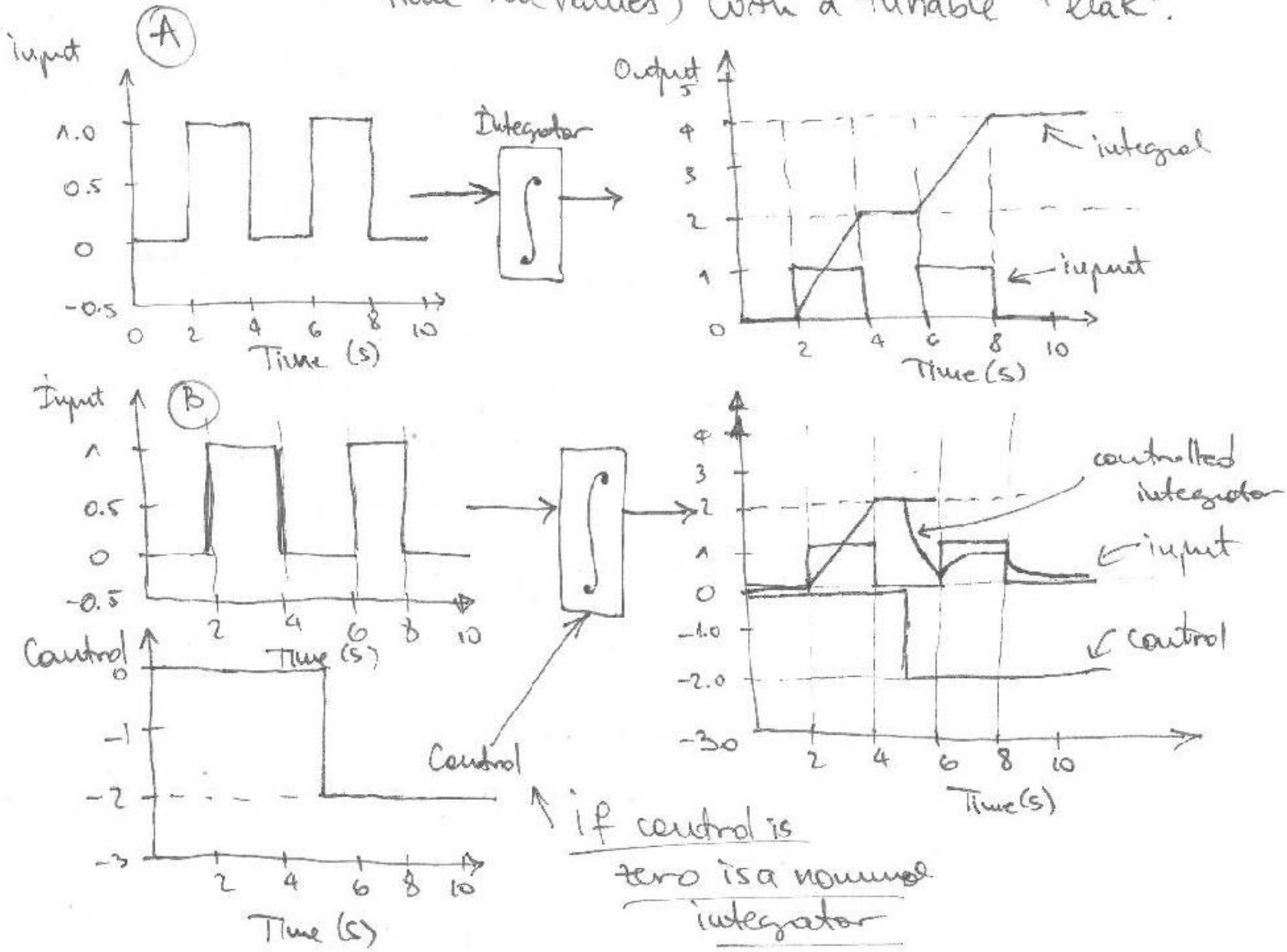
Spiking Neural Computation

- Neural Engineering Framework (NEF), 2003, 2013 Eliasmith, Anderson; (Nengo - software opensource)
- NEF as a general methodology that allows you to build large-scale, biologically plausible, neural models of cognition;
- doesn't make assumptions about what specific functions the brain performs;
- it is a set of 3 principles that help determine how the brain performs some given function;
- NEF is a "neural compiler"
  * if you have a guess about the high-level function of a certain brain area or how neurons there spond NEF provides a way to connect groups of neurons ("populations" or "layers") to realize that function.
- The core of NEF is based on 3 principles:
  1. Representation(s): are defined by the combination of nonlinear encoding and weighted linear decoding.
  2. Transformations: are functions of the variables represented by neural populations.
  3. Dynamics: characterized by considering neural representations as states of a dynamical system => control theory analysis.

- The basic processing unit in NEF is typically a population of neurons, rather than single cells.
- Let's describe the 3 principles through an example: a "controlled integrator".
  * a simple neural circuit functioning as a simple memory
  * it can be loaded with data, hold the information and then erase it
  * this is in fact an integrator (i.e. accumulates in time the values) with a tunable "leak".



↑ if control is zero is a normal integrator

(A). Standard integrator: continuously adds its input to its current state, the mathematical integral of the input.

(B). Controlled integrator: values less than 0 for the control makes the integrator forget (the speed of forgetting is proportional with the control variable).

NEF offers the possibility to solve problems with neurons:

* specify the properties of the neurons;
* specify the values to be represented and the functions to be computed;
* learns / solves for the connection weights between populations that will implement the desired function

NEF works with:

- feed-forward and recurrent connections.
- allows for complex dynamical systems such as integrators, oscillators, Kalman filters.
- incorporates local error-driven learning rules allowing for online adaptation and optimization of responses.

NEF has been used for:

- model visual attention,
- inductive reasoning,
- reinforcement learning,
- building the world's largest functional brain model using 2.5 million neurons to perform eight different tasks by interpreting visual input and producing hand-written output via a simulated 6-muscle arm.

SPAUN ←

↓

RBM Deep Belief Network (vision system)

+

working memory (cortex) including motor port)

+

action selection (basal ganglia)

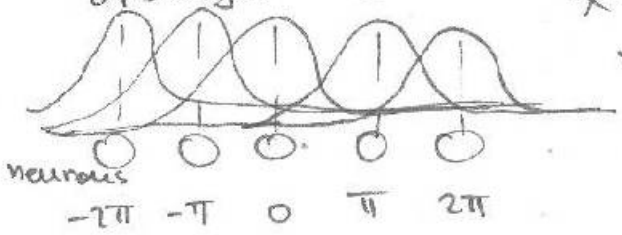
+

information routing (thalamus)

Representation in NEF

- distributed representations.
- differentiates between the activity of neurons in a population and the value being represented.
- the value is usually thought of as a vector x
- to map between value x and activity a every neuron i has an encoding vector e_i

Example:



* the encoding is the preffered direction vector for that neuron: the vector for which the neuron will fire most strongly (tuning curves).

- given a value x, the activity of neuron i is:

$$a_i = G(\alpha_i \cdot e_i \cdot x + b_i)$$

where
- α_i - gain parameter
- e_i - encoding vector → Kernel
- b_i - bias
- G - nonlinear activation function (sigmoid, tanh, or LIF neurons)

- in order to also decode the value represented by a neural population activation we use a linear decoder d_i:

$$\hat{x} = \sum_i a_i \cdot d_i$$

d_i - is a set of weights that maps the activity a_i back into a value, an estimate of x called $\hat{x}$.

- finding d_i is an optimization problem (least-squares minimization), we need to find the set of weights that minimizes the difference between x and its estimate $\hat{x}$:

input mapping to state | internal (population) mapping

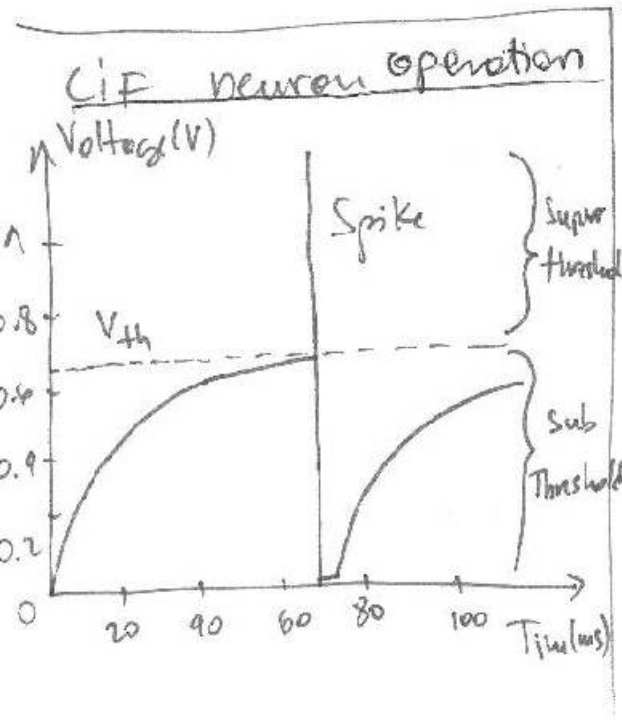
$$\Upsilon_j = \sum_x a_j \cdot x \quad , \quad \Gamma_{ij} = \sum_x a_i a_j$$

$$d = \Gamma_{ij}^{-1} \cdot \Upsilon$$

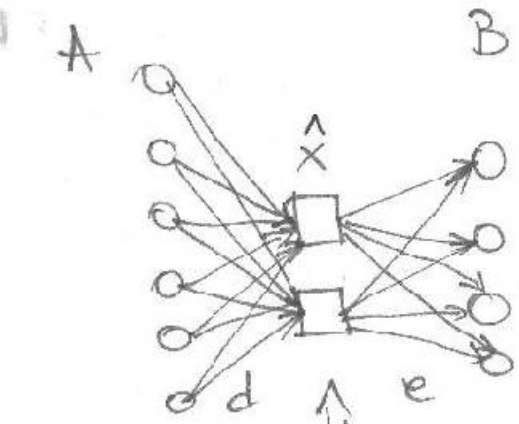
- having this decoder allows you to determine how accurately a group of neurons is representing some value.

Transformation in NEF

- in order to do computations we need to connect neural populations together.
- consider 2 populations A and B and you want to pass information from A to B:
  - * if you set A to represent the value 0.2, then the synaptic connection weights between A and B should cause B to also represent value 0.2; Obviously this is computing the function $f(x) = x$.
  - this cannot be realized by simply connecting neurons in A to neurons in B (might be different numbers) and the nonlinearity G makes this inaccurate.



- To create this in an accurate manner let's first assume we have an intermediate group of perfectly ideal linear neurons as shown next:

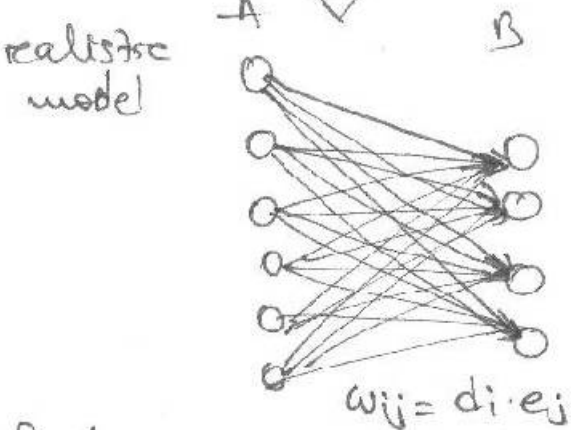


d_i is the set of weights to compute $\hat{x}$

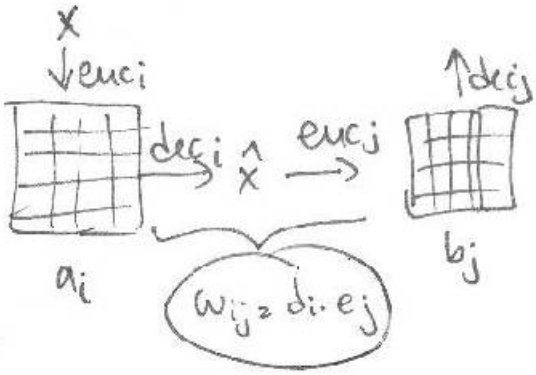
$$\hat{x} = \sum_i a_i d_i$$

using the encoder e of B we can make B represent x

$$a_i = G(d_i e_i \hat{x} + b_i)$$



or



- Such an approach is not limited to computing $f(x)=x$ in fact adjusting the weights $d^{f(x)}$ one can approximate any function $f(x)$.

- This approach can compute nonlinear functions with a single layer of connections → no error back-propagation is required!

$$d^{f(x)} = \Gamma^{-1} \cdot \Upsilon^{f(x)}$$

$$w_{ij}^{f(x)} = \alpha e_j d_i$$

where $\Gamma_{ij} = \sum_x a_i a_j$ and $\Upsilon = \sum_x a_j \cdot f(x)$

Γ: intra population connections

Υ: activity to function mapping

- this is possible because distributed representations allow for complex functions to be computed in a single set of connections ⇒ similar to SVM there is a projection to a high-dimensional space to do computation

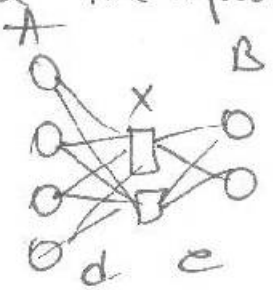
- random variations in α_i and b_i are required in

$$a_i = G(\alpha_i \cdot e_i \cdot X + b_i)$$

because the function $f(x)$ will be approximated out of linear sums of the tuning curves

→ more diverse tuning curves leads to a better approximation!

- such an approach is fast to simulate because instead of multiplying the activity of A by the full weight matrix you multiply by the decoders and then multiply by the encoders to produce the input to B.



⇓

Savings in memory & simulation time!

Dynamics in NEF

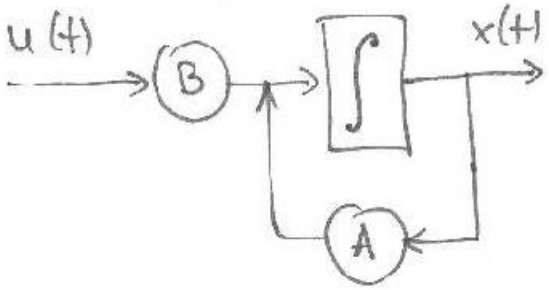
- the representation and transformation are sufficient to produce models capable of computing functions of the form $y = f(x)$.
- NEF also offers the possibility to compute dynamic functions of the form $\frac{dx}{dt} = A(x) + B(u)$

  where x is the value being represented, u is some input, A and B are arbitrary functions

- a special case, is returning to our initial example, the integrator, described by $\frac{dx}{dt} = u \Rightarrow x(t) = \int u(t)dt$

  - if $u = 0$, the system maintains the current state, because $\frac{dx}{dt} = 0$
  - if $u > 0$, the system will store an increasing value
  - if $u < 0$, the system will store a decreasing value

Analogy to control systems:



LTI state equation $\dot{x}(t) = Ax(t) + Bu(t)$

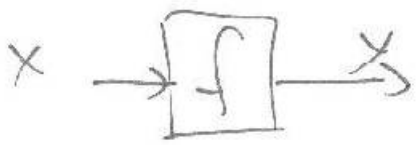
A – determines what aspects of the current state affect the future state

B – maps the input to the state space

$\int$ – the transfer function here is perfect integration

Deep Learning VS. NEF

- connectionist model vs. biologically plausible dynamical model
- basic formulation:



Error, $E = \frac{1}{2}(y-x)^2$

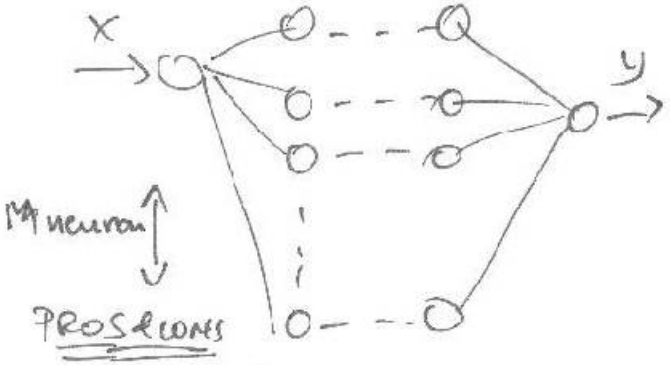
Gradient descent

$$\nabla f = \frac{\partial E}{\partial w}$$

in deep nets

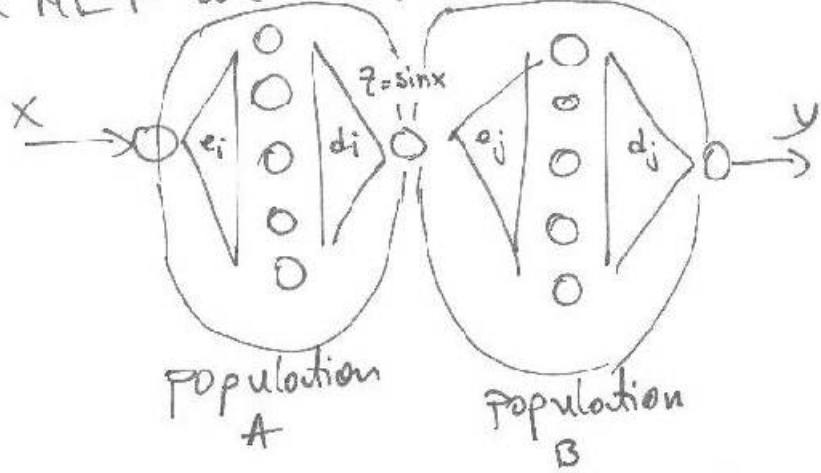
is easy to compute but also error prone, and problematic in large networks.

Another problem is that is computationally expensive.



Let's consider a simple problem of learning $f(x) = \sin^2(x)$

In NEF we would need an architecture as:



PROS & CONS

- this is fast
- interpretable
- require more "design details" by the modeller

Math adden: Least-squares optimization (LSO)

- consider n points (pairs) (x_i, y_i) $i = 1 \dots n$.
- find the parameters of a model to fit the data.
- the model is of the form $f(x, \beta)$ where β contains the m parameters of the model.
- the fit of a model is measured by a residual defined as a difference between the actual value y_i and the prediction $f(x, \beta)$:

$$r_i = y_i - f(x, \beta)$$

- LSO finds the optimal β by minimizing S the sum of squared residuals:

$$S = \sum_{i=1}^{n} r_i^2$$

- Example: linear least squares model

$$f(x, \beta) = \beta_0 + \beta_1 x$$

$$\frac{\partial S}{\partial \beta_j} = 2 \sum_i r_i \frac{\partial r_i}{\partial \beta_j} \quad \text{and because}$$

$r_i = y_i - f(x_i, \beta)$ the gradient equations become

$$-2 \sum_i r_i \frac{\partial f(x_i, \beta)}{\partial \beta_j} = 0 \,, \; j = 1, \dots$$

This method is used to find the decoding weights d_i for $\hat{x} = \sum_i q_i d_i$ in MEF.

Immunological Computation & Artificially Immune Systems

Immunity: the reaction to foreign substances (pathogens) which includes
- * primary response
- * secondary response

Immune System: organs, cells, molecules
coordinated response to a pathogen ⇩ Immune response

Antigens (fungi, bacteria, viruses, other protozoa)
* have the capability to trigger an immune response
* depending on:
  - foreignness
  - molecular size
  - chemical composition

The Biological Immune System (BIS)
- detect and respond to foreign substances
- inherently distributed
- fault-tolerant
- complex behavior through interactions among its components
- self/nonself discrimination

Two types of immunity:

(1) * innate immunity
(2) * adaptive immunity

(1) Innate immunity

- initiates & regulates immune responses

(2) Adaptive immunity (Acquired immunity)

* directed against specific invaders
* consists mainly of lymphocytes (white blood cells):
  - → Bcells
  - → Tcells
* lymphocytes recognize & destroy certain substances
* substances that trigger a lymphocyte response are called antigens / immunogens
* adaptive immune responses are antigen specific!

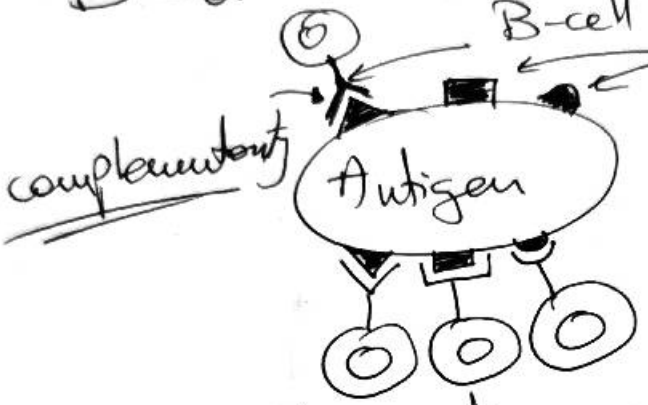
Biological defense mechanisms:

Nonspecific defense mechanism		Specific defense mechanism
Anatomic Barrier	Innate Immunity	Adaptive Immunity
• Skin • Mucous membranes • Secretions of Skin & membranes	• Phagocytic white blood cells • Antimicrobial proteins • Inflammatory response	• Lymphocytes • Antibodies

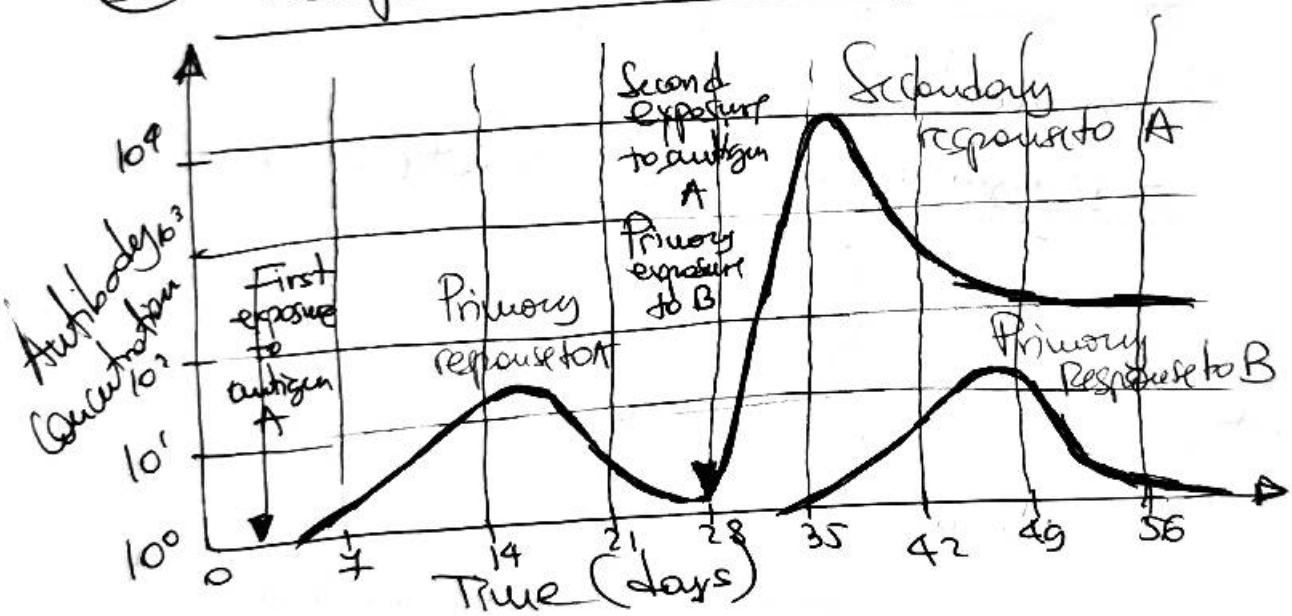
Immune System Dynamics

(A). Immune recognition (matching & binding)

- molecular bonds established between the surface of the antigen and receptors of B- or T-cells (complementary)



(B). Response to antigens



(C) T-cell maturation

* produced initially by bone marrow
* initially inert, they need to go through a maturation process in thymus
* during maturation they develop a T-cell receptor to recognize non-self peptides using selection (positive & negative)

(D) B-cell Proliferation (affinity maturation)

* when binding to an antigen B-cells undergo proliferation & differentiation
* the result are B-cells that are used as:
  - (A) Memory cells: remember previous exposure to antigen to trigger faster response
  - (B) Plasma cells: secrete large quantities of antibodies

Immunological Computation

1. Pattern matching
   - recognize antigens & generate specific responses
   - recognition based on binding ← molecular shape, electrostatic charge
2. Feature extraction
   - immune receptors (B, T cells) bind to sequences/portions of peptides and not to complete antigens (i.e. epitopes)
   - filter to extract matching segments in presence of molecular noise.
3. Learning & memory
   - learns through interaction with environment
   - proliferation of cells to determine memory & adaptive responses

(4.) Diversity

- selection (clonal selection) are testing for new detectors configurations
- highly combinatorial process to find optimal receptors
- exploration & exploitation for reproducing promising individuals

(5.) Distributed Processing

- not centrally controlled (i.e. nervous system) opposite to
- local detection & response
- distributed behavior of molecules & cells that make decisions in local environment

(6.) Self-Regulation

- adaptive response intensity depending on attack strength
- strong attacks need many resources
- after attack regulates to stop generating cells (new resources) and release the used resources.

(7.) Self-Protection

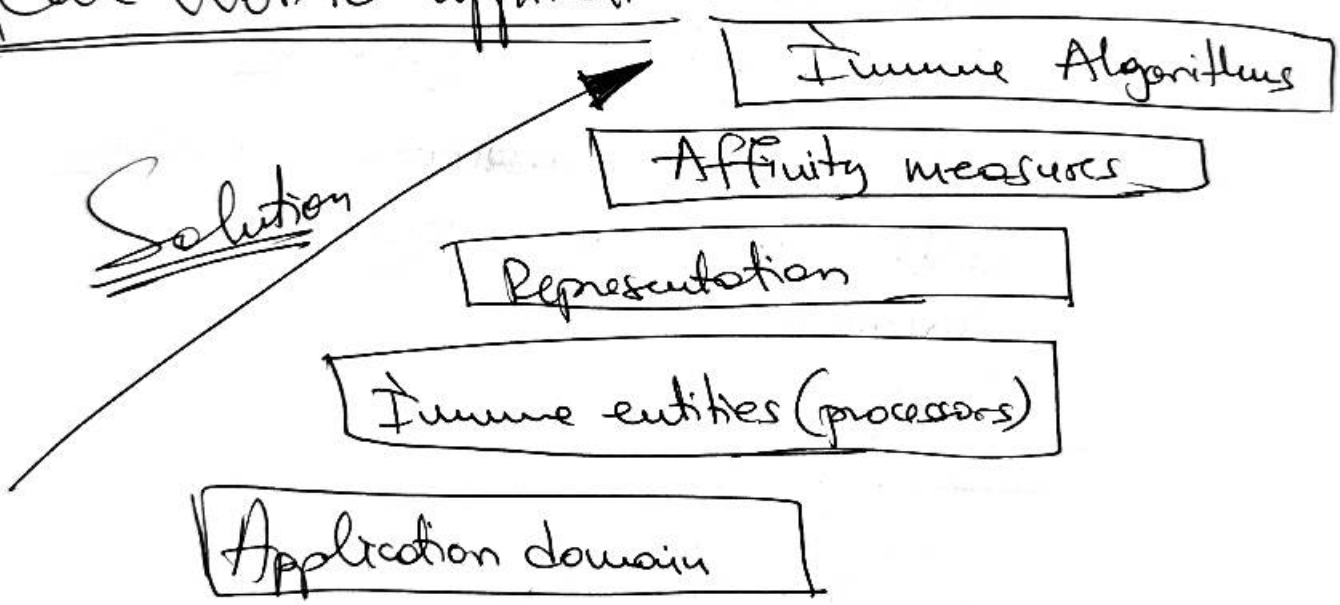
- self-defending anomalies as (auto-immune diseases)

Artificially Immune Systems

- mathematical & computational modelling of immunology
- algorithm design & implementation

1. Negative selection
2. Clonal Selection
3. Immune Networks

Real-World applications



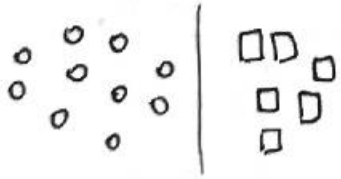
Examples: Computer security, Robot control, Defect/Fault diagnosis, Scaning, Software Testing

Other types of Machine Learning Models and Systems: Statistical & Stochastic ML

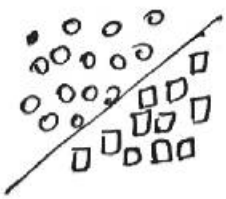
Support Vector Machines (SVM)

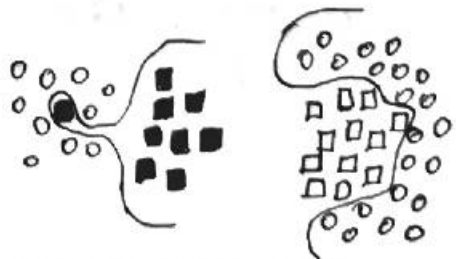
* Motivation:
  - MLPs are simple but they lack optimality and they converge slowly (remember backprop).
  - in a classification problem classes might not be linearly separable, and classes will overlap ⇒ we moved from single neuron to a MLP
  - what if we can construct a nonlinear separation boundary by using a linear boundary in a large, transformed version of the feature space?

- SVMs: • construct a hyperplane as the decision surface in such a way that the margin of seperation between positive & negative examples is maximized for each training sample (binary classification)
  • are feed forward networks, a binary learning machine basically.



Linearly separable classes





Non-linearly separable classes

A. Optimal hyperplane for linearly separable patterns

Consider the training sample $\{(x_i, d_i)\}_{i=1}^{N}$ where x_i is the input pattern for the i-th example and d_i is the corresponding target output (class label)

We assume that the pattern (class) represented by $d_i = +1$ and the pattern $d_i = -1$ are "linearly separable".

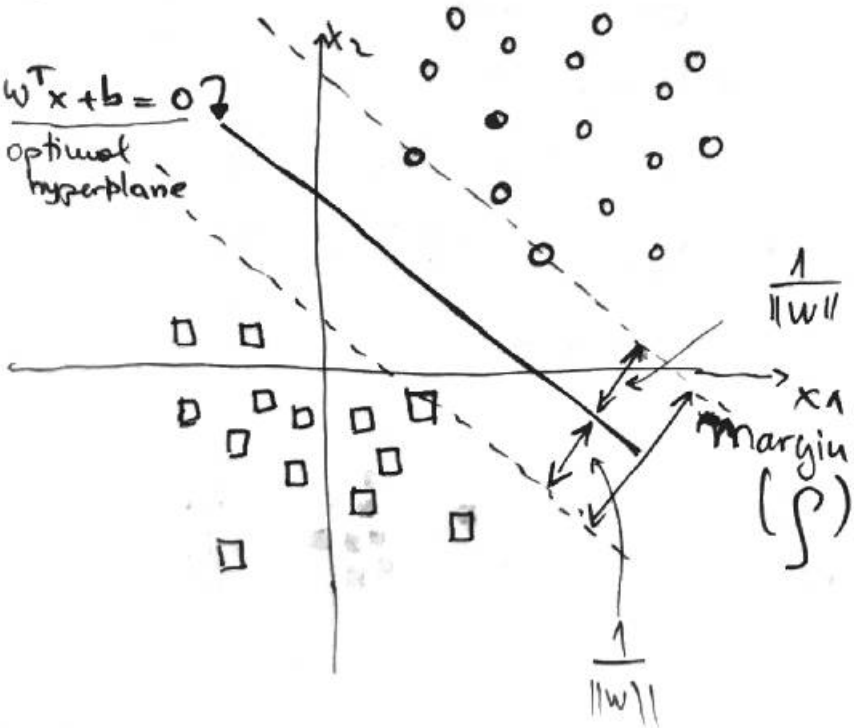
The equation of a decision surface (hyperplane) that does the separation is:

$$\boxed{w^T \cdot x + b = 0}$$

where x = input vector, w = adjustable weight vector, b = bias. So, we can write:

$$\begin{cases} w^T x_i + b \geq 0 & \text{for } d_i = +1 \\ w^T x_i + b < 0 & \text{for } d_i = -1 \end{cases}$$

For a 2D case



The goal of the SVM is to find a particular hyperplane for which the separation margin is maximized!

⇓

find the optimal hyperplane

If we let w_o and b_o be the optimal values of the weight vector and bias, respectively, the optimal hyperplane representing a multidimensional linear decision surface is defined as:

$$w_o^T \cdot x + b_o = 0$$

The discriminant function

$$g(x) = w_o^T \cdot x + b_o$$

gives an algebraic distance from x to the optimal hyperplane, where x can be represented as

$$x = x_p + r \cdot \frac{w_o}{\|w_o\|}$$

where x_p = the normal projection of x on the hyperplane and r = the desired algebraic distance; r is positive if x is on the positive side of the hyperplane and negative if x is on the negative side.

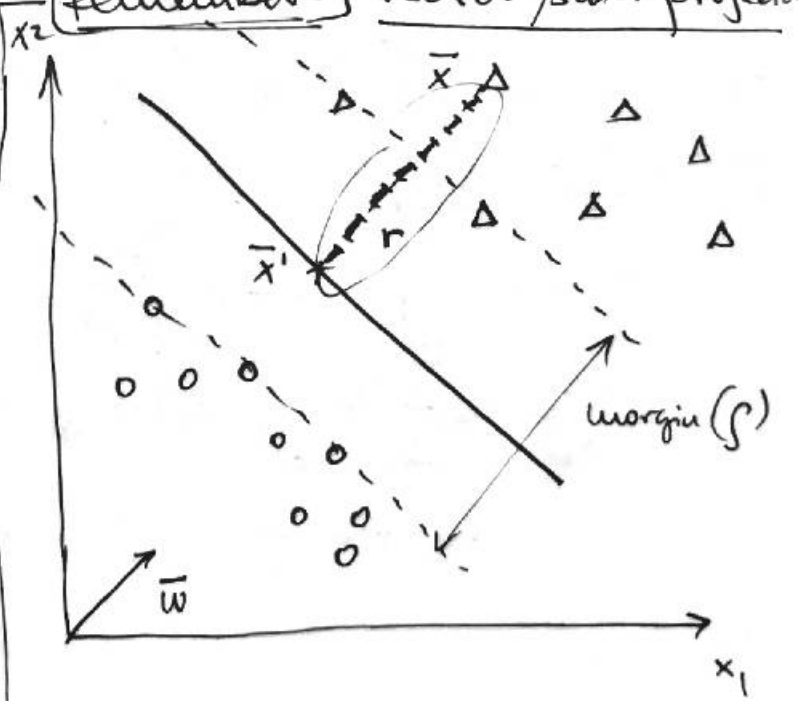
Remember vector/scalar projections

What is the Euclidian distance from $\bar{x}$ to the decision boundary?

The shortest distance from a point to a hyperplane is perpendicular to the plane, so parallel to $\bar{w}$. A unit vector in this direction is $\frac{\bar{w}}{\|\bar{w}\|}$

The marked line (----) is a translation of $r \cdot \frac{\bar{w}}{\|\bar{w}\|}$. Then the projection $\bar{x}'$ is

$$\bar{x} = \bar{x}' + r \cdot \frac{\bar{w}}{\|\bar{w}\|}$$



In our case if we consider $g(x) = w_o^T x + b_o$ and $x = x_p + r \cdot \frac{w_o}{\|w_o\|}$ then

$x_p = x - r \cdot \frac{w_o}{\|w_o\|}$ and because x_p is on the hyperplane $g(x_p) = 0$ so,

$$g(x_p) = w_o^T \left(x - r \frac{w_o}{\|w_o\|} \right) + b_o = 0$$

$$w_o^T \cdot x - w_o^T \cdot r \cdot \frac{w_o}{\|w_o\|} + b_o = 0$$

$$\left(w_o^T \cdot x + b_o \right) - w_o^T \cdot w_o \cdot \frac{r}{\|w_o\|} = 0$$

$$w_o^T \cdot w_o \cdot \frac{r}{\|w_o\|} = w_o^T x + b_o \Rightarrow r = \frac{w_o^T x + b_o}{\|w_o\|}$$

but $\|x\| = \sqrt{x^T \cdot x}$ in Euclidian space dotproduct is inner product

$$r = \frac{g(x)}{\|w_o\|}$$ q.e.d

- The points closest to the optimal hyperplane are called "support vectors" and have an important role in SVM (& they are the hardest to classify).

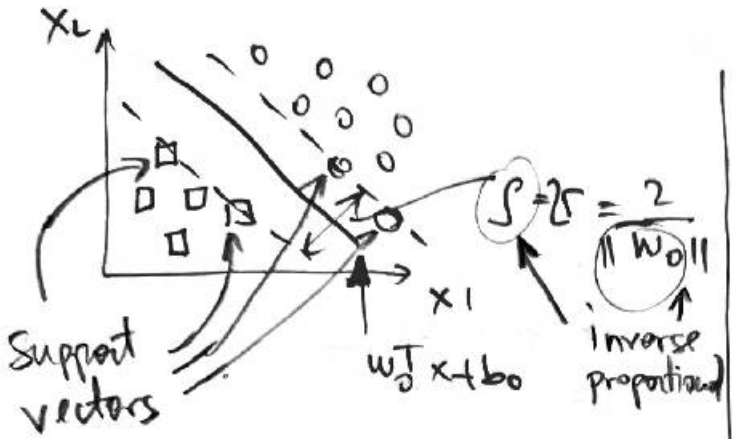
As we shown that

$r = \frac{g(x)}{\|w_o\|}$ the algebraic distance from a support vector $x^{(s)}$ to the optimal hyperplane is

$$r = \frac{g(x^{(s)})}{\|w_o\|} = \begin{cases} \frac{1}{\|w_o\|} & \text{if } d^{(s)} = +1 \\ -\frac{1}{\|w_o\|} & \text{if } d^{(s)} = -1 \end{cases}$$

margin of separation

so $\rho = 2r = \frac{2}{\|w_o\|}$



Maximizing the margin of separation ρ is equivalent to minimizing the Euclidian norm of the weight vector w.

B. Optimal hyperplane for non-linearly separable patterns

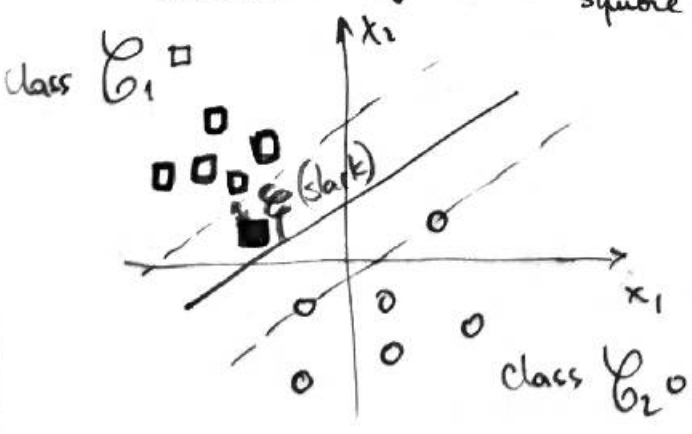
We want to find a hyperplane which minimizes the probability of classification error, averaged over the training samples.

The margin of separation between classes its said to be soft if a data point (x_i, d_i) violates the following condition:

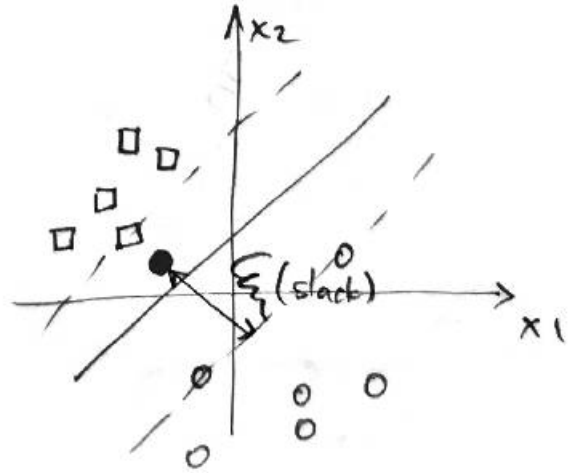
$$d_i \cdot (w^T \cdot x_i + b) \geq +1 \quad , i = 1, 2, \ldots N$$

2 cases

(i) The point falls inside the separation region but on the correct side of the decision surface (bold point)



(ii) The point falls on the wrong side of the decision surface (bold point circle)



In order to measure the deviation of the data points from the condition of separability we introduce the slack variable ξ_i such that:

$$d_i \cdot (w^T x_i + b) \geq 1 - \xi_i \quad , i = 1, 2, \ldots N$$

falling on the correct side $0 < \xi_i \leq 1$

falling on the wrong side $\xi_i > 1$

Learning in SVM

The task is to find the separating hyperplane for which the misclassification error, averaged over the training data, is minimized.

