Nonlinear Classifiers I

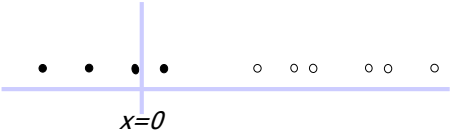
Nonlinear Classifiers: Introduction

- Classifiers
  - Supervised Classifiers
    - Linear Classifiers
      - Perceptron
      - Least Squares Methods
      - Linear Support Vector Machine
    - **Nonlinear Classifiers**
      - **Part I: Multi Layer Neural Networks, Convolutional Neural Network**
      - Part II: Nonlinear Support Vector Machine
    - Decision Trees
  - Unsupervised Classifiers

Nonlinear Classifiers: Introduction

- An example: Suppose we're in 1-dimension

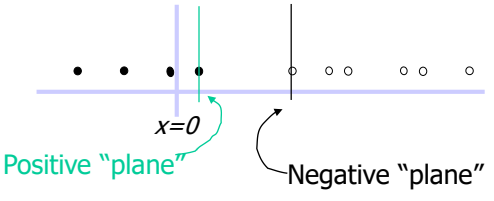
What would a linear SVMs do with this data?



Nonlinear Classifiers: Introduction

- An example: Suppose we're in 1-dimension

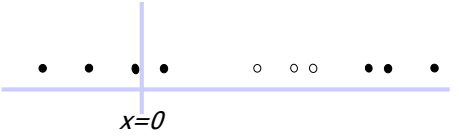
Not a big surprise



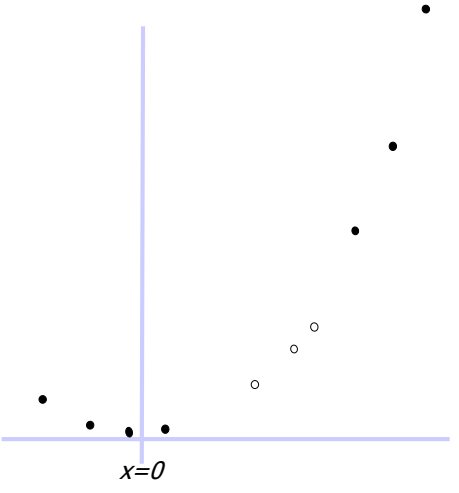
Nonlinear Classifiers: Introduction

- Harder 1-dimensional dataset

What can be done about this?



Nonlinear Classifiers: Introduction

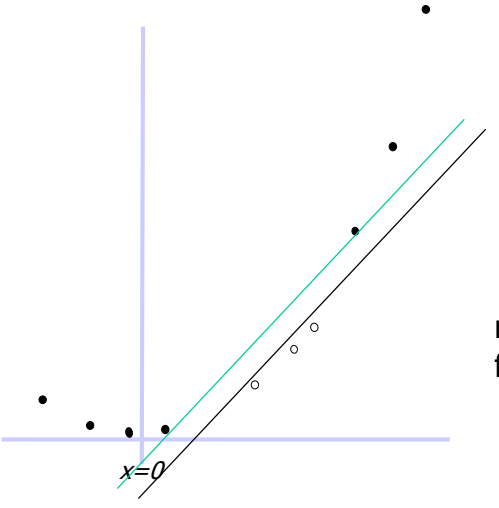


non-linear basis function

$$\mathbf{z}_k = (x_k, x_k^2)$$

Nonlinear Classifiers: Introduction

7

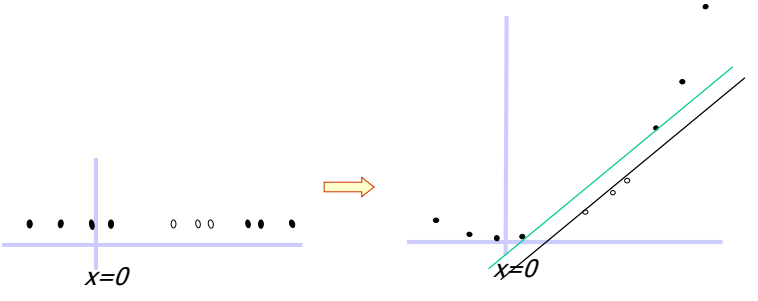


non-linear basis function

$$\mathbf{z}_k = (x_k, x_k^2)$$

Nonlinear Classifiers: Introduction

8



- **Linear classifiers** are simple and computationally efficient.
- However for nonlinearly separable features, they might lead to very inaccurate decisions.
- Then we may trade simplicity and efficiency for accuracy using a **nonlinear classifier**.

