RESCIENCE C

Replication / Computational Neuroscience

[Re] A Reservoir Computing Model of Reward-Modulated Motor Learning and Automaticity

Remya Sankar[1,2,3], Nicolas Thou[1], Nicolas P. Rougier[1,2,3] Arthur Leblois[3]

[1]INRIA Bordeaux Sud-Ouest, Bordeaux, France – [2]LaBRI, Université de Bordeaux, Institut Polytechnique de Bordeaux, Centre National de la Recherche Scientifique, UMR 5800, Talence, France – [3]Institut des Maladies Neurodégénératives, Université de Bordeaux, Centre National de la Recherche Scientifique, UMR 5293, Bordeaux, France

Edited by
Benoît Girard

Reviewed by
Daniel Schmid
Robin Gutzen

Received
12 March 2020

Published
22 November 2021

DOI
10.5281/zenodo.5718075

A replication of pyle2019.

1 Introduction

Pyle and Rosenbaum [1] introduce a novel learning algorithm to the reservoir computing framework, which harnesses the dynamics of a recurrently connected network to generate time series. Most existing algorithms are built on fully supervised learning rules (e.g. FORCE [2]), which limits their potential applications, or the more biologically-realistic reinforcement learning techniques (e.g. RMHL [3]) which unfortunately fail to converge on complex spatio-temporal signal generation tasks. Pyle and Rosenbaum [1] use the advantages of these two learning rules, while averting their individual shortcomings, by combining the two algorithms to form the SUPERTREX model. The workings of this model are aligned to the theory of motor learning involving the basal ganglia, from rodent and songbird literature [4]. This hypothesises that a cortical pathway works in tandem with the basal ganglia for motor skill acquisition, wherein the basal ganglia pathway functions as a tutor, providing guiding signals that would ultimately be consolidated in the primary cortical pathway in charge of production of the motor commands [5]. Here, the basal ganglia pathway, which uses reward-modulated exploration based learning, akin to the RMHL algorithm, works in parallel with the cortical pathway, modeled using the fully supervised FORCE algorithm. The SUPERTREX model uses both these pathways in parallel, with the RMHL-based pathway providing the supervisory signal that the FORCE-based pathway requires.
In this article, we provide a modular and user-friendly Python re-implementation of the model presented by Pyle and Rosenbaum [1]. We were able to successfully reproduce the model performance in Python, for two tasks out of the three presented in the original article. For the third task, we were able to do so with limited robustness. We address this by introducing some modifications, and discuss how their inclusion vastly improves the robustness as well as scalability of the model.

Terminology –

- Original scripts: the MATLAB scripts used by the authors to produce the results presented in [1].
- Python adaptation: our Python adaptation of the original MATLAB scripts.
- Python re-implementation: an improved version of our Python adaptation.

Correspondence should be addressed to Remya Sankar (Remya.Sankar@inria.fr)
The authors have declared that no competing interests exists.
Code is available at https://github.com/rsankar9/Reimplementation-SUPERTREX/releases/tag/v3.0 – DOI https://doi.org/10.5281/zenodo.4596425.
Open peer review is available at https://github.com/ReScience/submissions/issues/50.

1.1 Framework

Pyle and Rosenbaum [1] proposes a model for sensorimotor learning using the framework of reservoir computing. The model is based on two existing reservoir computing techniques: FORCE and RMHL.

$$\tau \frac{d\mathbf{x}}{dt} = -\mathbf{x} + J\mathbf{r} + Q\mathbf{z} \tag{1}$$

$$\mathbf{r} = \tanh(\mathbf{x}) + \epsilon \tag{2}$$

FORCE (or first-order reduced and controlled error) is a fully supervised learning rule, which is widely used within the reservoir computing framework [2]. A recurrently connected reservoir, composed of rate-coded neurons is trained to produce a target time series by modifying the readout weights between the reservoir and the output layer (Eq 3, 4). The output, in turn, interacts with the reservoir by providing feedback (Eq 1, 2). FORCE can accurately generate complex dynamical target time-series. However, the model must have explicit knowledge of the target function, as FORCE requires a fully supervisory signal of the correct output in order to compute the error during training.

$$\mathbf{z}_1 = W_1\mathbf{r} \tag{3}$$

$$\tau_{w_1} \frac{dW_1}{dt} = -\mathbf{e}\mathbf{r}^T P \tag{4}$$

RMHL (or Reward-Modulated Hebbian Learning) is built on the concept of reinforcement learning, and uses only a scalar error signal indicating reward, allowing it to be applicable in a wider range of scenarios than FORCE [3]. RMHL introduces perturbations in the performance of the model, and uses the information gained from this exploration to find the target (Eq 5, 6). This is akin to dopamine-dependent Hebbian learning in the basal ganglia. However, RMHL fails to converge to an accurate solution on several complex tasks. Moreover, it has been observed in songbirds that while the basal ganglia provides a tutor signal in the early stages, learning is eventually consolidated in a parallel cortical pathway, which is primarily responsible for motor activity [5]. RMHL cannot account for such empirical observations.

$$\mathbf{z}_2 = W_2\mathbf{r} + \Psi(e)\boldsymbol{\eta} \tag{5}$$

$$\tau_{w_2} \frac{dW_2}{dt} = \Phi(\hat{e})\hat{\mathbf{z}}\mathbf{r}^T \tag{6}$$

SUPERTREX (Supervised Learning Trained by Reward Exploration), the model proposed by the authors, tries to merge the advantages of both of these algorithms by combining both models. The more wide-ranged applicability of RMHL, owing to its usage of a one dimensional error signal, is used to train the model, while the superior maintenance ability of FORCE is recruited to consolidate the tutoring of the RMHL pathway. This could also potentially support the empirical evidence showing the basal ganglia and cortical pathways working in tandem for motor skill acquisition, discussed above. Thus, the SUPERTREX model consists of two parallel pathways, one based on RMHL (exploratory) and one based on FORCE (mastery), each consisting of its own set of weights. The mastery pathway uses the output of the exploratory pathway as its supervisory signal (Eq 7, 8, 9).

$$\mathbf{z} = \mathbf{z}_1 + \mathbf{z}_2 \tag{7}$$

$$\tau_{w1}\frac{dW_1}{dt} = (\mathbf{z} - \mathbf{z}_1)\,\mathbf{r}^T P \tag{8}$$

$$\tau_{w2}\frac{dW_2}{dt} = \Phi(\hat{e})\hat{\mathbf{z}}\mathbf{r}^T \tag{9}$$

where $\mathbf{x}$ denotes the reservoir dynamics, J the recurrent connectivity matrix, Q the feedback weights and $\mathbf{r}$ the reservoir activity. The output $\mathbf{z}$ of the SUPERTREX model is the combination of the outputs $\mathbf{z_1}$ and $\mathbf{z_2}$ of the FORCE and RMHL pathways, respectively. W_1 and W_2 denote the readout weights of the two pathways, respectively, e denotes the squared distance between the output and the target trajectory, τ is the corresponding timescales for learning, η is the exploratory noise, ϵ is a small noise term and P is a running estimate of the inverse of the correlation matrix of rates. Ψ and Φ are two sublinear functions that serve to damp runaway oscillations during learning and control weight update, respectively. $\hat{x}$ is a high-pass filtered version of x, which represents the recent changes in x.

1.2 Task

The authors test the SUPERTREX model on three motor tasks, with increasing difficulty, and compare its performance to those of FORCE and RMHL. The target of each task is to learn to produce a given spatio-temporal signal, under different constraints. Task 1 tests the performance of the model when the target output is known. This task requires the spatio-temporal signal to be produced directly by the model. Thus, the error signal is a direct indicator of the change required in the output of the model, i.e. fully supervisory. Task 2 and Task 3 use the paradigm of exploration by a multi-segmented arm, pivoted at a point.

Task 2 tests the performance of the model when the target output is unknown, and only an indirect error signal is provided. The task requires the angles between the arm segments to be generated by the model, which would in turn produce the trajectory of the target spatio-temporal signal. In this case, the error signal is not a direct indicator of the change required in the output of the model. A non-linear inverse transformation of the trajectory would be required to compute the desired angles between the arm segments. It is, thus, not a fully supervisory signal, but simply a reinforcement signal. In Task 3, the movement of the arm segments are penalised variably. This creates the need to choose one from multiple candidate solutions by optimising the cost of changing the angles between the arm segments.

