Implementing the SCAN Language by Neural Networks

R. Brause
J.W. Goethe-University, Frankfurt, FRG
brause@informatik.uni-frankfurt.de

Abstract

The real-time encryption of pictures is an important subject for many applications, e.g. television broadcast stations, network security, etc. The paper shows how the previously introduced SCAN encryption method can be easily implemented using binary neural network autoassociative memory.

1 Introduction

For many applications, e.g. pay TV broadcast stations, cheap real-time encryption of pictures is an important subject. Here, the transformation of parallel accessible picture elements (pels) to a sequential TV signal can be used to encode it by a special arrangement of the pel sequence. This corresponds to a special scan order of the picture. Since we have for a picture of n·m pels (n·m)! scan orders, an important feature of the encryption is simplicity. For this kind of problem many schemes have been proposed, for instance a method based on the SCAN context free language for pyramid data structures [Bour87].

In previous papers, the language has been mathematically defined [Alex89] and a parallel implementation has been proposed [Bour89]. Here we investigate the implementation by specific proportions of a neural network model.

2 The SCAN method

Let us shortly review the SCAN methods for picture encryption. The main idea consists of deviding the picture into subpictures. Each subpicture is treated as a picture element (pel). On the next level, each pel can be subdevided into smaller pels, and those again, repeating the devision until the pels are reduced to pixels. Hereby, every devision defines a level *i* of squared subpictures with $n_i=a_i \cdot a_i$ pels of a certain size. In figure 1 an example of three levels is shown with $a_1=2$, $a_2=4$, $a_3=2$.

On each level, we have a certain scan order for the pels, marked by dotted arrows in figure 1. If we denote each order by a symbol L_i, we achieve for the set of scan orders a set of symbols, an alphabet $\{L_i\}$ for the SCAN language. A complete pyramid of N layers is denoted by the expression $L_1a_1\#L_2a_2\#..\#L_Na_N$ with a_i being a power of

2. In our example of figure 1 this is denoted as B2#A4#X2.

The sequential algorithm passes recursively from the top level of the pyramid to the bottom and back again, scanning the pels at each level by the appropriate strategie L_i. Thus, the iterative application of a scan order from one level to the next higher level generates a certain sequential order of pels, and finally, of pixels.

In our example of figure 1, level 1 has n_1=4 pels (index 0 .. 3), level 2 has 16 pels (index 0 .. 15) and level 3 has 4 pixels (index 0 .. 3). Since each pel of level 2 has 4 pixels, the whole picture of level 2 of 16 pels or 4·16=64 pixels and, thereby, each pel of level 1 has 64 pixels and the whole picture of level 1 has 4·64=256 pixels. The whole array of the 256 indices is generated by succesively changing the indices of the pels at each level. This is analog to the mapping of a multi-dimensional array to a one-dimensional array structure for storage purposes. Here, the task is more complex because we have scan pels of different sizes in different orders on a 2-dimensional image. To map the corrsponding pixels or pels (blocks of pixels) properly we scan the image in two different ways: one in the raster scan order of all levels on the new picture and one in a differently defined encryption scan order order on the old picture.

The sequential transformation itself can be easily done by several nested `FOR..DO` loops for the different levels to generate the new picture. The main idea of the encryption lies in the fact that several simple operations are performed in a nested manner to obtain complicated results. Instead of simply incrementing the indices, each new row and column pel index of the scanned old picture is a function of the previous level defined by its scan type. Note that even if all the scan orders are of the raster type "R", the resulting pixel scan order is not identical to sequentially numbering all pixels of the image in one pass. In figure 2 the principal algorithm is shown in a pseudocode for our example of an $Ba_1\#Aa_2\#Xa_3$ SCAN code.

Since this ordering is deterministic, a parallel scheme can be devised by succesively expanding the scan ordering of pels into a scan ordering of pixels [Bour89]. Here, the scan ordering is performed by the parallel transform of the whole set of specially ordered pel indices of one level to the relative pixel index at the next level. At the highest level, the relative pixel indices are the ones of the whole picture, and thus are absolute.

