nature machine intelligence

Article

https://doi.org/10.1038/s42256-023-00668-8

A neural machine code and programming framework for the reservoir computer

In the format provided by the authors and unedited

Supplement to "A Neural Machine Code and Programming Framework for the Reservoir Computer"

Jason Z. Kim

Department of Bioengineering, University of Pennsylvania, Philadelphia, PA, 19104 and

Department of Physics, Cornell University, Ithaca, NY 14850

Dani S. Bassett

Departments of Bioengineering, Physics & Astronomy,

Electrical & Systems Engineering, Neurology, and Psychiatry,

University of Pennsylvania, Philadelphia, PA, 19104

Santa Fe Institute, Santa Fe, NM 87501 and

To whom correspondence should be addressed: dsb@seas.upenn.edu

(Dated: April 11, 2023)

I. DECOMPILING INTO THE STATE NP IN CONTINUOUS-TIME

In the main text, we provide the outline of a derivation for the neural programming framework. Here, we expand upon this derivation in more detail for continuous-time RNNs. As in the main text, we begin with the reservoir equation

$$\frac{1}{\gamma}\dot{\boldsymbol{r}}(t) = -\boldsymbol{r}(t) + \boldsymbol{g}(A\boldsymbol{r}(t) + B\boldsymbol{x}(t) + \boldsymbol{d}). \tag{1}$$

As in the main text, we will omit the explicit denotation of the time variable using (t) for conciseness and clarity. Our goal is to write $\boldsymbol{r}$ as an explicit and analytic function of $\boldsymbol{x}$, which is difficult to do for nonlinear systems. Hence, the first step is to linearize this equation about a static operating point $\boldsymbol{r}^*$, which yields

$$\frac{1}{\gamma}\dot{\boldsymbol{r}} \approx -\boldsymbol{r} + \boldsymbol{g}(A\boldsymbol{r}^* + B\boldsymbol{x} + \boldsymbol{d}) + \mathrm{d}\boldsymbol{g}(A\boldsymbol{r}^* + B\boldsymbol{x} + \boldsymbol{d}) \circ (A\boldsymbol{r} - A\boldsymbol{r}^*),$$

where $\circ$ is the element-wise product operation. Rearranging this equation to group the terms with the $\boldsymbol{r}$ variable yields

$$\frac{1}{\gamma}\dot{\boldsymbol{r}} \approx -\boldsymbol{r} + \underbrace{\mathrm{d}\boldsymbol{g}(A\boldsymbol{r}^* + B\boldsymbol{x} + \boldsymbol{d}) \circ (A\boldsymbol{r})}_{\text{bilinear}} + \boldsymbol{g}(A\boldsymbol{r}^* + B\boldsymbol{x} + \boldsymbol{d}) - \mathrm{d}\boldsymbol{g}(A\boldsymbol{r}^* + B\boldsymbol{x} + \boldsymbol{d}) \circ (A\boldsymbol{r}^*).$$

However, we notice a *bilinear* term in this expansion that contains both $\boldsymbol{x}$ and $\boldsymbol{r}$. Hence, the second step in this approximation is to linearize the bilinear term by evaluating $\boldsymbol{x}$ at an operating point $\boldsymbol{x}^*$ to yield

$$\frac{1}{\gamma}\dot{\boldsymbol{r}} \approx \underbrace{(\mathrm{d}\boldsymbol{g}(A\boldsymbol{r}^* + B\boldsymbol{x}^* + \boldsymbol{d}) \circ A - I)}_{A^*}\boldsymbol{r} + \boldsymbol{g}(A\boldsymbol{r}^* + B\boldsymbol{x} + \boldsymbol{d}) - \mathrm{d}\boldsymbol{g}(A\boldsymbol{r}^* + B\boldsymbol{x} + \boldsymbol{d}) \circ (A\boldsymbol{r}^*).$$

Rewriting this equation using the additional substitution of $\boldsymbol{d}^* = A\boldsymbol{r}^* + \boldsymbol{d}$ yields

$$\frac{1}{\gamma}\dot{\boldsymbol{r}} \approx A^*\boldsymbol{r} + \underbrace{\boldsymbol{g}(B\boldsymbol{x} + \boldsymbol{d}^*) - \mathrm{d}\boldsymbol{g}(B\boldsymbol{x} + \boldsymbol{d}^*) \circ (A\boldsymbol{r}^*)}_{\boldsymbol{u}(\boldsymbol{x}(t))}. \tag{2}$$

The term in the underbrace—$\boldsymbol{u}(\boldsymbol{x}(t))$—is precisely the effective input in the main text. This equation is now a linear system of equations whose states we can write as the convolution

$$\boldsymbol{r}(t) \approx \gamma \int_{-\infty}^{t} e^{\gamma A^*(t-\tau)}\boldsymbol{u}(\boldsymbol{x}(\tau))\mathrm{d}\tau.$$

Substituting the expression for the effective input yields

$$\boldsymbol{r}(t) \approx \gamma \int_{-\infty}^{t} e^{\gamma A^*(t-\tau)}\boldsymbol{g}(B\boldsymbol{x}(\tau) + \boldsymbol{d}^*)\mathrm{d}\tau - \gamma \int_{-\infty}^{t} e^{\gamma A^*(t-\tau)}\mathrm{d}\boldsymbol{g}(B\boldsymbol{x}(\tau) + \boldsymbol{d}^*) \circ (A\boldsymbol{r}^*)\mathrm{d}\tau. \tag{3}$$

While this convolution is technically an analytic function of $\boldsymbol{x}$, we would like a simpler algebraic expression that does not involve integrals. Hence, we evaluate the integrals by performing a series expansion of $\boldsymbol{g}$ and $\mathrm{d}\boldsymbol{g}$ with respect to $\tau = t$ to yield

$$\begin{aligned}\boldsymbol{g}(B\boldsymbol{x}(\tau) + \boldsymbol{d}^*) &\approx \boldsymbol{g}(B\boldsymbol{x}(t) + \boldsymbol{d}^*) - (t-\tau)\mathrm{d}\boldsymbol{g}(B\boldsymbol{x}(t) + \boldsymbol{d}^*) \circ (B\dot{\boldsymbol{x}}),\\ \mathrm{d}\boldsymbol{g}(B\boldsymbol{x}(\tau) + \boldsymbol{d}^*) &\approx \mathrm{d}\boldsymbol{g}(B\boldsymbol{x}(t) + \boldsymbol{d}^*) - (t-\tau)\mathrm{d}^2\boldsymbol{g}(B\boldsymbol{x}(t) + \boldsymbol{d}^*) \circ (B\dot{\boldsymbol{x}}).\end{aligned} \tag{4}$$

Using the following integration identities:

$$\begin{aligned}\int_{-\infty}^{t} e^{A(t-\tau)}\mathrm{d}\tau &= -A^{-1}\\ \int_{-\infty}^{t} e^{A(t-\tau)}(t-\tau)\mathrm{d}\tau &= A^{-2},\end{aligned}$$

we can evaluate the integrals to arrive at an algebraic expression

$$\begin{aligned}\boldsymbol{r}(t) \approx & A^{*-1}(\mathrm{d}\boldsymbol{g}(B\boldsymbol{x} + \boldsymbol{d}^*) \circ (A\boldsymbol{r}^*) - \boldsymbol{g}(B\boldsymbol{x} + \boldsymbol{d}^*)) + \\ & \frac{1}{\gamma}A^{*-2}\left((\mathrm{d}^2\boldsymbol{g}(B\boldsymbol{x} + \boldsymbol{d}^*) \circ (A\boldsymbol{r}^*) - \mathrm{d}\boldsymbol{g}(B\boldsymbol{x} + \boldsymbol{d}^*)) \circ (B\dot{\boldsymbol{x}})\right).\end{aligned} \tag{5}$$

The right-hand side of this equation is precisely the analytic function $\boldsymbol{h}$ in the main text. Finally, by taking the Taylor series expansion of Eq. 5 with respect to $\boldsymbol{x}$, we obtain an algebraic expression of the reservoir neuron states, $\boldsymbol{r}(t)$, as a weighted sum of polynomials in $\boldsymbol{x}$ and $\dot{\boldsymbol{x}}$. By taking additional, higher-order terms in the expansion of Eq. 4, we resolve additional time derivatives $\ddot{\boldsymbol{x}}$, $\dddot{\boldsymbol{x}}, \cdots$ in the expansion, yielding the analytic function $\boldsymbol{h}(\boldsymbol{x}, \dot{\boldsymbol{x}}, \ddot{\boldsymbol{x}}, \cdots)$ in the main text.

Importantly, the above integration identities only hold true for A^* when the real components of the eigenvalues of A^* are strictly negative. For $g = \tanh$, we know that its derivative $1 - \tanh^2$ is bounded between 0 and 1, such that the elements of $\boldsymbol{z} = \mathrm{d}\boldsymbol{g}(A\boldsymbol{r}^* + B\boldsymbol{x}^* + \boldsymbol{d})$ are also bounded between 0 and 1. Because $A^* = \boldsymbol{z} \circ A - I$, if the spectral radius of A lies within the unit circle, then the spectral radius of $\boldsymbol{z} \circ A$ also lies within the unit circle, and the real components of $A^* = \boldsymbol{z} \circ A - I$ are guaranteed to be negative.

II. DECOMPILING INTO THE STATE NP IN DISCRETE-TIME

Here, we expand upon the derivation of the reservoir state in more detail for discrete-time RNNs. As in the main text, we begin with the reservoir equation

$$\boldsymbol{r}_{t+1} = \boldsymbol{g}(A\boldsymbol{r}_t + B\boldsymbol{x}_t + \boldsymbol{d}). \tag{6}$$

Our goal is to write $\boldsymbol{r}_{t+1}$ as an explicit and algebraic function of $\boldsymbol{x}_t$, which is difficult to do for nonlinear systems. Hence, the first step is to linearize the equation about a static operating point $\boldsymbol{r}^*$, which yields

$$\boldsymbol{r}_{t+1} \approx \boldsymbol{g}(A\boldsymbol{r}^* + B\boldsymbol{x}_t + \boldsymbol{d}) + \mathrm{d}\boldsymbol{g}(A\boldsymbol{r}^* + B\boldsymbol{x}_t + \boldsymbol{d}) \circ (A\boldsymbol{r}_t - A\boldsymbol{r}^*).$$

Rearranging this equation to group the terms with the $\boldsymbol{r}_t$ variables yields

$$\boldsymbol{r}_{t+1} \approx \mathrm{d}\boldsymbol{g}(A\boldsymbol{r}^* + B\boldsymbol{x}_t + \boldsymbol{d}) \circ (A\boldsymbol{r}_t) + \boldsymbol{g}(A\boldsymbol{r}^* + B\boldsymbol{x}_t + \boldsymbol{d}) - \mathrm{d}\boldsymbol{g}(A\boldsymbol{r}^* + B\boldsymbol{x}_t + \boldsymbol{d}) \circ (A\boldsymbol{r}^*).$$

As in the continuous-time case, we approximate the bilinear term by evaluating $\boldsymbol{x}$ at an operating point $\boldsymbol{x}^*$ to yield

$$\boldsymbol{r}_{t+1} \approx \underbrace{\mathrm{d}\boldsymbol{g}(A\boldsymbol{r}^* + B\boldsymbol{x}^* + \boldsymbol{d}^*) \circ A}_{A^*}\boldsymbol{r}_t + \boldsymbol{g}(A\boldsymbol{r}^* + B\boldsymbol{x}_t + \boldsymbol{d}) - \mathrm{d}\boldsymbol{g}(A\boldsymbol{r}^* + B\boldsymbol{x}_t + \boldsymbol{d}) \circ (A\boldsymbol{r}^*).$$

