Frontiers in Neuroinformatics

TYPE Technology and Code
PUBLISHED 27 June 2023
DOI 10.3389/fninf.2023.1099510

EDITED BY
Matthias H. Hennig,
University of Edinburgh, United Kingdom

REVIEWED BY
Russell Jarvis,
Western Sydney University, Australia
Hans Ekkehard Plesser,
Norwegian University of Life Sciences, Norway

*CORRESPONDENCE
Benjamin J. Arthur
✉ arthurb@janelia.hhmi.org

†PRESENT ADDRESS
Ran Darshan,
Department of Physiology and Pharmacology, Sackler Faculty of Medicine, Tel Aviv University, Tel Aviv, Israel;
Sagol School of Neuroscience, Tel Aviv University, Tel Aviv, Israel;
The School of Physics and Astronomy, Tel Aviv University, Tel Aviv, Israel

RECEIVED 15 November 2022
ACCEPTED 05 June 2023
PUBLISHED 27 June 2023

CITATION
Arthur BJ, Kim CM, Chen S, Preibisch S and Darshan R (2023) A scalable implementation of the recursive least-squares algorithm for training spiking neural networks.
Front. Neuroinform. 17:1099510.
doi: 10.3389/fninf.2023.1099510

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, United States, [2]Laboratory of Biological Modeling, National Institute of Diabetes and Digestive and Kidney Diseases, National Institutes of Health, Bethesda, MD, United States

Training spiking recurrent neural networks on neuronal recordings or behavioral tasks has become a popular way to study computations performed by the nervous system. As the size and complexity of neural recordings increase, there is a need for efficient algorithms that can train models in a short period of time using minimal resources. We present optimized CPU and GPU implementations of the recursive least-squares algorithm in spiking neural networks. The GPU implementation can train networks of one million neurons, with 100 million plastic synapses and a billion static synapses, about 1,000 times faster than an unoptimized reference CPU implementation. We demonstrate the code's utility by training a network, in less than an hour, to reproduce the activity of $> 66,000$ recorded neurons of a mouse performing a decision-making task. The fast implementation enables a more interactive *in-silico* study of the dynamics and connectivity underlying multi-area computations. It also admits the possibility to train models as *in-vivo* experiments are being conducted, thus closing the loop between modeling and experiments.

KEYWORDS

integrate and fire neuron, dynamical system, Neuropixels dense silicon probe, balanced networks, excitation-inhibition

1. 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 (Hofer et al., 2011; Fisher et al., 2013; Rajan et al., 2016; Andalman et al., 2019; Daie et al., 2021; Finkelstein et al., 2021). Such network models can range from purely artificial networks that are far from being biological (Sussillo and Abbott, 2009; Rajan et al., 2016; Daie et al., 2021; Finkelstein et al., 2021), to biophysical neuronal networks that include spiking activity (Kim and Chow, 2018) of different cell types that operate in a brain-like dynamical state (Kim et al., 2023). The neural dynamics and the connectivity structure of the trained network can then be analyzed to gain insights into the underlying neural mechanisms. For instance, Rajan et al. (2016) demonstrated that memory-related sequential activity can be produced in highly heterogeneous but partially structured recurrent neural networks, Finkelstein et al. (2021) showed that decision-related information can be gated from distracting inputs by forming increasingly stable fixed points that move away from the decision boundary, and Kim et al. (2023) provided general circuit mechanisms for spreading task-related neural activities from a small subset of trained neurons to the rest of neurons in the network.

With the increase in the size of experimentally recorded neural data sets, the ability to fit the activity of neurons in model networks 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 (Stevenson and Kording, 2011). At present, it is possible to simultaneously record in a single session about 1,000 neurons using electrophysiology, and up to 100,000 neurons using calcium imaging in behaving animals (Urai et al., 2022). When combining several sessions of recordings, the amount of data becomes huge and will likely 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 tens to hundreds of thousands of neurons to reproduce the activity of neural data. RLS (Haykin, 1996) 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 (also known as FORCE; Sussillo and Abbott, 2009). 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 (Rajan et al., 2016; Daie et al., 2021; Finkelstein et al., 2021), chaotic activity of a random network (Laje and Buonomano, 2013), teacher networks (DePasquale et al., 2018) and arbitrary functions (Kim and Chow, 2018). Although most existing studies apply RLS to rate-based networks, it can also be implemented in spiking neural networks for performing complex tasks (Nicola and Clopath, 2017) and reproducing neural activities (Kim and Chow, 2018, 2021; Kim et al., 2023).

Starting with the code in Kim et al. (2023), we generalized it to include different models of single cell dynamics, as well as extended it to account for external noise. We then optimized it for run-time speed. Finally, we demonstrated its performance by training more than 66,000 spiking neurons to reproduce the activity of recordings from multiple brain areas of mice using Neuropixels probes (Jun et al., 2017) in a decision making task (Inagaki et al., 2022; Chen et al., 2023). Fitting these data, which were sampled at 20 ms for 3 s, takes about 10 h on a multi-core CPU and less than an hour on a GPU. The code is freely available.

