ARTICLE

https://doi.org/10.1038/s41467-020-16108-9 OPEN

Accurate deep neural network inference using computational phase-change memory

Vinay Joshi[1,2], Manuel Le Gallo[1✉], Simon Haefeli[1,3], Irem Boybat[1,4], S. R. Nandakumar[1], Christophe Piveteau[1,3], Martino Dazzi[1,3], Bipin Rajendran[2], Abu Sebastian[1✉] & Evangelos Eleftheriou[1]

In-memory computing using resistive memory devices is a promising non-von Neumann approach for making energy-efficient deep learning inference hardware. However, due to device variability and noise, the network needs to be trained in a specific way so that transferring the digitally trained weights to the analog resistive memory devices will not result in significant loss of accuracy. Here, we introduce a methodology to train ResNet-type convolutional neural networks that results in no appreciable accuracy loss when transferring weights to phase-change memory (PCM) devices. We also propose a compensation technique that exploits the batch normalization parameters to improve the accuracy retention over time. We achieve a classification accuracy of 93.7% on CIFAR-10 and a top-1 accuracy of 71.6% on ImageNet benchmarks after mapping the trained weights to PCM. Our hardware results on CIFAR-10 with ResNet-32 demonstrate an accuracy above 93.5% retained over a one-day period, where each of the 361,722 synaptic weights is programmed on just two PCM devices organized in a differential configuration.

[1] IBM Research - Zurich, Säumerstrasse 4, 8803 Rüschlikon, Switzerland. [2] King's College London, Strand, London WC2R 2LS, UK. [3] ETH Zurich, Rämistrasse 101, 8092 Zurich, Switzerland. [4] Ecole Polytechnique Federale de Lausanne (EPFL), 1015 Lausanne, Switzerland. ✉email: anu@zurich.ibm.com; ase@zurich.ibm.com

Deep neural networks (DNNs) have revolutionized the field of artificial intelligence and have achieved unprecedented success in cognitive tasks such as image and speech recognition. Platforms for deploying the trained model of such networks and performing inference in an energy-efficient manner are highly attractive for edge computing applications. In particular, internet-of-things battery-powered devices and autonomous cars could especially benefit from fast, low-power, and reliably accurate DNN inference engines. Significant progress in this direction has been made with the introduction of specialized hardware for inference operating at reduced digital precision (4–8-bit), such as Google's tensor processing unit[1] and low-power graphical processing units such as NVIDIA T4[2]. While these platforms are very flexible, they are based on architectures where there is a physical separation between memory and processing units. The models are typically stored in off-chip memory, leading to constant shuttling of data between memory and processing units, which limits the maximum achievable energy efficiency.

In order to reduce the data transfers to a minimum in inference accelerators, a promising avenue is to employ in-memory computing using non-volatile memory devices[3–5]. Both charge-based storage devices, such as Flash memory[6], and resistance-based (memristive) storage devices, such as metal-oxide resistive random-access memory[7–10] and phase-change memory (PCM)[11–14] are being investigated for this. In this approach, the network weights are encoded as the analog charge state or conductance state of these devices organized in crossbar arrays, and the matrix-vector multiplications during inference can be performed in-situ in a single time step by exploiting Kirchhoff's circuit laws. The fact that these devices are non-volatile (the weights will be retained when the power supply is turned off) and have multi-level storage capability (a single device can encode an analog range of values as opposed to 1 bit) is very attractive for inference applications. However, due to the analog nature of the weights programmed in these devices, only limited precision can be achieved in the matrix-vector multiplications and this could limit the achievable inference accuracy of the accelerator.

One potential solution to this problem is to train the network fully on hardware[13–15], such that all hardware non-idealities would be de facto included as constraints during training. Another similar approach is to perform partial optimizations of the hardware weights after transferring a trained model to the chip[9,16,17]. The drawback of these approaches is that every neural network would have to be trained on each individual chip before deployment. Off-line variation-aware training schemes have also been proposed, where hardware non-idealities such as device-to-device variations[18,19], defective devices[19], or IR drop[18] are first characterized and then fed into the training algorithm running in software. However, these approaches would require characterizing and training the neural network from scratch for every chip. A more practical approach would be to have a single custom generic training algorithm run entirely in software, which would make the network immune to most of the hardware non-idealities, but at the same time would require only very little knowledge about the specific hardware it will be deployed on. In this way, the model would have to be trained only once and could be deployed on a multitude of different chips. To this end, several works have proposed to inject noise in the training algorithm to the layer inputs[20], synaptic weights[21], and pre-activations[22,23]. However, previous demonstrations have generally been limited to rather simple and shallow networks, and experimental validations of the effectiveness of the various approaches have been missing. We are aware of one recent work that analyzed more complex problems such as ImageNet classification[23], however the hardware model used was rather abstract and no experimental validation was presented.

In this work, we explore injecting noise to the synaptic weights during the training of DNNs in software as a generic method to improve the network resilience against analog in-memory computing hardware non-idealities. We focus on the ResNet convolutional neural network (CNN) architecture, and introduce a number of techniques that allow us to achieve a classification accuracy of 93.7% on the CIFAR-10 dataset and a top-1 accuracy of 71.6% on the ImageNet benchmark after mapping the trained weights to PCM synapses. In contrast to previous approaches, the noise injected during training is crudely estimated from a one-time all-around hardware characterization, and captures the combined effect of read and write noise without introducing additional noise-related training hyperparameters. We validate the training approach through hardware/software experiments, where each of the 361,722 weights of ResNet-32 is programmed on two PCM devices of a prototype chip, and the rest of the network functionality is simulated in software. We achieve an experimental accuracy of 93.75% after programming, which stays above 92.6% over a period of 1 day. To improve the accuracy retention further, we develop a method to periodically calibrate the batch normalization parameters to correct the activation distributions during inference. We demonstrate a significant improvement in the accuracy retention with this method (up to 93.5% on hardware for CIFAR-10) compared with a simple global scaling of the layers' outputs, at the cost of additional digital computations during calibration. Finally, we discuss our training approach with respect to other methods and quantify the trade-offs in terms of accuracy and ease of training.

Results

Problem statement. For our experiments, we consider two residual networks on two different datasets: ResNet-32 on the CIFAR-10 dataset, and ResNet-34 on the ImageNet dataset[24]. As shown in Fig. 1a, ResNet-32 consists of three different ResNet blocks with ten 3×3 kernels each, and is used to classify 32×32-pixel RGB images that belong to one out of ten classes (see Methods). The network contains 361,722 synaptic weights. The ResNet-34 network used for the 1000-class ImageNet dataset is shown in Supplementary Fig. 1. The main differences compared to ResNet-32 are the number and size of the ResNet blocks, and a larger number of input/output channels (see Methods).

The weights of all convolution layers along with the fully connected layer of ResNet-32 can be mapped on memristive crossbar arrays as explained in Fig. 1b[25]. Each synaptic weight can be mapped on a differential pair of memristive devices that are located on two different columns. For a given layer l, the synaptic weight W_{ij}^{l} of the (i, j)th synaptic element is represented by the effective synaptic conductance G_{ij}^{l} given by

$$G_{ij}^{l} = G_{ij}^{l,+} - G_{ij}^{l,-}, \tag{1}$$

where $G_{ij}^{l,+}$ and $G_{ij}^{l,-}$ are the conductance values of the two devices forming the differential pair. Those device conductance values are defined as the effective conductance perceived in the operation of a non-ideal memristive crossbar array, and therefore include all the circuit non-idealities from the crossbar and peripheral circuitry.

The mapping between the synaptic weight W_{ij}^{l} obtained after software training and the corresponding synaptic conductance is given by

$$G_{ij}^{l} = W_{ij}^{l} \times \frac{G_{\max}}{W_{\max}^{l}} + \delta G_{ij}^{l} = G_{\mathrm{T},ij}^{l} + \delta G_{ij}^{l}, \tag{2}$$

where $G_{\max}$ is the maximum reliably programmable device conductance and $W_{\max}^{l}$ is the maximum absolute synaptic weight