Rewriting this equation using the additional substitution of $\boldsymbol{d}^* = A\boldsymbol{r}^* + \boldsymbol{d}$ yields

$$\boldsymbol{r}_{t+1} \approx A^*\boldsymbol{r}_t + \underbrace{\boldsymbol{g}(B\boldsymbol{x}_t + \boldsymbol{d}) - \mathrm{d}\boldsymbol{g}(B\boldsymbol{x}_t + \boldsymbol{d}) \circ (A\boldsymbol{r}^*)}_{\boldsymbol{u}(\boldsymbol{x}_t)}.$$

The term $\boldsymbol{u}(\boldsymbol{x}_t)$ is precisely the effective input in the main text. Because this system is now linear, we may write the state evolution of the RNN as

$$\boldsymbol{r}_{t+1} = \sum_{n=-\infty}^{t} A^{*(t-n)}\boldsymbol{u}(\boldsymbol{x}_n). \tag{7}$$

The right-hand side of this equation is precisely the analytic function $\boldsymbol{h}$ in the main text. Finally, by taking the Taylor series expansion of Eq.7 with respect to the variables $\boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \cdots$, we obtain an algebraic expression of the reservoir neuron states $\boldsymbol{r}_{t+1}$ as a weighted sum of polynomials in $\boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \cdots$. As an important detail, to enforce $\boldsymbol{r}^*$ as a fixed point, we solved for the bias term $\boldsymbol{d}$ that satisfies the equation

$$\boldsymbol{r}^* = \tanh(A\boldsymbol{r}^* + B\boldsymbol{x}^* + \boldsymbol{d}),$$

which produces

$$\boldsymbol{d} = \tanh^{-1}(\boldsymbol{r}^*) - A\boldsymbol{r}^* - B\boldsymbol{x}^*.$$

We note that the constant term of the programming matrix (i.e., $\boldsymbol{h}^*$) is equal to $\boldsymbol{r}^*$.

III. A STEP-BY-STEP EXAMPLE OF THE PROGRAMMING PROCEDURE

In the main text, we provided a detailed explanation for all of the components of the programming procedure. Here, we provide additional detail following Fig.1 in a step-by-step guide for the rotation example in Fig.1d-f.

First, we require a machine, which is the continuous-time RNN. In this example, the machine has 3 inputs, $\boldsymbol{x} = [x_1;\ x_2;\ x_3]$, 3 outputs, $\boldsymbol{o} = [o_1;\ o_2;\ o_3]$, and 1000 neurons $\boldsymbol{r} = [r_1;\ r_2;\ \cdots;\ r_{1000}]$. Next, we have to pick our instruction set, which in this example is the continuous-time dynamics given by

$$\frac{1}{\gamma}\dot{\boldsymbol{r}} = -\boldsymbol{r} + \tanh(A\boldsymbol{r} + B\boldsymbol{x} + \boldsymbol{d}).$$

Then, we have to choose our machine code for the RNN, which are the 1000×3 matrix B, the 1000×1000 matrix A, and the 1000×1 vector $\boldsymbol{d}$. In this example, the recurrent connections A are chosen randomly to be 5% dense and scaled to have a spectral radius of $\rho = 0.01$. Up till now, we have completed steps Fig.1a, Fig.1b, and Fig.1c. Given the choice of $\boldsymbol{d}$, we have a different fixed point $\boldsymbol{r}^*$. We choose $\gamma = 100$, and we choose our operating point to be $\boldsymbol{x}^* = [0; 0; 0]$.

Now, we wish to construct our programming matrix, given by R_c, shown in Fig.1d. To construct R_c, we simply substitute A, B, $\boldsymbol{d}$, and $\boldsymbol{r}^*$ into Eq. 5, while keeping $\boldsymbol{x}, \dot{\boldsymbol{x}}, \cdots$ as variables. This substitution will yield a 1000×1 algebraic function $\boldsymbol{h}$ with respect to the variables $\boldsymbol{x}, \dot{\boldsymbol{x}}, \cdots$. To construct the programming matrix R_c, we take the Taylor series expansion of $\boldsymbol{h}$ with respect to the variables $\boldsymbol{x}, \dot{\boldsymbol{x}}, \cdots$, and the matrix R_c will be comprised of the coefficients of this expansion. More specifically, the term in the (i, j)-th entry of R_c will be the Taylor series coefficient of the algebraic function h_i with respect to the j-th expansion term. The very first column of R_c is given by the evaluation of Eq. 5 at the equilibrium point $\boldsymbol{x}^* = \dot{\boldsymbol{x}}^* = \cdots = \boldsymbol{0}$, yielding

$$\boldsymbol{h}^* \approx A^{*-1}(\mathrm{d}\boldsymbol{g}(\boldsymbol{d}^*) \circ (A\boldsymbol{r}^*) - \boldsymbol{g}(\boldsymbol{d}^*)),$$

the first column of R_c. The second column is given by the partial derivative of $\boldsymbol{h}$ with respect to x_1, evaluated at the equilibrium point, given by $\frac{\partial \boldsymbol{h}}{\partial x_1}\big|_{\boldsymbol{x}^*=\dot{\boldsymbol{x}}^*=\cdots=\boldsymbol{0}}$. This process continues until the desired number of expansion terms is reached. In our specific example, we took 40 expansion terms. The first 10 correspond to the 1 constant term, $\boldsymbol{h}^* = \boldsymbol{h}\big|_{\boldsymbol{x}^*=\dot{\boldsymbol{x}}^*=\cdots=\boldsymbol{0}}$, the 3

linear terms $\frac{\partial \boldsymbol{h}}{\partial x_i}\big|_{\boldsymbol{x}^*=\dot{\boldsymbol{x}}^*=\cdots=\boldsymbol{0}}$, and the 6 cross terms $\frac{\partial^2 \boldsymbol{h}}{\partial x_i \partial x_j}\big|_{\boldsymbol{x}^*=\dot{\boldsymbol{x}}^*=\cdots=\boldsymbol{0}}$. The next 10 correspond to the first 10, except with an additional partial derivative with respect to $\dot{x}_1$. The next 10 correspond to the first 10, except with an additional partial derivative with respect to $\dot{x}_2$. The final 10 correspond to the first 10, except with an additional partial derivative with respect to $\dot{x}_3$. Hence, in this specific example, the full R_c matrix is a 1000×40 matrix of coefficients of subsequent Taylor series expansions. R_c is the *programming matrix* for this example.

Now that we have our programming matrix, we wish to write some code. To do this, we first write (in variables) the output function that we want. In this particular example, we wish to rotate the input by 90 degrees about the x_3 axis. We can write this operation in variables as $o_1 = -x_2$, $o_2 = x_1$, and $o_3 = x_3$. In matrix form, this operation looks like

$$\begin{bmatrix} o_1 \\ o_2 \\ o_3 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}.$$

As can be seen, all of the outputs can be written as a weighted sum of the input variables. More specifically, the outputs can be written as a linear combination of the 40 Taylor series terms in our expansion. Hence, we convert this algebraic rotation operation into our source code, which is a matrix O_c that is a 3×40 matrix. Recall that the columns of R_c correspond to the coefficients belonging to a specific term. For example, the first column of R_c corresponds to the constant term. The second column of R_c corresponds to the coefficients that multiply the x_1 term. The third column of R_c corresponds to the coefficients that multiply the x_2 term. The fourth column of R_c corresponds to the coefficients that multiply the x_3 term.

Hence, the first four columns of the expansion already possess enough terms to encode our rotation. As such, we construct O_c by putting zeros into the first column, because our output function has no constant terms. Next, we put a 1 in the second row, second column of O_c, because the second output (corresponding to the second row) is defined as 1 times x_1, and the second column of O_c corresponds to the coefficients that multiply the x_1 term. Next, we put a -1 in the first row, third column of O_c, because the first output (corresponding to the first row) is defined as -1 times x_2, and the third column of O_c corresponds to the coefficients that multiply the x_2 term. Next, we put a 1 in the third row, fourth column of O_c, because the third output (corresponding to the third row) is defined as 1 times x_3, and

the fourth column of O_c corresponds to the coefficients that multiply the x_3 term. Now that we have accounted for all of the terms in the output function, we fill the remaining entries of O_c with zeros.

Now that we have our 1000×40 programming matrix R_c, and the 3×40 source-code matrix O_c, we wish to compile this code into the machine code of the RNN: W. To compile our program, we solve the least-squares problem $\min \|WR_c - O_c\|$, which yields a 3×1000 matrix W that maps R_c to O_c. This matrix W has thus been compiled to convert the neural representation of the inputs, $\boldsymbol{r}(\boldsymbol{x}(t))$, to the desired output, $\boldsymbol{o}(t)$, through the matrix transformation $\boldsymbol{o}(t) \approx W\boldsymbol{r}(\boldsymbol{x}(t))$. This means that when we drive the RNN with some input signal $\boldsymbol{x}(t)$, the output is a rotated version of that input.

IV. UPPER BOUND ON THE GOODNESS OF THE LINEARIZATION

Given the number of steps that we took in deriving the reservoir state as an algebraic function of inputs $\boldsymbol{x}$ and their time derivatives, it is important to quantify the goodness of the approximation. The first and most fundamental step in the approximation process was the linearization of the system to obtain Eq. 2. Hence, we will first compare the trajectory $\boldsymbol{r}(t)$ of the full nonlinear equation using Eq. 1 to the trajectory $\boldsymbol{r}'(t)$ of the linearized equation using Eq. 2. To systematically study the approximation, we compute the relative error given by the norm of $\boldsymbol{r}(t) - \boldsymbol{r}'(t)$ divided by the norm of $\boldsymbol{r}(t) - \boldsymbol{r}^*$, where $\boldsymbol{r}^*$ is the operating point of the system.

We vary two important parameters that determine the goodness of the approximation: the magnitude of A and the magnitude of B. Specifically, we draw the entries of A and B from uniform, independent, and random distributions ranging from -1 to 1, and then linearly scale A by a scalar a, and B by a scalar b. We fix the RC to have $n = 1000$ neurons, and we require A to be 5% dense and B to be fully dense. We also fix $\gamma = 100$, and we standardize the operating point to $\boldsymbol{r}^* = \boldsymbol{0}$.

