Technical Neural Networks Topics:

1. Basis
   - Biological Background
   - Learning Paradigms
   - History of Neural Networks
   - Perzeptron

Neural Network Paradigms

→ For every paradigm Structure, Functionality and Learning Rule has to be learned

2. Supervised Learning
   - 2.1 MLP
   - 2.2 RBF
   - 2.3 ART
   - 2.4 Neocognitron
   - 2.5 SVM
   - 2.6 Recurrent MLPs
     - Elman-Network
     - Jordan-Network

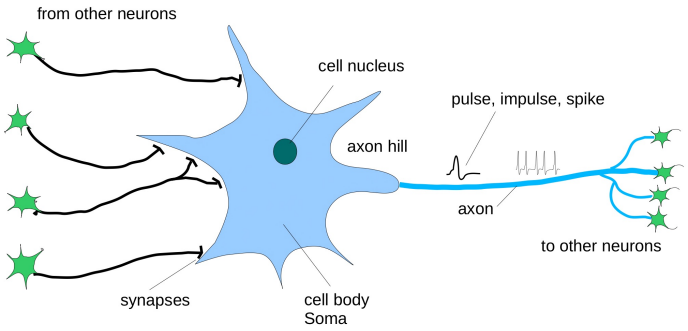
3. Unsupervised Learning
   - 3.1 SOM
   - 3.2 Multi-SOM
   - 3.3 Neural-Gas
   - 3.4 Multi-Neural-Gas
   - 3.5 k-means clustering
   - 3.6 Learning Vector Quantization

4. Reinforcement Learning
   - Reward Functions
   - Value Functions
   - Learning a policy

1. Basis

Biological Background

Biological Neurons:



Principles:

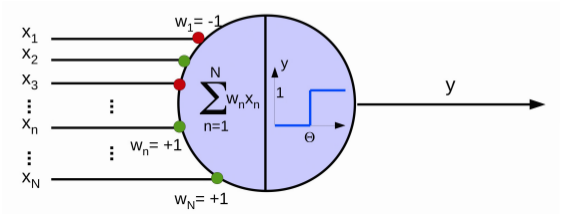
- High interconnectivity
- Neurons structured in layers
- Neurons are activated above a certain threshold

Hebbian Learning Rule:

Connection between neurons is strengthend if both neurons are activated in the same time window

$$\Delta w_{nm} = \eta * y_m * x_n$$

Technical Neuron:



History of Technical Neural Networks

1940-1960: Beginning
→ first neural paradigms such as the Perzeptron inspired by biological background, the applicability is not yet proven

1960-1982: "Quiet Years"
→ no funding and progress in the area due to no applicable learning rule, a single perzeptron can only deal with linear separable data

1980-1990: Renaissance
→ the quiet years result in many different neural paradigms leading to a restart and explosion in ideas

1990-1999: Consolidation
→ Neural networks are used in many different areas through efficient learning algorithms

Since 2000: 2nd start-up
→ Through new paradigms and higher computational power neural networks can solve highly complex tasks

Learning Paradigms:

Supervised Learning:

Learning is performed through the utilization of a teacher value $\hat{Y}$. Given input X, the error between the output Y and $\hat{Y}$ is trained to be minimized

Unsupervised Learning:

Only data driven learning. The learner tries to map statistical properties of the input X.

Reinforcement Learning:

The learner only receives a sparse feedback to his output Y which he evaluates his action on

Applications of Neural Networks

Function Approximation:

A neural network can be used as a universal function approximator. The goal is to approximate a goal function as closely as possible given samples from it.

Pattern Recognition:

The goal is to map the input to a discrete set of values which classifies the input to pre-defined or self-learned classes.

Binary Classification

Task: Assign to an input a binary class value

Further assumption: linear separable data

Goal of Learning:

Find parameters α of function/classifier f_α, such that:

1. **Minimize Risk:** $$R(\alpha) = \int \frac{1}{2} |y - f_\alpha(\vec{x})| \, dP(\vec{X}, y)$$
   → usually not observable

   Empirical Risk: $$E_\alpha = \frac{1}{L} \sum_{i=1}^{L} \frac{1}{2} |y - f_\alpha(\vec{x}_i)|$$
   → Training error
   → Approximation of the risk, can deviate due to overfitting etc.

2. **Structural Risk Minimization**

   Minimize the gap between the risk and the empirical risk

   $$R(\alpha) \leq E_\alpha + \theta(h, L, \eta)$$

   Confidence Term Θ:

   $$\theta(h, L, \eta) = \sqrt{\frac{h\left(\log\left(\frac{2L}{h}\right)+1\right) - \log\left(\frac{\eta}{4}\right)}{L}}$$

   L: Number of training patterns
   h: VC capacity of function class f_α
   η: Probability

   For h<L: This inequality holds with $(1-\eta)$

   Reduce Θ:
   1. Take a larger amount of training data L
   2. keep VC-dimension low, prevent overfitting

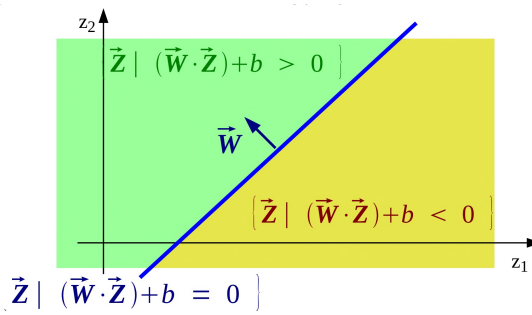
VC Capacity h: Measure to quantify the expressive power of a class of functions

Maximal number of points that can be shattered by the class of functions.

Shattering: All dichotomies of h points can be realized

1. Show there exist a set of h points that can be shattered
2. Show there exist no set of (h+1) points

Hyperplane: $\left\{ \vec{Z} \mid (\vec{W} \cdot \vec{Z}) + b = 0 \right\}$



$\left\{ \vec{Z} \mid (\vec{W} \cdot \vec{Z}) + b = 0 \right\}$

In 2D:

$$(\vec{W} \cdot \vec{Z}) + b = 0 \qquad b \in \mathbb{R}^1 \quad \vec{W} \in \mathbb{R}^2 \quad \vec{Z} \in \mathbb{R}^2$$

$$w_1 z_1 + w_2 z_2 + b = 0$$

$$w_2 z_2 = -w_1 z_1 - b$$

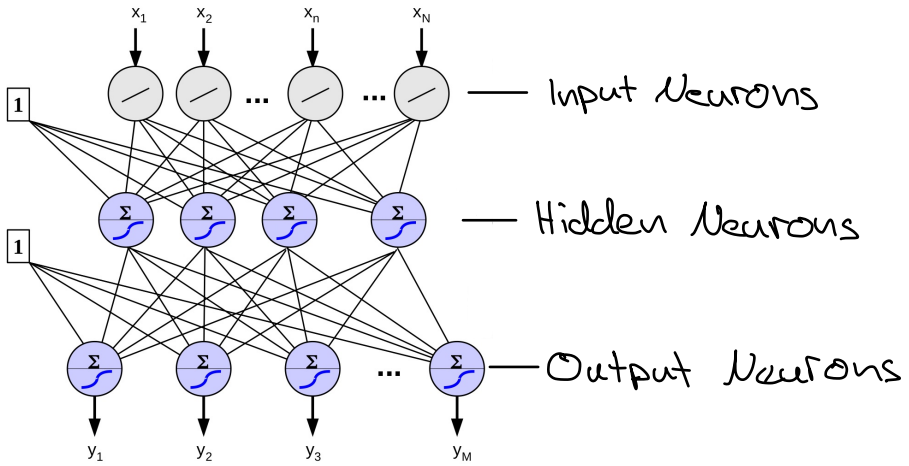
$$z_2 = -\frac{w_1}{w_2} z_1 - \frac{b}{w_2} \qquad \text{case } w_2 \neq 0$$

Binary classifier: $f_{W,b}(\vec{Z}) = \text{sign}\left((\vec{W} \cdot \vec{Z}) + b \right)$

2. Supervised Learning

Multi-Layer Perzeptron

Structure:



Learning Rule: Backpropagation of Error

Error Function:

$$ {}^{p}E = \frac{1}{2}\sum_{m=1}^{M}({}^{p}\hat{y}_m - {}^{p}y_m)^2 $$

Weight Change:

$$ \Delta w_{ij} = \eta \cdot \delta_j \cdot \widetilde{out}_i $$

↖ Delta-rule

Output Neuron:

$$ \delta_m = (\hat{y}_m - y_m) \cdot f'(net_m) $$

Hidden Neuron:

$$ \delta_h = (\sum_{k=1}^{K} \delta_k w_{hk}) \cdot f'(net_h) $$

Derivation:

Calculate the gradient w.r.t the error function to minimize the error through gradient descent

Output Neuron:

$$ \Delta W_{nm} = -\eta \frac{\partial {}^{p}E(W_{nm})}{\partial W_{nm}} \quad | \text{Definition} $$

$$ \frac{\partial {}^{p}E}{\partial {}^{p}net_m} \cdot \frac{\partial net_m}{\partial W_{nm}} \quad | \text{Chain rule} $$

$$ \frac{\partial {}^{p}E}{\partial {}^{p}y_m} \cdot \frac{\partial f({}^{p}net_m)}{\partial {}^{p}net_m} \quad | \text{Chain rule} $$

$$ \delta_m = ({}^{p}\hat{y}_m - {}^{p}y_m) \cdot f'({}^{p}net_m) $$

$$ \Delta W_{nm} = \eta \cdot ({}^{p}\hat{y}_m - {}^{p}y_m) \cdot f'({}^{p}net_m) \cdot {}^{p}\widetilde{out}_m $$

