Compressing Gradient Optimizers via Count-Sketches

Ryan Spring [1] **Anastasios Kyrillidis** [1] **Vijai Mohan** [2] **Anshumali Shrivastava** [1]

Abstract

Many popular first-order optimization methods (e.g., Momentum, AdaGrad, Adam) accelerate the convergence rate of deep learning models. However, these algorithms require auxiliary variables, which cost additional memory proportional to the number of parameters in the model. The problem is becoming more severe as deep learning models continue to grow larger in order to learn from complex, large-scale datasets. Our proposed solution is to maintain a linear sketch to compress the auxiliary variables. We demonstrate that our technique has the same performance as the full-sized baseline, while using significantly less space for the auxiliary variables. Theoretically, we prove that count-sketch optimization maintains the SGD convergence rate, while gracefully reducing memory usage for large-models. On the large-scale 1-Billion Word dataset, we save 25% of the memory used during training (8.6 GB instead of 11.7 GB) with minimal accuracy and performance loss. For an Amazon extreme classification task with over 49.5 million classes, we also reduce the training time by 38%, by increasing the mini-batch size 3.5× using our count-sketch optimizer.

1. Introduction

An emerging trend in natural language processing is to train a language model in an unsupervised fashion on a large text corpus, and then to fine-tune the model for a specific task (Radford et al., 2018; Puri et al., 2018; Devlin et al., 2018). The language model often takes the form of an LSTM (Jozefowicz et al., 2016) or a Transformer (Vaswani et al., 2017). These models already contain millions of parameters and will continue to grow even larger to achieve better performance.

[1]Department of Computer Science, Rice University, Houston, TX, USA [2]Amazon Search, Palo Alto, CA, USA. Correspondence to: Ryan Spring <rdspring1@rice.edu>.

Proceedings of the 36^{th} International Conference on Machine Learning, Long Beach, California, PMLR 97, 2019.

Training large-scale models efficiently is a challenging task. There are numerous publications that describe how to leverage multi-GPU data parallelism and mixed precision training effectively (Hoffer et al., 2017; Ott et al., 2018; Micikevicius et al., 2018). A key tool for improving training time is to increase the batch size, taking advantage of the massive parallelism provided by GPUs. However, increasing the batch size also requires significant amounts of memory. A practitioner will sometimes sacrifice their batch size for a larger, more expressive model. For example, (Puri et al., 2018) showed that doubling the dimensionality of a multiplicative LSTM (Krause et al., 2016) from 4096 to 8192 forced them to reduce the batch size per GPU by $4\times$.

One culprit that aggravates the memory capacity issue is the auxiliary parameters used by first-order optimization algorithms. Our proposed solution is to compress the auxiliary parameters of the optimizer using the count-sketch data structure (Charikar et al., 2002), freeing up memory for either a more expressive model or a larger batch size for faster training.

The count-sketch data structure is ideally suited for compressing the auxiliary variables. We can easily tune the capacity of the count-sketch to maintain the optimizer's performance without increasing the cost of updating or querying the structure. In Section 4, we formally prove this graceful memory trade-off by analyzing the convergence rate of our count-sketch optimizer. Furthermore, the computational cost associated with the count-sketch scales with gradient sparsity. We can insert sparse gradients directly into the count-sketch, and then retrieve the corresponding approximation without accessing the entire auxiliary variable.

Leveraging gradient sparsity is useful for saving additional memory during training. Consider the language modeling task where the Embedding and Softmax layers contain a significant portion of the model's parameters and the set of active features or classes is extremely sparse. Sparsity naturally occurs in the Embedding layer because only a few words out of a large vocabulary are present in each sentence. Moreover, several algorithms impose sparsity on the Softmax layer to improve training time significantly. For example, (Shrivastava & Li, 2014; Vijayanarasimhan et al., 2014; Spring & Shrivastava, 2017; Yen et al., 2018a) have proposed using approximate nearest-neighbor search to find

the most relevant output classes. Sampled Softmax (Jean et al., 2014) is used frequently because the distribution of words follows a power-law distribution.

On the 1-Billion Word (LM1B) dataset, we trained an LSTM language model using the Adam optimizer, leveraging our count-sketch technique. By compressing the auxiliary variables, we reduced the memory usage during training by **25**% without any accuracy or performance penalty. For an Amazon extreme classification task with over **49.5** million classes, we reduced the training time by **38%** by increasing the mini-batch size **3.5**× using our count-sketch optimizer.

2. Count-Sketch and Streaming Setting

Algorithm 1 Count-Sketch Tensor

v universal hash functions h_j, random sign functions s_j
Initialize count-sketch tensor $\mathbf{S} \in \mathbb{R}^{v,w,d} = 0$

UPDATE(Count-Sketch S, item i, update $\Delta \in \mathbb{R}^d$):
Update component i with update Δ
for $j = 1$ **to** v **do**
  $\mathbf{S}_{j,h_j(i),:} \leftarrow \mathbf{S}_{j,h_j(i),:} + s_j(i) \cdot \Delta$
end for

QUERY(Count-Sketch S, item i, Function F):
Query sketch for an estimate for item i
$F \leftarrow$ MIN for non-negative values; otherwise MEDIAN
$F_{j \in \{1,2,\ldots,v\}}(s_j(i) \cdot \mathbf{S}_{j,h_j(i),:})$

In the traditional streaming setting, we are given a high-dimensional vector $\boldsymbol{x} \in \mathbb{R}^p$ that is too costly to store in memory. We only see a very long sequence of updates over time. The only information available at time t is of the form (i, Δ), which means that coordinate i is updated by the amount Δ. We are given a limited amount of storage, so we cannot store the entire vector. Sketching algorithms aim to estimate the value of current item i, after any number of updates using only $\mathcal{O}(\log p)$ memory.

