Chapter 3

Neural Networks

Literature

- W. KINZEL and G. REENTS, *Physik per Computer*, Spektrum akademischer Verlag, 1996.
- J. HERTZ, A. KROGH, and R.G. PALMER, *Introduction to the theory of neural computers*, Addison Wesley 1991.
- B. MÜLLER, J. REINHARDT, and M. STRICKLAND, *Neural Networks: An Introduction*, Springer 1995.
- CH.M. BISHOP, *Neural Networks for Pattern Recognition*, Oxford University Press 1995.

Can computers simulate our brain? Everybody, considering this question will readily realize that the answer is NO. Recent Computers are extremely powerful, but up to now, nobody knows how the information processing between the 10^{11} nerve-cells (neurons) and the respective 10^{14} links (synapses) can lead to processes like: training, cognition, recognition, thinking, emotions, self-reflection, etc.

We know little, but many scientists and engineers are trying hard to develop computer programs that try to mimic the architecture and the behavior of the human brain. Such algorithms and the corresponding hardware realizations are called neural networks or rather neural computers. As a matter of fact, their features are significantly different from those of conventional computers.

A neural net, or its emulation on a conventional computer, gathers information and stores it by modifying the strengths of the synaptic links. There is no such thing as a program. At each stage of the training process or thereafter, the neural computer is capable of generalizing the acquired

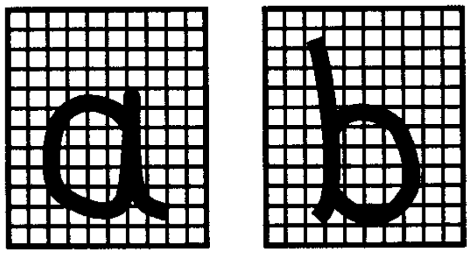


Figure 3.1: Illustration of two hypothetical images representing handwritten versions of the characters 'a' and 'b'. Each image is described by an array of pixel values x_i which range from 0 to 1 according to the fraction of the pixel square occupied by black ink.

knowledge. In other words, the neural net recognizes rules in examples (training sets).

The information in neural nets is spread over the entire net of synapses, in contrast to conventional computers, where information is stored locally. Hence, if parts of the neural net are defect, the neural computer still works and is still able to reason based on the remaining information. The logic is not yes or no, but rather more or less.

Numerous impressive demonstrations of neural networks have been presented: speaking English texts, recognition of digits and letters, playing games (e.g. backgammon), analysis of spectra (such as: quantification of the chemical composition of gases), detection of defective engines by sound, prediction of stock prices. It appeared that neural nets are often comparable or even better than complicated and elaborate conventional computer codes.

3.1 Pattern Recognition

The term *pattern recognition* encompasses a wide range of information processing problems, from speech recognition and identification of handwritten characters, to fault detection in machinery and medical diagnosis. The most general, and most natural, framework in which to formulate solutions to pattern recognition problems is a statistical one, which recognizes the probabilistic nature both of the information we seek to process, and of the form in which we should express the results.

We can introduce many of the fundamental concepts of statistical pattern recognition by considering a simple, hypothetical, problem of distinguishing handwritten versions of the characters 'a' and 'b'. Images of the characters might be captured by a television camera and fed to a computer, and we seek an algorithm which can distinguish as reliably as possible between the two characters. An image is represented by an array of pixels, as illustrated in Fig. 3.1, each of which carries an associated value which we shall denote by x_i (where the index i labels the individual pixels). The value of x_i might, for instance, range from 0 for a completely white pixel to 1 for a completely black pixel. It is convenient to gather the x_i variables together and denote them by a single vector $\mathbf{x}^T = (x_1, \ldots, x_N)^T$, where N is the total number of such variables.

The goal in this classification problem is to develop an algorithm which will assign any image, represented by a vector $\mathbf{x}$, to one of two classes which we shall denote C_k, $k = 1, 2$, so that C_1 corresponds to the character 'a' and C_2 corresponds to 'b'. We shall also suppose that we are provided with a large number of examples of images corresponding to both 'a' and 'b' which have already been classified by a human. Such a collection is called *data set* or *sample*.

