# International Master in Computer Vision

# **Fundamentals of machine learning for computer vision**

Eva Cernadas

USC UNIVERSIDADE DE SANTIAGO DE COMPOSTELA
Escola Técnica Superior de Enxeñaría
imcv

# Contents

- **Machine learning theory (Dr. Jaime Cardoso)**
- **Linear regression and optimization (Dr. Jaime Cardoso)**
- **Clustering**
- **Model selection and evaluation**
- **Classical classification models**
- **Artificial neural networks**
- **Support vector machines (SVM)**
- **Ensembles: bagging, boosting and random forest**

# Artificial neural networks

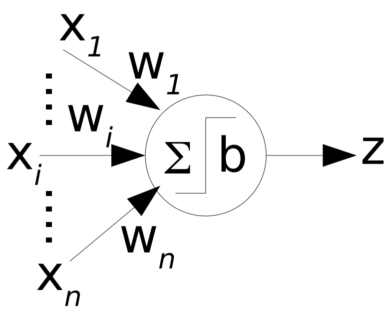
- **ANN: Artificial Neural Networks**
- Neural network: combination of local processing units (**neurons**)
- Neuron: simple processing unit of various **inputs** and one **output**.
- Input connections with **weights** for each input.
- The output is determined by **activation functions** from the inputs and weights.
- The weights are **persistent**: they are the **memory** of the neural net.
- The weights are calculated to approximate *n*-dimensional functions.
- The neurons are grouped in **layers**: multi-layers networks: input, hidden (one or various) and output layers.

# Neural network

- The term ANN are normally used to **multi-layer perceptron** (MLP).
- There are supervised and unsupervised neural networks.
- **Supervised** training: weights are calculated using training patterns and true outputs.
- Test pattern propagates through the net to calculate the output.
- The network **performance** is evaluated comparing true and predicted output using the previous measurements
- ANN may exhibit **overfitting**

# Perceptron (I)

- One neuron with *n* inputs and one output, binary activation function (0,1), e.g. binary classification:




$$z(\boldsymbol{x}) = f\left(\sum_{i=1}^{n} w_i x_i - b\right) = f\left(\boldsymbol{w}^T \boldsymbol{x} - b\right)$$

Step activation function

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

- Need a threshold *b* (for example *b*=0.5) to calculate the output (-1 or 1). Varying *b*, a ROC curve is generated.

- Connection weights $w_1 \ldots w_n$ : iteratively calculated

- A differential function *f*(*t*) may also be used

# Perceptron (II)

- Gradient descent (training algorithm): searches to minimize the difference between the desired ($y$) and predicted ($z$) outputs, summed over the training set, $\mu$ is the learning speed

$$J=\sum_{i=1}^{N}(y_i-z_i)^2; \quad \mathbf{w}(t+1)=\mathbf{w}(t)-\mu\frac{\partial J}{\partial \mathbf{w}}; \quad \frac{\partial J}{\partial \mathbf{w}}=-\sum_{i=1}^{N}(y_i-z_i)f'(\mathbf{w}^T\mathbf{x}_i-b)\mathbf{x}_i$$

$$\mathbf{w}(t+1)=\mathbf{w}(t)+\mu\sum_{i=1}^{N}(y_i-z_i)f'(\mathbf{w}^T\mathbf{x}_i-b)\mathbf{x}_i$$

Note that $y_i$-$z_i \in \{0, \pm 1\}$

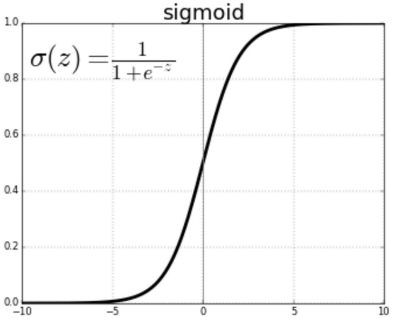
- If the data are linearly separable, this iterative training finds the separating hyperplane (**w**,b) for what $y_i\, z_i>0$ for i=1..N.

# Perceptron (III)

- Soft activation function *f*: sigmoid or hyperbolic tangent:

