LOW-RANK TRAINING OF DEEP NEURAL NETWORKS FOR EMERGING MEMORY TECHNOLOGY

Anonymous authors
Paper under double-blind review

ABSTRACT

The recent success of neural networks for solving difficult decision tasks has incentivized incorporating smart decision making "at the edge." However, this work has traditionally focused on neural network *inference*, rather than *training*, due to memory and compute limitations, especially in emerging non-volatile memory systems, where writes are energetically costly and reduce lifespan. Yet, the ability to train at the edge is becoming increasingly important as it enables applications such as real-time adaptability to device drift and environmental variation, user customization, and federated learning across devices. In this work, we address four key challenges for training on edge devices with non-volatile memory: low weight update density, weight quantization, low auxiliary memory, and online learning. We present a low-rank training scheme that addresses these four challenges while maintaining computational efficiency. We then demonstrate the technique on a representative convolutional neural network across several adaptation problems, where it out-performs standard SGD both in accuracy and in number of weight updates.

1 INTRODUCTION

Deep neural networks have shown remarkable performance on a variety of challenging inference tasks. As the energy efficiency of deep-learning inference accelerators improves, some models are now being deployed directly to edge devices to take advantage of increased privacy of user data, reduced network bandwidth, and lower inference latency. Despite edge deployment, training happens predominately in the cloud. This limits some of the privacy advantages of running models on-device and results in static models that do not adapt to evolving data distributions in the field.

Efforts aimed at on-device training are solving some of these challenges. Federated learning aims to keep data on-device by training models in a distributed fashion (Konecný et al., 2016). On-device model customization has been achieved by techniques such as weight-imprinting (Qi et al., 2018), or by retraining limited sets of layers. And on-chip training has also been demonstrated for handling analog hardware imperfections in 6-transistor SRAM cells (Zhang et al., 2017; Gonugondla et al., 2018). Despite this progress with small models, on-chip training of larger models is bottlenecked by limited memory size and compute horsepower available in processors designed for edge use.

Emerging non-volatile (NVM) memories such as resistive random access memory (RRAM) have shown great promise as energy and area-efficient inference engines (Yu, 2018). However, on-chip training requires a large number of writes to the memory, and RRAM writes are significantly more energy intensive compared to reads. Additionally, RRAM endurance is on the order of 10^6 writes (Grossi et al., 2019), shortening the lifetime of a device as the memory undergoes writes for on-chip training.

In this paper, we present an online training scheme amenable to NVM memories to enable next generation smart edge devices. Our contributions are (1) an algorithm called Streaming Kronecker Sum Approximation (SKS) which addresses the four key challenges of low weight update density, weight quantization, low auxiliary memory, and online learning; (2) two techniques "gradient max-norm" and "streaming batch norm" to help training specifically in the online setting; (3) a suite of adaptation experiments to demonstrate the advantages of our approach.

2 RELATED WORK

Resistive hardware arrays for NN acceleration. There is currently great interest in accelerating machine-learning with resistive arrays due to prospects of fundamentally higher density and energy efficiency over digital architectures. Small arrays of analog-programmable purely resistive (1R) crossbars (Prezioso et al., 2015), and 1-transistor 1-resistor (1T-1R) cells (Yao et al., 2017; Li et al., 2018) have been demonstrated for single and dual-layer perceptron algorithms. In these works, high cell-to-cell and chip-to-chip variation is overcome by including the chip inside the training loop. Though most work incorporates off-chip hardware and software for chip-in-the-loop training, a 54 x 108 resistive crossbar array was recently combined with on-chip ADCs and DACs and an OpenRISC processor (Cai et al., 2019), marking a move towards completely on-chip training. The largest of the analog arrays is 128 x 64, however much larger resistive arrays with quantized 2-level weight storage have demonstrated increasingly complex tasks, including a 16 Mb array for an MNIST task (Yu et al., 2016), and a 1 Mb array for a CIFAR-10 task (Xue et al., 2019).

Efficient training for resistive arrays. Several works have aimed at improving the efficiency of training algorithms on resistive arrays. Weight updates are the most challenging to accelerate, as forward and backward propagation can be run fully-in-parallel on the array. Stochastic weight update schemes (Gokmen & Vlasov, 2016) allow programming of all cells in a crossbar at once, as opposed to row-wise or column-wise updating. Online Manhattan rule updating (Zamanidoost et al., 2015) can be similarly used to update all of the weights at once. Several works have proposed new memory structures to improve the efficiency of training (Soudry et al., 2015; Ambrogio et al., 2018). The number of writes has also been quantified in the context of chip-in-the-loop training of a binary network for MNIST (Yu et al., 2016).

Distributed gradient descent. Distributed training in the data center is another problem that suffers from expensive weight updates. In this scenario, the model is replicated onto many compute nodes and in each training iteration, the mini-batch is split across the nodes to compute gradients. The distributed gradients are then accumulated on a central node which computes the updated weights and broadcasts them to all devices. These systems can be limited by communication bandwidth, and compressed gradient techniques (Aji & Heafield, 2017) have been developed to address this. In Lin et al. (2017), the gradients are accumulated over multiple training iterations on each compute node and only gradients which exceed a threshold are communicated back to the central node. In the context of on-chip training with weights in NVM, this method helps to reduce the number of weight updates. However, the gradient accumulator requires as much memory as the weights themselves which negates the density benefits of NVM.

3 CHALLENGES ADDRESSED

Neural networks are typically trained using a variant of stochastic gradient descent (SGD) on large hardware such as a GPUs or FPGAs. When trying to adapt these training techniques for smaller devices with NVM memory, we run into a number of challenges which are described below.

Low weight update density. In NVM, updating weights at every sample is extremely costly in energy and endurance. For example, the 18 KB RRAM macro described in (Wu et al., 2019) consumes 10.9 pJ/b for writes versus only 1.76 pJ/b for reads. Additionally, as previously described, some technologies suffer from poor endurance, limiting the number of writes to less than 10^6. For both reasons, we want to minimize the number of writes to NVM.

Weight quantization. To leverage emerging NVM for on-chip weight storage, quantization of weights is necessary. Low bitwidths may be important for mixed-signal applications, while digital realizations of NVM may allow for higher bitwidths.

Low auxiliary memory. NVM is the most dense form of memory storage. For example, in 40nm technology, RRAM 1T-1R bitcells @ 0.085 um^2 (Chou et al., 2018) are 2.8x smaller than standard 6-transistor SRAM bitcells @ 0.242 um^2 (TSMC, 2019). Therefore, NVM should be used to store the memory-intensive weights. A corollary of this is that no other on-chip memory should come close to the size of the on-chip NVM. More explicitly, if our $b-$bit NVM stores a weight matrix of size $n \times m$, then we would like to use at most $r(n+m)b$ auxiliary non-NVM memory where r is a small constant.

Online learning. Devices should process data in real time, making small- or single-size batches important (Mahdavinejad et al., 2018). Single size batches also decrease the memory requirements. It is unclear whether SGD with batch size of 1 is compatible with efficient learning (Sahoo et al., 2017), and especially when considering quantization (Li et al., 2017). SGD minibatches typically have batch sizes in excess of 32, which results in at least $32\times$ higher activation storage costs.

In the next section, we address these four challenges in separate steps, leading to the SKS algorithm.

4 Optimal Kronecker Sum Approximation

4.1 Motivation

Consider a linear regression task of finding a matrix $\boldsymbol{W}^{opt} \in \mathbb{R}^{n\times m}$ that minimizes the L_2 loss $\mathcal{L} = \frac{1}{2}||\boldsymbol{z}-\boldsymbol{t}||_2^2 = \frac{1}{2}||\boldsymbol{W}^{opt}\boldsymbol{a}-\boldsymbol{t}||_2^2$, where $\boldsymbol{a}, \boldsymbol{z}, \boldsymbol{t}$ are input features, predicted outputs, and true outputs, respectively. $\boldsymbol{W}^{opt}$ can be found from some initial $\boldsymbol{W}^{(0)}$ through gradient descent with gradients $\nabla_{\boldsymbol{W}}\mathcal{L} = (\boldsymbol{W}\boldsymbol{a}-\boldsymbol{t})\cdot\boldsymbol{a}^\top$. For convenience and to tie it to the notation for the full neural network case, we will represent the errors as $\boldsymbol{dz} = \boldsymbol{W}\boldsymbol{a}-\boldsymbol{t}$. Then, minibatch SGD with learning rate α and batch size B gives us:

$$\boldsymbol{W}^{(j)} = \boldsymbol{W}^{(j-1)} - \alpha \sum_{i=jB+1}^{(j+1)B} (\boldsymbol{dz}^{(i)} \otimes \boldsymbol{a}^{(i)}) \tag{1}$$

where $\boldsymbol{dz}^{(i)} \otimes \boldsymbol{a}^{(i)}$ represents an outer product $\boldsymbol{dz}^{(i)} \cdot \boldsymbol{a}^{(i)\top}$, $\boldsymbol{dz}^{(i)} = \boldsymbol{W}^{(\lfloor i/B \rfloor)}\boldsymbol{a}^{(i)} - \boldsymbol{t}^{(i)}$ and the optimal $\boldsymbol{W}^{opt} = \lim_{j\to\infty} \boldsymbol{W}^{(j)}$.

This formulation requires large amounts of auxiliary memory. Either $B(n+m)b$ bits of memory are required to store $\boldsymbol{dz}^{(i)}$ and $\boldsymbol{a}^{(i)}$ or partial sums can be taken that require a scratch space of nmb bits of memory. What we need is a potentially lossy way to capture the important information in B samples using only $r(n+m)b$ bits of auxiliary memory where $r \ll B$.