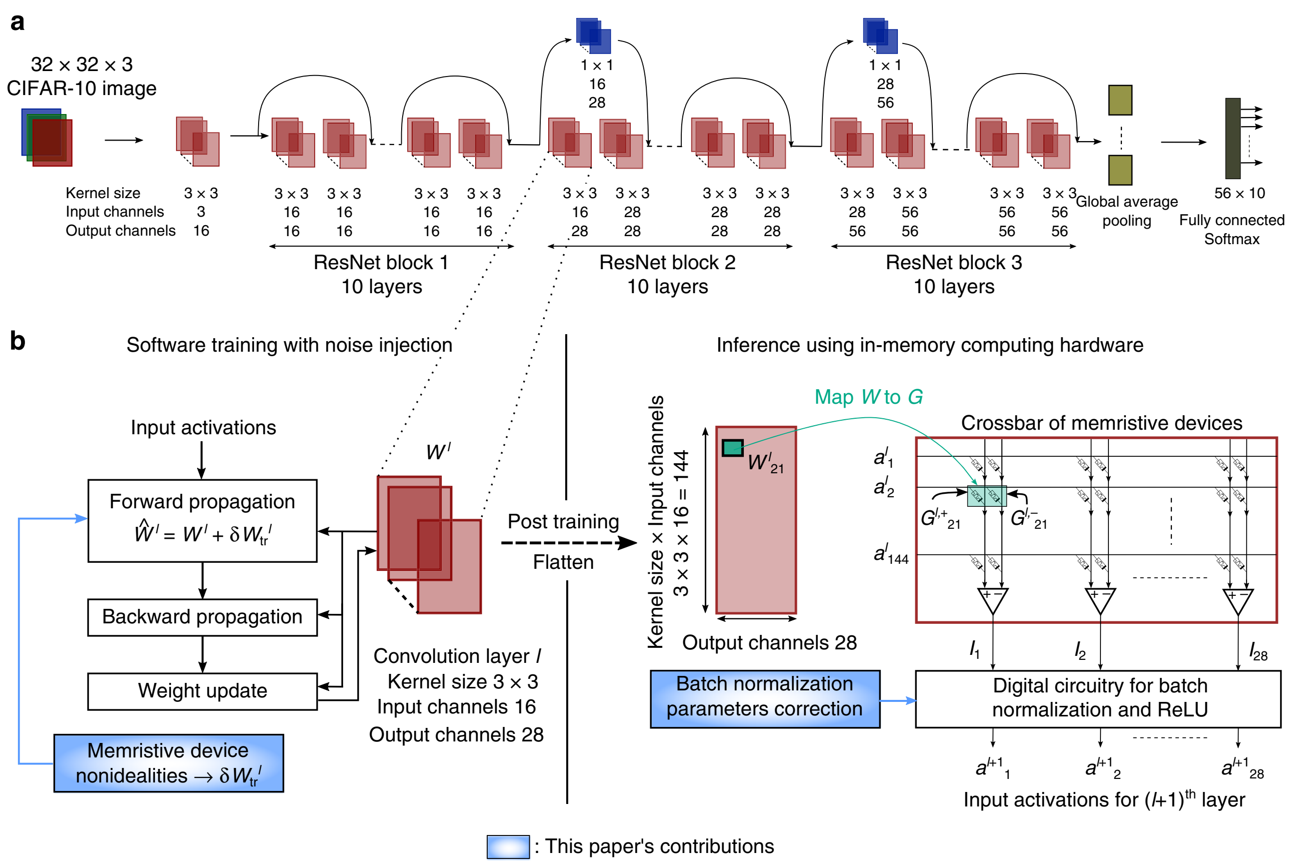

Fig. 1 Training and inference methodology. a ResNet-32 network architecture for CIFAR-10 image classification. The architecture of ResNet-32 used in this study is a slightly modified version of the original implementation[24] with fewer input and output channels in ResNet blocks 2 and 3. This allows each layer to fit on a single crossbar array of size 512 × 512 in an eventual computational memory accelerator. **b** Training and inference of an example layer of ResNet-32 according to the methodology proposed in this paper. Software training is performed by injecting a random noise term to the weights used during the forward propagation, δW^l_{tr}, which is representative of the combined read and write noise of the memristive devices used during inference (see section Training procedure). When transferring the weights of a convolution layer to memristive crossbars for inference, they are flattened into a 2-D matrix by collapsing each filter into a single vector programmed on a crossbar column, and stacking all filters on separate columns[25]. The weights are then programmed as the differential conductance of two memristive devices. Input activations a^l are applied as voltages on the crossbar rows. The output current from the column containing the G^- devices is subtracted from the one from the column containing the G^+ devices. The differential current output I from the crossbar then routes to a digital circuitry that performs batch normalization and the corresponding rectified linear unit (ReLU) activation function, in order to obtain the input activations for the next layer a^{l+1}. The final softmax activation function can be performed off-chip if required. An optional correction of the batch normalization parameters can be periodically performed to improve the accuracy retention over time (see sections Hardware/software inference experiment on CIFAR-10 and Adaptive batch normalization statistics update). The input image is padded with zero values at the border to ensure that the convolution operation preserves the height and width of the image. Therefore, considering an input image of size $n \times n$, the convolution operation can be performed in n^2 matrix-vector multiplication cycles.

value of layer l. δG^l_{ij} represents the synaptic conductance error from the ideal target conductance value $G^l_{\mathrm{T},ij} = W^l_{ij} \times \frac{G_{\max}}{W^l_{\max}}$. δG^l_{ij} is a time-varying random variable that describes the effects of non-ideal device programming (inaccuracies associated with write) and conductance fluctuations over time (inaccuracies associated with read). Possible factors leading to such conductance errors include inaccuracies in programming the synaptic conductance to $G^l_{\mathrm{T},ij}$, $1/f$ noise from memristive devices and circuits, temporal conductance drift, device-to-device variations, defective (stuck) devices, and circuit non-idealities (e.g., IR drop).

Clearly, a direct mapping of the synaptic weights of a DNN trained with 32-bit floating point (FP32) precision to the same DNN with memristive synapses is expected to degrade the network accuracy due to the added error in the weights arising from δG^l_{ij}. For existing memristive technologies, the magnitude of δG^l_{ij} may range from 1 to 10% of the magnitude of $G^l_{\mathrm{T},ij}$[11], which in general is not tolerable by DNNs trained with FP32 without any constrains. Imposing such errors as constraints during training can be beneficial in improving the network accuracy. In fact, quantization of the weights or activations[26], and injecting noise on the weights[27], activations[28], or gradients[29] have been widely used as DNN regularizers during training to reduce overfitting on the training dataset[30,31]. These techniques can improve the accuracy of DNN inference when it is performed with the same model precision as during training. However, achieving baseline accuracy while performing DNN inference on a model which is inevitably different from the one obtained after training, as is the case for any analog in-memory computing hardware, is a more difficult problem and requires additional investigations.

Although a large body of efficient techniques to train DNNs with reduced digital precision has been reported[32,33], it is unlikely that such procedures can generally be applied as-is to analog in-memory computing hardware due to the random nature of δG^l_{ij}.

Since quantization errors coming from rounding to reduced fixed-point precision are not random, DNNs trained in this way are not a priori expected to be suitable for deployment on analog in-memory computing hardware. Techniques that inject random Gaussian noise during training are a much more natural fit to make the network robust to errors from analog in-memory computing hardware. As early as in 1994, it was shown that injecting noise on the synaptic weights during training enhances the tolerance to weight perturbations of multi-layer perceptrons, and the application of this technique to analog neural hardware was discussed[34]. Recent works have also proposed to apply noise to the layer inputs or pre-activations in order to improve the network tolerance to hardware noise[20,23]. In this work, we follow the original approach of Murray et al.[34] of injecting Gaussian noise to the synaptic weights during training. Next, we discuss different techniques that we introduced together with synaptic weight noise in order to improve the accuracy of inference on ResNet and achieve close to software-equivalent accuracy after transferring the weights to PCM hardware.

Training procedure. When performing inference with analog in-memory computing hardware, the DNN experiences errors primarily due to (i) inaccurate programming of the network weights onto the devices (write noise) and (ii) temporal fluctuations of the hardware weights (read noise). We can cast the effect of these errors into a single error term δG^l_{ij} that distorts each synaptic weight when performing forward propagation during inference. Hence, we propose to add random noise that corresponds to the error induced by δG^l_{ij} to the synaptic weights at each forward pass during training (see Fig. 1b). The backward pass and weight updates are performed with weights that did not experience this noise. We found that adding noise to the weights only in the forward propagation is sufficient to achieve close to baseline accuracy for a noise magnitude comparable to that of our hardware, and adding noise during the backward propagation did not improve the results further. For simplicity, we assume that δG^l_{ij} is Gaussian distributed, which is usually the case for analog memristive hardware. Weights are linearly mapped to the entire conductance range $G_{\max}$ of the hardware, hence the standard deviation $\sigma^l_{\delta W_{\mathrm{tr}}}$ of the Gaussian noise on weights to be applied during training, for a layer l, can be obtained as

$$\frac{\sigma^l_{\delta W_{\mathrm{tr}}}}{W^l_{\max}} \equiv \eta_{\mathrm{tr}} = \frac{\sigma_{\delta G}}{G_{\max}}, \tag{3}$$

where $\sigma_{\delta G}$ is a representative standard deviation of the combined read and write noise measured from hardware. During training, the weight distribution of every layer and hence $W^l_{\max}$ changes, therefore $\sigma^l_{\delta W_{\mathrm{tr}}}$ is recomputed after every weight update so that η_{tr} stays constant throughout training. We found this to be especially important in achieving good training convergence with this method.

Weight initialization can have a significant effect on DNN training[35]. Two different weight initializations can lead to completely different minima when optimizing the network objective function. The network optimum when training with additive noise could be closer to the FP32 training optimum than to a completely random initialization. So it can be beneficial to initialize weights from a pretrained baseline network and then retrain this network by injecting noise. A similar observation was reported for training ResNet with reduced digital precision[33]. For achieving high classification accuracy in our experiments, we found this strategy more helpful than random initialization.

The noise injected during training according to Eq. (3) is closely related to the maximum weight of a layer, and can thus grow uncontrollably with outlier weight values. Controlling the weight distribution in a desirable range can improve the network training convergence and make the mapping of weights to hardware with limited conductance range easier. We therefore clip the synaptic weights at layer l after every weight update in the range $[-\alpha\times\sigma_{W^l}, \alpha\times\sigma_{W^l}]$, where σ_{W^l} is the standard deviation of weights in layer l and α is a tunable hyperparameter. In our studies, $\alpha = 2.0$ and $\alpha = 2.5$ worked the best for ResNet-32 and ResNet-34, respectively.

DNN convergence accuracy, in general, is sensitive to the learning rate used during training. Since we initialize the network parameters from a baseline network, using the same learning rate scheduling as that of the baseline network does not guarantee accurate convergence. To choose appropriate learning rate scheduling for ResNet-32, we first forward propagate the training set on the pretrained baseline network with injected synaptic weight noise and note the resulting accuracy. We note the learning rate evolution starting from this accuracy in the baseline network training curve until convergence, and use the same learning rate evolution while retraining the network by injecting noise.

We performed simulations to characterize the inference performance after training incorporating the injection of Gaussian noise in conjunction with the techniques presented above. We computed the classification accuracy for different amounts of injected noise η_{tr} during training. We also show how the accuracy is affected when the inference weights are perturbed by a certain amount of relative noise $\eta_{\mathrm{inf}} \equiv \frac{\sigma^l_{\delta W_{\mathrm{inf}}}}{W^l_{\max}}$, where $\sigma^l_{\delta W_{\mathrm{inf}}}$ is the standard deviation of the noise injected to the weights of layer l before performing inference on the test dataset.

The test accuracy of ResNet-32 on CIFAR-10 obtained for different amounts of noise injected during training, without inducing any perturbation during inference ($\eta_{\mathrm{inf}} = 0$), is plotted in Fig. 2a. It can be seen that the training algorithm is able to achieve a test accuracy close to the software baseline of 93.87% with up to approximately $\eta_{\mathrm{tr}} = 8\%$. The tolerance of the networks trained with different amounts of η_{tr} to weight perturbations during inference, η_{inf}, is shown in Fig. 2b. For a given value of η_{inf}, in general, the highest test accuracy can be obtained for the network that has been trained with a comparable amount of synaptic weight noise, i.e. for $\eta_{\mathrm{tr}} \approx \eta_{\mathrm{inf}}$. The test accuracy for $\eta_{\mathrm{tr}} = \eta_{\mathrm{inf}}$ is shown in Fig. 2c. It can be seen that for up to $\eta_{\mathrm{inf}} = 5\%$, an accuracy within 0.5% of the software baseline is achievable. The impact of the weight initialization, clipping, and learning rate scheduling on the accuracy is shown in Supplementary Fig. 2. Not incorporating one of those three techniques results in at least 1% drop in test accuracy for $\eta_{\mathrm{tr}} = \eta_{\mathrm{inf}} = 3.8\%$.

