Optimized Weight Programming for Analogue Memory-based Deep Neural Networks

Charles Mackin (Charles.Mackin@ibm.com)
IBM Research

Malte Rasch
Beijing Normal University https://orcid.org/0000-0002-7988-4624

An Chen
IBM Research

Jonathan Timcheck
Stanford University https://orcid.org/0000-0002-2071-2668

Robert Bruce
IBM T.J. Watson Research Center

Pritish Narayanan
IBM Research

Stefano Ambrogio
IBM Research–Almaden

Manuel Le Gallo
IBM Research - Zurich https://orcid.org/0000-0003-1600-6151

Nandakumar Rajaleksh
IBM https://orcid.org/0000-0002-7930-508X

Andrea Fasoli
IBM Research

Jose Luquin
IBM Research

Alexander Friz
IBM Research

Abu Sebastian
IBM Research - Zurich https://orcid.org/0000-0001-5603-5243

HsinYu Tsai
IBM Watson Research Center

Geoffrey Burr
IBM Research–Almaden https://orcid.org/0000-0001-5717-2549

Article

Keywords: Analogue memory-based Deep Neural Networks (DNNs), analogue memories, weight programming optimization

Posted Date: November 10th, 2021

DOI: https://doi.org/10.21203/rs.3.rs-1028668/v1

Version of Record: A version of this preprint was published at Nature Communications on June 30th, 2022. See the published version at https://doi.org/10.1038/s41467-022-31405-1.

Optimized Weight Programming for Analogue Memory-based Deep Neural Networks

Charles Mackin[1], Malte Rasch[2], An Chen[1], Jonathan Timcheck[3], Robert L. Bruce[2], Pritish Narayanan[1], Stefano Ambrogio[1], Manuel Le Gallo[4], S. R. Nandakumar[4], Andrea Fasoli[1], Jose Luquin[1], Alexander Friz[1], Abu Sebastian[4], Hsinyu Tsai[1] & Geoffrey W. Burr[1]

[1]*IBM Research–Almaden, San Jose, CA, USA*

[2]*IBM Research–Yorktown Heights, NY, USA*

[3]*Stanford University, Stanford, CA, USA*

[4]*IBM Research–Zürich, Switzerland*

Analogue memory-based Deep Neural Networks (DNNs) provide energy-efficiency and per-area throughput gains relative to state-of-the-art digital counterparts such as graphic processing units (GPUs). Recent advances focus largely on hardware-aware algorithmic training and improvements in circuits, architectures, and memory device characteristics. Optimal translation of software-trained weights into analogue hardware weights—given the plethora of complex memory non-idealities—represents an equally important goal in realizing the full potential of this technology. We report a generalized computational framework that automates the process of crafting complex weight programming strategies for analogue memory-based DNNs, in order to minimize accuracy degradations during inference, particularly over time. This framework is agnostic to DNN structure and is shown to generalize well across Long Short-Term Memory (LSTM), Convolution Neural Networks (CNNs), and

Transformer networks. Being a highly-flexible numerical heuristic, our approach can accommodate arbitrary device-level complexity, and is thus broadly applicable to a variety of analogue memories and their continually evolving device characteristics. Interestingly, this computational technique is capable of optimizing inference accuracy without the need to run inference simulations or evaluate large training, validation, or test datasets. Lastly, by quantifying the limit of achievable inference accuracy given imperfections in analogue memory, weight programming optimization represents a unique and foundational tool for enabling analogue memory-based DNN accelerators to reach their full inference potential.

The generation, storage, and processing of ever-increasing amounts of data in support of rapid and sophisticated decision-making has spurred remarkable advances in Deep Neural Networks (DNNs) in recent years[1]. DNNs have become ubiquitous within image classification, language processing, prediction, and similar critical tasks across a spectrum of industries. Advancements in deep learning algorithms, architectures, and hardware now enable DNNs to boast near-human—and in some cases—supra-human capabilities. This performance, however, comes at tremendous computational cost in terms of time and energy consumption. A distributed implementation of AlphaGo, which beat the human European champion of the Go strategy board game, required 1,202 CPUs, 176 GPUs, and hundreds of kilowatts[2]. Similarly, a state-of-the-art language prediction model such as Generative Pre-Trained Transformer 3 (GPT-3) contains approximately 175 billion weights, cost tens of millions of dollars to train, and requires approximately eleven Tesla V100 GPUs and thousands of watts for inference[3]. Highly optimized GPUs and tensor pro-

cessing units (TPUs) form the hardware substrate supporting of these systems. Such compute engines, however, are based on conventional von Neumann architectures, in which the memory blocks that store the synaptic weights are physically separate from the computational blocks that process data. This requires high bandwidth and continual data transport between memory and computational blocks, exacting unavoidable time and energy penalties and limiting overall performance (i.e. the 'von Neumann' bottleneck). This has spurred interest in the development of alternative non-von Neumann architectures for DNN acceleration.

DNNs rely extensively on vector-matrix multiplication (VMM) operations, which lend themselves naturally to non-von Neumann crossbar array structures. Within crossbar arrays, analogue memory elements encode the synaptic weights of the network. DNN activations are applied along rows of the memory array, multiplied by the synaptic weights according to Ohm's law, and summed along each column according to Kirchhoff's current law. This enables the crossbar array to implement VMM operations at the location of the data to reduce the impact of the von Neumann bottleneck. This approach was recently shown capable of 280$\times$ speedup in per-area throughput while providing 100$\times$ enhancement in energy-efficiency over state-of-the-art GPUs[4].

Analogue memory-based DNN accelerators are being widely developed in academia and industry using a variety of memories[5], including resistive RAM (ReRAM)[6,7], conductive-bridging RAM (CBRAM)[8], NOR flash[9–12], magnetic RAM (MRAM), and phase-change memory (PCM)[13,14]. To date, each type of analogue memory device exhibits some form of non-ideal behavior such as limited resistance contrast, significant non-linearity and stochasticity in conductance-vs-pulse

characteristics, strong asymmetry during bidirectional programming, read noise, and conductance drift after programming to name a few[15–19]. These memory imperfections ultimately introduce errors into the VMM computations, and can often lead to diminished DNN accuracy relative to state-of-the-art digital systems. That said, said state-of-the-art digital systems are currently being optimized to deliver identical DNN accuracies even when activation-precision and weight-precision are reduced from FP32 to INT4 or less[20,21]. Thus if DNN models are inherently capable of delivering accurate predictions despite low-digital precision compute, there is a strong expectation that the minimum Signal-to-Noise Ratios (SNRs) within analogue-memory based systems needed for similar DNN accuracy should not be excessively high.

Incorporating hardware non-idealities within DNN training (i.e. 'Hardware-Aware' algorithmic training) is effective in making analogue memory-based DNNs more resilient to hardware imperfections[22–24]. Hardware-aware training typically captures various types of memory non-idealities along with circuit nonlinearities such as IR-drops within the crossbar array, and activation quantization due to analogue-to-digital converters (ADCs) and pulse-width modulators (PWMs). Both conventional and novel hardware-aware training produce DNN models comprised of 'unitless' synaptic weights. As shown in Figure 1, before programming into the analogue memory of choice, these unitless DNN model-weights must be converted into target conductances, typically in units of microSiemens. Since analogue memory weights can be encoded across multiple memory devices, there can be infinitely many ways to implement the same synaptic weight. However, each of these choices for how the weight gets distributed across multiple conductances, will not produce equivalent weight errors. This is further complicated by the fact that DNNs are typically comprised