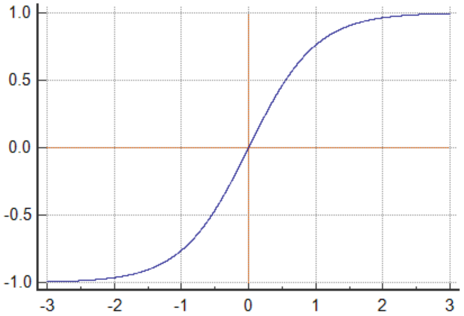
Logistic sigmoid function: 0,1

$$f(t) = \sigma(t) = \frac{1}{1+e^{-at}}$$




Hyperbolic tangent function: $\pm 1$

$$f(t) = \tanh at = \frac{e^{-at} - e^{at}}{e^{at} + e^{-at}}$$



# Kernel perceptron

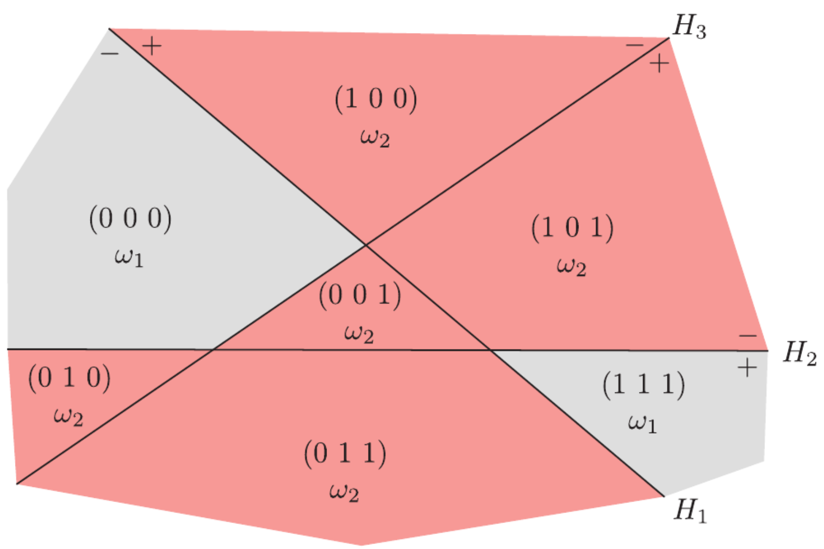
- Kernel perceptron: linear classification $z = f\left(\boldsymbol{w}^T \vec{\Phi}(\boldsymbol{x}) + b\right)$ in the multi-dimensional projected hidden space (*f* : step function) mapped by $\boldsymbol{\Phi}(\mathbf{x})$:

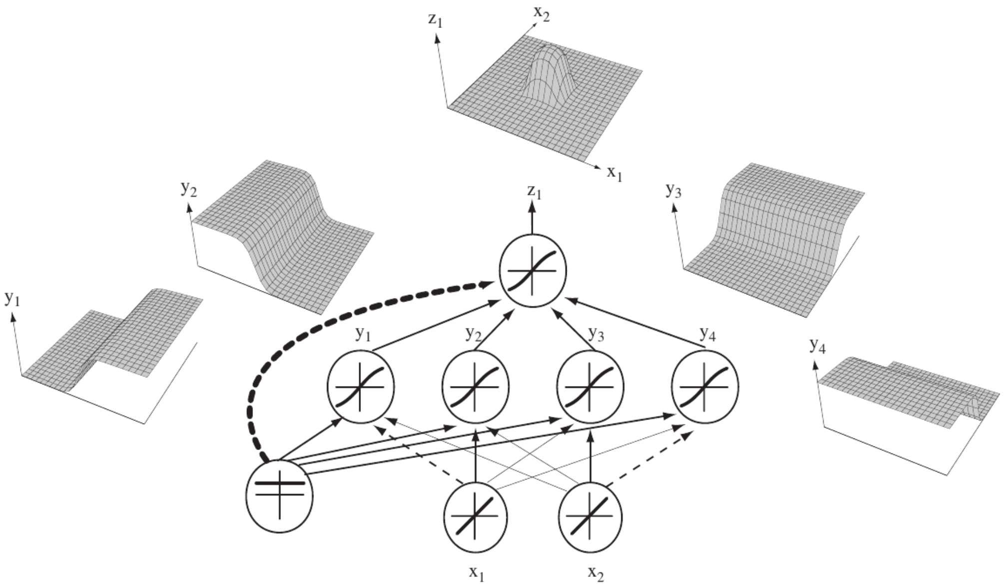
$$\boldsymbol{w} = \sum_{i=1}^{N} a_i y_i \vec{\Phi}(\boldsymbol{x}_i) \qquad b = \sum_{i=1}^{N} a_i y_i$$

- Initially, $a_i=0$, $i=1..N$; $a_i=a_i+1$ if $y_i\, z_i<0$ (misclassified patted) until that $y_i\, z_i>0$ (pattern correctly classified) for $i=1,...N$
- For datasets that are not linearly separable.