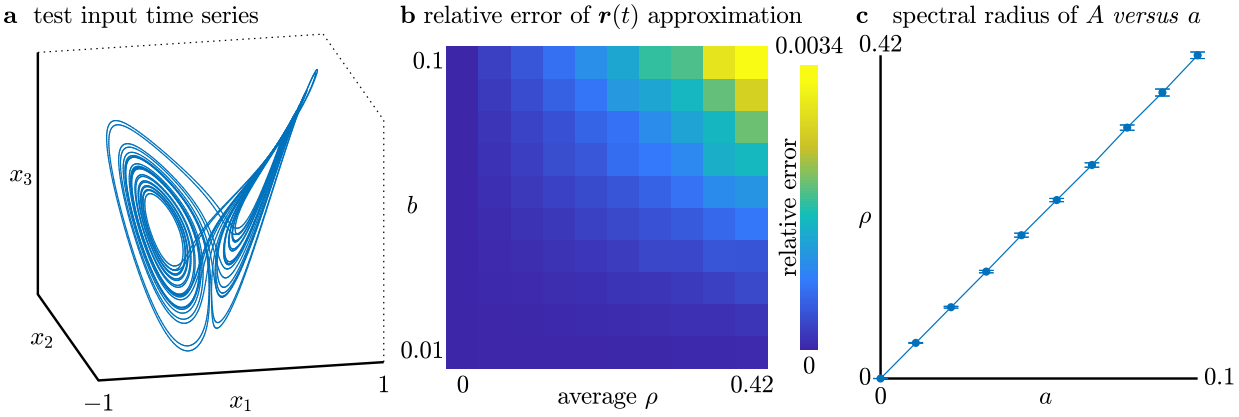

FIG. 1. **Testing the goodness of the linearization.** (**a**) Chaotic Lorenz attractor which is the test input into our RNN. All axes range from -1 to 1. (**b**) Relative error of the reservoir states between those generated using the full nonlinear expression, $\boldsymbol{r}(t)$ (Eq. 1), and those generated using the linearized expression, $\boldsymbol{r}'(t)$ (Eq. 2). We notice the errors are less than 1%. (**c**) Plot of the means and standard deviations of the spectral radius of the adjacency matrix A given by the magnitude of its largest eigenvalue ρ. Each point is the average ρ across the 10 different instantiations of A for each value of a.

As our test input, we will use a time series generated from the 3-dimensional Lorenz attractor that has been shifted to be centered around $\boldsymbol{x} \approx \mathbf{0}$ and scaled such that its states evolve roughly between $-1 \leq x_1, x_2, x_3 \leq 1$ using the equations

$$\begin{aligned} \dot{x}_1 &= \sigma(x_2 - x_1) \\ \dot{x}_2 &= x_1(\rho - (20x_3 + 27)) - x_2 \\ \dot{x}_3 &= 20x_1x_2 - \beta\left(x_3 + \frac{27}{20}\right), \end{aligned} \tag{8}$$

where $\sigma = 10$, $\rho = 28$, and $\beta = 8/3$. The trajectory used for training is shown in Fig. 1a. As can be seen, the relative error increases with larger scaling values of a and b, but the error remains solidly below 1% for this wide range of parameters (Fig. 1b).

To contextualize these scaling values, we first note that any dependence of the approximation on b can be addressed simply by rescaling the input. For example, if we were to rescale the input time series in Fig. 1a to one tenth of its current magnitude, then the relative error in Fig. 1b at $b = 0.1$ would significantly decrease to the relative error at $b = 0.01$. This is because mathematically, B and $\boldsymbol{x}$ are multiplied, such that rescaling the input can always provide smaller relative error.

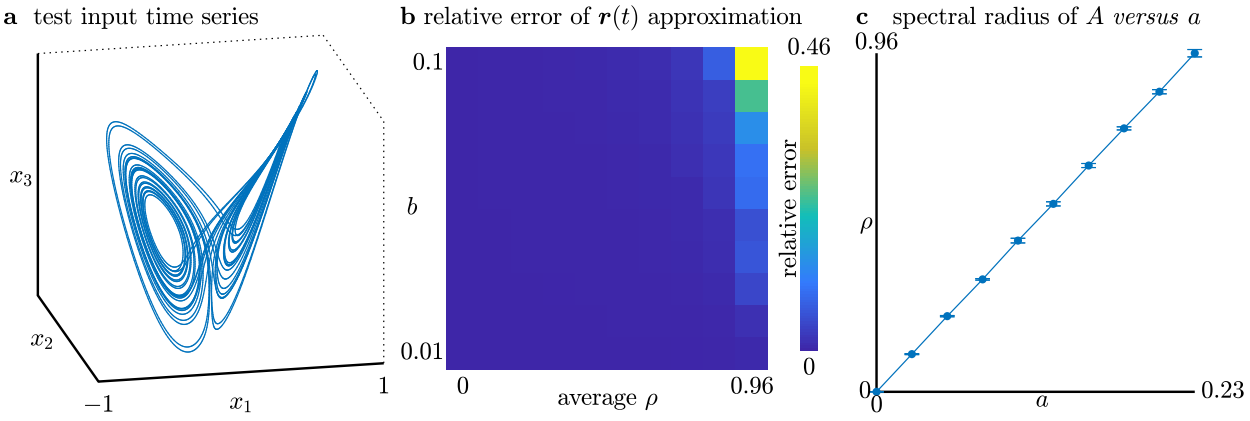

FIG. 2. **Testing the goodness of the linearization for a larger spectral radius.** (**a-c**) Same procedure as the previous figure, except we vary the multiplier a from 0 to 0.23, which corresponds to varying the spectral radius from 0 to around 0.96. Outside of the largest spectral radii, the relative error of the approximation remains quite low, with a maximum of around 0.06 for the second largest spectral radius at $b = 0.1$. Error bars in panel **c** are standard deviations of the spectral radius of the adjacency matrix A.

To further contextualize this result, we next note that the behavior of the RC is intimately tied to both the scaling a and the number of neurons n. Specifically, the dynamical behavior of the RC depends fundamentally on the *spectral radius* ρ of the adjacency matrix, which is given by the magnitude of its largest eigenvalue. We show that in our $n = 1000$ neuron network, the largest scaling value corresponds to a spectral radius of $\rho \approx 0.42$, thereby demonstrating that our approximation accurately captures the non-trivial internal dynamics of the RC (Fig. 1c).

To measure the goodness of the linearization for larger spectral radii, we perform the same analysis but sweep across larger values of a, thereby yielding RNNs with spectral radii near 1. At the largest radius around $\rho \approx 0.96$, we observe large relative error (≈ 0.47). However, even at the next largest spectral radius ($\rho \approx 0.86$), the relative error drops dramatically to ≈ 0.06 (Fig. 2a-c).

In sum, we demonstrate that a naive test of the linearization accuracy yields a good and robust approximation across a typical range of parameters. Further, this accuracy can be improved by rescaling the magnitude of the input—a common practice in many disciplines—as well as the selection of a small spectral radius.

V. UPPER BOUND ON THE GOODNESS OF THE LINEARIZATION FOR SPECIFIC EXAMPLES IN THE MAIN TEXT

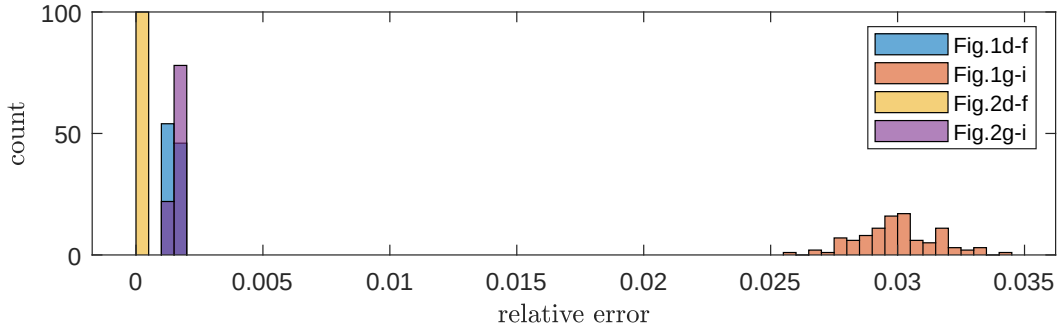

FIG. 3. **Testing the goodness of the linearization for specific examples.**

Here, we quantify and discuss the goodness of the linearization for the specific examples in the main text. We calculate the relative error of the RNN states generated from the nonlinear equations $\boldsymbol{r}(t)$ against those generated from the linearization $\boldsymbol{r}'(t)$ for Fig.1d-f, Fig.1g-i, Fig.2d-f, and Fig.2g-i in the main text. For each figure, we randomly generate all parameters in the RNN and the inputs for 100 instances, and we plot a histogram of the relative error over those 100 instances (Fig. 3).

For Fig.1d-f of the main text (Fig. 3, blue), we notice a small but non-zero error. This error arises as a result of the small spectral radius of the connectivity matrix A, which was set to 0.01. Because of this small but non-zero spectral radius, the linearization correspondingly has a small but non-zero error.

For Fig1.g-i, we notice a larger relative error with a mean of about 0.03. This larger error arises as a result of the larger spectral radius of the connectivity matrix A, which was set to 0.2. This larger spectral radius allows us to retain more of a time history of the inputs, but comes at the cost of increased error. Nevertheless, a relative error of 0.03 is still small enough for us to consistently produce the high-pass filtered output in Fig.1g-i of the main text. Importantly, a simple and highly effective way to retain an arbitrarily long time-history of the inputs without producing a large relative error is provided in Fig.2g-i of the main text.

For Fig.2d-f and Fig.2g-i of the main text, the RNNs have an initial spectral radius of 0.0001 and 0.01, respectively, thereby leading to good linearization approximations (Fig. 3).

VI. GOODNESS OF THE EXPANSIONS IN INPUT VARIABLES AND TIME

Now that we have a clear idea of how our parameter choices affect the accuracy of the linearization, we next quantify the accuracy of the subsequent approximations in time (Eq. 4) and in the input variables (Eq. 5. To do this, we perform the same protocol as in the previous section by driving the RNN according to the chaotic Lorenz attractor (Eq. 8). We measure the relative error between the reservoir states generated by the full nonlinear equation in Eq. 1, $\boldsymbol{r}(t)$, and those generated by the polynomial expansion of Eq. 5, $\boldsymbol{r}^{\dagger}(t)$, as a function of the number of expansions in time (i.e., more time derivatives) and in input variables (i.e., higher-order powers of $\boldsymbol{x}$). We use the same n, γ, and $\boldsymbol{r}^*$ distribution.

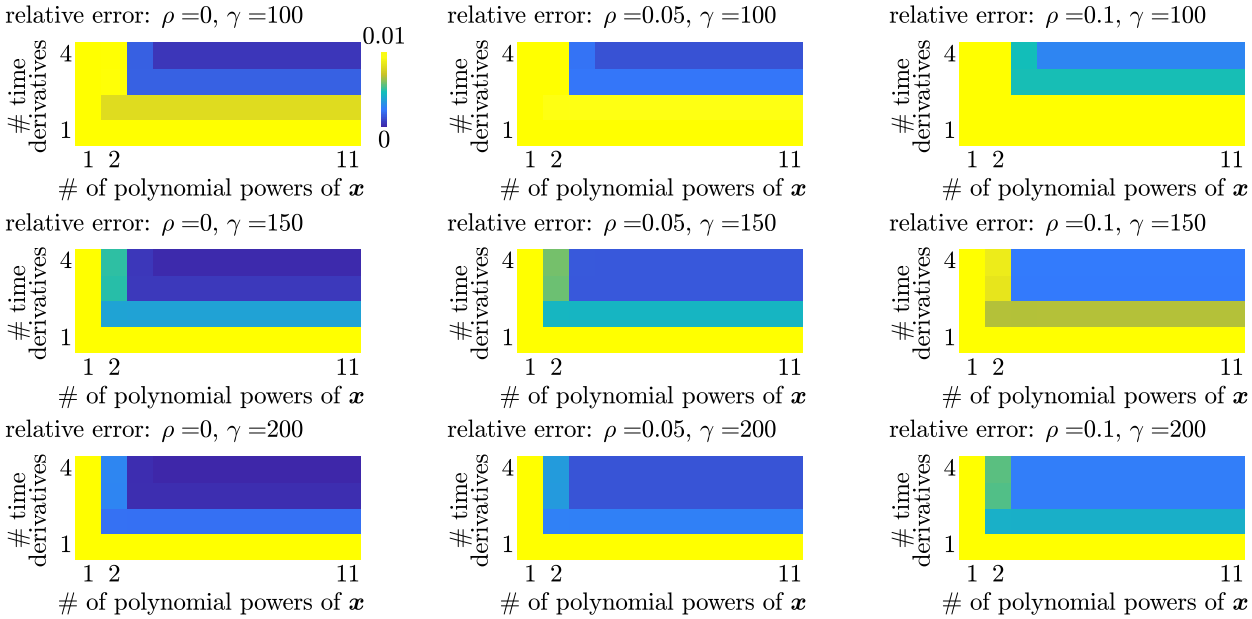

FIG. 4. **Goodness of the expansion.** Relative error between the true RC time series (Eq. 1) and the estimated RC time series (Eq. 5) at various spectral radii ρ and time constants γ. Generally, the approximation improves with smaller ρ, larger γ, and more expansion terms in the number of time derivatives and the number of polynomial powers of $\boldsymbol{x}$.

