nature machine intelligence

Article

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

A neural machine code and programming framework for the reservoir computer

Received: 28 April 2022

Accepted: 26 April 2023

Published online: 12 June 2023

Check for updates

Jason Z. Kim[1,2] & Dani S. Bassett[3,4]

From logical reasoning to mental simulation, biological and artificial neural systems possess an incredible capacity for computation. Such neural computers offer a fundamentally novel computing paradigm by representing data continuously and processing information in a natively parallel and distributed manner. To harness this computation, prior work has developed extensive training techniques to understand existing neural networks. However, the lack of a concrete and low-level machine code for neural networks precludes us from taking full advantage of a neural computing framework. Here we provide such a machine code along with a programming framework by using a recurrent neural network—a reservoir computer—to decompile, code and compile analogue computations. By decompiling the reservoir's internal representation and dynamics into an analytic basis of its inputs, we define a low-level neural machine code that we use to program the reservoir to solve complex equations and store chaotic dynamical systems as random-access memory. We further provide a fully distributed neural implementation of software virtualization and logical circuits, and even program a playable game of pong inside of a reservoir computer. Importantly, all of these functions are programmed without requiring any example data or sampling of state space. Finally, we demonstrate that we can accurately decompile the analytic, internal representations of a full-rank reservoir computer that has been conventionally trained using data. Taken together, we define an implementation of neural computation that can both decompile computations from existing neural connectivity and compile distributed programs as new connections.

Neural systems possess an incredible capacity for computation. From biological brains that learn to manipulate numeric symbols and run mental simulations[1–3] to artificial neural networks that are trained to master complex strategy games[4,5], neural networks are outstanding computers. What makes these neural computers so compelling is that they are exceptional in different ways from modern-day silicon computers: the latter relies on binary representations, rapid sequential processing[6], and segregated memory and central processing unit[7], while the former utilizes continuum representations[8,9], parallel and distributed processing[10,11], and distributed memory[12]. To harness these distinct computational abilities, prior work has studied a vast array of different network architectures[13,14], learning algorithms[15,16] and information-theoretic frameworks[17–19] in both biological and artificial neural networks. Despite these substantial advances, the relationship between neural computers and modern-day silicon computers remains an analogy due to our lack of a concrete and low-level neural machine code, thereby limiting our access to neural computation.

[1]Department of Bioengineering, University of Pennsylvania, Philadelphia, PA, USA. [2]Department of Physics, Cornell University, Ithaca, NY, USA. [3]Departments of Bioengineering, Physics & Astronomy, Electrical & Systems Engineering, Neurology, and Psychiatry, University of Pennsylvania, Philadelphia, PA, USA. [4]Santa Fe Institute, Santa Fe, NM, USA. e-mail: dsb@seas.upenn.edu

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

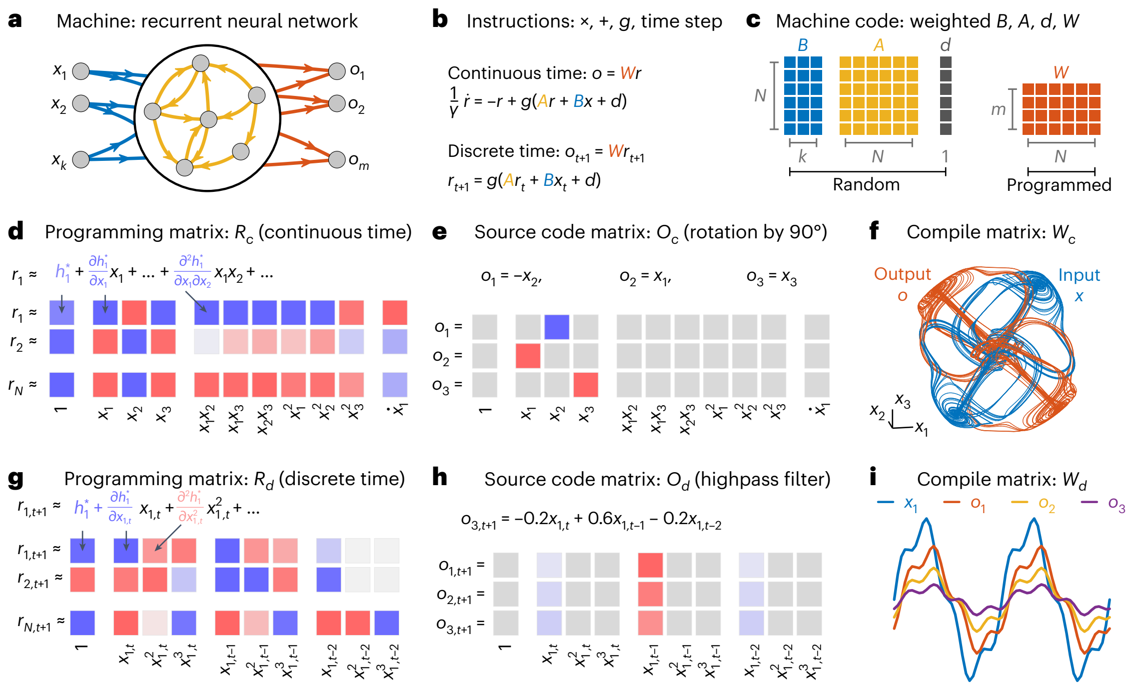

Fig. 1 | Open-loop neural computer architecture for SNP. a, An RNN conceptualized as a computing machine with inputs $\boldsymbol{x}$, neurons $\boldsymbol{r}$ and outputs $\boldsymbol{o}$, which are all treated as variables. **b**, The low-level instructions supported by this RNN. **c**, We randomly instantiate B, A and $\boldsymbol{d}$, and compile only the output matrix W. **d**, To convert the machine code into an algebraic form (that is, as a function of $\boldsymbol{x}$), we decompile the machine code into the SNP matrix R_c by first approximating the RNN state as a function of the powers and the time derivatives of the inputs $\boldsymbol{x}$ (equations (8)–(10)), and then by taking the Taylor series expansion of $\boldsymbol{h}$. Hence, the (i, j) term of R_c corresponds to the Taylor series coefficient of neuron i and term j. **e**, This SNP matrix is used to program a desired output (for example, a rotation about the x_3 axis) by filling in the entries of the source code matrix O_c, which correspond to the coefficients in front of the desired output functions. **f**, This program is compiled by training an output matrix W_c that maps R_c to O_c (equation (1)). The RNN output, $W_c\boldsymbol{r}(t)$, successfully rotates the three-dimensional Thomas attractor input $\boldsymbol{x}(t)$. **g**, The discrete-time decompiler is also composed of a Taylor series expansion, but the function $\boldsymbol{h}$ is now from equations (11) and (12), and the time derivative terms are replaced by time lags. **h**,**i**, We program highpass filters of various cut-off frequencies by weighting the appropriate time-lagged terms (**h**), which yields high-passed versions of the inputs (**i**).

To bring this analogy to reality, we seek a neural network with a simple set of governing equations that demonstrates many computer-like capabilities[20]. One such network is a reservoir computer (RC), which is a recurrent neural network (RNN) that receives inputs, evolves a set of internal states forward in time and outputs a weighted sum of its states[21,22]. True to its namesake, RCs can be trained to perform fundamental computing functions such as memory storage[23,24] and manipulation[25,26], prediction of chaotic systems[22] and model-free control[27]. Owing to the simplicity of the governing equations, the theoretical mechanism of training is understood, and recent advances have dramatically shortened training requirements by using a more efficient and expanded set of input training data[28]. But can we skip the training altogether and program RCs without any sampling or simulation just as we do for silicon computers?