The top-1 accuracy of ResNet-34 on ImageNet for $\eta_{\mathrm{tr}} = \eta_{\mathrm{inf}}$ is shown in Fig. 2d. Consistent with previous observations[23,33], we found that the network recovers high accuracy extremely quickly when retraining with additive noise due to quick updates of the batch normalization parameters (see Supplementary Note 1), and obtained satisfactory convergence after only 8 epochs. The accuracy on ImageNet is much more sensitive to the noise injected during training than for CIFAR-10, and when noise is injected on all layers, there is more than 0.5% accuracy drop from the baseline even down to ~1.2% relative noise. In the literature, many network compression techniques allow higher precision for the first and last layers, which are more sensitive to noise[33,36]. We applied the same simplification to our problem, which means that we removed the noise during training on the first convolutional

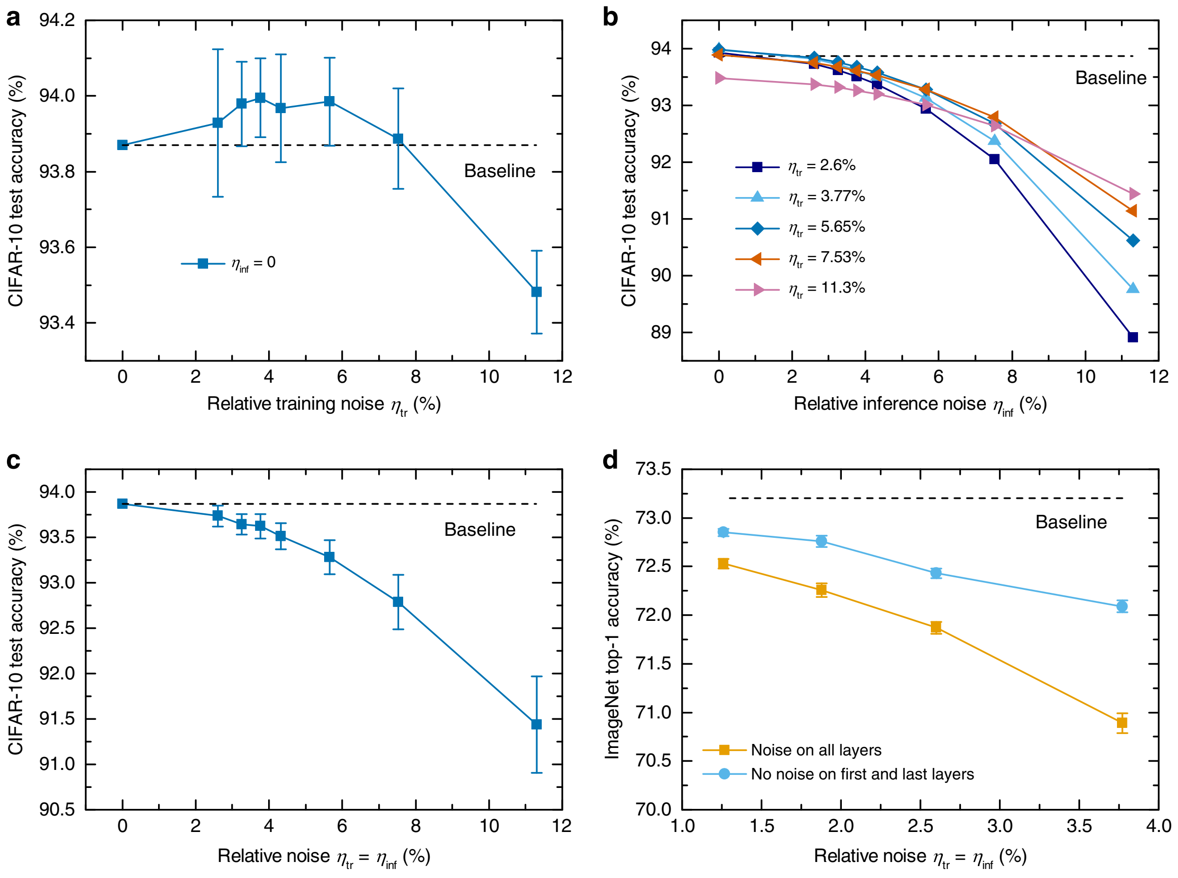

Fig. 2 Impact of injecting weight noise during training and inference on network accuracy. a Test accuracy on CIFAR-10 obtained for different amounts of relative injected weight noise during training η_{tr} without inducing any perturbation during inference ($\eta_{inf} = 0$). When $\eta_{tr} > 8\%$, the training convergence starts to become affected by the high noise and it is not possible anymore to reach the software baseline within the same number of epochs. The error bars represent the standard deviation over 10 training runs. **b** Test accuracy on CIFAR-10 obtained for the networks trained with different amounts of relative weight noise η_{tr} as a function of the weight noise injected during inference η_{inf}. In most cases, η_{tr} can be increased above η_{inf} up to a certain point and still lead to comparable or slightly higher (within ≈0.1%) test accuracy than for $\eta_{tr} = \eta_{inf}$. However, when η_{tr} becomes much higher than η_{inf}, the test accuracy decreases due to the inability of the network to achieve baseline accuracy when $\eta_{tr} > 8\%$. Each data point represents the average over 10 training runs and 100 inference runs. **c** Test accuracy on CIFAR-10 as a function of $\eta_{tr} = \eta_{inf}$. The error bars represent the standard deviation over 100 inference runs averaged over 10 training runs. **d** Top-1 accuracy on ImageNet as a function of $\eta_{tr} = \eta_{inf}$. The error bars represent the standard deviation over 10 inference runs on a single training run.

layer and the last dense layer, and performed inference with the first and last layer without noise. The obtained accuracy after training, by injecting the same training and inference noise as previously, can be increased by more than 1% with this technique (see Fig. 2d).

Weight transfer to PCM-based synapses. In order to validate the effectiveness of the above training methodology, we performed experiments on a prototype multi-level PCM chip comprising 1 million PCM devices fabricated in 90 nm CMOS baseline technology[37]. PCM is a memristive technology, which records data in a nanometric volume of phase-change material sandwiched between two electrodes[38,39]. The phase-change material is in the low-resistive crystalline phase in an as-fabricated device. By applying a current pulse of sufficient amplitude (typically referred to as the RESET pulse) an amorphous region around the narrow bottom electrode is created via a melt-quench process. The device will be in a low conductance state if the high-resistive amorphous region blocks the current path between the two electrodes. The size of the amorphous region can be modulated in an almost completely analog manner by the application of suitable electrical pulses. Hence, a continuum of conductance values can be programmed in a single PCM device over a range of more than two orders of magnitude.

An optimized iterative programming algorithm was developed to program the conductance values in the PCM devices with high accuracy (see Methods). The experimental cumulative distributions of conductance values for 11 representative programmed levels, measured approximately 25 s after programming, are shown in Fig. 3a. The standard deviation of these distributions is extracted and fitted with a polynomial function of the target conductance (dashed lines in Fig. 3a) as shown in Fig. 3b. For all levels, we achieve a standard deviation less than 1.2 μS, which is more than two times lower than that reported in previous works on nanoscale PCM arrays for a similar conductance range[40,41].

We studied the impact of weight transfer to PCM synapses on the inference accuracy of networks trained with FP32 weights, 4-bit and ternary digital weights, and additive weight noise. The weight transfer to PCM is simulated by mapping each trained network weight to a synaptic conductance value G_{ij}^{l}, computed from Eq. (2), using $G_{max} = 25$ μS and the conductance standard deviation measured from hardware. The conductance error δG_{ij}^{l} in Eq. (2) is modeled as a Gaussian distributed random variable with zero mean and standard deviation given by the fitted curve

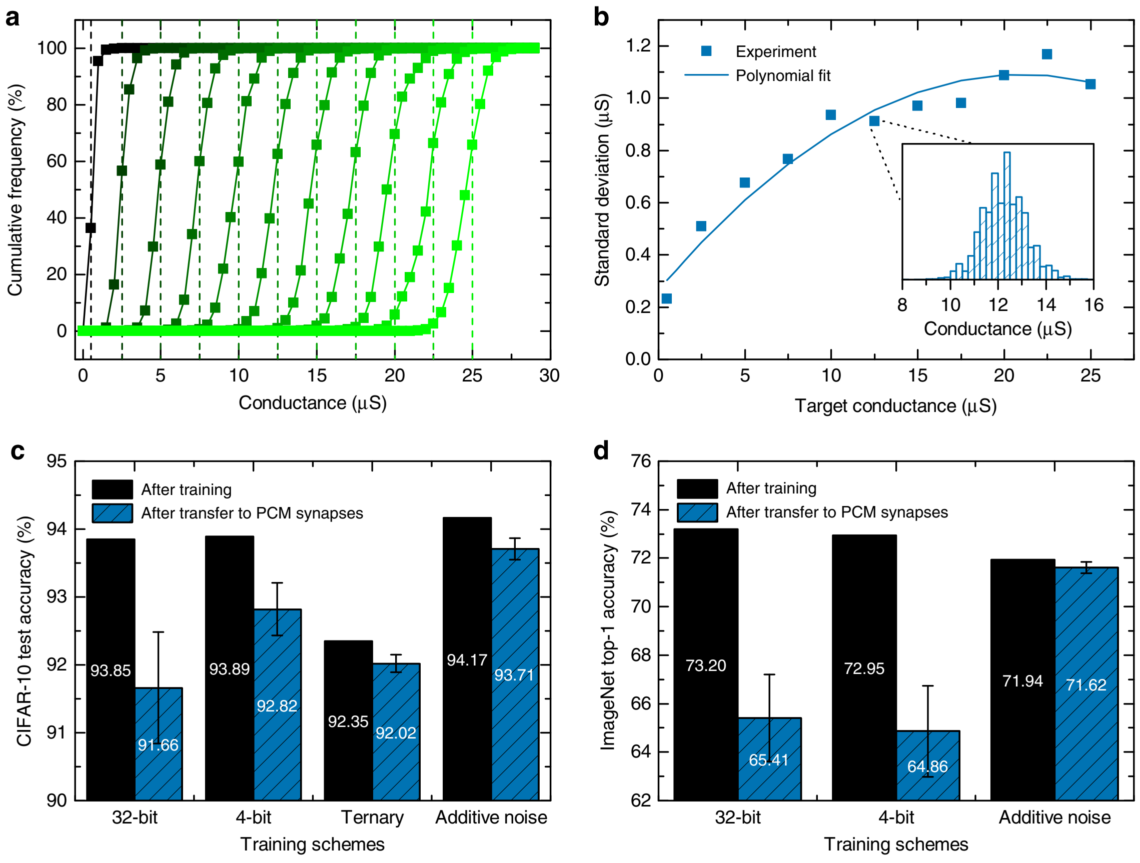

Fig. 3 Impact of weight transfer to PCM synapses on inference accuracy. a Cumulative distributions of 11 representative iteratively programmed conductance levels on 10,000 PCM devices per level. The vertical dashed lines denote the target conductance for each level. **b** Conductance standard deviation of the 11 levels as a function of target conductance. The inset shows a representative conductance distribution of one level. **c** Test accuracy on CIFAR-10 after software training and after weight transfer to PCM synapses for different training schemes. **d** Top-1 accuracy on ImageNet after software training and after weight transfer to PCM synapses for different training schemes. The weights of all layers of ResNet-34 are transferred to PCM synapses, including the first and the last. In **c** and **d** the error bars represent the standard deviation over 10 inference runs.

of Fig. 3b for its corresponding target conductance $|G^{l}_{\mathrm{T},ij}|$. This procedure accurately reproduces the experimental conductance values obtained when iteratively programming any conductance level within the range of 0–25 μS. For all training schemes, only one differential pair is used to encode a network weight, and we do not perform any additional quantization of the weights before mapping them to PCM conductance values.

