Neurocomputing: From neurons to systems

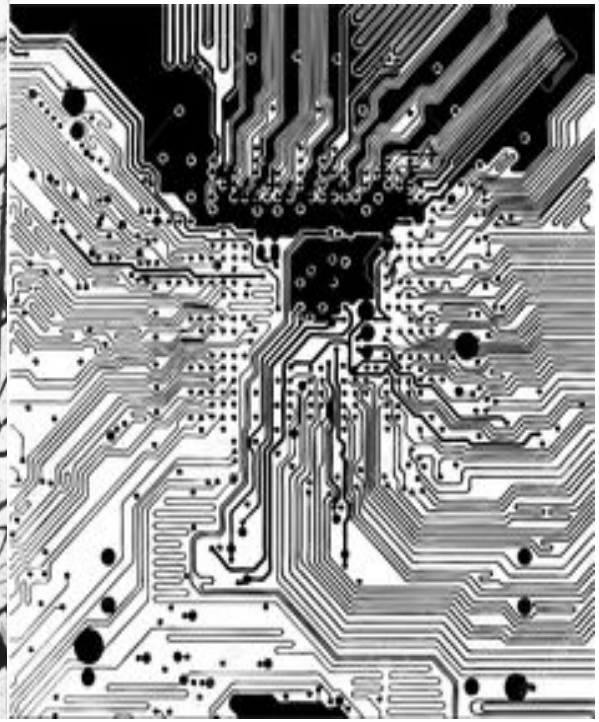
Basecamp.ai Classes

Dr. Cristian Axenie

Technische Universität München

Neuroscientific System Theory Lab

Vienna, 26-27 Jan 2017



Class overview

Supervised neural computation

- Biological neurons vs. artificial neurons
- Learning in artificial neurons
- From single neurons to neural networks
- Learning in neural networks: Error Backpropagation in Multi-Layer Neural Networks
- Supervised learning: tips and tricks

Unsupervised neural computation

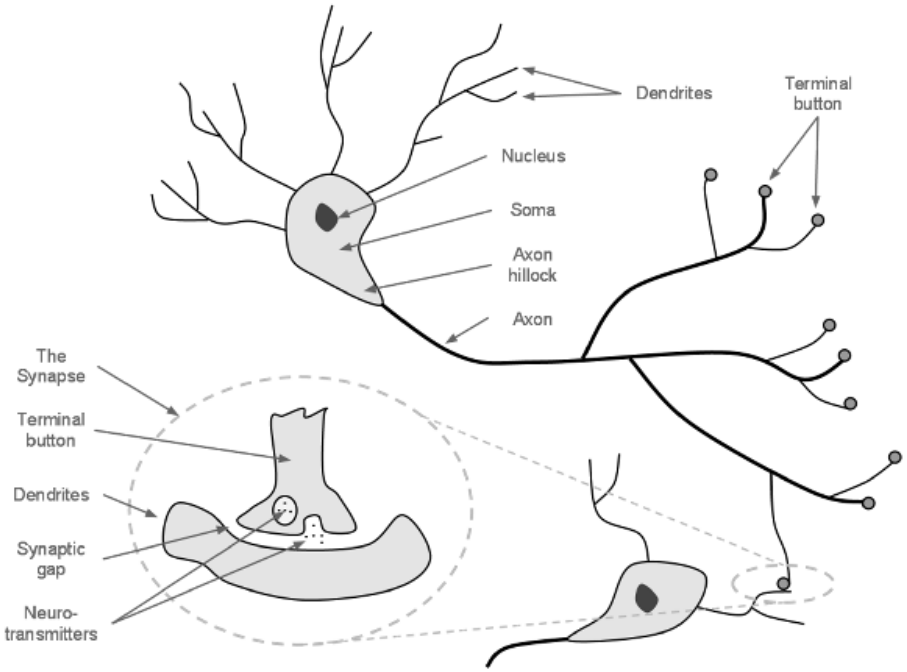
- Introduction to unsupervised learning
- Radial Basis Functions
- Vector Quantization
- Kohonen's Self-Organizing-Maps
- Hopfield Networks

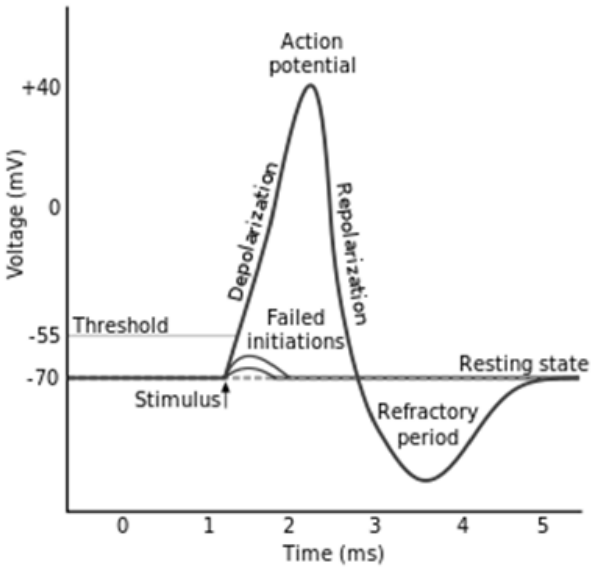
Neuromorphic engineering

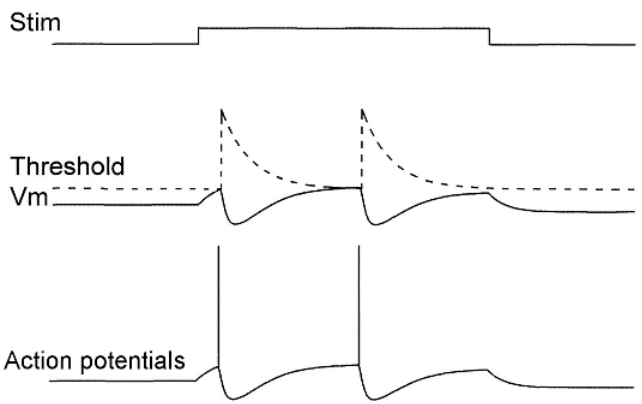
- What is neuromorphic engineering?
- Sample neurobotics applications

Supervised neural computation

- Biological neurons vs. artificial neurons







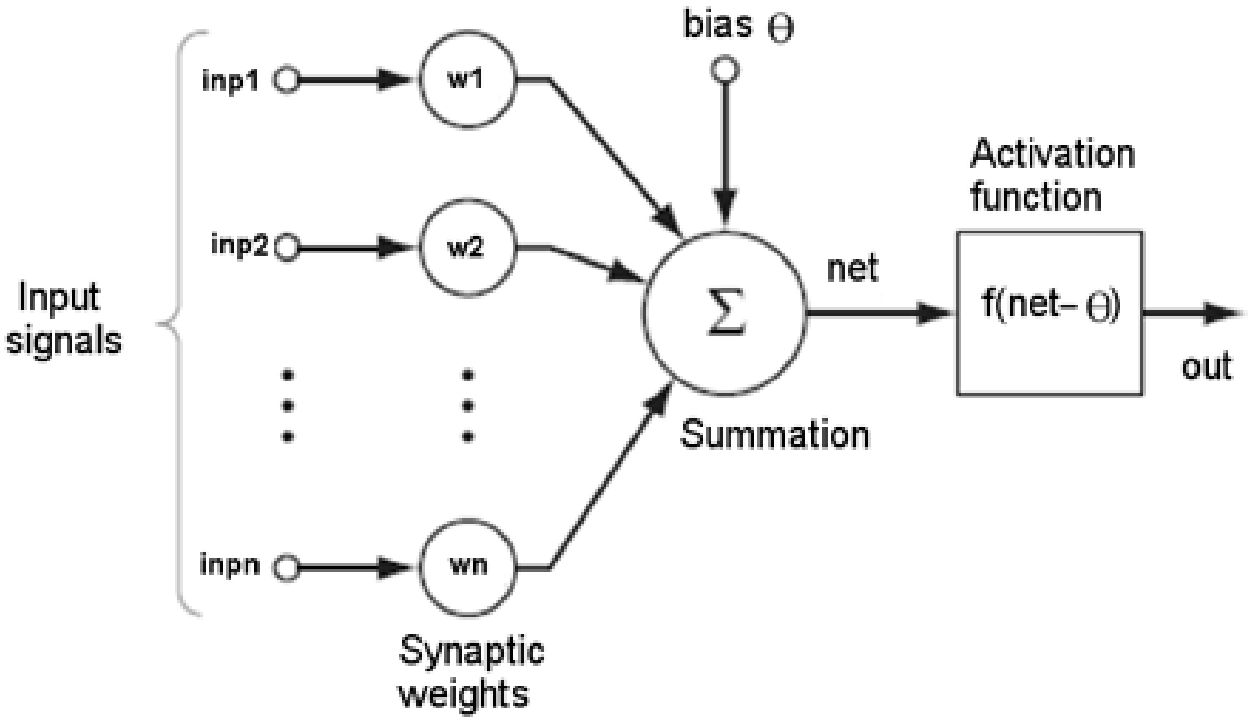
Input signal (i.e. stimulus)

Cell membrane voltage changes

Output (i.e. spike / action potential)

Supervised neural computation

- Biological neurons vs. artificial neurons

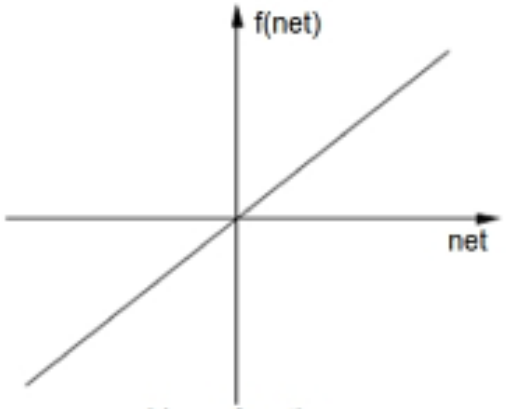


$$net^t = \sum_{i=1}^{n} inp_i\, w_i = inp_1 w_1 + inp_2 w_2 + \cdots + inp_n w_n$$

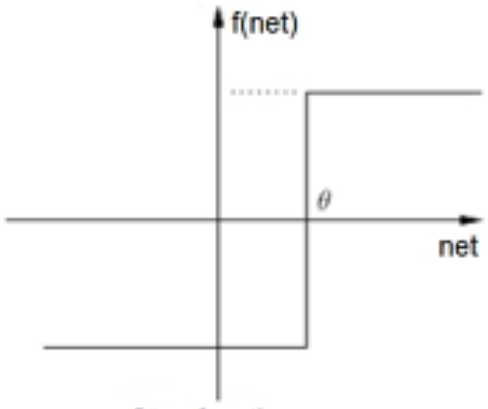
$$out^t = f(net^t - \theta)$$

Supervised neural computation

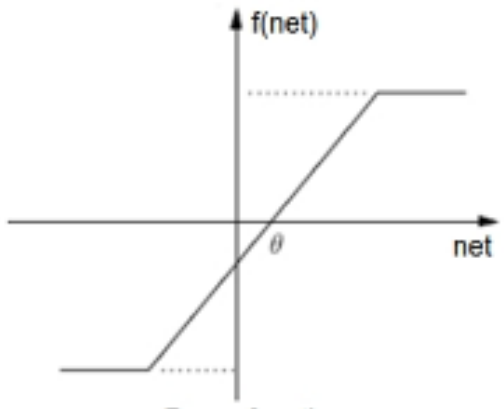
- Biological neurons vs. artificial neurons



