Machine Learning Algorithms for High-Frequency Trading

Ege Yilmaz

Abstract

In this thesis Reinforcement Learning (RL) is utilized to train randomized neural networks, namely Echo State Networks (ESNs) and Extreme Learning Machines (ELMs) in a High-Frequency Trading (HFT) setting. The choice of the learning framework and the neural architectures is supported by both theory and experiment. Our RL agents trade on the mid-prices of the financial services company Garanti BBVA's stocks from the year 2017, obtaining positive net returns. By introducing artificial transaction costs to the reward signals, the validation performances of the models are improved. An ELM surpasses other models by paying the half of the bid-ask spread and gaining a positive net return.

Contents

1 Introduction

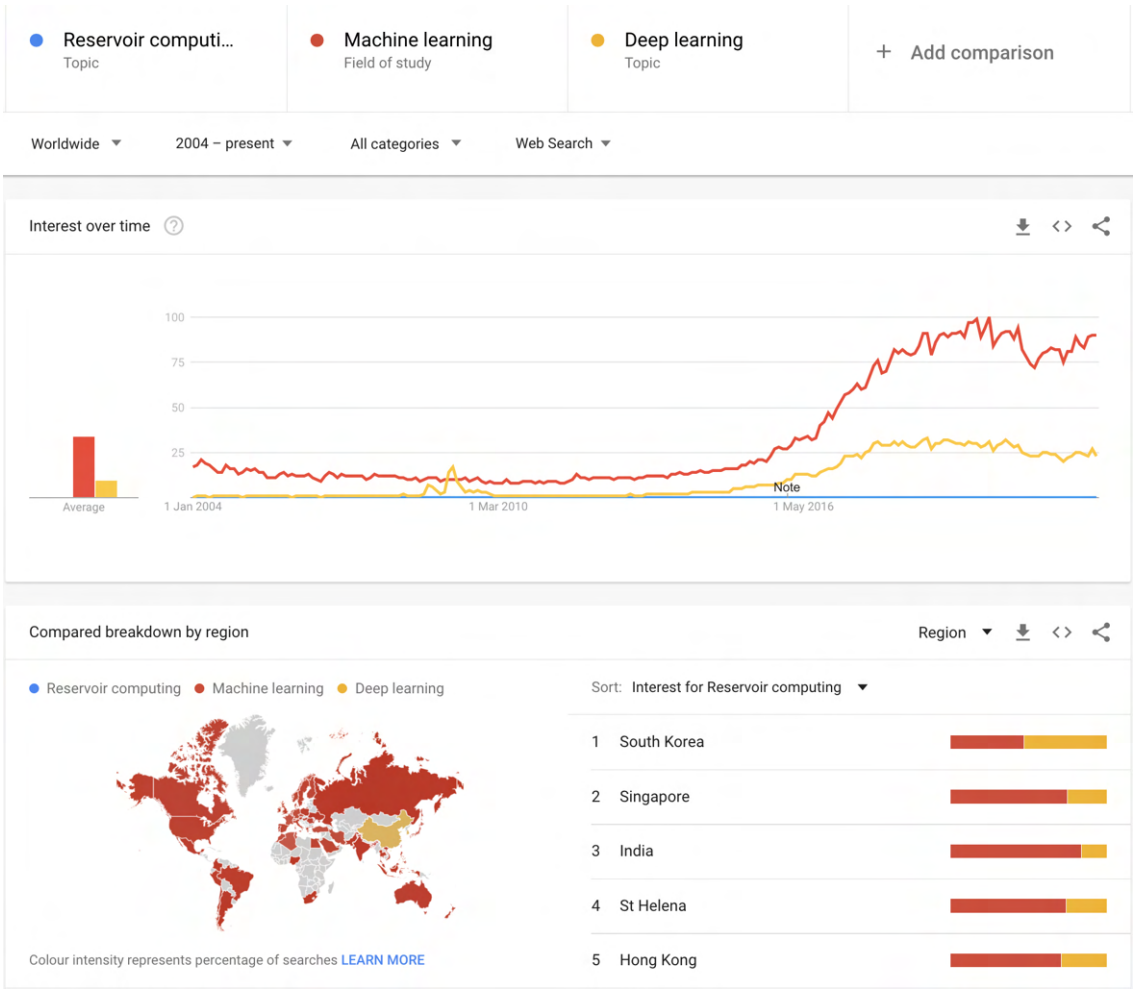
Artificial Intelligence (AI) is one of the most vibrant technical terms in the 21st century. It is an attractive field of study, since it has the promise of changing our lives for the better. With its vast application areas such as cybersecurity, automation services, marketing, automotive, agriculture, gaming, human resource, healthcare, finance, manufacturing and retail, it has been a driving force in the Fourth Industrial Revolution. According to Verified Market Research, the global AI market size was valued at USD 51.08 Billion in 2020 and is projected to reach USD 641.30 Billion by 2028, growing at a compound annual growth rate of 36.1% from 2021 to 2028. Machine learning (ML) as a major contributor to AI, has been trending in both academy and industry due to the increasing abundance of high quality data and availability of the systems to run the ML algorithms. A major part of active research in ML deals with optimization and generalization, which is utilizing data at hand to increase out of sample performance of the models. The approaches in this context can be divided into three categories, namely supervised learning, unsupervised learning and RL. In our study supervised learning and RL are covered in relation to various use cases. In particular, popular deep learning (DL) architectures are juxtaposed against randomized ML paradigms employed on financial data.

Feedforward Neural Networks (FNNs) struggle with processing complex time-varying signals, since they operate on fixed-size rolling windows. They have to work with the data contained in the window, which is not ideal for signals with different rates of change and longer term dependencies. By contrast, recurrent neural networks (RNNs) contain nodes with cyclical connections, which act as activations of inputs from a previous time step to influence predictions at the current time step. The internal states of the network holding these activations are then actually keeping long-term temporal information. This mechanism allows RNNs to operate on a dynamically changing contextual window over the input history rather than a static one [104]. Today's predominant RNN choices are Long Short-Term Memory (LSTM) [48] and Transformer [121] architectures.

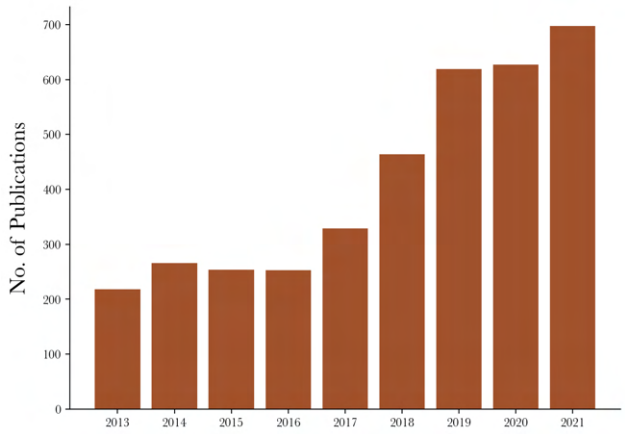
Reservoir computing (RC) presents an important playing ground belonging to both recurrent and randomized ML schemes, which lies at the heart of the algorithmic aspect of this study. It has been chosen as a major focus in our study due to its speed, ease of use and remarkable out of sample performance, specifically at tasks requiring short-term memory. Unlike today's RNN choices, RCs are computationally cheap, yet in many tasks they outperform fully trained RNNs in terms of speed and accuracy [73]. This is also shown in Section 2.3. Although rising in popularity, RC is yet to become one of the most popular choices in ML such as DL (see Figure 1), but we start to see its use in enhancing state of the art RNN architectures such as Reservoir Transformers [111]. Theory and specifications of RC is provided in Section 2, where the reader is presented with different RC architectures, universality properties and experimental results. Examples provided in Section 2.3 include comparisons to FNNs and RNNs consisting of the popular recurrent units LSTMs in various areas including financial time series forecasting (FTSF). Subsequently, theory and applications of RL is given in Section 3. Here, the emphasis is on different RL training schemes regarding RC. On the basis of empirical evidence, the algorithmic choices are justified, which make up the backbone of the HFT setup used in obtaining the main results of this study. In

Section 4, discussion of these results are presented after providing background on HFT, the chosen data and methodologies. It is intended to be an overarching segment on bringing RC and RL paradigms together in quest of successful algorithms trading on mid-prices of the financial services company Garanti BBVA's stocks in 2017 and gaining excess returns.

The back end used for obtaining the results is available at [34]. A summary of the road map, results and demonstrations with code written in Jupyter Notebooks can be found on the website [33] of this study. Lastly, the RC software written for the study can be found in [29] with its documentation on [30] and in Appendix E. All code has been written in the programming language Python [120] from scratch except for the implementation of the RL algorithm Proximal Policy Optimization (PPO) in Stable Baselines 3 library [103], which has been then customized for our purposes (see Sections 3 and 4). Other libraries that we have used are NumPy [46] for mathematical operations on arrays, pandas [83] for handling of data frames, scikit-learn [98] for Linear and Ridge Regression, OpenAI Gym for RL environments, tqdm [24] for progress bars, Optuna [5] for hyperparameter optimization, Matplotlib [53, 35] for plots, TensorFlow [81] for construction of RNNs with LSTMs, Pytorch [97] and torchinfo [129] for gradient-based optimization and model summaries respectively.



(a) Worldwide popularity of Google Search queries for RC (blue), ML (red) and DL (yellow) over a time period between 2004 and present. Taken from Google Trends on 14.04.22.



(b) Annual number of publications on RC from 2013 to 2021. Data taken from Elsevier's abstract and citation database Scopus on 14.04.22.

Fig. 1: Popularity of reservoir computing according to Google Trends and Scopus.

2 Reservoir Computing

A reservoir computer is a type of RNN characterized by the following two equations:

$$\mathbf{x}_t = F(\mathbf{x}_{t-1}, \mathbf{z}_t) \tag{2.1}$$
$$\mathbf{y}_t = h(\mathbf{x}_t) \tag{2.2}$$

The equation (2.1) describes the *state-space transformation* by the *reservoir map* $F\colon \mathbb{R}^N \times \mathbb{R}^n \to \mathbb{R}^N$, where $\mathbf{x}_t \in \mathbb{R}^N$ is the *state vector* with dimension $N \in \mathbb{N}$, which is referred to as the *size of the reservoir* or the *number of nodes* in the reservoir layer, $\mathbf{z}_t \in \mathbb{R}^n$ the $n \in \mathbb{N}$ dimensional input signal and t usually denotes the time step. The equation (2.2) describes the *readout* by the *output map* $h\colon \mathbb{R}^N \to \mathbb{R}^d$, where $\mathbf{y}_t \in \mathbb{R}^d$ is the $d \in \mathbb{N}$ dimensional output of the system. In other words, a reservoir computer is an input/output (I/O) system transforming (or filtering) a real infinite dimensional discrete-time input $\mathbf{z} = (..., \mathbf{z}_{-1}, \mathbf{z}_0, \mathbf{z}_1, ...) \in (\mathbb{R}^n)^{\mathbb{Z}}$ into a real infinite discrete-time output $\mathbf{y} = (..., \mathbf{y}_{-1}, \mathbf{y}_0, \mathbf{y}_1, ...) \in (\mathbb{R}^d)^{\mathbb{Z}}$ following the above equations.

A summary of different reservoir computer constructions as universal approximants is given in Section 2.1. Some of the realized RC architectures are introduced in Section 2.2 with an emphasis on the ESN [61], which is one of the chosen architectures to carry out the RC tasks in the study. A discussion on its universality is also given in Section 2.1, where the choice of RC and in particular ESN as randomized technology for approximating nonlinear dynamical systems is justified. Although they are not RNNs, we would like to cover ELMs [50] in the context of RC, which also utilize random projections as ESNs do [17]. Demonstration of the performance of ESNs is provided in Section 2.3 with comparisons to LSTMs and FNNs on concrete examples defined on finite dimensional spaces.

2.1 Universality of Reservoir Computers

In this section, we lay out universal families of reservoir computer systems. A family of transformations is said to be *universal* if its elements can approximate a class of transformations with an arbitrary accuracy [43]. An example for such a family would be polynomials approximating continuous functions by Weierstrass theorem. Uniform approximation theorems for dynamical systems similar to their static versions [26, 49] date back to Volterra and Fréchet involving Volterra series under compactness assumptions on the input space and time interval. The generalization to infinite time intervals was carried out by Boyd and Chua [12]. They overcome the compactness requirements due to the use of Stone-Weierstrass theorem in the previous works by introducing the so-called *fading memory property* (FMP), which specifies an operator fed with input signals which are close, i.e. their peak deviation is small, in the recent past but not necessarily close in the remote past, yielding present outputs which are close. FMP is the building block of the universality theorems for RC, since it relates to a unique steady state property [22] of the reservoir, where the reservoir states are asymptotically independent of their initial values (see Theorem 6 in [12]).

The proofs of the universality results shared below are left out and the reader is referred to the original sources of the theorems for these. The aim of this section is to convey the

story of the evolution of the universality results in RC ending at a strong point, where the choice of RC and the specific architecture to obtain the main results of the study is justified.

2.1.1 Causal Time-Invariant Fading Memory Filters

We share the universality theorems by Grigoryeva and Ortega [44] for fading memory reservoir computers or *reservoir filters* (RFs), after some necessary definitions and the formulation of the FMP of the respective filter family.

We set the output dimension to 1 and define a filter U as a map of the type $U\colon (D_n)^{\mathbb{Z}} \to \mathbb{R}^{\mathbb{Z}}$, where $(D_n)^{\mathbb{Z}}$ denotes the infinite sequences with elements in the subset $D_n \subset \mathbb{R}^n$. $(D_n)^{\mathbb{Z}^+}$ and $(D_n)^{\mathbb{Z}_-}$ denote right and left infinite sequences in the same subset, respectively. Maps of type $H\colon (D_n)^{\mathbb{Z}} \to \mathbb{R}$ are referred to as *functionals*. A filter U is *causal* if $U(\mathbf{z})_t = U(\mathbf{w})_t$ for all $\mathbf{z}, \mathbf{w} \in (D_n)^{\mathbb{Z}}$ with $\mathbf{z}_\tau = \mathbf{w}_\tau$ for all $\tau \leq t$ for a given $t \in \mathbb{Z}$. This means that the filter yields the same output (at t) for inputs, which share the same past (before t). A filter U is called *time-invariant* (TI) if it commutes with the *time delay operator* $U_{\tau\in\mathbb{Z}}$, defined by $U_\tau(\mathbf{z})_t \coloneqq \mathbf{z}_{t-\tau}$, so $U_\tau \circ U = U \circ U_\tau$. This means that any temporal shift of the input by some amount τ causes a temporal shift of the output by the same amount τ [76]. In Theorem 1, the functional is used instead of the causal TI filter owing to the existence of a bijection between the two [12, 44]. Hence, the *functional associated to the filter* U, $H_U\colon (D_n)^{\mathbb{Z}_-} \to \mathbb{R}$ is defined. Filters determined by the reservoir map are denoted as $U^F\colon (D_n)^{\mathbb{Z}} \to (D_N)^{\mathbb{Z}}$ with $U^F(\mathbf{z})_t \coloneqq \mathbf{x}_t \in D_N$ and filters determined by the entire reservoir system described by (2.1) & (2.2) are denoted as $U_h^F\colon (D_n)^{\mathbb{Z}} \to \mathbb{R}^{\mathbb{Z}}$ with $U_h^F(\mathbf{z})_t \coloneqq h(U^F(\mathbf{z})_t) = y_t$. U^F and U_h^F are causal and TI [44]. The *reservoir functional* $H_h^F \coloneqq (D_n)^{\mathbb{Z}_-} \to \mathbb{R}$ associated to U_h^F is determined by $H_h^F \coloneqq H_{U_h^F}$. Closed ball with radius M with respect to the Euclidean norm $\|\cdot\|$ centered at $\mathbf{0} \in \mathbb{R}^n$ is denoted as $\overline{B_n(\mathbf{0}, M)}$. The symbol $|||\cdot|||$ is reserved for the norms of operators or functionals defined on infinite dimensional spaces, where

$$K_M \coloneqq \{\mathbf{z} \in (\mathbb{R}^n)^{\mathbb{Z}_-} \mid \|\mathbf{z}_t\| \leq M, \forall t \in \mathbb{Z}_-\} = \overline{B_n(\mathbf{0}, M)}^{\mathbb{Z}_-}, \quad M > 0 \tag{2.1.3}$$

$$|||U|||_\infty \coloneqq \sup_{\mathbf{z}\in K_M} \{\|U(\mathbf{z})\|_\infty\} = \sup_{\mathbf{z}\in K_M} \left\{ \sup_{t\in\mathbb{Z}_-} \{\|U(\mathbf{z})_t\|\} \right\} \tag{2.1.4}$$

$$|||H|||_\infty \coloneqq \sup_{\mathbf{z}\in K_M} \{\|H(\mathbf{z})\|\}. \tag{2.1.5}$$

Lastly, $\mathbf{L}(V, W)$ denotes the set of continuous linear mappings from V to W, where V, W are Banach spaces.

Definition 1. ***Weighted Norms***
Let $w\colon \mathbb{N} \to (0, 1]$ be an arbitrary strictly decreasing sequence with zero limit such that $w_0 = 1$, referred to as a weighting sequence. A weighted norm associated to a weighting sequence is defined as

$$\begin{aligned} \|\cdot\|_w\colon (\mathbb{R}^n)^{\mathbb{Z}_-} &\longrightarrow \overline{\mathbb{R}^+} = \mathbb{R}^+ \cup \{\infty\} \\ \mathbf{z} &\longmapsto \|\mathbf{z}\|_w \coloneqq \sup_{t\in\mathbb{Z}_-} \|\mathbf{z}_t w_{-t}\|, \end{aligned}$$

where $\|\cdot\|$ denotes the Euclidean norm in $\mathbb{R}^n$.

Definition 2. ***Fading Memory Property for Causal Time-Invariant Filters***

Let $H_U\colon (D_n)^{\mathbb{Z}_-} \to \mathbb{R}$ be the functional associated to the causal and time-invariant filter $U\colon (D_n)^{\mathbb{Z}_-} \to \mathbb{R}^{\mathbb{Z}_-}$. Then, there exists a weighting sequence $w\colon \mathbb{N} \to (0,1]$ such that the map $H_U\colon ((D_n)^{\mathbb{Z}_-}, \|\cdot\|_w) \to \mathbb{R}$ is continuous. From this, it follows that U has the fading memory property, which means that for all $\mathbf{z} \in (D_n)^{\mathbb{Z}_-}$ and for all $\epsilon > 0$, there exists a $\delta(\epsilon) > 0$ such that for all $\mathbf{s} \in (D_n)^{\mathbb{Z}_-}$

$$\|\mathbf{z} - \mathbf{s}\|_w = \sup_{t \in \mathbb{Z}_-} \|(\mathbf{z}_t - \mathbf{s}_t) w_{-t}\| < \delta(\epsilon) \implies |H_U(\mathbf{z}) - H_U(\mathbf{s})| < \epsilon. \tag{2.1.6}$$

If the weighting sequence is given by $w_t = \lambda^t$ for a $\lambda \in (0,1)$ and all $t \in \mathbb{N}$, the filter U is said to have λ-exponential fading memory property.

Theorem 1. *Let $K_M := \overline{B_n(\mathbf{0}, M)}^{\mathbb{Z}_-} \subset (\mathbb{R}^n)^{\mathbb{Z}_-}$, a subset of left infinite real discrete time sequences, uniformly bounded by $M > 0$ wrt. Euclidean norm and*

$$\mathcal{R}_w := \{H_h^F\colon K_M \to \mathbb{R} \mid h \in C^\infty(D_N), F\colon D_N \times \overline{B_n(\mathbf{0}, M)} \to D_N, N \in \mathbb{N}\} \tag{2.1.7}$$

be the set of all fading memory RFs wrt. a weighted norm $\|\cdot\|_w$ with uniformly bounded inputs in K_M. $\mathbb{R}_w$ is then universal, i.e. it is dense in the set of real-valued functions $(C^0(K_M), \|\cdot\|_w)$ on $(K_M, \|\cdot\|_w)$.

2.1.2 State Affine Systems

The universality result in Theorem 1 concerns a large family of RFs. Therefore, the authors of [44] move on to much smaller subfamilies of reservoirs with the same universality property. These are linear reservoirs with polynomial readouts given by

$$\mathbf{x}_t = A\mathbf{x}_{t-1} + \mathbf{c}\mathbf{z}_t, \quad A \in \mathbb{M}_{N\times N}, \mathbf{c} \in \mathbb{M}_{N\times n} \tag{2.1.8}$$

$$y_t = h(\mathbf{x}_t), \quad h \in \mathbb{R}[\mathbf{x}], \tag{2.1.9}$$

where F in (2.1) is replaced by a linear map and h is a real-valued multivariate polynomial on $\mathbf{x}_t$ with real coefficients. After naming the associated reservoir functional in this case as the *linear reservoir functional* $H_h^{A,\mathbf{c}}\colon K_M \to \mathbb{R}$, we give the authors' universality result for this family of reservoirs as well as for other smaller subfamilies of it.

Theorem 2. *For $0 < \epsilon < 1$, consider the RF family $\mathcal{L}_\epsilon$ as in (2.1.8) & 2.1.9 with A having spectral norm $\sigma_{\max}(A) < 1 - \epsilon$. Then $\forall \rho \in (0,1)$, the elements in $\mathcal{L}_\epsilon$ generate λ_ρ-exponential fading memory reservoir functionals with $\lambda_\rho := (1-\epsilon)^\rho$. This is equivalent to $\mathcal{L}_\epsilon$ being a subset of $\mathcal{R}_{w^\rho} \subset \mathcal{R}_w$, $\mathcal{R}_w$ as in Theorem 1, with $w_t^\rho := \lambda_\rho^t$. These functionals are given by*

$$H_h^{A,\mathbf{c}}(\mathbf{z}) = h\left(\sum_{i=0}^{\infty} A^i \mathbf{c}\mathbf{z}_{-i}\right) \tag{2.1.10}$$

for all $\mathbf{z} \in K_M$ and the corresponding family is dense in $(C^0(K_M), \|\cdot\|_{w^\rho})$. This universality result is valid, when A is nilpotent or diagonal with $\sigma_{max}(A) < 1 - \epsilon$.

A consequence of Theorem 2 is that any fading memory I/O system can be approximated by a finite memory one, since an element in $\mathcal{L}_\epsilon$ with a linear reservoir map determined by a nilpotent A has a well-defined functional similar to the one given in (2.1.10) but with a finite sum [44]. However in RC, linear readouts are used almost exclusively, as opposed to the polynomial readouts in this family of systems. The reduction of the training to a linear regression is one of RC's features which makes it faster than its alternatives (see Section 2.3). To this point, the authors provide a universality theorem for a linear readout family called the *non-homogeneous state-affine system* (SAS) determined by the following transformations:

$$\mathbf{x}_t = p(z_t)\mathbf{x}_{t-1} + q(z_t), \quad p(z) \in \mathbb{M}_N[z], q(z) \in \mathbb{M}_{N,1}[z] \tag{2.1.11}$$

$$y_t = \mathbf{W}^\top \mathbf{x}_t, \quad \mathbf{W} \in \mathbb{R}^N, \tag{2.1.12}$$

where p and q are two polynomials on the variable z with matrix coefficients in $\mathbb{M}_{N\times N}$ and $\mathbb{M}_{N\times 1}$, respectively. The realization of these polynomials with a given degree $r \in \mathbb{N}$ is as the following:

$$\mathbb{M}_{a,b}[z] \coloneqq \{A_0 + zA_1 + z^2A_2 + \cdots + z^rA_r \mid a, b, r \in \mathbb{N}, A_0, A_1, A_2, \cdots, A_r \in \mathbb{M}_{a\times b}\}. \tag{2.1.13}$$

The authors' result (see Theorem 3.12 in [44]) tells that the family $\mathcal{S}_\epsilon$ of functionals induced by (2.1.11) & (2.1.12) on uniformly bounded semi-infinite inputs $H^{p,q}_{\mathbf{W}} \colon [-1,1]^{\mathbb{Z}_-} \to \mathbb{R}$ forms a polynomial subalgebra of $\mathbb{R}_{w^\rho}$ (from Theorem 2), under the condition that the maximum spectral norms of p and q from (2.1.11) over possible inputs, $\max_{z\in[-1,1]} \sigma_{\max}(p(z))$ and $\max_{z\in[-1,1]} \sigma_{\max}(q(z))$ are both smaller than $1-\epsilon$. The details on the operations (products and linear combinations) that form the polynamial algebra are given by (3.1) in [44]. Elements of $\mathcal{S}_\epsilon$ uniformly approximate any causal, TI fading memory filter $H \colon [-1,1]^{\mathbb{Z}_-} \to \mathbb{R}$, that is, there exist p, q and $\mathbf{W}$ as in (2.1.11) & (2.1.12) and $H^{p,q}_{\mathbf{W}} \in \mathcal{S}_\epsilon$ such that

$$\|H - H^{p,q}_{\mathbf{W}}\|_\infty \coloneqq \sup_{z\in[-1,1]^{\mathbb{Z}_-}} |H(z) - H^{p,q}_{\mathbf{W}}(z)| < \epsilon \tag{2.1.14}$$

for all $\epsilon < 0$. For the proofs of the presented universality statements, we refer the reader to [44]. We note that the authors extend their findings on universality of deterministic filters to the stochastic setup by utilizing the so-called *deterministic-stochastic transfer principle* (see Theorem 4.4 in [44]), where the input and output sequences are uniformly bounded discrete time stochastic processes. The universality property of the SAS family is the last result we share from the work of Grigoryeva and Ortega [44] and we move on to their universality result on ESNs in the next section.

2.1.3 Fading Memory Echo State Networks

ESNs are specific type of reservoir computers with input, reservoir and readout connections $C \in \mathbb{M}_{N\times n}, A \in \mathbb{M}_{N\times N}, W \in \mathbb{M}_{d\times N}$, respectively, as well as a bias vector $\boldsymbol{\zeta} \in \mathbb{R}^N$ and an *activation function* $\boldsymbol{\sigma}$ realizing the transformation for all $t \in \mathbb{Z}$ given by

$$\mathbf{x}_t = \boldsymbol{\sigma}(A\mathbf{x}_{t-1} + C\mathbf{z}_t + \boldsymbol{\zeta}), \tag{2.1.15}$$
$$\mathbf{y}_t = W\mathbf{x}_t. \tag{2.1.16}$$

Its uniform approximation capabilities are presented in this section as given in [43, 42]. The previous universality results have to do with an *external approximation* approach, where a given general filter is approximated by a reservoir one. Another approach is the *internal approximation*, which is approximating an RF with another one. This is the approach taken by the authors in the case of fading memory ESNs, where the proof of Theorem 3 involves approximating the universal SAS family by the ESN family. We note that we are still in the domain of causal, TI, discrete-time fading memory filters with uniformly bounded (left infinite) inputs. For a causal and TI filter it suffices to work with its restriction $U : (D_n)^{\mathbb{Z}_-} \to (D_n)^{\mathbb{Z}_-}$ instead of $U : (D_n)^{\mathbb{Z}} \to (D_n)^{\mathbb{Z}}$, since the former uniquely determines the latter:

$$U(\mathbf{z})_t = (T_{-t}(U(\mathbf{z})))_0 = (U(T_{-t}(\mathbf{z})))_0, \tag{2.1.17}$$

where $\mathbf{z} \in (D_n)^{\mathbb{Z}}$. ESNs get their name from the fact that there exists a function E such that at time step t the reservoir has the state $\mathbf{x}_t = E(\mathbf{z}_t, \mathbf{z}_{t-1}, ...)$, which can metaphorically be understood as $\mathbf{x}_t$ being an "echo" of the input history [61]. The property of having these unique *echo states* has the well-know name in the literature (see [43] and the references therein), which is the *echo state property* (ESP). Formally, an RF has the echo state or *uniqueness of solutions* property, if the system has the *existence of solutions property* and the solutions $\mathbf{x} \in (D_n)^{\mathbb{Z}}$ of (2.1.15) are unique. The existence of solutions property holds, when there exists $\mathbf{x} \in (D_n)^{\mathbb{Z}}$ that satisfies (2.1.15) for all $\mathbf{z} \in (D_n)^{\mathbb{Z}}$. A consequence of a reservoir map F without explicit time dependence having ESP is that the corresponding filter U^F is automatically causal and TI. Furthermore, if F is a contraction, that is, there exists $0 < r < 1$ such that $\|F(\mathbf{u}, \mathbf{z}) - F(\mathbf{v}, \mathbf{z})\| \leq r\|\mathbf{u} - \mathbf{v}\|$ for all $\mathbf{u}, \mathbf{v} \in \overline{B_N(\mathbf{0}, L)}$ with $\mathbf{z} \in \overline{B_n(\mathbf{0}, M)}$, then we have ESP and FMP. Unfortunately, it is difficult to check whether an ESN has the ESP in practice. Based on observations, a widely used routine is to set the spectral radius of the reservoir matrix smaller than unity, although it is not a sufficient criterion and does not always work [130]. The universality of ESNs is stated in Theorem 3. The proof is in [42].

Theorem 3. *Let $U\colon K_M \to K_L$ be a causal time-invariant fading memory filter for $M, L > 0$. Let the componentwise application of the activation in* (2.1.15)*, $\sigma\colon \mathbb{R} \to \mathbb{R}$ be Lipschitz-continuous, non-constant and bounded. Then, there exists an echo state network given by* (2.1.15) *&* (2.1.16) *with echo state and fading memory properties. Its associated filter*

$$U_{ESN}\colon K_M \to (\mathbb{R}^d)^{\mathbb{Z}_-}$$

satisfies

$$\left\|\left\|U - U_{ESN}\right\|\right\|_\infty < \epsilon \tag{2.1.18}$$

for all $\epsilon > 0$.

In practice, the connections of ESNs are chosen randomly (see Section 2.2.1). Although Theorem 3 proves the existence of universal ESNs, it does not state that these can be achieved with random matrices. Next section is particularly interesting as it sheds light on the randomness in RC architectures.

2.1.4 Signature State Affine Systems and Random Projections

Concluding the section on universality of RFs, we introduce the Signature State Affine System (SigSAS) from the work by Cuchiero et al. [25]. SigSAS shares fundamental properties with *Signatures*, which are objects from the rough path theory [75]. Signature of a path is a powerful feature map extracting its geometrical properties independent of its fine structure, allowing a simple linear map to approximate any function of it [23]. This is in alignment with the universality results given in this section. A definition of Signature is given in Definition 4.

The state-space transformation of SigSAS is a SAS with states defined in tensor spaces. With uniformly bounded scalar inputs $z \in [-M, M]^{\mathbb{Z}_-}$ for $M > 0$ and $l, p \in \mathbb{N}$, it is given by

$$\mathbf{x}_t = \lambda \pi_l(\mathbf{x}_{t-1}) \otimes \widetilde{\mathbf{z}}_t + \widehat{\mathbf{z}}_t^0, \quad 0 < \lambda < \min\left\{1, \frac{1-M}{1-M^{p+1}}\right\}, \tag{2.1.19}$$

where the states $\mathbf{x}_t \in T^{l+1}(\mathbb{R}^{p+1})$ are in the $\binom{l+1}{0}$-tensor space on $\mathbb{R}^{p+1}$

$$T^l(\mathbb{R}^d) \coloneqq \left\{ \sum_{i_1,\dots,i_l=1}^{d} a_{i_1,\dots,i_l} \mathbf{e}_{i_1} \otimes \cdots \otimes \mathbf{e}_{i_l} \mid a_{i_1,\dots,i_l} \in \mathbb{R} \right\} \tag{2.1.20}$$

with

$$\widetilde{\mathbf{z}}_t = \sum_{i=1}^{p+1} z_t^{i-1} \mathbf{e}_i \in \mathbb{R}^{p+1},$$
$$\widehat{\mathbf{z}}_t^0 = \sum_{i \in I_0} z_t^{i-1} \mathbf{e}_1 \otimes \cdots \otimes \mathbf{e}_1 \otimes \mathbf{e}_i \in T^{l+1}(\mathbb{R}^{p+1}),$$
$I_0 \subset \{1, ..., p+1\}$ subset of cardinality higher than 1 containing the element 1,
$\{\mathbf{e}_1, ..., \mathbf{e}_d\}$ the canonical basis of $\mathbb{R}^d$

and

$$\pi_l \colon T^{l+1}(\mathbb{R}^d) \to T^l(\mathbb{R}^d)$$

the *order lowering map* contracting the first index of any vector

$$\mathbf{v} \coloneqq \sum_{i_1,\dots,i_{l+1}=1}^{d} a_{i_1,\dots,i_{l+1}} \mathbf{e}_{i_1} \otimes \cdots \otimes \mathbf{e}_{i_{l+1}} \in T^{l+1}(\mathbb{R}^d)$$

as

$$\pi_l(\mathbf{v}) \coloneqq \sum_{i_2,\dots,i_{l+1}=1}^{d} a_{1,i_2,\dots,i_{l+1}} \mathbf{e}_{i_2} \otimes \cdots \otimes \mathbf{e}_{i_{l+1}} \in T^l(\mathbb{R}^d).$$

The authors note the contraction property of the state map $F^{\text{SigSAS}}_{\lambda,l,p}$ inducing (2.1.19) (see Proposition 2.2 in [25]), which implies that the system has ESP and FMP and the associated SigSAS filter $U^{\text{SigSAS}}_{\lambda,l,p}$ is causal and TI. This implication is clarified in Section 2.1.3. The explicit form of $U^{\text{SigSAS}}_{\lambda,l,p}$ is left out here and can be found in Proposition 2.2 in [25]. Universality of the SigSAS filter is stated in Theorem 4.

Theorem 4. *Let $M, L > 0, d = 1$ and B_M denote $B_{\|\cdot\|_\infty}(\mathbf{0}, M)$ the ball centered at $\mathbf{0} \in \mathbb{R}^d$ with radius M and supremum norm $\|\mathbf{z}\|_\infty := \sup_{t\in\mathbb{Z}_-} \|\mathbf{z}_t\|$. Define $\ell^\infty_-(\mathbb{R}^n) := \{\mathbf{z} \in (\mathbb{R}^n)^{\mathbb{Z}_-} \mid \|\mathbf{z}\|_\infty < \infty\}$ and $\widetilde{B}_M := B_M \cap \{\mathbf{z} \in (\mathbb{R}^d)^{\mathbb{Z}_-} \mid \sum_{t\in\mathbb{Z}_-} \|\mathbf{z}_t\| < \infty\}$. Let $U\colon K_M \subset \ell^\infty_-(\mathbb{R}^d) \to K_L \subset \ell^\infty_-(\mathbb{R}^m)$ be a causal and time-invariant fading memory filter whose restriction $U|_{B_M}$ is analytic as a map between open sets in the Banach spaces $\ell^\infty_-(\mathbb{R}^d)$ and $\ell^\infty_-(\mathbb{R}^m)$ and satisfies $U(\mathbf{0}) = \mathbf{0}$. Then, there exists a monotonically decreasing sequence w^U with zero limit such that for all $p, l \in \mathbb{N}$ and $0 < \lambda < \min\left\{1, \frac{1-M}{1-M^{p+1}}\right\}$, there exists a linear map $W \in \mathbf{L}(T^{l+1}(\mathbb{R}^{p+1}), \mathbb{R}^m)$ such that for all $\mathbf{z} \in \widetilde{B}_M$*

$$\left\| U(\mathbf{z})_t - WU^{\mathrm{SigSAS}}_{\lambda,l,p}(\mathbf{z})_t \right\| \leq w^U_l + L\left(1 - \frac{\|\mathbf{z}\|_\infty}{M}\right)^{-1}\left(\frac{\|\mathbf{z}\|_\infty}{M}\right)^{p+1}. \tag{2.1.21}$$

The type of universality in Theorem 4 is different than the previously stated universality results, since the approximation of the filter U is achieved by a linear readout on the higher order monomials in the tensor space $T^{l+1}(\mathbb{R}^{p+1})$ generated by the SigSAS filter $U^{\mathrm{SigSAS}}_{\lambda,l,p}$. Given a different causal TI fading memory filter U', the approximant $U^{\mathrm{SigSAS}}_{\lambda,l,p}$ remains unchanged and only the readout W needs to be learned. This type of universality is referred to as a *strong* universality by the authors. Note that the bound on the error of the approximation is filter dependent, since the decreasing sequence w^U_l depends on how fast the filter U "forgets" its past inputs. The result in Theorem 4 resembles very much the actual usage of RFs, where the linear readout is trained by means of linear regression to do a fit on a given output sequence, yet it does not allude to the randomly generated connections in RC. In order to achieve such randomnesses as well as a dimensionality reduction in SigSAS, the authors utilize the *Johnson-Lindenstrauss Lemma* [25, 28] stated in Theorem 5.

Theorem 5. *Given a Hilbert space $(V, \langle\cdot,\cdot\rangle)$ of dimension n and a subset Q of V with m elements, there exists a linear map $f\colon V \to \mathbb{R}^k$ such that for all $\mathbf{u}, \mathbf{v} \in Q$ and $0 < \epsilon < 1$*

$$(1-\epsilon)\|\mathbf{u}-\mathbf{v}\|^2 \leq \|f(\mathbf{u}) - f(\mathbf{v})\|^2 \leq (1+\epsilon)\|\mathbf{u}-\mathbf{v}\|^2 \tag{2.1.22}$$

with $\|\cdot\|$ in $\mathbb{R}^k$ satisfying the parallelogram identity and $k \in \mathbb{N}$ satisfying

$$k \geq \frac{24\log m}{3\epsilon^2 - 2\epsilon^3}. \tag{2.1.23}$$

The map f can be found in randomized polynomial time.

According to Theorem 5, an almost isometric embedding of a higher dimensional space into a lower dimensional one is possible, respecting the distortion parameter ϵ and the number of elements in the space to be embedded. Note that the target dimension is independent of the dimension of the original vector space.

The statement in Theorem 4 relies on the fact that a causal TI fading memory filter can be approximated by a truncated Volterra series on uniformly bounded inputs (see Theorem 2.1 in [25]) and the SigSAS filter $U^{\mathrm{SigSAS}}_{\lambda,l,p}$ is a representation of this high dimensional truncated Volterra series. The responsible parameters for the truncation are $l, p \in \mathbb{N}$. The parameter l

controls the extent by which one allows contributions from past inputs and p the degree of the polynomial approximation used in the Taylor expansion leading to the Volterra series.

In order to use the Johnson-Lindenstrauss (JL) Lemma, the authors identify a crossnorm on $T^{l+1}(\mathbb{R}^{p+1})$ and promote it to a Hilbert space $(T^{l+1}(\mathbb{R}^{p+1}), \langle\cdot,\cdot\rangle)$. Section C in [25] gives a brief discussion on the procedure for how this is done. Using a JL map as in (2.1.22), the truncated, yet high dimensional reservoir is shown to be reducible to a lower dimensional one without losing the ESP and FMP for sufficiently large $p, l \in \mathbb{N}$. Noting the dimension $N \coloneqq (p+1)^{l+1}$ of $\{\mathbf{e}_{i_1} \otimes \cdots \otimes \mathbf{e}_{i_{l+1}}\}_{i_1,\ldots,i_{l+1}\in\{1,\ldots,p+1\}}$, the canonical basis of $T^{l+1}(\mathbb{R}^{p+1})$, the new SAS using a JL projection $f\colon \mathbb{R}^N \to \mathbb{R}^k$ is given by

$$\begin{aligned} F^{\mathrm{SigSAS}}_{\lambda_0,l,p,f}\colon \mathbb{R}^k \times [-M, M] &\to \mathbb{R}^k \\ (\mathbf{x}, z) &\mapsto \sum_{i=1}^{p+1} z^{i-1} A_i \mathbf{x} + B\,(1, z, \cdots, z^p)^\top, \end{aligned}$$

where $A_i \in \mathbb{M}_{k\times k}$ and $B \in \mathbb{M}_{k\times p+1}$. The JL projection is realized by the matrices A_i, whose entries are randomly sampled from $\mathcal{N}(0, \frac{\delta^2}{4k\widetilde{M}^2})$, where δ is small enough such that

$$\begin{aligned} \lambda_0 &\coloneqq \frac{\delta}{2\widetilde{M}}\sqrt{\frac{k}{N_0}} < \min\left\{\frac{1}{\widetilde{M}}, \frac{1}{\widetilde{M}|||f|||^2}, 1\right\}, \\ \widetilde{M} &\coloneqq \frac{1-M^{p+1}}{1-M}, \\ N_0 &\coloneqq (p+1)^l \end{aligned}$$

and the probability of achieving ESP and FMP is at least $1-\delta$. The operator norm of f is wrt. the Euclidean norms in $\mathbb{R}^N$ and $\mathbb{R}^k$, whose expectation can be bounded by *Gordon's Theorem* [122, 25]. B accounts for the bias term $\widehat{\mathbf{z}}^0$ in (2.1.19) with entries

$$B_{a,b} \sim \begin{cases} \mathcal{N}(0, \frac{1}{k}) & \text{if } b \in I_0, \\ 0 & \text{otherwise.} \end{cases}$$

The authors achieve the same approximation result in (2.1.21) for the associated filter $U^{\mathrm{SigSAS}}_{\lambda_0,l,p,f}$ of $F^{\mathrm{SigSAS}}_{\lambda_0,l,p,f}$, this time with an additive term $I_{l,p}$ on the error bound and the linear readout map $\overline{W} = W \circ f^* \in \mathbf{L}(\mathbb{R}^k, \mathbb{R}^m)$:

$$\left\|U(\mathbf{z})_t - \overline{W}U^{\mathrm{SigSAS}}_{\lambda_0,l,p,f}(\mathbf{z})_t\right\| \le w_l^U + L\left(1 - \frac{\|\mathbf{z}\|_\infty}{M}\right)^{-1}\left(\frac{\|\mathbf{z}\|_\infty}{M}\right)^{p+1} + I_{l,p}, \tag{2.1.24}$$

where $f^*\colon \mathbb{R}^k \to T^{l+1}(\mathbb{R}^{p+1})$ is the adjoint of f and $I_{l,p}$ is either

$$I_{l,p} \coloneqq |||W|||\epsilon^{\frac{1}{2}} N^{\frac{3}{4}} \widetilde{M}\left(1 - \frac{\delta}{2}\sqrt{\frac{k}{N_0}}\right)^{-2} (1 + |||f|||^2)^{\frac{1}{2}}$$

or

$$I_{l,p} \coloneqq |||W|||\epsilon N \widetilde{M}\left(1 - \frac{\delta}{2}\sqrt{\frac{k}{N_0}}\right)^{-2},$$

where the JL distortion $0 < \epsilon < 1$ satisfies (2.1.23) with m replaced by N.

Many successes have been achieved by RC with random reservoir connections and trained linear readouts [40]. Just like the way RFs are used in the literature, the remarkable universality result given in (2.1.24) consists of a randomly generated reservoir map and a linear readout. In summary, the authors show that the truncated Volterra series representation of causal and TI fading memory filters admits a SAS with linear readouts in a high dimensional tensor space, which is then reduced in dimension via JL projections up to an ϵ-error. From this, we conclude that with high probability, the successful architectures reported in the literature actually correspond to exponentially larger dimensional precise approximants.