We do not require too many time derivatives or polynomial powers of $\boldsymbol{x}$ to achieve a relative error that is significantly lower than 0.01 (Fig. 4), and the approximation converges rapidly with smaller spectral radius ρ and larger γ. Similar to the previous section, what matters is not the absolute value of γ, but rather the relative ratio of γ to the time derivatives $\dot{\boldsymbol{x}}, \ddot{\boldsymbol{x}}, \cdots$, and we can improve the accuracy simply by changing the time scale of $\boldsymbol{x}(t)$.

VII. PERFORMANCE COMPARISON WITH ECHO-STATE AND FORCE FOR TASKS INVOLVING LONG RETENTION OF TIME-HISTORY

Given the fact that our feedback examples in the main text started with RNNs with weak initial connectivity (i.e., usually $\rho \leq 0.01$), one concern may be that such RNNs will not be capable of storing any time history or memory of their inputs. This concern is resolved by realizing that once we compile our program into matrix $\bar{W}$ and perform feedback, our initially unconnected RNN is given an internal connectivity defined by $A = \bar{A} + \bar{B}\bar{W}$. Hence, one contribution of our framework is to program a connectivity matrix that stores a precisely specifiable time-history for transforming data, solving equations, and running simulations, rather than initializing a random connectivity matrix A that generates a random time-history of its inputs. The power of this ability to precisely specify how time-history is represented and manipulated is seen in the extended capabilities of programmed RNNs that are otherwise very difficult to train using conventional training methods, such as the ability to simulate a playable game or to precisely solve algebraic equations.

However, an important question remains: how does this RNN programming framework hold up to conventional reservoir techniques for solving conventional reservoir problems that require long time history? While we presented a demonstration of our framework where we precisely programmed a playable video game, the internal governing physics of the game states and the user inputs, along with the simulation of the long-term time-history of the game states, all directly compiled into the recurrent connections of an RNN, how effective is our framework for outputting a function of the input time-history compared to conventional reservoir computing? What are the tradeoffs compared to conventional techniques?

Here we explore these questions and comparisons through a simple task: the time-lagging of random noise. Specifically, we consider two RNNs. The first RNN is programmed according to the time-lag feedback procedure used in Fig.2g-i of the main text, except now at the limit of maximum performance with initial connectivity $\bar{A} = 0$. We scale the input matrix B whose entries are dense and drawn from a random uniform distribution in $[-0.5, 0.5] * b$. After programming and feedback, the RNN is driven by a scalar input time series, x_t, and has recurrent connections defined by $A = \bar{B}\bar{W}$. The second RNN is a conventional reservoir whose recurrent connectivity matrix A is randomly generated to be 5% dense and scaled to have a spectral radius of $\rho = 1$, has an input matrix B whose entries are dense and

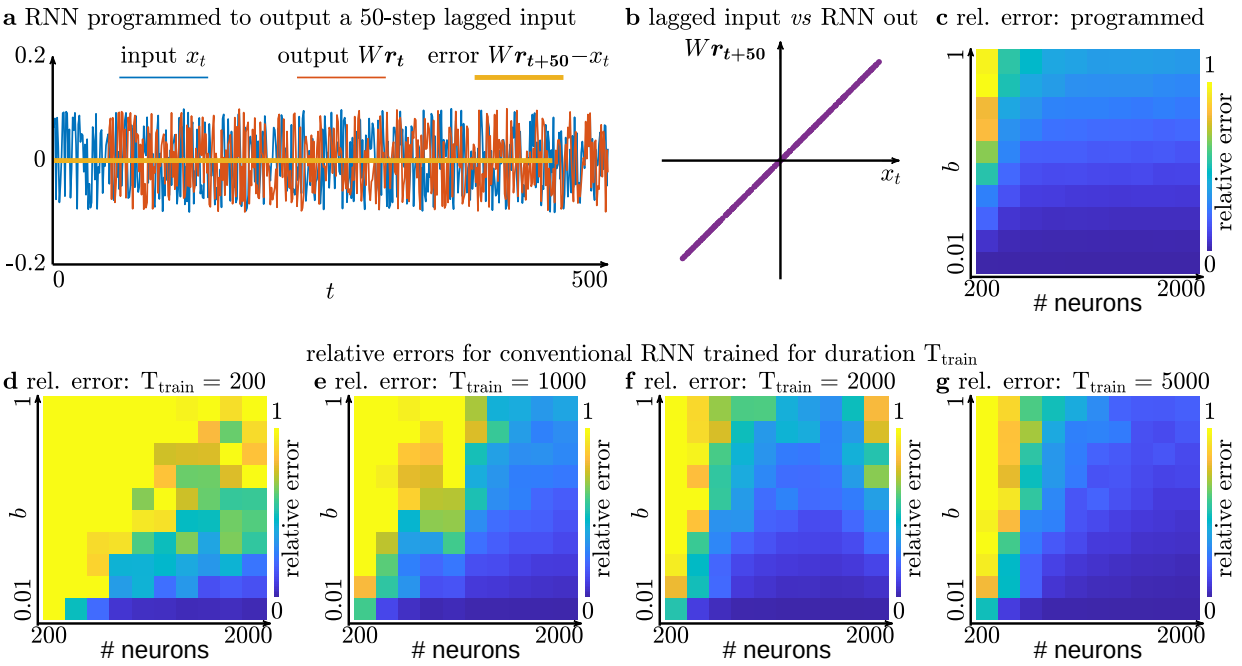

FIG. 5. **Performance comparison between a programmed RNN and a conventional RNN with spectral radius $\rho = 1$ for an n-lag task.** (**a**) The input into all RNNs is a randomly generated time signal drawn from a uniform distribution between -0.1 and 0.1 (blue). The RNN is programmed to lag the signal by 50 steps according to the same protocol in the main text Fig. 2g-i, thereby producing a time-lagged output (orange). Notice that for the first 50 time steps, the output of the RNN is close to 0. This is not because the RNN is not receiving input. Rather, by programming the RNN to only output the 50th lagged term, the inputs drive the RNN along the nullspace of the output matrix W for the first 50 time steps. (**b**) Plot of the RNN output shifted by 50 time steps *versus* the input. (**c**) The relative error between the true signal and the RNN output shifted by 50 time steps for a wide range of parameters. (**d-g**) Relative errors of a conventional RNN with randomly generated recurrent connections having a spectral radius of $\rho = 1$ for different amounts of training. While performance increases with the training time, the conventional RNN consistently underperforms the programmed RNN across all tested parameters.

drawn from a random uniform distribution in $[-0.5, 0.5] * b$, and is also driven by the same scalar input time series, x_t. The second RNN is trained by first driving it with the random noise signal for T_{train} time steps, and solving for an output matrix W that reproduces a 50 time-step lagged version of the input. As can be seen the second, trained RNN consistently underperforms the first, programmed RNN across all tested parameters (Fig. 5d-g). If we

extend the difficulty of this task to 100 lags, we see an even more dramatic underperformance of the conventional reservoir compared to the programmed one (Fig. 6).

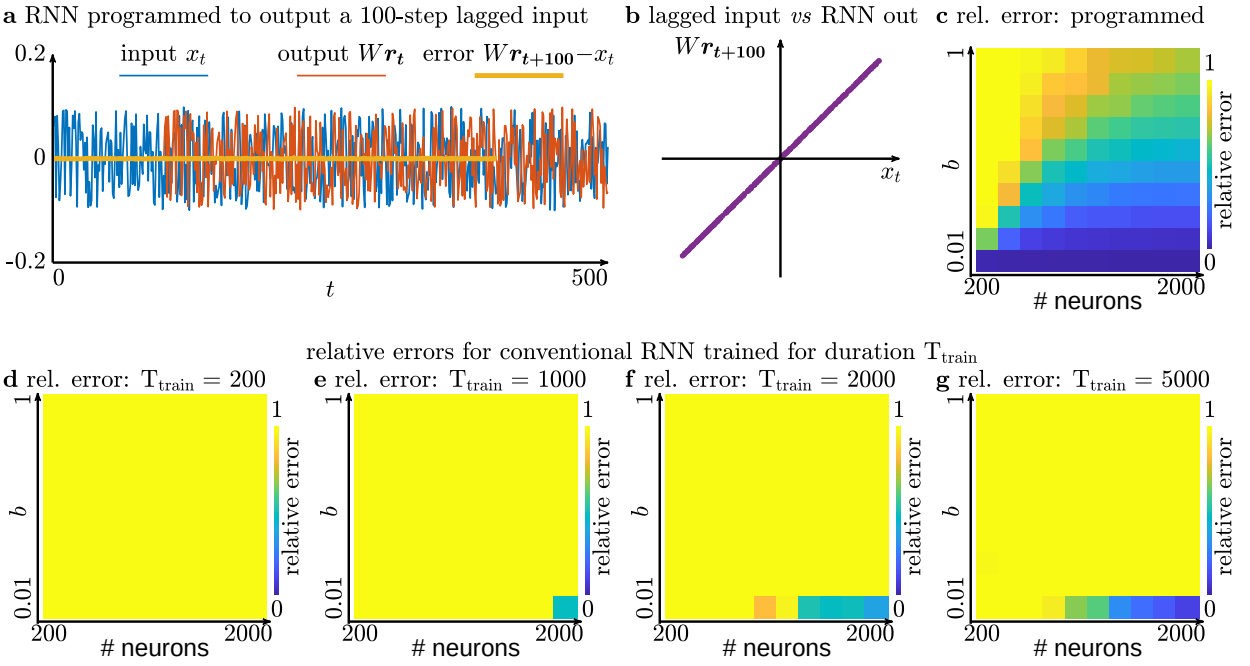

FIG. 6. **Performance comparison between a programmed RNN and a conventional RNN with spectral radius $\rho = 1$ for an n-lag task.** The same analysis as in Fig. 5, except now for a 100-lag task.

To explore these questions and comparisons in a more realistically useful task, we perform the same comparison between programmed RNNs and conventional reservoir RNNs through the short-time discrete Fourier transform shown in Fig. 2g-i in the main text. Specifically, we program the first RNN the same way we do in Fig. 2g-i in the main text, and compute the relative error compared to the true Fourier transform. Then we train the second RNN to produce the same output as the first RNN using training data of varying lengths, and compute the relative error compared to the true Fourier transform. For a window size of 49 samples, the second trained RNN performs comparably to the programmed RNN (Fig. 7d-g). However, as we increase the window size to 75 (Fig. 8) and 99 (Fig. 9), the performance of the conventional reservoir drops off substantially faster than that of the programmed RNN.

