A Forex Trading System Using Evolutionary Reinforcement Learning

by

Yupu Song

A Research Project

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Financial Mathematics

by

May 2017

APPROVED:

Professor Marcel Y. Blais, Advisor

Professor Luca Capogna, Head of Department

Abstract

Building automated trading systems has long been one of the most cutting-edge and exciting fields in the financial industry. In this research project, we built a trading system based on machine learning methods. We used the Recurrent Reinforcement Learning (RRL) [1] algorithm as our fundamental algorithm, and by introducing Genetic Algorithms (GA) [2] in the optimization procedure, we tackled the problems of picking good initial values of parameters and dynamically updating the learning speed in the original RRL algorithm. We call this optimization algorithm the Evolutionary Recurrent Reinforcement Learning algorithm (ERRL), or the GA-RRL algorithm. ERRL allows us to find many local optimal solutions easier and faster than the original RRL algorithm. Finally, we implemented the GA-RRL system on EUR/USD at a 5-minute level, and the backtest performance showed that our GA-RRL system has potentially promising profitability. In future research we plan to introduce some risk control mechanism, implement the system on different markets and assets, and perform backtest at higher frequency level.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Machine Learning and Algorithmic Trading

Machine learning is one of the most exciting branches of modern science. Basically it is defined as the field of study that looks into computers' ability to learn without being explicitly programmed[3]. Machine learning is being applied in different facets of applications. For example, it has been used in pattern recognition, using which the system can segment out individual objects from photos and label them. Google has developed the famous AlphaGo agent which beated the world champion Lee Sedol. Many scientists believe that machine learning will impact our lives.

Algorithmic trading is likewise a relatively new and promising field in the financial industry. In algorithmic trading, the trading strategies are programmed and implemented by computers. In the past decades, predictive mathematical models, statistical arbitrage methods, and event-driven systems have been commonly used in algorithmic trading. For manual traders one of the most important issues is the tendency to be undisciplined. For instance, manual traders are likely to continue trading after suffering a big loss because the more money they have lost, the more they are desperate for getting them back. Unlike manual traders, algorithms which set explicit rules will faithfully implement them, thus avoiding the man-caused interruptions and improving trading performance.

To implement machine learning techniques in algorithmic trading is a pretty natural choice. In the financial market, the price of the assets changes over time very frequently. There are large uncertainties in the market, so even if there might exist a pattern for a short period of time, the pattern will tend to collapse in the long run. A human will find it difficult to go through loads of dynamic data and recognize different patterns to predict the future state. However, an appropriate machine learning algorithm, using considerable computing power, might be able to help find such dynamic patterns. These dynamic models will be changing over time, but in most cases we only have to update certain parameters, which can be done automatically. So once the machine learning model is built, the future maintenance requires only routine work. In the financial industry, many companies have been trying to use smart machine learning algorithms plus powerful calculating devices in trading. Not only hedge funds like Bridgewater, Two Sigma, but also investment banks like J.P. Morgan are using machine learning in trading[4]. They hire machine learning experts in academia to develop the most advanced trading systems, and upgrade their trading facilities to get the fastest calculating speed and the most stable environment.

1.2 Related Work

Traders and scholars have been striving to design automated trading systems for a long time. Generally the trading systems can be classified as trend following systems and mean reversion systems.

For trend following systems, the most famous technique is to use the moving average (MA) indicator, which is the average value of the previous prices during a certain period. However, simply using MA is not likely to generate a stable profitable system, so MA is often considered as an input in some sophisticated systems. Another famous strategy is momentum trading. The momentum here usually means the previous performance of the asset, or technically, the previous returns in a certain period. Jegadeesh and Titman's research [5] shows that buying past winners and selling past losers will give a larger return than the return for the market index.

Compared to trend following strategies, the mean reversion strategies are more difficult to construct, because it is much more difficult to predict an exact turning point. One common approach is to use standard deviations based on the belief that prices will finally fall into a stable interval. However, this fails to be true when the price does breakout the previous price range, and one famous example is the collapse of Long-Term Capital Management, which failed at the arbitrage trading of Royal Dutch Shell. One can also use regression methods in a short term, or use ARCH or GARCH to construct time series models. For pairs traders or market makers, a famous approach is to use the Ornstein-Uhlenbeck (OU) process, see S. Rampertshammer's research[6] for a framework, and T. Leung and X. Li's paper[7] discussing the optimal entry and exit price. Besides, professional traders also proposed some interesting findings using miscellaneous techniques, for example, see J. Ehlers's report[8] which builds a system consisting of the fancy super smoother and a two-pole high-pass filter, turning stochastic oscillators from a delayed indicator into a predictive indicator.

In recent decades, the design of automated trading systems became more sophisticated, and many statistical and machine learning methods have been tried on different markets. For example, G. Creamer and Y. Freund used the alternative decision tree (ADT), proposed by Y. Freund and L. Mason [9], with some well-known technical indicators and fundamental financial indicators to predict the excess returns. Due to the complicated market conditions, a profitable pattern may be highly non-linear, so naturally, people implemented Artificial Neural Networks (ANN) to approximate a complicated model. For ANNs the two major issues are features selection and parameters optimization: what features should be selected and how much should they weight repectively both have a significant influence on the performance of the trading system. The features (independent variables) can be the price momentum or other technical indicators, and the optimization process may use the back propagation method[10] and genetic algorithms. See K. Kim and I. Han's research[11] which constructs ANNs to predict the stock index and uses genetic algorithms for parameters optimization.

To achieve a good trade-off between a trend following system and a mean reversion system is another important issue. The market is always changing, and thus pure trend following or pure mean reversion strategies cannot always fit in with the market conditions. Based on the Fama-French model[12], R. Balvers and Y. Wu[13] have explored a linear combination model of momentum and mean reversion strategies in different national stock markets, and the hybrid strategy gives abnormal excessive returns. A. Serban[14] applied the similar combination method on the forex market which likewise created excessive higher returns and an even higher Sharpe ratio than those obtained by investing in the stock market index.

The Recurrent Reinforcement Learning (RRL) algorithm was first proposed by J. Moody and L. Wu[15], which constructed an ANN and used back-propagation to optimize the parameters. The original paper tested the algorithm in an artificial market and showed promising results. J. Moody and L. Wu then continued their research[1] which tested the algorithm in stock and US Treasury Bills markets, compared the performance between RRL trader and Q-learning trader, and analyzed the sensitivity of some parameters. C. Gold[16] continued to explore RRL and applied it to the foreign exchange market, in which he first used rolling window methods to train and test the algorithm. He also suggested the movement/spread ratio to measure the feasibility of implementing RRL on certain assets, and gave an empirical analysis for optimizing parameters. M. Dempster and V. Leemans's paper[17] constructed a more sophisticated and practical automated forex trading system based on RRL with multiple layers. They used 1-minute data rather than the half-hour data of the previous paper to test the performance. The system trades only between 9 a.m. and 5 p.m London time, and was incorporated with risk control mechanisms such as trailing stop loss, constant holding period, and optimizing self-defined utility functions in terms of risk. Besides, from the empirical research in [17], using extra inputs with 14 classical technical indicators seems not to be significantly beneficial. However, the influence of using extra technical indicators can be intricate. J. Zhang and D. Maringer[18] used genetic algorithms to improve the RRL system by introducing technical indicators that are not relevant to price returns, and the results showed that this outperformed the original RRL system. Since the influence of incorporating technical indicators are complicated to judge, here we will not consider any other technical indicators in our following trading system.

1.3 Forex Trading

1.3.1 Foreign Exchange Market

The foreign exchange (forex) market is the place where people trade currencies. Most of the forex trading are completed electronically through computer networks all over the world, and there is no centralized physical exchange such as the New York

Exchange for the stock market. Therefore, the forex market is an over-the-counter (OTC) market. The market participants are mostly large financial institutions such as banks and hedge funds, big corporations doing international trades such like General Electric (GE), and individuals as consumers, travellers and investors. Forex market is open for 24 hours every day in 5 days of a week, and it is the largest market in the world, with a daily volume of around \$5 trillion, and the world GDP of a year is about \$73 trillion[1]. The causes for forex rate change is complicated, which include the changes of interest rates, inflation and the demand and supply of the currency.

Forex can be traded in mainly three markets: the spot market, the forwards market and the futures market. We do not go further into the latter two markets here, since when people mention the forex market usually they refer to the spot market. In the forex market, currencies are traded in pairs. In the electronic spot forex market, no physical currencies exchange really happens. It can be regarded that we are only trading the number, the exchange rate of currency pairs, and our accounts are usually US dollar denominated. For example, the exchange rate of EUR/USD, the most widely traded pair in the world. It represents how many US dollars one euro can buy. If the EUR/USD rate is now 1.36, then 1 euro can buy 1.36 dollars. If you think that the US dollar value will increase, you would sell the EUR/USD pair, which means that you get US dollars by selling euros, and wait for the EUR/USD rate to decrease. When such price movement happens, you can buy the EUR/USD pair, which means that you buy euros by selling US dollars. Since the US dollar's value decreased, theoretically you can get more euros, thus make a profit. In practice your account is denominated in US dollars, and you will earn the difference between the rate multiplied by some constant number. The next subsection will show an example.

1.3.2 Forex Trading Process

As a retail trader, to start forex trading, one needs to choose a broker. Basically, there are two types of brokers: market makers (MMs) and electronic communications

[1]Statistics come from the Bloomberg Market Concepts tutorial.

networks (ECNs). MMs set the bid and ask price in their own system thus they actually "make" the market, and provide liquidity to the real market. ECNs, on the other hand, gather all the bid and ask quotes from many market participants and only serve as the connecting bridges. The bid and ask quotes on ECNs' systems are more volatile and closer to the real market conditions. Here we will use the transactions on an ECN broker's platform (IC Markets CTrader platform) in our transaction example. Forex trading is very different from stock trading. For instance, almost all traders in forex market use leverage to magnify their positions. Let us go through some essential concepts and learn the trading process by conducting a simulated transaction.

Transaction Example:

*At 10:13:49 EST, Feb 15, 2017, the bid price of EUR/USD is 1.05696 at which I would like to sell the EUR/USD pair. I used an account with a **leverage** of 50:1, and used market sell order to sell 0.2 **lots** of the pair. However, it ended up with an average entry price of 1.05693, and cost me $ 422.77 for the **margin**. I suffered a **slippage** of* $1.05693 - 1.05696 = -0.00003$*. At 10:35:26 EST, 15 Feb 2017, I closed my position using a market order, with a closing price of 1.05750, and the total commissions are 1.26. The net **pips** I gained is $-5.7, and the gross profit I made is $ 0.2 × (-5.7) = -11.40. After adding the commissions, my net profit is $-12.66. Transaction Done.*

Let's clarify some concepts in the example above:

- **Lot:** The lot is the standard amount of the forex contract, which usually equals 100,000 units of a certain currency. For example, if you buy one lot of EUR/USD at price 1.05693, you actually buy 100,000 euros at the cost of $1.05693 * 100,000 = 105,693$ US dollars. In forex trading, one may trade a smaller size such as 0.5 lots, 0.2 lots, but the minimum is 0.01 lots.

- **Leverage and Margin:** Non-leveraged forex trading requires a large amount of money. For example, even trading 0.01 lots of EUR/USD requires about 1,000 dollars. Besides, the fluctuation of forex is often small, generally within

2% percent a day. To trade with less money and magnify their profit, at least hopefully, people use leverage to trade. In the previous example, the leverage is 50:1, which means that for one lot worth 100,000 units of a certain currency, we can trade it with only $100,000/50 = 2000$ units. These 2000 units are required as the deposit to hold one lot's position, and is called margin. The leverage will magnify both profit and loss. For example, if the currency pair rate increases or decreases for 1% and you hold a long position, then for one lot you will win or lose $100,000 \times 1\% = \$1000$. However, for a 50:1 leveraged account we trade one lot EUR/USD with only \$2000, so we will actually win or lose $1000/2000 = 50\%$. In the previous example, we trade 0.2 lots worth $0.2 \times 1.05693 \times 100,000 = 211,138.6$ using a 50:1 leverage, so we need to put \$ $211,138/50 = 422.77$ for margin.

- **Slippage:** The currency pair rate changes very fast. If we intend to trade at a certain price and use a market order, usually the order will be filled at a different price. The difference between our intended price and the real entry price is called the slippage. The slippage can be either positive or negative. In the previous example, I suffered a slippage of -0.00003: I intended to build a short position at 1.05696 but in fact the order was filled at 1.05693. Using a market order, the slippage is one of the costs of building a position.

- **Pips:** A pip is usually 1/100th of a cent. For one lot one pip is worth 10 units of the currency. In our previous example, the EUR/USD pair decreased from 1.05693 to 1.05750, total $(1.05693 - 1.05750)/0.0001 = -5.7$ pips. Since we only trade 0.2 lots, the gross profit is $-5.7 * 10 * 0.2 = -11.4$ US dollars.

In live trading for long-term investors the commissions and slippage can be almost neglected. However, for short-term day traders the commissions and slippage are very influential on their profit and loss. In our project we will take into account the commissions and slippage and try to develop an algorithm that can be used in live trading.

