Motivation/Rationale

Artificial Neural Networks (ANNs) are developed using mathematical models of biological neurons. The flexibility in the structures and learning mechanisms of ANNs enable us to model various real-world problems and solve them using ANNs.

Syllabus

Unit 3 – Artificial Neural Networks (ANNs) [_ hrs.]

3.1. History of ANNs (Mc Culloch and Pitts, Connectionist, XOR Problem)
3.2. Feedback (auto-associative networks)
3.3. Perceptrons
3.4. Multi-Layered Perceptrons

ARTIFICIAL NEURAL NETWORKS

Artificial Neural Networks can be modeled to compute as a conventional von Neuman processor. For this purpose, the primitive functions are located in the nodes of the network and the composition rules are contained implicitly in the interconnection pattern of the nodes.

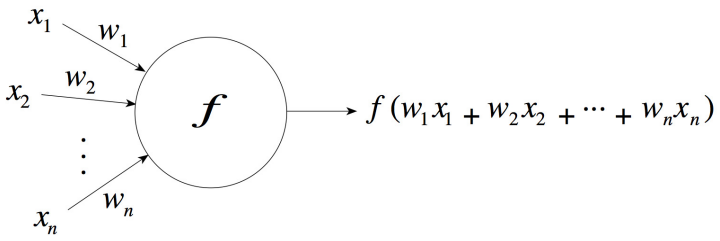


Fig. 1 : An abstract neuron

Each input channel i can transmit a real value xi. The **primitive function** f computed in the body of the abstract neuron can be selected arbitrarily. Usually the input channels have an associated weight, which means that the incoming information xi is multiplied by the corresponding weight wi. The transmitted information is integrated at the neuron (usually just by adding the different signals) and the primitive function is then evaluated.

Various functional models of ANNs can be designed by using the primitive functions.

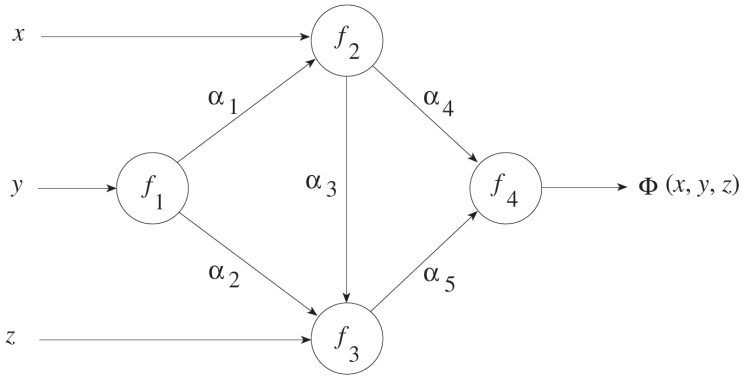


Fig. 2: Functional model using primitive functions

Threshold Logic Unit (TLU)

Early efforts at modeling neural networks used threshold logic, and are still valid for some types of applications.

- The input values are discrete, say {0, 1}.

- The weighted sum minus the threshold is called the "activation" or "net" value.
- The neuron fires if activation value is above a threshold value associated with the neuron.

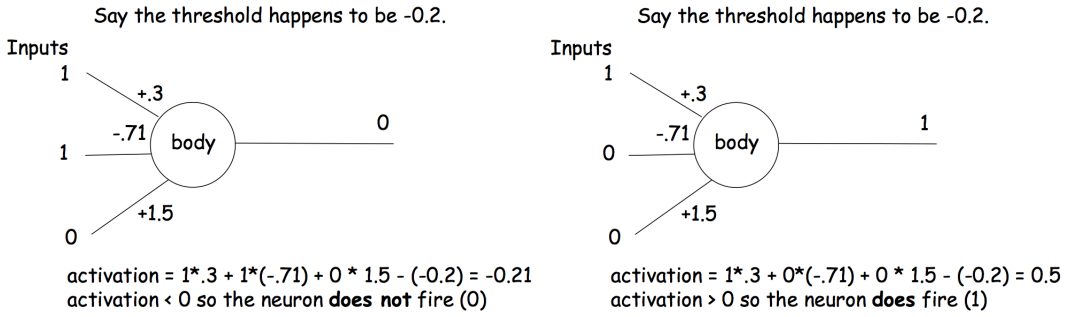


Fig. 3 : TLU with no fire and with fire.

The inputs with positive weights are called **excitatory** and inputs with negative weights are called **inhibitory**.

There are two types of primitive artificial neurons.

- McCulloch & Pitts
- Connectionist

McCulloch & Pitts Neuron (1943)

- These neurons are two state neurons i.e. inputs and output can be either 1 or 0
- The networks are composed of directed un-weighted edges of excitatory or of inhibitory type.
- Each McCulloch-Pitts unit is also provided with a certain threshold value θ.

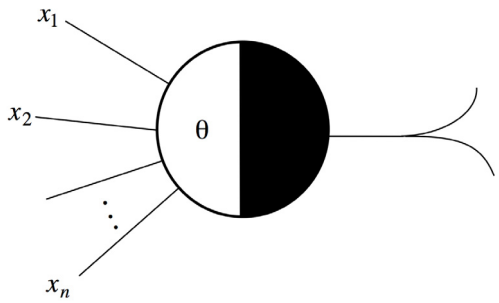


Fig. 4 : McCulloch and Pitts Neuron

The rule for evaluating the input to a McCulloch–Pitts unit is the following:

- Assume that a McCulloch–Pitts unit gets an input $x1, x2, \ldots, xn$ through n excitatory edges and an input $y1, y2, \ldots, ym$ through m inhibitory edges.
- If $m \geq 1$ and at least one of the signals $y1, y2, \ldots, ym$ is 1, the unit is inhibited and the result of the computation is 0.
- Otherwise the total excitation $x = x1 + x2 + \ldots + xn$ is computed and compared with the threshold θ of the unit *(if $n=0$ then $x=0$)*. If $x \geq \theta$ the unit fires a1, if $x<\theta$ the result of the computation is 0.

This rule implies that a McCulloch–Pitts unit can be inactivated by a single inhibitory signal, as is the case with some real neurons. When no inhibitory signals are present, the units act as a threshold gate capable of implementing many other logical functions of n arguments.

Following figure shows the activation function of a unit, the so-called step function. This function changes discontinuously from zero to one at θ. When θ is zero and no inhibitory

signals are present, we have the case of a unit producing the constant output one. If θ is greater than the number of incoming excitatory edges, the unit will never fire.

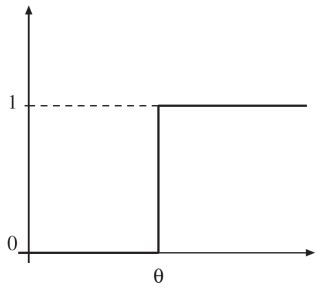


Fig. 5 : The step function with threshold θ

Synthesis of Boolean functions

The power of threshold gates of the McCulloch–Pitts type can be illustrated by showing how to synthesize any given logical function of n arguments. We deal firstly with the more simple kind of logic gates – conjunction, disjunction, negation.