3 The neural network model

For a real-time application like the encoding of TV signals, the index transformation mechanism must be implemented by a simple, high-speed module. Principally, this module must be able to produce a sequence of arbitrary stored numbers on the input of a keyword, e.g. "X" or "R". The conventional solution would provide a high-speed signal processor working on a RAM. For a 1024·1024 (=2^{20}) sized picture we will have to deal with *2^{20}!* possible encodings. Since the processor can not store this huge number of sequences in RAM, it have to be produced by an PROM-based algorithm which involves additional overhead and necessary performance speed of the processor to produce a sequence with a constant bit rate. For a picture of 2^{20} pixels and a non-interleaved refresh rate of 50 Hz this comes up to 20ms for 10^6 pixels or 20 nanoseconds per pixel address. Since this in the order of the picture RAM read cycle time, the necessity for a cheap, non-processor based hardware solution is obvious. This paper proposes a new approach for this kind of problems, using binary artificial neural networks as base address sequencing modules.

3.1 The autoassociative memory

The base mechanism of the sequencer is the well-known one used in associative correlation memory, see [Koh84]. Let us review shortly this model.

Here, the task consists of storing *M* tupels $(\mathbf{x}(1),\mathbf{y}(1))$ .. $(\mathbf{x}(M),\mathbf{y}(M))$ of input patterns $\mathbf{x}(t)=(x_1,..,x_n)$ and desired output patterns $\mathbf{y}(t)=(y_1,..,y_m)$, $x_i,y_i \in \Re$ in such a way that the output is recalled whenever the input key pattern is fed into the system.

The correlation associative memory consists mainly by a matrix $\mathbf{W}$ of real-valued weights (w_{ij}) between input lines x_j and output activity lines z_i. Very often, the suppression of activity noise is obtained by a nonlinear output function $y_i=S(z_i)$.

The storage is obtained by the learning rule for the weight matrix after the t-th presentation of the tupels

$$\mathbf{W}(t) = \mathbf{W}(t-1) + c(t)\mathbf{y}(t)\mathbf{x}(t)^T \qquad (3.1)$$

correlation storage

such that after M presentations the weight matrix becomes

$$\mathbf{W} = \sum_{t=1}^{M} c(t)\mathbf{y}(t)\mathbf{x}(t)^T \qquad (3.2)$$

where the transpose of a vector $\mathbf{x}$ is denoted by $\mathbf{x}^T$. The term $\mathbf{y}\mathbf{x}^T$is the outer vector product, a matrix.

After storing the patterns, the recall process can take place. If we encode all key patterns orthogonally (i.e. $\mathbf{x}(k)^T\mathbf{x}(p)=a$ if k=p, else zero), on the input of a pattern $\mathbf{x}(k)$ the activity $\mathbf{z}$ will become

$$\mathbf{z} = \mathbf{W}\mathbf{x}(k) = \sum_t c(t)\mathbf{y}(t)\mathbf{x}(t)^T\mathbf{x}(k)$$
$$= c(t)\,\mathbf{y}(k)\,\mathbf{x}(k)^T\,\mathbf{x}(k) \qquad (3.3)$$

If we choose appropriate constants $c(t)=[\mathbf{x}(t)^T\mathbf{x}(t)]^{-1}$in eq.(3.1), we will obtain directly the output pattern $\mathbf{y}(k)$ associated to the input $\mathbf{x}(k)$.

The binary thresholded model

This was the base function of the model. Now, if we have *n* components in the input, we can have at most M=*n* orthogonal base vectors or input-output tupels. This is not much. We can increase the number of tupels, if we consider also tupels $\mathbf{x}$ which are not orthogonal to all the already stored ones. Certainly, by the linear activity eq.(3.3) all non-orthogonal components will result in cross-talk or noise between the output activity lines. For binary input and output $x_i,y_i \in \{0,1\}$ we can suppress the additional activity by a suitable threshold s_i

$$y_i = S(z_i) = \begin{cases} 1 & z_i \geq s_i \\ 0 & z_i < s_i \end{cases} \qquad (3.4)$$

which have to be activity-dependend for

arbitrary input **x** (see [Bra88]). In case of |**x**|=const the noise suppression gets more easy and we can choose a constant threshold s_i=|**x**|. By the introduction of a threshold the recall process becomes a classification: a set of different input patterns is mapped on the same output.

