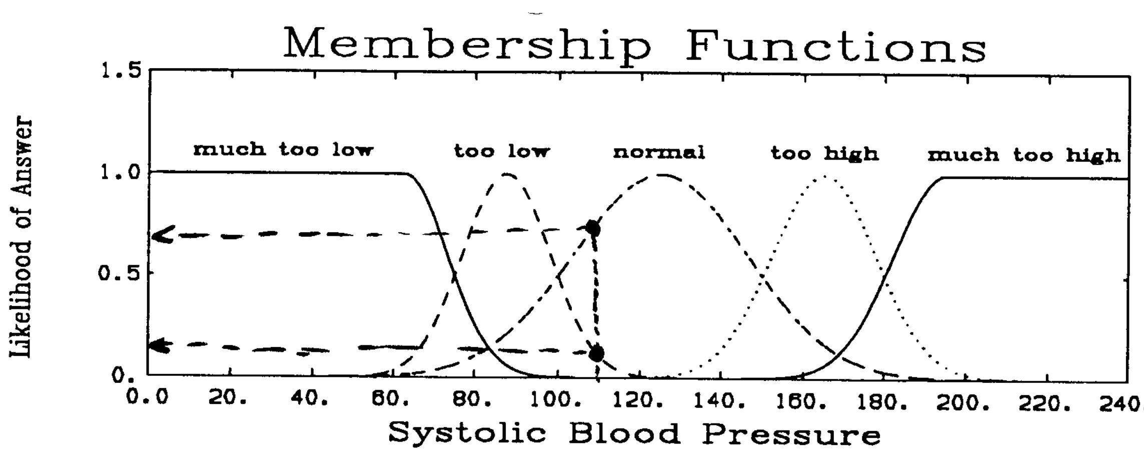
Any discretization scheme suffers from reduced resolution:

fine grid ⟺ good resolution ⟺ many classe[s]
course grid ⟺ poor resolution ⟺ few classe[s]

This means, in order to make the optimization fast, we have to sacrifice resolution.

<u>Remedy:</u>

⟹ <u>Fuzzy Discretization:</u>



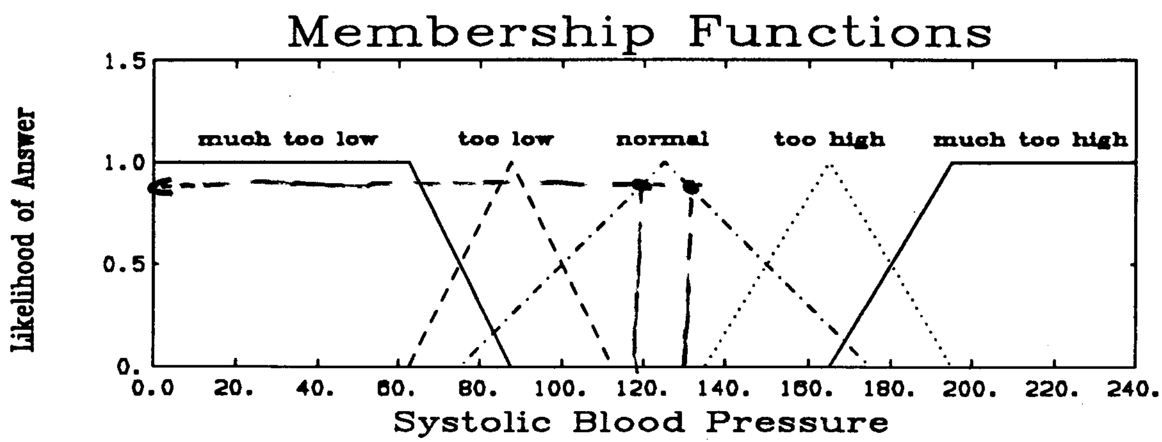
Instead of saying that a blood pressure (a parameter) with a value of 110 is "normal," we can say that it is "normal" with a likelihood of 0.7

and "too low" with a likelihood of 0.15. The likelihood value is called the fuzzy membership function, in the above example, a set of bell-shaped curves associated with each class:

$$M_i(x) = e^{-k_i(x-\mu_i)^2}$$

where μ_i is the center of the "i^{th}" class, and k_i is chosen such that the Membership value, $M_i(x)$, decays to a value of 0.5 at the left and right borders of the "i^{th}" class, i.e., the landmarks separating the i^{th} class from the $(i-1)^{st}$ class and the $(i+1)^{st}$ class.

