Expert Systems With Applications 140 (2020) 112891

Contents lists available at ScienceDirect

Expert Systems With Applications

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

Continuous control with Stacked Deep Dynamic Recurrent Reinforcement Learning for portfolio optimization

Amine Mohamed Aboussalah, Chi-Guhn Lee*

Department of Mechanical and Industrial Engineering, University of Toronto, ON M5S 3G8, Canada

ARTICLE INFO

Article history:
Received 29 January 2019
Revised 22 July 2019
Accepted 19 August 2019
Available online 20 August 2019

Keywords:
Reinforcement learning
Policy gradient
Deep learning
Sequential model-based optimization
Financial time series
Portfolio management
Trading systems

ABSTRACT

Recurrent reinforcement learning (RRL) techniques have been used to optimize asset trading systems and have achieved outstanding results. However, the majority of the previous work has been dedicated to systems with discrete action spaces. To address the challenge of continuous action and multi-dimensional state spaces, we propose the so called Stacked Deep Dynamic Recurrent Reinforcement Learning (SDDRRL) architecture to construct a real-time optimal portfolio. The algorithm captures the up-to-date market conditions and rebalances the portfolio accordingly. Under this general vision, Sharpe ratio, which is one of the most widely accepted measures of risk-adjusted returns, has been used as a performance metric. Additionally, the performance of most machine learning algorithms highly depends on their hyperparameter settings. Therefore, we equipped SDDRRL with the ability to find the best possible architecture topology using an automated Gaussian Process ($\mathcal{GP}$) with Expected Improvement ($\mathcal{EI}$) as an acquisition function. This allows us to select the best architectures that maximizes the total return while respecting the cardinality constraints. Finally, our system was trained and tested in an online manner for 20 successive rounds with data for ten selected stocks from different sectors of the S&P 500 from January 1st, 2013 to July 31st, 2017. The experiments reveal that the proposed SDDRRL achieves superior performance compared to three benchmarks: the rolling horizon Mean-Variance Optimization (MVO) model, the rolling horizon risk parity model, and the uniform buy-and-hold (UBAH) index.

1. Introduction

The development of intelligent trading agents has attracted the attention of investors as it provides an alternative way to trade known as automated data-driven investment, which is distinct from traditional trading strategies developed based on microeconomic theories. The intelligent agents are trained by using historical data and a variety of Machine Learning (ML) techniques have been applied to execute the training process. Examples include Reinforcement Learning (RL) approaches that have been developed to solve Markov decision problems. RL algorithms can be classified mainly into two categories: actor-based (sometimes called direct reinforcement or policy gradient/policy search methods) (Baxter & Bartlett, 2001; Moody & Wu, 1997; Moody, Wu, Liao, & Saffell, 1998; Ng & Jordan, 2000; Williams, 1992) where the actions are learned directly, and critic-based (also known as value-function-based methods) where we directly estimate the value functions. The choice of a particular method depends upon the nature of the problem being addressed. One of the direct reinforcement techniques is called recurrent reinforcement learning (RRL) and it is presented as a methodology to solve stochastic control problems in finance (Moody & Wu, 1997). RRL has advantages of finding the best investment policy which maximizes certain utility functions without resorting to predicting price fluctuations and it is often incorporated with a neural network to determine the relationship (mapping) between historical data and investment decision making strategies. It produces a simple and elegant representation of the underlying stochastic control problem while avoiding Bellman's curse of dimensionality.

In the past, there have been several attempts to use a value-based reinforcement learning approach in the financial industry: a TD(λ) approach has been applied in finance (Van Roy, 1999) and Neuneier (1996) applied Q-Learning to optimize asset allocation decisions. However, such value-function methods are less-than-ideal for online trading due to their inherently delayed feedback (Moody & Saffell, 2001) and also because they imply having a discrete action space. Moreover, the Q-learning approach turns out to be more unstable compared to the RRL approach when presented with noisy data (Moody & Saffell, 2001). In fact, Q-learning algorithm is more sensitive to the value function

* Corresponding author.
E-mail addresses: amine.aboussalah@mail.utoronto.ca (A.M. Aboussalah), cglee@mie.utoronto.ca (C.-G. Lee).

https://doi.org/10.1016/j.eswa.2019.112891
0957-4174/

selection, while RRL algorithm offers more flexibility to choose between different utility functions that can be directly optimized such as profit, wealth, or risk-adjusted performance measures. A comparison study between Direct Reinforcement and Q-Learning methods for asset allocation was conducted by Moody and Saffell (2001). Moreover, Moody and Saffell (2001) and Deng, Kong, Bao, and Dai (2015) suggest an actor-based direct reinforcement learning that is able to provide immediate feedback of the market conditions to make optimal decisions. Therefore, it suits better than Q-learning with regard to the nature of market and dynamic trading. Another recent paradigm, Deep Q-Network designed initially to play Atari games (Mnih et al., 2015) inspired the deep Q-trading system which learns the Q-value function for the control problem (Wang et al., 2017). Other papers using deep RL in portfolio management have recently been published. Liang, Chen, Zhu, Jiang, and Li (2018) implemented three state-of-art continuous control RL algorithms. All of them are widely-used in game playing and robotic. Jiang, Xu, and Liang (2017) presented a financial model-free RL framework to provide a deep ML solution to the portfolio management problem using an online stochastic batch learning scheme. They introduced the concept of Ensemble of Identical Independent Evaluators topology and the Portfolio-Vector Memory. Zarkias, Passalis, Tsantekidis, and Tefas (2019) introduced a novel price trailing formulation, where the RL agent is trained to trail the price of an asset rather than directly predicting the future price. Zhengyao and Liang (2017) used a convolutional neural network (CNN) trading based approach with historic prices of a set of financial assets from a cryptocurrency exchange to output the portfolio weights. Recent applications of RRL in algorithmic trading succeed in single asset trading. Maringer and Ramtohul (2012) presented the regime-switching RRL model and described its application to investment problems. A task-aware scheme was proposed by Deng, Bao, Kong, Ren, and Dai (2016) to tackle vanishing/exploding gradient in RRL and Lu (2017) deploys long short-term memory (LSTM) to handle the same deficiency. Almahdi and Yang (2017) proposed a RRL method with a coherent risk adjusted performance objective function to obtain both buy and sell signals and asset allocation weights.

Multi-asset investment, also known as portfolio management, has a cardinality constraint that has to be satisfied as well, which requires portfolio weights to sum to one. Another major challenge concerns the return of investment, which is naturally path-dependent. Previous decisions drastically affect future decisions and therefore this brings us to the question of how to take advantage of the history of the previous decisions without losing in terms of time complexity.

To address these issues, we introduce the Stacked Deep Dynamic Recurrent Reinforcement Learning (SDDRRL) algorithm that takes multiple continuous investment actions for each asset while enforcing the cardinality constraint. We use a gradient clipping sub-task based Backpropagation Through (BPTT) to address the problem of vanishing gradients that may occur due to the presence of a memory gate responsible for taking into account the previous investment decisions into the new ones (Bengio, Simard, & Frasconi, 1994). Moreover, to find out how many past decisions should be incorporated into the model in order to compute the current optimal investment decisions without losing in terms of time efficiency, we define the concept of Time Recurrent Decomposition (TRD) that takes into account the temporal dependency. The number of time-stacks has been optimized by equipping the agent with the ability to find the best possible configuration of those time-stacks along with other hyperparameters using an automated Gaussian Process ($\mathcal{GP}$). Moreover, a noteworthy pattern emerges following the application of the automated $\mathcal{GP}$: the architectures presenting the best performances seem to present an hourglass shape topology (similar to autoencoders). Finally, another advantage of the proposed architecture is that it is by construction modular and perfectly deployable in real-time trading platforms. The remaining Sections are organized as follows: Section 2 describes the model formulation and Section 3 introduces the learning algorithm in more detail. Section 4 shows the experimental results including the Bayesian hyperparameter optimization, the distribution of portfolio weights generated by our algorithm and the performance comparison against some commonly used benchmarks. Finally, Section 5 concludes the article.

2. Formulation

The key framework of RRL is to find the optimal decisions $\delta_t(\boldsymbol{\theta})$ in order to maximize a specific utility function $U_T(.)$ that represents the wealth of investors. The simplest way is to directly maximize $U_T(.)$ over a time horizon period T:

$$\max_{\boldsymbol{\theta}} U_T(R_1, R_2, R_3, \ldots, R_T|\boldsymbol{\theta}) \tag{1}$$

where $\boldsymbol{\theta}$ denotes the optimal trading system parameters and R_t for $t \in \{1, 2, \ldots, T\}$ the realized returns. The optimization aims to determine the vector parameter $\boldsymbol{\theta}$ that gives the optimal decisions leading to a maximal utility.

