UNIVERSIDAD POLITÉCNICA DE MADRID

**ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN**

MASTER THESIS WORK

MASTER OF SCIENCE IN SIGNAL THEORY AND COMMUNICATIONS
(Signal Processing and Machine Learning for Big Data)

DEEP REINFORCEMENT LEARNING APPLIED TO GENERATE TRADING SIGNALS FOR FINANCIAL MARKETS

BORJA GÓMEZ SOLÓRZANO

2019

MASTER THESIS WORK

Título:	Deep Reinforcement Learning applied to generate trading signals for financial markets
Autor:	D. Borja Gómez Solórzano
Tutor:	D. Eduardo López
Departamento:	Departamento de Señales, Sistemas y Radiofrecuencia

TRIBUNAL:

Presidente:	D.
Vocal:	D.
Secretario:	D.
Suplente:	D.

Fecha de lectura:

Calificación:

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN

MASTER THESIS WORK

MASTER OF SCIENCE IN SIGNAL THEORY AND COMMUNICATIONS
(Signal Processing and Machine Learning for Big Data)

DEEP REINFORCEMENT LEARNING APPLIED TO GENERATE TRADING SIGNALS FOR FINANCIAL MARKETS

BORJA GÓMEZ SOLÓRZANO

2019

Abstract

The goal of this master thesis work is to present an automated trading strategy over a single asset based on a reinforcement learning approach. The DL model will be combined with DRL to design a real-time trading system for financial asset trading. I employ Direct Reinforcement Learning (DRL) to solve it, the difference between DRL and the more common Q-Learning approach is the learning phase; the first tries to train the model optimizing the immediate rewards and the second the long term rewards. In the case of price series of financial asset DRL is more suitable, because of the high noise and the continuity in the signal. The target is to maximize a utility function, in this master thesis work I will introduce some of them, like the Sharpe ratio. The trader's action is done by a decision function, this function is the output of a neural network with a deep architecture, the input of the network is the asset price series and additional information such as other asset price series and indicators, this being necessary for filtering the trading actions over the main asset traded. In this implementation I will have to face, in the training phase, with some problems of the backpropagation algorithm like named vanishing gradient. Finally, I include graphics to show the execution of the algorithm over time, the most important are: The profit/loss, the trader's action and the evolution of the utility function.

Index items— Deep Learning, financial signal processing, Reinforcement Learning, Backpropagation Through Time, Direct Reinforcement Learning, Recurrent Neural Network, Sharpe ratio

Acknowledgements

I want to thank darwinex and specially the CEO’s Javier Colón and Juan Colón for giving me financial support to study this master and for letting me to combine it with my job in the company. And to my workmate Ali S. Khan for reviewing this thesis work.

NOMENCLATURE

$\mathcal{A}$	Set of actions
$\mathcal{S}$	Set of states
$\mathcal{R}_s^a$	Set of rewards. $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$
$\mathcal{P}_{ss'}^a$	Transition probabilities matrix with dimensions: $\lvert\mathcal{S}\rvert\lvert\mathcal{A}\rvert \times \lvert\mathcal{S}\rvert$
$\mathcal{Z}_{s'o}^a$	Probability of receive observation o in new state s'
$< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} >$	Markov Decission Process (MDP)
$< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \mathcal{O}, \mathcal{Z} >$	Partially Observable Markov Decission Process (POMDP)
$\pi(s\lvert a)$	Policy to follow from state s taking action a
θ	Model parameters set
$\mathcal{O}_t$	Set of observations until time t
$\mathcal{H}_t$	History until time t
$\tanh(x)$	Tangent function
F_t	Trader's action at time t
R_t	Rewards of trader in time interval $[t-1, t]$
U_t	Trader's utility function at time t
r_t	Price differences between times $t-1$ and t
$\mu(x)$	Average
$\sigma(x)$	Standard deviation
$\langle ., . \rangle$	Inner product
$\{...\}$	Discrete set
$[...]$	Array (ordered set)
$x \in \mathcal{X}$	x belongs to set $\mathcal{X}$
$f(a; b)$	Function f with variable a and parameter b
$\frac{df}{d\theta}$	f derivative with respect to θ
$\sum_{i=i_s}^{i_e}$	Discrete sum

List of Figures

Contents

Introduction and goals

Traditional trading algorithms

Market price making are based on the decisions of all market participants, mostly influenced on big players. There is evidence that liquidity providers have the ability to make market movements, so we can identify over long periods, different market regimes (or hidden states in decission processes theory); in time series analysis terms we say that the price series is non-stationary. What happens is that over the long term, an automated trading strategy tends to fail due to the difficulty to adapt to these different regimes.

These strategies are built using technical analysis techniques based on asset price indicators that removes the noise from the signal, like the moving average that captures trends

$$\mathrm{MA}(price, p) = \frac{1}{p}\sum_{i=1}^{p} price_{t-i} \tag{1}$$

or the RSI (Relative Strength Index) indicator that tries to capture reversals in the price series

$$U = \max\{0, price_t - price_{t-1}\} \tag{2}$$

$$D = \max\{0, price_{t-1} - price_t\} \tag{3}$$

$$\mathrm{RS} = \frac{\mathrm{MA}(U, n)}{\mathrm{MA}(D, n)} \tag{4}$$

$$\mathrm{RSI} = 100 - 100\frac{1}{1 + \mathrm{RS}} \tag{5}$$

where the value is bounded in the interval $[0, 100]$, high values means the market is over bought and the price is likely to goes down, and the opposite for low values.

These two classical trading techniques are very sensitive to overfitting, due to the fixed parameters selection, chosen by running different backtests over the history. For instance, the moving average feature is good enough to describe the trend but may suffer significant losses in a mean-reverting market. The principal drawbacks of this approach is the difficult to adapt to different market regimes and its poor generalization ability.

Reinforcement Learning

Trading is an unsupervised online decision taking problem, the actions are not optimal but "good" in an attempt to maximize profits. During the process the trader agent suffers some problems that must be taken into account in the model, for instance, the transaction costs due to market impact like commissions and spreads affect significantly the performance, so the trader has to avoid frequently changing the trading positions.

Reinforcement Learning encompasses adaptative learning techniques that have the capability to learn from the interactions with the environment (the market in our case) and to adapt to the changes in this environment (market regimes). In this process a trading policy is learned based on the trading history (the recurrent memory), and the most recent market conditions (the non-recurrent or standard memory). The reason for the recurrence is to solve the well known *temporal credit assignment* problem, which occurs because there are different action regimes (the trader's action distribution is non-stationary) so the time at which the actions were taken is important in order to make the new trading decission. Another problem comes from the structure of the market environment, like the impossibility of determining the current environmental state which is important in order to take the best decision, and the regime changes in the price series (is non-stationary series). Lastly, another problem comes from the architecture of some algorithms which treat the states and the actions in a discrete way, so to take the best action needs to save all the possible states and actions. This problem is known as *the curse of dimensionality.* We will try to propose a solution to all of them.

Deep Learning

Deep Learning is a bio-inspired emerging technique used to feature learning from big data, trying to simulate knowledge discovery mechanisms in the brain. These methods are well known in multiple fields like image categorization and speech recognition. Because of the high number of model parameters, this methods need to use lot of data for training in order to avoid bias. It is hence challenging to use this techniques in a highly dynamic and noisy environment like financial markets, this being the reason there exist few works involving Deep Learning in this field.

One of the most important steps in Reinforcement Learning is to do feature learning dynamically to take the best actions in order to mitigate the impact of noise and uncertainty and summarize the market conditions. Adding some layers to the deep hierarchical architecture of the network finds more complex relationship between the transformed features. As showed in [3], a robust feature representation is important in a dynamic decision environment, especially in financial markets that have high degree of uncertainty (noise in signal processing language), jumps (or market shocks in finance language) and high non-stationarity, due to all the decisions taken by the market participants. So a failure in the feature extraction part affects negatively the trading performance. Another improvement of these techniques with respect to technical indicators like the moving average explained before, is the generalization capability of neural networks over the pre-defined behaviour of the indicators.

