

Neural Networks

1 Preprocessing

1.1 Input Normalization

Linear rescaling to avoid scaling problems. Useful for radial basis function and multilayer networks. Each input treated independently for scaling. Calculate mean and variance of training set $\mathbf{x}^{(m)}$ with $m \in [0, M-1]$ and normalize input vectors to $\mu_i = 0$ and $\sigma_i = 1$.

1.2 Feature Extraction

For **speech recognition** use AnaFB, power estimation ($|\cdot|^2$), mel-filterbank (15-30 normalized overlapping triangular filters), cepstrum estimation (IDFT/IDCT of log power spectrum - to decorrelate input features - symmetric), temporal features (combine successive features vectors - delta or delta-delta features), dimension reduction (feature space transformation, e.g. LDA - dimensionality reduction by preserving class discriminatory information - variance of features corresponding to one class is minimized, distance between classes maximized).

1.2.1 LDA

Dataset of M observations of N -dimensional Euclidian variable \mathbf{x} . Project \mathbf{x} to one dimension using a projection vector \mathbf{w} , such that $y^{(m)} = \mathbf{w}^T \mathbf{x}^{(m)}$. Cost function $\max_{\mathbf{w}} \mathbf{w}^T (\mu_1 - \mu_0)$ to separate class means. **Fisher's idea:** Large separation between projected class means while granting small variance within each class:

$$J(\mathbf{w}) = \frac{\mathbf{w}^T \mathbf{S}_b \mathbf{w}}{\mathbf{w}^T \mathbf{S}_w \mathbf{w}} \Rightarrow \mathbf{w}_{\text{opt}} = \mathbf{S}_w^{-1} (\mu_1 - \mu_0)$$

with \mathbf{S}_b being between class covariance matrix, \mathbf{S}_w within class covariance matrix.

2 Threshold Logic Units - Single Perceptrons

Neural Network is a blackbox/mapping machine/network of functions with N -dim. input \mathbf{x} and M -dim. output \mathbf{y} . A computing element/unit/node has unlimited fan-in and is a primitive function f of N arguments.

2.1 McCulloch-Pitts Neuron

Binary signals transmitted over edges produce binary result, driven by threshold Φ . N excitatory edges x_i and K inhibitory edges x'_j . Inhibitory edges can inactivate the whole unit if at least one is active. Else it works as a threshold gate. Neuron is activated if $\sum_{k=0}^{N-1} x_k \geq \Phi$.

2.2 Perceptron

With threshold Φ and input weights w_k , so the output is active if $\sum_{k=0}^{N-1} w_k x_k \geq \Phi$. Simple Perceptron and McCulloch-Pitts Unit are equivalent. Geometric interpretation can be used to check if a function can be computed by a Perceptron using a separating line/plane. XOR function with 2 vars can not be solved by one perceptron - not linearly

separable ($0 < \Phi$, $w_0 > \Phi$, $w_1 > \Phi$, $w_0 + w_1 < \Phi$). Two sets of A and B with different output values are linearly separable if $\sum_{x_k \in A} x_k w_k \geq \Phi$ and $\sum_{x_k \in B} x_k w_k < \Phi$. Solve XOR problem using network of perceptrons $(x_0 \wedge \bar{x}_1) \vee (\bar{x}_0 \wedge x_1)$. First layer labels the region, output layer decodes the classification.

2.3 Training

2.3.0.1 Supervised learning Weights of the network are initialized randomly. On misclassification network parameters are adjusted. Desired output is always known, thus it is called learning with a teacher. In **reinforcement learning** only input vector is used for the weight adjustment. In **corrective learning** the magnitude of the error together with the input vector are used for the weight correction. Training rule:

$$\Phi_{\text{new}} = \Phi_{\text{old}} + \Delta\Phi, \quad w_{\text{new},i} = w_{\text{old},i} + \Delta w_i$$

2.3.0.2 Unsupervised learning For a given input the output is unknown - learning without a teacher.

2.4 Perceptron Learning Algorithm

Transform perceptron into zero-valued threshold by turning Φ into a weight w_N (bias) such that $\sum_{k=0}^{N-1} x_k w_k - \Phi \geq 0$.

start : \mathbf{w}_0 is generated randomly

$n = 0$

test : Select $\mathbf{x} \in P \cup F$ randomly

if $\mathbf{x} \in P$ and $\mathbf{w}_t \mathbf{x} > 0$ go to test

if $\mathbf{x} \in P$ and $\mathbf{w}_t \mathbf{x} \leq 0$ go to add

if $\mathbf{x} \in N$ and $\mathbf{w}_t \mathbf{x} < 0$ go to test

if $\mathbf{x} \in N$ and $\mathbf{w}_t \mathbf{x} \geq 0$ go to subtract

add : set $\mathbf{w}_{n+1} = \mathbf{w}_t + \mathbf{x}$ and $n = n + 1$, goto test

subtract : set $\mathbf{w}_{n+1} = \mathbf{w}_t - \mathbf{x}$ and $n = n + 1$, goto test

