UNIVERSITY OF TURIN

DEPARTMENT OF PHYSICS

MASTER DEGREE IN PHYSICS OF COMPLEX SYSTEMS

Options Market with Agent-based Simulation and Neural Networks for pricing

Floriana Coello

Supervisor: Prof. Pietro Terna

Examiner: Prof. Michele Caselle

A.A. 2015/2016

"A nonno Gino"

"You can always go further than you never stop"

Jack Kerouac

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Is it possible to replace Black–Scholes formula for European options pricing with neural networks? The answer seems to be positive, following the results of this project.

The recent financial crisis has highlighted the need for a deep understanding of financial instruments and how to properly deal with them. Complex derivatives, as options, have been one of the principal causes for the collapse of several banks in USA. It was argued that the Black–Scholes equation for options pricing was the mathematical justification behind the trading that contributed to the outbreak of financial world in 2008. Despite it is considered the holy grail of investors, this formula is valid only under strict market'conditions, which cannot be found in real markets. In 2012 Stewart published an article in the Guardian describing how Black–Scholes contributed to the 2008 financial crisis. The following words explain his misgivings about BS method: «The equation is based on the idea that big movements are actually very, very rare. The problem is that real markets have these big changes much more often that this model predicts. [...]And the other problem is that everyone's following the same mathematical principles, so they are all going to get the same answer.»

This dissertation aims to find an alternative method to the Black–Scholes formula for options pricing in an artificial market under high volatility. It has been developed an Agent–based model for the simulation of an option market and several artificial neural networks for prices prediction.

Agent–based modelling is a powerful simulation technique that allows to run micro–scale simulations of simultaneous operations and interactions of multiple agents, with the aim to re–create and predict the appearance of complex phenomena. In order to design this model, it has been used the NetLogo platform, which is the most widely used software for ABM simulation, specifically for simulating natural and social phenomena.

NetLogo offers a well–stocked models library, in which one can find the *Op-*

tionsMarket [1] model implemented by Elliott: this model provides an important starting point for this project. The *OptionsMarket* aims to simulate a market trading options made by intelligent agents, who buy and sell options according to Blake–Scholes formula. The intelligence of traders is modelled through features of patience and judgement. Patience concerns to how quickly the agent enters the market and how much of the past market it looks at in it evaluations, especially in considering the underlying stock volatility. Judgement refers to how far the trader considers personal judgement and deviates from Black–Scholes formula. These individual properties are used by agents to calculate their own volatility, which is necessary in Black–Scholes formula. So that the BS formula is valid, the spot price has to be updated with Geometric Brownian motion.

In order to get the most truthful possible simulation, changes have been necessary. In real markets there are not only intelligent traders, but also agents who operate without experience or careful consideration. Following this idea, the agents of the original model have been split in two different classes: the first class of traders buys and sells options according to patience and judgement, while the other one behaves in a random way because they have not individual characteristics and their volatility is a random number between zero and one. The presence of random agents makes the options market a market with high variability.

As previously mentioned, the prediction is obtained with artificial neural networks, implemented in R pseudo–code. R is a software which provides not only a wide variety of statistical and graphical techniques but also a great number of packages to implement different types of neural networks.

Prediction of BS formula is the first object of this project. All traders collect data about time to maturity, spot (updated with GBM), volatility and option prices evaluated using BS formula in lists, which will used for training neural networks. This means that, so far, at the end of training period neural networks are able to predict options prices according to Black–Scholes formula. Consequently, despite neural networks perform optimally, their outputs are erroneous, as the market presents randomness, under which BS theory is no more valid.

The next and final step of this work is proposed to correct the call and put prices predicted by neural networks. The BS theory is based on different hypothesis, one of them is that the spot price must follow a Gaussian Brownian Motion. However, in real market, the price of the underlying does not follow a Gaussian Brownian Motion. The idea is to introduce in the option market model a continuous double auction (CDA) model, which provide the simulation of the stock market. Under this assumption, the BS Formula is not more valid and the neural networks, properly trained, can predict options prices.

Chapter 2 introduces the the options as financial instruments, giving initially an overview of what they are, what types and classes of options exist and for

what purposes they are used. After a brief history of their birth and when they have been regularized, Section 2.2 provides a detailed explanation about how the options market works. Finally, Section 2.3 covers theories for options pricing, from Binomial model (one stage BM and multi stage BM) to Black–Scholes formula. Section 2.4 is completely reserved to the mathematical proof of Black–Scholes formula and the explanation of the assumptions under which the theory is valid and the reasons why the formula is incorrect in real market, despite it is the only still used in financial world.

Chapter 3 begins with a brief overview of Agent–based modelling before going into more details, considering all the elements necessary for Agent–based simulation, as the agents, environment and interactions. Section 3.1 presents a detailed list of the main benefits of ABM (emergent phenomena, heterogeneity, flexible, etc.), while Section 3.2 introduces the theory of complex systems and explains in depth the importance of Agent–based model for the study of emergent phenomena. Subsequently, a brief introduction concerning the ABMs applications in financial markets and the software NetLogo used for simulating ABMs.

Chapter 4 provides an introduction to artificial neural networks. Initially, it presents the first neural network models of McCulloch–Pitts and Rosenblatt in order to give a general overview on the ways these mathematical models work. Sections 4.3 and 4.4 regards the architectures (feedforward and recurrent) and the classes of learning (supervised and unsupervised) of neural networks, giving a particular importance to backpropagation learning algorithm. Section 4.5 dives into the specific types of neural networks as multilayer perceptron, radial basis function, Kohonen networks, Hopfield networks and convolutional neural networks. Finally, some studies in financial market using artificial neural networks for prediction.

Chapter 5 presents the statistical software R, which is the programming language used to implement neural networks in this work. In Section 5.4 several possibilities of neural networks packages are listed; in particular, one can find a more detailed explanation about the package employed in this work (*neuralnet*).

Chapter 6 is the main part of the thesis, as it concerns the models which have been designed. First of all, it is presented the *OptionsMarket* [1] model by Elliott, used as starting point. Thus, the following sections, concern the changes and optimizations to the code in order to obtain an options market model as realistic as possible, which provides options price prediction using neural networks and no more Black–Scholes formula.

Chapters 7 and 8 cover the analysis of results obtained by simulations of markets and conclusions, respectively.

Chapter 2

Options Theory

Options are financial instruments that can be used effectively under almost every market condition and for almost every investment goal.

Options are contracts which give the buyer the right, but not the obligation, to buy or sell an underlying asset at a specific price (strike price) on or before a certain date. The date on which the contract may be exercised depends on option's style (or family). American style options allow the holder to exercise at any time before or at expiration, on the other hand European style options allow the holder to exercise only at expiration.

The two most common types of options contracts are put and call options, which give the holder the right to sell or the right to buy respectively the underlying asset at the strike if the spot price (the price of the underlying) crosses the strike.

Options are regulated by the Securities and Exchange Commission (SEC) and are traded by individuals, institutional investors and professional traders, in securities market places like the NASDAQ Options Market, International Securities Exchange, Chicago Board Options Exchange, Boston Options Exchange, NYSE AMEX, etc.

2.1 History

The use of options as a financial instruments is not a new innovation at all, indeed the basic concept of options contracts is believed to have been established in Ancient Greece.

The first example of options has been documented on a book written in the mid fourth century BC by Aristotle. In his work "Politics", the Greek philosopher told the tale of another philosopher and mathematician, Thales of Miletus and how he made a fortune by snapping up options on the right to use olive presses before a particularly strong harvest. He managed to predict that there would a significant

demand for olive presses and wanted to corner the market. However, Thales did not have sufficient funds to own all the olive presses, so he paid the owners of olive presses a sum of money in order to secure the rights to use them at harvest time. When the huge harvest time came around, Thales resold his rights on olive presses to those who needed them and made a considerable profit. Although the term "option" was not used at that time, Thales was the first who used a call option with olive presses as the underlying security.

The tulip bulb mania of 1636 in Holland was another relevant occurrence in the history of options. In this period the contracts prices for tulips bulbs reached incredibly high levels (some tulips sold for more than ten times the annual income of a skilled craftsman) and then suddenly collapsed. The increasing popularity of tulips market led investors to use options, in particular for hedging purposes. Tulip growers would buy puts to protect their profits just in case the price of tulips bulbs decrease, while tulip wholesalers would buy calls to protect against the risk of the price of tulips bulbs increase. The speculators were attracted by the tulip market and also families invested huge quantities of money in these contracts: many merchants sold all of their belongings to purchase a few tulip bulbs for the purpose of cultivating and selling them for more profit. Like all speculative bubbles, the Dutch tulips bulbs bubble continued to inflate beyond people's wildest expectations until it abruptly "popped" in the winter of 1636-37, when some tulip contracts reached a level about 20 times the level of three months earlier. Ordinary people had lost all their money and their homes. The brutal popping of the tulips bulbs bubble ended the Dutch Golden Age and hurled the country into a mild economic depression that lasted for several years. The tulip bulb mania is considered the first speculative economic bubble in history.

During history, options have been banned numerous times in many parts of the world: largely in some states of America, Japan, Europe and in particular in London. Despite the development of an organized market for calls and puts during the late 1600s and the increasing appeal for many investors, opposition to them was not overcome and eventually options were made illegal in the early eighteenth century.

A notable development in the history of options trading involved an American financier by the name of Russell Sage. It was not until the late 19th century that Sage conceived a method of pricing options in relation to the price of the underlying security and interest rates, creating a form of standardized pricing, according to Louisiana State University professor of finance Don Chance. In the early 1900s, brokers began to place advertisements in financial journals on the part of potential options buyers and sellers, in hopes of attracting another interested party. The options trading market that Sage started, and that he gave up after he lost a fortune in the market crash of 1884, proceeded to function without his participation.

Options continued to trade in an unregulated manner all the way till the establishment of the SEC (Security and Exchange Commission) after the great depression. In 1935 the SEC granted the Chicago Board of Trade (CBOT) a license to register as a national securities exchange. However, that license went unused for more than three decades as the market continued to trade non–standardized privately negotiated options contracts. The Put and Call Brokers and Dealers Association was created in these same years to better organize the over–the–counter markets.

About 100 years following the introduction of options trading to the US market, the most important event in modern options trading history took place with the formation of the Chicago Board of Exchange (CBOE) and the Options Clearing Corporation (OCC) in 1973. The establishment of both institutions is a milestone in the options trading history and have defined how options are traded over a public exchange in the way it is traded today. The most important function of the CBOE was the standardisation of the stock options: for the first time, the general public is able to trade call options under the performance guarantee of the OCC and the liquidity provided by the market maker system. By 1977, also put options were introduced by the CBOE.

In 1973, not only the CBOE opened its doors, but two economists, Fischer Black and Myron Scholes, published an article titled *The Pricing of Options and Corporate Liabilities* in the University of Chicago's Journal of Political Economy. Black and Scholes devised a mathematical formula for pricing put and call options, based on stochastic calculus. During these years, their colleague Robert Merton published an additional study and mathematical amplification of the Black–Scholes model. In 1997, Merton and Scholes received the Nobel Memorial Prize in Economic Sciences for their work. Black was mentioned as a contributor by the Swedish Academy, although ineligible because of his death in 1995.

In 1982, the listed options market reached more than 500,000 contracts traded in a single day. Options popularity continued to increase with more than 3.8 billion equity options traded during 2014, with an average of 16.9 million contracts traded each day across a dozen exchanges.

2.2 Options Market

Options are traded both on exchanges and in over–the–counter market [2]. As stated before, there are two types of options: call option which provides its owner the right to buy the asset by a certain date at a given price and put option which provides its owner the right to sell the asset by a certain date at a given price. Only European options are treated in this work, namely options contract which can be exercised only at expiration date. In the exchange–traded equity option market, one contract is usually an agreement to buy or sell 100 shares [3].

The power of options lies in their versatility. There are several reasons for trading options, as these contracts offer interesting investment possibilities. First and foremost, options are mainly used for hedging risk, actually these contracts were created for this purpose. For instance, a portfolio composed by a stock and a put option is equivalent to a portfolio limited on a possible loss in the stock value. This means that the put contract ensures against a drop in the overall market value and this portfolio is called "insured portfolio". Furthermore, also financial leverage is one of the biggest benefits of trading options. With many financial instruments (as stocks) the only way to take advantage of leverage is to borrow funds and this is not always possible. Buying options is similar to borrow money for investing in stocks, but without needing to use borrowed capital and with the possibility to control a larger number of shares for the same investment, than purchasing shares themselves. However, these investments might be risky.

Every option contract has two parties involved. One is the person who will own the option, named buyer, holder or owner, the other one is the person who will sell the option, named seller or writer. There are four types of options traders in the market:

- buyers of calls
- sellers of calls
- buyers of puts
- sellers of puts

One says that buyers are "long" (long position) while seller are "short" (short position).

Consider European options, in particular a call with a maturity T which provides the right to buy an asset at expiration for the strike price K. Indicate by $S(t)$ the spot price (the market value of the underlying) at time t. At maturity, if the spot price $S(T)$ is bigger than the strike price K, then the owner decide to exercise the call option because he will pay K for an asset that is worth more than K. On the contrary, if the spot price $S(T)$ is smaller than the strike price K, the holder will not exercise the contract, as the asset can be buy at a lower price in the market. Mathematically, the payoff of an European call option is:

$$C = \max(0, S(T) - K) \tag{2.1}$$

Formula (2.1) represents the payoff that the writer of the call pays to the buyer at maturity. More precisely, the value C is the payoff the seller has to cover in the contract as the seller delivers the asset worth $S(T)$ and he gets K in returns

if $S(T) > K$, namely if the option is exercised. If $S(T) < K$, the payoff is zero and the call is not exercised.

An European put option, which gives the right to sell the underlying $S(T)$ at K, is exercised only if the strike price is lower than the spot price at expiration date T. The holder decide to sell the asset if $S(T) < K$, because otherwise he would sell the underlying in the market for the price $S(T) > K$. The mathematical formula to evaluate the payoff of a put option is:

$$P = \max(0, K - S(T)) \tag{2.2}$$

Options can be *in the money*, *out of the money* or *at the money*, according to the relative position of the spot price of an underlying asset with respect to the strike price of an options (moneyness). If an option would make money if it is exercised at maturity, it is said to be *in the money* (ITM), while if it would not make money it is said to be *out of the money* (OTM) and if the spot price and the strike price are equal, it is said to be *at the money* (ATM) [3].

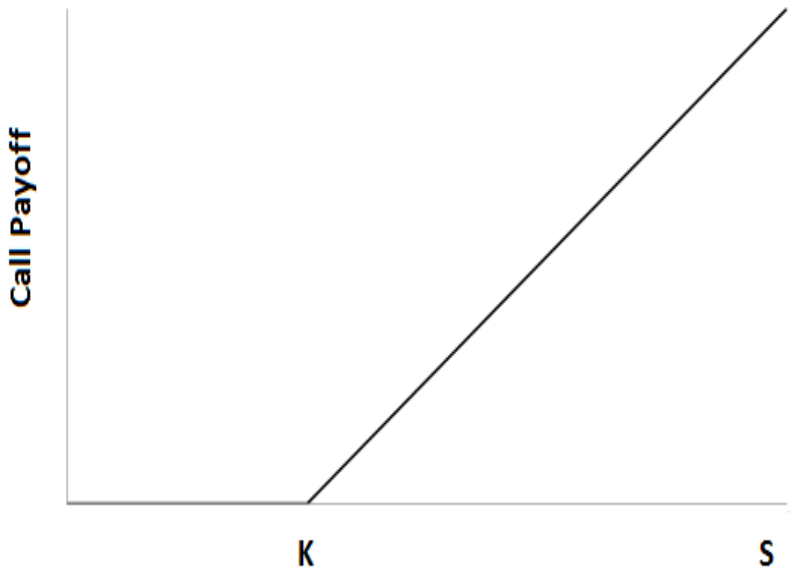


Figure 2.1: Call option at exercise time

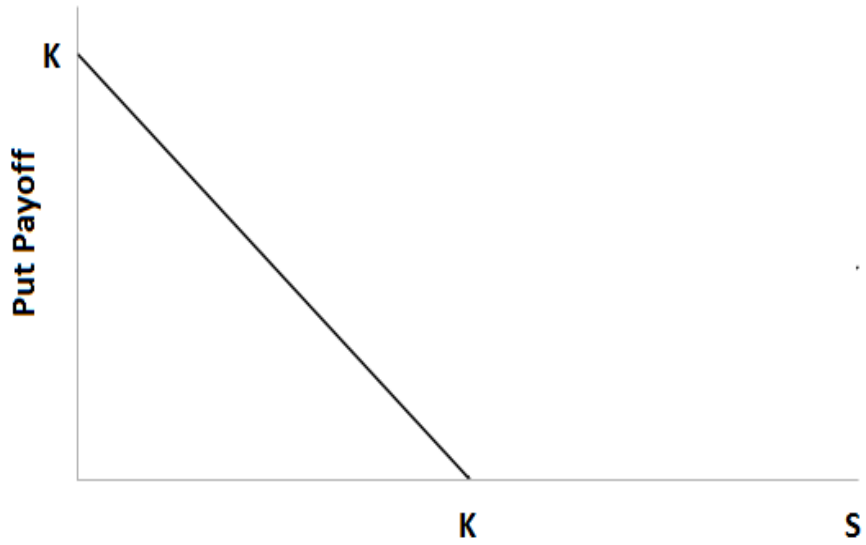


Figure 2.2: Put option at exercise time

2.3 Options Pricing

Generally, value of any financial instrument is evaluated as the present value of the expected cash flows on that asset. However, the value of options, as that of all derivative instruments, depend on the value of other assets. Indeed, the price of an option is determined by six variables relating to the market and, in particular, to the underlying.

- *Spot Price of the Underlying*, is the market value of the underlying asset. Variations of spot price affect the value of the option on that asset: call option value will increase if the spot increases; on contrary, put option value will decrease if the spot increases.
- *Variance of the Underlying*, is directly correlated to options value, both calls and puts. Greater is the volatility, greater is the option price.
- *Strike Price of the Option*, is the price at which the option is exercised. In particular, the strike is the price at which the owner of a call option can

purchase the underlying or the price at which the owner of a put option can sell the underlying.

- *Maturity*, is the date of expiration of the contract. If the option is European, it can be exercised only at maturity, not before. American options can be exercised at any moment.

- *Risk-Free Interest Rate*, for standard options pricing models (as Black-Scholes model), are used risk free one-year Treasury rates. An increase in interest rate causes an increase of call options value and a decrease of put options value.

- *Dividends Paid on Underlying*, affects options value through their effect on the underlying price. The asset value is expected to drop if dividends on the asset are paid during the option contract. Therefore, an increment of dividend payments will cause a decrease of call options price and an increase of put options price.

Before the current used model for options pricing was developed, most of previously study has been expressed in terms of warrants. Sprenkle (1961), Ayres (1963), Boness (1964), Samuelson (1965), Baumol, Malkiel, Quandt (1966) and Chien (1970) all elaborated pricing formulas of the same form, with one or more arbitrary parameters and then not complete. Sprenkle developed a formula similar to the final Black–Scholes formula, which is reported below.

$$\alpha\, S\, N(d_2) - \beta\, K\, N(d_2) \tag{2.3}$$

$$d_1 = \frac{\ln(\alpha\, S/K) + \frac{1}{2}\,\sigma^2\,(T-t)}{\sigma\sqrt{T-t}} \tag{2.4}$$

$$d_2 = \frac{\ln(\alpha\, S/K) - \frac{1}{2}\,\sigma^2\,(T-t)}{\sigma\sqrt{T-t}} \tag{2.5}$$

The Sprenkle's formula uses some of the elements previously described: the spot price (S), strike price (K), volatility of the underlying asset (σ), maturity T and the current time (t). $N(d_1)$ and $N(d_2)$ are the cumulative normal density function of parameters d_1 and d_2. Furthermore, Sprenkle defined parameters α and β as the ratio of the expected price of the underlying at maturity to the current stock price and as the discount factor depending on the volatility of the stock, respectively.

In turn, Samuelson [4] introduced in the pricing formula two other parameters: one for the rate of the expected return on the stock and one for the rate of the expected return on the warrant. The most important contribution that Samuelson offered to option pricing theory was the introduction of the "Geometric " or "Economic Brownian motion" in order to explain the fluctuation of stock market (instead of the ``arithmetical Brownian motion ''used by Bachelier). This supposition lead to the fact that the stock values are lognormally distributed. Then, he took the expected value of this distribution, cut it off at the strike and discounted it to the present at the rate of expected return on the warrant.

Few years later, Samuelson and Merton [5] published a paper on realistic theory of warrant price with the goal to overcome some deficiencies of the previous work of Samuelson. They recognised that discounting the expectation of the distribution at warrant expiration date was not a suitable procedure. Furthermore, the paper had the intention to demonstrate that the option price depends on stock price. Their final evaluation formula is determined on the shape of the utility function that they accept for the typical investor.

Another important breakthrough was made by Thorp and Kassouf [6], who designed a warrant–stock diagram to understand how warrant prices move up and down with stock prices. They found the rule that warrant and stock prices from day to day usually increase or decrease together. From this rule, Thorp and Kassouf obtain a pricing formula for warrants and they use it to compute the ratio between the shares of stock and the options needed to create a hedged position by going short in one asset and long in another one. These results were the starting point for the most complete option pricing formula which is still used: the Black–Scholes formula.

2.4 Black-Scholes Model

In 1973, Robert C. Merton, Fischer Black and Myron Scholes reached the greatest achievement in the history of options pricing. The model received the Nobel Memorial Prize in Economic Sciences in 1997. From the Black–Scholes–Merton model, one can derive the Black–Scholes formula for pricing European options [7]. As mentioned before, this formula involved a boom in options trading and legitimized the operations of Chicago Board Options Exchange and other options markets in the world.

BS formula was designed to evaluate options using a "replicating portfolio", that is a portfolio made up of the underlying and the risk–free asset, which has the same cash flows as the option, and under no arbitrage conditions. Furthermore, the evaluation is based upon the assumption that the option exercise does not affect the price of the underlying.

The complete and detailed list of the "ideal conditions" under which the formula is valid is shown below.

Hypothesis on the market:

- No arbitrage opportunity.
- Possibility to buy and sell any amount of the stock. Also short selling are allowed
- No transaction costs in buying or selling the option or the stock
- Possibility to borrow and lend any amount of cash at the risk–free rate.
- Market is efficient

Hypothesis on the underlying asset and option:

- The spot price follows a random walk in continuous time with a variance proportional to the square root of stock price. Thus, the instantaneous log–return of stock price is normally distributed. This property derives from the fact that stock prices follows a Geometrical Brownian Motion (GBM) with drift and volatility constant (Appendix A)
- The rate of return of the risk–free asset is known and constant through time. It is called risk–free interest rate
- No dividend are paid by the stock
- Option must be European

Under these conditions the price of the option depends only on spot, stock volatility and time and other variables known and constant (risk–free interest rate and strike price). Obviously, real market does not satisfy all these assumptions. In particular, the stock price does not follow a Gaussian Brownian Motion, as a stock market exists which governs the prices. However, Black–Scholes formula is considered the best formula for European options pricing and it is used in all options market around the world.

2.4.1 Binomial Model

Before deriving the Black–Scholes formula, is good practice explain an alternative method to evaluate options in discrete time. Binomial Model involves construction of binomial trees, which represents the different paths the underlying asset can follow still option maturity. The fundamental assumption is that the stock price follows a random walk, more precisely each time step the stock price can go up or down by certain amounts and with certain probabilities. In the limit time step goes to zero, one finds the lognormal distribution of stock prices return and thus the Black–Scholes formula. In other words, the Binomial Model is the Black–Scholes model in discrete time.

The assumptions under which the model works are:

- Complete and efficient Market
- No arbitrage conditions
- No dividend yield

Furthermore, the spot price can develop in two possibles states whatever the initial price.

One stage Binomial Model

Initially, consider a one stage model and a stock asset whose initial value is S_0 and an option on this asset whose initial value is f_0. Suppose that the option expires at $T > 0$ and that during its lifetime the stock price could increase to $S_T = S_0 u$ or decrease to $S_T = S_0 d$, with probabilities $(1 - q)$ and q, respectively. The proportional increase of the stock price when there is an upward movement is $(u - 1)$, whereas the proportional decrease of the stock price when there is a downward movement is $(1 - d)$. In order to rule out any possibility of arbitrage, it must have $d < r < u$, where r is the risk–free interest rate.

Since initial stock price is such that:

$$S_0 = \frac{q\, S_0\, d + (1 - q)\, S_0\, u}{1 + r} \tag{2.6}$$

it is simply derive the probabilities for down state (2.7) and up state (2.8) of spot price:

$$q = \frac{u - (1 - r)}{u - d} \tag{2.7}$$

$$1 - q = \frac{(1 + r) - d}{u - d} \tag{2.8}$$

With the aim to understand how the Binomial Model evaluate options, consider an European call option with maturity T and strike price K. Indicate c_u the final value of the call option if the spot goes up and f_d the final value of the call option if spot goes down. At expiration date the call can assume one of these two values:

$$c_d = \max(S_0\, d - K, 0) \tag{2.9}$$

$$c_u = \max(S_0\, u - K, 0) \tag{2.10}$$

Consider two–period economy, the initial option price can be written as:

$$c_0 = \frac{c_d\, q + c_u\, (1 - q)}{1 + r} \tag{2.11}$$

Substituting the expressions (2.7) (2.8) for probabilities and (2.9) (2.10) for option states in equation (2.11) , it get the formula for European call option price for one stage Binomial Model:

$$c_0 = \frac{\frac{u-(1-r)}{u-d} \max(S_0\, d - K, 0) + \frac{(1+r)-d}{u-d} \max(S_0\, u - K, 0)}{(1 + r)} \tag{2.12}$$

Equation (2.12) shows a striking property that the current value of the option does not depend in any way on the probabilities q and $1 - q$. This property means that the option price f_0 is the same whether the price of the underlying S_0 goes to $S_0 u$ with high or low probability. This characteristic is due to the existence of a portfolio that replicates perfectly the payoff of the option. Two agents may have different subjective probabilities associated to two possible different final prices of the underlying; however, the hedged portfolio does not depended on risk appetite or opinions of agents, but only on the assumption that ones attempts to take advantage of any arbitrage opportunity. Therefore, it is not necessary know the preferences structure of investors. Investors with this property are named ″risk neutral investors‟and thus the probabilities that spot goes up or down, q and $(1 - q)$, are "risk neutral probabilities" [1].

[1]Risk neutral probability, or equivalent martingale probability, is a probability measure such that the asset fair value is computed as its expectation discounted by risk–free interest rate under this measure. Its name comes from the fact that, under risk neutral probabilities, all financial assets in economy have the same expected return (the risk–free return), regardless of their risk level. These measures satisfy the fundamental theorem of asset pricing, which provides conditions for a market to be complete and arbitrage–free; indeed, risk neutral probability rules out arbitrage possibilities. The risk–neutral probability measure is indicated by Q

Multiple stage Binomial Model

The Binomial Model also enables a multi–period stage model. Although the model with one stage offers good numerical results, it is not closed to reality assume spot price can take only two states during option's lifetime. To overcome this limit, it is possible divide in N periods the time corresponding to the option lifetime and thus get nn different possible path for spot price with j jumps up ($j : 1, N$). Figure 2.3 shows a typical multi–period tree of Binomial Model. Paths are mutually exclusive, namely the stock price can follow only one path, and steps are independent over time. Under these assumptions, the expression for call option price under no arbitrage condition becomes:

$$c_0 = (1+r)^{-N} \sum_{j=0}^{N} \max(S_0\, u^j\, d^{N-j} - K,\, 0)\, Q\, nn \tag{2.13}$$

Q is the binomial distribution probability which in this case represents the probability that in each single path of N periods there are j jumps up. Consequently, the formula (2.13) can be rewritten in the following way:

$$c_0 = (1+r)^{-N} \sum_{j=0}^{N} \max(S_0\, u^j\, d^{N-j} - K,\, 0)\, (1-q)^j\, q^{N-j} \binom{N}{j} \tag{2.14}$$

The Binomial multi–period procedure described for evaluating European call options can also be applied to the evaluation of European put options, without substantial alterations.

2.4.2 Derivation of Black Scholes Formula

In *The Pricing of Options and Corporate Liabilities* [7], Black and Scholes derived the partial differential equation, now known as the Black–Scholes equation, whose result is the formula for calls and puts pricing. The BS differential equation is an equation which has to be satisfied by the price of any derivative dependent on a stock which pays no dividend. Considering f the price of an European call or put option, function of stock price S and time t, the Black–Scholes differential equation is:

$$\frac{\partial f}{\partial t} + r\, S\, \frac{\partial f}{\partial S} + \frac{1}{2}\, \sigma^2\, S^2\, \frac{\partial^2 f}{\partial S^2} - r\, f = 0 \tag{2.15}$$

From a strictly financial point of view, the equation says that one can perfectly hedge the call or put simply buying or selling the underlying stock in the right way

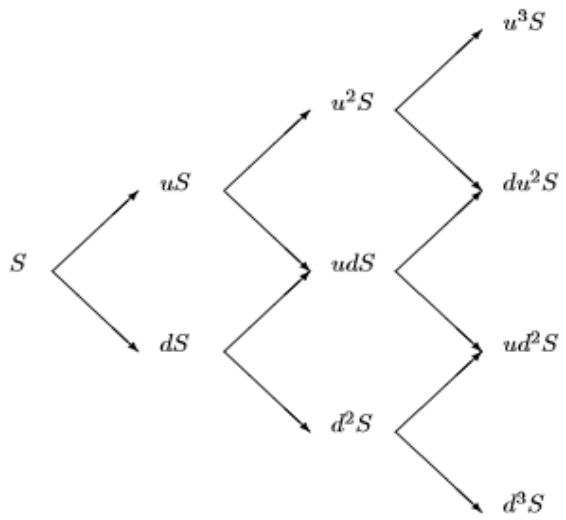


Figure 2.3: Multiple stage tree of Binomial Model

to rule out the risk. This implies that there is only one fair value for an option, which is given by Black–Scholes formula:

$$c(S,t) = S(t)\,N(d_1) + K\,N(d_2)e^{-r(T-t)} \tag{2.16}$$

$$p(S,t) = K\,N(-d_2)\,e^{-r(T-t)} - S(t)\,N(-d_1) \tag{2.17}$$

$$d_1 = \frac{\ln(S/K) + \frac{1}{2}\,\sigma^2\,(T-t)}{\sigma\sqrt{T-t}} \tag{2.18}$$

$$d_2 = d_1 - \sigma\sqrt{(T-t)} \tag{2.19}$$

Let:

- S, spot price
- K, strike price
- σ , standard deviation of the underlying asset
- r, annualized risk–free interest rate

- $T - t$, time to maturity

At last, $N(x)$ is the standard normal cumulative distribution function:

$$N(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{z^2}{2}}\, dz \tag{2.20}$$

Derivation via replicating portfolio

The original proof of BS formula starts from the assumption that the hedge portfolio must replicate the payoff of the option at maturity. First of all, the replicating portfolio V has to be built with n securities S and the option, for instance a call c. Consider $c = c(S, T - t)$ the price of an European call option.

$$V = n\, S + c \tag{2.21}$$

Computing the infinitesimal change of the portfolio V , by applying the Itô Lemma [2] to the call c, one gets:

$$dV = n\, dS + \frac{\partial c}{\partial t}\, dt + \frac{\partial c}{\partial S}\, dS + \frac{1}{2} \frac{\partial^2 c}{\partial S^2}\, S^2\, \sigma^2\, dt \tag{2.22}$$

Underlying stock price is a random variable lognormally distributed, thus its differential is:

$$dS(t) = r\, S\, dt + \sigma\, S\, dW_t^Q \tag{2.23}$$

where dW_t^Q is a stochastic variable (Wiener process) and drift and variance are the risk–free interest rate and volatility on the underlying, respectively. Notice that the Wiener process W represents the only source of uncertainty. Substituting Formula (2.23) in (2.22), one gets:

$$dV = n\,(r\, S\, dt + \sigma S\, dW_t^Q) + \frac{\partial c}{\partial t}\, dt + \frac{\partial c}{\partial S}\,(r\, S\, dt + \sigma S\, dW_t^Q) + \frac{1}{2} \frac{\partial^2 c}{\partial S^2}\, S^2\, \sigma^2\, dt \tag{2.24}$$

[2] Itô Lemma corresponds to the Taylor's expansion for stochastic calculus. Consider an Itô process (or generalised Wiener process) x, which evolves with the dynamics $dx(t) = \mu(x,t)\, dt + \sigma(x,t)\, dW_t$ and define $g \in C^2$ such that $Y = g(x,t)$. Itô Lemma states that $dY = \frac{\partial g}{\partial t}\, dt + \frac{\partial g}{\partial x}\, dx + \frac{1}{2} \frac{\partial^2 g}{\partial x^2}\, \sigma^2(x,t)\, dt$.

In order to obtain an instantaneously riskless portfolio, the stochastic terms dW_t^Q must be eliminated. The percentage of securities n^* which, included in Formula (2.24), reduces to zero the risk is:

$$n^* = -\frac{\partial c}{\partial S} \tag{2.25}$$

Furthermore, to satisfy the assumption of no arbitrage, the rate of return of a riskless portfolio must be equal to the risk–free interest rate r, that is:

$$dV = r\,V\,dt \tag{2.26}$$

Finally, replacing the explicit expression for variables dV, V and n^* in Formula (2.24), one gets:

$$\frac{\partial c}{\partial S}\,S\,r\ \ - r\,c + \frac{\partial c}{\partial t} + \frac{1}{2}\,\frac{\partial^2 c}{\partial S^2}\,S^2\,\sigma^2 = 0 \tag{2.27}$$

that is the Black–Scholes partial differential equation (2.15). The boundary condition needed to solve the PDE is:

$$C(S,T) = \max(S - K,\,0) \tag{2.28}$$

when the option is at maturity.

Using the Feynman–Kac formula, one finds that the solution of the PDE in Equation (2.27) is given by:

$$c = e^{-r\,(T-t)}\,\mathbb{E}^Q\left[\max(S_T - K,\,0)\right] \tag{2.29}$$

Q indicates that the expected value is computed under risk neutral probability. Notice that Black–Scholes model works only in a risk-neutral world, as risk neutral probabilities allow to avoid arbitrage opportunities.

Rewrite the Formula (2.29) as:

$$c = e^{-r\,(T-t)}\left\{\,\mathbb{E}^Q\left[S_T\,1_{(S_T-K)}\right] - K\,\mathbb{P}(S_T > K)\right\} \tag{2.30}$$

Next goal is to demonstrate the two following equalities:

$$e^{-r(T-t)}\,\mathbb{E}^Q\left[S_T\,1_{S_T-K}\right] = S\,N(d_1) \tag{2.31}$$

$$\mathbb{P}(S_T - K) = N(d_2) \tag{2.32}$$

First of all, let $t = 0$ and remember that because of underlying asset price is lognormally distributed ($S \sim logN$), the spot at time $t = 0$ can be expressed as:

$$S_T = S_0\, e^{(r-\frac{1}{2}\sigma^2)\,T+\sigma\,W_T^Q} \tag{2.33}$$

Since

$$S_T = S_0\, e^{m\,T+\sigma\,W_T^Q} \;>\; K \tag{2.34}$$

$$W_T^Q \;>\; \frac{\ln(K/S) - m\,T}{\sigma} \equiv \gamma \tag{2.35}$$

where $m = (r - \sigma^2/2)$. After substituting S and making some adjustments the expression for call value become:

$$c = e^{-r\,(T-t)}\,\{\, S_0\, e^{m\,T}\,\mathbb{E}^Q\left[\, e^{\sigma\,W_T^Q}\,1_{(W_T^Q\,>\,\gamma)}\right] - K\,\mathbb{P}(S_T \;>\; K)\} \tag{2.36}$$

Given $X \sim N(M, Y^2)$, one can obtain

$$\mathbb{E}[e^{a\,X}\,1_{(X\,>\,Z)}\,] = e^{M\,a+Y^2\,a^2/2}\,N\left(\frac{M - Z + a\,Y^2}{Y}\right) \tag{2.37}$$

Thus, if X becomes $W_T^Q = N(0, T)$ and $a = \sigma$

$$S_0\, e^{m\,T}\,\mathbb{E}^Q\left[\, e^{\sigma\,W_T^Q}\,1_{(W_T^Q\,>\,\gamma)}\right] = S_0\, e^{\,m\,T+T\,\frac{\sigma^2}{2}}\;N(d_1) \tag{2.38}$$

Remembering the definition of probability density function, one find that the probability to exercise the call $\mathbb{P}(W_T^Q \;>\; \gamma)$ is equal to $N(d_1)$.

$$\mathbb{P}\left(W_T^Q \;>\; \frac{\ln(K/S) - m\,T}{\sigma}\right) = \mathbb{P}\left(-\frac{W_T^Q}{\sqrt{T}} \;<\; \frac{\ln(S/K) + m\,T}{\sigma\,\sqrt{T}}\right) = N(d_1) \tag{2.39}$$

Finally, one gets the Black–Scholes valuation of European call option at maturity:

$$c(S,T) = S_T\, N(d_1) + K\, N(d_2) e^{-r\,T} \tag{2.40}$$

Similarly, it is possible obtain the proof of the Black–Scholes formula for European put options price.

2.4.3 Limits of Black-Scholes Formula

Despite Black–Scholes formula is considered one of the most cardinal concept in modern financial theory, it has several limitations [8]. Indeed, the majority of assumptions under which BS model works cannot be satisfied in real markets. In Section 2.4, all BS conditions are been listed and explained; now "non real life" hypothesis will be discussed.

Efficient Market Hypothesis (EMH)

This assumption suggests that one cannot regularly predict the direction of stocks prices. Indeed, one of the main hypothesis of the model is that the stock price evolves as a random walk. This means that the price of the underlying can increase or decrease with same probabilities and the value at time t is independent from that at time $t + dt$ (martingale property of Brownian motion). This is not close to reality, because of prices are determined by several economic factors.

No commission and costless trading

Black–Scholes model assumes that there are no barriers to trading and no cost for buying or sell. However, usually traders have to pay commissions to buy or sell options. The ignorance of fees can often distort the output of the model. To overcome this limitation, in 1976 Jonathan Ingerson relaxed the the assumption of no fees or transaction costs.

Perfect liquidity market

Real investors are limited by capital they can invest and policy of the company.

Possibility to buy and sell any amount of assets

In real market, it may not be possible buy and sell a fractional amount of assets.

Geometrical Brownian motion

Black–Scholes model is mainly based on the assumption that stock price, in continuous time, follows a Geometric Brownian motion. This condition implies that

volatility and drift (in this case risk–free interest rate) remain constant and the log–return of stock price is a random variable normally distributed. Actually, there is evidence that the return is not normally distributed, but it has a leptokurtic distribution. This discrepancy leads to a substantial underpricing or overpricing of the option. In fact, although GBM offers a good approximation for simulating the movements of spot prices, they evolve in stocks markets which are influenced by many economic factors.

Constant and known volatility

Volatility is the most critical element for pricing options, as options values are very sensible to a little changes in volatility. The model assume this parameter constant, but since the 1987 stock market crash this hypothesis has proven false [9]. Volatility can be constant for short term period, but in long time period there are several fluctuation of volatility value. Furthermore, volatility cannot be observed, but it must be estimated. A typical method to compute volatility of a given underlying asset is to use the option price. Given the option price, formula and other inputs, one can deduce the volatility implied by the option price. This volatility is named "implied volatility", and it is used in Black–Scholes method. It also considers that moneyness (OTM, ATM, ITM) of the options does not affect implied volatility; on the contrary, in reality a volatility skew is observed. Figure 2.4 shows that higher values of volatility correspond to lower strike prices. In conclusion, since measures of volatilities are negatively correlated with stock price returns, whereas the number of trades are positively correlated, the constant volatility assumption of BS model is completely unrealistic over time.

Constant and known interest rate

This assumption is true only for short term options, indeed interest rate cannot be constant during long time period. Furthermore, the model employs the risk–free rate for this constant and known rate. In the real markets, there is no such thing as a riskfree rate; it is possible to use the U.S. Government Treasury Bills 30–day rate which is considered enough similar to riskless interest rate. However, these treasury rates can change in times of increased volatility. In 1976, Merton removed the restriction of constant interest rates.

No dividends paid

In real markets, several companies pay dividends to their equityholders. However there are extensions of original BS model which include dividends in options evaluation. A common way is to discount value of a future dividend and subtract it from the stock price.

Figure 2.4: Volatility skew

European Options

The Black–Scholes formula is designed to evaluate only the price of options which can be exercised only at expiration date. This formula is not useful for American options pricing.

Although the Black–Scholes formula represents the holy grail of investors, the explanations above show that it does not work in a real market. Rely completely on a mathematical pricing model can lead to an excessive exposure to risk. A proof of this was the financial crash of 2008–2009, attributed to the housing bubble, which could not be accounted in the Black–Scholes model. The history teaches that the formula is useful if used wisely and abandoned when the market conditions are not even remotely appropriate. In absence of large fluctuations in the stock market, the model performs well; however large fluctuations in real markets are more frequent than Brownian motion predicts. Black–Scholes may have contributes to the financial crisis in 2009, but only because it has been used incorrectly and under wrong conditions. This means that mathematical models are not improper tools in financial world, rather they are the key for understanding constantly evolving markets, with a variety of instruments which become more and more complex, as options. The aim is to optimize Black–Scholes model, so that it is also valid in conditions closer to the reality of financial markets.

Chapter 3

Agent Based Models

The ability to manage the heterogeneity of the agents involved and gather the emergence of the property of self–organization by agents are two typical features of Agent–based models (ABMs) which cannot be found in other simulation techniques. For these and other reasons, Agent–based modelling is a powerful simulation technique that has had a myriad of applications in the last few years, including applications to biology, business, technology, social science, network theory and real–world financial problems. The brilliancy of this technique is the combination of elements of complex systems, multi–agent systems, game theory, emergence, computational sociology, and evolutionary programming, which permit to simulate an artificial–real situation. Effectively, ABMs are a kind of micro–scale model that simulate the simultaneous operations and interactions of multiple agents in an attempt to recreate and predict the appearance of complex phenomena [10].

One of the key features, which appears to be one of strengths of this methodology, is the bottom–up approach. Considering the interactions of the individual elements, this method attempts to determine the emerging characteristics produced by these interactions. These features are not explicit of the system as a whole, but emerge from the group of agents which act in a separate way, oriented to the achievement of a specific objective and interact in a shared environment. According to Marceau [11] the emphasis on the interactions between agents and their environment is what distinguishes the Agent–based model from other forms of modelling systems.

Robert Axelrod [12] emphasizes that the aim of ABMs is to enrich the understanding of fundamental processes which can appear in a variety of applications. This requires the "Keep It Simple, Stupid" (K.I.S.S.) principle, enunciated by the aircraft engineer Kelly Johnson, as an essential concept in ABM theory. The K.I.S.S. is a designed rule which states that many systems work best when they have simple designs rather than complex ones. This principle allows to solve complex problems using fewer lines of simple code and makes the model more flexible,

easier to extend and modify when new requirements arrive. In particular, it is necessary consider the K.I.S.S. principle when dealing with interactions between agents. Indeed, in this case this principle is also associated to the fact that complex behaviours emerge from simple interactions. In this sense (and not only) this concept is widely used in the ABMs.

Components of ABMs

According to a first definition by Ferber [13] (of Multy–agent systems but also adaptable for ABMs) ABMs are composed of five fundamental elements:

- an *environment*, which represents the space of interaction between entities of the model;
- a set of *objects*, with different characteristics, associated to the environment;
- a set of *agents*, which represents a subset of the objects and are the active entities of the system;
- a set of *relationships*, which link together objects and agents;
- a set of *operations*, which make agents able to perceive, produce, process and manipulate objects;

A second definition given by Macal and North [14] reduce to three the number of components of ABMs and focus on the concept of complex adaptive system [1]. These three basic components are:

- a set of *agents* with individual characteristics;
- a set of *relationships*, which define interactions between agents;
- an *environment*, in which (and with which) agents interact;

Although the two definitions are not identical, they agree on the presence of *agents*, *environment* and *interactions* between agents themselves and environment.

[1] Complex adaptive systems are complex systems composed of intelligent and autonomous agents which interact each other. In complex adaptive systems the agents are able to learn and adapt to the variation of the system.

Agents

As mentioned before, in Agent–based simulations the model is composed by a collection of autonomous and intelligent decision–making entities able to learn, adapt and evolve, named agents. For definition of agent consider Jennings and Wooldridge [15]: *An agent is an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives.* Agents may have various individual features, which are typical of the system they represent; in particular, they can be designed using concepts that are normally applied to the description of human beings, such as intelligence and emotions. In this sense, agents may show properties as cooperation and learning, highly important concepts in social science simulation. Jennings and Wooldridge define agents as autonomous entities. This assumption is led by the fact that agents operate without the direct intervention of researchers and have some sort of control over their own actions and internal state.

