nature nanotechnology

SUPPLEMENTARY INFORMATION

DOI: 10.1038/NNANO.2017.83

In the format provided by the authors and unedited.

# Sparse coding with memristor networks

Patrick M. Sheridan, Fuxi Cai, Chao Du, Wen Ma, Zhengya Zhang & Wei D. Lu*

## 1. Measurement Setup

Figure S1 shows a schematic of the board along with an optical micrograph of the test board with an integrated memristor chip. The setup can measure arrays in sizes of up to 32 rows and 32 columns and perform a broad range of tests and array operations.

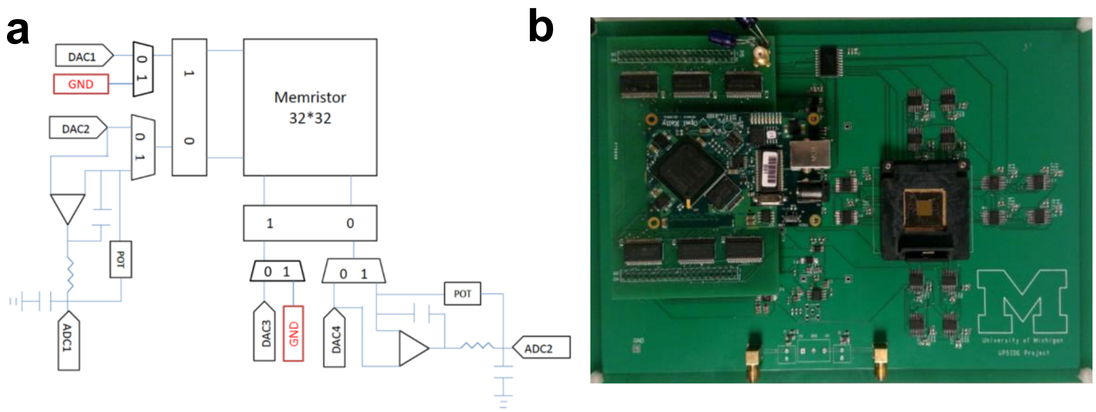


**Figure S1** | **a** Block diagram of the test board. **b** Optical micrograph of the test board with an integrated memristor chip.

To achieve weight updates in the experiments, we used a single pulse with different pulse widths to program the memristor resistance. In this so-called "single shot" programming scheme, read verify and repeated programming steps were not employed.

Specifically, during write we apply $V_{write}$ to the rows containing the target memristors and 1/2 $V_{write}$ to the rest of the rows. Initially, all rows are set to be 1/2 $V_{write}$ to protect the memristors from being disturbed. The selected column is grounded and the other columns are set at 1/2 $V_{write}$.

When programming starts, the rows containing the target memristors are set to $V_{write}$, with different pulse widths corresponding to the target weight update values. When a pulse ends, the voltage on the corresponding row drops back to 1/2 $V_{write}$. The programming is completed when all target devices on the selected column are programmed. Fig. S2 below illustrates the programming procedure.

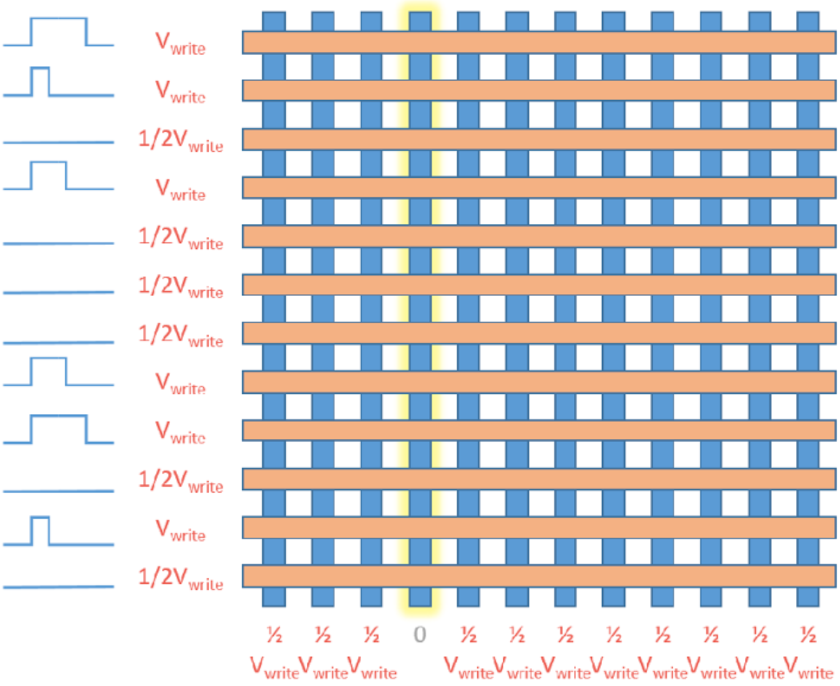


**Figure S2** | Memristor array programming scheme.

All devices in the 32×32 array work well and have been tested for different patterns beyond the pattern shown in Fig. 1c in the main text. Below are a few examples showing test patterns written in and read out from the 32×32 crossbar array.

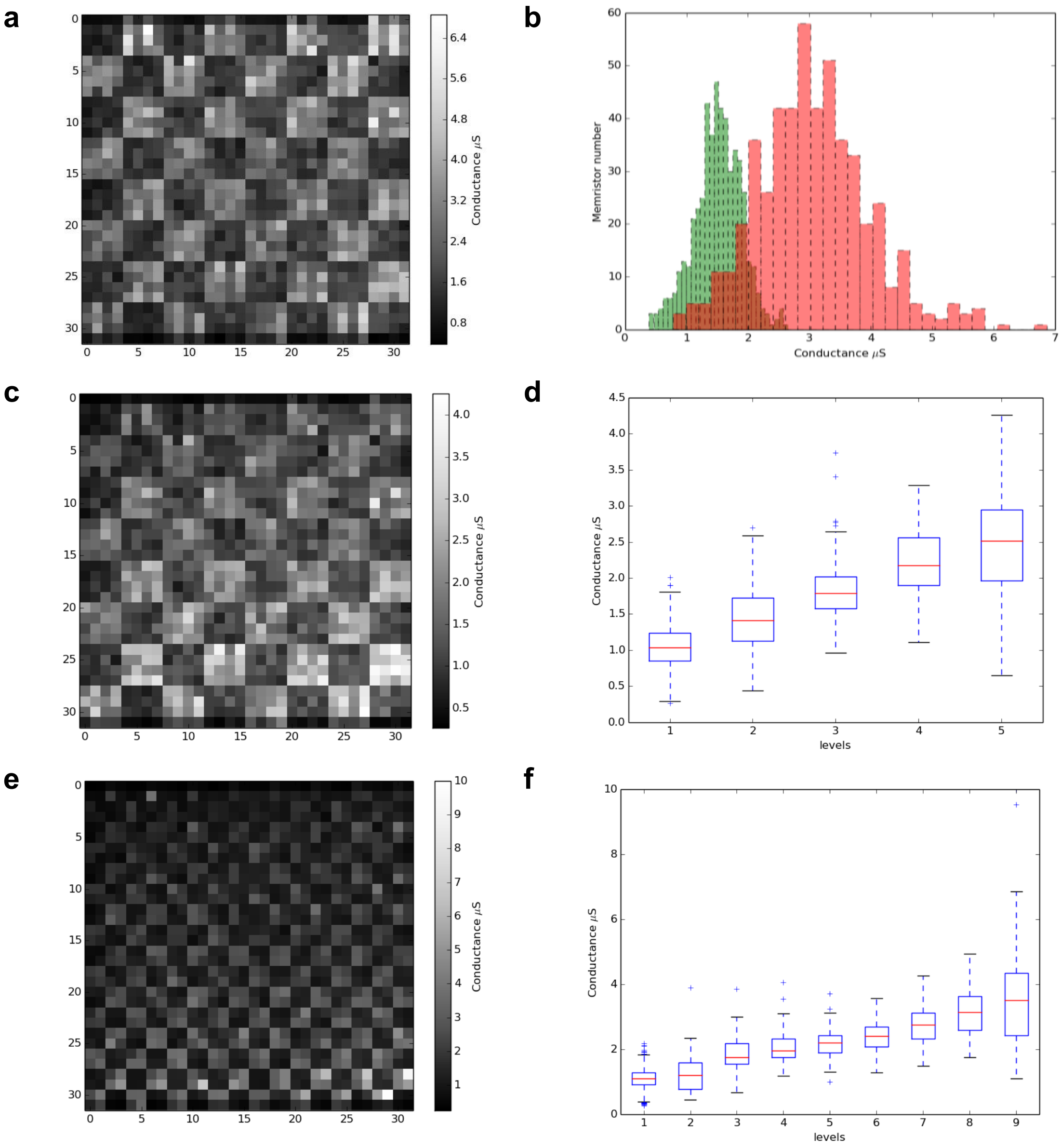


**Figure S3 | a** A binary 4×4 checkerboard pattern stored in the array. **b** Histogram of the device conductance values. **c** A 4×4 checkerboard pattern with 5 conductance levels stored in the same array. **d** Distribution of the stored conductance values showing the average (red lines) and standard deviation. **e** A 2×2 checkerboard pattern stored in the same array in 9 levels. **f** Distribution of the stored conductance values showing the average (red lines) and standard deviation.

## 2. 200 nm pitch 32×32 memristor array fabrication and testing results

Memristor arrays based on narrower linewidths, e.g. 200 nm linewidth, have also been fabricated. Fig. S4 shows SEM images of a fabricated 32×32 array with 200 nm linewidth.