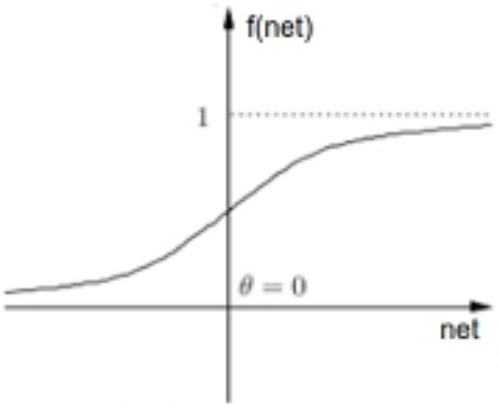
Linear function



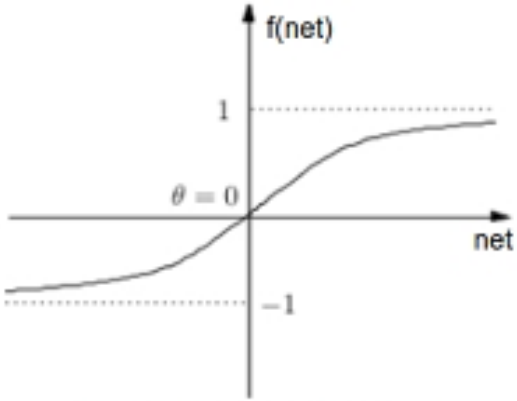
Step function



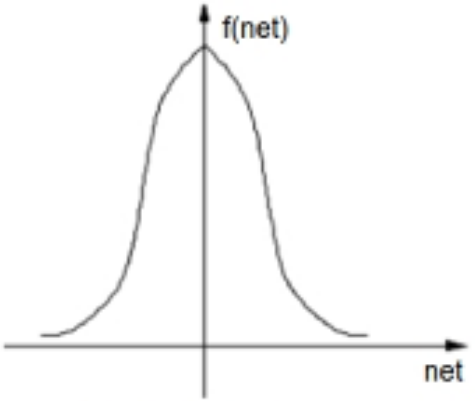
Ramp function



Sigmoid function



Hyperbolic tangent function

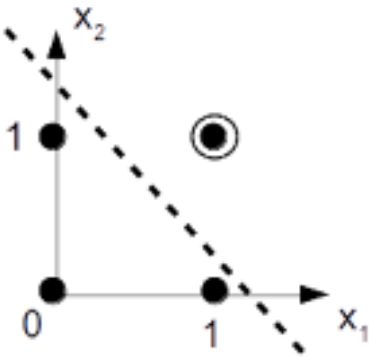


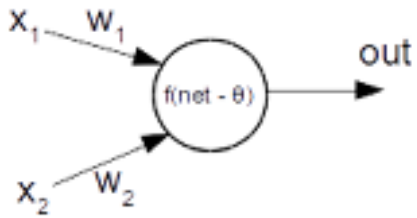
Gaussian function

Supervised neural computation

- Biological neurons vs. artificial neurons

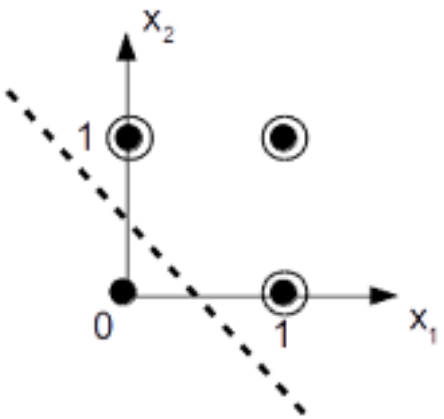
AND problem

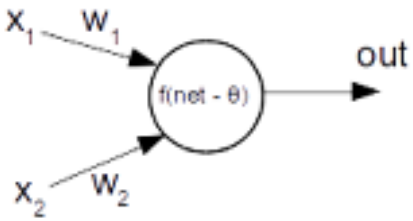




Sample solution:
$w_1 = w_2 = 0.8$
$\theta = 1$

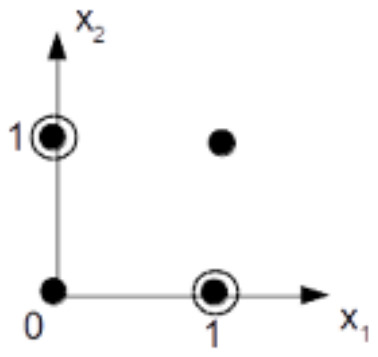
OR problem





Sample solution:
$w_1 = w_2 = 0.8$
$\theta = 0.5$

XOR problem



Sample solution:
The problem is not linearly separable → we need a **multi-layer perceptron**

Supervised neural computation

- Learning in artificial neurons

learning rule in a single neuron

We define the error signal for the entire dataset, *E*, and compute the error for each single training example, E_p, using the current set of weights w_i:

$$E = \sum_{points} \left(true_p - out_p\right)^2 \rightarrow Ep = \left(true_p - out_p\right)^2$$

Given the error signal, we compute the gradient (derivative) with respect to the input weights of a linear neuron (i.e. here for simplicity the **activation function is linear**)

$$\Delta w_i(t) = \eta G_i(t), \quad \text{with} \quad G_i(t) = \frac{\partial E_p(t)}{\partial w_i(t)},$$

which we can rewrite as (chain rule)

$$G_i(t) = \frac{\partial E_p(t)}{\partial w_i(t)} = \frac{\partial E_p(t)}{\partial out_p(t)} \cdot \frac{\partial out_p(t)}{\partial w_i(t)} \quad \text{with} \quad out_p(t) = \sum_{i=1}^{n} inp_i(t) \cdot w_i\ (t)$$

Supervised neural computation

- Learning in artificial neurons

learning rule in a single neuron

Analyzing both factors individually yields

$$(1)\ \frac{\partial E_p(t)}{\partial out_p(t)} = \frac{\partial (true_p - out_p)^2}{\partial out_p(t)} = 2 \cdot (true_p - out_p) \cdot (-1) = -2 \cdot (true_p - out_p)$$

and

$$(2)\ \frac{\partial out_p(t)}{\partial w_i(t)} = \frac{\partial \sum_{j=1} inp_j(t) \cdot w_j(t)}{\partial w_i(t)} = inp_i \qquad \text{(only for j=i the derivative exists)}$$

Combined, the **gradient in the direction of the weight w_i** is given by

$$G_i(t) = \frac{\partial E_p(t)}{\partial w_i(t)} = \frac{\partial E_p(t)}{\partial out_p(t)} \cdot \frac{\partial out_p(t)}{\partial w_i(t)} = -2 \cdot (true_p - out_p) \cdot inp_i$$

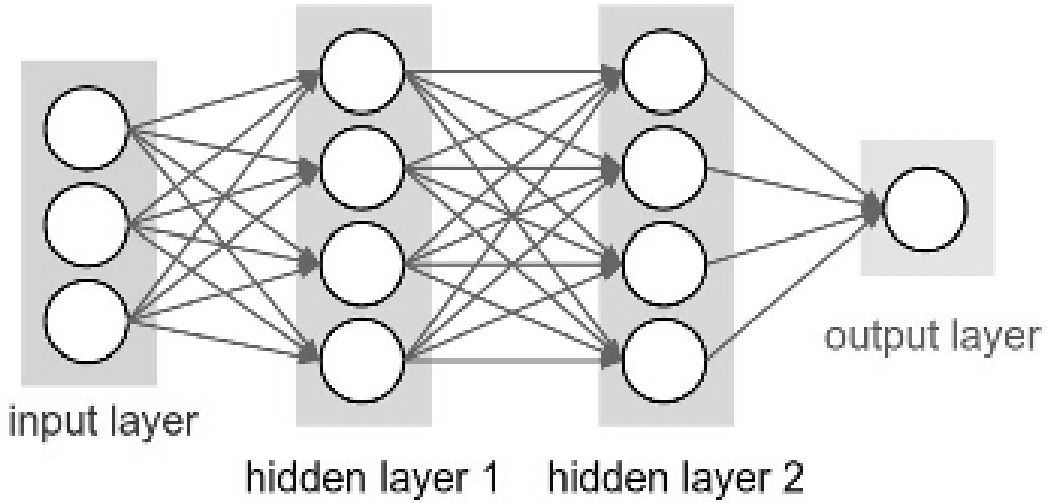
and we can use this to **adapt weights**, such that the **error value** is **minimized**.

This approach holds also for neurons with **non-linear activation functions**, as long as this activation function is **differentiable**.

Supervised neural computation

- From single neurons to neural networks

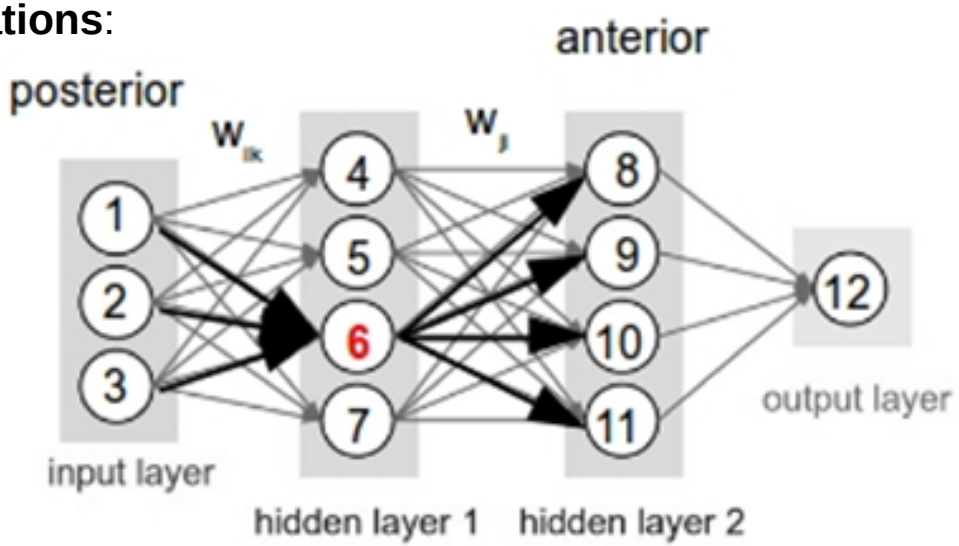
The advanced goal is to **learn "arbitrary" data**, not just such that happens to fit the given neuron transfer function. This we achieve by combining multiple neurons in a **neural network** using feed forward **connectivity** between the neuron layers. In such a structure each **neuron represents some aspect of the data** and the neurons higher up in the **hierarchy** combine these.



