Applied Soft Computing Journal 73 (2018) 783–794

Contents lists available at ScienceDirect

Applied Soft Computing Journal

journal homepage: www.elsevier.com/locate/asoc

# Reinforcement learning applied to Forex trading

João Carapuço, Rui Neves *, Nuno Horta

*Instituto de Telecomunicações, Instituto Superior Técnico, Torre Norte, Lisboa, Portugal*

## HIGHLIGHTS

- Develops a reinforcement learning system to trade Forex.
- Introduced reward function for trading that induces desirable behavior.
- Use of a neural network topology with three hidden-layers.
- Customizable pre-processing method.

## ARTICLE INFO

*Article history:*
Received 2 February 2018
Received in revised form 20 July 2018
Accepted 16 September 2018
Available online xxxx

*Keywords:*
Machine learning
Neural networks
Reinforcement learning
Financial trading
Foreign exchange

## ABSTRACT

This paper describes a new system for short-term speculation in the foreign exchange market, based on recent reinforcement learning (RL) developments. Neural networks with three hidden layers of ReLU neurons are trained as RL agents under the Q-learning algorithm by a novel simulated market environment framework which consistently induces stable learning that generalizes to out-of-sample data. This framework includes new state and reward signals, and a method for more efficient use of available historical tick data that provides improved training quality and testing accuracy. In the EUR/USD market from 2010 to 2017 the system yielded, over 10 tests with varying initial conditions, an average total profit of 114.0 ± 19.6% for an yearly average of 16.3 ± 2.8%.



## 1. Introduction

Reinforcement learning (RL) is a sub-field of machine learning in which a system learns to act within a certain environment in a way that maximizes its accumulation of rewards, scalars received as feedback for actions. It has of late come into a sort of Renaissance that has made it very much cutting-edge for a variety of control problems. Some high-profile successes ushered in this new era of reinforcement learning. First, in 2013, a London-based artificial intelligence (AI) company called Deepmind, stunned the AI community with a system based on the RL paradigm that had taught itself to play 7 different Atari video-games, 3 of them at human-expert level, using simply pixel positions and game scores as input and without any changes of architecture or learning algorithm between games [1]. Deepmind was bought by Google, and by 2015 the system was achieving performances comparable to professional human game testers in over 20 different Atari games [2]. Then, that same company achieved wide mainstream exposure when its Go-playing program AlphaGo, which uses a somewhat similar approach to the Atari playing system, beat the best Go player in the world in an event that reached peaks of 60 million viewers.

This was made possible by the use of neural networks, another sub-field of machine learning. These networks consist of interconnected artificial neurons inspired by the biological brain which process information by their dynamic state response to external inputs. This combination has been used before but often proved unreliable, especially for more complex neural networks. Advances in the neural networks field such as the ReLU neuron and gradient descent algorithms with adaptive learning rate, along with contributions from Mnih et al. [1] (and many others afterwards) aimed specifically at this mixed approach, have made it much more effective. From the point of view of reinforcement learning, neural networks provide much needed generalization power to find patterns for decision making that lead to greater reward. From the point of view of neural networks, reinforcement learning is useful because it automatically generates great amounts of labeled data, even if the data is more weakly labeled than through direct human intervention, which is usually a limiting factor for neural networks [3].

In this paper our aim is to find how to adapt these new developments in RL to the creation of an algorithmic system that generates profitable trading signals in financial markets. This requires successfully accomplishing the following steps:

* Corresponding author.
*E-mail addresses:* jmcarapuco@gmail.com (J. Carapuço), rui.neves@tecnico.ulisboa.pt (R. Neves), nuno.horta@tecnico.ulisboa.pt (N. Horta).

https://doi.org/10.1016/j.asoc.2018.09.017
1568-4946/

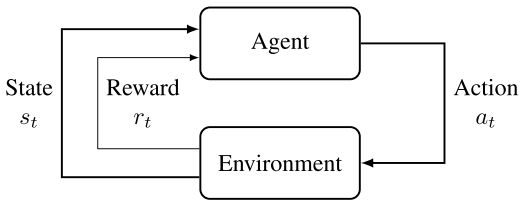



**Fig. 1.** The RL learning framework.

- Obtain a system that is able to stably, without diverging, learn and thus improve its financial performance on the dataset it is being trained on;
- Show that the system's learning generalizes to unseen data and that this generalization power can be harnessed to generate profitable decisions on a realistic simulation of live trading.

The foreign exchange market (Forex) was chosen as the testing ground for accomplishing these goals as having the largest volume of trades out of all financial markets, with roughly 25% of that volume concentrated on the EUR/USD pair [4] makes it ideal for short-term speculation. To reach the objectives above the main contributions offered in this work are:

- First adaptation of various state of the art reinforcement learning methodologies to foreign exchange trading:
  - ReLU (rectified linear unit) neurons and a gradient descent algorithm with adaptive learning rate to allow stable learning with deeper topology;
  - Experience replay mechanic and auxiliary Q-network for update targets from Mnih et al. [2], with double Q-learning by van Hasselt et al. [5];
- Introduction of a novel framework for trading using tick data, which includes a customizable preprocessing procedure shown to extract features which allow the agent to learn patterns that consistently generalize to out-of-sample data;
- Creation of a method for more efficient use of historical tick data resulting in both better training and more accurate testing;

This paper is organized as follows: Section 2 provides some background on the RL paradigm, the role of neural networks in RL and its adaptation to Forex trading, while Section 3 briefly overviews past work on automated trading with RL. In Section 4 we discuss our implementation of the RL signals, the simulated market environment which trains a neural network to trade by interacting with those signals, and the choice of hyper-parameters. The testing procedure and results of testing the system on the EUR/USD currency pair are detailed in Section 5. Finally, Section 6 draws conclusions from the results obtained and suggests future work to improve this approach.

## 2. Background

The problem in reinforcement learning is, to put it succinctly, that of learning from interaction how to achieve a certain goal [6]. We frame this problem by identifying within it two distinct elements and detailing their interactions, as depicted in Fig. 1. The elements are the learner/decision-maker which we call the agent, and that with which the agent interacts, known as the environment.

Considering discrete steps $t = 0, 1, 2, 3, \ldots$, at each $t$ the agent receives some representation of the environment's state, $S_t \in \mathbb{S}$, where $\mathbb{S}$ is the set of possible states, and uses that information to select an action $A_t \in \mathbb{A}(S_t)$, where $\mathbb{A}(S_t)$ is the set of actions available in state $S_t$. The agent chooses an action according to its policy $\pi_t$, where $\pi_t(s)$ represents the action chosen if $S_t = s$ for a deterministic policy. On the next time step $t+1$, the agent receives its reward $R_{t+1}$ as a consequence of action $A_t$, and information about the new state $S_{t+1}$. The goal is to find a policy $\pi_t$ that maximizes the sequence of rewards an agent will receive after step $t$:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{1}$$

where $\gamma$ is a discount rate that allows for a prioritization of immediate versus future rewards. To relate this discounted sum $G_t$ with a policy $\pi$, we use the action-value function $q_\pi$. This function tells us how valuable we expect to be performing a certain action $a$ in a given state $s$, and then following a policy $\pi$ thereafter:

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a]. \tag{2}$$

Q-learning [7] works by trying to estimate the action-value function $q_*(s, a)$ associated with the optimal policy, the policy which maximizes expected $G_t$ for any state. Q-learning takes advantage of an identity known as the Bellman optimality equation:

$$q_*(s, a) \doteq \mathbb{E}[R_t + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a], \tag{3}$$

by turning it into an update rule that can build an approximation of $q_*(s, a)$ iteratively:

$$q_{k+1}(s, a) = (1 - \alpha) q_k(s, a) + \alpha[r + \gamma \max_{a'} q_k(s', a')], \tag{4}$$

where $r$ is the reward obtained in the transition from $s$ to $s'$ with action $a$.

The classic implementation, a table with an entry for each state–action pair $Q(s, a) \approx q_*(s, a)$ updated through Eq. (4), has been shown to converge to $q_*$ with probability 1 under a few simple assumptions [6]. This tabular implementation is insufficient for complex problems such as financial trading. The state space is too large to be represented through a table and since the action-value function is estimated separately for each sequence there is no generalization power, which is essential for trading.

