



(51) International Patent Classification:

Not classified

SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

(21) International Application Number:

PCT/US2024/030221

Published:

— without international search report and to be republished upon receipt of that report (Rule 48.2(g))

(22) International Filing Date:

20 May 2024 (20.05.2024)

(25) Filing Language:

English

(26) Publication Language:

English

(30) Priority Data:

63/503,324

19 May 2023 (19.05.2023)

US

(71) Applicants: **NTT RESEARCH, INC.** [US/US]; 940 Stewart Dr., Sunnyvale, California 94085 (US). **PRESIDENT AND FELLOWS OF HARVARD COLLEGE** [US/US]; 17 Quincy Street, Cambridge, Massachusetts 02138 (US). **THE BOARD OF TRUSTEES OF THE LELAND STANFORD JUNIOR UNIVERSITY** [US/US]; Office of the General Counsel Building 170, Third Floor, Main Quad, P.O. Box 20386, Stanford, California 94305-2038 (US).

(72) Inventors: **DINC, Fatih**; 940 Stewart Dr., Sunnyvale, California 94085 (US). **TANAKA, Hidenori**; 940 Stewart Dr., Sunnyvale, California 94085 (US).

(74) Agent: **HOLOUBEK, Charlotte E.**; 1718 E Olive Way, Seattle, Washington 98102 (US).

(81) Designated States (unless otherwise indicated, for every kind of national protection available): AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CV, CZ, DE, DJ, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IQ, IR, IS, IT, JM, JO, JP, KE, KG, KH, KN, KP, KR, KW, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, MG, MK, MN, MU, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, WS, ZA, ZM, ZW.

(84) Designated States (unless otherwise indicated, for every kind of regional protection available): ARIPO (BW, CV, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SC, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, ME, MK, MT, NL, NO, PL, PT, RO, RS, SE,

(54) Title: SYSTEMS AND METHODS OF SCALABLE CONVEX RECURRENT NEURAL NETWORK TRAINING FOR INTERPRETING BRAIN DYNAMICS

(57) Abstract: Computer-implemented methods and systems for training a neural network to interpret neural dynamics, the method comprising obtaining functional connectivity between brain regions and artificial neurons simulated by the neural network, capturing neural activities between biological neurons, processing the captured neural activities to obtain training data for the neural network, and constructing the neural network using the training data, to reproduce neural dynamics between the artificial neurons.

WO 2024/243136 A2

[0001] SYSTEMS AND METHODS OF SCALABLE CONVEX RECURRENT NEURAL NETWORK TRAINING FOR INTERPRETING BRAIN DYNAMICS

[0002] CROSS-REFERENCE TO RELATED APPLICATIONS/PRIORITY

[0003] The present invention claims priority to United States Provisional Patent Application Number 63/503,324 filed May 19, 2023, which is incorporated by reference into the present disclosure as if fully restated herein. Any conflict between the incorporated material and the specific teachings of this disclosure shall be resolved in favor of the latter. Likewise, any conflict between an art-understood definition of a word or phrase and a definition of the word or phrase as specifically taught in this disclosure shall be resolved in favor of the latter.

[0004] BACKGROUND

Understanding how the brain works is a complex problem that researchers so far have attacked from various angles. Despite the many beneficial applications of such understanding, current technologies fail for many reasons. There remains a real and pressing need for the networks to scale to 10000s of neurons and be able to learn and predict in real-time support of brain-machine interfaces.

[0005] SUMMARY

[0006] Wherefore, it is an object of some embodiments of the presently disclosed invention to overcome some or all of above-mentioned shortcomings and drawbacks associated with the current technology.

[0007] The presently disclosed invention relates to computer-implemented methods for training a neural network to interpret neural dynamics, the method comprising obtaining functional connectivity between brain regions and artificial neurons simulated by the neural network, capturing neural activities between biological neurons, processing the captured neural activities to obtain training data for the neural network, and constructing the neural network using the training data to reproduce neural dynamics between the artificial neurons. In some embodiments the neural

network includes a convex solver. In some embodiments the convex solver comprises a convex weighted loss function. In some embodiments, computer-implemented method further comprises optimizing a loss of the convex solver via a fixed-point estimation. In some embodiments, computer-implemented method further comprises predicting a spatial and temporal activity of neural population dynamics. In some embodiments the neural network is configured to extract large-scale functional connectivity between brain regions and/or simulated networks. In some embodiments the neural network is trained to predict the temporal activity of a corresponding biological neuron in the brain. In some embodiments the neural network is configured to uncover attractor structures within the brain dynamics. In some embodiments the neural network is further configured to design external interventions to control neural population dynamics in the brain. In some embodiments the neural network is trained to interpret neural dynamics in real-time to support brain-machine interfaces. In some embodiments wherein the convex solver is further configured to perform optimization updates accelerated on a graphics processing unit (GPU). In some embodiments the convex solver is part of an alternating direction method of multipliers (ADMM) framework. In some embodiments the convex solver is configured to enforce constraints on the neural network during training. In some embodiments the constraints include preventing self-excitation of neurons within the neural network. In some embodiments the convex solver is configured to utilize biological priors for inductive biases during training. In some embodiments the convex solver is configured to handle input noise and conversion noise during the training of the neural network. In some embodiments the convex solver is configured to perform outlier detection during the training process. In some embodiments the convex solver is configured to scale to datasets collected from more than 10,000 neurons for offline processing.

[0008] The presently disclosed invention further relates to non-transitory computer-readable mediums comprising instructions that, when executed by one or more processors, cause a computer system to obtain functional connectivity between brain regions and artificial neurons simulated by the neural network, capture neural activities between biological neurons, process the captured neural activities to obtain training data for the neural

network, and construct the neural network using the training data to reproduce neural dynamics between the artificial neurons.

[0009] The presently disclosed invention further relates to a computer system for training a neural network to interpret neural dynamics, the system comprising one or more processors and a non-transitory computer-readable medium including instructions that, when executed by the one or more processors, cause the computer system to obtain functional connectivity between brain regions and artificial neurons simulated by the neural network, capture neural activities between biological neurons, process the captured neural activities to obtain training data for the neural network, and construct the neural network using the training data to reproduce neural dynamics between the artificial neurons. In some embodiments the neural network is a data constrained recurrent neural network.

[0010] The presently disclosed invention is directed to methods and apparatuses systems and methods that relate to a scalable convex recurrent neural network trained for interpreting brain dynamics are described herein. These systems and methods allow extraction of large-scale functional connectivity between brain regions and/or simulated networks. Embodiments of the present disclosure may include a computer-implemented method for training a neural network to interpret neural dynamics, including obtaining functional connectivity between brain regions and artificial neurons simulated by the neural network, including capturing neural activities between biological neurons. Embodiments may also include processing the neural activities between biological neurons to obtain training data for the neural network.

[0011] Embodiments may also include constructing the neural network using the training data. In some embodiments, the neural network may include a convex solver. Embodiments may also include reproducing neural dynamics between the artificial neurons simulated from the neural network. In some embodiments, the neural network may be a data constrained recurrent neural network.

[0012] In some embodiments, the convex solver includes a convex weighted loss function. In some embodiments, the method may include optimizing a loss of the convex solver via

a fixed-point estimation. In some embodiments, the method may include predicting a spatial and temporal activity of neural population dynamics.

[0013] In some embodiments, the convex solver includes a convex weighted loss function. In some embodiments, the instructions, when executed, cause the one or more processors to further perform optimizing a loss of the convex solver via a fixed-point estimation. In some embodiments, the instructions, when executed, cause the one or more processors to further perform including predicting a spatial and temporal activity of neural population dynamics.

[0014] Embodiments of the present disclosure may also include a non-transitory computer readable medium including instructions to cause the one or more processors to perform obtaining a functional connectivity between brain regions and artificial neurons simulated by the neural network, including capturing neural activities between biological neurons. Embodiments may also include processing the neural activities between biological neurons to obtain training data for the neural network.

[0015] Embodiments may also include constructing the neural network using the training data. In some embodiments, the neural network may include a convex solver. Embodiments may also include reproducing neural dynamics between the artificial neurons simulated from the neural network. In some embodiments, the neural network may be a data constrained recurrent neural network.

[0016] Embodiments of the present disclosure may also include a computer system for training a neural network to interpret neural dynamics, including one or more processors. Embodiments may also include a non-transitory computer readable medium including instructions, when executed by the one or more processors, causing the computer system, to obtain a functional connectivity between brain regions and artificial neurons simulated by the neural network, including capturing neural activities between biological neurons.

[0017] Embodiments may also include processing the neural activities between biological neurons to obtain training data for the neural network. Embodiments may also include constructing the neural network using the training data. In some embodiments, the neural network may include a convex solver. Embodiments may also include

reproducing neural dynamics between the artificial neurons simulated from the neural network. In some embodiments, the neural network may be a data constrained recurrent neural network.

[0018] In some embodiments, the convex solver includes a convex weighted loss function. In some embodiments, the instructions, when executed, cause the one or more processors to further perform optimizing a loss of the convex solver via a fixed-point estimation.

[0019] In some embodiments, the instructions, when executed, cause the one or more processors to further perform the step of predicting a spatial and temporal activity of neural population dynamics.

[0020] The foregoing summary is illustrative only and is not intended to be in any way limiting. Features from any of the disclosed embodiments can be used in combination with one another, without limitation. In addition to the illustrative aspects, embodiments, and features described above, further aspects, embodiments, and features will become apparent by reference to the drawings and the following detailed description.

[0021] Various objects, features, aspects, and advantages of the present invention will become more apparent from the following detailed description of preferred embodiments of the invention, along with the accompanying drawings in which like numerals represent like components. The present invention may address one or more of the problems and deficiencies of the current technology discussed above. However, it is contemplated that the invention may prove useful in addressing other problems and deficiencies in a number of technical areas. Therefore, the claimed invention should not necessarily be construed as limited to addressing any of the particular problems or deficiencies discussed herein.

[0022] BRIEF DESCRIPTION OF THE DRAWINGS

[0023] The accompanying drawings, which are incorporated in and constitute a part of the specification, illustrate various embodiments of the invention and together with the general description of the invention given above and the detailed description of the drawings given below, serve to explain the principles of the invention. It is to be

appreciated that the accompanying drawings are not necessarily to scale since the emphasis is instead placed on illustrating the principles of the invention. The invention will now be described, by way of example, with reference to the accompanying drawings in which:

[0024] Fig. 1 is a schematic representation showing interpreting and controlling brain dynamics with data constrained recurrent neural networks (dRNNs). The inventors present a large-scale functional connectivity task for artificial recurrent neural networks (RNNs), where each artificial neuron is trained to predict the temporal activity of a corresponding biological neuron in the brain. By extracting RNNs with functional connectivity, the inventors aim to achieve three goals in future work: (i) predict neural population dynamics, (ii) uncover attractor structures, and (iii) design external interventions to control neural population dynamics in the brain.

[0025] Fig. 2 is a schematic representation showing CoRNN: Convex and scalable solver for dRNNs via ADMM framework. The flowchart illustrates their CoRNN algorithm, which optimizes the parameters of a recurrent neural network to match the activity of a hidden unit with that of a generator (brain). The loss landscape is convex due to their careful choice of objective function. The algorithm starts by finding a fixed point for initialization and computing the inverse of the Hessian. It then starts a loop that goes through a certain number of iterations. At each iteration, the algorithm predicts the objectives, calculates the prediction error, updates the parameter values, and checks for convergence. The final output is the optimized parameters, which represent the functional connections between the neurons in the recurrent neural network.

[0026] Figs. 3A and 3B are two graphs showing CoRNN achieves the best combination of speed and accuracy, dramatically accelerating convergence. The plot illustrates the relationship between accuracy (correlation between ground truth and inferred weights) and training time, measured in seconds on a log scale. The blue line represents the performance of their newly developed method, which combines fixed-point initialization and CoRNN, and achieves high accuracy while significantly accelerating convergence compared to other methods. Parameters: Fig. 3A: $\alpha = 0.1$, n_{rec}

$= 200$, $T = 3000$, $\epsilon^{\text{conv}} \sim \mathcal{N}(0, 10^{-4})$, $\epsilon^{\text{input}} \sim \mathcal{N}(0, 10^{-2})$ and Fig. 3B: $\alpha = 0.1$, $n_{\text{rec}} = 200$, $T = 3000$, $\epsilon^{\text{conv}} \sim \mathcal{N}(0, 10^{-5})$, $\epsilon^{\text{input}} \sim \mathcal{N}(0, 10^{-1})$. For both, error bars are standard error of the mean (s.e.m.) over 10 randomly initialized networks.

[0027] Figs. 4A and 4B are two graphs showing CoRNN is much more stable and tolerant to variations in hyperparameter choices than FORCE. This plot describes the relationship between accuracy (correlation between ground truth and inferred weights) and the strength of regularization applied to the weights during training. The results show that while the FORCE algorithm, as previously applied to a similar task, is highly sensitive to the choice of hyperparameters, CoRNN maintains high accuracy across scales regardless of the strength of regularization. Fig. 4A with fixed point initialization for all and Fig. 4B with random initialization for Pytorch models and FORCE, whereas CoRNN is fixed-point initialized to define comparable scales. Parameters are the same as Fig. 3A. Error bars are sem.

[0028] Fig. 5 is a graph that shows ablating the fixed-point initialization degrades the convergence speed of the resulting models. The inventors conducted an ablation study to investigate the effectiveness of their proposed fixed-point initialization method. The results show that the CoRNN algorithm with fixed-point initialization (blue) outperforms all other methods, including the vanilla CoRNN without this fixed-point initialization technique. Parameters: $\alpha = 0.1$, $n_{\text{rec}} = 500$, $T = 10000$, $\epsilon^{\text{conv}} \sim \mathcal{N}(0, 10^{-5})$, $\epsilon^{\text{input}} \sim \mathcal{N}(0, 10^{-1})$. Error bars are sem.

[0029] Figs. 6A – 6C are experimental results showing CoRNN can reproduce an RNN performing 3-bit flip flop task through dRNN training. The inventors use CoRNN to reproduce the behavior of a synthetic RNN trained on a k-bit flip-flop task, where the network is given a short input pulse at a random time. If the input pulse is +1, the output of the network must transition to +1 or stay at +1. If the input pulse is -1, the network's output must transition to -1 or stay at -1, ignoring input pulses that do not match. Through dRNN training, CoRNN is able to accurately reproduce (Fig. 6A) the inputs and outputs of the task-trained RNN, (Fig. 6B) spatiotemporal pattern of the neural population dynamics, and (Fig. 6C) the inputs and recurrent weights of the synthetic RNN. Parameters: $\alpha = 0.9$, $n_{\text{rec}} = 1000$, $T = 200$ trials (100 data points per

trial), $\epsilon^{\text{conv}} \sim N(0, 10^{-3})$, $\epsilon^{\text{input}} \sim N(0, 10^{-2})$.

[0030] Figs. 7A and 7B are two graphs showing Increasing Trial Data is Necessary for Learning from Larger Networks. Fig. 7A shows reconstruction accuracy of network outputs, measured as the correlation between the output units, and Fig. 7B shows reconstruction accuracy of network weights, measured as the correlation between recurrent weights, are plotted as a function of dataset size (number of trials) averaged over three networks. The different colors in the plot correspond to different number of neurons in the generator. Parameters: $\alpha = 0.9$, $\epsilon^{\text{conv}} \sim N(0, 10^{-3})$, $\epsilon^{\text{input}} \sim N(0, 10^{-2})$.

[0031] Fig. 8 is two plots showing CoRNN can scale to the largescale datasets. Both CPU and GPU implementation of CoRNN are fast and scale faster than data generation process itself. Parameters: $\alpha = 0.9$, $\epsilon^{\text{conv}} \sim N(0, 10^{-3})$, $\epsilon^{\text{input}} \sim N(0, 10^{-2})$.

[0032] Fig. 9 is a block diagram illustrating a computing device, according to some embodiments of the present disclosure.

[0033] Fig. 10 is a flowchart illustrating a computer-implemented method for training a neural network, according to some embodiments of the present disclosure.

[0034] DETAILED DESCRIPTION

[0035] The present invention will be understood by reference to the following detailed description, which should be read in conjunction with the appended drawings. It is to be appreciated that the following detailed description of various embodiments is by way of example only and is not meant to limit, in any way, the scope of the present invention. In the summary above, in the following detailed description, in the claims below, and in the accompanying drawings, reference is made to particular features (including method steps) of the present invention. It is to be understood that the disclosure of the invention in this specification includes all possible combinations of such particular features, not just those explicitly described. For example, where a particular feature is disclosed in the context of a particular aspect or embodiment of the invention or a particular claim, that feature can also be used, to the extent possible, in combination with and/or in the context of other particular aspects and embodiments of the invention, and in the invention generally. The terms “comprise(s),” “include(s),”

“having,” “has,” “can,” “contain(s),” and grammatical equivalents and variants thereof, as used herein, are intended to be open-ended transitional phrases, terms, or words that do not preclude the possibility of additional acts or structures. are used herein to mean that other components, ingredients, steps, etc. are optionally present. For example, an article “comprising” (or “which comprises”) components A, B, and C can consist of (i.e., contain only) components A, B, and C, or can contain not only components A, B, and C but also one or more other components. The singular forms “a,” “and” and “the” include plural references unless the context clearly dictates otherwise. Where reference is made herein to a method comprising two or more defined steps, the defined steps can be carried out in any order or simultaneously (except where the context excludes that possibility), and the method can include one or more other steps which are carried out before any of the defined steps, between two of the defined steps, or after all the defined steps (except where the context excludes that possibility).

[0036] The term “at least” followed by a number is used herein to denote the start of a range beginning with that number (which may be a range having an upper limit or no upper limit, depending on the variable being defined). For example, “at least 1” means 1 or more than 1. The term “at most” followed by a number is used herein to denote the end of a range ending with that number (which may be a range having 1 or 0 as its lower limit, or a range having no lower limit, depending upon the variable being defined). For example, “at most 4” means 4 or less than 4, and “at most 40% means 40% or less than 40%. When, in this specification, a range is given as “(a first number) to (a second number)” or “(a first number)-(a second number),” this means a range whose lower limit is the first number and whose upper limit is the second number. For example, 25 to 100 mm means a range whose lower limit is 25 mm, and whose upper limit is 100 mm.

[0037] The embodiments set forth the below represent the necessary information to enable those skilled in the art to practice the invention and illustrate the best mode of practicing the invention. For the measurements listed, embodiments including measurements plus or minus the measurement times 5%, 10%, 20%, 50% and 75% are also contemplated. For the recitation of numeric ranges herein, each intervening number there between

with the same degree of precision is explicitly contemplated. For example, for the range of 6-9, the numbers 7 and 8 are contemplated in addition to 6 and 9, and for the range 6.0-7.0, the number 6.0, 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, 6.9, and 7.0 are explicitly contemplated.

[0038] The term “substantially” means that the property is within 80% of its desired value. In other embodiments, “substantially” means that the property is within 90% of its desired value. In other embodiments, “substantially” means that the property is within 95% of its desired value. In other embodiments, “substantially” means that the property is within 99% of its desired value. For example, the term “substantially complete” means that a process is at least 80% complete, for example. In other embodiments, the term “substantially complete” means that a process is at least 90% complete, for example. In other embodiments, the term “substantially complete” means that a process is at least 95% complete, for example. In other embodiments, the term “substantially complete” means that a process is at least 99% complete, for example.

[0039] The term “substantially” includes a value that is within 10% less than or greater than the indicated value. In certain embodiments, the value is within 5% less than or greater than of the indicated value. In certain embodiments, the value is within 2.5% less than or greater than of the indicated value. In certain embodiments, the value is within 1% less than or greater than of the indicated value. In certain embodiments, the value is within 0.5% less than or greater than of the indicated value.

[0040] The term “about” includes when value is within 10% of the indicated value. In certain embodiments, the value is within 5% of the indicated value. In certain embodiments, the value is within 2.5% of the indicated value. In certain embodiments, the value is within 1% of the indicated value. In certain embodiments, the value is within 0.5% of the indicated value.

[0041] In addition, the invention does not require that all the advantageous features and all the advantages of any of the embodiments need to be incorporated into every embodiment of the invention.

[0042] Turning now to Figs. 1 - 12, a brief description concerning the various components of the present invention will now be briefly discussed.

- [0043] Experimental neuroscience is advancing rapidly. As this progress continues, it becomes increasingly important to update theoretical and computational tools to keep pace. With the recent advances in targeted stimulation of individual neurons, biologically interpretable computational models with generative power are needed to supply these experiments with real-time computational control.
- [0044] Among those biologically interpretable models are the data-constrained recurrent neural networks (dRNNs), where the number of units in the artificial network is matched to the number of neurons recorded, and the RNNs are trained to match the observed neural activities. The previous work in this paradigm uses FORCE learning, which was originally developed as an alternative to the reservoir computing paradigm for training read-out weights to produce a target function and then later applied to training recurrent weights to create dynamic attractors.
- [0045] Data-constrained RNNs have immense potential to generalize the decades old approaches of estimating functional connectivity between brain regions through co-fluctuations that are still being utilized in systems neuroscience to allow reverse engineering underlying computations, to provide real-time computational control for causal experimentation, and to serve as a computational complement to the widely used optogenetics. Influencing biological experiments computationally on the fly requires models that can be trained and provide predictions in real-time. Additionally, biological interpretability of the underlying architecture is necessary for reverse-engineering the underlying computation. These requirements render many traditional machine learning approaches trained in large clusters with billions of parameters impractical for a direct application to experiments performed with limited computational resources, a large portion of which would be allocated to other computational procedures such as real-time extraction of neural signals.
- [0046] When developing optimization procedures for data-constrained RNNs, another area of particular interest is the scalability of the training to data collected from several days and from more than 10,000 neurons for offline processing. With a model having scalable and fast learning capabilities, the inventors could potentially process multi-day imaging datasets and/or the brain-wide large datasets of the future and test

biological hypothesis regarding large-scale computations at an instance. Although seven years passed since the original attempt at reproducing neural activities with data-constrained interpretable RNNs, the ability of the data constrained RNN training paradigm to influence biological/experimental progress has been limited if any. Perhaps one major contribution to this is the incompatibility of the FORCE approach with the realities of the experimental data, as well as its extremely low speed of convergence. In fact, the dRNN training has never been truly validated via a comparative study of potential optimizers. Moreover, previous work attempted to solve two problems at once, e.g. designing an optimizer for the data constrained RNN training paradigm and transforming the experimental neural activities to a format trainable by a vanilla recurrent neural network with sigmoid nonlinearity. Yet, both problems are interdisciplinary in nature and require full attention of their own. Thus, for the scope of this work, their focus is explicitly on the former, i.e. the validation of the data-constrained RNN training. The inventors leave the latter problem, e.g. the transformation of the experimentally observed neural activities into a suitable data format, to future work. In this disclosure, the inventors first present a task called “large-scale functional connectivity extraction,” in which the inventors attempt to learn the functional connectivity between “recorded neurons” simulated from a generator model. Then, the inventors introduce a convex and scalable solver, called CoRNN, and validate its accuracy and speed with respect to alternatives including FORCE employed by the seminal works. Contributions include the following:

- The inventors introduce the large-scale functional connectivity extraction task and validate reproduction ability of data constrained RNNs on ground truth simulations
- The inventors introduce a new convex solver called CoRNN: Scalable, stable, and interpretable convex recurrent neural networks for copying brain dynamics.
- The inventors introduce a fixed-point initialization strategy to speed up convergence not only for CoRNN but for FORCE as well.

[0047] Unlike the traditional setting of dynamical system reconstruction task, where powerful algorithms (with potentially hidden units and deep layers) are trained to reproduce the underlying dynamical systems of an observation, the data-constrained RNN training

aims to match the artificial units with the number of observed neurons to interpret currents shared between brain regions and/or population of neurons as proxy for functional connectivity. While ignoring the hidden units may seem too restrictive and confounding when reproducing observed neural activities, several critical biological observations motivate the use of dRNNs: 1) the observed neural activities tend to lie on low dimensional manifolds, 2) the noise correlations limit the maximum amount of information in the population, which can be recovered with few thousands of neurons, and 3) the currents shared between brain regions and/or simulated networks of networks are robust to sub sampling of individual neurons. In fact, instead of viewing dRNNs as performing a more restricted version of dynamical system reconstruction task, a more biologically relevant view would be to consider dRNNs as a generalization of functional connectivity estimation, which was originally performed via considering co-fluctuations between brain regions.

[0048] 2. Theory 2.1. Data constrained RNN training for large scale functional connectivity extraction task the inventors start by introducing the large-scale functional connectivity extraction task, as shown in Fig. 1, which aims to reproduce interactions between brain regions and/or groups of neurons (e.g. functional connectivity). The inventors assume that the observations of real data can be captured by a generator RNN model. Then, the goal of the dRNNs is to reproduce the internal dynamics of the generator RNN. In most general case, observed neurons can be subsampled, or the generator might follow a different statistical model than dRNN, in which case the connectivity matrix learned by dRNN would not necessarily coincide with the underlying synaptic connectivity; rather be functional in essence.

[0049] For the scope of this work, the inventors aim to provide validity evidence for the optimization problem; thus, focus on the case with perfectly matching generator and learners. In this case, the terms functional and synaptical connectivity coincide, allowing a simple interpretation/scoring of the reproduction quality. The inventors test not only the accuracy of optimizers, but also focus on real-time learning and control possibility, robustness to hyperparameters, scalability to whole brain. In its essence, large scale functional connectivity task is a compromise between co-fluctuations

looked traditionally through noise and/or canonical correlations between brain regions; and the dynamical system reconstruction task, which traditionally uses more accurate/powerful approaches.

- [0050] 2.2. Generator model and estimator family of interest Following the trend in neuroscience, the inventors choose to focus on a recurrent neural network-based generator model. Specifically, the generators are assumed to be of the form:

$$\begin{aligned}\tau \frac{dr_i}{dt} &= -r_i + f(z_i) + \epsilon_i^{\text{con}}, \\ z_i &= \sum_{j=1}^{N_{\text{rec}}} W_{ij}^{\text{rec}} r_j + \sum_{j=1}^{N_{\text{in}}} W_{ij}^{\text{in}} u_j + \epsilon_i^{\text{input}},\end{aligned}\quad (1)$$

where z_i is the total input to the recurrent unit i , r_i is the firing rate of the unit i , and $f(\cdot)$ is a pre-defined non-linearity, W_{ij}^{rec} is the weight matrix for the recurrent connections, W_{ij}^{in} is the weight matrix from the input to the recurrent units, u is the input vector, and $\epsilon_i^{\text{input}}$ is the input noise to the recurrent units. Unlike previous literature, given their goal of reconstruction, the inventors also consider the existence of a conversion noise ϵ_i^{con} that accounts for errors and/or miss-matches in the non-linearity. Additionally, the generator model has the constraint $W_{ii}^{\text{rec}} = 0$ preventing any self-excitation of neurons.

- [0051] Suitable estimators for reconstructing such a model can be obtained (recursively) by discretizing the differential equation:

$$\hat{r}_{t+1,i} = (1 - \alpha)\hat{r}_{t,i} + \alpha f(\hat{z}_{t,i}), \quad (2)$$

- [0052] where the time scale parameter is defined as $\alpha = \frac{\Delta t}{\tau}$. Here, the model parameters influence $\hat{z}_{t,i}$ linearly such that

$$\hat{z}_{t,i} = \sum_j \hat{x}_{t,j} \hat{\theta}_{ji} = X\hat{\theta}, \quad (3)$$

where we define the short-hand notations $x = [r; u]$ and $\hat{\theta} = [W^{\text{rec}}; W^{\text{in}}]^T$ as concatenated matrices. The goal of the estimation problem is to find $\hat{\theta}$ such that $\hat{\theta} \approx \theta$.

[0053] 2.3 Convexification of the loss. Given that the overall problem is a regression problem, a reasonable choice would be to minimize the L2 loss:

$$\tilde{\mathcal{L}}_2(\hat{\theta}) = \sum_{i=1}^{n_{\text{rec}}} \sum_{t=1}^T (\hat{r}_{t,i} - r_{t,i})^2. \quad (4)$$

However, owing to the time recurrent definition of the general estimator $\hat{r}_{t,i}$, this minimization problem would require backpropagation through time. This approach would likely miss one of the inventors' main goals: being able to scale well to tens of thousands of neurons and hours of recordings on a personal computer.

[0054] As a first step, the inventors consider a smaller subset of estimators during training where $\hat{r}_{t-1,i}$ is teacher-forced to $r_{t-1,i}$ when computing $\hat{r}_{t,i}$. This turns the potentially infinite time problem into a single time-step one, which are called "single step prediction error paradigm". Under this condition, the L2 error transforms to a simpler form:

$$\mathcal{L}_2(\hat{\theta}) = \alpha^2 \sum_{i=1}^{n_{\text{rec}}} \sum_{t=1}^T (\hat{d}_{t,i} - d_{t,i})^2, \quad (5)$$

where we define $d_{t,i} = \frac{rt+1,i - (1-\alpha)rt,i}{\alpha}$ and correspondingly $\hat{d}_{t,i} = f(\hat{z})$ becomes an estimator consisting of a linear term through a single non-linearity. However, this simplified version of the loss is not convex and leads to a counter-intuitive global loss function with vanishing gradients in extremely large errors, as shown in Fig. 2.

[0055] Instead of minimizing the naive L2 loss, owing to the observation that $\frac{1+\hat{d}_{t,i}}{2}$ would follow a linear + sigmoid form, the inventors replace the L2 loss with a cross-entropy loss that is well known to be convex under this estimator, see logistic regression. On the other hand, instead of a naive replacement, the inventors use a weighted loss of the form

$$\mathcal{L}_{\text{weighted}}(\hat{\theta}) = \sum_{i=1}^{n_{\text{rec}}} \sum_{t=1}^T c_{t,i} \text{CE} \left(\frac{1 + \hat{d}_{t,i}}{2}, \frac{1 + d_{t,i}}{2} \right),$$

$$\text{CE}(a, b) = -b \log(a) - (1 - b) \log(1 - a). \quad (6)$$

Here, the inventors use CE to denote the cross-entropy loss and $c_{t,i}$ are weighting factors chosen as $c_{t,i} = [1 - d_{t,i}^2]^{-1}$ for the CoRNN solver for scalability as discussed below. For now, the regularized general loss function is written as

$$\mathcal{L}_{\text{CoRNN}}(\hat{\theta}) = \frac{1}{T} \mathcal{L}_{\text{weighted}}(\hat{\theta}) + \frac{\lambda}{2} \|\hat{\theta}\|_F^2, \quad (7)$$

where $\|\hat{\theta}\|_F$ is the Frobenius norm of $\hat{\theta}$.

[0056] 2.4. Loss minimization with a scalable solver. Having convexified the loss function, the inventors next observe that the loss function itself is perfectly separable such that $L_{\text{CoRNN}}(\hat{\theta}) = \sum_i L_i(\hat{\theta}_{:,i})$. Thus, without loss of generality and for any given i , the subproblems concerning each scalar outputs $d_i := d_{t,i}$ and corresponding (vector, not matrix) parameters $\hat{\beta} := \hat{\beta}(i) = \hat{\theta}_{:,i}$ and $c_i := c_{t,i}$ can be considered separately (See Appendix A) The gradient and Hessian of this smaller problem can be found as:

$$(\nabla_{\hat{\beta}} \mathcal{L}_i(\hat{\beta}))_k = \frac{1}{T} \sum_{t=1}^T c_t [\hat{d}_t - d_t] x_{t,k} + \lambda \hat{\beta}_k, \quad (8a)$$

$$(\nabla_{\hat{\beta}}^2 \mathcal{L}_i(\hat{\beta}))_{kl} = \frac{1}{T} \sum_{t=1}^T c_t (1 - \hat{d}_t^2) x_{t,k} x_{t,l} + \lambda \delta_{kl} \quad (8b)$$

[0057] In this naive form, the Hessian would need to be computed separately for each subproblem and after each parameter update as it depends on $\hat{d}_t$. This motivates the choice of $c_t = [1 - d_t^2]^{-1}$, since near the solution $\hat{d}_t \approx d_t$, the Hessian approximates to:

$$\begin{aligned}
(\nabla_{\hat{\beta}}^2 \mathcal{L}_i(\hat{\beta}))_{kl} &\approx \frac{1}{T} \sum_{t=1}^T x_{t,k} x_{t,l} + \lambda \delta_{kl} \\
&= \frac{1}{T} X^T X + \lambda I = H_{\text{fp}},
\end{aligned} \tag{9}$$

which is both parameter $\hat{\beta}$ (and thus $\hat{\theta}$) and target d_t independent; this is commonly shared across all subproblems as long as the convex program starts and remains nearby the optimum. Owing to its fixed nature, this Hessian is defined as the fixed-point Hessian and use this globally to perform optimization updates. Finally, the gradient of the full problem can be obtained by promoting Eq. 8a to a matrix:

$$\begin{aligned}
[\nabla_{\hat{\theta}} \mathcal{L}_{\text{CoRNN}}(\hat{\theta})]_{ji} &= -\frac{1}{T} \sum_{t=1}^T \underbrace{\frac{[d_{t,i} - \hat{d}_{t,i}]}{1 - d_{t,i}^2}}_{E_{t,i}} x_{t,j} + \lambda \hat{\theta}_{ji}, \\
&= -\frac{1}{T} X^T E + \lambda \hat{\theta},
\end{aligned} \tag{10}$$

where we define the prediction error matrix E . It is worth noting that each column of the gradient matrix contains the gradient for the scalar subproblem corresponding to the i th output. Then, defining the inverse (fixed-point) Hessian matrix as

$$A^+ = [X^T X + T\lambda I]^{-1}. \tag{11}$$

we obtain the following update rule (See Appendix A)

$$\theta^{k+1} = \underline{A^+ X^T X} \theta^k + \underline{A^+ X^T E}^k, \tag{12}$$

[0058] where all quantities that can be pre-computing the prediction error followed by matrix multiplications are underlined; this is well suited to be accelerated on a graphics processing unit (GPU).

[0059] 2.5 Choosing a suitable fixed point. The approximation performed at the local Hessian stage, e.g. $\frac{1-d_t^2}{1-d_t^2} \approx 1$, requires that we initialize the parameters $\theta^{(0)} := \theta_{\text{fp}}$ close to $\approx \theta^*$, where θ^* is the solution to the optimization problem. If this condition is met, the

approximate local Hessian derived in Eq. 9 would describe the local topology near the solution, the fixed point, and the steps in-between. As the inventors show in Appendix A, a suitable fixed-point candidate is the solution to the least squares problem on the currents Z :

$$\theta_{ls} := \underline{A^+ X^T} Z. \quad (13)$$

This step is computationally negligible, as the underlined matrix is already pre-computed for the iterations that follow.

[0060] 3. Results. 3.1 Convex recurrent neural networks (CoRNN) via ADMM framework. When the inventors derived the update equations of the fast solver in the previous section, the inventors omitted the constraints $W_{ii}^{rec} = 0$. However, these constraints could be biologically relevant, and perhaps be vital to eliminate overfitting in the reproduced network. In fact, for the generator model given in Eq. 1, $r_{t,i} \approx r_{t+1,i}$ for small α , making the unconstrained problem ill-posed since self-excitation would explain a good amount of the activity of each neuron. Thus, to incorporate the equality constraint, the inventors developed a solver using the alternating direction method of multipliers (ADMM), summarized in Fig.2.

[0061] In the ADMM solver, the primal problem is divided into two subproblems that are solved individually during separate iterations, yet linked through a consensus variable that ensures agreement between the solutions of both subproblems upon convergence. In this framework, first subproblem solves the unconstrained optimization problem whereas the second one enforces the constraints (See Appendix A for derivations). While the inventors primarily consider the equality constraint in this work, ADMM framework can be used to add additional L1 and/or nuclear norm regularizations to the learned connectivity matrix, and utilize biological priors for further inductive biases. Finally, the inventors note that given the highly irregular nature of the prediction error when $d_{t,i} \approx \pm 1$, the inventors supply the fast solver with a simple automated outlier detection step explained in Appendix A.

[0062] 3.2 Chaotic recurrent neural network reproduction. To validate CoRNN's reproduction abilities, the inventors utilize randomly connected chaotic recurrent neural networks

without any external input as generators. Specifically, the inventors pick the recurrent weights randomly from a normal distribution $W_{ii}^{\text{rec}} \sim N\left(0, \frac{g}{\sqrt{n_{\text{rec}}}}\right)$ with zero mean and standard deviation of $\frac{g}{\sqrt{n_{\text{rec}}}}$. For $g > 1$ and sufficiently large n_{rec} , these networks can sustain their activities without any external stimulus and exhibit chaotic behavior. For the chaotic generators, the inventors pick $g = 3$ throughout this disclosure to ensure that network is deep into the sustainable spontaneous activity regime. The inventors pick the chaotic regime specifically, since previous work established that teacher forcing when training RNN to output a function can have potentially harmful consequences to the stability of the network training. The inventors aim to test whether this generalizes to the reproduction task considered here.

[0063] To validate that even chaotic dynamics can be reconstructed via a single step prediction error paradigm, the inventors initialized randomly connected recurrent neural networks as generators, extracted neural activities for a fixed duration of length T , and trained the dRNNs using CoRNN, Newton's solver, gradient descent, and FORCE on varying number of iterations to benchmark their speed and accuracy. All algorithms were able to train, some achieving near perfect accuracies in reproducing the internal connectivity of the generator network, as shown in Fig. 3.

[0064] Whether minimizing weighted or the naive logistic loss leads to more accurate reconstruction depended on the underlying noise distribution of the data as shown in Fig. 3, yet CoRNN outperformed all other algorithms in speed in both cases with several orders of magnitude and converged in sub-second times even with the CPU implementation, whereas others took up to several seconds for training. In fact, FORCE was not able to converge in a pre-allocated maximum training time, showing the inefficiency of FORCE training for the reproduction task. To compare, CoRNN converged faster at a given run than the computer could import the Pytorch module at the start of the code.