Therefore, the agent, in the context of the social simulation, is a software object in a virtual environment, with rules that allow interactions with the environment and other agents. The autonomy of the agents implies that any situation faced in the model must be solved using the rules implemented during the model building stage. Notice that human operator has the task of observing and controlling, not driving: the agent reacts to external stimuli from other agents (simulating social interactions) and from the environment. In addition, the agents usually behave in order to reach a goal. The definition of rules and the autonomy of the agents ensure that the effects observed at the aggregate level are actually produced by the single–agent rules.

Environment

The environment in ABMs is the virtual world in which the agents act and interacts. Usually, the environment represents the geographical space of the simulation, for instance, if the model simulates the traffic in a city the environment will be the map of the road of that city, and in a model of migrations the environment will show states and nations. In other models the environment is simply the Cartesian coordinate system in which agents have coordinates to specify their locations. Another option is to have no spatial representation (geographical or Cartesian), but only links which connect agents to other agents in a network. In this case the only indication about relationships of an agent with other agents is the list of agents to which it is connected.

While agents play a key role in simulation, it must also be noted that also the environment plays a prominent role because [16]:

- it deeply influences the behaviours of agents, as they interact also with environment;
- the aim of the simulation is to observe some aggregate level behaviour, that can actually only be observed in the environment.

Interactions

Interaction is a key concept in ABMs. The overall dynamics of the systems is not defined in terms of a global function, but rather the result of individual actions and interactions of autonomous agents [16]. Indeed, emergent phenomena are the results of these interactions between independent and intelligent entities which operate according to different modes of interaction.

It is obvious that the design of an interaction model largely depends on the model to be simulated and has a huge impact on the definition of agents themselves.

Finally, notice that the interactions can be direct, for instance exchanging messages, or indirect; in the last case the model has to provide the creation of artefacts which represent a media for the agents.

In conclusion, one can state that the ABM simulation is based on agents which repetitively make individual decisions and interact with each other according to their own characteristics, in order to simulate a simil–reality and explore the complex dynamics out of the reach of pure mathematical methods. Indeed, even if an AMB is a simple simulation of agents and their behaviours, it can exhibit complex interactions pattern and provide information about real–world dynamics.

3.1 Benefits of ABM

The usage of computational agent models arises when mathematical models can be written down but not completely solved. In this case the ABM brought an important contribution to the solution of these class of problems, for which writing down equations is not a suitable answer [17].

Furthermore, there are various equation–based models (EBMs) which are largely unable to capture the algorithmic nature of behavioural data and identify the casual relationship [18]. EBMs and ABMs are both simulation techniques, which differ in the implementation of the model and how it is executed. In ABM, the model is made up by a set of agents with properties and behaviours of the different individuals which constitute the system; the simulation has the aim to emulate these behaviours and find emergent phenomena as results. In EBM, the model is a set of equations and the execution consists of solving them [19]. It is easy to understand that, as mentioned before, not all sets of equations can be solved and,

when this is possible, it is common that the theoretical results are not congruent with the data. Finally, in such circumstances, the agent–based computational models may be the only method available to explore and solve certain processes.

In addition to the strong motivation previously explained, Agent–based simulation has become increasingly popular as a modelling approach in the social sciences also because of four main benefits:

- ABM captures emergent phenomena
- AMB allows heterogeneity
- ABM is flexible
- ABM provides a natural description of a system

ABM captures emergent phenomena An emergent behaviour or emergent phenomenon can appear when a number of simple entities interact each other in an environment, forming more complex behaviours as a collective. An emergent phenomenon can have properties that are inconsistent from the properties of the single parties. For example, in real estate market the spontaneous formation of a group of people who want to sell—who are therefore in competition to seek buyers—can lead to a collapse of the market and a consequent vertiginous fall in prices in a short time. An analogous collective phenomenon is the one concerning commuters who struggle for a bit of space on the same road at the same time. The consequence of that is a traffic jam, which is the equivalent, in terms of traffic, to the collapse of the market [20]. Consequently, this property of emergent phenomena makes them difficult to understand and predict: emergent phenomena can be largely counterintuitive.

By its nature, ABM is the standard approach to model emergent phenomena: ABM simulates the behaviour of the agents constituting the system and their interactions, capturing emergence from the bottom up when the simulation is executed. Usually, emergent phenomena appear under specific conditions. For instance, they come out when individual behaviours are non–linearity and exhibits memory, path–dependence, hysteresis, non–markovian behaviour or temporal correlations, including learning and adaptation and when agent interactions are heterogeneous and can generate network effects [21].

Interestingly, since ABM produces emergent phenomena from the bottom up, it raises the question of what could be an explanation of such phenomena. According to Epstein and Axtell [22], «[ABM] may change the way we think about explanation in the social sciences. What constitutes an explanation of an observed social phenomenon? Perhaps one day people will interpret the question, 'Can you explain it?'as asking 'Can you grow it?'.»

ABM allows heterogeneity As mentioned in the section above, ABMs may represent systems of heterogeneous agents. Each agent owns different characteristics which define its own behaviour. The behaviour is also determined by the ability of the agents to find and process information; how the information is processed depends on how each single agent has been designed in order to interpret the world around it and its own past experiences.

Heterogeneity is a property that makes ABMs particularly attractive because these models succeed to simulate situations very close to reality, since real actors operate according to their own preferences, experience and features.

ABM is flexible The flexibility in modelling provided by ABMs allows replication of phenomena of interest with a higher degree of realism than in other traditional simulation technique. The flexibility in AMBs can be observed along several dimensions. For instance, the individual characteristics of an agent may be modified in every moment during the simulation; such as the ability to evolve or learn, the behaviour and the degree of interaction. Furthermore, the operator can decide the level of description and aggregation: one can decide to execute the simulation with all agents, a subgroup of them or only one agent.

ABM provides a natural description of the system ABM is mainly used for describing and simulating systems composed of behavioural entities. For the many qualities already listed, whether one is attempting to simulate a traffic jam, the stock market, voters or how an organization works, ABM makes the model closer to reality [21]. Unlike simulations based on equations, ABMs are based on the employment of decision–making agents with individual properties, which act and interact in, and with, a suitable environment. These features not only make the model simplest to execute, but make the simulation significantly closer to the real world. Therefore, as stated several times, ABM is often the most appropriate method of describing what is actually happening in reality.

3.2 Complexity, emergence and ABMs

The definition of complexity of a system is itself complex. Despite the complex systems are one of the most popular and growing field in recent years, it can be very difficult to find a single definition for the concept of complexity. Commonly, the scientific community refers to two fundamentals manifests for the beginning of the definition and the method of complex science. The first one is mostly identified with the paper *Science and Complexity* by Weaver [23], published in 1948. In this work, Weaver clearly outlined the field of interest of the complexity, splitting the scientific problems into three categories:

- *Problems of Simplicity*, they are those that Weaver defines as problems with two variables. The typical example is a billiard table with two balls; classical mechanics, given the initial conditions, is able to predict with absolute precision the position and speed of the balls after the shot.

- *Problems of Disorganized Complexity*, if in the previous example the balls become billions or hundreds of billions the classical mechanics it is no longer able to say anything because of the enormous number of elements. But if the motion of the elements is disorganized, or completely random, as the molecules of a gas, then the statistical mechanics developed by Gibbs and Boltzmann can effectively describe the system on the basis of average values, such as pressure and temperature.

- *Problems of Organized Complexity*, is the intermediate between the previous case, where the number of elements continues to be too high to be solved by classical mechanics, and can not be treated statistically as it is not random.

With the following words, Weaver explains his thinking regarding this last class of problems and the importance that these have on the understanding of fundamental phenomena in the medical, biological, physical, political and economic fields.

> These problems–and a wide range of similar problems in the biological, medical, psychological, economic, and political sciences–are just too complicated to yield to the old nineteenth century techniques which were so dramatically successful on two-, three-, or four-variable problems of simplicity. These new problems, moreover, cannot be handled with the statistical techniques so effective in describing average behaviour in problems of disorganized complexity. These new problems, and the future of the world depends on many of them, requires science to make a third great advance, an advance that must be even greater than the nineteenth century conquest of problems of simplicity or the twentieth century victory over problems of disorganized complexity. Science must, over the next 50 years, learn to deal with these problems of organized complexity.

The birth of modern science in the seventeenth century is identified with a drastic and initially successful choice: the study of nature as an organic whole was abandoned in order to focus on simple, quantifiable phenomena, isolating them from the rest. This new method proceeded by disassembling a complex mechanism, and reducing it in many parts, small enough to be well understood in their evolutionary processes and described by simple mathematical laws. This methodological approach, which is named "reductionism", is at the origin of the

most impressive advances in the history of the knowledge of nature: in little more than two centuries were discovered the laws of gravity that govern the motion of the planets, the laws of thermodynamics, which allowed the construction of the internal combustion engine, the equations of electromagnetism, optics and modern telecommunications, and finally, at the beginning of the next century, relativity and quantum physics which disclosed the deepest behaviour of matter, but also the history of the universe from the big–bang to the present day.

These unbelievable successes however led to distort cognitive perspective: only the simple broken parts were considered important and fundamental, ignoring the complex system from which they were isolated. From winning methodology, reductionism gradually was no longer the basis of many scientific investigations and lost its importance.

After the Second World War, precisely in 1972 Anderson, who five years later would win the Nobel Prize for Physics for his fundamental contributions to the theoretical understanding of the electronic structure of magnetic and disordered systems, published the second article intended to become a manifesto of the theory of complexity, entitled *More is different* [24]. In his article, Anderson supported completely innovative theories, in particular he exhibited a fierce criticism against the reductionist method:

> The reductionist hypothesis may still be a topic for controversy among philosophers, but among the great majority of active scientists I think it is accepted without question.
>
> [...] The main fallacy in this kind of thinking is that the reductionist hypothesis does not by any means imply a "constructionist" one: The ability to reduce everything to simple fundamental laws does not imply the ability to start from those laws and reconstruct the universe.
>
> [...] The constructionist hypothesis breaks down when confronted with the twin difficulties of scale and complexity. The behaviour of large and complex aggregates of elementary particles, it turns out, is not to be understood in terms of a simple extrapolation of the properties of a few particles. Instead, at each level of complexity entirely new properties appear, and the understanding of the new behaviours requires research which I think is as fundamental in its nature as any other.

Furthermore, from this extract, one realises that according to Anderson the complexity is a phenomenon that is generated as a result of the interactions between agents, which act in an environment as part of a whole. Indeed, one difference between "simple" systems and complex systems, is closely linked to the way in which the constituent entities of complex systems interact with each other and with their environment.

At this stage, it could be expected one effective definitions of what complexity might mean. Unfortunately, although a single universal definition of complex systems does not yet exist, it is useful to list the key features which these systems may have:

- *Nonlinearity*, internal processes and interactions between inputs and outputs are non–linear. This means that a small perturbation in the input or internal to the system can produce several types of effects, for example, one large (such as the butterfly effect), one proportional to perturbation or no effect to the limit. On the contrary, in linear systems the effect is always directly proportional to the cause.

- *Emergence*, complex systems may show emergent behaviours, that is even if the results can be deterministic, they may exhibit properties that are comprehensible only to a higher level. Certainly, one must state that emergence is necessary for complex systems: if a system does not exhibit emergent phenomena it is not complex. However, emergence is not a sufficient condition in order that a system is complex.

- *Feedback loops*, complex systems always contains circular process in which the output of the system is returned or "fed back" to the system as input. System have negative feedback loops (for damping and stabilization) and positive loops (for amplifying). Effects of the behaviour of an element of the system are reintroduced in such a way that the same element is altered. In general negative rings are essential for the stabilization of the system and for its homoeostasis, that is for its existence.

- *Robustness*, in complex system the order is robust, because under perturbation the system remains stable. More precisely, a system is said robust if final consequences of an accidental event are limited in proportion to the severity of the event itself. Notice that robustness may be explained in computational language as the property of a system to correct errors in its structure.

- *Memory*, the story of a complex system can be essential since they are dynamic systems, which change over time and internal passed states can have influence on present states, for instance, they may show hysteresis.

- *Numerosity*, complex systems consist of a large number of elements. When the number of these elements is relatively small, the behaviour can be explained using classical methods. However, when the system reaches a huge quantity of elements, the classical theories are no longer able to give an adequate explanation and it is necessary to use the theory of complexity for the resolution of these problems. Notice that the title of the work of Anderson,

More is Different, alludes to the fact that many more than a handful of single elements need to interact in order to create a complex system.

- *Open boundaries*, usually complex systems are open systems, that is they interact with their environment. In fact, it may be difficult to define the boundaries of a complex system.

- *Hierarchical organisation*, complex systems do not have central control systems; they must be dynamic and adaptable, not rigid or invariable. Consequently the notion of hierarchy is resisted [25]: complex systems consist of many levels of organization that can be thought of as forming a hierarchy of systems and sub–systems. Emergent phenomena occurs when order that arises from interactions among parts at a lower level is robust. It should be noted that such robustness exists only within a particular regime. The ultimate result of all the properties of a complex system is an entity that is arranged into a variety of levels of structure and features which interact with the level above and below and exhibit lawlike and causal regularities, and various kinds of symmetry, order and periodic behaviour [26].

- *Spontaneous order (self–organization)*, the behaviour of a complex system is not easily deducible from the simple sum of the behaviours of individual elements, but comes from the self–organization and reallocation of the global information and complex interactive reaction mechanisms and internal feedback. This means that the order in a complex system emerges spontaneously from the aggregation of a huge number of interactions between elements. Notice that spontaneous orders are to be distinguished from organizations: spontaneous orders are scale–free networks, while organizations are hierarchical networks. In addition, organization may be a pat of spontaneous order (reverse is not true). A complex system necessarily has to exhibit some kind of spontaneous order.

Again considering the work of Anderson [24], he state that complexity arises when agents act and interact, as a part of a whole, and the number of agents is relevant. Therefore, in order to see through the lens of complexity, one needs easily implementable models able to reproduce a system composed of many elements, whose interactions with each other and the environment have as result emergent phenomena. As a result of the above considerations and properties analysed previously, it is clear that ABMs are the ideal technique for the study of complex phenomena.

3.3 Financial Markets and ABM

The need to employ ABMs also in economic analysis, came up against the difficulty of mathematical formalization of the economy and the inadequacy of standard economic models [18]. According to the possibility of introducing new tools for the analysis of macroeconomic problems, the former president of the European Central Bank, Jean–Claude Trichet during the Central Banking Conference in Frankfurt on 18th December 2010 enunciated the following words:

> When the crisis came, the serious limitations of existing economic and financial models immediately became apparent. Arbitrage broke down in many market segments, as markets froze and market participants were gripped by panic. Macro models failed to predict the crisis and seemed incapable of explaining what was happening to the economy in a convincing manner. As a policy-maker during the crisis, I found the available models of limited help. In fact, I would go further: in the face of the crisis, we felt abandoned by conventional tools. In the absence of clear guidance from existing analytical frameworks, policy–makers had to place particular reliance on our experience. Judgement and experience inevitably played a key role.
>
> [...] The atomistic, optimizing agents underlying existing models do not capture behavior during a crisis period. We need to deal better with heterogeneity across agents and the interaction among those heterogeneous agents. We need to entertain alternative motivations for economic choices. Behavioral economics draws on psychology to explain decisions made in crisis circumstances. Agent-based modeling dispenses with the optimization assumption and allows for more complex interactions between agents. Such approaches are worthy of our attention.

Triches referred explicitly to ABMs as new technique able to simulate the heterogeneity of the agents involved in the economic system and the interactions among them. The encouragement of the employment of ABMs is based on several successful applications of ABMs in financial markets.

After the pioneering Monte Carlo simulations such as those by Stigler [28] or by Cohen et al. [29], the first Agent–based model is the one proposed by Kim and Markowitz [30] in 1989. The Kim–Markowitz Agent–based model was designed to investigate the stock market crash on 17th October 1987, when the US stock market decreased by more than twenty per cent. Many economists of that time believed that the explanation for this crash was the automatic overreaction of computer–based "dynamic hedging" strategies, become popular among institutional investors in the years before. However, such models including market interactions

between a huge number of investors following these strategies are obviously difficult to solve using analytical methods. Consequently, Kim and Markowitz tried to explore the the dynamic of hedging strategies via a complicated model of price formation in an artificial financial market. They investigated the relationship between the volatility of the market and the share of agents with portfolio insurance strategies [31].

The portfolio insurers model by Kim and Markovitz, involved two group of individual investors: "rebalancers" and "insurers". The rebalancers aim to keep a constant composition of their portfolio, while the insurers follow appropriate strategies to insure that their eventual losses will not exceed a certain fraction of the investment per time period. The latter investors use the constant proportion portfolio insurance (CPPI) method by Black and Jones.

A rebalancer acts to keep one–half of his wealth in stocks and the other half in cash. If the stock price increases, then the stock weight in the portfolio will rise and the rebalance investor will sell shares in order to restore the starting proportion. In the case of falling stock price, the value of the stock in the portfolio will decrease and the investor will buy shares until the stock again constitutes the fifty percent of the portfolio. Therefore, the rebalances have a stabilizing influence on the market, selling when the stock price rises and buying when stock price falls.

A CPPI investor has as main objective not to lose more then a presetted percentage (e.g 25%) of their initial wealth in a time period (e.g. 65 trading days). Therefore, he has to insure that at every cycle the 75% of his wealth is out of risk. The investor assumes the stock price will not decrease in a day more than a certain quantity, for example 2. Consequently, he always keep in stock twice the difference between the current wealth and the 75% of the initial wealth. This means that the amount of the insurer investor is bidding or offering at each cycle. When the stock price decreases, the amount he want to keep will decrease and he will sell, destabilizing the market. When the stock price rise, the amount he want to keep will increase and he will buy, supporting a public bubble [32]. Therefore, contrary to the rebalancing strategy, the CPPI strategy implies a pro–cyclical and thus potentially destabilizing investments behaviour.

The simulation reveal that the reason of the market instability has roots on individual efforts of insurers to cut their losses by selling when the stock price falls down. Therefore, it has been proved that even a small number of CPPI agents can destabilize the market and causes a market crush. Finally, one can say that the reason of the US market crash in 1987 has been investigated and found using one of the first Agent–based simulations.

After the succeed of the Kim–Markowitz portfolio insurer model, many other economists employed ABMs as simulation technique to investigate of abnormal economic phenomena or easily to predict the market trend.

Arthur, Holland, Lebaron, Palmer and Tyler [33] implemented an ABM which is focused on the concept of co–evolution. In this model, each agent manages its investments in order to maximally exploit the market dynamics generated by the strategies of other agents. This situation leads to an "ever–evolving" artificial market, driven endogenously by the continuous changes of investors. In this case, this Agent–based model aim at proving that market fluctuations can be induced this endogenous co–evolution, rather then by exogenous events [32].

Even the brilliant success of the network theory struck positively the economists which introduced new tools for analyses of financial modelling. In particular, they focused on information and communication between investors, random–field and percolation models, which were considered important and innovative tools for the investigation of information transmission or formation of opinions among agents.

Cont and Bouchaud applied the percolation theory to financial market, implementing an ABM able to simulate a stock market in which each agent is connected to any other agent with a certain probability. These links could be considered as communication channels through which agents exchange information.

In the model, all agents that are connected to each other form a coalition, which acts in unison: the members of one coalition all may buy, sell a unit of stock or stay inactive. The connection probability is $p = c/N$ where N is the total number of agent and c is the percolation factor which control the willingness of agents to form cluster, that may be interpreted as mutual founds or specific trading strategies followed by investors in real market.

Whether the value of factor c is (slightly) less then one the distribution of the cluster size is cut off by an exponential tail, while at the percolation threshold $c = 1$, the probability density for the distribution of the cluster size decreases asymptotically as a power law. Usually, in a naive market model, given random demand and supply of stock, the asset return would most likely have a normal distribution. The endogenous formation of trading clusters when c is less then one, however, mimics non–sequential herding process through communication process. As the analytically derived asset price shows a distribution characterised by fat tails and excess of kurtosis as in real stock return, the Agent–based model of Cont and Bouchaud proposes that in financial market the herding behaviour is responsible for the stylized facts [34].

During years, many other models have been developed by economists and econophysicists in order to create and simulate an artificial financial market and investigate emergent phenomena. For example, Lux and Marchesi [35] developed a "terribly complicated" Agent–based model to endogenously explain the heavy tail distribution of asset returns and clustering volatility.

An option market by Tran, Pham–Hi and Duong [36] has been simulated using Agent–based models. They modelled an artificial European option market under

unknown volatility with liquidity costs, proving that both circumstances have significant effect in pricing bias. More precisely, the model suggests that option prices increase slightly when liquidity cost increases and pricing bias tends to decrease as the drift increasing in case of low volatility.

In conclusion, one may state that Agent–based simulation models have enriched economic and financial systems, succeeding to go beyond limits of traditional economic methods.

3.4 NetLogo

NetLogo is a platform for agent-based modelling, specifically for simulating natural and social phenomena. It can capture and reproduce the dynamic aspect of these phenomena, allowing the interactions between thousands of independent and heterogeneous agents, which operate in parallel. This make it possible to explore the micro-level connections and the macro–level consequences that arise from interactions.

3.4.1 History

NetLogo was designed by Uri Wilensky, who founded and directs the Centre for Connected Learning and Computer-Based Modelling at Northwestern University. NetLogo, as its rival StrarLogo, born in the spirit of the programming language Logo (from the Greek *logos* meaning word or thought), a learning tools written in 1967 by Daniel G. Bobrow, Wally Feurzeig, Seymour Papert and Cynthia Solomon at MIT. Logo is a multi-paradigm adaptation and dialect of Lisp, a functional programming language. The goal of Logo was to create a maths land where kids could play with words and sentences. Indeed, it is designed to have a "low threshold and no ceiling": it is accessible to novices to learn maths and other discipline but also to expert users for supporting complex explorations and sophisticated projects.

In 1969 was created the first working Logo turtle robot, originally a robotic entity that sat on the floor and could be directed to move around by typing commands at the computer. The ″turtle‟name of the agents derives from the fact that in earlier models these agents were moving very slowly, just like the turtles. Modern turtles, instead, are faster, can change shape like fish, sheep, cars, persons, etc. and are used to draw shapes, designs, and pictures on the computer graphic screen.

3.4.2 How NetLogo Works

As previously mentioned, NetLogo is the most widely used software for Agent–based model simulations. There are several significant reasons for choosing NetLogo: first of all its pseudo–code was designed to be easily readable, which makes it a great tool for learnability; despite its apparent simplicity, NetLogo provides methods to simulate complex social systems, as traffic, stock markets, epidemic spread, etc. and to explore the causes of the emerging phenomena. What creates a trade-off between the ease of implementation and the power of the simulations is the Java core. Indeed, NetLogo is an open–source software platform programmed in Java and Scala.

Many scientific papers have been published using NetLogo in a variety of domains, such as chemistry, economics, biology, physics, dynamic systems, psychology, and networks theory.

When the program is opened, the display shows a window with three labels: Interface, Info and Code, as it can be seen in Figure 3.1. In the Interface section is present a black square populated by many agents interacting each other, which is the NetLogo world. Next to the agent–world, there are several other buttons, sliders, switchers, etc., which allow the user to start or stop the simulation, insert inputs, decided the pattern and the number of agents and other data. The interface shows also plots, which monitor the evolutions of some variables. The Info section explains what the model tries to reproduce and how it works, namely it provides all the information that the user needs. Finally, in the Code section it can be found the pseudo–code.

3.4.3 Agents in NetLogo

The basic entities in NetLogo are the agents. Agents are being that can follow the instructions and that can communicate and interact with each other. There are four different types of agents: observer, patches, turtles, and links.

The observer, unique in the model, is the global instance which provides global variables, manages and gives instructions to the other agents. It does not have a location: one can imagine it as looking out over the world of turtles, links and patches.

The patches are fixed square pieces which all together form the the grid of the NetLogo's world, over which turtles can move. Patches have pre-defined Cartesian coordinates, colour and label. However, patches are, like other agents, programmable: the user can change their colour and add further variables to them. In a Netlogo program there is not the possibility to assign different features to different patches, namely there are not patches that act in different way. As previously noted, patches have coordinates: the patch at (0,0) is named the ″origin‟and the

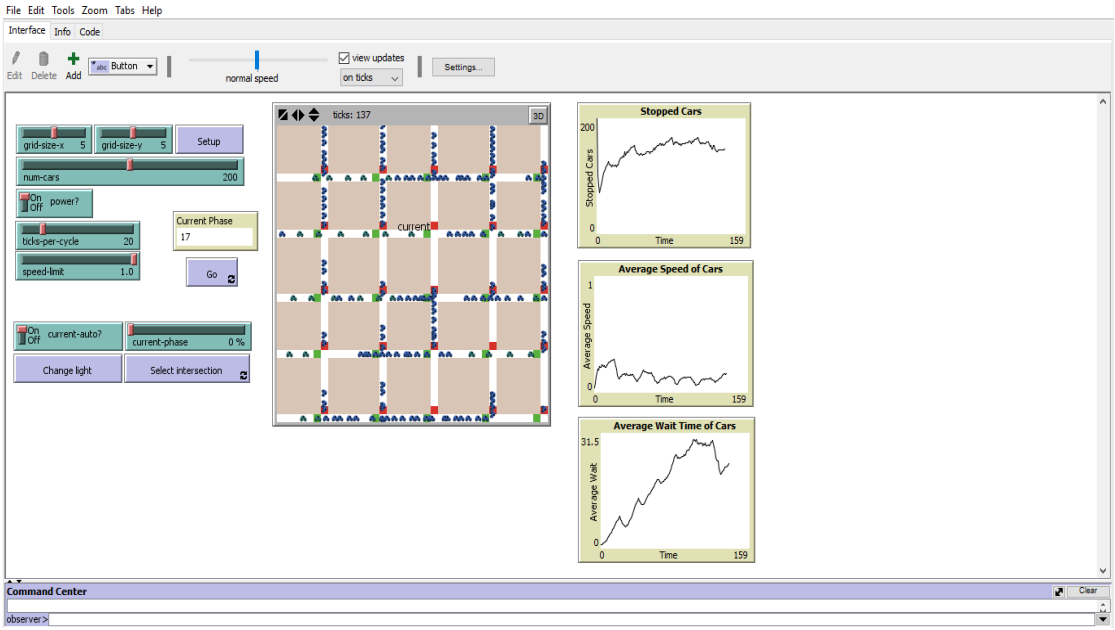


Figure 3.1: Interface of the *Traffic grid* model (Netlogo's Models Library)

coordinates of the other patches are the horizontal and vertical distances from this one. The name of the patch's coordinates are *pxcor* and *pycor* and just like in the standard Cartesian plane the *pxcor* increases to the right and *pycor* increases upwards. The patches coordinates range from -16 to 16 and this means that there are 1089 patches total in the grid.

Contrary to the patches, turtles are mobile agents and they can move on the grid in continuous space within the world defined by the patches. Turtles do not exist when the program starts up: they are created by the user or by the patches too. Turtles have Cartesian coordinates (*xcor*,*ycor*), but if patch's coordinates are always integers, turtle's coordinates can have decimals. This because any turtle can stay in any point of the grid, not only in the centre of the patch. This type of agent has pre–defined variables, as colour, shape, label, coordinates, etc. but the user can modify them. In fact it is possible to assign different shape as person, plane, bee, flower, etc. to the turtle–agents and also the user can design new shapes with the tool *Turtle shape editor*. It is possible to declare different types of turtles, called *breeds*. Breeds inherit all variables of the turtles, but can have additional own variables. Supplementary attributes can be defined as follows:

```
turtles-own [ energy age money ]
```

In the same manner it is possible assign extra variables to the patches, by changing the command ″turtles–own‟in ″patches–own‟.

Finally, the links are agents with no coordinates which connect two turtles. Links have two ends and they can be direct (from one turtle to another turtle) or undirected (one turtle with another turtle). If either turtle dies, the link dies too. A link is represented visually as a line connecting the two turtles.

Chapter 4

Artificial Neural Networks

Artificial Neural Networks (ANNs) are mathematical models designed to simulate the way in which the human brain processes information and then to solve "easy–for—a–human, difficult–for–a–machine" problems. ANNs can be employed to approximate any complex functional relationship. Unlike, generalised linear models, it is not necessary to previously define the relationship between the input and output variables. This property makes neural networks a useful and efficient statistical tool.

Human brain is a biological network of fully interconnected neurons, which continuously transmit elaborate patterns of electrical signals. The activity of a biological neural network is produced by this exchange of electrical signals between neurons. The structure of connections which allow communications between neurons are named *synapses*.

The central nervous system consists of nervous cells, called neurons, able to perform very simple processes; the intelligent behaviour emerges from the great number of elements and connections between cells. A neuron is an electrically excitable cell which consist of a *soma*, the body cell, *dendrites*, short extensions around cell body, and *axons*, long extensions which connects neurons. The cell body has a nucleus containing informations on hereditary characteristics, while the cell membrane contains some electrochemical pumps (sodium–potassium pumps) which balance the charge inside and outside the membrane. Dendrites receive signals (impulses) through the synapses of other neurons. Signals can be inputs from sensory organs or stimuli from external environment. The cell body process these impulses over time. If the signal received is strong enough (surpasses a certain threshold), the neuron is activated and the cell body turns the processed value into an output that is sent to other neurons through axon and synapses.

The complexity of biological neural networks is only partially reproduced in artificial neural networks. Artificial neurons consist of inputs, that are multiplied by weights, conveniently evaluated through learning algorithm, and then calcu-

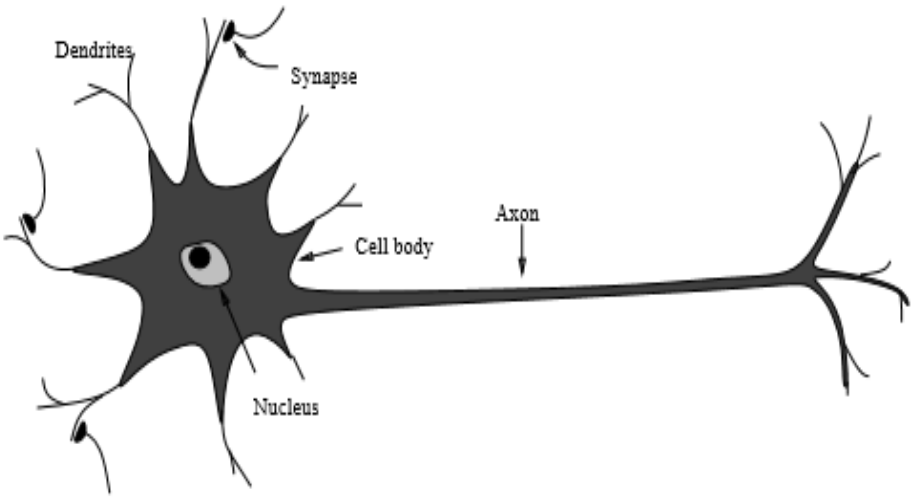


Figure 4.1: A sketch of a biological neuron [37]

lated by a mathematical function which determine if neurons are excited or not. Another function (which can be the same) determines the output of the artificial neural networks. Artificial neural networks combine artificial neurons with the aim to process information and solve problems. Therefore, the power of neural computations derives from connections of artificial neurons in a fully connected network.

4.1 McCulloch-Pitts Neuron Model

In 1943, neuropsychiatrist Warren McCulloch and mathematician Walter Pitts [38] showed one of the first applications of logical calculus to the elements of a biological system. Their study was the first comprehensive analysis in the field of neural networks, which showed how simple units with excitatory and inhibitory synapses with specific threshold are able, by virtue of a collective process, to represent complex sentences. Constructing a Boolean logic, McCulloch and Pitts succeed to reproduce neural events and their relations, based on the "all–or–none" character of nervous activity [39].

The calculus of McCulloch and Pitts was based on the following assumption:

- The activity of the neuron is an "all–or–none" process. This means that the neuron can be excited or unexcited.
- The structure of the neural network remain the same during time.

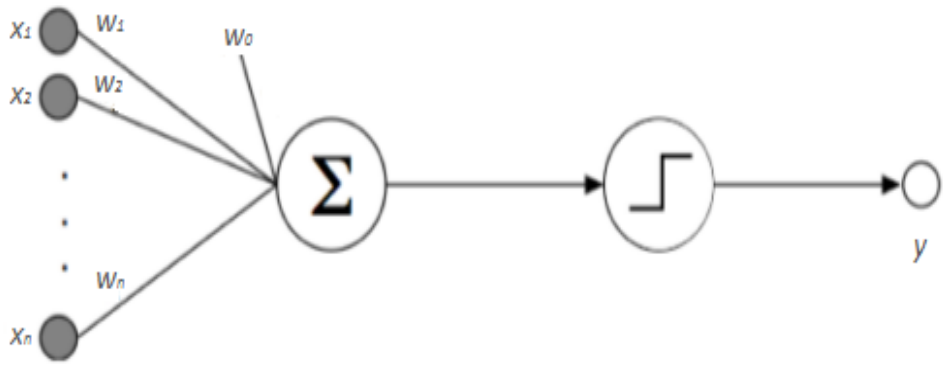


Figure 4.2: McCulloch-Pitts artificial neuron

- During period of latent addition, a certain number of synapses must be exercised in order to excite the neuron at any time excite; this number does not depend on previous activity and position of the neuron.
- The synaptic delay is the only significant delay in the neural networks.
- The inhibitory activity of the synapses prevents neuron excitation at any time.

The binary model that the two scientist design was called the "formal neuron". This artificial neuron accepts multiple binary data as input (representing dendrites) and returns a single output (representing a neuron's axon); an appropriate number of such elements, connected with the aim to form a network, is able to calculate simple Boolean functions.

The artificial neuron reported in Figure 4.2 has n inputs $(x_1, x_2, ..., x_n)$, each one connected to the neuron with a link. These ones assign to each input a weight $(w_1, w_2, ..., w_n)$: in this context, weights corresponds to the synaptic connections in biological systems. Inputs and weights are real value; if the weight is positive the connection is excitatory, if the weights is negative the connection is inhibitory. Usually, the biological threshold is represented by θ; however, if θ is positive, the threshold is referred as bias[1]. For simplicity, one takes $w_0 = \theta$ with $x_0 = 1$.

[1]The bias w_0 is a constant value which can be identified as the activation threshold (it is the threshold that the weighted sum has to exceed in order to activate the neuron). Formally, the bias has the same role as weights that function as controllers of the intensity of the emitted or received signal.

$$a = \sum_{i:1}^{n} (w_i \, x_i) + \theta = \sum_{i:0}^{n} w_i \, x_i \tag{4.1}$$

The weighted sum is passed through a non–linear function, named activation function[2]. The activation function is usually a sigmoid function, but can also be a step function or sign function (g). The output y of the neuron depends on its activation, referring to the firing frequency of the biological neurons:

$$y = g(a) = g(\sum_{i:0}^{n} w_i \, x_i) \tag{4.2}$$

Despite its simple structure, the McCullogh–Pitts neuron is a powerful computational tools and is the fundamental element to develop artificial neural networks.

4.2 Rosenblatt Perceptron

Following the studies of McCullogh and Pitts, in 1957 Frank Rosenblatt [41] developed the first neural network, known as "Perceptron". It is essentially a network of interconnected artificial neurons arranged in input and output layers. The Perceptron is a feedforward neural network because there are no connections between neurons that form node.

This one is considered a binary classifier, which can predict if an input, represented by a vector of number, belongs to a class or another. In other words, it is a function that maps inputs x_i into a binary output y.

$$y = g(x) = \begin{cases} 1 & if \ \sum_{i:1}^{n} w_i \, x_i + w_0 \ > \ 0 \\ 0 & otherwise \end{cases} \tag{4.3}$$

This type of architecture has the big limit to solve only linearly separable problems: the original Perceptron can only compute a line (or a plane) which divide the space into two regions. In general, the "separation line" is $\sum_{i:1}^{n} w_i \, x_i + w_0 = 0$ and it is always perpendicular to vector of weights. In order to solve more complicated and non–linear problems, one has to add one or more hidden layers to the simple Perceptron.

The activation functions used can be divided into two classes.

No–differentiable activation function:

[2]The activation function is a function that determines the output given a vector of inputs. From a biologically point of view, it would represent the rate of the action potential which fires in the cell.

- *Heaviside function*: $y(x) = \begin{cases} 0 & if\, x < 0 \\ 1 & if\, x > 0 \end{cases}$
- *Sign function*: $y(x) = \begin{cases} -1 & if\, x < 0 \\ 0 & if x = 0 \\ 1 & if\, x > 0 \end{cases}$

Differentiable activation functions:

- *Sigmoid function*: $y(x) = \dfrac{1}{1 + e^{-x}}$
- *Hyperbolic tangent function*:$y(x) = tanh(x) = \dfrac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$

Usually it is advantageous to consider the sigmoid or hyperbolic tangent functions, particularly if one wants to use the backpropagation algorithm for training (as this method requires differentiable activation functions).

The most important feature of Perceptron is that the weights are not pre–calculated as in McCulloch–Pitts neuron but are adjusted by a iterative process called training. Indeed, the crucial breakthrough that Rosenblatt brought to the field of neural networks was precisely the fact that the Perceptron can learn how to solve problems and then operate autonomously. The Perceptron learns using example (desired outputs) and modifying weights, in order to be able to understand the formula that connect input values to predetermined output values. At the end of the training process, the neural network should succeed to "generalize"; this means that given new inputs (never seen before), the Perceptron should be able to predict the output relative to them. Algorithms which employ target outputs to train the neural networks are called "supervised learning algorithms".

Perceptron learning rule

As mentioned above, Rosenblatt introduced a learning rule for training Perceptrons to solve problems of pattern recognition. He demonstrated that his learning rule always converges to the optimal weights, if weights which solve the problem exist. The training process is very simple: examples of desired outputs are presented to the neural network, which learns from its mistakes and consequently changes weights.

Consider two sets, one of desired outputs d_i^{μ} and one of actual output of the neural network y_i^{μ}, where μ is the index of patterns ($\mu : 1, p$) and i is that of output node. As every input is applied to the Perceptron, the network output is compared to the target; the learning rule then adjusts the weights and biases of

the neural network in order to move the real output closer to the target. This process is governed by the Rosenblatt Perceptron learning rule, that is:

$$\Delta w_{i,j} = \eta \, H(-d_i^\mu \, h_i^\mu) \, d_i^\mu \, x_i^\mu \tag{4.4}$$

where h_i^μ is the weighted sum of the inputs and H is the Heaviside'function. Behind the mathematical formulation, this rule states that the weights are updated only if the product $d_i^\mu \, h_i^\mu$ is negative. A positive value of this quantity means that $d_i^\mu = y_i^\mu$, which means that the convergence is obtained and then $\Delta w_{i,j} = 0$. Usually, one includes in the rule a stronger constraint, imposing $d_i^\mu \, h_i^\mu > N \, k$. Quantity N is the number of inputs and k is a fixed value. The Perceptron learning rule becomes:

$$\Delta w_{i,j} = \eta \, H(N \, k - d_i^\mu \, h_i^\mu) \, d_i^\mu \, x_i^\mu \tag{4.5}$$

Perceptron Convergence Theorem

The theorem states that if a solution exists (optimal weights), the Perceptron learning rule always converge in a finite number of steps. In order to understand this statements in mathematical terms, define:

$$D(w) \equiv \frac{1}{|w|} \min_\mu (w^T \, z^\mu) \tag{4.6}$$

where $z^\mu \equiv d^\mu \, x^\mu$. Remembering the formula of Rosenblatt learning rule, one can write that:

$$y^\mu = d^\mu \Leftrightarrow w^T \, z^\mu = d^\mu \, w^T \, x^\mu \;\; (\, i f \, w^T \, z^\mu > 0) \tag{4.7}$$

Accordingly, when $D(w) > 0$, the direction of the weights is "good" , namely for $|w|$ sufficiently larger $w^T \, z^\mu > 0$ for each pattern μ. Finally, define:

$$D_{max} = \max_w D(w) \tag{4.8}$$

If $D_{max} > 0$, exists a vector of weights which solves the problem. If $D_{max} < 0$, the problem cannot be solved.

Following the demonstration of the Perceptron Convergence Theorem.

Proof. Consider one–layer Perceptron with threshold activation function. During training process $w^T\, z^\mu \,<\, N\, k$ and thus $H(N\, k - d_i^\mu\, h_i^\mu) = 1$; the updating quantity results $\Delta w_{i,j} = \eta\, d_i^\mu\, x_j^\mu$.

Define the quantity M which represents the total number of patterns presented during training step, as:

$$M = \sum_\mu M^\mu \tag{4.9}$$

such that:

$$w = \eta \sum_\mu M^\mu z^\mu = M\, \Delta w \tag{4.10}$$

where M^μ is the number of times the μ–pattern has been presented. If one gets a superior limit for M, it means that the solution w^* is reached in a finite number of steps.

Suppose the problem can be resolvable and w^* is the vector of weights which solve the problem. The idea is to use the Schwartz's inequality (Formula (4.11)) to demonstrate the thesis.

$$|w\, w^*|^2 \,\leqslant\, |w|^2\; |w^*|^2 \tag{4.11}$$

Consequently, one has to determine the quantities $w\, w^*$ and Δw:

$$w^T\, w^* = \eta \sum_\mu M^\mu\, z^\mu\, w^* \,\geqslant\, \eta\, M(\min z^\mu\, w^*) = \eta\, M\, D(w^*)\; |w^*|$$

$$w^T\, w^* \,\geqslant\, \eta\, M\, D(w^*)\; |w^*| \tag{4.12}$$

$$\Delta w = \eta\, z^\mu = (w + \eta\, z^\mu) - w \tag{4.13}$$

$$\begin{aligned} \Delta\, |w|^2 &= (w + \eta\, z^\mu)^2 - |w|^2 \\ &= |w|^2 + \eta^2\, z^{\mu\,2} + 2\, \eta\, w\, z^\mu - |w|^2 \\ &= \eta^2\, z^{\mu\,2} + 2\, \eta\, w\, z^\mu \\ &\leqslant\, \eta^2\, z^{\mu\,2} + 2\, \eta\, k\, N \\ &= N\, \eta(\eta + 2\, k) \end{aligned} \tag{4.14}$$

Now applying the Schwart', one gets that the number of patterns M has always a superior limit:

$$M \,\leqslant\, \frac{N(\eta + 2\, k)}{\eta\, D^*(w)} \tag{4.15}$$

This implies that one–layer Perceptron with threshold activation function which is trained using the Rosenblatt learning rule, always reaches the convergence.

□

4.3 Architectures of Neural Networks

Artificial neural networks are all made up by a number of artificial interconnected neurons, in order to reproduce the biological neuronal system. However, it is possible recognize different types of networks structures (Figure 4.3).

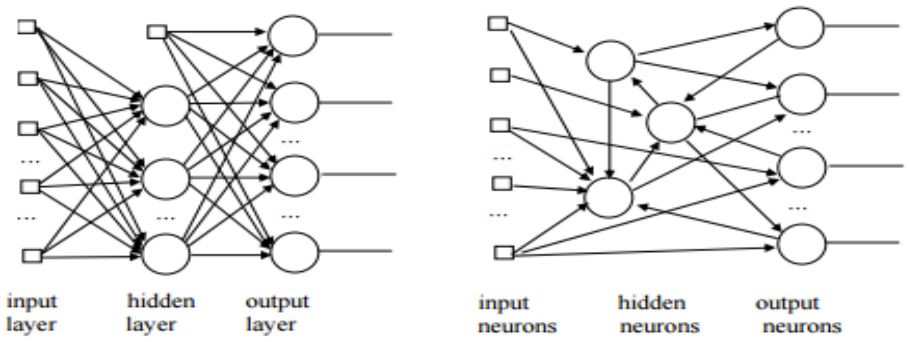


Figure 4.3: Layered feedforward neural network (right). No–layered recurrent neural network(left).

Feedforward Neural Networks

In feedforward neural networks, the artificial neurons are fully connected each others through weighted links and are arranged in layers. There must be absolutely one input layer and one output layer and possibly, according to the complexity of the problem to solve, it can add one or more hidden layers. If a network is composed only of input and output layers, then it is named single layer network; instead, if there are one or more hidden layers, it is named multilayer network. Notice that in this type of architecture, there are no connections which form cycles. This means that connections between neurons of the same layer or to neurons of previous layers are forbidden. Neurons in a layer obtain input from the previous layer and send their output to the next layer. The signal is forward propagated.