Neural networks acting as action value function approximators, Q-networks, are known for their capacity to process both linear and nonlinear relationships and for being the best available method for generalization power [3]. The Q-network computes a function $Q(s, a; \mathbb{W}) \approx q_*(s, a)$, where $\mathbb{W}$ is the set of parameters of the network. The approximation is improved by adjusting its set of parameters $\mathbb{W}$ in a way that minimizes a sequence of cost functions computed using observed transitions $e = [s, a, r, s']$, as described in the next section.

These concepts are applied to Forex trading by devising suitable reinforcement learning signals. Reward should be either the financial profit directly or a quantity correlated with financial profit, so that the estimated action-values steer the system to profitable actions. The action signal should mimic the actions available to a Forex trader, namely, opening and closing positions. The state signal is more elusive. To make consistent financial gains, the system must base its decisions on information with potential for price prediction. There is no academic consensus about such information, but technical analysis, which focuses on finding patterns in historical market data has a sizable amount of literature [8] supporting its effectiveness and surveys show that it is generally the main approach used by Forex traders [9,10].

We use market data in the form of ticks, the smallest granularity available, which allows for a more accurate portrayal of the market state and for a maximization of the amount of data available to our

Q-network. A tick $T_i$ is put out whenever the market updates prices:

$$T_i = \begin{bmatrix} B_i & A_i & Vb_i & Va_i \end{bmatrix}, \quad (5)$$

where $B_i$, $A_i$ are the bid price and ask price at the time $T_i$ was put out and $Vb_i$, $Va_i$ were the volume of units traded at those respective prices.

## 3. Related work

In this section we briefly discuss some previous applications of RL to financial trading. Table 1 summarizes the results obtained by those systems which were tested in the foreign exchange market. Unfortunately, these are few and far between as most systems focus solely on stock trading.

Direct reinforcement approaches disregard the use of value functions, and simply optimize the policy directly under the discussed RL framework. This is a popular approach for financial trading agents since Moody and Saffell [11] in 2001 introduced a direct reinforcement approach dubbed recurrent reinforcement learning (RRL) which outperformed a Q-learning implementation. Moody's RRL trader is a threshold unit representing the policy, in essence a one layer NN, which takes as input the past eight returns and its previous output and aims to maximize a function of risk-adjusted profit. This trader was tested on the first 8 months of 1996 with the currency pair USD/GBP, half-hourly data, achieving an annualized profit of 15%.

Gold [12] further tested the RRL approach on other currency markets with half hourly data from the entire year of 1996. Final profit level varied considerably across the different currency pairs, from −82.1% to 49.3%, with an average of 4.2% over ten pairs. In 2004 Dempster and Leemans [13] introduced what they dubbed as adaptive reinforcement learning (ARL), which was built upon the RRL approach. Using a RRL trader at its core, also with returns as the input, their system had an added risk management layer and dynamic hyper-parameter optimization layer. The ARL system was tested on 2 years of EUR/USD historical data, from January 2000 to January 2002, with 1 min granularity, achieving an average 26% annual return.

Deng et al. [15] created a system combining deep NN with Direct RL and using fuzzy learning to summarize the market trend. It was concluded that the system was able to select features from raw data due to the DL mechanism and learn effectively. This is one of the few examples, other than the work presented here, of state of the art RL methodologies applied to the creation of a financial trading system, however, it was developed and tested only for stock trading.

Value-based reinforcement, the category of methods in which this work falls into, has not been nearly as popular as the direct reinforcement approaches for Forex trading. Cumming [14] in 2015 introduced a RL trading algorithm based on least-squares temporal difference (LSTD), a technique that estimates the state value function. Their state signal consists of the open, highest, lowest and close prices (bid only) from the last 8 periods, where each period covers a minute. The reward given to the agent is purely the profit from each transaction. Training and testing used one minute data from 2014-01-02 to 2014-12-19. The reported annualized profit for the EUR/USD pair was 1.64%.

The main advantage of our system is its adaptability as it has shown an ability to accumulate profits with remarkably low drawdowns across a much larger span of time, which attests to its low risk across a variety of market conditions. Furthermore, its annualized profitability is quite high especially taking into consideration the fact that it is averaged over a larger span of time, and thus more reliable, and that these results were obtained with much more recent data, which is considered more difficult to be profitable in since profit opportunities from technical rules have declined over time [16].

## 4. Implementation

Our system, a Q-network, interacts with a simulated market environment designed to train and test it in a manner consistent with trading in the real foreign exchange market. In this section we describe how this simulated market environment coordinates the flow of information that reaches the system, the implementation of the three RL signals communicated between environment and system, the Q-network's topology and learning mechanisms, and the hyper-parameter selection.

### 4.1. Market environment

The market simulation follows prices from a tick dataset $\mathbb{T} = \{T_0, \ldots, T_K\}$, but the system is only allowed to make a decision every *time_skip* ticks, set according to the desired trading frequency. We selected and optimized the system for *time_skip* = 5000, which on average is somewhat less than two hours for 2013 EUR/USD tick data. This fits with our aim of a short-term speculator without being so high frequency as to be overly affected by practical concerns such as latency and lack of broker liquidity. The use of tick data with the *time_skip* parameter, over simply using market data with granularity of the desired trading frequency, allows for a more efficient use of available market history as will be described later in this subsection.

At each step $t$ the market environment is at the price in tick $T_{i=t\cdot time\_skip+b}$, where $b$ is the chosen starting point for that passage over the data. A state $S_t$ is sent to the system and a response in the form of an action signal $A_t$ is received. The market environment goes to the price in $T_{i+time\_skip}$ and drafts a new state $S_{t+1}$ and a scalar reward $R_{t+1}$ for the action $A_t$, which are both sent to the system.

In training passages over the data, the actions selected by the system have an exploratory component. The system has a probability $\epsilon$ to select a random action rather than the one with highest estimated action value. Also, observed experiences are stored and used to update the network weights. In test passages the action with highest estimated action value is always chosen, no updates are performed to the Q-network and profit obtained over the passage is recorded as a measure of performance.

To train the network we repeatedly perform training passages through a training dataset and two tests. First a test on the training dataset itself, to assess learning progress, and then a test on a validation dataset. The validation dataset contains data from a period of time immediately following the training dataset and is meant to assess ability to generalize to out-of-sample data. We rely on validation performance to tell us how many epochs to train for via early stopping [17]. The first few epochs are ignored in the early stop procedure to allow the Q-network to leave vicinity of weight hyperspace where it was randomly initialized.

Having a flexible starting point $b$ is a core addition to our market environment. Changing $b$ between each passage results in different paths taken through the dataset. This dynamic, made possible by the surplus of data from using tick granularity but only trading every *time_skip* ticks, is defined by a *nr_paths* parameter:

$$b \in \{\, x \cdot \frac{time\_skip}{nr\_paths} \mid x \in \mathbb{N}_0,\ x < nr\_paths \,\}. \quad (6)$$

The relationship between *time_skip* and *nr_paths* is depicted schematically in Fig. 2.

The paths depicted in Fig. 2 would be very similar, but for larger values of *time_skip* the distance between ticks visited in different paths increases and so does the quality of information added by this dynamic.

We select a new value of $b$ for each training passage. With this mechanism the network experiences greater variety of data, increasing learning quality. Fig. 3 shows the impact of this mechanic:

**Table 1**
Summary of performance obtained by RL trading systems tested on the foreign exchange market.

| Paper | Method | Asset | Time Frame | Annualized Profit |
|---|---|---|---|---|
| Moody and Saffell [11] | RRL | USD/GBP | First 8 months of 1996 | 15% |
| Gold [12] | RRL | Average over 10 currency pairs | Full year of 1996 | 4.2% |
| Dempster and Leemans [13] | ARL | EUR/USD | January 2000 to January 2002 | 26% |
| Cumming [14] | EUR/USD | LSTD | January 2014 to December 2014 | 1.64% |