In the case of binary weights Palm [Palm80] even showed that in the limit of sparsely coded tupels the associative memory has a capacity of 69% of the ordinary RAM equivalent which is quite effective. Since each binary weight is saturated by only one non-zero contribution, the storage equation (3.2) becomes an OR relation instead of the sum

$$w_{ij} = \bigvee_k y_i(k)x_j(k) = \max_k y_i(k)x_j(k) \quad (3.5)$$

The sparse coding and the condition |**x**|=const can be easily obtained by 1-out-of-d encoders which generate exact one "1" on one of *d* lines if a code number of *0..d-1* is presented [Palm84]. Hereby, a pattern consisting of *k* densely coded digits, each one representing a number in the range *0..d-1*, is transformed to *k* ones on k·d activity lines. Since the number of binary lines grow up considerably by this measure, the step from dense coding to sparse coding can be done only on the chip level. Thus, the kernel memory with its binary weights will be located on chip accompanied by the encoders. This design is shown in figure 3 with additional feedback lines, explained in the next section.

3.2 Sequence generation by autoassociative memory

The associative memory presented so far can also be used to obtain a a sequence of patterns synchronized by a clock cycle. For this, the input pattern **x**(0)=(key, **y**(0)) has been associated to the output pattern **y**(1). At the clock cycle, the input **x**(0) produces the output **y**(1). Now, this is feed back, so the next input pattern will be **x**(1)=(key, **y**(1)). If we have already associated and stored this to **y**(2), the output will become **y**(2) at the next clock cycle and so on. We can close the sequence and make a pattern cycle by defining **y**(0)=(0..0), because the last pattern **y**(M) will produce no output, resulting in **y**(M+1)=(0..0)=**y**(0).

We see that it sufficies to store tuples of the form (key+**y**(t),**y**(t+1)) for all t to generate a sequence by a feedback associate memory. This has been already proposed by Kohonen [Koh84].

4 The encryption system

Now we can easily implement the encoding system discribed in section 2 by the mechanism introduced in section 3.

For a better understanding of the underlying mechanism let us use a simple example, e.g. a B2#R2 scheme. In figure 4 the four pels of the B2 scheme are scanned in the order {0,1,3,2}, denoted in the right upper corner of each pel, whereas within each pel the four pixels are scanned in the order {0,1,2,3}, denoted in the right hand lower corner of each pixel. The resulting pixel index and its corresponding binary equivalent is shown in the table on the right hand side of the figure. Note that the four bits of the pixel addresses are devided into two groups: two high bits for the rows and two low bits for the columns. This can be generalized: since the number of pixels is nxn and n is a power of 2, the whole binary pixel address is a multiple of 2 in each row and we have a multiple of 2 of rows in the whole picture. Since we have n lines, we have $\log_2(n)$ bits for the rows (the high bits) and $\log_2(n)$ bits for the columns (the low bits) to form the $\log_2(nxn)=2\log_2(n)$ address bits for all image pixels. Each row (and each column) is devided into a_1 segments. With a_1 also a power of 2, the highest $\log_2(a_1)$ bits will denote the address bits of the first scan level, whereas the next lower $\log_2(a_2)$ bits denote the bits of the second scan level, and so on.

In figure 5 an overview is shown over the whole system design for the example of figure 1. The system consists of three stages, one for each encoding level. Since we choose the number $n_i=a_i^2$ of elements at each level to be a power of two, each pixel address can consist of three distinguished parts. Each part is generated in parallel by the binary output pattern of each stage. Please note that the correct sequential pixel address has to be obtained by regrouping the column and row parts of the output into the two domains in order to form the pixel address.

As the clock cycle reference and synchronization signal the pixel clock is used. This is one scan step, corresponding to the innermost loop of the sequential algorithm shown in figure 2 . All outer loops are generated by the subsequent binary devider stages which generate one pulse after a loop has finished, resetting the associative memory by activating the appropriate scan pattern scheme number and triggering the next memory step of the next upper level.

Naturally, the necessary binary and analog circuits for sending and receiving the encoded signal and loading or activating the scan pattern numbers are not of interest and therefore not shown here.