The shape of the Membership functions is not important. We could just as well use triangular functions:



The problem here is that the <u>fuzzification</u> may no longer be fully reversible.

<u>For example</u>:

P		C	M
118	$\longrightarrow$	normal	0.9
130	$\longrightarrow$	normal	0.9

Since there is not enough overlap, we can no longer conclude unambiguously what the blood pressure was from the knowledge that it is "normal" with a likelihood of 0.9.

<u>Solution</u>:

We throw the tails away, i.e., we <u>only</u> consider the most likely answer ($M \in [0.5, 1.0]$), but add a third piece of information, the <u>side value</u>.

<u>Examples</u>:

(1) <u>Gaussian Membership</u>:

$$p = 110 \longleftrightarrow \begin{cases} C = \text{"normal"} \\ M = 0.7 \\ S = \text{"left"} \end{cases}$$

because we are on the "left" side of the maximum of the Membership function for class "normal."

(2) <u>Triangular Membership</u>:

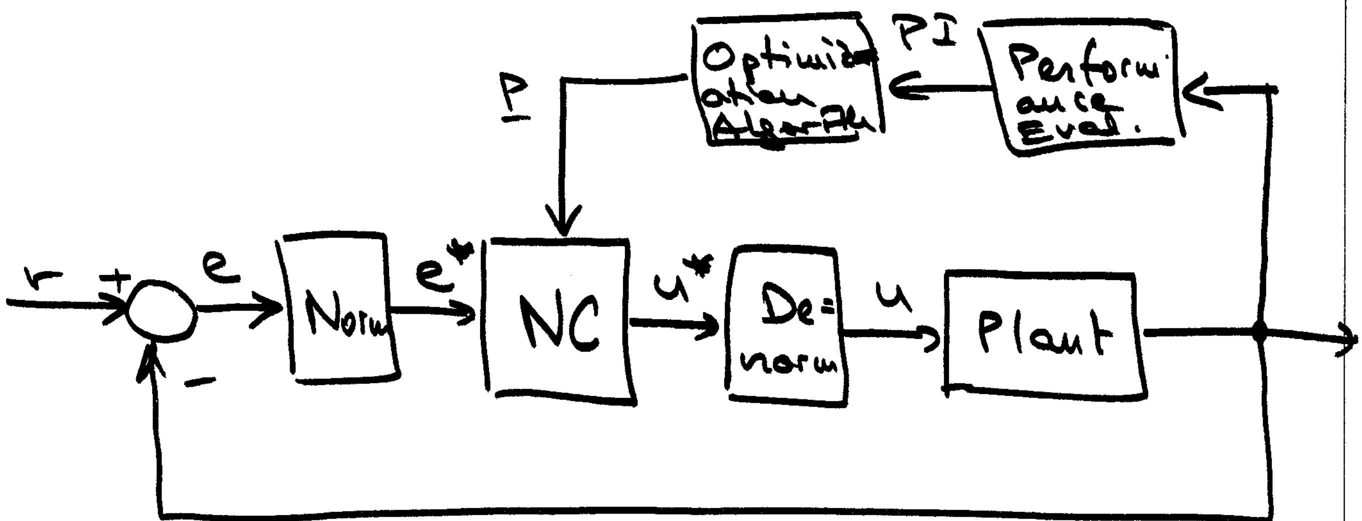
$$p = 130 \longleftrightarrow \begin{cases} C = \text{"normal"} \\ M = 0.9 \\ S = \text{"right"} \end{cases}$$

We can use the class value for <u>optimization</u>, and the membership and side values for <u>interpolation</u>, i.e., after we found the optimal class, we can reproduce an estimate of the continuous parameter by <u>defuzzification</u>, i.e., by converting the estimates of the <u>qualitative triple</u> (consisting of class, memb, and side) back to an estimate for the original parameter.