The simulation for each task includes a training phase and a testing phase. In the training phase, for ten periods, the time-series is generated by the model while the weights are being updated according to the current error feedback. After this, in the testing phase (lasting five periods), the readout weights are frozen and the time-series is generated using these frozen weights, without any further feedback-based update. In the SUPERTREX model, the exploratory pathway is also deactivated. It is also worth noting that the authors use teacher-forcing in the testing phase, which considerably improves the model performance by limiting the dependence on the stability of the learned solution (refer to Section "State information provides stability of learned output" in [1]).

Disclaimer — Pyle and Rosenbaum [1] proceed to test the model under variations of the above tasks, including disrupted learning and with additional state information. However, these variations have not been replicated by us. We only test the performance of the three learning rules on the three tasks, specified above.

2 Comparison with Python Adaptation

In this section, we compare the results presented in the paper [1] with the MATLAB implementation by the authors and our Python adaptation. The original scripts, although not available online, are readily available on request. We present our adaptation of this model in the open source framework Python, which has been built based on the paper and the MATLAB scripts provided by the authors. In contrast to the original scripts, it is modular and is easily modifiable with external json descriptor files. We compare the results presented in the paper, with simulations of the MATLAB scripts, provided by the authors, and also with our adaptation in Python [1].

To validate our Python adaptation, we test the three algorithms on three tasks by simulating them using both the original scripts and our Python adaptation. For each algorithm-task combination, we produce ten simulations with arbitrary seeds initialising the random generator and one additional simulation using the default seed of MATLAB (equivalent to seed 5489 of the numpy random generator). Except for the default seed, the ten arbitrary seeds are different for the Python and MATLAB simulations, and for each algorithm-task combination. Both MATLAB and numpy use the Mersenne Twister pseudo-random number generator [6]. To evaluate the performance of the algorithm, the authors plot the "distance from target", i.e. the square root of the low pass filtered version of the mean squared error, over the progression of the simulation. In order to categorise the model performance as satisfactory or unsatisfactory, we further compute a **deviation metric** by calculating the mean "distance from target" over the testing phase. If this deviation metric is below the threshold of 0.5 (set by visual inspection), the model is said to have satisfactorily learnt and produced the target output.

2.1 Task 1

Here, we compare the simulations of the original scripts and our adaptation for Task 1, using FORCE, RMHL and SUPERTREX, with the results presented in the article. Task 1 is designed to test the performance of these three algorithms when generating a known target output. The objective of this task is to produce a time-series of 2-D coordinates required to traverse a target trajectory, in this case, the parameterized curve of a butterfly. The model is trained to generate an output which closely matches the target function.

The article claims that:

- under the FORCE framework, the target time-series is learned accurately and is maintained in a stable manner during the testing phase (Figure 1a).
- under the RMHL framework, the target time-series is generated accurately during the training phase, however is not maintained perfectly during the testing phase (Figure 1b).
- under the SUPERTREX framework, the target time-series is learned accurately and is also maintained in a stable manner during testing phase, albeit not as well as FORCE (Figure 1c).

We validate these observations with the MATLAB scripts provided by the authors as well as with our Python adaptation. To do so, we run the simulations with the default seed and repeat it ten times with different (arbitrarily chosen) seeds initialising the random number generator.
We observe that:

[1] In Figures 1- 4, the results presented in the paper have been reused in the column titled "original".

- under the FORCE framework, the target time-series is learned accurately and is maintained in a stable manner during the testing phase, as claimed. The mean deviation over eleven simulations, for both the original scripts (0.003 ± 0.002; n=11) and the Python adaptation (0.004 ± 0.003; n=11) is much lower than the threshold of 0.5 (Figure 1a, 2a).
- under the RMHL framework, the target time-series is generated accurately during the training phase, however is not maintained perfectly during the testing phase, as claimed. The mean deviation, for both the original scripts (0.168 ± 0.038; n=11) and the Python adaptation (0.182 ± 0.046; n=11), is higher than that with FORCE (Figure 1b, 2b).
- under the SUPERTREX framework, the target time-series is learned accurately and is also maintained in a stable manner during testing phase, albeit not as well as FORCE, as claimed. The mean deviation, for both the original scripts (0.006 ± 0.003; n=11) and the Python adaptation (0.006 ± 0.003; n=11), is much better than that for RMHL, but slightly worse than with FORCE (Figure 1c, 2c).

Both the original scripts and the Python adaptation are able to successfully closely reproduce the results presented in the paper for Task 1 (Figure 1,2; Table 1, 2).

2.2 Task 2

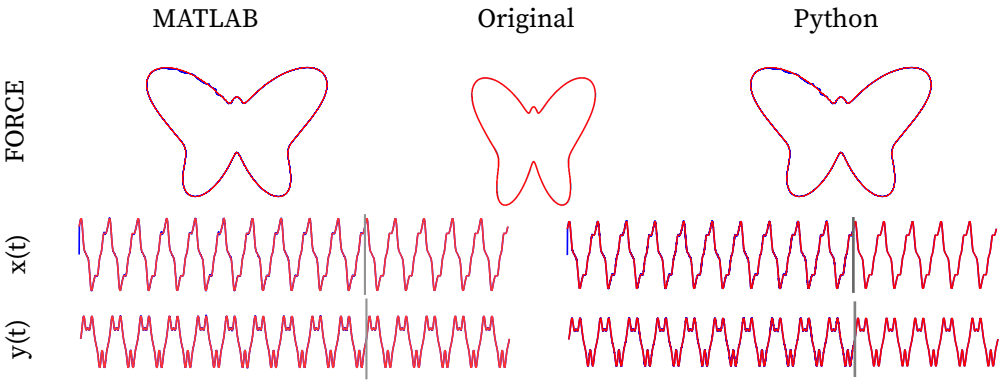
Here, we compare the simulations of the original scripts and our Python adaptation for Task 2, using FORCE, RMHL and SUPERTREX, with the results presented in the article. Task 2 is designed to test the performance of these three algorithms when generating an unknown target from an indirect error signal. Using the paradigm of a pivoted multi-segmented arm, the objective of this task is to produce a time-series by generating the angles between the arm segments. Motor output does not control the position of the end-effector of the arm, but instead controls the angles of the arm joints, which are non-linearly related to end-effector position.

The article claims that:

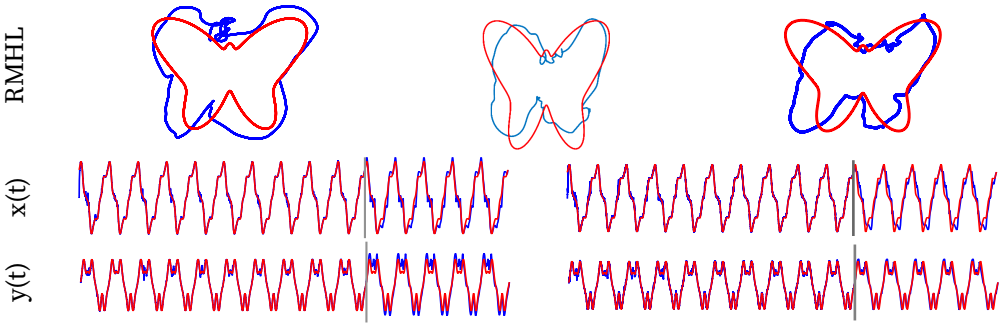
- the FORCE framework cannot be applied to this task, as FORCE requires the exact target to be provided as a supervisory error, which in this case would be the unknown target angles. Since, we do not have this information beforehand, and require the model to derive it, the FORCE framework is inapplicable to this task.
- under the RMHL framework, the target time-series is imitated well by the model during the training phase, however the weights do not converge, and hence, it is unable to maintain the time-series in a stable manner during the testing phase (Figure 3a).
- under the SUPERTREX framework, the target time-series is learned accurately and is also generated in a stable manner, with minor divergences, during testing phase, owing to the contribution of the pathway based on the FORCE algorithm (Figure 3b).