Deep models are complex and have a lot of parameters and network architectures, the selection of the best is beyond the scope of this thesis. I will highlight the importance of the optimization algorithm in a dedicated chapter.

Goals

Summarizing, due to the difficulty of using Deep Neural Network models to detect price movements we are going to introduce in this work the Reinforcement Learning Agent, who analyzes the information from the Deep Learning model and process it to take automated trading decisions. The result is a Deep Reinforcement Learning model called the Single Trader Agent. In this work we'll introduce the model and we'll show the python code and the results obtained with different model parameters.

Chapter 1

Direct Reinforcement methods

We can separate reinforcement learning methods into three sets depending on the policy learning approach. The first are the called critic methods where the objective is to estimate a value function and the policy is implicit. These kind of methods are known as value function methods and are based on the well known Bellman equations; an example of critic methods is Q-Learning. The second group is the actor methods, where the estimation is done over the policy and there isn't any value function, an example is the policy gradient algorithm. The third and the last set of methods is the actor-critic. In actor-critic algorithms we can distinguish two steps. The first is a policy evaluation step, the role of the critic that evaluates the current policy in order to reduce the variance of the learning. The second is the policy improvement the actor that estimates the policy and can be done in multiple ways but the principal are epsilon-greedy and gradient descent ways.

Direct Reinforcement Learning (DRL) encompasses actor-based methods where the policy is learned directly from observation to action through a utility or pay-off function without learning a value function. The absence of a value function avoids the Bellman's *curse of dimensionality*, because there are no memory needs to save all the actions and states. We are going to enumerate some Direct Reinforcement based methods:

- Policy Gradient Algorithm with stochastic and non-recurrent policies, where the policy improvement is done through a gradient approach, and requires only a differentiable objective function with latent parameters.
- Recurrent Policy Gradient Algorithm, called Recurrent Reinforcement Learning (RRL). With deterministic recurrent policies and applications in finance (as in our case).
- Stochastic Direct Reinforcement (SDR). With more general, recurrent and stochastic policies.

The adding of a recurrence term in the two latest examples can solve recurrent, non-Markovian partially observable problems, and the direct learning approach solves the *curse of dimensionality.*

As marked in [3] in classical value function the Bellman's equations involves a term recoding the future discounted returns, but in trading, returns are observed in real-time. It is hence dangerous to make a prediction over future prices because of the noise in the price signal, so value function methods aren't adequate for dynamic online trading. Another drawbacks in the value based methods are the following, noted in [4]. In Q-Learning small changes in the value function can produce large changes in the policy. Price series contains large amounts of noise and nonstationarity, which could cause big problems using it as input of a value function method.

Chapter 2

Recurrent Reinforcement Learning

If the states aren't completely observables by the agent we have to distinguish between the environment states and the agent states $s^e, s^a \in \mathcal{S}$, the agent internal representation of the environment states are based on the observations $o \in \mathcal{O}$ and have their corresponding transition probabilities $\mathcal{Z}$. In the case of an MDP dynamic system $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} >$ both s^e, s^a are equal.

2.1 Recurrent neural network

Financial markets don't follow a Markov Decission Process (MDP). Supposing the market efficient hypothesis doesn't work, we are unable to observe, for every market agent, their consensus on current market condition. In this case we are talking about an Partially Observable Markov Decission Process (POMDP) $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \mathcal{O}, \mathcal{Z} >$. So, the agent has knowledge of the history and the pay-off (or rewards) structure of the trading, in terms of a performance function, but doesn't know anything about how the market is created (it's internal state). Because of this partial observability the trader has to include the previous actions taken in the model in order to take them into account to configure its new agent state representation. We'll call the performance function defined before U_t (in finance words is called utility function, here comes the U) with marginal utility depending on action a_t as $U_t(a_t) = U(a_t, o_t)$. These values are known after an action is taken, in our case when we are trading in financial markets for example, the closed profits are calculated when the new action is taken.

For the feature learning we are going to use a deep feature learning following the next equation

$$h_i = f(h_{i-1}, o_i; W), i = t, t-1, ..., t-m \tag{2.1}$$

and we define $g_t(W) = h_t$ where f is a deterministic function (activation function in neural network terms). In a neural network approach h are the hidden layers of the deep neural network model. We call the set of observations, actions and rewards $H_t = \{o_1, a_1, r_1, ..., o_t, a_t, r_t\}$ as the history. The observable set $\mathcal{O}_t$ is composed of the past prices and side data until time t, $o_t = [p_t^1, ..., p_t^k]$.

The action taken at time t follows the equation

$$a_t = f(g_t(W), a_{t-1}; W_a) \tag{2.2}$$

where f, again, is a deterministic function in the interval $[-1, 1]$. We call the set $\theta = \{W, b, W_a\}$ the parameters set of the neural network, where b is the bias term.

2.2 The algorithm

This description of the Recurrent Reinforcement Learning algorithm is for a trader that optimizes inmediate estimates of performance U_t for specific actions taken.

Algorithm 2.1 Recurrent Reinforcement

```
 1: Input: $\mathcal{O}_t$
 2: Output: $\pi_\theta$ policy.
 3: Initialize $\theta$ randomly
 4: for each episode do
 5:     for each time $t$ do
 6:         apply deep learning $g_t(o_t, \theta)$
 7:         take action $a_{t+1}$ from $f(g_t, a_t)$
 8:     end for
 9:     $u(a; \theta) = U_T(R_1, R_2, ...R_T)$
10:     $\pi_\theta$: improved from a local approximation of the $u$ gradient
11: end for
12: return $u, \pi_\theta$
```

where π is the policy, learned directly used the deep neural network, and θ represents the parameters (weights) of the network. $\frac{du}{d\theta}$ is the local performance estimate of the utility function.

Chapter 3

The single trader agent model

Our trading model is based on a trader who trades only one asset and keeps opened only one operation at any time. The trader uses some side information (indicators) for better action decision. The goal of the learning algorithm is to find parameters θ for the best policy according to an agent's expected utility. The entirety of this chapter is based on the paper [1], we are going to follow here the same notation.

3.1 The trader's action

Consider an agent that takes action F_t given by its policy function $F \in [-1, 1]$ is the trader action (the percentage of asset bought), we assume in this first approach presented that the asset is infinitely divisible, we use for this the tanh function as the trader's decision function. Following [5] as reference paper we are going to separate the two memories in the model

$$F_t = F(O_{t-1}^{(n)}, F_{t-1}^{(m)}; \theta_t) \tag{3.1}$$

where $\mathcal{F}_{t-1}^{(m)} = \{F_{t-1}, F_{t-2}, ..., F_{t-m}\}$ are the most m recent actions. We call this memory recurrent of order m, and the external information set $\mathcal{O}_{t-1}^{(n)} = \{o_{t-1}, o_{t-2}, ..., o_{t-n}\}$ the non recurrent or the standard memory of length n, is all the information available at time $t-1$ including the prices and external information history, the combination of action and information sets is the observed history $\mathcal{H}_t = \{\mathcal{F}_t, \mathcal{O}_t\}$. The authors in [5] call this model a dynamic system of order (m, n), because they want to emphasize the importance of separating the contribution of the standard and the recurrent memories to the model. The $\theta = \{W, W_a\}$ model parameters reflects this separation between $F_{t-1}^{(m)}$ and $\mathcal{O}_{t-1}^{(n)}$, where W are the weights associated with the standard memory and W_a are the weights associated with the recurrent memory.

3.2 Performance function: Sharpe ratio

We are going to introduce the trader rewards and the utility function as in [1].

The formula for the closed rewards is

$$R_t = F_{t-1} r_t - c|F_t - F_{t-1}| \tag{3.2}$$

where c is the transaction cost per order (including spreads and volume commissions) and r_t are the price differences between times t and $t-1$ multiplied by F_{t-1} because we are calculating closed rewards. As shown in [1], the latest part of the trader returns formula 3.2 can discourage the agent from big changes between trading decisions and to avoid huge transaction costs. This is another important aspect of the recurrence, to minimize the trading cost impact discovering better policies.

