One problem with backpropagation is that it assumes we know the output ($\underline{u}$). In a control context, this is not usually the case. All we know is a performance index, something like:

$$PI = \int_0^\infty \underline{e}' Q \underline{e}\, dt \overset{!}{=} \min .$$

or:

$$PI = \int_0^\infty (\underline{e}' Q \underline{e} + \underline{u}' R \underline{u})\, dt \overset{!}{=} \min .$$

$\Rightarrow$ We have a non-linear plant/controller with a bunch of parameters. We can use any dynamic non-linear programming technique.

Assume:

$$PI = \int_0^{t_f} \underline{e}' \cdot \underline{e}\, dt \overset{!}{=} \min .$$

$$\Rightarrow \frac{\partial PI}{\partial \underline{p}} = \frac{\partial}{\partial \underline{p}} \int_0^{t_f} \underline{e}'(\underline{p}) \cdot \underline{e}(\underline{p})\, dt$$

$$= \int_0^{t_f} \frac{\partial}{\partial \underline{p}} \left[\underline{e}'(\underline{p}) \cdot \underline{e}(\underline{p}) \right] dt = \int_0^{t_f} 2 \underline{e}'(\underline{p}) \frac{\partial \underline{e}(\underline{p})}{\partial \underline{p}} dt$$

$$\underline{e} = \underline{r} - \underline{y} \Rightarrow \frac{\partial \underline{e}}{\partial \underline{p}} = - \frac{\partial \underline{y}}{\partial \underline{p}}$$

$$\Rightarrow \frac{\partial PI}{\partial \underline{p}} = -2 \int_0^{t_f} \underline{e}'(\underline{p}) \cdot \frac{\partial \underline{y}}{\partial \underline{p}} dt$$

$\frac{\partial \underline{y}}{\partial \underline{p}}$:= Sensitivity of output $\underline{y}$ with respect to parameter vector $\underline{p}$.

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

$$\frac{\partial \underline{y}}{\partial \underline{p}} = \frac{\partial \underline{g}}{\partial \underline{p}}(\underline{x}, \underline{u}, t)$$

$\hookrightarrow$ use partial differentiation.

Example:

$$\left.\begin{aligned}\dot{y}_1 &= a(1-y_2)y_1\\ \dot{y}_2 &= -c(1-y_1)y_2\\ x_2 &= b\cdot y_2\end{aligned}\right\}$$ Lotka-Volterra Model of a Population Dynam Problem

$$e = \log(x_2) - \log(\hat{x}_2)$$ ← Measurements of Population x_2

$$PI = \int_0^{t_f} e^2 dt$$

$$\Rightarrow \quad \frac{\partial PI}{\partial a} = \int_0^{t_f} PIadot \; dt$$

$$PIadot = 2\cdot e\cdot \frac{\partial e}{\partial a}$$

$$\frac{\partial e}{\partial a} = \frac{1}{x_2}\cdot\frac{\partial x_2}{\partial a}$$

$$\frac{\partial x_2}{\partial a} = b\cdot\frac{\partial y_2}{\partial a}$$

$$\left(\frac{\partial y_1}{\partial a}\right)^{\bullet} = (1-y_2)y_1 + a(1-y_2)\frac{\partial y_1}{\partial a} - a\cdot\frac{\partial y_2}{\partial a}\cdot$$

$$\left(\frac{\partial y_2}{\partial a}\right)^{\bullet} = -c(1-y_1)\frac{\partial y_2}{\partial a} + c\cdot\frac{\partial y_1}{\partial a}\cdot y_2$$