A. For the linear case the learning problem boils down to a constrained optimization problem:

Given the training sample $\{(x_i, d_i)\}_{i=1}^N$, find the optimal weights w and b so that

$$d_i \cdot (w^T x_i + b) \geq 1, \quad i = 1, 2, \ldots N$$

and the weight vector w minimizes the cost function

$$\phi(w) = \frac{1}{2} \cdot w^T w$$

(B.) In the nonlinear case given the training sample $\{(x_i, d_i)\}_{i=1}^{N}$ find the optimum values of the weight vector w and bias b such that they satisfy the constraint:

$$d_i \cdot (w^T \cdot x_i + b) \geq 1 - \xi_i \quad i = 1, 2, \ldots N$$

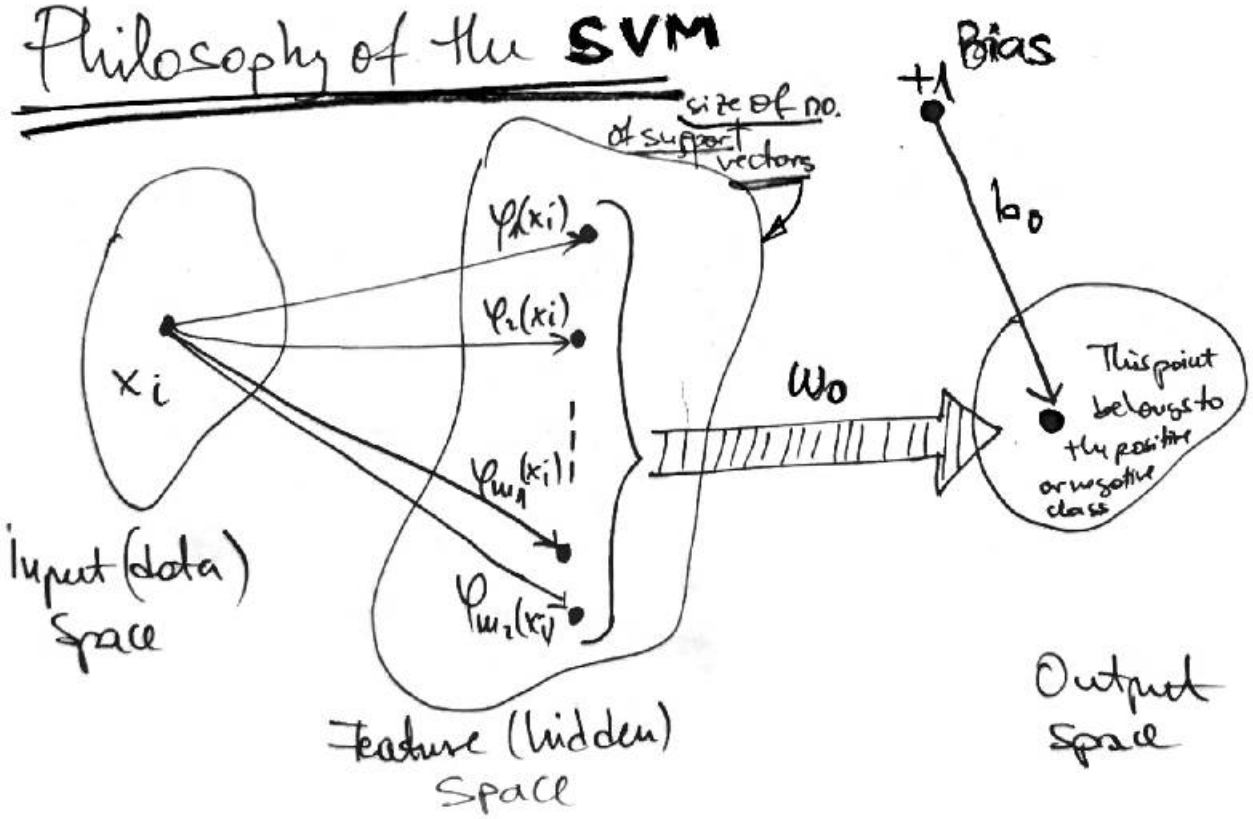
$$\xi_i \geq 0 \text{ for all } i$$

and such that the weight vector w and the slack variables ξ_i minimize the cost function:

$$\phi(w, \xi_i) = \frac{1}{2} \cdot w^T \cdot w + C \cdot \sum_{i=1}^{N} \xi_i$$

C - user defined positive parameter

Philosophy of the SVM



Unsupervised Learning - AIML Course

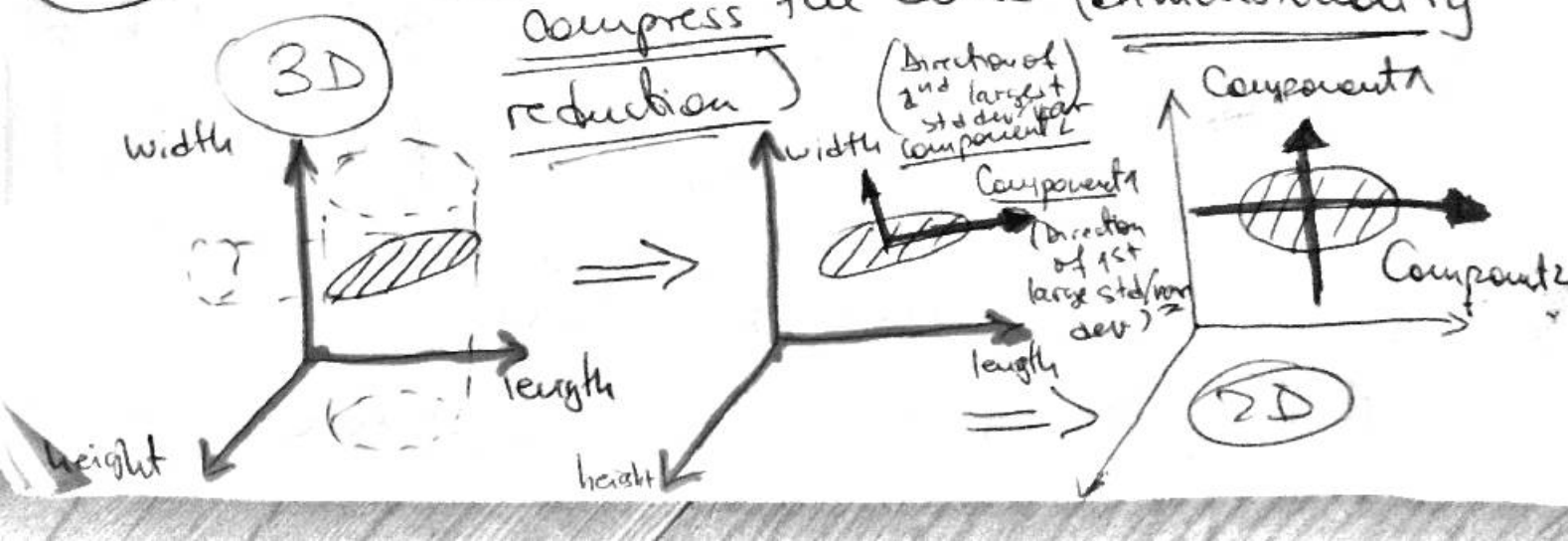
Recall we can have 3 types of learning algorithms:

- Supervised learning (learning with a teacher => we have an error signal).
- unsupervised learning (learning without a teacher => we do not have an error signal). ↙ clustering, learning structure
- reinforcement learning (learn through the interaction with the environment => reward signal to adjust policy).

Unsupervised Learning assumes:

- self-organization to produce output vectors by updating weights;
- does not require labels, discovers classes or clusters in the data;
- learns the statistical regularities of the data;
- developing a representation of the internal structure of data or compress the data (dimensionality reduction)

PCA



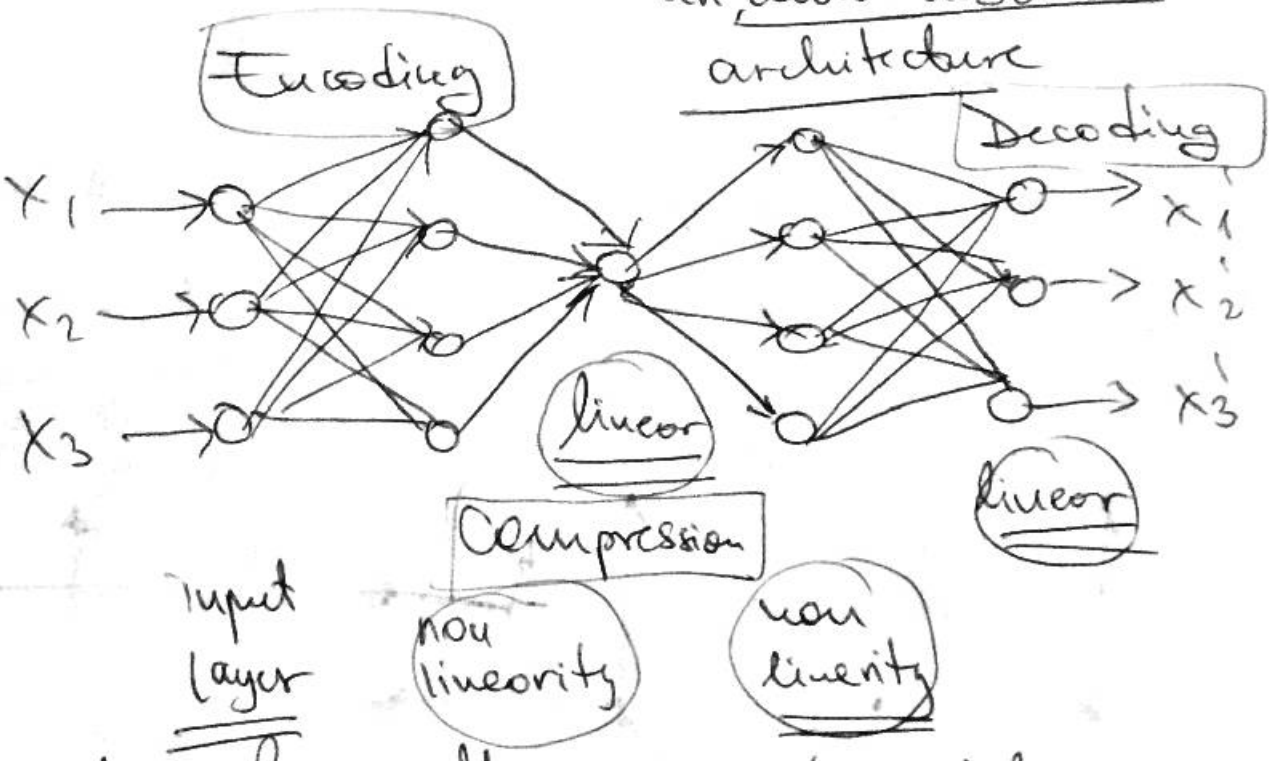
PCA – finds a suitable low-dimensional representation of a high-dimensional space:

- provides a sequence of best linear approximations to a given high-dim observation;
- uses a low-dimensional manifold first 2 components (i.e. largest variance) to represent the most variability in the input space;
- it applies a linear transformation so its application is limited

⇓ a solution

nonlinear PCA

- achieved with a neural network – an MLP with an auto-associative architecture



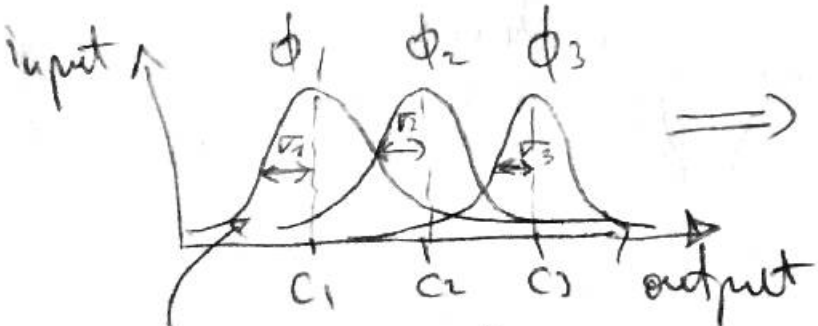
Architecture for: pattern recognition, data compression

Radial Basis Functions (RBF)

- when solving nonlinearly separable pattern-classification there is a benefit to map the input space to a higher dimensional space;
- a nonlinear mapping can help to transform a nonlinear separable classification into a linearly separable one with high probability
- RBFs offer an approach to approximate any arbitrarily function if a sufficient number of radial-basis functions is given with appropiate parameters:

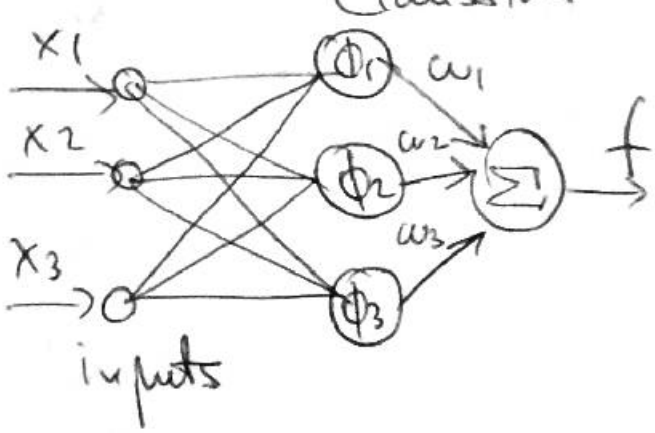
$$f(x) = \sum_{i=1}^{N} w_i \cdot \phi\left(\|x - c_i\|\right) \quad / \frac{x^2}{\beta}$$

where ϕ can take many shapes, e.g. $\phi(x) = e$ Gaussian



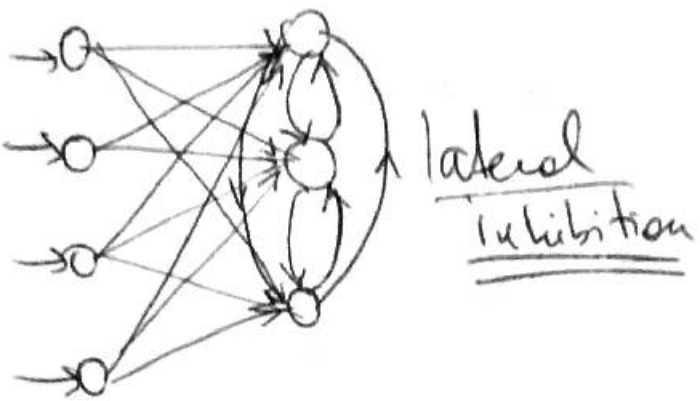
Receptive fields

- area of the input space where the neuron will elicit a response



Vector Quantization (VQ)

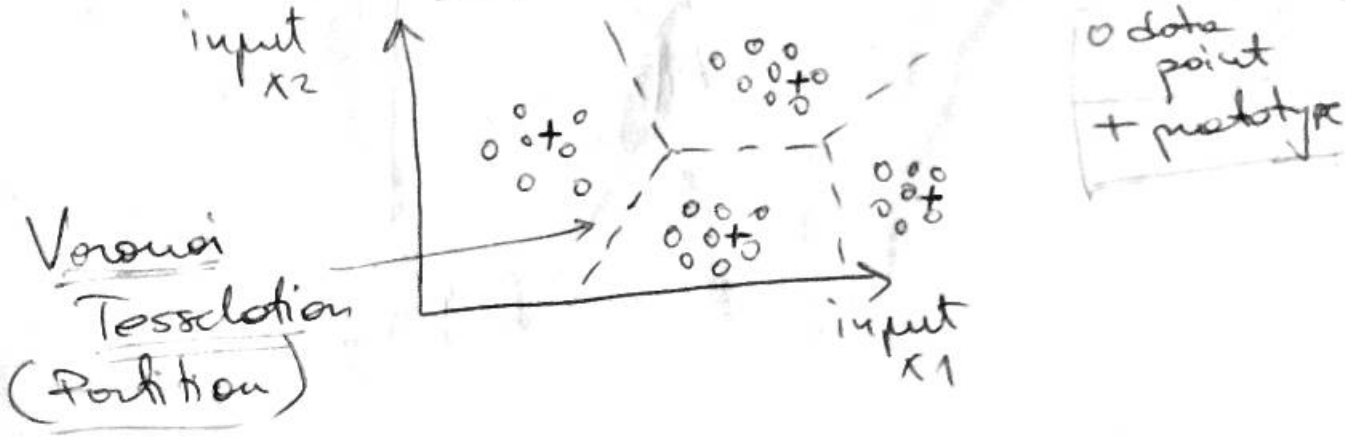
- in neurobiology synapses are strengthen or weakened in a competitive process;
- in the learning process there is competition among neurons which to fire first;
- the neuron with the greatest total input will "win" the competition and will fire, whereas the others will switch off in a process named "Winner-Take-All" (WTA)
- Typical network to implement such a behavior:



- The basic mechanism is to find a winning unit and update its weight to make it more likely to win in the future if similar input is given to the network.

VQ is a form of competitive learning

- finds / discovers structure in the input data
- is a form of lossy compression (some information in the input is lost in the process.)



Algorithm:

#1 Choose the no of clusters, m

#2. Randomly initialize prototypes $w_1, w_2, \ldots w_m$

#3. Repeat until "good enough"

#4. Randomly pick an input x

#5. Find the winning prototype w_k

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

Competition process

#6. Update weights for the winner

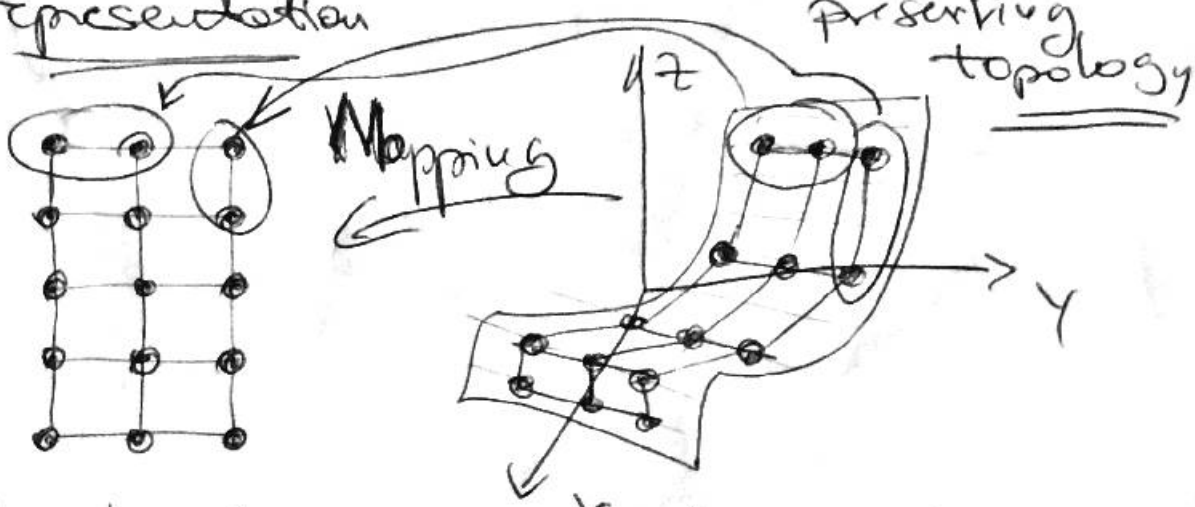
$$w_k(t+1) = w_k(t) + \eta \cdot (x - w_k(t))$$

$\| \|$ or $\|\|_2$ - distance, norm or metric (e.g. Euclidian)

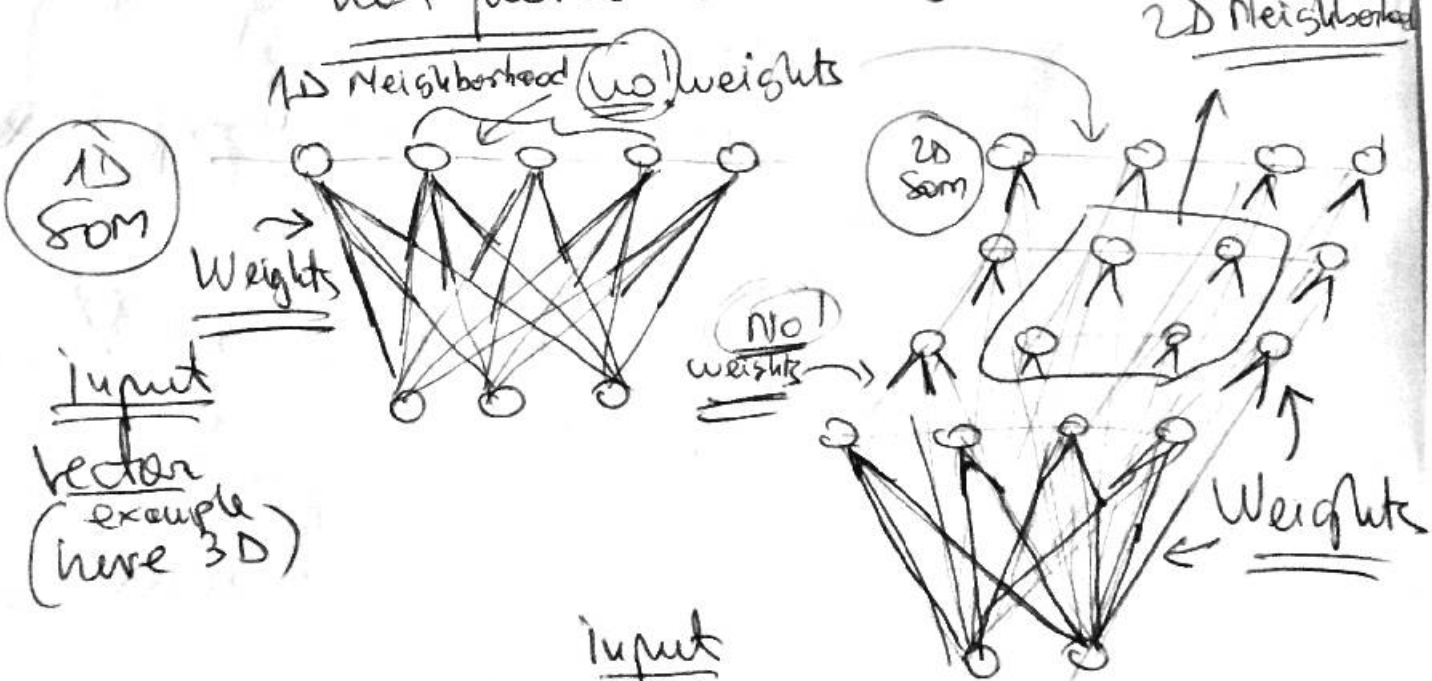
η - learning rate

Self-Organizing Maps (SOM)

- motivated by the retino-cortex mapping
- topologically preserving mapping: close points in the input space will be close also in the representation



- Think about a squashed flower 3D → 2D
- we can have 1D or 2D SOM.
  - 1D - proved to converge mathematically
  - 2D - not proved to converge



- SOM can develop an internal representation of the input and create new classes automatically

Algorithm:

#1. Initialize weights w_{ij}, define neighborhood function $\phi(i,j)$ which changes with Euclidian dist & $T(t)$

#2. Select input x and find the winner (Best Matching unit)

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

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

$$w_j(t+1) = w_j(t) + \eta\, \phi(i,j)(x - w_j(t))$$

η - learning rate

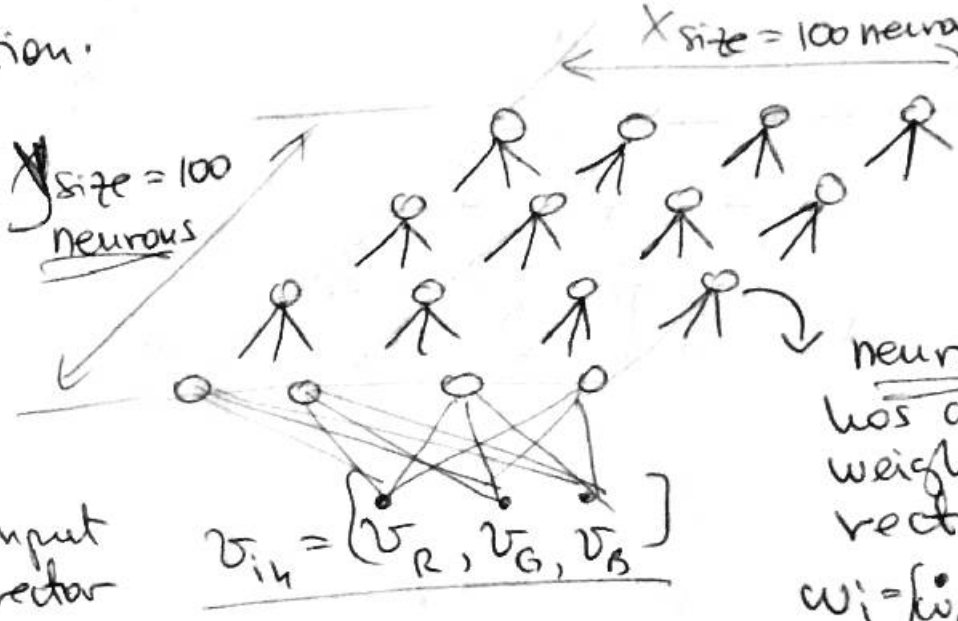
#4. Repeat from #2 until convergence and update η and ϕ

Typical problems:

* Mapping to 2D - faulty convergence "knot" appears in the weights
* Mapping to a too low dimension 3D cube to 2D - losing a dimension

SOM example (demo)

Ex: Mapping colors from 3D colorspace (RGB) to a 2D representation.



Setup: 50 input vectors
1000 training epochs
η initial learning rate 0.5

Training:

#1. Initialize each neuron's weight randomly in the range of the input 0-255 (range of colors)

#2. Pick a random input vector v_{in} from dataset and present it to the network

#3. Find the winner / Best Matching Unit (BMU) by comparing each neuron's weight vector to the input

#4. Compute the radius of the neighborhood function and adjust it.

#5. Update the weights of the neurons in the neighborhood of the BMU closer to the input.

#6. Repeat #2 for N epochs = 1000

Implementation details:

(A.) Computing the winning neuron (BMU)

- use Euclidian distance to find the "closest" neuron to the input

2 Norm $\|v_{in} - w_{BMU}\| = \min_i \{\|v_{in} - w_i\|\}$

where $\|\ \| = D = \sqrt{\sum_{i=0}^{n} (v_{in}[i] - w[i])^2}$

(B.) The neighborhood function shrinks in time

$\sigma(t) = \sigma_0 \cdot e^{\left(-\frac{t}{\lambda}\right)}$ $t = 1, 2, 3 \ldots$

σ_0 - initial value

λ - time constant.

$\sigma_0 = \frac{\max(X_{size}, Y_{size})}{2}$

$\lambda = \frac{N}{\log(\sigma_0)}$

more

less impact modulation less

(C.) Weight update

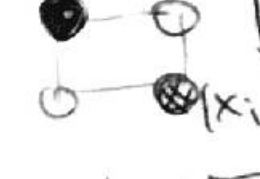
- each neuron in the winner's vicinity gets updated
- the closer a neuron is to the winner, the stronger the update

$w(t+1) = w(t) + \eta(t) \cdot \phi(t) (v_{in}(t) - w(t))$

$\phi(t) = e^{\left(\frac{-dist^2(t)}{2\sigma^2(t)}\right)}$

$\eta(t) = \eta_0 \cdot e^{\left(-\frac{t}{\lambda}\right)}$ $t = 1, 2, 3 \ldots$

$\lambda = \frac{N}{\log(\sigma_0)}$; $\eta_0 = 0.5$

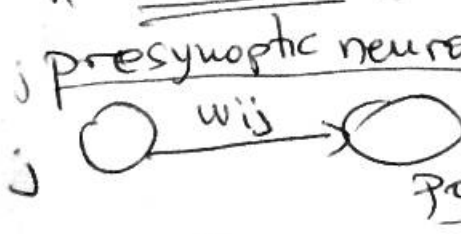


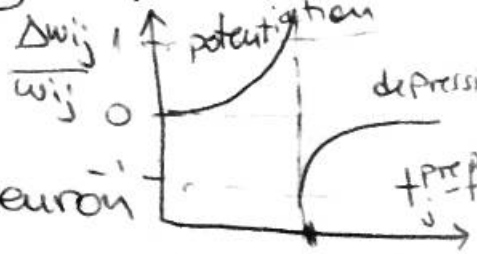
dist - topological distance between a neuron and the winner (BMU)

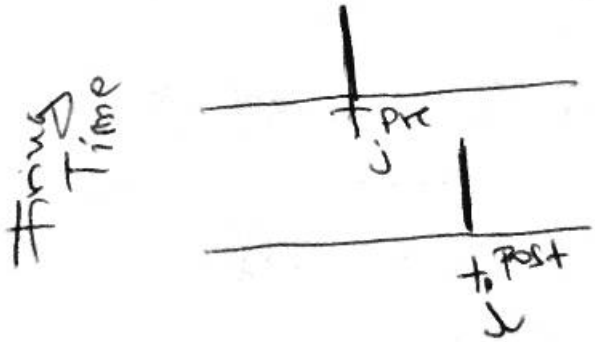
$dist = \sqrt{(x_{BMU} - x_i)^2 + (y_{BMU} - y_i)^2}$

Hopfield Networks (HN)

* based on Hebbian learning principle;
* capable of store and recover associations
* pre-/and post-synaptic neurons where the activation patterns (time) is the core mechanism;
* can be used as associative or content addressable memory.
* uses time in the dynamics using feedback loops (recurrent network);
* recognize learned patterns & can reconstruct corrupted patterns.
* STDP (Spike Timing Dependent Plasticity)





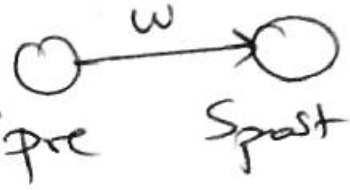


if pre and post synaptic action potentials are correlated then there is an enhancement in synaptic strength.

* Abstracting from Biology → HN neuron

$$w = s_{post} \cdot s_{pre}$$

- w: change in weight
- s_{post}: state post-synaptic neuron
- s_{pre}: state pre-synaptic neuron



Extending to

for binary neurons states $\{0,1\}$

pre \ post	0	1
0	0	0
1	0	1

$\Rightarrow$ Δw

pre \ post	0	1
0	x	x
1	x	↗

not OK!

↗ only increase?!

for bipolar neurons states $\{-1,1\}$

pre \ post	-1	1
-1	1	-1
1	-1	1

$\Rightarrow$

pre \ post	-1	1
-1	↗	↘
1	↘	↗

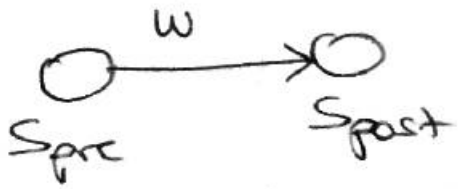
ok ✓

⇓

This representation allows for both correlation & anticorrelation of the states.

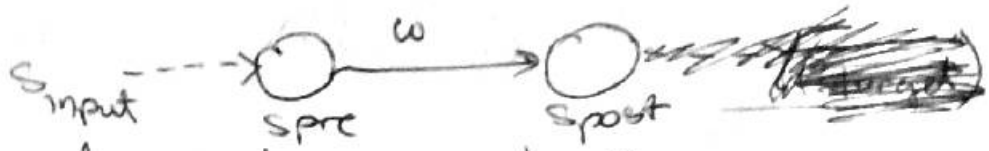
But how can such a system learn patterns?

- consider only 2 neurons



Hebbian learning produces a weight w such that a learned input s_{input} maps onto the corresponding output s_{target}.

Algorithm



#1. Connect input s_{input} to s_{pre}

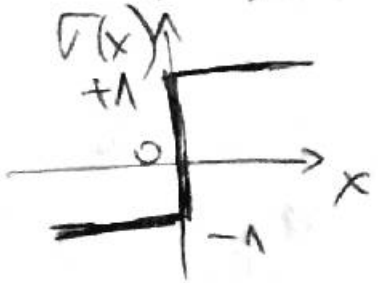
$$s_{pre} = s_{input}$$

#2. Squashing/activation function ∇

$$s_{post} = \nabla(w \cdot s_{pre})$$

Binary threshold / Step function

$$\nabla(x) = \begin{cases} +1, & x \geq 0 \\ -1, & x < 0 \end{cases}$$



#3. Weight calculation with Hebbian Learning

$$w = s_{target} \cdot s_{input}$$

Here we can learn only 1 pattern

because

correlated	$+1 \rightarrow +1$ & $-1 \mapsto -1$
anticorrelated	$+1 \rightarrow -1$ & $-1 \mapsto +1$

#4. substitute #3 in #2 and we get

$$s_{post} = \nabla(s_{target} \cdot s_{input} \cdot s_{pre})$$

#5. But $s_{pre} = s_{input}$ (check #1)

$$s_{post} = \nabla(s_{target} \cdot \underbrace{s_{input} \cdot s_{input}}_{= 1 \text{ because}})$$

$(-1)(-1) = 1$
$(+1)(+1) = 1$

$$s_{post} = \nabla(s_{target})$$

(q.e.d)

Assume we want to learn multiple patterns and correct errors in the input?

* The learning rule can be extended to a network of neurons:

$$W = \bar{s}_{post} \times \bar{s}_{pre} = \begin{pmatrix} & & \\ & & \\ & & \end{pmatrix}$$ matrix

$\bar{a}$ - vector a

$$\bar{s}_{target} = \Gamma\left(W \cdot \bar{s}_{inp}\right)$$

* Such a network can remember a vector as input.

Let's assume we learn a vector and then we flip some bit(s) and feed it to the network.

Let's learn $\bar{s}_{in} = (-1\ -1\ -1\ 1\ -1\ +1) \in \mathbb{R}^6$

$\bar{s}_{target} = (-1\ 1\ -1\ -1\ 1) \in \mathbb{R}^5$

A Hopfield network will converge to a learned pattern given the noisy (corrupted) pattern $\bar{s}_x$

Consider $\bar{s}_x = (-1\ \boxed{1}\ -1\ 1\ -1\ \boxed{-1})$

2 differences from $\bar{s}_{in}$ (learned)

$$\bar{s}_{out} = \Gamma\left(W \cdot \bar{s}_x\right) = \Gamma\left(\begin{pmatrix} 1 & 1 & 1 & -1 & 1 & -1 \\ -1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & 1 & -1 & 1 & -1 \\ 1 & 1 & 1 & -1 & 1 & -1 \\ -1 & -1 & -1 & 1 & -1 & 1 \end{pmatrix} \begin{pmatrix} -1 \\ 1 \\ -1 \\ 1 \\ -1 \\ -1 \end{pmatrix}\right)$$

$W = \bar{s}_{post} \times \bar{s}_{pre} = \bar{s}_{target} \times \bar{s}_{in}$

where $\bar{s}_{post} = \bar{s}_{target}$, $\bar{s}_{pre} = \bar{s}_{in}$

computed as $W = \bar{s}_{target}^T \cdot \bar{s}_{in}$

$$\Rightarrow \nabla(W \cdot \bar{S}_x) = \nabla \begin{pmatrix} -2 \\ +2 \\ -2 \\ -2 \\ +2 \end{pmatrix} = \begin{pmatrix} -1 \\ 1 \\ -1 \\ -1 \\ 1 \end{pmatrix} = \bar{S}_{out} = \bar{S}_{target}$$

If 4 out of 6 bits are correct the network finds the correct pattern!

How to learn more than a pattern?

- simply add weight matrices of all patterns $\Rightarrow$ but patterns will interfere
- the one pattern which is closest to the input is called attractor.
- use orthogonal vectors (dot prod. is zero) to avoid confusion (interference).
- the number of patterns that can be learned is $0.15 \cdot N$ (N - no. of neurons) for reliable performance.

Hopfield network example pseudocode

Pattern recovery task

Steps:

#1. Acquire training patterns and testing patterns (distorted)

train_pat [n_pat] [idx]
test_pat [n_pat] [idx]

n_pat - no of patterns
idx - val of current pixel in patterns

#2. Use bipolar representation {-1, 1}

Training phase:

#3. for pt = 1 : n_pat
#4. for i = 1 : n_neurons
#5. for j = 1 : n_neurons
#6 W [i] [j] += train_pat [pt] [i] . train_pat [pt] [j]

↑ equivalent to
$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}$ where $\Delta w_{ij} = s_i \cdot s_j$
(i = pre, j = post)

$w_{jj} = w_{ii} = 0$

#7. end for j
#8. end for i
#9. end for pt

Recovery phase:

#10. for iter = 1 : max_iters
#11. for ptd = 1 : n_pat
#12. for n1 = 1 : n_neurons
#13. out [n1] = 0
#14. for n2 = 1 : n_neurons
#15. out [n1] += W [n1] [n2] . test_pat [ptd] [n2]

↑ here you can use a sign() function as activation

#16. end for n2
#17. end for n1
#18. end for ptd

#19. if (out == one of the testing patterns) ← convergence condition

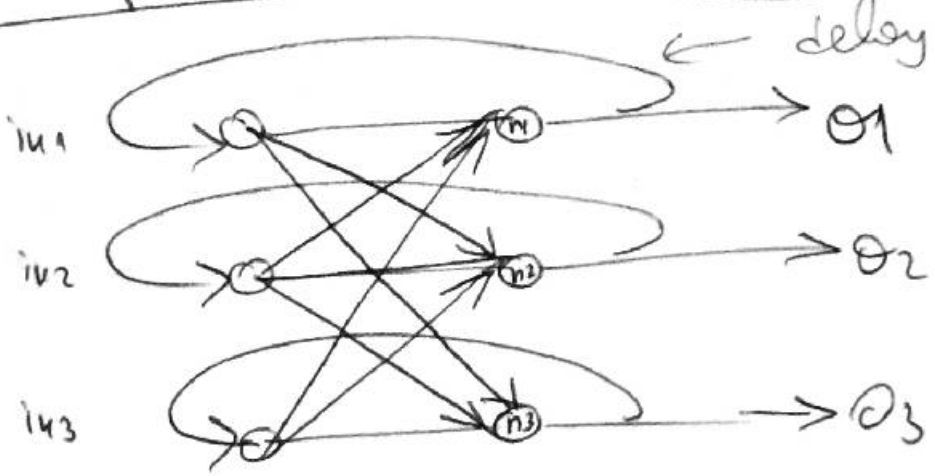
#20. break

#21. else testing_pat = out ; ← feedback the output

#22. end for iter

#23. print recovered pattern

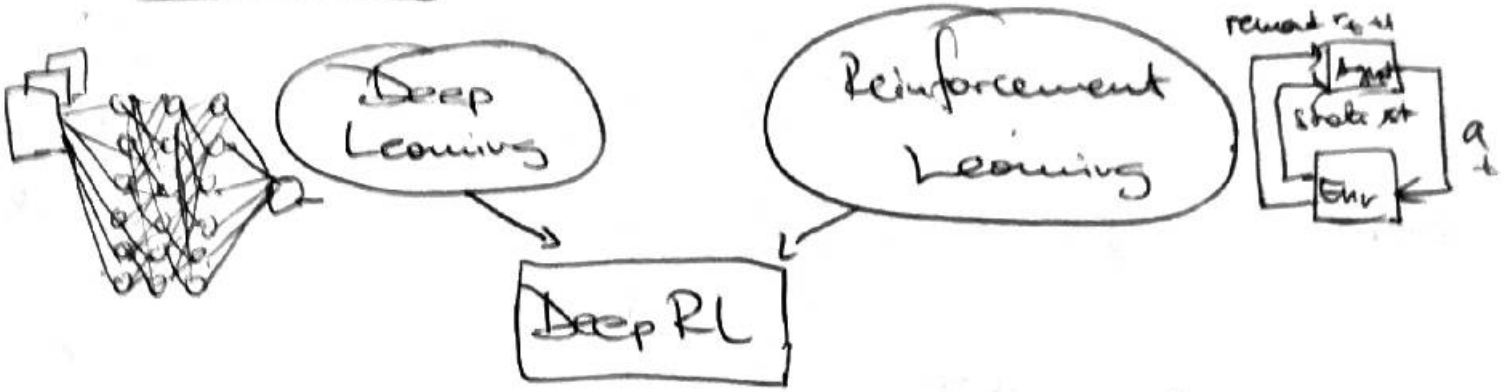
Sample network architecture



AIML Class 1 Deep Reinforcement Learning

Apractical overview

The Basics:

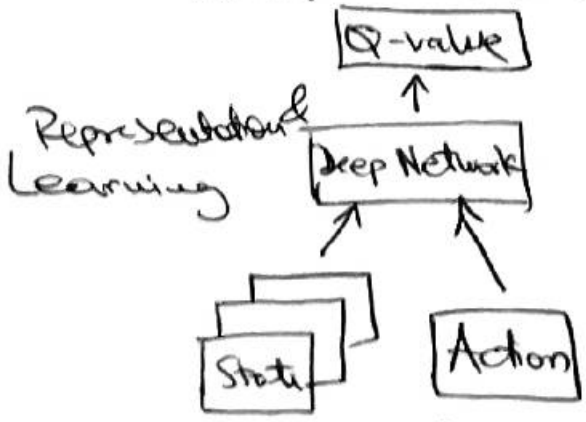


* Use deep nets to represent value function (V(s)) or policy ($\pi(s)$) or the model.
* Use SGD to optimize V(s), $\pi(s)$ or model end-to-end.