5 Simulations

For a simulation of the parallel, binary hardware operations we can use the parallel hardware build in ordinary CPU’s. Since the binary operations of eq. (3.5) can be implemented by logical operations, we have chosen a highly hardware independent, but very efficient program implementation in the C language. In figure 6, an implementation kernel of the readout operation of the binary associative memory is shown which breaks all binary vectors of input, output and storage matrix rows into the length of machine words. All bits in each word are processed in parallel by the CPU. Since the basic operations are very simple, they can be held in the instruction and data caches without problems.

For the simulation, a real picture has been encoded and decoded in the described sequential manner. In figure 7, the original 400x400 picture is shown. In figure 8, for the partially encrypted picture, coded by the

B2 scheme for the high level pels, is shown. We see that the first scan scheme just order the picture on the level of whole pixel blocks. The next figure 9 shows how each block is mixed up by the next level, in this case A5 defined by the order {0, 1, 6, 5, 2, 7, 12, 11, 10, 3, 8, 13, 18, 17, 16, 15, 4, 9, 14, 19, 24, 23, 22, 21, 20}. Finally, figure 10 shows the full B2#A5#I5#R8 encrypted picture. Note that in this case also a non-standard picture with a sidelength not equal to a power of 2 was shown to be encrypted; this is not possible in all cases.

6 Discussion

This paper showed how the SCAN encoding language can be efficiently implemented by ordinary feed-back associative neural networks.

Furthermore, simulations show that the binary version of these sequence-generating modules can also be implemented by conventional computer hardware using boolean operations. These short programs can be held in the cache buffer, allowing very fast operations in ordinary RISC computers.

Nevertheless, for real time operations this is still too slow and special VLSI implementations should be considered. They will represent the core of a more complex system which sends and receives the encryption/decryption keys along with the pixel stream. Since the associations are evoked in only one clock cycle, the keys can rapidly change (for instance during the synchronization time period of ordinary TV frames) without causing any delay in the encoding/decoding process. Thus, the corresponding decryption cards are very flexible and can not be copied by conventional reverse engineering approaches without the special neural network chips which practically prohibits pay TV decoder piracy .

References

[Alex89] C. Alexopoulos: A mathematical modeling of the SCAN language; PhD thesis, University of Patras (1989)

[Bour89] N. Bourbakis, C. C. Alexopoulos, A. Klinger: A parallel implementation of the SCAN language; Comp. Lang., Vol. 14, No.4, pp.239-254, 1989

[Bour87] N. Bourbakis, C. Alexopoulos: Picture data encryption using SCAN patterns; Georg-Mason University, Fairfax, Report GMU-ECE-TR-1987.

[Bra88] R. Brause: Fault Tolerance in Non-linear Networks;
Informatik Fachberichte Vol. 188, pp. 412-433, Springer Verlag 1988

[Koh84] T. Kohonen: Self-Organisation and Associative Memory;
Springer Verlag 1984

[Palm80] G. Palm: On Associative Memory; Biol. Cybernetics, Vol.36, pp. 19-31 (1980)

