A scalable implementation of the recursive least-squares algorithm for training spiking neural networks

Benjamin J. Arthur[1], Christopher M. Kim[1,2], Susu Chen[1], Stephan Preibisch[1], and Ran Darshan[1]

[1]Janelia Research Campus, Howard Hughes Medical Institute, Ashburn, VA, USA
[2]Laboratory of Biological Modeling, NIDDK/NIH, Bethesda, MD, USA

Abstract

Training spiking recurrent neural networks on neuronal recordings or behavioral tasks has become a prominent tool to study computations in the brain. With an increasing size and complexity of neural recordings, there is a need for fast algorithms that can scale to large datasets. We present optimized CPU and GPU implementations of the recursive least-squares algorithm in spiking neural networks. The GPU implementation allows training networks to reproduce neural activity of an order of millions neurons at order of magnitude times faster than the CPU implementation. We demonstrate this by applying our algorithm to reproduce the activity of $> 66,000$ recorded neurons of a mouse performing a decision-making task. The fast implementation enables efficient training of large-scale spiking models, thus allowing for *in-silico* study of the dynamics and connectivity underlying multi-area computations.

Introduction

Cognitive functions involve networks of interconnected neurons with complex dynamics that are distributed over multiple brain areas. One of the fundamental missions of system neuroscience is to understand how complex interactions between large numbers of neurons underlie the basic processes of cognition.

An increasingly popular data-driven modeling approach for investigating the neural mechanisms that support behavioral tasks is to train neurons in an artificial neural network to reproduce the activity of recorded neurons in behaving animals [1, 2, 3, 4, 5, 6]. Such network models can range from purely artificial networks that are far from being biological [2, 3, 4, 7], to biophysical neuronal networks that include spiking activity [8] of different neuronal cell types that operate in a brain-like dynamical state [9]. The neural dynamics and the connectivity structure of the trained network can then be analyzed to gain insights into the underlying neural mechanisms.

With the increase in experimentally recorded neural data, the ability to fit the activity of neurons in model networks to large data sets is becoming a challenge. For example, the number of simultaneously recorded neurons in behaving animals has been increasing in the last few years at an exponential rate [10]. At present, it is possible to simultaneously record in a single session about 1,000 neurons using electrophysiology, and up to 100,000 using calcium imaging in behaving animals [11]. When combining several sessions of recordings, the amount of data becomes huge and can grow to millions of recorded neurons in the near future.

Here, we developed a scalable implementation of the recursive least-squares algorithm (RLS) to train spiking neural networks of large size (e.g., 10s to 100s of thousands of neurons) to reproduce the activity of neural data. RLS [12] (also known as FORCE [7]) was initially applied to train the outputs of a recurrent neural network for performing complex tasks, such as implementing 3-bit memory or generating motor movements. Subsequently, RLS was adopted for training the individual neurons within a recurrent neural network to reproduce target neural activities. Examples of target activities include activity of neurons recorded from the brain [2, 4, 3], chaotic activity of a random network [13], teacher networks [14] and arbitrary functions [8]. Although most existing studies apply RLS to rate-based networks, it can also be implemented in spiking neural networks for performing complex tasks [15] and reproducing neural activities [8, 16, 9].

We report the strategies we took to optimize the code and demonstrate its performance by training more than 66,000 spiking units to reproduce the activity of neurons recorded using Neuropixels probes [17] in a decision making task[18]. Fitting these data, which were sampled at 20ms for 3 seconds, takes about 10 hours on a CPU and less than an hour on a GPU. The code is freely available.

Results

We implemented the RLS algorithm to train individual neurons within a spiking neural network to reproduce the spiking activity of a large number of neurons (Figure 1A). Specifically, we considered networks of integrate-and-fire spiking neurons, in which the neurons could fire irregularly due to the recurrent interactions, known as the fluctuation-driven regime [19, 20, 21], or due to external noise.

The learning objective was to train the synaptic current $u_i(t)$ of each neuron $i = 1, ..., N$ such that it followed a target activity pattern $f_i(t)$ on a time interval $t \in [0, T]$. These activity patterns could be, for example, extracted from the peri-stimulus time histograms (PSTHs) of recorded neurons in the brain (see Appendix: Generating target trajectories). To trigger the target response, each neuron was stimulated by a constant input with random amplitude for a short duration. We treated every neuron's synaptic current as a read-out, which made our task equivalent to training N recurrently connected read-outs. For the current-based synapses considered in this study, neuron i's synaptic current u_i can be expressed in terms of the spiking activities of other neurons $r_j, j = 1, ..., N$ through the exact relationship $u_i = \sum_j W_{ij} r_j$ (see Eqs. (4) and (5) for details). Therefore, we adjusted the incoming synaptic connections $W_{ij}, j = 1, ..., N$ to neuron i by the RLS algorithm in order to generate the target activity. This training scheme allowed us to set up independent objective functions for each neuron and update them in parallel (see Appendix: Recursive least-squares).

Starting with a working implementation of the algorithm [9], we profiled the code to find slow sections, optimized those lines for performance using the strategies below, and iterated until there were no further easy gains to be had. We achieved an order of magnitude improvement in run times using the CPU alone compared to the reference code (Fig. 1C), and another order of magnitude by refactoring to use a GPU. These trends held true when random static connections were replaced with a Gaussian noise model (Fig. 1D). The advantage of this noise model is that run times are relatively independent of firing rates (Fig. 1G).

