Volatility Smile Extrapolation with an Artificial Neural Network

Mark Michael Richter
In partial fulfillment of the requirements for the
Technical Degree in Computer Science (Systems) of
THE OPEN UNIVERSITY OF CATALUNYA
Barcelona, Spain

December 27, 2012

Abstract

I use a multi-layer feedforward perceptron, with backpropagation learning implemented via stochastic gradient descent, to extrapolate the volatility smile of Euribor derivatives over low-strikes by training the network on parametric prices. The percetron must adapt itself to the smile implied by the SABR model and Kainth's parametric extension [6], and extrapolate over unobservable prices in the low-strike region while maintaining certain technical conditions specific to probability density functions. This is useful for computing, for example, price sensitivity to volatilities at strikes close to zero or even negative. The efficient implementation and use of the model requires a relatively large library of classes and functions to read market data, calibrate and price the derivatives. This was done in Matlab via an object-orientated paradigm. We also use a radial basis network to calibrate to the SABR implied volatility smile and directly extrapolate over low-strikes. In both cases the results were not as good as we would have expected. We propose in the conclusions that perhaps more complex networks might adjust to prices more correctly.

1 Introduction

The term *neural network* was traditionally used to refer to a network of biological neurons. Biological neural networks are made up of biological neurons that are connected or functionally related in a nervous system. In the field of neuroscience, they are often identified as groups of neurons that perform a specific physiological function in laboratory analysis. Academics, drawing inspiration on the functionality of this natural phenomenon, led the initiative to replicate this web of neurons. Such efforts have led to the development of artificial neural networks which are composed of interconnecting artificial neurons.
The artificial neuron receives one or more inputs[1] and sums them to produce an output[2]. Usually the sums of each node are weighted, and the sum is passed through a non-linear function known as an activation function. These activation functions usually have a sigmoid shape, but they may also take the form of other non-linear functions, piecewise linear functions or step functions[3].
Artificial neural networks may either be used to gain an understanding of biological neural networks, or for solving artificial intelligence problems without necessarily creating a model of a real biological system. The biological nervous system is highly complex, artificial neural network algorithms attempt to abstract this complexity and focus on what may hypothetically matter most from an information processing point of view. Good performance as measured by good predictive ability and low generalization error or performance mimicking animal or human error patterns can then be used as one source of evidence towards supporting the hypothesis that the abstraction really captured something important from the point of view of information processing in the brain. Another incentive for these abstractions is to reduce the amount of computation required to simulate artificial neural networks, so as to allow one to experiment with larger networks and train them on larger data sets.
In this project we will use a neural network as an interpolation and extrapolation mechanism of a highly complex function. The multilayer perceptron is also known as the universal interpolator and for that reason we believe it will be of particularly good use to solve a financial problem we will detail in greater depth further on in this project.
We will avoid giving a general definition of a neural network at this point. So many models have been proposed in the literature which dither in so many respects that any definition trying to encompass this variety would be unnecessarily clumsy. We will not begin by building neural networks with *high powered* computing units but rather start our investigations with the general notion that a neural network is a network of functions in which synchronization can be considered explicitly, or not, along with a series of other general properties.

[1]Representing the one or more biological dendrites.
[2]Representing a biological neuron's axon.
[3]Generally they are monotonically increasing, continuous, differentiable and bounded.

2 Taxonomy of Artificial Neural Networks

A neural network is a mapping from inputs to outputs. The mapping may be broken down into three distinct elements: the interconnected pattern between different layers of neurons, the learning process and the style of activation function. The mapping from inputs to outputs may be viewed as a functional or a probabilistic model both of which are largely equivalent. The most common types of network are feedforward, which graphically are directed acyclic graphs and networks with cycles which are known as recurrent. More complex typologies are radial basis function networks, learning vector quantization networks, modular networks and fuzzy networks. We will only focus on the most common and simple typology, the feedforward architecture. One of the most important aspects of neural network design consists of determining a cost function and a learning paradigm, which is usually determined by the nature of the objective of the network and may be classified in: supervised, unsupervised and reinforcement paradigms. The specific choice of algorithm within the chosen paradigm must be determined by the data available, the type of network and our final objectives. In this project we will focus on the backpropagation algorithm within a supervised learning paradigm.
A reasonable approach to classifying neural networks is by studying the individual computational units, the neurons, which compose the network. The simplest kind of computing units used to build artificial neural networks are a generalization of the common logic gates used in conventional computing. An important characteristic of the neuron is that they generally activate by comparing their output value with a threshold value. From the introduction where we looked at the characteristics and structure of biological neural networks which provided us with the initial motivation for a deeper inquiry into the properties of networks of abstract neurons. From the viewpoint of the engineer, it is important to define how a network should behave, without having to specify completely all of its parameters, which are to be found in a learning process. Artificial neural networks are used in many cases as a black box, a certain input should produce a desired output, but how the network achieves this result is left to a self-organizing process.
A neural network behaves as a *mapping machine*, capable of modeling a function $F : \mathbb{R}^n \to \mathbb{R}^m$. If we look at the structure of the network being used, some aspects of its dynamics must be defined more precisely. When the function is evaluated with a network of primitive functions, information flows through the directed edges of the network. Some nodes compute values which are then transmitted as arguments for new computations. If there are no cycles in the network, the result of the whole computation is well-defined and we do not have to deal with the task of synchronizing the computing units. We just assume that the computations take place without delay. If the network contains cycles, however, the computation is not uniquely defined by the interconnection pattern and the temporal dimension must be considered. When the output of a unit is fed back to the same unit, we are dealing with a recursive computation without an explicit halting condition. We must define what we expect from the network: is the fixed point of the recursive evaluation the desired result or one of the intermediate computations? To solve this problem we assume that every computation takes a certain amount of time at each node[4]. If the arguments for a unit have been transmitted at time t, its output will be produced at time $t + 1$. A recursive computation can be stopped after a certain number of steps and the last computed output taken as the result of the recursive computation.

[4]For example a time unit.

We will deal in this project solely with acyclic networks for simplicity. The first model of neuron we consider was proposed in 1943 by Warren McCulloch and Walter Pitts [9].
The nodes of the networks we consider will be called computing elements or simply units. We assume that the edges of the network transmit information in a predetermined direction and the number of incoming edges into a node is not restricted by some upper bound. This is called the unlimited fan-in property of our computing units. The primitive function computed at each node is in general a function of n arguments. Normally, however, we try to use very simple primitive functions of one argument at the nodes. This means that the incoming n arguments have to be reduced to a single numerical value. Therefore computing units are split into two functional parts: an integration function g reduces the n arguments to a single value and the output or activation function f produces the output of this node taking that single value as its argument. Usually the integration function g is the addition function. McCulloch Pitts networks are even simpler than this, because they use solely binary signals. The nodes produce only binary results and the edges transmit exclusively ones or zeros. The networks are composed of directed unweighted edges of excitatory or of inhibitory type[5]. The latter are marked in diagrams using a small circle attached to the end of the edge. Each McCulloch Pitts unit is also provided with a certain threshold value[6]. At first sight the McCulloch Pitts model seems very limited, since only binary information can be produced and transmitted, but it already contains all necessary features to implement the more complex models.
An algorithmic description of the McCulloch Pitts unit can be itemized as follows:

- Assume that a McCulloch Pitts unit gets an input $x_1, x_2, \ldots, x_n$ through n excitatory edges and an input $y_1, y_2, \ldots, y_m$ through m inhibitory edges.

- If $m = 1$ and at least one of the signals $y_1, y_2, \ldots, y_m$ is 1, the unit is inhibited and the result of the computation is 0.

- Otherwise the total excitation $x = x_1 + x_2 + \cdots + x_n$ is computed and compared with the threshold of the unit (if $n = 0$ then $x = 0$). If x equal the threshold value the unit fires a 1, if x is below the threshold the result of the computation is 0.

This rule implies that a McCulloch Pitts unit can be inactivated by a single inhibitory signal, as is the case with some real neurons. When no inhibitory signals are present, the units act as a threshold gate capable of implementing many other logical functions of n arguments. The power of threshold gates of the McCulloch Pitts type can be illustrated by showing how to synthesize any given logical function of n arguments, that is $f : [0,1]^n \to [0,1]$. We may ask, to what point is inhibition necessary in McCulloch Pitts units? This brings us to the follow proposition[7].

Proposition 1 *Uninhibited threshold logic elements of the McCulloch Pitts type can only implement monotonic[8] logical functions.*

[5]An inhibitory edge implies that should it be activated the neuron will not activate, irrespective of the value from the other inputs.

[6]The function must obtain a value higher then the threshold for the neuron to activate.

[7]We choose to simply announce the propositions and not offer proof so as to focus more on the practical aspect of the material.

[8]A monotonic logical function f of n arguments is one whose value at two given n-dimensional points $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$ is such that $f(x) = f(y)$ whenever the number of ones in the input y is a subset of the ones in the input x.

An interesting addition to this proposition is the following.

Proposition 2 *Any logical function $F : [0,1]^n \to [0,1]$ can be computed with a McCulloch Pitts network of two layers.*

We can build simpler circuits by using units with more general properties, for example weighted edges and relative inhibition. However, circuits of McCulloch Pitts units can emulate circuits built out of high-powered units by exploiting the trade-off between the complexity of the network versus the complexity of the computing units. For example, weighted and unweighted networks are equivalent. Another, more interesting example, is that two classes of inhibition can be identified: absolute inhibition corresponds to the one used in McCulloch Pitts units. Relative inhibition corresponds to the case of edges weighted with a negative factor and whose effect is to lower the firing threshold when a 1 is transmitted through this edge.

Proposition 3 *Networks of McCulloch Pitts units are equivalent to networks with relative inhibition.*

A final proposition which leads from the previous ones indicates the power of these networks:

Proposition 4 *McCulloch Pitts units can be used to build networks capable of computing any logical function and of simulating any finite automaton.*

Before concluding we should consider the differences between binary signals and pulse coding. An additional question which can be raised is whether binary signals are not a very limited coding strategy. Are networks in which the communication channels adopt any of ten or fifteen different states more efficient than channels which adopt only two states, as in McCulloch Pitts networks? To give an answer we must consider that unit states have a price, in biological networks as well as in artificial ones. The transmitted information must be optimized using the number of available switching states. The binary nature of information transmission in the nervous system seems to be an efficient way to transport signals. However, we may also assume that the communication channels can transport arbitrary real numbers. This makes the analysis simpler than when we have to deal explicitly with frequency modulated signals, but does not lead to a minimization of the resources needed for a technical implementation. Some researchers prefer to work with so-called weightless networks which operate exclusively with binary data.
To conclude this chapter, I detail the final taxonomy of different neural networks which can be understood, albeit superficially, based on the ideas and concepts put forth in this chapter. The first clear separation line runs between weighted and unweighted networks. It has already been shown that both classes of models are equivalent. The main difference is the kind of learning algorithm that can be used. In unweighted networks only the thresholds and the connectivity can be adapted. In weighted networks the topology is not usually modified during learning, although there exist some algorithms capable of doing this, and only an optimal combination of weights is sought. The second clear separation is between synchronous and asynchronous models. In synchronous models the output of all elements is computed instantaneously. This is always possible if the topology of the network does not contain cycles. In some cases the models contain layers of computing units and the activity of the units in each layer is computed one after the other, but in each layer simultaneously. Asynchronous models compute the activity of each unit independently of all others and at different stochastically selected times, as in Hopfield networks. In these kinds of models,

cycles in the underlying connection graph pose no particular problem. Finally, we can distinguish between models with or without stored unit states. If the number of states and possible inputs is finite, we are dealing with a finite automaton. Since any finite automaton can be simulated by a network of computing elements without memory, these units with a stored state can be substituted by a network of McCulloch Pitts units. Networks with storedstate units are thus equivalent to networks without stored-state units. Data is stored in the network itself and in its pattern of recursion. It can be also shown that time varying weights and thresholds can be implemented in a network of McCulloch Pitts units using cycles, so that networks with time varying weights and thresholds are equivalent to networks with constant parameters, whenever recursion is allowed.

3 Perceptrons

The types of network that can be built out of McCulloch Pitts neurons are not very relevant. The computing units are too similar to conventional logic gates and the network must be completely specified before it can be used. There are no free parameters which could be adjusted to suit different problems. Learning can only be implemented by modifying the connection pattern of the network and the thresholds of the units, but this is necessarily more complex than just adjusting numerical parameters. For that reason, we turn our attention to weighted networks and consider their most relevant properties. In the last section of this chapter we show that simple weighted networks can provide a computational model for regular neuronal structures in the nervous system. The essential innovation was the introduction of numerical weights and a special interconnection pattern. In the original Rosenblatt model [11] the computing units are threshold elements and the connectivity is determined stochastically. Learning takes place by adapting the weights of the network with a numerical algorithm. Rosenblatts stochastic model was refined and perfected in the 1960s and its computational properties were carefully analyzed by Minsky and Papert [10]. Minsky and Papert distilled the essential features from Rosenblatts model in order to study the computational capabilities of the perceptron under different assumptions.