etc.

```
PROGRAM Lotka-Volterra model (larch bud moth)

  INITIAL

    INTEGER  icount
    CONSTANT  tmx = 24.0, icount = 0,  opga = 0.00001
    CONSTANT  opgb = 0.01,  opgc = 0.0001,  opgd = 0.0001
    CONSTANT  opge = 0.0001
    CINTERVAL  cint = 0.1
    XERROR  y1 = 1.0E-6, y2 = 1.0E-6

    TABLE inshi,1,30/ 1949.0000, 1950.0000, 1951.0000, 1952.0000, ...
                      1953.0000, 1954.0000, 1955.0000, 1956.0000, ...
                      1957.0000, 1958.0000, 1959.0000, 1960.0000, ...
                      1961.0000, 1962.0000, 1963.0000, 1964.0000, ...
                      1965.0000, 1966.0000, 1967.0000, 1968.0000, ...
                      1969.0000, 1970.0000, 1971.0000, 1972.0000, ...
                      1973.0000, 1974.0000, 1975.0000, 1976.0000, ...
                      1977.0000, 1978.0000, ...
                         0.0630,    0.2490,    1.2530,    6.4210, ...
                       121.6230,  767.7910,  252.1130,   42.5580, ...
                         4.4090,    0.2580,    0.2000,    0.8050, ...
                         3.2980,   45.5560,  413.5200,  362.4400, ...
                         5.1400,    0.0970,    0.0420,    0.2150, ...
                         0.5670,    1.8720,   19.3070,  368.5900, ...
                       401.8720,  333.6690,   18.5850,    0.0550, ...
                         0.0180,    0.1420/

    TABLE insme,1,30/ 1949.0000, 1950.0000, 1951.0000, 1952.0000, ...
                      1953.0000, 1954.0000, 1955.0000, 1956.0000, ...
                      1957.0000, 1958.0000, 1959.0000, 1960.0000, ...
                      1961.0000, 1962.0000, 1963.0000, 1964.0000, ...
                      1965.0000, 1966.0000, 1967.0000, 1968.0000, ...
                      1969.0000, 1970.0000, 1971.0000, 1972.0000, ...
                      1973.0000, 1974.0000, 1975.0000, 1976.0000, ...
                      1977.0000, 1978.0000, ...
                         0.0180,    0.0820,    0.4440,    4.1740, ...
                        68.7970,  331.7600,  126.5410,   21.2800, ...
                         2.2460,    0.0850,    0.0800,    0.3710, ...
                         1.6380,   22.8780,  248.8170,  184.2720, ...
                         3.1160,    0.0190,    0.0020,    0.0590, ...
                         0.1970,    1.0680,   10.5690,  173.9320, ...
                       249.6120,  176.0230,    4.7490,    0.0140, ...
                         0.0080,    0.0560/

    TABLE inslo,1,30/ 1949.0000, 1950.0000, 1951.0000, 1952.0000, ...
                      1953.0000, 1954.0000, 1955.0000, 1956.0000, ...
                      1957.0000, 1958.0000, 1959.0000, 1960.0000, ...
                      1961.0000, 1962.0000, 1963.0000, 1964.0000, ...
                      1965.0000, 1966.0000, 1967.0000, 1968.0000, ...
                      1969.0000, 1970.0000, 1971.0000, 1972.0000, ...
                      1973.0000, 1974.0000, 1975.0000, 1976.0000, ...
                      1977.0000, 1978.0000, ...
                         0.0070,    0.0090,    0.0020,    0.3040, ...
                        23.8660,  208.0050,   56.3300,   11.7120, ...
                         1.3550,    0.0001,    0.0120,    0.0850, ...
                         0.5710,    9.5570,  103.9700,   68.6700, ...
                         1.0000,    0.0001,    0.0001,    0.0001, ...
                         0.0210,    0.3330,    2.5660,   50.0000, ...
                        82.5000,   62.5630,    0.9170,    0.0001, ...
```

```
                    0.0001,     0.0170/

    a = 0.2971
    b = 36.0
    c = 2.948
    d = 0.8359
    e = 6.589

    ast = a
    bst = b
    cst = c
    dst = d
    est = e

    abold = 0.0
    acold = 0.0
    adold = 0.0
    aeold = 0.0

    gold = 1.0E20

L1..icount = icount + 1

    PRINT L7, icount, a, b, c, d, e
L7..FORMAT(1X,I3,5E12.4)

  END $ "of INITIAL"

  DYNAMIC

    DERIVATIVE

      time = t + 1954.0
      insmax = inshi(time)
      insmed = insme(time)
      insmin = inslo(time)

      y1dot   =  a*(1.0 - y2)*y1
      y2dot   = -c*(1.0 - y1)*y2
      x2      =  b*y2
      er      =  alog(x2) - alog(insmed)
      PIdot   =  er*er
      y1      = INTEG(y1dot,d)
      y2      = INTEG(y2dot,e)
      PI      = INTEG(PIdot,0.0)

      y1adot =  (1.0 - y2)*y1 + a*(1.0 - y2)*y1a - a*y2a*y1
      y2adot = -c*(1.0 - y1)*y2a + c*y1a*y2
      x2a    =  b*y2a
      era    =  x2a/x2
      PIadot =  2.0*er*era
      y1a    = INTEG(y1adot,0.0)
      y2a    = INTEG(y2adot,0.0)
      PIa    = INTEG(PIadot,0.0)

      y1bdot =  a*(1.0 - y2)*y1b - a*y2b*y1
      y2bdot = -c*(1.0 - y1)*y2b + c*y1b*y2
      x2b    =  y2 + b*y2b
      erb    =  x2b/x2
      PIbdot =  2.0*er*erb
```

```
        y1b     = INTEG(y1bdot,0.0)
        y2b     = INTEG(y2bdot,0.0)
        PIb     = INTEG(PIbdot,0.0)

        y1cdot =  a*(1.0 - y2)*y1c - a*y2c*y1
        y2cdot =  (y1 - 1.0)*y2 - c*(1.0 - y1)*y2c + c*y1c*y2
        x2c    =  b*y2c
        erc    =  x2c/x2
        PIcdot =  2.0*er*erc
        y1c    = INTEG(y1cdot,0.0)
        y2c    = INTEG(y2cdot,0.0)
        PIc    = INTEG(PIcdot,0.0)

        y1ddot =  a*(1.0 - y2)*y1d - a*y2d*y1
        y2ddot = -c*(1.0 - y1)*y2d + c*y1d*y2
        x2d    =  b*y2d
        erd    =  x2d/x2
        PIddot =  2.0*er*erd
        y1d    = INTEG(y1ddot,1.0)
        y2d    = INTEG(y2ddot,0.0)
        PId    = INTEG(PIddot,0.0)

        y1edot =  a*(1.0 - y2)*y1e - a*y2e*y1
        y2edot = -c*(1.0 - y1)*y2e + c*y1e*y2
        x2e    =  b*y2e
        ere    =  x2e/x2
        PIedot =  2.0*er*ere
        y1e    = INTEG(y1edot,0.0)
        y2e    = INTEG(y2edot,1.0)
        PIe    = INTEG(PIedot,0.0)

      END $ "of DERIVATIVE"

      TERMT(t.ge.tmx)

    END $ "of DYNAMIC"

    TERMINAL

      PRINT L9, PIa, PIb, PIc, PId, PIe
  L9..FORMAT(4X,5E12.4)

      "Calculate length of gradient"
      g  = PIa*PIa + PIb*PIb + PIc*PIc + PId*PId + PIe*PIe

      "If length of gradient is sufficiently small, stop"
      IF( g .LE. 1.0E-5 ) GOTO L5

      "If iteration does not converge, stop also"
      IF( icount .GT. 100 ) GOTO L4

      "If length of gradient has grown, decrease opg"
      IF( g .LE. gold ) GOTO L2
      opga = opga/2.0
      opgb = opgb/2.0
      opgc = opgc/2.0
      opgd = opgd/2.0
      opge = opge/2.0

      "If norm of differences between angles has not changed much,"
```

```
     "increase opg"
L2..ab = PIb/PIa
    dab = ab - abold
    IF( dab*dab .LT. 0.03 ) opgb = 1.5*opgb
    ac = PIc/PIa
    dac = ac - acold
    IF( dac*dac .LT. 0.03 ) opgc = 1.5*opgc
    ad = PId/PIa
    dad = ad - adold
    IF( dad*dad .LT. 0.03 ) opgd = 1.5*opgd
    ae = PIe/PIa
    dae = ae - aeold
    IF( dae*dae .LT. 0.03 ) opge = 1.5*opge

    "If new parameter values become negative, system becomes instable"
    "Reduce opg also, and repeat"
La..ast = a - opga*PIa
    IF( ast.GT.0.0 ) GOTO Lb
    opga = opga/2.0
    GOTO La
Lb..bst = b - opgb*PIb
    IF( bst.GT.0.0 ) GOTO Lc
    opgb = opgb/2.0
    GOTO Lb
Lc..cst = c - opgc*PIc
    IF( cst.GT.0.0 ) GOTO Ld
    opgc = opgc/2.0
    GOTO Lc
Ld..dst = d - opgd*PId
    IF( dst.GT.0.0 ) GOTO Le
    opgd = opgd/2.0
    GOTO Ld
Le..est = e - opge*PIe
    IF( est.GT.0.0 ) GOTO L3
    opge = opge/2.0
    GOTO Le

    "Okay! A new set of parameter values has been found.  Iterate"
L3..a = ast
    b = bst
    c = cst
    d = dst
    e = est

    abold = ab
    acold = ac
    adold = ad
    aeold = ae

    gold = g

    PRINT L8, PI, opga, opgb, opgc, opgd, opge
L8..FORMAT(4X,6E12.4)

    GOTO L1

    "The iteration did not converge.  Print out message"
L4..CONTINUE

    "That's it"
```

```
L5..CONTINUE
  END $ "of TERMINAL"
END $ "of PROGRAM"
```