Optimization strategies

We used the Julia programming language [22] for training the spiking neural networks since rapid prototyping and fine-grained performance optimizations, including custom GPU kernels, can be done in the same language. Several strategies and techniques were used to make the code performant. Benchmarking was performed on synthetic data consisting of sinusoidal target functions with random phases.

Parallel updates of the state variables. We exploited the fact that synaptic weights (for algorithmic details, see Appendix: Recursive least squares) and neuronal voltages and currents can be updated in parallel (except during an action potential) and used multiple threads to loop over the neurons to update these state variables. We benchmarked CPU multi-threading on a machine with 48 physical cores and found that performance plateaued after 16 threads for a large model (Figure 1E). For small network models there was an optimum number of threads, and using more threads was actually slower.

Given that GPUs are purpose-built to thread well, we investigated whether the RLS algorithm would scale better with them. Whereas the CPU run times were linear with model size, GPU

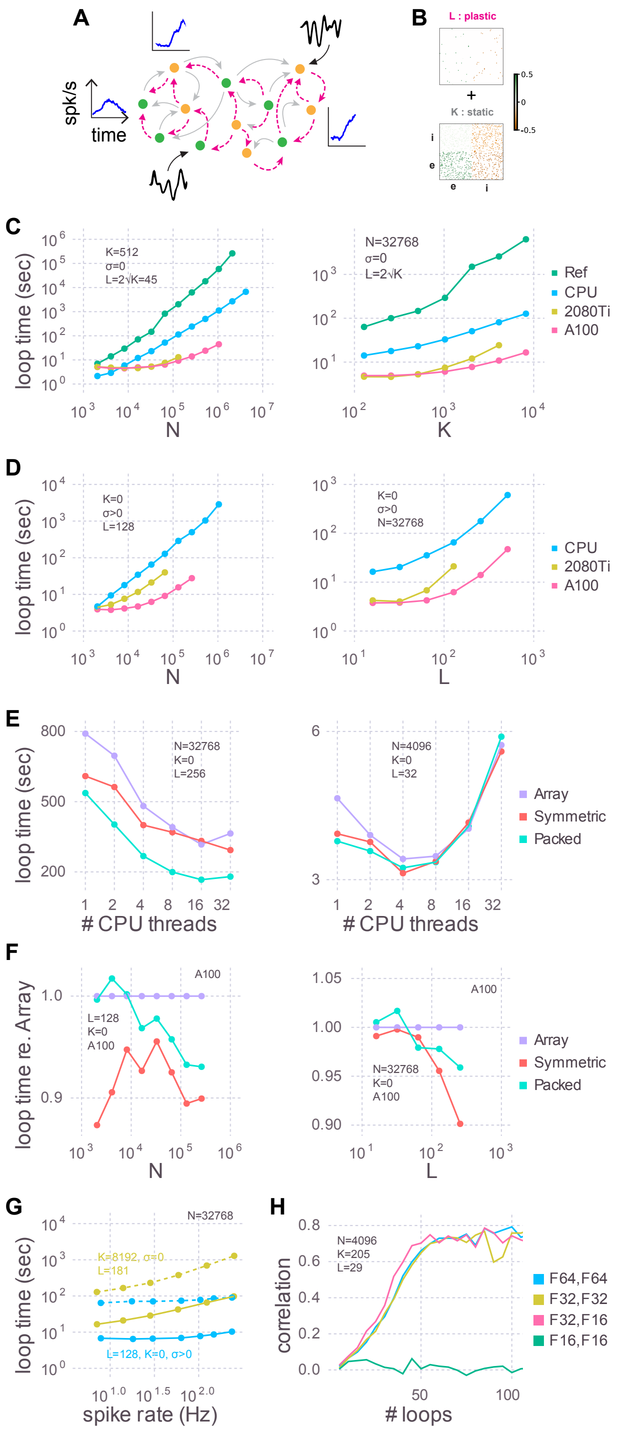

Figure 1: Training speed for models of various sizes. **(A)** Cartoon of a recurrent spiking neural network. Excitatory (green circles) and inhibitory (yellow circles) neurons have plastic (magenta lines) and, optionally, static (grey lines) connections to each other. Each neuron receives, on average, K static and L plastic connections. Associated with each neuron are temporal activity patterns (blue insets; only three shown). For models with no static connections ($K = 0$) we injected into each neuron a white noise with variance σ^2 (black insets; only two shown). **(B)** Plastic (top) and static (bottom) connectivity matrices. **(C)** The time taken by each training iteration for models with static connections. N is the number of neurons. Compared are the single-threaded CPU code used by [9] (Ref), our optimization of the same CPU code (CPU), and our refactoring of the same algorithm for GPUs using a consumer grade card (2080Ti) and an enterprise grade board (A100). In each case we tested the model sizes up to the largest that would fit in memory: 768 GB for the CPU, 11 GB for the 2080Ti, and 80 GB for the A100. The reference code used 64-bit floating point numbers and a full dense array for the large $\boldsymbol{P}$ matrix; all new code presented here uses 32-bit floating point numbers and a packed symmetric array for $\boldsymbol{P}$. **(D)** Similar to **(C)** but with a Gaussian noise model instead of static connections ($K = 0, \sigma > 0$). **(E)** Strong scaling of the optimized CPU code for large and small models. Purple: code uses full $\boldsymbol{P}$ matrix (array). Red: symmetric matrix. Cyan: packed matrix. **(F)** The effect of the matrix storage format on the GPU code as a function of model size. **(G)** The time taken for each training iteration as a function of spike rate. The latter was varied with the external input to neurons (X_{bal} in Eq. (2)). Solid lines indicate GPU code and dotted are the CPU code. **(H)** The effect of storage precision on learning. The first column in the legend indicates the bit precision of all state variables *except* the $\boldsymbol{P}$ matrix, which is indicated in the second column.