2 Methodology Overview

2.1 The Trading System Architecture

Our trading system is designed as follows:

First, using training data, we implemented the evolutionary RRL algorithm for optimization. Based on RRL, we used GA to pick some potentially good initial values for the parameters and dynamically change the learning speed of the RRL algorithm during the evolution. Here we can run this procedure many times and decide to use the parameters with the best performance on the training set, or by observing the trading behaviors of the different systems.

Second, we will introduce some risk control mechanisms, for example, a constant holding time limit, trailing stop loss, volatility stop loss, hibernation, and hedging with mean-reversion systems, and here again we can use GA to optimize the related parameters. After this step our trading algorithms are completely optimized and finished.

Finally, using a rolling window technique, we will test our strategy on the whole dataset to evaluate the performance. We can refer to the statistics such as the maximum drawback, win rate, Sharpe ratio, and profit per trade to evaluate the trading performance. Here we also need to optimize the best rolling window length.

In this project, we have only implemented the first step, and we did some basic backtests on several small datasets. The problem here is that this trading system can be very complicated and requires optimizing many parameters, and it will be more intricate when it comes to high-frequency data and involves risk control mechanisms and rolling window method. However, we do not have the computation resources powerful enough to do many experiments in the limited time. Besides, in practice it requires some manual observation and refinement. For example, by observing the algorithm's trading behavior in historical transactions, we may find some overfitting examples, or some unreasonable trades that are not expected to happen but somehow did, and some dramatic losses in certain extreme conditions. We will continue doing the research of the problems mentioned above in the future.

2.2 Data Description

We used the EUR/USD currency pair price of 1 minute frequency from 2006.12.31 22:00:00 UTC[2], to 2017.02.03 21:59:00 UTC. The raw data is provided by a professional forex data provider who asked to remain anonymous, which consists of timestamps, 5-digit price combinations (open price, highest price, lowest price, close price) and volume for each 1-minute bar. The total number of data points is $3,793,452$. Here we only consider the close prices in our model.

Time (UTC)	Open	High	Low	Close	Volume
2017.02.03 21:55:00	1.07855	1.07864	1.07851	1.07859	81.8
2017.02.03 21:56:00	1.07864	1.0787	1.07856	1.07857	115.73
2017.02.03 21:57:00	1.07857	1.0787	1.07847	1.07855	53.28
2017.02.03 21:58:00	1.07851	1.0786	1.07851	1.07853	32.8
2017.02.03 21:59:00	1.07852	1.07859	1.07818	1.07818	30.71

Table 2.1: Sample Raw Data

3 The Recurrent Reinforcement Learning Algorithm

3.1 Algorithm Principles

The Recurrent Reinforcement Learning algorithm (RRL) is based on the Artificial Neural Network (ANN). Below are some definitions to better illustrate algorithm.

[2]Coordinated Universal Time, which can be considered equivalent to the Greenwich Mean Time (GMT), or the London time without Daylight Saving Time adjustment.

- r_t **(Returns):** The return r_t is defined as $r_t = p_t - p_{t-1}$, in which p_t is the price at time t. In our ANN, if we consider the previous M periods of returns, we will use $r_t, r_{t-1}, ..., r_{t-M}$ in our inputs.
- ν **(Dummy Variable):** The dummy variable ν acts as the integration of noises.
- F_t **(Position):** The position at time t, which can be $1, 0, -1$ respectively for long, neutral, and short positions.
- μ **(Lots):** The lots we are trading, equivalent to the shares traded in the stocks market.
- δ **(commissions)**: The commission fees for a single transaction per lot.

The ANN for RRL is shown as the following graph:

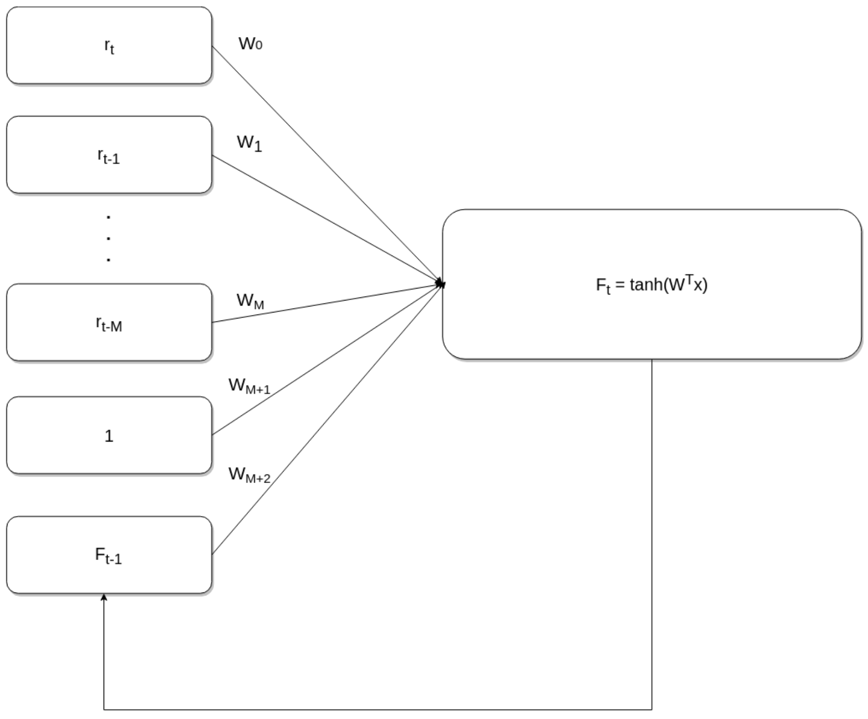

Figure 3.1: RRL Network

Here the input vector is $\vec{x} = (r_t, r_{t-M}, 1, F_{t-1})$, the weights are $\vec{w} = (w_0, ..., w_M, w_{M+1}, w_{M+2})^T$. Note that we not only use the previous returns, the dummy variable (default value 1, to be adjusted by the weight), but also the previous position F_{t-1}, and that is why this system is called a recurrent learning system. The total input is $w^T x$, which will be transformed by a hypertangent function so that the F_t value is within $[-1, 1]$. Subsequently, we can calculate the position return at each timestamp, defined as

$$R_t = \mu(F_{t-1}r_t - \delta(|F_t - F_{t-1}|)) \tag{1}$$

To optimize the algorithms, we need to optimize a reasonable utility function. Here we use the Sharpe ratio defined as

$$S_T = \frac{Ave(R_t)}{Std(R_t)} = \frac{E(X)}{\sqrt{E(X^2) - [E(X)]^2}} = \frac{\frac{1}{T}\sum_{t=1}^{T} R_t}{\sqrt{\frac{1}{T}\sum_{t=1}^{T} R_t^2 - (\frac{1}{T}\sum_{t=1}^{T} R_t)^2}} \tag{2}$$

To optimize the Sharpe ratio, we will use back propagation with a gradient descent method. We need to compute $\frac{dS_T}{dw}$, and the basic updating rule is

$$w^k = w^{k-1} + \rho \frac{dS_T}{dw}. \tag{3}$$

The superscript represents the number of training times, and ρ is the learning speed. To calculate $\frac{dS_T}{dw}$ we have two approaches. The first is accumulative learning, which calculates the total derivatives to implement back propagation. The second approach surrounds using the the on-line learning method to optimize a moving averaged Sharpe ratio as will be mentioned later. In accumulative training, w updates only after the whole training data is processed, while in online training, w updates at each timestamp.

3.1.1 Accumulative Learning

Define $A_T = E(R_t) = \frac{1}{T}\sum_{t=1}^{T} R_t$ and $B_T = E({R_t}^2) = \frac{1}{T}\sum_{t=1}^{T} R_t^2$, then (2) can be written as

$$S_T = \frac{A}{\sqrt{B - A^2}} \tag{4}$$

Therefore the total derivatives can be calculated as follows:

$$\frac{dS_T}{dw^k} = \sum_{t=1}^{T} (\frac{dS_T}{dA_T}\frac{dA_T}{dR_t} + \frac{dS_T}{dB_T}\frac{dB_T}{dR_t})(\frac{dR_t}{dF_t}\frac{dF_t}{dw_t^k} + \frac{dR_{t-1}}{dF_{t-1}}\frac{dF_{t-1}}{dw_{t-1}^k}) \tag{5}$$

in which

$$\frac{dS_T}{dA_T} = (B_T - A_T^2)^{-0.5} + A_T^2(B - A^2)^{-1.5} \tag{6}$$

$$\frac{dS_T}{dB_T} = -0.5A_T(B_T - A_T^2)^{-1.5} \tag{7}$$

$$\frac{dA_T}{dR_t} = \frac{1}{T} \tag{8}$$

$$\frac{dB_T}{dR_t} = \frac{2R_t}{T} \tag{9}$$

$$\frac{dR_t}{dF_t} = -\mu\delta sign(F_t - F_{t-1}) \tag{10}$$

$$\frac{dR_t}{dF_{t-1}} = \mu\delta sign(F_t - F_{t-1}) + \mu r_t \tag{11}$$

and recurrently updating $\frac{dF_t}{dw}$:

$$\frac{dF_t}{dw} = (1 - tanh^2(\vec{w}^T\vec{x}))(\frac{\partial F_t}{\partial w_t} + \frac{\partial F_t}{\partial F_{t-1}}\frac{\partial F_{t-1}}{\partial w_{t-1}}) \tag{12}$$

$$= (1 - tanh^2(\vec{w}^T\vec{x}))(\vec{x} + w_{M+2} \cdot \frac{\partial F_{t-1}}{\partial w_{t-1}}) \tag{13}$$

All equations (6) - (12) can be calculated at each timestamp, thus we can calculate (5) and update the weight $\vec{w}$ using (3).

3.1.2 Online learning

An online learning approach will update the parameters only based on the most recent returns. This can be achieved by using the *moving average Sharpe ratio (MA-sharpe)* as a moving average form of (2). We still define the Differential Sharpe Ratio in the same form of (4) with

$$A_t = A_{t-1} + \Delta A_t = A_{t-1} + \eta(R_t - A_{t-1}) \tag{14}$$

$$B_t = B_{t-1} + \Delta B_t = B_{t-1} + \eta(R_t^2 - B_{t-1}) \tag{15}$$

in which η is the coefficient of moving average. By Taylor's expansion we have for a small $\eta > 0$

$$S(\eta, t) = S(0, t) + \eta \frac{dS_t(0, t)}{d\eta} + O(\eta^2) \tag{16}$$

$$= S(0, t-1) + \eta \frac{dS_t(0, t)}{d\eta} + O(\eta^2) \tag{17}$$

in which S(η, t) represents the Sharpe ratio at time t with a moving average coefficient η defined as in (14) and (15). When $\eta = 0$, we write $S(0, t)$ as S_t in short, clearly $A_t = A_{t-1}$, $B_t = B_{t-1}$, so $S_t = S_{t-1}$. Here we define the differential Sharpe ratio (D-sharpe) as

$$D_t = \frac{dS_t}{d\eta} = \frac{B_{t-1}\Delta A_t - \frac{1}{2} \cdot A_{t-1}\Delta B_t}{(B_{t-1} - A_{t-1}^2)^{\frac{3}{2}}}. \tag{18}$$

In (16), regard S_{t-1} and η as constants, we have $dS_t \approx \eta dD_t$. Therefore we have $\frac{dS_t}{dR_t} \approx \eta \frac{dD_t}{dR_t}$, in which

$$\frac{dD_t}{dR_t} = \frac{B_{t-1} - A_{t-1}R_t}{(B_{t-1} - A_{t-1}^2)^{\frac{3}{2}}}. \tag{19}$$

Our online optimization updating rule is

$$w_t = w_{t-1} + \hat{\rho}\frac{dS_t}{dw} \tag{20}$$

in which

$$\frac{dS_t}{dw_t} \approx \rho \frac{dD_t}{dR_t}\left(\frac{dR_t}{dF_t}\frac{dF_t}{dw_t} + \frac{dR_{t-1}}{dF_{t-1}}\frac{dF_{t-1}}{dw_{t-1}}\right). \tag{21}$$

Here $\rho = \hat{\rho}\eta$, and the other differentiations and recurrent updating are the same as in the accumulative learning (10) - (12).

When we use online learning with rolling windows, if we always set initial A_0, B_0 to be some fixed small values at the beginning, according to (14), (15), and (19), the learning speed in the first several steps can be extremely high, thus the learning process may be seriously deviated. In practice, we will set the initial values of A_0, B_0 in the next training process as the final values of A_t and B_t in the previous training process.

3.2 Algorithm Disadvantages

One primary disadvantage, as noted in the previous work section, is the potential optimization issues faced by RRL algorithms; initially, the algorithm tends to remain at the local optimum when the learning rate is low, or otherwise stride over the local optimum and head to the worst case if the learning rate is too high. Second, the algorithm can be very sensitive to certain parameters such as the initial weights, learning rate, the number of returns used for training, the number of training times, the length of training and test window, and some parameters for risk controlling that will be touched on later. In some cases it will be nearly impossible to achieve a positive solution with a bad initial set of parameters in given time.