2. Results

We implemented the RLS algorithm to train individual neurons within a large spiking recurrent neural network to reproduce a pre-determined target activity (Figure 1A). Specifically, we considered networks of integrate-and-fire spiking neurons connected by synapses with varying spike-filtering time scales but no explicit spike-propagation delays, in which the neurons could fire irregularly due either to recurrent interactions, known as the fluctuation-driven regime (Van Vreeswijk and Sompolinsky, 1996; Brunel, 2000; Amsalem et al., 2022), or to external noise, or both. We chose the standard leaky integrate-and-fire neuron as our model neuron. In the code, the cell model is a plugin, making it easy for users to customize, and we provide code that implements all five of the Generalized Leak Integrate and Fire (GLIF) models described in Teeter et al. (2018).

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 $t \in [0, T]$ (see Appendix: recursive least squares in Supplementary material). These activity patterns were extracted from the peri-stimulus time histograms (PSTHs) of recorded neurons in the brain (see Appendix: generating target trajectories in Supplementary material). To trigger a target response, each neuron in the network was stimulated by a constant input with random amplitude for a short duration. These external stimuli were applied to all neurons simultaneously, such that the network was set to a specific state at the end of stimulus duration. The synaptic weights were trained with this stimulated network state as the initial state, which allowed the trained network to produce the target response whenever the stimulus was applied to reset the network state. 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 Equations A9 and A10 in Supplementary material 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. Specifically, we randomly chose L plastic connections per neuron and trained only these connections. This training scheme allowed us to set up independent objective functions for each neuron and update them in parallel (see Appendix: recursive least-squares in Supplementary material). The linearity of u in terms of r makes it possible to directly apply the RLS algorithm to train the synaptic activity. Alternatively, the firing rates of neurons can also be trained if an appropriate F-I curve (Equation A1 in Supplementary material) for the neuron model is used in the RLS algorithm. This procedure, however, involves differentiating the non-linear F-I curve, which can slow down learning during subthreshold activity, as shown previously (Kim and Chow, 2018).

Besides the plastic connections, each neuron can also receive an average of K pre-synaptic static inputs. These connections are referred to as "static" because they remain unchanged during the learning process. The average static synaptic weights were chosen such that the network was in the balanced state (see Appendix in Supplementary material and Kim et al., 2023 for further details). In addition, an external noise was optionally injected into each neuron, with a variance of σ^2.

In the case where the neurons are driven by external noise and there are no static connections ($K = 0, \sigma > 0$), the number of plastic weights (L) is only bounded by N (and memory). However, in the presence of static weights, we follow our previous work (Kim et al., 2023) and choose them such that $L = \mathcal{O}(\sqrt{K})$. As discussed in Kim et al. (2023), this relationship, along with the scaling of $1/L$

for each plastic synapse, which is comparable to the static synapses (i.e., $1/\sqrt{(K)}$), facilitates learning without disturbing the balanced state of the network.

Starting with a working implementation of the algorithm (Kim et al., 2023), 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 almost two orders of magnitude improvement in run times for large networks using the CPU alone compared to the reference code (Figure 1B), and another order of magnitude or two by refactoring to use a GPU. These trends held true when random static connections were replaced with a Gaussian noise model (Figure 1C). The advantage of this noise model is that run times are relatively independent of firing rates (Figure 1F).

2.1. Optimization strategies

We used the Julia programming language (Bezanson et al., 2017) 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, both in terms of run time as well as memory use. Benchmarking was performed on synthetic data consisting of sinusoidal target functions with random phases.

2.1.1. Parallel updates of the state variables

Pre-synaptic weights and neuronal voltages can be updated in parallel as they are independent of each other. Synaptic currents can also be updated in parallel, except during an action potential, when different threads might overwrite each others updates to a post-synaptic neuron's current if multiple of its pre-synaptic neurons spike simultaneously. In the CPU code, we used multiple threads to loop over the neurons to update the weights, voltages, and the exponential decay of currents. As Julia does not support atomic operations on elements of a vector, and locking mechanisms can be slow, a subsequent non-threaded loop was used to update the post-synaptic currents to avoid the race condition when a spike occurred.

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 1D). For small network models there was an optimal number of threads, and using more threads was actually slower. As there is no communication between threads, the optimum number is presumably determined by the balance between the overhead in launching each thread and the time spent performing computations therein. There is also a dependence of loop times on spike rate, due to the second non-threaded loop, the slope of which is proportional to $K + L$ (Figure 1F).