[Palm84] G. Palm: Local synaptic modification can lead to organized connectivity patterns in associative memory; in: E. Frehland (eds.) Synergetics: from microscopic to macroscopic order, Springer Verlag 1984

```
InIndex:=0; OutIndex:=0;
FOR level1:=0 TO a1*a1-1 DO
   ppp1:=n DIV a1;          (* pixel per pel level1 *)
   InAbsIndex1:=B[level1]; (*define B SCAN order *)
   InPixRow1 :=(InAbsIndex1 DIV a1)*ppp1;
   InPixCol1 :=(InAbsIndex1 MOD a1)*ppp1;
   OutAbsIndex1:=level1;
   OutPixRow1:=(OutAbsIndex1 DIV a1)*ppp1;
   OutPixCol1:=(OutAbsIndex1 MOD a1)*ppp1;

   FOR level2:=0 TO a2*a2-1 DO
       ppp2:=n DIV (a2*a1);     (* pixel per pel level2 *)
       InAbsIndex2:=A[level2]; (*define A SCAN order *)
       InPixRow2 :=(InAbsIndex2 DIV a2)*ppp2;
       InPixCol2 :=(InAbsIndex2 MOD a2)*ppp2;
       OutAbsIndex2:=level2;
       OutPixRow2:=(OutAbsIndex2 DIV a2)*ppp2;
       OutPixCol2:=(OutAbsIndex2 MOD a2)*ppp2;

       FOR level3:=0 TO a3*a3-1 DO
         ppp3:=n DIV (a3*a2*a1);   (* pixel per pel level3 *)
         InAbsIndex3:=X[level3];    (*define X SCAN order *)
         InPixRow3 :=(InAbsIndex3 DIV a3)*ppp3;
         InPixCol3 :=(InAbsIndex3 MOD a3)*ppp3;
         OutAbsIndex3:=level3;
         OutPixRow3:=(OutAbsIndex3 DIV a3)*ppp3;
         OutPixCol3:=(OutAbsIndex3 MOD a3)*ppp3;

           InPixRow := InPixRow1+InPixRow2+InPixRow3;
           InPixCol := InPixCol1+InPixCol2+InPixCol3;
           InIndex  := InPixRow*n + InPixCol;

           OutPixRow := OutPixRow1+OutPixRow2+OutPixRow3;
           OutPixCol := OutPixCol1+OutPixCol2+OutPixCol3;
           OutIndex  := OutPixRow*n + OutPixCol;

           NewPicture[OutIndex] := OldPicture[InIndex];

       END; (* level3 *)
   END; (* level2 *)
END; (*level1 *)
```