[0065] 3.3. Robustness to regularization Having tested the speed of the algorithms, the inventors next turn their attention to robustness to regularization. If the instability claims regarding network training in generalizes to the network reproduction problem,

then the single-step prediction error paradigm, regardless of the optimizer, would only be successful for a narrow range of regularization parameters.

[0066] To test this scenario, the inventors first simulated a set of networks with parameters matching those in Fig. 3A and trained using various optimizers for a given number of iterations and varying (normalized) regularization levels. The results are shown in Fig. 4A, with regularization level = 1 corresponding to the default regularization parameter of the respective algorithm. Unlike expectations, all single-step prediction error solvers showed high robustness to changes in the regularization level for several orders of magnitude, whereas FORCE either only maintained the accuracy achieved already the fixed-point initialization or outperformed the initialization only for a hyper-tuned regularization value and led to a completely incorrect reproduction for others. Then, the inventors tested whether the fixed-point initialization can account for the high accuracies of the single-step prediction error solvers. Specifically, the inventors initialized the networks to a random initialization instead of the fixed point and re-did the analysis of Fig. 4A in Fig. 4B. The results indicate that fixed point initialization is not necessary to obtain high accuracies, although it has two distinct advantages: i) fixed point initialization leads to a slightly higher accuracy, which the inventors test more explicitly next; and ii) it stabilizes the FORCE solver for the dRNN training. In Fig. 4B, the inventors still initialize CoRNN to the fixed point to have a reference-points between two subfigures, although CoRNN can learn without fixed-point initialization as discussed next.

[0067] 3.4. Fixed point initialization increases convergence speed. Inspired by their observation in Fig. 4B that fixed point initialization increases the accuracies slightly, the inventors next tested whether the asymptotic accuracy is increased with the better initialization, or whether the convergence speed is slowed down with the random initialization compared to fixed point initialization. To test this scenario, the inventors simulate a slightly larger network of 500 neurons, which would amplify any difference in convergence speeds compared to a smaller network, for $T = 10000$ data points and re-ran the algorithms without fixed-point initialization. As shown in Fig. 5, CoRNN with or without fixed point initialization eventually converges to the same asymptotic

accuracy, while the fixed-point initialization increases the speed of convergence at least an order of magnitude for CoRNN.

- [0068] 3.5. 3-bit flip flop task Having provided validity evidence for the speed and accuracy of CoRNN, or single-step prediction error solvers in general, via the randomly connected chaotic RNN generators; the inventors next focus on an RNN with a more structured connectivity. Specifically, the inventors train a model RNN to perform a 3-bit flip flop task following as shown in Fig. 6.
- [0069] The 3-bit flip flop task had 3 dimensions of input and 3 of output. Each trial lasted 100 timepoints. Input dimensions could take on the values of 0, -1, or 1. At every timepoint, and for each dimension independently, there was a $p = 0.05$ probability of being in a -1 or 1 input state with equal probability, otherwise the state was 0. The output dimensions were initialized to 0, and would “flip” to -1 or 1 depending on the input in the same dimension, and stay in that state until the input dimension took on the other value. For instance, if the first input dimension was -1 at a certain time, then the output in that dimension would be -1 from that time forward, until the first input dimension took on a value of 1. At that point the first output dimension would take on a value of 1, until the first input dimension took on a value of -1. Thus, values in the input dimensions “flip” the state of the output dimensions, like a switch.
- [0070] Fig. 6A shows the output of an example test trial, not used for the generator RNN training or the CoRNN reproduction, with 200 time points followed by 100 time points of intertrial interval. Not only CoRNN reproduced network can perform the task, it gives the same output as the original network; potentially making similar mistakes as the network. Thus, as a first step, the inventors confirm that CoRNN can learn to output the same outputs as the original network.
- [0071] Next, the inventors look at underlying neural activities of an example (original) generator and reproduced (CoRNN learned) network in Fig. 6B. The reproduction matches not only the output of the network, but also the internal firing dynamics of individual units. Yet, as shown in Fig. 6B, the synaptic connections are not perfectly learned even in such a simple scenario with matching models, which requires more trials and potentially interventional data as explored next. Although surprisingly, the

input synaptic connections are perfectly learned in this scenario as shown in Fig. 6C, a potentially useful observation for causal biological experiments. 3.6. More trials may be necessary for larger networks Inspired by their observation that firing rates may be perfectly predicted even in test trials for large networks without exactly reproducing the underlying synaptic connectivity, the inventors next tested whether increasing the number of trials could lead to better reproduction of the original network.

[0072] Fig. 7 shows that the output of the network can be reproduced with far smaller numbers of trials, whereas accurately reproducing the internal connectivity requires significantly more trials. The data hungriness of the estimation process increases with the number of units in the generator network, which again emphasizes the importance of an optimization algorithm that can scale well to multiple day recordings. 3.7. CoRNN scales well to larger networks and recordings So far, the inventors showed that CoRNN works faster than alternative algorithms with run times at most in few minutes. Now the inventors show that, owing to the first order nature of the updates, CoRNN also scales well to larger networks and data points as shown in Fig. 8. Remarkably, CoRNN remains sub-minute for 200 trials (approximating an hour-long recording) and 1000 neurons, meaning that CoRNN can indeed be trained real-time even in batches in biological experiments.

[0073] 4. Discussion. In this disclosure, the inventors introduced a convex and scalable solver (CoRNN) for training data-constrained recurrent neural networks to reproduce brain activity. The inventors validated CoRNN on simulated data via the large-scale functional connectivity extraction task, where the inventors used both randomly connected chaotic networks and RNNs trained to perform a 3-bit flip flop task as generators. The inventors showed that CoRNN is as accurate yet faster than existing approaches by orders of magnitude and can easily scale to the large datasets of today, training in seconds to minutes depending on the number of trials and neurons. For this work, the inventors focused on developing a scalable, fast, and accurate solver for the large-scale functional connectivity extraction task. Thus, the inventors conceive several practical aspects such as subsampling of neuronal populations, miss-matches between non-linearities, miss-matches in time scales, and bringing experimental data in a

suitable format for the dRNN training. Further development of data constrained RNNs can support systems neuroscience research aiming to understand interarea interactions and replace the canonical/pairwise correlations as a proxy for functional connectivity estimation. Moreover, reverse-engineering the learned network can provide deeper understanding of how neuronal populations compute. Finally, with the advent of interventional data collection methods such as single-cell targeted optogenetics, dRNNs can be trained to test causal connections computationally and supply theoretical predictions at a scale previously unachievable for experimentation.

[0074] Fig. 9 is a functional block diagram illustrating an example computing device 100 configured to train a neural network to interpret brain dynamics, incorporated with teachings of the present disclosure, according to some embodiments. Fig. 9 illustrates only one particular example of computing device 100, and many other examples of computing device 100 may be used in other instances and may include a subset of the components included in the example computing device 100. Computing device 100 may include additional components not shown in FIG. 9.

[0075] As shown in the example of FIG. 9, computing device 100 includes one or more processing devices 102, one or more input devices 104, one or more output devices 106, one or more communication devices 108, one or more storage devices 110, network interface 112, and memory devices 114. Processing devices 102, input devices 104, output devices 106, communication devices 108, storage devices 110, network interface 112, and memory devices 114 are interconnected via bus 190.

[0076] Processing devices 102 may include one or more execution cores (CPUs). For example, computing device 100 may also include a peripheral controller hub (PCH) (not shown). In another example, computing device 100 may also include a sensors hub (not shown). Input devices 104 and output devices 106 may include, for example, user interface device(s) including a display, a touch-screen display, printer, keypad, keyboard, etc., sensor(s) including accelerometer, global positioning system (GPS), gyroscope, etc., communication logic, wired and/or wireless, storage device(s) including hard disk drives, solid-state drives, removable storage media, etc. I/O ports for input devices 104 and output devices 106 may be configured to transmit and/or receive commands and/or

data according to one or more communications protocols. For example, one or more of the I/O ports may comply and/or be compatible with a universal serial bus (USB) protocol, peripheral component interconnect (PCI) protocol (e.g., PCI express (PCIe)), or the like.

[0077] Processing devices 102 may implement functionality and/or execute instructions within computing device 100. The instructions may be stored completely or partially within storage devices 110 and memory devices 114. For example, processing devices 102 on computing device 100 may receive and execute instructions stored by storage devices 110 that provide the functionality of training a neural network to interpret brain dynamics. These instructions executed by processing devices 102 may cause computing device 100 to store and/or modify information within storage devices 48 during program execution.

[0078] Storage devices 110 may generally comprise a random-access memory ("RAM"), a read-only memory ("ROM"), and a permanent mass storage device, such as a disk drive or SDRAM (synchronous dynamic random-access memory). Computing device 100 may store program code for modules and/or software routines.

[0079] Storage devices 110 may also store operating system, kernel, kernel space, user space, and user protected address space. Storage devices 110 may store one or more processes (i.e., executing software application(s)). One or more processes may generally execute in parallel, i.e., as a plurality of processes and/or a plurality of threads. These software components may be loaded from a non-transient computer-readable storage medium into storage devices 110 using a drive mechanism associated with a non-transient computer-readable storage medium, such as a floppy disc, tape, DVD/CD-ROM drive, memory card, or other like a storage medium. In some embodiments, software components may also or instead be loaded via a mechanism other than a drive mechanism and computer-readable storage medium (e.g., via network interface 112).

[0080] Embodiments of the presently disclosed invention are a computer software system and a solution to an important optimization problem. Some embodiments include convex optimization method that fits a recurrent neural network to reproduce the neuronal activities of mice or other animals observed from experimental data through calcium

and voltage imaging. Some embodiments could fall within the domain of “listening to” the neural activities, but beyond listening, embodiments model the activities. Moreover, some embodiments have practical implications for brain machine interface research.

[0081] A distinguishing factor of some embodiments of the presently disclosed invention to previous technology is the formulation of the problem in a language that matches the biological reality of experiments. FORCE, the previous approach, has several shortcomings that is described herein. Moreover, the inventors develop the convex solver that is 1-2 order of magnitude faster than the closest competition.

[0082] Among many possible uses or applications of some embodiments of the presently disclosed invention are brain-machine interfaces using this approach, research regarding large-scale connectivity/discovery of functionally connected brain circuits from calcium imaging recordings etc. Embodiments of the presently disclosed invention may be used with efferent BMIs, passive BMIs, sensory BMIs, two-photon (2P) calcium imaging, genetically encoded calcium indicators (GECOs), for example.

[0083] The invention illustratively disclosed herein suitably may explicitly be practiced in the absence of any element which is not specifically disclosed herein. While various embodiments of the present invention have been described in detail, it is apparent that various modifications and alterations of those embodiments will occur to and be readily apparent those skilled in the art. However, it is to be expressly understood that such modifications and alterations are within the scope and spirit of the present invention, as set forth in the appended claims. Further, the invention(s) described herein is capable of other embodiments and of being practiced or of being carried out in various other related ways. The present disclosure also contemplates other embodiments “comprising,” “consisting of” and “consisting essentially of,” the embodiments or elements presented herein, whether explicitly set forth or not. In addition, it is to be understood that the phraseology and terminology used herein is for the purpose of description and should not be regarded as limiting. The use of “including,” “comprising,” or “having” and variations thereof herein is meant to encompass the

items listed thereafter and equivalents thereof as well as additional items, while only the terms “consisting of” and “consisting only of” are to be construed in the limitative sense.

APPENDIX A

CORNN: Convex optimization of recurrent neural networks for rapid inference of neural dynamics

Fatih Dinc*Department of Applied Physics
Stanford University
Stanford, CA 94305**Adam Shai***CNC Program
Stanford University
Stanford, CA 94305**Mark J. Schnitzer†**Howard Hughes Medical Institute
CNC Program
Stanford University
Stanford, CA 94305**Hidegori Tanaka‡**Physics & Informatics Laboratories, NTT Research, Inc.
Sunnyvale, CA 94085
Center for Brain Science, Harvard University
Cambridge, MA 02138**Abstract**

Advances in optical and electrophysiological recording technologies have made it possible to record the dynamics of thousands of neurons, opening up new possibilities for interpreting and controlling large neural populations in behaving animals. A promising way to extract computational principles from these large datasets is to train data-constrained recurrent neural networks (dRNNs). Performing this training in real-time could open doors for research techniques and medical applications to model and control interventions at single-cell resolution and drive desired forms of animal behavior. However, existing training algorithms for dRNNs are inefficient and have limited scalability, making it a challenge to analyze large neural recordings even in offline scenarios. To address these issues, we introduce a training method termed Convex Optimization of Recurrent Neural Networks (CORNN)¹. In studies of simulated recordings, CORNN attained training speeds ~ 100 -fold faster than traditional optimization approaches while maintaining or enhancing modeling accuracy. We further validated CORNN on simulations with thousands of cells that performed simple computations such as those of a 3-bit flip-flop or the execution of a timed response. Finally, we showed that CORNN can robustly reproduce network dynamics and underlying attractor structures despite mismatches between generator and inference models, severe subsampling of observed neurons, or mismatches in neural time-scales. Overall, by training dRNNs with millions of parameters in subminute processing times on a standard computer, CORNN constitutes a first step towards real-time network reproduction constrained on large-scale neural recordings and a powerful computational tool for advancing the understanding of neural computation.

1 Introduction

Understanding the relationship between neural dynamics and computational function is fundamental to neuroscience research [1]. To infer computational structure from neural population dynamics,

*These authors contributed equally to this work.

†These authors co-supervised this work.

¹CORNN software and reproduction code are available at <https://github.com/schnitzer-lab/CORNN-public>

APPENDIX A

neuroscientists regularly collect and analyze large-scale neural recordings using electrophysiological or optical imaging recording methods. Both recording modalities have undergone rapid progress in recent years, yielding a steady increase in the numbers of cells that can be recorded and manipulated [2–7]. Owing to this progress, an urgent need has emerged for developing and refining theoretical and computational tools suited for the analysis of large recording datasets.

Data-driven modeling of neural network dynamics, in which recordings of neural activity are transformed into a computational model through network reconstruction, provides a promising way of bridging experiments and theory [8]. This approach not only complements recent experimental breakthroughs in targeted stimulation of individual neurons ([9, 10]) for hypothesis testing and brain-machine interfaces but also facilitates the distillation of fundamental computational principles from large amounts of experimental data [1, 11, 12]. Historically, especially for recordings of individual or small numbers of cells, hand-crafted network models were used [13–20]. While valuable, traditional hand-crafted models often struggle to handle the large datasets generated in contemporary neuroscience.

To bridge the gap, a new approach has recently emerged in which artificial networks are trained on tasks mimicking those performed in the laboratory by animal models [21–23]. However, this approach lacks the ability to transform observed patterns of neural activity into data-driven, mechanistic computational models. A promising solution to these issues, which retains the advantages of neural network-based modeling, is the training of dRNNs [24–29]. These models, part of a broader suite of computational strategies [11, 21, 30–32], align the dynamics of the neural units within the RNN to the activities of recorded neurons. This data-constrained methodology for RNN training has the potential to modernize estimations of functional connectivity [33]—still a mainstay in systems neuroscience [6]—and provides a viable path to distill underlying computations by extracting the state-space representations of the learned dRNNs [12]. Additionally, dRNNs may open the door to computational control for causal experimentation [9, 10], serving as a valuable computational adjunct to widely used techniques like optogenetics [8, 34].

Despite recent advances in the dRNN framework, how to perform fast and scalable reconstructions of neural activity traces from large-scale empirical recordings has remained unclear. Notably, the slowness of existing dRNN optimization algorithms may often necessitate the use of high-performance clusters and several days of computation in large-scale experiments. These limitations might have been a barrier to the widespread adoption of dRNN approaches. Moreover, to fully harness the potential of dRNNs, it is essential to extend their range of applicability from offline analyses to on-the-fly applications within individual recording sessions. An approach to training dRNNs in a nearly immediate manner would be a key advancement that would enable novel experimental approaches, such as theory-driven real-time interventions targeting individual cells with specific computational or functional roles (Fig. 1). However, realizing these benefits requires having an optimization routine that is fast, robust, and scalable to large networks.

In this work, we present CORNN, a convex solver designed for fast and scalable network reproduction. We demonstrate its accuracy and efficiency compared to alternative network training methods such as FORCE [35] and back-propagation through time (BPTT), which have been employed in previous seminal works utilizing dRNNs [24–29]. Our main contributions include the development of CORNN (Fig. 2), the introduction of an initialization strategy to accelerate convergence (Figs. 2, S4, and S6), and the demonstration of CORNN’s scalable speed (Figs. 3, 5, S2, S3, S5, S8, and S9), which enables rapid training on the activity patterns of thousands of neurons (Figs. 4, 5, S7, S8, S9, S10, and S11). CORNN’s performance can be further enhanced by 1–2 orders of magnitude through the use of a standard graphical processing units (GPU) on a desktop computer (Figs. S5 and S8). Unlike BPTT and FORCE, CORNN does not require fine-tuning (Figs. S1, S6), making it a user-friendly technology for biologists. Lastly, we highlight CORNN’s robustness against non-idealities such as mismatches in the assumed dynamical system equations (Figs. 4, 5, S10, and S11), subsampling of neural populations (Fig. 5), differences in dynamical time-scales (Fig. S10), and existence of non-Gaussian (Fig. S3) or correlated noise (Fig. S11).

By enabling user-friendly, subminute training on standard computers, CORNN represents a first necessary step in transforming data-constrained recurrent neural networks from a theoretical concept into an experimental/computational technology. While this work focuses on introducing and validating the fast solver on simulated benchmarks, future work with CORNN should focus on addressing the challenges that arise when analyzing large-scale neural recordings from experimental neuroscience.

APPENDIX A

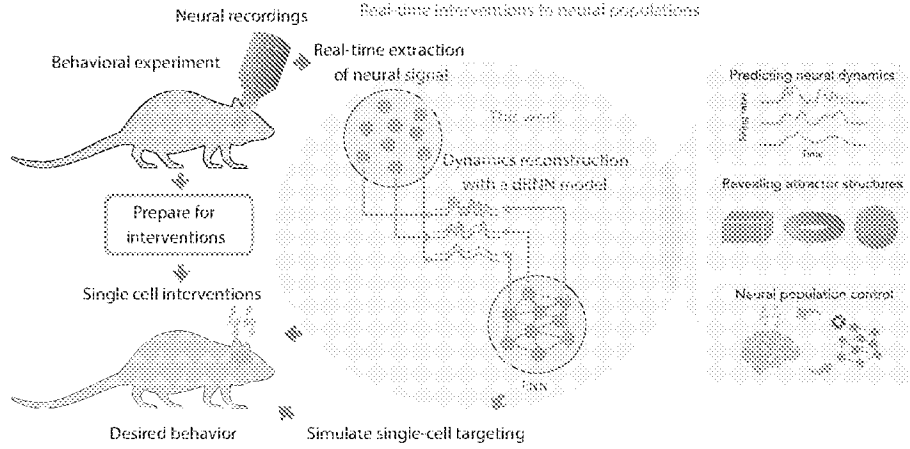


Figure 1: Using data-constrained recurrent neural networks for the interpretation and manipulation of brain dynamics within a potential experimental pipeline. This approach centers around online modeling of network dynamics, which can enhance hypothesis testing at the single-cell level and support advancements in brain-machine interface research. The training process is motivated by three objectives: (i) predicting the patterns of neural populations, (ii) revealing inherent attractor structures, and (iii) formulating optimal control strategies for subsequent interventions.

2 Approach

2.1 Experimental setup for real-time interventions using dRNNs

Previous work suggested the use of dRNNs for real-time feedback between experimental and computational research [8]. To date, however, the use of dRNNs has remained offline. Past studies extracted computational principles from neural recordings by fitting dRNNs *after* data collection [24–29]. Moreover, the use of dRNNs has generally required the expertise of computational neuroscientists, likely due to the complications associated with the optimization and training of neural network models. However, with the advent of large-scale recordings and data-driven approaches, real-time brain-machine interface research is likely a forthcoming application [36–38]. In such use cases, neurobiologists might wish to train dRNNs via a user-friendly approach. Hence, a fast and straightforward dRNN training procedure would enable a new breed of interventional experiments to dissect the brain’s microcircuitry.

To appreciate the advantages of fast data-driven reconstruction of neural dynamics, consider a hypothetical scenario in which dRNNs, facilitated by CORNN, enable real-time interventions at the single-cell level (Fig. 1). In this scenario, neural activities from mice performing behavioral tasks are captured and extracted in real-time using advanced imaging or electrophysiological technologies and pre-processing algorithms [2, 3, 7, 39–41]. As the experiment progresses, neural activity traces from each mouse are reproduced by training dRNNs, which are then reverse-engineered to reveal underlying attractor structures. Rapid dRNN training allows for a tight feedback loop between incoming measured data and optimal experimental design, in order to refine the inferred model. Techniques similar to adversarial attacks [42] might be used to devise optimal cell targeting strategies from these dRNNs [8], allowing one to test hypotheses about the computational roles of individual neurons or to identify optimal neurons for use within brain-machine interfaces. Once a perturbation strategy is determined, it can be tested on subsequent trials of the experiment, using the same animal whose recorded neural dynamics led to the trained dRNN. Thus, a fast dRNN training algorithm could allow for better fitting of a dynamical system model to the experimental data and provide a natural testbed to probe hypotheses about the computational structure of the biological neural circuitry.

Motivated by the goals described above, our paper focuses on training dRNNs accurately and as fast as possible. However, experimental concerns and the need for biological interpretability lead to several constraints. Firstly, to ensure real-time communication with the experimental apparatus, we require that the training process takes place on standard lab computers, not clusters. Secondly, given the computational complexity of real-time processing in large scale recordings, especially with thousands of neurons, we expect that at least one of the GPUs is reserved for extracting neural activities

APPENDIX A

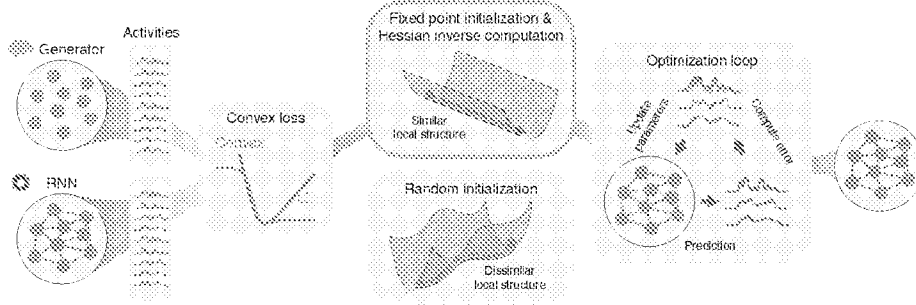


Figure 2: **CORNN: Convex and scalable solver for dRNNs via ADMM [51]**. The CORNN algorithm optimizes the parameters of a recurrent neural network so that activity in hidden units align with activities measured from a ground-truth system, we refer to as a generator. The choice of objective function results in a convex loss landscape. The algorithm starts by finding a fixed point for initialization, where the Hessians of all subproblems are aligned to the correlation matrix of neural activities, which can be pre-computed and pre-inverted. The optimization loop then iterates through predicting the neural activities, computing the prediction error, updating the parameter values, and checking for convergence. The final output is the set of optimized parameters, which represents the functional connections between the neurons in the data-constrained recurrent neural network.

from brain-imaging movies [43] (though see [44] for a promising development in smaller scale experiments), or spikes from electrophysiological recordings [7], leaving the central processing unit (CPU), or perhaps a second GPU, for training dRNNs. Finally, maintaining biological interpretability via 1-1 matching of observed and modeled neurons is crucial, as arbitrariness brought by hidden units can explain away existing functional connectivity and/or contributions of observed neurons to underlying attractor structures. These concerns make many traditional machine learning architectures and optimization routines, typically trained on large clusters with billions of parameters, unsuitable for our purposes. This includes recent work on the dynamical system reconstruction paradigm [31, 45–50], which chiefly aims to maximize reconstruction accuracy of dynamical systems without regard to biological interpretability, training speed, and scalability.

2.2 Inference model and estimator family of interest

Recurrent neural networks are universal approximators for dynamical systems, whose internal computations can be reverse engineered and interpreted [12, 52]. Thus, we choose leaky firing-rate RNNs as the inference models of CORNN, which follow the dynamical system equations [22]:

$$\tau \frac{dr_i}{dt} = -r_i + \tanh(z_i) + \epsilon_i^{\text{con}}, \quad z_i = \sum_{j=1}^{N_{\text{rec}}} W_{ij}^{\text{rec}} r_j + \sum_{j=1}^{N_{\text{in}}} W_{ij}^{\text{in}} u_j + \epsilon_i^{\text{input}}, \quad (1)$$

where τ is the neural decay time, z_i is the total input to the recurrent unit i , r_i is the firing rate of the unit i , and $\tanh(\cdot)$ is the pre-defined non-linearity, W_{ij}^{rec} is the weight matrix for the recurrent connections, W_{ij}^{in} is the weight matrix from the input to the recurrent units, u_j is the input vector, and $\epsilon_i^{\text{input}}$ is the input noise to the recurrent units. Unlike previous literature, given our goal of reconstruction, we also consider the existence of a conversion noise ϵ_i^{con} that accounts for errors and mismatches in the non-linearity. Additionally, the inference model has the constraint $W_{ii}^{\text{rec}} = 0$ preventing any self-excitation of neurons.

Suitable estimators for reconstructing such a model can be obtained (recursively) by discretizing the differential equation:

$$\hat{r}_{t+1,i} = (1 - \alpha)\hat{r}_{t,i} + \alpha f(\hat{z}_{t,i}), \quad (2)$$

where we define the time scale parameter $\alpha = \frac{\Delta t}{\tau}$. Here, the model parameters influence $\hat{z}_{i,t}$ explicitly linearly, and implicitly through previous time activities, such that $\hat{z}_{t,i} = \sum_j \hat{x}_{t,j} \hat{\theta}_{ji}$. We define the short-hand notations $x = [r; u]$ and $\hat{\theta} = [W^{\text{rec}}; W^{\text{in}}]^T$ as concatenated matrices. The goal of the estimation problem is to find $\hat{\theta}$ minimizing a loss function.

APPENDIX A

A reasonable choice for the loss function of this regression problem is the traditional $\mathcal{L}_2$ loss:

$$\mathcal{L}_2(\hat{\theta}) = \sum_{i=1}^{n_{\text{rec}}} \sum_{t=1}^T (\hat{r}_{t,i} - r_{t,i})^2. \quad (3)$$

However, due to the time recurrent definition of the general estimator $\hat{r}_{t,i}$, this minimization problem would require backpropagation through time (BPTT). It is not apriori clear if BPTT is the right optimization method for network reproduction, for which one has access to high information content regarding the internal dynamics of the network. We discuss this in the results below.

Instead of using BPTT, we consider a smaller subset of estimators during training where $\hat{r}_{t,i}$ is teacher-forced to $r_{t,i}$ when computing $\hat{r}_{t+1,i}$. This turns the potentially infinite time problem into a single time-step one, which we call “single step prediction error paradigm.” Under this condition, the $\mathcal{L}_2$ loss function transforms to a simpler form:

$$\mathcal{L}_2(\hat{\theta}) = \alpha^2 \sum_{i=1}^{n_{\text{rec}}} \sum_{t=1}^T (\hat{d}_{t,i} - d_{t,i})^2, \quad (4)$$

where we define $d_{t,i} = \frac{r_{t+1,i} - (1-\alpha)r_{t,i}}{\alpha}$ and correspondingly $\hat{d}_{t,i} = \tanh(\hat{z}_{t,i})$ becomes an estimator consisting of a linear term through a single non-linearity. However, this simplified version of the loss is not convex and leads to a counter-intuitive global loss function with vanishing gradients (Fig. 2).

2.3 Convexification of the loss function

Rather than minimizing the naive $\mathcal{L}_2$ loss function, we observe that $\frac{1+\hat{d}_{t,i}}{2}$ follows a linear + sigmoid form. As a result, we replace the $\mathcal{L}_2$ loss function with a cross-entropy loss function that is well known to be convex under this estimator. In fact, in the limit of $d_{t,i} \rightarrow \pm 1$, the problem reduces to logistic regression. Instead of a naive replacement, we use a weighted loss of the form:

$$\mathcal{L}_{\text{weighted}}(\hat{\theta}) = \sum_{i=1}^{n_{\text{rec}}} \sum_{t=1}^T c_{t,i} \text{CE} \left(\frac{1+\hat{d}_{t,i}}{2}, \frac{1+d_{t,i}}{2} \right). \quad (5)$$

Here, we use CE to denote the cross-entropy loss function and $c_{t,i}$ are non-negative weighting factors chosen as $c_{t,i} = [1 - d_{t,i}^2]^{-1}$ for the CORNN solver. The specific choice of $c_{t,i}$, which resembles (but is not) preconditioning [53], is motivated from a theoretical view in Supplementary Section S1.3. The reader can verify that the Hessian of this loss, derived in Eq. (S4), is positive semi-definite. For now, we write down the regularized loss function for CORNN as:

$$\mathcal{L}_{\text{CORNN}}(\hat{\theta}) = \frac{1}{T} \mathcal{L}_{\text{weighted}}(\hat{\theta}) + \frac{\lambda}{2} \|\hat{\theta}\|_F^2, \quad (6)$$

where $\|\hat{\theta}\|_F$ is the Frobenius norm of $\hat{\theta}$ and λ is the regularization parameter.

2.4 Fundamental principle of CORNN: Subproblem Hessians align to the correlation matrix of neural activity traces

Having convexified the loss function, we next observe that the loss function itself is perfectly separable such that $\mathcal{L}_{\text{CORNN}}(\hat{\theta}) = \sum_i \mathcal{L}_i(\hat{\theta}_{:,i})$, where $\hat{\theta}_{:,i}$ denotes the i th column of the parameter matrix $\hat{\theta}$ (See Eq. S1). This means that the original problem can be divided into n_{rec} sub-problems and solved independently. However, inspired by the fact that solving least-squares for a vector or a matrix target has the same complexity, we can devise a faster method. Specifically, the choice of the specific weighting factors, $c_{t,i} = [1 - d_{t,i}^2]^{-1}$, leads to a shared (approximate) Hessian for all subproblems (See Eq. (S4)):

$$H \approx \frac{1}{T} X^T X + \lambda I, \quad (7)$$

where X stands for $x_{t,i}$ in the matrix form, I is the identity matrix, and λ is the regularization parameter. We note that the approximation is exact in the limit $\hat{d}_{t,i} = d_{t,i}$. In other words, we align the Hessian of each subproblem to the correlation matrix of neural activities. Unlike for

APPENDIX A

least-squares minimization, this is a local, not global, Hessian; but as long as we can initialize the network sufficiently close to the optimal solution, we expect that the descent direction with the approximate Hessian converges quickly. In this sense, the approximation in Eq. (7) shows similarities with quasi-Newton methods [53] but is distinct in the pre-computed nature of the Hessian.

Following straightforward algebra, we obtain the gradient as

$$\nabla_{\hat{\theta}} \mathcal{L}_{\text{CORNN}}(\hat{\theta}) = -\frac{1}{T} X^T E + \lambda \hat{\theta}, \quad (8)$$

where we define the prediction error matrix $E_{t,i} := \frac{[d_{t,i} - \hat{d}_{t,i}]}{1 - \hat{d}_{t,i}^2}$. It is worth noting that each column of the gradient matrix contains the gradient for the scalar subproblem corresponding to the i th output. Then, defining the inverse (fixed-point) Hessian matrix as

$$A^+ = [X^T X + T\lambda I]^{-1}, \quad (9)$$

we obtain the following update rule (See Eq. (S20))

$$\theta^{k+1} = A^+ X^T X \theta^k + A^+ X^T E^k, \quad (10)$$

where we highlight all quantities that can be pre-computed with blue. In essence, the update rule, which we call ‘‘Hessian aligned prediction error (HAPE) update,’’ consists of computing the prediction error followed by matrix multiplications; thus is well suited to be accelerated on a GPU if available, but still can be efficiently computed on CPU cores.

The alignment of subproblem Hessians relies on the assumption that we can find a ‘‘close-enough’’ starting position, called the fixed-point, for the initial parameters $\hat{\theta}^{(0)}$. As we show in Supplementary Section S1.2, a suitable candidate is the approximate least-squares solution:

$$\theta_{ls} := A^+ X^T Z. \quad (11)$$

Then, the initial set of predictions becomes $\hat{d}_{t,i} := \tanh(\sum_j x_{t,j}(\theta_{ls})_{j,i})$, which is subsequently refined through the HAPE iterations. This step is computationally negligible, as the blue colored matrix is already pre-computed for the iterations that follow.

In this section, we focused on the leaky firing rate RNNs as described in Eq. (1). However, CORNN can be applied to the other widely used variant of RNNs, *i.e.*, the leaky current RNN described in Eq. (S28) and regularly employed in neuroscience literature [12, 26, 52]. The reproduction of the leaky current RNNs can be made convex by observing the direct link between the firing rates, r , and the currents, z , via a linear plus non-linear relationship, *i.e.*, $r = \tanh(z)$ and subsequently replacing $d_{t,i}$ with $r_{t,i}$ in Hessian and gradient calculations. To prevent introducing unnecessary complexity, we focus on leaky firing rate RNNs in this work.

3 Results

3.1 CORNN as a versatile base model

While Hessian alignment, and subsequent HAPE updates, are the core principles driving the fast solver of CORNN, a versatile base model should be able to incorporate various regularization schemes and sparsity constraints. For example, the addition of constraints $W_{ii}^{\text{rec}} = 0$ or a potential low-rank regularization would prevent the Hessian alignment and subsequently the use of fast HAPE updates. However, these constraints could be biologically relevant and perhaps vital to eliminate overfitting in the reproduced network. In fact, for the inference model given in Eq. (1), $r_{t,i} \approx r_{t-1,i}$ for small α ; making the unconstrained problem ill-posed since self excitation would explain a good amount of the activity of each neuron. Thus, to incorporate the equality constraint and many others, we developed a solver using the alternating direction method of multipliers (ADMM) [51] (summarized in Fig. 2).

In the ADMM solver, the primary problem is divided into two subproblems that are solved individually during subsequent iterations, yet linked through a consensus variable that ensures agreement between the solutions of both subproblems upon convergence. In this framework, the first subproblem solves the unconstrained optimization problem, whereas the second one enforces the constraints/regularization (See Supplementary Section S1 for derivations). This division allows the

APPENDIX A

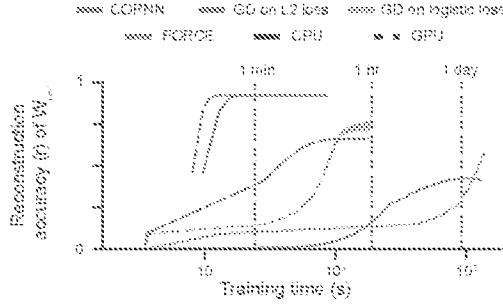


Figure 3: **CORNN reduces training times by several orders of magnitude.** The plots illustrate the relationship between reconstruction accuracy (Pearson's correlation coefficient between ground truth and inferred weights) and training time, measured in seconds on a log scale. The FORCE approach (here, on firing rates) is the default method in neuroscience literature for dRNN training [24, 25, 28, 29]. Parameters: $\alpha = 0.1$, $n_{\text{rec}} = 5000$, $T = 30000$. No input. $\epsilon^{\text{conv}} \sim \text{Poisson}(10^{-3})$, $\epsilon^{\text{input}} \sim \mathcal{N}(0, 10^{-4})$. Lines: median. Error bars: s.d. over 7 networks.

use of fast HAPE updates in the computationally extensive first subproblem, where Hessians can be aligned.

While we only considered the equality constraint in this work, the ADMM framework can be used to add additional L1 and/or nuclear norm regularizations to the learned connectivity matrix [51, 54], and utilize biological priors for further inductive biases. Finally, we note that given the highly irregular nature of the prediction error when $d_{t,i} \approx \pm 1$, we supplied the fast solver with a simple automated outlier detection step explained in Supplementary Section S1.5.

3.2 Backpropagation through time on the $\mathcal{L}_2$ loss is suboptimal

In a traditional setting, in which neural activations are hidden and need to be learned through training to provide a correct output, backpropagation through time (BPTT) is necessary to learn long-term dependencies in the data. However, in the network reproduction where neural activations are no longer hidden, it is not clear whether BPTT would be beneficial for the learning, since long-term dependencies are inherently present in the state space of the dynamical system, whose equations are given in Eq. (1). Specifically, given that Eq. (1) has only a single time derivative, the time evolution is performed in a Markovian manner, *i.e.*, given the current state, the next state can be computed without need for additional information (up to a corruption by random noise).

To test whether BPTT is a suitable training algorithm for network reproduction, we examined randomly connected chaotic recurrent neural networks as a synthetic ground-truth benchmark (See Section S3.1 for details). As shown in Fig. S1, contrary to expectation, teacher-forcing enhanced the convergence speed of the BPTT. Moreover, we also observed higher reproduction accuracy with the cross-entropy loss compared to the traditional $\mathcal{L}_2$ loss, inline with our observation from Fig. 2 that $\mathcal{L}_2$ loss leads to vanishing gradients in large error regime. As expected, BPTT took around a minute even in the toy example of Fig. S1 with 100 time points and 100 neurons, approximately two orders of magnitude smaller than a standard calcium imaging session in both dimensions.

3.3 CORNN decreases training times several orders of magnitude

Having shown that teacher-forcing can speed-up convergence in chaotic networks, we next validated that chaotic dynamics can be reconstructed via a single step prediction error paradigm that CORNN utilizes. Specifically, we initialized randomly connected recurrent neural networks as generators, extracted neural activities for a fixed duration of length T , and trained the dRNNs using CORNN, Newton's solver, gradient descent, and FORCE on varying number of iterations to benchmark their speed and accuracy (See Supplementary Sections S2 and S3.1 for details). All algorithms were able to train, some achieving near perfect accuracies in reproducing the internal connectivity of the generator network (See Figs. 3, S2, and S3). However, FORCE, the current default method in neuroscience literature for dRNN training [24, 25, 28, 29], was 4 orders of magnitude slower (Fig. 3) and needed fine-tuned hyperparameters to barely outperform the fixed-point initialization (Fig. S4).