# Multi-layer neural network (MLP)

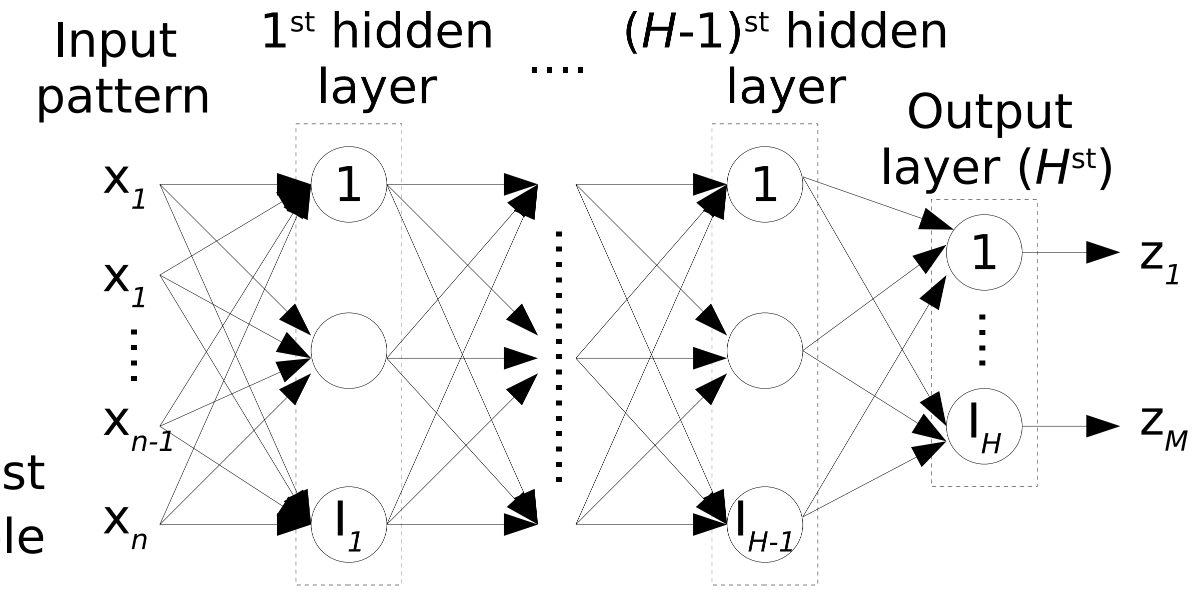
- Each neuron is a linear classifier of the input space (hyperplane).
- Divides the space in two subspaces with outputs 0,1 (sigmoid function) or $\pm 1$ (hiperbolic function).
- Joining various neurons into one layer, divides the space into regions with piecewise linear borders (surfaces).



# Multi-layer Layer Perceptron

- Hidden layer: sigmoidal/tanh activation function
- Output layer: step activation for clasificación, linear for regression
- We need: training set, cost function, and derivable activation




- $\mathbf{w}_j^k$: weight vector of neuron $j$=1..$l_k$ in layer $k$=1..$H$
- $a_{ij}^k=(\mathbf{w}_j^k)^T\mathbf{h}_i^{k-1}+b_j^k$ , $i$=1..$N$ (pattern), $k$=1..$H$ (layer), $j$=1..$l_k$ (neuron)
- $\mathbf{h}_i^{k-1}$: output of layer $k-1$ ($l_{k-1}$ values) for pattern $\mathbf{x}_i$
- $\mathbf{h}_i^k$: output of layer *k for pattern* $\mathbf{x}_i$: $h_{ij}^k=f(a_{ij}^k)$ with $j$=1..$l_k$
- $y_{ij}$ =true output for $j$-th output neuron and training pattern $\mathbf{x}_i$

# Backpropagation (I)

- Gradient descent:

$$\Delta \boldsymbol{w}_j^k = -\mu \frac{\partial J}{\partial \boldsymbol{w}_j^k}, \Delta b_j^k = -\mu \frac{\partial J}{\partial b_j^k}, k=1,\ldots,H, j=1,\ldots,I_k$$

- Sum $J$ of squared errors (SSE) $J_i$ evaluated on output layer for the *i pattern*:

$$J = \sum_{i=1}^{N} J_i \qquad J_i = \frac{1}{2}\sum_{j=1}^{I_H}\left(y_{ij} - h_{ij}^H\right)^2 = \frac{\left|\boldsymbol{y}_i - \boldsymbol{h}_i^H\right|^2}{2}$$

- The derivative is:

$$\frac{\partial J_i}{\partial \boldsymbol{w}_j^k} = \frac{\partial J_i}{\partial a_{ij}^k}\frac{\partial a_{ij}^k}{\partial \boldsymbol{w}_j^k}, \frac{\partial J_i}{\partial b_j^k} = \frac{\partial J_i}{\partial a_{ij}^k}\frac{\partial a_{ij}^k}{\partial b_j^k}$$