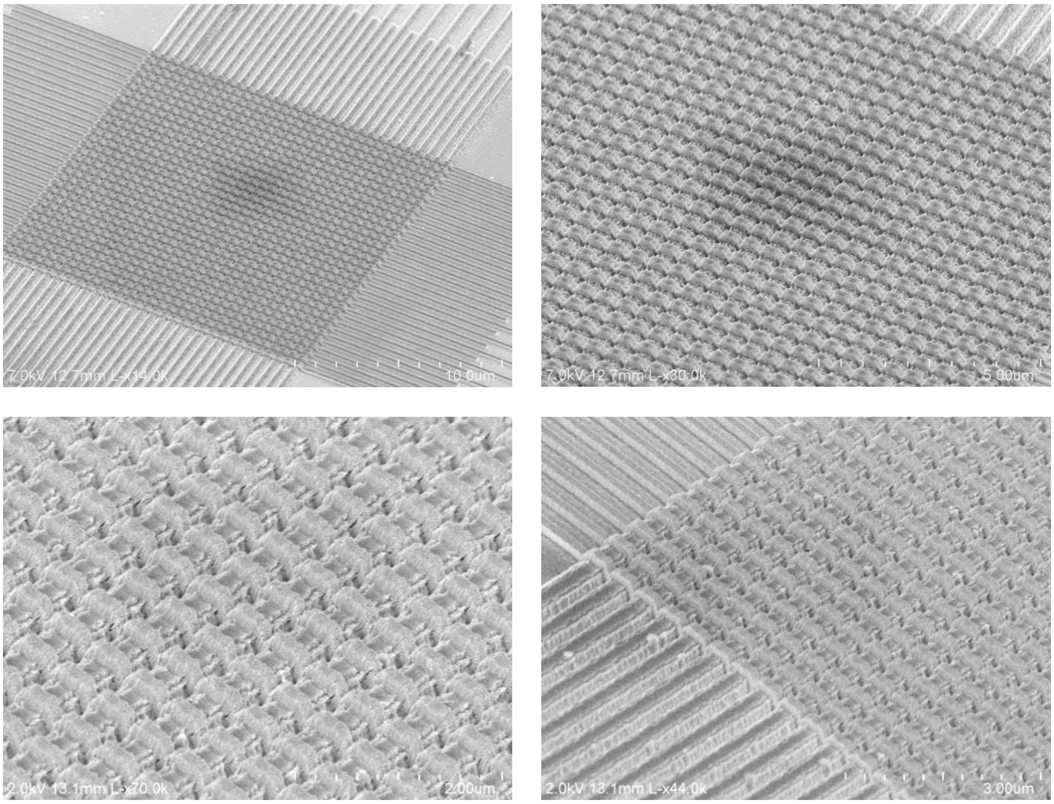


**Figure S4** | SEM images of a 200 nm linewidth memristor array.

Patterns such as checkerboards were also successfully programmed in the higher density, 200 nm linewidth arrays, as shown in Fig. S5. However, devices in the outer 4 lines (2 on each side) in the 200 nm arrays typically show higher resistance values compared to the inner devices, possibly due to lithography and etch non-uniformity issues during the fabrication process.

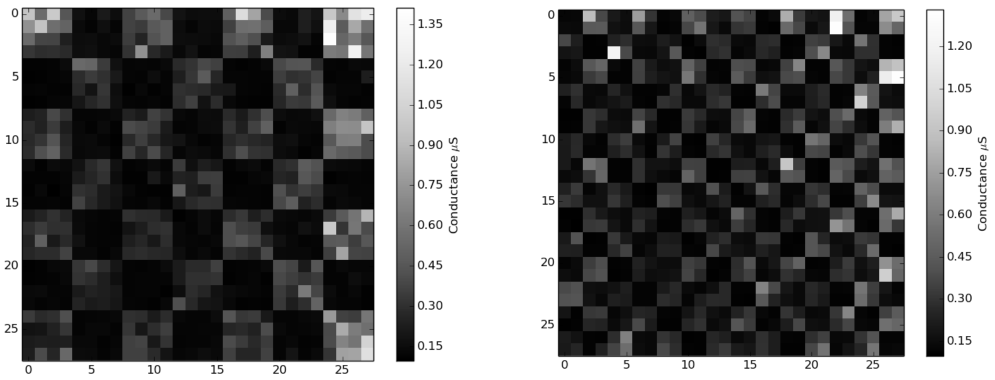


**Figure S5** | 4×4 and 2×2 checkerboard patterns programmed in the 200 nm linewidth array.

## 3. Additions Notes on Sparse Coding Implementation

In the locally competitive algorithm (LCA) [R1] a vector of signal inputs (*i.e.* image pixels) is used to excite the network. In our approach the input values (such as the intensity of pixels in a gray-scale image) are translated to voltage pulse durations with a fixed voltage amplitude, so that the total charge passed through the memristors is linearly proportional to the input, weighted by the memristor conductance, as discussed in the text. For each output neuron, the crossbar modulates the inputs with a synaptic weight vector (represented by the conductances of the memristors in the same column) and converts them into currents that flow into the neuron. In this sense, the memristor network performs the matrix dot-product operation $x^T \cdot D$ through a *single* read operation, where the matrix multiplication operations are performed in parallel. Here $x$ is the input vector and $D$ is the memristor weight matrix, as discussed in the main text.

After converting the input through the memristor weight matrix, the obtained current is then integrated to determine the neuron's membrane potential. Additionally, in LCA the membrane potential is affected by a leakage term, as well as inhibition from other active neurons. The inhibition effect is an important component of LCA and the strength of the inhibition is proportional to the similarity of the neurons' receptive fields. This feature is critical in ensuring sparsity by preventing duplicate neurons from firing with the same/similar receptive fields. Mathematically, the neuron's membrane potential dynamics is determined by:

$$\frac{du}{dt} = \frac{1}{\tau}(-u + x^T \cdot \mathrm{D} - a \cdot (\mathrm{D}^\mathrm{T}\mathrm{D} - I)) \qquad \text{(S1)}$$

Here $a$ is the activity of the neuron. In conventional approaches (e.g. GPU or MS-CMOS), the inhibition is achieved by either computing $\mathrm{D}^\mathrm{T}\mathrm{D}$ on the fly which is very compute-intensive, or by storing all the inhibition weights $\mathrm{D}^\mathrm{T}\mathrm{D}$ in a separate 'feedback' memory. However, since inhibition is all-to-all, the feedback memory scales with $n^2$, where $n$ is the number of output neurons and will grow very fast and become impractical as the input becomes larger.

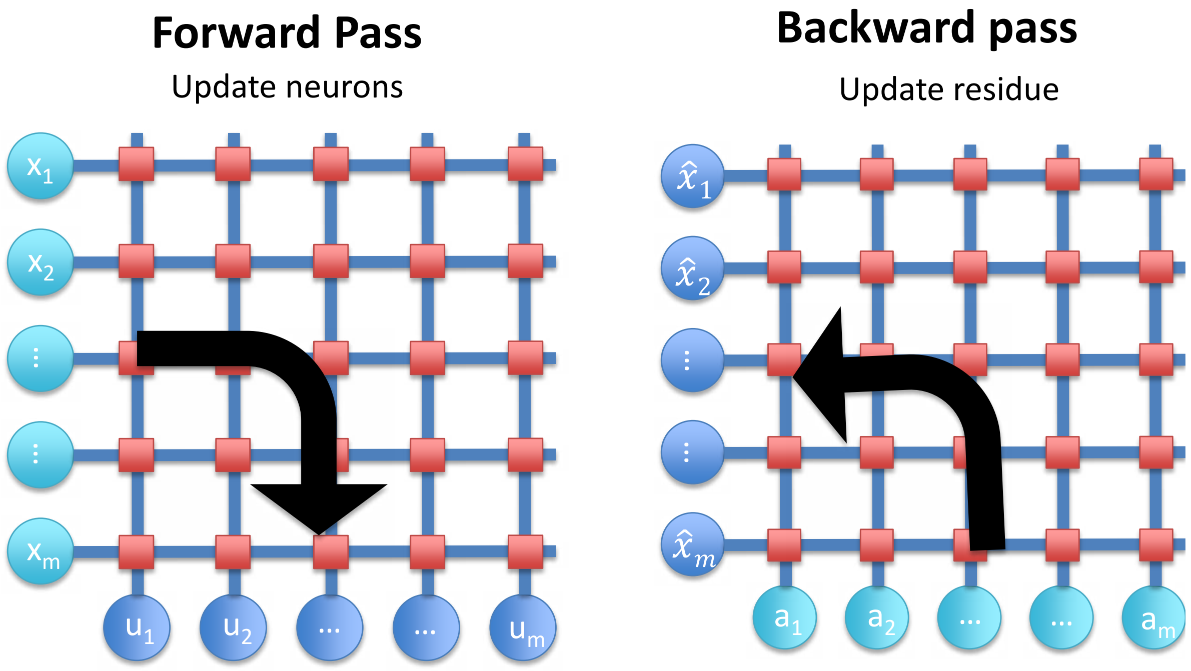
Eq. S1 can be re-written as

$$\frac{du}{dt} = \frac{1}{\tau}(-u + (x - \hat{x})^T D + a) \qquad \text{(S2)}$$

Where the original input $x^T$ is replaced with $(x - \hat{x})^T$ (the residual term), where

$$\hat{x} = Da^T \qquad \text{(S3)}$$

is the reconstructed signal based on the activities of the output neurons $a$ and the receptive field matrix $D$. This residual term is then fed to the network as the input. This approach equivalently achieves inhibition since the features associated with the firing neurons are now removed from the input, so the membrane potentials of neurons with similar receptive fields as the firing ones will be suppressed. This process eliminates calculating $\mathrm{D}^\mathrm{T}\mathrm{D}$ or the feedback memory, but it requires the dot-product operation of the neuron activity vector $a$ and the transpose of the weight matrix $D^T$, which are again very compute-intensive. However, with the memristor network, the operation $a \cdot D^T$ can also be readily implemented by *a single operation* by feeding the neuron activity vector $a$ backwards through the matrix, as shown in Fig. S6, where the dot-product $a \cdot D^T$ can be achieved through a single read at the input. The new input $(x - \hat{x})^T$ will then be calculated, and forward-feed into the matrix and this process is repeated until the network settles and a sparse representation of the original input can be obtained from the activity vector $a$.


Forward Pass

Update neurons

Backward pass

Update residue

**Figure S6** | The forward pass that updates the neuron membrane potentials and the backward pass that updates the residual input.

During the experiments, the time constant for the leaky integrating neuron (τ in Eq. 2a of the main text and Eq. S2) was chosen to be 60. The choice of τ must balance system stability with coding efficiency. A lower τ causes neurons to charge and leak more quickly which can yield a sparse code in fewer iterations of the algorithm, but can also cause neurons to oscillate near the threshold. A τ of 60 allowed the sparse code to be obtained within 60 iterations while allowing the network to stabilize with neurons settling either above or below the threshold.

In our implementation, the neuron circuit is implemented digitally in software, using discrete time steps following Eqs. S2 and S3. The unit of τ is the time step used in the discrete time implementation, *i.e.* τ = 60 means the integration and decay time constant in Eq. S2 is 60 time steps. The duration of each time step can be calculated from the unit read pulse width (60 us in our experiments) and the input/output numbers. Counting both the forward and backward read cycles, the physical time for each time step is ~ 2.88 ms using the test board.