This concept can be applied to <u>any</u> discretization scheme, including GA.

We haven't discussed yet how we estimate M_i and s_i. We'll do so later.

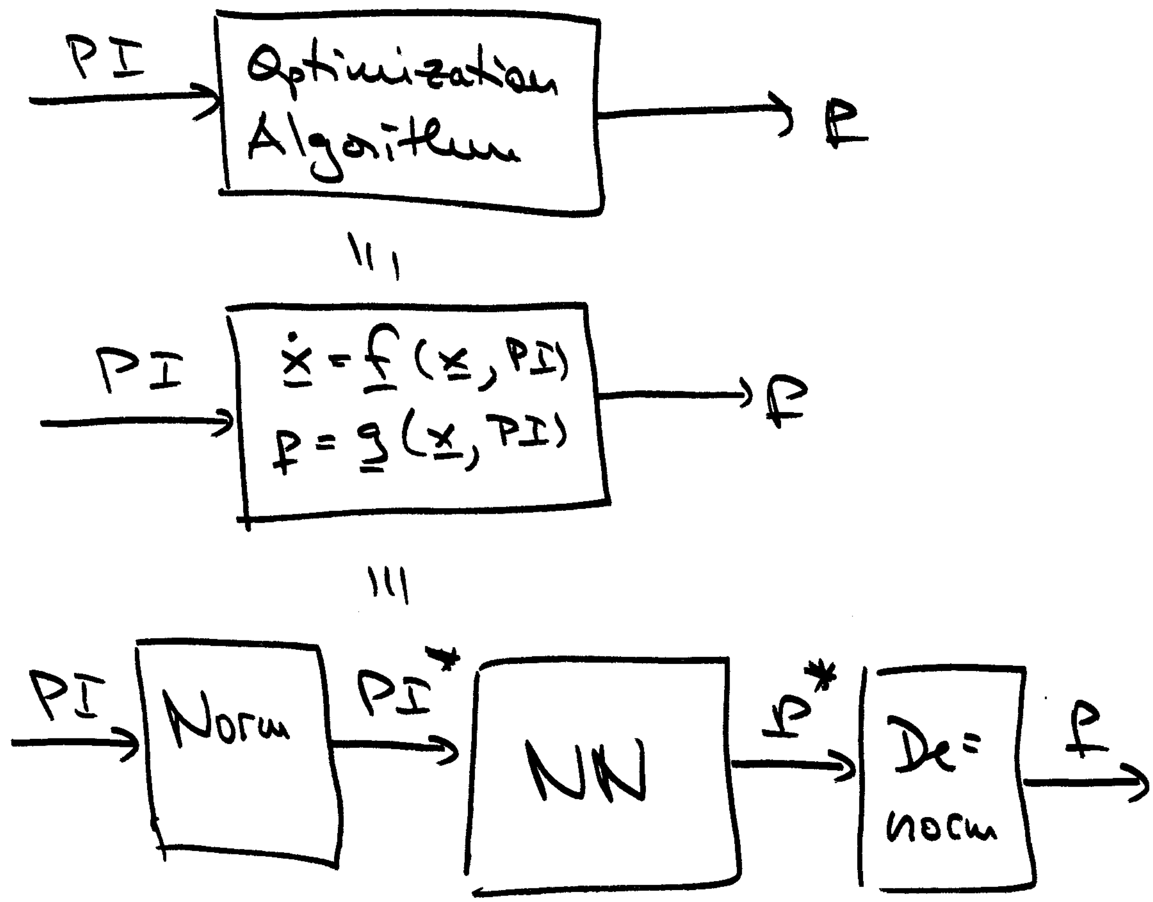
All optimization schemes can be depicted as follows:



The optimization algorithm uses knowledge about old values of $\underline{P}$ and PI and current values of $\underline{P}$ and PI to propose a new value of $\underline{P}$.