In Sections 4.2, 4.3, 4.4, we build up to a method that allows for this computation. In Section 4.5 we analyze how it solves all of the challenges from Section 3. Finally, in Section 4.6 we present an efficient algorithm for implementing the technique on-chip.

4.2 Towards a Low-Memory Kronecker Sum Approximation

The critical term we need to compute is the Kronecker sum from (1):

$$\sum_{i=jB+1}^{(j+1)B} (\boldsymbol{dz}^{(i)} \otimes \boldsymbol{a}^{(i)}) \tag{2}$$

One way to approximate this computation with $r(n+m)b$ bits, where $r \ll B$, is to maintain approximate prefix sums of the computation in a rank-r representation $\boldsymbol{L}\cdot\boldsymbol{R}^\top$ where $\boldsymbol{L} \in \mathbb{R}^{n\times r}$ and $\boldsymbol{R} \in \mathbb{R}^{m\times r}$. Explicitly, we iteratively update $\boldsymbol{L}, \boldsymbol{R}$ as:

$$\tilde{\boldsymbol{L}}, \tilde{\boldsymbol{R}} := \text{optLR}(\boldsymbol{L}\cdot\boldsymbol{R}^\top + \boldsymbol{dz}^{(i)} \otimes \boldsymbol{a}^{(i)}) \tag{3}$$

for $i = jB$ to $(j+1)B$ where *optLR* minimizes reconstruction error between $\tilde{\boldsymbol{L}}\cdot\tilde{\boldsymbol{R}}^\top$ and $\boldsymbol{L}\cdot\boldsymbol{R}^\top + \boldsymbol{dz}^{(i)} \otimes \boldsymbol{a}_{(i)}$ while maintaining the rank-r constraint. This can be accomplished by selecting the top r components of a singular value decomposition (SVD) of $\boldsymbol{L}\cdot\boldsymbol{R}^\top + \boldsymbol{dz}^{(i)} \otimes \boldsymbol{a}^{(i)}$.

Unfortunately, this solution introduces new roadblocks. First, it requires computing $\boldsymbol{L}\cdot\boldsymbol{R}^\top + \boldsymbol{dz}^{(i)} \otimes \boldsymbol{a}^{(i)}$ explicitly, which already uses nmb bits of memory. Second, an SVD on an $n \times m$ matrix is computationally expensive, taking $\mathcal{O}(n\, m \cdot \min(n,m))$ time. A more subtle issue is that selecting the top-r components of an SVD produces a biased estimate of the actual matrix of interest.

4.3 UNBIASED ONLINE RECURRENT OPTIMIZATION (UORO)

One potential solution is to use the UORO technique (Tallec & Ollivier, 2017). From Proposition 1, they find a rank-1 approximation $\tilde{\boldsymbol{A}}$ of a matrix $\boldsymbol{A} = \sum_{i=1}^{k} \boldsymbol{v}_i \otimes \boldsymbol{w}_i$ as:

$$\tilde{A} = \left(\sum_{i=1}^{k} \rho_i \nu_i \boldsymbol{v}_i\right) \otimes \left(\sum_{i=1}^{k} \frac{\nu_i \boldsymbol{w}_i}{\rho_i}\right) \tag{4}$$

where ν is a vector of independent random signs and ρ is a vector of scales that can be used to reduce the variance of $\tilde{A}$. From (4), it is clear that we can get by with $(n+m)b$ bits of memory by maintaining two matrices $\boldsymbol{L} \in \mathbb{R}^{n\times 1}, \boldsymbol{R} \in \mathbb{R}^{m\times 1}$ which we update with $\rho_i \nu_i \boldsymbol{v}_i$ or $\nu_i \boldsymbol{w}_i / \rho_i$, respectively, for each sample $\boldsymbol{v}_i \otimes \boldsymbol{w}_i$ that comes in. This solution solves all of the roadblocks from Section 4.2, however it now suffers from high variance and limited rank, $r = 1$.

4.4 OPTIMAL KRONECKER SUM APPROXIMATION (OK)

To solve the problems introduced by UORO, we instead employ the "OK" algorithm (Benzing et al., 2019), which is shown to produce the minimum variance rank-r unbiased estimate for a Kronecker sum of q terms. We specifically focus on the case where $q = r + 1$ since repeated application of this method allows us to maintain a rank-r approximation regardless of how many training samples we see.

The OK algorithm can be understood in two key steps: first, an efficient method of computing the SVD of a Kronecker sum; second, a method of splitting the singular value matrix $\boldsymbol{\Sigma}$ into two rank-r matrices whose outer product is a minimum-variance, unbiased estimate of $\boldsymbol{\Sigma}$. Rigorous details can be found in their paper, however we include a high-level explanation in Sections 4.4.1 and 4.4.2 to aid discussions in following sections. Note that our variable notation differs from Benzing et al. (2019).

4.4.1 EFFICIENT SVD OF KRONECKER SUMS

Consider a Kronecker sum of $q = r + 1$ terms, which we can rewrite as:

$$\sum_{i=1}^{q} (\boldsymbol{dz}^{(i)} \otimes \boldsymbol{a}^{(i)}) = \boldsymbol{L}\boldsymbol{R}^\top \tag{5}$$

where the i^{th} column of $\boldsymbol{L} \in \mathbb{R}^{n\times q}$ is $\boldsymbol{dz}^{(i)}$ and the i^{th} column of $\boldsymbol{R} \in \mathbb{R}^{m\times q}$ is $\boldsymbol{a}^{(i)}$. In $\mathcal{O}((n+m)q^2)$ time, we can QR-factorize $\boldsymbol{L} = \boldsymbol{Q}_L \boldsymbol{R}_L$ and $\boldsymbol{R} = \boldsymbol{Q}_R \boldsymbol{R}_R$ where $\boldsymbol{Q}_L \in \mathbb{R}^{n\times q}, \boldsymbol{Q}_R \in \mathbb{R}^{m\times q}$ are orthogonal so that $\boldsymbol{L}\boldsymbol{R}^\top = \boldsymbol{Q}_L(\boldsymbol{R}_L \boldsymbol{R}_R^\top)\boldsymbol{Q}_R^\top$ (Björck, 1967). Let $\boldsymbol{C} = \boldsymbol{R}_L \boldsymbol{R}_R^\top \in \mathbb{R}^{q\times q}$. Then we can find the SVD of $\boldsymbol{C} = \boldsymbol{U}_C \boldsymbol{\Sigma} \boldsymbol{V}_C^\top$ in $\mathcal{O}(q^3)$ time (Cline & Dhillon, 2006), making it computationally feasible on small devices. Now we have:

$$\boldsymbol{L}\boldsymbol{R}^\top = \boldsymbol{Q}_L(\boldsymbol{U}_C \boldsymbol{\Sigma} \boldsymbol{V}_C^\top)\boldsymbol{Q}_R^\top = (\boldsymbol{Q}_L \boldsymbol{U}_C)\boldsymbol{\Sigma}(\boldsymbol{Q}_R \boldsymbol{V}_C)^\top \tag{6}$$

which gives us the SVD of $\boldsymbol{L}\boldsymbol{R}^\top$ since $\boldsymbol{Q}_L \boldsymbol{U}_C$ and $\boldsymbol{Q}_R \boldsymbol{V}_C$ are orthogonal and $\boldsymbol{\Sigma}$ is diagonal. Overall, this SVD computation has a time complexity of $\mathcal{O}((n+m+q)q^2)$ and a space complexity of $\mathcal{O}((n+m)q)$.

4.4.2 MINIMUM VARIANCE, UNBIASED ESTIMATE OF $\boldsymbol{\Sigma}$

In Benzing et al. (2019), it is shown that the problem of finding a rank-r minimum variance unbiased estimator of $\boldsymbol{L}\boldsymbol{R}^\top$ can be reduced to the problem of finding a rank-r minimum variance unbiased estimator of $\boldsymbol{\Sigma}$ and plugging it in to (6).

Further, it is shown that such an optimal approximator for $\boldsymbol{\Sigma} = \text{diag}(\sigma_1, \sigma_2, \ldots, \sigma_q)$, where $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_q$ will involve keeping the largest singular values and mixing the smaller singular values $\sigma_m, \ldots, \sigma_q$ using a technique inspired by the random signs of UORO. Let:

$$m = \min\ i \ \text{ s.t. } \ (q-i)\sigma_i \leq \sum_{j=i}^{q} \sigma_j \qquad\qquad s_1 = \sum_{i=m}^{q} \sigma_i \qquad\qquad k = q - m$$

$$\boldsymbol{x}_0 = \left(\sqrt{1 - \frac{\sigma_m k}{s_1}}, \ldots, \sqrt{1 - \frac{\sigma_q k}{s_1}} \right)^\top \qquad\qquad \mathbf{s} \in \{-1, 1\}^{(k+1)\times 1}$$

where $\mathbf{s}$ are uniform random signs. Note that $||\boldsymbol{x}_0||_2 = 1$. Let $\boldsymbol{X} \in \mathbb{R}^{(k+1)\times(k)}$ be orthogonal such that its left nullspace is the span of $\boldsymbol{x}_0$. Then $\boldsymbol{X} \cdot \boldsymbol{X}^\top = I - \boldsymbol{x}_0 \cdot \boldsymbol{x}_0^\top$. Now, let:

$$\boldsymbol{X}_s = (\mathbf{s} \odot \boldsymbol{X}_{:,1}, \ldots, \mathbf{s} \odot \boldsymbol{X}_{:,k}) \qquad\qquad \boldsymbol{Z} = \sqrt{\frac{s_1}{k}} \cdot \boldsymbol{X}_s$$