Given that GPUs are purpose-built to thread well, we investigated whether the RLS algorithm would scale better with them. Since accessing individual elements in a vector is very slow on a GPU, due to each kernel call incurring a large overhead, we made a functionally equivalent copy of the CPU code and refactored it to use vector operations instead of `for`-loops over elements. Specifically, to reset the membrane potential in a performant way

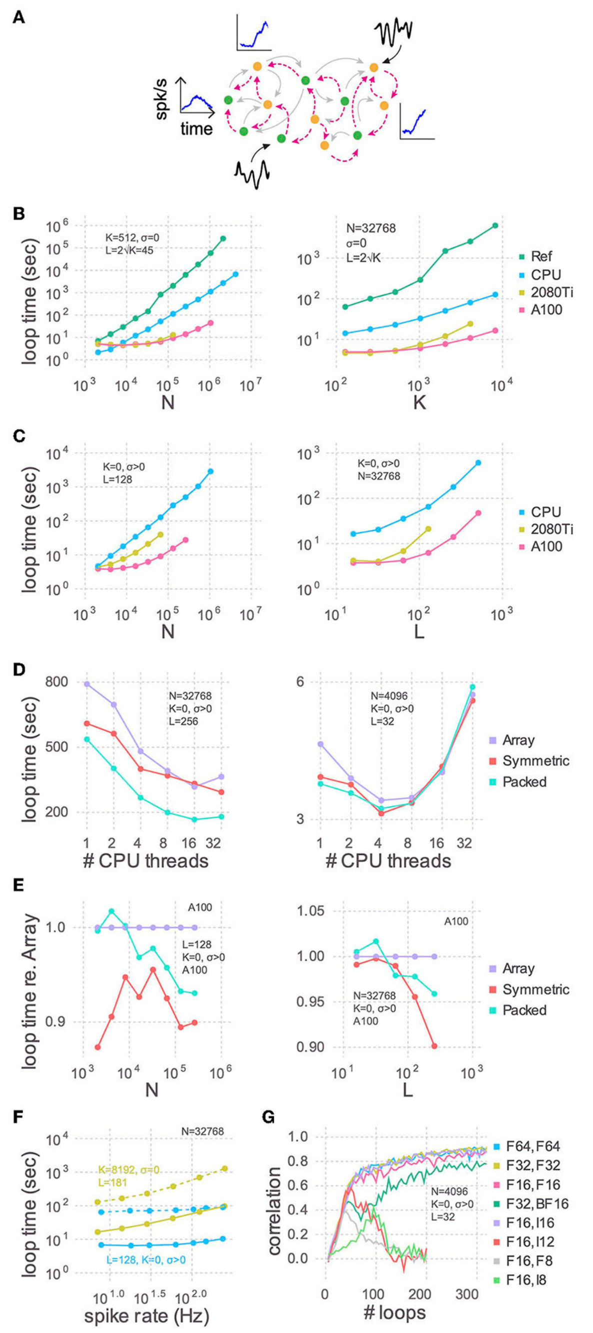

FIGURE 1
Training speed for models of various sizes. **(A)** Cartoon of a recurrent spiking neural network. Excitatory (green circles) and inhibitory (yellow circles) neurons have plastic (magenta lines) and, optionally, static (gray lines) connections to each other. Each neuron receives, on average, K static and L plastic connections. Associated with each neuron are the target 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)** The time taken by each training iteration for models with static connections vs. the number of neurons (N; left) or the number of static connections (K; right). Compared are the single-threaded CPU code used by Kim et al. (2023), 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:
(Continued)

FIGURE 1 (Continued)
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 $\boldsymbol{P}$. **(C)** Similar to **(B)** but with a Gaussian noise model instead of static connections ($K = 0, \sigma > 0$), when varying the number of neurons (left) or the number of plastic connections (L; right). **(D)** Strong scaling of the optimized CPU code for large (left) and small (right) models. Purple: code stores the $\boldsymbol{P}$ matrix in a full dense matrix. Red: symmetric matrix. Cyan: packed symmetric matrix. **(E)** The effect of the matrix storage format on the GPU code as a function of the number of neurons (left) or plastic synapses (right). **(F)** 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 Equation A7 in Supplementary material). Solid lines indicate GPU code and dotted are the CPU code. **(G)** The effect of storage precision on learning. The first column in the legend indicates the bit precision of all state variables *except* $\boldsymbol{P}$, which is indicated in the second column. For integer types, $\boldsymbol{P}$ was scaled by $2^{nbits-2}$.

when a spike occurs, we replaced the `for`-loops in the CPU code, and the `if`-statements therein, with a broadcasted `ifelse` statement that inputs a boolean vector indicating which neurons spiked. Doing so results in the membrane potential of all of the neurons which do not spike being "reset" to its current value, but this is still faster even for low spike rates. To update the post-synaptic neurons, we moved the `for`-loop to be inside a custom GPU kernel where CUDA atomic instructions are available to handle the race condition. Just one kernel call is hence made, thereby minimizing the overhead. A dependency of loop time on spike rate also exists, just like in the CPU code, due to the atomic locking.

Whereas, the CPU run times were linear with model size, GPU 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 3,456 threads.

2.1.2. Symmetric and packed arrays

The RLS algorithm uses the running estimate of the inverse of the correlation matrix of the activity of the pre-synaptic neurons (plus regularization terms), which for each of the N neurons is a symmetric matrix of size $L \times L$ that we denote as $\boldsymbol{P}$ (see Equation A18 in Supplementary material). To perform mathematical operations on $\boldsymbol{P}$, as well as other state variables, we utilized Basic Linear Algebra Subprograms (BLAS), a highly-engineered library of mathematical operations commonly used in high-performance computing. While some BLAS 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 (Figures 1D, E), particularly for models with large number of plastic synapses.

