Educational Technologies
Образователни технологии

NEURAL NETWORKS

Reinhard Magenreuter
University of Finance, Business and Entrepreneurship

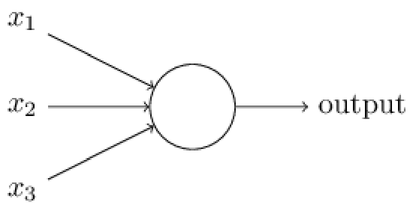
Abstract. There are four backbones to analyze time-series in general and forecast time-series for financial markets: Chaos-theory, Fuzzy logic, Neural networks and Genetic algorithms. The first one is considered in (Magenreuter, 2016), while the present paper is dedicated to the third backbone including an example with promising outcomes in financial markts.

Keywords: perceptron, sigmoid, neural network, financial market, time-series, forecasting

1. Introduction. Neural Networks, or artificial neural networks, are information processing systems, which consist of huge amount of simple units (cells, neurons), sending information to each other in form of activation of directed connections. One essential element of the artificial neural networks is its ability to learn, the ability to learn autonomously, e.g. the task of classification, out of training examples, without necessity, that the network must be programmed explicitly. Neural Networks find applications in research areas like biology, neuro-biology, neurophysiology, medicine, picture recognition, economics…etc. Everywhere, where huge amounts of data are used as basis for developing classifications and forecasts in different industrial and scientific areas, adaptive systems like neural networks are the right choice.

2. Perceptrons. Principle of maximal similarity and generalization. (Nielsen, 2015) We will start with a type of artificial neuron called a *perceptron*. Perceptrons were developed in the 1950s and 1960s by Frank Rosenblatt (1928 – 1971), inspired by earlier work by Warren McCulloch (1898 – 1969) and Walter Pitts (1923 – 1969). Today, it is more common to use other models of artificial neurons. The main neuron model used is one called the *sigmoid neuron*. We will get to sigmoid neurons shortly. But to understand why sigmoid neurons are defined the way they are, it is worthy to first explain perceptrons.

A perceptron takes several binary inputs $x_1, x_2, \ldots$ and produces a single binary output:



In the example shown the perceptron has three inputs x_1, x_2, x_3. In general it could have more or fewer inputs. Rosenblatt proposed a simple rule to compute the output. He introduced *weights*, $w_1, w_2, \ldots$, real numbers expressing the importance of the respective inputs to the output. The neuron's output, 0 or 1, is determined by whether the weighted sum $\sum_j w_j x_j$ is less than or greater than some *threshold value*. Just like the weights, the threshold is a real number which is a parameter of the neuron. To put it in more precise algebraic terms:

$$\text{output} = \begin{cases} 0 \ \text{ if} \sum_j w_j x_j \leq \text{treshold} \\ 1 \ \text{ if} \sum_j w_j x_j > \text{treshold} \end{cases}.$$

That is the basic mathematical model. A way you can think about the perceptron is that it is a device that makes decisions by weighing up evidence. There is a not very realistic example, but still it is easy to understand. Suppose the weekend is coming up, and you have heard that there is going to be a folk festival in your city. You like folk and are trying to decide whether or not to go to the festival. You might make your decision by weighing up three factors:

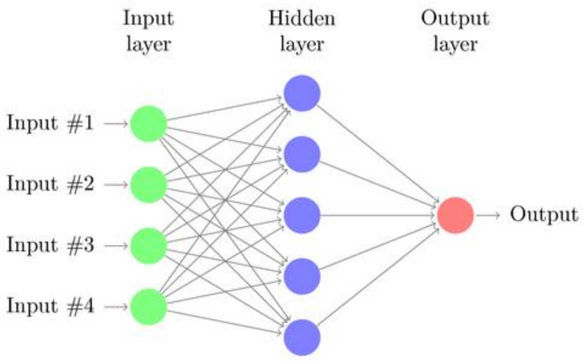
Is the weather good?
Does your friend want to accompany you?
Is the festival near public transit?

We can represent these three factors by corresponding binary variables x_1, x_2, x_3. For instance, we have $x_1 = 1$ if the weather is good, and $x_1 = 0$ if the weather is bad. Similarly, $x_2 = 1$ if your friend wants to go, and $x_2 = 0$ if not. And similarly again for x_3 and public transit.