The resulting test accuracy obtained after software training and after weight transfer to PCM synapses for ResNet-32 on CIFAR-10 is shown in Fig. 3c. It can be seen that the standard FP32 training without constraints performs the worst after transfer to PCM synapses. Training with 4-bit digital weights (using the method described in the ref. [33]), which is roughly the effective precision of our PCM devices[40], improves the performance after transfer with respect to FP32, but nevertheless the accuracy decreases by more than 1% after transferring the 4-bit weights to PCM. Training ternary digital weights[42] leads to a lower performance drop (<0.5%) when transferring weights to PCM, although we were not able to reach the FP32 baseline with ternary weights on this network. Therefore the accuracy after transfer is worse than for the 4-bit weights. When performing training by injecting Gaussian noise with $\eta_{\mathrm{tr}} = 3.8\%$, corresponding to $\sigma_{\delta G} = 0.94$ μS (median of the 11 values reported in Fig. 3b), the best overall performance after transfer to PCM is obtained. The resulting accuracy of 93.7% is less than 0.2% below the FP32 baseline. A rather broad range of values of η_{tr} lead to a similar resulting accuracy (see Supplementary Fig. 3), demonstrating that η_{tr} does not have to be very precisely determined for obtaining satisfactory results on PCM. The accuracy obtained without perturbing the weights after training by injecting noise is slightly higher than the FP32 baseline, which could be attributed to improved generalization resulting from the additive noise training.

The top-1 accuracy for ResNet-34 on ImageNet after transfer to PCM synapses for different training procedures is shown in Fig. 3d. Training with additive noise increases the accuracy by approximately 6% on PCM compared with FP32 and 4-bit[33] training. The accuracy of 71.6% achieved with additive noise training on PCM is significantly higher than that reported in Fig. 2d with $\eta_{\mathrm{inf}} = 3.8\%$, which could be attributed to a high percentage of network weights mapped to low conductance values with lower standard deviation than the median of 0.94 μS.

Hardware/software inference experiment on CIFAR-10. In addition to the weight transfer simulations of the previous section, we performed a hardware/software experiment in which all weights of ResNet-32 were physically programmed on the PCM chip. This allows us to address the effect of imperfect yield and temporal conductance fluctuations on the accuracy experimentally. Although we could achieve good test accuracy after weight transfer to PCM synapses, an important challenge for any analog in-memory computing hardware is to be able to retain this accuracy over time. This is especially true for PCM due to the high $1/f$ noise experienced in these devices as well as temporal

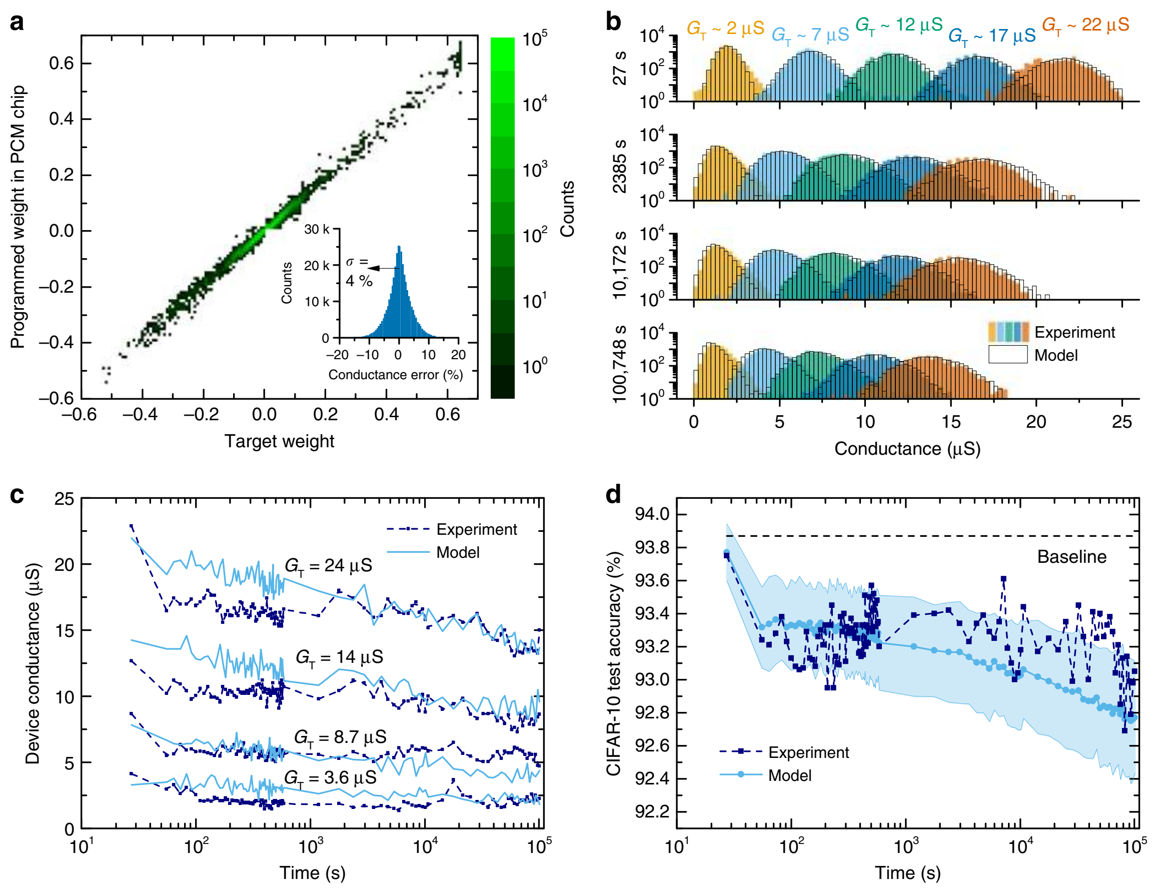

Fig. 4 Inference experiment with ResNet-32 on CIFAR-10. a Scatter plot of weights programmed in the PCM chip versus target weights obtained after training. The inset shows the distribution of the relative error between programmed and target synaptic conductance, $(G^l_{ij} - G^l_{\mathrm{T},ij})/G_{\max}$, and its standard deviation σ. **b** Distributions of programmed devices whose target conductance fall within 0.25 µS from five representative G_T values. The distributions are shown at four different times spanning the experiment duration of one day. The filled bars are the measured hardware data, the black lines are the PCM model. **c** Individual device conductance evolution over time of four arbitrarily picked devices from the chip programmed to four distinct G_T values, along with one PCM model realization for the same G_T values. **d** Measured test accuracy over time from the weights of the PCM chip, along with the corresponding PCM model match. A global drift compensation (GDC) procedure is performed for every layer before performing inference on the test set. The filled areas from the PCM model correspond to one standard deviation over 25 inference runs. The baseline refers to the FP32 software accuracy of 93.87%.

conductance drift. The conductance values in PCM drift over time t according to the relation $G(t) = G(t_0)(t/t_0)^{-\nu}$, where $G(t_0)$ is the conductance measured at time t_0 after programming and ν is the drift exponent, which depends on the device, phase-change material, and phase configuration of the PCM (ν is higher for the amorphous than the crystalline phase)[43]. In our PCM devices, $\nu \approx 0.06$ on average. Therefore, it is essential to measure experimentally how the test accuracy evolves over time during inference with PCM.

All 361,722 synaptic weights of ResNet-32 trained with $\eta_{\mathrm{tr}} = 3.8\%$ were programmed individually on two PCM devices of the chip. Depending on the sign of $G^l_{\mathrm{T},ij}$, either $G^{l,+}_{ij}$ or $G^{l,-}_{ij}$ was iteratively programmed to $|G^l_{\mathrm{T},ij}|$, and the other device was RESET close to 0 µS with a single pulse of 450 µA amplitude and 50 ns width. The iterative programming algorithm converged on 99.1% of the devices programmed to nonzero conductance, and no screening for defective devices on the chip was performed prior to the experiments. The scatter plot of the PCM weights measured approximately 25 s after programming versus the target weights W^l_{ij} is shown in Fig. 4a. After programming, the PCM analog conductance values were periodically read from hardware over a period of 1 day, scaled to the network weights, and reported to the software that performed inference on the test dataset (see Methods).

In addition to the experiment, we developed an advanced behavioral model of the hardware in order to precisely capture the conductance evolution over time during inference (see Supplementary Note 2). The model is built based on an extensive experimental characterization of the array-level statistics of hardware noise and drift[44]. Conductance drift is modeled using a Gaussian distributed drift exponent across devices, whose mean and standard deviation both depend on the target conductance state $|G^l_{\mathrm{T},ij}|$. Conductance noise with the experimentally observed $1/f^{1.21}$ frequency dependence is also incorporated with a magnitude that depends on the target conductance state and time. The model is able to accurately reproduce both the array-level statistics (see Fig. 4b) and individual device behavior (see Fig. 4c) observed over the duration of the experiment. Accurate modeling of all the complex dependencies of noise and drift as a function of time and conductance state was found to be very critical in being able to reproduce the experimental evolution of the accuracy on ResNet.

The resulting accuracy on CIFAR-10 over time is shown in Fig. 4d. The test accuracy measured 25 s after programming is 93.75%, which is very similar to the result obtained in Fig. 3c.

However, if nothing is done to compensate for conductance drift, the accuracy quickly decreases down to 10% (random guessing) within approximately 1000 s. This is because the magnitude of the PCM weights gradually reduces over time due to drift and this prevents the activations from properly propagating throughout the network. A simple global scaling calibration procedure can be used to compensate for the effect of drift on the matrix-vector multiplications performed with PCM crossbars. As proposed in ref. [40], the summed current of a subset of the columns in the array can be periodically read over time at a constant voltage. The resulting total current is then divided by the summed current of the same columns but read at time t_0. This results in a single scaling factor that can be applied to the output of the entire crossbar in order to compensate for a global conductance shift (see Methods and Supplementary Fig. 4). Since this factor can be combined with the batch normalization parameters, it does not incur any additional overhead when performing inference. This simple global drift compensation (GDC) procedure was implemented for every layer before carrying out inference on the test set, and the results are shown in Fig. 4d. It can be seen that GDC allows the retention of a test accuracy above 92.6% for 1 day on the PCM chip, and effectively prevents the effect of global weight decay over time as illustrated in Supplementary Fig. 4. A good agreement of the accuracy evolution between model and experiment is obtained, hence validating its use for extrapolating results over a longer period of time and for assessing the accuracy of larger networks that cannot fit on our current hardware.