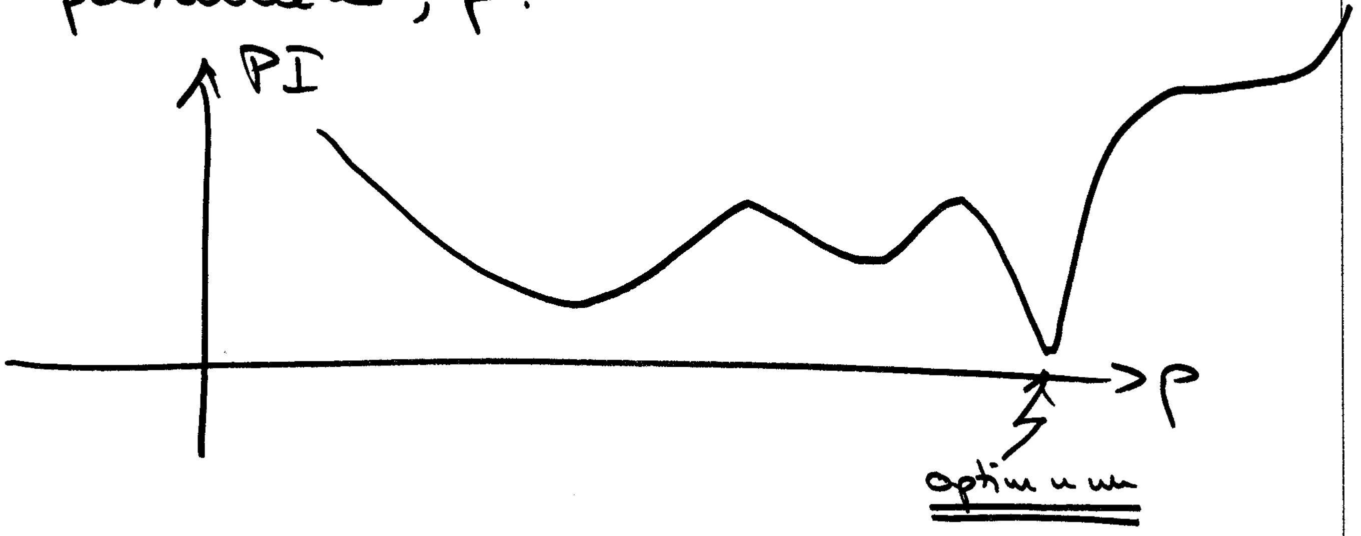
Alternatively, we need to come up with a technique to estimate $\underline{y}_{opt}$ given $\underline{r}$ and $\underline{y}_{desired}$ (cf. later). Then we can use backpropagation again.

Of course, we can also program the non-linear dynamical system in Simulink (or Dymola, generating Simulink code), leave the parameters in the model, and use the optimization toolbox of Matlab to find the best parameter values.

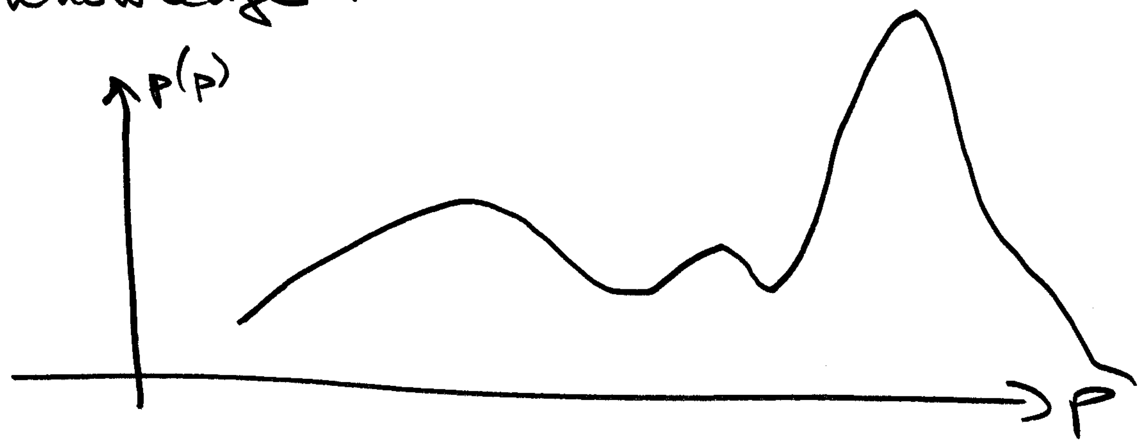
→ Monte Carlo Methods:

The previously discussed methods all require knowledge or at least an estimate of the gradient of the performance index. Sometimes, this may be too costly.