Now, suppose you absolutely adore folk, so much so that you are happy to go to the festival even if your friend is uninterested and the festival is hard to get to. But perhaps you really load the bad weather, and there is no way you would go to the festival if the weather is bad. You can use perceptrons to model this kind of decision-making. One way to do this is to choose a weight $w_1 = 6$ for the weather, and $w_2 = 2$ and $w_3 = 2$ for the other conditions. The larger value of w_1 indicates that the weather matters a lot to you, much more than whether your friend joins you, or the nearness of public transit. Finally, suppose you choose a threshold of 5 for the perceptron. With these choices, the perceptron implements the desired decision-making model, outputting 1 whenever the weather is good, and 0 whenever the weather is bad. It makes no difference to the output whether your friend wants to go, or whether public transit is nearby.

By varying the weights and the threshold, we can get different models of decision-making. For example, suppose we instead chose a threshold of 3. Then the perceptron would decide that you should go to the festival whenever the weather was good *or* when both the festival was near public transit *and* your friend was willing to join you. In other words, it would be a different model of decision-making. Dropping the threshold means you are more willing to go to the festival.

Obviously, the perceptron is not a complete model of human decision-making. But what the example illustrates is how a perceptron can weigh up different kinds of evidence in order to make decisions. And it should seem plausible that a complex network of perceptrons could make quite subtle decisions. Like we humans, a neural network is able to generalize. Neural networks must not be programmed. It is enough to present even huge amounts of data as input to the system. The neural network then transforms this data within a so called input-layer through hidden layers to so called output layers. Within the hidden layers the so called neurons compare the input values with the desired values and adjusts during each iteration the weighs so, that a given error will be minimized. This is a simple scheme, how a back-propagation network processes in general:



In this network, the first column of perceptrons – what we shall call the first *layer* of perceptrons – is making three very simple decisions, by weighing the input evidence. What about the perceptrons in the second layer? Each of those perceptrons is making a decision by weighing up the results from the first layer of decision-making. In this way a perceptron in the second layer can make a decision at a more complex and more abstract level than perceptrons in the first layer. And even more complex decisions can be made by the perceptron in the third layer. In this way, a many-layer network of perceptrons can engage in sophisticated decision making.

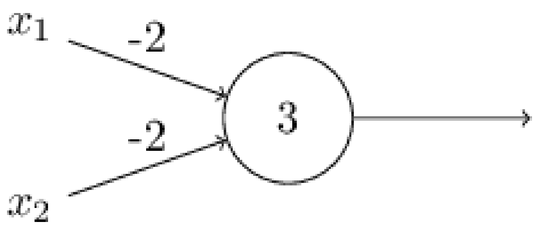
Incidentally, when we defined perceptrons we said that a perceptron has just a single output. In the network above the perceptrons look like they have multiple outputs. In fact, they are still single output. The multiple output arrows are merely a useful way of indicating that the output from a perceptron is being used as the input to several other perceptrons. It is less unwieldy than drawing a single output line which then splits.

Let us simplify the way we describe perceptrons. The condition $\sum_j w_j x_j > \text{treshold}$ is cumbersome, and we can make two notational changes to simplify it. The first change is to write $\sum_j w_j x_j$ as a dot product, i.e. $w \cdot x = \sum_j w_j x_j$, where w and x are vectors whose components are the weights and inputs, respectively. The second change is to move the threshold to the other side of the inequality, and to replace it by what is known as the perceptron's *bias*, $b \equiv -\text{threshold}$. Using the bias instead of the threshold, the perceptron rule can be rewritten:

$$\text{output} = \begin{cases} 0 & \text{if} \quad w \cdot x + b \leq 0 \\ 1 & \text{if} \quad w \cdot x + b > 0 \end{cases} .$$

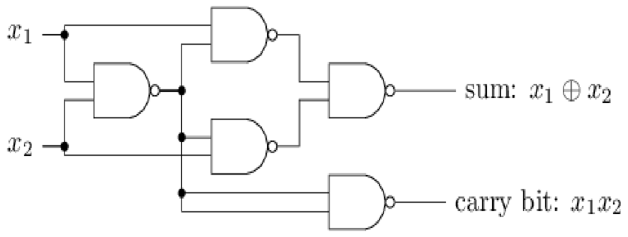
One can think of the bias as a measure of how easy it is to get the perceptron to output a 1. Or to put it in more biological terms, the bias is a measure of how easy it is to get the perceptron to *fire*. For a perceptron with a really big bias, it is extremely easy to output a 1. But if the bias is *very* negative, then it is difficult for the perceptron to output a 1. Obviously, introducing the bias is only a small change in how we describe perceptrons, but it could lead to further notational simplifications.