Recurrent Neural Networks

The neural networks which present cycles in their structure are called recurrent. As for the Boolean functions, the presence of cycles leads to more complex computations which involve sequences and not single patterns. Indeed, unlike feedforward networks, recurrent networks operate as a sequential system, that means they associate output pattern to input pattern, with a dependence on an internal state which, in turn, evolves in time with the submitted inputs. In general, a recurrent network does not establish a mapping between input and output patterns, but a mapping between the set formed by the initial internal state and by the sequence of input patterns with a sequence of output patterns. Indicating the internal state as $Z(t)$, one can represent the operation of the recurrent network with the two following equations:

$$Z(t+1) = f\left(X(t),\, Z(t)\right) \tag{4.16}$$

$$Y(t) = g\left(X(t)\, Z(t)\right) \tag{4.17}$$

These equations represents a dynamic system, in general no–linear. Therefore, a sequence of pattern define trajectory in its relative definition space. The most important example of recurrent network is the Hopfield neural network.

4.4 Training of Neural Networks

Optimal values of neural networks weights are determined on the basis of a training (or learning) process. Training process is a numerical optimisation problem and may be performed in two different way, according to the types of neural networks and the problem to solve.

4.4.1 Supervised Learning

Supervised algorithm is a type of machine learning that uses a set of example to train neural networks in order to enables them to solve problems. Each example, in the “training dataset", is a pair consisting of input data (typically a vector of more elements) and a target output value (the desired output).

All supervised learning algorithms are based on the assumption that if the neural network is trained with a sufficient number of examples, it will be able to create a function y which approximate the function f. If the approximation is suitable, once new inputs, never analysed before, are submitted to the neural network, the function y should be able to provide output similar to that provided by f.

The efficiency of these algorithms depends on the examples that are presented. First of all, it is necessary to verify that all inputs are correlated with the target

output, and then provide a number of examples sufficient to create the approximating function but not excessive to overtrain the neural network. An excessive training could lead to the overfitting problem: a neural network that has been overfit could loose generalisation property. Overfitting occurs when the model memorises the training inputs rather than learn to generalise over trend.

Supervised training problems can be classified into "regression" and "classification" problems. In regression problem a neural network attempts to predict continuous outputs; this means that it tries to map inputs to some continuous function. Conversely, in a classification problem a neural network attempts to predict discrete output; in other words it tries to map inputs to some discrete function.

Undoubtedly, the most common supervised learning algorithm is the backpropagation algorithm, based on the descent gradient method.

Backpropagation Algorithm

Backpropagation algorithm is the supervised training method most widely used for multilayer perceptrons. Discovery of this method in the mid–80s made possible the training for multilayer networks and laid the foundations for the next significant development of studies on neural networks over the past two decades. This algorithm works in conjunction with an optimization method such as gradient descent. Backpropagation algorithm requires outputs target, as supervised learning method, and a differentiable activation function.

In order to understand how the algorithm operates, an explanation on gradient descend method is required. Gradient descend is a first–order iterative optimisation method based on the computation, at each iteration, of the gradient of the error function with respect to the weights of the neural network. The gradient computed is used afterwards to update the values of the weights found at the previous iteration. Consider the following elements:

- *desired output*, d_i^μ
- *real output*, $y_i^\mu = g\left(\sum_j w_j\, x_j^\mu\right)$, where g is the differentiable activation function.

μ is the pattern index ($\mu : 1, p$ where p is the number of total patterns) and i indicates the respective output. Given those, one can define the "error function":

$$E(w) = \frac{1}{2} \sum_{\mu,\, i} (d_i^\mu - y_i^\mu)^2 \tag{4.18}$$

Error function measures the mean square deviation between the desired output d_i^μ and the actual output y_i^μ. The descent gradient method aims to find the optimal

weights of the neural network which minimise the error function $E(w)$. With the aim to find the minimum, one has to take steps proportional to the negative of the gradient of the function[3]. The step size is defined as:

$$\Delta w_{i,j} = -\eta \, \nabla E(w) \tag{4.19}$$

and the updating method is:

$$w_i = w_{i-1} - \eta \, \nabla E(w) \tag{4.20}$$

Usually, one begins with a vector of random weights $\bar{\mathrm{w}}$, which then is modified in the opposite direction to the gradient. The error function may be rewritten more explicitly as:

$$E(w) = \frac{1}{2} \sum_{\mu,i} \left(d_i^{\mu} - g \left(\sum_j w_{i,j} \, x_j \right) \right)^2 \tag{4.21}$$

Therefore the step to update the weights become:

$$\Delta w_{i,j} = \eta \sum_{\mu} \left(d_i^{\mu} - y_i^{\mu} \, g'(h_i) \, x_j^{\mu} \right) \tag{4.22}$$

where h_i is the weighted sum of outputs of inputs neurons. The factor η refers to the learning parameter, that is the step length through which one reaches the convergence. If the learning rate is too small, the error function decreases very slowly; if the learning rate is too large, the error may increase or oscillate [40]. Therefore, a good technique is to take η variable: initially the learning rate should be small to speed up the convergence, then smaller as it gets closer to the minimum. In case of paraboloids in n dimensions, there is the possibility to get stuck in local minima. Local minimum is one of the main problems in training of neural networks, because when the algorithm converges in one of these minima, the error may also be acceptable but the result would not be optimal. There are several methods to reduce, and eventually rule out, the possibility to get stuck in local minima; one of these is the online mode. In learning online mode the samples of the training set are acquired (or used) incrementally during the training process; on the contrary,

[3]The gradient gives the versus in which the function grows more rapidly. From a pure mathematically point of view, the gradient is a vector of partial derivatives of a function with respect to all its variables: $\nabla E(w) = \left(\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \frac{\partial E}{\partial w_3}, ..., \frac{\partial E}{\partial w_n} \right)$

learning batch mode assumes that the entire training set is available before the training begins. Here below the method for updating weights in online mode:

$$\Delta w_j = \eta \, (d^\mu - y^\mu) \, g'(h) \, x_j^\mu \tag{4.23}$$

and in batch mode:

$$\Delta w_j = \eta \sum_\mu (d^\mu - y^\mu) \, g'(h) \, x_j^\mu \tag{4.24}$$

Now the backpropagation algorithm can be illustrated. As stated before, this algorithm computes the gradient of the error function to find the optimal vector of weights which minimise $E(w)$. To perform that, backpropagation algorithm requires two step:

- *Forward propagation of the signal*, in which a training pattern of input (chosen randomly) is propagated through the neural network in order to generate the real output (of the neural network).
- *Backwards propagation of the error*, in which the descent gradient method is applied and the deltas are generated.

Consider a neural network composed by one input layer, one hidden layer and one output layer.

The forward propagation phase requires the following steps:

1. The μ–pattern x^μ is chosen randomly and used as input
2. Hidden neurons receive as input:

$$h_j^\mu = \sum_k w_{j,k} \, x_k^\mu \tag{4.25}$$

   where $w_{j,k}$is the weight between input neuron k and hidden neuron j. In turn, hidden neurons generate outputs:

$$\alpha_j^\mu = g\left(\sum_k w_{j,k} \, x_k^\mu\right) \tag{4.26}$$

3. Output neurons receive as inputs:

$$h_i^\mu = \sum_j v_{i,j} \, \alpha_j^\mu = \sum_j v_{i,j} \, g\left(\sum_k w_{j,k} \, x_k^\mu\right) \tag{4.27}$$

where $v_{i,j}$ is the weight between hidden neuron j and output neuron i. Thus, the output of the neural network is:

$$y_i^\mu = g\left(\sum_j v_{i,j}\, g\left(\sum_k w_{j,k}\, x_k^\mu\right)\right) \tag{4.28}$$

Once it has obtained the actual output y_i^μ, the error backward propagation phase can start:

1. The error function must be determined:

$$\begin{aligned} E(w) &= \frac{1}{2}\sum_{\mu,\,i}(d_i^\mu - y_i^\mu)^2 \\ &= \frac{1}{2}\sum_{\mu,\,i}\left[d_i^\mu - g\left(\sum_j v_{i,j}\, g\left(\sum_k w_{j,k}\, x_k^\mu\right)\right)\right]^2 \end{aligned} \tag{4.29}$$

2. Applying the gradient descent to connections between output and hidden neurons, one gets:

$$\begin{aligned} \Delta v_{i,j} &= -\eta\,\frac{\partial E}{\partial v_{i,j}} \\ &= \eta\sum_\mu (d_i^\mu - y_i^\mu)\, g'\left(h_i^\mu\right)\alpha_j^\mu \\ &= \eta\sum_\mu \delta_i^\mu\,\alpha_j^\mu \end{aligned} \tag{4.30}$$

The quantity δ_i^μ is called "delta", and it is used to updated weights.

$$\delta_i^\mu = (d_i^\mu - y_i^\mu)\, g'\left(h_i^\mu\right) \tag{4.31}$$

3. Applying the gradient descent to connections between hidden and input neurons, one gets:

$$\begin{aligned}
\Delta w_{i,j} &= -\eta \frac{\partial E}{\partial w_{i,j}} \\
&= -\eta \sum_{\mu} \frac{\partial E}{\partial \alpha_j^{\mu}} \frac{\partial \alpha_j^{\mu}}{\partial w_{i,j}} \\
&= \eta \sum_{\mu,i} (d_i^{\mu} - y_i^{\mu})\, g'\left(h_i^{\mu}\right) v_{i,j}\, g'(h_j^{\mu})\, x_k^{\mu} \\
&= \eta \sum_{\mu,i} \delta_i^{\mu}\, v_{i,j}\, g'(h_j^{\mu})\, x_k^{\mu} \\
&= \eta \sum_{\mu} \delta_j^{\mu}\, x_k^{\mu}
\end{aligned} \tag{4.32}$$

The delta at this step is:

$$\delta_j^{\mu} = g'(h_j^{\mu}) \sum_i v_{i,j}\, \delta_i^{\mu} \tag{4.33}$$

To summarised, the recipe for backpropagation algorithm is:

1. Consider a multilayer perceptron with L layers and initialise weights of each layer randomly.

2. Take a random pattern x_k^{μ} and apply it to input neurons.

3. Propagate forward the signal through the neural network:

$$y_i^l = g(h_i^l) = g\left(\sum_j w_{i,j}^l\, y_j^{l-1}\right) \tag{4.34}$$

   until the final outputs y_i^L are calculated. notice that the index l indicates the number of the layer $(l : 1, L)$.

4. Evaluate the deltas for output neurons:

$$\delta_i^L = (d_i^{\mu} - y_i^L)\, g'\left(h_i^L\right) \tag{4.35}$$

5. Evaluate the deltas for other layers:

$$\delta_j^{l-1} = g'(h_j^{l-1}) \sum_i w_{i,j}^t\, \delta_j^t \tag{4.36}$$

6. After computing deltas for each layer, one uses them to update weights:

$$\Delta w_{i,j}^{l} = \eta \, \delta_i^l \, y^{l-1_j} \tag{4.37}$$

for each layer:

$$w_{i,j}^{new} = w_{i,j}^{old} + \Delta w_{i,j} \tag{4.38}$$

7. Start again from step 2.

Obviously, the algorithm must be repeated until the error function reaches a threshold defined a priori.

In conclusion, backpropagation algorithm is one of the best and most used learning algorithm for training feedforward neural networks. However, as mentioned before, is not always guaranteed to find the global minimum of the error function; but it is possible to get stuck in local minima, if the error function is non–convex. Notice that the backpropagation method does not require scaled input, however the normalization procedure may optimize the training step, both regarding the speed that the convergence.

4.4.2 Unsupervised Learning

Neural networks classified as unsupervised learning must be able to determine autonomously weights of their connections, on the basis of input patterns presented in succession. Input sets must be necessarily redundant, because these networks learn to react better to input patterns which present the most frequent components.

During training, weights are updated according to an internal rule specified a priori. The more widely used rules are the Hebb's rule and the Oja's rule (which is an optimisation of Hebb's rule).

Hebbian lerning

Hebb's rule is a method to evaluate weights changes in neural networks, in which a change in the strength of a connection between neurons depends on the pre –and post–synaptic neural activities. In particular, it specifies how much the weight of the link between two neurons should be increased or decreased proportionally to the product of their activation. This method is based on the reinforcement of synapses with synchronous activity, and for this reason it belongs to the class of reinforcement learning.

The rule, defined in 1949 by the psychologist Donald Olding Hebb, states that connections between two units might be reinforced if the neurons fire simultaneously. If a pre–synaptic neuron A repeatedly activates a post–synaptic neuron B,

connection between these two neurons will be strengthen. The activities of neurons should be close in time and synchronous.

The Hebb's principle is formulated in the following way:

$$\Delta w_i = \eta \, x_i \, x_j \tag{4.39}$$

where η is the learning rate, x_i and x_j are the activities of pre–and post–synaptic neurons respectively and w_{ij} the strength of the connection between them.

Inputs that occur most frequently are those with greater influence in the determination of weights; therefore, when one present a new input belonging to the set of the most frequent examples, the output y will be maximum. The Hebbian Rule works well as long as all the input patterns are orthogonal or uncorrelated.

Despite the success, the Hebb rule has limitations regarding the unlimited growth of weights. Indeed, weights can increase without restrictions and there is not a decreasing mechanism. In order to overcome this limit one can use several solutions. It is possible to force a threshold to updating or normalize the weights each time these are update. The method most widely employed is the Oja's rule.

Oja learning rule include an additional term proportional to y^2 in the Hebb's rule:

$$\Delta w_i = \eta \, y \, (x_i - w_i \, y) \tag{4.40}$$

The updating method in Formula (4.40) limits the increase of synaptic weights without having to resort to normalization, but only with the introduction of a "forgetting factor" that is proportional to the current input. In this way the network converges to the solution in which the output expresses the direction of maximum variance of the input distribution; patterns that lie along this direction activate a greater response (synaptic reinforcement). Therefore, Oja's solves all stability problems and generates an algorithm for principal components analysis.

4.5 Types of Neural Networks

4.5.1 Multilayer Perceptron

The multilayer Perceptron (MLP) is a feedforward neural network model which estimate a relationship between a vector of input data and the corresponding outputs. It has roots in the Rosenblatt one–layer Perceptron. The latter is not so fascinating because the presence of only input layer and output layer, enables it to solve exclusively linearly separable problems (Section 4.2). The MLP is composed of one or more hidden layers of artificial neurons with non–linear activation functions, in addition to input and output layers. The role of hidden units is precisely

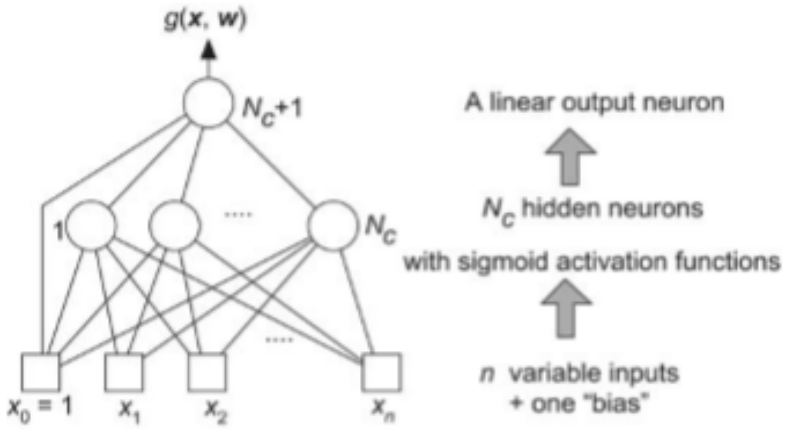


Figure 4.4: One hidden layer Perceptron with n inputs, c hidden neurons and one output [40].

to discriminate data that is not linearly separable (or separable by a hyper–plane) and allow therefore the resolution of non–linear and more complex problems.

Multilayer Perceptrons is trained using backpropagation algorithm and find application in different fields such as time series prediction, image recognition, data compression (Principal Component Analysis—PCA), character recognition, autonomous driving, computer vision, financial prediction, etc.

Consider now one of the most widely and important class of feedforward multilayer Perceptrons: networks composed of a single layer of hidden neurons with a sigmoid activation function and one linear output neuron (Figure 4.4). The output of the network is given by:

$$\begin{aligned} y(x,\, w) &= \sum_{i:1}^{N_c} \left[w_{N_c+1,i}\, tanh \left(\sum_{j:1}^{n} (w_{ij}\, x_j) + w_{i0} \right) \right] + w_{N_c+1,0} \\ &= \sum_{i:1}^{N_c} \left[w_{N_c+1,i}\, tanh \left(\sum_{j:0}^{n} w_{ij}\, x_j \right) \right] + w_{N_c+1,0} \end{aligned} \tag{4.41}$$

where x is the vector of $(n+1)$ input elements and w is the vector of $(n+1)\, N_c + (N_c+1)$ weights. The output $y(x, w)$ is a linear function of the weights connecting hidden layer to output node, and a non–linear function of weights connecting input neurons to hidden neurons. Thus, the output of a multilayer perceptron with sigmoid activation function is a non–linear in its inputs and its parameters.

Universal Approximation Theorem

The Universal Approximation Theorem states that a MLP with one hidden layer trained using backpropagation algorithm can approximate continuous functions on compact subset $\mathbb{R}^2$, under constraint on activation function.

Let $g(x)$ be a non–constant, bounded, and monotonically–increasing continuous function. Let I_N refer the N–dimensional hypercube $[0,\, 1]^N$ and let $C(I_N)$ be the space of continuous functions on I_N. Given a function $f \in C(I_N)$ and $\varepsilon > 0$, there exists an integer M and constant real ensembles v_i, θ_i and w_{ij}, where $i : 1,\, M$ and $j : 1\, N$, such that it can be defined the function:

$$y(x_1, x_2, ..., x_N) = \sum_{i:1}^{M} v_i\, g\left(\sum_{j:1}^{N} w_{ij}\, x_j - \theta_i\right) \tag{4.42}$$

Equation (4.42) represents the output of the MLP and is an approximation of the function f, that is:

$$\sup_x |f(x) - y(x)| < \varepsilon \quad \forall\, x \in I_N \tag{4.43}$$

Finally, one can affirm that standard multilayer feedforward perceptrons are able to approximate any measurable function to any desired degree of accuracy. This is maybe the most important property of MLP and is the reason of the strong success of this class of neural networks.

4.5.2 Radial Basis Function

Radial Basis Function (RBF) networks are neural networks which use radial functions as activation functions. In general, these networks are used to solve exact interpolation problems, but also for classification or time prediction problems. RBF networks have usually three layers: one input layers, one hidden layers composed of neurons with radial activation functions and one output layer. Each input neuron is connected with all hidden neurons. The output of the neural network is a linear combination of RBFs of the input units:

$$y(x) = \sum_{\mu:1}^{M} w_\mu\, \phi(|x - x^\mu|) \tag{4.44}$$

where M is the number of hidden neurons and x^μ is a centre vector of neuron μ. Function $, \phi(|x - x^\mu|)$ is the radial activation function of hidden nodes: this function depends only on the distance between the input x and the centre x^μ. Types of radial basis function are:

- *Gaussian function*: $\phi(r) = e^{\frac{-r^2}{s\,\sigma^2}}$
- *Cauchy function*: $\phi(r) = \dfrac{\sigma^2}{r^2 + \sigma^2}$
- *Multiquadris function*: $\phi(r) = \dfrac{(r^2 + \sigma^2)^{\frac{1}{2}}}{\sigma^2}$
- *Inverse Multiquadrics function*: $\phi(r) = \dfrac{(r^2 + \sigma^2)^{\frac{-1}{2}}}{\sigma^2}$

The exact interpolation problem requires each input vector has exactly the value of the corresponding target. This means that, given the target t^μ, the goal is to find a function (the output of the network) such that:

$$y(x^\mu) = t^\mu \tag{4.45}$$

Training session of RBF neural networks consists in two steps. In first step, one has to determine the centres x^μ and usually these are obtained using k-means algorithm. Notice that this process is unsupervised. Second learning step is used simply to find the optimal weights using, usually, the gradient descent method.

4.5.3 Kohonen Self-Organizing Maps

In 1983, Teuvo Kohonen studied a neural model able to reproduce the formation process of sensorial maps in human brain. Self–organizing maps are utilised to cluster input patterns into set of similar patterns.

Kohonen networks are a type of unsupervised learning, in particular they are a generalisation of the "winner–takes–all"[4] (WTA)networks; more precisely, the choice method of the winner neuron is the same as in WTA, but the training process il slightly different. Indeed, Kohonen networks are based on the biological property that neurons closest to active neurons have strong (excitatory) connections, conversely distant neurons have weak (inhibitory) connections. Consequently, not only the vector of weights of winner neuron is updated, but also the vectors of other neurons proportionally to the proximity of winner neuron. The training of Kohonen networks are different from that of WTA networks, because the first networks have the aim to create self–organised maps in which topologically close neurons must respond maximally to similar stimuli.

[4]WTA is an unsupervised learning method by which neurons compete for activation. In WTA networks only one winner at a time exist, that is the excited neuron: only the vector of weights of the winner neuron will be updated. The winner neuron may be the neurons closest to input vector or that with maximum activity. WTA is a good method for classification [48].

The structure of the network is composed of one input layer fully connected to the output layer, whose neurons are connected to a neighbourhood of neurons in accordance with a lateral inhibition system defined as a Mexican hat (long range inhibition and short range excitation). In input layer there is only a single winner unit (with the maximum activation value) for each stimulus: this neurons represents an input class to which it belongs.

The main advantages of Kohonen SOMs are that they perform very well and are easy to understand. However one big problem of these networks is getting data; indeed, it is so difficult acquire all data (one value for each dimension of each member of samples) needed to create maps.

4.5.4 Hopfielf Networks

Hopfield networks are fully connected neural networks used particularly for the realization of associative memories. An associative memory is the ability to remember the information randomly related; from a modelling point of view, it corresponds to a minimum of the energy function. The collective properties of the model produce an associative memory for recognition of corrupted configurations and rescue of missing information. Furthermore, Hopfield believed that every physical system can be a potential memory device, if it has a certain number of stable states which serve as attractors of the system itself. He formulated the thesis that the stability and the placement of these attractors are spontaneous properties of systems consisting of a large number of attractors.

The effectiveness of these networks is low in comparison to other networks more complete, but is very important from a theoretical point of view because the model is very similar to biological networks thanks to neurons fully connected to each other; in addition, these networks are trained using Hebbian learning algorithm and shows a strong connection with dynamic systems. This model is classified as unsupervised learning: the network learns without having examples, only with the use of the "energy concept".

Hopfield networks are:

- *Fully connected*, each neurons is linked to all other neurons end is devoid of auto–connections ($w_{ii} = 0 \ \forall i$).
- *Symmetric*, these networks have a symmetric weights matrix ($w_{ij} = w_{ji} \ \forall i, j$).
- *Non–linear*, each unit has a non–linear and invertible activation function.
- *Recurrent*, the structure presents direct cycles .

The symmetric matrix property, is essential for generating associative memories, as it leads to the convergence: under this assumption, a Lyapunov function

(Hemiltonian function) exists and is minimised during process evolution. Minima of Lyapunof function corresponds to attractors of the system.

Furthermore, the network may be discrete or continuous. First model of Hopfield neural networks allows neurons to assume only two values, 0 or 1; in successive models the neuron can be a value in the interval $[-1, 1]$.

4.5.5 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are biologically–inspired variants of multilayer Perceptrons [42]. In particular, neurons connectivity patterns of CNNs are influenced by visual cortex of animals, whose individual neurons are organised in order to respond to the overlapping regions tiling the visual field. These regions are named "receptive fields".

The name "convolutional neural network" indicates that the network uses a mathematical operation named convolution. Convolution is a specialized kind of linear operation between two functions which produce a third convoluted function that is a modified version of the first function. CNNs are simply neural networks which employ convolution instead of general matrix multiplication in one or more of their layers [43].

CNNs are mainly employed in image and video recognition problems, but they have also applications in natural language processing and classification problems.

When used for image recognition, input consists of images and this lead to a 3–dimensional architecture of neurons (width, height, depth) in order to take into account the spacial structure of data. Unlike traditional neural networks, neurons in each layer are only connected to a small region of the layer before it, instead of being fully connected. Usually, the number of layers is larger than those of classical MLPs, and they are separated in three main types of layers: Convolutional Layer, Pooling Layer and Fully–Connected Layer (as regular MLPs).

The basic idea is to start with a complete image and continually split it up, step–by–step, until a single result is achieved. Obviously, more convolution steps are performed, more complicated features the network is able to learn to recognize. The input of a convolutional layer is a $n \times n \times m$ image, where n represents width and height, and m the depth (e.g. a RGB colour image has $m = 3$). The convolutional layer must have k kernels (learnable filters with small receptive fields) of dimension $r \times r \times q$. r has to be smaller than the size of the input image and q can be the equal to the depth m or smaller and can be different for each kernel. During the forward pass, every filter is convolved across the width and height, computing the dot product between the entries of the kernel and the input and producing k activation maps of size $n - m + 1$ [44]. Consequently, the network learns kernels which activate when they observe some specific type of feature at

some spatial position in the input image. All activation maps are setted along the depth dimension, with the aim to form the output volume of the convolution layer.

As mentioned before, convolutional architectures usually use pooling layers after each convolution. These pooling layers simplify the information of the previous convolution layer, by selecting the most prominent value (max pooling) or averaging the values calculated in by the convolution (average pooling) [45]. The idea at the basis of pooling is that once a feature has been found, it is more important its approximate location relative to other features than its exact location. Pooling layer has the function to reduce the spatial size of the input image and thus to reduce the number of parameters and the computational complexity of the neural network.

Finally, after convolutional and pooling layers, there are fully connected layers which have the same architecture of traditional neural networks and perform like these. The high–level reasoning of CNNs is done in fully connected layers.

Convolutional neural networks are the most important innovation in Image Classification field and are the core of most Computer Vision systems today. Further benefits of CNNs are that they are easy to train (using backpropagation learning algorithm) and need less free parameters than traditional fully connected networks with the same number of hidden neurons.

4.6 Neural Networks in Finance

The development of neural networks applications covering different parts of the financial world: banks, hedge funds and private investors. It has been estimated that after the military applications, financial applications are the field in which research on neural networks has received the most financial support. Moreover, the incentive to invest in this field is very strong: the goal is to achieve a superior system (compared to the rest of the market) of selection, classification or prediction.

For instance, researchers have recently developed decision making tools which can be used to help decision makers with their decisions. A stock trader no longer needs to analyse complicated charts in order to make a decision. Alternatively, agents can use neural networks which are able to assist them in making decisions [46].

In financial field, neural networks are mainly used for time series prediction [47]. Indeed, due to the importance of markets, investments are usually led by some form of prediction. The majority of these applications aims to realise speculative profits through trading (usually for short time) of financial assets, mainly stocks, exchange rates, futures and options.

Some studies state that forecasts with neural networks are better than those

obtained with classical methods. Proof of it is the work of Yao et al. [50] about option prices prediction, which suggest that for volatile markets a neural network option pricing model outperforms the traditional Black–Scholes model. Even other works like those of Amilon [51] and Hutchinson et al. [52] examined whether a neural network (multilayer perceptron or radial basis function network) can be used to find options pricing formula better then the BS formula. Comparisons reveal that the neural network model beats the classical model both in pricing and hedging performances.

Same results can be find in stock price prediction: Ahangar et al. [53] showed that artificial neural network method is more efficient than linear regression method in estimating the stock price of activated companies in Tehran stock exchange.

In conclusion, artificial neural networks provide attractive alternative method especially for forecasting problem. One of distinguishing features with respect to classical methods is that ANNs are data–driven self–adaptive methods in that there are few a priori assumptions about the models for problems under study. They learn from example and capture the functional relationship between input data and the target output given as example, even if the underlying relationship is unknown or hard to examined. Thus, artificial neural networks are suitable and working well tools for problems whose solutions is difficult to describe or as efficient alternative to traditional (and not optimal) methods [54].

Chapter 5

R

R is a programming language and software environment for statistical computing and graphics supported by the R Foundation for Statistical Computing [55]. This language was initially written almost twenty years ago with the aim to allow academic statisticians with sophisticated programming tools to execute complex data statistical analysis and display results in a multitude of visual graphics. Indeed, R provides a wide variety of statistical (linear and non–linear modelling, classical statistical tests, time–series analysis, classification, clustering, etc.) and graphical techniques, and it is highly extensible. One of R's best qualities is the ease with which well–designed publication–quality plots can be produced, including mathematical symbols and formulae where needed. R software is freely available under the GNU General Public License and it is provided for various operating systems (for example Unix, GNU/Linux, macOS, Microsoft Windows).

In August 2016, the TIOBE Programming Community Index ranks R to 20th place [56]. Its popularity is also due for the large willingness of modules distributed under the GPL licence and organized in a special website called "CRAN" (Comprehensive R Archive Network).

5.1 History

R is named in honour of the first names of the two authors Ross Ihaka and Robert Gentleman [57], who first implemented R in 1991 at the Department of Statistics of the University of Auckland, New Zealand. In 1995, two years after the presentation of R to the public, Mächeler convinced Ihaka and Gentleman to use the General Public License (GNU) to make R a free available software. Next year a public mailing list was created (R–help and R–devel). Following the introduction of the mailing lists, development on R accelerated; this was partly because the authors obtained many more reports and suggestions and partly because they also began to

receive patches and code contributions [58]. The number of contributors and users became so great that Ihaka and Gentleman, joined by Mächeler, were no longer able to satisfy the requested changes. As a result, in the mid of 1997 a larger core group, the "R Core Team", has been established which still currently develops and modify the R source code. The group currently consists of Doug Bates, John Chambers (the creator of S language), Peter Dalgaard, Seth Falcon, Robert Gentleman, Kurt Hornik, Stefano Iacus, Ross Ihaka, Friedrich Leisch, Uwe Ligges, Thomas Lumley, Martin Maechler, Duncan Murdoch, Paul Murrell, Martyn Plummer, Brian Ripley, Deepayan Sarkar, Duncan Temple Lang, Luke Tierney, and Simon Urbanek.

Two existing languages have heavily influenced the design of R: S and Scheme. S is a language for statistical computing conceived by Chambers at Bell Labs in the mid 1970s, based on non–functional languages like C and Fortran. Scheme is a elegant, functional and concise language created during the same period at MIT AI LAB by Steele and Sussman. Apparently, the R's language is very similar to the S's one, but the underlying implementation and semantics are derived from Scheme. Indeed, the authors of R tied to design a software with the statistical skills of S and the functionality of Scheme.

5.2 Statistical features

«R is more than a programming language. It is an interactive environment for doing statistics» [59]. R has a fantastic mechanism for doing statistical analysis, including linear and non–linear modelling, classifications, traditional statistical tests, clustering, time–series analysis, etc. Another strength of R are its libraries, which implement a wide variety of static graphical techniques. For dynamical and interactive graphics several packages are available.

R was created by statistician to make statistical analysis easier. This feature allows R being a very powerful tool for performing virtually any statistical computations. However, thanks to the wide users contributions and development of packages for every purpose, nowadays R is not only more a software for statistical analysis. Actually, it is a suitable application for a wide variety of non–statistical tasks, including graphical visualization, data processing and analysis of all sorts.

5.3 Programming features

R can appear disheartening at first, that is because R syntax is different from that of many other programming languages, not necessarily because it is more complicated than others.

R is an interpreted languages which means that, contrary to compiled languages as C, C++, Pascal and Java, it is not necessary a compiler to create a program before using it. As all interpreted languages, R is accessible by user through a command line interpreter. The user has simply to write down the code at the R command prompt and send it to R, which executes the code, making procedures really easy. In practice, if one types a mathematical operation (e.g. $4 + 4$) at the command prompt and presses enter, the computer will return the result of such operation. Nevertheless, this ease comes at the cost of speed of code execution.

In R data can be structured as vectors, lists, matrices, arrays or data frames; the users can operate on them using functions for performing statistical analyses and producing graphs. Notice that, R is a vector–based language, thus scalar data type is not a data structure. Scalars in R are represented like vectors of unitary length.

Unlike most other programming languages, R allows to apply functions to the whole vector in a single procedure without the necessity of an explicit loop. This possibility to perform many operations in a single step is one of R's features that makes it so useful and powerful for data analysis.

Another important feature of R concerns import/export functionality. Importing data from other software is pretty simple such as a comma delimited text file, Excel, SPSS, SAS, Stata, etc. In addition to R's default functions, several packages have been created to provide import/export data (for instance *csvy*, *DataLoader*, *haven*, *ImportExport*,*RDML*, *readstata13*, *rgexf*, *SAScii*, etc.).

R has a system for object–oriented programming [1], based on generic functions. As S language, R has two object systems, named as S3 and S4. S3 is the original and most common type in R: it uses generic function style and has no formal definition. Conversely, S4 is much more rigorous and has a formal definition, that provides more control, but is less interactive.

5.4 Package to implement ANN

«You have a lot of prepackaged stuff that's already available, so you are standing on the shoulders of giants» Google's chief economist told The New York Times in 2009.

The skills of R are enhanced by powerful user–created packages, which allow more specialised statistical analysis, graphical tools, import/export possibilities, interfaces to communicate with other software, etc. Among almost 4000 packages for various purposes developed by users and programmers around the world, CRAN also provides packages for applications of neural networks.

[1] Object–oriented programming is a programming language model oriented around objects and their interactions, which uses them to design applications and computer programs.

This thesis has been developed using *neuralnet* [60] package (for reasons explained in the next section), but there are many other packages that deal with all types of neural networks. Here below some R packages containing functions to train neural networks:

- *nnet* [61] provides the training of feed–forward neural networks with one hidden layer using traditional backpropagation, and for multinomial log-linear models;
- *RSNNS* [62] offers a suitable interface to the popular Stuttgart Neural Network Simulator. SNNS is a comprehensive application for neural network model building, training, and testing, which contains several network architectures implemented as multilayer perceptrons, recurrent Elman and Jordan networks, radial basis function networks, RBF with dynamic decay adjustment, Hopfield networks, time–delay neural networks, self–organizing maps, associative memories, learning vector quantization networks, and different types of adaptive resonance theory networks [63];
- *simpleNeural* [64] trains multilayer perceptrons with one hidden layer for bi–or multiclass classification problems;
- *AMORE* [65], A MORE flexible neural network package, which releases the TAO–robust learning algorithm;
- *elmNN* [66], provides function for training and prediction of single hidden–layer feed–forward neural networks using the Extreme Learning Machine algorithm;
- *deeplearning* [67], contains functions for deep neural network with rectifier linear units trained with stochastic gradient descent method and batch normalization for regressions and classifications;
- *rnn* [68], includes an example of recurrent neural networks.

5.4.1 neuralnet

neuralnet is a R package built to train multi-layer perceptrons in the context of regression analyses, i.e. to approximate functional relationships between covariates and response variables [69]. It depends on two other Rpackages: *grid* and *MASS* and its usage leaned towards that of functions dealing with regression analyses like *lm* and *qlm*.

The package *neuralnet* focuses on supervised learning, in particular it trains neural networks using backpropagation, resilient backpropagation (RPROP) with

(Riedmiller, 1994) or without weight backtracking (Riedmiller and Braun, 1993) or the modified globally convergent version (GRPROP) by Anastasiadis et al. (2005).

neuralnet is a very flexible package. The user is able to include several hidden layers with one or more neurons for each layer and he can theoretically handle an arbitrary number of covariate (input variables) and response variable (output variables). Furthermore, the package provides different choices for activation function and error function.

As previously stated, one of the main problems in ANN training occurs when a neural network gets stuck in a local minimum. Neural network trained with *neuralnet* has a lower probability to get stuck in a minimum with respect to other training packages, because of its random rearrangement of training data.

In addition, the package provides functions to train and visualize the neural network and also a function which compute predictions for new covariate combinations.

neuralnet

The function *neuralnet* trains the neural network using backpropagation (″backprop‟), resilient backpropagation with (″rprop+‟) or without (″rprop-‟) weight backtracking, or the modified globally convergent version by Anastasiadis et al. (″grprop‟).

By default, the training function uses the resilient backpropagation (Rprop), in particular ″rprop+‟. Rprop is a first-order optimization algorithm, based on backpropagation algorithm, which was created by Martin Riedmiller and Heinrich Braun in 1992. It is more complex then classical backpropagation, but Rprop has advantage in training speed and efficiency, indeed it is one of the fastest weight update mechanisms. Moreover, resilient backpropagation algorithm does not require the specification of any free parameters, in particular the learning rate (unlike the traditional backpropagation). Differently from backpropagation, Rprop takes into account only the sign of the partial derivatives of the error function to update the weights, instead of the magnitude and acts independently on each weight. This guarantees an equal influence of the learning rate over the entire neural network. Resilient backpropagation is considered highly suitable for applications where the gradient is numerically estimated or the error is noisy.

Here below a comparison between the pseudo–code of learning algorithms. Regular backpropagation pseudo–code is given by:

```
for all weights{
    weights := weights - grad*delta
}
```

while pseudo–code of resilient backpropagation (with backtracking) is given by:

```
for all weights{
    if (grad.old*grad>0){
        delta := min(delta*eta.plus, delta.max)
        weights := weights - sign(grad)*delta
        grad.old := grad
    }
    else if (grad.old*grad<0){
        weights := weights + sign(grad.old)*delta
        delta := max(delta*eta.minus, delta.min)
        grad.old := 0
    }
    else if (grad.old*grad=0){
        weights := weights - sign(grad)*delta
        grad.old := grad
    }
 }
```

Alternatively, the ANN may be trained using the globally convergent algorithm. Grprop is based on the resilient backpropagation algorithm without backtracking and it modifies one learning rate, either the one associated with the smallest absolute gradient (sag) or the smallest learning rate itself (slr). It has been shown that grprop exhibits better convergence speed and stability than rprop.

As was mentioned early, *neuralnet* gives the opportunity to define number of hidden layers and that of hidden neurons of each layer, according to the complexity of the problem to solve. Notice that the addiction of hidden layers and hidden neurons increases the computational costs of neural networks. Learning algorithm and number of hidden layers are not the only parameters that costumer has to choice to train the ANN. Below the pseudo–code shows the usage of *neuralnet* function with default arguments.

```
neuralnet(formula, data, hidden = 1, threshold = 0.01, stepmax = 1e+05,
          rep = 1, startweights = NULL, learningrate.limit = NULL,
          learningrate.factor = list(minus = 0.5, plus = 1.2),
          learningrate=NULL, lifesign = "none", lifesign.step = 1000,
          algorithm = "rprop+", err.fct = "sse", act.fct = "logistic",
          linear.output = TRUE, exclude = NULL, constant.weights = NULL,
          likelihood = FALSE)
```

Following, descriptions of *neuralnet* arguments:

- *formula*, a description of the model to be fitted (i.e. *output* $\sim$ *input1+input2*)

- *data*, a dataframe containing the variables specified in *formula* (names of dataframe's columns have to correspond to names used in *formula*)
- *hidden*,a vector of integers specifying the number of hidden neurons in each layer (i.e. the vector(3,4) generate a neural network with two hidden layers, the first one with three and the second one with four neurons)
- *threshold*, a numeric value specifying the threshold for the gradient of the error function as stopping criteria
- *stepmax*, the maximum number of steps for the training. A neural network stops training when reaches *stepmax*
- *rep*, the number of the repetition for the training process
- *startweights*, a vector containing starting values of the weights. If NULL, starting weights are randomly initialised.
- *learningrate.limit*, a vector or a list containing the lowest and the highest limit for the learning rate. Argument used only for rprop and grprop algorithms.
- *learningrate.factor*,a vector or a list containing the multiplication factors for the upper and lower learning rate. Argument used only for rprop and grprop algorithms.
- *learningrate*, a numeric value specifying the learning rate. Argument used only for traditional backpropagation algorithm.
- *lifesign* a string specifying how much the function will print during the calculation of the neural network. Argument may be ″none‟, ″minimal‟or ″full‟
- *lifesign.step*, an integer specifying the stepsize to print the minimal threshold in ″full‟lifesign mode
- *algorithm*, the supervised learning algorithm to calculate the neural network's weights. The possible choices are ″prop‟(traditional backpropagation), ″rprop+‟and ″rprop-‟(resilient backpropagation with or without weights backtracking), ″sag‟and ″slr‟(modified globally convergent algorithm, grprop). ″sag‟and ″slr‟define the learning rate that is changed according to all others (″sag‟refers to the smallest absolute derivatives, while ″slr‟to the smallest learning rate)
- *err.fct*, a differentiable function that is used for the evaluation of the error. The alternatives are: ″sse‟and ″ce‟which refer to the sum of squared error and the cross-entropy

- *act.fct*, a differentiable activation function. The string ″logistic‟and ″tanh‟are possible for logistic function and tangent hyperbolicus
- *linear.output*, a logical value. If *act.fct* should not be applied to output neurons, *linear.output* has to be TRUE, otherwise FALSE
- *exclude*, a vector or a matrix specifying weights that should be excluded from training.If given as a vector, the exact positions of the weights must be known. A matrix with n rows and 3 columns will exclude n weights, where the first column indicates the layer, the second one the input neuron of the weight and the third one the output neuron of the weight.
- *constant.weights*, a vector indicating the values of the weights that are excluded form the training step and treated as fix
- *likelihood*, a logical value. If *err.fct* is equal to the negative log-likelihood function, the information criteria AIC and BIC will be calculated

neuralnet returns an object of class *nn*, which is a list containing the following results and information:

- *call*, the matched call
- *data*, the *data* argument (see above)
- *response*, the output target variables extracted from the *data* argument
- *covariate*, the input variables extracted from the *data* argument
- *model.list*, a list containing the covariate and the response variables extracted from the *formula* argument
- *err. fct*, the error function
- *act.fct*, the activation function
- *net.result*, a list containing the overall results of the neural network for each repetition
- *weights*, a list containing the weights of the neural network computed for each repetition
- *generalised.weights*, a list containing the generalised weights of the neural network computed for each repetition

- *result.matrix*, a matrix containing the reached threshold, needed step, AIC and BIC (if *likelihood=TRUE*) and weights for every repetitions. Each column represents one repetition.
- *startweights*, a list containing the starting weights for each repetition

The usage of *neuralnet* function is described in the following example. A neural network is trained to be able to take a number and calculate the square root (one input neuron and one output neuron). First of all, a training dataframe has to be created: this model needs one list of input variables and one list of output variables, which have been named *input* and *output* respectively. For this problem, it is sufficient only one hidden layer containing four hidden neurons. The *threshold* is set to 0.01 (as default) and the number of repetition is set to 3. Onto the code.

```
> library(neuralnet)
Carico il pacchetto richiesto: grid
Carico il pacchetto richiesto: MASS
> input<-as.data.frame(runif(50, min=0, max=100))
> output<-sqrt(input)
> trainingdata<-cbind(input, output)
> colnames(trainingdata)<-c("input", "output")
>
> net.sqrt<-neuralnet(output~input, trainingdata, hidden=4,
                      threshold=0.01, rep=3, algorithm="rprop+")
>
> net.sqrt
Call: neuralnet(formula = output ~ input, data = trainingdata, hidden = 4,
                threshold = 0.01, rep = 3)

3 repetitions were calculated.

  Error             Reached Threshold Steps
2 0.001263621275    0.009400960828     4071
1 0.003659434043    0.009658088542    10202
3 0.004259410571    0.009463728980     3551
```

As indicated above, *neuralnet* gives back an object of class *nn* (*net.sqrt*), in which basic information about training process have been saved. A summary of main results is stored in *result.matrix*.

```
> net.sqrt$result.matrix
```

```
                                1                2                3
error                0.003659434043   0.001263621275   0.004259410571
reached.threshold    0.009658088542   0.009400960828   0.009463728980
steps            10202.000000000000 4071.000000000000 3551.000000000000
Intercept.to.1layhid1   -1.883961075389  -2.910224500552   2.148240763507
input.to.1layhid1       -0.265391834373   0.040199734597  -0.030395550513
Intercept.to.1layhid2    4.585840337683  -1.575558408263  -0.513535158093
input.to.1layhid2       -0.057037759309   0.186171717682   0.088484756683
Intercept.to.1layhid3   -0.353190411417  -1.450757636045   0.055798417477
input.to.1layhid3        0.049861459137   0.744584639683  -0.450746813127
Intercept.to.1layhid4    0.452234237166  -2.472719555633  -1.822289469615
input.to.1layhid4       -0.026524268418   0.088878154708   0.037145849786
Intercept.to.output      2.882150627551  -0.873178578053   2.521695223339
1layhid.1.to.output    -14.350463147195   5.573664842619  -3.287963462269
1layhid.2.to.output     -2.868685922399   2.324315929552   4.554093672822
1layhid.3.to.output      8.020296596875   1.851872614122  -3.033181417953
1layhid.4.to.output     -1.857288942925   2.461500030119   4.365224987574
```

result.matrix shows the data information divided in three columns, one for each repetition. Considering the first repetition, the training process needed 10202 steps in order that all absolute partial derivatives of the error function were smaller than the *threshold* specifying above (0.01). The remaining data refer to the estimated weights. For instance, the weight between the input and the first hidden neuron is -0.26 and the weight between the intercept of the hidden layer and the second hidden neuron is 4.58.

plot

The *neuralnet* package also has the possibility to visualize the generated model and show the found weights. To sketch the neural network, the function *plot* needs the neural network (*nn* object) previously calculated and eventually the number of the repetition, usually the one with minimum error (if not stated all repetitions will be plotted, each in a different window). Typing ″best‵for *rep* parameter, the repetition with the smallest error will be plotted. Additionally, it is possible select colour, coordinates and dimension of neurons and synapses. The plot shows also some information about the training process like the overall error and the number of step needed to converge.