Mappings from $\{0, 1\}^n$ onto $\{0, 1\}$ are called logical or Boolean functions. Simple logical functions can be implemented directly with a single McCulloch–Pitts unit. The output value 1 can be associated with the logical value true and 0 with the logical value false. It is straightforward to verify that the two units of following figure compute the functions AND and OR respectively.

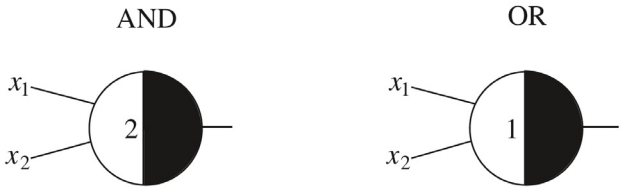


Fig. 6: Implementation of AND and OR gates

AND and OR gates alone cannot be combined to produce all logical functions of n variables. The uninhibited threshold logic elements are capable of implementing more general functions than conventional AND or OR gates. For example;

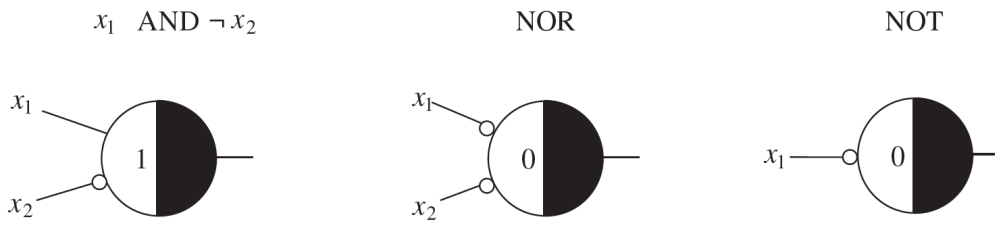


Fig. 7: Logical functions and their realization

The Connectionist Neuron

The connectionist neuron is more generic than MacCulloch and Pitts's. A neuron is capable of receiving multiple weighted inputs, integrate the inputs, and fire an action. The inputs and the weights can be any **real number**.

The action depends on various activation methods (Threshold, Linear Piecewise, Sigmoid etc.).

Linearly Separable and XOR problem

The logic functions (AND, OR, NOT) developed by using primitive ANNs exhibit linearly separable feature.

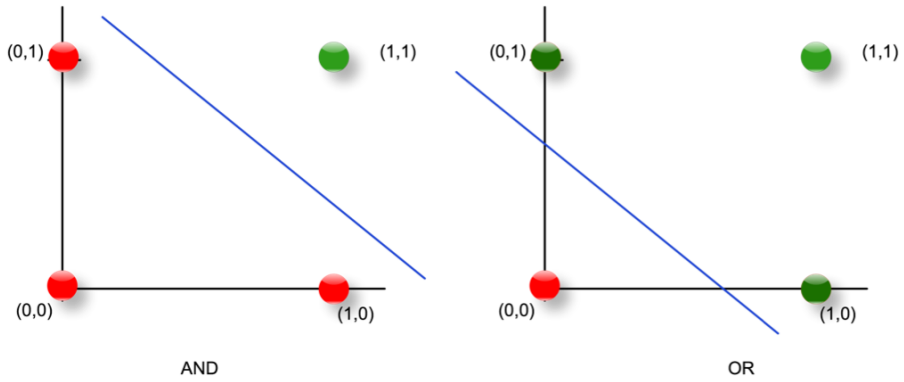


Fig. 8 : Linear separability

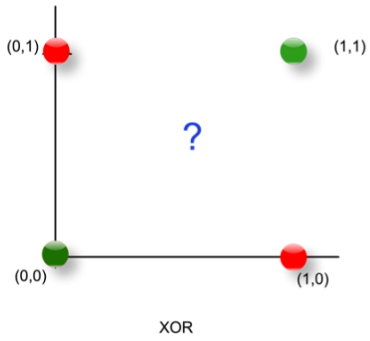


Fig. 9 : The XOR Problem

Auto-associative networks

An ANN is supplied with an input vector and same target-output vector. The functional output F(x) (output vector) of the network might not be same as output vector. Therefore, an error vector is calculated. The error is back propagated as a feedback to adjust the weights. The functional output F(x) is again calculated.

This process of calculating the error, back-propagating it as feedback, and readjusting the weights is continued until the output vectors match with input vectors. In such case, the neural network represent an association with the input vector itself. This type of ANN is known as auto-associative networks.

Auto-associative networks generally use back-propagation learning method which uses error-corrective feedback mechanism.

Perceptrons

A perceptron is a **linear threshold gate**. The decision made by perceptron is linear however, the structure of perceptron is **dynamic** i.e. different values of weights. The weights are adjusted by training the perceptron.

Structure of Perceptron

Rosenblatt's classical perceptron was designed for responses to retinal input signals.

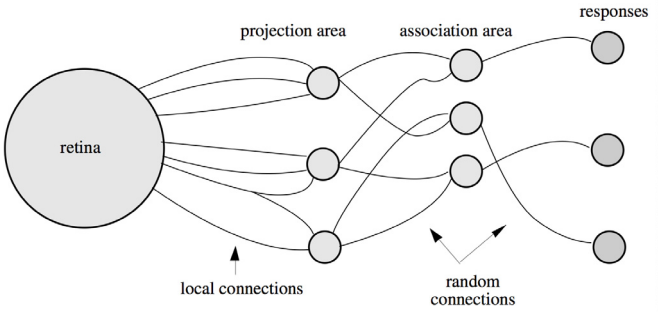


Fig. 10 : The classical perceptron (Rosenblatt)

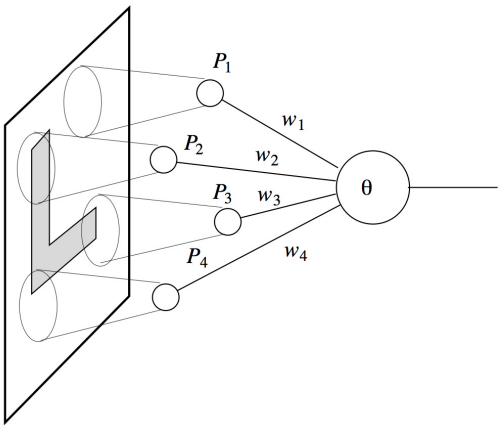
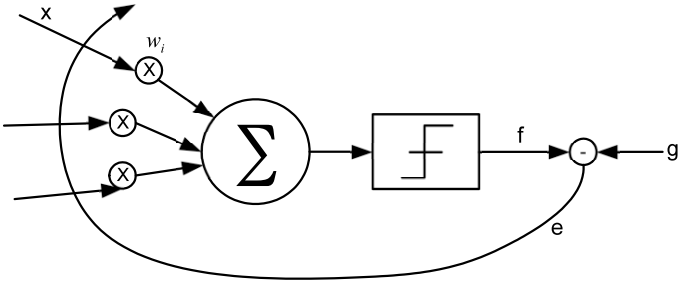


Fig. 11: Predicates and weights of a perceptron

The threshold element collects the outputs of the predicates through weighted edges and computes the final decision. The system consists in general of n predicates $P_1, P_2, \ldots, P_n$ and the corresponding weights $w_1, w_2, \ldots, w_n$. The system fires only when $\sum_{i=1}^{n} w_i . P_i \geq \theta$, where θ is the threshold of the computing unit at the output.