Definition 1 *A simple perceptron is a computing unit with threshold θ which, when receiving the n real inputs $x_1, x_2, \ldots, x_n$ through edges with the associated weights $w_1, w_2, \ldots, w_n$, outputs* 1 *if the inequality $\sum_{i=1}^{n} w_i x_i \geq \theta$ holds and otherwise* 0*.*

A perceptron network is capable of computing any logical function, since perceptrons are even more powerful than unweighted McCulloch Pitts elements. If we reduce the network to a single element, which functions are still computable? For example, with two binary inputs the XOR cannot be computed. More generally, certain n parameter logical functions may also not be computed.
This fact has to do with the geometry of the n- dimensional hypercube whose vertices represent the combination of logic values of the arguments. Each logical function separates the vertices into two classes. If the points whose function value is 1 cannot be separated with a linear cut from the points whose function value is 0, the function is not perceptron-computable. Basically a perceptron can only calculate linearly separable functions:

Definition 2 *Two sets of points A and B in an n-dimensional space are called linearly separable if $n+1$ real numbers $w_1, \ldots, w_{n+1}$ exist, such that every point $(x_1, x_2, \ldots, x_n) \in A$ satisfies $\sum_{i=1}^{n} w_i x_i \geq w_{n+1}$ and every point $(x_1, x_2, \ldots, x_n) \in B$ satisfies $\sum_{i=1}^{n} w_i x_i < w_{n+1}$.*

The computation performed by a perceptron can be visualized as a linear separation of input space. However, when trying to find the appropriate weights for a perceptron, the search process can be better visualized in weight space. When m real weights must be determined, the search space is the whole of $\mathbb{R}^m$. For a perceptron with n input lines, finding the appropriate linear separation amounts to finding $n+1$ free parameters, n weights and the bias. These $n+1$ parameters represent a point in $(n+1)$-dimensional weight space. Each time we pick one point in weight space we are choosing one combination of weights and a specific linear separation of input space. This means that every point in $(n+1)$-dimensional weight space can be associated with a hyperplane in $(n+1)$-dimensional extended input space. Each combination of three weights, w_1, w_2, w_3, which represent a point in weight space, defines a separation of input space with the plane $w_1x_1 + w_2x_2 + w_3x_3 = 0$.

There is the same kind of relation in the inverse direction, from input to weight space. If we want the point x_1, x_2, x_3 to be located in the positive half-space defined by a plane, we need to determine the appropriate weights $w_1, w_2 and w_3$. The inequality $w_1x_1 + w_2x_2 + w_3x_3 \geq 0$ must hold. However this inequality defines a linear separation of weight space, that is, the point (x_1, x_2, x_3) defines a cutting plane in weight space. Points in one space are mapped to planes in the other and vice versa. This complementary relation is called duality. Input and weight space are dual spaces and we can visualize the computations done by perceptrons and learning algorithms in any one of them. Given two sets of patterns which must be separated by a perceptron, a learning algorithm should automatically find the weights and threshold necessary for the solution of the problem. The *perceptron learning algorithm* can accomplish this for threshold units. This idea was proposed by Ronsenblatt [11].

3.1 Perceptron Learning

In the preceding sections we discussed two closely related models, McCulloch Pitts units and perceptrons, but the question of how to find the parameters adequate for a given task was left open.
The perceptron learning algorithm deals with this problem. A learning algorithm is an adaptive method by which a network of computing units self-organizes to implement the desired behavior. This is done in some learning algorithms by presenting some examples of the desired input-output mapping to the network. A correction step is executed iteratively until the network learns to produce the desired response. The learning algorithm is a closed loop of presentation of examples and of corrections to the network parameters.
In some simple cases the weights for the computing units can be found through a sequential test of stochastically generated numerical combinations. However, such algorithms which look blindly for a solution do not qualify as *learning*. A learning algorithm must adapt the network parameters according to previous experience until a solution is found, if it exists.
Learning algorithms can be divided into supervised and unsupervised methods. Supervised learning denotes a method in which some input vectors are collected and presented to the network. The output computed by the network is observed and the deviation from the expected answer is measured. The weights are corrected according to the magnitude of the error in the way defined by the learning algorithm. This kind of learning is also called learning with a teacher, since a control process knows the correct answer for the set of selected input vectors. Unsupervised learning is used when, for a given input, the exact numerical output a network should produce is unknown.
Supervised learning is further divided into methods which use reinforcement or error correction. Reinforcement learning is used when after each presentation of an input-output example we only know whether the network produces the desired result or not. The weights are updated based on this information[9] so that only the input vector can be used for weight correction.
In learning with error correction, the magnitude of the error, together with the input vector, determines the magnitude of the corrections to the weights, and in many cases we try to eliminate the error in a single correction step. The perceptron learning algorithm is an example of supervised learning with reinforcement. Some of its variants use supervised learning with error correction. In the following section we deal with learning methods for perceptrons. To simplify the notation we

[9]The Boolean values true or false.

adopt the following conventions. The input $(x_1, x_2, \ldots, x_n)$ to the perceptron is called the input vector. If the weights of the perceptron are the real numbers $w_1, w_2, \ldots, w_n$ and the threshold is θ then the threshold computation of a perceptron will be expressed using scalar products:

$$wx \geq \theta.$$

A usual approach for starting the learning algorithms to initialize the network weights randomly and to improve these initial parameters, looking at each step to see whether a better separation of the training set can be achieved.The error of a perceptron with weight vector w is the number of incorrectly classified points. The learning algorithm must minimize this error function $E(w)$. One possible strategy is to use a local greedy algorithm which works by computing the error of the perceptron for a given weight vector, looking then for a direction in weight space in which to move, and updating the weight vector by selecting new weights in the selected search direction. The optimization problem we are trying to solve can be understood as descent on the error surface but also as a search for an inner point of the solution region.
Assume that the set A of input vectors in n-dimensional space must be separated from the set B of input vectors in such a way that a perceptron computes the binary function f_w with $f_w(x) = 1$ for $x \in A$ and $f_w(x) = 0$ for $x \in B$. The binary function f_w depends on the set w of weights and threshold. The error function is the number of false classifications obtained using the weight vector w. It can be defined as: $E(w) = \sum_{x \in A}(1 - f_w(x)) + \sum_{x \in B} f_w(x)$. This is a function defined over all of weight space and the aim of perceptron learning is to minimize it. Since $E(w)$ is positive or zero, we want to reach the global minimum where $E(w) = 0$. This will be done by starting with a random weight vector w, and then searching in weight space a better alternative, in an attempt to reduce the error function $E(w)$ at each step.
A perceptron makes a decision based on a linear separation of the input space. This reduces the kinds of problem solvable with a single perceptron. More general separations of input space can help to deal with other kinds of problem unsolvable with a single threshold unit. Functions used to discriminate between regions of input space are called decision curves. Some of the decision curves which have been studied are polynomials and splines. In statistical pattern recognition problems we assume that the patterns to be recognized are grouped in clusters in input space. Using a combination of decision curves we try to isolate one cluster from the others. One alternative is combining several perceptrons to isolate a convex region of space. In the general case we want to distinguish between regions of space. A neural network must learn to identify these regions and to associate them with the correct response. The main problem is determining whether the free parameters of these decision regions can be found using a learning algorithm.
We are now in a position to introduce the perceptron learning algorithm. The training set consists of two sets, P and N, in n-dimensional extended input space. We look for a vector w capable of absolutely separating both sets, so that all vectors in P belong to the open positive half-space and all vectors in N to the open negative half-space of the linear separation. The perceptron learning algorithm may be defined as follows:

Definition 3 start: *The weight vector w_0 is generated randomly, set $t := 0$*
test: *A vector $x \in P \cup N$ is selected randomly, if $x \in P$ and $w_t x > 0$ go to test, if $x \in P$ and $w_t x \leq 0$ go to add, if $x \in N$ and $w_t x < 0$ go to test, if $x \in N$ and $w_t x \geq 0$ go to subtract.*
add: *set $w_{t+1} = w_t + x$ and $t := t + 1$, goto test*
subtract: *set $w_{t+1} = w_t - x$ and $t := t + 1$, goto test*

This algorithm makes a correction to the weight vector whenever one of the selected vectors in P or N has not been classified correctly. The perceptron convergence theorem guarantees that if the two sets P and N are linearly separable the vector w is updated only a finite number of times. The routine can be stopped when all vectors are classified correctly. The corresponding test must be introduced in the above pseudocode to make it stop and to transform it into a fully-fledged algorithm.
There are two alternative ways to visualize perceptron learning, one more effective than the other. Given the two sets of points $P \in \mathbb{R}^2$ and $N \in \mathbb{R}^2$ to be separated, we can visualize the linear separation in extended input space. We extend the input vectors and look for a linear separation through the origin, that is, a plane with equation $w_1x_1 + w_2x_2 + w_3x_3 = 0$. The vector normal to this plane is the weight vector $w = (w_1, w_2, w_3)$. The perceptron learning algorithm starts with a randomly chosen vector w_0. If a vector $x \in P$ is found such that $wx < 0$, this means that the angle between the two vectors is greater than 90 degrees. The weight vector must be rotated in the direction of x to bring this vector into the positive halfspace defined by w. This can be done by adding w and x, as the perceptron learning algorithm does. If $x \in N$ and $wx > 0$, then the angle between x and w is less than 90 degrees. The weight vector must be rotated away from x. This is done by subtracting x from w. The vectors in P rotate the weight vector in one direction, the vectors in N rotate the negative weight vector in another. If a solution exists it can be found after a finite number of steps.
Intuitively we can think that the learned vectors are increasing the *inertia* of the weight vector. Vectors lying just outside of the positive region are brought into it by rotating the weight vector just enough to correct the error. This is a typical feature of many learning algorithms for neural networks. They make use of a so-called learning constant, which is brought to zero during the learning process to consolidate what has been learned. The perceptron learning algorithm provides a kind of automatic learning constant which determines the degree of adaptivity of the weights.

Proposition 5 *If the sets P and N are finite and linearly separable, the perceptron learning algorithm updates the weight vector w_t a finite number of times. In other words, if the vectors in P and N are tested cyclically one after the other, a weight vector w_t is found after a finite number of steps t which can separate the two sets.*

Clearly, the perceptron learning algorithm selects a search direction in weight space according to the incorrect classification of the last tested vector and does not make use of global information about the shape of the error function. It is a greedy, local algorithm. This can lead to an exponential number of updates of the weight vector.
If the learning set is not linearly separable the perceptron learning algorithm does not terminate. However, in many cases in which there is no perfect linear separation, we would like to compute the linear separation which correctly classifies the largest number of vectors in the positive set P and the negative set N. Gallant [4] proposed a very simple variant of the perceptron learning algorithm capable of computing a good approximation to this ideal linear separation. The main idea of the algorithm is to store the best weight vector found so far by perceptron learning while continuing to update the weight vector itself. If a better weight vector is found, it supersedes the one currently stored and the algorithm continues to run.

4 Layered Networks

In the previous sections the computational properties of isolated threshold units have been analyzed extensively. The next step is to combine these elements and look at the increased computational power of the network. In this chapter we consider feed-forward networks structured in successive layers of computing units.

The networks we want to consider must be defined in a more precise way in terms of their architecture. The atomic elements of any architecture are the computing units and their interconnections. Each computing unit collects the information from n input lines with an integration function $\Psi : R^n \to R$. The total excitation computed in this way is then evaluated using an activation function $\Phi : R \to R$. In perceptrons the integration function is the sum of the inputs. The activation (also called output function) compares the sum with a threshold. Later we will generalize Φ to produce all values between 0 and 1. In the case of Ψ some functions other than addition can also be considered. In this case the networks can compute some difficult functions with fewer computing units.

It has been proven that this architecture can approximate any continuous function to any degree of accuracy of a compact set. The multi-layer perceptron has been termed the universal approximator. However, it is never known exactly how many hidden layers of neurons will ensure optimum network convergence and if the weight matrix that corresponds to that error goal can be found. These solutions are unique to each neural network and the input and output data applied.