Supervised neural computation

- Error Backpropagation in Multi-Layer Neural Networks

The **error backpropagation algorithm** was originally introduced in the 1970s, but its importance wasn't fully appreciated until a famous paper published 1986 by David Rumelhart, Geoffrey Hinton, and Ronald Williams. The backpropagation algorithm searches the **minimum of the error function** in **weight space**, using **gradient descent**. The particular combination of weights which minimizes the error function is considered to be the solution for learning a representation of data. In order to introduce the formalism of backpropagation we introduce the following **notations**:



Notation:
o - output
j - posterior
i - current
k - anterior

$W_{ji} = W_{<TO><FROM>}$

Example:
j - {8, 9, 10, 11}
i - 6
k - {1, 2, 3}

Supervised neural computation

- Error Backpropagation in Multi-Layer Neural Networks

1. Definitions

(1) The error signal for a certain unit i at training time *t* is given by:

$$\delta_i(t) = \frac{-\partial E_i(t)}{\partial net_i(t)}$$

where the net input to neuron *i* is

$$net_i(t) = \sum_{k \in A} w_{ik}(t) \cdot out_k(t)$$

(2) The weight change for weight w_{ik} is given by

$$\Delta w_{ik}(t) = -\frac{\partial E_i(t)}{\partial w_{ik}(t)}$$

2. Understanding the gradient for weight change

Starting from the weight change at the neuron level we can infer the representation of the update in terms of the output of the previous layer and the error at the current neuron:

$$\Delta w_{ik}(t) = -\frac{\partial E_i(t)}{\partial w_{ik}(t)} = \frac{-\partial E_i(t)}{\partial net_i(t)} \cdot \frac{\partial net_i(t)}{\partial w_{ik}(t)} = \delta_i(t) out_k(t)$$

Supervised neural computation

- Error Backpropagation in Multi-Layer Neural Networks

$$\Delta w_{ik}(t) = -\frac{\partial E_i(t)}{\partial w_{ik}(t)} = \frac{-\partial E_i(t)}{\partial ne^t{}_i(t)} \cdot \frac{\partial ne^t{}_i(t)}{\partial w_{ik}(t)} = \delta_i(t) out_k(t)$$

with the error signal for node *i* computed as

$$\frac{-\partial E_i(t)}{\partial net_i(t)} = \delta_i(t) \qquad \text{(by definition (1))}$$

$$\frac{\partial net_i(t)}{\partial w_{ik}(t)} = \frac{\partial \sum_{l \in Ai} w_{il}(t) \cdot out_l(t)}{\partial w_{ik}(t)} = out_k(t) \qquad \text{(non-zero only for l=k)}$$

3. Forward activation of the network

In this phase the "input" is applied to the bottom layer, and we compute all neurons' outputs layer by layer towards the target output:

$$out_i(t) = f_i(\, net_i(t)\,) = f_i(\sum_{k \in A} w_{ik}(t) \cdot out_k(t)\,)$$

Supervised neural computation

- Error Backpropagation in Multi-Layer Neural Networks

4. Calculating the error of the output neuron

For the final output the dataset contains a desired value, $target_o$, hence we can compute the error signal at the network output, out_o, (similar to chapter 3.2):

$$\delta_o(t) = \frac{-\partial E_o(t)}{\partial net_o(t)} = 2 \cdot (target_o - out_o)$$

Note that for simplicity we assume a linear output unit (without loss of generality).

5. Propagating the error back through the network

After computing the error at the network output we propagate the error signal back through the network (hence the name of the learning mechanism).

$$\delta_i(t) = \frac{-\partial E_i(t)}{\partial net_i(t)}$$

Supervised neural computation

- Error Backpropagation in Multi-Layer Neural Networks

Applying the Chain Rule:

$$\delta_i(t) = \frac{\partial E_i(t)}{\partial net_i(t)} = \sum_{j \in P} \frac{-\partial E_i(t)}{\partial net_j(t)} \cdot \frac{\partial net_j(t)}{\partial out_i(t)} \cdot \frac{\partial out_i(t)}{\partial net_i(t)}$$

Where

(1) $\frac{-\partial E_i(t)}{\partial net_j(t)} = \delta_j(t)$ (by definition (1))

(2) $\frac{\partial net_j(t)}{\partial out_i(t)} = \frac{\partial \sum_{m \in P} w_{jm}(t) \cdot out_m(t)}{\partial out_i(t)} = w_{ji}$ (only nonzero for m=i)

(3) $\frac{\partial out_i(t)}{\partial net_i(t)} = \frac{\partial f(net_i(t))}{\partial net_i(t)} = f'(net_i(t))$

Combining those three equations we compute the error signal for a neuron *i* in the network:

$$\delta_i(t) = \sum_{j \in P} \delta_j(t) \cdot w_{ji}(t) \cdot f'(net_i(t))$$

Supervised neural computation

- Error Backpropagation in Multi-Layer Neural Networks

and given the activation function *f* is independent of the j^{th} node, we can rewrite the error:

$$\delta_i(t) = f'(net_i(t)) \cdot \sum_{j \in P} \delta_j(t) \cdot w_{ji}(t)$$

- We observe that the **error for neuron *i*** (δ_i) only depends on the known error of neurons in **"higher" levels** j of the network hierarchy (δ_j).

6. Computing the weight update for weight anterior to posterior (k → i)

Computing the weight update using the weight increment and the error signal

$\Delta w_{ik}(t) = \delta_i(t) \cdot out_k(t)$ $\quad$ $\delta_i(t) = f'(net_i(t)) \cdot \sum_{j \in P} \delta_j(t) \cdot w_{ji}(t)$

The final **weight update** is

$$\Delta w_{ik}(t) = f'(net_i(t)) \cdot \sum_{j \in P} \delta_j(t) \cdot w_{ji}(t) \cdot f(net_k(t))$$

Supervised neural computation

- Supervised learning: tips and tricks

1. Weight initialization

A possible option is to set all the initial weights to zero (or any other constant). This is a fatal mistake, because if every neuron in the network computes the same output, they will also all learn based on identical gradients during error-backpropagation and undergo the exact same parameter updates. In other words, there is no source of asymmetry between neurons if their weights are initialized the same; they all act as they are a single (highly redundant) neuron.

A common method to break symmetry early on is to **initialize all neurons' weights to small random numbers**. Thereby, neurons all behave uniquely, so they will compute distinct updates and develop into diverse contributors of the full network.

Use small weights, as small weights are likely to get the net input into the steep region of the neurons' transfer functions. The gradient will initially be large, so neurons quickly differentiate.

Tip: initialize the weight with small random numbers, e.g. -0.001…0.001

Supervised neural computation

- Supervised learning: tips and tricks

2. Input/Output normalization

Normalization refers to normalizing each of the data dimensions so that they are all on approximately similar scale. There are two common ways of achieving this normalization. One is to divide each dimension by its standard deviation, once it has been zero-centered. Another form of this preprocessing normalizes each dimension so that the min and max along the dimension is -1 and 1 respectively.

Tip: use a linear (final) output neuron transfer, and normalize all input data to [-1 .. +1].

3. Small weight updates

Weight updates are controlled by a parameter, $\eta(t)$, **the learning rate**, which determines how fast the weight change will take place. At every update only a single data point is processed; hence a "full update" will match this point well but neglect others. The learning rate allows small updates towards consecutive examples, which improves overall network behavior. The learning rate should typically be very small.

Tip: use a constant small learning rate, e.g. $\eta(t)=0.001$

Supervised neural computation

- Supervised learning: tips and tricks

4. Avoiding local minima in weight space

In training neural networks (as in any local gradient based method) the training might get stuck in local minima, instead of finding a global minimum. A technique called **Simulated Annealing** might help to overcome local minima but regularly perturbing the current set of weight. This perturbation ("shaking") shall initially be large and decay with training success, so that initially the network likely "jumps" out of local minima; but later likely stays within a found solution.

Tip: use small occasional random perturbation ("shaking") of weights to escape local minima.

5. Network size

Number of Input channels and output channels given by problem data set (this we cannot decide, but it is given by the problem to be solved).

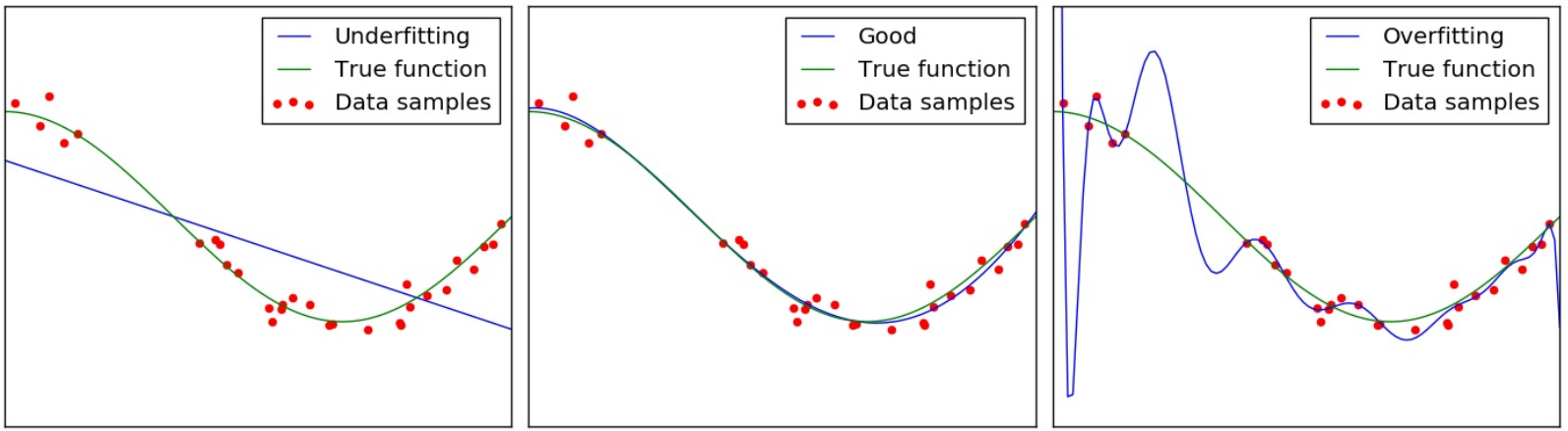
The important **metric** for the design of neural networks are the number of **neurons**, or more precise the number of free parameters. How do we decide on what architecture to use when faced with a practical problem? How many layers? How many neurons per layer? First, note that as we increase the size and number of layers in a neural network, the **capacity of the network** increases.

