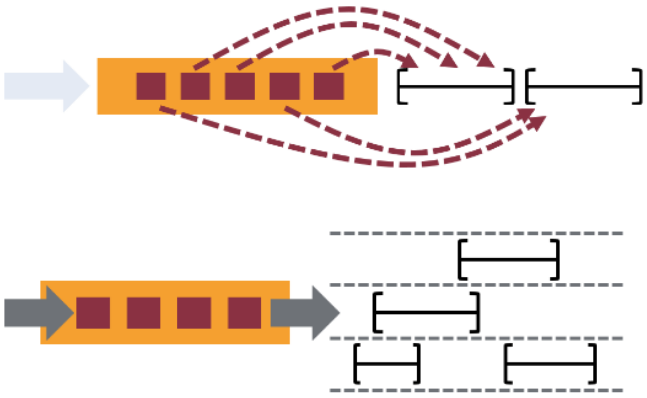


Figure 6 Stream windowing mechanisms

Windowing based on **time**, **count**, and **data-driven** windows. Windows can be customized with flexible **triggering conditions** to support sophisticated streaming patterns.

When executed, stream processor engine (e.g. Flink) programs are **mapped to streaming dataflows**, consisting of **streams** and **transformation** operators. Each dataflow starts with one or more sources and ends in one or more sinks. The dataflows resemble arbitrary **directed acyclic graphs** (**DAGs**).

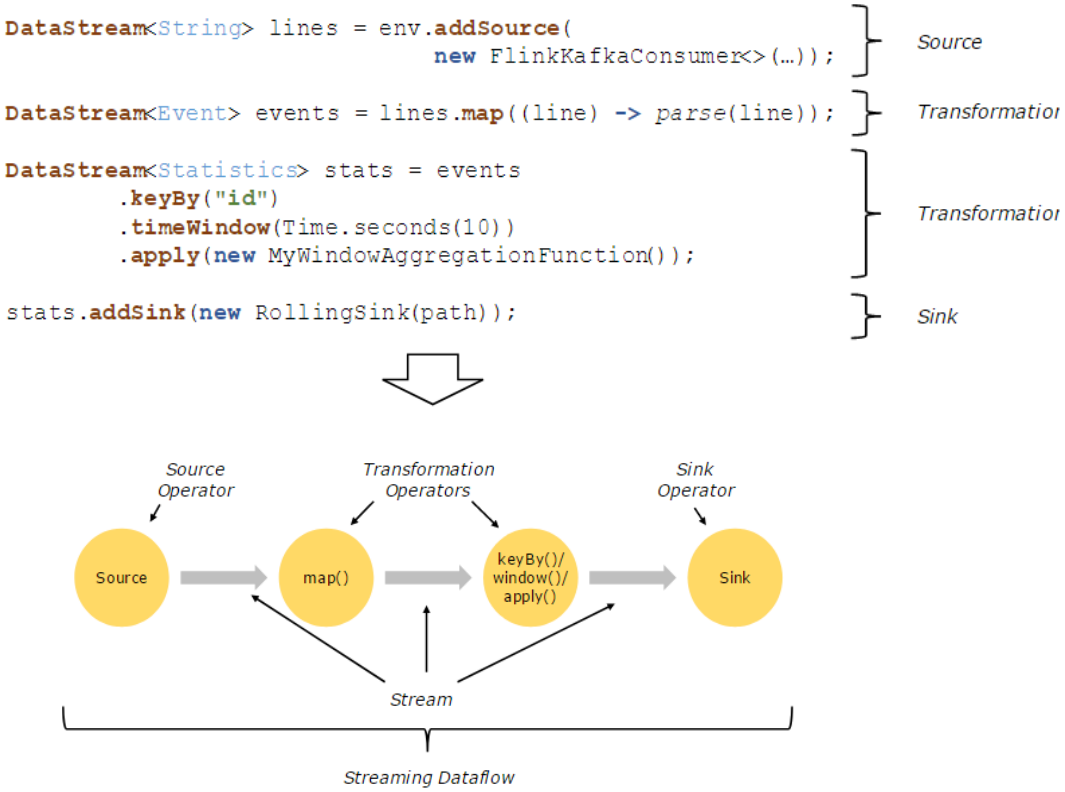


Figure 7 Stream processor execution

For **distributed execution**, Flink for example, **chains operator** subtasks together into tasks. Each **task** is **executed** by **one thread**. **Chaining operators** together into tasks is a useful **optimization**: it reduces the **overhead** of thread-to-thread handover and buffering, and **increases** overall **throughput** while **decreasing latency**.

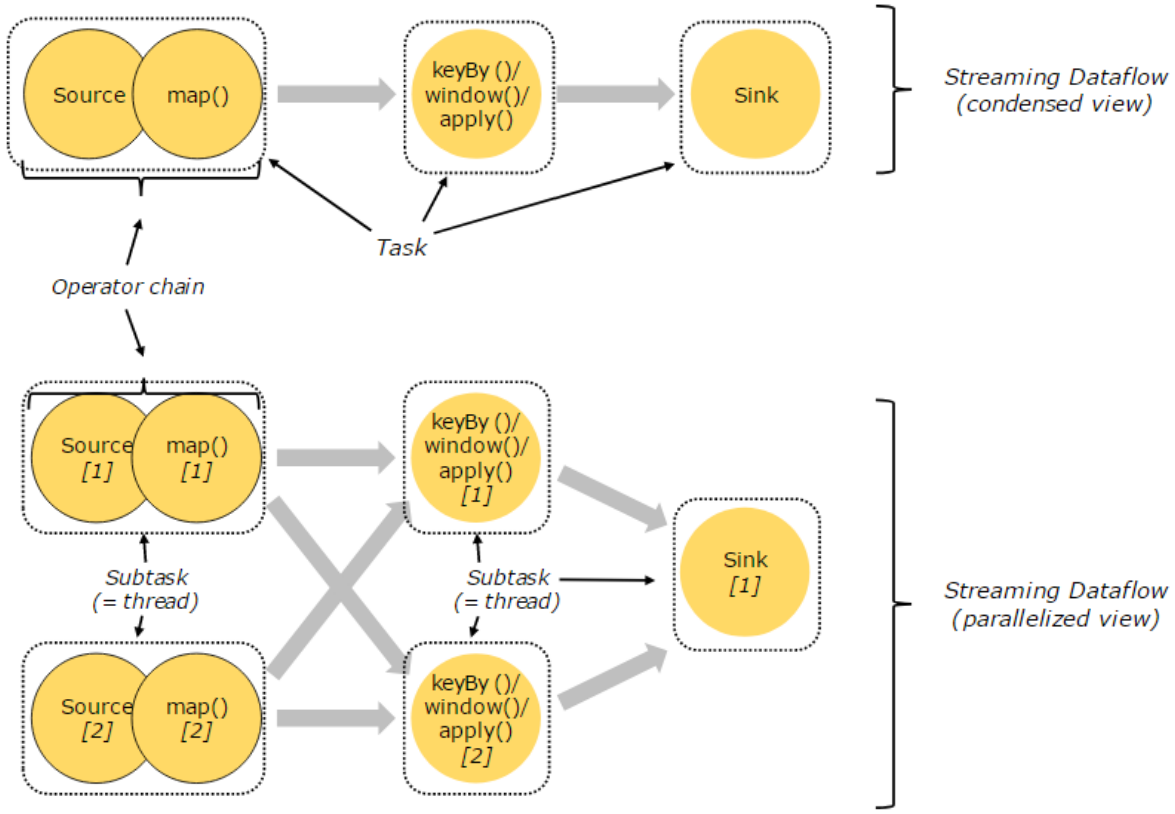


Figure 8 Distributing stream processing

A **stream processing engine** exploits **in-memory data streaming**, and natively executes **iterative processing algorithms** which are **common in ML**. This allows data scientists to **test their models locally** and using subsets of data, and then use the same code to **run** their **algorithms** at a much **larger scale** in a cluster setting.

10.3 Machine Learning in Real-Time Big Data Analytics

In order to deal with **evolving data streams**, the **model** learned from the streaming data must be able to **capture up-to-date trends** and **transient patterns** in the stream. To do this, as we **revise the model** by **incorporating new examples**, we must also **eliminate** the effects of **outdated examples** representing outdated concepts.

Dealing with **time-changing data** requires **strategies** for **detecting** and **quantifying change**, **forgetting** stale examples, and for **model revision**. Fairly **generic strategies** exist for **detecting change** and deciding when **examples** are **no longer**

relevant. **Model revision** strategies, on the other hand, are in most cases **method-specific**.

A good idea is to **encapsulate** all the **statistical calculations** having to do with **detecting change** and **keeping updated statistics** from a stream in an abstract data type that can then be used to replace the **counters** and **accumulators** that typically all **machine learning** and **data mining algorithms use** to make their **decisions**, including when **change has occurred**.

Big Data Stream Learning is more challenging than **batch** or **offline learning**, since the data may **not preserve the same distribution** over the lifetime of the stream. Moreover, each **example** coming in a stream can only be **processed once**, or needs to be **summarized** with a **small memory footprint**, and the learning **algorithms** must be **efficient**.

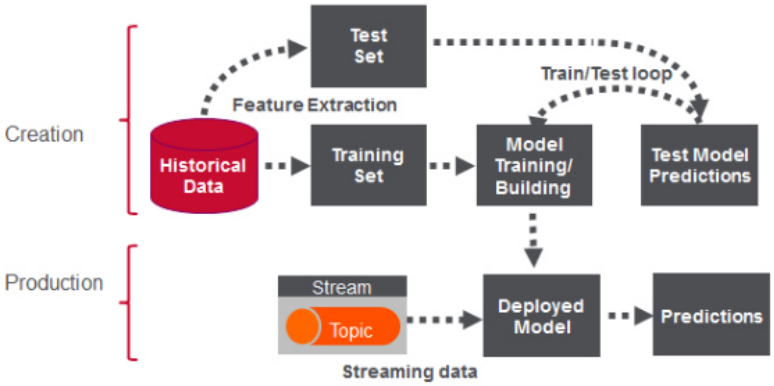


Figure 9 Big Data Stream Learning – A generic view

In order to deal with evolving data streams, the **model** learnt from the streaming data must **capture up-to-date trends** and **transient patterns** in the stream. Updating the model by **incorporating new examples**, we must also **eliminate** the effects of **outdated examples** representing outdated concepts.

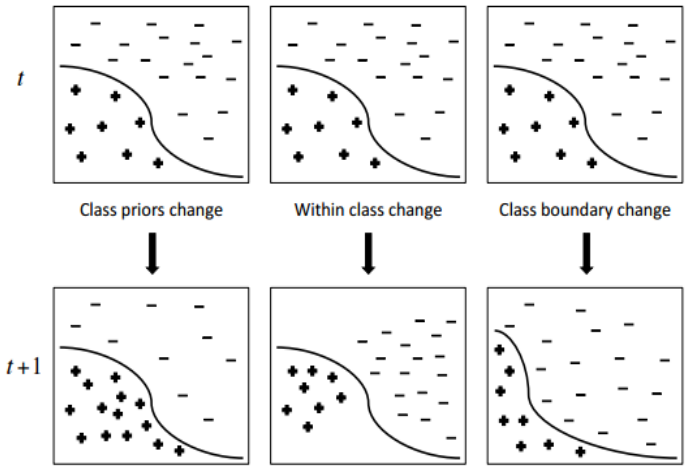


Figure 10 Changes in data streams

How to compute the entropy of a **collection of infinite data**, where the **domain of the variables** can be **huge** and the **number of classes** of objects is **not known** a priori?

How to maintain the **k-most frequent items** in a retail data warehouse with **3 TB of data, 100s of GB** of new sales records updated daily with **1000000s** different items?

What becomes of **statistical computations** when the **learner** can only afford **one pass through each data sample** because of **time** and **memory constraints**; when the learner has to **decide on-the-fly** what is relevant and process it and what is **redundant** and could be **discarded**?

Most strategies use variations of the **sliding window technique**: a window is maintained that keeps **the most recently read examples**, and from which older examples are dropped according to some set of rules.

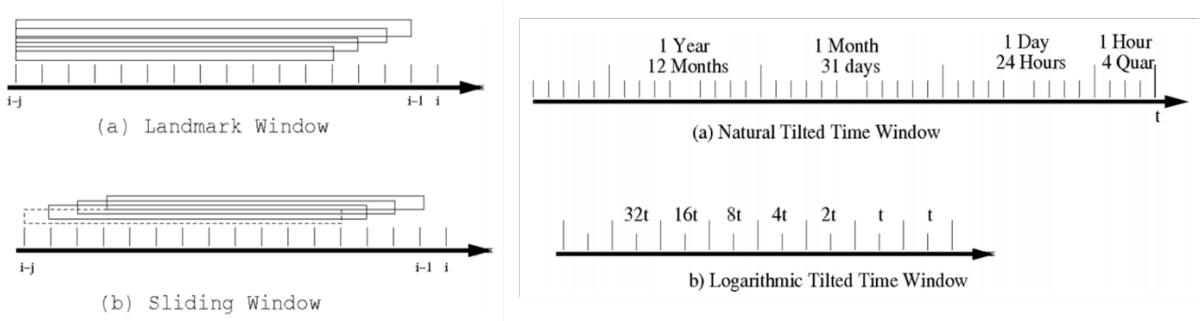


Figure 11 Sliding window mechanisms

The contents of the **sliding window** can be used for the **three tasks**:

- to **detect change** (e.g., by using some statistical test on different sub-windows),
- to obtain **updated statistics / criteria** from the **recent examples**, and
- to have data to **rebuild or update the model** after data has changed.