Adaptive batch normalization statistics update. Although GDC can compensate for a global conductance shift across the array, it cannot mitigate the effect of $1/f$ noise and drift variability across devices. From the model, we observe that $1/f$ noise is responsible for the random accuracy fluctuations, whereas drift variability and its dependence on the target conductance state cause the monotonous accuracy decrease over time (see Supplementary Fig. 5). In order to improve the accuracy retention further, we propose to leverage the batch normalization parameters to correct the activation distributions during inference such that their mean and variance match those that were optimally learned during training. During inference, batch normalization is performed by normalizing the preactivations by their corresponding running mean μ and variance σ^2 computed during training. Then, scale and shift factors (γ and β) that were learned through back-propagation are applied to the normalized preactivations. Since γ and β are learnable parameters, it is not desirable to change them since it would require retraining the model on the PCM devices. However, updating μ and σ^2 is more intuitive, since the mean and variance of the preactivations are affected by noise and drift. Leveraging this idea, we introduce a new compensation technique called adaptive batch normalization statistics update (AdaBS), which improves the accuracy retention beyond GDC at the cost of additional computations during the calibration phase.

As described in Fig. 5a, the calibration phase consists in sending multiple mini-batches from a set of calibration images that come from the same distribution than the images seen during inference. In this study, we use the images from the training dataset as calibration images. The running mean and variance of preactivations are computed across the entire calibration dataset. The new values of μ and σ^2 computed during calibration are then used for subsequent inference. The main advantage of this technique is that it does not incur additional digital computations nor weight programming during inference, because we are only updating the batch normalization parameters μ and σ^2 when the calibration is performed. However, injecting the entire training dataset to compute μ and σ^2 in the calibration phase would bring significant overhead. When reducing the amount of injected images, the number of updates of the running statistics becomes smaller, and if the momentum used for computing μ and σ^2 is not properly tuned to account for this, the network accuracy decreases significantly. To tackle this issue, we developed a procedure to obtain the optimal momentum as a function of the number of mini-batches used for calibration (see Methods and Supplementary Note 3). With this method, we were able to reduce the number of calibration images down to 5.2% of the CIFAR-10 training dataset (2600 images) without affecting the accuracy. With that number of images, the overhead in terms of digital computations of the AdaBS calibration is about 52% of performing batch normalization during inference on the whole CIFAR-10 test set (see Supplementary Note 3). Furthermore, AdaBS requires additional off-chip memory resources to store the calibration images, leading to extra off-chip data transfers during calibration (see Supplementary Note 3). It may appear cumbersome to send so many images to the device to perform the calibration, however since it is only performed periodically over time when the device is idle and not every time an image is inferred by the network, the calibration cost can be amortized. The calibration overhead can be further reduced by using more efficient variants of batch normalization such as the L^1-norm version (see Supplementary Note 3). Moreover, although we used AdaBS (and GDC) to compensate solely for the drift of the PCM devices, the same procedure can be applied to mitigate conductance changes due to ambient temperature variations, a critical issue for any analog in-memory computing hardware.

The resulting accuracy when performing AdaBS on ResNet-32 with hardware weights before carrying out inference on the test set is shown in Fig. 5b. AdaBS allows to retain a test accuracy above 93.5% over one day, an improvement of 0.9% compared with GDC. This improvement becomes 1.8% for one year when extrapolating the results using the PCM model. This improvement results from optimally tuning scale and shift factors for each individual column of the crossbar with AdaBS, instead of a single scale factor for all columns in the case of GDC. Because of this more extensive parameter optimization, AdaBS can effectively compensate for drift variability and, more generally, random conductance variations across devices (see Supplementary Figs. 5–6).

We also applied AdaBS on the ImageNet classification task with ResNet-34, trained with $\eta_{\text{tr}} = 3.8\%$, using the PCM model to simulate the weight evolution for one year. By applying the same AdaBS method as for CIFAR-10 using only 0.1% of the ImageNet training dataset for calibration (1300 images), the accuracy after one year is increased by 7% compared with GDC when all layers are implemented with PCM synapses (see Fig. 5c). When the first and last layers are implemented in digital FP32, the initial accuracy increases to 71.9% and the retention is significantly improved. This technique, combined with AdaBS, allows the retention of an accuracy above 71% for one year. Drawbacks in efficiency when performing inference on hardware in this way have to be mentioned, but they stay limited given the small number of parameters and input size of the first and last layers (the first and last layers contain less than 3% of the network weights and are responsible for about 3.12% of the multiply-and-accumulate operations during inference with ResNet-34).

Discussion

Combined together, the strategies developed in this study allow us to achieve the highest accuracies reported so far with analog resistive memory on the CIFAR-10 and ImageNet benchmarks with residual networks close to their original implementation[24]. Although there is still room for improvement especially on

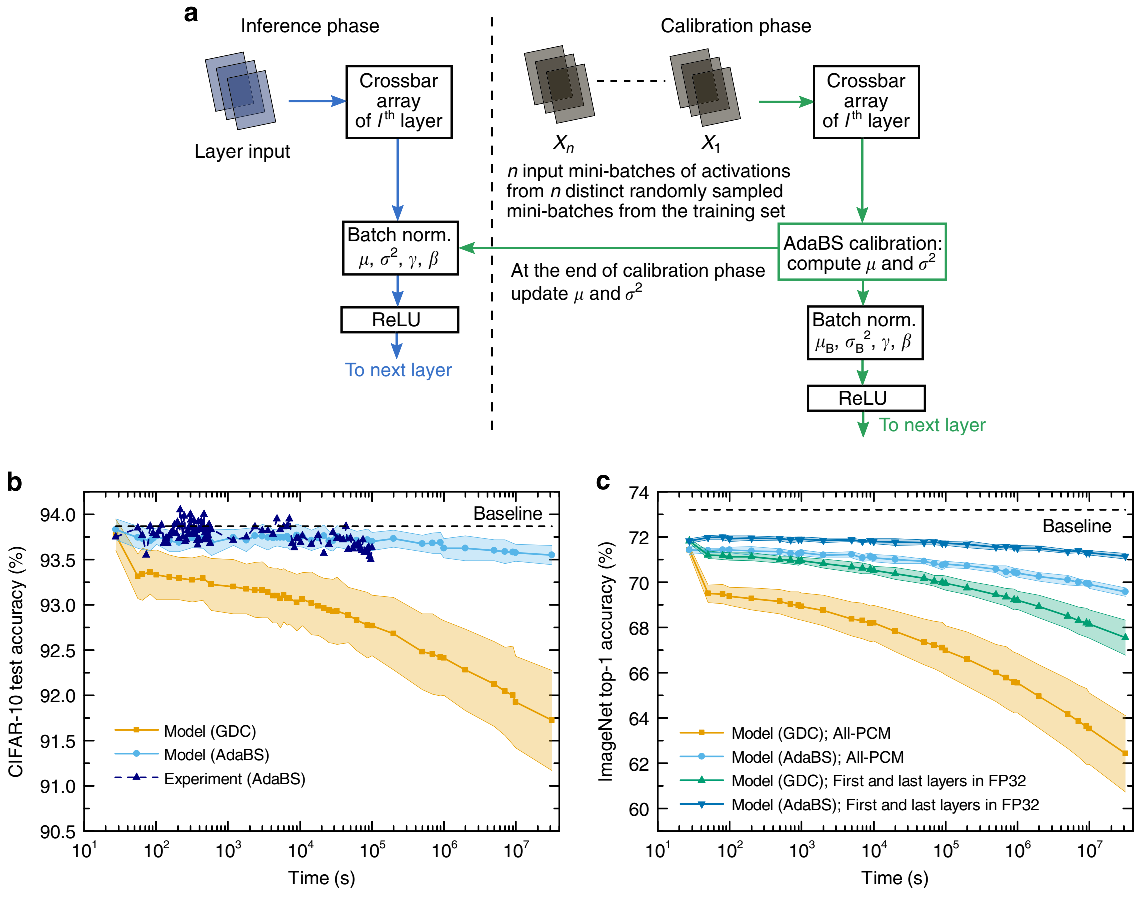

Fig. 5 Adaptive batch normalization statistics update with PCM synapses. a The AdaBS calibration procedure consists in updating the running mean μ and variance σ^2 parameters of the batch normalization performed in the digital unit of the in-memory computing hardware. The calibration is performed periodically when the device is idle, and after calibration the values of μ and σ^2 of every layer are updated for subsequent inference. Note that during the calibration phase, batch normalization is performed using the mini-batch mean μ_B and variance σ_B^2 instead of μ and σ^2 (see Methods and Supplementary Note 3). **b** Test accuracy of ResNet-32 on CIFAR-10 with GDC and AdaBS using the PCM model, along with experimental test accuracy obtained by applying AdaBS on ResNet-32 with the measured weights from the PCM chip. The filled areas from the PCM model correspond to one standard deviation over 25 inference runs. **c** Top-1 accuracy of ResNet-34 on ImageNet with GDC and AdaBS computed using the PCM model. Implementations using PCM synapses for all layers as well as first and last layers in digital FP32, and PCM synapses for all other layers, are shown. In the latter, no noise is applied on the first and last layers during training. The filled areas correspond to one standard deviation over 10 inference runs. In **b** and **c** the baseline refers to the FP32 software accuracy.

ImageNet, those accuracies are already comparable or higher than those reported on ternary weight networks[42], for example 71.6% top-1 accuracy of ResNet-34 on ImageNet with first layer in FP32[45]. Importantly, the accuracies we report are achieved with just two nanoscale PCM devices encoding the value of a weight. A common approach that could improve the accuracy further is to use multiple devices to encode different bits of a weight[3,41], at the expense of area and energy penalty, and additional support required by the peripheral circuitry. Aligned with previous observations[23,33], we notice that retraining ResNet with additive noise results mainly in adapting the batch normalization parameters, whereas the weights stay close to the full-precision weights trained without noise. Hence, retraining by injecting noise from a pretrained baseline network rather than from scratch is very effective since the network recovers high accuracy very quickly, especially for ImageNet. Although our experiments are not done on a fully-integrated chip that supports all functions of deep learning inference, the most critical effects of array-level variability, noise, and drift, are fully accounted for because each weight of the network is programmed on individual PCM devices of our array. Aspects of a fully-integrated chip that are not entirely captured in our experiments such as IR drop and additional circuit nonidealities such as offsets and noise have been studied in previous works and could be mitigated by additional retraining methods[18,20]. Experiments on fully-integrated memristor chips could be used to faithfully quantify their impact on the inference accuracy[7,9,46]. With respect to device yield, additional simulations showed that additive noise training also helps to mitigate the impact of devices stuck at low or high conductance on the inference accuracy (see Supplementary Fig. 7). Additional errors due to quantization coming from the crossbar data converters are analyzed further below.