Hidden Neurons:

$$\Delta w_{hm} = -\eta \frac{\partial {}^{p}E(w_{hm})}{\partial w_{hm}}$$

$$\frac{\partial {}^{p}E(w_{gh})}{\partial {}^{p}net_h} \cdot \frac{\partial {}^{p}net_h}{\partial w_{gh}}$$ | chain rule

| only input g contributes

$$-{}^{p}\delta_h = \frac{\partial {}^{p}E}{\partial out_h} \cdot \frac{\partial out_h}{\partial {}^{p}net_h}$$ | chain rule

$$\frac{\partial f({}^{p}net_h)}{\partial {}^{p}net}$$

$$\sum_{k=1}^{K} \frac{\partial E^{p}}{\partial net_k} \cdot \frac{\partial net_k}{\partial out_h}$$

$$\frac{\partial \left(\sum_{j=0}^{H} {}^{p}out_j \, w_{jk}\right)}{\partial w_{jk}}$$ | only output h contributes

$$-\sum_{k=1}^{K} \delta_k \cdot w_{hk}$$ | contributes of h in next layer

$$\delta_h = \left(\sum_{k=1}^{K} \delta_h \cdot w_{hk}\right) \cdot f'({}^{p}net_h)$$

$$\Delta w_{hm} = \eta \cdot \left(\sum_{k=1}^{K} \delta_h \cdot w_{hk}\right) \cdot f'({}^{p}net_h) \cdot {}^{p}out_g$$

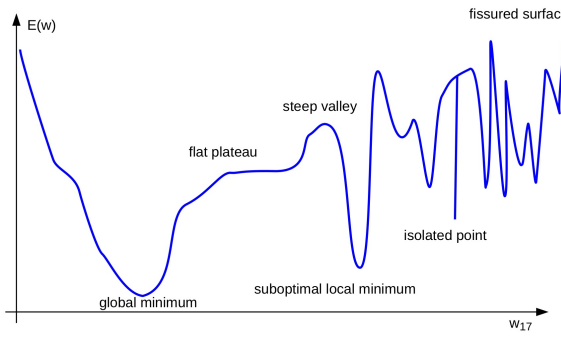
Transfer Functions:

	Function:	Derivation:
Logistic	$f_{logist}(z) = \frac{1}{1+e^{-z}}$	$f'_{logist}(z) = f_{logist}(z) \cdot (1.0 - f_{logist}(z))$
Tanh	$f_{tanh}(z) = \tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$	$f'_{tanh}(z) = \tanh'(z) = 1.0 - \tanh^{2}(z)$
ReLu	$f_r(z) = \max(0,z)$	$f' = \begin{cases} 1 & \text{, for } z > 0 \\ 0 & \text{, else} \end{cases}$
Softplus	$f_{softplus}(z) = \ln(1+e^{z})$	$f'_{softplus}(z) = \frac{1}{1+e^{-z}}$

Implementation Details:

Network Choice:

$$\frac{\#\,\text{Patterns}}{\#\,\text{Neurons}} \approx 3$$



Training Procedure:

1. **Single step learning**: Weight update for single patterns

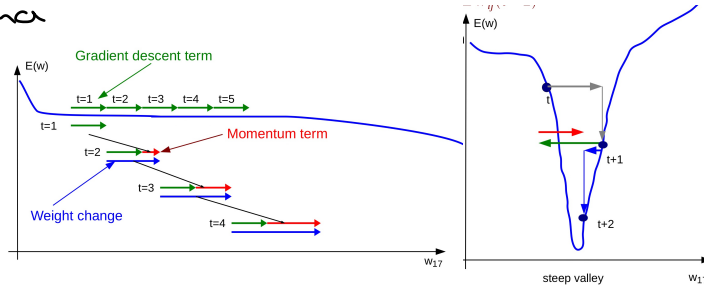
   Batch Learning: Weight update on a basis of a batch of patterns

2. Present patterns in a new random sequence every epoch

3. **Momentum**: Taking the momentum from the last step helps dealing with plateaus an isolated local minima

$$\Delta w_{ij}(t) = \eta \cdot \delta_j \cdot {}^{p}\widetilde{out}_i + \alpha \cdot \Delta w_{ij}(t-1)$$

Weight change — Gradient descent term — Momentum term



4. **Early Stopping**: Stop traing when the model starts overfitting or progress stagnates

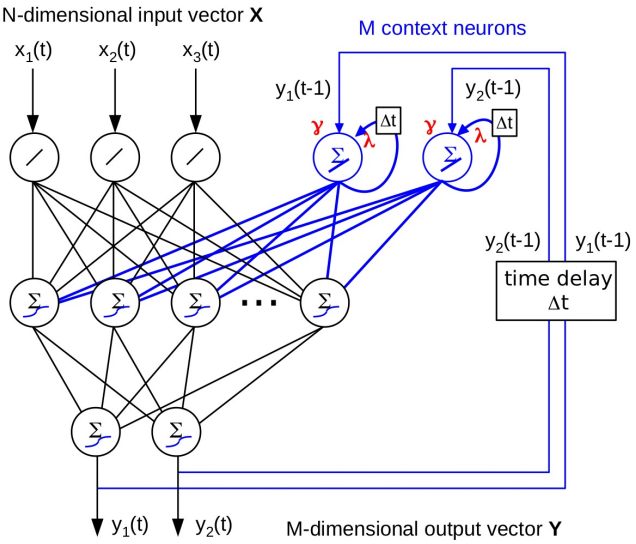
Recurrent MLPs

Based on normal MLPs with extra feedback connections. This introduces an explicit time delay. The result is a dynamic behavior. Static weights and input generates a sequence of outputs.

Jordan Networks

Network with additional context neurons that feedback the complete output of the network. The context neurons can have an additional direct feedback to themselfs.

Structure:

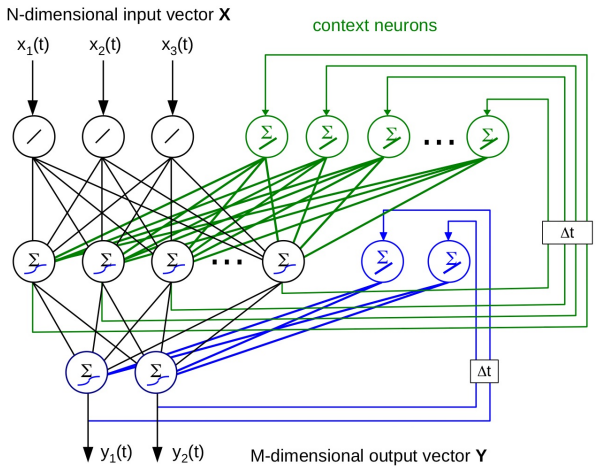


Typically: $\gamma = 1$, $\lambda = 0$

Elman Networks

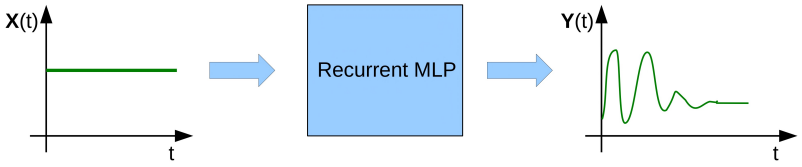
Network with feedback connections from every layer to the very same layer

Structure:

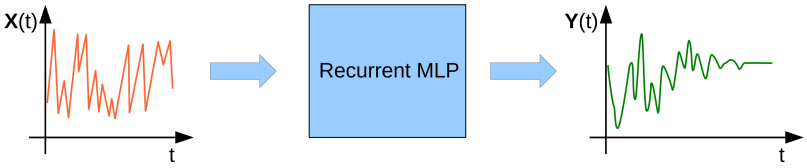


Operation Modes:

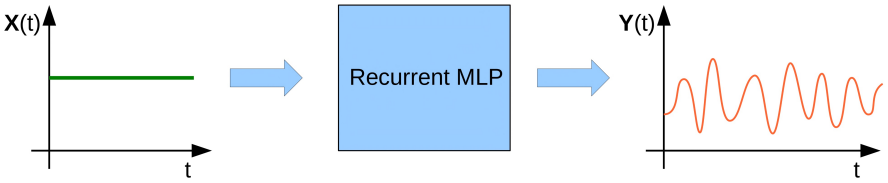
Mode 1: Input is constant ⟶ Output converges



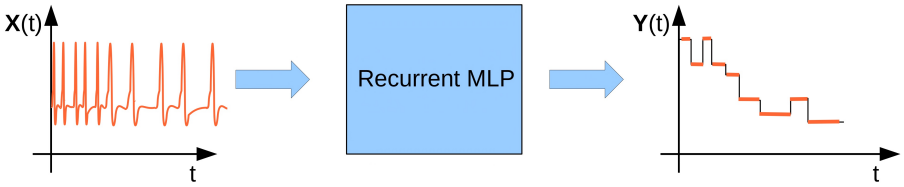
Mode 2: Input varying in time ⟶ Output converges



Mode 3: Input is constant ⟶ Output generates a sequence of states, values or continuous time series

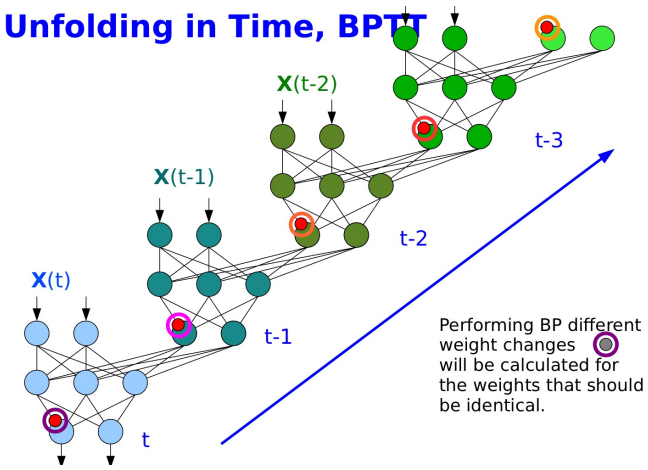


Mode 4: Input varying in time ⟶ Output generates a sequence of states, values or continuous time series