We may want to optimize the performance index, PI, w.r.t. one parameter, p:



We can pick arbitrary values of p, evaluate PI, and then shape the probability density function as 1/PI, as we gain more knowledge :



Thereby, future draws will concentrate more and more on the valleys, and we need less experiments.

This idea can be easily generalized to multiple parameters.

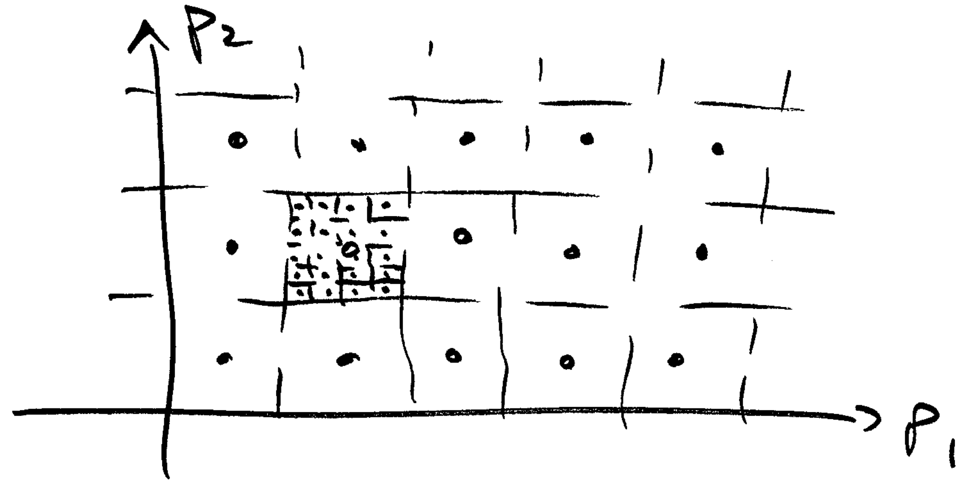
⇒ These methods are <u>slower</u>, but <u>more robust</u>.

⇒ In general, the convergence speed and the convergence range of an optimization method are in competition with each other.

→ Discretization Methods:

One way to speed up the optimization is to make the problem less hard by reducing the search space.

Example:



Evaluate PI only for centers of cubes. Take the smallest value, consider only that cube further, and discretize again, etc.

→ Genetic Algorithms (GA):

Are a class of discrete Monte Carlo methods.

Idea: Combine the discretization idea with the Monte Carlo idea.

⟹ Even after discretization, there may still be to many points to visit them all. We may visit them randomly. GA is nothing but a semi-intelligent scheme to guide the random visit, i.e., an indirect way of shaping the distribution density function to favor "good" solutions.

• As in the case of NNs, the fact that GA borrows from biology (Darwinism) is totally irrelevant.

14.11 Genetic Learning

Let us now return to the mechanisms of learning. I had mentioned earlier that gradient techniques are dangerous because of potential stability problems, besides the fact that they are not biologically plausible.

In this section, I shall introduce another optimization technique that does not exhibit the stability problems characteristic of gradient techniques, and while this approach is not biologically plausible in the context of neural learning, it has at least been inspired by biology. Genetic algorithms were first developed by John Holland in the late 1960s [14.14]. As with the neural networks, the basic idea behind genetic algorithms encompasses an entire methodology. Thus, many different algorithms can be devised that are all variations of the same basic scheme.

The idea behind genetic algorithms is fairly simple. Let me describe the methodology by means of a particular dialect of the genetic algorithms applied to the previously introduced linear system backpropagation network. In that problem, we started out by initializing the weighting matrices and bias vectors to small random numbers. The randomization was necessary in order to avoid stagnation effects during startup. Yet we have no reason to believe that the initial choice is close to optimal or even that the weights remain small during optimization. Thus, the initial weights (parameters) may differ greatly from the optimal weights, causing the optimization to require many iterations. Also, since backpropagation learning is basically a gradient technique, the solution may converge on a local rather than a global minimum, although this didn't happen in this particular example.

Genetic algorithms provide us with a means to determine optimal parameter values more reliably even in "rough terrain," i.e., when applied to systems with a cost function that has many "hills" and "valleys" in the parameter space.

Let us assume that we already know approximate ranges for the optimal weights. In our case, the optimal weights belonging to the $\mathbf{W}^1$ matrix assume values between -2.0 and $+2.0$, those belonging to the $\mathbf{W}^2$ matrix assume values in the range -0.5 to 0.5, those from the $\mathbf{b}^1$ vector are between -0.05 and 0.05, and those from $\mathbf{b}^2$ are bounded by -0.005 and 0.005. I am cheating a little. Since I solved the backpropagation problem already, I know the expected outcome. The more we can restrict the parameter ranges, the faster the genetic algorithm will converge.

We can *categorize* the parameter values by classifying them as 'very small,' 'small,' 'large,' and 'very large,' respectively. In terms of the terminology used in Chapter 13, we transform the formerly quantitative parameter vector into a qualitative parameter vector. A semiquantitative meaning can be associated with the qualitative parameters using fuzzy membership functions as shown in Fig.14.23 for the parameters stored in $\mathbf{W}^1$.

The number of levels can, of course, be chosen freely. In our example, we decided to use four levels, $nlev = 4$. Let us now denote each class by a single upper–case character:

$$\begin{aligned} A &\Leftrightarrow \text{very small} \\ B &\Leftrightarrow \text{small} \\ C &\Leftrightarrow \text{large} \\ D &\Leftrightarrow \text{very large} \end{aligned}$$

Thus, each qualitative parameter can be represented through a single character. We may now write all qualitative parameter values into a long character string such as:

$$ABACCBDADBCBBADCA$$

where the position in the string denotes the particular parameter and the character denotes its class. The length of the string is identical to the number of parameters in the problem. This is our qualitative parameter vector.