$$\tilde{\boldsymbol{\Sigma}}_L = \tilde{\boldsymbol{\Sigma}}_R = \text{diag}\left(\sqrt{\sigma_1}, \ldots, \sqrt{\sigma_{m-1}}, \boldsymbol{Z}\right) \tag{7}$$

where $\odot$ is an element-wise product. Then $\tilde{\boldsymbol{\Sigma}}_L \cdot \tilde{\boldsymbol{\Sigma}}_R^\top = \tilde{\boldsymbol{\Sigma}}$ is a minimum variance, unbiased rank-r approximation of $\boldsymbol{\Sigma}$. Plugging $\tilde{\boldsymbol{\Sigma}}$ into (6),

$$\boldsymbol{L}\boldsymbol{R}^\top = (\boldsymbol{Q}_L \boldsymbol{U}_C)\boldsymbol{\Sigma}(\boldsymbol{Q}_R \boldsymbol{V}_C)^\top \approx (\boldsymbol{Q}_L \boldsymbol{U}_C)\tilde{\boldsymbol{\Sigma}}(\boldsymbol{Q}_R \boldsymbol{V}_C)^\top = (\boldsymbol{Q}_L \boldsymbol{U}_C \tilde{\boldsymbol{\Sigma}}_L)(\boldsymbol{Q}_R \boldsymbol{V}_C \tilde{\boldsymbol{\Sigma}}_R)^\top \tag{8}$$

Thus, $\tilde{\boldsymbol{L}} = \boldsymbol{Q}_L \boldsymbol{U}_C \tilde{\boldsymbol{\Sigma}}_L \in \mathbb{R}^{n\times r}$ and $\tilde{\boldsymbol{R}} = \boldsymbol{Q}_R \boldsymbol{V}_C \tilde{\boldsymbol{\Sigma}}_R \in \mathbb{R}^{m\times r}$ gives us a minimum variance, unbiased, rank-r approximation $\tilde{\boldsymbol{L}} \cdot \tilde{\boldsymbol{R}}^\top$.

4.5 Benefits of Method

The key benefit of the OK algorithm is that it is decouples the quantization bitwidth, low auxiliary memory, and sparse update requirements.

Previously, a small batch size could be employed to satisfy the low auxiliary memory constraint at the expense of having dense, low-magnitude weight updates. Large bitwidth weights would also be required, since weight updates must be proportionally smaller leading to a larger dynamic range (Goyal et al., 2017). On the other hand, a large effective batch size could be employed to reduce the density of updates per training sample and the weight bitwidths at the expense of requiring more expensive high-endurance memory for storing the $\boldsymbol{a}^{(i)}, \boldsymbol{dz}^{(i)}$.

With the OK algorithm, the batch size (B), can be made arbitrarily large without a increasing memory requirements. It is also inherently online, since each incoming sample is incorporated into a rank-r running approximation of the batch. Therefore, it addresses the challenges discussed in Section 3.

There are two new costs introduced by increasing the effective batch size, however. First, the network will be slower to respond to changes in the training data distribution. One way to address this is to update a small subset of the learnable parameters — the 1-dimensional tensor weights such as biases — at each training sample, since they can be made high bitwidth and high-endurance without incurring a heavy memory penalty. Second, the variance of our batch approximation grows with the number of samples. We address this issue by showing empirically in Section 6 that relatively large B of hundreds, or even thousands for convolutions, still permit training.

4.6 Streaming Kronecker Sum Approximation (SKS)

Although the standalone OK algorithm presented by Benzing et al. (2019) has good asymptotic computational complexity, there are some useful optimizations that can improve performance even further. In this section we present these optimizations, and we refer readers to the explicit implementation called Streaming Kronecker Sum Approximation (SKS) in Algorithm 1 of Appendix A.

4.6.1 Maintain Orthogonal $\boldsymbol{Q}_L, \boldsymbol{Q}_R$

The main optimization is a method of avoiding recomputing the QR factorization of $\boldsymbol{L}$ and $\boldsymbol{R}$ at every step. Instead, we keep track of orthogonal matrices $\boldsymbol{Q}_L, \boldsymbol{Q}_R$, and weightings $\boldsymbol{c}_x$ such that

$\tilde{\boldsymbol{L}} = \boldsymbol{Q}_L \cdot \text{diag}(\sqrt{\boldsymbol{c}_x})_{[:r]}$ and $\tilde{\boldsymbol{R}} = \boldsymbol{Q}_R \cdot \text{diag}(\sqrt{\boldsymbol{c}_x})_{[:r]}$. Upon receiving a new sample, a single inner loop of the numerically-stable modified Gram-Schmidt (MGS) algorithm (Björck, 1967) can be used to update $\boldsymbol{Q}_L$ and $\boldsymbol{Q}_R$. The orthogonal basis coefficients $\boldsymbol{c}_L = \boldsymbol{Q}_L^\top \cdot \boldsymbol{dz}^{(i)}$ and $\boldsymbol{c}_R = \boldsymbol{Q}_R^\top \cdot \boldsymbol{a}^{(i)}$ computed during MGS can be used to find the new value of $\boldsymbol{C} = \boldsymbol{c}_L \cdot \boldsymbol{c}_R^\top + \text{diag}(\boldsymbol{c}_x)$.

After computing $\tilde{\boldsymbol{\Sigma}}_L = \tilde{\boldsymbol{\Sigma}}_R$ in (7), we can orthogonalize these matrices into $\tilde{\boldsymbol{\Sigma}}_L = \tilde{\boldsymbol{\Sigma}}_R = \boldsymbol{Q}_x \boldsymbol{R}_x$. Then from (8), we have $\tilde{\boldsymbol{L}}\tilde{\boldsymbol{R}}^\top = (\boldsymbol{Q}_L \boldsymbol{U}_C \boldsymbol{Q}_x)(\boldsymbol{R}_x \boldsymbol{R}_x^\top)(\boldsymbol{Q}_R \boldsymbol{V}_C \boldsymbol{Q}_x)^\top$. With this formulation, we can maintain orthogonality in $\boldsymbol{Q}_L, \boldsymbol{Q}_R$ by setting:

$$\boldsymbol{Q}_L \leftarrow \boldsymbol{Q}_L \boldsymbol{U}_C \boldsymbol{Q}_x \qquad \boldsymbol{Q}_R \leftarrow \boldsymbol{Q}_R \boldsymbol{V}_C \boldsymbol{Q}_x \qquad \boldsymbol{c}_x \leftarrow \text{diag}(\boldsymbol{R}_x \boldsymbol{R}_x^\top)$$

These matrix multiplies require $\mathcal{O}((n+m)q^2)$ multiplications, so this optimization does not improve asymptotic complexity bounds. This optimization may nonetheless be practically significant since matrix multiplies are easy to parallelize and would typically not be the bottleneck of the computation compared to Gram-Schmidt.

In the next section we discuss how to orthogonalize $\tilde{\boldsymbol{\Sigma}}_L$ efficiently and why $(\boldsymbol{R}_x \boldsymbol{R}_x^\top)$ is diagonal.

4.6.2 Orthogonalization of $\tilde{\boldsymbol{\Sigma}}_L$

Orthogonalization of $\tilde{\boldsymbol{\Sigma}}_L$ is relatively straightforward. From (7), the columns of $\tilde{\boldsymbol{\Sigma}}_L$ are orthogonal since $\boldsymbol{Z}$ is orthogonal. However, they do not have unit norm. We can therefore pull out the norm into a separate diagonal matrix $\boldsymbol{R}_x$ with diagonal elements $\sqrt{\boldsymbol{c}_x}$:

$$\boldsymbol{Q}_x = \begin{bmatrix} \boldsymbol{I}_{m-1} & 0 \\ 0 & \boldsymbol{X}_s \end{bmatrix} \qquad \sqrt{\boldsymbol{c}_x} = (\sqrt{\sigma_1}, \ldots, \sqrt{\sigma_{m-1}}, \underbrace{\sqrt{s_1/k}}_{q-m+1 \text{ times}})$$

4.6.3 Finding Orthonormal Basis $\boldsymbol{X}$

We generated $\boldsymbol{X}$ by finding an orthonormal basis that was orthogonal to a vector $\boldsymbol{x}_0$ so that we could have $\boldsymbol{X} \cdot \boldsymbol{X}^\top = I - \boldsymbol{x}_0 \cdot \boldsymbol{x}_0^\top$. An efficient method of producing this basis is through Householder matrices $(\boldsymbol{x}_0, \boldsymbol{X}) = \boldsymbol{I} - 2\, \boldsymbol{v} \cdot \boldsymbol{v}^\top / ||\boldsymbol{v}||^2$ where $\boldsymbol{v} = \boldsymbol{x}_0 - \boldsymbol{e}^{(1)}$ and $(\boldsymbol{x}_0, \boldsymbol{X})$ is a $k+1 \times k+1$ matrix with first column $\boldsymbol{x}_0$ and remaining columns $\boldsymbol{X}$ (Householder, 1958; user1551, 2013).

4.6.4 Efficiency Comparisons to Standard Approach

The OK/SKS methods require $\mathcal{O}((n+m+q)q^2)$ operations per sample and $\mathcal{O}(nmq)$ operations after collecting B samples, giving an amortized cost of $\mathcal{O}((n+m+q)q^2 + nmq/B)$ operations per sample. Meanwhile, a standard approach expands the Kronecker sum at each sample, costing $\mathcal{O}(nm)$ operations per sample. If $q \ll B, n, m$ then the low rank method is both memory and computationally simpler than standard minibatch SGD.