performance was flat below a certain size. This is likely because models that are sufficiently small don't use all the parallelism that the GPU provides. The Nvidia A100, for example, has 108 multiprocessors and each one can execute 32 threads simultaneously, yielding a total of 3456 threads.

Symmetric arrays. The RLS algorithm uses the pairwise inverse covariance between trained neurons, which is a symmetric matrix that we denote as $\boldsymbol{P}$ (Eq (9)). While some routines specialized to operate on symmetric matrices are faster, others are slower. Consider the function `syr`, for example, which computes $\boldsymbol{A} = \alpha \boldsymbol{x}\boldsymbol{x}^T + \boldsymbol{A}$, where $\boldsymbol{A}$ is a symmetric matrix, $\boldsymbol{x}$ is a vector, and α is a scalar. Here $\boldsymbol{A}$ is being updated, and since it is symmetric, there are only half as many elements to update compared to `ger` which computes the non-symmetric counterpart. `syr` is hence typically faster than `ger`. Conversely, `symv` computes $\boldsymbol{y} = \alpha \boldsymbol{A}\boldsymbol{x} + \beta \boldsymbol{y}$, where $\boldsymbol{A}$ is a symmetric matrix, $\boldsymbol{x}$ and $\boldsymbol{y}$ are vectors and α and β are scalars. Here every element of $\boldsymbol{A}$ must be accessed to update $\boldsymbol{y}$. Since extra logic must be used to ensure that indexing operations only access a particular triangle, `symv` is typically slower than `gemv`. We found that on balance it was slightly faster to use routines which operate on the symmetric $\boldsymbol{P}$ matrices (Figure 1E,F), particularly for models with large number of plastic inputs.

Further, the size of $\boldsymbol{P}$ is $N \times L^2$, with L being the number of plastic synapses per neuron, making it consume by far more memory than any other variable. Packing the columns of just the upper or lower triangle by concatenating them into a vector saves close to half the memory, thereby permitting models to be proportionately larger. Though a bit slower on the GPU overall compared to their unpacked counterparts (`symv` and `syr`; Figure 1F), BLAS routines specialized for packed symmetric matrices (`spmv` and `spr`) are much faster on the CPU (Figure 1E) for large models.

BLAS. Basic Linear Algebra Subprograms (BLAS) is a highly-engineered library of mathematical operations which are common in high-performance computing. We found that simple refactorings of our CPU code to directly use BLAS resulted in substantial performance gains. For example, the RLS algorithm computes $\boldsymbol{k} = \boldsymbol{P}\boldsymbol{r}$, which is a product of the inverse covariance matrix, $\boldsymbol{P}$, and the synaptically filtered spike trains, $\boldsymbol{r}$ (see Eqs (4) and (9)). Preallocating and reusing $\boldsymbol{k}$ and then calling $mul!(\boldsymbol{k}, \boldsymbol{P}, \boldsymbol{r})$, which is a thin wrapper around BLAS' `gemv` matrix-vector multiplication function, is faster and uses less memory than doing the dot product directly.

A further performance improvement was realized by using Intel's Math Kernel Library, which is a superset of BLAS hand-crafted for the x86 architecture, instead of the default cross-platform OpenBLAS.

For the GPU version of our code we wrote our own GPU kernels which batched several BLAS routines, specifically `gemv` and `ger` plus their symmetric (`symv`, `syr`) and packed symmetric equivalents (`spmv`, `spr`). Writing kernels which internally iterate over N was necessary to overcome what would otherwise be a huge performance penalty in calling a non-batched kernel N times, as the overhead of calling functions is much larger on the GPU than the CPU. Nvidia only provides batched implementations of `gemm`, `gemv`, and `trsm`.

Reduced precision. Our original reference code in [9] used 64-bit floating point precision for all variables, which can represent numbers up to 1.8×10^{308} with a machine epsilon of 2.2×10^{-16} around 1.0. We found that models can be trained just as accurately and with the same number of iterations using 32-bit floats, whose range is only 3.4×10^{38} and machine epsilon is 1.2×10^{-7} (Figure 1H). Doing so not only permits models twice as large to be trained, but also yields shorter loop times on the CPU (data not shown). Models failed to learn no matter how many iterations were used when the representation was further reduced to 16-bits. Fortunately, it is seemingly not the $\boldsymbol{P}$ matrix which needs the extra precision, as learning proceeds normally when P is stored in 16-bits and all other variables in 32-bits.

Pre-computed division. The time constants of the synaptic currents and membrane voltage are in the denominator of the equations that govern the neural dynamics. Since they do not vary, we pre-computed their inverses so that the code could use multiplication instead of division, which is generally slower. Loop times were about 2% shorter doing so for the CPU code.