These ideas have a rich history of exploration, notably including the Neural Engineering Framework[29] and system hierarchies for brain-inspired computing[30]. The former defines the guiding principles—representation, transformation and dynamics—to implement complex computations and dynamics in neuron models[31]. The latter defines an extension of Turing completeness to neuromorphic completeness, and builds an interface between neuromorphic software and hardware for program-level portability. Our work sits at the intersection of these two areas by defining an extension of the former for RCs to program computations in existing RCs with full-rank connectivity, decompile the computations performed by conventionally trained RCs and omit any sampling of state space in the optimization procedure (Supplementary Section X). Combined with the substantial advances in experimental RC platforms[32], it is now timely to formalize a programming framework to develop neural software atop RC hardware.

In this Article, we provide two such programming frameworks—state neural programming (SNP) and dynamic neural programming (DNP)—by constructing an analytic representational basis of the RC neurons, alongside two architectures: open loop and closed loop. Through SNP, we program RCs to perform operations and solve analytic equations. Through DNP, we program RCs to store chaotic dynamical systems as random-access memories (dRAM), virtualize the dynamics of a guest RC, implement neural logic AND, NAND, OR, NOR, XOR and XNOR, and construct neural logic circuits such as a binary adder, flip-flop latch and multivibrator circuit. Using these circuits, we define a simple scheme for game development on RC architectures by programming an RC to simulate a variant of the game 'pong'. Finally, we decompile computations from conventionally trained RCs as analytic functions.

Open-loop architecture with SNP for output functions

We begin with the simplest architecture, which is an open-loop neural computer architecture that treats the RNN as a function approximator. We conceptualize an RNN as a computing machine comprising N neurons $\boldsymbol{r}$, which receive k inputs $\boldsymbol{x}$ and produce m outputs $\boldsymbol{o}$ (Fig. 1a). This machine typically supports the basic instructions of multiplication of the neural states and inputs by weights A, B and W, the addition of these weighted states with each other and some bias term $\boldsymbol{d}$, the transformation of the resultant quantities through an activation function g, and evolution in time (equations (6) and (7)). The output is given by $\boldsymbol{o} = W\boldsymbol{r}$ (Fig. 1b). Hence, the weights B, A, $\boldsymbol{d}$ and W specify the set of instructions for the RNN to run, which we conceptualize as the low-level neural machine code (Fig. 1c). Unlike conventional computers, these

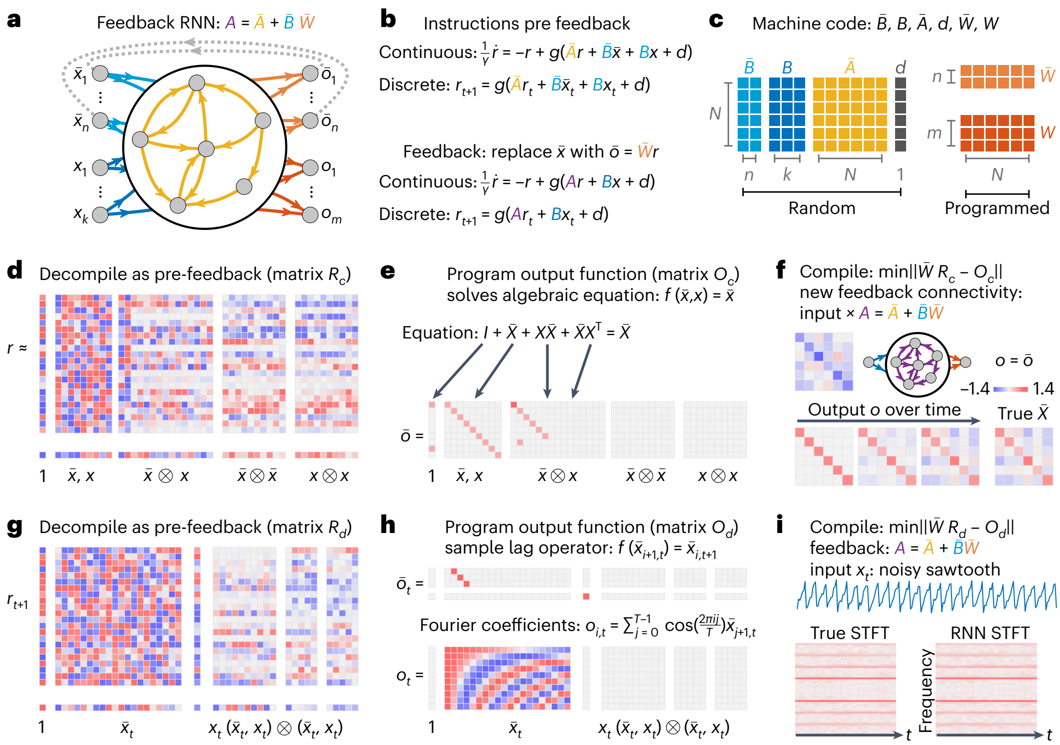

Fig. 2 | Closed-loop neural computer architecture for SNP. **a**, Schematic of the closed-loop architecture with two input sets, $\bar{\boldsymbol{x}}$ and $\boldsymbol{x}$, and two output sets, $\bar{\boldsymbol{o}}$ and $\boldsymbol{o}$, where feedback occurs by setting $\bar{\boldsymbol{x}} = \bar{\boldsymbol{o}}$. **b**, Instruction before and after feedback for continuous-time and discrete-time RNNs. **c**, $\bar{B}$, B, $\bar{A}$ and $\boldsymbol{d}$ are randomly initialized, while $\bar{W}$ and W are programmed. **d**, A scaled subset of the decompiled matrix R_c containing constant, linear, and quadratic terms in the inputs. **e**, These terms are used to program the left-hand side of the Lyapunov equation into the source code matrix O_c. **f**, After compiling O_c into W and performing feedback, a new RNN is defined with recurrent connections $A = \bar{A} + \bar{B}\bar{W}$, and when this RNN is driven with input matrix X, its output converges to the solution of the Lyapunov equation. Here, we set $W = \bar{W}$ such that output $\boldsymbol{o}$ matched the feedback output $\bar{\boldsymbol{o}}$. **g**, Decompiled R_d for the discrete-time RNN. **h**, A sample lag operator is programmed for the output used for feedback, $\bar{\boldsymbol{o}}$, and a Fourier transformation is programmed into the output that is not fed back, $\boldsymbol{o}$. **i**, When both outputs are compiled and $\bar{W}$ is fed back, then driving the resultant RNN with a noisy sawtooth successfully lags the input x_t and generates the short-time Fourier transform of the input.

instructions are simultaneously evaluated in parallel by the global set of neurons. We randomly instantiate B, A and $\boldsymbol{d}$, and program W.

To define a programming framework, we take the approach of translating the machine code into a representation that is meaningful to the user. We choose that representation to be the input variables $\boldsymbol{x}$, as the inputs are usually meaningful and interpretable in most applications. This translation from the low-level to the high-level programming matrix is referred to as decompiling, and involves writing the neural states $\boldsymbol{r}$ as a function $\boldsymbol{h}$ of the inputs $\boldsymbol{x}$ given the machine code B, A and $\boldsymbol{d}$ (Methods, equations (8)–(10)). To write $\boldsymbol{h}$ in a more understandable form, we perform a Taylor series expansion of $\boldsymbol{h}$ with respect to all of the input variables $\boldsymbol{x}, \dot{\boldsymbol{x}}, \cdots$, thereby writing the state of every neuron as a weighted sum of linear, quadratic and higher-order polynomial terms of $\boldsymbol{x}, \dot{\boldsymbol{x}}, \cdots$. The weights that multiply these terms are precisely the coefficients of the Taylor series expansion, and form an $N \times K$ matrix of coefficients R_c, where K is the number of terms in the expansion (Fig. 1d). This matrix of coefficients R is our SNP matrix in the open-loop architecture.