The Perceptron

The Perceptron is a learning algorithm that *adjusts the weights* w_i of its weight vector $\underline{w}$ such that for all examples $\underline{x}_i$:

$$\underline{w}^T \underline{x} > 0 \quad \forall \underline{x}_i \in \omega_1$$
$$\underline{w}^T \underline{x} < 0 \quad \forall \underline{x}_i \in \omega_2$$

It is assumed that the problem is linearly separable. Hence this vector $\underline{w}$ exists.

Here, the intercept is included in w : $\underline{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_l \\ w_0 \end{bmatrix}$ $\underline{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_l \\ 1 \end{bmatrix}$

$$\Rightarrow \quad g(\underline{x}) \equiv \underline{w}^T \underline{x} = 0$$

The Perceptron

- $\underline{w}$ must minimize the classification error.
- $\underline{w}$ is found using an *optimization algorithm*.

General steps towards a classifier:

1. Define a cost function to be minimized.
2. Choose an algorithm to minimize it.
3. The minimum corresponds to a solution.

The Perceptron Cost Function

Goal: $\underline{w}^T \underline{x} > 0 \quad \forall \underline{x} \in \omega_1$

$\underline{w}^T \underline{x} < 0 \quad \forall \underline{x} \in \omega_2$

Cost function: $J(\underline{w}) = \sum_{x \in Y} \delta_x \underline{w}^T \underline{x}$

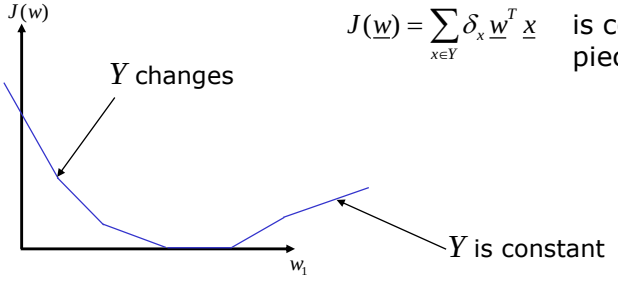
Y: subset of the training vectors which are **misclassified** by the hyperplane defined by w.

$\delta_{x_i} = -1$ if $\underline{x}_i \in \omega_1$ but is classified in ω_2

$\delta_{x_i} = +1$ if $\underline{x}_i \in \omega_2$ but is classified in ω_1

$$\Rightarrow \delta_x \underline{w}^T \underline{x} > 0 \quad \forall \underline{x} \in Y \Rightarrow \begin{array}{ll} J(\underline{w}) > 0 & \forall \underline{w} : Y \neq \varnothing \\ J(\underline{w}) = 0 & \text{if } Y = \varnothing \end{array}$$

The Perceptron Algorithm



$J(\underline{w}) = \sum_{x \in Y} \delta_x \underline{w}^T \underline{x}$ is continuous and piecewise linear.

$J(\underline{w})$ is minimized by *gradient descent:*

(update w by taking steps that are proportional to the negative of the gradient of the cost function J(w))

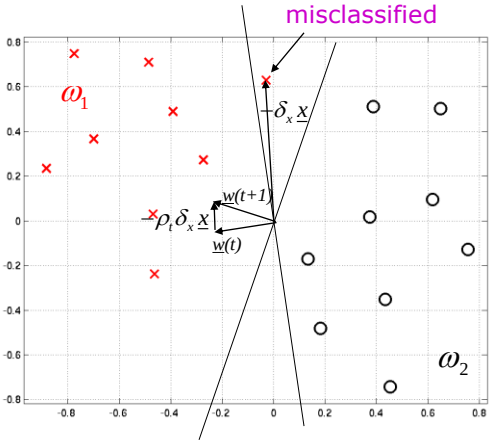
$$\underline{w}(t+1) = \underline{w}(t) + \Delta \underline{w} \quad \Rightarrow \quad \Delta \underline{w} = -\rho_t \frac{\partial J(\underline{w})}{\partial \underline{w}}$$

$$\frac{\partial J(\underline{w})}{\partial \underline{w}} = \frac{\partial}{\partial \underline{w}} \left(\sum_{x \in Y} \delta_x \underline{w}^T \underline{x} \right) = \sum_{x \in Y} \delta_x \underline{x}$$

$$\underline{w}(t+1) = \underline{w}(t) - \rho_t \sum_{x \in Y} \delta_x \underline{x}$$

The Perceptron Algorithm

Example:



$$\underline{w}(t+1) = \underline{w}(t) - \rho_t \sum_{x \in Y} \delta_x \underline{x}$$