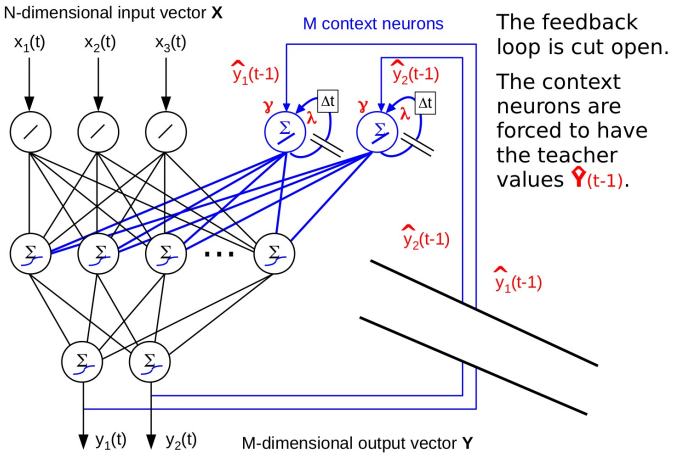
Backpropagation Through Time

The network is unfolded over time for a fixed number of timesteps. Backpropagation is applied to each unfolded network and the weight changes are accumulated.
→ high computational / memory efforts



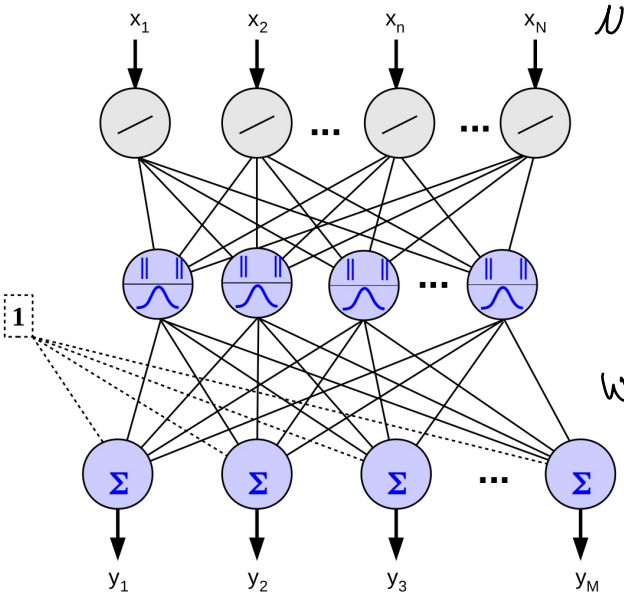
Open Loop Training

The feedback connections are cut open and the context neurons are artificially set to the teacher values. For Elman networks this technique is more complex, since there are no teacher values for the hidden neurons. Training is performed using Backpropagation



Radial Basis Function Networks

Structure:



N-dim. Input X

N Input Neurons

K RBF-Neurons

Weights w_{km}

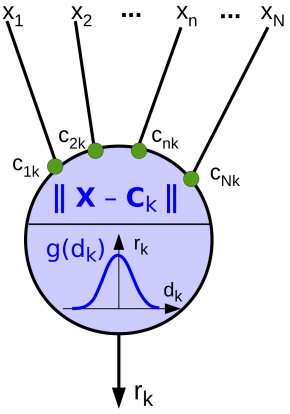
M Output Neurons

RBF Neuron:

Defined by center vector c_k

1. Calculate distance between input and center:

$$d_k = \| X - C_k \|$$



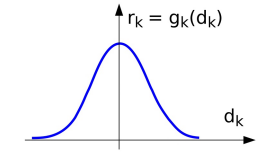
2. Calculate radial function g_k: $r_k = g(d_k)$
→ nonlinear, uni-modal function

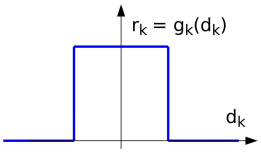
Radial Basis Functions:

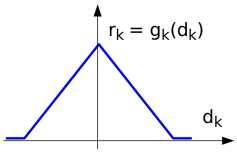
Gaussian Bell:

$$r_k = e^{-\frac{d_k^2}{2s_k^2}} = e^{-\frac{\|X-C_k\|^2}{2s_k^2}}$$

with parameter s_k







RBF Layer:

Accumulation of K RBF neurons

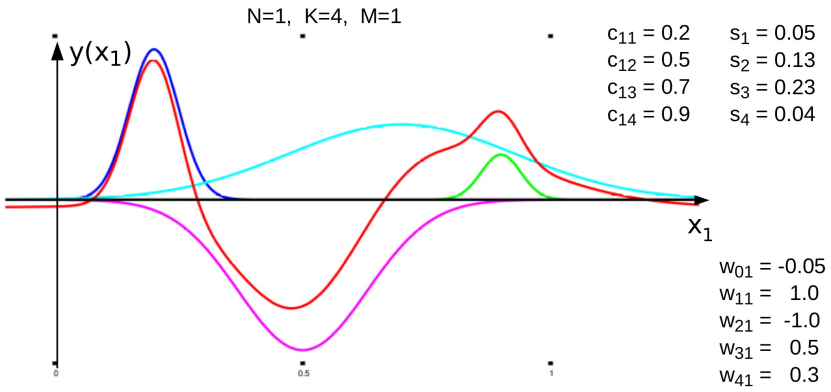
Output Layer

M neurons that calculate the weighted sum of the RBF neurons output:

$$y_m = \sum_{k=1}^{K} w_{km} \cdot r_k$$

Functionality:

Universal Function Approximator:



Adjustable Parameters

c_{nk}: centers ⟶ Mean of Gaussia
s_k: deviation ⟶ width
w_{km}: weight ⟶ height

Setting the RBF neurons

Completely uninformed: Cover input space uniformly

Input Data driven:

Centers are choosen w.r.t. the distribution of input data:

- Subset of training data
- Statistical Analysis
- Clustering
- Learning

Error driven:

Centers are adjusted w.r.t. an error function using the teacher values and gradient descent.

Weight Adjustment

Simplifications

1. The RBF layer is fixed, fixed c_k and s_k
2. Type of solution is split in three cases:
   - $P < k$: not interesting
   - $P = k$: Interpolation
   - $P > k$: Approximation, Regression
3. Output size $M = 1$

P=k: Interpolation

Teacher values $\hat{y}_i$ can be matched exactly by the output

Linear Equation System

$$
\begin{aligned}
{}^1\hat{y}_m &\stackrel{!}{=} {}^1y_m(w_{km}) = \sum_{k=1}^{K} w_{km} \cdot {}^1r_k \\
{}^2\hat{y}_m &\stackrel{!}{=} {}^2y_m(w_{km}) = \sum_{k=1}^{K} w_{km} \cdot {}^2r_k \quad \Leftrightarrow \\
&\vdots \\
{}^P\hat{y}_m &\stackrel{!}{=} {}^Py_m(w_{km}) = \sum_{k=1}^{K} w_{km} \cdot {}^Pr_k
\end{aligned}
$$

$$
\underset{P\times 1}{\hat{\underline{Y}}} \stackrel{!}{=} \underset{P\times 1}{\underline{Y}} = \underset{P\times K}{\underline{\underline{R}}} \cdot \underset{K\times 1}{\underline{W}}
$$

$$
\underline{\underline{R}}^{-1} \cdot \hat{\underline{Y}} = \underline{\underline{R}}^{-1} \underline{\underline{R}} \cdot \underline{W} = \underline{\underline{1}} \cdot \underline{W}
$$

$$
\underset{K\times P}{\underline{\underline{R}}^{-1}} \cdot \underset{P\times 1}{\hat{\underline{Y}}} = \underset{K\times 1}{\underline{W}}
$$

⇒ Weights can be derived directly analytically

P>k: Approximation/Regression

R is not quadratic and thus not invertible

Approximation:

Use Moore-Penrose-Pseudo-Inverse: $\underline{\underline{R}}^{+} = \left(\underline{\underline{R}}^{T} \cdot \underline{\underline{R}}\right)^{-1} \cdot \underline{\underline{R}}^{T}$

$$
\Rightarrow \quad \underset{K\times P}{\underline{\underline{R}}^{+}} \cdot \underset{P\times M}{\hat{\underline{Y}}} = \underset{K\times M}{\underline{W}}
$$

Regression

Learn the weights: $\Delta w_{km} = \eta \cdot \delta_m \cdot {}^pr_k = \eta \cdot \left({}^p\hat{y}_m - {}^py_m\right) \cdot {}^pr_k$

Learning Vector Quantization

Structure: Neurons structured in a grid like the SOM

Learning Rule:

0: Initialize SOM

Define: Task, Architecture, Patterns, Learning parameters

Initialize center vectors:

1. completely random
2. subset of training patterns
3. Initialize by prior knowledge

1: Choose stimulus and apply to SOM

2: Calculate response: $r_k = \|C_k - {}^pX\|_2$

3: Determine winner

Winner takes it all: Neuron with the closest distance

$$\|C_i - {}^pX\|_2 \leq \|C_j - {}^pX\|_2 \quad \text{for } i,j = 1 \dots K$$

4: Calculate weight changes ΔC

LVQ 1:

$\Delta C_i = \eta(t) * (+1) * ({}^pX - C_i)$ if $Y_i = {}^p\hat{Y}$ Pulled towards, if correct

$\Delta C_i = \eta(t) * (-1) * ({}^pX - C_i)$ if $Y_i \neq {}^p\hat{Y}$ Pushed away, if not

$\Delta C_k = 0$ if $k \neq i$ only winner neuron is trained

LVQ 2.1. Consider the two best neurons Y_i and Y_j

Learn. if: 1. Classes are different: $Y_i \neq Y_j$

2. One is correct: $Y_i = \hat{Y} \vee Y_j = \hat{Y}$
3. Stimulus is close to the border of Y_i and Y_j

$\Delta C_i = \eta(t) * (+1) * ({}^pX - C_i)$

$\Delta C_j = \eta(t) * (-1) * ({}^pX - C_j)$

else $\Delta C_k = 0$

LVQ 3: Addition to LVQ 2.1

If both classes are correct: $\Delta C_i = \eta(t) * \alpha * ({}^pX - C_i)$, $\Delta C_j = \eta(t) * \alpha * ({}^pX - C_j)$

