

Accelerating Deep Neural Networks with Phase-Change Memory Devices



Katie Spoon , Stefano Ambrogio , Prithish Narayanan , Hsinyu Tsai, Charles Mackin , An Chen , Andrea Fasoli , Alexander Friz, and Geoffrey W. Burr 

Abstract In this chapter, we discuss recent advances in the hardware acceleration of deep neural networks with analog memory devices. Analog memory offers enormous potential to speed up computation in deep learning. We study the use of Phase-Change Memory (PCM) as the resistive element in a crossbar array that allows the multiply-accumulate operation in deep neural networks to be performed in-memory. With this promise comes several challenges, including the impact of conductance drift on deep neural network accuracy. Here we introduce popular neural network architectures and explain how to accelerate inference using PCM arrays. We present a technique to compensate for conductance drift (“slope correction”) to allow in-memory computing with PCM during inference to reach software-equivalent deep learning baselines for a broad variety of important neural network workloads.

1 Introduction

Today’s world has a high demand for an ability to quickly make sense of a rapidly expanding flow of data [1]. In this context, machine learning techniques have become widely popular to help extract meaningful information from a variety of data such as images, text and speech [2]. In the last decade, the confluence of large amounts of labelled datasets, reliable algorithms such as stochastic gradient descent and increased computational power from CPUs (Central Processing Units) and GPUs (Graphics Processing Units) has enabled deep learning, a branch of machine learning, to revolutionize many fields, from image classification to speech recognition to language translation [3–6].

Neural networks typically have input and output layers, with many hidden layers in between. Each layer contains many neurons, where the output of each neuron

K. Spoon · S. Ambrogio (✉) · P. Narayanan · H. Tsai · C. Mackin · A. Chen · A. Fasoli · A. Friz · G. W. Burr
IBM Research-Almaden, 650 Harry Road, San Jose, CA 95120, USA
e-mail: stefano.ambrogio@ibm.com

K. Spoon
e-mail: katherine.spoon@colorado.edu

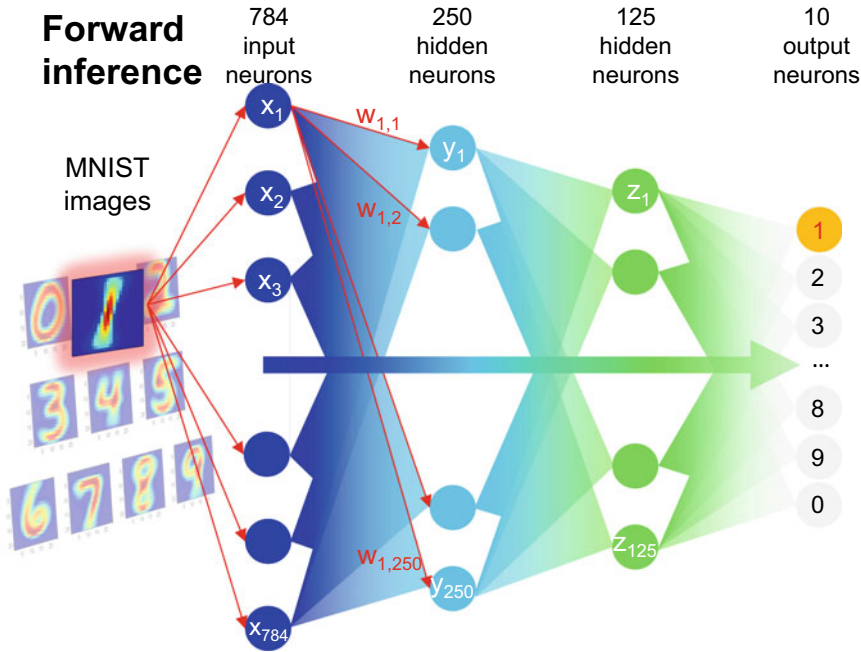
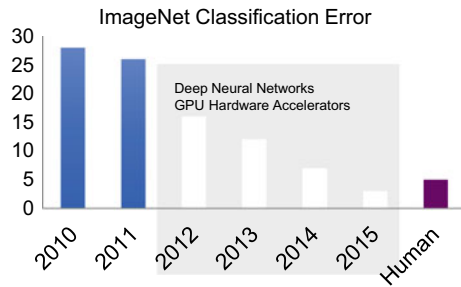


Fig. 1 A simple task is to classify images of handwritten digits from the MNIST dataset, where the network’s goal is to output the number the image contains. The input layer has 784 input neurons, the number of pixels in a 28×28 image. In the network shown, there are two hidden layers with 250 and 125 neurons, and a final layer with 10 output neurons, one for each possible classification (digits 0 through 9). There are connections between every pair of neurons in one layer to the next: $w_{1,1}$ is the weight from x_1 to y_1 , $w_{1,2}$ is the weight from from x_1 to y_2 , and so on. During forward inference, given an image, e.g. an image of a “one”, the example proceeds through the trained network, which should select the output “1” out of the 10 choices. Adapted with permission from [7]. Copyright 2017 IEEE

is determined by the weights feeding into that neuron, passed through a non-linear activation function [1]. The simplest neural network is a fully connected network, where every neuron in one layer is connected to every neuron in the next layer, as shown in Fig. 1. This is also referred to as a Multi-Layer Perceptron (MLP). In this particular example, a very simple dataset is used: the MNIST (Modified National Institute of Standards and Technology) database of handwritten digits [6]. Here, the goal is to train the network to recognize input digits using what is called a training dataset, followed by testing the network’s ability to classify previously unseen images from the test dataset [2].

While MNIST classification provides limited challenges, more complicated network architectures and datasets have recently been used in computer vision to solve more interesting and challenging tasks. One of the best-known is the ImageNet problem, where the goal is to classify real-world images into one thousand categories (cat, dog, etc.) based on their content [8]. By 2015, the best deep neural network was

Fig. 2 Neural networks existed long before 2015, however, the explosive switch to deep learning occurred due to the compute power and large data sets that became more widely available in recent years



making so few classification errors that it had effectively surpassed human ability, as shown in Fig. 2. This was largely due to the accelerated training of a large convolutional neural network using high performance GPUs [9]. The main advantage of GPUs resides in the highly parallel and efficient vector-matrix multiplication, which constitutes the core of neural network computations. This improvement has recently enabled the training of increasingly larger networks with millions or even billions of adjustable weights, providing increasingly higher accuracy classification and recognition performances [3, 10].