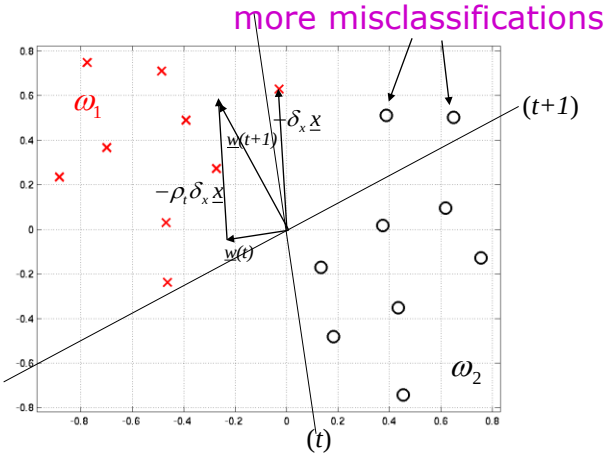
Here, δ_x =-1 because $\underline{x} \in \omega_1$

Here, ρ_t =0.2

The Perceptron Algorithm

Example:

Here, ρ_t =1



Note that ρ_t must be chosen carefully, if it is too large, more errors will occur.

ρ_t is a critical parameter of the algorithm !

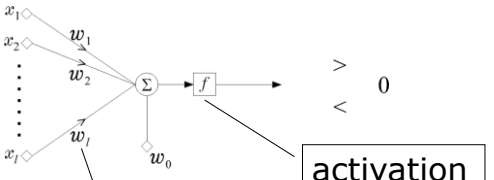
The Perceptron as a Neural Network

Once the perceptron is trained, it is used to perform the classification:

$$\text{if } w^T x > 0 \quad \text{assign } x \text{ to } \omega_1$$

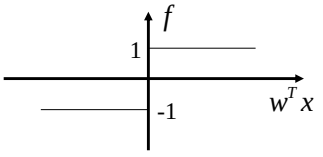
$$\text{if } w^T x < 0 \quad \text{assign } x \text{ to } \omega_2$$

The perceptron is the simplest form of a "Neural Network":

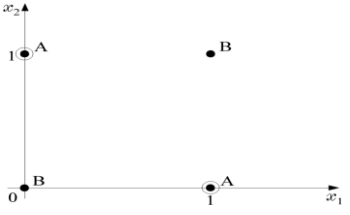


activation function

synaptic weights

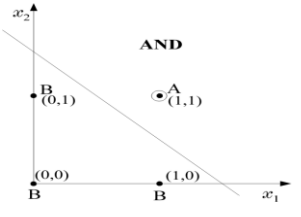


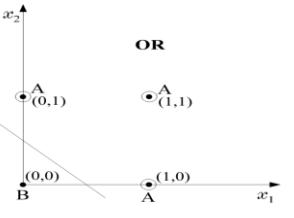
The XOR problem



x_1	x_2	XOR	Class
0	0	0	B
0	1	1	A
1	0	1	A
1	1	0	B

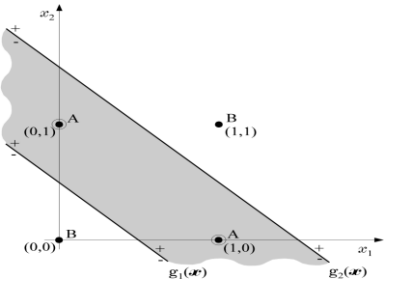
- There is no single line (hyperplane) that separates class A from class B. On the contrary, AND and OR operations are linearly separable problems.





The Two-Layer Perceptron

- For the XOR problem, draw two lines, instead of one.
- Then class B is outside the shaded area and class is A inside.
- Was called: two-step design.

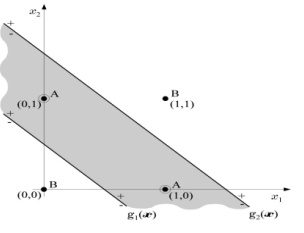


The Two-Layer Perceptron

- Step 1: Draw two lines (hyperplanes)

$$g_1(\underline{x}) = 0,$$

$$g_2(\underline{x}) = 0$$



Each of them is realized by a perceptron.
The outputs of the perceptrons will be

$$y_i = f(\, g_i(\underline{x})\,) \;=\; \begin{cases} 0 \\ 1 \end{cases} \qquad i = 1,\ 2$$

depending on the value of $\underline{x}$ (f is the activation function).

- Step 2: Find the 'position' of $\underline{x}$ *w.r.t.* both lines, based on the values of y_1, y_2.