On generic computational hardware with heavy parallelization, the asymptotic benefits of SKS are obscured by the speed of GEMM methods. However, on smaller edge devices where absolute number of FLOPs is important for energy, the computational benefits may be more apparent. It is worth noting, however, that computational efficiency of SKS is secondary to the goal of memory efficiency.

5 Implementation Details

Here, we briefly summarize the remaining pieces necessary to train a neural network using SKS.

Applicability to Neural Networks. In Section 4.1, we motivated SKS with a linear regression example. However, the technique can be generalized to gradients of weight matrices in most neural networks. For example, Benzing et al. (2019) applies the technique to RNNs. In CNNs, there are two important layer types: fully-connected layers and convolutional layers. Application of SKS to fully-connected layers is straightforward, since it is nearly identical to the motivating linear regression

example. For convolution layers, convolutions can be converted into matrix multiplies through the `im2col` operation, reducing the problem to just another linear regression, in this case on a per-pixel rather than per-sample basis. See Appendices B.1 and B.2 for further elaboration.

Quantization. The network is quantized in both the forward and backward directions with uniform power-of-2 quantization, where the clipping ranges are fixed at the start of training[1]. Weights are quantized to 8 bits between -1 and 1, biases to 16 bits between -8 and 8, activations to 8 bits between 0 and 2, and gradients to 8 bits between -1 and 1. Both weights $\boldsymbol{W}$ and the weight updates $\Delta\boldsymbol{W}$ are quantized so that weights cannot be used for accumulation beyond the fixed quantization dynamic range. This is in contrast to the much simpler problem of standard quantization-in-the-loop training, where weights accumulate in floating point, but are quantized during forward inference. See Appendix C for more explicit details on quantization.

Gradient Max-Norming. State-of-the-art methods in training, such as Adam (Kingma & Ba, 2014), often utilize auxiliary memory per parameter to normalize the gradients. Unfortunately, we lack the memory budget to support this many additional variables, especially if they must be updated every sample. SKS could potentially approximate Adam. SKS on a^2, dz^2 allows for a low-rank approximation of the variance of the gradients assuming a, dz are zero-mean. This is unlikely to work well in practice because of numerical stability where, e.g., estimated variances might be negative. Instead, we propose dividing each gradient tensor by the max absolute-value of elements in the gradient tensor. This stabilizes the dynamic range of gradients across samples. See Appendix D for more details on gradient max-norming. In the experiments, we refer to this method as "max-norm" and lack of this method as "no-norm."

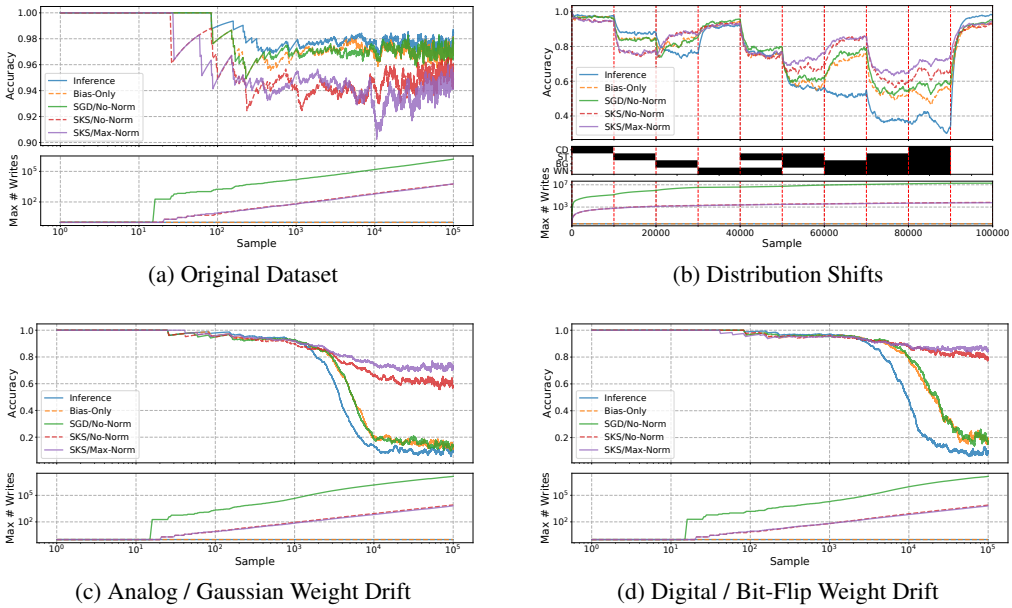
Streaming Batch Normalization. Batch normalization (Ioffe & Szegedy, 2015) is a powerful technique for improving training performance which has been suggested to work by smoothing the loss landscape (Santurkar et al., 2018). We hypothesize that this may be especially helpful when parameters are quantized as in our case. However, in the online setting, we receive samples one-at-a-time rather than in batches. We therefore propose a streaming batch norm that uses moving average statistics rather than batch statistics as described in detail in Appendix E.

6 EXPERIMENTS

To test the effectiveness of SKS, experiments are performed on a representative CNN comprising four 3×3 convolution layers and two fully-connected layers. We generate "offline" and "online" datasets based on MNIST as described in Appendix F, including one in which the statistical distribution shifts every 10k images. We then optimize an online SGD and SKS model for fair comparison, as described in Appendix G. To see the importance of different training techniques, we run several ablations in Appendix H. Finally, we compare these different training schemes in different environments, meant to model real life. In these hypothetical scenarios, a model is first trained on the offline training set, then is deployed to a number of devices at the edge, where they must make supervised predictions — they make a prediction, then are told what the correct prediction would have been.

We present results on four hypothetical scenarios. First, a control case where both external/environment and internal/NVM drift statistics are exactly the same as during offline training. Second, a case where the input image statistical distribution shifts every 10k samples, selecting from augmentations such as spatial transforms and background gradients, as described in Section F. Third and fourth are cases where the internal NVM drifts from the programmed values, roughly modeling NVM memory degredation. In the third case, Gaussian noise is applied to the weights as if each weight was a single multi-level memory cell whose analog value drifted in a Brownian way. In the fourth case, random bit flips are applied as if each weight was represented by b memory cells, which through random drift may flip from a 0 to 1 or vice versa (see Appendix F for additional details). For each hypothetical scenario, we plot five different training schemes: pure quantized inference (no training), bias-only training, standard SGD training, SKS training, and SKS training with max-normed gradients. In SGD training and for training biases, parameters are updated at every step in an online fashion. These are seen as different colored curves in Figure 1.

[1]Interesting future work might look into how to change these clipping ranges, but that is beyond the scope of our research.



(a) Original Dataset

(b) Distribution Shifts

(c) Analog / Gaussian Weight Drift

(d) Digital / Bit-Flip Weight Drift

Figure 1: Adaptation of various training schemes over four different training environments (a) to (d). In each training environment, the top plot shows the exponential moving averages (0.999) of the per-sample online accuracy of the five training schemes, while the bottom plot shows the maximum number of updates applied to any given convolution or fully-connected kernel memory cell. For the distribution shifts in (b), the enabled augmentations at each contiguous 10k samples is shown (CD = class distribution, ST = spatial transforms, BG = background gradients, WN = white noise).

Inference does best in the control case, but does poorly in adaptation experiments. SGD does not improve significantly on bias-only training, possibly because SGD can not accumulate gradients less than a weight LSB. SKS, on the other hand, shows significant improvement, especially after several thousand samples in the weight drift cases. Additionally, SKS shows an approximately three-order-of-magnitude improvement compared to SGD in the worst case maximum number of weight updates. Much of this reduction is due to the convolutions, where weight updates are applied at each pixel. However, reduction in fully-connected writes is still important because of the potential energy savings. SKS/max-norm performs best in terms of accuracy across all environments and has similar weight update cost to SKS/no-norm. Finally, we note that weights can adapt rapidly as shown in the distribution shifts experiment.

7 Conclusion

In this work, we have demonstrated the potential for SKS to solve the major challenges facing online training on NVM-based edge devices: low weight update density, weight quantization, low auxiliary memory, and online learning. SKS is a computationally-efficient, memory-light algorithm capable of significantly reducing write operations to memory during training. Additionally, it may allow for training under severe weight quantization constraints as rudimentary gradient accumulations are handled by the L, R matrices, which can have high bitwidths, rather than the weight tensors themselves. Across a variety of online adaptation problems, SKS is shown to have performance matching or exceeding SGD or inference-only while using a small fraction of the number of updates of SGD. Finally, we suspect that these techniques could be applied to a broader range of problems. Auxiliary memory minimization for our application may be analogous to communication minimization in training strategies such as federated learning, where gradient compression is important. Our methods may also offer another advantage here - security from the compression of hundreds of samples into a low-rank update. There are many promising avenues for future research in this area.

REFERENCES

Alham Fikri Aji and Kenneth Heafield. Sparse communication for distributed gradient descent. *arXiv preprint arXiv:1704.05021*, 2017.

Stefano Ambrogio, Pritish Narayanan, Hsinyu Tsai, Robert M. Shelby, Irem Boybat, Carmelo di Nolfo, Severin Sidler, Massimo Giordano, Martina Bodini, Nathan C. P. Farinha, Benjamin Killeen, Christina Cheng, Yassine Jaoudi, and Geoffrey W. Burr. Equivalent-accuracy accelerated neural-network training using analogue memory. *Nature*, 558(7708):60–67, June 2018. ISSN 1476-4687. doi: 10.1038/s41586-018-0180-5.

Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

Frederik Benzing, Marcelo Matheus Gauy, Asier Mujika, Anders Martinsson, and Angelika Steger. Optimal Kronecker-sum approximation of real time recurrent learning. In Kamalika Chaudhuri and Ruslan Salakhutdinov (eds.), *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 604–613, Long Beach, California, USA, 09–15 Jun 2019. PMLR. URL `http://proceedings.mlr.press/v97/benzing19a.html`.

Åke Björck. Solving linear least squares problems by gram-schmidt orthogonalization. *BIT Numerical Mathematics*, 7(1):1–21, 1967.

Fuxi Cai, Justin M. Correll, Seung Hwan Lee, Yong Lim, Vishishtha Bothra, Zhengya Zhang, Michael P. Flynn, and Wei D. Lu. A fully integrated reprogrammable memristor–CMOS system for efficient multiply–accumulate operations. *Nature Electronics*, 2(7):290–299, July 2019. ISSN 2520-1131. doi: 10.1038/s41928-019-0270-x.

C. Chou, Z. Lin, P. Tseng, C. Li, C. Chang, W. Chen, Y. Chih, and T. J. Chang. An N40 256K$\times$44 embedded RRAM macro with SL-precharge SA and low-voltage current limiter to improve read and write performance. In *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, pp. 478–480, February 2018. doi: 10.1109/ISSCC.2018.8310392.

Alan Kaylor Cline and Inderjit S Dhillon. Computation of the singular value decomposition, 2006.

M. Ernestus. Elastic transformation of an image in python. `https://gist.github.com/erniejunior/601cdf56d2b424757de5`, 2016.

Tayfun Gokmen and Yurii Vlasov. Acceleration of Deep Neural Network Training with Resistive Cross-Point Devices. *Frontiers in Neuroscience*, 10, July 2016. ISSN 1662-453X. doi: 10.3389/fnins.2016.00333.

S. K. Gonugondla, M. Kang, and N. R. Shanbhag. A Variation-Tolerant In-Memory Machine Learning Classifier via On-Chip Training. *IEEE Journal of Solid-State Circuits*, 53(11):3163–3173, November 2018. doi: 10.1109/JSSC.2018.2867275.

Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

A. Grossi, E. Vianello, M. M. Sabry, M. Barlas, L. Grenouillet, J. Coignus, E. Beigne, T. Wu, B. Q. Le, M. K. Wootters, C. Zambelli, E. Nowak, and S. Mitra. Resistive ram endurance: Array-level characterization and correction techniques targeting deep learning applications. *IEEE Transactions on Electron Devices*, 66(3):1281–1288, March 2019. doi: 10.1109/TED.2019.2894387.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.

Alston S Householder. Unitary triangularization of a nonsymmetric matrix. *Journal of the ACM (JACM)*, 5(4):339–342, 1958.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Jakub Konecný, H. Brendan McMahan, Daniel Ramage, and Peter Richtárik. Federated optimization: Distributed machine learning for on-device intelligence. *CoRR*, abs/1610.02527, 2016. URL http://arxiv.org/abs/1610.02527.

Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Can Li, Daniel Belkin, Yunning Li, Peng Yan, Miao Hu, Ning Ge, Hao Jiang, Eric Montgomery, Peng Lin, Zhongrui Wang, Wenhao Song, John Paul Strachan, Mark Barnell, Qing Wu, R. Stanley Williams, J. Joshua Yang, and Qiangfei Xia. Efficient and self-adaptive in-situ learning in multilayer memristor neural networks. *Nature Communications*, 9(1):2385, June 2018. ISSN 2041-1723. doi: 10.1038/s41467-018-04484-2.

Hao Li, Soham De, Zheng Xu, Christoph Studer, Hanan Samet, and Tom Goldstein. Training quantized nets: A deeper understanding. In *Advances in Neural Information Processing Systems*, pp. 5811–5821, 2017.

Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

Fuyun Ling, Dimitris Manolakis, and John Proakis. A recursive modified gram-schmidt algorithm for least-squares estimation. *IEEE transactions on acoustics, speech, and signal processing*, 34(4): 829–836, 1986.

Mohammad Saeid Mahdavinejad, Mohammadreza Rezvan, Mohammadamin Barekatain, Peyman Adibi, Payam Barnaghi, and Amit P Sheth. Machine learning for internet of things data analysis: A survey. *Digital Communications and Networks*, 4(3):161–175, 2018.

M. Prezioso, F. Merrikh-Bayat, B. D. Hoskins, G. C. Adam, K. K. Likharev, and D. B. Strukov. Training and operation of an integrated neuromorphic network based on metal-oxide memristors. *Nature*, 521(7550):61–64, May 2015. ISSN 1476-4687. doi: 10.1038/nature14441.

Hang Qi, Matthew Brown, and David G. Lowe. Low-Shot Learning with Imprinted Weights. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 5822–5830, Salt Lake City, UT, June 2018. IEEE. ISBN 978-1-5386-6420-9. doi: 10.1109/CVPR.2018.00610.

Jimmy SJ Ren and Li Xu. On vectorization of deep convolutional neural networks for vision tasks. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

Doyen Sahoo, Quang Pham, Jing Lu, and Steven CH Hoi. Online deep learning: Learning deep neural networks on the fly. *arXiv preprint arXiv:1711.03705*, 2017.

Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? In *Advances in Neural Information Processing Systems*, pp. 2483–2493, 2018.

Patrice Y. Simard, Dave Steinkraus, and John Platt. Best practices for convolutional neural networks applied to visual document analysis. Institute of Electrical and Electronics Engineers, Inc., August 2003. URL https://www.microsoft.com/en-us/research/publication/best-practices-for-convolutional-neural-networks-applied-to-visual-document-analysis/.

D. Soudry, D. Di Castro, A. Gal, A. Kolodny, and S. Kvatinsky. Memristor-Based Multilayer Neural Networks With Online Gradient Descent Training. *IEEE Transactions on Neural Networks and Learning Systems*, 26(10):2408–2421, October 2015. doi: 10.1109/TNNLS.2014.2383395.

Corentin Tallec and Yann Ollivier. Unbiased online recurrent optimization. *arXiv preprint arXiv:1702.05043*, 2017.

TSMC. 40nm Technology - Taiwan Semiconductor Manufacturing Company Limited. https://www.tsmc.com/english/dedicatedFoundry/technology/40nm.htm, 2019.

user1551. Rotation matrix in arbitrary dimension to align vector. Mathematics Stack Exchange, 2013. URL `https://math.stackexchange.com/q/525587`. URL:https://math.stackexchange.com/q/525587 (version: 2013-10-14).

T. F. Wu, B. Q. Le, R. Radway, A. Bartolo, W. Hwang, S. Jeong, H. Li, P. Tandon, E. Vianello, P. Vivet, E. Nowak, M. K. Wootters, H. . P. Wong, M. M. S. Aly, E. Beigne, and S. Mitra. 14.3 a 43pj/cycle non-volatile microcontroller with 4.7s shutdown/wake-up integrating 2.3-bit/cell resistive ram and resilience techniques. In *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, pp. 226–228, Feb 2019. doi: 10.1109/ISSCC.2019.8662402.

C. Xue, W. Chen, J. Liu, J. Li, W. Lin, W. Lin, J. Wang, W. Wei, T. Chang, T. Chang, T. Huang, H. Kao, S. Wei, Y. Chiu, C. Lee, C. Lo, Y. King, C. Lin, R. Liu, C. Hsieh, K. Tang, and M. Chang. 24.1 A 1Mb Multibit ReRAM Computing-In-Memory Macro with 14.6ns Parallel MAC Computing Time for CNN Based AI Edge Processors. In *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, pp. 388–390, February 2019. doi: 10.1109/ISSCC.2019.8662395.

Peng Yao, Huaqiang Wu, Bin Gao, Sukru Burc Eryilmaz, Xueyao Huang, Wenqiang Zhang, Qingtian Zhang, Ning Deng, Luping Shi, H.-S. Philip Wong, and He Qian. Face classification using electronic synapses. *Nature Communications*, 8:15199, May 2017. ISSN 2041-1723. doi: 10.1038/ncomms15199.

S. Yu, Z. Li, P. Chen, H. Wu, B. Gao, D. Wang, W. Wu, and H. Qian. Binary neural network with 16 Mb RRAM macro chip for classification and online training. In *2016 IEEE International Electron Devices Meeting (IEDM)*, pp. 16.2.1–16.2.4, December 2016. doi: 10.1109/IEDM.2016.7838429.

Shimeng Yu. Neuro-inspired computing with emerging nonvolatile memorys. *Proceedings of the IEEE*, 106(2):260–285, 2018.

E. Zamanidoost, F. M. Bayat, D. Strukov, and I. Kataeva. Manhattan rule training for memristive crossbar circuit pattern classifiers. In *2015 IEEE 9th International Symposium on Intelligent Signal Processing (WISP) Proceedings*, pp. 1–6, May 2015. doi: 10.1109/WISP.2015.7139171.

J. Zhang, Z. Wang, and N. Verma. In-Memory Computation of a Machine-Learning Classifier in a Standard 6T SRAM Array. *IEEE Journal of Solid-State Circuits*, 52(4):915–924, April 2017. ISSN 0018-9200. doi: 10.1109/JSSC.2016.2642198.

