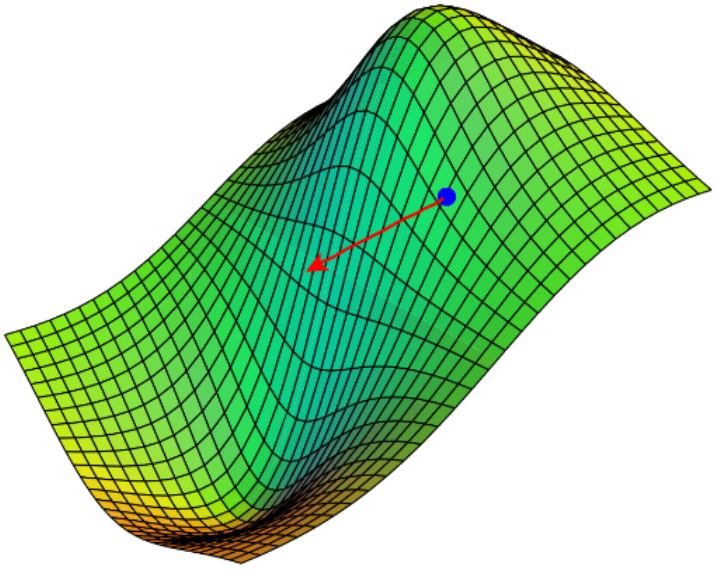
UNIVERSITÉ DE NANTES

IAE NANTES
ÉCONOMIE & MANAGEMENT

Quantitative Research on Market Predictability:
Machine Learning and the Efficient Market Hypothesis



Aurouet Lucas

Master EKAP
IAE Nantes
Under the supervision of Prof. O.DARNE 01/03/2021

Contents

1 Introduction **2**

2 Theory **3**

2.1 Statistics & Machine Learning 3

2.1.1 Artificial Neural Network 3

2.1.2 Networks & Backpropagation 6

2.2 Finance 9

3 Application **9**

4 Conclusion **9**

1 Introduction

Trading is futile. This is an oversimplification of the Efficient Market Hypothesis. According to Eugene Fama who formulated this hypothesis in 1970[1], prices at a certain date incorporate all the available and relevant information at that date. There are never overvalued nor undervalued assets, fundamental and/or technical analysis are therefore pointless as all the signals are already reflected by prices on the market. One cannot beat the market, is another way of expressing the EMH, which brings us to our initial statement; trading is futile. As trading basically consists in predicting an asset performance, under the EMH, trading is a wasted effort. It is quite hard to understand how, if the EMH holds, financial institutions have poured significant amounts of money into trying to outperform the market. Even less understandable is how some have succeeded. I think it is reasonable to define what 'outperforming the market' means. 'Outperforming the market' encompasses the achievement of greater returns than a portfolio containing every asset available on the market, with the same level of risk offered by such portfolio. Alternatively, it could mean achieving the same return, with less risk. Eugene Fama states that the only way to expect returns greater than the market is to take additional risks, or that you cannot achieve that same return on investment with a risk inferior to the market risk. We are now able to ask the following question: does the EMH holds with real life data ? Is trading futile ? Can you achieve greater returns than the market for a corresponding market risk ? Or is one constantly better off holding a large index like the S&P 500 for instance ? In my opinion, questioning the EMH is the most riveting topic in quantitative finance as it answers questions from a very theoretical point of view, typically what is observed in academics but also from a very practical point of view, the kind we observe on a trading floor. Econometrics is the science of combining these two aspects, using theoretical tools to shine lights on very down to earth problems. Therefore this work will try to reflect both dimensions. There are many ways to prove or disprove the EMH, but consistently beating the market is one of them (consistency is essential here). The objective here is to use data science to try and outperform the market. The results should definitely not be taken as an attempt to generate profits but rather as an exercise where the objective is to make use of quantitative techniques to elucidate a question. Machine Learning is a very powerful tool when a problem comes to identifying patterns and regularities in the data. As we will discuss later on, it is particularly adapted in our case. One fact often played down is that Machine Learning boils down to a list of algorithms applied in a particular order, and it is very easy to make mistakes when using automated techniques. We hope to have sufficient knowledge to question our own results, especially when working in quantitative finance. Although we are not, as mentioned above, attempting to concretely apply our findings, We think it is only natural to add some perspective when working with financial data. Finance is connected to the economy by many pipelines and misleading Artificial Intelligence can induce disastrous consequences. Artificial Intelligence is a very broad term and Machine Learning is one of its component, the later relies on the idea of being able to find patterns without explicitly defining these patterns, whereas Artificial Intelligence relies on the idea of writing a program capable of the same thought process as a human. They can sometimes be interchangeable and the frontier between the two is often blurry, here we are focusing on learning patterns from data, hence we will almost exclusively refer to Machine Learning as opposed to Artificial intelligencve. We will also refer to Statistical Learning, which could be seen as the gap between traditional statistical