Supervised neural computation

- Supervised learning: tips and tricks

6. Overtraining/overfitting

The final and often most critical issue in developing a neural network is **generalization**: how well will the network make predictions for cases that are not shown in the training set? Artificial neural networks can suffer from either **underfitting** or **overfitting**.



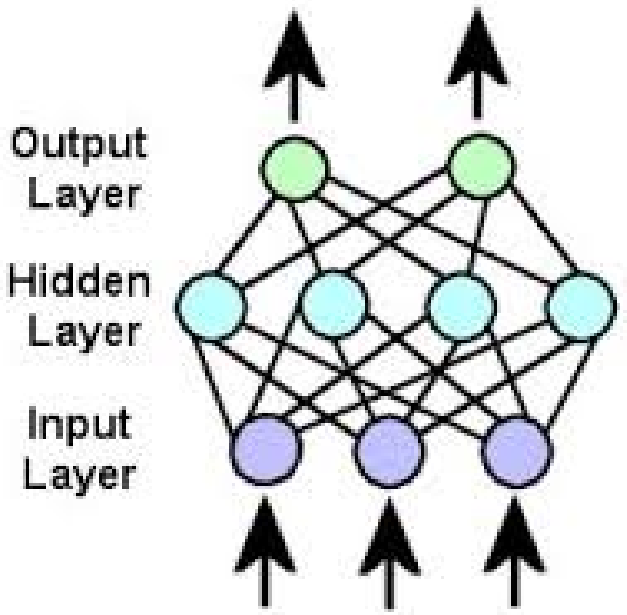
The best way to avoid overfitting is to use **large amounts of training data**. Given a fixed amount of training data, there are several approaches to avoiding underfitting and overfitting, and hence improve generalization: model selection, jittering, weight decay, Bayesian learning, combining networks, and – most commonly used - **Early stopping**.

Supervised neural computation

- From connectionst models to spiking neural networks

Biological neurons use short and sudden increases in voltage to send information. These signals are more commonly known as **action potentials, spikes** or **pulses**.

Connectionism is a theoretical framework for cognition whose principal tenets are (1) that all **cognitive phenomena** arise from the **propagation of activation** among simple neuronlike processing units and (2) that such **propagation** is mediated by **weighted synapselike** connections between units.

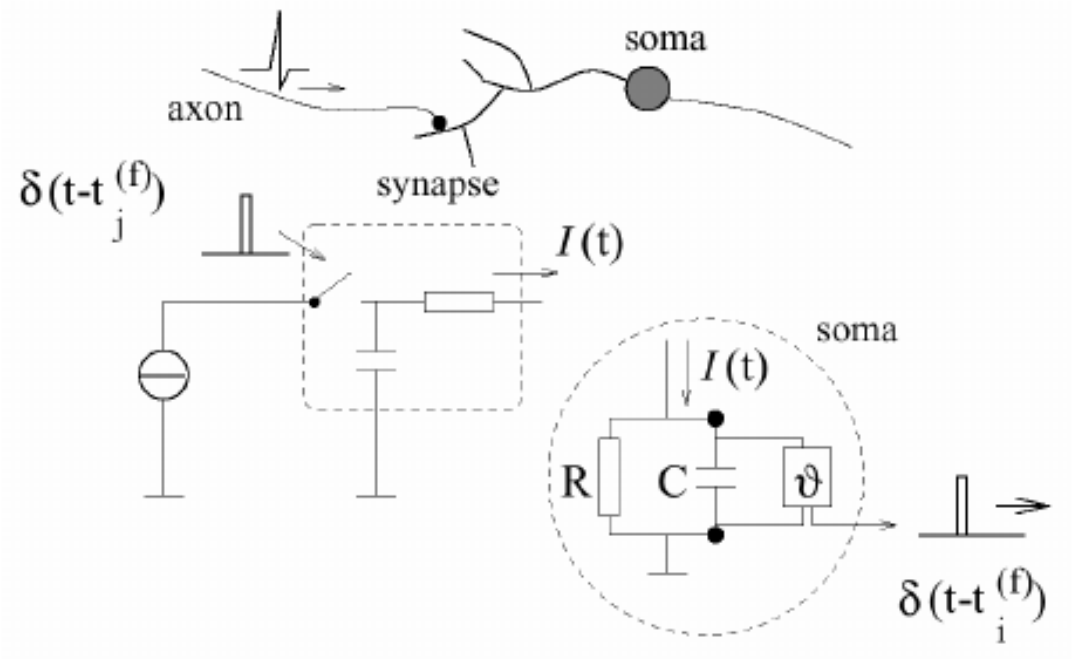


Supervised neural computation

- From connectionst models to spiking neural networks

Neurological research has shown that **neurons encode information in the timing** of single spikes, and not only just in their **average firing frequency**.

Networks of spiking neurons are more powerful than their non-spiking predecessors as they can **encode temporal information** in their signals, but therefore do also need different and biologically more **plausible rules for synaptic plasticity**.



Supervised neural computation

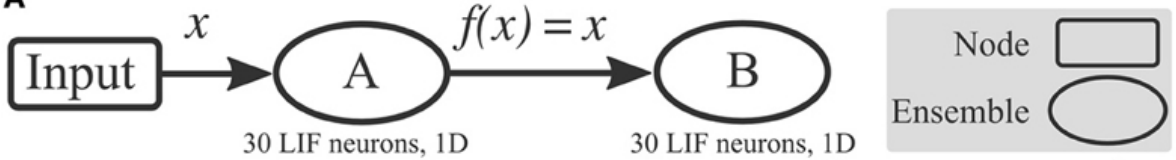
- Spiking neural computation and the Neural Engineering Framework

Spiking Neuron Networks (SNNs) are often referred to as the **3rd generation** of neural networks. They derive their strength and interest from an accurate modeling of **synaptic interactions** between neurons, taking into account the **time of spike firing**.

Nengo is a Python library for building and simulating large-scale brain models using the methods of the **Neural Engineering Framework**. The Neural Engineering Framework (NEF) is the set of theoretical methods that are used in Nengo for constructing spiking neural models.

DEMO

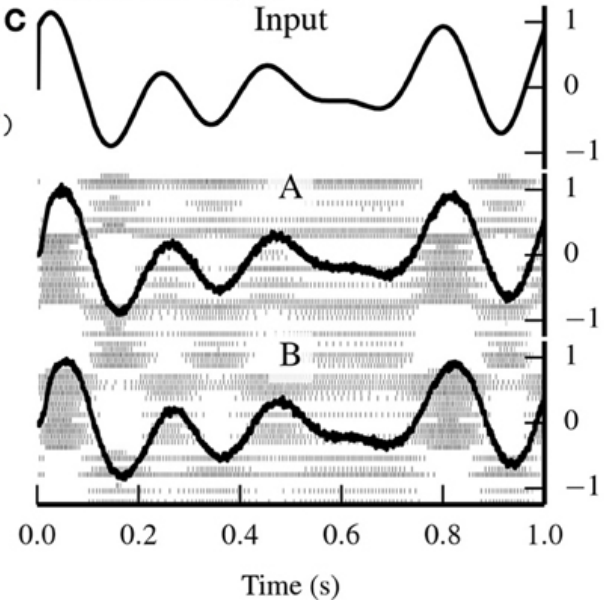
A



B

```
import nengo
from nengo.helpers import white_noise
model = nengo.Model("Communication Channel")
input = nengo.Node(white_noise(1, 5))
a_ens = nengo.Ensemble(nengo.LIF(30), 1)
b_ens = nengo.Ensemble(nengo.LIF(30), 1)
nengo.Connection(input, a_ens)
nengo.Connection(a_ens, b_ens)
a_val = nengo.Probe(
    a_ens, "decoded_output", filter=0.01)
a_spikes = nengo.Probe(a_ens, "spikes")
sim = nengo.Simulator(model)
sim.run(1)
a_data = sim.data(a_val)
a_spikedata = sim.data(a_spikes)
...
```

C



Supervised neural computation

- Spiking neural computation and the Neural Engineering Framework

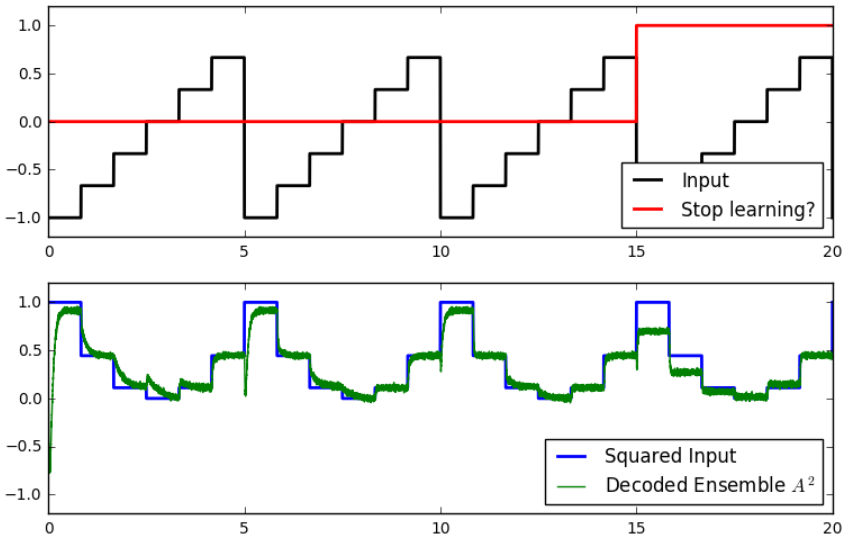
Nengo is a Python library for building and simulating large-scale brain models using the methods of the **Neural Engineering Framework**. The Neural Engineering Framework (NEF) is the set of theoretical methods that are used in Nengo for constructing spiking neural models.

Learning to square the input

Prescribed Error Sensitivity (PES) learning rule.

Modifies a connection's decoders to minimize an error signal provided through a connection to the connection's learning rule. It uses an adaptive learning rate, a scalar indicating the rate at which weights will be adjusted and a fixed timing constant on activities of neurons in pre population.