Next, we define the source code, which is the set of programmable output functions given by the rowspace of R. This is because the output of our RNN is determined by a linear combination of neural states, W, which are weighted sums of the K expansion terms from the decompiler. Here, programming refers to specifying m outputs as a weighted sum of the K terms from the decompiler, which forms an $m \times K$ matrix O_c: the source code matrix. In this example, the RNN receives three inputs: x_1, x_2 and x_3. To program a 90° rotation about x_3, we specify the coefficients in matrix O_c for three outputs o_1, o_2 and o_3 (Fig. 1e). Finally, to compile the source code O_c into machine code W, we solve

$$W = \underset{W}{\operatorname{argmin}} \parallel WR - O \parallel . \tag{1}$$

When we drive the RNN with the complex, chaotic time series $\boldsymbol{x}(t)$ from the Thomas attractor (Fig. 1f, blue), the RNN output is a rotated attractor $W\boldsymbol{r}(t)$ (Fig. 1f, orange).

This process of decompiling, programming and compiling also holds in discrete-time RNNs (Fig. 1g; see Methods, equations (11) and (12)). Our programming matrix now consists of polynomial powers of time-lagged inputs, which allows us to program operations such as highpass filters (Fig. 1h). After compiling the program O_d into W, the RNN outputs (Fig. 1i, orange, yellow and purple) filter away the lower-frequency components of an input signal (Fig. 1i, blue). See Supplementary Section IX for the parameters used for all examples.

Closed-loop architecture with SNP to solve algorithms

To increase the computational power of these RNNs, we use the same SNP, but introduce a closed-loop neural architecture. The idea is to use SNP to program some output function of the inputs as $\bar{W}\boldsymbol{r} = \boldsymbol{f}(\bar{\boldsymbol{x}}, \boldsymbol{x})$, and then feed that output back into the inputs to define an equivalence relation between the outputs and the inputs to solve $\bar{\boldsymbol{x}} = \boldsymbol{f}(\bar{\boldsymbol{x}}, \boldsymbol{x})$. This feedback modifies the internal recurrent weights in the RNN, allowing it to store and modify a history of its states and inputs to store and run algorithms. Hence, the closed-loop architecture is no longer solely a function approximator. We consider the same RNN as in Fig. 1a, except now with two sets of inputs, $\bar{\boldsymbol{x}} \in \mathbb{R}^n$ and $\boldsymbol{x} \in \mathbb{R}^k$, and two sets of outputs,

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

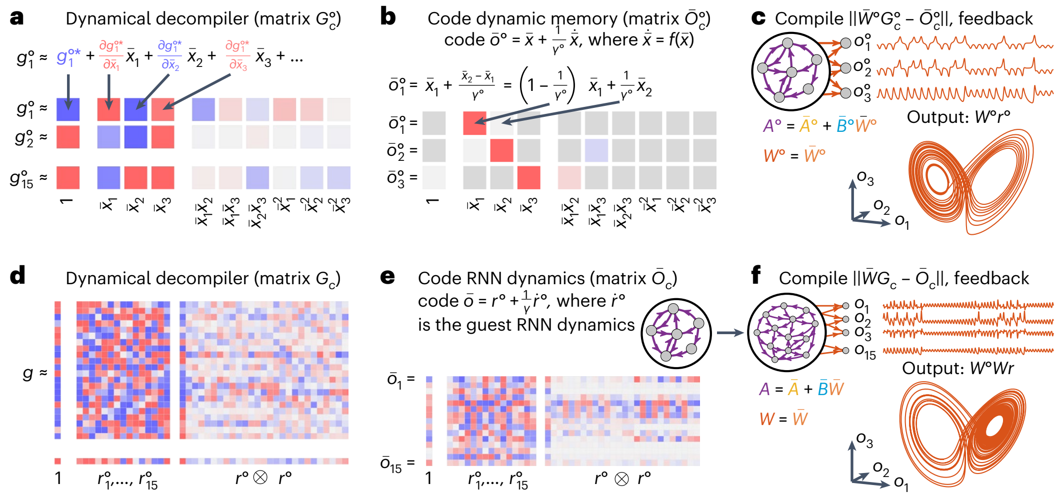

Fig. 3 | Simulation and virtualization using DNP. a, The machine code ($\bar{B}^\circ$, $\boldsymbol{d}^\circ$) of the guest RNN with 14 neurons and 3 inputs is decompiled into DNP matrix G_c° by taking the Taylor series coefficients of $\boldsymbol{g}^\circ$ as equation (3). **b**, Therefore, the programs we write are not output functions of $\bar{\boldsymbol{x}}$, but are rather given by $\bar{\boldsymbol{x}} + \frac{1}{\gamma^\circ}\dot{\bar{\boldsymbol{x}}}$, where $\dot{\bar{\boldsymbol{x}}} = f(\bar{\boldsymbol{x}})$ for a dynamical system. **c**, After compiling into $\bar{W}^\circ$ and performing feedback, the evolution of the guest RNN projects onto the programmed Lorenz dynamical system. **d**, The machine code ($\bar{B}$, $\boldsymbol{d}$) of the 2,000 neuron host RNN with 15 inputs is decompiled into DNP matrix G_c. **e**, The program is the set of coefficients for the state and dynamics $\boldsymbol{r}^\circ + \frac{1}{\gamma}\dot{\boldsymbol{r}}^\circ$ of the guest RNN. **f**, After compiling and feedback, the time evolution of the host RNN emulates the time-evolution of the guest RNN, which itself is simulating a Lorenz attractor.

$\bar{\boldsymbol{o}} \in \mathbb{R}^n$ and $\boldsymbol{o} \in \mathbb{R}^m$ (Fig. 2a and equation (13)). We feed one set of outputs back into the inputs such that $\bar{\boldsymbol{o}} = \bar{\boldsymbol{x}}$, which will determine the internal connectivity of the RNN. Using SNP, we program an output $\bar{\boldsymbol{o}} = \bar{W}\boldsymbol{r} = \bar{\boldsymbol{f}}(\bar{\boldsymbol{x}},\boldsymbol{x})$ and perform feedback as $A = \bar{A} + \bar{B}\bar{W}$ (Fig. 2b,c and equation (14)).

As a demonstration, we program an RNN to solve the continuous Lyapunov equation,

$$I + \bar{X} + X\bar{X} + \bar{X}X^{\mathrm{T}} = \bar{X}, \tag{2}$$

where X and $\bar{X}$ are 6×6 matrices such that the pre-programmed RNN receives $n = 36$ inputs as $\bar{\boldsymbol{x}}$ and $k = 36$ inputs as $\boldsymbol{x}$. Given X, the solution $\bar{X}$ to equation (2) is important for control theory[33] and neuroscience[34], and is often referred to as the controllability Gramian. To program equation (2) into our RNN, we first decompile the neural states $\boldsymbol{r}$ into the SNP matrix R_c with respect to our input variables $\bar{\boldsymbol{x}}$ and $\boldsymbol{x}$ (Fig. 2d). Next, we fill in matrix O_c with the coefficients of all constant, linear and quadratic terms from equation (2) (Fig. 2e). Then, we compile the program O_c by solving for $\mathrm{argmin}_{\bar{W}} \parallel \bar{W}R_c - O_c \parallel$, and define the recurrent connections of a new feedback RNN, $A = \bar{A} + \bar{B}\bar{W}$, which evolves as equation (14). By driving this new RNN with a matrix X, the output converges to the solution $\bar{X}$ of equation (2) (Fig. 2f).

As a demonstration for discrete-time systems, we program an RNN to store a substantial time history of a stochastic, non-differentiable time series x_t, and perform a short-time Fourier transform. Starting with our decompiled RNN states (Fig. 2g), we write a program, $\bar{O}_d$, to store time history across $n = 49$ inputs for $\bar{\boldsymbol{x}}_t$, by defining a sample lag operator that shifts the state of all inputs down by one index (Fig. 2h and equation (15)). Then, using this lagged history, we write another program, O_d, that outputs a short-time Fourier transform (equation (16)) with a sliding window of length n. We compile these two programs by minimizing $\parallel \bar{W}R_d - \bar{O}_d \parallel$ and $\| WR_d - O_d \|$, and define the recurrent connectivity of a new feedback RNN—where $A = \bar{A} + \bar{B}\bar{W}$—according to equation (14). By driving this RNN with a stochastic sawtooth wave of amplitude 0.2, period of eight samples and noise that is uniformly distributed about 0 with a width of 0.1, the RNN matches the true short-time Fourier transform (for a performance comparison with conventional RCs and FORCE, see Supplementary Section VII).