# Backpropagation (II)

- Define: $\delta_{ij}^{k} \equiv \frac{\partial J_i}{\partial a_{ij}^{k}}$

$a^k_{ij} = (\mathbf{w}^k_j)^T \mathbf{h}_i^{k-1} + b^k_j$

- Thus: $\frac{\partial a_{ij}^{k}}{\partial \boldsymbol{w}_j^{k}} = \boldsymbol{h}_i^{k-1}, \frac{\partial a_{ij}^{k}}{\partial b_j^{k}} = 1$

- So:

$$\Delta \boldsymbol{w}_j^k = -\mu \sum_{i=1}^{N} \delta_{ij}^k \boldsymbol{h}_i^{k-1}; \Delta b_j^k = -\mu \sum_{i=1}^{N} \delta_{ij}^k$$

# Backpropagation (III)

$$\delta_{ij}^{k} \equiv \frac{\partial J_i}{\partial a_{ij}^{k}}$$

$$J_i = \frac{1}{2}\sum_{j=1}^{I_H}\left(y_{ij} - h_{ij}^{H}\right)^2$$

$h_{ij}^{k} = f(a_{ij}^{k})$

- For the output layer ($k=H$):

$$\delta_{ij}^{H} = \left(y_{ij} - h_{ij}^{H}\right) f'\left(a_{ij}^{H}\right) = \varepsilon_{ij}^{H} f'\left(a_{ij}^{H}\right)$$

$$\Delta \boldsymbol{w}_j^{k} = -\mu \sum_{i=1}^{N} \varepsilon_{ij}^{H} f'\left(a_{ij}^{H}\right) \boldsymbol{h}_{ij}^{k-1}$$

$$\Delta b_j^{k} = -\mu \sum_{i=1}^{N} \varepsilon_{ij}^{H} f'\left(a_{ij}^{H}\right)$$

- For the layer $k<H$:

$$\delta_{ij}^{k} = \frac{\partial J_i}{\partial a_{ij}^{k}} = \sum_{l=1}^{I_{k+1}} \frac{\partial J_i}{\partial a_{il}^{k+1}} \frac{\partial a_{il}^{k+1}}{\partial a_{ij}^{k}} = \sum_{l=1}^{I_{k+1}} \delta_{il}^{k+1} \frac{\partial a_{il}^{k+1}}{\partial a_{ij}^{k}}$$

# Backpropagation (IV)

- Since $a_{il}^{k+1}=(\mathbf{w}_l^{k+1})^T\mathbf{h}_i^k+b_l^{k+1}$ and $h_{ij}^k=f(a_{ij}^k)$, then:

$$a_{il}^{k+1}=\sum_{m=1}^{I_k} w_{lm}^{k+1} f\left(a_{im}^k\right)+b_l^{k+1} \qquad \frac{\partial a_{il}^{k+1}}{\partial a_{ij}^k}=w_{lj}^{k+1} f'\left(a_{ij}^k\right)$$

- So $\delta_{ij}^k$ is a recursive function depending on $\delta_{ij}^{k+1}$ and $\mathbf{w}_l^{k+1}$:

$$\delta_{ij}^k=\sum_{l=1}^{I_{k+1}} \delta_{il}^{k+1} w_{lj}^{k+1} f'\left(a_{ij}^k\right)=f'\left(a_{ij}^k\right)\sum_{l=1}^{I_{k+1}} \delta_{il}^{k+1} w_{lj}^{k+1}, k=K-1,\ldots,1$$

- Denoting $\varepsilon_{ij}^k=\sum_{l=1}^{I_{k+1}} \delta_{il}^{k+1} w_{lj}^{k+1}$, we obtain: $\delta_{ij}^k=\varepsilon_{ij}^k f'\left(a_{ij}^k\right)$
- The derivatives are, respectively, $f'(t)=af(t)[1-f(t)]$ and $f'(t)=a[1-f^2(t)]$ for sigmoid and tanh activation functions.

# Backpropagation (V)

**repeat** # epoch loop
    **for** i=1:$N$
        **for** k=1:$H$ # direct pattern propagation
            **for** j=1:$I_k$
                $a_{ij}^k=(\mathbf{w}_j^k)^T\mathbf{h}_i^{k-1}+b_j^k$; $h_{ij}^k=f(a_{ij}^k)$
            **endfor**
        **endfor**
        **for** j=1:$I_H$
            $\varepsilon_{ij}^H=y_{ij}-h_{ij}^H$; $\delta_{ij}^H=\varepsilon_{ij}^H f'(a_{ij}^H)$
        **endfor**
        **for** k=$H$-1:-1:1 # error backpropagation
            **for** j=1:$I_k$
                $\varepsilon_{ij}^k=\sum_{l=1}^{I_{k+1}}\delta_{il}^{k+1}w_{lj}^{k+1}$; $\delta_{ij}^k=\varepsilon_{ij}^k f'(a_{ij}^{k+1})$
            **endfor**
        **endfor**
    **endfor**
    **for** k=1:$H$ # weight updating
        **for** j=1:$I_k$
            $\Delta\mathbf{w}_j^k=-\mu\sum_{i=1}^{N}\delta_{ij}^k\mathbf{h}_i^{k-1}; \Delta b_j^k=-\mu\sum_{i=1}^{N}\delta_{ij}^k$
            $\mathbf{w}_j^k=\mathbf{w}_j^k+\Delta\mathbf{w}_j^k$; $b_j^k=b_j^k+\Delta b_j^k$
        **endfor**
    **endfor**
**until** stop criterion