From Eq. S2, the membrane potential $u$ in our implementation should have the same unit as the dot-product $x^T D$. As discussed in the main text, in our implementation the vector-matrix dot-product was calculated by measuring the total charge of a forward read pass, which can be written and measured as $G^TVt$, where $G$, $V$, $t$ represents conductance, voltage and pulse width, respectively. As a result, the membrane potential has a unit of charge. A typical value of the membrane potential can be readily estimated. As shown in the example in Fig. S3, $G$ has conductance values of 3~7 uS, $V$ is 0.6 V, the minimum pulse width is around 60 µs during the forward and backward passes, leading to membrane potential values on the level of 1e-10 C.

## 4. Other Experimental Examples of Network Dynamics

Additional examples obtained from the experimental setup showing the neuron membrane potential dynamics during the LCA analysis of the bar patterns can be seen in Figs. S7-S9.

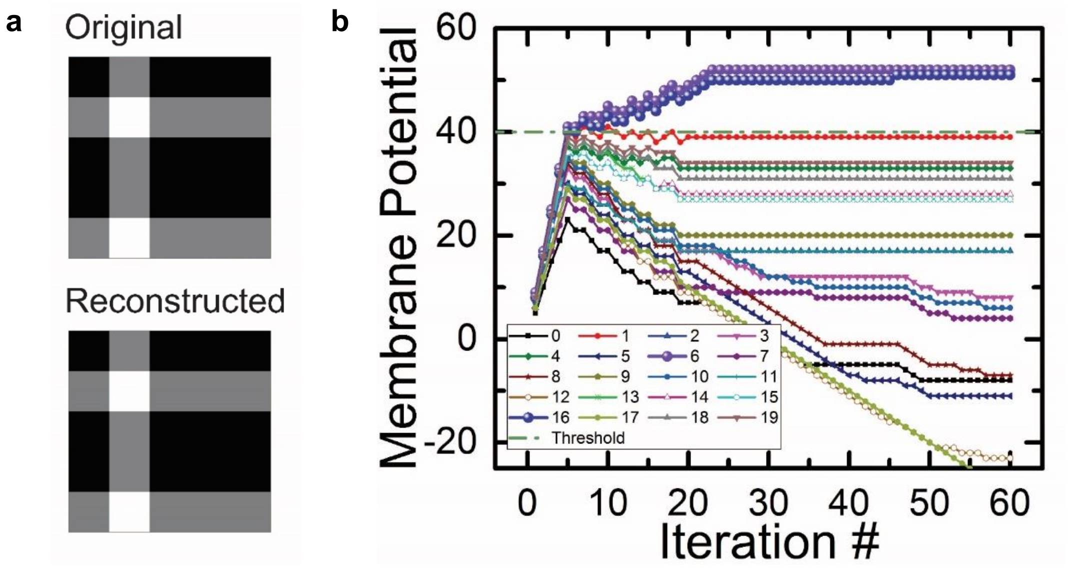


**Figure S7** | **a** Original and reconstructed image based on neurons 6 and 16. **b** Neuron dynamics during LCA analysis.

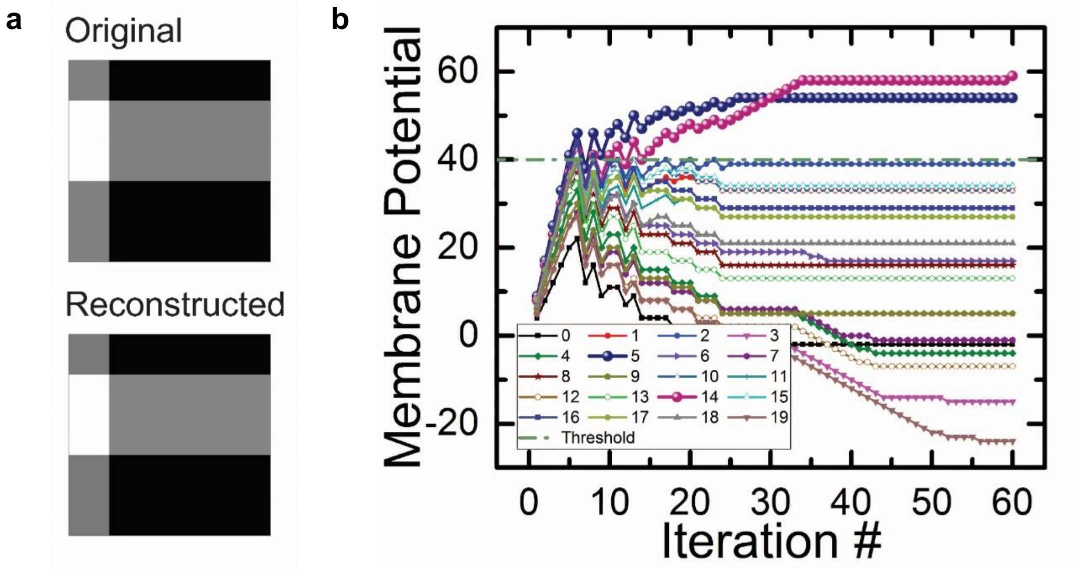


**Figure S8** | **a** Original and reconstructed image based on neurons 5 and 14. **b** Neuron dynamics during LCA analysis.

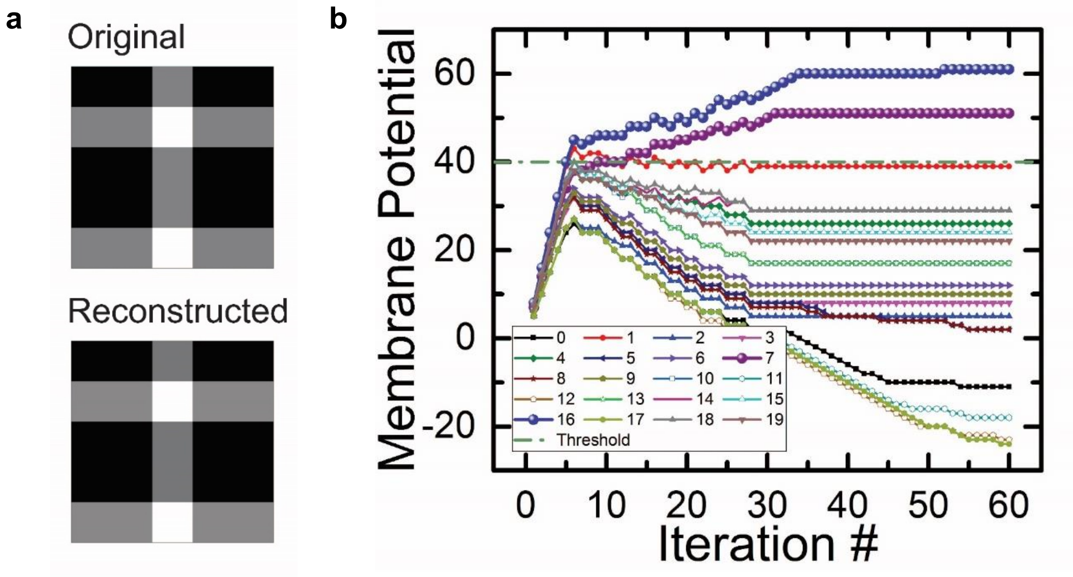


**Figure S9** | **a** Original and reconstructed image based on neurons 7 and 16. **b** Neuron dynamics during LCA analysis.

## 5. Simulation of the Memristor Network

A simulation package based on Python was developed to simulate and analyze the behaviors of the memristor crossbar network. For simplicity and efficiency, sneak-path currents were neglected since the output neurons are essentially grounded. Line resistance was also neglected in the simulation. Large-scale, algorithmic simulation was performed in Python using the NumPy framework and utilities from the SciKit-Image.

The memristor devices were modeled using the device model discussed in Section 8. The network was initialized by randomly initializing the matrix storing the state variable $w$ of the memristor devices using a Gaussian distribution.

The effects of device variations were added by using the parameters in Fig. S16c during weight updates (online learning) or weight storage (offline learning). An example of the offline learning case, shown in Fig. S14, verified that the dictionary obtained from the simulation method captures the main features of the experimentally stored dictionary in the presence of device variations.

During simulation of sparse coding using the memristor array (*e.g.* image reconstruction using bar patterns or natural images), the pixel intensity of the original image/patch was converted into a vector, and multiplied with the $D$ matrix in the simulation package to calculate the vector-matrix dot-product. Membrane potentials $u$ and neuron activities $\underline{a}$ were initialized as 0, and updated following Eqs. S2 and S3. Afterwards, another vector-matrix multiplication of the current neuron activity vector $a$ and the $D$ matrix was performed to calculate the residual term in Eq. S3. The residual was then used as the new input of the next iteration, and the whole process was repeated in a tic-toc manner.

Fig. S10 below shows an example of the simulated memristor system operation. A bar pattern example, used in Fig. 3b-c in the main text, was tested for this purpose. In the simulation, all parameters including $\lambda$ and $\tau$ were chosen to be the same as the experimental study shown in Fig. 3 in the main text. The simulation generated the same correct reconstruction (Figure S10a) and reproduced similar neuron dynamics (Figure S10b) as observed experimentally using the memristor system, shown in Fig. 3c.

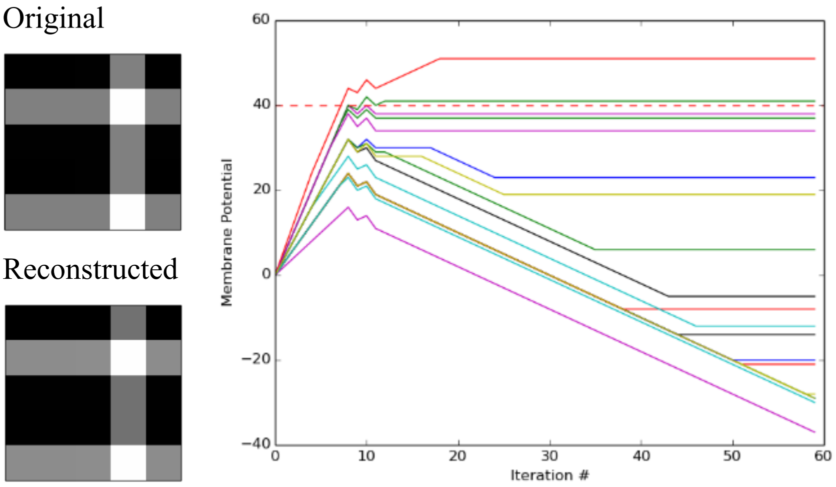


**Figure S10** | Simulation results of the bar pattern reconstruction using the device model. Similar network dynamics can be reproduced in the simulation.