α: learning factor

5: Update step: $\text{new}C_k = \text{old}C_k + \Delta C_k$

6: Termination criterium, else continue learning (step 1)

Criteriums: Pre-defined performance, Convergence, Time

7: Labeling

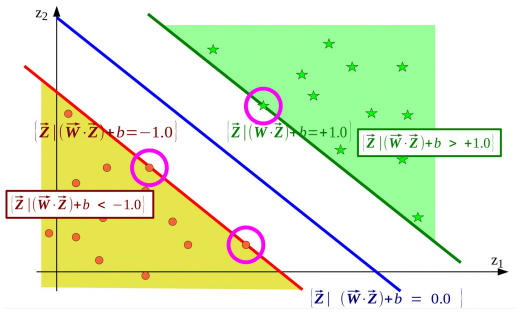
Give each neuron a semantic meaning (label / place coding)

Support Vector Machines

Task: Binary classification from linear separable data

Optimal linear separating hyperplanes

Hyperplane laying in the middle of the Maximum Margin corridor



Width: $$\left(\frac{\vec{W}}{\|\vec{W}\|}\cdot(\vec{Z}_* - \vec{Z}_o)\right) = \frac{2}{\|\vec{W}\|}$$

This corridor is spanned by multiple support vectors

Goal of Learning: Find the optimal linear separating hyperplane

Let $X_* \longrightarrow 1$ and $L_o \longrightarrow -1$

1. Maximize the margin

Maximize the margin $\frac{2}{\|\vec{W}\|} \Leftrightarrow$ Minimize $\|\vec{W}\| \Leftrightarrow$ Minimize $\frac{1}{2}\|\vec{W}\|^2$

2. Classify all patterns correctly

Decision function: $$f_{W,b}(\vec{Z}) = \text{sign}\left((\vec{W}\cdot\vec{Z})+b\right)$$

For all training values X_i:

$$\begin{aligned} f(\mathbf{X}_*) &= +1 \\ f(\mathbf{X}_o) &= -1 \end{aligned} \Longleftrightarrow \begin{aligned} y_i\left((\vec{W}\cdot\vec{X}_i)+b\right) &\geq +1.0 \\ y_i\left((\vec{W}\cdot\vec{X}_i)+b\right)-1 &\geq 0.0 \end{aligned}$$

Lagrange Formalism: $$L(\vec{W},b,\vec{\alpha}) = \frac{1}{2}\|\vec{W}\|^2 - \sum_{i=1}^{L}\alpha_i\left(y_i\left((\vec{W}\cdot\vec{X}_i)+b\right)-1\right)$$

α_i: Lagrange-Multipliers

Minimize Lagrange function L w.r.t W and b

Maximize Lagrange function L w.r.t α_i

Optimal solution at $\frac{\partial L(W,b,\alpha)}{\partial W} = \frac{\partial L(W,b,\alpha)}{\partial b} = \frac{\partial L(W,b,\alpha)}{\partial \alpha} = 0$

Optimization Problem:

Dual Representation: $$D(\vec{\alpha}) = \sum_{i=1}^{L}\alpha_i - \frac{1}{2}\sum_{i=1}^{L}\sum_{j=1}^{L}\alpha_i\alpha_j y_i y_j (\vec{X}_i\cdot\vec{X}_j)$$

Maximize w.r.t. the lagrange multipliers α_i with $\sum_{i=1}^{L}\alpha_i y_i = 0$ & $\alpha_i \geq 0$

Derivation: $\max_\alpha \min_{w,b} L(\mathbf{w},b,\alpha) = \frac{1}{2}\mathbf{w}.\mathbf{w} - \sum_j \alpha_j\left[\left(\mathbf{w}.\mathbf{x}_j + b\right)y_j - 1\right]$

$\alpha_j \geq 0,\ \forall j$

$$\left.\begin{aligned} \frac{\partial L}{\partial \mathbf{w}} = 0 \quad &\Rightarrow \mathbf{w} = \sum_j \alpha_j y_j \mathbf{x}_j \\ \frac{\partial L}{\partial b} = 0 \quad &\Rightarrow \sum_j \alpha_j y_j = 0 \end{aligned}\right\} \Rightarrow$$

$$\begin{aligned} &\text{maximize}_\alpha \ \sum_i \alpha_i - \tfrac{1}{2}\sum_{i,j}\alpha_i\alpha_j y_i y_j \mathbf{x}_i.\mathbf{x}_j \\ &\sum_i \alpha_i y_i = 0 \\ &\alpha_i \geq 0 \end{aligned}$$

Decision Function: $$f_{\vec{\alpha},b}(\vec{Z}) = \text{sgn}\left(\sum_{i=1}^{L}\alpha_i y_i (\vec{X}_i\cdot\vec{Z}) + b\right)$$

→ Scalar product between training patterns X and input Z
→ W got eliminated

Embedding in Higher Feature Space F

Motivation:

If data is not linearly separable in problem space G, it is linearly separable in a higher dimensional feature space F with a high probability

Proof: **Thm. (Cover, 1964):** the number of linearly separable dichotomies of n points in *general position* in $\mathbb{R}^d$ is

$$C(n,d) = 2\sum_{k=0}^{d}\binom{n-1}{k}$$

Find mapping Φ: $\boldsymbol{G} \xrightarrow{\Phi} \boldsymbol{F}$ $\quad \vec{Z} \xrightarrow{\Phi} \Phi(\vec{Z})$ $\quad \vec{X}_i \xrightarrow{\Phi} \Phi(\vec{X}_i)$

$$\Rightarrow D(\vec{\alpha}) = \sum_{i=1}^{L}\alpha_i - \frac{1}{2}\sum_{i=1}^{L}\sum_{j=1}^{L}\alpha_i\alpha_j y_i y_j\left(\Phi(\vec{X}_i)\cdot\Phi(\vec{X}_j)\right)$$

$$f_{\vec{\alpha},b}(\vec{Z}) = \operatorname{sgn}\left(\sum_{i=1}^{L}\alpha_i y_i\left(\Phi(\vec{X}_i)\cdot\Phi(\vec{Z})\right) + b\right)$$

Φ	problem space	feature space
space	$\boldsymbol{G} = \mathbb{R}^N$	$\boldsymbol{F} = \mathbb{R}^M$
input vectors	$\vec{Z} \in \boldsymbol{G}$	$\Phi(\vec{Z}) \in \boldsymbol{F}$
training vectors	$\vec{X}_i \in \boldsymbol{G}$	$\Phi(\vec{X}_i) \in \boldsymbol{F}$
scalar product	$(\vec{X}_i \cdot \vec{Z})$	$(\Phi(\vec{X}_i)\cdot\Phi(\vec{Z}))$

→ Embedding in high dimensional spaces is highly inefficient

Kernel Trick: Optimization problem: $D(\vec{\alpha}) = \sum_{i=1}^{L}\alpha_i - \frac{1}{2}\sum_{i=1}^{L}\sum_{j=1}^{L}\alpha_i\alpha_j y_i y_j \boldsymbol{k_\Phi}(\vec{X}_i,\vec{X}_j)$

Decision Function: $f_{\vec{\alpha},b}(\vec{Z}) = \operatorname{sgn}\left(\sum_{i=1}^{L}\alpha_i y_i \boldsymbol{k_\Phi}(\vec{X}_i,\vec{Z}) + b\right)$

Instead of embedding both vectors in F, we can calculate the the scalar product of both vectors in F without the mapping through

Mercer Kernels:

$$\left(\Phi(\vec{X}_i)\cdot\Phi(\vec{Z})\right) = \boldsymbol{k_\Phi}(\vec{X}_i,\vec{Z})$$

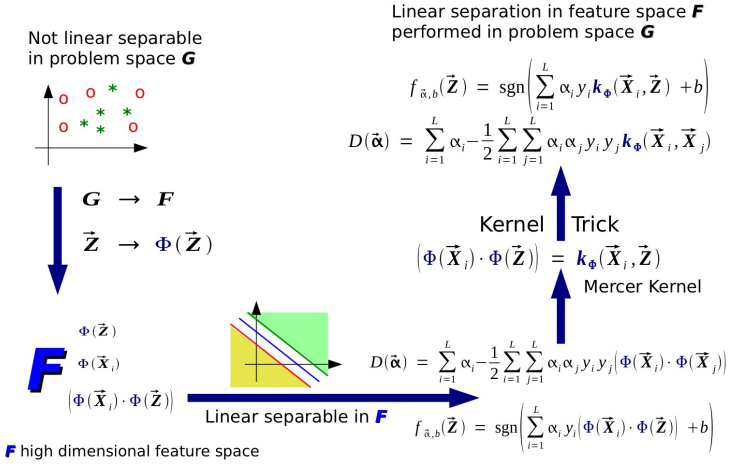
Mercer Kernels:

Linear Kernel: $\boldsymbol{k}(\vec{X}_i,\vec{Z}) = (\vec{X}_i\cdot\vec{Z})$

Polynomial Kernel: $\boldsymbol{k}(\vec{X}_i,\vec{Z}) = (\vec{X}_i\cdot\vec{Z})^d$ $\quad \boldsymbol{k}(\vec{X}_i,\vec{Z}) = (s(\vec{X}_i\cdot\vec{Z})+q)^d$

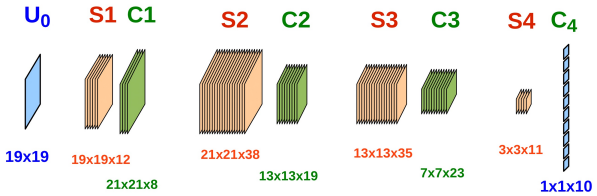
Gaussian Kernel: $\boldsymbol{k}(\vec{X}_i,\vec{Z}) = \exp\left(-\left\|\vec{X}_i-\vec{Z}\right\|^2/s\right)$

