A. Quantizing Lookup Tables

Since the lookup table entries naturally occupy more than 8 bits even for 8-bit data (since products of 8-bit values require 16 bits), some means of quantizing these entries is necessary to enable vectorization. Unfortunately, existing quantization methods are not applicable to our problem setting. The scheme of Blalock & Guttag (2017) requires knowledge of $\boldsymbol{B}$ at training time, while the scheme of André et al. (2017) and André et al. (2019) is only applicable for nearest-neighbor search. We instead use the following approach, where $\boldsymbol{T} \in \mathbb{R}^{M \times C \times K}$ is the tensor of lookup tables for all M columns of $\boldsymbol{B}$, $\boldsymbol{T}^q$ is the quantized version of $\boldsymbol{T}$, $\boldsymbol{\delta} \in \mathbb{R}^C$ is a vector of table-specific offsets, and α^{-1} is an overall scale factor:

$$\boldsymbol{\delta}_c \triangleq \min_{m,k} \boldsymbol{T}_{m,c,k} \tag{12}$$

$$\alpha^{-1} \triangleq 2^l, l = \max_c \left\lfloor \log_2 \left(\frac{255}{\max_{m,k}(\boldsymbol{T}_{m,c,k} - \delta_c)} \right) \right\rfloor \tag{13}$$

$$\boldsymbol{T}^q_{m,c,k} \triangleq \alpha^{-1}(\boldsymbol{T}_{m,c,k} - \delta_c). \tag{14}$$

This is similar to equations 15 and 16, but with the scale factor pooled across all codebooks instead of unique to each input column. The α used here is the same as that in equation 2, and the matrix β in equation 2 has entries equal to $\sum_c \boldsymbol{\delta}_c$ (plus the debiasing constant from our averaging-based aggregation).

B. Quantization and MADDNESSHASH

The use of at most 16 leaves is so that the resulting codes use 4 bits. This allows the use of these same shuffle instructions to accelerate the table lookups as in Blalock & Guttag (2017).

The only subtlety in vectorizing our hash function is that one must execute line 4 using shuffle instructions such as `vpshufb` on x86, `vtbl` on ARM, or `vtbl` on PowerPC. In order to do this, the split values and scalars x_{j^t} must be 8-bit integers. We quantize them by learning for each split index j a pair of scalars (γ_j, δ_j), where

$$\delta_j \triangleq \min_i \boldsymbol{v}_i^j \tag{15}$$

$$\gamma_j \triangleq 2^l, l = \left\lfloor \log_2 \left(\frac{255}{\max_i \boldsymbol{v}_i^j - \delta_j} \right) \right\rfloor \tag{16}$$

This restriction of γ_j to powers of two allows one to quantize x_{j^t} values with only shifts instead of multiplies. The $\boldsymbol{v}$ values can be quantized at the end of the training phase, while the x_{j^t} values must be quantized within Algorithm 1 before line 5.

C. Subroutines for Training MADDNESSHASH

The `optimal_split_threshold` algorithm (Algorithm 3) finds the best threshold at which to split a bucket within a given dimension. To do this, it uses the `cumulative_sse` function (Algorithm 4) to help evaluate the loss associated with the resulting child buckets.

These algorithms exploit the fact that the sum of squared errors can be computed using only the sum of values and sum of squared values, both of which can be updated in $O(1)$ time when a vector is moved from one side of the split to the other.

Algorithm 3 Optimal Split Threshold Within a Bucket

```
1: Input: bucket $\mathcal{B}$, index $j$
2: $\boldsymbol{X} \leftarrow$ as_2d_array($\mathcal{B}$)
3: $\boldsymbol{X}^{sort}$ = sort_rows_based_on_col($\boldsymbol{X}, j$)
4: sses_head $\leftarrow$ cumulative_sse($\boldsymbol{X}^{sort}$, false)
5: sses_tail $\leftarrow$ cumulative_sse($\boldsymbol{X}^{sort}$, true)
6: losses $\leftarrow$ sses_head
7: losses$_{1:N-1}$ $\leftarrow$ losses$_{1:N-1}$ + sses_tail$_{2:N}$
8: $n^* \leftarrow \mathrm{argmin}_n$ losses$_n$
9: return $(\boldsymbol{X}^{sort}_{n^*,j} + \boldsymbol{X}^{sort}_{n^*+1,j})/2$, losses$_{n^*}$
```

Algorithm 4 Cumulative SSE

```
1: Input: 2D array $\boldsymbol{X}$, boolean reverse
2: $N, D \leftarrow$ shape($\boldsymbol{X}$)
3: if reverse then
4:     $\forall_i$ swap($\boldsymbol{X}_{i,d}, \boldsymbol{X}_{N-i+1,d}$)
5: end if
6: out $\leftarrow$ empty($N$)
7: cumX $\leftarrow$ empty($D$)
8: cumX2 $\leftarrow$ empty($D$)
   // Initialize first row of output and cumulative values
9: out$_1$ $\leftarrow 0$
10: for $d \leftarrow 1$ to $D$ do
11:     cumX$_d$ $\leftarrow X_{1,d}$
12:     cumX2$_d$ $\leftarrow (X_{1,d})^2$
13: end for
    // Compute remaining output rows
14: for $n \leftarrow 2$ to $N$ do
15:     out$_n$ $\leftarrow 0$
16:     for $d \leftarrow 1$ to $D$ do
17:         cumX$_d$ $\leftarrow$ cumX$_d$ $+ X_{1,d}$
18:         cumX2$_d$ $\leftarrow$ cumX2$_d$ $+ (X_{1,d})^2$
19:         out$_n$ $\leftarrow$ out$_n$+cumX2$_d$$-($cumX$_d$$\times$cumX$_d$$/n)$
20:     end for
21: end for
22: return out
```