The Two-Layer Perceptron

- Equivalently:

  1. The computations of the first step perform a mapping

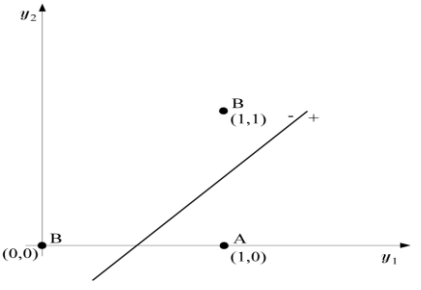
     $$\underline{x} \rightarrow \underline{y} = [y_1, y_2]^T$$

  2. The decision is then performed on the transformed data $\underline{y}$.

1st step				2nd step
x_1	x_2	y_1	y_2	
0	0	0(-)	0(-)	B(0)
0	1	1(+)	0(-)	A(1)
1	0	1(+)	0(-)	A(1)
1	1	1(+)	1(+)	B(0)

The Two-Layer Perceptron

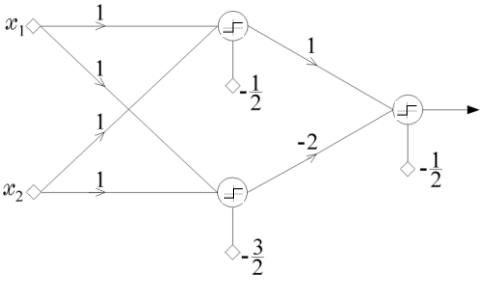
- This decision can be performed via a second line $g(\underline{y}) = 0$, which can also be realized by a perceptron.



➢ Computations of the first step perform a mapping that transforms the nonlinearly separable problem to a linearly separable one.

The Two-Layer Perceptron

- The architecture



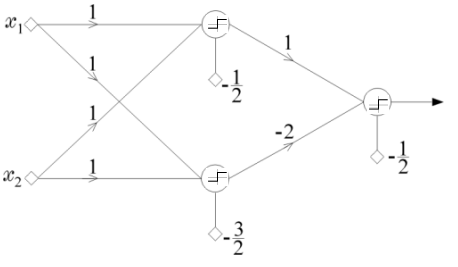
- This is known as the two layer perceptron with one hidden and one output layer.
  The activation functions are:

$$f(.)=\begin{cases}1 & x\geq 0\\ 0 & x<0\end{cases}$$

The Two-Layer Perceptron

- The nodes (neurons) of the figure realize the following lines (hyper planes).

$$g_1(\underline{x})=1x_1+1x_2-\frac{1}{2}=0$$
$$g_2(\underline{x})=1x_1+1x_2-\frac{3}{2}=0$$
$$g_3(\underline{y})=1y_1-2y_2-\frac{1}{2}=0$$
$$f(.)=\begin{cases}1 & x\geq 0\\ 0 & x<0\end{cases}$$



- Classification capabilities:
  All possible mappings performed by the first layer are onto the vertices of the unit side square,
  e.g., (0, 0), (1, 0), (1, 0), (1, 1).

Classification capabilities

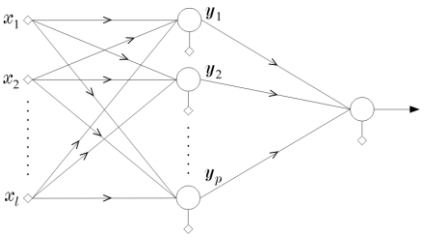
- The more general case

$\underline{x} \in R^l,$

$y_i \in \{0,\ 1\} \qquad i = 1,\ 2, \ldots p$

$\underline{x} \rightarrow \underline{y} = [y_1, \ldots y_p]^T, \quad \underline{y} \in R^p$

$y_i = f(g_i)$



$$g_i(\underline{x}) = \sum_{k=1}^{l} w_{ik} x_k + w_{i0} = 0$$

$$g_i(\underline{x}) = \underline{w}_i^T \underline{x} + w_{i0} = 0 \qquad \underline{w}_i,\ \underline{x} \in R^l$$

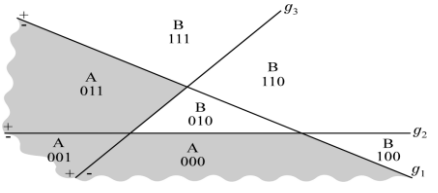
$$g_j(\underline{y}) = \sum_{k=1}^{p} w_{jk} y_k + w_{j0} = 0$$

$$g_j(\underline{y}) = \underline{w}_j^T \underline{y} + w_{j0} = 0 \qquad \underline{w}_j,\ \underline{y} \in R^p$$

- $g_i(x)$'s perform a mapping of a vector x onto y representing the vertices of the unit side $\boldsymbol{H_p}$ hypercube.

Classification capabilities

- The mapping is achieved with p nodes each realizing a hyperplane. The output of each of these nodes is 0 or 1 depending on the relative position of $\underline{x}$ w.r.t. the hyperplane.

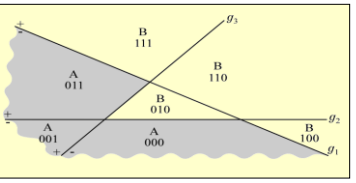


Intersections of these hyperplanes form regions in the l-dimensional space. Each region corresponds to a vertex of the H_p unit hypercube.