Tanh Kernel: $\boldsymbol{k}(\vec{X}_i,\vec{Z}) = \tanh(s(\vec{X}_i\cdot\vec{Z})+q)$

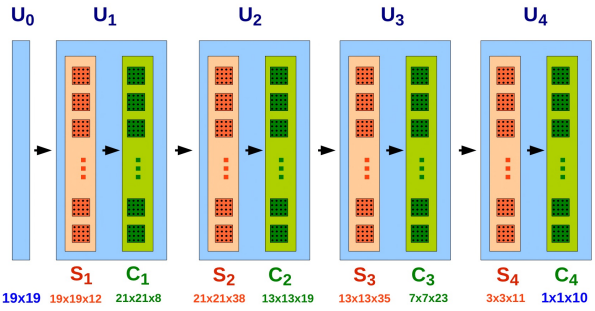


Neocognitron

Structure:



Modules - S+C Layers - S+C Planes - Neurons

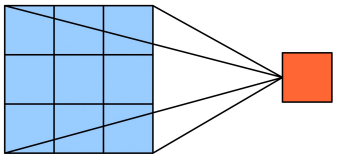


Feed forward sequence of modules (U_0, U_1, U_2, ...):

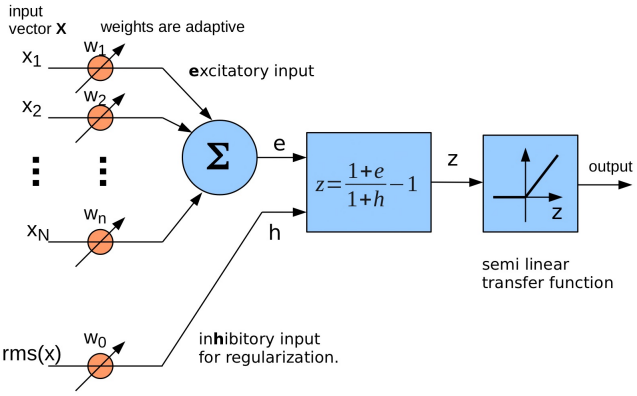
- Each module consists of two layers (**S1**, **C1**) (**S2**, **C2**), ...
  - Each layer is composed by several planes **S-planes** and **C-planes**
    - Each plane is a 2D arrangement of neurons

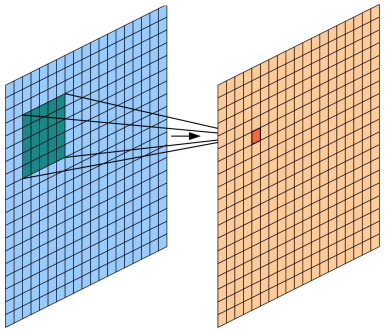
Feature Extractor Cell (S-Cells):

Cell with a n×n receptive field that is active if a specific pattern is present.



3 x 3 receptive field



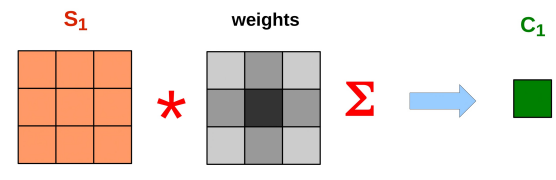


U_0: 19 x 19 input layer

U_{S1}: one 19 x 19 S-plane

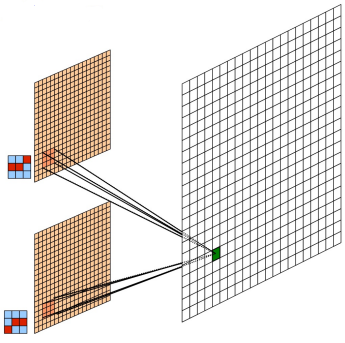
Complex Cells (C-Cells):

Cell consisting of a fixed weight matrix

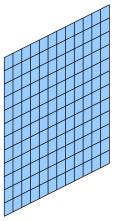


3 x 3 receptive field

3 x 3 weight matrix



Functionality



Module U_0: Input Layer consisting of a photoreceptor array

Size: 19×19

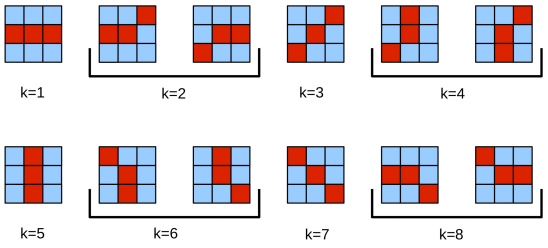
cells: $u_0(n)$, with 2D coordinates $n = (n_x, n_y)$

Module U_1:

S_1: Extracts simple line features

C_1: Implements a more robust representation

Features: $12 \times 19 \times 19 \longrightarrow 8 \times 19 \times 19$

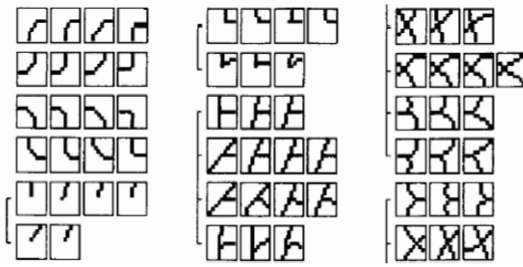


Module U_2:

S_2: Extracts more complex features like corners, interactions and curves

C_2: Robustness (fuzziness)

Features: $38 \times 21 \times 21 \longrightarrow 19 \times 13 \times 13$

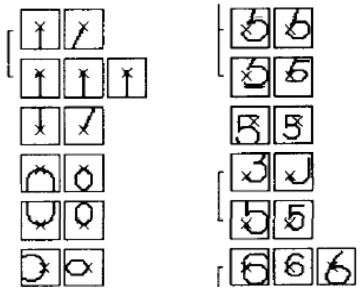


Module U_3:

S_3: More complex features

C_3: Robustness (fuzziness)

Features: $35 \times 13 \times 13 \longrightarrow 23 \times 7 \times 7$

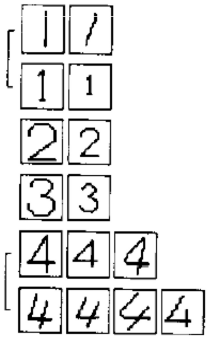


Module U_4:

S_4: High-level features

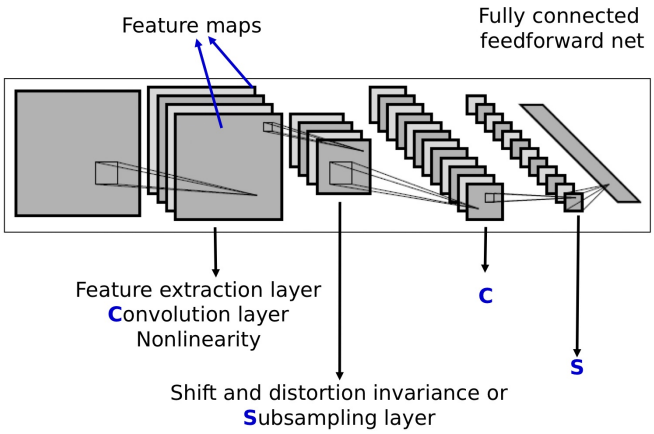
C_4: Mapping to a one out of 10 coding

Features: $11 \times 3 \times 3 \longrightarrow 10 \times 1 \times 1$



Convolutional Neural Networks

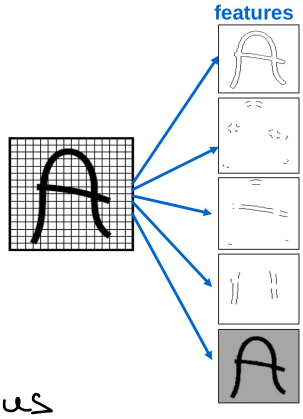
Structure:



Convolutional Layer:

Extracts features using convolution with a weight matrix and an additional nonlinear mapping.

Features: Represented through the weights
→ weights need to be identical within a single convolutional layer



Channels: Several features can be extracted parallel in a layer through an additional channel in the kernel

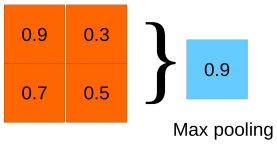
Nonlinear Activation: Typically Relu, Softplus

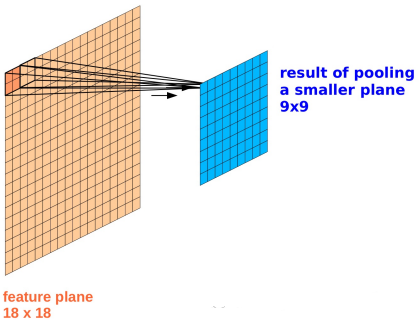
Weight Sharing: Accumulate weight changes

Subsampling/Pooling Layers

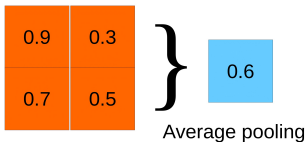
Accumulate positional close features in a single result to implement a more positional independ, sparse representation. Also enables capturing features on a larger scale.