One problem we face stems from the high dimensionality of the data which we are collecting. For a typical image size of 256×256 pixels each image can be represented as a point in an N-dimensional space, where $N = 65536$. The axes of this space represent the gray-level values of the corresponding pixels which in this example might be presented by 8-bit numbers. In principle we might think of storing every possible image together with its corresponding class label. In practice, of course, this is completely impractical due to the very large number of possible images: for a 256×256 image with 8-bit pixel values, there would be $2^{8\times256\times256} \simeq 10^{158000}$ different images. By contrast, we might typically have a few thousand examples in our training set. Thus, the classifier system must be designed so as to be able to classify correctly a previously unseen image vector. One technique to help alleviate such problems is to combine input variables together to make a smaller number of new variables called *features*. In the present example, we could, for instance, evaluate the ratio of the height of the character to its width (denoted $\tilde{x}_1$) since we might expect that characters from class C_2 will typically have larger values of $\tilde{x}_1$ than characters from class C_2.

We can now represent the outcome of the classification in terms of a variable y which takes the value 1 if the image is classified as C_1 and the value 0 if it is classified as C_2. Thus, the overall system can be viewed as a mapping from a set of input variables $x_1, \ldots, x_N$, representing the pixel intensities, to an output variable y representing the class label. In more complex problems there may be several output variables y_k, $k = 1, \ldots, c$.

In general it will not be possible to determine a suitable form for the required mapping, except with the help of a data set of examples. The mapping is therefore modeled in terms of some mathematical function which contains a number of adjustable parameters whose values are determined with the help of the data set. We can write such functions in the form

$$y_k = y_k(\mathbf{x}; \mathbf{w}), \tag{3.1}$$

where $\mathbf{w}$ denotes the vector of parameters. A neural network model can be regarded as a particular choice for the set of functions $y_k(\mathbf{x}; \mathbf{w})$. In this case, the parameters comprising $\mathbf{w}$ are often called *weights*. The process of determining the values for these parameters on the basis of the data set is called *learning* or *training* and for this reason the data set of examples is generally referred to as a *training set*.

Polynomial Curve Fitting

Many of the important issues concerning the application of neural networks can be introduced in the simpler context of polynomial curve fitting. Here, the problem is to fit a polynomial to a set of N data points by the technique of minimizing an error function. We consider the M-the order polynomial given by

$$y(x) = w_0 + w_1 x + \cdots + w_M x^M = \sum_{j=0}^{M} w_j x^j. \tag{3.2}$$

This can be regarded as a non-linear mapping which takes x as input and produces y as output. The precise form of the function $y(x)$ is determined by the values of the parameters $w_0, \ldots, w_M$ which are analogous to the weights in a neural network. We denote the set of parameters $w_0, \ldots, w_M$ by a vector $\mathbf{w}$ and the polynomial can then be written as a functional mapping in the form $y = y(x, \mathbf{w})$ as was done for more general non-linear mappings in Eq. (3.1).

We shall label the data with the index $1, \ldots, N$, so that each data point consists of a value x, denoted by $x^{(n)}$, and a corresponding desired value for the output y, which we shall denote $t^{(n)}$ because these desired output values are called *target values*. In oder to find suitable values for the coefficients in the polynomial, it is convenient to consider the error between the desired output $t^{(n)}$ for a particular input $x^{(n)}$ and the corresponding value predicted by the polynomial function given by $y(x^{(n)}; \mathbf{w})$. Standard curve-fitting procedures involve minimizing the square of this error, summed over all data points, given by

$$E = \frac{1}{2} \sum_{n=1}^{N} \left\{ y(x^{(n)}; \mathbf{w}) - t^{(n)} \right\}^2. \tag{3.3}$$

We can regard E as being a function of $\mathbf{w}$, and so the polynomial can be fitted to the data by choosing a value for $\mathbf{w}$ which minimizes E. Note that the polynomial (3.2) is a linear function of the parameters $\mathbf{w}$ and so (3.3) is a quadratic function of $\mathbf{w}$. This means that the minimum of E can be found in terms of the solution of a set of linear algebraic equations. Functions which depend linearly on the adaptive parameters are called *linear models*, even though they may be non-linear functions of the original input variables.