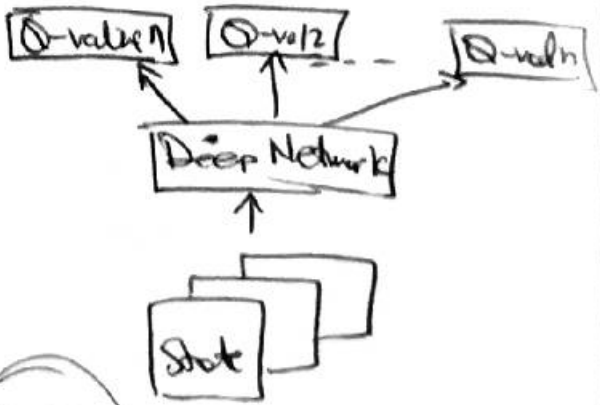
Applications:

- Atari Game player (December 2013)
- "Human Level Control through deep reinf. learning" => Deep Q-Network (DQN)

DeepMind => AlphaGo (2016)

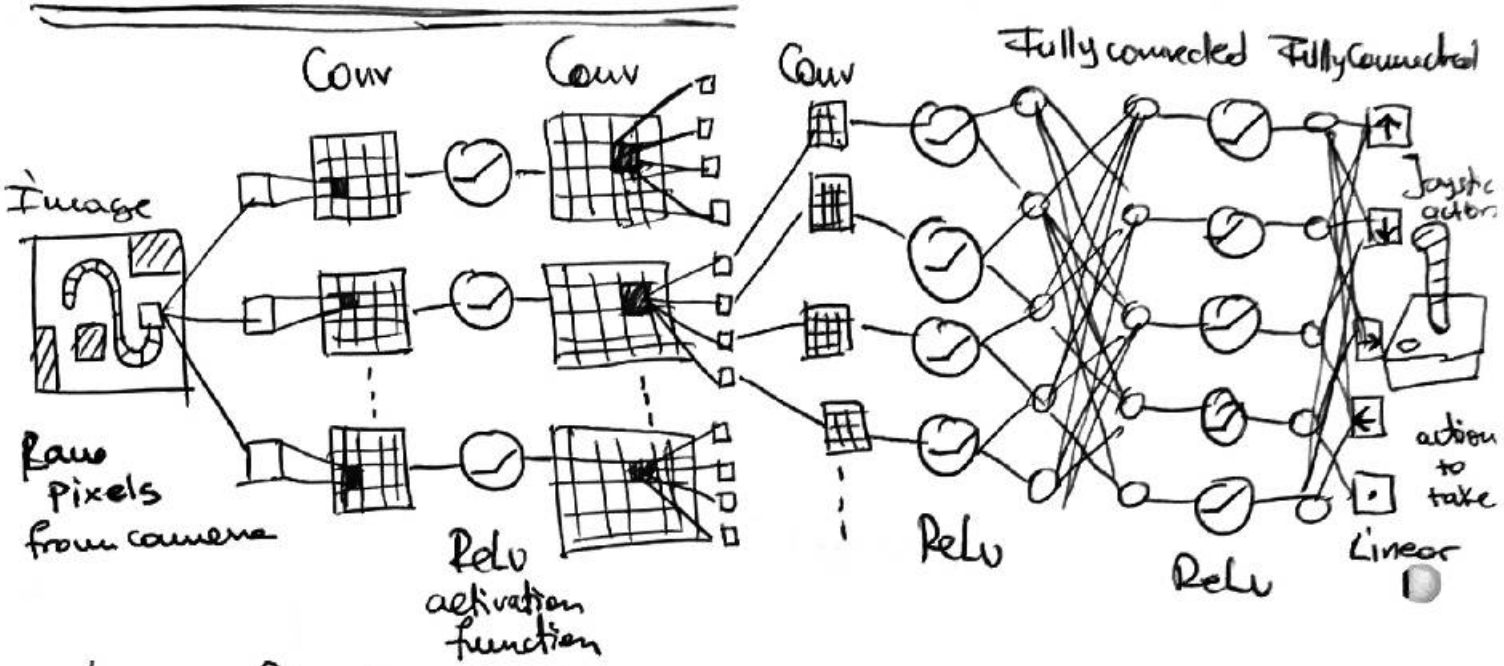


Naive Formulation



DQN in DeepMind

DQN - architecture



Loss function definition

$$L = \frac{1}{2}\left[\underbrace{r(s) + \max_{a'}\left(Q(s', a')\right)}_{\text{target}} - \underbrace{Q(s,a)}_{\text{Prediction}}\right]^2$$

Q table update algorithm :

#1 Do a feedforward pass on current s to predict Q for all a.

#2. Do a feedforward pass for next state s' and calculate $\max_{a'} Q(s', a')$.

#3. Set $Q = r + \gamma \max_{a'} Q(s', a')$.

#4. Update weights using backprop.

Weights update for Value Iteration

- represent value function Q depending on weights w

$$Q(s,a,w) \approx Q^{\pi}(s,a)$$

- define objective function by MSE (Mean Squared Error) in Q-values:

$$L(w) = \mathbb{E}\left[\left(\underbrace{r + \gamma \max_{a'} Q(s',a',w)}_{\text{target}} - Q(s,a,w)\right)^2\right]$$

$\mathbb{E}$: expectation operator (mean value)

Learning rule: ⇩

$$\frac{\partial L(w)}{\partial w} = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q(s',a',w) - Q(s,a,w)\right) \cdot \frac{\partial Q(s,a,w)}{\partial w}\right]$$

Optimize using SGD on $\frac{\partial L(w)}{\partial w}$

Weights update using Policy Iteration

* Same representation of value depending on weights as in value iteration:

$$Q(s,a,w) \approx Q^{\pi}(s,a)$$

- Define objective function by MSE in Q-values:

$$L(w) = \mathbb{E}\left[\left(\underbrace{r+\gamma \cdot Q(s',a',w)}_{\text{target}} - Q(s,a,w)\right)^2\right]$$

$\mathbb{E}$: expectation operator (mean value)

Gradient ⇓ update rule using SGD on

$$\frac{\partial L(w)}{\partial w} = \mathbb{E}\left(r+\gamma Q(s',a',w) - Q(s,a,w)\right)\frac{\partial Q(s,a,w)}{\partial w}$$

Stability in Deep RL

- Oscillations & divergence in naïve networks for Deep RL:
  - Policy changes with slight changes in Q-values - given the data distribution.
  - rewards and Q-values scales are not known ⇒ large unstable backpropagation updates.

DQN handles stability:

(A) use experience replay:
- break correlations in data.
- learn from past policies.

(B) freeze target Q-network:
- avoid oscillations.
- break connection between Q-network and target.

(C) provide stable gradients:
- clip rewards, typically $[-1, +1]$
- normalize network adaptively.

(A) - Build a dataset from agent's own experience
- take action a_t according to greedy policy ε.
- store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory D.
- sample mini batch of transitions (s, a, r, s') from D.

- Optimize MSE between Q-network and Q-learning targets:

$$L(w) = \mathbb{E}_{s,a,r,s' \sim D}\left[\left(r + r \max_{a'} Q(s', a', w) - Q(s, a, w)\right)^2\right]$$

(B.) - Fix parameters used in Q-learning target

- compute Q-learning targets w.r.t old fixed parameter w^-

$$r + \gamma \max_{a'} Q(s', a', w^-)$$

- optimize MSE between Q-network and Q-learning targets:

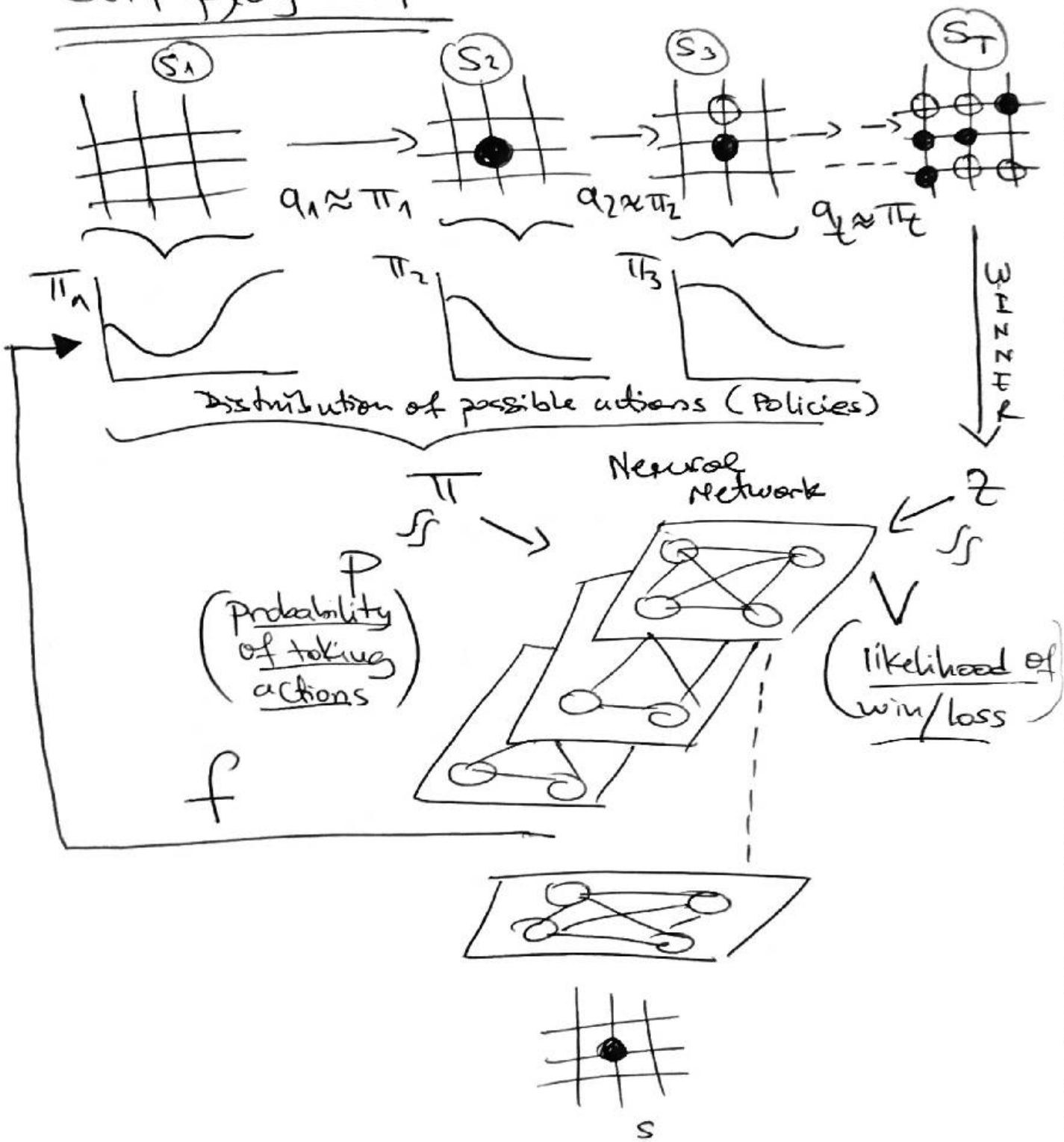
$$L(w) = \mathbb{E}_{s,a,r,s' \sim D}\left[\left(r + \gamma \max_{a'} Q(s', a', w^-) - Q(s, a, w)\right)^2\right]$$

- periodically update $w^- \leftarrow w$

Deep RL from "blank slate" ("tabula rasa")

- Alpha Go Zero!
- Improve planning and evaluation with feedback from self-play (No, Zero human game data!)

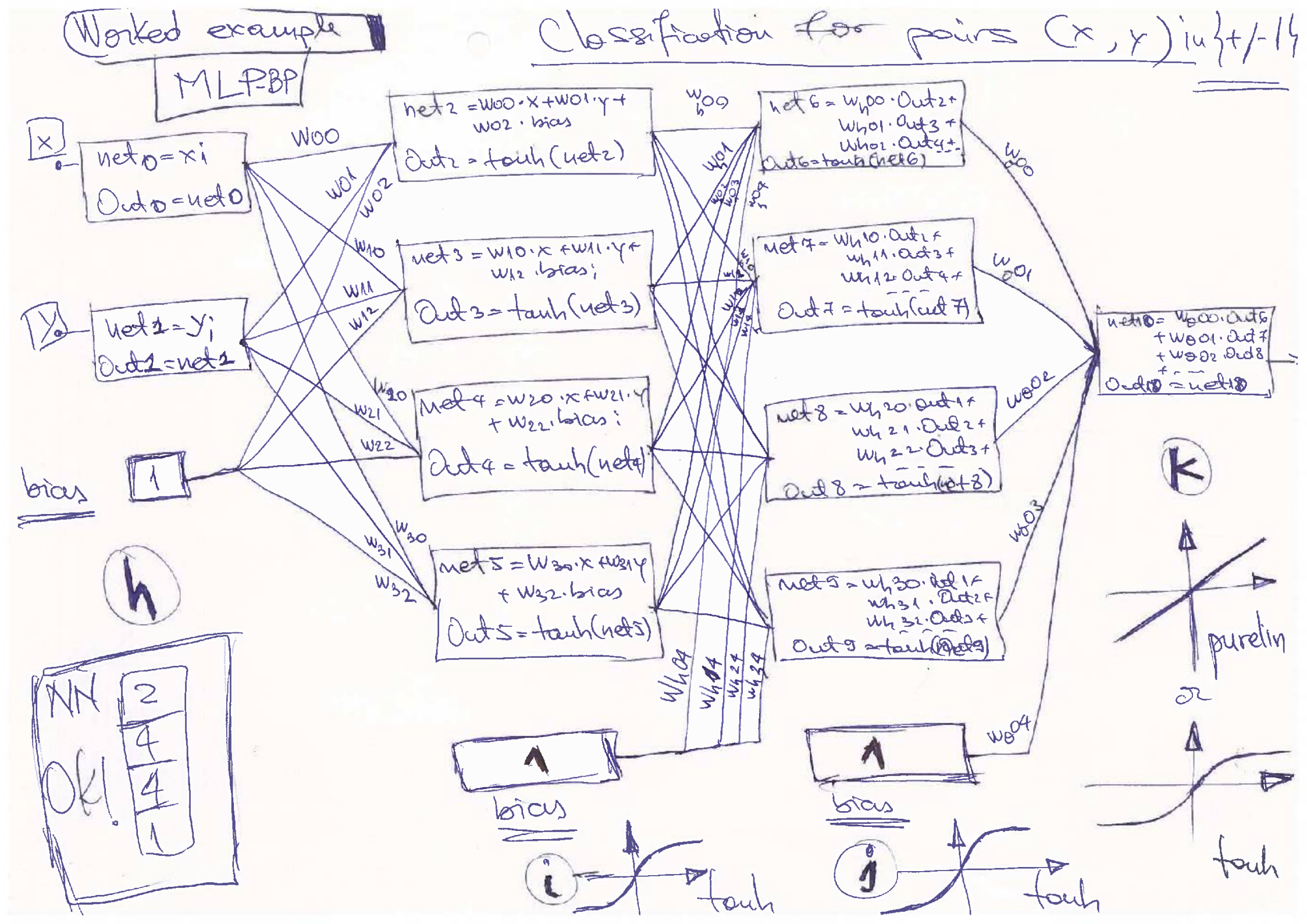
Self play Loop



Worked example

MLP-BP

Classification for pairs (x, y) in {+/-1}



Backpropogation

$\delta_{10} = \delta_{out} = 2 \cdot (target - out)$ (0)

$\delta_9 = \delta_{10} \cdot w_{o03} \cdot (1 - Out_9^2)$

$\delta_8 = \delta_{10} \cdot w_{o02} \cdot (1 - Out_8^2)$ (1)

$\delta_7 = \delta_{10} \cdot w_{o01} \cdot (1 - Out_7^2)$

$\delta_6 = \delta_{10} \cdot w_{o00} \cdot (1 - Out_6^2)$

$\delta_5 = w_{h33} \cdot \delta_9 \cdot (1 - Out_5^2) + w_{h23} \cdot \delta_8 (1 - Out_5^2) + w_{h13} \cdot \delta_7 (1 - Out_5^2) + w_{h03} \delta_6 (1 - Out_5^2)$

$\delta_4 = w_{h32} \cdot \delta_9 (1 - Out_4^2) + w_{h22} \cdot \delta_8 (1 - Out_4^2) + w_{h12} \cdot \delta_7 (1 - Out_4^2) + w_{h02} \delta_6 (1 - Out_4^2)$

$\delta_3 = w_{h31} \delta_9 (1 - Out_3^2) + w_{h21} \cdot \delta_8 \cdot (1 - Out_3^2) + w_{h11} \delta_7 (1 - Out_3^2) + w_{h01} \delta_6 (1 - Out_3^2)$

$\delta_2 = w_{h30} \delta_9 (1 - Out_2^2) + w_{h20} \cdot \delta_8 \cdot (1 - Out_2^2) + w_{h10} \cdot \delta_7 \cdot (1 - Out_2^2) + w_{h00} \delta_6 \cdot (1 - Out_2^2)$ (II)

$\delta_1 = w_{31} \cdot \delta_5 + w_{21} \cdot \delta_4 + w_{11} \cdot \delta_3 + w_{01} \cdot \delta_2$ (III)

$\delta_0 = w_{30} \cdot \delta_5 + w_{20} \cdot \delta_4 + w_{10} \cdot \delta_3 + w_{00} \cdot \delta_2$

Update weights

Output:

$\Delta w_{o04} = \eta \cdot \delta_{10} \cdot 1;$

$\Delta w_{o03} = \eta \cdot \delta_{10} \cdot out_9;$

$\Delta w_{o02} = \eta \cdot \delta_{10} \cdot out_8;$

$\Delta w_{o01} = \eta \cdot \delta_{10} \cdot out_7;$

$\Delta w_{o00} = \eta \cdot \delta_{10} \cdot out_6;$

Hidden:

$\Delta w_{h33} = \eta \cdot \delta_9 \cdot Out_5$

$\Delta w_{h23} = \eta \cdot \delta_8 \cdot Out_5$

$\Delta w_{h13} = \eta \cdot \delta_7 \cdot Out_5$

$\Delta w_{h03} = \eta \cdot \delta_6 \cdot Out_5$

$\Delta w_{h32} = \eta \cdot \delta_9 \cdot Out_4$

$\Delta w_{h22} = \eta \cdot \delta_8 \cdot Out_4$

$\Delta w_{h12} = \eta \cdot \delta_7 \cdot Out_4$

$\Delta w_{h02} = \eta \cdot \delta_6 \cdot Out_4$

$\Delta w_{h31} = \eta \cdot \delta_9 \cdot Out_3$

$\Delta w_{h21} = \eta \cdot \delta_8 \cdot Out_3$

$\Delta w_{h11} = \eta \cdot \delta_7 \cdot Out_3$

$\Delta w_{h01} = \eta \cdot \delta_6 \cdot Out_3$

$\Delta w_{h30} = \eta \cdot \delta_9 \cdot Out_2$

$\Delta w_{h20} = \eta \cdot \delta_8 \cdot Out_2$

$\Delta w_{h10} = \eta \cdot \delta_7 \cdot Out_2$

$\Delta w_{h00} = \eta \cdot \delta_6 \cdot Out_2$

+ bias

Input:

$\Delta w_{31} = \eta \cdot \delta_5 \cdot y$

$\Delta w_{21} = \eta \cdot \delta_4 \cdot y$

$\Delta w_{11} = \eta \cdot \delta_3 \cdot y$

$\Delta w_{01} = \eta \cdot \delta_2 \cdot y$

$\Delta w_{30} = \eta \cdot \delta_5 \cdot x$

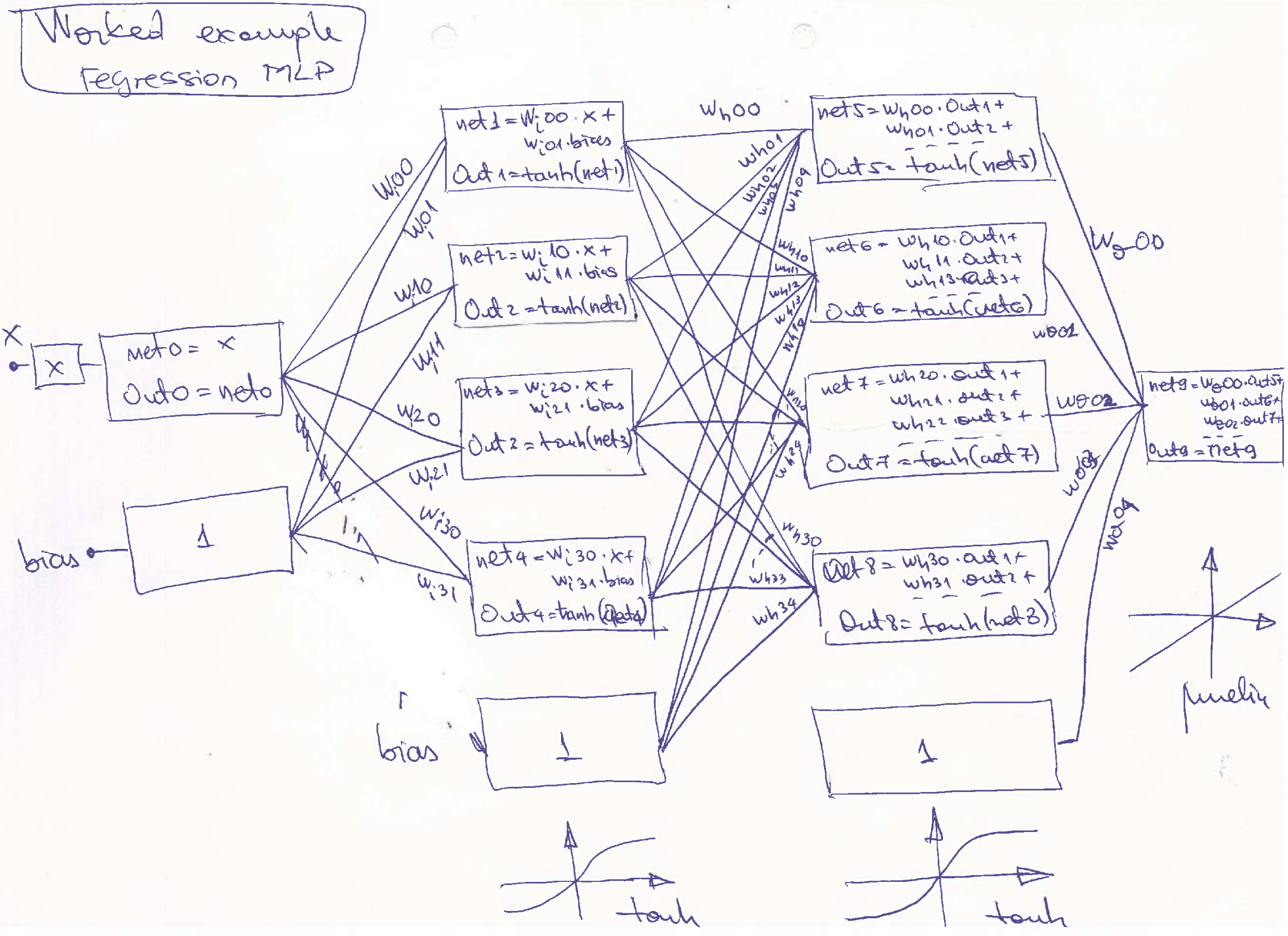
$\Delta w_{20} = \eta \cdot \delta_4 \cdot x$

$\Delta w_{10} = \eta \cdot \delta_3 \cdot x$

$\Delta w_{00} = \eta \cdot \delta_2 \cdot x$

$w_{ij} += \Delta w_{ij}$ — update weights

Worked example
Regression MLP



Backpropagation

$$\delta_9 = \delta_{out} = 2 \cdot (target - out)$$
$$\delta_8 = \delta_9 \cdot w_{o03} \cdot (1 - Out_8^2)$$
$$\delta_7 = \delta_9 \cdot w_{o02} \cdot (1 - Out_7^2)$$
$$\delta_6 = \delta_9 \cdot w_{o01} \cdot (1 - Out_6^2)$$
$$\delta_5 = \delta_9 \cdot w_{o00} \cdot (1 - Out_5^2) \quad (1)$$

$$\delta_4 = w_{h03} \cdot \delta_5 \cdot (1 - Out_4^2) + w_{h13} \cdot \delta_6 \cdot (1 - Out_4^2) + w_{h23} \cdot \delta_7 \cdot (1 - Out_4^2) + w_{h33} \cdot \delta_8 \cdot (1 - Out_4^2)$$
$$\delta_3 = w_{h02} \cdot \delta_5 \cdot (1 - Out_3^2) + w_{h12} \cdot \delta_6 \cdot (1 - Out_3^2) + w_{h22} \cdot \delta_7 \cdot (1 - Out_3^2) + w_{h32} \cdot \delta_8 (1 - Out_3^2)$$
$$\delta_2 = w_{h01} \cdot \delta_5 \cdot (1 - Out_2^2) + w_{h11} \cdot \delta_6 \cdot (1 - Out_2^2) + w_{h21} \cdot \delta_7 \cdot (1 - Out_2^2) + w_{h31} \cdot \delta_8 (1 - Out_2^2)$$
$$\delta_1 = w_{h00} \cdot \delta_5 \cdot (1 - Out_1^2) + w_{h10} \cdot \delta_6 \cdot (1 - Out_1^2) + w_{h20} \cdot \delta_7 \cdot (1 - Out_1^2) + w_{h30} \cdot \delta_8 (1 - Out_1^2) \quad (11)$$

$$\delta_0 = w_{i00} \cdot \delta_1 + w_{i10} \cdot \delta_2 + w_{i20} \cdot \delta_3 + w_{i30} \cdot \delta_4$$

Weight update

Output:

$$\Delta w_{o04} = \eta \cdot \delta_9 \cdot bias;$$
$$\Delta w_{o03} = \eta \cdot \delta_9 \cdot Out_8;$$
$$\Delta w_{o02} = \eta \cdot \delta_9 \cdot Out_7;$$
$$\Delta w_{o01} = \eta \cdot \delta_9 \cdot Out_6;$$
$$\Delta w_{o00} = \eta \cdot \delta_9 \cdot Out_5;$$

Hidden:

$\Delta w_{h34} = \eta \cdot (\delta_8 + \cancel{\delta_6 + \delta_7 + \delta_8}) \cdot 1$

$\Delta w_{h24} = \eta \cdot (\delta_7) \cdot bias$

$\Delta w_{h14} = \eta \cdot (\delta_6) \cdot bias$

$\Delta w_{h04} = \eta \cdot (\delta_5) \cdot bias$

$\Delta w_{h33} = \eta \cdot \delta_8 \cdot out_4$

$\Delta w_{h23} = \eta \cdot \delta_7 \cdot out_4$

$\Delta w_{h13} = \eta \cdot \delta_6 \cdot out_4$

$\Delta w_{h03} = \eta \cdot \delta_5 \cdot out_4$

$\Delta w_{h32} = \eta \cdot \delta_8 \cdot out_3$

$\Delta w_{h22} = \eta \cdot \delta_7 \cdot out_3$

$\Delta w_{h12} = \eta \cdot \delta_6 \cdot out_3$

$\Delta w_{h02} = \eta \cdot \delta_5 \cdot out_3$

$\Delta w_{h31} = \eta \cdot \delta_8 \cdot out_2$

$\Delta w_{h21} = \eta \cdot \delta_7 \cdot out_2$

$\Delta w_{h11} = \eta \cdot \delta_6 \cdot out_2$

$\Delta w_{h01} = \eta \cdot \delta_5 \cdot out_2$

$\Delta w_{h30} = \eta \cdot \delta_8 \cdot out_1$

$\Delta w_{h20} = \eta \cdot \delta_7 \cdot out_1$

$\Delta w_{h10} = \eta \cdot \delta_6 \cdot out_1$

$\Delta w_{h00} = \eta \cdot \delta_5 \cdot out_1$

Input:

$\Delta w_{i30} = \eta \cdot \delta_4 \cdot x$

$\Delta w_{i20} = \eta \cdot \delta_3 \cdot x$

$\Delta w_{i20} = \eta \cdot \delta_2 \cdot x$

$\Delta w_{i00} = \eta \cdot \delta_1 \cdot x$

Update

$w_{ij} += \Delta w_{ij}$

ARMA models for Time Series prediction or Time Series Forecasting using Stochastic Models

A time series is a series of data points indexed in time order.

||→ Time series analysis: methods & tools to analyze data in order to extract statistics & other characteristics of data;

||→ Time series forecasting: using a model to predict future values based on previously observed values.

- We previously learnt about MLPs which were able to learn functions out of (input, output) pairs using supervised learning. The functions could be arbitrarily non-linear.
- The simpler approach is to use linear models, which for forecasting offer good results at low computational cost, simplicity and with ease of understanding.

We will now look at linear and Non-linear Stochastic time series models.

- Autoregressive Moving Average (ARMA) Models
  - is a combination of AR(p) and MA(q) models
  - in an AR(p) model the future value of a variable is assumed to be a linear combination of p past observations and a random error:

$$y_t = c + \sum_{i=1}^{p} \varphi_i \cdot y_{t-i} + \varepsilon_t$$

p - order of the model

$$y_t = c + \varphi_1 \cdot y_{t-1} + \varphi_2 \cdot y_{t-2} + \dots + \varphi_p \cdot y_{t-p} + \varepsilon_t$$

where y_t - actual value, ε_t - random error at time t

φ_i - model parameters

A MA(q) model uses past q errors as variables to predict y_t:

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

$$y_t = \mu + \theta_1 e_{t-1} + \theta_2 e_{t-2} + \dots + \theta_q e_{t-q} + \varepsilon_t$$

Where μ - mean of the series, θ_j - model parameters

q - order of the model

$\varepsilon_t \sim \mathcal{N}(0, \sigma^2)$ random, normally distributed noise

The ARMA (p, q) model can be mathematically described as:

$$y_t = c + \varepsilon_t + \sum_{i=1}^{p} \varphi_i y_{t-i} + \sum_{j=1}^{q} \theta_j e_{t-j}$$

Usually the models φ_i and θ_j are polynomials

$$\varphi(L) = 1 - \sum_{i=1}^{p} \varphi_i L^i \quad \text{and} \quad \theta(L) = 1 + \sum_{j=1}^{q} \theta_j \cdot L^j$$

$L y_t = y_{t-1}$ where L is the lag operator.

- **Nonlinear models : NAR (Nonlinear Auto Regressive) and NARMA (Nonlinear Autoregressive Moving Average)**

- A simple extension to nonlinear systems can be done as a **NAR model**, generically defined as:

$$y_t = h(y_{t-1}, y_{t-2}, y_{t-3} \cdots y_{t-p}) + \varepsilon_t$$

where $h(\cdot)$ is a smooth unknown function with the assumption that the best (i.e. min mean square error) prediction of y_t given previous $y_{t-1}, y_{t-2}, \cdots y_{t-p}$ is its conditional mean:

$$\hat{y}_t = \mathbb{E}\left(y_t \mid y_{t-1}, \cdots y_{t-p}\right) = h\left(y_{t-1}, \cdots, y_{t-p}\right)$$

($\mathbb{E}$: expectation operator)

- The feedforward network able to implement a NAR model is a nonlinear approximation of h such as:

$$\hat{y}_t = h\left(y_{t-1}, \cdots y_{t-p}\right) = \sum_{i=1}^{I} W_i \cdot f_i\left(\sum_{j=1}^{P} w_{ij} y_{t-j}\right)$$

(f_i: activation function)

- NARMA generalize ARMA, with h unknown

$$y_t = h\left(y_{t-1}, y_{t-2} \cdots y_{t-p}, \varepsilon_{t-1}, \cdots \varepsilon_{t-q}\right)$$

- then we have a recurrent neural network to approximate it

$$\hat{y}_t = h\left(y_{t-1}, \cdots, y_{t-p}\right) = \sum_{i=1}^{I} W_i \cdot f_i\left(\underbrace{\sum_{j=1}^{P} w_{ij} y_{t-j}}_{AR} + \underbrace{\sum_{j=1}^{I} w'_{ij} \overbrace{\left(y_{t-j} - \hat{y}_{t-j}\right)}^{\varepsilon_{t-j}}}_{MA}\right)$$

(f_i: activation function)

• Autoregressive Integrated Moving Average (ARIMA)

- ARMA models are limited to stationary time series.
- in applications such as socio-economic systems and business the phenomena are non-stationary.
- ARIMA is a generalization of ARMA.

ARIMA (p, d, q) is described mathematically as:

$$\varphi(L)\cdot(1-L)^{d}\cdot y_t = \Theta(L)\varepsilon_t$$

(d → finite differencing level)

p – order of autoregression
q – order of moving average
d – order of the integrated model

d – controls differencing → $d = 1$ typical
$d = 0 \Rightarrow$ ARMA model

$$y_t = \frac{\Theta(L)\cdot\varepsilon_t}{\varphi(L)\cdot(1-L)^{d}}$$

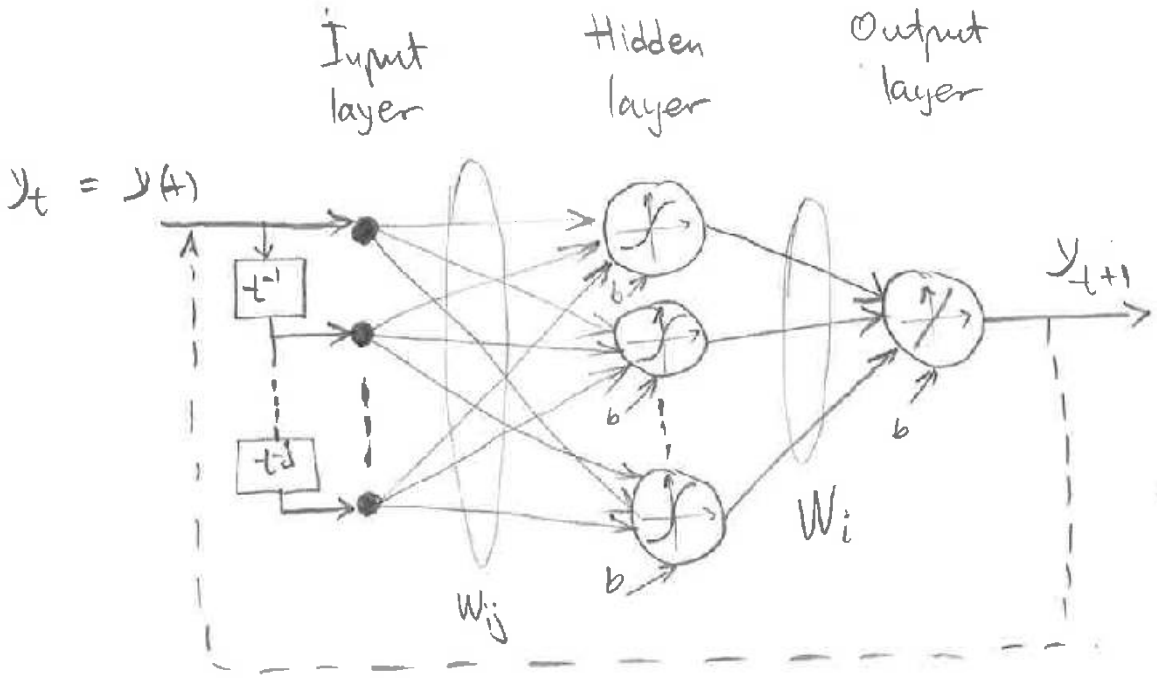
where

$$\varphi(L) = 1 - \sum_{i=1}^{p} \varphi_i L^i$$

$$\Theta(L) = 1 + \sum_{j=1}^{q} \Theta_j L^j$$

where L is the lag operator (delay) which has the property of $y_t = L\,y_{t+1}$

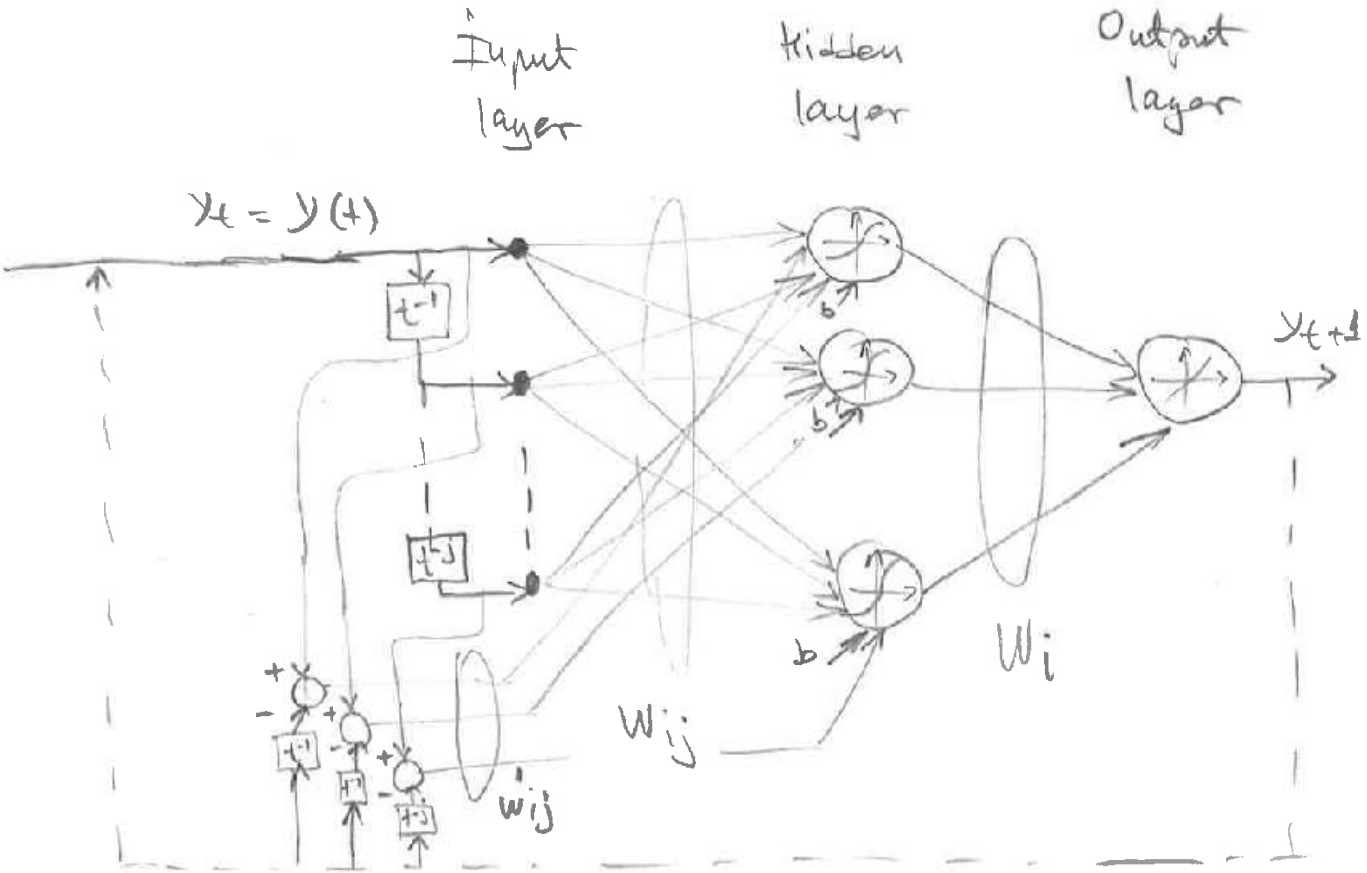
NAR network architecture example



Feedback is open for training & closed for prediction.

$$\hat{y}_t = h\left(y_{t-1} \cdots y_{t-p}\right) = \sum_{i=1}^{I} W_i \cdot f\left(\sum_{j=1}^{p} w_{ij} y_{t-j}\right)$$

NARMA network architecture



Liquid State Machines (LSMs)

- model(s) computations in cortical microcircuits of the brain; Wolfgang Maass developed it.
- model of real-time computations on continuous streams of data; (Reservoir Computing)
- uses a high-dimensional dynamical system and have the inputs continuously perturbing it!

  → project inputs to a higher dimensional space.

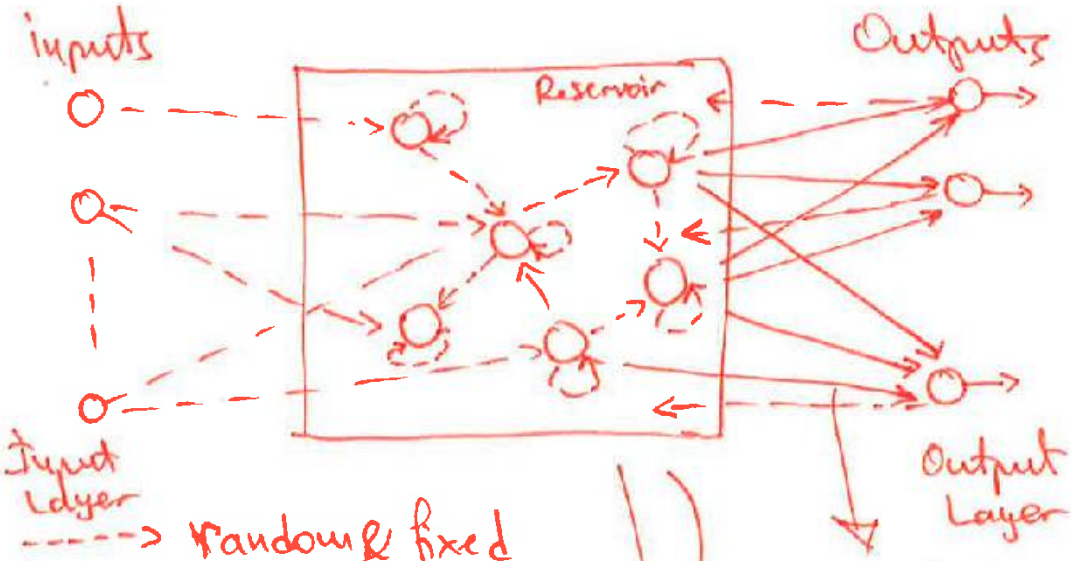
  → internal dynamics of a recurrent spiking neural network.

  → the liquid itself doesn't generate any output it is just a "reservoir" for the inputs.

  → there is a readout function to compute the output of the LSM.

  → given a time series input the LSM can produce a time series, and to obtain the desired values one needs to adjust the weights on the connections between reservoir and output.