A generic perceptron is represented as;



The inputs are represented as;	input vector - X
The weights are repsented as;	weight vector - W
Target;	goal – G
Functional output;	sum of product of input and weight – F
Error;	(goal – sum) - E

A problem of retinal image classification can be modeled as follows;

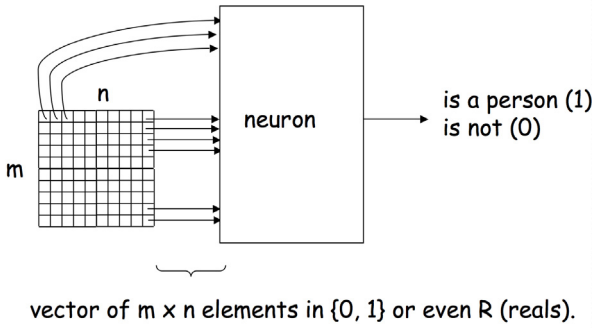


Fig. 12: Retinal image classification

In this retinal image classification problem, inputs from retina are taken as input vectors and output is defined by two possible results – identifiable image or un-identifiable image. In such case, output can be modeled as a vector with *1* and *0*.

Problem Solving

A perceptron must find a linear association between the input vectors and the output vectors.

The general idea is to find a vector of weights $\{w_i\}$ and a threshold θ, such that;

$$\text{Output} = \begin{cases} 1 \text{ if } \sum w_i x_i > \theta \\ 0 \text{ otherwise} \end{cases}$$

In other words,

Output = 1 if $\{X_i\}$ represents a vector in the set
0 otherwise (not in the set)

Issues

- **Existence**: Given a set of input vectors, does there **exist** a perceptron that correctly classifies the given inputs?
- **Solving**: Can the weights be found **analytically**?
- **Training**: Can the weights be found simply by presenting examples?

One example;

Suppose the signals are real-valued, and the following vectors are classified as shown:

input	**output**
(3, 4)	1
(6, 1)	1
(4, 1)	0
(1, 2)	0

Is there a perceptron that classifies the set as shown, and if so, what are its weights and threshold?

Let's have a geometric insight of the problem given.

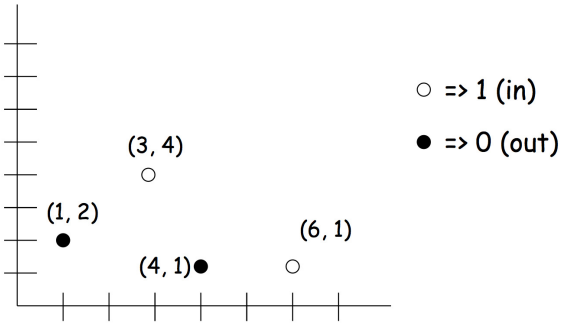


Fig. 13: Geometric insight of classification problem

At this point, a general line equation can be derived such as;

- $w_1 x_1 + w_2 x_2 = \theta$
- Output is
  - 1 if $w_1 x_1 + w_2 x_2 > \theta$
  - 0 otherwise

Regarding the classification problem, a question arises;

Will $\{w_i\}$ and θ always exist?

Generalizing to n dimensions

Given function $d:Rn \rightarrow \{0, 1\}$, to find θ, $wi \in R$ such that

$$w_1 x_1 + w_2 x_2 + ... + w_n x_n = \theta$$

separates the points:

- $w_1 x_1 + w_2 x_2 + ... + w_n x_n > \theta$ when $d(x1, x2, ..., xn) = 1$
- $w_1 x_1 + w_2 x_2 + ... + w_n x_n < \theta$ when $d(x1, x2, ..., xn) = 0$

The equation $w_1 x_1 + w_2 x_2 + ... + w_n x_n = \theta$ defines a ***hyperplane*** in n-space. If such a hyperplane exists for a classification problem, the problem is called **linearly-separable**.

The vector of weights $(w_1, w_2, ... w_n)$ is normal (**perpendicular**) to the hyperplane.

Threshold is proportional to the **distance** of the hyperplane from the origin.

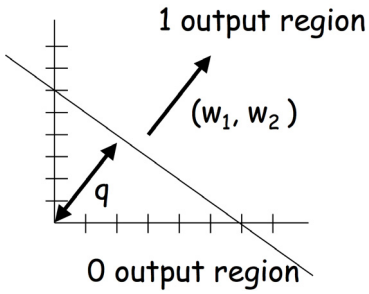


Fig. 14: Perceptron classification hyperplane

One more example of problem solving by perceptron (taken from Rojas)

Let's consider that a perceptron can decide whether a figure is connected or not. Consider the four patterns shown;

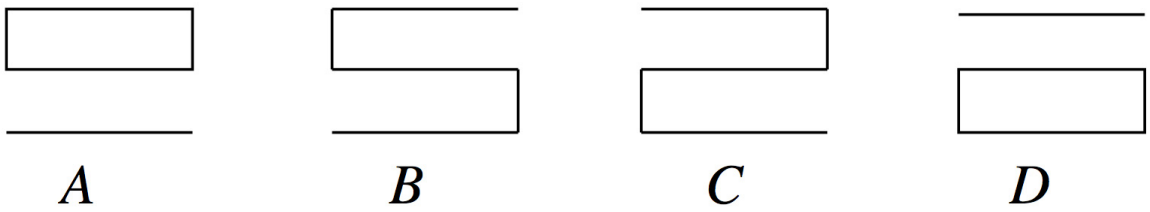


Fig. 15 : Connected graphs

Notice that only the middle two are connected. Since the diameters of the receptive fields are limited, the patterns can be stretched horizontally in such a way that no single receptive field contains points from both the left and the right ends of the patterns. In this case we have three different groups of predicates: the first group consists of those predicates whose receptive fields contain points from the left side of a pattern. Predicates of the second group are those whose receptive fields cover the right side of a pattern. All other predicates belong to the third group. In following figure the receptive fields of the predicates are represented by circles.