Classification capabilities

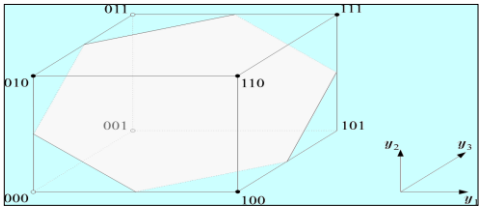
For example, the 001 vertex corresponds to the region which is located

to the (-) side of $g_1(\underline{x}) = 0$
to the (-) side of $g_2(\underline{x}) = 0$
to the (+) side of $g_3(\underline{x}) = 0$

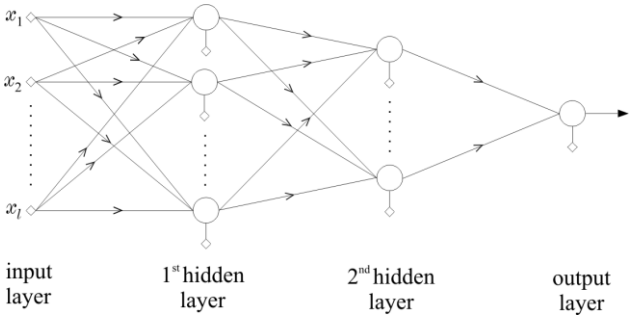


The **output** node realizes a hyperplane in the $\underline{y}$ space, that separates some of the vertices from the others. Thus, the two layer perceptron has the capability to classify vectors into **classes that consist of unions of polyhedral regions**.

But <u>not ANY</u> union. It depends on the relative position of the corresponding vertices.



The Three-Layer Perceptron



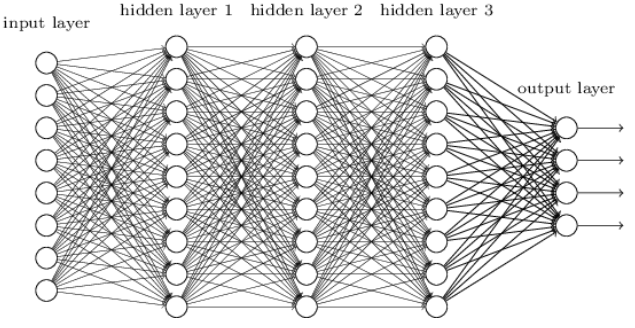
- This is capable to classify vectors into classes consisting of <u>ANY</u> union of polyhedral regions.
- The idea is similar to the XOR problem. It realizes more than one plane in the space.

- The reasoning
  - For each vertex, corresponding to class A, construct a hyperplane which leaves THIS vertex on one side (+) and ALL the others to the other side (-).
  - The output neuron realizes an OR gate.
- Overall:

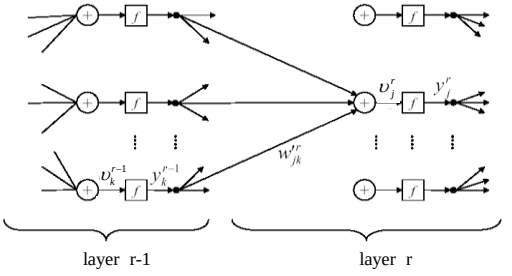
The first layer of the network forms the hyperplanes, the second layer forms the regions and the output nodes forms the classes.

Multi-Layer Neural Networks

- Many parameter
  - Number of layers
  - Number of nodes in each layer.
  - Number of connections of each node with previous the layer (e.g. fully connected)



The Multi-Layer Neural Network



for the i-th trainings pair

$y_k^{r-1}(i)$ output of the k-th node at layer r-1

$\upsilon_j^r(i)$ argument for $f(.)$ for the i-th trainings pair layer r

$$\upsilon_j^r(i) = \sum_{k=1}^{k_{r-1}} w_{jk}^r y_k^{r-1}(i) + w_{j0}^r \equiv \sum_{k=0}^{k_{r-1}} w_{jk}^r y_k^{r-1}(i), \quad with \; y_0^r(i) \equiv +1$$

$$\upsilon_j^r(i) = \underline{w}_j^r \underline{y}^{r-1}(i) \qquad y_j^r(i) = f\left(\upsilon_j^r(i)\right) = f\left(\underline{w}_j^r \underline{y}^{r-1}(i)\right)$$

Multi-Layer Neural Networks

- Designing Multilayer Networks
  - One strategy could be to adopt the above rationale and construct a structure that classifies correctly all the training patterns. (usually impossible)
  - Second strategy: Start with a (large) network structure and compute the w's, often called 'synaptic weights', to optimize a cost function.
  - Back Propagation is an algorithmic procedure that computes the synaptic weights iteratively, so that an adopted cost function is minimized (optimized).