We verify these observations with the MATLAB scripts provided by the authors as well as with our Python adaptation. To do so, we run the simulations with the default seed of MATLAB and re-simulate it with ten arbitrary seeds initialising the random number generator. We do not modify any task conditions or model hyper-parameters.
We observe that:

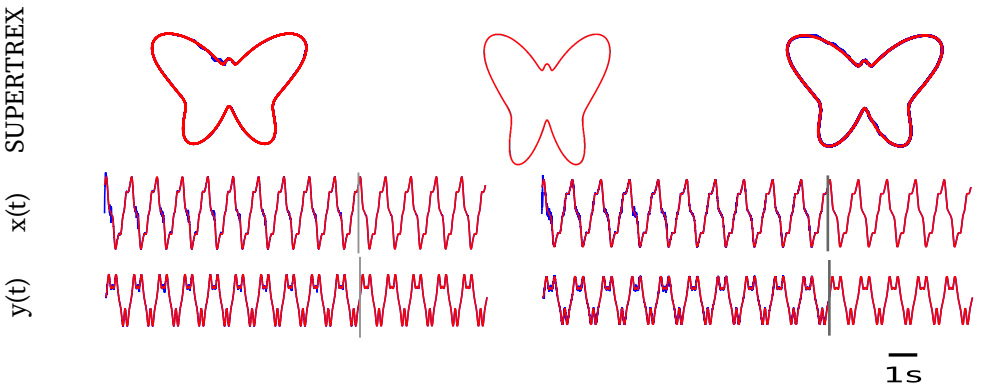
- indeed, the FORCE framework is inapplicable to this task.



(a) Results for Task 1 with the FORCE algorithm. The target time-series is learned accurately during the training phase and is maintained in a stable manner during the testing phase, in both implementations, as presented in [1].

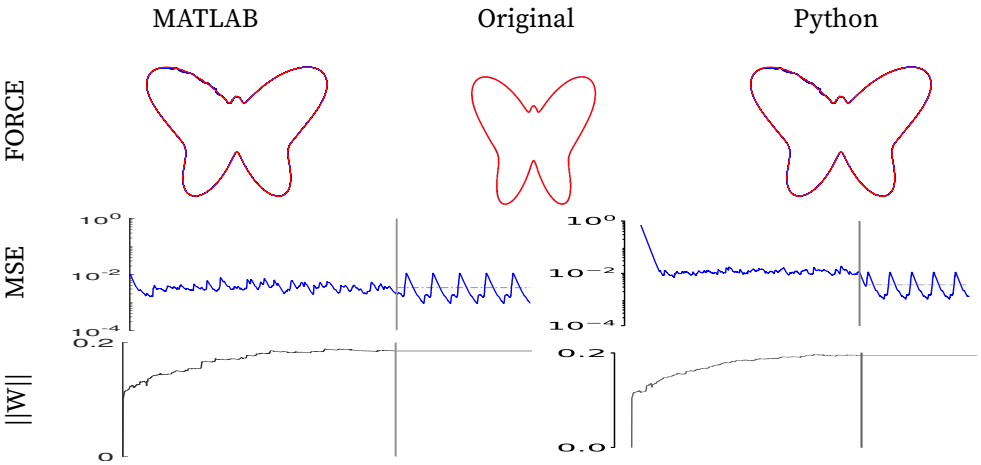


(b) Results for Task 1 with the RMHL algorithm. The target time-series is learned accurately during the training phase, though not maintained perfectly during the testing phase, in both implementations, as presented in [1].

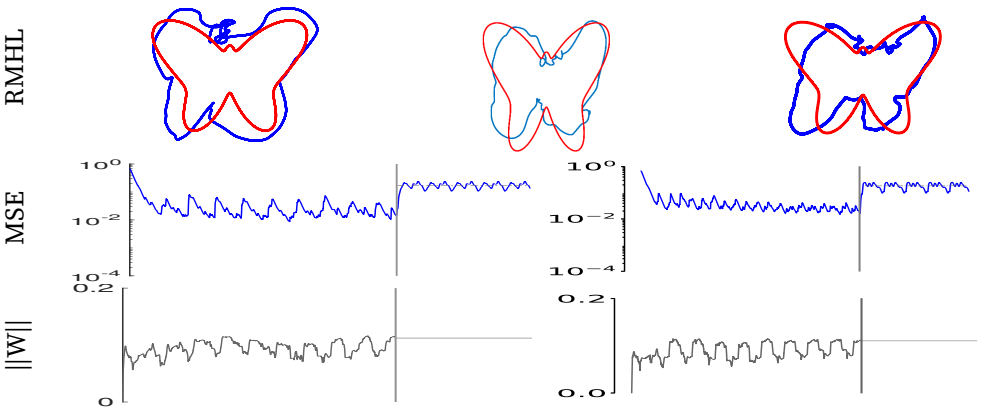


(c) Results for Task 1 with the SUPERTREX algorithm. The target time-series is learned accurately during the training phase, and is also maintained in a stable manner during testing phase, albeit not as well as FORCE, in both implementations, as presented in [1].

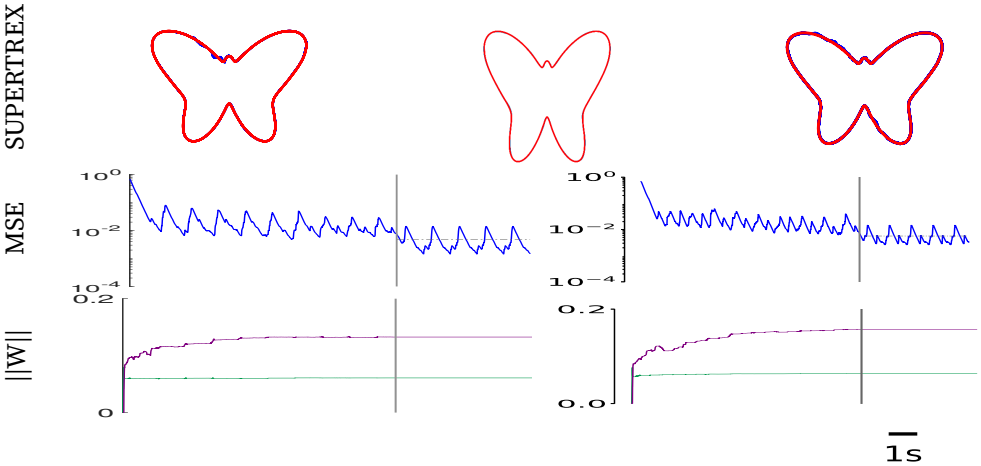
Figure 1. Comparison of the performances of the MATLAB scripts (left column) and the Python adaptation (right column) with the results presented in the original article (center column), for the three learning algorithms on Task 1 [1]. All simulations shown here use the MATLAB default (5489) as the seed for the random number generator. In each subfigure, the top row shows the target trajectory (red) with the trajectory generated by the model (blue) throughout the test phase. The second row shows the time-series (blue) generated by the model (x and y coordinates, in this case) along with the target time-series (red). The grey vertical line marks the separation of the training and testing phase.



(a) Results for Task 1 with the FORCE algorithm. The target time-series is learned accurately during the training phase and is maintained in a stable manner during the testing phase, as presented in [1].



(b) Results for Task 1 with the RMHL algorithm. The target time-series is learned accurately during the training phase, though not maintained perfectly during the testing phase, as presented in [1].



(c) Results for Task 1 with the SUPERTREX algorithm. The target time-series is learned accurately during the training phase, and is also maintained in a stable manner during testing phase, albeit not as well as FORCE, as presented in [1].

Figure 2. Comparison of the performances of the MATLAB scripts (left column) and the Python adaptation (right column) with the results presented in the original article (center column), for the three learning algorithms on Task 1 [1]. All simulations shown here use the MATLAB default (5489) as the seed for the random number generator. In each subfigure, the top row shows the target trajectory (red) with the trajectory generated by the model (blue) throughout the test phase. The second row shows the error metric (blue) over the simulation (x and y coordinates, in this case), using the log scale for the y axis. The bottom row shows the progression of the corresponding weight matrices (SUPERTREX: W_1 in purple; W_2, in green). The horizontal grey line, in the test phase, indicates the deviation metric.

- under the RMHL framework, the target time-series is imitated well by the model during the training phase, however the weights do not converge, and hence, it is unable to maintain the time-series in a stable manner during the testing phase. The mean deviation over eleven simulations, for both the original scripts (0.759 ± 0.284; n=11) and the Python adaptation (0.814 ± 0.288; n=11) is higher than the threshold of 0.5 (Figure 3a).
- under the SUPERTREX framework, the target time-series is learned accurately and is also generated in a stable manner, with minor divergences, during testing phase, owing to the contribution of the pathway based on the FORCE algorithm. The mean deviation over eleven simulations, for both the original scripts (0.011 ± 0.003; n=11) and the Python adaptation (0.012 ± 0.005; n=11) is below the threshold of 0.5 and much lower than that with RMHL (Figure 3b).