A multi-layer perceptron builds on the architecture of the single layer perceptron. The single layer perceptron is not very useful because of its limited mapping ability; it is only really applicable to linearly separable inputs. It will fail if the inputs are not linearly separable. The single layer perceptron however, can be used as a building block for larger, much more practical structures. Using multi-layer architectures, non-binary activation functions and more complex training algorithms mean the limitations of a simple perceptron may be overcome. A typical multi-layer perceptron network consists of a set of source nodes forming the input layer, one or more hidden layers of computation nodes, and an output layer of node.

Definition 4 *A network architecture is a tuple (I, N, O, E) consisting of a set I of input sites, a set N of computing units, a set O of output sites and a set E of weighted directed edges. A directed edge is a tuple (u, v, w) whereby $u \in I \cup N$, $v \in N \cup O$ and $w \in R$.*

The input sites are just entry points for information into the network and do not perform any computation. Results are transmitted to the output sites. The set N consists of all computing elements in the network. Note that the edges between all computing units are weighted, as are the edges between input and output sites and computing units.

Layered architectures are those in which the set of computing units N is subdivided into ι subsets $N_1, N_2, ..., N_\iota$ in such a way that only connections from units in N_1 go to units in N_2, from units in N_2 to units in N_3, etc. The input sites are only connected to the units in the subset N_1, and the units in the subset N_ι are the only ones connected to the output sites. In the usual terminology, the units in N are the output units of the network. The subsets N_i are called the layers of the network. The set of input sites is called the input layer, the set of output units is called the output layer. All other layers with no direct connections from or to the outside are called hidden layers.

Usually the units in a layer are not connected to each other[10] and the output sites are omitted from graphical representations.
A neural network with a layered architecture does not contain cycles. The input is processed and relayed from one layer to the other, until the final result has been computed. In layered architectures normally all units from one layer are connected to all other units in the following layer. If there are m units in the first layer and n units in the second one, the total number of weights is $m \times n$. The total number of connections can become rather large and one of the problems with which we will deal is how to reduce the number of connections, that is, how to prune the network.
The properties of one and two-layered networks can be discussed using the case of the XOR function as an example. A single perceptron cannot compute this function, but a two-layered network can. Increasing the number of units in the hidden layer increases the number of possible combinations available, we say that the capacity of the network increases. This is a general feature of layered architectures: the first layer of computing units maps the input vector to a second space, called classification or feature space. The units in the last layer of the network must decode the classification produced by the hidden units and compute the final output. We can now understand in a more general setting how layered networks work by visualizing in input space the computations they perform. Each unit in the first hidden layer computes a linear separation of input space. Assume that input space is the whole of $\mathbb{R}^2$. In general, any union of convex polytopes in input space can be classified in this way: units in the first hidden layer define the sides of the polytopes, the units in the second layer the conjunction of sides desired, and the final output unit computes whether the input is located inside one of the convex polytopes.

4.1 Radial Basis Networks

A Radial Basis Function (RBF) neural network has an input layer, a hidden layer and an output layer. The neurons in the hidden layer contain Gaussian transfer functions whose outputs are inversely proportional to the distance from the center of the neuron. RBF networks are similar to K-Means clustering and PNN/GRNN networks. The main difference is that PNN/GRNN networks have one neuron for each point in the training file, whereas RBF networks have a variable number of neurons that is usually much less than the number of training points. For problems with small to medium size training sets, PNN/GRNN networks are usually more accurate than RBF networks, but PNN/GRNN networks are impractical for large training sets.
Although the implementation is very different, RBF neural networks are conceptually similar to K-Nearest Neighbor (k-NN) models. The basic idea is that a predicted target value of an item is likely to be about the same as other items that have close values of the predictor variables. RBF networks have three layers:

- Input layer There is one neuron in the input layer for each predictor variable. In the case of categorical variables, N-1 neurons are used where N is the number of categories. The input neurons (or processing before the input layer) standardizes the range of the values by subtracting the median and dividing by the interquartile range. The input neurons then feed the values to each of the neurons in the hidden layer.

[10] Although some neural models make use of this kind of architecture.

- Hidden layer This layer has a variable number of neurons (the optimal number is determined by the training process). Each neuron consists of a radial basis function centered on a point with as many dimensions as there are predictor variables. The spread (radius) of the RBF function may be different for each dimension. The centers and spreads are determined by the training process. When presented with the x vector of input values from the input layer, a hidden neuron computes the Euclidean distance of the test case from the neurons center point and then applies the RBF kernel function to this distance using the spread values. The resulting value is passed to the the summation layer.

- Summation layer The value coming out of a neuron in the hidden layer is multiplied by a weight associated with the neuron $W_1, W_2, \ldots, W_n$ and passed to the summation which adds up the weighted values and presents this sum as the output of the network. Not shown in this figure is a bias value of 1.0 that is multiplied by a weight W_0 and fed into the summation layer. For classification problems, there is one output (and a separate set of weights and summation unit) for each target category. The value output for a category is the probability that the case being evaluated has that category.

The following parameters are determined by the training process of such a neural network:

- The number of neurons in the hidden layer.
- The coordinates of the center of each hidden-layer RBF function.
- The radius (spread) of each RBF function in each dimension.
- The weights applied to the RBF function outputs as they are passed to the summation layer.

Various methods have been used to train RBF networks. One approach first uses K-means clustering to find cluster centers which are then used as the centers for the RBF functions. However, K-means clustering is a computationally intensive procedure, and it often does not generate the optimal number of centers. Another approach is to use a random subset of the training points as the centers. Alternative approaches include a training algorithm which uses an evolutionary approach to determine the optimal center points and spreads for each neuron, then it computes when to stop adding neurons to the network by monitoring the estimated leave-one-out error and terminating when the LOO error beings to increase due to overfitting. The computation of the optimal weights between the neurons in the hidden layer and the summation layer is done using ridge regression. An iterative procedure is used to compute the optimal regularization λ parameter that minimizes generalized cross-validation error.

5 Backpropagation

We saw in the last chapter that multilayered networks are capable of computing a wider range of Boolean functions than networks with a single layer of computing units.
However the computational effort needed for finding the correct combination of weights increases substantially when more parameters and more complicated topologies are considered. In this chapter we discuss a popular learning method capable of handling such large learning problems.
This numerical method was used by different research communities in different contexts, was discovered and rediscovered, until in 1985 it found its way into connectionist artificial intelligence. It has been one of the most studied and used algorithms for neural networks learning ever since.
In this chapter we present a proof of the backpropagation algorithm based on a graphical approach in which the algorithm reduces to a graph labeling problem. This method is not only more general than the usual analytical derivations, which handle only the case of special network topologies, but also much easier to follow. It also shows how the algorithm can be efficiently implemented in computing systems in which only local information can be transported through the network.
The backpropagation algorithm looks for the minimum of the error function in weight space using the method of gradient descent. The combination of weights which minimizes the error function is considered to be a solution of the learning problem. Since this method requires computation of the gradient of the error function at each iteration step, we must guarantee the continuity and differentiability of the error function. Obviously we have to use a kind of activation function other than the step function used in perceptrons because the composite function produced by interconnected perceptrons is discontinuous, and therefore the error function too. One of the more popular activation functions for backpropagation networks is the sigmoid, a real function $s_c : \mathbb{R} \to (0,1)$ defined by the expression:

$$s_c(x) = \frac{1}{1 + e^{-cx}}.$$

The constant c can be selected arbitrarily and its reciprocal $1/c$ is called the temperature parameter in stochastic neural networks. The shape of the sigmoid changes according to the value of c.
We have already shown that, in the case of perceptrons, a symmetrical activation function has some advantages for learning. An alternative to the sigmoid is the symmetrical sigmoid $S(x)$ defined as:

$$S(x) = 2s(x) - 1.$$

Many other kinds of activation functions have been proposed and the backpropagation algorithm is applicable to all of them. A differentiable activation function makes the function computed by a neural network differentiable (assuming that the integration function at each node is just the sum of the inputs), since the network itself computes only function compositions. The error function also becomes differentiable. Since we want to follow the gradient direction to find the minimum of this function, it is important that no regions exist in which the error function is completely flat. As the sigmoid always has a positive derivative, the slope of the error function provides a greater or lesser descent direction which can be followed. We can think of our search algorithm as a physical process in which a small sphere is allowed to roll on the surface of the error function until it reaches the bottom.
A price has to be paid for all the positive features of the sigmoid as activation function. The most important problem is that, under some circumstances, local minima appear in the error function

which would not be there if the step function had been used. In many cases local minima appear because the targets for the outputs of the computing units are values other than 0 or 1. If a network for the computation of XOR is trained to produce 0.9 at the inputs $(0, 1)$ and $(1, 0)$ then the surface of the error function develops some protuberances, where local minima can arise.
In this section we show that backpropagation can easily be derived by linking the calculation of the gradient to a graph labeling problem. General network topologies are handled right from the beginning, so that the proof of the algorithm is not reduced to the multilayered case. Recall that in our general definition a feed-forward neural network is a computational graph whose nodes are computing units and whose directed edges transmit numerical information from node to node. Each computing unit is capable of evaluating a single primitive function of its input. In fact the network represents a chain of function compositions which transform an input to an output vector. The network is a particular implementation of a composite function from input to output space, which we call the network function. The learning problem consists of finding the optimal combination of weights so that the network function approximates a given function f as closely as possible. However, we are not given the function f explicitly but only implicitly through some examples. Consider a feed-forward network with n input and m output units. It can consist of any number of hidden units and can exhibit any desired feed-forward connection pattern. We are also given a training set $(x_1, t_1), \ldots, (x_p, t_p)$ consisting of p ordered pairs of n- and m-dimensional vectors, which are called the input and output patterns. Let the primitive functions at each node of the network be continuous and differentiable. The weights of the edges are real numbers selected at random. When the input pattern $\mathbf{x_i}$ from the training set is presented to this network, it produces an output $\mathbf{o_i}$ different in general from the target $\mathbf{t_i}$. What we want is to make $\mathbf{o_i}$ and $\mathbf{t_i}$ identical for $i = 1, \ldots, p$, by using a learning algorithm. More precisely, we want to minimize the error function of the network, defined as

$$E = \frac{1}{2} \sum_{i=1}^{p} \parallel \mathbf{o_i} - \mathbf{t_i} \parallel^2 .$$

After minimizing this function for the training set, new unknown input patterns are presented to the network and we expect it to interpolate. The network must recognize whether a new input vector is similar to learned patterns and produce a similar output. The backpropagation algorithm is used to find a local minimum of the error function. The network is initialized with randomly chosen weights. The gradient of the error function is computed and used to correct the initial weights. Our task is to compute this gradient recursively.
Every one of the j output units of the network is connected to a node which evaluates the function $\frac{1}{2}(o_{ij} - t_{ij})^2$, where o_{ij} and t_{ij} denote the j-th component of the output vector o_i and of the target t_i. The outputs of the additional m nodes are collected at a node which adds them up and gives the sum E_i as its output. The same network extension has to be built for each pattern t_i. A computing unit collects all quadratic errors and outputs their sum $E_1 + \cdots + E_p$. The output of this extended network is the error function E. We now have a network capable of calculating the total error for a given training set. The weights in the network are the only parameters that can be modified to make the quadratic error E as low as possible. Because E is calculated by the extended network exclusively through composition of the node functions, it is a continuous and differentiable function of the l weights $w_1, w_2, \ldots, w_l$ in the network. We can thus minimize E by

using an iterative process of gradient descent, for which we need to calculate the gradient:

$$\nabla E = (\frac{\delta E}{\delta w_1}, \frac{\delta E}{\delta w_2}, \dots, \frac{\delta E}{\delta w_l}).$$

Each weight is updated using the increment:

$$\triangle w_i = -\gamma \frac{\delta E}{\delta w_i}, i = 1, \dots, l,$$

where γ represents a learning constant; a proportionality parameter which defines the step length of each iteration in the negative gradient direction.
Our objective is to find a method for efficiently calculating the gradient of a one-dimensional network function according to the weights of the network. Because the network is equivalent to a complex chain of function compositions, we expect the chain rule of differential calculus to play a major role in finding the gradient of the function. We take account of this fact by giving the nodes of the network a composite structure, each node calculates its value - up to this point nothing changes but now it will also calculate the derivative of the primitive function for the same input. Note that the integration function can be separated from the activation function by splitting each node into two parts. The first node computes the sum of the incoming inputs, the second one the activation function s. The derivative of s is s' and the partial derivative of the sum of n arguments with respect to any one of them is just 1. This separation simplifies our discussion, as we only have to think of a single function which is being computed at each node and not of two. The network is evaluated in two stages: in the first one, the feed-forward step, information comes from the left and each unit evaluates its primitive function f in its right side as well as the derivative f' in its left side. Both results are stored in the unit, but only the result from the right side is transmitted to the units connected to the right. The second step, the backpropagation step, consists in running the whole network backwards, whereby the stored results are now used. There are three main cases which we have to consider.