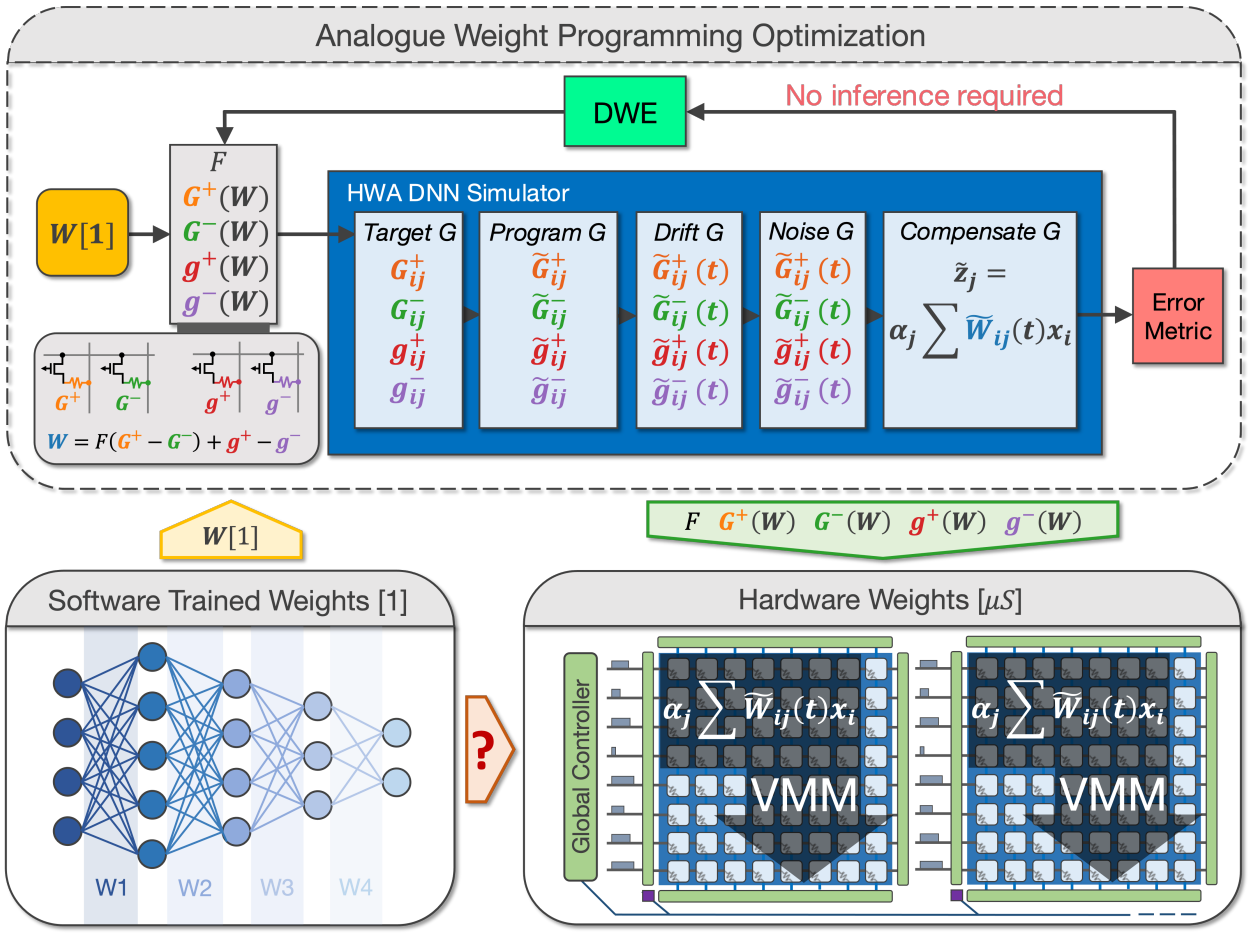

Figure 1: 'Unitless' weights from software DNN models must be re-scaled into an optimal hardware range (microSiemens), and can then be encoded across multiple analogue memory devices: G^+, G^-, g^+, and g^-. A weight programming optimization framework captures all memory imperfections and hardware compensatory techniques, and produces optimal weight programming strategies using an iterative Differential Weight Evolution (DWE) technique to minimize inference accuracy degradations for analogue memory-based DNNs, including degradations that change over time. This can be achieved without the need to run costly inference simulations at multiple time-steps using large datasets.

of millions of weights, ranging from large positive to near-zero to large negative weight values. The high degree of inherent interconnectedness present in DNNs also means that any systemic weight errors introduced through sub-optimal weight translation strategies will almost certainly propagate and compound throughout the network. This causes the trained DNN, which has been highly optimized for a specific task, to be perturbed with virtually zero probability of coincidentally landing on a similarly optimal configuration that was not discovered during the training process, especially due to the high dimensionality. This ultimately leads to degraded DNN inference accuracy because there exists a discrepancy between the DNN that was trained—hardware-aware or otherwise—and the analogue memory-based DNN that actually exists in the hardware. Worse yet, in the presence of conductance drift after programming, this degradation is also changing over time.

Analogue memory-based weights typically introduce programming errors due to stochasticity in conductance-vs-pulse curves, device variability, and imperfect yield. An ideal weight programming strategy should determine the target conductances for programming that provide the best possible outcome—despite errors in programming the conductances at time t_0, the subsequent evolution of these weights due to conductance drift, and the read noise associated with performing VMM computations at each point in time (Figure 2a). Conductance drift is typically modeled using a power law: $G(t) = G_0(t/t_0)^{-\nu}$, where G_0 is the initial conductance at a reference time t_0, and ν is the drift coefficient that determines how the conductance changes with time[25]. Conductance drift is not captured during training, but can be considered during the weight translation process in order to minimize degradations in inference accuracy over time. For instance, if all devices drifted with exactly the same ν coefficient, then we could simply amplify the integrated column currents

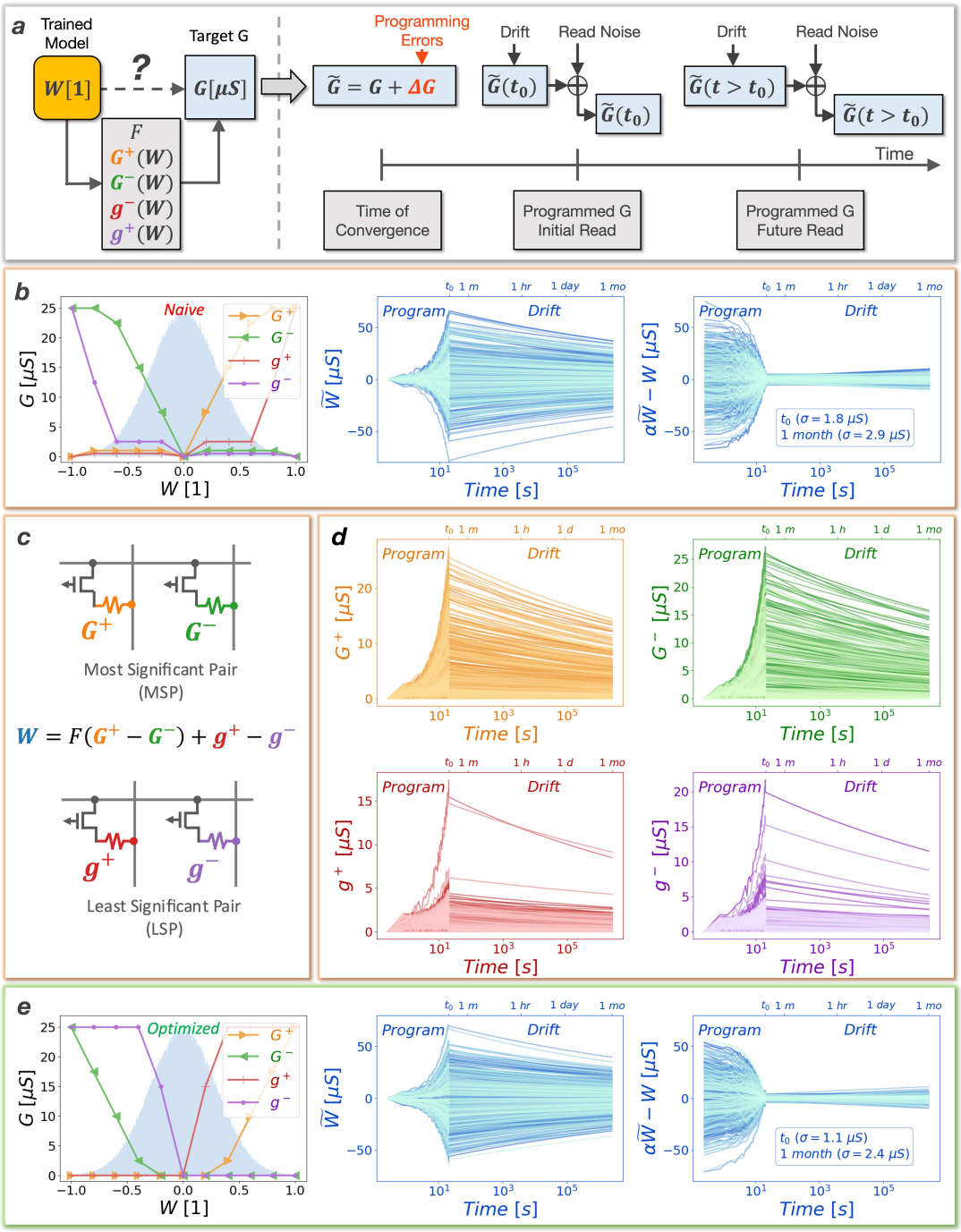

Figure 2: a) Unitless weights of software trained DNN models are translated to analogue memory-based synaptic weights comprised of multiple conductances, which are subject to imperfections including programming errors, conductance drift, and read noise. b) A sub-optimal weight programming strategy leads to outsized hardware weight errors at t_0 that become progressively worse with time, even after drift compensation factor α is applied. c) Each synaptic weight may be comprised of multiple conductances of varying significance as indicated by the factor F, which separates the Most Significant Pair (MSP) and Least Significant Pair (LSP). d) Individual conductance programming errors are compounded by subsequent drift over time. e) An optimized weight programming strategy results in minimal weight errors as indicated visually and by the lower standard deviation in weight errors.