## 6. Impact of the Threshold Parameter λ on Network Performance

In LCA the sparsity coefficient λ directly affects the neuron activity and the reconstruction. As seen in Fig. 3c of the main text, the membrane potential of neuron 1 becomes suppressed after 21 iterations and remains (just) below the threshold. As a result, the activity of neuron 1 is exactly zero (following Eq. 2b) and the reconstruction of the input is obtained from the activities of only neurons 8 and 16. Similarly, in Fig. 2c the membrane potentials of neurons 9 and 17 are maintained just above the threshold. A question can then be asked as to how sensitive the reconstruction is to the sparsity coefficient λ, *i.e.* will neuron 1 in Fig. 3c become active if a slightly lower λ is used?

To investigate the role of the threshold parameter on the sparse code outcome, a series of experiments were performed using the memristor-based test board, using the same input but with progressively larger threshold values.

The bar pattern shown in Fig. S11a was used in this study. The threshold parameter λ was varied between 0 and 140, in increments of 10 and the sparse reconstructions for each setting are shown in Fig. S11b. The threshold setting is shown above each panel, along with the number of the active neurons, $L_0$. The trend is summarized in Fig. S11c.

Overall, a higher threshold parameter causes the network to favor sparsity over representation accuracy, as predicted by Eq. 1 in the main text, reproduced as Eq. S4 below:

$$\min_{a}( |x - Da^T|_2 + \lambda|a|_0 ) \qquad \text{(S4)}$$

At low threshold values, more neurons are used to represent an input and can thus potentially capture more input features (*e.g.* the greyscale distinction between the two horizontal bars). However, increasing λ does not necessarily result in a reduced number of active neurons, as seen in the cases of $\lambda = 90$ or $\lambda = 120$. In fact, by changing λ, the network dynamics will adapt and often the same set of active neurons whose receptive fields optimally represent features in the input will prevail in the end. In Figs. 2c, 3c in the main text and Figs. S7-S9, it can be seen that neurons often settle close to the threshold. This is a consequence of the network dynamics: when a neuron surpasses the threshold, the residual is immediately reduced, retarding further increase of similar neurons; similarly, if a neuron is driven below threshold, the residue error it was introducing is immediately removed, reducing the negative drive. Therefore, if the threshold is adjusted upwards, the neuron dynamics will shift accordingly so that the neurons will settle around the new threshold and resulting in the same optimized output. On the other hand, if the threshold adjustment is large enough, the balance between accuracy and sparsity can shift to favor more sparse solutions and affect the network dynamics, as shown in the step-like changes around λ = 30, 90, and 120. The choice of threshold allows a level of tunability in balancing between representation accuracy and the sparsity of the solution, while the ability of the network to adapt to the threshold parameter relaxes the requirement of optimizing the threshold parameter. This is particularly helpful because of the discrete nature of the $L_0$-norm which results in a non-convex optimization problem, and there exists no analytic expression for choosing λ to achieve a given level of sparsity.

It is also interesting to note that a worse representation is obtained in the case of $\lambda = 30$ (and below) where 4 (or more) neurons are active, while a better reconstruction is obtained for $\lambda = 40$ with a more sparse representation using fewer (2) neurons. The error at low threshold, e.g. $\lambda = 30$, is due to device variability within the network that allow a neuron to charge faster than others, and the insufficient inhibition due to the low threshold could not drive the incorrect neuron below the threshold. This problem is solved at higher threshold $\lambda = 40$, which by emphasizing more on sparsity has the desired effect of suppressing the less-optimally matched neurons. This finding presents another argument for the use of a sparsity constraint when coding with emerging hardware: the enforcement of sparsity helps to suppress errors resulting from device non-idealities present in real hardware systems and leads to more accurate analysis.

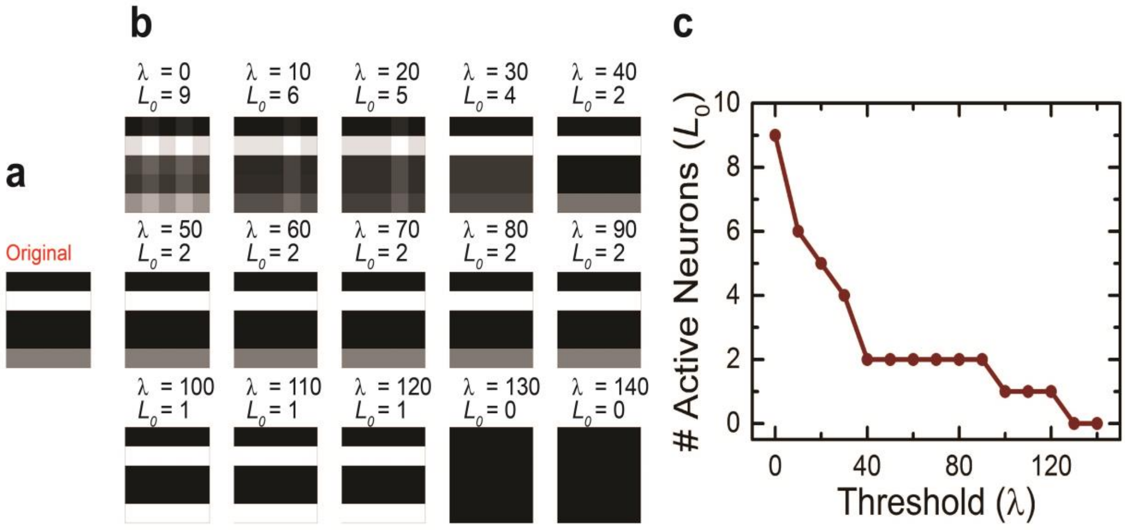


**Figure S11 | Impact of the threshold parameter λ network performance. a** Original image to be sparsely encoded. **b** Image reconstructions for $\lambda \in [0,140]$. The number of active neurons ($L_0$) is shown above each reconstruction. **c** $L_0$ as a function of λ. A higher threshold in general leads to more sparse representation.

## 7. Dictionary Learning – Offline Training and Weight Storage

The image reconstruction with bar patterns and stripe-like patterns shown in Figs. 2-3 in the main text involved sparse coding with a known dictionary *D*, without the process of dictionary learning. Training and finding the dictionary *D* is another critical optimization problem. Since the conductances of the crossbar memristors are tunable, we can implement a learning algorithm to find the dictionary elements during the training stage. We first demonstrated this concept through offline learning by using our memristor model combined with a Hebbian learning rule. After learning, the learned dictionary *D* was written into the physical memristor crossbar to perform image reconstruction using the LCA-based sparse coding algorithm.

Specifically, we first used a winner-take-all (WTA) training algorithm, combined with Oja’s rule to find the overcomplete dictionary *D*. More elegant learning, based also on sparse coding, are discussed in Section S10. The WTA learning algorithm allows fast training of the network. During WTA learning, the network finds the dictionary element that mostly resembles the input by identifying the element (the winner) that produces the largest dot-product with the input. After identifying the winning neuron, its synaptic weights, forming the receptive field $\phi_w$ associated with the neuron, is then updated by following Oja's rule (Eq. S5).

$$y = x^T\phi_w \tag{S5}$$

$$\Delta\phi_w = \beta(X - y\phi_w)y \tag{S6}$$

Where $x$ is the input and $y$ is the output from the winning neuron, representing the maximum dot-product of the input vector and the receptive fields in the network (Eq. S6).

To train the network, 4×4 patches are randomly sampled from a training set of nine natural images (128x128 pixels) as the training input, shown in Fig. S12.

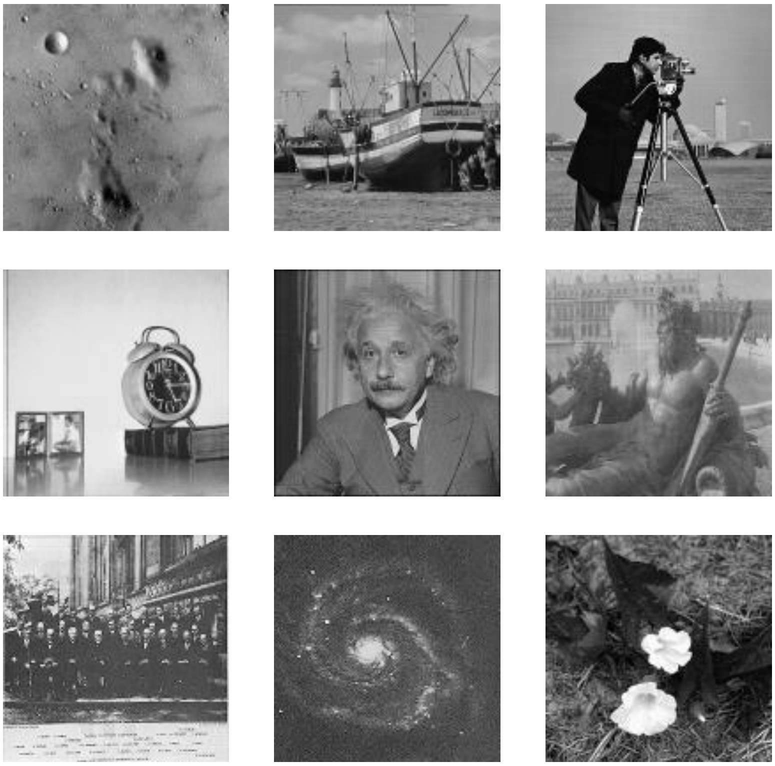
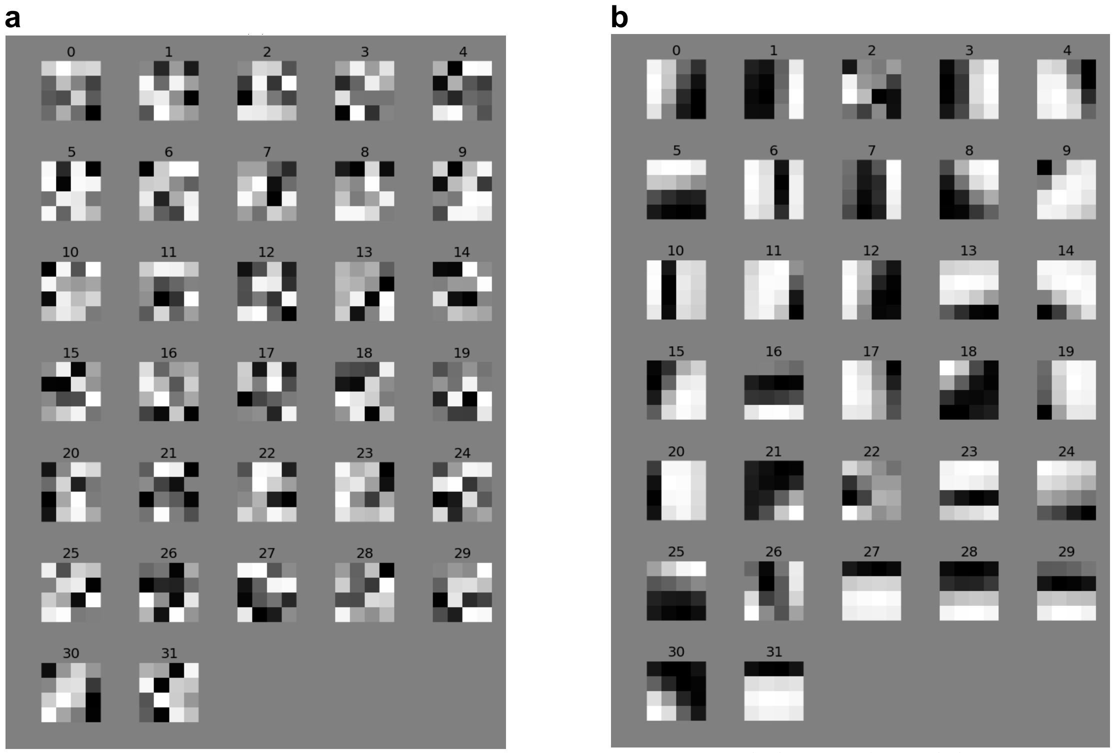