There exist many different methods of training a neural network with noise that aim to improve the resilience of the model to analog mixed-signal hardware. These include injecting additive noise on the inputs of every layer[20], on the preactivations[22,23], or just adding noise on the input data[47]. Moreover, injecting multiplicative Gaussian noise to the weights[34] ($\sigma^l_{\delta W_{tr},ij} \propto |W^l_{ij}|$) is also defensible regarding the observed noise on the hardware. We analyzed the four aforementioned methods, attempting to reach the same accuracy demonstrated previously after weight transfer to PCM devices, to identify their possible benefits and drawbacks (see Supplementary Note 4). We found that it is possible to adjust the training procedure of all four methods to achieve a similar accuracy on CIFAR-10 after transferring the weights to PCM synapses. Somewhat surprisingly, even adding noise on the input data during training, which is just a simple form of data augmentation, leads to a model which is more resilient to weight perturbations during inference. This shows that it is not necessary to train a model with very complicated noise models that imitate the observed hardware noise precisely. As long as the data propagated through the network is corrupted by a Gaussian noise of the right magnitude, the model is expected to be robust to mapping on PCM devices. However, all four methods require one or multiple noise scaling factor hyperparameters to tune in order to reach satisfactory accuracy after transfer to PCM. In contrast, our proposed methodology estimates the additive noise to inject on the weights, η_{tr}, from a simple hardware characterization, avoiding any hyperparameter search for noise scaling factors. The value of η_{tr} does not have to be very precise either, because there are a range of values that lead to similar accuracy after transfer to PCM (see Supplementary Fig. 3). Moreover, we found that injecting noise on weights achieves better accuracy retention over time (see Supplementary Note 4), which suggests that weight noise mimics the behavior of the PCM hardware better.

A critical issue for in-memory computing hardware is the need for digital-to-analog (analog-to-digital) conversion every time data goes in (out) of the crossbar arrays. These data conversions lead to quantization of the activations and preactivations, respectively, which introduce additional errors in the forward propagation. Based on a recent ADC survey[23], 8-bit data conversion is a good tradeoff between precision and energy consumption. Hence, we analyzed the effect of quantizing the input and output of every layer of ResNet-32 and ResNet-34 to 8-bit on the inference accuracy. We set the input/output quantization ranges to the 99.995th percentile of the activation/preactivation distributions that are obtained when forward propagating 10 k randomly sampled images from the training dataset through the baseline network. As shown in Supplementary Fig. 8, even though the 8-bit quantization is not included in our training algorithm, the quantization has a minimal effect on the mean accuracy of ResNet-32 on CIFAR-10 (<0.05% drop) and ResNet-34 on ImageNet (<0.15% drop) after weight transfer to PCM synapses. The accuracy evolution over time, retaining the same quantization ranges, does not degrade significantly further and stays well within one standard deviation of that obtained without quantization. The small accuracy deviations could be potentially overcome by including the quantization in the retraining process, which will likely be necessary if less than 8-bit resolution is desired for higher energy efficiency.

Although a computational memory accelerates the matrix-vector multiplication operations in a DNN, communicating activations between computational memory cores executing different layers can become a bottleneck. This bottleneck depends upon two factors, (i) the way different layers are connected to each other and (ii) the latency of the hardware implementation to transfer activations from one core to another. Designing optimal interconnectivity between the cores for state-of-the-art deep CNNs is an open research problem. Indeed, having the network weights stationary during execution in a computational memory puts limits on what portion of the computation can be forwarded to different cores. This ultimately results in long-established hardware communication fabrics being ill-fit for the task. One topology for communication fabrics that is well-suited for computational memory is proposed by Dazzi et al.[48]. It is based on a 5 parallel prism (5PP) graph topology and facilitates inter-layer pipelined execution of CNNs[3]. The proposed 5PP topology allows the mapping of all the primary connectivities of state-of-the-art neural networks, including ResNet, DenseNet and Inception-style networks[48]. As discussed in ref. [48], the ResNet-32 implementation with 5PP can result in potentially 2× improvement in pipeline stage latency with similar bandwidth requirements compared with a standard 2D-mesh. Assuming that each layer is mapped onto a crossbar of size 512 × 512 having a computational cycle time of 100 ns, 8-bit activations, communication links with data rate of 5Gbps[49], and a pipeline scheme identical to that proposed in ref. [3], a single image inference latency of 52 μs and frame rate of 38,600 frames per second (FPS) for ResNet-32 on CIFAR-10 is estimated. The energy efficiency of a computational PCM core of this size implementing a layer of ResNet-32 is estimated to be 11.9 TOPS W^{-1} (tera operations per seconds per watt), with an area expenditure of approximately 0.57 mm^2 (see Supplementary Note 5). As an approximate comparison, YodaNN[50], a digital DNN inference accelerator for binary weight networks with ultra-low power budget, achieves 434.8 FPS in high throughput mode for a 9-layer CNN (BinaryConnect[51]) on CIFAR-10. Although not a direct comparison, the proposed topology and pipelined execution of ResNet-32 could result in 88× speedup, with a deeper network than the digital solution.

In summary, we introduced strategies for training ResNet-type CNNs for deployment on analog in-memory computing hardware, as well as improving the accuracy retention on such hardware. We proposed to inject noise to the synaptic weights in proportion to the combined read and write conductance noise of the hardware during the forward pass of training. This approach combined with judicious weight initialization, clipping, and learning rate scheduling, allowed us to achieve an accuracy of 93.7% on the CIFAR-10 dataset and a top-1 accuracy on the ImageNet benchmark of 71.6% after mapping the trained weights to analog PCM synapses. Our methods introduce only a single additional hyperparameter during training, the weight clip scale α, since the magnitude of the injected noise can be easily deduced from a one-time hardware characterization. After programming the trained weights of ResNet-32 on 723,444 PCM devices of a prototype chip, the accuracy computed from the measured hardware weights stayed above 92.6% over a period of 1 day, which is to the best of our knowledge the highest accuracy experimentally reported to-date on the CIFAR-10 dataset by any analog resistive memory hardware. A global scaling procedure was used to compensate for the conductance drift of the PCM devices, which was found to be critical in improving the accuracy retention. However, global scaling could not mitigate the effect of $1/f$ noise and drift variability across devices, which led to accuracy fluctuations and monotonous accuracy decrease over time, respectively. Periodically calibrating the batch normalization parameters before inference allowed to partly alleviate those issues at the cost of additional digital computations, increasing the 1-day accuracy to 93.5% on hardware. These results demonstrate the feasibility of realizing accurate inference on complex DNNs through analog in-memory computing using existing PCM devices.

Methods

Experiments on PCM hardware platform. The experimental platform is built around a prototype PCM chip that comprises 3 million PCM devices. The PCM array is organized as a matrix of word lines (WL) and bit lines (BL). In addition to the PCM devices, the prototype chip integrates the circuitry for device addressing, and for write and read operations. The PCM chip is interfaced to a hardware platform comprising a field programmable gate array (FPGA) board and an analog-front-end (AFE) board. The AFE board contains the digital-to-analog converters, and provides the power supplies as well as the voltage and current reference sources to the PCM chip. The FPGA board implements the data acquisition and the digital logic to interface with the PCM device under test and with all the electronics of the AFE board. The FPGA board is also used to implement the overall system control and data management as well as the interface with the host. The experimental platform is operated from a host computer, and a Matlab environment is used to coordinate the experiments.

The PCM devices are integrated into the chip in 90-nm CMOS technology using the key-hole process described in the ref. [52]. The phase-change material is doped $Ge_2Sb_2Te_5$. The bottom electrode has a radius of ~20 nm and a height of ~50 nm. The phase-change material is ~100 nm thick and extends to the top electrode, whose radius is ~100 nm. All experiments performed in this work are done on an array containing 1 million devices accessed via transistors, which is organized as a matrix of 512 WL and 2048 BL.

A PCM device is selected by serially addressing a WL and a BL. To read a PCM device, the selected BL is biased to a constant voltage (300 mV) by a voltage regulator via a voltage generated off chip. The sensed current is integrated by a capacitor, and the resulting voltage is then digitized by the on-chip 8-bit cyclic analog-to-digital converter (ADC). The total duration of applying the read pulse and converting the data with the ADC is 1 μs. The readout characteristic is calibrated via on-chip reference polysilicon resistors. To program a PCM device, a voltage generated off chip is converted on chip into a programming current. This current is then mirrored into the selected BL for the desired duration of the programming pulse.

Iterative programming involving a sequence of program-and-verify steps is used to program the PCM devices to the desired conductance values[53]. The devices are initialized to a high-conductance state via a staircase-pulse sequence. The sequence starts with a RESET pulse of amplitude 450 μA and width 50 ns, followed by six pulses of amplitude decreasing regularly from 160 to 60 μA and with a constant width of 1000 ns. After initialization, each device is set to a desired conductance value through a program-and-verify scheme. The conductance of all devices in the array is read five times consecutively at a voltage of 0.3 V, and the mean conductance of these reads is used for verification. If the read conductance of a specific device does not fall within 0.25 μS from its target conductance, it receives a programming pulse where the pulse amplitude is incremented or decremented proportionally to the difference between the read and target conductance. The pulse amplitude ranges between 80 and 400 μA, and the pulse width is 1000 ns. This program-and-verify scheme is repeated for a maximum of 55 iterations.

In the hardware/software inference experiments, the analog conductance values of the PCM devices encoding the network weights, $G_{ij}^{l,+}$ and $G_{ij}^{l,-}$, are serially read individually with the 8-bit on-chip ADC at predefined timestamps spaced over a period of one day. The read conductance values at every timestamp are reported to a TensorFlow-based software. This software performs the forward propagation of the CIFAR-10 test set on the weights read from hardware and computes the resulting classification accuracy. The drift compensation techniques, GDC and AdaBS, are performed entirely in software at every timestamp based on the conductance values read from hardware.

PCM-based deep learning inference simulator. We developed a simulation framework to test the efficacy of DNN inference using PCM devices. We chose Google's TensorFlow[54] deep learning framework for the simulator development. The large library of algorithms in TensorFlow enables us to use native implementation of required activation functions and batch normalization. Moreover, any regular TensorFlow code of a DNN can be easily ported to our simulator. As shown in Supplementary Fig. 9, custom made TensorFlow operations are implemented that generate PCM conductance values from the behavioral model of hardware PCM devices that was developed (see Supplementary Note 2). All the nonidealities including conductance range, programming noise, read noise, and conductance drift are implemented in TensorFlow following the equations shown in Supplementary Note 2. The simulator can also take the PCM conductance data measured from hardware as input, in order to perform inference on the hardware data. Data converters that simulate digital quantization of data at the input and output of crossbars are also implemented with tunable quantization ranges and precision. In this study, the data converters were turned off for all simulations except those presented in Supplementary Fig. 8. The drift correction techniques are implemented post quantization of the crossbar output.