Further, $\boldsymbol{P}$ consumes by far more memory than any other variable since its footprint scales as $N \times L^2$. All other state variables are only one or two dimensional. 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 1E), BLAS routines specialized for packed symmetric matrices (`spmv` and `spr`) are much faster on the CPU (Figure 1D) for large models. We speculate that this performance difference is due to the sophisticated hierarchical caches on a CPU being better utilized with packed matrices, compared to a GPU.

2.1.3. BLAS, pre-allocated memory, and pre-computed division

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 symmetric matrix, $\boldsymbol{P}$, and the presynaptically filtered spike trains, $\boldsymbol{r}$ (see Equations A9 and A18 in Supplementary material). 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 (MKL: https://software.intel.com/en-us/intel-mkl) for the CPU, which is a superset of BLAS hand-crafted for the x86 architecture, instead of the default cross-platform OpenBLAS. Decrements in loop times were most pronounced for models with $K > 0$. Specifically, for a model with $N = 32,768, K = 1,638, L = 81$ we saw a 59% reduction using MKL, whereas for $N = 32,768, K = 0, L = 128$ there was only a 5.8% improvement, about 10-fold less.

BLAS functions frequently input multiplicative constants, forcing the user to manually do a division ahead of time if the constant is in the denominator. Following this lead for the sections of code that do not use BLAS directly, we precomputed, just once, the inverse of the synaptic currents and membrane voltage time constants as they are in the denominator of the equations that govern the neural dynamics. Loop times were about 2% quicker performing this multiplication, instead of the corresponding division, for the CPU code.

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`). Such batched GPU kernels are critical to our learning algorithm since we apply the RLS algorithm to each of the N trained neurons, and calling a non-batched kernel N times would incur a huge performance penalty due to the overhead of calling functions on GPUs. The solution was to write new BLAS kernels that internally iterate over N, which was necessary as NVIDIA only provides batched implementations of `gemm`, `gemv`, and `trsm`.

2.1.4. Reduced precision number formats

Our original reference code in Kim et al. (2023) 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 the correlations between target activity patterns and learned synaptic currents, which measure the accuracy of the training, are just as high and require 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 1G). Doing so not only permits models twice as large to be trained, but also yields quicker loop times on CPUs and consumer grade GPUs (data not shown).

Our custom batched BLAS kernels for the GPU can operate on floating point numbers of any precision, and even integers, unlike the CPU BLAS libraries. To further reduce memory consumption, and hence increase model size, we tried Float16, whose range is only 6.5×10^{4} and machine epsilon is 9.7×10^{-4}, and found that models can be trained almost as accurately as with 64 and 32 bits. To see if the small decrement in correlation could be recovered with a different partitioning of the bits between the exponent and the fraction, we tried BFloat16, which uses eight bits for the exponent, just like Float32, and hence has the same range, instead of Float16's five. Correlation coefficients for BFloat16 were worse. As there are no hardware-supported 16-bit floating point formats that allocate more bits to the fraction, we tried scaling $\boldsymbol{P}$ by 2^{14} and storing it in a 16-bit integer, effectively giving us 14 fractional bits. There is no overflow with this scheme, as $\boldsymbol{P}$ typically ranges from -1 to 2. Scaled 16-bit integers had correlations that were indistinguishable from Float32 and Float64. Furthermore, there is no speed penalty with 16-bit integers.

Modern hardware does not support 8-bit floats (though see the forthcoming NVIDIA H100 GPU), but they can be simulated in software with, for example, MicroFloatingPoint.jl. We tried 8-bit floats with two, three, four, or five exponential bits and found that correlations were best with four and reached a peak around half that of 32 and 64 bits and then declined. Scaling $\boldsymbol{P}$ by 2^{6} and storing it in an Int8 resulted in a correlation peak of similar height. While Int8 required more iterations to reach the peak than Float8, the wall clock time was actually less as the loop times were much shorter due to native hardware support.

There is also no direct hardware support for 12-bits, but given the large difference between our 8 and 16 bit results, we simulated them in software by using an Int16 [sic] scaled by 2^{10}. Since the magnitude of $\boldsymbol{P}$ is less than two, the four most significant bits here are entirely unused, and the 10 least significant are used as the fraction. As with scaled Int8, we observed a peak in the correlation coefficient, this time a bit larger at about two-thirds that of 32 and 64 bits, followed by decline. A 12-bit integer is not an unreasonable type to imagine using to reduce memory consumption, as two Int12 will pack into three bytes. In fact, UInt12Arrays.jl is a Julia package which does precisely this for unsigned 12-bit integers.

In summary, we recommend using Float32 unless the model does not fit in memory, in which case Int16 can be used. If it still does not fit and a drop in correlation can be tolerated, we recommend using Int8, or, time permitting, developing a proper Int12 package.