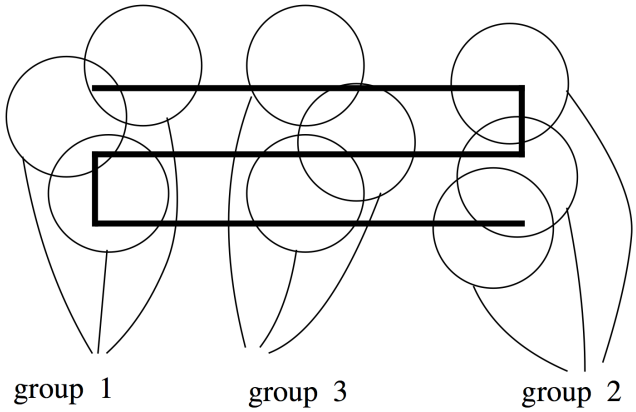


Fig. 16 : Receptive fields of predicates

All predicates are connected to a threshold element through weighted edges which we denote by the letter w with an index. The threshold element decides whether a figure is connected or not by performing the computation

$$S = \sum_{P_i \in \text{group 1}} w_{1i} P_i + \sum_{P_i \in \text{group 2}} w_{2i} P_i + \sum_{P_i \in \text{group 3}} w_{3i} P_i - \theta \geq 0$$

If S is positive the figure is recognized as connected.

Training the Perceptron (Learning)

(Practical - Training a perceptron for logic gate functions.)

The association between input vectors and output is defined by the weight vectors. A perceptron selected for the specific classification problem should be able to create association between input vectors and output. This is achieved by training the perceptron. The **perceptron learning algorithm** deals with the training problem.

Error correction

A learning algorithm is an adaptive method by which a network of computing units self-organizes to implement the desired behavior. This is done in some learning algorithms by presenting some examples of the desired input-output mapping to the network. A correction step is executed iteratively until the network learns to produce the desired response. The learning algorithm is a closed loop of presentation of examples and of corrections to the network parameters, as shown in following figure.

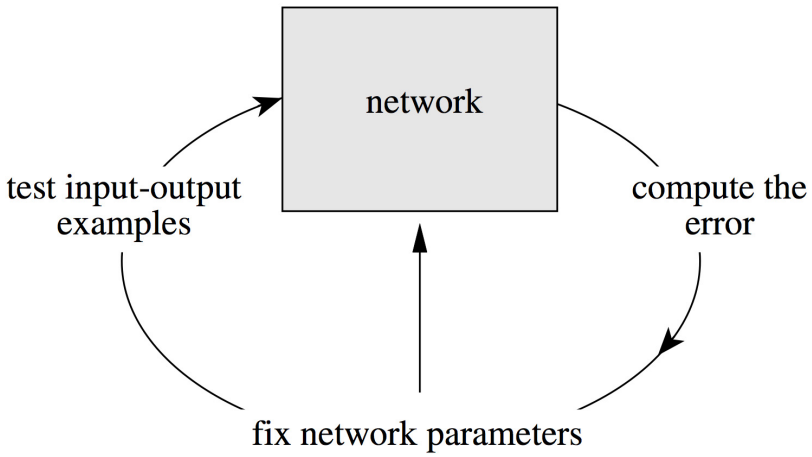


Fig. 18: The learning process

The training - complete

- Use a set of **training samples**:
  - Input vectors, each paired with desired output
- Choose a network structure.
- Initialize the weights and thresholds arbitrarily.
- Repeat:
  - Choose a sample
  - Test the network against the sample
  - If answer is incorrect, **adjust** weights
- until all samples test correctly.

Weight adjustment

The Perceptron learning rule (Rosenblatt):

- If the perceptron gives the correct answer, do nothing.
- If the perceptron gives the wrong answer, nudge the weights and threshold "**in the right direction**", so that it eventually gives the right answer.

When does a Perceptron give the wrong answer?

Correct answers:

$w1\ x1 + w2\ x2 + ... + wn\ xn > \theta$ when $d(x1, x2, ..., xn) = 1$;
$w1\ x1 + w2\ x2 + ... + wn\ xn < \theta$ when $d(x1, x2, ..., xn) = 0$

Wrong answers:

$w1\ x1 + w2\ x2 + \ldots + wn\ xn > \theta$ when $d(x1, x2, \ldots, xn) = 0$;
$w1\ x1 + w2\ x2 + \ldots + wn\ xn < \theta$ when $d(x1, x2, \ldots, xn) = 1$

Desired Output ***d*** vs Actual Output ***a***

Let $a(x1, x2, \ldots, xn) =$ 1 if $w1\ x1 + w2\ x2 + \ldots + wn\ xn > \theta$
0 otherwise.

Note that ***a*** is an implied function of the weights and threshold.

Error Value

We can capture correct vs. incorrect answers succinctly by introducing an error value ε:

$\varepsilon = d(x1, x2, \ldots, xn) - a(x1, x2, \ldots, xn)$ = desired - actual

So that

- ε = 0 when the correct answer is given
- ε = 1 when $w1\ x1 + w2\ x2 + \ldots + wn\ xn < \theta$ but $d(x1, x2, \ldots, xn) = 1$
- ε = -1 when $w1\ x1 + w2\ x2 + \ldots + wn\ xn > \theta$ but $d(x1, x2, \ldots, xn) = 0$

Simplification: Threshold conversion

The threshold can be treated as if one of the weights by introducing a "phantom" input of -1:

$w1\ x1 + w2\ x2 + \ldots + wn\ xn > \theta$

iff

$w1\ x1 + w2\ x2 + \ldots + wn\ xn - \theta > 0$

iff

$w0\ x0 + w1\ x1 + w2\ x2 + \ldots + wn\ xn > 0$
where w0 is defined to be θ and
x0 = -1 unchangingly.

Bias vs. Threshold

Instead of subtracting the threshold, we could add a "bias", in which case the phantom input would be 1 rather than –1. The actual value doesn't really matter, as long as it is not 0.

Perceptron training (continued)

Wrong answer of the first type (ε = 1): w0 x0 + w1 x1 + w2 x2 + …. + wn xn < 0 when d(x1, x2, …., xn) = 1

i.e., w0 x0 + w1 x1 + w2 x2 + …. + wn xn is **too low**.

To correct for this, we need to make the sum **higher**.

Wrong answer of second type (ε = -1): w0 x0 + w1 x1 + w2 x2 + …. + wn xn > 0 when

d(x1, x2, …., xn) = 0

i.e., w0 x0 + w1 x1 + w2 x2 + …. + wn xn is **too high**.

To correct for this, we need to make the sum **lower**.

Therefore, make w0 x0 + w1 x1 + w2 x2 + …. + wn xn **lower** or **higher** by adjusting weights.

- ε = 1: Make each contribution wi xi **higher**
- ε = -1: Make each contribution wi xi **lower**

In either case, can **add** some multiple η (called the "learning rate" - α) of ε.xi to wi to get the desired effect.

Add **εηxi** to wi to get the desired effect:

$(wi + \varepsilon \eta xi)\, xi = (wi\, xi + \varepsilon \eta\, xi^2)$

$> wi\, xi$ if $\varepsilon > 0$

$< wi\, xi$ if $\varepsilon < 0$

(ε = desired - actual, so in the first case need to bring actual up, in the second bring it down)

Learning Rate η

- η governs the rate at which the training rule converges toward the correct solution.
- Typically η < 1.
- Too small an η produces slow convergence.
- Too large of an η can cause oscillations in the process.