- Function composition: In the feed-forward step, incoming information into a unit is used as the argument for the evaluation of the nodes primitive function and its derivative. In this step the network computes the composition of the functions f and g. The correct result of the function composition has been produced at the output unit and each unit has stored some information on its left side. In the backpropagation step the input from the right of the network is the constant 1. Incoming information to a node is multiplied by the value stored in its left side. The result of the multiplication is transmitted to the next unit to the left. We call the result at each node the traversing value at this node. The final result of the backpropagation step, which is $f'(g(x))g'(x)$, i.e., the derivative of the function composition $f(g(x))$ implemented by this network.

- Function addition: The next case to consider is the addition of two primitive functions. The partial derivative of the addition function with respect to any one of the two inputs is 1. In the feed-forward step the network computes the result $f_1(x) + f_2(x)$. In the backpropagation step the constant 1 is fed from the left side into the network. All incoming edges to a unit fan out the traversing value at this node and distribute it to the connected units to the left. Where two right-to-left paths meet, the computed traversing values are added. The result

$f_1'(x) + f_2'(x)$ of the backpropagation step, which is the derivative of the function addition $f_1 + f_2$ evaluated at x. A simple proof by induction shows that the derivative of the addition of any number of functions can be handled in the same way.

- Weighted edges could be handled in the same manner as function compositions, but there is an easier way to deal with them. In the feed-forward step the incoming information x is multiplied by the edges weight w. The result is wx. In the backpropagation step the traversing value 1 is multiplied by the weight of the edge. The result is w, which is the derivative of wx with respect to x. From this we conclude that weighted edges are used in exactly the same way in both steps: they modulate the information transmitted in each direction by multiplying it by the edges weight.

We can now formulate the complete backpropagation algorithm and prove by induction that it works in arbitrary feed-forward networks with differentiable activation functions at the nodes. We assume that we are dealing with a network with a single input and a single output unit.

Definition 5 *Consider a network with a single real input x and network function F. The derivative $F'(x)$ is computed in two phases: Feed-forward: the input x is fed into the network. The primitive functions at the nodes and their derivatives are evaluated at each node. The derivatives are stored. Backpropagation: the constant 1 is fed into the output unit and the network is run backwards. Incoming information to a node is added and the result is multiplied by the value stored in the left part of the unit. The result is transmitted to the left of the unit. The result collected at the input unit is the derivative of the network function with respect to x.*

The previous definition follows from the points noted earlier. We omit a rigorous proof of this result as it escapes the scope of the project.
We consider again the learning problem for neural networks. Since we want to minimize the error function E, which depends on the network weights, we have to deal with all weights in the network one at a time. The feed-forward step is computed in the usual way, but now we also store the output of each unit in its right side. We perform the backpropagation step in the extended network that computes the error function and we then fix our attention on one of the weights, say w_{ij} whose associated edge points from the i-th to the j-th node in the network. This weight can be treated as an input channel into the subnetwork made of all paths starting at w_{ij} and ending in the single output unit of the network. The information fed into the subnetwork in the feed-forward step was $o_i w_{ij}$, where o_i is the stored output of unit i. The backpropagation step computes the gradient of E with respect to this input, $\frac{\delta E}{\delta o_i w_{ij}}$. Since in the backpropagation step o_i is treated as a constant, we finally have:

$$\frac{\delta E}{\delta w_{ij}} = o_i \frac{\delta E}{\delta o_i w_{ij}}$$

Summarizing, the backpropagation step is performed in the usual way. All subnetworks defined by each weight of the network can be handled simultaneously, but we now store additionally at each node i:

- The output o_i of the node in the feed-forward step.

- The cumulative result of the backward computation in the backpropagation step up to this node. We call this quantity the backpropagated error.

If we denote the backpropagated error at the j-th node by θ_j , we can then express the partial derivative of E with respect to w_{ij} as:

$$\frac{\delta E}{\delta w_{ij}} = o_i \theta_j.$$

Once all partial derivatives have been computed, we can perform gradient descent by adding to each weight w_{ij} the increment

$$\triangle w_{ij} = \gamma o_i \theta_j.$$

This correction step is needed to transform the backpropagation algorithm into a learning method for neural networks. This illustration of the backpropagation algorithm applies to arbitrary feed-forward topologies.

5.1 Stochastic Gradient Descent

Stochastic gradient descent is a gradient descent optimization method for minimizing an objective function that is written as a sum of differentiable functions.
Both statistical estimation and machine learning consider the problem of minimizing an objective function that has the form of a sum:

$$Q(w) = \sum_{i=1}^{n} Q_i(w),$$

where the parameter w is to be estimated and where typically each summand function $Q_i(.)$ is associated with the i-th observation in the data set (used for training).
In classical statistics, sum-minimization problems arise in least squares and in maximum-likelihood estimation (for independent observations). The general class of estimators that arise as minimizers of sums are called M-estimators. However, in statistics, it has been long recognized that requiring even local minimization is too restrictive for some problems of maximum-likelihood estimation. Therefore, contemporary statistical theorists often consider stationary points of the likelihood function (or zeros of its derivative, the score function, and other estimating equations). The sum-minimization problem also arises for empirical risk minimization: in this case, $Q_i(w)$ is the value of loss function at i-th example, and $Q(w)$ is the empirical risk.
When used to minimize the above function, a standard gradient descent method would perform the following iterations:

$$w := w - \alpha \sum_{i=1}^{n} \bigtriangledown Q_i(w),$$

where α is a step size.
In many cases, the summand functions have a simple form that enables inexpensive evaluations of the sum-function and the sum gradient. For example, in statistics, one-parameter exponential families allow economical function-evaluations and gradient-evaluations.
However, in other cases, evaluating the sum-gradient may require expensive evaluations of the gradients from all summand functions. When the training set is enormous and no simple formulas exist, evaluating the sums of gradients becomes very expensive, because evaluating the gradient requires evaluating all the summand functions' gradients. To economize on the computational cost at every iteration, stochastic gradient descent samples a subset of summand functions at every

step. This is very effective in the case of large-scale machine learning problems. In stochastic gradient descent, the true gradient of $Q(w)$ is approximated by a gradient at a single example:

$$w := w - \alpha \bigtriangledown Q_i(w).$$

As the algorithm sweeps through the training set, it performs the above update for each training example. Several passes over the training set are made until the algorithm converges. Typical implementations may also randomly shuffle training examples at each pass and use an adaptive learning rate. There is a compromise between the two forms where the true gradient is approximated by a sum over a small number of training examples.
The convergence of stochastic gradient descent has been analyzed using the theories of convex minimization and of stochastic approximation. Briefly, when the learning rates α decrease with an appropriate rate, and subject to relatively mild assumptions, stochastic gradient descent converges almost surely to a global minimum when the objective function is convex or pseudoconvex, and otherwise converges almost surely to a local minimum.

6 Interest-rate derivatives and the extrapolation problem

6.1 Introduction

Companies, both financial and non-financial, are subject to interest rate risk[11]. This risk may be derived from loans subject to floating interest rates or long-term assets which must be discounted to present value. Interest rate risk is that which exists in an interest-bearing asset, such as a loan or a bond, due to the possibility of a change in the asset's value resulting from the variability of interest rates. Interest rate risk management has become very important, and assorted instruments have been developed to deal with interest rate risk.
An interest rate derivative is a derivative[12] where the underlying asset is the right to pay or receive a notional amount of money at a given interest rate. These structures are popular for investors with customized cashflow needs or specific views on the interest rate movements (such as volatility movements or simple directional movements) and are therefore usually traded over-the-counter[13]. The interest rate derivatives market is the largest derivatives market in the world. The Bank for International Settlements estimates that the notional amount outstanding in June 2009 were 437 trillion US dollars for OTC interest rate contracts, and 342 trillion US dollars for OTC interest rate swaps. According to the International Swaps and Derivatives Association, 80% of the world's top 500 companies as of April 2003 used interest rate derivatives to control their cashflows. This compares with 75% for foreign exchange options, 25% for commodity options and 10% for stock options.

6.2 The Cap and Floor Derivatives

We will focus on a very concrete case of interest rate derivative: a cap/floor option on a Euribor interest rate. Lets consider the underlying, the Euribor interest rate. The Euro Interbank Offered Rate (Euribor) is a daily reference rate based on the averaged interest rates at which Eurozone banks offer to lend unsecured funds to other banks in the Euro interbank market. This underlying is known as a simply compounded spot interest rate and can be mathematically defined as:

Definition 6 *The simply compounded spot interest rate prevailing at time t for the maturity T is denoted by $L(t,T)$ and is the constant rate at which an investment has to be made to produce an amount of one unit of currency at maturity, starting from $P(t,T)$ units of currency at time t, when accruing occurs proportionally to the investment time. $L(t,T) = \frac{1-P(t,T)}{(t,T)P(t,T)}$ The market LIBOR rates are simply-compounded rates, which motivates why we denote by L such rates. LIBOR rates are typically linked to zero coupon-bond prices by the Actual/360*[14] *day-count convention for computing (t,T).*

$P(t,T)$ is known as a zero-coupon bond and is defined as:

Definition 7 *A T-maturity zero-coupon bond (pure discount bond) is a contract that guarantees its holder the payment of one unit of currency at time T, with no intermediate payments. The contract value at time $t < T$ is denoted by $P(t,T)$. Clearly, $P(T,T) = 1$ for all T.*

[11]Not only companies are subject to these risks, consider an individual paying mortgage installments.

[12]The term *derivative* indicates that the product *derives* its value from an underlying security

[13]In other words, contracts with very specific characteristics which cannot be exchange-traded due to their lack of standardization.

[14]This is a standard financial convention for calculating year fractions between two dates.

An interest rate cap puts an upper limit on a borrowers variable interest rate. The organization with the debt to be hedged pays an upfront fee to purchase a cap from a financial counterparty. Pricing depends on current rate movements, on how high the cap is set, and for how long. The financial counterparty is then responsible for any interest costs beyond the cap rate, should rates rise that high.
The organization retains the full benefit of lower variable rates while protecting itself from a spike in interest rates above the cap level. A cap cannot become a liability to the organization (unlike a swap), as it contains only one-way obligations to the financial counterparty after the upfront fee has been paid. Therefore, an organizations credit risk is not considered in the pricing of the cap. Lastly, if the organization wants to terminate a cap before it matures, the organization will receive a payment for the caps residual value; the higher current rates are, the higher the value. Ultimately, a cap is best suited for an organization looking to minimize its exposure to rising short-term interest rates.
In order to represent mathematically how much money a cap/floor[15] option will pay requires the introduction of two further mathematical elements:

Definition 8 *A bank account is valued at $B(t)$ at time $t \geq 0$. We assume that $B(0) = 1$ and its evolution is governed by $dB(t) = r(t)B(t)dt$.*

$r(t)$ is known as the instantaneous rate at which the bank account accrues[16].

Definition 9 *The stochastic discount factor between t and T, $D(t, T)$, is the monetary amount at time t that is equivalent to one unit of currency payable at time T: $D(t, T) = \frac{B(t)}{B(T)}$.*

The question of how much money a cap option will pay can be mathematically represented as:

$$\sum_{i=\alpha+1}^{\beta} D(t, T_i) N_i (L(T_{i-1}, T_i) - K)^+.$$

A floor option is mathematically equivalent with the only difference of $(K - L(T_i - 1, T_i))^+$ where K is a deterministic constant known as the *strike* and N is the notional of the contract and can be considered to be $N = 1$. The α, β and in the the above equation are time indexes with the following interpretation: the cap option pays for each year fraction $= {}_{\alpha+1}, \ldots, {}_{\beta}$, where ${}_i$ is the year fraction between T_{i-1} and T_i, the difference between the Euribor rate $L(T_{i-1}, T_i)$, fixed at T_{i-1} and payed at T_i, and the strike K.

6.3 Introducing Randomness

Until now we have not talked about where the stochastic nature of future values comes from. The origin is in the instantaneous rate of the bank account, $r(t)$. This element is in fact a stochastic process. As all the previous mathematical relationships we have given are derived from $r(t)$ it is clear they will all be random variables.

[15]Also known as call/put option.

[16]It is reasonable to ask what is the relationship between $r(t)$ and $L(t, T)$ which is the underlying we want to study: $r(t) = \lim_{T \to t^+} L(t, T)$.