Closed-loop RNN with DNP to simulate and virtualize

Here we define a second, dynamical programming framework, DNP, which allows explicit programming of time history for continuous-time RNNs (for an extended discussion, see Supplementary Section VIII). Building on SNP where we decompiled the neural states $\boldsymbol{r}$, we now decompile the activation function $\boldsymbol{g}$, which encodes both state and dynamic information through a rearrangement of equation (6). We substitute $\boldsymbol{r} \approx \boldsymbol{h}(\bar{\boldsymbol{x}},\boldsymbol{x})$ into equation (13) as

$$\boldsymbol{r} + \frac{1}{\gamma}\dot{\boldsymbol{r}} = \boldsymbol{g}(A\boldsymbol{h}(\bar{\boldsymbol{x}},\boldsymbol{x}) + \bar{B}\bar{\boldsymbol{x}} + B\boldsymbol{x} + \boldsymbol{d}). \tag{3}$$

Hence, our DNP decompiler takes the Taylor series coefficients of $\boldsymbol{g}$ instead of $\boldsymbol{h}$, allowing us to program not only output functions of the input states, but also the input time history.

As a demonstration, we consider an RNN with 15 states, $\boldsymbol{r}^\circ \in \mathbb{R}^{15}$, which receives three inputs $\bar{\boldsymbol{x}} \in \mathbb{R}^3$. We will use the closed-loop architecture where $\bar{A}^\circ$, $\bar{B}^\circ$ and $\boldsymbol{d}^\circ$ are randomly initialized, which is decompiled according to equation (3) (Fig. 3a). Because our decompiled code consists of analytic state and time-derivative variables, we can compile $\bar{W}^\circ$ to map RNN states to input states, and RNN time derivatives to input time derivatives. Prior work has trained RNNs to simulate time-evolving systems by copying exemplars[22] or sampling the dynamical state space[35]. Here we achieve the same simulation without any samples in a chaotic Lorenz attractor that evolves according to $\dot{\bar{\boldsymbol{x}}} = f(\bar{\boldsymbol{x}})$, such that

$$\bar{W}\boldsymbol{g}(\bar{\boldsymbol{x}}) = \bar{W}\left(\boldsymbol{r} + \frac{1}{\gamma}\dot{\boldsymbol{r}}\right) = \bar{\boldsymbol{x}} + \frac{1}{\gamma}\dot{\bar{\boldsymbol{x}}} = \bar{\boldsymbol{x}} + \frac{1}{\gamma}f(\bar{\boldsymbol{x}}). \tag{4}$$

Here, 'programming' refers to the construction of a matrix $\bar{O}_c^\circ$ comprising the coefficients of $\bar{\boldsymbol{x}} + \frac{1}{\gamma}f(\bar{\boldsymbol{x}})$ preceding the variables $1, \bar{x}_1, \bar{x}_2, \bar{x}_3, \bar{x}_1\bar{x}_2, \cdots$ of the program matrix G_c° (Fig. 3b). Once we compile this code and perform feedback by defining new connectivity $A = \bar{A} + \bar{B}\bar{W}$, the evolution of the RNN simulates the Lorenz attractor (Fig. 3c).

More generally, DNP allows us to program systems of the form $\dot{\bar{\boldsymbol{x}}} = f(\bar{\boldsymbol{x}})$, which raises an interesting phenomenon. We program another

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

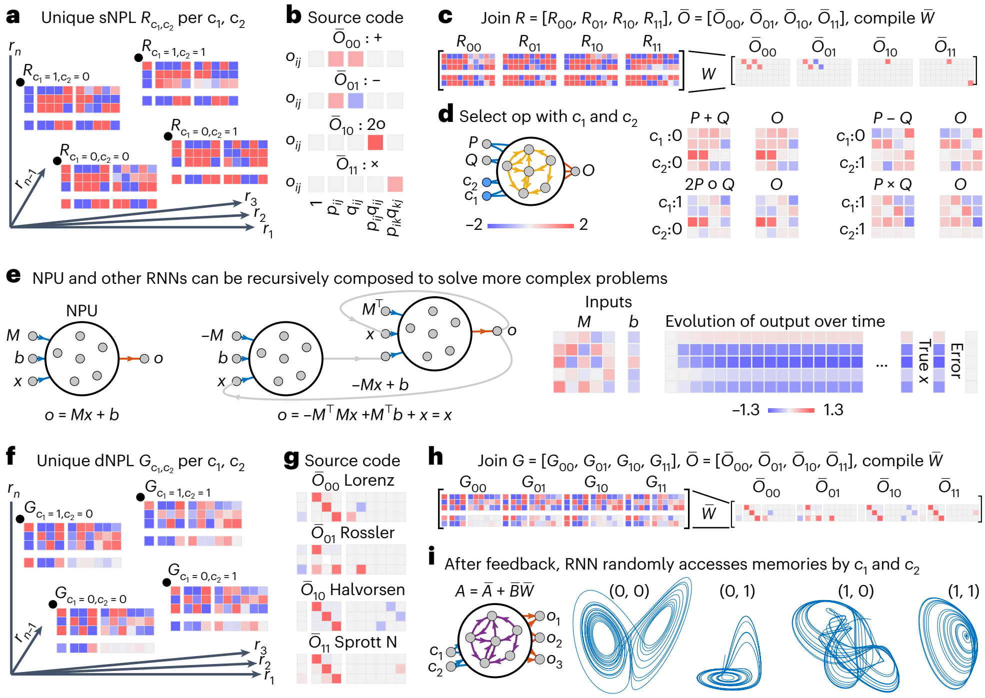

Fig. 4 | Extensions to op-codes, composition and dynamic RAM. a, Biasing an RNN with different control inputs c_1, c_2 produces a unique SNP R_{c_1,c_2}. **b**, Source code for programming four matrix operations: elementwise addition, subtraction and multiplication, and matrix multiplication. **c**, The code is simultaneously compiled by concatenating all SNPs into R and all source codes into $\bar{O}$, and solving $\mathrm{argmin}_{\bar{W}} \parallel \bar{W}R - \bar{O} \parallel$. **d**, The RNN is driven by two matrices, P and Q, while switching the op-code c_1, c_2 to yield the desired operation. **e**, A neural processing unit (NPU) that performs matrix multiplication and vector addition can be connected to another NPU such that their composition yields a more complex algebraic equation. Here the equation is for least-squares regression, and the programmed RNN composition evolves forward in time to solve the least-squares problem. **f**, Biasing an RNN with different control inputs c_1, c_2 produces a unique DNP G_{c_1,c_2}. **g**, Source code for different chaotic attractors. **h**, The code is simultaneously compiled by concatenating all DNPs into G, and all source codes into $\bar{O}$, and solving $\mathrm{argmin}_{\bar{W}} \parallel \bar{W}G - \bar{O} \parallel$. **i**, By changing c_1 and c_2, we retrieve the stored memories, thereby forming our dRAM.

RNN $\boldsymbol{r} \in \mathbb{R}^{2000}$—the host—to emulate the dynamics of the feedback RNN $\boldsymbol{r}^{\circ} \in \mathbb{R}^{15}$—the guest—that itself was programmed to evolve about the Lorenz attractor. We decompile the host RNN using DNP (Fig. 3d), write the code of the guest RNN in the format $\boldsymbol{r}^{\circ} + \frac{1}{\gamma}\dot{\boldsymbol{r}}^{\circ}$ (Fig. 3e), and compile the code into matrix $\bar{W}$ (Fig. 3f). The 2,000 state host RNN emulates the 15 state guest RNN, which is simulating a Lorenz attractor. A larger host can emulate multiple guests as a virtual machine[36].