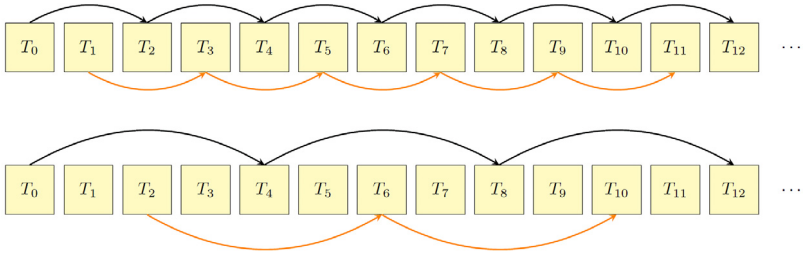
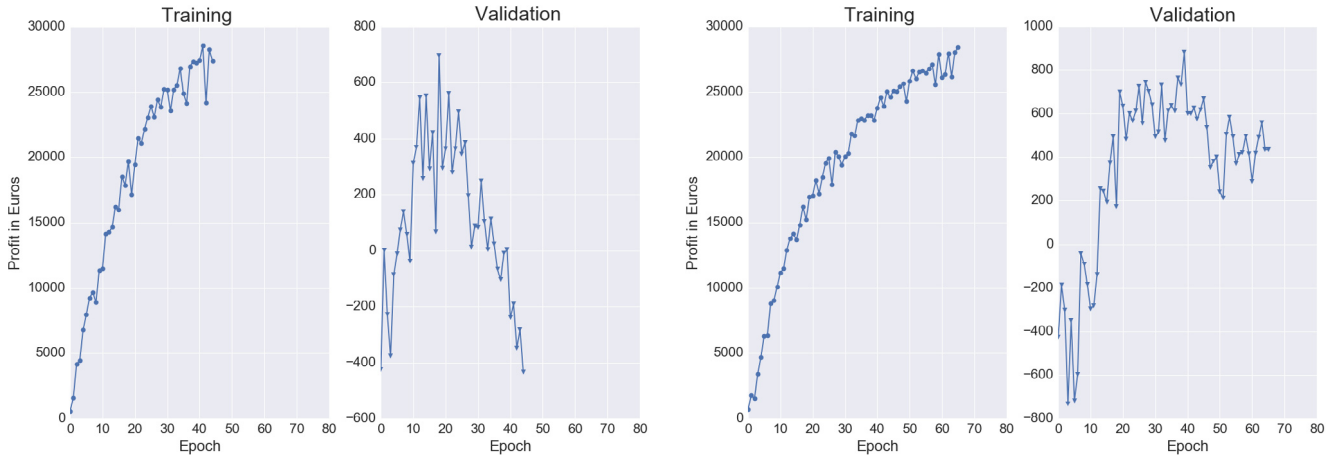



**Fig. 2.** Schematic of the simulated market environment passing through the tick dataset with *nr_paths* = 2, one path depicted with black arrows and a second one in orange arrows. Top: *time_skip* = 2, Bottom: *time_skip* = 4. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)




(a) With $nr\_paths = 1$.

(b) With $nr\_paths = 10$.

**Fig. 3.** Example of learning curves with and without the *nr_paths* dynamic. Training dataset: 01/2011 to 01/2012. Validation dataset: 01/2012 to 07/2012.

performance in the validation dataset improves and overfitting is delayed and less severe.

Furthermore, standard testing approaches deliver a path-dependent distribution of gains thus increasing the chances of a "lucky" trading strategy. Making each test result an average over a number of test passages starting from all different initial ticks $T_b$ strongly mitigates this issue. The standard deviation of performance between these passages also provides a measure of uncertainty of the assessment. This makes our results more accurate representations of the expected real trading performance.

### 4.2. State signal

The state signal in this work was created to make gradient learning faster, more stable and less prone to getting stuck in local minima by instilling, as far as possible, the following characteristics [18]:

- Uncorrelated input variables;
- Input variables with a similar range;
- Input variables with an average over the training dataset close to zero for each input variable.

The main component of the state signal $S_t$ are features extracted from market data with the aim of creating a compact representation of tick data which contains most relevant information while reducing its dimensionality. With the simulated market at tick $T_i$ we divide the tick data preceding it into windows described by the parameter array:

$$TW = \begin{bmatrix} TW_0 = 0 & TW_1 & TW_2 & \dots & TW_N \end{bmatrix}$$

so that the $n$th window, where $n \in [1, 2, \dots, N]$, includes the ticks $T_k$ with $k \in [i - TW_{n-1}, i - TW_n]$. From each window, features $F_i^n$ are extracted to build a feature array $F_i = F_i^1 \cup F_i^1 \cup \dots \cup F_i^N$. Parameter $TW$ is set empirically to strike a balance between the level of detail (number of ticks per window), the scope (total number of ticks included) and the dimensionality (number of windows) of our compact representation of tick data.

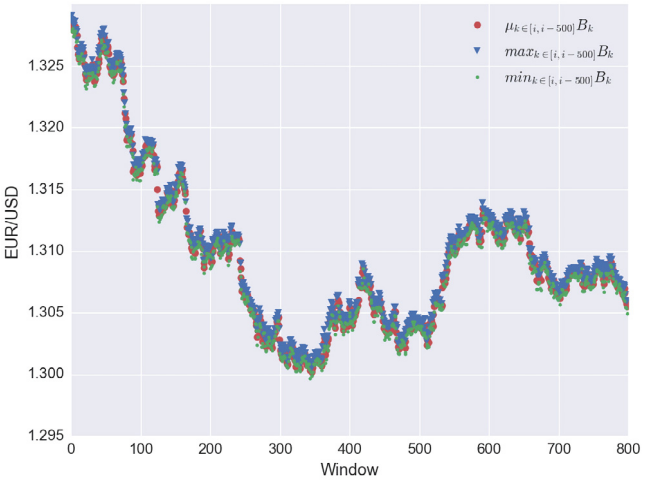



**Fig. 4.** Features extracted from 400,000 ticks divided into 500-tick windows. Data is EUR/USD pair market data from 2013, retrieved from Duskacopy broker.

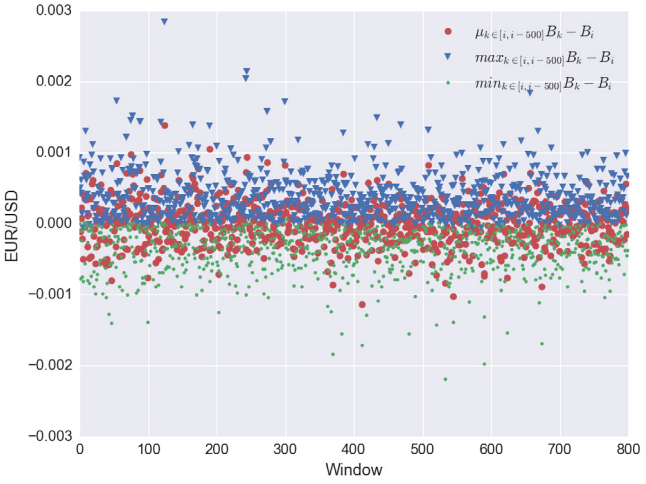



**Fig. 5.** Same as Fig. 4, but with our solution for feature correlation.

The features extracted from each window are simple descriptive statistics – mean, maximum, standard deviation and minimum values – for the different components of tick data: bid/ask prices and volumes. Since the bid and ask prices are almost perfectly correlated, we replace ask price with spread $S_i \equiv A_i - B_i$.

The price distribution is strongly non-stationary, there is a large slow-varying underlying bias value around which prices fluctuate. For this reason the mean, maximum and minimum bid price values from a given window are all strongly correlated between themselves and with values from neighboring windows, as illustrated in Fig. 4. This would make learning very difficult for the neural network. Our solution is to subtract from these features an estimate of the underlying bias: the bid value of the current tick $B_i$. Fig. 5 depicts how this change solves the problem. Moreover, looking at Fig. 4 if we divided the data there into a left half for a training dataset and a right half for a validation dataset, we see that the range of values for these features obtained while training could be completely different from those found out-of-sample. With our solution the range of values is much more predictable and uniform, making generalization of acquired knowledge easier.

Thus, the features $F_i^n$ extracted from each window of ticks are:

$$\begin{vmatrix} \underset{k\in[i-TW_{n-1},i-TW_n]}{\mu} B_k - B_i & \underset{k\in[i-TW_{n-1},i-TW_n]}{\mu} S_k \\ \underset{k\in[i-TW_{n-1},i-TW_n]}{\mu} Vb_k & \underset{k\in[i-TW_{n-1},i-TW_n]}{\mu} Va_k \\ \underset{k\in[i-TW_{n-1},i-TW_n]}{\max} B_k - B_i & \underset{k\in[i-TW_{n-1},i-TW_n]}{\max} S_k \\ \underset{k\in[i-TW_{n-1},i-TW_n]}{\max} Vb_k & \underset{k\in[i-TW_{n-1},i-TW_n]}{\max} Va_k \\ \underset{k\in[i-TW_{n-1},i-TW_n]}{\text{std}} B_k & \underset{k\in[i-TW_{n-1},i-TW_n]}{\text{std}} S_k \\ \underset{k\in[i-TW_{n-1},i-TW_n]}{\text{std}} Vb_k & \underset{k\in[i-TW_{n-1},i-TW_n]}{\text{std}} Va_k \\ \multicolumn{2}{c}{\underset{k\in[i-TW_{n-1},i-TW_n]}{\min} B_k - B_i} \end{vmatrix}$$

where $\mu$, max, std and min are the mean, maximum, standard deviation and minimum values in the interval. Features $\min_{k\in[i-TW_{N-1},i-L]} S_k$, $\min_{k\in[i-TW_{N-1},i-L]} Vb_k$ and $\min_{k\in[i-TW_{N-1},i-L]} Va_k$ were removed from the feature array as analysis of their distribution and empirical tests both indicated they do not contain useful information[1] and simply contributed to overfitting.