**Figure S12** | Training set used to obtain the dictionaries.

Figure *S13*Figure S13 shows the offline training results using the memristor model on a 16×32 crossbar array, corresponding to a 2× overcomplete dictionary with 16 inputs and 32 output neurons (dictionary elements). Before training, the weights are randomly initialized, as shown in Fig. S13a. The network is trained with 150k samples. After training is completed, receptive fields resembling Gabor filters, represented by the conductances of memristors associated with each output neuron, can be clearly observed, as shown in Fig. S13b.



Initial weights

Dictionary after training

**Figure S13** | All 32 receptive fields before (**a**), and after (**b**) training using WTA, using the natural images shown in Fig. S12. Each receptive field is represented by the conductance values of the 16 memristors associated with the given neuron.

The trained dictionary (Fig. S13b) is then experimentally programmed into the 16×32 memristor array using the test board. Fig. S14a shows the receptive fields experimentally stored in and read out from the memristor crossbar and used for the experimental natural image reconstructions in Fig. 4b-e of the main text. Due to intrinsic device variations, the patterns stored in the array (Fig. S14a) are not exactly the same as the ideal dictionary (Fig. S13b), but generally maintain the main features of the learned dictionary elements. Here the receptive fields were written into the array using a single shot method, without repeated read verification and re-programming steps.

Using the learned dictionary that is now stored in the memristor crossbar, we successfully performed reconstruction of 120×120 pixel grayscale images using the 16×32 memristor crossbar, as shown in Fig. 4 in the main text. During the process, the 120×120 input images were divided into 4×4 patches and each patch was experimentally processed using the memristor crossbar and the LCA algorithm (Fig. 4b-d). After the memristor network stabilizes (typically after 80 forward/backward iterations, Fig. 4d), the patch was reconstructed using the neuron activities and the corresponding receptive fields stored in the crossbar array.

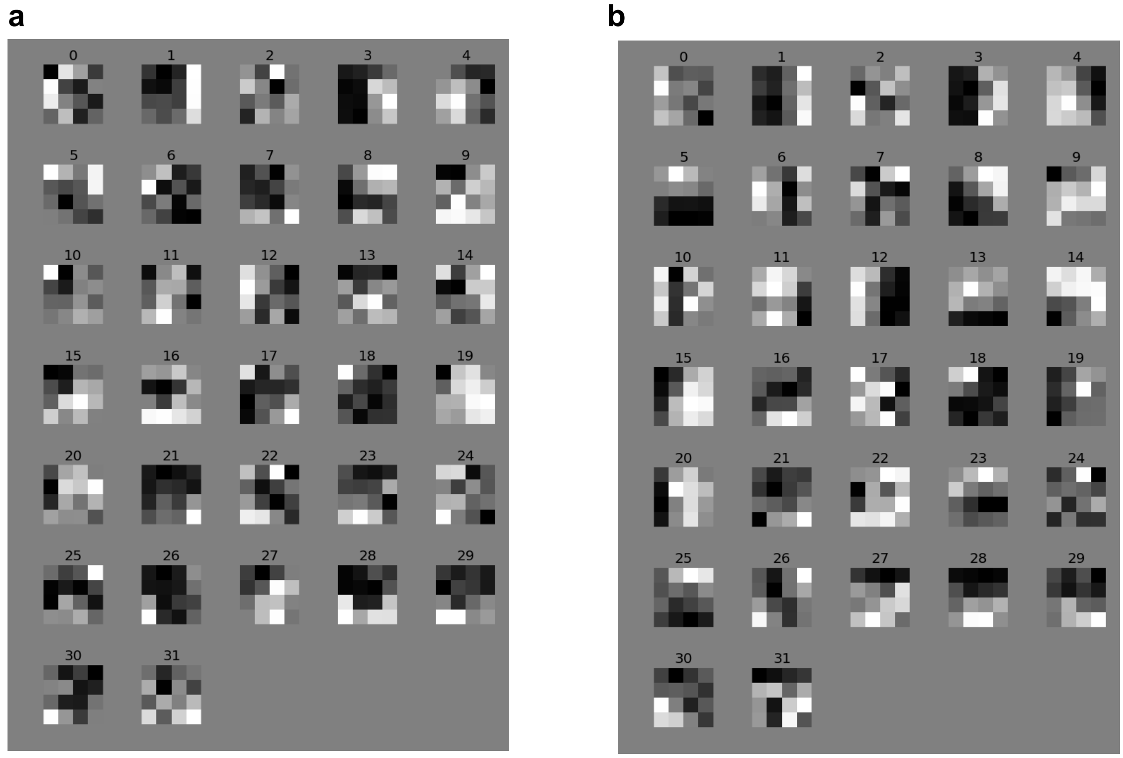


**Figure S14 | a** Experimentally stored dictionary elements in the memristor crossbar. **b** Simulated stored weights after considering device variations.

The experimental results can be qualitatively explained by considering device variations during storage of the receptive fields into the memristor array in the offline dictionary case. Details of the device variation model are discussed in Section 8. Fig. S14b shows the simulated stored dictionary elements after adding device variations to the offline learned dictionary in Fig. S13b. Qualitatively similar behaviors can be observed compared with the experimentally stored dictionary elements shown in Fig. S14a.

Simulations of image reconstruction were then performed using the simulation package discussed in Section 5 and the simulated stored offline-trained dictionary elements, following the same procedure as the experimental reconstruction of the 120×120 natural images using 4×4 patch inputs. Figs. 4e-f in the main text show comparison of the experimentally obtained images from the memristor crossbar test board, and simulated reconstruction results based on the receptive fields shown in Fig. S14, respectively. The experimental results can be consistently reproduced by the simulation that includes effects of device variations.

## 8. Device Variation Model

Ideally one would like to perform *online* learning experimentally using the same memristor crossbar system. However, the slow speed of the board combined with the large training set prevented us from experimentally implementing online learning in the memristor crossbar at this stage. To test the feasibility of online learning we instead performed a detailed, realistic simulation using a device model that incorporates device variations during the incremental weight updates. The largest difference between the online learning case and the offline learning case is that in the offline case the receptive fields (dictionary elements) were obtained using an ideal device model and variations were only considered when the receptive fields were stored in the memristor array; while in the online learning case device variations were carefully considered during the learning stage that required large numbers of incremental weight updates. Additionally, in the online case the same array that was trained was used for reconstructions of test images, thus dictionary storage is no longer needed. Single shot programming schemes were also used for the weight updates during the online training case.

To model the device variations during weight updates, we experimentally measured the incremental conductance changes in 288 devices in the memristor array using pulsed programming/erasing conditions. The results are shown in Fig. S15a. Here each device was programmed with 20 write pulses and followed by 20 erase pulses, and the device conductance was monitored after each write/erase pulse by a read pulse.

We fitted the experimental data with the memristor model we developed in Ref. [S2], with the following equations:

$$I = w(\gamma \sinh(\delta V)) + (1 - w)(\alpha(1 - e^{-\beta V})) \qquad \text{(S7)}$$

$$\frac{dw}{dt} = \eta_1 \sinh(\eta_2 V) F(w, V) \qquad \text{(S8)}$$

where Eq. S7 is the current-voltage equation dependent on the internal state variable *w*, and Eq. S8 describes the update rate of the state variable. F(w,V) is a window function:

$$F(w, V) = \begin{cases} 1 - w, if\ V > 0 \\ w,\ if\ V < 0 \end{cases} \qquad \text{(S9)}$$