The generic model is depicted in the following figure:



----> random & fixed connections

——> linear & trainable connections

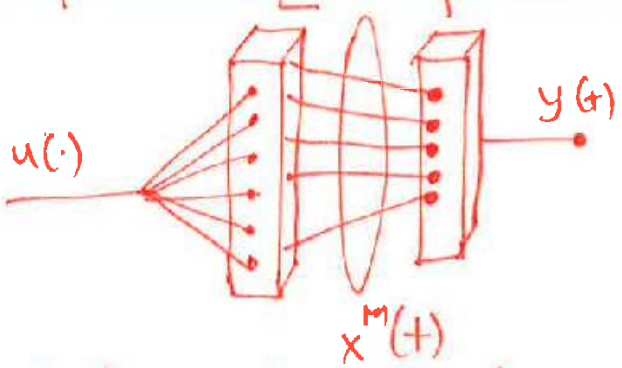
Output can be trained for various tasks (sum, count, f)

(Arbitrary figure)

this mixed feedforward/recurrent pool of neurons can compute a large variety of nonlinear functions

Model uses LIF neurons (Leaky Integrate & Fire Neurons) - biologically plausible

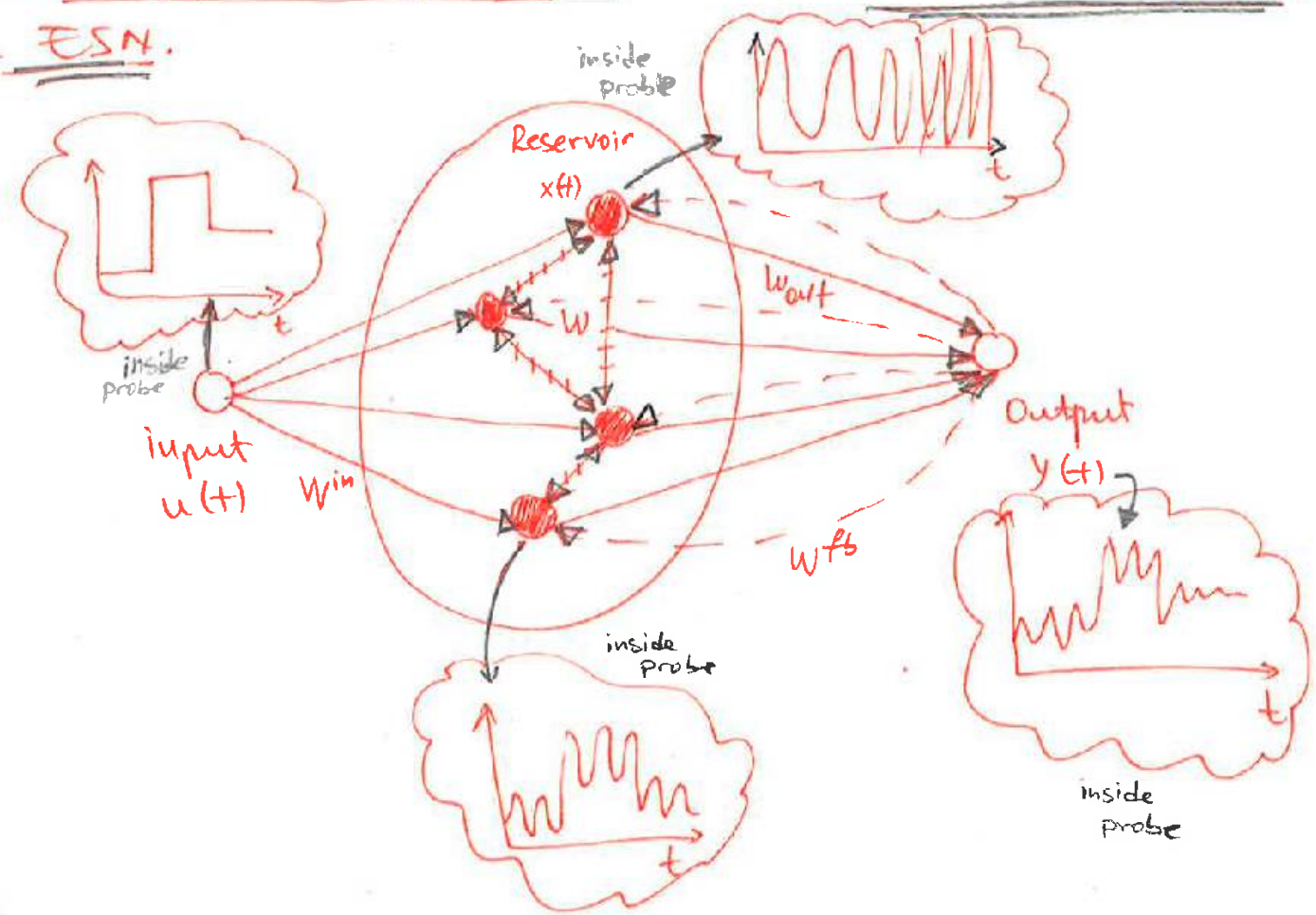
Formal definition



- A filter maps between two functions of time $u(\cdot) \to y(\cdot)$
- A LSM, $M = \langle L^M, f^M \rangle$, L^M = filter, f^M = readout
- State at time t, $x^M(t) = (L^M \cdot u)(t)$
- Output at time t, $y(t) = (Mu)(t) = f^M(x^M(t))$

Train with backprop for example

In practical applications LIF (Leaky Integrate and Fire neuron) dynamics are too complex and not necessary. Another Reservoir Computing technique showing the basic idea of LSM is the Echo State Network or ESN.



ESN implementation

- $u(t)$ – input signal
- W^{in} – inputs weight vector (random, fixed)
- W – reservoir recurrent weight vector (random, fixed)
- W^{out} – output weight vector (trainable)
- W^{fb} – feedback (optional) weight vector (fixed)
- $x(t)$ – state vector of the network
- $f(\cdot)$ – tanh activation function (for example)

$$x(t+1) = f\left(W^{in} \cdot u(t) + W \cdot x(t) + W^{fb} \cdot y(t)\right)$$

Artificial Intelligence & Machine Learning

Class 1

Introduction to AI/ML

What is AI?

According to wikipedia: the ability of a computer to learn from specific data or experimental observation.

In general: AI describes a set of nature-inspired computational methodologies & approaches to address complex problems which are hard to solve or model mathematically or through traditional methods.

But what is Intelligence?

60's
- Chess playing as ultimate challenge now even a phone CPU using searching & sorting can do the task

2016
- A PC using a programmed ANN beats GO world master 4/6 games

But why is it artificial?

Transfering principles of computation & data processing from nature to computer systems. But also putting math at work in statistical learning.

Intelligence can be viewed as a (superposition) (sequence) of steps applied to an input, by a system, to compute an output. → See robotics for example!

In computational theory: A Turing machine = a mathematical model of a hypothetical computing machine that uses a set of rules to determine a result from a set of input variables. ⟹ The Turing Test

Focus of the class:

⟹ Sorting & searching in graphs (Traditional computation)

- * Strategies for sorting
- * pros & cons at scale
- * graph traversals / graph processing
- * dynamic programming

⟹ Neural networks (Supervised learning)

- * from biology to technical systems
- * learning in artificial neurons
- * from single neurons to neural networks
- * learning in neural networks backpropagation
- * supervised learning tips & tricks

Tasks:

- * regression
- * classification

⟹ Unsupervised learning

- * introduction to unsupervised learning
- * learning structure from data: SOM, VQ, RBF
- * extract underlying clusters: VQ
- * reconstruct distorted patterns after learning them: Hopfield

⟹ Deep Neural Learning

* Fundamentals of deep learning
* Common architectures for deep learning
* Building blocks of deep networks
* Major deep network architectures

⟹ Technical implementations

* Recurrent neural networks LSTM, ESN, LSM
* Time series prediction (ARMA, NARMA)
* SVM

⟹ Reinforcement Learning

* Introduction to RL, how is it different from supervised & unsupervised learning
* Q-learning as a practical way to instantiate RL

⟹ Evolutionary Programming

* Introduction to evolutionary programming
* Genetic algorithms

⟹ Fuzzy Inference Systems

* Introduction to fuzzy logic
* Fuzzy control systems

⟹ Online distributed streaming machine learning

* Big Data Analytics
* Stream processors & building online ML on top
* ML for Big ~~Data~~ on streams

  → VHDT classifiers
  → AMR Regressors
  → Perceptrons

⟹ Spiking Neural computation

– computing with biologically plausible neurons

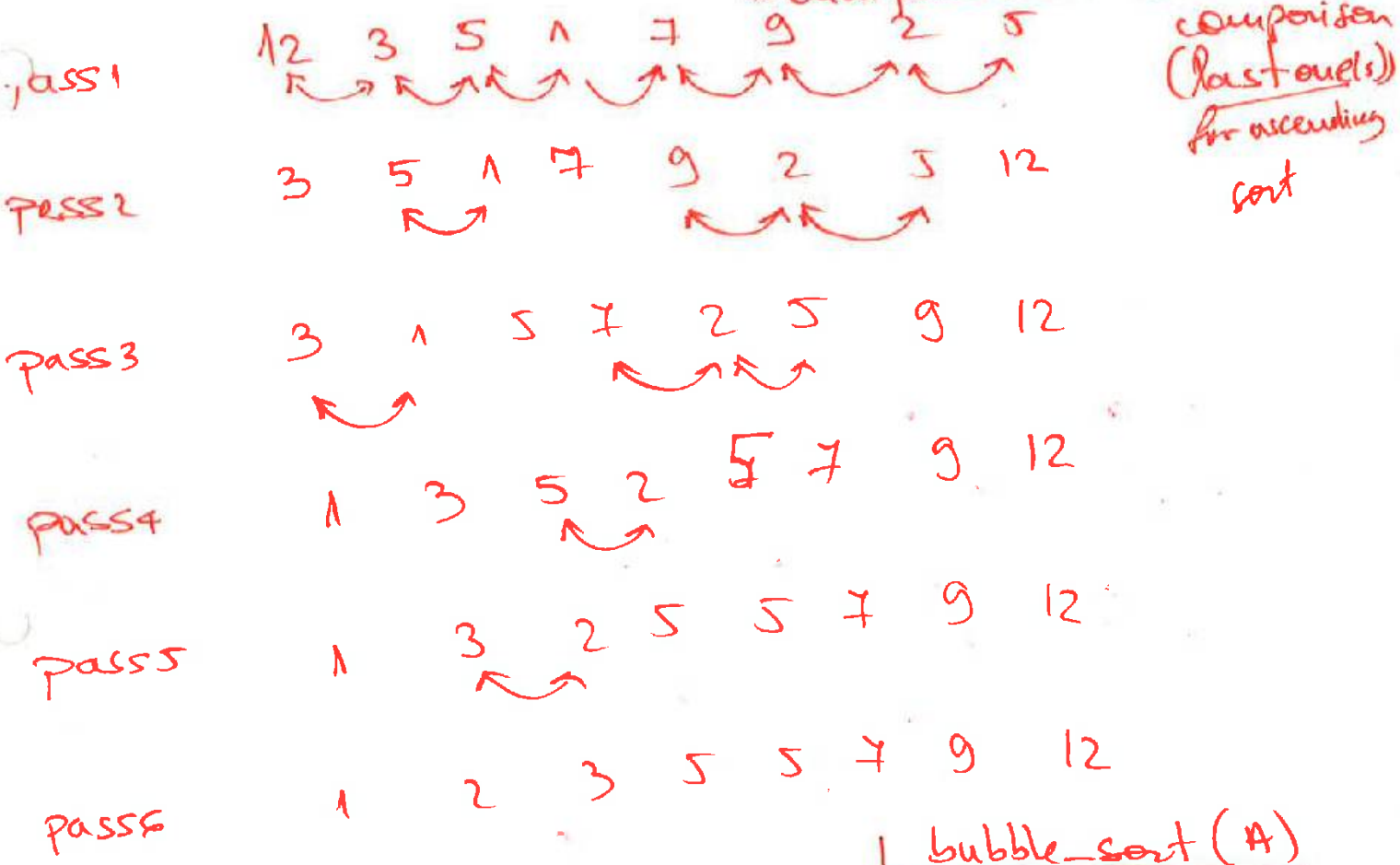
Traditional Computation

Sorting : organizing elements in a sequence in a certain order ⟹ metric is complexity, the Big O notation for time execution

⟹ efficient execution typed resource-wise

1. Bubble sort

* start and compare adjacent elements and swap if not in order
* each pass all el. doesn't need comparison (last one(s)) for ascending sort



→ 15 moves
→ 42 compares

```
bubble_sort(A)
do
    swapped = false
    for i = 1 to len(A)
        if(A[i-1] > A[i]) then
            swap(A[i-1], A[i])
            swapped = true
        endif
    endfo
while(swapped)
```

* Wast case runtime $\Theta(n^2)$
* overhead for large, nearly-sorted data

2. # Insertion sort

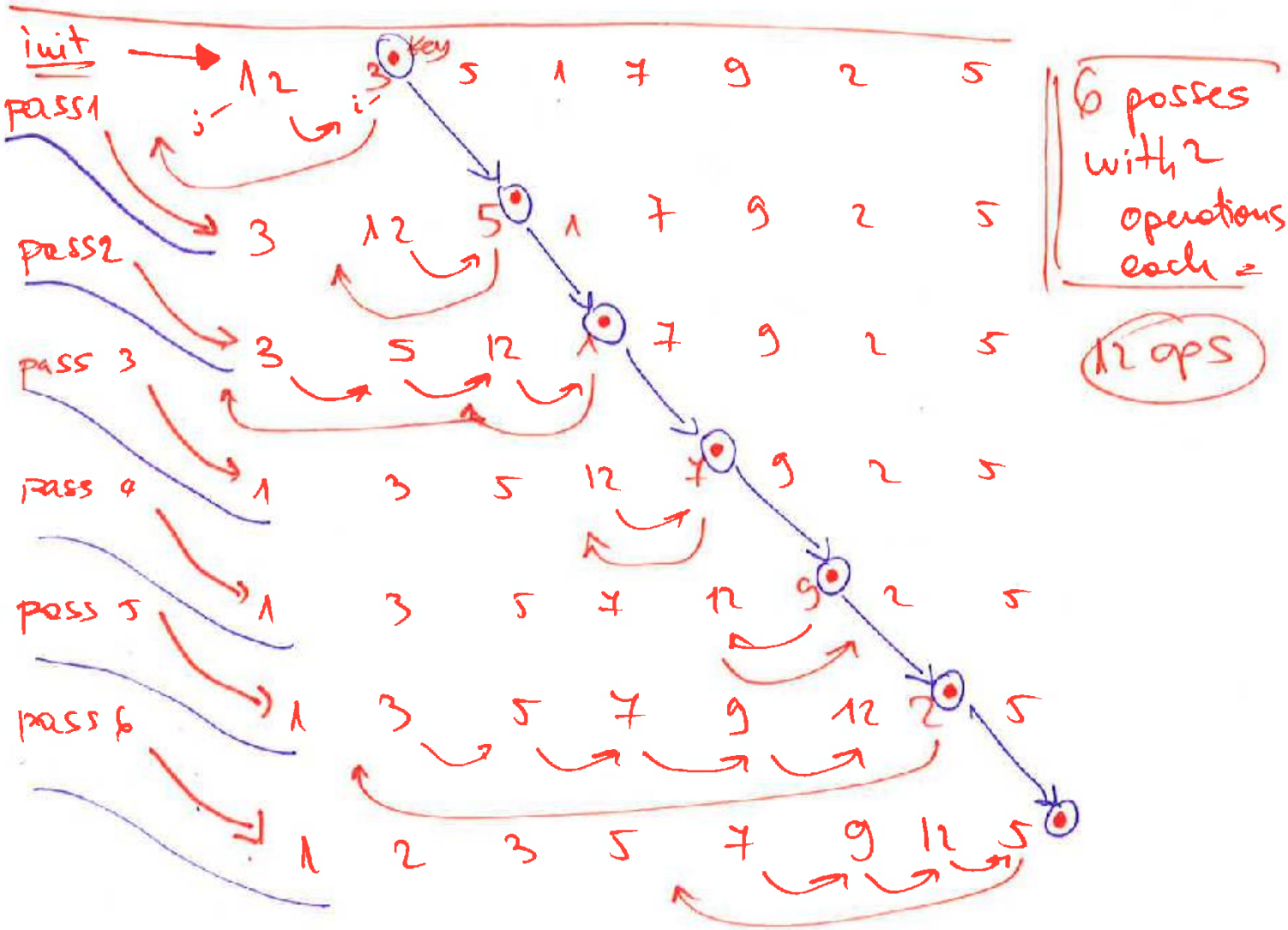
```
insertion_sort(A)
for i = 1 to len(A)
    keyval = A[i]
    j = i-1
    while (A[j] > keyval && j >= 0)
        A[j+1] = A[j]
        j = j-1
    end while
    A[j+1] = keyval
end for
```

Reversed order

worstcase runtime : $O(n^2)$

best runtime : $O(n)$

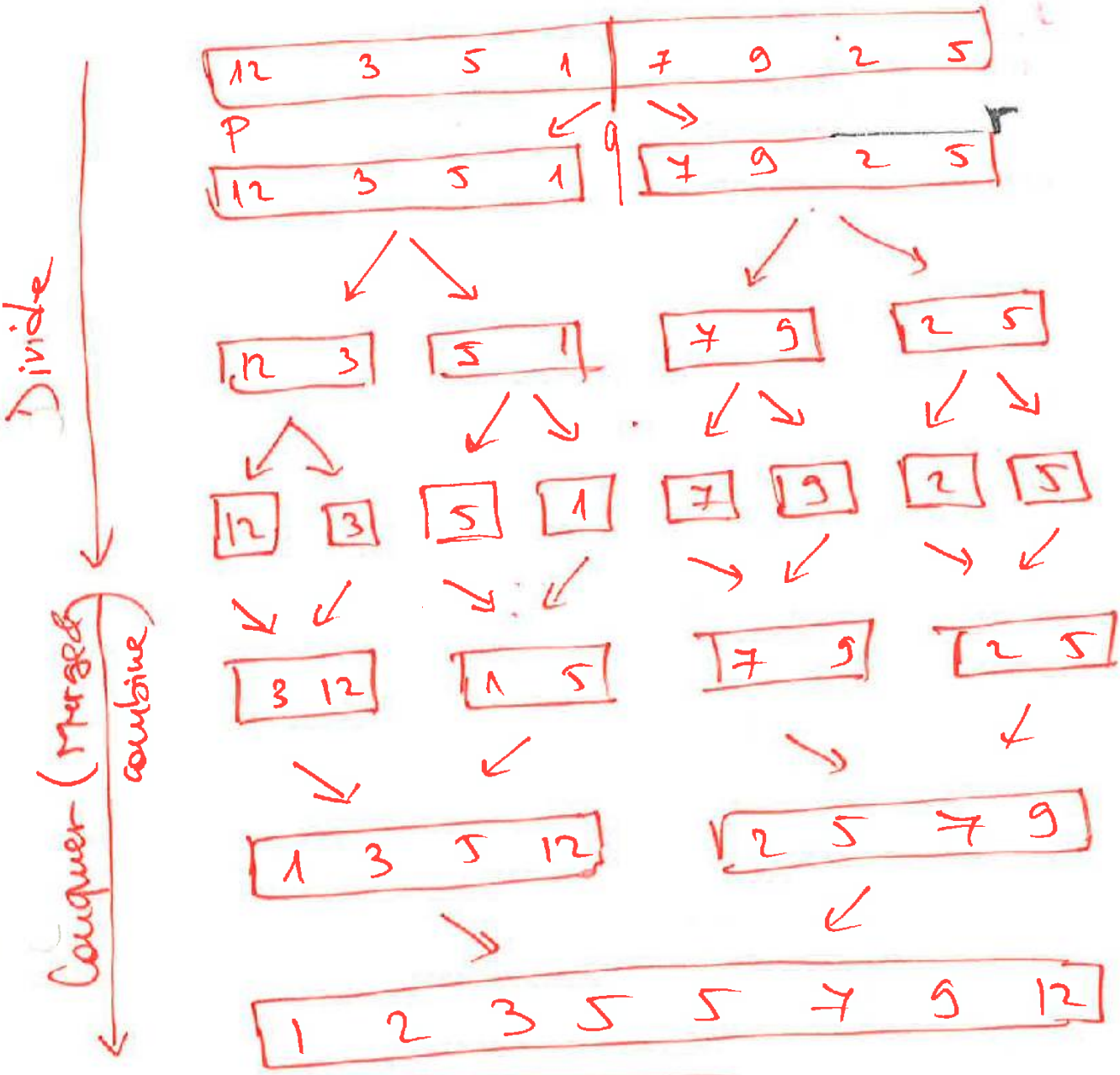
already sorted



(3.) # Merge Sort

Analysis:
- 47 comparisons
- 24 moves

* Divide & Conquer
* uses recursion



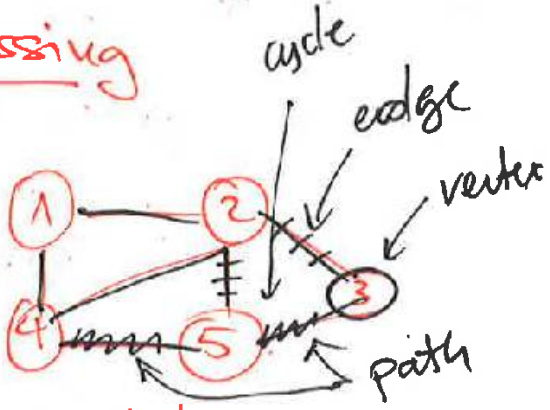
Code

A[ ] = input array
p = left limit
r = right limit
q = middle
L = left partition
R = right partition

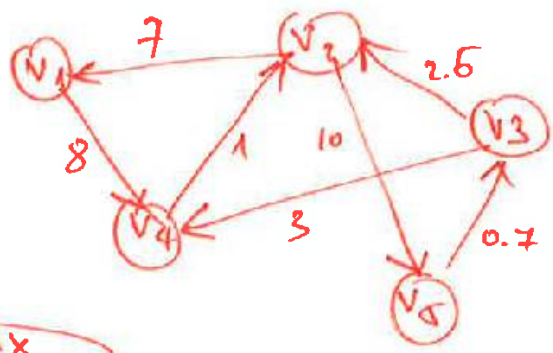
```
merge_sort (A, p, r)
    if (p < r)
        q = (p+r)/2
        merge_sort (A, p, q)
        merge_sort (A, q+1, r)
        merge (A, p, q, r)
endif
```

```
merge (A, p, q, r)
    n1 = q - p + 1      ; left partition size
    n2 = r - q          ; right partition size
    for i = 1 to n1
    |       L[i] = A[p + i - 1];
    end for
    for j = 1 to n2
    |       R[j] = A[q + j];
    end for
    i = j = 1
    for k = p to r
    |       if L[i] <= R[j]
    |       |       A[k] = L[i]
    |       |       i++
    |       else
    |       |       A[k] = R[j]
    |       |       j++
    |       end if
    end for
```

Graphs & graph processing



- a powerful representation & processing tool;
- elements:
  - * Vertex = node, represented by a key and also some additional information (payload)
  - * edge = arc, connecting vertices & depicts a relationship between them can be one/two way
  - * weight = arcs/edges can be weighted so that there is a cost associated with an edge → can have different meanings
  - * path - a sequence of vertices connected by edges in a graph
  - * Cycle = a path that starts & ends in the same vertex

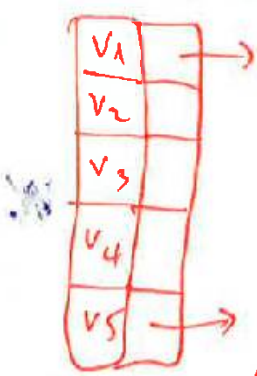


Representing connectivity:

Adjacency matrix

	v1	v2	v3	v4	v5
v1		7		8	
v2	7		2.5	1	10
v3		2.5		3	0.7
v4	8	1	10		
v5		10	0.7		

Adjacency list



id = "v1"
adj = {v2:7, v4:8}

id = "v5"
adj = {v2:10, v3:0.7}

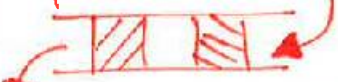
AI gorithms on graphs

- graphs can model many types & processes in information systems, both biological & technical
- graph traversal is an interesting problem and refers to visiting all nodes in a graph
  BFS, DFS, Dijkstra, A*

BFS (Breadth First Search)

```
BFS(G, s)
    s.parent = null
    s.color  = grey
    s.distance = 0
    enqueue(Q, s)
    while Q != { }
        n = dequeue(Q)
        for v = n.allNeighbors
            if v.color == white
                v.color = gray
                v.distance = n.distance + d(n,v)
                v.parent = n
                n.neighbors = n.neighbors + v
                enqueue(Q, v)
            endif
        n.color = black
        end for
    end while
```

queue Q (FIFO)

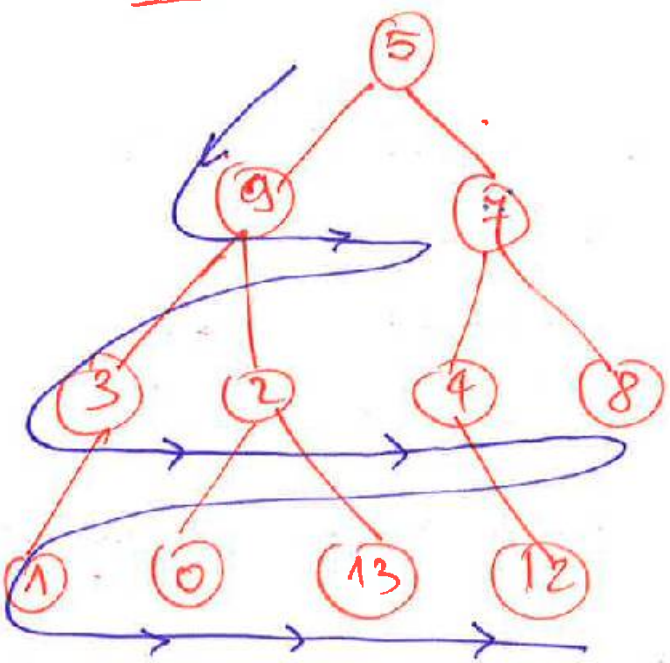


* BFS colors the vertices (graph.color.v):
* white - undiscovered
* gray - visited unprocessed
* black - visited & processed

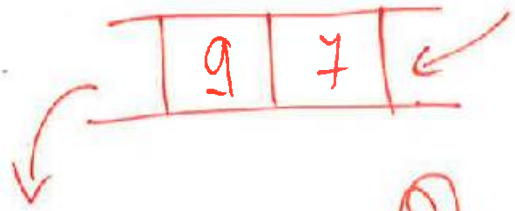
Each node represented by triplets:

- parent id
- color
- distance

Example



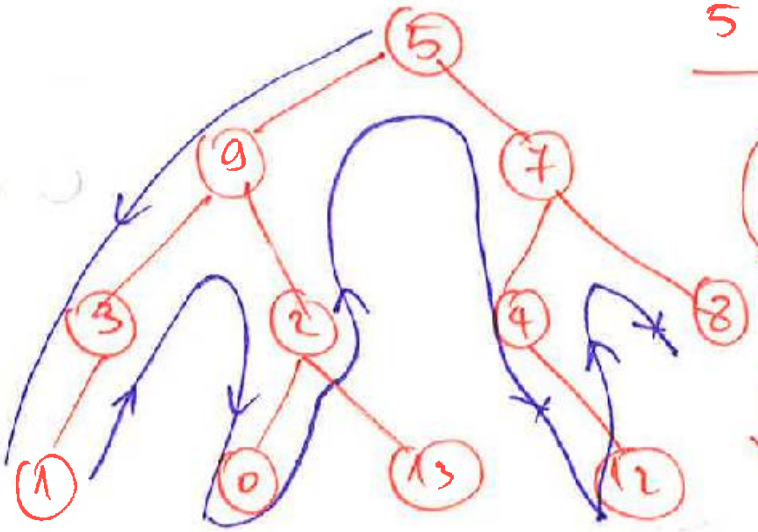
5 9 7 3 2 4 8 1 0 13 12



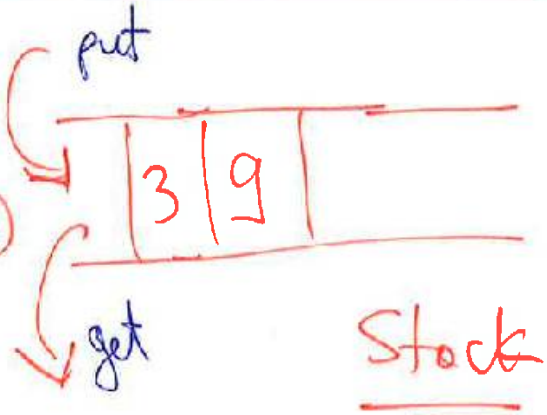
Queue

Depth First Search (DFS)

- Similar to BFS but add vertices at the beginning instead of the end of the queue
- uses recursion



5 9 3 1 2 0 13 7 4 12 8



Stack

Dijkstra's algorithm

- algorithm to compute shortest path among vertices in a graph
- extends BFS with associating a cost on each edge => Cheapest First Search (CFS)

Elements : each vertex has 3 embedded properties
- color = {white, grey, black}
- cost = {positive numeric value}
- parent = id of the parent node

Algorithm:

```
n(all).color = white      ; all nodes unvisited & unprocessed
n(all).cost = ∞           ; cost unknown
n(s).parent = 0           ; start node has no parent
Q = {all nodes}           ; all nodes are in the queue
while Q != {} {
    n = dequeue(Q)        ; extract a node
    n.color = grey        ; visited & not processed
    for c = n.all_children
        if c.cost > n.cost + edge(c,n)
            c.cost = n.cost + edge(c,n)
            c.parent = n
        endif
    end for
    n.color = black
end while
```

AIML Class — Genetic Algorithms & Evolutionary Programming

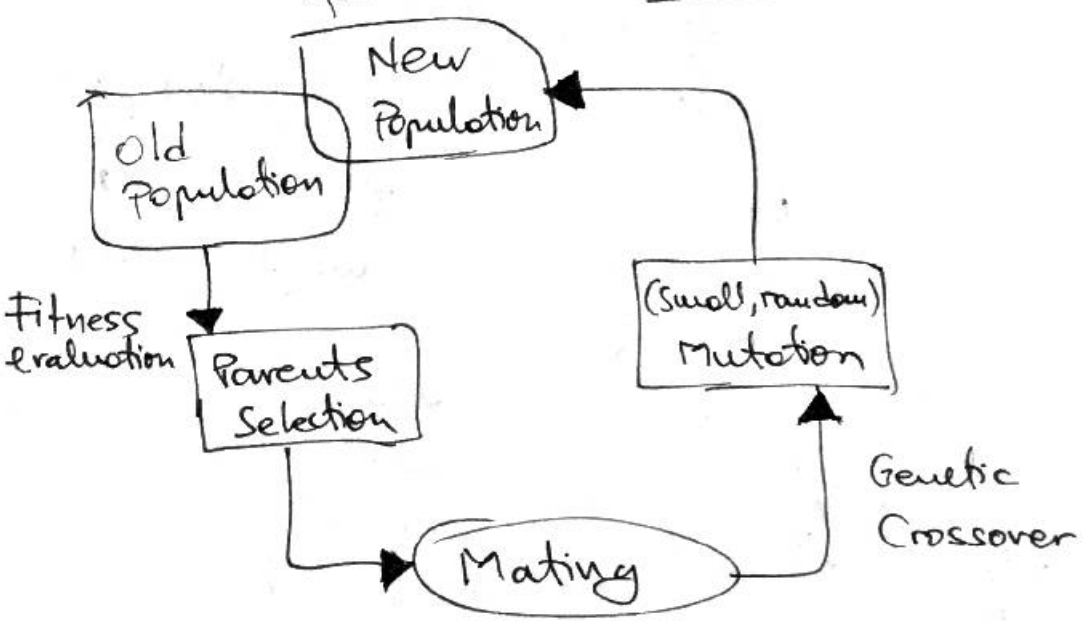
GA & EP

⊛ Evolutionary Programming (EP) is a method for simulating evolution: rooted in Darwin's (1859) theory of evolution, natural selection and survival of the best/fittest.

Level of evolution

- there are 3 main lines of research:
  1. Genetic Algorithms (GA) * chromosome
  2. Evolution Strategies (ES) * individual
  3. Evolutionary Programming (EP) * specie

- the core idea of all of them:
  - population of competing solutions subject to random alterations.
  - competition to be retained as parents for successive reproduction epochs.



- EP is an abstraction from biological theory of computation/evolution used for optimization problems.

_EP emerged as an alternative to AI:

- result emulate neural computation or human behaviors.
- models processes that generate organisms or increase intellect over time:
  - intelligence defined here as ability to achieve goals and evolution is optimization.

⊛ Genetic Algorithms (GA)

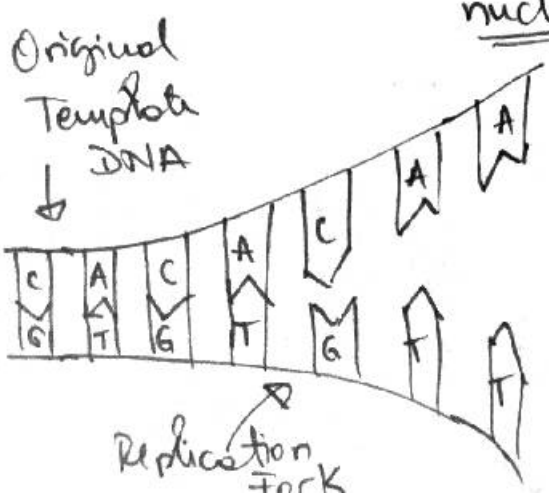
- introduced formally in 1970 by John Holland.
- attractive for optimization problems.
- work well on mixed (continuous and discrete) combinatorial problems.

Basic Formulation:

- the gene is a part of the DNA that encodes information; chromosomes contain multiple genes.
- the human genome has ~ 20500 genes.
- the DNA sequence is composed of complementary nucleotides:
  - Adenine (A)
  - Cytosine (C)
  - Guanine (G)
  - Thymine (T)
  - Uracil (U)

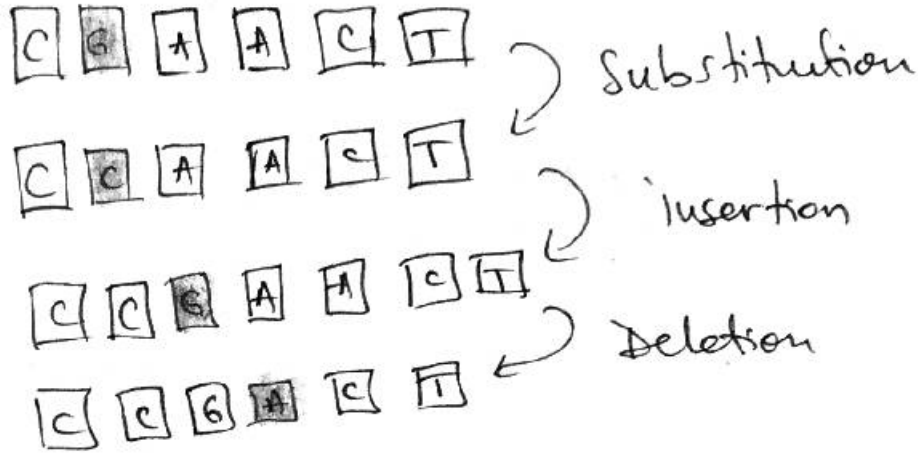
These bases form pairs between the 2 strands:

- A with T
- C with G

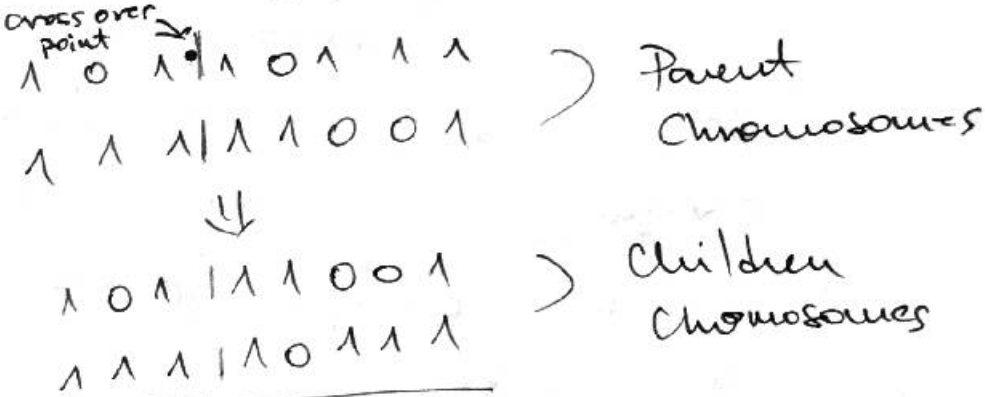


- during reproduction some processes can occur:
  - mutations (substitution, insertion, deletion).
  - cross-over.

Mutation



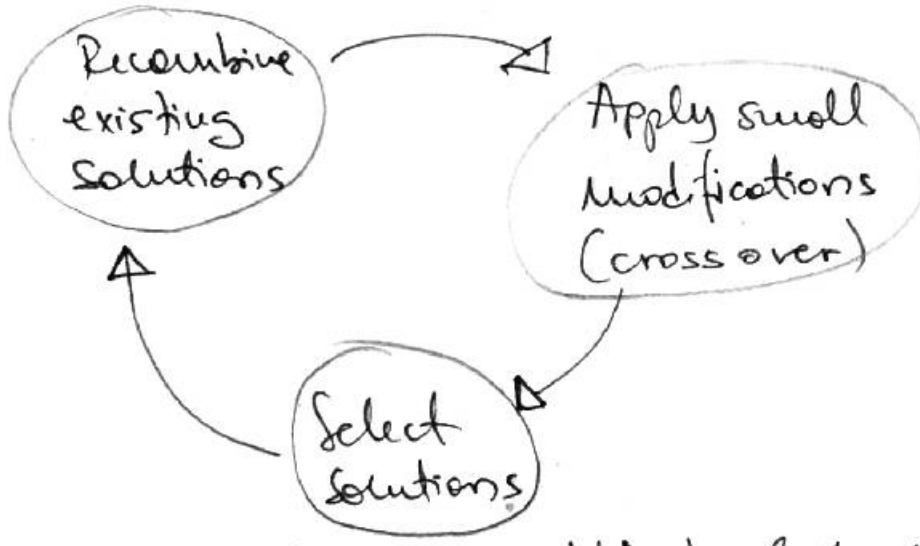
Cross-over (assume binary genes)



But how can we apply this in technology?

- consider the problem of searching a large state-space or n-dimensional surface.
- GAs offer advantages over traditional optimization techniques (BFS, DFS, Dijkstra etc.) => but they also tend to be computationally expensive.
- GAs are adaptive search algorithms based on genetics which offer an intelligent random search.

- although randomized GAs are not random
- they exploit historical information to direct the search into a region of better performance.
- it is seen as a 3-step iterative process of stochastic exploration:



- GAs are less susceptible to fall in local minima than gradient methods.
- GAs can optimize nonlinear, discontinuous functions without an analytical formulation.

Elements of GA:

4. Encoding
   - a chromosome is composed of genes and should represent the solution.
   - usually a binary string
   - encoding is problem dependent!

B. Population (of chromosomes)

- the size of the population is problem dependent
- initial population is generated randomly to allow spanning entire search space.
- the size determines the problem complexity and exploration ability.

C. Fitness function

- evaluates how "good" a potential solution is relative to others.
- usually returns a value reflecting how optimal a solution is.
- this is the most important design aspect of the GA !

D. Selection Criteria

- selecting parents for recombination and crossover.
- usually select better parents (fitness wise) to produce better children.
- uses elitism or other similar strategy
  * proportional selection;
  * tournament selection;
  * rank based selection.

Save best solution for next generation

E. Reproduction / mutation operators

- crossover operates on parent chromosomes generating offspring chromosomes

- typically choose a random crossover point and copy from first parent with the remaining genes from the second parent.
- after crossover mutation takes place.
- mutation ensures that the search doesn't fall in a local minima by randomly changing offspring resulting from crossover.

Using GAs in practice: Optimizing Nonlinear Functions

Problem: Given a nonlinear function f

$f(x) = x \cdot \sin(10\pi x) + 1$ find the x value such that $f(x)$ is maximum.

Considering $x \in [-1, 2]$, the problem assumes finding x_0 such that $f(x_0) \geq f(x)$ $(\forall) x_0 \in [-1, 2]$ and $x \in [-1, 2]$.

Let's look first at the analytical approach.

- compute the derivative $f'(x)$

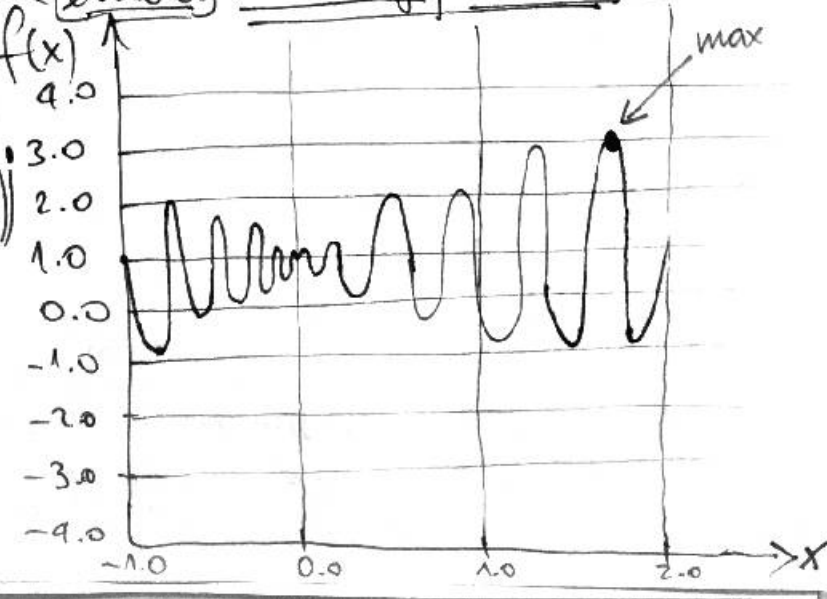
$$f'(x) = \frac{df(x)}{dx} = x' \sin(10\pi x) + x(\sin(10\pi x))'$$

$$f'(x) = 1 \cdot \sin(10\pi x) + 10\pi x \cos(10\pi x)$$

we make $f'(x) = 0$

$$\Downarrow$$

$$\sin(10\pi x) + 10\pi x \cos(10\pi x) = 0$$



we divide with $\cos(10\pi x) > 0$ so we have

$$\frac{\sin(10\pi x)}{\cos(10\pi x)} + 10\pi x = 0$$

$$\tan(10\pi x) = -10\pi x$$

This eq. has an infinite nr. of solutions given by

$$\begin{cases} x_i = \frac{2i-1}{20} + \varepsilon_i, & i = 1, 2 \dots \\ x_0 = 0 \\ x_i = \frac{2i+1}{20} + \varepsilon_i, & i = -1, -2, \dots \end{cases}$$

Because we look at $x \in [-1, 2]$ then $f(x)$ reaches the max for: $x = 1.85 \Rightarrow f(1.85) = 1.85 \cdot \sin(18.54) + 1$

$$f(1.85) = 1.85 + \sin\left(18\pi + \frac{\pi}{2}\right) + 1$$

$f(1.85) = 2.85$ which is the max value

The GA approach

Step 1 Encoding

A. Use binary encoding to represent x.

B. choose the dimension (nr. of bits/gene)

for our case: we have 3 types of values (see analytic formulation)

⊙ we would like 6 places after the decimal point (precision)

— so, we should split the $[-1,2)$ interval in 3×10^6 equal sized ranges.

$$2^{21} = 2097152 < \overbrace{3000000}^{3 \times 10^6} < 2^{22} = 4194304$$

$$\Downarrow$$

So, we use a 22 bit/genes representation for chromosomes

$$\langle b_{21}\, b_{20}\, b_{19} \dots b_0 \rangle$$

c. perform the mapping from binary representation of chromosomes to real values $[-1,2]$

step 1: convert binary to decimal

$$\left(\langle b_{21} b_{20} \dots b_0 \rangle\right)_2 = \left(\sum_{i=0}^{21} b_i \cdot 2^i\right)_{10} = X^*$$

step 2: encode value in interval $[x_{min}, x_{max}]$

$$x^* = (x - x_{min}) \cdot \left(\frac{2^n - 1}{x_{max} - x_{min}}\right)$$

where $\begin{cases} x_{min} = -1 \\ x_{max} = 2 \end{cases}$ in our case

$$\Downarrow$$

$$\boxed{x = x_{min} + x^* \cdot \frac{x_{max} - x_{min}}{2^n - 1}}$$

and for our case

$$\boxed{x = -1 + x^* \frac{3}{2^{22} - 1}}$$

ex: to which x the next chromosome corresponds?

$$c = (\underbrace{1}_{b_{21}}\underbrace{0}_{b_{20}}\underbrace{0}_{b_{19}}\ 0\ 1\ 0\ 1\ 1\ 1\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ 0\ 0\ 0\ \underbrace{1}_{b_2}\underbrace{1}_{b_1}\underbrace{1}_{b_0})$$

$$x^* = 1\cdot 2^{21} + 0\cdot 2^{20} + 0\cdot 2^{19} + \dots + 1\cdot 2^{4} + \dots + 1\cdot 2^{2} + 1\cdot 2^{1} + 1\cdot 2^{0} \Rightarrow$$

$$x^* = 2288967$$

But we know that $x = -1.0 + x^* \frac{3}{2^{22}-1}$

$$\Downarrow$$

$$x = -1.0 + \frac{3 \cdot 2288967}{2^{22}-1}$$

$$x = 0.637197.$$

We can easily see

$$c = (\underbrace{0 \_\_\_ 0}_{22 \text{ bits/genes}}) = -1$$

$$c = (\underbrace{1 \_\_\_ 1}_{22 \text{ bits/genes}}) = 2.$$

Step 2 Initialize the population

- randomly initialize the 22 bits in each chromosome of the population.
- define the number of chromosomes in the population.