[1]Efficient Capital Markets: A Review of Theory and Empirical Work - E.Fama

models and Machine Learning models. Statistical Learning solves data related problems by formulating them in a formal statistical problem which will then be solved using parts of Machine Learning algorithms. This gives us a large panel of tools to extract useful information from the data, coerce this information into a functional form that can then be exploited to infer the behavior of unseen data. We will first gaze over the Statistical and Machine Learning methods we will be using ranging from penalized regression to neural networks, we will then detail the financial side of the problem, going over technical analysis, fundamental analysis and the necessary financial mathematics. The second part is a step-by-step journal of the progress of the overall project and what modifications have been made to improve the data processing[2].

This project is coded in Python 3.8 and is associated with a Jupyter Notebook, available on GitHub. The integrity of the code is open source.

I often compare open source to science. To where science took this whole notion of developing ideas in the open and improving on other peoples' ideas and making it into what science is today and the incredible advances that we have had. - Linus Torvalds, developer of Linux and Git

2 Theory

2.1 Statistics & Machine Learning

2.1.1 Artificial Neural Network

Artificial Neural Networks (ANN or NN for short) are Machine Learning models designed to replicate the way the human brain processes information. It is heavily bi biology hence its name. In a human brain, the information passes through a network of neurons which can activate to pass the signal to the next ones. Neurons are connected by branches in which the information travel, starting from a particular point, these branches along with the neurons that activated create a path to an end point which outputs a decision depending on the problem to treat. The choice for each neuron to carry on the information is what determines the final output. ANN replicate this logic by creating a stream of branches and neurons inside which the information is processed to a particular output.

The Perceptron is a particular ANN which consists of only one neuron. It is easier to understand the logic behind ANN using a Perceptron, the results can then be generalized ton a network with more neurons.

We begin by giving the model inputs that represent the exogenous variables. Each input is multiplied by a weight and all weighted inputs are summed. The weighted sum goes through an activation function, that outputs 1 if the sum exceeds a certain threshold, 0 otherwise. For instance, we can think of inputs as values for technical indicators such as the RSI, the MACD and the stochastic oscillator and the output as 1 if we should "buy" and 0 if we should "sell". We take any data point and extract the values for the inputs and

[2]Although data processing often refers to the pre-screening of the data before feeding it into a model, here we refer to the entire process from the data gathering to the diagnosis of the results