We have described perceptrons as a method for weighing evidence to make decisions. Another way perceptrons can be used is to compute the elementary logical functions we usually think of as underlying computation, functions such as `AND`, `OR` and `NAND`. For example, suppose we have a perceptron with two inputs, each with weight −2, and an overall bias of 3. Here is the perceptron:

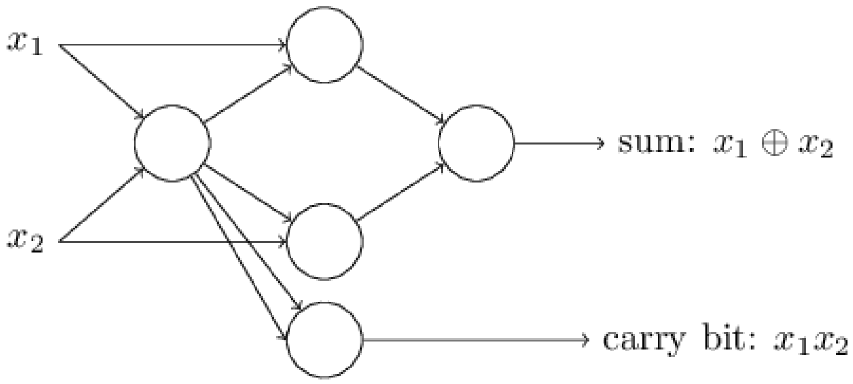


Then we see that input 00 produces output 1, since (−2)∗0 + (−2)∗0 + 3 = 3 is positive. Here, we have introduced the ∗symbol to make the multiplications explicit. Similar calculations show that the inputs 01 and 10 produce output 1. But the input 11 produces output 0, since (−2)∗1 + (−2)∗1 + 3 = −1 is negative. And so our perceptron implements a `NAND` gate!

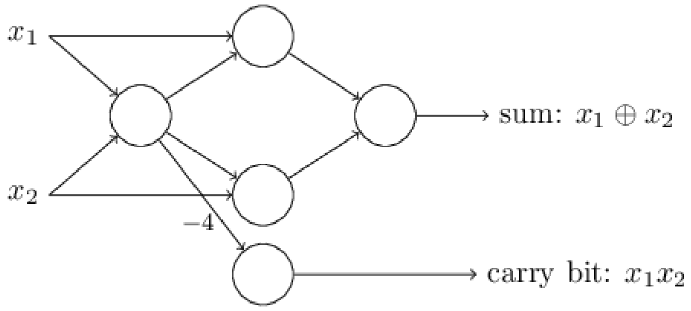
The `NAND` example shows that we can use perceptrons to compute simple logical functions. In fact, we can use networks of perceptrons to compute *any* logical function at all. The reason is that the `NAND` gate is universal for computation, that is, we can build any computation up out of `NAND` gates. For example, we can use `NAND` gates to build a circuit which adds two bits, x_1 and x_2. This requires computing the bitwise sum $x_1 \oplus x_2$, as well as a carry bit which is set to 1 when both x_1 and x_2 are 1, i.e., the carry bit is just the bitwise product x_1x_2:



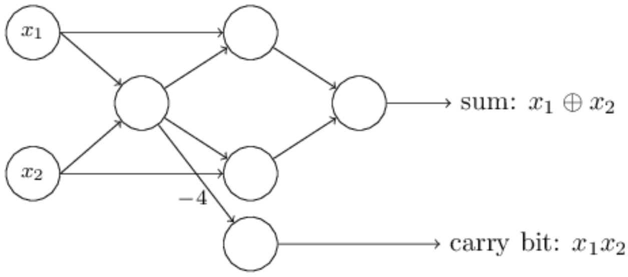
To get an equivalent network of perceptrons we replace all the `NAND` gates by perceptrons with two inputs, each with weight −2 and an overall bias of 3. Here is the resulting network. Note that we have moved the perceptron corresponding to the bottom right NAND gate a little, just to make it easier to draw the arrows on the diagram:



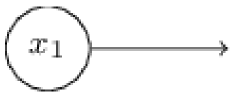
One notable aspect of this network of perceptrons is that the output from the leftmost perceptron is used twice as input to the bottom most perceptron. When we defined the perceptron model we did not say whether this kind of double-output-to-the-same-place was allowed. Actually, it does not much matter. If we do not want to allow this kind of thing, then it is possible to simply merge the two lines, into a single connection with a weight of –4 instead of two connections with –2 weights. With that change, the network looks as follows, with all unmarked weights equal to –2, all biases equal to 3, and a single weight of –4, as marked:



Up to now we have been drawing inputs like x_1 and x_2 as variables floating to the left of the network of perceptrons. In fact, it is conventional to draw an extra layer of perceptrons – the *input layer* – to encode the inputs:



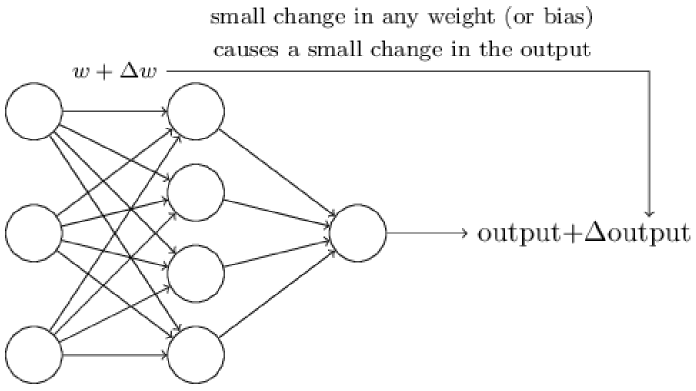
This notation for input perceptrons, in which we have an output, but no inputs,



is a shorthand. It does not actually mean a perceptron with no inputs. To see this, suppose we did have a perceptron with no inputs. Then the weighted sum $\sum_j w_j x_j$ would always be zero, and so the perceptron would output 1 if $b > 0$ and 0 if $b \le 0$. That is, the perceptron would simply output a fixed value, not the desired value (x_1 , in the example above). It is better to think of the input perceptrons as not really being perceptrons at all, but rather special units which are simply defined to output the desired values x_1 , x_2 , …

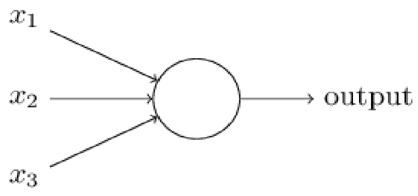
The adder example demonstrates how a network of perceptrons can be used to simulate a circuit containing many `NAND` gates. And because `NAND` gates are universal for computation, it follows that perceptrons are also universal for computation. The computational universality of perceptrons is simultaneously reassuring and disappointing. It is reassuring because it tells us that networks of perceptrons can be as powerful as any other computing device. But it is also disappointing, because it makes it seem as though perceptrons are merely a new type of `NAND` gate. However, the situation is better than this view suggests. It turns out that we can devise *learning algorithms* which can automatically tune the weights and biases of a network of artificial neurons. This tuning happens in response to external stimuli, without direct intervention by a programmer. These learning algorithms enable us to use artificial neurons in a way which is radically different to conventional logic gates. Instead of explicitly laying out a circuit of `NAND` and other gates, our neural networks can simply learn to solve problems, sometimes problems where it would be extremely difficult to directly design a conventional circuit. The detailed will be considered in another publication.

3. Sigmoid neurons. (Nielsen, 2015) Learning algorithms sound terrific. But how can we devise such algorithms for a neural network? Suppose we have a network of perceptrons that we would like to use to learn to solve some problem. For example, the inputs to the network might be the raw pixel data from a scanned, handwritten image of a digit. And we would like the network to learn weights and biases so that the output from the network correctly classifies the digit. To see how learning might work, suppose we make a small change in some weight (or bias) in the network. What we would like is for this small change in weight to cause only a small corresponding change in the output from the network. This property, known as *stabilty property*, will make learning possible. Schematically, here is what we want (obviously this network is too simple to do handwriting recognition):


small change in any weight (or bias)
causes a small change in the output
$w + \Delta w$
output+Δoutput

If it were true that a small change in a weight (or bias) causes only a small change in output, then we could use this fact to modify the weights and biases to get our network to behave more in the manner we want. For example, suppose the network was mistakenly classifying an image as an "8" when it should be a "9". We could figure out how to make a small change in the weights and biases so the network gets a little closer to classifying the image as a "9". And then we will repeat this, changing the weights and biases over and over to produce better and better output. The network would be learning.