1.1 The Promise of Analog AI

Deep neural networks are attractive to hardware designers due to the nature of the core operation in a neural network: the vector-matrix product. Today this operation is typically performed by digital accelerators (i.e. GPUs), which are set up as shown in Fig. 3a, with the processor on one side and memory on the other, connected by a bus. This is known as the Von Neumann Architecture [11]. The bus can become a bottleneck since the data is sent back and forth from memory to processor, therefore a limited amount of data can be moved at any time in the communication bus [3, 10, 12]. Analog accelerators, on the other hand, perform computations directly in memory (Fig. 3b), which also behaves as a processor [13–19]. In the context of neural networks, analog accelerators offer a more natural implementation of fully connected layers. Since non-volatile memory (NVM) devices are organized in crossbar arrays, which provide a full connection between every input and every output, the mapping of fully connected layers into memory arrays becomes straightforward, thus increasing the density of programmable weights, the computation efficiency and the processing speed [13, 20].

How is the network mapped into memory? There are many different choices for the resistive element of the crossbar array, and in this work we use Phase Change Memory (PCM) [21, 22]. The fully connected neural network in Fig. 1 can be mapped onto crossbars arrays with the neurons stored in peripheral circuitry and the weights

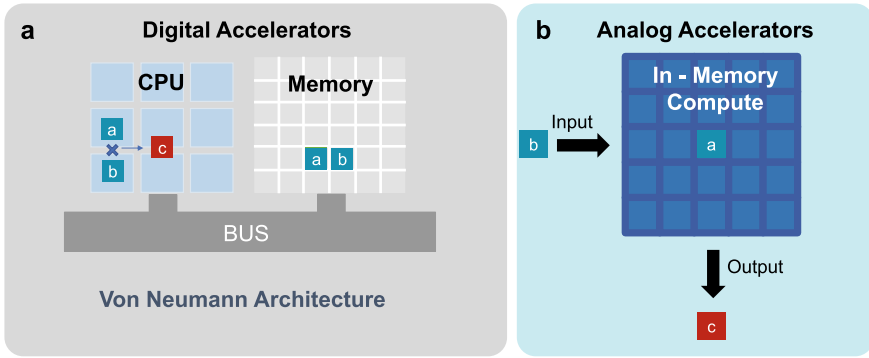


Fig. 3 Suppose we need to multiply two numbers, a and b . **a** In a typical setup, a and b start sitting in memory. They are both sent to the processor, and the answer c is computed and sent back to the memory. This consumes a lot of energy in data movement. Additionally, the bus can become a bottleneck. **b** Analog accelerators perform the computation in memory by storing a in the memory (crossbar array) and sending b in as a voltage, producing the answer c in-place

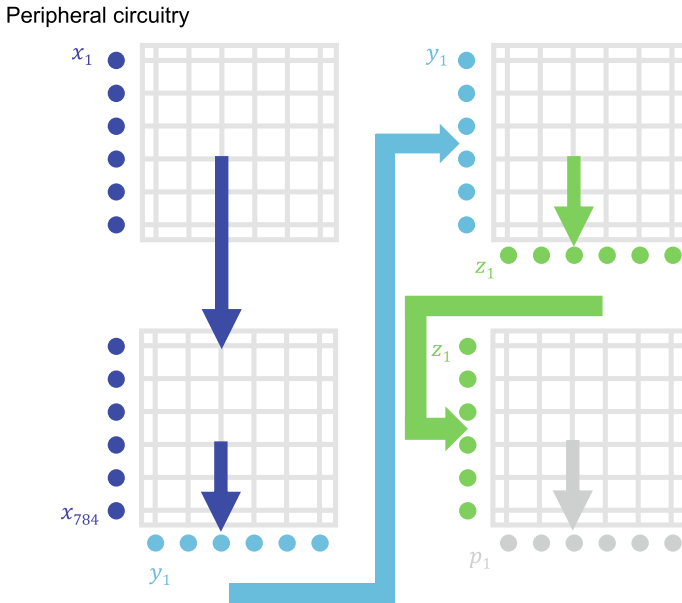


Fig. 4 The simple MLP from Fig. 1 has an input layer, two hidden layers, and an output layer. These can be mapped onto crossbar arrays of size 512×512 , for example. Since the input layer is 784×250 , two arrays are needed for the first layer, with 784 input neurons representing the rows and 250 hidden neurons representing the columns. There is an additional crossbar array for the 250×125 hidden layer, and a final array for the 125×10 output layer. The arrows represent the signal propagation through the network during forward inference

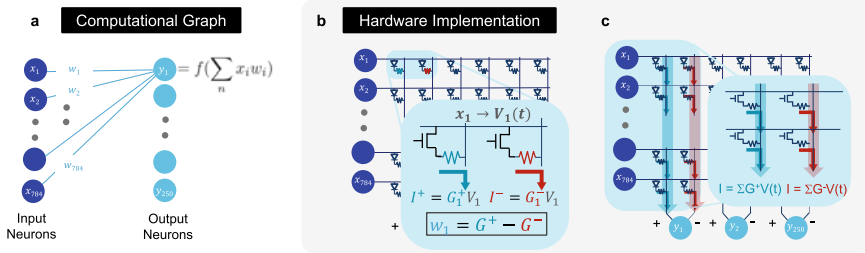


Fig. 5 **a** To compute the output of a neuron, for example, y_1 , the inputs x times the weights w are summed, then an activation function f is applied. This function could be the sigmoid mentioned in the discussion of the LSTM network in Sect. 1.2, or any other number of functions. **b** Ohm’s law is used to multiply $x * w$, where x is represented as a voltage $V(t)$, and (c) Kirchoff’s law is used to accumulate (sum) the results along the columns

of the neural network stored in the crossbar array (Fig. 4). During forward inference, the signal propagates through the crossbar arrays all the way to the output [16].