Step 3 Define fitness function, e

- e is applied on binary values c and is equivalent to function f: $e(c) = f(x)$

- e can be seen as the environment rating potential solutions in terms of their fitness.

e.g: Consider 3 chromosomes:

$$\begin{cases} C_1 = (1\,000\,10\;\;111\,0\,1\,1\,0\,1\,0\,1\,000\;\;111) \\ C_2 = (000\,000\;\;111\,000\,000\,000\,1\,0\,0\,000) \\ C_3 = (111\,000\;\;000\,0\,111\,111\,000\,1\,0\,1) \end{cases}$$

corresponding to

$$\begin{cases} x_1 = 0.637197 \\ x_2 = -0.958073 \\ x_3 = 1.627888 \end{cases}$$

for which the evaluation / fitness function gives the following results:

$e(c_1) = f(x_1) = 1.586345$

$e(c_2) = f(x_2) = 0.078878$

$e(c_3) = f(x_3) = 2.250650$

↖ chromosome 3 is the fittest!

Step 4 | Crossover

e.g.
- consider c2 and c3 for crossover.
- the crossover point is after gene / bit index 17.

crossover point bit 17

$c_2 = (00000 \mid 011100000001000000)$

$c_3 = (11100 \mid 00000111111000101)$

⇓ crossover resulting children

$c_2' = (00000 \mid 00000111111000101)$

$c_3' = (11100 \mid 011100000001000000)$

- if we evaluate the fitness of the 2 children

$e(c_2') = f(x_{c_2}') = f(-0.998113) = 0.940$

$e(c_3') = f(x_{c_3}') = f(1.66) = 2.459$

$e(c_2) = 0.078878$

$e(c_3) = 2.250650$

Second offspring has a better fitness than both parents!

Step 5 [Mutation]

- alters one/more genes with a probability equal to the mutation rate

e.g: assume que 17th of c_3 was selected for mutation:

$c_3 = (11100000001111111000101)$

⇓ mutation

$c_3' = (11101000001111111000101)$

$$x_3' = -1.0 + x_3'^{*} \frac{3}{2^{22}-1} = 1.721638$$

$$e(c_3') = f(x_3') = -0.082257$$

but $e(c_3) = 2.250650$

⇓

in this case: mutation determined a decrease of the fitness value!

- If we mutate gene 11 in chromosome c_3:

$$c_3'' = (111000000\textcircled{1}111111000101)$$

with

$$x_3'' = 1.630818 \Rightarrow e(c_3'') = f(x_3'') = 2.343555$$

⇓

in this case: improvement of fitness over initial value!

Generic GA algorithm pseudocode

#1. Initialize chromosomes in the population P
#2. Evaluate $e(c) = f(x)$
#3. Select "good" individuals
#4. Repeat
#5. for $i = 0$ to λ (# of children to produce)
#6. select P_1, P_2 parents out of P
#7. generate x_i by crossover P_1, P_2
#8. mutate x_i
#9. evaluate $f(x_i)$
#10. add x_i to P'
#11. end select new P out of P' (different selection strategies)
#12. forget P'
#13.
#14. Until fitness is "good" enough

In our example to maximize $f(x) = x \sin(10\pi x) + 1$ after 150 generations the fittest chromosome is

$c_{max} = (11110011010001000000101)$ which corresponds to $x_{max} = 1.850000$ and $f(x_{max}) = 2.850000$.

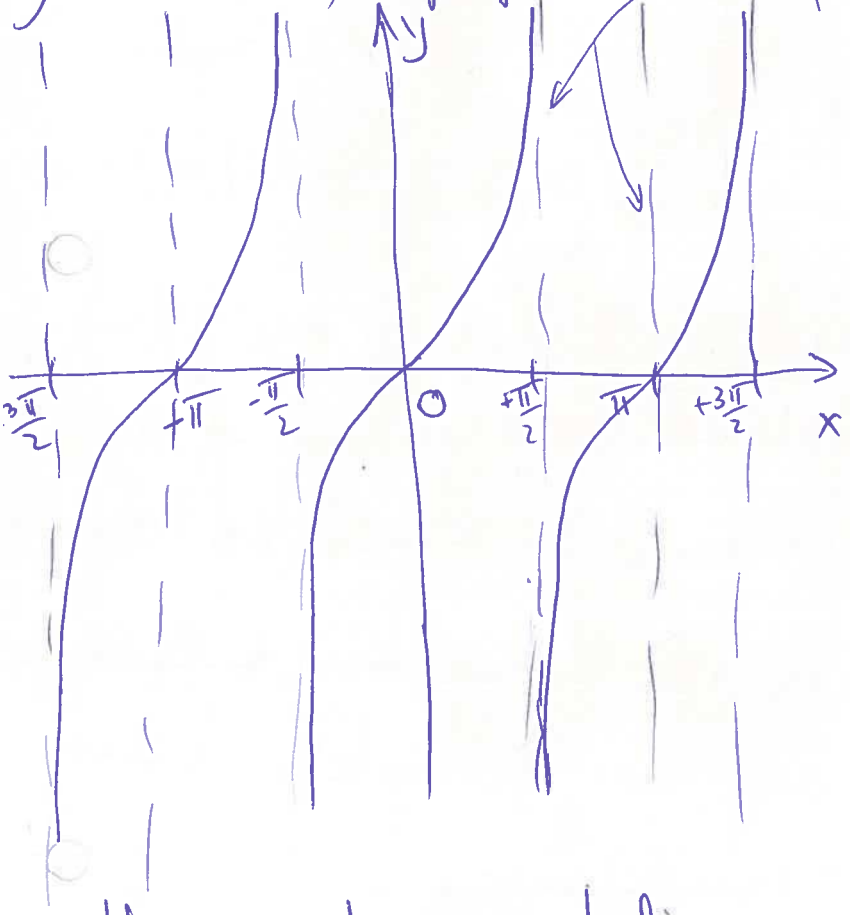
Trigonometric equation proof

Appendix 1

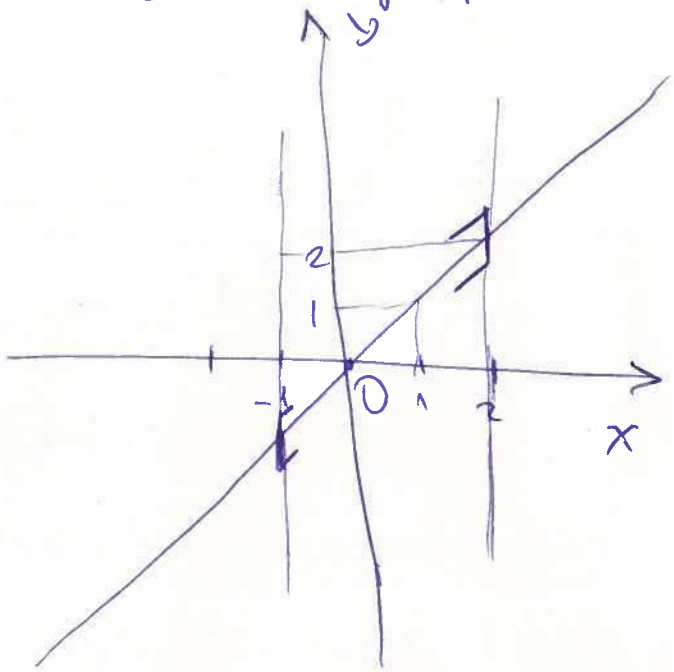
$\tan(10\pi x) = -10\pi x$ brought to the simplest form $\tan(x) = x$

$x \in [-1, 2]$

$y = \tan(x)$ graph asymptotes



$y = x$ graph



the eq. has a solution in every interval of the following form

$$\left[\frac{(2k-1)\pi}{2}, \frac{(2k+1)\pi}{2}\right] = I(k)$$

$k \in \mathbb{N}$

$\Leftrightarrow 0$

$$\frac{(2k-1)\pi}{2} < x < \frac{(2k+1)\pi}{2}$$

When x is in the $I(k)$ and since $x = y$ is clear that y is in the $I(k)$ and so

$$|y| < \frac{(1+2*|k|)*\pi}{2}$$

So if we put things together

– for values of x close to the left end of $I(k)$

$$\tan(x) < -\frac{(2|k|+1)\pi}{2} < x \Rightarrow \tan(x) - x < 0$$

– for values of x close to the right end of $I(k)$

$$\tan(x) > \frac{(2|k|+1)\pi}{2} > x \Rightarrow \tan(x) - x > 0$$

Since $\tan(x)$ is continuous on $I(k)$ there must be at least one value of x between where $\tan(x) - x = 0$ so $\tan(x) = x$

so if we consider we have one solution for each k there are infinitely many solutions to $\tan(x) = x$.

Appendix 2 Generic mapping function n-bits of represent.

$$X^* = (X - X_{min}) \left(\frac{2^n - 1}{X_{max} - X_{min}} \right)$$

$$X^* = \frac{X(2^n - 1)}{X_{max} - X_{min}} - \frac{X_{min}(2^n - 1)}{X_{max} - X_{min}}$$

$$X^*(X_{max} - X_{min}) = (2^n - 1)(X - X_{min})$$

$$X - X_{min} = \frac{X^*(X_{max} - X_{min})}{2^n - 1}$$

$$X = X_{min} + \frac{X^* \cdot (X_{max} - X_{min})}{2^n - 1}.$$

Fuzzy Logic

Ex① Worked example on Mamdani FIS.

Evaluate the risk of a software engineering project by refering only 2 criteria, the project funding and the project staffing.

Step 1 : I/O variables identification

2 inputs : project funding (%) (PF)
project staffing (%) (PS)

1 output : project risk (%) (PR)

---

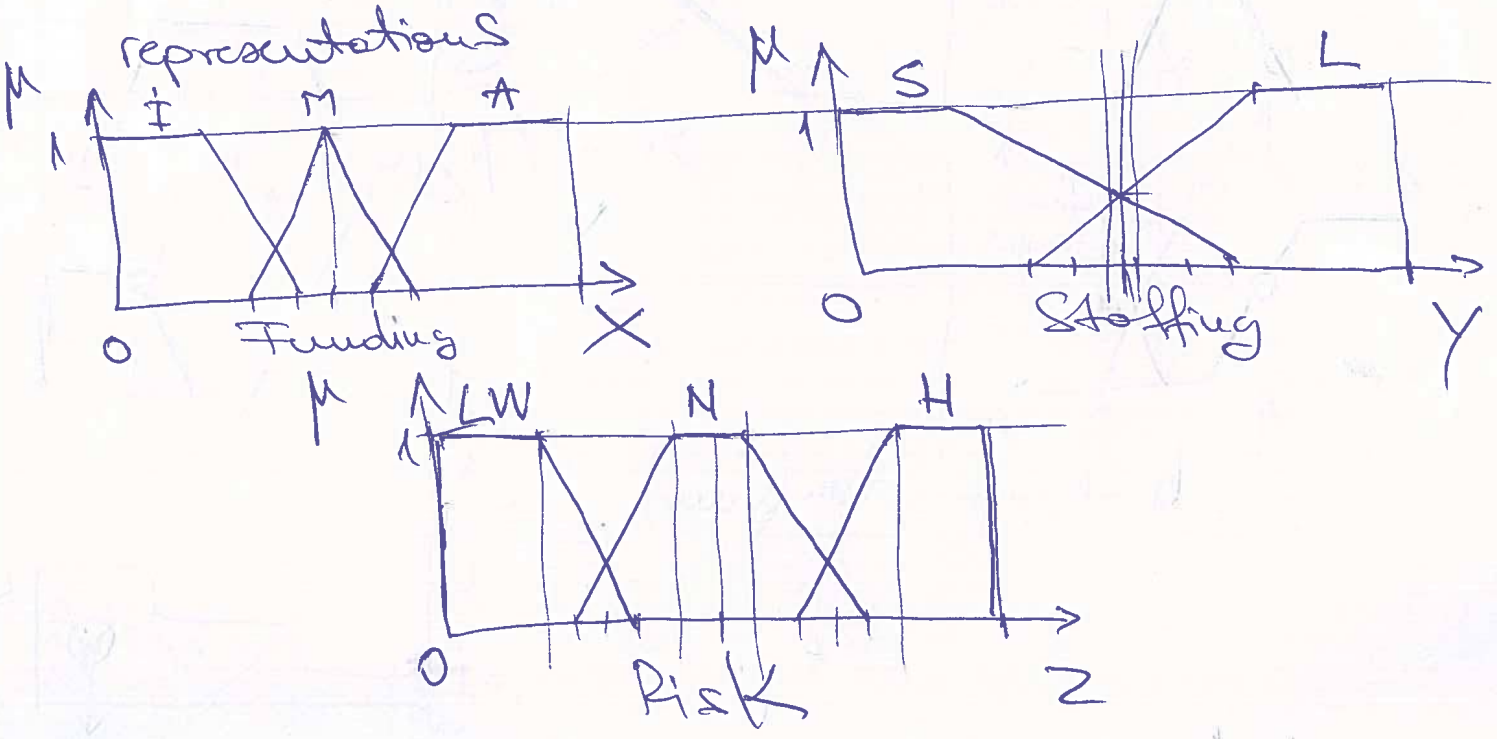
universe of discourse ⟶ fuzzy sets

X = { inadequate (I), marginal (M), adequate (A) } for PF

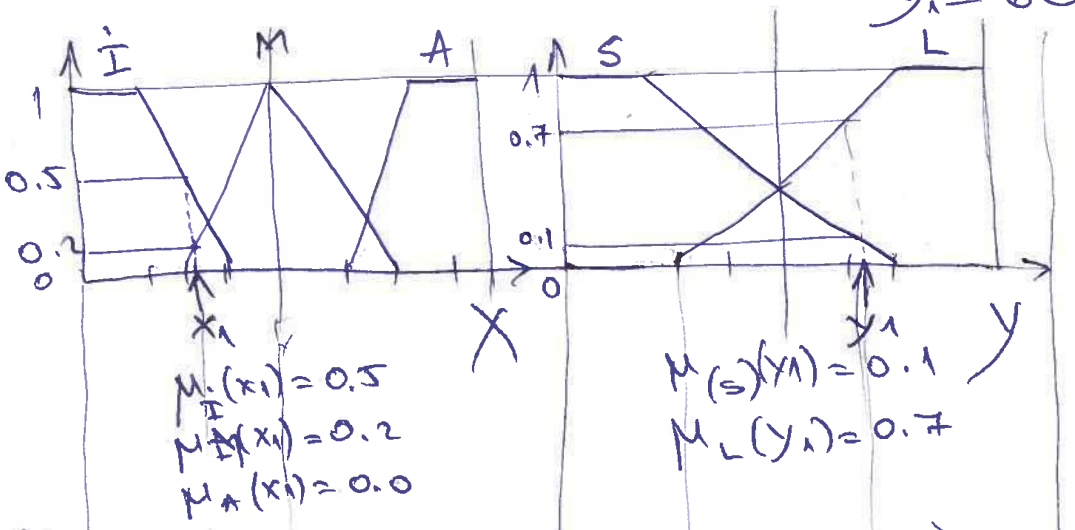
Y = { small (S), large (L) } for PS

Z = { low (LW), normal (N), high (H) } for PR

representations



Step2 Fuzzification ex $x_1 = 35\%$ PF
$y_1 = 60\%$ PS



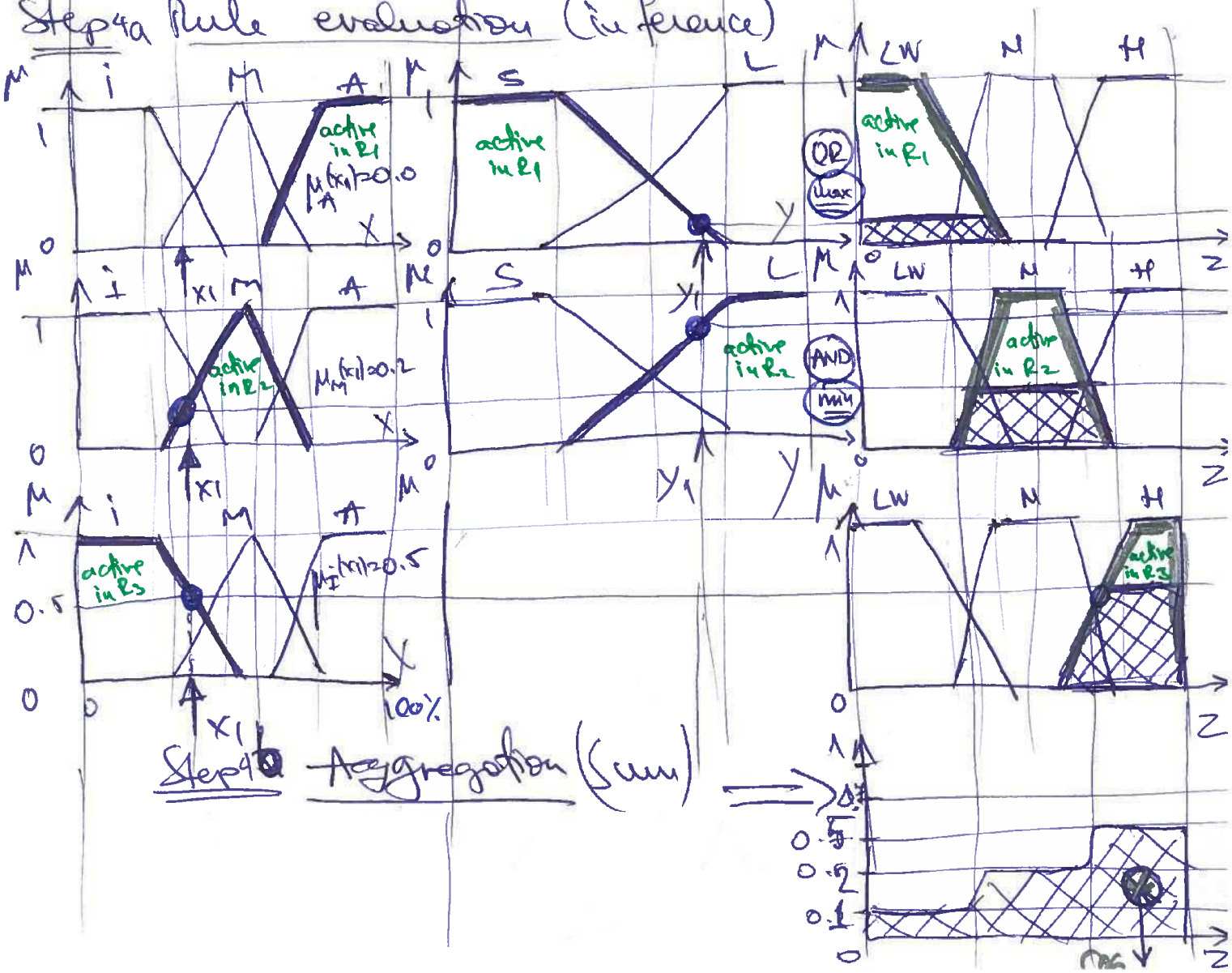
Step3 Rule design (example rules)

R1: IF PF is A (OR) PS is S THEN PR is LW.

R2: IF PF is M (AND) PS is L THEN PR is N.

R3: IF PF is I THEN PR is H.

Step4a Rule evaluation (inference)

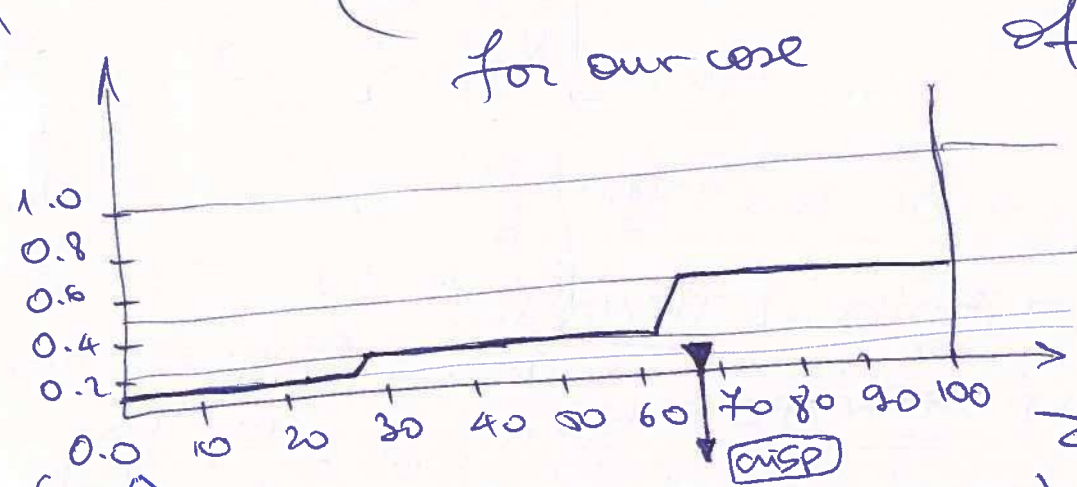


Step4b Aggregation (sum)

Step 5 Defuzzification (using COG)

$$COG = \frac{\int_a^b \mu_A(x)\,x\,dx}{\int_a^b \mu_A(x)\,dx} \longrightarrow COG = \frac{\sum_{x=a}^{b} \mu_A(x)\,x}{\sum_{x=a}^{b} \mu_A(x)}$$

(continuous) for our case — Over a sample of points (discrete)



(0%) ... (100%)

$$COG = \frac{(0+10+20)\times 0.1 + (30+40+50+60)\times 0.2 + (70+80+90+100)\times 0.5}{0.1+0.1+0.1+0.2+0.2+0.2+0.2+0.5+0.5+0.5+0.5} =$$

$$COG = 67.4\ (\%)$$

Ex ② Analogous problem solved with TSK FIS

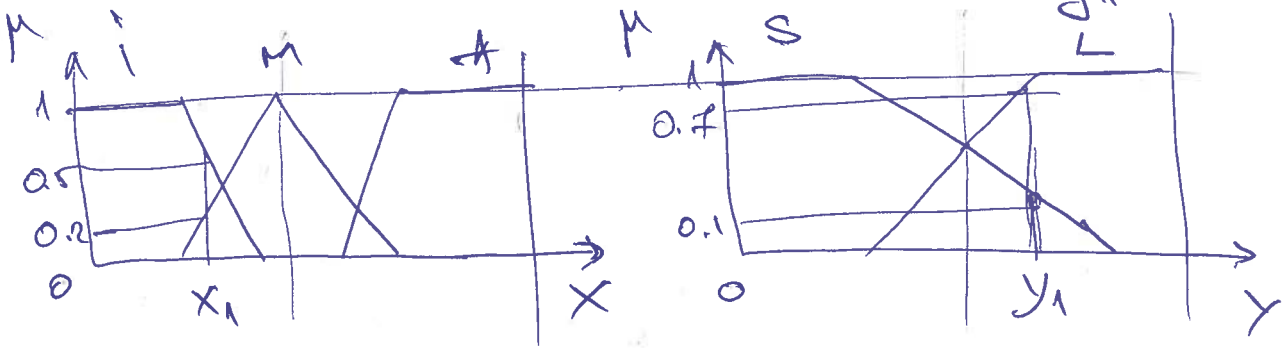
- the inference time is shortened by eliminating the defuzzification and by using singletons as outputs
- Singleton = fuzzy set with $\mu = 1$ at a single point in the universe of discourse and 0 elsewhere.

The output of the TSK is a function of the input variables!

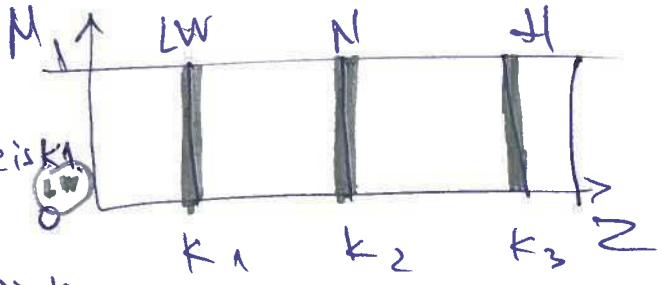
Step 2 Fuzzification

ex $x_1 = 35\%$ PF

$y_1 = 60\%$ PS



Step 3 Rule design



R_1: IF PF is A (OR) PS is S THEN PR is k_1 (LW)

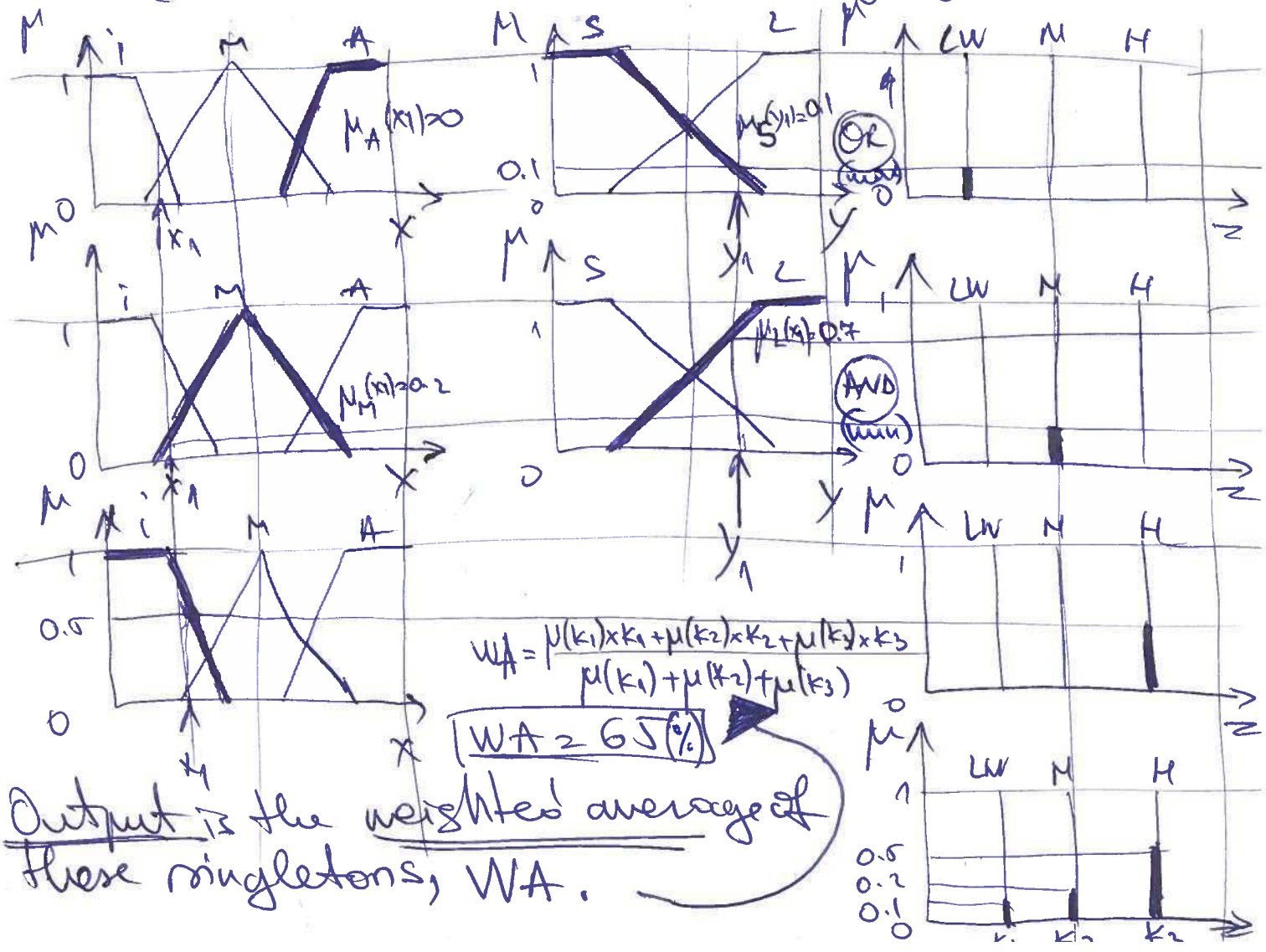
R_2: IF PF is M (AND) PS is L THEN PR is k_2. (N)

R_3: IF PF is I THEN PR is k_3. (H)

Output representation in TSK (singletons)

k1 = 20, k2 = 50, k3 = 80

Step 4 Rule evaluation & aggregation



$$WA = \frac{\mu(k_1)\times k_1 + \mu(k_2)\times k_2 + \mu(k_3)\times k_3}{\mu(k_1)+\mu(k_2)+\mu(k_3)}$$

$$\boxed{WA = 65\%}$$

Output is the weighted average of these singletons, WA.

AIML Course 9 | Fuzzy Logic & Fuzzy Inference Systems

Intro:

- Data representation and computation inspired by the way humans describe phenomena through language
- Uses imprecission and involves modifiers of linguistic terms (e.g. "quite, fairly, very") and uncertainty:

Q: "How comfortable is this room? Is it warm or cold?"

A: "It's quite cold here.
It's fairly warm.
It's too hot for me"

- Trades off between significance and precision in computing with words.

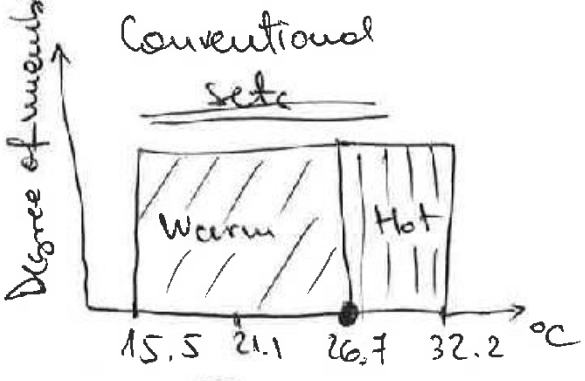
Formalism and terminology:

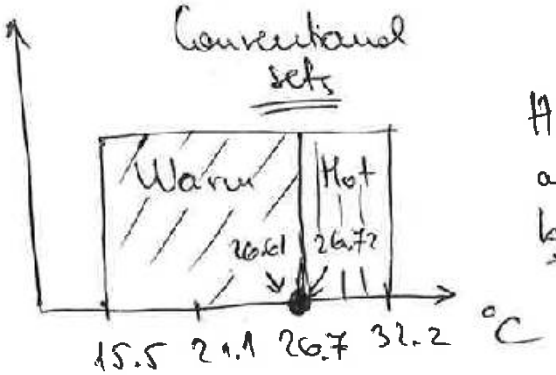
- introduced by Lotfi Zadeh, 1965, "Fuzzy Sets"
- fuzzy logic is a polivalent (multi-valued) logic
- boolean logic is a special case of fuzzy logic
- fuzzy logic originates from set theory:

  * crisp sets = conventional set, the degree of membership of any element is 0 or 1.
  * fuzzy sets = sets whose elements have degrees of membership typically between 0 and 1

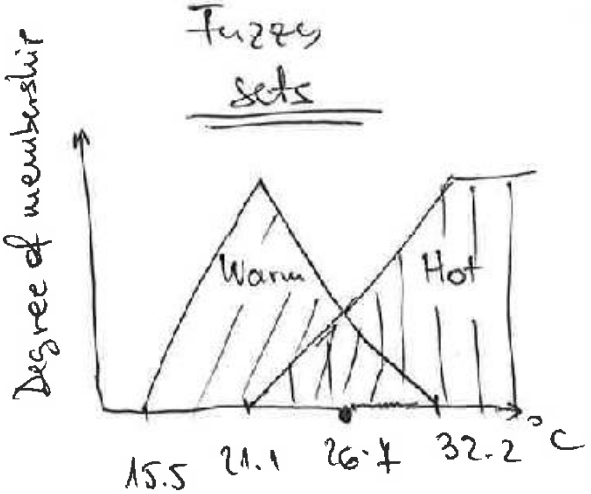
Intuition

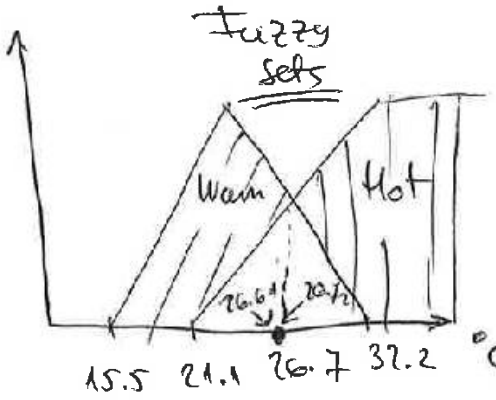
For example is 26.7°C warm or hot?





Here 26.61°C and 26.72°C belong to two different sets.

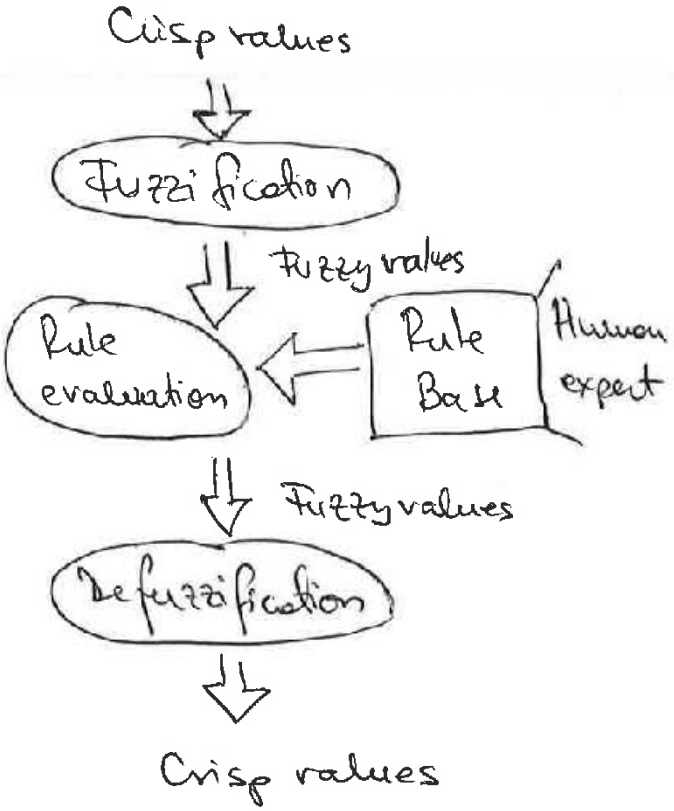




Here 26.61° and 26.72°C belong to the same sets to approxim. the same degree.

Crisp values (temperature)

Using Fuzzy logic

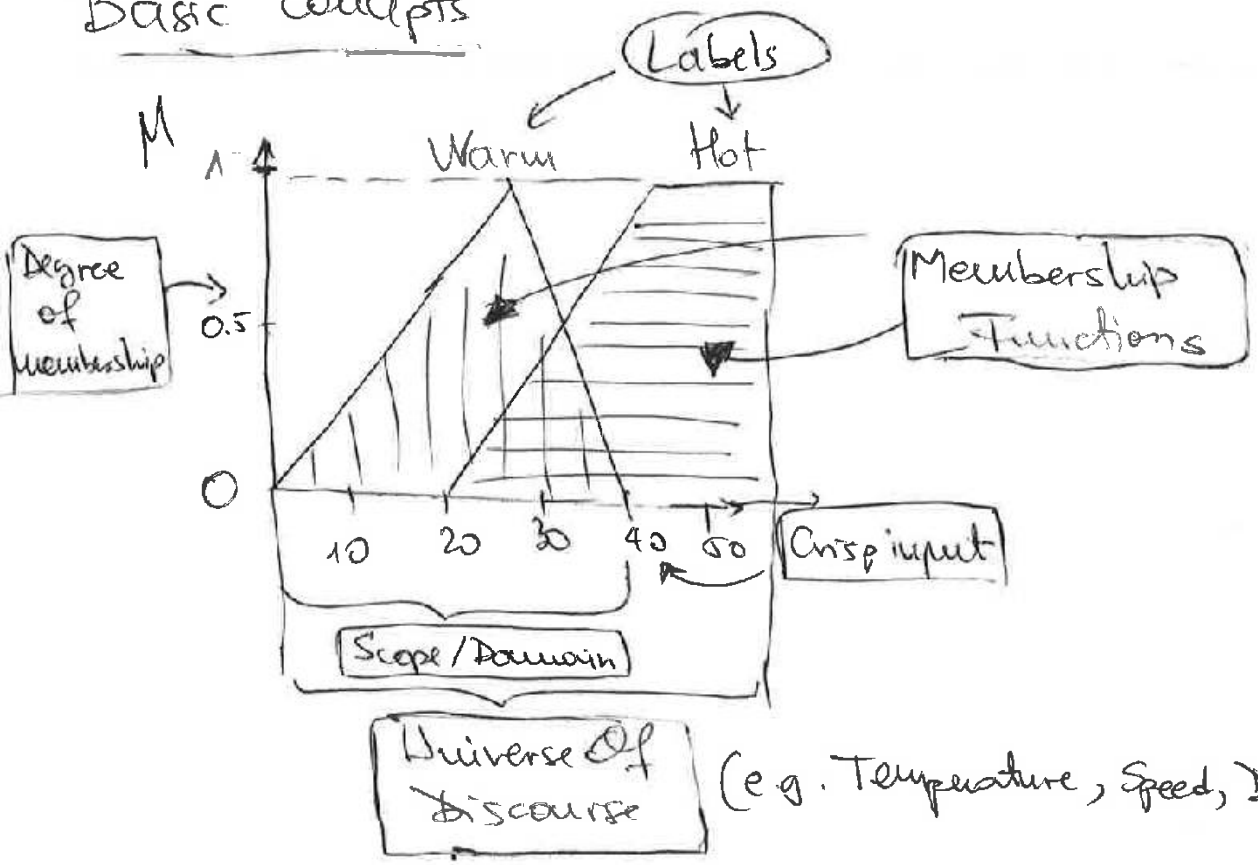


Fuzzification:
- fuzzy sets & logic
- linguistic variables

Rule evaluation:
- fuzzy set operations
- fuzzy relations & inference
- fuzzy composition & reasoning

Defuzzification:
- extract crisp information from fuzzy sets

Basic concepts



(e.g. Temperature, Speed, Density etc.)