There exist many different local optima in a RRL training set, and to find the overall optimum will be difficult, time-consuming and may cause overfitting. Actually, sourcing the global optimum is not necessary, as the backtest results in Section 5 showed that overfitting may generate a system with good training performance which loses much money in the test dataset. Although theoretically we may use GA to find the global optimum, here our strategy surrounds finding a certan amount of local optimums, and then choose and utilize the best one based on particular criteria, which will be further elaborated on later. Besides, this primitive trading system does not include any risk control mechanisms of which we have to introduce, and the extra risk-related parameters also require optimization.

4 The Evolutionary RRL System

4.1 Genetic Algorithm (GA)

Genetic Algorithm is a commonly used algorithm for optimization, which was firstly proposed by J. H. Holland in 1975 in his famous book *Adaptation in Natural and Artificial Systems*[2]. The schema theorem[2] and empirical evidence indicate that GA is possible to find good solutions, but the convergence properties can be very complex. Using Markov chain theory, G. Rudolph[19] has proved that the canonical GA does not converge to the global optimum in static optimization problems, but some variants do. In this project we will use the elitist genetic algorithm (EGA), which converges to the global optimal solution with any choice of initial population[20].

By imitating the biological evolution process, GA chooses parents from the present population, and allows their genes to crossover and mutate to generate the individuals in the next generation, thus introducing stochastic variants. GA picks the parents based on the individual's fitness level in the environment; therefore, descendants with better fitness scores will have a better chance at survival and relaying their genes to the next generation. Compared to random search, GA improves the convergence rate, however, it still stands the risk of getting stuck in local optima.

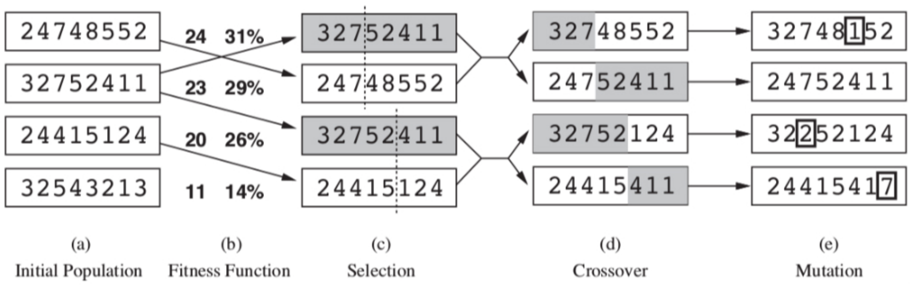

Figure 4.1: Genetic Algorithm

Let us illustrate GA using the following example[21]. To start GA we need to initialize a population with some random individuals; here we use digital strings to represent those individuals' chromosomes. See Figure 4.1 (a).

In each generation a fitness function is used to evaluate the fitness of the individuals in the environment, see Figure 4.1 (b). We then pick the individuals from the present population, and the rule is that the higher the fitness score, the more likely the individuals will be chosen. In Figure4.1 (c), the first individual was picked once, and the second picked twice, while the third picked only once, however, the last individual has a relatively low fitness score and was not picked even once, so it failed to transmit its gene to the next generation. Note that here we fixed the population size the same as the previous generation's.

We introduce stochastic invariants in the next steps. In Figure 4.1 (d) the chromosomes of the parents have a small probability of crossing over at some certain crossover points, which can be a fixed or random number. In the crossover process, the first child gets the first part of the first parent, and the second part of the second parent, while the second child gets the first part of the second parent, and the second part of the first parent.

At last, after crossover, each gene locus has a very small probability of mutation. See Figure 4.1 (e). For example, the sixth locus of the first child mutated from 5 to 1.

A brief version of pseudo codes are shown as follows[21]:

Algorithm 1 Genetic Algorithm

```
1: function GA(population, FITNESS) inputs:
2:     population, a group of individuals
3:     FITNESS, the fitness function
4: loop:
5:     new_population ← an empty set
6:     for i=1 to SIZE(population) do
7:         $x$ ← RANDOM-SELECTION(population, FITNESS)
8:         $y$ ← RANDOM-SELECTION(population, FITNESS)
9:         child ← REPRODUCE-CROSSOVER($x, y$)
10:        if RANDOM-NUMBER() $<$ some small number then
11:            child ← MUTATE(child)
12:        new_population.APPEND(child)
13:        population ← new_population
14:        if individuals fit enough or enough generations then
15:            break
16: return population
```

Here the REPRODUCE-CROSSOVER funtion will determine whether to crossover. If crossover process is not implemented, it will just return the original children x and y.

4.2 The GA-RRL System

In our optimization process the chromosomes of the individual consist of four parts of genetic loci: weights of neurons, learning speed, threshold, and moving average coefficient. Each of the latter three parts only has one genetic locus inside, whereas the part of the weights for neurons are composed of many genetic loci:

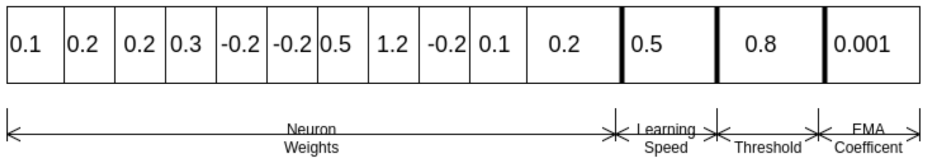

Figure 4.2: The GA-RRL Chromosome

The Figure 4.2 above shows an example of the chromosome in which the first 11 loci are the weights of the neurons, the 12th for the learning speed, the 13th for the threshold of the hypertangent function, and the 14th for the exponential moving average coefficient. For the weights of the neurons, the first 9 loci determine the weights for the previous 9 bars' returns. The 10th determines the weight for the previous position held, and the 11th is for the noises.

As mentioned in the previous section, pure RRL system requires good initial guesses for parameters, which are difficult to determine in practice. Intuitively, we may start by using all zeros for the neurons' weights, but empirical results show that this initial setting may not achieve a local optimum in reasonable given time, and is worse than other local optima. However, after combining RRL with GA, we can start with random values of parameters, and the parameters can change during the GA evolution in which the individuals that better fit the environment will be more likely to survive.

We expect the GA-RRL system to improve the convergence speed, but this GA-RRL system actually has a two-sided effect. Two main negative effects are as follows:

I. During the selection process, although we pick the parents by probability proportional to the fitness scores, sometimes it is possible that good individuals are all missed, leaving only bad individuals in the next generation. This is more likely to happen when the population size is small.

II. When the solution is near a local optimum, although RRL with an appropriate learning speed leads to the optimum, GA may drive the solution away. This means that RRL and GA may conflict. A typical case is a small population with a relatively high mutation rate, and when it approaches the optimum at a certain generation, all the individuals may encounter mutation during the evolution into the next generation, thus the solution will deviate far away from the optimum.

To avoid the first problem, we used the genetic algorithm with elitist model (EGA)[20], the core idea of which is to always preserve the best individual in the previous generation.

As to the second problem, at the beginning of the research we actually used the integrated GA-RRL method with constant parameters all the way through the optimization, only to find that the convergence becomes substantially slower than expected, and it cannot give a reasonable solution in affordable time. To fully take advantage of the GA but avoid conflict with RRL, we redesigned the GA-RRL system in the following way.

I. We use GA and RRL together to search for a good initial set of parameters. Here we initialize the parameters using uniform random numbers, and in each following evolutionary step, we first implement GA and then RRL to update the parameters. In GA we use a small but relatively higher crossover and mutation rate compared to the fine-tuning mode to be discussed later, and we only run this searching process for a small number of generations k. If we cannot find a good initial set of parameters after k generations it means that our random guess is not good, so we reset the algorithm and start over again, until we can find a good initial set whose fitness score is larger than some certain threshold. Here the threshold should be neither too high nor too low. If it is too high, as mentioned previously, the GA may take too much time to hit that threshold, whereas if it is too low the initial solution may be too far from the optimum.

II. After getting a good initial set of parameters, we start fine-tuning the parameters. Here we mainly use RRL and only use GA to adjust the learning speed. For the loci of neurons' weights, threshold, and EMA coefficient, we set the crossover and mutation rate to be zero. For learning speed we still allow a mutation rate high enough, and for computing convenience, the mutation process can be set to only switch the learning speed among some fixed values. In this fine-tuned mode we generate a few (for example 10) inidividuals at each generation, and the difference among them mainly lies in the learning speed. During the evolutionary process the population in each generation consists of individuals with different learning speeds but all other parameters almost the same. In addition, the EGA will guarantee the best one to survive in the next generation.

In the first step our goal is to approach a certain local optimum closely enough. GA weights more in the first step as we allow cross-over and mutation on the entire chromosome. Then in the second step, we use mainly RRL to fine-tune the parameters. We do this because RRL guarantees the correct direction of learning without random guesses, so with an appropriate learning speed RRL converges to the optimum a lot faster than GA. However, RRL itself cannot dynamically decide on a good learning speed up to date, so we will fine tune the parameters with GA only adjusting the learning speed in the second step.

The performance of this evolutionary RRL system depends on many parameters. We have no time to do sensitivity analysis for all of them, and here we select the parameters by experimental results. See the following Table 4.1 for a brief description.

Parameter Name	Default Value	Description
neurons	10/18/32	Number of neurons in the ANN, equal to the number of previous returns used plus two.
timelevel	5 mins	The time interval to be used.
w_range	[-1,1]	Initial range of weight for each neuron, used in GA initial value picking process. The final weight can exceed this range in the finue-tune mode.
ls_range	[0.001,2]	Range of learning speed. In the fine-tune mode, the learning speed can only be some fixed values.
thres_range	[0,1]	Range of the threshold of the hypertangent function to decide the position.
coef_range	[0.0001,0.01]	Exponential moving average coefficient.
lowest_sp	-0.01	The least stop value of Sharpe ratio in the initial parameters search.
fine_tune	False	When it is True, the mutation and crossover rate will be very small and parameters can only change in a very small range, and the learning speed can only be chosen from some fixed values.
cross_rate	0.1/0	Crossover rate in GA. The default value may be different in the fine-tune mode.
mute_rate	0.02/0	Mutation rate in GA, except learning speed. The default value may be different in the fine-tune mode.
ls_mute_rate	0.5/0.5	Mutation rate for learning speed in GA. The default value may be different in the fine-tune mode.
init_pop_num	10/5	Initial population size. The default value may be different in the fine-tune mode.
pop_limit	20/10	Maximum population size. The default value may be different in fine-tune mode.
gen_limit	10/100	Maximum number of generation, the default value may be different in fine-tune mode.
end_sp	0.10	The least stop value of sharpe ratio in fine-tuning parameters search.
ls_prop	[0,0.005,0.025, 0.1,0.5,1]	The proportions of the maximum learning speed, used for fixed learning speed choices in fine-tune mode.
windows	[0,400000, 400000,450000]	The start and end point of training window and testing window.
com	3	The commission fees for one unit of change of position per lot. For example, to buy 1 lot and then close totally change 2 units positions.

Table 4.1: Parameters Description

5 Performance Evaluation

5.1 Algorithm Performance

Although we have a large amount of 1-minute EUR/USD historical data, the limited computing resource restricts our ability to perform backtest research thoroughly on the whole data set. Here we only used 5-minute level data, and we picked three subsets of the data with equal quantity, used different numbers of neurons at three fixated level, and ran our learning system three times for each scenario. The detailed performance on each data set is shown as follows.

5.1.1 Dataset I

The first dataset contains the data points from number 200,000 to 400,000 for the training set, and number 400,001 to 420,000 for the test set. The training set ranges from UTC 2007/7/12 18:17:00 to UTC 2008/1/23 16:37, roughly half a year, and the test set ranges from UTC 2008/1/23 16:38 to UTC 2008/2/12 13:57, roughly three weeks. We ran our GA-RRL algorithm using 10, 16, 32 neurons. Due to limited computing resources, each test took about 2 hours. Here for each number of the neurons we just ran 3 tests.