The problem is that this is not what happens when our network contains perceptrons. In fact, a small change in the weights or bias of any single perceptron in the network can sometimes cause the output of that perceptron to completely flip, say from 0 to 1. That flip may then cause the behaviour of the rest of the network to completely change in some very complicated way. So while the "9" might now be classified correctly, the behaviour of the network on all the other images is likely to have completely changed in some hard-to-control way. That makes it difficult to see how to gradually modify the weights and biases so that the network gets closer to the desired behaviour. Perhaps there is some clever way of getting around this problem. But it is not immediately obvious how we can get a network of perceptrons to learn.We can overcome this problem by introducing a new type of artificial neuron called a *sigmoid* neuron. Sigmoid neurons are similar to perceptrons, but modified so that small changes in their weights and bias cause only a small change in their output. That is the crucial fact which will allow a network of sigmoid neurons to learn. We shall depict sigmoid neurons in the same way we depicted perceptrons:



Just like a perceptron, the sigmoid neuron has inputs, x_1, x_2, ... But instead of being just 0 or 1, these inputs can also take on any values *between* 0 and 1. So, for instance 0,638 is a valid input for a sigmoid neuron. Also just like a perceptron, the sigmoid neuron has weights for each input, w_1, w_2, ... and an overall bias b. But the output is not 0 or 1. Instead, it is $\sigma(w{\cdot}x+b)$, where σ is called the *sigmoid function* (Incidentally, σ is sometimes called the *logistic function*, and this new class of neurons is called *logistic neurons*. However, we shall stick with the sigmoid terminology.), and is defined by:

$$\sigma \equiv \frac{1}{1+e^{-z}}.$$

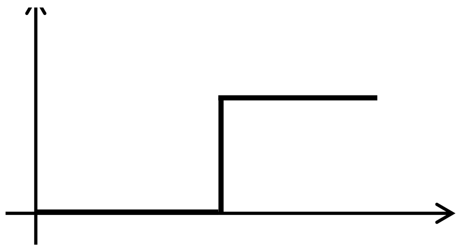
To put it all a little more explicitly, the output of a sigmoid neuron with inputs $x_1, x_2, \ldots$, weights $w_1, w_2, \ldots$ and bias b is

$$\frac{1}{1 + \exp\left(-\sum_j w_j x_j - b\right)}.$$

At first sight, sigmoid neurons appear very different to perceptrons. The algebraic form of the sigmoid function may seem opaque and forbidding. In fact, there are many similarities between perceptrons and sigmoid neurons, and the algebraic form of the sigmoid function turns out to be more of a technical detail than a true barrier to understanding.

To understand the similarity to the perceptron model, suppose $z \equiv w \cdot x + b$ is a large positive number. Then $e^{-z} \approx 0$ and so $\sigma(z) \approx 1$. In other words, when $z \equiv w \cdot x + b$ is large and positive, the output from the sigmoid neuron is approximately 1, just as it would have been for a perceptron. Suppose on the other hand that $z \equiv w \cdot x + b$ is very negative. Then $e^{-z} \to \infty$ and $\sigma(z) \approx 0$. So when $z \equiv w \cdot x + b$ is very negative, the behaviour of a sigmoid neuron also closely approximates a perceptron. It is only when $w \cdot x + b$ is of modest size that there is much deviation from the perceptron model.

The exact form of σ is not so important. What really matters is the shape of the function when plotted. The shape in fact is a smoothed out version of a step function:



If σ had in fact been a step function, then the sigmoid neuron would *be* a perceptron, since the output would be 1 or 0 depending on whether $w \cdot x + b$ was positive or negative. (Actually, when $w \cdot x + b = 0$, the perceptron outputs 0, while the step function outputs 1. So, strictly speaking, we would need to modify the step function at that one point.) By using the actual σ function we get, as already implied above, a smoothed out perceptron. Indeed, it is the smoothness of the σ function that is the crucial fact, not its detailed form. The smoothness of σ means that small changes Δw_j in the weights and Δb in the bias will produce a small change Δ-outputt in the output from the neuron. In fact, calculus tells that Δ-output is well approximated by

$$\Delta \text{ output} \approx \sum_j \frac{\partial \text{ output}}{\partial w_j} \Delta w_j + \frac{\partial \text{ output}}{\partial b} \Delta b ,$$

where the sum is over all the weights w_j and $\frac{\partial \text{ output}}{\partial w_j}$ and $\frac{\partial \text{ output}}{\partial b}$ denote partial derivatives of the outputoutput with respect to w_j and b, respectively. While the expression above looks complicated, with all the partial derivatives, it is actually saying something very simple: Δ-output is a *linear function* of the changes Δw_j and Δb in the weights and bias. This linearity makes it easy to choose small changes in the weights and biases to achieve any desired small change in the output. So while sigmoid neurons have much of the same qualitative behaviour as perceptrons, they make it much easier to figure out how changing the weights and biases will change the output.