We define $U_T = U(R_1, R_2, ..., R_T)$ as the utility function, which is path dependent. It is a measure of the improvements of the trader's actions based on the rewards and is the one that we want to maximize. In our case we choose as utility function the Sharpe ratio:

$$S = \frac{\mu(R_t)}{\sigma(R_t)} \tag{3.3}$$

where μ is the average and σ is the standard deviation, which is a risk-adjusted measure of the strategy, the equivalent in finance to signal to noise ratio in signal processing. Other performance ratios shown in [2] are the downside deviation ratio $\frac{\mu(R_t)}{DownsideDev(R_t)}$ and the sterling ratio $\frac{\mu(R_t)}{DrawDown(R_t)}$. Because of the use of these risk based performance functions as utility functions we can call this reinforcement Risk-Averse, meaning we want to maximize the earnings while minimizing the risk taken, as we can see in the formula 3.3.

3.3 The optimization problem

Following the nomenclature in [3]

$$\begin{aligned} \underset{\theta}{\text{maximize}} \quad & U_T(R_1, ..., R_T) \\ \text{subject to} \quad & R_t = F_{t-1} - c|F_t - F_{t-1}| \\ & F_t = f(< W, g_t > + b + W_a F_{t-1}) \\ & g_t = \varphi(o_t) \end{aligned} \tag{3.4}$$

where $\theta = \{W, b, W_a\}$ are the weights of the deep feature learning, the bias and the weight of the action before contribution, and g is the deep learning function. The problem is not convex due to non-linerarity, so there is no convergence guarantee.

The learning algorithm is based on the gradient ascent with updating formula as in [1]:

$$\frac{dU_T(\theta)}{d\theta} = \frac{dU_T}{dR_T}\frac{dR_T}{d\theta} = \sum_{t=1}^{T}\frac{dU_T}{dR_t}\frac{dR_t}{d\theta} = \sum_{t=1}^{T}\frac{dU_T}{dR_t}\left\{\frac{dR_t}{dF_t}\frac{dF_t}{d\theta} + \frac{dR_t}{dF_{t-1}}\frac{dF_{t-1}}{d\theta}\right\} \tag{3.5}$$

$$\frac{dR_t}{dF_t} = \frac{d}{dF_t}(F_{t-1}r_t - \delta|F_t - F_{t-1}|) = \frac{d}{dF_t}(-\delta|F_t - F_{t-1}|) = -\delta sign(F_t - F_{t-1}) \tag{3.6}$$

$$\frac{dR_t}{dF_{t-1}} = \frac{d}{dF_{t-1}}(F_{t-1}r_t - \delta|F_t - F_{t-1}|) = r_t + \delta sign(F_t - F_{t-1}) \tag{3.7}$$

where the last part of the equation 3.5 is derived used the chain rule, because $R_t = f(F_t, F_{t-1})$. The formula is path dependent because the derivatives $\frac{dF_t}{d\theta}$ depends on the previous trades. Here we can see the recurrence of the trader's action. The reason to separate the section before the previous actions from the observations is clear here, because assuming that the set $\mathcal{O}_{t-1}^{(n)}$ has no dependency on θ, if we include the previous actions set $\mathcal{A}_{t-1}^{(m)}$ within the observations, then we are ignoring the dependence on θ of F_{t-1} and then, the recurrent term $\frac{dF_{t-1}}{d\theta}$ is zero. As remarked in [5], this m order recurrence is expressed in the equation 3.5 where, in our case, $m = 1$. The information of this gradients is propagated forward in time, the gradient recursions are causal, and the model is non-Markovian because it is dependent on past decisions. This influence on past actions tries to solve the *temporal credit assignment* problem.

The optimization of the recurrent network weights is done by an algorithm called Back Propagation Through Time (BPTT). It could be possible to implement an online training using stochastic gradient as in [2], but this is beyond the scope of this thesis.

3.4 The deep neural network architecture

We include in this section the deep model implemented in [3], a complex neural network with deep and recurrent structures. We consider as an example the following first order recurrent model with one hidden layer. We can also extend it to second order, third order, etc. depending of the number of terms F_{t-i}, and also add more layers easily in the code. The neural network is dense, meaning that all the neurons between two consecutive layers are fully connected.

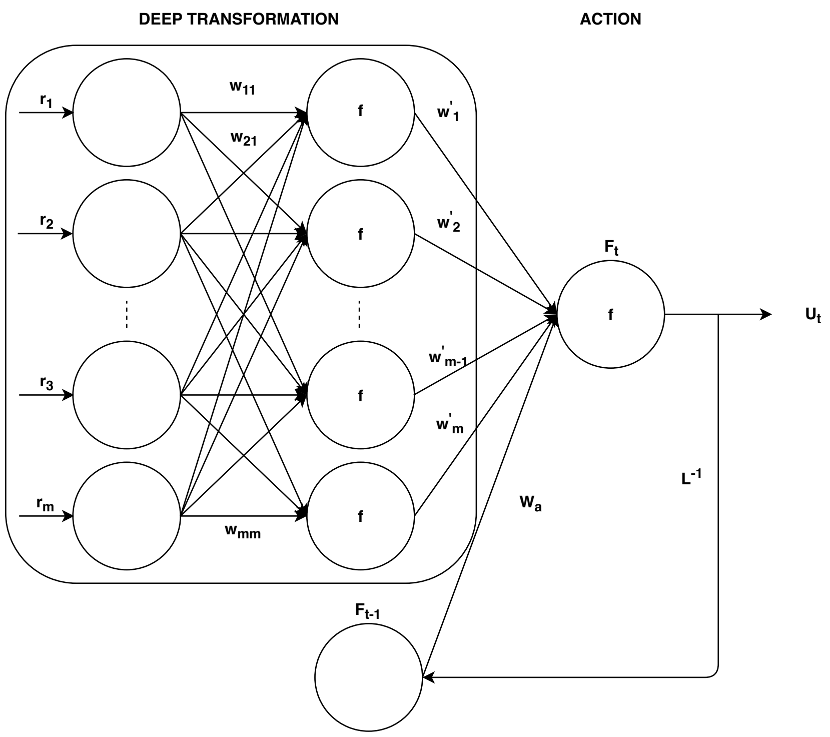


Figure 3.1: Neural network architecture

the equations for each one of the layers are

$$F_t = f(\sum_{i=1}^{m} w_i' h_i + W_a F_{t-1} + b') \tag{3.8}$$

$$h_i = f(\sum_{j=1}^{m} w_{ji} r_{t-i} + b) \tag{3.9}$$

where f is the activation function. Note that all the hidden layers in the neural network have the same number of neurons, because is a denoising model.

Note: The bias term is not included in the graphic for simplicity.

3.5 Side information

In order to improve the performance of the actions taken by the trader agent we will need to incorporate additional data independent of the prices; we call this data side data. In terms of implementation into the neural network we need to assign each block to a block of weights.

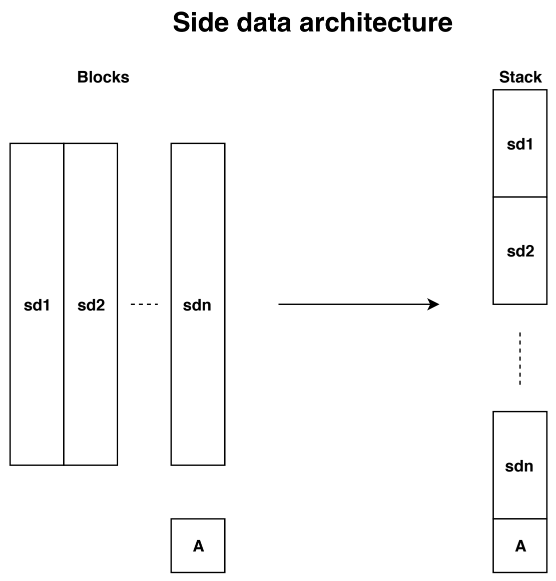


Figure 3.2: Side data architecture

So the formula 3.9 becomes

$$h_i = f(\sum_{j=1}^{m_1+..+m_n} w_{ji} r_{t-i} + b) \tag{3.10}$$

where m_i is cardinality of the block i and $w_{ji}, j \in [m_{k-1}, m_k]$ are the weights associated with it.