The MATLAB scripts provided by the authors and the Python adaptation are able to successfully closely reproduce the results presented for Task 2 in the paper, with the default seed as well as with the 10 arbitrary seeds (Figure 3; Table 1, 2).

2.3 Task 3

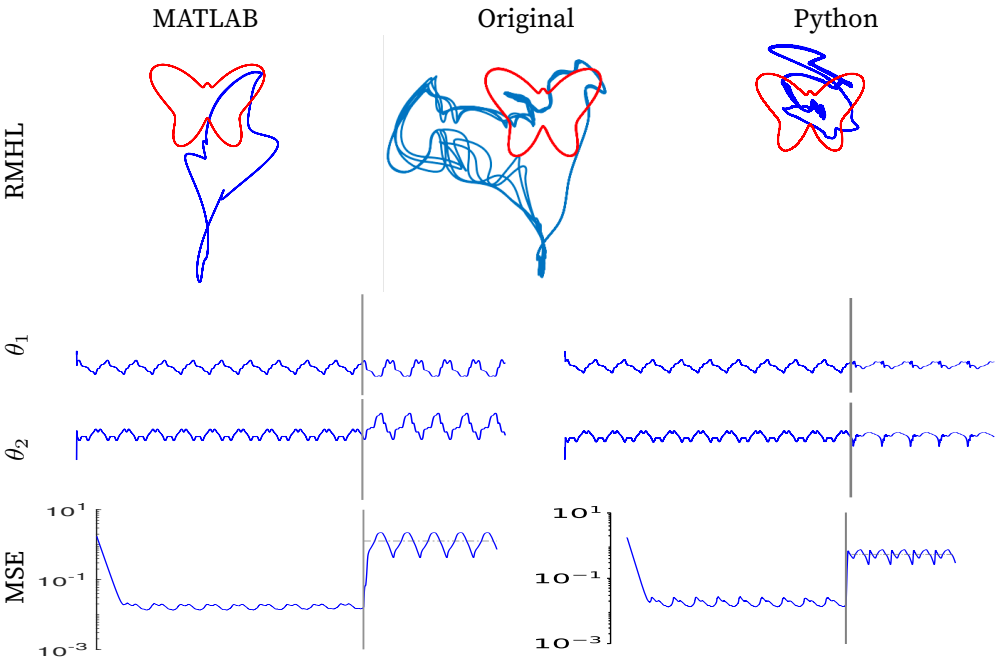
Here, we compare the performance of the MATLAB scripts and our Python adaptation on Task 3, for the three algorithms, with the results presented in the article. Task 3 is an extension of Task 2, designed to test the constraint optimisation ability of these three algorithms when generating an unknown target from an indirect error signal. Using the paradigm of a pivoted multi-segmented arm, the objective of this task is to produce a time-series by generating the angles between the arm segments, while also optimising the movement cost of each arm segment. Hence, the arm is required to traverse the butterfly, while carefully choosing the segment to rotate, in order to minimise the movement cost of its segments. Post the training phase, the readout weights are frozen and in the SUPERTREX model, the exploratory pathway is deactivated.

The article claims that:

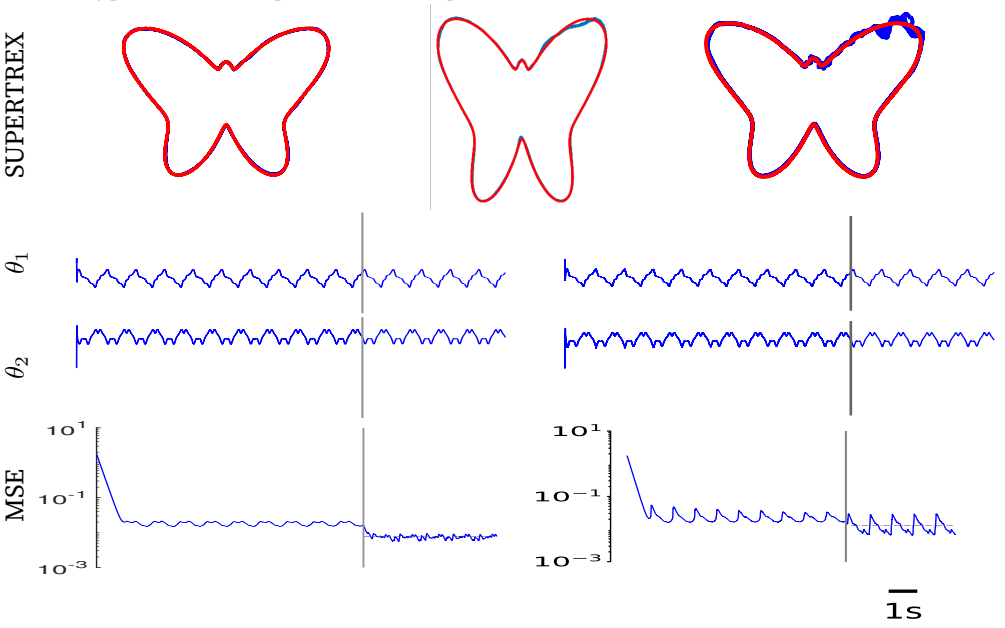
- FORCE can't be applied to this task, as explained for Task 2.
- under the RMHL framework, the target time-series is imitated well by the model during the training phase, however the weights do not converge, and hence, it poorly maintains the time-series during the testing phase (Figure 4a).
- under the SUPERTREX framework, the performance is much better than RMHL. The target time-series is learned accurately and is also generated with minor divergences, during testing phase (Figure 4b).

We verify these observations with the MATLAB scripts provided by the authors as well as with our Python adaptation. To do so, we run the simulations with the default seed of MATLAB and re-simulate it with ten arbitrary seeds initialising the random number generator.
We observe that:

- indeed, the FORCE framework is inapplicable to this task, as claimed.
- under the RMHL framework, the target time-series is imitated well by the model during the training phase, however the weights do not converge, and hence, it poorly maintains the time-series during the testing phase, as claimed. The mean deviation over eleven simulations, for both the original scripts (0.850 ± 0.313; n=11) and the Python adaptation (0.658 ± 0.216; n=11) is higher than the threshold of 0.5. All eleven simulations with different seeds did not generate the target output in a satisfactory manner (i.e. deviation > 0.5 for 11/11 seeds) (Figure 4a).



(a) Results for Task 2 with the RMHL algorithm. The target time-series is imitated well by the model during the training phase (not shown), however, it is unable to maintain the time-series in a stable manner during the testing phase, in both implementations, as presented in [1].



(b) Results for Task 2 with the SUPERTREX algorithm. The target time-series is learned accurately during the training phase, and is also maintained in a stable manner, during the testing phase, in both implementations, as presented in [1].

Figure 3. Comparison of the performances of original scripts (left column) and Python adaptation (right column) with the results presented in the original article (center column), for the RMHL and SUPERTREX, on Task 2 [1]. All simulations shown here use the MATLAB default (5489) as the seed for the random number generator. In each subfigure, the top row shows the target trajectory (red) with the trajectory generated by the algorithm (blue) throughout the test phase. The second row shows the time-series (blue) generated by the model (joint angles (θ_i), in this case). The bottom row shows the distance from target metric (blue) over the simulation (x and y coordinates, in this case), using the log scale for the y axis. The horizontal grey line, in the test phase, indicates the deviation metric. The grey vertical line marks the separation of the training and testing phase.