with a single scaling coefficient that depended only on the elapsed time since programming. The only drawback would be that eventually we would be amplifying the small amount of background noise enough that overall SNR might start to decrease. Unfortunately, conductances typically have complex drift characteristics where ν coefficients exhibit stochastic intra-device ('shot-to-shot') variability. Thus we cannot precisely know the value of ν that will ensue after any given programming event, even for devices that have been carefully characterized. Furthermore, conductances also tend to drift more quickly or slowly depending on the magnitude of the conductance programmed, and the variability in ν coefficients is sometimes observed to be conductance-dependent as well[26].

In Figure 2c, each synaptic weight is comprised of multiple conductances with opposing polarities that will drift at different rates (Figure 2d) to define the overall evolution of the weight with time. Conductances within a weight may also have varying significance as determined by the scaling factor F between the Most Significant Pair (MSP) and Least Significant Pair (LSP)[27]. This can be advantageous because the MSP can be used to increase the overall dynamic range and program the bulk of the weight, whereas the LSP can be used to fine tune the programmed weight for better precision. The significance factor can be implemented in a number of ways, but is limited to discrete values in this case, which can be readily implemented by multiplying the durations of the input activations applied to the MSP relative to the LSP. The use of multiple conductances per weight also introduces a level of redundancy to mitigate device variability and occasional device failures (i.e. imperfect yield).

Figure 2b depicts an example of what is termed a *naive* programming strategy, in which the majority of the weight is programmed in the MSP, and the LSP is then intuitively used to fine tune the weight in an attempt to eliminate weight errors. Programming and drift characteristics are plotted for the individual conductances and the resulting weights. The evolution of weight errors $W_E = \alpha\widetilde{W} - W$ as a function of time is depicted, including read noise. Due to the conductance changes over time (Figure 2d), drift causes weight magnitudes to decline with time, which causes the activations stemming from VMM computations to also decline and adversely affect inference accuracy. As mentioned earlier, this can be mitigated using a drift compensation technique[28], where activations are amplified to their original levels using a digital scale factor α, which may or may not be uniform along the column-wise dimension of the crossbar array. Drift compensation factors can be calculated using a calibration technique where one or more randomized input vectors are applied to the crossbar array immediately after weight programming. The resulting output activations are saved either locally or off-chip and can be compared with future applications of the same randomized vectors to determine the appropriate drift compensation factor α. For this particular set of drift characteristics, an alpha factor of 1.8 is needed after approximately one month (Figure 2b,e right side).

Finally, Figure 2e illustrates how an optimized weight programming strategy can allow drift compensated hardware programmed weights $\alpha\widetilde{W}$ to more closely track ideal weights W, including as a function of time. This is indicated by the lower standard deviation in weight errors associated with Figure 2e relative to Figure 2b. In this case, weight errors are reduced by approximately 39% at t_0 immediately after programming and by approximately 17% at one month. In such optimized

weight-programming strategies, individual target conductances exhibit complex dependencies on the input unitless software weights: $G^{+}(W)$, $G^{-}(W)$, $g^{+}(W)$, $g^{-}(W)$ (left side of Figure 2e). Allowing for these complex programming schemes provides the flexibility necessary to mitigate equally complex distortions in DNN weights resulting from the combination of multiple analogue memory non-idealities. Weight errors also become amplified by the drift compensation factor α, making it critical to find optimal weight programming strategies to limit accuracy degradations over time. Both examples (Figures 2b,e) use identical device models for programming errors, drift, and read noise and are re-scaled into the same hardware weight range for fair comparison.

As seen in Figure 2, given the many different potential error sources injected into VMM computations by analogue memory, along with the complexity of device-level models and the infinite number of potential weight programming strategies, any uninformed or *naive* weight programming strategy will almost certainly result in sub-optimal weight fidelity and excessive accuracy degradation for analogue memory-based DNNs, particularly as conductances drift over time. This problem is further complicated by the fact that it becomes impractical to run time-consuming inference simulations and evaluate training, validate or test datasets in an iterative fashion—especially at multiple time-steps to include drift—while exploring the vast search-space of number of possible weight programming strategies.

The overarching objective is to find software-to-hardware translation functions $G^{+}(W), G^{-}(W)$, $g^{+}(W)$, and $g^{-}(W)$ for weight programming such that $\beta_{hw}W = F[G^{+}(W) - G^{-}(W)] + g^{+}(W) - g^{-}(W)$, where W is the unitless software weight, β_{hw} is the software-to-hardware weight re-

scaling factor, and F is the MSP to LSP significance factor. In this paper, we present a generalized framework capable of producing complex weight programming strategies for analogue memory-based DNNs in light of these constraints. The framework is agnostic to DNN structure, and is shown to generalize well across a variety networks including Long Short-Term Memory (LSTM), Convolutional Neural Networks (CNNs), and Transformers. The numerical framework is capable of accommodating arbitrary device-level complexity and automates the process of finding optimal weight programming strategies—a critical capability given the continual evolution of analogue memory devices. Solving this problem represents a pivotal step towards allowing analogue memory-based DNNs to realize their full energy-efficiency and throughput benefits, while helping to close accuracy gaps with state-of-the-art digital approaches.

Results

We solve a complex and highly-capable form of weight programming optimization, in which each synaptic weight is comprised of four conductances G^+, G^-, g^+, and g^- and includes a varying significance factor F. This results in a $\sim 4N$ dimensional parameter space, where N is the number of weights within the network (typically millions). Two additional dimensions are added to the problem: β_{hw}, which is the scale factor converting the unitless software weights into hardware weights; and F, which is the MSP to LSP significance factor. That brings the total number of parameters to $4N+2$, making the dimensionality of the weight programming optimization problem potentially larger than the DNN itself. Exploring such a large space, especially with the inference simulations at multiple time-steps within the feedback loop to optimize for drift, quickly becomes

intractable. It then becomes critical to reduce the dimensionality of the optimization problem without hampering the ability to find advantageous weight programming strategies.

Our proposal is to identify the optimal programming strategy for a handful of discretized weights across the useful weight programming range—as opposed to the entire continuous weight distribution—and then linearly interpolate these results for all intermediate weights. The dimensionality can be further reduced by taking advantage of symmetry in DNN weight distributions. Weight distributions for the trained LSTM, CNN, and Transformer networks are highly symmetric about the origin. Thus the programming optimization results for positive weights, for instance, can be mirrored for negative weights—further reducing the number of parameters by a factor of two. The framework remains readily extensible to strongly asymmetric weight distributions—the dimensionality of the problem simple doubles when it is important to solve for distinct programming strategies for positive and negative weights. For symmetric weight distributions, however, the dimensionality has now been effectively reduced from $\sim 4N + 2$ down to $4D + 2$, where D is the number of discretized positive weights. The reported results make use of six discretized weight intervals. As an example, our transformer model, BERT-base, has approximately 86 million weights, of which 53 million are unique. Our dimensional-reduction approach has decreased the number of potential weight programming parameters from approximately 212 million down to twenty-six. Despite this significant reduction, there still exist infinitely many different weight programming strategies to explore, because the search space is still continuous for each conductance parameter and there are infinitely many ways to combine programming strategies for each of the unique discretized weights D.