Number of Returns	Running Test Number	Training P&L	Training Sharpe Ratio	Test P&L	Test Sharpe Ratio
8	1	34132	0.021772	3485	0.019190
8	2	27912	0.018401	3791	0.021641
8	3	34556	0.021976	2963	0.016285
14	1	39173	0.024569	894	0.004849
14	2	33556	0.021575	2693	0.014945
14	3	33004	0.021065	2188	0.012038
30	1	32590	0.020627	1516	0.008296
30	2	39528	0.024809	-374	-0.002031
30	3	36673	0.023412	-1014	-0.005579

Table 5.1: Backtest Results on Dataset I

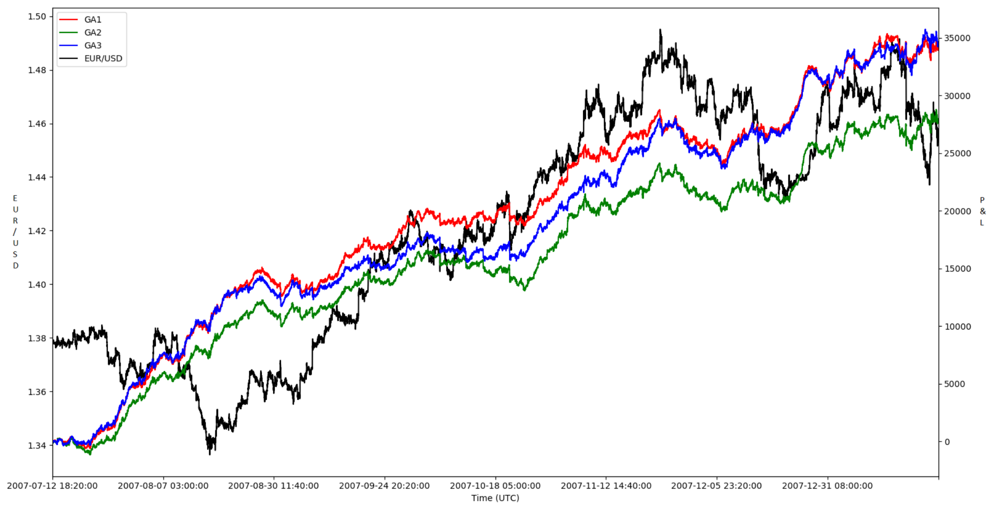

Figure 5.1: Training Performance on Dataset I, 8 Neurons

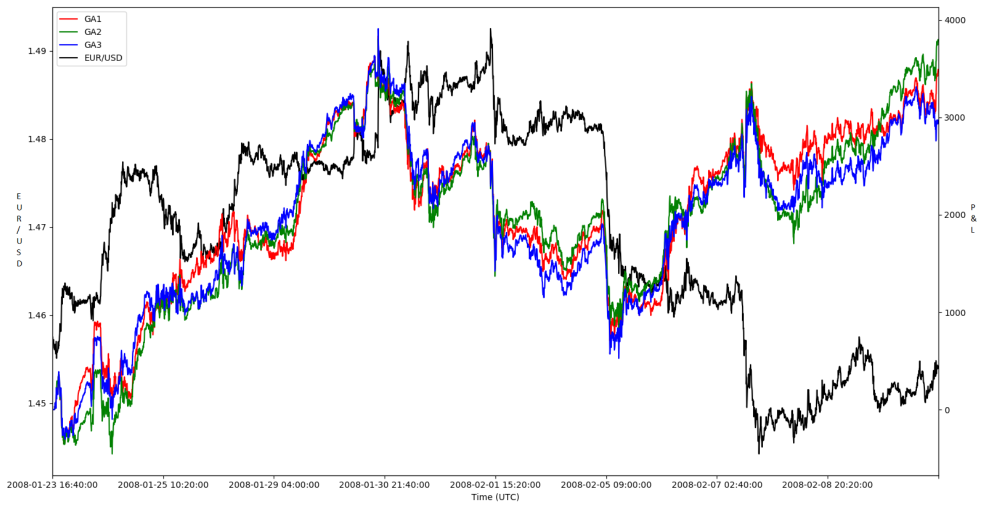

Figure 5.2: Test Performance on Dataset I, 8 Neurons

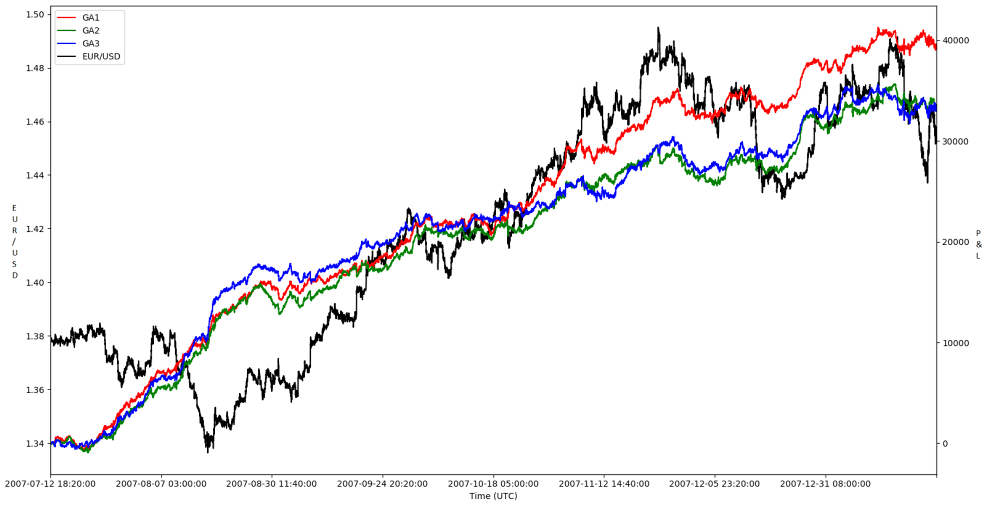

Figure 5.3: Training Performance on Dataset I, 16 Neurons

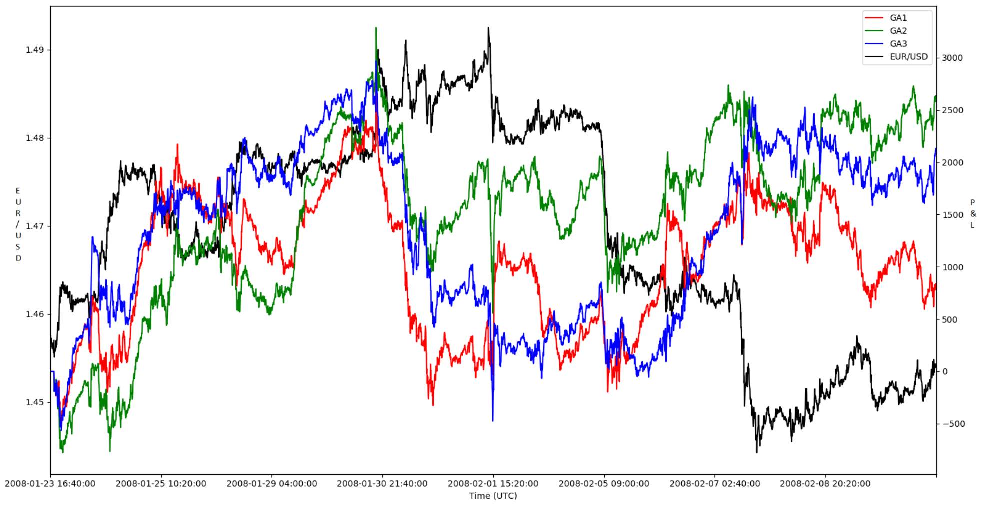

Figure 5.4: Test Performance on Dataset I, 16 Neurons

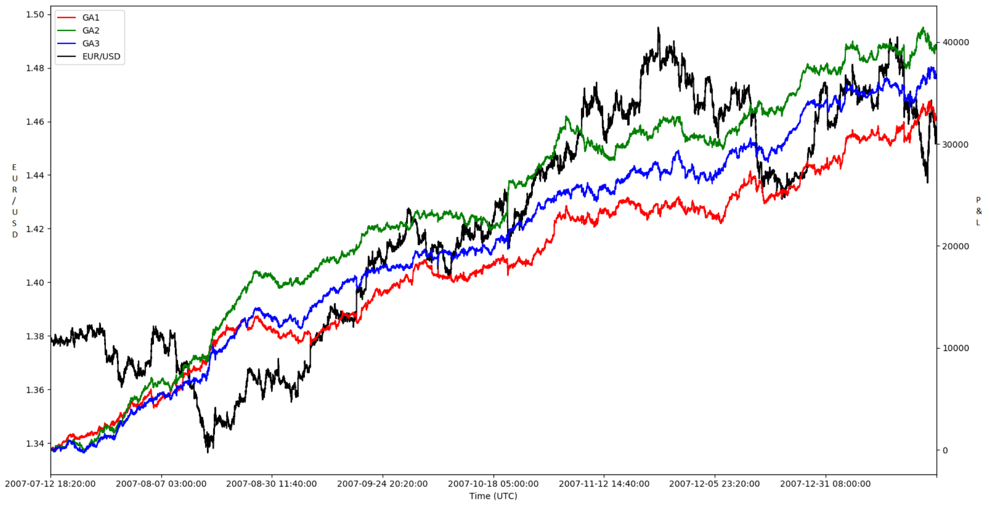

Figure 5.5: Training Performance on Dataset I, 30 Neurons

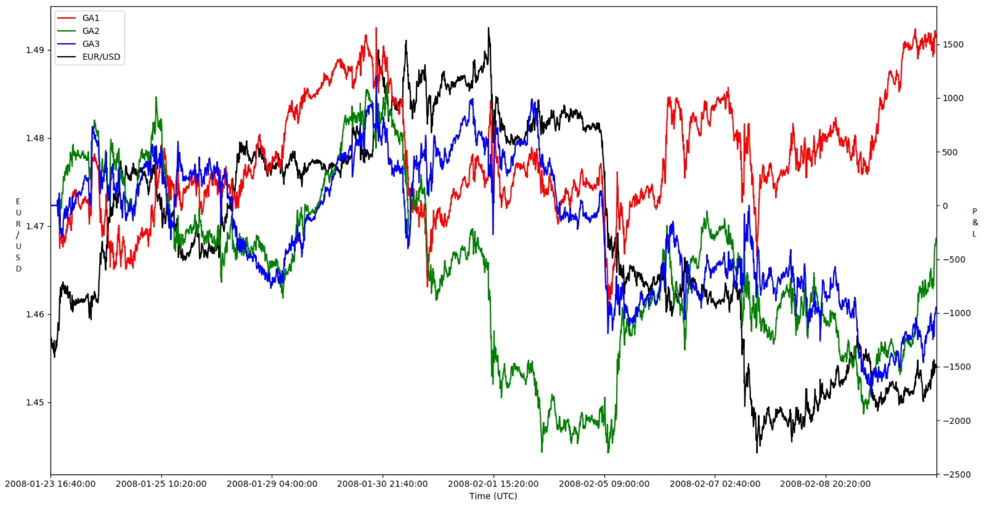

Figure 5.6: Test Performance on Dataset I, 30 Neurons

Among all three training datasets, the forex price in the first training dataset shows the most stable upward trend. The training performance is amazing: for the 9 running tests, compared to the simple long-position holding strategy (LHS) which profits 7809.0, the average profit is 34569.3, which amounts 442% of the LHS. Besides, the best GA-RRL running result profits 39173.0 (501% LHS) and the worst profits 27912.0 (357% LHS).

The test data, however, outlined a more unstable unilateral trend on the whole set than the training data, but it illustrates an upward trend in the first half and a downward trend in the second half. On the test dataset, while the simple long-position holding strategy only profits 250.0, the GA-RRL system gains an average profit 1793.6 (717% LHS). However, the performance varies, with the best profits 3791.0 (1516%) and the worst losses 1014 (-405% LHS).

In the training set the system with more neurons performs better, however, in the test set, the system with the least neurons wins. This is probably because the price changes depend more on the recent returns and using returns from too far in the past may cause overfitting.

Among the 9 GA-RRL tests, 7 of them made a profit. Considering that this GA-RRL system has no risk-control mechanism, the result shows a great prospect in practical trading.

5.1.2 Dataset II

The second dataset contains the data points from number 1,200,000 to 1,400,000 for the training set, and number 1,400,001 1,420,000 for the test set. The training set ranges from UTC 2010/3/11 05:57:00 to UTC 2010/9/22 02:17, roughly half a year, and the test set ranges from UTC 2010/9/22 02:18 to UTC 2010/10/11 23:37, roughly three weeks.