Max Pooling: Take max value





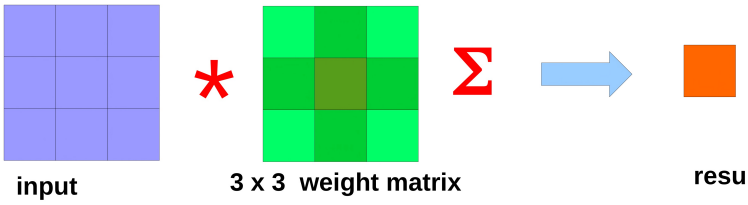
Average Pooling: Take average value

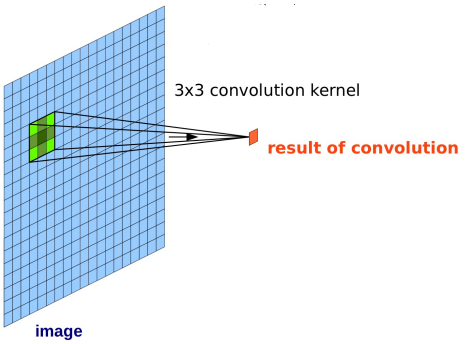


Decision: Feed-forward Net

A final MLP is used to use the extracted features for a classification task or accumulate them in a feature descriptor

Discrete Convolution: Weighted summation

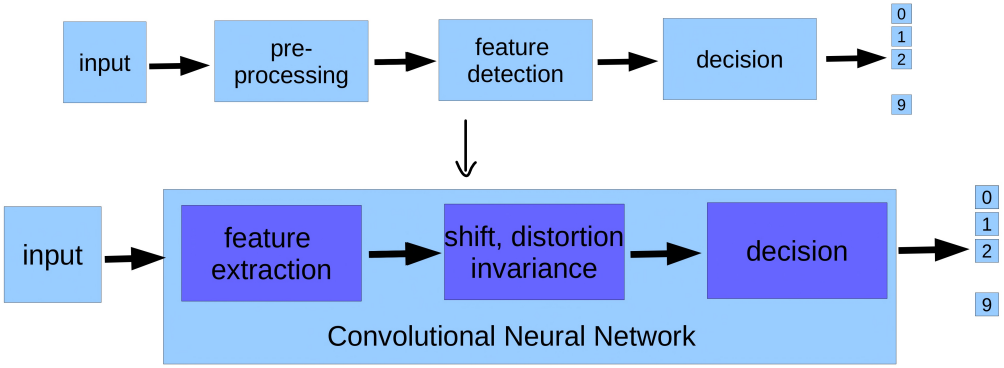




Kernels:

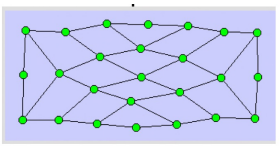
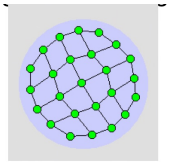
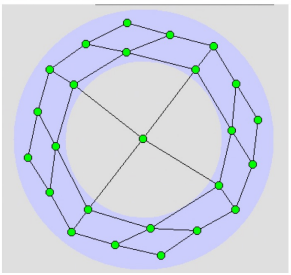
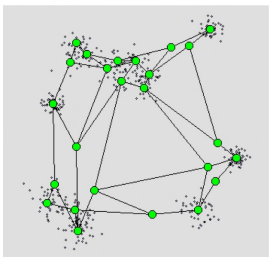
Task:

Classification tasks, predominantly in the computer vision.



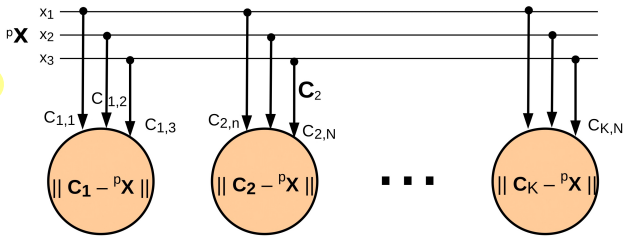
3. Unsupervised Learning

Self Organizing Feature Maps

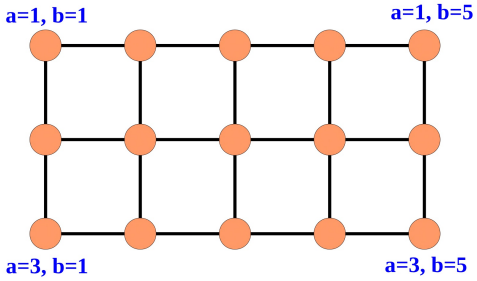





Structure:

Neurons:



Grid:



Task:

- Representation of data by a distribution / grid
  → topology preserving
  → distribution preserving
- Dimensionality reduction
- Clustering

Functionality:

1. Choose stimulus (Input vector X) and apply to SOM
2. Calculate response (Comparison to center vectors): $\| C_k - {}^pX \|_2$
3. Determine winner (closest neuron, WTA)
4. Interpret results: labeling: Neurons are labeled
   place-encoding: position in the grid reveals interpretation

Learning Rule:

0: Initialize SOM

Define: Task, Architecture, Patterns, Learning parameters

Initialize center vectors:

1. completely random
2. subset of training patterns
3. Initialize by prior knowledge

1: Choose stimulus and apply to SOM

2: Calculate response: $r_k = \|\mathbf{C}_k - {}^p\mathbf{X}\|_2$

3: Determine winner

Winner takes it all: Neuron with the closest distance

$$\|\mathbf{C}_i - {}^p\mathbf{X}\|_2 \leq \|\mathbf{C}_j - {}^p\mathbf{X}\|_2 \quad \text{for } i,j = 1 \dots K$$

4: Calculate weight changes ΔC

$$\Delta\mathbf{C}_j = \eta(t) * h(\text{dist}(i,j),t) * ({}^p\mathbf{X} - \mathbf{C}_j)$$

dist(i,j): Distance function defined on the grid

h(dist,t): Neighborhood function

5: Update step: $\text{new}\mathbf{C}_k = \text{old}\mathbf{C}_k + \Delta\mathbf{C}_k$

6: Termination criterium, else continue learning (step 1)

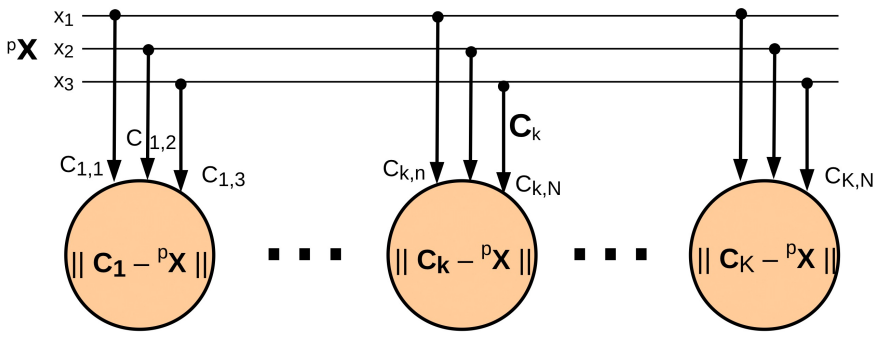
Criteriums: Pre-defined performance, Convergence, Time

7: Labeling

Give each neuron a semantic meaning (label / place coding)

Neural Gas

Structure:



Line: Neurons are structured in a sorted list w.r.t the respons
→ a mesh can be formed after training

Task:

- Representation of data by a distribution / grid
  → topology preserving
  → distribution preserving
- Dimensionality reduction
- Clustering

Functionality:

1. Choose stimulus (Input vector X) and apply to SOM
2. Calculate response (Comparison to center vectors): $\| C_k - {}^pX \|_2$
3. Determine winner (closest neuron, WTA)
4. Interpret results: labeling: Neurons are labeled
   place-encoding: position in the grid reveals interpretation

Comparison to SOM:

- Can preserve topology of more complex structures

Learning Rule:

0: Initialize Neural Gas

Define: Task, Architecture, Patterns, Learning parameters

Initialize center vectors:
1. completely random
2. subset of training patterns
3. Initialize by prior knowledge

1: Choose stimulus and apply to Neural Gas

2: Calculate response: $r_k = \|\mathbf{C}_k - {}^p\mathbf{X}\|_2$

3: Determine winner

Winner takes it all: Neuron with the closest distance

$$\|\mathbf{C}_i - {}^p\mathbf{X}\|_2 \leq \|\mathbf{C}_j - {}^p\mathbf{X}\|_2 \quad \text{for } i,j = 1 \dots K$$

4: Calculate weight changes ΔC

$$\Delta\mathbf{C}_j = \eta(t) * h(\text{dist}(i,j),t) * ({}^p\mathbf{X} - \mathbf{C}_j)$$

dist(i,j): Neurons are structured in a sorted list w.r.t their response.
Distance function is the position in the list

h(dist,t): Neighborhood function
→ Gaussian, Linear, hyperbolic
→ Should include all neurons

5: Update step: $\text{new}\mathbf{C}_k = \text{old}\mathbf{C}_k + \Delta\mathbf{C}_k$

6: Termination criterium, else continue learning (step 1)

Criteriums: Pre-defined performance, Convergence, time

7: Generate a mesh to connect the neurons

- k-nearest neighbor
- Delaunay Triangulation / Voronoi Tesselation
- Statistics from the learning process
- application from prior knowledge

8: Labeling

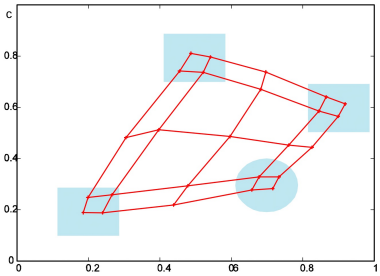
Give each neuron a semantic meaning (label / place coding)

Multi SOMs/Neural GAS

Problem SOMs/Neural GAS:

- Clustered data can not be approximated in a topology-preserving manner
- Results in neurons representing data not existing
- Neurons are close that are not close w.r.t. their semantic meaning

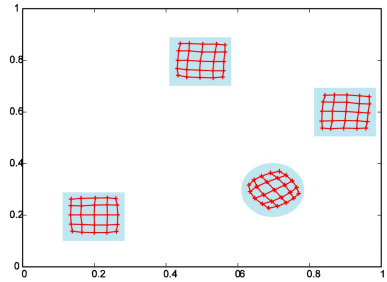
1 SOM:



Number of SOMs/Neural GAS:

- Determined through prior knowledge
- Learned during training

4 SOM:



Learning Rule:

Only the SOM/Neural Gas including the winner neuron is trained with their respective learning rule

k-means Clustering

Step 0: Get some **training data**

Step 1: **Set** the number k of codebook-vectors

Step 2: **Initialize** the codebook-vectors $\mathbf{C}_k$

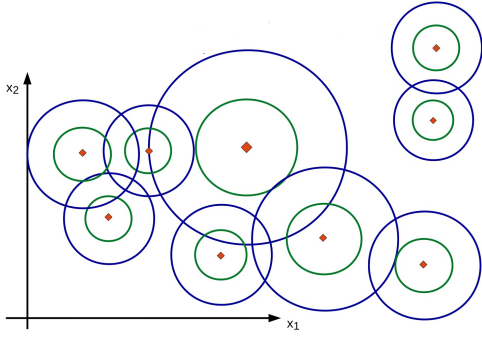
Step 3: **Assign** all patterns to the closest codebook-vector

Step 4: Calculate **centroids** (mean) for each cluster

Step 5: Make the centroids to be the new codebook-vectors $\mathbf{C}_k$

Step 6: Continue with step 3 **until convergence**

Regional and Online Learnable Fields (ROLFs)

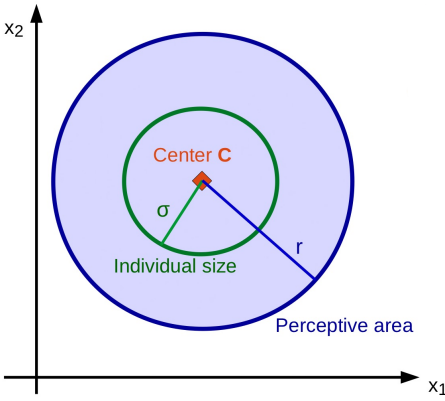


Structure:

Neurons: Center C_k
Individual Size σ_k
Perceptive Area with size $r_k = \rho \cdot \sigma_k$
→ ρ is globally defined

Neuron accepts, if: Stimulus is in perceptive area

→ Neurons are not structured in a grid



Learning Rule:

0: Initialization
Acquire data, set ρ and choose σ-strategy

1: FOR EACH pattern ${}^pX$

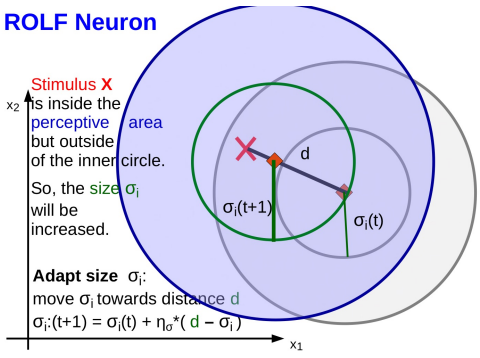
2: IF a neuron accepts

2.1 Calculate distances: $\boxed{d = \| {}^pX - C_i \|}$

2.1 Adapt neuron i with closest distance

Center c_i: $C_i(t+1) = C_i(t) + \eta_c * ({}^pX - C_i)$
Size σ_i: $\sigma_i(t+1) = \sigma_i(t) + \eta_\sigma * (d - \sigma_i)$

Perceptive area: $r_i = \rho * \sigma_i$



3: ELSE Create a new ROLF neuron

Initialize center c_k: $C_{k+1} = {}^pX$

Initialize size σ_k:

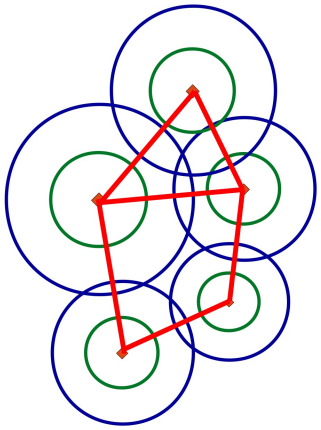
Init-σ: Use an initial value σ_{init}
Minimum-σ: Use the minimum of all σ_k
Maximum-σ: Use the maximum of all σ_k
Mean-σ: Use the mean of all σ_k

4. In rare cases: repeat for all patterns

5. Build a mesh

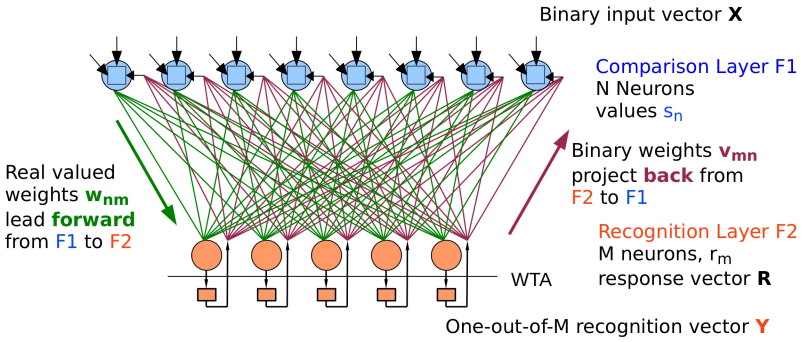
Two neurons are connected, if the perceptive areas overlap:

$$\| C_j - C_k \| \leq (\sigma_j + \sigma_k) * \rho$$



Adaptive Resonance Theory (ART-1)

Structure:



F1: Comparison Layer
F2: Recognition Layer
} Short-term memory

Forward weight matrix W: Connecting F1 to F2
→ real valued

Backward weight matrix V: Connecting F2 to F1
→ binary valued

} Long-term memory

Tasks:

- Pattern recognition
- Content-addressable memory
- Unsupervised Learning
- Clustering

Functionality:

F1 comparison layer:

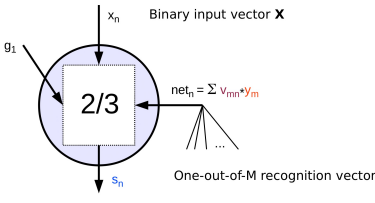
Check, if the prototypic pattern from the recognition layer fits the input vector X

Inputs:

1. Binary input vector: $X = \{x_1, x_2, x_3, \dots, x_n, \dots, x_N\}$
2. Pattern from the recognition layer:
   $net_n = \Sigma_m v_{mn} * y_m = v_{jn} * y_j = v_{jn}$
3. Control sign gain g1, identical for all neurons

Neuron output:

$$s_n = \begin{cases} 1: & \text{if } (x_n * net_n == 1) \text{ OR } (x_n * g1 == 1) \text{ OR } (net_n * g1 == 1) \\ 0: & \text{else} \end{cases}$$



F2 recognition layer:

Recognize a class and output a one-out-of-M output

Neuron output:

1. Response: $r_m = \Sigma_n w_{nm} * s_n$
2. Winner-takes-it-all:

$$y_j = \begin{cases} 1: & \text{if } r_j \geq r_m \\ 0: & \text{else} \end{cases}$$

ART-1 Functionality: Structured in 4 phases controlled by g1 & g2

1. **Recognition:** Mapping input X to output Y

   $X \longrightarrow F1 \longrightarrow S \longrightarrow W \longrightarrow F2 \longrightarrow WTA \longrightarrow Y$

2. **Comparison:** Mapping output Y back to F1

   $Y \longrightarrow V \longrightarrow F1 \longrightarrow S'$

   (two-third rule)

3. **Search:** Find adequate class in recognition layer

   Pattern Recognition:
   1. Present input X
   2. Network converges to a stable state
   3. Active neuron in F2 corresponds to class

   Content Addressable Memory:
   1. Set one F2 neuron to 1 (corresponding to the class)
   2. Corresponding pattern is created in F1

4. **Training:** Set weights of forward and backward weight matrix

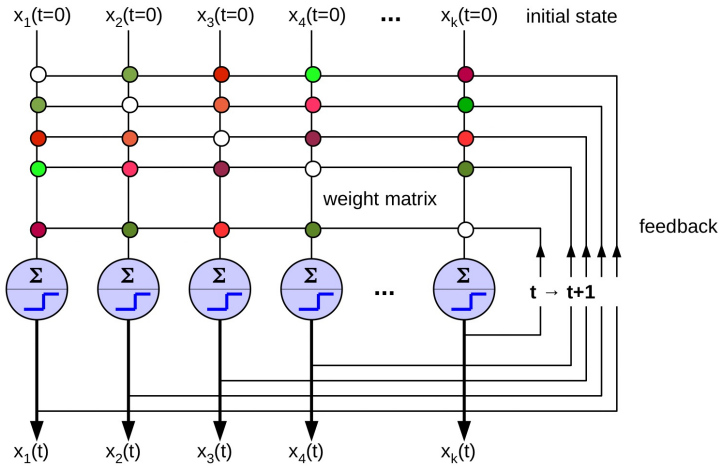
   Forward weight matrix V:
   1. Hebbian learning rule: WTA results Y as teacher or iterative adaption using response
   2. Setting directly to WTA based values

   Backward weight matrix V:

   X can be used as a teacher, weights can be set to the desired S

Hopfield Networks

Structure:



Neurons: binary states

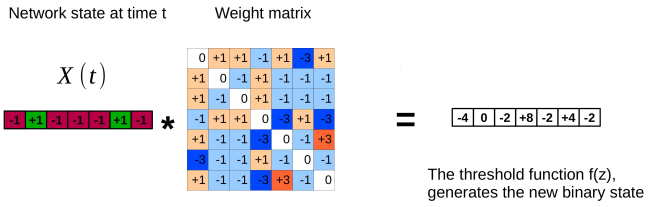
Input: Weigted sum of the input and feedback connections from every neuron

Output: Step function

Weights: k×k weight matrix

Tasks:

- content addressable memory
- associative memory
- solve optimization problem



Functionality:

State of the neurons X(t) is the processed information

1. Initialize with initial pattern X(t=0)
2. Update X(t+1) until convergence or specific t is reached:

Update rule: $x_k(t+1) = f_k\left(\sum_{j=1}^{K} w_{jk} \cdot x_j(t)\right)$, with $f_k(Z) = \begin{cases} +1 & if \quad z > \theta_k \\ -1 & if \quad z \leq \theta_k \end{cases}$

or $f_k(Z) = \begin{cases} +1 & if \quad z > \theta_k \\ X_k(t) & if \quad z = \theta_k \\ -1 & if \quad z < \theta_k \end{cases}$