The count-sketch is a popular data structure for this streaming setting. The algorithm uses a matrix $\boldsymbol{S}$ of size $v \times w \sim \mathcal{O}(\log p)$, where v and w are chosen based on the desired accuracy guarantees. For each row j, the algorithm uses v random hash functions h_j to map the vector's components to w different bins, $h_j : \{1,\ 2, ...,\ p\} \rightarrow \{1,\ 2, ...,\ w\}$. In addition, count-sketch uses v random sign functions s_j to map the components of the vectors randomly to $\{+1,\ -1\}$, $s_j : \{1,\ 2, ...,\ p\} \rightarrow \{+1, -1\}$. The count-sketch supports two operations: UPDATE(item i, update Δ) and QUERY(item i). For any update Δ to an item i, the UPDATE operation adds $s_j(i) \cdot \Delta$ to each bin $\mathcal{S}_{j,h_j(i)}, \forall j \in \{1,\ 2, ...,\ v\}$. The QUERY operation returns the median of all the bins associated with item i. If the updates are strictly non-negative, we return the minimum value.

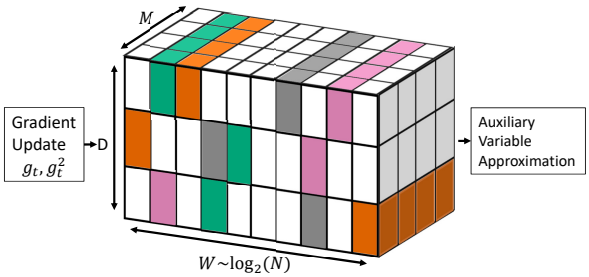

Figure 1. Visualization of Count Sketch Tensor. Each color represents a unique feature. For each row, each feature is mapped randomly to a different vector. Each vector is read from and written to memory in contiguous chunks. Preserving the last dimension of the auxiliary variable keeps structured sparsity in the count-sketch data structure, which is necessary for high performance.

3. Count-Sketch Optimizers

Majority of the parameters in deep neural networks are contained in the fully-connected layers (Han et al., 2015). Fortunately for the embedding and softmax layers, the set of active features or classes and their corresponding gradient updates are sparse. Our insight is to use the count-sketch data structure to accurately represent the auxiliary variables in a compressed manner. We will insert the gradient information into the count-sketch and retrieve an approximate value for the auxiliary variable whenever needed.

In the deep learning setting, the high-dimensional vector $\boldsymbol{\beta}$ is analogous to the matrices used to represent the auxiliary variables. The auxiliary variables are represented with $\mathbb{R}^{n,d}$ matrices where n is the number of features in the embedding layer, the number of classes in the softmax layer, or the number of nodes in the hidden layers. Since the dimensionality of the columns d is usually in the low thousands ($< 10K$), we represent the auxiliary variables with a count-sketch tensor $\mathbb{R}^{v,w,d}$ where $v \cdot w \ll n$. This count-sketch tensor preserves structured sparsity where values are read from memory in contiguous chunks along the last dimension of the tensor. See Fig. 1 for a visualization. This tensor structure maintains high performance with GPUs and CPU SIMD vector instructions. On the other hand, the n rows are compressed by randomly combining features and classes together.

Here is a brief overview of three popular first-order optimizers whose auxiliary variables we seek to compress: Momentum (Sutskever et al., 2013; Polyak, 1964) remembers a history of gradient updates, which smooths out random oscillations and accelerates convergence. Adaptive gradient descent algorithms alter the learning rate for each feature based on the frequency of its updates. Sparse, rare features

are given larger updates and a higher learning rates. These methods track a history of squared gradients for each feature. Adagrad (Duchi et al., 2011) divides the gradient by the square root of the cumulative squared gradient. Adam (Kingma & Ba, 2014) combines momentum and adaptive learning rates together, so it tracks an exponential average of the gradients and squared gradients.

The count-sketch data structure expects to receive a stream of updates Δ. For the Momentum and Adam optimizers, we need to transform the update operation into a form that is compatible with the count-sketch. For an auxiliary variable $\boldsymbol{X}$, the desired update operation is $\boldsymbol{X}$ $+= \Delta$. Given the appropriate update operation, we replace the addition assignment operator $+=$ for the original matrix with the Update-Query operation for the count-sketch Tensor.

For Momentum, given some gradient g_t, the update rule is $m_t = \gamma \cdot m_{t-1} + g_t \longleftrightarrow m_t \mathrel{+}= (\gamma - 1) \cdot m_{t-1} + g_t$. For an exponential moving average with the Adam optimizer, given some constant c and an update Δ, the update rule is $x_t = c \cdot x_{t-1} + (1-c) \cdot \Delta \longleftrightarrow x_t \mathrel{+}= (1-c) \cdot (\Delta - x_{t-1})$.

The count-sketch is essentially a plug and play replacement that saves memory, while retaining the speed and accuracy of the original matrix. Normally, algorithms that compress memory to save space are slower than their dense counterparts. However, the count-sketch can leverage sparsity by lazily performing updates with high efficiency. In addition, we can gracefully increase the size of the count-sketch for greater accuracy with minimal additional computational cost.

Count-Min Sketch Cleaning Heuristic: Since the Count-Min Sketch only accepts non-negative values, it always overestimates the desired value. The Count-Min Sketch is used to estimate the adaptive learning rate for the Adagrad and Adam optimizers. Therefore, an overestimate will prematurely slow the learning rate for certain elements. Our heuristic solution is to clean the sketch periodically by multiplying the tensor by a constant α where $0 \leq \alpha \leq 1$ every C iterations.

Periodic cleaning works well with the Count-Min Sketch because it provides a better estimate for the top-k elements. During training, the accumulation of updates allows for the heavy hitter estimates to emerge in the sketch (Aghazadeh et al., 2018). Due to stochastic gradient descent, there is a certain amount of noise in the gradient, so cleaning immediately after each update destroys the internal state of the sketch. Furthermore, cleaning reduces the scale of the sketch, reducing the overall noise level. If the signal to noise ratio is too high, future heavy hitter are ignored because there values are equal to the noise in the sketch.