Bach processing

- Initial random low weights $\mathbf{w}_j^k$ and biases $b_j^k$
- Epoch: processing of the whole training set
- Stop criterion: 1) $J$ or gradient of $J$ below a threshold; 2) maximum of epochs
- Speed $\mu$ with intermediate values: avoid slowness and oscilations
- Select the best among different initializations in order to avoid falls into local minima with high $J$
- Weight updating pattern by pattern (online): it can avoid local minimums and converges faster

# Enhancements over backpropagation (I)

- **Preprocessing**: inputs using 0 mean and the same variance as the activation range $f(t)$.
- **Symmetrical activation** (tanh) instead of the sigmoid function.
- **Moment** ($\alpha$): inertia in the learning:

$$\Delta \boldsymbol{w}_j^k(t+1)=\alpha\Delta \boldsymbol{w}_j^k(t)-\mu\sum_{i=1}^{N}\delta_{ij}^k\boldsymbol{h}_i^{k-1}$$

$t$=iteration; $\alpha\in[0.7\text{-}0.95]$; $\Delta\mathbf{w}_j^k$ reduce aprox. in 1-$\alpha$

# Enhancements over backpropagation (II)

- **RMSProp** (*root mean square propagation*): use a second order moment instead of first order moment; $\eta,\beta$: hyper-parameters

epoch=presentation of training set

Use the squared gradient to scale the learning speed.

$S_w=0; S_b=0$

**for** epoch=1:nepoch

calculate $\Delta\mathbf{w}$ and $\Delta b$

$S_w=\beta S_w+(1-\beta)|\Delta\mathbf{w}|^2; S_b=\beta S_b+(1-\beta)\Delta b^2$

$$\mathbf{w}(t+1)=\mathbf{w}(t)-\frac{\eta\Delta\mathbf{w}}{\varepsilon+\sqrt{S_w}}; b(t+1)=b(t)-\frac{\eta\Delta b}{\varepsilon+\sqrt{S_b}}$$

**endfor**

# Enhancements over backpropagation (III)

- **Adaptive learning speed**: $\mu(t=0) \in [0.03\text{-}0.1]$
  - $\mu(t+1)=(1+\varepsilon_i)\mu(t)$ if $J(t+1)<J(t)$
  - $\mu(t+1)=(1-\varepsilon_d)\mu(t)$ and $a=0$ if $J(t+1)>(1+\varepsilon_c)J(t)$, with

    $\varepsilon_i=0.05$, $\varepsilon_d=0.3$ e $\varepsilon_c=0.04$
- **Speed reduction**: $t$=epoch; $\mu_o,\beta,k$: hyper-parameters

$$\mu(t)=\frac{\mu_0}{1+t\beta} \qquad \mu(t)=0.95^t\mu_0 \qquad \mu(t)=\frac{k\mu_0}{\sqrt{t}}$$

# Enhancements over backpropagation (IV)

- **Different speed for each weight**: increase $\mu$ when the gradient of that weight has the same sign in two iteractions.

- **Desired outputs** $y_{ij}$ corresponding with activation function (sigmoid or hyperbolic).

- If $y_{ij} \in [0,1]$, they can be seen as probabilities and **cross entropy** can be used as cost function instead of SSE:

$$J = -\sum_{i=1}^{N} \sum_{j=1}^{I_H} \left[ y_{ij} \ln h_{ij}^{H} + \left(1 - y_{ij}\right) \ln\left(1 - h_{ij}^{H}\right) \right]$$

# Enhancements over backpropagation (V)

- **Karhunen-Loewe divergence** or **relative entropy**, using *softmax* activation function for the output, can also be used as a cost function:

$$J=-\sum_{i=1}^{N}\sum_{j=1}^{I_H} y_{ij}\ln\frac{h_{ij}^{H}}{y_{ij}} \qquad h_{ij}^{H}=\frac{e^{a_{ij}^{H}}}{\sum_{l=1}^{I_H} e^{a_{il}^{H}}}$$

- The number $H$ of layers and neurons $I_k$ in each layer $k$=1..$H$ must be decided: tunable hyper-parameter

- With many neurons (and weights $\mathbf{w}_j^k$, that are trainable parameters), produces overfitting.