Example

Train a perceptron to classify according to:

(4, 5) 1
(6, 1) 1
(4, 1) 0
(1, 2) 0

There will be three weights (w0, w1, w2) where is the w0 threshold, corresponding to phantom input -1.

Start with "random" weights, say (0, +1, -1)

Choose η = 1.

weights	input	desired	actual	error	new weights
(0, 1, -1)	(-1, 4, 5)	1			
	(-1, 6, 1)	1			
	(-1, 4, 1)	0			
	(-1, 1, 2)	0			

weights	input	desired	actual	error	new weights
(0, 1, -1)	(-1, 4, 5)	1	0	1	(-1, 5, 4)
(-1, 5, 4)	(-1, 6, 1)	1	1	0	no change
(-1, 5, 4)	(-1, 4, 1)	0	1	-1	(0, 1, 3)
(0, 1, 3)	(-1, 1, 2)	0	1	-1	(1, 0, 1)

weights	input	desired	actual	error	new weights
(1, 0, 1)	(-1, 4, 5)	1			
	(-1, 6, 1)	1			
	(-1, 4, 1)	0			
	(-1, 1, 2)	0			

weights	input	desired	actual	error	new weights
(1, 0, 1)	(-1, 4, 5)	1	1	0	no change
(1, 0, 1)	(-1, 6, 1)	1	0	1	(0, 6, 2)
(0, 6, 2)	(-1, 4, 1)	0	1	-1	(1, 2, 1)
(1, 2, 1)	(-1, 1, 2)	0	1	-1	(2, 1, -1)

weights	input	desired	actual	error	new weights
(2, 1, -1)	(-1, 4, 5)	1	0	1	(1, 5, 4)
(1, 5, 4)	(-1, 6, 1)	1	1	0	no change
(1, 5, 4)	(-1, 4, 1)	0	1	-1	(2, 1, 3)
(2, 1, 3)	(-1, 1, 2)	0	1	-1	(3, 0, 1)

weights	input	desired	actual	error	new weights
(3, 0, 1)	(-1, 4, 5)	1	1	0	no change
(3, 0, 1)	(-1, 6, 1)	1	0	1	(2, 6, 2)
(2, 6, 2)	(-1, 4, 1)	0	1	-1	(3, 2, 1)
(3, 2, 1)	(-1, 1, 2)	0	1	-1	(4, 1, -1)

weights	input	desired	actual	error	new weights
(4, 1, -1)	(-1, 4, 5)	1	0	1	(3, 5, 4)
(3, 5, 4)	(-1, 6, 1)	1	1	0	no change
(3, 5, 4)	(-1, 4, 1)	0	1	-1	(4, 1, 3)
(4, 1, 3)	(-1, 1, 2)	0	1	-1	(5, 0, 1)

weights	input	desired	actual	error	new weights
(5, 0, 1)	(-1, 4, 5)	1	0	1	(4, 4, 6)
(4, 4, 6)	(-1, 6, 1)	1	1	0	no change
(4, 4, 6)	(-1, 4, 1)	0	1	-1	(5, 0, 5)
(5, 0, 5)	(-1, 1, 2)	0	1	-1	(6, -1, 3)

weights	input	desired	actual	error	new weights
(6, -1, 3)	(-1, 4, 5)	1	1	0	no change
(6, -1, 3)	(-1, 6, 1)	1	0	1	(5, 5, 4)
(5, 5, 4)	(-1, 4, 1)	0	1	-1	(6, 1, 3)
(6, 1, 3)	(-1, 1, 2)	0	1	-1	(7, 0, 1)

weights	input	desired	actual	error	new weights
(7, 0, 1)	(-1, 4, 5)	1	0	1	(6, 4, 6)
(6, 4, 6)	(-1, 6, 1)	1	1	0	no change
(6, 4, 6)	(-1, 4, 1)	0	1	-1	(7, 0, 5)
(7, 0, 5)	(-1, 1, 2)	0	1	-1	(8, -1, 3)

weights	input	desired	actual	error	new weights
(8, -1, 3)	(-1, 4, 5)	1	1	0	no change
(8, -1, 3)	(-1, 6, 1)	1	0	1	(7, 5, 2)
(7, 5, 2)	(-1, 4, 1)	0	1	-1	(8, 1, 3)
(8, 1, 3)	(-1, 1, 2)	0	0	0	(8, 1, 3)

weights	input	desired	actual	error	new weights
(8, 1, 3)	(-1, 4, 5)	1	1	0	no change
(8, 1, 3)	(-1, 6, 1)	1	1	0	no change
(8, 1, 3)	(-1, 4, 1)	0	0	0	no change
(8, 1, 3)	(-1, 1, 2)	0	0	0	no change

We found that the perceptron with weights (8, 1, 3) correctly classifies all inputs.

The "yes" criterion is therefore:

$$-8 + x1 + 3\,x2 > 0 \text{ [i.e. } x1 + 3\,x2 > 8\text{]}$$

Check:

input x1, x2	desired	actual
(4, 5)	1	1
(6, 1)	1	1
(4, 1)	0	0
(1, 2)	0	0

Perceptron Training Algorithm

Inputs:

- A list of training samples, each of the form
  - [d(x1, x2 , . . ., xn), -1, x1, x2 , . . ., xn]
  - (d is the desired output, -1 the phantom input)
- An initial weight vector [w0, w1, w2 , . . ., wn]
  - (w0 is the threshold)
- A learning rate η

Outputs:

- If the set of samples is linearly separable, a vector of weights [w0, w1, w2 , . . ., wn] such that with these weights the perception properly separates the training samples.
- If the set of samples is not linearly separable, then the algorithm diverges.

Operation:

- Set [w0, w1, w2 , . . ., wn] = initial weights;
- **while**(there is a sample not correctly classified)
  - Let [d(x1, x2 , . . ., xn), -1, x1, x2 , . . ., xn] be an incorrectly classified sample.
  - Let ε = d(x1, x2 , . . ., xn) - a(x1, x2 , . . ., xn), where a(x1, x2 , . . ., xn) = 1 if (Σ wi xi > 0), 0 otherwise.
  - Vector-add to [w0, w1, w2 , . . ., wn] the vector Δw = ε η [-1, x1, x2 , . . ., xn] - **perceptron learning rule**

Perceptron Training Algorithm Modifications for practical usage

- Put a **limit** on the number of iterations, so that the algorithm will terminate (without perfect classification) even if the sample set is not linearly separable.
- Include an **error bound** as an extra input. The algorithm can stop as soon as the portion of mis-classified samples is less than this bound (as opposed to requiring perfect classification, which would be an error bound of 0).
- Generate the initial weights **randomly**, so that the user does not have to specify them.

