TYPE Methods
PUBLISHED 27 March 2024
DOI 10.3389/fnsys.2024.1269190

OPEN ACCESS

EDITED BY
Susan E. Shore,
University of Michigan, United States

REVIEWED BY
Alianna Maren,
Themesis, Inc., United States
Shuangming Yang,
Tianjin University, China

*CORRESPONDENCE
Cecilia Jarne
cecilia.jarne@unq.edu.ar

RECEIVED 29 July 2023
ACCEPTED 11 March 2024
PUBLISHED 27 March 2024

CITATION
Jarne C (2024) Exploring Flip Flop memories and beyond: training Recurrent Neural Networks with key insights.
Front. Syst. Neurosci. 18:1269190.
doi: 10.3389/fnsys.2024.1269190

Exploring Flip Flop memories and beyond: training Recurrent Neural Networks with key insights

Cecilia Jarne[1,2,3]*

[1]Departamento de Ciencia y Tecnologia de la Universidad Nacional de Quilmes, Bernal, Quilmes, Buenos Aires, Argentina, [2]CONICET, Buenos Aires, Argentina, [3]Department of Clinical Medicine, Center of Functionally Integrative Neuroscience, Aarhus University, Aarhus, Denmark

Training neural networks to perform different tasks is relevant across various disciplines. In particular, Recurrent Neural Networks (RNNs) are of great interest in Computational Neuroscience. Open-source frameworks dedicated to Machine Learning, such as Tensorflow and Keras have produced significant changes in the development of technologies that we currently use. This work contributes by comprehensively investigating and describing the application of RNNs for temporal processing through a study of a 3-bit Flip Flop memory implementation. We delve into the entire modeling process, encompassing equations, task parametrization, and software development. The obtained networks are meticulously analyzed to elucidate dynamics, aided by an array of visualization and analysis tools. Moreover, the provided code is versatile enough to facilitate the modeling of diverse tasks and systems. Furthermore, we present how memory states can be efficiently stored in the vertices of a cube in the dimensionally reduced space, supplementing previous results with a distinct approach.

KEYWORDS

recurrent neural networks, dynamics, computational neuroscience, flip flop, eigenvalue distribution

1 Introduction

Machine learning methods, especially Deep Learning, have achieved remarkable success across diverse tasks in various domains. These include speech processing (Ogunfunmi et al., 2019), bioinformatics (Min et al., 2016), where algorithms predict protein structures, discover drugs and analyze gene expression data, and image recognition (Litjens et al., 2017) where deep learning classifies images and detects objects.

The emergence of open-source frameworks dedicated to Machine Learning, such as Pytorch, Tensorflow and Keras (Abadi et al., 2015; Chollet et al., 2015; Paszke et al., 2019) has produced huge changes in the development of technologies we use every day for different tasks in research. Due to their novelty and complexity, it can be challenging to properly learn how to utilize these frameworks in different relevant scientific domains, such as the development of models in Computational Neuroscience, which will be the aim of the present work.

To bridge the gap between theoretical knowledge and practical application, clear tutorials or primers are crucial. These resources should equip researchers not only with the ability to implement the algorithms but also with the skills to solve diverse problems pertinent to their field.

Recurrent Neural Networks (or RNNs) were originally invented by Paul Werbos, who also invented backpropagation, a fundamental tool for training these models (Werbos, 1990). This also includes the concept of latent variables. The problem of training neural networks to perform different tasks is relevant across various disciplines that go beyond Machine Learning. In particular, RNNs are of great interest in different scientific communities. These models also have great relevance concerning control systems and other areas such as electronics (Alianna J. Maren and , Auth.; Deng, 2013; Dinh et al., 2014; Mohajerin and Waslander, 2017). One relevant problem to address with them is how to build models for the study of dynamical systems and how to extract meaningful information from them.

In general, Neural Networks are algorithms that allow us to model different systems. According to the Universal Approximation Theorem, a neural network with one hidden layer containing a sufficient but finite number of neurons can approximate any continuous function to a reasonable accuracy under certain conditions for activation functions (Hornik, 1991). This theorem has been extended to RNNs. It is well known that dynamical systems can be approximated by continuous-time RNNs (Funahashi and Nakamura, 1993).

In particular, RNNs are widely used in the field of Computational Neurosciences to describe the behavior of cortical areas, which presents great recurrence in their connections (Murphy and Miller, 2009). They are related to the processing of temporal information and the production of time-dependent outputs.

The basic premise of RNNs is that the feedforward connection weights in a Multilayer Perceptron (MLP) neural network (McCulloch and Pitts, 1943) can be modified using prior activation history as well as the immediately presented stimulus. This mechanism can be considered to encapsulate, in a very simple model, the much broader and more interesting task of guiding neural behavior. Factors that influence neural interactions and even growth can be included within this simple model. In this context, the broader scope of systems neuroscience relates to a detailed and careful analysis of RNNs.

The realm of temporal influence within systems neuroscience has a long and substantive history. The work by Levi-Montalcini and Booker (1960); Levi-Montalcini (1987), was among the earliest to show how specific signaling proteins (nerve growth factors, or NGFs) could influence temporal evolution within an organism. More recently, Baldassarro et al. (2023) showed, in an in vitro study, that NGFs could influence the proliferation of fetal brain multipotent stem cells, pushing them into a specific oligodendrocyte cell lineage and also influencing the differentiation of oligodendrocyte precursor cells. These works are simply examples of how the complex process of influencing neural cell growth and differentiation can be influenced over time, by introducing specific signaling mechanisms. For this, the notion of RNNs encapsulates a much larger suite of neural processes.

In this way, RNNs allow the incorporation of realistic characteristics at the biological level, such as Dale's law (Dale, 1935; Rajan and Abbott, 2006; Song et al., 2016; Jarne and Caruso, 2023), sparsity or different characteristics of interest in anima l models.

In the field of Machine learning, more sophisticated architectures such as LSTM (Long Short Term Memory units) or GRU (Gated recurrent units) are widely spread and have been used to process temporal sequences since they do not have the same limitations as RNNs to process long time dependencies (Bengio et al., 1994; Pascanu et al., 2013; Chung et al., 2014; SHI et al., 2015; Gudowska-Nowak et al., 2020). Other powerful models are based on spiking neural networks (SNNs). Several recent studies have made significant contributions to the field of brain-inspired intelligence. These studies demonstrate the potential of this field to achieve high-level intelligence, high accuracy, high robustness, and low power consumption (Yang et al., 2022a,b, 2023; Yang and Chen, 2023a,b).

The primary reason for using simple RNN models lies in their ability to comprehend neural computation through collective dynamics, a phenomenon intricately linked to motor control, brain temporal tasks, decision-making (Mante et al., 2013), neural oscillations and working memory (Vyas et al., 2020; Jarne and Caruso, 2023; Pals et al., 2024).

Analyzing the dynamics inherent in these models allows us to formulate various hypotheses regarding the functioning of different brain areas and to offer an interpretation for the experimental results observed (Barak, 2017; Kao and Hennequin, 2019). An illustrative instance involves the recent utilization of RNNs to transfer learned dynamics and constraints to a spiking recurrent neural network in a one-to-one fashion (Kim et al., 2019).

A well-established fact is that the dynamics of a network are heavily influenced by the eigenvalue spectrum of the weight matrix describing synaptic connections (Zhou et al., 2009). Thus, the significance of investigating this distribution lies in elucidating various aspects of the dynamic behavior of the system, which is why, in Section 5.2, such analysis will be presented and described.

There are general tutorials available on artificial neural networks, such as Yang and Wang (2020). However, in this work, we will focus extensively on RNNs and their application in Computational Neuroscience because they play a relevant role in understanding complex neural processes and dynamics. Throughout this tutorial, we will delve into the architecture, training methodologies, and practical aspects of the RNN implementation. We explore also their significance and potential contributions to the field.

A simple RNN was chosen and trained to perform a time-series processing task inspired by Computational Neuroscience studies (Sussillo, 2014). The implementation of the network, the training, and the tools are carefully described here, as well as different forms to obtain the information that allows a suitable description of the system under study.

Training an RNN to perform temporal tasks has many difficulties and can be done through various paradigms. Here it is proposed to approach the problem through supervised learning. The entire procedure is described in detail.