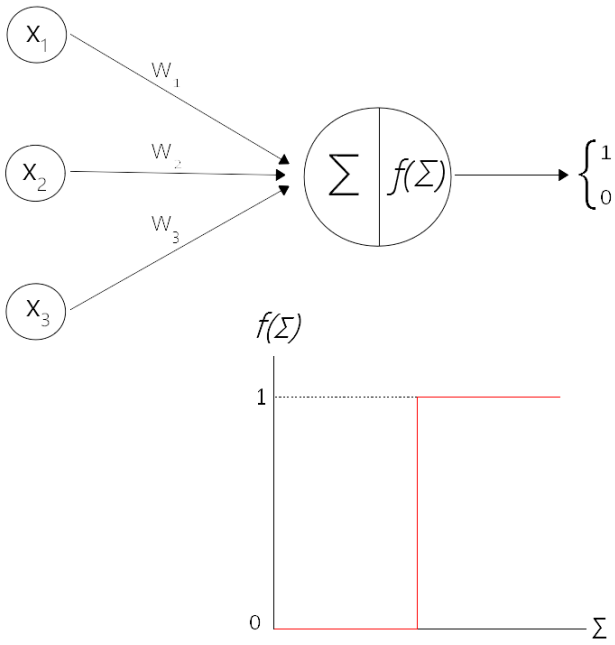


Figure 1: Diagram of a Perceptron

for the output, we compute the weighted sum and if that number exceeds the threshold, the Perceptron will output 1, or "buy". If the correct position was indeed buying, we can move to another data point. If the correct position was to sell, we update the weights and then we move on to the next data point. Weights are updated whenever the Perceptron outputs the wrong value,and kept the same whenever the Percetron outputs the correct values. As weights are the only element of the network that can be manipulated, by constantly updating them we will eventually come to the right combination of weights, which outputs the correct buy or sell output for every data points. The weights are then no longer updated and the network has finished training. For each new set of values for the technical indicators, the weights will be such that the weighted sum of inputs activate the neuron only when necessary, or when buying is the right call, and stay deactivated when the right position is to sell. We can calculate a loss function for every update, the loss will be 0 if the output corresponds the the actual value and different from 0 if the Perceptron fails. This loss function represents the error, or how accurate the ANN with respect to every weights combination there is. We can minimize this loss function by differentiating it with respect to each weights, which solves for the weights that outputs the minimal error, 0 ideally, but in reality the minimal error lies somewhere between 0 and 1. This closely relates to Ordinary Least Squares where we minimize the Mean Squared Error in order to find coefficients that minimize the squared errors. Although with OLS, one can find the solution to the differential equation analytically, it is often impossible with ANN as they do not have closed form solution. To find the correct weights we are required to use numerical methods, gradient descent in particular.

Gradient descent is a numerical algorithm to help find the local minimum of a function, assuming it is differentiable. Starting with random initial values for the weights we compute the error and the vector of partial derivatives with respect to every weights of the loss function, also called the gradient of the loss function. This vector of partial derivatives will point in the direction of steepest ascent where any increment in the weights

results in the largest possible step on the curve. If for a particular set of weights we land at a point a on the loss function, the gradient tells us in which direction the loss function increases the most or which new combination of weights will land on a' as far up as possible from a. As our objective is to find the minimum, we take the inverse of the gradient, which then leads us to the the direction of steepest descent. The inverse gradient tells us how to update weights so that the loss function decreases the most rapidly. For each data point we compute the error and the gradient at that point, take the inverse and increment the weights in the direction pointed by the inverse gradient. By always choosing the steepest descent we will eventually converge to a local minimum, given that the step size is adapted. Until now we where considering a step function[3] that switches from 0 to 1 depending on the value of Σ and a threshold as the activation. This kind of function is not differentiable, hence we are unable to compute partial derivatives. To compute the gradient of the loss function, which contains the activation function, we also need the gradient of the activation function. To use the gradient descent we must use more evolved function such as the sigmoïd function $f(x) = \frac{1}{1+e^{-x}}$, which is differentiable.