For simplicity let us consider that the cap option only has one *caplet*, $\alpha + 1 = \beta$, the present value of its future payoff will then be:

$$D(t, T_i)N_i(L(T_{i-1}, T_i) - K)^+.$$

As indicated previously $r(t) = \lim_{T \to t^+} L(t, T)$ where $r(t)$ is an unknown real world random variable and therefore the above payoff is also random. A famous, and market-standard, formula for the valuation of the above random payoff is known as the Black-Scholes formula and assumes that $L(t, T)$ is a log-normally distributed random variable. This formula states that the discounted expectation of the future payoff is equal to:

$$P(t, T_i)N_i\Psi(K, F(t, T_{i-1}, T_i), v_i, 1), \Psi(K, F, v, w) = Fw\Phi(wd_1(K, F, v)) - Kw\Phi(wd_2(K, F, v))$$

$$d_1(K, F, v) = \frac{ln(\frac{F}{K}) + \frac{v^2}{2}}{v}$$

$$d_1(K, F, v) = \frac{ln(\frac{F}{K}) - \frac{v^2}{2}}{v}$$

w indicates whether the derivative is a cap, 1 or a floor, -1. Φ is the distribution function of a standard normal distribution. The derivation of this model is beyond the scope of this project but it is important to see that in the framework of the Black-Scholes model the parameter v is the volatility of the random variable $F(t, T_{i-1}, T_i)$ which is the forward rate of our simply compounded Euribor which was defined earlier.

Definition 10 *The simply-compounded forward interest rate prevailing at time t for the expiry $T > t$ and maturity $S > T$ is denoted by $F(t, T, S)$ and is defined by $F(t, T, S) := \frac{1}{(T,S)}\left(\frac{P(t,T)}{P(t,S)} - 1\right)$.*[17]

6.4 The Smile

As stated in the introduction there exists a large liquid[18] market in these options. Therefore we can obtain prices for a large interval of strikes, K. Further, with respect to the Black-Scholes formula, the only variable which is not directly observable is the parameter v. If we were to take market prices for different K, while keeping all other variables equal, and place them into the Black-Scholes formula we would expect to obtain a unique v for each $F(t, T_{i-1}, T_i)$. Unfortunately this is not the case, we instead observe a small curvature in the market, a sort of smile. The reasons for this have been documented in many academic publications.
If we have liquid quotes for the entire interval of K, the smile would not be a substantial problem. The fact is that the cap/floor market is only liquid for a local interval of K around $F(t, T_{i-1}, T_i)$. Therefore outside of this interval we do not have readily available prices to then extract a volatility and vice-versa. One may ask, is it necessary to have volatilities for $K \in (-\infty, \infty)$? The answer is yes. Theoretically, one should be able to price a cap at any K and furthermore there exist more exotic derivatives that are sensitive to a large part, or the entire, smile. The question now

[17]Of particular interest is the fact that conditional on the t-filtration the $F(t, T, S) = E^{B(t)}(L(T, S))$, $E^{B(t)}(.)$ is the expectation of a random variable under the unique probability measure which makes $B(t)$ a martingale.

[18]Market terminology meaning a market place with a large amount of activity and therefore reliable and ready-tradeable prices.

becomes, if the Black-Scholes framework does not allow smile yet there is smile in the market, how do market participants model the smile? The answer is using the Stochastic Alpha-Beta-Rho (SABR) model.

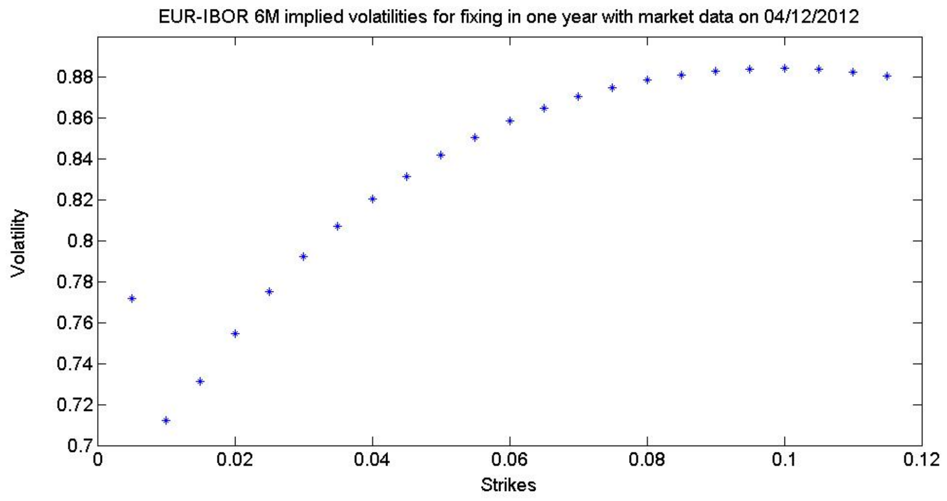


Figure 1: Market implied Black-Scholes volatilities for EUR-IBOR 6M fixing in one year.

6.5 SABR

Under the following hypothesis[19]:

$$dF(t, T_{i-1}, T_i) = \alpha F(t, T_{i-1}, T_i) dW_1(t),$$

$$d\alpha = \nu \alpha dW_2(t),$$

where $dW_1(t)$ and $dW_2(t)$ are infinitesimal increments of a Weiner processes, with correlation $dW_1 dW_2 = \rho dt$. In [5] the authors use singular perturbation techniques to isolate the v we saw in the last subsection as a function of the parameters of the above model. Using the previously established notation, they obtain:

$$v(K) = \frac{\alpha}{(fK^{\frac{1-\beta}{2}})\left(1 + (\frac{(1-\beta)^2}{24} log^2(\frac{f}{K})) + (1-\beta)^4 1920 log^4(\frac{f}{K})\right)} \times \dots$$

$$\dots \times \frac{z}{x(z)} \left(1 + \frac{(1-\beta)^2}{24} \frac{\alpha^2}{(fK)^{1-\beta}} + \frac{\rho\beta\nu\alpha}{4(fK)^{\frac{1-\beta}{2}}} + (2 - 3\rho^2 24\nu^2)_i\right),$$

$$f = F(t, T_{i-1}, T_i),$$

$$z = \frac{\nu}{\alpha} log(\frac{f}{K})(fK)^{\frac{1-\beta}{2}},$$

$$x(z) = log(\frac{\sqrt{1 - 2\rho z + z^2} + z - \rho}{1 - \rho}).$$

[19] α and β are SABR model parameters not the time indices we referenced earlier.

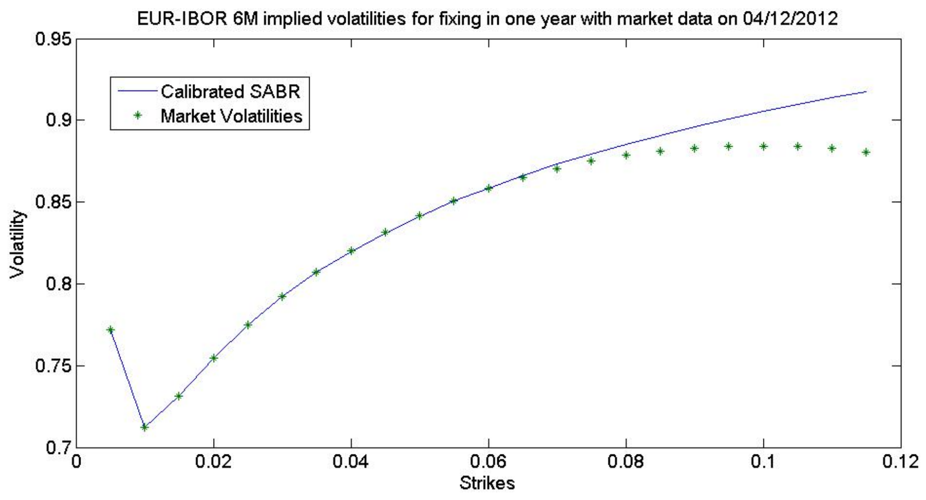


Figure 2: SABR implied volatilities for EUR-IBOR 6M fixing in one year.

6.6 What are the Implied Probabilities?

If we exclude the notional, discount factor and year fraction from the payoff of a Euribor floor derivative we end up with $(K - L(T_{i-1}, T_i))^+$. Taking the first derivative of the expectation of this random function we obtain:

$$\frac{\delta \mathbb{E}(K - L(T_{i-1}, T_i))^+}{\delta K}(x) = F_{L(T_{i-1},T_i))}(x),$$

where $F(.)$ is the probability distribution function of the random variable $L(T_{i-1}, T_i))$[20].

[20]To not over complicate things, I have excluded the measure over which the expectation is taken. This measure is the one which makes the stochastic process followed by $L(T_{i-1}, T_i))$ a martingale

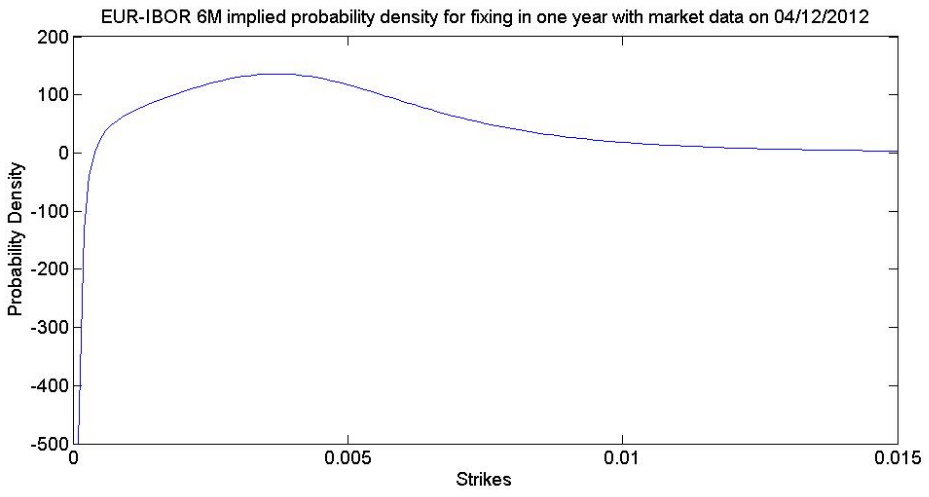


Figure 3: SABR implied probability density for EUR-IBOR 6M fixing in one year.

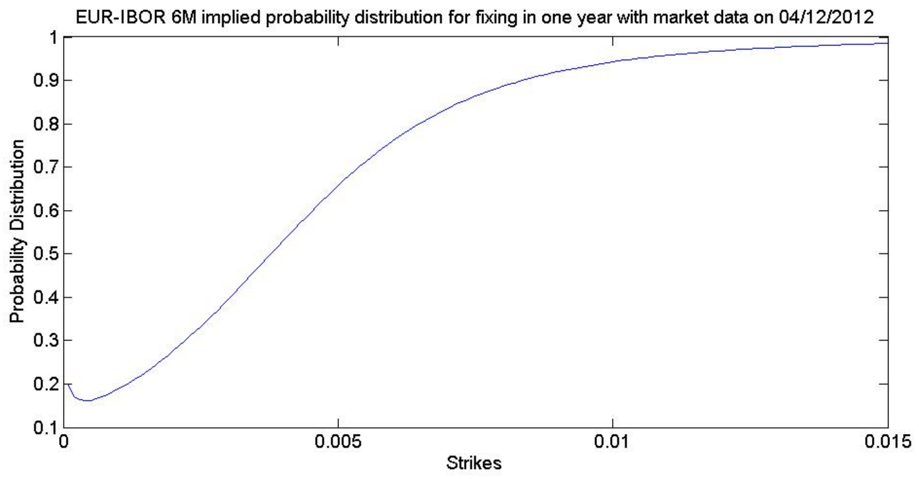


Figure 4: SABR implied probability distribution for EUR-IBOR 6M fixing in one year.

6.7 Market-standard ways of fixing SABR

Observing the implied probability density of the EUR-IBOR 6M rate we can clearly see that it does not conform to a standard probability density function; for example, in the left tail probabilities are negative. This is equivalent to saying that the second derivative in strike of floor prices must always be positive. For various technical reasons detailed in [5] this is often not the case for low strikes[21].