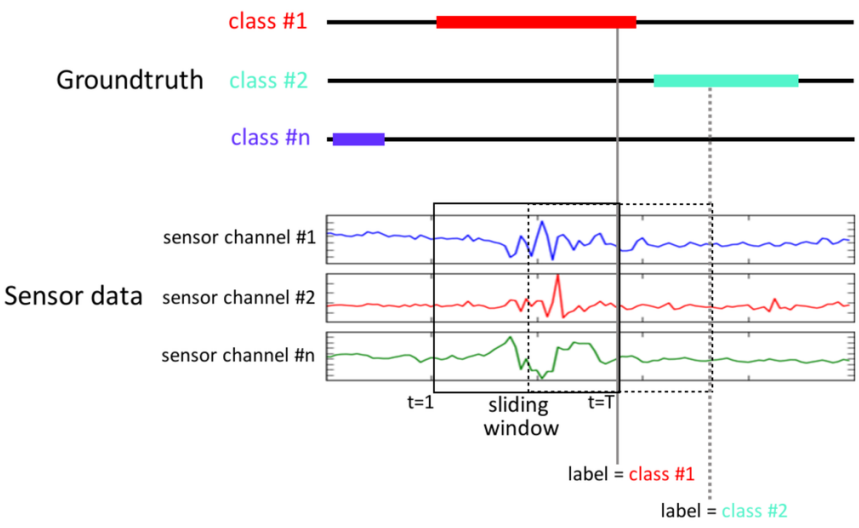


Figure 12 Sliding window processing

Normally, the user is caught in a **tradeoff** without solution:

- a **small size** (so that the window reflects accurately the current distribution)
- a **large size** (so that many examples are available to work on, increasing accuracy in periods of stability).

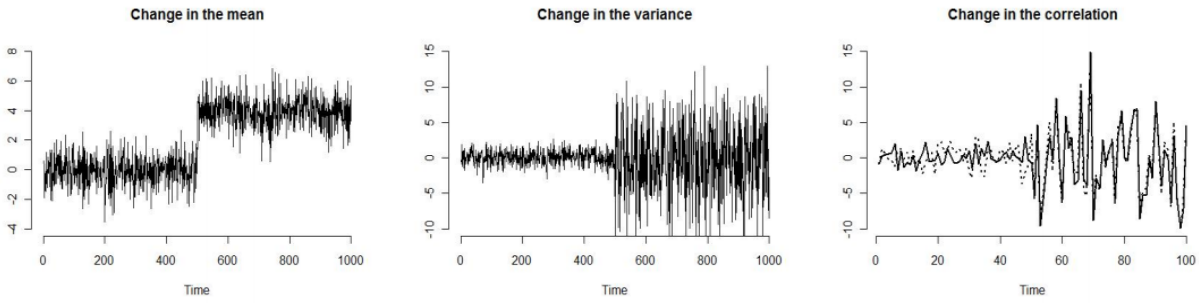


Figure 13 Distribution change in stream processing

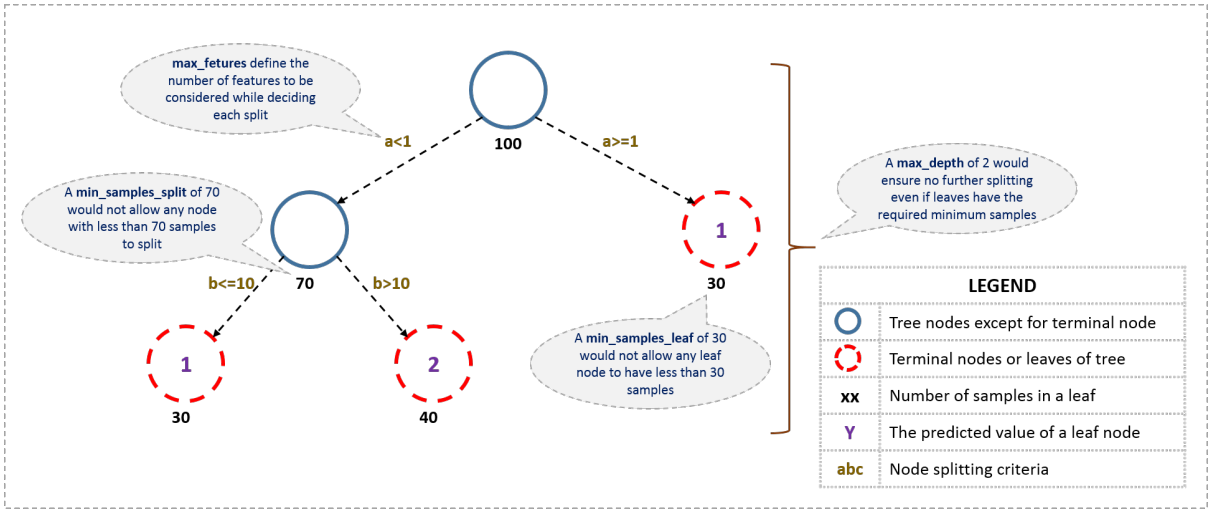
Currently, it has been proposed to use **windows of variable size**.

10.4 Vertical Hoeffding Tree Classifiers

Most **conventional data mining** techniques have to be **adapted** to **run** in a **streaming environment**, because of the underlying **resource constraints** in terms of **memory** and **running time**. Furthermore, the data stream may often show **concept drift**, because of which **adaptation of conventional algorithms** becomes more challenging. One such important conventional data mining problem is that of **classification**.

In the **classification** problem, we attempt to **model** the **class variable** on the basis of one or more **feature variables**. While this problem has been extensively studied from a conventional mining perspective, it is a much more challenging problem in the data stream domain.

In this section we introduce and illustrate a method for **developing decision trees algorithms** that can **adaptively learn** from **data streams** that **change over time**. We take the **Hoeffding Tree learner**, an **incremental decision tree inducer** for data streams, and use as a basis it to build two new methods that **can deal with distribution and concept drift**. In the basic formulation a decision tree functionality is described in Figure 14.



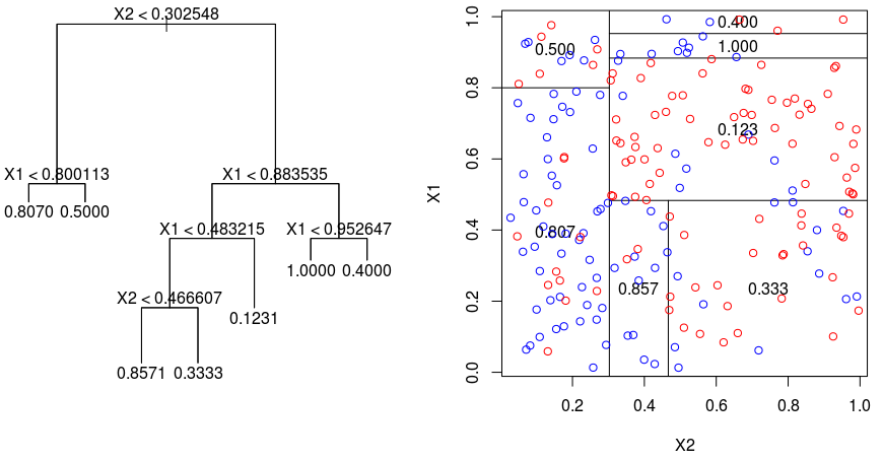


Figure 14 Decision tree processing and tree induction process

Given a set of training examples belonging to n different classes, **a classifier algorithm** builds a model that predicts for every unlabeled instance x the class C to which it belongs, synthetically depicted in Figure 15.

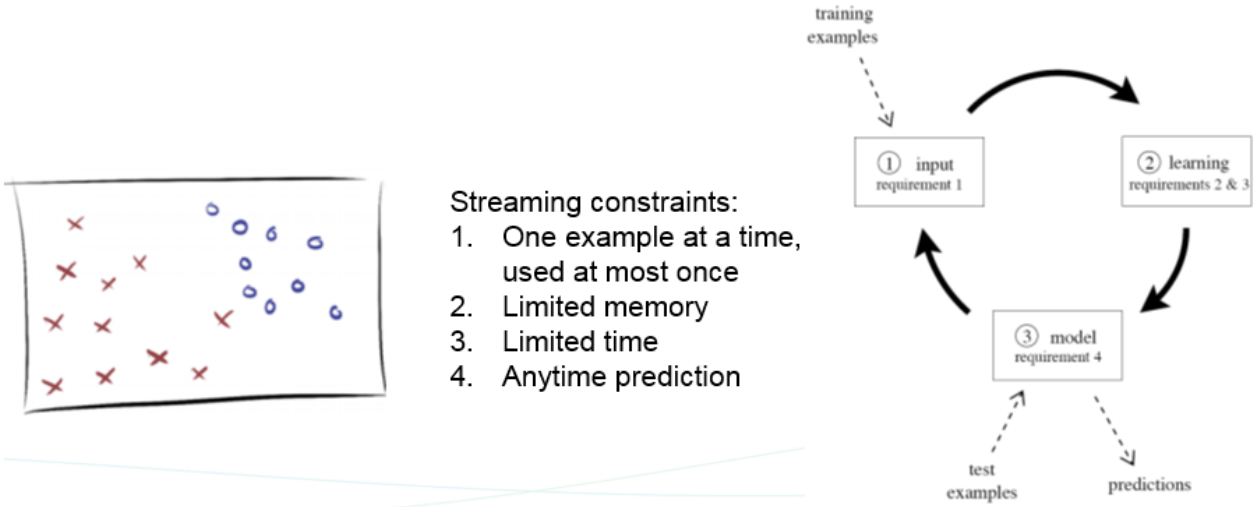


Figure 15 Classification in streaming

The basic decision tree can be adapted for streaming operation, as shown in Figure 16, and the core principle applied to evolving data, by modifying the number of sufficient statistics and decision at the split level.

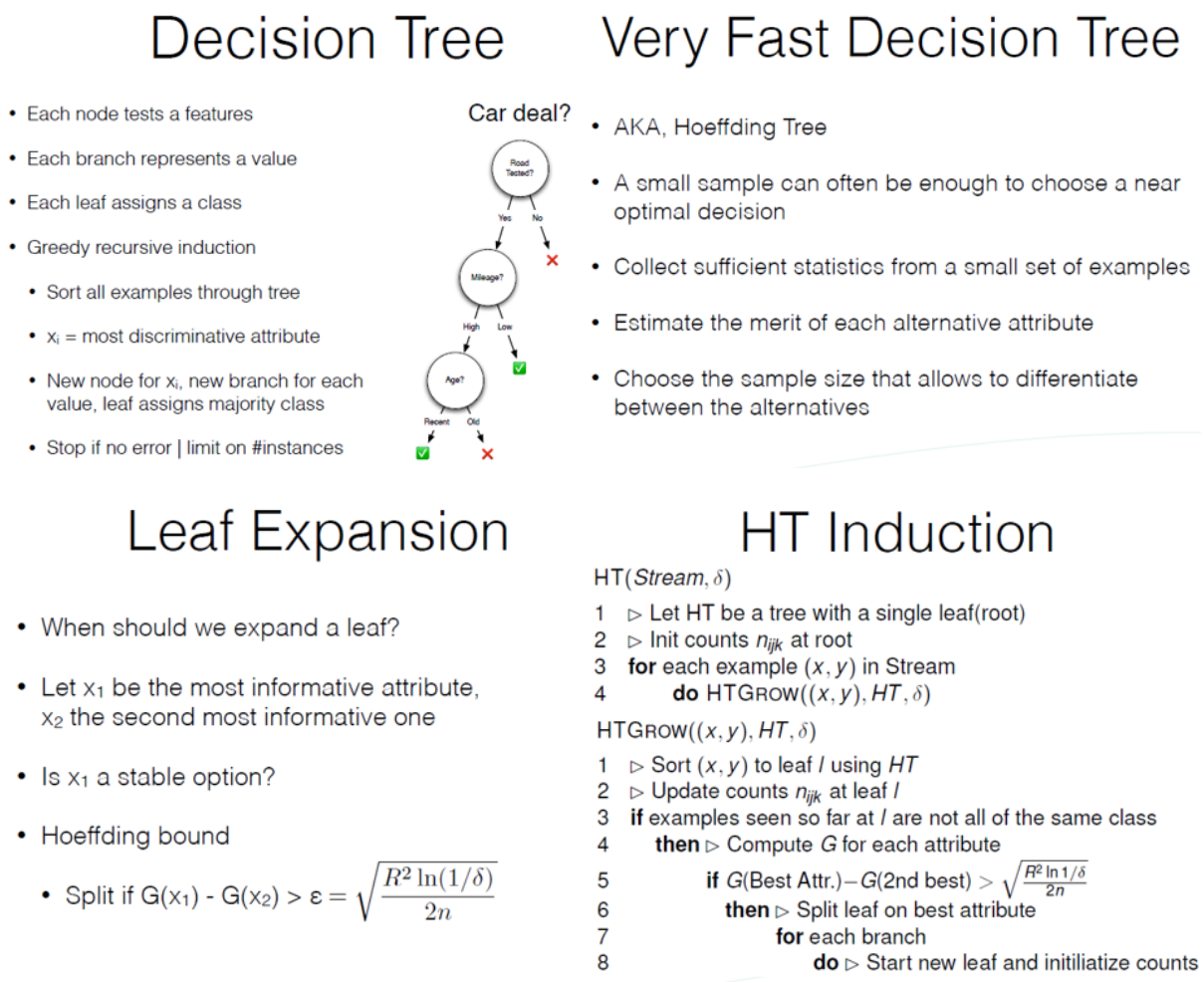


Figure 16 Decision tree for streaming classification and tree induction

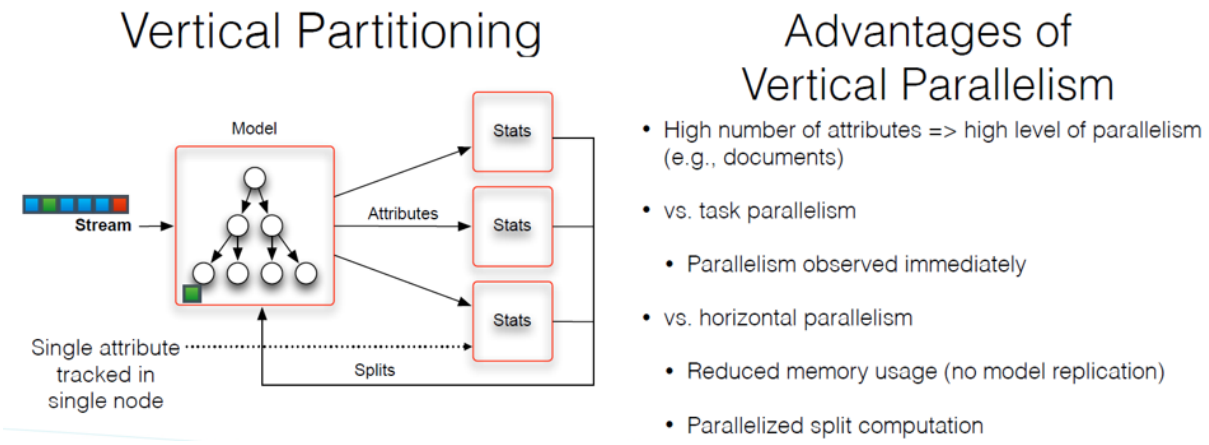


Figure 17 Decision tree for streaming classification using Vertical Hoeffding Trees