Correctness of the Perceptron Training Algorithm

Some Simplifications

- All training vectors x (including the phantom -1) can be **normalized**, by dividing by the length ||x||, so that ||x|| = 1.
- This is because only the **sign** of wx matters in classification, and the sign of wx is the same as that of wx/||x||.
- The weight vector w can also be normalized, so that ||w|| = 1, by the same rationale.
- All training samples can be assumed to have positive desired value.
- If the desired value were to be negative, it can be made positive just be complementing all of the components.

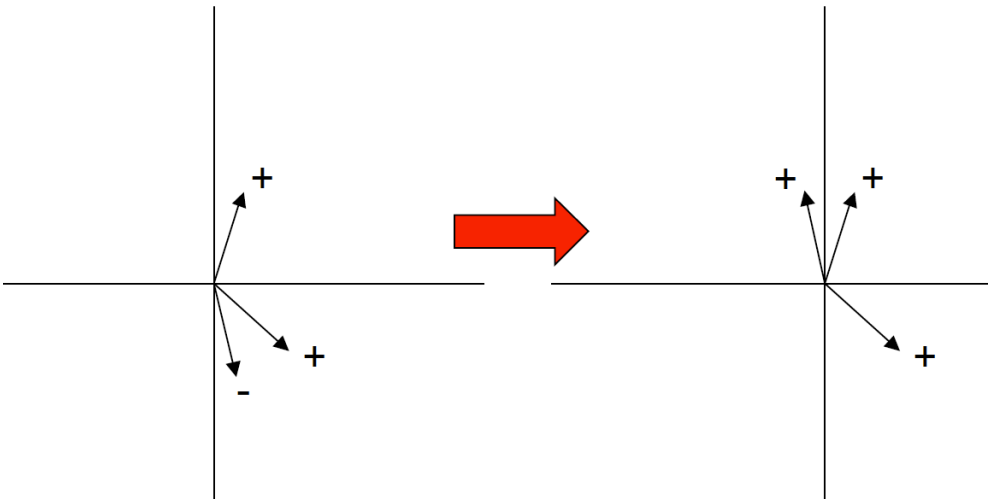


Fig. 20 : Complementing the sign

Visualization

- For normalized w and x, the value wx is the **cosine** of the angle ρ between w and x.
- We want to find a weight vector which has a positive value of cos ρ with each x.
- A training step consists of adding vector Δw (proprtional to x) to w, to push w away if cos ρ < 0.

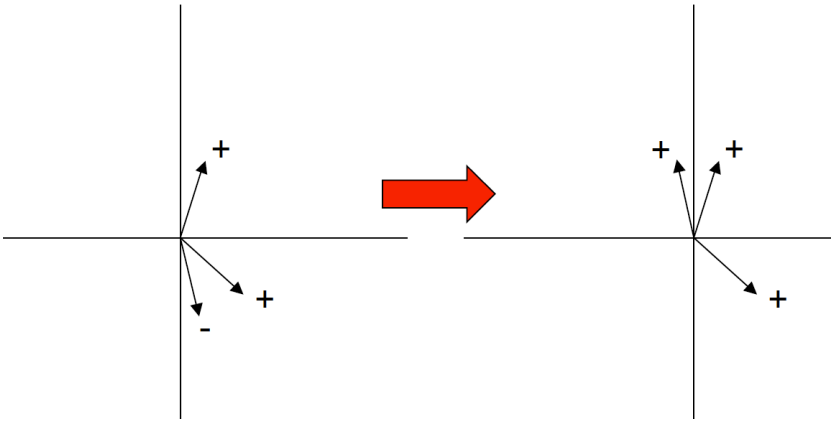


Fig. 21 : Learning

Initialization Heuristic

If all inputs are positive and normalized, average them to get the initial weight value.

Convergence Proof

- Consider all samples are positive and normalized.
- Consider the weights are normalized.
- **Claim**: If a weight vector w* exists which correctly classifies all samples, one will be found by the perceptron training algorithm.

Proof (following Rojas, pp 88-89)

- Assume w* exists (w* classifies correctly).
- Let wt be the weight vector after t steps of the algorithm.
- Consider step t+1, which resulted from some vector xi being misclassified by wt.
- Thus wt+1 = wt + xi.

- Consider ρ where cos ρ = w* wt+1 / || wt+1 ||, which must be ≤1.
- In the **numerator**,

  w* wt+1 = w*(wt + xi).

  = w*wt + w*xi.

≥ w*wt + δ,
where δ = min{w*xj | samples xj } > 0, by strict classification.

- By inductive substitution, w* wt+1 ≥ w*w0 + δ(t+1).

- Again, w* wt+1 / || wt+1 || ≤ 1.
- Squaring the **denominator**,
  || wt+1 ||² = (wt + xi) (wt + xi)
  = ||wt ||² + 2wtxi + ||xi ||²
- But wtxi < 0, because xi was misclassified, so
  || wt+1 ||² ≤ ||wt ||² + ||xi ||²
  ≤ ||wt ||² + 1, since xi are normalized.
- By inductive substitution,
  || wt+1 ||² ≥ ||w0 ||² + (t + 1)

- From two inequalities derived

$$1 \geq \frac{(w^* w_0 + \delta(t+1))}{\text{sqrt}(||w_0||^2 + (t+1))}$$

- The RHS grows with t as sqrt(t).
- Since since δ > 0, that growth must be bounded.
- Hence there is a **maximum** t for which classification is incorrect, i.e. the algorithm terminates.

Convergence

At the beginning, the perceptron will have some amount of error. Through out the training, the error (difference between target value and output of the perceptron) will be reduced. Such reduction in error value is known as **convergence**. The rate of convergence is determined by **learning rate -** η.

Smaller the learning rate, smoother the convergence but slower training. Larger value of learning rate results into **saw-tooth effect**.

Accelerating convergence

- Instead of using the learning rate to control weight increments, add "just enough" weight to make the incorrectly-classified sample be classified correctly.
- Instead of:
  Δw = ε η [-1, x1, x2 , . . ., xn]

  Use, Δw = α [-1, x1, x2 , . . ., xn]
  where α is such that

  (w + Δw) [-1, x1, x2 , . . ., xn] > 0.

  i.e. (w + α x) x > 0, while wx < 0.

- Want α x x > -wx, i.e. α > -wx / ||x||2
- Pick α = -wx / ||x||2 + β for some small β.

Multiple Categories

- If there are more than 2 categories, the classification strategy must be extended

somehow.

- Two ideas that don't work perfectly are:
  - Have a separate {yes, no} for each category. [Could have multiple yes answers, or none.]
  - Do pair-wise discrimination between all pairs of categories. [No guarantee of transitivity.]

Solution: Winner-take-all (WTA) Strategy

- Have a separate perceptron for each category.
- Rather than using the discrete output, use the weighted sum.
- Pick the category with the highest weighted sum (argmax).
- Break ties arbitrarily.