Application

With a fast RLS codebase in hand, we next set out to show that large models can actually be trained to recapitulate the dynamics in real-world big data sets. We used 66,002 peri-stimulus time histograms (PSTHs) of neurons, recorded using Neuropixels probes [17] from multiple brain areas of mice performing a delayed-response task (Figure 2A; [18, 23]). We first converted the PSTHs to the corresponding underlying synaptic currents by inverting the activation function of integrate-and-fire neurons in the presences of noise (see Appendix: Generating neural trajectories). The synaptic currents were then used as the target functions, and external noise was used instead of static recurrent connections ($K = 0, \sigma > 0$; see Appendix: Network dynamics). Training performance increased with the number of plastic inputs (L), and reached a plateau in half an hour (Figure 2B). Examination of the learned synaptic currents (Figure 2D) and PSTHs (Figure 2C,E) showed a close correspondence with the activity patterns of the recorded neurons.

Discussion

We presented optimized CPU and GPU implementations of the recursive least-square (RLS) algorithm for training spiking neural networks. Our code can simulate and train a spiking network consisting of about one million neurons and 100 million synapses on a modern high-end workstation.

The size of the networks is limited by memory usage, which mainly depends on the size of the inverse covariance matrix ($\boldsymbol{P}$) used in the RLS algorithm (Appendix). This matrix scales linearly with the number of neurons and quadratically with the number of plastic synapses ($N \times L^2$). We found that about $L \approx 100$ synapses per neuron suffices to train the network to reproduce the activity of neurons recorded in mice performing a delayed response task (Figure 2B). However, the number of plastic synapses needed to train the network is expected to increase with the number of tasks to be learned, as well as the complexity of neuronal dynamics in each task. Therefore, how large the network model could be could depend on the number and the complexity of the tasks to be learned.

Finally, our code implementation was tailored to be used on a single computer, instead of on multiple computers, such as in [24]. This enabled fast execution speed, thanks to the absence of inter-process communication overhead, but limited the network size due to memory limitations. Specifically, the CPU implementation uses threads (not processes), which have precisely zero communication overhead because they share memory. The GPU implementation has currently only been tested on a single GPU. However, modern hardware and the associated software toolkits support an abstraction of a unified memory across multiple GPUs within a single computer that is backed by high-speed interconnects. If in future, Moore's law does not outpace advancements in neural recording technology, we plan to investigate how much performance is decremented when using this feature.

Acknowledgements

We would like to thank Shaul Druckmann, Nuo Li, Yi Liu and Karel Svoboda for sharing the Neuropixels data set. CMK would like to thank the Visiting Scientist Program at Janelia Research Campus for their support. This work was supported by the Howard Hughes Medical Institute and the Intramural Research Program of National Institute of Diabetes and Digestive and Kidney Diseases at the National Institutes of Health.

Code availability

The main repository is located at `https://github.com/SpikingNetwork/TrainSpikingNet.jl`. We also release two new Julia packages that it depends on, and put them in separate repositories for easy composability with other code bases: one for packing symmetric matrices and the other for batching (packed) symmetric BLAS routines on the GPU. See `https://github.com/JaneliaSciComp/SymmetricFormats.jl` and `https://github.com/`