Nonlinear Classifiers: Agenda

Part I: Nonlinear Classifiers

Multi Layer Neural Networks

- XOR problem
- Two-Layer Perceptron
- **Backpropagation algorithm to train multilayer perceptrons**
- Choice of the network size
- Model selection techniques
- Applications: XOR, ZIP Code, OCR problem
- Demo: SNNS, BPN

The Backpropagation Algorithm (BP)

The Steps:

1. Adopt an optimizing cost function $J(i)$, e.g.,
   - Least Squares Error
   - Relative Entropy

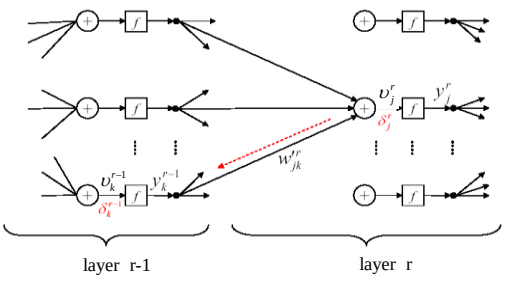
   between desired responses and actual responses of the network for the available training patterns.

➔ That is, from now on we have to live with errors resulting from structure and cost function. We only try to minimize them, using certain criteria.

The Backpropagation Algorithm

The Steps:

2. Adopt an algorithmic procedure for the optimization of the cost function with respect to the weights w e.g.:
   - Gradient descent
   - Newton's algorithm
   - Conjugate gradient



The Backpropagation Algorithm

The Steps:

3. The task is a **nonlinear** optimization e.g. with gradient descent.

$$\underline{w}_j^r(\text{new}) = \underline{w}_j^r(\text{old}) + \Delta\underline{w}_j^r$$

$$\Delta\underline{w}_j^r = -\mu\frac{\partial J}{\partial \underline{w}_j^r} \qquad J = \sum_{i=1}^{N} E(i)$$

BackProp: Step 3 nonlinear optimization

Detail: Computation of the Gradients.

$$\underline{w}_j^r(\text{new}) = \underline{w}_j^r(\text{old}) + \Delta\underline{w}_j^r$$

$$\Delta\underline{w}_j^r = -\mu\frac{\partial J}{\partial \underline{w}_j^r} \quad \text{with} \quad J = \sum_{i=1}^{N} E(i) \quad \text{and} \quad \upsilon_j^r(i) = \sum_{k=1}^{k_{r-1}} w_{jk}^r y_k^{r-1}(i) + w_{j0}^r$$

$$\frac{\partial E(i)}{\partial \underline{w}_j^r} = \frac{\partial E}{\partial \upsilon_j^r(i)}\frac{\partial \upsilon_j^r(i)}{\partial \underline{w}_j^r}$$

$$\frac{\partial}{\partial \underline{w}_j^r}\upsilon_j^r(i) = \begin{bmatrix} \frac{\partial}{\partial w_{j0}^r}\upsilon_j^r(i) \\ \vdots \\ \frac{\partial}{\partial w_{jk_{r-1}}^r}\upsilon_j^r(i) \end{bmatrix} = \underline{y}^{r-1}(i)$$

$$\frac{\partial E}{\partial \upsilon_j^r(i)} \equiv \delta_j^r(i)$$

$$\Leftrightarrow \quad \underline{y}^{r-1}(i) = \begin{bmatrix} +1 \\ y_1^{r-1}(i) \\ \vdots \\ y_{k_{r-1}}^{r-1}(i) \end{bmatrix}$$

$$\Delta\underline{w}_j^r = -\mu\sum_{i=1}^{N}\delta_j^r(i)\underline{y}^{r-1}(i)$$

BackProp: Step 3 nonlinear optimization

Detail: Computation of $\delta_j^r(i)$ for Least Squares

Case $r = L$ (**L**ast Layer)

$$\delta_j^r(i) = \frac{\partial E}{\partial \upsilon_j^r(i)} \quad \Rightarrow \quad \delta_j^L = e_j(i) f'\left(\upsilon_m^L(i)\right)$$

$$E(i) = \frac{1}{2}\sum_{m=1}^{k_L} e_m^2(i) = \frac{1}{2}\sum_{m=1}^{k_L}\left(f\left(\upsilon_m^L(i)\right) - \hat{y}_m(i)\right)^2$$