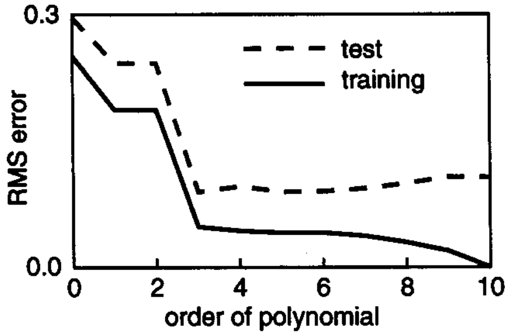


Figure 3.2: Plots of the RMS error (3.4) as a function of the order of the polynomial for both training and test sets. The error with respect to the training set decreases monotonically with M, while the error in making predictions for new data (as given by the test set) shows a minimum.

Generalization

In order to assess the capability of the polynomial to generalize to new data, it is convenient to consider the root-mean-square (RMS) error

$$E^{\mathrm{RMS}} = \sqrt{\frac{1}{N}\sum_{n=1}^{N}\{y(x^{(n)};\mathbf{w}^*) - t^{(n)}\}^2}, \tag{3.4}$$

where $\mathbf{w}^*$ represents the vector of coefficients corresponding to the minimum of the error function, so that $y(x;\mathbf{w}^*)$ represents the fitted polynomial. For the purpose of evaluating the effectiveness of the polynomial at predicting data, this is a more convenient quantity to consider than the original sum-of-squares error (3.3) since the strong dependence on the number of data points has been removed. Fig. 3.2 shows a plot of E^{RMS} for both the training data set and the test data set as a function of the order M of the polynomial. We see that the training set error decreases steadily as the order of the polynomial increases. The test set error, however, reaches a minimum at some smaller value of M and thereafter increases as the order of the polynomial is increased. The ability of the polynomial to generalize to new data (*i.e.:* to the test set) therefore reaches an optimum value for a polynomial of a particular degree of complexity. Generalization is a trade-off between *bias* and *variance*. A model which has little flexibility, such as the linear polynomial, has a high bias, while a model which has too much flexibility, such as a high order polynomial, has a high variance. The point of best generalization is determined by the trade-off between these two competing properties and occurs when the number of degrees of freedom in the model is relatively small compared to the size of the data set.

The common denominator of the two examples discussed here is the basic concept of a simple neural net, the *perceptron*.

3.2 Perceptron

It was developed, applied and studied already in the 60-s. Here we will study the perceptron guided by a simple example. Neurons are modeled by binary variables $S_i \in \{1, -1\}$, where $+1$ represents an active and -1 an inactive state.

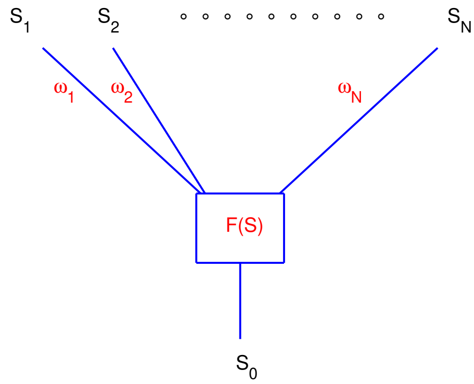


Figure 3.3: *Perceptron*

The perceptron consists of an input layer of neurons

$$\mathcal{S} = \{S_1, S_2, \dots, S_N\},$$

and one output neuron S_0, which is coupled to all input neurons via synaptic weights

$$\omega_i \,;\, i = 1, \dots, N.$$

The synaptic weights $\omega_i \in \mathbb{R}$ model the strength with which the incoming signals S_i of the neurons are transformed into the output signal S_0. The synapses describe bio-chemical processes, with $\omega_i > 0$ ($\omega < 0$) standing for enhancement (suppression) of the respective incoming signal. Like in real nerve cells, the output neuron reacts on the sum of the activities of all neurons which are coupled via synaptic weights. There are $M = 2^N$ different input sequences $\mathcal{S}_\alpha$; $(\alpha = 1, 2, \dots M)$, to each of which the output neuron can react with either $+1$ or -1. The output is determined by the synaptic weights and is described by the *boolean* function F:

$$F : \qquad \{+1, -1\}^N \rightarrowtail \{+1, -1\}.$$

The function is entirely described by the specification of the output value for each of the $M = 2^N$ input sequences $\mathcal{S}_\alpha$. Hence, there are

$$L = 2^M = 2^{2^N}$$

different boolean functions. Already in 1943 complex and time-dependent biochemical processes have been approximated by a simple mathematical formula