Whether minimizing the weighted or logistic loss led to more accurate reconstruction depended on the underlying noise distribution (we tested two noise distributions: Poisson(mean) and $\mathcal{N}(\text{mean}, \text{variance})$) of the data as shown in Figs. 3, S2, and S3. However, CORNN outperformed all other algorithms in speed in both cases with several orders of magnitude and converged

APPENDIX A

in sub-second times even with the CPU implementation, whereas others took up to several days for training (Fig. 3). We also observed increased efficiency for CORNN on a GPU (Fig. S5) and that fixed-point initialization could stabilize other algorithms (Figs. S4 and S6).

3.4 CORNN runtimes scale linearly with data size and polynomially with the network size

To this point, we provided validity evidence for the speed and accuracy of CORNN, or single-step prediction error paradigm in general, via the randomly connected chaotic RNN generators. Next, we focused on RNNs with more structured connectivity matrices. Specifically, we trained a model RNN using BPTT to perform a 3-bit flip flop task following [12] as shown in Fig. S7 (See Supplementary Section S3.2 for details). Fig. S7A shows the output of an example test trial, not used for the generator RNN training or the CORNN reproduction, with 200 time points followed by 100 time points of inter trial interval. We observed that CORNN reproduced network performed the task the same way the original network did; potentially making similar mistakes. Thus, as a first step, we confirmed that CORNN can learn to output the same outputs as the original network.

Next, we looked at underlying neural activities of an example (original) generator and reproduced (CORNN learned) network in Fig. S7B. The reproduction matched not only the output of the network, but also the internal firing dynamics of individual units. Yet, as shown in Fig. S7C, the synaptic connections were not perfectly learned even in such a simple scenario with matching models, which required more trials and potentially interventional data [15]. Thus, we next tested whether increasing the number of trials could lead to better reproduction of the original network and whether CORNN can scale well to fit the increased data and network size.

Fig. S8 shows that accurately reproducing the large networks required significantly more trials. Moreover, the data size requirement for the estimation process increased with the number of units in the generator network, which emphasizes the importance of an optimization algorithm that can scale well to multiple day recordings [55]. Fortunately, CORNN had a near-linear scaling of training times vs. the increasing number of trials and hence can handle large datasets (See Fig. S8). In the other direction, we observed that the training times scaled polynomially (Fig. S9), following n_{rec}^3 .

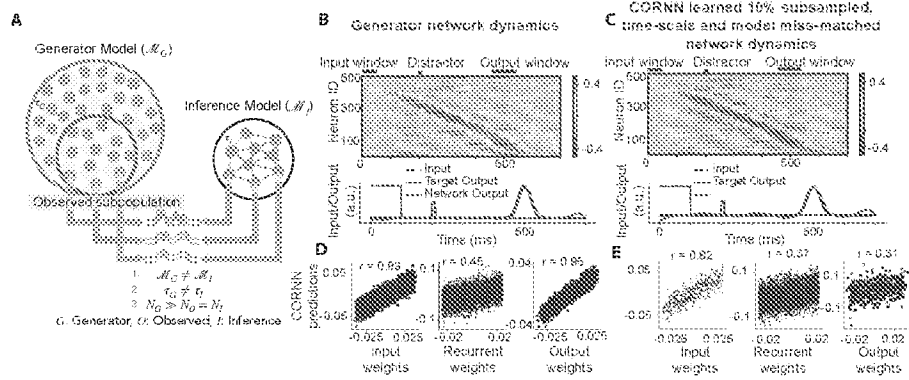


Figure 4: CORNN can reproduce underlying neural dynamics and attractor structures despite several non-idealities in the timed-response task. **A.** To assess the robustness of CORNN against time-scale mismatches, assumed inference models, and unobserved influences from subsampled neural populations, we trained 10 networks using a different generator model than CORNN for the timed-response task. **B.** We conducted a set of novel test trials to evaluate the reproduction accuracy of CORNN when the network was disturbed by a small distractor, which was not present during the learning of the timed-response task. **C.** Despite observing only 500 out of 5000 neurons and with an average 10% mismatch between α_G and α_I , the CORNN-learned network reproduced both the neural dynamics and the output. **D.** When reproducing the original network without subsampling or time-scale mismatches but with generator-inference mismatches, the learned weights exhibited imperfect, but non-zero, correlation with the ground truth. **E.** Similar to **D**, we reproduced the network shown in **C** with subsampling and time-scale mismatches. Parameters: $n_G = 5000$, $\epsilon^{RNN} \sim \mathcal{N}(0, 10^{-2})$ for 100 training trials, one example network, $\alpha_G = \frac{\Delta t}{\tau_G} = 0.1$, and $\Delta t = 1ms$. **B, D:** $n_O = 5000$, $\alpha_I = 0.1$. **C, E:** $n_O = 500$, $\alpha_I \sim \mathcal{N}(0.1, 10^{-4})$.

APPENDIX A

with $\beta \in [1.3, 2.2]$, vs. the increasing number of neurons. Bringing both observations together, we concluded that there is an $O(n_{\text{rec}}^\beta T)$ scaling of the empirical training times.

3.5 CORNN is robust to a diverse set of non-ideal conditions

To this point, we had considered ideal scenarios in which the generator network shared the same dynamical system equations as the inference network, all neurons were observed, and the time-scales of the generator and inference RNN units matched perfectly. Next, we considered cases in which these assumptions are violated (See Fig. 4A). Specifically, we trained a generator RNN with different governing equations (See Supplementary Section S3.3) to perform a timed-response task (See Fig. 4B), which put the network in a limit-cycle like dynamical attractor state initiated by an input. We tested the existence of the underlying attractor through a novel distractor input, which was not present during the training of the task (See Fig. 4B, C). If CORNN was able to learn the underlying state-space geometry, the reproduced network activities should be able to return to their attractive trajectory under novel perturbations.

When we subsampled the generator networks by 10% and introduced jitter in the time scales, we observed that the inference model reproduced the neural dynamics in the observed population in a novel perturbation trial (See Fig. 4C), even though the sub-connectivity matrix was not well reproduced (See Fig. 4D, E). On the one hand, this observation reinforces the findings of previous literature that the connectivity matrices learned by dRNNs should be interpreted as functional connections, accounting for the behavior of networks at the level of neural dynamics, and not synaptic connections [15, 25]. On the other hand, the ability of the mismatched network to reproduce neural activities in a novel perturbation trial provided evidence that the effects of the underlying attractor structures can be robustly reproduced with CORNN despite severe experimental non-idealities. We quantified these robustness aspects in Figs. 5A,B (subsampling), S10 (time-scale mismatch), and S11 (correlated noise). Moreover, we observed the n_{rec}^β scaling for the training times with increased number of network size (Fig. 5C), inline with the results of Fig. S9.

4 Discussion

To place in perspective the experimental paradigm enabled by CORNN introduced in this work, we now return to the experimental scenario in Fig. 1. Imagine an experiment in which mice perform a predefined task several times, *e.g.*, for an half hour, with imaging of 3000-4000 neurons. This experimental scenario yields roughly ten million parameters to be trained in the dRNN. The current study showcased CORNN's efficacy in training networks of thousands of neurons, requiring just $O(10)$ iterations, each comparable in complexity to gradient computation and taking seconds. Consequently, training such a network from the initial imaging session would take less than a minute. Once trained, the network can enable real-time planning of experimental interventions, testing

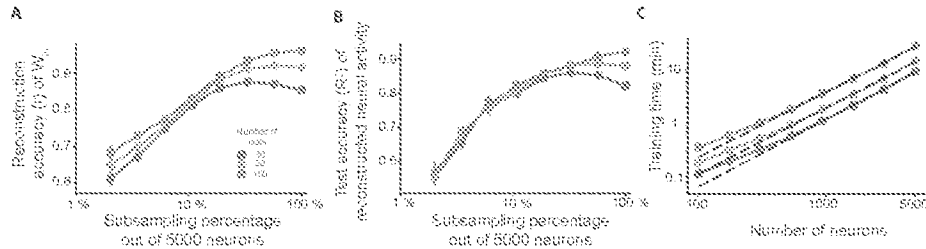


Figure 5: CORNN runtimes scale polynomially with increasing number of neurons, whereas reconstruction accuracies remain robust to subsampling. We use the timed-response task as a testbed for quantifying the robustness of CORNN to unobserved influences due to subsampling in neural populations. **A** Reconstruction accuracy of input weights are plotted as a function of subsampling ratio (fraction of observed neurons), **B** reconstruction accuracy of neural activities, measured as the R^2 between the ground truth and predicted activations, and **C** the linear scaling of the training times with asymptotic slopes $\in [1.2, 1.3]$ vs increasing number of neurons on a log-log plot, hinting at polynomial scaling. The different colors in the plot correspond to different number of training trials. Parameters: $\alpha = 0.1$, $\epsilon^{\text{RNN}} \sim \mathcal{N}(0, 10^{-2})$ for the training trials. See Supplementary Section S3.3 for further details. Data points: mean, error bars: s.e.m. over 10 networks.

APPENDIX A

multiple scenarios in parallel to identify several optimal targeting strategies, a template of potential interventions, for neuron combinations driving desired animal behavior. Moreover, with CORNN's rapid convergence and typical inter-trial intervals of a few seconds in behavioral experiments, the network and the interventional strategy can be refined between trials with incoming data streams, facilitating real-time learning.

To understand the timescales for real-time inference with a dRNN for within-trial intervention experiments using calcium imaging, we can look at typical values. Acquiring each image frame takes about 30 ms [2]. While new frames continue to be acquired, motion correction and neural activity extraction can happen in ~ 5 ms per frame [56]. In addition to these ongoing imaging steps, the simulation and decision-making with the CORNN-fitted dRNN needs to be performed. The dRNN starts with the current observed activity and can quickly simulate multiple future responses under different input scenarios, *i.e.*, potential interventions. Since the simulation involves only iterative matrix multiplications and point-wise nonlinearities, it finishes very quickly—in just a few ms. For instance, running a 1000neuron RNN forward for 10 timesteps ($\sim$ a second), for 100 different initial conditions, takes < 6 ms on a GPU, and < 25 ms on a CPU, in our hands. Once a desired intervention is identified from the simulations, the phase mask on a spatial light modulator can be updated in ~ 10 ms to optically stimulate the chosen neurons [56]. Putting all the steps together, in under 100 ms one could capture brain activity, simulate future responses, decide on an intervention, and update the optical stimulation parameters. This is fast enough for real-time closed-loop applications. By streamlining network training, CORNN provides dRNNs for these types of experiments.

An important final point in considering the use of CORNN in experimental settings comes from the fact that neural networks are, in general, non-identifiable [57]. That is, for any given settings of the parameters, there are other settings which give the same input-output function. This means that CORNN does not aim to infer the true underlying synaptic connectivity matrix from a neural activity dataset. Instead, the main use of CORNN is to infer an RNN model which recapitulates the dynamical trajectories in a neural population. CORNN may also capture the underlying attractor structure of a system. However, we caution that any claim having to do with attractor structures must be experimentally validated with perturbation experiments that directly test for attractors. In our work, we simulated such experimental validation (Figs. 4 and 5) and found that in the setting tested, CORNN was able to predict the dynamical effects of perturbations on the neural population.

5 Conclusion

In this work, we introduced a fast and scalable convex solver (CORNN) for rapid training of data-constrained recurrent neural networks. We showed that CORNN is as accurate yet faster than existing approaches by orders of magnitude and can easily scale to the large datasets of today, training in seconds to a few minutes depending on the data, network size and the availability of a GPU. CORNN, as a base model, lays the groundwork for future developments and can be improved by integrating experimentally relevant assumptions and regularizations within the ADMM framework.

When applied to simulated data from structured networks, CORNN picked up the underlying attractor structure despite non-idealities (Figs. 4, 5, S10, and S11). Inspired by this observation, further development of data-constrained RNNs can support systems neuroscience research aiming to understand inter-area interactions and can complement or augment studies of canonical or pairwise correlations toward characterizations of functional connectivity [33]. Moreover, reverse engineering the learned network may provide a deeper understanding of how neuronal populations contribute to emergent computation, cognition, and memory [12, 30]. Finally, with the advent of interventional methods such as single-cell targeted optogenetics [10], dRNNs can be trained to test causal connections computationally and supply theoretical predictions at a scale previously unachievable for single-cell targeted experimentation [8].

This work constitutes a first step towards the application of CORNN to experimental data. However, several steps remain to apply dRNNs in real-time to interventional experiments. Some example steps may include the transformation of calcium traces or spike trains into traces of firing rates normalized within $[-1, 1]$, applying the CORNN solver developed in this work into first offline and then online experimental scenarios, estimation of neuronal time-scales from the experimental data instead of tuning them as hyperparameters, and perhaps implementing a low-rank regularization approach that opens the door to interpreting the observed dynamics in terms of latent variables [26, 52].

APPENDIX A

Acknowledgements

We would like to thank Liam Storan, Udith Haputhanthri, Parth Nobel, Dr. Itamar Landau, Dr. Yoshihisa Yamamoto, Dr. Surya Ganguli, Dr. Jay Mclelland, Dr. John Duchi, and Dr. Stephen Boyd for valuable feedback and insightful discussions. FD receives funding from Stanford University's Mind, Brain, Computation and Technology program, which is supported by the Stanford Wu Tsai Neuroscience Institute. MJS gratefully acknowledges funding from the Simons Collaboration on the Global Brain and the Vannevar Bush Faculty Fellowship Program of the U.S. Department of Defense. FD expresses gratitude for the valuable mentorship he received at PHI Lab during his internship at NTT Research.

References

- [1] Christopher Langdon, Mikhail Genkin, and Tatiana A Engel. A unifying perspective on neural manifolds and circuits for cognition. *Nature Reviews Neuroscience*, pages 1–15, 2023.
- [2] Tony Hyun Kim and Mark J Schnitzer. Fluorescence imaging of large-scale neural ensemble dynamics. *Cell*, 185(1):9–41, 2022.
- [3] Anne E Urai, Brent Doiron, Andrew M Leifer, and Anne K Churchland. Large-scale neural recordings call for new insights to link brain and behavior. *Nature neuroscience*, 25(1):11–19, 2022.
- [4] Nicholas A Steinmetz, Christof Koch, Kenneth D Harris, and Matteo Carandini. Challenges and opportunities for large-scale electrophysiology with neuropixels probes. *Current opinion in neurobiology*, 50:92–100, 2018.
- [5] Nicholas James Sofroniew, Daniel Flickinger, Jonathan King, and Karel Svoboda. A large field of view two-photon mesoscope with subcellular resolution for in vivo imaging. *elife*, 5:e14472, 2016.
- [6] Sadegh Ebrahimi, Jérôme Lecoq, Oleg Rumyantsev, Tugce Tasci, Yanping Zhang, Cristina Irimia, Jane Li, Surya Ganguli, and Mark J Schnitzer. Emergent reliability in sensory cortical coding and inter-area communication. *Nature*, 605(7911):713–721, 2022.
- [7] Nicholas A Steinmetz, Cagatay Aydin, Anna Lebedeva, Michael Okun, Marius Pachitariu, Marius Bauza, Maxime Beau, Jai Bhagat, Claudia Böhm, Martijn Broux, et al. Neuropixels 2.0: A miniaturized high-density probe for stable, long-term brain recordings. *Science*, 372(6539): eabf4588, 2021.
- [8] Matthew G Perich and Kanaka Rajan. Rethinking brain-wide interactions through multi-region ‘network of networks’ models. *Current opinion in neurobiology*, 65:146–151, 2020.
- [9] Shenqin Yao, Peng Yuan, Ben Ouellette, Thomas Zhou, Marty Mortrud, Pooja Balaram, Soumya Chatterjee, Yun Wang, Tanya L Daigle, Bosiljka Tasic, et al. Recv recombinase system for in vivo targeted optogenomic modifications of single cells or cell populations. *Nature methods*, 17(4):422–429, 2020.
- [10] Kayvon Daie, Karel Svoboda, and Shaul Druckmann. Targeted photostimulation uncovers circuit motifs supporting short-term memory. *Nature Neuroscience*, 24(2):259–265, 2021.
- [11] David Sussillo, Mark M Churchland, Matthew T Kaufman, and Krishna V Shenoy. A neural network that finds a naturalistic solution for the production of muscle activity. *Nature neuroscience*, 18(7):1025–1033, 2015.
- [12] David Sussillo and Omri Barak. Opening the black box: low-dimensional dynamics in high-dimensional recurrent neural networks. *Neural computation*, 25(3):626–649, 2013.
- [13] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.
- [14] Sung Soo Kim, Hervé Rouault, Shaul Druckmann, and Vivek Jayaraman. Ring attractor dynamics in the drosophila central brain. *Science*, 356(6340):849–853, 2017.

APPENDIX A

- [15] Abhranil Das and Ila R Fiete. Systematic errors in connectivity inferred from activity in strongly recurrent networks. *Nature Neuroscience*, 23(10):1286–1296, 2020.
- [16] Mikail Khona and Ila R Fiete. Attractor and integrator networks in the brain. *Nature Reviews Neuroscience*, pages 1–23, 2022.
- [17] Sukbin Lim and Mark S Goldman. Balanced cortical microcircuitry for maintaining information in working memory. *Nature neuroscience*, 16(9):1306–1314, 2013.
- [18] Rodrigo Laje and Dean V Buonomano. Robust timing and motor patterns by taming chaos in recurrent neural networks. *Nature neuroscience*, 16(7):925–933, 2013.
- [19] Xiao-Jing Wang. Probabilistic decision making by slow reverberation in cortical circuits. *Neuron*, 36(5):955–968, 2002.
- [20] Hidehiko K Inagaki, Susu Chen, Kayvon Daie, Arseny Finkelstein, Lorenzo Fontolan, Sandro Romani, and Karel Svoboda. Neural algorithms and circuits for motor planning. *Annual Review of Neuroscience*, 45:249–271, 2022.
- [21] Daniel LK Yamins, Ha Hong, Charles F Cadieu, Ethan A Solomon, Darren Seibert, and James J DiCarlo. Performance-optimized hierarchical models predict neural responses in higher visual cortex. *Proceedings of the national academy of sciences*, 111(23):8619–8624, 2014.
- [22] Nicolas Y Masse, Guangyu R Yang, H Francis Song, Xiao-Jing Wang, and David J Freedman. Circuit mechanisms for the maintenance and manipulation of information in working memory. *Nature neuroscience*, 22(7):1159–1167, 2019.
- [23] Guangyu Robert Yang, Madhura R Joglekar, H Francis Song, William T Newsome, and Xiao-Jing Wang. Task representations in neural networks trained to perform many cognitive tasks. *Nature neuroscience*. 22(2):297–306, 2019.
- [24] Kanaka Rajan, Christopher D Harvey, and David W Tank. Recurrent network models of sequence generation and memory. *Neuron*, 90(1):128–142, 2016.
- [25] Matthew G Perich, Charlotte Arlt, Sofia Soares, Megan E Young, Clayton P Mosher, Juri Minxha, Eugene Carter, Ueli Rutishauser, Peter H Rudebeck, Christopher D Harvey, et al. Inferring brain-wide interactions using data-constrained recurrent neural network models. *bioRxiv*, pages 2020–12, 2021.
- [26] Adrian Valente, Jonathan W Pillow, and Srdjan Ostojic. Extracting computational mechanisms from neural data using low-rank rnns. *Advances in Neural Information Processing Systems*, 35: 24072–24086, 2022.
- [27] Lea Duncker and Maneesh Sahani. Dynamics on the manifold: Identifying computational dynamical activity from neural population recordings. *Current opinion in neurobiology*, 70: 163–170, 2021.
- [28] Zach Cohen, Brian DePasquale, Mikio C Aoi, and Jonathan W Pillow. Recurrent dynamics of prefrontal cortex during context-dependent decision-making. *bioRxiv*, pages 2020–11, 2020.
- [29] Arseny Finkelstein, Lorenzo Fontolan, Michael N Economo, Nuo Li, Sandro Romani, and Karel Svoboda. Attractor dynamics gate cortical information flow during decision-making. *Nature Neuroscience*, 24(6):843–850, 2021.
- [30] Valerio Mante, David Sussillo, Krishna V Shenoy, and William T Newsome. Context-dependent computation by recurrent dynamics in prefrontal cortex. *nature*, 503(7474):78–84, 2013.
- [31] Chethan Pandarinath, Daniel J O’Shea, Jasmine Collins, Rafal Jozefowicz, Sergey D Stavisky, Jonathan C Kao, Eric M Trautmann, Matthew T Kaufman, Stephen I Ryu, Leigh R Hochberg, et al. Inferring single-trial neural population dynamics using sequential auto-encoders. *Nature methods*, 15(10):805–815, 2018.
- [32] Ruxandra Barbulescu, Gonçalo Mestre, Arlindo L Oliveira, and Luís Miguel Silveira. Learning the dynamics of realistic models of c. elegans nervous system with recurrent neural networks. *Scientific Reports*, 13(1):467, 2023.

APPENDIX A

- [33] Ed Bullmore and Olaf Sporns. Complex brain networks: graph theoretical analysis of structural and functional systems. *Nature reviews neuroscience*, 10(3):186, 2009.
- [34] Karl Deisseroth. Optogenetics. *Nature methods*, 8(1):26–29, 2011.
- [35] David Sussillo and Larry F Abbott. Generating coherent patterns of activity from chaotic neural networks. *Neuron*, 63(4):544–557, 2009.
- [36] Kelly B Clancy, Aaron C Koralek, Rui M Costa, Daniel E Feldman, and Jose M Carmena. Volitional modulation of optically recorded calcium signals during neuroprosthetic learning. *Nature neuroscience*, 17(6):807–809, 2014.
- [37] Riichiro Hira, Fuki Ohkubo, Yoshito Masamizu, Masamichi Ohkura, Junichi Nakai, Takashi Okada, and Masanori Matsuzaki. Reward-timing-dependent bidirectional modulation of cortical microcircuits during optical single-neuron operant conditioning. *Nature communications*, 5(1):1–12, 2014.
- [38] Aaron C Koralek, Xin Jin, John D Long II, Rui M Costa, and Jose M Carmena. Corticostriatal plasticity is necessary for learning intentional neuroprosthetic skills. *Nature*, 483(7389):331–335, 2012.
- [39] Andrea Giovannucci, Johannes Friedrich, Matt Kaufman, Anne Churchland, Dmitri Chklovskii, Liam Paninski, and Eftychios A Pnevmatikakis. Onacid: Online analysis of calcium imaging data in real time. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL <https://proceedings.neurips.cc/paper/2017/file/4edaa105d5f53590338791951e38c3ad-Paper.pdf>.
- [40] Andrea Giovannucci, Johannes Friedrich, Pat Gunn, Jeremie Kalfon, Brandon L Brown, Sue Ann Koay, Jiannis Taxisidis, Farzaneh Najafi, Jeffrey L Gauthier, Pengcheng Zhou, Baljit S Khakh, David W Tank, Dmitri B Chklovskii, and Eftychios A Pnevmatikakis. Caiman: An open source tool for scalable calcium imaging data analysis. *eLife*, 8:e38173, 2019.
- [41] Hakan Inan, Murat A Erdogdu, and Mark Schnitzer. Robust estimation of neural signals in calcium imaging. *Advances in neural information processing systems*, 30, 2017.
- [42] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.
- [43] Hakan Inan, Claudia Schmuckermair, Tugce Tasci, Biafra O Ahanonu, Oscar Hernandez, Jérôme Lecoq, Fatih Dinç, Mark J Wagner, Murat A Erdogdu, and Mark J Schnitzer. Fast and statistically robust cell extraction from large-scale neural calcium imaging datasets. *bioRxiv*, 2021.
- [44] Zhe Chen, Garrett J Blair, Changliang Guo, Jim Zhou, Juan-Luis Romero-Sosa, Alicia Izquierdo, Peyman Golshani, Jason Cong, Daniel Aharoni, and Hugh T Blair. A hardware system for real-time decoding of in vivo calcium imaging data. *Elife*, 12:e78344, 2023.
- [45] Steven L Brunton, Joshua L Proctor, and J Nathan Kutz. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the national academy of sciences*, 113(15):3932–3937, 2016.
- [46] Kadierdan Kaheman, J Nathan Kutz, and Steven L Brunton. Sindy-pi: a robust algorithm for parallel implicit sparse identification of nonlinear dynamics. *Proceedings of the Royal Society A*, 476(2242):20200279, 2020.
- [47] Dominik Schmidt, Georgia Koppe, Zahra Monfared, Max Beutelspacher, and Daniel Durstewitz. Identifying nonlinear dynamical systems with multiple time scales and long-range dependencies. *arXiv preprint arXiv:1910.03471*, 2019.
- [48] Georgia Koppe, Hazem Toutounji, Peter Kirsch, Stefanie Lis, and Daniel Durstewitz. Identifying nonlinear dynamical systems via generative recurrent neural networks with applications to fmri. *PLoS computational biology*, 15(8):e1007263, 2019.

APPENDIX A

- [49] Ibrahim Ayed, Emmanuel de Bézenac, Arthur Pajot, Julien Brajard, and Patrick Gallinari. Learning dynamical systems from partial observations. *arXiv preprint arXiv:1902.11136*, 2019.
- [50] Daniel Durstewitz, Georgia Koppe, and Max Ingo Thurm. Reconstructing computational system dynamics from neural data with recurrent neural networks. *Nature Reviews Neuroscience*, pages 1–18, 2023.
- [51] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, Jonathan Eckstein, et al. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine learning*, 3(1):1–122, 2011.
- [52] Manuel Beiran, Alexis Dubreuil, Adrian Valente, Francesca Mastrogiuseppe, and Srdjan Ostojic. Shaping dynamics with multiple populations in low-rank recurrent networks. *Neural Computation*, 33(6):1572–1615, 2021.
- [53] Jorge Nocedal and Stephen J Wright. *Numerical optimization*. Springer, 1999.
- [54] Jian Yang, Lei Luo, Jianjun Qian, Ying Tai, Fanlong Zhang, and Yong Xu. Nuclear norm based matrix regression with applications to face recognition with occlusion and illumination changes. *IEEE transactions on pattern analysis and machine intelligence*, 39(1):156–171, 2016.
- [55] Walter G Gonzalez, Hanwen Zhang, Anna Harutyunyan, and Carlos Lois. Persistence of neuronal representations through time and damage in the hippocampus. *Science*, 365(6455):821–825, 2019.
- [56] Zihui Zhang, Lloyd E Russell, Adam M Packer, Oliver M Gauld, and Michael Häusser. Closed-loop all-optical interrogation of neural circuits in vivo. *Nature methods*, 15(12):1037–1040, 2018.
- [57] A Emin Orhan and Xaq Pitkow. Skip connections eliminate singularities. *arXiv preprint arXiv:1701.09175*, 2017.
- [58] Peter J Huber. Robust estimation of a location parameter. In *Breakthroughs in statistics*, pages 492–518. Springer, 1992.
- [59] Brian DePasquale, Christopher J Cueva, Kanaka Rajan, G Sean Escola, and LF Abbott. full-force: A target-based method for training recurrent networks. *PloS one*, 13(2):e0191527, 2018.
- [60] Haim Sompolinsky, Andrea Crisanti, and Hans-Jurgen Sommers. Chaos in random neural networks. *Physical review letters*, 61(3):259, 1988.

APPENDIX A

S1 Details on CORNN derivations and implementation**S1.1 Separation into subproblems and derivation of the gradient and Hessian**

The loss function given in Eq. (6) can be written in details as:

$$\mathcal{L}(\hat{\theta}) = \sum_{i=1}^{n_{\text{rec}}} \underbrace{\left[\frac{1}{T} \sum_{t=1}^T c_{t,i} \text{CE} \left(\frac{1 + \hat{d}_{t,i}}{2}, \frac{1 + d_{t,i}}{2} \right) \right]}_{\mathcal{L}_i(\hat{\beta}^{(i)})} + \sum_{j=1}^{n_{\text{tot}}} \frac{\lambda}{2} \hat{\theta}_{j,i}^2, \quad (\text{S1})$$

where $n_{\text{tot}} = n_{\text{rec}} + n_{\text{in}}$. We note that for a given i , the functions $\mathcal{L}_i$ depend only on $\hat{\beta}^{(i)} := \hat{\theta}_{:,i}$, i.e., the i th column of the $\hat{\theta}$ matrix. In other words, the lack of cross-talk between the columns of $\hat{\theta}$ in the loss function means each output can be regressed independently with respect to the corresponding part of the loss function, i.e., $\mathcal{L}_i(\hat{\theta}_i)$. Thus, for the rest of this section, without loss of generality, we simply considered one instance of the regression, i.e., fix i , derived the learning rule, and later generalized to the full problem.

The loss function of interest for the subproblem is:

$$\mathcal{L}(\hat{\beta}) = \frac{1}{T} \sum_{t=1}^T c_t \text{CE} \left(\frac{1 + \hat{d}_t}{2}, \frac{1 + d_t}{2} \right) + \lambda \sum_{j=1}^{n_{\text{tot}}} \frac{(\hat{\beta}_j)^2}{2}. \quad (\text{S2})$$

The gradient can be computed as:

$$\begin{aligned} (\nabla_{\hat{\beta}} \mathcal{L}(\hat{\beta}))_k &= \frac{1}{T} \sum_{t=1}^T c_t \left[\nabla_{\hat{\beta}} \text{CE} \left(\frac{1 + \hat{d}_t}{2}, \frac{1 + d_t}{2} \right) \right]_k + \lambda \hat{\beta}_k, \\ &= \frac{1}{T} \sum_{t=1}^T \frac{c_t}{2} (1 - \hat{d}_t)(1 + \hat{d}_t) \left[\frac{1 - d_t}{1 - \hat{d}_t} - \frac{1 + d_t}{1 + \hat{d}_t} \right] x_{t,k} + \lambda \hat{\beta}_k, \\ &= \frac{1}{T} \sum_{t=1}^T c_t (\hat{d}_t - d_t) x_{t,k} + \lambda \hat{\beta}_k \end{aligned} \quad (\text{S3})$$

The Hessian follows directly from the gradient and the fact that $\tanh'(x) = (1 - \tanh^2(x))$:

$$(\nabla_{\hat{\beta}}^2 \mathcal{L}(\hat{\beta}))_{kl} = \frac{1}{T} \sum_{t=1}^T c_t (1 - \hat{d}_t^2) x_{t,k} x_{t,l} + \lambda \delta_{kl}. \quad (\text{S4})$$

At this point, it is worth emphasizing that while $x_{t,i}$ are shared across all subproblems, d_t are not. In fact, for any neuron i , a different set of $d_{t,i}$ would contribute to the Hessian of the corresponding subproblem. Thus, in its current form, using the exact Hessian would require repeating matrix inversions for each of the N_{rec} neurons. We performed benchmarking with the exact Newton method. See Supplementary Section S2, though the repeated matrix inversion was no longer feasible for large networks (Fig. 3). To mitigate this, we took a fixed-point solver approach in CORNN, described below.

First, we assumed that one can initialize the full problem with an initial point, call it $\bar{d}_t = \tanh(\hat{\beta}_{\text{fp}}^T x_t)$, such that $\bar{d}_t \approx d_t$. We discussed in Supplementary Section S1.2 on how to estimate $\hat{\beta}_{\text{fp}}$. Then, we replaced the Hessian with a local approximation $\nabla_{\hat{\beta}}^2 \mathcal{L}(\hat{\beta}) \approx \nabla_{\hat{\beta}}^2 \mathcal{L}(\hat{\beta}_{\text{fp}})$ within the vicinity of $\bar{d}_t \approx d_t$, leading to the the local approximate Hessian:

$$(\nabla_{\hat{\beta}}^2 \mathcal{L}(\hat{\beta}_{\text{fp}}))_{ij} = \frac{1}{T} \sum_{t=1}^T \underbrace{\frac{1 - \hat{d}_t^2}{1 - \bar{d}_t^2}}_{\approx 1} x_{t,i} x_{t,j} + \lambda \delta_{ij} \approx \frac{1}{T} X^T X + \lambda I = H_{\text{fp}}. \quad (\text{S5})$$

From this equation, we can infer why the choice of c_t in the main text was scalable. The local Hessian depends only on the inputs, which are shared across all the n_{rec} independent subproblems, and not on

APPENDIX A

the predicted or true outputs ($\hat{d}_t$ or d_t). Moreover, it needs to be computed once, and at the beginning of the optimization only.

Using the approximate Hessian and gradient, the parameter update rule simply follows:

$$\Delta_{\text{fp}} = -H_{\text{fp}}^{-1} \nabla_{\hat{\beta}} \mathcal{L}(\hat{\beta}). \quad (\text{S6})$$

Even though $\nabla_{\hat{\beta}} \mathcal{L}(\hat{\beta})$ was a vector for the subproblem, given that the (approximate) Hessians for all subproblems are aligned to the correlation matrix, we can promote the gradient to a matrix $\nabla_{\hat{\theta}} \mathcal{L}(\hat{\theta})$ to solve all subproblems at once. Thus, one can *vectorize* the computation for each subproblem, *i.e.*, minimization of $\mathcal{L}_i(\hat{\beta}^{(i)})$ for $i = 1, \dots, N_{\text{rec}}$, without the need for an external for loop.

At this point, we note that since $H_{\text{fp}} \succeq 0$ is positive semi-definite (PSD), the direction of the parameter updates is guaranteed to be a descent direction. This is because the update direction has a negative angle with the gradient following $\Delta_{\text{fp}}^T \nabla_{\hat{\theta}} \mathcal{L}(\hat{\theta}) = -\nabla_{\hat{\theta}} \mathcal{L}(\hat{\theta})^T H_{\text{fp}}^{-1} \nabla_{\hat{\theta}} \mathcal{L}(\hat{\theta}) \leq 0$ by positive semi-definiteness. Thus, even if the error between $\hat{d}_t$ and d_t is large for an instance, *e.g.*, fixed-point is incorrectly chosen, it can only hurt the speed of the optimization. Bringing all together, fixed-point updates can be written as:

$$\theta^{(k+1)} := \theta^{(k)} - \gamma_k H_{\text{fp}}^{-1} \nabla_{\theta} \mathcal{L}(\theta^{(k)}), \quad (\text{S7})$$

where γ_k is the user-controlled learning parameter chosen either as 1 (to bypass the line search if fixed-point was close enough) or through back-tracking line search. For this work, we simply picked $\gamma_k = 1$ without any line search as we did not observe any convergence issues. We will simplify Eq. (S7) to its final form when discussing the ADMM solver in Supplementary Section S1.4.

S1.2 Fixed point initialization

To derive the fixed point update in Eq. (11), we performed a Taylor approximation of the gradient and carried out a single update near the vicinity of the putative fixed-point:

$$\begin{aligned} \hat{\beta}^{(t+1)} &:= \hat{\beta}^{(t)} - H_{\text{fp}}^{-1} \nabla_{\hat{\beta}} \mathcal{L}(\hat{\beta}^{(t)}), \\ &\approx \hat{\beta}^{(t)} - \left(\frac{1}{T} X^T X + \lambda I \right)^{-1} \left[\left(\frac{1}{T} X^T X + \lambda I \right) \hat{\beta}^{(t)} - \frac{1}{T} X^T Z \right], \\ &:= \hat{\beta}^{(t)} - \hat{\beta}^{(t)} + \left(\frac{1}{T} X^T X + \lambda I \right)^{-1} \frac{1}{T} X^T Z, \\ &:= \hat{\beta}_{\text{ls}}, \end{aligned} \quad (\text{S8})$$

where a single-step converges to $\hat{\beta}_{\text{ls}} = (X^T X + T\lambda I)^{-1} X^T Z$, *i.e.*, the solution to the least squares problem on the currents Z . Thus, $\hat{\beta}_{\text{ls}}$ is expected to be near to both the solution β^* and the regime described by the fixed-point Hessian H_{fp} ; thus a suitable candidate to be the fixed-point itself. As an added bonus, this choice did not increase the complexity of the solver; since $(X^T X + T\lambda I)^{-1} X^T$ has already been pre-computed for the iterative updates.

S1.3 Interpretation of the weighted convex loss

To interpret CORNN's loss function, we consider the low error limit of the cross entropy $\text{CE}(\hat{p}, p)$ around $\hat{p} \approx p$:

$$\begin{aligned} \text{CE}(\hat{p}, p) &= -p \log(\hat{p}) - (1-p) \log(1-\hat{p}), \\ &= -H(p) - \frac{d}{d\hat{p}} [p \log(\hat{p}) + (1-p) \log(1-\hat{p})] \Big|_{\hat{p}=p} (\hat{p} - p) \\ &\quad - \frac{d^2}{2 d\hat{p}^2 [t]} [p \log(\hat{p}) + (1-p) \log(1-\hat{p})] \Big|_{\hat{p}=p} (\hat{p} - p)^2, \\ &= -H(p) + \frac{(\hat{p} - p)^2}{2p(1-p)} + O((\hat{p} - p)^3). \end{aligned}$$

APPENDIX A

where $-H(p)$ is a constant independent of the optimization variables, and thus can be ignored. Then, the low-error limit of the CORNN loss is:

$$\begin{aligned} \frac{1}{T} \sum_{t=1}^T c_t \text{CE} \left(\frac{1 + \hat{d}_t}{2}, \frac{1 + d_t}{2} \right) &\approx \frac{1}{T} \sum_{t=1}^T \frac{(d_t - \hat{d}_t)^2}{2(1 - d_t^2)^2}, \\ &\approx \frac{1}{T} \sum_{t=1}^T \frac{(d_t - d_t - (1 - d_t^2)[\hat{z} - f^{-1}(d_t)])^2}{2(1 - d_t^2)^2}, \\ &= \frac{1}{T} \sum_{t=1}^T \frac{[\hat{z} - f^{-1}(d_t)]^2}{2}. \end{aligned} \quad (\text{S9})$$

Here, in the second row, we performed a Taylor approximation around the optimal value $\hat{d}_t \approx d_t + (\hat{d}_t)'|_{\hat{d}_t=d_t}(\hat{z} - f^{-1}(d_t))$. In this low error limit, minimizing the scalable weighted loss approximately accounts to minimizing the $\mathcal{L}_2$ loss on the currents. The main difference is, since the original weighted loss is on the firing rates and not currents, the weighted loss can account for the conversion noise, or non-linearity mismatches, that would induce bias in the explicit conversion $z = f^{-1}(d)$. The observation that low-error limit of the CORNN loss approximates the least-squares provided additional support to the claim that least-squares solution serves as a suitable initialization.