2.1. Financial objective function

The dynamic nature of trading problems requires investors to make sequential decisions and each of these decisions will result in an instantaneous reward/return R_t. The accumulated rewards generated from the beginning up to the current time step T define an economic utility function $U_T(R_1, R_2, R_3, \ldots, R_T)$. In this context, a variety of financial objective functions have been developed and reported in the literature. By way of illustration, the most natural utility function used by risk-insensitive investors is the profit, which can be seen as the sum of total rewards. Others use logarithmic cumulative return instead to maximize their wealth. However, maximizing the cumulative return does not mitigate the unseen risks in the investment which is one of the major concerns of risk-averse investors. Alternatively, most modern fund managers would optimize the risk-adjusted return which is an indicator that refines returns by measuring how much risk is involved in producing that return. The Sharpe ratio (Sharpe, 1994) is one of the most widely accepted measures of risk-adjusted returns. The Sharpe ratio is also known as the reward-to-variability ratio. It measures how much additional return that will be received for the additional volatility of holding the risky assets over a risk-free asset. Under the setting of an investment with multiple periods, the Sharpe ratio is the average risk premium per unit of volatility in an investment:

$$U_T = \frac{\frac{1}{T}\sum_{t=1}^{T} R_t - r_f}{\sqrt{\frac{1}{T}\sum_{t=1}^{T} R_t^2 - (\frac{1}{T}\sum_{t=1}^{T} R_t)^2}} \tag{2}$$

where r_f denotes the risk-free rate of return and it is defined as being the theoretical rate of return of an investment with zero risk. The risk-free rate can be interpreted as the interest an investor would expect from an absolutely risk-free investment over a specified period of time. Since r_f is a constant in Eq. (2), it can be disregarded during the optimization phase (i.e. we consider $r_f = 0$ in the following Sections).

2.2. Portfolio optimization model

A signal that represents the current market condition will be fed into a neural network and pass through multiple layers and finally output the decision vector δ_t at time t. Suppose that we have

a set of assets $\{1, 2, \ldots, m\}$ indexed by i throughout the paper. By combining the suggestion from Deng et al. (2016) with the additional setting of a multi-asset portfolio, the input signal is defined as $f_t = \{f_{1,t}, f_{2,t}, f_{3,t}, \ldots, f_{m,t}\}$, where m is the total number of the assets considered in the portfolio optimization problem. f_t represents the current market conditions in which each element: $f_{i,t} = \{\Delta P_{i,t-a-1}, \Delta P_{i,t-a}, \ldots, \Delta P_{i,t}\} \in R^a$ indicates price changes within the decision making epoch. According to Merton (1969), price change is a movement independent of its history. Therefore, using price changes instead of prices themselves as input signals for the policy network improves the learning efficiency because it removes the trend from the signal and it makes the data appear stationary. However, in Deng et al. (2016), f_{t+1} is obtained simply by sliding forward one element in each signal $f_{i,t}$ and consequently there exists a significant overlap between f_t and f_{t+1}. This will drastically hinder the learning efficiency as it learns insignificant information from f_t to f_{t+1}. Therefore, we shrink the intervals between each feature in the signals. In our definition, the decision will be made hourly $t \in \{1, 2, \ldots, T\}$ where T is the total number of trading hours present in the dataset and each element in $f_{i,t}$ represents the 15-min price change of stock i and as a result $f_{i,t} \in R^4$. The overlap between signals are eliminated under this setting and the signal will be fed into a neural network so that the information it carries will be extracted gradually when it passes through the neural network layers.

Under the setting of a portfolio containing more than one asset, the investment decisions are represented by the vector $\delta_t = [\delta_{1,t}, \delta_{2,t}, \delta_{3,t}, \ldots, \delta_{m,t}] \in \mathbb{R}^{1\times m}$, with m being the total number of assets in the portfolio. In plain words, each element in δ_t represents the weight of an asset in the portfolio at time step t and $\delta_{i,t} \in [0,\ 1]$ for $i \in \{1, 2, \ldots, m\}$ as short-selling is disallowed to avoid infinite losses. The immediate return R_t is defined as follows:

$$R_t = \delta_{t-1} \cdot Z_t - c \sum_{i=1}^{m} \mid \delta_{i,t} - \delta_{i,t-1} \mid \tag{3}$$

where $Z_t = [Z_{1,t}, Z_{2,t}, Z_{3,t}, \ldots, Z_{m,t}]^T \in \mathbb{R}^{m\times 1}$, $Z_{i,t} = \frac{P_{i,t}}{P_{i,t-1}} - 1$ for $i \in \{1, 2, \ldots, m\}$ and $P_{i,t}$ is the price of asset i at time t. Therefore, $Z_{i,t}$ indicates the rate of price change within a trading period (one hour in our model) and c represents the transaction commission rate that is taken into consideration both during the training and testing periods. At each time step, rebalancing the portfolio will result in a transaction cost and it is subtracted from the investment returns. Moreover, each decision in the vector δ_t represents the weight of a stock in the portfolio and the cardinality constraint requires that $\sum_{i=1}^{m} \delta_{i,t} = 1$. It leads to a constrained optimization problem. One way to enforce the cardinality constraint is to apply a softmax transformation to the decision layer such that the constrained decisions become $\delta^c_{i,t} = \frac{\exp(\delta_{i,t})}{\sum_{j=1}^{m} \exp(\delta_{j,t})}$ (Moody et al., 1998). However, applying this transformation is equivalent of applying a multiclass discriminant function on the decision weights, which can drastically enlarge the difference between them. This could result in an undiversified portfolio and therefore subject to a higher risk when a precipitous drop in price is encountered. In addition, it requires an extra layer in our architecture which also makes the overall model more computationally expensive. Instead, the penalty method can be used to enforce the cardinality constraint. In addition, it is also known that using both L_2 regularization (Phaisangittisagul, 2016) along with dropout increases the accuracy (Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov, 2014). Thus, the portfolio optimization model can be formulated at time t as follows:

$$\begin{aligned}
\max_{\theta} \quad & U_t\big(R_1, R_2, R_3, \ldots, R_t|\boldsymbol{\theta}\big) - p\left(1 - \sum_{i=1}^{m} \delta_{i,t}\right)^2 \\
& -\beta \sum_{l=1}^{n} \sum_{k=1}^{N_l} w_{l,k}^2 \\
\text{s.t.} \quad & R_t = \delta_{t-1} \cdot Z_t - c \sum_{i=1}^{m} \mid \delta_{i,t} - \delta_{i,t-1} \mid \\
& \delta_t = sigmoid(W^{(n)} h_{n-1} + b^{(n)} + \nu \odot \delta_{t-1}) \\
& h_{n-1} = ReLU(W^{(n-1)} h_{n-2} + b^{(n-1)}) \\
& h_{n-2} = ReLU(W^{(n-2)} h_{n-3} + b^{(n-2)}) \\
& \ldots \\
& h_1 = ReLU(W^{(1)} f_t + b^{(1)})
\end{aligned} \tag{4}$$

where:

- $\odot$ represents the element-wise multiplication symbol;
- N_l denotes the number of neurons in a given layer l;
- $W^{(l)} = [w^{(l)}_{i,j}] \in \mathbb{R}^{N_l \times N_{l-1}}$ is the weight matrix and $b^{(l)} \in \mathbb{R}^{N_l}$ the bias vector for layer l;
- $\boldsymbol{\theta} = \{(W^{(1)}, b^{(1)}), \ldots, (W^{(n)}, b^{(n)}, \nu)\}$ represents the trading parameters of the policy network.

In our work, we didn't use dropout since we don't have very deep neural networks. The improvement that we show in the paper is due to L2-regularization only (weigh decay), where $w_{l,k}$ is defined as being the weight connecting the neuron present in the lth layer, kth position. The recurrent part in (4) is due to the presence of a memory gate at the decision layer δ_t responsible for taking into account the previous investment decisions into the new ones. The penalty coefficient p and regularization coefficient β were treated as hyperparameters. The penalty term penalizes the utility function U_t whenever the cardinality constraint is unsatisfied and the magnitude of the penalty can be controlled by the penalty coefficient p. The advantage of trying the penalty method is that it is universally applicable to any equality or inequality constraints (round-lot, asset class, return, cardinality etc.) and fits for more advanced portfolio optimization approaches. However, the addition of this penalty term to our objective function gives no guarantee that the cardinality constraint will be respected. Thus to avoid any deficiency risk in our portfolio, we decided to add a normalization layer after the decision layer as shown in Fig. 1. To bypass the use of a softmax function, each weight was simply divided by the sum of all the decision weights. The Table 2 in Section 4 present a detailed comparative study of the performance obtained with the five best online architectures that we found.