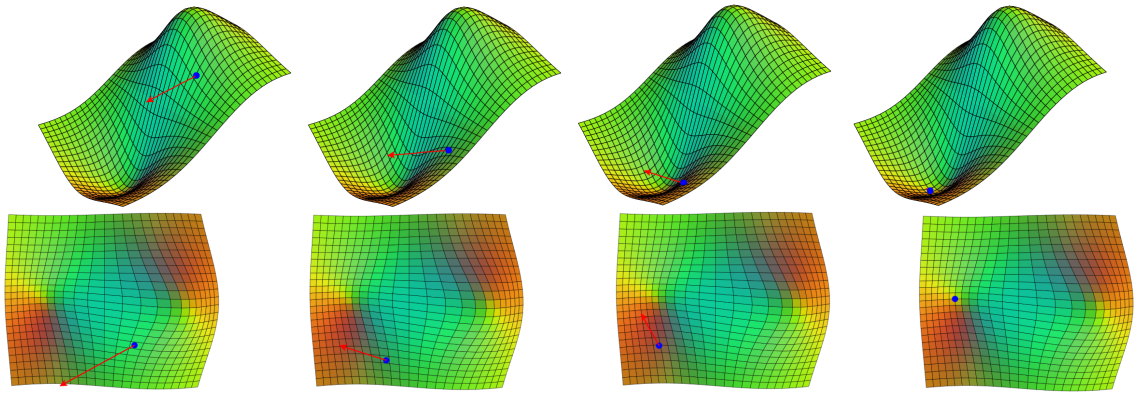


Figure 2: Gradient Descent

The Perceptron is limited to linearly separable problems, where it is possible to find an hyperplane in the inputs space that separates the two classes. In two dimensions, the hyperplane is a straight line, in three dimensions it would be a plane etc...The diagram below illustrates two different problems, the first one which is linearly separable and the second one which is not.

[3] Also sometimes called "Heaviside"

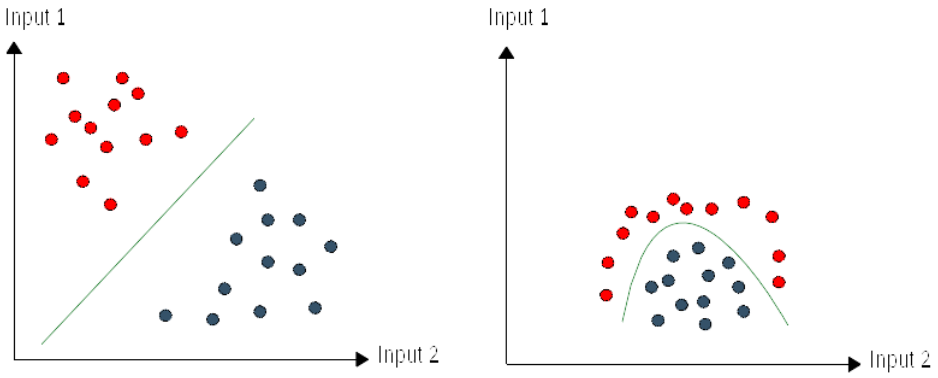


Figure 3: Linear separability

Once the neuron has been trained and we the finals weights are obtained we can calculate $g(x) = W^T x + w_0$ to trace a decision boundary. Every point falling on a particular side of the boundary will be associated with the corresponding class and any point falling on the other side of the frontier will be associated to the other class. As discussed earlier, the decision boundary can be a straight line, a plane or an hyperplane depending on the number of inputs. Because Peceptron is a linear classifier, we need to upgrade it in order to solve non linear problems such as the one presented on the diagram above.

2.1.2 Networks & Backpropagation

The use of Neural Networks allows us to find non-linear decision boundaries and separate classes that could not be indentified by a Perceptron. Neural Networks are a series of Perceptrons interconnected with each other. The consist in one entry layer, one output layer and intermediate "hidden layers" which size may vary. The size of a layer is determined by the number of neurons and the ultimate size of the network is determined by the size of each layer multiplied by the number of layers. The following example is a Neural Network of 1 entry layer of 3 neurons, one hidden layer of 3 neurons and 1 output layer of 1 neuron.

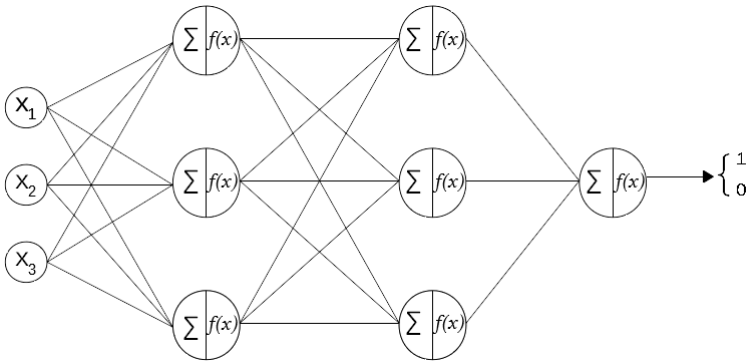


Figure 4: Neural Network with one hidden layer

The core principles remain the same, although one major modification has to be made. The gradient descent algorithm requires the ability to compute all partial derivatives with respect to each weights. Because the first weights are passed through an activation function before reaching the next layers, we need this activation function to be differentiable as we discussed earlier. In that case, we can apply the chain rule to compute partial derivatives from one end of the network to the other.