Figure 5 shows how a single crossbar array (for example, the first layer) is implemented in analog hardware. The essential calculation is the **multiply-accumulate operation** as shown in Fig. 5a. To perform this operation in hardware, the weights of the network are encoded as the difference between a pair of conductances $G+$ and $G-$ [13], which are analog tuned using proper programming schemes [20, 23]. Values of the input neuron x are encoded as voltages $V(t)$ and applied to the elements in a row. The multiplication operation between the x and w terms is performed through Ohm’s law (Fig. 5b). There are two possible ways to implement the input x , either by keeping the pulse-width constant and tuning the voltage amplitude, or by encoding the x value in the pulse duration and keeping the voltage amplitude constant. While the first scheme allows better time control since all pulses show identical duration, the second scheme helps counteract the read voltage non-linearity that all NVM devices experience, and prevents unwanted device programming for excessive read amplitudes. By Kirchoff’s current law, all of the product terms generated on a single column by all devices are accumulated in terms of aggregate current (Fig. 5c) to produce the final result at the bottom of the array in the form of an accumulated charge on a peripheral capacitor. Through this process, the multiply-accumulate operation, which is the most computationally expensive operation performed in a neural network, can be done in constant time, without any dependence on neural network layer size [13].

If these multiply-accumulate operations needed to be done with 32-bit or even 16-bit floating-point precision, this approach would probably not be feasible due to intrinsic noisy operations such as PCM read or write, exact weight programming or peripheral circuit conversion precision. But research on deep neural networks has shown that we can reduce the precision of weights and activations down to 8-bit integers, or even lower, without significant consequence to network accuracy [24]. For this reason, analog AI is a promising approach for these multiply-accumulate

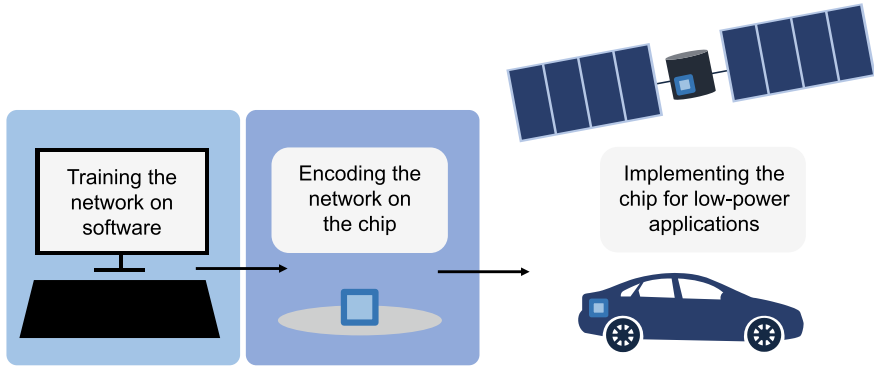


Fig. 6 The goal is to train a neural network in software, then encode those trained weights of the network onto a chip, then implement the chip for low-power applications, such as in a self driving car

computations in the context of deep neural networks, due to the intrinsic resilience that such networks provide to noisy operations [14].

There are two main stages of any deep learning pipeline: training and inference. During training, the network’s weights are updated, via back-propagation [2, 6], to better distinguish between different labeled data examples. Once the model is trained, it can be used to predict the labels of new examples during forward inference. There are hardware opportunities for both training and inference, but in this chapter we focus on the design of forward inference chips that can run quickly at low power (Fig. 6).

1.2 Two Common Neural Network Architectures

There are many applications of deep learning and, among those, computer vision and natural language processing are two of the most notable. It is helpful to use different types of neural networks depending on the structure of the input data, and, as an example, images and text have very different structures. Additionally, the MNIST handwritten digit task in Fig. 1 is very simple, and the most interesting problems today are difficult, requiring larger and more sophisticated networks. In this section we briefly review two of the most popular neural networks used in each of these two areas [25].

Computer Vision: Convolutional Neural Networks (CNNs) A slightly more challenging task than classifying handwritten digits is image classification into categories, like birds and dogs. A convolutional neural network (CNN) differs from the fully connected network in Fig. 1 by using convolutions to better process the image data. The network utilizes a set of filters that scan across the image, taking in a small amount of information at a time. The CNN shown in Fig. 7 has one convolutional

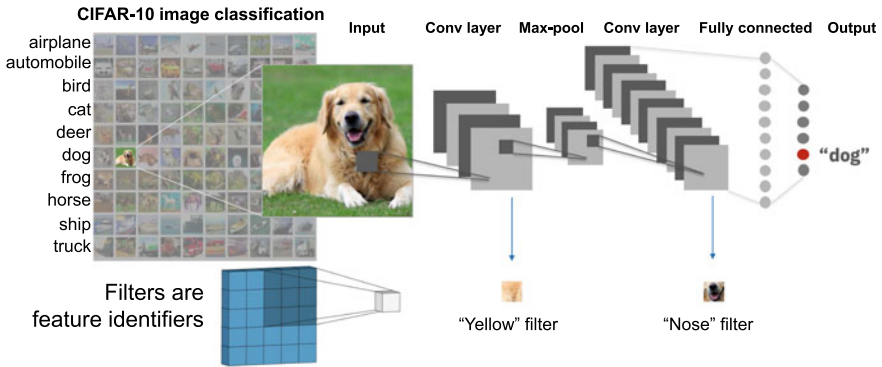


Fig. 7 In this example, the goal is to classify an image of a dog correctly out of the ten classes of images in the CIFAR10 dataset [26]. The input to the network is a tensor of 3 dimensions: [image-width, image-height, color-depth]. For instance, a single color image can be considered as three closely-related 2D images, one each for the red, green, and blue channels. The output from the network is a 1D vector, with one element for each class (dog, cat, horse, and so on)

layer followed by a max-pooling layer that reduces the dimensionality and determines the most important information from the previous layer, followed by another convolutional layer and a fully connected layer at the end to classify the image.

Each filter can be thought of as a feature identifier. For example, the earlier features might be the lines and colors of the dog, whereas later filters may represent very specific features, like noses or ears. It is important to note that these filters are **not designed by humans**; in a deep neural network, these filters are learned, as the filter weights are slowly updated during repeated exposure to the very large training dataset. These filters allow CNNs to successfully capture the spatial and temporal dependencies in the image, loosely mimicking the human vision system.

In reality, much deeper CNNs are used for challenging tasks. For example, ResNet [27] still has filter sizes of 3×3 , however instead of only 4 filters in the first layer, there are 64 filters for the first layer, 64 for the next, and so on (see Fig. 8).