We hypothesize that the improved performance of the programmed RNNs is because we can efficiently assign the input time-history that is retained, as opposed to conventional RNNs which generate a diverse, yet random and less efficient representation of the input

time-history. Hence, in this supplementary section, we demonstrate that programming time-history into RNNs is often more efficient and accurate than conventional reservoir RNNs.

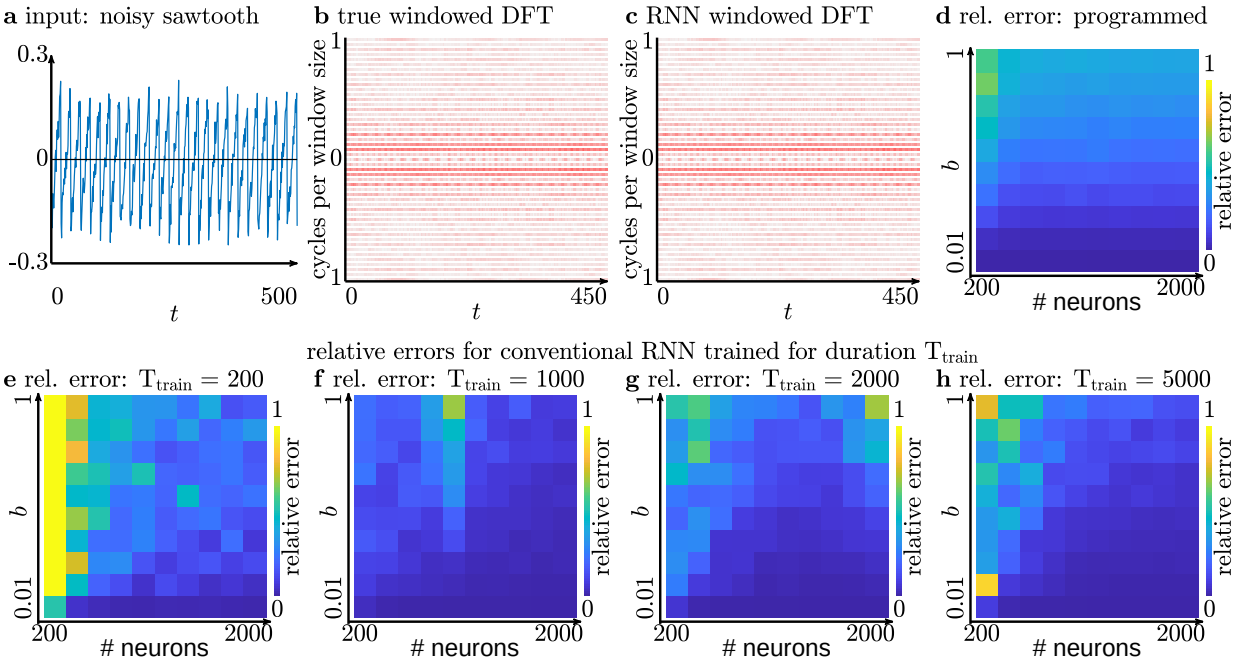

FIG. 7. **Performance comparison between a programmed RNN and a conventional RNN with spectral radius $\rho = 1$ for a short-time discrete Fourier transform with a window size of 49 samples.** (**a**) The input signal into both the programmed and trained RNNs is a noisy sawtooth wave with amplitude 0.2 and noise drawn from a uniform random distribution between -0.05 and 0.05. (**b**) The true windowed DFT of the input signal, and (**c**) the DFT from the output of the programmed RNN. (**d**) The relative error between the true windowed DFT and that from the output of the programmed RNN across a range of parameters. (**e-h**) The relative error between the true windowed DFT and that from the output of the trained RNNs across a range of training windows.

There are two important points of discussion here. First, our programmed RNN, although it has a spectral radius of 0 before programming, ends up having a much larger spectral radius after programming. For Fig. 5, we find the programmed connectivity to have a spectral radius that falls between 0.92 and 0.98. However, the intention behind these benchmarks is not to compare spectral radii at all. Rather, the point we are making here is one of random state retention *versus* programmed state retention. In a randomly initialized system such as the trained RNN with a spectral radius of 1, the system retains a substantial time history

of its inputs as desired, but needs many neurons to retain both the relevant and irrelevant variables (Fig. 8). The first statement we are making here is that an RNN can instead be programmed to primarily retain only the relevant variables (i.e., a time history of linear terms) to achieve better performance with fewer neurons (i.e., $N = 200$ in Fig. 8d,h).

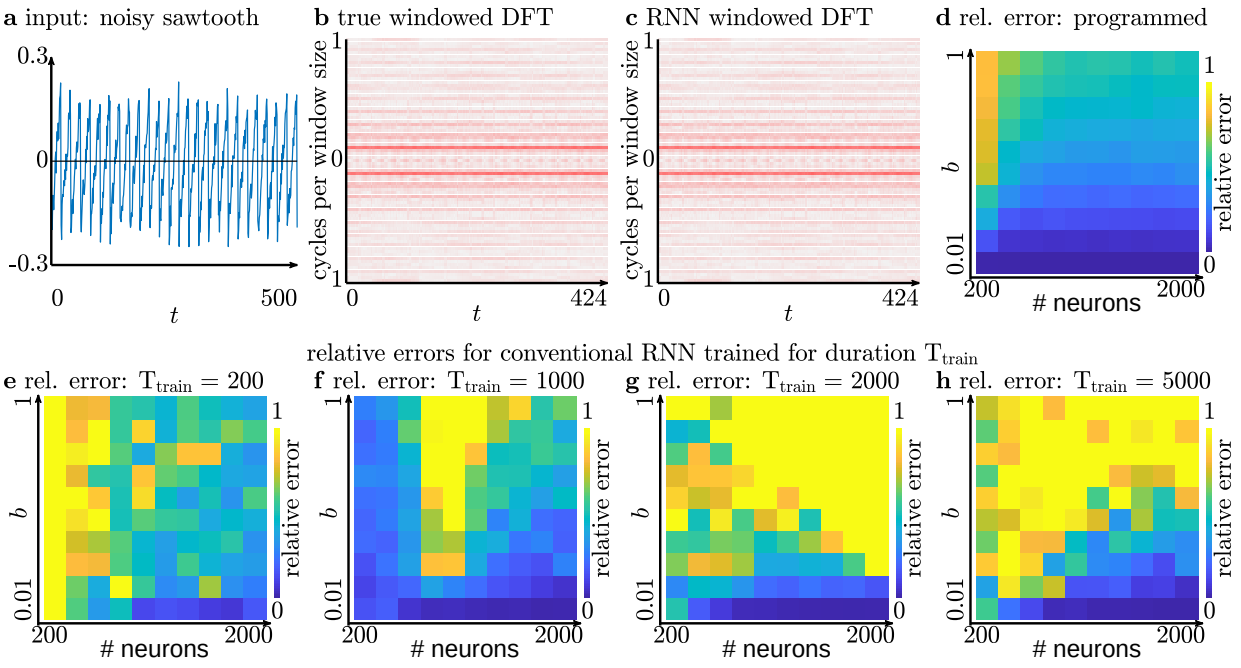

FIG. 8. **Performance comparison between a programmed RNN and a conventional RNN with spectral radius $\rho = 1$ for a short-time discrete Fourier transform with a window size of 75 samples.**

Second, the training of the non-programmed RNNs with $\rho = 1$ is not performed in an adaptive manner, but in a one-shot, least-squares manner common for echo-state approaches. Hence, there are no issues of getting trapped in local minima using gradient methods, and no concerns about vanishing or exploding gradients, which remain some of the crucial advantages of reservoir computing.

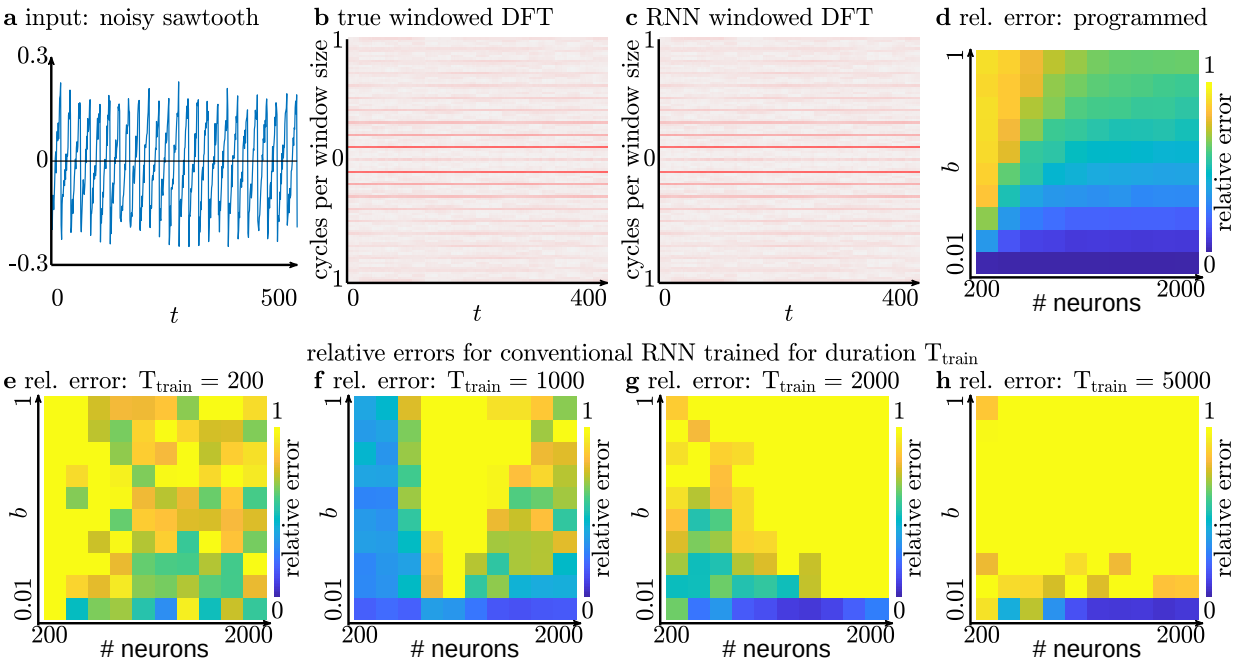

FIG. 9. **Performance comparison between a programmed RNN and a conventional RNN with spectral radius $\rho = 1$ for a short-time discrete Fourier transform with a window size of 99 samples.**

A. Comparison with FORCE

In addition to providing benchmarks against conventional RC paradigms, we also include a comparison with FORCE learning [1] using example code from the original paper. Adapting the original continuous-time reservoir to a discrete-time reservoir gives rise to a FORCE RNN that is identical to our current RNN equation without a bias term,

$$\begin{aligned} \boldsymbol{r}_t &= \tanh(A\boldsymbol{r}_{t-1} + \bar{B}\boldsymbol{z}_{t-1} + Bx_{t-1}), \\ \boldsymbol{z}_t &= W_{t-1}\boldsymbol{r}_t. \end{aligned}$$

Here, $\boldsymbol{z}_t$ is the output that generates the error signal for the adaptation of W, A has density $\rho = 0.05$ with elements drawn from a normal distribution and scaled by $g/\sqrt{\rho N}$, and both $\bar{B}$ and B are scaled in the same manner as the previous examples in this section. We implement the identical architecture in the FORCE RNNs as we do in the programmed RNNs, such that $\bar{B}$ is an $N \times k$ matrix of k time-step lags, and $\boldsymbol{z}$ is a $k \times 1$ vector containing all k lags.