To remove outliers from these features we apply a simple percentile-based filter. The $q$th percentile of feature $x_j$, $q^{th}(x_j)$, is computed as the value $\frac{q}{100}$ of the way from the minimum to the maximum of a sorted copy of an array containing all entries of the feature for a dataset. The filter rule used is:

$$x_{j,i} = \begin{cases} (1-q)^{th}(x_j), & \text{if } x_{j,i} < (1-q)^{th}(x_j) \\ q^{th}(x_j), & \text{if } x_{j,i} > q^{th}(x_j) \\ x_{j,i}, & \text{otherwise} \end{cases} \quad (7)$$

where $x_{j,i}$ is the $j$th element of $F_i$. With outliers removed each feature can be normalized to the range $[-1, 1]$ via min–max normalization without skewing the main body of the distribution and compromising sensitivity.

The market may have higher volume and volatility at certain hours of the day due to the working hours of the major trading hubs, thus we want to relay that information to the system to aid in its interpretation of the market. We use two input variables to encode the hour:

$$time_i^1 = \sin(2\pi \frac{seconds_i}{86400})$$
$$time_i^2 = \cos(2\pi \frac{seconds_i}{86400})$$

where $seconds_i$ is the time when $T_i$ was put out converted to seconds. This encoding method effectively conveys the cyclical nature of hours to the neural network.

Price prediction features are the bulk of the state signal, but there are also a few "functional" inputs. We relay to the system if it currently has a position open and the value of that position using an integer scalar $h_t$:

$$h_t = \begin{cases} 1, & \text{if long position open} \\ 0, & \text{if no position open} \\ -1, & \text{if short position open} \end{cases} \quad (8)$$

and a float scalar $v_t$ with the unrealized profit of the currently open position. $v_t$ is normalized to the range $[-1, 1]$ through min–max normalization.

Finally, a float scalar $c_t$ is included which tells the system the current size of the account compared its initial size. $c_t$ is normalized and clipped to the range $[-1, 1]$, where the maximum and

[1] Values for these features seemed artificial, maybe due to a lower bound imposed by the broker on the data making it non-representative of the real market conditions when below a certain threshold of activity.

**Table 2**
Interpretation of each value of the scalar action signal by the market environment.

| Signal | Current position | Action |
|---|---|---|
| $a_0$ | Long | Hold |
| | None | Nothing |
| | Short | Hold |
| $a_1$ | Long | Hold |
| | None | Open Long |
| | Short | Close Short |
| $a_2$ | Long | Close Long |
| | None | Open Short |
| | Short | Hold |

minimum extremes represent, respectively, a *safe* and *failure* levels set by the user. Should the current account size reach the *failure* threshold, the system flags it as a RL terminal state. The terminal state condition was added for its anchoring effect, by providing the system a state with an exactly known action value, and to further punish strings of consecutive bad decisions. Empirically we verified that a failure threshold closer to 1, meaning more instances of terminal states, increases learning speed, although if it is set too close to 1 learning quality is disrupted.

### 4.3. Action signal

We impose that the agent can only invest one unit of constant size *position_size* of the chosen asset at a time. Therefore the action signal can be simplified to $\mathbb{A}(s) = [a_0, a_1, a_2]\ \forall s \in \mathbb{S}$, where $[a_0, a_1, a_2]$ are interpreted by the environment as described in Table 2.

### 4.4. Reward signal

A reward signal based only on the profit obtained was tested, but it was observed to introduce consistent behavioral flaws. Since the system is not punished for holding into positions with negative unrealized profits, but is punished at moment of their closing, it learns to keep a position open until its unrealized profit bounces back to positive values however long that may take, as observed in Fig. 6. This is a critical flaw, the asset may never regain its former value or take so long to do so as to make the system inviable.

With our solution actions to close positions are still rewarded by the resulting profit, but a new reward component is added. Actions to open positions or hold positions open are rewarded with the variation of unrealized profit: the difference between unrealized profit in the state $S_t$ in which they were taken and the unrealized profit in the state $S_{t+1}$ to which they lead. Tests showed both approaches obtain similar profit, but that this second approach effectively corrects for the behavior flaws previously observed, as exemplified in Fig. 7, which means that profit is obtained with less risk.

The most common methods to adjust profit for risk are the Sharpe ratio, which divides the profit by the standard deviation of unrealized profit, and its modification the Sortino ratio, which divides profits by the downside deviation of unrealized profit. Both aim to penalize large variations of unrealized profit, which are interpreted as risk.

Our reward is in essence very similar, especially to the Sortino ratio since positive volatility is not punished but rewarded. The difference is that rather than introduce the risk punishment/reward at the end of the transaction by adjusting the profit reward, it is spread out over its life cycle with the variation at each step. We observed this to have the benefit of speeding up the learning process, possibly by alleviating the credit assignment problem [6].

The reward signal is normalized to the range $[-1, 1]$ with min–max normalization to make learning more stable by limiting the size of the cost function gradients [2]. In the case of the variation of unrealized profit, it is first subjected to a percentile filter to clip values above a percentile *return_percentile* of all possible unrealized profit variations in a given training dataset. This filtering is essential as it prevents weekend-gap variations or other outliers from skewing the reward distribution, and also allows for control over how much importance the system gives to unrealized profit variation rewards compared to profit rewards.

### 4.5. Q-network

The core of the system is a fully connected neural network, depicted schematically in Fig. 8. This network is tasked with computing $Q(s; \mathbb{W}_k)$, where $\mathbb{W}_k$ is the set of weights and biases of the network at iteration $k$, an approximation of the action value function for the market environment.

The input layer has a number of neurons defined by the elements in our state signal (constructed as described in Section 4.2), which per the hyper-parameter *TW* chosen in Section 4.6 results in 148 input neurons (i.e. the state signal has 148 dimensions). This input layer is followed by three hidden layers with 20 ReLU neurons each.

The ReLu activation function was chosen due to its documented superiority for training multi-layer neural networks [19,20] over other widely used activations. Number of hidden neurons and layers was determined through observing changes in performance and learning curve shape during various informal tests (see [17] and [18] for practical recommendations) using the same datasets used for all hyper-parameter selection, and choosing those settings which offered the most satisfactory results. This process was performed in tandem with the choice of parameter *TW* to make sure there was a balance between power of the network and number of input variables. We found this balance hard to achieve, with the Q-network easily slipping into overfitting, not surprising given the noisy nature of financial data.

Tests with L1 regularization, L2 regularization and dropout regularization to prevent overfitting were not successful and no such methods were included in the final architecture. We found significant performance gain from the three hidden layer topology versus a two hidden layer or a single hidden layer with the same total number of neurons, although adding a fourth layer made training difficult and resulted in decayed performance.

The output layer has three neurons with linear activations to represent action values, which may take any real value. With this topology we have chosen to approximate the optimal action value function in its vectorial form:

$$Q(s; \mathbb{W}_k) \approx (q_*(s, a_0), q_*(s, a_1), q_*(s, a_2)), \tag{9}$$

rather than the arguably more intuitive:

$$Q(s, a_n; \mathbb{W}_k) \approx q_*(s, a_n), \tag{10}$$

as this second option would require a forward propagation to obtain the Q-value of each possible action, while this approach proposed by Mnih et al. [1], requires only a single forward propagation, saving computational resources.