S1.4 Derivation of the ADMM updates

So far, we discussed the fixed-point solver for minimizing the unconstrained CORNN loss with L2 regularization. In this section, we discuss how to add constraints on weights. Specifically, in a general scenario, it is unexpected that all neurons would be connected to each other. Rather, in many cases, one might be interested in introducing a trainable parameter mask such that $\forall (i, j) \in B(i, j) \ W_{ij} = 0$. To account for this scenario, we incorporated our fast solver into the ADMM framework.

S1.4.1 Setting up the full problem

In this section, we will first setup the full problem of interest with constraints, and then review the ADMM equations [51]. To start with, we considered a problem of the following form:

$$\begin{aligned} \text{minimize} \quad \mathcal{L}(\hat{\beta}) &= \frac{1}{T} \sum_{t=1}^T c_t \text{CE} \left(\frac{1 + \hat{d}_t}{2}, \frac{1 + d_t}{2} \right) + \lambda \sum_{j=1}^{n_{\text{tot}}} \frac{(\hat{\beta}_j)^2}{2}, \\ \text{subject to} \quad &\forall k \in B(k), \hat{\beta}_k = 0. \end{aligned} \quad (\text{S10})$$

We transformed this problem by adding a new variable $\hat{\chi}$, which was constrained to be equal to $\hat{\beta}$:

$$\begin{aligned} \text{minimize} \quad &\frac{1}{T} \sum_{t=1}^T c_t \text{CE} \left(\frac{1 + \hat{d}_t}{2}, \frac{1 + d_t}{2} \right) + \lambda \sum_{j=1}^{n_{\text{tot}}} \frac{(\hat{\beta}_j)^2}{2}, \\ \text{subject to} \quad &\forall k \in B(k), \hat{\chi}_k = 0, \quad \hat{\chi} = \hat{\beta}. \end{aligned} \quad (\text{S11})$$

Defining the indicator function $1(x)$ as

$$1(x) = \begin{cases} 0 & \text{if } x \text{ is true,} \\ \infty & \text{if } x \text{ is false} \end{cases} \quad (\text{S12})$$

The ADMM loss function became:

$$\begin{aligned} \text{minimize} \quad \mathcal{L}_{\text{ADMM}}(\hat{\beta}, \hat{\chi}) &= \underbrace{\frac{1}{T} \sum_{t=1}^T c_t \text{CE} \left(\frac{1 + \hat{d}_t}{2}, \frac{1 + d_t}{2} \right) + \lambda \sum_{j=1}^{n_{\text{tot}}} \frac{(\hat{\beta}_j)^2}{2}}_{\mathcal{L}_{\text{CORNN}}(\hat{\beta})} + \underbrace{\sum_{k \in B(k)} 1(\hat{\chi}_k = 0)}_{\mathcal{L}_{\text{constraint}}(\hat{\chi})}, \\ \text{subject to} \quad &\hat{\chi} = \hat{\beta}. \end{aligned} \quad (\text{S13})$$

APPENDIX A

ADMM Lagrangian has two separable loss functions of independent variables, connected through a linear equality constraint, which can be augmented to obtain:

$$\mathcal{L}_\rho(\hat{\beta}, \hat{\chi}, \kappa) = \mathcal{L}_{\text{ADMM}}(\hat{\beta}, \hat{\chi}) + \kappa^T(\hat{\beta} - \chi) + \frac{\rho}{2} \|\hat{\beta} - \hat{\chi}\|_2^2, \quad (\text{S14})$$

where κ is the dual variable. The ADMM steps (after re-defining $\kappa \rightarrow (1/\rho)\kappa$) became (See Eqs. (3.5-3.7) in [51]):

$$\hat{\beta}^{k+1} := \arg \min_{\hat{\beta}} \left(\mathcal{L}_{\text{CORNN}}(\hat{\beta}) + (\rho/2) \|\hat{\beta} - \hat{\chi}^k + \kappa^k\|_2^2 \right), \quad (\text{S15a})$$

$$\hat{\chi}^{k+1} := \arg \min_{\hat{\chi}} \left(\mathcal{L}_{\text{constraint}}(\hat{\chi}) + (\rho/2) \|\hat{\beta}^{k+1} - \hat{\chi} + \kappa^k\|_2^2 \right), \quad (\text{S15b})$$

$$\kappa^{k+1} := \kappa^k + \hat{\beta}^{k+1} - \hat{\chi}^{k+1}. \quad (\text{S15c})$$

S1.4.2 Solving first ADMM iteration via fixed-point updates

Similar to the unconstrained case, we started by finding the gradient and the fixed-point Hessian:

$$\nabla_{\hat{\beta}} \left[\mathcal{L}_{\text{CORNN}}(\hat{\beta}) + (\rho/2) \|\hat{\beta} - \hat{\chi}^k + \kappa^k\|_2^2 \right] = -\frac{1}{T} X^T E + (\lambda + \rho) \hat{\beta} + \rho(\kappa^k - \hat{\chi}^k), \quad (\text{S16a})$$

$$\nabla_{\hat{\beta}}^2 \left[\mathcal{L}_{\text{CORNN}}(\hat{\beta}) + (\rho/2) \|\hat{\beta} - \hat{\chi}^k + \kappa^k\|_2^2 \right] \approx \frac{1}{T} X^T X + (\lambda + \rho) I, \quad (\text{S16b})$$

where we recall the prediction error matrix:

$$E_{t,i} = \frac{d_{t,i} - \hat{d}_{t,i}}{1 - d_{t,i}^2}. \quad (\text{S17})$$

Then, a single fixed-point update step became:

$$\begin{aligned} \Delta_{\text{fp}} &= - \left[\frac{1}{T} X^T X + (\lambda + \rho) I \right]^{-1} \left[-\frac{1}{T} X^T E + (\lambda + \rho) \hat{\beta} + \rho(\kappa^k - \hat{\chi}^k) \right] \\ &\quad - \left[X^T X + T(\lambda + \rho) I \right]^{-1} \left[X^T E - T(\lambda + \rho) \hat{\beta} + T\rho(\hat{\chi}^k - \kappa^k) \right], \\ &= A^+ X^T E - (\tilde{\lambda} + \tilde{\rho}) A^+ \hat{\beta} + \tilde{\rho} A^+ (\hat{\chi}^k - \kappa^k), \end{aligned} \quad (\text{S18})$$

with $\tilde{\lambda} = T\lambda$ and $\tilde{\rho} = T\rho$. Here, we defined the short-hand notation for the inverse matrix:

$$A^+ = \left[X^T X + (\tilde{\lambda} + \tilde{\rho}) I \right]^{-1}. \quad (\text{S19})$$

Consequently, the update rule for $\gamma_k = 1$ became:

$$\begin{aligned} \hat{\beta}^{k+1} &= \hat{\beta}^k + A^+ X^T E^k - (\tilde{\lambda} + \tilde{\rho}) A^+ \hat{\beta}^k + \tilde{\rho} A^+ (\hat{\chi}^k - \kappa^k), \\ &= A^+ X^T X \hat{\beta}^k + A^+ X^T E^k + \tilde{\rho} A^+ (\hat{\chi}^k - \kappa^k), \end{aligned} \quad (\text{S20})$$

where we highlighted all quantities that can be pre-computed with blue. For $\rho = 0$, this reproduces the update rule for the unconstrained problem in Eq. (10). As before, we promoted the vector $\hat{\beta}$ to a matrix $\hat{\theta}$, hence vectorized across the full problem.

S1.4.3 Solving second ADMM iteration via projection

For this step, the goal is to minimize

$$\begin{aligned} &\text{minimize} \sum_{j=1}^{n_{\text{tot}}} (\rho/2) (\hat{\beta}_j^{k+1} - \hat{\chi}_j + \kappa_j^k)^2, \\ &\text{subject to } \forall k \in B(k), \hat{\chi}_k = 0. \end{aligned} \quad (\text{S21})$$

Fortunately, this problem was perfectly separable across the vector entries of $\hat{\chi}_j$. For cases where $\chi_k = 0$ was constrained, this was the only feasible point and thus the answer to that particular problem. For when $j \notin B(j)$, we obtained the minimum loss value by simply picking $\hat{\chi}_j = \hat{\beta}_j^{k+1} + \kappa_j^k$. In short notation, the update rule for this subproblem became:

$$\hat{\chi}^{k+1} := \arg \min_{\hat{\chi}} \left(\mathcal{L}_{\text{constraint}}(\hat{\chi}) + (\rho/2) \|\hat{\beta}^{k+1} - \hat{\chi} + \kappa^k\|_2^2 \right) = \Pi_B \left(\hat{\beta}^{k+1} + \kappa^k \right), \quad (\text{S22})$$

where we defined Π_B as the projection operator to the constraint satisfying subspace (i.e., $\forall k \in B(k)$, $\hat{\chi}_k = 0$). Similar to before, we promoted the vector $\hat{\chi}$ to a matrix to vectorize the full problem.

APPENDIX A

S1.4.4 Variable initialization and choice of ρ

A priori, it was not clear how ρ should be chosen. However, looking at the Hessian inverse A^+ , we observed that $\rho \geq \lambda$ was an appropriate choice for the step size ρ to have a reasonable impact on the optimization problem. Empirically, we observed that $\rho \gg \lambda$ lead to faster convergence.

We initialized the primal and dual variables with $\chi^0 = \hat{\beta}^0 = \hat{\beta}_{\text{tp}}$, *i.e.* the pre-computed least-squares solution, $\hat{\beta}_{\text{tp}} = A^+ X^T Z$, without additional computational complexity. Moreover, given that ADMM ensured convergence $\hat{\chi} = \hat{\theta}$, the dual variable κ was bound to converge to non-zero values only at the diagonal such that the 2nd ADMM step would converge.

To sum up, without any additional computational cost, we defined the initial conditions for all three variables as:

$$\hat{\beta}^0 = A^+ X^T Z, \quad (\text{S23a})$$

$$\hat{\chi}^0 = A^+ X^T Z, \quad (\text{S23b})$$

$$\kappa^0 = 0. \quad (\text{S23c})$$

S1.5 Automated outlier detection and the pseudo-code for the full solver

Upon close inspection of the prediction error, we observed that the extremely large error values stemming from $d_t \approx 1$ would amplify the conversion noise and carry little-to-no learning signal. Thus, we implemented a simple outlier detection by ignoring errors larger than some value such that $|E_{t,i}| > v \implies E_{t,i} = 0$. In the CPU solver, we scaled the gradient to account for the missing data points. Since these data points constituted small percentages, mostly $< 1\%$ of the full dataset, we did not perform the scaling for the GPU based solver to prevent unnecessary overhead.

It is worth emphasizing that this choice of outlier detection takes away the convexity of the problem since zeroing out part of the prediction error leads to a non-convex global loss. However, the outliers tended to be ignored throughout the optimization process, thus the problem remained convex in remaining data points. Future work can consider a Huber-like [58] global convex robust loss that would automatically detect outliers. For the purpose of this work, we observed that the version of outlier detection we employed here performed well on empirical benchmarks.

For all solvers, we also defined a working precision $\delta (= 10^{-6})$ and whenever $|d_t| \geq |1 - \delta|$, we projected d_t back to $\pm(1 - \delta)$ to prevent numerical overflow errors. For experiments in Figs. S1 and S6 with BPTT, we projected the firing rates $|r_t| \geq |1 - \delta|$ to the boundary $r_t = \pm(1 - \delta)$ for the same reasons. Empirically, this stabilized all solvers and was performed to ensure a fair comparison.

Bringing all together, we state the full CORNN solver in Algorithm 1. We note that for the figures in this work, we simply used a maximum number of iterations (30-100 depending on experiment) instead of monitoring the convergence of CORNN; as CORNN runtimes remained sub-minute and a few more iterations than ideal were perfectly tolerable.

S2 Implementation details for other solvers**S2.1 Back-propagation through time**

We performed back-propagation through time for the experiments in Figs. S1 and S6. In our experiments, the teacher forcing means that the neural activities at time $t + 1$ is computed via Eq. (2), but we replace $\hat{x}_{t,i}$ in $\hat{z}_{t,i} = \sum_j \hat{\theta}_{t,j} \hat{x}_{t,j}$ with the data $x_{t,i}$, and $\hat{r}_{t,i}$ with $r_{t,i}$. Initially, $\hat{x}_{t,i}$ depends on $\hat{\theta}$ implicitly, so this replacement cuts off the time-dependent computation graph for the gradient. The probability of teacher forcing is the probability with which the replacement is performed at every single time point during the forward propagation of the network. Specifically, we use the following forward-propagation equations:

$$\hat{r}_{t+1,i} = \begin{cases} (1 - \alpha)r_{t,i} + \alpha f(z_{t,i}), & u < p, \\ (1 - \alpha)\hat{r}_{t,i} + \alpha f(\hat{z}_{t,i}), & u \geq p, \end{cases} \quad (\text{S24})$$

APPENDIX A

Algorithm 1: Convex and scalable solver for CORNN via ADMM framework

```

CORNN.fit ( $X, D, \lambda, \text{niter} = 30, \rho_{\text{in}} = 100$ );
%  $X$  contains the aggregated input, e.g. firing rates and input units. It is of size  $[T \times N_{\text{tot}}]$ ;
%  $D$  contains the targets. It is of size  $[T \times N_{\text{rec}}]$ ;
% To start, rescale the regularization parameter  $\lambda := T\lambda$ 
 $\rho := \rho_{\text{in}}\lambda$  %  $\rho$  is picked to be comparable to entries of  $X^T X$  and rescaled  $\lambda$ ;
% Pre-compute the inverses once during initialization;
 $A^+ := (X^T X + (\rho + \lambda)I)^{-1}$ ;
 $X^+ := A^+ X^T$ ;
 $X^- := A^+ X^T X - X^+ X$ ;
 $Z := f^{-1}(D)$  % Compute the currents from the targets ;
 $\theta_{\text{fp}} := X^+ Z$  % Initialize the fixed-point to the ls solution;
% Initialize the primary and dual variables;
 $\theta := \theta_{\text{fp}}$  % Initialize to the least-squares solution;
 $\chi := \theta_{\text{fp}}$  % Initialize to the least-squares solution;
 $u := 0 * \theta_{\text{fp}}$  % Initialize the dual variable;
for  $i = 1 : \text{niter}$ ;
    Predict the targets  $\hat{D} = f(X\theta^k)$ ;
    Compute the prediction error  $E^k = (D - \hat{D}) \oslash (1 - D \odot D)$  and perform outlier detection
    %  $\oslash/\odot$  stand for element-wise division/product ;
    Update first primal variable  $\theta^{k+1} := X^- \theta^k + X^+ E^k + \rho A^+ (\chi^k - \kappa^k)$ ;
    Update second primal variable  $\chi^{k+1} := \Pi_B (\theta^{k+1} + u^k)$ ;
    Update the dual variable  $u^{k+1} := \kappa^k + \theta^{k+1} - \chi^{k+1}$ ;
    Probe convergence by checking  $\|\chi^{k+1} - \theta^{k+1}\|$ , continue if not converged;
end
return  $\chi$ 

```

where u is a uniform random number generated between 0 and 1. It is worth emphasizing that the teacher forcing in this context is distinct from [59], where the teacher signal is acquired not from the real-data but from a second network, though the overall idea is similar.

S2.2 Current and firing rate based FORCE

We followed the approach by [25] for training the recurrent weights to reproduce target neural activities via FORCE. For both network types, we defined the prediction errors as:

$$e[t] = (\hat{r}[t] - r[t])/\alpha, \quad \text{or} \quad e[t] = \hat{x}[t] - x[t]. \quad (\text{S25})$$

Here, $\hat{r}[t]$ is the predicted firing rate at time point t , vice versa for $\hat{x}[t]$. We no longer performed any teacher forcing, thus $r[t-1]$ and $x[t-1]$ needed to predict $\hat{r}[t]$ came from the previous prediction step of the network, and were not the ground truth values.

The recurrent weights and the corresponding inverse covariance matrix $P[t]$ were updated by the following rule:

$$P[t] = P[t-1] - \frac{P[t-1]\hat{r}[t-1]\hat{r}^T[t-1]P[t-1]}{1 + \hat{r}^T[t-1]P[t-1]\hat{r}[t-1]}, \quad (\text{S26a})$$

$$W_{\text{rec}}[t] = W_{\text{rec}}[t-1] - e[t](P[t]\hat{r}[t-1])^T. \quad (\text{S26b})$$

Here, $e[t]$ was an N_{rec} dimensional vector, rather than a scalar as was the case for the original work [35]. We initialized $P[t] = \lambda^{-1}I$, where λ is the regularization parameter.

Short-comings of FORCE for dRNN training

Despite its wide use in the literature for dRNN training [24, 25, 27–29], there are several short-comings of this way of training RNNs using FORCE that render it incompatible with the realities of experiments:

APPENDIX A

1. The training procedure requires a particular range of λ to converge (See Fig. S4). Even when FORCE does converge, this is for a very limited range of λ and epochs of training.
2. FORCE is initially designed for theoretical investigations and requires the training dataset to be continuous. Even subtle jumps in the data results in stability issues with FORCE training, as the premise of FORCE approach is to have small error from the start to the end [35].

Point 1 necessitates low-level hyperparameter tuning. The only work around to point 2 is to re-initialize the inverse covariance matrix P whenever there is a jump in data points. This practically means that if we want to perform multiple training steps on the same data, each epoch simply corresponds to picking a better initialization for the next epoch, whereas all the learned correlation structure, which is stored in the inverse covariance matrix P , needs to be forgotten. The latter point showcases that FORCE is fundamentally ill-suited to many aspects of experimental data, for example, when multiple imaging sessions need to be combined.

S2.3 Newton’s method and gradient descent with single-step prediction error

We implemented Newton’s method by using the exact Hessian given in Eq. (S5) for two different scenarios: i) when $c_t = 1$ corresponding to minimizing cross entropy loss, and ii) when $c_t = [1 - d_t^2]^{-1}$ corresponding to the weighted loss. When implementing ii), we performed the outlier detection not only on the gradient, but also the Hessian level; since the division by $[1 - d_t^2]$ could have destabilized the training. Given that each subproblem had a unique Hessian for the Newton descent, we solved the separate subproblems in parallel using multiprocessing tool in numpy.

To implement the gradient descent, we used a single-layer Pytorch model with a linear + tanh structure. We implemented both a CPU and GPU version, and used the ADAM and SGD optimizers. Gradient descent was implemented for two types of loss functions: i) loss function, and ii) cross-entropy with $c_t = 1$.

S3 Benchmarking details for generator RNNs

S3.1 Randomly initialized chaotic RNNs

When benchmarking dRNNs on randomly initialized chaotic RNNs, we picked the recurrent weights of the generator randomly from a normal distribution $W_{ij}^{rec} \sim \mathcal{N}\left(0, \frac{g^2}{n_{rec}}\right)$ with zero mean and standard deviation of $\frac{g}{\sqrt{n_{rec}}}$. For $g > 1$ and sufficiently large n_{rec} , these networks can sustain their activities without any external stimulus and exhibit chaotic behavior [60]. For these chaotic generators, we picked $g = 3$ throughout this work to ensure the networks were deep into the sustainable spontaneous activity regime. We chose the chaotic regime specifically, since previous work established that teacher-forcing when training an RNN to output a function can have potentially harmful consequences to the stability of the network training [35]. We aimed to test whether this generalized to the reproduction task we considered in our work.

We note that in figures with randomly initialized chaotic RNNs, where we compared CORNN to other optimizers, we omitted the zero self-excitation constraint, i.e., $W_{ii}^{rec} = 0$, to remain conservative in our speed comparisons. This is because 2nd order methods (FORCE and Newton descent) cannot incorporate the zero self-excitation constraint without additional optimization steps since projected 2nd order steps are not necessarily descent directions, whereas projected gradient descent is notoriously slower compared to the unconstrained version. In contrast, CORNN was approximately equally fast with or without the ADMM solver; since both ADMM and HAPE updates converged in more or less the same number of iterations.

S3.2 Training of RNNs for the 3-bit flip flop task

To benchmark CORNN on diverse conditions, on top of the randomly connected RNNs discussed above, we established ground truth data by training RNNs on a 3-bit flip flop task. The architecture of the generator RNN was as follows:

$$r(t) = (1 - \alpha)r(t-1) + \alpha \tanh[W_{in}u(t) + W_{rec}r(t-1)], \quad (S27a)$$

$$o(t) = W_{out}r(t), \quad (S27b)$$

APPENDIX A

where $r(t)$ is the hidden state of the RNN at time t , α is the time-constant associated with the dynamics, W_{in} is the weight matrix governing input into the RNN, u is the input into the RNN, W_{rec} are the recurrent weights, o is the output of the RNN and W_{out} are the weights which take the hidden state to the output. Since CORNN does not learn W_{out} , to compute the outputs for the CORNN learned networks in Fig. S8, we used the same projection matrix W_{out} as the original network. See the accompanying code for further details on network training and benchmarking experiments.

S3.3 Training of RNNs for the timed-response task

To test robustness of CORNN to mismatches between generator and inference network dynamics, we established ground truth data by training leaky current RNNs on a timed-response task with the following architecture:

$$x(t) = (1 - \alpha)x(t - 1) + \alpha W_{\text{in}}u(t) + \alpha W_{\text{rec}}r(t - 1) + \alpha \epsilon_{\text{RNN}}, \quad (\text{S28a})$$

$$r(t) = \tanh(x(t)), \quad (\text{S28b})$$

$$o(t) = W_{\text{out}}r(t). \quad (\text{S28c})$$

Here, unlike the CORNN and 3-bit flip flop equations (See Eq. (S27)), the time-derivative was at the level of currents (x), not firing rates (r). Upon receiving an input during $t \in [0, 100]ms$, we trained 10 networks to output a Gaussian pulse centered at $t = 500ms$ with standard deviation $20ms$; otherwise zero everywhere.

During test trials in Figs. 5, S10, and S11, we introduced a novel distractor input ten times stronger than the cue at $t \in [400, 410]ms$. To allow fair comparison between networks trained on different numbers of neurons, we computed the R^2 of the neural activities from the first 100 neurons across three 1s trials and performed a uniform average. For the illustrative Fig. 4, to allow visualization of finer details in time-activities after distraction, we instead plotted a scenario with an earlier and weaker (half strength than the cue) distractor at $t \in [200, 210]ms$. To obtain the correlated noise in Fig. S11, we first sampled random noise from a normal distribution, then convolved the i.i.d. noise with a 2D Gaussian kernel across time points (s.d. of $3ms$) and neurons (s.d. of 5 neurons), and re-scaled by the targeted standard deviation. See accompanying code for further details.

S4 Benchmarking CORNN's speed and scalability

To obtain the CORNN's solver in Algorithm 1, we took five distinct steps:

1. *Single-step prediction error paradigm*, where we teacher-forced $\hat{r}_{t,i}$, the RNN unit activities at time t , to the observed neural activities, $r_{t,i}$, to predict $r_{t+1,i}$, with a single-step back-propagation.
2. *Convexification of the loss function*, where we replaced the $\mathcal{L}_2$ loss function with a cross-entropy loss.
3. *Hessian alignment to neural activity correlations*, where we weighted individual samples to align subproblem Hessians to a pre-computable quantity, *i.e.*, the correlation matrix of observed neural activities.
4. *Initialization to a fixed-point*, where we used the approximate least square solution to initialize the networks.
5. *Using ADMM approach* to enforce constraints without increasing algorithmic runtimes.

In this work, we performed several benchmarking experiments to delineate the contribution of each step we take towards the final convex solver to the scalability and speed of CORNN. We proved, with targeted experiments, the need for each of these steps, as we discuss below.

S4.1 Single-step prediction error paradigm

As discussed in Section 2.2, the first step we took was to restrict the family of interest for the inference models to the models that can perform single-step predictions well. This simplifying assumption mapped the recurrent estimation problem to a single layer feed-forward network with a sigmoid/tanh non-linearity. Here, the inputs of the network are $u_{t,i}$ and $r_{t,i}$, whereas the outputs are $r_{t+1,i}$. Since

APPENDIX A

this step eliminated the need for the back-propagation through time, both the training times and memory requirements decreased significantly (Compare Figs. S1 and S2).

However, a priori, it was not clear whether the restrictive assumption of single-step prediction error minimization would lead to decreased performance. To test this, we performed the experiments in Fig. S1 and observed that teacher forcing increases the efficiency of the training, and was not deleterious. A potential explanation may lie in the observation that, unlike the traditional scenario where a sparse error signal needs to be propagated back in time to perform credit assignment, network reproduction training had access to rich variety of error signals at each time step and for each neuron.

S4.2 Convexification of the loss function

As discussed in Section 2.3, the convexification of the loss function plays an important role in speeding up the learning process. However, by itself alone, convexification is not the reason of the speed improvement. For example, in Fig. 3, gradient descent on $\mathcal{L}_2$ loss converges faster than the gradient descent on the logistic loss. On the other hand, for smaller networks with 200 neurons (Figs. S2, S3, S5), convex loss function merits the use of Newton descent and leads to faster convergence than gradient descent. Yet, as shown in Figs. S2 and S5, CORNN is at least an order of magnitude faster even for these small networks. Hence, though convexification of the loss function is crucial, it cannot explain the speed or the scalability of CORNN by itself.

S4.3 Hessian alignment

The main contribution of our work is to pick $c_{t,i}$, which approximates the problem as a local least-squares as discussed in Supplementary Section S1.3, and aligns the Hessians of each subproblem to a pre-computable quantity, *i.e.*, the correlation matrix of neural activities, as discussed in Section S1. This alignment allows the use of HAPE updates that have complexity of gradient descent but converge in $O(10)$ iterations thanks to the approximate Hessian. Figs. S2 and S5 show that Newton's descent with exact Hessian computations on the weighted CORNN loss is an order of magnitude slower than CORNN. Hence, the Hessian alignment was a crucial step to speed up the learning, but most importantly allow scaling to large networks due to its gradient-level complexity.

S4.4 Fixed-point initialization

The fixed-point initialization relies on an approximate solution when the final error is low (See Supplementary Sections S1.2 and S1.3). This initialization is a good first guess, since, if there was no conversion noise at all, one could have simply performed least-squares on the currents estimated via $z = \text{arctanh}(r)$. This is akin to regressing to the inverse of the output before the non-linearity in a single layer network. Though such an approach can provide a good first guess, it is not robust to conversion noise. This was shown in Figs. 3 and S3, where non-Gaussian noise components lead to low accuracy for the initialization. Finally, the least-squares solution does not satisfy any additional constraints on the connectivity matrix, thus is not suitable as a valid solution itself. Yet, since the least-squares solution is computationally negligible, we used it as an initialization for CORNN.

S4.5 Alternating direction method of multipliers

We chose to use ADMM approach to enforce the no self-excitation constraints and present a versatile base model that can incorporate additional design choices such as L1 and/or nuclear norm regularization. In our comparison experiments, we omitted any constraints since other solvers were considerably slower, or outright incompatible. However, in our speed benchmarking results in Figs. 5, S8, and S9, we used the full CORNN solver with the ADMM described in Algorithm 1. The choice of ADMM for enforcing constraints is rooted in our observation that both the ADMM steps and the original HAPE updates take $O(10)$ iterations to converge. Moreover, other parts of the ADMM updates are negligible compared to the primal problem, where the HAPE updates take place. In other words, the fact that HAPE updates and ADMM converge in similar steps creates a mutual synergy that increases accuracy and promotes versatility in our work without hurting convergence times.

APPENDIX A

S5 Reproducibility

We provided the code to reproduce each figure in our Github. All experiments, except for those in Figures 3, 5, and S4, were run on a standard computer with Geforce RTX 3090 Ti GPU and Intel Core i9-9900X Skylake X 10-Core processor. Experiments in Fig. 5 were run on a computer with two Geforce RTX 3080 Ti GPUs (though using only one) and Intel Core i9-10980XE 18-Core processor. Experiments in Figure S4 were divided to run on both of the aforementioned computers. Experiments in Figure 3 were run on a computer with two NVIDIA Geforce RTX 4090 (though using only one) and an AMD Ryzen Threadripper PRO 5975WX 32-Core processor (though using only 5 cores due to memory limitations). Illustrative figures are plotted on an Apple Macbook Air with M1 Chip. See the accompanying code provided in the Github repo² for the specific hyperparameters used for the experiments.

²Reproduction code is available at <https://github.com/schnitzer-lab/CORNN-public>

APPENDIX A

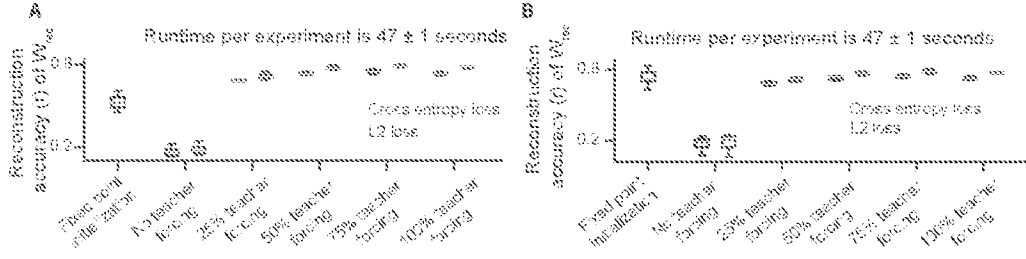


Figure S1: **Backpropagation through time (BPTT) on an $\mathcal{L}_2$ loss function is suboptimal, whereas teacher-forcing can increase the reconstruction accuracy.** To assess the optimality of BPTT on an $\mathcal{L}_2$ loss function and investigate the potential drawbacks of teacher-forcing, we conducted reproduction experiments on chaotic networks with low (A) and high (B) input to conversion noise ratio. We varied the levels of teacher forcing applied to BPTT and employed two distinct loss functions: cross-entropy and $\mathcal{L}_2$ loss function. Surprisingly, our results challenge traditional beliefs, since cross-entropy with teacher forcing emerged as the superior method compared to BPTT. This superiority may be attributed to the rich information content available during network reproduction training. Parameters: A, B: $\alpha = 0.9$, $n_{rec} = 100$, $T = 100$, $n_{epoch} = 3000$, $\forall t = 1, \dots, T$, $u(t) = 0.1$ with $W^{in} \sim \mathcal{N}(0, g^2)$ and $g = 3$. A: $\epsilon^{conv} \sim \mathcal{N}(0, 10^{-8})$, $\epsilon^{input} \sim \mathcal{N}(0, 10^{-4})$. B: $\epsilon^{conv} \sim \mathcal{N}(0, 10^{-10})$, $\epsilon^{input} \sim \mathcal{N}(0, 10^{-2})$. Box plots are obtained from 10 random generator networks shared across optimization algorithms. Orange lines denote median values; boxes span the 25th to 75th percentiles; outliers are not shown.

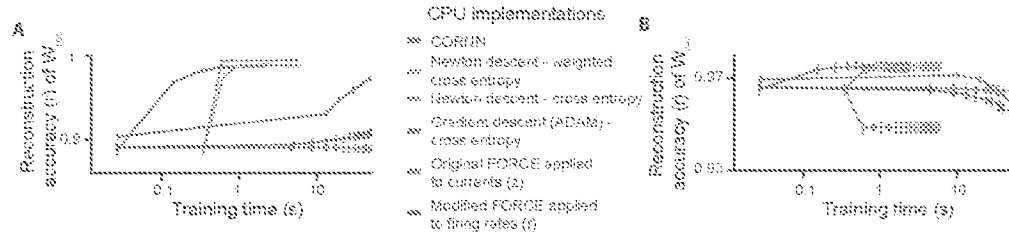


Figure S2: **CORNN achieves a better combination of speed and accuracy compared to alternatives, dramatically accelerating convergence.** The plots illustrate the relationship between accuracy (Pearson's correlation coefficient between ground truth and inferred weights) and training time, measured in seconds on a log scale, with low (A) and high (B) input to conversion noise ratio. The blue line represents the performance of our newly developed method, which combines fixed-point initialization and CORNN, and achieves high accuracy while significantly accelerating convergence compared to other methods. Parameters: A-B: $\alpha = 0.1$, $n_{rec} = 200$, $T = 3000$. No input. A: $\epsilon^{conv} \sim \mathcal{N}(0, 10^{-8})$, $\epsilon^{input} \sim \mathcal{N}(0, 10^{-4})$. B: $\epsilon^{conv} \sim \mathcal{N}(0, 10^{-10})$, $\epsilon^{input} \sim \mathcal{N}(0, 10^{-2})$. In all plots, data points indicate median values, and error bars denote standard error of the mean (SEM) over 100 runs of the simulation.

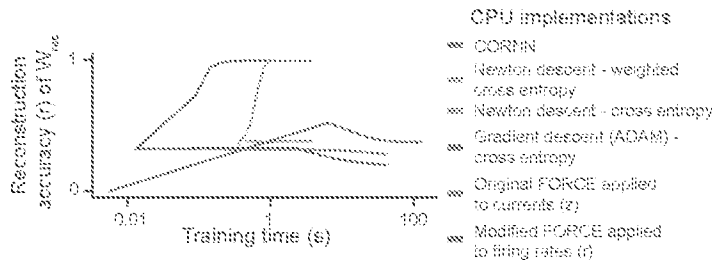


Figure S3: **CORNN is robust to non-Gaussian noise injections.** Same as in Figure S2, but with parameters: $\alpha = 0.1$, $n_{rec} = 200$, $T = 1500$. No input. $\epsilon^{conv} \sim \text{Poisson}(10^{-1})$, $\epsilon^{input} \sim \text{Poisson}(10^{-2})$.

APPENDIX A

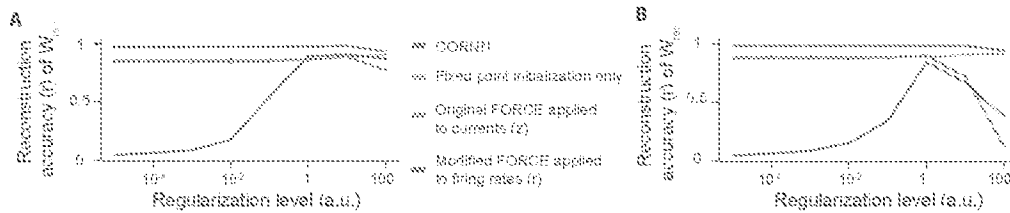


Figure S4: The traditional FORCE approach is severely suboptimal, underperforms compared to fixed point initialization, and is hyper-sensitive to the regularization level. This plot describes the relationship between the reconstruction accuracy (correlation between ground truth and inferred weights) and the strength of regularization applied to the weights during training. The results show that while the FORCE algorithm, as previously applied to a similar task [24, 25, 27–29], is highly sensitive to the choice of hyperparameters, CORNN, whose accuracy is practically bounded below by the initialization, is not. Moreover, in majority of the cases, FORCE cannot train beyond the fixed-point initialization we developed in this work. **A** with fixed point initialization and **B** with random initialization. Parameters are the same as in Fig. S2A. In all plots, data points indicate mean values, and error bars denote SEM over 10 runs of the simulation.

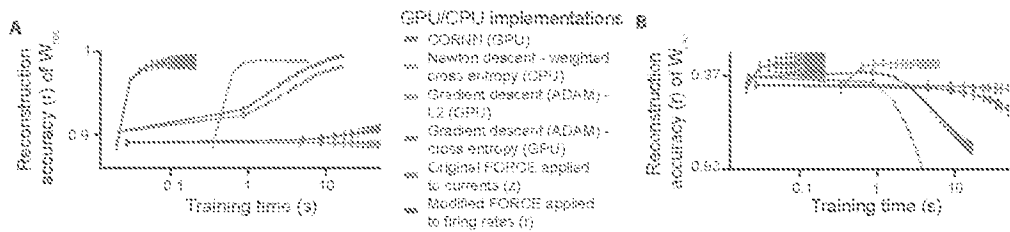


Figure S5: CORNN can be dramatically accelerated with on a standard GPU. The same as in Fig. S2, but with GPU acceleration for CORNN and gradient descent algorithms.

APPENDIX A

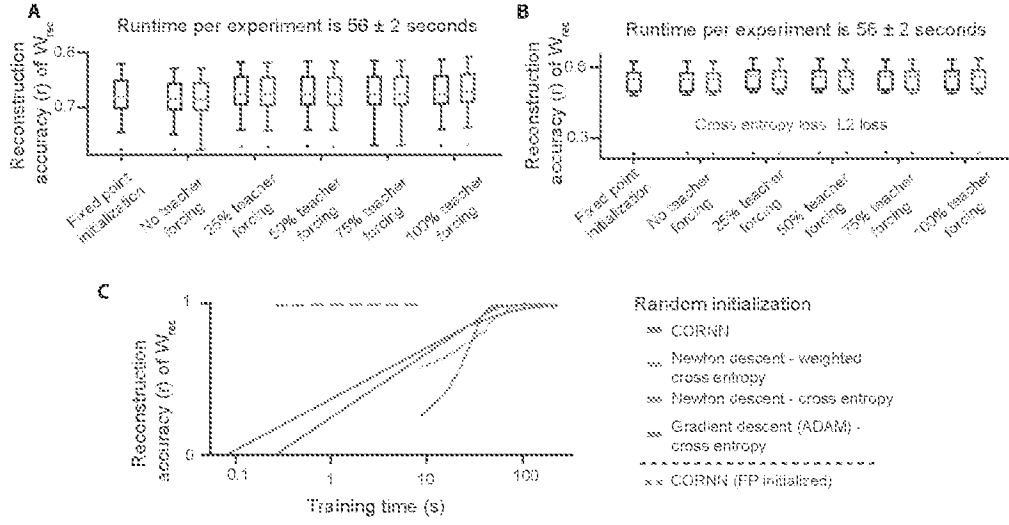


Figure S6: **Fixed-point initialization speeds up the convergence.** **A-B** Same as in Figure S1, but with fixed-point initialization. **C** We conducted an ablation study to investigate the effectiveness of our proposed fixed-point initialization method. The results show that the CORNN algorithm with fixed-point initialization (purple) outperforms all other methods, including the vanilla CORNN without this fixed-point initialization technique. Parameters: $\alpha = 0.1$, $n_{rec} = 500$, $T = 10000$, $\epsilon^{conv} \sim \mathcal{N}(0, 10^{-10})$, $\epsilon^{input} \sim \mathcal{N}(0, 10^{-2})$. All algorithms are ran on CPU. For **A,B**, the boxes cover from the lower quartile to the upper quartile, with an orange line at the median. For **C**, in all plots, data points indicate median values, and error bars denote SEM over 100 runs of the simulation.