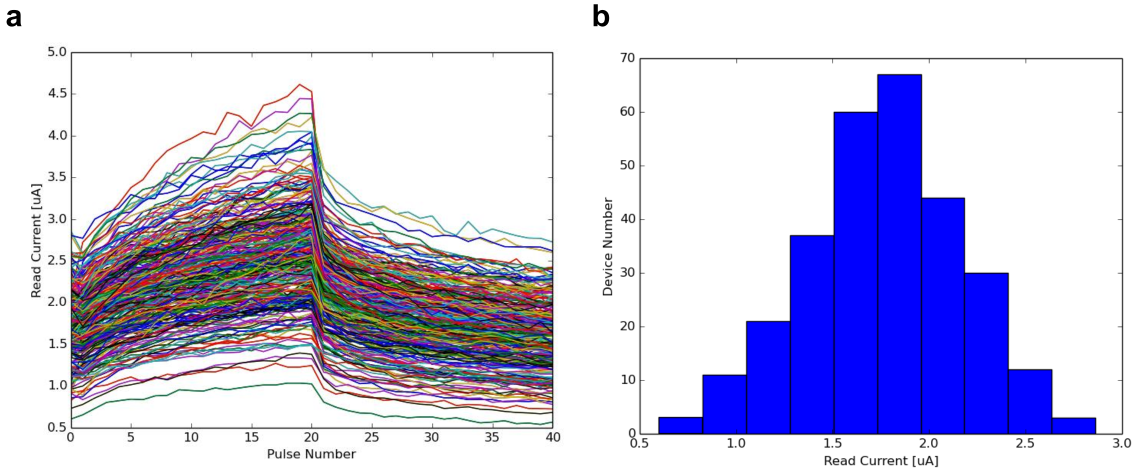
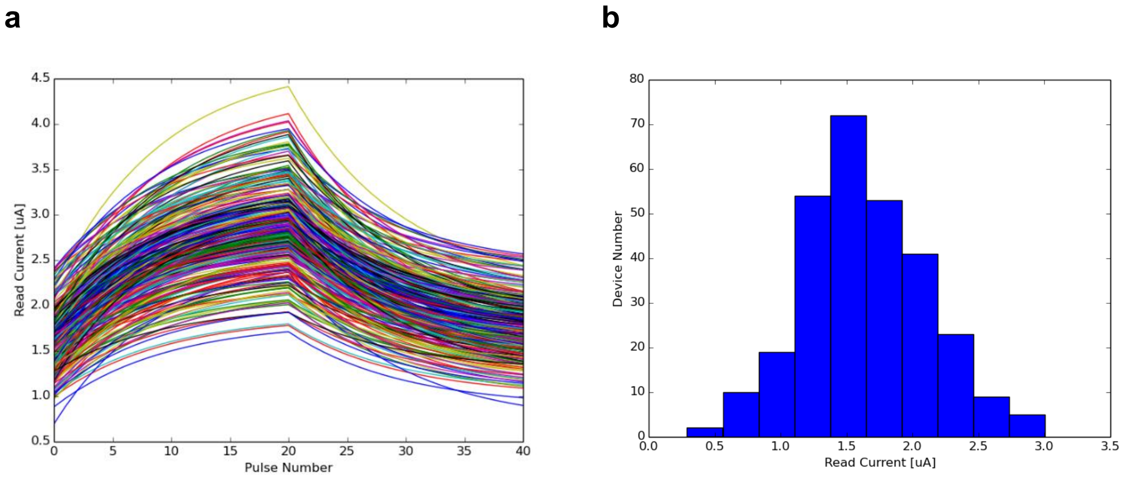


**Figure S15 | a** Pulse write/erase data measured from 288 devices in the memristor array. **b** Initial conductance distribution.

To perform online learning, we need to account for the device variations during weight updates. We first note that the initial conductances of the devices follow a Gaussian distribution, as shown in Fig. S15b. Thus we assumed that the initial values of the state variable ($w_0$) from different devices follow the same Gaussian distribution as observed in Fig. S15b. During weight updates, we further assumed that the parameters $\eta_1$ and $\eta_2$ in the weight update equation (Eq. S8) are different for different devices, also following Gaussian distributions. Exact parameters used in the model are shown in Fig. S16c. With these modifications to the original ideal device model, the experimentally observed variations during weight updates can be realistically accounted for, as shown in Fig. S16a, for the same programming/erasing conditions used in the experiments.



c

$$p(w_o = w) = \frac{1}{\sqrt{2\sigma^2\pi}} e^{-\frac{(w-\mu)^2}{2\sigma^2}}, \quad \mu = 0.03, \quad \sigma = 0.009$$

$$\gamma = \text{1e-5} \quad \delta = 4 \quad \alpha = \text{1e-8} \quad \beta = 0.5$$

$$\eta_1 = \text{9e-8 w/ 3\% variation} \quad \eta_2 = \text{15.5 w/ 1\% variation}$$

**Figure S16** | **a** Simulated incremental conductance changes that match well with the experimental results. **b** Gaussian distribution of the initial weights used in the simulation. **c** Parameters used in the device model.

## 9. Dictionary Training – Online Learning with WTA

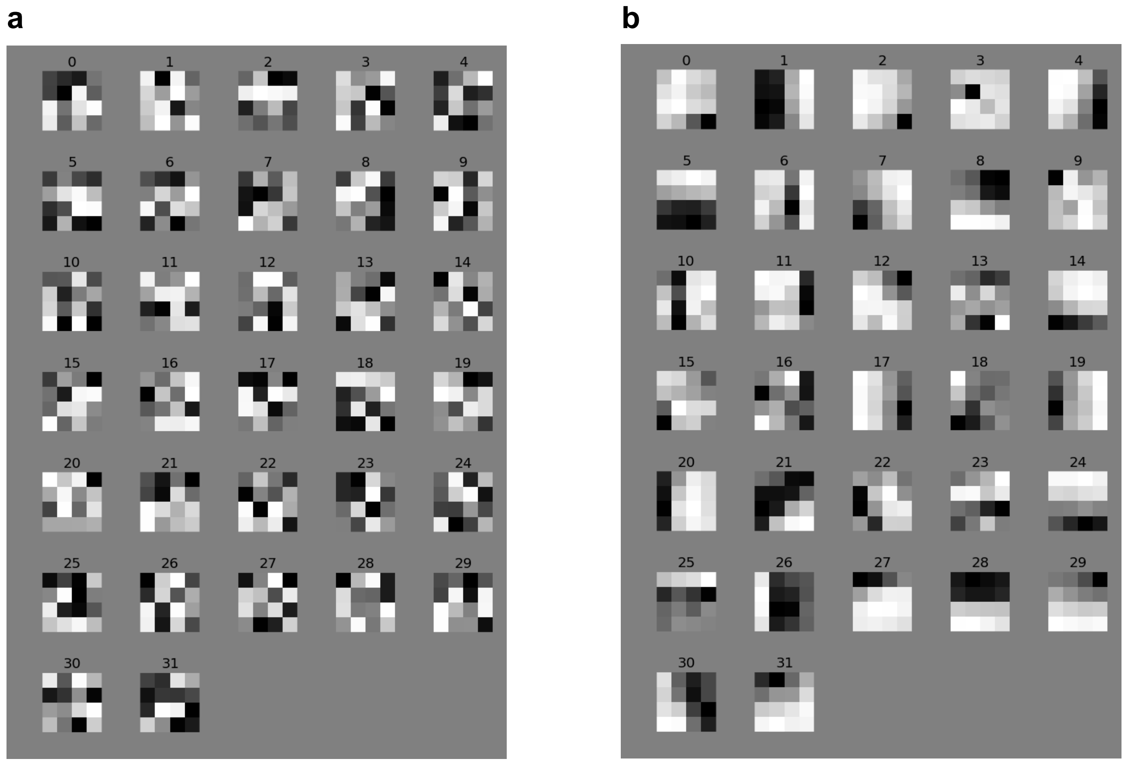


**Figure S17 | a** Initial weights in the memristor array. **b** Simulated learned weights considering device variabilities during online learning.

The model incorporating device variations during weight updates was then used to simulate online learning in the memristor crossbar. WTA based learning was first analyzed. The dictionary after learning based on WTA is shown in Fig. S17b. After learning, the same crossbar was used to process test images using the learned dictionary. Image reconstruction results using the online learned dictionary are shown in Fig. S18a. For comparison, simulation results of reconstructed images using an ideal dictionary with zero device variations are also included (Fig. S18b).

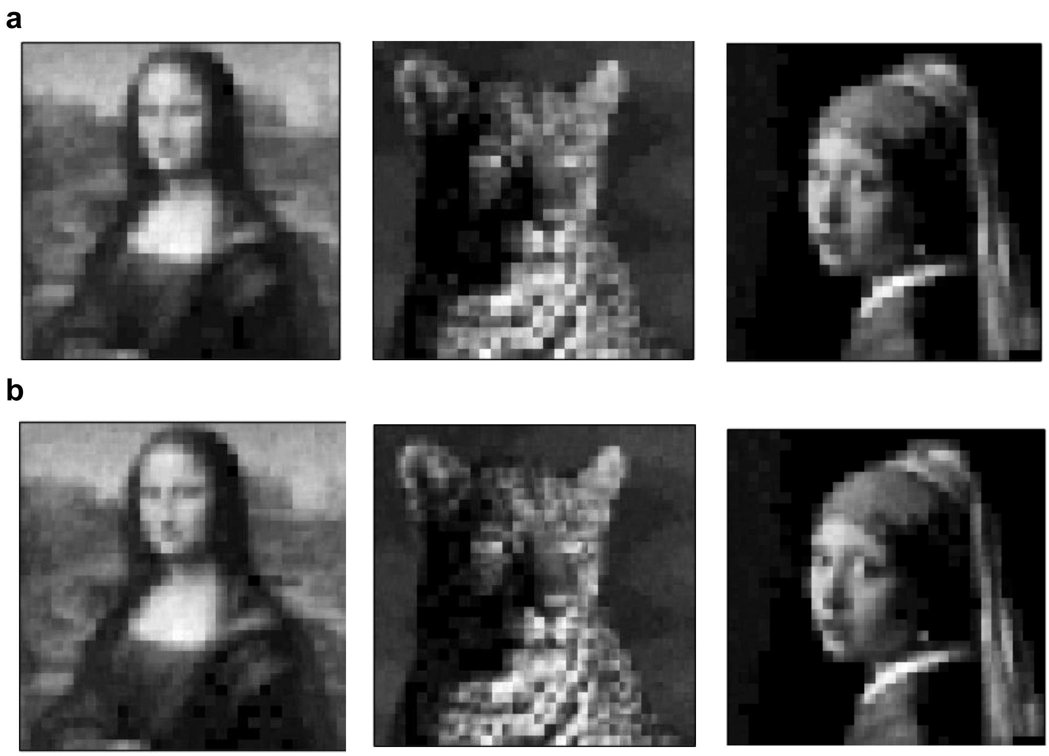


**Figure S18 | a** Reconstructed image with online learned dictionary based on WTA training. **b** Reconstructed image with ideal dictionary based on WTA training without device variations.

An interesting observation is that better results, closer to the ideal dictionary case, are obtained with the online learning process compared with results obtained from the stored, offline-learned weights, as shown Fig. S18. This effect can be explained from the fact that the learning algorithm is self-adaptive and adjusts to the device variabilities during the training stage. As a result, online learning can more effectively handle device variations compared to the offline training and weight storage method, where the differences of specific devices were not factored into the learned dictionary and can lead to larger errors during image reconstruction.

## 10. Dictionary Training – Dictionary Learned via Sparse Coding

We note WTA-trained dictionaries are not ideally suited for analysis based on sparse coding, since WTA forces the learned dictionary elements to resemble the most dominate patter in the input patches (*e.g.* Fig. S13b) while in sparse coding multiple dictionary elements are needed to reconstruct the input. To improve the image reconstruction results, we performed analysis on image processing using dictionary learned via sparse coding.