Although the dimensionality of the problem has been reduced to something tractable, it is still important to address the time-consuming inference simulations within the weight programming optimization loop. Ideally, the best way to gauge the quality of a weight programming strategy is to simulate weight programming using a particular strategy, run inference simulations on the test dataset at multiple time-steps to account for drift, and record the DNN accuracy as a function of time. Because there still exist infinitely many programming strategies to explore, running inference simulations repeatedly at different time-steps to optimize for drift while using large datasets becomes impractical—even given the highly parallelized compute capabilities of a large GPU cluster.

We propose an alternative metric to serve as a proxy for DNN inference accuracy and allow accelerated exploration of the weight programming space, without the need for inference within the optimization loop. We observe that in the limit as weight errors approach zero, hardware weights become exact replicas of the software trained weights and DNN accuracy becomes identical to the baseline trained accuracy. It then follows that minimizing weight errors (i.e. preserving weight fidelity), including across multiple time-steps after conductance programming, should improve DNN inference accuracy over time. One reason this works well is that it remains highly unlikely that introducing large stochastic weight errors will coincidentally move the DNN into a better weight configuration than the one discovered during training, especially given the high dimensionality of the DNN. As a result, the closer a system with imperfect conductances can stay to the initial target DNN weights, the better it should perform.

We propose a time-averaged and normalized mean-squared-error metric as a less computationally expensive proxy for inference accuracy in the weight programming optimization process. The error metric is $\sum_{i=1}^{T}\sum_{j=1}^{D}\gamma_j\sum_{k=1}^{S}[(\alpha_i\widetilde{W}_{ijk}/\beta_{hw}-W_{ijk})/\max(|\mathbf{W}|)]^2$, where T is the number of time steps over which to optimize inference accuracy, D is the number of discretized weights selected for optimization, and S is the number of sample weights simulated at each discretized weight to estimate variance in weight errors. γ_j is the relative importance of the discretized weight within the DNN weight distribution, α_i is the drift compensation factor, $\widetilde{W}_{ijk}$ is the target weight including all hardware associated errors (e.g. programming errors, conductance drift, read noise), β_{hw} is the software-to-hardware weight distribution re-scaling factor, W_{ijk} is the ideal unitless target weight from software, and $\mathbf{W}$ represents the entirety of the unitless software DNN weight distribution. Minimizing mean-squared-error encourages weight errors to be normally distributed with zero mean, which helps prevent introducing unwanted bias terms that would adversely impact accuracy. This error metric is normalized by the weight range $\max(|\mathbf{W}|)$ to minimize errors in relation to the overall width of the distribution.

Our error metric includes a temporal component to enable DNN inference optimization over time in the presence of drift. We opt for a time-averaged error metric that implies all time-steps are of equal importance. This is equivalent to saying inference accuracy at one second is just as important as inference accuracy at one hour. This time weighting, however, is easily modified to account for situations where inference accuracy may be non-uniformly important over time. It may also be beneficial to introduce different temporal weighting schemes, or to organize the time-steps in a non-uniform (e.g., logarithmic) way (due to the power law nature of conductance drift

in Phase-Change Memory (PCM), for instance). Lastly, all weight errors are treated as equally important, which results in errors being weighted using γ_j according to their relative frequency (density) in the DNN weight distribution. This stems from the fact that we find zero correlation between weight values and their gradients during the last epoch of training (Figure 3a,b,c). These gradients are a direct estimate of the adverse impact of weight perturbations (errors) on the DNN loss function (accuracy) during the last steps of training, and thus can serve as a proxy for the network's sensitivity to errors on each weight.

Figure 3 provides a high-level overview of several key steps in the weight programming optimization process for three very different DNNs: LSTM, ResNet-32, and BERT-base (Figure 3a-c). Optimization is performed on a $4D + 2$ dimensional hypercube (Figure 3d) because $W = F(G^+ - G^-) + g^+ - g^-$ represents a hyperplane, which reduces the dimensionality of the search space but also adds constraints to the optimization problem. To avoid $4D$ inter-dependent conductance constraints, optimization is performed on a hypercube, which is then 'de-normalized' into valid conductance combinations in an intermediate step shown in Figure 3e. Further details on this de-normalization process are provided in the Supplementary Information.

Figure 3f represents a two-dimensional projection of the weight programming exploration space, including an example of an optimized weight programming strategy $G^+(W)$, $G^-(W)$, $g^+(W)$, and $g^-(W)$ indicated by solid lines. Violin plots in the background highlight the range of conductance values being explored, illustrating some of the conductance constraints. For instance, there is a great amount of overlap in violin plots at $0\ \mu S$ because small weights can be

constructed using small and large conductances alike. Large positive weights, however, must be constructed using large G^+ and g^+ values while minimizing G^- and g^- values, especially if we are to make effective use of the dynamic range and minimize relative weight errors. This is evidenced by the reduction in the range of valid conductance combinations explored when determining how to program larger-magnitude weights.

Figure 3g outlines the correlation between hardware weights and ideal weights at various points in time, but with the drift compensation factor α included, so that correlation plots for different points in time now overlap each other. In order to distinguish the distributions at different times, each has been plotted only at the locus of points corresponding to an outline of the distribution (i.e. capturing 100% of weights). Thus optimization seeks to bring these curves closer to the diagonal, yet because weights drift in complex ways due to conductance-dependent ν values and ν variability, the weight errors become progressively worse and the curves inherently move out away from the diagonal over time. The objective here, as described in the error metric, is to preserve weight fidelity to the extent possible given the complex behaviors of the different underlying devices models, and to do so across multiple time-steps. The distribution of the weight errors depicted in Figure 3h shows a similar inevitable outward diffusion over time due to drift. Lastly, Figure 3i shows the same distribution after normalization with respect to the maximum weight, which reflects how our error metric is computed. Weight errors depicted in Figure 3d-f have not been discretized, to better conceptually illustrate what is happening to the DNN weight distribution as a whole.