2.2 Architectures for Reservoir Computing

An overview on the specifications of the RC architectures commonly found in the literature is given in this section. Quantum RC, a new paradigm in RC, is also touched on at the end. The emphasis is put on the RC architecture that is used in this study, which is the ESN architecture. ESN architectures which are based on the original one, such as self-normalizing [124], hierarchical [57] or deep reservoirs [39] are not included.

2.2.1 Echo State Networks

ESNs are discrete-time recurrent neural networks. Different to the discussion in Section 2.1.3, here we take on a practical perspective and present it the way it finds usage in the literature. Specifically, the differences lie in the randomness and the dimensions of the weight matrices. We follow the notation in [61].

An ESN consists of K input units $\mathbf{u}(n) = (u_1(n), ..., u_K(n))^\top \in \mathbb{R}^K$, N internal units $\mathbf{x}(n) = (x_1(n), ..., x_N(n))^\top \in \mathbb{R}^N$ and L output units $\mathbf{y}(n) = (y_1(n), ..., y_L(n))^\top \in \mathbb{R}^L$. The internal units are recurrently connected by the update

$$\mathbf{x}(n+1) = (1-\alpha)\mathbf{x}(n) + \alpha\, \mathbf{f}(\mathbf{W}\mathbf{x}(n) + \mathbf{W}^{\text{in}}[1; \mathbf{u}(n+1)^\top]^\top + \mathbf{W}^{\text{back}}\mathbf{y}(n)) \tag{2.2.25}$$

and the forecast is obtained with

$$\mathbf{y}(n+1) = \mathbf{f}^{\text{out}}(\mathbf{W}^{\text{out}}[1; \mathbf{u}(n+1)^\top; \mathbf{x}(n+1)^\top; \mathbf{y}(n)^\top]^\top), \tag{2.2.26}$$

where the connections of an ESN are:

- Input to Reservoir with $\mathbf{W}^{\text{in}}_{N\times(K+1)}$
- Reservoir to Reservoir with $\mathbf{W}_{N\times N}$
- (Input + Reservoir + Output) to Output with $\mathbf{W}^{\text{out}}_{L\times(K+N+L+1)}$
- Output to Input with $\mathbf{W}^{\text{back}}_{N\times L}$

The notation $[;]$ describes concatenation (horizontal stacking) of vectors and matrices. The concatenations with 1 account for the bias. The activation functions of an ESN are the

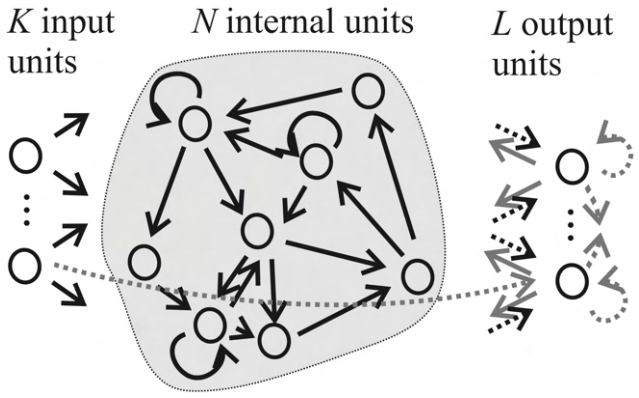


Fig. 2: Illustration of ESN [61].

reservoir activation $\mathbf{f}$ and the *output activation* $\mathbf{f}^{\text{out}}$, which are usually chosen as the hyperbolic tangent, sigmoid function or Rectified Linear Unit (ReLU). The reservoir activation creates the nonlinear dynamics in the reservoir and the output activation is often realized by applying the inverse of $\mathbf{f}^{\text{out}}$ on the output sequence. The *leaking rate* $\alpha \in (0, 1]$ takes the place of the sampling interval Δt [72, 123], when the reservoir update in (2.2.25) is seen as the Euler discretization

$$\frac{\Delta \mathbf{x}}{\Delta t} = \frac{\mathbf{x}(n+1) - \mathbf{x}(n)}{\Delta t} \approx \dot{\mathbf{x}}$$

of the Ordinary Differential Equation (ODE)

$$\dot{\mathbf{x}} = -\mathbf{x} + \mathbf{f}(\mathbf{W}\mathbf{x} + \mathbf{W}^{\text{in}}[1; \mathbf{u}^\top]^\top + \mathbf{W}^{\text{back}}\mathbf{y}).$$

Thus, a large leaking rate causes large updates to the internal state and fast dynamics, whereas a small leaking rate creates slow dynamics and increases the duration of the short-term memory [59] but decreases its ability to recall the most recent inputs [17]. ESNs with $\alpha \neq 1$ are called Leaky Integrator Echo State Networks (LiESNs).

The equation (2.2.25) is the generalized update equation, where different terms in it drop depending on the usage of ESN. The first term drops for ESNs, which are not LiESNs. The $\mathbf{W}^{\text{back}}\mathbf{y}(n)$ term inside the reservoir activation is called output feedback, which is left out in non-generative and purely input-driven problems, due to stability issues [62]. If the output $\mathbf{y}(n)$ depends on the previous outputs, then the output feedback might be necessary [61]. In such tasks, the outputs might be used as inputs and the term $\mathbf{W}^{\text{in}}[1; \mathbf{u}(n+1)^\top]^\top$ does not appear. For the case without output feedback, the outputs in the readout equation (2.2.26) disappear and for the case of no inputs but output feedback, the input term is not present. Lastly, the biases are also optional. In all these different cases, the dimensions of the connection matrices are adjusted accordingly. The settings with or without inputs or outputs create different training and validation schemes, which are summarized in Table 1. If we use both inputs and output feedback during training there are two options for the validation type. One could continue feeding the outputs next to the inputs, which is the *predictive* approach, or feed the ESN's own predictions back to the network, which is called the *generative* approach. Similarly, if inputs are not present during both training and validation, there is the option to take a generative approach. Since in that case no data is fed to the reservoir externally, it acts as an autonomous system. An important characteristic

of an ESN is that it requires a "warm-up" phase at the beginning of a training, where some of the first iterations of (2.2.25) are discarded. These are referred to as the *initial transient*, since echo states in the reservoir are obtained only after a transient effect at the beginning has passed. We denote the time steps belonging to this phase as $I_{warm} \coloneqq \{1, \dots, n_{warm}\}$ for a transient length of $n_{warm} \geq 1$. Similarly, the training and validation time steps are denoted as $I_{train} \coloneqq \{n_{warm}+1, \dots, n_{train}\}$ and $I_{val} \coloneqq \{n_{train}+1, \dots, n_{val}\}$, respectively, where $n_{train} > n_{warm}$ and $n_{val} > n_{train}$ and $n_{warm}, n_{train}, n_{val} \in \mathbb{N}$.

As mentioned in Section 2.1, the connections of an ESN are generated randomly, except for the readout $\mathbf{W}^{\text{out}}$, which is trained via linear or ridge regression

$$\mathbf{W}^{\text{out}} = \mathbf{Y}^{\text{target}}\mathbf{X}^{\top}\left(\mathbf{X}\mathbf{X}^{\top} + \beta\mathbf{I}\right)^{-1}, \tag{2.2.27}$$

where

$$\mathbf{Y}^{\text{target}} = \mathbf{W}^{\text{out}}\mathbf{X}, \tag{2.2.28}$$

$\mathbf{I}$ is the identity matrix and β is the regularization parameter. Closed form solution for linear regression is obtained by setting β to zero. The equation (2.2.28) is (2.2.26) rewritten in matrix notation, where the matrices $\mathbf{X} \in \mathbb{M}_{(K+N+L+1)\times T}$ and $\mathbf{Y}^{\text{target}} \in \mathbb{M}_{L\times T}$ are the collection of the input and target sequence histories, respectively, over the training period $n \in I_{train}$ with training length $\#I_{train} = T$. The output activation is left out, since it can be realized by applying its inverse on the target sequence. In case $\left(\mathbf{X}\mathbf{X}^{\top}\right)$ is not invertible, the Moore-Penrose pseudoinverse of $\mathbf{X}$ is used. Another widely practised routine is to select the random reservoir matrix $\mathbf{W}$ with sparse entries. The sparsity/connectivity of the reservoirs are typically between 80%/20% and 95%/5% [15]. Similar results can be obtained with random matrices, which are not sparse [56, 72, 16] (see Figure 3).

TRAINING		VALIDATION	
Data	Setting	Data	Setting
$\mathbf{u}, \mathbf{y}$	Input Driven/Teacher Forced	$\mathbf{u}, \mathbf{y}$ $\mathbf{u}$	Predictive Generative
$\mathbf{u}$	Input Driven	$\mathbf{u}$	Input Driven
$\mathbf{y}$	Teacher Forced	$\mathbf{y}$ None	Teacher Forced Autonomous

Table 1: Compatible training and validation settings of ESN. $\mathbf{u}$ stands for inputs and $\mathbf{y}$ stands for outputs. A training setting is followed by a validation setting specified in the same row.

The size of the reservoir N determines the *model capacity*, where the bias-variance trade-off of the model can be controlled by this parameter [62] as well as its *memory capacity* (MC) [60]. As N increases, it becomes easier to find a linear combination of the signals to match the target [72] but the increase in the number of learnable parameters could cause overfitting. The latter can be remedied by ridge regression or addition of noise to the system. The notion of the (short-term) MC of the reservoir comes from the question of how well it can approximate past targets given its current state. MC is derived and given in [60], by

$$\sum_{k=1}^{\infty} \text{MC}_k = \sum_{k=1}^{\infty} \frac{cov^2(\mathbf{u}(n-k), \mathbf{y}_k(n))}{\sigma^2(\mathbf{u}(n-k))\sigma^2(\mathbf{y}_k(n))} \tag{2.2.29}$$

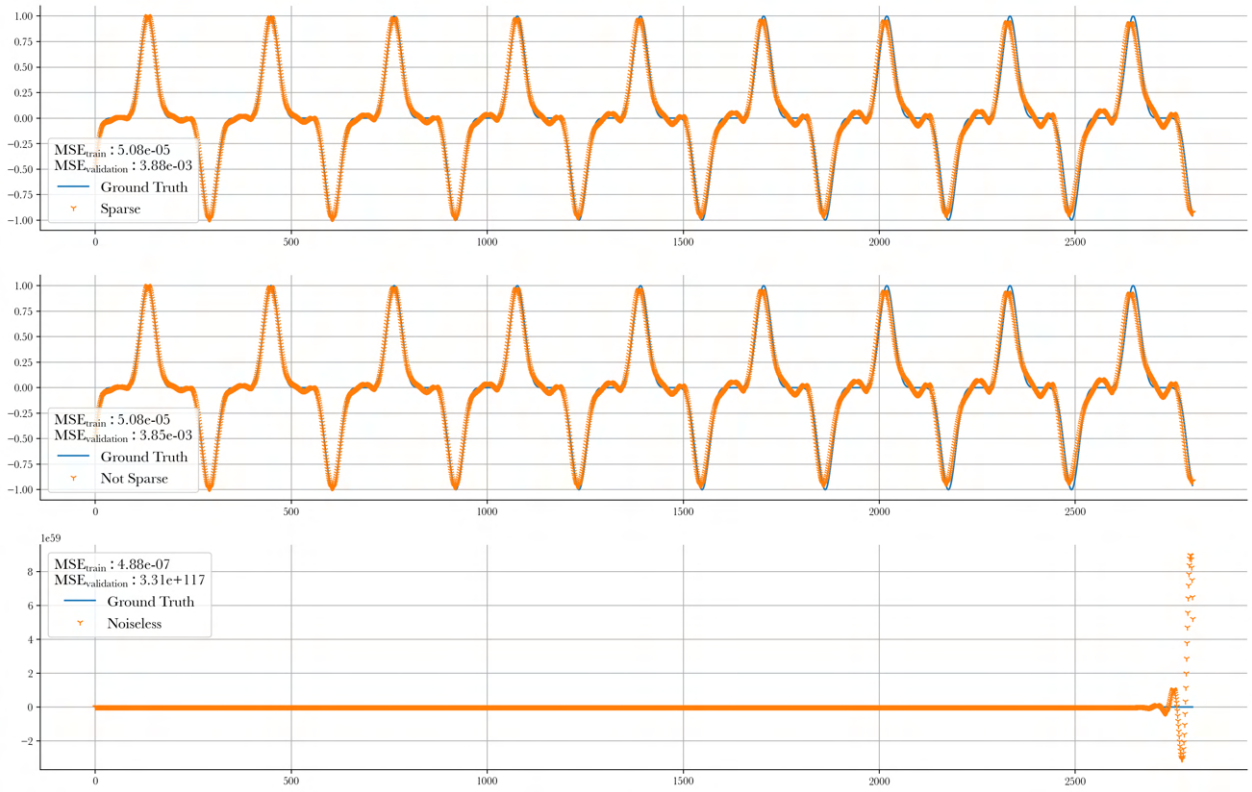


Fig. 3: Synchronization capabilities of sparse and not sparse ESNs with $f(t) = \sin^7(t/5)$. The plots show their validation performances in the autonomous setting after teacher forced trainings. The errors are shown in terms of Mean Squared Error (MSE). The sparse ESN has 20% connectivity and the not sparse ESN has reservoir matrix with entries drawn from a uniform distribution over $[-0.047, 0.047]$. They are all 1000-unit networks with spectral radius $\lambda_{max} = 0.88$ and leak rate $\alpha = 0.1$. The initialization/training/validation split is 800:2400:2800. A uniform noise over $[-0.1, 0.1]$ has been added to the teacher signal of ESNs except for the one in the last plot.

the sum of squared correlations between its estimates $\mathbf{y}_k(n)$ and the delayed targets $\mathbf{u}(n-k)$ over delays $k \in \mathbb{N}$, $n \in I_{val}$. Since it is a measure of correlation, it is possible to have high MC in case the model estimates have bias but preserve the "shape" of the targets. MC can be calculated by executing an input driven training, where the data to be fit to are the k-delayed inputs $\mathbf{u}(n-k)$, $n \in I_{train}$. After obtaining $\mathbf{W}^{\text{out}}$ through regression on the delayed inputs, the model is validated by (2.2.28), where $\mathbf{X}$ consists of inputs $\mathbf{u}(n)$ and reservoir states $\mathbf{x}(n)$ over the validation time steps $n \in I_{val}$ and $\mathbf{Y}^{\text{target}}$ contains the model estimates of the past inputs. For i.i.d. inputs, the MC of a reservoir without output activation ($\mathbf{f}^{\text{out}} = \mathbf{id}$) is bounded by the reservoir size N [60]. An example of this is shown in Figure 4. Our experiments with not i.i.d. inputs show that the leaking rate also affects the MC, which is illustrated in Figure 5 & 6. A higher leaking rate results in larger updates to the reservoir states at each time step; thus, the emphasis in terms of memory is put on the more recent inputs. As a result, better estimates are obtained for shorter delays compared to the networks with lower leak rates, whereas better estimates are obtained for longer delays with lower leak rates.

The stability of an ESN is ensured by the ESP. An important necessary condition to achieve it, is to have $\lambda_{max}(\mathbf{W}) < 1$, where $\lambda_{max}(\mathbf{W})$ is the spectral radius of $\mathbf{W}$. This necessary condition is valid for tasks which involve the zero input $\mathbf{u}(n) = 0$ and has been reported to be enough to achieve echo states in most situations [72]. The sufficient but restrictive condition for the ESP is to have $\sigma_{max}(\mathbf{W}) < 1$ [61]. Stability can also be achieved by addition of noise $\nu(n)$ into the system, especially in the presence of output feedback, where it acts as a regularizer and prevents ill-conditioning of the readout [58]. An example of this is included in Section 2.3. Figure 3 shows that adding noise can enhance ESN's generalized synchronization [100] capabilities, where the reservoir is trained with output feedback and validated by letting it run freely in its autonomous phase. A prominent phenomenon in ESNs is the so-called *edge of criticality* or the *edge of chaos*, where the computational capabilities of an ESN are maximized [11]. It is the border between the stable and unstable dynamics of the reservoir (see Figure 7). In an ESN with ESP, the effects of the initial conditions die out and the reservoir displays ordered dynamics. Hence, a common practice is to study the sensitivity of the reservoir to perturbations of its initial conditions and see whether it creates chaotic dynamics [16, 9]. Considering one perturbed and one unperturbed paths, the rate of their divergence is given by the Lyapunov exponent

$$\lambda = \lim_{k \to \infty} \frac{1}{k} \ln \left(\frac{\gamma_k}{\gamma_0} \right),$$

where γ_0 is the initial distance between the paths and γ_k is the distance at time step k. The edge of criticality lies at $\lambda \approx 0$ and ESN has ordered states for $\lambda < 0$ or unstable states for $\lambda > 0$ [36]. The procedure for estimating this quantity for ESNs [11] consists of (after an initial transient) perturbing one of the nodes x_l^2 of the perturbed state $\mathbf{x}^2$, $1 \leq l \leq N$, which separates the perturbed state from the unperturbed $\mathbf{x}^1$ by an amount γ_0 and calculating the distances $\gamma_k = \|\mathbf{x}^1(k) - \mathbf{x}^2(k)\|_2$ for time steps $k \in \{1, \ldots, T\}$, T denoting the length of the test. The introduced perturbation is of order 10^{-12} in [11]. To avoid numerical overflows, the perturbed state is renormalized by setting it to $\mathbf{x}^1(k) + (\gamma_0/\gamma_k)(\mathbf{x}^2(k) - \mathbf{x}^1(k))$ at each step. After determining this quantity, the network can be benchmarked by looking at its MC or choosing a regression task.

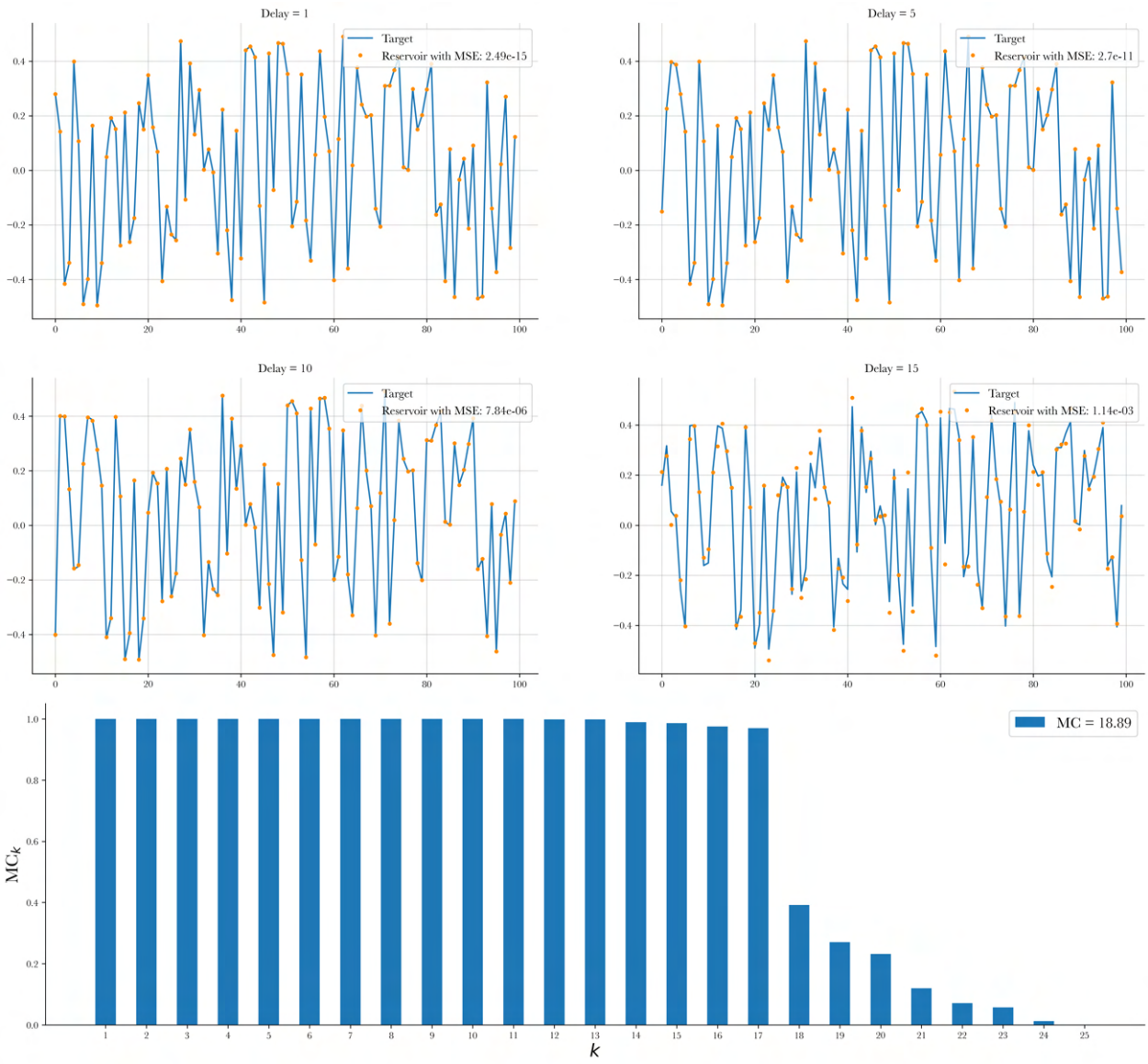


Fig. 4: MC of an ESN of size 20 with $\lambda_{max} = 0.8$ and $\alpha = 1$. Initialization/training/validation split is 100:100:100. The inputs are sampled from a uniform distribution over $[-0.5, 0.5]$. The connectivity of the reservoir is 20%. Reservoir and output activations are set to **id**, hence the MC is bounded by the size of the reservoir. The first four plots show the fitting accuracy of the network over delays 1, 5, 10 and 15. Lastly, MC_k are shown for delays between 1 and 25. A maximum delay of 100 has been used for the calculation of MC, i.e. $\text{MC} = \sum_{k=1}^{100} \text{MC}_k \approx 18.89$. Example taken from [60].

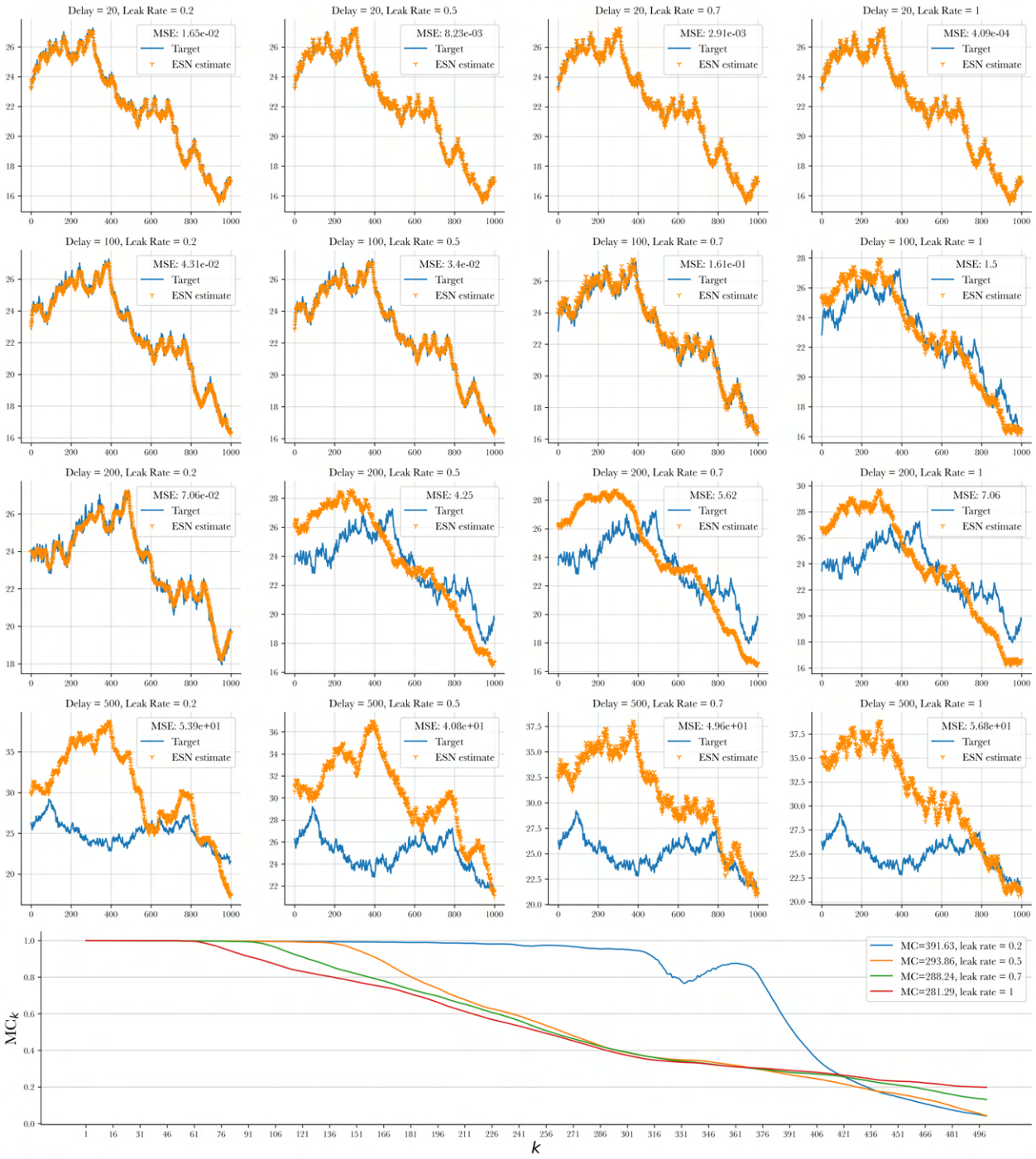


Fig. 5: MC of an ESN of size 100 with $\lambda_{max} = 0.8$ and leak rates 0.2, 0.5, 0.7 and 1. Initialization/training/validation split is 1000:1000:1000. The inputs are sampled [31] from a rough Heston model with parameters $\kappa \approx 5.49, \theta \approx 0.28, \sigma \approx 0.19, \rho = -0.62, \alpha = -0.25$. The connectivity of the reservoir is 20%. Reservoir and output activations are set to **id**. The plots at the first four rows show the fitting accuracy of the networks over delays 20, 100, 200 and 500. Lastly, MC_k are shown for delays between 1 and 500. A maximum delay of 500 has been used for the calculation of MC, i.e. $\mathrm{MC} = \sum_{k=1}^{500} \mathrm{MC}_k$.

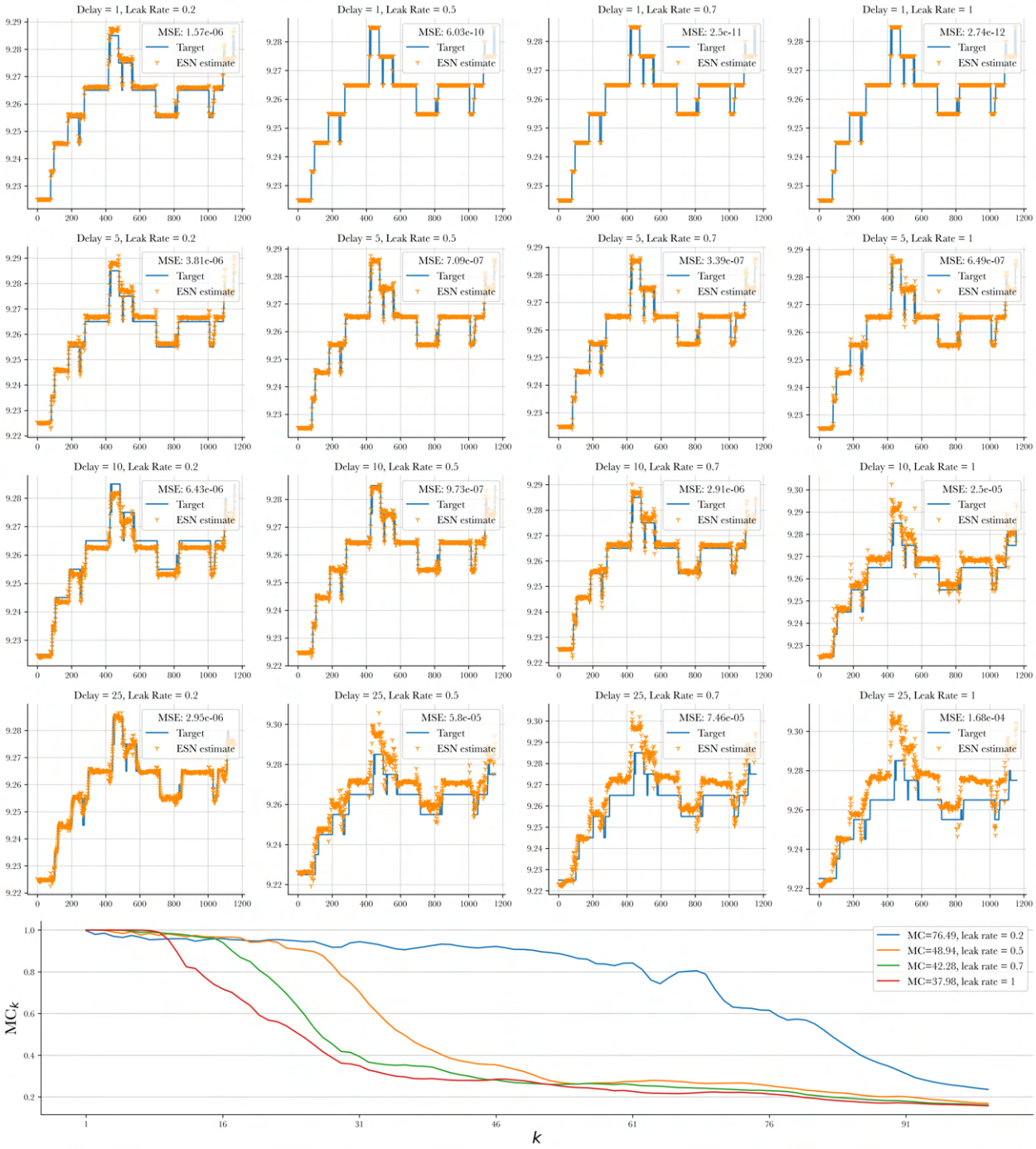


Fig. 6: MC of an ESN of size 100 with $\lambda_{max} = 0.4$ and leak rates 0.2, 0.5, 0.7 and 1. Initialization/training/validation split is 1000:2500:1158. The inputs are limit order prices and volumes from a real-world financial data, whose details are given in Section 4.1. The targets are mid-prices derived from it. Here, the usage of mid-prices as inputs alone yields poor performance (see Appendix B). The reservoir matrix has entries sampled from a uniform distribution over $[-0.65, 0.65]$. Reservoir and output activations are set to **id**. The plots at the first four rows show the fitting accuracy of the networks over delays 1, 5, 10 and 25. Lastly, MC_k are shown for delays between 1 and 100. A maximum delay of 100 has been used for the calculation of MC, i.e. $\mathrm{MC} = \sum_{k=1}^{100} \mathrm{MC}_k$.

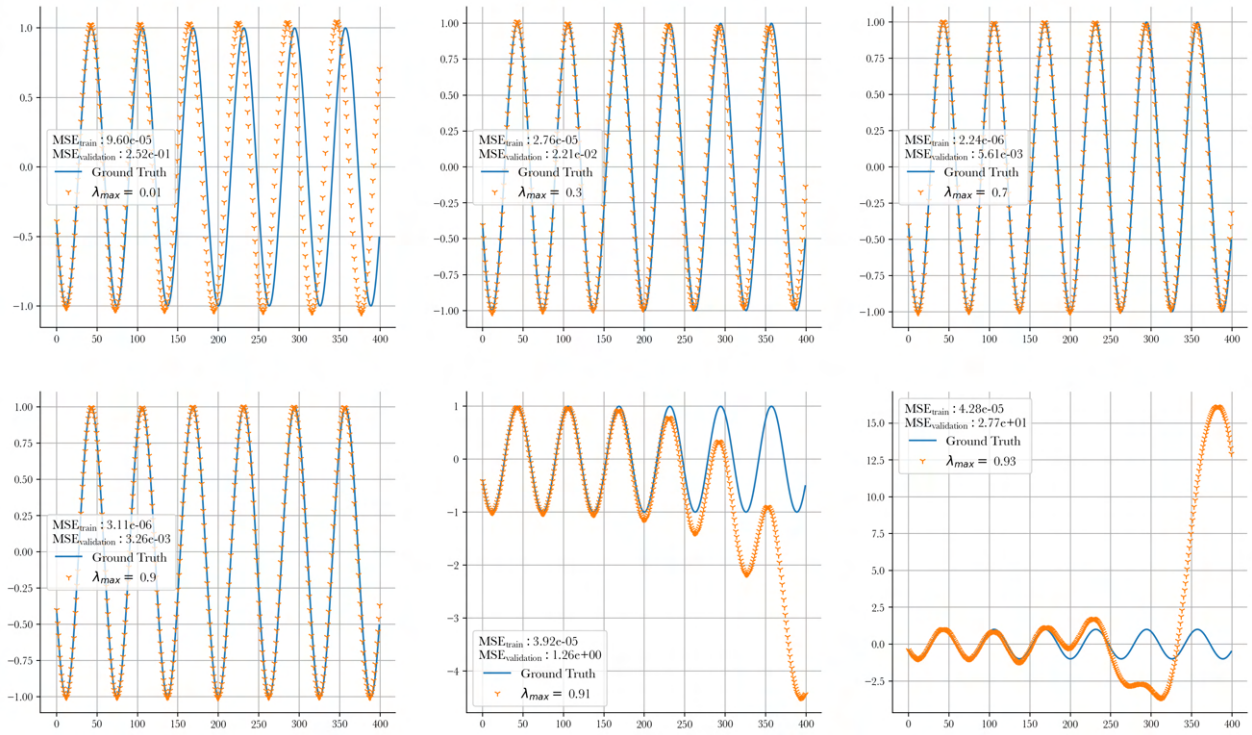


Fig. 7: Synchronization of ESNs with $f(t) = \sin(t)$. The plots show validation performances in the autonomous setting after teacher forced trainings. The ESNs have 250 units with varying spectral radii and leak rate $\alpha = 0.1$. The initialization/training/validation split is 800:800:400. A uniform noise over $[-0.1, 0.1]$ has been added to the teacher signal. The edge of chaos is located around $\lambda_{max} = 0.9$. As expected, performance increases up to this point. It is one of the cases where the necessary condition $\lambda_{max} < 1$ for ESP is not sufficient.

2.2.2 Extreme Learning Machines

ELMs are single-hidden layer FNNs (SLFNs) with randomly chosen input weights $\mathbf{W}^{\text{in}}$. Adopting the same notation from Section 2.2.1, their output function for a hidden layer with size N is given by (2.2.28), where this time the n^{th} column of $\mathbf{X} \in \mathbb{M}_{(N+1)\times T}$ is given by

$$\mathbf{f}(\mathbf{W}^{\text{in}}[1; \mathbf{u}(n)^\top]^\top), \quad n \in \{1, \dots, T\}$$

and the output $\mathbf{W}^{\text{out}} \in \mathbb{M}_{L\times(N+1)}$ is learned by means of gradient-based or more popularly, least squares methods, as in the case of ESNs. The universal approximation property of ELMs is proven in [51] (Theorem 2.2). The main difference between ESNs and ELMs is that the ELMs have static reservoirs or hidden layers, i.e. the update (2.2.25) is not applied to the hidden states in $\mathbf{X}$. Hence, they are not RNNs. As mentioned at the introduction of Section 2, we include them in the context of RC despite this fact, because they also randomly project the inputs and their hidden layer can be interpreted as a static reservoir. If the reservoir states $\mathbf{x}(n)$ in (2.2.25) are never stored in memory, i.e. $\mathbf{x}(n)$ is set to zero *after* the output $\mathbf{y}(n)$ is computed at each iteration $n \in \{1, \dots, T\}$, the resulting network is an ELM. This is what is meant by the term "static reservoir".

The randomized SLFN architecture presented as ELMs in [50] dates back to the work by Schmidt et al. [107]. The authors draw attention to the contradiction between the success of networks with high number of parameters and the Cramér-Rao error bound, which states the inverse proportionality of the number of parameters in an estimator and their variance. Recognizing the existence of the vast possibilities for the choice of parameters of a classifier with statistically acceptable performance, the authors investigate the classification performance of SLFNs with randomized hidden layers, which are ELMs with output biases.

Due to the work in [51] and others [116] a controversy on the origins of ELMs is present. For the sake of unity with the majority of the literature, we refer to the presented architecture as ELM.

2.2.3 Liquid State Machines

Liquid State Machines (LSMs) are reservoir computers proposed by Maass et al. [78] for modeling computations in cortical microcircuits, better than traditional models such as Turing machines or attractor-based models. They are developed independently of and simultaneously with ESNs [56], based on the same idea of having a recurrent reservoir layer with randomized connections and a trained readout tapping into the memory of the past inputs provided by the reservoir states. In the literature of LSMs, the reservoir layer, connections and states are referred to as the *circuit*, *synapses* and *liquid states*, respectively [78, 76]. The term "liquid" appears due to the analogy of having a liquid medium perturbed by external disturbances (inputs) and thereby containing information on present and past inputs at any given time. The output of the *liquid filter* L^M of an LSM M is the vector of outputs

$$(L^M u)(t) = \langle (B_1 u)(t), \dots, (B_m u)(t) \rangle$$

given by finitely many basis filters $B_1, \dots, B_m$ for a B-uniformly bounded and B'-Lipschitz-continuous function of time u for some $B, B' > 0$, with the *pointwise separation property*

$$\exists s \leq 0 \text{ s.t. } u(s) \neq v(s) \Rightarrow (B_i u)(0) \neq (B_i v)(0),\ 1 \leq i \leq m,$$

where v is a function with the same properties as u. The filters here are to be understood as the continuous time versions of the filters introduced in Section 2.1.1, where $(B_i u)(t)$ denotes the application of the filter B_i on the input function u at time $t \in \mathbb{R}$. The readout f^M of the LSM M has the *approximation property*

$$|h(x) - f(x)| \leq \rho \ \ \forall \rho > 0, h\colon X \to \mathbb{R}, x \in X \text{ with compact X } \subseteq \mathbb{R}^m.$$

The output of the LSM M is given by $f^M(x^M(t))$ and any causal TI fading memory filter on infinite continuous time compact inputs can be approximated arbitrarily close by it (Appendix A, Theorem 1 in [78]), where $x^M(t)$ is the liquid state $(L^M u)(t)$ at time $t \in \mathbb{R}$.

LSMs execute real-time computations on, mostly, streams of spike trains [78, 76, 52, 133]. Spike trains represent neural activity, which are obtained in neurophysiological studies by detecting action potentials, but preserving only the time instant at which they occur [96]. They can be considered as a list of the times $(t_i \in \mathbb{R})_{i \in \mathbb{Z}}$ where spikes occur [119]. The neurons in the circuit and the readout are *spiking neurons* (or *integrate-and-fire neurons*) with membrane potentials, firing upon their membrane potential reaching a certain threshold [78]. The membrane potential is the sum of so-called *excitatory postsynaptic potentials* and *inhibitory postsynaptic potentials*, which result from the firing of other neurons connected through synapses. The strengths of the synapses or the weights control the amount of contributions to each neuron from the others [77]. For the readout, these weights are learned, whereas in the circuit these are randomly generated and kept fixed [52]. In addition, the weights have positive values due to the inhibitory neurons with negative outputs [77, 52]. The readout neurons are modeled with *membrane time constants* [78], which are relaxation mechanisms for the membrane potentials specifying the time required for potentials to fall from the resting to 63% of their final value in the charging curve [55]. The main implementation area of LSMs is computer modeling for neural microcircuits [78], which carry out the specific functions of different brain regions [1]. The models for generic cortical microcircuits are beyond the scope of this study and the reader is referred to ref. [68] for the detailed descriptions of these.

2.2.4 Quantum Reservoir Computers

Quantum computing is an emerging paradigm based on manipulating properties of quantum states such as superposition and entanglement. The information of the state is stored in quantum bits, also known as qubits. Quantum computers (QCs) can perform computations beyond the limits of classical systems and their power is expected to grow at a double-exponential rate [7]. They offer the possibility for algorithms in certain areas with lower time complexities compared to classical ones [84].

A quantum reservoir computer (QRC) is a quantum machine learning (QML) [10] tool, where the classical reservoir is replaced by a quantum substrate, i.e. qubits are used instead of classical reservoir nodes. As expected, the exponentially large number of degree of freedom in Hilbert spaces has been reported to provide rich dynamics in QRCs comparable to classical systems with substantially larger reservoirs [37, 20, 118]. Fine tuning or precise engineering of the dynamics in the reservoir is not needed. Therefore, QRC is an especially convenient choice for QML during the noisy intermediate-scale quantum (NISQ) era [102], circumventing the requirement on robust control and resilience to noise and error [38, 63, 93, 21]. On the

contrary, the phenomenon of regularization and memory capacity increase through noise addition to the classical reservoir has been confirmed in its quantum counterpart [27, 115], which makes NISQ devices even more suitable for RC. Moreover, quantum substrates are naturally compatible with quantum inputs, opening up the possibility for edge computing in quantum systems with increasing complexity [91].

Since QRCs are fairly new in the ML scene, we provide examples from the literature showing its promise. The recent work by Fujii and Nakajima [37] using a nuclear magnetic resonance spin ensemble system, has pioneered the research on QRCs with spin-based architecture. Their 5-7 qubit systems match the performance of reservoir computers with 100-500 nodes, which is demonstrated through numerical examples typically used in benchmarking of classical reservoir computers. In a follow-up paper, they improve upon their previous results by introducing "spatial multiplexing", which combines multiple networks with the same input into a single output enabling multidimensional inputs and reducing the overall noise. Another work by Tran and Nakajima [118] combines this technique with classical connections between quantum reservoirs leading to increased scalability. Using dissipative quantum systems, Chen and Nurdin [20] provide the first proof of universality of a quantum reservoir filter subclass and investigate the capabilities of this model by means of numerical examples. Their comparisons to classical RCs align with the results obtained in ref. [37]. In their next work [21] they prove the universality of another QRC subclass, which is then implemented on IBM's cloud-based superconducting devices (IBMQ) [54]. In their experiments, the 10-qubit processor performs better than the 5-qubit ones in almost all cases. To increase the performance of the two 5-qubit systems, they are combined by spatial multiplexing and a twofold increase in performance is observed. QRCs' FTSF power is demonstrated in the work by Kutvonen et al. [66]. QRC, Autoregressive Integrated Moving Average (ARIMA) models and LSTMs are brought into comparison, where the QRC displays the best performance at predicting the future stock (closing) values 14 days ahead on the Standard and Poor's (S&P) 500 companies data set from February 8^{th} 2013 to December 29^{th} 2017. Another study [71] implements a quantum neuron model [64] on the ESN. Their quantum ESN is put to test in various tasks including the typical Nonlinear Autoregressive-Moving Average (NARMA) and Mackey-Glass system benchmarks as well as FTSF including S&P 500 index and foreign exchange data sets. Its performance in terms of all selected error measures is superior to ARIMA with Explanatory Variable model's and ESN's. Finally, using S&P 500 logarithmic returns as input, Dasgupta et al. [27] are able to do one step ahead prediction of the implied volatility index of S&P 500 options with their QRC on IBMQ.