10.5 Adaptive Model Rules Regressors

Given a set of training examples with a numeric label, **a regression algorithm** builds a model that predicts for every unlabeled instance x the value y=f(x) with high accuracy.

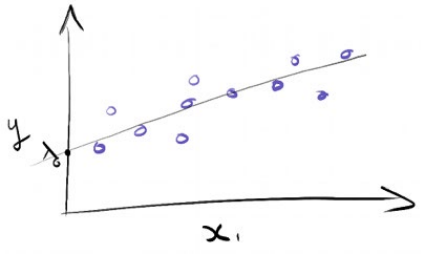


Figure 18 Typical regression problem

Regression analysis is a technique for estimating a **functional relationship** between a **dependent variable** and a **set of independent variables**. It has been widely studied in statistics, pattern recognition, machine learning and data mining. The most expressive data mining **models for regression** are **model trees** and **regression rules**.

Model trees and model rules are among the most performant ones. Trees and rules do **automatic feature selection**, being **robust to outliers** and **irrelevant features**; exhibit high degree of **interpretability**; and **structural invariance** to monotonic transformation of the independent variables. One important aspect of rules is **modularity**: each rule can be interpreted per se.

The **AMRules algorithm**, is one of the first **one-pass algorithm** for **learning regression rule sets** from **time-evolving** streams.

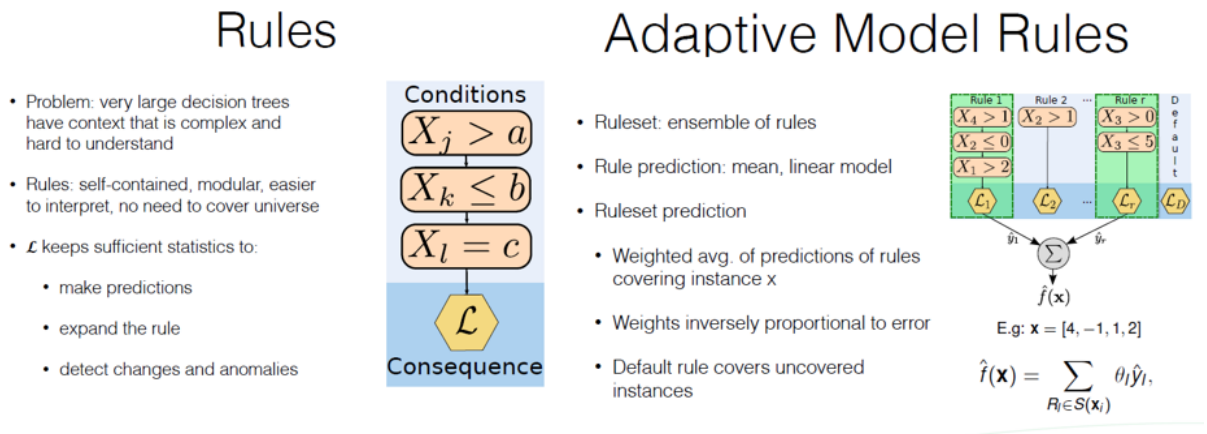


Figure 19 Basics Adaptive Model Rules

The AMRules algorithm is a **one-pass algorithm**, able to adapt the current rule set to changes in the process generating examples. It is able to induce **ordered** and

unordered rule sets, where the consequent of **a rule contains a linear model trained** with the perceptron rule, for example.

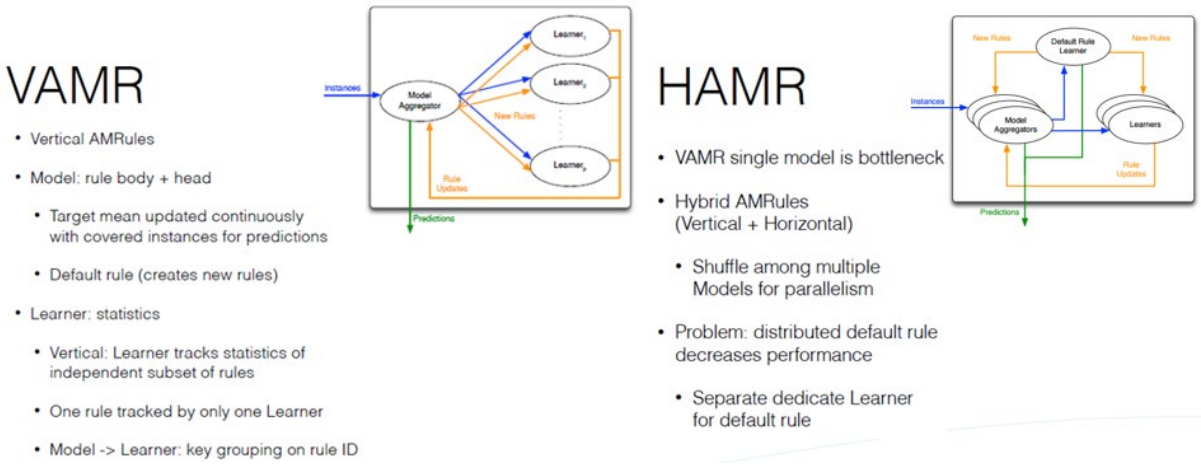


Figure 20 Vertical and Horizontal Adaptive Model Rules

The experimental results point out that, in comparison to ordered rule sets, unordered rule sets are more competitive in terms of performance (MAE and RMSE). AMRules is **competitive** against **batch learners** even for **medium sized datasets**.

A huge **advantage** of **decision rules** is **comprehensibility**, required in many business decision making applications. We begin by **pipelining the processing** of each instances into two steps: **training** and **predicting** and assigning these steps to learner and model aggregator processors in VAMR. This approach has proved to increase the **throughput** for "complex" datasets. Besides, VAMR also provides **memory scalability** as the **memory consumption** of the model (the rule set) is spread among multiple learners. However, VAMR is not scalable in terms of throughput due to the bottleneck at the single model aggregator. To address this issue, the **HAMR**, an extended version of **VAMR** with **multiple replicated model aggregators**. HAMR is shown to be scalable as it can improve the throughput proportionally to the number of model aggregators while maintaining good accuracy.

10.6 Perceptrons in streaming

Recalling from the previous chapters, a **perceptron** is basically a **linear binary classifier**. Its input is a vector x = [x1, x2…, xd] with **real-valued components**. Associated with the perceptron is **a vector of weights** w = [w1, w2…, wd], also with real-valued components. Each perceptron has a **threshold** θ. The output of the perceptron is +1 if w.x > θ, and the output is −1 if w.x < θ. The special case where w.x = θ will always be regarded as "wrong". The **weight vector** w **defines a hyperplane** of dimension d−1 – the set of all points x such that w.x = θ. Points on the positive side of the hyperplane are classified +1 and those on the negative side are classified −1. A **perceptron classifier works only for data that is linearly separable**, in the sense

that there is some hyperplane that separates all the positive points from all the negative points. If there are many such hyperplanes, the perceptron will converge to one of them, and thus will correctly classify all the training points. **If no such hyperplane** exists, then the **perceptron cannot converge** to any particular one. For a regression task the perceptron model and algorithms is presented in Figure 21, along with the **stochastic gradient descent (SGD) algorithm** used to train it.

- Linear regressor
- Data stream: $\langle \vec{x}_i, y_i \rangle$
- $\tilde{y}_i = h_{\vec{w}}(\vec{x}_i) = \vec{w}^T \vec{x}_i$
- Minimize MSE $J(\vec{w}) = \frac{1}{2}\sum(y_i - \tilde{y}_i)^2$
- SGD $\vec{w}' = \vec{w} - \eta \nabla J \vec{x}_i$
  - $\nabla J = -(y_i - \tilde{y}_i)$
  - $\vec{w}' = \vec{w} + \eta(y_i - \tilde{y}_i)\vec{x}_i$

Figure 21 Perceptron learning for regression in streaming

The **extension** from the basic model to **a stream classifier** is depicted in Figure 22 along with the simple algorithmic modifications to allow it to operate in evolving data streams.

- Linear classifier
- Data stream: $\langle \vec{x}_i, y_i \rangle$
- $\tilde{y}_i = h_{\vec{w}}(\vec{x}_i) = \sigma(\vec{w}_i^T \vec{x}_i)$
- $\sigma(x) = 1/(1+e^{-x})$ $\sigma' = \sigma(x)(1-\sigma(x))$
- Minimize MSE $J(\vec{w}) = \frac{1}{2}\sum(y_i - \tilde{y}_i)^2$
- SGD $\vec{w}_{i+1} = \vec{w}_i - \eta \nabla J \vec{x}_i$
  - $\nabla J = -(y_i - \tilde{y}_i)\tilde{y}_i(1-\tilde{y}_i)$
  - $\vec{w}_{i+1} = \vec{w}_i + \eta(y_i - \tilde{y}_i)\tilde{y}_i(1-\tilde{y}_i)\vec{x}_i$

PERCEPTRON LEARNING(*Stream*, η)
1 **for** each class
2 **do** PERCEPTRON LEARNING(*Stream*, *class*, η)

PERCEPTRON LEARNING(*Stream*, *class*, η)
1 ▷ Let w_0 and $\vec{w}$ be randomly initialized
2 **for** each example $(\vec{x}, y)$ in Stream
3 **do if** *class* $= y$
4 **then** $\delta = (1 - h_{\vec{w}}(\vec{x})) \cdot h_{\vec{w}}(\vec{x}) \cdot (1 - h_{\vec{w}}(\vec{x}))$
5 **else** $\delta = (0 - h_{\vec{w}}(\vec{x})) \cdot h_{\vec{w}}(\vec{x}) \cdot (1 - h_{\vec{w}}(\vec{x}))$
6 $\vec{w} = \vec{w} + \eta \cdot \delta \cdot \vec{x}$

PERCEPTRON PREDICTION($\vec{x}$)
1 **return** $\arg\max_{class} h_{\vec{w}_{class}}(\vec{x})$

Figure 22 Perceptron learning for classification in streaming

Induced by ubiquitous scenarios **finite training sets**, **static models**, and **stationary distributions must be** completely **redefined**.

The **characteristics of the streaming** data entail a new vision due to the fact that:

- Data are made available through unlimited streams that continuously flow, eventually at high speed, over time;
- The underlying regularities may evolve over time rather than being stationary;
- The data can no longer be considered as independent and identically distributed;
- The data are now often spatially as well as time situated.

Data streams are a computational challenge to data mining and machine learning problems because of the additional algorithmic constraints created by the large volume and velocity of data. In addition, the problem of temporal locality leads to a number of unique mining challenges in the data stream case, which we tried to cover in the present chapter of the lecture.

Contents

Lectures

11. Immunological Computation and Artificial Immune Systems

Basics

The **vertebrate immune system** (the one which has been used to inspire the vast majority of Artificial Immune Systems to date) is composed of diverse sets of **cells and molecules** that work in **collaboration** with other systems, such as the **neuronal** and **endocrine**, to maintain a steady state of operation within the host: this is termed **homeostasis**.