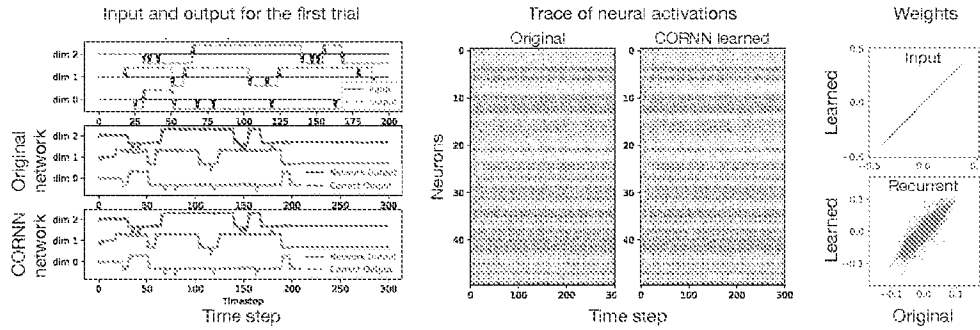


Figure S7: **CORNN can reproduce an RNN performing 3-bit flip flop task through dRNN training.** We use CORNN to reproduce the behavior of a synthetic RNN trained on a 3-bit flip-flop task [12], where the network is given a short input pulse at a random time. If the input pulse is +1, the output of the network must transition to +1 or stay at +1. If the input pulse is -1, the network's output must transition to -1 or stay at -1, ignoring input pulses that do not match. Through dRNN training, CORNN is able to accurately reproduce, *left*, the inputs and outputs of the task-trained RNN, *middle*, spatiotemporal pattern of the neural population dynamics, and, *right*, the inputs and (partially) recurrent weights of the synthetic RNN. Parameters: $\alpha = 0.9$, $n_{rec} = 1000$, $T = 200$ trials (100 data points per trial), $\epsilon^{conv} \sim \mathcal{N}(0, 10^{-6})$, $\epsilon^{input} \sim \mathcal{N}(0, 10^{-4})$.

APPENDIX A

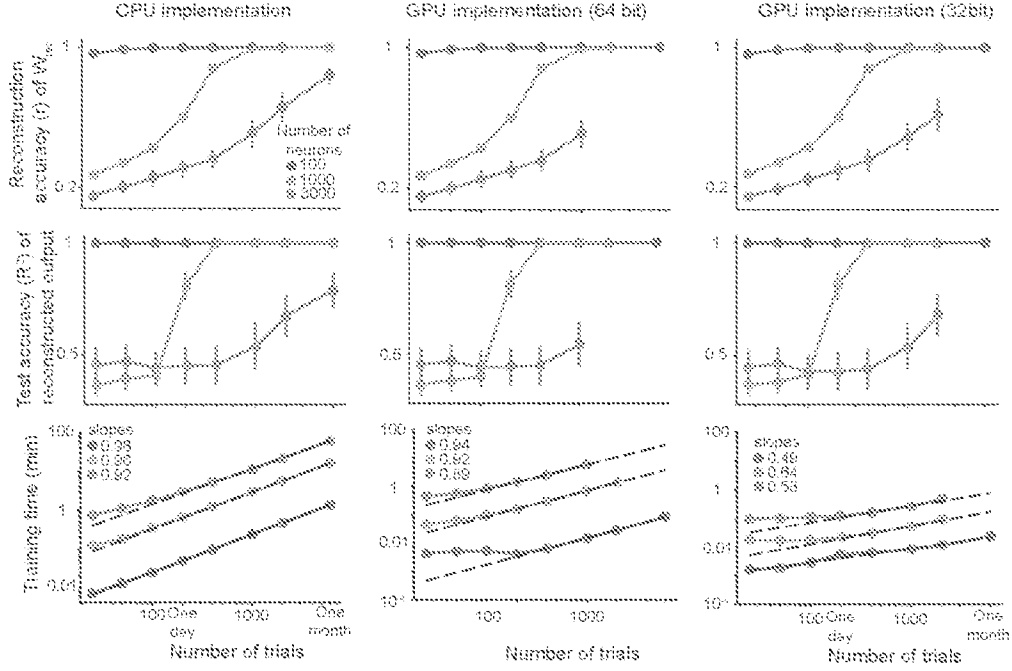


Figure S8: **CORNN runtimes scale linearly with increasing trial data, which can facilitate learning from larger networks.** *Top*, reconstruction accuracy of network weights, measured as the correlation between recurrent weights, are plotted as a function of dataset size (number of trials), *middle*, reconstruction accuracy of network outputs, measured as the R^2 between the output units, and, *bottom*, the linear scaling of algorithmic wallclock runtimes with near-unity slope vs increasing number of trials on a log-log plot. The different colors in the plot correspond to different number of neurons in the generator network. Missing data points correspond to the limits of the GPU memory, which can be mitigated by computing prediction errors in temporal chunks in real-data applications, though not implemented for this case study. Parameters: $\alpha = 0.5$, $\epsilon^{\text{conv}} \sim \mathcal{N}(0, 10^{-6})$, $\epsilon^{\text{input}} \sim \mathcal{N}(0, 10^{-4})$. In all plots, data points indicate mean values, and error bars denote SEM over 20 runs of the simulation.

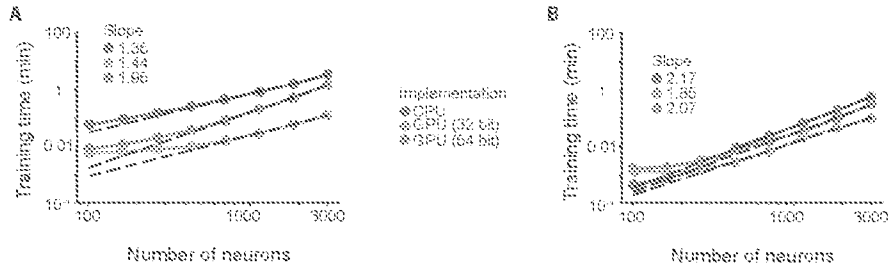


Figure S9: **CORNN scales polynomially to large network sizes.** Using the networks with 3000 neurons trained for the 3-bit flip flop task in Figure S7, we subsampled neural populations and collected CORNN runtimes for CPU and GPU implementations. Parameters: $\alpha = 0.5$, $\epsilon^{\text{conv}} \sim \mathcal{N}(0, 10^{-6})$, $\epsilon^{\text{input}} \sim \mathcal{N}(0, 10^{-4})$. **A** with 200 trials (20,000 data points) and **B** with a single trial (100 data points).

APPENDIX A

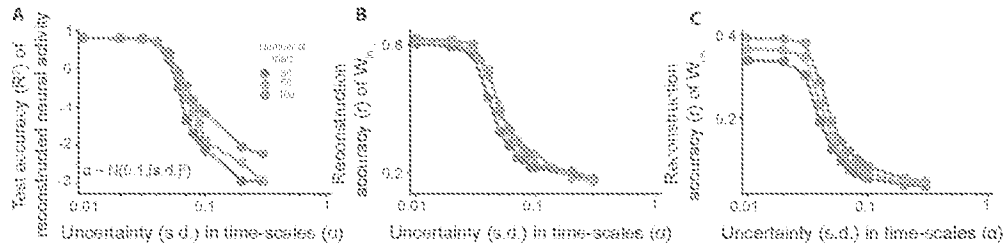


Figure S10: **CORNN is robust to variations in the time-scales compared to the ground truth.** Same as in Figure 5, but with 500 observed neurons and varying levels of uncertainties in time-scales.

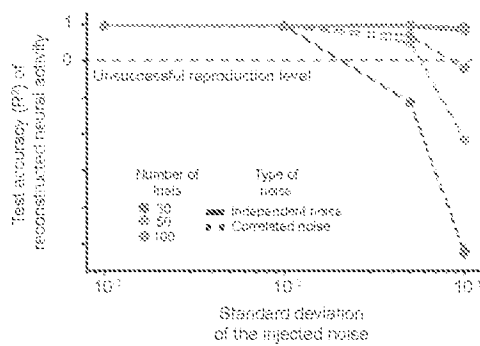


Figure S11: **Correlated noise presents a challenge for CORNN, which can be mitigated by increasing the data size.** Same as in Figure 5, but with the fully observed network and varying levels of injected random and correlated noise (for details, see Section S3.3).

APPENDIX A

CODE FOR EXPERIMENT 1

optimize_variables.py

#!/usr/bin/env python3

-*- coding: utf-8 -*-

"""

Created on Wed Dec 21 17:52:09 2022

@author: dinc

"""

```
import numpy as np
from scipy.stats import pearsonr
from rnn_class import RNN
import matplotlib.pyplot as plt
from numpy.linalg import inv
from utils import solve_cornn_gpu
from utils import solve_cornn
from utils import solve_newton_descent
from utils import solve_gradient_descent
from utils import solve_pytorch_gpu
from utils import solve_pytorch
from utils import fit_FORCE
from utils import fit_FORCE_gpu
```

```
opts = {}
opts['g'] = 3;
opts['n_rec'] = 200
opts['n_in'] = 0
opts['sigma_input'] = 1e-2
opts['sigma_conversion'] = 1e-4
T_data = 3000;
m1 = RNN(opts)
r = m1.get_time_evolution(T = T_data)
gnd = m1.rnn['w_rec'].flatten()
```

```
#####
w = solve_cornn(r[:-1,:],r[1:,:],u_in = None,l2 =
1*1e-5,
               verbose = 2,gnd = gnd, threshold = 1,
               num_iters = 30,check_convergence = 0,
               solver_type = 'weighted',alph = 0.1)
w_rec_co = w[:opts['n_rec'],:].T.flatten()
```

```
#####
w3 = solve_pytorch_gpu(r[:-1,:],r[1:,:],u_in =
None,alph = 0.1,
                    l2 = 1e-8,verbose = 2,gnd = gnd,
                    initialize_fp = 0,num_iters = 5000,
                    learning_rate = 0.01,solver_type =
'logistic')
```

```
w_rec_pt = w3[:opts['n_rec'],:].T.flatten()
```

#####

```
w1 = solve_newton_descent(r[:-1,:],r[1:,:],u_in =
None,alph = 0.1,
                        l2 = 1e-5,verbose = 1,threshold =0.2,
                        initialize_fp = 0,num_iters = 10,
                        solver_type = 'weighted')
```

```
w_rec_nt = w1[:opts['n_rec'],:].T.flatten()
```

#####

```
w4 = fit_FORCE_gpu(r,None,alph = 0.1,
                  lam = 100,g_in = 2,verbose = 2,
                  initialize_fp = 1,num_iters = 100,
                  gnd = gnd,solver_type = 'firing_rates')
```

```
w_rec_fc = w4[:opts['n_rec'],:].T.flatten()
```

#####

```
plt.scatter(gnd,w_rec_pt)
plt.scatter(gnd,w_rec_co)

print('NT:', pearsonr(w_rec_nt,gnd)[0])
print('GD:', pearsonr(w_rec_pt,gnd)[0])
print('CoRNN:', pearsonr(w_rec_co,gnd)[0])
```

rnn_class.py

#!/usr/bin/env python3

-*- coding: utf-8 -*-

"""

Created on Fri Jun 10 14:48:40 2022

@author: dinc

"""

```
import numpy as np
import multiprocessing
from joblib import Parallel, delayed
from scipy.stats import pearsonr
from numpy.linalg import solve
from scipy.linalg import solve as solve2
```

```
import time as time_now
```

```
class RNN():
```

```
    # Define some global variables that are mostly
    shared across all mice
```

APPENDIX A

```

def __init__(self,opts=None):
    self.opts = self.get_defaults(opts);
    self.initialize_network()

    self.rnn = nn

def get_defaults(self,opts = None):
    if opts is None:
        opts = {};

    if 'n_rec' not in opts:
        opts['n_rec'] = 30

    if 'n_in' not in opts:
        opts['n_in'] = 0

    if 'g' not in opts:
        opts['g'] = 3

    if 'sigma_input' not in opts:
        opts['sigma_input'] = 0

    if 'sigma_conversion' not in opts:
        opts['sigma_conversion'] = 0

    if 'alpha' not in opts:
        opts['alpha'] = 0.1

    if 'input_noise_type' not in opts:
        opts['input_noise_type'] = 'Gaussian'

    if 'conversion_noise_type' not in opts:
        opts['conversion_noise_type'] = 'Gaussian'

    return opts

def initialize_network(self):
    n_rec = self.opts['n_rec']
    n_in = self.opts['n_in']
    g = self.opts['g']

    # Initialize the input weights
    nn={}
    if self.opts['n_in']>0:
        nn['w_in']=np.random.normal(0,g/np.sqrt(n_in),n_in*n_rec).reshape(n_rec,n_in)

    # Initialize the recurrent weights
    temp=np.random.normal(0,g/np.sqrt(n_rec),n_rec*n_rec).reshape(n_rec,n_rec)
    np.fill_diagonal(temp,0)
    nn['w_rec']=temp

def get_time_evolution(self,r_in = None,u = None,T = None):
    if T is None:
        T = 1000
    if r_in is None:
        r_in = np.random.uniform(-1,1,self.opts['n_rec'])
    if u is None:
        if self.opts['n_in']>0:
            u = np.zeros([T,self.opts['n_in']])

    if self.opts['n_in']>0:
        w_in = self.rnn['w_in']

    w_rec = self.rnn['w_rec']
    alpha = self.opts['alpha']
    sigma_input = self.opts['sigma_input']
    sigma_conversion = self.opts['sigma_conversion']

    n_rec = w_rec.shape[0]

    r = np.zeros([T+1,n_rec])

    r[0,:]= r_in
    for i in range(T):
        r_temp=r[i,:];

        if self.opts['input_noise_type'] == 'Gaussian':
            noise_input = np.random.normal(0,sigma_input,n_rec)
        elif self.opts['input_noise_type'] == 'Laplace':
            noise_input = np.random.laplace(0,sigma_input,n_rec)
        elif self.opts['input_noise_type'] == 'Poisson':
            noise_input = np.random.poisson(sigma_input,n_rec)

        if self.opts['n_in']>0:
            z = np.dot(w_rec,r_temp) + np.dot(w_in,u[i,:]) \
                + noise_input;

```

APPENDIX A

```

else:
    z = np.dot(w_rec, r_temp) + noise_input;

    if self.opts['conversion_noise_type'] ==
'Gaussian':
        noise_conversion =
np.random.normal(0, sigma_conversion, n_rec)
    elif self.opts['conversion_noise_type'] ==
'Laplace':
        noise_conversion =
np.random.laplace(0, sigma_conversion, n_rec)
    elif self.opts['conversion_noise_type'] ==
'Poisson':
        noise_conversion =
np.random.poisson(sigma_conversion, n_rec)

    temp_val = alpha*np.tanh(z) + (1-
alpha)*r_temp + alpha*noise_conversion;

    # Newly added error could throw firing rates
out of bounds, correct for it!
    #temp_val[temp_val >= 1] = 1;
    #temp_val[temp_val <= -1] = -1;

    r[i+1,:] = temp_val;

return r

def run_forward_propagation(self, r, u = None):
    """
    When inputting r, make sure you input it as r[:-
    1,:], as we want to compute the firing rates
    at time t+1, but r[T+1,:] is not defined, as r is
    defined between 0 and T, both inclusive!
    """
    T = r.shape[0]
    if u is not None and u.shape[0] != T:
        raise Exception('There is a mismatch
between u and r dimensions!!!')

    if self.opts['n_in'] > 0:
        w_in = self.rnn['w_in']
        w_rec = self.rnn['w_rec']
        alpha = self.opts['alpha']

    if self.opts['n_in'] > 0:
        z = np.dot(r, w_rec.T) + np.dot(u, w_in.T)
    else:
        z = np.dot(r, w_rec.T)

```

```

r_out = alpha*np.tanh(z) + (1-alpha)*r;
return r_out

```

run_experiment_1a.py

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""

```

Created on Wed Dec 21 17:52:09 2022

```

@author: dinc
"""

```

```

import numpy as np
from scipy.stats import pearsonr
from scipy.stats import spearmanr
from rnn_class import RNN
import matplotlib.pyplot as plt
from numpy.linalg import inv
from utils import solve_cornn
from utils import solve_cornn_gpu
from utils import solve_newton_descent
from utils import solve_gradient_descent
from utils import solve_pytorch
from utils import solve_pytorch_gpu
from utils import fit_FORCE
import time as time_now

```

```
time_limit = 120;
```

```

opts = {};
opts['g'] = 3;
opts['n_rec'] = 200 # 300 for short
opts['n_in'] = 1
opts['sigma_input'] = 1e-2
opts['sigma_conversion'] = 1e-4
opts['alpha'] = 0.1
opts['input_noise_type'] = 'Gaussian'
opts['conversion_noise_type'] = 'Gaussian'
opts['verbose'] = False;
opts['lambda_reg'] = 1e-5
opts['num_cores'] = 4
opts['parallel'] = 1
T_data = 3000; # 3000 for short

```

```

num_algs = 9;
num_exps = 100;
iter_list = np.linspace(0, 20, 21).astype(int)
slopes =
np.zeros([num_algs, num_exps, iter_list.shape[0]])
+ np.nan

```

APPENDIX A

```

rmse =
np.zeros([num_algs,num_exps,iter_list.shape[0]])
+ np.nan
rmedse =
np.zeros([num_algs,num_exps,iter_list.shape[0]])
+ np.nan
cor_p =
np.zeros([num_algs,num_exps,iter_list.shape[0]])
+ np.nan
cor_s =
np.zeros([num_algs,num_exps,iter_list.shape[0]])
+ np.nan
times =
np.zeros([num_algs,num_exps,iter_list.shape[0]])
+ np.nan

for idx_exp in range(num_exps):
    counter = np.zeros(num_algs)
    temp = time_now.localtime()
    current_time = time_now.strftime("%H:%M:%S",
temp)
    print('%s: Running Experiment %d'
%(current_time,idx_exp))
    m1 = RNN(opts)
    r = m1.get_time_evolution(T = T_data)
    gnd = m1.rnn['w_rec'].flatten()
    #%%
    for iters in iter_list:
        temp = time_now.localtime()
        current_time =
time_now.strftime("%H:%M:%S", temp)
        print('\t %s: Running iteration %d'
%(current_time,iters))

        if iters == 0:
            try:
                w = solve_newton_descent(r[:-
1,:],r[1:,:],u_in = None,alpha = 0.1,
l2 = 1e-5,verbose = 0,threshold
=0.2,
initialize_fp = 1,num_iters = iters,
solver_type = 'weighted')
            except:
                w = 0

            start_time = time_now.perf_counter()
            if counter[3] == 0:
                try:
                    w = solve_newton_descent(r[:-
1,:],r[1:,:],u_in = None,alpha = 0.1,
l2 = 1e-5,verbose = 0,threshold
=0.2,

```

```

initialize_fp = 1,num_iters =
iters,
solver_type = 'weighted')
times[3,idx_exp,iters] =
time_now.perf_counter() - start_time;
w_rec = w[:opts['n_rec'],:].T.flatten()
slopes[3,idx_exp,iters] = (gnd @ w_rec) /
(gnd@gnd)
rmse[3,idx_exp,iters] = np.sqrt(np.mean(
(w_rec-gnd)**2))
rmedse[3,idx_exp,iters] =
np.sqrt(np.median( (w_rec-gnd)**2 ))
cor_p[3,idx_exp,iters] =
pearsonr(w_rec,gnd)[0]
cor_s[3,idx_exp,iters] =
spearmanr(w_rec,gnd)[0]
print('\t \t \t NT,weighted. Cor %.3f'
%(cor_p[3,idx_exp,iters]))
if time_now.perf_counter() - start_time >
time_limit:
    counter[3] = 1
except:
    w = 0
else:
    print('\t \t \t NT, weighted skipped due to
timeout error ')

start_time = time_now.perf_counter()
if counter[5] == 0:
    try:
        w = solve_newton_descent(r[:-
1,:],r[1:,:],u_in = None,alpha = 0.1,
l2 = 1e-5,verbose = 0,threshold
=0.2,
initialize_fp = 1,num_iters =
iters,
solver_type = 'logistic')
times[5,idx_exp,iters] =
time_now.perf_counter() - start_time;
w_rec = w[:opts['n_rec'],:].T.flatten()
slopes[5,idx_exp,iters] = (gnd @ w_rec) /
(gnd@gnd)
rmse[5,idx_exp,iters] = np.sqrt(np.mean(
(w_rec-gnd)**2))
rmedse[5,idx_exp,iters] =
np.sqrt(np.median( (w_rec-gnd)**2 ))
cor_p[5,idx_exp,iters] =
pearsonr(w_rec,gnd)[0]
cor_s[5,idx_exp,iters] =
spearmanr(w_rec,gnd)[0]

```

APPENDIX A

```

        print('\t\t\t NT,logistic. Cor %.3f'
%(cor_p[5,idx_exp,itors]))
        if time_now.perf_counter() - start_time >
time_limit:
            counter[5] = 1
        except:
            print('\t\t\t NT,logistic terminated due to
timeout error')
            counter[5] = 1
        else:
            print('\t\t\t NT,logistic skipped due to
timeout error')

###
start_time = time_now.perf_counter()
if counter[0] == 0:
    try:

        w = solve_corrnr(r[:-1,:],r[1:,:],u_in =
None,l2 = 1*1e-5,
            verbose = 0,gnd = gnd, threshold =
0.2,
                num_iters =
5*itors,check_convergence = 0,
                solver_type = 'weighted')
        times[0,idx_exp,itors] =
time_now.perf_counter() - start_time;
        w_rec = w[:opts['n_rec'],:].T.flatten()
        slopes[0,idx_exp,itors] = (gnd @ w_rec) /
(gnd@gnd)
        rmse[0,idx_exp,itors] = np.sqrt(np.mean(
(w_rec-gnd)**2))
        rmedse[0,idx_exp,itors] =
np.sqrt(np.median( (w_rec-gnd)**2))
        cor_p[0,idx_exp,itors] =
pearsonr(w_rec,gnd)[0]
        cor_s[0,idx_exp,itors] =
spearmanr(w_rec,gnd)[0]
        print('\t\t\t CoRNN, CPU. Cor %.3f'
%(cor_p[0,idx_exp,itors]))
        if time_now.perf_counter() - start_time >
time_limit:
            counter[0] = 1
        except:
            print('\t\t\t CoRNN, CPU terminated due
to timeout error')
            counter[0] = 1

```

```

    else:
        print('\t\t\t CoRNN, CPU skipped due to
timeout error')

    ###
    if iters == 0:
        w = solve_corrnr_gpu(r[:-1,:],r[1:,:],u_in =
None,l2 = 1*1e-5,
            verbose = 0,gnd = gnd, threshold =
0.2,
                num_iters = 5*itors,
                solver_type = 'weighted')
        start_time = time_now.perf_counter()
        if counter[8] == 0:
            try:

                w = solve_corrnr_gpu(r[:-1,:],r[1:,:],u_in =
None,l2 = 1*1e-5,
                    verbose = 0,gnd = gnd, threshold =
0.2,
                        num_iters = 5*itors,
                        solver_type = 'weighted')
                times[8,idx_exp,itors] =
time_now.perf_counter() - start_time;
                w_rec = w[:opts['n_rec'],:].T.flatten()
                slopes[8,idx_exp,itors] = (gnd @ w_rec) /
(gnd@gnd)
                rmse[8,idx_exp,itors] = np.sqrt(np.mean(
(w_rec-gnd)**2))
                rmedse[8,idx_exp,itors] =
np.sqrt(np.median( (w_rec-gnd)**2))
                cor_p[8,idx_exp,itors] =
pearsonr(w_rec,gnd)[0]
                cor_s[8,idx_exp,itors] =
spearmanr(w_rec,gnd)[0]
                print('\t\t\t CoRNN, GPU. Cor %.3f'
%(cor_p[8,idx_exp,itors]))
                if time_now.perf_counter() - start_time >
time_limit:
                    counter[8] = 1
                except:
                    print('\t\t\t CoRNN, GPU terminated due
to timeout error')
                    counter[8] = 1
                else:
                    print('\t\t\t CoRNN, GPU skipped due to
timeout error')

            ###
            start_time = time_now.perf_counter()
            if counter[1] == 0:

```

APPENDIX A

```

try:

    w = solve_pytorch_gpu(r[:-1,:],r[1:,:],u_in =
None,alpha = 0.1,
                        l2 = 1e-8,verbose = 0,gnd =
gnd, #1e-7 for short
                        initialize_fp = 1,num_iters =
iters*1000,
                        learning_rate = .01,solver_type
= 'logistic')
    times[1,idx_exp,iters] =
time_now.perf_counter() - start_time;
    w_rec = w[:opts['n_rec'],:].T.flatten()
    slopes[1,idx_exp,iters] = (gnd @ w_rec) /
(gnd@gnd)
    rmse[1,idx_exp,iters] = np.sqrt(np.mean(
(w_rec-gnd)**2 ))
    rmedse[1,idx_exp,iters] =
np.sqrt(np.median( (w_rec-gnd)**2 ))
    cor_p[1,idx_exp,iters] =
pearsonr(w_rec,gnd)[0]
    cor_s[1,idx_exp,iters] =
spearmanr(w_rec,gnd)[0]
    print('\t \t \t Pytorch,logistic,GPU. Cor
%.3f' %(cor_p[1,idx_exp,iters]))
    if time_now.perf_counter() - start_time >
time_limit:
        counter[1] = 1
    except:
        print('\t \t \t Pytorch,logistic,GPU
terminated due to timeout error')
        counter[1] = 1
    else:
        print('\t \t \t Pytorch,logistic,GPU skipped
due to timeout error')

    #%%
    start_time = time_now.perf_counter()
    if counter[2] == 0:
        try:

            w = solve_pytorch(r[:-1,:],r[1:,:],u_in =
None,alpha = 0.1,
                            l2 = 1e-8,verbose = 0,gnd =
gnd,
                            initialize_fp = 1,num_iters =
iters * 1000,
                            learning_rate = .01,solver_type
= 'logistic')
            times[2,idx_exp,iters] =
time_now.perf_counter() - start_time;
            w_rec = w[:opts['n_rec'],:].T.flatten()

```

```

            slopes[2,idx_exp,iters] = (gnd @ w_rec) /
(gnd@gnd)
            rmse[2,idx_exp,iters] = np.sqrt(np.mean(
(w_rec-gnd)**2 ))
            rmedse[2,idx_exp,iters] =
np.sqrt(np.median( (w_rec-gnd)**2 ))
            cor_p[2,idx_exp,iters] =
pearsonr(w_rec,gnd)[0]
            cor_s[2,idx_exp,iters] =
spearmanr(w_rec,gnd)[0]
            print('\t \t \t Pytorch,logistic, CPU. Cor
%.3f' %(cor_p[2,idx_exp,iters]))
            if time_now.perf_counter() - start_time >
time_limit:
                counter[2] = 1
            except:
                print('\t \t \t Pytorch,logistic, CPU
terminated due to timeout error')
                counter[2] = 1
            else:
                print('\t \t \t Pytorch,logistic, CPU skipped
due to timeout error')

            #%%
            start_time = time_now.perf_counter()
            if counter[4] == 0:
                try:

                    w = solve_pytorch_gpu(r[:-1,:],r[1:,:],u_in =
None,alpha = 0.1,
                                            l2 = 1e-8,verbose = 0,gnd =
gnd, #1e-7 for short
                                            initialize_fp = 1,num_iters =
iters*1000,
                                            learning_rate = .01,solver_type
= 'l2')
                    times[4,idx_exp,iters] =
time_now.perf_counter() - start_time;
                    w_rec = w[:opts['n_rec'],:].T.flatten()
                    slopes[4,idx_exp,iters] = (gnd @ w_rec) /
(gnd@gnd)
                    rmse[4,idx_exp,iters] = np.sqrt(np.mean(
(w_rec-gnd)**2 ))
                    rmedse[4,idx_exp,iters] =
np.sqrt(np.median( (w_rec-gnd)**2 ))
                    cor_p[4,idx_exp,iters] =
pearsonr(w_rec,gnd)[0]
                    cor_s[4,idx_exp,iters] =
spearmanr(w_rec,gnd)[0]
                    print('\t \t \t Pytorch,l2,GPU. Cor %.3f'
%(cor_p[4,idx_exp,iters]))

```

APPENDIX A

```

        if time_now.perf_counter() - start_time >
time_limit:
            counter[4] = 1
        except:
            print('\t\t\t Pytorch,l2,GPU terminated
due to timeout error')
            counter[4] = 1
        else:
            print('\t\t\t Pytorch,l2,GPU skipped due to
timeout error')

###
start_time = time_now.perf_counter()
if counter[6] == 0:
    try:

        w = fit_FORCE(r,None,alpha = 0.1,
            lam = 200,g_in = 2,verbose = 0,
            initialize_fp = 1,num_iters = 5*iters,
            gnd = gnd,solver_type = 'currents')
        times[6,idx_exp,iters] =
time_now.perf_counter() - start_time;
        w_rec = w[:opts['n_rec'],:].T.flatten()
        slopes[6,idx_exp,iters] = (gnd @ w_rec) /
(gnd@gnd)
        rmse[6,idx_exp,iters] = np.sqrt(np.mean(
(w_rec-gnd)**2 ))
        rmedse[6,idx_exp,iters] =
np.sqrt(np.median( (w_rec-gnd)**2 ))
        cor_p[6,idx_exp,iters] =
pearsonr(w_rec,gnd)[0]
        cor_s[6,idx_exp,iters] =
spearmanr(w_rec,gnd)[0]
        print('\t\t\t Force,currents. Cor %.3f'
%(cor_p[6,idx_exp,iters]))
        if time_now.perf_counter() - start_time >
time_limit:
            counter[6] = 1
        except:
            print('\t\t\t Force,currents terminated due
to timeout error')
            counter[6] = 1
        else:
            print('\t\t\t Force,currents skipped due to
timeout error')

###
start_time = time_now.perf_counter()
if counter[7] == 0:
    try:

```

```

        w = fit_FORCE(r,None,alpha = 0.1,
            lam = 200,g_in = 2,verbose = 0,
            initialize_fp = 1,num_iters = 5*iters,
            gnd = gnd,solver_type =
'firing_rates')
        times[7,idx_exp,iters] =
time_now.perf_counter() - start_time;
        w_rec = w[:opts['n_rec'],:].T.flatten()
        slopes[7,idx_exp,iters] = (gnd @ w_rec) /
(gnd@gnd)
        rmse[7,idx_exp,iters] = np.sqrt(np.mean(
(w_rec-gnd)**2 ))
        rmedse[7,idx_exp,iters] =
np.sqrt(np.median( (w_rec-gnd)**2 ))
        cor_p[7,idx_exp,iters] =
pearsonr(w_rec,gnd)[0]
        cor_s[7,idx_exp,iters] =
spearmanr(w_rec,gnd)[0]
        print('\t\t\t Force, firing rates. Cor %.3f'
%(cor_p[7,idx_exp,iters]))
        if time_now.perf_counter() - start_time >
time_limit:
            counter[7] = 1
        except:
            print('\t\t\t Force, firing rates terminated
due to timeout error')
            counter[7] = 1
        else:
            print('\t\t\t Force, firing rates skipped due to
timeout error')

        if iters == 0:
            counter = np.zeros(num_aigs)

np.savez('experiment1a_results_fp.npz',slopes =
slopes,
        rmse = rmse, rmedse = rmedse,cor_p =
cor_p,cor_s = cor_s,
        times = times)

run_experiment_1b.py
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Dec 21 17:52:09 2022

@author: dinc
"""

import numpy as np
from scipy.stats import pearsonr

```

APPENDIX A

```

from scipy.stats import spearmanr
from rnn_class import RNN
import matplotlib.pyplot as plt
from numpy.linalg import inv
from utils import solve_corrn
from utils import solve_corrn_gpu
from utils import solve_newton_descent
from utils import solve_gradient_descent
from utils import solve_pytorch
from utils import solve_pytorch_gpu
from utils import fit_FORCE
import time as time_now

time_limit = 120;

opts = {};
opts['g'] = 3;
opts['n_rec'] = 200 # 300 for short
opts['n_in'] = 1
opts['sigma_input'] = 1e-1
opts['sigma_conversion'] = 1e-5
opts['alpha'] = 0.1
opts['input_noise_type'] = 'Gaussian'
opts['conversion_noise_type'] = 'Gaussian'
opts['verbose'] = False;
opts['lambda_reg'] = 1e-5
opts['num_cores'] = 4
opts['parallel'] = 1
T_data = 3000; # 3000 for short

num_algs = 9;
num_exps = 100;
iter_list = np.linspace(0,20,21).astype(int)
slopes =
np.zeros([num_algs,num_exps,iter_list.shape[0]])
+ np.nan
rmse =
np.zeros([num_algs,num_exps,iter_list.shape[0]])
+ np.nan
rmedse =
np.zeros([num_algs,num_exps,iter_list.shape[0]])
+ np.nan
cor_p =
np.zeros([num_algs,num_exps,iter_list.shape[0]])
+ np.nan
cor_s =
np.zeros([num_algs,num_exps,iter_list.shape[0]])
+ np.nan
times =
np.zeros([num_algs,num_exps,iter_list.shape[0]])
+ np.nan

```

```

for idx_exp in range(num_exps):
    counter = np.zeros(num_algs)
    temp = time_now.localtime()
    current_time = time_now.strftime("%H:%M:%S",
temp)
    print('%s: Running Experiment %d'
%(current_time,idx_exp))
    m1 = RNN(opts)
    r = m1.get_time_evolution(T = T_data)
    gnd = m1.rnn['w_rec'].flatten()
    %%
    for iters in iter_list:
        temp = time_now.localtime()
        current_time =
time_now.strftime("%H:%M:%S", temp)
        print('\t %s: Running iteration %d'
%(current_time,iters))

        if iters == 0:
            try:
                w = solve_newton_descent(r[:-
1,:],r[1,:],u_in = None,alpha = 0.1,
l2 = 1e-5,verbose = 0,threshold
=0.2,
initialize_fp = 1,num_iters = iters,
solver_type = 'weighted')
            except:
                w = 0

        start_time = time_now.perf_counter()
        if counter[3] == 0:
            try:
                w = solve_newton_descent(r[:-
1,:],r[1,:],u_in = None,alpha = 0.1,
l2 = 1e-5,verbose = 0,threshold
=0.2,
initialize_fp = 1,num_iters =
iters,
solver_type = 'weighted')
            times[3,idx_exp,iters] =
time_now.perf_counter() - start_time;
            w_rec = w[:opts['n_rec'],:].T.flatten()
            slopes[3,idx_exp,iters] = (gnd @ w_rec) /
(gnd@gnd)
            rmse[3,idx_exp,iters] = np.sqrt(np.mean(
(w_rec-gnd)**2))
            rmedse[3,idx_exp,iters] =
np.sqrt(np.median((w_rec-gnd)**2))
            cor_p[3,idx_exp,iters] =
pearsonr(w_rec,gnd)[0]
            cor_s[3,idx_exp,iters] =
spearmanr(w_rec,gnd)[0]

```


APPENDIX A

```

        print('\t\t\t NT,weighted. Cor %.3f'
%(cor_p[3,idx_exp,itors]))
        if time_now.perf_counter() - start_time >
time_limit:
            counter[3]=1
        except:
            w = 0
        else:
            print('\t\t\t NT, weighted skipped due to
timeout error')

start_time = time_now.perf_counter()
if counter[5] == 0:
    try:
        w = solve_newton_descent(r[:-
1,:],r[1:,:],u_in = None,alpha = 0.1,
                                l2 = 1e-5,verbose = 0,threshold
=0.2,
                                initialize_fp = 1,num_iters =
iters,
                                solver_type = 'logistic')
        times[5,idx_exp,itors] =
time_now.perf_counter() - start_time;
        w_rec = w[:opts['n_rec'],:].T.flatten()
        slopes[5,idx_exp,itors] = (gnd @ w_rec) /
(gnd@gnd)
        rmse[5,idx_exp,itors] = np.sqrt(np.mean(
(w_rec-gnd)**2 ))
        rmedse[5,idx_exp,itors] =
np.sqrt(np.median( (w_rec-gnd)**2 ))
        cor_p[5,idx_exp,itors] =
pearsonr(w_rec,gnd)[0]
        cor_s[5,idx_exp,itors] =
spearmanr(w_rec,gnd)[0]
        print('\t\t\t NT,logistic. Cor %.3f'
%(cor_p[5,idx_exp,itors]))
        if time_now.perf_counter() - start_time >
time_limit:
            counter[5]=1
        except:
            print('\t\t\t NT,logistic terminated due to
timeout error')
            counter[5] = 1
        else:
            print('\t\t\t NT,logistic skipped due to
timeout error')