		MATLAB			Python adaptation			Python re-implementation		
Task	Model	Mean	Median	Std	Mean	Median	Std	Mean	Median	Std
#1	FORCE	0.003	0.002	0.002	0.004	0.003	0.003	0.003	0.002	0.003
	RMHL	0.168	0.165	0.038	0.182	0.182	0.046	0.201	0.203	0.053
	ST	0.006	0.004	0.003	0.006	0.005	0.003	0.004	0.003	0.003
#2	RMHL	0.759	0.740	0.284	0.814	0.799	0.288	0.697	0.681	0.263
	ST	0.011	0.010	0.003	0.012	0.011	0.005	0.010	0.009	0.004
#3	RMHL	0.850	0.794	0.313	0.658	0.647	0.216	0.849	0.794	0.360
	ST	0.881	0.845	0.224	0.837	0.827	0.241	0.140	0.116	0.071
#2'	RMHL	0.846	0.807	0.299	0.738	0.713	0.256	0.839	0.794	0.310
	ST	0.016	0.015	0.007	0.009	0.008	0.003	0.067	0.062	0.035

Table 1. Deviation metric showing the performance of the original MATLAB scripts, Python adaptation and Python re-implementation on different tasks. Each variant is simulated with the default seed (5489) and ten additional seeds. The mean, median and standard deviation of the deviation metric over these eleven simulations are tabulated here. Note that for task #2', the SUPERTREX statistics have been computed using only 2 simulations, for the original MATLAB scripts and Python adaptation. (ST: SUPERTREX; #2': 3 segment variant of Task 2)

		MATLAB		Python adaptation		Python re-implementation	
Task	Model	Satisfactory	Total	Satisfactory	Total	Satisfactory	Total
#1	FORCE	11	11	11	11	11	11
	RMHL	11	11	11	11	11	11
	ST	11	11	11	11	11	11
#2	RMHL	0	11	0	11	0	11
	ST	11	11	11	11	11	11
#3	RMHL	1	11	2	11	0	11
	ST	5	11	4	11	10	11
#2'	RMHL	0	11	1	11	0	11
	ST	2	2	2	2	11	11

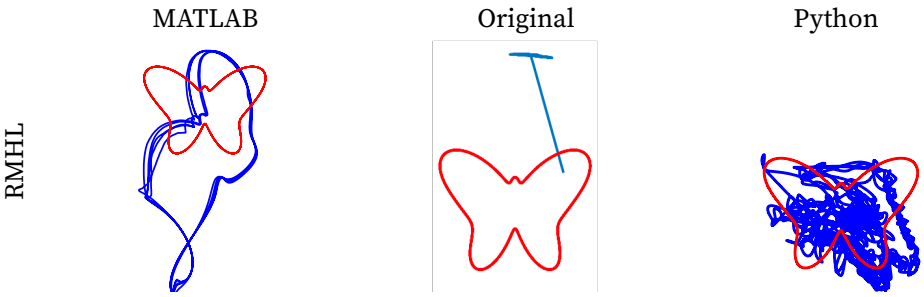
Table 2. The proportion of model simulations categorised as having satisfactory performance. Each variant is simulated with the default seed (5489) and ten additional seeds. Number of satisfactory simulations refers to the number of simulations that were below the threshold (0.5) for the deviation metric. The total number of simulations refer to the number of simulations which successfully reached completion, without the weights growing exponentially. (ST: SUPERTREX; 2': 3 segment variant of Task 2)

- under the SUPERTREX framework, the performance is not much better than RMHL, contrary to the article's claim. The target time-series is not generated in a satisfactory manner, during the testing phase, for more than 50% of the tested simulations (Original scripts: 6/11 and Python adaptation: 7/11). The mean deviation over eleven simulations, for both the original scripts (0.881 ± 0.224; n=11) and the Python adaptation (0.837 ± 0.241; n=11) is above the threshold of 0.5 and comparable with that of RMHL (Figure 4b).

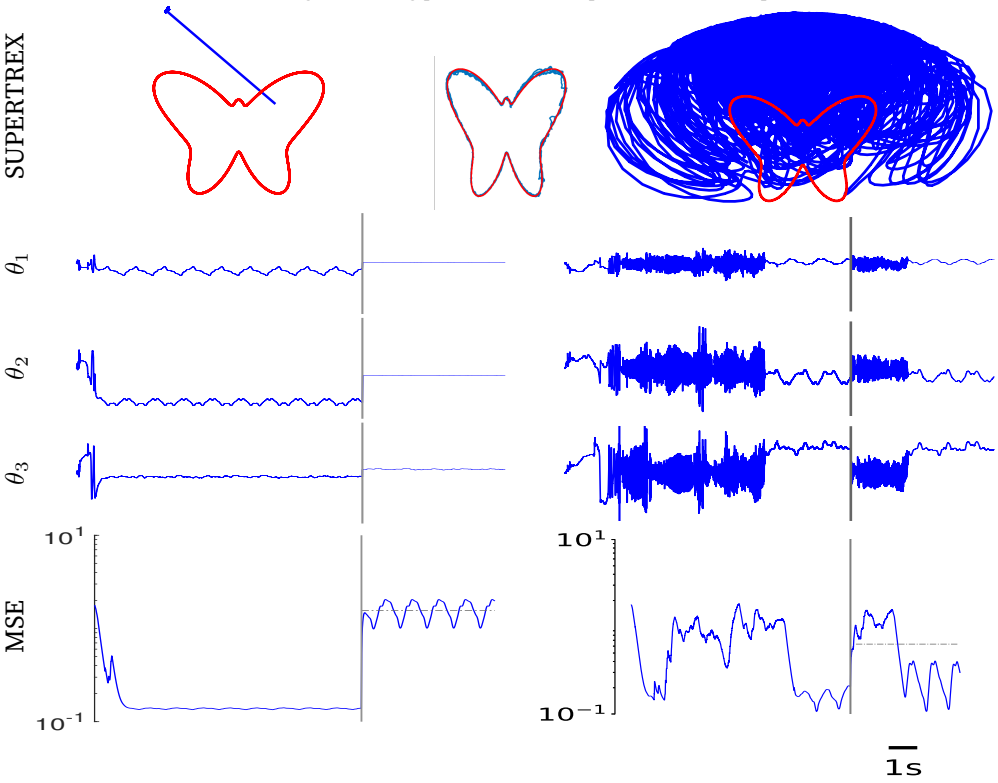
The original scripts and the Python adaptation are able to successfully reproduce the results presented in the paper with the default seed as well as with the 10 arbitrary seeds for the RMHL algorithm, but not for the SUPERTREX algorithm (Figure 3; Table 1, 2).

3 Modification

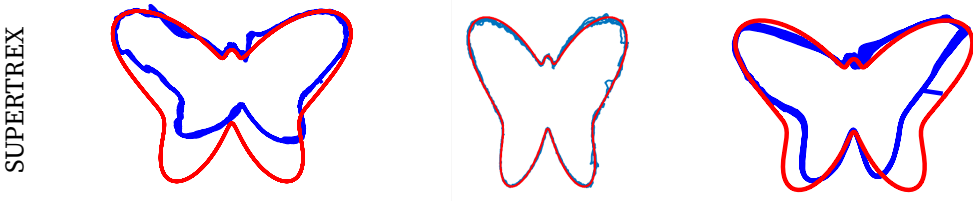
The Python adaptation is a close adaptation of the original MATLAB scripts provided by the authors. However, on simulating their performance on the three tasks, we observed that while, for the first two tasks, the models performed as described in Pyle and Rosenbaum [1], the performance of the SUPERTREX algorithm on Task 3 was not consistent, and was dependent on the seed used for the random number generator. On inspecting



(a) Results for Task 3 with the RMHL algorithm, using the default seed (5489) for the random number generator. The target trajectory is imitated well by the model during the training phase (not shown), however, it poorly maintains the time-series during the testing phase, in both implementations, as presented in [1].



(b) Results for Task 3 with the SUPERTREX algorithm using the default seed (5489) for the random number generator. The target time-series is learned accurately during the training phase, but is not maintained during the testing phase, in both implementations, in contrast to the results presented in [1].



(c) Results for Task 3 with the SUPERTREX algorithm using different implementations (MATLAB, left and Python adaptation, right) and different seeds (295728336, left and 5624282, right) for the random number generator. The target trajectory is learned accurately during the training phase, and is also maintained in a stable manner, with slight divergences (Deviation: 295728336: 0.215 ± 0.073; 5624282: 0.190 ± 0.054) , during the testing phase, in both implementations, similar to the results presented in [1].

Figure 4. Comparison of the performances of original scripts (left column) and Python adaptation (right column) with the results presented in the original article (center column), for RMHL and SUPERTREX, on Task 3 [1]. Each subfigure shows the target trajectory (red) with the trajectory generated by the algorithm (blue) throughout the test phase. In the second subfigure, the the middle rows show the time-series (blue) generated by the model (joint angles (θ_i), in this case). The bottom row shows the distance from target metric (blue) over the simulation (x and y coordinates, in this case), using the log scale for the y axis. The horizontal grey line, in the test phase, indicates the deviation metric. The grey vertical line marks the separation of the training and testing phase.

further, we notice that this was, in some cases, due to the uncontrolled exponential increase in the readout weights.