Chapter 4

Optimization drawbacks

The closed analytical manner to solve the optimization problem raises certain when it comes to computing recursively the gradient $\frac{dF_t}{d\theta}$, because the other derivatives in the formula 3.5 are only known once the utility function U_t is known, which is very dificult because it depends on the trader's action formula F. A simpler approach is to use the algorithm called Back Propagation Through Time (BPTT).

In the Recurrent Deep Neural Network the recurrence comes from F_{t-1} which is used as input to calculate the new output F_t, each block of the model consists in the deep structure plus the action and there are as many as times t, as represented in figure 3.1. As noticed in [3], once the model is unfolded, is the same as a Back Propagation method used for training neural networks, the time reference in the name is used following conventions observed in recurrent networks.

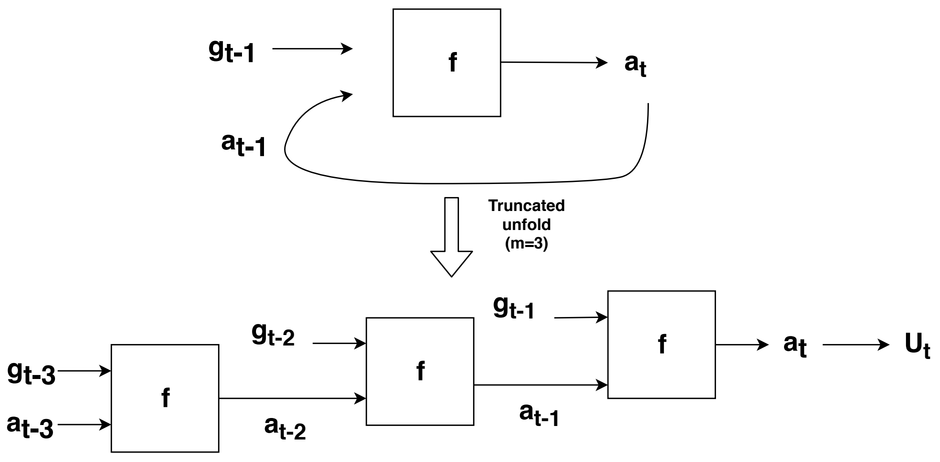


Figure 4.1: Back Propagation Through Time unfold

where g is the deep feature learning. Following the reference paper [3] we run the back propagation algorithm from the whole unfolded structure and, in each block t we run the back propagation algorithm in g from the output a_t, so we distinguish the two.

As we can see, the model has a deep timed-based unfolding structure because it repeats multiple times the blocks. As marked in [4], during the back propagation the derivative associated with the recurrent action suffers the effect of multiplying by its weight, which are the same for all the blocks; m times. Consequently the magnitude of this transition weight has a large impact on the learning process. If this weight is small the gradient contribution is small and tends to diminish. On the other hand if it is large, then the contribution of the gradient is large and tends to explode.

Solutions to this gradient problem are:

- Evolutionary algorithms, for instance particle swarm optimization helps avoid getting stuck in a local minimum using emerging intelligence. Each swarm particle tries to optimize locally and then with the information provided by other swarms tries to reach a global optimum solution.

- Other more complex architectures like LSTM networks instead our dense model [4] have the ability to "forget" past results. The explanation of these techniques is beyond the scope of this thesis.

- Another way to solve the problem is to change the activation function of the trader's action, because tanh function squashes the space in the small region $[-1, 1]$, so gradient of high values doesn't change too much and gets stuck, but this option is unfeasible, because this trader's action interval is bounded by definition (is a buy or sell percentage).

Chapter 5

The python code

The model was implemented using tensorflow, due to the flexibility of including multiple network architectures without needing to calculate all the derivatives for calculating the gradient of the utility function. The code is available on `https://github.com/BorjaGomezSolorzano/deep-trader`, including the data and some interactive graphics.

5.1 Price feed

This script contains the code for loading data from csv files with a header composed of date, open price in the timeframe (in our case daily), highest price in the day, the lowest price, the close price and the volume executed in this day. Here we generate the price differences for calculating the rewards and the input of the model, i.e., the price series and the side data. The method called flat is used to reshape the columns of the csv (the side data) to a one dimensional array, to feed the neural network.

```python
from model import *

def flat(x1):
    l1 = n_layers[0]
    l = l1 * n_features
    x = np.zeros((1, l))
    for k in range(n_features):
        for j in range(l1):
            x[0][k * l1 + j] = x1[j][k]

    return x

def process():
    df = pd.read_csv(instrument_filename, skiprows=1)
    df['prices_diff'] = df.iloc[:,instrument_idx].diff(periods=1)
    df['prices_diff'] = df['prices_diff'].shift(-1)
    df = df[pd.notnull(df['prices_diff'])]

    df = df.tail(last_n_values)

    dataset = df.values

    dates = dataset[:, 0]
    instrument = np.copy(dataset[:, instrument_idx])

    returns_idx = dataset.shape[1] - 1
    X_aux, y_aux = [], []
    for i in range(len(dataset)):
        X_aux.append(dataset[i, features_idx])
        y_aux.append(dataset[i, returns_idx])

    X = np.array(X_aux, dtype=float_type_np)
    y = np.array(y_aux, dtype=float_type_np)

    return X, y, dates, instrument
```

5.2 Weights and biases

The weights of the deep learning (plus the action taken before) are the same for all the blocks that compound the recurrent structure for each time t. The first weight corresponds to the input and have dimension $[n+1, 1]$ and the biases has dimension 1 in the case of one layer, and is the only weight. If the network has more than one hidden layer, then all the weights except the latest has dimensions $[n, n]$ (and the biases n), and the latest has dimensions $[n+1, 1]$ (and the biases 1). The $+1$ corresponds to the extra dimension of the action taken.

```
from model import *

def weights_and_biases():
    Ws = []
    bs = []

    l = len(n_layers)

    for i in range(1, l):
        dim = n_layers[i - 1] * n_features
        Ws.append(tf.Variable(tf.random_uniform([dim, dim], 0, 1)))
        bs.append(tf.Variable(tf.zeros([dim])))
    dim = n_layers[l - 1] * n_features + 1
    Ws.append(tf.Variable(tf.random_uniform([dim, 1], 0, 1)))
    bs.append(tf.Variable(tf.zeros([1])))

    return Ws, bs
```

5.3 Trader's action

Once we have the weights, the action taken is the output of a neural network of n hidden layers and the latest layer (which has as output the new action) is augmented with the information of the previous action. All the layers have the same activation function tanh. This is the block structure that is repeated each time t.

```
from model import *

def action(input_standard, action, Ws, bs):
    l = len(n_layers)
    layer = input_standard
    for i in range(1, l):
        layer = constants.f(tf.add(tf.matmul(layer, Ws[i - 1]), bs[i - 1]))

    layer_augmented = tf.concat((layer, action), axis=1)
    a = constants.f(tf.add(tf.matmul(layer_augmented, Ws[l - 1]), bs[l - 1]))

    return a
```