A SKS ALGORITHM

Algorithm 1 Streaming Kronecker Sum Approximation

State: $\boldsymbol{Q}_L \in \mathbb{R}^{n\times q}$; $\boldsymbol{Q}_R \in \mathbb{R}^{m\times q}$; $\boldsymbol{c}_x \in \mathbb{R}^{q\times 1}$
Input: $\boldsymbol{dz}^{(i)} \in \mathbb{R}^{n\times 1}$; $\boldsymbol{a}^{(i)} \in \mathbb{R}^{m\times 1}$ for $i \in [1, B]$
for $i = 1 \dots B$ **do**
  {Modified Gram-Schmidt.}
  $\boldsymbol{c}_L, \boldsymbol{c}_R \leftarrow 0^{q\times 1}$
  for $j = 1 \dots r$ **do**
    $c_{L,j} \leftarrow \boldsymbol{Q}_{L,j} \cdot \boldsymbol{dz}^{(i)}; \quad \boldsymbol{dz}^{(i)} \leftarrow \boldsymbol{dz}^{(i)} - c_{L,j} \cdot \boldsymbol{Q}_{L,j}$
    $c_{R,j} \leftarrow \boldsymbol{Q}_{R,j} \cdot \boldsymbol{a}^{(i)}; \quad \boldsymbol{a}^{(i)} \leftarrow \boldsymbol{a}^{(i)} - c_{L,j} \cdot \boldsymbol{Q}_{L,j}$
  end for
  $c_{L,q} \leftarrow ||\boldsymbol{dz}^{(i)}||; \quad \boldsymbol{Q}_{L,q} \leftarrow \boldsymbol{dz}^{(i)}/c_{L,q}$
  $c_{R,q} \leftarrow ||\boldsymbol{a}^{(i)}||; \quad \boldsymbol{Q}_{R,q} \leftarrow \boldsymbol{a}^{(i)}/c_{R,q}$
  {Generate $\boldsymbol{C}$ and find its SVD.}
  $\boldsymbol{C} \leftarrow \boldsymbol{c}_L \cdot \boldsymbol{c}_R^\top + \text{diag}(\boldsymbol{c}_x)$
  $\boldsymbol{U}_C \cdot \text{diag}(\boldsymbol{\sigma}) \cdot \boldsymbol{V}_C^\top \leftarrow \text{SVD}(\boldsymbol{C})$
  {Minimum-variance unbiased estimator for $\boldsymbol{\Sigma}$.}
  $m \leftarrow \min\ j \ \text{ s.t. } \ (q-j)\sigma_j \le \sum_{\ell=j}^{q} \sigma_\ell$
  $s_1 \leftarrow \sum_{i=m}^{q} \sigma_i, \quad k \leftarrow q - m$
  $\boldsymbol{v} \leftarrow \sqrt{1 - k/s_1 \cdot \boldsymbol{\sigma}_{[m:]}} - \boldsymbol{e}^{(1)}$
  $\mathbf{s} \leftarrow \{-1, 1\}^{(k+1)\times 1}$ {Ind. uniform random signs.}
  $\boldsymbol{X}_s \leftarrow \left(\boldsymbol{I} + (\mathbf{s} \odot \boldsymbol{v}) \cdot (\boldsymbol{v}/v_1)^\top\right)_{[2:]}$ {Householder.}
  {QR-factorization of $\tilde{\boldsymbol{\Sigma}}_L$.}
  $\boldsymbol{Q}_x \leftarrow \begin{bmatrix} \boldsymbol{I} & 0 \\ 0 & \boldsymbol{X}_s \end{bmatrix} \in \mathbb{R}^{q\times r}$
  $\boldsymbol{c}_x \leftarrow (\sigma_1, \dots, \sigma_{m-1}, \underbrace{s_1/k, \dots, s_1/k}_{q-m+1 \text{ times}})$
  {Update the first r columns of $\boldsymbol{Q}_L, \boldsymbol{Q}_R$.}
  $\boldsymbol{Q}_{L[:r]} \leftarrow \boldsymbol{Q}_L \cdot \boldsymbol{U}_C \cdot \boldsymbol{Q}_x$
  $\boldsymbol{Q}_{R[:r]} \leftarrow \boldsymbol{Q}_R \cdot \boldsymbol{V}_C \cdot \boldsymbol{Q}_x$
end for
{Compute final $\tilde{\boldsymbol{L}}, \tilde{\boldsymbol{R}}$ where $\nabla_{\boldsymbol{W}}\mathcal{L} \approx \tilde{\boldsymbol{L}} \cdot \tilde{\boldsymbol{R}}^\top$.}
$\tilde{\boldsymbol{L}} \leftarrow \left(\boldsymbol{Q}_L \cdot \text{diag}(\sqrt{\boldsymbol{c}_x})\right)_{[:r]}$
$\tilde{\boldsymbol{R}} \leftarrow \left(\boldsymbol{Q}_R \cdot \text{diag}(\sqrt{\boldsymbol{c}_x})\right)_{[:r]}$

B KRONECKER SUMS IN NEURAL NETWORK LAYERS

B.1 DENSE LAYER

A dense or fully-connected layer transforms an input $\boldsymbol{a} \in \mathbb{R}^{m\times 1}$ to an intermediate $\boldsymbol{z} = \boldsymbol{W} \cdot \boldsymbol{a} + \boldsymbol{b}$ to an output $\boldsymbol{y} = \sigma(\boldsymbol{z}) \in \mathbb{R}^{n\times 1}$ where σ is a non-linear activation function. Gradients of the loss function with respect to the weight parameters can be found as:

$$\nabla_{\boldsymbol{W}}\mathcal{L} = \underbrace{(\nabla_{\boldsymbol{z}}\mathcal{L})}_{\boldsymbol{dz}} \odot \underbrace{(\nabla_{\boldsymbol{W}}\boldsymbol{z})}_{\boldsymbol{a}^\top} = \boldsymbol{dz} \otimes \boldsymbol{a} \tag{9}$$

which is exactly the per-sample Kronecker sum update we saw in linear regression. Thus, at every training sample, we can add $(\boldsymbol{dz}^{(i)} \otimes \boldsymbol{a}^{(i)})$ to our low rank estimate with SKS.

B.2 CONVOLUTIONAL LAYER

A convolutional layer transforms an input feature map $\mathbf{A} \in \mathbb{R}^{h_{in} \times w_{in} \times c_{in}}$ to an intermediate feature map $\mathbf{Z} = \mathbf{W}_{kern} * \mathbf{A} + \boldsymbol{b} \in \mathbb{R}^{h_{out} \times w_{out} \times c_{out}}$ through a 2D convolution $*$ with weight kernel $\mathbf{W}_{kern} \in \mathbb{R}^{c_{out} \times k_h \times k_w \times c_{in}}$. Then it computes an output feature map $y = \sigma(z)$ where σ is a non-linear activation function.

Convolutions can be interpreted as matrix multiplications through the `im2col` operation which converts the input feature map $\mathbf{A}$ into a matrix $\boldsymbol{A}_{col} \in \mathbb{R}^{(h_{out} w_{out}) \times (k_h k_w c_{in})}$ where the i^{th} row is a flattened version of the sub-tensor of a which is dotted with $\mathbf{W}_{kern}$ to produce the i^{th} pixel of the output feature map (Ren & Xu, 2015). We can multiply $\boldsymbol{A}_{col}$ by a flattened version of the kernel, $\boldsymbol{W} \in \mathbb{R}^{c_{out} \times (k_h h_w c_{in})}$ to perform the $\mathbf{W}_{kern} * \mathbf{A}$ convolution operation with a matrix multiplication. Under the matrix multiplication interpretation, weight gradients can be represented as:

$$\nabla_{\boldsymbol{W}} \mathcal{L} = \underbrace{(\nabla_{\boldsymbol{Z}_{col}} \mathcal{L})}_{\boldsymbol{dZ}_{col}^\top} \odot \underbrace{(\nabla_{\boldsymbol{W}} \boldsymbol{Z})}_{\boldsymbol{A}_{col}} = \sum_{i=1}^{h_{out} w_{out}} \boldsymbol{dZ}_{col,i}^\top \otimes \boldsymbol{A}_{col,i}^\top \tag{10}$$

which is the same as $h_{out} w_{out}$ Kronecker sum updates. Thus, at every output pixel j of every training sample i, we can add $(\boldsymbol{dZ}_{col,j}^{(i)\top} \otimes \boldsymbol{A}_{col,j}^{(i)\top})$ to our low rank estimate with SKS.

Note that while we already save an impressive factor of B/q in memory when computing gradients for the dense layer, we save a much larger factor of $B h_{out} w_{out}/q$ in memory when computing gradients for the convolution layers, making the low rank training technique even more crucial here.

However, some care must be taken when considering activation memory for convolutions. For compute-constrained edge devices, image dimensions may be small and result in minimal intermediate feature map memory requirements. However, if image dimensions grow substantially, activation memory could dominate compared to weight storage. Clever dataflow strategies may provide a way to reduce intermediate activation storage even when performing backpropagation[2].

C HARDWARE QUANTIZATION MODEL

In a real device, operations are expected to be performed in fixed point arithmetic. Therefore, all of our training experiments are conducted with quantization in the loop. Our model for quantization is shown in Figure 2. The green arrows describe the forward computation. Ignoring quantization for a moment, we would have $a^\ell = \text{ReLU}\left(\alpha^\ell W^\ell * a^{\ell-1} + b^\ell\right)$, where $*$ can represent either a convolution or a matrix multiply depending on the layer type and α^ℓ is the closest power-of-2 to He initialization (He et al., 2015). For quantization, we rely on four basic quantizers: Qw, Qb, Qa, Qg, which describe weight quantization, bias and intermediate accumulator quantization, activation quantization, and gradient quantization, respectively. All quantizers use fixed clipping ranges as depicted and quantize uniformly within those ranges to the specified bitwidths.