D. Aggregation Using Pairwise Averages

Recall that we estimate sums of low-bitwidth integers by averaging pairs of values, then pairs of pairs, and so on. One could reduce all C values this way, but we find that one obtains a better speed-accuracy tradeoff by computing the average of blocks of U values and then upcasting to obtain exact sums of these averages. Multiplying this sum of averages by U and adding in a bias correction term gives one the overall estimate of the sum. One could tune U for a particular problem and hardware, but we simply set $U = 16$ in all our experiments. Having a larger U imposes less overhead because upcasting happens less often, but there are sharp diminishing returns to this; once upcasting is rare, doing it even less often is of little help thanks to Amdahl's law.

Because of our assumption that we are operating on matrices, rather than a matrix and a vector, we can also improve on the aggregation of existing methods (Blalock & Guttag, 2017; André et al., 2017; 2019) by fusing the aggregation of two or more output columns to hide read latency. Conceptually, this amounts to tiling the loop over output columns and alternating reads between the two corresponding tables within the innermost loop. This fusion does not change the output of the computation—only how efficiently it runs.

Having addressed these practical details, we may now proceed to the analysis of our estimator's bias.

Definition D.1 (Averaging Integer Sum Estimator)**.** *Let $\boldsymbol{x} \in \{0,1\}^C, C \,\%\, U = 0, U = 2^p, p \geq 0$. The Averaging Integer Sum Estimator (AISE) $\hat{s}(\boldsymbol{x})$ is defined as:*

$$\hat{s}(\boldsymbol{x}) \triangleq \sum_{k=1}^{C/U} \hat{s}_U(\boldsymbol{x}_{i_k:j_k}) \tag{17}$$

$$\hat{s}_U(\boldsymbol{x}) \triangleq \begin{cases} x_1 & \boldsymbol{x} \in \mathbb{R}^1 \\ \lfloor \frac{1}{2}(\hat{s}_U(\boldsymbol{x}_{left}) + \hat{s}_U(\boldsymbol{x}_{right}) + 1) \rfloor & \textit{otherwise} \end{cases} \tag{18}$$

where $i_k = (k-1) \cdot U + 1, j_k = i_U + U$ and $\boldsymbol{x}_{left}$ and $\boldsymbol{x}_{right}$ denote vectors formed by taking the initial and final $D/2$ indices of a given $\boldsymbol{x} \in \mathbb{R}^D$.

Definition D.2 (Pairwise Integer Sum and Sum Estimator)**.** *For integers a and b, define*

$$s(a,b) \triangleq a + b \tag{19}$$

$$\hat{s}(a,b) \triangleq 2\mu(a,b) \tag{20}$$

where $\mu(a,b) \triangleq \lfloor \frac{1}{2}(a+b+1) \rfloor$.

Lemma D.1 (Bias when averaging one pair)**.** *Consider two scalars a and b, with $a, b \overset{iid}{\sim}$ Bernoulli(.5). Define $\varepsilon(a,b) \triangleq \hat{s}(a,b) - s(a,b)$. Then*

$$E[\varepsilon(a,b)] = \frac{1}{2}$$

Proof. The proof follows immediately from considering the four equiprobable realizations of the pair a, b. In the cases $(0,0)$ and $(1,1)$, $2\mu(a,b) = s(a,b)$. In the cases $(0,1)$ and $(1,0)$, $2\mu(a,b) = 2$, while $s(a,b) = 1$. □

Lemma D.2 (Variance of error when averaging one pair)**.** *Consider two scalars a and b, $a, b \overset{iid}{\sim}$ Bernoulli(.5). Then*

$$E[\varepsilon(a,b)^2] - E[\varepsilon(x,y)]^2 = \frac{1}{4}$$

Proof. Using Lemma D.1, the above can be rewritten as:

$$E[\varepsilon(a,b)^2] = \frac{1}{2}$$

The proof then follows by again considering the four equiprobable cases as in Lemma D.1. In the cases $(0,0)$ and $(1,1)$, $\varepsilon(a,b)^2 = 0$. In the cases $(0,1)$ and $(1,0)$, $(2\hat{s}(a,b) - s(a,b))^2 = (2-1)^2 = 1$. □