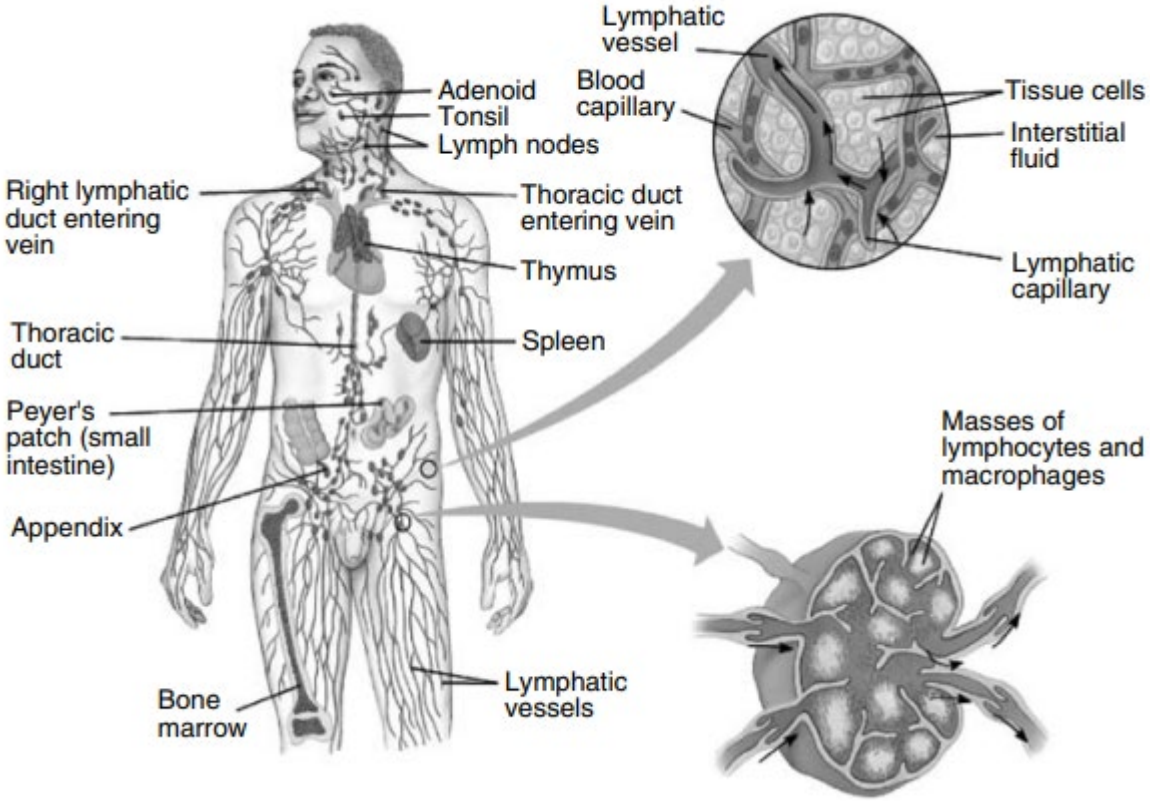
The **role of the Biological Immune System (BIS)** is typically viewed as one of **protection** from **infectious agents** such as viruses, bacteria, fungi and other parasites. On the surface of these agents are **antigens** that allow the **identification** of the **invading agents** (pathogens) by the **immune cells** and **molecules**, which in turn provoke an **immune response**.

There are **two basic types of immunity**, **innate** and **adaptive**. **Innate immunity** is not directed towards specific invaders into the body, but against any pathogens that enter the body. The innate immune system plays a vital role in the **initiation** and **regulation** of **immune responses**, including adaptive immune responses. Specialized cells of the innate immune system evolved so as to recognize and bind to common molecular patterns found only in micro-organisms. However, the innate immune system is by no means a complete solution to protecting the body.

Adaptive, or **acquired immunity**, is **directed against specific invaders**, and cells are modified by exposure to such invaders. The **adaptive immune system** mainly consists of **lymphocytes**, which are white blood cells, more specifically **B and T-cells**. These cells aid in the process of **recognizing** and **destroying specific substances**. Any **substance** that is capable of generating such a response from the lymphocytes is called an **antigen** or **immunogen**. **Antigens** are not the invading microorganisms themselves; they are substances such as **toxins** or **enzymes** in the **microorganisms** that the immune system considers foreign. **Adaptive immune responses** are normally directed against the antigen that provoked them and are said to be **antigen-specific**.

One of the main capabilities of the immune system is to distinguish own body cells from foreign substances, which is called **self / non-self discrimination**. In general, the BIS is capable of recognizing the dangerous elements and deciding an appropriate response while tolerating self-molecules and ignoring many harmless substances.

The immune system is a collection of organs, cells, and molecules responsible for dealing with potentially harmful invaders; it also has other functionalities in the body, as shown in the next extended diagram.



The **organs**, which constitute the immune system, can be classified into central lymphoid organs and peripheral lymphoid organs. The purpose of **central lymphoid organs** is to **generate** and **assist mature immune** cells (**lymphocytes**). Such organs include the bone marrow and the thymus. However, **peripheral lymphoid organs (e.g. lymph nodes, the spleen)** facilitate the interaction between lymphocytes and antigens, as the antigen concentration increases in these organs.

Bone Marrow In an abstract sense, **naive immune cells** are initially generated in the bone marrow. These stem cells divide into either **mature immune cells** (to perform immunological function) or **precursors** of **cells** that migrate out of the bone marrow to continue their maturation process elsewhere. **B-cells** are **produced** in the bone marrow along with other red blood cells and platelets.

Thymus In simple terms, the function of the **thymus** is to produce **mature T cells**. Through a **maturation process**, sometimes referred to as **"thymic education",** T cells that are **beneficial** to the immune system are **kept**, whereas those **T cells** that might cause a detrimental **autoimmune response** are **eliminated.**

Spleen The spleen is an organ, which is **made up of B cells, T cells**, macrophages, dendritic cells, natural killer cells, and red blood cells. An immune response is initiated when macrophages or dendritic cells present the antigen to the appropriate B or T cells. This organ can be thought of as an **immunological "conference center".**

Lymph Nodes The function of **lymph nodes** is to act as an **immunologic filter** for the fluid known as **lymph**. Lymph nodes are found throughout the body and they are mostly composed of T cells, B cells, dendritic cells, and macrophages.

The **immune system** is composed of a variety of **cells** and **molecules**, which **interact among themselves** to achieve appropriate **immunological responses** (biological defense).

Lymphocytes, T Lymphocytes, and B Lymphocytes **White blood cells**, also called **lymphocytes**, are very important constituents of the immune system. These cells are produced in the **bone marrow**, circulate in the **blood** and **lymph system**, and reside in various lymphoid organs to perform **immunological functions**. The primary lymphoid organs provide sites where lymphocytes mature and become antigenically committed. **B and T cells constitute the major population of lymphocytes**.

T cells are specialized cells of the immune system, which are matured in the thymus. B cells are another important class of immune cells, which can recognize particular antigens. There are billions of these cells circulating the body, constituting an effective and distributed anomaly detection and response system. **Antibodies (Abs)** are a particular kind of molecules, called **immunoglobulins** found in the blood and **produced by mature B cells**, also known as **plasma cells**.

Macrophages **Macrophages** are specialized cells, which engulf large particles such as bacteria, yeast, and dying cells by a process called **phagocytosis**. When a **macrophage ingests a pathogen**, the pathogen becomes trapped in a food vacuole, which then fuses with a lysosome. Enzymes and toxic oxygen compounds **digest** the **invader** within the lysosome.

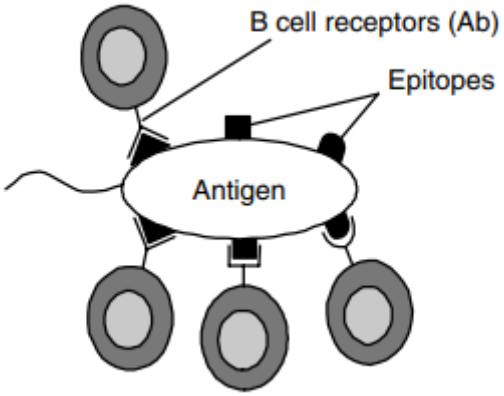
Dendritic cells **Dendritic cells** are immune cells that form part of the **mammal immune system**. These cells are present in small amounts in those tissues that are in contact with the **external environment** such as the **skin** and the inner covering of **nose**, **lungs**, **stomach**, and **intestines**.

Immune System Dynamics

The dynamics of the BIS are provided by a series of processes.

Immune Recognition: Matching and Binding

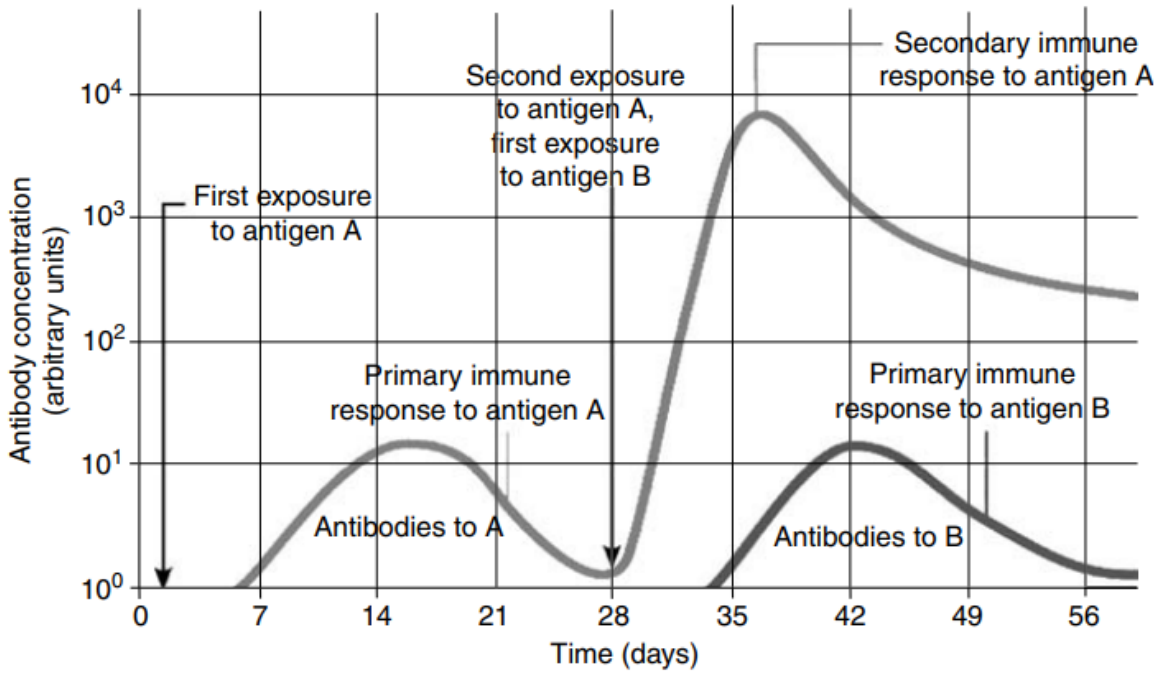
Several immunological processes require an element (cell or molecule) of the immune system to recognize the presence of another element. T cell recognition is based on the complementarity between the binding region of the cell molecule and the receptor. For instance, antigens are detected when a molecular bond is established between the antigen and receptors on the surface of B cells, as shown in the next figure.



Because of the large size and complexity of most antigens, only parts of the antigen, discrete sites called **epitopes**, get **bound to B cell receptors**. Multiple receptors bind to an antigen with **varying affinity**, that is, the more complementary the structures of the epitope and the B cell receptor are, the more likely for a stronger bond to occur.

The **response to the presence of antigens** is composed of **two interlinked mechanisms**: **innate immunity and adaptive immunity.**

When the immune system has been **exposed to an antigen for a second time**, it **reacts quickly** and **rigorously** (measured by the production of antibodies). This is called **secondary immune response**, in contrast to the **first encounter** with the antigen, in which a slower response, called **primary immune response** occurs, as shown in the following diagram.



This **augmented antibody response** is due to the existence of **memory cells**, which rapidly produce plasma cells on antigen stimulation. Thus, the **immune system**

learns from encounters with antigens to **improve its response** in subsequent encounters, producing a so-called **immunological memory.**

Immunological computation

From the point of view of **information processing**, the biological immune system exhibits many interesting characteristics; some of which are:

Pattern matching The **immune system** is able to recognize **specific antigen**s and generate **appropriate responses**. This is accomplished by a **recognition mechanism** based on **chemical binding** of receptors and antigens. This binding depends on their **molecular shapes** and **electrostatic charges**.

Feature extraction Generally, **immune receptors** do not bind to a complete antigen, but rather to **portions of it (peptides)**. Accordingly, the immune system can recognize an antigen just by **matching segments** of it. Peptides are presented to lymphocyte receptors by Antibody Presenting Cells (APC). Therefore, such APCs act as **filters** that can **extract** the important **information** and **remove** the **molecular noise**.

Learning and memory A major feature of the **adaptive immune system** is that it is able to **learn through its interaction with the environment**. The first time an antigen is detected, a **primary response** is induced, which includes **proliferation** and subsequent reduction of lymphocytes. Some of these lymphocytes are kept as **memory cells**. The **next time** the same antigen is detected, **memory cells generate a faster** and more **intense response** (**secondary response**). Accordingly, **memory cells** work as an **associative (highly) distributed memory**.

Diversity **Clonal selection and hypermutation mechanisms** are constantly testing **different detector configuration** for known and unknown antigens. This is a **highly combinatorial process** that explores the **space of possible configurations** looking for **close-to-optimum receptors** that can cope with all types of antigens. **Exploration** is balanced with **exploitation** by favoring the **reproduction of promising individuals**.

Distributed processing Unlike the nervous system, the **immune system is not centrally controlled**. **Detection** and **response** can be **executed locally and immediately** without communicating with any central organ. This **distributed behavior** is accomplished by billions of immune molecules and cells that circulate around the blood and lymph systems and are capable of **making decisions** in a **local collaborative environment**.