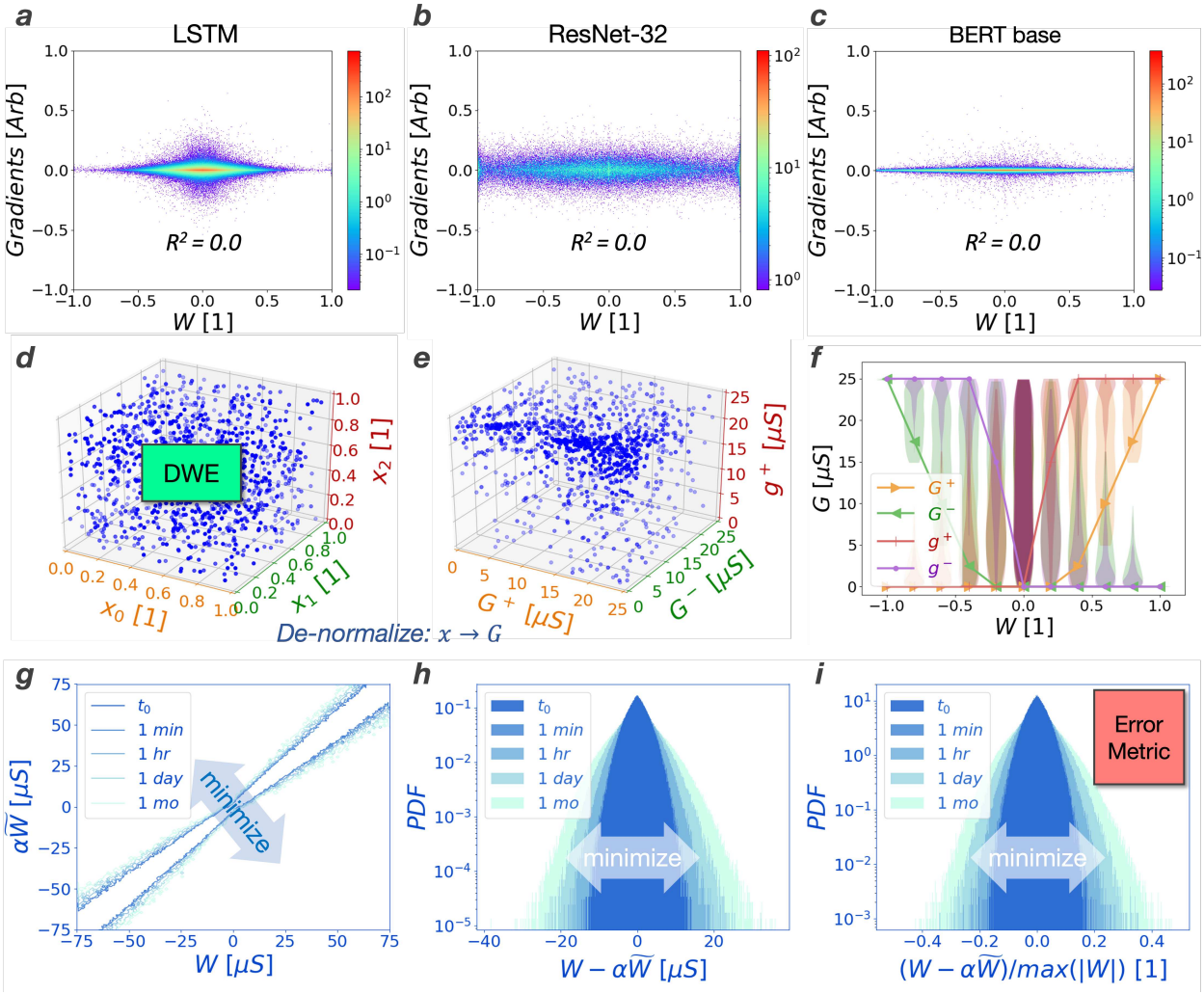

Figure 3: DNN gradients are uncorrelated with weight value as shown for a) LSTM, b) ResNet-32, and c) BERT base. This leads to a weight error importance γ_j, as defined in our error metric, which depends solely on weight density. The weight programming parameter space is then explored using d) Differential Weight Evolution (DWE) on parameter vectors x within a $\sim 4D + 2$ dimensional hypercube, where D represents the number of positive discretized weights. e) De-normalized hypercube parameters produce valid conductance combinations that capture optimization constraints due to conductance inter-dependencies. f) A two-dimensional projection of programming strategies, with violin plots showing coverage for the weight programming space explored, reveals underlying programming constraints. g) outlines of correlation distributions for drift compensated hardware weights $\alpha\widetilde{W}$ versus ideal weights W, plotted at plus-minus 1 standard-deviation, showing an outward diffusion over time. h) The corresponding probability density function of weight errors across all weight magnitudes, showing a similar outward diffusion with time. i) The resulting normalized weight error distribution used to define the error metric.

Although the constrained optimization problem has been transformed into the exploration of a hypercube, simultaneously accommodating multiple nonlinear and stochastic analogue memory device models still translates into optimizing a non-convex and stochastic (e.g., noisy) error metric. This renders gradient descent-based optimization techniques—which could potentially further accelerate the search through the weight programming space—ineffective. In general, non-convex and stochastic search-spaces pose difficulties for many optimizers. We found that least-squares, Nelder-Mead Simplex, gradient descent-based methods like Newton-Raphson, and combinations of gradient-descent and basin-hopping (i.e. simulated annealing) approaches all failed to reliably find adequate minima. We report, however, that the evolutionary algorithm approach known as Differential Evolution[29], when well-tuned and populated with different starting points, consistently performed well and identified good weight programming strategies. As a result, we appropriately refer to this heuristic optimization strategy as Differential Weight Evolution (DWE).

Now that we have enabled extensive optimization in a high-dimensional space across multiple time-steps—in a way that is completely agnostic to network structure, size, and test datasets thanks to the use of a proxy error metric—the question becomes whether the resulting weight programming strategies can actually materially improve inference accuracy in analogue memory-based DNNs.

Generalization across different DNN types We simulate the programming of tens of millions of weights according to the strategies derived from weight programming optimization computational technique, and evaluate a variety of DNNs and test datasets at multiple time-steps, to show

that weight programming optimization consistently enhances the accuracy of analogue memory-based DNNs. We find that our weight programming optimization generalizes well across different DNN types, including recurrent neural networks (RNNs), Convolutional Neural Networks (CNNs), and Transformer-based networks. For RNNs, we evaluate a two-layer Long Short-Term Memory (LSTM) network on the Penn Treebank dataset[30] (Figure 4a). For CNNs, we examine a ResNet-32 network using the CIFAR-10 dataset[31] (Figure 4b). And for Transformer-based networks, we evaluate BERT-base on the MNLI dataset[32] (Figure 4c). In this way, we not only demonstrate that weight programming optimization enhances the accuracy of analogue memory-based DNNs relative to sub-optimal programming strategies, but we also prove that our optimization approach generalizes well across a wide variety of very different network architectures. These accuracy enhancements are achieved while being completely agnostic to any network feature other than the weight distribution including network type, structure, complexity, type of nonlinear activations function employed, etc.

Even before optimized weight programming is brought to bear, Figure 4d-f vividly illustrates the benefits of hardware-aware (HWA) DNN training (solid) relative to floating-point (FP) trained networks (dashed), even when using *naive* programming strategies (MSP only, MSP/LSP (50/50)). Simulation results are the compilation of twenty-five independent weight programming and inference simulations over time, showing the average result with central lines surrounded by shaded regions representing plus-minus one standard deviation. By subjecting the DNN to noisy weights during training[22–24], HWA-training clearly makes the DNN more resilient to the various hardware non-idealities, and significantly enhances network accuracy relative to floating-point training.

HWA training alone, however, is insufficient for achieving and maintaining iso-accuracy as compared to the training baseline, especially as weights evolve after programming due to conductance drift.

Weight programming optimization is thus introduced to *augment* HWA-training. Figure 4a-c describes the particular set of analogue memory models used to define programming errors, drift coefficient, and read noise, all as a function of conductance. For each device characteristic, both the average response (solid line) and the variability around this (shaded region, corresponding to plus-minus one standard deviation) are used by the optimization algorithm. As shown by the green lines in Figure 4d-f, the combination of HWA-training together with weight programming optimization further enhances DNN accuracy, driving the inference accuracy (solid green) as close as it can possibly get—given the underlying memory non-idealities—to iso-accuracy with the trained model (dashed-dot purple line).

Results using non-optimized or *naive* programming strategies are shown as a reference, clearly demonstrating the added benefit of weight programming optimization. In the first *naive* weight programming strategy, the entirety of the weight is programmed into the MSP, while making minimal use of the LSP (similar to Figure 2b). In the second *naive* approach, weights are split equally between the MSP and LSP. This is similar to the weight slicing approach[33,34], which has been shown to offer accuracy benefits by effectively countering some of the variability present in analogue memory. Weight programming optimization, however, is able to devise much more complex weight programming strategies where programmed conductances can be functions of the

unitless weight: $G^{+}(W)$, $G^{-}(W)$, $g^{+}(W)$, and $g^{-}(W)$. The optimal programming strategies for each DNN are quite similar, which is reasonable given that each DNN is implemented using the same device models. This also demonstrates repeatability of the heuristic given that each optimization initialized with a random set of programming strategies. Minor variations in programming strategies are likely due to differences in the DNN weight distributions. Hyper-parameter scans for the hardware-aware and floating-point training for each DNN are provided in the Supplementary Information.

It is the combination of hardware-aware DNN training and subsequent weight programming optimization that drives inference accuracy as close as possible to iso-accuracy for analogue memory-based DNNs. As such, weight programming optimization represents a novel computational technique that can contribute significantly towards the eventual elimination of any accuracy gap between analogue memory-based DNNs and state-of-the-art digital approaches. This in turn enables analogue memory-based DNNs to better highlight their energy-efficiency and per-area throughput benefits, while minimizing potential trade-offs in accuracy.