Utilizing the rich dynamics and large state spaces offered by the exponentially scaling Hilbert spaces, QRCs already produce fruitful results. Having relatively light requirements on the reservoir's internal tunings and the ability to utilize noise as a regularizer and memory capacity enhancer makes them a unique and fitting choice for QML applications in the NISQ era.

2.3 Experimental Results

ESNs date back to the work by Michael B. Matthews under the name of "the recurrent network" [82] but their performance in ML has been made aware of by the work of Herbert Jaeger [61], recently. Here, results obtained by following the task descriptions in [61] are

reported on. An FTSF example is also included at the end.

2.3.1 House of the Rising Sun

We are interested in learning the vocals of the song "House of the Rising Sun" (see Figure 8). It is achieved by a teacher forced training followed up by an autonomous validation. The notes in the vocals range from G# to A at the higher octave, making up 14 notes. In the original paper the notes are encoded by integers starting from -1 to 14, thought as covering all notes starting from G# and incremented by semitones up to A, i.e. G#, A, A#, B, B#, C, C#, D, D#, E, E#, F, F#, G, G# and A, making up 16 notes. The reason for that is not accounting for the fact that B# is enharmonic to C and E# is enharmonic to F. Here, $\{-1, 0, 1, \ldots, 12\}$ is used for encoding, which is then squashed into the interval $[-1, 1]$ by dividing each integer by 24.

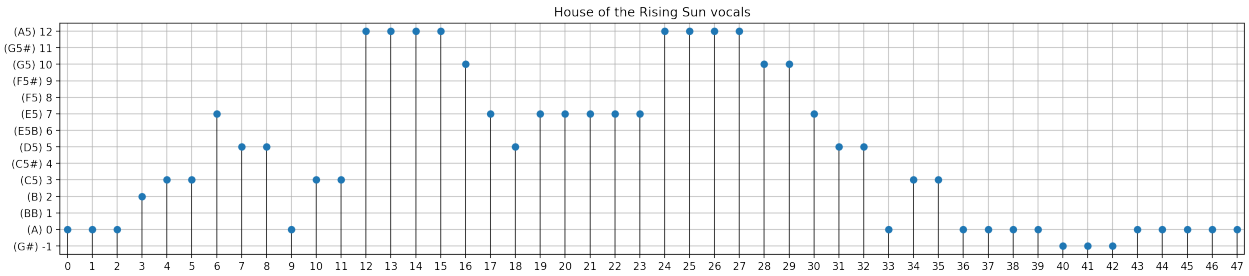


Fig. 8: Can be listened to at [33].

The chosen network has 400 reservoir nodes with 98.75% sparsity. The spectral radius is approximately 0.93 and the leak rate is set to 1. The reservoir and output activation functions are sigmoid $\mathbf{f} = \boldsymbol{\sigma}$ and hyperbolic tangent $\mathbf{f}_{\text{out}} = \mathbf{tanh}$, respectively. No bias is used and the concatenation of the teacher signal in (2.2.26) is omitted, i.e. the forecasts solely rely on the internal units. We note that this has been a necessity in our experiments. In addition, this task is one of the cases, where a not sparse network, in our experience, does not yield the desired results. The feedback matrix $\mathbf{W}_{\text{back}} \in \mathbb{M}_{400\times 1}$ has entries drawn from a uniform distribution over $[-2, 2]$. The teacher signal has eight consecutive zeros, connecting the end and the start of the melody. This means that an extensive history of the signal is required in the reservoir to achieve the desired synchronization, namely a memory span of at least nine iterations. This requirement of longer lasting input responses is accomplished by selecting a high value for the spectral radius λ_{max} [61, 72], in this case close to 1.

The ESN is trained with 1,500 training iterations, of which the first 500 iterations containing the initial transient are discarded. The goal is to do one step ahead forecasting, i.e. estimating the next musical note given the current one. One encounters an interesting problem here. The melody is periodic, consisting of 48 steps in each period. After the initialization of 500 steps, the reservoir states $\mathbf{x}(n)$ converge to a periodic sequence. Thus, the equation system that needs to be solved to obtain the readout matrix becomes underdetermined, since the effective number of equations is now 48. The fit by the regression algorithm represents basically one of the points on the hypersurface of perfect solutions. The MSE for the training is of order 10^{-19}, which is given in Figure 9. The regression results in a fit on the described subsystem leading to a weak validation performance. To overcome this problem,

one introduces uniform noise to the teacher signal over $[-0.001, 0.001]$, which eliminates the linear dependencies in the equation system to be solved. After the training, the network is put to test by feeding it with the teacher signal for 500 iterations as the "warm-up" phase. In our case, the "warm-up" works well enough for any number of iterations higher than 20. Then, the ESN is validated by letting it run freely in its autonomous phase for another 500 steps. We report that our network is able to stay synchronized for about 10,000 steps with a validation MSE of order 10^{-4}. We also note that selecting a $\lambda_{max} > 1$ does not lead to instability in our setup, where the chaotic regime starts around $\lambda_{max} = 11.1$. This is not contradictory to the previously stated necessary condition for the ESP, since the zero signals in the song are taken care of by the addition of the uniform noise. Possible overlaps of the zero values from the original signal and from the uniform distribution samples have been explicitly checked for.

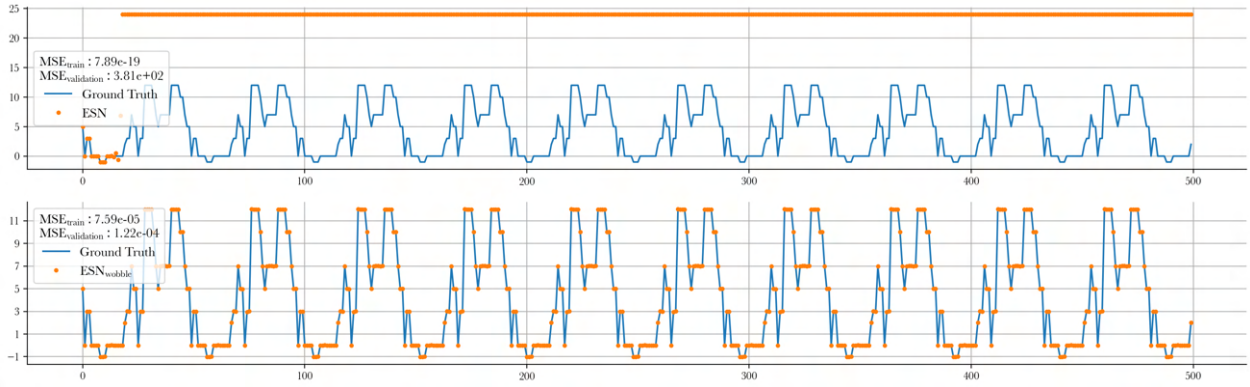


Fig. 9: Teaching ESNs the song "House of the Rising Sun". The ESN at the top represents the noiseless case, where the regression task involves a linearly dependent system. At the bottom we have the so-called "wobble" case, where a uniform noise is added to the teacher signal to remedy the problem.

2.3.2 Mackey-Glass Attractor

We are interested in learning the dynamics of the Mackey-Glass equations, which are delay differential equations

$$\dot{y} = \alpha \frac{y(t-\tau)}{1 + y^{\beta}(t-\tau)} - \gamma y(t),$$

displaying chaotic behavior for delay $\tau > 16.8$. The data with the delay parameter $\tau = 17$ has been acquired from [74]. As in the previous example, one step ahead forecasting is carried out. An ESN's performance in both teacher forced and autonomous validation settings is explored. The reservoir size is chosen to be 1000 and the spectral radius is set to 1.25 through scaling of the reservoir matrix, which is fully connected. The bias strength is set to 1 and the leaking rate is set to 0.3. The reservoir activation is the hyperbolic tangent. Ridge regression with regularization parameter set to 10^{-10} is used for the training of the readout. The results obtained in the teacher forced and autonomous cases are shown in Figure 10. The initialization/training/validation split is 1000:8000:2000. A long initialization period

benefits the network, since the spectral radius is high, resulting in a slow forgetting of the initial state of the reservoir [61].

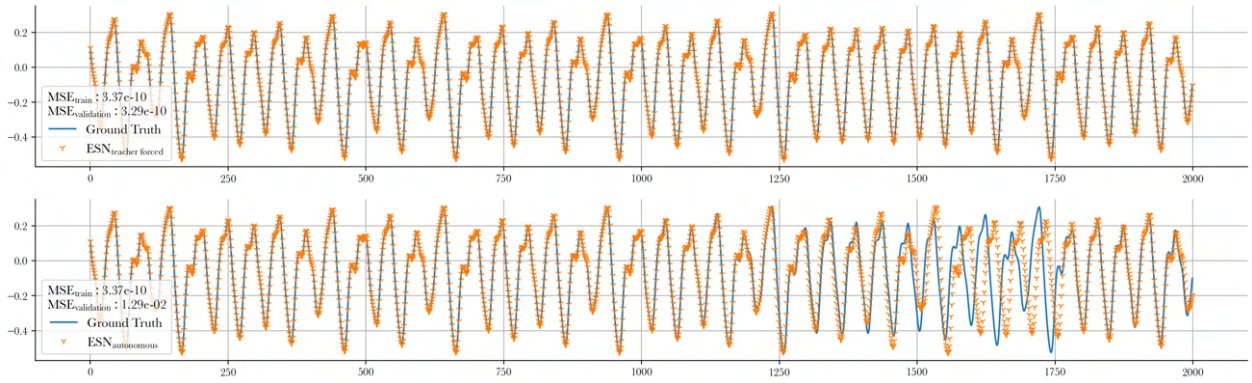


Fig. 10: Learning the Mackey-Glass chaotic dynamics with ESNs. The task is mastered by the teacher forced approach and the autonomous ESN displays similar performance approximately up to the 1250^{th} iteration. It resynchronizes around the 1800^{th} iteration.

As the next step, we compare ESNs to RNNs with LSTM units on the same task. The specifics of the chosen network architecture can be seen in Table 2. The network is constructed using the TensorFlow library [81]. Figure 11 shows the RNN's teacher forced and autonomous performance. The RNN is trained for 3000 epochs with batch size set to 1200. Adam optimizer with learning rate set to 0.001 and MSE as loss function is used. Note that the Mackey-Glass sequence is not distributed into subsequences to be fed to the RNN. In the autonomous case, it is evaluated the same way as in the case of the ESN, where data points are fed one by one. RNN has good teacher forced performance, yet the ESN is significantly better. It is the case despite the fact that the RNN has 5261 trainable parameters, whereas this number is 1002 for the ESN. Looking at the autonomous performances, it is not possible to follow this approach in the case of RNNs. Training and validation of the RNN takes about 3 minutes, whereas the same procedures take about 10 seconds for the ESN[1].

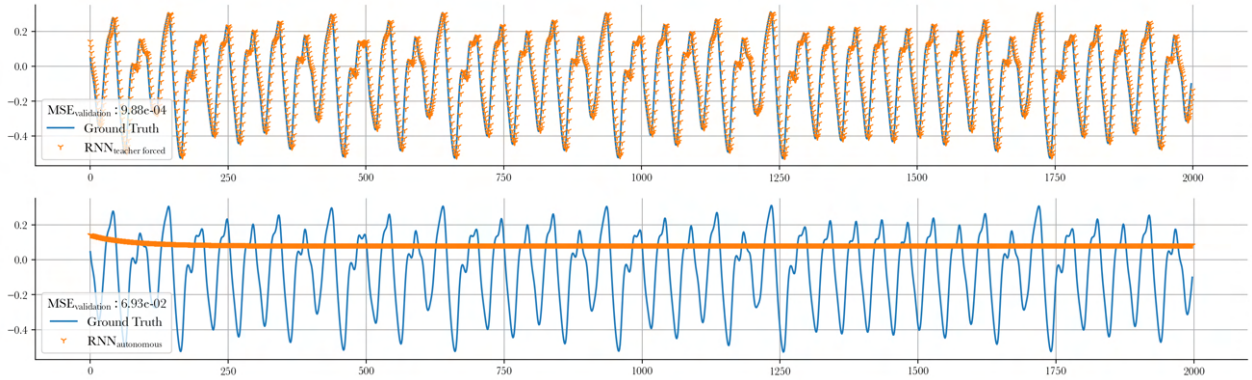


Fig. 11: Learning the Mackey-Glass chaotic dynamics with LSTMs. The teacher forced setting displays good performance. On the other hand, the autonomous approach does not work.

[1]A MacBook Pro (15-inch, 2018) has been used for the computations.

Network Structure		
Layer (type)	Output Shape	Number of Parameters
LSTM-1	[1, 20]	1760
LSTM-2	20	3280
Dense-3	10	210
ReLU-4	10	0
Dense-5	1	11

Table 2: RNN with LSTM layers. Number of parameters in the LSTM units is calculated by $4[(d_{in} + d_{out})d_{out} + d_{out}]$, where d_{in} and d_{out} are the input and output dimensions , respectively. In our case, $d_{in} = 1$ for LSTM-1 and $d_{in} = 20$ for LSTM-2. The output dimension d_{out} is set to 20 for both LSTMs. Biases are enabled for every layer.

2.3.3 Mid-price Forecasting

The last example is an adaptation of a previous work [32] to ESNs. There, FNNs are trained in a supervised learning setting to forecast mid-prices, price expectations and price variances for the financial services company Garanti BBVA's stocks (GARAN) in 2017 and are compared to linear models. The architecture of the FNNs together with the settings used in their training are given in Table 3. Limit order prices and volumes from a real-world financial data are used as inputs, whose details are given in Section 4.1. Memory is provided to the FNNs by means of rolling windows, which contain past information. The size of the windows determine how far into the past this information extends. The input size depends on the rolling window size. For a window size of 1, the input size is 21, which is the number of features in the data set, made up of the best five bid and ask prices and their corresponding volumes together with the time axis. Number of nodes in the output layer is set to 5 in accordance with the number of quantities to be forecast.

The training is carried out using data encompassing 202 trading days and validation is done on the remaining 51 trading days. The high-frequency data with milliseconds precision is reduced to minute scale by taking the last entries in each minute. The results obtained with ESN are shown in Figure 12. It is trained in the input driven/teacher forced setting. Rolling windows are not required, since ESN is able to store information from each iteration in its memory. Then, it is validated using the predictive setting and the generative setting (see Table 1). In the predictive setting the network is given the data with the previously discussed 21 features as inputs $\mathbf{u}$ and the mid-prices as teacher signal $\mathbf{y}$. During the generative validation, the teacher signal is not provided to the ESN. Instead, its own predictions are used as the teacher signal. It has 600 reservoir units with spectral radius 0.975 and 98.75% sparsity. The bias is enabled and the leak rate is set to 0.05. Hyperbolic tangent is chosen as the reservoir activation. No output activation is used. The fitting procedure is done by means of ridge regression with regularization parameter set to $3 \cdot 10^{-6}$. The best performing FNNs and linear models are shown in Table 4 for comparison. The predictive ESN outperforms them significantly in terms of both MSE and Mean Absolute Percentage Error (MAPE) and the generative ESN yields worse results than the ones obtained by FNNs. The best performing FNNs have 35,983 and 49,423 trained parameters utilizing windows of sizes 20 and 30, respectively. They are trained for over 48 hours using GPUs on the central clients of

Network Structure			HYPERPARAMETERS		CONFIGURATIONS
Input size: Window size × Number of variables in data set			DATA	Stock	GARAN
Layer (type)	Output Shape	Number of Parameters		Data Set	$\in \{\text{LOB}, \text{LIQ}, \text{LOB+LIQ}\}$
Linear-1	[202, 64]	Input size × 64 + 64		Window Size	$\in \{10, 20, 30, 40, 50, 60\}$
BatchNorm1d-2	[202, 64]	64 × 2 = 128	TRAINING	Split Ratio	0.8
LeakyReLU-3	[202, 64]	0		Batch Size	202
Linear-4	[202, 64]	64 × 64 + 64 = 4160		Validation Batch Size	1
BatchNorm1d-5	[202, 64]	64 × 2 = 128		Number of GPUs	1
LeakyReLU-6	[202, 64]	0	MODEL	Loss Function	MSE
Linear-7	[202, 64]	64 × 64 + 64 = 4160		LeakyReLU Slope	0.5
BatchNorm1d-8	[202, 64]	64 × 2 = 128	OPTIMIZER	LR	0.001
LeakyReLU-9	[202, 64]	0		Method	Adam
Linear-10	[202, 5]	64 × 5 + 5 = 325		LR Scheduler	ReduceLROnPlateau
BatchNorm1d-11	[202, 5]	5 × 2 = 10		Scheduler Patience	50 epochs
LeakyReLU-12	[202, 5]	0		Minimum LR	10^{-7}

Table 3: FNN architecture and hyperparameters used in the work [32]. **On the Left:** FNN with three hidden layers, each having 64 nodes. Biases are enabled for every layer. **On the Right:** Hyperparameter settings of the networks. LOB refers to limit order books and LIQ refers to liquidity measures [32], which are handcrafted features extracted from limit order books.

the department of mathematics of ETH Zurich. On the other hand, the ESN has 623 trained parameters and the training takes under 15 seconds[2].

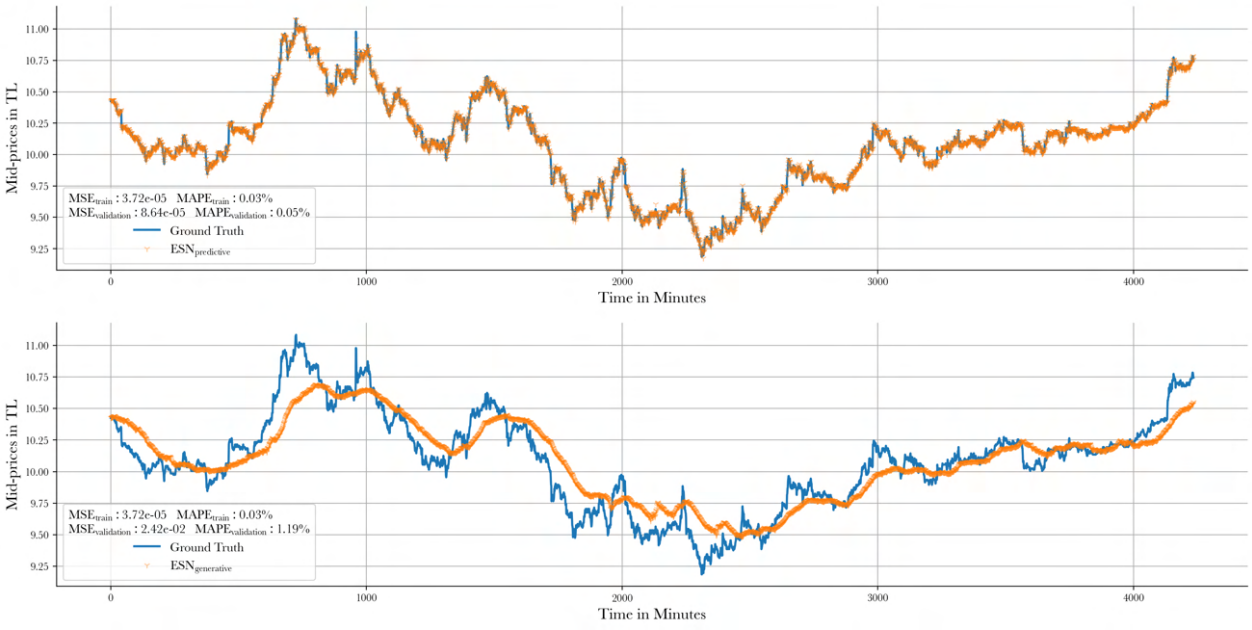


Fig. 12: Mid-price forecasting with ESNs. Errors are given in terms of MSE and MAPE for both training and validation. 51 trading days are merged together and each iteration represents a minute in the corresponding trading day. Mid-prices are given in Turkish Lira (TL).

[2]A MacBook Pro (15-inch, 2018) has been used for the computations.

	MSE			MAPE (%)		
Best Results	Model	Window Size	Validation Error	Model	Window Size	Validation Error
Mid Price	**FNN**	**30**	$\mathbf{0.37 \cdot 10^{-3}}$	**FNN**	**20**	**0.11**
Bid Price Expectation	FNN	30	$0.37 \cdot 10^{-3}$	FNN	20	0.11
Ask Price Expectation	FNN	30	$0.37 \cdot 10^{-3}$	FNN	20	0.11
Bid Price Variance	LinReg	60	$0.1 \cdot 10^{-8}$	LinReg	60	14.6
Ask Price Variance	LinReg	10	$0.13 \cdot 10^{-8}$	LinReg	20	18.3

Table 4: Results from the work done in [32]. Performance of each model is assessed in terms of MSE and MAPE. The best models in each forecast category is displayed. LinReg refers to a linear model.

2.4 Conclusion

RC is proven to be a powerful ML tool for learning nonlinear dynamical systems in the universality results provided Section 2.1. Different RC architectures are presented in Section 2.2, where ESNs as our choice of RC framework are studied extensively in Section 2.2.1. This choice is supported by numerical examples and comparisons in Section 2.3, where ESNs outperform FNNs and RNNs in terms of speed and validation scores in various tasks.

3 Reinforcement Learning

The chosen learning framework to obtain the main results of this study is RL. Therefore, a background on the theory and optimization tools used to derive these results is given. Markov decision processes are studied throughout the section. Discussions on experiments, which involve ESNs trained in an RL setup are also included. The decided optimization method presented in Section 4.2 is justified through these experiments. Section 3 is based on [114], unless specified otherwise.

3.1 Fundamentals of Reinforcement Learning

Consider the context, where a learner is expected to make decisions based on observations to solve a specific problem. Such learners in RL are called *agents*. As the decisions of the agent depend on observations or *states* coming from an *environment*, the observations depend on the decisions or *actions* of the agent. For the vast majority of the problems of practical interest, a priori knowledge of the complete specification of the environment is not available [114], which makes classical optimization methods and supervised learning techniques inapplicable. Instead, this type of information can be revealed by interacting with the environment and developing strategies or *policies* based on the responses from the environment. These responses include the state of the environment and a reward signal based on the actions taken by the agent. The process of taking an action based on an observation and receiving a response as a result is called a *transition* (see Figure 13). RL tasks can be episodic, where transitions lead to a terminal state at a certain point and the task is restarted. They can also be continuous, where this is not the case. These concepts are formalized in Section 3.1.1.

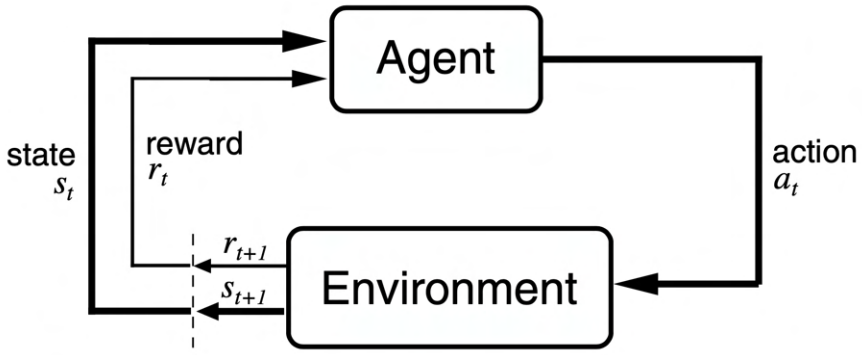


Fig. 13: The transition scheme in RL. An agent acts upon receiving a response from the environment. The response contains a state s_t and a reward r_t at time t. The agent receives a new response as a result of a new action a_t, which consists of a state s_{t+1} and a reward r_{t+1}. This is called a transition. Time steps could refer to arbitrary decision-making iterations, which do not necessarily have fixed intervals of real-time. Picture taken from [114].

3.1.1 Markov Decision Processes

In classical mechanics, a free particle's future trajectory is fully determined by its current location in the phase space, i.e. the future trajectory is independent of the path, which has led the particle to have its current position and velocity. Signals like position and velocity in the given context are said to be *Markov* or have the *Markov property*. They retain all information relevant to the upcoming event. Working in finite spaces with $t \in \{0, \dots, T\}$, we denote

- T as the finite number of transitions,
- $\mathcal{S}$ as the set of states,
- $\mathcal{A}$ as the set of actions and $\mathcal{A}(s) \subseteq \mathcal{A}$ as the set of possible actions in the state $s \in \mathcal{S}$,
- P_a with $P_a(s', s) \coloneqq Pr(s_{t+1} = s' \mid s_t = s, a_t = a, t < T)$ as the transition probability,
- R_a as the reward associated to action a, where $R_a(s', s) \coloneqq \mathbb{E}[r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s', t < T]$.

Then, a finite *Markov Decision Process* (MDP) is characterized by the 4-tuple $(\mathcal{S}, \mathcal{A}, P_a, R_a)$, where the dynamics of the process reduces from

$$Pr(s_{t+1} = s', r_{t+1} = r' \mid s_t, a_t, r_t, s_{t-1}, a_{t-1}, \dots, r_1, s_0, a_0) \tag{3.1.30}$$

to

$$Pr(s_{t+1} = s', r_{t+1} = r' \mid s_t, a_t). \tag{3.1.31}$$

An example for an MDP would be the *pole balancing* task [86, 8], also known as the *CartPole* system [65] (see Figure 14). A cart is moved horizontally by applying forces to it. A pole is hinged to the cart and positioned such that there is a 90° angle between the two objects. The goal is to keep the pole straight as long as possible by pushing the cart to right or left. The game is over, if the pole falls below a certain angle or the cart leaves its platform. The decision to push the cart right or left at a certain time is only dependent on the cart's current position and velocity and the pole's current angle and angular velocity. If the information provided by an environment comprehends all of them, the pole-balancing problem is an MDP.

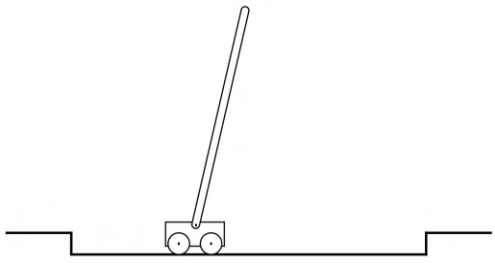


Fig. 14: Illustration of the pole-balancing problem. Taken from [114].

3.1.2 Value Functions

A crucial element in RL optimization algorithms is the *value function*. It provides an estimate of the *value* of a state s or a *state-action pair* (s, a) based on the discounted return

$$G_t = \sum_{k=0}^{T-t} \gamma^k r_{t+k+1},$$

where $\gamma \in [0, 1]$ is the *discount factor*. The discount factor has the role of adjusting the importance of the immediate or future rewards, where a smaller γ puts emphasis on shorter-term rewards. It is useful to define value functions on policies

$$\begin{aligned} \pi \colon \mathcal{S} \times \mathcal{A} &\to [0, 1] \\ (s, a) &\mapsto \pi(a \mid s). \end{aligned}$$

Policies are probability distributions on the space of state-action pairs. Agents are characterized by them, since policies yield probabilities for possible actions in a given state. Then, the *state-value function for policy* π can be written for MDPs as

$$V^\pi(s) = \mathbb{E}_\pi[G_t \mid s_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{T-t} \gamma^k r_{t+k+1} \mid s_t = s \right], \; s \in \mathcal{S}, \tag{3.1.32}$$

where $\mathbb{E}_\pi$ denotes the expected value, given that the agent follows the policy π. V^π quantifies the value of being in a state s for an agent with the policy π. Similarly, one can write the value for taking the action a under the policy π after observing the state s as

$$Q^\pi(s, a) = \mathbb{E}_\pi[G_t \mid s_t = s, a_t = a], \tag{3.1.33}$$

which is called the *action-value function for policy* π. The goal in RL tasks is to maximize the rewards over the long run and policies that achieve this goal are of interest. A policy π is said to be better than another policy π' if and only if $V^\pi(s) > V^{\pi'}(s)$, $\forall s \in \mathcal{S}$. There is always at least one policy, which is better than or equal to other policies. Such a policy is called an *optimal policy* π^*, which is expressed by

$$V^{\pi^*}(s) = \max_{a \in \mathcal{A}(s)} Q^{\pi^*}(s, a).$$

Denoting V^{π^*} and Q^{π^*} as V^* and Q^*, respectively, one can find a relationship between the value of a state and the values of its successor states to write the condition for an optimal policy $\forall s \in \mathcal{S}$ given by

$$\begin{aligned} V^*(s) &= \max_{a \in \mathcal{A}(s)} Q^*(s, a) \\ &= \max_a \mathbb{E}_{\pi^*}[G_t \mid s_t = s, a_t = a] \\ &= \max_a \mathbb{E}_{\pi^*}[r_{t+1} + \gamma G_{t+1} \mid s_t = s, a_t = a] \\ &= \max_a \sum_{s'} P_a(s', s)[R_a(s', s) + \gamma V^*(s')] \qquad (3.1.34) \\ &= \max_a \mathbb{E}[r_{t+1} + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a], \qquad (3.1.35) \end{aligned}$$

where in the second last equality we have used

$$\mathbb{E}_{\pi^*}[r_{t+1} \mid s_t = s] = \sum_a \pi^*(a \mid s) \sum_{s'} P_a(s', s) R_a(s', s)$$

and

$$\begin{aligned}
\mathbb{E}_{\pi^*}[\gamma G_{t+1} \mid s_t = s] &= \mathbb{E}_{\pi^*}\left[\gamma \sum_{k=0}^{T-t-1} \gamma^k r_{t+k+2} \mid s_t = s\right] \\
&= \sum_a \pi^*(a \mid s) \sum_{s'} P_a(s', s) \gamma \mathbb{E}_{\pi^*}\left[\sum_{k=0}^{T-t-1} \gamma^k r_{t+k+2} \mid s_{t+1} = s'\right] \\
&\overset{t \to t+1}{=} \sum_a \pi^*(a \mid s) \sum_{s'} P_a(s', s) \gamma V^*(s').
\end{aligned}$$

The equations (3.1.34) and (3.1.35) are two forms of the *Bellman optimality equation* (BOE) for V^*. Similarly, the BOE for Q^* $\forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$ is

$$Q^*(s, a) = \mathbb{E}[r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') \mid s_t = s, a_t = a] \tag{3.1.36}$$

$$= \sum_{s'} P_a(s', s)[R_a(s', s) + \gamma \max_{a'} Q^*(s', a')]. \tag{3.1.37}$$

In the BOE for the state-value function, we find ourselves in state s and look for the action which causes the transition from the state s to s' and maximizes the expectation of the sum of the return obtained through the transition and the (discounted) value of the new state s'. In the BOE for the action-value function, we consider new actions a' with the maximum expected (discounted) value, taken after reaching state s'. This is illustrated in Figure 15.

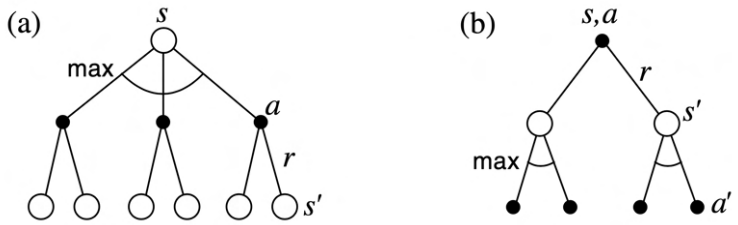


Fig. 15: Backup diagrams for (a) V^* and (b) Q^*. Taken from [114].

3.1.3 Model-free Optimization

In Section 3.2, model-free solutions to RL problems are considered, as opposed to model-based ones. A model provides information to the agents about the environment that they can use to produce predictions of the outcome of their actions. This can be done by using explicit probability distributions to take the place of P_a and R_a. A model-free approach would be to simulate such a model by an agent gathering samples through experience or trial-and-error, while interacting with the environment [127]. Specifically, the state-value and action-value functions can be learned for each sampled state-action pair by means of

iterative updates. Since the quality of the value estimates rely on the gathered samples, the *exploration-exploitation* trade-off needs to be balanced to obtain an optimal policy [127].

Methods, which use value functions to search for optimal policies can be categorized by Dynamic Programming (DP), Monte Carlo (MC) and Temporal Difference (TD) methods. They all share an underlying mechanism called *generalized policy iteration* (GPI) (see Figure 16), which consists of two interacting submechanisms called *policy evaluation*, given by

$$V_{k+1}(s) = \mathbb{E}_\pi[r_{t+1} + \gamma V_k(s_{t+1}) \mid s_t = s] \tag{3.1.38}$$

and *policy improvement*, given by

$$\pi'(s) = \arg\max_a \mathbb{E}_\pi[r_{t+1} + \gamma V^\pi(s_{t+1}) \mid s_t = s, a_t = a]. \tag{3.1.39}$$

Policy evaluation changes the state-value estimate to be more like the true state-value function for the policy π. Following similar calculations that lead to (3.1.34), it can be shown that

$$V^\pi(s) = \sum_a \pi(s,a) \sum_{s'} P_a(s',s)[R_a(s',s) + \gamma V^\pi(s')], \tag{3.1.40}$$

which is the *Bellman equation for* V^π. Comparing (3.1.38) with (3.1.40), it can be seen that $V_k = V^\pi$ is a fixed point for the update. The updated value function is then used to improve the existing policy by the policy improvement, where the action values are evaluated for every state to find a policy with better action suggestions. The policy π' in (3.1.39) is called the *greedy policy*. From (3.1.39) follows that $\forall s \in \mathcal{S}$

$$V^{\pi'} = \max_a \mathbb{E}[r_{t+1} + \gamma V^{\pi'}(s_{t+1}) \mid s_t = s, a_t = a],$$

which is the BOE (3.1.35). Hence, policy improvement yields strictly better policies π' except when π is already optimal. Note that deterministic policies are considered in the given relations. The results can be extended to stochastic policies by $Q^\pi(s,a) \to Q^\pi(s,\pi(s)) = \sum_a \pi(s,a) Q^\pi(s,a)$.

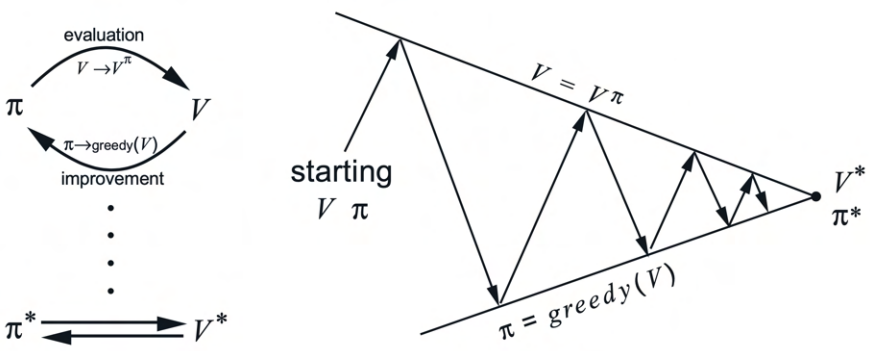


Fig. 16: Illustration of the GPI scheme [127]. Policy and value function converge hand in hand to optimal ones through policy evaluation and policy improvement.

3.2 Temporal Difference Methods

DP uses model-based algorithms, where they rely on accurate models of the environments. Both MC and TD methods rely on collected experience to update their value estimates. MC methods wait for the end of an episode to update their value estimates, whereas TD methods update their estimates every time step. For the simplest TD method TD(0), this is given by

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)], \tag{3.2.41}$$

where α is the learning rate. The term $r_{t+1} + \gamma V(s_{t+1})$ in update (3.2.41) is called the *target* and the *TD residual* δ_t is defined as

$$\delta_t \coloneqq r_{t+1} + \gamma V(s_{t+1}) - V(s_t).$$

The target in MC is G_t, which is not available at time $t+1$, if the visited state s_{t+1} is a nonterminal state. From (3.1.32), we know that

$$V^\pi(s) = \mathbb{E}_\pi[G_t \mid s_t = s] \tag{3.2.42}$$

$$= \mathbb{E}_\pi[r_{t+1} + \gamma V^\pi(s_{t+1}) \mid s_t = s]. \tag{3.2.43}$$

In MC methods the real expected return in (3.2.42) is not known and they average sample returns observed during interaction [127]. DP methods use models to compute (3.2.43), but they use the current estimate V_t instead of the true V^π. TD methods incorporate both ideas from MC and DP by sampling the expected value in (3.2.43) and using the value estimate V_t at time t instead of V^π. Since the target of TD(0) is an existing estimate, it is called a *bootstrapping method*. A pseudocode for TD(0) is shown in Algorithm 1.

Algorithm 1 TD(0)

```
Initialize V(s) arbitrarily, π to the policy to be evaluated
for each episode do
    Initialize s
    while s is not terminal do
        a ← action given by π for s
        Take action a; observe reward r and next state s′
        V(s) ← V(s) + α[r + γV(s′) − V(s)]
        s ← s′
    end while
end for
```

3.2.1 SARSA

State-action-reward-state-action (SARSA) is a TD algorithm. It is also an *on-policy* learning algorithm, i.e. it updates its value estimate based on the action selected by its current value estimate, given by

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]. \tag{3.2.44}$$

Note that the action-value function is learned instead of the state-value function. Theorems that guarantee the convergence of the state values in TD(0) apply here as well [114]. In (3.2.44), the state-action pair transition quintuple $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ is needed, hence the name SARSA. A pseudocode for SARSA is shown in Algorithm 2. An ϵ-greedy action as in Line 3 and 6 is given by

$$a = \begin{cases} \textit{randomly selected action with probability } \epsilon \\ \arg\max_{\tilde{a}} Q(s, \tilde{a}) \textit{ with probability } 1 - \epsilon \end{cases}, \tag{3.2.45}$$

where $s \in \mathcal{S}, a \in \mathcal{A}(s), 0 < \epsilon < 1$. A higher value for ϵ favors exploration over exploitation.

Algorithm 2 SARSA

1: Initialize $Q(s, a)$ arbitrarily
2: **for** each episode **do**
3: Initialize s, choose action a ϵ-greedily from s
4: **while** s is not terminal **do**
5: Take action a; observe reward r and next state s'
6: $a' \leftarrow \epsilon$-greedy action given s'
7: $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
8: $s \leftarrow s'$; $a \leftarrow a'$
9: **end while**
10: **end for**

3.2.2 Deep Q-Learning

Another example of TD methods is Q-learning [125]. It is an *off-policy* algorithm, i.e. it updates its current policy according to the action derived from another policy. From the update in Line 7 of Algorithm 3, we see that the other policy is a greedy policy, where the action with the maximum value is selected. In contrast to SARSA, a single transition is needed before a value update. SARSA and Q-learning require lookup tables, where the

Algorithm 3 Q-Learning

1: Initialize $Q(s, a)$ arbitrarily
2: **for** each episode **do**
3: Initialize s
4: **while** s is not terminal **do**
5: Choose action a ϵ-greedily from s
6: Take action a; observe reward r and next state s'
7: $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
8: $s \leftarrow s'$
9: **end while**
10: **end for**

value of each action $a \in \mathcal{A}(s)$ for all states $s \in \mathcal{S}$ in the environment's state space is stored.

These values are referred to as the *Q-values*. In the case of high dimensional or discretized continuous state spaces, storing a value for each state-action pair becomes infeasible. A scalable way of dealing with this problem is to approximate the action-value function $Q^*(s,a)$ by a neural network $Q^*(s,a:\theta)$, where θ are the weights of the so-called *Q-network*. Following this approach in the Q-learning setting means that bootstrapping, off-policy training and function approximation elements are used simultaneously. The combination of these three elements can lead to instabilities and divergences during the learning process, which is known as the *deadly triad* in the RL literature [113]. Successful learning of control policies in the deadly triad setting has been achieved by the *Deep Q-Learning* (DQL) algorithm [89], where mechanisms are implemented to tackle the challanges of utilizing nonlinear function approximators in Q-learning. These challanges include the sparse and noisy reward signals, which are temporally disjoint from the responsible actions and highly correlated observations, whose distributions change as the algorithm learns new behaviours. A mechanism called the *experience replay* [69] is implemented to tackle these issues, where previous transitions are randomly sampled in order to reduce the correlations between the states. This technique consists of collecting so-called *rollouts* or *trajectories* with a predefined size $n_{steps} \in \mathbb{N}^+$ and transitions

$$\{(s_0,a_0,r_1,s_1),\dots,(s_{T_1-1},a_{T_1-1},r_{T_1},s_{T_1}),(s_0,a_0,r_1,s_1),\dots,(s_{T_n-1},a_{T_n-1},r_{T_n},s_{T_n})\},$$

where $T_1,\dots,T_{n\in\mathbb{N}} \geq 1$ are the total time steps upon reaching a terminal state with $\sum_i T_{i=1}^n = n_{steps}$. Note that T_n might not represent the time step of a terminal state but the time step upon the buffer reaching its predefined size. In practice, if the replay buffer is full, the old transitions in it get overwritten by the new ones [4]. Then, the rollouts are stored to be randomly sampled over a discrete uniform distribution from a so-called *buffer* during the training of the neural network. The training of the Q-network is carried out by means of stochastic gradient descent on the sequence of objective functions [89]

$$L_i(\theta_i) = \mathbb{E}_{s\sim\rho_\pi, a\sim\pi}\left[(y_i - Q(s,a;\theta_i))^2\right] \tag{3.2.46}$$

at each iteration i, where the gradient wrt. the weights θ_i of the Q-network at iteration i is given by

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s\sim\rho_\pi, a\sim\pi; s'\sim\mathcal{E}}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right].$$

The target y_i in (3.2.46) is given by $\mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) \mid s,a\right]$ with the *target network* $Q(s,a;\theta_{i-1})$, where s' is sampled from the environment $\mathcal{E}$. The action a on observing the state s results in the reward r and state s' and ρ_π is the distribution of states under the policy π. The whole DQL procedure is summarized in Algorithm 4, where the action-value function in the ϵ-greedy action selection (3.2.45) is to be understood as its neural network counterpart $Q(s,a;\theta)$.

An important point about today's implementations of DQL is that the target network is often a lagged version of the Q-network. Following the proposed method in [88], the target network's weights are kept constant for a specified number of iterations and updated periodically instead of being assigned the Q-network's weights from the previous iteration in order to reduce the correlations between the values from the Q-network and the target

Algorithm 4 Deep Q-Learning

Initialize replay buffer $\mathcal{D}$ with capacity n_{steps}
Initialize Q-network with random weights
for each episode **do**
    Initialize s
    for each iteration **do**
        Choose action a ϵ-greedily from s
        Take action a; observe reward r and next state s'
        Store transition (s, a, r, s') in $\mathcal{D}$
        $s \leftarrow s'$
        Sample batch of transitions $(s_i, a_i, r_{i+1}, s_{i+1})$ randomly from $\mathcal{D}$
        if s_{i+1} is terminal **then** $y = r_{i+1}$ **else** $y = r_{i+1} + \gamma \max_{a'} Q(s_{i+1}, a'; \theta)$
        Perform gradient descent step on $(y - Q(s_i, a_i; \theta))^2$
    end for
end for

network. Such target networks are referred to as *frozen target networks* in the literature [95]. Another method that leads to a lagged target is *Polyak's update*, where the last instance of the target weights $\tilde{\theta}$ is updated by the last instance of the Q-network's weights θ according to $\tilde{\theta} \leftarrow (1 - \tau)\theta + \tau\tilde{\theta}$ with $\tau \in (0, 1)$, aiming to regularize the target weights [99].