We can view this algorithm as a dynamic system generating the outputs $\underline{P}$ from the input PI.

Since the algorithm is application independent, we can implement this algorithm once and for all, e.g. in the form of a NN:



NN is a "teacher's NN" that is application independent, and can be trained once and for all.

NC is a "student NN" that is to be trained by the teacher's NN.

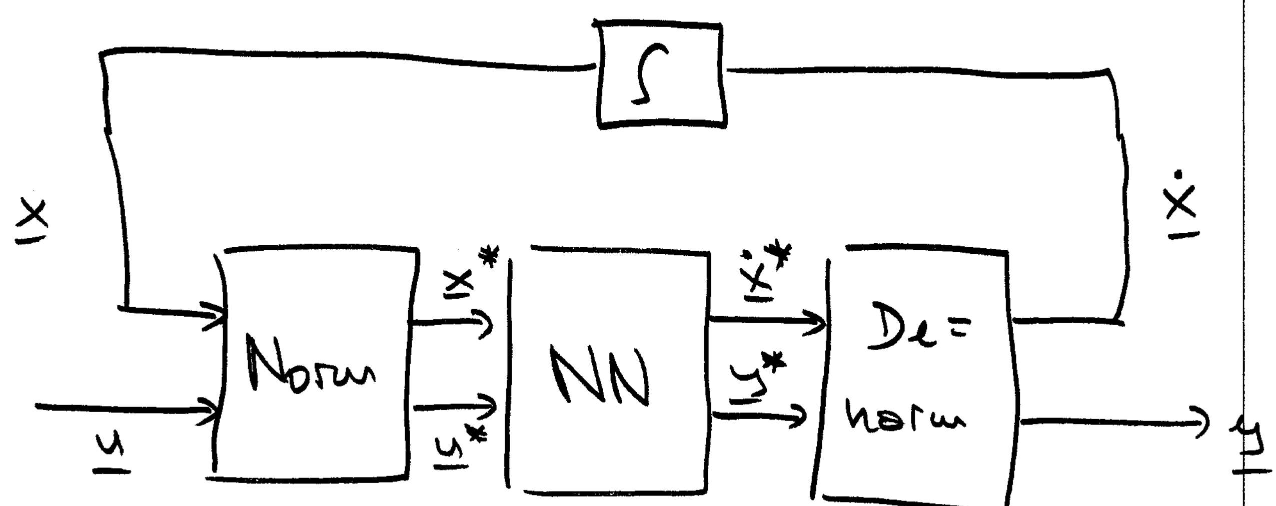
One problem is that we usually need to train systems rather than static functions.

$\Rightarrow$ The NN needs memory, i.e., feedback.

We can do this easily:

$$\left| \begin{aligned} \dot{\underline{x}} &= \underline{f}(\underline{x}, \underline{u}, t) \\ \underline{y} &= \underline{g}(\underline{x}, \underline{u}, t) \end{aligned} \right|$$

are both static functions, thus



However, in my experience, this doesn't work very well. The smallest mismatch between $\dot{\underline{x}}_{NN}$ produced by NN and $\dot{\underline{x}}_{syst}$ produced by $\underline{f}(\underline{x}, \underline{u}, t)$ leads to a bias that gets integrated.

$\Rightarrow$ Stability problems!