The FORCE algorithm is implemented by keeping track of a running estimate of the

inverse covariance P, such that for every time step t, we have a rank-1 update

$$P_t = P_{t-1} - \frac{(P_{t-1}\boldsymbol{r}_t)(P_{t-1}\boldsymbol{r}_t)^\top}{1 + \boldsymbol{r}_t^\top P_{t-1}\boldsymbol{r}_t},$$

where $P_0 = I$. The error signal is calculated as

$$\boldsymbol{e}_t = \boldsymbol{z}_t - \begin{bmatrix} x_{t-1} \\ x_{t-2} \\ \vdots \\ x_{t-\tau} \end{bmatrix},$$

where τ is the number of time lags (e.g., 50 or 100). Finally, the update to the output matrix W is given by

$$W_t = W_{t-1} - \left(\frac{P_{t-1}\boldsymbol{r}_t}{1 + \boldsymbol{r}_t^\top P_{t-1}\boldsymbol{r}_t}\right)\boldsymbol{e}_t^\top.$$

We note that these equations were transferred directly from the example code from the original publication. For the following results in Fig. 10,11, we tried a range of scaling factors $0.1 <= g <= 1.5$ for the initial A, and report the best performing results.

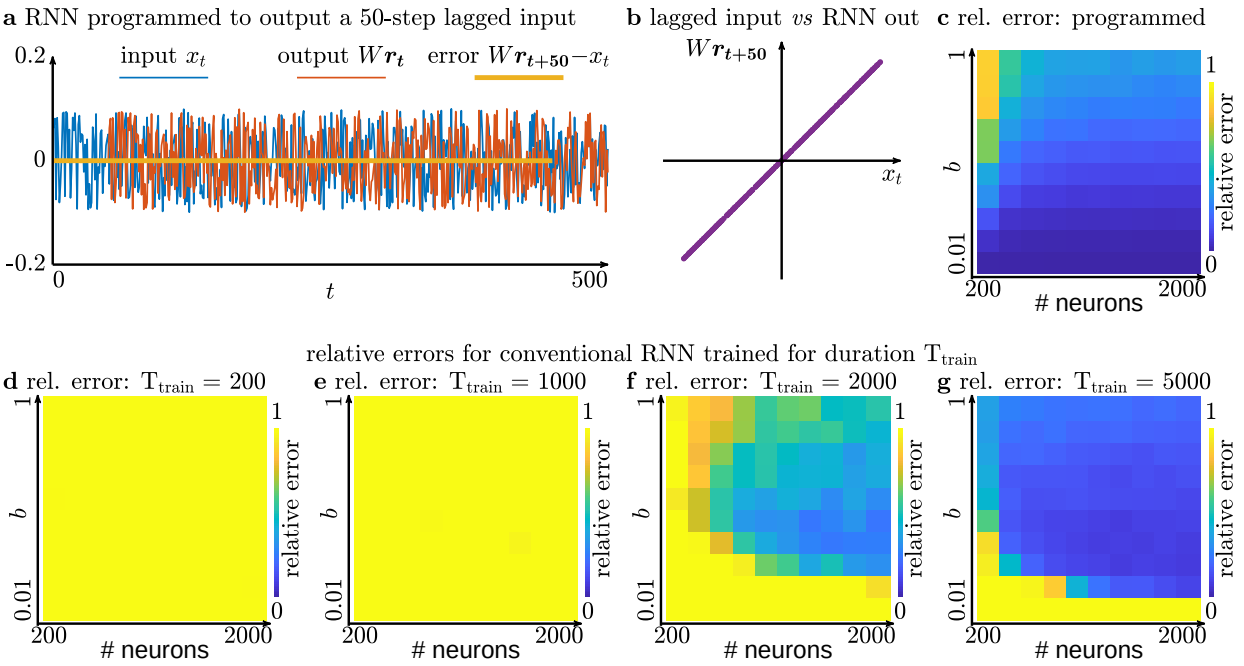

FIG. 10. **Performance comparison between a programmed RNN and the feedback FORCE RNN with spectral radius $g = 0.8$ for the 50-lag task.**

For the 50-lag task in Fig. 10, we observe good performance in the FORCE network. Contrary to the programmed and echo-state networks, the FORCE network performs optimally at an intermediate scaling of the input matrix B. Importantly, we note that the performance was worse for the same training times when using $g > 0.8$.

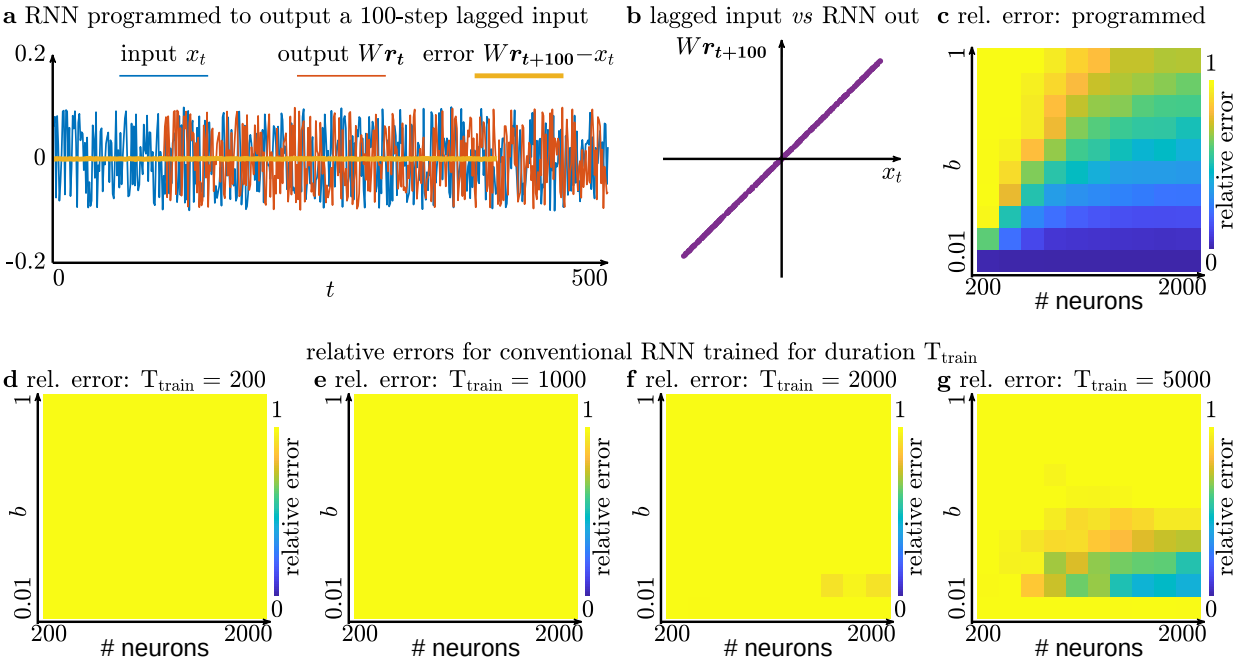

FIG. 11. **Performance comparison between a programmed RNN and the feedback FORCE RNN with spectral radius $g = 1.0$ for the 100-lag task.**

For the 100-lag task in Fig. 11, we observe poor performance from the FORCE network until the largest training window of 5000. Hence, we increased the training times by an order of magnitude (Fig. 12). We observe that with longer training windows, the performance of the FORCE network continues to improve.

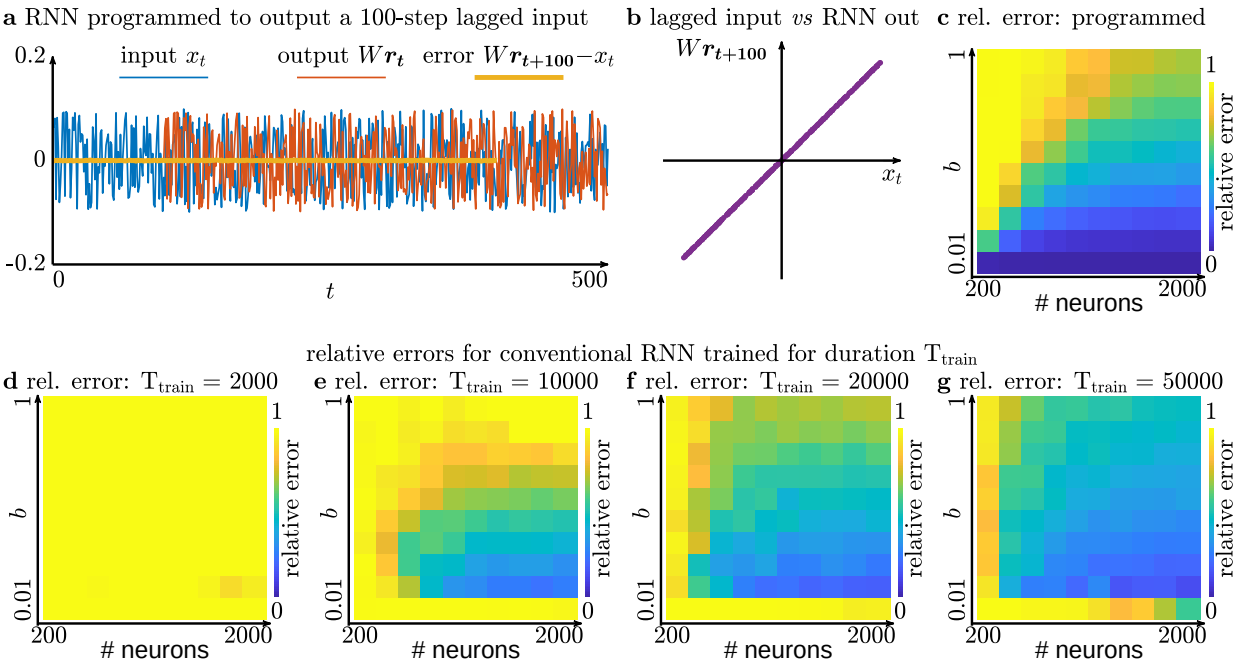

FIG. 12. **Performance comparison between a programmed RNN and the feedback FORCE RNN with spectral radius $g = 1.0$ for the 100-lag task.**

VIII. DNP PROGRAMMING DYNAMICS IN THE REGIME OF $\bar{A} << 1$

Throughout the results, there arises a seeming paradox in SNP: the approximation of the decompiler works best when $\bar{A} << 1$, but the lack of recurrent connections preclude a meaningful storage of time-history in the inputs. This paradox is precisely the problem that feedback solves. Although our decompiling of the neural states is only valid for small initial $\bar{A}$, it is by programming a large $A = \bar{A} + \bar{B}\bar{W}$ that we explicitly encode hidden variables and relationships between them to perform exactly the complex operations that we want.