Op-codes, composition and dynamic RAM

Here we extend more of the functionality of general purpose computers to RNNs. The first functionality is support for op-codes, which is typically a string of bits specifying which instruction to run. We add control inputs $\boldsymbol{c}$ as a string of 0s and 1s such that

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

which pushes the pre-programmed RNN to different fixed points, thereby generating a unique SNP at each point (Fig. 4a). Then, we program different matrix operations (Fig. 4b), and simultaneously compile each source code at a different SNP (Fig. 4c) into matrix $\bar{W}$. When we drive our RNN with matrices P and Q at different $\boldsymbol{c}$, the RNN outputs each operation (Fig. 4d).

The second functionality is the ability to compose more complicated programs from simpler programs. We note that, in SNP, the output is programmed and compiled to perform an operation on the inputs, such as a matrix multiplication and vector addition for a neural processing unit (NPU, Fig. 4e). By feeding these outputs into another NPU, we can perform a successive series of feedback operations to define and solve more complex equations, such as least-squares regression (NPU, Fig. 4e).

The third functionality is the random access of chaotic dynamical memories. The control inputs $\boldsymbol{c}$ drive the RNN to different fixed points, thereby generating unique DNPs G_{c_1,c_2} (Fig. 4f). By compiling a single matrix W that maps each DNP to a unique attractor (Fig. 4g,h), the feedback RNN with internal connectivity $A = \bar{A} + \bar{B}\bar{W}$ autonomously evolves about each of the four chaotic attractors at different values of $\boldsymbol{c}$ (Fig. 4i).

A logical calculus using recurrent neural circuits

This dynamical programming framework allows us to greatly expand the computational capability of our RNNs by programming neural implementations of logic gates. While prior work has established the ability of biological and artificial networks to perform computations, here we provide an implementation that makes full use of existing computing frameworks. We program logic gates into distributed RNNs by using a simple dynamical system

$$\dot{x} = ax^3 + bx + z, \tag{5}$$

where a, b and z are parameters. This particular system has the nice property of hysteresis, where when $z = 0.1$, the value of x converges to $x = 0.1$, but when $z = -0.1$, the value of x jumps discontinuously to converge at $x = -0.1$ (Fig. 5a). This property enables us to program logic gates (Fig. 5b). Specifically, by defining the variable z as a product of two input variables p and q, we can program in the dynamics in equation (5) to evolve to −0.1 or 0.1 for different patterns of p and q.

These logic gates can now take full advantage of existing computing frameworks. For example, we can construct a full adder using neural circuits that take Boolean values p and q as the two numbers to be added, and a 'carry' value from the previous addition operation. The adder outputs the sum s and the output carry v. We show the inputs and

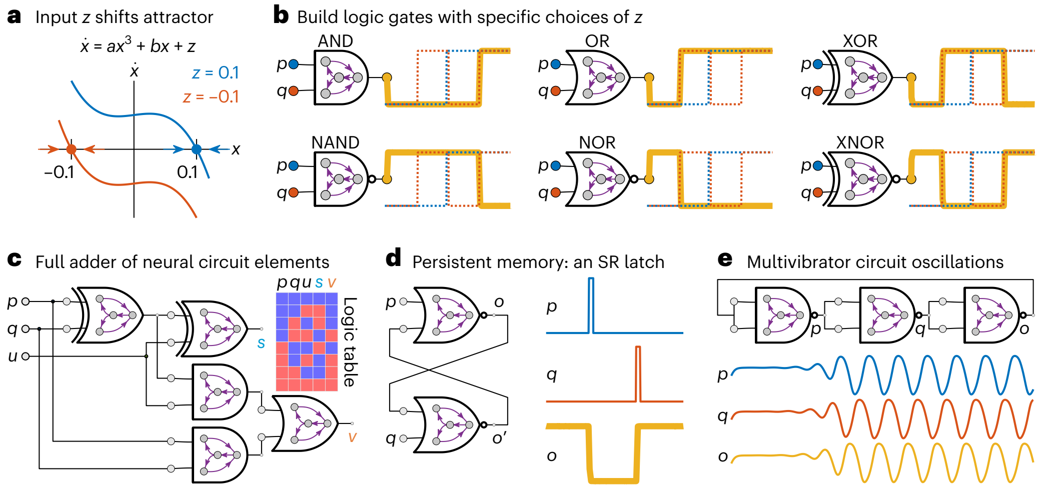

Fig. 5 | Programming logic gates and circuits using dynamical neural networks. a, Phase diagram of a cubic dynamical system. When $z = 0.1$, the variable x tends towards the stable fixed point $x^* = 0.1$. When $z = -0.1$, the system bifurcates, and a new stable fixed point emerges at $x^* = -0.1$. **b**, By setting z equal to various products of two input variables p and q, the output evolves according to different Boolean logic gates, and we program these logic gate dynamics into our RNNs. **c**–**e**, By connecting these neural logic gates, we can form neural circuits that add Boolean numbers (**c**), store persistent Boolean states according to a SR latch with output o and hidden variable o' (**d**), and oscillate at a fixed phase difference due to the propagation delay of inversion operations (**e**).

outputs of a fully neural adder in Fig. 5c, forming the basis of our ability to program neural logic units, which are neural analogues of existing arithmetic logic units.

The emulation of these neural logic gates to circuit design extends even to recurrent circuit architectures. For example, the set–reset (SR) latch—commonly referred to as a flip-flop—is a circuit that holds persistent memory, and is used extensively in computer random-access memory (RAM). We construct a neural SR latch using two NOR gates with two inputs, p and q (Fig. 5d). When $p = 0.1$ is pulsed high, the output $o = -0.1$ changes to low. When q is pulsed high, the output changes to high. When both p and q are held low, then the output is fixed at its most recent value (Fig. 5d). As another example, we can chain an odd number of inverting gates (that is, NAND, NOR and XOR) to construct a multivibrator circuit that generates oscillations (Fig. 5e). Because the output of each gate will be the inverse of its input, if p is high, then q is low and o is high. However, if we use o as the input to the first gate, then p must switch to low. This discrepancy produces constant fluctuations in the states of p, q and o, which generate oscillations that are offset by the same phase (Fig. 5e).

Game development and decompiling trained RNNs

To demonstrate the flexibility and capacity of our framework, we program a variant of the game 'pong' into our RNN as a dynamical system. We begin with the game design by defining the relevant variables and behaviours (Fig. 6a). The variables are the coordinates of the ball, x, y, the velocity of the ball, $\dot{x}$, $\dot{y}$, and the position of the paddle, x_p. Additionally, we have the variables that determine contact with the left, right and upper walls as c_l, c_r and c_u, respectively, and the variable that determines contact with the paddle, c_p. The behaviour that we want is for the ball to travel with a constant velocity until it hits either a wall or the paddle, at which point the ball should reverse direction.

Here we run into our first problem: how do we represent contact detection—a fundamentally discontinuous behaviour—using continuous dynamics? Recall that we have already done so to program logic gates in Fig. 5a by using the bifurcation of the cubic dynamical system in equation (5). Here we will use the exact same equation, except rather than changing the parameter z to shift the dynamics up and down (Fig. 5a), we will set the parameter b to skew the shape. As an example, for the right-wall contact c_r, we will let $b = x - x_r$ (Fig. 6b). When the ball is to the left such that $x < x_r$, then c_r approaches 0. When the ball is to the right such that $x > x_r$, then c_r becomes non-zero. To set the velocity of the ball, we use the SR latch developed in Fig. 5d. When neither wall is in contact, then c_r and c_l are both low, and the latch's output does not change. When either the right or the left wall is in contact, then either c_r or c_l pulses the latch, producing a shift in the velocity (Fig. 6c). Combining these dynamical equations together produces the code for our pong program (Fig. 6d), and the time evolution of our programmed RNN simulates a game of pong (Fig. 6e).