Among the different tasks, the Flip Flop was chosen as a case example. On one hand, a Flip Flop is the simplest sequential system that one can build Floyd (2003). To be precise, a 3-bit memory was studied, which is a task composed of a set of Flip Flops as the one shown in Figure 1. This is also a working memory task considered previously in other works in Computational Neuroscience (Sussillo and Barak, 2013; Barak, 2017; Jarne, 2022). The parameterization of the chosen task, one fundamental key in any work related to trained RNNs, is as described in Sussillo and Barak (2013). It is also revisited here. Gradient descendant minimization was used to take advantage of different optimized implementations of the available algorithms. The code implementation is presented using Tensorflow and Keras. The reason for this choice is that such scientific libraries are open-source, their use is rapidly growing, and they are becoming increasingly popular. One can find excellent documentation for software development about them Gulli and Pal (2017); Ramsundar and Zadeh (2018); Singh and Manure (2019). Also, we have new tools such as Google Colaboratory that allow implementing and testing models directly online.

The focus of this paper is on elucidating how a trained RNN operates, with code provided for detailed study. The "Flip Flop problem" is chosen to illustrate the study. Every step is thoroughly explained, from parameterizing the task to describing the dynamics of trained networks.This example is used to show how the problem of training networks can be studied using these computing tools applied in any temporal task in general, but also to discuss the limitations that networks have and the alternatives to solve them.

The rest of the paper is organized as follows. In Section 2, the description of the dynamics, discretization and code examples are presented. In Section 3, the task parametrization is shown. Section 4 describes the training protocol. In Section 5, the results, different analyses of the network, tools and software are discussed in detail. Finally, Section 6 includes the final remarks.

2 Model

The dynamics of the units in the RNN model is inspired by Equation 1, based on a model for a large network of neurons with a graded response (Hopfield, 1984).

$$\frac{dh_i(t)}{dt} = -\frac{h_i(t)}{\tau} + \sigma\left(\sum_{j=1}^{N} W_{ij}^{\mathrm{rec}} h_j(t) + \sum_{k=1}^{M} W_{ik}^{\mathrm{in}} x_k(t)\right) \tag{1}$$

The dynamics of the RNN model of N units is described in terms of the activity column vector function $\boldsymbol{h}{=}(h_1, \cdots, h_N)^{\mathrm{t}}$, where t represent the matrix transposition. The i–activity component h_i, where $i = 1, \cdots, N$ satisfies the differential equation as a function of time t. τ represent a characteristic time of the system and σ is a non-linear activation function. The elements W_{ij}^{rec} are the synaptic connection strengths of to the recurrent weight matrix $\boldsymbol{W}^{\mathrm{rec}}{\in}\mathbb{R}^{N\times N}$ and x_k are the component of the column vector function of input signal $\boldsymbol{x}{=}(x_1, \cdots, x_M)^{\mathrm{t}}$. The elements W_{ik}^{in} conform the input weight matrix $\boldsymbol{W}^{\mathrm{in}}{\in}\mathbb{R}^{N\times M}$ which connects the input signal $\boldsymbol{x}$ to each of N units with activity vector $\boldsymbol{h}$.

The network is fully connected, and matrices have weights given by a certain parametrization of interest. For the example, we considered a normal distribution with zero mean and variance $\frac{1}{N}$.

The network has three layers: the input, the recurrent hidden layer, and the output layer. The readout, in terms of the matrix elements W_i^{out}, from W^{out} is described by Equation 2.

$$z(t) = \sum_{i=1}^{N} W_i^{\mathrm{out}} h_i(t). \tag{2}$$

In terms of the output weight matrix, which in this work is a row vector, it could be written as:

$$\boldsymbol{W}^{\mathrm{out}}{=}(W_1^{\mathrm{out}}, \cdots, W_N^{\mathrm{out}}). \tag{3}$$

We considered $\sigma() = tanh()$ and $\tau = 1$, without loss of generality. The model is discretized using Euler's method following Ingrosso and Abbott (2019); Bondanelli and Ostojic (2020); Bi and Zhou (2020); Jarne (2022); Jarne and Laje (2023).

In vector form, the Equations 1 and 2 can be written as:

$$\frac{d\boldsymbol{H}(\mathbf{t})}{dt} = -\frac{\boldsymbol{H}(t)}{\tau} + \sigma(\boldsymbol{W^{Rec}}\boldsymbol{H}(t) + \boldsymbol{W^{in}}\boldsymbol{X}(t)) \tag{4}$$

and respectively:

$$\boldsymbol{Z(t)} = \boldsymbol{W^{out}}\boldsymbol{H}(t) \tag{5}$$

The system represented by Equation 1 is approximated using Euler's method as previously indicated, with a step time δt. A value of $\tau = 1$ was considered. Then, the dynamics of the discrete-time RNN is given by Equation 6

$$\boldsymbol{H}(t+\delta t) = \boldsymbol{H}(t) + (-\boldsymbol{H}(t) + \sigma(\boldsymbol{W^{Rec}}\boldsymbol{H}(t) + \boldsymbol{W^{in}}\boldsymbol{X}(t))), \tag{6}$$

The value considered for the time step is $\delta t = 1$ to obtain the time evolution. Usually, the amplitude of the activity $H(t)$ is adimensional or expressed in arbitrary units. It will depend on context. Then, from Equation 6, the activity of the recurrent units at the next time step is given by Equation 7.

$$\boldsymbol{H}(t+1) = \sigma(\boldsymbol{W^{Rec}}\boldsymbol{H}(t) + \boldsymbol{W^{in}}\boldsymbol{X}(t))) \tag{7}$$

A simple schema of the model is presented in Figure 2. The network have three inputs and three outputs corresponding to the inputs and memory states of the 3-bit Flip Flop task.

As described in Section 1, the model is implemented in Python using Keras and Tensorflow (Abadi et al., 2015; Chollet et al., 2015). This allows us to use all current algorithms and optimization methods developed and maintained by a massive research community. Tensorflow has a recurrent layer directly implemented to represent Equation 7, where it is possible to choose the initialization of the parameters, number of units and activation function. This is shown in the following code box.

```
tf.keras.layers.SimpleRNN(units, activation="tanh",
   kernel_initializer="glorot_uniform",
   recurrent_initializer="orthogonal",**kwargs)
```

Code for a Recurrent layer defined in Tensor Flow.

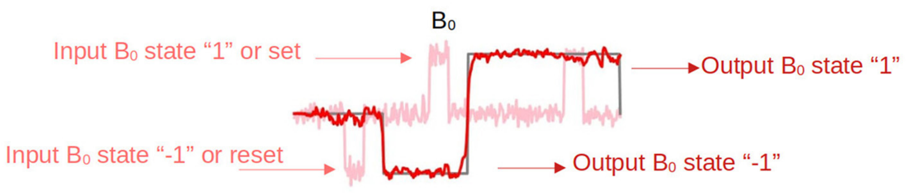

FIGURE 1
A Flip Flop. Binary task designed to store one bit of information. It has two inputs, Set (S) and Reset (R), which in here will be represented by the temporal signal states of B_0 +1 and -1 in amplitude, and one output Q. The output represents the current state of the Flip Flop and can be either 1 or -1.

The time scale of the Equation 7 is arbitrary. If we are interested, for example, in scales related to cognitive processes, we can consider 1 ms of temporal resolution.

The RNN can be initialized with different weight distributions. Several options can be selected using TensorFlow. This choice will depend, on the one hand, on the existence of some physical motivation or hypothesis of the models. On the other hand, from the ML perspective, it will depend on the performance associated with the considered initialization.

To build an RNN with the topology shown in Figure 2, it is possible to define a sequential model with layers such as:

The choice of the appropriate architecture will depend on the system to be modeled. Particular features, such as bias terms, can be also considered. In some cases, it is possible, to build architectures with additional features that are not pre-defined. This can be done by using the class structures in the framework.

For example, it is possible to define your own RNN cell layer (the inner part of the for loop) with custom behavior and use it with the generic `keras.layers.RNN` layer (the for loop itself). For more details see: https://www.tensorflow.org/guide/keras/working_with_rnns.

```
model = Sequential()
model.add(SimpleRNN(units=N, input_shape=(None,3),activation="tanh"))
model.add(Dense(units=3,input_dim=N))
```

Code for the sequential model defined for the network in Figure 2.

Where `input_shape=(None,3)` means the shape of the input vector, `activation='tanh'` corresponds to the definition of activation function, and `Dense` is a fully connected output layer. In this way, we completed the first step which is of the model definition in terms of the code. Orher acivation funcions are avaliable at https://www.tensorflow.org/api_docs/python/tf/keras/activations.

Other network architectures, such as Gated Recurrent Units (Chung et al. (2014)) or Long Short Term Memories (SHI et al., 2015), could be selected if there was any motivation from the perspective of the mechanisms to take into account. Both are already implemented in TensorFlow. Such code options are shown in the boxes below.