Natural Language Processing: Recurrent Neural Networks (RNNs) To recognize text, different strategies are adopted, given the sequential nature of the input data. For example, assume we want to build a chatbot, that, given a query, will route the user to the correct category of answers. If a user asks “How do I request an account?”, the chatbot should classify the question as a “new account” question. This is a very different application from image classification. CNNs *could* be used for this problem, but it makes more sense to use a network that took advantage of the **sequential nature of text** as the input.

First, the sentence is split into words (tokens) to be sent through the network one by one (Fig. 9a). The first word is the first input to the hidden layer (Fig. 9b). This produces an output, similar to the fully connected network in Fig. 1. The second word is where the recurrence comes in. When “do” is sent into the network, the output is calculated using both the current hidden state as well as the output of the

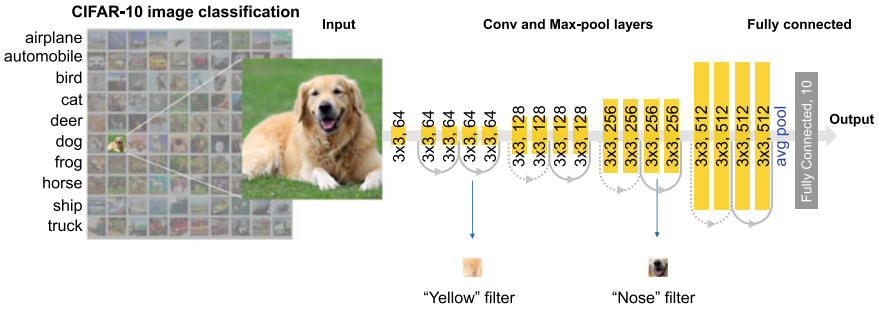


Fig. 8 ResNet is a much deeper network than the standard CNN in Fig. 7, so information can sometimes be lost in deeper layers. The arrows show residual connections, designed to help the information stay relevant deeper in the network (hence the “Res” in “ResNet”). One of the 64 early filters could be a “yellow” identifier, and one of the later filters could be a “ear” identifier, for example

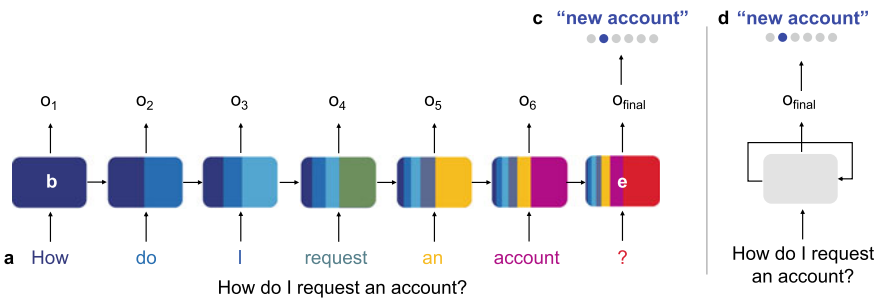


Fig. 9 An unrolled recurrent neural network (RNN) processing a single input sequence query is shown. **a** The sequence is split into words or tokens and they are sent into the network sequentially, starting by **b** sending the first word “How” through the hidden state and producing an output o_1 . On the next word, “do”, the previous hidden state and the current input produce the next output. This continues until the entire sequence has been processed, then **c** the final output though the fully connected layer produces the output class. The network has determined that this query, “How do I request an account?” is a “new account” question. **d** Currently the network is unrolled, but it can be represented rolled up, with an arrow representing the hidden state feeding back into itself before finishing the sequence. **e** Recurrent neural networks are useful for processing sequential data, but they can lose information over time. For example, in the last hidden state, information from the first hidden state is only represented by a small sliver

previous hidden state. This continues throughout inference of the sequence, until the final hidden state output is fed to a fully connected layer that classifies the query as a “new account” question (Fig. 9c).

Unfortunately, RNNs lose information in long sequences - they have “short-term memory”. The last hidden state illustrates this: it contains only a small sliver of the earlier words of the sentence, largely depending instead on the last word or symbol, in this case the question mark (Fig. 9e).

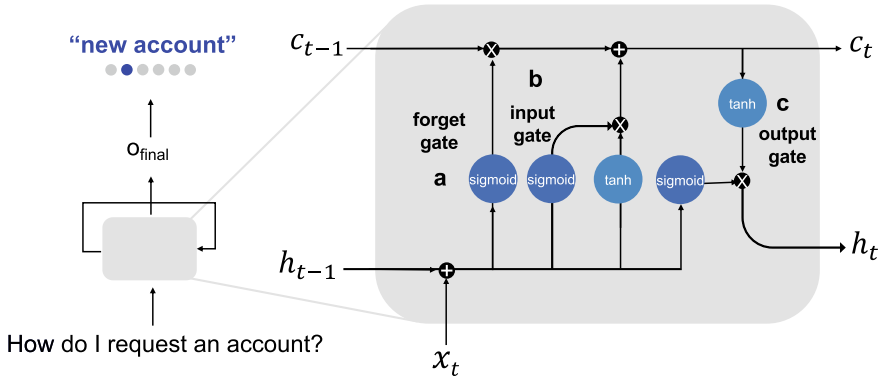


Fig. 10 An LSTM network is still a recurrent neural network (RNN), with the only difference being the structure of the hidden state. Here a closeup of the cell shows the various gates the input goes through before exiting the hidden state. For example, the first input x is “How”, for time step $t = 0$. The next word, “do” is $x_{t=1}$. There are three gates: **a the forget gate**, determining which information from the previous hidden state should be forgotten based on the current input, **b the input gate**, deciding which values to update in the cell state (maintained along the top), and finally **c the output gate**, producing the final hidden state value

To solve this problem, a modified RNN called a **Long Short-Term Memory Network (LSTM)** is often used. The hidden state is replaced by a more sophisticated cell that still unrolls for each new word, however, it also contains a sequence of gates that determine which information from the new input token and the old cell state are relevant and which should be thrown out. This adjustment has allowed for major advancements in natural language processing applications.

In a closeup of this cell in Fig. 10, instead of weighting both the current input and the old hidden state equally, three gates determine how much to keep or throw away.

Information is maintained along the top, in what is called the cell state, or c_t . The different gates determine how to update the cell state. The first gate is the forget gate (Fig. 10a), which uses a sigmoid function to process the input and previous hidden state and determine which information in the cell state should stay and what should be forgotten. For example, it could be that during training, the network has determined that the word “account” is really important for prediction, in which case the forget gate would output a 1, indicating to fully keep that information.