Somehow, this string bears a mild resemblance to our genetic code. The individual parameters mimic the amino acids as they alternate within the DNA helix. Of course, this is an extremely simplified version of a genetic code.

In our example, let us choose a hidden layer of length $lhid = 8$. Consequently, the size of $\mathbf{W}^1$ is 8×4, since our system has four inputs, and the size of $\mathbf{b}^1$ is 8. The size of $\mathbf{W}^2$ is 3×8, since the system has three targets, and the size of $\mathbf{b}^2$ is 3. Therefore, the total number of parameters of our problem $npar$ is 67. Thus, the parameter string must be of length 67 as well.

The algorithm starts out with a genetic pool. We arbitrarily generate $nGenSt = 100$ different genetic strings and write them into a matrix of size 100×67. In CTRL–C (or MATLAB), it may be more convenient to represent the genes by integer numbers than by characters. The genetic pool can be created as follows:

$$\begin{aligned} GenPool = &\text{ROUND}(nlev * \text{RAND}(nGenSt, npar) \\ &+ 0.5 * \text{ONES}(nGenSt, npar)) \end{aligned}$$

Initially, we pick 10 arbitrary genetic strings (row vectors) from our genetic pool. We assign quantitative parameter values to them using their respective fuzzy membership functions by drawing random numbers using the fuzzy membership functions as our distribution functions. Next we generate weighting matrices and bias vectors from them by storing the quantitative parameters back into the weighting matrices in their appropriate positions. Finally, we evaluate our feedforward network 301 times using the available input/target pairs for each of these 10 parameter sets. The result will be 10 different *figures of merit*, which are the total errors, the sums of the individual errors for each training pair, found for the given weighting matrices. We sort the 10 performance indices and store them in an array. This gives us a vague first estimate of network performance.

We then arbitrarily pick two genetic strings (the parents) from our pool, draw an integer random number k from a uniform distribution between 1 and 67, and simulate a *crossover*. We pick the first k characters of one parent string (the head), and combine them with the remainder (the tail) of the other parent string. In this way, we obtain a new qualitative genetic string called the child. We then generate quantitative parameter values for the child using the fuzzy membership functions and simulate again. If the resulting performance of the child is worse than the fifth of the ten currently stored performance indices, we simply throw the child away. If it is better than the fifth string in the performance array, but worse than the fourth, we arbitrarily replace one genetic string in the pool with the child and place the newly found performance index in the performance array. The worst performance index is dropped from the performance array. If it is better than the fourth, but worse than the third stored performance, we duplicate the child once and replace two genetic strings in the pool with the two copies of the child. Now 2 of the 100 strings in the pool are (qualitatively) identical twins. If it is better than the third but worse than the second performance, we replace four genetic strings in the pool with the child. If it is better than the second but worse than the first, we replace six genetic strings in the pool with the child. Finally, if the child is the all–time champion, we replace 10 arbitrary genetic strings in the genetic pool with copies of our genius.

We repeat this algorithm many times, deleting poor genetic material while duplicating good material. As time passes, the quality of our genetic pool hopefully improves.

It could happen that the very best combination cannot be generated in this way. For example, the very best genetic string may require an A in position 15. If (by chance) none of the randomly generated 100 genetic strings had an A in that position or if those genes that had an A initially got purged before they could prove themselves, we will never produce a child with an A in position 15. For this reason, we add yet another rule to the genetic game. Once every $nmuta = 50$ iterations, we arbitrarily replace one of the characters in the combined string with a randomly chosen new value, simulating a *mutation*. Eventually, this mutation will generate an A in position 15.

Obviously, this algorithm can be applied in an adaptive learning mode. Our genetic pool will hopefully become better and better, and with it, our forecasting power will increase.

Of course, this algorithm can be improved. For instance, if we notice that a particular parameter stabilizes into one class, we can recategorize the parameter by taking the given class for granted and selecting new subclasses within the given class. In our example, we might notice that the parameter 27, which belongs to $\mathbf{W}^1$, always assumes a qualitative value of C, i.e., its quantitative value is in the range between 0.0 and 1.0. In this case, we can subdivide this range. We now call values between 0.0 and 0.25 'very small' and assign a character of A to them. Values between 0.25 and 0.5 are now called 'small' and obtain a character value of B, etc. I decided to check for recategorization once every 50 iterations, whenever I simulated a mutation. I decided that a recategorization was justified whenever 90% of the genes in one column of *GenPool* had assumed the same value, $nperc = 0.9$.

I ran my genetic algorithm over 800 iterations, which required roughly 2 hours of CPU time on our VAX-11/8700. The execution time was less than that of the backpropagation program since each iteration contains only the forward pass and no backward pass and the length of the hidden layer was reduced from 16 to 8. Thus, for a fair comparison between the two techniques, I should have allowed the genetic algorithm to iterate 2800 times. The results of this optimization are shown in Fig.14.24a.

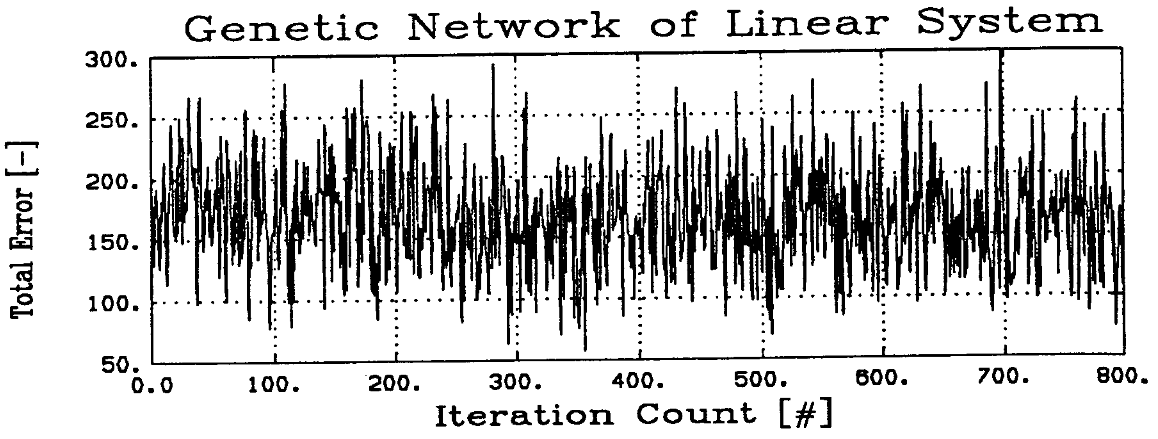


Figure 14.24a. Optimization of linear system with genetic algorithm.