2.5 Fast learning algorithm - delta rule

If $\mathbf{x} \in P$ is classified erroneously we have $\mathbf{w}_n^T \mathbf{x} \leq 0$ so we get the error $\Delta(n) = -\mathbf{w}_n^T \mathbf{x}$ so that the weight vector gets corrected: $\mathbf{w}_{n+1} = \mathbf{w}_n \frac{\Delta(n) + \epsilon^+}{\|\mathbf{x}\|^2} \mathbf{x}$. So that

$\mathbf{w}_{n+1}^T \mathbf{x} = \left(\mathbf{w}_n \frac{\Delta(n) + \epsilon^+}{\|\mathbf{x}\|^2} \mathbf{x} \right)^T \mathbf{x} = \epsilon^+ > 0$. ϵ guarantees that the new weight vector barely skips over the border of the region with higher error. For $\mathbf{x} \in F$ use ϵ^- . A variant is using $\gamma(\Delta(n) + \epsilon) \mathbf{x}$ with γ as learning factor.

2.6 Convergence

If P and N are finite and linearly separable.

$$\begin{aligned} \cos(\phi) &= \frac{\mathbf{w}_{\text{des}}^T \mathbf{w}_{n+1}}{\|\mathbf{w}_{n+1}\|} \\ &= \frac{\mathbf{w}_{\text{des}}^T \mathbf{w}_0 + (n+1)\delta}{\sqrt{\|\mathbf{w}_0\|^2 + (n+1)}} \end{aligned}$$

3 Multilayer perceptrons

ANN is a (L, C) tuple with a set of nodes L and a set of directed edges C . $c = (v, l) \in C$ are directed edges from node v to node l . L_{in} input layer subset, L_{hid} hidden layer subset, L_{out} output layer. Set $L_i^{(pre)}$ consists of prior nodes of node l (predecessors). Set $L_i^{(suc)}$ consists of subsequent nodes of node l (successors). Every edge has a weight $w_v^{(l)}$.

3.0.0.3 Generalized neuron Consists of network input function $f_{net}^{(l)}$, activation function $f_{act}^{(l)}$ and network output function $f_{out}^{(l)}$ and three states respectively $y_{net}^{(l)}$, $y_{act}^{(l)}$, $y_{out}^{(l)}$, also external input $x_{ref}^{(l)}$.

3.0.0.4 Properties of multilayer perceptrons A NN with layered architecture doesn't contain cycles. A MP is a feed-fwd NN with strictly layered structure. Normally all units in a layer connected to all other units of the next layer.

3.0.0.5 Weight matrices Given $L_{in} = \{v_0, \dots, v_{N-1}\}$ and $L_{out} = \{l_0, \dots, l_{M-1}\}$ the connection weights are gathered in a matrix

$$\mathbf{W} = \begin{bmatrix} w_{v_0}^{(l_0)} & w_{v_1}^{(l_0)} & \dots & w_{v_{N-1}}^{(l_0)} \\ w_{v_0}^{(l_1)} & w_{v_1}^{(l_1)} & \dots & w_{v_{N-1}}^{(l_1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{v_0}^{(l_{M-1})} & w_{v_1}^{(l_{M-1})} & \dots & w_{v_{N-1}}^{(l_{M-1})} \end{bmatrix} \in \mathbb{R}^{M \times N}$$

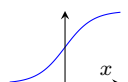
so that $\mathbf{y}_{net}^{(L_1)} = \mathbf{W} \mathbf{y}_{out}^{(L_0)}$.

3.0.0.6 Network functions NETWORK INPUT FUNCTION of hidden and output neurons is the weighted sum of the inputs $\sum_{v \in L_i^{(pre)}} w_v^{(l)} x_v^{(l)} = y_{net}^{(l)}$.

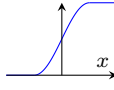
ACTIVATION FUNCTION of each hidden neuron is a monotonously increasing function. Sigmoid function $\lim_{x \rightarrow -\infty} f_{act}^{(l)}(x) = 0$ and $\lim_{x \rightarrow \infty} f_{act}^{(l)}(x) = 1$. Or bipolar sigmoid function with $\lim_{x \rightarrow -\infty} f_{act}^{(l)}(x) = -1$. The activation function of the output neuron is either a sigmoid or a linear function: $f_{act}^{(l)}(y_{net}^{(l)}, \Phi^{(l)}) = \beta y_{net}^{(l)} - \Phi^{(l)}$.