2.2. Application

With a fast RLS codebase in hand, we next demonstrated that large models can be successfully 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 (Jun et al., 2017) from multiple brain areas of mice performing a delayed-response task (Figure 2A; Guo et al., 2014; Chen et al., 2023). We first converted the PSTHs to the corresponding underlying synaptic currents by inverting the activation function of leaky integrate-and-fire neurons in the presence of noise (see Appendix: generating neural trajectories in Supplementary material). 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 in Supplementary material). As our goal was to show the scalability of the code and not to study the trained network, we did not use any prior knowledge on mesoscopic connectivity between or within brain regions, but simply initialized the plastic weights by randomly connecting neurons in an Erdos-Rényi graph. In addition, the trained plastic weights in our network were allowed to flip signs, and hence possibly violate Dale's law. However, in principle one could use mesoscopic connectivity to constrain the plastic weights and the RLS algorithm could be further developed to obey Dale's law (Kim and Chow, 2018).

We then used the RLS algorithm to train the neurons in the spiking network to follow the target functions. The correlation between the learned and target synaptic currents increased with the number of plastic inputs (L), and reached a plateau in half an hour (Figure 2B). 256 plastic synapses was the largest that would fit in the 80 GB of memory in an A100 GPU with $N = 66{,}002$. For comparison, a model of that size would take multiple days on a single thread of a CPU using the reference code of Kim et al. (2023). After training, we ran the spiking network multiple times, with the plastic weights kept frozen to the trained values (i.e., the weights are no longer changed with the RLS algorithm), and compared the learned synaptic currents (Figure 2D) and PSTHs (Figures 2C, E). The activity of neurons in the trained network showed a close correspondence with the activity patterns of the recorded neurons.

Note that in this experiment each trained neuron successfully learned two activity patterns corresponding to lick right and lick left trials. Learning more than two patterns would likely require additional plastic synapses, which could potentially limit the task complexity that can be achieved using a single GPU.

3. Discussion

We present optimized CPU and GPU implementations of the recursive least-square (RLS) algorithm for training spiking neural