Lemma D.3 (Bias of AISE within a subspace)**.** *Suppose that the scalar elements x_i of $\boldsymbol{x}$ are drawn from independent Bernoulli(.5) distributions. Then*

$$E[s_U(\boldsymbol{x}) - \hat{s}_U(\boldsymbol{x})] = U \log_2(U)/4 \tag{21}$$

Proof. Observe that the computation graph can be cast as a balanced binary tree with U leaves and each parent equal to the integer average of its children. Consider the bias introduced at each level t of the tree, where $t = 0$ corresponds to the leaves and $t = \log_2(U)$ corresponds to the root. The expected error $E[\xi(t,n)]$ introduced at a node n in level $t > 0$ is given by:

$$E[\xi(t,n)] = \frac{1}{2} \cdot 2^{t-1} \tag{22}$$

where the $\frac{1}{2}$ follows from Lemma D.1 and the scale 2^{t-1} is the number of leaf nodes to which the bias is effectively applied. E.g., adding one to the estimated average of four leaf nodes would increase the estimated sum by four. Since there are $U \cdot 2^{-t}$ nodes per level, this means that the total expected error introduced at level t is $\frac{1}{2} \cdot 2^{t-1} \cdot 2^{-t} = \frac{1}{4}$. Summing from $t = 1$ to $t = \log_2(U)$ completes the proof of the expected error. Note that $t = 0$ is omitted since the leaf nodes are not the result of averaging operations and so introduce no error.

□

Theorem D.1 (Bias of AISE)**.** *Suppose that the scalar elements x_i of $\boldsymbol{x}$ are drawn from independent Bernoulli(.5) distributions. Then*

$$E[s(\boldsymbol{x}) - \hat{s}(\boldsymbol{x})] = C \log_2(U)/4 \tag{23}$$

Proof. This follows immediately from Lemma D.3, the fact that the overall sum is estimated within each of C/U subspaces of size U, and the assumption that the errors in each subspace are independent. $\square$

We also verified Theorem D.1 numerically by summing large numbers of integers drawn uniformly from the interval $0, \ldots, 255$.

Note that the assumption that the elements are independent is not especially strong in reality. This is because this section focuses on the effects on the least significant bits (which are the ones affected by each averaging operation), and the least significant bit does tend to be nearly uniformly random in a great deal of real-world data.

E. Additional Experimental Details

E.1. Choice of Matrix Multiplication Tasks

Because nearly all existing work on approximate matrix multiplication either focuses on special cases that do not satisfy our problem definition (André et al., 2019; Jegou et al., 2011; Ge et al., 2014) or synthetic matrices, there is not a clear set of benchmark matrix multiply tasks to use. We therefore propose a collection of tasks that we believe are both reproducible and representative of many real-world matrices. To the best of our knowledge, our experiments use over an order of magnitude more matrices than any previous study.

E.2. Choice of Single-Threaded Benchmarks

Given the ubiquity of GPUs and multicore CPUs, it may not be obvious why single-threaded experiments are desirable. There are a number of reasons we restrict our focus to CPUs and the single-threaded case:

- To enable fair comparisons to existing work, particularly the nearest rival, Bolt (Blalock & Guttag, 2017).
- To facilitate fair comparisons to our work by future authors—single-threaded experiments are much easier to reproduce and extend than multithreaded ones.
- Matrix multiplication is embarrassingly parallel with respect to rows of the input and columns of the output. There is therefore nothing "interesting" about how our method parallelizes relative to any other; all methods reduce to a single-threaded kernel that can easily be applied to disjoint submatrices. While we certainly could spend the considerable time required to construct and debug multicore benchmarks, this would be unlikely to yield any useful insights.
- Parallelization in modern numerical libraries is often managed at a higher level than the lowest-level subroutines. For example, the authors of FBGEMM (Khudia et al., 2018) state: *"Internally, FBGEMM is intentionally designed not to create any threads. Usually, such a library is intended to be used as a backend by deep learning frameworks, such as PyTorch and Caffe2, that create and manage their own threads."*[3] I.e., a multithreaded library calls into single-threaded subroutines (such as a matrix multiplication function); it is this single-threaded subroutine where we make contributions, and therefore where we focus our experimental efforts. Independent of common practices in modern libraries, this pattern is also the only sensible strategy for small matrices, like many of those we consider—the overhead of launching and joining threads is extremely unlikely to be worth it

[3]https://engineering.fb.com/ml-applications/fbgemm/

for sufficiently small matrices. We could perhaps characterize where this breakpoint is, but this is a hardware-specific result that has little to do with our contributions.

- While training of deep neural networks is typically done on GPUs or other accelerators, trained models (including, but not limited to, neural networks) are commonly deployed on smartphones with just CPUs and/or graphics acceleration that is no better than the CPU (Wu et al., 2019). Since most of the billions of smartphones in the world tend to be low-end or old, the need to deploy models on CPUs (including those with few cores) is unlikely to change for many years.
- Creating, benchmarking, and analyzing a performant implementation of our method for GPUs would require a great deal of engineering work. We plan to create such an implementation in the future, but believe that the many empirical and theoretical results we currently have are more than adequate proof of concept and already worth sharing with the community.

E.3. SparsePCA Details

We took steps to ensure that SparsePCA's results were not hampered by insufficient hyperparameter tuning. First, for each matrix product, we tried a range of λ values which we found to encompass the full gamut of nearly 0% to nearly 100% sparsity: $\lambda \in 2^i, i \in \{-5, -4, -3, -2, -1, 0, 1, 2, 3\}$. Second, because different sparsity patterns may yield different execution times, we report not times from the single matrix SparsePCA produces for a given (d, λ) pair, but the best times from any of 10 random matrices of the same size and at most the same sparsity. Finally and most importantly, we plot only the Pareto frontier of (speed, quality) pairs produced for a given matrix multiply. I.e., we let SparsePCA cherry-pick its best results on each individual matrix multiply.