More activation functions:


- step function $f_{act}(x, \Phi) = \begin{cases} 1, & x \geq \Phi \\ 0 & \text{else} \end{cases}$
- semi-linear function: $f_{act}(x, \Phi) = \begin{cases} 1, & x > \Phi + 0.5 \\ 0, & x < \Phi - 0.5 \\ (x - \Phi) + 0.5 & \text{else} \end{cases}$

- logistic function $f_{act}(x, \Phi) = \frac{1}{1 + e^{-(x - \Phi)}}$ 

- sine until saturation:

$$f_{act}(x, \Phi) = \begin{cases} 1, & x > \Phi + \pi/2 \\ 0, & x < \Phi - \pi/2 \\ \frac{\sin(x-\Phi)+1}{2} & \text{else} \end{cases}$$


- (bipolar sigmoid:) hyperbolic tangent

$$f_{act}(x, \Phi) = \tanh(x - \Phi)$$


3.0.0.7 Function approximation Approximate desired function by step function and let NN compute the step function. Any Riemann integrable function can be approximated with arbitrary accuracy by a 4 layer perceptron. Every quantization range is represented by a neuron in the second hidden layer. One hidden layer can be omitted when using relative heights.

To achieve PIECEWISE APPROXIMATION use semi-linear activation functions with $\Delta x = x_{i+1} - x_i$ and $\Phi_i = \frac{x_i}{\Delta x}$

3.1 Regression analysis

3.1.0.8 Linear regression Training neural networks is closely related to linear regression. Regression line $g(x)$ minimizes the sum of squared errors $\sum_{m=0}^{M-1} (g(x^{(m)}) - y^{(m)})^2$ to the samples of the data set.

3.1.0.9 Multi-linear regression TODO

3.1.0.10 Non-linear regression Try to find a transformation for NON-POLYNOMIAL FUNCTIONS to a linear/polynomial case, e.g. $y = ax^c \Rightarrow \ln y = \ln a + c \ln x$. Special case: For NN if a logistic function is used it is important to have a transformation into a linear form (logistic regression). Ideal output: $y^{(m)} = \frac{1}{1 + \exp(-\sum_{i=0}^N w_i x_i^{(m)})}$, which is a non-linear regression - or a linear regression for $0 < y^{(m)} < 1$ satisfying: $\sum_{i=0}^N w_i x_i^{(m)} = \ln \frac{1-y^{(m)}}{y^{(m)}}$ which is a linear regression model.

3.1.0.11 Logistic regression (only applicable for 2-layer perceptron) Assume logistic function $y = \frac{A}{1+e^{-(wx+\Phi)}}$. Build inverse $\frac{A-y}{y} = e^{-(wx+\Phi)}$ and apply Logit-Transformation: $\ln \frac{y}{A-y} = wx + \Phi$. Logistic regression can be computed using a single neuron with $y_{net} = wx$, $y_{act} = \frac{1}{1+e^{-(y_{net}-\Phi)}}$ and $y_{out} = Ay_{act}$.

4 Training multilayer perceptrons

4.0.0.12 Definition and properties Popular learning algorithm is backpropagation algorithm, which searches for the minimum of the error function in weight space using gradient descent. Solution of the learning problem is the combination of weights that minimizes the error function, which must be

continuous and differentiable (same as the activation function - popular: sigmoide function).

4.0.0.13 Gradient descent Goal: Output $y_{out}^{(m)}$ and desired output $y_{ref}^{(m)}$ should be identical $\forall m$.

$$\min e = \sum_{m \in M_{ref}} \sum_{v \in L_{out}} (y_{ref}^{(v,m)} - y_{out}^{(v,m)})^2$$

Calculate direction of correction step

$\nabla_{\mathbf{w}^{(l)}} e = \left(\frac{\partial e}{\partial w_{p_0}^{(l)}}, \dots, \frac{\partial e}{\partial w_{p_{N-1}}^{(l)}}, -\frac{\partial e}{\partial \Phi^{(l)}} \right)^T$. Error of multilayer perceptron is the sum of individual errors over the training patterns and the individual error depends on weights through the network input $y_{net}^{(l,m)} = (\mathbf{w}^{(l)})^T \mathbf{x}^{(l,m)}$. Using chain rule:

$$\frac{\partial e^{(m)}}{\partial \mathbf{w}^{(l)}} = \frac{\partial e^{(m)}}{\partial y_{net}^{(l,m)}} \frac{\partial y_{net}^{(l,m)}}{\partial \mathbf{w}^{(l)}}$$

with the individual error at the output:

$$e^{(m)} = \sum_{v \in L_{out}} (y_{ref}^{(v,m)} - y_{out}^{(v,m)})^2. \text{ BLABLA}$$

4.0.0.14 i) l is an output neuron Gradient

$$\frac{\partial e^{(m)}}{\partial \mathbf{w}^{(l)}} = -2(y_{ref}^{(l,m)} - y_{out}^{(l,m)}) \frac{\partial y_{out}^{(l,m)}}{\partial y_{net}^{(l,m)}} \mathbf{x}^{(l,m)}$$

Update network weights in negative gradient direction with learning constant γ defining the step length.