3. SDDRRL architecture

As aforementioned, the backpropagation when we have recurrent structures is slightly different from the regular one since the computation of δ_t requires δ_{t-1} as an extra input at the decision layer. One simple way to backpropagate the flow of information through the network is BPTT. For instance at time step T, the gradient of the objective function w.r.t. θ is obtained by the chain rule:

$$\frac{\partial U_T}{\partial \theta} = \sum_{t=1}^{T} \frac{\partial U_T}{\partial R_t} \left(\frac{\partial R_t}{\partial \delta_t} \frac{\mathrm{d}\delta_t}{\mathrm{d}\theta} + \frac{\partial R_t}{\partial \delta_{t-1}} \frac{\mathrm{d}\delta_{t-1}}{\mathrm{d}\theta} \right) \tag{5}$$

As $\delta_t = sigmoid(W^{(n)} h_{n-1} + b^{(n)} + \nu \odot \delta_{t-1})$, the previous decision serves as an input to the calculation of the current decision, therefore,

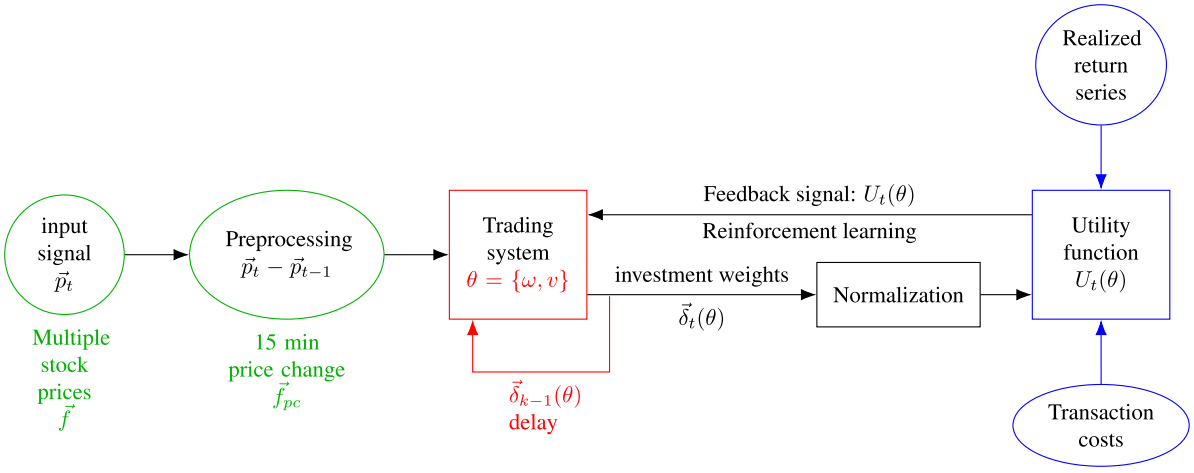

Fig. 1. SDDRLL process.

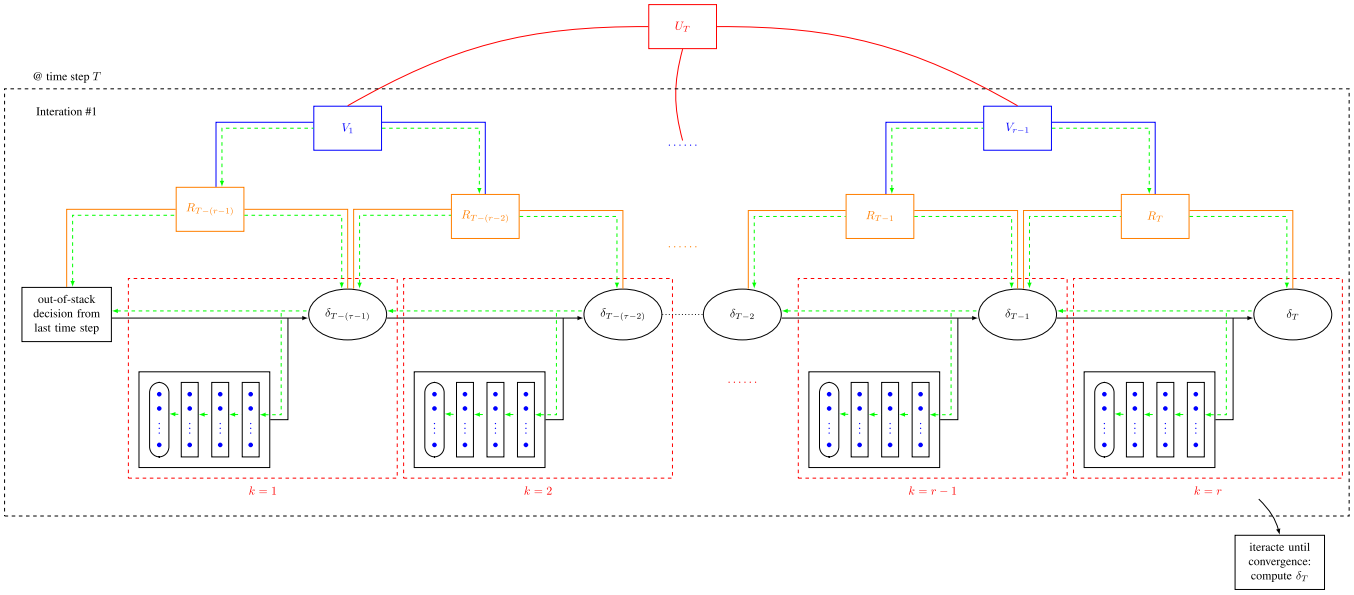

Fig. 2. SDDRRL training phase. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

$$\frac{\mathrm{d}\delta_t}{\mathrm{d}\theta} = \nabla_\theta \delta_t + \frac{\partial \delta_t}{\partial \delta_{t-1}} \frac{\mathrm{d}\delta_{t-1}}{\mathrm{d}\theta} \tag{6}$$

The above Eqs. (5) and (6) assume differentiability of the trading decision function δ_t. According to the chain rule, the gradient at the current state involves the partial derivative w.r.t all decisions from the beginning to the current step and the calculation needs to be recursively evolved. By taking recurrence into account, the gradient is expressed as follows:

$$\begin{aligned}\frac{\partial U_T}{\partial \theta} = \sum_{t=1}^{T} \frac{\partial U_T}{\partial R_t}\left(\frac{\partial R_t}{\partial \delta_t}\left(\frac{\partial \delta_t}{\partial \theta} + \frac{\partial \delta_t}{\partial \delta_{t-1}}\left(\frac{\partial \delta_{t-1}}{\partial \theta} + \frac{\partial \delta_{t-1}}{\partial \delta_{t-2}}\left(\frac{\partial \delta_{t-2}}{\partial \theta} \cdots \frac{\mathrm{d}\delta_1}{\mathrm{d}\theta}\right)\right)\right)\right.\\ \left. + \frac{\partial R_t}{\partial \delta_{t-1}}\left(\frac{\partial \delta_{t-1}}{\partial \theta} + \frac{\partial \delta_{t-1}}{\partial \delta_{t-2}}\left(\frac{\partial \delta_{t-2}}{\partial \theta} + \frac{\partial \delta_{t-2}}{\partial \delta_{t-3}}\left(\frac{\partial \delta_{t-3}}{\partial \theta} \cdots \frac{\mathrm{d}\delta_1}{\mathrm{d}\theta}\right)\right)\right)\right)\end{aligned} \tag{7}$$

Eqs. (5)–(7) imply that the further we go back in time, the less impact previous decisions would have on the current one. This observation is consistent with the vanishing gradient issue in recurrent neural networks. In addition, unfolding the entire memory is computationally expensive especially at large time steps.

In the typical recurrent reinforcement learning (RRL) approach, the training of the neural network requires the optimization of U_T, in which all trading decisions δ_t for $t \in \{1, 2, \ldots, T\}$ need to be adjusted accordingly to the new market conditions. However, old decisions are not as influential as new market conditions when it comes to making a new decision. Therefore, we introduce the concept of Time Recurrent Decomposition (TRD) that takes into account the necessary temporal dependency by stacking RRL structures as shown in Fig. 2, resulting in the Stacked Deep Dynamic Recurrent Reinforcement Learning (SDDRRL). In SDDRRL, we re-optimize only the recent decisions instead of those in the entire history, and the number of time-stacks (denoted as τ) specifies how many recent decisions should influence the current decision. That is, the number of stacks τ is the level of time dependency of U_T on previous decisions. For instance, $\tau = 2$ means there are two time-stacks, i.e. the current decision δ_T needs to be computed and the most recent decision δ_{T-1} needs to be adjusted.