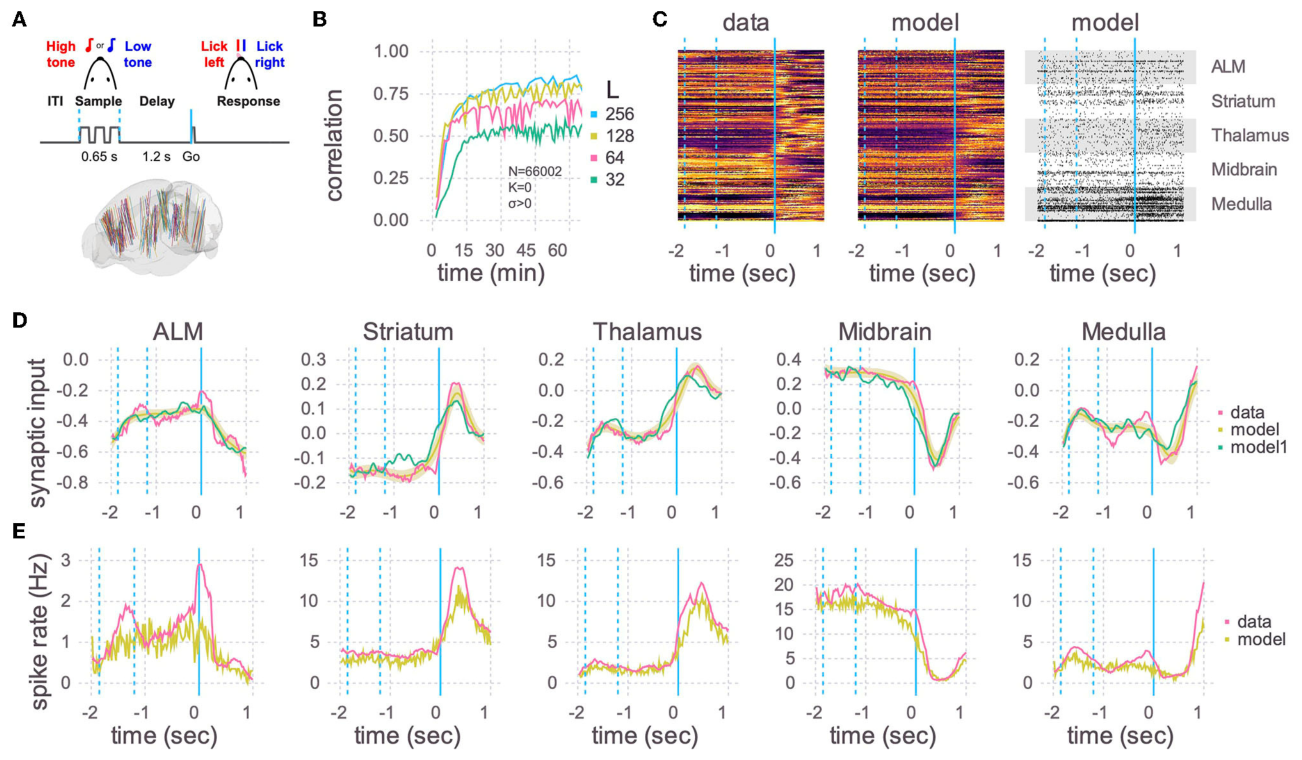

FIGURE 2
Application to Neuropixels data. **(A)** Top: Schematic of experimental setup. Mice learned to lick to one of two directions (left/right) after a delay period, depending on which of two tones were played. Bottom: Multiple Neuropixels probes, each with hundreds of recording sites, were placed in the brain while a mouse was performing the task (Chen et al., 2023). **(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 learning rate and final plateau level both increased with the number of plastic connections (L). The number of training iterations was 375, 330, 235, and 110, respectively, for L = 32, 64, 128, and 256. **(C)** Heat maps of the peri-stimulus time histograms (PSTHs) of the activity patterns in the data (left) and the trained network from B with $L = 256$ (middle) and one realization of spike trains of the same trained network (right). Fifty neurons are shown for each of five brain areas. Vertical blue lines are the same as in **(A)**, where a time of 0 s corresponds to the Go cue. **(D)** Learned synaptic currents averaged over 1,000 trials for one exemplar neuron from each of the five brain areas. Vertical blue lines are the same as in **(A)**, and the time axis is the same as in **(C)**. A single realization (non-averaged over trials) is also shown (model1). **(E)** Same as **(D)** but for PSTHs.

networks. Our code can simulate and train a spiking network consisting of about one million neurons and 100 million synapses on a single modern high-end workstation.

Our updated code is significantly faster than the previous version (Kim et al., 2023), as demonstrated by the green line in Figure 1B. Networks consisting of millions of neurons can be trained 1,000 times faster using the new code. We benchmarked the code for various scaling numbers (N, K, L) and also expanded its capabilities to include external noise (σ), allowing for training of both balanced (Van Vreeswijk and Sompolinsky, 1996; Darshan et al., 2018) and generic spiking neural networks. This increased efficiency means that large networks of a million neurons can now be trained in a matter of hours instead of weeks, thereby greatly speeding up the scientific discovery process and more efficiently using resources. Additionally, fast training times enables models to be created on the fly, allowing for *in-silico* training while *in-vivo* experiments are being conducted, thus closing the loop between modeling and experiments. For example, predictions could be made about perturbation experiments by fitting models to data and suggesting perturbation protocols in real-time.

While we used simple integrate-and-fire neurons, our code can be easily extended to include more realistic neurons using a cell-model plugin system that provides users the means to provide custom code. A simple integrate-and-fire neuron allowed us to easily convert PSTHs to synaptic currents using the known F-I curve (Equation A1 in Supplementary material; Roxin et al., 2011). It remains as future work to develop principled methods to convert the PSTHs to synaptic currents in more complex neuron models that include slow dynamic variables, such as adaptation currents or time-dependent spike thresholds (Teeter et al., 2018). However, the RLS algorithm used here allows one to train the synaptic currents of GLIF neurons, if the target synaptic currents are already available (see Kim and Chow, 2018 for training a network of Izhikevich neurons).

Additionally, although the plastic weights in our networks were random, our framework does not exclude adding complexity to the network architecture, such as including layered cortical networks or even different cell types, or using known mesoscopic connectivity when initializing the plastic weights in the network. This flexibility is again achieved through a set of plugins through which the user can provide custom code to define the adjacency matrices.

The RLS algorithm is designed to minimize the discrepancy between the PSTHs generated by the model neurons and the recorded neurons. It does not optimize for other spiking statistics, such as the coefficients of variation for interspike-intervals or Fano factors. These statistics can be influenced by adjusting the network's hyper-parameters, such as the level of noise applied to each neuron (σ). Tools are available to assess the network's performance against these other measures while varying the hyper-parameters (e.g., https://github.com/INM-6/NetworkUnit).

The size of the networks is limited by memory usage, which mainly depends on the size of the matrix $\boldsymbol{P}$ used in the RLS algorithm (see Equation A18 in Supplementary material). 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 would have to be could depend on the number and the complexity of the tasks to be learned.

Advances in GPU computing and strong interest in neuromorphic computing have led to various efficient implementations of spiking neural networks. Recent work that implements simulations of spiking neural networks in GPUs include the following: a code-generation based system that generates CUDA code for GPU (GeNN, Knight and Nowotny, 2018) and a popular Python-based simulator for spiking neural networks, Brian2, extended for generating CUDA code directly (Brian2CUDA, Alevi et al., 2022) or through GeNN (Brian2GeNN, Stimberg et al., 2020). Similarly, highly efficient CPU-based simulations of spiking neural networks can be implemented in NEST (NEural Simulation Tool, Gewaltig and Diesmann, 2007). Spiking neural networks can also be implemented in neuromorphic hardware as demonstrated in SpiNNaker (Furber et al., 2014), Intel's Loihi (Davies et al., 2018), and TrueNorth (Merolla et al., 2014). See Steffen et al. (2021) and references therein for benchmarks for these systems. We note that, although the aforementioned systems allow for biologically plausible plasticity (Stimberg et al., 2020) and for learning complex tasks (Merolla et al., 2014; Davies et al., 2018), the contribution of our work is different in that we developed an efficient algorithm for learning to generate activity patterns in recurrently connected spiking neural networks.