Obviously, this optimization didn't work too well. Figure 14.24b shows a moving average computed over 100 iterations. The first value in Fig.14.24b is the average of the first 100 values of Fig.14.24a, the second value is the average of values 2 through 101 of Fig.14.24a, etc. I computed the moving average using the AVERAGE function of SAPS-II.

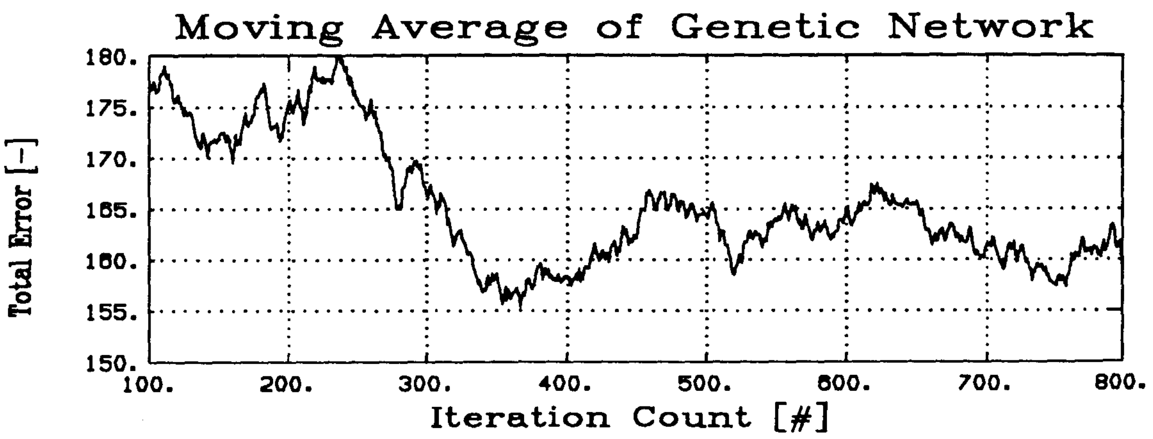


Figure 14.24b. Moving average of linear system.

The genetic algorithm does indeed learn. However, progress is painfully slow. My interpretation of these results is as follows: The terrain (in the parameter space) is very rough. Therefore, since we decided to use only four levels, each level contains both high mountains and deep gulches. Since we only retain the class values but not the quantitative values themselves, we throw away too much information. Consequently, I decided to rerun the optimization with *nlev* = 16. Since there are now more possible outcomes, I decided to consider 30% a solid majority vote, and thus, I reduced *nperc* to 0.3. Another 1.2 CPU–hours later, I obtained the results for the modified algorithm. The simulation required less time because the optimization was terminated after 445 iterations. The results are shown in Fig.14.25.

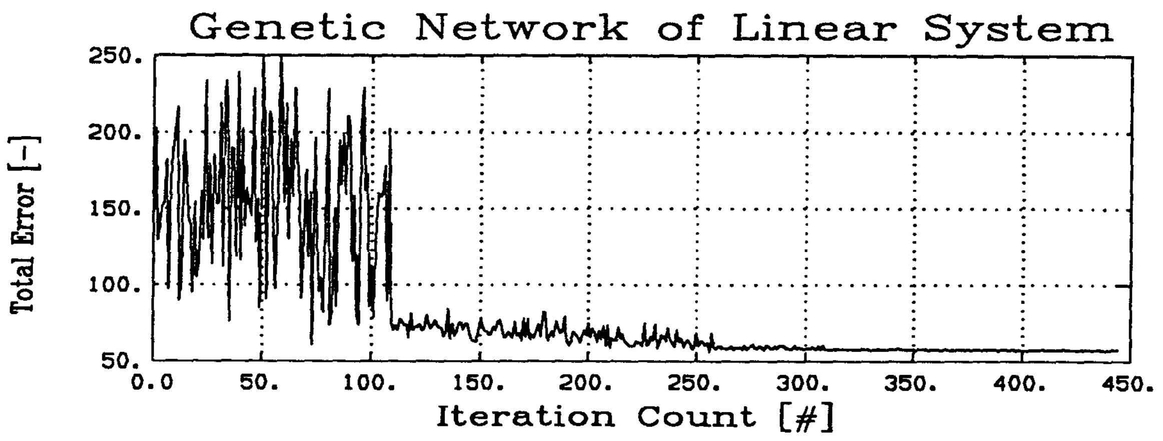


Figure 14.25. Optimization of linear system with genetic algorithm.

This time, the genetic algorithm learned the weights much faster. Unfortunately, good genetic material was weeded out too quickly, and the algorithm ended up in a ditch.