Consider the portfolio optimization problem given in Eq. (4) at time T. The optimization problem is decomposed into τ tasks $\{V_1, V_2, \ldots, V_\tau\}$ where $V_k, k \in \{1, 2, \ldots, \tau\}$ is defined in Eqs. (8)–(11),

and V_k is assigned to a time-stack k. At time-stack k, V_k is optimized to find the optimal decision $\delta_{T-(\tau-k)}$. To optimize $\delta_{T-(\tau-k)}$, we consider only the terms of the objective function U_T involving $\delta_{T-(\tau-k)}$, and we compute the gradient to update the parameter vector θ. The task V_k is a combination of two components that includes $\delta_{T-(\tau-k)}$: (1) the transaction cost in $R_{T-(\tau-k)}$; (2) the realized return in $R_{T-(\tau-k)+1}$.

$$V_k = \widetilde{U}_T\left(R_{T-(\tau-k)}, R_{T-(\tau-k)+1}\right) - p\left(1 - \sum_{i=1}^{m} \delta_{i,T-(\tau-k)}\right)^2 - \beta \sum_{l=1}^{n} \sum_{k=1}^{N_l} w_{l,k}^2 \tag{8}$$

where

$$\widetilde{U}_T\left(R_{T-(\tau-k)}, R_{T-(\tau-k)+1}\right) = U_T\left(\ldots, R_{T-(\tau-k)}, R_{T-(\tau-k)+1}, \ldots\right) \tag{9}$$

with

$$R_{T-(\tau-k)} = \delta_{T-(\tau-k)-1} \cdot Z_{T-(\tau-k)} - c \sum_{i=1}^{m} \left|\delta_{i,T-(\tau-k)} - \delta_{i,T-(\tau-k)-1}\right| \tag{10}$$

and

$$R_{T-(\tau-k)+1} = \delta_{T-(\tau-k)} \cdot Z_{T-(\tau-k)+1} - c \sum_{i=1}^{m} \left|\delta_{i,T-(\tau-k)+1} - \delta_{i,T-(\tau-k)}\right| \tag{11}$$

$\widetilde{U}_T(R_{T-(\tau-k)}, R_{T-(\tau-k)+1})$ represents the component in the utility function U_T involving only the realized returns $R_{T-(\tau-k)}$ and $R_{T-(\tau-k)+1}$. The other realized returns that do not involve the computation of $\delta_{T-(\tau-k)}$ are considered fixed. Once $\delta_{T-(\tau-k)}$ is computed by the updated parameters from backpropagation, it will be fed into the next time-stack to compute the optimal $\delta_{T-(\tau-k)+1}$ which is assigned with the task V_{k+1}. The pseudocode Algorithm 1 summarizes the training phase and Fig. 2 illustrates the SDDRRL training process. The utility function U_T at time T is decomposed into $V_1, V_2, \ldots, V_\tau$, which are assigned to stacks $1, 2, \ldots, \tau$. The red lines connect $V_1, V_2, \ldots, V_\tau$ from U_T indicating the time-stack decomposition. The yellow lines show how instantaneous returns are defined with investment decisions. The green dotted lines show gradient information from decomposed tasks all the way down to multi-layered neural network in time-stacks, which is boxed by red dotted rectangles, to perform backpropagation. It is necessary to point out that the decision prior to the first RRL block ($k = 1$) is taken from the last time step (*i.e.* if we are currently maximizing U_T for example, then out-of-stack decision will be from the last time step U_{T-1}). However, it is problematic that we will need to foresee the information coming from $\delta_{T-(\tau-k)+1}$ to perform backpropagation at time-stack k.

At this time-stack label, $\delta_{T-(\tau-k)}$ is learned by solving the optimization problem defined in Eq. (4) for the task V_k as defined in (8). It is impossible in practice to explicitly know the next decision when the computation of the current decision is still undergoing. Therefore, instead of foreseeing the next decision magically, we can approximate the value of it by assuming the next decision $\delta_{T-(\tau-k)+1}$ will remain the same as $\delta_{T-(\tau-k)}$, which could be interpreted as temporarily canceling the transaction cost at the first iteration ($i = 1$). This is a conservative assumption often used in approximate dynamic programming methods (Powell, 2011) when the next signal f_{t+1} is temporarily absent at time step t. Afterward, $\delta_{T-(\tau-k)}$ is generated and it will flow to the next time-stack to compute $\delta_{T-(\tau-k)+1}$. After the first iteration for all time-stacks (i.e. when $k = \tau$), the system will switch to the second iteration ($i = 2$) and repeat the calculation of $\delta_{T-(\tau-k)}$ starting one more time from the first time-stack $k = 1$ up to the last time-stack $k = \tau$, then we move on to the next optimizer iteration and the process repeats itself until convergence. However, when $i = 2$, the approximated value of $\delta_{T-(\tau-k)+1}$ at the time-stack k will be replaced by the explicit results of $\delta_{T-(\tau-k)+1}$ from last iteration ($i = 1$). By doing that, the future market situation is captured during the training phase from the last optimizer iteration and helps to learn the correct decision at the current time-stack by taking future information into account. Therefore, besides the first iteration ($i = 1$), the value of next decisions will be approximated by the explicit computed values $\delta_{T-(\tau-k)+1}$ from previous iterations $i \in \{2, \ldots, N\}$. In addition, the estimation of the next decision at $k = \tau$ is unnecessary since

Algorithm 1: Training algorithm for SDDRRL.

assign:
  values to α, γ_1, γ_2, ϵ, T, N and τ
initialize:
  $\theta_0 \sim \text{Normal}(0, 1)$, $m_0 = 0$, $v_0 = 0$, $i = 0$, $\delta_{last} = 0$
    /* holdings before investment */
while $t \le T$ **do** /*iterate in all time steps*/
  while $i \le N$ **do** /*iterate until converge*/
    $i \leftarrow i + 1$
    while $k \le \tau$ **do**
      /*iterate in all time-stacks*/
      if $k = 1$ **then**
        $\delta_{previous} = \delta_{last}$
          /* from last time step */
            /* if out-of-stack */
      else
        $\delta_{previous} = \delta_{T-(\tau-k)-1}$
          /* from last time-stack */
      if $i = 1$ **then**
        $\delta_{next\_approx} - \delta_{T-(\tau-k)} = 0$
            /* assume unchanged */
              /* next decision */
      else
        $\delta_{next\_approx} = \delta_{T-(\tau-k)+1}$
      determine the task $V_k(\theta_{i-1}, \delta_{previous}, \delta_{next\_approx})$
      $g_i = \nabla_\theta V_k(\theta_{i-1}, \delta_{previous}, \delta_{next\_approx})$

      $m_i = \gamma_1 m_{i-1} + (1 - \gamma_1) g_i$
      $v_i = \gamma_2 v_{i-1} + (1 - \gamma_2) g_i^2$
      $\alpha_i = \alpha \dfrac{\sqrt{1 - \gamma_2^i}}{1 - \gamma_1^i}$
      $\theta_i = \theta_{i-1} - \alpha_i \dfrac{m_i}{\sqrt{v_i} + \epsilon}$

      compute $\delta_{T-(\tau-k)}(\theta_i)$
        /* by forward-propagating the learned weights */
      compute $\delta^{norm}_{T-(\tau-k)}(\theta_i)$
        /* by normalizing the decision vector */
      return θ_i, $\delta^{norm}_{T-(\tau-k)}(\theta_i)$
      $k \leftarrow k + 1$
  $\delta_{last} = \delta^{norm}_{T-(\tau-1)}(\theta_N)$
    /* update out-of-stacks decision for next time step */

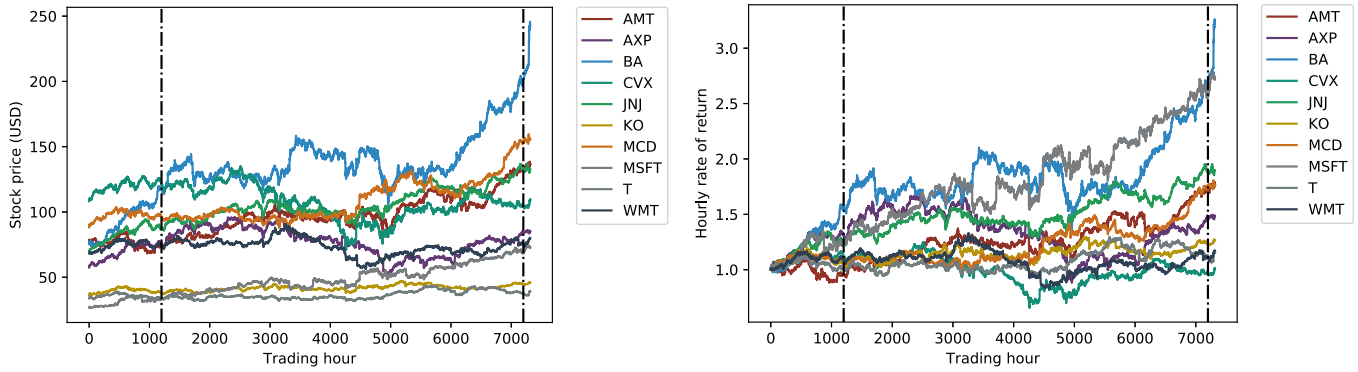

Fig. 3. (Left): The hourly price change for the 10 companies in our portfolio during 7316 trading hours (entire dataset window). (Right): The hourly rate of return. The period between the two vertical dashed lines represent the testing window: 6000 trading hours (3.5 years).