To demonstrate the capacity of our programming framework beyond compiling programs, we decompile the internal representation of a reservoir that has been trained to perform an operation. We instantiate a reservoir with random input matrix B and random recurrent matrix A, drive the reservoir with a sum of sinusoids—thereby generating a time series $\boldsymbol{r}(t)$—and train the output weights W to reconstruct highpass-filtered versions of the inputs (Fig. 6f). To understand what the reservoir has learned, we decompile the reservoir state given A and B into the SNP matrix R_d according to equation (12), and find that the output WR_d as an analytic function of the inputs and time derivatives closely matches the true filter coefficients (Fig. 6g).

Discussion

Neural computation exists simultaneously at the core of and the intersection between many fields of study. From the differential binding of neurexins in molecular biology[37] and neural circuits in neuroscience[38–42], to the RNNs in dynamical systems[43] and neural replicas of computer architectures in machine learning[44], the analogy between neural and silicon computers has generated growing and interdisciplinary interest. Our work provides one possible realization of this analogy by defining a dynamical programming framework for RNNs that takes full advantage of their natively continuous and analytic representation, their parallel and distributed processing, and their dynamical evolution.

This work also makes an important contribution to the increasing interest in alternative computation. A clear example is the vast array of systems—such as molecular[45], DNA[46] and single photon[47]—that implement Boolean logic gates. Other examples include the design of materials that compute[48,49] and store memories[50,51]. Perhaps of most relevance are physical instantiations of reservoir computing in electronic, photonic, mechanical and biological systems[32]. Our work demonstrates the potential of alternative computing frameworks to be fully programmable, thereby shifting paradigms away from imitating silicon computer hardware[6], and towards defining native programming frameworks that bring out the full computational capability of each system.

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

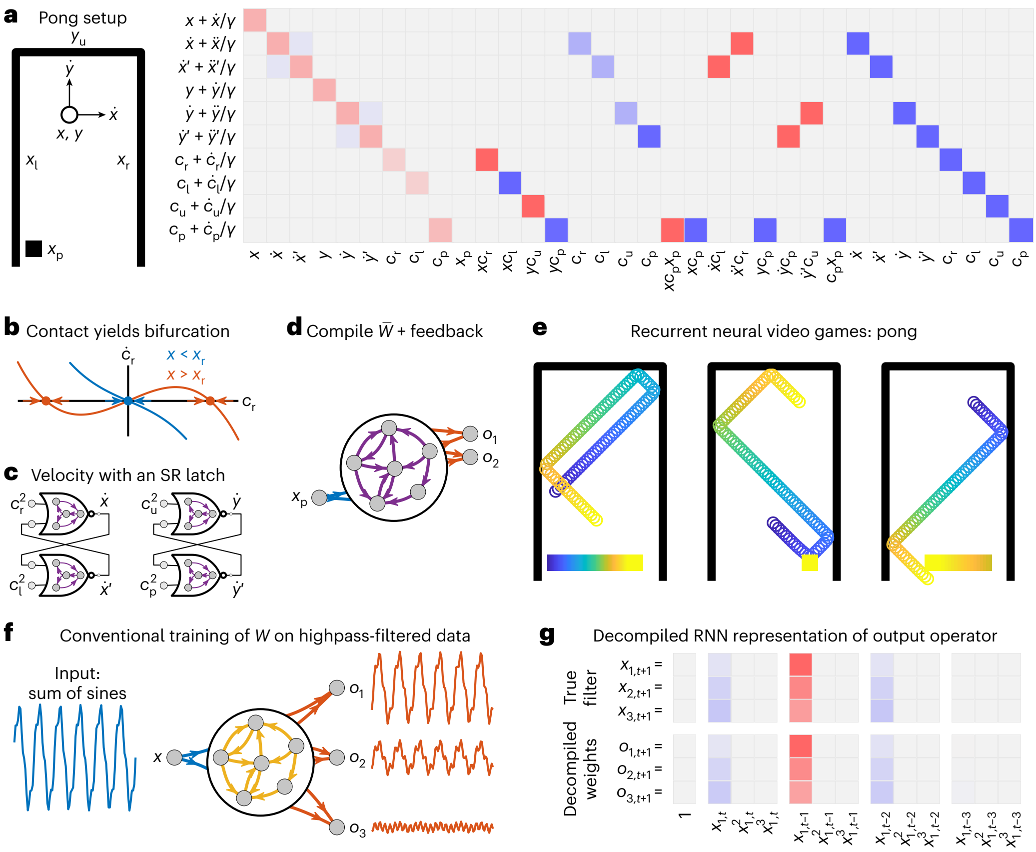

Fig. 6 | Programming pong using neural circuits and bifurcations. a, Design of a pong variant. The wall positions (x_l, x_r, y_u) and the paddle's y-coordinate are fixed as constants. The variables are the ball's position (x, y) and velocity ($\dot{x}$, $\dot{y}$), the paddle's position (x_p), and the variables determining contact with the walls and paddle (c_l, c_r, c_u, c_p). The code matrix $\bar{O}_c$ (scaled for visualization) is shown. **b**, Contact detection with the right wall is implemented using a supercritical pitchfork bifurcation by scaling the b term in equation (5) by $x - x_r$. When $x < x_r$, the contact variable c_r goes to 0. When $x > x_r$, a bifurcation occurs and c_r becomes non-zero. **c**, These contact variables are used to drive an SR latch whose output is the ball's velocity. **d**, An RNN simulating a playable game of pong in its head. **e**, The colour from blue to yellow represents the evolution of time. The bottom square is the movement of the paddle, and the circle is the movement of the marker. **f**, Conventional training of a reservoir by first driving it with an input time series to generate the reservoir time series $\boldsymbol{r}(t)$, and then training an output matrix W to reconstruct highpass-filtered versions of the input. **g**, The decompiled analytic outputs of the trained reservoir closely match the true highpass filter coefficients.

One of the main current limitations is the linear approximation of the RC dynamics. While prior work demonstrates substantial computational ability for RCs with largely fluctuating dynamics (that is, computation at the edge of chaos[52]), the approximation used in this work requires that the RC states stay reasonably close to the operating points. While we are able to program a single RC at multiple operating points that are far apart, the linearization is a prominent limitation. Future extensions would use more advanced dynamical approximations into the bilinear regime using Volterra kernels[53] or Koopman composition operators[54] to better capture non-linear behaviours.

Finally, we report in the Supplementary Section XI an analysis of the gender and the racial makeup of the authors we cited in a Citation Diversity Statement.

Methods

Open-loop architecture with SNP

In our framework, we conceptualize an RNN comprising N neurons $\boldsymbol{r} \in \mathbb{R}^N$, which receive k inputs $\boldsymbol{x} \in \mathbb{R}^k$ and produce m outputs $\boldsymbol{o} \in \mathbb{R}^m$. This machine has weights $A \in \mathbb{R}^{N\times N}$, $B \in \mathbb{R}^{N\times k}$, and $W \in \mathbb{R}^{m\times N}$, and some bias term $\boldsymbol{d} \in \mathbb{R}^{N\times 1}$. If the RNN evolves in continuous time, the instructions are

$$\frac{1}{\gamma}\dot{\boldsymbol{r}}(t) = -\boldsymbol{r}(t) + \boldsymbol{g}(A\boldsymbol{r}(t) + B\boldsymbol{x}(t) + \boldsymbol{d}), \tag{6}$$

where $1/\gamma$ is a time constant. If the RNN evolves in discrete time, these instructions are

$$\boldsymbol{r}_{t+1} = \boldsymbol{g}(A\boldsymbol{r}_t + B\boldsymbol{x}_t + \boldsymbol{d}). \tag{7}$$