Generalization across different device models We now modify the underlying analogue memory device characteristics and repeat the weight programming optimization, to see if our optimization approach generalizes well to different device models. This is a critical feature given that analogue memories are continually being modified and improved. If our weight programming optimization technique generalizes well across different device characteristics, we can effectively automate the process of finding optimal weight programming strategies. This represents an important step not

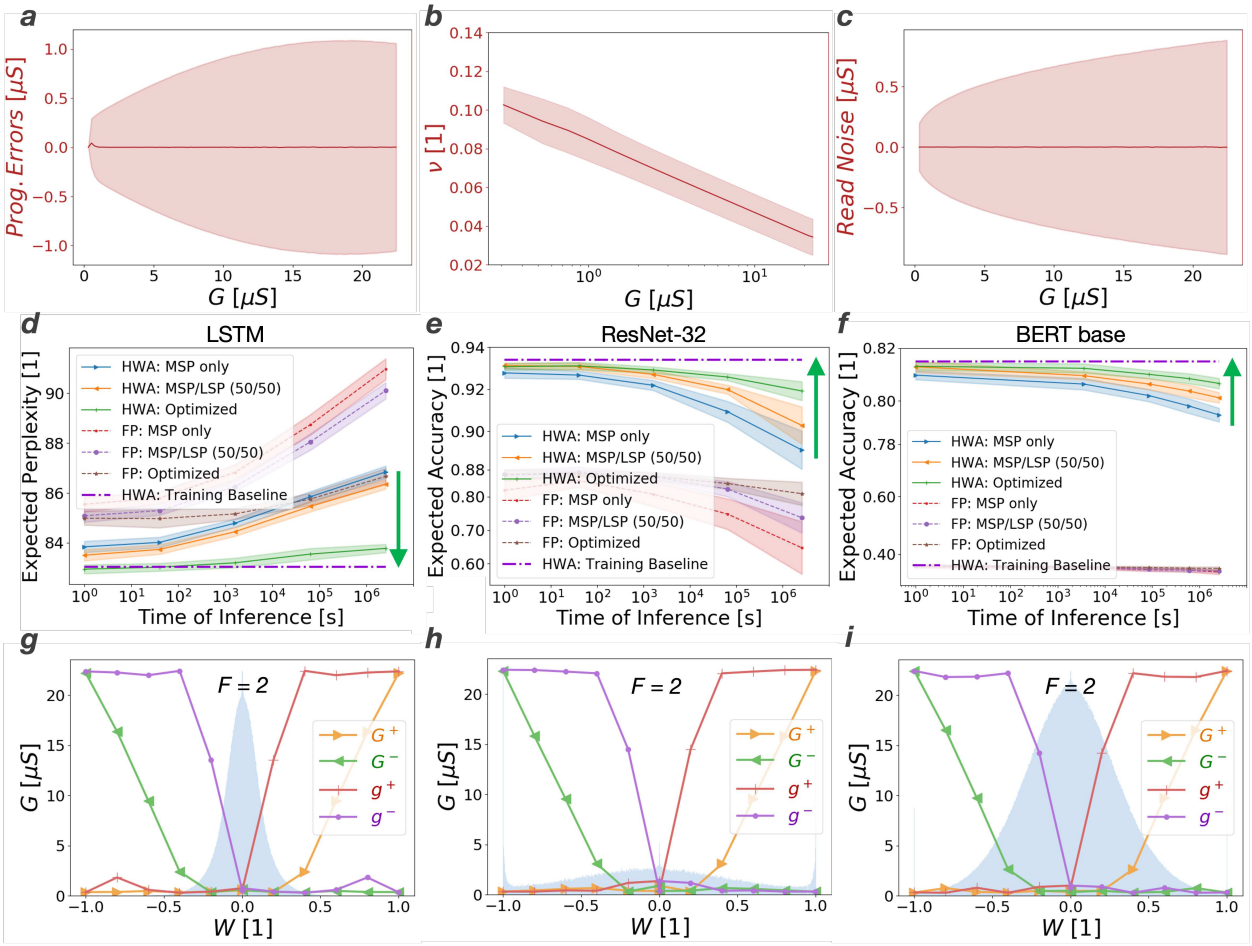

Figure 4: Underlying stochastic analogue memory device models for a) conductance-dependent programming errors, b) conductance-dependent drift coefficients, and c) conductance-dependent read noise, with solid red lines representing the mean and shaded red regions representing plus-minus one standard deviation. Inference results show the benefits of both the Hardware-Aware (HWA) training approach[22] as well as the weight programming optimization process introduced in this paper, showing good generalization across d) a recurrent neural network such as a two-layer Long Short-Term (LSTM) network evaluated on the Penn Treebank dataset, e) Convolutional Neural Networks such as ResNet-32 evaluated on the CIFAR-10 dataset, and f) Transformer-based networks such as BERT-base, evaluated on the MNLI dataset. Average inference performance and plus-minus one standard deviation are denoted by lines and shaded regions, respectively. g-i) The corresponding optimized programming strategies and inherent weight distribution for each network. Inference simulation results are compiled from twenty-five independent inference accuracy simulations over time for various training and weight programming strategies. The optimal MSP/LSP significance factor F was determined to be two in each scenario.

just for closing potential accuracy gaps, but also for establishing a way to reliably and rapidly connect device characteristics to resulting DNN accuracy. Weight programming optimization allows one to determine, for the first time to our knowledge, the expected best-case inference accuracy potential of a given set of complex analogue memory characteristics, using a modest set of simulations for each network type. As a result, we can now effectively and objectively compare proposed devices against each other in terms of best-possible DNN inference performance. Weight programming optimization then becomes a critical tool for guiding the evolution trajectory of analogue memory devices.

This is depicted in Figure 5, where the underlying analogue memory conductance drift model has been modified to a match a different Phase-Change Memory (PCM) device previously reported[26]. Despite different conductance-dependent and stochastic conductance drift models, the weight programming optimization again effectively drives the inference accuracy results as close to the hardware-aware training baseline as possible (dash-dotted line). Comparison of Figures 4 and 5 shows that the weight programming optimization technique generalizes well across different device models. This comparison also shows, however, that the analogue memory device of Figure 5 actually performs worse across the board for LSTM, CNN, and Transformer models relative to the device described in Figure 4, when both are evaluated in the limit of what is optimally achievable with either device. This is counter-intuitive because the memory device of Figure 5 provides a larger dynamic range with $g_{max} = 30\ \mu S$ and exhibits lower conductance-dependent drift on average. Furthermore, if one had compared these two devices under *naive* programming strategies, one might have incorrectly concluded that the device of Figure 5 was better (compare

the orange curves for HWA: MSP/LSP (50/50) between Figure 4d and Figure 5d).

Because our computational technique enables the extraction of optimal programming strategies and the corresponding maximum accuracy potential for each set of device characteristics, we can now more definitely say that it is preferable to implement DNNs using the device of Figure 4 than the device introduced in this section. This is a key finding. Figures 4 and 5 show that there is considerable spread or variability in inference accuracy results when sub-optimal weight programming strategies are employed. In the absence of the weight programming optimization approach introduced in this paper, this uncertainty makes it very virtually impossible to evaluate—analytically or through intuition—the true inference potential from a given set of device characteristics. Our weight programming optimization approach can thus—given a fairly modest set of conductance-programming, drift and noise characteristics (Figures 4,5a,b,c)—provide uniquely accurate feedback as to which device will eventually provide the best DNN accuracy.

Interestingly, the derived programming strategies shown in Figure 4 and 5 are quite similar to each other. This is likely because while the underlying device drift model changed between the two devices, the programming error model—which exerts a large influence on the resulting optimal programming strategy—remained quite similar. We also note that $F = 2$ produced the optimal weight programming strategy, probably because this choice increases the overall dynamic range of the weight distribution. However, one might intuitively think that this implied that one should program the bulk of the weight in the MSP first, and then use the LSP for fine tuning. In contrast, our weight programming optimization framework instead chooses to program the entirety of the

weight in the LSP whenever possible, and only makes use of the MSP for larger weights when it becomes absolutely necessary. This is because any programming errors in the MSP get amplified by the $F = 2$ factor—the strategy thus avoids such error amplification whenever possible. These types of programming strategies can be counter-intuitive at first glance, but often make sense in hindsight. The beauty of this weight programming optimization process is that it reliably automates finding these strategies, and does so in a quantitative fashion. A series of LSTM and ResNet-32 weight programming optimization results are provided in the Supplementary Information across a variety of conductance-drift models. These results provide further proof that this computational technique can reliably identify optimal programming strategies, and that the resulting inference accuracy consistently out-performs *naive* and other manually-constructed programming strategies.