In general, the sigmoid function outputs numbers between zero and one, identifying which elements should be let through. A value of zero means “keep nothing”, whereas one means “keep everything”. A tanh function outputs numbers between -1 and 1 and is used to help regulate the values flowing through the network.

The input gate (Fig. 10b) determines which values to add to the cell state, and a tanh function applied to the same input generates the new candidate values that could be added based on the current input. To update the cell state, first the previous cell state c_{t-1} is multiplied by the forget gate output. Then the result is added to the input

gate times the candidate values provided by the tanh function. Last, the output gate (Fig. 10c) determines the output of the cell.

Unlike the multi-dimensional tensors in the image application, the cell-state, hidden-state, inputs and outputs are all one-dimensional vectors, and the multiplies and adds within the LSTM are performed in an element-wise fashion. As a result, a natural language processing system will often include an encoder—to turn words in English or another language into a vector of floating-point numbers—and a decoder, to convert each output vector into a set of predicted probabilities identifying what the most probable next word in the sentence is. By choosing the right encoder and decoder with careful training, an LSTM can be used for machine translation from one language to another.

2 Software-Equivalent Accuracy in Analog AI

While the implementation of CNNs and LSTMs using crossbar arrays of PCM devices can potentially provide fast and energy efficient operations, Phase Change Memory also presents many non-idealities that need to be corrected in order to reach software-equivalent accuracy with analog hardware.

2.1 Programming Strategies

To accurately perform neural network inference, weights must be precisely programmed by tuning the conductance of the devices. Write and read noise affect all NVM types, while PCM also experiences an additional non-ideality, **conductance drift**, namely a reduction of conductance over longer times, which degrades the computation precision, eventually decreasing the classification accuracy [28, 29]. Even small changes from the trained weights to the weights programmed on the chip can have a crucial impact on accuracy. Ideally, the relationship between programming pulse and achieved conductance state should be predictable, where a certain number and shape of pulses will always program a certain conductance value G . However, actual programming traces show a large variability, as can be seen from simulated traces in Fig. 11, with each device behaving slightly differently from the others (inter-device variability). Even a single device experiences different conductance traces under the same programming conditions (intra-device variability), potentially causing a drop in neural network accuracy [30].

In order to make the programming process more precise, a closed-loop tuning (CLT) procedure can be adopted, and a more complex unit cell is often used that contains not only one most significant pair (MSP) of conductances $G+$ and $G-$, but also includes a least significant pair $g+$ and $g-$ (see Fig. 12) [14, 23, 30]. This CLT operation consists of tuning the weights in four phases, one for each of the four conductances $G+$, $G-$, $g+$ and $g-$. To accelerate the write speed, programming can be

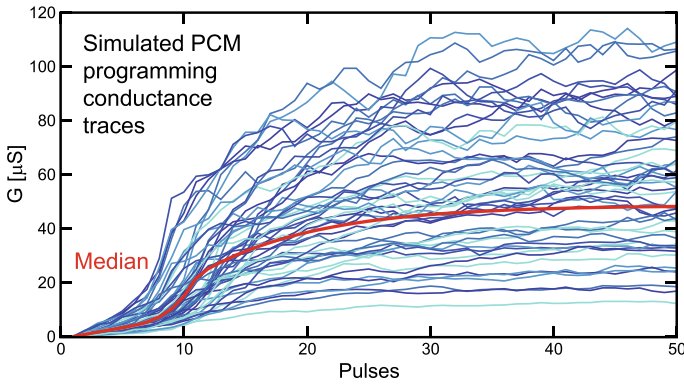


Fig. 11 Each simulated curve represents a different device programming behavior. Ideally we want to be able to predict and better control these trajectories. Adapted with permission from [23]. Copyright 2019 John Wiley and Sons

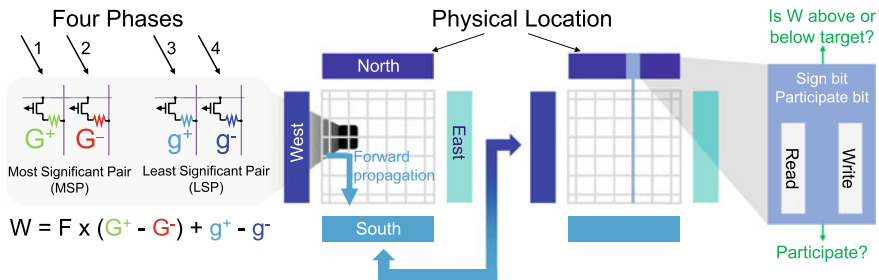


Fig. 12 The unit cell contains four conductance values, two for the most significant pair (MSP): $G+$ and $G-$, and two for the least significant pair (LSP): $g+$ and $g-$. The programming strategy focuses on each one of these values in turn, iteratively updating the weight value depending on how close it is to the target weight (the goal weight). Each column additionally contains two pieces of information: whether or not the current weight is above or below the target weight, along with whether or not this column will participate in the current programming phase. Adapted with permission from [23]. Copyright 2019 John Wiley and Sons

performed in a row-wise parallel fashion, tuning all weights in one row at the same time. To enable this, each weight is checked against the target after each read step. If a weight reaches, or overcomes, the target weight, further programming needs to be prevented. Each column contains a sign bit and a participation bit. The sign bit represents whether or not the full weight W is above or below the target weight in the current stage, and the participation bit represents whether that column should participate in the current phase or stop [23].