Obviously, one big difference between perceptrons and sigmoid neurons is that sigmoid neurons do not just output 0 or 1. They can have as output any real number between 0 and 1, so values such as 0,173 and 0,689 are legitimate outputs. This can be useful, for example, if we want to use the output value to represent the average intensity of the pixels in an image input to a neural network. But sometimes it can be a nuisance. Suppose we want the output from the network to indicate either “the input image is a 9” or “the input image is not a 9”. Obviously, it woul be easiest to do this if the output was a 0 or a 1, as in a perceptron. But in practice we can set up a convention to deal with this, for example, by deciding to interpret any output of at least 0,5 as indicating a “9”, and any output less than 0,5 as indicating “not a 9”.

4. Mathematical background, retail case study. As mentioned above, a neural network in its basic architectural model, consists of three layers: input, hidden and output. If we remove the hidden layer(s), the neural network becomes a simple regression (for estimation) or logistic regression (for classification) architectural model. The input layer for e.g. a retail case study ‘offers’ some input variables like:

+ Recency: # of recent visits to the company’s website and purchases;
+ Frequency: time lag between purchases in the last 6 months;
+ Payment mode used: cash on delivery, credit card, internet banking etc.;
+ Marketing data aggregator’s: life-stage segmentations (i.e. luxury buffs, up-scale ageing, first-time earners etc.);
+ Last year’s expenditure trend: amount spent last year;
+ Coupon usage pattern of customer.

The output layer, for the classification problem to identify customers who will respond to campaigns, is 0 for non-responders and 1 for responders. The following expression represents the weighted sum of input variables that the hidden nodes take as input:

$$(\text{Hidden Node})_i = W^T_{i(\text{Input}\rightarrow\text{Hidden})} \times (\text{Input Variables})_i + W_0$$

"To begin with, the above weights W_i (Input→Hidden) & W_0 are chosen at random, then they are modified iteratively to match the desired outputs (in output layer)"…" In the hidden layer, the above linear weighted sum [(Hidden Node)$_i$] is converter to non-linear form through a non-linear function. This conversion is usually performed using the sigmoid activation function, yes this is the same logit function of the logistic regression. The following expression represents this process:

$$P(\text{Hidden})_j = \frac{e^{(\text{Hidden Node})_i}}{1 + e^{(\text{Hidden Node})_i}},$$

where $0 \leq P(\text{Hidden})_j \leq 1$; these output [$P(\text{Hidden})_j$] for the different hidden nodes (j) becomes the input variables for the final output node. As described below:

$$(\text{Output}) = U^T_{i(Hidden\rightarrow Output)} \times P(\text{Hidden})_j + U_0$$

This linear weighted output is again converted to non-linear form through sigmoid function. The following is the probability of conversion of a customer P (Customer Response) based on his/her input variables:

$$P(\text{Customer Response}) = \frac{e^{(\text{Output})}}{1 + e^{(\text{Output})}}.$$

Neural network algorithms (like back propagation) iteratively modify weights for both links (i.e. Input→Hidden→Output) to reduce the error of prediction. Remember the weights for our architect are weights $W_{i(\text{Input}\rightarrow\text{Hidden}),}$ W_0 weights $U_{j(\text{Hidden}\rightarrow\text{Output})}$ & U_0."[1]

5. Popular Example. Example for generalization: if we meet a person, who is, let us say 30 meters away, our brain generalizes with direct access, if this person could be e.g. a former classmate we met last time 20 years ago. It is not necessary to access our whole data base of persons we ever met in our life from *a* to *z*. The same, artificial neural networks are able to achieve, because they work with the principle of 'maximal similarity'. Let us say a neural network shall 'decide' which animal is presented to a camera as input. Some animals have been trained before. The learned attributes were: long tail, he has four legs, he has wings, he can fly and he eats mice. That is a cat, because a cat has three, the maximum features: he has a long tail, he eats mice and he has four legs. A fish, a bird and a penguin have less features, a fish 0, a bird 2 and a penguin 1.