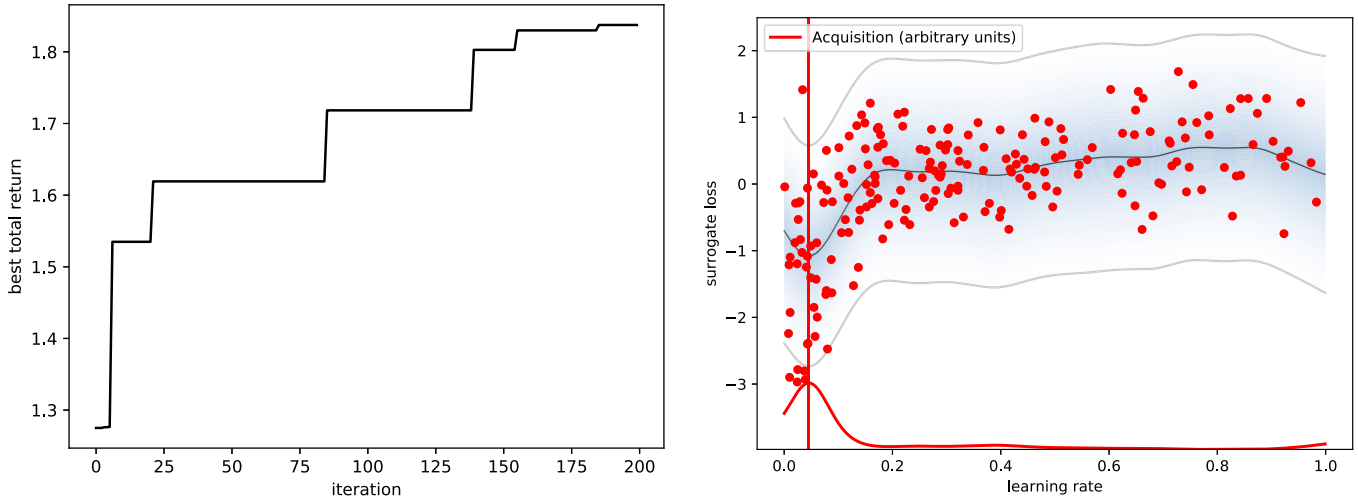

Fig. 4. Left: Convergence plot of the total return with respect to the number of $\mathcal{GP}$ iterations. Right: Acquisition function (red curve) guiding the sampling of the learning rate using Gaussian surrogate loss. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

δ_T will be the last one. Hence, the task assigned to δ_T is:

$$V_\tau = \widetilde{U}_T(R_T) - p\left(1 - \sum_{i=1}^{m} \delta_{i,T}\right)^2 - \beta \sum_{l=1}^{n} \sum_{k=1}^{N_l} w_{l,k}^2 \tag{12}$$

where

$$R_T = \delta_{T-1} \cdot Z_T - c \sum_{i=1}^{m} |\delta_{i,T} - \delta_{i,T-1}| \tag{13}$$

4. Experiments

4.1. Dataset

In our experiments, the investment decisions are made hourly and each element in the input signal f_t represents the per 15-minute price changes within the trading hour. SDDRRL is trained and tested through twenty successive rounds. In each round, the training section covers 1200 trading hours while the testing section covers the next 300. Due to this testing mechanism, the testing periods are made short because the trained system will be mostly effective for short periods. In the first round, SDDRRL is trained for the first 1200 trading hours and tested from the trading hour 1200 to 1500. In the next round, the training and testing data are shifted 300 trading hours forward (*i.e.* the second training round starts from hour 300 to 1500 and the second testing round from hour 1500 to 1800) and it will move ahead in this way for the rest of the rounds. In fact, the volatility of the market is a major concern in most of ML-based trading systems. Models trained with historical data are not effective on testing periods since the new market conditions are not learned in the trained model. Therefore, SDDRRL is trained and tested in an online manner so that the model can quickly adapt to the new market conditions. It should be noted that our test periods include the transaction costs. We used the typical cost due to bid-ask spread and market impact that is 0.55%. We believe these are reasonable transaction costs for the portfolio trades. For each round, SDDRRL will be trained with 1200 trading hour data points and when the testing period starts, the last signal during testing will be added as an input to the system and the parameters will be updated to adapt the most recent market conditions. However, the size of the training and testing windows should be optimized in order to potentially obtain better results. We have developed SDDRRL as a new architecture combining deep neural networks with recurrent reinforcement learning and tailored specifically for multi-asset portfolios. In this sense, future investigation of the size of the training and testing windows should be made.

The SDDRRL model is evaluated on a portfolio consisting of the following ten selected stocks: American Tower Corp. (AMT), American Express Company (AXP), Boeing Company (BA), Chevron Corporation (CVX), Johnson & Johnson (JNJ), Coca-Cola Co (KO), McDonald's Corp. (MCD), Microsoft Corporation (MSFT), AT&T Inc. (T)

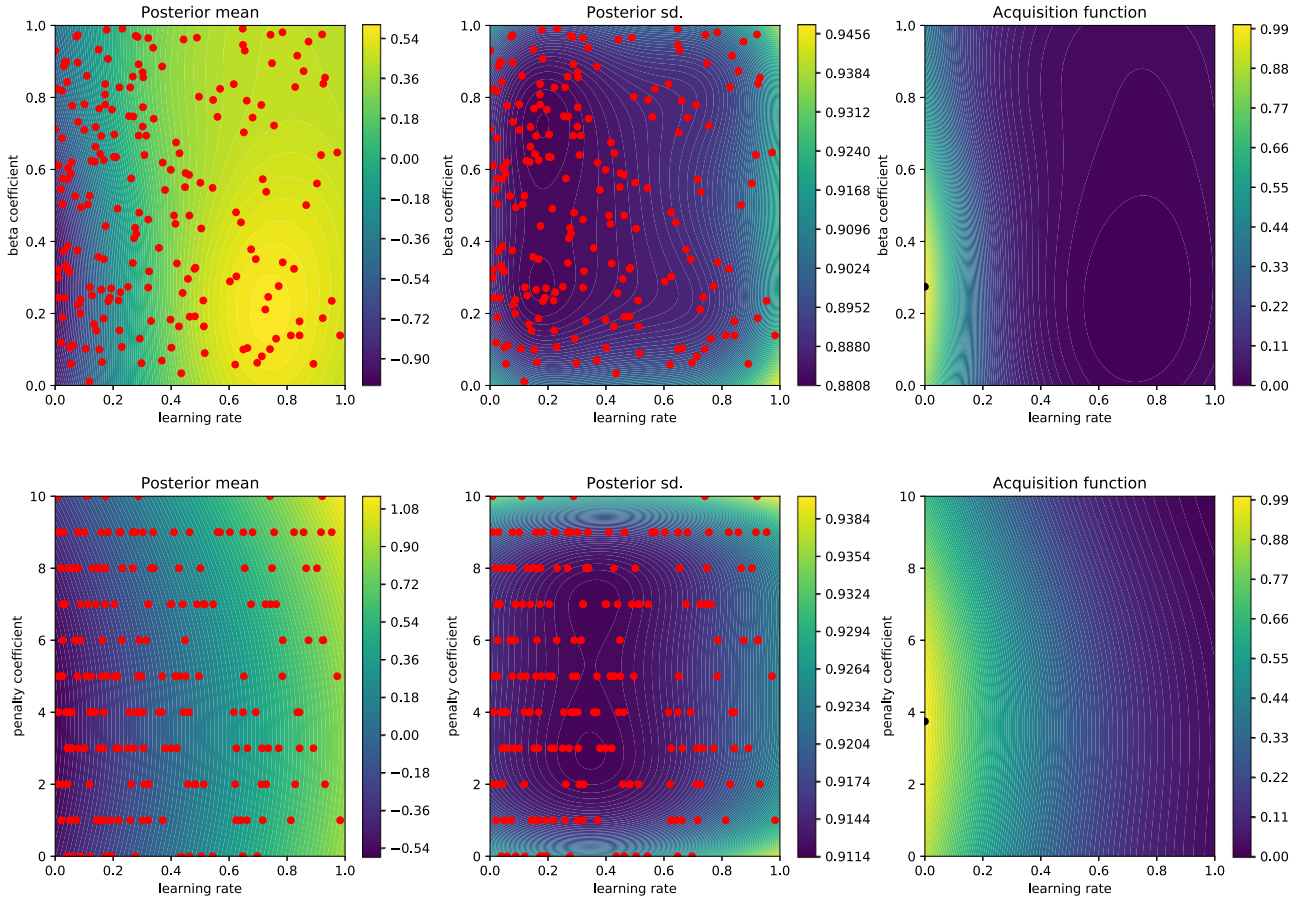

Fig. 5. Loss surfaces of the Posterior mean, the Posterior standard deviation (sd) and the Acquisition function for: **beta coefficient** VS **learning rate** and **penalty coefficient** VS **learning rate**.

and Walmart Inc. (WMT). To promote the diversification of the portfolio, these stocks are selected from different sectors of S&P 500, so that they are uncorrelated as much as possible as shown in Fig. 3.

The data source comes from finam[1] database that has intraday data for 42 of the most liquid US stocks on BATS Global Markets.[2] Our dataset starts from January 1st 2013 up to July 31st 2017, resulting in 7928 trading hours. It should be noted that there was missing data for the 10 stocks at different time periods, which were cleaned in our raw data. Therefore, instead of having 7928 trading hours, there are still 7316 data points. This represents a duration of 4 years and 7 months (4.58333 years), which ultimately comes down to approximately 1596.2 trading points per year, *i.e.* 245.57 trading days per year, thus, 6.5 trading hours per day.