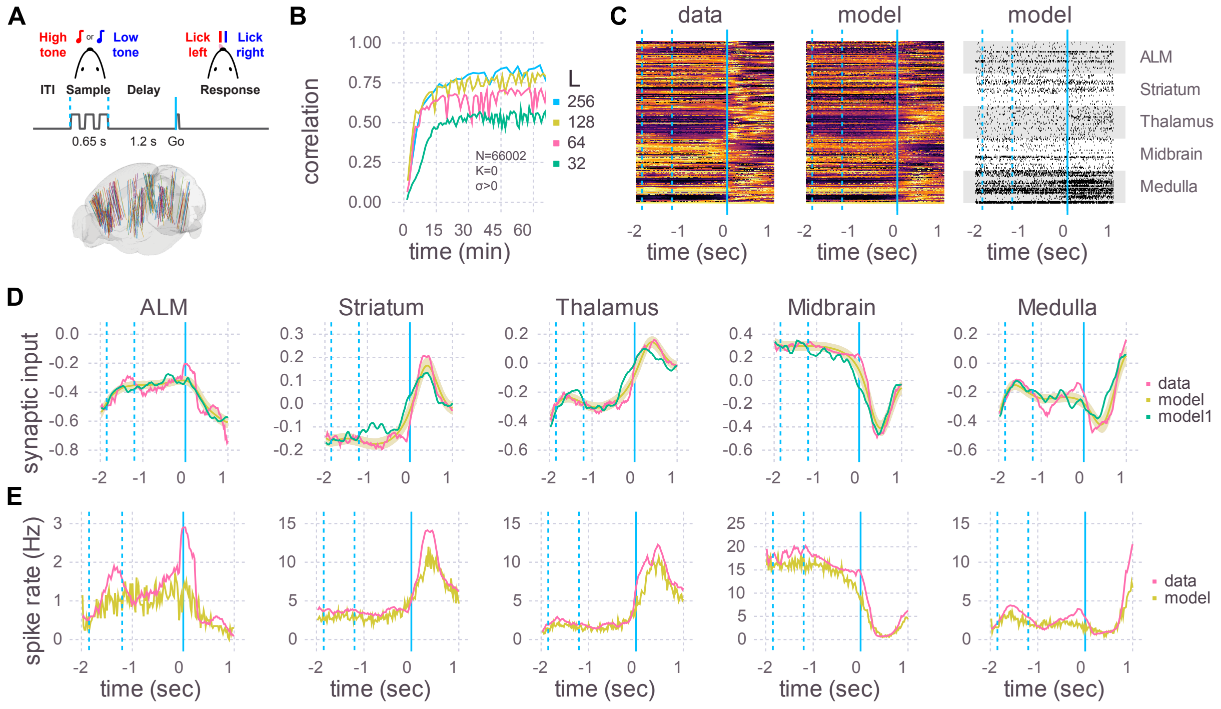

Figure 2: Application to Neuropixels data. **(A)** Top: Schematic of experimental setup. Mice learned to lick to one of two directions (left/right) after a delay period, depending on which of two tones were played. Bottom: Multiple Neuropixels probes, each with hundreds of recording sites, were placed in the brain while a mouse was performing the task. [18] **(B)** A recurrent spiking neural network with 66,002 neurons and no static connections ($K = 0, \sigma > 0$) was trained to learn the neural activity patterns. Initial training rate and final plateau level both increased with the number of plastic connections (L). **(C)** Heat maps of the peri-stimulus time histograms (PSTHs) of the activity patterns in the data (left) and the trained network (middle) and one realization of spike trains of the trained network (right). 50 neurons are shown for each of five brain areas. Vertical blue lines are the same as in **A**. **(D)** Learned synaptic currents averaged over 1000 trials for one exemplar neuron from each of the five brain areas. Vertical blue lines are the same as in **A**. A single realization (non-averaged over trials) is also shown (model1). **(E)** Same as D but for PSTHs.

`JaneliaSciComp/BatchedBLAS.jl`, respectively.

References

[1] Dimitry Fisher, Itsaso Olasagasti, David W Tank, Emre RF Aksay, and Mark S Goldman. A modeling framework for deriving the structural and functional architecture of a short-term memory microcircuit. *Neuron*, 79(5):987–1000, 2013.

[2] Kanaka Rajan, Christopher D. Harvey, and David W. Tank. Recurrent Network Models of Sequence Generation and Memory. *Neuron*, 90(1):128–142, 2016.

[3] Kayvon Daie, Karel Svoboda, and Shaul Druckmann. Targeted photostimulation uncovers circuit motifs supporting short-term memory. *Nature Neuroscience*, 24(2):259–265, 2021.

[4] 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.

[5] Sonja B Hofer, Ho Ko, Bruno Pichler, Joshua Vogelstein, Hana Ros, Hongkui Zeng, Ed Lein, Nicholas A Lesica, and Thomas D Mrsic-Flogel. Differential connectivity and response dynamics of excitatory and inhibitory neurons in visual cortex. *Nature neuroscience*, 14(8):1045–1052, 2011.

[6] Aaron S Andalman, Vanessa M Burns, Matthew Lovett-Barron, Michael Broxton, Ben Poole, Samuel J Yang, Logan Grosenick, Talia N Lerner, Ritchie Chen, Tyler Benster, et al. Neuronal dynamics regulating brain and behavioral state transitions. *Cell*, 177(4):970–985, 2019.

[7] David Sussillo and Larry Abbott. Generating Coherent Patterns of Activity from Chaotic Neural Networks. *Neuron*, 63(4):544–557, 2009.

[8] Christopher M Kim and Carson C Chow. Learning recurrent dynamics in spiking networks. *eLife*, 7:e37124, 2018.

[9] Christopher M Kim, Arseny Finkelstein, Carson C Chow, Karel Svoboda, and Ran Darshan. Distributing task-related neural activity across a cortical network through task-independent connections. *bioRxiv*, 2022.

[10] Ian H Stevenson and Konrad P Kording. How advances in neural recording affect data analysis. *Nature neuroscience*, 14(2):139–142, 2011.

[11] 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.

[12] Simon Haykin. *Adaptive Filter Theory (3rd Ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[13] 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.

[14] 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.

[15] Wilten Nicola and Claudia Clopath. Supervised learning in spiking neural networks with force training. *Nature communications*, 8(1):2208, 2017.

[16] Christopher M. Kim and Carson C. Chow. Training Spiking Neural Networks in the Strong Coupling Regime. *Neural Computation*, 33(5):1199–1233, 04 2021.

[17] James J Jun, Nicholas A Steinmetz, Joshua H Siegle, Daniel J Denman, Marius Bauza, Brian Barbarits, Albert K Lee, Costas A Anastassiou, Alexandru Andrei, Çağatay Aydın, et al. Fully integrated silicon probes for high-density recording of neural activity. *Nature*, 551(7679):232–236, 2017.

[18] Hidehiko K Inagaki, Susu Chen, Margreet C Ridder, Pankaj Sah, Nuo Li, Zidan Yang, Hana Hasanbegovic, Zhenyu Gao, Charles R Gerfen, and Karel Svoboda. A midbrain-thalamus-cortex circuit reorganizes cortical dynamics to initiate movement. *Cell*, 185(6):1065–1081, 2022.