Fig.2 The loop structure for sequential SCAN encoding

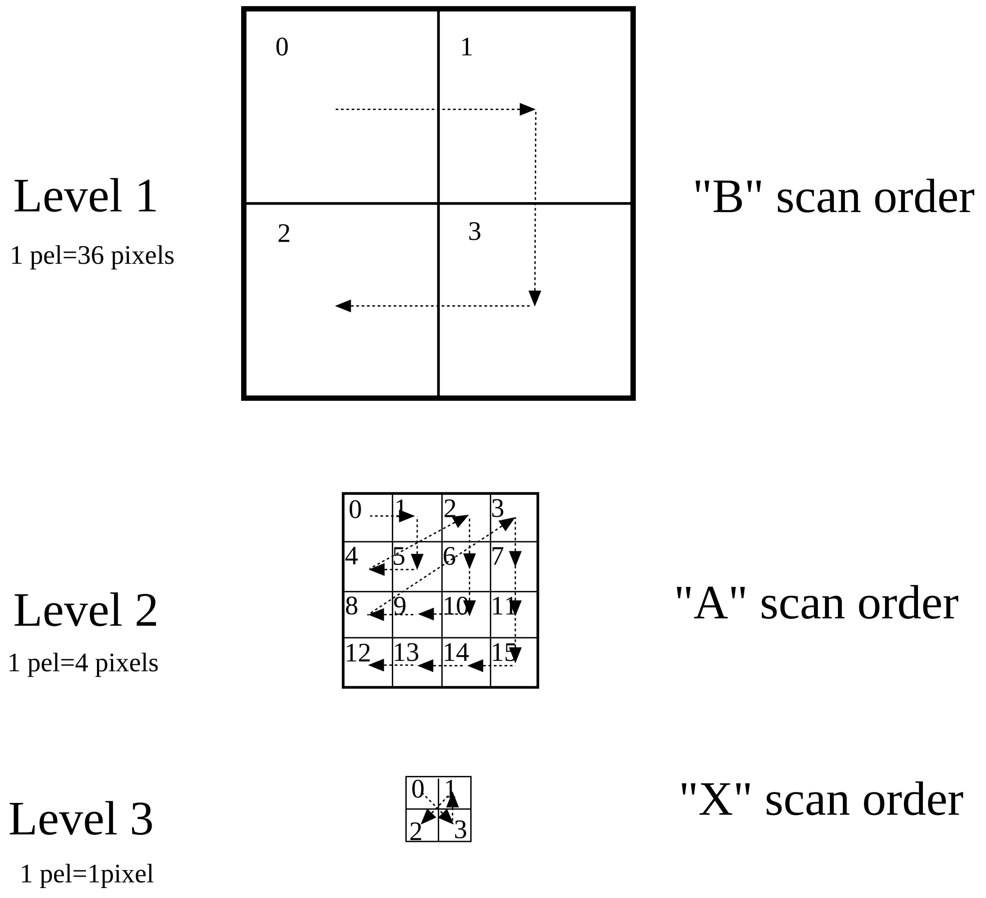


Fig.1 A scan pyramid with different pel sizes, denoted by B2#A4#X2

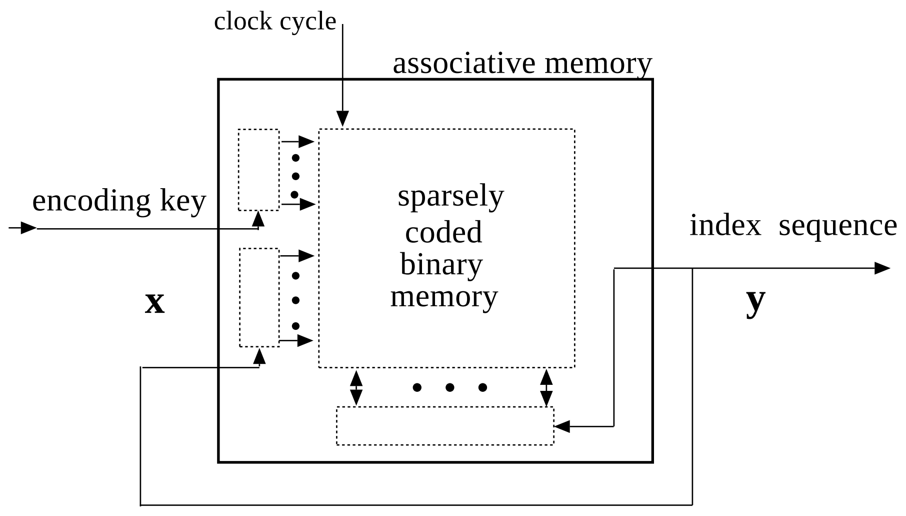
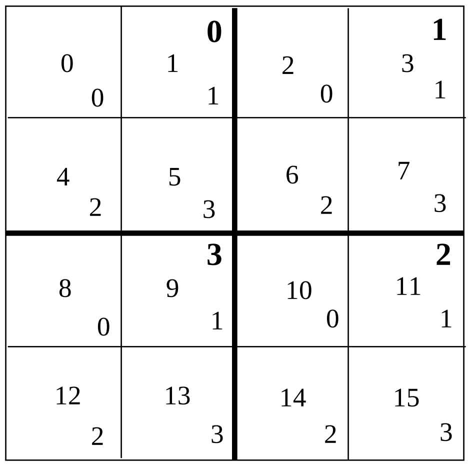


Fig.3 The binary autoassociative memory layout



B	R	pix index	row	col bits
0	0	0	0 0	0 0
0	1	1	0 0	0 1
0	2	4	0 1	0 0
0	3	5	0 1	0 1
1	0	2	0 0	1 0
1	1	3	0 0	1 1
1	2	6	0 1	1 0
1	3	7	0 1	1 1
2	0	10	1 0	1 0
2	1	11	1 0	1 1
2	2	14	1 1	1 0
2	3	15	1 1	1 1
3	0	8	1 0	0 0
3	1	9	1 0	0 1
3	2	12	1 1	0 0
3	3	13	1 1	0 1