Definitions:

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

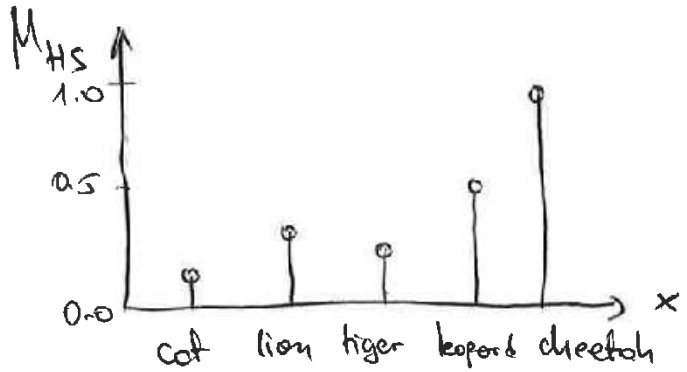
$$A = \{ (x, \mu_A(x)) \mid x \in X \}$$

E.g. Universe of discourse: cat families species

$$X = \{ \text{"cat"}, \text{"lion"}, \text{"tiger"}, \text{"leopard"}, \text{"cheetah"} \}$$

Fuzzy set HS for animals with high speed

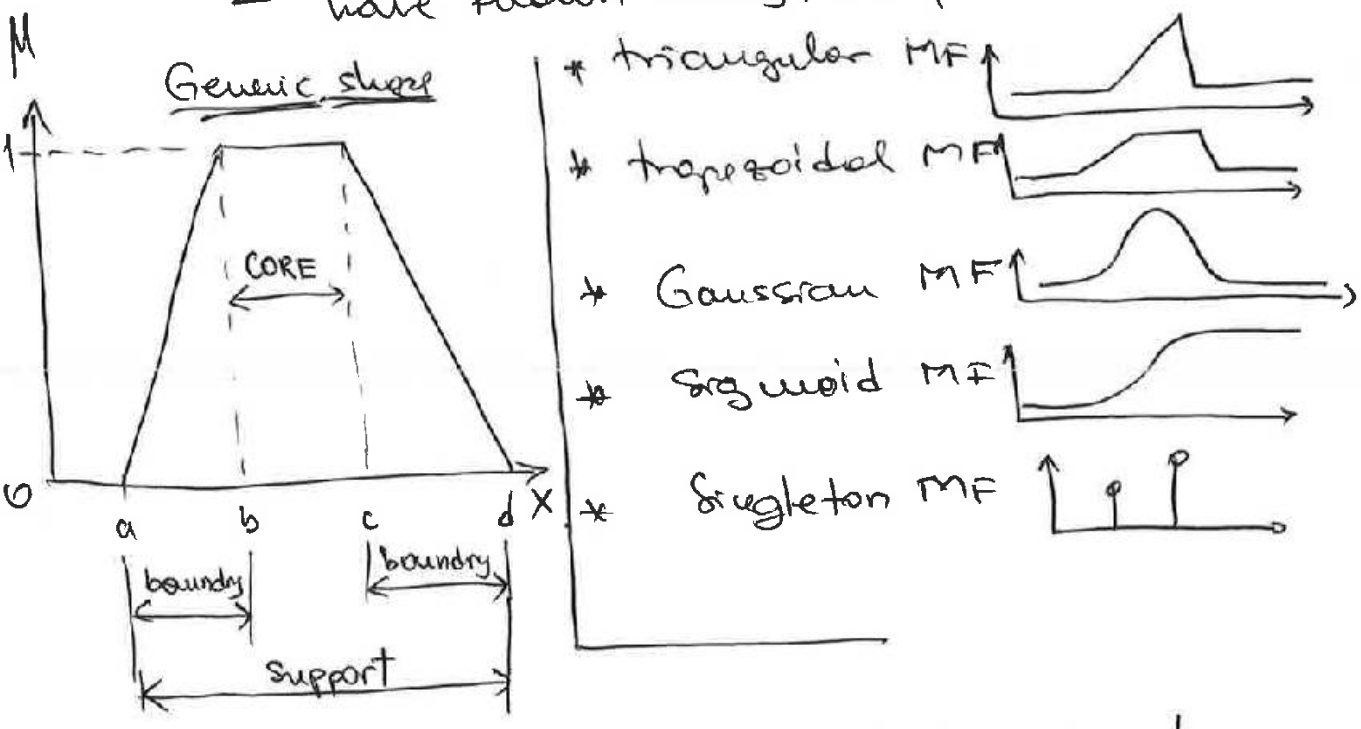
$$HS = \{ (x, \mu_{HS}(x)) \mid x \in X \} = \{ (cat, 0.1), (lion, 0.3), (tiger, 0.2), (leopard, 0.5), (cheetah, 0.9) \}$$



Internals of fuzzy logic and fuzzy inference systems.

Membership functions

- mathematical functions that give numerical meaning to a fuzzy set.
- map crisp inputs from a domain to membership degrees in $[0,1]$.
- have known analytical formulations:



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

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

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

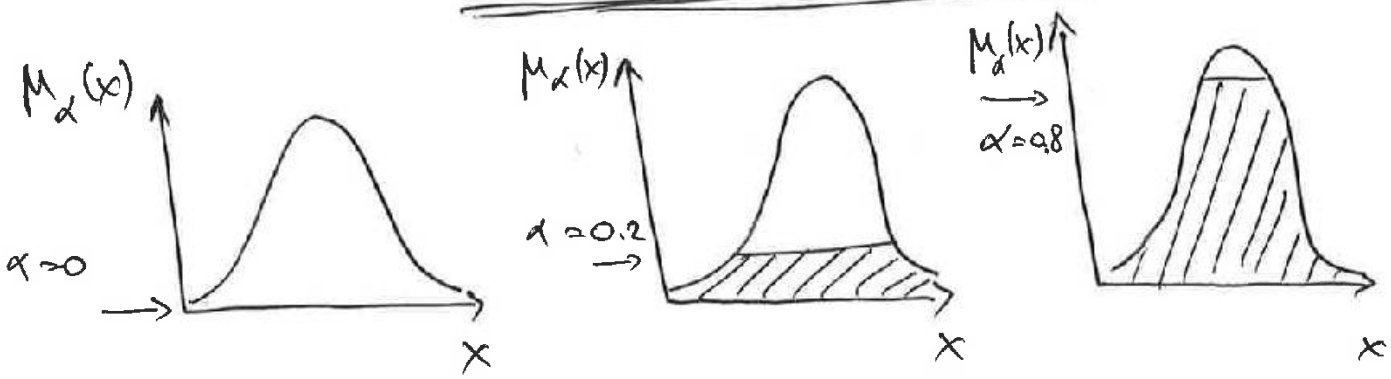
Singleton : fuzzy set with 1 element $\text{supp}(A) = \text{core}(A) = \{x_0\}$

Operations on membership functions

Alpha cut set A_α is a **crisp set** of a **fuzzy set** where

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

and if $A_\alpha = \{x \in X \mid \mu_A(x) > \alpha\}$ we have a **strong alpha cut set**.

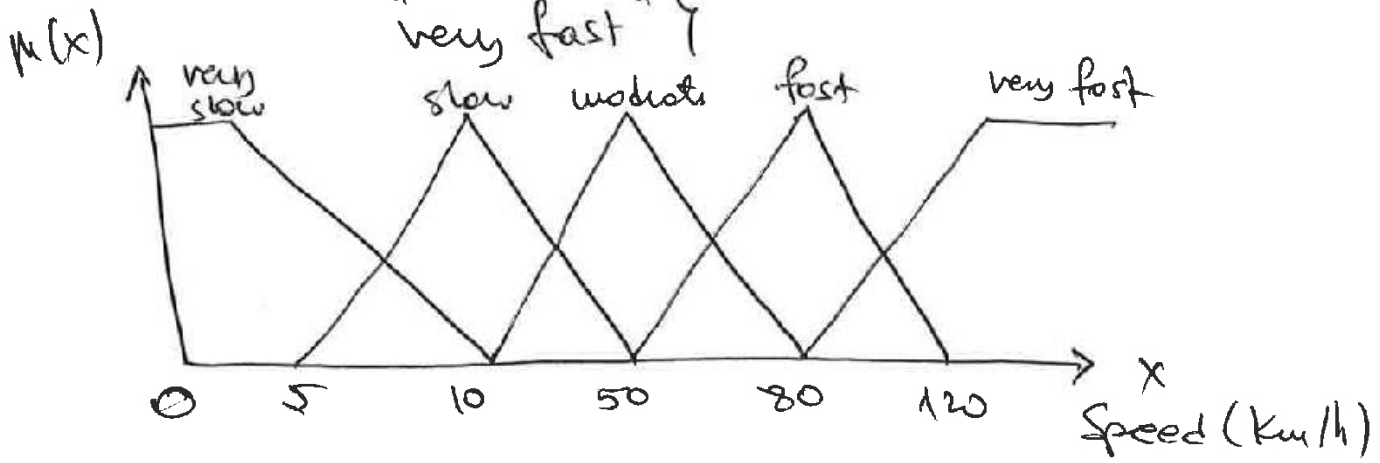


Linguistic variables

* identifiers for membership functions.
* define partitions of the universe of discourse.
* depend on the application.

E.g. a variable "speed" for a control system

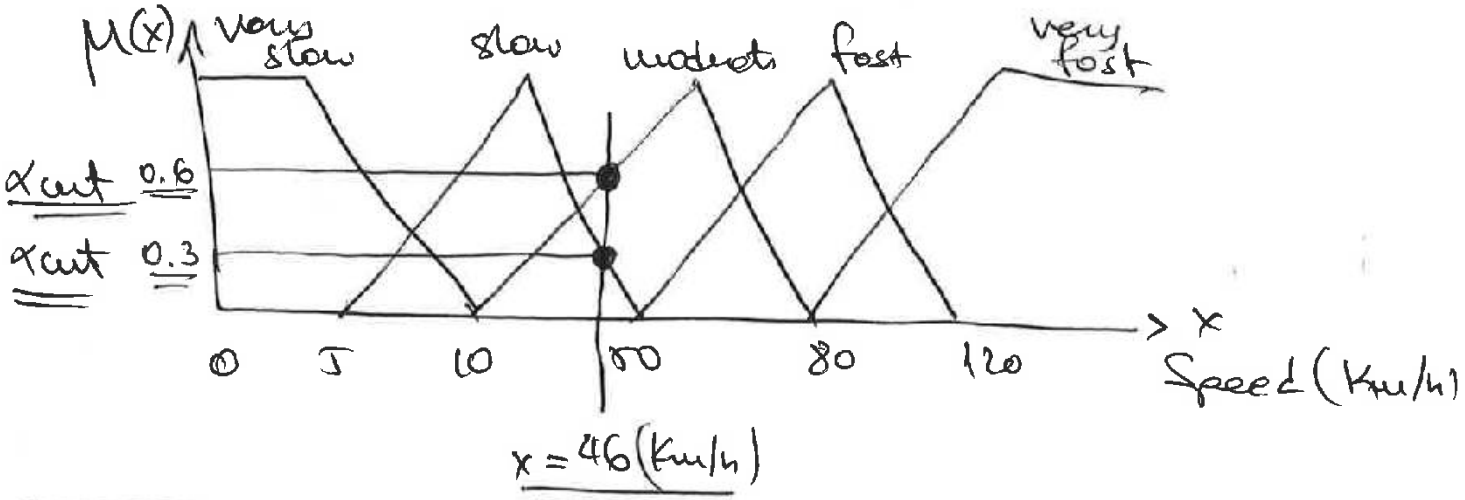
$S = \{$ "very slow", "slow", "moderate", "fast", "very fast" $\}$



- Linguistic variables ⟹ computing with words

- assign a membership degree for a crisp value for each label of a linguistic variable.

- E.g. Speed partitioning for an autonomous car.



Fuzzification: finds degrees of membership for crisp values:

$\mu_{\text{very slow}}(x=46) = 0.0$ (no intersection with very slow MF)

$\mu_{\text{slow}}(x=46) = 0.3$

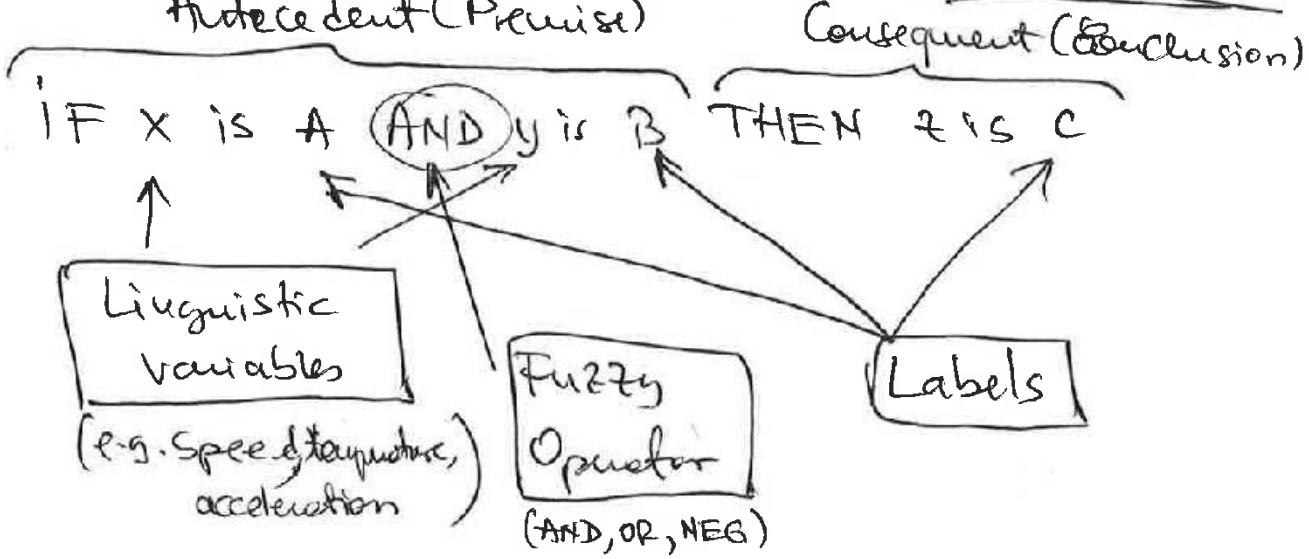
$\mu_{\text{medium}}(x=46) = 0.6$

$\mu_{\text{fast}}(x=46) = 0.0$ (no intersection with fast MF)

$\mu_{\text{very fast}}(x=46) = 0.0$ (no intersection with very fast MF)

Rule evaluation

- rules in fuzzy logic are using first order logic:
- IF-THEN clauses.



- if there are several input variables $x_1 \dots x_n$ with several connections (AND, OR) the IF-THEN clauses can be written as:

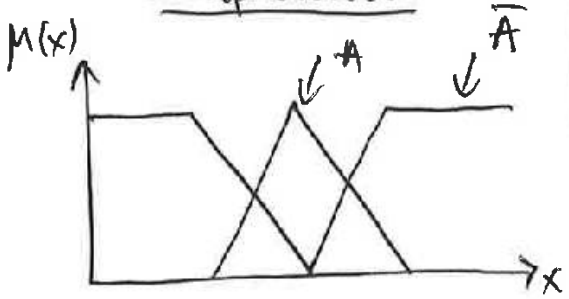
IF x_1 is A_{r_1} AND x_2 is A_{r_2} OR x_3 is A_{r_3} --- AND x_n is A_{r_n}

THEN y is B_r

where $r = 1 \dots n$ is the rule number

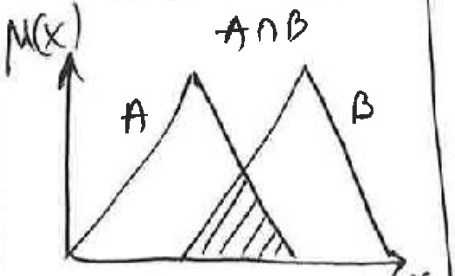
Basic operators

Complement



$$\mu_{\bar{A}}(x) = 1 - \mu_A(x)$$

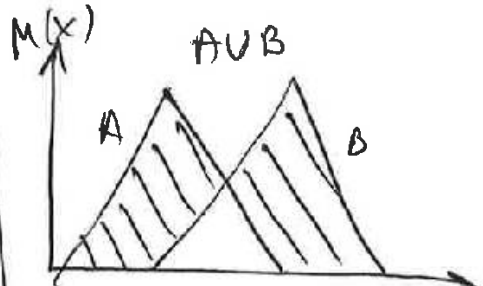
Intersection



MIN: $\min\{\mu_A(x), \mu_B(x)\}$

PROD: $\mu_A(x)\mu_B(x)$

Union



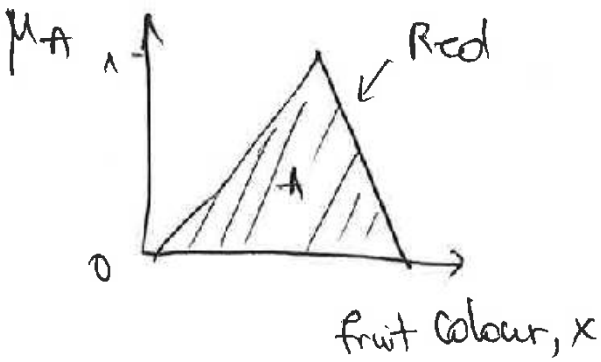
MAX: $\max\{\mu_A(x), \mu_B(x)\}$

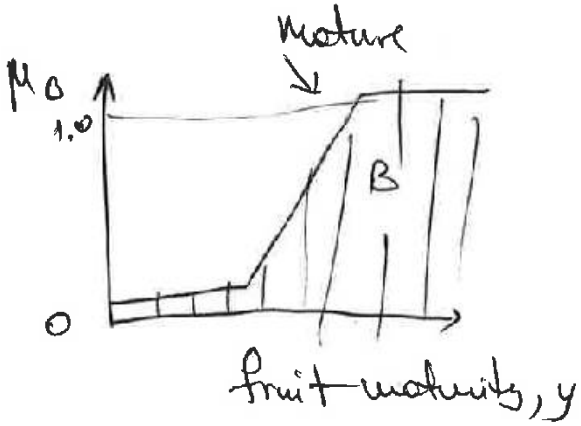
ASUM: $\mu_A(x) + \mu_B(x) - \mu_A(x)\mu_B(x)$

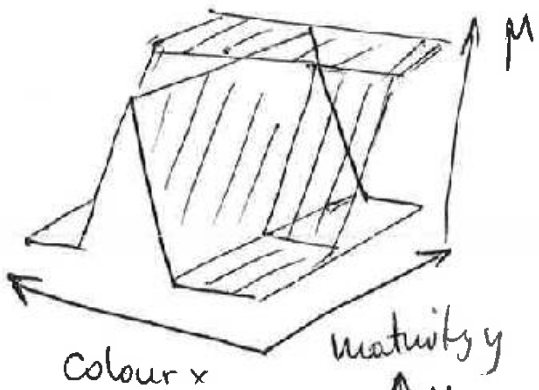
Fuzzy Implication

* used to model dependencies, correlations or connections among variables.
* generalization of a fuzzy set from 2D to 3D.
* describes the degree of association among variables.

E.g. relation between color of a fruit X and the grade of maturity Y



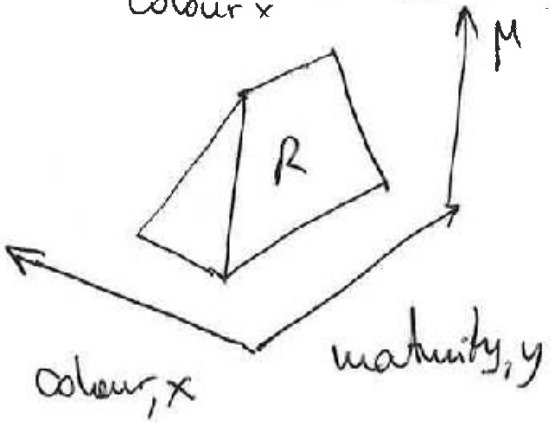




$R: (X = A) \rightarrow (y = B)$

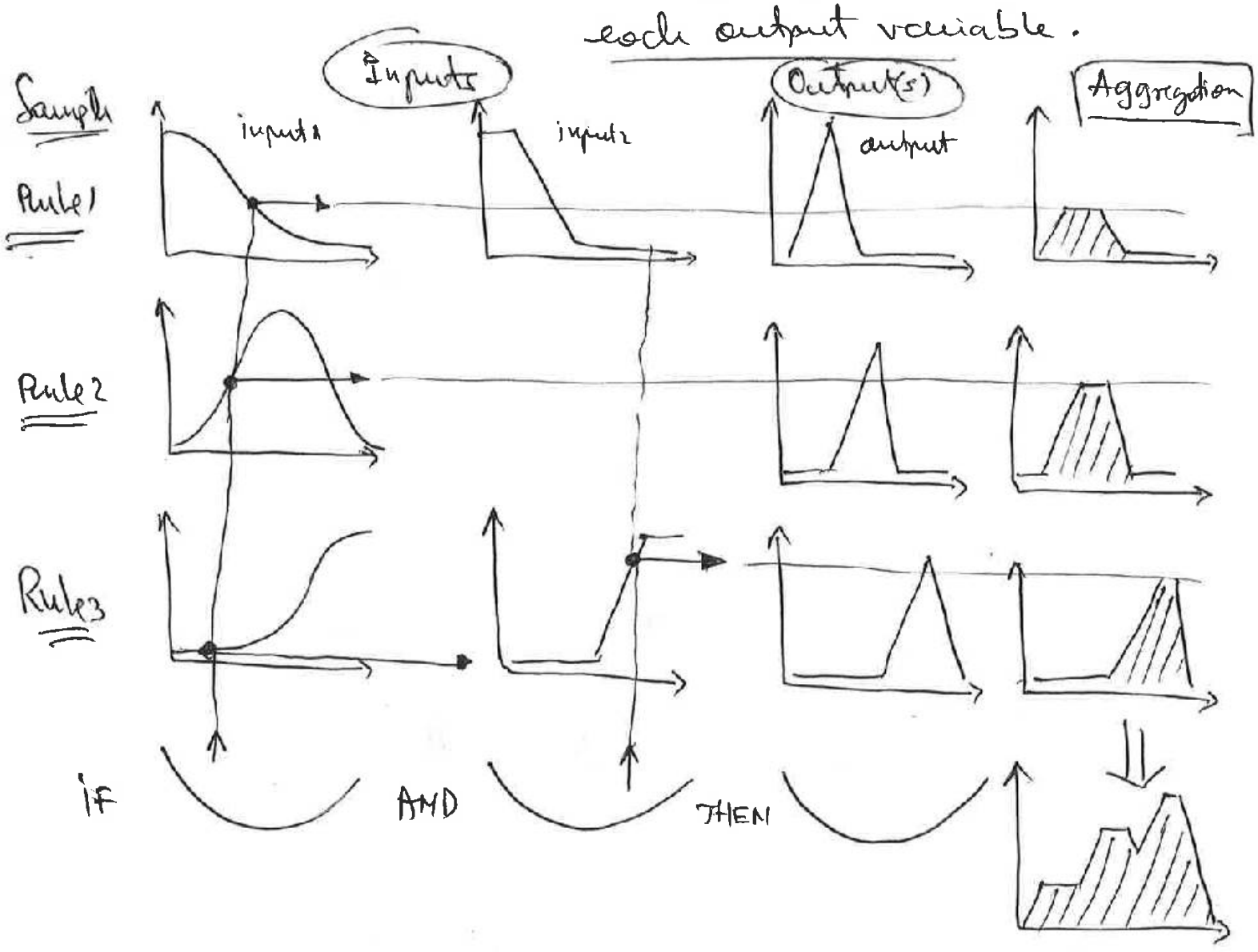
IF x is A THEN y is B

"if the colour is red then the fruit is mature."

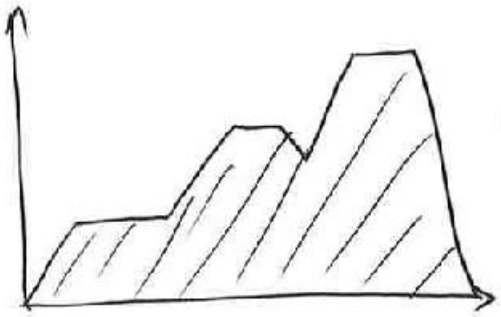


Fuzzy aggregation

- decisions are taken based on testing all the rules.
- rules must be combined for inference.
  - → outputs are combined in a single fuzzy set.
  - → once for each output variable.
  - → the output is a fuzzy set for each output variable.



Defuzzification

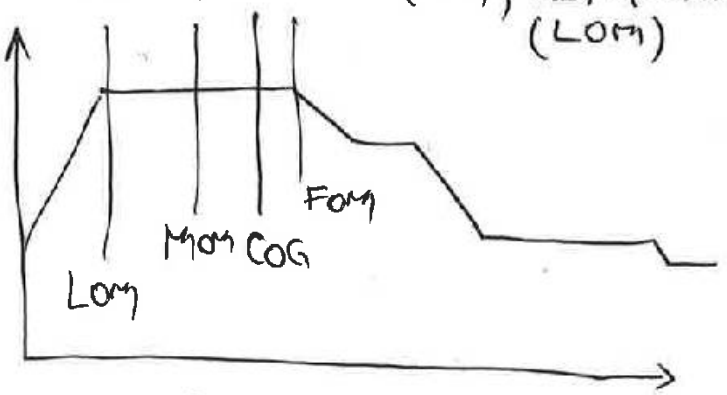


* after inference the result is a fuzzy value;
* the result needs to be mapped to crisp;
* done using the MF of the output variable,
* various mechanisms:

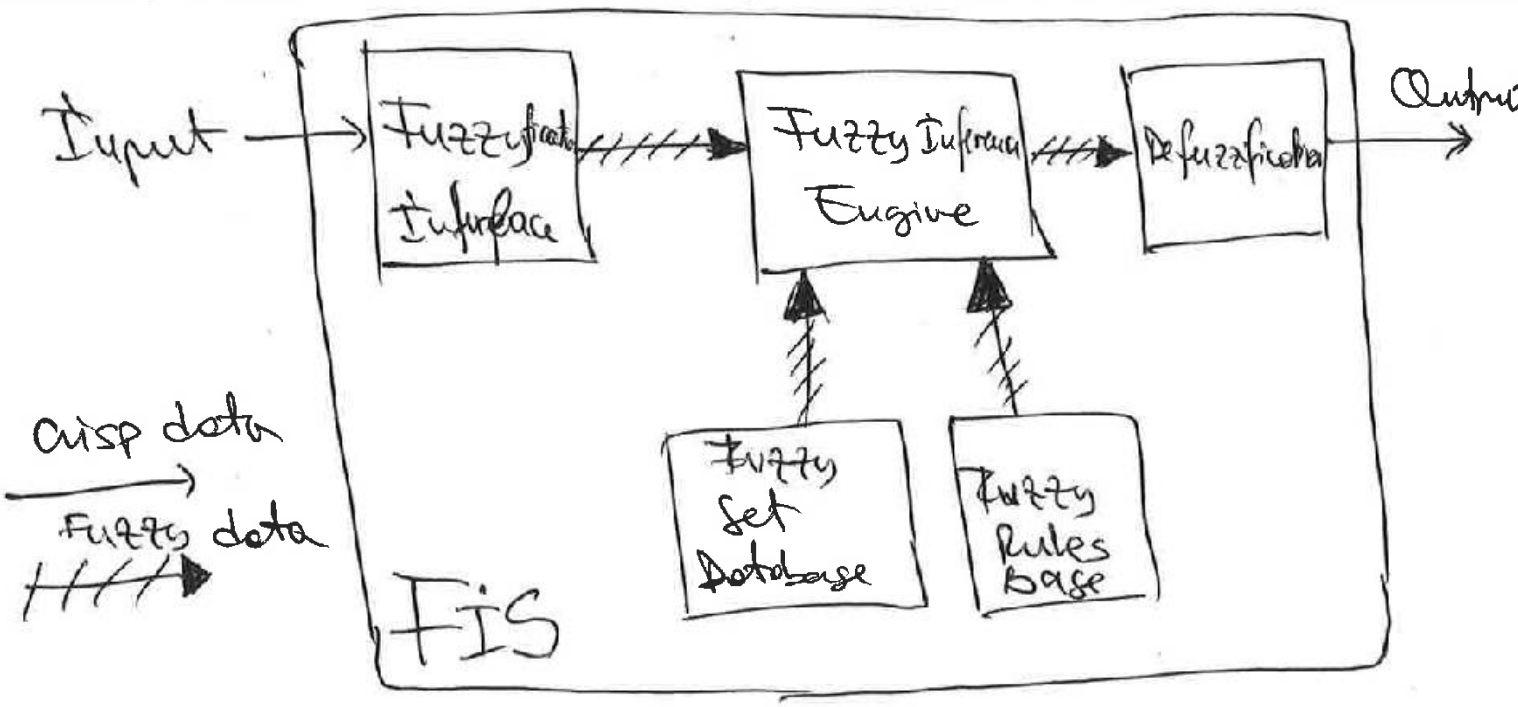
$$COG = \frac{\int_{x_{min}}^{x_{max}} x \cdot \mu_A(x)\, dx}{\int_{x_{min}}^{x_{max}} \mu_A(x)\, dx}$$

center of gravity (COG), center of (COM) maxima, mean of maxima (MOM), first of maxima (FOM), last of maxima (LOM)

Example methods:



From fuzzy logic to fuzzy inference systems



Worked example of FIS

Problem: Evaluate the risk of a software engineering project by refering only 2 criteria : the project funding (PF) and the project staffing (PS).

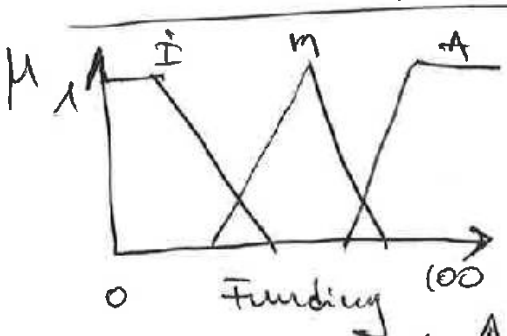
Step 1: Identify input/output variables

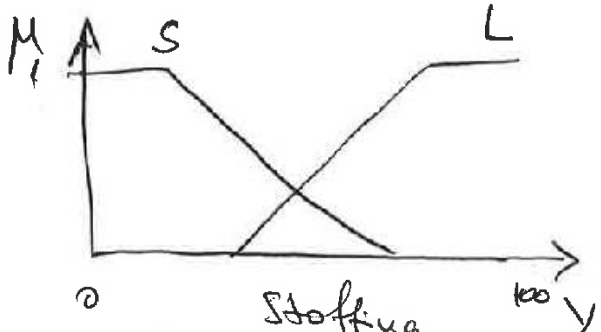
2 inputs: project funding (%) (PF)
project staffing (%) (PS)
1 output: project risk (%) (PR)

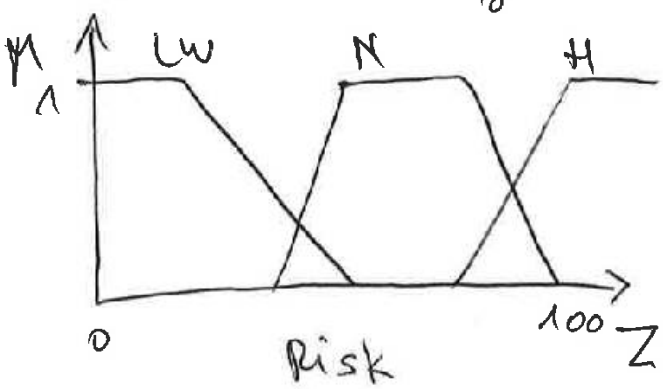
Universe of discourse → fuzzy sets

$X = \{$ inadequate (I), marginal (M), adequate (A) $\}$ for PF
$Y = \{$ small (S), large (L) $\}$ for PS
$Z = \{$ low (LW), normal (N), high (H) $\}$ for PR

Membership functions

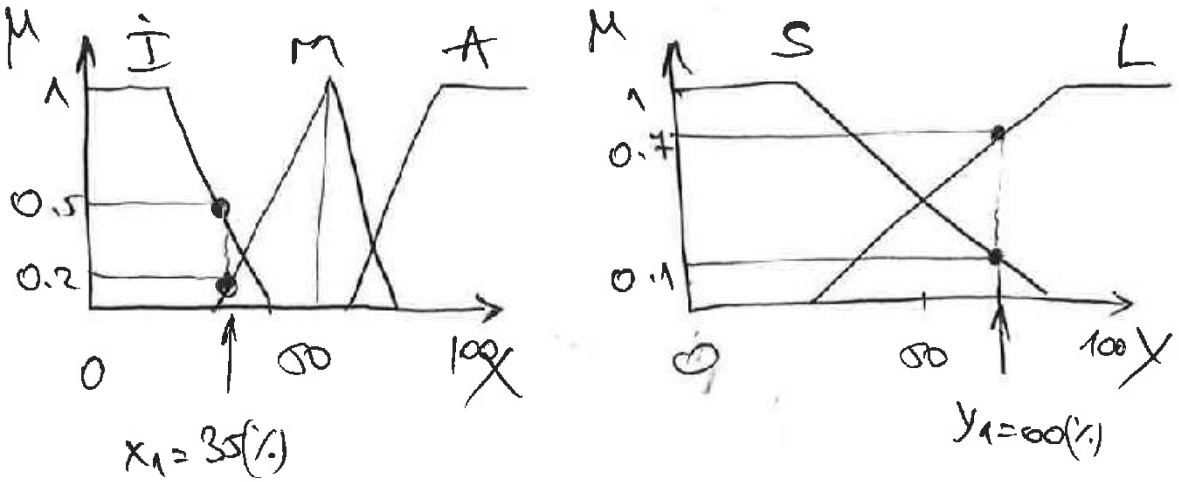






Step 2: Fuzzification

Consider $x_1 = 35\%$ PF and $y_1 = 60\%$ PS



$x_1 = 35(\%)$ $\qquad$ $y_1 = 60(\%)$

$\mu_I(x_1=35) = 0.5$ $\qquad$ $\mu_S(y_1=60) = 0.1$

$\mu_M(x_1=35) = 0.2$ $\qquad$ $\mu_L(y_1=60) = 0.7$

$\mu_A(x_1=35) = 0.0$

Step 3: Rule design (here for example only 3)

Rule 1: IF PF is A (OR) PS is S THEN PR is LW.

Rule 2: IF PF is M (AND) PS is L THEN PR is M.

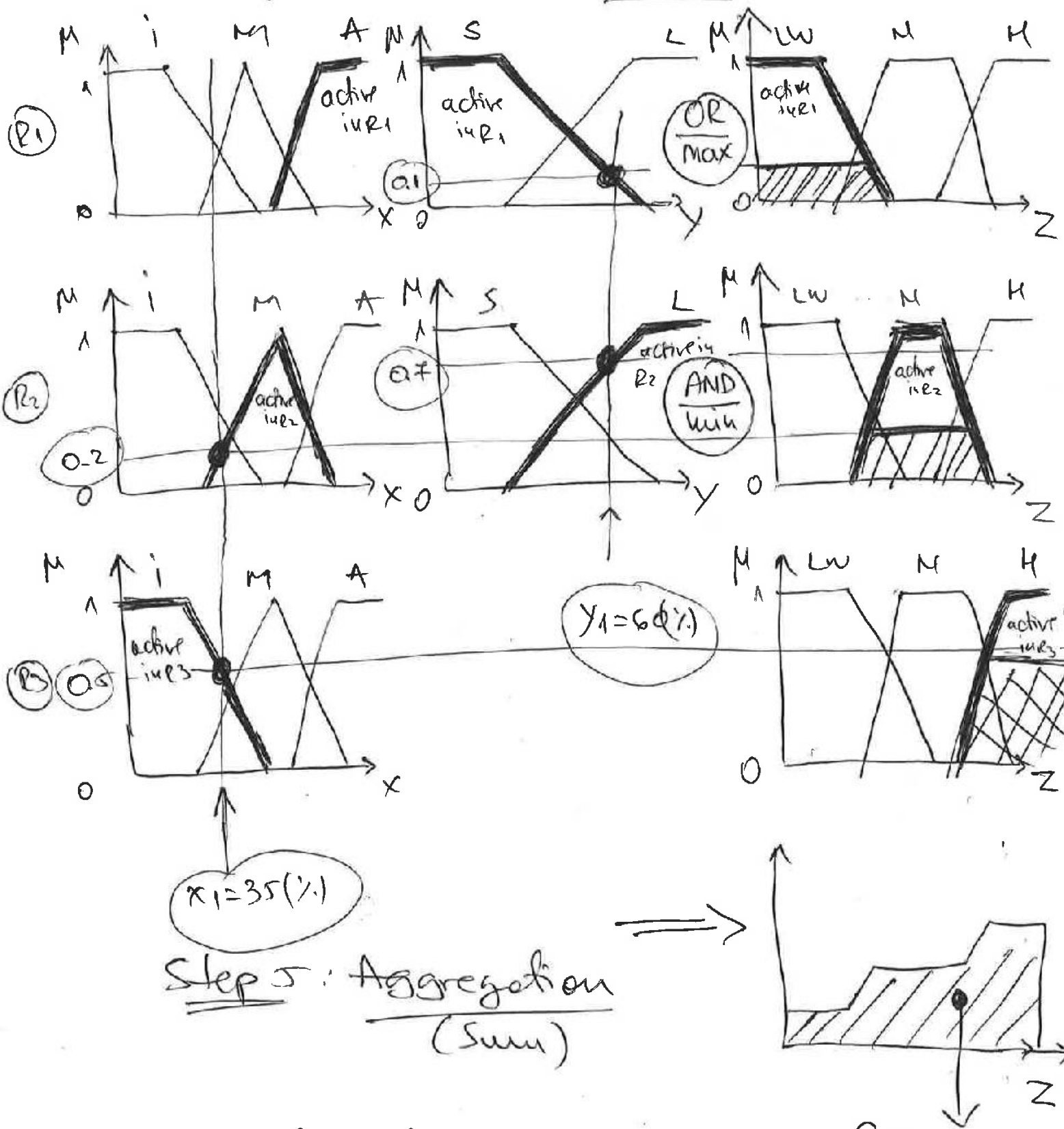
Rule 3: IF PF is I THEN PR is H.

Observations:

* One can generate many rules to make sure one (covers) the entire space of solutions.

* some rules are redundant, through logical conditions they mutually include or exclude.