Experiments were run on a 40-core machine with 384GB of memory. All algorithms were implemented in Python using Keras and Tensorflow libraries. Each method is executed in an asynchronously parallel set up of 2–4 GPUs, that is, it can evaluate multiple models in parallel, with each model on a single GPU. When the evaluation of one model finishes, the methods can incorporate the result and immediately re-deploy the next job without waiting for the others to finish. We use 20 K80 (12GB) GPUs with a budget of 10 h.

[1] https://www.finam.ru/profile/moex-akcii/gazprom/export.
[2] http://markets.cboe.com/.

4.2. Bayesian optimization for hyperparameter tuning

Many optimization problems in ML are black box optimization problems due to the unknown nature of the objective function $f(\boldsymbol{x})$. If the objective function were inexpensive to evaluate, then we could sample at many points *e.g.* via grid search, random search or numeric gradient estimation where we explore the space of hyperparameters without any prior knowledge about the configurations seen before. If it were instead expensive, as is typical with tuning hyperparameters of deep neural networks in a time-sensitive application such as finance, then it would become crucial to minimize the number of samples drawn from the black box function f. This is where Bayesian Optimization ($\mathcal{BO}$) techniques are most useful. They attempt to find the global optimum in a minimum number of steps.

$\mathcal{BO}$ incorporates prior belief about f and updates the prior with samples drawn from f to get a posterior that better approximates f. The model used for approximating the objective function is called surrogate model. One of the most popular surrogate models for $\mathcal{BO}$ is the Gaussian Process (GP). $\mathcal{BO}$ also uses an acquisition function that guides sampling in the search space to areas where an improvement over the current best observation is likely (right plot in Fig. 4).

Automatic hyperparameter tuning methods aim to construct a mapping between the hyperparameter settings and model performance in order to rationally sample the next configuration of hyperparameters. The paradigm of automatic hyperparameter tuning belongs to a class known as Sequential Model-Based Optimization (SMBO) (Hutter, Hoos, & Leyton-Brown, 2011). SMBO algorithms

Table 1
Range of hyperparameters.

Parameters	Search space	Type
Learning rate	0–1	Float
Number of stacks	1–10	Integer
Number of units per layer	10–2000	Integer
Regularization coefficient : β	0–1	Float
Penalty coefficient : p	0–10	Integer
Optimizer	0 : Adam 1 : Adadelta 2 : Adagrad	Categorical

mainly differ in the way they take into account historical observations to model either the surrogate function or some kind of transformation applied on top of it. They also differ in the way they apply derivative-free methods while optimizing those surrogates. For our work, we used the $\mathcal{BO}$ approach which is a subclass of SMBOs. It was shown that $\mathcal{BO}$ can outperform human performance on many benchmark datasets (Snoek, Larochelle, & Adams, 2012) and its standard design is described below:

1. The surrogate is modeled by a Gaussian Process ($\mathcal{GP}$): $f(\boldsymbol{x}) \sim \mathcal{GP}(\mu(\boldsymbol{x}), k(\boldsymbol{x}, \boldsymbol{x'}))$.
   In other words, f is a sample from a $\mathcal{GP}$ with mean function μ and covariance function k and $\boldsymbol{x}$ represents the best set of hyperparameters we are looking for. Here we used a Gaussian kernel as a dissimilarity measure in the sample space: $k(\boldsymbol{x}, \boldsymbol{x'}) \propto \exp(-\frac{\|\boldsymbol{x}-\boldsymbol{x'}\|^2}{2\sigma^2})$, where σ^2 is a parameter that reflects the degree of uncertainty in our model.
2. To find the next best point to sample from f, we will choose the one that maximizes an acquisition function. One of the most popular acquisition functions is of Expected Improvement type ($\mathcal{EI}$), which represents the belief that new trials will improve upon the current best configuration. The one with the highest $\mathcal{EI}$ will be tried next. It is defined as: $\mathcal{EI}(\boldsymbol{x}) = \mathbb{E}[\max(0, f(\boldsymbol{x}) - f(\hat{\boldsymbol{x}})]$, where $\hat{\boldsymbol{x}}$ is the current optimal set of hyperparameters. Maximizing $\mathcal{EI}(\boldsymbol{x})$ informs about the region from which we should sample in order to gain the maximum information about the location of the global maximum of f.

The hyperparameters for the SDDRRL architecture listed in Table 1 were optimized using $\mathcal{BO}$. We use GPyOpt (2016) python routine version 1.2.1 to implement the $\mathcal{BO}$. The optimization was initialized with 25 random search iterations followed by up to 150 iterations of standard $\mathcal{GP}$ optimization, where the total return is used as the surrogate function and $\mathcal{EI}$ as the acquisition function. The results are reported in the left plot in Fig. 4 showing that after only few iterations, we are able to get a total return of 15.65%. Random search then boosts very quickly the total return up to 53.59% after only 18 iterations and thus remains until the end of the random search cycle (iteration #25). Using $\mathcal{GP}$, we can show that we constantly improve our process of searching for the best architecture that maximizes the overall return. In our case, we stopped at iteration #200 but nothing prevents us from exploring even more the configuration of the hyperparameter space. The goal of this Section is just to illustrate the methodology. At iteration #200, we find that the best architecture gives 94.71% total return and the top-5 architectures give more than 78.77% total return. It should be noted that with the $\mathcal{GP}$, the agent comes very quickly to probe the region of interest that maximizes the total return and targets remarkably the most sensitive hyperparameters that leads to the best architecture. Next, the right plot in Fig. 4 shows the acquisition function in red, the red dots show the history of the points that have already been explored and the surrogate function with $\pm 96\%$ confidence.

In order to keep an eye on what the agent is doing with $\mathcal{BO}$, it is interesting to compute the loss surface as a function of the hyperparameters as shown in Fig. 5. Essentially, the agent was able to probe most of the hyperparameter configuration space and this gives us precisely an estimate of where the true optimum of the loss surface is located.

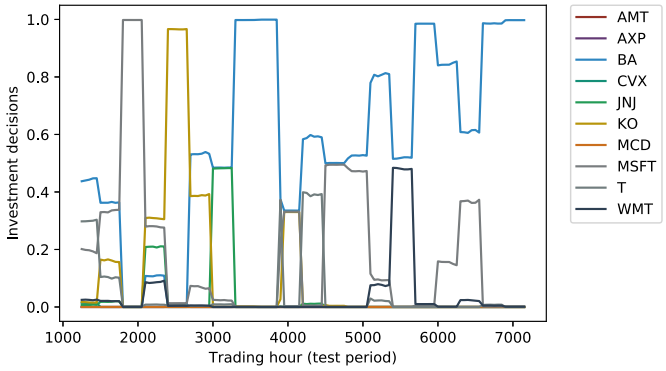

Fig. 6. The distribution of portfolio weights of the best SDDRRL trading agent over the test period.

In particular, an interesting key element deserves to be pointed out. The architectures that give the best performance have to a certain extent several features in common, such as the learning rate (almost the same order of magnitude), the number of time-stacks approximately being 5 or 6, and the type of optimizer. But the most important characteristic is that almost all of them typically present an hourglass shape architecture (similar to denoising autoencoder) as a neural network candidate for SDDRRL (Table 2). This result opens up a new avenue of investigation. Indeed, it would be interesting to understand why such a characteristic emerges essentially when processing non-deterministic and non-stationary signals such as financial data. An attempt for answering this question would be the subject of our next work. In addition, our experiments have shown that when $p = 0$, the trader agent no longer respects the cardinality constraint, while when $\beta = 0$, we overfit, which results in a poor performance due to the lack of generalization capacity. The Fig. 6 shows the optimal decision weights for the best SDDRRL architecture over the test period.

4.3. Alternative active trading strategies

4.3.1. Rolling horizon mean-variance optimization model

The mean-variance optimization (MVO) framework has been proposed by Markowitz (1952, 2010). It is a quantitative tool traditionally used in dynamic portfolio allocation problem where risk and return are traded off. In order to make portfolios' performance comparable we use the same risk-adjusted measure of return used in SDDRLL that is Sharpe ratio (reward-to-variability ratio). As defined above in Section 2.1, Sharpe ratio measures the excess return per unit of risk (deviation) in an investment asset or a portfolio. The MVO problem using Sharp ratio can be written in a general way as follows:

$$\begin{aligned} \max_{x \in \mathbb{R}^m} \quad & \frac{\mu^T x - r_f}{\sqrt{x^T Q x}} \\ \text{s.t.} \quad & \sum_i x_i = 1 \\ & l \leq Ax \leq u \\ & x \geq 0 \end{aligned} \tag{14}$$

where,

1. μ, the vector of mean returns.
2. Q, the covariance matrix.
3. $\sum_i x_i = 1$; (cardinality constraint).
4. $l \leq Ax \leq u$, (other linear constraints if needed).

Table 2
Best top five architectures.