The error is defined as the difference between the actual programmed weight and the target weight. The error should go to 0 throughout the four phases. For positive (negative) target weights, Phase 1 starts by applying pulses to $G+$ ($G-$) until the weight exceeds the threshold. Then the process is repeated for $G-$ ($G+$), this time applying

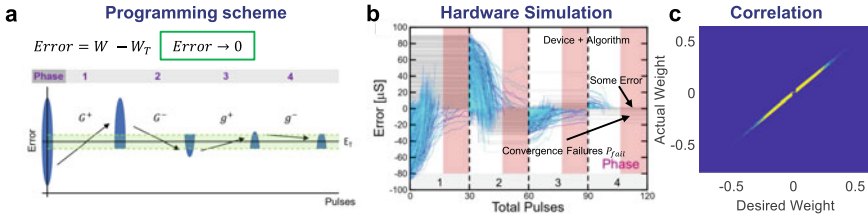


Fig. 13 a Iterative programming strategy that successively tunes the four conductance in a weight, b leading to c a strong correlation between the desired programmed weights and the actual programmed weights. Adapted with permission from[23]. Copyright 2019 John Wiley and Sons

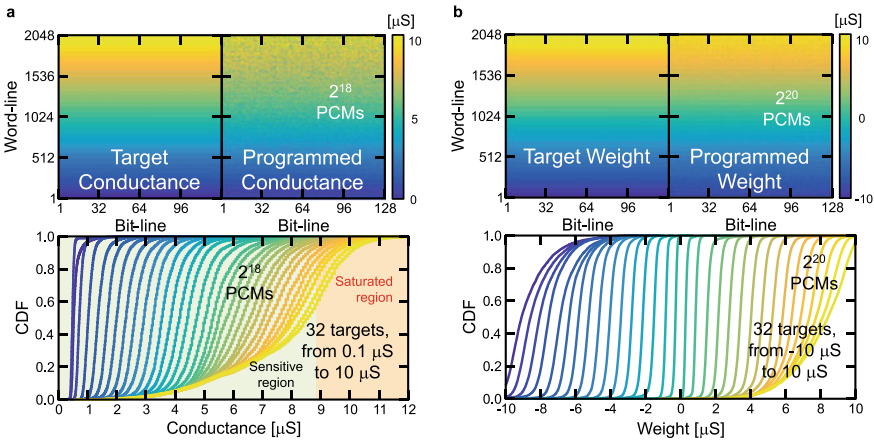


Fig. 14 Experimental comparison between single PCM (a) and full weight (b) closed-loop tuning. Top figures show the target conductances and weights, together with the achieved programmed results. Bottom figures show the corresponding cumulative distribution functions (CDF) for variable targets. While single-device CDFs (a) reveal increasingly broadened distributions due to programming error and device saturation for increasingly larger targets, weight CDFs (b) reveal steeper curves, which is a consequence of the improved programming precision, and reduced saturation due to increased weight dynamic range. Adapted with permission from [29]. Copyright 2019 IEEE

pulses until it drops below the threshold. This successive approximation technique continues for the LSP (Fig. 13a). In this case, since the contribution from g^+ and g^- is reduced by a constant factor F around 2–4, the program precision increases. By simulating this process, Fig. 13b shows that $\sim 98\%$ of weights can be programmed effectively using this strategy, with only $\sim 2\%$ of weights out of target, thus providing a very strong correlation between the programmed and target weights (Fig. 13c).

To verify the impact of MSP and LSP pairs on write precision, actual experiments are shown in Fig. 14. The lower conductance values are easier to reach, but as the target value gets higher, the PCMs less reliably program the desired value (Fig. 14a) due to variability in the maximum achievable conductance for each device [29]. This can also be seen by studying the cumulative distribution functions (CDF), which

show that the percentage of PCM devices that reach a certain conductance target decreases for increasing target values. In addition, CDF curves are not very steep due to programming errors that limit the precision we can achieve in tuning a single PCM device.

When the weights are split into four conductances with a simplified version of the programming strategy described, more of the PCMs reach the desired conductance values (Fig. 14b) due to an increased dynamic range. In addition, CDF distributions are steeper, revealing a better control of the weight programming, leveraging both MSP and LSP conductance pairs [29]. Using this programming strategy, software-equivalent accuracy was demonstrated, with a mixed software-hardware experiment, on LSTM networks similar to the one introduced in Sect. 1.2 [29].

2.2 Counteracting Conductance Drift Using Slope Correction

Additionally, even after programming the desired values, phase change memory exhibits another non-ideality called conductance drift, where the device conductance decays with increasing time due to the structural relaxation of the amorphous phase [28]. Since weights are encoded in the conductance state of PCM devices, drift degrades the precision of the encoded weight, and needs to be counteracted [31, 32].

Drift typically affects reset and partial reset states, with an empirical power law describing the time dependence:

$$G(t) = G_0 * \left(\frac{t}{t_0}\right)^{-\nu},$$

where G_0 is the very first conductance measurement obtained at time t_0 after the programming time. For increasing times t , conductance decays with a power law defined by a negative drift coefficient ν , indicating the drift rate.

Drift is a very rapid process right after programming but slows down considerably as time continues. While all PCM devices experience drift, each device drifts at a slightly different rate. To properly evaluate the drift coefficient distribution across multiple PCMs, ν coefficients for 20,000 devices were extracted by measuring the conductances at multiple times over 32 h [31]. Figure 15a shows the ν distribution for all 20,000 devices as a function of the initial conductance measurement $G(t_1)$. The corresponding median $\bar{\nu}$ and standard deviation σ_ν are then extracted and shown in Fig. 15b. As a first approximation, we can consider $\bar{\nu}$ equal to 0.031, and σ_ν equal to 0.007. These parameters have been implemented in our model to study the impact of drift on neural network inference [31].

In order to understand how to correct conductance drift, it can be helpful to look more carefully at the activation function used during the multiply-accumulate operation discussed earlier. This activation function f (also known as a squashing function) transforms the data. Different functions will result in different transforms.

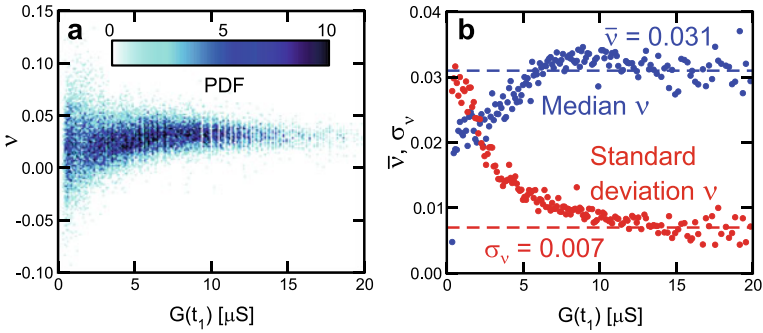


Fig. 15 Experimental drift characterization on 20,000 PCM devices. After PCM programming, conductances have been measured up to 32h, and corresponding v have been extracted and plotted as a function of the initial conductance (a). The extracted median \bar{v} and standard deviation σ_v are plotted in (b). Adapted with permission from [31]. Copyright 2019 IEEE