$$\Delta \mathbf{w}^{(l,m)} = -\frac{\gamma}{2} \frac{\partial e^{(m)}}{\partial \mathbf{w}^{(l)}}$$

online training: apply weight corrections sequentially after each pattern presentation **batch training:** compute weight corrections over all training patterns and use the sum of corrections for the update
Update rule:

$$\mathbf{w}_{new}^{(l)} = \mathbf{w}_{old}^{(l)} + \Delta \mathbf{w}^{(l,m)}$$

4.0.0.15 ii) l is a hidden neuron Recursive equation called error backpropagation:
TODO
Simplified update equation

$$\Delta \mathbf{w}^{(l,m)} = \gamma \left(\sum_{s \in L_l^{(suc)}} \delta^{(s,m)} w_l^{(s)} \right) y_{out}^{(l,m)} (1 - y_{out}^{(l,m)}) \mathbf{x}^{(l,m)}$$

4.0.0.16 Summary INITIALIZATION: $y_{out}^{(u,m)} = x_{ref}^{(u,m)}$

FWD PROPAGATION up to the hidden layer:

$$y_{out}^{(l,m)} = \frac{1}{1 + \exp(-\sum_{p \in L_l^{(pre)}} w_p^{(l)} x_p^{(l,m)})}$$

ERROR TERM and WEIGHT UPDATE at the output layer:

$$\delta^{(l,m)} = (y_{ref}^{(l,m)} - y_{out}^{(l,m)}) y_{out}^{(l,m)} (1 - y_{out}^{(l,m)}) \text{ and}$$

$$\Delta w_p^{(l,m)} = \gamma \delta^{(l,m)} y_{out}^{(p,m)}. \text{ BACKWARD PROPAGATION and WEIGHT UPDATE at the hidden layers:}$$

$$\delta^{(l,m)} = \left(\sum_{s \in L_l^{(suc)}} \delta^{(s,m)} w_l^{(s)} \right) y_{out}^{(l,m)} (1 - y_{out}^{(l,m)}) \text{ and}$$

$$\Delta w_p^{(l,m)} = \gamma \delta^{(l,m)} y_{out}^{(p,m)}$$

4.0.0.17 Update rules :

- standard backpropagation (gradient descent)
 $\Delta w(t) = -\frac{\gamma}{2} \nabla_w e(t)$
- Manhattan-Training (useful for error functions that have a flat characteristic) $\Delta w(t) = -\gamma \text{sgn}(\nabla_w e(t))$
- momentum method using the last update term. Accelerates training in flat regions but decreases in an uniform direction $\Delta w(t) = -\frac{\gamma}{2} \nabla_w e(t) + \alpha \Delta w(t-1)$ with $\alpha < 1$
- Self adaptive error backpropagation. γ is increased ($\alpha \in [0.3, 0.5]$) if the current and the previous gradients have the same sign, and is decreased ($\beta \in [0.05, 0.2]$) when they have different signs. Use only for batch training.
- Quick propagation: Approximate error function by a parabola at current weight position. From current and previous gradient the apex of the parabola is determined to use as the new weight value
 $\Delta w(t) = \frac{\nabla_w e(t)}{\nabla_w e(t-1) - \nabla_w e(t)} \Delta w(t-1)$
- Weight decay: Large weights are inappropriate as they reach the saturation region of the logistic function, resulting in very low gradients. Also overfitting may occur. $\Delta w(t) = -\frac{\gamma}{2} \nabla_w e(t) - \zeta w(t)$ with $0 < \zeta \ll 1$

4.0.0.18 Sensitivity analysis Understand knowledge of a NN after training. Analyze change of output relative to change of input (derivative of the outputs w.r.t. the inputs).

5 Radial Basis Function Networks

Feed-forward NNs, similar to multilayer perceptrons and trained using supervised training algorithms. Consist of 3 layers with $L_{in} \cap L_{out} = \emptyset$. Only adjacent layers are connected. Training of RBFN are relatively fast compared to backpropagation networks. As activation function a radial function is used. The activation of a hidden unit is determined by the distance between the input vector and the connection weight vector. Weights from the input layer to a hidden neuron characterize the center of a radial basis function. Each radius (distance from the center) of a RBF is assigned to its corresponding activation.

5.0.0.19 Distance functions Distance between input vector and weight vector is used as input function for each hidden neuron: $f_{net}^{(l)}(\mathbf{w}^{(l)}, \mathbf{x}^{(l)}) = d(\mathbf{w}^{(l)}, \mathbf{x}^{(l)}) \quad \forall l \in L_{hid}$ with distance function d . Distance between input and weight vector $d_k(\mathbf{w}, \mathbf{x}) = (\sum_{i=0}^{N-1} |w_i - x_i|^k)^{\frac{1}{k}}$. Special cases: $k = 1$: Manhattan distance, $k = 2$: Euclidian distance, $k = \infty$: Maximum distance, $d_{\infty}(\cdot) = \max_{0 \leq i < N} |w_i - x_i|$