We associate each weight w to the two nodes it is connecting such that w^L_{jk} connects node j in the previous layer to node k in the current layer. The current layer is denoted as L and previous layers are denoted as $L-1$, $L-2$, ...,$L-M$. We also define the loss function as C_n, $n = 1, 2, ..., N$, N being the sample size and the activation function as g. The weighted sum of inputs arriving to node j is denoted as z^L_j and the output from node j is denoted as a^L_j. To use gradient descent we need to compute the gradient of the loss function to minimze. The gradient is therefore expressed as:

$$\nabla = \begin{bmatrix} \frac{\partial C_n}{\partial w^L_{j,k}} \\ ... \\ \frac{\partial C_n}{\partial w^{L-m}_{J,K}} \end{bmatrix} \tag{1}$$

We will show how to compute the gradient for one particular data point $n = 0$. We need to find the partial derivative of the loss function with respect to each weight in the network. For this example we will use the Mean Squared Error but other functions are appropriate to classification problems. For the output layer, we can draw the following diagrams:

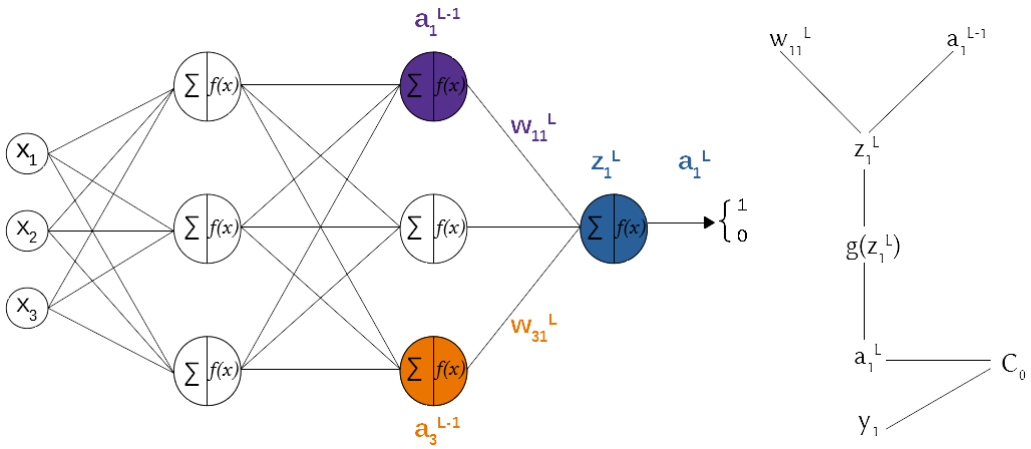


Figure 5: Output layer of a simple network

The diagram shows how the output from node 1 in layer $L-1$ is multiplied by a particular weight $w^L_{1,1}$ to obtain z^L_1. z^L_1 is passed throught the activation function $g(z)$ and produces the output for node 1 in layer L. Finding partial derivatives with respect to weights on the last hidden layer $L-1$ to the output layer L is straightforward because we can decompose the effect of weight $w^L_{1,1}$ with the chain rule:

$$\frac{\partial C_0}{\partial w_{1,1}^L} = \frac{\partial C_0}{\partial a_1^L} * \frac{\partial a_1^L}{\partial z_1^L} * \frac{\partial z_1^L}{\partial w_{1,1}^L} \tag{2}$$

The partial derivative of the loss function with respect to the weight $w_{1,1}^L$ is equal to the product of the partial derivative of the loss function with respect to the output of the last layer, the partial derivative of the output of the last layer with respect to the weighted sum z_1^L and the partial derivative of the weighted sum with respect to the weight $w_{1,1}^L$. because the loss function is a function of the output given by the network, which is a function of the weighted sum which in turn is a function of the weights, the partial derivative of the loss function is equal to the product of the partial derivatives of the function that compose it. We yet have to compute these derivatives.