Step 4 : Rule evaluation (inference)



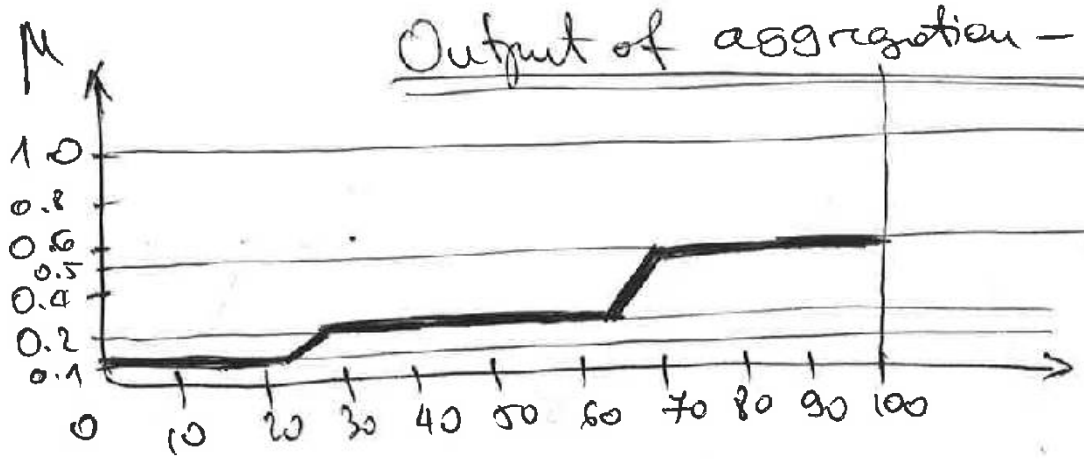
Step 5 : Aggregation (Sum)

Step 6 : Defuzzification (using COG)

$$COG = \frac{\int_a^b x \mu_A(x)\,dx}{\int_a^b \mu_A(x)\,dx} \quad \text{Continuous}$$

$$\Rightarrow COG = \frac{\sum_{x=a}^{b} x \mu_A(x)}{\sum_{x=a}^{b} \mu_A(x)} \quad \text{discrete}$$

Output of aggregation – a fuzzy set



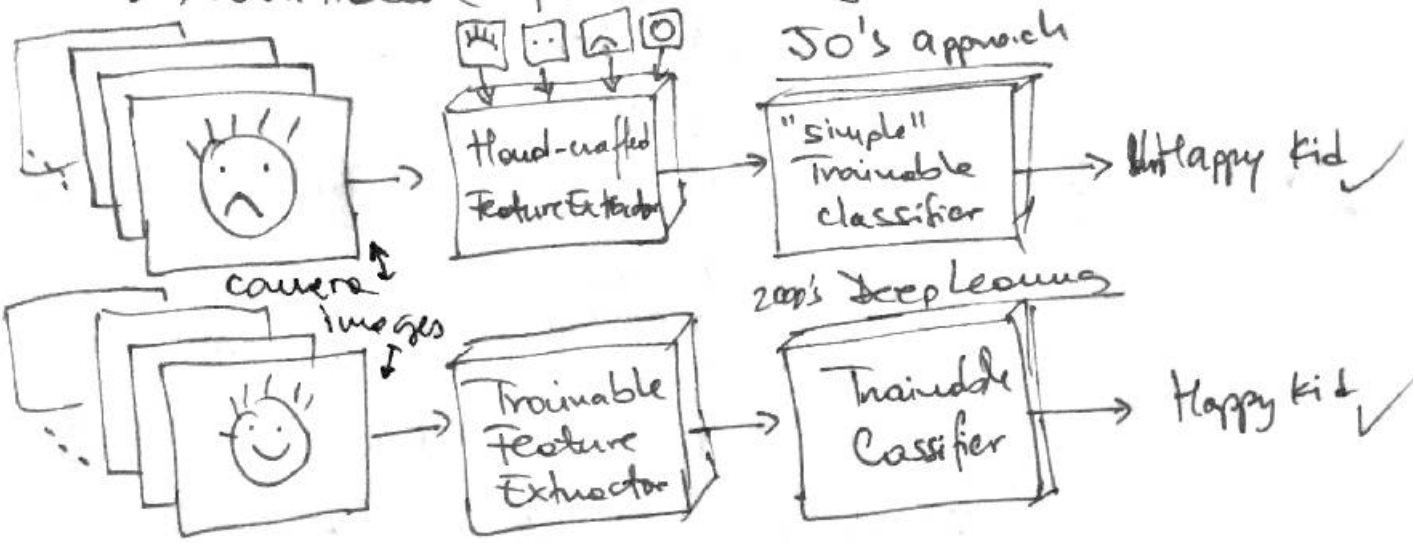
$$COG = \frac{(0+10+20)\times 0.1 + (30+40+50+60)\times 0.2 + (70+80+50+100)\times 0.5}{(0.1+0.1+0.1)+(0.2+0.2+0.2+0.2)+(0.5+0.5+0.5+0.5)}$$

$$COG = 67.9\ (\%)$$

AIML Class: Deep (Neural) Learning

Deep Learning = Learning Representations/features

- Traditional pattern recognition



- Deep Learning learns hierarchical representations
  - Hierarchy with increasing level of abstraction
  - each stage is a trainable feature transformation
  - Inspired by the hierarchical processing in the visual cortex with multiple pathways and stages (HMAX model)
  - low-level features are shared among categories
  - high-level features are more global and more invariant

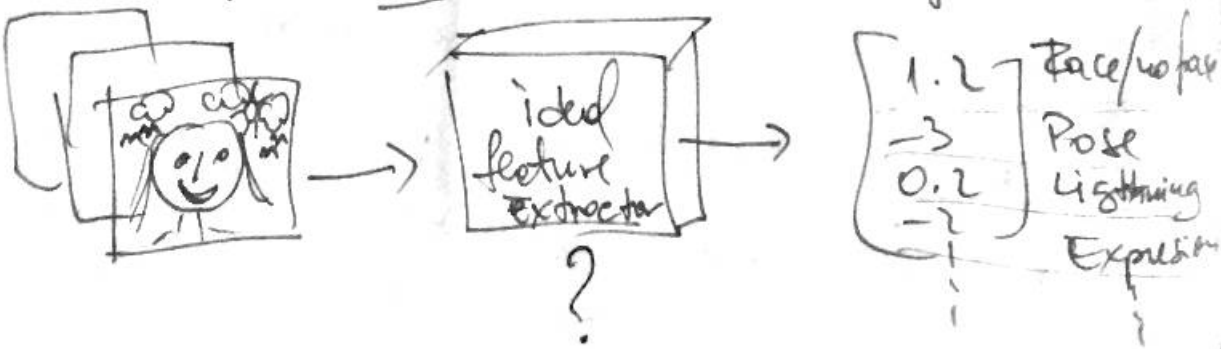
- 3 Types of training:
  - (A). purely supervised: – init parameters randomly
    – train in supervised mode using backpropagation
    e.g. speech & image recognition

(B) • Unsupervised, layerwise and a supervised classifier on top
- train each layer unsupervisedly
- train a classifier on top, keeping all other layers fixed
↦ good when training set with labels is limited

(C) • Unsupervised, layerwise and a global supervised fine-tuning
- train each layer unsupervisedly
- add a classifier layer and retrain the whole network supervisedly
↦ good when labelset is poor

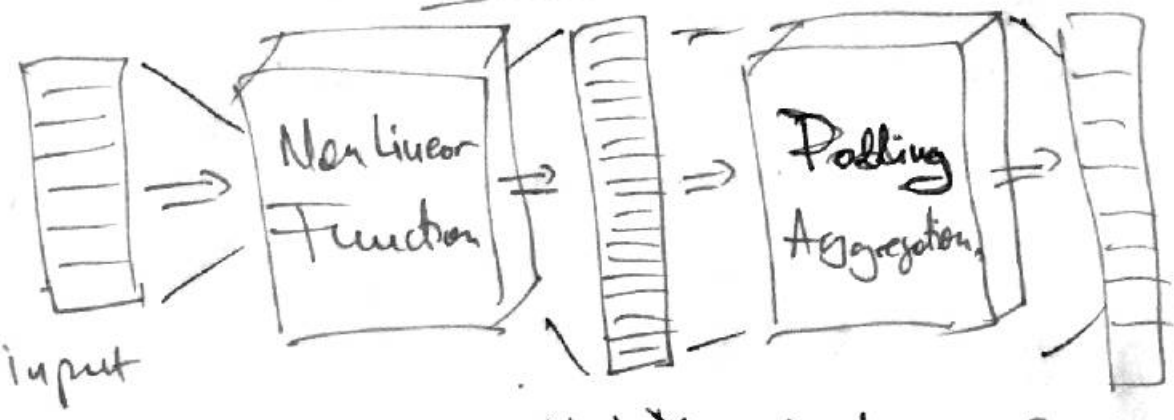
The manifold hypothesis

- natural data lives in a low-dimensional (nonlinear) manifold
- there is (no) general method to learn functions that turn images (for example) into a low-dimensional representation

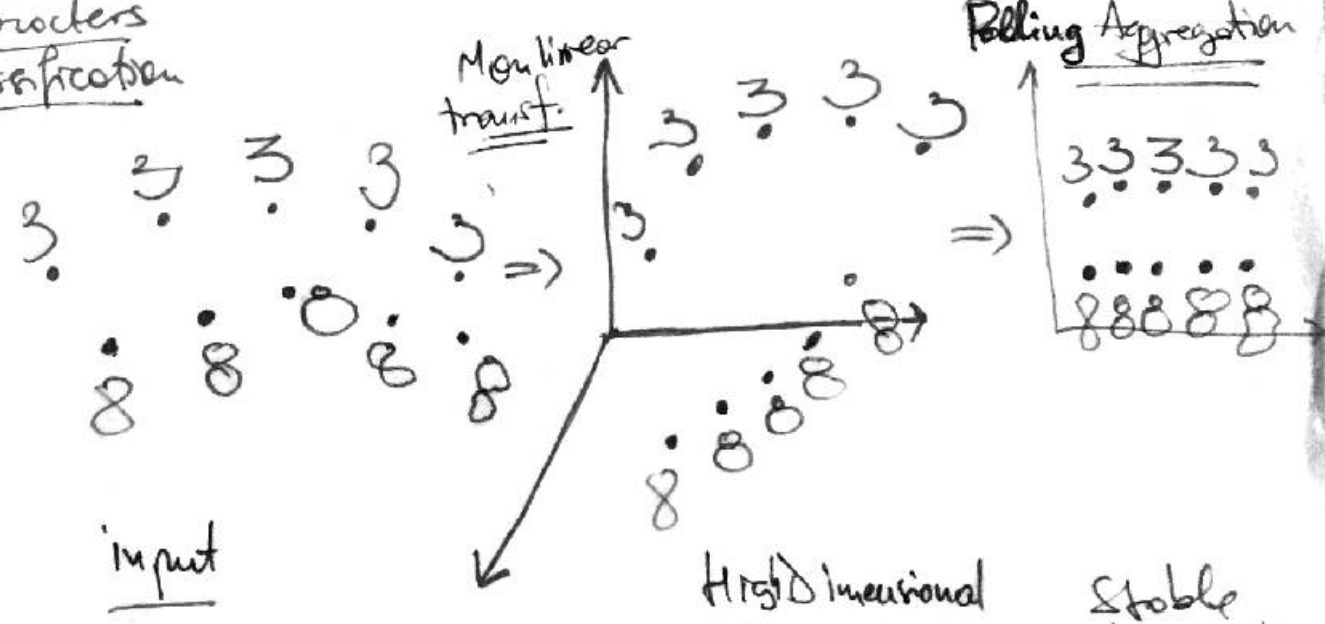


The inner workings of a deep learning network

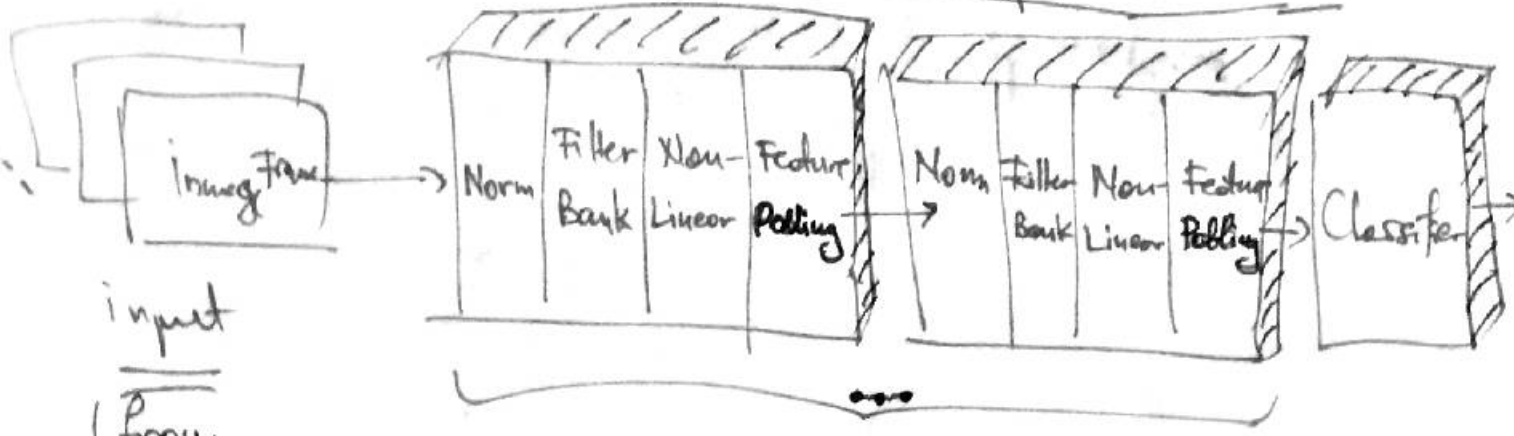
- **embeds** the input into a high(er) dimensional space
  - → in this new space features that were not separable might became separable
  - → bring together things that are semantically similar



Eg.: hand-written characters classification



Generic architecture of a Deep Network



input (from Camera) for example

Stack multiple such layers

Norm : - normalization : filtering, average removal ;

Filter Bank : - dimension expansion ;

Non Linearity : - activation function ;
- WTA (Winner-Take-All).

Pooling : aggregation over space or feature type.
(selection)

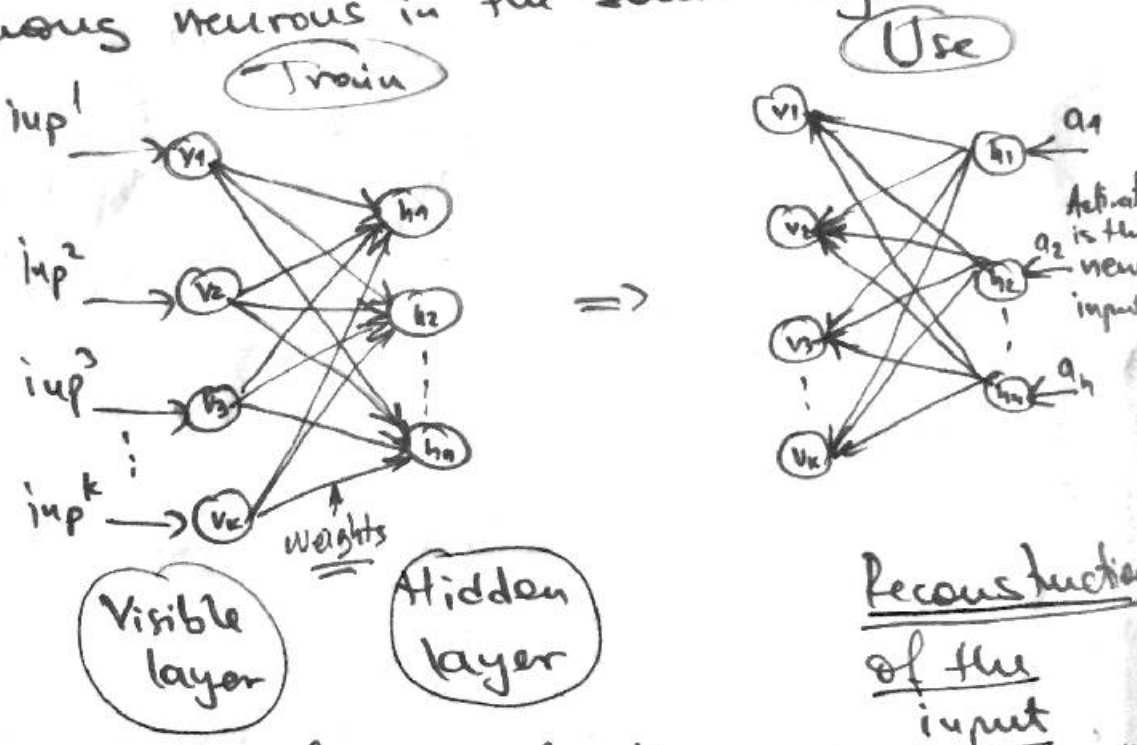
Deep Networks Building Blocks

Typically: Feed forward multi-layer neural nets → we discussed in previous lectures

Now we look at:
- Restricted Boltzmann Machines (RBMs)
- Autoencoders

• RBMs (Restricted Boltzmann Machines)

- model probability distributions in data
- used for feature extraction
- used for dimensionality reduction
- the "restricted" comes from the fact that the network doesn't have connections among neurons in the same layer



- Use gradients in a learning algorithm called contrastive divergence.
- Unsupervised learning.
- Usually size of input layer > size of hidden layer to create a bottleneck ⇒ a compact representation of the input space by extracting its structure.

(e.g. Handwritten Recognition MNIST dataset)

- RBMs are energy based models, where the joint probability of visible (v) and hidden units (h) is proportional with the energy between them $E(v,h)$:

$$P(v,h) \propto e^{-E(v,h)}$$

- Manipulating the energy $E(v,h)$ we can generate the probability $P(v,h)$.

→ we minimize the energy which maximizes the probability

→ this is done by learning the weight matrix W between input (visible) units and the hidden units (h):

$$E(v,h) = -v \cdot W \cdot h$$

Unsupervised learning allows to learn distributions (Remember SOM?!)

→ Contrastive divergence algorithm is similar to gradient descent and assumes a weight update:

$$W \leftarrow \lambda \cdot W \cdot cd(v_t)$$

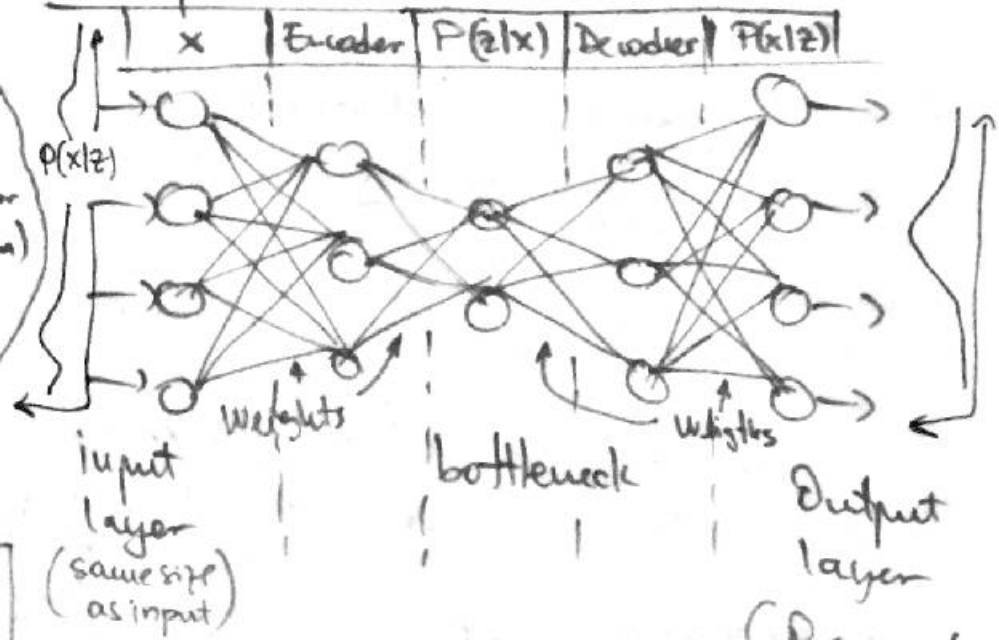
where $cd(v_t)$ is the contrastive divergence of instance v_t and λ = learning rate

Observations:

- Iterative algorithm
- $cd(v_t)$ minimizes the difference between the real distribution of the data and the guess of the net.

⊙ Autoencoders

- a variant of feed forward neural nets.
- have an extra bias for calculating error of reconstructing the input.
- used to learn compressed representations of the data
  ⇒ their output is an efficient reconstruction of the input
- difference from MLP is that autoencoders have an output layer with same size as input layer
- Sample network:



Looking In depth:

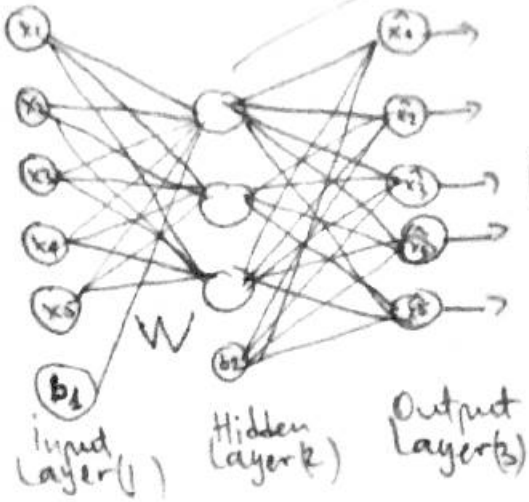
x = data instance
$z^i \sim p(z)$ = prior
$x^i \sim p(x|z)$ = generator (data distribution)

The net learns an estimate of $P(z|x)$ to decode $P(x|z)$.

! Learns the distribution of the data

- Such a structure is useful in detecting anomalies (i.e. new patterns) after being trained on good patterns.

Autoencoders - addons



→ this layer can be "compressive" or "sparse"

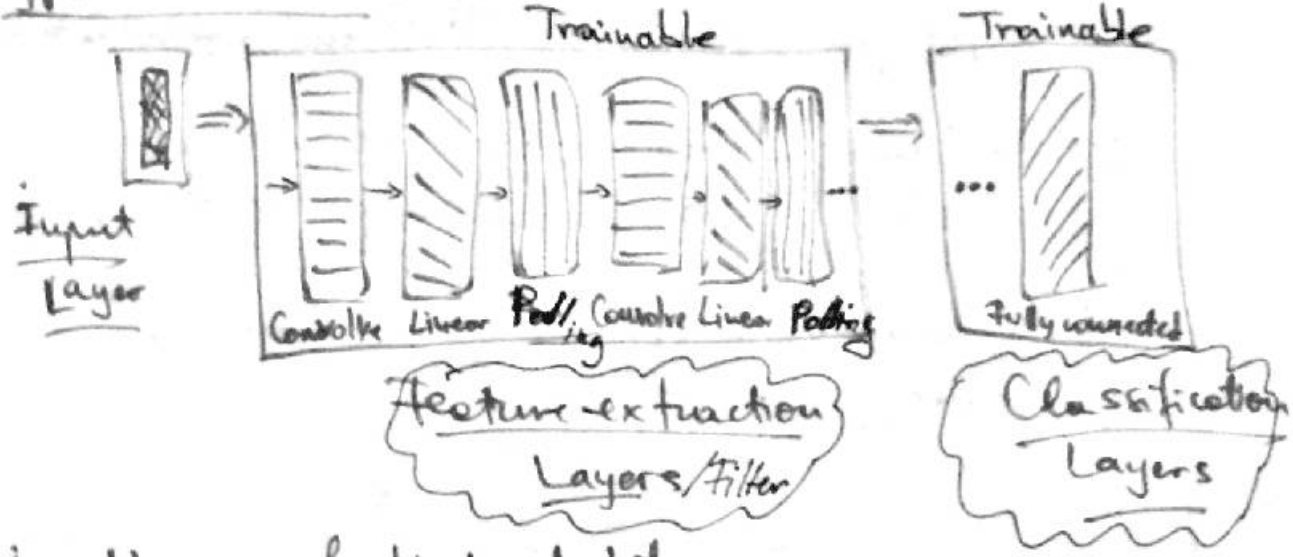
$h_{W,b}(x) \approx x$ approximation of the identity function

- $\{x_1, x_2, x_3, x_4, x_5\}$ - unlabeled set of training examples, there is no y_i
- the Autoencoder applies back propagation setting the targets to be equal to the inputs $(y_i = x_i)$
- it tries to learn unsupervisedly a function $h_{W,b}(x) \approx x$ which is an approximation to the identity function so that $\hat{x}$ is similar to x => but isn't this trivial?!

Yes, but => due to the network structure ("bottleneck" in hidden layer) the network learns a "compressed" representation of the input: e.g. $10 \times 10_{px}$ image as input, so $n = 100$ input neurons, consider $h = 50$ hidden neurons and 100 output neurons => if there are correlations in the input the auto encoder(s) extracts that (similar to PCA)

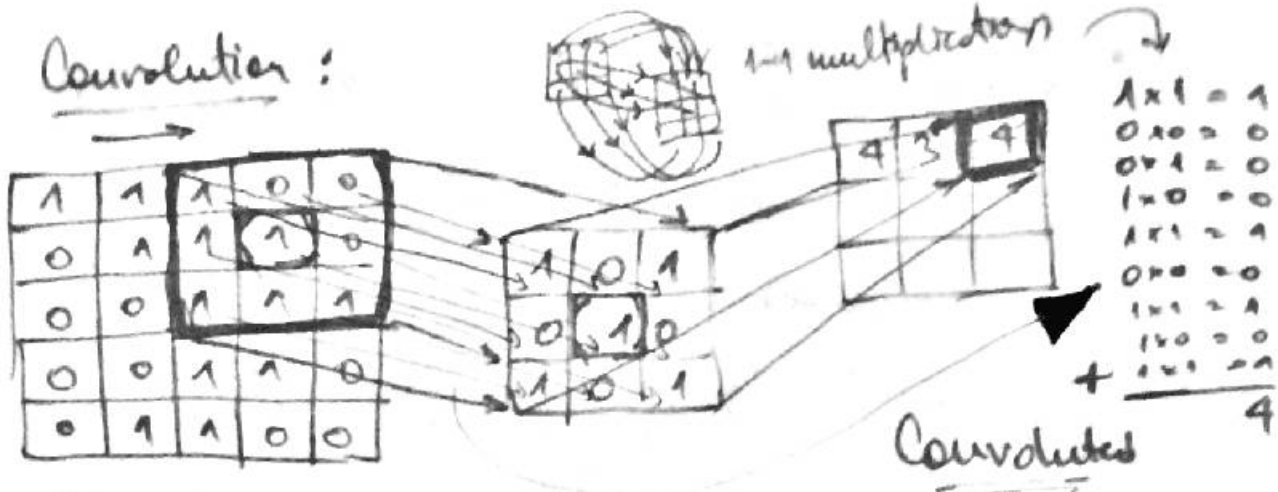
- Convolutional Neural Networks (CNN)
  - learn high-level features using convolutions
  - used for object recognition & image classification
  - build position & rotation invariant features from the training data (it learns the structure).
    ↳ So, ideal input are images and sounds with repeating patterns
  - supervised learning

Typical Structure:



- input layers: feed in input data
- Convolution layers: - core of the CNN
  - 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
  - feature detector layer
  - interpreted as a filter

Convolution :



Convoluted feature

- Input data (e.g. image B&W pixels)

Principle of convolution

Center element of the kernel (eg 1) is placed over the source pixel (eg 1). The source pixel is then replaced with a weighted sum of itself & nearby pixels. (Kernel x pixel) = dot product)

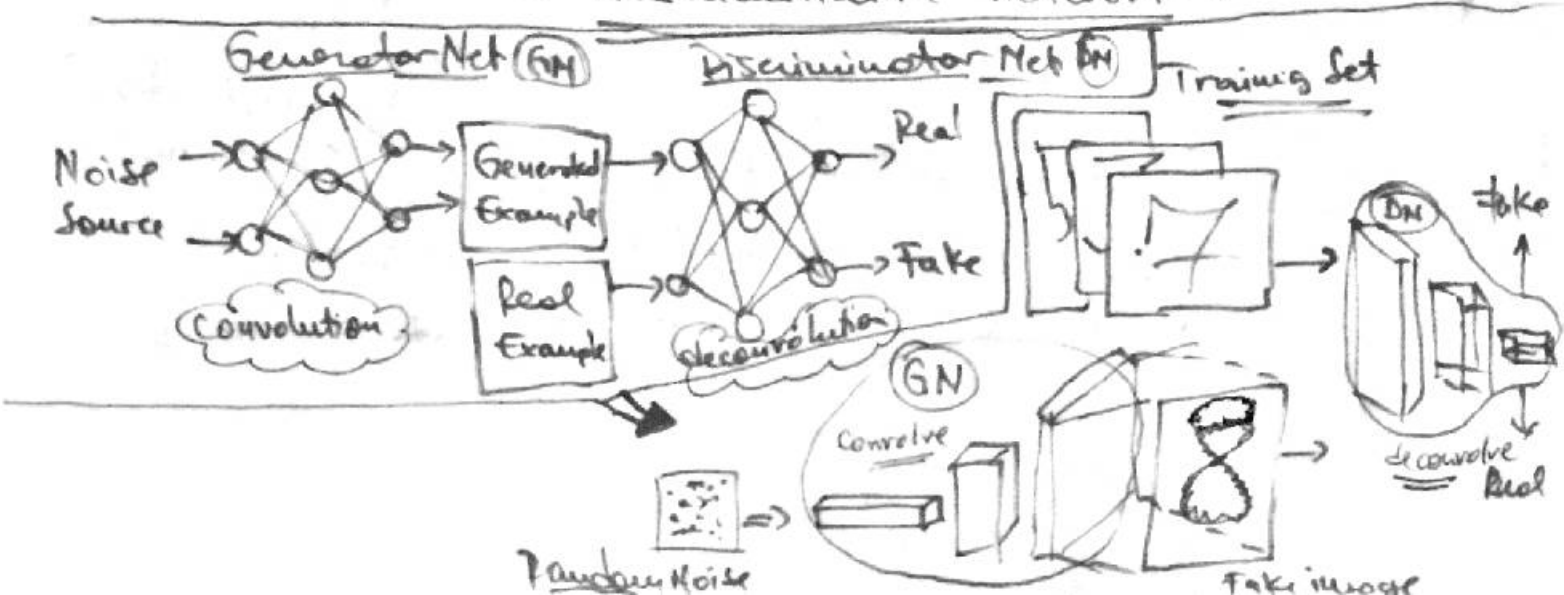
- pooling layers : reduce data representation progressively and control overfitting, can implement different operations (mean, max ...).
- fully connected layers : Compute class scores.

Major Deep Networks Architectures

- **Deep Belief Network (DBN)**
  - composed of layers of RBMs for the pretrain phase and feed-forward net for fine tuning phase
  - can extract high-level features of raw data

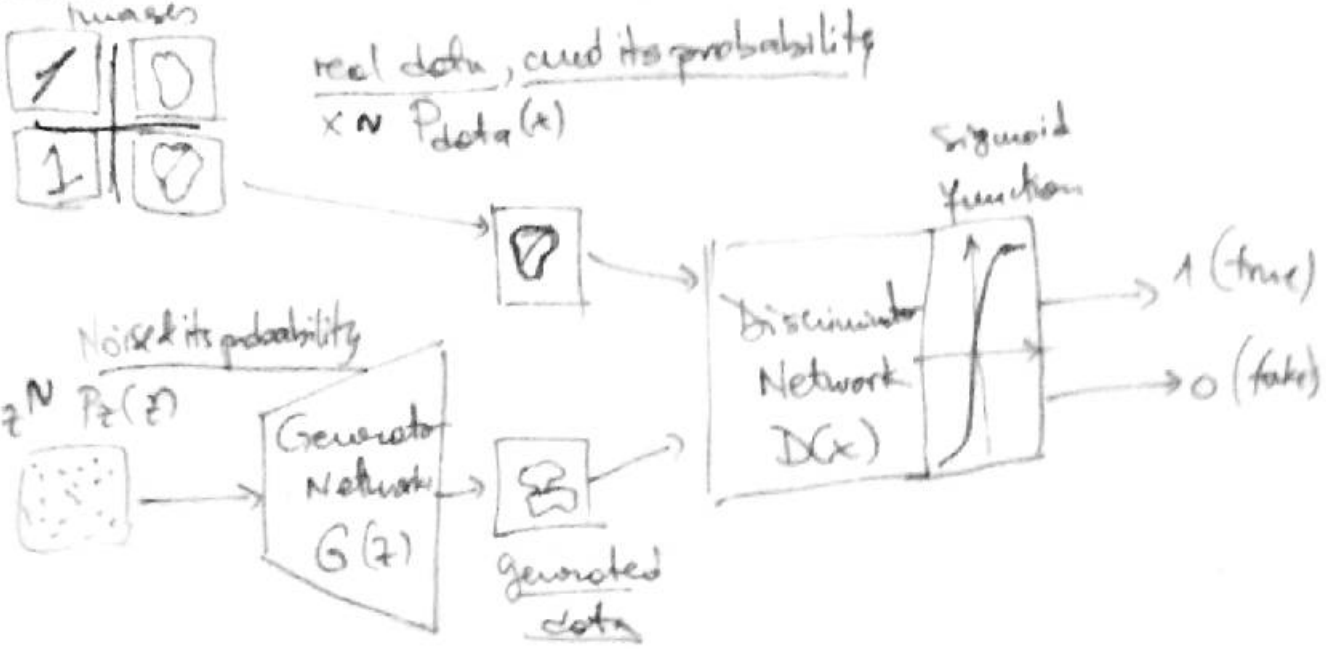
Operation:
- → the RBMs learn the features unsupervisedly & reconstructs the input ⇒ feed into
- → the feed-forward part is trained with backprop to make predictions on the labels (softmax).

- **Generative Adversarial Networks (GAN)**
  - used to synthesize (generate) new images out of other training example images
  - unsupervised learning to train two models in parallel
  - efficient representation of the input data
  - composed of a generative network and a discriminative network:



Generative Adversarial Networks – add-ons

e.g.: handwritten characters images



Training a GAN is a "fight" between generator & discriminator, or in math:

can be also seen as cross entropy

$$\min_G \max_D V(D,G)$$

where (game theory ? player minimax criteria)

$$V(D,G) = \underset{x \sim P_{data}(x)}{\mathbb{E}} [\log D(x)] + \underset{z \sim P_z(z)}{\mathbb{E}} [\log(1 - D(G(z)))]$$

entropy that the data from real distribution passes through discriminator (best case) ✓ (D tries to maximize to 1)

entropy that the data from random source passes through discriminator worst case (✗) (D tries to maximize to 0)

- The discriminator (D) tries to maximize V whereas the generator (G) tries to minimize it (i.e. minimum difference between real

Training phases for GAN

Pass 1: Train discriminator & freeze generator
just a forward pass and no backpropagation of information:

D(x) → update

G(x) → just compute output (no update)

Pass 2: Train generator & freeze discriminator

D(x) → just get output (no update)

G(x) → update

More explicit steps:

1. Train D on real data
2. Generate fake inputs for generator G and train D on fake data
3. Train G with the output of D
4. Repeat #1 to #3 for a number of epochs

If you would have a fully functional generator you could duplicate almost anything: generate news, books and novels with unimaginable stories etc. :) It would become real!

Additional math

- The entropy H of a discrete random variable X with possible values $\{x_1, \dots x_n\}$ and probability mass function $P(X)$ is:

$$H(X) = \mathbb{E}[I(X)] = \mathbb{E}[-\ln(P(X))]$$

$\mathbb{E}$ — expectation operator; I — information content of X

$$H(X) \overset{\text{or explicitly}}{=} \sum_{i=1}^{n} P(x_i) I(x_i) = -\sum_{i=1}^{n} P(x_i) \log_b P(x_i)$$

b - base of logarithm

e.g. $\begin{cases} b = 2 & \text{bits (Euler number)} \\ b = e & \text{nats} \\ b = 10 & \text{bans} \end{cases}$

- Conditional entropy

$$H(X|Y) = -\sum_{ij} P(x_i, y_j) \log \frac{P(x_i, y_j)}{P(y_j)}$$

the amount of randomness or uncertainty in the variable X given the event Y.

- Cross entropy

$$H(p, q) = \mathbb{E}[-\log q] = H(p) + D_{KL}(p \| q)$$

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

→ Kullback Leibler divergence

$$D_{KL}(p \| q) = \sum_{i} P(i) \cdot \log \frac{P(i)}{Q(i)}$$

AUROC & AUPRC

Precision

Reality

Prediction	F	T
F	TN	FN
T	FP	TP

What fraction of the predicted true are actually true?

Recall

Reality

Prediction	F	T
F	TN	FN
T	FP	TP

What fraction of the actual true are predicted to be true?

Same thing

True Positive Rate

Reality

Prediction	F	T
F	TN	FN
T	FP	TP

What fraction of the actual true are predicted to be true?

True Negative Rate

Reality

Prediction	F	T
F	TN	FN
T	FP	TP

What fraction of the actually negative are predicted to be negative.

AUPRC

Note that TN is not considered in AUPRC, so it may be unhelpful when the TN rate is low... when there is low prevalence

Does my predictor catch most of the true positives without causing lots of false positives?

AUROC

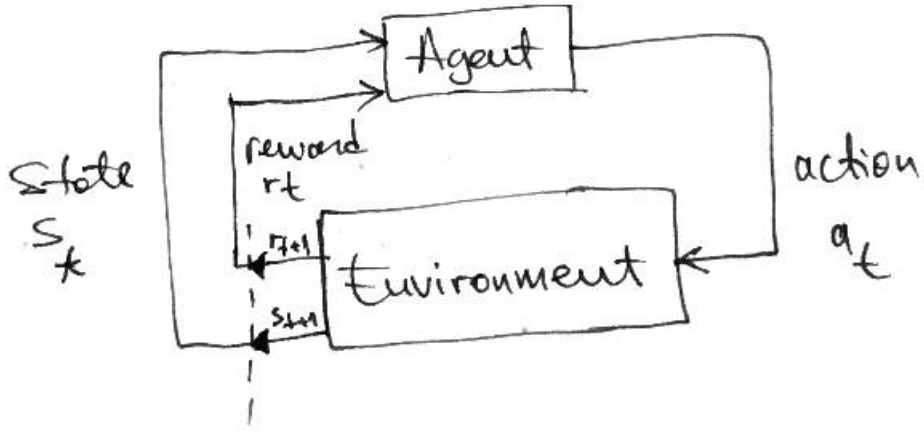
Does my predictor provide good results for positive and negative predictive value?

AIML Course : Reinforcement Learning =

* "If an action is followed by a satisfactory state of affairs, the tendency of the system to produce that action is strengthened or reinforced".
* Richard Sutton, 1991 – inventor of Reinforcement Learning.
* initially framed in adaptive optimal control.
* solution for solving non-linear control problems for which there is no excessive resource (memory) or CPU-time consumption but rather the designer doesn't know how to program "the correct things/actions", e.g. pole balancing, airplane stabilization.
* RL has been seen as a derivative of supervised learning based on trial-and-error (and reward):
  - but there is no teacher;
  - output is quantified as positive or negative reward if system is closer or further from the goal respectively.

* Basic formulations & elements:

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



Brief description:

→ The policy: shows how to choose a good action for a given state;

→ The value function: shows how good/valuable a state is;

→ The reward: shows how much reward does being in a certain state bring;

(RL) searches a mapping from state to action by trial-and-error.

The environment:

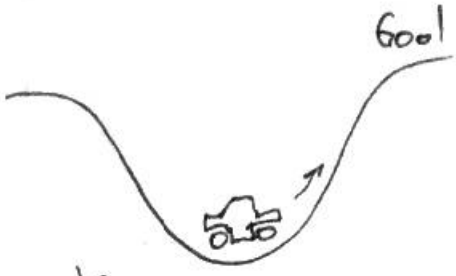
- must be observable through sensory readings & actions;
- must be deterministic, such that same actions lead to same results in repeated trials;
- these constraints are relaxed in more complex algorithms.

The reward:

- the reward r(s), or the reinforcement signal depicts a mapping from state to action.
- the environment gives reward to the learning system (agent).
- the agent tries to maximize all expected future reward!
- there are at least 3 types of reward:
  (a.) pure delayed reward: all reward is zero until the end state is reached and the sign of the final reward indicates success or failure: e.g. backgammon, inverted pendulum

(b.) minimum time to goal reward:

e.g. get a car uphill, the reward would be -1 at every step and 0 once the car reached the goal.



(c.) multi-player games:

→ two or more agents work simultaneously to achieve potentially opposing goals: e.g. predator-prey scenario.

- The policy:
  - the policy $\pi(s)$ is responsible to map from state to action to be taken, this needs to be learnt → how to choose a good action for a given state.

The value function:

- the value function $V(s)$ shows how good / valuable a state is.
- the value is defined as the sum of the expected future rewards when starting from a state s following a policy $\pi(s)$.
- the value can be represented as: a look-up table, equation or even a neural network (e.g. Deep Mind system playing GO).

There are 2 models for value functions:

1. Finite horizon model:

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

2. Infinite horizon model:

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

γ = discount factor

$\gamma^t \in]0,1]$

Such problems can also be solved with Markov Decision Processes (MDP).

- these models assume the complete state of the world is available to the agent → this is unrealistic.

An extension are the Partially Observable MDP or POMDP.

- model the information available to the agent by using a function from the hidden state to the observables.
- the goal is to find a mapping from observations (not states!) to actions

---

In real problems:

- environments contain non-determinism
- agent's actions can fail
- sensors can be inaccurate
- MDPs can help to take decisions under uncertainty

⇓

formulate the total reward!

We need to 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 one maximizing this equation!

There are several methods to solve this problem.

1. Value iteration

- if we know the true value of each state choose the action that maximizes the utility.
- but we don't know a state's initial value, just its immediate reward
- let's assume we have a look-up table of values $r(s)$ and we sweep though the table to update v_t'; the value at next step:

$$V'_{t+1} = \max_{\forall \Pi(s)} \left(r(s(t+1)) + \gamma^{\Pi} V(s(t+1)) \right)$$

or incrementally

$$\Delta V'_{t+1} = \eta \left(\max_{\Pi(s)} \left(r(s(t+1)) + \gamma^{\Pi} V(s(t+1)) \right) \right)$$

- the value iteration algorithm finds the optimal policy by taking into account long term reward $r(s)$
- but we need to know the results of all actions ($\max_{\Pi(s)}$ given by) in a given state and needs long time to converge

The algorithm for value iteration:

#1. initialize V(s) randomly for all states except target (goal).
#2. initialize V with target value for the goal.
#3. repeat:
#4. Initialize $\Delta = 0$
#5. for all $s \in S$ (states) in random order
#6. Calculate the difference $v - V(s)$
#7. Calculate value $V(s) = \max_{\pi(s)} \left(r(s(t+1)) + \gamma^t V(s(t+1)) \right)$
#8. Calculate difference for update $\Delta = \max(\Delta, |v - V(s)|)$
#9. end for
#10. until $\Delta == 0$ or $\Delta \leq 0$

2. Continuous states, Residual Gradient Algorithm

- RL converges (guaranteed) for look-up tables yet is unstable for function approximation systems which can contain even small amounts of generalization.
- we can see the value function as

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

$V^*(s)$ is the "perfect" value of the state and $e(s)$ is the error.

- a solution is to use Neural Networks to approximate V(s)! ⟹

$\Rightarrow$ such that

$$V^*(s(t)) = V(s(t), w(t)) = \text{NeuralNet}(s(t), w(t))$$

– the change in weights of the NN will be:

$$\Delta w(t) = -\eta \left(\max_{\Pi(s)} \left(r(s(t+1)) + \gamma^n V(s(t+1)) \right) - V(s(t+1)) \right) \frac{\partial V(s(t)), w(t)}{\partial w(t)}$$

where

$$-\eta \left(\max_{\Pi(s)} \left(r(s(t+1) + \gamma^n \cdot V(s(t+1)) \right) - V(s(t+1)) \right)$$

produces an error for current V

and $\dfrac{\partial V(s(t), w(t))}{\partial w(t)}$ is the change in V with respect to the weights (backprop)

– this method allows to "interpolate" between states and use RL even when not all states are explored

---

final remarks:

– value iteration has weaknesses:

* long convergence time
* it computes the value of each state which we don't need, that's just useful to find the optimal policy.

– value iteration can be adopted to policy iteration.

Q-Learning

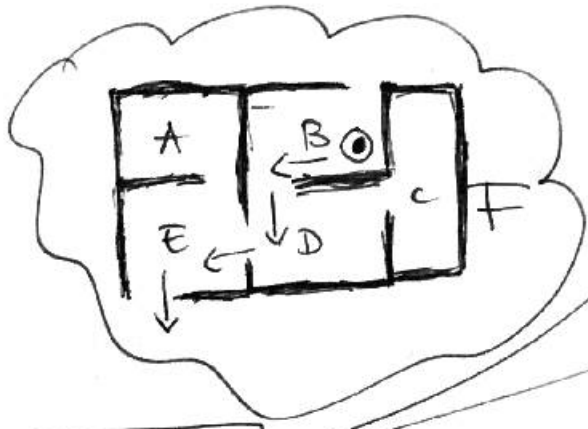
- value/policy iteration work well in finding an optimal policy but they assume that the agent has a great deal of domain knowledge.
- trade-off between learning time and apriori knowledge : Q-learning.
- model-free RL algorithm.
- agent doesn't have any model of the environment.
- agent has only knowledge of what states exist and what actions are possible in each state.
- The our idea is to map the tuple (state, action) to reward.
- We define a function $Q(s,a)$ as a reward for a given action in a state plus all future rewards along 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 $s(t+1)$ and not a large number of $s(t+1)$.

Let's introduce the algorithm through an example

— a search and rescue robot is supposed to start in any of the rooms and find the best way to reach the outside world (F) from that room.



if γ close to 1 introduce delay and wait for greater reward

if γ close to 0 consider only immediate reward.

Algorithm:

#1. set parameter γ and reward matrix R

#2. initialize Q with $\emptyset$

#3. for each period of exploration (episode)

#4. select random initial state

#5. do while not reached the goal (F)

#6. • select one among all possible actions for the current state

#7. • using this action, consider going to next state

#8. • get max Q value of this next state for all possible actions

#9 • calculate $Q(s,a) = r(s,a) + \gamma \max Q(s^*, a)$

#10. • set next state as current state

end

- again for continuous states we have a MLP to estimate the change in $Q(s,a)$:

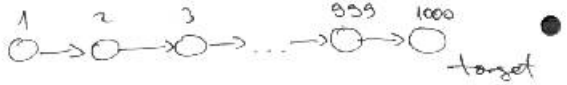
$$\Delta Q(s(t),a(t)) = \left(r(s(t),a(t)) + \gamma \max Q(s'(t),a(t)) - Q(s(t),a(t))\right)$$

and the change in weights is

$$\Delta w(t) = -\eta \Delta Q(s(t),a(t)) \frac{\partial Q(s(t),a(t))}{\partial w(t)}$$

A more general method to learn the value function and the Q function is TD-Learning or temporal difference learning.