Market participants have several ways of fixing this problem. One way is to modify the SABR model, for example by keeping the general SABR structure but allowing for negative values of the

[21]One begins to see why the SABR model breaks down for low strikes by simply noticing that the SABR model imposes strictly positive values on the underlying rate. Evidently if the underlying is positive but very close to zero, we begin to stretch the limits of the theoretical model.

underlying:

$$dF(t, T_{i-1}, T_i) = \alpha dW_1(t),$$

$$d\alpha = \nu\alpha dW_2(t).$$

Under such a configuration, which is known as a Normal-SABR and is also detailed in [5] we obtain a much more theoretically correct implied density:

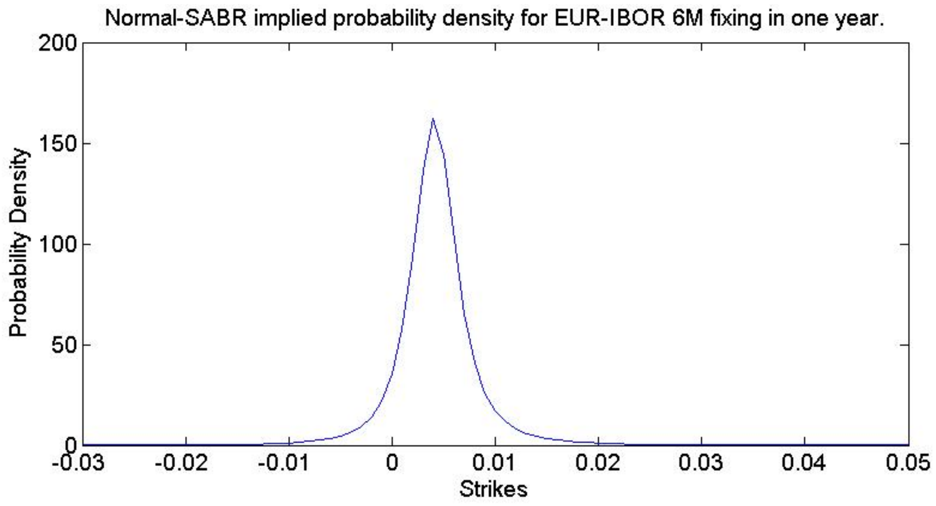


Figure 5: Normal-SABR implied probability density for EUR-IBOR 6M fixing in one year.

There exist various alternatives to modifying the entire SABR model. As the hypothesis of log-normality is a market standard[22], it would be optimal to maintain log-normality and use a parametric extrapolation to fix the negative density problem. One way of doing this is assigning floor prices the following values below a certain strike (K^-):

$$Floor(K) = K^{\mu} exp(a + bK + cK^2),$$

This idea comes from [6]. Let us fix K^- just before the log-normal implied density becomes negative. The parameters a, b and c are determined by equaling the SABR-implied prices, first and second derivatives to those of the above parametric equation at the strike K^-. This gives us the following result:

[22]It implies values and sensitivities which market participants fully understand and are accustomed to working with. The Normal-SABR model implies values which are vastly different from that of the standard SABR.

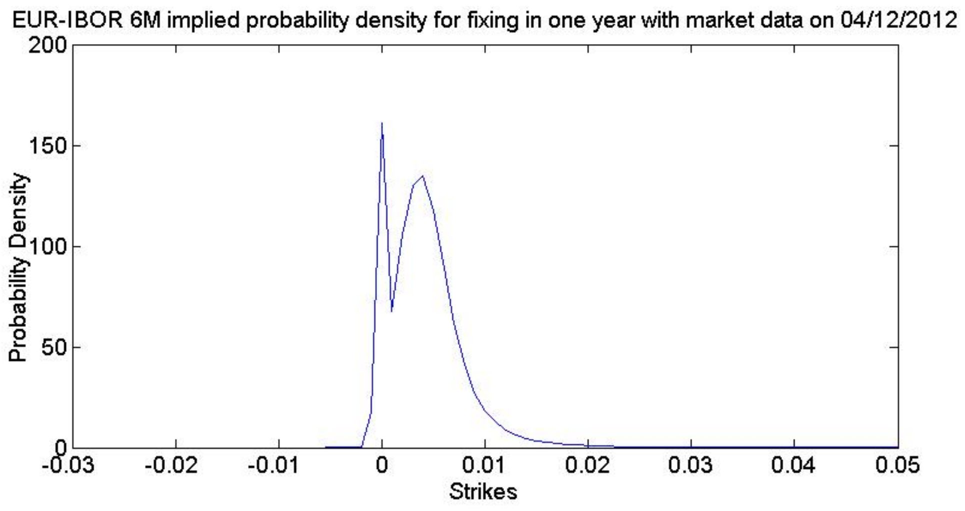


Figure 6: SABR with parametric extrapolation implied probability density for EUR-IBOR 6M fixing in one year.

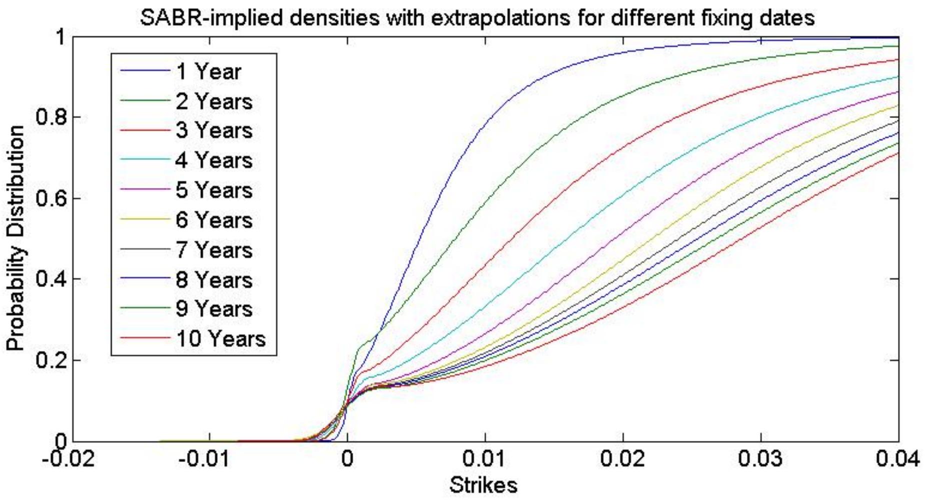


Figure 7: SABR implied probability distribution for EUR-IBOR 6M fixing in different instants.

7 Implementation

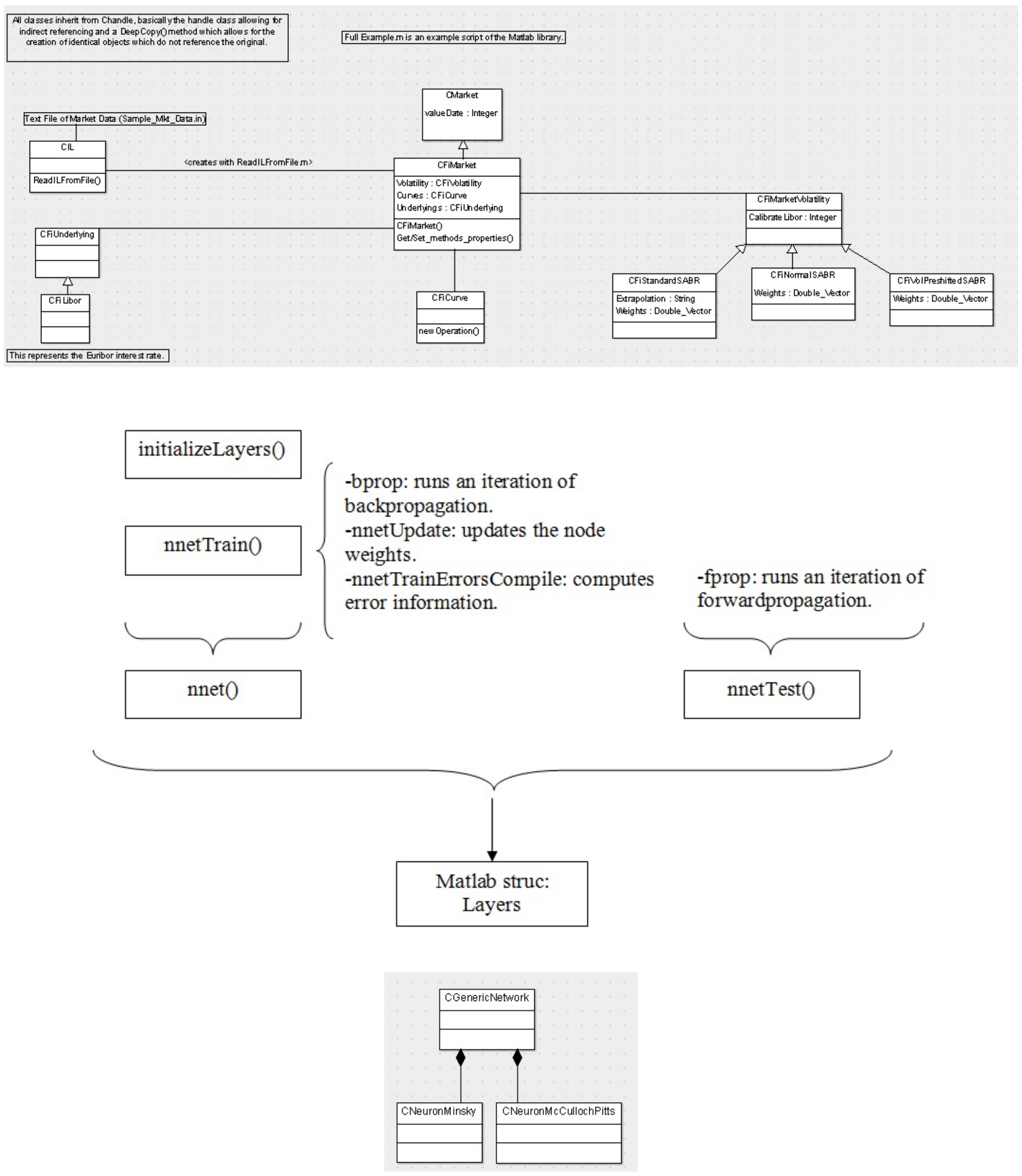
7.1 General Architecture

Market data, including the Libor volatilities and the discount curve needed to implement the model are found in Sample_Mkt_Data.in and is structured in an array of arrays. The class CIL is responsible for reading this data as a string and placing it in a dictionary format with the final data stored as an array.
This object may then be read via the readMarketIL function which extracts the data to form the market knowledge on which we will construct our model. This market object, CFiMarket, contains three important attributes, the discount curve, the volatility surface and the underlying, in this case the Libor interest rate.
The method GetFiVolatilities of the CFiMarket class extracts the volatility class which must be calibrated in order to obtain the SABR parameters. For the generic SABR case this is done by calling the function GetStandardSABRVola of the CFiStandardSABR class, note that the procedure is identical for CFiNormalSABR and CFiVolPreshiftedSABR models. This calibrated the three SABR parameters[23] with a best-fit approach between the volatilities implied by the SABR formula for the calibrated parameters and the actual market volatilities. The three volatility classes have a common Weights attribute which indicates the relative importance we assign to the different strikes of a specific maturity. The CFiStandardSABR class implements the standard SABR formula.

7.2 Neural Network Arquitecture

The McCulloch Pitts and Minsky neurons are trivially implemented. The feedforward architecture is implemented as a collection of functions. The final product of this implementation is a Matlab struc object which results from executing the nnet() function. nnetTest() runs forward propagation to allow for prediction of new data points. Adding structure to a network of neurons is somewhat more difficult. The network classes have an attribute which is a vector of its neurons and another atribute indicating structure. Structure is a matrix of 0s and 1s. The amount of rows corresponds to to the number of neurons in a specific layer, while the columns indicate the number of layers. A 0 corresponds to no neuron occupying that position wheras a 1 indicates the position is occupied. Relationships is a cube which indicates what connections are present between the different neurons of the consecutive layers. The first segment of the cube corresponds to the first neuron, this slice is a matrix with columns indicating layers and rows, the consecutive neurons. A 1 indicates a connection and a 0 indicates that the two neurons are not connected. The implementation of the radial basis architecture was taken directly from the Matlab neural network toolbox for comparative purposes.

[23] β is not calibrated due to specific technical issues detailed in [5] and is therefore fixed at 0.50.


All classes inherit from Chandle, basically the handle class allowing for indirect referencing and a DeepCopy() method which allows for the creation of identical objects which do not reference the original.

Full Example.m is an example script of the Matlab library.

This represents the Euribor interest rate.

-bprop: runs an iteration of backpropagation.
-nnetUpdate: updates the node weights.
-nnetTrainErrorsCompile: computes error information.

-fprop: runs an iteration of forwardpropagation.

Figure 8: General and Neural/Feedforward infrastructure.