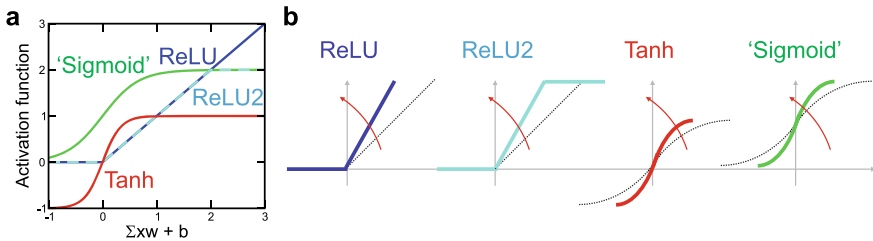


Fig. 16 A wide variety of activation functions are used in practice, four examples of which are shown in (a). The sigmoid function is expanded to better compare to the tanh function. **b** We can apply activation amplification factors (“slope correction” factors) to these functions to counteract drift over time. Adapted with permission from [31]. Copyright 2019 IEEE

Here four main functions are introduced: ReLU (Rectified Linear Unit), clamped ReLU, rescaled sigmoid, and tanh, as shown in Fig. 16a. Except for ReLU, which is unbounded, all the other squashing functions are studied here at the same amplitude. As the standard deviation of the drift coefficients approaches 0, the effects of drift can be factored out of the multiply-accumulate operation as a single time-varying constant. This allows for the compensation of conductance drift by applying a time-dependent activation amplification equal to $(t/t_0)^{+v_{correction}}$, where the previously extracted \bar{v} is chosen as $v_{correction}$. This factor is applied to the slope of the activation function (Fig. 16b) [31].

This technique has been evaluated on several networks. Initially, without any correction, the fully connected network experiences a marked accuracy degradation, over time, on the MNIST dataset of handwritten digits (Fig. 17a), due to conductance drift. With slope correction, however, results for all the activation functions studied improve significantly (Fig. 17b). The technique has also been evaluated on a ResNet-18 CNN trained on the CIFAR-10 image classification dataset (Fig. 18a) and an LSTM, trained on the text from the book Alice in Wonderland (Fig. 18b). With these

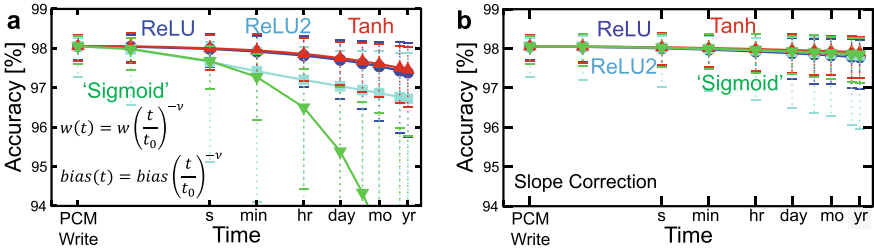


Fig. 17 **a** With no slope correction, even a small MLP suffers a strong accuracy loss from conductance drift. **b** However, with slope correction, accuracy barely degrades over time. The remaining decay is due to the σ_ν spread. Adapted with permission from [31]. Copyright 2019 IEEE

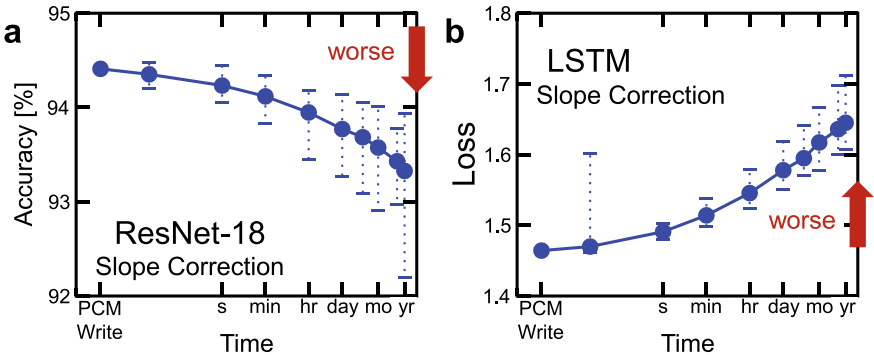


Fig. 18 On deeper, more sophisticated networks, the results still hold and slope correction is vital for maintaining accuracy over time for both ResNet and LSTM. LSTM results are measured according to the loss, where a lower loss is better, as a higher accuracy for ResNet is better. Adapted with permission from [31]. Copyright 2019 IEEE

more complicated networks, without any slope correction, accuracy was found to dramatically drop. However, slope correction is still highly effective in reducing the impact of conductance drift, as shown in Fig. 18. It is important to note that slope correction will not remove the entire impact of drift since the σ_ν is non-zero, meaning not all PCM devices drift at the same rate. However, even so, slope correction still shows a large impact, while remaining accuracy degradation can be compensated by using slightly larger networks [31].

3 Conclusion

In summary, analog hardware accelerators with phase change memory are a promising alternative to GPUs for neural network inference. The multiply-accumulate operation, which is the most computationally expensive operation in a neural network, can be performed at the location of the data, saving power and time. However,

with this promise comes some non-idealities. Recent software-equivalent inference results [23, 29, 30] include strategies to program the weights more precisely using 4 PCM devices, and a slope correction technique can be used to reduce the impact of resistance drift [31].