5.4 Rewards and utility function

The rewards (or profits) are computed as the output of the deep model (the action) times the daily prices difference (because the trader acts when the days starts), minus the contribution of the commission relative to the new volume invested in percentage and in absolute value. The rewards are in the currency associated with the asset we trade, for instance, if the asset is SPX500, the rewards are in USD, because the value of the index is in USD. Note that the commission contribution is different as in equation 3.2. This is because I take commission as a percentage of the volume traded by order, so I need to multiply it by the price in order to obtain its value in the same currency as the profit. As we said before, the function driving the optimization process is the Sharpe ratio.

```
from model import *

def reward_tf(u, action_t, action_t1, price_t):
    return (u * action_t1 - c * price_t * tf.abs(action_t - action_t1))*multiplier

def reward_np(u, action_t, action_t1, price_t):
    return (u * action_t1 - c * price_t * np.abs(action_t - action_t1))*multiplier

def sharpe(returns):
    mu = 0
    for i in range(0, window_size):
        mu += returns[i]

    mu /= float(window_size)

    sigma = 0
    for i in range(0, window_size):
        sigma += (mu - returns[i]) * (mu - returns[i])
    sigma = (sigma / float(window_size)) ** (0.5)

    return 0 if sigma == 0 else mu / sigma


def utility(returns):
    return sharpe(returns)
```

5.5 Recurrent action model

We define here the placeholders to feed the model with the price inputs and the previous action. We need to calculate the Sharpe ratio, based on the actions taken, in a time window. We want to maximize it using a gradient acceleration based algorithm called Adam, for more information about this algo and its convergence see the reference [6].

```
from model import *

def place_holders():
    input_phs = []
    output_phs = []
    prices_phs = []
    for i in range(window_size):
        dim = n_layers[0] * n_features
        input_phs.append(tf.placeholder(float_type_tf, shape=[1, dim]))
        output_phs.append(tf.placeholder(float_type_tf, shape=()))
        prices_phs.append(tf.placeholder(float_type_tf, shape=()))

    action_ph = tf.placeholder(constants.float_type_tf, shape=(1, 1))
    return input_phs, output_phs, action_ph, prices_phs

def recurrent_model(Ws, bs, input_ph, output_phs, action_ph, prices_phs):
    rewards_train = []
    a_t1 = action_ph
    for t in range(window_size):
        a_t = action(input_ph[t], a_t1, Ws, bs)
        r_t = reward_tf(output_phs[t], a_t[0][0], a_t1[0][0], prices_phs[t])
        rewards_train.append(r_t)
        a_t1 = a_t

    u = utility(rewards_train)
    optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(-u)

    return a_t, u, optimizer
```

5.6 Online execution

Finally we execute all the steps before to initialize the model. Another pre-step is to scale the inputs to help the optimization algorithm to converge. We use the scikit learn library MinMaxScaler to normalize the data to $[-1, 1]$ range, we use this range because the action taken, which is part of the model, is also in these bounds. The variables (the neural network weights) are initialized for each online optimization. For the training phase, before feeding the placeholders we need to flatten the side data using the flat function seen before. For each epoch we run the Adam optimizer and obtain the latest action we use to obtain the reward in the test phase.

```
from model import *

def execute(X, y, dates, prices):

    tf.set_random_seed(1)

    Ws, bs = weights_and_biases()

    input_phs, output_phs, action_ph, prices_phs = place_holders()

    last_a, u, opt = recurrent_model(Ws, bs, input_phs, output_phs, action_ph,
                                      prices_phs)

    scaler = MinMaxScaler(feature_range=(-1, 1))
```

```
accum_rewards = 0
a_t1 = np.zeros((1,1))
actions_returned = []
simple_rewards = []
dates_o = []
instrument_o = []
rew_epochs = [0 for _ in range(epochs)]

init = tf.initialize_all_variables()
with tf.Session() as sess:

    for j in range(n_layers[0], n_layers[0] + n_actions):

        sess.run(init)

        x_dic = {}
        for index in range(window_size):
            i = index + j
            x_i = np.copy(X[((i+1)-n_layers[0]):(i+1)])
            x_dic[index] = scaler.fit_transform(x_i)

        #Train
        for ep in range(epochs):
            feed_dict = {action_ph: np.zeros((1,1))}
            for index in range(window_size):
                i = index + j
                x = flat(x_dic[index])
                feed_dict[input_phs[index]] = x
                feed_dict[output_phs[index]] = y[i]
                feed_dict[prices_phs[index]] = prices[i]

            u_value, a_t, _ = sess.run([u, last_a, opt], feed_dict=feed_dict)
            rew_epochs[ep] += u_value

        i+=1

        # Test

        dates_o.append(dates[i])
        instrument_o.append(prices[i])

        actions_returned.append(a_t[0][0])
```

```
            rew = reward_np(y[i], a_t[0][0], a_t1[0][0], prices[i])
            accum_rewards += rew
            a_t1 = a_t

            simple_rewards.append(rew)

            print('iteration', str(i),
                  ', action predicted: ', str(a_t[0][0]),
                  ', reward: ', str(rew),
                  ', accumulated reward: ', str(accum_rewards))

    for k in range(epochs):
        rew_epochs[k] /= epochs

    return simple_rewards, actions_returned, dates_o, instrument_o, rew_epochs
```

5.7 Model parameters

As defined in the YAML configuration file called "config.yaml", the model parameters are:

instrument: XAUUSD. Ticker for the instrument

features_idx: [1,2,3]. Column indexes of the features, has to contain the price series (and side data as optional). 2 and 3 corresponds to the higher and the lower prices of the timeframe (is an interval of time: daily, 4 hours, ...).

instrument_idx: 1. Column index of the price series. Is fixed for the csv files given.

c: 0.000025. Commissions as percentage of the volume sent to the market.

learning_rate: 0.001. Learning rate for the model's optimization algorithm, is the multipliyer for the gradient increment contribution. High values speed up the convergence but tends to fail finding a maximum, low values slow down the algorithm.

epochs: 100. Number of learning epochs in the model.

n_layers: [30,30]. Number of hidden layers in the model as a list [first, second,...]. Note: Has to be the same number in each layer, because as seen in section 3.4 it is a denoising model.

window_size: 21. The number of points taken to calculate the utility function (Sharpe ratio) value.

n_actions: 500. Number of actions taken by the trader (the number of result points).

last_n_values: 1000. Run the code over the last n values of the csv.

multiplier: 100. This is the size of the asset we want to buy, and depends on the instrument traded. In the simulation the trader buys a percentage of this value.

Chapter 6

Experiments

First we are going to present the problem for the experiment. Our trader is an agent based system limited to a single asset. It is a Deep Recurrent Reinforcement Learning model with multiple hidden layers and neurons, and the utility function selected is the Sharpe ratio. Experiments are done with three different assets. The first is Gold (XAUUSD ticker), second is the SPX500 index (based on the most important 500's companies in the United States) and the third is the EURUSD exchange rate. All of them have been tested over a significant period of time, in a daily timeframe. We tested also some commissions values in order to see the effect on the performance and plot the results. Here we'll show graphically the dependence of trader's action with respect to the commission. Another market costs as slippage, the difference between the last price seen and the execution price, are included as commissions too. The higher the commission, the lower the trading decisions, as we'll see in the following box-plots. As side data I have used the higher and the lower values of the price over a day. For neural network convergence we plot the graphic with the average of Sharpe ratio per epoch vs the number of epochs, and for trading results we consider as performance metric the Sharpe ratio (the utility function we want to maximize), both for trading and testing data. In order to compare the results I have used the python command *tf.set_random_seed(1)* to fix the random initialization of the model variables. All the graphics showed here are interactive and are located in the public repository mentioned in chapter 5, for a better analysis it is recommended to go there.

Lastly, we are going to define first the fixed parameters for the experiments:

- The learning rate for the optimizer is 0.001.
- The number of epochs is 100 in all the examples.
- The size of the window for the Sharpe ratio calculations is 21 trading days (1 natural month more or less).
- The number of trading decisions is 800, starting from the latest 1000 value. Looking

at the price series, in this interval we can see multiple regimes for good testing.