Number of Returns	Running Test Number	Training P&L	Training Sharpe Ratio	Test P&L	Test Sharpe Ratio
8	1	-1540	-0.000684	-715	-0.003056
8	2	-1545	-0.000699	2512	0.010867
8	3	-1523	-0.000677	3263	0.013976
16	1	23524	0.010466	-9809	-0.041818
16	2	5577	0.002511	-4271	-0.018387
16	3	12245	0.005464	-4191	-0.018077
30	1	10020	0.004422	-13195	-0.056105
30	2	13509	0.005978	987	0.004221
30	3	8179	0.003639	-4902	-0.021078

Table 5.2: Backtest Results on Dataset II

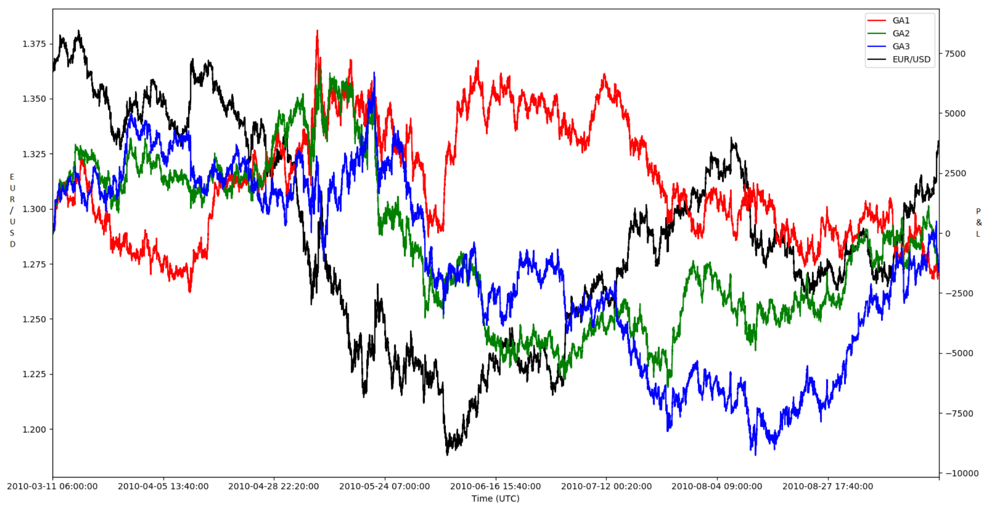

Figure 5.7: Training Performance on Dataset II, 8 Neurons

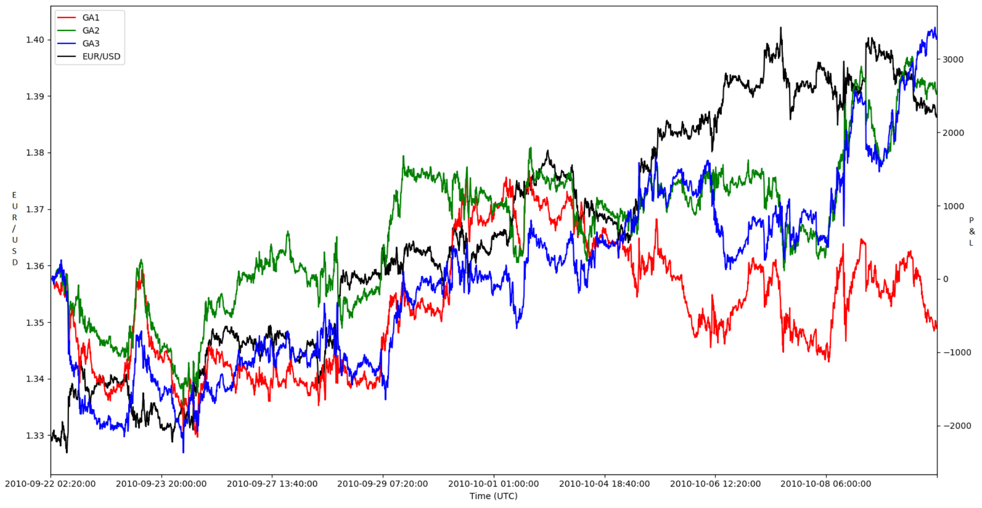

Figure 5.8: Test Performance on Dataset II, 8 Neurons

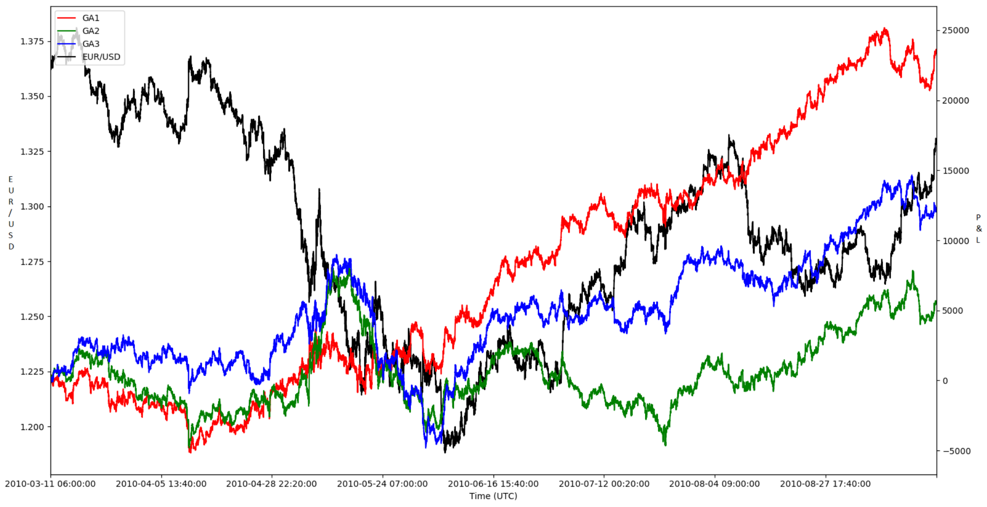

Figure 5.9: Training Performance on Dataset II, 16 Neurons

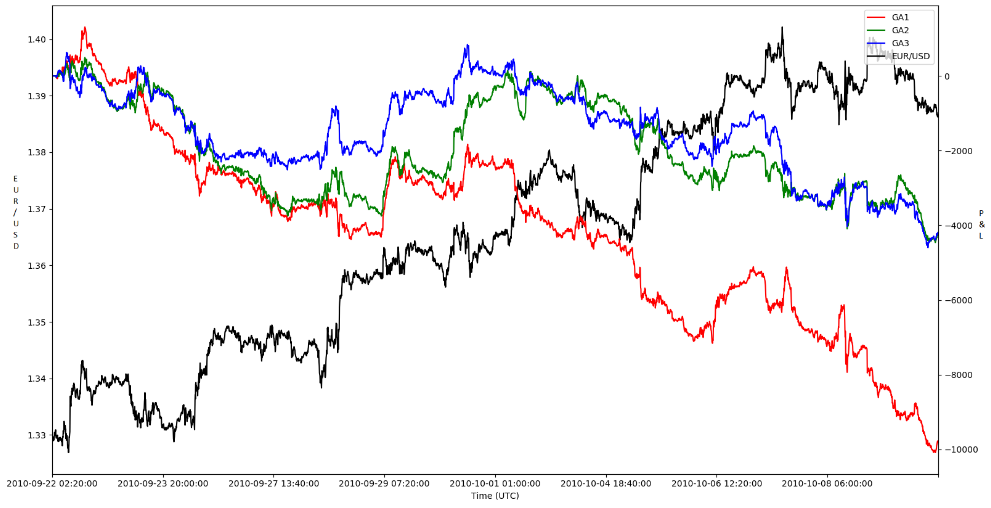

Figure 5.10: Test Performance on Dataset II, 16 Neurons

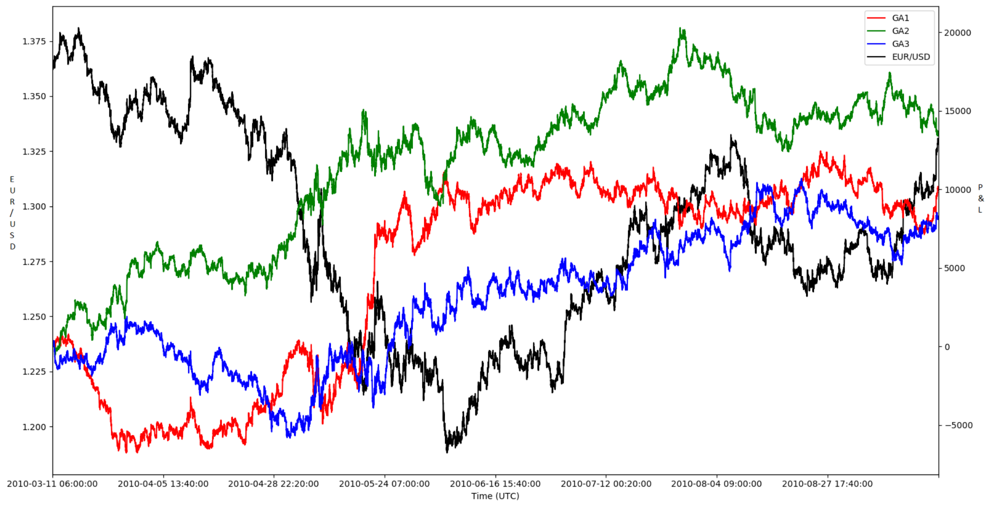

Figure 5.11: Training Performance on Dataset II, 30 Neurons

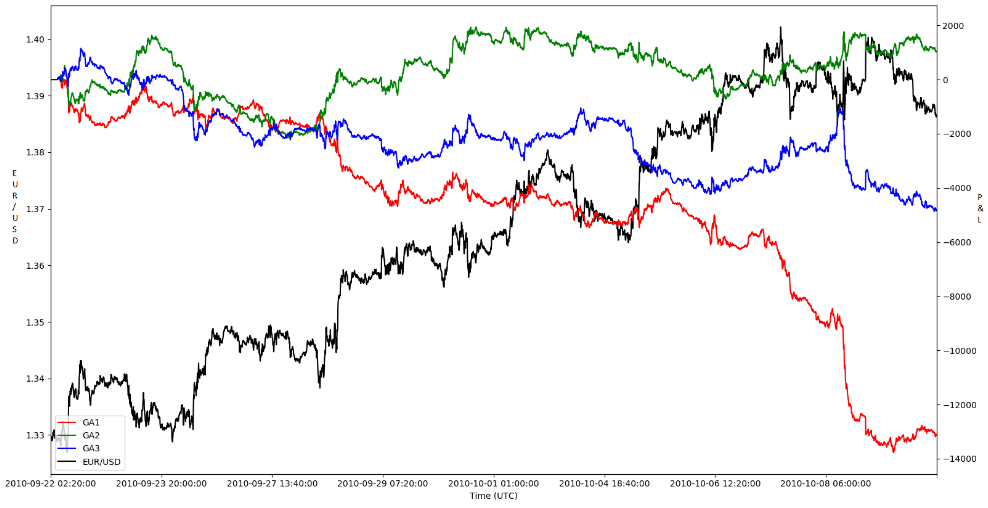

Figure 5.12: Test Performance on Dataset II, 30 Neurons

When running on the second dataset, the program seems to struggle during the initial parameter search. It is harder to train GA-RRL for the second dataset than the first one, probably because the training price data is shaky and does not show a stable moving trend as in the first dataset. On the training data, for the 9 running tests, a simple short-position holding strategy (SHS) profits 3483.0, while the average of GA-RRL system profits 7605, around 218% of SHS. The best result profits 23524 (675% SHS) whereas the worst loses 1540 (-44% SHS).

The test data shows a better trending property than the training data. On the test dataset, LHS wins 5724, and GA-RRL system loses on average -3369 (-58% LHS). The performance varies significantly, with the best profits being 3263 (41% LHS) and the worst losses being 13195 (-230% LHS). These facts suggest that the GA-RRL system is not always stable and needs to be refined before being used in practice.

Similarly, in the training set, the systems with more neurons outperform, but tend to show worse performance on the test data; the systems using less neurons may perform worse on the training data, but tend to outperform on the test data. For example, all 8-neuron systems lose about a thousand dollars in training, but both turn to make a profit of more than two thousand on the test set, and among the 9 cases they perform as the top 2. However, the system that wins the most on training data (675% SHS) turns out to lose 9809 (-171% LHS), showing the second worst performance among all. This is probably due to the overfitting problem; we may find some hints from the figures.

In figure 5.7, while GA2 and GA3 show similar moving paths of P&L, GA1 seems to be a lot different. There are more big gaps in GA1, for example, at around the beginning of June 2016, GA1 profits almost 5000 in two gaps. This indicates potential overfitting problems, and in the test dataset, GA1 does perform worse than the other two.

Further, in figure 5.9 GA1 shows a strikingly stable performance regardless of the volatile price in the training dataset. Although in some cases this is possibly reasonable and the parameters will also be good on a test set, we still have to be careful when it seems too good to be true. In our test data set, GA1 results in a straight loss, as striking as the profit made on the training dataset. Therefore, good performance on the training dataset does not guarantee good results on the test dataset. We have to observe the trading behavior carefully on the training set and ponder on the possible overfitting conditions.

5.1.3 Dataset III

The third dataset contains the data points from number 2,600,000 to 2,800,000 for the training set, and number 2,800,001 to 2,820,000 for the test set. The training set ranges from UTC 2013/12/2 11:17:00 to UTC 2014/6/13 07:37, roughly half a year, and the test set ranges from UTC 2014/6/13 07:38 to UTC 2014/7/3 04:57, roughly three weeks.