DEMO



Supervised neural computation

What we learned:

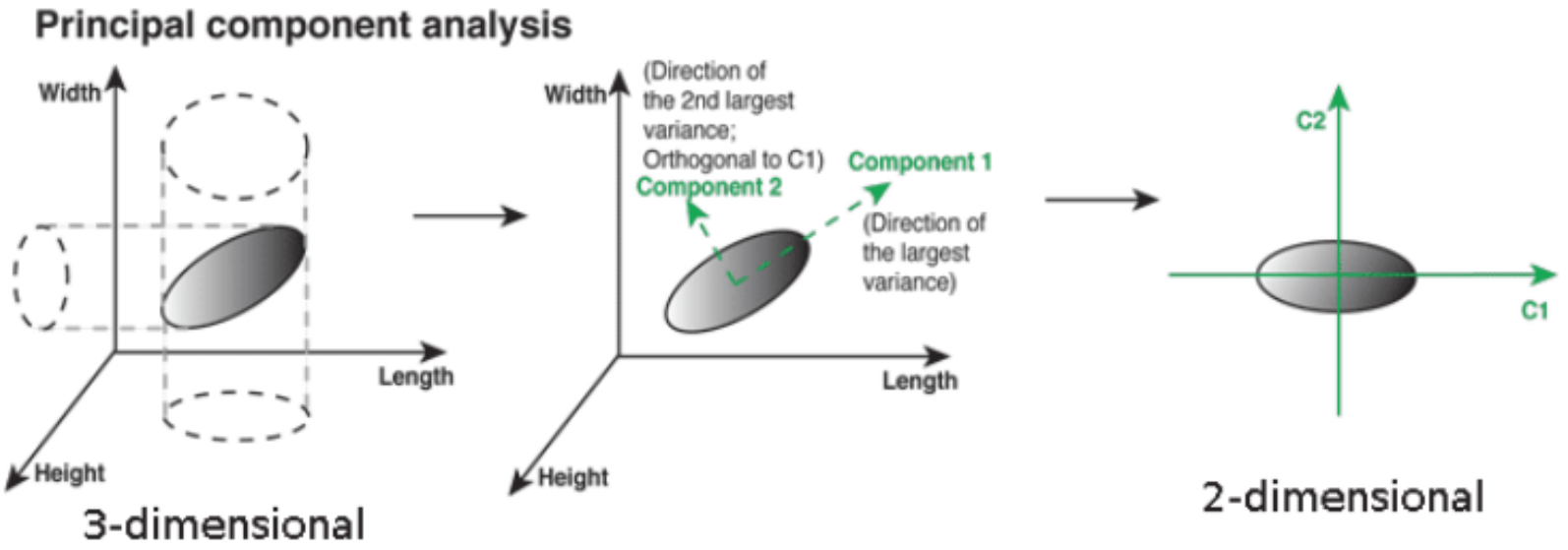
- Biological neurons vs. artificial neurons
- Learning in artificial neurons
- From single neurons to neural networks
- Learning in neural networks: Error Backpropagation in Multi-Layer Neural Networks
- Supervised learning: tips and tricks
- From connectionist models to spiking neural networks
- Supervised spiking neural computation and the Neural Engineering Framework

Unsupervised neural computation

- Introduction to unsupervised learning

Unsupervised learning does not require target vectors for the outputs. Without input-output training pairs as external teachers, unsupervised learning is **self-organized** to produce consistent output vectors by **modifying weights**. That is to say, there are **no labelled examples** of the function to be learned by the network.

For a specific task-independent measure, once the **network** has become tuned to the **statistical regularities** of the input data, the network develops the ability to **discover internal structure** for **encoding features** of the input or **compress the input data,** and thereby to **create new classes** automatically.



Unsupervised neural computation

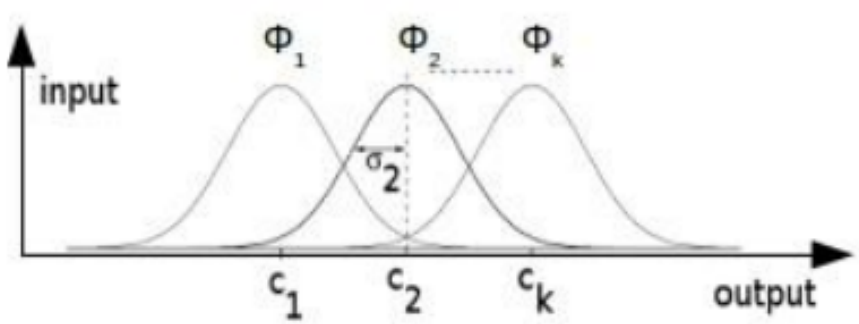
- Radial Basis Functions

The Radial Basis Functions (RBF) technique consists in selecting such a mapping function, F:

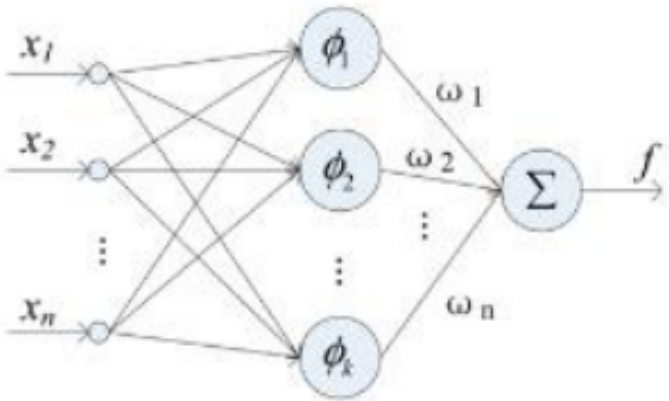
$$f(x) = \sum_{i=1}^{N} w_i \ \phi(\|x - c_i\|),$$

where $\{\phi(\|x - x_i\|) | \ i = 1,2, \dots, N\}$ is the set of *N* arbitrary (generally nonlinear) functions known as radial-basis functions, c_i is the *i-th* center, and $\|\,.\,\|$ denotes a distance metric, usually the Euclidian distance. Typical choices for radial basis functions are Gaussian functions, $\phi(x) = e^{\frac{-x^2}{\beta}}$

Radial-basis functions (RBF)



Radial-Basis Functions Network (RBN)



Unsupervised neural computation

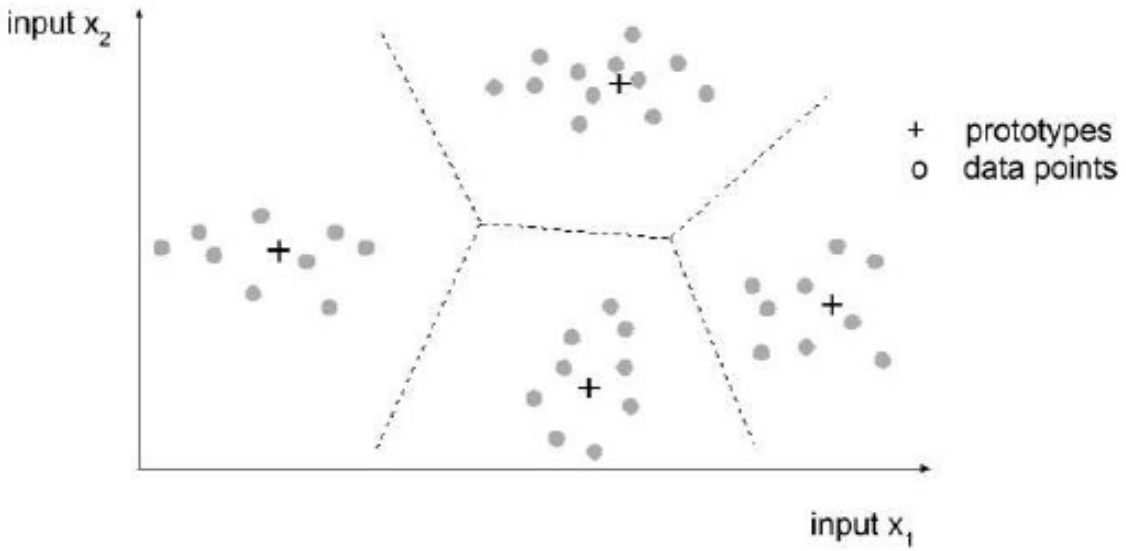
- Radial Basis Functions

	RBFs	MLPs
Hidden units	$f(\|\|x - c_i\|\|)$ Decreasing with increasing distance ("localized)	$f(\sum_{i=1}^{N} w_i \ inp_i)$ Usually nonlinear, monotonically increasing
Output	Only a few active contributors	Many contributors Most "hidden" neurons are active Problems with local minima
Network topology	Simple 3-layer structure	Many structures possible dependent on problem
Training	2-stage process: 1. Finding Gaussian params 2. Training weights	All weights are simultaneously adapted through backpropagation

Unsupervised neural computation

- Vector Quantization

Vector quantization (VQ) is a form of competitive learning. Such an algorithm is able to discover structure in the input data. Generally speaking, vector quantization is a form of **lossy data compression**—lossy in the sense that some information contained in the input data is lost as a result of the compression.



An input data point belongs to a certain class if its position (in the 2D space) is closest to the class prototype, fulfilling the **Voronoi partitioning** (i.e. partitioning of a plane into regions based on distance to points in a specific subset of the plane.

Unsupervised neural computation

- Vector Quantization

Algorithm:

#1 Choose the number of clusters, *M*

#2 Initialize the prototypes $w_1, w_2, \ldots, w_n$ (hint: pick random input samples but distributed evenly in the input space)

#3 Repeat until "good enough"