Discussion

Optimal translation of software-trained weights into analogue hardware weights represents a critical step in achieving and maintaining iso-accuracy over time for analogue memory-based DNNs. We report a computational framework that automates the process of crafting complex weight programming strategies for analogue memory-based DNNs in order to minimize accuracy degradations during inference, including over time. We solve a complex and highly-flexible form of weight programming optimization, where each synaptic weight comprises of four conductances G^+, G^-, g^+, and g^- of varying significance F. The optimization framework is agnostic to all DNN features (e.g., size, layer type, activation function) with the exception of weight distribution, and is shown to consistently improve inference accuracy across a variety of networks including Long Short-Term

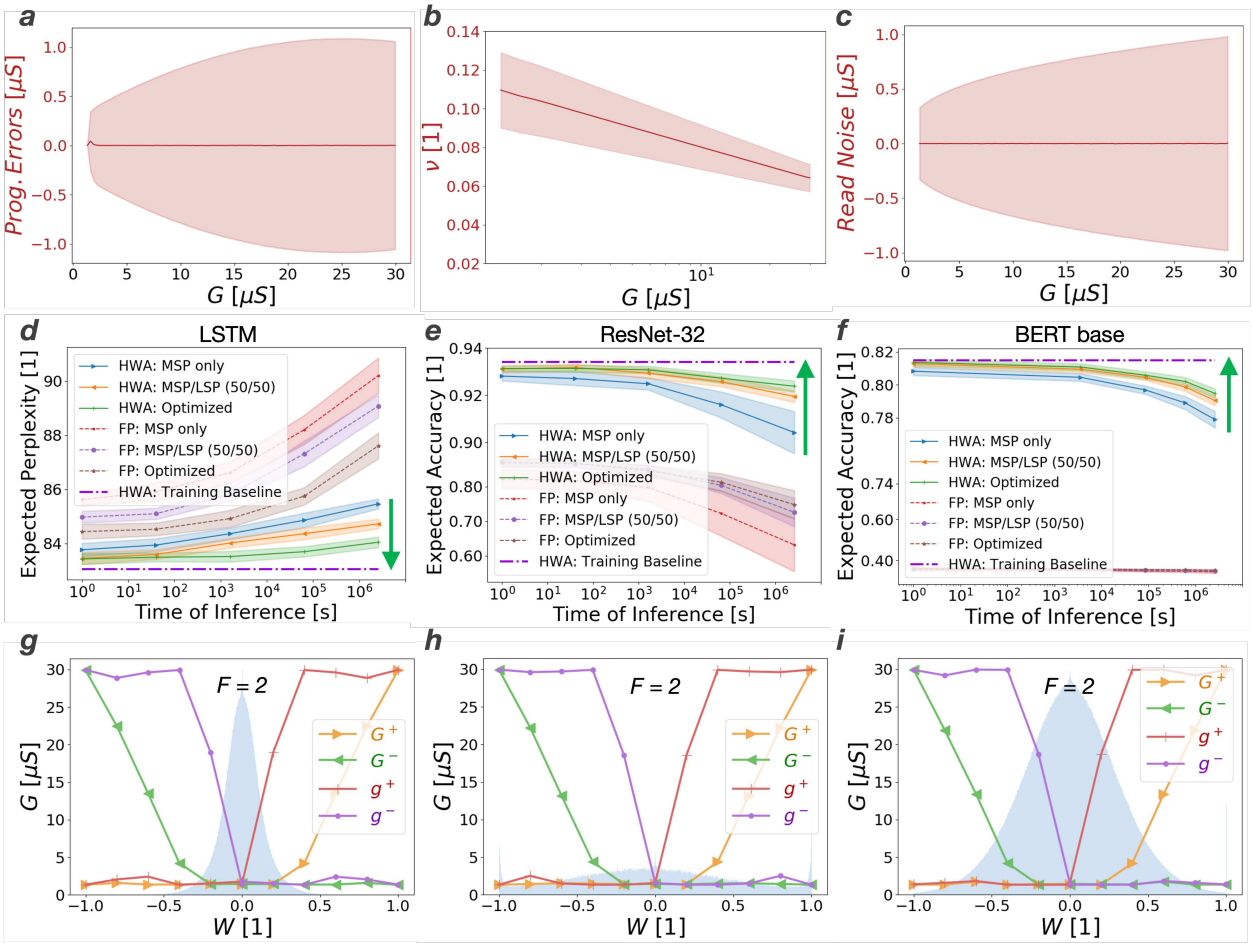

Figure 5: An alternative device that with different underlying stochastic analogue memory device models for a) conductance-dependent programming errors, b) conductance-dependent drift coefficients, and c) conductance-dependent read noise with and solid red lines representing the mean and shaded red regions representing plus-minus one standard deviation. Inference results still generalize well across a) a two-layer Long Short-Term (LSTM) network evaluated on the Penn Treebank dataset, b) ResNet-32 evaluated on the CIFAR-10 dataset, and c) BERT-base evaluated on the MNLI dataset. Although this device exhibits better performance under *naive* programming strategies (compare orange curves in part a) against Figure 4a), the best possible inference performance achievable with this device is worse than the device used for Figure 4. Average inference performance and plus-minus one standard deviation are denoted by lines and shaded regions, respectively. g-i) the corresponding optimized programming strategies for each network are similar to those in Figure 4, with only subtle changes. Simulation results are compiled from twenty-five independent inference accuracy simulations over time for various training and weight programming strategies. The optimal MSP/LSP significance factor F was determined to be two in each scenario.

Memory (LSTM), Transformer, and Convolution Neural Networks (CNNs).

This highly flexible numerical heuristic accommodates arbitrary device-level complexity, and is thus broadly applicable to a variety of analogue memories with rapidly-evolving device characteristics. Our approach also identifies the limit of achievable inference accuracy given imperfections in analogue memory. As such, this optimization framework represents a new and critical tool for enabling analogue memory-based DNNs to reach their full inference potential. Such a capability also allows analogue memory characteristics to be more objectively compared, since we can now readily evaluate the best-possible accuracy potential of new devices, as constrained by the complex and subtle interplay of their memory non-idealities. Interestingly, this computational technique optimizes inference accuracy without ever running inference simulations or evaluating training, validation, or test datasets. Weight programming optimization becomes a way to extend and augment the benefits of Hardware-Aware training, in terms of helping to reach and maintain iso-accuracy.

All weight programming in this work was performed for the entirety of the DNN weight distribution. Weight programming optimization could, however, be performed individually for each crossbar array in the network. Similarly, drift compensation can be readily performed channel-wise in hardware, implying that weight programming optimization could potentially be performed uniquely for each and every array-column within the analogue memory. While this would likely lead to additional accuracy improvements, this would not be feasible without considerable numerical acceleration of the presented technique, in order to run what will likely become hundreds of

thousands of independent weight programming optimization simulations in parallel.

It is important to emphasize that the weight programming optimization presented is not dependent on any unique hardware information and is not a form of calibration. Instead, weight programming optimization represents a one-time computational cost that should be performed for each unique DNN and unique set of underlying analogue memory device models. The optimized weight programming strategy can then be used to program all instances of that DNN into devices that exhibit those particular device characteristics. Finally, one can imagine more complex weight programming optimization frameworks that incorporate additional considerations such as minimization of energy consumption by the analogue memory. In these cases, our approach could be adapted to adopt programming strategies that drive towards both high inference accuracy while also considering the energy-efficiency implications.

Methods

Hardware-Aware Training We incorporate hardware-specific non-idealities during the forward propagation during hardware-aware training. Software weight updates during backward propagation are based on stochastic gradient descent (SGD) and carried out at full precision without additional noise. While this makes DNN models more resilient to weight errors including those resulting from conductance drift, hardware-aware training does not explicitly incorporate any conductance drift models. Later, during inference evaluation of the test dataset over time, all hardware non-idealities—MAC cycle-to-cycle non-idealities, PCM programming noise, read noise, 1/f noise, conductance-dependent drift, drift variability, and drift compensation—are considered.

Delayed Verification This works assumes that the analogue memory devices within crossbar arrays are programmed in a row-wise iterative fashion using a delayed verification strategy. Because analogue memory devices can exhibit some degree of conductance instability after the application of a programming pulse, it makes sense to maximize the time between successive programming pulses to allow the analogue memory devices as much time as possible to stabilize. We therefore cycle through the rows of the crossbar array many times while applying only one programming pulse per row (where appropriate), as opposed to programming an entire row to completion before moving onto the next row. This results in $G(t) = G_0(t/t_0)^{-\nu}$, where $t_0 = 20$ seconds. We have previously employed this weight programming time-scale as an effective compromise between conductance stability and programming speed for the programming of millions of weights[35].