3.2.3 Proximal Policy Optimization

The previously presented TD methods all have their policies expressed through value functions. Another option would be to learn a parametrized policy

$$\pi_\theta(a \mid s) \equiv \pi(a \mid s\,;\, \theta) = Pr(a_t = a \mid s_t = s, \theta_t = \theta),$$

independent from the value function by doing gradient ascent on a scalar performance measure $J(\theta)$ to maximize it. This allows smooth changes in the action probabilities as opposed to ϵ-greedy action probabilities, which can change dramatically for an arbitrarily small change in the value estimates. Other advantages of policy gradient methods over action-value methods are that they can learn appropriate levels of exploration and handle continuous action spaces. Moreover, considering episodic tasks with a non-random starting state s_0, the *policy gradient theorem* (PGT)

$$J(\theta) = V^{\pi_\theta}(s_0) \;\Rightarrow\; \nabla_\theta J(\theta) \propto \sum_s \rho_{\pi_\theta}(s) \sum_a \nabla_\theta \pi_\theta(a \mid s) Q^{\pi_\theta}(s, a) \tag{3.2.47}$$

gives an exact formula for how performance is affected by the policy parameter. The policy gradient in (3.2.47) can be further written as

$$\begin{aligned}\sum_s \rho_{\pi_\theta}(s) \sum_a \nabla_\theta \pi_\theta(a \mid s) Q^{\pi_\theta}(s,a) &= \mathbb{E}_{s\sim\rho_{\pi_\theta}}\left[\sum_a \nabla_\theta \pi_\theta(a \mid s) Q^{\pi_\theta}(s,a)\right] \\ &= \mathbb{E}_{s\sim\rho_{\pi_\theta}}\left[\sum_a \pi_\theta(a \mid s) Q^{\pi_\theta}(s,a) \frac{\nabla_\theta \pi_\theta(a \mid s)}{\pi_\theta(a \mid s)}\right] \\ &= \mathbb{E}_{s\sim\rho_{\pi_\theta}, a\sim\pi_\theta}\left[Q^{\pi_\theta}(s,a) \nabla_\theta \log \pi_\theta(a \mid s)\right] \\ &\equiv \mathbb{E}_{\pi_\theta}\left[Q^{\pi_\theta}(s,a) \nabla_\theta \log \pi_\theta(a \mid s)\right].\end{aligned}$$

Note that PGT does not involve any derivatives of the state distribution ρ_{π_θ}. This is crucial, since the effects of the changes in policy action distribution on the distribution of states ρ_{π_θ} under policy π_θ are typically unknown [113].

Different expressions for the vanilla policy gradient can be found in the literature, where it has the form [108, 109]

$$\hat{\mathbb{E}}_t\left[\psi_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\right].$$

$\hat{\mathbb{E}}_t[\dots]$ is the empirical average over a finite batch of samples and ψ_t could be, for instance, $Q^{\pi_\theta}(s_t, a_t)$ or the *advantage function* $A^{\pi_\theta} \coloneqq Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t)$. It is apparent that the policy gradient has value function and policy as main components. If the value function is also approximated by some parameters, in addition to the policy, the policy gradient method is called an *actor-critic method*. The approximated action-value $Q_w(s,a)$ or state-value $V_w(s)$ function with parameters w acts as a critic, where it values the states and/or actions of the agent. Then, the parameters θ of the policy π_θ are updated in the direction suggested by the critic. A state of the art example of on-policy actor-critic methods is the so-called *Proximal Policy Optimization* (PPO) [109]. The authors of PPO improve upon the ideas presented in the *Trust Region Policy Optimization* (TRPO) paper [110]. With the idea of importance sampling (see Definition 5), TRPO uses a surrogate objective given by

$$\underset{\theta}{\text{maximize}} \quad \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)} \hat{A}_t\right] \tag{3.2.48}$$

$$\text{subject to} \quad \hat{\mathbb{E}}_t\left[\mathrm{KL}[\pi_{\theta_{old}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t)]\right] \leq \delta, \tag{3.2.49}$$

where $\pi_{\theta_{old}}$ is the policy before the update, $\hat{A}_t$ the advantage function estimate and

$$\mathrm{KL}[\pi_{\theta_{old}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t)] = \sum_a \pi_{\theta_{old}}(a \mid s_t) \log\left(\frac{\pi_{\theta_{old}}(a \mid s_t)}{\pi_\theta(a \mid s_t)}\right)$$

the *Kullback-Leibler divergence* (see Definition 6) between the old and the current policy [79], which is kept within the parameter δ. The Kullback-Leibler (KL) divergence is used as a statistical measure to calculate how the current policy differs from the old one. The constraint (3.2.49) is implemented due to the fact that the objective (3.2.48) displays a behaviour similar to the vanilla objective

$$\hat{\mathbb{E}}_t\left[\log \pi_\theta(a_t \mid s_t) \hat{A}_t\right],$$

whose gradient leads to destructively large policy updates [109]. The authors of PPO identify that the theory justifying TRPO actually suggests implementing the constraint (3.2.49) as a penalty term instead of a hard constraint, which leads to

$$\underset{\theta}{\text{maximize}} \;\; \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)} \hat{A}_t - \beta \, \mathrm{KL}[\pi_{\theta_{old}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t)] \right]$$

for some coefficient β. A fixed penalty coefficient β does not perform well during the learning process with changing characteristics. The novel idea in PPO addressing this issue of controlling the update sizes is the clipped surrogate objective

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min(\tilde{r}_t(\theta)\hat{A}_t, \mathrm{clip}(\tilde{r}_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right], \tag{3.2.50}$$

where $\tilde{r}_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ and ϵ is a hyperparameter, usually set to 0.2. Note that with the first term inside the min in (3.2.50), the TRPO objective (3.2.48) is recovered. The advantage estimates $\hat{A}_t$ for a policy run for T time steps are given by the Generalized Advantage Estimator (GAE) [108]

$$\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} = \sum_{l=0}^{T-t-1} (\gamma\lambda)^l \delta_{t+l}, \tag{3.2.51}$$

which is an exponentially weighted average with the weight parameter $\lambda \in [0,1]$ and γ is the discount factor. The TD residuals in (3.2.51) are given in terms of $V^{\pi_{\theta_{old}}}$. Lastly, the complete objective function of PPO, which is used for gradient updates at each iteration, is given by

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \right], \tag{3.2.52}$$

where S denotes an entropy bonus [128, 87] to ensure sufficient exploration, $L_t^{VF}(\theta)$ is the value loss [103]

$$(V^{\pi_\theta}(s_t) - V_t^{target})^2 \;\; \text{with} \;\; V_t^{target} \coloneqq \hat{A}_t + V^{\pi_{\theta_{old}}}(s_t),$$

c_1 and c_2 are the value and entropy coefficients, respectively. It can be shown [112] that the defined value target V_t^{target} is the λ-*return* [113]

$$G_t^\lambda \coloneqq (1-\lambda) \sum_{l=0}^{T-t-1} \lambda^l G_t^{(l+1)}$$

with the n-*step return* [113]

$$G_t^{(n)} \coloneqq r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V^{\pi_{\theta_{old}}}(s_{t+n}).$$

The complete algorithm is shown in Algorithm 5.

The dynamics of the clipped objective L^{CLIP} is illustrated in Figure 17. If a sampled action of $\pi_{\theta_{old}}$ yields a better than expected return, the sample advantage estimate A is positive, and vice versa. If the advantage is positive $A > 0$ and due to a previous gradient step the sampled action is a lot more probable under the current policy, i.e. $\tilde{r}_t(\theta) > 1+\epsilon$, then the gradient of L^{CLIP} is zero and the outlier sample is ignored. Similarly, if the advantage estimate is negative and the action becomes a lot less probable, no further update occurs.

This mechanism prevents the policy to undergo large updates in a certain direction such that the actions do not become a lot more/less likely due to outlier samples. The non-clipped parts in Figure 17 depict another important feature of the objective L^{CLIP}. If the advantage is negative and due to a previous gradient step the sampled action is more probable, i.e. $\tilde{r}_t(\theta) > 1$, then the gradient of L^{CLIP} is negative and contributes to making the action less probable. Similarly, if the advantage estimate is positive and the action becomes less probable, then the gradient has a likelihood increasing effect on the action.

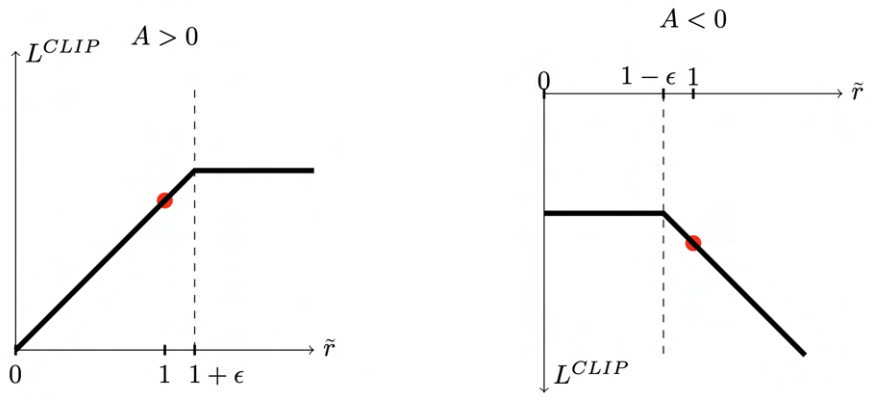


Fig. 17: Clipped surrogate objective's behaviour in relation to $\tilde{r}$ and sign change of the advantage. L^{CLIP}, A and $\tilde{r}$ in the plots are considered as items from a single time step. Picture taken from [109]

Algorithm 5 PPO

Initialize number of environments n_{envs}
Initialize number of PPO epochs n_{epochs}
Initialize maximum number of time steps n_{max}
$n \leftarrow 0$
while $n < n_{max}$ **do**
    Initialize buffer $\mathcal{D}$ with size $T \cdot n_{envs}$
    for $t = 1, 2, \ldots, T$ **do**
        Compute $\{(s_t^i, a_t^i, r_{t+1}^i, V^{\pi_{\theta_{old}}}(s_t^i), \pi_{\theta_{old}}(\cdot \mid s_t^i))\}_{i=1,\ldots,n_{envs}}$ and store in $\mathcal{D}$
    end for
    Compute advantage estimates $(\hat{A}_1^i, \ldots, \hat{A}_T^i)_{i=1,\ldots,n_{envs}}$ and store in $\mathcal{D}$
    $n \leftarrow n + T \cdot n_{envs}$
    for epoch = 1,2,$\ldots, n_{epochs}$ **do**
        $n_{index} \leftarrow 0$
        while $n_{index} \leq T \cdot n_{envs}$ **do**
            Randomly sample batches of size $M \leq T \cdot n_{envs}$ from $\mathcal{D}$
            Optimize surrogate $L^{CLIP+VF+S}$ on the batch wrt. θ
            $n_{index} \leftarrow n_{index} + M$
        end while
    end for
    $\theta_{old} \leftarrow \theta$
end while

3.3 Experimental Results

We present our results on using FNNs and ESNs in different RL settings. The optimization methods that we have used are based on DQL and PPO. Hyperparameters used in each task is shared and denoted as:

- γ: Discount factor
- c_1: Value coefficient in (3.2.52)
- c_2: Entropy coefficient in (3.2.52)
- clip range: ϵ in (3.2.50)
- λ_{GAE}: λ in (3.2.51)
- n_{epochs}: Number of PPO epochs in Algorithm 5
- batch size: Batch size M in Algorithm 5
- n_{envs}: Number of environments to train on
- n_{steps}: Number of samples to be collected for each environment, T in Algorithm 5

Adam optimizer is used in all PPO implementations presented in this study.

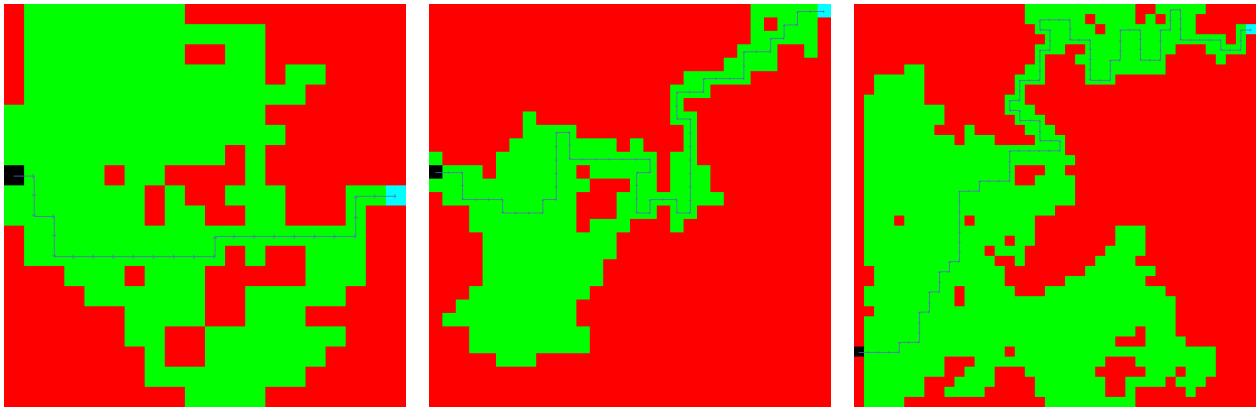
3.3.1 A Maze Game

We start by evaluating the PPO method described in Section 3.2.3 on a simple maze game, which we have written. The mechanics of the game are similar to the *Frozen Lake* game, which can be found on the website of OpenAI [14]. The agent is in a square grid world and can move one tile up, down, right or left. The tiles have colors red and green. The game is terminated if the agent visits a red tile or an already visited tile, leaves the map or reaches the exit, which is located at one of the rightmost tiles and is white. The agent's starting point is at one of the leftmost tiles, which is black. At each time point, the agent observes a 5×5 area, whose center tile is the agent's current location. If the agent is around the borders of the map, parts of this area, which are outside of the map, are imputed with red tiles. In addition, the observations include the agent's current coordinates and the distance between the current location and the exit. A scalar reward 1 is given if the agent gets closer to the exit in terms of x- or y-coordinate. The network architecture and hyperparameters of PPO are given in Table 5, which are chosen manually. Our results show that the agent is able to beat the game in up to 40×40 maps. We report that the training is susceptible to "wrong starts", where the agent adopts unfavorable behaviours at the beginning of the training. Results on some of the generated paths are shown in Figure 18.

Network Structure		
Layer (type)	Output Shape	No. of Parameters
Linear-1	128	3712
Tanh-2	128	0
Linear-3	64	8256
Tanh-4	64	0
Linear-5	4 or 1	260 or 65

HYPERPARAMETERS	
γ	1
c_1	0.5
c_2	0.001
clip range	0.2
λ_{GAE}	0.95
learning rate	0.0001
n_{epochs}	30
batch size	64
n_{steps}	2048

Table 5: The maze game. **On the Left:** FNN architecture for both value and policy network. Note that they are not shared. Their structures are identical up to the last layer. The last layer has output shape 4 for the policy and 1 for the value network. Hyperbolic tangent is used as activation functions. Biases are enabled for the linear layers. **On the Right:** Hyperparameter settings of PPO.



(a) 20×20 map (b) 30×30 map (c) 40×40 map

Fig. 18: Results from the maze game. The training times are approximately (from left to right) 5, 10 and 30 minutes on a MacBook Pro (15-inch, 2018), where one optimization cycle of 30 PPO epochs takes approximately three seconds.

3.3.2 Solving Cartpole Problem with Echo State Networks

We are interested in using ESNs as RL agents. The chosen environment to test ESNs in an RL setup is the cartpole environment, which is introduced in Section 3.1.1. The environment yields the cart's current position and velocity and the pole's current angle and angular velocity. Thus, the cartpole problem is an MDP.

As discussed in Section 2.3, ESNs have the advantage of optimizing their readouts by means of linear regression as opposed to gradient-based optimization methods. It is of interest to retain this feature in an RL setup. A recursive least squares (RLS) algorithm proposed in the work by Zhang et al. [132] is used to train LiESNs on the cartpole environment. The algorithm strongly builds on the ideas from DQL. A pseudocode of an analogous algorithm can be found in [131]. The contributions of the ESNRLS-Q method are the replacement of the gradient descent step in DQL by an RLS update and the incorporation of ESNs in DQL with mini-batching mechanism. Each sample in the experience replay buffer of the ESNRLS-Q algorithm is a series consisting of T successive transitions. If a terminal state is encountered before the series is filled, it is forward filled with the last transition it contains. An on-policy version of the algorithm with the name ESNRLS-Sarsa is also proposed. Unfortunately, we have not been able to replicate the authors' results. Therefore, we report on a slightly altered version of their proposed algorithm. The difference lies in the calculation of the target value in (3.3.53). To give an overview on the algorithm, we start with the specification of the ESNs. The reservoir size is set to 256. The reservoir states are initialized to $\mathbf{0}$ and updated according to

$$\mathbf{h}_{t,i}^{(k)} = \mathbf{h}_{t,i}^{(k-1)} + \mathbf{f}(\mathbf{W}^{\text{in}}[1; (\mathbf{s}_{t,i}^{(k)})^\top]^\top),$$

where k stands for the k^{th} transition in the i^{th} sample $(\mathbf{s}_{t,i}^{(k)}, a_{t,i}^{(k)}, \mathbf{s'}_{t,i}^{(k)}, r_{t,i}^{(k)})$ in the current mini-batch $\mathcal{M}_t = \{(\mathbf{s}_{t,i}^{(k)}, a_{t,i}^{(k)}, \mathbf{s'}_{t,i}^{(k)}, r_{t,i}^{(k)})\}_{i=1,\dots,M;k=1,\dots,T}$. The input weights $\mathbf{W}^{\text{in}} \in \mathbb{M}_{256\times 5}$, policy and target readouts $\mathbf{W}^{\text{out}}, \mathbf{W}^{\text{out}}_{target} \in \mathbb{M}_{2\times 260}$ and the output biases $\mathbf{b}^{\text{out}}, \mathbf{b}^{\text{out}}_{target} \in \mathbb{R}^2$ are initialized randomly. The reservoir activation $\mathbf{f}$ is set to be the ReLU activation function and the batch size M is set to 64. Note that the observation and action spaces of the problem are four and two dimensional, respectively. Although it is not a part of the original ESNRLS algorithm, we set the length of each series T to 10 and use the first 5 transitions to reduce the effects of the initial transient. This leads to a $\widetilde{T} = 5$ effective series length. In our experience, this method benefits the training performance. The rest of the algorithm is analogous to DQL. The Q values of the policy network is given by

$$Q(\mathbf{S}_t^{(k)}, \mathcal{A}, \Theta_{t-1}) = \Theta_{t-1} \cdot \mathbf{U}_t^{(k)},$$

where

$$\begin{aligned}
\Theta_{t-1} &= [\mathbf{W}^{\text{out}}; \mathbf{b}^{\text{out}}] \in \mathbb{M}_{2\times 261},\\
\mathbf{U}_t^{(k)} &= [(\mathbf{S}_t^{(k)})^\top; (\mathbf{H}_t^{(k)})^\top; \mathbf{1}]^\top \in \mathbb{M}_{261\times M},\\
\mathbf{S}_t^{(k)} &= [\mathbf{s}_{t,1}^{(k)}; \dots; \mathbf{s}_{t,M}^{(k)}] \in \mathbb{M}_{4\times M},\\
\mathbf{H}_t^{(k)} &= [\mathbf{h}_{t,1}^{(k)}; \dots; \mathbf{h}_{t,M}^{(k)}] \in \mathbb{M}_{256\times M},
\end{aligned}$$

and $\mathbf{1} \in \mathbb{R}^M$ is a full-one vector. Then the target value is calculated using the target network, given by

$$Q^\pi(\mathbf{S}_t^{(k)}, \mathcal{A}) = [\mathbf{r}_t^{(k)}; \mathbf{r}_t^{(k)}]^\top + \gamma \cdot Q^\pi(\mathbf{S}_\mathbf{t}^{\prime(\mathbf{k})}, \mathcal{A}, \widetilde{\Theta}), \tag{3.3.53}$$

where

$$\begin{aligned}\widetilde{\Theta} &= [\mathbf{W}^{\text{out}}_{target}; \mathbf{b}^{\text{out}}_{target}] \in \mathbb{M}_{2\times 261},\\ \mathbf{S}'^{(k)}_t &= [\mathbf{s}'^{(k)}_{t,1}; \ldots; \mathbf{s}'^{(k)}_{t,M}] \in \mathbb{M}_{4\times M},\\ \mathbf{r}^{(k)}_t &= (r^{(k)}_{t,1}, \ldots, r^{(k)}_{t,M})^\top.\end{aligned}$$

Moving on to the proposed mean-approximation method for the RLS update, we average over the elements in the series and the samples in the mini-batches to acquire a TD error $Q^\pi_t - Q_t$ at each iteration, given by

$$\begin{aligned}\bar{\mathbf{u}}_t &= \frac{1}{M\widetilde{T}}\sum_{k=1}^{\widetilde{T}}\sum_{i=1}^{M}\mathbf{U}^{(k)}_{t,i}, \text{ where } \mathbf{U}^{(k)}_{t,i} \text{ is the } i^{th} \text{ column vector of } \mathbf{U}^{(k)}_t,\\ \bar{Q}^\pi_t &= \frac{1}{M\widetilde{T}}\sum_{k=1}^{\widetilde{T}}\sum_{i=1}^{M}Q^\pi(\mathbf{s}^{(k)}_{t,i}, \mathcal{A}), \text{ where } Q^\pi(\mathbf{s}^{(k)}_{t,i} \text{ is the } i^{th} \text{ column vector of } Q^\pi(\mathbf{S}^{(k)}_t, \mathcal{A}),\\ \bar{Q}_t &= \frac{1}{M\widetilde{T}}\sum_{k=1}^{\widetilde{T}}\sum_{i=1}^{M}Q(\mathbf{s}^{(k)}_{t,i}, \mathcal{A}, \Theta_{t-1}),\end{aligned}$$

where $Q(\mathbf{s}^{(k)}_{t,i}, \mathcal{A}, \Theta_{t-1})$ is the i^{th} column vector of $Q(\mathbf{S}^{(k)}_t, \mathcal{A}, \Theta_{t-1})$. The output weights are then updated with the learning rate

$$\alpha_t = \frac{P^\top_{t-1}}{\lambda + \mathbf{v}^\top_t \bar{\mathbf{u}}_t}$$

and the gradient term

$$\nabla_{t-1} = (\bar{Q}^\pi_t - \bar{Q}_t)\bar{\mathbf{u}}^\top_t$$

and the regularization term

$$L_1 = -\kappa \cdot \mathbf{sgn}(\Theta_{t-1})P^\top_{t-1},$$

yielding

$$\Theta_t = \Theta_{t-1} + \nabla_{t-1}\alpha_t + L_1,$$

where **sgn** is the signum function, $\kappa = 10^{-5}$, $\lambda = 0.99999$, $P_0 = 0.4\mathbf{I} \in \mathbb{M}_{261\times 261}$ and

$$\begin{aligned}P_t &= \frac{1}{\lambda}(P_{t-1} - \mathbf{g}_t\mathbf{v}^\top_t),\\ \mathbf{v}_t &= P_{t-1}\bar{\mathbf{u}}_t,\\ \mathbf{g}_t &= \frac{\mathbf{v}_t}{\lambda + \mathbf{v}^\top_t \bar{\mathbf{u}}_t}.\end{aligned}$$

Training is done over 100 episodes, which takes about 12 minutes[3]. An episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from

[3]A MacBook Pro (15-inch, 2018) has been used for the computations.

the center, or the maximum length of an episode (200 time steps) is reached. Each successful transition is rewarded with a scalar reward 1. During training, if a terminal state is reached without reaching the maximum episode length, a reward of -10 is given. The training and validation results are shown in Figure 19. For validation, the agent plays the game for 1000 episodes and each episode's length, which corresponds to the total reward, is registered. We see that the convergence in training is not stable, where the policy fluctuates by large margins. Despite the fact that the training convergence is not satisfactory, the agent is able to complete the game about 80% of the time over 1000 trials with an approximate mean reward of 192.

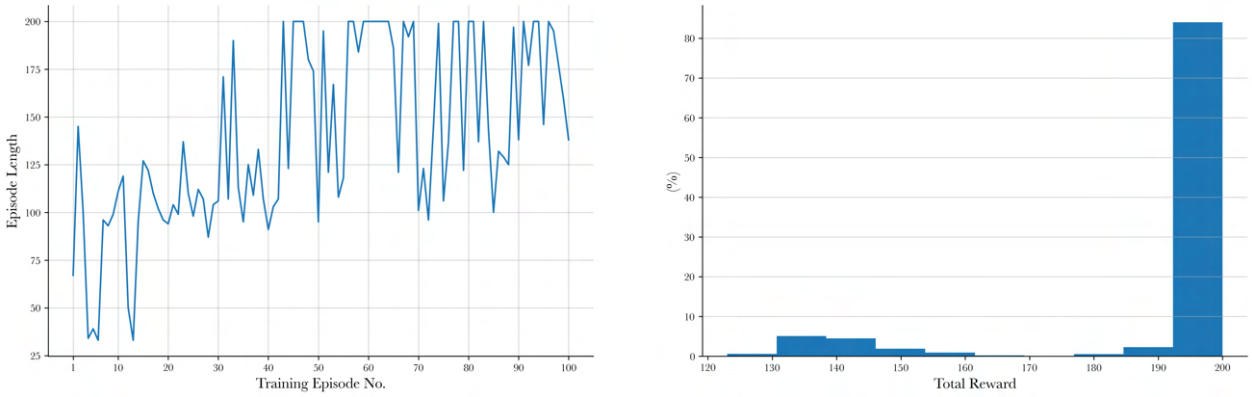


Fig. 19: Training and validation performances of the ESNRLS method on the cartpole environment. **On the Left:** Episode lengths during training. The training scheme is analogous to DQL algorithm 4. **On the Right:** Histogram of the durations of 1000 validation episodes.

As a next step, the RLS method is compared to PPO, where the readout of the ESN is trained by gradient-based optimization. For this purpose, customizations on the original PPO algorithm are made. Firstly, we report on our finding, which is the fact that reducing the number of randomly sampled batches in Line 14 of Algorithm 5 from $T \cdot n_{envs}$ to T improves the training and validation performances of our models by possibly reducing batch correlations further and having a regularizing effect on the training. Consequently, training times are reduced too. Secondly, Algorithm 5 needs to be adapted to trainings with reservoirs. This involves using dynamic reservoir shapes, where for a reservoir of size N, a reservoir layer

$$\mathbf{H} = [\mathbf{h}_1; \ldots; \mathbf{h}_D] \in \mathbb{M}_{N \times D}, \tag{3.3.54}$$

is required during rollout collection phase with $D = n_{envs}$ and during the optimization steps the reservoir update (2.2.25) is done using batches of size M, i.e. $D = M$. Lastly, the reservoir nodes are set to $\mathbf{0} \in \mathbb{R}^N$ before each reservoir update, i.e. the ESN functions like an ELM with the difference that the unprocessed inputs and the reservoir layer are concatenated (see (2.2.26)) before reaching the readout layer. After these modifications, ESNs are directly plugged into the original PPO algorithm to solve the cartpole problem. Our results are shown in Figure 20. We see that the training with PPO yields better results than the ESNRLS method, both in terms of training and validation performances. The training convergence is stable and a perfect validation score is obtained with the ESN. In addition, the training takes about 45 seconds, whereas this is 12 minutes with the ESNRLS method[4].

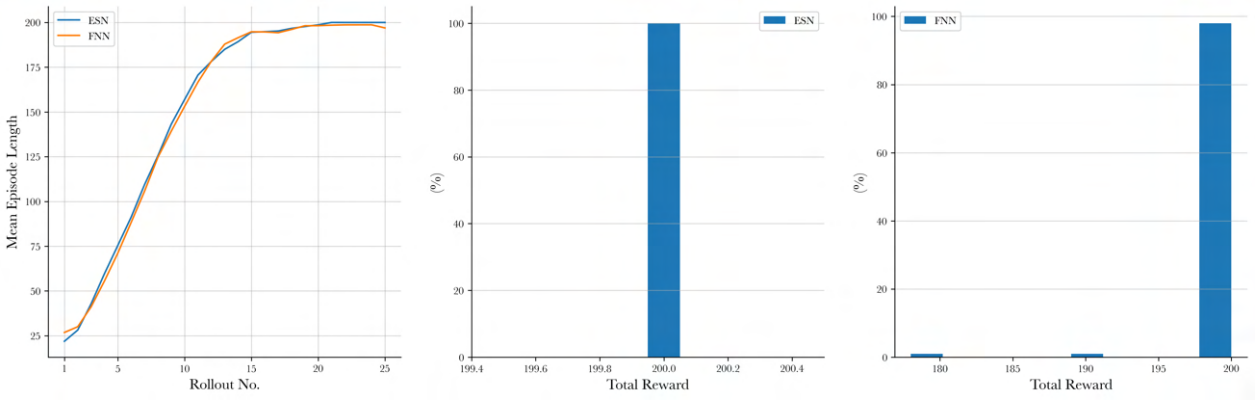


Fig. 20: Training and validation performances of ESN and FNN on the cartpole (MDP) problem. On the left, training convergence curves of ESN and FNN are shown. Mean over the episode lengths in the rollout buffer is calculated for each rollout. Both models are validated for 100 episodes. Histograms show the distribution of validation episode lengths for ESN and FNN. Mean validation episode lengths for ESN and FNN are 200 and 199.67, respectively.

ESN		
Layer (type)	Output Shape	No. of Parameters
BatchNorm-1	4	8
ESN-2	64	8512
Tanh-3	64	0
Linear-4	2 or 1	130 or 65
Total number of trainable parameters: 17,235		
FNN		
Layer (type)	Output Shape	No. of Parameters
BatchNorm-1	4	8
Linear-2	128	640
Tanh-3	64	0
Linear-4	64	8256
Tanh-5	64	0
Linear-6	2 or 1	130 or 65
Total number of trainable parameters: 18,003		

HYPERPARAMETERS	
ESN	
reservoir size	128
leak rate	1
spectral norm	0.99
reservoir activation	**tanh**
bias strength	1
PPO	
γ	0.99
c_1	0.5
c_2	0
clip range	0.2
λ_{GAE}	0.95
learning rate	0.0003
n_{epochs}	10
batch size	64
n_{steps}	2048
n_{envs}	1

Table 6: Cartpole (MDP). **On the Left:** ESN and FNN network architectures used in the cartpole experiment. The policy and value networks are not shared. Their structures are identical up to the last layer. The last layer has output shape 2 for the policy and 1 for the value network. Hyperbolic tangent is used as activation functions. Biases are enabled for the linear layers. **On the Right:** Hyperparameter settings of PPO and the ESN, which are manually selected.

We compare FNNs to ESNs, both having similar number of parameters. The network architectures and hyperparameters are shown in Table 6. Although the ESN has slightly better validation score, both models display similar training and validation performances. This is expected, since the task is an MDP and similar number of parameters are used. Hence, ESNs are not used in an appropriate setting to show an advantage over FNNs in the sense that there is no path dependence to capture from the observed signals. If path dependence is present, the reduction from (3.1.30) to (3.1.31) is not possible. By storing the information of previous transitions in (3.1.30) in the form of echo states

$$\mathbf{h}_t = E(s_t, a_t, r_t, s_{t-1}, a_{t-1}, \dots, r_1, s_0, a_0)$$

with the echo function E, whose discussion is given in Section 2.1.3, Markov property

$$Pr(s_{t+1} = s', r_{t+1} = r' \mid \mathbf{h}_t) \approx Pr(s_{t+1} = s', r_{t+1} = r' \mid s_t, a_t, r_t, s_{t-1}, a_{t-1}, \dots, r_1, s_0, a_0)$$

can be recovered. This is demonstrated in the next section.

3.3.3 Partially Observable Cartpole Problem

Recalling the discussion in Section 3.1.1, the Markovian cartpole problem is turned into a Partially Observable Markov Decision Process (POMDP) by removing the information of the cart's velocity and the pole's angular velocity from the observation space. The new problem requires memory, i.e. the agent needs to reconstruct the velocities from past observations [90], which consist of the past locations of the cart and the past angles of the pole. This can be accomplished by RNNs, such as ESNs.

In order to solve the partially observable cartpole (PO-Cartpole) problem with ESNs, we propose modifications to the PPO algorithm, additional to the ones discussed in Section 3.3.2, to establish recurrent processing of the observations. In the original PPO algorithm, batches sampled from the buffer contain transitions across all environments, i.e. the replays collected from different environments are mixed together. By separating experiences from different environments and sampling the t^{th} mini-batch of size M belonging to the i^{th} environment, which contains consecutive transitions and estimates

$$\{(s^i_{t,k}, a^i_{t,k}, r^i_{t,k}, V^{\pi_{\theta_{old}}}(s^i_{t,k}), \pi_{\theta_{old}}(\cdot \mid s^i_{t,k}), \hat{A}^i_{t,k})\}_{1 \le k \le M}, \tag{3.3.55}$$

the reservoir states can be updated according to

$$\widetilde{\mathbf{h}}^i_t(k) = (1-\alpha)\widetilde{\mathbf{h}}^i_t(k-1) + \alpha\, \mathbf{f}(\mathbf{W}^{\text{in}}\mathbf{u}^i_t(k) + \mathbf{W}\widetilde{\mathbf{h}}^i_t(k-1)),$$

where $\mathbf{u}^i_t(k)$ is the observation $s^i_{t,k}$ and $\widetilde{\mathbf{h}}^i_t(0) = \mathbf{0}\ \forall i \in \{1, \dots, n_{envs}\}$. Subsequently, with the same idea as in (3.3.54), we assemble the reservoir layer

$$\widetilde{\mathbf{H}}^i_t = [\widetilde{\mathbf{h}}^i_t(1); \dots; \widetilde{\mathbf{h}}^i_t(M)] \in \mathbb{M}_{N \times M},\ \forall i \in \{1, \dots, n_{envs}\}. \tag{3.3.56}$$

During the rollout, the reservoir layer

$$\mathbf{H}(n) = [\mathbf{h}_1(n); \dots; \mathbf{h}_{n_{envs}}(n)] \in \mathbb{M}_{N \times n_{envs}} \tag{3.3.57}$$

[4] A MacBook Pro (15-inch, 2018) has been used for the computations.

is used, where the usual reservoir update (2.2.25) is applied to $\mathbf{h}_j$ using the observations from the j^{th} environment. Upon reaching a terminal state, $\mathbf{h}_j$ is set to its initial state $\mathbf{0}$. The reservoir layer $\mathbf{H}$ can be thought of as having a separate reservoir for each environment. The customized PPO algorithm is summarized in Algorithm 6, where the policy π has reservoir layer $\mathbf{H}$ from (3.3.57) and $\widetilde{\mathbf{H}}_t^i$ from (3.3.56) during rollout and optimization, respectively. Its parameters θ are given by the ESN's readout $\mathbf{W}^{\text{out}}$. The algorithm is successfully applied to the PO-Cartpole problem and the results are shown in Figure 21. Network architectures and hyperparameters can be found in Table 7. Compared to the Markovian cartpole problem, a longer training session is required for the PO-Cartpole problem. The new algorithm 6 does not increase the computational complexity in terms of time, where both Markovian cartpole and PO-Cartpole have PPO epochs with a duration of approximately 0.3 seconds[5].

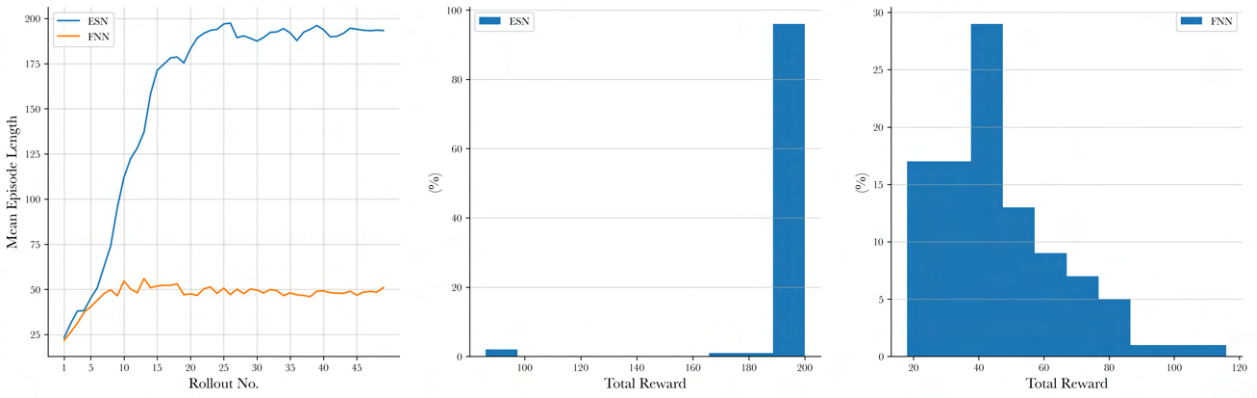


Fig. 21: Training and validation performances of ESN and FNN on the PO-Cartpole problem. The training of FNN and ESN takes about 5 and 9 minutes on a MacBook Pro (15-inch, 2018), respectively. On the left, training convergence curves of the ESN and the FNN are shown. Mean over the episode lengths in the rollout buffer is calculated for each rollout. Both models are validated for 100 episodes. Histograms show the distribution of validation episode lengths for the ESN and the FNN. Mean validation episode lengths for the ESN and the FNN are 197.31 and 46.22, respectively.

3.4 Conclusion

After building up the necessary foundation, popular RL optimization methods are elaborated on, including the state of the art PPO algorithm in Section 3.1. In quest of optimal ways to incorporate ESNs in an RL setup, ESNRLS and PPO methods are introduced slight modifications and successfully applied to the Markovian cartpole problem with ESNs in Section 3.3.2. Although ESNs are conventionally trained by least squares methods, our observations conveyed in Section 3.3.2 have led us to the decision of using PPO as the RL optimization method to train ESNs. Note that in terms of its usage in the Markovian cartpole problem with PPO, the ESN is equivalent to an ELM. In order to solve the path dependent problem, PO-Cartpole, a customized PPO algorithm to train ESNs is presented in Algorithm 6 and successful results are obtained with it (see Figure 21).

[5]A MacBook Pro (15-inch, 2018) has been used for the computations.

ESN		
Layer (type)	Output Shape	No. of Parameters
BatchNorm-1	2	4
ESN-2	64	32,192
Tanh-3	64	0
Linear-4	2 or 1	130 or 65
Total number of trainable parameters: 64,587		
FNN		
Layer (type)	Output Shape	No. of Parameters
BatchNorm-1	2	4
Linear-2	500	1,500
Tanh-3	64	0
Linear-4	64	32,064
Tanh-5	64	0
Linear-6	2 or 1	130 or 65
Total number of trainable parameters: 67,331		

HYPERPARAMETERS	
ESN	
reservoir size	500
leak rate	1
spectral norm	0.99
reservoir activation	**tanh**
input bias strength	1
PPO	
γ	0.99
c_1	0.5
c_2	0
clip range	0.2
λ_{GAE}	0.95
learning rate	0.0003
n_{epochs}	30
batch size	256
n_{steps}	2048
n_{envs}	5

Table 7: Cartpole (POMDP). **On the Left:** ESN and FNN network architectures used in the PO-Cartpole experiment. The policy and value networks are not shared. Their structures are identical up to the last layer. The last layer has output shape 2 for the policy and 1 for the value network. Hyperbolic tangent is used as activation functions. Biases are enabled for the linear layers. **On the Right:** Hyperparameter settings of PPO and the ESN, which are manually adjusted.

Algorithm 6 PPO for ESN

Initialize number of environments n_{envs}
Initialize number of PPO epochs n_{epochs}
Initialize maximum number of time steps n_{max}
$n \leftarrow 0$
while $n < n_{max}$ **do**
    Initialize buffer $\mathcal{D}$ with size $T \cdot n_{envs}$
    Initialize $\mathbf{H}$ from (3.3.57) with $[\mathbf{h}_1; \dots; \mathbf{h}_{n_{envs}}]$
    for $t = 1, 2, \dots, T$ **do**
        Compute $\{(s_t^i, a_t^i, r_{t+1}^i, V^{\pi_{\theta_{old}}}(s_t^i), \pi_{\theta_{old}}(\cdot \mid s_t^i))\}_{i=1,\dots,n_{envs}}$ and store in $\mathcal{D}$
        if s_t^i terminal **then**
            $\mathbf{h}_i \leftarrow \mathbf{0}$
        end if
    end for
    Compute advantage estimates $(\hat{A}_1^i, \dots, \hat{A}_T^i)_{i=1,\dots,n_{envs}}$ and store in $\mathcal{D}$
    $n \leftarrow n + T \cdot n_{envs}$
    for epoch = 1,2,$\dots, n_{epochs}$ **do**
        $n_{index} \leftarrow 1$
        Initialize $I = \{1, \dots, n_{envs}\}$
        Permute the elements of I
        while $n_{index} \leq n_{envs}$ **do**
            $i \leftarrow n_{index}^{th}$ element of I
            Collect the batch $\mathcal{D}^i$ belonging to the i^{th} environment from $\mathcal{D}$
            Randomly sample an ordered mini-batch of size $M \leq T$ from $\mathcal{D}^i$
            Compute $\widetilde{\mathbf{H}}$ from (3.3.56)
            Optimize surrogate $L^{CLIP+VF+S}$ on the mini-batch wrt. θ
            $n_{index} \leftarrow n_{index} + 1$
        end while
    end for
    $\theta_{old} \leftarrow \theta$
end while

4 Machine Learning for High-Frequency Trading

HFT is a form of algorithmic trading, operating at the microstructure level to exploit inefficiencies and predictable behaviours in a market. The main objective of a high-frequency trading algorithm is to act on arising opportunities before a competitor does. Hence, slippage is one of the major concerns in HFT, which refers to the latency cost due to the discrepancy between the quote observed at the time of an order placement and the realized market price. The type of the order in the context of slippage is an active order type called Market Order (MO). By sending MOs, traders execute immediate trades at the best available price. This includes paying the *bid-ask spread*, which is the difference between the best ask and the best bid price. It can be seen as the premium paid in exchange for immediate order execution. In addition to the speed of trading activity, slippage is dependent on the available liquidity and the amount of arriving MOs to consume liquidity. Simultaneously placed MOs cause fast changes in the market prices due to the consumption of the finite liquidity. This type of change in the market prices following an order are referred to as *market impact* or *price impact*. Costs can be incurred due to market impact, if the transaction itself changes the market price. In case control over the price at which the trade is executed is prioritized rather than the timing of it, the passive order type named Limit Order (LO) is utilized. The type of costs associated to LOs are called *opportunity costs*, which occur when the market price does not cross the specified limit price, causing the order to fail. The aforementioned transaction costs go under the category of *implicit execution costs* as opposed to *transparent execution costs*. An example of transparent execution costs is the broker commission, which is paid for connectivity to exchanges, facilitation of order executions and other custom services [47, 6, 19].