Algorithm 2 Momentum - Count Sketch Optimizer

Initialize Count-Sketch Tensor $\mathbf{M} \in \mathbb{R}^{v,w,d} = 0$
v universal hash functions h_j
v random sign functions s_j
Decay Rate γ, Learning Rate η

MOMENTUM
(Item i, Parameter $x \in \mathbb{R}^d$, Gradient $g_t \in \mathbb{R}^d$):
$m_{t-1} \leftarrow$ Query($\mathbf{M}$, i, MEDIAN)
$\Delta_M \leftarrow (\gamma - 1) \cdot m_{t-1} + g_t$
Update($\mathbf{M}$, i, Δ_M)
$\hat{m}_t \leftarrow$ Query($\mathbf{M}$, i, MEDIAN)
$x_t = x_{t-1} - \eta_t \cdot m_t$

Algorithm 3 Adagrad - Count Sketch Optimizer

Initialize Count-Min Sketch Tensor $\mathbf{V} \in \mathbb{R}^{v,w,d} = 0$
v universal hash functions h_j
Learning Rate η

ADAGRAD
(Item i, Parameter $x \in \mathbb{R}^d$, Gradient $g_t \in \mathbb{R}^d$):
$\Delta_V \leftarrow g_t^2$
UPDATE($\mathbf{V}$, i, Δ_V)
$v_t \leftarrow$ QUERY($\mathbf{V}$, i, MIN)
$x_t = x_{t-1} - \eta_t \cdot \frac{g_t}{\sqrt{v_t} + \epsilon}$

Algorithm 4 Adam - Count Sketch Optimizer

Initialize Count-Sketch Tensor $\mathbf{M} \in \mathbb{R}^{v,w,d} = 0$
Initialize Count-Min-Sketch Tensor $\mathbf{V} \in \mathbb{R}^{v,w,d} = 0$
v universal hash functions h_j
v random sign functions s_j
1st Moment Decay Rate β_1, 2nd Moment Decay Rate β_2
Learning Rate η

ADAM
(Item i, Parameter $x \in \mathbb{R}^d$, Gradient $g_t \in \mathbb{R}^d$):
// Count-Sketch - 1st Moment
$m_{t-1} \leftarrow$ Query($\mathbf{M}$, i, MEDIAN)
$\Delta_M \leftarrow (1 - \beta_1)(g_t - m_{t-1})$
Update($\mathbf{M}$, i, Δ_M)
$m_t \leftarrow$ Query($\mathbf{M}$, i, MEDIAN)

// Count-Min Sketch - 2nd Moment
$v_{t-1} \leftarrow$ Query($\mathbf{M}$, i, MIN)
$\Delta_V \leftarrow (1 - \beta_2)(g_t^2 - v_{t-1})$
Update($\mathbf{V}$, i, Δ_V)
$v_t \leftarrow$ Query($\mathbf{M}$, i, MIN)

$\hat{m}_t = m_t / (1 - \beta_1^t)$
$\hat{v}_t = v_t / (1 - \beta_2^t)$
$x_t = x_{t-1} - \eta_t \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$

4. Theoretical Analysis

For stochastic non-convex optimization (Zaheer et al., 2018), we measure how the algorithm converges to a stationary point at iteration x_t—i.e., $\|\nabla f(x_t)\|^2 \leq c$ for some small constant c. In our analysis, we focus on the Count-Min Sketch Adam optimizer where we do not track the 1st moment—i.e., $\beta_1 = 0$.

We assume that the function f is L-smooth with bounded gradients: Function f has bounded gradients - $[f(x_t)]_i \leq G_i, \forall x \in \mathbb{R}^d, i \in [d], G = \left\|\vec{G}\right\|_\infty$. In addition, we receive an unbiased stochastic gradient estimate g_t with fixed variance σ^2. Then, the following theorem holds:

Theorem 4.1. *Let the learning rate $\eta_t = \eta, \forall t \in [T]$. Assume β_2, η, and ϵ are selected such that $\eta \leq \frac{\epsilon}{2L}$ and $\sqrt{1-\beta_2} \leq \frac{\epsilon}{4G}$. Given a Count-Min Sketch matrix with width $\Theta(\frac{1}{\epsilon_1})$ and depth $\Theta(\log(\frac{dT}{\delta}))$, we have the following bound that holds for Count-Min Sketch Adam with probability $(1-\delta)$ where $M = \left(\frac{G\sqrt{1-\beta_2}}{\epsilon^2}\sum_{i=0}^{d}\left\|\vec{G}\right\|_2^2\right)$:*

$$\min_t \mathbb{E}\|\nabla f(x_t)\|^2 \leq \mathcal{O}\Big(\frac{f(x_0) - f(x_*)}{\eta T} + \sigma^2 + \epsilon_1 M\Big)$$

The proof of Theorem 4.1 is found in the Appendix. For comparison, we have the convergence bound for the standard Adam optimizer where $\beta_1 = 0$, proven in (Zaheer et al., 2018):

$$\min_t \mathbb{E}\|\nabla f(x_t)\|^2 \leq \mathcal{O}\Big(\frac{f(x_0) - f(x_*)}{\eta T} + \sigma^2\Big)$$

Discussion: The bounds are similar except for the additional term caused by the Count-Min Sketch approximation. The theorem states that the Count-Min Sketch Adam converges to a region around a stationary point with radius $O(\sigma^2 + \epsilon_1 M)$. For each dimension $i \in [d]$, there is an error $\epsilon_1 M$ that depends on the adaptivity of the optimizer β_2, the error rate ϵ_1 of the sketch, and the gradient norm $\|G\|_2^2$. The error rate ϵ_1 is proportional to the width of the sketch $\epsilon_1 = 1/w$ and corresponds with the number of collisions along each row in the sketch. We can improve convergence gracefully by increasing the sketch's width, which reduces the error caused when multiple components collide in the same bin. When the sketch width $w = \Theta(d)$, the error term becomes a small constant. The failure probability δ of exceeding the Count-Min Sketch error bound is proportional to the depth of the sketch $\delta = dT/e^v$. In our theoretical results, the depth of the sketch depends on the number of parameters d and the number of time steps T. However, in practice, our experiments show that a modest depth size of 3-5 is sufficient.