Finally, our code implementation was tailored to be used on a single computer, instead of on multiple computers, such as in Jordan et al. (2018). 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, the size of GPU memory does not increase at a rate comparable with advances in neural recording technology, we plan to investigate how much performance is decremented if our code is refactored to use multiple GPUs within the same workstation.

Data availability statement

Publicly available datasets were analyzed in this study. Our code can be found here: https://github.com/SpikingNetwork/TrainSpikingNet.jl; https://github.com/JaneliaSciComp/BatchedBLAS.jl; https://github.com/JaneliaSciComp/SymmetricFormats.jl.

Author contributions

CK and RD conceived of the algorithm. CK provided and helped understand the reference implementation. BA optimized the code for performance, with the assistance of CK and RD and prepared the figures. SC did the experiments and collected and analyzed the Neuropixels data. BA, CK, SP, and RD wrote the manuscript. All authors contributed to the article and approved the submitted version.

Funding

This work was supported by the Howard Hughes Medical Institute, the Visiting Scientist Program at Janelia Research Campus, and the Intramural Research Program of National Institute of Diabetes and Digestive and Kidney Diseases at the National Institutes of Health.

Acknowledgments

We would like to thank Shaul Druckmann, Nuo Li, Yi Liu, and Karel Svoboda for sharing the Neuropixels data set.

Conflict of interest

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

Publisher's note

Supplementary material

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fninf.2023.1099510/full#supplementary-material

References

Alevi, D., Stimberg, M., Sprekeler, H., Obermayer, K., and Augustin, M. (2022). Brian2CUDA: flexible and efficient simulation of spiking neural network models on GPUs. *Front. Neuroinform.* 16, 883700. doi: 10.3389/fninf.2022.883700

Amsalem, O., Inagaki, H., Yu, J., Svoboda, K., and Darshan, R. (2022). Subthreshold neuronal activity and the dynamical regime of cerebral cortex. *bioRxiv*. doi: 10.1101/2022.07.14.500004

Andalman, A. S., Burns, V. M., Lovett-Barron, M., Broxton, M., Poole, B., Yang, S. J., et al. (2019). Neuronal dynamics regulating brain and behavioral state transitions. *Cell* 177, 970–985. doi: 10.1016/j.cell.2019.02.037

Bezanson, J., Edelman, A., Karpinski, S., and Shah, V. B. (2017). Julia: a fresh approach to numerical computing. *SIAM Rev.* 59, 65–98. doi: 10.1137/141000671

Brunel, N. (2000). Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons. *J. Comput. Neurosci.* 8, 183–208. doi: 10.1023/A:1008925309027

Chen, S., Liu, Y., Wang, Z., Colonell, J., Liu, L. D., Hou, H., et al. (2023). Brain-wide neural activity underlying memory-guided movement. *bioRxiv*. doi: 10.1101/2023.03.01.530520

Daie, K., Svoboda, K., and Druckmann, S. (2021). Targeted photostimulation uncovers circuit motifs supporting short-term memory. *Nat. Neurosci.* 24, 259–265. doi: 10.1038/s41593-020-00776-3

Darshan, R., Van Vreeswijk, C., and Hansel, D. (2018). Strength of correlations in strongly recurrent neuronal networks. *Phys. Rev. X* 8, 031072. doi: 10.1103/PhysRevX.8.031072

Davies, M., Srinivasa, N., Lin, T.-H., Chinya, G., Cao, Y., Choday, S. H., et al. (2018). Loihi: a neuromorphic manycore processor with on-chip learning. *IEEE Micro* 38, 82–99. doi: 10.1109/MM.2018.112130359

DePasquale, B., Cueva, C. J., Rajan, K., Escola, G. S., and Abbott, L. (2018). full-force: a target-based method for training recurrent networks. *PLoS ONE* 13, e0191527. doi: 10.1371/journal.pone.0191527

Finkelstein, A., Fontolan, L., Economo, M. N., Li, N., Romani, S., and Svoboda, K. (2021). Attractor dynamics gate cortical information flow during decision-making. *Nat. Neurosci.* 24, 843–850. doi: 10.1038/s41593-021-00840-6

Fisher, D., Olasagasti, I., Tank, D. W., Aksay, E. R., and Goldman, M. S. (2013). A modeling framework for deriving the structural and functional architecture of a short-term memory microcircuit. *Neuron* 79, 987–1000. doi: 10.1016/j.neuron.2013.06.041

Furber, S. B., Galluppi, F., Temple, S., and Plana, L. A. (2014). The spinnaker project. *Proc. IEEE* 102, 652–665. doi: 10.1109/JPROC.2014.2304638

Gewaltig, M.-O. and Diesmann, M. (2007). Nest (neural simulation tool). *Scholarpedia* 2, 1430. doi: 10.4249/scholarpedia.1430

Guo, Z. V., Li, N., Huber, D., Ophir, E., Gutnisky, D., Ting, J. T., et al. (2014). Flow of cortical activity underlying a tactile decision in mice. *Neuron* 81, 179–194. doi: 10.1016/j.neuron.2013.10.020

Haykin, S. (1996). *Adaptive Filter Theory, 3rd Edn.*. Upper Saddle River, NJ: Prentice-Hall, Inc.