E.4. Exact Matrix Multiplication

We also implemented our own matrix product function specialized for tall, skinny matrices. In all cases, we report the timings based on the faster of this function and Eigen's (Guennebaud et al., 2010) matrix multiply function for a given matrix product.

E.5. Additional Baselines

We also tested Frequent Directions / Fast Frequent Directions (Liberty, 2012; Ghashami et al., 2016; Desai et al., 2016), many variations of the sampling method of Drineas et al. (2006a), projection using orthogonalized Gaussian random matrices (Ji et al., 2012), projection using matrices of scaled i.i.d. Rademacher random variables (Achlioptas, 2001), projection using orthonormalized matrices of Rademacher random variables, the co-occurring directions sketch (Mroueh et al., 2016), OSNAP (Nelson & Nguyên, 2013), Product Quantization (Jegou et al., 2011), and Optimized Product Quantization (Ge et al., 2014).

The poor performance of many of these methods is unsurprising in our setting. Given that we have access to a training set on which to learn the true principal components, the Eckart-Young-Mirsky theorem (Eckart & Young, 1936) indicates that PCA should outperform any other individual matrix sketching method employing dense projection matrices, at least in the limit of infinite training data. Also, since PQ and OPQ use 256 dense centroids (except in the Bolt / QuickerADC variations), it is also impossible for them to perform well when $\min(D, M)$ is not significantly larger than 256.

E.6. UCR Time Series Archive

We set the number of returned neighbors to 128 (results with 64 and 256 were similar). We omitted datasets with fewer than 128 training examples, since it is not possible for Stochastic Neighbor Compression to draw 128 samples without replacement in this case.

In addition to being a large, public corpus of over a hundred datasets from a huge variety of different domains, the UCR Time Series Archive also has the advantage that it can be used to produce matrix multiplication tasks of a fixed size. This is necessary for meaningful comparison of speed versus accuracy tradeoffs across datasets. We constructed training and test matrices $\tilde{\boldsymbol{A}}$ and $\boldsymbol{A}$ by resampling each time series in each dataset's train and test set to a length of 320 (the closest multiple of 32 to the median length of 310). We obtained the matrix $\boldsymbol{B}$ for each dataset by running Stochastic Neighbor Compression (Kusner et al., 2014) on the training set with an RBF kernel of bandwidth one. We set the number of returned neighbors to 128 (results with 64 and 256 were similar), yielding a $\boldsymbol{B}$ matrix of size 320×128. Since different datasets have different test set sizes, all results are for a standardized test set size of 1000 rows. We wanted the length to be a multiple of 32 since existing methods operate best with sizes that are either powers of two or, failing that, multiples of large powers of two.

We approximate Euclidean distances using the identity $\|\boldsymbol{x} - \boldsymbol{y}\|_2^2 = \|\boldsymbol{x}\|_2^2 - 2\boldsymbol{x}^\top \boldsymbol{y} + \|\boldsymbol{y}\|_2^2$. We approximate only the inner products $\boldsymbol{x}^\top \boldsymbol{y}$, since $\|\boldsymbol{y}\|_2^2$ can be precomputed for fixed exemplars $\boldsymbol{y}$ and $\|\boldsymbol{x}\|_2^2$ doesn't affect the class prediction since it is constant across all exemplars for a given input $\boldsymbol{x}$.

E.7. Caltech101

We only extracted valid windows—i.e., never past the edge of an image. We extracted the windows in CHW order, meaning that scalars from the same color channel were placed at contiguous indices. The "first" images are based on filename in lexicographic order.

We used pairs of filters because using a single filter would mean timing a matrix-vector product instead of a matrix-matrix product.

To allow meaningful speed comparisons across images, we resized and center-cropped each image to 224×224 as commonly done in image classification pipelines (He et al., 2016a;b; Huang et al., 2017). We then extracted sliding windows of the appropriate size and used each (flattened) window as one row of $\tilde{\boldsymbol{A}}$ or $\boldsymbol{A}$. We similarly flattened the filters, with each set of coefficients forming one column of $\boldsymbol{B}$. In both cases, $\boldsymbol{B}$ has two columns—this is because using a single filter would mean timing a matrix-vector product instead of a matrix-matrix product. Two columns also made sense since Sobel filters are often used in horizontal and vertical pairings, and Gaussian filters are often used together to perform difference-of-Gaussians transforms.

Even though the RGB values at each position are naturally unsigned 8-bit integers, we allowed all rival methods to operate on them as 32-bit floating point, without including the conversion when timing them. Because it only requires checking whether values are above a threshold, MADDNESS can operate on 8-bit data directly.

E.8. Why Not Speed Up Whole Neural Nets?