Self-regulation Depending on the **severity of the attack**, response of the immune system can range from very light and almost imperceptible to very strong. A **stronger response uses a lot of resources** to help ward off the attacker. Once the invader is

eliminated, the immune system **regulates itself** to **stop** the delivery of **new resources** and **release** the **used ones**.

Self-protection By protecting the body as a whole, the **immune system** is also **protecting itself**. It means that there is **no other additional system** to protect the immune system; hence, it can be said that the **immune system is self-defending**.

Given these features and there have been developed a series of **Immunity-Based Computational Models** and based on the **Specific Immunological Concepts** we introduced in the Basics section. These are summarized in the following table with references to seminal work.

Immunological Concepts and Entities	*Immunity-Based Models*	*Computational Problem*
Self or nonself recognition T cell	Negative selection algorithms (Forrest et al., 1994)	Anomaly, fault, and change detection
Idiotypic network, immune memory, and B cell	Immune network theory (Hunt and Cooke, 1995)	Learning (supervised and unsupervised)
Clonal expansion, affinity maturation, and B cell	Clonal selection algorithm (De Castro and Von Zuben, 2000)	Search and optimization
Innate immunity	DT (Aickelin and Cayzer, 2002)	Defense strategy

Common terminologies that are used in most **immune algorithms** and their **corresponding terms** used in **machine learning** are listed in the following table.

Machine Learning	*Immune Models*
Detectors, clusters, classifiers, and strings	T cells, B cells, and antibodies
Positive samples, training data, and patterns	Self-cells, self-molecules, and immune cells
Incoming data, verifying data samples, and test data	Antigens, pathogens, and epitopes
Distance and similarity measures	Affinity measure in the shape–space
String-matching rule	Complementary rule and other rules

Artificial Immune Systems

Artificial Immune Systems (AIS) is a diverse **area of research** that attempts to **bridge** the divide between **immunology** and **engineering** and are developed through the application of techniques such as **mathematical and computational modeling of immunology**, abstraction from those models into **algorithm** (and **system**) **design** and **implementation** in the context of engineering. AIS has become known as an **area of computer science and engineering** that uses **immune system metaphors** for the creation of novel solutions to problems.

In this section we **outline** a few of the **basic immune algorithms**. We provide **pseudocode** and an **outline description.**

Algorithms

Negative Selection

The process of deleting self-reactive lymphocytes is termed clonal deletion and is carried out via a mechanism called negative selection that operates on lymphocytes during their maturation. For T-cells this mainly occurs in the thymus, which provides an environment rich in antigen presenting cells that present self-antigens. Immature T-cells that strongly bind these self-antigens undergo a controlled death (apoptosis). Thus, the T-cells that survive this process should be unreactive to self-antigens. The property of lymphocytes not to react to the self is called immunological tolerance

Negative selection algorithms are inspired by the main mechanism in the thymus that produces a set of mature T-cells capable of binding only non-self antigens. The first negative selection algorithm was proposed by Forrest *et al* (1994) to detect data manipulation caused by a virus in a computer system. The starting point of this algorithm is to produce a set of self-strings, S, that define the normal state of the system. The task then is to generate a set of detectors, D, that only bind/recognize the complement of S. These detectors can then be applied to new data in order to classify them as being self or non-self, thus in the case of the original work by Forrest *et al* , highlighting the fact that data has been manipulated. The algorithm of Forrest *et al* produces the set of detectors via the process outlined in below.

```
input   : S_seen = set of seen known self elements
output  : D = set of generated detectors

begin

   repeat
      Randomly generate potential detectors and place them in a set P
      Determine the affinity of each member of P with each member of the self set S_seen
      If at least one element in S recognises a detector in P according to a recognition threshold,
         then the detector is rejected, otherwise it is added to the set of avaliable detectors D
   until  Stopping criteria has been met

end
```

Clonal Selection

According to Burnet's 1959 clonal selection theory, the immune system repertoire undergoes a selection mechanism during the lifetime of the individual. The theory states that on binding with a suitable antigen, activation of lymphocytes occurs. Once activated, clones of the lymphocyte are produced expressing identical receptors to the original lymphocyte that encountered the antigen. Thus a clonal expansion of the original lymphocyte occurs.

This ensures that only lymphocytes specific to an activating antigen are produced in large numbers. The clonal selection theory also stated that any lymphocyte that have antigen receptors specific to molecules of the organism's own body must be deleted during the development of the lymphocyte. This ensures that only antigen from a pathogen might cause a lymphocyte to clonally expand and thus elicit a destructive adaptive immune response. In this sense, the immune system can be viewed as a classifier of antigens into either self-antigen or non-self antigen, with non-self antigen assumed to be from a pathogen and thus needs to be removed from the body.

During the clonal expansion of B-cells (but not T-cells), the average antibody affinity increases for the antigen that triggered the clonal expansion. This phenomenon is called affinity maturation, and is responsible for the fact that upon a subsequent exposure to the antigen, the immune response is more effective due to the antibodies having a higher affinity for the antigen. Affinity maturation is caused by a somatic hypermutation and selection mechanism that occurs during the clonal expansion of B-cells. Somatic hypermutation alters the specificity of antibodies by introducing random changes to the genes that encode for them.

The clonal selection theory has been used as inspiration for the development of AIS that perform computational optimization and pattern recognition tasks. In particular, inspiration has been taken from the antigen driven affinity maturation process of B-cells, with its associated hypermutation mechanism. These AIS also often utilize the idea of memory cells to retain good solutions to the problem being solved. In de Castro and Timmis' book, they highlight two important features of affinity maturation in B-cells that can be exploited from the computational viewpoint. The first of these is that the proliferation of B-cells is proportional to the affinity of the antigen that binds it, thus the higher the affinity, the more clones are produced. Secondly, the mutations suffered by the antibody of a B-cell are inversely proportional to the affinity of the antigen it binds. Utilizing these two features, de Castro and Von Zuben developed one of the most popular and widely used clonal selection inspired AIS called CLONALG, which has been used to perform the tasks of pattern matching and multi-modal function optimization.

When applied to pattern matching, a set of patterns, S, to be matched are considered to be antigens. The task of CLONALG is to then produce a set of memory antibodies, M, that match the members in S. This is achieved via the algorithm outlined below.

```
 input   : S = set of patterns to be recognised, n the number of worst elements to select for removal
 output  : M = set of memory detectors capable of classifying unseen patterns

begin

   Create an initial random set of antibodies, A

   forall  patterns in S   do
      Determine the affinity with each antibody in A
      Generate clones of a subset of the antibodies in A  with the highest affinity.
         The number of clones for an antibody is proportional to its affinity
      Mutate attributes of these clones to the set A , and place a copy of the highest
         affinity antibodies in A  into the memory set, M
      Replace the n lowest affinity antibodies in A with new randomly generated antibodies
   end

end
```

Immune Networks

In 1974, Jerne proposed an immune network theory to help explain some of the observed emergent properties of the immune system, such as learning and memory. The premise of immune network theory is that any lymphocyte receptor within an organism can be recognized by a subset of the total receptor repertoire. The receptors of this recognizing set have their own recognizing set and so on, thus an immune network of interactions is formed. Immune networks are often referred to as idiotypic networks. In the absence of foreign antigen, Jerne concluded that the immune system must display a behavior or activity resulting from interactions with itself, and from these interactions immunological behavior such as tolerance and memory emerge.

```
input   : S = set of patterns to be recognised, nt  network affinity threshold,
          ct  clonal pool threshold, h  number of highest affinity clones, a  number of
          new antibodies to introduce
output  : N = set of memory detectors capable of classifying unseen patterns

begin

   Create an initial random set of network antibodies, N
   repeat

      forall  patterns in  S do
         Determine the affinity with each antibody in N
         Generate clones of a subset of the antibodies in N  with the highest affinity. The number of clones for
            an antibody is proportional to its affinity
         Mutate attributes of these clones to the set A , a and place h  number of
            the highest affinity clones into a clonal memory set, C
         Eliminate all elements of C  whose affinity with the antigen is less than a predefined threshold ct
         Determine the affinity amongst all the antibodies in C  and eliminate those antibodies whose affinity with each
            other is less than the threshold ct
         Incorporate the remaining clones of C  into N
       end

      Determine the affinity between each pair of antibodies in N  and eliminate all antibodies whose affinity
         is less than the threshold nt
      Introduce a random number of randomly generated antibodies and place into N

   end until a stopping criteria has been met
end
```

Real-world Applications

Immunological computation (IC) techniques (or artificial immune systems) have been used as a problem solver in **a wide range of domains** such as **optimization**, **classification**, **clustering**, **anomaly detection**, **machine learning**, **adaptive control**, and **associative memories**.

They have also been used in **conjunction with other methods** (**hybridized**) such as genetic algorithms (GAs), neural networks, fuzzy logic, and swarm intelligence. IC includes **real-world applications** of computer security, fraud detection, robotics, fault detection, data mining, text mining, image and pattern recognition, bioinformatics, games, scheduling, etc.

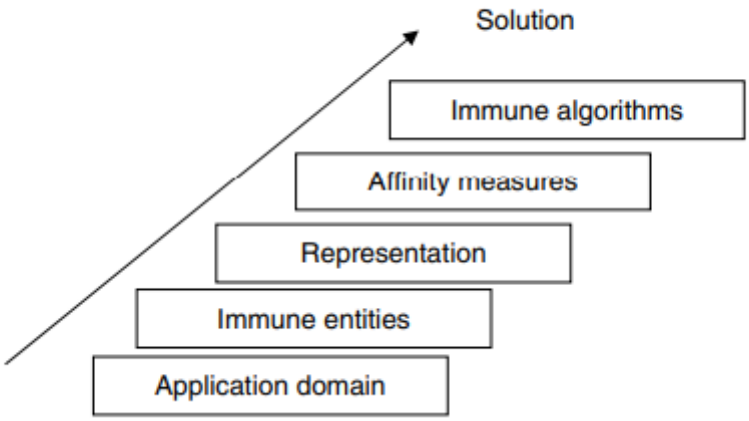
Methodology of applying AIS

To apply an immunity-based model to solve a particular problem in a specific domain, one should select the immune algorithm depending on the type of problem that needs to be solved.

Accordingly, the **first step** should be to **identify the elements** involved in the problem and how they can be **modeled as entities** in a particular AIS. To **encode** such entities, a **representation scheme** for these elements should be chosen, such as a **string representation**, **real-valued vector**, or **hybrid representation**.

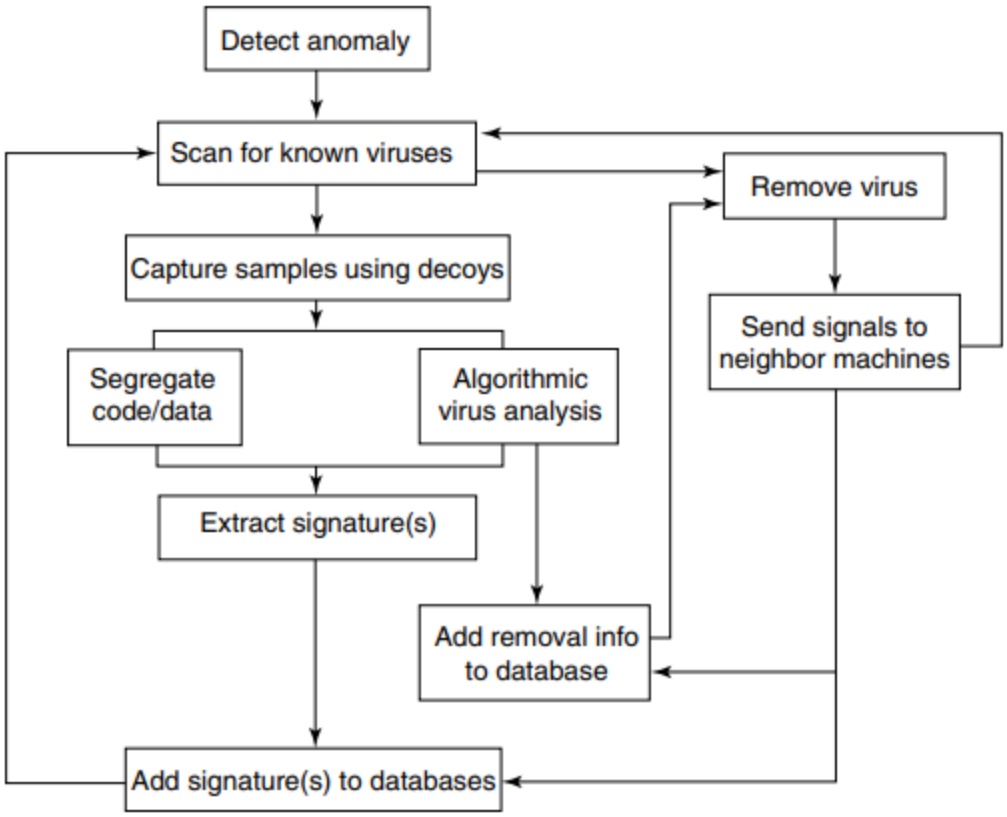
Subsequently, **appropriate affinity/distance measures**, which are to be used to determine corresponding **matching rules**, should be defined.

The next step should be to **decide** which AIS will be better to **generate a set of suitable entities** that can provide a **good solution** to the problem at hand. The following diagram shows the necessary **steps** to **solve problems** using an immunological approach.