The accumulated state of LOs are registered in a so-called Limit Order Book (LOB). In our study, LOBs are used as the primary data set to accomplish the ML task of training randomized neural networks by means of RL in a high-frequency trading setup. Our results are given in Section 4.2.2 & 4.2.3. In Section 4.2.2, the bid-ask spread is not taken into account, where the agents trade on mid-prices. Then, small artificial spreads are introduced in Section 4.2.3. In both sections, market impact and latency costs are neglected. The next section gives a description of our LOB data as well as an additional data type called Trade Books (TBs).

4.1 Limit Order and Trade Books

Borsa Istanbul (BIST) is the exchange entity bringing all the exchanges operating in the Turkish capital markets together under the same roof [3] and Garanti BBVA with the ticker symbol "GARAN" is one of the flagship stocks in BIST 30 index. The data used for model training is based on LOBs of GARAN from the year 2017. The entries in LOBs, which are of interest to this study are these liquid continuous trades, which happen during continuous session hours. Here, orders are matched at different price levels according to continuous trading facilitated by market makers, which allows increased liquidity to securities with high trading and market value compared to the other two session types in BIST, opening and closing sessions, where auctions at the open and the close are carried out. Then, the auctions conclude on a price level, which enables the execution of the maximum amount

of transactions. In summary, this single price method consists of order collection, price determination and execution of transactions [80, 126]. The continuous sessions in the morning are roughly between 10:00 a.m. and 13:00 p.m. followed by lunch breaks. Then, continuous sessions resume in the afternoon, roughly between 14:00 p.m. and 18:00 p.m. followed by the closing sessions. The given times refer to Turkey Time in 2017.

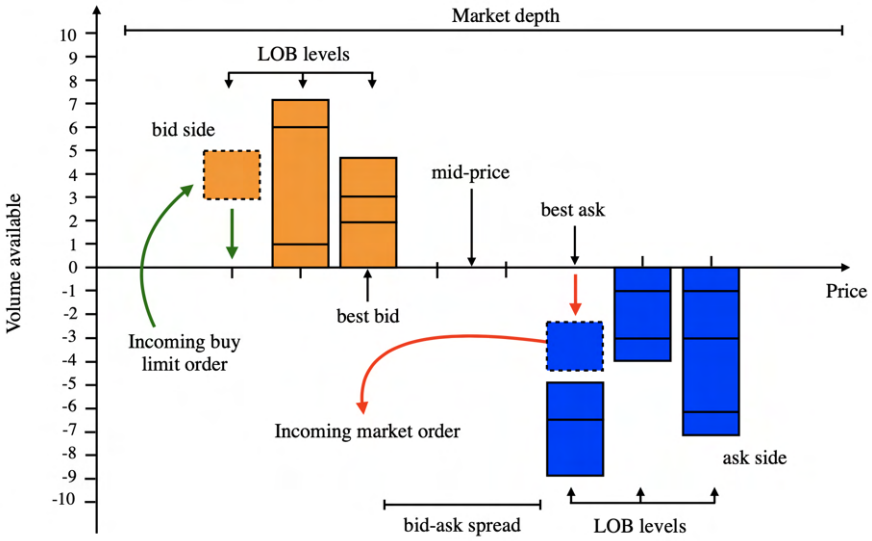


Fig. 22: Illustration of the limit order book dynamics. Picture taken from [13].

Limit order books are electronic files, which contain at any given point in time, on a given market, the list of all the transactions with information on orders sent to the market with their characteristics such as as the sign of the order (buy or sell), the price, the quantity, a timestamp giving the time the order was recorded by the market [2]. LOBs used in this study have level depth equal to 5, i.e. they contain the five best bid and ask prices and the respective number of shares with a milliseconds time precision (see Table 8). This makes up 21 different variables in a LOB together with the time entries. The rows of the LOBs, which contain these entries are often referred to as "bars" by finance practitioners [47]. We note that the order signs are not included in our LOB data. They are collected in TBs (see Table 9). TBs provide information on the orders, which get matched by the stock exchange. They include the time, the side, the size, the price and the initiation type of the orders, next to the session type during which the order is executed. There are two bars for each trade execution, namely one for the the buy side and one for the sell side, where the trade is initiated by one of the sides. Upon the matching of the sides, the accumulated limit order volumes are consumed, and replenished by incoming limit orders (see Figure 22). By matching the times of the entries in the LOBs and the session types given by TBs, we select continuous session bars in the LOBs, where opening and closing sessions are disregarded.

The standard bar sampling methods are time, tick, volume and dollar bars, most popular among them being the time bars [101]. Time bars are created by taking entries, which are apart by a fixed time interval. Volume bars are sampled upon the exchange of a predefined amount of units of a security. Taking the market value of transactions into account, dollar volumes are used as reference to create dollar bars. The change in the perception of the market activity through each sampling method is illustrated in Figure 23. Aiming to keep

time	bid1	bsize1	bid2	bsize2	bid3	bsize3	bid4	bsize4	bid5	bsize5	ask1	asize1	ask2	asize2	ask3	asize3	ask4	asize4	ask5	asize5
95508.179	7.61	110328	7.60	242605	7.59	40687	7.58	97797	7.57	150126	7.61	378930	7.62	121087	7.63	161225	7.64	71092	7.65	443326
95508.323	7.60	242605	7.59	40687	7.58	97797	7.57	150126	7.56	54095	7.61	268592	7.62	121087	7.63	161225	7.64	71092	7.65	443326
100000.123	7.60	242105	7.59	43187	7.58	97797	7.57	152626	7.56	54095	7.61	266342	7.62	121087	7.63	161225	7.64	71092	7.65	443326
100000.227	7.60	132105	7.59	43187	7.58	97797	7.57	152626	7.56	54095	7.61	266342	7.62	121087	7.63	161225	7.64	71092	7.65	443326
100000.334	7.60	132105	7.59	48187	7.58	107797	7.57	152626	7.56	64095	7.61	266342	7.62	121087	7.63	161225	7.64	71092	7.65	443326

Table 8: The first five bars of GARAN's LOB from the first trading day of the year 2017, Jan 2$^{\text{nd}}$. The bid and ask prices are denoted as "bid" and "ask", respectively. The best bid and ask prices are "bid1" and "ask1" and the worst bid and ask prices are "bid5" and "ask5", respectively. Their corresponding volumes are denoted as "bsize" and "asize". The prices are given in Turkish Lira (TL).

Date	Time	Side	Size	Price	TL	Session Type	Initiation Type
2017/01/02	09:55:08	S	1	7.61	7.61	Opening Session	P
2017/01/02	10:00:05	B	10	7.61	76.1	Continuous Session	A
2017/01/02	10:00:05	S	10	7.61	76.1	Continuous Session	P
2017/01/02	18:05:02	S	23257	7.6	176753.2	Closing Session	P

Table 9: Some snapshots from the TB of January. "A" and "P" stand for active and passive, respectively. "B" and "S" stand for buy and sell side, respectively. The TL column gives the dollar volume of the trade (in Turkish Liras). The second and the third bars represent the two parties matched for the buy side initiated transaction. During the opening and closing sessions, the TB accumulates bids and offers without matching them until a clearing price is determined. Bars of the TBs used in the study are updated with seconds precision except for November's TB with milliseconds precision.

the high-frequency information provided by our LOB data and considering that time bars have the problem of oversampling information during low-activity periods and undersampling during high-activity periods [101], we utilize tick bars, where each bar represents an update in the LOB due to the arrival of new market information, such as execution of an MO or incoming LO.

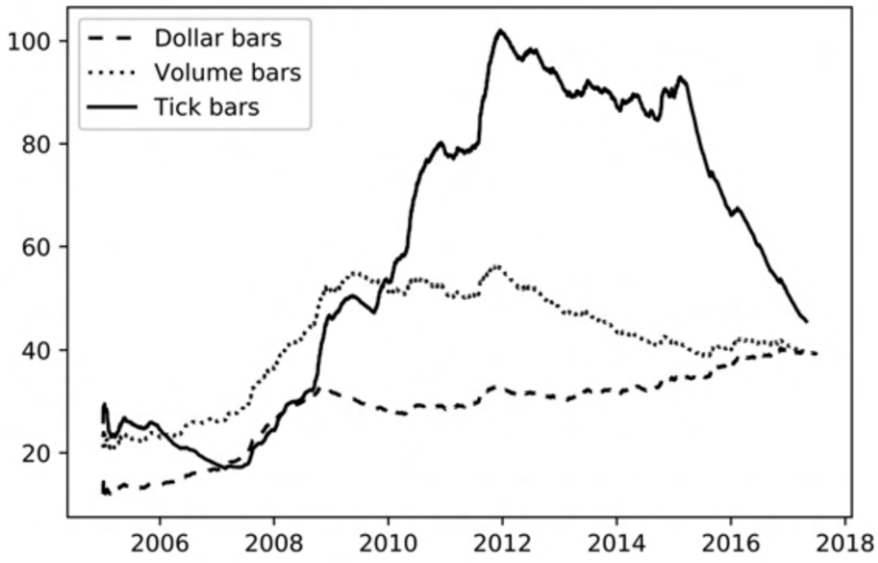


Fig. 23: Synchronization to market activity with different sampling rates. Picture taken from [101].

4.2 High-Frequency Trading Agents

Considering that the primary focus of ML is inferring predictive models from large data sets, applying ML techniques in HFT is a natural choice [47]. Our approach to the matter breaks down into handling high-frequency data by means of RL environments and inquiring the performance of high-frequency trading agents in the form of different neural network architectures, namely FNNs, ELMs and ESNs. Usage of ELMs and ESNs is motivated by the expectation of reduced training times, where it can be cumbersome to fully train a network in our rather complex RL setup involving high-frequency trading. Additionally, the reduction described by (3.1.30) and (3.1.31) in the presence of possible path dependencies in the LOBs is aimed to be achieved through the utilization of ESNs (see Section 3.3.2 & 3.3.3).

The RL framework that we work with is based on the work by Briola et al. [13]. The instructions therein are followed for data preparation. After applying a training/validation split with the ratio 3:2 on our annual LOB data, it is processed into smaller LOB samples with higher signal-to-noise ratios. This is done by splitting each training day's LOB data into 5 smaller LOBs to be scanned for large absolute relative differences between mid-prices

$$1 - \frac{\min_i m_i}{\max_i m_i}, \tag{4.2.58}$$

where

$$m_t = \frac{p_{a,t}^{best} + p_{b,t}^{best}}{2}, \quad t \in \{1, \dots, \widetilde{L}\}$$

is the mid-price from the t^{th} bar of the given LOB with total number of bars $\widetilde{L}$ and $p^{best}_{a,t}, p^{best}_{b,t}$ are the best ask and best bid prices, respectively. Among the training samples, 25 of them yielding the largest value for (4.2.58) are used as RL environments. The resulting environments' frequency and price information is provided in Figure 24. There, the returns are given by

$$\Delta m_t \coloneqq m_{t+1} - m_t, \quad t \in \{1, \dots, \widetilde{L} - 1\}. \tag{4.2.59}$$

The numbers of zero-returns are omitted from the histograms, but shown in their corresponding tables. The LOB update times are the time differences between each bar's timestamp in the LOB. The tick size in our LOBs is 0.01 TL. This results in a tick size of 0.005 TL for the mid-prices. Irregular time intervals in Figure 24 are observed due to market activities at varying intensities in a given day. The average number of bars per LOB sample in our training data set is approximately 4,135.

4.2.1 The Limit Order Book Environment

The RL environment that we use is a close replica of the one proposed in [13]. The agents are expected to enter positions based on the observations provided by the LOB environments to obtain excess returns. For t and $\tau > t$ being the opening and closing times of a position, respectively, the return associated with the position is given by

$$R_{l/s} = p^{best}_{b,\tau/t} - p^{best}_{a,t/\tau}, \tag{4.2.60}$$

where l and s stand for long and short positions, respectively. We note that (4.2.60) is not in alignment with the return defined by the authors of ref. [13], which is $\mathcal{R}_{l/s} = p^{best}_{a/b,\tau} - p^{best}_{b/a,t}$. Remembering that the ask price is always higher than the bid price, a sanity check would be to set $t = \tau$ and see if it leads to instantaneous profit. The authors' definition of the return leads to such instantaneous profits for long positions.

The states yielded by the environment at each time step are the last $\widetilde{N}$ LOB bars (including the current bar), the agent's current position (neutral, long or short) and the mark-to-market (MTM) value at the current time step. The (MTM) value is defined as the return that the agent would obtain upon deciding to close the current open position [13]. It is zero, in case the agent does not hold any position. The number of observed LOB bars $\widetilde{N}$ is kept as a variable for now. Additionally, only mid-prices are taken into the observation space, instead of bid and ask prices from all 5 levels. Thus, an observation has dimension

$$10(\widetilde{N} + 1) + 2. \tag{4.2.61}$$

At each time step, the agent is allowed to:

- short, long or do nothing (preserve his current position)
- buy, sell or hold at most one unit of GARAN
- execute a daily stop-loss

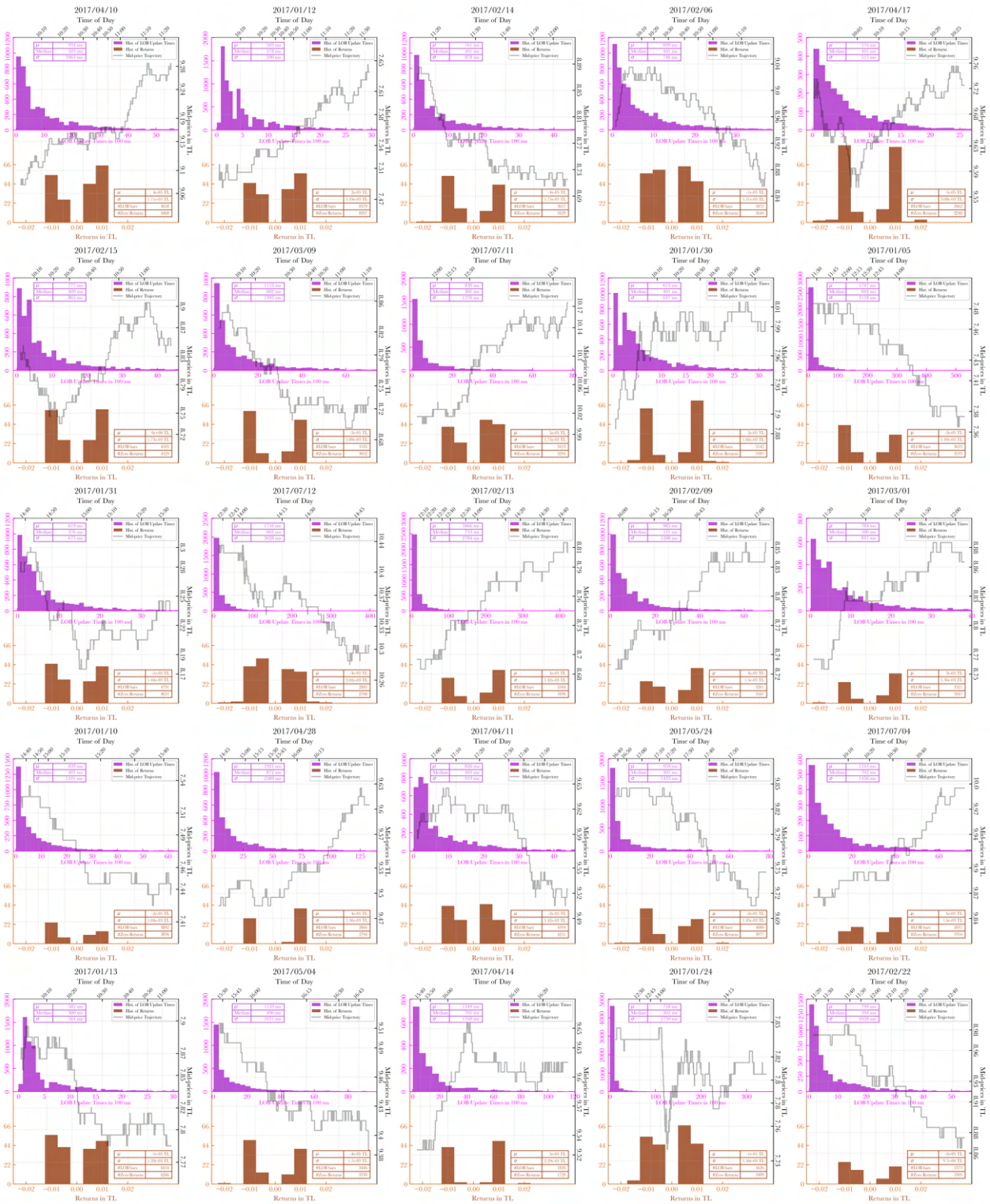


Fig. 24: The original LOBs of each trading day are subsampled into 25 LOBs with high signal-to-noise ratio. The figure shows the mid-price trajectories, the distribution of returns (4.2.59) and the distribution of the bar-update intervals in the subsampled LOBs. Note that the time jumps at noon are due to the lunch breaks (see Section 4.1).

For the daily stop-loss, the current state of profit and loss (P&L) is checked. In the case of a negative total return, any open position is closed and further trading is stopped, i.e. the environment is skipped. The action space of the agent at each time step is given by

$$\mathcal{A} = \{sell, stay, buy, stop\text{-}loss\}.$$

A long (short) position is entered upon a *buy* (*sell*) action. Through the *stay* action, no position is entered or the current position is held. Since the agent can possess only one unit of stock at each time step, a *buy* action while in long position is equivalent to a *stay* action, whereas a *sell* action closes the position. Similarly, a *sell* action while shorting would be equivalent to a *stay* action, whereas a *buy* action closes the position. Since the minimum available liquidity at any price level is one unit, allowing transactions with only one unit of GARAN makes disregarding market impact costs more reasonable. However, in a realistic scenario, the agent could consume the available liquidity at a price level through multiple trades during the complete trading session. As mentioned in the introduction of Section 4, a market impact cost analysis of the trades of our agents is left out in this study.

A difference in our experiments compared to the the work [13] should be noted, which is that our agents trade on mid-prices, where the return is given by

$$R_{l/s}^{mid} = m_{\tau/t} - m_{t/\tau}. \tag{4.2.62}$$

In a more realistic setup, where the bid-ask spread is taken into account, our agents are not able to achieve any learning on the training samples but complete the training with a total reward of zero, i.e. there is no net loss or gain. Same results are obtained for validation. A setup with small artificial spreads is considered in Section 4.2.3. The dynamics of our LOB environment is shown in Table 10. The goal of the agent is to make the correct decision in terms of entering a short or a long position at the best possible time. If succeeded, the agent can realize a profit by looking at the MTM value, which is a part of the observation space. Consider the case, where a wrong position is entered in the sense that it is the opposite position, which leads to a profit. The agent could wait until the MTM becomes positive to make up for the wrong decision. Then, not only it may never be the case but also other possibly profitable opportunities are missed out on. To build further intuition, observe the price evolution $(m_t)_{1\leq t\leq 4} = (7.5, 7.6, 7.4, 7.4)$. If the agent enters a long position at 7.5 and closes at 7.6, it is worse than entering a short position at 7.6 and closing at 7.4. Note that the agent is not allowed to close the long position at 7.6 and enter a short position at the same time step. The problem of achieving the maximum return in the given setup is not trivial.

4.2.2 Mid-price Trading

In this section, we provide our results on the performance of our high-frequency trading agents, specifically in the mid-price trading case. The PPO algorithm is the chosen optimization method to train the agents. As mentioned in Section 4.2, we employ ESNs to capture possible path dependencies in the LOBs, which could generate better strategies yielding larger returns. Therefore, Algorithm 6 with the recurrent processing of the observations is an apparent choice for the training of ESNs. We report that the usage of this algorithm

	Observation														
time	mid-price	bsize1	bsize2	bsize3	bsize4	bsize5	asize1	asize2	asize3	asize4	asize5	MTM	position	action	reward
2017-04-10 10:00:03.724	9.085	23201	331244	233557	384130	26609	237469	367265	233265	153218	163187	0	neutral	*sell*	0
2017-04-10 10:00:03.824	9.08	331244	233557	390170	26609	55065	237469	367265	239305	153218	163187	0.005	short	*buy*	0.005
2017-04-10 10:00:03.927	9.085	10000	331244	233557	396210	26609	237469	367265	245345	153218	163187	0	neutral	*buy*	0
2017-04-10 10:00:04.131	9.085	8500	331244	233557	396210	26609	237469	367265	245345	153218	163187	0	long	*stay*	0
2017-04-10 10:00:04.637	9.075	299744	239597	396210	26609	55065	4133	237469	373305	245345	153218	−0.01	long	*sell*	−0.01
2017-04-10 10:00:04.735	9.075	299744	251597	396210	26609	55065	4133	237469	373305	245345	153218	0	neutral	*stop-loss*	0

Table 10: A demonstration of the agent-environment interactions. Upon observing the current state of the LOB, the agent decides on an action and collects the resulting reward (4.2.62).

in the HFT setting with multiple LOB environments results in large KL divergences. The reason for this lies in the design of the customized algorithm. As discussed in Section 3.3.3, Algorithm 6 does not mix experiences from different environments together, i.e. agents are trained on one environment after the other. Considering the way that the LOB environments are sampled and looking at the resulting samples shown in Figure 24, it is apparent that each LOB environment is different than the other in terms of state spaces. The large KL divergences that we encounter might be pointing at that the optimization of the policy on one LOB sample causes it to step towards behaviours, which are adequate in that environment but not on the next one. This would lead to large policy updates during the training on the subsequent LOB and the model would not be able to generalize over different LOBs. Hence, mixing of experiences across different environments in each batch as in the original PPO algorithm is necessary. The procedure followed in the case of PO-Cartpole cannot be followed here. Fortunately, the states of the LOBs are not dependent on the actions of the agents, since market impact is not taken into account. Note that this is not the case in the PO-Cartpole problem and there, the usage of Algorithm 6 is necessary. The independence of the LOB states from the agents' actions allows the offline computation of the echo states

$$\begin{aligned} &\mathbf{x}_t = (1-\alpha)\mathbf{x}_{t-1} + \alpha\,\mathbf{f}(\mathbf{W}\mathbf{x}_{t-1} + \mathbf{W}^{\text{in}}[1;\widetilde{\mathbf{s}}_t^\top]^\top), \quad t \in \{1,\dots,\widetilde{L}\}, \\ &\mathbf{x}_0 = \mathbf{0}, \end{aligned} \tag{4.2.63}$$

where the input $\widetilde{\mathbf{s}}_t$ is the t^{th} bar of a LOB with $\widetilde{L}$ bars. Then, the collected reservoir states $\mathbf{x}_t$ for each bar $\widetilde{\mathbf{s}}_t$ can be made part of the original observation space $\mathcal{S}$, given by

$$\mathbf{S}_t = \left[\mathbf{x}_t^\top; \mathbf{s}_t^\top\right]^\top \in \mathcal{S}_{res},$$

where

$$\begin{aligned} &\mathbf{s}_t = \left[\widetilde{\mathbf{s}}_t^\top; \text{MTM}_t; \text{position}_t\ \right]^\top \in \mathcal{S}, \\ &\widetilde{\mathbf{s}}_t = (m_t, \text{bsize1}_t, \dots, \text{bsize5}_t, \text{asize1}_t, \dots, \text{asize5}_t)^\top \in \mathbb{R}^{11}. \end{aligned} \tag{4.2.64}$$

The volumes in (4.2.64) are squashed into the range $[0,1]$ by means of the logarithm with base 10^{10}. For the prices, the logarithm with base 100 is used and the mid-prices are calculated accordingly. In our experience, this procedure is not a necessity for the static networks, which do not use echo states and similar results are obtained without it. In terms of the calculation of echo states without the squashing, sufficient experimentation has not been carried out. Yet it is a standard procedure to scale the inputs to fit them into the interval

$[-1, 1]$ when working with ESNs [61, 72]. After forming the new observation space $\mathcal{S}_{res}$ with echo states, the original PPO algorithm can be used with the inputs $\mathbf{S}_t$. This procedure increases the input dimension of the networks by the size of the reservoir. Note that the MTM values and the agent's previous positions, which are a part of the observation space, cannot be included in the offline calculation of the reservoir states, since they depend on the agent's actions. Inclusion of MTM values together with the position types in the reservoir states would provide the extra information of the prices at which the previous positions are entered. But this would be adding noisy information to the reservoir, since the MTM values and the position types are results of unoptimized actions. Furthermore, the best possible decision of entering or closing a position at a certain time step is dependent on future price movements, which are independent of the agent's previous positions and the types of these. This is the case due to the fact that we do not factor in any market impact. Improving upon current decisions by learning from past decisions is accomplished by the RL algorithm, which utilizes past states containing the MTM values and the position types.

HYPERPARAMETERS (PPO)										
γ	c_1	c_2	clip range	λ_{GAE}	learning rate	n_{epochs}	batch size	n_{steps}	n_{envs}	KL_{target}
0.9587	0.579	0.000645	0.2	0.936	0.000942	30	4000	4000	25	0.0207054

Table 11: Hyperparameter settings of PPO used for the training of high-frequency trading agents. KL_{target} stands for the desired maximum KL divergence.

As reported in Section 3.3.2, reducing the number of randomly sampled batches in Line 14 of Algorithm 5 from $T \cdot n_{envs}$ to T improves our results. We employ this modification here as well. The agents are trained on LOBs subsampled from the first 60% of the year 2017. A training consists of 250 PPO rollouts. During each rollout, the agents in each environment trade for as many episodes as they can until a total number of steps n_{steps} is reached. By taking the mean of the rewards collected during each environment's rollout over all episodes, the total mean rewards

$$\widetilde{R} \coloneqq \frac{1}{n_{ep}} \sum_{i=1}^{n_{envs}} \sum_{k=1}^{n_{ep}} \widetilde{R}_k^i \tag{4.2.65}$$

is calculated, where $\widetilde{R}_k^i$ denotes the total reward obtained from the k^{th} episode in the i^{th} environment's rollout. After each training session, the model state at the iteration with the best rollout in terms $\widetilde{R}$ is taken to be validated. Data from the rest of the year (40%) is reserved for validation and used as it is, i.e. no subsampling procedure is carried out. Hyperparameters of PPO are optimized using Optuna [5], where the validation performance of the FNN used in [13] is taken as reference. The validation references for PPO's hyperparameter optimization are sampled in the same way as the subsampled LOBs used for training. Batch size and the number of rollout steps n_{steps} are set manually. The resulting hyperparameters are shown in Table 11. In terms of network architectures, the number of layers and the number of nodes in each layer are adjusted manually. Hyperbolic tangent is chosen as the activation function between layers. For the training of the models, GPUs on the central clients of the department of mathematics of ETH Zurich are utilized. Training sessions take two to three hours, approximately. We observe that training times on the central clients vary. Due to the fluctuations, fair comparisons between model training times are not possible. Before moving

on to our main results, we report on the performance of the model proposed in [13] (see Appendix C.1). The model gains a profit of 54.52 TL during the 103 validation days. The authors' model uses shared value and policy networks and they take $\widetilde{N} = 10$ in (4.2.61). In our experience, using smaller values for $\widetilde{N}$ consistently improves both training and validation results. We set this to 1. Separating value and policy networks also yields better results. Lastly, we use networks with larger hidden layers compared to the the authors' choice of 64 nodes in each layer and the networks are always connected to a value and a policy readout with number of nodes 1 and 4, respectively.

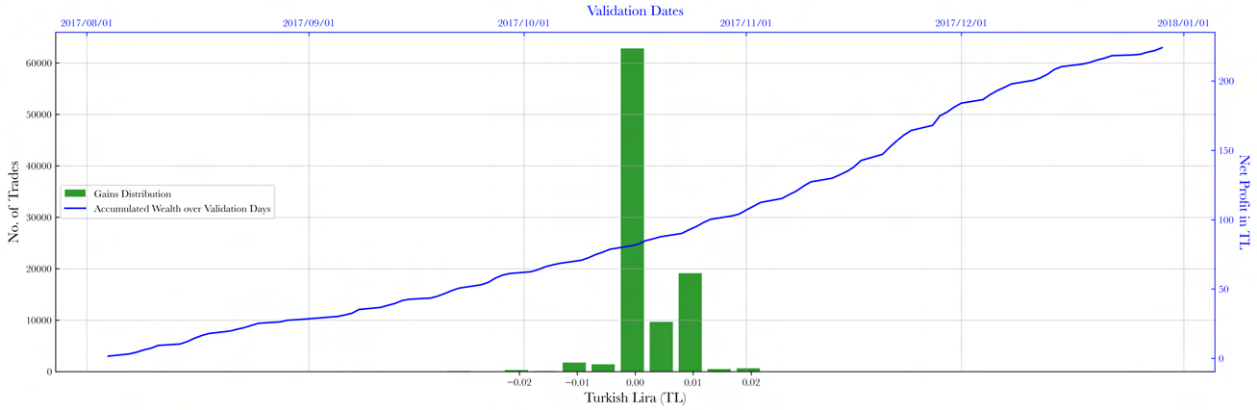
FNN		
Layer (type)	Output Shape	No. of Parameters
Linear-1	900	12,600
Tanh-2	900	0
Linear-3	64	57,664
Tanh-4	64	0
Linear-5	4 or 1	260 or 65
Total number of trainable parameters: 140,853		

Table 12: Network architecture of the FNN used for mid-price trading. The value and the policy networks are not shared. The last layer has output shape 4 for the policy representing the four possible actions *sell*, *stay*, *buy*, *stop-loss*. Biases are enabled for the linear layers.

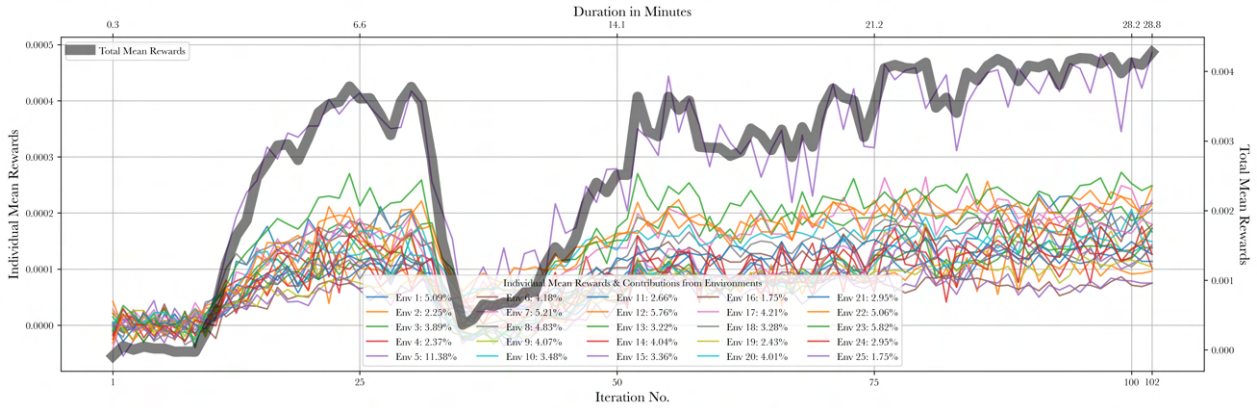
We start by presenting our FNN's training and validation results (see Figure 25). The network architecture of it is given in Table 12. The model gains a profit of 223.97 TL during the 103 validation days. The consistent increase of the net profit is shown in Figure 25a. The model state is taken from the 102^{nd} iteration, where the highest value for $\widetilde{R}$ is reached. In Figure 25b, a sudden drop in $\widetilde{R}$ around the 30^{th} is observed. After looking at the training logs, a high KL divergence about 0.1 is detected at this exact iteration. This value is way above our targeted KL divergence $\mathrm{KL}_{target} \approx 0.021$. The KL divergence decreases from 0.1 to about 0.00027 at the 36^{th} iteration and the training stability is restored for the rest of the training. Despite the clipping in (3.2.50), it is still possible to end up with a new policy which is too far from the old policy [94]. The PPO implementation that we use [103] deals with such cases by *early stopping* if the KL_{target} is exceeded, where the PPO optimization cycle is interrupted before reaching the number of PPO optimization steps n_{steps}.

As mentioned in Section 4.2, we are interested in using ELMs in order to reduce training times in terms of the 250 rollout iterations. In our experience, all models have similar ranges of training times and ELMs do not show any significant advantage in terms of training durations. On the other hand, they have better validation performance compared to FNNs. This is shown in Figure 26a. The parameters of Linear-1 in Table 13 are sampled from the standard normal distribution $\mathcal{N}(0, 1)$. The results show that we accomplish a significantly better validation performance by randomizing the first layer of the FNN. The most of the trades of both models are break-evens. The returns 0.005 TL and 0.01 TL are the main sources of the profits, where the returns 0.01 TL are more common than the returns 0.005 TL for both models. The ELM gains more from both return levels and surpasses the FNN in terms of total validation rewards. In addition, the training convergence of ELM is stabler than FNN's (see Figure 26b).

After replacing the first layer of the FNN with a static reservoir and observing a perfor-



(a) Validation performance. The model gains a profit of 223.97 TL during the 103 validation days.

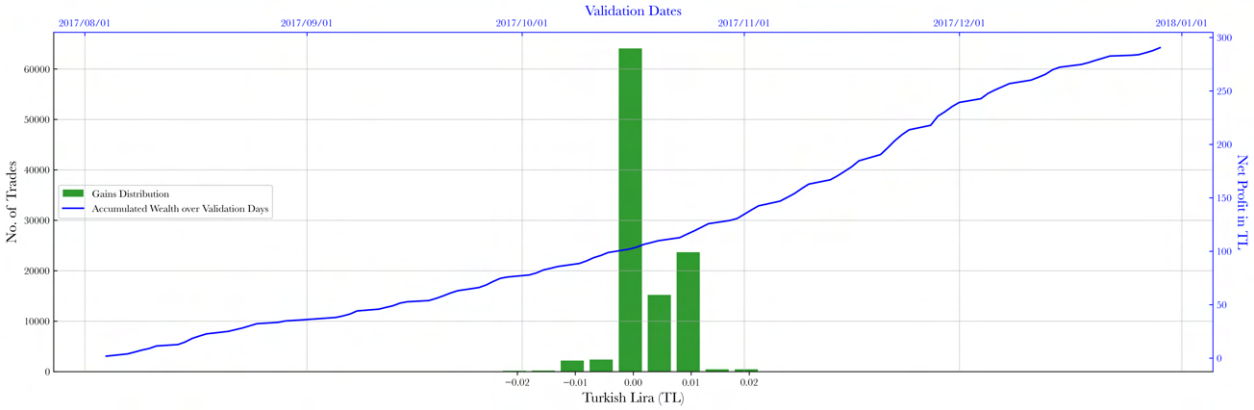


(b) Convergence of the mean rewards of individual environments as well as the total mean rewards $\widetilde{R}$ in (4.2.65). The contribution from each environment to $\widetilde{R}$ is given in percentages. The distribution of the mean rewards for each environment is shown in Figure 38.

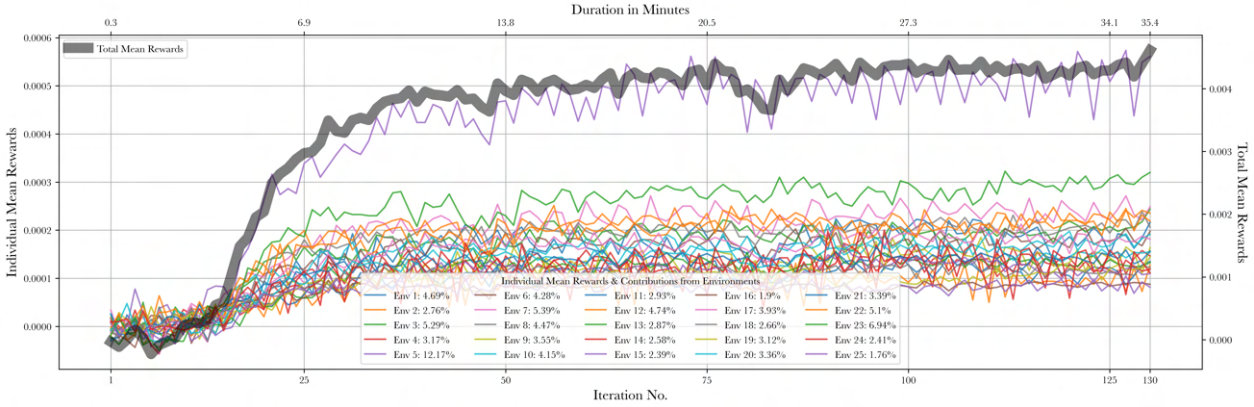
Fig. 25: Training and validation performance plots of the FNN used for mid-price trading with the architecture shown in Table 12.

ELM		
Layer (type)	Output Shape	No. of Parameters
Linear-1	900	(11,700)
Tanh-2	900	0
Linear-3	64	57,600
Tanh-4	64	0
Linear-5	4 or 1	260 or 65
Total number of trainable parameters: 115,525		

Table 13: Network architecture of the ELM used for mid-price trading. The value and the policy networks are not shared. Non-trainable layers have their number of parameters written inside round brackets. This makes up 23,400 non-trainable parameters for the ELM. Biases for Linear-1 and Linear-3 are disabled.



(a) Validation performance. The model gains a profit of 290.505 TL during the 103 validation days.



(b) Convergence of the mean rewards of individual environments as well as the total mean rewards $\widetilde{R}$ in (4.2.65). The contribution from each environment to $\widetilde{R}$ is given in percentages. The model state is taken from the 130^{th} iteration, where the highest value for $\widetilde{R}$ is reached. The distribution of the mean rewards for each environment is shown in Figure 39.

Fig. 26: Training and validation performance plots of the ELM used for mid-price trading with the architecture shown in Table 13.

ESN		
Layer (type)	Output Shape	No. of Parameters
ESN-1	256	29,184
Tanh-2	256	0
Linear-3	4 or 1	1,028 or 257
Total number of trainable parameters: 59,653		

(a) Network architecture of the ESN used for mid-price trading. The value and the policy networks are not shared.

HYPERPARAMETERS	
reservoir size	100
leak rate	0.7
spectral radius	0.4
reservoir activation	**id**
bias strength	1

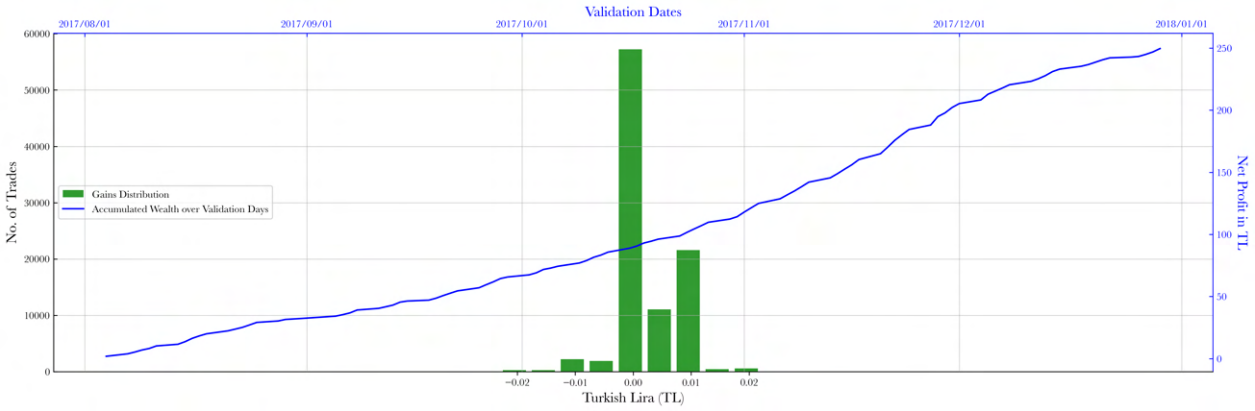
(b) Hyperparameters of the reservoir, which are used to calculate the echo states (4.2.63).

Table 14: ESN with output activation $\mathbf{f}^{\text{out}} = \mathbf{tanh}$. **On the Left:** ESN architecture. **On the Right:** Hyperparameter, which the echo states are generated by.

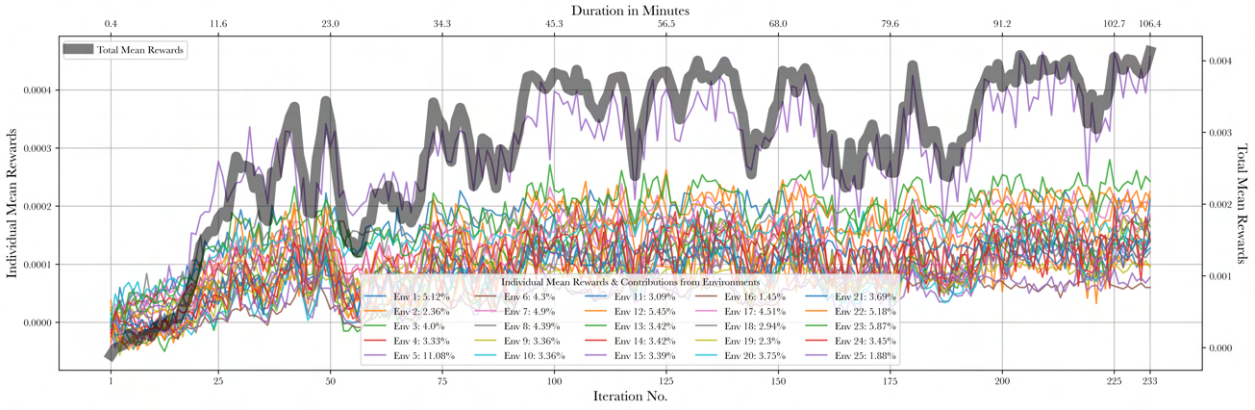
mance increase, we substitute the static reservoir with a dynamic one. The resulting network is an ESN with the architecture given in Table 14a. The ESN is realized by the offline calculation of the echo states (4.2.63) and feeding these to a linear layer with 256 nodes. The hyperparameters of the reservoir used to calculate the echo states is given in Table 14b. The input matrix of the reservoir $\mathbf{W}^{\text{in}}$ has entries sampled from $\mathcal{N}(0,1)$ and the reservoir matrix $\mathbf{W}$ has uniformly distributed entries over $[-0.5, 0.5]$.