#%
start_time = time_now.perf_counter()
if counter[0] == 0:
    try:
        w = solve_corrnr(r[:-1,:],r[1:,:],u_in =
None,l2 = 1*1e-5,
                        verbose = 0,gnd = gnd, threshold =
0.2,
                        num_iters =
5*iters,check_convergence = 0,
                        solver_type = 'weighted')
        times[0,idx_exp,itors] =
time_now.perf_counter() - start_time;
        w_rec = w[:opts['n_rec'],:].T.flatten()
        slopes[0,idx_exp,itors] = (gnd @ w_rec) /
(gnd@gnd)
        rmse[0,idx_exp,itors] = np.sqrt(np.mean(
(w_rec-gnd)**2 ))
        rmedse[0,idx_exp,itors] =
np.sqrt(np.median( (w_rec-gnd)**2 ))
        cor_p[0,idx_exp,itors] =
pearsonr(w_rec,gnd)[0]
        cor_s[0,idx_exp,itors] =
spearmanr(w_rec,gnd)[0]
        print('\t\t\t CoRNN, CPU. Cor %.3f'
%(cor_p[0,idx_exp,itors]))
        if time_now.perf_counter() - start_time >
time_limit:
            counter[0] =1
        except:
            print('\t\t\t CoRNN, CPU terminated due
to timeout error')
            counter[0] = 1
        else:
            print('\t\t\t CoRNN, CPU skipped due to
timeout error')

#%
if iters == 0:
    w = solve_corrnr_gpu(r[:-1,:],r[1:,:],u_in =
None,l2 = 1*1e-5,
                        verbose = 0,gnd = gnd, threshold =
0.2,
                        num_iters = 5*iters,
                        solver_type = 'weighted')
    start_time = time_now.perf_counter()
    if counter[8] == 0:
        try:

```

APPENDIX A

```

w = solve_corrnn_gpu(r[:-1,:],r[1:,:],u_in =
None,l2 = 1*1e-5,
verbose = 0,gnd = gnd, threshold =
0.2,
num_iters = 5*iters,
solver_type = 'weighted')
times[8,idx_exp,iters] =
time_now.perf_counter() - start_time;
w_rec = w[:opts['n_rec'],:].T.flatten()
slopes[8,idx_exp,iters] = (gnd @ w_rec) /
(gnd@gnd)
rmse[8,idx_exp,iters] = np.sqrt(np.mean(
(w_rec-gnd)**2))
rmedse[8,idx_exp,iters] =
np.sqrt(np.median((w_rec-gnd)**2))
cor_p[8,idx_exp,iters] =
pearsonr(w_rec,gnd)[0]
cor_s[8,idx_exp,iters] =
spearmanr(w_rec,gnd)[0]
print('\t \t \t CoRNN, GPU. Cor %.3f'
%(cor_p[8,idx_exp,iters]))
if time_now.perf_counter() - start_time >
time_limit:
counter[8] = 1
except:
print('\t \t \t CoRNN, GPU terminated due
to timeout error')
counter[8] = 1
else:
print('\t \t \t CoRNN, GPU skipped due to
timeout error')

##
start_time = time_now.perf_counter()
if counter[1] == 0:
try:

w = solve_pytorch_gpu(r[:-1,:],r[1:,:],u_in =
None,alpha = 0.1,
l2 = 1e-8,verbose = 0,gnd =
gnd, #1e-7 for short
initialize_fp = 1,num_iters =
iters*1000,
learning_rate = .01,solver_type
= 'logistic')
times[1,idx_exp,iters] =
time_now.perf_counter() - start_time;
w_rec = w[:opts['n_rec'],:].T.flatten()
slopes[1,idx_exp,iters] = (gnd @ w_rec) /
(gnd@gnd)

```

```

rmse[1,idx_exp,iters] = np.sqrt(np.mean(
(w_rec-gnd)**2))
rmedse[1,idx_exp,iters] =
np.sqrt(np.median((w_rec-gnd)**2))
cor_p[1,idx_exp,iters] =
pearsonr(w_rec,gnd)[0]
cor_s[1,idx_exp,iters] =
spearmanr(w_rec,gnd)[0]
print('\t \t \t Pytorch,logistic,GPU. Cor
%.3f' %(cor_p[1,idx_exp,iters]))
if time_now.perf_counter() - start_time >
time_limit:
counter[1] = 1
except:
print('\t \t \t Pytorch,logistic,GPU
terminated due to timeout error')
counter[1] = 1
else:
print('\t \t \t Pytorch,logistic,GPU skipped
due to timeout error')

```

```

##
start_time = time_now.perf_counter()
if counter[2] == 0:
try:

```

```

w = solve_pytorch(r[:-1,:],r[1:,:],u_in =
None,alpha = 0.1,
l2 = 1e-8,verbose = 0,gnd =
gnd,
initialize_fp = 1,num_iters =
iters * 1000,
learning_rate = .01,solver_type
= 'logistic')
times[2,idx_exp,iters] =
time_now.perf_counter() - start_time;
w_rec = w[:opts['n_rec'],:].T.flatten()
slopes[2,idx_exp,iters] = (gnd @ w_rec) /
(gnd@gnd)
rmse[2,idx_exp,iters] = np.sqrt(np.mean(
(w_rec-gnd)**2))
rmedse[2,idx_exp,iters] =
np.sqrt(np.median((w_rec-gnd)**2))
cor_p[2,idx_exp,iters] =
pearsonr(w_rec,gnd)[0]
cor_s[2,idx_exp,iters] =
spearmanr(w_rec,gnd)[0]
print('\t \t \t Pytorch,logistic, CPU. Cor
%.3f' %(cor_p[2,idx_exp,iters]))
if time_now.perf_counter() - start_time >
time_limit:
counter[2] = 1

```

APPENDIX A

```

except:
    print('\t\t\t\t\tPytorch,logistic, CPU
terminated due to timeout error')
    counter[2] = 1
else:
    print('\t\t\t\t\tPytorch,logistic, CPU skipped
due to timeout error')

%%
start_time = time_now.perf_counter()
if counter[4] == 0:
    try:

        w = solve_pytorch_gpu(r[:-1,:],r[1:,:],u_in =
None,alpha = 0.1,
                                l2 = 1e-8,verbose = 0,gnd =
gnd, #1e-7 for short
                                initialize_fp = 1,num_iters =
iters*1000,
                                learning_rate = .01,solver_type
= 'l2')
        times[4,idx_exp,iters] =
time_now.perf_counter() - start_time;
        w_rec = w[:opts['n_rec'],:].T.flatten()
        slopes[4,idx_exp,iters] = (gnd @ w_rec) /
(gnd@gnd)
        rmse[4,idx_exp,iters] = np.sqrt(np.mean(
(w_rec-gnd)**2))
        rmedse[4,idx_exp,iters] =
np.sqrt(np.median( (w_rec-gnd)**2))
        cor_p[4,idx_exp,iters] =
pearsonr(w_rec,gnd)[0]
        cor_s[4,idx_exp,iters] =
spearmanr(w_rec,gnd)[0]
        print('\t\t\t\t\tPytorch,l2,GPU. Cor %.3f'
%(cor_p[4,idx_exp,iters]))
        if time_now.perf_counter() - start_time >
time_limit:
            counter[4] = 1
        except:
            print('\t\t\t\t\tPytorch,l2,GPU terminated
due to timeout error')
            counter[4] = 1
        else:
            print('\t\t\t\t\tPytorch,l2,GPU skipped due to
timeout error')

%%
start_time = time_now.perf_counter()
if counter[6] == 0:
    try:

        w = fit_FORCE(r,None,alpha = 0.1,
                        lam = 200,g_in = 2,verbose = 0,
                        initialize_fp = 1,num_iters = 5*iters,
                        gnd = gnd,solver_type = 'currents')
        times[6,idx_exp,iters] =
time_now.perf_counter() - start_time;
        w_rec = w[:opts['n_rec'],:].T.flatten()
        slopes[6,idx_exp,iters] = (gnd @ w_rec) /
(gnd@gnd)
        rmse[6,idx_exp,iters] = np.sqrt(np.mean(
(w_rec-gnd)**2))
        rmedse[6,idx_exp,iters] =
np.sqrt(np.median( (w_rec-gnd)**2))
        cor_p[6,idx_exp,iters] =
pearsonr(w_rec,gnd)[0]
        cor_s[6,idx_exp,iters] =
spearmanr(w_rec,gnd)[0]
        print('\t\t\t\t\tForce,currents. Cor %.3f'
%(cor_p[6,idx_exp,iters]))
        if time_now.perf_counter() - start_time >
time_limit:
            counter[6] = 1
        except:
            print('\t\t\t\t\tForce,currents terminated due
to timeout error')
            counter[6] = 1
        else:
            print('\t\t\t\t\tForce,currents skipped due to
timeout error')

%%
start_time = time_now.perf_counter()
if counter[7] == 0:
    try:

        w = fit_FORCE(r,None,alpha = 0.1,
                        lam = 200,g_in = 2,verbose = 0,
                        initialize_fp = 1,num_iters = 5*iters,
                        gnd = gnd,solver_type =
'firing_rates')
        times[7,idx_exp,iters] =
time_now.perf_counter() - start_time;
        w_rec = w[:opts['n_rec'],:].T.flatten()
        slopes[7,idx_exp,iters] = (gnd @ w_rec) /
(gnd@gnd)
        rmse[7,idx_exp,iters] = np.sqrt(np.mean(
(w_rec-gnd)**2))
        rmedse[7,idx_exp,iters] =
np.sqrt(np.median( (w_rec-gnd)**2))
        cor_p[7,idx_exp,iters] =
pearsonr(w_rec,gnd)[0]

```

APPENDIX A

```

        cor_s[7,idx_exp,itors]=
spearmanr(w_rec,gnd)[0]
        print('\t\t\t Force, firing rates. Cor %.3f'
%(cor_p[7,idx_exp,itors]))
        if time_now.perf_counter() - start_time >
time_limit:
            counter[7]=1
            except:
                print('\t\t\t Force, firing rates terminated
due to timeout error')
                counter[7] = 1
            else:
                print('\t\t\t Force, firing rates skipped due to
timeout error')

```

```

if iters == 0:
    counter = np.zeros(num_algs)

np.savez('experiment1b_results_fp.npz',slopes =
slopes,
        rmse = rmse, rmedse = rmedse,cor_p =
cor_p,cor_s = cor_s,
        times = times)

```

run_experiment_1c.py

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""

```

Created on Wed Dec 21 17:52:09 2022

```

@author: dinc
"""

```

```

import numpy as np
from scipy.stats import pearsonr
from scipy.stats import spearmanr
from rnn_class import RNN
import matplotlib.pyplot as plt
from numpy.linalg import inv
from utils import solve_corrn
from utils import solve_corrn_gpu
from utils import solve_newton_descent
from utils import solve_gradient_descent
from utils import solve_pytorch
from utils import solve_pytorch_gpu
from utils import fit_FORCE
import time as time_now

```

```

time_limit = 200;

```

```

opts = {};
opts['g'] = 3;
opts['n_rec'] = 500 # 300 for short
opts['n_in'] = 1
opts['sigma_input'] = 1e-1
opts['sigma_conversion'] = 1e-5
opts['alpha'] = 0.1
opts['input_noise_type'] = 'Gaussian'
opts['conversion_noise_type'] = 'Gaussian'
opts['verbose'] = False;
opts['lambda_reg'] = 1e-5
opts['num_cores'] = 4
opts['parallel'] = 1
T_data = 10000; # 3000 for short

```

```

num_algs = 9;
num_exps = 100;
iter_list = np.linspace(0,10,11).astype(int)
slopes =
np.zeros([num_algs,num_exps,iter_list.shape[0]])
+ np.nan
rmse =
np.zeros([num_algs,num_exps,iter_list.shape[0]])
+ np.nan
rmedse =
np.zeros([num_algs,num_exps,iter_list.shape[0]])
+ np.nan
cor_p =
np.zeros([num_algs,num_exps,iter_list.shape[0]])
+ np.nan
cor_s =
np.zeros([num_algs,num_exps,iter_list.shape[0]])
+ np.nan
times =
np.zeros([num_algs,num_exps,iter_list.shape[0]])
+ np.nan

```

```

for idx_exp in range(num_exps):
    counter = np.zeros(num_algs)
    temp = time_now.localtime()
    current_time = time_now.strftime("%H:%M:%S",
temp)
    print('%s: Running Experiment %d'
%(current_time,idx_exp))
    m1 = RNN(opts)
    r = m1.get_time_evolution(T = T_data)
    gnd = m1.rnn['w_rec'].flatten()
    %%
    for iters in iter_list:
        temp = time_now.localtime()

```

APPENDIX A

```

current_time =
time_now.strftime("%H:%M:%S", temp)
print('\t %s: Running iteration %d'
%(current_time, iters))

if iters == 0:
    try:
        w = solve_newton_descent(r[:-
1,:], r[1,:], u_in = None, alph = 0.1,
                                l2 = 1e-5, verbose = 0, threshold
=0.2,
                                initialize_fp = 0, num_iters = iters,
                                solver_type = 'weighted')
    except:
        w = 0

start_time = time_now.perf_counter()
if counter[3] == 0:
    try:
        w = solve_newton_descent(r[:-
1,:], r[1,:], u_in = None, alph = 0.1,
                                l2 = 1e-5, verbose = 0, threshold
=1,
                                initialize_fp = 0, num_iters =
iters,
                                solver_type = 'weighted')
        times[3, idx_exp, iters] =
time_now.perf_counter() - start_time;
        w_rec = w[:opts['n_rec'],:].T.flatten()
        slopes[3, idx_exp, iters] = (gnd @ w_rec) /
(gnd@gnd)
        rmse[3, idx_exp, iters] = np.sqrt(np.mean(
(w_rec-gnd)**2))
        rmedse[3, idx_exp, iters] =
np.sqrt(np.median( (w_rec-gnd)**2))
        cor_p[3, idx_exp, iters] =
pearsonr(w_rec, gnd)[0]
        cor_s[3, idx_exp, iters] =
spearmanr(w_rec, gnd)[0]
        print('\t \t \t NT, weighted. Cor %.3f. %.1f
mins' %(cor_p[3, idx_exp, iters], \
(
time_now.perf_counter() - start_time)/60 ))
        if time_now.perf_counter() - start_time >
time_limit:
            counter[3] = 1
        except:
            w = 0
        else:
            print('\t \t \t NT, weighted skipped due to
timeout error')

```

```

start_time = time_now.perf_counter()
if counter[5] == 0:
    try:
        w = solve_newton_descent(r[:-
1,:], r[1,:], u_in = None, alph = 0.1,
                                l2 = 1e-5, verbose = 0, threshold
=1,
                                initialize_fp = 0, num_iters =
iters,
                                solver_type = 'logistic')
        times[5, idx_exp, iters] =
time_now.perf_counter() - start_time;
        w_rec = w[:opts['n_rec'],:].T.flatten()
        slopes[5, idx_exp, iters] = (gnd @ w_rec) /
(gnd@gnd)
        rmse[5, idx_exp, iters] = np.sqrt(np.mean(
(w_rec-gnd)**2))
        rmedse[5, idx_exp, iters] =
np.sqrt(np.median( (w_rec-gnd)**2))
        cor_p[5, idx_exp, iters] =
pearsonr(w_rec, gnd)[0]
        cor_s[5, idx_exp, iters] =
spearmanr(w_rec, gnd)[0]
        print('\t \t \t NT, logistic. Cor %.3f. %.1f
mins' %(cor_p[5, idx_exp, iters], \
( time_now.perf_counter() -
start_time)/60 ))
        if time_now.perf_counter() - start_time >
time_limit:
            counter[5] = 1
        except:
            print('\t \t \t NT, logistic terminated due to
timeout error')
            counter[5] = 1
        else:
            print('\t \t \t NT, logistic skipped due to
timeout error')

#%
start_time = time_now.perf_counter()
if counter[0] == 0:
    try:
        w = solve_corr(r[:-1,:], r[1,:], u_in =
None, l2 = 1*1e-5,

```

APPENDIX A

```

        verbose = 0, gnd = gnd, threshold =
1, initialize_fp = 0,
        num_iters =
300*iters, check_convergence = 0,
        solver_type = 'weighted')
    times[0, idx_exp, iters] =
time_now.perf_counter() - start_time;
    w_rec = w[:opts['n_rec'],:].T.flatten()
    slopes[0, idx_exp, iters] = (gnd @ w_rec) /
(gnd@gnd)
    rmse[0, idx_exp, iters] = np.sqrt(np.mean(
(w_rec-gnd)**2))
    rmedse[0, idx_exp, iters] =
np.sqrt(np.median((w_rec-gnd)**2))
    cor_p[0, idx_exp, iters] =
pearsonr(w_rec, gnd)[0]
    cor_s[0, idx_exp, iters] =
spearmanr(w_rec, gnd)[0]
    print('\t\t\t CoRNN, CPU. Cor %.3f'
%(cor_p[0, idx_exp, iters]))
    if time_now.perf_counter() - start_time >
time_limit:
        counter[0] = 1
    except:
        print('\t\t\t CoRNN, CPU terminated due
to timeout error')
        counter[0] = 1
    else:
        print('\t\t\t CoRNN, CPU skipped due to
timeout error')

    #%
    if iters == 0:
        w = solve_corrn_gpu(r[:-1,:], r[1:,:], u_in =
None, l2 = 1*1e-5,
        verbose = 0, gnd = gnd, threshold = 1,
        num_iters = 500*iters, float_type =
'64bit',
        solver_type = 'weighted')
        start_time = time_now.perf_counter()
        if counter[8] == 0:
            try:

                w = solve_corrn_gpu(r[:-1,:], r[1:,:], u_in =
None, l2 = 1*1e-5,
                verbose = 0, gnd = gnd, threshold =
1, initialize_fp = 0,
                num_iters = 300*iters, float_type =
'32bit',
                solver_type = 'weighted')
                times[8, idx_exp, iters] =
time_now.perf_counter() - start_time;

```

```

        w_rec = w[:opts['n_rec'],:].T.flatten()
        slopes[8, idx_exp, iters] = (gnd @ w_rec) /
(gnd@gnd)
        rmse[8, idx_exp, iters] = np.sqrt(np.mean(
(w_rec-gnd)**2))
        rmedse[8, idx_exp, iters] =
np.sqrt(np.median((w_rec-gnd)**2))
        cor_p[8, idx_exp, iters] =
pearsonr(w_rec, gnd)[0]
        cor_s[8, idx_exp, iters] =
spearmanr(w_rec, gnd)[0]
        print('\t\t\t CoRNN, GPU. Cor %.3f'
%(cor_p[8, idx_exp, iters]))
        if time_now.perf_counter() - start_time >
time_limit:
            counter[8] = 1
        except:
            print('\t\t\t CoRNN, GPU terminated due
to timeout error')
            counter[8] = 1
        else:
            print('\t\t\t CoRNN, GPU skipped due to
timeout error')

    #%
    start_time = time_now.perf_counter()
    if counter[6] == 0:
        try:

            w = solve_corrn(r[:-1,:], r[1:,:], u_in =
None, l2 = 1*1e-5,
            verbose = 0, gnd = gnd, threshold =
1, initialize_fp = 1,
            num_iters =
5*iters, check_convergence = 0,
            solver_type = 'weighted')
            times[6, idx_exp, iters] =
time_now.perf_counter() - start_time;
            w_rec = w[:opts['n_rec'],:].T.flatten()
            slopes[6, idx_exp, iters] = (gnd @ w_rec) /
(gnd@gnd)
            rmse[6, idx_exp, iters] = np.sqrt(np.mean(
(w_rec-gnd)**2))
            rmedse[6, idx_exp, iters] =
np.sqrt(np.median((w_rec-gnd)**2))
            cor_p[6, idx_exp, iters] =
pearsonr(w_rec, gnd)[0]
            cor_s[6, idx_exp, iters] =
spearmanr(w_rec, gnd)[0]
            print('\t\t\t CoRNN, CPU. Cor %.3f'
%(cor_p[6, idx_exp, iters]))

```

APPENDIX A

```

        if time_now.perf_counter() - start_time >
time_limit:
            counter[6] = 1
        except:
            print('\t\t\t CoRNN, CPU terminated due
to timeout error')
            counter[6] = 1
        else:
            print('\t\t\t CoRNN, CPU skipped due to
timeout error')

```

```

#%
if iters == 0:
    w = solve_corrnn_gpu(r[:-1,:],r[1:,:],u_in =
None,l2 = 1*1e-5,
        verbose = 0,gnd = gnd, threshold = 1,
        num_iters = 500*iters, float_type =
'64bit',
            solver_type = 'weighted')
    start_time = time_now.perf_counter()
    if counter[7] == 0:
        try:

            w = solve_corrnn_gpu(r[:-1,:],r[1:,:],u_in =
None,l2 = 1*1e-5,
                verbose = 0,gnd = gnd, threshold =
1, initialize_fp = 1,
                num_iters = 5*iters, float_type =
'32bit',
                    solver_type = 'weighted')
            times[7,idx_exp,iters] =
time_now.perf_counter() - start_time;
            w_rec = w[:opts['n_rec'],:].T.flatten()
            slopes[7,idx_exp,iters] = (gnd @ w_rec) /
(gnd@gnd)
            rmse[7,idx_exp,iters] = np.sqrt(np.mean(
(w_rec-gnd)**2))
            rmedse[7,idx_exp,iters] =
np.sqrt(np.median( (w_rec-gnd)**2))
            cor_p[7,idx_exp,iters] =
pearsonr(w_rec,gnd)[0]
            cor_s[7,idx_exp,iters] =
spearmanr(w_rec,gnd)[0]
            print('\t\t\t CoRNN, GPU. Cor %.3f'
%(cor_p[7,idx_exp,iters]))
            if time_now.perf_counter() - start_time >
time_limit:
                counter[7] = 1
            except:
                print('\t\t\t CoRNN, GPU terminated due
to timeout error')
                counter[7] = 1

```

```

        else:
            print('\t\t\t CoRNN, GPU skipped due to
timeout error')

#%
start_time = time_now.perf_counter()
if counter[1] == 0:
    try:

        w = solve_pytorch_gpu(r[:-1,:],r[1:,:],u_in =
None,alpha = 0.1,
            l2 = 1e-8,verbose = 0,gnd =
gnd, #1e-7 for short
            initialize_fp = 0,num_iters =
iters*500,
            learning_rate = .01,solver_type
= 'logistic')
        times[1,idx_exp,iters] =
time_now.perf_counter() - start_time;
        w_rec = w[:opts['n_rec'],:].T.flatten()
        slopes[1,idx_exp,iters] = (gnd @ w_rec) /
(gnd@gnd)
        rmse[1,idx_exp,iters] = np.sqrt(np.mean(
(w_rec-gnd)**2))
        rmedse[1,idx_exp,iters] =
np.sqrt(np.median( (w_rec-gnd)**2))
        cor_p[1,idx_exp,iters] =
pearsonr(w_rec,gnd)[0]
        cor_s[1,idx_exp,iters] =
spearmanr(w_rec,gnd)[0]
        print('\t\t\t Pytorch,logistic,GPU. Cor
%.3f' %(cor_p[1,idx_exp,iters]))
        if time_now.perf_counter() - start_time >
time_limit:
            counter[1] = 1
        except:
            print('\t\t\t Pytorch,logistic,GPU
terminated due to timeout error')
            counter[1] = 1
        else:
            print('\t\t\t Pytorch,logistic,GPU skipped
due to timeout error')

#%
start_time = time_now.perf_counter()
if counter[2] == 0:
    try:

        w = solve_pytorch(r[:-1,:],r[1:,:],u_in =
None,alpha = 0.1,

```

APPENDIX A

```

l2 = 1e-8, verbose = 0, gnd =
gnd,
initialize_fp = 0, num_iters =
iters * 500,
learning_rate = .01, solver_type
= 'logistic')
times[2, idx_exp, iters] =
time_now.perf_counter() - start_time;
w_rec = w[:opts['n_rec'], :].T.flatten()
slopes[2, idx_exp, iters] = (gnd @ w_rec) /
(gnd@gnd)
rmse[2, idx_exp, iters] = np.sqrt(np.mean(
(w_rec-gnd)**2))
rmedse[2, idx_exp, iters] =
np.sqrt(np.median( (w_rec-gnd)**2 ))
cor_p[2, idx_exp, iters] =
pearsonr(w_rec, gnd)[0]
cor_s[2, idx_exp, iters] =
spearmanr(w_rec, gnd)[0]
print('\t \t \t Pytorch, logistic, CPU. Cor
%.3f' %(cor_p[2, idx_exp, iters]))
if time_now.perf_counter() - start_time >
time_limit:
counter[2] = 1
except:
print('\t \t \t Pytorch, logistic, CPU
terminated due to timeout error')
counter[2] = 1
else:
print('\t \t \t Pytorch, logistic, CPU skipped
due to timeout error')

#%
start_time = time_now.perf_counter()
if counter[4] == 0:
try:

w = solve_pytorch_gpu(r[:-1, :], r[1:, :], u_in =
None, alph = 0.1,
l2 = 1e-8, verbose = 0, gnd =
gnd, #1e-7 for short
initialize_fp = 0, num_iters =
iters*500,
learning_rate = .01, solver_type
= 'l2')
times[4, idx_exp, iters] =
time_now.perf_counter() - start_time;
w_rec = w[:opts['n_rec'], :].T.flatten()
slopes[4, idx_exp, iters] = (gnd @ w_rec) /
(gnd@gnd)
rmse[4, idx_exp, iters] = np.sqrt(np.mean(
(w_rec-gnd)**2))

```

```

rmedse[4, idx_exp, iters] =
np.sqrt(np.median( (w_rec-gnd)**2 ))
cor_p[4, idx_exp, iters] =
pearsonr(w_rec, gnd)[0]
cor_s[4, idx_exp, iters] =
spearmanr(w_rec, gnd)[0]
print('\t \t \t Pytorch, l2, GPU. Cor %.3f'
%(cor_p[4, idx_exp, iters]))
if time_now.perf_counter() - start_time >
time_limit:
counter[4] = 1
except:
print('\t \t \t Pytorch, l2, GPU terminated
due to timeout error')
counter[4] = 1
else:
print('\t \t \t Pytorch, l2, GPU skipped due to
timeout error')

if iters == 0:
counter = np.zeros(num_algs) # Force takes
hours to run iteration 1 ....

np.savez('experiment1c_results_fp.npz', slopes =
slopes,
rmse = rmse, rmedse = rmedse, cor_p =
cor_p, cor_s = cor_s,
times = times)

```

try_timeout.py

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""

```

Created on Wed Dec 21 17:52:09 2022

@author: dinc

```

"""

```

```

import numpy as np
from scipy.stats import pearsonr
from rnn_class import RNN
import matplotlib.pyplot as plt
from numpy.linalg import inv
from utils import solve_corn_gpu
from utils import solve_corn
from utils import solve_newton_descent
from utils import solve_gradient_descent
from utils import solve_pytorch_gpu
from utils import solve_pytorch

```


APPENDIX A

```

from utils import fit_FORCE
from utils import fit_FORCE_gpu
import signal
import time as time_now

class TimeoutException(Exception):
    pass

def timeout_handler(signum, frame):
    raise TimeoutException

signal.signal(signal.SIGALRM, timeout_handler)

TIMEOUT = 10 # seconds

signal.alarm(TIMEOUT)

opts = {};
opts['g'] = 3;
opts['n_rec'] = 1000
opts['n_in'] = 0
opts['sigma_input'] = 1e-2
opts['sigma_conversion'] = 1e-4
T_data = 3000;
m1 = RNN(opts)
r = m1.get_time_evolution(T = T_data)
gnd = m1.rnn['w_rec'].flatten()

###
signal.alarm(TIMEOUT)
try:

    w = solve_corrn_gpu(r[:-1,:],r[1:,:],u_in = None,l2
= 1*1e-5,
        verbose = 2,gnd = gnd, threshold = 0.2,
        num_iters = 10,check_convergence = 0,
        solver_type = 'weighted',alph = 0.1)
    w_rec_co = w[:opts['n_rec'],:].T.flatten()
except TimeoutException:
    print('function terminated')

###
signal.alarm(TIMEOUT)
try:
    w3 = solve_pytorch_gpu(r[:-1,:],r[1:,:],u_in =
None,alph = 0.1,
        l2 = 1e-8,verbose = 2,gnd = gnd,
        initialize_fp = 1,num_iters = 5000,
        learning_rate = 0.01,solver_type =
'logistic')
    w_rec_pt = w3[:opts['n_rec'],:].T.flatten()
except TimeoutException:

        print('function terminated')

###
signal.alarm(TIMEOUT)
try:
    w1 = solve_newton_descent(r[:-1,:],r[1:,:],u_in =
None,alph = 0.1,
        l2 = 1e-5,verbose = 1,threshold =0.2,
        initialize_fp = 0,num_iters = 10,
        solver_type = 'weighted')
    w_rec_nt = w1[:opts['n_rec'],:].T.flatten()
except TimeoutException:
    print('function terminated')

###
signal.alarm(TIMEOUT)
try:
    w4 = fit_FORCE_gpu(r,None,alph = 0.1,
        lam = 100,g_in = 2,verbose = 2,
        initialize_fp = 1,num_iters = 100,
        gnd = gnd,solver_type = 'firing_rates')

    w_rec_fc = w4[:opts['n_rec'],:].T.flatten()
    signal.alarm(0)
except TimeoutException:
    print('function terminated')

###
plt.scatter(gnd,w_rec_pt)
plt.scatter(gnd,w_rec_co)

print('NT:', pearsonr(w_rec_nt,gnd)[0])
print('GD:', pearsonr(w_rec_pt,gnd)[0])
print('CoRNN:', pearsonr(w_rec_co,gnd)[0])

utils.py
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Jun 10 14:48:40 2022

@author: dinc
"""
import numpy as np
from scipy.linalg import solve
import re
import time as time_now
from numpy.linalg import inv

```

APPENDIX A

```

from scipy.stats import pearsonr
import multiprocessing
from joblib import Parallel, delayed
import torch
import torch.nn as nn
import torch.optim as optim

def solve_corrnr(r_in, r_out, u_in = None, alph = 0.1,
                l2 = 1e-4, threshold = 1, initialize_fp = 1,
                verbose = 0, check_convergence = 0,
                num_iters = 30, gnd = None, solver_type =
'weighted'):
    if verbose:
        temp = time_now.localtime()
        current_time =
time_now.strftime("%H:%M:%S", temp)
        print('%s: Running CoRNN solver: '
%(current_time))
        start_time = time_now.perf_counter()

    T_data = r_in.shape[0]
    n_rec = r_in.shape[1]
    if u_in is not None:
        n_in = u_in.shape[1]
    else:
        n_in = 0

    # Get the inputs x and targets d
    if n_in > 0:
        x = np.c_[r_in, u_in];
    else:
        x = r_in.copy()
    d = (r_out - (1 - alph) * r_in) / alph
    d[d <= -1 + 1e-6] = -1 + 1e-6;
    d[d >= 1 - 1e-6] = 1 - 1e-6;

    if verbose == 2:
        temp = time_now.localtime()
        current_time =
time_now.strftime("%H:%M:%S", temp)
        print('\t %s: Performing initial inverse
operations' %(current_time))

    # Scale the regularization and compute the fixed
point
    l2 = l2 * T_data;

    A = x.T @ x + (l2) * np.diag(np.ones(x.shape[1]));
    n_tot = n_rec + n_in

    Ainv = inv(A);
    Xp = Ainv @ x.T;

    Xm = Xp @ x;
    z = np.arctanh(d);
    theta_fp = Xp @ z;
    if initialize_fp:
        theta = theta_fp;
    else:
        theta =
np.random.normal(0, 1 / np.sqrt(n_rec), n_rec *
n_tot).reshape(n_tot, n_rec)

    if verbose == 2:
        temp = time_now.localtime()
        current_time =
time_now.strftime("%H:%M:%S", temp)
        print('\t %s: Starting iterations'
%(current_time))

    for idx in range(num_iters):
        theta_old = theta.copy()
        # Compute predictions and prediction errors
        dhat = np.tanh(x @ theta);
        if solver_type == 'weighted':
            E_pred = (d - dhat) / (1 - d**2);
            scale = np.sum(abs(E_pred) > threshold, 0);

            if verbose:
                per_not_used = np.mean(scale) / T_data;

            scale = T_data / (T_data - scale);

            E_pred[abs(E_pred) > threshold] = 0
            E_pred = E_pred * scale;
        elif solver_type == 'standard':
            E_pred = (d - dhat);
            per_not_used = 0;
        elif solver_type == 'robust':
            E_pred = (d - dhat) / (1 - d**2);
            per_not_used = 0;
            E_pred[abs(E_pred) > threshold] = threshold *
np.sign(E_pred[abs(E_pred) > threshold])

        # Perform the first primal variable update
        theta = Xm @ theta + Xp @ E_pred;
        conv = np.sqrt(n_rec) * np.sqrt(np.mean((theta
- theta_old) ** 2))

        if ((verbose == 2) & (gnd is not None)):
            prd = (theta[:n_rec, :].T).flatten();

```

APPENDIX A

```

    p_cor = pearsonr(gnd,prd)[0];
else:
    p_cor = np.nan;

    if ((verbose == 2) & (np.mod(idx,300) == 299)):
        temp = time_now.localtime()
        current_time =
time_now.strftime("%H:%M:%S", temp)
        slope = (gnd @ prd) / (gnd@gnd)
        rmse = np.sqrt(np.mean( (prd-gnd)**2 ))
        print('\t\t %s: Iteration %d finished. RMSE:
%.4f. Correlation %.3f. Slope %.3f. Not used %.2f.
Convergence %.7f.' \
            %(current_time,idx +
1,rmse,p_cor,slope,100*per_not_used,conv))

    if ((check_convergence>0) & (idx > 10)):
        if conv < 1e-5:
            break

    if verbose:
        temp = time_now.localtime()
        current_time =
time_now.strftime("%H:%M:%S", temp)
        tt = time_now.perf_counter() - start_time
        print('%s: CoRNN solver finished in %.2f mins'
%(current_time,tt/60))

    return theta

def solve_gradient_descent(r_in,r_out,u_in,alpha =
0.1,
                        l2 = 1e-5,verbose = 0,
                        initialize_fp = 0,num_iters = 1e4,
                        threshold = 1,learning_rate = 0.001,
                        gnd = None,momentum =
0,solver_type = 'weighted'):

    if verbose:
        temp = time_now.localtime()
        current_time =
time_now.strftime("%H:%M:%S", temp)
        print('%s: Running GD solver.'
%(current_time))
        start_time = time_now.perf_counter()

    T_data = r_in.shape[0]
    n_rec = r_in.shape[1]
    if u_in is not None:

```

```

        n_in = u_in.shape[1]
    else:
        n_in = 0
    n_tot = n_rec + n_in;

    # Get the inputs x and targets d
    if n_in > 0:
        x = np.c_[r_in,u_in];
    else:
        x = r_in.copy()
    d = (r_out - (1-alpha)*r_in)/alpha
    d[d<=-1+1e-6] = -1 + 1e-6;
    d[d>=1-1e-6] = 1-1e-6;

    # Scale the regularization and compute fixed
point
    l2_scaled = l2*T_data;
    if initialize_fp:
        A = x.T @ x + (l2_scaled) *
np.diag(np.ones(x.shape[1]));

        Ainv = inv(A);
        Xp = Ainv @ x.T;
        z = np.arctanh(d);
        theta_fp = Xp @ z;

    if initialize_fp:
        theta = theta_fp
    else:
        theta = np.zeros([n_tot,n_rec ])

    for idx in range(num_iters):

        grad =
compute_gradient(x,d,theta,l2,threshold,solver_ty
pe)
        del_gd = - grad * learning_rate ;
        if idx == 0:
            del_gd_old = del_gd.copy()

        delta_gd = momentum * (del_gd_old) + (1-
momentum) * del_gd

        del_gd_old = delta_gd.copy()

        theta = theta + delta_gd
        if ((verbose == 2) & (np.mod(idx,300) == 299) ):
            prd = (theta[:n_rec,:].T).flatten();
            if gnd is not None:
                p_cor = pearsonr(gnd,prd)[0];

```

APPENDIX A

```

else:
    p_cor = np.nan
    temp = time_now.localtime()
    current_time =
time_now.strftime("%H:%M:%S", temp)
    rmse = np.sqrt(np.mean((prd-gnd)**2))
    slope = (gnd @ prd) / (gnd@gnd)
    print("\t %s: iteration %d finished. RMSE:
%.4f. Correlation %.3f. Slope %.3f." \
        %(current_time,idx +
1,rmse,p_cor,slope))

if verbose:
    temp = time_now.localtime()
    current_time =
time_now.strftime("%H:%M:%S", temp)
    tt = time_now.perf_counter() - start_time
    print('%s: gradient descent finished in %.2f
mins' %(current_time,tt/60))

return theta

def solve_newton_descent(r_in,r_out,u_in,alpha =
0.1,
                        l2 = 1e-5,verbose = 0,
                        initialize_fp = 0,num_iters = 10,
                        threshold = 1,solver_type =
'weighted'):
    if verbose:
        temp = time_now.localtime()
        current_time =
time_now.strftime("%H:%M:%S", temp)
        print('%s: Running NT solver.' %(current_time))
        start_time = time_now.perf_counter()

    T_data = r_in.shape[0]
    n_rec = r_in.shape[1]
    if u_in is not None:
        n_in = u_in.shape[1]
    else:
        n_in = 0
    n_tot = n_rec + n_in;
    theta = np.zeros([n_tot,n_rec]);

    # Get the inputs x and targets d
    if n_in > 0:
        x = np.c_[r_in,u_in];
    else:
        x = r_in.copy()
    d = (r_out - (1-alpha)*r_in)/alpha

```

```

d[d<=-1+1e-6] = -1 + 1e-6;
d[d>=1-1e-6] = 1-1e-6;

# Scale the regularization and compute fixed
point
l2_scaled = l2*T_data;
if initialize_fp:
    A = x.T @ x + (l2_scaled) *
np.diag(np.ones(x.shape[1]));