5. Related Work

Feature Compression: A straight-forward option is to use dimensionality reduction techniques to minimize the number of features, which in turn decreases the size of the model and optimizer. (Tito Svenstrup et al., 2017) describes a hash embedding scheme where the embedding for a feature is a weighted sum between the base embedding vectors and a weight vector. Their goal was to minimize the size of the embedding layer while preserving its flexibility. However, dramatically reducing the parameter space may sacrifice model accuracy. For example, training the BERT language model (Devlin et al., 2018) on a GPU with 12-16 GB memory requires a smaller, less effective architecture than the full-sized model trained on the 64 GB Google TPU.

Gradient Checkpointing: (Siskind & Pearlmutter, 2018; Chen et al., 2016) describe an orthogonal approach where training an N-layer neural network requires $\sqrt{N}$ memory. Their insight was that storing the activations for the back-propagation pass is the most memory-intensive part of training. Instead of storing all the activations, their algorithm checkpoints certain sections of the neural network and lazily recomputes the activations during the back-propagation phase. In other words, their approach saves memory by sacrificing extra computation time.

Low-Rank Approximation: A low-rank approximation has the potential to reduce the number of parameters from $O(nd)$ to $O(nr + rd)$ where $r \ll \min(n, d)$. However, updating the low-rank matrices is non-trivial. (Shazeer & Stern, 2018) demonstrated that there exists a unique, fast update rule for a rank-1 approximation that minimizes the I-divergence between the approximation and original matrix. Their rank-1 approximation was limited to non-negative matrices, so only the second moment of the Adam optimizer was compressed in their experiments. The drawback of this approach is that it requires materializing the entire matrix via an outer-product, which is prohibitive for large-scale embedding and softmax layers. Since their update rule only applies for rank-1 vectors, their approach lacks the flexibility to increase the model's memory capacity gracefully.

Count-Sketch: The original objective of the count-sketch data structure was to estimate the frequency of various events in the streaming setting. Recently, (Aghazadeh et al., 2018; Tai et al., 2018) demonstrated that the count-sketch can learn a compressed model that accurately preserves the features with the largest weights. Their objective focused on feature extraction in ultra-high dimensional settings and was limited to simple, linear models. In this work, we seek to use the count-sketch to preserve the different auxiliary variables maintained by commonly used first-order optimizers. The ideal solution is for the memory cost of the optimizer to grow sub-linearly with the model size, giving us the flexibility to increase the model's capacity.

Table 1. Trade-offs between the Count-Sketch and Low-Rank Approximation. k is the number of active features or classes. r is the rank of the two factors where $r << min(n, d)$.

Type	Count-Sketch	Low-Rank
Memory	$\mathcal{O}(n \cdot \log d)$	$\mathcal{O}(nr + rd)$
Gradient Type	Sparse + Dense	Dense
Memory Control	Flexible	Fixed
Query Time	$\mathcal{O}(nk)$	$\mathcal{O}(nrm)$

6. Experiments

All of the experiments were performed with the PyTorch framework on a single machine - 2x Intel Xeon E5-2660 v4 processors (28 cores / 56 threads) with 512 GB of memory using a single Nvidia Tesla V100. The code[1] for the Count-Sketch Optimizer is available online.

We designed the experiments to answer these questions:
1. How accurate is our estimate of the auxiliary variables retrieved from the count-sketch data structure?
2. What the effect of cleaning the count-min sketch on convergence time and accuracy?
3. How well does our count-sketch optimizer compare against the low-rank approximation given the same number of parameters?
4. Does our count-sketch optimizer match original baseline in terms of speed and accuracy?

Here are the six datasets used in the experiments:
Wikitext-2 (Merity et al., 2016) - This dataset was extracted from Wikipedia and contains 2M training tokens with a vocabulary size of 33,278.
Wikitext-103 (Merity et al., 2016) - A larger version of the Wikitext-2 dataset that contains 103M training tokens and its vocabulary size is 267,735.
1-Billion Word (LM1B) (Chelba et al., 2013) - This large-scale corpus contains 0.8 billion training tokens and a vocabulary with 793,471 words. An open-sourced PyTorch model is available online[2].
MegaFace (Nech & Kemelmacher-Shlizerman, 2017) - A facial recognition dataset derived from MegaFace—Challenge 2. Each person is a candidate class, but we only select classes with at least 10 images. Thus, this sampled dataset contains 1,943,802 examples with 80,204 classes. 10K images are randomly sampled to create the test dataset.
ImageNet (Russakovsky et al., 2015) - An image classification dataset that contains over 1 million examples and 1000 object classes.
Amazon - This sampled recommendation dataset contains 70.3 million examples and over 49.5 million object classes. (20.9 GB)

[1] https://github.com/rdspring1/Count-Sketch-Optimizers
[2] https://github.com/rdspring1/PyTorch_GBW_LM

We utilized these baselines to compare against our approach:
Non-Negative Matrix Factorization (NMF) — This decomposition minimizes the I-divergence between the auxiliary variable and the approximation formed from two Rank-1 vectors. However, it is limited to non-negative matrices, so it cannot compress the auxiliary variables for Momentum or the 1st Moment of Adam. (Shazeer & Stern, 2018)
ℓ_2 **Rank-1** — After each update, we perform an SVD decomposition of the auxiliary variable, and only keep the top singular value and its corresponding vectors. During the subsequent update, the auxiliary variable is reconstructed via an outer product. Unlike the NMF Rank-1 Approximation, this approach is not limited to non-negative values, but it is extremely slow and cannot be used in practice.
Count-Sketch — As described in Section 3. This approach is also not limited to non-negative values and is capable of compressing the auxiliary variables for all optimizers efficiently.