The usual input dimension, given by (4.2.61) with $\widetilde{N} = 1$ is now increased by the reservoir size, yielding an input size of 113. Our experiments with higher values for $\widetilde{N}$ result in consistently worse validation performances, i.e. networks do not benefit from a constant feed of market information extending further into the past. To provide examples, the FNN and the ELM given in Table 12 & 13 have a total validation return of 77.9 TL and 242.27 TL as a result of trainings with $\widetilde{N} = 10$, respectively. For the ELM with $\widetilde{N} = 50$, it further reduces to 206.83 TL. This has the possible implication that for the majority of times, the past bars increase noise in the data feed and do not carry extra information for the agent to utilize at the current time step. Thus, mid-price trading problem with high-frequency LOB data seems to be more often Markovian than path dependent. Recalling the discussion on the MC of ESNs given in Section 2.2.1, we choose the reservoir size, spectral radius and the leak rate given in Table 14b accordingly. A larger reservoir leads to a larger MC and a large spectral radius has the effect of slower forgetting of the past inputs. As shown in Figure 5 & 6, a higher leak rate puts the emphasis on the more recent inputs and leads to a better recollection of the more recent past. Combining our knowledge on the MC dynamics with our findings on the effect of $\widetilde{N}$ on the validation performances of our models, a relatively small reservoir with a relatively small spectral radius and a relatively high leak rate is chosen. The resulting reservoir is used on the mid-price forecasting task shown in Figure 6. It corresponds to the reservoir of the ESN having the best performance for the delay set to 5. The training and validation results on mid-price trading with ESNs are shown in Figure 27. The validation result obtained with the ESN shows an improvement compared to the validation result of FNN, where the net return is increased from 223.97 TL to 249.61 TL. Yet it stays below the net return of 290.505 TL achieved by the ELM. The issue of large KL divergences persists, where a maximum KL divergence of approximately 0.14 is reached at the 152^{nd} iteration. Overall, the ESN achieves a higher net return than FNN, where its total number of trainable parameters is approximately 52% and 42% of the ELM's and the FNN's total number of trainable parameters, respectively.

The ESN architecture in Table 14a can be seen as an FNN receiving echo states as inputs. From this perspective, the introduction of echo states benefits the FNN setup. However, it is unclear whether this is the case due to the feed of past information by means of a reservoir or the randomized nature of a reservoir, i.e. the effect that is seen in the case of the ELM. This question could be investigated by feeding echo states to the ELM and observing the possible changes in the training and validation performance of the ELM. The new model is shown in Table 15. Since the resulting architecture is an ELM with echo states as inputs, we refer to it as "Echo State Machine" (ESM). The ESM has the the same precalculated echo states as the ones used for the ESN. The training and validation performances of the ESM is shown in Figure 28. No issue in terms of KL divergence is present with the exception of the 52^{nd} iteration, where a KL divergence of approximately 0.07 is reached. The model is able to recover from this large policy update and have a stable convergence for the rest of



(a) Validation performance. The model gains a profit of 249.61 TL during the 103 validation days.



(b) The model state is taken from the 233^{th} iteration. The distribution of the mean rewards for each environment is shown in Figure 40.

Fig. 27: Training and validation performance plots of the ESN used for mid-price trading with the architecture shown in Table 14.

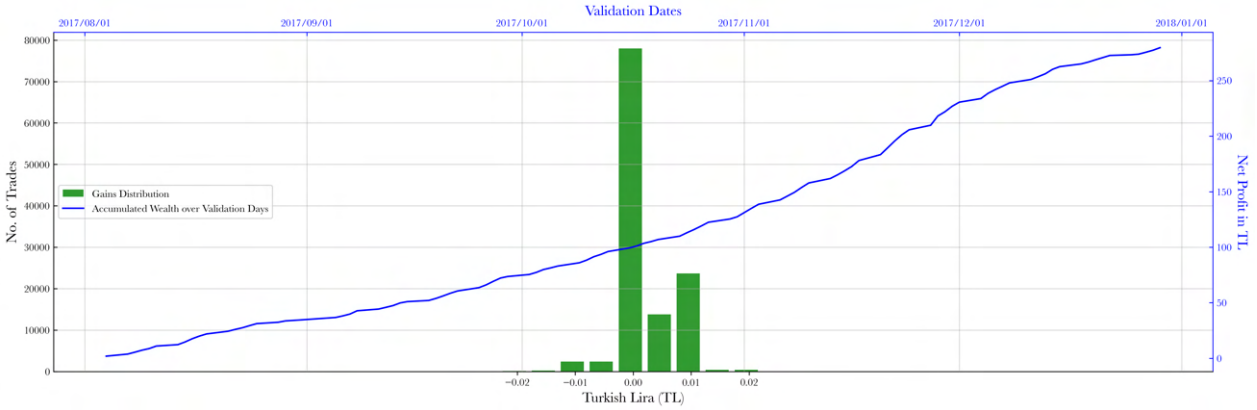
ESM		
Layer (type)	Output Shape	No. of Parameters
Linear-1	1000	(113,000)
Tanh-2	1000	0
Linear-3	64	64,000
Tanh-4	64	0
Linear-5	4 or 1	260 or 65
Total number of trainable parameters: 128,325		

Table 15: Network architecture of the ESM used for mid-price trading. The value and the policy networks are not shared. Non-trainable layers have their number of parameters written inside round brackets. This makes up 226,000 non-trainable parameters for the ESM. Biases for Linear-1 and Linear-3 are disabled.

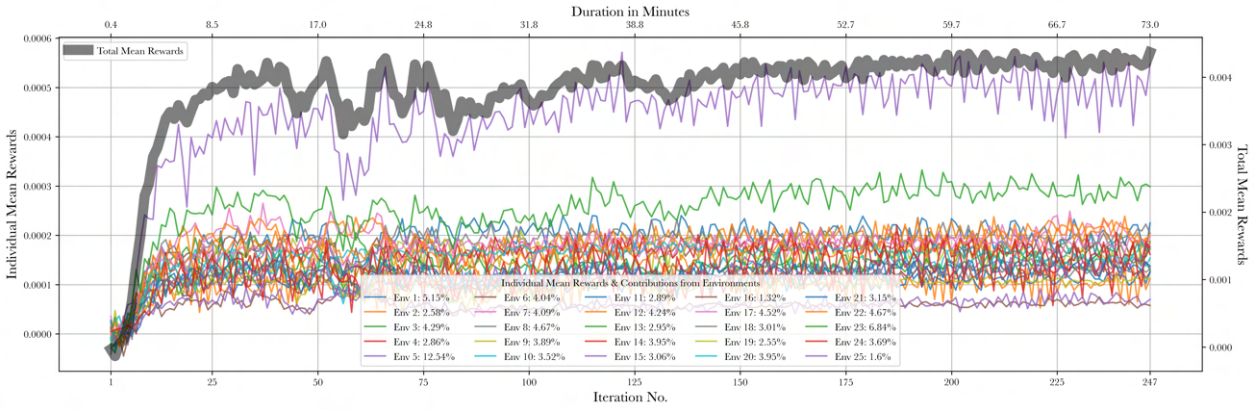
the training. The ESM gains a profit of 279.665 TL during the 103 validation days, which is close to but below the one obtained by the ELM. Hence, no improvement is observed by introducing echo states to the ELM. This indicates that the echo states in our mid-price trading with high-frequency data setup does not provide the expected utility of capturing path dependencies in the LOB and yielding better strategies.

The ESM executes more trades than the other models (see Figure 29a). This brings up the question of the trade frequencies of each strategy. Note that the term trade frequency is not used for real time intervals but tick time intervals. Comparing the other models to the ESM, the difference in number of trades is accumulated at the zero-return level, which constitutes more than half of the trades of all strategies. By looking at the duration distribution under the zero-return level in Figure 29a, it can be seen that these result from trades with higher frequencies. As one moves away from the break-evens, trades with lower frequencies are encountered.

In Figure 30, histograms in the center belong to four categories. The first quadrant shows the short positions with profits, the second quadrant longs with profits, the third quadrant longs with losses and the fourth quadrant shorts with losses. The two dimensional (2D) histograms at the right top corner show the trade frequencies over the spectrum of returns, again divided into the same four categories. Additionally, the frequency profile considering all the trades (including break-evens) is shown by the (dark magenta) histogram at the left side. The 2D histograms in Figure 30 show that the profits are mostly gained through high-frequency activity, where the most of the profitable trades have shorter position durations. We also see that the trades with profits 0.01 TL have longer durations than the trades with profits 0.005 TL. The ESM and the FNN have relatively lower frequency short position profits. All strategies extend to lower frequencies at the side of the short positions compared to the long positions. This behaviour is more balanced for the ELM and the ESM compared to the FNN. Another difference between the FNN and the other models, is that the FNN's total number of trades and average trade frequency are lower. In addition, it has the highest break-even percentage. The ESM's trade profile has the highest frequency on average, but this is due to the ESM's higher number of zero-return trades, which have high-frequency nature. Disregarding the break-evens, the pie charts in Figure 29b show that the long and short position proportions are fairly balanced across all strategies. Although the success rate of the FNN is similar to the other models in terms of both (non-zero) long and short positions, its number of profitable trades is lower than the others, resulting in a lower net return. Taking the break-evens into account, the FNN relies heavily (76.6 %) on long positions, but turns only 20% of these into profitable ones. It is a consistent behaviour among all models that imbalanced position preferences result in a lower profit probability due to the higher break-even rates. In terms of preferences, the ESM emphasizes short positions, but gains approximately the same amount of the profit that it gains from its long positions. On the other hand, the ELM emphasizes long positions, but gains approximately the same amount of the profit that it gains from its short positions. Despite having opposite inclinations in terms of the trade side they emphasize, the ELM gains more profits from its short positions than the amount that the ESM gains from its short positions. In summary, the ELM with a long position emphasis surpasses the ESM with a short position emphasis by being more efficient with its short positions (50% vs. 29%), than the ESM is with its long positions (29% vs. 36%).

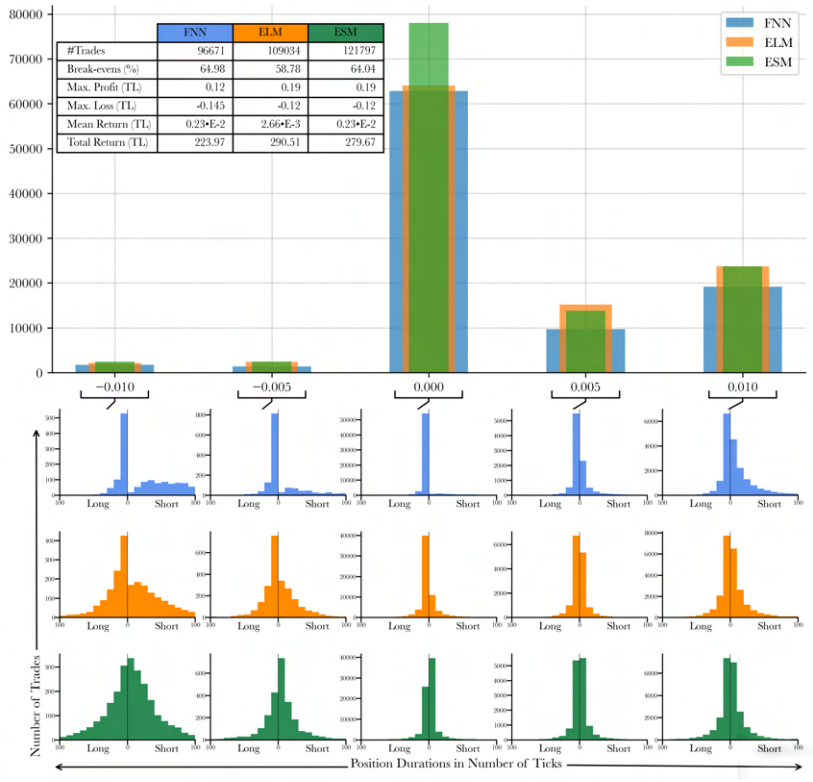


(a) Validation performance. The model gains a profit of 279.665 TL during the 103 validation days.

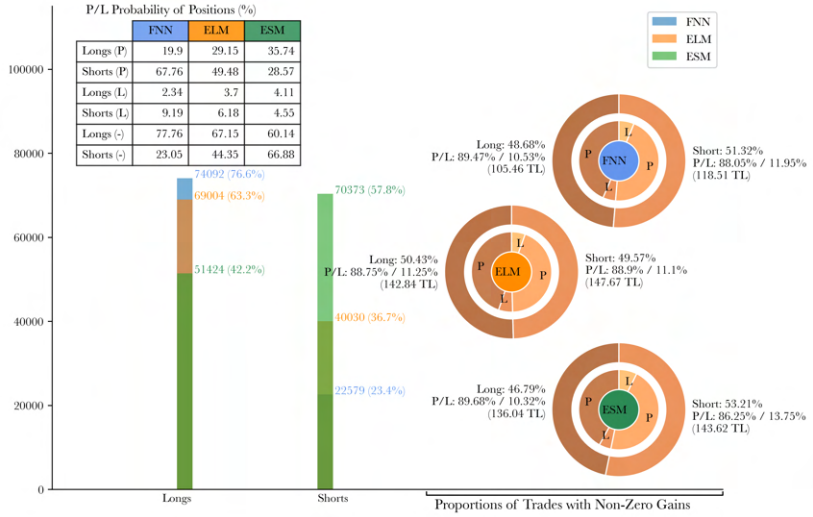


(b) The model state is taken from the 247^{th} iteration. The distribution of the mean rewards for each environment is shown in Figure 41.

Fig. 28: Training and validation performance plots of the ESM used for mid-price trading with the architecture shown in Table 15.

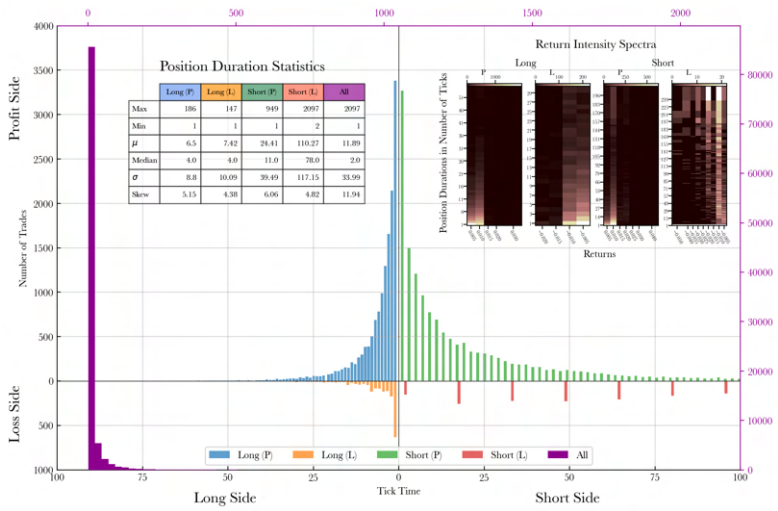


(a) Distribution of returns (TL) for each model. Each return level's frequency profile is shown for both long and short positions. Returns which are rare are not depicted.

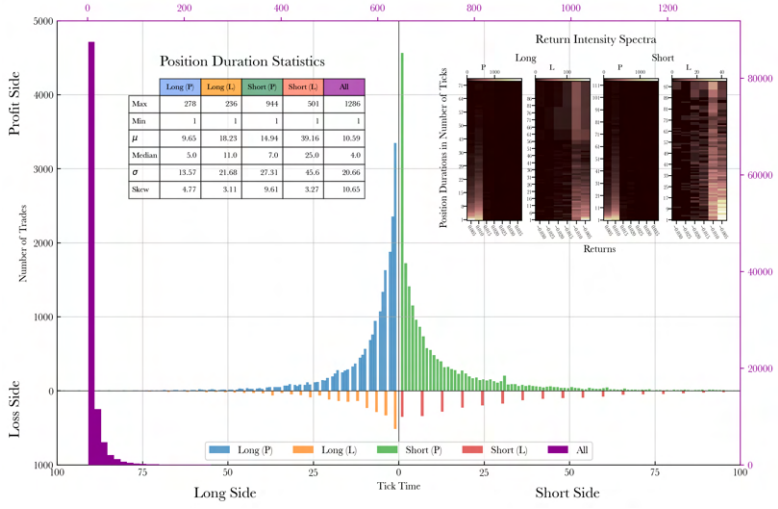


(b) Distributions of long and short positions of each model. The table shows profit (P), loss (L) and break-even (-) percentages of the strategies. The pie charts show the proportions and success rates of long and short positions of each model.

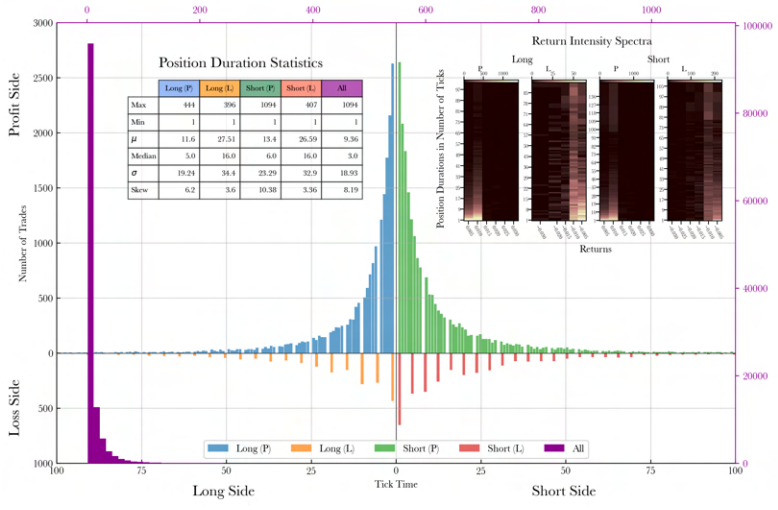
Fig. 29: Return profiles for mid-price trading.



(a) Trade frequency profile of FNN.



(b) Trade frequency profile of ELM.



(c) Trade frequency profile of ESM.

Fig. 30: Trade frequency profiles for mid-price trading.

4.2.3 Trading with Artificial Spreads

As mentioned in Section 4.2.1, we have not been able to replicate the results in [13] on our data set, where the authors factor in the bid-ask spread for a more realistic setup. Instead, we experiment with small artificial spreads and observe at which level of spread, our agents cannot gain any profits.

The results from the previous section show that all agents have approximately 60% or higher break-even rates, where they are able to frequently enter the corresponding positions and exit after several ticks without any costs. These trades would cause large losses for all models in a scenario with bid-ask spreads. Hence, it is of interest to investigate model performances after adding small perturbations to the system with the expectation of reduced numbers of break-even trades. This is done by introducing spreads artificially. It can be shown that for a constant spread $\mathfrak{s}$, (4.2.60) is given in terms of (4.2.62) by

$$\begin{aligned} R_{l/s} &= p^{best}_{b,\tau/t} - p^{best}_{a,t/\tau} \\ &= \left(m_{\tau/t} - \frac{\mathfrak{s}_{\tau/t}}{2}\right) - \left(m_{t/\tau} + \frac{\mathfrak{s}_{t/\tau}}{2}\right) \\ &= m_{\tau/t} - m_{t/\tau} - \frac{\mathfrak{s}_{\tau/t} + \mathfrak{s}_{t/\tau}}{2} \\ &= R^{mid}_{l/s} - \mathfrak{s}, \end{aligned} \tag{4.2.66}$$

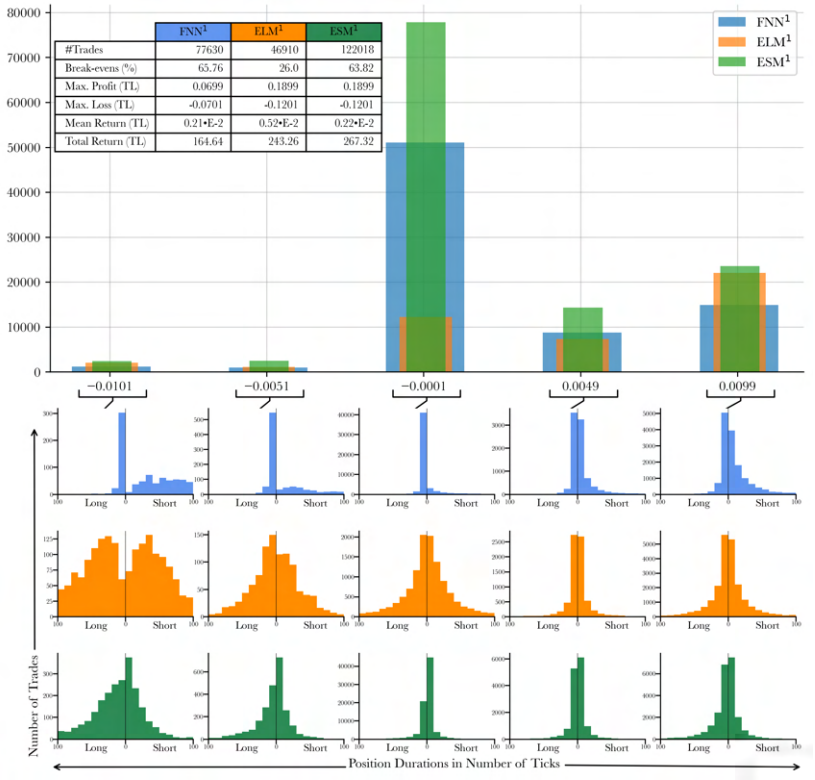
where $\mathfrak{s}_t$ denotes the spread at the t^{th} bar and in the last line the constant spread $\mathfrak{s}$ is inserted. Thus, we simply subtract a constant term from each reward obtained by the agent upon closing a position. Our starting point is $\mathfrak{s}^1 = 10^{-4}$ TL. The ESM has a total number of trades equal to 121,797 in the mid-price trading case. If the spread $\mathfrak{s}^1$ is subtracted from all of these, it results in a reduction of approximately 12.18 TL. This is approximately 4.35% of the ESM's total profit of 279.665 TL. The cost due to $\mathfrak{s}^1$ is small, yet the break-even trades now have negative reward signals, which is expected to cause the agents avoid some portion of these. Note that in this section the term "break-even" refers to $R^{mid}_{l/s} = 0$, where the return is given by (4.2.66). We continue to use the architecture in Table 15 and the echo states with the hyperparameters given in Table 14b for the ESM. We increase the size of the first layers of the FNN and the ELM from 900 to 1000. The results for $\mathfrak{s}^1$ are shown in Figure 31. We observe a significant drop in the total number of the break-even trades of the ELM with $\mathfrak{s}^1$ (ELM^1) compared to the case with $\mathfrak{s} = 0$. As expected, the number of the break-even trades are reduced considerably, except for the ESM with $\mathfrak{s}^1$ (ESM^1). A reduction in the number of the break-even trades for the FNN with $\mathfrak{s}^1$ (FNN^1) is also present and the total number of trades is slightly increased for ESM^1. The trade frequencies at individual return levels remain similar for FNN^1 and ESM^1. The frequency profiles are included in Appendix D, where Figure 42c shows that the profitable short and long positions of ESM^1 are shifted towards higher and lower frequencies, respectively. The total validation profits for FNN^1, ELM^1 and ESM^1 are approximately 164.64 TL, 243.26 TL and 267.32 TL, respectively. For ESM^1, this is in alignment with our previous calculation for the expected reduction by the spread $\mathfrak{s}^1$. On the other hand, the rest of the models experience a performance loss beyond this expectation. Although their long and short positions with $R^{mid}_{l/s} \neq 0$ have good success rates, the decrease in the number of entered positions causes the net profits to sink. ESM^1 is the model, which considers a higher number of trading opportunities in the presence of the

spread, while keeping a success rate, which is similar to its mid-price trading performance. The high number of trades of ESM^1 with $R_{l/s}^{mid} = 0$ does not cause serious losses, since the spread is small. However, its break-even trades, which are high in number, would cause significant losses in a scenario with real bid-ask spreads.

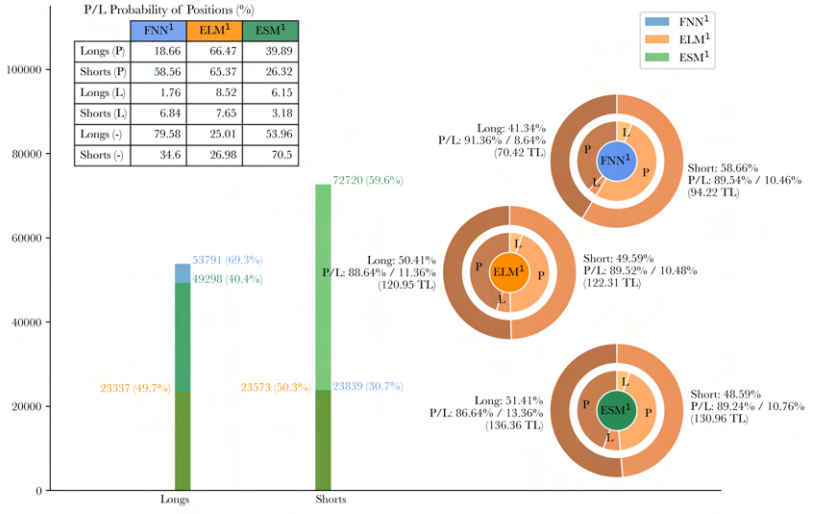
The next spread we employ is $\mathfrak{s}^2 = 3 \cdot 10^{-4}$ TL. Hence, the expected decrease in total returns is 36.54 TL for the ESM. In this setup, the ELM gains a total validation return of just 12.5475 TL and it is left out of the comparisons. The total returns of the FNN and the ESM are 110.0608 TL and 252.1088 TL, respectively. Thus, the total returns of the ESM is now better than $\mathfrak{s}^2$ subtracted from the old policy. A larger spread $\mathfrak{s}^3 = 5 \cdot 10^{-4}$ TL is used to investigate it further, whether the FNN and the ESM can still generate profits. Note that in this setting, subtracting the spread from the old policy yields a total return of approximately 218 TL. The FNN trained with $\mathfrak{s}^2$ (FNN^2), the ESM trained with $\mathfrak{s}^2$ (ESM^2) and the ESM trained with $\mathfrak{s}^3$ (ESM^3) have approximately the total validation profits 110.64 TL, 252.11 TL and 230.83 TL, respectively (see Figure 33). The FNN with $\mathfrak{s}^3$ yields a total validation return of 14.674 TL and it is not considered for a comparison with the rest of the models. We observe that the total number of trades decreases with increasing spread for the ESMs. The major contribution to this reduction comes from the decrease of the trades with $R_{l/s}^{mid} = 0$. The ESMs keep the numbers of trades at the rest of the return levels similar to the previous ones. Same statements apply to the FNN. In terms of the number of positions, a significant decrease in the number of long positions of the FNN is observed, while the major changes for the ESMs due to the spread are at the side of short positions. However, these numbers are not directly proportional to the success probabilities of the models in a given position, which can be recognized by comparing the rate at which the number of trades and the total validation returns vary. As expected, spreads enforce a type of efficiency in the sense that the models do not display the behavior of frequently entering positions which yield zero, or in this case close to zero returns ($R_{l/s} = -\mathfrak{s}$). Thus, they have higher profit rate per position, which is crucial in the scenario with real bid-ask spreads. Although the spread is increased, ESM^3 is able to display a performance comparable to ESM^2. While ESM^2 is still emphasizing short positions, ESM^3 has a more balanced policy between long and short positions.

All models have high success rates in terms of the trades with $R_{l/s}^{mid} \neq 0$. However, the ESM is the only architecture, which adapts to the increasing artificial spreads by changing its policy accordingly and achieving higher total returns. In terms of trading frequencies across all models and different spread levels, the range of profitable position durations is for the most part between 1 to 100 ticks, approximately. For the losses, the distributions are noisier, since our models have high success rates for trades with $R_{l/s}^{mid} \neq 0$ and the losses are rarer. The trade frequency profiles of the models shown in Figure 33 are included in Appendix D. Overall, the reason for a model having less total returns compared to another is not because of suffering heavier losses, but due to not utilizing as many trading opportunities as the other.

Our results show that the composite reservoir architecture ESM surpasses other static models in the spread cases $\mathfrak{s} \in \{1 \cdot 10^{-4}, 3 \cdot 10^{-4}, 5 \cdot 10^{-4}\}$, where it does not provide additional utility in the mid-price case compared to the ELM architecture. The main difference between the networks FNN, ELM and ESM is the fact that the ESM has additional information in its input space provided by the precalculated echo states. The results indicate that the inclusion of this information to the inputs helps with the training of the ESMs in the artificial spread cases, whereas the other models settle at subpar local optima. Recalling the discussions on



(a) Distribution of returns (TL) for each model. Note that break-even refers to $R_{l/s}^{mid} = 0$ in this section, where the return is given by (4.2.66). Each return level's frequency profile is shown for both long and short positions. Returns which are rare are not depicted.



(b) Distributions of long and short positions of each model. The table shows profit (P), loss (L) and break-even (-) percentages of the strategies. The pie charts show the proportions and success rates of long and short positions with mid-price return $R_{l/s}^{mid} = 0$.

Fig. 31: Return profiles for the artificial spread case $\mathfrak{s} = 10^{-4}$ TL.

the Markov property and path dependence, it is also possible that the introduction of the spreads to the system creates new settings, where the importance of the utilization of path dependent information is increased in terms of profitable trading strategies. This would be in agreement with the consistent worsening of the static architectures' performance as the spread is increased, whereas the ESM adapts to the new system with appropriate policies, keeping the total returns high. Further experimentation with larger spreads and ESMs has not been carried out and will be a part of future research.

As intended, the number of trades with $R_{l/s}^{mid} = 0$ are reduced by means of adding artificial spreads to the training in the form of constant transaction costs, where the models get small negative reward signals for executing such trades. This reduction is essential, since the high numbers of the break-even trades of the models in the mid-price case would lead to large losses in a scenario with real bid-ask spreads. In order to see if our training method with artificial spreads brings any utility, we collect all results obtained in our experiments in Figure 32, where the net profits of the models subject to spreads in the interval $[0, 0.0055]$ are displayed. We are looking for models, whose lines have higher offsets and slopes, where the offset gives the net profit in the absence of any spread and the slope characterizes a model's sensitivity to spreads. The slopes of the lines belonging to the models trained with $\mathfrak{s}^{i\geq 1}$ are higher than the slopes of the others, where a slope closer to zero characterizes a trading strategy with higher tolerance to increasing spreads. We see that the model, which can tolerate the largest spread (0.005 TL) by having a positive net return (13.4 TL) is ELM^1, although it is the fourth best model in the absence of any spread. We note that the average spread in our validation data set is approximately 0.0107 TL and the spread is for 93.4% of the bars equal to 0.01 TL. Thus, a positive net return is achieved by ELM^1 at a spread level, which approximately corresponds to the half of the real spread value in our validation data. Although the net return is small, we remind ourselves that the agent is allowed to operate on only one unit of the stock at a time.

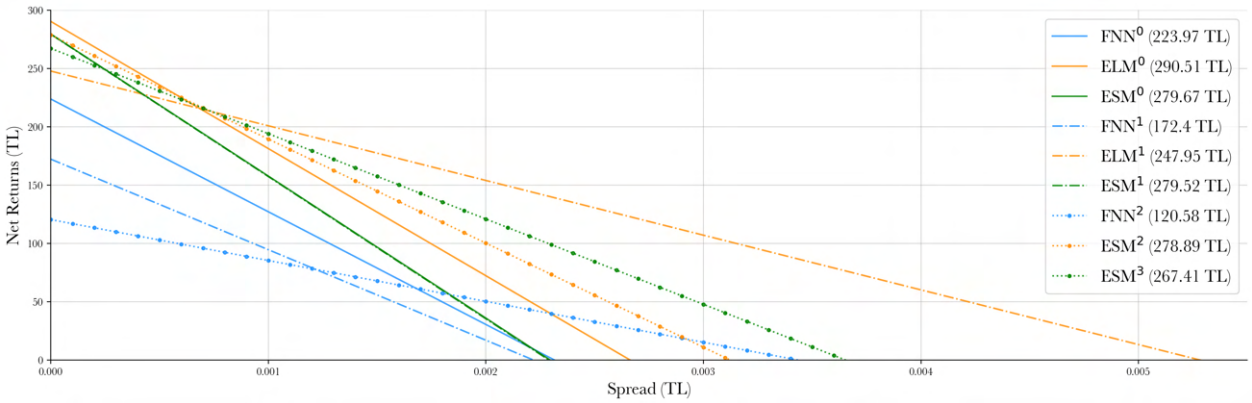
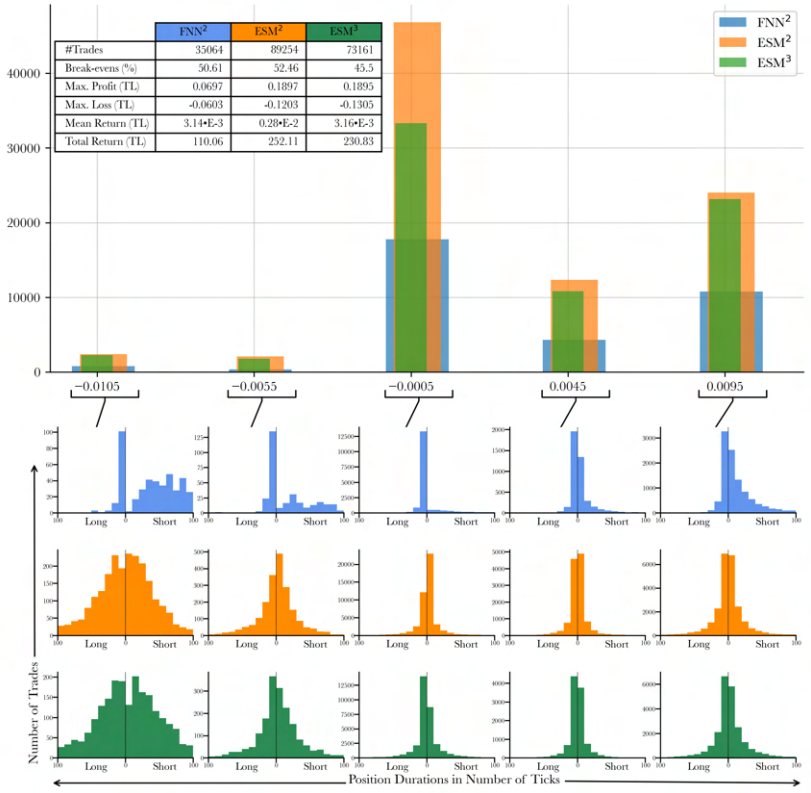
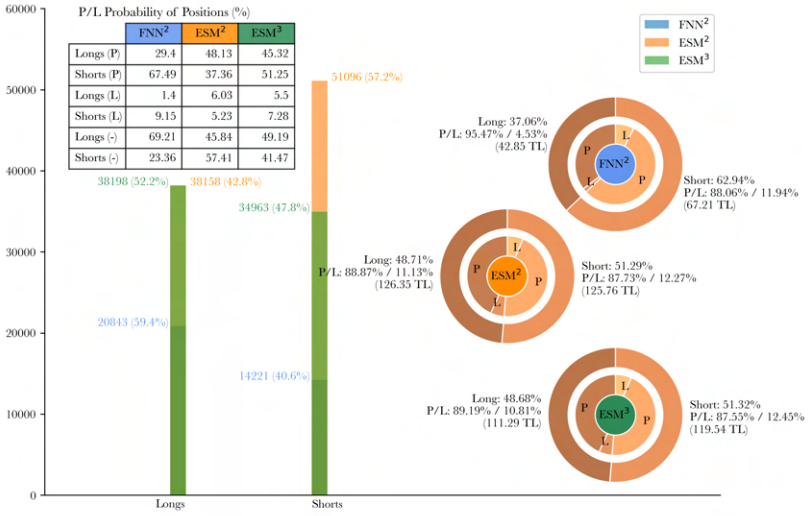


Fig. 32: Net returns of the models from mid-price trading ($\mathfrak{s} = 0$) and artificial spread ($\mathfrak{s}^{i\geq 1}$) cases are shown as functions of constant spread in the interval $[0, 0.0055]$. The superscript 0 denotes the mid-price trading case. The starting values (offsets) of the lines are displayed in the legend, where the spread is zero. The lines of ESM^0 and ESM^1 overlap, because they have similar number of total trades and their net profits are close when the spread is zero.



(a) Distribution of returns (TL) for each model. Note that break-even refers to $R^{mid}_{l/s} = 0$, where the return is given by (4.2.66). FNN^2 stands for the FNN trained with spread $\mathfrak{s}^2$. ESM^i stands for the ESM trained with spread $\mathfrak{s}^i$.



(b) Distributions of long and short positions of each model. The table shows profit (P), loss (L) and break-even (-) percentages of the strategies. The pie charts show the proportions and success rates of long and short positions of each model.

Fig. 33: Return profiles with artificial spreads $\mathfrak{s}^2$ and $\mathfrak{s}^3$.

4.3 Conclusion

We successfully train high-frequency trading RL agents trading on mid-prices of GARAN in the last 40% of the year 2017 using LOB data. Here, we utilize the randomized technologies ELMs and ESNs, where the numbers of learned parameters are significantly smaller than their fully trained counterparts, such as FNNs and RNNs with LSTMs. Although considerable differences in the training times of FNNs, ELMs and ESNs are not detected, the randomized networks perform consistently better than FNNs. By experimenting with the number of LOB bars $\widetilde{N}$ in the inputs of the models in the mid.price trading case, we observe that increasing $\widetilde{N}$ results in worse validation performance. In alignment with this observation, the recurrent models do not show any advantage over ELMs and the HFT problem with our data set seems to be more often Markovian than path dependent. However, in the artificial spread cases, the echo states given as inputs to the ESM seem to facilitate the trainings, where the static architectures' trainings end up at subpar local optima.

Our mid-price trading agents gain positive net returns, where the highest return is obtained by an ELM. Recognizing that the high numbers of break-even trades of the mid-rice trading models would cause serious losses in a scenario with real bid-ask spreads, we suggest a training scheme with small artificial spreads, which results in a reduction in the number of the break-even trades. ELM[1] benefits from the new training scheme the most compared to the other models and achieves the highest tolerance to increasing spreads. It is able to gain positive returns in the presence of a constant spread of 0.005 TL, which is the half of the spread in our validation data set for 93.4% of the bars. Recalling that our agents are allowed to hold at most a single position at any given time, changing this mechanic in the LOB environment by allowing the agents to leverage their profitable trades might lead to larger profits. Here, a trade size determination mechanism could be employed, related to the confidence of the agents in their actions given by their policies' probability distribution $\pi_\theta(\cdot \mid s)$ at a given state $s \in \mathcal{S}$ in the state space of the LOB. For a realistic assessment of the model performances, such a mechanism would require the market impact costs to be taken into consideration, as opposed to the case presented in this study.

5 Conclusions and Future Research

In this study reservoir computers as powerful ML tools are presented by means of various experiments and comparisons to other ML models such as FNNs and RNNs with LSTMs in Section 2.3. The utility of ESNs in RL is shown in Section 3.3.2 & 3.3.3, where the usually cumbersome training of RNNs with large number of trainable parameters are substituted with the training of a linear readout consisting of relatively small number of parameters. A customized version of the RL algorithm PPO is designed and given in Algorithm 6, which enables the training of ESNs with PPO. This algorithm is successfully applied to the path dependent problem PO-Cartpole in Section 3.3.3, demonstrating the utility of echo states in RL. Further modifications to the customized algorithm could be made, where the learning of the value function given in the objective (3.2.52) as a convex term is achieved by means of linear regression with the reservoir readout, which is expected to speed up the training. Another improvement could be made by removing the gradients associated to the samples in a batch, which are connected to the initial transients of the reservoir states.

In Section 4, a simple HFT framework in the context of RL from [13] is used for training high-frequency trading agents on LOB environments using PPO, where randomized neural networks and echo states as input features are utilized. The agents successfully trade on the mid-prices of GARAN in 2017 gaining positive net returns. A different training scheme is suggested in Section 4.2.3, where small artificial spreads added to the reward signals lead to trading strategies, which are better suited to the scenario with real bid-ask spreads.

As mentioned in Section 4.3, trade size determination mechanisms can be employed, based on the probability distributions of the actions given by $\pi_\theta(\cdot \mid s)$ at a given state $s \in \mathcal{S}$ in the state space of the LOB, which characterize the confidences associated with the actions. A successful implementation of this approach could be followed by latency and market impact cost simulations. However, such a setup might require different trading frameworks than what is used in this study, since the main source of profits for the mid-price trading agents in our experiments is at a return level of 0.01 TL, which corresponds to the bid-ask spread in our data. A conclusion on the matter requires further experimenting with LOBs of different assets.

According to the efficient-market hypothesis, news which contain newly received or noteworthy information should have an impact on the stock market [85]. This has been demonstrated by various studies using econometric and ML models to forecast future behaviour of returns [67]. Studies show that RC is well suited to time series forecasting [41, 70, 17] and recently natural language processing [18, 105, 106, 111]. Considering these, a subject to study could be detecting keywords based on text classification and sentiment analysis from news and social media databases using reservoirs, which have impact on the stock prices. The natural language based financial forecasting literature is ubiquitous and successful approaches have already been proposed [92], which could be used as starting points. In addition, values and directions of mid-price and volatility using high-frequency data could be forecast, similar to Section 2.3.3. Then, these signals could be employed in frameworks such as the one presented in this study [13] or the triple barrier method described in [101] to achieve excess returns in backtesting. In a low frequency setting such as the triple barrier method, the introduction of directional and news based forecasts as additional information to the network is expected to be especially useful. Here, a combination of RL and supervised learning

approaches could be used to not only learn the labels of the barriers, but also to learn the optimal barriers themselves. Lastly, other state of the art reservoir architectures [57, 39, 124, 23] could be experimented with. In Section 2.2.4, the promising results obtained with QRCs are presented and the compatibility of RC and quantum computing is pointed out. QRCs could be utilized and compared to classical reservoirs in the above mentioned application areas, where today's quantum simulators can be used as workarounds in case control over noise or larger numbers of qubits are needed.

References

[1] *A Focus On: Neural Microcircuits*. https://oxfordmedicine.com/page/112/a-focus-on-neural-microcircuits, Last accessed on 2022-05-10.

[2] Frédéric Abergel et al. *Physics of society: Econophysics and sociophysics: Limit order books*. Jan. 2016, pp. 1–218. DOI: 10.1017/CBO9781316683040.

[3] *About Borsa Istanbul*. URL: https://www.borsaistanbul.com/en/sayfa/2096/about-us. (Last accessed on 2022-05-06).

[4] Adam Paszke. *Reinforcement Learning (DQN) Tutorial*. https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html, Last accessed on 2022-05-02. 2021.

[5] Takuya Akiba et al. "Optuna: A Next-generation Hyperparameter Optimization Framework". In: *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2019.

[6] Irene Aldridge. *High-frequency trading : a practical guide to algorithmic strategies and trading systems*. eng. 2nd ed. Wiley trading series. Hoboken, N.J: Wiley, 2013. ISBN: 1-118-41682-1.

[7] Frank Arute et al. "Quantum supremacy using a programmable superconducting processor". In: *Nature* 574.7779 (Oct. 2019), pp. 505–510. ISSN: 1476-4687. DOI: 10.1038/s41586-019-1666-5. URL: https://doi.org/10.1038/s41586-019-1666-5.

[8] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. "Neuronlike adaptive elements that can solve difficult learning control problems". In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-13.5 (1983), pp. 834–846. DOI: 10.1109/TSMC.1983.6313077.

[9] Nils Bertschinger, Thomas Natschläger, and Robert Legenstein. "At the Edge of Chaos: Real-time Computations and Self-Organized Criticality in Recurrent Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by L. Saul, Y. Weiss, and L. Bottou. Vol. 17. MIT Press, 2004. URL: https://proceedings.neurips.cc/paper/2004/file/f8da71e562ff44a2bc7edf3578c593da-Paper.pdf.

[10] Jacob Biamonte et al. "Quantum machine learning". In: *Nature* 549.7671 (Sept. 2017), pp. 195–202. ISSN: 1476-4687. DOI: 10.1038/nature23474. URL: https://doi.org/10.1038/nature23474.