#4 Randomly pick an input *x*

#5 Determine the winning prototype node *k* such that

$$|w_k - x| \leq |w_i - x| \text{ for all nodes } i$$

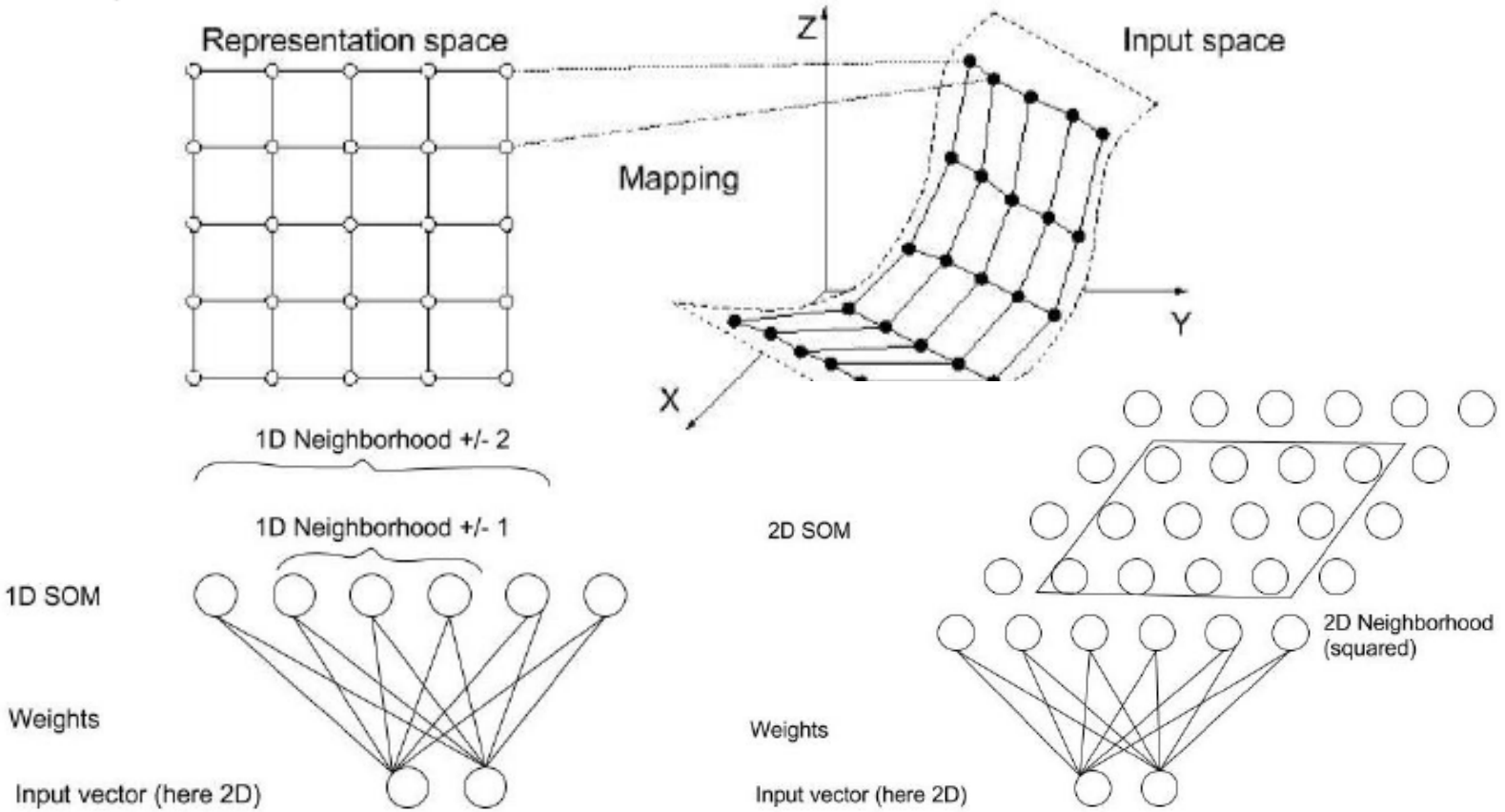
#6 Update the winning prototype weights

$w_k(t+1) = w_k(t) + \eta(x - w_k(t))$, where η is the learning rate.

Unsupervised neural computation

- Kohonen's Self-Organizing-Maps

Kohonen's self-organizing map (**SOM**) is one of the most **popular unsupervised neural network** models. Developed for an associative memory model, it is an unsupervised learning algorithm with a simple structure and computational form, and is motivated by the **retina-cortex mapping**. The SOM can provide **topologically preserved mapping** from input to output spaces, such that "nearby" sensory stimuli are represented in "nearby" regions.



Unsupervised neural computation

- Kohonen's Self-Organizing-Maps

Algorithm:

#1 Initialize all weights w_{ij} and define the neighborhood function $\phi(i,j)$

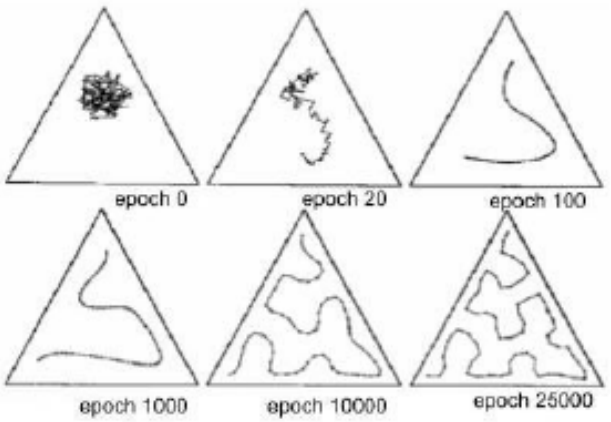
#2 Select input x and determine the winning unit i such that

$$|x - w_k| \leq |x - w_j| \text{ for all nodes } j \neq i$$

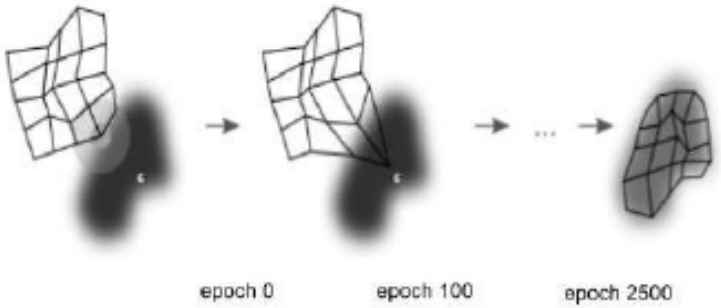
#3 Update weights for all units j given the winner unit i

$$w_j(t+1) = w_j(t) + \eta\phi(j,i)(x - w_j(t)), \text{ where } \eta \text{ is the learning rate}$$

#4 Repeat from step #2 until convergence is reached and update η and ϕ



1D SOM Network training



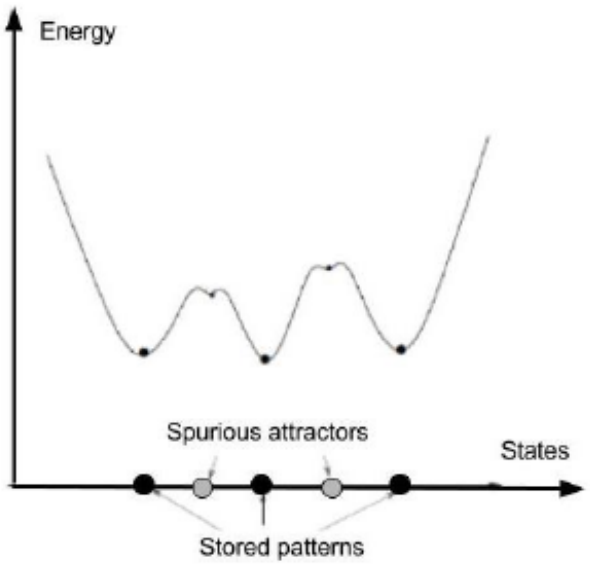
2D SOM Network training

Unsupervised neural computation

- Hopfield Networks

The model is a **recurrent neural network** with fully interconnected neurons. The number of **feedback** loops is equal to the number of neurons. Basically, the output of each neuron is fed back, via a **unit-time delay** element, to each of the other neurons in the network. Such a structure allows the network to **recognise** any of the **learned patterns** by exposure to only partial or even some **corrupted information** about that pattern, i.e., it eventually settles down and returns the closest pattern or the best guess.

In the application of the Hopfield network as **a content-addressable memory**, we know a priori the **fixed points (attractors)** of the network in that they correspond to the patterns to be stored. However, the synaptic weights of the network that produce the desired fixed points are unknown, and the problem is how to determine them.



Unsupervised neural computation

- Hopfield Networks

Update rule:

Assume N neurons $= 1, \cdots, N$ with values $x_i = \pm 1$

The update rule is for the node i is given by:

$$\text{If } h_i \geq 0 \text{ then } 1 \leftarrow x_i \text{ otherwise } -1 \leftarrow x_i$$

where $h_i = \sum_{j=1}^{N} w_{ij} x_j + b_i$ is called the **field** at i, with $b_i \in \mathbb{R}$ a bias.

Thus, $x_i \leftarrow \text{sgn}(h_i)$, where $\text{sgn}(r) = 1$ if $r \geq 0$, and $\text{sgn}(r) = -1$ if $r < 0$.

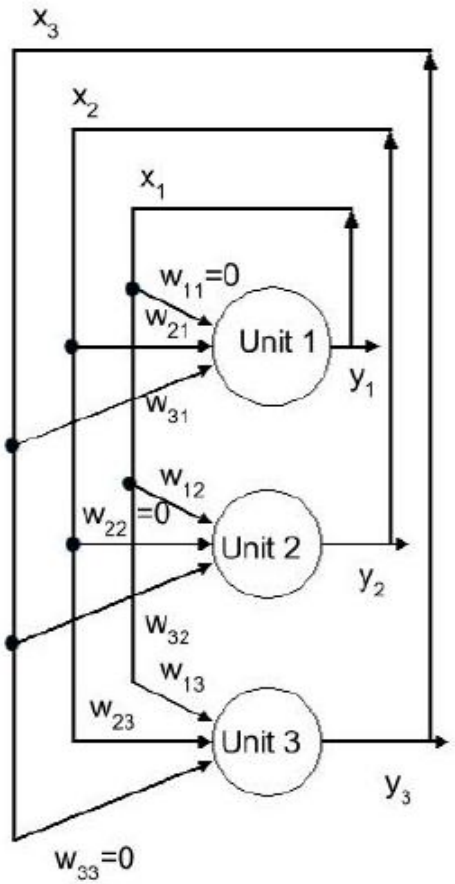
We put $b_i = 0$ for simplicity as it makes no difference to training the network with random patterns.

We therefore assume $h_i = \sum_{j=1}^{N} w_{ij} x_j$.

There are now two ways to update the nodes:

Asynchronously: At each point in time, update one node chosen randomly or according to some rule.

Synchronously: Every time, update all nodes together.



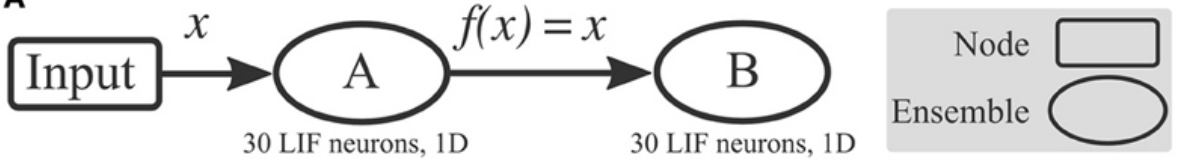
Unsupervised neural computation

- Spiking neural computation and the Neural Engineering Framework

Spiking Neuron Networks (SNNs) are often referred to as the **3rd generation** of neural networks. They derive their strength and interest from an accurate modeling of **synaptic interactions** between neurons, taking into account the **time of spike firing**.

Nengo is a Python library for building and simulating large-scale brain models using the methods of the **Neural Engineering Framework**. The Neural Engineering Framework (NEF) is the set of theoretical methods that are used in Nengo for constructing spiking neural models.