Case $r < L$

$$\frac{\partial E(i)}{\partial \upsilon_j^{r-1}(i)} = \sum_{k=1}^{k_r}\frac{\partial E(i)}{\partial \upsilon_k^r(i)}\frac{\partial \upsilon_k^r(i)}{\partial \upsilon_j^{r-1}(i)}$$

$$\delta_j^{r-1}(i) = \sum_{k=1}^{k_r}\delta_k^r(i)\frac{\partial \upsilon_k^r(i)}{\partial \upsilon_j^{r-1}(i)}$$

$$\delta_j^{r-1}(i) = \underbrace{\left[\sum_{k=1}^{k_r}\delta_k^r(i) w_{kj}^r\right]}_{e_j^{r-1}(i)} f'\left(\upsilon_j^{r-1}(i)\right)$$

$$\Leftrightarrow \quad \delta_j^{r-1}(i) = e_j^{r-1}(i) f'\left(\upsilon_j^{r-1}(i)\right)$$

$$\frac{\partial \upsilon_k^r(i)}{\partial \upsilon_j^{r-1}(i)} = \frac{\partial\left[\sum_{m=0}^{k_{r-1}} w_{km}^r y_m^{r-1}(i)\right]}{\partial \upsilon_j^{r-1}(i)}$$

with $\quad y_m^{r-1}(i) = f\left(\upsilon_m^{r-1}(i)\right)$

$$\Rightarrow \quad \frac{\partial \upsilon_k^r(i)}{\partial \upsilon_j^{r-1}(i)} = w_{kj}^r f'\left(\upsilon_j^{r-1}(i)\right)$$

BackProp: Step 3 summary

3. The task is a **nonlinear** optimization.

e.g. gradient descent.

$$\underline{w}_j^r(\text{new}) = \underline{w}_j^r(\text{old}) + \Delta\underline{w}_j^r \qquad \Delta\underline{w}_j^r = -\mu\frac{\partial J}{\partial w_j^r}$$

with the following up-date rules:

$$\Delta\underline{w}_j^r = -\mu\sum_{i=1}^{N}\delta_j^r(i)\,\underline{y}^{r-1}(i)$$

$$\delta_j^L(i) = e_j(i)\,f'\left(\upsilon_j^L(i)\right)$$

$$\delta_j^{r-1}(i) = \left[\sum_{k=1}^{k_r}\delta_k^r(i)\,w_{kj}^r\right]f'\left(\upsilon_j^{r-1}(i)\right)$$

Error $e_j(i)$: Difference of actual and desired response for the j-th output neuron

$$e_j(i) = \underset{\text{output}}{y_j^L(i)} - \underset{\text{target}}{\hat{y}_j(i)}$$

The Backpropagation Algorithm

The Procedure:

1. **Initialization:**
   Initialize unknown weights randomly with small values.
2. **Forward computations:**
   For each of the training examples compute the output of all neurons of all layers. Compute the cost function for the current estimate of weights.
3. **Backward computations:**
   Compute the gradient terms backwards, starting with the weights of the last (e.g. 3rd) layer and then moving towards the first.
4. **Update:** Update the weights.
5. **Termination:**
   Repeat until a termination procedure is met

The Backpropagation Algorithm

- In a large number of optimizing procedures, computation of derivatives are involved. Hence, discontinuous activation functions pose a problem, i.e.,

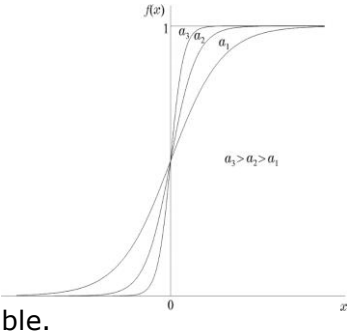
$$f(x) = \begin{cases} 1 & x > 0 \\ 0 & x < 0 \end{cases}$$

- There is always an escape path!!! e.g. the logistic function:

$$f(x) = \frac{1}{1+\exp(-ax)}$$

$$f'(x) = \alpha f(x)\big(1 - f(x)\big)$$

Other differentiable functions are also possible and in some cases more desirable.

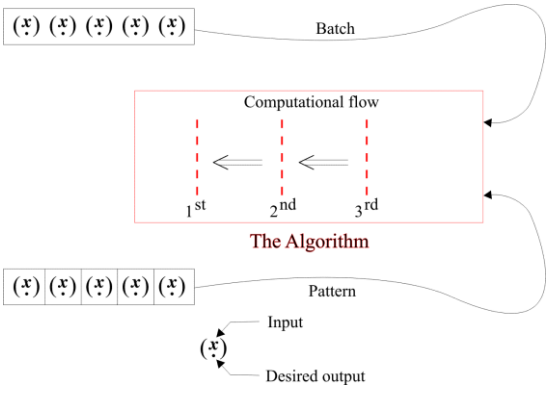


The Backpropagation Algorithm

Two major philosophies:

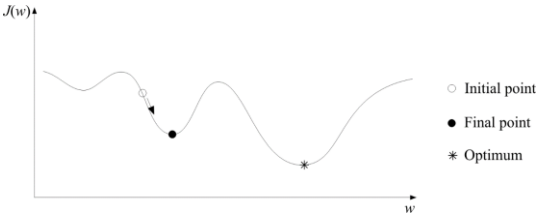
- Batch mode: The gradients of the last layer are computed once ALL training data have appeared to the algorithm, i.e., by summing up all error terms.