In reference to previous example, the following command allows the program to plot the calculated neural network:

```
> plot(net.sqrt, rep="best")
```

In Figure 5.1 the resulting plot.

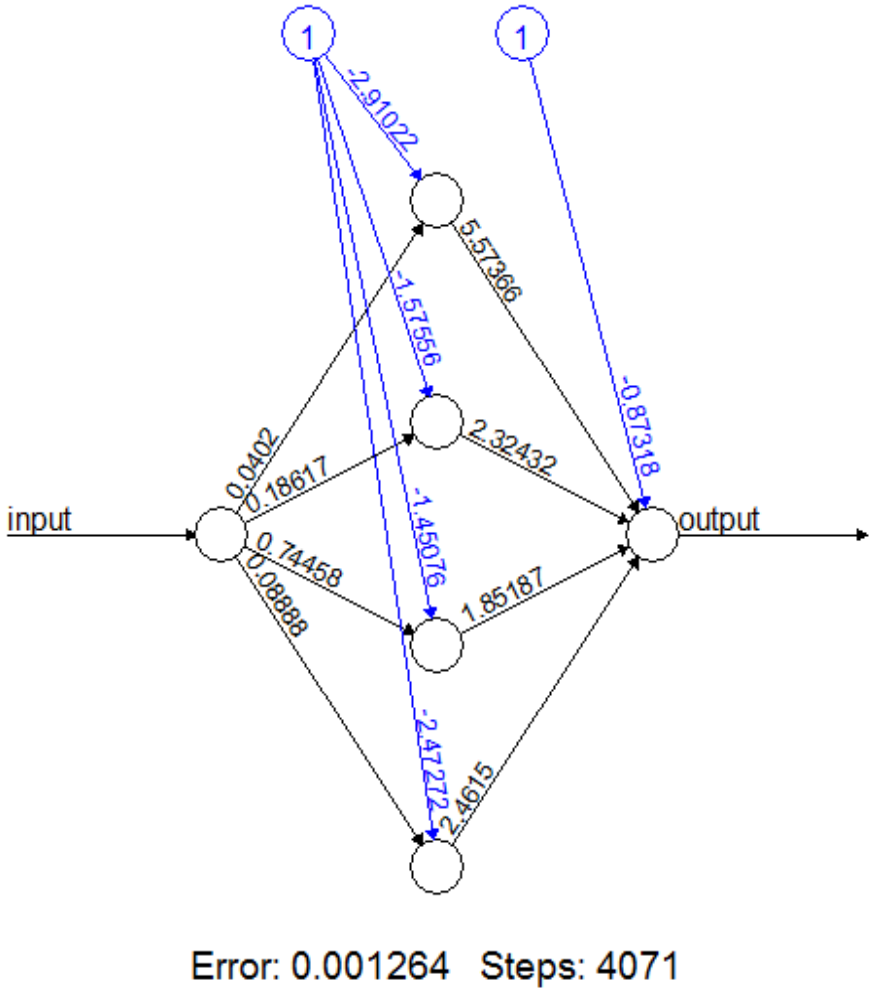


Figure 5.1: Plot of the trained neural network including calculated weights and some information about training process

compute

compute is a function for class *nn* to compute the outputs of all neurons for specifying new covariate vector. Obviously, *compute* only works after a neural network has been trained. The usage and arguments are reported below.

```
compute(x, covariate, rep = 1)
```

- *x*, an object of class *nn* (trained neural network)

- *covariate*, a dataframe containing the variables used to train the neural network. The covariates have to be in the same order of in the training dataframe
- *rep*, an integer specifying the number of the repetition which should be used (usually the one with minimum error)

Function *compute* returns a list containing the following components:

- *neurons*, a list of neurons'output for every layer
- *net.result*, a matrix containing the predicted outputs

To stay with the example, predicted outputs can be evaluated for instance for number 9.

```
> test<-as.data.frame(9)
> result<-compute(net.sqrt, test, rep=2)
>
> result$neurons
[[1]]
1 9
[1,] 1 9
[[2]]
[,1]          [,2]          [,3]          [,4]          [,5]
[1,]    1 0.07253223946 0.5249759583 0.9947824915 0.1580490723
>
> result$net.result
[,1]
[1,] 2.982550047
```

The choice of *neuralnet* package for this thesis has been based on three reasons:

- flexibility in the parameters choice
- possibility to plot computed neural networks
- low probability to end up in a local minimum

Chapter 6

The Models

The purpose of this work has been achieved through the use of Agent–based model and neural networks. Starting with a relatively straightforward market model, it has been implemented a model that can simulate (in the same NetLogo's world) two markets, one of options and one of stock, whose agents learn to predict call and put prices via neural networks.

Traders who use to evaluate options via Black–Scholes formula have been replaced by traders which gain experience and appraise call or put options based on it. Neural networks represent in this sense the ability of every agent to price these financial instruments. Obviously, each agent is more or less able than others, namely it has gained more or less experience in options pricing. It was essential to reproduce the heterogeneity of strategies and experiences of the agents, as this is an essential condition of real markets. As it will be explained later, in addition to the different methods of calculating volatilities, each agent has its own neural network, trained with data relating to single agent which is collected in different spans of time. The fact that neural networks are subjected to training processes of different lengths, implies that agents will be more or less accurate in the prediction phase.

In order to reproduce a market of options in which intelligent and random agents exchange calls and puts according to the underlying price established by the trend of its market (and not by the Gaussian Brownian motion), it was necessary create intermediate models. Table 6.1 shows the three models realised in this project and summarises their characteristics.

Following sections cover the detailed explanations of Elliott'model and the three models mentioned in Table 6.1.

OptionsMarket model		
pricing via BS	**mimicking BS via NNs**	**learning from stock market**
Market with high volatility	Market with high volatility	Market with high volatility
Spot updated via GBM	Spot updated via GBM	CDA model for stock prices
Option pricing via BS	Option pricing via NNs	Option pricing via NNs
Call or put market	Call or put market	Call and put markets

Table 6.1: Comparison between the three models

6.1 OptionsMarket by Elliott

The *OptionsMarket* model [1], provides an important starting point for this project. This model aims to simulate a market trading options using the agent-based modelling language NetLogo.

The market is made by patches, which represent traders whose aim is to buy and sell options, according to Blake-Scholes formula. The agents are arranged on a grid based on patience and judgement. These properties are individual and they make agents intelligent and conscious. Patience refers to how quickly the trader enters the market and how much of the past market the trader looks at in calculations, especially in considering the key unknown value in the formula: underlying stock volatility. Judgement refers to how far the trader incorporates personal judgement and deviates from the Black-Scholes formula. The x and y values of patches indicate their judgement and patience respectively.

The actual stock's volatility is decided by the user and is hidden from the simulated traders. They calculate their own best volatility, called *guessed-sigma*, according to patience and judgement and use it in Black–Scholes formula. Traders enter the market when the patience in days has been reached. An important consideration is that the spot has not an endogenous market, but his price is updated with Geometric Brownian motion.

When markets run, the traders gain and lose money as they buy and sell options: each agent observes the evolution of underlying stock price, updates his own volatility using individual patience and judgement, then calculates its target price based on the Black–Scholes formula (*Buy* and *Sell*). Buy and sell prices are

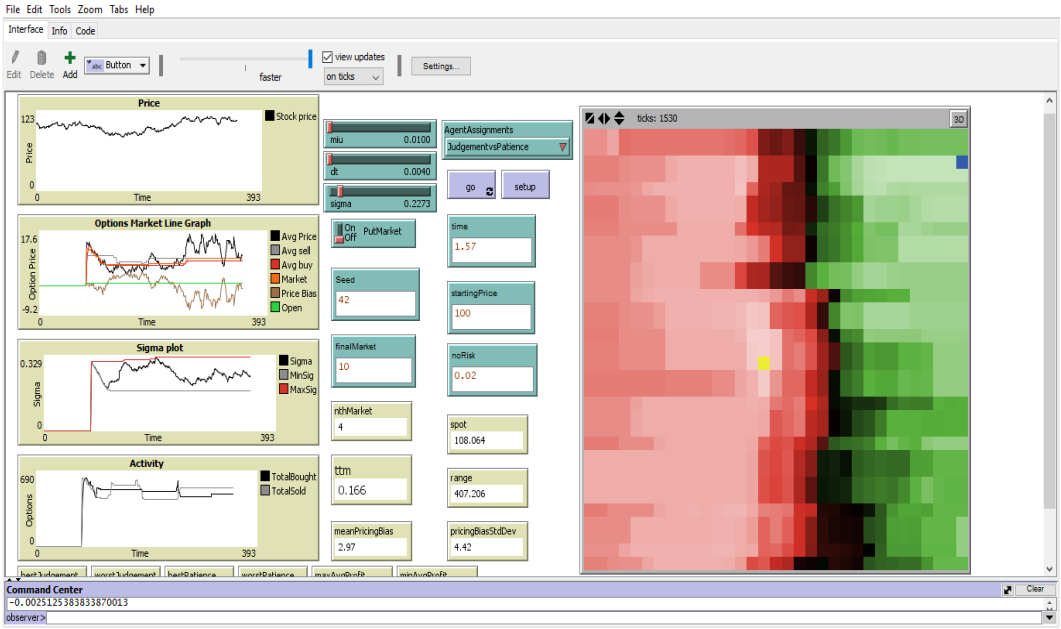


Figure 6.1: NetLogo interface of *OptionsMarket* model by Elliott

compared with the market price, which is estimated as the average between the maximum buy price and the minimum sell price, in order to decide whether to buy or to sell the option. If buy price is higher than market price the trader buys the option, while if sell price is smaller than market price the agent sells the option. The number of options, cash and profit are updated accordingly to their decisions.

Figure 6.1 shows the NetLogo world when market runs. The traders assume colours depending on their profit success: red represents loss, green represents gain. The darker the colour, the closer the traders are to zero profit or zero loss. On the other hand, lighter colours represent the extremes of success or loss. Additionally the program finds the best and the worst values of patience and judgement among the traders: the blue square is the most successful trader, while the yellow square is the least successful trader.

Then, the model evaluates the *pricingBias*, which is a value representing how the market price differs from the option price calculated with the actual volatility (the volatility put in by the user and hidden from the agents) using Black-Scholes formula. Finally, successive markets were aggregated in a Monte Carlo method to converge on coherent results.

Once Elliott's model has been introduced, the subject can be moved onto a deeper explanation. As previously mentioned, the NetLogo pseudo–code has been designed to be easily readable; however it is very important to analyse each row to get a full comprehension of the model. At the beginning of the pseudo–code there

are the *globals* variables, namely variables which are accessible by any agent and can be used anywhere in the model.

```
globals [
          dS
          dW
          ttm
          retVal
          retList
          marketPrice
          Overalld1
          Overalld2
          pricingBias
          truePrice
          marketDayCount
          strike
          marketOpen
          totalBought
          totalSold
          range
          nthMarket
          spot
          maxTotalProfit
          minTotalProfit
          maxAvgProfit
          minAvgProfit
          bestJudgement
          worstJudgement
          bestPatience
          worstPatience
          pricingBiasStdDev
          meanPricingBias
          pricingBiascount
          totalPricingBias
          totalPricingBiassq ]
```

Conversely, the next rows show what are the characteristic variables of any single patch (remember that in this model the agents are patches):

```
patches-own [
              patience
              judgement
              guessed-sigma
              price
```

```
Sell
Buy
maxSigma
minSigma
money
options
d1
d2
callPrice
totalProfit
totalProfitsq
stdDevProfit
maxProfit
minProfit
optPrice ]
```

Both *globals* and *patches-own* variables can only be used at the beginning of a program, before any function definition, and they can be added through a switch, slider, chooser or input box included in the interface. In each NetLogo model there is always a *setup* button with its own code: it initializes some variables and set–up the agents before starting.

```
to setup
  clear-all
  if seed != 0 [ random-seed seed ]
  set ttm -1
  ask patches [ setup-patches ]
  ask patches [ set totalProfit 0 ]
  ask patches [ set maxProfit 0 ]
  ask patches [ set minProfit 0 ]
  set totalPricingBias 0
  set pricingBiascount 0
  set totalPricingBiassq 0
  set maxTotalProfit 0
  set minTotalProfit 0
  set maxAvgProfit 0
  set minAvgProfit 0
  reset-ticks
end
```

The *setup* procedure includes the *setup-patches* procedure, which is responsible for the assignment of the characteristics of judgement and patience to each agent. By switch the user can attribute the feature of *JudgementvsPatience* or *Scattered* to the global variable *AgentAssignments*: choosing the first option, judgement and

patience are assigned to the patches, following the x and y axis respectively, as shown in Figure 6.2 (the agent in coordinates (2,1) has judgement and patience smaller than those of the agent in (4,6)); otherwise, the distinctive features are allotted in a random way. Following the code of this procedure:

```
to setup-patches
   if AgentAssignments = "Scattered" [
      set patience minPatience + random (maxPatience - minPatience)
      set judgement minJudgement + random-float
                                    (maxJudgement - minJudgement) ]
   if AgentAssignments = "JudgementvsPatience" [
      set judgement minJudgement +
        ((maxJudgement - minJudgement) *
        ((pxcor - min-pxcor) / (max-pxcor - min-pxcor)))
      set patience minPatience +
        ((maxPatience - minPatience) *
        ((pycor - min-pycor) / (max-pycor - min-pycor))) ]
end
```

The main part of each program is the procedure *go*: the related button launches the simulation and other main functions, which are the core of the model. This procedure calculates profit and other variables associated with it (as minimum and maximum profit): if the number of the markets performed is less than the *finalMarket*, namely the number of market to run, and if the time to maturity (*ttm*) is negative, the program asks each trader to calculate its own profit and its standard deviation, which helps to evaluate the minimum and maximum profit.

```
to go
  ifelse ((nthMarket - 1) < finalMarket) [
  if ttm <= 0 [
  ask patches [
      let profit (money + (options * marketPrice))
      set totalProfit (totalProfit + profit)
      set totalProfitsq (totalProfit * totalProfit)
      if nthMarket > 0
       [ set stdDevProfit sqrt( (TotalProfitSq / nthMarket) -
             ((TotalProfit / nthMarket) * (TotalProfit / nthMarket)) ) ]
      set maxProfit (totalProfit + stdDevProfit)
      set minProfit (totalProfit - stdDevProfit)
      ]
```

Proceeding in the analysis about *go* procedure, the patches get now asked to work out the *pricingBias* and its standard deviation, which represents, as stated before, how far off the market price is from the price calculated with the actual volatility.

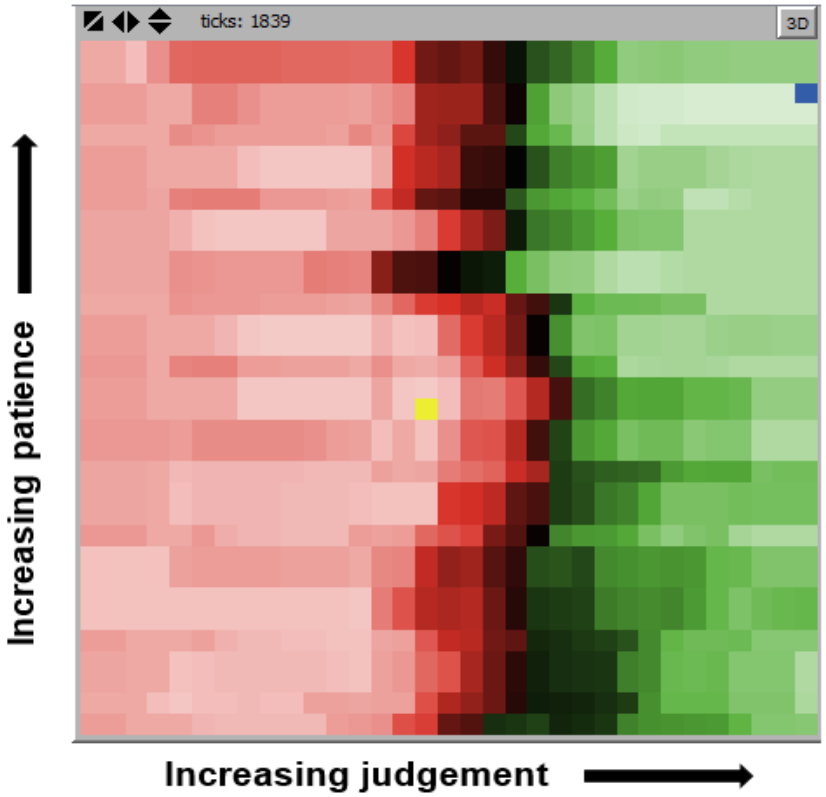


Figure 6.2: Plot of judgement versus patience

```
if nthMarket > 0
  [ set range (max [ abs (totalProfit / nthMarket) ]
        of patches) * 1.10 + 10 ]
    set totalPricingBias (totalPricingBias + pricingBias)
    set totalPricingBiassq (totalPricingBiassq +
                            (PricingBias * PricingBias))
    if pricingBiasCount > 0
    [ set meanPricingBias (TotalPricingBias / pricingBiasCount)
      set pricingBiasStdDev sqrt(totalPricingBiasSq /
        pricingBiasCount - meanPricingBias * meanPricingBias) ]
      set pricingBiasCount (pricingBiasCount + 1)
```

Now the best and worst values of judgement and patience are decided according to the total profit of single agents. Furthermore, if the number of markets run are positive, the program sets up variables associated to total profit.

```
set bestJudgement [ judgement ] of max-one-of patches [ totalProfit ]
```

```
  set worstJudgement [ judgement ] of min-one-of patches [ totalProfit ]
  set bestPatience [ patience ] of max-one-of patches [ totalProfit ]
  set worstPatience [ patience ] of min-one-of patches [ totalProfit ]

  if (nthMarket > 0)
   [ set maxTotalProfit (max [ totalProfit ] of patches ) / nthMarket
     set minTotalProfit (min [ totalProfit ] of patches ) / nthMarket
     set maxAvgProfit (maxTotalProfit / nthMarket)
     set minAvgProfit (minTotalProfit / nthMarket)
     ]
```

Finally, the procedure is completed by updating the number of market's executions and by defining three new procedures: *plot-profit*, *setup-market*, *run-market*.

```
  ask patches [ plot-profit ]
      setup-market
      set nthMarket (nthMarket + 1)
       ]
   run-market
   tick
]
   [ stop ]
end
```

The *plot-profit* procedure defines the patches colour according to profit: if the ratio between *totalProfit* and *nthMarket* is positive, namely if agent is gaining money by trading, the associated patch becomes green, vice versa if the ratio is negative, the patch becomes red. As previously explained, patches can get lighter or darker colours, respectively if they are on the extremes of gain and loss or if they are close to zero profit and zero loss. The command *scale-color* allows NetLogo to assign right colours at each agent, based on the instructions above. At last, the trader with best judgement and patience is set blue, while the one with worst judgement and patience is set yellow.

```
  to plot-profit
    if nthMarket > 0 [ let avgProfMag abs (totalProfit / nthMarket)
    if (totalProfit / nthMarket) > 0 [
    set pcolor scale-color green avgProfMag 0 (range) ]
    if (totalProfit / nthMarket) <= 0 [
    set pcolor scale-color red avgProfMag 0 (range) ] ]
    if (patience = worstPatience) [ if (judgement = worstJudgement)
     [ set pcolor yellow ] ]
    if (patience = bestPatience) [ if (judgement = bestJudgement)
     [ set pcolor blue ] ]
  end
```

Another function launched in *go* is the *setup-market* procedure. In this section some variables are initialized and the four graphs are set up. The spot and strike prices are set equal to the *startingPrice*, decided by the user into the interface, as well as the time to maturity. At each time step (dt) the spot price is updated by Gaussian Brownian motion, while strike price remains constant throughout the whole simulation. The program creates and initializes a list named *retList*, in which the spot price updated is placed after each *dt*.

```
to setup-market
  set totalBought 0
  set totalSold 0
  set marketOpen 0
  set marketPrice 0
  ask patches [ set callPrice 0 ]
  set pricingBias 0
  set-current-plot "Price"
  clear-plot
  set-plot-x-range 0 ceiling (time / dT)
  set-current-plot "Options Market Line Graph"
  clear-plot
  set-plot-x-range 0 ceiling (time / dT)
  set-current-plot "Sigma plot"
  clear-plot
  set-plot-x-range 0 ceiling (time / dT)
  set-current-plot "Activity"
  clear-plot
  set-plot-x-range 0 ceiling (time / dT)
  set marketDayCount 0
  set strike startingPrice
  set spot startingPrice
  set retList [ 0 0 ]
  set retList lput spot retList
  set ttm time
  ask patches [
  set minSigma 0.0
  set maxSigma 0.0
  set Buy 0
  set Sell 0
  set money 0
  set options 0
  set optPrice 0
 ]
end
```

The last procedure included in *go* is *run-market*. In turn, *run-market* contains

other procedures which update spot price, buy and sell prices, calculate market and options prices and simulate buying and selling. Furthermore, the *pricingBias* is worked out as the difference between market price and *truePrice*(the call or put price calculated by Black-Scholes formula using the volatility set by the user) and adjust the volatility values of the patches (*guessed-sigma*).

```
to run-market
   set totalBought 0
   set totalSold 0
   update-spot
   update-patchPrices
   if marketOpen = 1 [ set-marketPrice ]
   buyAndSell
   if putMarket = false [ set truePrice calculate-callPrice sigma ]
   if putMarket = true [ set truePrice calculate-putPrice sigma ]
   if marketOpen = 1 [ set pricingBias (marketPrice - truePrice) ]
   set ttm ttm - dt
   set marketDayCount (marketDayCount + 1)
   if marketOpen = 0 [
   if marketDayCount > (max [ patience ] of patches) + 40 [
   set marketOpen 1
   ask patches [ set maxSigma guessed-sigma ]
   ask patches [ set minSigma guessed-sigma ]
    ]
    ]
end
```

To update the spot price, Elliott's model employs the Gaussian Brownian motion. Shown below the pseudo–code of the procedure *update-spot*, in which also the spot list *retList* is updated:

```
to update-spot
  set dW random-normal 0 sqrt(dt)
  set dS spot * (miu * dt + sigma * dW) ;'miu'= drift coefficient
  set spot spot + dS
  set retVal dS / spot
  set retList lput retVal retList
end
```

In a real options market there are different types of traders, each one mainly characterised by a supposed volatility. Elliott tries to reproduce this essential feature with patches which hypothesize their own volatility, according to judgement and patience. The procedure *calculate-sigma* estimates the *guessed-sigma* of each single agent with an algorithm: greater the personal judgement, greater the sigma; additionally, if an agent has a high value of patience, more elements of spots list

(*retList*) are included in the of standard deviation and volatility increases. Indeed, an agent with a higher value of patience takes into account the past more than an agent with a smaller value of patience.

```
to calculate-sigma
   if length retList > patience + 2
    [ set guessed-sigma ((standard-deviation sublist retList
                         (length retList - patience)
                         (length retList)) / (sqrt(dt)) * judgement) ]
end
```

Let's now define functions which work out the put and call options prices, using Black–Scholes formula. These procedures are employed for the calculation of *Buy* and *Sell* prices and even for the *truePrice*, as mentioned before. As shown in the code below, *calculate-callPrice* and *calculate-putPrice* are ″report procedures‟, that are procedures which contain in their body the command *report* to report a value of the procedure. After the name of the function, between square brackets, there is the argument: in this case it is volatility, which could be the *sigma* or the *guessed-sigma*, if procedure computes *truePrice* or *Buy* and *Sell* prices respectively.

```
to-report calculate-callPrice [ _sigma ]
  let _d1 calculate-d1 _sigma
  let _d2 calculate-d2 _d1 _sigma
  let _callPrice spot * normcdf(_d1) - strike *
                 exp( - noRisk * ttm) * normcdf(_d2)
  report _callPrice
end

to-report calculate-putPrice [ _sigma ]
  let _d1 calculate-d1 _sigma
  let _d2 calculate-d2 _d1 _sigma
  let _putPrice strike * exp( - noRisk * ttm) * normcdf(- _d2) -
                spot * normcdf(- _d1)
  report _putPrice
end
```

To calculate put and call prices, Black-Scholes formula needs the strike, spot, risk-free interest rate, time to maturity, volatility and the quantities *d1* and *d2* (see the Chapter 2 on Options Theory). To be more precise, it needs the normal distribution cumulative function of *d1* and *d2*. The codes of these report–procedures are posted here:

```
to-report calculate-d1 [ _sigma ]
  let _d1 ((ln (spot / strike)) + (noRisk + (_sigma * _sigma) / 2) * ttm) /
          (_sigma * sqrt(ttm))
```

```
  report _d1
end

to-report calculate-d2 [ _d1 _sigma ]
  let _d2 (_d1 - (_sigma * sqrt(ttm)))
  report _d2
end
```

Normal distribution cumulative density function code:

```
to-report normcdf [lclx]
  let lclt lclx
  let y 0.5 * erfcc ((-1) * lclt / ( 1 * sqrt 2.0))
  if ( y > 1.0 ) [ set y 1.0 ]
  report y
end
```

Normal distribution probability density function code:

```
to-report normpdf [lclx]
  report 1 / ( sqrt (2 * pi)) * exp ( - lclx * lclx / 2.0)
end
```

Complementary error function code:

```
to-report erfcc [lclx]
  let z abs lclx
  let lclt 1.0 / (1.0 + 0.5 * z)
  let r lclt *  exp ( - z * z - 1.26551223 + lclt *
       (1.00002368 + lclt * (0.37409196 + lclt *
       (0.09678418 + lclt * (-0.18628806 + lclt *
       (.27886807 + lclt * (-1.13520398 + lclt *
       (1.48851587 + lclt * (-0.82215223 + lclt * .17087277 ))))))))))
   ifelse (lclx >= 0) [ report r ] [report 2.0 - r]
end
```

Now, if the market is open, the program calculates and updates buy and sell prices for patches, using the procedures analysed above. The trader supposes its own volatility, then if the *guessed-sigma* is greater than *maxSigma*, it evaluates its *Sell* price (price a trader is willing to cash in for the option), while if the *guessed-sigma* is smaller than *minSigma* it computes its *Buy* price (price a trader is willing to pay for an option).

```
to update-patchPrices
  ask patches [ calculate-sigma
   if marketOpen = 1 [
```

```
    if guessed-sigma > maxSigma
      [ set maxSigma guessed-sigma
       if putMarket = false [ set Sell calculate-callPrice guessed-sigma ]
       if putMarket = true [ set Sell calculate-putPrice guessed-sigma ] ]

    if guessed-sigma < minSigma
      [set minSigma guessed-sigma
       if putMarket = false [ set Buy calculate-callPrice guessed-sigma ]
       if putMarket = true [ set Buy calculate-putPrice guessed-sigma ] ]
       if putmarket = false [ set optPrice calculate-callPrice guessed-sigma ]
       if putmarket = true [ set optPrice calculate-putPrice guessed-sigma ]
 ] ]
end
```

The market price is easily calculated as the average between the highest buy and lowest sell prices of the patches.

```
to set-marketPrice
 let maxBuy max [ Buy ] of patches
 let minSell min [ Sell ] of patches
 set marketPrice (maxBuy + minSell) / 2
end
```

Eventually, when program gets all of the values it needs, the market is open and the *retList* is long enough, the traders compare their own sell and buy prices with the market price, in order to decide whether to buy or to sell the option. If *Buy* is greater than *marketPrice*, they buy the option, otherwise they sell it if the *Sell* price is lower than *marketPrice*. Once the trading succeeds, the number of options and quantity of money are updated.

```
to buyAndSell
 if marketOpen = 1 [
 ask patches [
  if length retList > patience + 2 [
   if Buy >= marketPrice [
    set options options + 1
    set money money - marketPrice
    set totalBought totalBought + 1
    ]
   if Sell <= marketPrice [
    set options options - 1
    set money money + marketPrice
    set totalSold totalSold + 1
```

```
    ]
   ]
  ]
 ]
end
```

This process is executed at each time step for a number of times equal to the *nthMarket*, chosen by user at interface.

6.2 OptionsMarket pricing via BS

Now that the reference model has been explained, the subject can be moved onto optimisation of the model and how it has been modified to meet the request of this work. The first model implemented is simply a modification of the Elliott's model in which prices are still evaluated via Black–Scholes formula, but agents are now divided into intelligent and random traders in order to simulate a more real options market with high volatility.

The *OptionsMarket* model shows all intelligent agents, endowed with judgement and patience and able to compute their own volatility. Actually, in real markets, there are agents who trade wisely and others who operate without experience or consideration. To reproduce this situation it is necessary to embody agents as turtles and no more as patches. Indeed, it is not possible create different types of agents if they are patches.

In Figure 6.3, the world of the modified model shows two different types of agents: persons and sheep. The first one are the intelligent traders, who buy and sell options according to their own judgement and patience. The sheep are naïve traders: they behave in a random way because they have not individual characteristics. The buy and sell procedure is the same as the Elliott's model, but the volatilities of random agents are random numbers between zero and one (it is no more updated following judgement and patience, as for intelligent agents).

As previously mentioned, the agents are now turtle–agents and they belong to two different classes. The keyword *breed*, at the beginning of the program, defines these categories: *PGAgents* and *randomAgents*, intelligent and naïve respectively. The code below shows the turtles–variables of these two groups, while the globals remain the same as before:

```
breed [randomAgents randomAgent]
breed [PJAgents PJAgent]

PJAgents-own [
```

Figure 6.3: NetLogo's world populated by *PJAgents* (person) and *randomAgents* (sheep)

```
patience
judgement
guessed-sigma
price
Sell
Buy
maxSigma
minSigma
money
options
d1
d2
callPrice
totalProfit
totalProfitsq
stdDevProfit
maxProfit
```

```
                minProfit
                optPrice ]

randomAgents-own [
                    guessed-sigma
                    price
                    Sell
                    Buy
                    maxSigma
                    minSigma
                    money
                    options
                    d1
                    d2
                    callPrice
                    totalProfit
                    totalProfitsq
                    stdDevProfit
                    maxProfit
                    minProfit
                    optPrice ]
```

NetLogo's world shows now agents having shaped of persons and sheep arranged tidily on a blue background, made by patches. When the user presses the button *setup*, the program creates a number (*nrandomAgents*, select on the interface) of *randomAgents*, gives their the shape of sheep and arranges this set in order to cover all the world.

```
ask patches [set pcolor 92]
  create-randomAgents nRandomAgents
  let side sqrt nRandomAgents
  let step world-width / side
  ask randomAgents
  [ set shape "sheep"
  set size 2
  set color gray]

  let n 0
  let x step / 2
  let y step / 2

  while [n < nRandomAgents]
  [
  ask randomAgent n [setxy x y]
```

```
  set x x + step
  if x > world-width [set x step / 2
  set y y + step]
  if y > world-width [set y 0.20 + step / 2]
  set n n + 1
  ]
```

Now the world is populated only by naive traders. The easiest way to create intelligent traders is the following: the user has to select a number of *PJAgents* (*nPJAgents*) and then a set of *randomAgents*, equal to this number, is converted into *PJAgents*.

This set of turtles has patience and judgement, which are assigned at each agent in a random way (there is not relation between the turtle's position and the x–y axis).

```
ask n-of nPJAgents randomAgents
 [set breed PJAgents
  set shape "person"
  set color gray
  set patience minPatience + random (maxPatience - minPatience)
  set judgement minJudgement + random-float (maxJudgement - minJudgement)
 ]
```

As patches have been turned into turtles, it is necessary that all the *ask* commands are written as ″ask turtles [ ] ″. In order to simplify the code, the program puts together the two groups of agents in one set, named *tradingAgents*, in this way:

```
set tradingAgents (turtle-set randomAgents PJAgents)
```

The *go* procedure and all the other procedures which are involved in the updating of prices and in the buying and selling procedure, are the same of the original model.

The main difference is the calculation of the *guessed-sigma*. As the pseudo–code below shows, the individual volatility is evaluated in two different ways: if the trader is a *randomAgent*, the *guessed-sigma* is set equal to a random number between zero and one (in order to reproduce the randomness by which a naïve agent trades). On the other hand, if it is a *PJAgent* the volatility is worked out with the algorithm already explained above, based on patience and judgement.

```
to calculate-sigma
  ifelse breed = randomAgents [ set guessed-sigma random-float 1]
                              [ if length retList > patience + 2
                              [ set guessed-sigma ((standard-deviation
                              sublist retList (length retList - patience)
```

```
                                    (length retList)) / (sqrt(dt)) * judgement)
                                    ]]
end
```

As mentioned above, the addition of random agents into the model implies that the simulated market shows now a high volatility. In presence of randomness, the assumptions of Black–Scholes are not more verified and the formula does not generate correct results. However, the adjusted *OptionsMarket* model just explained is the base model of this project: it will be optimised in order to implement neural networks for the options prices prediction and subsequently a stocks market for the simulation of underlying's price evolution.

6.3 OptionsMarket mimicking BS via Neural Networks

The second designed model is completely equal to that described in the previous section, with the difference that options prices will no longer be evaluated via Black–Scholes formula but through neural networks.

First purpose of the work is compare the classic Black–Scholes method results with those of artificial neural networks in a market with high volatility, due to random traders. The idea is to substitute the traditional method with multilayer perceptrons, suitably trained with data resulting from previous BS evaluations. This means that neural networks do not minimise the error of BS pricing in presence of randomness, but this step is needed to verify that neural networks can be a good alternative for options pricing. Once verified the accuracy and effectiveness of this method, the model will be optimised in order to obtain more correct result.

Two ways have been used to achieve it. Initially, only one neural network for all agents has been trained with global data: this means that each agent evaluates its own *Buy* or *Sell* price according to its own volatility, but the neural network which provides the price prediction does't have an individual experience. In order to attain this singular experience, it has been necessary create a number of artificial neural networks equal to the number of agents in the market: in this way each trader is able to predict its own price, differently to the others. Obviously, the second method requires that each neural network must be trained with inputs and output relative to only one agent.

The learning algorithm, as the training parameters (error threshold, number of hidden neurons, activation function, number of repetition, etc.), will be the same if the model employs one or more networks; what changes is how the training data must be collected.

As previously mentioned, these neural networks are implemented in R pseudo–code,

in particular using the package *neuralnet* (see Subsection 5.4.1). In order to insert neural networks in the NetLogo simulation model, it is necessary to create a link between R and NetLogo. The NetLogo-*Rserve*–extension provides primitives to use the statistical software R via the *Rserve* package (based on TCP/IP connection) within a NetLogo model. See more in Appendix B.

6.3.1 One Neural Network

First, consider the model with only one neural network. As stated before, the dataframe for training is made by inputs and outputs of different traders. In particular, in order to shorten learning time, only a percentage of agents, chosen randomly, contributes to the lists formation. These lists, which form the training dataframe, are updated each time step *dt*, during the first market run (only when *nthMarket = 1*). Since the purpose of the network is to predict Black–Scholes price, the learning inputs are spot prices, time to maturity and volatilities of traders (*guessed-sigma*), while the outputs target are, obviously, BS prices. Strike price and risk–free interest rate remain constant during all markets, then it is not necessary to use them as inputs of the network.

First of all, it is necessary a correlation analysis between those variables used for neural networks training, in order to verify that all inputs are correlated (positively or negatively) with the output. The analysis was done using software R and its function *cor*, which evaluate the correlations between all data presented.

	spot	**sigma**	**ttm**	**OptionPrice**
spot	1.00000000	−0.02156932	−0.81366702	0.5225142
sigma	−0.02156932	1.00000000	0.04030424	0.7250344
ttm	−0.81366702	0.04030424	1.00000000	−0.2723905
OptionPrice	0.5225142	0.7250344	−0.2723905	1.00000000

Table 6.2: Correlation analysis between input and output data for the training of neural network

Table 6.2 shows that all input variables are properly correlated with output; this means that spot price, sigma and time to maturity are correct and necessary data to evaluate option price. This fact was already known from the study of Black–Scholes formula, but a correlation analysis is always required before any prediction. It should be noted that sigma is low correlated with spot prices and

time to maturity. This is not a mistake, as *guessed-sigma* is computed randomly by *randomAgents*, while *PJAgents* evaluate *guessed-sigma* according patience and judgement which are not correlated with spot prices or time to maturity.

After verifying the correlation between data, it proceeded with writing code. Here below the pseudo–code regarding the lists updating:

```
if marketOpen = 1 [
  if  nthMarket = 1 [ let l 0
      while [ l < (percentage * nRandomAgents / 100 ) ]
       [ let o random nRandomAgents
        ask turtle o [set spotList lput spot spotList
         set ttmList lput ttm ttmList
         set gsigList lput guessed-sigma gsigList
         output-train
         set outputList lput output outputList]
         set l l + 1 ]
    ] ]

  to output-train
  if putMarket = false [ set output calculate-callPrice guessed-sigma ]
  if putMarket = true  [ set output calculate-putPrice guessed-sigma ]
end
```

As previously mentioned, the training period involves only the first market run and collection of data begins only if the market is open.

Summarizing, the four lists are the following:

- *spotList* is made by spot prices updated with Gaussian Brownian Motion (see in section 7.1 the code of *update-spot* procedure);
- *ttmList* is the list of time to maturity;
- *gsigList* is the list of *guessed-sigma*, namely the volatility calculated by single agent;
- *outputList* is the list of call or put prices evaluated by single agent with BS method.

The *percentage* must be chosen by user from interface, as the number of random agents (*nRandomAgents*).

The idea is to use the first market run to collect the data and to train the neural network, in order to substitute BS formula with neural network prediction from the second market execution. Therefore, at the end of first market, the program calls two procedures: *make-datatrain*, which scale the data and assemble the dataframe, and *train*, which train the network using *neuralnet* R package.

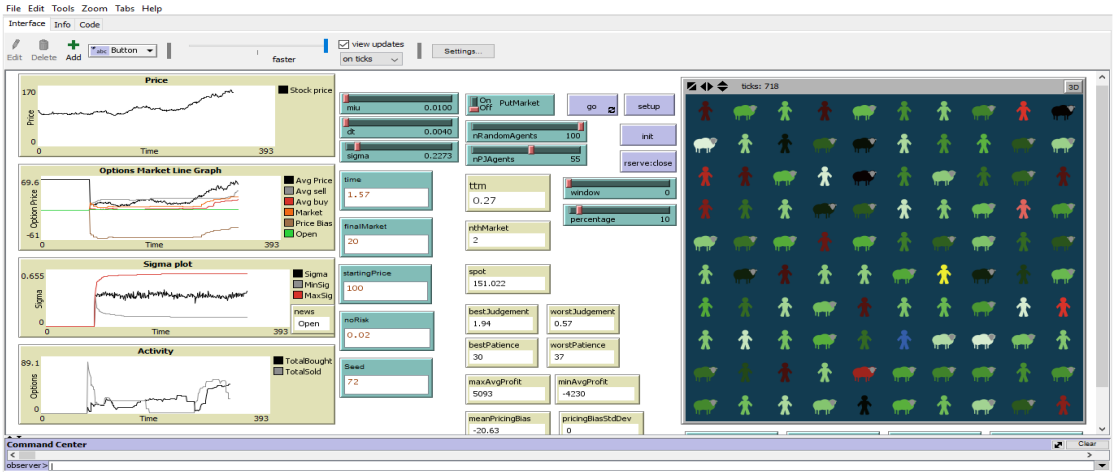


Figure 6.4: Interface of the *OptionMarket* model with one neural network

```
if nthMarket = 1 [if ttm <= ( time - (dT * (ceiling ( time / dT ))) )
 [ make_datatrain
   train ]
]
```

As described in Section 5.4.1 , *neuralnet* package allows hidden layers of neurons, trains the network using backpropagation algorithm and, most important, has lower possibilities to find local minima. *neuralnet* requires a dataframe (not a matrix) of inputs and outputs, with their own labels. The outputs must be scaled, while it is not necessary for inputs. However, to shorten the training period, also the inputs have been scaled with the same method used for scaling output. The scaling procedure has been made in R, using ″min–max‟method.

Before making up the dataframe, each list has been sent to R and all elements of them have been converted to numeric values with *as.numeric* function, just to be safe that R reads only numbers (not characters).

Finally, the dataframe has been composed using *cbind* R function and each columns have been scaled with min–max method. All minimums and maximums must to be saved in NetLogo for later.

```
to make_datatrain

  rserve:put "spotList" spotList
  rserve:put "gsigList" gsigList
  rserve:put "ttmList" ttmList
  rserve:put "outputList" outputList

  rserve:eval "S<-as.numeric(spotList)"
```

```
  rserve:eval "sigma<-as.numeric(gsigList)"
  rserve:eval "ttm<-as.numeric(ttmList)"
  rserve:eval "OptionPrice<-as.numeric(outputList)"

  rserve:eval "datatrain<-cbind(S, sigma, ttm, OptionPrice)"
  rserve:eval "colnames(datatrain)<-c('S','sigma','ttm','OptionPrice')"

  rserve:eval "maxsd<-apply(datatrain, 2, max)"
  rserve:eval "minsd<-apply(datatrain, 2, min)"
  rserve:eval "scaledtrain<-as.data.frame (scale(datatrain,
               center = minsd , scale = maxsd - minsd))"

  set min1 rserve:get "minsd[1]"
  set max1 rserve:get "maxsd[1]"
  set min2 rserve:get "minsd[2]"
  set max2 rserve:get "maxsd[2]"
  set min3 rserve:get "minsd[3]"
  set max3 rserve:get "maxsd[3]"
  set min4 rserve:get "minsd[4]"
  set max4 rserve:get "maxsd[4]"

end
```

The *train* procedure, called at the end of first market execution, is involved in training and plotting the neural network.

First of all it is necessary to explicate the ″formula‟, which is only the relation between inputs and output target ($Output \sim Inputs$). Then the parameters of the network have to be chosen. After a careful analysis, in which all parameters have been independently modified in order to get the optimal compromise between learning time and error threshold, it was concluded that: the optimal number of neurons in the hidden layer is three (as the number of inputs), the threshold for the partial derivatives of the error function is 0,03 and the number of repetitions ten. Smaller values of *threshold* entail that the algorithm does not converge in more then one repetition.

Below the code of the *train* procedure:

```
to train
  rserve:eval "nn<-neuralnet ( OptionPrice~S+sigma+ttm, scaledtrain,
                               hidden = 3, threshold = 0.03, rep=10)"
  rserve:eval "rn<-which.min(nn$result.matrix[1,])"
  rserve:eval "plot(nn, rep = rn)"
  print rserve:get "rn"
end
```

Once the network has been trained, there are available some information about it, like errors, reached thresholds and repetitions, useful in subsequent commands. Indeed, the program finds the number of the repetition with minimum error and calls it *rn*; this repetition will be used to plot the neural network and to get the prediction.

Now the network has been trained and it is ready for the pricing prediction.

```
if guessed-sigma > maxSigma
[ set maxSigma guessed-sigma
if putMarket = false [ ifelse nthMarket = 1
[set Sell calculate-callPrice guessed-sigma ]
[ set Sell predict spot guessed-sigma ttm ]]
if putMarket = true  [ ifelse nthMarket = 1
[set Sell calculate-putPrice guessed-sigma ]
[ set Sell predict spot guessed-sigma ttm]]
]

if guessed-sigma < minSigma
[set minSigma guessed-sigma
if putMarket = false [ ifelse nthMarket = 1
[set Buy calculate-callPrice guessed-sigma ]
[ set Buy predict spot guessed-sigma ttm ]]
if putMarket = true  [ ifelse nthMarket = 1
[set Buy calculate-putPrice guessed-sigma ]
[ set Buy predict spot guessed-sigma ttm ]]
]
```

The code transcribed above, shows how the program evaluates the options prices: if the market is at its first execution, the prices are calculated with Black–Scholes formula using *calculate-callPrice* or *calculate-putPrice* procedures; instead, from the second execution, the trained neural network becomes operative, in particular the program employs the new procedure *predict* to estimate the options prices.

The procedure *predict* is a to–report procedure. It needs the same inputs used to train the neural network: each time step, all agents call this procedure to compute their own *Buy* or *Sell* prices and they have to send the spot price, the time to maturity and their own volatility. As the inputs of the training dataframe have been scaled, even this variables need to be scaled with min–max method, using for each variable minimum and maximum of the respective column. The inputs have to been collect in form of dataframe.