Using our ideas to accelerate overall neural networks and other layer types would be a valuable contribution. In fact, we are actively working on this problem. However, as we state in the introduction and problem statement, our focus in this paper is *approximate matrix multiplication* (AMM) and we deliberately make no claim about accelerating entire neural networks or convolutional layers. We limit our scope in this way for several reasons:

1. Approximate matrix multiplication is an established research problem of general interest independent of deep learning.
2. Lifting a method from accelerating a single layer to an overall network is challenging. Just as scalar quantization of network parameters is simple for a single layer but an active area of research for an entire network, so too is using our method on multiple layers at once an open research problem. For example, it is not clear how to deal with the fact that distributions of activations change throughout training, or how to efficiently incorporate our non-differentiable hash function. We could show how to accelerate one internal FC layer in a network, but we dont want to risk misleading the reader—it would be unclear what conclusions to draw from such results, particularly given the difficulty of retraining / fine-tuning without introducing many lurking variables (c.f., (Blalock et al., 2020)).
3. It is correct that convolution can be reduced to GEMM using im2col, and that accelerating convolution using our ideas would be a valuable contribution. However, state-of-the-art algorithms for convolution exploit structure that is not available to general matrix multiply algorithms. To match the performance of specialized Winograd, direct, FFT-based, and hybrid convolution schemes that do exploit this additional structure, we would have to make modifications to our approach that would make it less general. For example, the individual spatial positions should be encoded only once, and then reused at multiple filter positions. Regarding Section 5.5: while we do test our method on matrices of flattened image patches, we do not claim that the overall pipeline of flattening + matrix multiply constitutes a state-of-the-art convolution method—we only claim that using our method in this pipeline outperforms using other AMM methods there.

In short, while we believe that our ideas show great promise for accelerating full neural networks and more layer types, making this happen requires much more research.

E.9. Additional Results

In Section 5, we showed the classification accuracy as a function of wall time for the CIFAR-10 and CIFAR-100 softmax classifiers, as well as on the UCR datasets. In Figure 8 and Figure 9, we instead show normalized mean squared error versus time. In Figure 10 and Figure 11, we show accuracy or NMSE versus number of operations performed, where one operation is either one multiply-add or one table lookup, depending on the method. The first two figures illustrate that NMSE is closely related to classification accuracy, but with imperfect NMSE still yielding excellent accuracy in many cases. The second two figures show that our method's superior results are not merely caused by the use of faster CPU instructions, but also by the use of fewer basic operations at the algorithm level.

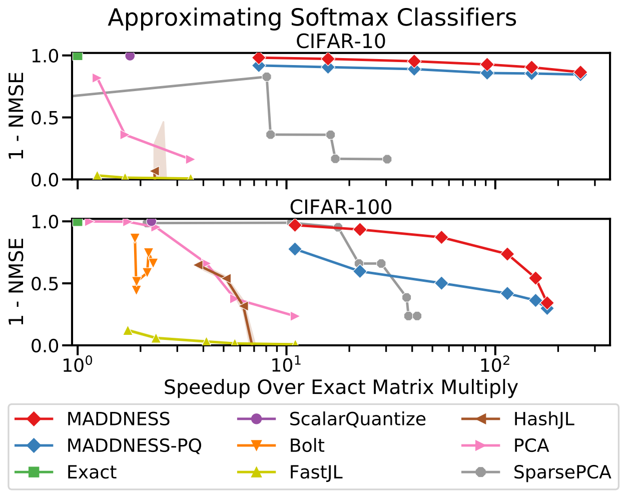

Figure 8: MADDNESS achieves a far better speed versus squared error tradeoff than any existing method when approximating two softmax classifiers. These results parallel the speed versus classification accuracy results, except that the addition of our ridge regression is much more beneficial on CIFAR-100.

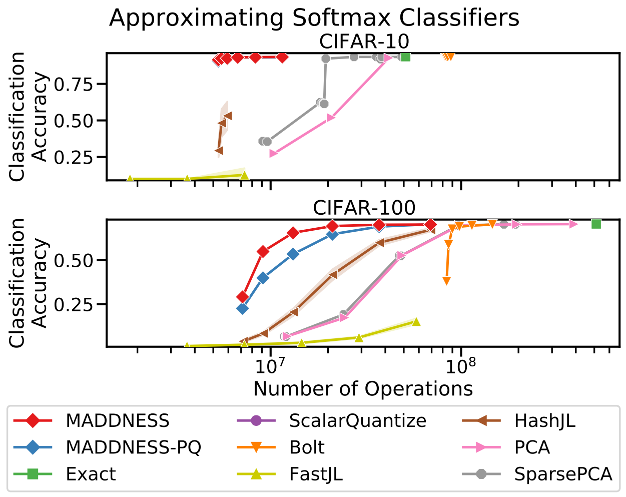

Figure 10: MADDNESS achieves the best speed versus accuracy tradeoff on the CIFAR datasets of any method even when speed is measured as number of operations instead of wall time. Note that fewer operations with a high accuracy (up and to the left) is better.