5.0.0.20 Network functions (hidden and output layer)

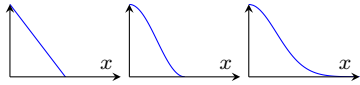
Each hidden neuron uses a radial function as activation function with $f_{act}^{(l)}(0) = 1$ and $\lim_{x \rightarrow \infty} f_{act}^{(l)}(\infty) = 0$. The network input function of each output neuron is the weighted sum of the input vector:

$$f_{net}^{(l)}(\mathbf{w}^{(l)}, \mathbf{x}^{(l)}) = \sum_{v \in L_l^{(pre)}} w_v^{(l)} x_v^{(l)} = y_{net}^{(l)}.$$

Linear function as activation function for each output neuron: $f_{act}^{(l)}(y_{net}^{(l)}, \theta^{(l)}) = \beta y_{net}^{(l)} - \theta^{(l)}$.

5.0.0.21 Radial activation functions

- rectangle function $f_{act}(x, \theta) = \begin{cases} 1, & x \leq \theta \\ 0 & \text{else} \end{cases}$
- triangle function $f_{act}(x, \theta) = \begin{cases} 1 - x/\theta, & x \leq \theta \\ 0 & \text{else} \end{cases}$
- cosine function $f_{act}(x, \theta) = \begin{cases} \frac{\cos(\pi x / (2\theta)) + 1}{2}, & x \leq 2\theta \\ 0 & \text{else} \end{cases}$
- gaussian function $f_{act}(x, \theta) = \exp(-\frac{1}{2}(\frac{x}{\theta})^2)$



5.0.0.22 Initialization - Simple RBFN Hidden layer has as many neurons as the number of training patterns. Connection weights from the input neurons to the hidden neurons are initialized by the elements of the input patterns from the training set $\mathbf{w}^{(v_m)} = \mathbf{x}_{ref}^{(m)}$. Determine radii for Gaussian function heuristically $\theta^{(v_m)} = \frac{d_{max}}{\sqrt{2M}}$. Maximum distance between two input vectors: $d_{max} = \max_{0 \leq m < k < Md} d(\mathbf{x}_{ref}^{(m)}, \mathbf{x}_{ref}^{(k)})$.

Output neuron network input and activation function are linear, so connection weights from hidden to output can be solved via $\mathbf{Y}_{out} \mathbf{w}^{(l)} = \mathbf{y}_{ref}^{(l)}$ with bias $\theta^{(l)} = 0$, so that

$$\mathbf{w}^{(l)} = \mathbf{Y}_{out}^{-1} \mathbf{y}_{ref}^{(l)}.$$

Drawbacks:

Number of training patterns are often huge, and it is reasonable to capture many learning tasks with the same RBF.

5.0.0.23 Initialization - Standard RBFN Lower hidden neurons than training patterns. For init a subset of training patterns are used as centers for the RBF. Coordinates of the training patterns are copied into the weight vectors. Radii determined heuristically similar to simple RBFN.

INITIAL WEIGHTS: \mathbf{Y}_{out} is a $M \times K + 1$ Matrix (non-quadratic). Use Moore-Penrose Pseudo Inverse: $\mathbf{Y} = (\mathbf{Y}^T \mathbf{Y})^{-1} \mathbf{Y}^T$ so that $\mathbf{w}^{(l)} = \mathbf{Y}_{out}^+ \mathbf{y}_{ref}^{(l)}$.

In real applications only an approximation of the desired solution is achieved due to the overdetermination. Final solution is achieved after a subsequent training for standard RBFN.

5.0.0.24 Clustering - K-Means Cost function: Minimize average distance: $F = \frac{1}{M} \sum_{m=0}^{M-1} \sum_{k=0}^{K-1} r_{mk} \|\mathbf{x}_{ref}^{(m)} - \mathbf{c}_k\|^2$. Use centroids as initial centers for the RBFs. Use std.dev. between cluster centers and their features as radii of RBFs.

5.0.0.25 Training RBFNs Train using gradient descent. First stage: connection weights from hidden to output neurons and bias values. Second stage: hidden layer parameters - connections weights and radii of RBFs.

OUTPUT PARAMETER update terms: $\Delta \mathbf{w}^{(l,m)} = \frac{\gamma_3}{2} \frac{\partial e^{(m)}}{\partial \mathbf{w}^{(l)}}$.

HIDDEN PARAMETER update rule for center coordinates: $\Delta \mathbf{w}^{(v,m)} =$

$$\gamma_1 (\sum_{s \in L_v^{(suc)}} (y_{ref}^{(s,m)} - y_{out}^{(s,m)}) w_v^{(s)} \frac{\partial y_{out}^{(v,m)}}{\partial y_{net}^{(v,m)}} \frac{\partial y_{net}^{(v,m)}}{\partial \mathbf{w}^{(v)}}).$$
 Often

Euclidian distance as input function and Gaussian function as activation.