To look into the robustness of the implementations further, we test the performance of the RMHL and SUPERTREX algorithms on Task 2 with certain modifications to the task parameters, specifically, the number of arm segments and the length of the arm segments. It would be expected for the behaviour to be comparable with the performance on the original task performance, or undergo a gradual decline. We test Task 2 on the arm parameters, which were used in Task 3, i.e. by increasing the number of arm segments from two to three and changing the length of each arm segment. We observe that RMHL performance is comparable to the original Task 2, wherein the time series is generated during the training phase, but is not maintained beyond (Original scripts: 0.846 ± 0.299, Python adaptation: 0.738 ± 0.256; n=11). On the other hand, simulations of the SUPERTREX model, with 2 out of 11 seeds, were able to produce the target output satisfactorily (Original scripts: 0.016 ± 0.007, Python adaptation: 0.009 ± 0.003; n=2) (Figure 5). However, in simulations with 9 out of 11 seeds, the weights increase exponentially, rendering the simulation unable to progress in a meaningful manner (Table 1, 2).

In order to improve the model performance, make the model more scalable in terms of task parameters, and also more robust (as seen in Task 3, with respect to reproducibility with different random seeds), we introduce two minor alterations.

1. We introduce a compensation factor to the update of the readout weights in the exploratory pathway, inversely proportional to the number of segments. Specifically, when the number of segments is greater than two, we multiply the weight update by $0.1/n\_segs$ for Task 2 and by $0.5/n\_segs$ for Task 3.

2. The SUPERTREX model transfers the information from the exploratory pathway to the mastery pathway, only if the error is consistently below a certain threshold. In the original scripts, this threshold is set at 1.5e-3 for Task 1 and Task 2, while at 1.5e-2 for Task 3. We change the transfer threshold for Task 2 from 1.5e-3 to 1.5e-2.

These slight modifications address the shortcomings we encountered earlier with the performance of SUPERTREX in Task 2 and 3. Alteration #1, by including a compensation factor for the change in number of arm segments, prevents the weights from increasing exponentially, and lets the simulation proceed in a meaningful manner. Alteration #2, by increasing the error threshold governing the transfer of information to the mastery pathway, makes the model more tolerant of fluctuations, while continuing to explore and learn a good solution. Although this does not lead to a critical change for Task 1 (Original scripts: 0.006 ± 0.003, n=11; Modified Python re-implementation: 0.004 ± 0.003, n=11) and Task 2 (Original scripts: 0.011 ± 0.003, n=11; Modified Python re-implementation: 0.010 ± 0.004, n=11), this alteration improves the performance of SUPERTREX on Task 3 (Original scripts: 0.881 ± 0.224, n=11; Modified Python re-implementation: 0.140 ± 0.071, n=11). Simulations with 10 out of 11 seeds had satisfactory performance (Deviation < 0.5), compared to 6 out of 11 simulations for the original scripts. Further, it unlocks the potential for the model to be more scalable. We find that with these alterations, on merely increasing the number of time-steps per training cycle and with no further fine tuning of hyper-parameters, the model is able to proceed without an exponential increase in weights over a wider range of task parameters. For instance, on adding surplus segments with length 0.1 each, the model is able to perform in a satisfactory manner, for most cases, with up to 50 arm segments (Table 3, Figure 6). Better accuracy can be achieved by further fine tuning of the hyper-parameters.

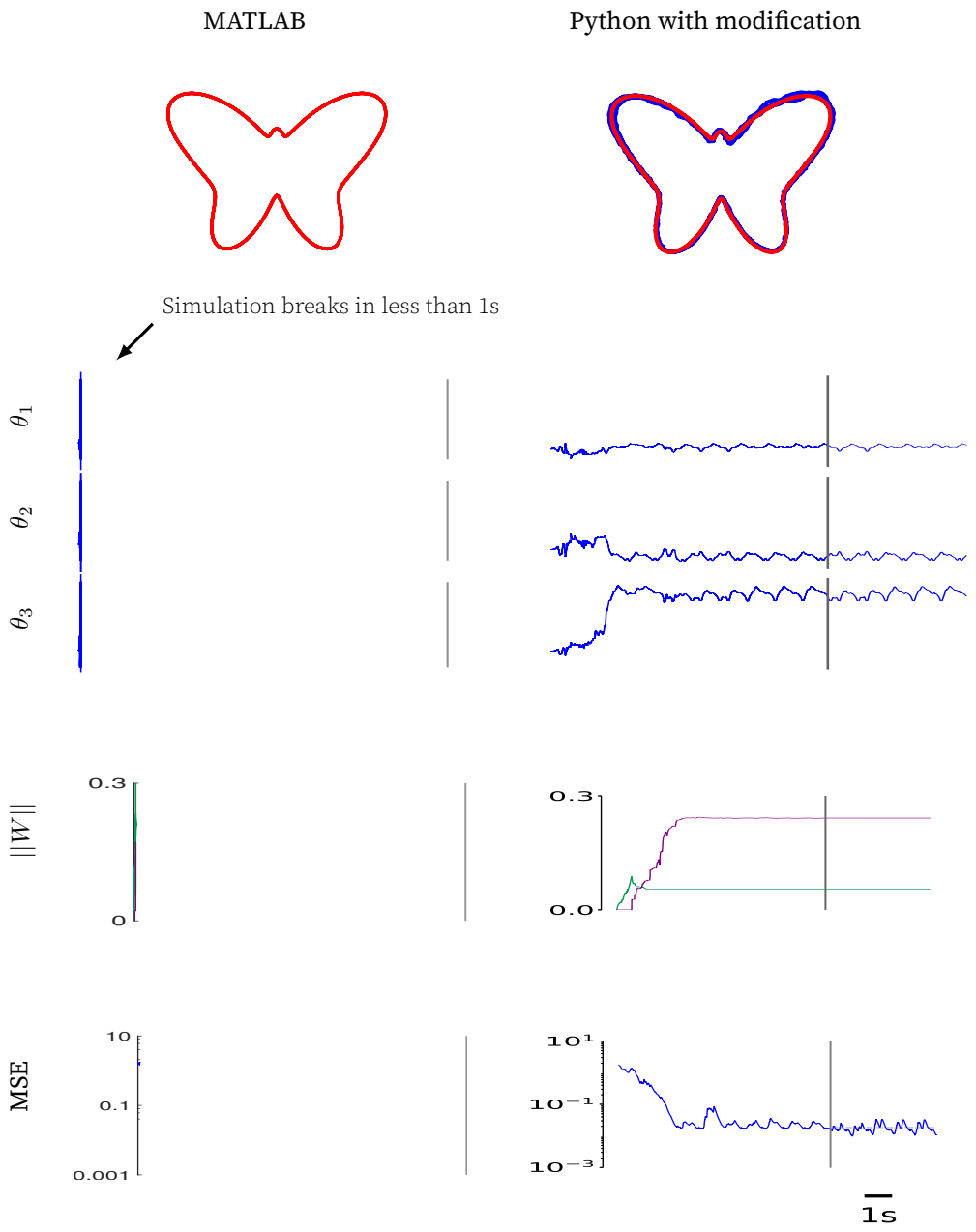


Figure 5. Robustness of the SUPERTREX model on a Task 2 variant. The performance of the original scripts (left column) and modified Python re-implementation (right column) is tested for the SUPERTREX learning algorithm on a variant of Task 2 with increased number of arm segments (lengths: 1.8, 1.2, 0.6). The top panel shows the target trajectory (red) with the trajectory generated by the algorithm (blue) throughout the test phase. The next three rows show the time-series (blue) generated by the model (joint angles (θ_i), in this case). The fourth row shows the progression of the norm of the weight matrix (W_1 in purple; W_2, in green). The bottom row shows the distance from target metric (blue) over the simulation, using the log scale for the y axis. The horizontal grey line, in the test phase, indicates the deviation metric. The grey vertical line marks the separation of the training and testing phase. Using the MATLAB scripts, the readout weights increase uncontrollably rendering the model unable to learn. The Python re-implementation, using a compensation factor to harness the weight update, is able to learn and converge to produce the target time-series.