[19] Carl Van Vreeswijk and Haim Sompolinsky. Chaos in neuronal networks with balanced excitatory and inhibitory activity. *Science*, 274(5293):1724–1726, 1996.

[20] Nicolas Brunel. Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons. *Journal of computational neuroscience*, 8(3):183–208, 2000.

[21] Oren Amsalem, Hidehiko Inagaki, Jianing Yu, Karel Svoboda, and Ran Darshan. Subthreshold neuronal activity and the dynamical regime of cerebral cortex. *bioRxiv*, 2022.

[22] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[23] Zengcai V Guo, Nuo Li, Daniel Huber, Eran Ophir, Diego Gutnisky, Jonathan T Ting, Guoping Feng, and Karel Svoboda. Flow of cortical activity underlying a tactile decision in mice. *Neuron*, 81(1):179–194, 2014.

[24] Jakob Jordan, Tammo Ippen, Moritz Helias, Itaru Kitayama, Mitsuhisa Sato, Jun Igarashi, Markus Diesmann, and Susanne Kunkel. Extremely scalable spiking neuronal network simulation code: From laptops to exascale computers. *Frontiers in Neuroinformatics, VOLUME =*, 2018.

[25] Henry Clavering Tuckwell. *Introduction to theoretical neurobiology: linear cable theory and dendritic structure*, volume 1. Cambridge University Press, 1988.

[26] Alex Roxin, Nicolas Brunel, David Hansel, Gianluigi Mongillo, and Carl van Vreeswijk. On the distribution of firing rates in networks of cortical neurons. *Journal of Neuroscience*, 31(45):16217–16226, 2011.

[27] Lior Lebovich, Ran Darshan, Yoni Lavi, David Hansel, and Yonatan Loewenstein. Idiosyncratic choice bias naturally emerges from intrinsic stochasticity in neuronal dynamics. *Nature Human Behaviour*, 3(11):1190–1202, 2019.

Appendix

Generating target neural trajectories from PSTHs. For each recorded neuron, we converted its PSTH to target synaptic activities to be used for training the synaptic inputs. Specifically, in Figure 2 we obtained for each spike rate r_{it} ($i = 1, ..., N$, and $t = T_{init} + \Delta t, ..., T_{end}$ where $T_{init} = -2$ and $T_{end} = 1$) the mean synaptic input f_{it} that needs to be applied to the the leaky integrate-and-fire neuron to generate the desired spike rate. To this end, we numerically solved the nonlinear rate equation

$$r_{it} = \phi(f_{it}, \sigma^2) \tag{1}$$

where

$$\phi(m, \sigma) = \tau_m^{-1}[\sqrt{\pi}\int_{\frac{V_{reset}-m}{\sigma}}^{\frac{V_{thr}-m}{\sigma}} dw e^{w^2} \mathrm{erfc}(-w)]^{-1}$$

is the transfer function of the leaky integrate-and-fire neuron given mean input, m, and variance of the input, σ^2 [25, 26]. This conversion yielded a set of vectors of target synaptic inputs $\mathbf{f}_1, ..., \mathbf{f}_N \in \mathbb{R}^T$ where $T = (T_{end} - T_{init})/\Delta t$ for the recorded neurons. Note that in Figure 2 we did not simulate the recurrent static connections ($K = 0$). Simulating both static connections and the external noise would require to estimate σ in Eq.(1). This can be done, for example, by estimating the synaptic noise in the neurons of the initial network [9].

Network connectivity. In Figure 1A, the spiking neural network consisted of randomly connected N_E excitatory and N_I inhibitory neurons. They were recurrently connected as in [9]. In short, the recurrent synapses consisted of static weights $\mathbf{J}$ that remained constant throughout training and plastic weights $\mathbf{W}$ that were modified by the training algorithm (Figure 1B). The static synapses connected neuron j in population β to neuron i in population α with probability $p_{\alpha\beta} = K_{\alpha\beta}/N_\beta$ and synaptic weight $\bar{J}_{\alpha\beta}/\sqrt{K_{\alpha\beta}}$, where $K_{\alpha\beta}$ is the average number of static connections from population β to α:

$$Pr(J_{ij}^{\alpha\beta} \neq 0) = \frac{K_{\alpha\beta}}{N_\beta}.$$

The strength of plastic synapses, $\bar{W}_{\alpha\beta}/\sqrt{K_{\alpha\beta}}$, was of the same order as the static weights. However, the plastic synapses connected neurons with a smaller probability:

$$Pr(W_{ij}^{\alpha\beta} \neq 0) = \frac{L_{\alpha\beta}}{N_\beta} \quad \text{with} \quad L_{\alpha\beta} = c\sqrt{K_{\alpha\beta}}$$

which made the plastic synapses much sparser than the static synapses [27, 9]. Here, c is an order 1 parameter that depends on training setup.

The static and plastic connections were non-overlapping in that any two neurons in the network can have only one type of synapse.

$$J_{ij}^{\alpha\beta} W_{ij}^{\alpha\beta} = 0.$$

Keeping them disjoint allowed us to maintain the initial network dynamics generated by the static synapses and, subsequently, introduce trained activity to the initial dynamics by modifying the plastic synapses.