The radius update term

$$\Delta \theta^{(v,m)} = \gamma_2 (\sum_{s \in L_v^{(suc)}} (y_{ref}^{(s,m)} - y_{out}^{(s,m)}) w_v^{(s)} \frac{\partial y_{out}^{(v,m)}}{\partial \theta^{(v)}})$$

5.0.0.26 Generalization - Mahalanobis Distance Prior distance functions are not appropriate for training patterns that are distributed ellipsoidal in the input space. Use a large number of RBFs concatenated along the feature contour line (increases complexity). Use Mahalanobis distance (hyper ellipsoid). Special case: Uncorrelated features with same variance in all directions yields equivalence of Mahalanobis and Euclidean distance.

5.0.0.27 MLP vs RBFN RBFN only has a single hidden layer. Hidden and output neurons have the same underlying function in MLP - RBFN uses distinct functions. All neurons in MLP are nonlinear, output layer of RBFNs is linear. Hidden layers in RBFs calculate the distance of input and center - MLP calculates inner product of input and weight vectors. Monotonously decreasing function as activation function for hidden neurons in RBFNs - MLP monotonously increasing function.

Advantages of RBFN: Simple feed-fwd. architecture, simple adaptation, extremely fast training. Applications: Processes

with quick adaptation, function approx., pattern recognition, control engineering.

6 LVQ

Can be seen as a special case of a supervised NN using competitive learning. No hidden neurons. Similar to RBFNs the distance between input and weight vector is used as input function for each output neuron:

$$f_{net}^{(l)}(\mathbf{w}^{(l)}, \mathbf{x}^{(l)}) = d(\mathbf{w}^{(l)}, \mathbf{x}^{(l)}).$$

DISTANCE FUNCTIONS same as in 5.0.0.19.

ACTIVATION FUNCTION use radial function (5.0.0.20) for output neuron.

OUTPUT FUNCTION of output neurons is not the identity, rather considers the activations of all output neurons simultaneously:

$$f_{out}^{(l)}(y_{out}^{(l)}) = \begin{cases} 1 & y_{act}^{(l)} = \max_{v \in L_{out}} y_{act}^{(v)} \\ 0 & \text{else} \end{cases}$$

The unit with maximal activation is selected to have an output of one (winner take all principle)
Same functions as in 5.0.0.21.

6.0.0.28 Learning Rule Init: Reference or codebook vectors are set on random positions in the input space
Training sample: Select training vector \mathbf{x} from training set
Distance measurement: Distance between all reference vectors and the training sample is computed, closest ref vector \mathbf{r} wins.
Adaptation:

- Attraction rule: (\mathbf{x} and \mathbf{r} have same class)
 $\mathbf{r}^{(new)} = \mathbf{r}^{(old)} + \gamma(\mathbf{x} - \mathbf{r}^{(old)})$
- Repulsion rule: (\mathbf{x} and \mathbf{r} have different classes)
 $\mathbf{r}^{(new)} = \mathbf{r}^{(old)} - \gamma(\mathbf{x} - \mathbf{r}^{(old)})$

6.0.0.29 Time dependent learning rate Fixed learning rate may cause oscillations, use time-dependent: $\gamma(t) = \gamma_0 K^t$
 $0 < K < 1$

6.0.0.30 Improved learning rule Not only update the winning neuron - rather update two ref vectors closest to the training samples. If classes of \mathbf{r}_j and \mathbf{r}_k are not the same and one of them belongs to the class of \mathbf{x} update the reference vectors:

$$\mathbf{r}_j^{(new)} = \mathbf{r}_j^{(old)} + \gamma(\mathbf{x} - \mathbf{r}_j^{(old)}), \quad \mathbf{r}_k^{(new)} = \mathbf{r}_k^{(old)} - \gamma(\mathbf{x} - \mathbf{r}_k^{(old)})$$

6.0.0.31 Window rule Only adapt if \mathbf{x} is close to classification boundary (inside a window), to omit driving ref. vectors away indefinitely. $\min(d_{jk}, d_{kj}) > \theta$ with $\theta = \frac{1-w}{1+w}$ with $0 < w \leq 1$ being the width of the window around the classification boundary.

6.0.0.32 Soft LVQ Soft assignment instead of winner-takes-all principle. Assumption: PDF of training data is described by a mixture of normal distributions. Each ref. vector describes one Gaussian distribution. Objective: Maximize log-likelihood ratio. Maximize rate of correct classification and minimize rate of misclassification. Update rule: Reference vectors with correct labels are attracted towards training sample, incorrect prototypes are repelled.

6.0.0.33 Hard LVQ If width of Gaussian components σ goes to zero, soft assignment becomes hard assignment. Always update closest vector of same/different class by attracting/repelling. No window rule needed, hard LVQ is stable.

6.0.0.34 Extensions Frequency sensitive competitive learning: Distance to ref. vector is modified dependant on the number of data points assigned to the ref. vector. Fuzzy learning vector quantization: Minimize network output error and distances between training patterns and competing neurons - leads to faster clustering.