Computer security seems to be analogous to the biological defense in many respects; thus we can learn a lesson from the immune system to develop digital

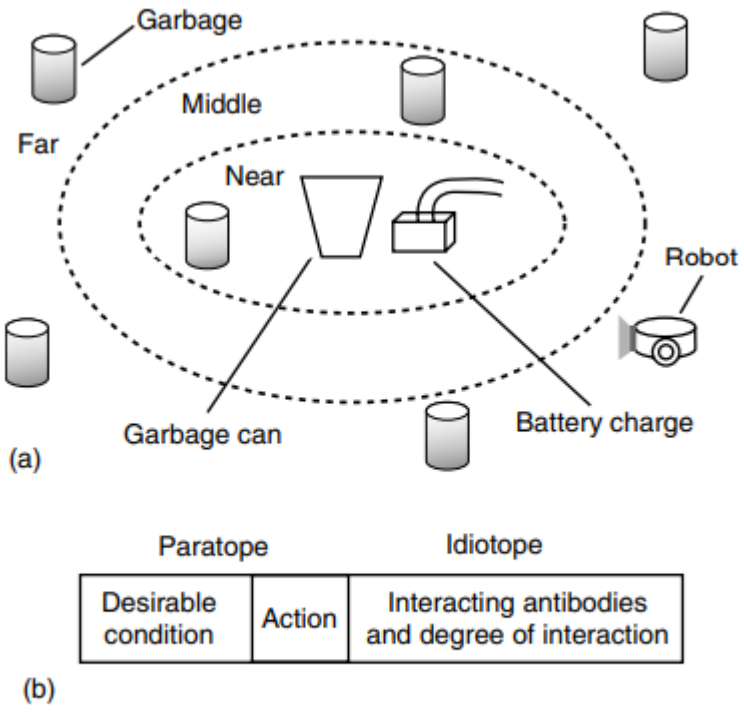
immunity. Majority of AIS works have been devoted to using some immunological metaphor for developing digital defense systems. AISs used varied notions of data protection and anomaly to provide a general-purpose protection system to augment current computer security systems. The security of computer systems depends on activities such as detecting unauthorized use of computer facilities, maintaining the integrity of data files, and preventing the spread of **computer viruses**, following a scheme as the one shown in the following diagram.



Robot control works focused on the development of a dynamic decentralized consensus-making mechanism based on the “immune network theory.” They attempted to create a mechanism by which a single, self-sufficient autonomous robot, called the immunoid, could perform the task of collecting various amounts of garbage from a constantly changing environment.

For the immunoid to make the best decision, it detects antigens and matches the content of the antigen with a selection of all the antibodies that it possesses. Their model included the concepts of “dynamics,” responsible for the variation of the concentration level of antibodies, and “metadynamics,” which maintained the appropriate repertoire of antibodies.

The authors used the metaphors of antibodies, which were potential behaviors of the immunoid; antigens corresponded to environmental inputs such as existence of garbage, wall, and home bases, as shown in the following diagram.



(a) Environment for testing the consensus-making algorithm based on immune networks, (b) definition of antibody.

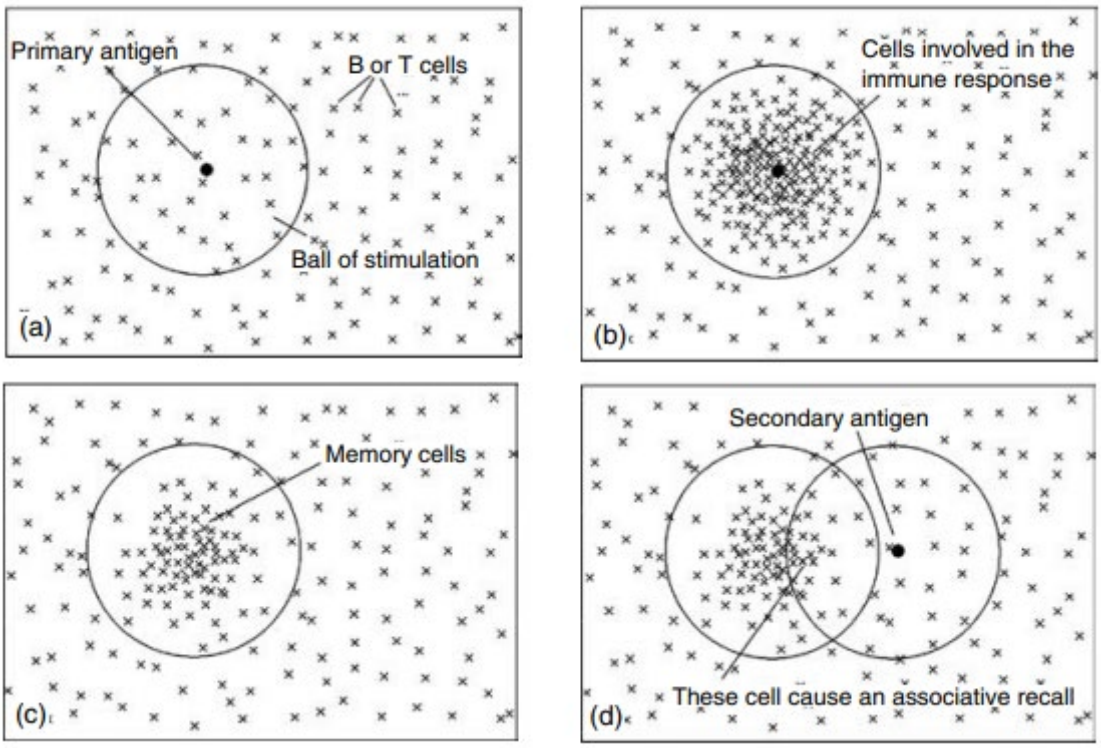
The field of fault diagnosis needs to accurately predict or recover from faults occurring in plants, machines such as refrigeration systems, communications such as telephone systems, and transportations such as aircrafts.

Active diagnosis continually monitors for consistency between the current states of the system with respect to the normal state. Each sensor can be equated with a B cell, connected through the immune network with each sensor maintaining a time-variant record of sensory reliability, thus creating a dynamic system.

An AIS technique was applied to refrigerated cabinets in supermarkets to detect the early symptoms of icing up.

An aircraft fault-detection system, called multilevel immune learning detection (MILD), was developed to detect a broad spectrum of known as well as unforeseen faults. Empirical study was conducted with datasets collected through simulated failure conditions using National Aeronautics and Space Administration (NASA) Ames C-17 flight simulator. Three sets of in-flight sensory information—namely, body-axes roll rate, pitch rate, and yaw rate were considered to detect five different simulated faults: one for engine, two for the tails, and two for the wings. The MILD implemented a real-valued negative selection (RNS) algorithm, where a small number of specialized detectors (as signatures of known failure conditions) and a set of generalized detectors (for unknown or possible faults) are generated.

Researchers argued that the **immunological memory** is a member of the family of **sparsely distributed memories**, and it derives **associative** and robust properties from a sparse and distributed nature of sampling. The following figure illustrates the formation of immune memory (as the concentration level of various immune cells) during the primary and secondary responses.



Another interesting application of **AIS is in gaming**. More precisely, for example, to the problem of playing knots and crosses. In this system, each B cell corresponded to a particular board state containing a nine-digit antibody. The good moves from one state to another meant that those two B cells would have strong affinity or a connection in the B cell network. Later, this group also applied this algorithm to the domain of case-based reasoning. In this system, each case is represented by a B cell object and the case memory is built with the B cell network, with similar cases being linked together. The memory was self-organizing in nature.

In a highly relevant AIS application the CLONALG algorithm was used for **software testing**. Generated test datasets are evaluated using the mutation testing adequacy criteria and are used to direct the search of new tests. Mutation testing generates versions of a program containing simple faults and then finds tests to indicate the program's symptoms. The developed immune system for mutation testing is based on the clonal selection algorithm.

Contents

Lectures

12. Neuromorphic Systems and Spiking Neural Networks

Basics of neuromorphic systems

Neuromorphic engineering is concerned with the design and fabrication of artificial neural systems whose architecture and design principles are based on those of biological nervous systems. Neuromorphic systems of neurons and synapses can be implemented in the electronic medium CMOS (complementary metal oxide semiconductor) using hybrid analog/digital VLSI (very large-scale integrated) technology.

The concept roots at Caltech during the mid-1980s, this time in the research of Carver Mead, who had already made major conceptual contributions to the design and construction of digital VLSI systems. He recognized that the use of transistors for computation had changed very little from the time when John Von Neumann first proposed the architecture for the programmable serial computer.

The design of biological neural computation is very different from that of modern computers. Neuronal networks process information using energy-efficient, asynchronous, event-based methods. Biology uses self-construction, self-repair, and self-programming, and it has learned how to flexibly compose complex behaviors from simpler elements. Of course, these biological abilities are not yet understood. But they offer an attractive alternative to conventional technology and have enormous consequences for future artificial information processing and behavior systems.

The challenge for neuromorphic engineering is to explore the methods of biological information processing in a practical electrical engineering context.

Digital and Analog in Neuromorphic VLSI Systems

The majority of integrated circuits represent numbers as binary digits. Binary digits are used because it is possible to standardize the behavior of transistors so that their state can be determined reliably to a single bit of accuracy. The reliable bits can then be combined to encode variables to an arbitrarily high precision.

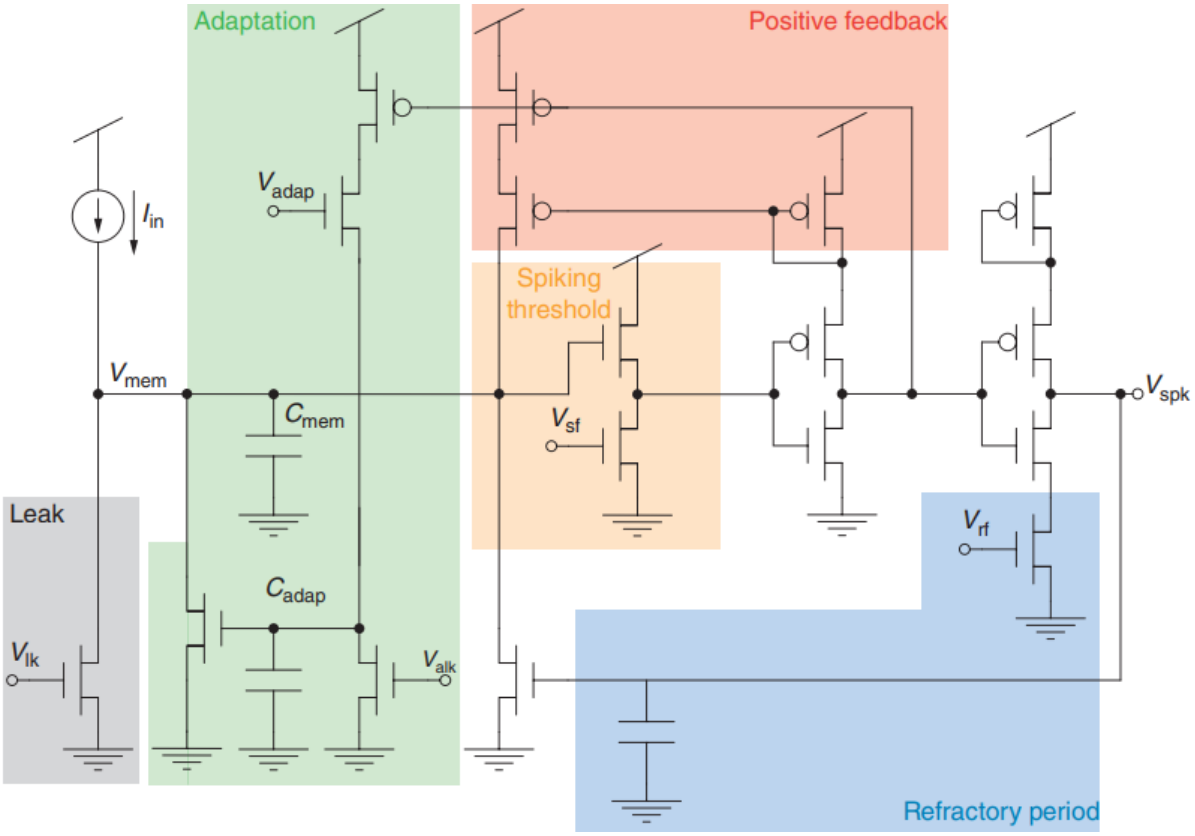
For many problems, particularly those in which the input data are ill-conditioned and the computation can be specified in a relative matter, biological solutions are many orders of magnitude more effective than those that engineers have been able to implement using digital methods. This advantage is due principally to biology’s use of elementary physical phenomena as computational primitives and to the representation of information by the relative values of analog signals rather than by the absolute values of digital signals. Typically, it is this style of processing that neuromorphic engineers explore. Their systems are large collections of communicating computational primitives implemented either in analog or, more commonly, in hybrid analog–digital circuits.

Neurons in Silicon

One of the central goals of neuromorphic engineering is to capture the computational principles of neurons and their networks in hardware. A cornerstone of this quest has been the development over the last decade of hybrid analog–digital VLSI neurons, together an infrastructure for composing networks of these neurons. It is now possible to assemble quite complex systems of such neurons.

Integrate-and-Fire Models

Neurons communicate by pulses (or spikes) that propagate along electrically lossy point-to-point wires that are the axons. Real neurons have a complex morphology and even more complex biophysics, whose full emulation is beyond the reach of present electronic technology. Nevertheless, the integrate-and-fire neuron (I&F), which is a bold simplification of real neurons, has proved to have significant explanatory power in understanding the behavior of neuronal networks both in theory and simulation. A Hardware depiction of such an analog I&F neuron in provided in the following diagram.