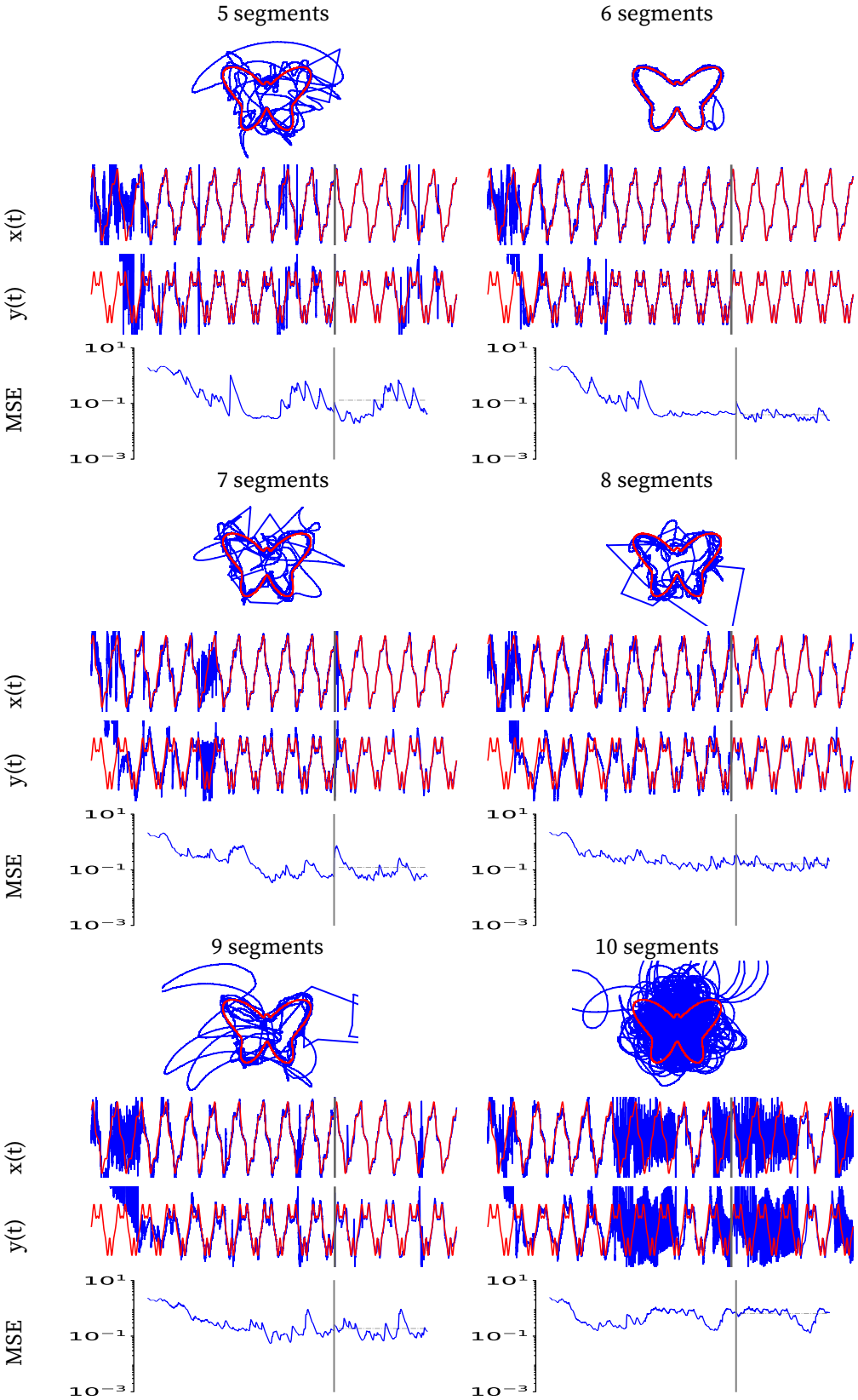


Figure 6. Scalability of the performance of the modified Python re-implementation using the SUPERTREX algorithm on Task 2. The lengths of the arm segments are 1.8, 1.2 and 0.6 for the first three segments (akin to Task 3) and 0.1 for each additional segment. Here, the simulations for Task 2 with 5 to 50 segments are shown, all using the default seed 5489 for the random number generator. (Continued on next page.)

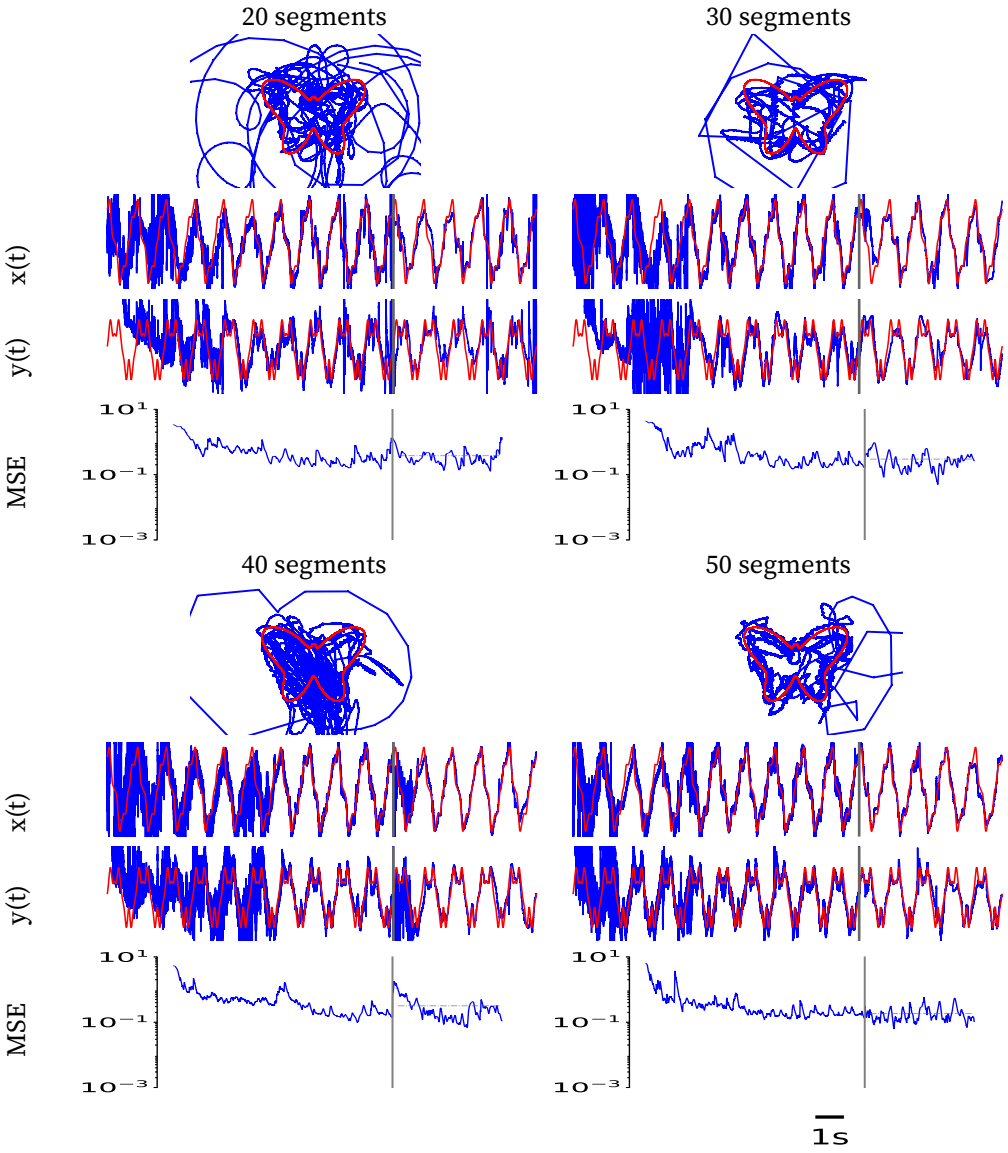


Figure 6. (Continued from previous page.) Scalability of the performance of the modified Python re-implementation using the SUPERTREX algorithm on Task 2. The lengths of the arm segments are 1.8, 1.2 and 0.6 for the first three segments (akin to Task 3) and 0.1 for each additional segment. Here, the simulations for Task 2 with 5 to 50 segments are shown, all using the default seed 5489 for the random number generator. In each subfigure, the top panel shows the produced trajectory, the middle panels show the evolution of the x and y coordinates of the end-effector of the arm (blue) throughout the training and test phase, along with the target coordinates (red). The grey vertical line marks the separation of the training and testing phase. The bottom panel shows the progression of the distance from target metric (blue) over the simulation, using the log scale for the y axis. The horizontal grey line, in the test phase, indicates the deviation metric.