The static recurrent synapses were *strong* in that the coupling strength between two connected neurons scaled as $\frac{1}{\sqrt{K_{\alpha\beta}}}$, while the average number of synaptic inputs scaled as $K_{\alpha\beta}$. As a result of this strong scaling, the excitatory and inhibitory synaptic inputs to a neuron from static synapses increased as $\sqrt{K_{\alpha\beta}}$, and thus were much larger than the spike-threshold for a large $K_{\alpha\beta}$. However, the excitatory and inhibitory currents were dynamically canceled, and, together with the external input, the sum was balanced to be around the spike-threshold [19, 9].

In contrast to the static synapses, each trained neuron received only about $\sqrt{K_{\alpha\beta}}$ plastic synapses. This made the plastic synapses much sparser than the sparse static EI connectivity (e.g., with $K = 1000$ static synapses, there are of the order of $\sqrt{K} \approx 30$ plastic synapses per neuron). Consequently, the EI plastic inputs of the initial network were independent of $K_{\alpha\beta}$ and substantially weaker than the EI balanced inputs for a large $K_{\alpha\beta}$. After training the plastic synapses, the total synaptic input, i.e., the sum of plastic and balanced inputs, to each trained neuron was able to follow the target patterns. With this scaling of plastic synapses, training was robust to variations in the network size, N, and the number of synaptic connections, $K_{\alpha\beta}$ (Fig. 1C).

In Figures 1D, F and Figure 2, on the other hand, the network had only plastic synapses but no static synapses, i.e., $K = 0$. Without the static excitatory-inhibitory synaptic connections, as in Figure 1, the network lacked internally generated noise. Therefore, we injected external noise to the neuron's voltage equation to induce variability in spiking activity (Fig. 2C). The variance of this white noise, σ^2, was chosen such that the externally injected noise was similar to internally generated fluctuating synaptic current when static connections were present ($K = 1000$).

Network dynamics. In the following mathematical description of network activity, the static connections present in Figure 1C ($K > 0$) produced the balanced inputs ($u_{bal,i}^\alpha$), which exhibited large fluctuations. For this reason, no additional external noise was injected to the network ($\sigma = 0$). On the other hand, as mentioned above, in Figure 2 (and also in Figs. 1D, F), the static connections did not exist ($K = 0$) and external noise was injected to the neurons ($\sigma > 0$).

We used integrate-and-fire neuron to model the membrane potential dynamics of i'th neuron:

$$\tau_m \dot{v}_i^\alpha = -v_i^\alpha + u_i^\alpha + X_i^\alpha + \sigma\xi_i$$

where a spike is emitted and the membrane potential is reset to v_{reset} when the membrane potential crosses the spike-threshold v_{thr}.

Here, u_i^α is the total synaptic input to neuron i in population α that can be divided into static and plastic inputs incoming through the static and plastic synapses, respectively:

$$u_i^\alpha = u_{bal,i}^\alpha + u_{plas,i}^\alpha.$$

X_i^α is the total external input that can be divided into constant external input, plastic external input, and the stimulus:

$$X_i^\alpha = X_{bal,i}^\alpha + X_{plas,i}^\alpha + X_{stim,i}^\alpha. \qquad (2)$$

$X_{bal,i}^\alpha$ is a constant input associated with the initial balanced network. It scales with the number of connections, i.e., proportional to $\sqrt{K_{\alpha\beta}}$, determines the firing rate of the initial network and stays unchanged [19]. $X_{plas,i}^\alpha$ is plastic input provided to trained neurons in the recurrent network from external neurons that emit stochastic spikes with pre-determined rate patterns. The synaptic weights from the external neurons to the trained neurons were modified by the training algorithm. $X^{\alpha}_{stim,i}$ is the pre-determined stimulus, generated independently from the Ornstein-Ulenbeck process for each neuron, and injected to all neurons in the network to trigger the learned responses in the trained neurons.

The synaptic activity was modeled by instantaneous jump of the synaptic input due to presynaptic neuron's spike, followed by exponential decay. Since the static and plastic synapses did not overlap, we separated the total synaptic input into static and plastic components as mentioned above:

$$\tau_{bal}\dot{u}^{\alpha}_{bal,i} = -u^{\alpha}_{bal,i} + \sum_{\beta\in\{E,I\}}\sum_{j\in\beta} J^{\alpha\beta}_{ij}\sum_{t^j_k<t}\delta(t-t^j_k)$$
$$\tau_{plas}\dot{u}^{\alpha}_{plas,i} = -u^{\alpha}_{plas,i} + \sum_{\beta\in\{E,I\}}\sum_{j\in\beta} W^{\alpha\beta}_{ij}\sum_{t^j_k<t}\delta(t-t^j_k). \tag{3}$$

Alternatively, the synaptic activity can be expressed as a weighted sum of filtered spike trains because the synaptic variable equations (Eq. 3) are linear in $\mathbf{J}$ and $\mathbf{W}$:

$$u^{\alpha}_{bal,i} = \sum_{\beta,j} J^{\alpha\beta}_{ij} r^{\beta}_{bal,j}$$
$$u^{\alpha}_{plas,i} = \sum_{\beta,j} W^{\alpha\beta}_{ij} r^{\beta}_{plas,j} \tag{4}$$

where

$$\tau_{bal}\dot{r}^{\beta}_{bal,i} = -r^{\beta}_{bal,i} + \sum_{t^i_k<t}\delta(t-t^i_k)$$
$$\tau_{plas}\dot{r}^{\beta}_{plas,i} = -r^{\beta}_{plas,i} + \sum_{t^i_k<t}\delta(t-t^i_k)$$

describe the dynamics of synaptically filtered spike trains.