The above-described Q-network is initialized with a random set of weights where each weight is drawn from an independent normal distribution with range $[-1, 1]$. As the network interacts with the market environment in a training passage, it collects and stores observed transitions in a sliding window buffer of size $N$. Every *update_q* steps a set of observed transitions, $\mathbb{S}_k = \{e_1, \ldots, e_B\}$ where $e_p = [s^p, a^p, r^p, s'^p]$, is randomly drawn from the buffer to be used for learning. This is the experience replay mechanism, which is regarded as more efficient than using consecutive observations

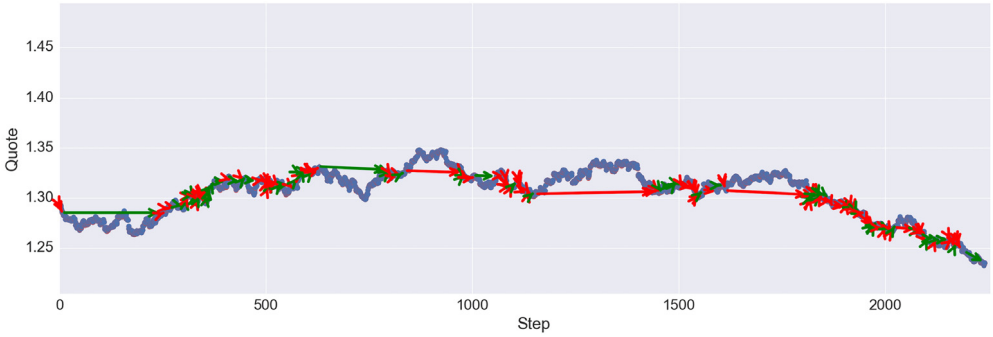



**Fig. 6.** Behavior with the initial reward approach on a 6 month validation dataset from 2012.01 to 2012.06. Each step skips 5000 ticks. Green arrows are long positions and red arrows are short positions. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

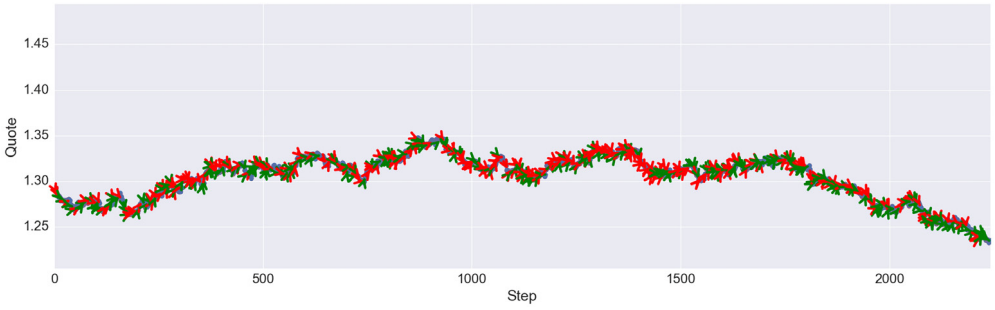



**Fig. 7.** Example of behavior with the final approach to reward on a 6 month validation dataset from 2012.01 to 2012.06. Each step skips 5000 ticks. Green arrows are long positions and red arrows are short positions. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

to learn [21]. The mean squared error between desired and actual output of the Q-network is computed for that set:

$$E(\mathbb{W}_k) = \sum_{p=1}^{B} \frac{E_p(\mathbb{W}_k)}{B} = \sum_{p=1}^{B} \frac{(y_p - Q_{a^p}(s^p; \mathbb{W}_k))^2}{B}, \tag{11}$$

where $y_p$ is the target output. Adapting Q-learning (Eq. (4)) directly would yield $y_p = r^p + \max_a Q_a(s'^p, \mathbb{W}_k)$. However, this target often causes learning issues with Q-networks.

As proposed by Mnih et al. [2] we introduce an auxiliary Q-Network, $Q(s, \mathbb{W}^-)$, topologically identical to the original Q-Network. Its weights $\mathbb{W}^-$ are static and periodically copied from the original set $\mathbb{W}_k$ every *update_q*$^-$ updates. This auxiliary Q-network is used to generate the targets for updates, $y_p = r^p + \max_a Q_a(s'^p, \mathbb{W}^-)$. This prevents them from shifting in a correlated manner with the Q-value estimations, which could make learning more difficult and cause it to spiral out of control through feedback loops. A last modification known as double Q-learning, implemented as suggested by van Hasselt et al. [5], decouples action choice from the target Q-value generation:

$$y_p = r^p + Q_a(s'^p, \mathbb{W}^-) \text{ with } a = \arg\max_n Q_n(s'^p, \mathbb{W}_k), \tag{12}$$

which is known to otherwise introduce a bias in the action value estimation resulting in poorer policies.

With $E(\mathbb{W}_k)$ computed, backpropagation [22] provides an efficient way of finding $\nabla_{\mathbb{W}_k} E(\mathbb{W}_k)$. The gradient is then used in a gradient descent algorithm to improve $\mathbb{W}_k$, *i.e.* to learn. The gradient descent implementation we used is known as RMSProp (Root Mean Square Propagation) [23]. RMSProp has been shown to work very well in a variety of benchmarks and practical applications [24,25] and has been successfully utilized for Q-Networks [2].

For a given parameter $\theta_{i,k} \in \mathbb{W}_k$, the learning rate is adjusted with a exponentially decaying average of squared previous gradients $v_k$:

$$\begin{aligned} v_k &= 0.9 v_{k-1} + 0.1 \left(\frac{\sum_{p=1}^{B} \nabla_{\theta_{i,k}} E_p(\mathbb{W}_k)}{B}\right)^2 \\ \theta_{i,k+1} &= \theta_{i,k} - \frac{\alpha}{v_k} \frac{\sum_{p=1}^{B} \nabla_{\theta_{i,k}} E_p(\mathbb{W}_k)}{B} \end{aligned} \tag{13}$$

where $\alpha$ is the general base learning rate.

This approach which uses a subset of size $B$ of the training cases in computing the error, and thus the update direction, is known as the mini-batch. It is a middle ground between the stochastic (one training case per update) and the full batch (all training cases per update) gradient descent methods. Each update with the first method is cheaper computationally, while each update with the second method is presumably a better step. But since batch updates can take advantage of the speed-up of matrix–matrix products over matrix–vector products [17] they somewhat lessen the computational cost gap. Thus, for convex, or relatively smooth cost manifolds batch updates may be preferred. But since gradients only represent the steepest descent locally, when the cost manifolds are highly non-convex even an error-less gradient update may not have the desired direction. In fact, it may become helpful to have some noise in the gradient updates: updates with less error will discover the minimum of whatever basin the weights are initially place in, while noisier updates can result in the weights jumping into the basin of another, potentially deeper local minimum [18].

Thus, we used the mini-batch with a carefully empirically chosen parameter $B$, small enough to avoid some of the poor local minima, but large enough that it does not avoid the global minima or better-performing local minima[2] and reaps advantage of matrix–matrix computational gains.

[2] This assumes that the best minima have a larger and deeper basin of attraction, and are therefore easier to fall into.

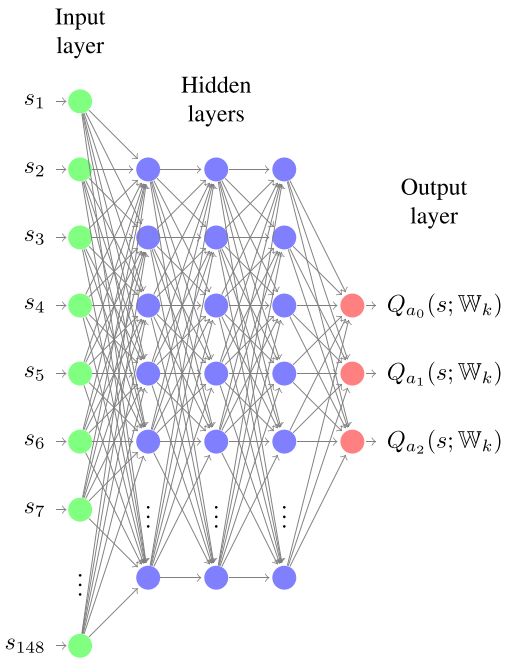



**Fig. 8.** Schematic of the trading system's Q-network. $s_n$ is the $n$th element of the vector representing the state $s$. Hidden and output neurons have ReLU and Linear activations respectively.