Number of Returns	Running Test Number	Training P&L	Training Sharpe Ratio	Test P&L	Test Sharpe Ratio
8	1	5622	0.005052	1194	0.01431
8	2	7165	0.006305	425	0.004956
8	3	4316	0.003814	817	0.009579
16	1	6603	0.005867	-861	-0.010007
16	2	6604	0.005943	133	0.0016
16	3	6961	0.00623	-1060	-0.012702
30	1	13302	0.011698	-1116	-0.013163
30	2	13396	0.011737	-570	-0.006639
30	3	10022	0.008865	-904	-0.010576

Table 5.3: Backtest Results on Dataset III

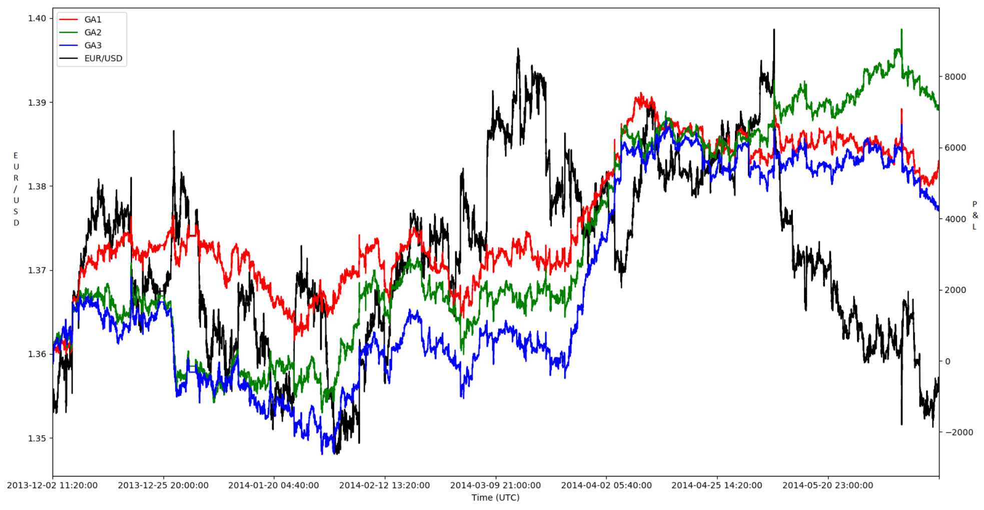

Figure 5.13: Training Performance on Dataset III, 8 Neurons

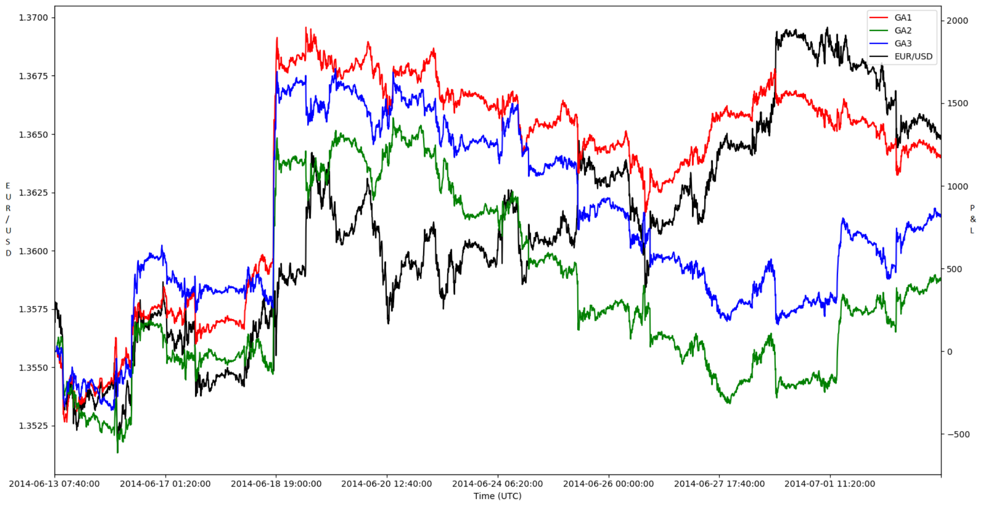

Figure 5.14: Test Performance on Dataset III, 8 Neurons

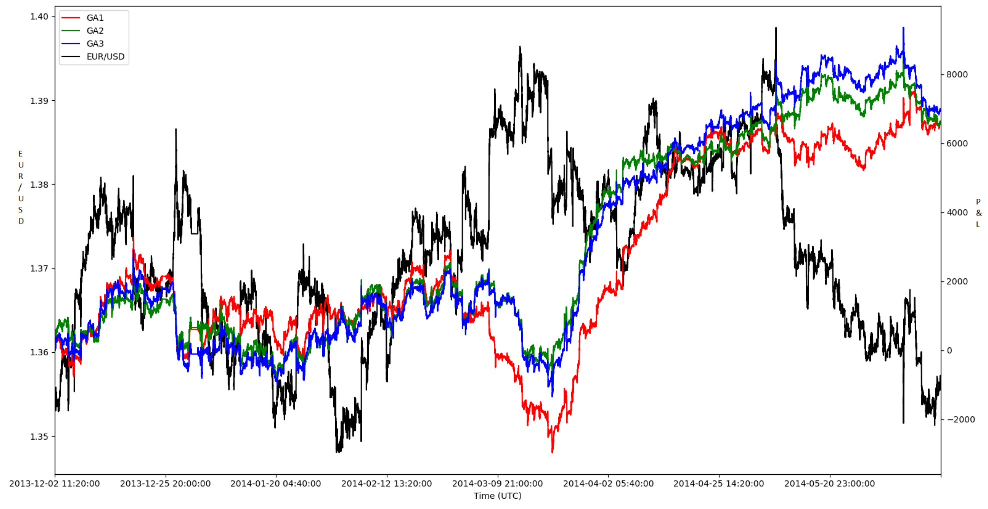

Figure 5.15: Training Performance on Dataset III, 16 Neurons

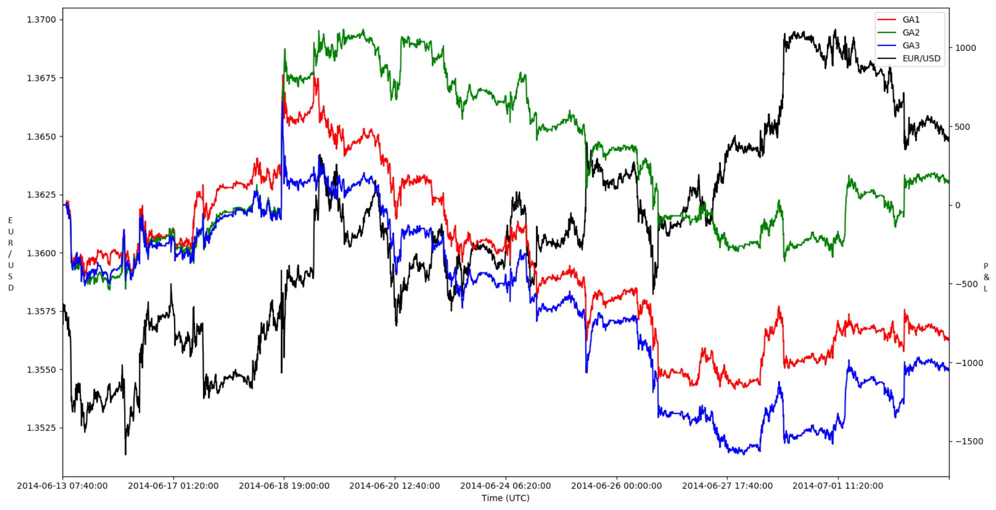

Figure 5.16: Test Performance on Dataset III, 16 Neurons

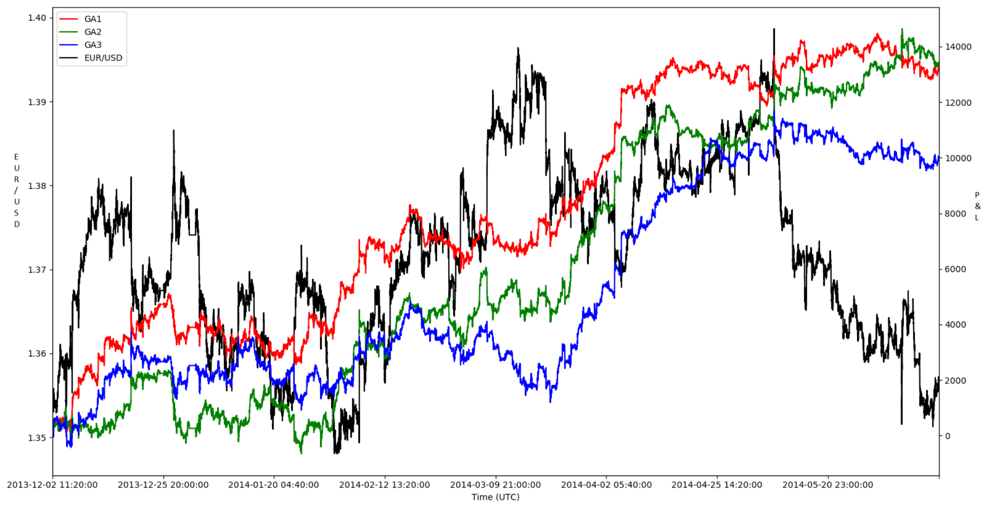

Figure 5.17: Training Performance on Dataset III, 30 Neurons

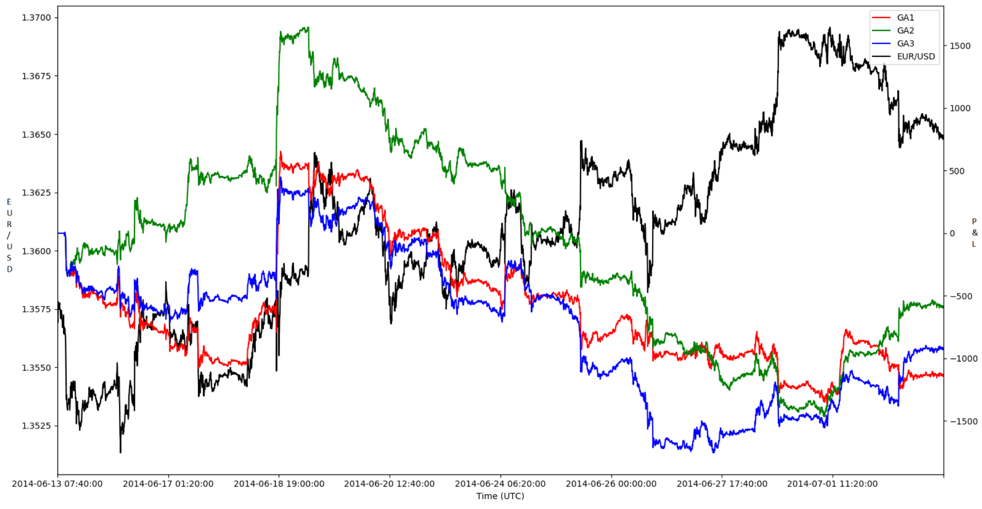

Figure 5.18: Test Performance on Dataset III, 30 Neurons

The historical price in dataset III shows the strongest mean-reversion property among the tree datasets. From start to finish the price only changes a total of 24.1 pips in the training data. For the 9 running tests a simple short-position holding strategy (SHS) profits 241, while the average of GA-RRL system profits 8221, which amounts to 3411% of the SHS. The best wins 13396 (5558% SHS) and the worst loses 1540 (1790% SHS).

The test data shows a slightly stronger trending property, but the price still frequently and dramatically retraces over the time. On the test dataset LHS makes a profit of 807, and the GA-RRL system loses on average 215 (-26% LHS). The performance on the test data varies, but almost within a small interval of $[-1200, 1200]$. The best system profits 1194 (147% LHS), and the worst loses 1116 (-138% LHS). Although GA-RRL systems do not continue their marvelous profitability on the training set, neither do they lose much money. This is a very important fact because we have learned previously that in dataset I, GA-RRL indicates a good profitability in a trending market. However, when the price is shaky, GA-RRL still shows a good property of resisting bumpy movements, which is a potential advantage of using the GA-RRL system in live trading.

In the training set, systems with the largest number of neurons outperform, but still perform weaker on test dataset, and vice versa. This seems a pattern.

5.2 Conclusions

Based on the performance and analyses above we have the following conclusions and inferences:

1. The GA-RRL system has potentially good profitability. It is likely to beat the market in a trending market condition, and may resist shaky prices and survive in a bumpy market.

2. The GA-RRL system is more likely to be a trend-following system than a mean-reversion system.

3. The GA-RRL system using less but an adequate number of previous returns may outperform the system using more previous returns.

4. The performance of GA-RRL systems varies since GA will generate different sets of parameters. We should observe the historical trading behavior to check the overfitting problem and decide which to use.

5. Good performance on training dataset does not guarantee good performance in practical trading. Overfitting problems require much attention.

6 Future Work

6.1 Risk Control Mechanism

Due to the complexity of programming and limited computing resources, we have not introduced any risk control mechanisms yet. Here are some techniques we plan to use in future research.

1. Trailing stop loss. A trailing stop loss order will automatically adjust the peak point for a stop loss reference, using the price at which the transaction has gained the highest profit. For example, if we send a trailing stop loss order of 5 pips, and the present EUR/USD pair price is 1.1000, then at this beginning time our stop loss price is 1.0095. If after 1 minute the price goes straight up to 1.1010 without any retracement, then our stop loss price will be updated to 1.1005, with the referencing peak point as 1.1010 instead of 1.1000. As a widely used mechanism, trailing stop loss can protect the trade from heavy loss and lock in the realized profit. Determining the stop loss price is an art which we do not venture further into in this project.

2. Maximum/minimum holding period. Setting up a maximum holding time for a transaction is likewise a useful technique to control risk. Some conservative trading systems rarely send trading signals, and once a transaction is triggered, it tends to hold the position for a long time during which no new trading signals are sent. By setting an appropriate maximum holding time, we may cut the unnecessary risk exposure, thus decreasing the risk. On the other hand, some trading systems send signals very frequently, many of which are actually noises. By setting a minimum holding time we can reduce the trading times, thus reducing the transaction costs and possibly the losses due to the noise signals.