# Enhancements over backpropagation (VI)

- In practice, it was demostrated that backpropagation did not provide good solutions using various hidden layers.
- It was demostrated Mathematically that ANN with only one hidden layer is a universal approximator of any function.
- But this affects the training error, not test error: overfitting
- Number of neurons by layer: start with many neurons and use regularization to remove the less informative weights (pruning).

# Enhancements over backpropagation (VII)

- **Weight reduction**: use $J'(\mathbf{W})=J(\mathbf{W})+\lambda|\mathbf{W}|^2$ regularizated by the squared norm of the weight matrix $\mathbf{W}=\{\mathbf{w}_j^k\}$, $k=1..H$, $j=1..I_k$
- Alternative to $|\mathbf{W}|^2$: $\sum_{l=1}^{N_w} \frac{\theta_l^2}{\theta_h^2+\theta_l^2}$

being $\theta_l$ the $l^{th}$ weight ($l=1..N_w$) and $\theta_h$ the threshold: removes the weights $\theta_l<\theta_h$

# Enhancements over backpropagation (VIII)

- **Sensitivity analysis**: the weights $\theta_l$ with low saliency $s_l = h_{ll}\theta_l^2/2$ are periodically removed. $h_{ll}$ measures the effect over $J$ of removing $\theta_l$):

$$h_{ll} = \frac{\partial^2 J}{\partial \theta_l^2}$$

- **Early stopping**: each epoch, the network is tested over a separated validation set
- Training is stopped when the validation error starts to increase (overfitting).

# Enhancements over backpropagation (IX)

- **Shared weights**: some conections are enforced to share weight values to guarantee certain in-variances (ex: translation, rotation and scale in images).
- Alternative: to use input features which are invariants to these transformations.

# Limitations of MLP (I)

- Slow training, stucking in non-optimal local minima, most frequently with several hidden layers
- Many tunable hyper-parameters: number of hidden layers (*H*), number of neurons in each layer ($I_1..I_H$), learning speed ($\mu$), momentum ($\alpha$), etc.
- For some time, the multilayer networks ($H>2$) were discarded instead of one hidden layer ($H=2$) networks, that are universal approximators.

# Limitations of MLP (II)

- However, the neurons required with one layer is higher than with various layers
- Backpropagation is based on $f'(a_{ij}^{k+1})$, where *f* is sigmoid or tanh and exhibits null derivative in most its domain
- This leads to null gradients stopping training and generating many problems.

# MLP in Python

Class **MLPClassifier** implements a multi-layer perceptron (MLP) algorithm that trains using Backpropagation.

MLP trains on two arrays: array X of size (n_samples, n_features), which holds the training samples represented as floating point feature vectors; and array y of size (n_samples,), which holds the target values (class labels) for the training samples:

```
>>> from sklearn.neural_network import MLPClassifier
>>> X = [[0., 0.], [1., 1.]]
>>> y = [0, 1]
>>> clf = MLPClassifier(solver='lbfgs', alpha=1e-5,
...                     hidden_layer_sizes=(5, 2), random_state=1
...
>>> clf.fit(X, y)
MLPClassifier(alpha=1e-05, hidden_layer_sizes=(5, 2), random_stat
              solver='lbfgs')
```

After fitting (training), the model can predict labels for new samples:

```
>>> clf.predict([[2., 2.], [-1., -2.]])
array([1, 0])
```

# MLP in octave

- Package **nnet** in octave**:** https://octave.sourceforge.io/nnet/
  - **prestd()**: preprocesses the data so that the mean is 0 and the standard deviation is 1.
  - **trastd()**: preprocess additional data for neural network simulation (for example the test set).
  - **newff()**: create a feed-forward backpropagation network.
  - **train()**: a neural feed-forward network will be trained.
  - **sim()**: is usuable to simulate a before defined and trained neural network.
- Similar functions in the **Matlab Neural Network Toolbox**

# MLP in R

- Package **nnet** in R: **https://cran.r-project.org/web/packages/nnet/index.html**

nnet *Fit Neural Networks*

### Description

Fit single-hidden-layer neural network, possibly with skip-layer connections.