3 Task selection and parametrization

The parameterization of the task will have strong consequences on the possible dynamics obtained from the system through network training (Jarne and Laje, 2023). Possible examples are considering training with noise vs. without noise. Another is to consider amplitude variations or pulses of variable width in the training set.

Previous works have considered some relevant tasks in Computational Neuroscience related to decision-making or working memory. For example in Jarne (2021, 2022); Jarne and Laje (2023). All these processes use time-varying signals, which are very different from the binary boolean operations considered with forward networks. There are other examples of widespread

```
model = Sequential()
model.add(layers.GRU(units=N, return_sequences=True))
...
```

Code using other architecture (GRU) for the sequential model defined for Figure 2.

```
model = Sequential()
model.add(layers.LSTM(units=N, input_shape=(None,3)))
...
```

Code using other architecture (LSTM) for the sequential model defined for Figure 2.

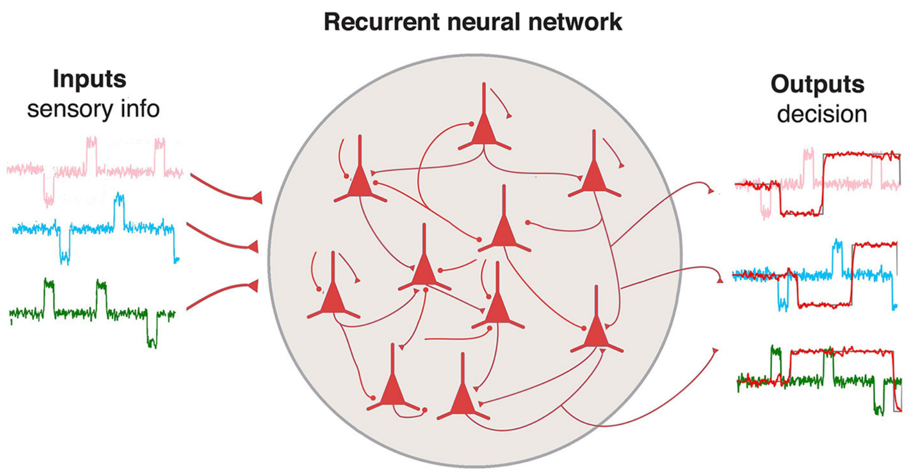

FIGURE 2
Schema for the RNN described by Equations 1 and 2. The network has three inputs and outputs to build the different memory states for the 3-bit Flip Flop task.

tasks also considered in Computational Neuroscience, such as "Perceptual Decision Making" (Britten et al., 1992) or "Context-dependent Decision Making" (Mante et al., 2013). Each task has different possible parameterizations. In particular, the task defined in Mante et al. (2013) has recently been used to study the cortex response (Zhang et al., 2021).

It is also possible to consider working memory tasks such as "Delay match to sample with two items" (Freedman and Assad, 2006) or "Parametric working memory" (Roitman and Shadlen, 2002). For present work, motivated by Sussillo and Barak (2013), a working memory task, a 3-bit Flip Flop was chosen.

A Flip Flop is a binary task designed to store one bit of information. It has two inputs, Set (S) and Reset (R), which in our case will be represented by the temporal signal states +1 and -1 in amplitude, and one output Q. The output represents the current state of the Flip Flop and can be either 1 or -1. The operation is based on the following rule: If the input is 1, the Flip Flop output is set to the "1" state. If input is -1, the Flip Flop output is reset to the "-1" state. If both inputs are 0, the Flip Flop remains in its current state.

Once the task is chosen, the requirements must be translated into an algorithm that allows us to generate the training set. To parameterize the task, the following criteria were applied here:

- The possible states of the Flip Flop are represented in such a way that a positive pulse represents a set and a negative pulse represents a reset.
- The state of the output will change corresponding to the input command.
- A certain delay in the response was considered after the falling edge of the input signal.

The training data set consists of time series with pulses of fixed duration that represent set and reset signals. Those signals can be activated randomly and are separated by a random time interval. In all time series, a certain noise level has been added to the input. Each input elicitate a target output according to the Flip Flop rule: if we have a set signal or positive pulse, the output is in a high state. If we have a reset signal or negative pulse, output is in a low state. Otherwise, the output remains in the previous state.

The number of inputs in the network corresponds to the number of memory states that can be stored. A Flip Flop is a one-bit memory, meaning that two states only can be stored. In this way, we have registers formed by three Flip Flops (a 3-bit memory), which means that we have 8 different memory states.

To complete the full training data set, it is necessary to generate tensors of size `sample_size` with the input time series of length `time_series_lengh` for each of the three inputs and outputs. To do that efficiently, we used Numpy arrays (Harris et al., 2020). In the present work, we provide the code to generate a Flip Flop data set. Three random components of the set `x_train-y_train` are shown in Figure 3. Input has amplitude noise of 10%. The target output, `y_train`, was simulated with a time delay answer of 20 ms. Each row (and color) corresponds to one of the inputs, and each column to a different sample. Each training sample consists of a Numpy array (Harris et al., 2020). This is shown in the following code box.

```
x_train[sample_size,time_series_lengh,3]
y_train[sample_size,time_series_lengh,3]
```

Training data set pairs defined as Numpy arrays.

4 Training protocol and parameter selection

Training methods for neural networks can be unsupervised or supervised. We focused on applying a supervised method.

Different approaches are available, but those in which a particular type of gradient descent method is applied stand out in

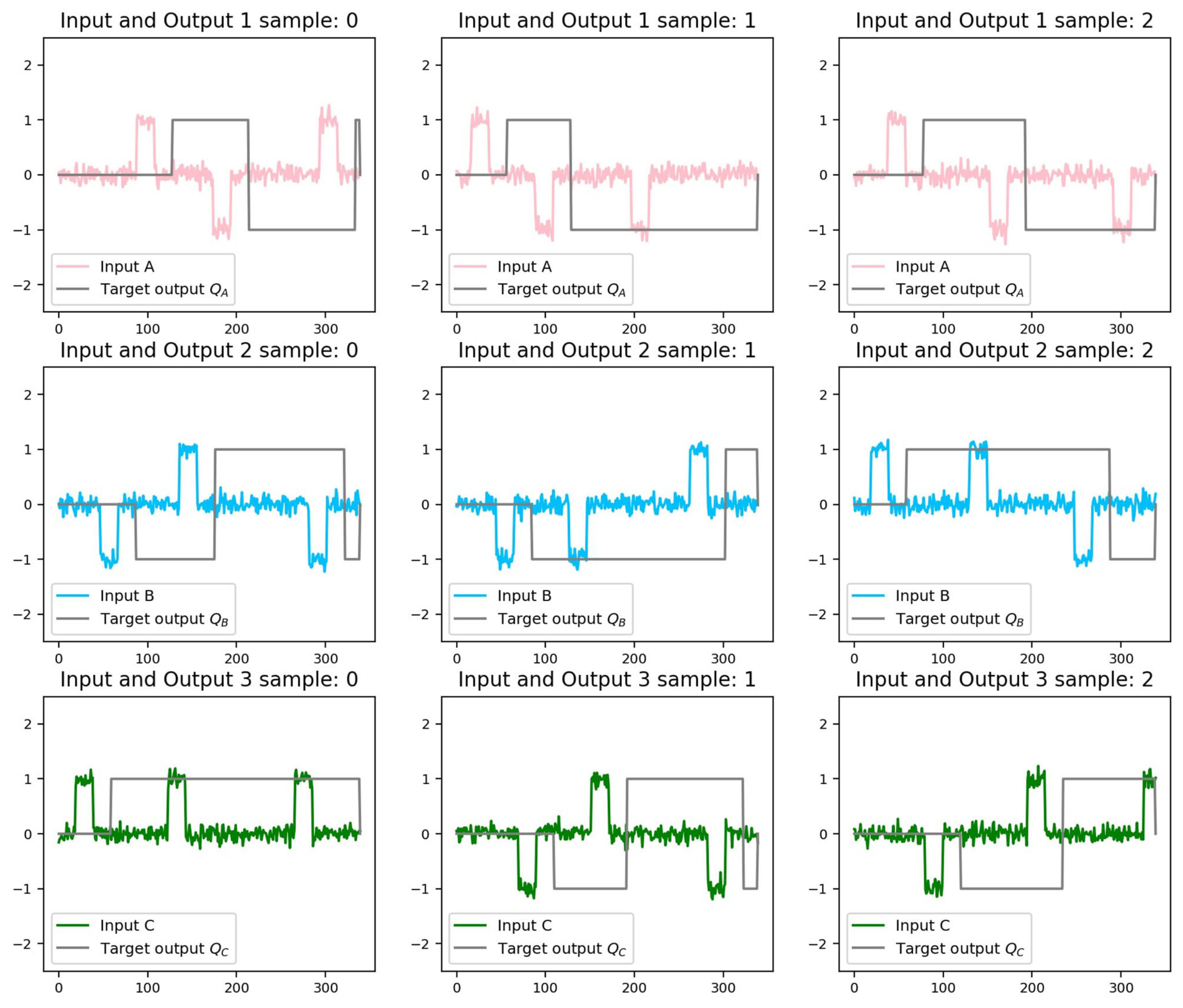

FIGURE 3
Three random samples of the training data set for each input channel. Each row (and color) corresponds to one of the inputs. Each column corresponds to a different sample. The gray line in each case represents the target output.

the literature. An example is the paradigm of Reservoir Computing, specifically the use of liquid (or echo-state networks) (Maass et al., 2002), where the modifications of the network weights are made only in the weights of the output layer, W^{out}.