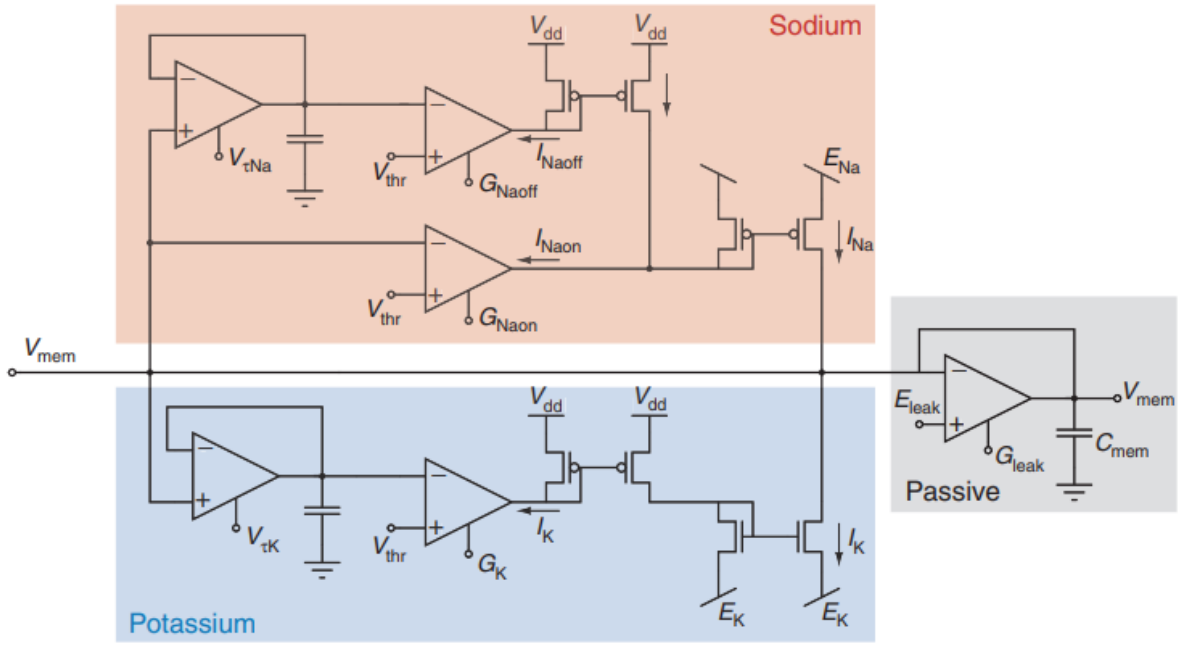
The input current Iin is integrated on to the neuron's membrane capacitor Cmem until the spiking threshold is reached. At that point the output signal Vspk goes from zero to the power supply rail, signaling the occurrence of a spike. Then the membrane capacitor is reset to zero, and the input current starts to be integrated again. The leak module implements a current leak on the membrane. The spiking threshold module controls the voltage at which the neuron spikes. The adaptation module subtracts a firing rate dependent on current from the input node. The amplitude of this current

increases with each output spike and decreases exponentially with time. The refractory period module sets a maximum firing rate for the neuron. The positive feedback module is activated when the neuron begins to spike and is used to reduce the transition period in which the inverters switch polarity, dramatically reducing power consumption. The circuit's biases (Vlk, Vadap, Valk, Vsf, and Vrf) are all subthreshold voltages that determine the neuron's properties.

Conductance-Based Models

These VLSI I&F neurons provide convenient approximations of the behavior of neuronal soma without committing to the overhead of emulating the plethora of voltage-dependent conductances and currents present in real neurons. But, if necessary, these conductances can be emulated using subthreshold CMOS circuits.

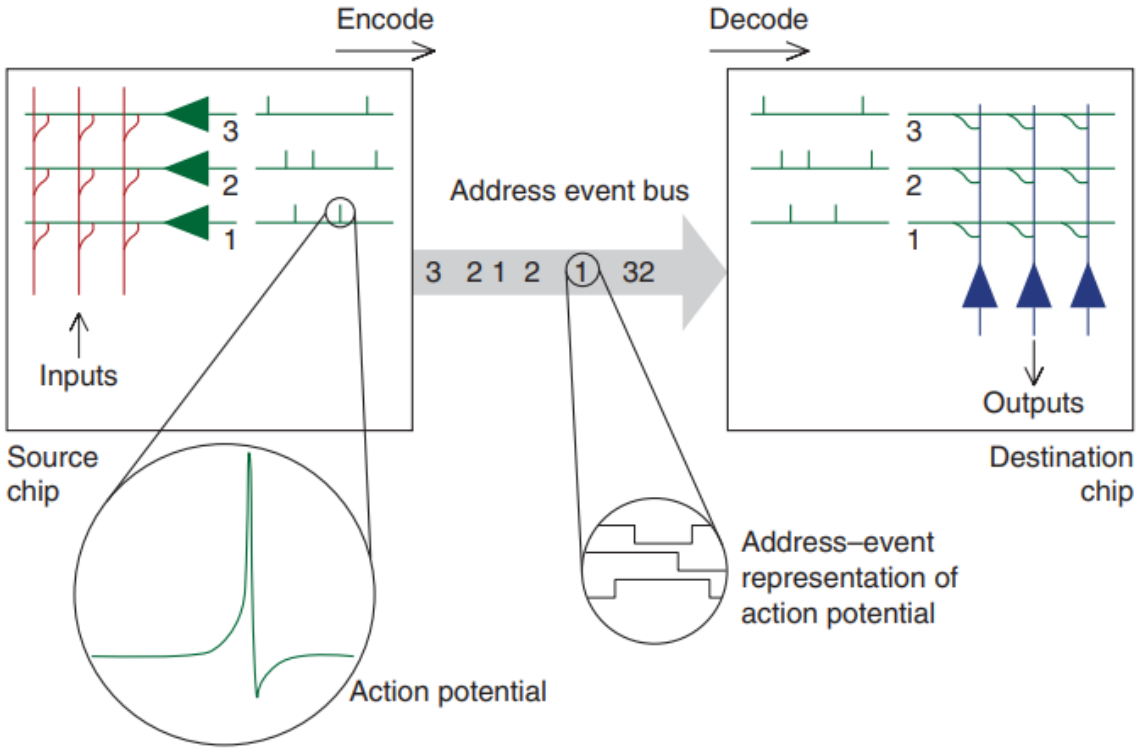
The dynamics of these types of circuits is qualitatively similar to the Hodgkin–Huxley mechanism without implementing their specific equations. An example of this type of silicon neuron circuit is shown in the next figure.



The passive module implements a conductance term that models the passive leak behavior of a neuron; in the absence of stimulation, the membrane potential Vmem leaks to Eleak following first-order low-pass filter dynamics. The sodium module implements the sodium activation and inactivation circuits that reproduce the sodium conductance dynamics observed in real neurons. The potassium module implements the circuits that reproduce the potassium conductance dynamics. The bias voltages Gleak, VtNa, and VtK determine the neuron's dynamic properties, whereas GNaon, GNaoff, GK, and Vthr are used to set the silicon neuron's action potential characteristics.

Axons, Action Potentials, and the Address–Event Representation

Biological neurons communicate with one another using dedicated point-to-point axons. The all-or-nothing action potential can be translated into a discrete level signal, which is robust against noise and inter-chip variability, and can be conveniently transmitted between chips and easily interfaced to standard logic and computer systems. In the Address Event Representation (AER) method, the action potentials generated by a particular neuron are transformed into an address that identifies the source neuron and then broadcast on a common data bus. Many silicon neurons can share the same bus because switching times in CMOS and on the bus are much faster than the switching times of neurons. Events generated by silicon neurons can be broadcast and removed from a data bus at frequencies greater than a megahertz. Therefore, more than 1000 address events could be transmitted in the time it takes one neuron to complete a single action potential. The addresses are detected by the target synapses, which then initiate their local synaptic action as shown in the next figure.



Asynchronous communication scheme between two chips (i.e. artificial neurons) using the address–event representation (AER). When a neuron on the source chip generates an action potential, its address is placed on a common digital bus. The receiving chip decodes the address events and routes them to the appropriate synapses.

Spiking Neural Networks

The hardware simulation of the brain is only one part of the problem. Although our current software and algorithms can operate on multi-processor machines with tens of cores, they are far from being able to run in parallel on brain-inspired machines with hundreds of thousands or millions of cores.

The **Neural Engineering Framework** is a method used for constructing neural simulations, and among one of the most mature and successful approaches for instantiating spiking neural networks in software.

The framework was initially developed for understanding neurobiological systems and is summarized below through the "Principles of Neural Engineering". The creators used these principles to define a methodology for constructing simulations of neural systems. They had great success in applying both the principles and the related methodology to constructing models of perceptual, motor, and cognitive systems.

Principles of Neural Engineering

1. **Neural representations** are defined by the combination of nonlinear encoding (exemplified by neuron tuning curves) and weighted linear decoding.
2. **Transformations** of neural representations are functions of variables that are represented by neural populations. Transformations are determined using an alternately weighted linear decoding (i.e., the transformational decoding as opposed to the representational decoding).
3. **Neural dynamics** are characterized by considering neural representations as control theoretic state variables. Thus, the dynamics of neurobiological systems can be analyzed using control theory.

NENGO

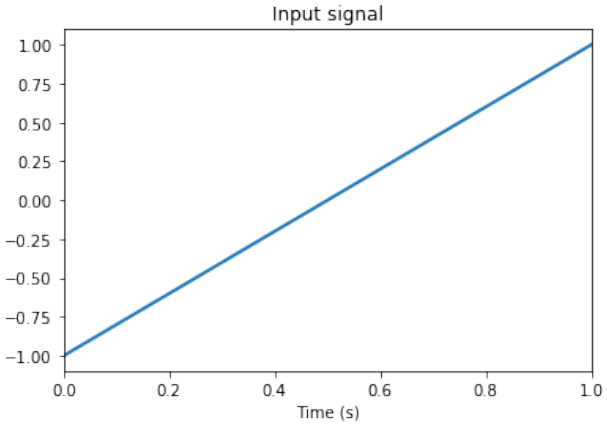
Nengo is a graphical and scripting based Python package for simulating large-scale neural networks that uses the three principles of **Neural Engineering Framework.**

Nengo is highly extensible and flexible. You can define your own neuron types, learning rules, optimization methods, reusable subnetworks, and much more. You can also get input directly from hardware, build and run deep neural networks, drive robots, and even implement your model on a completely different neural simulator or neuromorphic hardware.

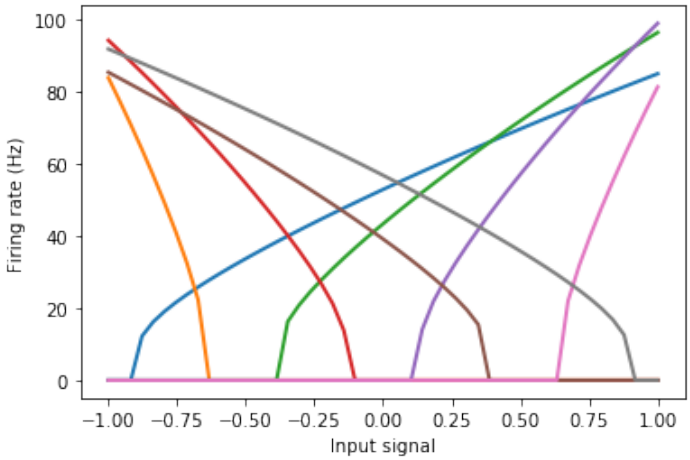
Nengo is a powerful development environment at every scale. Among other things, Nengo is used to implement networks for deep learning, vision, motor control, visual attention, serial recall, action selection, working memory, attractor dynamics, inductive reasoning, path integration, and planning with problem solving. Nengo has libraries specifically designed to help with cognitive modelling, deep learning, adaptive control, and accurate dynamics, to name a few.

Encoding

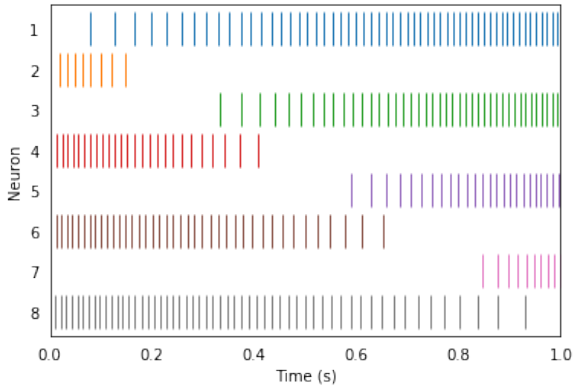
Neural populations represent time-varying signals through their spiking responses. A signal is a vector of real numbers of arbitrary length. This example is a 1D signal going from -1 to 1 in 1 second.



These signals drive neural populations based on each neuron's *tuning curve* (which is similar to the current-frequency curve, if you're familiar with that). The tuning curve describes how much a particular neuron will fire as a function of the input signal.

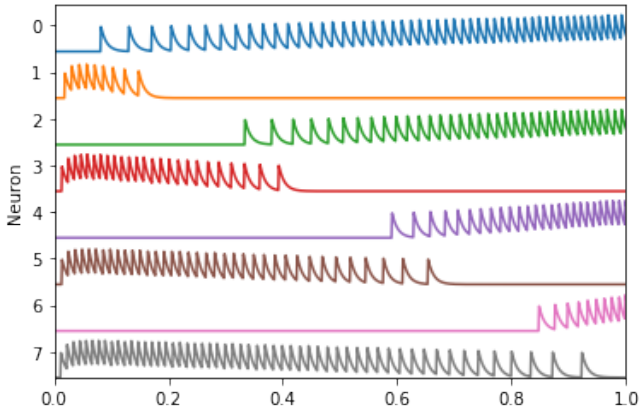


We can drive these neurons with our input signal and observe their spiking activity over time.