We decompile the neural states $\boldsymbol{r}$ as a function $\boldsymbol{h}$ of the inputs $\boldsymbol{x}$ given the machine code B, A and $\boldsymbol{d}$ in three steps. First, we linearize the dynamics in equation (6) about a stable fixed point $\boldsymbol{r}^*$ and an operating point $\boldsymbol{x}^*$ to yield

$$\frac{1}{\gamma}\dot{\boldsymbol{r}}(t) \approx A^*\boldsymbol{r}(t) + \underbrace{\boldsymbol{g}(A\boldsymbol{r}^* + B\boldsymbol{x}(t) + \boldsymbol{d}) - \mathrm{d}\boldsymbol{g}(A\boldsymbol{r}^* + B\boldsymbol{x}(t) + \boldsymbol{d}) \circ A\boldsymbol{r}^*}_{\boldsymbol{u}(\boldsymbol{x}(t))}, \tag{8}$$

where $A^* = (\mathrm{d}\boldsymbol{g}(A\boldsymbol{r}^* + B\boldsymbol{x}^* + \boldsymbol{d}) \circ A - I)$. Second, because our system is now linear, we can write the neural states as the convolution of the impulse response and the inputs as

$$\boldsymbol{r}(t) \approx \gamma \int_{-\infty}^{t} e^{\gamma A^*(t-\tau)}\boldsymbol{u}(\boldsymbol{x}(\tau))\mathrm{d}\tau. \tag{9}$$

Third, to obtain $\boldsymbol{r}(t)$ as an algebraic function without an integral, we perform a Taylor series expansion of this convolution with respect to t to yield

$$\boldsymbol{r}(t) \approx \boldsymbol{h}(\boldsymbol{x}(t), \dot{\boldsymbol{x}}(t), \ddot{\boldsymbol{x}}(t), \cdots). \tag{10}$$

We provide a detailed analytical derivation of $\boldsymbol{h}$ in the Supplementary Sections I–III, and demonstrate the goodness of the approximations in Supplementary Sections IV–VI.

To decompile discrete-time RNNs, first we linearize equation (7) about a stable fixed point $\boldsymbol{r}^*$ and operating point $\boldsymbol{x}^*$:

$$\boldsymbol{r}_{t+1} = A^*\boldsymbol{r}_t + \underbrace{\boldsymbol{g}(A\boldsymbol{r}^* + B\boldsymbol{x}_t + \boldsymbol{d}) - \mathrm{d}\boldsymbol{g}(A\boldsymbol{r}^* + B\boldsymbol{x}_t + \boldsymbol{d}) \circ A\boldsymbol{r}^*}_{\boldsymbol{u}(\boldsymbol{x}_t)}, \tag{11}$$

where $A^* = \mathrm{d}\boldsymbol{g}(A\boldsymbol{r}^* + B\boldsymbol{x}^* + \boldsymbol{d}) \circ A$. Second, we write $\boldsymbol{r}_{t+1}$ as the convolved sum of inputs

$$\boldsymbol{r}_{t+1} = \sum_{n=0}^{t} A^{*n}\boldsymbol{u}(\boldsymbol{x}_{t-n}) = \boldsymbol{h}(\boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \cdots), \tag{12}$$

which we Taylor series expand to yield the $N \times K$ coefficient matrix for K expansion terms.

Closed-loop architecture with SNP

For the closed-loop architecture with SNP, we begin with the pre-programmed RNNs,

$$\frac{1}{\gamma}\dot{\boldsymbol{r}} = -\boldsymbol{r} + \boldsymbol{g}(\bar{A}\boldsymbol{r} + \bar{B}\bar{\boldsymbol{x}} + B\boldsymbol{x} + \boldsymbol{d}), \boldsymbol{r}_{t+1} = \boldsymbol{g}(\bar{A}\boldsymbol{r}_t + \bar{B}\bar{\boldsymbol{x}}_t + B\boldsymbol{x}_t + \boldsymbol{d}), \tag{13}$$

for continuous-time and discrete-time systems, respectively (Fig. 2b). Using SNP, we program an output $\bar{\boldsymbol{o}} = \bar{W}\boldsymbol{r} = \bar{\boldsymbol{f}}(\bar{\boldsymbol{x}}, \boldsymbol{x})$ and perform feedback as $A = \bar{A} + \bar{B}\bar{W}$ to yield

$$\frac{1}{\gamma}\dot{\boldsymbol{r}} = -\boldsymbol{r} + \boldsymbol{g}((\bar{A} + \bar{B}\bar{W})\boldsymbol{r} + B\boldsymbol{x} + \boldsymbol{d}), \boldsymbol{r}_{t+1} = \boldsymbol{g}((\bar{A} + \bar{B}\bar{W})\boldsymbol{r}_t + B\boldsymbol{x}_t + \boldsymbol{d}), \tag{14}$$

for continuous-time and discrete-time systems, respectively.

The sample lag operator is defined as

$$\bar{o}_{i,t+1} = \bar{f}(\bar{\boldsymbol{x}}_t, x_t) = \begin{cases} \bar{x}_{i+1,t} & 1 \le i < n \\ x_t & i = n, \end{cases} \tag{15}$$

which shifts the state of all inputs down by one index. The short-time Fourier transform with a sliding window of length n is defined as

$$o_{i,t+1} = \sum_{j=0}^{n-1} \cos\left(\frac{2\pi ij}{n}\right)\bar{x}_{j+1,t}, o_{i+n,t+1} = \sum_{j=0}^{n-1} \sin\left(\frac{2\pi ij}{n}\right)\bar{x}_{j+1,t}. \tag{16}$$

Data availability

There are no data with mandated deposition used in the manuscript or supplement. All data in the main text and Supplementary Information are generated by the code that is publicly available online.

Code availability

All figures were directly generated in MATLAB from the code available on Code Ocean, available upon publication at https://codeocean.com/capsule/7077387/tree (ref. 55).

References

1. Nieder, A. & Dehaene, S. Representation of number in the brain. *Annu. Rev. Neurosci.* **32**, 185–208 (2009).
2. Salmelin, R., Hari, R., Lounasmaa, O. V. & Sams, M. Dynamics of brain activation during picture naming. *Nature* **368**, 463–465 (1994).
3. Hegarty, M. Mechanical reasoning by mental simulation. *Trends Cogn. Sci.* **8**, 280–285 (2004).
4. Silver, D. et al. Mastering the game of Go with deep neural networks and tree search. *Nature* **529**, 484–489 (2016).
5. Silver, D. et al. Mastering the game of Go without human knowledge. *Nature* **550**, 354–359 (2017).
6. Patterson, D. A. & Hennessy, J. L. *Computer Organization and Design ARM Edition: The Hardware Software Interface* (Morgan Kaufmann, 2016).
7. Von Neumann, J. First draft of a report on the EDVAC. *IEEE Ann. Hist. Comput.* **15**, 27–75 (1993).
8. Singh, C. & Levy, W. B. A consensus layer V pyramidal neuron can sustain interpulse-interval coding. *PLoS ONE* **12**, e0180839 (2017).
9. Gollisch, T. & Meister, M. Rapid neural coding in the retina with relative spike latencies. *Science* **319**, 1108–1111 (2008).
10. Sigman, M. & Dehaene, S. Brain mechanisms of serial and parallel processing during dual-task performance. *J. Neurosci.* **28**, 7585–7598 (2008).
11. Nassi, J. J. & Callaway, E. M. Parallel processing strategies of the primate visual system. *Nat. Rev. Neurosci.* **10**, 360–372 (2009).
12. Rissman, J. & Wagner, A. D. Distributed representations in memory: insights from functional brain imaging. *Annu. Rev. Psychol.* **63**, 101–128 (2012).
13. Cho, K., Van Merriënboer, B., Bahdanau, D. & Bengio, Y. On the properties of neural machine translation: encoder–decoder approaches. *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation* 103–111 (2014).
14. Towlson, E. K., Vértes, P. E., Ahnert, S. E., Schafer, W. R. & Bullmore, E. T. The rich club of the *C. elegans* neuronal connectome. *J. Neurosci.* **33**, 6380–6387 (2013).
15. Werbos, P. J. Backpropagation through time: what it does and how to do it. *Proc. IEEE* **78**, 1550–1560 (1990).
16. Caporale, N. & Dan, Y. Spike timing–dependent plasticity: a Hebbian learning rule. *Annu. Rev. Neurosci.* **31**, 25–46 (2008).
17. Tishby, N., Pereira, F. C. & Bialek, W. The information bottleneck method. Preprint at *arXiv* https://doi.org/10.48550/arXiv.physics/0004057 (2000).
18. Olshausen, B. A. & Field, D. J. Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature* **381**, 607–609 (1996).
19. Kline, A. G. & Palmer, S. Gaussian information bottleneck and the non-perturbative renormalization group. *New J. Phys.* **24**, 033007 (2021).
20. Lukoševičius, M., Jaeger, H. & Schrauwen, B. Reservoir computing trends. *Künstl. Intell.* **26**, 365–371 (2012).
21. Jaeger, H. The "echo state" approach to analysing and training recurrent neural networks—with an erratum note. *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report* **148**, 13 (2001).
22. Sussillo, D. & Abbott, L. F. Generating coherent patterns of activity from chaotic neural networks. *Neuron* **63**, 544–557 (2009).
23. Lu, Z., Hunt, B. R. & Ott, E. Attractor reconstruction by machine learning. *Chaos* **28**, 061104 (2018).
24. Kocarev, L. & Parlitz, U. Generalized synchronization, predictability, and equivalence of unidirectionally coupled dynamical systems. *Phys. Rev. Lett.* **76**, 1816 (1996).
25. Smith, L. M., Kim, J. Z., Lu, Z. & Bassett, D. S. Learning continuous chaotic attractors with a reservoir computer. *Chaos* **32**, 011101 (2022).
26. Kim, J. Z., Lu, Z., Nozari, E., Pappas, G. J. & Bassett, D. S. Teaching recurrent neural networks to infer global temporal structure from local examples. *Nat. Mach. Intell.* **3**, 316–323 (2021).
27. Canaday, D., Pomerance, A. & Gauthier, D. J. Model-free control of dynamical systems with deep reservoir computing. *J. Phys. Complex.* **2**, 035025 (2021).
28. Gauthier, D. J., Bollt, E., Griffith, A. & Barbosa, W. A. Next generation reservoir computing. *Nat. Commun.* **12**, 5564 (2021).