[11] Joschka Boedecker et al. "Information processing in echo state networks at the edge of chaos". In: *Theory in Biosciences* 131.3 (Sept. 2012), pp. 205–213. ISSN: 1611-7530. DOI: 10.1007/s12064-011-0146-8. URL: https://doi.org/10.1007/s12064-011-0146-8.

[12] S. Boyd and L. Chua. "Fading memory and the problem of approximating nonlinear operators with Volterra series". In: *IEEE Transactions on Circuits and Systems* 32.11 (1985), pp. 1150–1161. DOI: 10.1109/TCS.1985.1085649.

[13] Antonio Briola et al. *Deep Reinforcement Learning for Active High Frequency Trading*. 2021. DOI: 10.48550/ARXIV.2101.07107. URL: https://arxiv.org/abs/2101.07107.

[14] Greg Brockman et al. "Openai gym". In: *arXiv preprint arXiv:1606.01540* (2016).

[15] Michael Buehner and Peter Young. "A tighter bound for the echo state property". In: *IEEE transactions on neural networks* 17.3 (2006), pp. 820–824.

[16] Lars Buesing, Benjamin Schrauwen, and Robert Legenstein. "Connectivity, Dynamics, and Memory in Reservoir Computing with Binary and Analog Neurons". In: *Neural computation* 22 (Dec. 2009), pp. 1272–311. DOI: 10.1162/neco.2009.01-09-947.

[17] J.B. Butcher et al. "Reservoir computing and extreme learning machines for non-linear time-series data analysis". In: *Neural Networks* 38 (2013), pp. 76–89. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2012.11.011. URL: https://www.sciencedirect.com/science/article/pii/S0893608012003085.

[18] Jérémie Cabessa et al. "Efficient Text Classification with Echo State Networks". In: *2021 International Joint Conference on Neural Networks (IJCNN)*. 2021, pp. 1–8. DOI: 10.1109/IJCNN52387.2021.9533958.

[19] Álvaro Cartea. *Algorithmic and high-frequency trading*. eng. Cambridge: Cambridge University Press, 2015. ISBN: 9781107091146.

[20] Jiayin Chen and Hendra I. Nurdin. "Learning nonlinear input–output maps with dissipative quantum systems". In: *Quantum Information Processing* 18.7 (May 2019), p. 198. ISSN: 1573-1332. DOI: 10.1007/s11128-019-2311-9. URL: https://doi.org/10.1007/s11128-019-2311-9.

[21] Jiayin Chen, Hendra I. Nurdin, and Naoki Yamamoto. "Temporal Information Processing on Noisy Quantum Computers". In: *Physical Review Applied* 14.2 (Aug. 2020). DOI: 10.1103/physrevapplied.14.024065. URL: https://doi.org/10.1103%2Fphysrevapplied.14.024065.

[22] L. Chua and D. Green. "A qualitative analysis of the behavior of dynamic nonlinear networks: Steady-state solutions of nonautonomous networks". In: *IEEE Transactions on Circuits and Systems* 23.9 (1976), pp. 530–550. DOI: 10.1109/TCS.1976.1084258.

[23] Enea Monzio Compagnoni et al. *Randomized Signature Layers for Signal Extraction in Time Series Data*. 2022. DOI: 10.48550/ARXIV.2201.00384. URL: https://arxiv.org/abs/2201.00384.

[24] Casper da Costa-Luis et al. "tqdm: A fast, Extensible Progress Bar for Python and CLI". In: (Apr. 2022). DOI: 10.5281/zenodo.6412640.

[25] Christa Cuchiero et al. *Discrete-time signatures and randomness in reservoir computing*. 2020. DOI: 10.48550/ARXIV.2010.14615. URL: https://arxiv.org/abs/2010.14615.

[26] G. Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals and Systems* 2.4 (Dec. 1989), pp. 303–314. ISSN: 1435-568X. DOI: 10.1007/BF02551274. URL: https://doi.org/10.1007/BF02551274.

[27] Samudra Dasgupta, Kathleen E. Hamilton, and Arnab Banerjee. *Designing a NISQ reservoir with maximal memory capacity for volatility forecasting*. 2020. DOI: 10.48550/ARXIV.2004.08240. URL: https://arxiv.org/abs/2004.08240.

[28] Sanjoy Dasgupta and Anupam Gupta. "An Elementary Proof of a Theorem of Johnson and Lindenstrauss". In: *Random Struct. Algorithms* 22 (Jan. 2003), pp. 60–65. DOI: 10.1002/rsa.10073.

[29] Ege Yilmaz. *A framework for Echo State Network applications.* https://github.com/Quantumyilmaz/EchoStateNetwork, Last accessed on 2022-04-15. 2021.

[30] Ege Yilmaz. *A python framework for Echo State Network applications.* https://echostatenetwork.readthedocs.io/en/latest/, Last accessed on 2022-04-15. 2021.

[31] Ege Yilmaz. *Deep Calibration of the Rough Heston Model. A modest approach.* https://colab.research.google.com/drive/1KaIy8Olr-Lo2o92zXMriQCSM4laIZfV4, Last accessed on 2022-04-26. 2020.

[32] Ege Yilmaz. *Liquidity Measures for Limit Order Book data and Machine Learning. Semester Project Page*. https://quantumyilmaz.github.io/SPHS20/, Last accessed on 2022-03-30. 2020.

[33] Ege Yilmaz. *Machine Learning Algorithms for High Frequency Trading. Master's Thesis Page*. https://quantumyilmaz.github.io/MTHS21/, Last accessed on 2022-04-15. 2021.

[34] Ege Yilmaz. *Master's Thesis Repository*. https://github.com/Quantumyilmaz/MTHS21, Last accessed on 2022-04-15. 2021.

[35] Ege Yilmaz. *Matplotlib based framework to create plots*. https://github.com/Quantumyilmaz/Plotter, Last accessed on 2022-04-22. 2020.

[36] Igor Farkaš and Peter Gergel'. "Maximizing memory capacity of echo state networks with orthogonalized reservoirs". In: *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2017, pp. 2437–2442.

[37] Keisuke Fujii and Kohei Nakajima. "Harnessing Disordered-Ensemble Quantum Dynamics for Machine Learning". In: *Phys. Rev. Applied* 8 (2 Aug. 2017), p. 024030. DOI: 10.1103/PhysRevApplied.8.024030. URL: https://link.aps.org/doi/10.1103/PhysRevApplied.8.024030.

[38] Keisuke Fujii and Kohei Nakajima. "Quantum Reservoir Computing: A Reservoir Approach Toward Quantum Machine Learning on Near-Term Quantum Devices". In: *Reservoir Computing: Theory, Physical Implementations, and Applications*. Ed. by Kohei Nakajima and Ingo Fischer. Singapore: Springer Singapore, 2021, pp. 423–450. ISBN: 978-981-13-1687-6. DOI: 10.1007/978-981-13-1687-6_18. URL: https://doi.org/10.1007/978-981-13-1687-6_18.

[39] Claudio Gallicchio, Alessio Micheli, and Luca Pedrelli. "Deep reservoir computing: A critical experimental analysis". In: *Neurocomputing* 268 (2017), pp. 87–99.

[40] Daniel J Gauthier. "Reservoir computing: Harnessing a universal dynamical system". In: *Phys. Rev. Lett* 120.024102 (2018), p. 2018.

[41] Daniel J. Gauthier et al. "Next generation reservoir computing". In: *Nature Communications* 12.1 (Sept. 2021). DOI: 10.1038/s41467-021-25801-2. URL: https://doi.org/10.1038%2Fs41467-021-25801-2.

[42] Lukas Gonon and Juan-Pablo Ortega. *Fading memory echo state networks are universal*. 2020. DOI: 10.48550/ARXIV.2010.12047. URL: https://arxiv.org/abs/2010.12047.

[43] Lyudmila Grigoryeva and Juan-Pablo Ortega. *Echo state networks are universal*. 2018. DOI: 10.48550/ARXIV.1806.00797. URL: https://arxiv.org/abs/1806.00797.

[44] Lyudmila Grigoryeva and Juan-Pablo Ortega. *Universal discrete-time reservoir computers with stochastic inputs and linear readouts using non-homogeneous state-affine systems*. 2017. DOI: 10.48550/ARXIV.1712.00754. URL: https://arxiv.org/abs/1712.00754.

[45] Ben Hambly and Terry Lyons. "Uniqueness for the signature of a path of bounded variation and the reduced path group". In: *Annals of Mathematics* 171.1 (Mar. 2010), pp. 109–167. DOI: 10.4007/annals.2010.171.109. URL: https://doi.org/10.4007%2Fannals.2010.171.109.

[46] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585 (2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2.

[47] *High-frequency trading : new realities for traders, markets and regulators*. eng. London: Risk books - Incisive Media, 2013. ISBN: 9781782720096.

[48] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[49] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208.

[50] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. "Extreme learning machine: a new learning scheme of feedforward neural networks". In: *2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No.04CH37541)*. Vol. 2. 2004, 985–990 vol.2. DOI: 10.1109/IJCNN.2004.1380068.

[51] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. "Extreme learning machine: Theory and applications". In: *Neurocomputing* 70.1 (2006). Neural Networks, pp. 489–501. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2005.12.126. URL: https://www.sciencedirect.com/science/article/pii/S0925231206000385.

[52] Jiangshuai Huang, Yongji Wang, and Jian Huang. "The Separation Property Enhancement of Liquid State Machine by Particle Swarm Optimization". In: *International Symposium on Neural Networks*. Springer. 2009, pp. 67–76.

[53] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[54] *IBM Quantum Experience*. https://www.ibm.com/quantum-computing/, Last accessed on 2022-05-08.

[55] Masako Isokawa. "Membrane time constant as a tool to assess cell degeneration". In: *Brain Research Protocols* 1.2 (1997), pp. 114–116. ISSN: 1385-299X. DOI: https://doi.org/10.1016/S1385-299X(96)00016-5. URL: https://www.sciencedirect.com/science/article/pii/S1385299X96000165.

[56] H. Jaeger. "Echo state network". In: *Scholarpedia* 2.9 (2007). revision #196567, p. 2330. DOI: 10.4249/scholarpedia.2330.

[57] Herbert Jaeger. "Discovering multiscale dynamical features with hierarchical echo state networks". In: (2007).

[58] Herbert Jaeger. "Generating exponentially many periodic attractors with linearly growing echo state networks". In: (Sept. 2006).

[59] Herbert Jaeger. *Long Short-Term Memory in Echo State Networks: Details of a Simulation Study*. 2012.

[60] Herbert Jaeger. "Short Term Memory in Echo State Networks". In: (Jan. 2002).

[61] Herbert Jaeger. "The" echo state" approach to analysing and training recurrent neural networks-with an erratum note'". In: *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report* 148 (Jan. 2001).

[62] Herbert Jaeger et al. "Optimization and applications of echo state networks with leaky-integrator neurons". In: *Neural Networks* 20.3 (2007). Echo State Networks and Liquid State Machines, pp. 335–352. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2007.04.016. URL: https://www.sciencedirect.com/science/article/pii/S089360800700041X.

[63] W. D. Kalfus et al. *Neuromorphic computing with a single qudit*. 2021. DOI: 10.48550/ARXIV.2101.11729. URL: https://arxiv.org/abs/2101.11729.

[64] Noriaki Kouda et al. "Qubit neural network and its learning efficiency". In: *Neural Computing & Applications* 14.2 (July 2005), pp. 114–121. ISSN: 1433-3058. DOI: 10.1007/s00521-004-0446-8. URL: https://doi.org/10.1007/s00521-004-0446-8.

[65] Swagat Kumar. *Balancing a CartPole System with Reinforcement Learning – A Tutorial*. 2020. DOI: 10.48550/ARXIV.2006.04938. URL: https://arxiv.org/abs/2006.04938.

[66] Aki Kutvonen, Keisuke Fujii, and Takahiro Sagawa. "Optimizing a quantum reservoir computer for time series prediction". In: *Scientific Reports* 10.1 (Sept. 2020), p. 14687. ISSN: 2045-2322. DOI: 10.1038/s41598-020-71673-9. URL: https://doi.org/10.1038/s41598-020-71673-9.

[67] Darya Lapitskaya, Hakan Eratalay, and Rajesh Sharma. "Predicting Stock Returns: ARMAX versus Machine Learning". In: *Advances in Econometrics, Operational Research, Data Science and Actuarial Studies: Techniques and Theories*. Ed. by M. Kenan Terzioğlu. Cham: Springer International Publishing, 2022, pp. 453–464. ISBN: 978-3-030-85254-2. DOI: 10.1007/978-3-030-85254-2_27. URL: https://doi.org/10.1007/978-3-030-85254-2_27.

[68] Robert Legenstein and Wolfgang Maass. "Edge of chaos and prediction of computational performance for neural circuit models". In: *Neural Networks* 20.3 (2007). Echo State Networks and Liquid State Machines, pp. 323–334. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2007.04.017. URL: https://www.sciencedirect.com/science/article/pii/S0893608007000433.

[69] Long-Ji Lin. *Reinforcement learning for robots using neural networks*. Carnegie Mellon University, 1992.

[70] Xiaowei Lin, Zehong Yang, and Yixu Song. "Short-term stock price prediction based on echo state networks". In: *Expert Systems with Applications* 36.3, Part 2 (2009), pp. 7313–7317. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2008.09.049. URL: https://www.sciencedirect.com/science/article/pii/S0957417408006519.

[71] Junxiu Liu et al. "An echo state network architecture based on quantum logic gate and its optimization". In: *Neurocomputing* 371 (2020), pp. 100–107. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2019.09.002. URL: https://www.sciencedirect.com/science/article/pii/S0925231219312627.

[72] Mantas Lukoševičius. "A Practical Guide to Applying Echo State Networks". In: *Neural Networks: Tricks of the Trade: Second Edition*. Ed. by Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 659–686. ISBN: 978-3-642-35289-8. DOI: 10.1007/978-3-642-35289-8_36. URL: https://doi.org/10.1007/978-3-642-35289-8_36.

[73] Mantas Lukoševičius and Herbert Jaeger. "Reservoir computing approaches to recurrent neural network training". In: *Computer Science Review* 3.3 (2009), pp. 127–149. ISSN: 1574-0137. DOI: https://doi.org/10.1016/j.cosrev.2009.03.005. URL: https://www.sciencedirect.com/science/article/pii/S1574013709000173.

[74] Lukoševičius, Mantas. *Simple Echo State Network implementations*. https://mantas.info/code/simple_esn/, Last accessed on 2022-04-27. 2016.

[75] Terry J Lyons, Michael Caruana, and Thierry Lévy. *Differential equations driven by rough paths*. Springer, 2007.

[76] Wolfgang Maass. "Liquid state machines: motivation, theory, and applications". In: *Computability in context: computation and logic in the real world* (2011), pp. 275–296.

[77] Wolfgang Maass. "Networks of spiking neurons: the third generation of neural network models". In: *Neural networks* 10.9 (1997), pp. 1659–1671.

[78] Wolfgang Maass, Thomas Natschläger, and Henry Markram. "Real-time computing without stable states: A new framework for neural computation based on perturbations". In: *Neural computation* 14.11 (2002), pp. 2531–2560.

[79] David JC MacKay, David JC Mac Kay, et al. *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

[80] *Market Making*. URL: https://www.borsaistanbul.com/en/sayfa/3585/market-making. (Last accessed on 2022-05-06).

[81] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[82] Michael B. Matthews. "Approximating nonlinear fading-memory operators using neural network models". In: *Circuits, Systems and Signal Processing* 12.2 (June 1993), pp. 279–307. ISSN: 1531-5878. DOI: 10.1007/BF01189878. URL: https://doi.org/10.1007/BF01189878.

[83] Wes McKinney et al. "Data structures for statistical computing in python". In: *Proceedings of the 9th Python in Science Conference*. Vol. 445. Austin, TX. 2010, pp. 51–56.

[84] Alexandre Ménard et al. "A game plan for quantum computing". In: (2020). URL: https://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/a-game-plan-for-quantum-computing.

[85] Simone Merello et al. "Investigating Timing and Impact of News on the Stock Market". In: *2018 IEEE International Conference on Data Mining Workshops (ICDMW)*. 2018, pp. 1348–1354. DOI: 10.1109/ICDMW.2018.00191.

[86] Donald Michie and Roger A Chambers. "BOXES: An experiment in adaptive control". In: *Machine intelligence* 2.2 (1968), pp. 137–152.

[87] Volodymyr Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning". In: (2016). DOI: 10.48550/ARXIV.1602.01783. URL: https://arxiv.org/abs/1602.01783.

[88] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 1476-4687. DOI: 10.1038/nature14236. URL: https://doi.org/10.1038/nature14236.

[89] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. DOI: 10.48550/ARXIV.1312.5602. URL: https://arxiv.org/abs/1312.5602.

[90] Steven D. Morad et al. *Graph Convolutional Memory using Topological Priors*. 2021. DOI: 10.48550/ARXIV.2106.14117. URL: https://arxiv.org/abs/2106.14117.

[91] Pere Mujal et al. "Opportunities in Quantum Reservoir Computing and Extreme Learning Machines". In: *Advanced Quantum Technologies* 4.8 (2021), p. 2100027. DOI: https://doi.org/10.1002/qute.202100027. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/qute.202100027. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/qute.202100027.

[92] Thien Hai Nguyen, Kiyoaki Shirai, and Julien Velcin. "Sentiment analysis on social media for stock movement prediction". In: *Expert Systems with Applications* 42.24 (2015), pp. 9603–9611. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2015.07.052. URL: https://www.sciencedirect.com/science/article/pii/S0957417415005126.

[93] Johannes Nokkala et al. "Gaussian states of continuous-variable quantum systems provide universal and versatile reservoir computing". In: *Communications Physics* 4.1 (Mar. 2021), p. 53. ISSN: 2399-3650. DOI: 10.1038/s42005-021-00556-w. URL: https://doi.org/10.1038/s42005-021-00556-w.

[94] OpenAI. *Key Equations*. URL: https://spinningup.openai.com/en/latest/algorithms/ppo.html#key-equations. (Last accessed on 2022-05-07).

[95] Recep Özalp et al. "A Review of Deep Reinforcement Learning Algorithms and Comparative Results on Inverted Pendulum System". In: *Machine Learning Paradigms: Advances in Deep Learning-based Technological Applications*. Ed. by George A. Tsihrintzis and Lakhmi C. Jain. Cham: Springer International Publishing, 2020, pp. 237–256. ISBN: 978-3-030-49724-8. DOI: 10.1007/978-3-030-49724-8_10. URL: https://doi.org/10.1007/978-3-030-49724-8_10.

[96] António R.C. Paiva, Il Park, and José C. Príncipe. "Chapter 8 - Inner Products for Representation and Learning in the Spike Train Domain". In: *Statistical Signal Processing for Neuroscience and Neurotechnology*. Ed. by Karim G. Oweiss. Oxford: Academic Press, 2010, pp. 265–309. ISBN: 978-0-12-375027-3. DOI: https://doi.org/10.1016/B978-0-12-375027-3.00008-9. URL: https://www.sciencedirect.com/science/article/pii/B9780123750273000089.

[97] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[98] Fabian Pedregosa et al. "Scikit-learn: Machine learning in Python". In: *Journal of machine learning research* 12.Oct (2011), pp. 2825–2830.

[99] Alexandre Piché et al. *Beyond Target Networks: Improving Deep Q-learning with Functional Regularization*. 2021. DOI: 10.48550/ARXIV.2106.02613. URL: https://arxiv.org/abs/2106.02613.

[100] Jason A. Platt et al. *Forecasting Using Reservoir Computing: The Role of Generalized Synchronization*. 2021. DOI: 10.48550/ARXIV.2102.08930. URL: https://arxiv.org/abs/2102.08930.

[101] Marcos de Prado. *Advances in Financial Machine Learning*. eng. 1st edition. Wiley, 2018. ISBN: 1-119-48211-9.

[102] John Preskill. "Quantum Computing in the NISQ era and beyond". In: *Quantum* 2 (Aug. 2018), p. 79. DOI: 10.22331/q-2018-08-06-79. URL: https://doi.org/10.22331%2Fq-2018-08-06-79.

[103] Antonin Raffin et al. "Stable-Baselines3: Reliable Reinforcement Learning Implementations". In: *Journal of Machine Learning Research* 22.268 (2021), pp. 1–8. URL: http://jmlr.org/papers/v22/20-1364.html.

[104] Haşim Sak, Andrew Senior, and Françoise Beaufays. *Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition*. 2014. DOI: 10.48550/ARXIV.1402.1128. URL: https://arxiv.org/abs/1402.1128.

[105] Nils Schaetti. "Author Verification in Stream of Text with Echo State Network-based Recurrent Neural Models". In: *SwissText*. 2019.

[106] Nils Schaetti. "Behaviors of Reservoir Computing Models for Textual Documents Classification". In: *2019 International Joint Conference on Neural Networks (IJCNN)*. 2019, pp. 1–7. DOI: 10.1109/IJCNN.2019.8852304.

[107] W.F. Schmidt, M.A. Kraaijveld, and R.P.W. Duin. "Feedforward neural networks with random weights". In: *Proceedings., 11th IAPR International Conference on Pattern Recognition. Vol.II. Conference B: Pattern Recognition Methodology and Systems*. 1992, pp. 1–4. DOI: 10.1109/ICPR.1992.201708.

[108] John Schulman et al. *High-Dimensional Continuous Control Using Generalized Advantage Estimation*. 2015. DOI: 10.48550/ARXIV.1506.02438. URL: https://arxiv.org/abs/1506.02438.

[109] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. DOI: 10.48550/ARXIV.1707.06347. URL: https://arxiv.org/abs/1707.06347.

[110] John Schulman et al. *Trust Region Policy Optimization*. 2015. DOI: 10.48550/ARXIV.1502.05477. URL: https://arxiv.org/abs/1502.05477.

[111] Sheng Shen et al. *Reservoir Transformers*. 2020. DOI: 10.48550/ARXIV.2012.15045. URL: https://arxiv.org/abs/2012.15045.

[112] David Silver. *Lectures on Reinforcement Learning*. URL: https://www.davidsilver.uk/teaching/. 2015.

[113] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[114] Richard S. Sutton. *Reinforcement learning : an introduction*. eng. Adaptive computation and machine learning series. Cambridge, Massachusetts: MIT Press, 1998. ISBN: 9780262257053.

[115] Yudai Suzuki et al. "Natural quantum reservoir computing for temporal information processing". In: *Scientific Reports* 12.1 (Jan. 2022), p. 1353. ISSN: 2045-2322. DOI: 10.1038/s41598-022-05061-w. URL: https://doi.org/10.1038/s41598-022-05061-w.

[116] *The Official Homepage on Origins of Extreme Learning Machines (ELM)*. https://elmorigin.wixsite.com/originofelm, Last accessed on 2022-05-09.

[117] Surya T Tokdar and Robert E Kass. "Importance sampling: a review". In: *Wiley Interdisciplinary Reviews: Computational Statistics* 2.1 (2010), pp. 54–60.

[118] Quoc Hoan Tran and Kohei Nakajima. *Higher-Order Quantum Reservoir Computing*. 2020. DOI: 10.48550/ARXIV.2006.08999. URL: https://arxiv.org/abs/2006.08999.

[119] Wim van Drongelen. "14 - Spike Train Analysis". In: *Signal Processing for Neuroscientists*. Ed. by Wim van Drongelen. Burlington: Academic Press, 2007, pp. 219–243. ISBN: 978-0-12-370867-0. DOI: https://doi.org/10.1016/B978-012370867-0/50014-0. URL: https://www.sciencedirect.com/science/article/pii/B9780123708670500140.

[120] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

[121] Ashish Vaswani et al. *Attention Is All You Need*. 2017. DOI: 10.48550/ARXIV.1706.03762. URL: https://arxiv.org/abs/1706.03762.

[122] Roman Vershynin. "Introduction to the non-asymptotic analysis of random matrices". In: (2010). DOI: 10.48550/ARXIV.1011.3027. URL: https://arxiv.org/abs/1011.3027.

[123] David Verstraeten. "Reservoir Computing: computation with dynamical systems". PhD thesis. Ghent University, 2009.

[124] Pietro Verzelli, Cesare Alippi, and Lorenzo Livi. "Echo State Networks with Self-Normalizing Activations on the Hyper-Sphere". In: *Scientific Reports* 9.1 (Sept. 2019), p. 13887. ISSN: 2045-2322. DOI: 10.1038/s41598-019-50158-4. URL: https://doi.org/10.1038/s41598-019-50158-4.

[125] Christopher J. C. H. Watkins and Peter Dayan. "Q-learning". In: *Machine Learning* 8.3 (May 1992), pp. 279–292. ISSN: 1573-0565. DOI: 10.1007/BF00992698. URL: https://doi.org/10.1007/BF00992698.

[126] *What is the Single Price Method?* URL: https://www.borsaistanbul.com/files/single_price.pdf. (Last accessed on 2022-05-06).

[127] Marco Wiering and Martijn van Otterlo. *Reinforcement Learning : State-Of-the-Art*. Berlin, Heidelberg : Springer, 2012.

[128] Ronald J Williams and Jing Peng. "Function optimization using connectionist reinforcement learning algorithms". In: *Connection Science* 3.3 (1991), pp. 241–268.

[129] Tyler Yep. *torchinfo*. Mar. 2020. URL: https://github.com/TylerYep/torchinfo.

[130] Izzet Yildiz, Herbert Jaeger, and Stefan Kiebel. "Re-visiting the echo state property". In: *Neural networks : the official journal of the International Neural Network Society* 35 (July 2012), pp. 1–9. DOI: 10.1016/j.neunet.2012.07.005.

[131] Chunyuan Zhang, Qi Song, and Zeng Meng. "Minibatch Recursive Least Squares Q-Learning". In: *Computational Intelligence and Neuroscience* 2021 (2021).

[132] Chunyuan Zhang et al. *Recursive Least Squares Policy Control with Echo State Network*. 2022. DOI: 10.48550/ARXIV.2201.04781. URL: https://arxiv.org/abs/2201.04781.

[133] Yong Zhang et al. "A Digital Liquid State Machine With Biologically Inspired Learning and Its Application to Speech Recognition". In: *IEEE Transactions on Neural Networks and Learning Systems* 26.11 (2015), pp. 2635–2649. DOI: 10.1109/TNNLS.2015.2388544.

A Definitions

Definition 3. ***Controlled Differential Equation*** *[75]*
Let V, W be Banach spaces and $\mathbf{L}(V,W)$ denote the set of continuous linear mappings from V to W. Consider the differential equation

$$dY_t = f(Y_t)d\gamma_t, \;\; Y_0 = \xi \in W, \tag{A.67}$$

where $f\colon W \to \mathbf{L}(V,W)$ is the vector field, $\gamma\colon J \to V$ the path of bounded variation on a compact interval $J = [S,T]$ and the dynamics of the system that is of interest, which is characterized by the solution or the response $Y\colon J \to W$, is dependent on the path γ. The path is called the control and the type of differential equation is called a controlled differential equation.

Definition 4. ***Signature*** *[45]*
The Signature of the path γ as in Definition 3 is the sequence of definite iterated integrals

$$S(\gamma) = \left(1 + \int_{S<u<T} d\gamma_u + \cdots + \int_{S<u_1<\cdots<u_k<T} d\gamma_{u_1} \otimes \cdots \otimes d\gamma_{u_k} + \cdots\right) \tag{A.68}$$

regarded as an element of an appropriate closure of the extended tensor algebra $T(V) = \bigoplus_{n=0}^{\infty} V^{\otimes n}$.

Definition 5. ***Importance Sampling*** *[117]*
Let $P(x)$ be a probability density for a random variable X and define

$$\mu_f \coloneqq \mathbb{E}_P[f(X)] = \int P(x)f(x)dx.$$

Then, for another probability density $Q(x)$ that satisfies $Q(x) > 0$ whenever $P(x)f(x) \neq 0$,

$$\mu_f = \mathbb{E}_Q[w(X)f(X)],$$

where $w(x) \coloneqq P(x)/Q(x)$. Therefore, a sample of independent draws $x^{(1)}, \ldots, x^{(m)}$ from $Q(x)$ can be used to estimate μ_f by

$$\hat{\mu}_f = \frac{1}{m}\sum_{j=1}^{m} w(x^{(i)})f(x^{(i)}).$$

This result is called Importance Sampling.

Definition 6. ***Kullback-Leibler Divergence*** *[79]*
The Kullback-Leibler divergence between two probability distributions P, Q defined on the same space $\mathcal{X}$ is given by

$$D_{\mathrm{KL}}(P||Q) = \sum_{x\in\mathcal{X}} P(x)\log\frac{P(x)}{Q(x)},$$

satisfying the Gibbs' inequality

$$D_{\mathrm{KL}}(P||Q) \geq 0$$

with equality only if $P = Q$. The Kullback-Leibler divergence is not symmetric.

B Memory Capacity

Unlike the common usage, the memory capacity illustration in 6, cannot be achieved by using the same sequence as both input and target. One needs to use the complete information provided by the LOB data to obtain good fits. The results obtained by using mid-price as both input and target is presented in Figure 34. Since the shapes of estimates are similar to the targets' shape, the correlations are still high and so are the MCs.

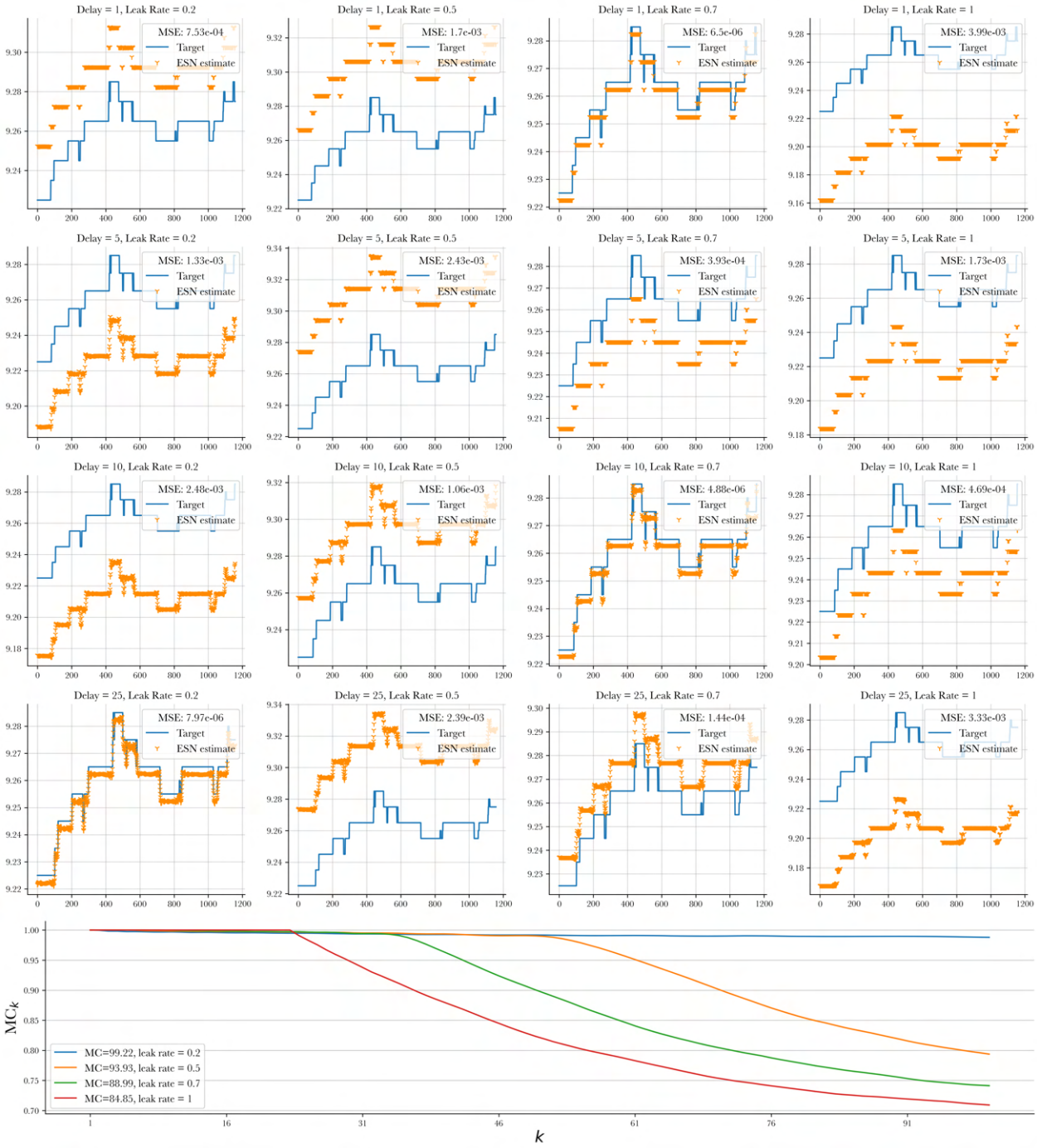


Fig. 34: MC of an ESN of size 100 with $\lambda_{max} = 0.4$ and leak rates 0.2, 0.5, 0.7 and 1. Initialization/training/validation split is 1000:2500:1158. The inputs are mid-prices derived from LOB data, whose details are given in Section 4.1. The reservoir matrix has entries sampled from a uniform distribution over $[-0.65, 0.65]$. Reservoir and output activations are set to **id**. The plots at the first four rows show the fitting accuracy of the networks over delays 1, 5, 10 and 25. Lastly, MC_k are shown for delays between 1 and 100. A maximum delay of 100 has been used for the calculation of MC, i.e. $\text{MC} = \sum_{k=1}^{100} \text{MC}_k$.

C Models for Mid-Price Trading

Here, we present our results obtained with models in the mid-price trading problem, which are not included in the main body.

C.1 FNN (Literature)

We present the results obtained by using PPO and the FNN architecture from [13]. The network architecture is given in Table 16. The hyperparameters of PPO are given in Table 11. The inputs consist of the last 10 LOB bars (including the current bar), the agent's current position and the MTM value at the current time step.

FNN		
Layer (type)	Output Shape	No. of Parameters
Linear-1	64	7,232
Tanh-2	64	0
Linear-3	64	4,160
Tanh-4	64	0
Linear-5	4 or 1	260 or 65
Total number of trainable parameters: 11,717		

Table 16: Network architecture of the model from [13]. The value and the policy networks are shared. The last layer has output shape 4 for the policy and 1 for the value network.

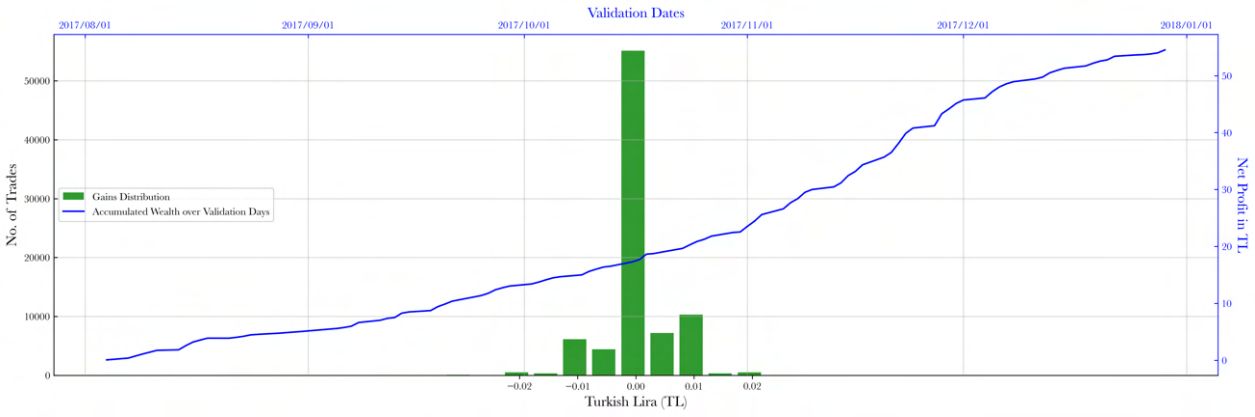


Fig. 35: FNN (Literature). Validation performance of the model from [13]. The architecture is shown in Table 16. A total profit of 54.52 TL in 103 days is gained.

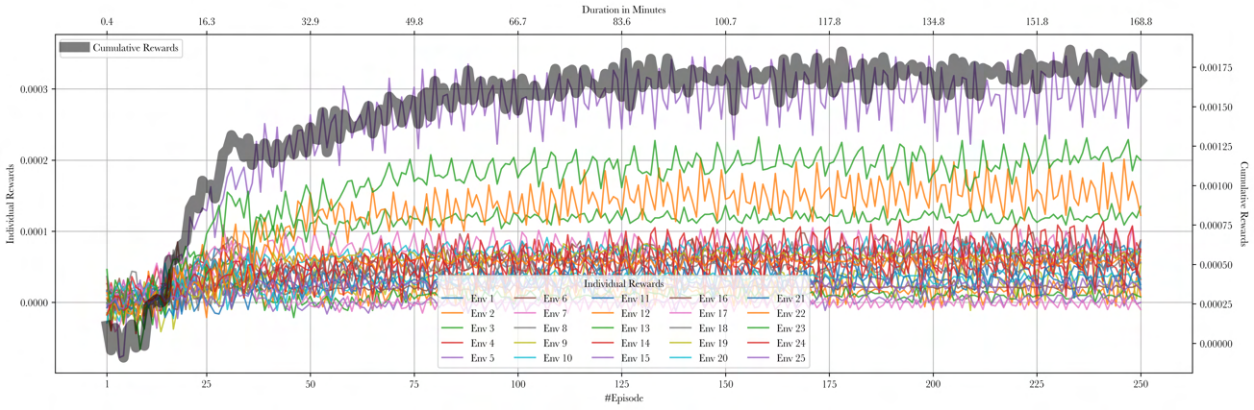


Fig. 36: FNN (Literature). Mean rewards of the PPO rollouts for each environment. The distributions of these can be seen in Figure 37. The total number of collected rollouts for each environment is 250.

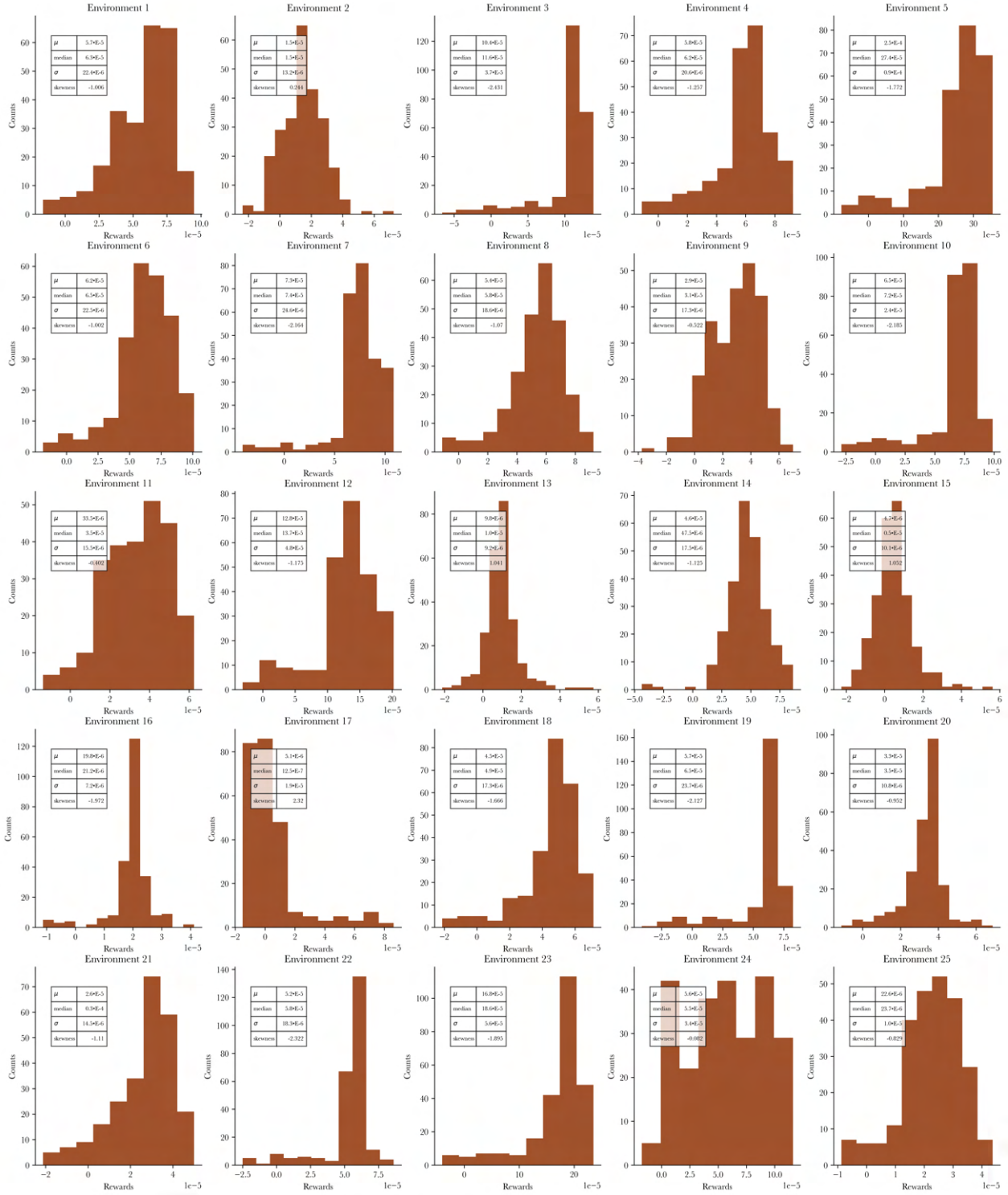


Fig. 37: FNN (Literature). The distributions of the mean rewards of the PPO rollouts. The total number of collected rollouts for each environment is 250.

C.2 FNN

The distributions of the mean rewards for each environment throughout the training is shown below. For the underlying trajectory of these, see Figure 25b.