Specifically, by programming an output function $\bar{\boldsymbol{o}} = \bar{\boldsymbol{f}}(\bar{\boldsymbol{x}}, \boldsymbol{x})$ through compiling an output matrix $\bar{W}$ that is a weighted sum of the neural representations of the input variables $\boldsymbol{r}(\bar{\boldsymbol{x}}, \boldsymbol{x})$, we can define an output that is a function of the neural states $\bar{\boldsymbol{o}} = \bar{W}\boldsymbol{r}$. By feeding these outputs back into the inputs, $\bar{\boldsymbol{o}} = \bar{\boldsymbol{x}}$, we fix an equivalence relationship between the outputs and inputs such that $\boldsymbol{f}(\bar{\boldsymbol{x}}, \boldsymbol{x}) = \bar{\boldsymbol{x}}$. The implementation of this feedback in our RNNs is to go from the original equation

$$\frac{1}{\gamma}\dot{\boldsymbol{r}} = -\boldsymbol{r} + \tanh(\bar{A}\boldsymbol{r} + \bar{B}\bar{\boldsymbol{x}} + B\boldsymbol{x} + \boldsymbol{d})$$

to the feedback equation

$$\frac{1}{\gamma}\dot{\boldsymbol{r}} = -\boldsymbol{r} + \tanh((\bar{A} + \bar{B}\bar{W})\boldsymbol{r} + B\boldsymbol{x} + \boldsymbol{d}).$$

Hence, this feedback is precisely how we go from $\bar{A} << 1$ in the initialized RNN to $A = \bar{A} + \bar{B}\bar{W}$ in the programmed feedback RNN. The recurrent connectivity $A = \bar{A} + \bar{B}\bar{W}$ contains all of the information about $\boldsymbol{f}(\bar{\boldsymbol{x}}, \boldsymbol{x}) = \bar{\boldsymbol{x}}$. One way to use this recurrence relation to store time history is precisely in the time-lag example of the main text, where we define a shift operator to lag the input signal at each time step across a vector of internal states.

Additionally, although a small initial recurrent connectivity precludes the intrinsic storage of long time-history, we use *dynamical neural programming*, or DNP to explicitly encode temporal information between hidden variables through feedback. Specifically, Sections II and III of the main text, corresponding to Fig.1 and Fig.2, deal with writing the neural states $\boldsymbol{r}$ as a function of the inputs $\boldsymbol{x}$. However, when the initial connectivity matrix $\bar{A} << 1$, then $\boldsymbol{r}(t)$ contains little information about the time history of $\boldsymbol{x}$, whether that be time-derivatives of $\boldsymbol{x}$ in continuous-time systems, or time-lags of $\boldsymbol{x}$ in discrete-time systems. To specify the dynamics to run temporal simulations and emulations (Fig.3), store dynamical

memories (Fig.4), simulate permanent persistent memory in logic gates (Fig.5), and simulate a playable game of pong (Fig.6), rather than using $\boldsymbol{r}(t)$, which contains little time-history information when $\bar{A} << 0$, we use $\tanh(\bar{A}\boldsymbol{r}+\bar{B}\bar{\boldsymbol{x}}+B\boldsymbol{x}+\boldsymbol{d})$, which does contain time-history information. Specifically, if we consider the RNN equation

$$\frac{1}{\gamma}\dot{\boldsymbol{r}} = -\boldsymbol{r} + \tanh(\bar{A}\boldsymbol{r} + \bar{B}\bar{\boldsymbol{x}} + B\boldsymbol{x} + \boldsymbol{d}),$$

and add $\boldsymbol{r}$ to both sides, then we obtain

$$\tanh(\bar{A}\boldsymbol{r} + \bar{B}\bar{\boldsymbol{x}} + B\boldsymbol{x} + \boldsymbol{d}) = \boldsymbol{r} + \frac{1}{\gamma}\dot{\boldsymbol{r}}.$$

Hence, by decompiling tanh with respect to the inputs $\boldsymbol{x}$ and $\bar{\boldsymbol{x}}$, we approximate not only $\boldsymbol{r}$, but also $\dot{\boldsymbol{r}}$. When we write our program source code, we are not trying to map $\boldsymbol{r}$ to functions of $\bar{\boldsymbol{x}}$, but rather

$$\bar{W}\tanh(\bar{A}\boldsymbol{r} + \bar{B}\bar{\boldsymbol{x}} + B\boldsymbol{x} + \boldsymbol{d}) = \bar{W}\left(\boldsymbol{r} + \frac{1}{\gamma}\dot{\boldsymbol{r}}\right) = \bar{\boldsymbol{x}} + \frac{1}{\gamma}\dot{\bar{\boldsymbol{x}}}.$$

We thus form the matrix of coefficients G that is formed *via* the Taylor series expansion of tanh with respect to polynomial variables $\bar{\boldsymbol{x}}$ and $\boldsymbol{x}$. Hence, whereas R was the programming matrix using neural states (and hence, the *state* neural programming, or SNP), we call G the dynamical neural programming. Once we perform feedback, we again define the internal recurrent connectivity of the RNN as

$$\frac{1}{\gamma}\dot{\boldsymbol{r}} = -\boldsymbol{r} + \boldsymbol{g}(\bar{A}\boldsymbol{r} + \bar{B}\bar{W}\boldsymbol{r} + B\boldsymbol{x} + \boldsymbol{d}),$$

which stores both the state, $\bar{\boldsymbol{x}}$, and the dynamic $\dot{\bar{\boldsymbol{x}}}$ information in the connections $A = \bar{A} + \bar{B}\bar{W}$.

We notice that the DNP G on the left-hand side is a function of the inputs $\bar{\boldsymbol{x}}$ and $\boldsymbol{x}$. Hence, the variables are still polynomial terms of $\bar{\boldsymbol{x}}$ and $\boldsymbol{x}$. This means that any program we write on the right-hand side must also be written with respect to polynomial terms of $\bar{\boldsymbol{x}}$ and $\boldsymbol{x}$. Hence, the term $\dot{\bar{\boldsymbol{x}}}$ on the right-hand side must take the form $\dot{\bar{\boldsymbol{x}}} = \boldsymbol{f}(\bar{\boldsymbol{x}}, \boldsymbol{x})$, which is precisely the form of a driven dynamical system. Therefore, by using G as our programming matrix, we can write source code that is a driven dynamical system, and perform feedback such that $A = \bar{A} + \bar{B}\bar{W}$ to program that driven dynamical system into the recurrent weights of the reservoir. Importantly, we acknowledge that these programmed dynamics will always be an *approximation* of the true dynamics $\dot{\bar{\boldsymbol{x}}} = \boldsymbol{f}(\bar{\boldsymbol{x}}, \boldsymbol{x})$ as a result of requiring a finite number of neurons, a truncation of the polynomial basis, and the linearization of the full nonlinear system. As a result, the *stability* of the programmed dynamics is not guaranteed.

IX. SIMULATION PARAMETERS

Throughout the main text, we provide many examples involving different dynamical equations and RC parameters. Here we explicitly write out the parameter choices and equations used. In line with the previous sections on the goodness of the approximation, the connectivity matrix A is 0.05% dense, whereas the input matrix B is fully dense. The elements of A and B are always drawn from uniform random distributions. We discuss B with respect to the amount by which it is scaled, and A with respect to its spectral radius ρ. All simulations were run by integrating the continuous-time differential equations using the 4th order Runge-Kutta method with a time step of $\mathrm{d}t = 0.001$. Hence, a simulation that is run for T time *units* means that the simulation was evolved by $T/\mathrm{d}t$ steps. Throughout, we used $\gamma = 100$, and drew $\boldsymbol{r}^*$ from a random uniform distribution whose values ranged between -0.5 and 0.5.

A. Parameters for "Open-Loop Architecture with SNP for Output Functions"

In the rotation example, we used a reservoir with $N = 1000$ neurons, $\rho = 0.01$, and B is scaled by 0.1. For the Thomas attractor, we chose an initial condition of $x_1(0) = 0, x_2(0) = 0$, and $x_3(0) = 1$, and then we evolved the Thomas attractor for 100 time units, drove the RC for those 100 time units, and discarded the first 20 time units to allow the RC to forget about its initial condition. In the high-pass filter example, we used a reservoir with $N = 1000$ neurons, $\rho = 0.2$, and B scaled by 0.1. For the sinusoidal input, we drive the RC with $x_t = \sin(2\pi t) + 0.2\sin(8\pi t)$.

B. Parameters for "Closed-loop Architecture with SNP to Solve Algorithms"

In the closed-loop examples for Fig. 2d-f of the main text, we used an RC with $N = 4000$ neurons, $\rho = 0.0001$, and B that was scaled by 0.0025. The input matrix is a 6×6 matrix whose elements are randomly generated between -0.5 and 0.5, and subsequently subtracted by the identity matrix. This subtraction is to ensure that M is stable—all real eigenvalues are negative—because the Lyapunov equation does not possess a solution otherwise. In the closed-loop examples for Fig. 2g-i of the main text, we used an RC with $N = 500$ neurons, $\rho = 0.01$, and B that was scaled by 0.0005. The noisy sawtooth is generated by

$x_t = 0.2\text{sawtooth}(\frac{2\pi t}{8}) + \epsilon_t$, where ϵ_t is drawn uniformly from -0.05 to 0.05.

C. Parameters for "Closed-Loop RNN with DNP to Simulate and Virtuailze"

In programming the RC of Fig. 3 in the main text, we used a guest RC with $N = 15$ neurons, and a host RC with $N = 2000$ neurons. For the guest and host RCs, $\rho = 0.01$, and B was scaled by 0.2. The Lorenz equation that was programmed was

$$\begin{aligned}
\dot{x}_1 &= x_2 - x_1 \\
\dot{x}_2 &= \frac{1}{10}x_1 - \frac{1}{10}x_2 - 20x_1x_3 \\
\dot{x}_3 &= 20x_1x_2 - \frac{4}{15}x_3 - 0.036.
\end{aligned}$$

D. Parameters for "Op-Codes, Composition, and Dynamic RAM"

For panels Fig.4a–d, we used one RC with $N = 5000$ neurons, B scaled by 0.01, and C scaled by 1.5. The elements of the 4×4 matrices P and Q were drawn randomly and uniformly between -1 and 1. For panel Fig.4e, the NPU has $N = 2500$ neurons, $\rho = 0$, and B scaled by 0.0005. The two NPUs are chained together to form a system of $N = 5000$ neurons. The elements of the 5×5 matrix M were drawn randomly and uniformly between -1 and 1, and the elements of the 5×1 vector $\boldsymbol{b}$ were drawn randomly and uniformly between -0.5 and 0.5. To find the op-code fixed points, we simply evolved the systems with only the op-code inputs until the RNN dynamics converged to the associated fixed points.

For panels Fig.4f-i, the RC has $N = 1000$ neurons, $\rho = 0.01$, and B that was scaled by 0.3. The equations for the dynamical attractors are

$$\begin{aligned}
\dot{x}_1 &= 10(x_2 - x_1) \\
\dot{x}_2 &= x_1 - x_2 - 20x_1x_3 \\
\dot{x}_3 &= 20x_1x_2 - \frac{8}{3}\left(x_3 - \frac{27}{20}\right),
\end{aligned}$$

for the Lorenz,

$$\begin{aligned}
\dot{x}_1 &= -5_2 - 5x_3 - 3 \\
\dot{x}_2 &= 5x_1 + x_2 \\
\dot{x}_3 &= \frac{250}{3}x_1x_3 + 50x_1 - 28.5x_3 - 17.04,
\end{aligned}$$

for the Rossler,

$$\dot{x}_1 = -2.1x_1 - 6x_2 - 6x_3 - 15x_2^2$$
$$\dot{x}_2 = -2.1x_2 - 6x_3 - 6x_1 - 15x_3^2$$
$$\dot{x}_3 = -2.1x_3 - 6x_1 - 6x_2 - 15x_1^2,$$

for the Halvorsen, and

$$\dot{x}_1 = -8x_2$$
$$\dot{x}_2 = \frac{25}{4}x_1 + 5x_3^2$$
$$\dot{x}_3 = \frac{5}{4} + 20x_2 - 10x_3,$$

for the Sprott N.