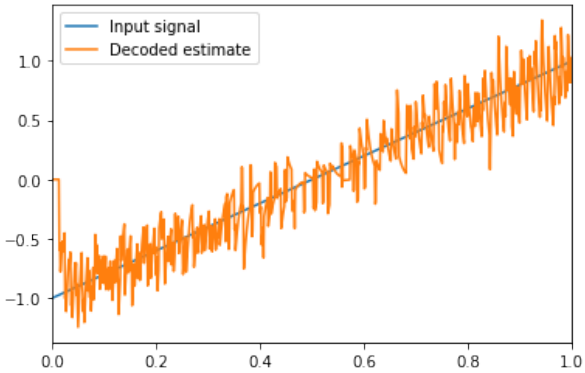
Decoding

We can estimate the input signal originally encoded by decoding the pattern of spikes. To do this, we first filter the spike train with a temporal filter that accounts for postsynaptic current (PSC) activity.



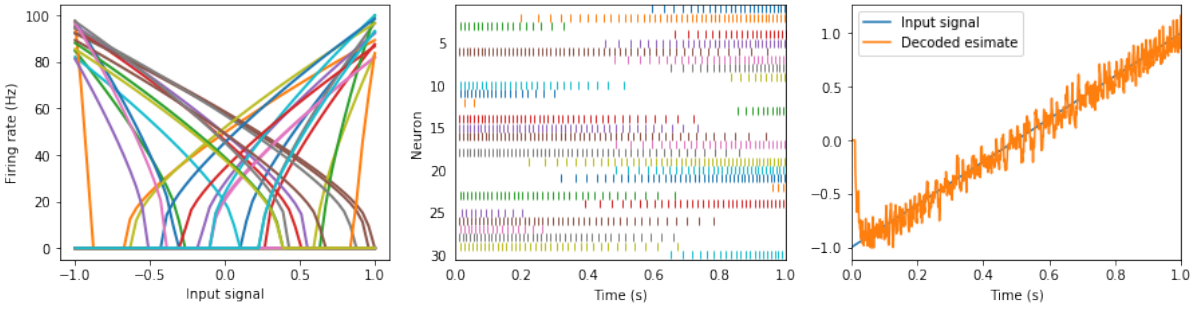
Then we multiply those filtered spike trains with decoding weights and sum them together to give an estimate of the input based on the spikes.

The decoding weights are determined by minimizing the squared difference between the decoded estimate and the actual input signal.

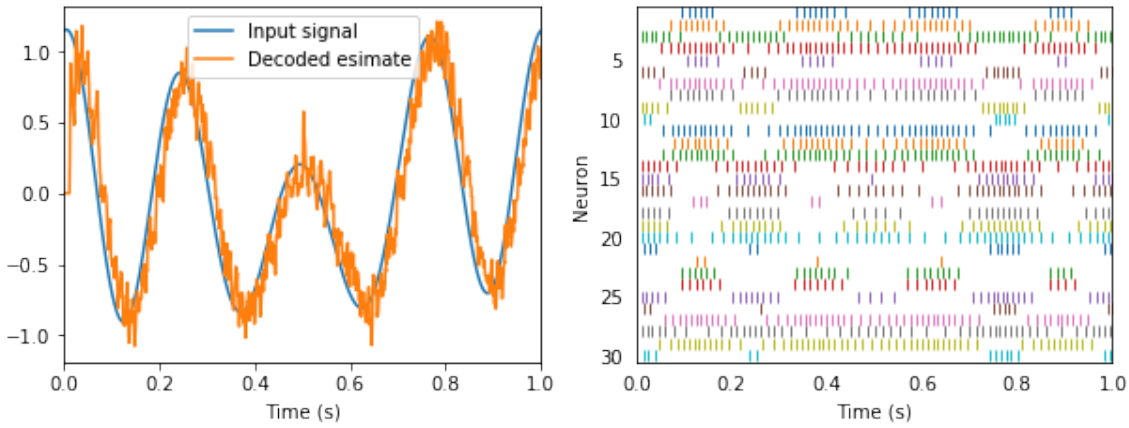


The accuracy of the decoded estimate increases as the number of neurons increases.

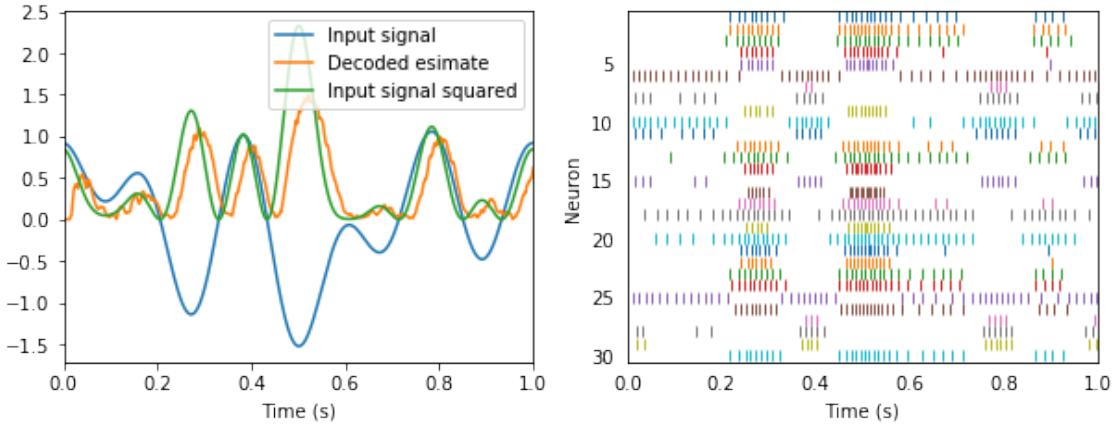
A complete overview can be visualized in the following diagram.



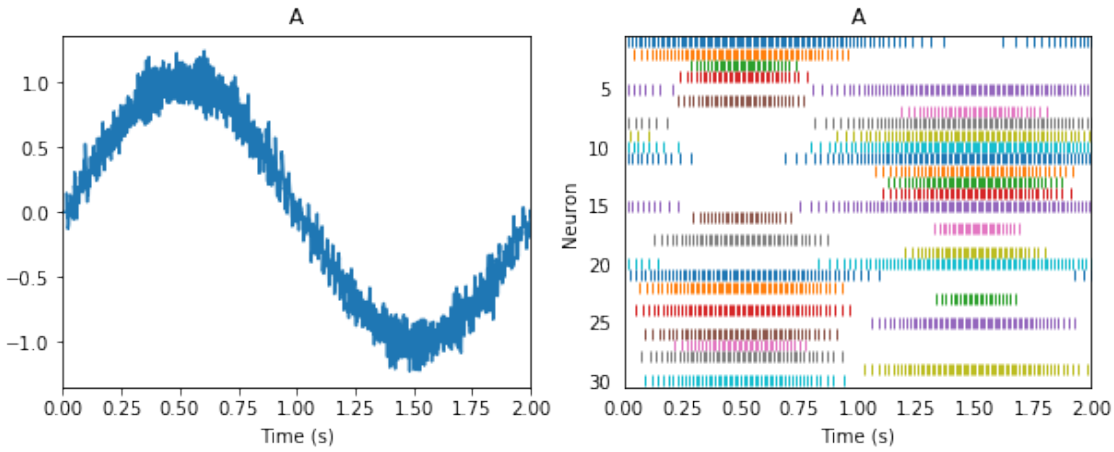
Any smooth signal can be encoded and decoded.



Encoding and decoding allow us to encode signals over time, and decode transformations of those signals. In fact, we can decode arbitrary transformations of the input signal, not just the signal itself (as in the previous example). Let's decode the square of our white noise input.

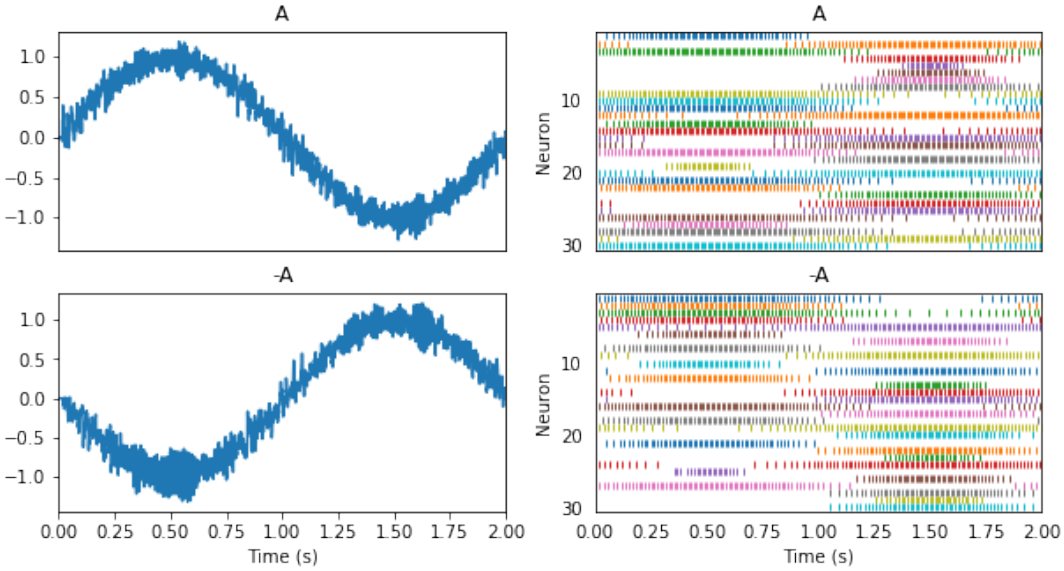


Notice that the spike trains are exactly the same. The only difference is how we're interpreting those spikes. We programmed Nengo to compute a new set of decoders that estimate the function x^2. In general, the transformation principle determines how we can decode spike trains to compute linear and nonlinear transformations of signals encoded in a population of neurons.

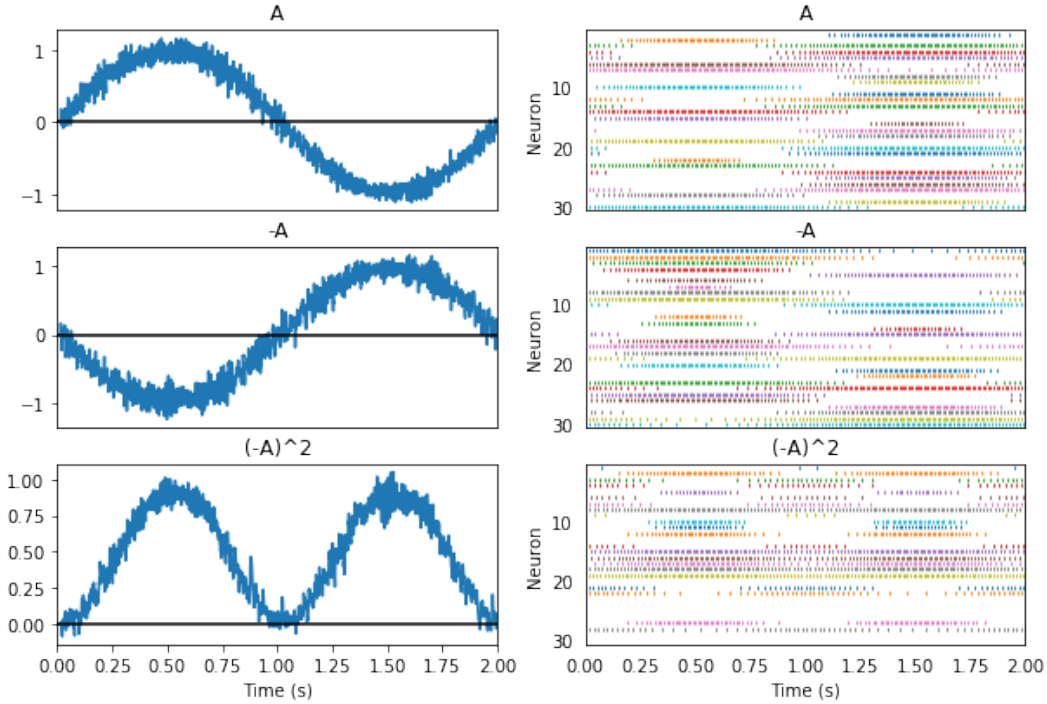


We can then project those transformed signals into another population, and repeat the process. Essentially, this provides a means of computing the neural connection weights to compute an arbitrary function between populations. Suppose we are representing a sine wave.

Linear transformations of that signal involve solving for the usual decoders, and scaling those decoding weights. Let us flip this sine wave upside down as it is transmitted between two populations (i.e. population A and population -A).



Nonlinear transformations involve solving for a new set of decoding weights. Let us add a third population connected to the second one and use it to compute $(-A)^2$



So far, we have been considering the values represented by ensembles as generic "signals." However, if we think of them instead as state variables in a dynamical system, then we can apply the methods of control theory or dynamic systems theory to brain models. Nengo automatically translates from standard dynamical systems descriptions to descriptions consistent with neural dynamics.

In order to get interesting dynamics, we can connect populations recurrently (i.e., to themselves). Below is a simple harmonic oscillator implemented using the third principle. It needs is a bit of input to get it started.