Uses

- Useful for dynamic systems for which mathematical characterization is unknown.
- Used for signal processing in adaptive filters.

Discussion of Retinal image classification

(Refer to : Rojas, Section 3.4 p.74 – p.83)

References

- Rojas, R., *Neural Networks – A Systematic Introduction*
- Anderson, J.A., *An Introduction to Neural Networks*
- Haykin, S., *Neural Networks – A Comprehensive Foundation*
- Keller, B., *Neural Networks, CS-152*

PRACTICAL # 2

Perceptrons adaptation
(June 7, 2009)

Objectives
To implement error-corrective learning for adaptation.

Description
Implement a perceptron and train it for adaptation using error-correction. The perceptron will be trained for 4 different train set of logic gate (AND/OR gate).

** This program shall be updated for the model described in page # 14 .

Platform
Any platform with any language of choice.

Model

The perceptron model as described.

Here's my program.

```
/*
 Program name: error corrective learning for finding association
 Description : This program demonstrates error corrective learning to establish
                        association  between inputs and logical output of logical gates
                        (AND, OR).

                        Since, the goal of the learning is to make association, activation
                        function is not used.
 By                      : Pramod Parajuli
 Date            : June 6, 2009
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define INPUTS                  2
#define TRAIN_SAMPLES    4

/* Let's write a function for uniform random number (0 - 1) generator. */
#define RANDOM_NUM (double) rand()/RAND_MAX


/* let's define a structure to represent the perceptron */
typedef struct perceptron
{
                double inp[INPUTS];                     // inputs to the perceptron
                double weights[INPUTS];          // weights of the inputs
                double outp;                            // output of the perceptron
                double error;                           // error of the perceptron
                double target;
}PERCEPTRON;

/* let's create a gate instance of PERCEPTRON*/
PERCEPTRON gate;

/* this structure defines template for one training set
            - 1 input vector
            - one expected output  */
typedef struct {
            double inputs[2];
            double expected_out;
} TRAIN_ELEMENT;

/* for logical gates with two inputs, let's define 4 training sets */
TRAIN_ELEMENT training_set[TRAIN_SAMPLES] = {
```

```
            {{0.0, 0.0}, 0.0},
            {{0.0, 1.0}, 1.0},
            {{1.0, 0.0}, 1.0},
            {{1.0, 1.0}, 1.0}
};

long counter;                   // general purpose counter

double LEARNING_RATE;           // learning rate of the perceptron
long EPOCH;                             // defines no. of iterations

void menu(void);                // displays a user help menu
void randomize(void);           // assigns random weights to the weight vector
void print(void);                       // print the weight vector
void train(void);                       // train the perceptron for adaptation
void feedforward(void);         // feedforward function  (inputs * weights)
void backpropagate(void);       // backpropagate (weight += error * input * learning rate)
void test(void);                // test the perceptron by using feedforward

int main (int argc, char * argv[]){
            int exit;                           // status variable for flow control
            char user_input[10];        // user input - string

            menu();                                         // display menu

            exit = 0;

            do{
               printf("\ncommand $  ");
               gets(user_input);
               //fflush(stdin);

               if(strcmp(user_input, "SHUFFLE") == 0){
                        srand(time(NULL));              // seed the random number generator
                        randomize();                            // assign random weights
                        print();                                // print the weight vector
               }
               else if(strcmp(user_input, "TRAIN") == 0){
                        printf("\nEnter the epoch : ");                 // epoch
                        scanf("%ld", &EPOCH);
                        printf("\nEnter the learning rate : ");  // learning rate
                        scanf("%lf", &LEARNING_RATE);

                        train();                                                                // train
               }
               else if(strcmp(user_input, "TEST") == 0){
                        test();
               }
               else if(strcmp(user_input, "PRINT") == 0){
                        print();
               }
               else if(strcmp(user_input, "EXIT") == 0){
                        exit = 1;
               }
               else{
                        menu();
               }

            }while(exit == 0);

            printf("\npress any key to exit...");
            getc(stdin);

            return 0;
}

/*---------------------------------------------------------------------
 Function name   : menu
 Description : Displays menu
---------------------------------------------------------------------*/
void menu(void){
            printf("\n\n\t\t\tPerceptron training");
            printf("\nPlease use following commands");
            printf("\n-------------------------------------------------------------------");
            printf("\nshuffle : to randomize weights   | train   : to train the perceptron");
            printf("\ntest    : to test the perceptron | print   : to print the weights");
```

```
            printf("\nhelp    : to display this menu   | exit    : to exit the program");
            printf("\n-------------------------------------------------------------------------\n");
}

/*-------------------------------------------------------------------------
 Function name   : randomize weights
 Description : randomizes the weights to new values
 -------------------------------------------------------------------------*/
void randomize(void){
            for(counter = 0; counter < INPUTS; counter++){
               gate.weights[counter] = RANDOM_NUM;
            }
}

/*-------------------------------------------------------------------------
 Function name   : print
 Description : prints the weights
 -------------------------------------------------------------------------*/
void print(void){
            for(counter = 0; counter < INPUTS; counter++){
               printf("\nWeight [%ld] : %0.30lf", counter, gate.weights[counter]);
            }
}

/*-------------------------------------------------------------------------
 Function name   : feedforward
 Description : feedforward the inputs by multiplying input vector and weight
                              vector
 -------------------------------------------------------------------------*/
void feedforward(void){
            double sum = 0.0;
            for(counter = 0; counter < INPUTS; counter++){
               sum += gate.inp[counter] * gate.weights[counter];
            }
            gate.outp = sum;
}

/*-------------------------------------------------------------------------
 Function name   : backpropagate
 Description : backpropagate by finding the error and update the weight
                              vector
 -------------------------------------------------------------------------*/
void backpropagate(void){
            gate.error = gate.target - gate.outp;

            for(counter = 0; counter < INPUTS; counter++){
               gate.weights[counter] += gate.error * LEARNING_RATE * gate.inp[counter];
            }
}

/*-------------------------------------------------------------------------
 Function name   : train
 Description : trains the perceptron using following algo.
            1. Select the first train set
            2. Feed-forward the inputs
            3. Back-propagate
            4. Select next train set
            5. Repeat till good level of convergence is achieved
-------------------------------------------------------------------------*/
void train(void){
            int samples;                              // keeps track of training sample
            long iterations = 0;                // training iterations
            //double MSE = 1.0;
            samples = 0;

            while(iterations < EPOCH){  // train for EPOCH no. of times

               // select the train set
               if(++samples == TRAIN_SAMPLES){
                        samples = 0;
               }

               /* set the input of perceptron to be inputs of current train set */
               for(counter = 0; counter < INPUTS; counter++){
                        gate.inp[counter] = training_set[samples].inputs[counter];
               }
```

```
                /* set the target of perceptron to be target of current train set */
                gate.target = training_set[samples].expected_out;

                /* just for debugging */
                printf("\nCurrent inputs %lf, %lf, current target %lf",
                           gate.inp[0], gate.inp[1], gate.target);

                feedforward();                  // feed forward

                // MSE = (gate.target - gate.outp) * (gate.target - gate.outp);
                // printf("\nCurrent MSE: %0.30lf", MSE);

                backpropagate();        // backpropagate

                iterations++;                   // go for next iteration
            }
}

/*-------------------------------------------------------------------------
 Function name   : test
 Description : test the perceptron by using feedforward
-------------------------------------------------------------------------*/
void test(void){
            long input;

            /* take user values for inputs */
            for(counter = 0; counter < INPUTS; counter++){
                printf("\nEnter input (%ld) : ", counter);
                scanf("%ld", &input);
                gate.inp[counter] = input;
            }

            /* feed forward */

            feedforward();

            printf("\nThe output of the perceptron : %0.30lf", gate.outp);

}
```