Synchronous Update:

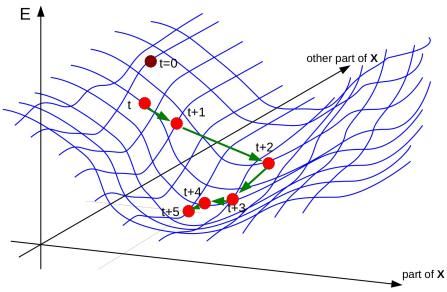
All states x(t+1) are updated in one step

Asynchronous Update:

In a timestep only one neuron is updated. The following updates depend on the already updated neurons.
The order of updates is random
→ used in Hopfield networks

Dynamics:

Energy function: $$E = -\frac{1}{2}\sum_{i}^{K}\sum_{j}^{K} w_{ij} x_i x_j + \sum_{k}^{K} \theta_k x_k$$



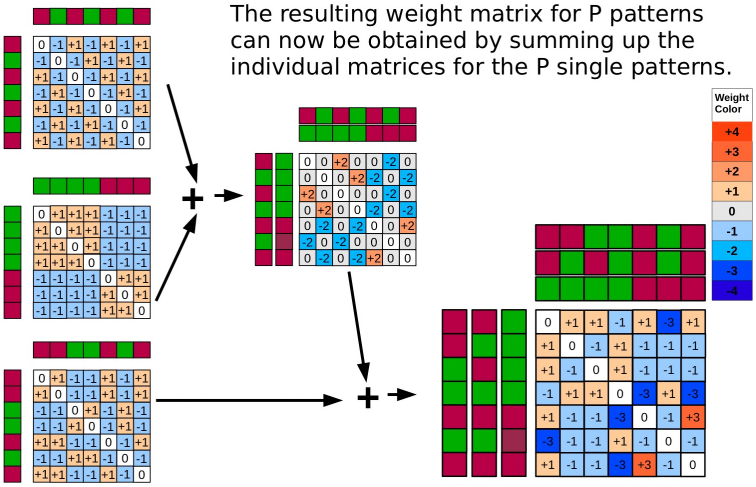
→ decreases thus network converges to a stable state after a while

Learning:

Learning is done by storing the patterns in the weight matrix

Weight x_{ik}: $$w_{ik} = \sum {}^{p}x_i * {}^{p}x_k$$

$$\begin{cases} 1 & , \text{if } x_i = x_k \\ -1 & , \text{else} \end{cases}$$



Maximal number of patterns: $Q = \alpha K$, with $\alpha \approx 0.138... \approx 0.14$

Modes:

Autoassociator Mode

Associate a pattern with itself. Weights are calculated w.r.t the components of the same pattern

$$w_{jk} = x_j \cdot x_k$$

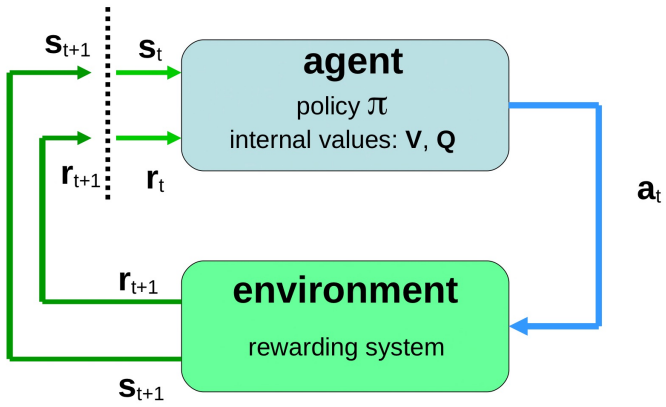
Heteroassociator

Associate pattern p with a second pattern q. Weights are calculated w.r.t to two patterns.

$$w_{jk} = {}^{p}x_j \cdot {}^{q}x_k$$

3. Reinforcement Learning

Learning Model:



Agent: Sees state S_t and decides on action a_t based on:

Policy π: A mapping from states to actions to maximize the amount of expected reward

State Value Function $V^{\pi}(s)$: $$V^{\pi}(s)=E_{\pi}\left[R_t \mid s_t=s\right]=E_{\pi}\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+1+k} \mid s_t=s\right\}$$

Expected reward w.r.t π starting from this state

Action Value Function $Q^{\pi}(s,a)$: $$Q^{\pi}(s,a)=E_{\pi}\left[R_t \mid s_t=s, a_t=a\right]=E_{\pi}\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+1+k} \mid s_t=s, a_t=a\right\}$$

Expected reward when in state s taking action a

Environment: Receives action a_t and responds with a new state S_{t+1}. Additionally a feedback as a reward following the action is provided

State Space: Space containing the possible discrete states

Action Space: Contains all possible actions performable by the agent

Rewarding System: Produces scalar rewards

Goal: Find a policy π that maximizes the reward and (hopefully) solves a given task optimal

Rewarding

True Return: $R^*_t = r^*_{t+1} + r^*_{t+2} + r^*_{t+3} + r^*_{t+4} + \ldots + r^*_T$

→ accumulated future rewards
→ in general, not obtainable

Expected Return: $R_t = r_{t+1} + r_{t+2} + r_{t+3} + r_{t+4} + \ldots + r_T$

Temporal Horizon: $T=\infty$: infinite temporal horizon
$T<\infty$: future can be split into finite temporal episodes

Discounting:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma\gamma r_{t+3} + \gamma\gamma\gamma r_{t+4} + \ldots$$

$$R_t = \sum_{k=0}^{T} \gamma^k r_{(t+1+k)} \qquad 0 \leq \gamma \leq 1.0$$

γ: Discounting Factor

Weigh impact of future rewards on the current return

Special Rewards:

1. **Pure delayed reward**

Only the last reward r_T of the episode, i.e. the result of the episode, holds a value, rest is zero, e.g.:

$r_t = r_T = 0.0$ if t<T or the outcome is draw, stalemate

$r_T = +1.0$ if game has been won

$r_T = -1.0$ if game has been lost

2. **Pure negative reward**

The agent is punished with a negative reward for every step he takes, e.g.:

$r_t = -1.0$ if t<T, pure negative reward during the episode

$r_T = 0.0$ no reward at the end of the episode

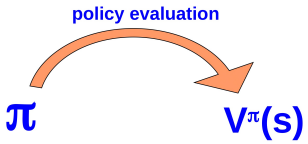
3. **Avoidance Strategy**

Almost all actions are not rewarded, except for a few with a negative reward that shall be avoided, e.g.:

$r_t = -1.0$ if $s_t \in \{$!,☹,†,☠,ϟ, $\}$ for selected situations.

$r_t = 0.0$ else, in the rest of the cases (majority).

Policy Evaluation



Determining the value functions given a specific policy π

Behavior Policy: Policy used for evaluation to explore the environment and determining the value functions

Prediction: Policy evaluation w.r.t the state value function V

Monte Carlo Policy Evaluation

Estimate the state value function by taking random actions. The value changes as soon as a return is available, i.e. a terminal state is reached.

First-visit MC evaluation:

Averages all returns after the first time state s was visited

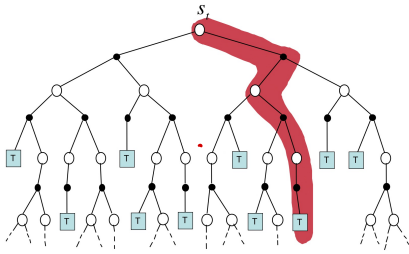
Every-visit MC evaluation:

Averages all returns for all visits of state s

Constant-α: $V(s_t) \leftarrow V(s_t) + \alpha[R_t - V(s_t)]$

Algorithm

0: Initialize policy π and state value function
1: UNTIL termination criterion:
2: | Generate episode using π
3: | FOR EACH state s in episode
4: | | $R \leftarrow$ Return following s
5: | | Append R to Returns(s)
6: | | $V(s) \leftarrow$ Average(Return(s))



Drawback: Returns are only available at the end of every episode

Temporal Difference Policy Evaluation

Instead of using the true return, approximate it through the obtained reward r_{t+1} and an estimate of the forthcoming rewards $\gamma V(s_{t+1})$.

Temporal Difference: $V(s_t) \leftarrow V(s_t) + \alpha\left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right]$

Derivation: $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+1+k} = r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+1+k+1} = r_{t+1} + \gamma V(s_{t+1})$

Control: Policy evaluation w.r.t. the action value function Q

On-policy Methods:

Evaluate/or improve the policy that is used to make decision

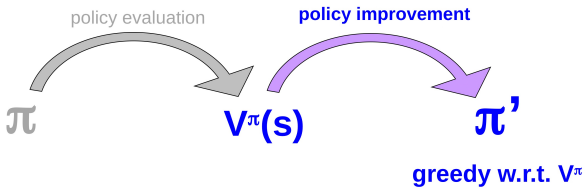
SARSA: $$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\right]$$

Off-policy Methods:

Separate policies to generate behavior (Behavior Policy) and to evaluate or improve (Estimation Policy)

Q-Learning: $$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_{t+1} + \gamma \max_a Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\right]$$

Policy Improvement



Improve the policy given the the state value function in a greedy manner w.r.t V^{π}.

Improved Policy: $\pi' \rightarrow \text{greedy}(V) \Leftrightarrow \pi(s) = \arg\max_a Q(s,a)$

$$\begin{aligned} Q^{\pi_k}(s,\pi_{k+1}(s)) &= Q^{\pi}(s,\arg\max_a Q^{\pi_k}(s,a)) \\ &= \max_a Q^{\pi_k}(s,a) \\ &\geq Q^{\pi_k}(s,\pi_k(s)) \\ &= V^{\pi_k}(s) \end{aligned}$$

$\Rightarrow$ The new policy is at least as good as the old one

General Policy Iteration

Iterate until convergence:

1. Policy Evaluation
2. Policy Improvement

Convergence: optimal policy and state value function