**Table 3**
Hyper-parameters chosen for the trading system.

| Hyperparameter | Value | Description |
|---|---|---|
| $q$ | 99 | Percentile filter for features |
| *time_skip* | 5000 | Ticks skipped on each step |
| *nr_paths* | $\frac{time\_skip}{500}$ | Nr. of paths through data |
| *return_percentile* | 90 | Percentile filter for returns |
| *failure*/*safe* | 0.8/2.0 | Parameters for terminal state |
| *update_q* | 8 | Steps between main NN updates |
| *update_q*$^-$ | 5000 | Steps between aux. NN copy |
| $B$ | 60 | Nr. elements in mini-batch |
| $N$ | 60000 | Size of experience buffer |
| $\gamma$ | 0.99 | Q-learning discount |
| $\alpha$ | 0.001 | RMSProp learning rate |
| $\epsilon_0/\epsilon_f$ | 1/0.3 | Initial/final value of $\epsilon$ |
| *init_acc_size* | 10000 | Initial account size |
| *position_size* | 10000 | Units invested per trade |

### 4.6. Hyper-parameters

To assess each configuration of hyper-parameters it is necessary to fully train the network and test its performance, a time consuming process. Also, there is a somewhat large number hyper-parameters to tune and they are related to each other in complex ways: changing one of them means a number of others may no longer be optimal. This made it unfeasible to perform a systematic grid search for the optimal set of hyper-parameters with the available resources.

Instead, the hyper-parameters were selected by performing an informal search using three EUR/USD datasets with 12 months of training and 6 months validation, with the goal of obtaining the most stable learning curve with the highest generalization capability as measured by peak performance on the validation dataset. The hyper-parameters thus selected are detailed in Table 3.

Distinct account size and position size parameters in the system allow for future implementation of a variable position size scheme to optimize profitability. Tests performed in Section 5 do not include such optimization, *init_acc_size* = *position_size* =10,000 was set as a placeholder value. The RL discount rate parameter

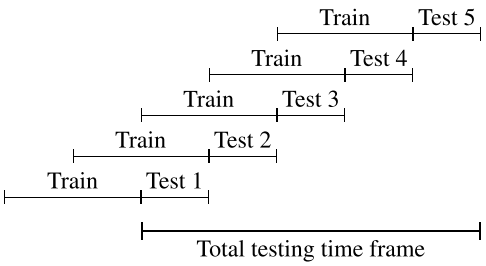



**Fig. 9.** Rolling window approach to testing.

$\gamma$ is close to 1, meaning future rewards are weighed heavily. This reflects the fact that a successful transaction requires future-oriented decision making. The learning rate $\alpha$ was kept at its default value for RMSProp of 0.001 [26], as it offered satisfactory results and deviations from this value did not result in improved learning. Parameter $\epsilon$ for action selection is made to decay over the training process from a initial value $\epsilon_0$ and final value $\epsilon_f$ so that the system takes advantage of acquired knowledge proportionally to the quality of that knowledge. We have found that *nr_paths* $= \frac{time\_skip}{500} = 10$, meaning 10 different paths with a distance of 500 ticks between them, is a balanced value for both testing and training passages, and use it throughout this work.

As for the parameter array $TW$, whose elements define the tick windows from which to extract features, the following values were selected:

$$TW = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 & 60 \end{bmatrix} \cdot \frac{time\_skip}{2}$$

which for *time_skip* $= 5000$ means 10 windows with 2500 ticks each, followed by a single window with 125,000 ticks. This offers a detailed account of the recent past mixed with an overview of a much more distant past, which may help by giving context to the more detailed segment. $TW$ is defined in terms of *time_skip* to allow a seamless transition to other *time_skip* values, although it should ideally be re-optimized.

## 5. Testing

While performance on the validation dataset is a good measure of generalization power, it has a positive bias over performance on actual live trading since training is stopped based on where validation performance reaches its peak. A test dataset is introduced to deliver an unbiased estimate of live trading performance by testing the model chosen through the training procedure on it.

The working premise is that a model that has learned to perform well on the training dataset and subsequently also performs well out-of-sample on a validation dataset is more likely to then be profitable on the test dataset. However, markets are known to be non-stationary [27,28] and if market dynamics change too much from the training/validation period to the testing period this assumption will not hold. Therefore we perform the tests using a rolling window as depicted in Fig. 9.

Theoretically, smaller steps of the rolling window, meaning smaller test datasets, tackle non-stationarity more effectively. However, this does entail a greater number of tests. Considering the running time of each test with our resources, we decided on a test dataset size of 4 months.

As for training and validation datasets, if they are smaller the data the network learns (training dataset) and the filter through which the model is selected (validation dataset) should be more similar to the test dataset and thus suffer less from non-stationarity. But since market data is remarkably noisy, if the training dataset is too small it will simply learn noise and lose generalization power.

If the validation dataset is too small the overall performance on the dataset is more easily influenced by noisy unpredictable events and will tend to select models that randomly perform well on those events rather than those that truly generalize well. A training and validation dataset size of 24 and 6 months was chosen through informal testing with one of the hyper-parameter selection datasets.

Below we describe the results of testing the system on post-2008 crisis EUR/USD pair data: from 2010 to 2017. These tests are meant to assess the hypothetical validity of the trading system by confirming a positive expectation of gains, rather than attempt to accurately project its full revenue potential. Thus, execution is not financially optimized: position sizing was set *a priori* and kept constant and no stop-loss or take-profit orders were used. While bid–ask spread commission is included, the volume commission is not as the rates depend strongly on a specific trader's resources.

### 5.1. Results

We refer to each test by year followed by quadrimester. In Fig. 10 a sample of the learning curves responsible for the selection of the final model for each of the tests is displayed. We observed again and again that training led to improved out-of-sample validation performance, which solidified the notion that this architecture is indeed capable of learning real predictive market patterns. The shape of the learning curves is, however, very dataset dependent, presumably due to noisy market events and variable degrees of market non-stationarity.

For the three 2010 test datasets we curtailed the training and validation dataset size to 12/1, 12/1 and 18/3 months respectively, to avoid including data from the 2008 crisis. For the 2011 p.3 test, the standard training/validation dataset size resulted in learning curves from which no candidate could be selected (validation performance only worsened with training). Halving the validation to 3 months solved this issue, suggesting a portion of the removed 3 months was incompatible with the training dataset. The final exception was the 2016 p.2 test. There was a marked increase in tick density in the last two quadrimesters of 2016 from an average of 6,738,296 ± 2,339,367 ticks in the remaining quadrimesters to 15,969,628 and 19.281.366. We could not ascertain the reason for this increase, most likely an internal change in the broker's method of producing tick data. We halved the validation dataset size for the 2016 p.2 test to include, proportionally, more data with the altered tick density so it would select a candidate that performs well in these new conditions, without success.

Table 4 details the test results in both absolute profit and profit relative to the initial account size. Note that results shown are an average of 10 test passages from different initial points following the *nr_paths* methodology, and their uncertainty is the standard deviation over those passages. The system is profitable in all but three tests, 2012 p.3, 2016 p.2 and 2016 p.3, although in two other cases, 2011 p.2 and 2012 p.1, it just about breaks even. This means it generated significant profit in roughly 75% of the tests, and significant losses in only 14%.

The simple and compounded total test results are concisely described in Table 5. We chose to use a fixed position size during each test and compound between tests, ie. change the fixed position size at the onset of a test by including the total profit of the previous tests in the account.

In Table 6 we look at the trades individually for a better insight into the behavior of the system. It is apparent that the system generally relies on a large number of quick trades, roughly 550 trades per year with 0.2% profit/loss per trade, which are profitable 53.5% of the time. There is a slight tendency to favor the short position, especially in trades that end up being profitable. This can be explained the fact that the Euro has on average been losing value relative to the Dollar since 2010. The duration that each position stays open is similar in both profitable and unprofitable trades, showing that the system has no problem cutting its losses and does not overly wait for rebounds.