Ainv = inv(A);
Xp = Ainv @ x.T;
z = np.arctanh(d);
theta_fp = Xp @ z;

num_cores =
np.array(multiprocessing.cpu_count()/2).astype(i
nt)
if verbose:
    temp = time_now.localtime()
    current_time =
time_now.strftime("%H:%M:%S", temp)
    print('%s: NT-descent runs with %d cores.'
%(current_time,num_cores))

def solve_descent(iters):
    if initialize_fp:
        theta_in = theta_fp[:,iters].copy()
    else:
        theta_in = None
    beta =
run_nt_algorithm(x,d[:,iters],num_iters,l2,theta_in
,threshold,solver_type)
    return beta

(result) =
Parallel(n_jobs=num_cores)(delayed(solve_desce
nt)(iters) for iters in range(n_rec))

theta = np.array(result).T

if verbose:
    temp = time_now.localtime()
    current_time =
time_now.strftime("%H:%M:%S", temp)
    tt = time_now.perf_counter() - start_time
    print('%s: NT-descent finished in %.2f mins'
%(current_time,tt/60))

return theta

```

APPENDIX A

```
def
run_nt_algorithm(x,d_tar,n_iter,l2,theta_in,thresho
ld,solver_type):
```

```
    if theta_in is not None:
```

```
        beta = theta_in
```

```
    else:
```

```
        beta = np.zeros([x.shape[1]])
```

```
    for idx in range(n_iter):
```

```
        grad =
```

```
compute_gradient(x,d_tar,beta,l2,threshold,solver
_type)
```

```
        hes =
```

```
compute_hessian(x,d_tar,beta,l2,threshold,solver
_type)
```

```
        del_nt = -solve(hes, grad,assume_a='pos')
```

```
        kappa = 1
```

```
        gamma = 0.4;
```

```
        chi = 0.6;
```

```
        count = 0
```

```
        while( compute_loss_function(x,d_tar,beta +
kappa * del_nt,l2) >= \
            compute_loss_function(x,d_tar,beta,l2) +
gamma * kappa* (grad @ del_nt) ):
```

```
            kappa = chi * kappa
```

```
            count = count + 1
```

```
            if count == 4:
```

```
                kappa = 1;
```

```
                break
```

```
        delta_gd = kappa * del_nt
```

```
        beta = beta + delta_gd
```

```
    return beta
```

```
def
```

```
compute_gradient(x,d_tar,beta,reg,threshold,solv
er_type):
```

```
    T_data = x.shape[0]
```

```
    dhat = np.tanh( x @ beta )
```

```
    if solver_type == 'weighted':
```

```
        E_pred = ( (dhat-d_tar) / (1-d_tar**2) ).T;
```

```
        scale = np.sum(abs(E_pred)>threshold,0);
```

```
        if np.max(scale) > 0.9 * T_data:
```

```
            scale =
```

```
np.sum(abs(E_pred)>10*threshold,0);
```

```
            threshold = 10* threshold;
```

```
            scale = T_data/(T_data-scale);
```

```
        E_pred[abs(E_pred)>threshold] = 0
```

```
        E_pred = E_pred * scale;
```

```
        E_pred = E_pred @ x / T_data
```

```
    else:
```

```
        E_pred = ( (dhat-d_tar) ).T;
```

```
        E_pred = E_pred @ x / T_data;
```

```
    gradient = E_pred.T + reg * beta
```

```
    return gradient
```

```
def
```

```
compute_hessian(x,d_tar,beta,reg,threshold,solve
r_type):
```

```
    T_data = x.shape[0]
```

```
    dhat = np.tanh( x @ beta )
```

```
    if solver_type == 'weighted':
```

```
        E_pred = ( (dhat-d_tar) / (1-d_tar**2) );
```

```
        scale = np.sum(abs(E_pred)>threshold,0);
```

```
        if np.max(scale) > 0.9 * T_data:
```

```
            scale =
```

```
np.sum(abs(E_pred)>10*threshold,0);
```

```
            threshold = 10* threshold;
```

```
            scale = T_data/(T_data-scale);
```

```
        temp = np.sqrt(1-dhat**2) / np.sqrt(1-
d_tar**2)
```

```
        temp[abs(E_pred)>threshold] = 0
```

```
        temp = temp * scale;
```

```
    else:
```

```
        temp = np.sqrt(1-dhat**2);
```

```
    r = x * temp[:,None];
```

```
    H = r.T @ r / r.shape[0] + reg *
np.diag(np.ones(r.shape[1]))
```

```
    return H
```

APPENDIX A

```

def compute_loss_function(x,d,beta,reg):

    dhat = np.tanh( x @ beta )
    dhat[dhat>=1] = 1-1e-16
    dhat[dhat<=-1] = -1+1e-16
    loss = - np.mean( 0.5*(d+1)*
np.log(0.5*(dhat+1)) + 0.5*(1-d)* np.log(0.5*(1-
dhat)) ) + reg * np.sum(beta**2)/2

    return loss

def solve_pytorch(r_in,r_out,u_in,alpha = 0.1,
                l2 = 1e-5,verbose = 0,
                initialize_fp = 0,num_iters = 1e4,
                threshold = 2,learning_rate = 0.001,
                gnd = None,solver_type = 'logistic'):
    if verbose:
        temp = time_now.localtime()
        current_time =
time_now.strftime("%H:%M:%S", temp)
        print('%s: Running Pytorch-GD solver'
%(current_time))
        start_time = time_now.perf_counter()

    n_rec = r_in.shape[1]
    if u_in is not None:
        n_in = u_in.shape[1]
    else:
        n_in = 0

    # Get the inputs x and targets d
    if n_in > 0:
        x = np.c_[r_in,u_in];
    else:
        x = r_in.copy()
    d = (r_out - (1-alpha)*r_in)/alpha
    d[d<=-1+1e-6] = -1 + 1e-6;
    d[d>=1-1e-6] = 1-1e-6;
    if initialize_fp:
        T_data = x.shape[0]
        A = x.T @ x + l2 *T_data *1e4*
np.diag(np.ones(x.shape[1]));

    Ainvs = inv(A);
    Xp = Ainvs @ x.T;
    z = np.arctanh(d);
    theta_fp = Xp @ z;

    # Define the network

```

```

class Model(nn.Module):
    def __init__(self, n_rec, n_in):
        super(Model, self).__init__()
        self.n_rec = n_rec
        self.n_in = n_in
        self.linear = nn.Linear(n_rec + n_in,
n_rec,bias=False)
        self.tanh = nn.Tanh()

    def forward(self, x_in):
        #x_in = torch.tensor(x_in,
dtype=torch.float32)
        d_out = self.tanh(self.linear(x_in))
        return d_out

d = torch.tensor(d, dtype=torch.float32)
x = torch.tensor(x, dtype=torch.float32)
model = Model(n_rec, n_in)
# define the multi target logistic regression loss
function
if solver_type == 'logistic':
    criterion = nn.BCELoss()
else:
    criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(),
lr=learning_rate, weight_decay=l2)
if initialize_fp:
    model.linear.weight.data =
torch.tensor(theta_fp.T, dtype=torch.float32)
for idx in range(num_iters):
    d_out = model(x)

    # compute the loss
    loss = criterion((1 + d_out)/2, (1 + d)/2)

    # backward
    loss.backward()

    # update the weights
    optimizer.step()
    # zero the gradients
    optimizer.zero_grad()

    if ((verbose == 2) & (np.mod(idx,300) == 299) ):
        weights = model.linear.weight
        prd = weights[:,
:n_rec].detach().numpy().flatten()
        if gnd is not None:
            p_cor = pearsonr(gnd,prd)[0];
        else:
            p_cor = np.nan

```

APPENDIX A

```

        slope = (gnd @ prd) / (gnd@gnd)

        temp = time_now.localtime()
        current_time =
time_now.strftime("%H:%M:%S", temp)
        rmse = np.sqrt(np.mean( (prd-gnd)**2 ))
        print("\t %s: Iteration %d finished. RMSE:
%.4f Correlation %.3f. Slope %.3f.\
        %(current_time,idx +
1,rmse,p_cor,slope))

    if verbose:
        temp = time_now.localtime()
        current_time =
time_now.strftime("%H:%M:%S", temp)
        tt = time_now.perf_counter() - start_time
        print('%s: Pytorch finished in %.2f mins'
        %(current_time,tt/60))

    return model.linear.weight.detach().numpy().T

def solve_pytorch_gpu(r_in,r_out,u_in,alpha = 0.1,
        l2 = 1e-5,verbose = 0,
        initialize_fp = 0,num_iters = 1e4,
        threshold = 2,learning_rate = 0.001,
        gnd = None,solver_type = 'logistic'):
    if verbose:
        temp = time_now.localtime()
        current_time =
time_now.strftime("%H:%M:%S", temp)
        print('%s: Running Pytorch-GD solver:
        %(current_time))
        start_time = time_now.perf_counter()

    n_rec = r_in.shape[1]
    if u_in is not None:
        n_in = u_in.shape[1]
    else:
        n_in = 0

    try:
        gpu_device = torch.device("cuda:0");
        a = torch.tensor(1,device = gpu_device);
    except:
        gpu_device = torch.device("mps");

    # Get the inputs x and targets d
    if n_in > 0:
        x = np.c_[r_in,u_in];
    else:

```

```

        x = r_in.copy()
        d = (r_out - (1-alpha)*r_in)/alpha
        d[d<=-1+1e-6] = -1 + 1e-6;
        d[d>=1-1e-6] = 1-1e-6;

    # Define the network
    class Model(nn.Module):
        def __init__(self, n_rec, n_in):
            super(Model, self).__init__()
            self.n_rec = n_rec
            self.n_in = n_in
            self.linear = nn.Linear(n_rec + n_in,
n_rec,bias=False)
            self.tanh = nn.Tanh()

        def forward(self, x_in):
            #x_in = torch.tensor(x_in,
dtype=torch.float32)
            d_out = self.tanh(self.linear(x_in))
            return d_out

        d = torch.tensor(d, dtype=torch.float32,device =
gpu_device)
        x = torch.tensor(x, dtype=torch.float32,device =
gpu_device)

        if initialize_fp:
            T_data = x.shape[0]
            n_tot = n_rec+ n_in
            reg_term = torch.tensor((l2) * T_data * 1e4 *
np.diag(np.ones(n_tot)),\
                device=gpu_device,dtype =
torch.float32)

        A = x.T @ x + reg_term;

        try:
            Ainv = torch.linalg.inv(A);
        except:
            Ainv =
torch.linalg.inv(A.cpu()).to(gpu_device);
            Xp = Ainv @ x.T;
            z = torch.arctanh(d);
            theta_fp = Xp @ z;
            theta_fp = theta_fp.cpu().detach().numpy()

        model = Model(n_rec, n_in)
        model.to(gpu_device)
        # define the multi target logistic regression loss
function
        if solver_type == 'logistic':

```

APPENDIX A

```

        criterion = nn.BCELoss()
    else:
        criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(),
lr=learning_rate, weight_decay=l2)
    if initialize_fp:
        model.linear.weight.data =
torch.tensor(theta_fp.T,
dtype=torch.float32).to(gpu_device)
    for idx in range(num_iters):
        d_out = model(x)

        # compute the loss
        loss = criterion((1 + d_out)/2, (1 + d)/2)

        # backward
        loss.backward()

        # update the weights
        optimizer.step()

        # zero the gradients
        optimizer.zero_grad()

    if ((verbose == 2) & (np.mod(idx,300) == 299)):
        weights = model.linear.weight
        prd = weights[:,
:n_rec].cpu().detach().numpy().flatten()
        if gnd is not None:
            p_cor = pearsonr(gnd,prd)[0];
        else:
            p_cor = np.nan
            slope = (gnd @ prd) / (gnd@gnd)

        temp = time_now.localtime()
        current_time =
time_now.strftime("%H:%M:%S", temp)
        rmse = np.sqrt(np.mean( (prd-gnd)**2 ))
        print('\t %s: Iteration %d finished. RMSE:
%.4f Correlation %.3f. Slope %.3f.' \
            %(current_time,idx +
1,rmse,p_cor,slope))

    if verbose:
        temp = time_now.localtime()
        current_time =
time_now.strftime("%H:%M:%S", temp)
        tt = time_now.perf_counter() - start_time
        print('%s: Pytorch finished in %.2f mins'
            %(current_time,tt/60))

```

```

    return
model.linear.weight.cpu().detach().numpy().T

def fit_FORCE(r,u=None,alpha = 0.1,
            lam = 100,lam_in = 1,g_in = 3,verbose = 0,
            initialize_fp = 0,num_iters = 1e3,
            gnd = None,solver_type = 'currents'):
    if verbose:
        temp = time_now.localtime()
        current_time =
time_now.strftime("%H:%M:%S", temp)
        print('%s: Running FORCE solver on %s.'
            %(current_time,solver_type))

    def predict_single(r_before,u_in = None):
        if u_in is not None:
            x = np.dot(w_rec,r_before) +
np.dot(w_in,u_in);
        else:
            x = np.dot(w_rec,r_before);
        rout = (1-alpha) * r_before + alpha * np.tanh(x)

    return rout,x

    if initialize_fp:
        if u is not None:
            n_in = u.shape[1]
        else:
            n_in = 0
        r_in = r[:,-1:]
        r_out = r[1,:];
        d = (r_out - (1-alpha)*r_in)/alpha
        d[d<=-1+1e-6] = -1 + 1e-6;
        d[d>=1-1e-6] = 1-1e-6;
        if n_in > 0:
            x = np.c_[r_in,u];
        else:
            x = r_in.copy()
        T_data = x.shape[0]
        A = x.T @ x + lam/T_data *
np.diag(np.ones(x.shape[1]));

        Ainv = inv(A);
        Xp = Ainv @ x.T;
        z = np.arctanh(d);
        theta_fp = Xp @ z;

    T = r.shape[0]-1
    n_rec= r.shape[1]

```


APPENDIX A

```

if u is not None and u.shape[0] != T:
    raise Exception("There is a mismatch
between u and r dimensions!!!")

temp=np.random.normal(0,g_in/np.sqrt(n_rec),n_
rec*n_rec).reshape(n_rec,n_rec)
np.fill_diagonal(temp,0)
w_rec = temp

if u is not None:
    n_in = u.shape[1]
    w_in
=np.random.normal(0,g_in/np.sqrt(n_in),n_in*n_r
ec).reshape(n_rec,n_in)
else:
    n_in = 0

if initialize_fp:
    w_rec = theta_fp[:n_rec,:].T
    if n_in > 0:
        w_in =
theta_fp[n_rec,:].T.reshape(n_rec,n_in)
    if (verbose == 2):
        prd = w_rec.flatten()
        if gnd is not None:
            p_cor = pearsonr(gnd,prd)[0];
        else:
            p_cor = np.nan
        print('\t %s: FP initialized: Correlation %.3f' \
            %(current_time,p_cor))

for k in range(num_iters):
    P = np.diag(np.ones(n_rec))/lam;
    if n_in > 0:
        P_in = np.diag(np.ones(n_in))/lam_in;
        r_bef = r[0,:];
        for i in range(T):
            temp = (r[i+1,:] - (1-alpha)*r[i,:])/alph
            temp[temp>=1-1e-6] = 1-1e-6
            temp[temp<=-1+1e-6] = -1+1e-6

            x_now = np.arctanh(temp)
            if n_in > 0:
                r_out,x_out = predict_single(r_bef,u[i])
            else:
                r_out,x_out = predict_single(r_bef)

            if solver_type == 'currents':
                e_min = x_out - x_now;

```

```

elif solver_type == 'firing_rates':
    e_min = (r_out - r[i+1,:])/alph;

    Pxr = P @ r_bef;
    rxPxr = r_bef @ (P @ r_bef) + 1
    delP = - np.outer(Pxr, Pxr) / rxPxr
    P = P + delP;
    delW = - np.outer(e_min, P @ r_bef)

    w_rec = w_rec + delW;

if n_in > 0:
    Pxu = P_in @ u[i]
    uxPxu = u[i] @ (P_in @ u[i]) + 1
    delP = - np.outer(Pxu, Pxu) / uxPxu
    P_in = P_in + delP;
    delW = - np.outer(e_min, P_in @ u[i])
    w_in = w_in + delW;
    r_bef,x_pred = predict_single(r_bef,u[i])
else:
    r_bef,x_pred = predict_single(r_bef)

if solver_type == 'currents':
    e_pls = x_pred - x_now
elif solver_type == 'firing_rates':
    e_pls = (r_bef - r[i+1,:])/alph;

if ((verbose == 2)):
    prd = w_rec.flatten()
    if gnd is not None:
        p_cor = pearsonr(gnd,prd)[0];
    else:
        p_cor = np.nan
    slope = (gnd @ prd) / (gnd@gnd)

    temp = time_now.localtime()
    current_time =
time_now.strftime("%H:%M:%S", temp)
    rmse = np.sqrt(np.mean( (prd-gnd)**2 ))
    print('\t %s: Iteration %d finished. RMSE:
%.4f Correlation %.3f. Slope %.3f. Conv. %.3f' \
        %(current_time,k +
1,rmse,p_cor,slope,np.min(e_pls/e_min)))

if n_in>0:
    theta = np.c_[w_rec,w_in]
else:
    theta = w_rec

return theta.T

```

APPENDIX A

```

def solve_corrn_gpu(r_in,r_out,u_in = None, alph =
0.1, float_type = '32bit',
    l2 = 1e-4, threshold = 0.2,initialize_fp = 1,
    verbose = 0,mask = None,
check_convergence = 0,
    num_iters = 30,gnd = None,solver_type =
'weighted'):
    if verbose:
        temp = time_now.localtime()
        current_time =
time_now.strftime("%H:%M:%S", temp)
        print('%s: Running CoRNN solver.'
%(current_time))
        start_time = time_now.perf_counter()
    try:
        gpu_device = torch.device("cuda:0");
        a = torch.tensor(1,device = gpu_device);
    except:
        gpu_device = torch.device("mps");
    num_iters = int(num_iters);
    T_data = r_in.shape[0]
    n_rec = r_in.shape[1]
    if u_in is not None:
        n_in = u_in.shape[1]
    else:
        n_in = 0

    # Get the inputs x and targets d
    if n_in > 0:
        x = np.c_[r_in,u_in];
    else:
        x = r_in.copy()
    d = (r_out - (1-alpha)*r_in)/alph
    d[d<=-1+1e-6] = -1 + 1e-6;
    d[d>=1-1e-6] = 1-1e-6;
    if float_type == '32bit':
        dtype = torch.float32
    else:
        dtype = torch.float64
    n_tot = x.shape[1]

    x = torch.tensor(x,device=gpu_device,
dtype=dtype)
    d = torch.tensor(d,device=gpu_device,
dtype=dtype)

    if verbose == 2:
        temp = time_now.localtime()
        current_time =
time_now.strftime("%H:%M:%S", temp)
        print('\t %s: Performing initial inverse
operations' %(current_time))

    # Scale the regularization and compute the fixed
point
    l2 = l2*T_data;

    if mask is None:
        mask =
np.r_[np.diag(np.ones(n_rec)),np.zeros([n_in,n_re
c])]
        mask = (mask == 1)

    reg_term = torch.tensor((l2) *
np.diag(np.ones(n_tot)),\
        device=gpu_device, dtype=dtype)

    A = x.T @ x + reg_term;

    try:
        Ainv = torch.linalg.inv(A);
    except:
        Ainv = torch.linalg.inv(A.cpu()).to(gpu_device);
    Xp = Ainv @ x.T;
    Xm = Xp @ x;
    z = torch.arctanh(d);
    theta_fp = Xp @ z;
    if initialize_fp:
        theta = theta_fp;
    else:
        w_rec =
np.random.normal(0,1/np.sqrt(n_rec),n_rec*n_tot
).reshape(n_tot,n_rec)
        theta =
torch.tensor(w_rec,device=gpu_device,
dtype=dtype);
        if ( (verbose == 2) & (gnd is not None) ) :
            prd = (theta[n_rec,:].T).flatten().cpu().numpy();
            p_cor = pearsonr(gnd,prd)[0];
            print('\t \t Fixed point correlation: %.3f'
%p_cor)
        else:
            p_cor = np.nan;

    if verbose == 2:
        temp = time_now.localtime()
        current_time =
time_now.strftime("%H:%M:%S", temp)
        print('\t %s: Starting iterations'
%(current_time))

    for idx in range(num_iters):
        # Compute predictions and prediction errors

```

APPENDIX A

```

dhat = torch.tanh(x @ theta);
if solver_type == 'weighted':
    E_pred = (d-dhat)/(1-d**2);
    per_not_used =
torch.sum(abs(E_pred)>threshold)/(n_rec *
T_data)
    E_pred[abs(E_pred)>threshold] = 0
elif solver_type == 'standard':
    E_pred = (d-dhat);
    per_not_used = 0;
elif solver_type == 'robust':
    E_pred = (d-dhat)/(1-d**2);
    per_not_used = 0;
    E_pred[abs(E_pred)>threshold] = threshold *
np.sign(E_pred[abs(E_pred)>threshold])

# Perform the first primal variable update
theta = Xm @ theta + Xp @ E_pred

if ( (verbose == 2) & (gnd is not None) ):
    prd =
(theta[:n_rec,:].T).flatten().cpu().numpy();
    p_cor = pearsonr(gnd,prd)[0];
else:
    p_cor = np.nan;

if ((verbose == 2) & (np.mod(idx,300) == 299) ):
    temp = time_now.localtime()
    current_time =
time_now.strftime("%H:%M:%S", temp)
    slope = (gnd @ prd) / (gnd@gnd)
    rmse = np.sqrt(np.mean( (prd-gnd)**2 ))
    print('\t\t %s: Iteration %d finished. RMSE:
%.4f. Correlation %.3f. Slope %.3f. Not used %.2f.'
\
        %(current_time,idx +
1,rmse,p_cor,slope,100*per_not_used))

if verbose:
    temp = time_now.localtime()
    current_time =
time_now.strftime("%H:%M:%S", temp)
    tt = time_now.perf_counter() - start_time

    print('%s: CoRNN solver finished in %.2f mins'
%(current_time,tt/60))

    return theta.cpu().numpy()

def fit_FORCE_gpu(r,u=None,alpha=0.1,
    lam=100,lam_in=1,g_in=3,verbose=0,
    initialize_fp=0,num_iters=1e3,
    gnd=None,solver_type='currents'):
    if verbose:
        temp = time_now.localtime()
        current_time =
time_now.strftime("%H:%M:%S", temp)
        print('%s: Running FORCE solver on %s.'
%(current_time,solver_type))

    try:
        gpu_device = torch.device("cuda:0");
        a = torch.tensor(1,device=gpu_device);
    except:
        gpu_device = torch.device("mps");

    def predict_single(r_before,u_in=None):
        if u_in is not None:
            x = w_rec @ r_before + w_in @ u_in;
        else:
            x = w_rec @ r_before;
        rout = (1-alpha) * r_before + alpha * torch.tanh(x)

        return rout,x

    if initialize_fp:
        if u is not None:
            n_in = u.shape[1]
        else:
            n_in = 0
        r_in = r[:,-1,:];
        r_out = r[1,:];
        d = (r_out - (1-alpha)*r_in)/alpha
        d[d<=-1+1e-6] = -1 + 1e-6;
        d[d>=1-1e-6] = 1-1e-6;
        if n_in > 0:
            x = np.c_[r_in,u];
        else:
            x = r_in.copy()

        x = torch.tensor(x,device=gpu_device,dtype=
torch.float32)
        d = torch.tensor(d,device=gpu_device,dtype=
torch.float32)
        T_data = x.shape[0]

```

APPENDIX A

```

    reg_term = torch.tensor(lam/T_data *
np.diag(np.ones(x.shape[1])),\
        device=gpu_device, dtype =
torch.float32)

    A = x.T @ x + reg_term;

    try:
        Ainv = torch.linalg.inv(A);
    except:
        Ainv =
torch.linalg.inv(A.cpu()).to(gpu_device);
    Xp = Ainv @ x.T;
    z = torch.arctanh(d);
    theta_fp = Xp @ z;

    r = torch.tensor(r, device=gpu_device, dtype =
torch.float32)
    alph =
torch.tensor(alph, device=gpu_device, dtype =
torch.float32)
    T = r.shape[0]-1
    n_rec= r.shape[1]
    if u is not None and u.shape[0] != T:
        raise Exception('There is a mismatch
between u and r dimensions!!!')

    temp=np.random.normal(0,g_in/np.sqrt(n_rec),n_
rec*n_rec).reshape(n_rec,n_rec)
    np.fill_diagonal(temp,0)
    w_rec =
torch.tensor(temp, device=gpu_device, dtype =
torch.float32)

    if u is not None:
        n_in = u.shape[1]
        w_in
= np.random.normal(0,g_in/np.sqrt(n_in),n_in*n_
rec).reshape(n_rec,n_in)
        w_in =
torch.tensor(w_in, device=gpu_device, dtype =
torch.float32)
    else:
        n_in = 0

    if initialize_fp:
        w_rec = theta_fp[n_rec,:].clone().detach().T
        if n_in > 0:

```

```

        w_in =
torch.tensor(theta_fp[n_rec,:].clone().detach()).T.r
eshape(n_rec,n_in), device=gpu_device, dtype =
torch.float32)
        if (verbose == 2):
            prd = w_rec.flatten().cpu().numpy();
            if gnd is not None:
                p_cor = pearsonr(gnd,prd)[0];
            else:
                p_cor = np.nan
            print('\t %s: FP initialized: Correlation %.3f.' \
                %(current_time,p_cor))

        for k in range(num_iters):
            P =
torch.tensor(np.diag(np.ones(n_rec))/lam, device=
gpu_device, dtype = torch.float32);
            if n_in > 0:
                P_in =
torch.tensor(np.diag(np.ones(n_in))/lam_in, device
=gpu_device, dtype = torch.float32);
                r_bef = r[0,:];
                for i in range(T):
                    temp = (r[i+1,:]- (1-alph)*r[i,:])/alph
                    temp[temp>=1-1e-6] = 1-1e-6
                    temp[temp<=-1+1e-6] = -1+1e-6

                    x_now = torch.arctanh(temp)
                    if n_in > 0:
                        r_out,x_out = predict_single(r_bef,u[i])
                    else:
                        r_out,x_out = predict_single(r_bef)

                    if solver_type == 'currents':
                        e_min = x_out - x_now;
                    elif solver_type == 'firing_rates':
                        e_min = (r_out - r[i+1,:])/alph;

                    Pxr = P @ r_bef;
                    rxPxr = r_bef @ (P @ r_bef) + 1
                    delP = - torch.outer(Pxr, Pxr) / rxPxr
                    P = P + delP;
                    delW = - torch.outer(e_min, P @ r_bef)

                w_rec = w_rec + delW;

            if n_in > 0:
                Pxu = P_in @ u[i]
                uxPxu = u[i] @ (P_in @ u[i]) + 1
                delP = - torch.outer(Pxu, Pxu) / uxPxu
                P_in = P_in + delP;

```

APPENDIX A

```

delW = - torch.outer(e_min, P_in @ u[i])
w_in = w_in + delW;
r_bef,x_pred = predict_single(r_bef,u[i])
else:
    r_bef,x_pred = predict_single(r_bef)

if solver_type == 'currents':
    e_pls = x_pred - x_now
elif solver_type == 'firing_rates':
    e_pls = (r_bef - r[i+1,:])/alph;

if ((verbose == 2)):
    prd = w_rec.flatten().cpu().numpy();
    if gnd is not None:
        p_cor = pearsonr(gnd,prd)[0];
    else:
        p_cor = np.nan
    slope = (gnd @ prd) / (gnd@gnd)

    temp = time_now.localtime()
    current_time =
time_now.strftime("%H:%M:%S", temp)
    rmse = np.sqrt(np.mean( (prd-gnd)**2 ))
    print("\t %s: Iteration %d finished. RMSE:
%.4f Correlation %.3f. Slope %.3f. Conv. %.3f" \
        %(current_time,k +
1,rmse,p_cor,slope,torch.min(e_pls/e_min)))

w_rec = w_rec.cpu().numpy()

if n_in>0:
    w_in = w_in.cpu().numpy()
    theta = np.c_[w_rec,w_in]
else:
    theta = w_rec

return theta.T

```

Figure scripts - plot_subfigure.py

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Jan 19 13:56:08 2023

@author: dinc
"""

import numpy as np
import matplotlib.pyplot as plt

plt.subplot(3,1,1)

```

```

f = np.load('experiment1a_results_fp.npz')
slopes = f['slopes']
rmse = f['rmse']
rmedse = f['rmedse']
cor_p = f['cor_p']
cor_s = f['cor_s']
times = f['times']

# 0 is CoRNN, CPU
# 1 is Pytorch, logistic, GPU
# 2 is Pytorch, logistic, CPU
# 3 is NT, weighted
# 4 is Pytorch, l2, GPU
# 5 is NT, logistic
# 6 is Force, currents
# 7 is Force, firing rates
# 8 is CoRNN, GPU

labels_all = ["Fixed point, weighted,
CPU","Pytorch, CE, GPU", \
    "Pytorch, CE, CPU", "Newton, weighted",
    "Pytorch, l2, GPU", \
    "Newton, CE", "Force, currents", "Force,
firing rates", \
    "Fixed point, weighted, GPU"]

for pick in [8,0,3,5,1,2,4,6,7]:

    temp = cor_p[pick,:].mean(0)
    temp_std = np.std(cor_p[pick,:],0) /
np.sqrt(cor_p.shape[1])
    x = np.nanmean(times[pick,:],0)
    x_std = np.std(times[pick,:],0) /
np.sqrt(cor_p.shape[1])
    plt.errorbar(temp,x,x_std,temp_std,label =
labels_all[pick])

plt.yscale('log')

plt.xlim([0.84,1.01])
plt.xticks([0.9,1],fontsize=14)
plt.yticks([0.1,1,10,100],fontsize=14)
#plt.xlabel('Correlation between ground truth and
inferred weights',fontsize = 15)
plt.ylabel('Time (s)',fontsize = 15)

plt.subplot(3,1,2)

```

APPENDIX A

```

f = np.load('experiment1b_results_fp.npz')
slopes = f['slopes']
rmse = f['rmse']
rmedse = f['rmedse']
cor_p = f['cor_p']
cor_s = f['cor_s']
times = f['times']

# 0 is CoRNN, CPU
# 1 is Pytorch, logistic, GPU
# 2 is Pytorch, logistic, CPU
# 3 is NT, weighted
# 4 is Pytorch, l2, GPU
# 5 is NT, logistic
# 6 is Force, currents
# 7 is Force, firing rates
# 8 is CoRNN, GPU

rmse = f['rmse']
rmedse = f['rmedse']
cor_p = f['cor_p']
cor_s = f['cor_s']
times = f['times']

# 0 is CoRNN, CPU
# 1 is Pytorch, logistic, GPU
# 2 is Pytorch, logistic, CPU
# 3 is NT, weighted
# 4 is Pytorch, l2, GPU
# 5 is NT, logistic
# 6 is Force, currents
# 7 is Force, firing rates
# 8 is CoRNN, GPU

labels_all = ["Fixed point, weighted,
CPU","Pytorch, CE, GPU", \
              "Pytorch, CE, CPU", "Newton, weighted",
"Pytorch, l2, GPU", \
              "Newton, CE", "Force, currents", "Force,
firing rates", \
              "Fixed point, weighted, GPU"]

for pick in [8,0,3,5,1,2,4,6,7]:

    temp = cor_p[pick,:].mean(0)
    temp_std = np.std(cor_p[pick,:],0) /
np.sqrt(cor_p.shape[1])
    x = np.nanmean(times[pick,:],0)
    x_std = np.std(times[pick,:],0) /
np.sqrt(cor_p.shape[1])
    plt.errorbar(temp,x,x_std,temp_std,label =
labels_all[pick])

plt.xlim([0.84,1.01])
plt.xticks([0.9,1])
plt.yticks([0.1,1,10,100],fontsize=14)
plt.ylabel('Time (s)',fontsize = 15)
plt.yscale('log')
plt.legend(loc='center left', bbox_to_anchor=(1,
0.5))

plt.subplot(3,1,3)
f = np.load('experiment1c_results_fp.npz')
slopes = f['slopes']

rmse = f['rmse']
rmedse = f['rmedse']
cor_p = f['cor_p']
cor_s = f['cor_s']
times = f['times']

# 0 is CoRNN, CPU
# 1 is Pytorch, logistic, GPU
# 2 is Pytorch, logistic, CPU
# 3 is NT, weighted
# 4 is Pytorch, l2, GPU
# 5 is NT, logistic
# 6 is Force, currents
# 7 is Force, firing rates
# 8 is CoRNN, GPU

labels_all = ["Fixed point, weighted,
CPU","Pytorch, CE, GPU", \
              "Pytorch, CE, CPU", "Newton, weighted",
"Pytorch, l2, GPU", \
              "Newton, CE", "Force, currents", "Force,
firing rates", \
              "Fixed point, weighted, GPU"]

for pick in [8,0,3,5,1,2,4,6,7]:

    temp = cor_p[pick,:].mean(0)
    temp_std = np.std(cor_p[pick,:],0) /
np.sqrt(cor_p.shape[1])
    x = np.nanmean(times[pick,:],0)
    x_std = np.std(times[pick,:],0) /
np.sqrt(cor_p.shape[1])
    plt.errorbar(temp,x,x_std,temp_std,label =
labels_all[pick])

plt.xlabel('Correlation between ground truth and
inferred weights',fontsize = 15)
plt.ylabel('Time (s)',fontsize = 15)
plt.yscale('log')

```

Wherefore, I/we claim:

1. A computer-implemented method for training a neural network to interpret neural dynamics, the method comprising:
 - obtaining functional connectivity between brain regions and artificial neurons simulated by the neural network;
 - capturing neural activities between biological neurons;
 - processing the captured neural activities to obtain training data for the neural network; and
 - constructing the neural network using the training data to reproduce neural dynamics between the artificial neurons.
2. The method of claim 1, wherein the neural network includes a convex solver.
3. The method of claim 2, wherein the convex solver comprises a convex weighted loss function.
4. The method of any one of claims 2 or 3, further comprising optimizing a loss of the convex solver via a fixed-point estimation.
5. The method of any one of claims 1-4, further comprising predicting a spatial and temporal activity of neural population dynamics.
6. The method of claim 1, wherein the neural network is configured to extract large-scale functional connectivity between brain regions and/or simulated networks.
7. The method of claim 1, wherein the neural network is trained to predict the temporal activity of a corresponding biological neuron in the brain.
8. The method of claim 1, wherein the neural network is configured to uncover attractor structures within the brain dynamics.
9. The method of claim 1, wherein the neural network is further configured to design external interventions to control neural population dynamics in the brain.
10. The method of any one of claims 1-9, wherein the neural network is trained to interpret neural dynamics in real-time to support brain-machine interfaces.
11. The method of any one of claims 2-4, wherein the convex solver is further configured to perform optimization updates accelerated on a graphics processing unit (GPU).
12. The method of any one of claims 2-4 and 11, wherein the convex solver is part of an alternating direction method of multipliers (ADMM) framework.
13. The method of any one of claims 2-4 and 11-12, wherein the convex solver is configured to enforce constraints on the neural network during training.

14. The method of claim 13, wherein the constraints include preventing self-excitation of neurons within the neural network.
15. The method of any one of claims 2-4 and 11-14, wherein the convex solver is configured to utilize biological priors for inductive biases during training.
16. The method of any one of claims 2-4 and 11-15, wherein the convex solver is configured to handle input noise and conversion noise during the training of the neural network.
17. The method of any one of claims 2-4 and 11-16, wherein the convex solver is configured to perform outlier detection during the training process.
18. The method of any one of claims 2-4 and 11-17, wherein the convex solver is configured to scale to datasets collected from more than 10,000 neurons for offline processing.
19. A non-transitory computer-readable medium comprising instructions that, when executed by one or more processors, cause a computer system to perform the method of any one of claims 1-18.
20. A computer system for training a neural network to interpret neural dynamics, the system comprising:
 - one or more processors; and
 - a non-transitory computer-readable medium including instructions that, when executed by the one or more processors, cause the computer system to perform the method of any one of claims 1-18.
21. The computer system of claim 22, wherein the neural network is a data constrained recurrent neural network.