- $\frac{\partial C_0}{\partial a_1^L}$, the loss function is equal to: $(a_j^L - y_j)^2$. Therefore its partial derivative is: $2(a_j^L - y_j)$.
- $\frac{\partial a_1^L}{\partial z_1^L}$, the output a_1^L is the result of the activation function evaluated at z_1^L. Therefore its partial derivative is equal to the partial derivative of the activation function, $g'(z)$
- $\frac{\partial z_1^L}{\partial w_{1,1}^L}$, z_1^L is the weighted sum of the previous output multiplied by the weight $w_{1,1}^L$, a slight change in the weight will cause z_1^L to change by the value of the previous output, a_1^{L-1}. The partial derivative of z_j^L with respect to the weight $w_{j,k}^L$ is equal to the previous output[4].

We were able to compute one element of the gradient vector:

$$\frac{\partial C_0}{\partial w_{1,1}^L} = 2(a_j^L - y_j) * g'(z_i^L) * a_1^{L-1} \tag{3}$$

This is only true for the partial derivatives with respect to weights on the last layer, because the loss function is a direct function of the output from the last layer. However, it is not a direct function of the outputs from previous layers. This is where backpropagation is used, to rewrite $\frac{\partial C_0}{\partial a_j^L - m}$, $m \neq 0$ such that C_0 becomes a function of $a_j^L - m$ using the chain rule. We want to solve:

$$\frac{\partial C_0}{\partial w_{j,k}^{L-1}} = \frac{\partial C_0}{\partial a_j^{L-1}} * \frac{\partial a_j^{L-1}}{\partial z_j^{L-1}} * \frac{\partial z_j^{L-1}}{\partial w_{j,k}^{L-1}} \tag{4}$$

$\frac{\partial a_j^{L-1}}{\partial z_j^{L-1}}$ and $\frac{\partial z_j^{L-1}}{\partial w_{j,k}^{L-1}}$ are derived exactly like if they were on the last layers as a_j^{L-1}, z_j^{L-1} and $w_{j,k}^{L-1}$ are direct functions of each other. We need to rewrite $\frac{\partial C_0}{\partial a_j^{L-1}}$ to be able to compute the gradient and identify the effect of weight $w_{j,k}^{L-1}$ on the loss function. Although the loss function is not a direct function of nodes that are not on the last

[4]We use indices j and k as this is true for every weight

layer, every node plays a role in the activation of all further neurons. Eventually, each neurons impacts the output neuron, which in turn is a variable on which the loss function directly depends. The idea of backpropagation is to take advantage of the structure of the network to reverse engineer the propagation of the activation of neurons to be able to compute partial derivative of the loss functions with respect to all weights in the network. We would like to remind the reader that with all the partial derivatives we can calculate the gradient and optimize the weights in order to minimize the loss function. The first step to reverse engineer the network is to find $\frac{\partial C_0}{\partial a_j^{L-1}}$ by applying the chain rule on that expression:

$$\frac{\partial C_0}{\partial a_j^{L-1}} = \sum \left[\frac{\partial C_0}{\partial a_j^L} * \frac{\partial a_j^L}{\partial z_j^L} * \frac{\partial z_j^L}{\partial a_j^{L-1}} \right] \tag{5}$$

We already demonstrated how to calculate $\frac{\partial C_0}{\partial a_j^L}$ and $\frac{\partial a_j^L}{\partial z_j^L}$. We will now focus on the new term:

$\frac{\partial z_j^L}{\partial a_j^{L-1}}$, the partial derivative of the weighted sum with respect to previous output is equal to the weight $w_{j,k}^L$ and $\frac{\partial C_0}{\partial a_j^{L-1}} = 2(a_j^L - y_i)g'(z)w_{j,k}^L$. Overall we were able to calculate the impact of the weights of the previous layer:

$$\frac{\partial C_0}{\partial w_{j,k}^{L-1}} = \underbrace{\frac{\partial C_0}{\partial a_j^{L-1}}}_{2(a_j^L - y_i)g'(z)w_{j,k}^L} * \frac{\partial a_j^{L-1}}{\partial z_j^{L-1}} * \frac{\partial z_j^{L-1}}{\partial w_{j,k}^{L-1}} \tag{6}$$

2.2 Finance

3 Application

4 Conclusion