Since the early 1980s several experiments to use neural networks for valuations and forecast tasks for the financial markets have been executed. Since that time the authors explores them and adapt them for forecast challenges with growing success over the years. Following, we will expose an interesting task of neural networks for detecting if a bank will go bankrupt or not in the near future.

To forecast, if a bank will go bankrupt or not during the next year or the next two years is quite good solvable with neural networks. The data material had been divided into two groups of banks, one with data one year forerun and one with two years forerun, referred of the time they went bankrupt. A dissociation between 'bankrupt' and 'not bankrupt' took place by four control measures: a) asset size; b) number of branches; c) age and d) charter status. There could be selected for both forerun times each 118 banks (59 went bankrupt, 59 did not went bankrupt), out of the basis data set. The status 'bankrupt' or 'not bankrupt' will be described by the following 19 financial operating numbers:

1. Capital/assets
2. Agricultural production & farm loans + real estate loans secured by farm land/ net loans and leases
3. Commercial and industrial loans/met loans and leases
4. Loans to individuals/net loans and leases
5. Real estate loans/net loans and leases
6. Total loans 90 days or more past due/net loans and leases
7. Total non-accrual loans & leases/net loans and leases
8. Provision for loan losses/average loans
9. Net charge-offs/average loans
10. Return on average assets
11. Total interest paid on deposits/total deposits
12. Total expense/total assets
13. Net income/total assets
14. Interests and fees on loans + income from lease financing rec/net loans & leases
15. Total income/total expense
16. Cash + U.S. treasury & government agency obligations/total assets
17. Federal funds sold + securities/total assets
18. Total loans & leases/total assets
19. Total loans & leases/total deposits

(Tam & Kiang, 1990)

This selection criterions has been Capital, Asset, Management, Equity and Liquidity, which can be used to predict a potential bankruptcy. The detailed analysis model of Mr. Tam can be read in (Tam & Kiang, 1990), see above. This 19 input neurons have been associated with two output-neurons: > 0,5 non-failed banks, < 0,5 failed banks.

Compared to the classical discriminant analysis the results of the neural networks surpassed them by far. The best error classification was 14,8 % for the one year forerun, respectively 85,2 % were correct.

As for analyzing and predicting financial markets, we need a couple of meaningful *fundamental* data (not derived mathematically), which can be taken as indicators for our target time-series. Such data can be: Interest indicators, interest sensitive indicators, prime rates, money supply data, gold prize, inflation data, economic cycle indicators, balance of payments, fundamental stock data like price/earnings rate, dividend payments, sentiment indicators etc.

Neural Networks are applied in different economic areas like: business finance, financial management, public finance, quantitative finance, rating modelling for e.g. bonds, stocks and properties and portfolio management, where time series forecasting is the highest challenge, because the factor 'time' is added to a relative simple 'two-dimensional' classification task.

Neural Networks differ a lot in their network-topology and adjustable parameters like e.g. learning rate and number of neurons. To deal with this challenge, we need to implement a genetic algorithm, which examines a huge space of ranges of such parameters. Such a powerful optimization algorithm will be presented in another publication.

"*We like so much to read into the future, because we want to lead to us*
through calm wishes, the mystical, which sways in it."
(J.W. Goethe)

NOTES

1. Roopam Upadhyay, International Finance Corporation, YOU CANalytics
2. J.W. Goethe, Maximen und Reflexionen V

REFFERENCES

Magenreuter, R. (2016). Forecasting of time-series for fianncial markets. *Mathematics and Informatics*, 59 (5).

Tam, K. & Kiang, M. (1990). Predicting Bank Failures: A neural Network Approach. *Applied Artificial Intelligence*, 4, pp. 265 – 282

Grozdev, S. (2007). *For high achievements in mathematics. The Bulgarian experience* (*Theory and practice*). Sofia: ADE.

Nielsen, M. (2015). *Neural networks and deep learning*. Determination Press. (This work is licensed under a Creative Commons Attribution-NonCommercial 3.0 Unported License. This means you're free to copy, share, and build on this book, but not to sell it.)

✉ **Mr. Reinhard Magenreuter, PhD student**
University of Finance, Business and Entrepreneurship
1, Gusla St.
1618 Sofia, Bulgaria
E-mail: rm@pariserplatz.eu