7 Kohonen Self-Organizing Maps

Output neurons and neighbourhood relationships (topology). Each input neuron is connected to all output neurons. Distance between input and weight vectors is used as input function (RBFN). Activation function is a radial function. Identity as output function for each output neuron. Topology described by distance function d_{nb} . Training highly influenced by topology. SOM always activates neuron with least distance to input pattern (winner takes all). Without neighborhood relationships SOM operates as VQ of the input space. SOM are trained without a teacher. Mostly 1D or 2D topology in a grid.

7.0.0.35 Training Start with random neuron centers \mathbf{r}_l . Select random input vector \mathbf{x} . Determine distance for every neuron and select winner neuron l_w with maximum activation. Update ref vectors using neighborhood/topology function $f_{nb}(\mathbf{x})$:

$$\mathbf{r}_l(t+1) = \mathbf{r}_l(t) + \mu(t)f_{nb}(d_{nb}^{(l,l_w)}(t), \rho(t))(\mathbf{x}(t) - \mathbf{r}_l(t))$$

Continue with selecting \mathbf{x} until max number of iterations reached. Time-varying learning rate: $\mu(t) = \mu_0 \alpha_\mu^t$, and time-varying neighborhood radius: $\rho(t) = \rho_0 \alpha_\rho^t$. Topology functions: Gaussian, triangular, mexican hat, rectangle. Common topology is Gaussian function $f_{nb}(d_{nb}^{(l,l_w)}(t), \rho(t)) = \exp(-\frac{\|g_l - g_{l_w}(t)\|^2}{2\rho(t)^2})$ with g_l representing the neuron on the grid. Training may fail if init not correctly, e.g. learning rate or neighbourhood radius too low. SOM can be used for dimensionality reduction, e.g. 2D SOM in 3D input space if trained with points of a rotation parabola with 3 input neurons (x,y,z).

Applications: Speech recognition (each neuron responds strongly to a specific phoneme). Dimension reduction: animal names and attributes.

8 Hopfield networks (special recurrent network)

Network with cycles trained using supervised training. All neurons are input as well as output neurons - no hidden. Every neuron connected to all other neurons. Connection weights are symmetric $w_{uv}^{(l)} = w_l^{(u)}$.

Network input function of each neuron is the weighted sum of the outputs of all other neurons $y_{net}^{(l)} = \sum_{v \in L - \{l\}} w_v^{(l)} y_{out}^{(v)}$.

Activation function of a neuron is a threshold function $y_{act}^{(l)} = \begin{cases} 1 & y_{net}^{(l)} \geq \theta^{(l)} \\ -1 & \text{else} \end{cases}$. Can also take into account previous

activation if $y_{net}^{(l)} = \theta^{(l)}$ use previous activation value. Output function is the identity.

Symmetric weight matrix:

$$\begin{bmatrix} 0 & w_{v_1}^{(l_0)} & \cdots & w_{v_{N-1}}^{(l_0)} \\ w_{v_1}^{(l_0)} & 0 & \cdots & w_{v_{N-1}}^{(l_1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{v_{N-1}}^{(l_0)} & w_{v_{N-1}}^{(l_1)} & \cdots & 0 \end{bmatrix}$$

Weights control update procedure of the network and take pos, neg or zero values. Positive weights force neurons to become equal (energetically more favorable), negative weights drive neurons to be different, 0 weights have no influence. Weights describe the way from current state of the network towards the next minimum of the energy function.

8.0.0.36 Updates Neurons are processed one after the other and activations are recalculated until convergence. Performance depends on update order. **Synchronous update** may generate oscillations. **Asynchronous update** always converges to stable state, which depends on the update order.

8.0.0.37 Simplified representation Combine symmetric connections with double arrows.

8.0.0.38 Convergence properties Hopfield network with symmetric weight matrix with zeros on diag always converges if updated asynchronously. At most $N2^N$ updates of single neurons needed. Minimum of energy surface searched in input states. Stops on minimum. Output is binary state string. **Energy function** $E = -\frac{1}{2} \mathbf{y}_{act}^T \mathbf{W} \mathbf{y}_{act} + \theta^T \mathbf{y}_{act}$. After each iteration the energy remains const or decreases.

Proof of convergence: $\Delta E = (y_{act}^{(l,old)} - y_{act}^{(l,new)})(y_{net}^{(l)} - \theta^{(l)})$ with 2 cases $y_{net}^{(l)} < \theta^{(l)}$ or $y_{net}^{(l)} \geq \theta^{(l)}$. In state graphs the energy always decreases. State graph isn't always symmetric (non-zero thresholds).

8.0.0.39 Ising model HN can be seen as a microscopic model of magnetism - communication between particles and particles try to reach energetically favourable state.