Table 2. Abbreviations

Title	Symbol
Count-Sketch	CS
Count-Min-Sketch	CMS
Low-Rank	LR
Adam 1st Moment	M
Adam 2nd Moment	V
Non-Negative Matrix Factorization	NMF

6.1. Small-Scale Experiments

Wikitext-2: The language model was a 2-layer LSTM with 672 hidden units. The dimensionality of the word embeddings was equal to the number of hidden units. The model was unrolled 35 steps for the back-propagation through time (BPTT). Dropout with a 50% chance of disabling a unit was used to regularize the model. We trained the model for 40 epochs with a mini-batch size of 20. For Momentum, the learning rate was 2.5, the decay rate γ was 0.9, and we clipped the gradient norm to 0.25. For Adam, the learning rate was 0.001, the beta values β_1, β_2 were 0.9 and 0.999, and gradient clipping was 1. We reduced the learning rate by $4\times$ whenever the validation error plateaued. We used the full softmax layer, so only the embedding layer was sparse for this dataset.

ℓ_2-Norm Approximation Error: Figure. 2 shows the ℓ_2-Norm between the approximation and the original auxiliary variable. The left figure is for the Momentum optimizer, and the right figure is for the 2nd Moment for the Adam optimizer. All of the methods were given roughly an equal amount of parameters to approximate the original auxiliary variable. For the Wikitext-2 dataset, the embedding and softmax layers were [33,278, 256] matrices. Therefore, the rank-1 decomposition used two vectors that use 33,278 +

Table 3. Test Perplexity for Momentum Optimizer on the Wikitext-2 dataset.

Momentum	CS	LR-NMF
94.25	95.93	176.31

Table 4. Test Perplexity for Adam Optimizer on the Wikitext-2 dataset. The modifiers indicate the compressed auxiliary variables.

Adam	CS-MV	CS-V	LR-NMF-V
105.14	109.24	106.32	106.21

256 = 33,534 parameters. The count-sketch data structure was represented with a [3, 16, 672] tensor, containing 32,256 parameters. Our count-sketch approach mapped the 33,278 word vocabulary into 16 distinct bins, so there were about 2,080 collisions for each bucket.

The Adam optimizer's 2nd Moment is strictly non-negative and is suitable for the NMF Rank-1 approximation. For the Momentum variable, we supplement the NMF decomposition with the ℓ_2 SVD decomposition. The ℓ_2 SVD decomposition maintains a good approximation of the Momentum variable. However, it is extremely slow during training, so we only show the approximation error for the first epoch of training. As expected, the NMF Rank-1 baseline poorly approximated the momentum variable because it was not strictly non-negative. It experienced significant variance in its approximation quality. The count-sketch was a consistent estimator for both variables with slightly more error for both variables.

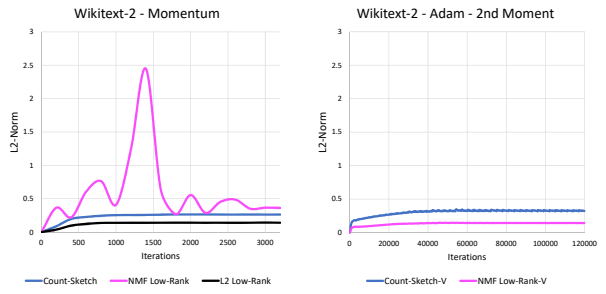

Figure 2. Left - Momentum, Right - Adam-2nd Moment. ℓ_2-Norm between the approximation and the original auxiliary variable.

Test Perplexity: Tables 3,4 show the test perplexity after training the model with the Momentum and Adam optimizers. For the Momentum optimizer, the NMF Low-Rank approximation performed poorly, reinforcing the results from Fig. 2. When only the 2nd moment was compressed, the NMF Low-Rank and Count-Sketch approximations have negligible differences. When we compressed both the 1st and 2nd moments with the count-sketch, there was some minor accuracy loss from the original optimizer.

MegaFace: For this experiment, we used pretrained embeddings from the FaceNet architecture (Schroff et al., 2015) trained on the MS-Celeb-1M dataset[3]. Afterwards, we trained a softmax classifier on the MegaFace dataset using LSH sampling (Yen et al., 2018b; Vijayanarasimhan et al., 2014). Our LSH sampling distribution was SimHash with K=15 bits per hash fingerprint and L=16 hash tables, rebuilt every 250 iterations. For Adam, the learning rate was 0.001 and the beta values β_1, β_2 were (0.9, 0.999). For Adagrad, the learning rate was 0.1. All the models were trained for 10 epochs.

Figure. 3 shows the effect of cleaning the Count-Min Sketch Tensor on its corresponding optimizer. We measured how the test accuracy, convergence rate, and auxiliary variable error changed because of cleaning. For Adam, the cleaning scheme was to multiply the count-min sketch by a constant 0.2 every 125 iterations. For Adagrad, the rate of cleaning was the same, but the constant was changed to 0.5.

For both Adam and Adagrad, there was a noticeable drop in ℓ_2-Norm error with the cleaning heuristic. For Adam, the count-sketch optimizer with cleaning closely matched the convergence rate of the baseline. The test accuracy for count-sketch with cleaning was 69.4%, while the baseline was 69.03%. Cleaning did not improve the initial convergence rate for Adagrad, but it allowed the final test accuracy to match the baseline.