Other outstanding approaches were developed by Sussillo and Abbot. They have developed a method called FORCE that allows the reproduction of complex output patterns, including human motion-captured data (Sussillo and Abbott (2009)). Modifications to the algorithm have also been applied successfully in various applications (DePasquale et al., 2018; Ingrosso and Abbott, 2019; Engelken et al., 2022).

The other method used for estimation of the gradient in RNNs is called Backpropagation Through Time (BPTT), and then an optimization method for minimizing the gradient. Given the recent advances in the implementation of this method with the open-source libraries previously mentioned, this is the method used here. Other back propagation-based methods have been published more recently. For example, in Khan et al. (2018), authors proposed to use fractional calculus to improve the conventional BPTT.

In this work, supervised learning was used, with standard backpropagation through time. An Adaptive Stochastic Gradient Descent training method provided by the Keras framework (Kingma and Ba, 2014) was applied.

First, recurrent weights were initialized using a random normal distribution with the orthogonal condition on the matrix. During training, noisy square pulse signals were used as the inputs, as the examples shown in Figure 3, and described in Section 3. In this way, sets of time series with 350 time points were generated containing random positive and negative pulses, with their corresponding output according to the operating rule described for the Flip Flop.

The appropriate loss function to train the model is the mean square error between the target function and the output of the network. It is defined as:

$$E(w) = \frac{1}{2}\sum_{t=1}^{M}\sum_{j=1}^{L}|\mathbf{Z_j}(t) - \mathbf{Z_j^{target}}(t)|^2, \tag{8}$$

where $\mathbf{Z_j^{target}}(t)$ is the desired target function and $\mathbf{Z_j}(t)$ is the actual output.

The training set consisted of more than 15000 different random samples. The previously mentioned training procedures correspond, in terms of the code structure, to the methods for compiling and fitting models. The loss function and the optimizer algorithm are chosen in the compiling step. Different information about the training data set, epochs, and other training characteristics can be specified with the fitting method. An example of implementation is shown in the following code box.

```
model.compile(loss = ''mse'', optimizer=ADAM)
model.fit(x_train[50:sample_size,:,:], y_train[50:sample_size,:,:],
    epochs=epochs, batch_size=128, shuffle=True)
```

Code for the compiling and training steps.

The main parameters of a neural network are the weights of the connections. These parameters are learned during the training stage. On the other hand, hyperparameters are parameters of your neural network that can not be learned via gradient descent or some other training method. These include the learning rate, number of layers, or the number of neurons in a given layer.

Tuning the hyperparameters means the process of choosing the best values of them. Typically, this is done by evaluating the performance of the network on a validation set. Then, we have to change the hyperparameters and re-evaluate the model, choosing the values that give the best performance on the validation set. Another approach for choosing them is to have an informed decision or hypothesis related to the physics or nature of the system under study.

How do we choose these values? Often, there is good standard initialization related to each particular task of interest. An example of the criteria is provided for the Flip Flop task in Table 1.

TABLE 1 Model's parameters and criteria for the network's implementation and training.

Parameter/criteria	Value
Units	400
Time step	1
Input Weight	3 × 400
Recurrent Weights	400 × 400
Output Weight	400 × 3
Training algorithm	BPTT ADAM
Initialization	Random Orthogonal
Regularization	None

Another aspect to consider is the regularization of the model. Regularization refers to training our model well enough that it can generalize over data it hasn't seen before.

To summarize, in the training stage, the main aspects we have to consider are the size of the network, data set, noise, and regularization terms that are appropriate for the considered task.

A good practice is to build a set of RNNs (at least a dozen) with different hyperparameters that are correctly trained to perform the same task and that can serve as a test set and allow us to compare the variations in the possible solutions.

5 Analyzing the results

The results obtained after training the RNN can be analyzed in several ways. On the one hand, we can consider the quality of the solutions obtained (and the robustness) by analyzing the predicted output concerning the target and stability against noise conditions. On the other hand, we can study the solutions in terms of the dynamics and collective behavior. In Section 5.1, we will first briefly discuss how to evaluate robustness in terms of the output obtained and how robust it can be against variations in the input stimuli. Then, in Section 5.2, we will discuss in-depth details of the dynamics and collective behavior.

5.1 Evaluating RNN performance and robustness

We can measure the rate of success for a set of networks in terms of Euclidean distance between target and output (Jarne and Laje, 2023). The distance between the network's predicted output and the target output could be estimated using the Numpy function `linalg.norm()`, which in this case is the Frobenius norm (or Euclidean norm) between the output vector of the trained network and the target output. We could use other metrics, such as Mean squared error (MSE), which measures the average squared difference between predicted and actual values.

```
from numpy import linalg as LA
euclidean_norm = LA.norm(Difference)
```

How to use linear algebra library from Numpy to calculate the norm of the vector "Difference" using the Euclidean norm.

We can also include noise and variations in amplitude to the data and find constraints on how the network is still able to accurately predict the target.

In addition to the metrics mentioned above, we can also evaluate the robustness using other available Accuracy metrics.

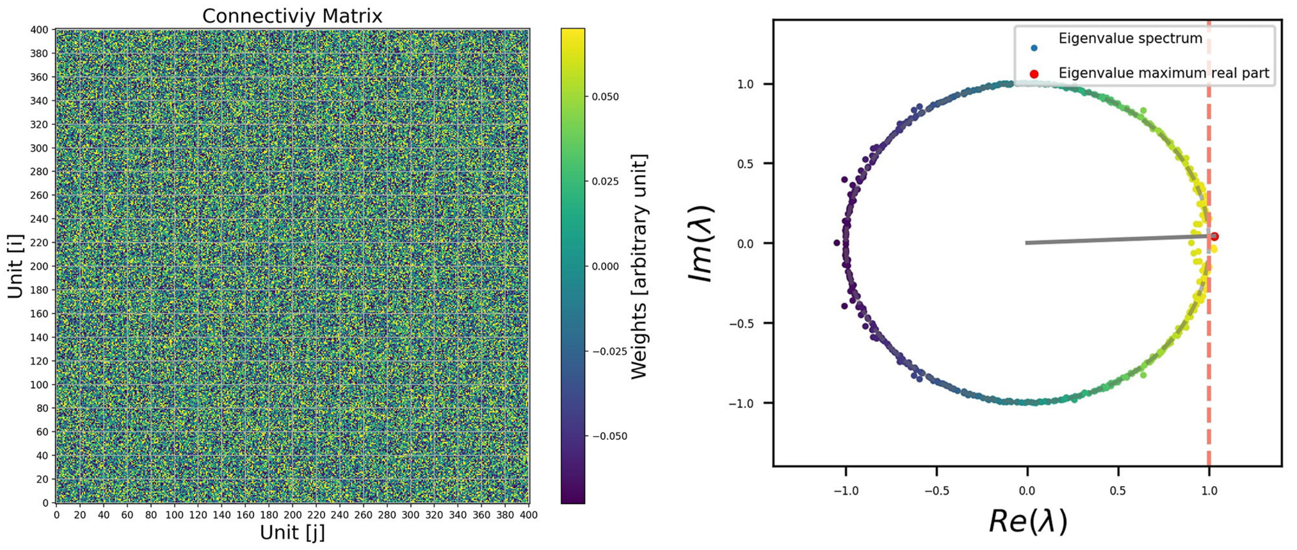

FIGURE 4
Left: Example of visualization for the connectivity matrix for a trained RNN. **Right**: Eigenvalue distribution of W^{rec} in the complex plane.