Learning rate	Number of stacks	Number of units in each layer	Regularization coefficient	Penalty coefficient	Optimizer	Absolute return	Annualized return
0.081	6	1160 - 780 - 580 - 1420	0.597	2	Adam	78.77%	18.05%
0.024	5	870 - 160 - 320 - 370	0.809	9	Adam	82.90%	18.83%
0.021	10	790 - 1730 - 140 - 1410	0.767	9	Adam	88.01%	19.77%
0.075	7	400 - 1410 - 1390 - 1920	0.154	9	Adam	90.81%	20.27%
0.063	6	940 - 1810 - 1020 - 1730	0.158	7	Adam	94.71%	20.97%

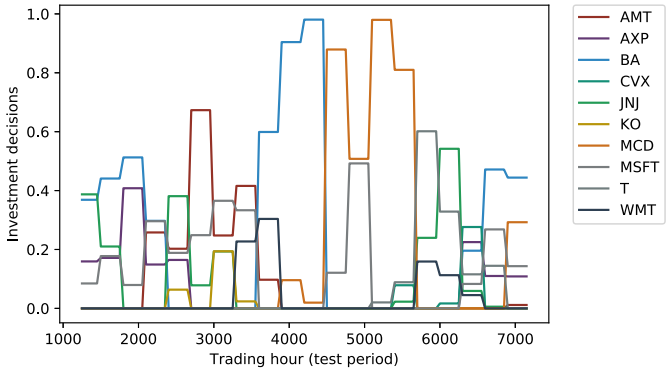

Fig. 7. The distribution of portfolio weights of the rolling horizon MVO model over the test period.

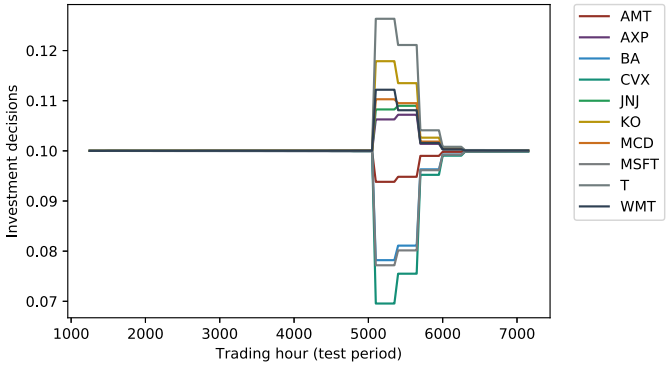

Fig. 8. The distribution of portfolio weights of the rolling horizon ERC model over the test period.

5. $x \geq 0$, portfolio weight vector.[3]
6. r_f, risk-free rate of return.

The solution of the optimization problem above is difficult to obtain because of the nature of its objective: (1) non-linear, (2) possibly non-convex. However, under a reasonable assumption,[4] it can be reduced to a standard convex quadratic program (Cornuejols & Tutuncu, 2006):

$$\begin{aligned} \min_{y \in \mathbb{R}^m, \kappa \in \mathbb{R}} \quad & y^T Q y \\ \text{s.t.} \quad & \sum_i (\mu_i - r_f) y_i = 1 \\ & \sum_i y_i = \kappa \\ & l.\kappa \leq Ay \leq u.\kappa \\ & \kappa \geq 0 \end{aligned} \tag{15}$$

The optimal solution of the problem (14) can be written according to the solution of the problem (15) as follows: $x^* = \frac{y}{\kappa}$.

As in SDDRRL case, the rolling horizon version of the problem (15) is considered using 20 successive rounds: the first round uses 1200 trading hours to estimate the portfolio decision weights and the next 300 h to test the trading strategy. In the next round, the training and testing data are shifted 300 trading hours forward and it will move ahead likewise for the rest of the rounds. The "ILOG CPLEX Optimization Studio" (IBM, 1988) has been used to solve (15). The Fig. 7 shows the optimal decision weights for the rolling horizon MVO model.

4.3.2. Risk parity model

The risk parity portfolio model is analogous to the equal weights "1/m" portfolio, but from a risk perspective. It attempts to diversify risk by ensuring each asset contributes the same level of risk. Risk Parity is sometimes referred to as "Equal Risk Contribution" (ERC). The complete risk parity optimization model is formulated as a least-squares approach that minimizes the difference of the following terms:

$$\begin{aligned} \min_{x \in \mathbb{R}^m} \quad & \sum_i \sum_j (x_i (Qx)_i - x_j (Qx)_j)^2 \\ \text{s.t.} \quad & \sum_i x_i = 1 \\ & l \leq Ax \leq u \\ & x \geq 0 \end{aligned} \tag{16}$$

where x is the portfolio weight vector, $x_i(Qx)_i$ represents the individual risk contribution of asset i and Q the covariance matrix.

The optimization problem (16) was solved successively along 20 rounds[5] using "Interior Point OPTimizer (IPOPT)" (Wächter & Biegler, 2006), which is a software library for large scale nonlinear optimization of continuous systems. The distribution of the optimal weights of the rolling horizon ERC model over the test period is shown in the Fig. 8.

4.4. Performance analysis

The performance of SDDRRL is evaluated based on the realized rate of returns of the 5 best architectures over the testing period horizon (approximately 3.5 years). SDDRRL performance is compared with three benchmarks: the rolling horizon MVO model, the rolling horizon risk parity model, and the uniform buy-and-hold (UBAH) strategy with initially equal-weighted setting among 10 stocks. In Fig. 9, we notice that even the 5 architectures do not have the same total return, it seems like they unanimously agreed on the investment policy towards the end of the testing horizon. This fact demonstrates that the five architectures do not earn equally during the same period or under the same market conditions, and therefore, they can be seen as five separate experts with five different investment strategies. The best online SDDRRL architecture achieves a total return of 94.71% compared to

[3] $x \geq 0$ means that short-selling is disallowed.

[4] There exists a vector x satisfying the constraints (3)–(5) in (14) such that: $\mu^T x - r_f > 0$. In other terms, we assume that our universe of assets is able to beat the risk-free rate of return.

[5] We use 1200 trading hours for training and 300 h to test the trading strategy in the first round, then we apply a sliding window of 300 trading hours forward until the last round.

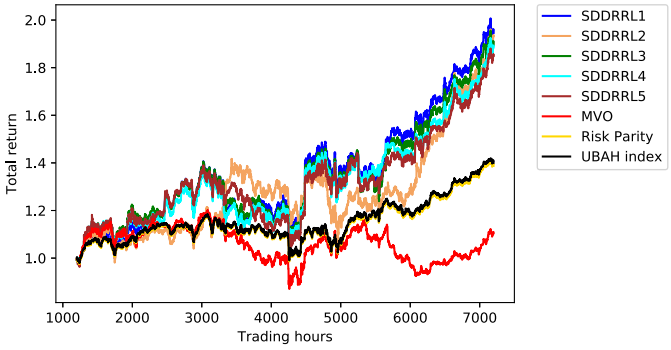

Fig. 9. Benchmarking the Top 5 SDDRRL's.

our benchmarks: 39.8% for the UBAH index, 38.7% the rolling horizon ERC, and 10.4% for the rolling horizon MVO. The poor performance of the rolling horizon MVO model is mainly due to sensitivity to transaction costs. We used the typical transaction cost of 0.55% due to bid-ask spread similar to what we used for SDDRRL. The only difference to report in the case of rolling horizon MVO is that the rebalancing operation for our portfolio happens in every 300 trading hour data points, *i.e.* after each sliding window, leading to only twenty portfolio rebalancing operations in total. Besides, MVO model makes the overly restrictive assumption of independently and identically distributed (IID) returns across different periods.

Moreover, it is clear from Fig. 9 that most of the experts had some difficulty to stand out from the market performance at the first trading hours. This was predictable since the agents had learned their investment policy during the first training cycle that took place in 2013, the year in which the S&P500 index posted a performance of 29.60%, which corresponds to the 4th best performance in the history of the S&P500. The following years have seen a significant drop 11.39% in 2014 and −0.73% in 2015. This has inevitably affected the online trading dynamics of our portfolio as can be seen in the range between 3000 and 4000 trading hours corresponding to these particular years. In fact, the majority of stocks in our portfolio show a devaluation during those periods. In other words, the five experts were trying to replicate the decision rules learned in 2013 during the ensuing testing period when market conditions drastically changed, which implies they were applying sub-optimal policies. But through the online learning cycle, as time progresses, the experts receive feedback on the market condition and allow gradual adjustment on their learned investment policy. Therefore, the experts clearly begin to gain the upper hand and perform well relatively to our benchmarks. In any case, it should be noted that the SDDRRL trading experts do not perform worse than the benchmarks. Based on these facts, SDDRRL investment decisions can be manually centralized by selecting the best investment strategy at the given trading time t or by adding an additional agent responsible for centralizing trading decisions by giving the right hand action to the best expert at each trading time period. The choice of one or the other is left to the discretion of the reader. One can also think about using advanced boosting techniques as surveyed by Zhou (2012) or the rainbow integrated agent developed recently by Hessel et al. (2017) to convert selected good learners into a better one. Indeed, the existence of different investment strategies that are not duplicated especially during periods of recession is a central point for a better reliability of the portfolio. In this way, it will be easier for us to avoid strategies that fail during specific volatile periods, which will be reflected on the total return at the end of the horizon.