In the backward pass, follow the orange arrows from δ^ℓ. Backpropagation follows standard backpropagation rules including using the straight-through estimator (Bengio et al., 2013) for quantizer gradients. However, because we want to perform training on edge devices, these gradients must themselves be quantized. The first place this happens is after passing backward through the ReLU derivitive. The other two places are before feeding back into the network parameters W^ℓ, b^ℓ, so that W^ℓ, b^ℓ cannot be used to accumulate values smaller than their LSB. Finally, instead of deriving ΔW^ℓ from a backward pass through the $*$ operator, the SKS method is used.

SKS collects $a^{\ell-1}, dz^\ell$ for many samples before computing the approximate $\Delta \tilde{W}^\ell$. It accumulates information in two low rank matrices L, R which are themselves quantized to 16 bits with clipping ranges determined dynamically by the max absolute value of elements in each matrix. While SKS

[2]For example, one could compute just a sliding window of rows of every feature map, discarding earlier rows as later rows are computed, resulting in a square-root reduction of activation memory. To incorporate backpropagation, compute the forward pass once fully, then compute the forward pass again, as well as the backward pass using the sliding window approach in both directions.

accumulates for B samples, leading to a factor of B reduction in the rate of updates to W^ℓ, b^ℓ is updated at every sample. This is feasible in hardware because b^ℓ is small enough to be stored in more expensive forms of memory that have superior endurance and write power performance.

Because of the coarse weight LSB size, weight gradients may be consistently quantized to 0, preventing them from accumulating. To combat this, we only apply an update if a minimum update density $\rho_{\text{min}} = 0.01$ would be achieved, otherwise we continue accumulating samples in L and R, which have much higher bitwidths. When an update does finally happen, the "effective batch size" will be a multiple of B and we increase the learning rate correspondingly. In the literature, a linear scaling rule is suggested (see Goyal et al. (2017)), however we empirically find square-root scaling works better (see Appendix G).

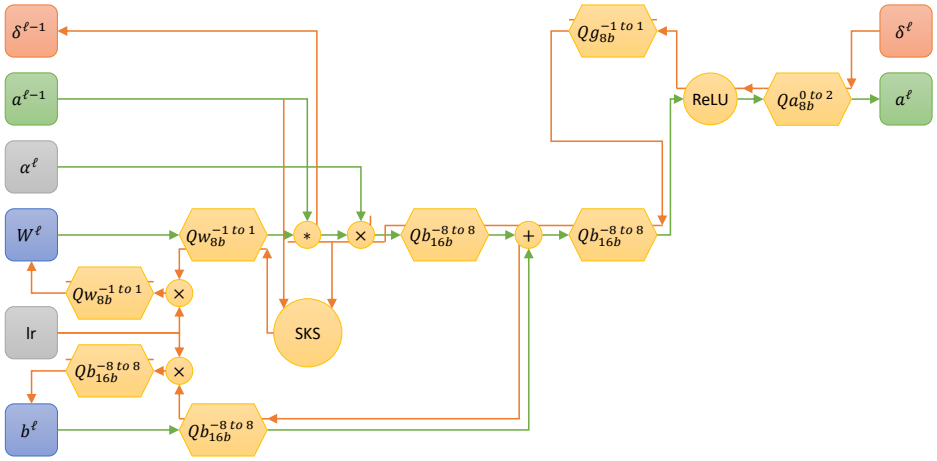


Figure 2: Signal flow graph for a forward and backward quantized convolutional or dense layer.

D GRADIENT MAX-NORMING

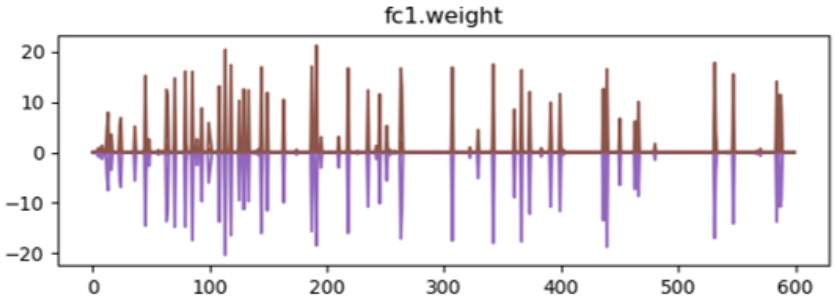


Figure 3: Maximum magnitude of weight gradients versus training step for standard SGD on a CNN trained on MNIST.

Figure 3 plots the magnitude of gradients seen in a weight tensor over training steps. One apparent property of these gradients is that they have a large dynamic range, making them difficult to quantize. Even when looking at just the spikes, they assume a wide range of magnitudes. One potential method of dealing with this dynamic range is to scale tensors so that their max absolute element is 1. Optimizers such as Adam, which normalize by gradient variance, provide a justification for why this sort of scaling might work well, although they work at a per-element rather than per-tensor level. We choose max-norming rather than variance-based norming because the former is easier computational and potentially more ammenable to quantization. However, a problem with the approach of normalizing tensors independently at each sample is that noise might be magnified during

regions of quiet as seen in the Figure. What we therefore propose is normalization by the maximum of both the current max element and a moving average of the max element.

Explicitly, max-norm takes two parameters - a decay factor $\beta = 0.999$ and a gradient floor $\varepsilon = 10^{-4}$ and keeps two state variables - the number of evaluations $k := 0$ and the current maximum moving average $x_{mv} := \varepsilon$. Then for a given input x, max-norm modifies its internal state and returns x_{norm}:

$$\begin{aligned} k &:= k + 1 \\ x_{max} &:= \max(|x|) + \varepsilon \\ x_{mv} &:= \beta \cdot x_{mv} + (1 - \beta) \cdot x_{max} \\ \tilde{x}_{mv} &:= \frac{x_{mv}}{1 - \beta^k} \\ x_{norm} &:= \frac{x}{\max(x_{max}, \tilde{x}_{mv})} \end{aligned}$$

E STREAMING BATCH NORMALIZATION

Standard batch normalization (Ioffe & Szegedy, 2015) normalizes a tensor x along some axes, then applies a trainable affine transformation:

$$y = \gamma \cdot \frac{x - \mu_b}{\sqrt{\sigma_b^2 + \varepsilon}} + \beta$$

where μ_b, σ_b are mean and standard deviation statistics of a minibatch and γ, β are trainable affine transformation parameters.

In our case, we do not have the memory to hold a batch of samples at a time and must compute μ_b, σ_b in an online fashion. To see how this works, suppose we knew the statistics of each sample μ_i, σ_i for $i = 1 \dots B$ in a batch of B samples. For simplicity, assume the i^{th} sample is a vector $x_i \in \mathbb{R}^n$ containing elements x_{ij}. Then:

$$\mu_b = \frac{1}{B} \sum_{i=1}^{B} \mu_i \tag{11}$$

$$\sigma_b^2 = \frac{1}{B} \sum_{i=1}^{B} \frac{1}{n} \sum_{j=1}^{n} x_{ij}^2 - \mu_b^2 = \frac{1}{B} \sum_{i=1}^{B} \left(\sigma_i^2 + \mu_i^2\right) - \mu_b^2 \neq \frac{1}{B} \sum_{i=1}^{B} \sigma_i^2 \tag{12}$$

In other words, the batch variance is not equal to the average of the sample variances. However, if we keep track of the sum-of-square values of samples $\sigma_i^2 + \mu_i^2$, then we can compute σ_b^2 as in (12). We keep track of two state variables: μ_s, sq_s which we update as $\mu_s := \mu_s + \mu_i$ and $sq_s := sq_s + \sigma_i^2 + \mu_i^2$ for each sample i. After B samples, we divide both state variables by B and apply (11, 12) to get the desired batch statistics. Unfortunately, in an online setting, all samples prior to the last one in a given batch will only see statistics generated from a portion of the batch, resulting in noisier estimates of μ_b, σ_b.

In *streaming* batch norm, we alter the above formula slightly. Notice that in online training, only the most recently viewed sample is used for training, so there is no reason to weight different samples of a given batch equally. Therefore we can use an exponential moving average instead of a true average to track μ_s, sq_s. Specifically, let:

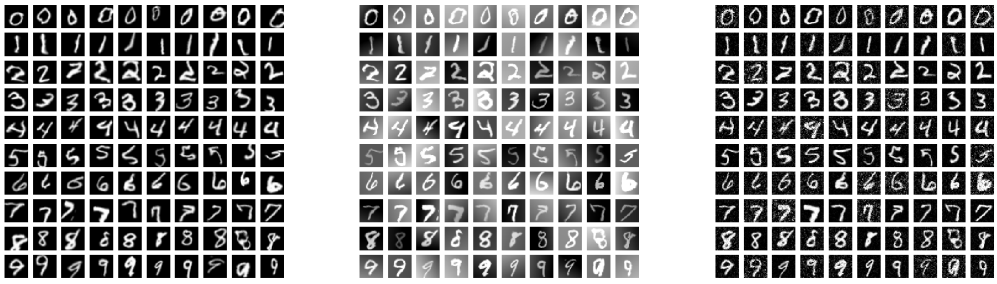
$$\begin{aligned} \mu_s &:= \eta \cdot \mu_s + (1 - \eta) \cdot \mu_i \\ sq_s &:= \eta \cdot sq_s + (1 - \eta) \cdot (\sigma_i^2 + \mu_i^2) \end{aligned}$$

If we set $\eta = 1 - 1/B$, a weighting of $1/B$ is seen on the current sample, just as in standard averages with a batch of size B, but now all samples receive similarly clean batch statistic estimates, not just the last few samples in a batch.