Learning Algorithms

Learning algorithms can be divided into **supervised** and **unsupervised** methods. **Supervised learning** denotes a method in which some input vectors are collected and presented to the network. The output computed by the network is observed and the deviation from the expected answer is measured. The weights are corrected according to the magnitude of the error in the way defined by the learning algorithm. This kind of learning is also called learning with a teacher, since a control process knows the correct answer for the set of selected input vectors.

Unsupervised learning is used when, for a given input, the exact numerical output a network should produce is unknown. Assume, for example, that some points in two-dimensional space are to be classified into three clusters. For this task we can use a classifier network with three output lines, one for each class (see following figure). Each of the three computing units at the output must specialize by firing only for inputs corresponding to elements of each cluster. If one unit fires, the others must keep silent. In this case we do not know a priori which unit is going to specialize on which cluster. Generally we do not even know how many well-defined clusters are present. Since no "teacher" is available, the network must organize itself in order to be able to associate clusters with units.

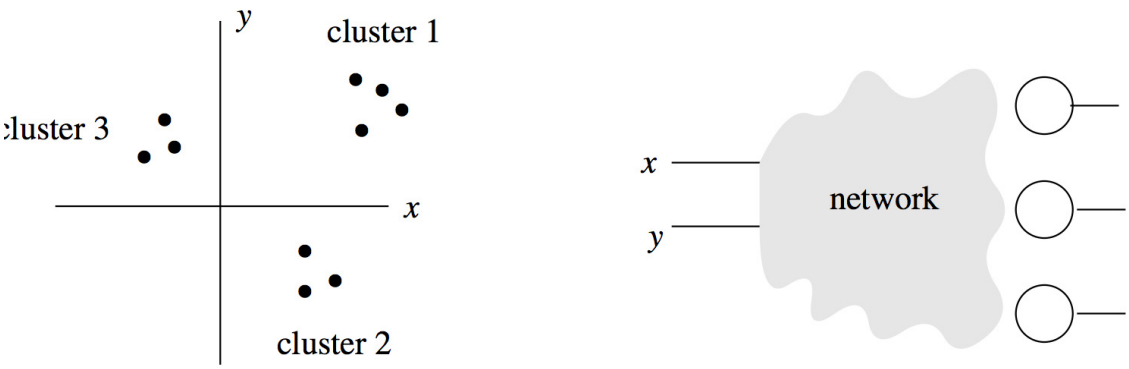


Fig. 22: Clusters and classifiers

Supervised learning is further divided into methods that use **reinforcement** or **error correction**. Reinforcement learning is used when after each presentation of an input-output example we only know whether the network produces the desired result or not. The weights are updated based on this information (that is, the Boolean values true or false) so that only the input vector can be used for weight correction. In learning with error correction, the magnitude of the error, together with the input vector, determines the magnitude of the corrections to the weights, and in many cases we try to eliminate the error in a single correction step.

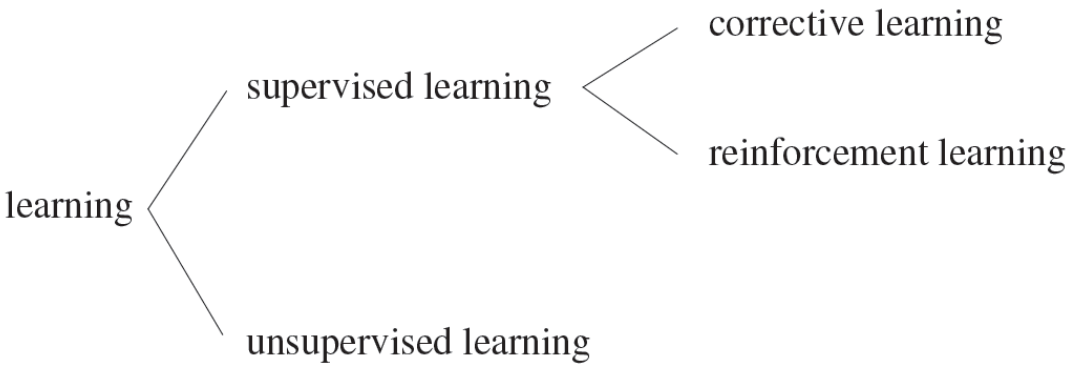


Fig. 23: Classes of learning algorithms

Vector notation

In the following sections we deal with learning methods for perceptrons. To simplify the notation we adopt the following conventions. The input (x1, x2, . . . , xn) to the perceptron is called the input vector. If the weights of the perceptron are the real numbers w1,w2, . . . ,wn and the threshold is _, we call w = (w1,w2, . . . ,wn,wn+1) with wn+1 = $-\theta$ the extended weight vector of the perceptron and (x1, x2, . . . , xn, 1) the extended input vector. The threshold computation of a perceptron will be expressed using scalar products. The arithmetic test computed by the perceptron is thus

$$w.x >= \theta$$

if w and x are the weight and input vectors, and

$$w.x >= 0$$

Algorithmic learning

The training set consists of two sets, P and N, in n-dimensional extended input space. We look for a vector w capable of absolutely separating both sets, so that all vectors in P belong to the open positive half-space and all vectors in N to the open negative half-space of the linear separation.

```
start:      The weight vector w0 is generated randomly,
            set t := 0

test:       A vector x ∈ P U N is selected randomly,
            if x 2 P and wt · x > 0 go to test,
            if x 2 P and wt · x _ 0 go to add,
            if x 2 N and wt · x < 0 go to test,
            if x 2 N and wt · x _ 0 go to subtract.

add:        set wt+1 = wt + x and t := t + 1, goto test

subtract:   set wt+1 = wt − x and t := t + 1, goto test
```

This algorithm makes a correction to the weight vector whenever one of the selected vectors in P or N has not been classified correctly. The perceptron convergence theorem guarantees that if the two sets P and N are linearly separable the vector w is updated only a finite number of times. The routine can be stopped when all vectors are classified correctly. The corresponding test must be introduced in the above pseudocode to make it stop and to transform it into a fully-fledged algorithm.

References

- Rojas, R., *Neural Networks – A Systematic Introduction*