Neuron	Atom
Activation state	Magnetic moment (spin)
Threshold	Strength of ext. magn. field
Connection weights	Magnetic coupling of atoms
Energy function	Hamilton operator

8.0.0.40 Associative memory Use HN as associative memory (stable states). Known pattern yields same exact pattern. Noisy patterns can be recognized or corrected. Find weights and thresholds that patterns are stable states of HN. M patterns \mathbf{p}_m of length N .

Storing a single pattern necessary condition: $F(\mathbf{W}\mathbf{p} - \theta) = \mathbf{p}$ with element-wise threshold (mapping) function F (1 if $x_i \geq 0$, -1 else). Simplify: $\theta = \mathbf{0}$. Find \mathbf{W} that has positive EV w.r.t \mathbf{p} : $\mathbf{W}\mathbf{p} = c\mathbf{p}$. Choose $\mathbf{W} = \mathbf{p}\mathbf{p}^T - \mathbf{I}$ which yields $\mathbf{W}\mathbf{p} = \mathbf{p}\mathbf{p}^T\mathbf{p} - \mathbf{I}\mathbf{p} = (N-1)\mathbf{p}$ with pattern \mathbf{p} being a stable state of HN.

Hebbian learning: neurons which fire together, wire together. Weights $w_{uv}^{(l)} = 1$ if neuron l and u are simultaneously active, 0 if $u = l$ and -1 else. Minimum of energy function in HN is located at $\mathbf{y}_{act} = \mathbf{p}$.

Storing several patterns: $\mathbf{W} = (\sum_{m=0}^{M-1} \mathbf{p}_m \mathbf{p}_m^T) - M\mathbf{I}$. Excitation of the units $\mathbf{W}\mathbf{p}_k = (N-M)\mathbf{p}_k + \sum_{m=0, m \neq k}^{M-1} \mathbf{p}_m (\mathbf{p}_m^T \mathbf{p}_k)$. 2nd term is perturbation term. Patterns stable if $M < N$ and perturbation is small. Capacity of HN very small compared to the number of possible states (2^n). Best results if \mathbf{p}_k orthogonal: $\mathbf{W}\mathbf{p}_k = (N-M)\mathbf{p}_k$. Complements $-\mathbf{p}_k$ are also stable states of HN.

Perceptron learning if Hebbian learning doesn't find a solution. Transform weight matrix into a vector

$$\mathbf{w} = \begin{pmatrix} w_{l_1}^{(l_0)} & w_{l_2}^{(l_0)} & \cdots & w_{l_{N-1}}^{(l_0)} \\ & w_{l_2}^{(l_1)} & \cdots & w_{l_{N-1}}^{(l_1)} \\ & & \ddots & \vdots \\ & & & w_{l_{N-1}}^{(l_{N-2})} \\ -\theta_{u_1} & -\theta_{u_2} & \cdots & -\theta_{u_n} \end{pmatrix}$$

Construct input vector \mathbf{z} so that the conditions yield $y(\mathbf{w}^T \mathbf{z}_{k,m}) = y_{act,m}^{(l_k)}$ for all neurons. Solution by linear separation of $\mathbf{z}_{k,m}$. If $y(\cdot) = 1$ \mathbf{z} belongs to the positive half-space, if $y(\cdot) = -1$ negative half-space. Perceptron learning can be used to compute \mathbf{w} for linear separation. Learning of HN with N units can be transformed into learning of perceptron of dim $N(N+1)/2$.

8.0.0.41 Optimization HN for which an energy function of a certain form exists can be used to solve optimization problems. Transform cost function into energy function.

Construct HN parameters from energy function. Init HN randomly. Can only find a local minimum.

9 Recurrent networks

Well suited for solving differential equations numerically by training of appropriate RN.

9.0.0.42 Cooling law General recursive formula

$\xi(t_i) = \xi(t_{i-1}) - p(\xi(t_{i-1}) - \xi_{env})\Delta t$. Calculate $\xi(t_i)$ with NN with only one neuron. $f_{net}^{(l)}(w, x(t_i)) = -p\Delta t x(t_i)$ with bias $\Phi^{(l)} = -p\xi_{env}\Delta t$ and $f_{act}^{(l)}(\cdot) = y_{act}^{(l)}(t_{i-1}) + y_{net}^{(l)}(t_i) - \Phi^{(l)}$

9.0.0.43 Representation of DiffEqs Decompose N -th order differential equation into N first order differential equations. Approximate differential quotient by difference quotient. Transform first order differential equations into recursive equations using difference quotient. Network input and activation for first $N - 1$ neurons are:

$$y_{net}^{(l_m)}(t_i) = y_{m+1}(t_i)\Delta t,$$

$y_{act}^{(l_m)}(t_i) = y_{act}^{(l_m)}(t_{i-1}) + y_{net}^{(l_m)}(t_i) - \Phi^{(l_m)}$. 2nd to last neuron depends on the chosen type of DE. The last neuron tracks time by $t_i = t_{i-1} + \Delta t$.

9.0.0.44 Learning Trained similarly to MLP. Unfolded over time between training patterns to eliminate feedback. One neuron created for each time interval