In general, what type of data sets we use to train the network is directly related to how robust we want to design our systems and what properties we are seeking to represent and should be taken into account when comparing the abstract models studied here with those obtained from experimental data.

Characterizing the limitations of the network concerning variations in input signals is a good practice that will allow us to better understand the results obtained.

Another analysis related to robustness that can be performed is to determine the minimum size of the network that allows the parameterized task to be performed (in our example Flip Flop), given a certain desired accuracy.

5.2 Analyzing the collective behavior of trained RNNs: visualization and dimensionality reduction techniques

After training, we obtained a set of RNNs that can perform the tasks of interest. We describe in this section the different aspects to analyze regarding the network's collective behavior. We selected a method for the model's visualization and a group of tools to extract the relevant information.

For example, it is possible to visualize the connectivity matrix (recurrent weight matrix), as it is shown on the left side of Figure 4. The columns represent the output connection of the i-neuron, and the rows are the input connection. They are also called post-synaptic and pre-synaptic. The color bar on the right side represents the intensity of the connections. We have to consider an appropriate scale for the visualization. Even so, it may not be entirely clear how to observe the relevant information, apart from the fact that, after training, most of the weights remain close to zero. As a first approach for visualization, a plot of the connectivity matrix could be useful, even if the case presented here does not reveal relevant information. It's important not to undervalue it. If the connectivity has some structure farther from a random distribution, it will be observed in the connectivity plot. For example, having null autoconnection terms will be reflected in the color of the diagonal terms of the matrix plot. Another example could be sparsity, which would be reflected in patches over the matrix. Or perhaps, in the case of having excitatory and inhibitory units, it would be easy to visualize the different columns corresponding to the same sign of out connection. In case of imposing such constraints on the connections, as Dales' Law (Dale, 1935), or any particular constraint, they will be visible in this stage, and this representation will be more useful.

If we don't obtain relevant information with this first visualization, we know that useful information could be still encoded in the connectivity matrix, but it may not be immediately distinguishable with a connectivity plot. There are different transformations or analyses that we can perform on the recurrent weight matrix with this aim. Different Linear Algebra operations are available in the Numpy Library (Harris et al., 2020) that are optimized to be used with the array structures. For example, if we perform a decomposition of W^{rec} in their eigenvectors and eigenvalues, we can obtain the eigenvalue distribution as it is shown in the right side of Figure 4. This analysis can be done using the code in the following code box.

```
from numpy import linalg as LA
eigenvalues, eigenvectors= LA.eig(Matrix)
```

How to use linear algebra library from Numpy for eigenvalue decomposition.

During training, the matrix associated with the network tends to be non-normal, which results in their eigenvalues lying closer to the unit circle. This behavior is explained in more detail in papers that study the dynamics of RNNs, where it is shown that

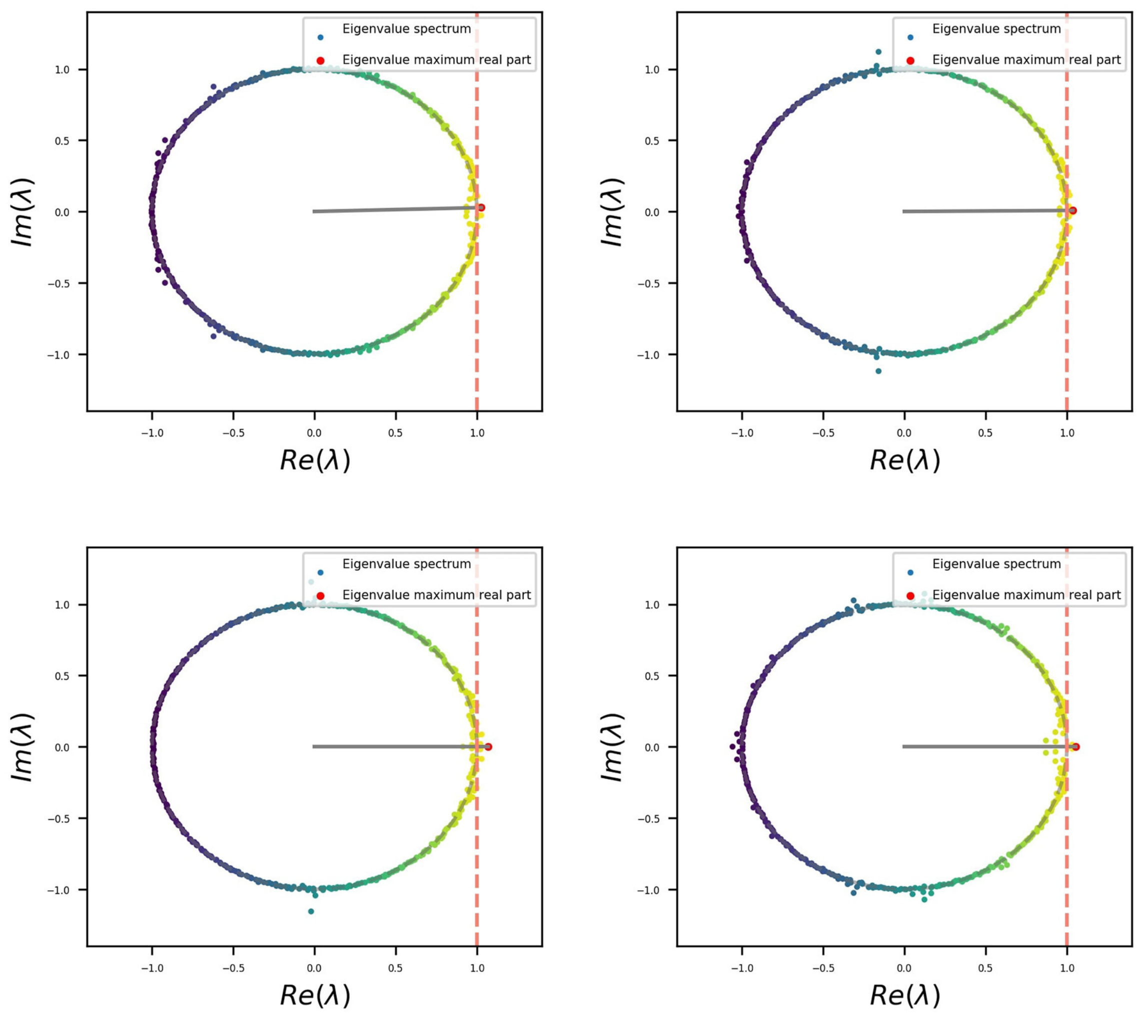

FIGURE 5
Four different examples of eigenvalue distributions of W^{rec} for trained RNNs which have been initialized before with an orthogonal distribution.

the presence of recurrent connections and the attractors in the network's dynamics can cause this accumulation of eigenvalues close to the unit circle (Asllani et al., 2018; Bondanelli and Ostojic, 2020; Jarne, 2022).

Additionally, in these studies, it is typically shown that this accumulation of eigenvalues on the unit circle leads to slowing down the dynamics of the network. They can be linked to the emergence of long-term memories related to the linearization of the system. Therefore, this behavior can be understood as a necessary condition for the network to effectively store and retrieve information over longer time scales.

In the case presented here, we can visualize that, except for a small group of eigenvalues that migrated out of the unit circle, the rest remain on it, which is related to the initial orthogonal condition. The same was replicated throughout all simulations. A set of four examples is shown in Figure 5, and the code provided allows us to reproduce more. Eigenvalues outside the unitary circle seem to be related to the behavior (or modes) observed for the different stimuli at the input as described in (Jarne, 2022). This is relevant in terms of the dynamics. Additional information related to the connectivity matrix could also be obtained (Jarne, 2022).

Other possible studies that we can perform are related to the response in terms of the activity of the network units when applying the different stimuli.

Since we have a large number of units, and for each, an activity vector, dimensionally reduction methods are appropriate to analyze such behavior. They have been used widely in different works related to large-scale neural recordings (Cunningham and Yu, 2014; Williams et al., 2018).

Scikit-learn (Pedregosa et al., 2011) is a Python open-source library based on Numpy that allows us to perform dimensionality reduction, feature extraction, and normalization, among others. It has efficient methods for predictive data analysis. A possible decomposition could be, for example, Principal Component Analysis (PCA) or also Single Value Decomposition (SVD). The following code box shows how to call the library's functions.