E. Parameters for "A Logical Calculus Using Recurrent Neural Circuits"

In programming the RCs of Fig. 5 in the main text, each logic gate used an RC with $N = 30$ neurons, with $\rho = 0$ and B scaled by 0.025. For the logic gates, we programmed in a 1-dimensional scalar dynamical system

$$\dot{x} = \frac{3}{13}x^3 - \left(123 + \frac{1}{13}\right)x + z, \tag{9}$$

where z is a function of p and q given by

$$z = -0.1 + (p + 0.1)(q + 0.1)/0.2, \qquad \text{AND}$$
$$z = 0.1 + (p + 0.1)(-q - 0.1)/0.2, \qquad \text{NAND}$$
$$z = 0.1 + (p - 0.1)(-q + 0.1)/0.2, \qquad \text{OR}$$
$$z = -0.1 + (p - 0.1)(q - 0.1)/0.2, \qquad \text{NOR},$$

and

$$z = -(p)(q)/0.1, \qquad \text{XOR}$$
$$z = (p)(q)/0.1, \qquad \text{XNOR}.$$

F. Parameters for "Game Development and Decompiling Trained RNNs"

The RC used to program our game of pong had $N = 1000$ neurons, $\rho = 0$, and B scaled by 0.05. The logic gates for the flip-flop were programmed using the same equations as in

the previous section (Eq. 9), multiplied by 20 to improve the speed of the response. The RC for the highpass filter had $N = 2000$ neurons, $\rho = 0.4$, and B scaled by 0.1.

X. COMPARISON WITH THE NEURAL ENGINEERING FRAMEWORK

The methodology bears much similarity to the Neural Engineering Framework (NEF) in both spirit and technical detail. Here we highlight the similarities and differences between our framework and NEF [2].

Principle 1: Representation The NEF is characterized by three key principles. The first principle is representation, whereby an input, $\boldsymbol{x}(t) \in \mathbb{R}^{k\times 1}$, is converted to a synaptic current $\boldsymbol{s}(t) \in \mathbb{R}^{N\times 1}$ *via* a projection matrix, $E \in \mathbb{R}^{N\times k}$, which is scaled by gains, $\boldsymbol{\alpha} \in \mathbb{R}^{N\times 1}_{>0}$, then shifted by a bias, $\boldsymbol{\beta} \in \mathbb{R}^{N\times 1}$ such that

$$\boldsymbol{s}(t) = \mathrm{diag}(\boldsymbol{\alpha})E\boldsymbol{x}(t) + \boldsymbol{\beta},$$

which is subsequently transformed to neural activity *via* a nonlinear spiking function G

$$\boldsymbol{a}(t) = G(\boldsymbol{s}(t)).$$

The decoding of this activity is achieved using a decoding matrix $D \in \mathbb{R}^{k\times N}$, and a postsynaptic current (PSC) which acts as a convolution kernel. A common PSC used is that of a leaky integrator

$$h(t) = \frac{1}{\tau}e^{-t/\tau},$$

alongside other more complex and nonlinear kernels. The representation is given by the projection of the convolved neural activity onto the decoding matrix, such that

$$\boldsymbol{o}(t) = D(\boldsymbol{a} * h)(t),$$

which forms the representation.

Here we observe crucial similarities and differences with NEF. If we assume that the spiking function G is the activation function tanh, and define different variables for the input matrix, $B = \boldsymbol{\alpha}E$, bias term, $\boldsymbol{d} = \boldsymbol{\beta}$, neural states, $\boldsymbol{r}(t) = \boldsymbol{a}(t)$, and output matrix $W = D$, then the neural states take the form

$$\boldsymbol{r}(t) = \tanh(B\boldsymbol{x}(t) + \boldsymbol{d}),$$

with an output

$$\boldsymbol{o} = W(\boldsymbol{r} * h)(t).$$

The similarity is that the outputs eventually become a convolved and nonlinear function of the inputs, $\boldsymbol{x}$.

One of the crucial differences is that in the present work, the convolution kernel h is not defined on a per-neuron basis as is by definition the case for the PSC of a neuron. Rather, by approaching the derivation from the reservoir equation in Eq. 1, the derived kernel is a distributed kernel that takes into account the initial connectivity between the neurons, A—before any imposed wiring—of the RNN:

$$\boldsymbol{h}(t) = e^{\gamma A^* t},$$

which enables us to engineer existing networks.

This difference expands the computational capability of a neural programming framework in two ways. First, because the RNN representation in our framework allows for signal mixing through an initial connectivity A, the machine code generated by our framework will possess cross terms in the inputs even if the encoding vectors—matrix E in NEF and matrix B in this work—possess no overlapping projections. From a connectionist viewpoint, whereas the NEF supports rank k internal connections, the present framework supports full rank N internal connections. Second, because our framework allows us to decompile existing, albeit weak, initial connectivity, we increase the number of parameters that determine our representation from the kN of the input matrix, to $kN + N^2$. Hence, we can now explicitly enforce information integration induced by existing connectivity, and study this efficiency in different network topologies. A simple demonstration of this advantage is provided in the last figure of the main text, where we decompile the computation performed by a full rank reservoir that has been trained to filter an input signal. Hence, a qualitative advantage of this full-rank support is the ability to not only compile programs, but to also *decompile* programs from conventionally trained neural networks.

Principle 2: Transformation The second principle is transformation, whereby the output is reconstructed to be a function of the input, $\boldsymbol{f}(\boldsymbol{x})$, such that

$$\boldsymbol{o}(t) = D(\boldsymbol{a} * h)(t) \approx (\boldsymbol{f}(\boldsymbol{x}) * h)(t).$$

The solution is derived through a convex optimization on the decoding matrix D by sampling the space of inputs, $\boldsymbol{v} \in S \subset \mathbb{R}^k$, over some finite domain. NEF avoids explicit simulations

of the neural dynamics by taking the averaged firing rate $\boldsymbol{r}(\boldsymbol{v})$ for each sampled input. The solved D^* takes the form of the solution to an optimization, a common form of which is

$$D^* = \underset{D}{\operatorname{argmin}} \int_S \|\boldsymbol{f}(\boldsymbol{v}) - D(\boldsymbol{r}(\boldsymbol{v}) + \eta)\| \, \mathrm{d}^k \boldsymbol{v},$$

where η is white additive noise. Herein lies one of the most fundamental differences between the NEF and our present work. Whereas NEF produces the decoder as a solution to the optimization problem, we take the intermediate step of providing an analytical basis for the neural representation (Eq. 5,7), which we refer to as our programming matrix.

The immediate benefits of such a programming matrix are numerous. One benefit is the lack of any need for sampling. Whether our input space is of dimension $k = 3$ for the Lorenz attractor, or $k = 100$ for the 100-lag task, the scalability of our framework is not contingent on a sampling of $\mathbb{R}^k$ space. This benefit is especially crucial in the design of complex nonlinear simulations such as the game of pong, where the performance of our programming is not contingent on the dense sampling of game states, many of which are highly nonlinear and require formation and collapse of unstable critical points. Hence, in the present framework, the lack of need for sampling is much more reflective of computer algorithms that are defined using algebraic relationships, rather than through the dense sampling of a complex static or dynamical manifold.

An additional benefit of such a programming framework is the ability to make explicit statements about the space of possible transformations given the RNN parameters, because the programming matrix defines a *linear basis* of reconstructable nonlinear functions. If the decoder is generated by an optimization procedure, the question of whether some complex function $\boldsymbol{f}(\boldsymbol{x})$ lies within the set of reconstructable functions must be checked one function at a time. In contrast, the decoder we provide is the solution to a simple linear regression, whereby the $N \times K$ programming matrix R defines the coefficients of K Taylor series terms in the input variables across N neurons, the $m \times K$ output matrix O defines the coefficients of K Taylor series terms of $\boldsymbol{f}(\boldsymbol{x})$ in the input variables across the m outputs, and the decoding matrix W is the least-squares solution to $WR = O$. Hence, O either lies within the columnspace of R, or it does not, and the decidability of whether a function $\boldsymbol{f}(\boldsymbol{x})$ is reconstructable is simply a check for whether O projects nontrivally to the nullspace of $R^\top$.

XI. CITATION DIVERSITY STATEMENT

Recent work in several fields of science has identified a bias in citation practices such that papers from women and other minority scholars are under-cited relative to the number of such papers in the field [3–7]. Here we sought to proactively consider choosing references that reflect thediversity of the field in thought, form of contribution, gender, race, ethnicity, and other factors. First, we obtained the predicted gender of the first and last author of each reference by using databases that store the probability of a first name being carried by a woman [7, 8]. By this measure (and excluding self-citations to the first and last authors of our current paper), our references contain 6.38% woman(first)/woman(last), 10.64% man/woman, 12.77% woman/man, and 70.21% man/man. This method is limited in that a) names, pronouns, and social media profiles used to construct the databases may not, in every case, be indicative of gender identity and b) it cannot account for intersex, nonbinary, or transgender people. Second, we obtained predicted racial/ethnic category of the first and last author of each reference by databases that store the probability of a first and last name being carried by an author of color [9, 10]. By this measure (and excluding self-citations), our references contain 10.99% author of color (first)/author of color(last), 15.25% white author/author of color, 21.99% author of color/white author, and 51.76% white author/white author. This method is limited in that a) names and Florida Voter Data to make the predictions may not be indicative of racial/ethnic identity, and b) it cannot account for Indigenous and mixed-race authors, or those who may face differential biases due to the ambiguous racialization or ethnicization of their names. We look forward to future work that could help us to better understand how to support equitable practices in science.

XII. REFERENCES

[1] Sussillo, D. & Abbott, L. F. Generating coherent patterns of activity from chaotic neural networks. *Neuron* **63**, 544–557 (2009).

[2] Voelker, A. R. Dynamical systems in spiking neuromorphic hardware (2019).

[3] Mitchell, S. M., Lange, S. & Brus, H. Gendered citation patterns in international relations journals. *International Studies Perspectives* **14**, 485–492 (2013).

[4] Dion, M. L., Sumner, J. L. & Mitchell, S. M. Gendered citation patterns across political science and social science methodology fields. *Political Analysis* **26**, 312–327 (2018).

[5] Caplar, N., Tacchella, S. & Birrer, S. Quantitative evaluation of gender bias in astronomical publications from citation counts. *Nature Astronomy* **1**, 0141 (2017).

[6] Maliniak, D., Powers, R. & Walter, B. F. The gender citation gap in international relations. *International Organization* **67**, 889–922 (2013).

[7] Dworkin, J. D. *et al.* The extent and drivers of gender imbalance in neuroscience reference lists. *bioRxiv* (2020). URL `https://www.biorxiv.org/content/early/2020/01/11/2020.01.03.894378`. https://www.biorxiv.org/content/early/2020/01/11/2020.01.03.894378.full.pdf.

[8] Zhou, D. *et al.* Gender diversity statement and code notebook v1.0 (2020). URL `https://doi.org/10.5281/zenodo.3672110`.

[9] Ambekar, A., Ward, C., Mohammed, J., Male, S. & Skiena, S. Name-ethnicity classification from open sources. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge Discovery and Data Mining*, 49–58 (2009).

[10] Sood, G. & Laohaprapanon, S. Predicting race and ethnicity from the sequence of characters in a name. *arXiv preprint arXiv:1805.02109* (2018).