Fig. 1

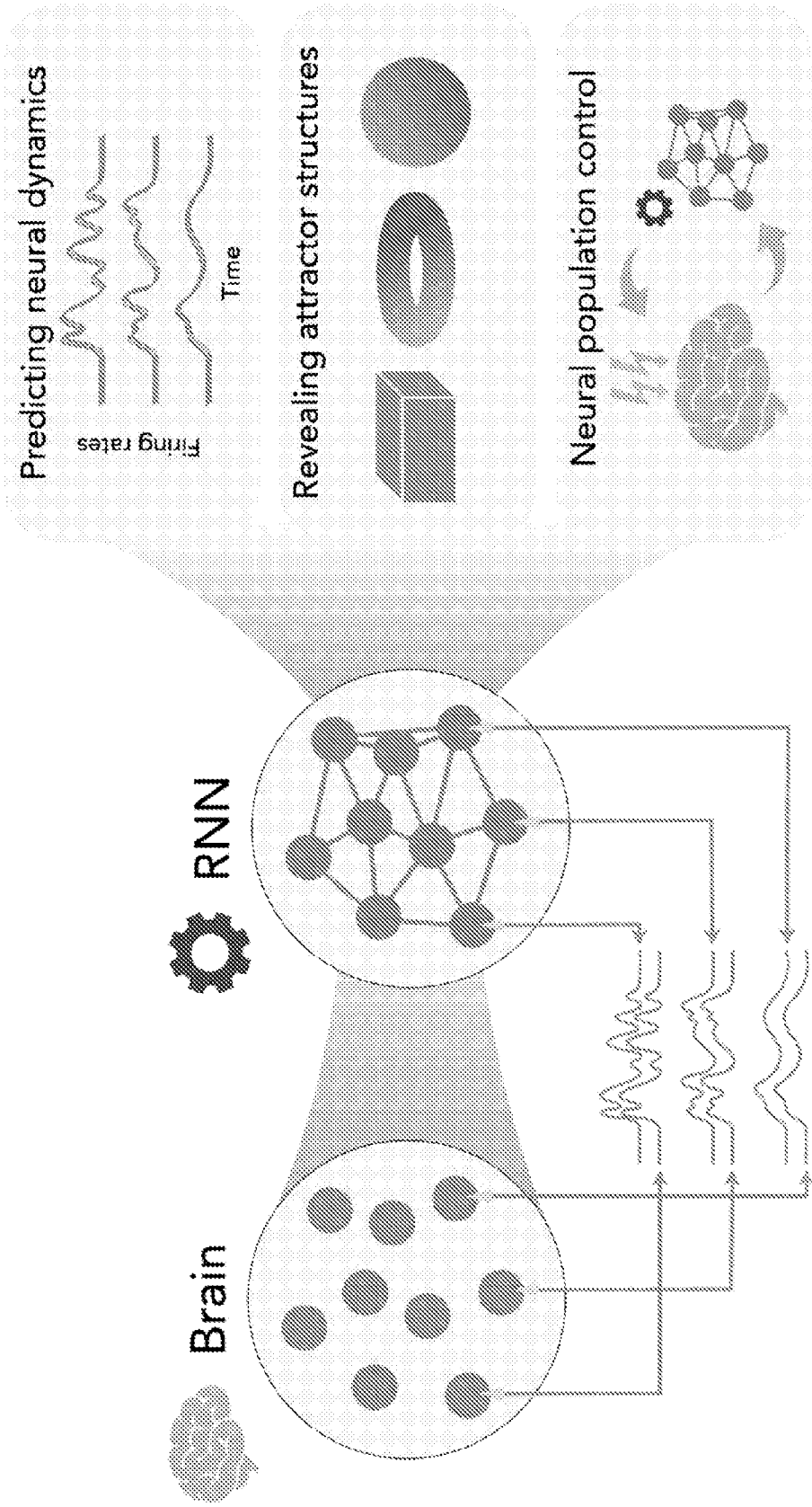


Fig. 2

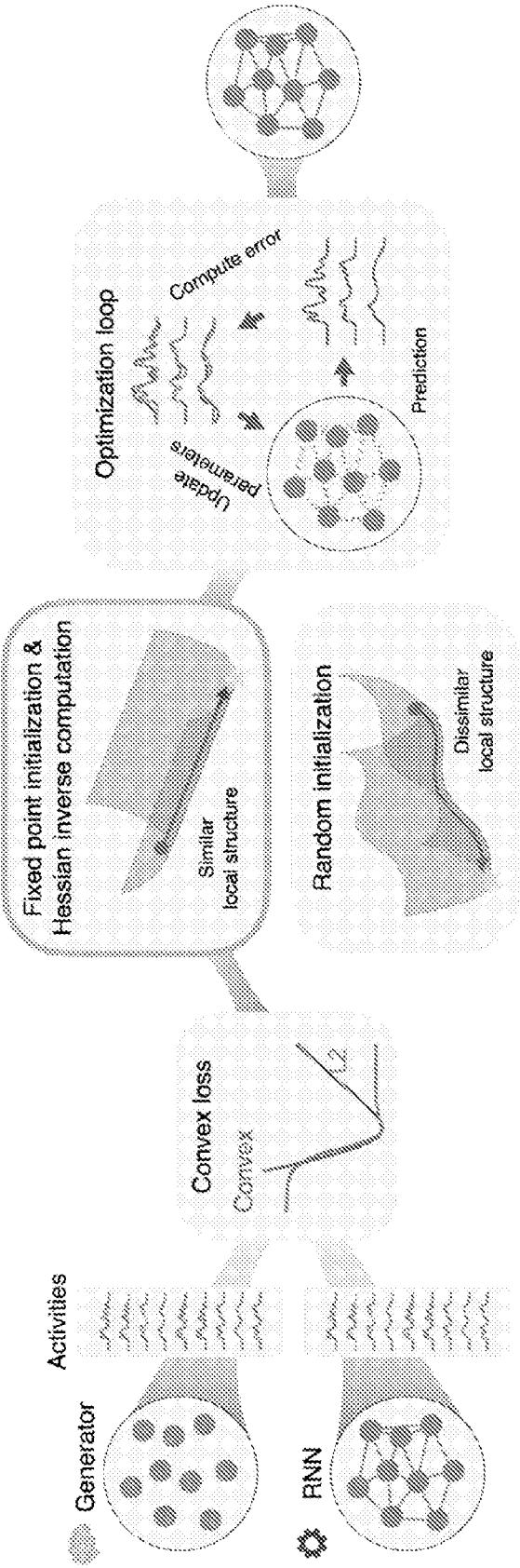


Fig. 3

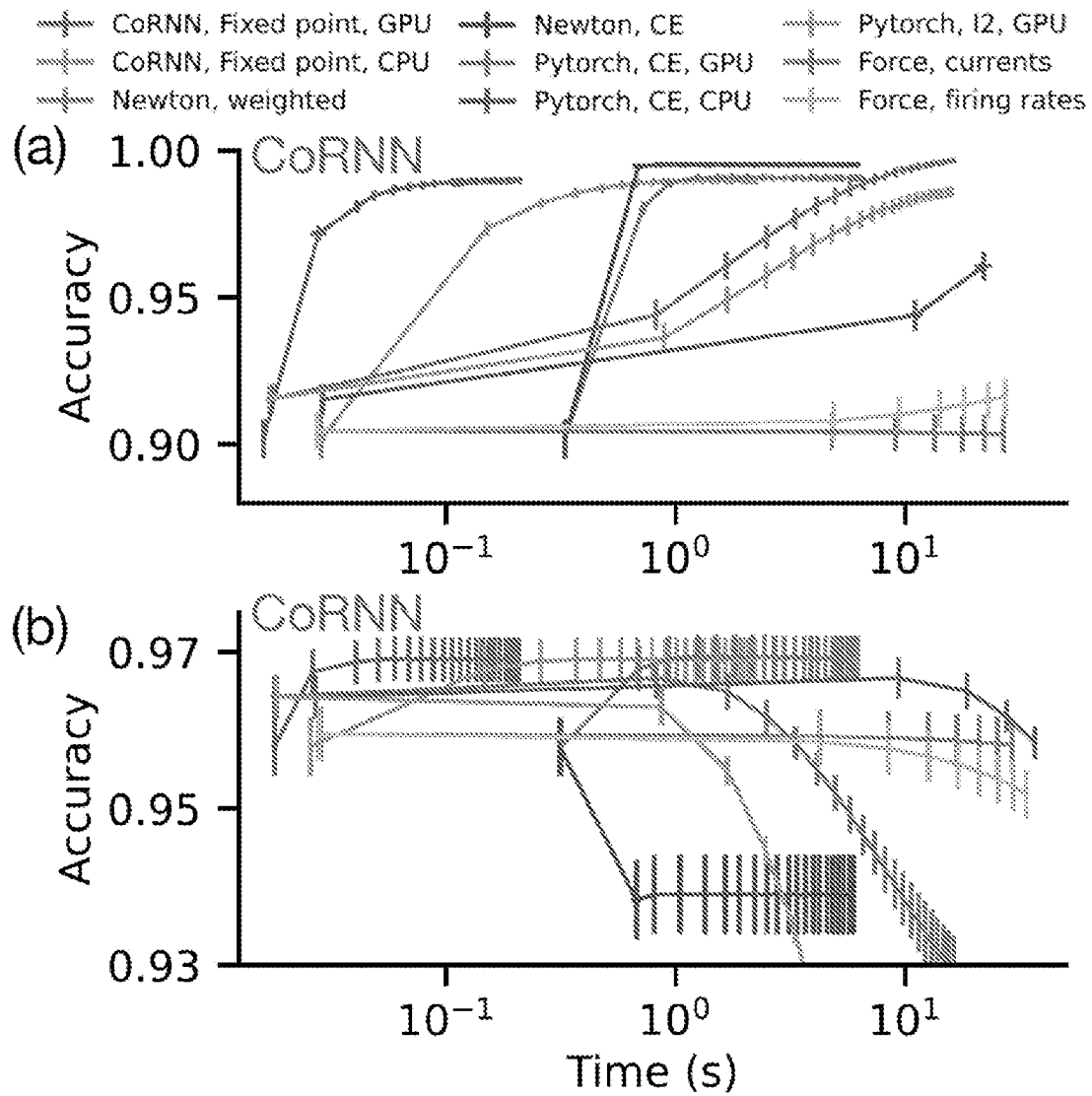


Fig. 4

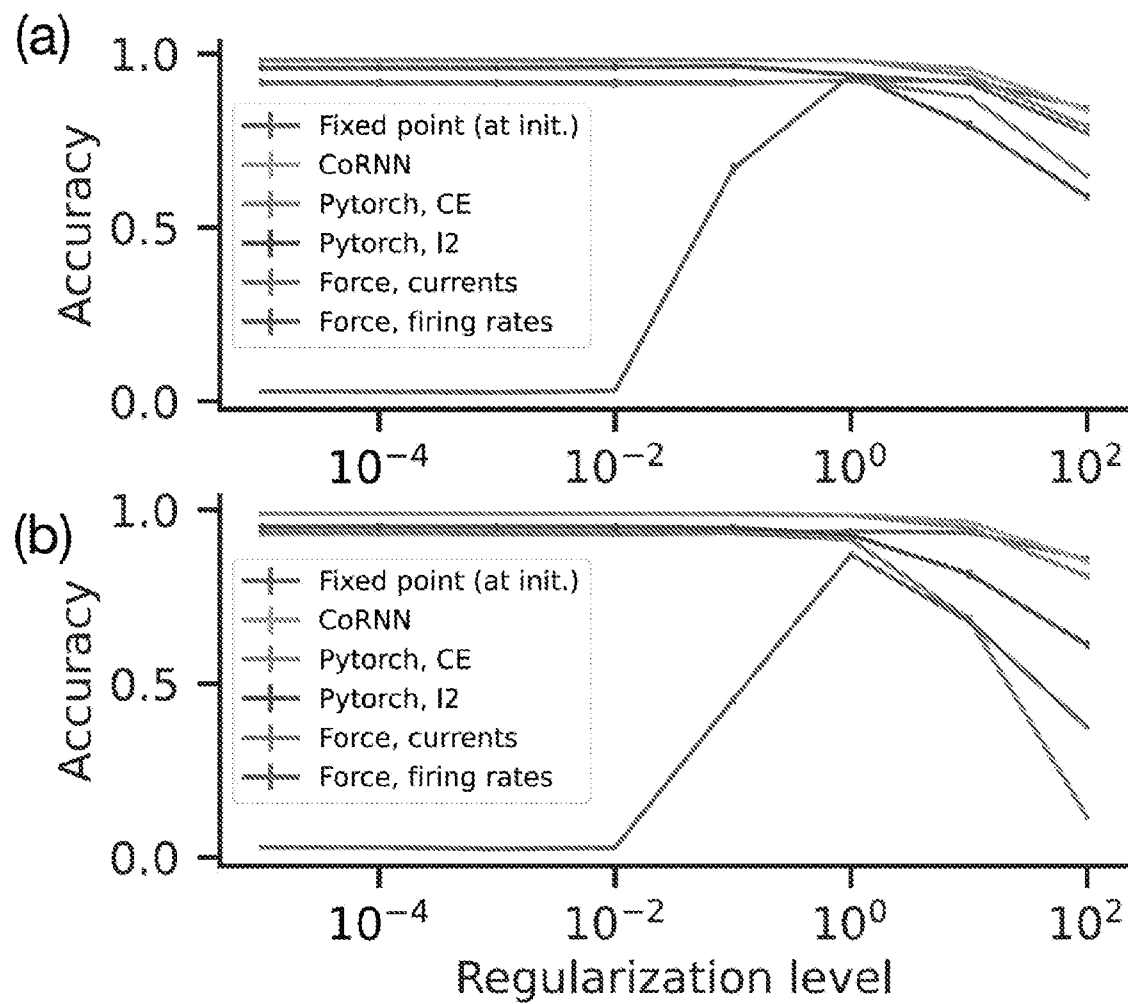
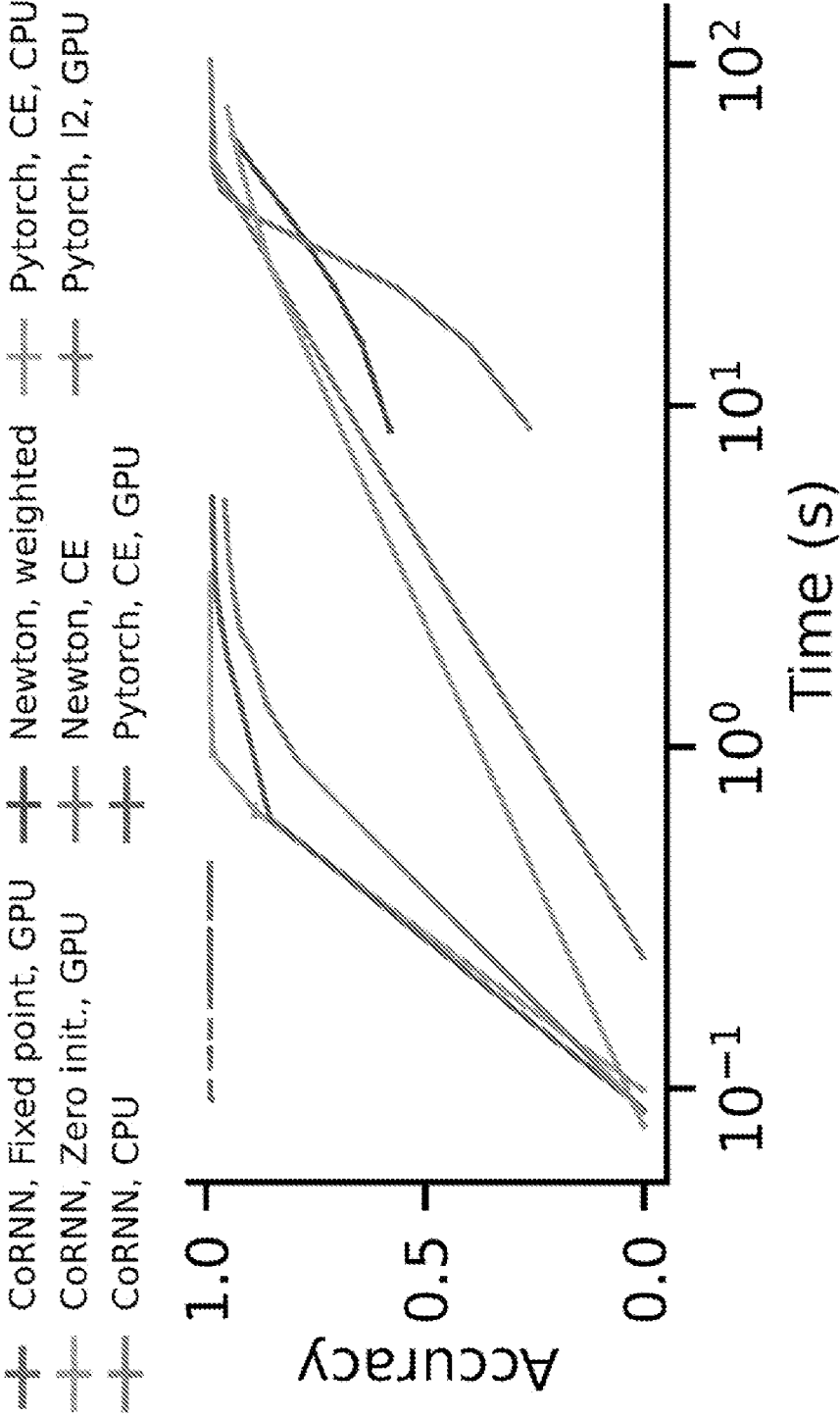
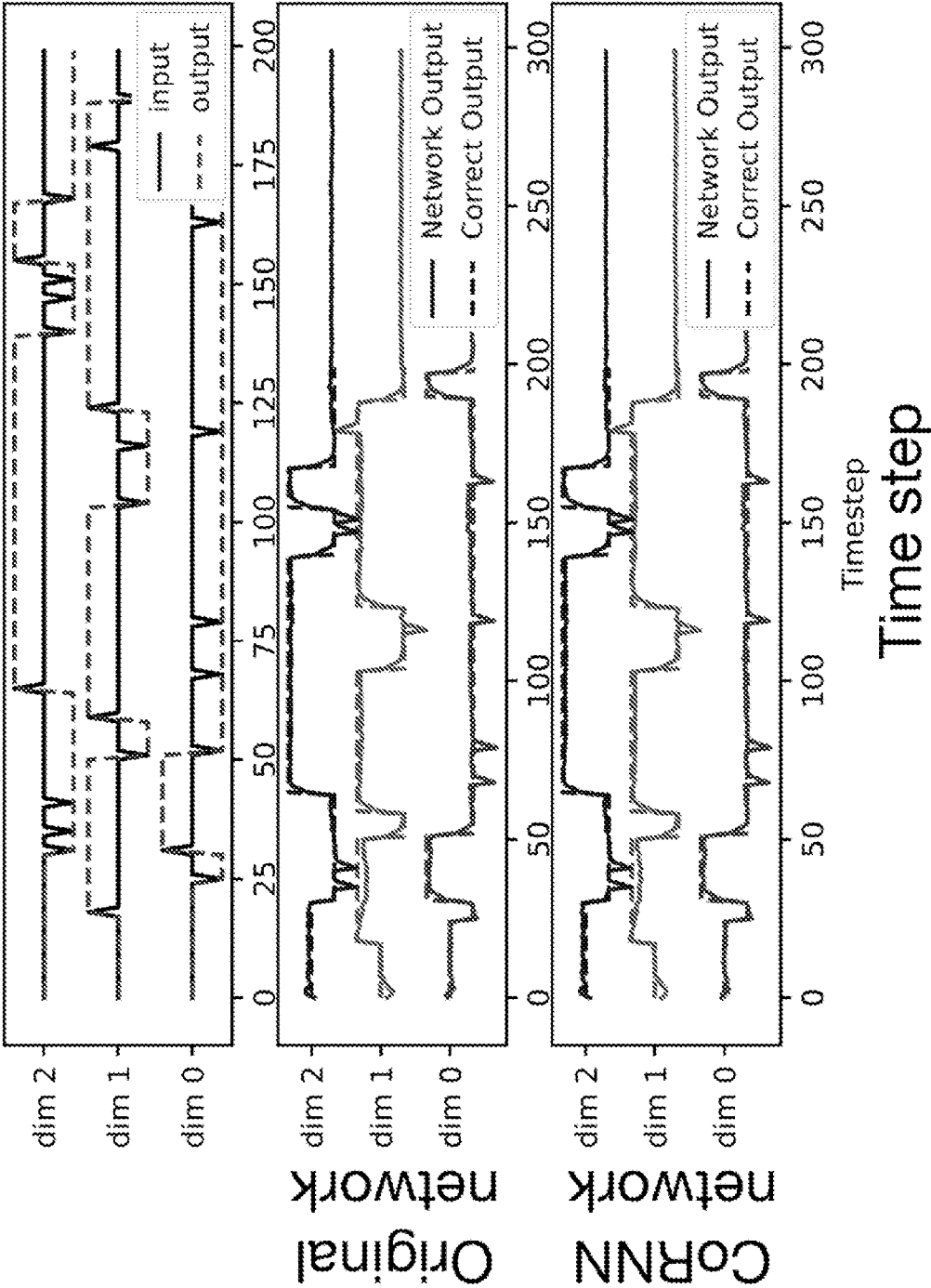


Fig. 5



(a) Input and output for the first trial



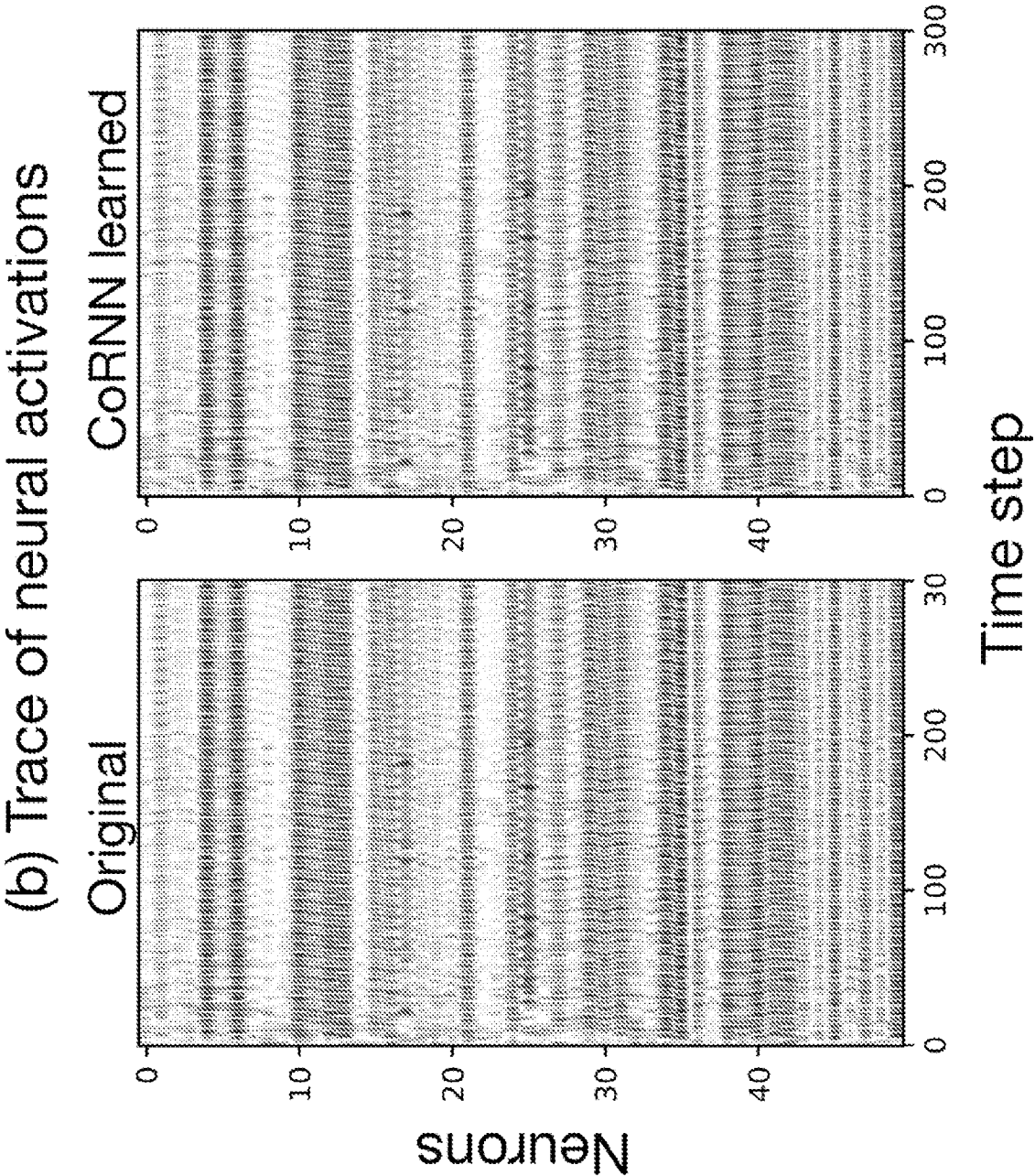


Fig. 6B

Fig. 6C

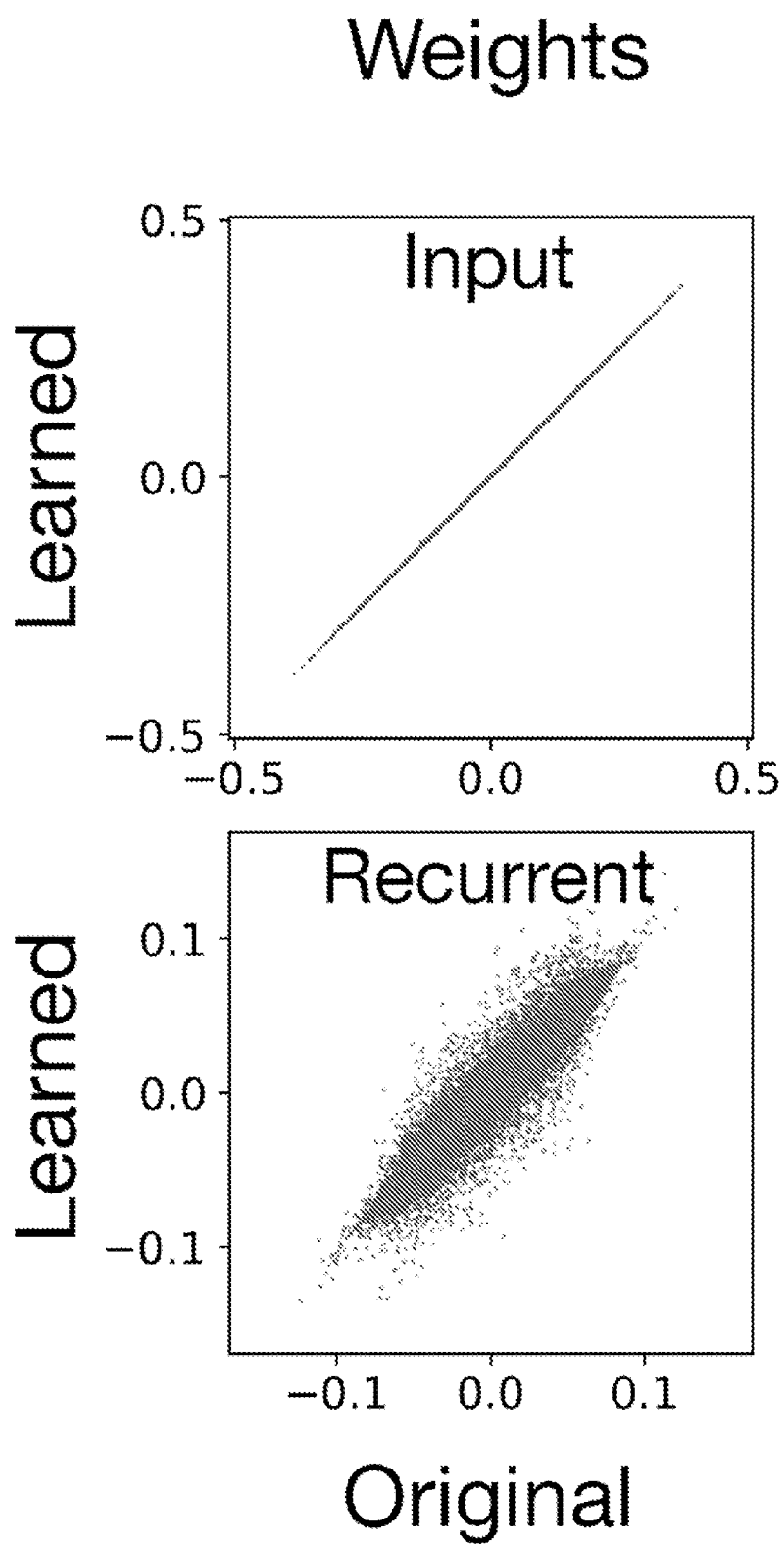


Fig. 7A

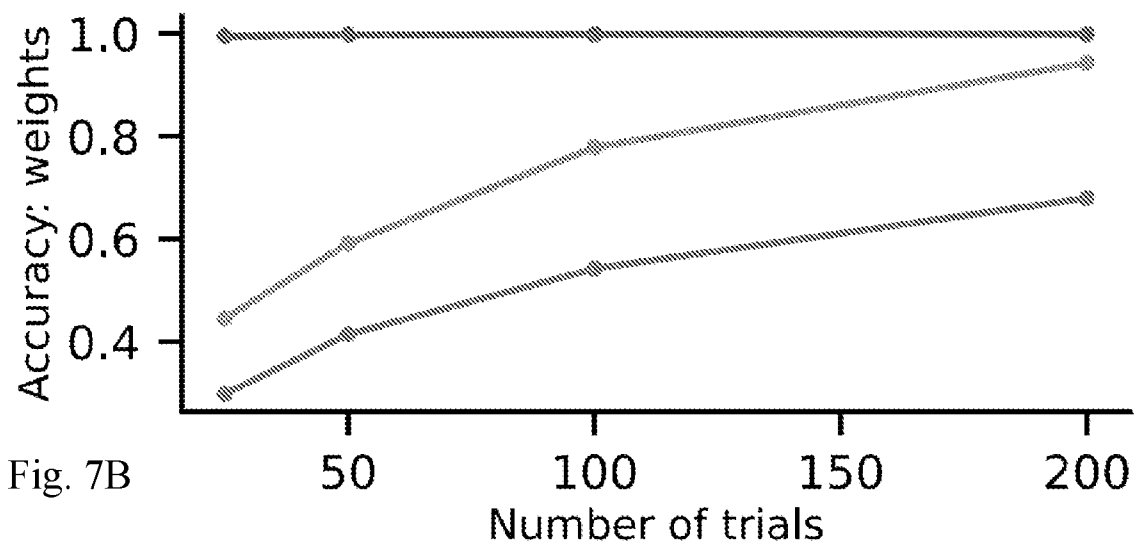
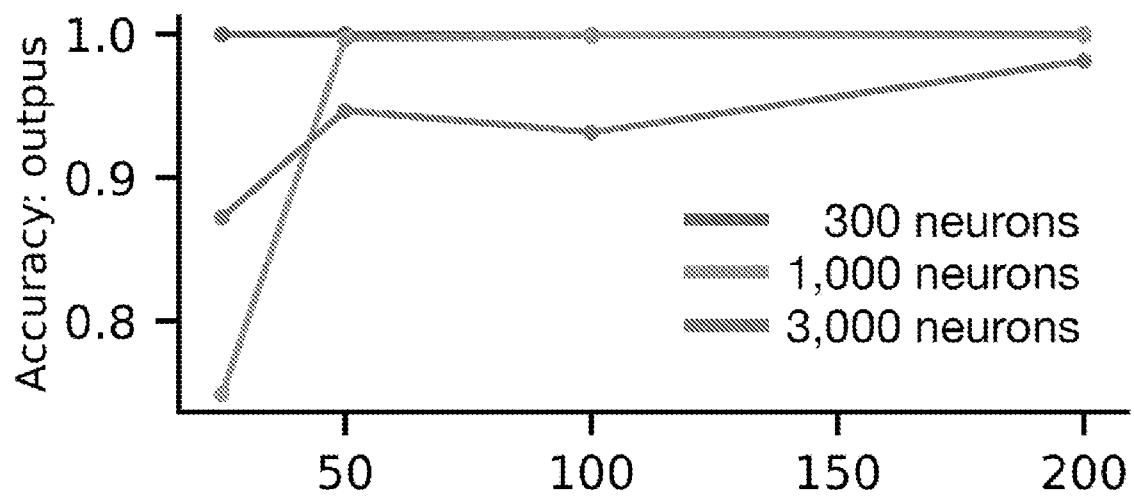


Fig. 7B

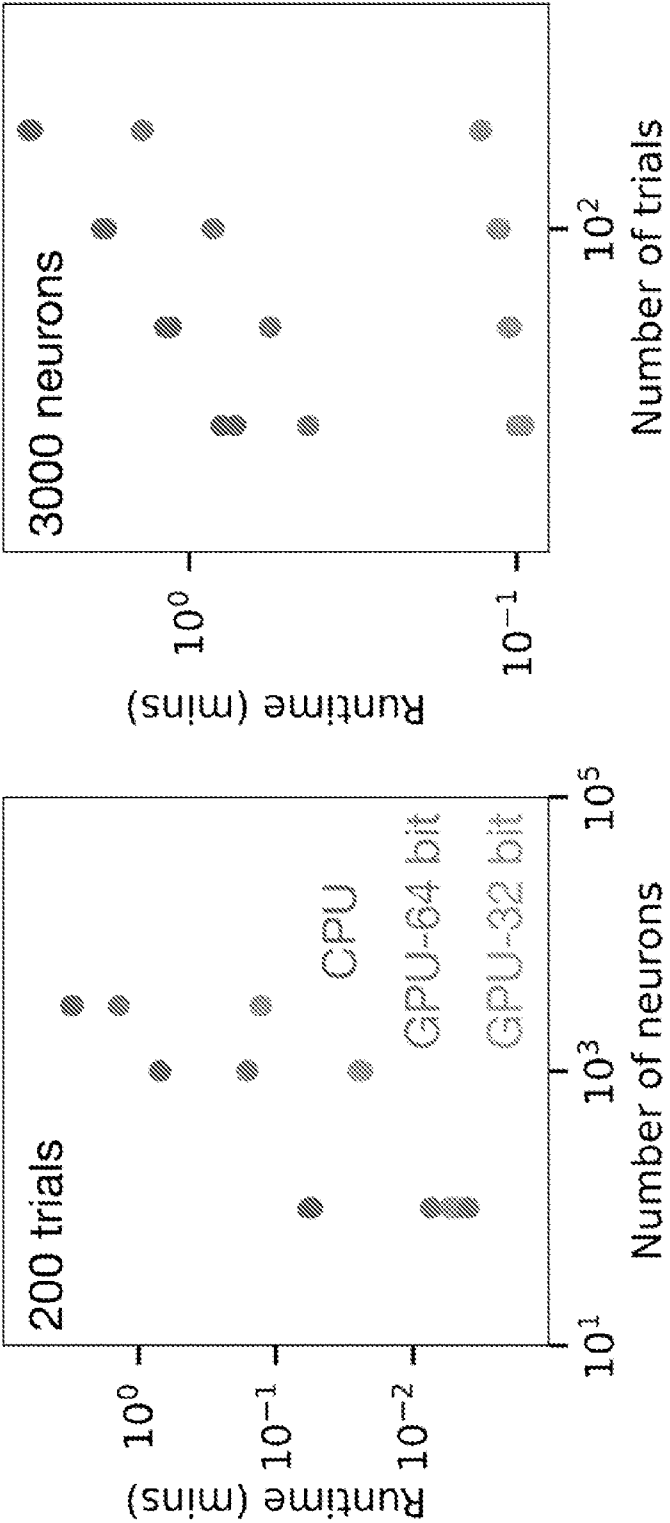


Fig. 8

Fig. 9

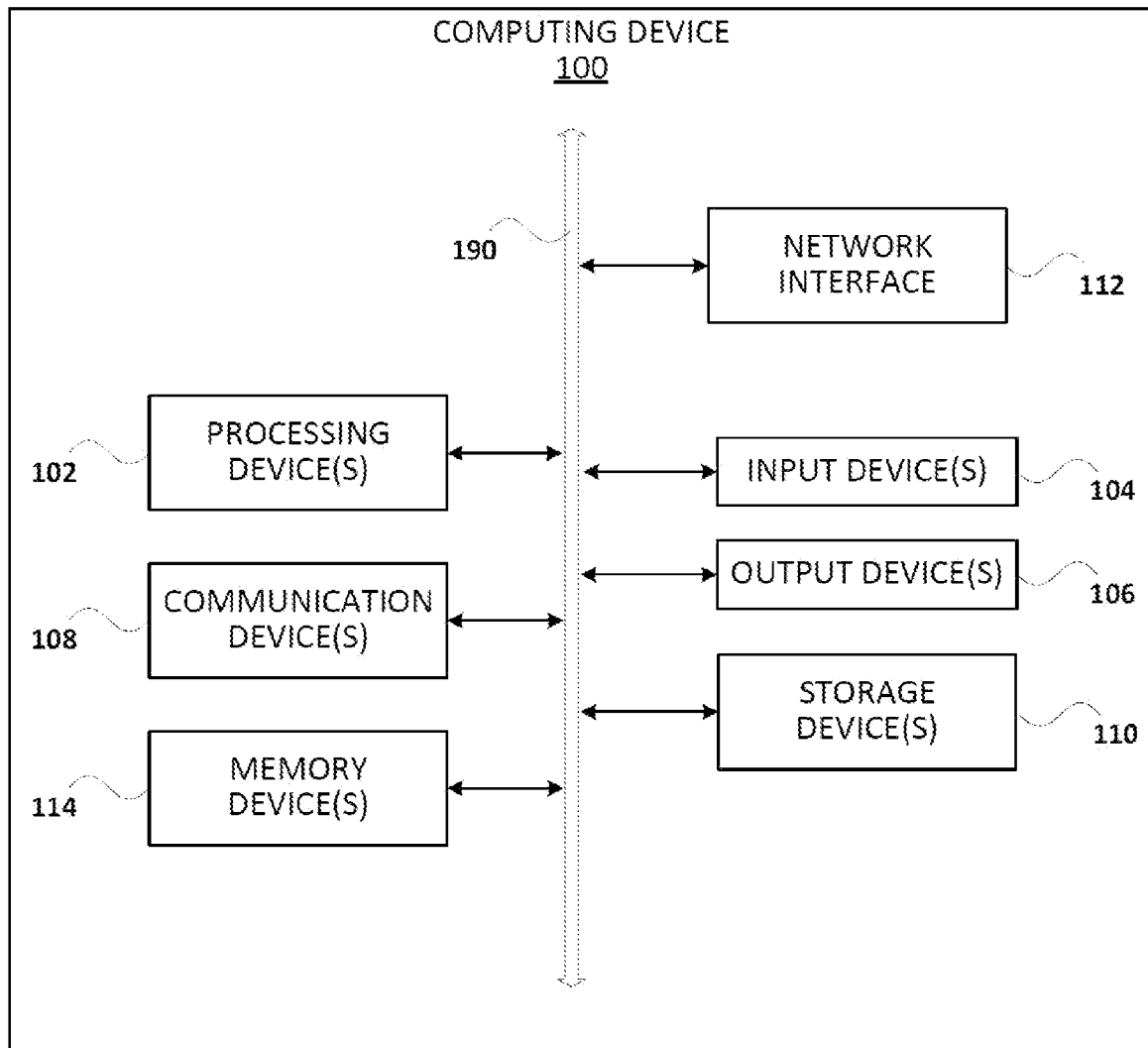


Fig. 10