During dictionary training, we initialized the memristor dictionary to random values, and then during each iteration of learning we obtained the residual error $(x - \hat{x})$ following the LCA algorithm, and updated the memristor weights directly by incrementing or decrementing their values according to stochastic gradient descent, as explained in Olshausen & Field [S3].

Specifically, the stochastic gradient descent update rule is described as:

$$\Delta\Phi^{\mathrm{T}} = \beta(x - \mathrm{D}^{T}a)\otimes a \qquad \text{(S10)}$$

where D is the matrix of dictionary elements (receptive fields), $\otimes$ is the outer product, $x - \mathrm{D}^{T}a$ represents the reconstruction error with $\mathrm{D}^{T}a$ being the reconstructed input based on LCA, and $\beta$ is the learning rate factor, which in our case is chosen to be 0.3.

We first performed the analysis using the algorithm without considering any device non-idealities. Indeed, much better results were obtained using the sparse coding trained dictionary, compared to what was obtained earlier based on dictionary learned using WTA, as shown in Fig. 4g in the main text. Note a larger patch (8×8) was also used during learning and inference, compared with the 4×4 patches used in Figs. 4e-f.

More importantly, after incorporating realistic memristor noise (conductance fluctuations) and variations (weight update fluctuations) in the model based on the memristor device model discussed in Section 8, very high quality reconstructions can still be obtained by performing *online* learning with memristors in the loop, as shown in Fig. 4h. These results again verify that online learning can more effectively handle device variations and is particularly suitable for emerging devices such as memristor-based systems where large device variations are expected.

## 11. Dictionary Training – Learned Dictionary with Whitened Inputs

Following Olshausen & Field [S3], we note high spatial features in the dictionary may be more effectively trained using a whitening technique that filters out the low spatial frequency components. Specifically, a combined whitening/low-pass filter with a frequency response shown in Eq. S11 has been shown to lead to desired performance [S3]:

$$R(f) = fe^{-\left(\frac{f}{f_0}\right)^n} \qquad \text{(S11)}$$

The same combined whitening/low-pass filter was used to pre-process the image patches before training, where $f_0$ was chosen to be 200 cycles/picture and $n$ was set to be 4.

The training data were sampled from ten 512×512 natural images with 12×12 patches. Before the training, all patches were pre-processed using the combined whitening/low-pass filter shown in Eq. S11. Fig. S19 shows the original image before and after whitening, as well as the profile of the combined whitening/low-pass filter in frequency domain.

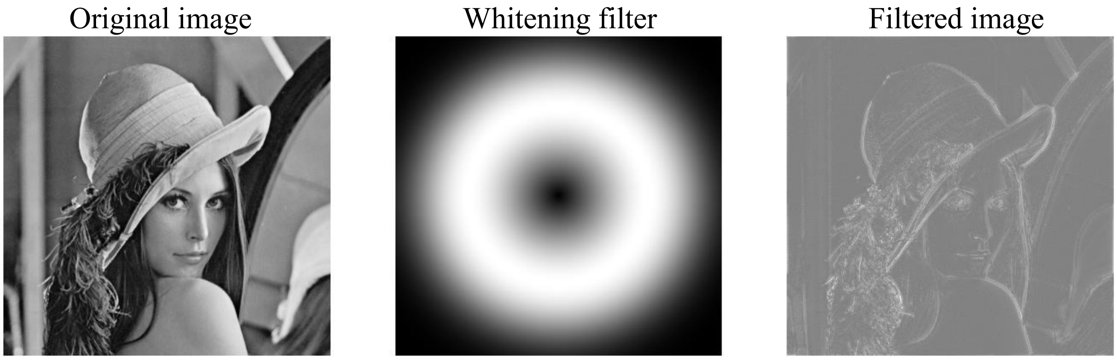


**Figure S19 |** The original image before and after whitening, along with the profile of the whitening/low-pass filter plotted in frequency domain.

12×12 whitened image patches, randomly sampled from the image set, were used as input patches for dictionary training, following the same training algorithm discussed in Section 10.

Fig. S20 shows the 576 dictionary elements after training (using 12×12 image patches and a 4× overcomplete dictionary). Similar to results obtained in Zylberberg et al. which used a comparable sparse-coding algorithm [S4], small unoriented features, oriented Gabor-like wavelets, and elongated edge-detectors can be learned in the dictionary.

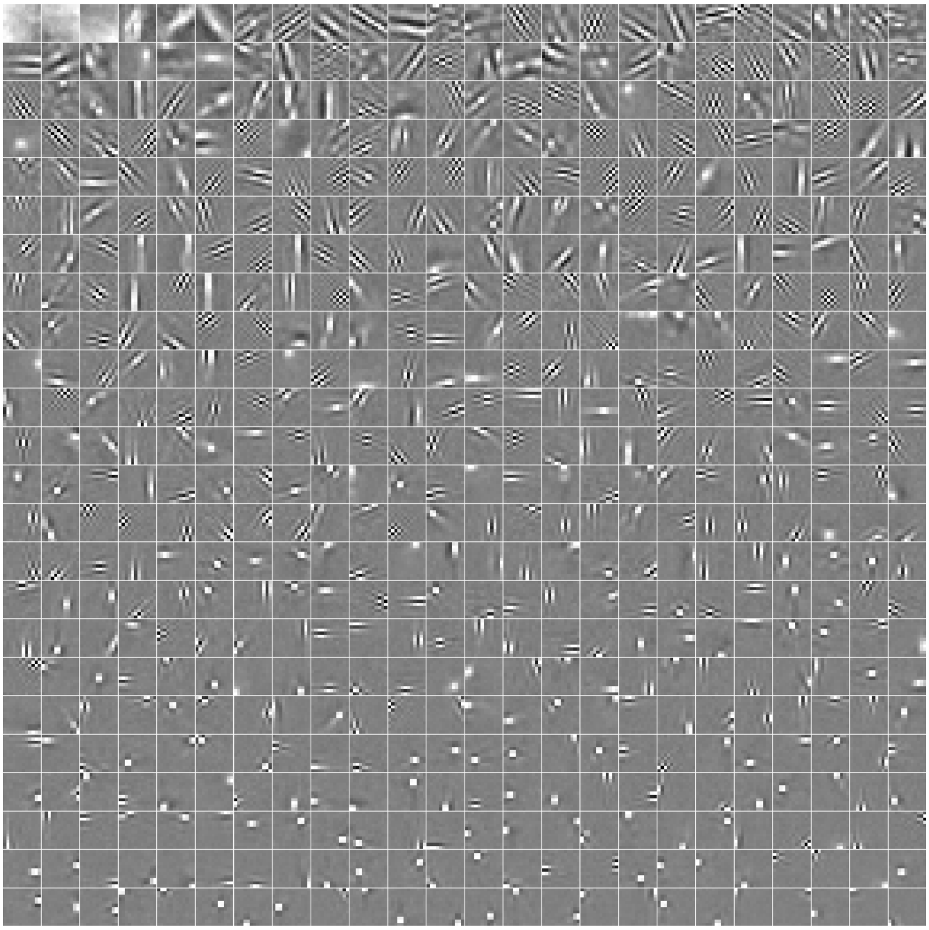


**Figure S20 |** Receptive fields obtained from gradient descent training using pre-preprocessed images. The resulting fields were roughly sorted using the ratio of the variance over the mean. The gray tone represents zero in all fields, with lighter pixels corresponding to positive values and darker pixels corresponding to negative values.

An example of image reconstruction using the whitened images and the trained dictionary in Fig. S20 is shown in Fig. S21, demonstrating high quality reconstruction using dictionary learned with whitened inputs.

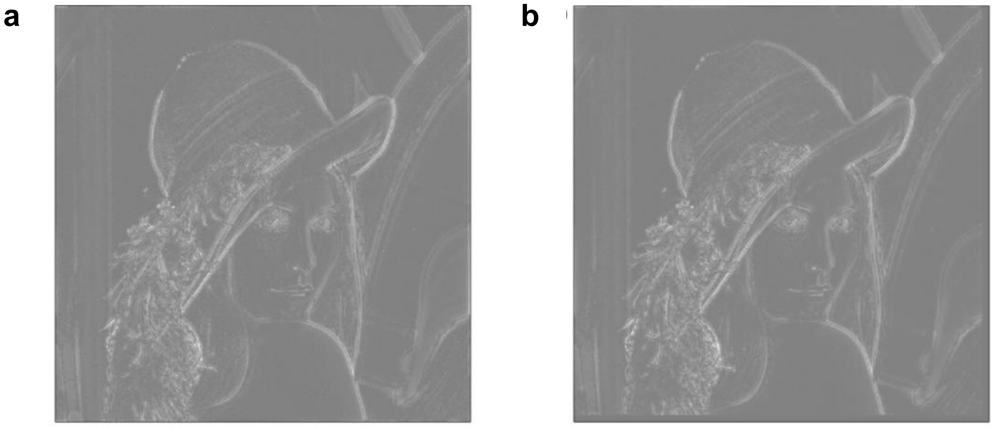


**Figure S21 | a** Original whitened image. **b** Reconstructed image using the dictionary in Fig. S20

### 12. Video Processing and Comparative Analysis

Many real-world applications such as video processing are well suited for sparse coding. We carried out analysis on how the memristor-based hardware will perform in video processing tasks. Fig. S22 shows an example of a 256×192 grayscale image, which was downsampled from an original 640×480 image. The image was then processed by the 16×32 memristor crossbar using 4×4 patches. During the process, 3072 (64×48) 4×4 patches were processed using the LCA algorithm and each patch took 300 iterations to allow the network to stabilize. Due to the limited data transfer rate between the memristor array and the digital circuitry in the existing test board, the current memristor board will not be able to perform this video analysis in real time. However, in an integrated memristor/CMOS system with a minimum possible read pulse width of 10 ns, our analysis shows that it will take 0.034 second to process such an image, meeting the requirement of real-time streaming video analysis at a rate of 24 frames/s (<0.042 second process time per frame).