The hourly rate of return presented in Fig. 3 shows that BA and MSFT have good Return on Investment (ROI) towards the end of the test period. This fact has been reflected in Fig. 6 where the SDDRRL trader agent is more likely to put more weight on these two stocks during the same period. This fact was also reflected in the total return as shown in Fig. 9.

5. Conclusions and future work

To the best of our knowledge, this is the first attempt to multi-asset dynamic and continuous control using deep recurrent neural networks with customized architectures. A gradient clipping subtask based Backpropagation Through Time has been used to avoid the vanishing gradient information problem and a Bayesian optimization technique has been deployed to effectively probe the hyperparameter space in order to estimate the set of hyperparameter values that lead to the maximum utility function while respecting the cardinality constraint. As a consequence, hourglass shape architectures (similar to auto-encoder) emerge and appear to be a natural choice for this kind of applications. Still, it would be interesting to investigate why such a pattern seems to be an appropriate choice and to examine whether there is a particular connection with non-stationary signals more generally. The optimized number of time-stacks was found to be approximately equal to 5 or 6, leading to annualized returns around 20% throughout our testing period. However, the size of the training and testing windows should be optimized and was left for future work.

Moreover, this procedure does not require any time series predictions which makes SDDRRL architecture relatively robust to price fluctuations. Also, SDDRLL is modular so it can be used with different neural network models such as Convolutional Neural Networks (ConvNets), Long Short-Term Memory (LSTMs) units, Gated Recurrent Units (GRUs) or any combination of these. A more comprehensive study using these models deserves to be done for comparison purposes.

Additionally and as illustrated above, different policy neural network architectures have different investment strategies over the same period of time, which could be interpreted as having five different portfolio management experts. By aggregating the decisions coming from these experts, we can be more robust in the face of market fluctuations. One way to do this would be to use techniques coming from the Ensemble Methods literature, namely, Bagging (Breiman, 1996), Boosting (Zhou, 2012), Bayesian parameter averaging (BPA) (Hoeting, Madigan, Raftery, & Volinsky, 1999) or Bayesian model combination (BMC) (Monteith, Carroll, Seppi, & Martinez, 2011) to combine high quality architectures. Another potential way to boost the overall performance would be to consider a Multi-Armed Bandits (MAB) approach (Auer, Cesa-Bianchi, & Fischer, 2002; Katehakis & Veinott, 1987) operating in a non-stationarity environment. The aim would be to select the right expert each time when we interact with the market.

Declaration of Competing Interest

None.

Credit authorship contribution statement

Amine Mohamed Aboussalah: Data curation, Conceptualization, Methodology, Project administration, Writing - review & editing, Writing - original draft. **Chi-Guhn Lee:** Project administration, Resources, Writing - review & editing, Supervision.

Acknowledgments

The authors are very grateful to Zixuan Wang and to Yassine Yaakoubi for their help and constructive comments regarding this work. This research is supported by the Fonds de recherche du Québec - Nature et technologies (FRQNT). 210036.

References

Almahdi, S., & Yang, S. Y. (2017). An adaptive portfolio trading system: A risk-return portfolio optimization using recurrent reinforcement learning with expected maximum drawdown. *Expert Systems with Applications, 87*, 267–279.

Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine Learning, 47*, 235–256.

Baxter, J., & Bartlett, P. L. (2001). Infinite-horizon gradient-based policy search. *Journal of Artificial Intelligence Research, 15*, 319–350.

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks, 5*(2), 157–166.

Breiman, L. (1996). Bagging predictors. *Machine Learning, 24*, 123–140.

Cornuejols, G., & Tutuncu, R. (2006). *Optimization methods in finance*. Cambridge.

Deng, Y., Bao, F., Kong, Y., Ren, Z., & Dai, Q. (2016). Deep direct reinforcement learning for financial signal representation and trading. *IEEE Transactions on Neural Networks and Learning systems, 28*(3), 653–664.

Deng, Y., Kong, Y., Bao, F., & Dai, Q. (2015). Sparse coding-inspired optimal trading system for HFT industry. *IEEE Transactions on Industrial Informatics, 11*(2), 467–475.

GPyOpt (2016). A Bayesian optimization framework in python. http://github.com/SheffieldML/GPyOpt.

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., et al. (2017). Rainbow: Combining improvements in deep reinforcement learning. In *AAAI conference on artificial intelligence*.

Hoeting, J. A., Madigan, D., Raftery, A. E., & Volinsky, C. T. (1999). Bayesian model averaging: A tutorial. *Statistical Science, 14*, 382–401.

Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization* (pp. 507–523).

IBM (1988). ILOG CPLEX optimization studio. https://www.ibm.com/ca-fr/products/ilog-cplex-optimization-studio.

Jiang, Z., Xu, D., & Liang, J. (2017). A deep reinforcement learning framework for the financial portfolio management problem. arXiv:1706.10059.

Katehakis, M., & Veinott, A. F. (1987). The multi-armed bandit problem: Decomposition and computation. *Mathematics of Operations Research, 12*, 262–268.

Liang, Z., Chen, H., Zhu, J., Jiang, K., & Li, Y. (2018). Adversarial deep reinforcement learning in portfolio management. arXiv:1808.09940.

Lu, D. W. (2017). Agent inspired trading using recurrent reinforcement learning and LSTM neural networks. arXiv:1707.07338.

Maringer, D., & Ramtohul, T. (2012). Regime-switching recurrent reinforcement learning for investment decision making. *Computational Management Science, 9*, 89–107.

Markowitz, H. (1952). Portfolio selection. *The Journal of Finance, 7*, 77–91.

Markowitz, H. (2010). Portfolio theory as I still see it. *Annual Review of Financial Economics, 2*, 1–23. https://doi.org/10.1146/annurev-financial-011110-134602.

Merton, R. C. (1969). Lifetime portfolio selection under uncertainty: The continuous-time case. *The Review of Economics and Statistics, 51*(3), 247–257.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M. G., et al. (2015). Human-level control through deep reinforcement learning. *Nature, 518*, 529–533.

Monteith, K., Carroll, J. L., Seppi, K., & Martinez, T. (2011). Turning Bayesian model averaging into Bayesian model combination. In *Proceedings of the international joint conference on neural networks IJCNN'11*.

Moody, J., & Saffell, M. (2001). Learning to trade via direct reinforcement. *IEEE Transactions on Neural Networks, 12*(4), 875–889.

Moody, J., & Wu, L. (1997). Optimization of trading systems and portfolios. *Decision Technologies for Financial Engineering*, 23–35.

Moody, J., Wu, L., Liao, Y., & Saffell, M. (1998). Performance functions and reinforcement learning for trading systems and portfolios. *Journal of Forecasting, 17*, 441–470.

Neuneier, R. (1996). Optimal asset allocation using adaptive dynamic programming. *Advances in neural information processing systems*.

Ng, A., & Jordan, M. (2000). PEGASUS: A policy search method for large MDPs and POMDPs. In *Proceedings of the sixteenth conference on uncertainty in artificial intelligence*.

Phaisangittisagul, E. (2016). An analysis of the regularization between l_2 and dropout in single hidden layer neural network. *Intelligent systems, modelling and simulation (ISMS)*.

Powell, W. B. (2011). Approximate dynamic programming: Solving the curses of dimensionality. *Wiley series in probability and statistics* (2nd ed.).

Sharpe, W. F. (1994). The Sharpe ratio. *The Journal of Portfolio Management, 21*(1), 49–58.

Snoek, J., Larochelle, H., & Adams, R. P. (2012). Practical Bayesian optimization of machine learning algorithms. In *IPS proceedings of the 25th international conference on neural information processing systems: 2* (pp. 2951–2959).

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research, 15*, 1929–1958.

Van Roy, B. (1999). Temporal-difference learning and applications in finance. In *Conference, computational finance*.

Wang, Y., Wang, D., Zhang, S., Feng, Y., Li, S., & Zhou, Q. (2017). Deep q-trading. *CSLT Technical Report-20160036*.

Williams, R. (1992). Simple statistical gradient-following algorithms for. connectionist reinforcement learning. *Machine Learning, 8*, 229–256.

Wächter, A., & Biegler, L. (2006). On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming. *Mathematical Programming, 106*, 25–57.

Zarkias, K. S., Passalis, N., Tsantekidis, A., & Tefas, A. (2019). Deep reinforcement learning for financial trading using price trailing. In *International conference on acoustics, speech and signal processing (ICASSP)*. IEEE.

Zhengyao, J., & Liang, J. (2017). Cryptocurrency portfolio management with deep reinforcement learning. In *Intelligent systems conference (IntelliSys)*. IEEE.

Zhou, Z.-H. (2012). *Ensemble methods: Foundations and algorithms*. Chapman & Hall/CRC.