In order to predict the output, the *neuralnet* package offers the function *compute*. It needs the neural network trained before, the inputs scaled and eventually, the number of the repetition: the program uses the best repetition found during

training. The procedure reports only the output predicted, then it is necessary extract it using the command ″$net.result[1,1]‶. Obviously, the output has to be re-scaled before being reported.

```
to-report predict [-spot -sigma -ttm ]
  let sspot (( -spot - min1 ) / ( max1 - min1 ))
  let ssigma (( -sigma - min2 ) / ( max2 - min2 ))
  let sttm (( -ttm - min3 ) / ( max3 - min3 ))

  (rserve:putdataframe "data" "S" sspot  "sigma" ssigma "ttm" sttm )
  rserve:eval "nprice<-compute(nn, data, rep = rn)$net.result[1,1]"
  set nprice rserve:get "nprice"
  let -nnprice  nprice * ( max4 - min4 ) + min4
  report -nnprice
end
```

6.3.2 More Neural Networks

As described in previous subsection, next step provides to create a number of artificial neural networks equal to the number of agents in the market. The idea is to make the model closer to the reality, by integrating the heterogeneity of agents decisions in options trading.

Differently from the model with only one neural network, now each turtle has to save its own data in lists after each time step. Therefore, each single agent collects spot prices, time to maturity, volatility and option prices in lists during the first market run and, when the execution comes to the end, creates its own dataframe for the training step.

```
if nthMarket = 1 [
if ttm <= ( time - (dT * (ceiling ( time / dT ))) )
             [ ask tradingAgents [ make_datatrain
                                   train ] ]
                    ]
```

The heterogeneity is given also by different experience in pricing: an agent with more data has more experience and estimates the option price more accurately than one with less data and therefore with lower experience. With the aim of reproducing this situation, the experience has been modelled with the length of the training lists. The idea is to assign to each agent a random number between one and the value corresponding to the length of the list, so that the new training lists begin from the element which has the position corresponding to this number. The expedient just explained allows to obtain a simulation in which the agents begin to collect data, and then to mature experience, in

different moments during market run. The pseudo-code below shows the *make-dataframe* procedure modified.

```
to make_datatrain

 rserve:put "spotList" spotList
 rserve:put "gsigList" gsigList
 rserve:put "ttmList" ttmList
 rserve:put "outputList" outputList
 rserve:eval "S<-as.numeric(spotList)"
 rserve:eval "sigma<-as.numeric(gsigList)"
 rserve:eval "ttm<-as.numeric(ttmList)"
 rserve:eval "OptionPrice<-as.numeric(outputList)"
 rserve:eval "f<-length(S)"
 rserve:eval "a<-sample(1:f, 1)"
 rserve:eval "S<-S[a:f]"
 rserve:eval "sigma<-sigma[a:f]"
 rserve:eval "ttm<-ttm[a:f]"
 rserve:eval "OptionPrice<-OptionPrice[a:f]"
 show rserve:get "length(S)"
 show rserve:get "length(sigma)"
 show rserve:get "length (ttm)"
 show rserve:get "length (OptionPrice)"

 rserve:eval "datatrain<-cbind(S, sigma, ttm, OptionPrice)"
 rserve:eval "colnames(datatrain)<-c('S','sigma','ttm','OptionPrice')"

 rserve:eval "maxsd<-apply(datatrain, 2, max)"
 rserve:eval "minsd<-apply(datatrain, 2, min)"
 rserve:eval "scaledtrain<-as.data.frame (scale(datatrain,
                                          center = minsd ,
                                          scale = maxsd - minsd))"
 print rserve:get "scaledtrain"

 set min1 rserve:get "minsd[1]"
 set max1 rserve:get "maxsd[1]"
 set min2 rserve:get "minsd[2]"
 set max2 rserve:get "maxsd[2]"
 set min3 rserve:get "minsd[3]"
 set max3 rserve:get "maxsd[3]"
 set min4 rserve:get "minsd[4]"
 set max4 rserve:get "maxsd[4]"

end
```

In order to assign one neural network to each agent, in the *train* procedure (called at the end of first market run by each agent) each neural network has been named *nn* plus the number of the turtle involved in the training. *Who* refers to the indicative number of the turtle.

```
to train

 rserve:eval "nn<-neuralnet ( OptionPrice~S+sigma+ttm,
                              scaledtrain, hidden = 3,
                              threshold = 0.03, rep=10)"
 rserve:eval "rn<-which.min(nn$result.matrix[1,])"
 rserve:eval "plot(nn, rep = rn)"
 rserve:put  "myNum" who
 rserve:eval "name<-paste('nn',myNum,sep='')"
 rserve:eval "assign(name,nn)"
 rserve:eval "print(name)"
 rserve:eval "print(get(name))"
 set rn rserve:get "rn"

end
```

rn refers again to the number of repetition with minimum error. However, it becomes a *turtles-own* variable as agents have to save this parameter for later, in particular for *predict* procedure.

The prediction works as before: each time a trader has to evaluate its *Buy* or *Sell* price, it calls the *predict* procedure, it sends spot price, time to maturity and its volatility (properly scaled), then the function reports the option price predicted with the neural network. The change is in the neural network used: now the traders employ the neural network trained with its own data, it means that each trader evaluates the option prices according to its own intelligence and experience and independently from the data of other traders.

In terms of pseudo–code, when an agent calls the *predict* procedure, NetLogo sends to R the number of best repetition (*rn*) and the distinctive number of the relative agent (*who*). Therefore, the program calls the neural network of the agent in order to start the price prediction.

Here below the modified pseudo–code of *predict* procedure.

```
to-report predict [-spot -sigma -ttm ]

 let sspot (( -spot - min1 ) / (max1 - min1 ))
 let ssigma (( -sigma - min2 ) / ( max2 - min2 ))
 let sttm (( -ttm - min3 ) / ( max3 - min3 ))

 rserve:put "rn" rn
```

```
 rserve:put "myNum" who
 rserve:eval "myNn<-paste('nn',myNum,sep='')"
 rserve:eval "nn <- get(myNn)"
 rserve:eval "print(nn)"

 let test rserve:get "myNn"
 show test
 show rn
 rserve:eval "tt<-names(nn)[13]=='result.matrix'"
 show rserve:get "tt"

 (rserve:putdataframe "data" "S" sspot  "sigma" ssigma "ttm" sttm )
 rserve:eval "nprice<-compute(nn, data, rep = rn)$net.result[1,1]"
 set nprice rserve:get "nprice"
 let -nnprice  nprice * ( max4 - min4 ) + min4
 report -nnprice
end
```

Note the following rows of code:

```
rserve:eval "tt<-names(nn)[13]=='result.matrix'"
show rserve:get "tt"
```

They are necessary to test the existence of the neural network. In case of existence, NetLogo shows on Command Centre the word *true*, on the contrary the word *false*; if *false* the simulation has to stop.

6.4 OptionsMarket learning from stock market

The main goal of this work is to find an alternative pricing method to Black–Scholes formula, in a market under high volatility. As seen in Section 2.4, the BS theory is based on different hypothesis, one of them is that the spot price must follow a Gaussian Brownian Motion. However, in real market, the price of the underlying does not follow a GBM; stock prices have a trend driven by transaction of assets among traders.

The idea is to introduce in the option market a continuous double auction model, which provides the simulation of the stock market. Under this assumption, BS formula is not more valid and options prices need to be evaluated through an alternative method. Results of this last market model simulation should be options prices closer to real market prices and less inaccurate than those evaluated via BS formula.

Moreover, always with the aim to obtain results as realistic as possible, actual model will simulate an options market in which agents may simultaneously trade both call and put options. In this way, it will possible compare the tends of these two market options and verify the presence of some symmetry.

This section is organised as follow: first subsection covers the explanation of the Continuous Double Auction (CDA) model, which is the model used to simulate the stock market, then the remaining part is reserved to a detailed description on how the model has been implemented and how it works.

6.4.1 CDA model

The Continuous Double Auction model is one of the common and main financial model for stock market simulations. It is able to reproduce the interactions among traders and provide the price formation of stock.

Almost all automated trading systems are based on CDA procedure: the model consists of agents with divisible cash and indivisible stocks, who can buy or sell stock asset. Formation of stock price depends on both buyers and sellers, and for this reason this type of auction is defined double. The auction is also referred continuous as every agents can set price at any time of market execution.

As in real markets, potential sellers and buyers declares simultaneously their bid and ask prices respectively, which are recorded in two different lists, named *books* (one list for ask prices and one for bid prices). Elements of book of bid prices are sorted by decreasing value, while those of book of ask price are sorted by increasing value. If the first element of bid list is greater than the first element of ask list there is a match, namely a trade occurs.

Terna developed a CDA models with the aim to understand the stock price formation and its trend in a market composed of two class of agents: *randomAgents* and *trendAgents*. *trendAgents* are traders who buy or sell according to the trend of the stock price: if the price goes down they decide to sell stock, conversely if the price goes up they decide to buy stock. On the other hand, the decision process for *randomAgents* to be a buyer or a seller (or eventually pass) is based on extraction of a number.

All agents, except those which pass the turn, place their own bid or ask price according to the last execution price (price at which the asset is traded), that is the price of the period before (t-1). More precisely, the bid or ask price is set equal to the last execution price in addition to a normal random number with zero mean and a standard deviation of one hundred. The matching procedure between buyer and seller is the same seen before: the asset transaction occurs only if the ask price is bigger than the bid price. This means that the agent who is buying the stock pays the ask price, while the agent who is selling the stock receive the bid price.

Interface

Figure 6.5 represents the interface of CDA model, which shows (among other thing) a virtual world in which traders exchange stock assets. Traders may assume different colours, based on their role in the market:

- Green, if agent is a buyer;
- Red, if agent is a seller;

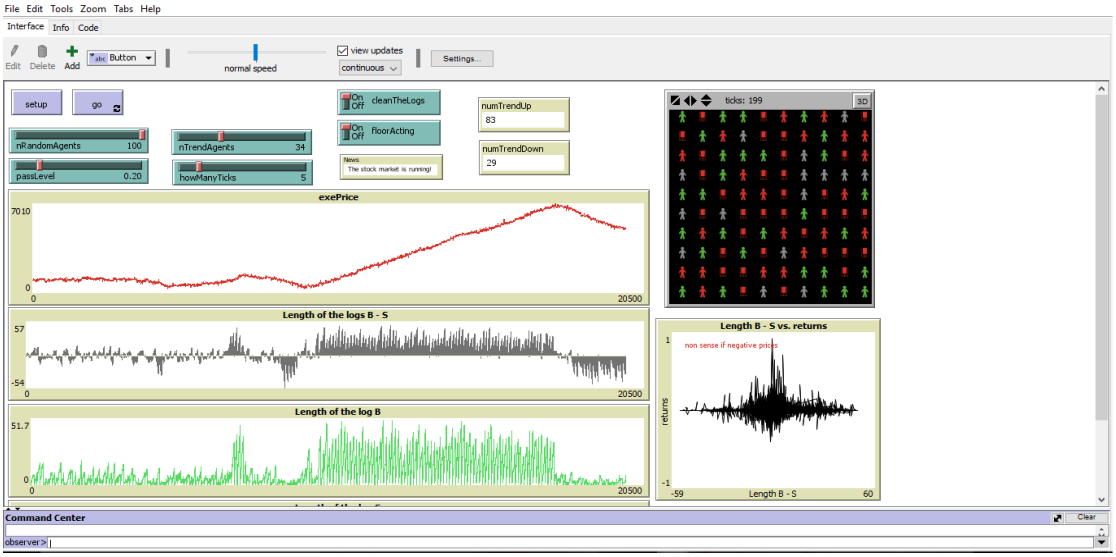


Figure 6.5: Interface of the CDA model designed by Terna.

- Gray, if agent decide to pass.

To launch the simulation, it is required to press two buttons:

- *setup*, to initialize the market and create agents;
- *go*, to start the trading;

In addition to these commands, there are the following variables which user has to choose:

- *nRandomAgents*, number of random agents set through a slider;
- *nTrendAgents*, number of intelligent agents set through a slider;
- *passLevel*, floating number which represents the probability to *randomAgents* to go out from the trading (set through a slider);
- *floorActing*, rules out the possibility to the price to go under a prefixed threshold (if the switch is set *on*);
- *creanTheLogs*, deletes unmatched bid and ask price at the end of trading day (if the switch is set *on*).

6.4.2 Design of the model

NetLogo does not offer the possibility to include more than one world to provide simultaneous simulations in a single model. Consequently, a different strategy is required to

include the stock market in the pre–existing options market model. The easiest way is to add in the same program, thus in the same NetLogo world, agents who exchange stocks. The result is a system composed of agents which trade and exchange options (options market) and other agents who evaluate and trade stocks (stock market). Moreover, unlike the original model where the user had to choose in advance whether to simulate a put or call market, now agents can decide randomly whether to exchange call or put. Thus, the markets of calls and puts are executed simultaneously.

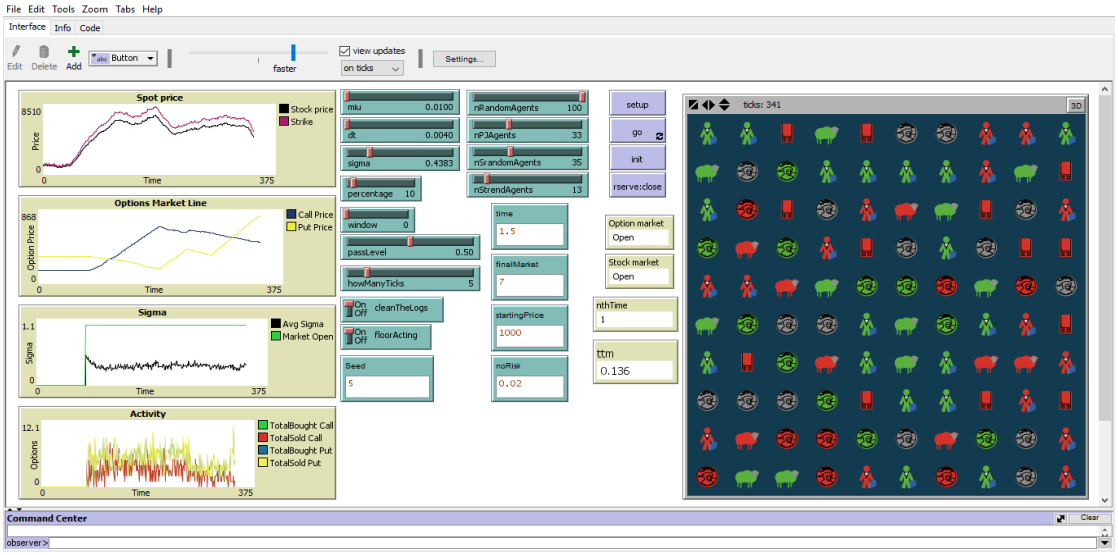


Figure 6.6: Interface of the *OptionsMarket* model with stock market simulation

As in options market, also in stock market, agents are divided in two different classes, according to the decision procedure which establishes their roles within the market (see Section 6.4.1). In order to distinguish the stock agents and their variables to those of options agent, the letter S has been applied at the beginning of each variable of stock market. Thus, the two types of stock agents are named *SrandomAgents* and *StrendAgents*. The variable of stock agents are the same for both classes:

```
SrandomAgents-own[Sbuy Ssell Spass Sprice Scash Sstocks]
StrendAgents-own[Sbuy Ssell Spass Sprice Scash Sstocks]
```

While the global variables to add are (referring to stock market): *SlogB, SlogS, SexePrice, SexePrice-1, StradingAgents, SpriceList, StrendUp, StrendDown, SnumTrendUp, SnumTrendDown, Snews, Snews2.*

The method to create stock agents is the same used to generate agents in options market (see Section 6.2). Number of *SrandomAgents* and *StrendAgents* is selected by the user through sliders from the interface (*nSrandomAgents*, *nStrendAgents*). Random agents are represented by coins and "intelligent" agents by computers; shapes of options agents remain the same as before.

```
create-randomAgents nRandomAgents
 let side sqrt nRandomAgents
 let step world-width / side
 ask randomAgents
 [ set shape "sheep"
   set size 2
   set color gray]

   let n 0
   let x step / 2
   let y step / 2

   while [n < nRandomAgents]
  [
    ask randomAgent n [setxy x y]
    set x x + step
    if x > world-width [set x step / 2
    set y y + step]
    if y > world-width [set y 0.20 + step / 2]
    set n n + 1
  ]

 ask n-of nPJAgents randomAgents
 [ set breed PJAgents
   set shape "person business"
   set color gray
   set patience minPatience + random (maxPatience - minPatience)
   set judgement minJudgement + random-float (maxJudgement - minJudgement)
 ]

 ask n-of nSrandomAgents randomAgents
 [ set breed SrandomAgents
   set shape "coin heads"
   set color gray ]

 ask n-of nSrandomAgents randomAgents
 [ set breed StrendAgents
   set shape "computer server"
   set color gray ]

 set tradingAgents (turtle-set randomAgents PJAgents)
 set StradingAgents (turtle-set SrandomAgents StrendAgents)
```

Notice that, at the beginning of the program, it is required to add the breeds of these new class of agents.

```
breed [SrandomAgents SrandomAgent]
breed [StrendAgents StrendAgent]
```

The program remains unchanged (unless initializations of the new variables [1]) until the procedure *update–turtlesPrices*, which in the original model provided only to evaluate option prices (using BS formula during first market execution and neural networks after). In the current model, such procedure become more complex and detailed, as it represents the main part of both markets:

```
to update-turtlePrices
  stock-market
  if marketOpen = 1  and  nthTime = 1 [ options-market ]
  if marketOpen = 1 and nthTime >= 2 [ options-NNmarket ]

  set strike SexePrice + SexePrice * 0.2
end
```

Following the addition of the stock market, in order to prevent confusion, subsequent executions of markets are nominated *nthTime*, instead of original *nthMarket*.

Notice that the strike price is no more a constant quantity but is updated at the end of each trading day as the 20% greater then the stock price. The reason is that each day an option is exchanged, whose strike must be suitable to the price of the underlying.

With the aim to ease the implementation of the program, it has been assumed that the stock market runs and ends before the beginning of options market, although in reality the two markets are simultaneous. Under this assumption, the agents observe all together and at the same time the execution price of the underlying (*SexePrice*), then they are facilitated. The model employed to simulate the stock market is the CDA model designed by Terna (see Section 6.4.1), and it is call by the procedure *stock-market*, which is describe below.

```
to stock-market

  set SpriceList lput SexePrice SpriceList

  if SexePrice  <= 0 [
  set Snews Snews2
  stop
  ]
```

[1] First stock price is initialised to 1000, while first call and put prices are evaluated using Black–Scholes formula, with spot equal to 1000 and sigma equal to the volatility set by user at the interface.

```
  ask SrandomAgents
  [
   let threshold 0.5
   if floorActing and SexePrice < 500 [set threshold 0.8]
   ifelse random-float 1 < passLevel [set Spass True][set Spass False]

   ifelse not Spass
   [ifelse random-float 1 < threshold [set Sbuy  True set Ssell False]
   [set Ssell True set Sbuy  False] ]
   [set Sbuy False set Ssell False]

  if Spass          [set color gray]
  if Sbuy           [set color green]
  if Ssell          [set color red]
]
```

SrandomAgents are traders with zero intelligence, which decide randomly whether to be buyer or seller, or pass the trading day. If the switch *floorActing* is set On and the price of the stock is low, the random agents have higher probability to become possible buyer. This is a realistic representation of the market, because when the price of an asset goes down it become more attractive and there are more agents who want to buy that asset. On the contrary, *StrendAgents* decide their roles in the market according the tend of stock price: if the trend is increasing the agent decide to be a possible buyer, if the trend is decreasing the agent decide to be a possible seller. The positive or negative direction of the trend is established by looking at the asset price of the preceding trading days.

```
if  length SpriceList >= howManyTicks + 1
 [
  let len length SpriceList
  let i len - 1
  set StrendUp   True
  set StrendDown True
  while [i >= len - howManyTicks]
 [
  if item (i - 1) SpriceList > item i SpriceList [set StrendUp   False]
  if item (i - 1) SpriceList < item i SpriceList [set StrendDown False]
  set i i - 1
  ]
 if StrendUp   [set SnumTrendUp   SnumTrendUp   + 1]
 if StrendDown [set SnumTrendDown SnumTrendDown + 1]

 ]
```

```
ask StrendAgents
 [set Spass True set Sbuy False set Ssell False set color gray
 if StrendUp   [set Spass False set Sbuy  True set color green]
 if StrendDown [set Spass False set Ssell True set color red]
 ]
```

As explained in subsection 6.4.1, each trader set its own ask or bid price equal to the last execution price added to a normal random number.

```
ask StradingAgents [ set Sprice SexePrice + (random-normal 0 100) ]
```

The cleaning operation, *cleanTheLogs*, is set On if a go cycle corresponds to a trading day. In this circumstance, the books of ask and bid are reset before the beginning of the day.

```
if cleanTheLogs[
set SlogB []
set SlogS []
]
```

Agents which not pass, create a temporary list (*tmp*) in which insert their price and label number. If the agent is a possible buyer updates the book of bid prices (*SlogB*), conversely if it is a possible seller updates the book of ask prices (*SlogS*). Successively, book *SlogB* is sorted in order of decreasing price, instead book *SlogS* is sorted in order of increasing price.

Once this process is completed, in order to establish whether the stock may be exchanged, it is required to compare the first item of *SlogB* (the biggest bid price) and that of *SlogS* (the smallest ask price). The stock asset is traded only if the bid price is greater then the ask price. This mean that if the agent is a buyer it pays the ask price, while if the agent is a seller it receives the bid price. This is the process by which the *SexePrice* is defined.

```
ask StradingAgents
[
 set SexePrice-1 SexePrice
 if not Spass
 [
  let tmp[]
  set tmp lput Sprice tmp
  set tmp lput who tmp

 if Sbuy [set SlogB lput tmp SlogB]
```

```
  set SlogB reverse sort-by [item 0 ?1 < item 0 ?2] SlogB

  if (not empty? SlogB and not empty? SlogS) and
  item 0 (item 0 SlogB) >= item 0 (item 0 SlogS)
  [set SexePrice item 0 (item 0 SlogS)
  let agB item 1 (item 0 SlogB)
  let agS item 1 (item 0 SlogS)
  ask turtle agB [ set Sstocks Sstocks + 1
                   set Scash Scash - SexePrice ]
  ask turtle agS [ set Sstocks Sstocks - 1
                   set Scash Scash + SexePrice ]
                   set SlogB but-first SlogB
                   set SlogS but-first SlogS
   ]

  if Ssell [set SlogS lput tmp SlogS]

  set SlogS sort-by [item 0 ?1 < item 0 ?2] SlogS

  if (not empty? SlogB and not empty? SlogS) and
  item 0 (item 0 SlogB) >= item 0 (item 0 SlogS)
  [set SexePrice item 0 (item 0 SlogB)
  let agB item 1 (item 0 SlogB)
  let agS item 1 (item 0 SlogS)
  ask turtle agB [ set Sstocks Sstocks + 1
                   set Scash Scash - SexePrice ]
  ask turtle agS [ set Sstocks Sstocks - 1
                   set Scash Scash + SexePrice ]
  set SlogB but-first SlogB
  set SlogS but-first SlogS
  ]
  show SexePrice
  ] ]
 end
```

In the previous model, during first "Time" execution, option prices were evaluated using Black–Scholes formula in order to create lists of prices used as input for training of neural networks, despite these prices were erroneous (BS formula does not work in a market with high volatility). Now, in addition to the high volatility of the market, the spot price is no longer updated with Gaussian Brownian motion, and consequently BS formula cannot be used to evaluate call and put prices. An alternative method is required for option pricing.

In real markets, options traders buy or sell options based on their own considerations

about future trend of underlying price. In a call options market, an agent who guesses the stock price will increase in time, he decides to buy a call on this asset, conversely an agent who guesses the stock price will decrease, he decides to sell a call option on this asset. The decision process is opposite in a put options market, a pessimistic prediction on stock price will push agents to buy put options on the underlying, on the contrary a optimistic prediction will push them to sell put options. The operation on which the options trade process is based is the same as that used for stock exchange. Each trader places its own buy or sell price in books *logB* and *logS* respectively, and only if buy price is greater then sell price the option will be exchange. Here below the NetLogo code and a more detailed description of the procedure *option–market*, which deals with decisional and trade process.

As mentioned above, options agents have to form their own opinion on the price the underlying will have at maturity (*SexePriceEND*) in order to decide their position in the market. To reproduce the complexity of real world, agents of the model must not follow the stock trend in deterministic way. This means that, for example, only the 70 of total agents agree with the market trend (*agree*), while the remaining 30 are convinced that the price will take the opposite direction (*notagree*). If the trend is positive, the agents *agree* will increase the current stock price by a quantity proportional to the number of days (ticks) before expiration of the option, conversely *notagree* agents will subtract a quantity equal to the product between the number of days before maturity and a normal random number with mean 0 and standard deviation 50. Reverse procedure if the trend is negative.

```
to options-market
  ask tradingAgents [ let f random 10
                      ifelse f > 3 [set agree True set notagree False]
                                   [set notagree True set agree False]
 ]

  ask tradingAgents [
  if StrendUp [ if agree [ if agree [ set SexePriceEND SexePrice
                                          * (1 + (ttm / dt) * ( 0.002))]
                if notagree [ set SexePriceEND SexePrice -
                                   (ttm / dt) * (random-normal 0 50)]]

  if StrendDown [ if agree [ if agree [ set SexePriceEND SexePrice
                                            * (1 - (ttm / dt) * ( 0.002))]
                  if notagree [ set SexePriceEND SexePrice +
                                    (ttm / dt) * ( random-normal 0 50)]]
    ]
```

Before deciding whether to be a buyer or seller, according to the estimated stock market trend, each agent has to choose randomly if deal in puts or calls. All variables related to the call market will be preceded by letter C, on the contrary variables related to put market will be preceded by letter P.

```
ask tradingAgents [let s random 10
                   ifelse s > 5 [set callmarket True set putmarket False]
                                [set putmarket True set callmarket False]
  ]
```

After that, all agents are called to decide their role in the market and set their buy or sell price of the option to insert in the books. If the agent is a call trader, it chooses to become a potential buyer if the call price (*CexePrice*) is much greater than the difference between the estimated value of the underlying at maturity (*SexePriceEND*) and the strike price; vice versa, if the investor has downside expectations, it prefers to become a potential seller of call option. On the contrary, if the agent is a put investor, it chooses to be a possible buyer if the put price (*PexePrice*) is much greater than the difference between the strike price and the estimated value of the underlying at expiration date; vice versa, if the trader has rise expectations, it decides to become a possible seller of put option. Regardless of the market's type, every agent evaluates its own option price by adding a normal random value from the last execution price (same procedure used to evaluate bid and ask price of stock asset).

Contrary on what occurred in original model in which agents are coloured according to their profit, here agents assume colour green if become buyers and colour red if sellers.

Moreover, since agents may decide if trade call or put, they must be able to evaluate both call and put prices, therefore each agent must have two different neural networks which have to be trained with suitable data. Consequently, all the lists that contain data for training are now "agents–own" (also lists of spot and time to maturity) and are two for each training variable: *CspotList*, *CgsigList*, *CttmList*, *CoutputList* for call price learning and *PspotList*, *PgsigList*, *PttmList*, *PoutputList* for put price learning.

```
ask tradingAgents [

  if callmarket [ calculate-sigma
     set CgsigList lput guessed-sigma CgsigList
     set CttmList lput ttm CttmList
     set CspotList lput SexePrice CspotList

     ifelse ( CexePrice + 50 ) < ( SexePriceEND - strike )
             [ set buyer True set seller False]
             [ set seller True set buyer False]

     if buyer [ set price CexePrice + (random-normal 2 2)
                set color green ]
     if seller [ set price CexePrice + (random-normal 2 2)
                 set color red ]

     set CoutputList lput price CoutputList ]
```

```
  if putmarket [ calculate-sigma
     set PgsigList lput guessed-sigma PgsigList
     set PttmList lput ttm PttmList
     set PspotList lput SexePrice PspotList

     ifelse ( PexePrice + 50 ) < (strike - SexePriceEND)
            [ set buyer True set seller False]
            [ set seller True set buyer False]

     if buyer [ set price PexePrice + (random-normal 2 2)
                set color green ]
     if seller [ set price PexePrice + (random-normal 2 2)
                 set color red ]

     set PoutputList lput price PoutputList ]
]
```

The *price* set by each investor updates the lists of call prices, *ClogB* and *ClogS*, and those of put prices, *PlogB* and *PlogS*. The processes of updating and sorting of books and the procedure to determine the execution price and the agents involved in the options exchange, are the same to those previously explained for stock market.

```
ask tradingAgents [

  if callmarket [ set CexePrice-1 CexePrice

  let tmp[]
  set tmp lput price tmp
  set tmp lput who tmp


  if buyer [ set ClogB lput tmp ClogB ]

  set ClogB reverse sort-by [item 0 ?1 < item 0 ?2] ClogB
  if (not empty? ClogB and not empty? ClogS)
  and item 0 (item 0 ClogB) >= item 0 (item 0 ClogS)
  [ set CexePrice item 0 (item 0 ClogS)
    let agB item 1 (item 0 ClogB)
    let agS item 1 (item 0 ClogS)

    ask turtle agB [ set calls calls + 1
                     set money money - CexePrice
                     set CtotalBought CtotalBought + 1 ]
    ask turtle agS [ set calls calls - 1
```

```
                    set money money + CexePrice
                    set CtotalSold CtotalSold + 1 ]
    set ClogB but-first ClogB
    set ClogS but-first ClogS
  ]


  if seller [ set ClogS lput tmp ClogS ]

  set ClogS sort-by [item 0 ?1 < item 0 ?2] ClogS
  if (not empty? ClogB and not empty? ClogS)
  and item 0 (item 0 ClogB) >= item 0 (item 0 ClogS)
  [ set CexePrice item 0 (item 0 ClogB)
    let agB item 1 (item 0 ClogB)
    let agS item 1 (item 0 ClogS)

    ask turtle agB [ set calls calls + 1
                     set money money - CexePrice
                     set CtotalBought CtotalBought + 1 ]
    ask turtle agS [ set calls calls - 1
                     set money money + CexePrice
                     set CtotalSold CtotalSold + 1 ]
    set ClogB but-first ClogB
    set ClogS but-first ClogS
 ]
set exePrice CexePrice
]

  if putmarket [ set PexePrice-1 PexePrice

  let tmp[]
  set tmp lput price tmp
  set tmp lput who tmp


  if buyer [ set PlogB lput tmp PlogB ]

  set PlogB reverse sort-by [item 0 ?1 < item 0 ?2] PlogB
  if (not empty? PlogB and not empty? PlogS)
  and item 0 (item 0 PlogB) >= item 0 (item 0 PlogS)
  [ set PexePrice item 0 (item 0 PlogS)
    let agB item 1 (item 0 PlogB)
    let agS item 1 (item 0 PlogS)
```

```
    ask turtle agB [ set puts puts + 1
                     set money money - PexePrice
                     set PtotalBought PtotalBought + 1 ]
    ask turtle agS [ set puts puts - 1
                     set money money + PexePrice
                     set PtotalSold PtotalSold + 1 ]
    set PlogB but-first PlogB
    set PlogS but-first PlogS
   ]

   if seller [ set PlogS lput tmp PlogS ]

   set PlogS sort-by [item 0 ?1 < item 0 ?2] PlogS
   if (not empty? PlogB and not empty? PlogS)
   and item 0 (item 0 PlogB) >= item 0 (item 0 PlogS)
   [ set PexePrice item 0 (item 0 PlogB)
   let agB item 1 (item 0 PlogB)
   let agS item 1 (item 0 PlogS)

   ask turtle agB [ set puts puts + 1
                    set money money - PexePrice
                    set PtotalBought PtotalBought + 1 ]
   ask turtle agS [ set puts puts - 1
                    set money money + PexePrice
                    set PtotalSold PtotalSold + 1]
                    set PlogB but-first PlogB
                    set PlogS but-first PlogS
   ]
   set exePrice PexePrice
  ]
 ]
```

Notice that, at the end of each trading day, namely at the end of each *dt*, the four books are reset so as the "features" of agents. This means that at the beginning of each day, every agent has to choose whether to deal in put or call options and whether to agree or disagree with the market trend.

About the procedure *calculate-sigma*, in previous model (see Section 6.1) *PJAgents* calculated their own sigma based on patience, judgement and *retList*, which is a list composed of ratios between *dS* and spot prices evaluated using Gaussian Brownian motion formula. In current model the spot is no more update by GBM, then the elements of *retList*, named *retVal*, cannot be calculated as before. For this reason, the program

provides a new algorithm [2] to update *retList* by the same quantity as before.

```
  if length SpriceList >= 2 [
  let a (length SpriceList) - 1
  let b (length SpriceList) - 2

  let now item a SpriceList
  let bef item b SpriceList

  set retVal (now - bef) / now
  set retList lput retVal retList
]
```

retVal is now evaluated as the ratio between the difference of actual stock price (*now*) and previous stock price (*bef*), and the actual stock price. In this way, criteria for the evaluation of sigma is the same as before.

After markets of stock and option have been run for once, each *tradingAgents* creates its own dataframe, which are used for training the individual neural networks for call and put price prediction. The learning process remains the same as the previous model.

What changes are the inputs and outputs that neural networks receive for training, in particular spot prices and option price. Indeed, new neural networks do not learn via Black–Scholes prices: spot prices updated via GMB are replaced by execution prices resulting from simulation of a stock market, while BS option prices are replaced by prices estimated by each trader according to the execution prices of calls or puts. On the contrary of preceding outputs, now predictions on options prices of neural networks are correct.

As previously mentioned, now that the agents have the possibility to choose if operate with call or put, they must be able to evaluate both call and put prices. For this reason, traders need two neural networks each one. This means that they have to generate two dataframe, one made up by lists referring to call prices and one made up by the other lists referring to put prices, and then use them to train the neural networks. The modified NetLogo code for the new procedures *make_ datatrain* and *train* is shown below:

```
  to make_datatrain

    rserve:put "CspotList" CspotList
    rserve:put "CgsigList" CgsigList
    rserve:put "CttmList" CttmList
    rserve:put "CoutputList" CoutputList
    rserve:eval "CS<-as.numeric(CspotList)"
    rserve:eval "Csigma<-as.numeric(CgsigList)"
```

[2] The algorithm to update *retList* is placed within the procedure *stock-market*.

```
rserve:eval "Cttm<-as.numeric(CttmList)"
rserve:eval "COptionPrice<-as.numeric(CoutputList)"
rserve:eval "f<-length(CS)"
rserve:eval "a<-sample(2:(f-10), 1)"
rserve:eval "CS<-CS[a:f]"
rserve:eval "Csigma<-Csigma[a:f]"
rserve:eval "Cttm<-Cttm[a:f]"
rserve:eval "COptionPrice<-COptionPrice[a:f]"
rserve:eval "Cdatatrain<-cbind(CS, Csigma, Cttm, COptionPrice)"
rserve:eval "colnames(Cdatatrain)<-c('CS','Csigma',
                                    'Cttm','COptionPrice')"
rserve:eval "Cmaxsd<-apply(Cdatatrain, 2, max)"
rserve:eval "Cminsd<-apply(Cdatatrain, 2, min)"
rserve:eval "Cscaledtrain<-as.data.frame (scale(Cdatatrain,
                          center = Cminsd , scale = Cmaxsd - Cminsd))"
set Cmin1 rserve:get "Cminsd[1]"
set Cmax1 rserve:get "Cmaxsd[1]"
set Cmin2 rserve:get "Cminsd[2]"
set Cmax2 rserve:get "Cmaxsd[2]"
set Cmin3 rserve:get "Cminsd[3]"
set Cmax3 rserve:get "Cmaxsd[3]"
set Cmin4 rserve:get "Cminsd[4]"
set Cmax4 rserve:get "Cmaxsd[4]"


rserve:put "PspotList" PspotList
rserve:put "PgsigList" PgsigList
rserve:put "PttmList" PttmList
rserve:put "PoutputList" PoutputList
rserve:eval "PS<-as.numeric(PspotList)"
rserve:eval "Psigma<-as.numeric(PgsigList)"
rserve:eval "Pttm<-as.numeric(PttmList)"
rserve:eval "POptionPrice<-as.numeric(PoutputList)"
rserve:eval "e<-length(PS)"
rserve:eval "b<-sample(2:(e-10), 1)"
rserve:eval "PS<-PS[b:e]"
rserve:eval "Psigma<-Psigma[b:e]"
rserve:eval "Pttm<-Pttm[b:e]"
rserve:eval "POptionPrice<-POptionPrice[b:e]"
rserve:eval "Pdatatrain<-cbind(PS, Psigma, Pttm, POptionPrice)"
rserve:eval "colnames(Pdatatrain)<-c('PS','Psigma',
                                    'Pttm','POptionPrice')"
rserve:eval "Pmaxsd<-apply(Pdatatrain, 2, max)"
```

```
  rserve:eval "Pminsd<-apply(Pdatatrain, 2, min)"
  rserve:eval "Pscaledtrain<-as.data.frame (scale(Pdatatrain,
                            center = Pminsd , scale = Pmaxsd - Pminsd))"
  set Pmin1 rserve:get "Pminsd[1]"
  set Pmax1 rserve:get "Pmaxsd[1]"
  set Pmin2 rserve:get "Pminsd[2]"
  set Pmax2 rserve:get "Pmaxsd[2]"
  set Pmin3 rserve:get "Pminsd[3]"
  set Pmax3 rserve:get "Pmaxsd[3]"
  set Pmin4 rserve:get "Pminsd[4]"
  set Pmax4 rserve:get "Pmaxsd[4]"
end

to train
  rserve:eval "Cnn<-neuralnet ( COptionPrice~CS+Csigma+Cttm,
                               Cscaledtrain, hidden = 3,
                               threshold = 0.03, rep=10)"
  rserve:eval "Crn<-which.min(Cnn$result.matrix[1,])"
  rserve:eval "plot(Cnn, rep = Crn)"
  rserve:put  "CmyNum" who
  rserve:eval "Cname<-paste('Cnn', CmyNum, sep='')"
  rserve:eval "assign(Cname, Cnn)"
  set Crn rserve:get "Crn"

  rserve:eval "Pnn<-neuralnet ( POptionPrice~PS+Psigma+Pttm,
                               Pscaledtrain, hidden = 3,
                               threshold = 0.03, rep=10)"
  rserve:eval "Prn<-which.min(Pnn$result.matrix[1,])"
  rserve:eval "plot(Pnn, rep = Prn)"
  rserve:put  "PmyNum" who
  rserve:eval "Pname<-paste('Pnn', PmyNum, sep='')"
  rserve:eval "assign(Pname, Pnn)"
  set Prn rserve:get "Prn"
end
```

Once that neural networks have been trained, simulations of the markets proceed following the order already established. Each time step, stock market runs as explained before, then when it ends, simulation of options market begins (through the procedure *options-NNmarket*). Also options market works the same way of the first market execution, with the difference that now the options prices to insert in books are evaluated using neural networks via the to–report procedure *Cpredict* for call prices and *Ppredict* for put prices. Here below the code of procedures *Cpredict*, *Ppredict* and *options-NNmarket*.

```
to-report Cpredict [-SexePrice -sigma -ttm ]
```

```
  let sspot (( -SexePrice - Cmin1 ) / (Cmax1 - Cmin1 ))
  let ssigma (( -sigma - Cmin2 ) / ( Cmax2 - Cmin2 ))
  let sttm (( -ttm - Cmin3 ) / ( Cmax3 - Cmin3 ))
  rserve:put "Crn" Crn
  rserve:put "CmyNum" who
  rserve:eval "CmyNn<-paste('Cnn', CmyNum, sep='')"
  rserve:eval "Cnn <- get(CmyNn)"
  let test rserve:get "CmyNn"
  show test
  rserve:eval "Ctt<-names(Cnn)[13]=='result.matrix'"

  (rserve:putdataframe "Cdata" "CS" sspot  "Csigma" ssigma "Cttm" sttm)
  rserve:eval "Cnprice<-compute(Cnn, Cdata, rep = Crn)$net.result[1,1]"
  set Cnprice rserve:get "Cnprice"
  let -nnprice  Cnprice * ( Cmax4 - Cmin4 ) + Cmin4
  report -nnprice
end

to-report Ppredict [-SexePrice -sigma -ttm ]
  let sspot (( -SexePrice - Pmin1 ) / (Pmax1 - Pmin1 ))
  let ssigma (( -sigma - Pmin2 ) / ( Pmax2 - Pmin2 ))
  let sttm (( -ttm - Pmin3 ) / ( Pmax3 - Pmin3 ))
  rserve:put "Prn" Prn
  rserve:put "PmyNum" who
  rserve:eval "PmyNn<-paste('Pnn', PmyNum, sep='')"
  rserve:eval "Pnn <- get(PmyNn)"
  let test rserve:get "PmyNn"
  show test
  rserve:eval "Ptt<-names(Pnn)[13]=='result.matrix'"

  (rserve:putdataframe "Pdata" "PS" sspot  "Psigma" ssigma "Pttm" sttm)
  rserve:eval "Pnprice<-compute(Pnn, Pdata, rep = Prn)$net.result[1,1]"
  set Pnprice rserve:get "Pnprice"
  let -nnprice  Pnprice * ( Pmax4 - Pmin4 ) + Pmin4
  report -nnprice
end

to options-NNmarket

 ask tradingAgents [ let f random 10
                     ifelse f > 3 [set agree True set notagree False]
                                  [set notagree True set agree False]
                   ]
```

```
ask tradingAgents [
     if StrendUp [ if agree [ if agree [ set SexePriceEND SexePrice
                                              * (1 + (ttm / dt) * ( 0.002))]
                      if notagree [set SexePriceEND SexePrice -
                                     (ttm / dt) * (random-normal 0 50)]]

     if StrendDown [ if agree [ if agree [ set SexePriceEND SexePrice
                                               * (1 - (ttm / dt) * ( 0.002))]
                        if notagree [ set SexePriceEND SexePrice +
                                     (ttm / dt) * ( random-normal 0 50)]]
  ]

ask tradingAgents [ let s random 10
                    ifelse s > 5
                           [set callmarket True set putmarket False]
                           [set putmarket True set callmarket False]
  ]

ask tradingAgents [ calculate-sigma ]

ask tradingAgents [
     if callmarket [ set price Cpredict SexePrice guessed-sigma ttm

      ifelse ( CexePrice + 50 ) < ( SexePriceEND - strike )
             [ set buyer True set seller False]
             [ set seller True set buyer False]
  ]

     if putmarket [ set price Ppredict SexePrice guessed-sigma ttm

      ifelse ( PexePrice + 50 ) < (strike - SexePriceEND)
             [ set buyer True set seller False]
             [ set seller True set buyer False]
  ] ]

ask tradingAgents [
  if callmarket [ set CexePrice-1 CexePrice

  let tmp[]
  set tmp lput price tmp
  set tmp lput who tmp
```

```
  if buyer [ set ClogB lput tmp ClogB ]

  set ClogB reverse sort-by [item 0 ?1 < item 0 ?2] ClogB
  if (not empty? ClogB and not empty? ClogS)
  and item 0 (item 0 ClogB) >= item 0 (item 0 ClogS)
  [ set CexePrice item 0 (item 0 ClogS)
    let agB item 1 (item 0 ClogB)
    let agS item 1 (item 0 ClogS)

    ask turtle agB [ set calls calls + 1
                     set money money - CexePrice
                     set CtotalBought CtotalBought + 1]
    ask turtle agS [ set calls calls - 1
                     set money money + CexePrice
                     set CtotalSold CtotalSold + 1 ]
    set ClogB but-first ClogB
    set ClogS but-first ClogS
   ]

  if seller [set ClogS lput tmp ClogS]

  set ClogS sort-by [item 0 ?1 < item 0 ?2] ClogS
  if (not empty? ClogB and not empty? ClogS)
  and item 0 (item 0 ClogB) >= item 0 (item 0 ClogS)
  [ set CexePrice item 0 (item 0 ClogB)
    let agB item 1 (item 0 ClogB)
    let agS item 1 (item 0 ClogS)

    ask turtle agB [ set calls calls + 1
                     set money money - CexePrice
                     set CtotalBought CtotalBought + 1 ]
   ask turtle agS [ set calls calls - 1
                    set money money + CexePrice
                    set CtotalSold CtotalSold + 1 ]
   set ClogB but-first ClogB
   set ClogS but-first ClogS
  ]

 set exePrice CexePrice
]

  if putmarket [ set PexePrice-1 PexePrice
```

```
 let tmp[]
 set tmp lput price tmp
 set tmp lput who tmp

 if buyer [set PlogB lput tmp PlogB]

 set PlogB reverse sort-by [item 0 ?1 < item 0 ?2] PlogB
 if (not empty? PlogB and not empty? PlogS)
 and item 0 (item 0 PlogB) >= item 0 (item 0 PlogS)
 [ set PexePrice item 0 (item 0 PlogS)
   let agB item 1 (item 0 PlogB)
   let agS item 1 (item 0 PlogS)

   ask turtle agB [ set puts puts + 1
                    set money money - PexePrice
                    set PtotalBought PtotalBought + 1 ]
   ask turtle agS [ set puts puts - 1
                    set money money + PexePrice
                    set PtotalSold PtotalSold + 1 ]
   set PlogB but-first PlogB
   set PlogS but-first PlogS
 ]

 if seller [ set PlogS lput tmp PlogS ]

 set PlogS sort-by [item 0 ?1 < item 0 ?2] PlogS
 if (not empty? PlogB and not empty? PlogS)
 and item 0 (item 0 PlogB) >= item 0 (item 0 PlogS)
 [ set PexePrice item 0 (item 0 PlogB)
   let agB item 1 (item 0 PlogB)
   let agS item 1 (item 0 PlogS)

   ask turtle agB [ set puts puts + 1
                    set money money - PexePrice
                    set PtotalBought PtotalBought + 1 ]
   ask turtle agS [ set puts puts - 1
                    set money money + PexePrice
                    set PtotalSold PtotalSold + 1 ]
   set PlogB but-first PlogB
   set PlogS but-first PlogS
 ]
 set exePrice PexePrice
]
```

```
 ]
end
```

Simulation ends when the markets have been run a number of times equal to the number of *finalMarket* set by user, unless the stock price goes to zero. In this case the simulation stops automatically.