8 Results

8.1 A First Attempt

The main problem we have encountered with modelling interest-rate volatility is that the market standard model implies negative densities for low strikes. We have also seen in the previous section two ways of fixing this problem, the first being to modify the entire model and allow for negative rates of the underlying. The second modification is to allow for a log-normal SABR above a certain strike and allow for a regime change below a certain strike, over which a parametric price extrapolation is used.
An immediate question that can be asked is; can a neural network adequately fix the smile extrapolation problem? The idea would be to train the network on the part of the smile which is theoretically sound and observe its behaviour for other strikes. With respect to which neural network would be most adequate for such a task, we believe it to be the radial basis style neural network as such networks have excellent interpolation properties. A radial basis function network is an artificial neural network that uses radial basis functions as activation functions. It is a linear combination of radial basis functions. They are used in function approximation, time series prediction, and control.
The first question we should ask is to what type of data do we need to train the neural network? For example, do we want to train the network to the implied probability density or rather to floor prices? The answer seems to be obvious, train the network to the most stable function. Given that floor prices are monotone and stable, the network fit will be better. Training the network to the density, which is simply the second derivative of floor prices, implies fitting the network to a function which is not monotone and of a peculiar shape. Further, the range of strikes on which we train the network must be on those where the implied probability density fulfills the necessary theoretical conditions.
For consistency with previous chapters, we focus on the market traded EUR-IBOR 6M fixing in one year. For this case the range of strikes where the density behaves correctly is $(0.04\%, 3.50\%)$.
We can graphically observe this case in the following graphs:

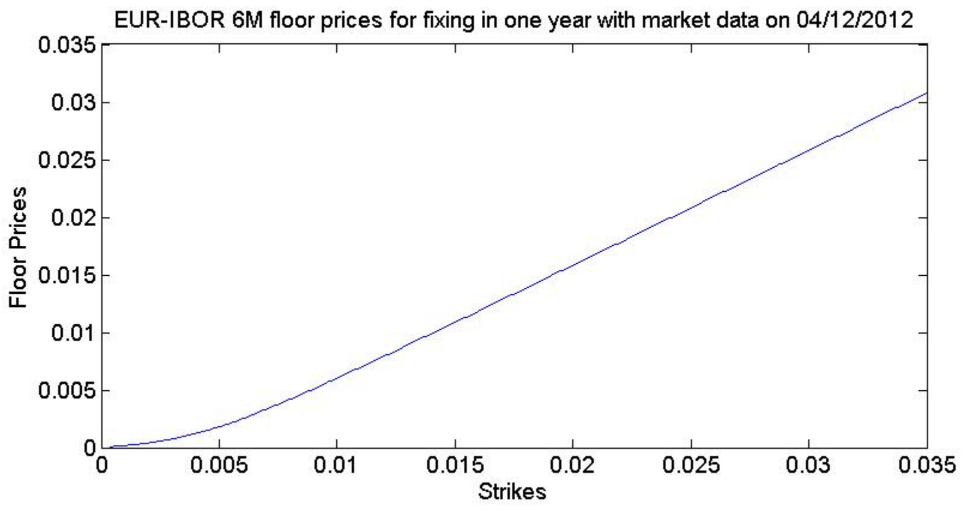


Figure 9: SABR with parametric extrapolation implied probability density for EUR-IBOR 6M fixing in one year.

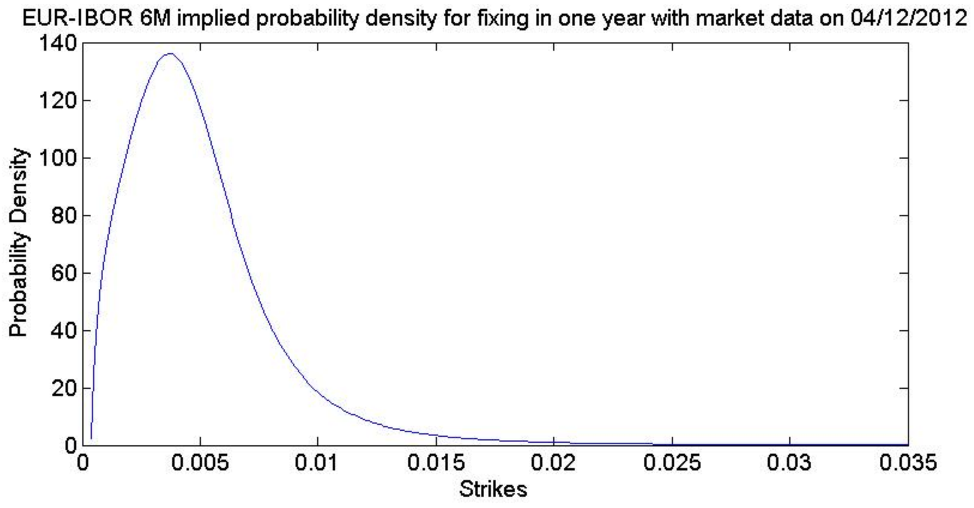


Figure 10: SABR with parametric extrapolation implied probability density for EUR-IBOR 6M fixing in one year.

The next step is to feed these prices (we will also attempt to calibrate to the density for comparison) into a radial basis network and observe the results. Calibrating to floor prices in the interval $(0.04\%, 3.50\%)$ in increments of 0.01% we obtain:

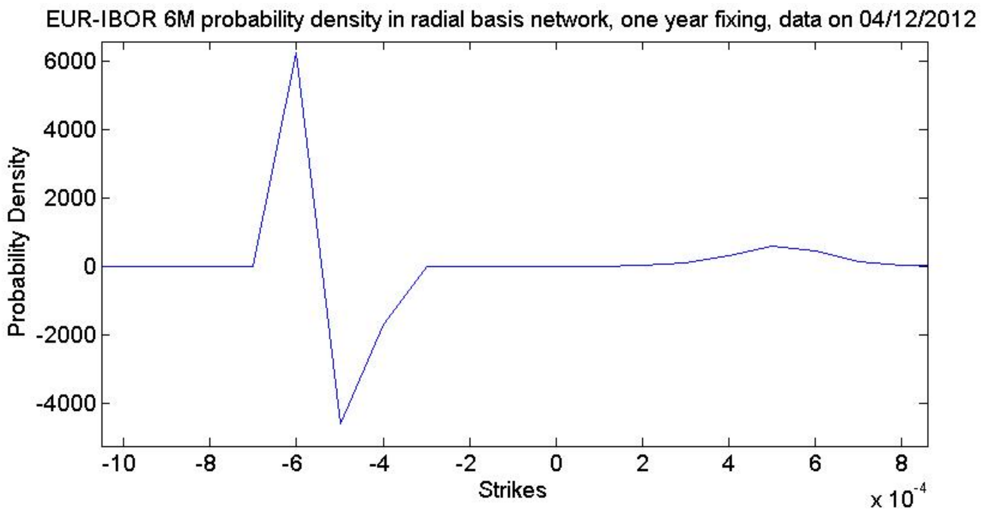


Figure 11: Radial basis neural network calibrated to prices: implied probability density for EUR-IBOR 6M fixing in one year.

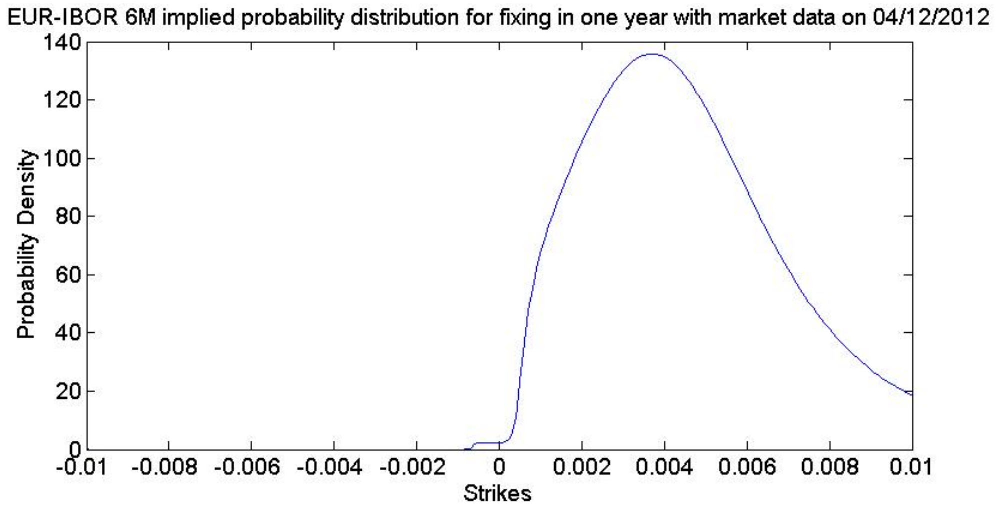


Figure 12: Radial basis neural network calibrated to density: implied probability density for EUR-IBOR 6M fixing in one year.

We can observe that generally the extrapolation capabilities of the radial basis network is suboptimal. From an interpolation point of view, the network is fitting correctly but as far as extrapolation is concerned the results are far from desirable. Contrary to what we initially expected calibrating the network to the SABR density provides notably better results than the network which is calibrated to prices. The implied density of this network shows large negative values in negative strikes close to zero. The network which is calibrated directly to the density provides better results, but evidently the recuperation of floor prices is not feasible.

8.2 Main Application

Before describing the main application of this project, let us review what major problem banks and other market participants may be facing in the current environment. We saw in the section which described the SABR model that the implicit density in this model is negative for small strikes. Historically this has not been a problem, when interest rates are high few caps and floors are created with low strikes, as they lie far away from the current forward rate. However in recent years, in the Euro market, interest rates have fallen to levels which are very close to zero and therefore many products are traded with strikes in the neighbourhood of zero.

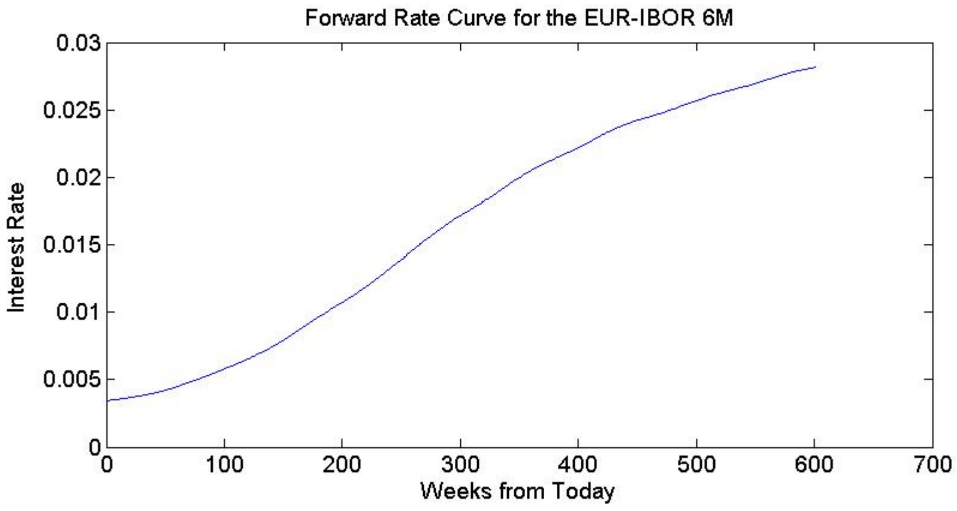


Figure 13: The forward interest rate curve for a six month tenor. Observe that for short maturities the curve is very close to zero.

While the interpolation results of a neural network are acceptable, we have seen that the extrapolation capabilities are not of the quality we were expecting. So we now proceed to consider another possible application. Market participants are accustomed to the lognormal assumption when pricing interest rate derivatives. Further, risk management systems generally have lognormal assumptions hardcoded into their implementations. It is therefore desirable to maintain this hypothesis.
We have seen that the only way to maintain lognormality and adequately model floor prices over negative strikes is to use the following extrapolation:

$$Floor(K) = K^{\mu} exp(a + bK + cK^2).$$

The problem with this approach is that the Black-Scholes framework does not allow us to move between prices and volatilities when either the forward $F(t, T_{i-1}, T_i)$ or the strike K is negative. One might ask if this is problematic? A necessary practice with respect to risk management and control of these derivatives is to know the variation of price due to small variations in the volatility[24]. Market standard practice for obtaining prices from volatilities is to use the Black-Scholes model. Therefore in low interest rate climates, there is a serious problem in assigning

[24]This is basically the first derivative of price to volatility.

valuation sensitivities to different strikes. A trivial solution to this problem is simply to assign all interest rate sensitivity to K^- which is where we began the extrapolation. The idea would be to calculate the numerical derivative at K^- of the value of a derivative with strike $K < K^-$. The problem is that all sensitivity to volatility would be located at K^- whereas the theoretical sensitivity lies at K.
The question then becomes, how might this be solved? One possibility is to use a feedforward neural network to obtain volatilities from floor prices and strikes. The training set will be the area of strikes where the implied probability density is well behaved, specifically; $(K^-,)$[25]. With such a configuration we will then be able to obtain price sensitivities to volatilities of strikes which we could not initially determine due to the theoretical structure imposed by Black-Scholes.