$$S_0 = \text{sign}\left(\sum_{i=1}^{N} S_i\,\omega_i\right). \tag{3.5}$$

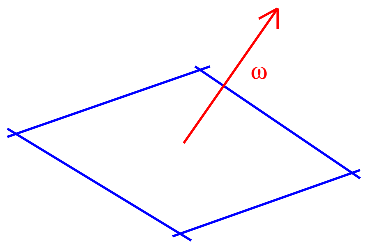
If

$$\sum_{i=1}^{N} S_i\,\omega_i = \boldsymbol{\omega}^T \boldsymbol{S} > 0 \tag{3.6}$$

the neuron fires, *i.e.:* $S_0 = +1$, otherwise it rests, *i.e.:* $S_0 = -1$. In (3.6) we used vector-notation. (3.5) covers only a subset of all possible boolean functions, the so-called linear separable boolean functions. It can be shown that there are less than

$$L' = 2^{N^2}$$

separable boolean functions. For $N = 10$ the numbers are $L = 10^{308}$ and $L' = 10^{30}$, respectively.



Linear separable functions can be visualized geometrically. The output neuron is $S_0 = +1$ if the point characterized by the vector $\boldsymbol{S}$ lies above the hyper-plain that is defined by the normal form

$$\boldsymbol{\omega}^T \boldsymbol{S} = 0,$$

as illustrated in the figure. The normal vector is given by the normalized vector $\boldsymbol{\omega}$.

3.3 Training and generalization

Like in real nerve cells, the neural net learns by adjusting the synaptic weights. Given a set of input/output pairs

$$\left(\boldsymbol{S}^{(\nu)}, S_0^{(\nu)}\right); \qquad \nu = 1, 2, \ldots, L,$$

the perceptron tries to determine the synaptic weights such that it yields the correct output

$$\text{sign}\,(\boldsymbol{\omega}^T \boldsymbol{S}^{(\nu)}) = S_0^{(\nu)} \tag{3.7}$$

for all examples, $\nu = 1, 2, \ldots, L$. Equation (3.7) can be simplified upon multiplying both sides by $S_0^{(\nu)}$

$$S_0^{(\nu)}\ \text{sign}\,(\boldsymbol{\omega}^T \boldsymbol{S}^{(\nu)}) = \text{sign}\,(S_0^{(\nu)}\ \boldsymbol{\omega}^T \boldsymbol{S}^{(\nu)}) = \left(S_0^{(\nu)}\right)^2 = 1.$$

It is expedient to introduce modified training vectors

$$\boldsymbol{\xi}^{(\nu)} = \boldsymbol{S}^{(\nu)}\, S_0^{(\nu)}, \tag{3.8}$$

and the correct response is now characterized by

$$\boldsymbol{\omega}^T \,\boldsymbol{\xi}^{(\nu)} > 0. \tag{3.9}$$

The goal of the training phase is to adjust the synaptic weights such that Equation (3.9) is fulfilled for *all* elements of the training set. One possible approach is to minimize the cost function $C(\boldsymbol{\omega}) = N_m$, which counts the number N_m of wrong predictions. The minimization could be achieved by stochastic optimization algorithms.

Another efficient approach is given by an iteration scheme, defined by the update rule

PERCEPTRON RULE

$$\Delta\boldsymbol{\omega} = \begin{cases} \frac{1}{N}\,\boldsymbol{\xi}^{(\nu)} & \text{if} \quad \boldsymbol{\omega}^T\boldsymbol{\xi}^{(\nu)} \leq 0 \\ 0 & \text{otherwise.} \end{cases} \tag{3.10}$$

In other words the weights are only modified, if the prediction was incorrect. In the latter case the new weight reads $\boldsymbol{\omega}' = \boldsymbol{\omega} + \Delta\boldsymbol{\omega}$. ROSENBLUTH proved in 1960 that it takes a finite number of steps to learn all examples, provided they can be trained at all, *i.e.:* if there exists a vector $\boldsymbol{\omega}$ satisfying (3.9) for all $\nu = 1, 2, \ldots, L$.