- assume we have a RL system in which a sequence of actions is deterministic:



- we init states randomly.
- "true" information flows from right to left, 1000 steps until "true" information arrives at state 1.
- The formulation will be:

$$Q_i^1(s(t),a(t)) = r(s(t),a(t)) + \gamma \max Q(s(t+1),a(t+1))$$

$$Q^2(s(t),a(t)) = r(s(t),a(t)) + \gamma r(s(t+1),a(t+1)) + \gamma^2 \max Q(s(t+2),a(t+2))$$

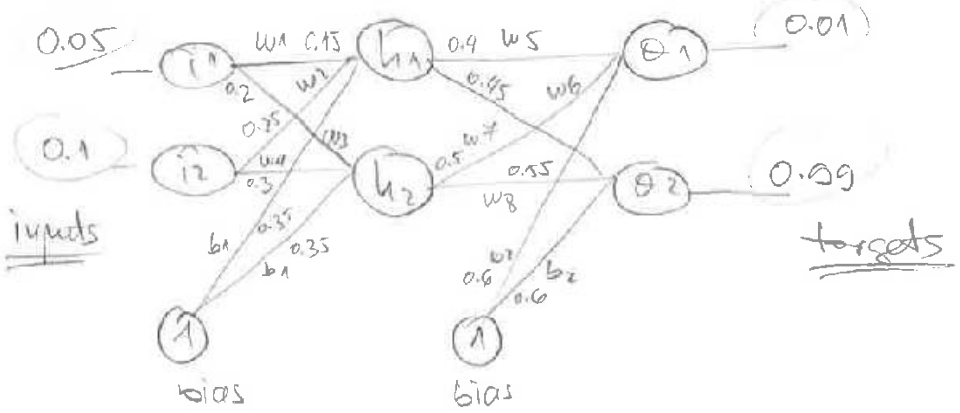
$$Q^3(s(t),a(t)) = r(s(t),a(t)) + \gamma r(s(t+1),a(t+1)) + \gamma^2 r(s(t+2),a(t+2)) + \gamma^3 \max Q(s(t+3),a(t+3))$$

$$Q^n(s(t),a(t)) = r(s(t),a(t)) + \gamma r(s(t+1),a(t+1)) + \gamma^2 r(s(t+2),a(t+2)) + \dots + \gamma^{n-1} r(s(t+n-1),a(t+n-1)) + \gamma^n \max Q(s(t+n),a(t+n))$$

↦ TD learning learns a guess from a guess, it bootstraps.

↦ TD Learning are naturally implemented online (e.g. important in robotic systems) and fully incrementally.

Back-prop - Worked example on MLP



Parameters

Activation function in hidden and output neurons is logistic

$$f(x) = \frac{1}{1+e^{-x}}$$

Learning rate

$\eta = 0.5$

Forward propagation

Hidden layer

$$net_{h_1} = w_1 \cdot i_1 + w_2 \cdot i_2 = 0.05 \cdot 0.15 + 0.1 \cdot 0.25 + 1 \cdot 0.35 = 0.3825$$

$$out_{h_1} = f(net_{h_1}) = \frac{1}{1+e^{-0.3825}} = \frac{1}{1.6821} = 0.6142$$

$$net_{h_2} = w_3 \cdot i_1 + w_4 \cdot i_2 = 0.05 \cdot 0.2 + 0.1 \cdot 0.3 + 1 \cdot 0.35 = 0.3900$$

$$out_{h_2} = f(net_{h_2}) = \frac{1}{1+e^{-0.3900}} = \frac{1}{1.6770} = 0.5963$$

Output layer

$$out_{o_1} = \frac{1}{1+e^{-net_{o_1}}}$$

$$net_{o_1} = w_5 \cdot out_{h_1} + w_6 \cdot out_{h_2} + b_2 \cdot 1 = 1.1438$$

$$out_{o_1} = \frac{1}{1.3186} = 0.7583$$

$$net_{o_2} = w_7 \cdot out_{h_1} + w_8 \cdot out_{h_2} + b_2 \cdot 1 = 1.2043$$

$$out_{o_2} = \frac{1}{1+e^{-1.2093}} = \frac{1}{1.2999} = 0.7692$$

Error at output

$$E_{total} = \sum \frac{1}{2}(target - out)^2 \quad \text{(we use MSE)}$$

$$E_{total} = E_{o_1} + E_{o_2}$$

$$E_{o_1} = \frac{1}{2}(target_{o_1} - out_{o_1})^2 = \frac{1}{2} \cdot (0.01 - 0.7583)^2 = 0.2799$$

$$E_{o2} = \frac{1}{2}\left(target_{o2} - out_{o2}\right)^2 = \frac{1}{2}\left(0.99 - 0.7692\right)^2 = 0.0243$$

$$E_{total} = E_{o1} + E_{o2} = 0.2795 + 0.0243 = 0.3092$$ (2)

Backward propagation

We only look at w_5 weight update in output layer

"chain" rule

$$\frac{\partial E_{total}}{\partial w_5} = \underbrace{\frac{\partial E_{total}}{\partial out_{o1}}}_{1)} \cdot \underbrace{\frac{\partial out_{o1}}{\partial net_{o1}}}_{2)} \cdot \underbrace{\frac{\partial net_{o1}}{\partial w_5}}_{3)}$$

1.) $$\frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial\left[\frac{1}{2}\left(target_{o1} - out_{o1}\right)^2 + \frac{1}{2}\left(target_{o2} - out_{o2}\right)^2\right]}{\partial out_{o1}}$$

$= 0$ no dependency on out_{o1}

$$\frac{\partial E_{total}}{\partial out_{o1}} = \frac{1}{2} \cdot 2 \cdot \left(target_{o1} - out_{o1}\right) \cdot (-1) + 0$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = -\left(target_{o1} - out_{o1}\right) = -\left(0.01 - 0.7583\right) = 0.7483$$

2.)

$$\frac{\partial out_{o1}}{\partial net_{o1}} = \frac{\partial f\left(net_{o1}(+)\right)}{\partial net_{o1}} = f'\left(net_{o1}(-1)\right)$$

The derivative of the activation function $f(x) = \frac{1}{1+e^{-x}}$ is $\frac{d}{dx} f(x) = f(x)\left(1 - f(x)\right)$

$$\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}\left(1 - out_{o1}\right) = 0.7583\left(1 - 0.7583\right) = 0.1832$$

3.) $net_{o1} = w_5 \cdot out_{h_1} + w_6 \cdot out_{h_2} + b_2 \cdot 1$

$$\frac{\partial net_{o1}}{\partial w_5} = 1 \cdot out_{h_1} + 0 + 0 = out_{h_1} = 0.6142$$

So $$\frac{\partial E_{total}}{\partial w_5} = 0.7483 \cdot 0.1832 \cdot 0.6142 = 0.0841$$

And so the full weight update for w_5

$$w_5(t+1) = w_5(t) - \eta \cdot \frac{\partial E_{total}(t)}{\partial w_5(t)} = 0.4 - 0.5 \cdot 0.0841$$

$$w_5(t+1) = 0.3579$$

Similar to w_6, w_7, w_8

Hidden layer weights update

$$\frac{\partial E_{total}}{\partial w_1} = \boxed{\frac{\partial E_{total}}{\partial out_{h_1}}}_{1)} \cdot \boxed{\frac{\partial out_{h_1}}{\partial net_{h_1}}}_{2)} \cdot \boxed{\frac{\partial net_{h_1}}{\partial w_1}}_{3)}$$

1.) $$\frac{\partial E_{total}}{\partial out_{h_1}} = \boxed{\frac{\partial E_{o1}}{\partial out_{h_1}}} + \frac{\partial E_{o2}}{\partial out_{h_1}}$$

$$\frac{\partial E_{o1}}{\partial out_{h_1}} = \frac{\partial E_{o1}}{\partial net_{o1}} \cdot \frac{\partial net_{o1}}{\partial out_{h_1}} = \boxed{\frac{\partial E_{o1}}{\partial out_{o1}} \cdot \frac{\partial out_{o1}}{\partial net_{o1}}} \cdot \frac{\partial net_{o1}}{\partial out_{h_1}}$$

$\frac{\partial E_{o1}}{\partial out_{o1}} \rightarrow 0.7483$ $\quad$ $\frac{\partial out_{o1}}{\partial net_{o1}} \rightarrow 0.1832$

$$\frac{\partial net_{o1}}{\partial out_{h1}} = \frac{\partial [w_5 \cdot out_{h1} + w_6 \cdot out_{h2} + b_2 \cdot 1]}{\partial out_{h1}} = w_5 =$$

$$= 0.4 \quad (\text{we use time } t)$$

(4)

$$\frac{\partial E_{o1}}{\partial out_{h1}} = 0.7413 \cdot 0.1932 \cdot 0.4 = 0.0548$$

Similarly $\frac{\partial E_{o2}}{\partial out_{h1}} = -0.0150$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}} = 0.0548 + (-0.0150) =$$

$$= 0.0358$$

(2.) $$\frac{\partial out_{h1}}{\partial net_{h1}} = out_{h1}(1 - out_{h1}) = 0.6142(1 - 0.6142)$$

$$= 0.2369$$

3.) $$\frac{\partial net_{h1}}{\partial w_1} = \frac{\partial [w_1 \cdot i_1 + w_2 \cdot i_2 + b_2 \cdot 1]}{\partial w_1} = i_1 = 0.05$$

Combining

$$\frac{\partial E_{total}}{\partial w_1} = 0.0358 \cdot 0.2369 \cdot 0.05 = 0.0004$$

weight update

$$w_1(t+1) = w_1(t) - \eta \cdot \frac{\partial E_{total}(t)}{\partial w_1(t)} = 0.15 - 0.5 \cdot 0.0009$$

$$= 0.1498$$

Similar w_2, w_3, w_4

For inputs 0.05 and 0.1 the net error was 0.3042. After running backprop of 1 step the error is 0.2940. If we repeat 10000 times error becomes 0.000035.

After 10000 epochs of training if we feed 0.05 and 0.1 the net will output 0.015 (vs. 0.01 target) and 0.989 (vs. 0.99 target)

Supervised neural computation

AIML Class

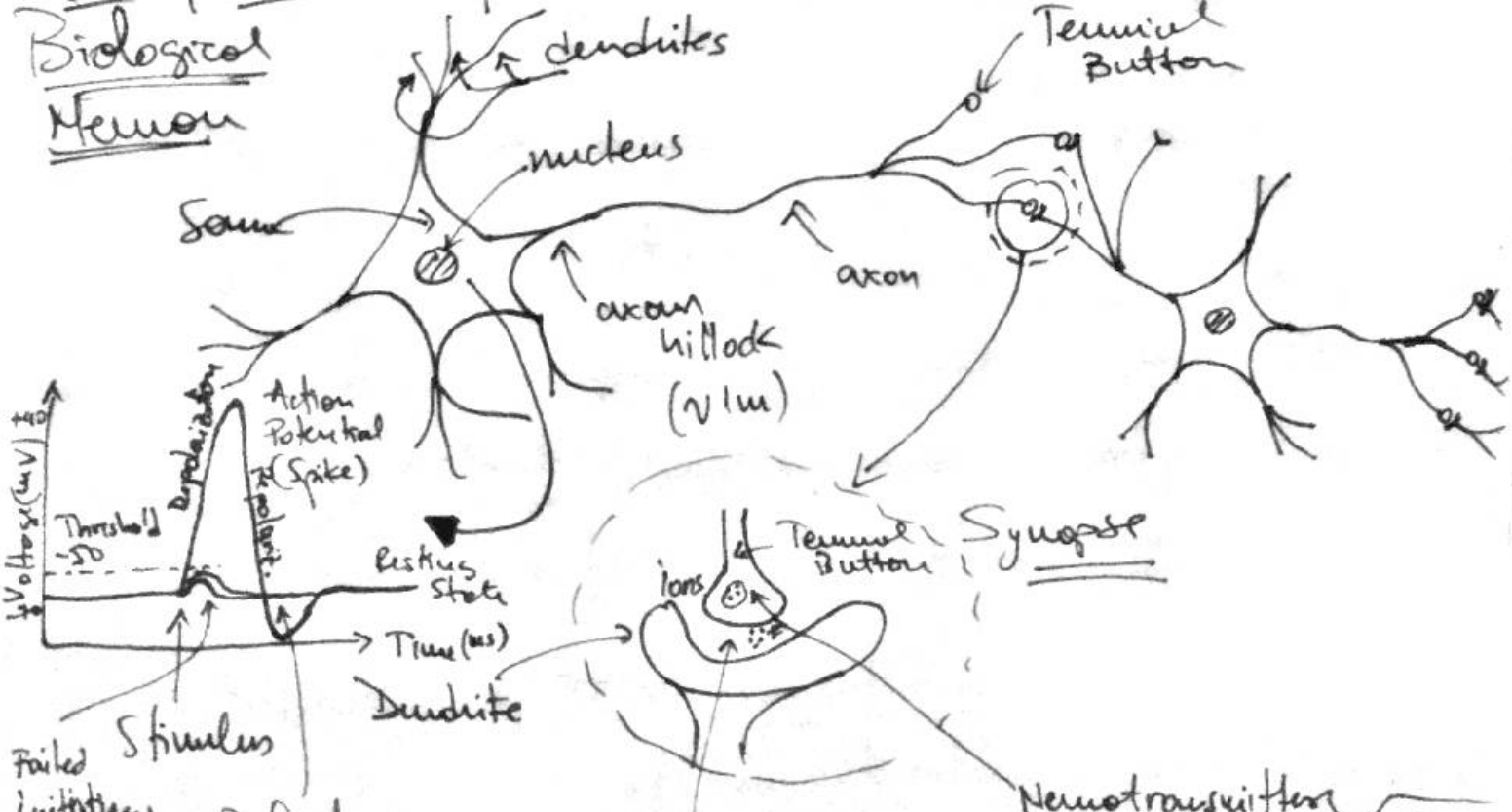
Introduction to neural networks

Neural systems have the ability to:

- extract meaning from complex, imprecise & often noisy data; (~ 10–17% of sensory input handled)
- learn patterns & trends;
- generalize & respond to unexpected inputs;

On short timescales, neurons act as devices which map inputs at synapses to sequence of action potentials (spikes) at the output.

The Biological Neuron

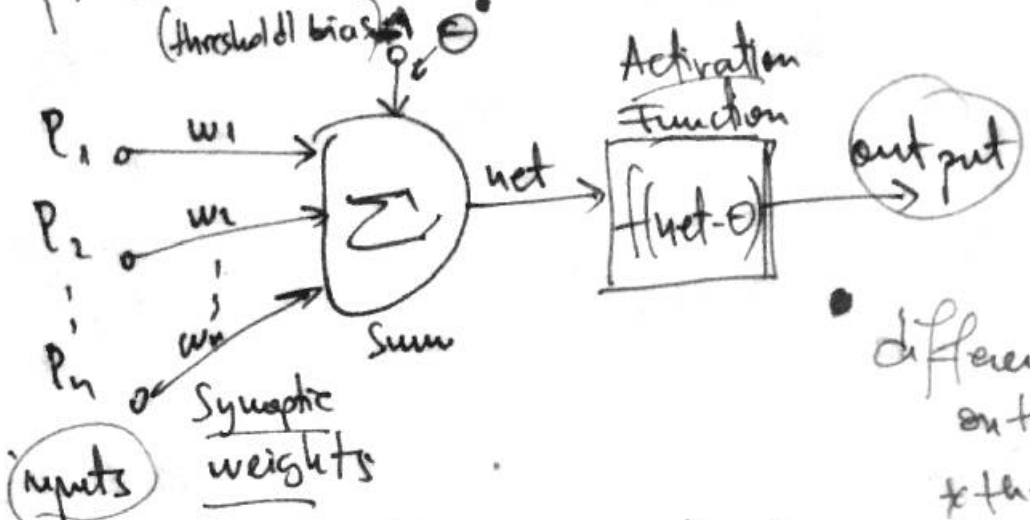


Spike freq. 10 Hz → 100 Hz

Human Brain
~ 10^{11} Neurons
~ 10^{14} Synapses

The Artificial Neuron

- emulates the biological one & collects input signals (pre-synaptic spikes) and provides an output after reaching a threshold (post-synaptic spike(s))



different views on the bias
* threshold
* persistent activity of the neuron when no input

The net input to the neuron is:

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

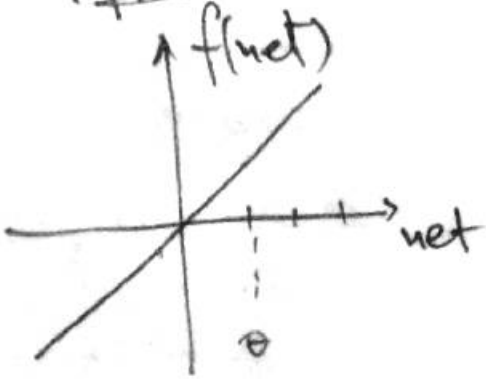
The output is computed as: (Θ, bias is an intrinsic property of the neuron)

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

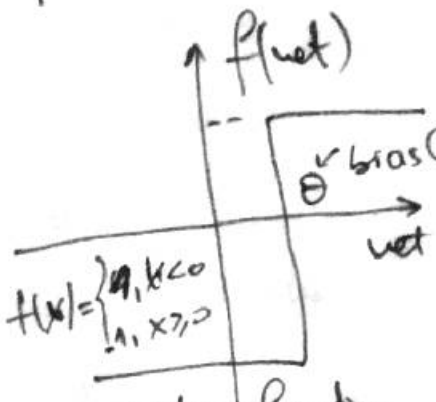
↑ viewed as threshold to fire

- The neuron performs a nonlinear mapping from input to output typically from a high-dimensional input space to a low-dimensional output space (1d, usually)
- mathematically is a weighted sum of the inputs and the application of an activation (squashing) function that determines the output of the neuron.

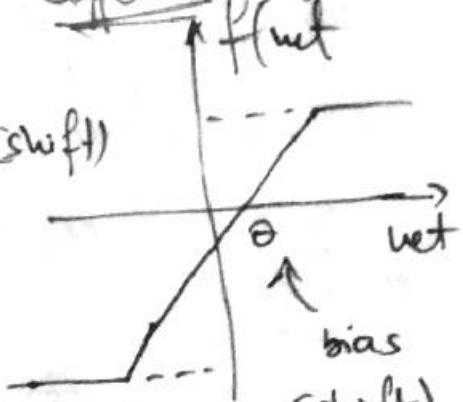
Typical nonlinearities (activation functions) are monotonically increasing functions with different effects on the output of the neuron. (they have to be differentiable).



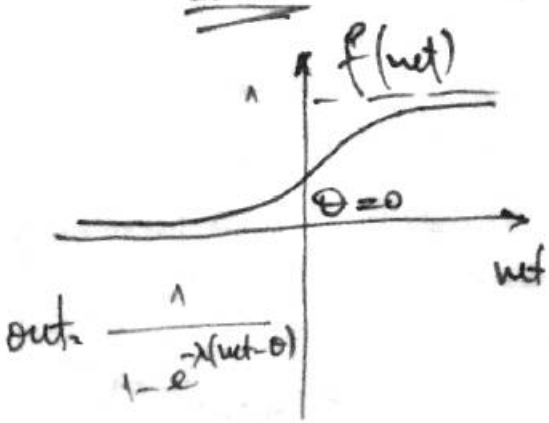
Linear



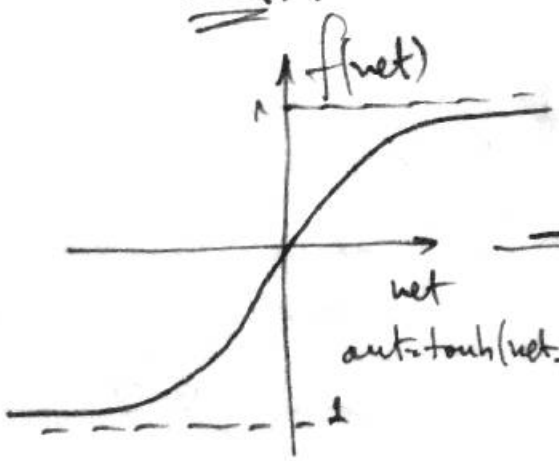
Step function



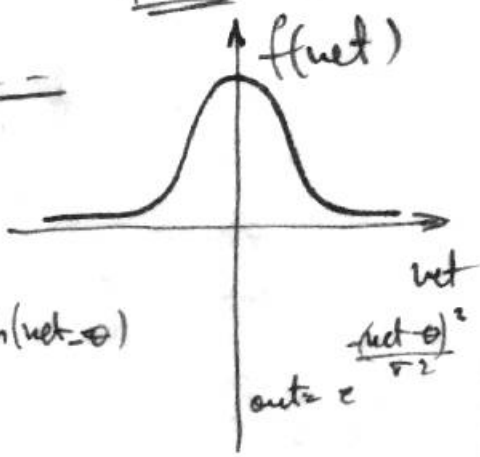
Ramp (shift)



Sigmoid



Hyperbolic Tangent



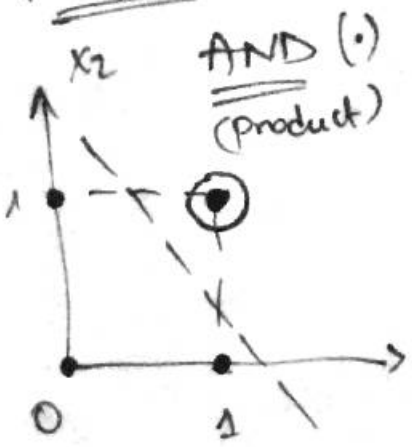
Gaussian

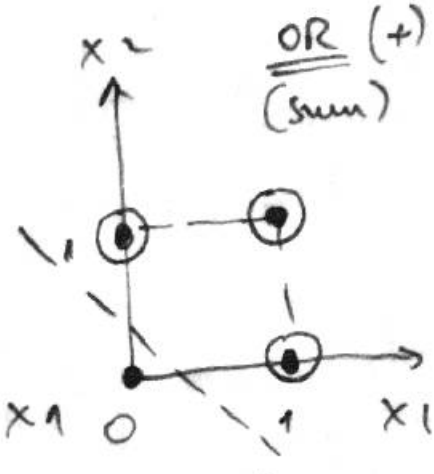
But, what can a single neuron compute? → Rosenblatt's definition

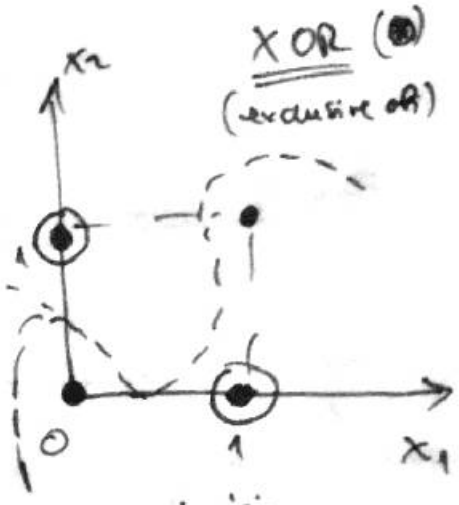
A 2-input perceptron with a single neuron with a step activation function is capable of separate / classify input patterns (simple e.g. nr. <0, ≥0).

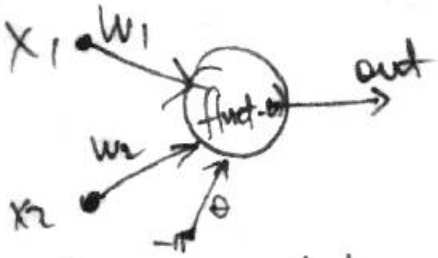
Such a classification implies a decision / separation boundary which is determined by the input vectors for which the net input is zero.

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









Sample solution:

$w_1 = w_2 = 0.8$

$\theta = 1$

Why?

$0.8x_1 + 0.8x_2 - 1 < 0$

for $x_1 = x_2 = 0$

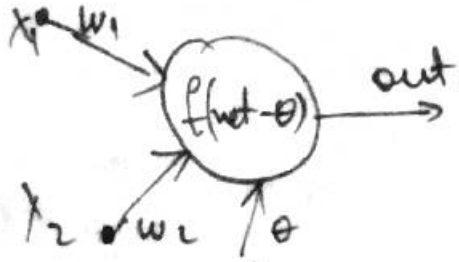
$x_1 = 1 \quad x_2 = 0$

$x_1 = 0 \quad x_2 = 1$

otherwise

$0.8x_1 + 0.8x_2 - 1 \geq 0$

if $x_1 = x_2 = 1$



Sample solution:

$w_1 = w_2 = 0.8$

$\theta = 0.5$

Why?

$0.8x_1 + 0.8x_2 - 0.5 < 0$

for $x_1 = x_2 = 0$

otherwise

$0.8x_1 + 0.8x_2 - 0.5 \geq 0$

for $x_1 = x_2 = 1$

$x_1 = 1 \quad x_2 = 0$

$x_1 = 0 \quad x_2 = 1$

(not) linearly separable problem!

⇓

We need a multi layer perceptron to cope with nonlinearity

⇓

Most lectures we build a network for such tasks

In order to build a linear decision boundary we need to train the system by updating the weights (for both inputs (ω) & bias (θ)).

Learning in artificial neurons

* training is based on Gradient Descent Learning Rule;
* minimize an error function of the mismatch between the target (expected) and the actual output of the neuron.

Error is given by: (Supervision)

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

t_p - target for pattern p use to train

out_p - output of the neuron for pattern p used to train

Squared error penalizes large errors more than small errors.

For each training pattern p the weight update follows the negative gradient in weight space to minimize the mismatch.

The equation for gradient descent weight update

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

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

Learning rate (can change or not in time)

If classes are linearly separable the algorithm converges in a finite number of steps.

Learning rule in a single neuron

$\mathcal{E}$ = error signal for the entire training dataset

$\mathcal{E}_p$ = error for pattern p used in training

w_i = current set of weights

$$\mathcal{E} = \sum_{\substack{\text{points in} \\ \text{dataset}}} (\text{true}_p - \text{out}_p)^2 \Rightarrow \mathcal{E}_p = (\text{true}_p - \text{out}_p)^2$$

Given the error we compute the gradient (derivative) of $\mathcal{E}_p$ w.r.t input weights of the linear neuron (we consider a linear activation function):

$$\Delta w_i(t) = \eta \cdot G_i(t) \quad \text{with} \quad G_i(t) = \frac{\partial \mathcal{E}_p(t)}{\partial w_i(t)}$$

which we can rewrite using the "chain rule":

$$G_i(t) = \frac{\partial \mathcal{E}_p(t)}{\partial w_i(t)} = \frac{\partial \mathcal{E}_p(t)}{\partial \text{out}_p(t)} \cdot \frac{\partial \text{out}_p(t)}{\partial w_i(t)}$$

with

- n = input size
- $\text{out}_p(t) = \sum_{i=1}^{n} P_i(t) \cdot w_i(t)$

And if we analyze both factors:

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

$$= -2 (true_p(t) - out_p(t))$$

and

(2) $$\frac{\partial out_p(t)}{\partial w_i(t)} = \frac{\partial \sum_{j=1}^{n} p_j(t) \cdot w_j(t)}{\partial w_i(t)} = p_i$$

only for $j = i$ the derivative exists

Combining (1) and (2) the gradient in the direction of the weight w_i is given by:

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

$$G_i(t) = -2 \cdot (true_p(t) - out_p(t)) \cdot p_i(t)$$

$$\Downarrow$$

$$\Delta w_i(t) = \eta \cdot (-2) \cdot (true_p(t) - out_p(t)) \cdot p_i(t)$$

η used as weight adaptation to minimize the error

& learning rate tells us how fast

One can use also non-linear activation functions as long as they are differentiable!

From single neurons to neural networks

(Single neuron perceptrons to multi-layer perceptrons)

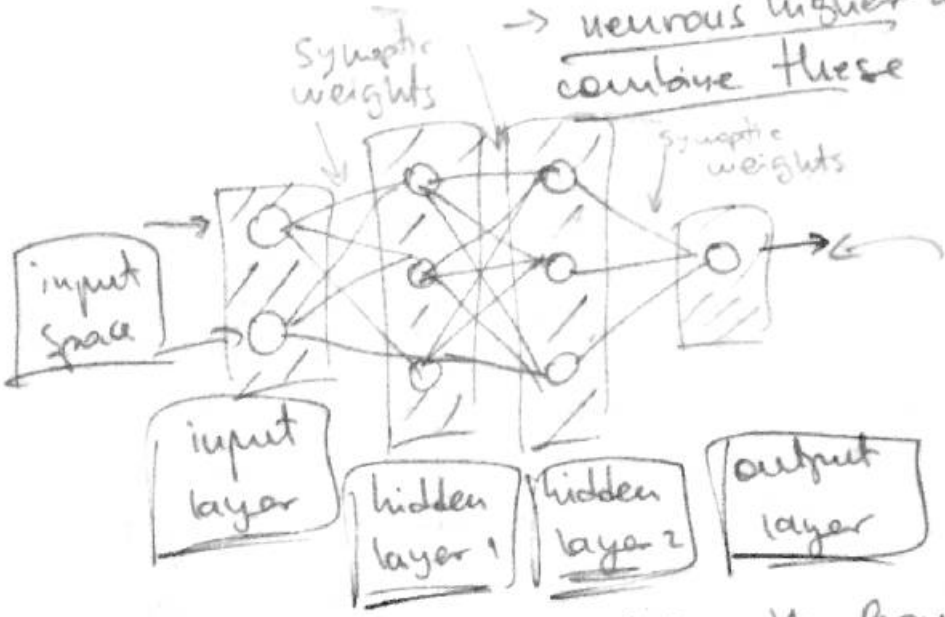
AIML Class

Q → What if the task is to learn "arbitrary data" which doesn't fit in a single neuron transfer function?

⇒ combine neurons in networks

→ each neuron represent an aspect of the data

→ neurons higher up in the hierarchy combine these



Typical structure for a feed-forward network

- The structure dictates the learning algorithm capabilities.
- We can have different processing models and learning capabilities depending on the network architecture: feed-forward, recurrent, deep etc.

Q What if we try to apply the gradient descent rule we used for a single neuron? Will it work for a network?

As here we have multiple layers of neurons, each one has a different contribution. We know indeed the output error of the network but no individual neuron error.

The core idea is to backpropagate the output (network) error through the layers in order to update the weights.

Backpropagation (Error backpropagation)

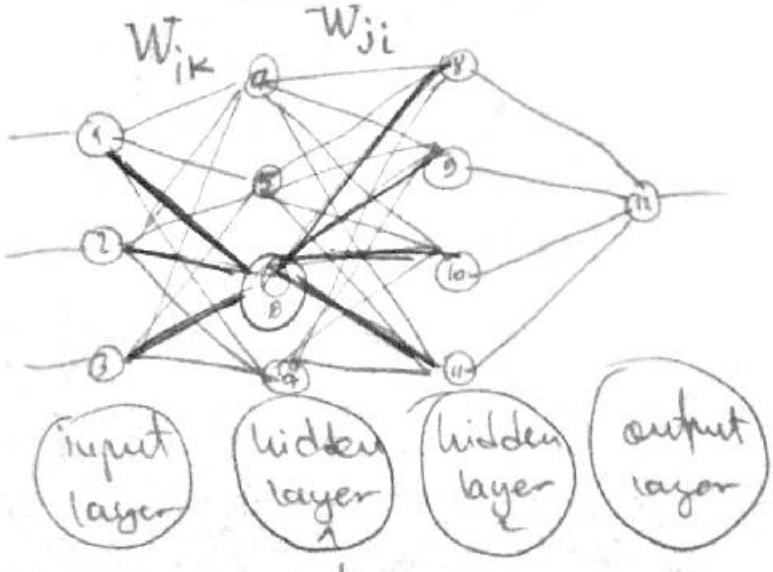
→ searches the minimum of the error function in weight space using gradient descent.

→ the particular combination of weights which minimizes the error function is considered to be the solution for learning a representation of the data.

CONDITIONS

→ requires computing the gradient at each training iteration, so we must guarantee that the error function is continuous and differentiable.

→ computes the partial derivatives of the cost/error function w.r.t. to any weights in the network



$K = \{1, 2, 3\}$
$i = 6$
$J = \{8, 9, 10, 11\}$

Notations:

O - output
j - posterior (P)
K - anterior (A)
i - current (C)

W_{ji} = W <TO> <FROM>

Definitions:

1. Error for unit i at time t

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

where the net input for neuron i is

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

output at previous layer

2. The weight change for $w_{ik}(t)$ is

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

- We need to rewrite the weight change in terms of the error at the current neuron ($\delta_i(t)$) and the output of the previous layer ($net_i(t)$) → this is the only information available at neuron i!

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

$$\boxed{\Delta w_{ik}(t) = \delta_i(t) \cdot out_k(t)}$$

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

$$\frac{\partial net_i(t)}{\partial w_{ik}(t)} = \frac{\partial \sum_{l \in A_i} w_{il}(t) \cdot out_l(t)}{\partial w_{ik}(t)} = out_k(t)$$

which is non-zero for $l = k$

3. Computing the forward activation of the network

- when input is applied to the network, we compute the output of all neurons through layers to the output

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

f_i: squashing function

(4.) Calculating the error at the output neuron

- Assuming that the expected / target value for the dataset point is $target_o$, then the error signal at the output, out_o, is

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

(here we assumed a linear output activation function)

(5.) Propagating the error back through the network

- after computing the error at the output of the net we propagate the signal back through the network
- the error at unit / neuron i is

$$\delta_i(t) = \frac{-\partial E_i(t)}{\partial net_i(t)}$$ and applying the "chain rule"

$$\delta_i(t) = \frac{-\partial E_i(t)}{\partial net_i(t)} = \sum_{j \in P} \underbrace{\left[\frac{-\partial E_i(t)}{\partial net_j(t)}\right]}_{1)} \cdot \underbrace{\left[\frac{\partial net_j(t)}{\partial out_i(t)}\right]}_{2)} \cdot \underbrace{\left[\frac{\partial out_i(t)}{\partial net_i(t)}\right]}_{3)}$$

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

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

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

- We combine 1), 2), 3) to calculate the error signal of neuron i in the network:

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

f is independent of j^{th} neuron so we can rewrite as:

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

⇓

the error for neuron i ($\delta_i(t)$) only depends on the known error in higher layers (posterior)

(6.) Computing the weight update for weight anterior to posterior ($k \leftarrow i$)

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

The final synoptic weight update rule is:

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

or

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

We use this rule to train multi-layer neural nets for tasks as classification & regression.

Supervised neural learning → tips & tricks

- the big problem with neural networks is parameter tuning ⟹ there is no optimal way to select network parameters.
- there are though some design choices for
  - (**) data pre-processing
  - (**) weight initialization
  - (**) error functions

1. Weights initialization

- zero initialization ⟹ each neuron computes same output they will also learn based on similar gradients and there will be no symmetry brake, and network will act as a single, highly redundant neuron
- to break symmetry initialize all neurons with small random numbers. (± 0.001)

2. IO normalization

- normalize each data dimension to be on approximately the same scale
- 2 ways:
  - * divide each dimension by its Std. dev., once is zero centred
  - * normalize to $(-1, 1)$ on each dimension

3. Small weight updates

- modulated by the learning rate $\eta(t)$, typically of > 0.001

4. Avoiding local minima in weight space

- due to the gradient based learning rule the training can get stuck in local minima.
- a solution is to use Simulated Annealing to slightly perturb the weights
  - large perturbation at the beginning to get the network out of minimum and then slow/decay with the training process until settles in a global solution (usually marked by small Δw)

5. Network size

- the no. of inputs & outputs given by the problem, cannot be changed.
- no. of neurons, no. of layers, type of squashing functions ?

  ⇓

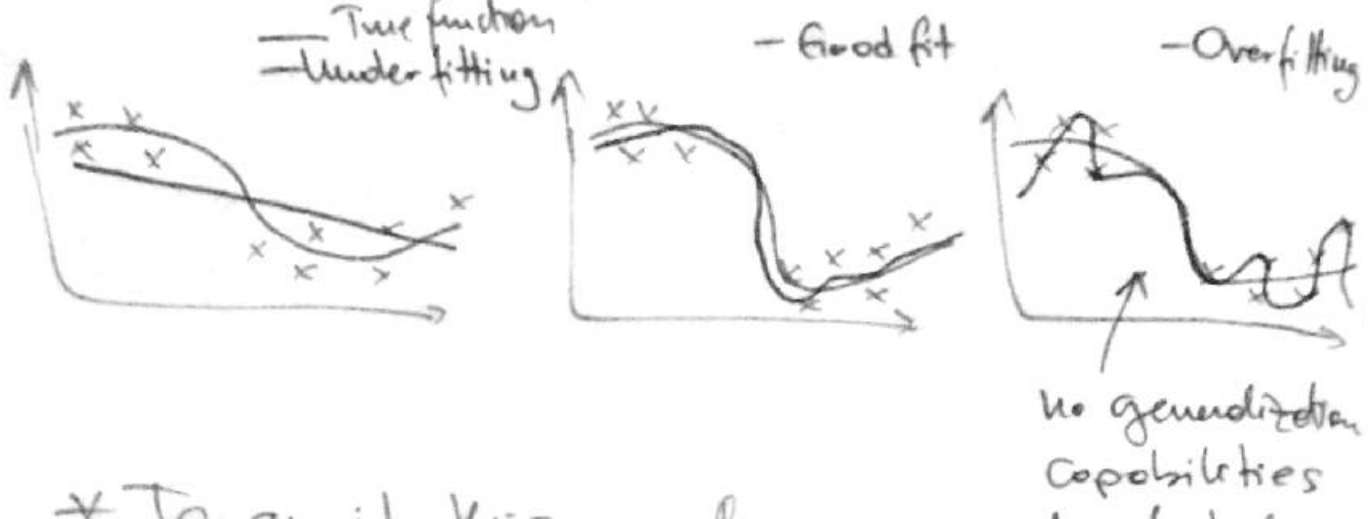
  a larger neuron count (capacity of the net) will allow the net to represent more complex functions

  ⇓

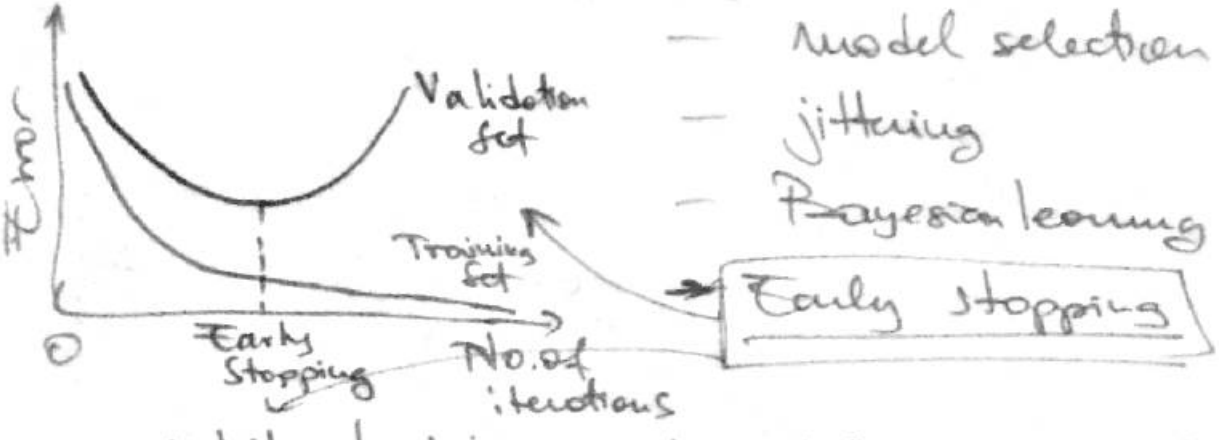
  but there is no universal recipe !

- Some "rules of thumb":
  - size of the hidden layer should be between the input layer size and the output size
  - in a single hidden layer MLP you need 2x more neurons than inputs

6. Overtraining / overfitting



* To avoid this, use large amounts of data as a rule!
* other methods are discussed:
  - model selection
  - jittering
  - Bayesian learning
  - Early stopping



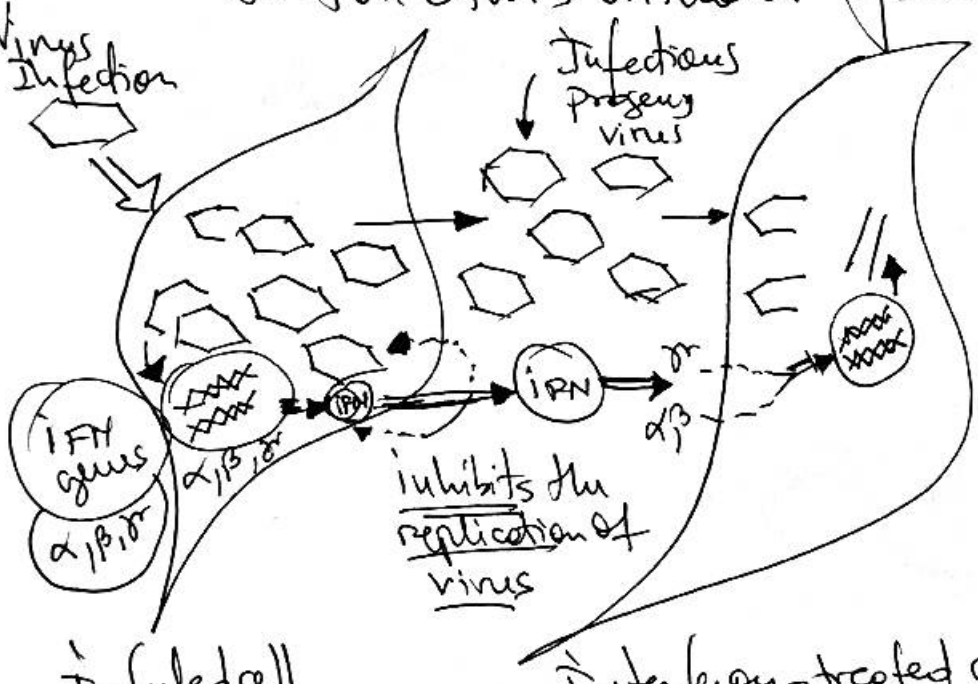
- while training on the dataset the network becomes better and better but at some point in time it starts to represent each point! → this is problematic → overfitting!
- We need to find the time where the network finishes generalizing and starts learning individual points
- Split dataset in 70% training + 30% testing
  (Training set) (Validation set)

Signaling & Message Passing Mechanisms in the Immune System

(1) signal diffusion
(2) dialogue
⟹ communication schemes

(1) Immune diffusion

- information passes from one immunocomponent to the others without feedback



- IFN has some shutdown across cells
- IFN travels to neighboring cells
- IFN regulated proteins inhibit virus replication

(2) Immune Dialogue

- immunocomponents exchange continuously molecular signals.
- sensitivity is context dependent.