- Pattern mode: The gradients are computed every time a new training data pair appears. Thus gradients are based on successive individual errors.



The Backpropagation Algorithm

A major problem:
The algorithm may converge to a local minimum.



The cost function choice
Examples:

- The Least Squares

$$J = \sum_{i=1}^{N} E(i)$$

$$E(i) = \frac{1}{2}\sum_{m=1}^{k} e_m^2(i) = \frac{1}{2}\sum_{m=1}^{k} (y_m(i) - \hat{y}_m(i))^2 \qquad i = 1, 2, ..., N$$

$\hat{y}_m(i)$: Desired response of m-th output node (1 or 0) for input $x(i)$.

$y_m(i)$: Actual response of m-th output node, in the interval [0, 1], for input $x(i)$.

The Backpropagation Algorithm

The cost function choice
Examples:

- The cross-entropy

$$J = \sum_{i=1}^{N} E(i)$$

$$E(i) = \sum_{m=1}^{k} \{\ y_m(i) \ln \hat{y}_m(i) + (1 - y_m(i)) \ln(1 - \hat{y}_m(i))\ \}$$

This presupposes an interpretation of y and ŷ as probabilities!

Classification error rate:

- Also known as discriminative learning.
- Most of these techniques use a smoothed version of the classification error.

The Backpropagation Algorithm

"Well formed" cost functions :

- Danger of local minimum convergence.
- "Well formed" cost functions guarantee convergence to a "good" solution.
- That is one that classifies correctly ALL training patterns, provided such a solution exists.
- The cross-entropy cost function is a well formed one. The Least Squares is not.

The Backpropagation Algorithm

optimally class a-posteriori probabilities:

Both, the Least Squares and the cross entropy lead to output values $\hat{y}_m(i)$ that approximate optimally class a-posteriori probabilities!

$$\hat{y}_m(i) \cong P(\omega_m | \underline{x}(i))$$

That is, the probability of class ω_m given $\underline{x}(i)$.

➔ It **does not** depend on the underlying distributions!!! It is a characteristic of **certain cost functions** and the **chosen architecture** of the network. It depends on the model how good or bad the approximation is.

➔ It is valid at the global minimum.

Nonlinear Classifiers: Agenda

Part I: Nonlinear Classifiers

Multi Layer Neural Networks

- XOR problem
- Two-Layer Perceptron
- Backpropagation
- **Choice of the network size**
  - Number of layers and of neurons per layer
  - Model selection techniques
    - Pruning techniques
    - Constructive techniques
- Applications: XOR, ZIP Code, OCR problem
- Demo: SNNS, BPN

Choice of the network size

How big a network can be. How many layers and how many neurons per layer?
There are two major techniques:

- **Pruning Techniques:**
  These techniques start from a large network and then weights and/or neurons are removed iteratively, according to a criterion.

- **Constructive techniques:**
  They start with a small network and keep increasing it, according to a predetermined procedure and criterion.

Choice of the network size

Idea: Start with a large network and leave the algorithm to decide which weights are small.

Generalization properties:

- Large network learn the particular details of the training set.
- Not be able to perform well when presented with data unknown to it.

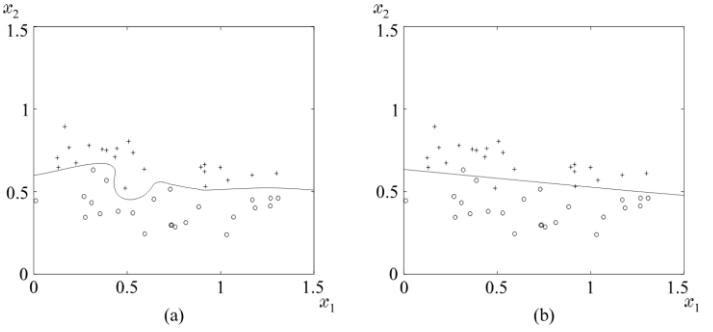
➔ The size of the network must be:

- Large enough to learn what makes data of the same class similar and data from different classes dissimilar.
- Small enough not to be able to learn underlying differences between data of the same class. This leads to the so called overfitting.

Choice of the network size

Example:

- Decision curve (a) before and (b) after pruning.



Choice of the network size

Overtraining is a common phenomena for classifiers that are very flexible in their decision surface, i.e., the network adapts to the peculiarities of the training set and therefore performs bad on new test data.