Fig.4 The B2#R2 SCAN sequential pixel lookup-table

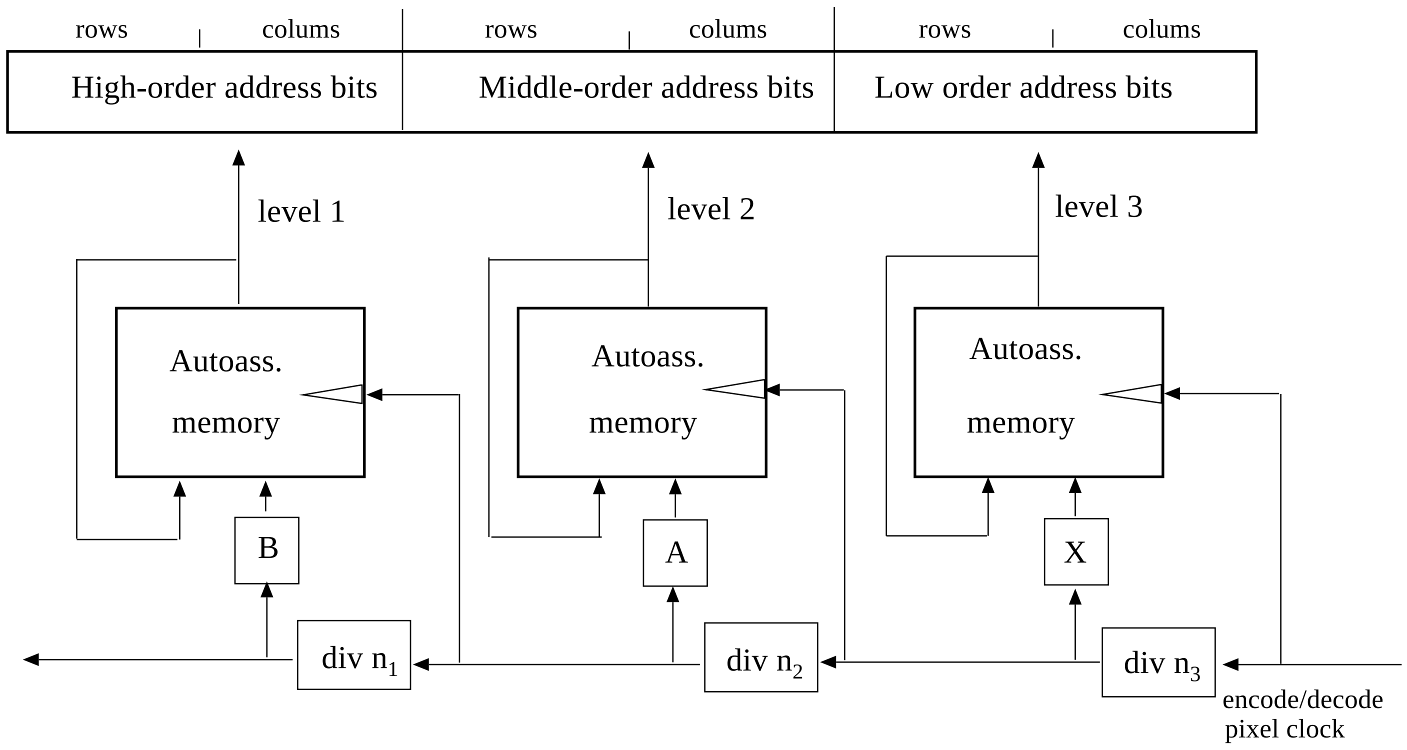


Fig.5 The neural network encoding system

```
DEFINE n 64                 /* number of binary input lines */
DEFINE m 64                 /* number of binary output lines */
int WordSize = sizeof(long int)*8;
int NumOfInputWords  = n/WordSize -1;
int NumOfOutputWords = m/WordSize -1;
int theta = WordSize;
long int OutMaskInit = 1;

main () {                   /* initialization part */
  OutMaskInit <<= WordSize; /* set highest bit of output mask */
}

readout (X,Y,M)
register long int *X,*Y,*M;
{
  register long int CorPat, OutMask;
  register short int cnt, sum, i, ii, j;

  for (i=NumOfOutputWords; i>=0; --i) {          /* all output words */
      *Y = 0;                          /* set output word bits to 0 */
      OutMask = OutMaskInit;           /* start with highest bit */
      for (ii=WordSize-1; ii>=0; --ii){ /* all bits of an output word */
         sum = 0;
         for (j=NumOfInputWords;j>=0;--j){ /* all input words */
             CorPat = (*M)&(*X);       /* memory Mij AND input Xj */
             for (cnt=WordSize-1;cnt>=0;--cnt){/* count # of 1's */
                if (CorPat<0) sum++;   /* test if sign bit set */
                CorPat <<= 1;          /* shift for next data bit  */
             }
             X++;                      /* next word of input */
             M++;                      /* next word of memory row */
         }
         IF (sum >= theta) *Y |= OutMask; /* set bit in output */
         OutMask >>= 1;                /* select next output bit */
      }                                /* next output bit */
      Y++;                             /* next word of output */
  }
} /* end readout */
```