Montana and Davis designed another genetic algorithm specifically for the purpose of training neural feedforward networks [14.25]. They argue against eliminating useful information by coding fuzzy information into our genetic strings. Indeed, both the crossover operator and the mutation operator can be applied to both quantitative (real) and qualitative (fuzzy) parameters. In addition, they designed a set of interesting more–advanced genetic operators. They claim that networks function due to the synergism between weights associated with individual nodes. Thus, instead of applying the crossover algorithm blindly, they keep all the incoming weights of a node intact, and use either those of the father or those of the mother. Also, they consider multiple crossovers. Each node with all its incoming weights is arbitrarily taken from either the father or the mother. Thus, they simulate multiple crossovers of entire features. This makes a lot of sense. Montana and Davis also developed a very interesting concept of node assessment. They evaluate the quality (error) of a network in exactly the same manner that I use, i.e., they add the errors of the network over all training pairs. Then they remove an individual node from the network, i.e., they lobotomize all incoming and outgoing connections of that node by setting the corresponding weights equal to zero and recompute the quality of the modified network. They repeat the same procedure over and over, each time lobotomizing exactly one node. Using this information, they define the node whose presence has the least effect on the overall quality as the *weakest node*. Their mutation algorithm influences all incoming and outgoing weights of the weakest node in the hope of thereby improving the quality of the overall network. Again, this algorithm makes a lot of sense from an engineering point of view. They use a different distribution function for randomizing the initial weights of the network. Initially, they evaluate the quality of the entire genetic pool. However, in each generation, they pair up only one couple (as I do) and produce only one child, which replaces the worst genetic string in the genetic pool (unless it is even worse). The parents are chosen randomly, but with a distribution function such that the second–best genetic string is chosen 0.9 times as often as the best, and the third–best string is chosen 0.9 times as often as the second best, etc. Also this rule makes a lot of sense. Figure 14.26 shows the results of a simulation of the same problem that was discussed earlier, but now using the algorithm by Montana and Davis [14.25].

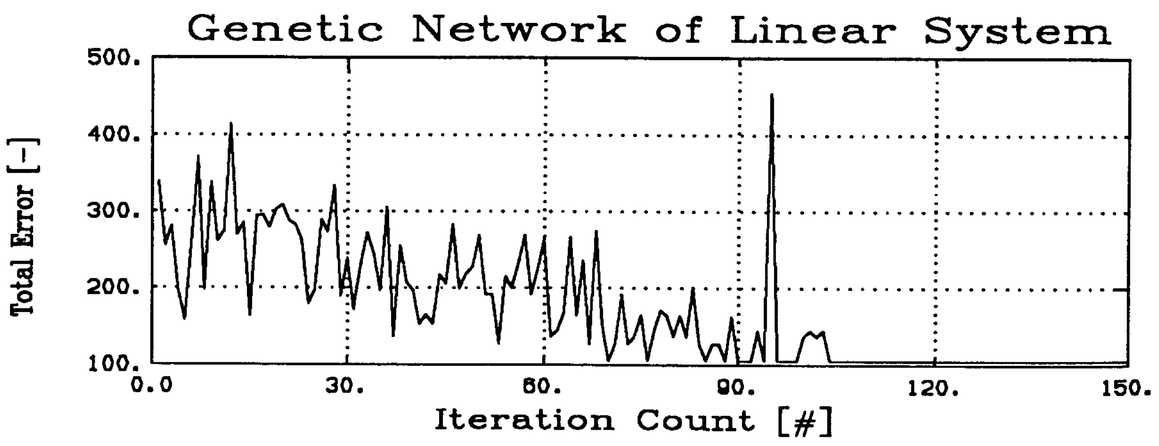


Figure 14.26. Montana–Davis optimization of linear system.

The algorithm is very efficient. The error is reduced quickly and the system learns fast. Unfortunately, it stagnates. Because the real parameter values are stored, no new information is entered into the system except through mutation. The system finds the smallest error among all the combinations of parameters present in the initial genetic pool reliably and quickly, but then it is stuck. The local superman wipes out his competition effectively and efficiently and becomes a tyrant ... unfortunately, he is but a midget in global terms.

This algorithm suffers from the same disease as mine. In both algorithms we were greedy and tried to retain as much good genetic material as possible. We never let a good genetic string die. This is the seed of stagnation.

Even the fittest among us must die for progress to survive.

Goldberg suggested using a genetic algorithm closer to a biological model [14.6]. He proposed the following genetic dialect: We start out with a randomly chosen qualitative genetic pool (as in my algorithm). We evaluate the quality of the entire genetic pool (as in the case of the algorithm by Montana and Davis). We rank the genetic strings according to their quality. We define the *fitness* of a genetic string as:

$$\text{fitness} = \frac{1.0}{\text{total error}} \tag{14.43}$$

We then add up the fitnesses of all genetic strings in the genetic pool and define the *relative fitness* of a genetic string as:

$$\text{relative fitness} = \frac{\text{fitness}}{\text{sum over all fitnesses}} \qquad (14.44)$$

We then replace the entire genetic pool by a new pool in which each genetic string is represented never, once, or multiple times proportional to its relative fitness. Poor genetic strings are removed, while excellent genetic strings are duplicated many times. We then pair the genetic strings up arbitrarily. Each pair produces exactly two offspring, one consisting of the head of the first string concatenated with the tail of the second and the other consisting of the head of the second string concatenated with the tail of the first. We then let the old generation die and replace the entire genetic pool by the new generation. The algorithm is repeated until convergence.

This algorithm grants fit adults many children with varying sex partners, potentially including twin siblings, and deprives unfit adults of the right to reproduce. The algorithm enforces strict birth control.

An obvious disadvantage of this genetic dialect is the need to evaluate the fitness of the entire genetic pool once per generation. Thus, we can optimize this algorithm over 16 generations only if we wish to compare it to the previously advocated dialect. However, I decided to compute 100 iterations anyway. Figure 14.27a shows the results of this optimization. I plotted the mean value of the total errors of all genetic strings in the genetic pool.