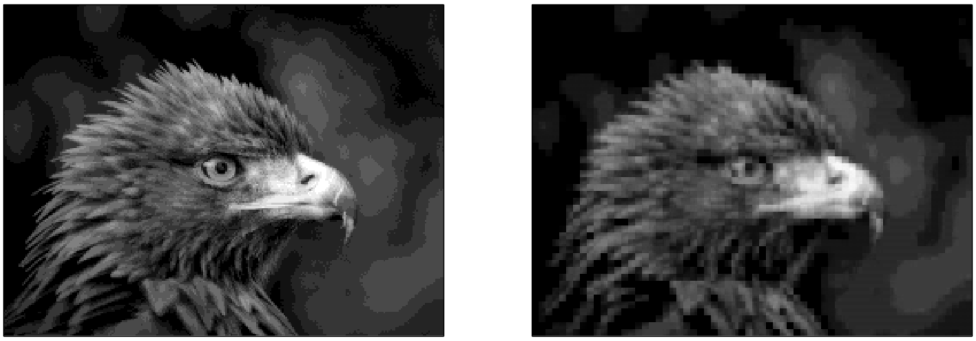


**Figure S22** | 256×192 video frame reconstructed using 4×4 patches using the 16×32 memristor crossbar.

To process standard 480p (640×480) videos with at least 24 frames/second frame rate without downsampling, larger patches (*e.g.* 10×10) will be needed. A memristor array size of 100×200 will be able to process the 480p videos in real time using 10×10 patches. Fig. S23 shows simulation results of image reconstruction using the larger memristor array.

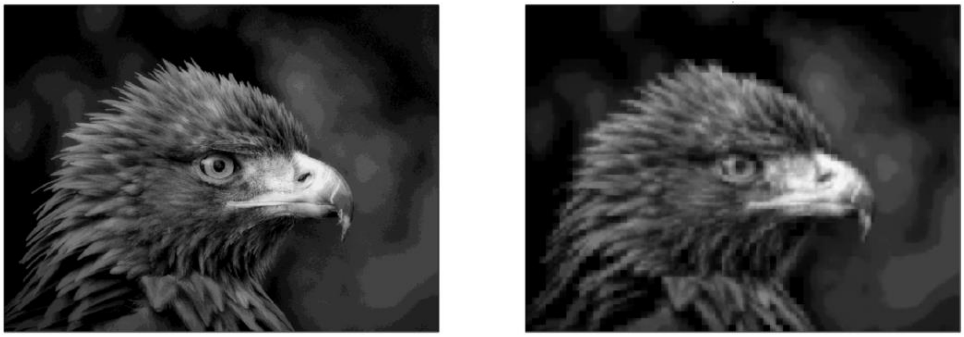


**Figure S23** | 640×480 video frame reconstructed using 10×10 patches with a 100x200 memristor crossbar.

We then compared the performance of the memristor system with efficient digital solutions. For a fair comparison of the crossbar-based analog solution with a digital solution, both methods need to be subjected to the same constraints, as tradeoffs can always be made between low energy consumption, fast speed and high reconstruction accuracy. In this application, we are targeting power and processing time as the main performance metrics for both systems.

To achieve the benchmarking results, we designed and analyzed an efficient digital system using efficient multiply-accumulation (MAC) circuits. Since an $a\times b$ crossbar can perform the $(1\times a)\times(a\times b)$ vector-matrix dot-product operation in *one* read process in the memristor system, the digital CMOS system was designed to match the same performance. Specifically, for benchmarking purpose we assumed that in an integrated memristor chip a read speed of 10 ns can be achieved. To obtain similar performance in the digital system, we used 4-bit × 4-bit multiplications to approximately match the dynamic range of the input and the stored values in the memristor crossbar. In the analysis, we used 10×10 patches (aimed for real time processing of 480p video) to sample and reconstruct the image. A 100×200 memristor crossbar was assumed for the memristor implementation. The equivalent digital system in 40 nm CMOS uses 1600 MAC circuits (8 MAC circuits per column and 200 columns operated in parallel) to accomplish one (1×100) × (100×200) vector-matrix dot-product operation in 10.4 ns. Schematic and parameters of the digital system are shown in Fig. S24 and Table S1, respectively. This design occupies 0.306 $mm^2$ and it is estimated to consume 274 mW during the forward pass (reconstruction phase) stage and 548mW during the backward pass (residual calculation) stage.

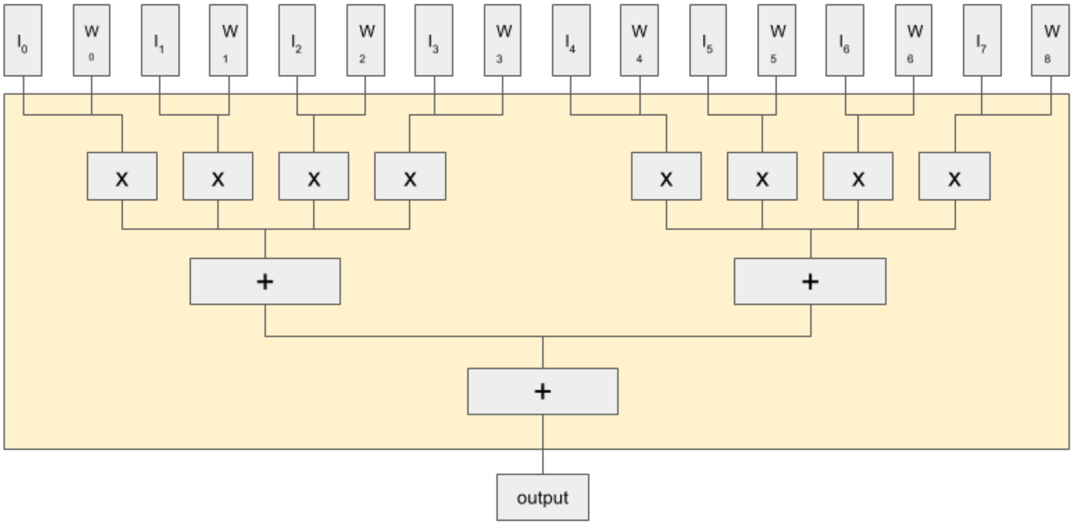
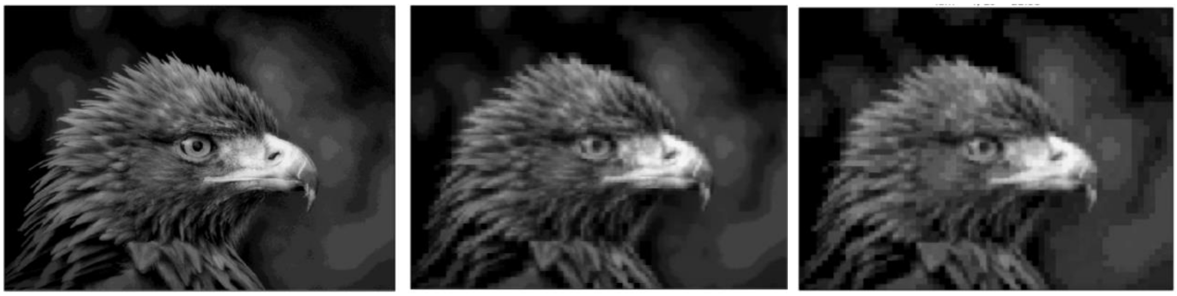


**Figure S24** | Architecture of the digital CMOS system, with 1,600 4b×4b multipliers, 8 per column; 1,600 adders, 8 per column; and 8 multiplication and accumulation (MAC) per clock cycle per column. Each MAC operation is pipelined to 3 stages. Latency: 16 clock cycles to complete one (1×100) × (100×200) vector-matrix dot-product operation.

**Table S1** | Equivalent digital CMOS design of a 100×200 crossbar using 40nm CMOS Technology

| | (1×100) × (100×200) operation |
|---|---|
| Number of Multiply-accumulate (MAC) | 1600 |
| Number of pipeline stages | 3 |
| Clock period (ns) | 0.65 |
| Latency (ns) | 10.4 |
| Power consumption (mW) | 274 |
| Silicon area (mm$^2$) | 0.306 |
| Power efficiency (TOPS/W) | 7.02 |
| Area efficiency (TOPS/mm$^2$) | 6.28 |

The comparison between the memristor solution and the digital solution is shown below in Fig. S25 and Table S2. The test is based on reconstructing 640×480 natural images at a rate of >24 images/second (*e.g.* real time processing of 480p videos).



Original Image Memristor solution Digital solution

**Figure S25** | Image reconstruction results based on a memristor system and an efficient digital approach.

**Table S2** | Performance comparison between the memristor solution and the digital solution

| | MSE | $L_0$ | Time | energy |
|---|---|---|---|---|
| Memristor | 1.933e-3 | 15.6% | 0.03607s | 719.0uJ |
| Digital | 2.226e-3 | 11.3% | 0.02636s | 11.82mJ |

As can be observed from the simulation results, the speed, error and sparsity parameters are similar for the digital solution and the memristor solution, by design. The digital solution resulted higher mean square error but lower $L_0$. These can be explained by the quantization effect of the digital approach. In the digital implementation, the synaptic weights were quantized into 4 bits (16 levels), which leads to a quantization error and subsequently slightly higher reconstruction error.

When it comes to power consumption, on the other hand, the memristor analog solution, based on parameters used in the current prototype devices, already demonstrates significant (~16×) improvement over the already very efficient digital solution. This is due to the fact that the vector-matrix dot-product is obtained in a single step by a parallel read process in the memristor system. To achieve the similar throughput in the digital system, eight 4b×4b multipliers were needed in a single column, leading to higher energy consumption. We expect the advantage of the memristor system will be even more pronounced for more complex tasks and with further optimized devices, since in the digital solution analysis we neglected the time and energy costs associated with moving data between the memory (not considered in this simple analysis) and the MAC circuitry, which will not be negligible for larger input data, while in the memristor system such data movement is not needed since the computation is directly performed in the same physical locations as the stored weights. The energy cost in the memristor system can also be significantly lowered further by optimizing the memristor devices, *i.e.* by using lower current memristor devices.

**References:**

[S1]: Rozell, C. J., Johnson, D. H., Baraniuk, R. G. & Olshausen, B. A. Sparse coding via thresholding and local competition in neural circuits. Neural Comput. 20, 2526–2563 (2008).

[S2]: Chang, Ting, et al. Synaptic behaviors and modeling of a metal oxide memristive device. Applied Physics A 102, 857-863 (2011).

[S3]: Olshausen, B.A. & Field, D.J. Sparse coding with an overcomplete basis set: A strategy employed by V1? Vision Research, 37, 3311–3325 (1997).

[S4]: Zylberberg, J., Murphy, J.T. & DeWeese, M.R. A sparse coding model with synaptically local plasticity and spiking neurons can account for the diverse shapes of v1 simple cell receptive fields. PLoS Comput. Biol. 7, e1002250 (2011).

The views expressed in this paper are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government. Approved for Public Release. Distribution unlimited. The authors acknowledge for helpful discussions.