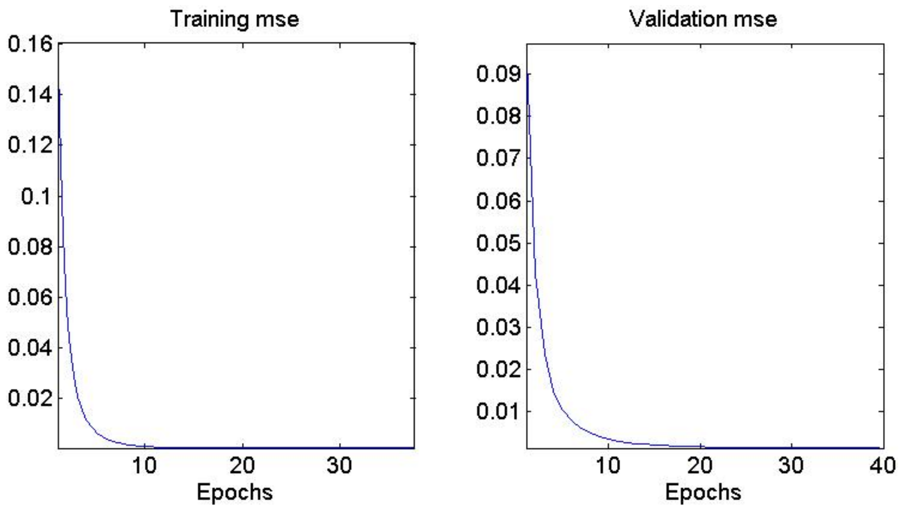


Figure 14: A two layer, ten neurons in each layer feedforward neural network with stochastic backpropagation over forty iterations.

In the following table we can see that the mean squared error[26] is stable and therefore we conclude that with forty epochs we have a sufficient calibration for the structure of the network.

[25]In practice we do not generate floor prices until but rather until such a strike where floor prices are almost linear in K.

[26]Understood to be the squared difference of the resulting volatility of the network and the actual volatility of the testing set.

Iter 31	mse: 0.00145462	Time: 0.811205
Iter 32	mse: 0.00144669	Time: 0.826805
Iter 33	mse: 0.00143991	Time: 0.842405
Iter 34	mse: 0.00143409	Time: 0.873606
Iter 35	mse: 0.00142909	Time: 0.920406
Iter 36	mse: 0.0014248	Time: 0.967206
Iter 37	mse: 0.0014211	Time: 1.01401
Iter 38	mse: 0.00141792	Time: 1.01401
Iter 39	mse: 0.00141516	Time: 1.04521
Iter 40	mse: 0.00141279	Time: 1.07641

Table 1: Mean squared error and computation time of the last forty-one iterations of the calibration.

Weights first layer	Weights second layer
0.600510186023136	-0.0709985988941079
-0.294864600721183	0.203317635474503
-0.166409786634788	0.0738165053838822
0.0100117464877854	0.591066809446162
0.220338548506403	-0.0862844562383130
0.424920390399659	0.400184799101871
0.178658987774449	0.375881447609364
-0.137954326759397	-0.166467787379095
0.323558826489155	0.209453481326522
0.184333809702096	0.283712897241195

Table 2: One weight per each neuron, ten neurons per layer and two layers making up the network.

Observing the following graph we see the implicit smile is not as smooth as expected:

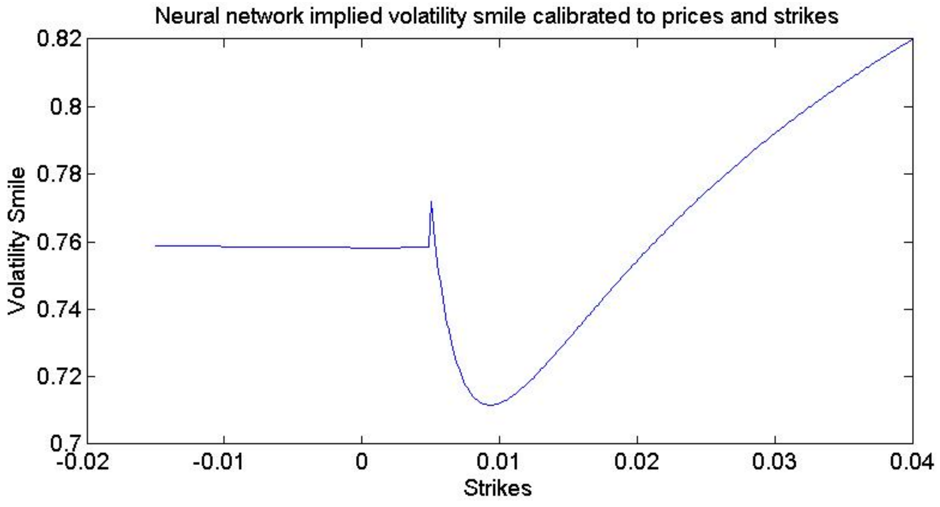


Figure 15: Volatility generation from two layer feedforward network.

Recalibrating the network to these volatilities and strikes to generate prices we obtain:

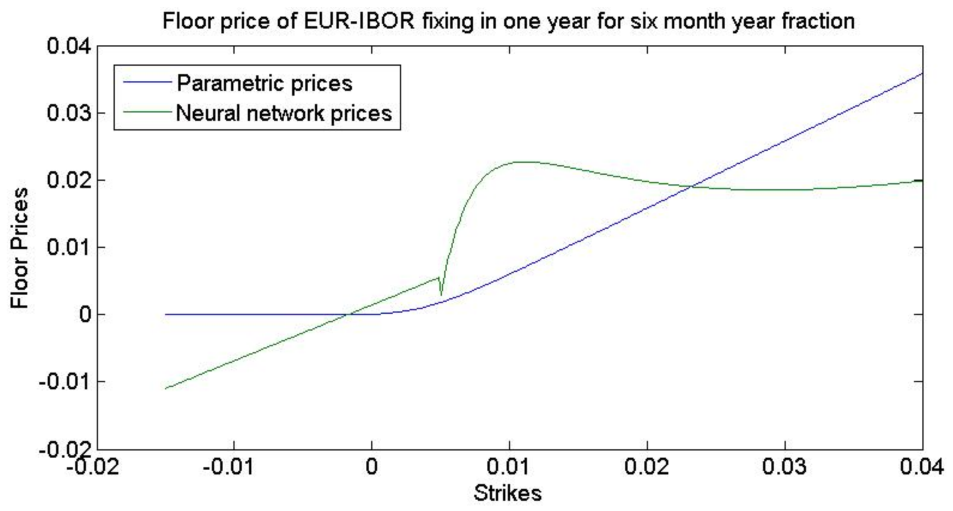


Figure 16: Floor prices implied by a two layer neural network.

We see that the results are not as optimal as we might have hoped. Let us observe the results based on sensibilities to the volatility smile for specific strikes:

Floor price with strike	Neural network	Extrapolation
0.0389	-0.23924674	0.000118371
0.0390	-0.239190984	0.000117775
0.0391	-0.239135461	0.000117182
0.0392	-0.239080169	0.000116594
0.0393	-0.239025109	0.00011601
0.0394	-0.238970279	0.00011543
0.0395	-0.238915677	0.000114855
0.0396	-0.238861303	0.000114283
0.0397	-0.238807155	0.000113716
0.0398	-0.238753233	0.000113153
0.0399	-0.238699535	0.000112594
0.0400	-0.23864606	0.000112038

Table 3: Comparison of the two models to calculate sensitivity to volatility of floor prices for a selection of strikes.

We see the two models imply vastly different values for the sensitivity of price to the volatilities of different strikes. Given that the neural network allows us to maintain a lognormal hypothesis and also assigns sensitivity below K^- it has a clear advantage. The problems with calibration of the neural network render it an unacceptable model for pricing purposes. Simply by viewing the implied floor prices in comparison to the SABR implied prices, we see that the neural network implies notable arbitrages and fails in maintaining smoothness and necessary theoretical properties.

9 Conclusions

We have studied the general theoretical properties of neural networks and have reviewed in depth specific types; the feedforward and radial basis network. The feedforward network uses backpropagation with stochastic gradient descent to calibrate the weights of the different nodes whereas the radial basis network calibrates with backpropagation using deterministic gradient descent but has a radial basis activation function and are the sort of network well suited to interpolation functions. The concrete applications that we have applied these structures to were an attempt to extrapolate the Black-Scholes implied volatility smile to strikes lower than the area of which the SABR model breaks down. Generally the results were not optimal, in other words the cumulative distribution function was not smooth and the density function displayed behaviour that was not fully consistent with the rest of the density.
As a second application we observe that it is advantageous to maintain the hypothesis of lognormality in the modelling of the underlying interest rate forward. In order to correct the problem of the implied density being negative for low strikes, we use a parametric extrapolation of floor prices which then modifies the SABR implied density and allows for a positive density over all strikes. The problem with such a mechanism is that the Black-Scholes mapping between prices and volatilities is only valid over positive strikes due to the hypothesis of lognormality. The application consists of calibrating a multilayer feedforward neural network onto a set of strikes and prices and have it learn the relationship with volatilities, we may then calibrate a new neural network to these previously generated volatilities and strikes to learn the relationship with floor prices. Using this structure, we can then bump the implied volatilities with a small increment to compute price variations in floor prices. This allows us to compute price sensitivities of floor derivatives to movements in volatility for strikes below the strikes of extrapolation. This is something that could not be done with the standard Black-Scholes framework. The result is made even more general by way of direct application to cap prices with Merton's [3] put-call parity formula:

$$Cap(K) - Floor(K) = F(t, t_{i-1}, t_i) - Ke^{-rt}.$$

Reviewing the previous chapter we see the results are less than adequate. Floor prices present a somewhat arbitrary structure when reproduced with the neural network, even for large amounts of epochs in the calibration and complex network structures[27].
As further developments for the networks which I have developed, I would like to extend the range of their applications. One possibility being a direct comparison between SABR and a neural network price, taking as inputs variables such as current forward interest rate, previous day's option price, time to expiry, etc. Further, due to the complex modelling relationship that exists between the inputs and outputs, it may be instructive to use recurrent networks and other more complex topologies to improve the final results.

[27] I attempted to calibrate a ten layer ten neuron architecture with results which did not improve notably with respect to the design in the previous chapter.

References

[1] Amilon, H., “A Neural Network Versus Black-Scholes”, J. of Forcast. **22**, 317-335 (2003).

[2] Bottou, L., “Stochastic Gradient Learning in Neural Networks”, ATT Bell Laboratories (2001).

[3] Brigo, D. and Mercurio, F., “Interest Rate Models: Theory and Practice”, Springer Finance (2001).

[4] Gallant, S. I., “Perceptron-based Learning Algorithms”, IEEE **1**, 179-191 (1990).

[5] Hagan, P. S., et. al., “Managing Smile Risk”, Wilmott Magazine. (2002).

[6] Kainth, D. S., et. al., “Smile Extrapolation and Pricing of CMS Spread Options”, Presentation Global Derivatives (2011).

[7] Kon, M., Plaskota, L., “Neural Networks, Radial Basis Functions and Complexity”, Unpublished.

[8] Malliaris, M., Salchenberger, L., “A Neural Network Model for Estimating Option Prices”, J. of App. Intell. **3**, 193-206 (1993).

[9] McCulloch, W. S. and Pitts, W. H., “A Logical Calculus of the Ideas Immanent in Nervous Activity”, B. of Math. Biophysics **5**, 115-133 (1943).

[10] Minsky, M. and Papert, S., “Perceptrons”, MIT Press (1969).

[11] Rosenblat, F., “The perceptron: A probabilistic model for information storage and organization in the brain”, Psychological Review **65**, 386-400 (1958).

[12] Sebastian, H., Werfel, J. and Xie, X., “Learning curves for stochastic gradient descent in linear feedforward networks”, Unpublished.

[13] Yu, H., et. al., “Comparison between traditional neural networks and radial basis function networks”, Unpublished.