29. Eliasmith, C. & Anderson, C. H. *Neural Engineering: Computation, Representation, and Dynamics in Neurobiological Systems* (MIT Press, 2003).
30. Zhang, Y. et al. A system hierarchy for brain-inspired computing. *Nature* **586**, 378–384 (2020).
31. Eliasmith, C. et al. A large-scale model of the functioning brain. *Science* **338**, 1202–1205 (2012).
32. Tanaka, G. et al. Recent advances in physical reservoir computing: a review. *Neural Netw.* **115**, 100–123 (2019).
33. Pasqualetti, F., Zampieri, S. & Bullo, F. Controllability metrics, limitations and algorithms for complex networks. *IEEE Trans. Control Netw. Syst.* **1**, 40–52 (2014).
34. Karrer, T. M. et al. A practical guide to methodological considerations in the controllability of structural brain networks. *J. Neural Eng.* **17**, 026031 (2020).
35. Bekolay, T. et al. Nengo: a Python tool for building large-scale functional brain models. *Front. Neuroinform.* **7**, 48 (2014).
36. Rosenblum, M. & Garfinkel, T. Virtual machine monitors: current technology and future trends. *Computer* **38**, 39–47 (2005).
37. Südhof, T. C. Synaptic neurexin complexes: a molecular code for the logic of neural circuits. *Cell* **171**, 745–769 (2017).
38. Lerner, T. N., Ye, L. & Deisseroth, K. Communication in neural circuits: tools, opportunities, and challenges. *Cell* **164**, 1136–1150 (2016).
39. Feller, M. B. Spontaneous correlated activity in developing neural circuits. *Neuron* **22**, 653–656 (1999).
40. Calhoon, G. G. & Tye, K. M. Resolving the neural circuits of anxiety. *Nat. Neurosci.* **18**, 1394–1404 (2015).
41. Maass, W., Joshi, P. & Sontag, E. D. Computational aspects of feedback in neural circuits. *PLoS Comput. Biol.* **3**, e165 (2007).
42. Clarke, L. E. & Barres, B. A. Emerging roles of astrocytes in neural circuit development. *Nat. Rev. Neurosci.* **14**, 311–321 (2013).
43. Sussillo, D. Neural circuits as computational dynamical systems. *Curr. Opin. Neurobiol.* **25**, 156–163 (2014).
44. Graves, A. et al. Hybrid computing using a neural network with dynamic external memory. *Nature* **538**, 471–476 (2016).
45. Kompa, K. & Levine, R. A molecular logic gate. *Proc. Natl Acad. Sci. USA* **98**, 410–414 (2001).
46. Zhang, M. & Ye, B.-C. A reversible fluorescent DNA logic gate based on graphene oxide and its application for iodide sensing. *Chem. Commun.* **48**, 3647–3649 (2012).
47. Pittman, T., Fitch, M., Jacobs, B. & Franson, J. Experimental controlled–not logic gate for single photons in the coincidence basis. *Phys. Rev. A* **68**, 032316 (2003).
48. Fang, Y., Yashin, V. V., Levitan, S. P. & Balazs, A. C. Pattern recognition with "materials that compute". *Sci. Adv.* **2**, e1601114 (2016).
49. Stern, M., Hexner, D., Rocks, J. W. & Liu, A. J. Supervised learning in physical networks: from machine learning to learning machines. *Phys. Rev. X* **11**, 021045 (2021).
50. Pashine, N., Hexner, D., Liu, A. J. & Nagel, S. R. Directed aging, memory, and nature's greed. *Sci. Adv.* **5**, eaax4215 (2019).
51. Chen, T., Pauly, M. & Reis, P. M. A reprogrammable mechanical metamaterial with stable memory. *Nature* **589**, 386–390 (2021).
52. Boedecker, J., Obst, O., Lizier, J. T., Mayer, N. M. & Asada, M. Information processing in echo state networks at the edge of chaos. *Theory Biosci.* **131**, 205–213 (2012).
53. Svoronos, S., Stephanopoulos, G. & Aris, R. Bilinear approximation of general non-linear dynamic systems with linear inputs. *Int. J. Control* **31**, 109–126 (1980).
54. Bevanda, P., Sosnowski, S. & Hirche, S. Koopman operator dynamical models: learning, analysis and control. *Annu. Rev. Control* **52**, 197–212 (2021).
55. Kim, J. Z. & Bassett, D. S. A neural machine code and programming framework for the reservoir computer. *Nat. Mach. Intell.* (2023).

Acknowledgements
We gratefully acknowledge M. X. Lim, K. A. Murphy, H. Ju, D. Zhou and J. Stiso for conversations and comments on the manuscript. J.Z.K. acknowledges support from the National Science Foundation Graduate Research Fellowship No. DGE-1321851, and the Cornell Bethe/KIC/Wilkins Theory Postdoctoral Fellowship. D.S.B. acknowledges support from the John D. and Catherine T. MacArthur Foundation, the ISI Foundation, the Alfred P. Sloan Foundation, an NSF CAREER award PHY-1554488, and the NSF through the University of Pennsylvania Materials Research Science and Engineering Center (MRSEC) DMR-1720530.

Author contributions
J.Z.K. conceived the initial idea and developed the analyses in conversation with D.S.B. J.Z.K. and D.S.B. prepared the manuscript. All authors contributed to discussions and approved the manuscript.

Competing Interests
The authors declare no competing interests.

Additional information
Supplementary information The online version contains supplementary material available at https://doi.org/10.1038/s42256-023-00668-8.

Correspondence and requests for materials should be addressed to Dani S. Bassett.

Peer review information *Nature Machine Intelligence* thanks Adrian Valente, Brian DePasquale and the other, anonymous, reviewer(s) for their contribution to the peer review of this work.