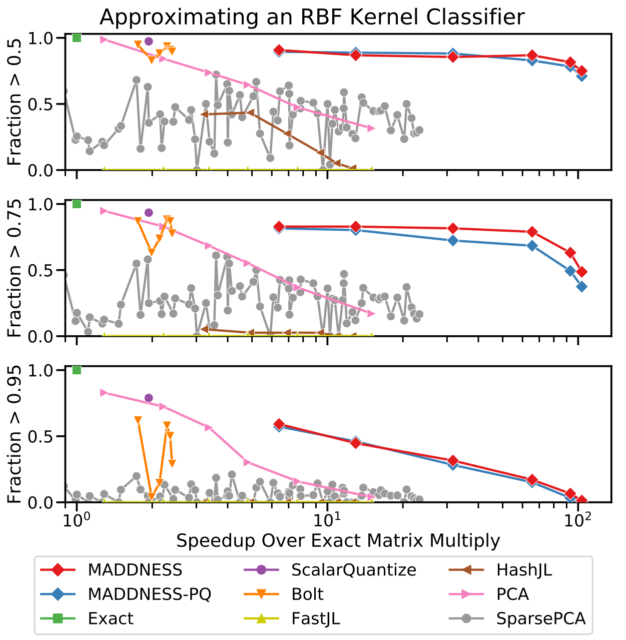

Figure 9: MADDNESS achieves the lowest squared error at high speedups on the UCR datasets. These results parallel the speed versus classification accuracy results.

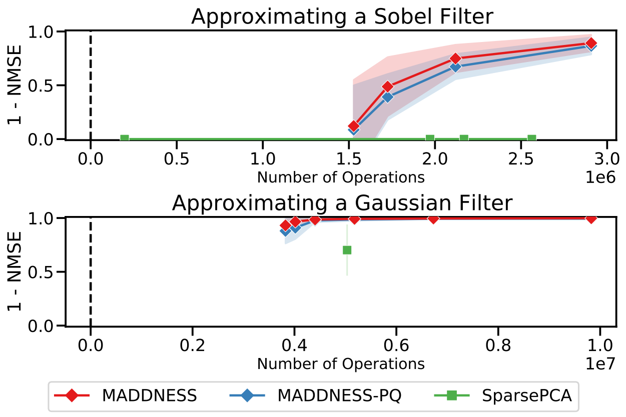

Figure 11: MADDNESS still achieves the best speed versus squared error tradeoff on the image processing tasks when speed is measured as number of operations instead of wall time.

F. Theoretical Analysis of MADDNESS

F.1. Complexity

Our encoding function $g(\boldsymbol{A}), \boldsymbol{A} \in \mathbb{R}^{N \times D}$ has complexity $\Theta(NC)$, since it does a constant amount of work per row per codebook. Our table creation function $h(\boldsymbol{B}), \boldsymbol{B} \in \mathbb{R}^{D \times M}$ has complexity $\Theta(MKCD)$, since it must compute the inner product between each column of $\boldsymbol{B}$ and KC prototypes of length D. This is a factor of C worse than PQ since we do not require the prototypes for different codebooks to have disjoint nonzero indices. However, this reduction in the speed of $h(\cdot)$ is not a concern because $N \gg M, D$; moreover, the matrix $\boldsymbol{B}$ is often known ahead of time in realistic settings, allowing $h(\boldsymbol{B})$ to be computed offline. Finally, the complexity of our aggregation function $f(\cdot)$ is $\Theta(NCM)$, since it performs C table lookups for each of M output columns and N output rows. This means our overall algorithm has complexity $\Theta(MC(KD + N))$, which reduces to $\Theta(NCM)$ since we fix $K = 16$ and our problem statement requires $N \gg D$.

F.2. Proof of Generalization Guarantee

In this section, we prove Theorem 4.1, restated below for convenience.

Theorem (Generalization Error of MADDNESS)**.** *Let $\mathcal{D}$ be a probability distribution over $\mathbb{R}^D$ and suppose that* MADDNESS *is trained on a matrix $\tilde{\boldsymbol{A}} \in R^{N \times D}$ whose rows are drawn independently from $\mathcal{D}$ and with maximum singular value bounded by σ_A. Let C be the number of codebooks used by* MADDNESS *and $\lambda > 0$ be the regularization parameter used in the ridge regression step. Then for any $\boldsymbol{b} \in \mathbb{R}^D$, any $\boldsymbol{a} \sim \mathcal{D}$, and any $0 < \delta < 1$, we have with probability at least $1 - \delta$ that*

$$\mathbb{E}_{\mathcal{D}}[\mathcal{L}(\boldsymbol{a}, \boldsymbol{b})] \leq \mathbb{E}_{\tilde{\boldsymbol{A}}}[\mathcal{L}(\boldsymbol{a}, \boldsymbol{b})] + \frac{C\sigma_A \|\boldsymbol{b}\|_2}{2\sqrt{\lambda}} \left(\frac{1}{256} + \frac{8 + \sqrt{\nu(C, D, \delta)}}{\sqrt{2n}} \right)$$

where $\mathcal{L}(\boldsymbol{a}, \boldsymbol{b}) \triangleq |\boldsymbol{a}^\top \boldsymbol{b} - \alpha f(g(\boldsymbol{a}), h(\boldsymbol{b})) - \boldsymbol{\beta}|$, α is the scale used for quantizing the lookup tables, $\boldsymbol{\beta}$ is the constants used in quantizing the lookup tables plus the debiasing constant of Section 4.4, and

$$\nu(C, D, \delta) \triangleq C(4 \lceil \log_2(D) \rceil + 256) \log 2 - \log \delta.$$