Proof:

We assume that there is a vector $\boldsymbol{\omega}^*$ satisfying

$$(\boldsymbol{\omega}^*)^T\boldsymbol{\xi}^{(\nu)} > 0 \qquad \forall\nu.$$

This implies that all 'points' $\boldsymbol{\xi}^{(\nu)}$ are on one side of the hyper-plain and the point closest to it has finite distance C from the plain, *i.e.:*

$$(\boldsymbol{\omega}^*)^T\boldsymbol{\xi}^{(\nu)} \geq C > 0 \qquad \forall\nu. \tag{3.11}$$

We start the training process with $\boldsymbol{\omega}(t=0) = 0$ and increment the counter t each time the prediction is wrong and the weight has to be modified according to the perceptron rule. Let ν be the wrongly predicted example at time t, *i.e.:*

$$\boldsymbol{\omega}(t)^T\boldsymbol{\xi}^{(\nu)} \leq 0, \tag{3.12}$$

then

$$\boldsymbol{\omega}(t+1) = \boldsymbol{\omega}(t) + \frac{1}{N}\,\boldsymbol{\xi}^{(\nu)}. \tag{3.13}$$

The next weight has the norm

$$\|\boldsymbol{\omega}(t+1)\|^2 = \|\boldsymbol{\omega}(t)\|^2 + \frac{2}{N}\,\boldsymbol{\omega}(t)^T\boldsymbol{\xi}^{(\nu)} + \frac{1}{N^2}\,\|\boldsymbol{\xi}^{(\nu)}\|^2.$$

The norm of the input signal vector ($\boldsymbol{\xi}^{(\nu)} = \mathbf{S}^{(\nu)}S_0^{(\nu)}$) is given by

$$\|\boldsymbol{\xi}^{(\nu)}\|^2 = \underbrace{\left|S_0^{(\nu)}\right|^2}_{=1}\ \underbrace{\|\boldsymbol{S}^{(\nu)}\|^2}_{N} = N,$$

and along with (3.12) we have

$$\|\boldsymbol{\omega}(t+1)\|^2 \leq \|\boldsymbol{\omega}(t)\|^2 + \frac{1}{N}.$$

Starting from $\|\boldsymbol{\omega}(0)\|^2 = 0$ the sequence of norms reads

$$\begin{aligned}
&\|\boldsymbol{\omega}(1)\|^2 \leq \frac{1}{N}\\
&\|\boldsymbol{\omega}(2)\|^2 \leq \|\boldsymbol{\omega}(1)\|^2 + \frac{1}{N} \leq \frac{2}{N},
\end{aligned}$$

which obeys the general formula

$$\|\boldsymbol{\omega}(t)\|^2 \leq \frac{t}{N}. \tag{3.14}$$

Next we derive a lower bound for $(\boldsymbol{\omega}^*)^T\boldsymbol{\omega}(t)$

$$\begin{aligned}
(\boldsymbol{\omega}^*)^T\boldsymbol{\omega}(t+1) &\overset{(3.13)}{=} (\boldsymbol{\omega}^*)^T\boldsymbol{\omega}(t) + \frac{1}{N}\underbrace{(\boldsymbol{\omega}^*)^T\boldsymbol{\xi}^{(\nu)}}_{\geq C}\\
&\geq (\boldsymbol{\omega}^*)^T\boldsymbol{\omega}(t) + \frac{C}{N}.
\end{aligned}$$

Again starting with $\boldsymbol{\omega}(0) = 0$ we obtain

$$(\boldsymbol{\omega}^*)^T\boldsymbol{\omega}(t) \geq \frac{C\,t}{N}. \tag{3.15}$$

Combining the Schwarz inequality

$$\|\boldsymbol{\omega}(t)\|^2\|\boldsymbol{\omega}^*\|^2 \geq \left|\boldsymbol{\omega}(t)^T\boldsymbol{\omega}^*\right|^2$$

with (3.14) and (3.15) yields

$$\begin{gathered}
\|\boldsymbol{\omega}(t)\|^2\|\boldsymbol{\omega}^*\|^2 \geq \left|\boldsymbol{\omega}(t)^T\boldsymbol{\omega}^*\right|^2\\
\frac{t}{N}\|\boldsymbol{\omega}^*\|^2 \overset{(3.14)}{\geq} \|\boldsymbol{\omega}(t)\|^2\|\boldsymbol{\omega}^*\|^2 \geq \left|\boldsymbol{\omega}(t)^T\boldsymbol{\omega}^*\right|^2 \overset{(3.15)}{\geq} \left(\frac{Ct}{N}\right)^2\\
\Rightarrow\\
\|\boldsymbol{\omega}^*\|^2\frac{\cancel{t}}{\cancel{N}} \geq C^2\left(\frac{t}{N}\right)^{\cancel{2}}.
\end{gathered}$$