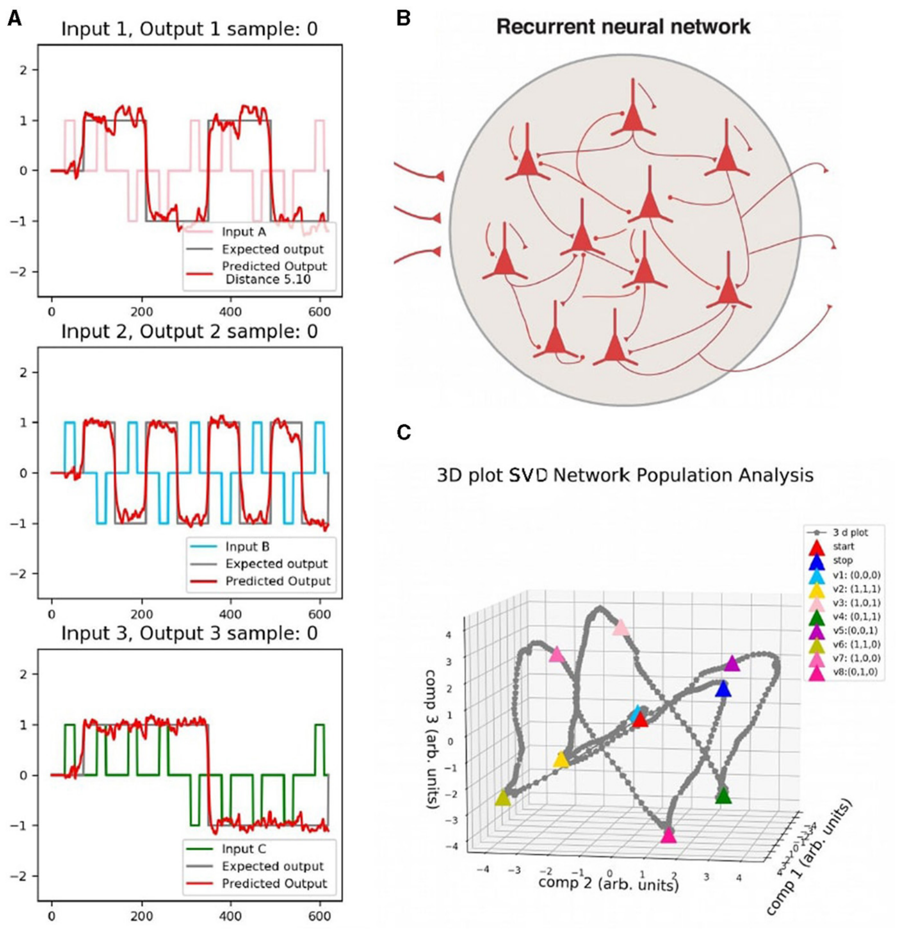

FIGURE 6
(A) Data set for testing. Each panel corresponds to one input and the predicted output of the Flip Flop, which is shown in red. **(B)** Schema of the network. **(C)** Single Value Decomposition applied to the activity vector H(*t*) of panel b) in the three components of greatest variance. Each color point corresponds to a different memory state.

```
from sklearn.decomposition import PCA
from sklearn.decomposition import TruncatedSVD
```

How to import scikit learn libraries to perform single value decomposition and principal component analysis.

These tools can be used to extract relevant features of the system. For this work, the behavior, in terms of the activity of the units, was analyzed.

It is well known that the different memory states in a 3-bit memory are distributed in the vertex of a cube-like form in the space state (Sussillo and Barak, 2013). This was shown when authors explored the hypothesis that fixed points, both stable and unstable, and the linearized dynamics around them, can reveal aspects of how RNNs implement their computations.

A data set was built for testing and reproducing the behavior. It generates eight different memory states, as shown on the left side of Figure 6, where each panel shows the input and output of the network. Time series of 600 ms were considered to generate all the different memory states of the 3-bit memory by choosing the correct commutation for the inputs in fixed time intervals. Output responses are shown in red in the figure.

The testing set is injected into the network (right upper panel of the figure), and then the activity of the units is analyzed by applying SVD on the activity vector $H(t)$. The behavior of the system was represented in the three axes of the greatest variance. The bottom right part of Figure 6 shows the activity in the reduced state space (3-dimensional). Each vertex corresponds to each memory state marked in different colors.

It is well known that different variations of the realizations, in terms of weight distribution and dynamical behavior, are possible when training networks for the same task (Jarne, 2021, 2022; Jarne and Laje, 2023). This was exemplified in Figure 5 and is also shown in Figure 7, where the four different realizations of the trained networks of Figure 5 were elicited with the same testing data set and a decomposition SVD analysis, was performed.

The vertices in this space of the main components are distributed in different positions. A cube-like structure always appears, similarly to what was observed in Sussillo and Barak (2013), and is rotated in different spatial directions for different

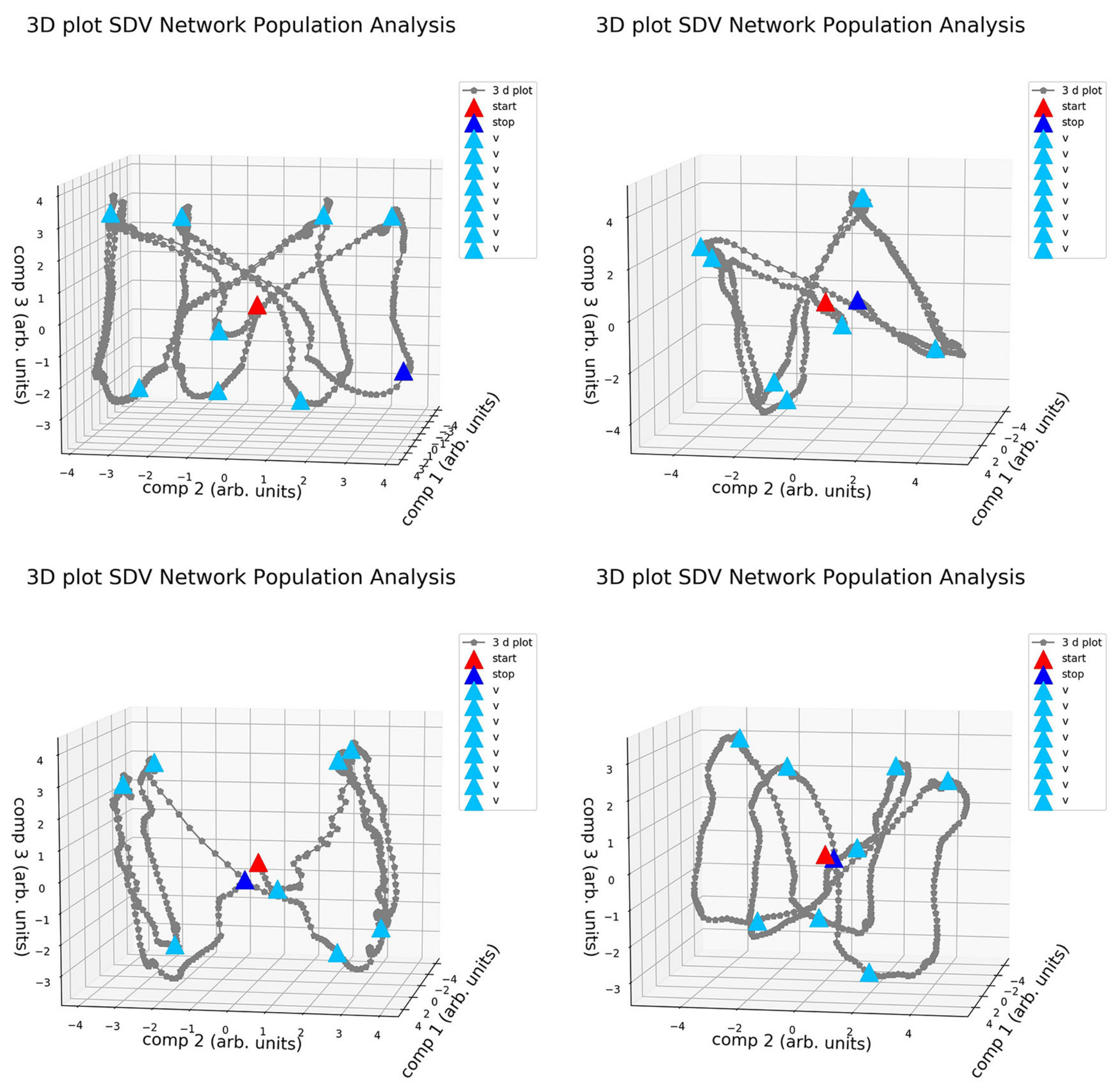

FIGURE 7
Activity of the four different realizations of RNNs trained for the same 3-bit Flip Flop task in the dimensionally reduced space. Configurations are similar to one presented in Figure 5.

realizations. It is possible to study and classify the behavior of the obtained systems by comparing the network obtained. This cube-like structure is characteristic of this task parametrization, and it appears even when we used a different training method and network parameters compared with previous studies (Sussillo and Barak, 2013).