F ONLINE DATASET

For our experiments, we construct a dataset comprising an offline training, validation, and test set, as well as an online training set. Specifically, we start with the standard MNIST dataset of LeCun et al. (1998) and split the 60k training images into partitions of size 9k, 1k, and 50k. Elastic transforms (Simard et al., 2003; Ernestus, 2016) are used to augment each of these partitions to 50k offline training samples, 10k offline validation samples, and 100k online training samples, respectively. Elastic transforms are also applied to the 10k MNIST test images to generate the offline test samples.

From the online training set, we also generate a "distribution shift" dataset by applying unique additional augmentations to every contiguous 10k samples of the 100k online training samples. Four types of augmentations are explored. Class distribution clustering biases training samples belonging to similar classes to have similar indices. For example, the first thousand images may be primarily "0"s and "3"s, whereas the next thousand might have many "5"s. Spatial transforms rotate, scale, and shift images by random amounts. Background gradients both scale the contrast of the images and apply black-white gradients across the image. Finally, white noise is random Gaussian noise added to each pixel. Figure 4 shows some representative examples of what these augmentations look like. The augmentations are meant to mimic different external environments an edge devices might need to adapt to.



(a) Spatial Transforms (b) Background Grads (c) White Noise

Figure 4: Samples of different types of distribution shift augmentations.

In addition to distribution shift for testing adaptation, we also look at internal statistical shift of weights in two ways - analog and digital. For analog weight drift, we apply independent additive Gaussian noise to each weight every $d = 10$ steps with $\sigma = \sigma_0/\sqrt{1M/d}$ where $\sigma_0 = 10$ and re-clip the weights between -1 and 1. This can be interpreted as each cell having a Gaussian cumulative error with $\sigma = \sigma_0$ after 1M steps. For digital weight drift, we apply independent binary random flips to the weight matrix bits every d steps with probability $p = p_0/(1M/d)$ where $p_0 = 10$. This can be interpreted as each cell flipping an average of p_0 times over 1M steps. Note that in real life, σ_0, p_0 depend on a host of issues such as the environmental conditions of the device (temperature, humidity, etc), as well as the rate of seeing training samples.

G HYPERPARAMETER SELECTION

In order to compare standard SGD with the SKS approach, we sweep the learning rates of both to optimize accuracy. In Figure 5, we compare accuracies across a range of learning rates for four different cases: SGD or SKS with or without max-norming gradients. Optimal accuracies are found when learning rate is around 0.01 for all cases. For most experiments, 8b weights, activations, and gradients, and 16b biases are used. Experiments similar to those in Section H are used to select some of the hyperparameters related to the SKS method in particular. In most experiments, rank-4 SKS

with batch sizes of 10 (for convolution layers) or 100 (for fully-connected layers) are used. Additional details can be found in the supplemental code.

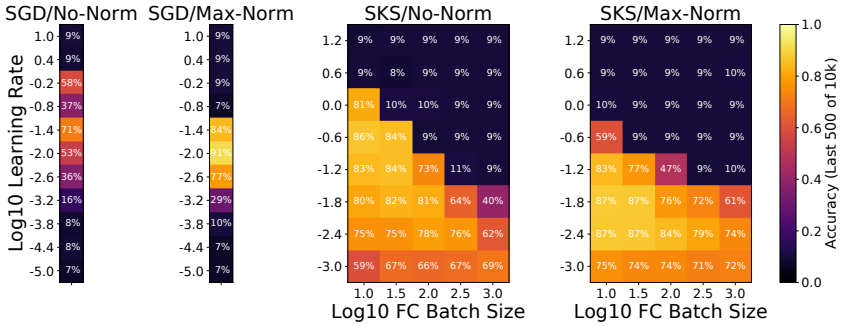


Figure 5: The left two heat maps are used to select the base / standard SGD learning rate. The right two heat maps are used to select the SKS learning rate using the optimal SGD learning rate for bias training from the previous sweeps. For the SKS sweeps, the learning rate is scaled proportional to the square-root of the batch size B. This results in an approximately constant optimal learning rate across batch size, especially for the max-norm case. Accuracy is reported averaged over the last 500 samples from a 10k portion of the online training set, trained from scratch.

H Additional Studies

In Figure 6, rank and weight bitwidth is swept for SKS with gradient max-norming. As expected, training accuracy improves with both higher SKS rank and bitwidth. In dense NVM applications, higher bitwidths may be achievable, allowing for corresponding reductions in the SKS rank and therefore, reductions in the auxiliary memory requirements.

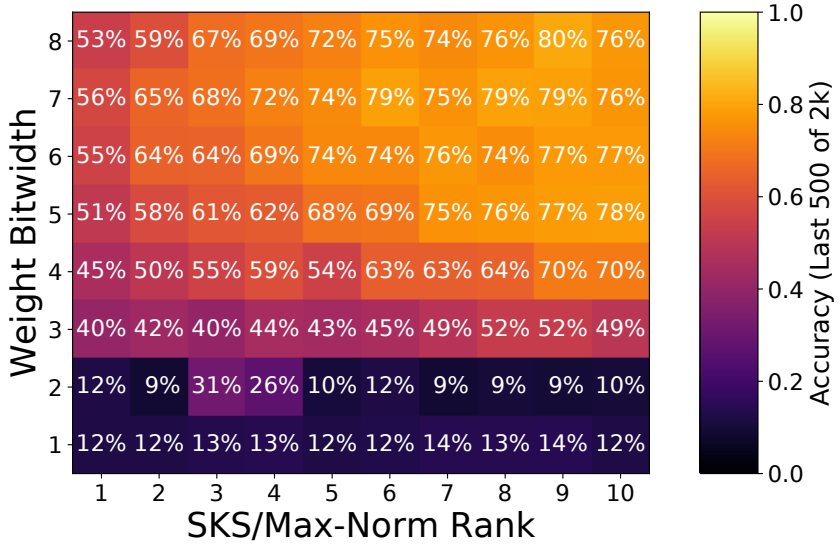


Figure 6: Accuracy across a variety of SKS ranks and weight bitwidths, showing the expected trends of increasing accuracy with rank and bitwidth. Accuracy is calculated by averaging the accuracy on the last 500 samples from a 2k portion of the training data. For bitwidths of 1 and 2, mid-rise quantization is used (e.g., 1 bit quantizes values to -0.5 and 0.5 instead of -1 and 0).

In Table 1, biased (zero-variance) and unbiased (low-variance) versions of SKS are compared. Accuracy improvements are generally seen moving from biased to unbiased SKS although the pattern differs between the no-norm and max-norm cases. In the no-norm case, a significant improvement is

seen favoring unbiased SKS for fully-connected layers. In the max-norm case, the choice of biased or unbiased SKS has only a minor impact on accuracy. It might be expected that as the number of accumulated samples for a given pseduobatch increases, lower variance would be increasingly important at the expense of bias. For our network, this implies convolutions, which receive updates at every pixel of an output feature map, would preferentially have biased SKS, while the fully-connected layer would preferentially be unbiased. This hypothesis is supported by the no-norm experiments, but not by the max-norm experiments.

Table 1: Importance of unbiased SVD. Accuracy is calculated from the last 500 samples of 10k samples trained from scratch. Mean and unbiased standard deviation are calculated from five runs of different random seeds.

Conv SKS	**FC SKS**	**Accuracy (no-norm)**	**Accuracy (max-norm)**
Biased	Biased	$79.7\% \pm 1.1\%$	$82.7\% \pm 1.3\%$
Biased	Unbiased	$83.0\% \pm 0.9\%$	$82.4\% \pm 1.2\%$
Unbiased	Biased	$77.7\% \pm 1.5\%$	$84.6\% \pm 2.0\%$
Unbiased	Unbiased	$81.0\% \pm 0.9\%$	$83.6\% \pm 2.5\%$

In Table 2, several ablations are performed on SKS with max-norm. Most notably, weight training is found to be extremely important for accuracy as bias-only training shows a $\approx 15 - 30\%$ accuracy hit depending on whether max-norming is used. Streaming batch norm is also found to be quite helpful, especially in the no-norm case.

Now, we explain the κ_{th} ablation. In Section 4.4.1, we found the SVD of a small matrix C and its singular values $\sigma_1, \ldots, \sigma_q$. This allows us to easily find the condition number of C as $\kappa(C) = \sigma_1/\sigma_q$. We suspect high condition numbers provide relatively useless update information akin to noise, especially in the presence of L, R quantization. Therefore, we prefer not to update L, R on samples whose condition number exceeds threshold κ_{th}. We can avoid performing an actual SVD (saving computation) by noting that C is often nearly diagonal, leading to the approximation $\kappa(C) \approx C_{1,1}/C_{q,q}$. Empirically, this rough heuristic works well to reduce computation load while having minor impact on accuracy. In Table 2, $\kappa_{th} = 10^8$ does not appear to ubiquitously improve on the default $\kappa_{th} = 100$, despite being $\approx 2\times$ slower to compute.

Table 2: Miscellaneous selected ablations. Accuracy is calculated from the last 500 samples of 10k samples trained from scratch. Mean and unbiased standard deviation are calculated from five runs of different random seeds.

Modified Condition	**Accuracy (no-norm)**	**Accuracy (max-norm)**
baseline (no modifications)	$80.2\% \pm 1.0\%$	$83.0\% \pm 1.1\%$
bias-only training	$51.8\% \pm 3.2\%$	$68.6\% \pm 1.4\%$
no streaming batch norm	$68.2\% \pm 1.9\%$	$81.8\% \pm 1.3\%$
no bias training	$81.3\% \pm 1.0\%$	$83.0\% \pm 1.4\%$
$\kappa_{\text{th}} = 10^8$ instead of 100	$79.8\% \pm 1.4\%$	$84.2\% \pm 1.4\%$