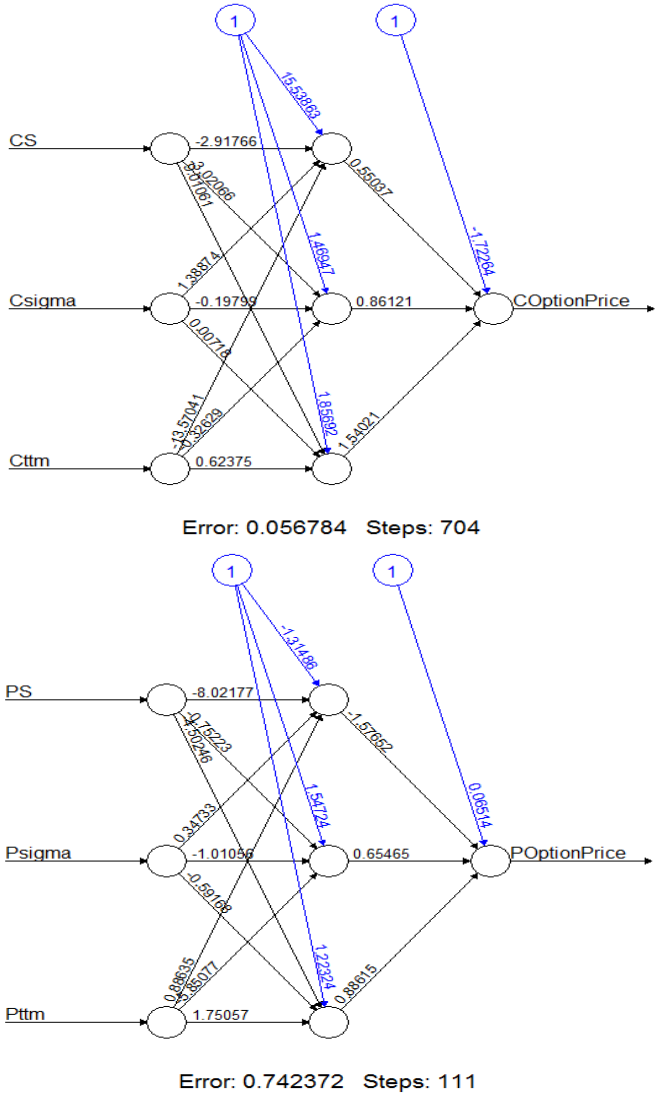


Figure 6.7: Plots of two neural networks: one for call price evaluation (high) and one for put price evaluation (low).

Chapter 7

Data and Results

The evaluation on performance is a key aspect in the design process of a neural network model. The judgement about the accuracy of forecasts must be made by comparisons among the data predicted by the model and related data observed. It is deemed that a sufficiently reliable estimate of accuracy and precision of predictions of a model is given by measurements made on the differences between the expected values and those observed, said "difference measures".

The root mean square error (RMSE) is a good method to evaluate the accuracy of a neural network's predictions. To obtain an index of accuracy of the neural networks used in this project, several dataframe of different agents have been consider, which refer to the entire execution period of the market; these dataframe have been randomly split into train set and test set. The train data have been used to training the networks with parameters mentioned in Section 6.3.1. Once training process finished, neural network predicted the values for the test set and calculated the root mean square error among the option prices collected in test set and the price predicted for this set. The value obtained is $RMSE = 0.1224$, which means that the neural network so designed and trained is able to make a good prediction of options prices.

An additional method for the evaluation of accuracy of the neural network can be achieved through the indexes of Willmott [71]. The Index of Agreement developed by Willmott is a standardized measure of the degree of model prediction error and varies between 0 and 1. A value of 1 indicates a perfect match and 0 indicates no agreement at all. Software R is useful to compute this index, as it offers the package *hydroGOF* which contains the function d set for the valuation of the index of agreement d:

$$d = 1 - \left[\frac{\sum_{i=1}^{N} (obs - pred)^2}{\sum_{i=1}^{N} (|obs - mean(pred)| + |pred - mean(obs)|)^2} \right] \tag{7.1}$$

Where *obs* refers to observed data and *pred* to predicted data. As further proof of the fact that neural networks used in these models have outstanding predictive ability, the Willmot's index evaluated with R's function d is equal to 0.998, which is so close to one, thus to the perfect match.

It should be noted that in the final model agents are designed with different experience, that is different ability to evaluate options. This feature is realized by implementing a neural network for each agent, which is trained with data collected over a more or less long time period by single agent. This means that not all neural networks are able to reach the previously calculated degree of accuracy of prediction, just for the fact that not all the networks of all agents have "complete" experience. However, the aim was exactly to recreate the heterogeneity of real market by modelling agents more or less able to evaluate these financial instrument.

To understand the success of the simulation, an analysis on the trend of options prices is required. As the final aim of this project is the simulation of an artificial options market whose agents evaluate calls and puts based on "real" prices of the underlying, the analysis will focus only on the third model.

This last model tries to simulate an on options market in which every day some agents trade call and put options. Each trading day agents randomly decide if will operate with calls or puts; independently of this choice, then they establish their personal opinion on the future price of the underlying, based on they agree or not with the trend of the stock. Traders operating in call market become possible buyer if call price is much smaller than the difference between spot price and strike price (quantity equivalent to the payoff if the option is in–the–money at maturity), vice versa they become possible seller. Instead, traders operating in put market become possible buyer if put price is much smaller than the difference between strike price and spot price (quantity equivalent to the payoff if the option is in–the–money at maturity), conversely they become possible seller. These conditions correspond to assume that investors who buy call options have long position, namely they estimate the price of the underlying will be greater than the sum of strike price and call price at expiration date; vice versa investor who buy put have short position, that is they estimate the price of the underlying will be smaller than the difference among strike price and put price.

This short reference of options theory in order to clarify the dependences of call and put prices on variables implied in their evaluation, in particular spot price. Table 7.1 shows how all these variables affect trend of call and put price. Arrow up represents direct proportional dependence and arrow down inverse proportional dependence.

	spot	**strike**	**sigma**	**ttm**	**riskfree rate**
call ↑	↑	↓	↑	↑	↑
put ↑	↓	↑	↑	↑	↓

Table 7.1: Call and put dependences on spot price, strike price, sigma, time to maturity and riskfree interest rate.

Observing the Table 7.1, one should expect that upon the occurrence of an increase of

the underlying price the call price increases, while put price decreases. On the contrary, in the case where the price of the underlying decreases, the price of the call should go down while the price of the put should go up. Opposite situation with regard to the strike price, which is positively correlated with put option and negatively correlated with call option. Both types of options are inversely proportional to time to maturity: this means that options are worth much less when they approach expiration date. This result comes from the fact that investors are no longer tempted to trade options when they are reaching maturity, consequently the price tends to go down due to low demand.

The images in Figure 7.1 show the results of three different simulation at the end of each execution of stock and options market (coincident with the expiration date of options contracts). In particular, each image represents four plots, from the top to the bottom are: the trend of spot and strike prices, the trend of call and put prices, the plot of average *guessed-sigma* and the plot of total amount of options bought and sold.

At the top of each of three groups, it is shown the four graphs at the end of first stock and options markets execution ($nthTime$=1). Comparing the first two plots, it is observed that the trends of call and put prices with respect to that of spot price are consistent with the statements explained before: when price of the underlying goes up, call price slightly increases and put price slightly decreases, vice versa if it goes down. Quantities of calls and puts bought and sold, depicted in the fourth graph, are congruent to the performance of the underlying market.

Further to this short evaluation, it may conclude that the first execution of stock market and options market, in which agents evaluate call and put according to the last execution price (no neural networks are implied for the moment), offers result adequately close to reality.

When first stock and options markets end to run, agents create their own dataframe, with whom neural networks will be trained. It should be remembered that, from second "time" of the market, options prices are no longer estimated by agents by adding a normal amount to the last execution prices, instead calls and puts are evaluated via neural network predictions.

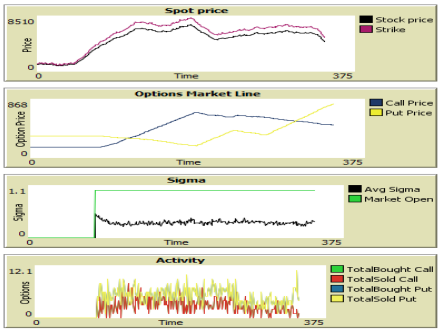
Figures placed below the first image, depict the interfaces at the end of successive markets executions ($nthTime$>1). Notice that, procedure *stock-market*, which "run" the simulation of stock market, remains unchanged, thus graphs of stock price still continues to show good trend with increments and decrements due to the instability of the market. Now attention must focus mainly on the graphs of options prices, estimated through the use of neural networks. From plots imported in Figure 7.1, one may observed that the trends of call and put options is globally congruent with respect to that of spot prices. Except some sporadic occasions, call price slightly goes up when spot increases and it goes down when spot decreases, conversely for put price.

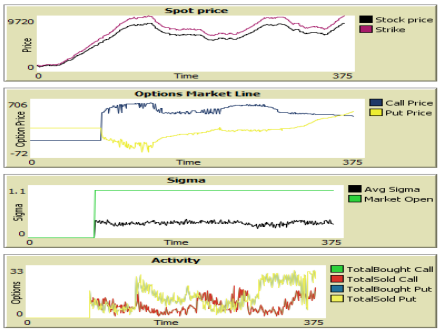
Unlike the plot of the first run, the graphs in question show plots with presence of quite high noise. This situation is a consequence of the fact that, as explained before, each agent (so each neural network) grows different experience and has different ability to evaluate price of options. As a result, an investor inexperienced and not very skilled appraises option inadequately and this will involve diagrams not "clean" and linear as

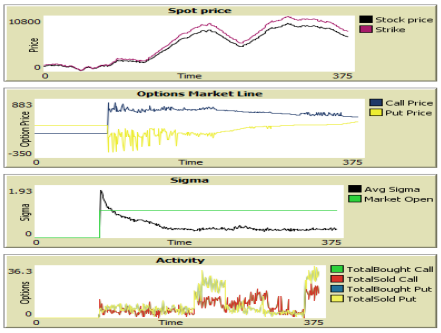
the previous one. It should be noted that this is not an error. Neural networks optimally trained succeed to predict prices almost perfectly, as the low RMSE demonstrate. However, the intention of this project is to simulate as much as possible the reality in an artificial markets. That was the intention behind the insertion in the model of random agents and also agents with different capability to evaluate financial instruments.

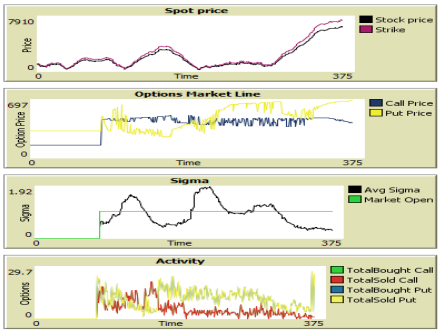
In conclusion, the simulation and prediction model experienced has produced good results both regards the simulation of artificial stock and options markets and regards the prediction of prices via neural networks. The project has been successful in realising an alternative way to Black–Scholes method for evaluating the evolution of call and put prices through an Agent–based simulation of the underlying and options markets and neural networks used for price prediction.

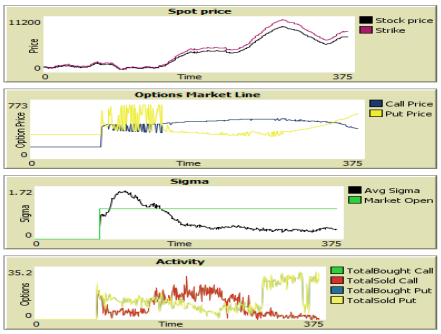
Furthermore, it has been possible verify that neural networks, if adequately designed and trained, may be optimal tools for options price predictions, as Agent–based model technique is a satisfactory approach for simulating financial markets.

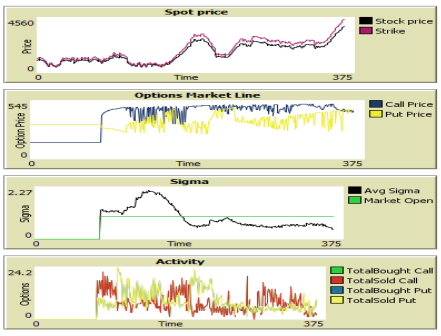

Spot price
8510
Price
0
Time
0
375
Stock price
Strike
Options Market Line
868
Option Price
0
Time
0
375
Call Price
Put Price
Sigma
1.1
Sigma
0
Time
0
375
Avg Sigma
Market Open
Activity
12.1
Options
0
Time
0
375
TotalBought Call
TotalSold Call
TotalBought Put
TotalSold Put


Spot price
9720
Price
0
Time
0
375
Stock price
Strike
Options Market Line
706
Option Price
-72
Time
0
375
Call Price
Put Price
Sigma
1.1
Sigma
0
Time
0
375
Avg Sigma
Market Open
Activity
33
Options
0
Time
0
375
TotalBought Call
TotalSold Call
TotalBought Put
TotalSold Put

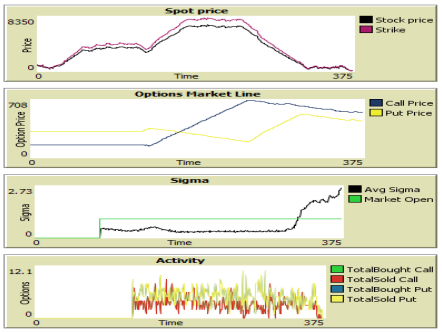

Spot price
10800
Price
0
Time
0
375
Stock price
Strike
Options Market Line
883
Option Price
-350
Time
0
375
Call Price
Put Price
Sigma
1.93
Sigma
0
Time
0
375
Avg Sigma
Market Open
Activity
36.3
Options
0
Time
0
375
TotalBought Call
TotalSold Call
TotalBought Put
TotalSold Put

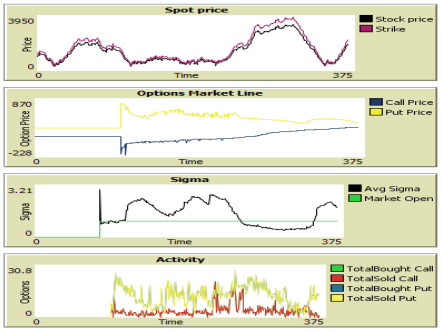

Spot price
7910
Price
0
Time
0
375
Stock price
Strike
Options Market Line
697
Option Price
0
Time
0
375
Call Price
Put Price
Sigma
1.92
Sigma
0
Time
0
375
Avg Sigma
Market Open
Activity
29.7
Options
0
Time
0
375
TotalBought Call
TotalSold Call
TotalBought Put
TotalSold Put

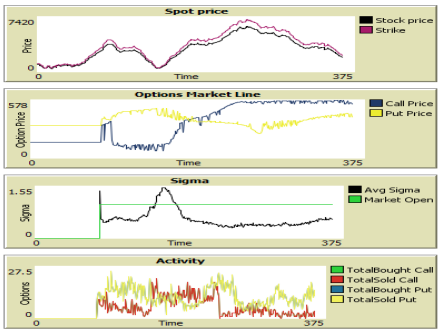

Spot price
11200
Price
0
Time
0
375
Stock price
Strike
Options Market Line
773
Option Price
0
Time
0
375
Call Price
Put Price
Sigma
1.72
Sigma
0
Time
0
375
Avg Sigma
Market Open
Activity
35.2
Options
0
Time
0
375
TotalBought Call
TotalSold Call
TotalBought Put
TotalSold Put

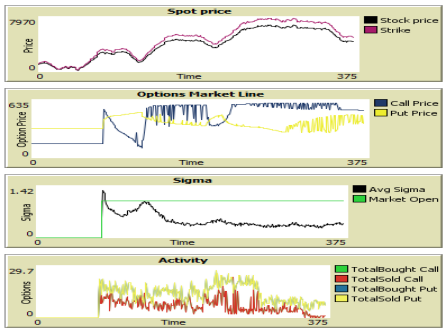

Spot price
4560
Price
0
Time
0
375
Stock price
Strike
Options Market Line
545
Option Price
0
Time
0
375
Call Price
Put Price
Sigma
2.27
Sigma
0
Time
0
375
Avg Sigma
Market Open
Activity
24.2
Options
0
Time
0
375
TotalBought Call
TotalSold Call
TotalBought Put
TotalSold Put

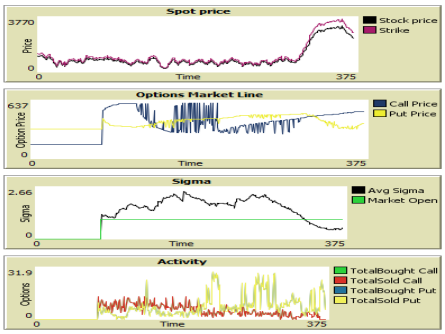

Spot price
6240
Price
0
Time
0
375
Stock price
Strike
Options Market Line
685
Option Price
0
Time
0
375
Call Price
Put Price
Sigma
1.1
Sigma
0
Time
0
375
Avg Sigma
Market Open
Activity
29.7
Options
0
Time
0
375
TotalBought Call
TotalSold Call
TotalBought Put
TotalSold Put

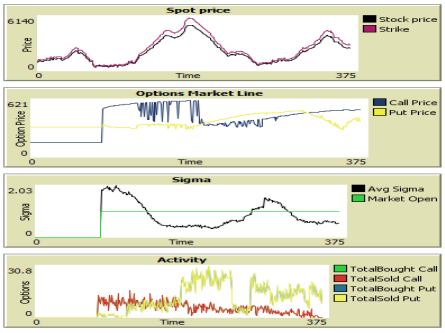

Spot price
8350
Price
0
Time
0
375
Stock price
Strike
Options Market Line
708
Option Price
0
Time
0
375
Call Price
Put Price
Sigma
2.73
Sigma
0
Time
0
375
Avg Sigma
Market Open
Activity
12.1
Options
0
Time
0
375
TotalBought Call
TotalSold Call
TotalBought Put
TotalSold Put

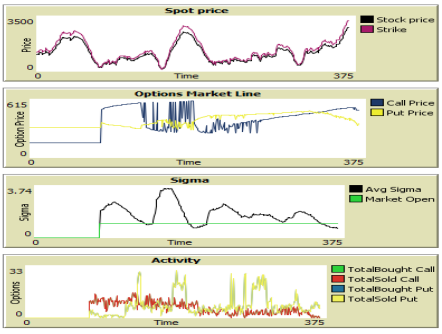

Spot price
3950
Price
0
Time
0
375
Stock price
Strike
Options Market Line
870
Option Price
-228
Time
0
375
Call Price
Put Price
Sigma
3.21
Sigma
0
Time
0
375
Avg Sigma
Market Open
Activity
30.8
Options
0
Time
0
375
TotalBought Call
TotalSold Call
TotalBought Put
TotalSold Put


Spot price
7420
Price
0
Time
0
375
Stock price
Strike
Options Market Line
578
Option Price
0
Time
0
375
Call Price
Put Price
Sigma
1.55
Sigma
0
Time
0
375
Avg Sigma
Market Open
Activity
27.5
Options
0
Time
0
375
TotalBought Call
TotalSold Call
TotalBought Put
TotalSold Put


Spot price
7970
Price
0
Time
0
375
Stock price
Strike
Options Market Line
635
Option Price
0
Time
0
375
Call Price
Put Price
Sigma
1.42
Sigma
0
Time
0
375
Avg Sigma
Market Open
Activity
29.7
Options
0
Time
0
375
TotalBought Call
TotalSold Call
TotalBought Put
TotalSold Put


Spot price
3770
Price
0
Time
0
375
Stock price
Strike
Options Market Line
637
Option Price
0
Time
0
375
Call Price
Put Price
Sigma
2.66
Sigma
0
Time
0
375
Avg Sigma
Market Open
Activity
31.9
Options
0
Time
0
375
TotalBought Call
TotalSold Call
TotalBought Put
TotalSold Put


Spot price
6140
Price
0
Time
0
375
Stock price
Strike
Options Market Line
621
Option Price
0
Time
0
375
Call Price
Put Price
Sigma
2.03
Sigma
0
Time
0
375
Avg Sigma
Market Open
Activity
30.8
Options
0
Time
0
375
TotalBought Call
TotalSold Call
TotalBought Put
TotalSold Put


Spot price
3500
Price
0
Time
0
375
Stock price
Strike
Options Market Line
615
Option Price
0
Time
0
375
Call Price
Put Price
Sigma
3.74
Sigma
0
Time
0
375
Avg Sigma
Market Open
Activity
33
Options
0
Time
0
375
TotalBought Call
TotalSold Call
TotalBought Put
TotalSold Put

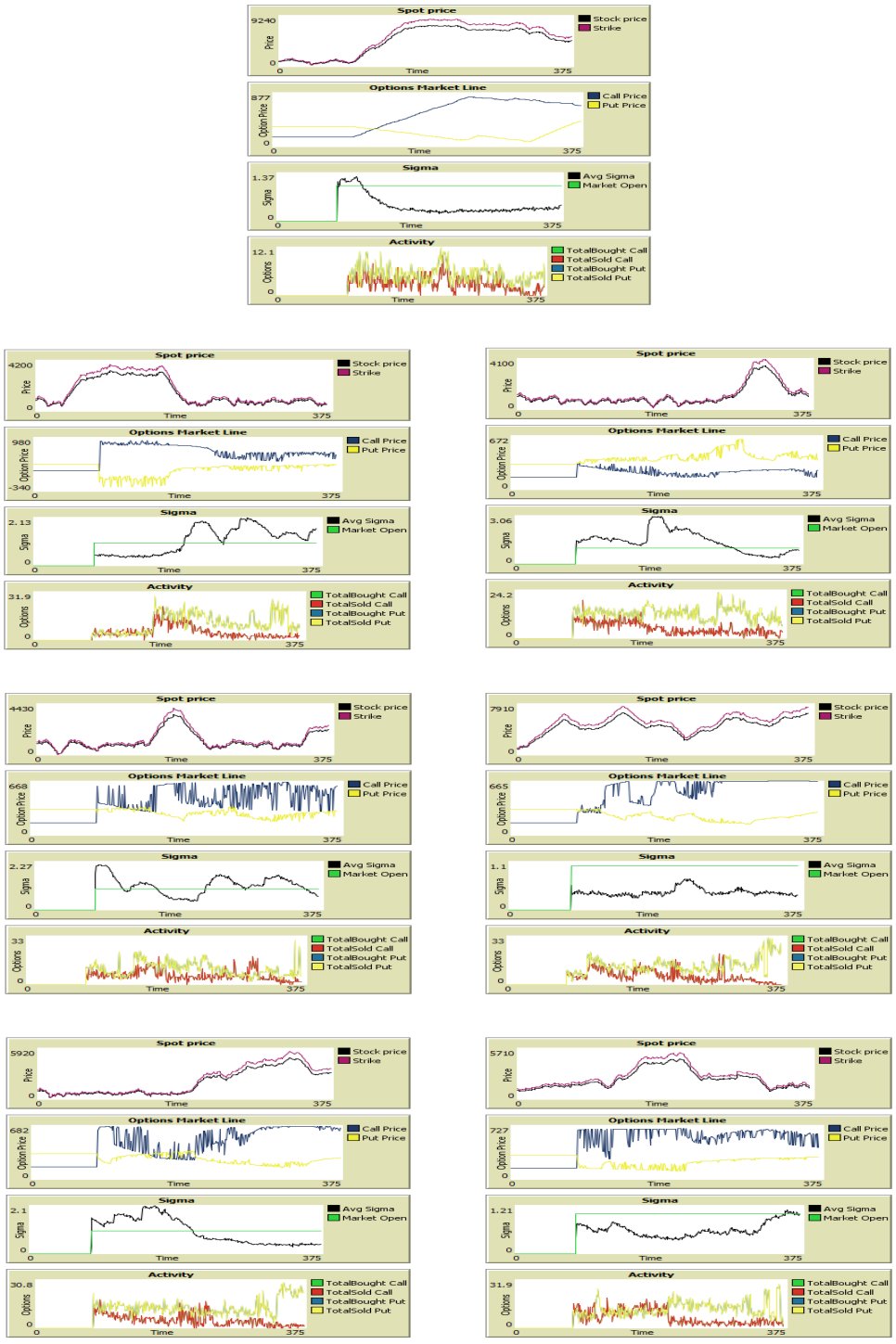


Figure 7.1: Three examples of different simulation. First image represents the results of the first execution of stock and options markets ($nthTime$=1), while the others show results of following executions ($nthTime$>1)

Chapter 8

Conclusions

The first part of this thesis covers introductions on theories, methods and techniques used to realise the model. Chapter 2 offers a comprehensive overview of options theory, from definition of these financial instruments to the demonstrations of most common pricing method. Attention is focused on Black–Scholes equation, which estimates the price of the European call and put option over time under several and rigorous conditions. Despite these assumptions are non–enforceable in real world, BS equation is the only formula universally accepted by option traders to determine the theoretical price of an option. The idea of this project sprang from limits imposed by BS theory and sets out to find an alternative method that can estimate the prices of European options under conditions closer to those of a real market.

Chapter 3 introduces the Agent–based model technique, which is the approach chosen for simulating the options market model. This choice was based on several features of ABMs, such as the ability to simulate the simultaneous operations and interactions of multiple agents and in particular to predict and recreate the appearances of complex phenomena. In addition, Agent–based modelling can be employed in several field, biology, network theory, social science and extensively in economics, as shown in Section 3.3.

Chapter 4 illustrates the theory and applications of artificial neural networks. In this project multilayer perceptrons have been used in order to mimic Black–Scholes formula (Section 6.3) and then to predict options price learning from a simulated stock market (Section 6.4).

Chapter 5 is concerned to explain functioning and features of language programming R. Particular attention has been paid to the *neuralnet* package, whose functions have been used to train the neural networks (*neuralnet*), plot them (*plot*) and predict (*compute*) options prices.

Finally, Chapter 6 includes a description of the Elliott's *OptionsMarket* model, used as starting point, and principally a detailed explanation of three models designed in this project, including some extract of NetLogo pseudo–code. Table 6.1 is shown below again in order to remember the steps realised to obtain the final model.

Despite only the third model represents the final goal of this work, the implementation of the first two models has been preliminary and necessary for the achievement of it.

OptionsMarket model		
pricing via BS	**mimicking BS via NNs**	**learning from stock market**
Market with high volatility	Market with high volatility	Market with high volatility
Spot updated via GBM	Spot updated via GBM	CDA model for stock prices
Option pricing via BS	Option pricing via NNs	Option pricing via NNs
Call or put market	Call or put market	Call and put markets

Indeed, the first model, which simulate a options market under high variability, can be considered the starting point of this work. It is simply the Elliott's *OptionsMarket* model, in which both intelligent and random agents trade options according to Black–Scholes formula. As mentioned before, the introduction of naive traders involves the presence of randomness in the market and consequently a world in which assumptions of BS theory are no more completely satisfied.

Inspiring by several previous works regarding applications of neural network in financial field, it has been decided to use multilayer perceptrons for option price prediction as alternative to BS formula. After a scrupulous evaluation, based on numerous parameters of efficiency and practicality, it has been chosen to implement multilayer perceptrons in R language, using the package *neuralnet*. Firstly, the program used only a "global" neural network, trained with data of different traders. Since this strategy is not suitable for simulating real market, next step provided to create a number of neural networks equal to the number of agents in the market. The purpose to reproduce heterogeneity of real market, it has been obtained by training neural networks for different time period and with variables characteristic of each investor. In this way each agent is able to evaluate options according to the different experience "acquired".

The data used for training process, are the same required in BS formula, that are time to maturity, volatility and spot price updated with GBM. As target output it has been used the option price evaluated by agents via BS formula. This means that neural networks learn to predict options prices as BS prices. The applications of neural networks, thus trained, in substitution to the Black–Scholes formula cannot reduce the error due to the presence of randomness in the market. However, it has been demonstrated that they are an optimal technique to mimic BS formula and therefore to price call and put options. Parameters and training procedure establish in this phase are the same employed in final model.

One of the greater limits of Black–Scholes theory is the assumption that the spot

price is required to be updated through Gaussian Brownian motion. Although it is a valid approximation, the underlying follows a trend driven by agents which, by buying and selling the asset, determine its price. In order to simulate the stock price formation as in the case of real market and to overcome the limit forced by BS method, it has been introduced a continuous double auction model which is one of the most common financial technique able to reproduce modern stock market.

Moreover, unlike what happens in previous simulation, the final model allows the simultaneous but independent execution of both call and puts markets. In addition to get a more realistic simulation, this optimisation permits to observe together the trends of call and put prices and then understand how they move dependently on other variables, as stock price, strike price, time and volatility.

After realizing the third model with all optimisations explained above and performing some simulations, it has been possible to observe that call and put prices moved appropriately with respect to the dependences of spot prices and other variables which affect option value. Slightly different results have been achieved at the end of first execution of stock and options markets and at the end of following executions, when neural networks are employed. This is due to the fact that in the first execution agents use to evaluate options based on the value of the trading day before, while in subsequent executions option prices are provided by neural networks trained with the aim to represent the different experiences and ability of each agent in the market. As explained in Chapter 7, outcomes of all simulations are rather satisfactory and consistent with result of real market.

The model realised in the project reaches all the purposes set, especially the fulfilment of a method to predict the trend of call and put prices, as an alternative of Black–Scholes formula, in an options market under high variability and spot price which does not follow Gaussian Brownian motion.

8.1 Further researches

The main goal of this project was to design a model of options market as realistic as possible. Although this purpose has been partially achieved with the final model, there are still many optimizations that could make this work more complete and realistic. For this reason, a number of future ideas are presented below for the improvement of the model.

All the models displayed in this project simulate a market in which options have all the same maturity and there are not new contracts that begin when market runs. The introduction of different options with different expiration dates may the first big step towards a model closer to reality.

The final model simulates a market of options in which agents evaluate puts and calls prices according to the trend of the underlying. As mentioned above, the stock market is executed before the beginning of options market. This means that two markets do not run simultaneously and the options traders see all together the same stock execution price at the same time. This circumstance is a simplification of the reality, since stock market and options market actually operate in parallel. An important breakthrough

would be to simulate the two markets simultaneously, requiring that options agents enter the market in different moments and observe different execution prices of the underlying.

Another optimization to the model regards the implementation of different technique for stock price formation. Remember that each agent sets its own price (which will insert in the books *SlogB* or *SlogS*) equal to the execution price of the day before plus a random number normally distributed. In real market, traders place their bid or ask prices according to the current execution stock price, not looking at the last execution price. A first improvement could be just the replacement of the method of price formation currently used with one more similar to actual methods.

These are just some of the many possibilities of improvement that could be applied to the model realized in this thesis, in order to obtain the best price forecasting and a more realistic artificial market than already planned.

Appendix A

Geometric Brownian Motion

Geometric Brownian motion is a stochastic process in continuous time, whose logarithm of the random variable follows a Brownian motion, more precisely a Wiener process [1]. The GBM is also said Economic Brownian motion, because a quantity that follows a Geometric Brownian motion can assume only values greater than zero, which reflects the nature of the price of a financial asset. This property makes it suitable to model financial markets, in particular for options pricing.

A stochastic process S which follows a Geometric Brownian Motion satisfy the following stochastic differential equation (SDE):

$$dS_t = \mu\, S_t\, dt + \sigma\, S_t dW_t \tag{A.1}$$

where dW_t is the Wiener process and μ (drift coefficient) and σ (diffusion coefficient) are real constant parameter.

The solution of the SDE is:

$$S_t = S0\, e^{\,(\mu - \frac{1}{2}\,\sigma^2) + \sigma\,(W_t - W_0)} \tag{A.2}$$

Here below the demonstration to solve the SDE in Equation (A.1) .

Proof. Define the function Y:

$$Y = g(S,t) = \ln(S) \tag{A.3}$$

One uses Itô's Lemma to derive the process followed by Y, when S follows the process in Equation (A.1).

[1] Wiener process is a Gaussian stochastic process in continuous time with independently increments, useful to model Brownian motion in applied mathematics, finance and physics fields.

$$dY = \frac{\partial g}{\partial t}\, dt + \frac{\partial g}{\partial S}\, dt + \frac{1}{2}\, \sigma^2\, S^2\, \frac{\partial^2 g}{\partial S^2} dt \tag{A.4}$$

The solutions of partial equations are:

$$\frac{\partial g}{\partial t} = 0 \tag{A.5}$$

$$\frac{\partial g}{\partial S} = \frac{1}{S} \tag{A.6}$$

$$\frac{\partial^2 g}{\partial S^2} = -\frac{1}{S^2} \tag{A.7}$$

Substituting them in Equation (A.4) and making explicit the expression of S, after some adjustments one gets:

$$dY = (\mu - \frac{1}{2}\sigma^2)\, dt + \sigma\, dW \tag{A.8}$$

As μ and σ are constant parameters, Equation (A.8) indicates that Y follows a generalized Wiener process. It has constant drift rate $(\mu - \frac{1}{2}\sigma^2)$ and constant variance rate σ^2.

Finally, integrate between [t, T] to get the solution S_T:

$$\int_t^T d\ln(S) = (\mu - \frac{1}{2}1, \sigma^2) \int_t^T du + \sigma \int_t^T dW(t) \tag{A.9}$$

$$\ln(S_T) - \ln(S_t) = (\mu - \frac{1}{2}1, \sigma^2)\,(T - t) + \sigma(W_T W_t) \tag{A.10}$$

$$S_T = S_t\, e^{(\mu - \frac{1}{2}\,\sigma^2)(T-t) + \sigma\,(W_T - W_t)} \tag{A.11}$$

□

It has been demonstrate that the GBM (A.2) is the solution of the SDE (2.23).

The chance in logarithm of the stochastic process $\ln(S)$ between time T and 0 is normally distributed:

$$\ln(\frac{S_T}{S_t}) \sim N\left[(\mu - \frac{1}{2}\,\sigma^2)\,(T - t)\,;\, \sigma^2(T - t)\right] \tag{A.12}$$

this means that S_T is normally distributed:

$$\ln(\frac{S_T}{S_t}) \sim \ln N\left[(\mu - \frac{1}{2}\,\sigma^2)\,(T - t)\,;\, \sigma^2(T - t)\right] \tag{A.13}$$

Appendix B

Rserve extension

Rserve is an R package developed by Simon Urbanek [70] useful to connect R with other software. More precisely, *Rserve* is a socket server (TCP/IP or local sockets) which allows other programs to use facilities of R from various languages such as C/C++, PHP and Java. Integrate the computational capabilities of R into other programs is a huge improvement in generating statistical models in languages which do not provide similar functions. As was explained in Chapter 6, R is a software environment which provides a wide variety of statistical (time series analysis, linear and non linear modelling, etc.) and graphical technique and is one of the open–source software for statistical analysis more indicated by textbooks.

Rserve is not just a package, but it acts as an application with the goal to offer an interface by which programs can perform computations in R. The connection between *Rserve* and the client is obtained via network sockets, usually over TCP/IP protocol. This permits the use of a central *Rserve* from remote computers, the employment of several *Rserve* by the remote client to distribute computation but also local communication on a single computer. One *Rserve* can serve more clients simultaneously (exception to Windows operating systems, which supports only a connection at a time) and each connection has its own working directory and data space. This means that objects created in one connection can not affect other connection and that each application opened can process in parallel. Notice that not only the connections are independent, but also the R system must be separated from the application itself, in order to avoid any dependence.

Speed is a crucial element when designing a new system. In order to get speed, the data transfer between the application and *Rserve* is performed in binary form. *Rserve* allows to create objects in R, evaluate the R code and return results from R to application. Intermediate objects are stored in *Rserve*. The contents of the objects are sent (returned) in binary form from the client to the server (from the server to the client). On the contrary, R code is sent in clear text to the server.

A representative purpose of *Rserve* package is to sent all necessary data to R, perform computations and give back the results. All data and object exist until connection between client and server is open.

Moreover, *Rserve* has an integrated authentication protocol which provides to add a

level of security for remote use.

Summarizing , the main features of *Rserve* are listed below.

- *fast*, programs do not need to initialise R or link to the R library
- *remote connection*, *Rserve* allows remote connection
- *binary transport*, the request to be sent to R are binary data
- *automatic type conversion*, R data types are converted in original language data type
- *separation of client and R environments*, every connection has a separate workspace and working directory. As the client is not linked to R, there are no threading problem like in RSJava
- *persistent*, every connection has its own namespace and each object crated persists until the connection is closed
- *security*, *Rserve* provides strong security by supporting encrypted user–password authentication. Rserve can be also configured to accept local connections only
- *configurable*,one configuration file is employed to control settings and to enable–disable options such as authorization, remote access or files transfer.

Rserve has to be installed as a regular R package. In order to install and run the package, the commands to be wrote in R window are the following:

```
> install.packages("Rserve")
> library(Rserve)
> Rserve()
```

Rserve runs in local mode by default and once the package is running any programs can use its capabilities.

Now that a global framework of *Rserve* has been explain, the subject can be moved onto how embedding R in NetLogo. Platform for implementing Agent-Based Models (NetLogo) and for statistical analysis (R) are separated and this represents a huge limit. The NetLogo *Rserve*–extension offers primitives to use the software GNU R via the *Rserve* package in NetLogo models.

Previously, it has been explained how install the package in R; now the focus is on how embed R in NetLogo. The procedure is simple but needs two steps: first of all one has to type

```
extensions [rserve]
```

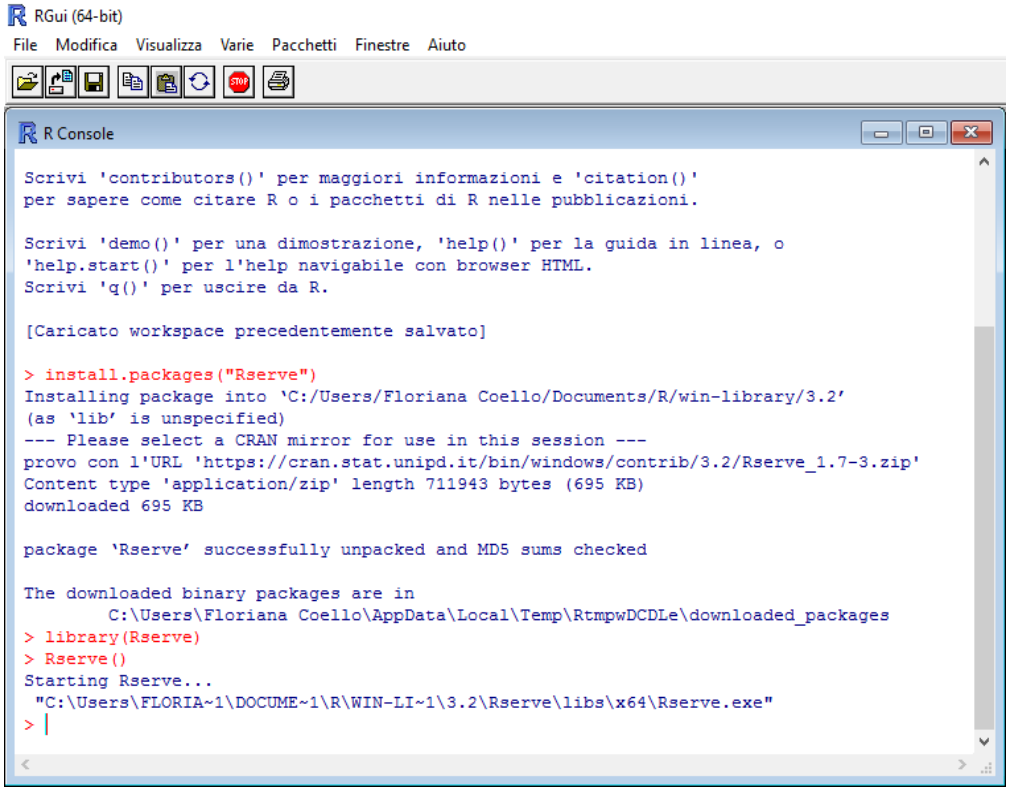


Figure B.1: R code to install and run *Rserve*

at the top of the NetLogo code. Then, one has to connect NetLogo to an *Rserve* server with the *rserve:init* primitive, followed by the number of the local host; the default TCP port is 6311 but can be modified. This primitive can be included into the pseudo-code of a procedure or a specific button can be created to include it. Every time one wants to open or close a connection, you need to press the buttons that include the following procedures respectively:

```
rserve:init 6311 "localhost"
rserve:close
```

Rserve in not the only package providing connection between R and NetLogo. An alternative to *Rserve* is the package *R*, which provides the interchange of data with R and adds statistical R functions in NetLogo as *Rserve*. The difference between *Rserve*–extension and *R*–extension is the technique to communicate with R. The first one communicates via a network connection with an Rserve server, while the other uses a direct path via the R package ″rJava‟. Furthermore, *Rserve* allows to connect not only to local servers but also to remote servers and this means that more users can share the same R installation via a network connection, not possible with *R*–extension.

Both extensions share the same syntax. Following the main primitives to data transfer and use R capabilities in NetLogo.

- *rserve:put*, sends NetLogo variables to R
- *rserve:get*, gets values from R
- *rserve:putagent*, sends a list of variables of NetLogo agents to R
- *rserve:putdataframe*, sends to R a dataframe of NetLogo variables and values (many functions needs dataframe as input)
- *rserve:putagentdf*, sends a dataframe of variables of NetLogo agents to R
- *rserve:putlist*, sends to R a list of NetLogo variables and values
- *putnamelist*, same as *rserve:putlist*, but creates a named R list
- *rserve:eval*, executes an R function
- *rserve:clear*, clears the R workspace and delete all variables

If one uses *R*–extension, it needs to replace *rserve* with *r*.

Some examples of use of *Rserve*–extension are included in the examples folder. Figure B.2 and Figure B.3 show the interface and the code of a NetLogo model which creates a link between R: the program gets from R ten random numbers, crates in R a list containing information about NetLogo agents and determines the correlation coefficient between turtles coordinates.