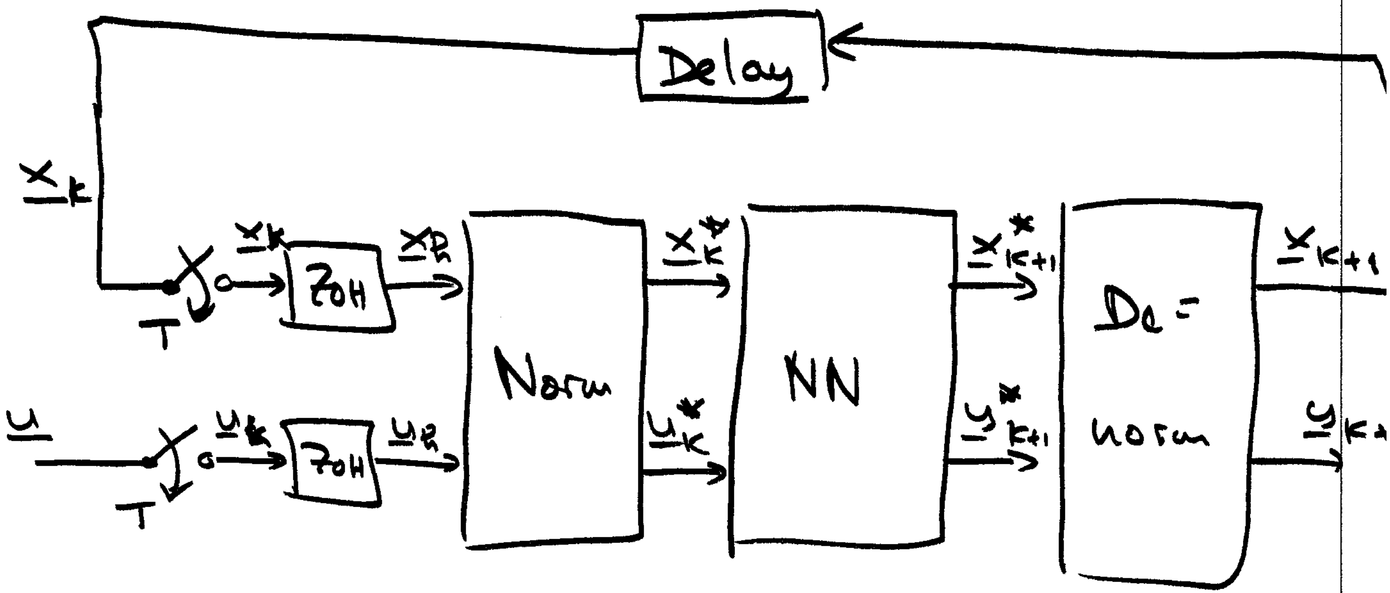
Better solution:

$$\left| \begin{array}{l} \dot{\underline{x}} = \underline{f}(\underline{x}, \underline{u}, t) \\ \underline{y} = \underline{g}(\underline{x}, \underline{u}, t) \end{array} \right|$$

$$\Big\downarrow \text{Approximation}$$

$$\left| \begin{array}{l} \underline{x}_d(k+1) = \underline{f}_d(\underline{x}_k, \underline{u}_k, t_k) \\ y(k+1) = \underline{g}_k(\underline{x}_k, \underline{u}_k, t_k) \end{array} \right|$$

Then:



works <u>much</u> better.

<u>We notice:</u>

- In order to train a NN, we need to solve an optimization problem.

- Optimization problems can be formulated as NNs.

⟹ By specifying the structure of a NN, we haven't really done anything yet. The entire information is in the parameters.

- What do we need a NN for? Why can't we formulate the non-linear controller directly as an optimization problem?

⟹ The <u>only</u> purpose of the NN was the parametrization of the optimization problem.

- The price we paid was high! NNs are extremely gullible. They don't retain any notion of the training data. Thus, if a NN (once trained) is presented with inputs, for which it hasn't been trained, it will happily predict outputs that have no meaning whatsoever.

Remedy:

Find a technique that

→ is <u>non-parametric</u>,

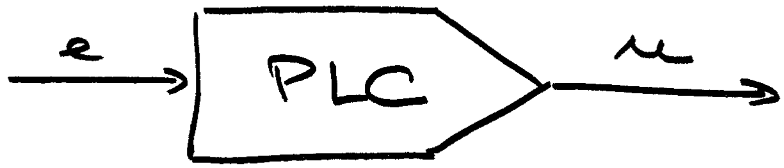
→ keeps the information about the training data,

→ has <u>self</u>-assessment capabilities,

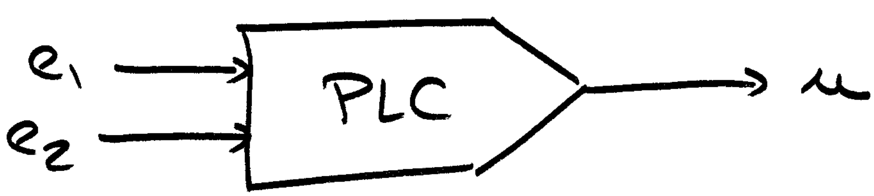
→ predicts not only an output, but also an estimate (in a statistical <u>sense</u>) of the reliability of the prediction made.