### Usage

```
nnet(x, ...)

## S3 method for class 'formula'
nnet(formula, data, weights, ...,
     subset, na.action, contrasts = NULL)

## Default S3 method:
nnet(x, y, weights, size, Wts, mask,
     linout = FALSE, entropy = FALSE, softmax = FALSE,
     censored = FALSE, skip = FALSE, rang = 0.7, decay = 0,
     maxit = 100, Hess = FALSE, trace = TRUE, MaxNWts = 1000,
     abstol = 1.0e-4, reltol = 1.0e-8, ...)
```

# Extreme Learning Machine (ELM)

- Network with $H$=2 layers: only one hidden layer, direct propagation
- Input weights $\mathbf{W}^1=\{w_{jk}^1\}$ and biases $\{b_j\}$, with $j=1..I_1$ and $k=1..n$, initialized with random values.
- Output weights $\mathbf{W}^2=\{w_{jk}^2\}$ and biases $\{b_j\}$, with $j=1..I_2$ ($I_2$=C, no. classes, for classification) and $k=1..I_1$
- $\mathbf{W}^2$ calculated using the pseudo inverse of activity matrix $\mathbf{H}$ and the desired outputs $\mathbf{Y}$ as $\mathbf{W}^2=\mathbf{Y}\ \mathbf{H}^\dagger$: direct and efficient for small datasets and networks
- $\mathbf{W}^2=(I_2 \times I_1)$, $\mathbf{Y}=(I_2 \times N)$, $\mathbf{H}=(I_1 \times N)$, $\mathbf{H}^\dagger=(N \times I_1)$

# Extreme Learning Machine (II)

- It was proved that training error converges to zero when $I_1 \rightarrow N$
- The activation function for the output layer is linear (for regression) or sigmoid+softmax (for classification):

Linear activation: $$z_{ij}=w_{jl}^{2} f\left(\sum_{m=1}^{n} w_{lm}^{1} x_{im}+b_{j}^{1}\right) \qquad i=1,\dots N;\ j=1\dots I_2$$

Sigmoid+ softmax activation: $$h_{ij}^{2}=f\left[w_{jl}^{2} f\left(\sum_{m=1}^{n} w_{lm}^{1} x_{im}+b_{l}^{1}\right)+b_{j}^{2}\right] \qquad z_{ij}=\frac{e^{h_{ij}^{2}}}{\sum_{k=1}^{I_2} e^{h_{kj}^{2}}}$$

- The number $I_1$ of hidden neurons is a tunable hyper-parameter, with best values lower than the number $N$ of training patterns

# Quick ELM (I)

- Problems of ELM with large datasets:
  - The activity **H** matrix is of order $I_1$ x $N$. With large datasets, it does not fit in memory
  - If **H** fits in memory, calculation of $\mathbf{H}^{\dagger}$ is not possible
  - Tuning of hidden layer size $I_1$ requires to repeat training many times, that is not possible
- Solution: **Quick ELM**
  - **Quick extreme learning machine for large-scale classification.** Audi Albtoush, Manuel Fernández-Delgado, Eva Cernadas and Senén Barro. Neural Computing and Applications, Vol. 34, pp. 5923–5938 (2022). DOI: 10.1007/s00521-021-06727-8