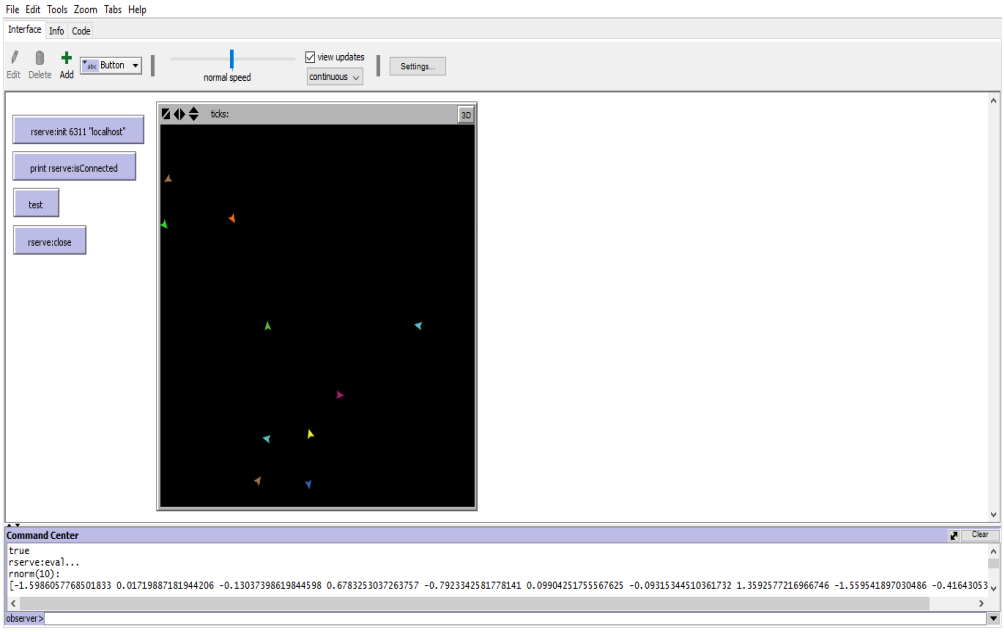


Figure B.2: Interface of the *example1* in *Rserve* example folder

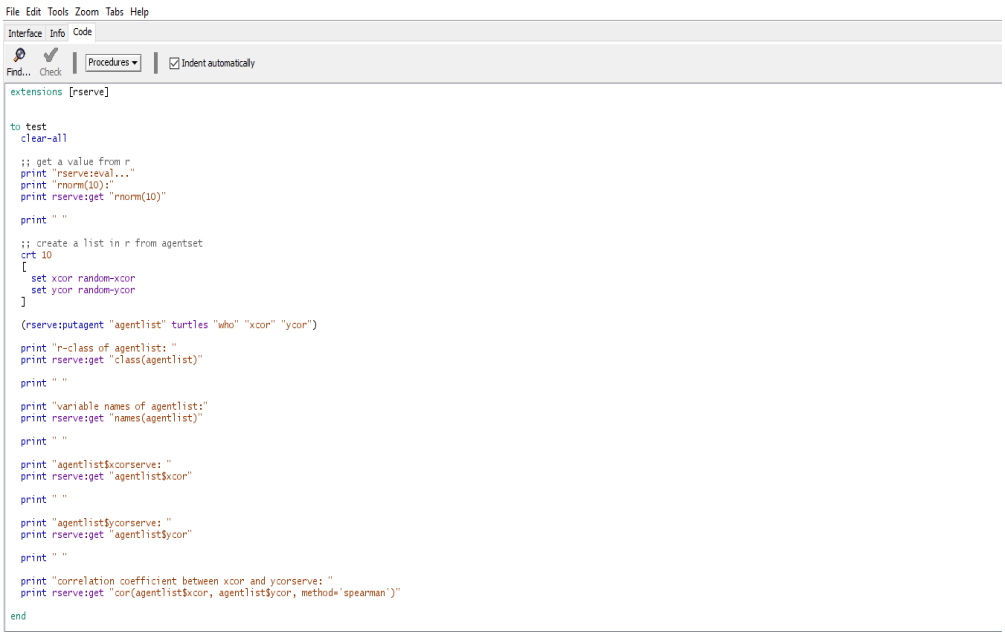


Figure B.3: Code of the *example1* in *Rserve* example folder

Appendix C

NetLogo Code

C.1 OptionsMarket pricing via Bs

The NetLogo code shown below represents the *OptionsMarket* model explained in Section 6.2. This options market is made up by random (*randomAgents*) and intelligent (*PJAgents*) agents, which trade call or put options evaluating prices via Black–Scholes formula. The spot price is updated using Gaussian Brownian Motion.

```
breed [randomAgents randomAgent]
breed [PJAgents PJAgent]

PJAgents-own [
              patience judgement
              guessed-sigma
              price
              Sell Buy
              maxSigma minSigma
              money
              options
              d1 d2
              callPrice
              totalProfit
              totalProfitsq
              stdDevProfit
              maxProfit minProfit
              optPrice
              ]

randomAgents-own [
                  guessed-sigma
                  price
```

```
                  Sell Buy
                  maxSigma minSigma
                  money
                  options
                  d1 d2
                  callPrice
                  totalProfit
                  totalProfitsq
                  stdDevProfit
                  maxProfit minProfit
                  optPrice
                ]

globals [
         dS
         dW
         ttm
         retVal retList
         marketPrice
         Overalld1 Overalld2
         pricingBias
         bsPrice
         marketDayCount
         strike
         marketOpen
         totalBought totalSold
         range
         nthMarket
         spot
         maxTotalProfit minTotalProfit
         maxAvgProfit minAvgProfit
         bestJudgement worstJudgement
         bestPatience worstPatience
         pricingBiasStdDev
         meanPricingBias
         pricingBiascount
         totalPricingBias
         totalPricingBiassq
         tradingAgents
         ]

to setup
  clear-all
```

```
ask patches [set pcolor 92]
if seed != 0 [ random-seed seed ]
set ttm -1
ask turtles [ set totalProfit 0 ]
ask turtles [ set maxProfit 0 ]
ask turtles [ set minProfit 0 ]
set totalPricingBias 0
set pricingBiascount 0
set totalPricingBiassq 0
set maxTotalProfit 0
set minTotalProfit 0
set maxAvgProfit 0
set minAvgProfit 0

create-randomAgents nRandomAgents
let side sqrt nRandomAgents
let step world-width / side
ask randomAgents
[set shape "sheep"
 set size 2
 set color gray]

let n 0
let x step / 2
let y step / 2

while [n < nRandomAgents]
[
ask randomAgent n [setxy x y]
set x x + step
if x > world-width [set x step / 2
set y y + step]
if y > world-width [set y 0.20 + step / 2]
set n n + 1
]

ask n-of nPJAgents randomAgents
[ set breed PJAgents
set shape "person"
set color gray
set patience minPatience + random (maxPatience - minPatience)
set judgement minJudgement + random-float (maxJudgement - minJudgement)
]
```

```
  set tradingAgents (turtle-set randomAgents PJAgents)

  reset-ticks
end

to go
  ifelse ((nthMarket - 1) < finalMarket) [
  if ttm <= 0 [
  ask tradingAgents [
   let profit (money + (options * marketPrice))
   set totalProfit (totalProfit + profit)
   set totalProfitsq (totalProfit * totalProfit)
   if nthMarket > 0
   [ set stdDevProfit sqrt( (TotalProfitSq / nthMarket)
         - ((TotalProfit / nthMarket)
         * (TotalProfit / nthMarket)) ) ]
     set maxProfit (totalProfit + stdDevProfit)
     set minProfit (totalProfit - stdDevProfit)
    ]
  if nthMarket > 0
  [ set range (max [ abs (totalProfit / nthMarket) ]
              of turtles) * 1.10 + 10 ]
    set totalPricingBias (totalPricingBias + pricingBias)
    set totalPricingBiassq (totalPricingBiassq +
        (PricingBias * PricingBias))
    if pricingBiasCount > 0
    [ set meanPricingBias (TotalPricingBias / pricingBiasCount)
      set pricingBiasStdDev sqrt(totalPricingBiasSq / pricingBiasCount
          - meanPricingBias * meanPricingBias) ]
    set pricingBiasCount (pricingBiasCount + 1)

    set bestJudgement [judgement] of max-one-of PJAgents [totalProfit]
    set worstJudgement [judgement] of min-one-of PJAgents [totalProfit]
    set bestPatience [patience] of max-one-of PJAgents [totalProfit]
    set worstPatience [patience] of min-one-of PJAgents [totalProfit]

  if (nthMarket > 0)
  [ set maxTotalProfit (max [ totalProfit ] of turtles ) / nthMarket
    set minTotalProfit (min [ totalProfit ] of turtles ) / nthMarket
    set maxAvgProfit (maxTotalProfit / nthMarket)
    set minAvgProfit (minTotalProfit / nthMarket) ]
```

```
  plot-profit
  setup-market
  set nthMarket (nthMarket + 1)
  ]
run-market
  tick
  ]
  [ stop ]
end

to setup-market
  set totalBought 0
  set totalSold 0
  set marketOpen 0
  set marketPrice 0
  ask turtles [ set callPrice 0 ]
  set pricingBias 0
  set-current-plot "Price"
  clear-plot
  set-plot-x-range 0 ceiling (time / dT)
  set-current-plot "Options Market Line Graph"
  clear-plot
  set-plot-x-range 0 ceiling (time / dT)
  set-current-plot "Sigma plot"
  clear-plot
  set-plot-x-range 0 ceiling (time / dT)
  set-current-plot "Activity"
  clear-plot
  set-plot-x-range 0 ceiling (time / dT)
  set marketDayCount 0
  set strike startingPrice
  set spot startingPrice
  set retList [ 0 0 ]
  set retList lput spot retList
  set ttm time
  ask tradingAgents [
  set minSigma 0.0
  set maxSigma 0.0
  set Buy 0
  set Sell 0
  set money 0
  set options 0
  set optPrice 0
```

```
  ]
end

to run-market
  set totalBought 0
  set totalSold 0
  update-spot
  update-turtlePrices
  if marketOpen = 1 [ set-marketPrice ]
  buyAndSell

  if putMarket = false [ set bsPrice calculate-callPrice sigma ]
  if putMarket = true [ set bsPrice calculate-putPrice sigma ]

  if marketOpen = 1 [ set pricingBias (marketPrice - bsPrice) ]

  set ttm ttm - dt
  set marketDayCount (marketDayCount + 1)

  if marketOpen = 0 [
  if marketDayCount > (max [ patience ] of PJAgents) + 40 [
   set marketOpen 1
   ask tradingAgents [ set maxSigma guessed-sigma ]
   ask tradingAgents [ set minSigma guessed-sigma ]
   ]
  ]
end

to update-spot
  set dW random-normal 0 sqrt(dt)
  set dS spot * (miu * dt + sigma * dW)
  set spot spot + dS
  set retVal dS / spot
  set retList lput retVal retList
end

to update-turtlePrices
  ask tradingAgents [ calculate-sigma
  if marketOpen = 1 [
  if guessed-sigma > maxSigma
   [set maxSigma guessed-sigma
    if putMarket = false [set Sell calculate-callPrice guessed-sigma]
    if putMarket = true [set Sell calculate-putPrice guessed-sigma]]
```

```
  if guessed-sigma < minSigma
   [set minSigma guessed-sigma
    if putMarket = false [set Buy calculate-callPrice guessed-sigma]
    if putMarket = true [set Buy calculate-putPrice guessed-sigma]]

   if putmarket = false [set optPrice calculate-callPrice guessed-sigma]
   if putmarket = true [set optPrice calculate-putPrice guessed-sigma]
    ] ]
end

to calculate-sigma
  ifelse breed = randomAgents [ set guessed-sigma random-float 1]
         [ if length retList > patience + 2 [ set guessed-sigma
           ((standard-deviation sublist retList
           (length retList - patience) (length retList))
            /(sqrt(dt)) * judgement) ]]
end

to plot-profit
  ask tradingAgents[
  if nthMarket > 0 [ let avgProfMag abs (totalProfit / nthMarket)
  if (totalProfit / nthMarket) > 0 [ set color scale-color
      green avgProfMag 0 (range) ]
  if (totalProfit / nthMarket) <= 0 [ set color scale-color
      red avgProfMag 0 (range) ] ]

  if breed = PJAgents[
  if (patience = worstPatience) [ if (judgement = worstJudgement)
              [ set color yellow ] ]
  if (patience = bestPatience) [ if (judgement = bestJudgement)
              [ set color blue ] ]
  ]
  ]

end

to set-marketPrice
  let maxBuy max [ Buy ] of turtles
  let minSell min [ Sell ] of turtles
  set marketPrice (maxBuy + minSell) / 2
 end
```

```
to buyAndSell
if marketOpen = 1 [
 ask tradingAgents [
  let patiencetmp 0
  if breed = PJAgents [ set patiencetmp patience]

 if length retList > patiencetmp + 2 [
  if Buy >= marketPrice [
   set options options + 1
   set money money - marketPrice
   set totalBought totalBought + 1
   ]
  if Sell <= marketPrice [
   set options options - 1
   set money money + marketPrice
   set totalSold totalSold + 1
   ]]
   ]]
end

to-report calculate-d1 [ _sigma ]
  let _d1 ((ln (spot / strike)) + (noRisk + (_sigma * _sigma) / 2) * ttm)
      /(_sigma * sqrt(ttm))
  report _d1
end

to-report calculate-d2 [ _d1 _sigma ]
  let _d2 (_d1 - (_sigma * sqrt(ttm)))
  report _d2
end

to-report calculate-callPrice [ _sigma ]
  let _d1 calculate-d1 _sigma
  let _d2 calculate-d2 _d1 _sigma
  let _callPrice spot * normcdf(_d1)
        - strike * exp( - noRisk * ttm) * normcdf(_d2)
  report _callPrice
end

to-report calculate-putPrice [ _sigma ]
  let _d1 calculate-d1 _sigma
  let _d2 calculate-d2 _d1 _sigma
  let _putPrice strike * exp( - noRisk * ttm) * normcdf(- _d2)
```

```
        - spot * normcdf(- _d1)
  report _putPrice
end

to-report normcdf [lclx]
  let lclt lclx
  let y 0.5 * erfcc ((-1) * lclt / ( 1 * sqrt 2.0))
  if ( y > 1.0 ) [ set y 1.0 ]
report y
end

to-report normpdf [lclx]
  report 1 / ( sqrt (2 * pi)) * exp ( - lclx * lclx / 2.0)
end

to-report erfcc [lclx]
  let z abs lclx
  let lclt 1.0 / (1.0 + 0.5 * z)
  let r lclt *  exp ( - z * z - 1.26551223 + lclt *
      (1.00002368 + lclt * (0.37409196 + lclt * (0.09678418 + lclt *
      (-0.18628806 + lclt * (.27886807 + lclt * (-1.13520398 + lclt *
      (1.48851587 + lclt * (-0.82215223 + lclt * .17087277 ))))))))))
  ifelse (lclx >= 0) [ report r ] [report 2.0 - r]
end
```

C.2 OptionsMarket mimicking BS via Neural Networks

C.2.1 One Neural Network

The following code refers to the model in which agents (random and intelligent) evaluate options prices using a "global" neural network which predict BS prices. As explained in Subsection 6.3.1, only a percentage of traders update lists of data that will be used to train the neural network. This means that all agents have the same experience and then evaluate options in the same manner.

```
extensions [rserve]

breed [randomAgents randomAgent]
breed [PJAgents PJAgent]
```

```
PJAgents-own [
              patience judgement
              guessed-sigma
              price
              Sell Buy
              maxSigma minSigma
              money
              options
              d1 d2
              callPrice
              totalProfit
              totalProfitsq
              stdDevProfit
              maxProfit minProfit
              optPrice
              output
              ]

randomAgents-own [
                  guessed-sigma
                  price
                  Sell Buy
                  maxSigma minSigma
                  money
                  options
                  d1 d2
                  callPrice
                  totalProfit
                  totalProfitsq
                  stdDevProfit
                  maxProfit minProfit
                  optPrice
                  output
                  ]

globals [
         dS dW
         ttm
         retVal retList
         marketPrice
         Overalld1 Overalld2
         pricingBias
```

```
          bsPrice
          marketDayCount
          strike
          marketOpen
          totalBought totalSold
          range
          nthMarket
          spot
          maxTotalProfit minTotalProfit
          maxAvgProfit minAvgProfit
          bestJudgement worstJudgement
          bestPatience worstPatience
          pricingBiasStdDev meanPricingBias
          pricingBiascount
          totalPricingBias
          totalPricingBiassq
          tradingAgents
          spotList
          ttmList
          news
          news2
          nnprice
          gsigList
          err
          nprice
          outputList
          scaledOutputList
          min1 min2 min3 min4
          max1 max2 max3 max4
           ]

to setup
  clear-all
  ask patches [set pcolor 92]
  if seed != 0 [ random-seed seed ]
  set ttm -1
  ask turtles [ set totalProfit 0 ]
  ask turtles [ set maxProfit 0 ]
  ask turtles [ set minProfit 0 ]
  set totalPricingBias 0
  set pricingBiascount 0
  set totalPricingBiassq 0
  set maxTotalProfit 0
```

```
  set minTotalProfit 0
  set maxAvgProfit 0
  set minAvgProfit 0

  create-randomAgents nRandomAgents
  let side sqrt nRandomAgents
  let step world-width / side
  ask randomAgents
  [ set shape "sheep"
  set size 2
  set color gray ]

  let n 0
  let x step / 2
  let y step / 2

  while [n < nRandomAgents]
  [
  ask randomAgent n [setxy x y]
  set x x + step
  if x > world-width [set x step / 2
  set y y + step]
  if y > world-width [set y 0.20 + step / 2]
  set n n + 1
  ]

  ask n-of nPJAgents randomAgents
  [set breed PJAgents
  set shape "person"
  set color gray
  set patience minPatience + random (maxPatience - minPatience)
  set judgement minJudgement + random-float
  (maxJudgement - minJudgement)
  ]
  set tradingAgents (turtle-set randomAgents PJAgents)
  reset-ticks
end

to go
  ifelse ((nthMarket - 1) < finalMarket) [
   if ttm <= 0 [
    ask tradingAgents [
     let profit (money + (options * marketPrice))
```

```
    set totalProfit (totalProfit + profit)
    set totalProfitsq (totalProfit * totalProfit)
    if nthMarket > 0 [
     set stdDevProfit sqrt( (TotalProfitSq / nthMarket) -
                          ((TotalProfit / nthMarket)
                           * (TotalProfit / nthMarket)))]
     set maxProfit (totalProfit + stdDevProfit)
     set minProfit (totalProfit - stdDevProfit)
           ]
   if nthMarket > 0 [set range (max [ abs (totalProfit / nthMarket)]
                                of turtles) * 1.10 + 10 ]
    set totalPricingBias (totalPricingBias + pricingBias)
    set totalPricingBiassq (totalPricingBiassq +
                          (PricingBias * PricingBias))
   if pricingBiasCount > 0 [
    set meanPricingBias (TotalPricingBias / pricingBiasCount)
    set pricingBiasStdDev sqrt( totalPricingBiasSq / pricingBiasCount
                         - meanPricingBias * meanPricingBias) ]
   set pricingBiasCount (pricingBiasCount + 1)

   set bestJudgement [judgement] of max-one-of PJAgents [totalProfit]
   set worstJudgement [judgement] of min-one-of PJAgents [totalProfit]
   set bestPatience [patience] of max-one-of PJAgents [totalProf]
   set worstPatience [patience] of min-one-of PJAgents [totalProfit]

   if (nthMarket > 0) [
    set maxTotalProfit (max [ totalProfit ] of turtles )
                                          / nthMarket
    set minTotalProfit (min [ totalProfit ] of turtles )
                       / nthMarket
    set maxAvgProfit (maxTotalProfit / nthMarket)
    set minAvgProfit (minTotalProfit / nthMarket) ]

  plot-profit
  setup-market
  set nthMarket (nthMarket + 1)
 ]
run-market
tick

if nthMarket = 1 [if ttm > (dt * window) and ttm < (dt * (window + 1))
                 [ make_datatrain
                   train ]
```

```
    ]
  ]
  [ stop ]
end

to setup-market
 set totalBought 0
 set totalSold 0
 set marketOpen 0
 set marketPrice 0
 ask turtles [ set callPrice 0 ]
 set pricingBias 0
 set-current-plot "Price"
 clear-plot
 set-plot-x-range 0 ceiling (time / dT)
 set-current-plot "Options Market Line Graph"
 clear-plot
 set-plot-x-range 0 ceiling (time / dT)
 set-current-plot "Sigma plot"
 clear-plot
 set-plot-x-range 0 ceiling (time / dT)
 set-current-plot "Activity"
 clear-plot
 set-plot-x-range 0 ceiling (time / dT)
 set marketDayCount 0
 set strike startingPrice
 set spot startingPrice
 set retList [ 0 0 ]
 set retList lput spot retList
 set spotList [ ]
 set ttm time
 ask tradingAgents [
  set minSigma 0.0
  set maxSigma 0.0
  set Buy 0
  set Sell 0
  set money 0
  set options 0
  set gsigList [ ]
  set ttmList [ ]
  set outputList [ ]
  set output 0.0
  ]
```

```
end

to run-market
  ifelse marketOpen = 1 [ set news "Open"] [set news "Close"]
   set totalBought 0
   set totalSold 0

   update-spot
   update-turtlePrices
   if marketOpen = 1 [ set-marketPrice ]
   buyAndSell

   if marketOpen = 1 [
   if  nthMarket = 1 [ let l 0
    while [ l < (percentage * nRandomAgents / 100 ) ]
          [ let o random nRandomAgents
            ask turtle o [set spotList lput spot spotList
            set ttmList lput ttm ttmList
            set gsigList lput guessed-sigma gsigList
            output-train
            set outputList lput output outputList]
            set l l + 1 ]
           ] ]

   if putMarket = false [ set bsPrice calculate-callPrice sigma
   if nthMarket > 1 [ set nnprice predict spot sigma ttm]
   ]
  if putMarket = true [ set bsPrice calculate-putPrice sigma
  if nthMarket > 1 [ set nnprice predict spot sigma ttm
  set err abs(bsPrice - nnprice)]
  ]

  if marketOpen = 1 [ set pricingBias (marketPrice - bsPrice) ]

  set ttm ttm - dt
  set marketDayCount (marketDayCount + 1)
  if marketOpen = 0 [
   if marketDayCount > (max [ patience ] of PJAgents) + 40 [
        set marketOpen 1
        ask tradingAgents [ set maxSigma guessed-sigma ]
        ask tradingAgents [ set minSigma guessed-sigma ] ]
      ]
end
```

```
to update-spot
  set dW random-normal 0 sqrt(dt)
  set dS spot * (miu * dt + sigma * dW)
  set spot spot + dS

  set retVal dS / spot
  set retList lput retVal retList
end

to update-turtlePrices
  ask tradingAgents [ calculate-sigma
   if marketOpen = 1 [
    if guessed-sigma > maxSigma
     [ set maxSigma guessed-sigma
      if putMarket = false [set Sell calculate-callPrice guessed-sigma]
      if putMarket = true [set Sell calculate-putPrice guessed-sigma]]

    if guessed-sigma < minSigma
     [ set minSigma guessed-sigma
     if putMarket = false [set Buy calculate-callPrice guessed-sigma]
     if putMarket = true [set Buy calculate-putPrice guessed-sigma]
    ]
   if putmarket = false [set optPrice calculate-callPrice guessed-sigma]
   if putmarket = true [set optPrice calculate-putPrice guessed-sigma]
   ] ]
end

to calculate-sigma
  ifelse breed = randomAgents [ set guessed-sigma random-float 1]
                              [ if length retList > patience + 2
                              [ set guessed-sigma ((standard-deviation
                               sublist retList (length retList - patience)
                               (length retList))/(sqrt(dt)) * judgement)]]
end

to plot-profit
  ask tradingAgents[
   if nthMarket > 0 [ let avgProfMag abs (totalProfit / nthMarket)
   if (totalProfit / nthMarket) > 0 [ set color scale-color green
                                      avgProfMag 0 (range) ]
   if (totalProfit / nthMarket) <= 0 [ set color scale-color red
                                       avgProfMag 0 (range) ] ]
```

```
   if breed = PJAgents[
   if (patience = worstPatience) [ if (judgement = worstJudgement)
                                   [ set color yellow ] ]
   if (patience = bestPatience) [ if (judgement = bestJudgement)
                                  [ set color blue ] ]
  ]
]
end

to set-marketPrice
  sell prices and finding average
  let maxBuy max [ Buy ] of turtles
  let minSell min [ Sell ] of turtles
  set marketPrice (maxBuy + minSell) / 2
end

to buyAndSell
  if marketOpen = 1 [
   ask tradingAgents [
    let patiencetmp 0
    if breed = PJAgents [ set patiencetmp patience]

    if length retList > patiencetmp + 2 [
     if Buy >= marketPrice [
      set options options + 1
      set money money - marketPrice
      set totalBought totalBought + 1
       ]
     if Sell <= marketPrice [
      set options options - 1
      set money money + marketPrice
      set totalSold totalSold + 1
       ]
      ]
     ]
    ]
end

to output-train
  if putMarket = false [ set output calculate-callPrice guessed-sigma ]
  if putMarket = true  [ set output calculate-putPrice guessed-sigma ]
end
```

```
to-report calculate-d1 [ _sigma ]
  let _d1 ((ln (spot / strike)) +
          (noRisk + (_sigma * _sigma) / 2) * ttm) / (_sigma * sqrt(ttm))
  report _d1
end

to-report calculate-d2 [ _d1 _sigma ]
  let _d2 (_d1 - (_sigma * sqrt(ttm)))
  report _d2
end

to-report calculate-callPrice [ _sigma ]
  let _d1 calculate-d1 _sigma
  let _d2 calculate-d2 _d1 _sigma
  let _callPrice spot * normcdf(_d1)
       - strike * exp( - noRisk * ttm) * normcdf(_d2)
  report _callPrice
end

to-report calculate-putPrice [ _sigma ]
  let _d1 calculate-d1 _sigma
  let _d2 calculate-d2 _d1 _sigma
  let _putPrice strike * exp( - noRisk * ttm) * normcdf(- _d2)
      - spot * normcdf(- _d1)
  report _putPrice
end

to-report normcdf [lclx]
  let lclt lclx
  let y 0.5 * erfcc ((-1) * lclt / ( 1 * sqrt 2.0))
  if ( y > 1.0 ) [ set y 1.0 ]
  report y
end

to-report normpdf [lclx]
  report 1 / ( sqrt (2 * pi)) * exp ( - lclx * lclx / 2.0)
end

to-report erfcc [lclx]
  let z abs lclx
  let lclt 1.0 / (1.0 + 0.5 * z)
  let r lclt *  exp ( - z * z - 1.26551223 + lclt * (1.00002368 + lclt *
```

```
    (0.37409196 + lclt * (0.09678418 + lclt * (-0.18628806 + lclt *
    (.27886807 + lclt * (-1.13520398 + lclt * (1.48851587 + lclt *
    (-0.82215223 + lclt * .17087277 ))))))))
  ifelse (lclx >= 0) [ report r ] [report 2.0 - r]
end

to make_datatrain
  rserve:put "spotList" spotList
  rserve:put "gsigList" gsigList
  rserve:put "ttmList" ttmList
  rserve:put "outputList" outputList
  rserve:eval "S<-as.numeric(spotList)"
  rserve:eval "sigma<-as.numeric(gsigList)"
  rserve:eval "ttm<-as.numeric(ttmList)"
  rserve:eval "OptionPrice<-as.numeric(outputList)")
  rserve:eval "datatrain<-cbind(S, sigma, ttm, OptionPrice)"
  rserve:eval "colnames(datatrain)<-c('S','sigma','ttm','OptionPrice')"
  rserve:eval "maxsd<-apply(datatrain, 2, max)"
  rserve:eval "minsd<-apply(datatrain, 2, min)"
  rserve:eval "scaledtrain<-as.data.frame (scale(datatrain,
              center = minsd , scale = maxsd - minsd))"
  set min1 rserve:get "minsd[1]"
  set max1 rserve:get "maxsd[1]"
  set min2 rserve:get "minsd[2]"
  set max2 rserve:get "maxsd[2]"
  set min3 rserve:get "minsd[3]"
  set max3 rserve:get "maxsd[3]"
  set min4 rserve:get "minsd[4]"
  set max4 rserve:get "maxsd[4]"
end

to train
  rserve:eval "nn<-neuralnet ( OptionPrice~S+sigma+ttm, scaledtrain,
                              hidden = 3, threshold = 0.03, rep=10)"
  rserve:eval "rn<-which.min(nn$result.matrix[1,])"
  rserve:eval "plot(nn, rep = rn)"
end

to-report predict [-spot -sigma -ttm ]
  let sspot (( -spot - min1 ) / ( max1 - min1 ))
  let ssigma (( -sigma - min2 ) / ( max2 - min2 ))
  let sttm (( -ttm - min3 ) / ( max3 - min3 ))
  (rserve:putdataframe "data" "S" sspot  "sigma" ssigma "ttm" sttm )
```

```
  rserve:eval "nprice<-compute(nn, data, rep = rn)$net.result[1,1]"
  set nprice rserve:get "nprice"
  let -nnprice  nprice * ( max4 - min4 ) + min4
  report -nnprice
end
```

C.2.2 More Neural Networks

Here below the NetLogo code of the model simulating an options market in which agents (random and intelligent) evaluate prices through neural networks trained via Black–Scholes formula. Each trader has its own neural networks, which is trained with different data and different time, in order to represent heterogeneity of real markets. Full explanation can be find in Subsection 6.3.2.

```
extensions [rserve]

breed [randomAgents randomAgent]
breed [PJAgents PJAgent]

PJAgents-own [
              patience judgement
              guessed-sigma
              price
              Sell Buy
              maxSigma minSigma
              money
              options
              d1 d2
              callPrice
              totalProfit
              totalProfitsq
              stdDevProfit
              maxProfit minProfit
              optPrice
              output
              outputList
              gsigList
              min1 min2 min3 min4
              max1 max2 max3 max4
              rn
              ]

randomAgents-own [
                  guessed-sigma
```

```
                price
                Sell Buy
                maxSigma minSigma
                money
                options
                d1 d2
                callPrice
                totalProfit
                totalProfitsq
                stdDevProfit
                maxProfit minProfit
                optPrice
                output
                outputList
                gsigList
                min1 min2 min3 min4
                max1 max2 max3 max4
                rn
                ]

globals [
         dS
         dW
         ttm
         retVal retList
         marketPrice
         Overalld1 Overalld2
         pricingBias
         marketDayCount
         strike
         marketOpen
         totalBought totalSold
         range
         nthMarket
         spot
         maxTotalProfit minTotalProfit
         maxAvgProfit minAvgProfit
         bestJudgement worstJudgement
         bestPatience worstPatience
         pricingBiasStdDev
         meanPricingBias
         pricingBiascount
         totalPricingBias
```

```
          totalPricingBiassq
          tradingAgents
          news news2
          nprice
          ttmList
          spotList
           ]


to setup
  clear-all
  ask patches [set pcolor 92]
  if seed != 0 [ random-seed seed ]
  set ttm -1
  ask turtles [ set totalProfit 0 ]
  ask turtles [ set maxProfit 0 ]
  ask turtles [ set minProfit 0 ]
  set totalPricingBias 0
  set pricingBiascount 0
  set totalPricingBiassq 0
  set maxTotalProfit 0
  set minTotalProfit 0
  set maxAvgProfit 0
  set minAvgProfit 0

  create-randomAgents nRandomAgents
  let side sqrt nRandomAgents
  let step world-width / side
  ask randomAgents
  [ set shape "sheep"
    set size 2
    set color gray ]

  let n 0
  let x step / 2
  let y step / 2

  while [n < nRandomAgents]
   [
    ask randomAgent n [setxy x y]
    set x x + step
    if x > world-width [set x step / 2
    set y y + step]
```

```
    if y > world-width [set y 0.20 + step / 2]
    set n n + 1
   ]

  ask n-of nPJAgents randomAgents
  [set breed PJAgents
   set shape "person"
   set color gray
   set patience minPatience + random (maxPatience - minPatience)
   set judgement minJudgement + random-float
                                (maxJudgement - minJudgement)
   ]
  set tradingAgents (turtle-set randomAgents PJAgents)
  reset-ticks
end

to go
  ifelse ((nthMarket - 1) < finalMarket) [
   if ttm <= 0 [
    ask tradingAgents [
     let profit (money + (options * marketPrice))
     set totalProfit (totalProfit + profit)
     set totalProfitsq (totalProfit * totalProfit)
     if nthMarket > 0 [ set stdDevProfit
                        sqrt( (TotalProfitSq / nthMarket)
                        - ((TotalProfit / nthMarket)
                        * (TotalProfit / nthMarket)) ) ]
     set maxProfit (totalProfit + stdDevProfit)
     set minProfit (totalProfit - stdDevProfit)
      ]
   if nthMarket > 0
    [ set range (max [ abs (totalProfit / nthMarket) ] of turtles)
                * 1.10 + 10 ]
      set totalPricingBias (totalPricingBias + pricingBias)
      set totalPricingBiassq (totalPricingBiassq +
                             (PricingBias * PricingBias))
      if pricingBiasCount > 0 [ set meanPricingBias (TotalPricingBias
                                    / pricingBiasCount)
      set pricingBiasStdDev sqrt(totalPricingBiasSq / pricingBiasCount
                           - meanPricingBias * meanPricingBias) ]
      set pricingBiasCount (pricingBiasCount + 1)

      set bestJudgement [ judgement ]
```

```
                        of max-one-of PJAgents [ totalProfit ]
      set worstJudgement [ judgement ]
                         of min-one-of PJAgents [ totalProfit ]
      set bestPatience [ patience ]
                       of max-one-of PJAgents [ totalProfit ]
      set worstPatience [ patience ]
                        of min-one-of PJAgents [ totalProfit ]

    if (nthMarket > 0) [ set maxTotalProfit
    (max [ totalProfit ] of turtles ) / nthMarket
     set minTotalProfit (min [ totalProfit ] of turtles ) / nthMarket
     set maxAvgProfit (maxTotalProfit / nthMarket)
     set minAvgProfit (minTotalProfit / nthMarket) ]

  plot-profit
  setup-market
  set nthMarket (nthMarket + 1)
]

 run-market

 if nthMarket = 1 [
  if ttm > (dt * window) and ttm < (dt * (window + 1))
    [ ask tradingAgents [ make_datatrain
                          train ] ]
 ]
 tick
 ]
 [ stop ]

end

to setup-market
  set totalBought 0
  set totalSold 0
  set marketOpen 0
  set marketPrice 0
  ask turtles [ set callPrice 0 ]
  set pricingBias 0
  set-current-plot "Price"
  clear-plot
  set-plot-x-range 0 ceiling (time / dT)
  set-current-plot "Options Market Line Graph"
```

```
  clear-plot
  set-plot-x-range 0 ceiling (time / dT)
  set-current-plot "Sigma plot"
  clear-plot
  set-plot-x-range 0 ceiling (time / dT)
  set-current-plot "Activity"
  clear-plot
  set-plot-x-range 0 ceiling (time / dT)
  set marketDayCount 0
  set strike startingPrice
  set spot startingPrice
  set retList [ 0 0 ]
  set retList lput spot retList
  set spotList [ ]
  set ttm time
  ask tradingAgents [
  set minSigma 0.0
  set maxSigma 0.0
  set Buy 0
  set Sell 0
  set money 0
  set options 0
  set gsigList [ ]
  set ttmList [ ]
  set outputList [ ]
  set output 0.0
]
end

to run-market
  ifelse marketOpen = 1 [ set news "Open"] [set news "Close"]
  set totalBought 0
  set totalSold 0

  update-spot
  update-turtlePrices
  if marketOpen = 1 [ set-marketPrice ]
  buyAndSell

  if putMarket = false [ set bsPrice calculate-callPrice sigma ]
   if putMarket = true  [ set bsPrice calculate-putPrice sigma ]

  if marketOpen = 1 [ set pricingBias (marketPrice - bsPrice) ]
```

```
   if marketOpen = 1 [ if  nthMarket = 1  [set ttmList lput ttm ttmList]]

   set ttm ttm - dt
   set marketDayCount (marketDayCount + 1)

   if marketOpen = 0 [

 if marketDayCount > (max [ patience ] of PJAgents) + 40 [
  set marketOpen 1
  ask tradingAgents [ set maxSigma guessed-sigma ]
  ask tradingAgents [ set minSigma guessed-sigma ]
  ]
 ]

 end

 to update-spot
   set dW random-normal 0 sqrt(dt)
   set dS spot * (miu * dt + sigma * dW)
   set spot spot + dS

   set retVal dS / spot
   set retList lput retVal retList

   if marketOpen = 1 [
   if  nthMarket = 1  [ set spotList lput spot spotList ]]
 end

to update-turtlePrices
  ask tradingAgents [ calculate-sigma
   if marketOpen = 1 [
    if guessed-sigma > maxSigma
    [ set maxSigma guessed-sigma
      if putMarket = false [ ifelse nthMarket = 1
                             [set Sell calculate-callPrice guessed-sigma]
                             [set Sell predict spot guessed-sigma ttm]]
      if putMarket = true  [ ifelse nthMarket = 1
                             [set Sell calculate-putPrice guessed-sigma]
                             [set Sell predict spot guessed-sigma ttm]]
   ]

    if guessed-sigma < minSigma
```

```
    [ set minSigma guessed-sigma
      if putMarket = false [ ifelse nthMarket = 1
                               [set Buy calculate-callPrice guessed-sigma]
                               [set Buy predict spot guessed-sigma ttm]]
      if putMarket = true  [ ifelse nthMarket = 1
                               [set Buy calculate-putPrice guessed-sigma]
                               [set Buy predict spot guessed-sigma ttm]]
     ]
   if putmarket = false [set optPrice calculate-callPrice guessed-sigma]
   if putmarket = true [set optPrice calculate-putPrice guessed-sigma]

    if  nthMarket = 1 [ set gsigList lput guessed-sigma gsigList
    output-train
   set outputList lput output outputList]
 ] ]
end

to calculate-sigma
  ifelse breed = randomAgents [set guessed-sigma random-float 1]
                              [if length retList > patience + 2
                              [set guessed-sigma ((standard-deviation
                              sublist retList (length retList - patience)
                              (length retList))/(sqrt(dt)) * judgement)
                              ]]
end

to plot-profit
  ask tradingAgents[
  if nthMarket > 0 [ let avgProfMag abs (totalProfit / nthMarket)
  if (totalProfit / nthMarket) > 0 [ set color
   scale-color green avgProfMag 0 (range) ]
  if (totalProfit / nthMarket) <= 0 [ set color
   scale-color red avgProfMag 0 (range) ] ]

  if breed = PJAgents[
  if (patience = worstPatience) [ if (judgement = worstJudgement)
     [ set color yellow ] ]
  if (patience = bestPatience) [ if (judgement = bestJudgement)
     [ set color blue ] ]
  ]
 ]
end
```

```
to set-marketPrice
  sell prices and finding average
  let maxBuy max [ Buy ] of turtles
  let minSell min [ Sell ] of turtles
  set marketPrice (maxBuy + minSell) / 2
end

to buyAndSell
  if marketOpen = 1 [
   ask tradingAgents [
    let patiencetmp 0
    if breed = PJAgents [ set patiencetmp patience]

    if length retList > patiencetmp + 2 [
     if Buy >= marketPrice [
      set options options + 1
      set money money - marketPrice
      set totalBought totalBought + 1
       ]
     if Sell <= marketPrice [
      set options options - 1
      set money money + marketPrice
      set totalSold totalSold + 1
       ]
 ] ] ]
end

to output-train
  if putMarket = false [ set output calculate-callPrice guessed-sigma ]
  if putMarket = true  [ set output calculate-putPrice guessed-sigma ]
end

to-report calculate-d1 [ _sigma ]
  let _d1 ((ln (spot / strike)) + (noRisk + (_sigma * _sigma) / 2) * ttm)
        / (_sigma * sqrt(ttm)) ;;change variables
  report _d1
end

to-report calculate-d2 [ _d1 _sigma ]
  let _d2 (_d1 - (_sigma * sqrt(ttm)))
  report _d2
end
```

```
to-report calculate-callPrice [ _sigma ]
  let _d1 calculate-d1 _sigma
  let _d2 calculate-d2 _d1 _sigma
  let _callPrice spot * normcdf(_d1)
       - strike * exp( - noRisk * ttm) * normcdf(_d2)
  report _callPrice
end


to-report calculate-putPrice [ _sigma ]
  let _d1 calculate-d1 _sigma
  let _d2 calculate-d2 _d1 _sigma
  let _putPrice strike * exp( - noRisk * ttm) * normcdf(- _d2)
       - spot * normcdf(- _d1)
  report _putPrice
end

to-report normcdf [lclx]
  let lclt lclx
  let y 0.5 * erfcc ((-1) * lclt / ( 1 * sqrt 2.0))
  if ( y > 1.0 ) [ set y 1.0 ]
  report y
end

to-report normpdf [lclx]
  report 1 / ( sqrt (2 * pi)) * exp ( - lclx * lclx / 2.0)
end

to-report erfcc [lclx]
  let z abs lclx
  let lclt 1.0 / (1.0 + 0.5 * z)
  let r lclt *  exp ( - z * z - 1.26551223 +
   lclt * (1.00002368 + lclt * (0.37409196 + lclt * (0.09678418 +
   lclt * (-0.18628806 + lclt * (.27886807 + lclt * (-1.13520398 +
   lclt * (1.48851587 + lclt * (-0.82215223 + lclt * .17087277 ))))))))
  ifelse (lclx >= 0) [ report r ] [report 2.0 - r]
end

to make_datatrain
  rserve:put "spotList" spotList
  rserve:put "gsigList" gsigList
  rserve:put "ttmList" ttmList
  rserve:put "outputList" outputList
```

```
  rserve:eval "S<-as.numeric(spotList)"
  rserve:eval "sigma<-as.numeric(gsigList)"
  rserve:eval "ttm<-as.numeric(ttmList)"
  rserve:eval "OptionPrice<-as.numeric(outputList)"
  rserve:eval "f<-length(S)"
  rserve:eval "a<-sample(1:f, 1)"
  rserve:eval "S<-S[a:f]"
  rserve:eval "sigma<-sigma[a:f]"
  rserve:eval "ttm<-ttm[a:f]"
  rserve:eval "OptionPrice<-OptionPrice[a:f]"

  rserve:eval "datatrain<-cbind(S, sigma, ttm, OptionPrice)"
  rserve:eval "colnames(datatrain)<-c('S','sigma','ttm','OptionPrice')"
  rserve:eval "maxsd<-apply(datatrain, 2, max)"
  rserve:eval "minsd<-apply(datatrain, 2, min)"
  rserve:eval "scaledtrain<-as.data.frame(scale(datatrain, center = minsd,

                            scale = maxsd - minsd))"
  set min1 rserve:get "minsd[1]"
  set max1 rserve:get "maxsd[1]"
  set min2 rserve:get "minsd[2]"
  set max2 rserve:get "maxsd[2]"
  set min3 rserve:get "minsd[3]"
  set max3 rserve:get "maxsd[3]"
  set min4 rserve:get "minsd[4]"
  set max4 rserve:get "maxsd[4]"
end

to train
  rserve:eval "nn<-neuralnet ( OptionPrice~S+sigma+ttm, scaledtrain,
  hidden = 3, threshold = 0.03, rep=10)"
  rserve:eval "rn<-which.min(nn$result.matrix[1,])"
  rserve:eval "plot(nn, rep = rn)"
  rserve:put  "myNum" who
  rserve:eval "name<-paste('nn',myNum,sep='')"
  rserve:eval "assign(name,nn)"
  rserve:eval "print(name)"
  rserve:eval "print(get(name))"
  set rn rserve:get "rn"
end

to-report predict [-spot -sigma -ttm ]
  let sspot (( -spot - min1 ) / (max1 - min1 ))
```

```
  let ssigma (( -sigma - min2 ) / ( max2 - min2 ))
  let sttm (( -ttm - min3 ) / ( max3 - min3 ))
  rserve:put "rn" rn
  rserve:put "myNum" who
  rserve:eval "myNn<-paste('nn',myNum,sep='')"
  rserve:eval "nn <- get(myNn)"
  rserve:eval "print(nn)"
  let test rserve:get "myNn"
  rserve:eval "tt<-names(nn)[13]=='result.matrix'"
  show rserve:get "tt"

  (rserve:putdataframe "data" "S" sspot  "sigma" ssigma "ttm" sttm)
  rserve:eval "nprice<-compute(nn, data, rep = rn)$net.result[1,1]"
  set nprice rserve:get "nprice"
  let -nnprice  nprice * ( max4 - min4 ) + min4
  report -nnprice
end
```