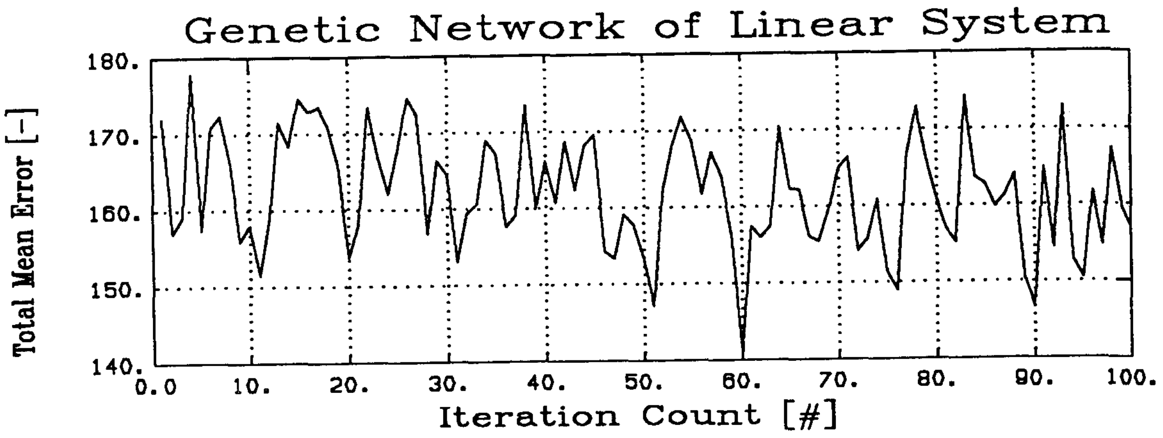


Figure 14.27a. Optimization of linear system with Goldberg's algorithm.

The results are disappointing. If the algorithm has learned anything, the improvement is lost in the noise. I then computed a moving average of the previously displayed mean values over 50 generations. The results are shown in Fig.14.27b.

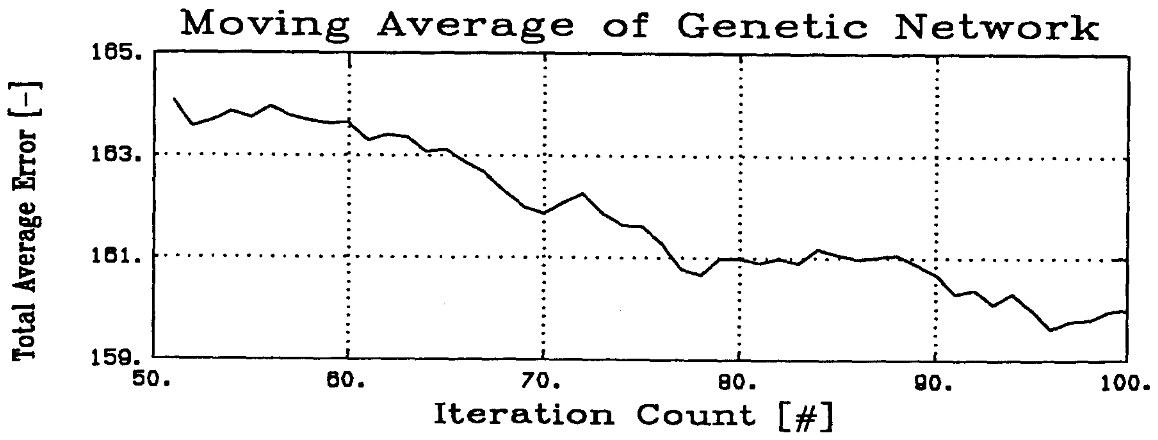


Figure 14.27b. Moving average of Goldberg's algorithm.

Notice that the algorithm does indeed learn. However, the progress is unbelievably slow. I ran this program in batch. It required just over 12 CPU–hours. Obviously, I cannot determine whether this algorithm will stagnate or find the true minimum, but I believe that it will eventually find the true minimum.

The problem with Goldberg's algorithm is the following. The entire idea of the genetic crossover operator bases on the naïve belief that the child of two fit parents is, at least in a statistical sense, a fit child. This belief is justified in nature since the genetic parameters reflect features and the child will inherit an entire feature either from the father or the mother. The overall fitness of a person is defined as the cumulative quality of all of his or her features. Thus, by inheriting features from both parents, fit parents will indeed have fit children. However, in our case, the individual parameters don't represent features. Each parameter influences all features, and each feature is influenced by all parameters. There is no compelling reason to believe that the crossover child of two fit parents is more fit than the average genetic string. Amazingly, the simulation results showed that such a child is indeed statistically more fit than the average genetic string ... but only by a narrow margin. This is why progress was so incredibly slow. It might have been worthwhile to combine Goldberg's algorithm with the previously proposed algorithm by Montana and Davis by combining the genetic operators of the latter (crossover of features and mutation of the weakest node) with the social behavior of the former (replacement of the entire population once per generation), but I was afraid that the director of our computer center would knock me over my head if I continued in this way.

We have just demonstrated the power of evolutionary development. We learned a lesson: For evolution to work, we must permit all individual genetic strings to die, irrespective of their quality. Retaining any individual string invariably leads to stagnation and the evolutionary process comes to a halt. It is the power of ever–changing nonrepetitive variations — we call this phenomenon a chaotic steady–state — which enables the evolutionary process to continue.

In the beginning, there was Chaos.
Chaos nurtures Progress.
Progress enhances Order.
Order tries to defy Chaos at all cost.
... But the day Order wins the final battle
against Chaos, there will be mourning.
'Cause Progress is dead.

Genetic algorithms are a class of simple stochastic optimization techniques. Their behavior was demonstrated here by means of a neural network learning problem. However, no direct relationship exists between the two. Genetic algorithms can be interpreted as one particular implementation of a Monte Carlo optimization technique and can be applied to arbitrary optimization problems. We shall return to this discussion in the companion book of this text in the context of general–purpose parameter–estimation methods. It made sense to introduce the genetic algorithms here due to their inspirational biological foundation.

In the context of artificial neural networks, the genetic algorithm provides us with a systematic and stable technique to optimize arbitrarily constructed networks. This idea is fairly new and hasn't yet been exploited to its full potential. The idea is fruitful, because it removes configuration constraints on artificial neural networks. For instance, it allows us to optimize arbitrarily connected perceptron networks in a general, systematic, and robust (though fairly inefficient) way.