Given that the Adam optimizer already contains an exponential decay term, it is surprising that cleaning is necessary. However, despite requiring more hyper-parameter tuning, the count-sketch optimizer with cleaning still achieves the best performance. One potential explanation is that since the gradients were sparse, only the non-zero elements were updated. Thus, the decay was applied in an irregular fashion for the elements in the sketch.

6.2. Image Classification

We trained a ResNet-18 architecture on the ImageNet dataset for 90 epochs with a batch size of 256. The baseline optimizer was RMSprop with a learning rate of 0.01. For the Count-Sketch optimizer, we used 20% of the original parameters for the auxiliary variables ($5\times$ fewer).

Table 5 shows that the Count-Sketch RMSprop optimizer has a negligible effect on training time and test accuracy. Our results are comparable with SGD, the gold standard optimizer for training convolutional networks. Majority of the memory usage comes from the gradients, so gradient checkpointing is more applicable for this architecture. However, we saved 100 MB compared to the RMSprop optimizer, despite the small model size (50 MB).

[3] https://github.com/davidsandberg/facenet

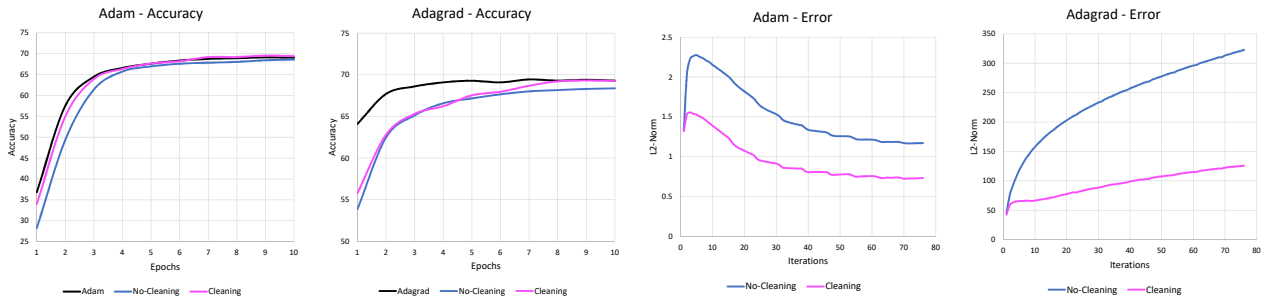

Figure 3. The effect of cleaning on the Count-Min Sketch Tensor and its corresponding optimizer for the MegaFace dataset.

To compress dense layers effectively without gradient sparsity, we avoid generating the auxiliary variables explicitly in global memory. Instead, we access them inside the shared memory of the GPU. During training, we update the auxiliary variables and parameters in a single CUDA kernel. Fusing the two separate operations together has the side benefit of improving memory locality and reducing the performance penalty from the randomized hash function.

Table 5. Running Time per Epoch, Memory, and Test Accuracy using the RMSprop optimizer on the ImageNet dataset.

Metric	SGD	CMS	RMSprop
Time (Hrs)	0.64	0.60	0.61
Size (MB)	9,257	9,495	9,603
Test Accuracy@1	69.76	69.12	65.27
Test Accuracy@5	89.08	88.87	86.09

6.3. Large-Scale Language Model

Since these datasets have large vocabularies, we used Sampled Softmax (Jean et al., 2014) for faster training and to induce sparsity in the Softmax layer. Each Count-Sketch tensor was $5\times$ smaller than the original variable.

Adagrad - Wikitext-103: Our language model was a single layer LSTM with 1024 hidden units. The dimensionality of the word embeddings was 256 and we used a projection layer between the LSTM and Softmax layers. The model was unrolled 35 steps BPTT. The model was regularized via Dropout with $p = 0.25$. We trained the model for 25 epochs with a mini-batch size of 1024. For the Adagrad optimizer, the gradient norm was clipped to 0.1 and the learning rate decayed linearly from 0.4 to 0 during training.

Results: By providing the count-sketch with more parameters, our method has notably better test accuracy than the NMF low-rank approximation while using only slightly more memory. In addition, despite using more parameters than the low-rank approximation, the count-sketch optimizer is still somewhat faster. Finally, the low-rank approximation fails to meet the same accuracy as the original baseline, while surprisingly the count-sketch optimizer has the best test perplexity.

Table 6. Running Time per Epoch, Memory, and Test Perplexity on the Wikitext-103 dataset using the Adagrad Optimizer.

Metric	Adagrad	CS	LR-NMF
Time (Hrs)	**6.4**	6.6	6.7
Size (MB)	10,625	10,089	**10,077**
Test Perplexity	57.63	**56.07**	58.27

RMSprop - LM1B - Transformer: Our language model was an 18-layer Transformer-XL model (Dai et al., 2019). The code is available online[4]. Each layer had 8 attention heads with 2048 hidden units. The dimensionality of the word embeddings was 256. The gradient norm was clipped to 0.25. Using the cosine annealing schedule, the learning rate decayed from 0.00025 to 0. We trained the model for 5 epochs with a batch size of 224. Instead of Sampled Softmax, we used Adaptive Softmax (Grave et al., 2017) to improve training speed and to reduce the memory costs. The overall model contained about 70 million parameters (283 MB). Our results show that our count-sketch RMSprop saves the most memory, while maintaining the same convergence rate, test perplexity, and running time.

Table 7. Running Time per Epoch, Memory, and Test Perplexity on the LM1B dataset using the RMSprop Optimizer.