References

1. Y. LeCun et al., *Nature* **521** (2015)
2. I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning* (MIT Press, 2016)
3. V. Sze, Y. Chen, T. Yang, J.S. Emer, *Proc. IEEE* **105**, 2295–2329 (2017)
4. S. Hochreiter, J. Schmidhuber, *Neural Comput.* **9**, 1735–1780 (1997)
5. J. Schmidhuber, *Neural Netw.* **61**, 85–117 (2015)
6. Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, *Proc. IEEE* **86**, 2278–2324 (1998)
7. I. Boybat, C. di Nolfo, S. Ambrogio, M. Bodini, N.C.P. Farinha, R.M. Shelby, P. Narayanan, S. Sidler, H. Tsai, Y. Leblebici, G.W. Burr, Improved deep neural network hardware-accelerators based on non-volatile-memory: the local gains technique, in *IEEE International Conference on Rebooting Computing (ICRC)* (2017)
8. J. Deng, W. Dong, R. Socher, L.J. Li, K. Li, L. Fei-Fei, ImageNet: a large-scale hierarchical image database. In: *CVPR09* (2009)
9. A. Krizhevsky, I. Sutskever, G.E. Hinton, Imagenet classification with deep convolutional neural networks. In: *Advances in Neural Information Processing Systems*, pp. 1097–11051 (2012)
10. B. Fleischer, S. Shukla, M. Ziegler, J. Silberman, J. Oh, V. Srinivasan, J. Choi, S. Mueller, A. Agrawal, T. Babinsky, N. Cao, C. Chen, P. Chuang, T. Fox, G. Gristede, M. Guillorn, H. Haynie, M. Klaiber, D. Lee, S. Lo, G. Maier, M. Scheuermann, S. Venkataramani, C. Vezirtzis, N. Wang, F. Yee, C. Zhou, P. Lu, B. Curran, L. Chang, K. Gopalakrishnan, A scalable multi-teraops deep learning processor core for ai trainina and inference, in *IEEE Symposium on VLSI Circuits*, pp. 35–36 (2018)
11. J.L. Hennessy, D. Patterson, *Morgan Kaufmann* (1989)
12. N.P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al., In-datacenter performance analysis of a tensor processing unit, in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 1–12 (ACM, 2017)
13. G.W. Burr, R.M. Shelby, S. Sidler, C. di Nolfo, J. Jang, I. Boybat, R.S. Shenoy, P. Narayanan, K. Virwani, E.U. Giacometti, B. Kurdi, H. Hwang, **62**, 3498–3507 (2015)
14. S. Ambrogio, P. Narayanan, H. Tsai, R.M. Shelby, I. Boybat, C. di Nolfo, S. Sidler, M. Giordano, M. Bodini, N.C.P. Farinha, B. Killeen, C. Cheng, Y. Jaoudi, G.W. Burr, *Nature* **558**, 60 (2018)
15. G.W. Burr, R.M. Shelby, A. Sebastian, S. Kim, S. Kim, S. Sidler, K. Virwani, M. Ishii, P. Narayanan, A. Fumarola, L.L. Sanches, I. Boybat, M.L. Gallo, K. Moon, J. Woo, H. Hwang, Y. Leblebici, *Adv. Phys. X* **2**, 89–124 (2017)
16. P. Narayanan, A. Fumarola, L. Sanches, S. Lewis, K. Hosokawa, R.M. Shelby, G.W. Burr, *IBM J. (Res, Dev)* **61** (2017)
17. I. Boybat, M.L. Gallo, R.S. Nandakumar, T. Moraitis, T. Parnell, T. Tuma, B. Rajendran, Y. Leblebici, A. Sebastian, E. Eleftheriou, *Nat. Commun.* **9**2514 (2018)
18. R.S. Nandakumar, M.L. Gallo, I. Boybat, B. Rajendran, A. Sebastian, E. Eleftheriou, in *IEEE ISCAS Proc*, pp. 1–5 (2018)
19. M.L. Gallo, A. Sebastian, R. Mathis, M. Manica, H. Giefers, T. Tuma, C. Bekas, A. Curioni, E. Eleftheriou, *Nat. Electron.* **1**, 246–253 (2018)
20. H.Y. Tsai, S. Ambrogio, P. Narayanan, R. Shelby, G.W. Burr, *J. Phys. D Appl. Phys.* **51**(2018)
21. S. Raoux, G.W. Burr, M.J. Breitwisch, C.T. Rettner, Y.C. Chen, R.M. Shelby, M. Salinga, D. Krebs, S.H. Chen, H.L. Lung, C.H. Lam, *IBM J. Res. Dev.* **52**, 465–479 (2008)

22. G.W. Burr, M.J. BrightSky, A. Sebastian, H.Y. Cheng, J.Y. Wu, S. Kim, N.E. Sosa, N. Papan-dreou, H.L. Lung, H. Pozidis, E. Eleftheriou, C.H. Lam, *IEEE J. Emerg. Sel. Topics Circuits Syst.* **6**, 146–162 (2016)
23. C. Mackin, H. Tsai, S. Ambrogio, P. Narayanan, A. Chen, G.W. Burr, *Adv. Electron. Mater.* **5**, 1900026 (2019)
24. S. Gupta, A. Agrawal, K. Gopalakrishnan, P. Narayanan, Deep learning with limited numerical precision, in *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pp. 1737–1746 (2015)
25. K. Spoon, S. Ambrogio, P. Narayanan, H. Tsai, C. Mackin, A. Chen, A. Fasoli, A. Friz, G.W. Burr, Accelerating deep neural networks with analog memory devices, in *IEEE International Memory Workshop (IMW)*, pp. 1–4 (2020)
26. A. Krizhevsky, V. Nair, G. Hinton, <http://www.cs.toronto.edu/kriz/cifar.html>
27. K. He, X. Zhang, S. Ren, J. Sun, Preprint 1512.03385 CoRR (2015) **abs/1512.03385** <http://arxiv.org/abs/1512.03385>
28. M. Boniardi, D. Ielmini, S. Lavizzari, A.L. Lacaita, A. Redaelli, A. Pirovano, *IEEE Trans. Electron Devices* **57**, 2690–2696 (2010)
29. H. Tsai, S. Ambrogio, C. Mackin, P. Narayanan, R.M. Shelby, K. Rocki, A. Chen, G.W. Burr, *2019 Symposium on VLSI Technology*, pp. T82–T83 (2019)
30. G. Cristiano, M. Giordano, S. Ambrogio, L. Romero, C. Cheng, P. Narayanan, H. Tsai, R.M. Shelby, G. Burr, *J. Appl. Phys.* **124**, 151901 (2018)
31. S. Ambrogio, M. Gallot, K. Spoon, H. Tsai, C. Mackin, M. Wesson, S. Kariyappa, P. Narayanan, C. Liu, A. Kumar, A. Chen, G.W. Burr, Reducing the impact of phase-change memory conductance drift on the inference of large-scale hardware neural networks, in *2019 IEEE International Electron Devices Meeting (IEDM)*, pp. 6.1.1–6.1.4 (2019)
32. V. Joshi, M.L. Gallo, S. Haefeli, I. Boybat, S.R. Nandakumar, C. Piveteau, M. Dazzi, B. Rajendran, A. Sebastian, E. Eleftheriou, *Nat. Commun.* 2473 (2020)