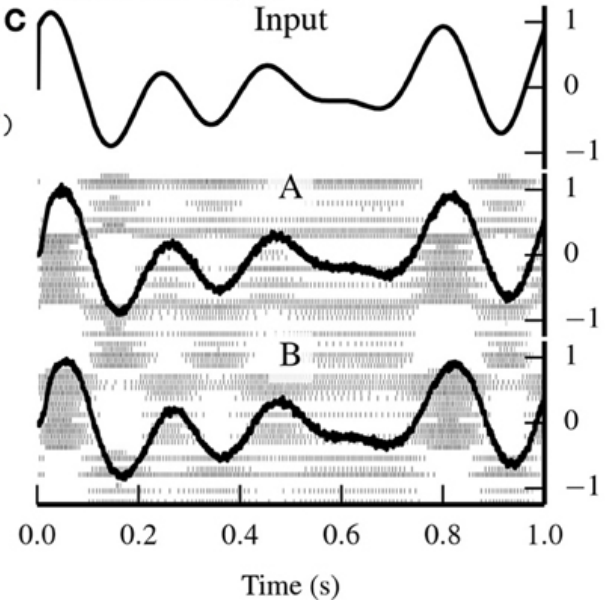
A



B

```
import nengo
from nengo.helpers import white_noise
model = nengo.Model("Communication Channel")
input = nengo.Node(white_noise(1, 5))
a_ens = nengo.Ensemble(nengo.LIF(30), 1)
b_ens = nengo.Ensemble(nengo.LIF(30), 1)
nengo.Connection(input, a_ens)
nengo.Connection(a_ens, b_ens)
a_val = nengo.Probe(
    a_ens, "decoded_output", filter=0.01)
a_spikes = nengo.Probe(a_ens, "spikes")
sim = nengo.Simulator(model)
sim.run(1)
a_data = sim.data(a_val)
a_spikedata = sim.data(a_spikes)
...
```

C



Unsupervised neural computation

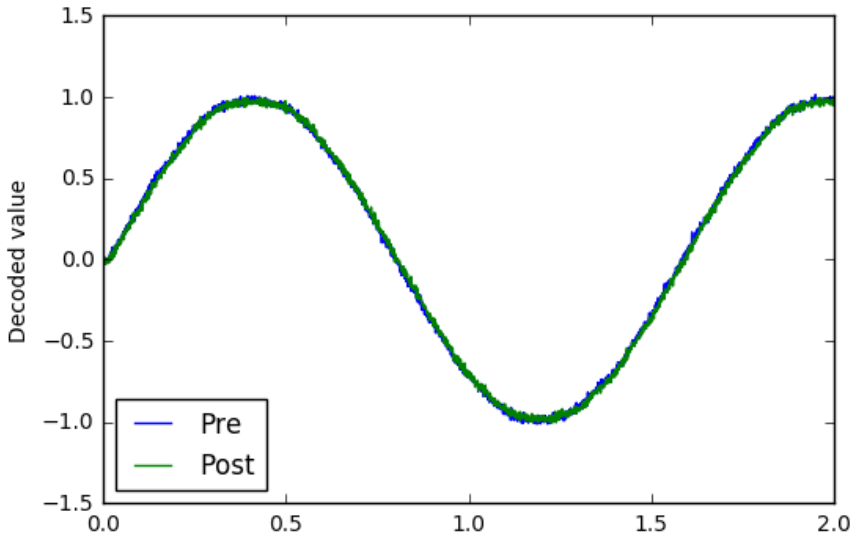
- Spiking neural computation and the Neural Engineering Framework

Nengo is a Python library for building and simulating large-scale brain models using the methods of the **Neural Engineering Framework**. The Neural Engineering Framework (NEF) is the set of theoretical methods that are used in Nengo for constructing spiking neural models.

Bienenstock-Cooper-Munroe (BCM) learning rule.

Modifies connection weights as a function of the presynaptic activity and the difference between the postsynaptic activity and the average postsynaptic activity. As parameters is uses a scalar indicating the time constant for theta integration, a timing constant for filtering activities of neurons in pre and post-synaptic populations and a scalar indicating the rate at which weights will be adjusted.

DEMO



Unsupervised neural computation

What we learned:

- Introduction to unsupervised learning
- Radial Basis Functions
- Vector Quantization
- Kohonen’s Self-Organizing-Maps
- Hopfield Networks
- Unsupervised spiking neural computation and the Neural Engineering Framework

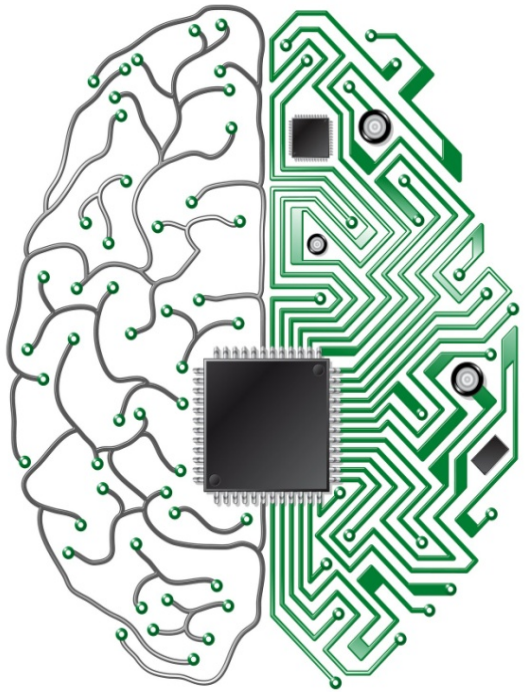
What is neuromorphic engineering ?

Neuro - morphic

"related to neurons"

"structure or form"

- **Discovering principles by which brains compute efficiently**
- **Implementing them in hardware that *intelligently* interacts with the environment**



Examples:

- **Sensors (Silicon Retina/Cochlea)**
- **Massively Parallel Self-Growing Computing Architectures**
- **Distributed Adaptive Control**
- **Neuro-Electronic Implants, Rehabilitation**

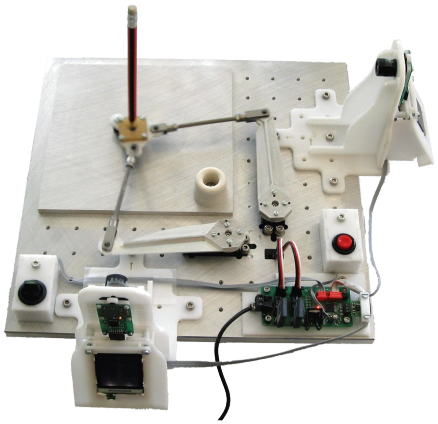
Sample neurobotics applications

Neuronal-style information processing in closed-loop systems

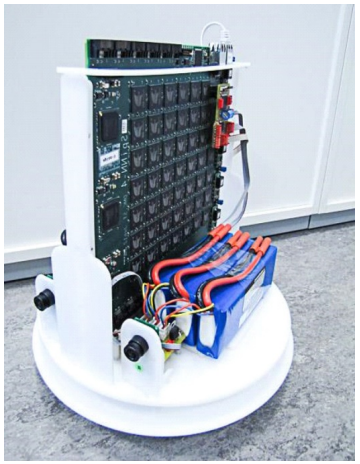
- Distributed Local Information Processing
- Growing and Adaptive Networks of Computational Units
- **Neuromorphic Sensor Fusion** and Actuator Networks
- Event-Based Perception – Action – Loops

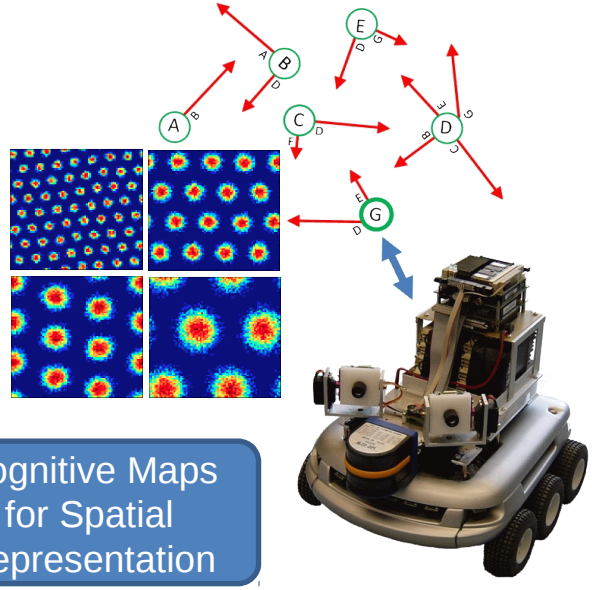
On-Board Vision-Based Control of Miniature UAVs

High-Speed Event-Based Vision



Self-Assembling Distributed Neuronal Computation

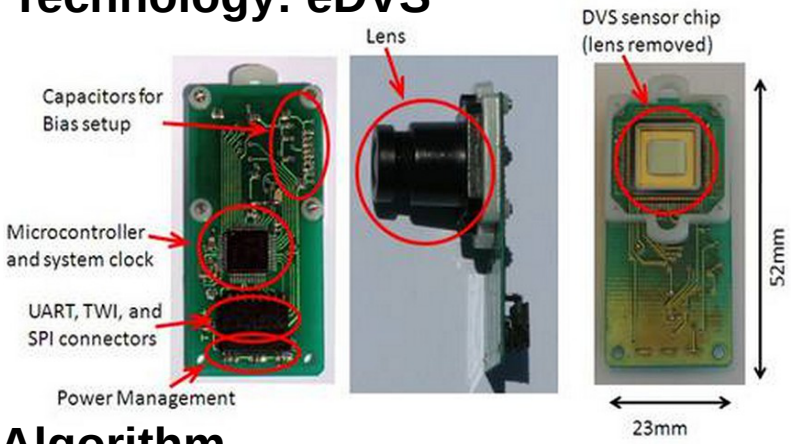




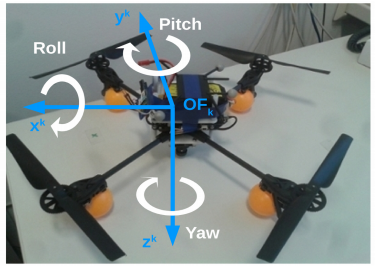
Cognitive Maps for Spatial Representation