The proof relies on the observation that MADDNESS's training procedure can be decomposed into two sequential subroutines: `Maddness-Build-Tree`, which learns the function $g(\boldsymbol{a})$ by constructing a binary decision tree, and `Maddness-Regress`, which learns the function $h(\boldsymbol{b})$ by optimizing a prototype matrix $\boldsymbol{P}$ such that $g(\tilde{\boldsymbol{A}})\boldsymbol{P} \approx \tilde{\boldsymbol{A}}$. This observation allows us to prove 4.1 by first providing a guarantee for `Maddness-Regress` for a fixed `Maddness-Build-Tree` hypothesis, and then union bounding over the hypothesis space for `Maddness-Build-Tree`. Bounding the size of the hypothesis space is straightforward (Lemma F.1), so the bulk of this section focuses on providing a guarantee for `Maddness-Regress`. We must also prove a bound on the loss contributed by quantizing the lookup tables array $\boldsymbol{P}^\top \boldsymbol{b}$.

Lemma F.1 (Number of Hypotheses for `Maddness-Build-Tree`)**.** *Let C be the number of codebooks used by* MADDNESS *and let D be the number of columns in the matrix $\tilde{\boldsymbol{A}}$ on which* MADDNESS *is trained. Then there are at most $2^{C(4\lceil \log_2(D) \rceil + 256)}$ unique trees that `Maddness-Build-Tree` can generate.*

Proof. `Maddness-Build-Tree` learns four sets of parameters for each of the C trees it produces: split indices, split offsets, split scales, and split values.

There are four split indices per tree because there is one for each of the tree's four levels. Each index requires $\lceil \log_2(D) \rceil$ bits to store, so the split indices require a total of $4 \lceil \log_2(D) \rceil$ bits per tree. For each split index, there is one split offset and scale, used to map floating point data in an arbitrary range to the interval $[0, 255]$ to match up with the 8-bit split values.

The offsets require at most 25 bits each for 32-bit floating point data, since the low seven bits can be dropped without affecting the post-scaling quantized output. The scales are constrained to be powers of two, and so require at most nine bits for non-subnormal 32-bit floating point inputs (which have one sign bit and eight exponent bits). The offsets and scales together therefore contribute $4(25 + 9) = 136$ bits per tree.

There are 15 split values because there is one for the root of each tree, then two for the second level, four for the third, and eight for the fourth. Each split value is stored using eight bits, so each tree requires $15 \cdot 8 = 120$ bits for split values. The total number of bits used for all trees is therefore $C(4 \lceil \log_2(D) \rceil + 256)$. Note that the constant 256 being a power of two is just an accident of floating point formats. The claimed hypothesis count follows from the number of expressible hypotheses being at most two to the power of the largest number of bits used to store any hypothesis. □

We now turn our attention to bounding the errors of the regression component of training. Our strategy for doing so is to bound the largest singular value of the learned matrix of prototypes $\boldsymbol{P}$. Given such a bound, the norms of both $g(\boldsymbol{a})^\top \boldsymbol{P}$ and $\boldsymbol{P}^\top \boldsymbol{b}$ can be bounded.

Lemma F.2 (Regularized Pseudoinverse Operator Norm Bound)**.** *Let $\boldsymbol{X} \in \mathbb{R}^{N \times D}$ be an arbitrary matrix with finite elements. Then every singular value σ_i of the matrix $\boldsymbol{Z} \triangleq (\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I})^{-1} \boldsymbol{X}^\top$, $\lambda > 0$ is at most $\frac{1}{2\sqrt{\lambda}}$.*

Proof. Let $\boldsymbol{U\Sigma V}^\top$ be the singular value decomposition of $\boldsymbol{X}$. Then we have

$$\boldsymbol{Z} = (\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I})^{-1} \boldsymbol{X}^\top \tag{24}$$
$$= (\boldsymbol{V\Sigma U}^\top \boldsymbol{U\Sigma V}^\top + \lambda \boldsymbol{I})^{-1} \boldsymbol{V\Sigma U}^\top \tag{25}$$
$$= (\boldsymbol{V\Sigma}^2 \boldsymbol{V}^\top + \lambda \boldsymbol{I})^{-1} \boldsymbol{V\Sigma U}^\top \tag{26}$$
$$= (\boldsymbol{V\Sigma}^2 \boldsymbol{V}^\top + \boldsymbol{V}\lambda \boldsymbol{I V}^\top)^{-1} \boldsymbol{V\Sigma U}^\top \tag{27}$$
$$= (\boldsymbol{V\Sigma}_\lambda \boldsymbol{V}^\top)^{-1} \boldsymbol{V\Sigma U}^\top \tag{28}$$
$$= \boldsymbol{V\Sigma}_\lambda^{-1} \boldsymbol{V}^\top \boldsymbol{V\Sigma U}^\top \tag{29}$$
$$= \boldsymbol{V\Sigma}_\lambda^{-1} \boldsymbol{\Sigma U}^\top \tag{30}$$
$$= \boldsymbol{V\Sigma}' \boldsymbol{U}^\top \tag{31}$$