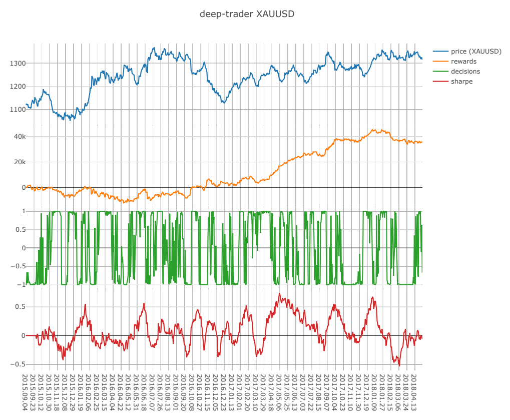
And the model variables are:

- The instrument: EURUSD, SPX500, XAUUSD.
- The commissions: charged, and 5 and 10 times the charged commissions.
- The features: Only the price, or including the max and the min value of the day.
- The number of hidden layers: One with 30 neurons or two with 30 neurons each one.

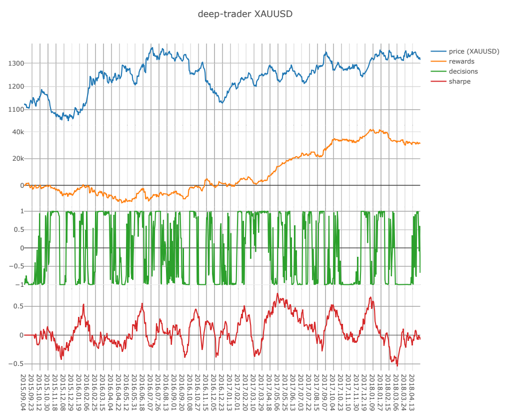
The other parameters are associated with the instrument.

6.1 XAUUSD

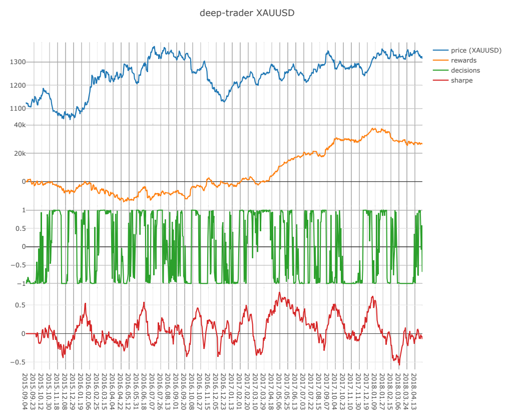
The graphics containing the four lines show four evolutions: the price of XAUUSD, the rewards (the profit expressed in USD), the decissions taken by the trader, and lastly, the Sharpe ratio. In the price series we can distinguish different zones, trends and lateral (with no trend), the results seems better on trending zones than lateral, in lateral zones the trader changes frequently the decision. Another thing to note is the presence of volatility in the evolution of the trader action, due to the no overfitting. The two following graphs have been done multiplying the commission parameters by a factor of 5 and 10 respectively. We can see the impact in the rewards evolution, an increment in the commissions conditions also the trader decissions. We can't appreciate here this impact, so for a better analysis the reader can go to the public github repository to see the differences between the decision evolution. In the latest graphic is clear the commissions impact on the decisions. I plotted here three box-plot graphics with the jumps between two consecutive actions in absolute value, as we can see the increment in the commissions produce small changes in the trading decisions, to see the values of the box plot is preferable to go to the interactive graphics too.



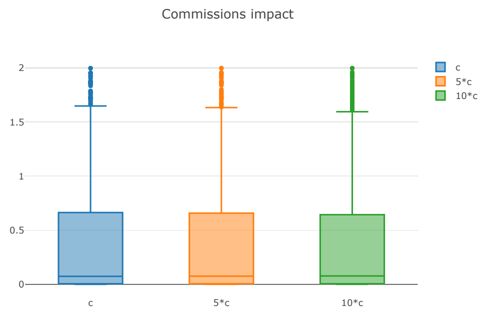
(a) One layer with 30 neurons and commissions c



(b) One layer with 30 neurons and commissions $5 * c$



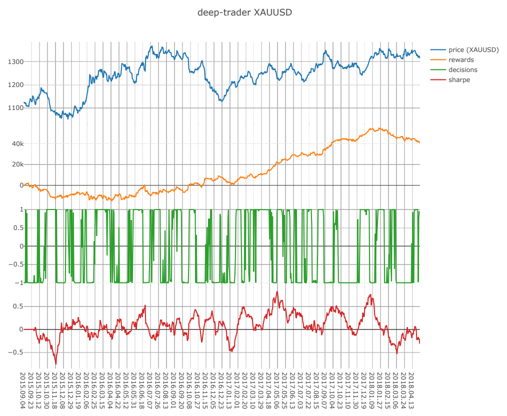
(c) One layer with 30 neurons and commissions $10 * c$



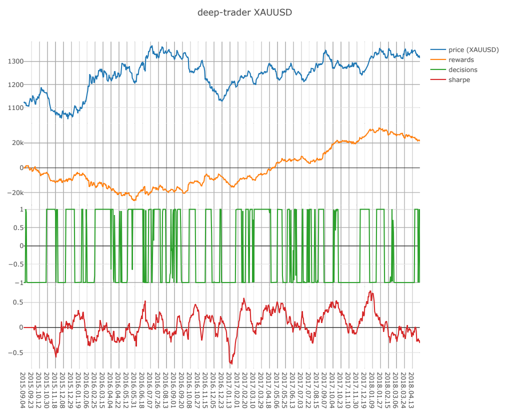
(d) Commissions impact

Figure 6.1: XAUUSD commission analysis

In the case of adding side data we can see an improvement with respect to the simpler case with only the price feature, the results are more stable looking at the variation of the Sharpe ratio. In the case of two hidden layers the results are worse than in the simpler case.



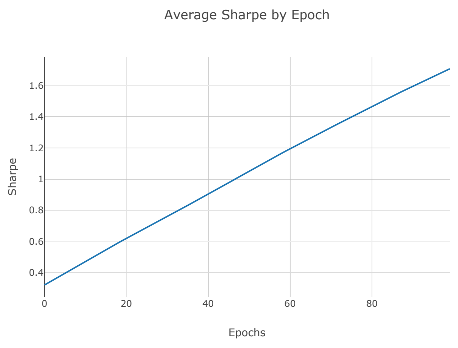
(a) One layer with 30 neurons commissions c and side data



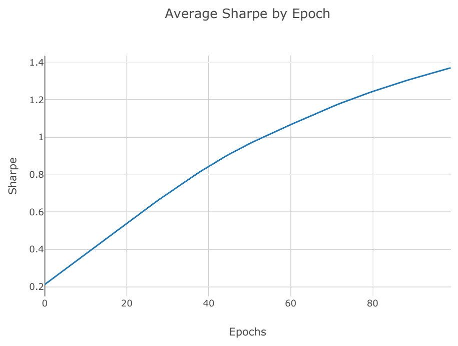
(b) Two layers with 30 and 30 neurons each one and commissions c

Figure 6.2: XAUUSD other models

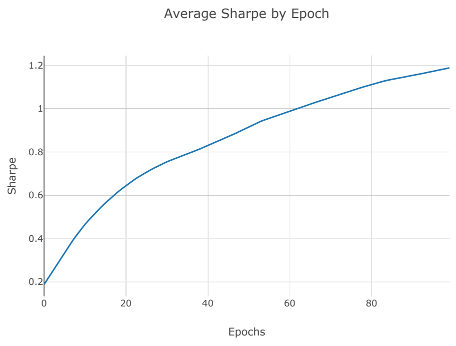
The graphics we are going to present are relative to the evolution in average of the reward for each epoch. It is significant the diference between the Sharpe ratio in the training phase compared to this testing phase. We can see that in the one layer case, with only 100 epochs the model do not converge. The problem is that if we increase enough the number of epochs we can overfit the model. Including side data we can achieve a faster convergence because we are using more data for training the same model parameters. In the two layers example we can see a convergence improvement too.