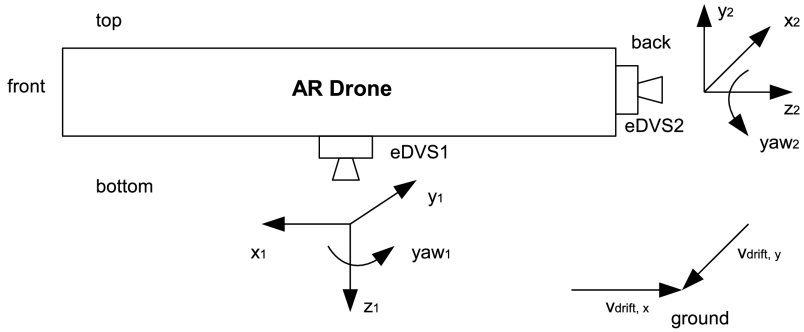
Vision based quadrotor control

Technology: eDVS

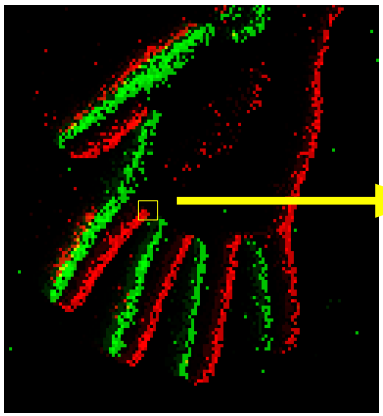


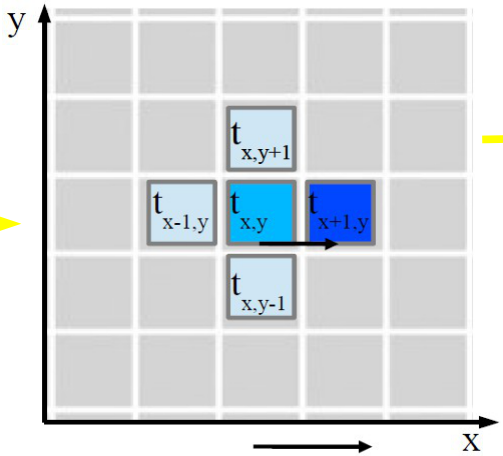
Scenario





Algorithm

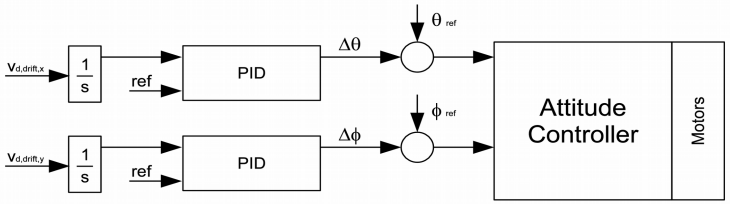




Local optic flow computation

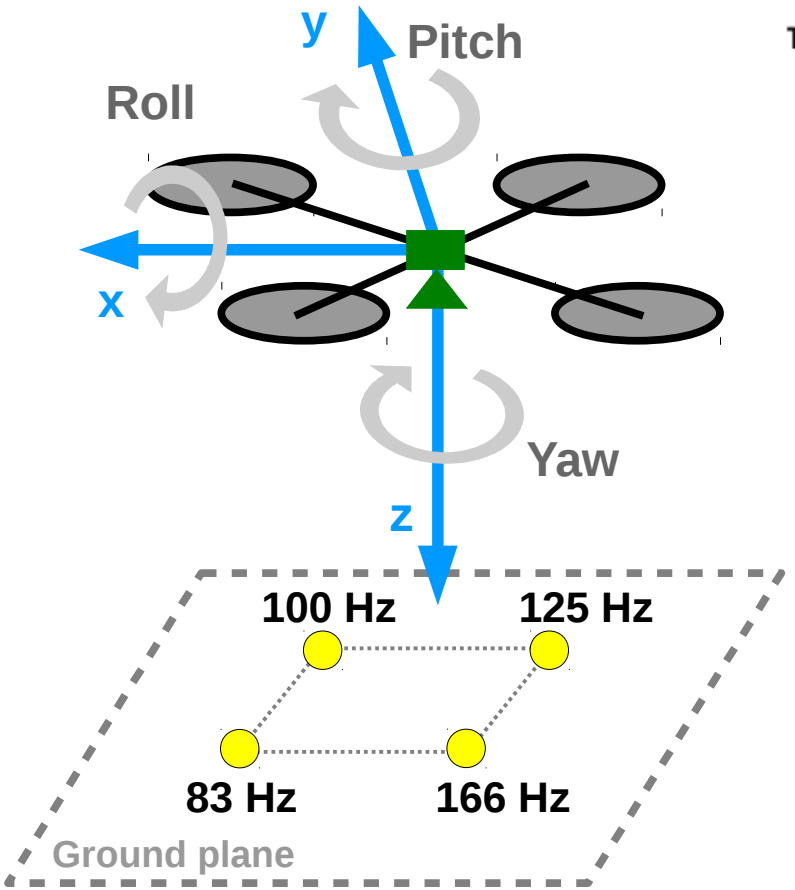
$$\vec{v} = \begin{pmatrix} v_x \\ v_y \end{pmatrix} = \begin{pmatrix} +\frac{a_{pixel}}{(t_{x,y} - t_{x-1,y})} - \frac{a_{pixel}}{(t_{x,y} - t_{x+1,y})} \\ +\frac{a_{pixel}}{(t_{x,y} - t_{x,y-1})} - \frac{a_{pixel}}{(t_{x,y} - t_{x,y+1})} \end{pmatrix}$$

Drift estimates are computed using optic flow

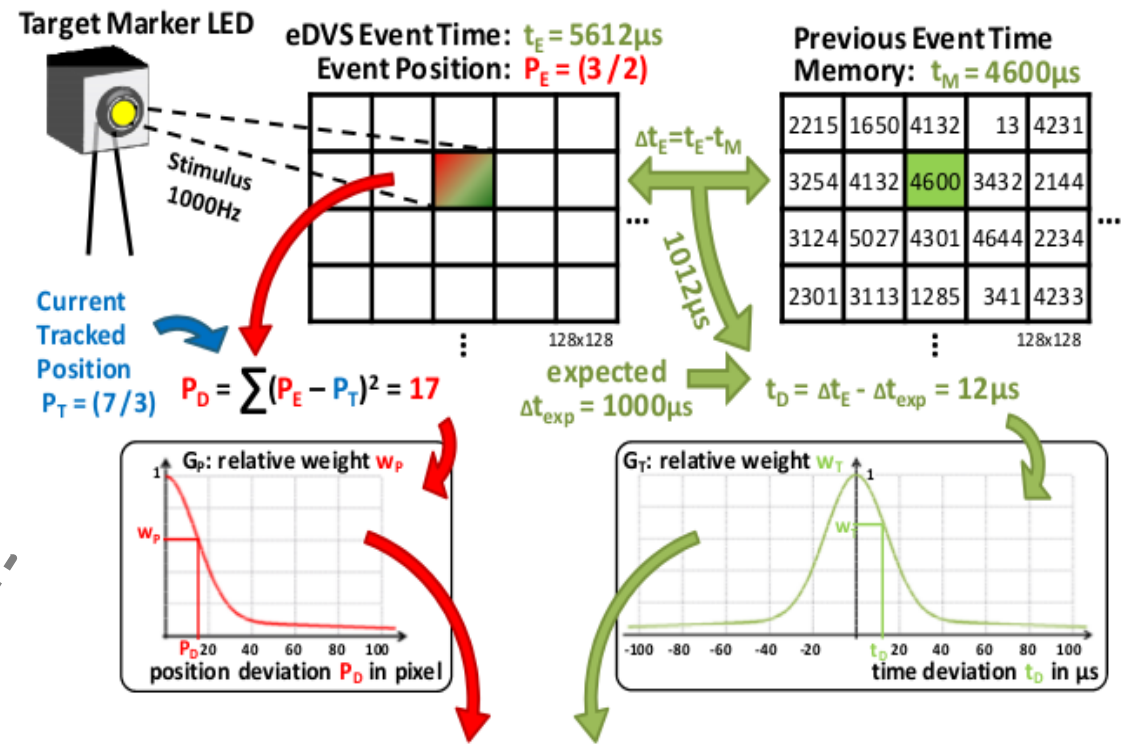


Vision based quadrotor control

Scenario

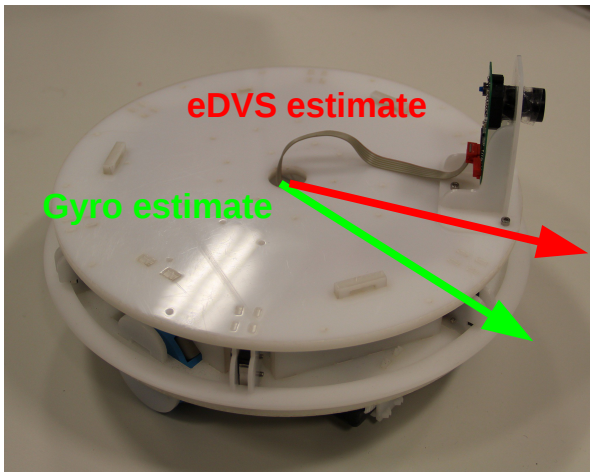


Tracking algorithm

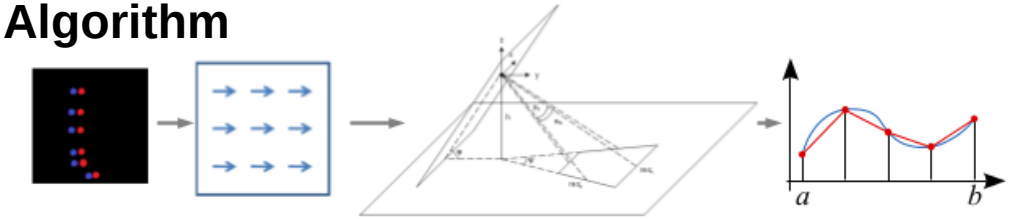


Neuromorphic vision sensor fusion

Scenario

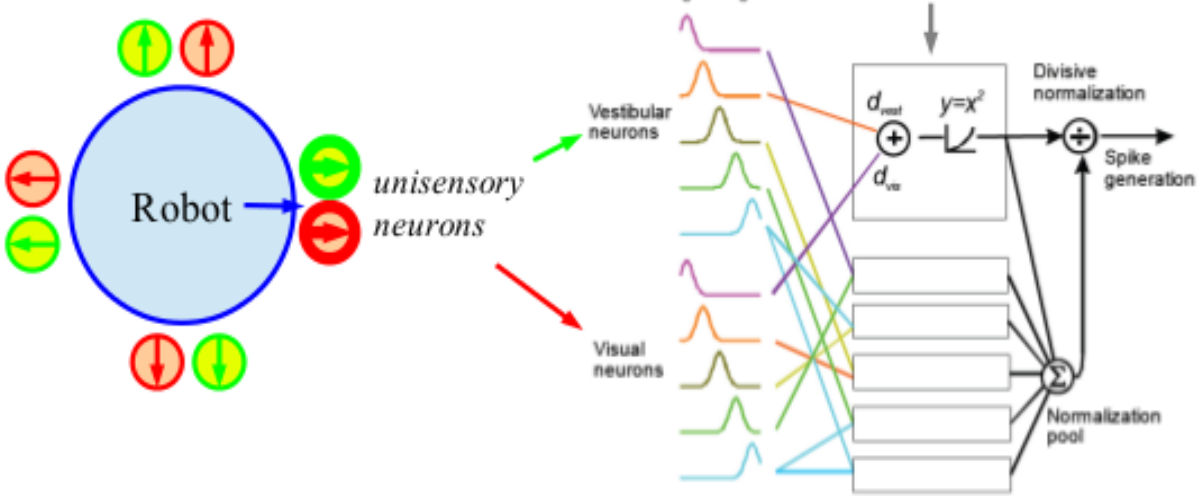


Algorithm



get events from eDVS — *compute optic flow and coherence* — *calculate angular speed considering sensor position* — *integrate angular speed to get angle*

Neural model basics



Neuromorphic engineering and neurobotics

What we learned

- What is neuromorphic engineering?
- Sample neurobotics applications

- Using principles from brain functionality for building artificial retinas, cochleas, neurons
- Intergate them in real-world interactive robotic systems
- Use neural learning and adaptation algorithms to address challenges in traditional robotics

Neurocomputing: From neurons to systems

For more info, visit:

http://neurorobotics.me/

or

http://www.nst.ei.tum.de/