3. Hibernation. There is no perfect trading system, and most systems will only fit in with certain market conditions. When the market conditions have changed, the trading system may no longer be profitable in a certain period. Therefore, when the system keeps losing sums of money during the recent time, it may indicate that the

system is no longer efficient. Therefore we can temporarily stop the trading system for some time and wait for the market condition to change.

4. Trading time restriction. In the previous section of performance evaluation, we did a backtest without choosing the specific trading time. Usually there would be more noises when the trading volume is low, thus lowering the efficiency of the algorithm. Empirical results have shown that many trading algorithms will perform much better in a specific time period. We will restrict the trading time in our future research.

6.2 The Rolling Window Backtest

In the rolling window backtest we use a batch of historical data for training and a smaller batch of data followed for testing. When one backtesting process is done we forward the backtesting window a little bit, update our training and test data, then repeat the procedure until we have backtested all the historical data.

We take the rolling window backtest on the data between 2012/1/1 and 2014/12/31 for an example. We start backtesting using the data from 2012/1/1 to 2012/12/31, with a test data from 2013/1/1 to 2013/1/31, which means that we use a rolling window of 1 year for training and 1 month for testing. After the first rolling window backtest is done, we forward the starting point of training data to 2012/2/1, ending point of training data to 2013/1/1, and use the data from 2013/2/1 to 2013/2/31 for testing. Repeating this procedure until the test data ends at 2014/12/31, we can evaluate the total performance on the rolling test data from 2013/1/1 to 2014/12/31.

For the same historical data, compared to simple static train-test split sampling, the rolling window method can provide more subsets of samples, and for each set of subsamples the training is more up-to-date. However, due to the strong correlation and a large proportion of overlapping seen by the samples of rolling windows, if there is a major change to the properties of data in a certain period, the rolling window method runs the risk of not adjusting the parameters quickly enough.

Appendix A Codes

1. Online Learning

```python
'''
Created on Mar 9, 2017

@author: ethansong
'''

import pandas as pd
import numpy as np


def position(num, prepos, weight, thres):
    """Determine the postion to take at a certain bar.

    Args:
        num: the number of the bar.
        prepos: previous postion.
        weight: weight vector used. For online learning it's the previous\
        bar, for accumulative learning it remains the same for one training process.
        thres: the threshold for hypertangent function to open a non-neutral position.

    Returns:
        posvec: the position vector to be multiplied by the weight.
        pos: the position to be taken at the bar.

    """
    length = len(weight)
    posvec = pips[num-length+3:num+1]
    posvec = np.append(posvec, [1,prepos])
    raw_pos = np.tanh(posvec.dot(weight))

    if abs(raw_pos) <= thres:
        pos = 0
    else:
        pos = np.sign(raw_pos)

    return posvec, pos


def daily_return(num,coms,lots,pos,prepos):
    """Calculate return for a single bar.

    Args:
        num: the number of the bar.
        coms: commissions for a single change of position.
        lots: number of lots traded.
        pos: present position.
        prepos: previous postion.

    Returns:
        dreturn: return for the bar

    """
    dreturn = lots * (10 * prepos * pips[num] - coms * abs(pos - prepos
    ))

    return dreturn


def ma_shapre(num,coef,dreturn,pre_A,pre_B):
    """Calculate moving averaged sharpe ratio.

    Args:
        num: the number of the bar.
        coef: moving average coefficient.
        dreturn: single bar return for the previous bar.
        pre_A: previous A, approximately equal to E(Rt).
        pre_B: previous B. approximately equal to E(Rt^2).

    Returns:
        A: approximate E(Rt).
        B: approximate E(Rt^2).

    """
    A = pre_A + coef * (dreturn - pre_A)
    B = pre_B + coef * (dreturn**2 - pre_B)

    return A, B


def diffs(num, weight, dreturn, pre_A, pre_B, pos, posvec, prepos, lots
    , pre_dfw, coms, coef):
    """Calculate the differentiations.

    Args:
        num: the number of the bar.
        weight: weight vector used. For online learning it's the
    previous\
        bar, for accumulative learning it remains the same for one
    training process.
        dreturn: single bar return for the previous bar.
        pre_A: previous A, approximately equal to E(Rt).
        pre_B: previous B. approximately equal to E(Rt^2).
        pos: the position to be taken at the bar.
        posvec: the position vector to be multiplied by the weight.
        prepos: previous postion.
        lots: number of lots traded.
        pre_dfw: the previous value of dFt/dw.
        coms: commissions for a single change of position.
        coef: moving average coefficient.

    Returns:
        dUt_dWt: the online approximate gradient of utility function.
        dfw: dFt/dw.

    """
    dDt_dRt = 0 if  (pre_B - pre_A**2) == 0 else (pre_B - pre_A *
    dreturn)/(pre_B - pre_A**2) ** (1.5)
    dRt_dFt = -coms * lots * np.sign(pos - prepos)
    dRt_dFtpre = coms * lots * np.sign(pos - prepos) + lots * pips[num]
    * 10
    dfw = (1 - np.tanh(weight.dot(posvec)) **2 ) * (posvec + weight[-1]
    * pre_dfw)
    dUt_dWt = dDt_dRt * (dRt_dFt * dfw + dRt_dFtpre * pre_dfw) * coef #
    online training

    return dUt_dWt, dfw


def training(weight, learn_speed, dfw, thres, lots, coms, coef, pre_A,
    pre_B, prep):
    """The online training process.

    Args:
        weight: weight vector used. For online learning it's the
    previous\
        bar, for accumulative learning it remains the same for one
    training process.
        learn_speed: learning speed.
        dfw: initial dFt/dw.
        thres: the threshold for hypertangent function to open a non-
    neutral position.
        lots: number of lots traded.
        coms: commissions for a single change of position.
        coef: moving average coefficient.
        pre_A: initial A, which will approximately become equal to E(Rt
    ).
        pre_B: initial B. which will approximately become equal to E(Rt
    ^2).
        prep: default position taken at the begining. {1,0,-1} for {
    long, neutral, short}.

    Returns:
        weight: optimized weight.
        pre_A: moving averaged E(Rt) at the end of training.
        pre_B: moving average E(Rt^2) at the end of training.
        profit: total profit and loss.
        spratio: final optimized sharpe ratio.

    """
    L = len(pips)
    pos = 0
    prepos = prep
    pre_dfw = dfw
    profit = np.zeros(len(weight))
    sum_Rt = np.array([])
    sum_Rt2 = np.array([])
    # With a very tiny coefficient,
    # you may use the sum of the online learning gradient
    # to approximate the accumatlive differential sharpe ratio
    # sum = 0
    for i in range(len(weight),L):
        num = i
        posvec, pos = position(num, prepos, weight, thres)
        dreturn = daily_return(num, coms, lots, pos, prepos)
        dUt_dWt, dfw = diffs(num, weight, dreturn, pre_A, pre_B, pos, posvec, prepos, lots, pre_dfw, coms, coef)

        prepos = pos
        pre_dfw = dfw
        pre_A, pre_B = ma_shapre(num, coef, dreturn, pre_A, pre_B)

        weight = weight + learn_speed *  dUt_dWt
        profit = np.append(profit, profit[-1] + dreturn)
        sum_Rt = np.append(sum_Rt, dreturn)
        sum_Rt2 = np.append(sum_Rt2, dreturn ** 2)
        # sum = sum + dUt_dWt

    # Approximate accumulative diff sharpe ratio using the sum of online gradient
    # weight = weight + learn_speed * sum
    sumr = np.sum(sum_Rt) / (L-len(weight))
    sumr2 = np.sum(sum_Rt2) / (L-len(weight))
    spratio = sumr / np.sqrt(sumr2 - sumr ** 2)

    # print(profit[-1], spratio, pre_A, pre_B,  weight)
    return weight, pre_A, pre_B, profit, spratio
```

2. Accumulative Learning

```
'''
Created on Mar 9, 2017

@author: ethansong
'''

import pandas as pd
import numpy as np

def position(num, prepos, weight, thres):
    """Determine the postion to take at a certain bar.

    Args:
        num: the number of the bar.
        prepos: previous postion.
        weight: weight vector used. For online learning it's the
    previous\
        bar, for accumulative learning it remains the same for one
    training process.
        thres: the threshold for hypertangent function to open a non-
    neutral position.

    Returns:
        posvec: the position vector to be multiplied by the weight.
        pos: the position to be taken at the bar.

    """
    length = len(weight)
    posvec = pips[num-length+3:num+1]
    posvec = np.append(posvec, [1,prepos])
    raw_pos = np.tanh(posvec.dot(weight))

    if abs(raw_pos) <= thres:
        pos = 0
    else:
        pos = np.sign(raw_pos)

    return posvec, pos


def daily_return(num,coms,lots,pos,prepos):
    """Calculate return for a single bar.

    Args:
        num: the number of the bar.
        coms: commissions for a single change of position.
        lots: number of lots traded.
        pos: present position.
        prepos: previous postion.

    Returns:
        dreturn: return for the bar

    """
    dreturn = lots * (10 * prepos * pips[num] - coms * abs(pos - prepos
   ))
    return dreturn


def sharpe(num,dreturn,pre_A,pre_B, l):
    """Calculate accumulative sharpe ratio.

    Args:
        num: the number of the bar.
        dreturn: single bar return for the previous bar.
        pre_A: previous A. E(Rt).
        pre_B: previous B. E(Rt^2).
        l: length of the neurons.

    Returns:
        A: E(Rt).
        B: E(Rt^2).
    """

    A = (pre_A * (num - l) + dreturn) / (num + 1 - l)
    B = (pre_B * (num - l) + dreturn ** 2 ) / (num + 1 - l)
    return A, B

def diffs(num, weight, dreturn, pos, posvec, prepos, lots, pre_dfw,
   coms, L):
    """Calculate the differentiations.

    Args:
        num: the number of the bar.
        weight: weight vector used. For online learning it's the
   previous\
        bar, for accumulative learning it remains the same for one
   training process.
        dreturn: single bar return for the previous bar.
        pos: the position to be taken at the bar.
        posvec: the position vector to be multiplied by the weight.
        prepos: previous postion.
        lots: number of lots traded.
        pre_dfw: the previous value of dFt/dw.
        coms: commissions for a single change of position.
        coef: moving average coefficient.

    Returns:
        dRt_dWt
        dfw: dFt/dw.
        dA_dRt
        dB_dRt

    """
    dA_dRt = 1.0 / (L - len(weight))
    dB_dRt = 2.0 * dreturn / (L - len(weight))

    dRt_dFt = -coms * lots * np.sign(pos - prepos)
    dRt_dFtpre = coms * lots * np.sign(pos - prepos) + lots * pips[num]
```

```
     * 10
    dfw = (1 - np.tanh(weight.dot(posvec)) **2 ) * (posvec + weight[-1]
     * pre_dfw)
    dRt_dWt = (dRt_dFt * dfw + dRt_dFtpre * pre_dfw)

    return dRt_dWt, dfw, dA_dRt, dB_dRt


def training(weight, learn_speed, dfw, thres, lots, coms, prep):
    """The accumulative training process.

    Args:
        weight: weight vector used. For online learning it's the
    previous\
        bar, for accumulative learning it remains the same for one
    training process.
        learn_speed: learning speed.
        dfw: initial dFt/dw.
        thres: the threshold for hypertangent function to open a non-
    neutral position.
        lots: number of lots traded.
        coms: commissions for a single change of position.
        coef: moving average coefficient.
        prep: default position taken at the begining. {1,0,-1} for {
    long, neutral, short}.

    Returns:
        weight: optimized weight.
        profit: total profit and loss.
        spratio: final optimized sharpe ratio.

    """
    L = len(pips)
    l = len(weight)
    pos = 0
    prepos = prep
    pre_dfw = dfw
    A, B = 0 , 0
    # sum_Rt = 0
    profit = np.zeros(len(weight))
    ARt, BRt = np.array([]),np.array([])
    RL = np.empty((0,l),float)
    for i in range(l,L):
        num = i

        posvec, pos = position(num, prepos, weight, thres)
        dreturn = daily_return(num, coms, lots, pos, prepos)

        dRt_dWt, dfw, dA_dRt, dB_dRt = diffs(num, weight, dreturn,pos,
    posvec, prepos, lots, pre_dfw, coms, L)

        prepos = pos
        pre_dfw = dfw
        A, B = sharpe(num, dreturn, A, B, l)

        profit = np.append(profit, profit[-1] + dreturn)

        ARt = np.append(ARt, dA_dRt)
        BRt = np.append(BRt, dB_dRt)
        RL = np.append(RL, np.array([dRt_dWt]), axis = 0)

    dSt_dA = (B - A**2) ** (-0.5) + A**2 * (B - A**2) ** (-1.5)
    dSt_dB = -0.5 * A * (B - A**2) ** (-1.5)
    dSt_dRt = dSt_dA * ARt + dSt_dB * BRt
    RL = (RL.T * dSt_dRt).T
    spratio = A / np.sqrt(B - A**2)

    sum_dU = np.sum(RL, axis = 0)

    weight = weight + learn_speed * sum_dU
    print(profit[-1], spratio, weight)
    return weight, profit, spratio
```