Implementation of ResNet-32 training on CIFAR-10. ResNet-32 has 31 convolution layers with 3 × 3 kernels, 2 convolution layers with 1 × 1 kernels, and a final fully-connected layer. The network contains 361,722 synaptic weights. It consists of three different ResNet blocks with 10 3 × 3 kernels each. After the first convolution layer, there is a unity residual feed forward connection after every two convolution layers, except the 1 × 1 residual convolution connection to make output channels compatible between two layers. Each convolution layer is followed by batch normalization[55]. ReLU activation is used after every batch normalization except in case of residual connections, where the ReLU activation is computed after summation. Each residual connection with 1 × 1 convolution and first layer of ResNet blocks 2, 3 downsample the input by using a stride of 2 pixels. The 8 × 8 output of the last convolution layer is then downsampled to 1 × 1 resolution using global average pooling[56], which is followed by a single fully-connected layer. For the last fully-connected layer, no batch normalization is performed. The architecture of ResNet-32 used in this study is a slightly modified version of the original implementation[24] with fewer input and output channels in ResNet blocks 2 and 3. This network is trained on the well-known CIFAR-10 classification dataset[57]. It has 32 × 32 pixels RGB images that belong to one of the ten classes.

The network is trained on the 50,000 images of the training set, and evaluation is performed on the 10,000 images of the test set. The training is performed using stochastic gradient descent with a momentum of 0.9. The network objective is categorical cross entropy function over ten classes of the input image. Learning rate scheduling is performed to reduce learning rate by 90% at every 50th training epoch. The initial learning rate for the baseline network is 0.1 and training converges in 200 epochs with a mini-batch size of 128. Weights of all convolution and fully connected layers of the baseline network are initialized using He Normal[35] initialization. The baseline network is retrained by injecting Gaussian noise for up to 150 epochs with weight clip scale $\alpha = 2$. We preprocess the training images by randomly cropping a 32 × 32 patch after padding 2 pixels along the height and width of the image. We also apply a random horizontal flip on the images from the train set. Additionally, we apply cutout[58] on the training set images. For both training and test set, we apply channel wise normalization for 0 mean and unit standard deviation.

Implementation of ResNet-34 training on ImageNet. The architecture of the ResNet-34 network for ImageNet classification is derived from ref. [24]. It has 32 convolution layers with 3 × 3 kernels, 3 convolution layers with 1 × 1 kernels, a first convolution layer with 7 × 7 kernels and a final fully-connected layer. The network has 21,797,672 synaptic weights. The first convolution layer downsamples the input by using a stride of 2 pixels, followed by a maxpooling layer with kernel size of 3 × 3 and stride of 2 to downsample the feature maps to the resolution of 56 × 56 pixels. Each residual connection with 1 × 1 convolution and first layer of ResNet blocks 2, 3, 4 downsample the input by using a stride of 2 pixels. A global average pooling layer before the final fully-connected layer downsamples the 7 × 7 input to 1 × 1 resolution. The final fully-connected layer computes the output prediction corresponding to 1000 classes.

We trained ResNet-34 on the ImageNet[59] dataset. The ImageNet dataset has 1.3 M images in the training set and 50 k images in the test set. Images in the ImageNet dataset are preprocessed by following the same preprocessing steps as that of the Pytorch baseline model. Training images are randomly cropped to a 224 × 224 patch and then random horizontal flip is applied on the images. Channel wise normalization is performed on the images in both training and test set for 0 mean and unit standard deviation. Only for the test set, images are first resized to 256 × 256 using bilinear interpolation method and then a center crop is performed to obtain the 224 × 224 image patch.

The network objective function is softmax cross entropy on network output and corresponding 1000 labels. The network objective is minimized using the stochastic gradient descent algorithm with a momentum of 0.9. We obtained our baseline network architecture and its parameters from the Pytorch model zoo (https://pytorch.org/docs/stable/torchvision/models.html). We use this network to perform additive noise training by injecting Gaussian noise for a total of 10 training epochs. In contrast to ResNet-32 on CIFAR-10, no learning rate scheduling was performed since the network was trained only for 10 epochs with additive noise. We use mini-batch size of 400 and learning rate of 0.001 for the additive noise training simulations. We also use L2 weight decay of 0.0001 and weight clip scale of $\alpha = 2.5$ for the additive noise training.

Global drift compensation (GDC) method. The GDC[40] calibration phase consists of computing the summed current of L columns in each array encoding a network layer (see Supplementary Fig. 4). Those L columns contain devices initially programmed to known conductance values $G_{mn}(t_0)$. By reading those column currents, I_m, periodically with applied voltage V_{cal} on all N rows, we can compensate for a global conductance shift in the array during inference. When input data is processed by the crossbar during inference, the crossbar output can be scaled by $1/\hat{\alpha}$, where

$$\hat{\alpha} = \frac{\sum_{m=1}^{L} I_m}{V_{\mathrm{cal}} \sum_{n=1}^{N} \sum_{m=1}^{L} G_{mn}(t_0)}.$$

This procedure is especially simple because L can be chosen to be small, enough to get sufficient statistics. Moreover, $\hat{\alpha}$ is computed from the device data itself, without resorting to any assumption on how the conductance changes nor requiring extra timing information[60]. The term $V_{\mathrm{cal}} \sum_{n=1}^{N} \sum_{m=1}^{L} G_{mn}(t_0)$ needs to be computed only once, stored in the digital memory of the chip, and is reused for all calibrations. Reading the subset of L columns of the crossbar can be done while the PCM array is idle, i.e., when there are no incoming images to be processed by the device. Performing the L current summations can be implemented either with on-chip digital circuitry or in the control unit of the chip. At the end of the calibration phase, $1/\hat{\alpha}$ is computed and stored locally in digital unit of the crossbar. The output scaling by $1/\hat{\alpha}$ during inference can be combined with batch normalization because it is a linear operation. In our experiments, the calibration procedure was performed using all columns of each layer (e.g., L is equal to two times number of output channels) every time before inference is performed on the whole test set.

Adaptive batch normalization statistics update (AdaBS) technique. Batch normalization is performed differently in the training and inference phases of a DNN. During the training of a DNN, the input to the batch normalization operation x_i is normalized to zero mean and unit variance by computing its mean

(μ_B) and variance (σ_B^2) over a mini-batch of m images

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{4}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_B)^2. \tag{5}$$

The normalized input is then scaled and shifted by γ and β. During the training phase, γ and β are learned through backpropagation. In parallel, a global running mean (μ) and variance (σ^2) are computed by exponentially averaging μ_B and σ_B^2 respectively, over all the training batches

$$\mu = p \cdot \mu + (1 - p) \cdot \mu_B \tag{6}$$

$$\sigma^2 = p \cdot \sigma^2 + (1 - p) \cdot \sigma_B^2, \tag{7}$$

where p is the momentum. After training, the estimates of the global mean and variance μ and σ^2 are then used during the inference phase. When performing forward propagation during inference, the batch normalization coefficients μ, σ^2, γ, and β are used for normalization, scale, and shift.

The calibration phase of AdaBS consists in recomputing and updating μ and σ^2 for every layer where batch normalization is present. We recompute μ and σ^2 by feeding a randomly sampled set of mini-batches from the training dataset. In recomputing μ and σ^2, hyper-parameters such as mini-batch size (m) and momentum (p) need to be carefully tuned to achieve the best network accuracy.

For AdaBS calibration, we observed that using an optimal value of the momentum is necessary to achieve good inference accuracy evolution over time. For this, we have developed an algorithm to estimate the optimal value of momentum by an empirical analysis, which is explained in Supplementary Note 3. Based on this analysis, the formula we used to compute the optimal momentum as a function of the number of injected mini-batches n is

$$p = 0.015^{(1/n)}. \tag{8}$$

Using Eq. (8) to compute the momentum, we found that with a fixed mini-batch size of $m = 200$ images, it is sufficient to inject $n = 13$ mini-batches for the AdaBS calibration of the ResNet-32 network, that is approximately 5% of the CIFAR-10 training set (2600 images). The sensitivity of the accuracy to the number of images used for AdaBS calibration is shown in Supplementary Note 3. For ResNet-34 on ImageNet, we used mini-batch size of $m = 50$ and $n = 26$ mini-batches, that is 0.1% of the ImageNet training set (1300 images). In the experiments presented in Fig. 5, AdaBS calibration was performed for every layer before performing inference on the test set, except the last layer because it does not have batch normalization.

Data availability

The data that support the findings of this study are available from the corresponding authors upon request.

Code availability

The code used to generate the results of this study is proprietary to IBM.