# Quick ELM (II)

- **Quick ELM:** efficient approach for classification

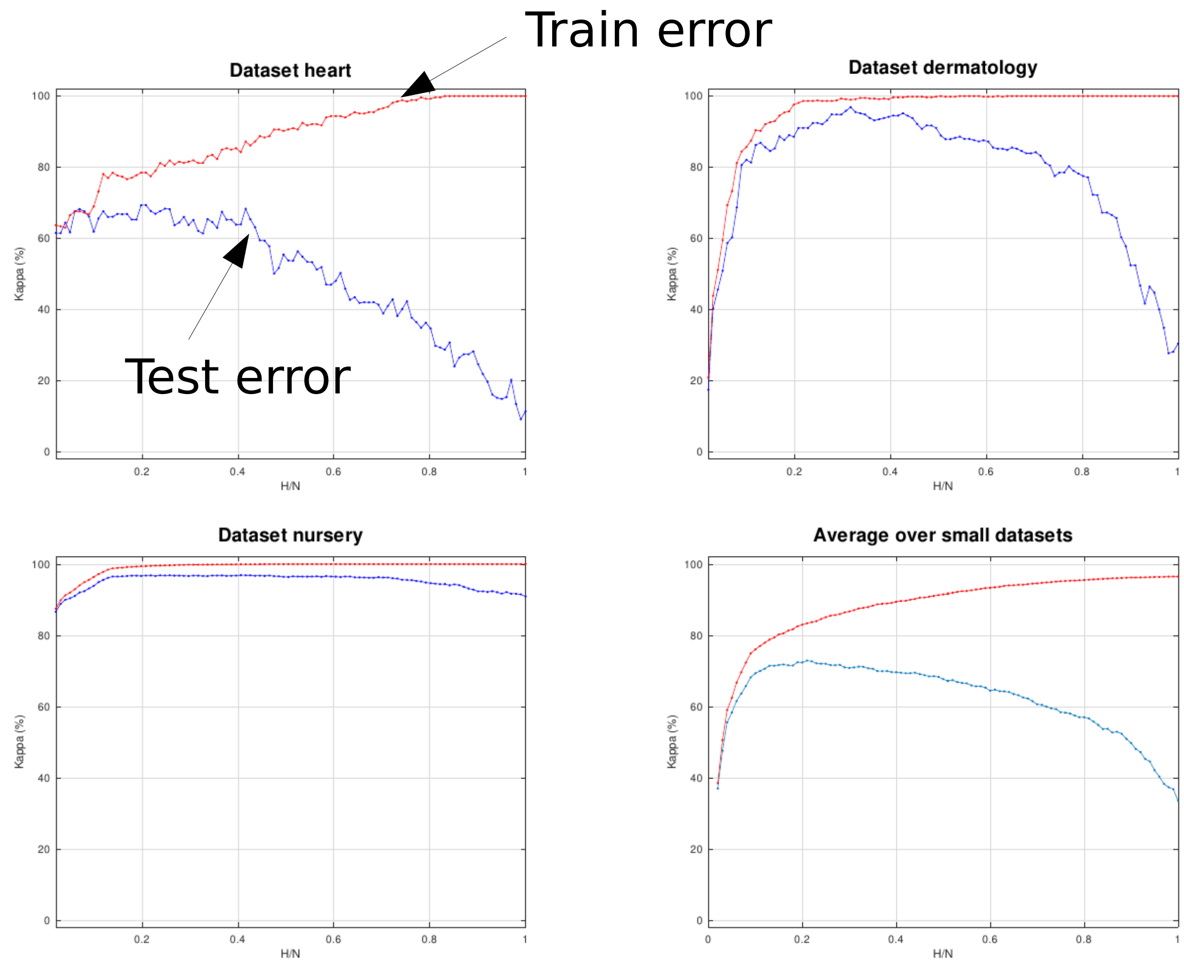
1) Avoids tuning of $I_1$ by estimating it from $N$
2) Bounds the size of matrix **H** for large datasets
3) Replaces patterns by prototypes to calculate **H**

- Works on datasets with 31 million patterns, 30,000 inputs and 130 classes
- Estimation of $I_1$:

$$I_1 = \lfloor \eta \min(N, N_0) \rfloor, \quad \eta = 0.15, \; N_0 = 15000$$

# Quick ELM (III)

- Based on behavior of performance vs $I_1$ / $N$
- The optimal $I_1$ is increasing with $N$
- Performance reduces when $I_1 \rightarrow N$
- Overtraining
- Empirically, we observed that $I_1$ about 0.15$N$ is a good choice
- Upper bounded by $N_0$



# Quick ELM (IV)

- Replaces $\mathbf{x}_n$ by $\mathbf{p}_{cl}$ (prototype) for $H_{kn}=g(\mathbf{w}_k^{1}\mathbf{x}_n+b_k)$ in matrix **H**
- Limited collection of prototypes $\{\mathbf{p}_{cl}\}$ with $c=1..C$, $l=1..L_c$
- The maximum number $L_c$ of prototypes per class depends on the class population $N_c$, with $L_c<100$
- The total number of prototypes is bounded to allow **H** fits in memory and be pseudo-inverted
- Each prototype $\mathbf{p}_{cl}$ is iteratively updated with its nearest training patterns of its class $c$:

$$c=y_n \qquad N_{cl}(t+1)=N_{cl}(t)+1$$

$$\boldsymbol{p}_{cl}(t+1)=\left[1-\frac{1}{N_{cl}(t)}\right]\boldsymbol{p}_{cl}(t)+\frac{\boldsymbol{x}_n}{N_{cl}(t)} \qquad l=\underset{j=1..L_c}{\mathrm{argmin}}\left\{\left|\boldsymbol{p}_{cj}-\boldsymbol{x}_n\right|\right\}$$

$N_{cl}(t)$: no. training patterns nearest to $l$-th prototype of class $c$

# Other neural networks

- Radial Basis Function (RBF) neural network
- Recursive networks: with feedback from outputs to inputs
- Self-Organized Map (SOM): non-supervised learning
- Learning vector quantization (LVQ): SOM with supervised learning
- Boltzmann machine

# Deep Learning

- Networks with many hidden layers:
  - Deep neural network (DNN)
  - Deep belief network (DBN)
  - Convolutional neural network (CNN)
  - Deep autoencoder network (DAN)
- **Non-supervised pre-training** for each layer separately: restricted Boltzman machine (RBM). Unsupervised. Locates the starting weights in areas of the weight space with good solutions
- **Supervised training** of the output weights
- **Fine training** of intermediate and output weights using back-propagation