Additional analysis could be considered depending on the aspects of interest to be studied. Here, a minimal analysis was proposed. We described in detail the steps, visualization tool, criteria, and implementation. The code for training and analysis is provided also in an open repository: https://github.com/katejarne/3-bit-FF-tutorial/tree/main/paper. It can be also used as an open framework to parametrize different tasks or additional studies. In this way, we can generate and compare the different realizations for the Flip Flop task or define different tasks for study.

6 Conclusions

In this work, all steps to build and analyze an RNN have been presented for a sample task. We started from the model description in terms of the equations, discretization, and code implementation. We discussed different options that are available for code implementation depending on the considered model and scientific questions. Then, we described the task parametrization and network training protocol. We also presented a set of tools to analyze the results using open-source scientific libraries making use of the different visualization tools that allow extracting relevant features.

We used the Flip Flop task as an example, but other relevant tasks could be considered, as mentioned in Section 3. For example, “Perceptual Decision Making” (Britten et al., 1992), “Context-dependent Decision Making” (Mante et al., 2013; Zhang et al., 2021), working memory tasks such as "Delay match to sample with two items" (Freedman and Assad, 2006) or "Parametric working memory" (Roitman and Shadlen, 2002). In this work, motivated by Sussillo and Barak (2013), a working memory task such as a 3-bit Flip Flop, was chosen to show the entire process: from the differential equations of the RNN model, discretization, through the parameterization of the task and the methods of analysis for the activity of the network against the different stimuli on the network.

The use of open-source scientific frameworks designed and maintained for large communities, such as the tools used here, allows enhancing research. This is why we are currently using tools that are more transparent in terms of code and documentation because they are open to being modified and improved by thousands of users.

Regarding the limitations, the proposed method was evaluated on a single cognitive task, namely Flip Flop. It is not clear whether the proposed pipeline would generalize to other more complex types of cognitive tasks. We did not include other explicit biological constraints in this example. We could extend it to include sparsity or Dale's law, for example. Further work could address such research directions to complement generalization and biological details.

Data availability statement

The datasets presented in this study can be found and also created using the online repository: https://github.com/katejarne/3-bit-FF-tutorial.

Author contributions

CJ: Conceptualization, Formal analysis, Investigation, Methodology, Resources, Software, Validation, Visualization, Writing – original draft, Writing – review & editing.

Funding

The author(s) declare that financial support was received for the research, authorship, and/or publication of this article. Author acknowledges support from PICT 2020-01413.

Conflict of interest

The author declares that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's note

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., et al. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Available online at: tensorflow.org (accessed March 19, 2024).

Alianna J., Maren, C. T. H., and Pap, R. M. (1990). *Handbook of Neural Computing Applications*. Cambridge: Elsevier Inc, Academic Press.

Asllani, M., Lambiotte, R., and Carletti, T. (2018). Structure and dynamical behavior of non-normal networks. Sci. Adv. 4, eaau9403. doi: 10.1126/sciadv.aau9403

Baldassarro, V. A., Cescatti, M., Rocco, M. L., Aloe, L., Lorenzini, L., Giardino, L., et al. (2023). Nerve growth factor promotes differentiation and protects the oligodendrocyte precursor cells from *in vitro* hypoxia/ischemia. *Front. Neurosci.* 17, 1111170. doi: 10.3389/fnins.2023.1111170

Barak, O. (2017). Recurrent neural networks as versatile tools of neuroscience research. Curr. Opini. Neurobiol. 46, 1–6. Computational Neuroscience. doi: 10.1016/j.conb.2017.06.003

Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Trans. Neural Netw.* 5, 157–166. doi: 10.1109/72.279181

Bi, Z., and Zhou, C. (2020). Understanding the computation of time using neural network models. *Proc. Nat. Acad. Sci.* 117, 10530–10540. doi: 10.1073/pnas.1921609117

Bondanelli, G., and Ostojic, S. (2020). Coding with transient trajectories in recurrent neural networks. *PLoS Comput. Biol.* 16, 1–36. doi: 10.1371/journal.pcbi.1007655

Britten, K., Shadlen, M., Newsome, W., and Movshon, J. (1992). The analysis of visual motion: a comparison of neuronal and psychophysical performance. *J. Neurosci.* 12, 4745–4765. doi: 10.1523/JNEUROSCI.12-12-04745.1992

Chollet, F., et al. (2015). *Keras*. Available online at: http://citebay.com/how-to-cite/keras/ (accessed March 19, 2024).

Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2014). "Empirical evaluation of gated recurrent neural networks on sequence modeling," in *NIPS 2014 Workshop on Deep Learning, December 2014.*

Cunningham, J. P., and Yu, B. M. (2014). Dimensionality reduction for large-scale neural recordings. *Nat. Neurosci.* 17, 1500–1509. doi: 10.1038/nn.3776

Dale, H. (1935). Pharmacology and nerve-endings. *Proc. R. Soc. Med.* 28, 319–332. doi: 10.1177/003591573502800330

Deng, J. (2013). Dynamic neural networks with hybrid structures for nonlinear system identification. *Eng. Appl. Artif. Intell.* 26, 281–292. doi: 10.1016/j.engappai.2012.05.003

DePasquale, B., Cueva, C. J., Rajan, K., Escola, G. S., and Abbott, L. F. (2018). full-force: a target-based method for training recurrent networks. *PLoS ONE* 13, 1–18. doi: 10.1371/journal.pone.0191527

Dinh, H., Kamalapurkar, R., Bhasin, S., and Dixon, W. (2014). Dynamic neural network-based robust observers for uncertain nonlinear systems. *Neural Netw.* 60, 44–52. doi: 10.1016/j.neunet.2014.07.009

Engelken, R., Ingrosso, A., Khajeh, R., Goedeke, S., and Abbott, L. F. (2022). Input correlations impede suppression of chaos and learning in balanced firing-rate networks. *PLoS Comput. Biol.* 18, 1–23. doi: 10.1371/journal.pcbi.1010590

Floyd, T. (2003). *Digital Fundamentals*. Hoboken: Prentice Hall.

Freedman, D. J., and Assad, J. A. (2006). Experience-dependent representation of visual categories in parietal cortex. *Nature* 443:85–88. doi: 10.1038/nature05078

Funahashi, K., and Nakamura, Y. (1993). Approximation of dynamical systems by continuous time recurrent neural networks. *Neural Networks* 6, 801–806. doi: 10.1016/S0893-6080(05)80125-X

Gudowska-Nowak, E., Nowak, M. A., Chialvo, D. R., Ochab, J. K., and Tarnowski, W. (2020). From synaptic interactions to collective dynamics in random neuronal networks models: critical role of eigenvectors and transient behavior. *Neural Comput.* 32, 395–423. doi: 10.1162/neco_a_01253

Gulli, A., and Pal, S. (2017). *Deep Learning with Keras: Implementing Deep Learning Models and Neural Networks with the Power of Python*. Birmingham: Packt Publishing.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., et al. (2020). Array programming with numpy. *Nature* 585, 357–362. doi: 10.1038/s41586-020-2649-2

Hopfield, J. J. (1984). Neurons with graded response have collective computational properties like those of two-state neurons. *Proc. Nat. Acad. Sci.* 81, 3088–3092. doi: 10.1073/pnas.81.10.3088

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks* 4, 251–257. doi: 10.1016/0893-6080(91)90009-T

Ingrosso, A., and Abbott, L. F. (2019). Training dynamically balanced excitatory-inhibitory networks. *PLoS ONE* 14, 1–18. doi: 10.1371/journal.pone.0220547

Jarne, C. (2021). Multitasking in RNN: an analysis exploring the combination of simple tasks. *J. Phys. Complex* 2, 015009. doi: 10.1088/2632-072X/abdee3

Jarne, C. (2022). Different eigenvalue distributions encode the same temporal tasks in recurrent neural networks. *Cogn. Neurodyn.* 17, 257–275. doi: 10.1007/s11571-022-09802-5

Jarne, C., and Caruso, M. (2023). Effect in the spectra of eigenvalues and dynamics of RNNs trained with excitatory-inhibitory constraint. *Cogn. Neurodyn.* doi: 10.1007/s11571-023-09956-w

Jarne, C., and Laje, R. (2023). Exploring weight initialization, diversity of solutions, and degradation in recurrent neural networks trained for temporal and decision-making tasks. *J. Comput. Neurosci.* 51, 407–431. doi: 10.1007/s10827-023-00857-9

Kao, T.-C., and Hennequin, G. (2019). Neuroscience out of control: control-theoretic perspectives on neural circuit dynamics. *Curr. Opin. Neurobiol.* 58, 122–129. doi: 10.1016/j.conb.2019.09.001

Khan, S., Ahmad, J., Naseem, I., and Moinuddin, M. (2018). A novel fractional gradient-based learning algorithm for recurrent neural networks. *Circu. Syst. Signal Proc.* 37, 593–612. doi: 10.1007/s00034-017-0572-z

Kim, R., Li, Y., and Sejnowski, T. J. (2019). Simple framework for constructing functional spiking recurrent neural networks. *Proc. Nat. Acad. Sci.* 116, 22811–22820. doi: 10.1073/pnas.1905926116

Kingma, D. P., and Ba, J. (2014). Adam: a method for stochastic optimization. *arXiv preprint* arXiv:1412.6980.