Task 2 variant		Deviation metric		
No. of segments	Time steps	Mean	Median	Standard Deviation
3	10000	0.057	0.032	0.069
4	10000	0.242	0.221	0.136
5	10000	0.141	0.080	0.147
6	10000	0.181	0.160	0.133
7	10000	0.173	0.129	0.130
8	10000	0.253	0.230	0.137
9	10000	0.331	0.297	0.168
10	10000	0.417	0.409	0.151
15	10000	0.538	0.512	0.179
20	15000	0.366	0.324	0.188
30	20000	0.549	0.489	0.236
40	20000	0.372	0.313	0.228
50	30000	0.375	0.283	0.281

Table 3. Deviation metric showing the performance of the modified Python implementation on increasing number of segments for Task 2. Each variant is simulated with the default seed (5489) and ten additional seeds. The mean, median and standard deviation of the deviation metric over these eleven simulations are tabulated here.

4 Discussion

In this article, we discussed the SUPERTREX model presented by Pyle and Rosenbaum [1]. We compared the results presented in the paper, both with the results obtained using the original scripts, and with our modular and user-friendly Python adaptation. Furthermore, we were able to improve the robustness and scalability of the model with two minor alterations.

The Python adaptation strives to be a close adaptation of the original scripts in MATLAB and differs mainly in the method of initialisation of the reservoir connectivity matrix. This is due to the usage of the function `sprandn` in the original scripts, whose internal implementation is not freely available. Figure 7 shows that, on importing initialisation matrix from MATLAB, the exact same results can be obtained in the Python adaptation, as well. Most of the details for the implementation of the models are also described in the paper. Only two necessary details were missing, both concerning the update of the readout weights of the exploratory pathway in the RMHL and SUPERTREX models: One, the inclusion of a crucial learning rate of 0.0005, for Tasks 1-3, and two, an additional compensatory factor of 0.5, for Task 3. There is another discrepancy in the function $psi(x)$ for Task 3. The scripts provided by the authors use a factor of 0.005, whereas the article mentions this factor to be 0.025.

The three algorithms (FORCE, RMHL and SUPERTREX) have been tested on three tasks, presented in Pyle and Rosenbaum [1]. For Task 1 and 2, we verify that the three algorithms function as presented in the paper, and validate that our Python re-implementation produces comparable results. For Task 3, the SUPERTREX model's behaviour is also reproducible, although the performance is dependent on the seed used for the random number generator. Furthermore, we observed that this implementation is quite sensitive to changes in task parameters, such as the number of arms. This was due to the uninhibited increase in the readout weights. We propose the inclusion of a compensation factor for the number of arm segments, which inhibits the growth of the readout weights, and allows the simulation to proceed in a meaningful manner. This considerably improves the robustness and the scalability of the original model.

We conclude that the results presented in the paper are reproducible for two tasks, using the original MATLAB scripts provided by the authors, and also, replicable in Python for all tasks with comparable performance.

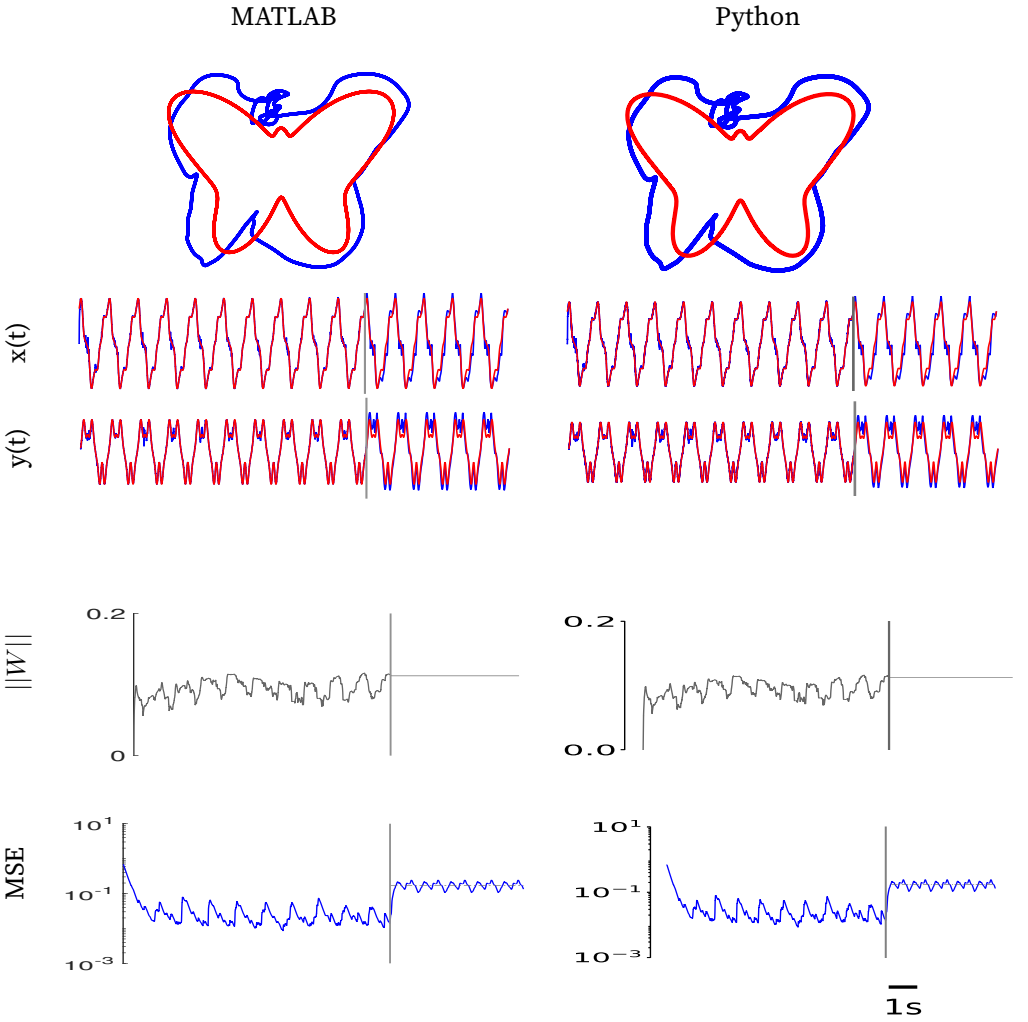


Figure 7. Similarity between the original scripts and the Python adaptation. The performance of the original scripts (left column) and the Python adaptation (right column) is tested for the RMHL learning algorithm on a Task 1. The reservoir connectivity matrix for the Python simulation was initialised using the MATLAB equivalent. Using this initialisation, the progression of the Python simulation is identical to that of the MATLAB simulation. The top panel shows the target trajectory (red) with the trajectory generated by the model (blue) throughout the test phase. The next two rows show the time-series (blue) generated by the model (x and y coordinates, in this case). The third row shows the progression of the norm of the weight matrix. The bottom row shows the distance from target metric (blue) over the simulation, using the log scale for the y axis. The horizontal grey line, in the test phase, indicates the deviation metric. The grey vertical line marks the separation of the training and testing phase.

References

1. R. Pyle and R. Rosenbaum. "A reservoir computing model of reward-modulated motor learning and automaticity." In: **Neural computation** 31.7 (2019), pp. 1430–1461.
2. D. Sussillo and L. F. Abbott. "Generating coherent patterns of activity from chaotic neural networks." In: **Neuron** 63.4 (2009), pp. 544–557.
3. G. M. Hoerzer, R. Legenstein, and W. Maass. "Emergence of complex computational structures from chaotic neural networks through reward-modulated Hebbian learning." In: **Cerebral cortex** 24.3 (2014), pp. 677–690.
4. M. S. Brainard and A. J. Doupe. "What songbirds teach us about learning." In: **Nature** 417.6886 (2002), pp. 351–358.
5. B. P. Ölveczky, T. M. Otchy, J. H. Goldberg, D. Aronov, and M. S. Fee. "Changes in the neural control of a complex motor sequence during learning." In: **Journal of neurophysiology** 106.1 (2011), pp. 386–397.
6. M. Matsumoto and T. Nishimura. "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator." In: **ACM transactions on modeling and computer simulation (TOMACS)** 8.1 (1998), pp. 3–30.