Differential Weight Evolution We employ the Scipy implementation of differential evolution for its ability to effectively search large non-convex and stochastic candidate spaces. Other gradient descent-based optimizers including standalone or combinations of simulated annealing (i.e. basin-hopping) with local gradient descent-based methods were found to be ineffective at finding advantageous weight programming strategies. We fine-tune a number of parameters to work well across DNN types and varying analogue memory device models. We use a population size of 100 initialized with Latin hypercube sampling. Optimization is parallelized using the maximum available CPUs (e.g. 'workers') per compute node, which is sixteen in our case. A recombination parameter of 0.6 is used along with dithering parameters of (0.0, 0.2), which change the mutation constant on a generation by generation basis. The termination criterion is a relative tolerance (e.g. 'tol') of 0.05, meaning that the population has converged on a solution with min-

imal variation. The absolute tolerance (e.g. 'atol') was set to 0.0. We allow for a small errors around weight hyperplanes using a parameter ΔG, which provides added flexibility to slightly 'over-program' or 'under-program' weights in anticipation of non-uniform drift rates to potentially minimize the error metric more effectively. This confinement around the weight hyperplane is defined by $W - \Delta W \leq F(G^+ - G^-) + g^+ - g^- \leq W + \Delta W$, where $\Delta W \approx 2(F+1)\Delta G$. Details regarding the hypercube denormalization to capture inter-dependent conductance constraints are included in the Supplementary Information due to space limitations.

1. LeCun, Y., Bengio, Y. & Hinton, G. Deep learning. *Nature* **521**, 436–444 (2015).

2. Silver, D. *et al.* Mastering the game of Go with deep neural networks and tree search. *Nature* **529**, 484–489 (2016).

3. Hutson, M. The Language Machines: A remarkable AI can write like humans — but with no understanding of what it's saying. *Nature* **591**, 22–25 (2021).

4. Ambrogio, S. *et al.* Equivalent-accuracy neuromorphic hardware acceleration of neural network training using analog memory. *Nature* **558**, 60–67 (2018).

5. Xia, Q. & Yang, J. J. Memristive crossbar arrays for brain-inspired computing. *Nature Materials* **18**, 309–323 (2019).

6. Jang, J.-W., Park, S., Jeong, Y.-H. & Hwang, H. ReRAM-based Synaptic Device for Neuromorphic Computing. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, 1054–1057 (2014).

7. Jang, J.-W., Park, S., Burr, G. W., Hwang, H. & Jeong, Y.-H. Optimization of Conductance Change in $Pr_{1-x}Ca_xMnO_3$-Based Synaptic Devices for Neuromorphic Systems. *IEEE Elec. Dev. Lett.* **36**, 457–459 (2015).

8. Fuller, E. J. *et al.* Parallel programming of an ionic floating-gate memory array for scalable neuromorphic computing. *Science* **364**, 570–574 (2019).

9. Guo, X. *et al.* Fast, Energy-Efficient, Robust, and Reproducible Mixed-Signal Neuromorphic Classifier Based on Embedded NOR Flash Memory Technology. In *IEEE International Electron Devices Meeting*, 151–154 (2017).

10. Merrikh-Bayat, F. *et al.* High-performance mixed-signal neurocomputing with nanoscale floating-gate memory cell arrays. *IEEE Trans. Neur. Netw. Learn. Sys.* (2017).

11. Fick, L. *et al.* Analog In-Memory Subthreshold Deep Neural Network Accelerator. In *IEEE Custom Integrated Circuits Conference (CICC)*, 1–4 (2017).

12. Tadayoni, M. *et al.* Modeling split-gate flash memory cell for advanced neuromorphic computing. In *IEEE International Conference on Microelectronic Test Structures*, 27–30.

13. Burr, G. W. *et al.* Experimental demonstration and tolerancing of a large-scale neural network (165 000 synapses) using phase-change memory as the synaptic weight element. *IEEE Trans. Electr. Dev.* **62**, 3498–3507 (2015).

14. Giannopoulos, I. *et al.* 8-bit precision in-memory multiplication with projected phase-change memory. In *IEEE International Electron Devices Meeting* (2018).

15. Burr, G. W. *et al.* Neuromorphic computing using non-volatile memory. *Advances in Physics: X* **2**, 89–124 (2017).

16. Nandakumar, S. R. *et al.* Phase-change memory models for deep learning training and inference. *IEEE International Conference on Electronics, Circuits and Systems* 727–730 (2019).

17. Tsai, H., Ambrogio, S., Narayanan, P., Shelby, R. M. & Burr, G. W. Recent progress in analog memory-based accelerators for deep learning. *Journal of Physics D: Applied Physics* **51**, 283001 (2018).

18. Sebastian, A. *et al.* Tutorial: Brain-inspired computing using phase-change memory devices. *Journal of Applied Physics* **124**, 111101 (2018).

19. Joshi, V. *et al.* Accurate deep neural network inference using computational phase-change memory. *Nature Communications* **11**, 1–13 (2020).

20. Sun, X. *et al.* Ultra-low precision 4-bit training of deep neural networks. In *Advances in Neural Information Processing Systems*, vol. 33, 1796–1807 (2020).

21. Agrawal, A. *et al.* 9.1 a 7nm 4-core AI chip with 25.6 TFLOPS hybrid FP8 training, 102.4 TOPS INT4 inference and workload-aware throttling. In *IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 64, 144–146 (2021).

22. Rasch, M. *et al.* A flexible and fast pytorch toolkit for simulating training and inference on analog crossbar arrays. *IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)* 1–4 (2021).

23. Kariyappa, S. *et al.* Noise-Resilient DNN: Tolerating Noise in PCM-Based AI Accelerators via Noise-Aware Training. *IEEE Transactions on Electron Devices* **68**, 1–7 (2021).

24. Spoon, K. *et al.* Toward Software-Equivalent Accuracy on Transformer-Based Deep Neural Networks With Analog Memory Devices. *Frontiers in Computational Neuroscience* **15**, 1–9 (2021).

25. Boniardi, M. *et al.* Statistics of resistance drift due to structural relaxation in phase-change memory arrays. *IEEE Transactions on Electron Devices* **57**, 2690–2696 (2010).

26. Bruce, R. *et al.* Mushroom-type phase change memory with projection liner: An array-level demonstration of conductance drift and noise mitigation. In *2021 IEEE International Reliability Physics Symposium (IRPS)*, 1–6 (2021).

27. Mackin, C. *et al.* Weight Programming in DNN Analog Hardware Accelerators in the Presence of NVM Variability. *Advanced Electronic Materials* 1–12 (2019).

28. 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*, 1–4 (2019).

29. Storn, R. & Price, K. Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces. *Journal of Global Optimization* **11**, 341–359 (1997).

30. Taylor, A., Marcus, M. & Santorini, B. The Penn Treebank: An overview (2003).

31. Krizhevsky, A. Learning multiple layers of features from tiny images (2009).

32. Williams, A., Nangia, N. & Bowman, S. R. A broad-coverage challenge corpus for sentence understanding through inference. *CoRR* **abs/1704.05426** (2017).

33. Sebastian, A., Le Gallo, M., Khaddam-Aljameh, R. & Eleftheriou, E. Memory devices and applications for in-memory computing. *Nature Nanotechnology* **15**, 529–544 (2020).

34. Pedretti, G. *et al.* Redundancy and analog slicing for precise in-memory machine learning–part I: Programming techniques. In *IEEE Transactions on Electron Devices*, vol. 68, 1–6 (2021).

35. Nandakumar, S. R. *et al.* Precision of synaptic weights programmed in phase-change memory devices for deep learning inference. In *IEEE International Electron Devices Meeting (IEDM)*, 1–4 (2020).

Competing Interests The authors declare that they have no competing financial interests.

Correspondence Correspondence and requests for materials should be addressed to Charles Mackin (email: charles.mackin@ibm.com).

Supplementary Files

This is a list of supplementary files associated with this preprint. Click to download.

- OptimizedWeightProgrammingforAnalogueAISupplement.pdf