(a) XAUUSD reward vs epochs 30 neurons



(b) XAUUSD reward vs epochs side data

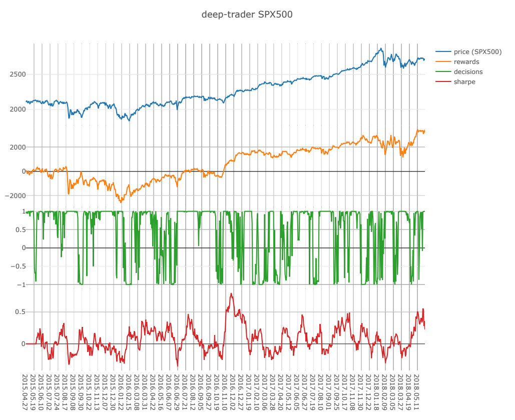


(c) XAUUSD reward vs epochs 30, 30 neurons

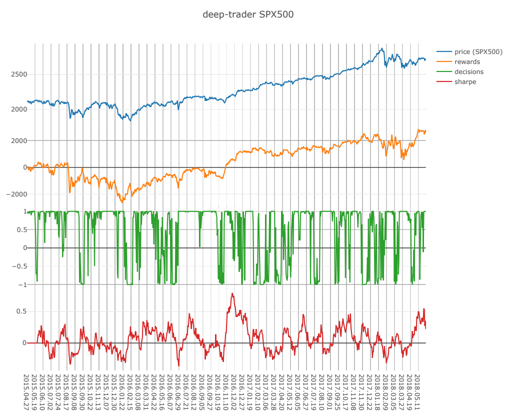
Figure 6.3: XAUUSD reward vs epochs

6.2 SPX500

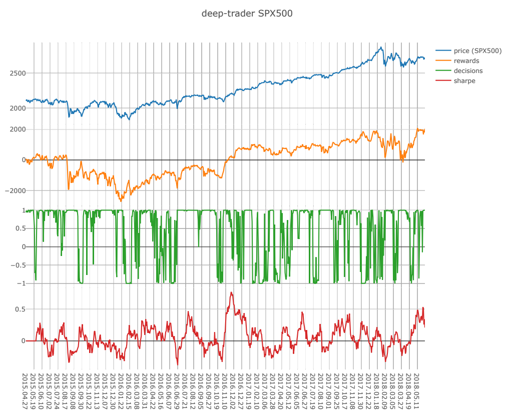
The explanations of the XAUUSD example are valid here.



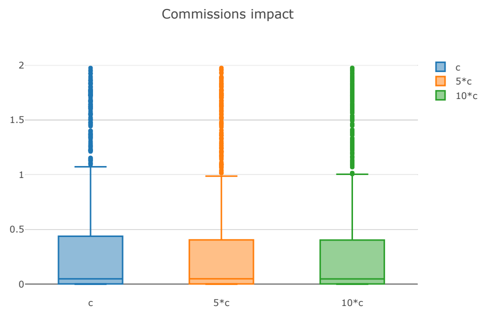
(a) One layer with 30 neurons and commissions c



(b) One layer with 30 neurons and commissions $5 * c$



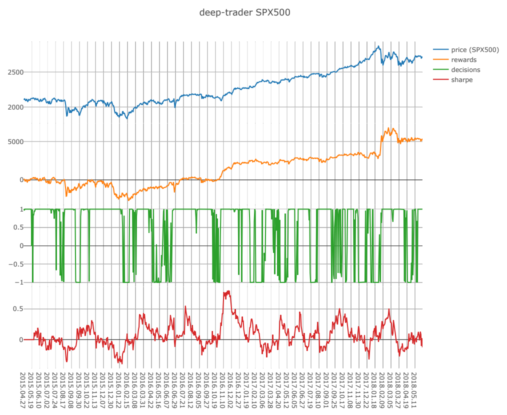
(c) One layer with 30 neurons and commissions $10 * c$



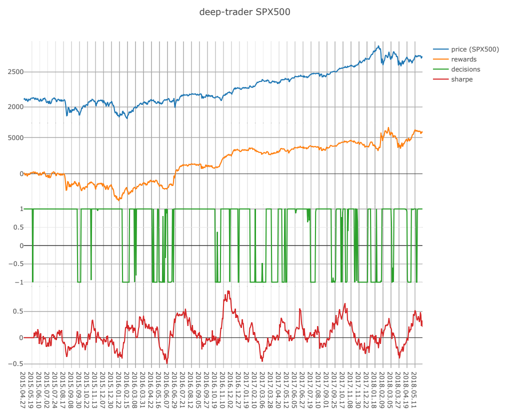
(d) Commissions impact

Figure 6.4: SPX500 commission analysis

The results of including new features are clearly better compared to the simpler case with no side data, especially in the lateral zones. In the case of the two hidden layers model the results are better than the simpler case but worse than the augmented data case, especially in the lateral market phase.



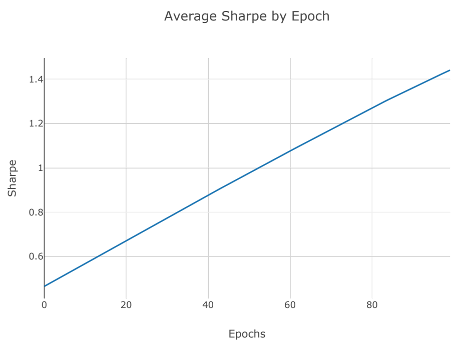
(a) One layer with 30 neurons commissions c and side data



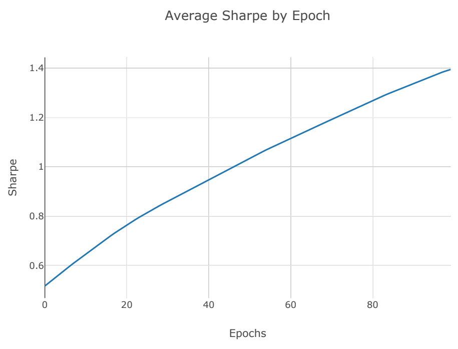
(b) Two layers with 30 and 30 neurons each one and commissions c

Figure 6.5: SPX500 other models

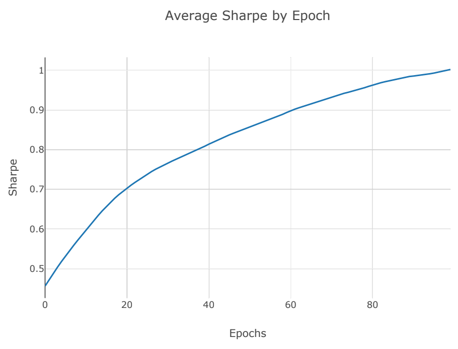
There exists a little convergence improvement with the side data example. In the case of two layers model we can see a convergence improvement, like in the XAUUSD case.



(a) SPX500 reward vs epochs 30 neurons



(b) SPX500 reward vs epochs side data

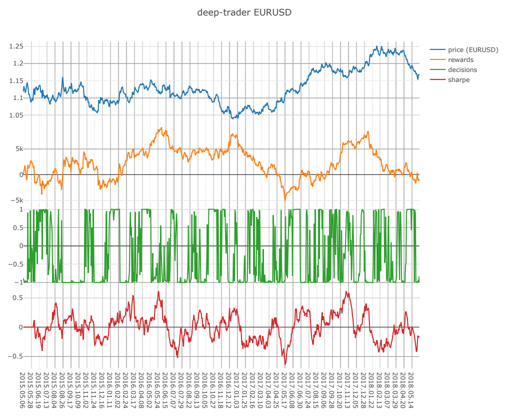


(c) SPX500 reward vs epochs 30, 30 neurons

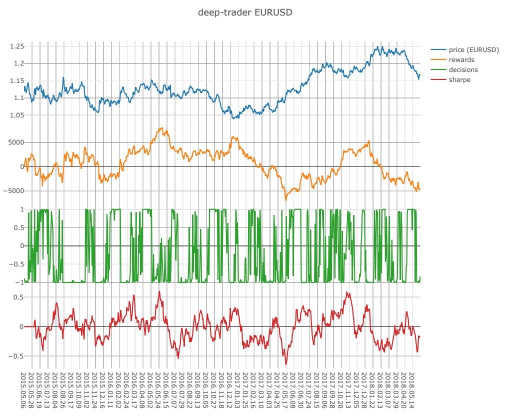
Figure 6.6: SPX500 reward vs epochs

6.3 EURUSD.

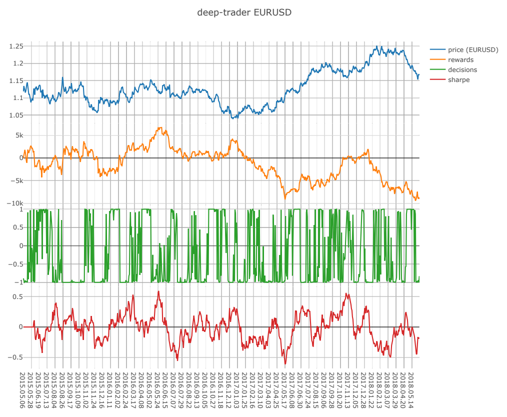
The results are worse than the XAUUSD and SPX500 cases, because there are more lateral phases in the EURUSD timeseries.