⇒ Fuzzy controllers can do all of the above.

Programmable logic controlers (PLC):



e	u
S	S
M	L
L	XL
XL	XL



e_2 \ e_1	S	M	L	XL
XL	XL	L	M	S
L	L	L	S	M
M	M	S	L	L
S	S	M	L	XL

A PLC is a discrete map from a set of inputs to a single output. We don't need to consider MIMO systems, because we can always decompose them into multiple MISO systems (one controller for each output).

A PLC can also be interpreted as a rule-base:

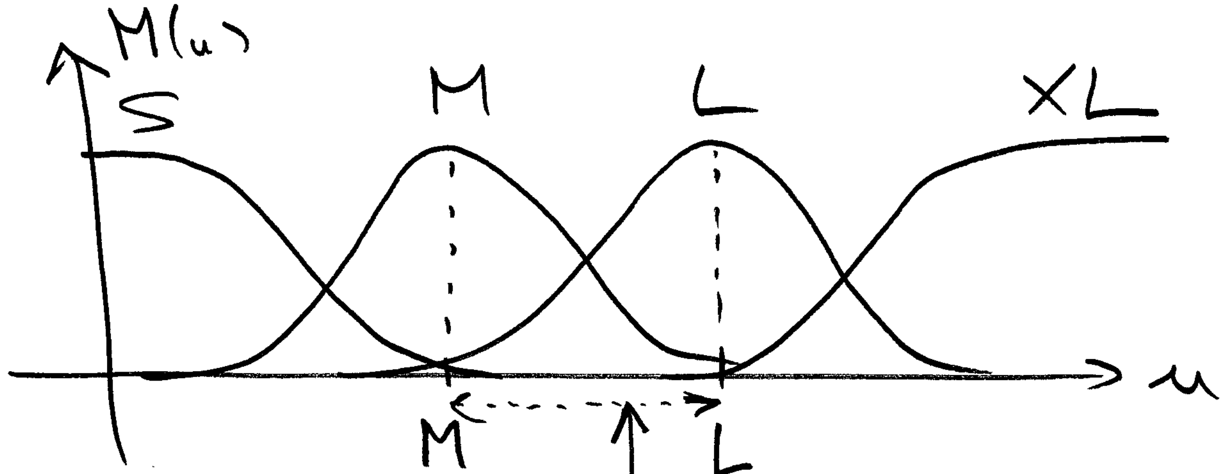
<u>if</u> e_1 is M <u>and</u> e_2 is L
<u>then</u> u is L

<u>etc</u>.

A fuzzy controller is a PLC with fuzzy rules:

<u>if</u> e_1 is M <u>and</u> e_2 is L
<u>then</u> u is L in 70% of the case
<u>and</u> u is M in 30% of the case

In "classical" fuzzy control, <u>de</u><u>fuzzification</u> is done by averaging the outputs using their likelihood:



70% L
∧ 30% M

There are several ways to do this:

MoM := Mean of Maxima
CoG := Center of Gravity

Shall be discussed later.

Dynamic fuzzy controllers use PLCs with memory:

$$\underline{\text{if}} \quad e_1(t) = L \quad \underline{\text{and}}$$
$$e_2(t-\Delta t) = S \quad \underline{\text{and}}$$
$$u(t-\Delta t) = M$$
$$\underline{\text{then}} \quad u(t) = XL \text{ in } 65\% \text{ of the cases}$$

<u>etc</u>.

Training a fuzzy controller is done in two steps:

(I) Find the underlying PLC

(II) Train the shape of the membership functions associated with the output.