3. Genetic Algorithm

```
'''
Created on Mar 24, 2017
```

```

@author: ethansong
'''

import rrlac as ac
import rrlol as ol
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import math


def ind_generator(neurons, w_low, w_high, ls_low, ls_high,\
                  thres_low, thres_high, coef_low, coef_high):
    """Generate individuals using uniform random numbers.

    Args:
        neurons = number of neurons to use
        w_low = lowest weight for individual neuron
        w_high = highest weight for individual neuron
        ls_low = lowest learn speed
        ls_high = highest learn speed
        thres_low = lowest threshold value
        thres_high = highest threshold value
        coef_low = lowest coefficient value
        coef_high = highest coefficient value

    Returns:
    ind: An individual representation.

    """
    w = np.random.uniform(w_low, w_high, neurons)
    ls = np.random.uniform(ls_low, ls_high)
    thres = np.random.uniform(thres_low, thres_high)
    coef = np.random.uniform(coef_low, coef_high)

    ind = [w, ls, thres, coef]
    return ind
```

```

# ind =  ind_generator(14, w_low, w_high, ls_low, ls_high,\
#                     thres_low, thres_high, coef_low, coef_high)
# print(type(ind))

def pop_copy(ind, num):
    """Generate a population by copying the same individual multiple
    times.

    Args:
        ind = individual to be copied.
        num = num of copies


    Returns:
    pop: population generated by copying.

    """
    pop = np.empty((0, 4), float)
    for _ in range(0, num):
        pop = np.vstack((pop, np.array([ind])))
    return pop

def pop_generator(num, neurons, w_low, w_high, ls_low, ls_high,\
                    thres_low, thres_high, coef_low, coef_high):
    """Generate a population using uniform random numbers.

    Args:
        num = number of the individuals in a population.
        neurons = number of neurons to use
        w_low = lowest weight for individual neuron
        w_high = highest weight for individual neuron
        ls_low = lowest learn speed
        ls_high = highest learn speed
        thres_low = lowest threshold value
        thres_high = highest threshold value
        coef_low = lowest coefficient value
        coef_high = highest coefficient value

    Returns:
    pop: A population.

    """
    pop = np.empty((0, 4), float)
    for _ in range(0, num):
        ind = ind_generator(neurons, w_low, w_high, ls_low, ls_high,\
                                    thres_low, thres_high, coef_low,
   coef_high)
        pop = np.vstack((pop, np.array([ind])))
    return pop


def fitness(population):
    """Calculate the fitness of the population.

    When calculating the fitness, The population will be updated by RRL
   .

    Args:
        population = the population to be measured.

    Returns:
    pop_new: The population after calculation and update.
    fitscores: fitness scores.
    normfscores: normalized fitness scores.
    pnl: profit and loss of the population.

    """
    pop_num = len(population)
    pop_new = np.empty((0, 4), float)
    fitscores = np.array([])
    pnl = np.array([])
    for i in range(pop_num):
        weight = population[i,0]
        learn_speed = population[i,1]
        thres = population[i,2]
        coef = population[i,3]
        weight, A, B, profit, sp = ol.training(weight, learn_speed,\
                                                   0, thres, 1, 3, coef,
    0, 0, 1)
        pop_new = np.vstack((pop_new, np.array([weight, learn_speed,
    thres, coef])))
        fitscores = np.append(fitscores, 10**(100*sp)) # You may adjust
     the parameter 100
        pnl = np.append(pnl,profit[-1])
    normfscores = fitscores / np.sum(fitscores)
    return pop_new, fitscores, normfscores, pnl


def evolution(pop_old, cross_rate, mute_rate, ls_mute_rate, gen_num,
    pop_lim,\
              w_low, w_high, ls_low, ls_high, thres_low, thres_high,
    coef_low, coef_high, lowest_sp, delicate = False):
    """Genetic algorithm evolution function.

    Args:
        pop_old = Initial population.
        cross_rate = cross rate.
        mute_rate = mutation rate for all parameters, except for the
    learning speed.
        ls_mute_rate = mutation rate for learning speed.
        gen_num = maximum generation number.
        pop_lim = maximum population size.
        w_low = lowest weight for individual neuron
        w_high = highest weight for individual neuron
        ls_low = lowest learn speed
        ls_high = highest learn speed
        thres_low = lowest threshold value
        thres_high = highest threshold value
        coef_low = lowest coefficient value
        coef_high = highest coefficient value
        lowest_sp = stop when the sharpe ratio is greater than this
    value
        delicate = default False. When it is True the mutation and
```

```
crossover\
        will be very less likely to happen and only change in a small
range.

    Returns:
    pop_max: The Individual that has the best sharpe ratio.
    sp_max: The best sharpe ratio among the population.
    pnl_max: The best pnl among the population.

    """
    ls = ls_mute_rate
    gen = 0
    pop_old, fitscores, normfscores, pnl = fitness(pop_old)
    pop_max = pop_old[normfscores.argmax()]
    sp_max = -9999
    while gen < gen_num and sp_max < lowest_sp:
        l = int(np.floor(min(pop_lim/2, len(pop_old))))
        pop_new = np.empty((0, 4), float)
        for _ in range(l-1):
            if gen == 0:
                ls_mute_rate = 0
            else:
                ls_mute_rate = ls
            par_x_num = np.random.choice(len(pop_old), p = normfscores)
            par_x = pop_old[par_x_num,:]
            par_y_num = np.random.choice(len(pop_old), p = normfscores)
            par_y = pop_old[par_y_num,:]
            child_a, child_b = reproduce(par_x, par_y, cross_rate)

            child_a = mutate(child_a, mute_rate, ls_mute_rate, w_low,
w_high, ls_low, ls_high,\
                    thres_low, thres_high, coef_low, coef_high, delicate
)
            child_b = mutate(child_b, mute_rate, ls_mute_rate, w_low,
w_high, ls_low, ls_high,\
                    thres_low, thres_high, coef_low, coef_high, delicate
)

```

```
            pop_new = np.vstack((pop_new, np.array([child_a])))
            pop_new = np.vstack((pop_new, np.array([child_b])))

        pop_new = np.vstack((pop_new, np.array([pop_max])))
        pop_elitist = np.copy(pop_max)
        # pop_elitist[1] = 0.0001
        pop_new = np.vstack((pop_new, np.array([pop_elitist])))
        pop_old, fitscores, normfscores, pnl = fitness(pop_new)

        gen += 1
        spratios = np.log10(fitscores)/100
        sp_max = np.max(spratios)
        pop_max = pop_old[spratios.argmax()]
        pnl_max = pnl[spratios.argmax()]
        # You may delete the following
        print(np.average(np.log10(fitscores)/100))
        print(pnl)
        print(np.log10(fitscores)/100)
        print(pop_max)
        print(gen, pnl_max, sp_max, "\n\n")
        if sp_max < -0.04:
            print('Too far from the optimum!')
            break

    return pop_max, sp_max, pnl_max

def reproduce(x, y, cross_rate):
    """The cross over procedure.

    Decide whether to cross over and if so implement it.

    Args:
        x, y = Parent individuals.
        cross_rate = cross over rate.

    Returns:
    x, y: Individuals after crossing over procedure.

    """
    if np.random.rand() < cross_rate:
        xw = x[0]
        yw = y[0]
        l = len(xw)
        split = np.random.choice(l)
        xw_new = np.append(xw[0:split], yw[split:])
        yw_new = np.append(yw[0:split], xw[split:])
        x[0] = xw_new
        y[0] = yw_new
    return x, y

def mutate(x, mute_rate, ls_mute_rate, w_low, w_high, ls_low, ls_high,\
                    thres_low, thres_high, coef_low, coef_high, delicate):
    """Mutation procedure.

        Decide whether to mutate and

    Args:

        mute_rate = mutation rate for all parameters, except for the learning speed.
        ls_mute_rate = mutation rate for learning speed.
        gen_num = maximum generation number.
        pop_lim = maximum population size.
        w_low = lowest weight for individual neuron
        w_high = highest weight for individual neuron
        ls_low = lowest learn speed
        ls_high = highest learn speed
        thres_low = lowest threshold value
        thres_high = highest threshold value
        coef_low = lowest coefficient value
        coef_high = highest coefficient value
        lowest_sp = stop when the sharpe ratio is greater than this value
        delicate = default False. When it is True the mutation and crossover\
        will be very less likely to happen and only change in a small
    range.

    Returns:
    pop_max: The Individual that has the best sharpe ratio.
    sp_max: The best sharpe ratio among the population.
    pnl_max: The best pnl among the population.

    """
    xw = x[0]
    l = len(xw)
    for i in range(l):
        if np.random.rand() < mute_rate:
            xw[i] = np.random.uniform(w_low, w_high)
        if abs(xw[i]) > w_high:
                xw[i] = np.sign(xw[i]) * w_high/2
    if np.random.rand() < ls_mute_rate:
        if delicate == False:
            x[1] = np.random.uniform(ls_low, ls_high)
        elif delicate == True:
            rge = ls_high - ls_low
            x[1] = np.random.choice([ls_low, ls_low+0.005*rge, ls_low
+0.025*rge,\
                                              ls_low+0.08*rge, ls_low+0.25*rge,
ls_low+0.50*rge,\
                                              ls_low+0.75*rge, ls_high])
    if np.random.rand() < mute_rate:
        x[2] = np.random.uniform(thres_low, thres_high)
    if np.random.rand() < mute_rate:
        x[3] = np.random.uniform(coef_low, coef_high)
    x[0] = xw
    return x
```

Acknowledgement

First of all, I would like to thank my parents for understanding and supporting me throughout my life. I always feel that I owe my family too much.

I wish to express my sincere thanks to Prof. Blais for his guidance during these two years. His advice has always been beneficial and he has enlightened me on both academic research and industrial experience.

I would like to thank my best friends, Jin Wang, Jialin Li, and Yolanda Zhang. Friendship is just an unexplainable understanding, and we have always been there for each other.

My special thanks go to Qi Li and Yuhui Gong for their help during these two years. Besides, I sincerely appreciate the mentorship by my previous supervisor, Mr. Kai Wang, who brought me into the world of quantitative trading.

To all who have helped me, I thank you so much.

And I wish for world peace.

References

[1] John Moody and Matthew Saffell. Learning to trade via direct reinforcement. *IEEE transactions on neural Networks*, 12(4):875–889, 2001.

[2] John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* MIT press, 1992.

[3] Arthur L Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229, 1959.

[4] Cade Metz. The rise of the artificially intelligent hedge fund, Jan 2016.

[5] Narasimhan Jegadeesh and Sheridan Titman. Returns to buying winners and selling losers: Implications for stock market efficiency. *The Journal of finance*, 48(1):65–91, 1993.

[6] Stefan Rampertshammer. An ornstein-uhlenbeck framework for pairs trading. *Preprint. Available at http://www. ms. unimelb. edu. au/publications/RampertshammerStefan. pdf*, 2007.

[7] Tim Leung and Xin Li. Optimal mean reversion trading with transaction costs and stop-loss exit. *International Journal of Theoretical and Applied Finance*, 18(03):1550020, 2015.

[8] John Ehlers. Predictive and successful indicators. *Technical Analysis of Stocks & Commodities*, Jan 2014.

[9] Yoav Freund and Llew Mason. The alternating decision tree learning algorithm. In *icml*, volume 99, pages 124–133, 1999.

[10] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[11] Kyoung-jae Kim and Ingoo Han. Genetic algorithms approach to feature discretization in artificial neural networks for the prediction of stock price index. *Expert systems with Applications*, 19(2):125–132, 2000.

[12] Eugene F Fama and Kenneth R French. Permanent and temporary components of stock prices. *Journal of political Economy*, 96(2):246–273, 1988.

[13] Ronald J Balvers and Yangru Wu. Momentum and mean reversion across national equity markets. *Journal of Empirical Finance*, 13(1):24–48, 2006.

[14] Alina F Serban. Combining mean reversion and momentum trading strategies in foreign exchange markets. *Journal of Banking & Finance*, 34(11):2720–2727, 2010.

[15] John Moody and Lizhong Wu. Optimization of trading systems and portfolios. In *Computational Intelligence for Financial Engineering (CIFEr), 1997., Proceedings of the IEEE/IAFE 1997*, pages 300–307. IEEE, 1997.

[16] Carl Gold. Fx trading via recurrent reinforcement learning. In *Computational Intelligence for Financial Engineering, 2003. Proceedings. 2003 IEEE International Conference on*, pages 363–370. IEEE, 2003.

[17] Michael AH Dempster and Vasco Leemans. An automated fx trading system using adaptive reinforcement learning. *Expert Systems with Applications*, 30(3):543–552, 2006.

[18] Jin Zhang and Dietmar Maringer. Using a genetic algorithm to improve recurrent reinforcement learning for equity trading. *Computational Economics*, 47(4):551–567, 2016.

[19] Günter Rudolph. Convergence analysis of canonical genetic algorithms. *IEEE transactions on neural networks*, 5(1):96–101, 1994.

[20] Dinabandhu Bhandari, CA Murthy, and Sankar K Pal. Genetic algorithm with elitist model and its convergence. *International Journal of Pattern Recognition and Artificial Intelligence*, 10(06):731–747, 1996.

[21] Stuart Russell, Peter Norvig, and Artificial Intelligence. A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25:27, 1995.