Received: 1 November 2019; Accepted: 3 April 2020;
Published online: 18 May 2020

References

1. Jouppi, N. P. et al. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)* 1–12 (IEEE, 2017).
2. Jia, Z., Maggioni, M., Smith, J. & Scarpazza, D. P. NVidia turing T4 GPU via microbenchmarking. Preprint at https://arxiv.org/abs/1903.07486 (2019).
3. Shafiee, A. et al. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)* 14–26 (IEEE, 2016).
4. Sebastian, A., Le Gallo, M., Khaddam-Aljameh, R. & Eleftheriou, E. Memory devices and applications for in-memory computing. *Nat. Nanotechnol.* https://doi.org/10.1038/s41565-020-0655-z (2020).
5. Wang, Z. et al. Resistive switching materials for information processing. *Nat. Rev. Mater.* **5**, 173–195 (2020).
6. Merrikh-Bayat, F. et al. High-performance mixed-signal neurocomputing with nanoscale floating-gate memory cell arrays. *IEEE Trans. Neural Netw. Learn. Syst.* **29**, 4782–4790 (2018).
7. Chen, W.-H. et al. CMOS-integrated memristive non-volatile computing-in-memory for AI edge processors. *Nat. Electron.* **2**, 420–428 (2019).
8. Hu, M. et al. Memristor-based analog computation and neural network classification with a dot product engine. *Adv. Mater.* **30**, 1705914 (2018).
9. Yao, P. et al. Fully hardware-implemented memristor convolutional neural network. *Nature* **577**, 641–646 (2020).
10. Yin, S. et al. Monolithically integrated RRAM- and CMOS-based in-memory computing optimizations for efficient deep learning. *IEEE Micro* **39**, 54–63 (2019).
11. Le Gallo, M. et al. Mixed-precision in-memory computing. *Nat. Electron.* **1**, 246–253 (2018).
12. Boybat, I. et al. Neuromorphic computing with multi-memristive synapses. *Nat. Commun.* **9**, 2514 (2018).
13. Ambrogio, S. et al. Equivalent-accuracy accelerated neural-network training using analogue memory. *Nature* **558**, 60–67 (2018).
14. Nandakumar, S. R. et al. Mixed-precision deep learning based on computational memory. *Front. Neurosci.* **14**, 406 (2020).
15. Nandakumar, S. R. et al. Mixed-precision architecture based on computational memory for training deep neural networks. In *International Symposium on Circuits and Systems (ISCAS)* 1–5 (IEEE, 2018).
16. Mohanty, A. et al. Random sparse adaptation for accurate inference with inaccurate multi-level RRAM arrays. In *2017 IEEE International Electron Devices Meeting (IEDM)* 6–3 (IEEE, 2017).
17. Gonugondla, S. K., Kang, M. & Shanbhag, N. R. A variation-tolerant in-memory machine learning classifier via on-chip training. *IEEE J. Solid-State Circuits* **53**, 3163–3173 (2018).
18. Liu, B. et al. Vortex: variation-aware training for memristor X-bar. In *Proc. of the 52nd Annual Design Automation Conference* 1–6 (ACM, 2015).
19. Chen, L. et al. Accelerator-friendly neural-network training: Learning variations and defects in RRAM crossbar. In *Proc. of the Conference on Design, Automation & Test in Europe* 19–24 (European Design and Automation Association, 2017).
20. Moon, S., Shin, K. & Jeon, D. Enhancing reliability of analog neural network processors. *IEEE Trans. Very Large Scale Integr. Syst.* **27**, 1455–1459 (2019).
21. Miyashita, D., Kousai, S., Suzuki, T. & Deguchi, J. A neuromorphic chip optimized for deep learning and CMOS technology with time-domain analog and digital mixed-signal processing. *IEEE J. Solid State Circuits* **52**, 2679–2689 (2017).
22. Klachko, M., Mahmoodi, M. R. & Strukov, D. Improving noise tolerance of mixed-signal neural networks. In *International Joint Conference on Neural Networks (IJCNN)* 1–8 (IJCNN, 2019).
23. Rekhi, A. S. et al. Analog/mixed-signal hardware error modeling for deep learning inference. In *Proc. of the 56th Annual Design Automation Conference* 81:1–81:6 (ACM, 2019).
24. He, K., Zhang, X., Ren, S. & Sun, J. Deep residual learning for image recognition. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition* 770–778 (IEEE, 2016).
25. Gokmen, T., Onen, M. & Haensch, W. Training deep convolutional neural networks with resistive cross-point devices. *Front. Neurosci.* **11**, 1–22 (2017).
26. Merolla, P., Appuswamy, R., Arthur, J., Esser, S. K. & Modha, D. Deep neural networks are robust to weight binarization and other non-linear distortions. Preprint at https://arxiv.org/abs/1606.01981 (2016).
27. Blundell, C., Cornebise, J., Kavukcuoglu, K. & Wierstra, D. Weight uncertainty in neural networks. In *Proc. of the 32nd International Conference on Machine Learning, Vol. 37 of ICML'15* 1613–1622 (JMLR.org, 2015).
28. Gulcehre, C., Moczulski, M., Denil, M. & Bengio, Y. Noisy activation functions. *Proc. 33rd Int. Conf. Mach. Learn.* **48**, 3059–3068 (2016).
29. Neelakantan, A. et al. Adding gradient noise improves learning for very deep networks. Preprint at https://arxiv.org/abs/1511.06807 (2015).
30. An, G. The effects of adding noise during backpropagation training on a generalization performance. *Neural Comput.* **8**, 643–674 (1996).
31. Jim, K., Horne, B. G. & Giles, C. L. Effects of noise on convergence and generalization in recurrent networks. In *Proc. of the 7th International Conference on Neural Information Processing Systems* 649–656 (MIT Press, 1994).
32. Gupta, S., Agrawal, A., Gopalakrishnan, K. & Narayanan, P. Deep learning with limited numerical precision. In *Proc. of the 32nd International Conference on Machine Learning (ICML-15)* 1737–1746 (PMLR, 2015).
33. McKinstry, J. L. et al. Discovering low-precision networks close to full-precision networks for efficient embedded inference. Preprint at https://arxiv.org/abs/1809.04191 (2018).
34. Murray, A. F. & Edwards, P. J. Enhanced MLP performance and fault tolerance resulting from synaptic weight noise during training. *IEEE Trans. Neural Netw.* **5**, 792–802 (1994).
35. He, K., Zhang, X., Ren, S. & Sun, J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *IEEE International Conference on Computer Vision (ICCV)* 1026–1034 (IEEE, 2015).
36. Rastegari, M., Ordonez, V., Redmon, J. & Farhadi, A. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European*

Conference on Computer Vision (eds Leibe, B., Matas, J., Sebe, N. & Welling, M.) 525–542 (Springer, 2016).

37. Close, G. et al. Device, circuit and system-level analysis of noise in multi-bit phase-change memory. In *2010 IEEE International Electron Devices Meeting (IEDM)* 29–5 (IEEE, 2010).
38. Burr, G. W. et al. Recent progress in phase-change memory technology. *IEEE J. Emerg. Sel. Top. Circuits Syst.* **6**, 146–162 (2016).
39. Le Gallo, M. & Sebastian, A. An overview of phase-change memory device physics. *J. Phys. D Appl. Phys.* **53**, 213002 (2020).
40. Le Gallo, M., Sebastian, A., Cherubini, G., Giefers, H. & Eleftheriou, E. Compressed sensing with approximate message passing using in-memory computing. *IEEE Trans. Electron Devices* **65**, 4304–4312 (2018).
41. Tsai, H. et al. Inference of long-short term memory networks at software-equivalent accuracy using 2.5 m analog phase change memory devices. In *2019 Symposium on VLSI Technology*, T82–T83 (IEEE, 2019).
42. Li, F., Zhang, B. & Liu, B. Ternary weight networks. Preprint at https://arxiv.org/abs/1605.04711 (2016).
43. Le Gallo, M., Krebs, D., Zipoli, F., Salinga, M. & Sebastian, A. Collective structural relaxation in phase-change memory devices. *Adv. Electron. Mater.* **4**, 1700627 (2018).
44. Nandakumar, S. R. et al. Phase-change memory models for deep learning training and inference. In *26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)* 727–730 (IEEE, 2019).
45. Venkatesh, G., Nurvitadhi, E. & Marr, D. Accelerating deep convolutional networks using low-precision and sparsity. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* 2861–2865 (IEEE, 2017).
46. Cai, F. et al. A fully integrated reprogrammable memristor-CMOS system for efficient multiply-accumulate operations. *Nat. Electron.* **2**, 290–299 (2019).
47. Bishop, C. M. Training with noise is equivalent to Tikhonov regularization. *Neural Comput.* **7**, 108–116 (1995).
48. Dazzi, M. et al. 5 Parallel prism: a topology for pipelined implementations of convolutional neural networks using computational memory. In *Proc. NeurIPS MLSys Workshop* (NeurIPS, 2019).
49. Sacco, E. et al. A 5 Gb/s 7.1fJ/b/mm 8x multi-drop on-chip 10 mm data link in 14 nm FinFET CMOS SOI at 0.5 V. In *2017 Symposium on VLSI Circuits*, C54–C55 (IEEE, 2017).
50. Andri, R., Cavigelli, L., Rossi, D. & Benini, L. YodaNN: an architecture for ultralow power binary-weight CNN acceleration. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **37**, 48–60 (2017).
51. Courbariaux, M., Bengio, Y. & David, J.-P. Binaryconnect: training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems* (eds Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M. & Garnett, R.) 3123–3131 (Curran Associates, Inc., 2015).
52. Breitwisch, M. et al. Novel lithography-independent pore phase change memory. In *Proc. IEEE Symposium on VLSI Technology* 100–101 (IEEE, 2007).
53. Papandreou, N. et al. Programming algorithms for multilevel phase-change memory. In *Proc. International Symposium on Circuits and Systems (ISCAS)* 329–332 (IEEE, 2011).
54. Abadi, M. et al. *TensorFlow: large-scale machine learning on heterogeneous systems*, https://www.tensorflow.org/ (2015).
55. Ioffe, S. & Szegedy, C. Batch normalization: accelerating deep network training by reducing internal covariate shift. In *Proc. of the 32nd International Conference on Machine Learning, Vol. 37 of ICML'15* (ed. Bach, F. & Blei, D.) 448–456 (PMLR, 2015).
56. Zhou, B., Khosla, A., Lapedriza, A., Oliva, A. & Torralba, A. Learning deep features for discriminative localization. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2921–2929 (IEEE, 2016).
57. Krizhevsky, A., Nair, V. & Hinton, G. *The CIFAR-10 dataset*, https://www.cs.toronto.edu/kriz/cifar.html (2009).
58. DeVries, T. & Taylor, G. W. Improved regularization of convolutional neural networks with cutout. Preprint at https://arxiv.org/abs/1708.04552 (2017).
59. Russakovsky, O. et al. Imagenet large scale visual recognition challenge. *Int. J. Comput. Vis.* **115**, 211–252 (2015).
60. Ambrogio, S. et al. Reducing the impact of phase-change memory conductance drift on the inference of large-scale hardware neural networks. In *IEEE International Electron Devices Meeting (IEDM)*, 6.1.1–6.1.4 (IEEE, 2019).

Acknowledgements
We thank our colleagues at IBM Research—Zurich, IBM Research—Almaden and IBM TJ Watson Research Center, in particular, U. Egger, N. Papandreou, and A. Petropoulos for experimental help, M. BrightSky for help with fabricating the PCM prototype chip used in this work, and R. Khaddam-Aljameh, M. Stanisavljevic and M. Rasch for technical input. We also thank O. Hilliges for discussions. V.J. and B.R. were affiliated with New Jersey Institute of Technology, USA, at the time of writing this paper, and gratefully acknowledge the partial support from the university. This work was partially funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement number 682675).

Author contributions
V.J., M.L., S.H., C.P., and A.S. conceived the training methodology. V.J., M.L., S.H., I.B., and A.S. conceived the drift correction techniques. V.J. and S.H. performed the software training and inference simulations under the guidance of M.L. I.B. performed the PCM hardware experiments with the support of V.J. S.R.N. and V.J. developed the PCM model. V.J. and C.P. developed the PCM deep learning inference TensorFlow-based software. M.D. provided critical in-memory computing hardware insights and performed the ResNet-32 performance estimation. M.L. wrote the manuscript with input from all authors. M.L., A.S., B.R., and E.E. supervised the project.

Competing interests
The authors declare no competing interests.

Additional information
Supplementary information is available for this paper at https://doi.org/10.1038/s41467-020-16108-9.

Correspondence and requests for materials should be addressed to M.L.G. or A.S.

Peer review information *Nature Communications* thanks Huaqiang Wu and the other, anonymous, reviewer(s) for their contribution to the peer review of this work.

Reprints and permission information is available at http://www.nature.com/reprints

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0/.

© The Author(s) 2020