Metric	RMSprop	CMS	LR-NMF
Time (Hrs)	0.47	0.48	0.5
Size (MB)	10,625	10,089	5,169
Test Perplexity	38.83	39.36	38.83

Adam - LM1B - LSTM: Our goal was to mimic multi-GPU distributed training on a single GPU. The original batch size was 128 with a learning rate of 5e-4. After increasing our batch size from 128 to 1024, we scaled our learning rate linearly by $8\times$ (Goyal et al., 2017). In addition, we decayed our learning rate linearly to zero over 5 training epochs. We doubled the LSTM size from 1024 to 2048, but kept the word embedding size at 256. The model was unrolled 20 steps BPTT. Dropout was kept nominally at $p = 0.01$ and the gradient norm was clipped to 1. A surprising side effect of increasing the batch size was that we reduced our training time by roughly $2\times$ from 12.25 hours to 6.25 hours per epoch despite using a single GPU.

[4] https://github.com/kimiyoung/transformer-xl

Results: Our primary comparison was only with the 2nd moment because the NMF low-rank approximation is not applicable to the 1st moment. The count-sketch was slightly more accurate than the low-rank approximation. When both the 1st and 2nd moments were compressed with the count-sketch tensor, its accuracy was on-par with the low-rank approximation that compressed only the 2nd moment. The count-sketch tensor was 8% faster than the low-rank approach while using substantially less GPU memory. For large matrices, there was a noticeable cost with reconstructing the entire matrix to update only a sparse subset of values.

Table 8. Running Time per Epoch (Hours) and Memory Consumption (MB) on the 1-Billion Word dataset for the Adam optimizer.

Metric	Adam	CS-MV	CS-V	LR-NMF-V
Time	**5.28**	5.42	5.35	5.84
Size	11,707	**8,591**	10,167	13,259

Table 9. Convergence Rate (Test Perplexity) after 5 epochs on the 1-Billion Word dataset. The modifiers indicate which auxiliary variables are compressed for the Adam optimizer.

Epoch	CS-MV	Adam	CS-V	LR-NMF-V
1	50.78	48.48	49.49	50.04
2	46.08	45.34	45.22	45.60
3	43.71	42.79	42.95	43.55
4	41.82	41.15	41.23	41.82
5	40.55	39.90	**39.88**	40.41

6.4. Extreme Classification

For the extremely large-scale classification task, we conducted our experiments on an Amazon recommendation dataset. The task was to predict an object out of over 49 million classes. The text query was parsed into trigram features. Feature hashing was applied to convert the strings into integers. The input feature dimension was 80K. On average, there were only 30 non-zero features per query, so the input layer was very sparse and suitable for our count-sketch optimizer. We trained a single hidden layer, fully-connected neural network with an embedding dimension of 1024.

A traditional softmax classifier would require over 200 GB of memory, which is well beyond the memory capacity of the largest GPUs. Instead, we leveraged a novel approach for extreme classification called Merged-Averaged Classifiers via Hashing (MACH) (Huang et al., 2018). This algorithm randomly merges the output classes into a manageable number of coarse-grained, meta-classes via universal hashing. Several independent, fully-connected neural networks are trained to solve this meta-class classification task. Each meta-classifier is associated with a unique hash function that creates a distinct class mapping. At inference time, we recover the scores for the original classes by aggregating the meta-class scores assigned to the original output class. For this experiment, we used 20K meta-classes in the output layer of each meta-classifier. For high-accuracy models, we used 32 meta-classifiers. Each individual meta-classifier required 414 MB of memory for a total of 12.95 GB. Therefore, our ensemble MACH classifier used $15\times$ less memory than a monolithic softmax classifier.

Since we are primarily interested in achieving faster training times, we limited ourselves to 4 meta-classifiers in this experiment. For our baseline, each meta-classifier was trained using the Adam optimizer with a batch size of 750. Given these settings, a single meta-classifier takes 4 GB of GPU memory, allowing us to train 4 models in parallel on a single GPU. For maximum memory savings, we eliminated the 1st moment and used a count-min sketch tensor for the 2nd moment **(1% of original size)**. By using the Adam Count-Sketch optimizer, we reduced the memory cost for each model from 4 GB to 2.6 GB **(45% smaller)**. We took advantage of this extra memory by increasing the batch size from 750 to 2600 **(3.5$\times$ larger)**. As a result, the running time per epoch decreased from 5.32 hours to 3.3 hours **(38% faster)**.

We measured the accuracy of the MACH model using the Recall@100 metric on a test dataset containing 20K queries. First, we evaluated the meta-classifiers and then aggregated their scores. Then, we checked how often the target class appears within the top 100 scores generated by the classifier. A major bottleneck during evaluation was sorting the 49.5 million classes to find the top 100 scores. Since we were only comparing the model's relative performance, we down-sampled the scores from 49.5 million to 1 million. The class subset contained the target classes for all 20K test queries and a random sample of the remaining classes. Given 16 meta-classifiers, the Adam baseline had a 0.6881 recall, while the count-sketch optimizer achieved a 0.6889 recall.

Table 10. Extreme Classification — A MACH ensemble with 4 meta-classifiers trained on a single GPU using Adam.

Type	Batch Size	Time (Hrs)	Recall@100
Adam	750	5.32	0.4704
CS-V	**2600**	**3.3**	**0.4789**

7. Conclusion and Future Work

In this paper, we presented the concept of a count-sketch tensor to compress the auxiliary variables associated with popular first-order optimizers. The count-sketch tensor retains the constant-time update and query operations, while maintaining the tensor structure for high-speed vectorized operations. The count-sketch tensor can reduce the memory usage of large-scale models with minimal cost and take advantage of the model's sparsity. Going forward, we are interested in leveraging recent ideas of adding sparsity to the hidden layers in order to increase the size of the model further without increasing its computational cost (Spring & Shrivastava, 2017; Shazeer et al., 2017; Wen et al., 2017).

Acknowledgements

TODO: Thanks to XXX and YYY for insightful discussions and making coffee.