$$t \le N \frac{\|\boldsymbol{\omega}^*\|^2}{C^2}. \tag{3.16}$$

This ends the proof that the number of steps, needed to learn all examples, provided they can be memorized by the neural net, is finite. If we use a normalized vector $\hat{\boldsymbol{\omega}}^*$ for the hyper-plain then Eq. (3.11) reads

$$(\hat{\boldsymbol{\omega}}^*)^T \boldsymbol{\xi}^{(\nu)} \ge \frac{C}{\|\boldsymbol{\omega}^*\|} =: C' > 0 \qquad \forall \nu,$$

where C' now stands for the shortest distance of the points $\boldsymbol{S}^{(\nu)}$ from the hyper-plain, and Eq. (3.16) becomes

MAXIMUM NUMBER OF TRAINING CYCLES

$$t \le \frac{N}{C'^2} =: t^*. \tag{3.17}$$

The number of steps increases linearly with the size of the neural net and inversely with the shortest distance squared. The points closest to the hyper-plain are the ones which are the most difficult to memorize since the response of the output neuron depends on which side of the hyper-plain the point lies.

The points $\boldsymbol{S}^{(\nu)}$ lie equally spaced on a hyper-sphere of radius $\sqrt{N}$. The more examples there are, the closer the points approach the hyper-plain and, hence, the longer it takes to learn.

A single application of the training rule (3.10) does not guarantee that the example will be correctly predicted next time. It is therefore necessary to repeat the training process several times, either by cycling through all examples several times and/or by training each example repeatedly.

After the training phase is finished the neural net can be tested, how well the neural computer predicts the signals of the output neuron $S_0^{(\nu)}$, given a set of M unknown (to the computer) input vectors

$$\boldsymbol{S}^{(\nu)}, \quad \nu = 1, 2, \ldots, M.$$

To this end we define the relative rate of success

$$r = \frac{\text{number of correct predictions}}{\text{total number of predictions}} = \frac{1}{M} \sum_{\nu=1}^{M} \theta \left[S_0^{(\nu)} (\boldsymbol{\omega}^*)^T \boldsymbol{S}^{(\nu)} \right]. \tag{3.18}$$

3.4 Analysis of time series

We consider a particularly simple application: time series of binary values ± 1, e.g. the sequence

$$\mathcal{F} = \big(+1, +1, -1, -1, -1, -1, +1, \ldots \big).$$

The perceptron reads a window of N bits

$$\boldsymbol{S}^{(\nu)} = \big(F_\nu, F_{\nu+1}, \ldots, F_{\nu+N-1}\big)$$

and predicts the next bit

$$S_0^{(\nu)} = F_{\nu+N}.$$

If the sequence possesses a periodicity of length K then there are K different examples and for $K < N$ the perceptron should be able to learn this pattern perfectly and make the correct predictions for the forthcoming bits.

If the predictions are made at random, the correct answer represents a random variable obeying a binomial distribution. The probability for m correct out of M predictions is

$$p(m|q, M) = \binom{M}{m}\, q^m\, (1-q)^{M-m},$$

with $q = 1/2$. The mean number of correct predictions is

$$\langle m \rangle = Mq = M/2$$

and the corresponding variance reads

$$\langle (\Delta m)^2 \rangle = Mq(1-q) = M/4.$$

The upper limit for the 2-σ region is

$$\frac{M}{2} + 2\frac{\sqrt{M}}{2},$$

and the corresponding relative factor of success is

$$r_{2\sigma} = \frac{1}{2}\,\big(1 + \frac{2}{\sqrt{M}}\big).$$

The probability that by random predictions the relative factor of success is greater or equal to $r_{2\sigma}$ is 2.3%. The following table contains $r_{2\sigma}$ for various sample sizes M

M	$r_{2\sigma}$
9	0.83
16	0.75
25	0.70
100	0.60
10000	0.51

Even for a sample of size $M = 100$ it is not unreasonable to get 60% correct answers if they are merely predicted at random.

Generalizations are obvious:

- Representation of numbers by their binary representation, which can be used to predict time series $s(t)$ of e.g. stock values.
- Recognition of patterns of a musical tune in a training phase and playing it.