C.3 OptionsMarket learning from stock market

Following code refers to the final model, in which options agents are now able to trade both calls and puts, and they evaluate prices according to a stock market. A CDA model has been introduced in the program in order to simulate a stock market: spot do not follow GBM, thus BS formula is no more valid and neural networks learn to predict from stock market. See Section 6.4 for a more detailed explanation.

```
extensions [rserve]

breed [randomAgents randomAgent]
breed [PJAgents PJAgent]
breed [SrandomAgents SrandomAgent]
breed [StrendAgents StrendAgent]

PJAgents-own [
              patience judgement
              guessed-sigma
              price
              money
              calls puts options
              d1 d2
              totalProfit
              totalProfitsq
              stdDevProfit
```

```
                maxProfit minProfit
                CoutputList PoutputList
                CgsigList PgsigList
                CspotList PspotList
                CttmList PttmList
                Cmin1 Cmin2 Cmin3 Cmin4
                Cmax1 Cmax2 Cmax3 Cmax4
                Pmin1 Pmin2 Pmin3 Pmin4
                Pmax1 Pmax2 Pmax3 Pmax4
                Crn Prn
                agree notagree
                SexePriceEND
                seller buyer
                putmarket callmarket
                ]

randomAgents-own [
                    price
                    money
                    calls puts options
                    d1 d2
                    totalProfit
                    totalProfitsq
                    stdDevProfit
                    maxProfit minProfit
                    CoutputList PoutputList
                    CgsigList PgsigList
                    CspotList PspotList
                    CttmList PttmList
                    Cmin1 Cmin2 Cmin3 Cmin4
                    Cmax1 Cmax2 Cmax3 Cmax4
                    Pmin1 Pmin2 Pmin3 Pmin4
                    Pmax1 Pmax2 Pmax3 Pmax4
                    Crn Prn
                    agree notagree
                    SexePriceEND
                    seller buyer
                    putmarket callmarket
                    ]

SrandomAgents-own[Sbuy Ssell Spass Sprice Scash Sstocks]
StrendAgents-own[Sbuy Ssell Spass Sprice Scash Sstocks]
```

```
globals [
         dS
         dW
         ttm
         strike
         retList retVal
         pricingBias
         marketDayCount
         marketOpen
         CtotalBought PtotalBought
         CtotalSold PtotalSold
         range
         nthTime
         maxTotalProfit minTotalProfit
         maxAvgProfit minAvgProfit
         bestJudgement worstJudgement
         bestPatience worstPatience
         pricingBiasStdDev
         meanPricingBias
         pricingBiascount
         totalPricingBias
         totalPricingBiassq
         tradingAgents
         news news2
         Cnprice Pnprice
         ClogB ClogS
         PlogB PlogS
         CexePrice CexePrice-1
         PexePrice PexePrice-1
         exePrice
         SlogB SlogS
         SexePrice SexePrice-1
         StradingAgents
         SpriceList
         StrendUp StrendDown
         SnumTrendUp SnumTrendDown
         Snews Snews2
         ]

to setup
  clear-all
  ask patches [set pcolor 92]
  if seed != 0 [ random-seed seed ]
```

```
set ttm -1

create-randomAgents nRandomAgents
let side sqrt nRandomAgents
let step world-width / side
ask randomAgents
[ set shape "sheep"
  set size 2
  set color gray ]

let n 0
let x step / 2
let y step / 2

while [n < nRandomAgents]
[
 ask randomAgent n [setxy x y]
 set x x + step
 if x > world-width [set x step / 2
 set y y + step]
 if y > world-width [set y 0.20 + step / 2]
 set n n + 1
 ]

 ask n-of nPJAgents randomAgents
 [ set breed PJAgents
   set shape "person business"
   set color gray
   set patience minPatience
                + random (maxPatience - minPatience)
   set judgement minJudgement
                 + random-float (maxJudgement - minJudgement)
  ]

 ask n-of nSrandomAgents randomAgents
 [ set breed SrandomAgents
   set shape "coin heads"
   set color gray ]

 ask n-of nStrendAgents randomAgents
 [ set breed StrendAgents
   set shape "computer server"
   set color gray ]
```

```
  set tradingAgents (turtle-set randomAgents PJAgents)
  set StradingAgents (turtle-set SrandomAgents StrendAgents)

  ask tradingAgents [ set totalProfit 0 ]
  ask tradingAgents [ set maxProfit 0 ]
  ask tradingAgents [ set minProfit 0 ]
  set maxTotalProfit 0
  set minTotalProfit 0
  set maxAvgProfit 0
  set minAvgProfit 0
  set PlogB [ ]
  set PlogS [ ]
  set ClogB [ ]
  set ClogS [ ]
  set SlogB [ ]
  set SlogS [ ]
  set Snews   "Open"
  set Snews2  "Close"

  reset-ticks
end

to go
  ifelse ((nthTime - 1) < finalMarket) [
  if ttm <= 0 [
   ask tradingAgents [
    let profit (money + (options * exePrice))
    set totalProfit (totalProfit + profit)
    set totalProfitsq (totalProfit * totalProfit)
    if nthTime > 0 [ set stdDevProfit sqrt( (TotalProfitSq / nthTime)
                         - ((TotalProfit / nthTime)
                         * (TotalProfit / nthTime)))]
    set maxProfit (totalProfit + stdDevProfit)
    set minProfit (totalProfit - stdDevProfit)
  ]
  if nthTime > 0
   [set range (max [ abs (totalProfit / nthTime) ] of tradingAgents)
                                                    * 1.10 + 10 ]
    set bestJudgement [judgement] of max-one-of PJAgents [totalProfit]
    set worstJudgement [judgement] of min-one-of PJAgents [totalProfit]
    set bestPatience [patience] of max-one-of PJAgents [totalProfit]
    set worstPatience [patience] of min-one-of PJAgents [totalProfit]
```

```
  if (nthTime > 0) [set maxTotalProfit (max [totalProfit] of tradingAgents)
                                        / nthTime
  set minTotalProfit (min [totalProfit] of tradingAgents ) / nthTime
  set maxAvgProfit (maxTotalProfit / nthTime)
  set minAvgProfit (minTotalProfit / nthTime)]

  setup-market

  set nthTime (nthTime + 1)
  ]

  run-market

  if nthTime = 1 [if ttm > (dt * window) and ttm < (dt * (window + 1))
     [ ask tradingAgents [ make_datatrain
                           train ] ]
  ]
  tick
  ]
 [ stop ]

 if SexePrice  <= 0 [
 set Snews Snews2
 stop
  ]
end

to setup-market
  set CtotalBought 0
  set CtotalSold 0
  set PtotalBought 0
  set PtotalSold 0
  set marketOpen 0
  set exePrice 0
  set strike startingPrice
  set pricingBias 0
  set-current-plot "Spot price"
  clear-plot
  set-plot-x-range 0 ceiling (time / dT)
  set-current-plot "Options Market Line Graph"
  clear-plot
  set-plot-x-range 0 ceiling (time / dT)
```

```
  set-current-plot "Sigma plot"
  clear-plot
  set-plot-x-range 0 ceiling (time / dT)
  set-current-plot "Activity"
  clear-plot
  set-plot-x-range 0 ceiling (time / dT)
  set marketDayCount 0

  set SexePrice 1000
  set SexePrice-1 SexePrice
  set PexePrice calculate-putPrice sigma SexePrice time
  set PexePrice-1 PexePrice
  set CexePrice calculate-callPrice sigma SexePrice time
  set CexePrice-1 CexePrice

  set retList [ 0 0 ]
  set retList lput SexePrice retList
  set ttm time
  ask tradingAgents [
  set money 0
  set options 0
  set PgsigList [ ]
  set CgsigList [ ]
  set CttmList [ ]
  set PttmList [ ]
  set CoutputList [ ]
  set PoutputList [ ]
  set CspotList [ ]
  set PspotList [ ]
  set options calls + puts
  ]
  set SpriceList [ ]
  set StrendDown False
  set StrendUp   False
  set SnumTrendUp   0
  set SnumTrendDown 0
end

to run-market
  ifelse marketOpen = 1 [ set news "Open"] [set news "Close"]
  set CtotalBought 0
  set CtotalSold 0
  set PtotalBought 0
```

```
  set PtotalSold 0

  update-turtlePrices

  set ttm ttm - dt
  set marketDayCount (marketDayCount + 1)
  if marketOpen = 0 [ if marketDayCount > (max [ patience ]
                                          of PJAgents) + 40 [
  set marketOpen 1
  ]
  ]
end

to update-turtlePrices
  stock-market
  if marketOpen = 1  and  nthTime = 1 [ options-market ]
  if marketOpen = 1 and nthTime >= 2 [ options-NNmarket ]
end

to calculate-sigma
  ifelse breed = randomAgents [ set guessed-sigma random-float 1]
    [ if length retList > patience + 2
    [ set guessed-sigma ((standard-deviation sublist retList
      (length retList - patience) (length retList))
      / (sqrt(dt)) judgement)]]
end

to-report calculate-d1 [ _sigma _spot _ttm]
  let _d1 ((ln (_spot / strike))
          + (noRisk + (_sigma * _sigma) / 2) * _ttm)
          / (_sigma * sqrt(_ttm))
  report _d1
end

to-report calculate-d2 [ _d1 _sigma _ttm ]
  let _d2 (_d1 - (_sigma * sqrt(_ttm)))
  report _d2
end

to-report calculate-callPrice [ _sigma _spot _ttm]
  let _d1 calculate-d1 _sigma _spot _ttm
  let _d2 calculate-d2 _d1 _sigma _ttm
  let _callPrice _spot * normcdf(_d1)
```

```
       - strike * exp( - noRisk * _ttm) * normcdf(_d2)
  report _callPrice
end

to-report calculate-putPrice [ _sigma _spot _ttm]
  let _d1 calculate-d1 _sigma _spot _ttm
  let _d2 calculate-d2 _d1 _sigma _ttm
  let _putPrice strike * exp( - noRisk * _ttm) * normcdf(- _d2)
      - _spot * normcdf(- _d1)
  report _putPrice
end

to-report normcdf [lclx]
  let lclt lclx
  let y 0.5 * erfcc ((-1) * lclt / ( 1 * sqrt 2.0))
  if ( y > 1.0 ) [ set y 1.0 ]
  report y
end

to-report normpdf [lclx]
  report 1 / ( sqrt (2 * pi)) * exp ( - lclx * lclx / 2.0)
end

to-report erfcc [lclx]
  let z abs lclx
  let lclt 1.0 / (1.0 + 0.5 * z)
  let r lclt *  exp ( - z * z - 1.26551223 + lclt * (1.00002368 + lclt
  * (0.37409196 + lclt * (0.09678418 + lclt * (-0.18628806 + lclt
  * (.27886807 + lclt * (-1.13520398 + lclt * (1.48851587 + lclt
  * (-0.82215223 + lclt * .17087277 ))))))))
  ifelse (lclx >= 0) [ report r ] [report 2.0 - r]
end

to stock-market
  set SpriceList lput SexePrice SpriceList
  if SexePrice  <= 0 [
   set Snews Snews2
   stop
   ]

  ask SrandomAgents
  [let threshold 0.5
   if floorActing and SexePrice < 500 [set threshold 0.8]
```

```
 ifelse random-float 1 < passLevel [set Spass True][set Spass False]
 ifelse not Spass
[ifelse random-float 1 < threshold [set Sbuy  True set Ssell False]
[set Ssell True set Sbuy  False] ]
[set Sbuy False set Ssell False]

if Spass          [set color gray]
if Sbuy           [set color green]
if Ssell          [set color red]
 ]

if  length SpriceList >= howManyTicks + 1
[let len length SpriceList
 let i len - 1
 set StrendUp   True
 set StrendDown True
 while [i >= len - howManyTicks]
  [if item (i - 1) SpriceList > item i SpriceList [set StrendUp False]
   if item (i - 1) SpriceList < item i SpriceList [set StrendDown False]
  set i i - 1]
  if StrendUp   [set SnumTrendUp   SnumTrendUp   + 1 ]
  if StrendDown [set SnumTrendDown SnumTrendDown + 1 ]
  ]

if length SpriceList >= 2 [
 let a (length SpriceList) - 1
 let b (length SpriceList) - 2
 let now item a SpriceList
 let bef item b SpriceList
 set retVal (now - bef) / now
 set retList lput retVal retList ]

ask StrendAgents
[ set Spass True set Sbuy False set Ssell False set color gray
  if StrendUp   [set Spass False set Sbuy  True set color green]
  if StrendDown [set Spass False set Ssell True set color red]
 ]

ask StradingAgents [ set Sprice SexePrice + (random-normal 0 100) ]

ask StradingAgents [
 set SexePrice-1 SexePrice
 if not Spass
```

```
  [
   let tmp[]
   set tmp lput Sprice tmp
   set tmp lput who tmp

   if Sbuy [set SlogB lput tmp SlogB]

   set SlogB reverse sort-by [item 0 ?1 < item 0 ?2] SlogB
   if (not empty? SlogB and not empty? SlogS) and
   item 0 (item 0 SlogB) >= item 0 (item 0 SlogS)
   [set SexePrice item 0 (item 0 SlogS)
    let agB item 1 (item 0 SlogB)
    let agS item 1 (item 0 SlogS)
    ask turtle agB [set Sstocks Sstocks + 1
                    set Scash Scash - SexePrice]
    ask turtle agS [set Sstocks Sstocks - 1
                    set Scash Scash + SexePrice]
    set SlogB but-first SlogB
    set SlogS but-first SlogS
    ]

   if Ssell [set SlogS lput tmp SlogS]

   set SlogS sort-by [item 0 ?1 < item 0 ?2] SlogS
   if (not empty? SlogB and not empty? SlogS) and
   item 0 (item 0 SlogB) >= item 0 (item 0 SlogS)
   [set SexePrice item 0 (item 0 SlogB)
    let agB item 1 (item 0 SlogB)
    let agS item 1 (item 0 SlogS)
    ask turtle agB [set Sstocks Sstocks + 1
                    set Scash Scash - SexePrice]
    ask turtle agS [set Sstocks Sstocks - 1
    set Scash Scash + SexePrice]
    set SlogB but-first SlogB
    set SlogS but-first SlogS
    ]
  ]
 ]

if cleanTheLogs [
 set SlogB [ ]
 set SlogS [ ]
 ]
```

```
end

to options-market
  ask tradingAgents [let f random 10
   ifelse f > 3 [ set agree True set notagree False ]
                [ set notagree True set agree False ]]

  ask tradingAgents [
   if StrendUp [ if agree [ if agree [ set SexePriceEND SexePrice
                                       * (1 + (ttm / dt) * ( 0.002))]
                 if notagree [set SexePriceEND SexePrice -  (ttm / dt) *
                                    (random-normal 0 50)] ]
   if StrendDown [ if agree if agree [ set SexePriceEND SexePrice
                                       * (1 - (ttm / dt) * ( 0.002))]
                   if notagree [set SexePriceEND SexePrice + (ttm / dt) *
                                     ( random-normal 0 50)]]]

  ask tradingAgents [ let s random 10
   ifelse s > 5 [ set callmarket True set putmarket False ]
                [ set putmarket True set callmarket False ]]

  ask tradingAgents [
   if callmarket   [  calculate-sigma
    set CgsigList lput guessed-sigma CgsigList
    set CttmList lput ttm CttmList
    set CspotList lput SexePrice CspotList

    ifelse ( CexePrice + 50 ) < ( SexePriceEND - strike )
            [ set buyer True set seller False ]
            [ set seller True set buyer False ]

    if buyer [ set price CexePrice + (random-normal 2 2)
               set color green ]
    if seller [ set price CexePrice + (random-normal 2 2)
               set color red ]
    set CoutputList lput price CoutputList ]

   if putmarket  [ calculate-sigma
    set PgsigList lput guessed-sigma PgsigList
    set PttmList lput ttm PttmList
    set PspotList lput SexePrice PspotList

    ifelse ( PexePrice + 50 ) < (strike - SexePriceEND)
```

```
        [ set buyer True set seller False]
        [ set seller True set buyer False]

 if buyer [ set price PexePrice + (random-normal 2 2)
            set color green ]
 if seller [ set price PexePrice + (random-normal 2 2)
            set color red ]
 set PoutputList lput price PoutputList ]
 ]

ask tradingAgents [
 if callmarket [
  set CexePrice-1 CexePrice
  let tmp[]
  set tmp lput price tmp
  set tmp lput who tmp

  if buyer [ set ClogB lput tmp ClogB ]

  set ClogB reverse sort-by [item 0 ?1 < item 0 ?2] ClogB
  if (not empty? ClogB and not empty? ClogS)
     and item 0 (item 0 ClogB) >= item 0 (item 0 ClogS)
  [ set CexePrice item 0 (item 0 ClogS)
    let agB item 1 (item 0 ClogB)
    let agS item 1 (item 0 ClogS)
    ask turtle agB [set calls calls + 1
                    set money money - CexePrice
                    set CtotalBought CtotalBought + 1 ]
   ask turtle agS [set calls calls - 1
                   set money money + CexePrice
                   set CtotalSold CtotalSold + 1 ]
   set ClogB but-first ClogB
   set ClogS but-first ClogS
  ]

 if seller [ set ClogS lput tmp ClogS]

 set ClogS sort-by [item 0 ?1 < item 0 ?2] ClogS
 if (not empty? ClogB and not empty? ClogS)
    and item 0 (item 0 ClogB) >= item 0 (item 0 ClogS)
 [ set CexePrice item 0 (item 0 ClogB)
   let agB item 1 (item 0 ClogB)
   let agS item 1 (item 0 ClogS)
```

```
       ask turtle agB [set calls calls + 1
                       set money money - CexePrice
                       set CtotalBought CtotalBought + 1 ]
       ask turtle agS [set calls calls - 1
                       set money money + CexePrice
                       set CtotalSold CtotalSold + 1 ]
       set ClogB but-first ClogB
       set ClogS but-first ClogS
      ]
set exePrice CexePrice
]


     if putmarket [
      set PexePrice-1 PexePrice
      let tmp[]
      set tmp lput price tmp
      set tmp lput who tmp

      if buyer [set PlogB lput tmp PlogB]

      set PlogB reverse sort-by [item 0 ?1 < item 0 ?2] PlogB
      if (not empty? PlogB and not empty? PlogS)
         and item 0 (item 0 PlogB) >= item 0 (item 0 PlogS)
      [set PexePrice item 0 (item 0 PlogS)
       let agB item 1 (item 0 PlogB)
       let agS item 1 (item 0 PlogS)
       ask turtle agB [set puts puts + 1
                       set money money - PexePrice
                       set PtotalBought PtotalBought + 1 ]
      ask turtle agS [set puts puts - 1
      set money money + PexePrice
      set PtotalSold PtotalSold + 1 ]
      set PlogB but-first PlogB
      set PlogS but-first PlogS
      ]

      if seller [set PlogS lput tmp PlogS]

      set PlogS sort-by [item 0 ?1 < item 0 ?2] PlogS
      if (not empty? PlogB and not empty? PlogS)
      and item 0 (item 0 PlogB) >= item 0 (item 0 PlogS)
      [set PexePrice item 0 (item 0 PlogB)
```

```
      let agB item 1 (item 0 PlogB)
      let agS item 1 (item 0 PlogS)
      ask turtle agB [set puts puts + 1
                      set money money - PexePrice
                      set PtotalBought PtotalBought + 1 ]
      ask turtle agS [set puts puts - 1
                      set money money + PexePrice
                      set PtotalSold PtotalSold + 1 ]
      set PlogB but-first PlogB
      set PlogS but-first PlogS
       ]
   set exePrice PexePrice
   ]
  ]

 set ClogB []
 set ClogS []
 set PlogB []
 set PlogS []
end

to options-NNmarket
  ask tradingAgents [let f random 10
   ifelse f > 3 [ set agree True set notagree False]
                [set notagree True set agree False ]]

  ask tradingAgents [
   if StrendUp [ if agree [ if agree [ set SexePriceEND SexePrice
                                          * (1 + (ttm / dt) * ( 0.002))]
                 if notagree [set SexePriceEND SexePrice -  (ttm / dt) *
                                  (random-normal 0 50)] ]

   if StrendDown [ if agree [if agree [ set SexePriceEND SexePrice
                                           * (1 + (ttm / dt) * ( 0.002))]
                   if notagree [set SexePriceEND SexePrice + (ttm / dt) *
                                    ( random-normal 0 50)] ] ]

  ask tradingAgents [ let s random 10
   ifelse s > 5 [set callmarket True set putmarket False]
                [set putmarket True set callmarket False]
   ]

  ask tradingAgents [ calculate-sigma ]
```

```
ask tradingAgents [
  if callmarket   [set price Cpredict SexePrice guessed-sigma ttm
   ifelse ( CexePrice + 50 ) < ( SexePriceEND - strike )
           [set buyer True set seller False]
           [set seller True set buyer False]
   if buyer [ set color green ]
   if seller [ set color red ] ]

  if putmarket  [ set price Ppredict SexePrice guessed-sigma ttm
   ifelse ( PexePrice + 50 ) < (strike - SexePriceEND)
          [set buyer True set seller False]
          [set seller True set buyer False]
   if buyer [ set color green ]
   if seller [ set color red ] ]
   ]

ask tradingAgents [
if callmarket [
 set CexePrice-1 CexePrice
 let tmp[]
 set tmp lput price tmp
 set tmp lput who tmp

 if buyer [set ClogB lput tmp ClogB]

 set ClogB reverse sort-by [item 0 ?1 < item 0 ?2] ClogB
 if (not empty? ClogB and not empty? ClogS)
    and item 0 (item 0 ClogB) >= item 0 (item 0 ClogS)
 [set CexePrice item 0 (item 0 ClogS)
  let agB item 1 (item 0 ClogB)
  let agS item 1 (item 0 ClogS)
  ask turtle agB [set calls calls + 1
                  set money money - CexePrice
                  set CtotalBought CtotalBought + 1]
  ask turtle agS [set calls calls - 1
                  set money money + CexePrice
                  set CtotalSold CtotalSold + 1]
  set ClogB but-first ClogB
  set ClogS but-first ClogS
  ]
```

```
  if seller [set ClogS lput tmp ClogS]

  set ClogS sort-by [item 0 ?1 < item 0 ?2] ClogS
  if (not empty? ClogB and not empty? ClogS)
     and item 0 (item 0 ClogB) >= item 0 (item 0 ClogS)
  [set CexePrice item 0 (item 0 ClogB)
   let agB item 1 (item 0 ClogB)
   let agS item 1 (item 0 ClogS)
   ask turtle agB [set calls calls + 1
   set money money - CexePrice
   set CtotalBought CtotalBought + 1]
   ask turtle agS [set calls calls - 1
   set money money + CexePrice
   set CtotalSold CtotalSold + 1]
   set ClogB but-first ClogB
   set ClogS but-first ClogS
   ]
 set exePrice CexePrice
]

 if putmarket [
  set PexePrice-1 PexePrice
  let tmp[]
  set tmp lput price tmp
  set tmp lput who tmp

  if buyer [set PlogB lput tmp PlogB]

  set PlogB reverse sort-by [item 0 ?1 < item 0 ?2] PlogB
  if (not empty? PlogB and not empty? PlogS)
  and item 0 (item 0 PlogB) >= item 0 (item 0 PlogS)
  [set PexePrice item 0 (item 0 PlogS)
   let agB item 1 (item 0 PlogB)
   let agS item 1 (item 0 PlogS)
   ask turtle agB [set puts puts + 1
                   set money money - PexePrice
                   set PtotalBought PtotalBought + 1]
   ask turtle agS [set puts puts - 1
                   set money money + PexePrice
                   set PtotalSold PtotalSold + 1]
   set PlogB but-first PlogB
   set PlogS but-first PlogS
   ]
```

```
    if seller [set PlogS lput tmp PlogS]

    set PlogS sort-by [item 0 ?1 < item 0 ?2] PlogS
    if (not empty? PlogB and not empty? PlogS)
       and item 0 (item 0 PlogB) >= item 0 (item 0 PlogS)
    [set PexePrice item 0 (item 0 PlogB)
     let agB item 1 (item 0 PlogB)
     let agS item 1 (item 0 PlogS)
     ask turtle agB [set puts puts + 1
                     set money money - PexePrice
                     set PtotalBought PtotalBought + 1]
     ask turtle agS [set puts puts - 1
                     set money money + PexePrice
                     set PtotalSold PtotalSold + 1]
     set PlogB but-first PlogB
     set PlogS but-first PlogS
     ]
    set exePrice PexePrice
   ]
  ]
end

to make_datatrain
  rserve:put "CspotList" CspotList
  rserve:put "CgsigList" CgsigList
  rserve:put "CttmList" CttmList
  rserve:put "CoutputList" CoutputList
  rserve:eval "CS<-as.numeric(CspotList)"
  rserve:eval "Csigma<-as.numeric(CgsigList)"
  rserve:eval "Cttm<-as.numeric(CttmList)"
  rserve:eval "COptionPrice<-as.numeric(CoutputList)"
  rserve:eval "f<-length(CS)"
  rserve:eval "a<-sample(2:(f-10), 1)"
  rserve:eval "CS<-CS[a:f]"
  rserve:eval "Csigma<-Csigma[a:f]"
  rserve:eval "Cttm<-Cttm[a:f]"
  rserve:eval "COptionPrice<-COptionPrice[a:f]"

  rserve:eval "Cdatatrain<-cbind(CS, Csigma, Cttm, COptionPrice)"
  rserve:eval "colnames(Cdatatrain)<-c('CS','Csigma','Cttm','COptionPrice')"
  rserve:eval "Cmaxsd<-apply(Cdatatrain, 2, max)"
  rserve:eval "Cminsd<-apply(Cdatatrain, 2, min)"
```

```
  rserve:eval "Cscaledtrain<-as.data.frame (scale(Cdatatrain,
                            center = Cminsd , scale = Cmaxsd - Cminsd))"

  set Cmin1 rserve:get "Cminsd[1]"
  set Cmax1 rserve:get "Cmaxsd[1]"
  set Cmin2 rserve:get "Cminsd[2]"
  set Cmax2 rserve:get "Cmaxsd[2]"
  set Cmin3 rserve:get "Cminsd[3]"
  set Cmax3 rserve:get "Cmaxsd[3]"
  set Cmin4 rserve:get "Cminsd[4]"
  set Cmax4 rserve:get "Cmaxsd[4]"

  rserve:put "PspotList" PspotList
  rserve:put "PgsigList" PgsigList
  rserve:put "PttmList" PttmList
  rserve:put "PoutputList" PoutputList
  rserve:eval "PS<-as.numeric(PspotList)"
  rserve:eval "Psigma<-as.numeric(PgsigList)"
  rserve:eval "Pttm<-as.numeric(PttmList)"
  rserve:eval "POptionPrice<-as.numeric(PoutputList)"
  rserve:eval "e<-length(PS)"
  rserve:eval "b<-sample(2:(e-10), 1)"
  rserve:eval "PS<-PS[b:e]"
  rserve:eval "Psigma<-Psigma[b:e]"
  rserve:eval "Pttm<-Pttm[b:e]"
  rserve:eval "POptionPrice<-POptionPrice[b:e]"

  rserve:eval "Pdatatrain<-cbind(PS, Psigma, Pttm, POptionPrice)"
  rserve:eval "colnames(Pdatatrain)<-c('PS','Psigma','Pttm','POptionPrice')"
  rserve:eval "Pmaxsd<-apply(Pdatatrain, 2, max)"
  rserve:eval "Pminsd<-apply(Pdatatrain, 2, min)"
  rserve:eval "Pscaledtrain<-as.data.frame (scale(Pdatatrain,
                            center = Pminsd , scale = Pmaxsd - Pminsd))"

  set Pmin1 rserve:get "Pminsd[1]"
  set Pmax1 rserve:get "Pmaxsd[1]"
  set Pmin2 rserve:get "Pminsd[2]"
  set Pmax2 rserve:get "Pmaxsd[2]"
  set Pmin3 rserve:get "Pminsd[3]"
  set Pmax3 rserve:get "Pmaxsd[3]"
  set Pmin4 rserve:get "Pminsd[4]"
  set Pmax4 rserve:get "Pmaxsd[4]"
end
```

```
to train
  rserve:eval "Cnn<-neuralnet ( COptionPrice~CS+Csigma+Cttm,
               Cscaledtrain, hidden = 3, threshold = 0.03, rep=10)"
  rserve:eval "Crn<-which.min(Cnn$result.matrix[1,])"
  rserve:eval "plot(Cnn, rep = Crn)"
  rserve:put  "CmyNum" who
  rserve:eval "Cname<-paste('Cnn', CmyNum, sep='')"
  rserve:eval "assign(Cname, Cnn)"
  set Crn rserve:get "Crn"

  rserve:eval "Pnn<-neuralnet ( POptionPrice~PS+Psigma+Pttm,
               Pscaledtrain, hidden = 3, threshold = 0.03, rep=10)"
  rserve:eval "Prn<-which.min(Pnn$result.matrix[1,])"
  rserve:eval "plot(Pnn, rep = Prn)"
  rserve:put  "PmyNum" who
  rserve:eval "Pname<-paste('Pnn', PmyNum, sep='')"
  rserve:eval "assign(Pname, Pnn)"
  set Prn rserve:get "Prn"
end

to-report Cpredict [-SexePrice -sigma -ttm ]
  let sspot (( -SexePrice - Cmin1 ) / (Cmax1 - Cmin1 ))
  let ssigma (( -sigma - Cmin2 ) / ( Cmax2 - Cmin2 ))
  let sttm (( -ttm - Cmin3 ) / ( Cmax3 - Cmin3 ))

  rserve:put "Crn" Crn
  rserve:put "CmyNum" who
  rserve:eval "CmyNn<-paste('Cnn', CmyNum, sep='')"
  rserve:eval "Cnn <- get(CmyNn)"
  let test rserve:get "CmyNn"
  rserve:eval "Ctt<-names(Cnn)[13]=='result.matrix'"

  (rserve:putdataframe "Cdata" "CS" sspot  "Csigma" ssigma "Cttm" sttm)
  rserve:eval "Cnprice<-compute(Cnn, Cdata, rep = Crn)$net.result[1,1]"
  set Cnprice rserve:get "Cnprice"
  let -nnprice  Cnprice * ( Cmax4 - Cmin4 ) + Cmin4
  report -nnprice
end

to-report Ppredict [-SexePrice -sigma -ttm ]
  let sspot (( -SexePrice - Pmin1 ) / (Pmax1 - Pmin1 ))
  let ssigma (( -sigma - Pmin2 ) / ( Pmax2 - Pmin2 ))
```

```
  let sttm (( -ttm - Pmin3 ) / ( Pmax3 - Pmin3 ))

  rserve:put "Prn" Prn
  rserve:put "PmyNum" who
  rserve:eval "PmyNn<-paste('Pnn', PmyNum, sep='')"
  rserve:eval "Pnn <- get(PmyNn)"
  let test rserve:get "PmyNn"
  rserve:eval "Ptt<-names(Pnn)[13]=='result.matrix'"

  (rserve:putdataframe "Pdata" "PS" sspot  "Psigma" ssigma "Pttm" sttm)
  rserve:eval "Pnprice<-compute(Pnn, Pdata, rep = Prn)$net.result[1,1]"
  set Pnprice rserve:get "Pnprice"
  let -nnprice  Pnprice * ( Pmax4 - Pmin4 ) + Pmin4
  report -nnprice
end
```

Bibliography

[1] W. D. Elliott *Analyzing Options Market Toxicity and the Black-Scholes Formula in the Presence of Jump Diffusion as Simulated with Agent-Based Modeling.* Undergraduate Economic Review: Vol. 11: Iss. 1, Article 14 (2015)

[2] J. C. Hull. *Options, Futures and Other Derivatives* Prentice Hall, Inc., 6th ed. (2006)

[3] J. Cvitaníc and F. Zapatero. *Introduction to the Economics and Mathematics of Financial Markets.* The MIT Press (2004)

[4] P. Samuelson, *Rational Theory of Warrant Pricing*, Industrial Management Rev. 6 (1965) p.13-39

[5] P. Samuelson, R. C. Merton, *A Complete Model of Warrant Pricing that Maximizes Utility*, Industrial Management Review (1969) p.17-46

[6] E. O. Thorp, S. T. Kassouf, *Beat the Market*, New York, Random House (1967)

[7] F. Black, M. Scholes. *The Pricing of Options and Corporate.* The Journal of Political Economy, Vol. 81, No. 3 (May - Jun., 1973), pp. 637-654. The University of Chicago Press (1973)

[8] D. Teneng, *Limitations of the Black-Scholes Model*, 68 Int?l.R.J.F.E. 99-102 (2011)

[9] O. H. Yalincak, *Criticism of the Black-Scholes Model: But Why Is It Still Used? (The Answer Is Simpler than the Formula)* New York University (2005)

[10] L. Gustafsson, M. Sternad, *Consistent micro, macro and state-based population modelling*, Mathematical Biosciences. 225 (2): 94?107 (2010)

[11] D.J. Marceau, *What can be learned from multi-agent systems?* In: Gimblett, R. (Ed.), Monitoring, Simulation and Management of Visitor Landscapes. University of Arizona Press, pp. 411?424 (2008)

[12] R. Axelrod, *The Complexity of Cooperation: Agent-Based Models of Competition and Collaboration*, Princeton University Press (1997)

[13] J. Ferber, *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*, Addison-Wesley, Reading, MA (1999)

[14] C. M. Macal, M. J. North, *Tutorial on agent-based modelling and simulation*, Journal of Simulation, 4(3), 151?162 (2010)

[15] N. R. Jennings, M. J. Wooldridge, *Agent Technology: Foundations, Applications, and Markets*, Springer Science & Business Media (Dec 2012)

[16] S. Bandini, S. Manzoni, G. Vizzari, *Agent Based Modeling and Simulation: An Informatics Perspective*, Journal of Artificial Societies and Social Simulation 12 (4) 4 <http://jasss.soc.surrey.ac.uk/12/4/4.html> (2009)

[17] R. Axtell, *Why Agents? On the Varied Motivation for Agent-Computing in the Social Sciences*, Brookings Institution CSED Technical Report No. 17, (Nov 2000)

[18] R. Boero, M. Morini, M. Sonnessa,P. Terna, *Agent-based Models of the Economy. From Theories to Applications* Palgrave Macmillan (2015)

[19] H. Van Dyke Parunak, R. Savit, R. L. Riolo, *Agent-Based Modeling vs. Equation-Based Modeling: A Case Study and Users Guide*, Proceedings of Multi-agent systems and Agent-based Simulation (MABS 98), 10-25, Springer, LNAI 1534 (1998)

[20] N. Johnson, *Due è facile, tre è complessità. Dal caos agli investimenti in Borsa*, Dedalo (2009)

[21] E. Bonabeau, *Agent-based modeling: Methods and techniques for simulating human systems*, Proceedings of the National Academy of Sciences of the United States of America Vol. 99, No. 10, Supplement 3, pp. 7280?7287 (May 2012)

[22] J. M. Epstein, R. L. Axtell, *Growing Artificial Societies: Social Science from the Bottom Up*, MIT Press, Cambridge, MA (1996)

[23] W. Weaver, *Science and complexity*, American Scientist, 36: 536-544 (1948)

[24] P. W. Anderson, *More Is Different*, Science , 177(4047), pp. 393?396 (1972)

[25] P. Cillieris, *Boundaries, Hierarchies and Networks in Complex Systems*, International Journal of Innovation Management, Vol. 5, No. 2 pp. 135–147 © Imperial College Press, (June 2001)

[26] J. Ladyman, J. Lambert, K. Wiesner, *What is a Complex System?*, `http://philsci-archive.pitt.edu/9044/4/LLWultimate.pdf`, (March 2012) (URL consulted on Aug 24, 2016)

[27] J-C. Trichet, *Reflections on the Nature of Monetary Policy Non-standard Measures and Finance Theory*, (2010) `http://www.ecb.europa.eu/press/key/date/2010/html/sp101118.en.html` (URL consulted on Aug 24, 2016)

[28] G. J. Stigler, *Public regulation of the securities market*, The Journal of Business (1964)

[29] K. J. Cohen, S. F. Maier, R. A. Schwartz,D. K. Whitcomb, *The Microstructure of Securities Markets*, Englewood Cliffs, NJ: Prentice-Hall (1986)

[30] G. W. Kim, H. M. Markowitz, *Investment rules, margin and market volatility*, J. Portfolio Manag. 16 45?52 (1989)

[31] E. Samanidou, E. Zschischang, D. Stauffer, T. Lux, *Agent–based models of financial markets*, IOP Publishing Reports on Progress in Physics 70 409?450 doi:10.1088/0034-4885/70/3/R03 (2007)

[32] R. A. Meyers, *Complex Systems in Finance and Econometrics* Volume 1, Springer Science & Business (Nov 2010)

[33] W. Arthur, J. H. Holland, B. Lebaron, R. Palmer, P. Tyler, *Asset pricing under endogenous expectation in an artificial stock market*, retrieved from `http://www2.econ.iastate.edu/tesfatsi/ahlpt96.pdf`

[34] N. Ehrentreich, *Agent-Based Modeling: The Santa Fe Institute Artificial Stock Market Model Revisited*, Springer Science & Business Media (Oct 2007)

[35] T. Lux, M. Marchesi, *Scaling and Criticality in a Stochastic Multi-Agent Model of a Financial Market*, Nature 397, pp. 498 - 500 (1999)

[36] M. Tran, D. Pham–Hi, T. Duong, *Agent-Based Modeling in Option Pricing under Unknown Volatility and Liquidity risk*, Proceedings of the Second Asia-Pacific Conference on Global Business, Economics, Finance and Social Sciences (AP15Vietnam Conference) ISBN: 978-1-63415-833-6 Danang-Vietnam, Paper ID: VL573 (July 2015)

[37] A. K. Jain,J. Mao, K. Mohiuddin, *Artificial Neural Network: A Tutorial*, IEEE Computer Special Issue on Neural Computing (1996)

[38] W.S. McCulloch, W. Pitts, *A logical calculus of the ideas immanent in nervous activity*, Bulletin of Mathematical Biophysics, vol. 5, pp. 115-133, (1943)

[39] T. H. Abraham, *(Physio)logical Circuits: the Intellectual Origins of the McCullogh & Pitts Neural Networks*, Journal of the History of the Behavioral Sciences, Vol. 38(1), 3–25 Winter 2002

[40] G. Dreyfus, *Neural Networks Methodology and Applications*, Springer Berlin Heidelberg New York (2004)

[41] F. Rosenblatt, *The perceptron: A probabilistic model for information storage and organization in the brain.*,Psychological Review, Vol 65(6), Nov 1958, 386-408 (1958)

[42] C. Gulcehre, http://deeplearning.net/tutorial/lenet.html (URL consulted on Aug 27, 2016)

[43] I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning*, Book in preparation for MIT Press, http://www.deeplearningbook.org (2016)

[44] `http://ufldl.stanford.edu/tutorial/supervised/ConvolutionalNeuralNetwork/` (URL consulted on Aug 27, 2016)

[45] M. Mourafiq, http://mourafiq.com/2016/08/10/playing-with-convolutions-in-tensorflow.html (URL consulted on Aug 27, 2016)

[46] T. Marwala, *Economic Modeling Using Artificial Intelligence Methods*, Springer London Heidelberg New York Dordrecht (2013)

[47] R. Sharda, *Neural networks for the MS/OR analyst: An application bibliography*, Interfaces 24 (2), 116?130 (1994)

[48] Y. Fang, M. A. Cohen, T. G. Kincaid, *Dynamics of a Winner–Takes–All Neural Network*, Neural Networks Vol. 9, No. 7, pp. 1141–1154 (1996)

[49] R. Adachi, A. Takemura *Sequential optimizing investing strategy with neural networks*, Expert Systems with Applications 38 12991-12998 (2011)

[50] J.Yao, Y. Li, C. L. Tan, *Option price forecasting using neural networks*, Omega, Elsevier, vol. 28(4), pages 455-466, (August 2000)

[51] H. Amilon, *A neural network versus Black-Scholes: a comparison of pricing and hedging performances*, Journal of Forecasting 22, 317-335 (2003)

[52] J. M. Hutchinson, A. W. Lo, T. Poggio, *A Nonparametric Approach to Pricing and Hedging Derivative Securities Via Learning Networks*, The Journal of Finance, 49: 851–889 (1994)

[53] R. G. Ahangar, M. Yahyazadehfar, H. Pournaghshband, *The Comparison of Methods Artificial Neural Network with Linear Regression Using Specific Variables for Prediction Stock Price in Tehran Stock Exchange*, (IJCSIS) International Journal of Computer Science and Information Security, Vol. 7, No. 2 (February 2010)

[54] G. Zhang, B. E. Patuwo, M. Y. Hu, *Forecasting with artificial neural networks: The state of the art*, International Journal of Forecasting 14 35?62 (1998)

[55] `https://www.r-project.org/` (URL consulted on Jun 13, 2016)

[56] TIOBE Index for August 2016 `http://www.tiobe.com/tiobe-index//` (URL consulted on Jun 13, 2016)

[57] R. Ihaka, R. Gentleman, *R: a Language for Data Analysis and Graphics*, Journal of Computation and Graphical Statistics, Volume 5, Number 3 (1996)

[58] R. Ihaka, *R: Past and Future History*

[59] `http://www.johndcook.com/blog/r\_language\_for\_programmers/introduction` (URL consulted on Jun 13, 2016)

[60] `https://cran.r-project.org/web/packages/neuralnet/index.html` (URL consulted on Jun 13, 2016)

[61] `https://cran.r-project.org/web/packages/nnet/index.html`(URL consulted on Jun 13, 2016)

[62] `https://cran.r-project.org/web/packages/RSNNS/index.html` (URL consulted on Jun 13, 2016)

[63] C. Bergmeir, J. M. Benítez, *Neural Networks in R Using the Stuttgart Neural Network Simulator: RSNNS*, Journal of Statistical Software, Volume 46, Issue 7 (Jan 2012)

[64] `https://cran.r-project.org/web/packages/simpleNeural/index.html` (URL consulted on Jun 13, 2016)

[65] `https://cran.r-project.org/web/packages/AMORE/index.html` (URL consulted on Jun 13, 2016)

[66] `https://cran.r-project.org/web/packages/elmNN/index.html` (URL consulted on Jun 13, 2016)

[67] `https://cran.r-project.org/web/packages/deeplearning/index.html` (URL consulted on Jun 13, 2016)

[68] `https://cran.r-project.org/web/packages/rnn/index.html` (URL consulted on Jun 13, 2016)

[69] F. Günther, S. Fritsc *neuralnet: Training of Neural Networks* , The R Journal Vol. 2/1, (June 2010)

[70] S. Urbanek *Rserve A Fast Way to Provide R Functionality to Applications.* Proceeding the 3rd International Workshop on Distributed Statistical Computing (DSC 2003), March 20-22, Vienna, Austria (2003)

[71] C. J. Willmott, S. G. Ackleson, R. E. Davis, J. J. Feddema, K. M. Klink, D. R. Legates, J. O'Donnell, *C. M. Rowe, Statistics for the Evaluation and Comparison of Models*, J. Geophys. Res., 90(C5), 8995-9005 (1985)

Ringraziamenti

Un grazie sincero al mio relatore, il Professor Terna che in questi mesi mi ha ispirata, spronata e seguita con una disponibilità e attenzione che mai avrei pensato di ricevere. Grazie per aver creduto in me e per avermi dato un essenziale aiuto in questo lavoro di tesi.

Ringrazio tutti i Professori che durante questo biennio hanno messo a disposizione loro stessi e le proprie conoscenze per formarci come studenti e come persone. Un particolare ringraziamento al Professor Caselle che è stato un punto di riferimento fondamentale durante questi due anni, sempre disponibile a elargire utili consigli.

Grazie a mio padre per tutte le parole giuste e gli abbracci rassicuranti che mi hanno permesso di trovare quella sicurezza che mi mancava quando un esame mi sembrava un ostacolo insormontabile. I tuoi «Andrà tutto bene, ci riuscirai!» mi hanno salvata innumerevoli volte. Grazie!

Grazie a mia madre che mi ha reso una persona forte e ambiziosa, sempre pronta a fissare e raggiungere nuovi traguardi. Proprio come lei.

Grazie a mia nonna che non si è mai dimenticata di recitare una preghierina prima di ogni mio esame e di ricordarmi quanto per lei sono brava.

Grazie ai miei compagni e compagne di università, grazie per aver condiviso ansie e gioie, dubbi e conoscenze. Grazie in particolare ad Alessia e Anthony che da sempre non sono stati solo colleghi ma grandi amici. Grazie per aver reso questi lunghi e duri anni così piacevoli da vivere con voi.

Grazie a Giulia e Marco i miei amici di sempre, presenti in ogni occasione senza dovr mai chiedere nulla. L'aiuto e l'amicizia che mi avete dato quando non ero più capace di volermi bene, non potrei ricambiarlo neanche in una vita intera.

Grazie a Sara, la mia Amica. Grazie per come hai trasformato la mia vita, per le parole dure che mi hanno migliorata, per quelle dolci che mi hanno fatta sentire protetta e ancora per quelle dure che mi hanno spronata. Grazie per quanto bene tu mi faccia sentire quando ti ho al mio fianco e quanto mi manchi quando so di non poterti ridere

o piangere tra le braccia. Garzie per tutti i NOI, invece dei tu o io. Grazie per avermi fatto capire che non potrei mai essere indipendente da te.

Grazie a Ruggero, la mia roccia. Grazie per il tuo amore che è un affetto immenso, ma non è solo questo. Non avrei mai pensato di poter vivere così intensamente una persona, di potermi fidare a tal punto da affidargli la mia stessa vita, di stimarla così tanto da vederla come punto di riferimento. Mi hai cresciuta, mi hai migliorata, mi hai reso la vita più piena e bella. Grazie per credere in me e in noi, sempre anche quando a me sembra impossibile. Grazie per voler condividere ogni giorno il mio stesso obiettivo, ricordandomi sempre che insieme raggiungeremo ogni nostro traguardo.

Grazie soprattutto a mio nonno. Ciò che più mi auguro dalla vita è di riuscire a trovare la tua voglia di vivere. Sei stato una grande persona e un nonno meraviglioso, da quando mi portavi tutti i pomeriggi a prendere il gelato a quando ti sedevi sul mio letto per guardarmi studiare. Mai in silenzio volevi imparare anche tu. So che sei ancora il mio più grande sostenitore e spero con tutto il mio cuore che tu possa ad oggi essere fiero di me.