Each external neuron emitted spikes stochastically at a pre-defined rate that changed over time. The rate patterns, followed by the external neurons, were randomly generated from an Ornstein-Ulenbeck process with mean rate of 5 Hz. The synaptically filtered external spikes were weighted by plastic synapses $\mathbf{W_X}$ and injected to trained neurons:

$$X^{\alpha}_{plas,i} = \sum_j W^X_{ij} r^X_j \tag{5}$$

where

$$\tau_{plas}\dot{r}^{X}_{plas,i} = -r^{X}_{plas,i} + \sum_{t^i_k<t}\delta(t-t^i_k)$$

Similarly, the external stimulus $X_{stim,i}$ applied to each neuron i in the network to trigger the learned response is generated independently from the Ornstein-Ulenbeck process.

In the following section, we will use the linearity of $\mathbf{W}, \mathbf{W_X}$ in Eqs. 4 and 5 to derive the training algorithm that modifies plastic synaptic weights.

Recursive least squares. We derive a synaptic update rule that modifies the plastic synapses to learn the target activities. The derivation presented here closely follows previous papers [8, 16, 9]. For notational simplicity, we drop the neuron index i in $\mathbf{w}_i$ and other variables, e.g., f_i, u_i, but the same synaptic update rule is applied to all the trained neurons. In fact, the plastic connections to every trained neuron can be updated simultaneously since each trained neuron has its own private target function and plastic connections. *Therefore, the CPU multithreading (Fig. 1E) or GPU implementation (Figs. 1C, D) presented in our study leverages the fact that the synaptic update rule can be applied in parallel to all plastic synapses.*

The gradient of the cost function with respect to the weights $\mathbf{w}$ is

$$\begin{aligned}\nabla_{\mathbf{w}} C &= \frac{1}{2}\nabla_{\mathbf{w}}\Big[\sum_t (f_t - u_t)^2 + \lambda\|\mathbf{w}\|^2 + \mu\Big((\mathbf{w}\cdot\mathbf{1}_E)^2 + (\mathbf{w}\cdot\mathbf{1}_I)^2\Big)\Big] \\ &= \sum_t (-f_t\mathbf{r}_t + \mathbf{r}_t\mathbf{r}_t'\mathbf{w}) + \lambda\mathbf{w} + \mu(\mathbf{1}_E\mathbf{1}_E' + \mathbf{1}_I\mathbf{1}_I')\mathbf{w}\end{aligned}$$

where we substitute the expression $u_t = \mathbf{w}\cdot\mathbf{r}_t$ in the first line to evaluate the gradient with respect to $\mathbf{w}$. To derive the synaptic update rule, we compute the gradient at two consecutive time points

$$\mathbf{0} = \nabla_{\mathbf{w}_n} C = \sum_{t=1}^{n} (-f_t\mathbf{r}_t + \mathbf{r}_t\mathbf{r}_t'\mathbf{w}_n) + \lambda\mathbf{w}_n + \mu(\mathbf{1}_E\mathbf{1}_E' + \mathbf{1}_I\mathbf{1}_I')\mathbf{w}_n \tag{6}$$

and

$$\mathbf{0} = \nabla_{\mathbf{w}_{n-1}} C = \sum_{t=1}^{n-1} (-f_t\mathbf{r}_t + \mathbf{r}_t\mathbf{r}_t'\mathbf{w}_{n-1}) + \lambda\mathbf{w}_{n-1} + \mu(\mathbf{1}_E\mathbf{1}_E' + \mathbf{1}_I\mathbf{1}_I')\mathbf{w}_{n-1}. \tag{7}$$

Subtracting Eqs (6) and (7) yields

$$\begin{aligned}\mathbf{w}_n &= \mathbf{w}_{n-1} + e_n P_n \mathbf{r}_n \\ e_n &= f_n - \mathbf{w}_{n-1}\cdot\mathbf{r}_n\end{aligned} \tag{8}$$

where

$$P^n = \left[\sum_{t=1}^{n} \mathbf{r}_t(\mathbf{r}_t)' + \lambda I + \mu\mathbf{1}_E\mathbf{1}_E' + \mu\mathbf{1}_I\mathbf{1}_I'\right]^{-1} \quad \text{for} \quad n \geq 1 \tag{9}$$

with the initial value

$$P^0 = [\lambda I + \mu\mathbf{1}_E\mathbf{1}_E' + \mu\mathbf{1}_I\mathbf{1}_I']^{-1}. \tag{10}$$

To update P^n iteratively, we use the Woodbury matrix identity

$$(A + UCV)^{-1} = A^{-1} - A^{-1}U(C^{-1} + VA^{-1}U)^{-1}VA^{-1} \tag{11}$$

where A is invertible and $N\times N$, U is $N\times T$, C is invertible and $T\times T$ and V is $T\times N$ matrices. Then P^n can be calculated iteratively

$$P^n = P^{n-1} - \frac{P^{n-1}\mathbf{r}_n(\mathbf{r}_n)'P^{n-1}}{1 + (\mathbf{r}_n)'P^{n-1}\mathbf{r}_n}.$$