Fig. 6 An efficient binary readout associative memory emulation procedure in the "C" language

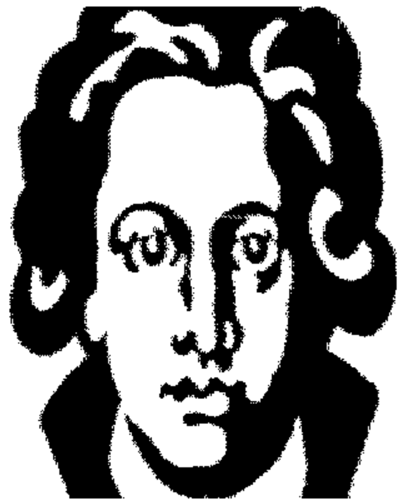


Fig. 7 The original 400x400 picture

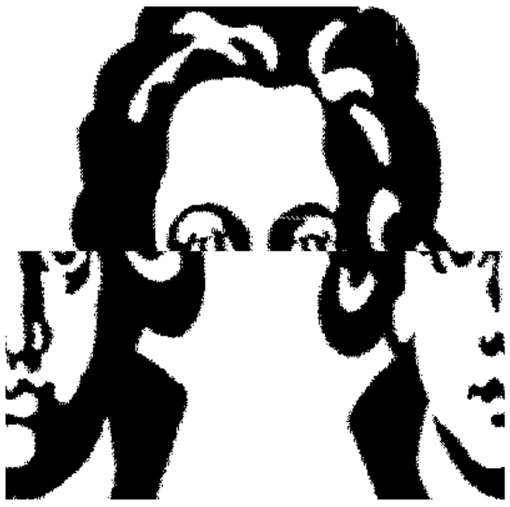


Fig. 8 The picture, scrambled by B2

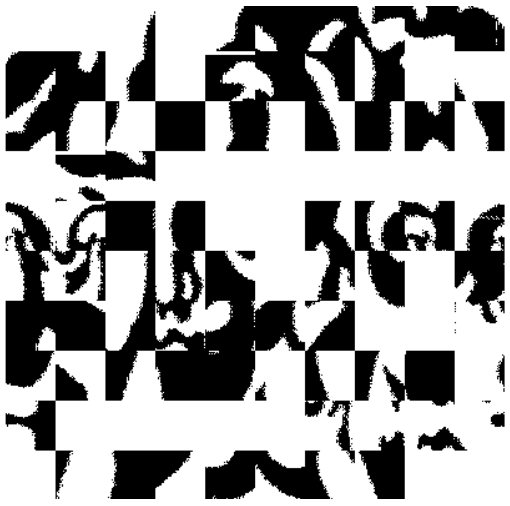


Fig. 9 The picture, scrambled by B2#A5

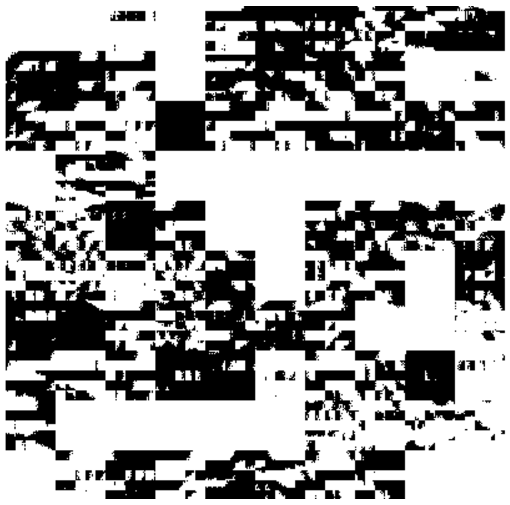


Fig. 10 The picture, scrambled by B2#A5#I5#R8