Levi-Montalcini, R. (1987). The nerve growth factor 35 years later. *Science* 237, 1154–1162. doi: 10.1126/science.3306916

Levi-Montalcini, R., and Booker, B. (1960). Excessive growth of the sympathetic ganglia evoked by a protein isolated from mouse salivary glands<sup>*</sup>. *Proc. Nat. Acad. Sci.* 46, 373–384. doi: 10.1073/pnas.46.3.373

Litjens, G., Kooi, T., Bejnordi, B. E., Setio, A. A. A., Ciompi, F., Ghafoorian, M., et al. (2017). A survey on deep learning in medical image analysis. *Med. Image Anal.* 42:60–88. doi: 10.1016/j.media.2017.07.005

Maass, W., Natschläger, T., and Markram, H. (2002). Real-time computing without stable states: a new framework for neural computation based on perturbations. *Neural Comput.* 14, 2531–2560. doi: 10.1162/089976602760407955

Mante, V., Sussillo, D., Shenoy, K. V., and Newsome, W. T. (2013). Context-dependent computation by recurrent dynamics in prefrontal cortex. *Nature* 503, 78–84. doi: 10.1038/nature12742

McCulloch, W. S., and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bull. Math. Biophys.* 5, 115–133. doi: 10.1007/BF02478259

Min, S., Lee, B., and Yoon, S. (2016). Deep learning in bioinformatics. *Brief. Bioinformatics* 18, 851–869. doi: 10.1093/bib/bbw068

Mohajerin, N., and Waslander, S. L. (2017). "State initialization for recurrent neural network modeling of time-series data," in *2017 International Joint Conference on Neural Networks (IJCNN)* (Anchorage: IEEE), 2330–2337. doi: 10.1109/IJCNN.2017.7966138

Murphy, B. K., and Miller, K. D. (2009). Balanced amplification: A new mechanism of selective amplification of neural activity patterns. *Neuron* 61, 635–648. doi: 10.1016/j.neuron.2009.02.005

Ogunfunmi, T., Ramachandran, R. P., Togneri, R., Zhao, Y., and Xia, X. (2019). A primer on deep learning architectures and applications in speech processing. *Circu. Syst. Signal Proc.* 38, 3406–3432. doi: 10.1007/s00034-019-01157-3

Pals, M., Macke, J. H., and Barak, O. (2024). Trained recurrent neural networks develop phase-locked limit cycles in a working memory task. *PLoS Comput. Biol.* 20, 1–23. doi: 10.1371/journal.pcbi.1011852

Pascanu, R., Mikolov, T., and Bengio, Y. (2013). "On the difficulty of training recurrent neural networks," in *Proceedings of the 30th International Conference on Machine Learning, PMLR, Vol. 28*, 1310–1318.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., et al. (2019). "Pytorch: an imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, eds. H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alché-Buc, E. Fox, and R. Garnett (New York: Curran Associates Inc), 8024–8035.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., et al. (2011). Scikit-learn: machine learning in python. *J. Mach. Learn. Res.* 12, 2825–2830. doi: 10.48550/arXiv.1201.0490

Rajan, K., and Abbott, L. F. (2006). Eigenvalue spectra of random matrices for neural networks. *Phys. Rev. Lett.* 97, 188104. doi: 10.1103/PhysRevLett.97.188104

Ramsundar, B., and Zadeh, R. B. (2018). *TensorFlow for Deep Learning: From Linear Regression to Reinforcement Learning*. Sebastopol, CA: O'Reilly Media Inc.

Roitman, J. D., and Shadlen, M. N. (2002). Response of neurons in the lateral intraparietal area during a combined visual discrimination reaction time task. *J. Neurosci.* 22, 9475–9489. doi: 10.1523/JNEUROSCI.22-21-09475.2002

SHI, X., Chen, Z., Wang, H., Yeung, D.-Y., Wong, W.-, k., and, W. O. O., et al. (2015). "Convolutional lstm network: a machine learning approach for precipitation nowcasting," in *Advances in Neural Information Processing Systems*, eds. C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett (New York: Curran Associates Inc), 802–810.

Singh, P., and Manure, A. (2019). *Learn TensorFlow 2.0: Implement Machine Learning and Deep Learning Models with Python*. New York: Apress.

Song, H. F., Yang, G. R., and Wang, X.-J. (2016). Training excitatory-inhibitory recurrent neural networks for cognitive tasks: a simple and flexible framework. *PLoS Comput. Biol.* 12, 1–30. doi: 10.1371/journal.pcbi.1004792

Sussillo, D. (2014). Neural circuits as computational dynamical systems. *Curr. Opini.Neurobiol.* 25, 156–163. doi: 10.1016/j.conb.2014.01.008

Sussillo, D., and Abbott, L. (2009). Generating coherent patterns of activity from chaotic neural networks. *Neuron* 63, 18. doi: 10.1016/j.neuron.2009.07.018

Sussillo, D., and Barak, O. (2013). Opening the black box: low-dimensional dynamics in high-dimensional recurrent neural networks. *Neural Comput.* 25, 626–649. doi: 10.1162/NECO_a_00409

Vyas, S., Golub, M. D., Sussillo, D., and Shenoy, K. V. (2020). Computation through neural population dynamics. *Annual Rev. Neurosci.* 43, 249–275. doi: 10.1146/annurev-neuro-092619-094115

Werbos, P. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE* 78:1550–1560. doi: 10.1109/5.58337

Williams, A. H., Kim, T. H., Wang, F., Vyas, S., Ryu, S. I., Shenoy, K. V., et al. (2018). Unsupervised discovery of demixed, low-dimensional neural dynamics across multiple timescales through tensor component analysis. *Neuron* 98, 1099–1115. doi: 10.1016/j.neuron.2018.05.015

Yang, G. R., and Wang, X.-J. (2020). Artificial neural networks for neuroscientists: a primer. *Neuron* 107, 1048–1070. doi: 10.1016/j.neuron.2020.09.005

Yang, S., and Chen, B. (2023a). Effective surrogate gradient learning with high-order information bottleneck for spike-based machine intelligence. *IEEE Trans. Neural Netw. Learn Syst.* 1–15.

Yang, S., and Chen, B. (2023b). Snib: Improving spike-based machine learning using nonlinear information bottleneck. *IEEE Trans. Syst. Man, Cybernet.: Syst.* 53, 7852–7863. doi: 10.1109/TSMC.2023.3300318

Yang, S., Linares-Barranco, B., and Chen, B. (2022a). Heterogeneous ensemble-based spike-driven few-shot online learning. *Front. Neurosci.* 16, 850932. doi: 10.3389/fnins.2022.850932

Yang, S., Pang, Y., Wang, H., Lei, T., Pan, J., Wang, J., et al. (2023). Spike-driven multi-scale learning with hybrid mechanisms of spiking dendrites. *Neurocomputing* 542, 126240. doi: 10.1016/j.neucom.2023.126240

Yang, S., Tan, J., and Chen, B. (2022b). Robust spike-based continual meta-learning improved by restricted minimum error entropy criterion. *Entropy* 24, 4. doi: 10.3390/e24040455

Zhang, X., Liu, S., and Chen, Z. S. (2021). A geometric framework for understanding dynamic information integration in context-dependent computation. *iScience* 24, 8. doi: 10.1016/j.isci.2021.102919

Zhou, Q., Jin, T., and Zhao, H. (2009). Correlation between eigenvalue spectra and dynamics of neural networks. Neural Computation, 21, 2931–2941. doi: 10.1162/neco.2009.12-07-671