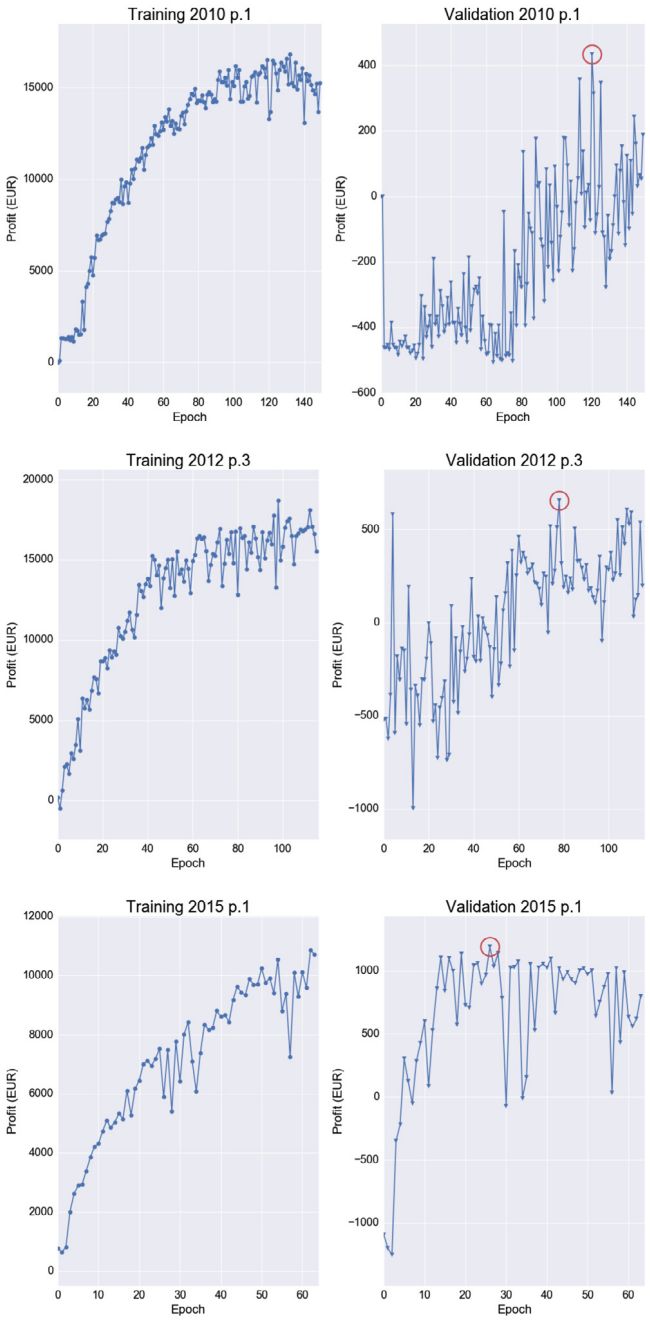



**Fig. 10.** Sample of validation learning curves. Red circle represents chosen model.

### 5.2. Result analysis

It was observed that validation profit is not a good predictor of test profit, with a Pearson correlation coefficient of −0.28 at *p*-value significance of 0.2 indicating no correlation between annualized validation and test profits. It is clear that while the training process creates a positive expectation for profits in a test setting, results are too dataset dependent to create an expectation on the magnitude of those profits.

On the other hand, the standard deviation between different paths obtained by the model in the validation dataset compared to that of the test dataset has a Pearson correlation coefficient of 0.73 with *p*-value significance 0.0002. This means that stabler candidates can be selected based on the validation process. Considering that our total profit, with triannual compounding, has an

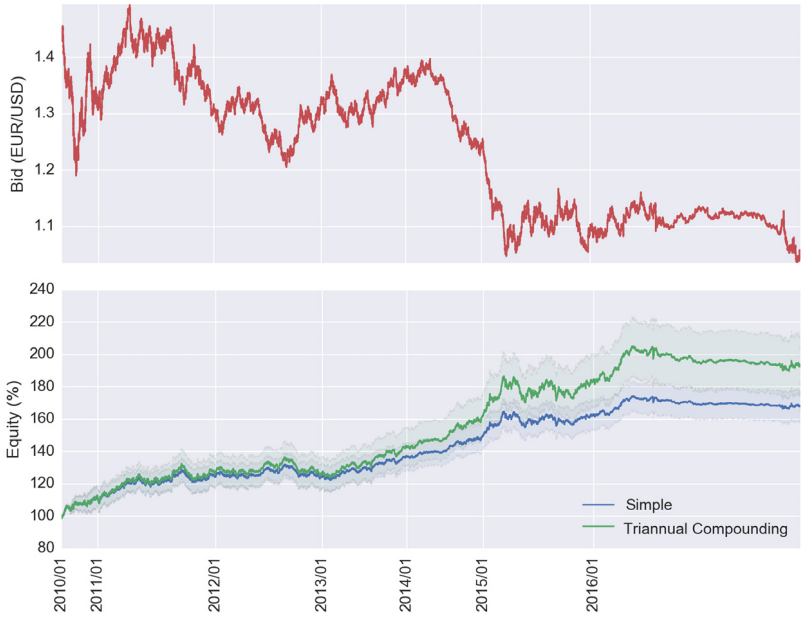



**Fig. 11.** Bid prices for EUR/USD pair (top) and equity growth curve with and without compounding (bottom). Light area around equity curves represents their uncertainty. *X*-axis is in units of ticks.

**Table 4**
Overview of the test results.

| Test Name | Test Profit | |
|---|---|---|
| | Abs. (EUR) | Rel. (%) |
| 2010 p.1 | 605 ± 312 | 6.1 ± 3.1 |
| 2010 p.2 | 159 ± 368 | 1.6 ± 3.7 |
| 2010 p.3 | 322 ± 377 | 3.2 ± 3.8 |
| 2011 p.1 | 1011 ± 75 | 10.1 ± 0.8 |
| 2011 p.2 | 36 ± 228 | 0.4 ± 2.3 |
| 2011 p.3 | 320 ± 343 | 3.2 ± 3.4 |
| 2012 p.1 | 70 ± 87 | 0.7 ± 0.9 |
| 2012 p.2 | 323 ± 218 | 3.2 ± 2.2 |
| 2012 p.3 | −327 ± 78 | −3.3 ± 0.8 |
| 2013 p.1 | 388 ± 231 | 3.9 ± 2.3 |
| 2013 p.2 | 342 ± 173 | 3.4 ± 1.7 |
| 2013 p.3 | 434 ± 69 | 4.3 ± 0.7 |
| 2014 p.1 | 249 ± 70 | 2.5 ± 0.7 |
| 2014 p.2 | 296 ± 141 | 3.0 ± 1.4 |
| 2014 p.3 | 841 ± 51 | 8.4 ± 0.5 |
| 2015 p.1 | 767 ± 36 | 7.7 ± 0.4 |
| 2015 p.2 | 131 ± 112 | 1.3 ± 1.1 |
| 2015 p.3 | 328 ± 227 | 3.3 ± 2.3 |
| 2016 p.1 | 962 ± 131 | 9.6 ± 1.3 |
| 2016 p.2 | −264 ± 311 | −2.6 ± 3.1 |
| 2016 p.3 | −212 ± 222 | −2.1 ± 2.2 |

**Table 5**
Simple and compounded total test profit.

| Compounding frequency | Total Test Profit (%) | Yearly Avg. Test Profit (%) |
|---|---|---|
| None | 67.8 ± 9.8 | 9.7 ± 1.4 |
| Yearly | 89.8 ± 16.9 | 12.8 ± 2.4 |
| Triannual | 92.5 ± 18.4 | 13.2 ± 2.6 |

uncertainty of almost 20%, our choice of candidate solely through peak validation performance was misguided, and exploring this correlation could be an important avenue for improving the system's performance.

We continue our result analysis by looking at equity growth trajectory. By equity we mean the current size of the account plus the value of any currently open positions, relative to the initial account size. Fig. 11 shows the equity curve obtained by our system, with the bid price over the 7 years of testing for context.

**Table 6**
Trade-by-trade analysis. Note that trades from all 10 paths are included. Duration is in number of ticks.

| | Profitable trades | Unprofitable trades |
|---|---|---|
| Nr. | 21038 | 18249 |
| Avg. profit (%) | 0.215 ± 0.337 | −0.211 ± 29.9 |
| Avg. Duration | 33679 ± 96107 | 36066 ± 77264 |
| % Longs | 48.8 | 49.6 |

The equity curve is somewhat regular, with no extreme drawdowns at any point, which is made more clear with Fig. 12. The maximum drawdown is −9.9 ± 12.1%, without compounding, and −16.6 ± 20.0% with triannual compounding. These are relatively low values, albeit with a large uncertainty due to how it is propagated, accumulating from both the peak and the drought. This amount of drawdown would allow for a comfortable use of leverage $L = 2$, which would have doubled our final profit.

Since a Q-network is a black box system and markets are a mostly inscrutable environment it is difficult to interpret the system's performance fluctuations over the datasets. However, a closer look at Fig. 11 can provide some speculation. It seems there is a pattern of struggling when there is no overall trend to the price changes, when the market is sideways. This is clear during the year 2012, where equity barely rises. Also, in the years 2011 and 2015, while they are successful years overall, gains come from the beginning and the end of the year where there are clear trends. In the middle, where there is mostly sideways movement, there is little to no equity growth. Possibly, when the market has a clear trend there are distinguishing patterns in the input features that allow exploitation, while more indecisive periods result in noisier inputs.