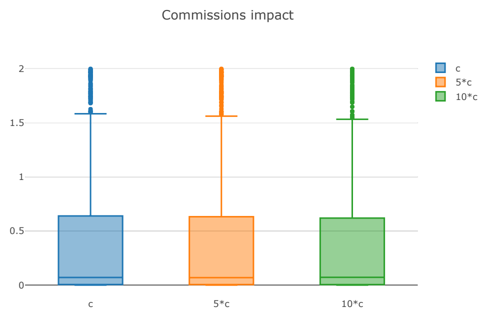
(a) One layer with 30 neurons and commissions c



(b) One layer with 30 neurons and commissions $5 * c$



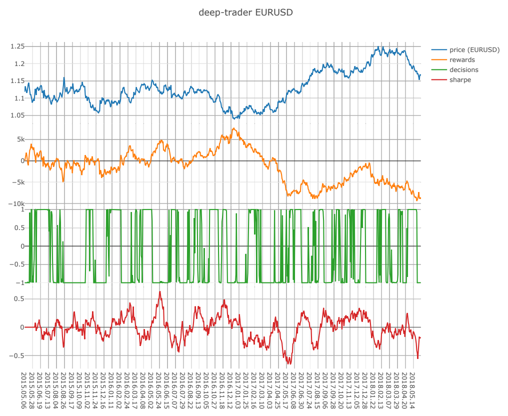
(c) One layer with 30 neurons and commissions $10 * c$



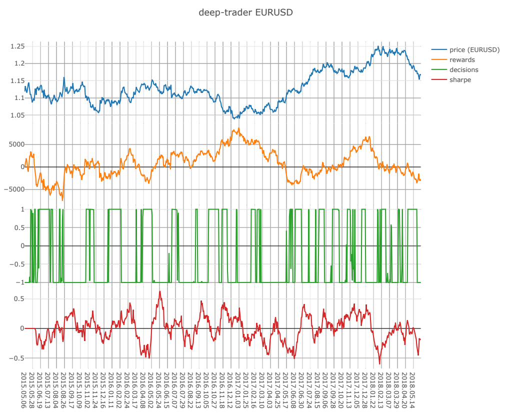
(d) Commissions impact

Figure 6.7: EURUSD commission analysis

We can see in the side data model two different behaviours. In the first part of the timeseries we can see an improvement with respect to the simpler case, in the second part the results are worse than in the one layer case. In the two layers model we can see a clear improvement in the first part of the timeseries, but again, the results in the latest part are worse but better than in the side data case.



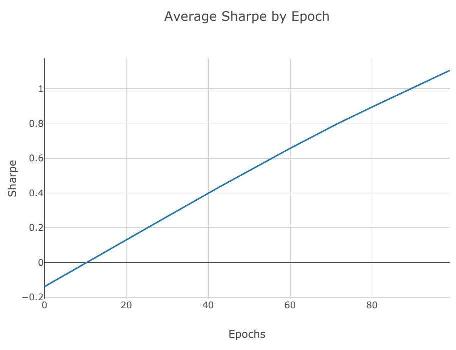
(a) One layer with 30 neurons commissions c and side data



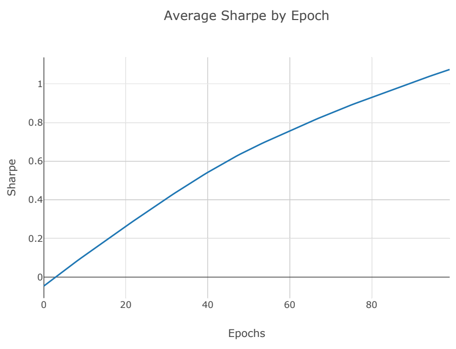
(b) Two layers with 30 and 30 neurons each one and commissions c

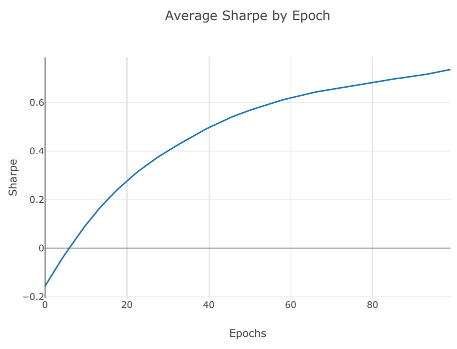
Figure 6.8: EURUSD other models

The explanations of the XAUUSD example are valid here.



(a) EURUSD reward vs epochs 30 neurons





(b) EURUSD reward vs epochs side data

(c) EURUSD reward vs epochs 30, 30 neurons

Figure 6.9: EURUSD reward vs epochs

Chapter 7

Conclusions and future work lines

After taking a look at the results we can write some conclusions:

- The simple agent trader is a trend follower algorithm, because it works better in trends than in lateral market phases.
- In most of the cases the side data augmented model improves the results of the simpler case with one hidden layer.
- In most of the cases the two hidden layer model improves the results of the model with one hidden layer.

The propose of this work is to present a trading algorithm based on Reinforcement Learning following the work presented on the paper [1] as principal. But this is not a closed work, there are many opened questions on it. In the case of the optimization, we could implement a bio-inspired algorithm for the optimization, or change the neural network model to use an LSTM model, like in [4]. For the deep structure part, we can include like in [3]a first layer based on fuzzy logic. For the agent we can add more recurrences not only the first. We can change the utility function like in [2] and we can change the model to trade multiple assets, like a different approach from classical portfolio investment based on Markowitz point of view. For the experiments part another work could be to include a step for simulating price series with different market regimes using for instance a mean reverting stochastic process like Ornstein–Uhlenbeck.

Bibliography

[1] John Moody and Matthew Saffell, Reinforcement Learning for Trading, *Advances in neural information processing systems 11. NIPS Conference, Denver, Colorado, USA, November 30 - December 5, 1998, pages 917-923*

[2] John Moody, Lizhong Wu, Yuansong Liao & Matthew Saffell, Performance functions and Reinforcement Learning for trading systems and portfolios*, Journal of Forecasting, Volume 17, pages 441-470, 1998.*

[3] Yue Deng, Feng Bao, Youyong Kong, Zhiquan Ren, and Qionghai Dai, Deep Direct Reinforcement Learning for Financial Signal Representation and Trading, *IEEE Transactions on Neural Networks and Learning Systems*, *2017, volume 28, pages 653-664.*

[4] David W. Lu, Agent Inspired Trading Using Recurrent Reinforcement Learning and LSTM Neural Networks. *2017.*

[5] John Moody, Yufeng Liu, Matthew Saffell and Kyoungju Youn, Stochastic Direct Reinforcement: Application to Simple Games with Recurrence. Appears in *Artificial Multiagent Learning, Sean Luke et al. editors, AAAI Press, Menlo Park, 2004.*

[6] Diederik P. Kingma, Jimmy Lei Ba. Adam: A method for stochastic optimization. *Published as a conference paper at ICLR 2015.*