References

Aghazadeh, A., Spring, R., Lejeune, D., Dasarathy, G., Shrivastava, A., and Baraniuk, R. MISSION: Ultra Large-Scale Feature Selection using Count-Sketches. In Dy, J. and Krause, A. (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 80–88, Stockholmsmssan, Stockholm Sweden, 10–15 Jul 2018. PMLR. URL http://proceedings.mlr.press/v80/aghazadeh18a.html.

Charikar, M., Chen, K., and Farach-Colton, M. Finding frequent items in data streams. In *Intl. Colloquium on Automata, Languages, and Programming*, pp. 693–703. Springer, 2002.

Chelba, C., Mikolov, T., Schuster, M., Ge, Q., Brants, T., Koehn, P., and Robinson, T. One billion word benchmark for measuring progress in statistical language modeling. *arXiv preprint arXiv:1312.3005*, 2013.

Chen, T., Xu, B., Zhang, C., and Guestrin, C. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

Dai, Z., Yang, Z., Yang, Y., Cohen, W. W., Carbonell, J., Le, Q. V., and Salakhutdinov, R. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Duchi, J., Hazan, E., and Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, large minibatch sgd: training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

Grave, E., Joulin, A., Cissé, M., Jégou, H., et al. Efficient softmax approximation for gpus. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 1302–1310. JMLR. org, 2017.

Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

Hoffer, E., Hubara, I., and Soudry, D. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 30*, pp. 1731–1741. Curran Associates, Inc., 2017.

Huang, Q., Wang, Y., Medini, T., and Shrivastava, A. Extreme classification in log memory. *arXiv preprint arXiv:1810.04254*, 2018.

Jean, S., Cho, K., Memisevic, R., and Bengio, Y. On using very large target vocabulary for neural machine translation. *arXiv preprint arXiv:1412.2007*, 2014.

Jozefowicz, R., Vinyals, O., Schuster, M., Shazeer, N., and Wu, Y. Exploring the limits of language modeling, 2016. URL https://arxiv.org/pdf/1602.02410.pdf.

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

Krause, B., Lu, L., Murray, I., and Renals, S. Multiplicative lstm for sequence modelling. *arXiv preprint arXiv:1609.07959*, 2016.

Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., and Wu, H. Mixed precision training. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=r1gs9JgRZ.

Nech, A. and Kemelmacher-Shlizerman, I. Level playing field for million scale face recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

Ott, M., Edunov, S., Grangier, D., and Auli, M. Scaling neural machine translation. *arXiv preprint arXiv:1806.00187*, 2018.

Polyak, B. T. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

Puri, R., Kirby, R., Yakovenko, N., and Catanzaro, B. Large scale language modeling: Converging on 40gb of text in four hours. *arXiv preprint arXiv:1808.01371*, 2018.

Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. Improving language understanding by generative pre-training. *Online*, 2018. URL https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/languageunsupervised/languageunderstandingpaper.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.

Schroff, F., Kalenichenko, D., and Philbin, J. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 815–823, 2015.

Shazeer, N. and Stern, M. Adafactor: Adaptive learning rates with sublinear memory cost. In Dy, J. and Krause, A. (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 4596–4604, Stockholmsmssan, Stockholm Sweden, 10–15 Jul 2018. PMLR. URL http://proceedings.mlr.press/v80/shazeer18a.html.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., and Dean, J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

Shrivastava, A. and Li, P. Asymmetric lsh (alsh) for sublinear time maximum inner product search (mips). In *Advances in Neural Information Processing Systems*, pp. 2321–2329, 2014.

Siskind, J. M. and Pearlmutter, B. A. Divide-and-conquer checkpointing for arbitrary programs with no user annotation. *Optimization Methods and Software*, 33(4-6): 1288–1330, 2018.

Spring, R. and Shrivastava, A. Scalable and sustainable deep learning via randomized hashing. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 445–454. ACM, 2017.

Sutskever, I., Martens, J., Dahl, G., and Hinton, G. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pp. 1139–1147, 2013.

Tai, K. S., Sharan, V., Bailis, P., and Valiant, G. Sketching linear classifiers over data streams. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pp. 757–772, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-4703-7. doi: 10.1145/3183713.3196930. URL http://doi.acm.org/10.1145/3183713.3196930.

Tito Svenstrup, D., Hansen, J., and Winther, O. Hash embeddings for efficient word representations. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 30*, pp. 4928–4936. Curran Associates, Inc., 2017.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. Attention is all you need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 30*, pp. 5998–6008. Curran Associates, Inc., 2017. URL http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf.

Vijayanarasimhan, S., Shlens, J., Monga, R., and Yagnik, J. Deep networks with large output spaces. *arXiv preprint arXiv:1412.7479*, 2014.

Wen, W., He, Y., Rajbhandari, S., Zhang, M., Wang, W., Liu, F., Hu, B., Chen, Y., and Li, H. Learning intrinsic sparse structures within long short-term memory. *arXiv preprint arXiv:1709.05027*, 2017.

Yen, I. E.-H., Kale, S., Yu, F., Holtmann-Rice, D., Kumar, S., and Ravikumar, P. Loss decomposition for fast learning in large output spaces. In Dy, J. and Krause, A. (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 5640–5649, Stockholmsmssan, Stockholm Sweden, 10–15 Jul 2018a. PMLR. URL http://proceedings.mlr.press/v80/yen18a.html.

Yen, I. E.-H., Kale, S., Yu, F., Holtmann-Rice, D., Kumar, S., and Ravikumar, P. Loss decomposition for fast learning in large output spaces. In Dy, J. and Krause, A. (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 5640–5649, Stockholmsmssan, Stockholm Sweden, 10–15 Jul 2018b. PMLR.

Zaheer, M., Reddi, S., Sachan, D., Kale, S., and Kumar, S. Adaptive methods for nonconvex optimization. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 31*, pp. 9815–9825. Curran Associates, Inc., 2018. URL http://papers.nips.cc/paper/8186-adaptive-methods-for-nonconvex-optimization.pdf.