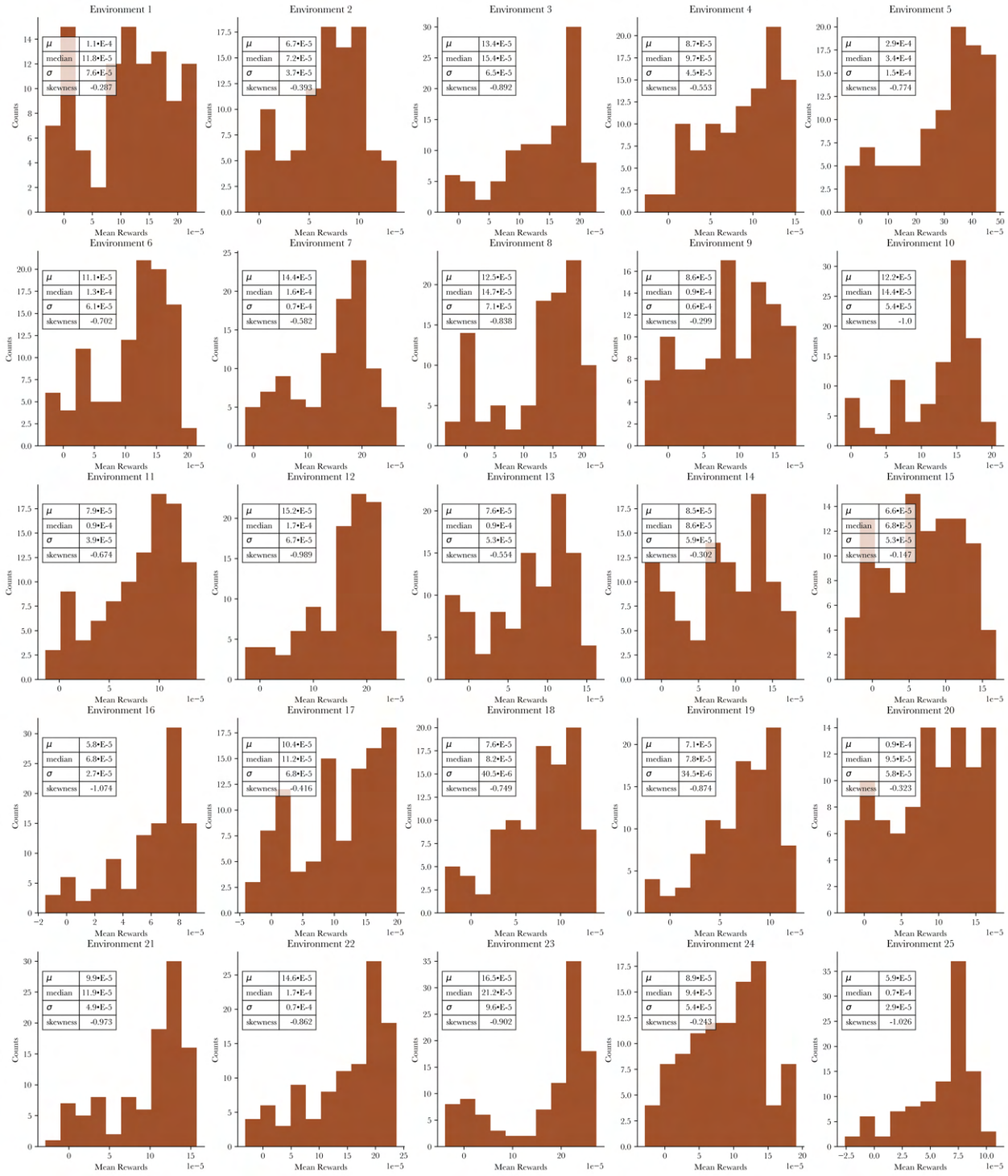


Fig. 38: FNN. The distributions of the mean rewards of the PPO rollouts. The total number of rollouts shown in the distributions is 102.

C.3 ELM

The distributions of the mean rewards for each environment throughout the training is shown below. For the underlying trajectory of these, see Figure 26b.

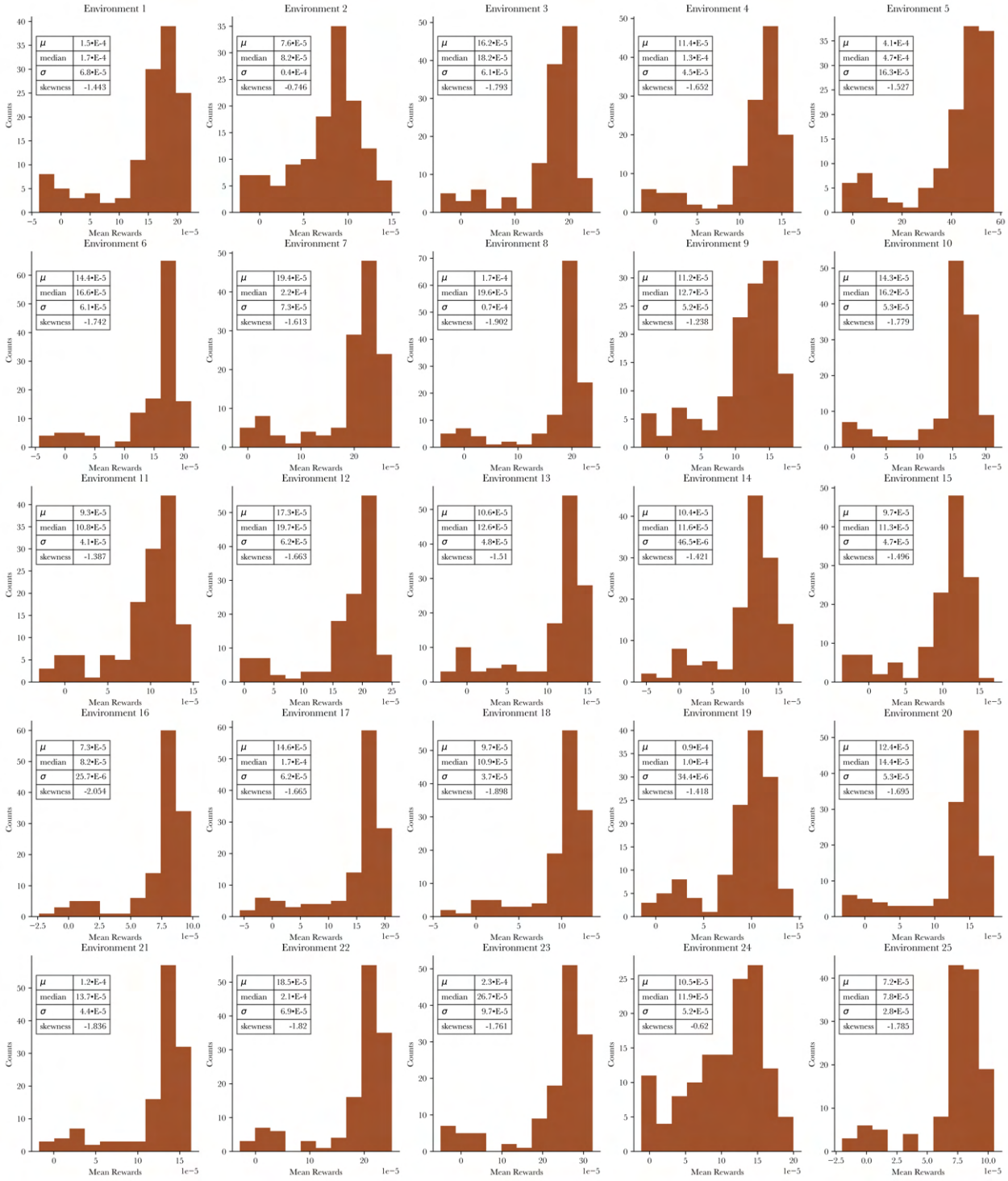


Fig. 39: ELM. The distributions of the mean rewards of the PPO rollouts. The total number of rollouts shown in the distributions is 130.

C.4 ESN

See Figure 27b for the underlying trajectory of the distributions.

Fig. 40: ESN. The total number of rollouts included in the distributions is 233.

C.5 ESM

See Figure 28b for the underlying trajectory of the distributions.

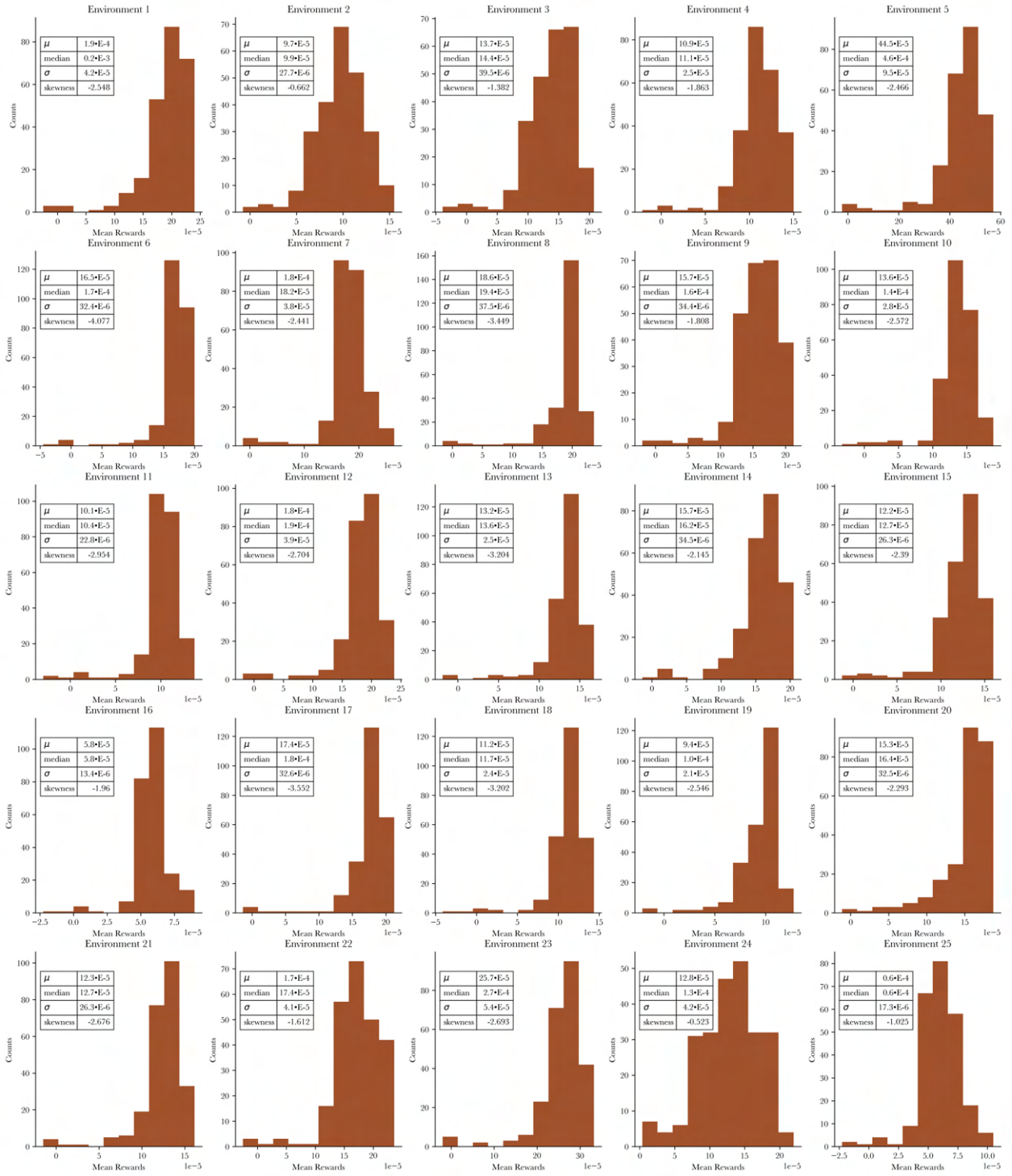
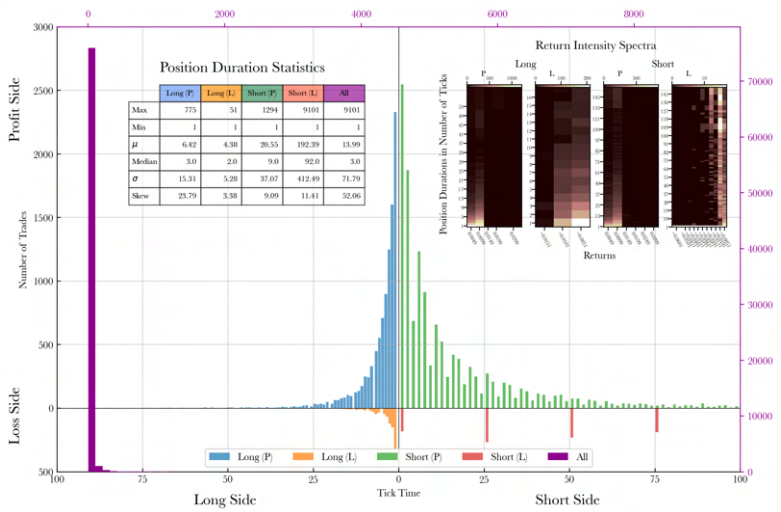


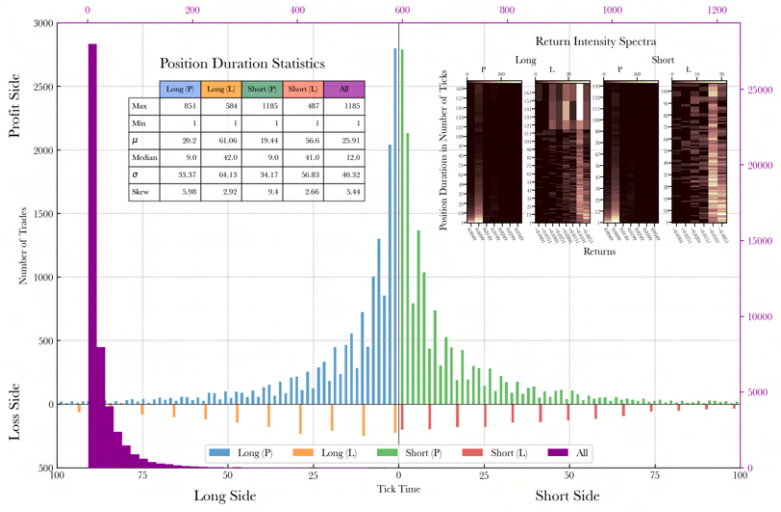
Fig. 41: ESM. The total number of rollouts included in the distributions is 247.

D Trade Frequency Profiles

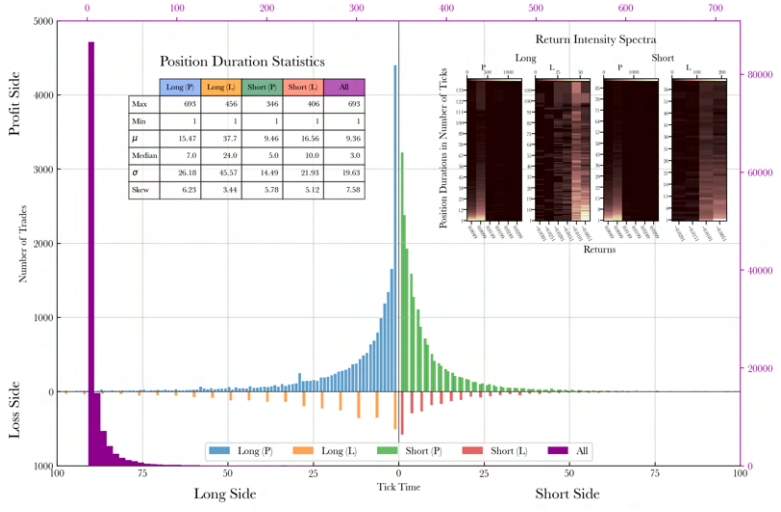
Here, we present the trade frequency profiles of the models used for trading with artificial spreads, which are not included in the main body.



(a) Trade frequency profile of FNN[1].

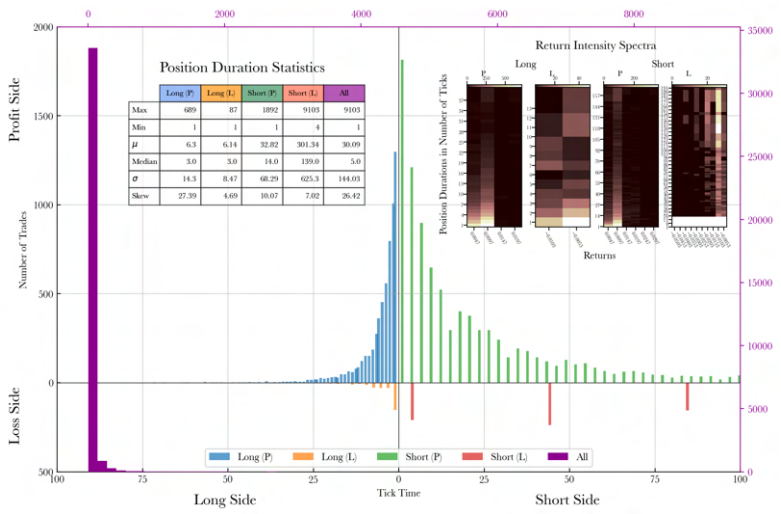


(b) Trade frequency profile of ELM[1].

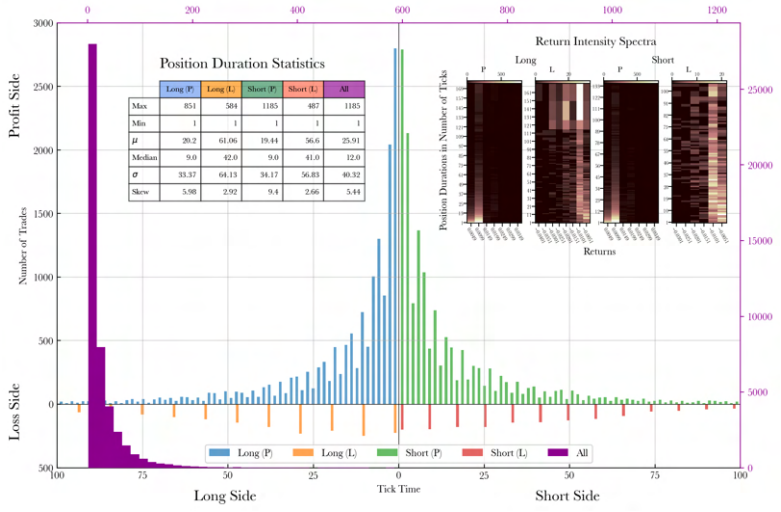


(c) Trade frequency profile of ESM[1].

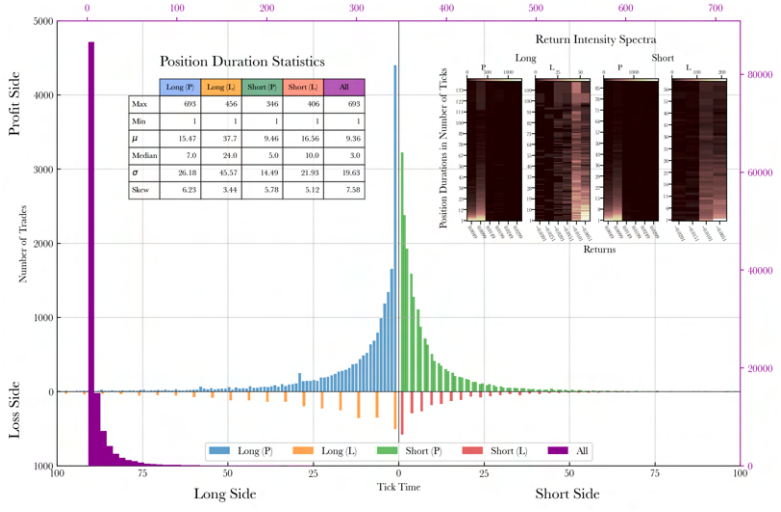
Fig. 42: Trade frequency profiles for $\mathfrak{s} = 10^{-4}$ TL.



(a) Trade frequency profile of FNN^2.



(b) Trade frequency profile of ESM^2.



(c) Trade frequency profile of ESM^3.

Fig. 43: Trade frequency profiles for spreads $\mathfrak{s}^2$ and $\mathfrak{s}^3$.

E Echo State Framework

Here, we include the documentation [30] of the Python class [29] written for the execution of ESN related tasks in the study.

CHAPTER

ONE

ESN

Echo State Network

class ESN(*resSize: Optional[int] = 400*, *xn: Optional[list[float]] = [0, 0.4, - 0.4]*, *pn: Optional[list[float]] = [0.9875, 0.00625, 0.00625]*, *random_state: Optional[float] = None*, *null_state_init: bool = True*, *custom_initState: Optional[np.ndarray] = None*, ***kwargs*)

Parameters

resSize Number of units (nodes) in the reservoir.

xn , pn User can provide custom random variable to control the connectivity of the reservoir. `xn` are the values and `pn` are the corresponding probabilities.

random_state Fix random state. If provided, `np.random.seed` and `torch.manual_seed` are called.

null_state_init If `True`, starts the reservoir from null state. If `False`, initializes randomly. Default is `True`.

custom_initState User can give custom initial reservoir state.

Keyword Arguments

verbose Give `False` to mute the messages.

f User can provide custom activation function of the reservoir. Default is identity. Functions in the pytorch or numpy libraries are accepted, including functions defined with `np.vectorize`. Some functions can also be given as strings. Accepted strings are:

- `'tanh'`
- `'sigmoid'`
- `'relu'`
- `'leaky_{slope}'`: e.g. `'leaky_0.5'` for LeakyReLU with slope equal to 0.5.
- `'softmax'`
- `'id'`: Identity.

f_out User can provide custom output activation. Default is identity.

leak_rate Leak parameter in Leaky Integrator ESN (LiESN). Default is 1.

leak_version Give `0` for Jaeger's recursion formula, give `1` for recursion formula in ESNRLS paper. Default is 0.

bias Set strength of bias in the input, reservoir and readout connections. Disabled by default.

W , Win , Wout , Wback User can provide custom reservoir, input, output, feedback matrices.

use_torch Use pytorch instead of numpy. Will use cuda if available.

device Give `'cpu'` if `use_torch` is `True` and CUDA is available on your device but you want to use CPU.

dtype Data type of reservoir. Default is `float64`.

Attributes

bias Strength of bias in the input layer. *float*

leak_rate Leak parameter in Leaky Integrator ESN (LiESN). *float*

leak_version Give `0` for Jaeger's recursion formula, give `1` for recursion formula in ESNRLS paper. *int*

f Activation function of the reservoir. *Callable*

f_out Output activation function of the reservoir. *Callable*

resSize Size of the reservoir. *int*

reservoir_layer Reservoir layer. *np.ndarray | torch.Tensor*

spectral_radius Spectral radius of the reservoir matrix. *float*

spectral_norm Spectral norm of the reservoir matrix. *float*

device Device, which the network computes on. *str*

dtype Data type of the network. *str*

W , Win , Wout , Wback Reservoir, input, output, feedback matrices. *np.ndarray | torch.Tensor*

1.1 ESN.scale_reservoir_weights

Scales the reservoir connection matrix to have certain spectral norm or radius. One can directly assign the desired value to `spectral_radius` or `spectral_norm` attributes instead of using this method.

scale_reservoir_weights(*desired_scaling: float*, *reference: str*) → None

Parameters

desired_scaling Scales the reservoir matrix to have the desired spectral norm or radius.

reference Give `'ev'` (eigenvalue) to choose spectral radius, `'sv'` (singular value) to choose spectral norm as reference.

1.2 ESN.reconnect_reservoir

Assigns new matrix to the reservoir with redefined connectivity.

reconnect_reservoir(*xn: list[float]*, *pn: list[float]*, ***kwargs*) → None

Parameters

xn , pn User can provide random variable to alter the connectivity of the reservoir. `xn` are the values and `pn` are the corresponding probabilities of the random variable.

verbose Set to False to mute the messages.

Keyword Arguments

verbose Give `False` to mute the messages.

1.3 ESN.excite

Time series data is used to update the reservoir nodes according to the formula:

$x_{n+1} = (1 - \alpha) \cdot x_n + \alpha \cdot f(\mathbf{W}_{in} \cdot u_{n+1} + \mathbf{W} \cdot x_n + \mathbf{W}_{back} \cdot y_n)$

, where x are the reservoir nodes, u inputs, y labels, α leaking rate, f activation function. This formula is for when both `u` and `y` are provided.

Note: The appropriate update formula is automatically recognized from the given parameters.

After initial transient, updated x are registered at each iteration and can be called via `reg_X` attribute (history of states used in regression).

excite(*u: Optional[np.ndarray] = None*, *y: Optional[np.ndarray] = None*, *initLen: Optional[int] = None*, *trainLen: Optional[int] = None*, *initTrainLen_ratio: Optional[float] = None*, *wobble: bool = False*, *wobbler: Optional[np.ndarray] = None*, ***kwargs*) → None

Parameters

u Input. Has shape [...,time].

y To be forecast. Has shape [...,time].

initLen Number of timesteps to be taken as initial transient tolarance. Will override initTrainLen_ratio. Will be set to an eighth of the training length if not provided.

trainLen Total number of training steps. Will be set to the length of input data.

initTrainLen_ratio Alternative to initLen, the user can provide the initialization period as ratio of the training length. The input `8` would mean that the initialization period will be an eighth of the training length.

wobble For enabling noise to be added to `y`.

wobbler User can provide custom noise. Default is `np.random.uniform(-1,1)/10000`.

Keyword Arguments

validation_mode Set to `True` to use `excite` in validation mode to prepare the reservoir for validation.

Note: To use this feature, `excite` must be called in training mode first.

1.4 ESN.fit

Does a regression to `y` using the registered reservoir updates, which can be obtained from attribute `reg_X` (history of states used in regression): $W_{out} = argmin_w ||y - Xw||_2^2 + \eta * ||w||_2^2$

fit(*y: np.ndarray*, *f_out_inverse: Optional[Callable] = None*, *regr: Optional[Callable] = None*, *reg_type: str = 'ridge'*, *ridge_param: float = 1e-08*, *solver: str = 'auto'*, *error_measure: str = 'mse'*, ***kwargs*) → np.ndarray

Parameters

y Data to fit.

f_out_inverse User can give custom output activation. Please give the INVERSE of the activation function. No activation is used by default.

regr User can give custom regressor. Overrides other settings if provided. If not provided, will be set to scikit-learn's regressor.

reg_type Regression type. Can be `'ridge'`, `'linear'` or `'pinv'` (Moore-Penrose pseudo inverse). Default is `'ridge'`.

ridge_param Regularization factor in ridge regression.

solver See scikit documentation.

error_measure Type of error to be displayed. Can be `'mse'` (Mean Squared Error) or `'mape'` (Mean Absolute Percentage Error).

Keyword Arguments

verbose For the error message.

reg_X Lets you overwrite `reg_X` attribute (history of states used in regression) with a custom one of your choice.

Tip: For online training purposes, i.e. you call `excite` up to a certain point in your data and do a forecast at that point and repeat it at later points in your data.

1.5 ESN.validate

Returns forecast.

validate(*u: Optional[np.ndarray] = None*, *y: Optional[np.ndarray] = None*, *valLen: Optional[int] = None*, ***kwargs*) → np.ndarray

Parameters

u Input series. Has shape [...,time].

y To be forecast. Has shape [...,time].

valLen Validation length.

Note: If `u` or `y` is provided it is not needed to be set. Mostly necessary for when neither `u` nor `y` is present.

Keyword Arguments

wobble For enabling random noise. Default is False.

wobbler User can provide custom noise. Disabled per default.

1.6 ESN.session

Executes a whole training/validation session by calling the methods `excite`, `train` and `validate`. Returns the forecasts.

session(*X_t: Optional[np.ndarray] = None*, *y_t: Optional[np.ndarray] = None*, *X_v: Optional[np.ndarray] = None*, *y_v: Optional[np.ndarray] = None*, *training_data: Optional[np.ndarray] = None*, *f_out_inverse: Optional[Callable] = None*, *initLen: Optional[int] = None*, *initTrainLen_ratio: Optional[float] = None*, *trainLen: Optional[int] = None*, *valLen: Optional[int] = None*, *wobble_train: bool = False*, *wobbler_train: Optional[np.ndarray] = None*, *null_state_init: bool = True*, *custom_initState: Optional[np.ndarray] = None*, *regr: Optional[Callable] = None*, *reg_type: str = 'ridge'*, *ridge_param: float = 1e-08*, *solver: str = 'auto'*, *error_measure: str = 'mse'*, ***kwargs*) → np.ndarray

Parameters

X_t Training inputs. Has shape [...,time].

y_t Training targets. Has shape [...,time].

X_v Validation inputs. Has shape [...,time].

y_v Validation targets. Has shape [...,time].

training_data Data to fit to in regression. It will be set to `y_t` automatically if it is not provided. Either way, `y_t` will be used when calling `excite`.

f_out_inverse Please give the INVERSE output activation function. No activation is used by default.

initLen No of timesteps to initialize the reservoir. Will override initTrainLen_ratio. Will be set to an eighth of the training length if not provided.

initTrainLen_ratio Alternative to initLen, the user can provide the initialization period as ratio of the training length. An input of 8 would mean that the initialization period will be an eighth of the training length.

trainLen Total no of training steps. Will be set to the length of input data, if not provided.

valLen Total no of validation steps. Will be set to the length of input data, if not provided.

wobble_train For enabling noise to be added to `y_t`.

wobbler_train User can provide custom noise. Default is `np.random.uniform(-1,1)/10000`.

null_state_init If `True`, starts the reservoir from null state. If `False`, initializes randomly. Default is `True`.

custom_initState User can give custom initial reservoir state.

regr User can give custom regressor. Overrides other settings if provided. If not provided, will be set to scikit-learn's regressor.

reg_type Regression type. Can be `'ridge'`, `'linear'` or `'pinv'` (Moore-Penrose pseudo inverse). Default is `'ridge'`.

ridge_param Regularization factor in ridge regression.

solver See scikit documentation.

error_measure Type of error to be displayed. Can be `'mse'` (Mean Squared Error) or `'mape'` (Mean Absolute Percentage Error).

Keyword Arguments

wobble_val For enabling noise to be added to `y_val` during validation. Default is False.

wobbler_val User can provide custom noise to be added to `y_val`. Disabled per default.

train_only Set to True to perform a training session only, i.e. no validation is done.

verbose Mute the training error messages.

1.7 ESN.update_reservoir_layer

Applies one-step update to the reservoir layer using: $x_{n+1} = (1-\alpha)\cdot x_n + \alpha \cdot f(\mathbf{W}_{in}\cdot u_{n+1} + \mathbf{W}\cdot x_n + \mathbf{W}_{back}\cdot y_n)$, where x are the reservoir nodes, u inputs, y labels, α leaking rate, f activation function. This formula is for when both u and y are provided.

Note: The appropriate update formula is automatically recognized from the given parameters.

---

update_reservoir_layer(*in_: Optional[np.ndarray | torch.Tensor] = None*, *out_: Optional[np.ndarray | torch.Tensor] = None*, *mode: Optional[str] = None*) → None

Parameters

in_ Input array.

out_ Output array.

mode Optional. Set to `'train'` if you are updating the reservoir layer for training purposes. Set to `'val'` if you are doing so for validation purposes. This will allow the reservoir object to name the training/validation modes, which can be accessed from `'training_type'` and `'val_type'` attributes.

1.8 ESN.update_reservoir_layers_serially

Warning: Resets reservoir layer. See *ESN.reset_reservoir_layer*.

When using the reservoir in `batch` or `ensemble` mode, the reservoir layer will be updated using $x^k = (1-\alpha)x^{k-1} + \alpha \cdot f(\mathbf{W}_{in} \cdot u^k + \mathbf{W} \cdot x^{k-1})$, where $1 \leq k \leq$ the batch size, u^k the k^{th} data point in the batch and x^0 is the initial reservoir layer before any updates.

x here is a matrix with shape:

(reservoir size,batch size) if in `batch` mode.

(number of reservoirs,reservoir size,batch size) if in `ensemble` mode.

Note: Reservoir can be set to `batch` or `ensemble` mode by using *ESN.set_reservoir_layer_mode* or one of the following:

- ESNX
- ESNS
- ESNN

update_reservoir_layers_serially(*in_: Optional[np.ndarray | torch.Tensor] = None*, *out_: Optional[np.ndarray | torch.Tensor] = None*, *mode: Optional[str] = None*, *init_size: int = 0*) → None

Parameters

in_ Input with shape

- (data point length,batch size + initialization length) (see `init_size`) if in `batch` mode.
- (number of reservoirs,data point length,batch size + initialization length) if in `ensemble` mode.

out_ **Not supported. WIP.**

leak_version Give `0` for Jaeger's recursion formula, give `1` for recursion formula in ESNRLS paper.

leak_rate Leak parameter in Leaky Integrator ESN (LiESN).

mode Optional. Set to `'train'` if you are updating the reservoir layer for training purposes. Set to `'val'` if you are doing so for validation purposes. This will allow the reservoir object to name the training/validation modes, which can be accessed from `'training_type'` and `'val_type'` attributes.

init_size The first `init_size` data points are expended for initial transient to pass.

1.9 ESN.reset_reservoir_layer

Resets reservoir layer, i.e. sets the reservoir nodes back to their initial state.

reset_reservoir_layer() → None

1.10 ESN.set_reservoir_layer_mode

Warning: Resets reservoir layer. See *ESN.reset_reservoir_layer*.

Sets the reservoir mode to `single`, `batch` or `ensemble` by expanding or collapsing the reservoir layer (see shapes below). Changes the shape of the reservoir layer, which can be obtained from `reservoir_layer` attribute.

- `single`: reservoir layer has shape (reservoir size,1)
- `batch`: reservoir layer has shape (reservoir size,batch size)
- `ensemble`: reservoir layer has shape (number of reservoirs,reservoir size,batch size)

set_reservoir_layer_mode(*mode: str*, *batch_size: Optional[int] = None*, *no_of_reservoirs: Optional[int] = None*) → None

Parameters

mode Set to `single`, `batch` or `ensemble`.

batch_size Necessary if using `batch` or `ensemble`. If not provided `batch_size` which was specified at initialization will be used.

no_of_reservoirs Necessary if using `ensemble`. If not provided `no_of_reservoirs` which was specified at initialization will be used.

1.11 ESN.copy_from

Copies the reservoir properties to the current reservoir.

copy_from(*reservoir: Self*, *bind: bool = False*, ***kwargs*) → None

Parameters

reservoir Reservoir to copy from.

bind Shares the same memory with the reservoir that is copied from.

Keyword Arguments

verbose Give `False` to mute the messages.

1.12 ESN.copy_connections_from

Similar to *ESN.copy_from* but copies only the connection matrices.

copy_connections_from(*reservoir: Self*, *bind: bool = False*, *weights_list: Optional[list[str]] = None*, ***kwargs*) → None

Parameters

reservoir Reservoir to copy from.

bind Shares the same memory with the reservoir that is copied from.

weights_list Give a sublist of the list `['Wout','W','Win','Wback']` if you do not want to copy all the connections.

Keyword Arguments

verbose Give `False` to mute the messages.

1.13 ESN.make_connection

Creates the desired connection of the network.

make_connection(*w_name: str*, *inplace: bool = False*, ***kwargs*) → Optional[np.ndarray | torch.Tensor]

Parameters

w_name Name of the connection: `'Win'`, `'W'` or `'Wback'`.

inplace Whether to overwrite the connection.

Keyword Arguments

size User should provide information on the size associated with the corresponding connection matrix: input size for `Win`, output size for `Wback`.

verbose Give `False` to mute the messages.

1.14 ESN.delete_connection

Deletes the undesired connection of the network.

delete_connection(*w_name: str*, ***kwargs*) → None

Parameters

w_name Name of the connection: `'Win'`, `'W'` or `'Wback'`.

Keyword Arguments

verbose Give `False` to mute the messages.

1.15 ESN.cpu

Sends the reservoir to cpu device.

cpu() → None

1.16 ESN.save

Pickles the reservoir to the provided path. Save path example: `'./saved_reservoir.pkl'`

save(*save_path: str*) → None
Parameters

save_path Path to pickle the reservoir to.

1.17 ESN.load

Loads the reservoir from the provided path. Load path example: `'./saved_reservoir.pkl'`

load(*load_path: str*) → None
Parameters

load_path Path to load the reservoir from.

1.18 ESN.mute

Toggles the verbosity of the network.

mute(*verbose: Optional[bool] = None*) → None
Parameters

verbose Use this parameter to force (un)verbosity by giving `True` or `False`. If not given, the method toggles the verbosity.

1.19 ESN.forward

Warning: Updates reservoir layer. See *ESN.update_reservoir_layer*.

One step forward pass through the whole network for given input and/or output.

forward(*in_: Optional[np.ndarray | torch.Tensor] = None*, *out_: Optional[np.ndarray | torch.Tensor] = None*) → np.ndarray | torch.Tensor
Parameters

in_ Input.

out_ Output.

1.20 ESN.__call__

Passes a given input **X** through the readout: $f_{out}(\mathbf{W}_{out} \cdot \mathbf{X})$

__call__(*x: np.ndarray | torch.Tensor*) → np.ndarray | torch.Tensor
Parameters

x Input.

CHAPTER

TWO

ESNX

Echo State Network X

ESN for multitasking such as when using (mini)batches. It will set the reservoir layer mode to `batch`.

Note: *ESNX* inherits all methods from ESN.

class ESNX(*batch_size: int*, *resSize: int = 450*, *xn: list = [0, 0.4, - 0.4]*, *pn: list = [0.9875, 0.00625, 0.00625]*, *random_state: float = None*, *null_state_init: bool = True*, *custom_initState: np.ndarray = None*, ***kwargs*)

Parameters

batch_size Specify the batch size.

resSize Number of units (nodes) in the reservoir.

xn , pn User can provide custom random variable to control the connectivity of the reservoir. `xn` are the values and `pn` are the corresponding probabilities.

random_state Fix random state. If provided, `np.random.seed` and `torch.manual_seed` are called.

null_state_init If `True`, starts the reservoir from null state. If `False`, initializes randomly. Default is `True`.

custom_initState User can give custom initial reservoir state.

Keyword Arguments

verbose Mute the initialization message.

f User can provide custom activation function of the reservoir. Default is identity. Functions in the pytorch or numpy libraries are accepted, including functions defined with `np.vectorize`. Some functions can also be given as strings. Accepted strings are:

- `'tanh'`
- `'sigmoid'`
- `'relu'`
- `'leaky_{slope}'`: e.g. `'leaky_0.5'` for LeakyReLU with slope equal to 0.5.
- `'softmax'`

- `'id'`: Identity.

f_out User can provide custom output activation. Default is identity.

leak_rate Leak parameter in Leaky Integrator ESN (LiESN). Default is 1.

leak_version Give `0` for Jaeger's recursion formula, give `1` for recursion formula in ESNRLS paper. Default is 0.

bias Set strength of bias in the input, reservoir and readout connections. Disabled by default.

W , Win , Wout , Wback User can provide custom reservoir, input, output, feedback matrices.

use_torch Use pytorch instead of numpy. Will use cuda if available.

device Give `'cpu'` if `use_torch` is `True`, CUDA is available on your device but you want to use CPU.

dtype Data type of reservoir. Default is `float64`.

CHAPTER

THREE

ESNS

Echo State Network S

Ensemble of ESNs for training with multiple environments using (mini)batches. It will set the reservoir layer mode to `ensemble`.

Uses Pytorch tensors strictly.

Note: *ESNS* inherits all methods from ESN.

class ESNS(*no_of_reservoirs: int*, *batch_size: int*, *resSize: int = 450*, *xn: list = [0, 0.4, - 0.4]*, *pn: list = [0.9875, 0.00625, 0.00625]*, *random_state: float = None*, *null_state_init: bool = True*, *custom_initState: np.ndarray = None*, ***kwargs*)

Parameters

no_of_reservoirs Specify the number of reservoirs in the ensemble.

batch_size Specify the batch size.

resSize Number of units (nodes) in the reservoir.

xn , pn User can provide custom random variable to control the connectivity of the reservoir. `xn` are the values and `pn` are the corresponding probabilities.

random_state Fix random state. If provided, `np.random.seed` and `torch.manual_seed` are called.

null_state_init If `True`, starts the reservoir from null state. If `False`, initializes randomly. Default is `True`.

custom_initState User can give custom initial reservoir state.

Keyword Arguments

verbose Mute the initialization message.

f User can provide custom activation function of the reservoir. Default is identity. Functions in the pytorch or numpy libraries are accepted, including functions defined with `np.vectorize`. Some functions can also be given as strings. Accepted strings are:

- `'tanh'`
- `'sigmoid'`
- `'relu'`

- `'leaky_{slope}'`: e.g. `'leaky_0.5'` for LeakyReLU with slope equal to 0.5.
- `'softmax'`
- `'id'`: Identity.

f_out User can provide custom output activation. Default is identity.

leak_rate Leak parameter in Leaky Integrator ESN (LiESN). Default is 1.

leak_version Give `0` for Jaeger's recursion formula, give `1` for recursion formula in ESNRLS paper. Default is 0.

bias Set strength of bias in the input, reservoir and readout connections. Disabled by default.

W , Win , Wout , Wback User can provide custom reservoir, input, output, feedback matrices.

use_torch Use pytorch instead of numpy. Will use cuda if available.

device Give `'cpu'` if `use_torch` is `True`, CUDA is available on your device but you want to use CPU.

dtype Data type of reservoir. Default is `float64`.

CHAPTER

FOUR

ESNN

Echo State Network N

Echo State Network using Pytorch's Module class. $\mathbf{W}_{out}$ is a Linear layer and can be trained via gradients.

It will set the reservoir layer mode to

- `batch` if `batch_size` is specified
- `ensemble` if `no_of_reservoirs` is specified

Note: *ESNN* inherits all methods from ESN.

class ESNN(*batch_size: int*, *in_size: int*, *out_size: int*, *no_of_reservoirs: int = None*, *resSize: int = 450*, *xn: list = [0, 0.4, - 0.4]*, *pn: list = [0.9875, 0.00625, 0.00625]*, *random_state: float = None*, *null_state_init: bool = True*, *custom_initState: np.ndarray = None*, ***kwargs*)

Parameters

batch_size Specify the batch size.

in_size Specify the input length.

out_size Specify the output length.

no_of_reservoirs Specify the number of reservoirs in the ensemble.

resSize Number of units (nodes) in the reservoir.

xn , pn User can provide custom random variable to control the connectivity of the reservoir. `xn` are the values and `pn` are the corresponding probabilities.

random_state Fix random state. If provided, `np.random.seed` and `torch.manual_seed` are called.

null_state_init If `True`, starts the reservoir from null state. If `False`, initializes randomly. Default is `True`.

custom_initState User can give custom initial reservoir state.

Keyword Arguments

verbose Mute the initialization message.

f User can provide custom activation function of the reservoir. Default is identity. Functions in the pytorch or numpy libraries are accepted, including functions defined with `np.vectorize`. Some functions can also be given as strings. Accepted strings are:

- `'tanh'`
- `'sigmoid'`
- `'relu'`
- `'leaky_{slope}'`: e.g. `'leaky_0.5'` for LeakyReLU with slope equal to 0.5.
- `'softmax'`
- `'id'`: Identity.

f_out User can provide custom output activation. Default is identity.

leak_rate Leak parameter in Leaky Integrator ESN (LiESN). Default is 1.

leak_version Give `0` for Jaeger's recursion formula, give `1` for recursion formula in ESNRLS paper. Default is 0.

bias Set strength of bias in the input, reservoir and readout connections. Disabled by default.

W , Win , Wout , Wback User can provide custom reservoir, input, output, feedback matrices.

use_torch Use pytorch instead of numpy. Will use cuda if available.

device Give `'cpu'` if `use_torch` is `True`, CUDA is available on your device but you want to use CPU.

dtype Data type of reservoir. Default is `float64`.

4.1 ESNN.forward

Forward pass of the network.

Note: Output feedback is not supported yet.

1. Updates reservoir layer.
2. Returns $(\mathbf{W}_{out}^T \cdot \mathbf{U})^T$, where $\mathbf{U}$ is the extended input $[x^T; \mathbf{H}^T; \mathbf{1}]^T$ and $\mathbf{H}$ is the expanded reservoir layer.

forward(*x: torch.Tensor*) → torch.Tensor
Parameters

x Input.

INDEX