This is exacerbated in the two final datasets, 2016 p.2 and 2016 p.3, which add to a mostly sideways moving market the sudden change in tick density by the broker. Due to how our system is designed, larger tick density leads to larger trading frequency. Thus, the system is forced to trade much more frequently than during training in a market context where it has been shown to


**Fig. 12.** Average drawdowns, uncertainty was not included for clarity. *X*-axis is in units of ticks.

**Table 7**
Simple and compounded total test profit for a mix of 5000 and 10,000 *time_skip*.

| Compounding frequency | Total Test Profit (%) | Yearly Avg. Test Profit (%) |
|---|---|---|
| None | $78.5 \pm 9.5$ | $11.2 \pm 1.4$ |
| Yearly | $109.1 \pm 17.8$ | $15.6 \pm 2.5$ |
| Triannual | $114.0 \pm 19.6$ | $16.3 \pm 2.8$ |

struggle, which unsurprisingly results in the largest losses of the whole testing period. It is only natural that upon realizing the change in tracking tick data by the broker, *time_skip* would be accordingly adjusted to maintain a trading frequency that had so far yielded results. An easy correction would be to increase the *time_skip* parameter to 10,000 once the larger tick frequency was detected. Doing so makes both those tests profitable, $2.5 \pm 2.2\%$ and $3.4 \pm 2.0\%$ respectively, and brings the results to those depicted in Table 7.

## 6. Conclusions

The challenges speculation trading present are completely different from those posed by environments such as Atari games or Go. Financial markets are not deterministic and the data on which the system bases its decisions is non-stationary and very noisy. Despite these challenges the main goals for this architecture were achieved. Learning in the training dataset is stable and it is apparent from a number of validation learning curves that the Q-network is indeed capable of finding relationships in financial data that translate to out-of-sample decision making. It was also shown that the model obtained from the training procedure can then be harnessed for profitable trading in a test dataset.

Overall, this approach has a great deal of potential to be explored. With more powerful optimization efforts for parameters such as *TW* for feature extraction, and design decisions such as network topology, weight initialization scheme (Glorot initialization [19] was briefly tested but with inconclusive results), choice of cost function and activation functions of the hidden layers, a performance boost could certainly be obtained without even changing the overall framework laid out in this paper. Topology in particular could be removed as a design decision by employing neuroevolution techniques [29]. Besides stronger optimization, there are two main weaknesses in this trading system that should merit further work:

- Training/validation/testing dataset size: a more interesting alternative based on identifying market regimes, for example, could be conjectured. Otherwise, our fixed size approach could be improved by exhaustive search rather than our limited informal testing.
- Model candidate selection: our approach to choose the model with highest validation performance was already shown to be overly simplistic when it became clear that the uncertainty of the model should have been taken into account. Moreover, rather than using just one candidate the trading account could be split among a number of candidates which would help dilute the risk through diversification.

Other than changes to the system itself, future work could focus on improving the financial facets of this work such as the use of other types of financial data and optimizing order execution.

## Acknowledgment

This work was supported in part by Fundação para a Ciência e a Tecnologia, Portugal (Project UID/EEA/50008/2013).

## References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller, Playing atari with deep reinforcement learning, 2013, arXiv preprint arXiv:1312.5602.
[2] V. Mnih, K. Kavukcuoglu, D. Silver, A.A. Rusu, J. Veness, M.G. Bellemare, A. Graves, M. Riedmiller, A.K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, D. Hassabis, Human-level control through deep reinforcement learning, Nature 518 (2015) 529–533.
[3] M. Krakovsky, Reinforcement renaissance, Commun. ACM 56 (8) (2016) 12–14.
[4] Bank for International Settlements, Foreign exchange turnover in april 2016, Triennial Central Bank Survey, 2016.
[5] H. van Hasselt, A. Guez, D. Silve, Deep reinforcement learning with double Q-learning, in: AAAI, vol. 2, Phoenix, AZ, 2016, p. 5.
[6] R.S. Sutton, A.G. Barto, Reinforcement Learning, An Introduction, MIT Press, 1998.
[7] C.J. Watkins, P. Dayan, Technical note: Q-learning, Mach. Learn. 8 (3) (1992) 279–292.
[8] C.-H. Park, S.H. Irwin, What do we know about the profitability of technical analysis? J. Econ. Surv. 21 (4) (2007) 786–826.
[9] Y.-W. Cheung, M. Chinn, Currency traders and exchange rate dynamics: a survey of the US market, J. Int. Money Finance 20 (4) (2001) 439–471.
[10] T. Gehrig, L. Menkhoff, Technical Analysis in Foreign Exchange - The Workhorse Gains Further Ground, Hannover economic papers (hep), Leibniz Universität Hannover, Wirtschaftswissenschaftliche Fakultät, 2003.
[11] J. Moody, M. Saffell, Learning to trade via direct reinforcement, IEEE Trans. Neural Netw. 12 (4) (2001) 875–889.
[12] C. Gold, FX trading via recurrent reinforcement learning, in: Computational Intelligence for Financial Engineering, 2003. Proceedings. 2003 IEEE International Conference on, IEEE, 2003, pp. 363–370.
[13] M.A.H. Dempster, V. Leemans, An automated FX trading system using adaptive reinforcement learning, Expert Syst. Appl. 30 (3) (2006) 543–552.
[14] J. Cumming, An Investigation into the Use of Reinforcement Learning Techniques within the Algorithmic Trading Domain (Master's thesis), Imperial College London, 2015.
[15] Y. Deng, F. Bao, Y. Kong, Z. Ren, Q. Dai, Deep direct reinforcement learning for financial signal representation and trading, IEEE Trans. Neural Netw. Learn. Syst. 28 (3) (2017) 653–664.
[16] P.-H. Hsu, Y.-C. Hsu, C.-M. Kuan, Testing the predictive ability of technical analysis using a new stepwise test without data snooping bias, J. Empir. Finance 17 (3) (2010) 471–484.
[17] Y. Bengio, Practical recommendations for gradient-based training of deep architectures, in: Neural Networks: Tricks of the Trade, Springer, 2012, pp. 437–478.
[18] Y.A. LeCun, L. Bottou, G.B. Orr, K.R. Müller, Efficient backprop, in: Neural Networks: Tricks of the Trade, Springer, 2012, pp. 9–48.
[19] X. Glorot, A. Bordes, Y. Bengio, Deep sparse rectifier neural networks, in: G.J. Gordon, D.B. Dunson (Eds.), Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS-11), vol. 15, Journal of Machine Learning Research - Workshop and Conference Proceedings, 2011, pp. 315–323.

[20] Q.V. Le, G. Brain, G. Inc, A Tutorial on Deep Learning Part 1: Nonlinear Classifiers and The Backpropagation Algorithm, 2015.
[21] L.-J. Lin, Reinforcement Learning for Robots Using Neural Networks (Ph.D. thesis), Carnegie Mellon University, Pittsburgh, PA, USA, 1992, UMI Order No. GAX93-22750.
[22] D.E. Rumelhart, G.E. Hinton, R.J. Williams, in: J.A. Anderson, E. Rosenfeld (Eds.), Neurocomputing: Foundations of Research, MIT Press, 1988, pp. 696–699.
[23] T. Tieleman, G. Hinton, Lecture 6.5-rmsprop: divide the gradient by a running average of its recent magnitude, COURSERA: Neural Networks for Machine Learning, 2012.
[24] T. Schaul, I. Antonoglou, D. Silver, Unit tests for stochastic optimization, 2013, arXiv preprint arXiv:1312.6055.
[25] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016.
[26] S. Ruder, An overview of gradient descent optimization algorithms, 2016, arXiv preprint arXiv:1609.04747.
[27] C.L. Giles, S. Lawrence, A.C. Tsoi, Noisy time series prediction using recurrent neural networks and grammatical inference, Mach. Learn. 44 (1) (2001) 161–183.
[28] J. Moody, Forecasting the economy with neural nets: A survey of challenges and solutions, in: Neural Networks: Tricks of the Trade, Springer, 1998, pp. 347–371.
[29] J. Nadkarni, R.F. Neves, Combining neuroevolution and principal component analysis to trade in the financial markets, Expert Syst. Appl. 103 (2018) 184–195.