Hofer, S. B., Ko, H., Pichler, B., Vogelstein, J., Ros, H., Zeng, H., et al. (2011). Differential connectivity and response dynamics of excitatory and inhibitory neurons in visual cortex. *Nat. Neurosci.* 14, 1045–1052. doi: 10.1038/nn.2876

Inagaki, H. K., Chen, S., Ridder, M. C., Sah, P., Li, N., Yang, Z., et al. (2022). A midbrain-thalamus-cortex circuit reorganizes cortical dynamics to initiate movement. *Cell* 185, 1065–1081. doi: 10.1016/j.cell.2022.02.006

Jordan, J., Ippen, T., Helias, M., Kitayama, I., Sato, M., Igarashi, J., et al. (2018). Extremely scalable spiking neuronal network simulation code: from laptops to exascale computers. *Front. Neuroinform.* 12, 2. doi: 10.3389/fninf.2018.00002

Jun, J. J., Steinmetz, N. A., Siegle, J. H., Denman, D. J., Bauza, M., Barbarits, B., et al. (2017). Fully integrated silicon probes for high-density recording of neural activity. *Nature* 551, 232–236. doi: 10.1038/nature24636

Kim, C. M., and Chow, C. C. (2018). Learning recurrent dynamics in spiking networks. *eLife* 7, e37124. doi: 10.7554/eLife.37124

Kim, C. M., and Chow, C. C. (2021). Training spiking neural networks in the strong coupling regime. *Neural Comput.* 33, 1199–1233. doi: 10.1162/neco_a_01379

Kim, C. M., Finkelstein, A., Chow, C. C., Svoboda, K., and Darshan, R. (2023). Distributing task-related neural activity across a cortical network through task-independent connections. *Nature Communications.* 14:2851. doi: 10.1101/2022.06.17.496618

Knight, J. C., and Nowotny, T. (2018). GPUs outperform current HPC and neuromorphic solutions in terms of speed and energy when simulating a highly-connected cortical model. *Front. Neurosci.* 12, 941. doi: 10.3389/fnins.2018.00941

Laje, R., and Buonomano, D. V. (2013). Robust timing and motor patterns by taming chaos in recurrent neural networks. *Nat. Neurosci.* 16, 925–933. doi: 10.1038/nn.3405

Merolla, P. A., Arthur, J. V., Alvarez-Icaza, R., Cassidy, A. S., Sawada, J., Akopyan, F., et al. (2014). A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science* 345, 668–673. doi: 10.1126/science.1254642

Nicola, W., and Clopath, C. (2017). Supervised learning in spiking neural networks with force training. *Nat. Commun.* 8, 2208. doi: 10.1038/s41467-017-01827-3

Rajan, K., Harvey, C. D., and Tank, D. W. (2016). Recurrent network models of sequence generation and memory. *Neuron* 90, 128–142. doi: 10.1016/j.neuron.2016.02.009

Roxin, A., Brunel, N., Hansel, D., Mongillo, G., and van Vreeswijk, C. (2011). On the distribution of firing rates in networks of cortical neurons. *J. Neurosci.* 31, 16217–16226. doi: 10.1523/JNEUROSCI.1677-11.2011

Steffen, L., Koch, R., Ulbrich, S., Nitzsche, S., Roennau, A., and Dillmann, R. (2021). Benchmarking highly parallel hardware for spiking neural networks in robotics. *Front. Neurosci.* 15, 667011. doi: 10.3389/fnins.2021.667011

Stevenson, I. H., and Kording, K. P. (2011). How advances in neural recording affect data analysis. *Nat. Neurosci.* 14, 139–142. doi: 10.1038/nn.2731

Stimberg, M., Goodman, D. F., and Nowotny, T. (2020). Brian2GENN: accelerating spiking neural network simulations with graphics hardware. *Sci. Rep.* 10, 410. doi: 10.1038/s41598-019-54957-7

Sussillo, D., and Abbott, L. (2009). Generating coherent patterns of activity from chaotic neural networks. *Neuron* 63, 544–557. doi: 10.1016/j.neuron.2009.07.018

Teeter, C., Iyer, R., Menon, V., Gouwens, N., Feng, D., Berg, J., et al. (2018). Generalized leaky integrate-and-fire models classify multiple neuron types. *Nat. Commun.* 9, 709. doi: 10.1038/s41467-017-02717-4

Urai, A. E., Doiron, B., Leifer, A. M., and Churchland, A. K. (2022). Large-scale neural recordings call for new insights to link brain and behavior. *Nat. Neurosci.* 25, 11–19. doi: 10.1038/s41593-021-00980-9

Van Vreeswijk, C., and Sompolinsky, H. (1996). Chaos in neuronal networks with balanced excitatory and inhibitory activity. *Science* 274, 1724–1726. doi: 10.1126/science.274.5293.1724