where $\boldsymbol{\Sigma}_\lambda \triangleq \boldsymbol{\Sigma}^2 + \lambda \boldsymbol{I}$ and $\boldsymbol{\Sigma}' \triangleq (\boldsymbol{\Sigma}^2 + \lambda \boldsymbol{I})^{-1} \Sigma$. Step 27 follows from the equality $\boldsymbol{V}\lambda \boldsymbol{I V}^\top = \lambda \boldsymbol{V V}^\top = \lambda \boldsymbol{I}$. Because the matrices $\boldsymbol{V}$ and $\boldsymbol{U}^\top$ are orthonormal and $\boldsymbol{\Sigma}'$ is diagonal, the singular values of $\boldsymbol{Z}$ are equal to the diagonal entries of $\boldsymbol{\Sigma}'$. Each entry σ_i' is equal to

$$\sigma_i' = \frac{\sigma_i}{\sigma_i^2 + \lambda}. \tag{32}$$

This expression attains its maximal value of $\frac{1}{2\sqrt{\lambda}}$ when $\sigma_i^2 = \lambda$. □

Lemma F.3 (Ridge Regression Singular Value Bound)**.** *Let $\boldsymbol{X} \in \mathbb{R}^{N \times D}$ and $\boldsymbol{Y} \in \mathbb{R}^{D \times M}$ be arbitrary matrices and let $\boldsymbol{W} \triangleq (\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I})^{-1} \boldsymbol{X}^\top \boldsymbol{Y}$, $\lambda > 0$ be the ridge regression weight matrix. Then $\|\boldsymbol{W}\|_\infty \leq \frac{\|\boldsymbol{Y}\|_\infty}{2\sqrt{\lambda}}$, where $\|\cdot\|_\infty$ denotes the largest singular value.*

Proof. Observe that $\boldsymbol{W} = \boldsymbol{ZY}$, where $\boldsymbol{Z} \triangleq (\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I})^{-1} \boldsymbol{X}^\top$. Then by applying Lemma F.2 and recalling that Schatten norms are submultiplicative, we have

$$\|\boldsymbol{W}\|_\infty \leq \|\boldsymbol{Z}\|_\infty \|\boldsymbol{Y}\|_\infty \leq \frac{\|\boldsymbol{Y}\|_\infty}{2\sqrt{\lambda}}. \tag{33}$$

□

Lemma F.4 (Bound on MADDNESS Embedding Norm)**.** *Let $\boldsymbol{g} = g(\boldsymbol{a})$ be the encoding of an arbitrary vector $\boldsymbol{a}$ using C codebooks and let $\boldsymbol{P}$ be the prototype matrix learned by MADDNESS using training matrix $\tilde{\boldsymbol{A}}$ with ridge regression parameter $\lambda > 0$. Then*

$$\|\boldsymbol{g}^\top \boldsymbol{P}\|_2 \leq \frac{C}{2\sqrt{\lambda}} \|\tilde{\boldsymbol{A}}\|_\infty \tag{34}$$

where $\|\tilde{\boldsymbol{A}}\|_\infty$ denotes the largest singular value of $\tilde{\boldsymbol{A}}$.

Proof. We have

$$\|\boldsymbol{g}^\top \boldsymbol{P}\|_2 \leq \|\boldsymbol{g}\|_2 \|\boldsymbol{P}\|_\infty \tag{35}$$
$$= C\|\boldsymbol{P}\|_\infty \tag{36}$$
$$\leq \frac{C}{2\sqrt{\lambda}} \|\tilde{\boldsymbol{A}}\|_\infty. \tag{37}$$

The first step follows from Cauchy-Schwarz. The second follows from $\boldsymbol{g}$ being zero except for exactly C ones. The last is an application of Lemma F.3. □

Lemma F.5 (Maximum Table Quantization Loss)**.** *Let $\hat{\boldsymbol{a}} = g(\boldsymbol{a})^\top \boldsymbol{P}$, where $g(\cdot)$ and P are trained using C codebooks and ridge regression penalty $\lambda > 0$ on a matrix $\tilde{\boldsymbol{A}}$ with maximum singular value at most σ_A, and $\boldsymbol{a} \in \mathbb{R}^D$ is an arbitrary vector. Then for any vector $\boldsymbol{b} \in \mathbb{R}^D$, $|\hat{\boldsymbol{a}}^\top \boldsymbol{b} - \hat{\boldsymbol{y}}| < \frac{C\sigma_A \|\boldsymbol{b}\|_2}{512\sqrt{\lambda}}$, where $\hat{\boldsymbol{y}} \triangleq \alpha g(\boldsymbol{a})^\top g(\boldsymbol{b}) + \boldsymbol{\beta}$ is MADDNESS's approximation to $\boldsymbol{a}^\top \boldsymbol{b}$. α and $\boldsymbol{\beta}$ are the scale and offsets used to quantize the lookup tables.*

Proof. If MADDNESS had infinite-precision lookup tables, $\hat{\boldsymbol{y}}$ would exactly equal $\hat{\boldsymbol{a}}^\top \boldsymbol{b}$. We therefore need only bound the error introduced by the quantization. By Lemma F.4, $\|\hat{\boldsymbol{a}}\|_2 \leq \frac{C\sigma_A}{2\sqrt{\lambda}}$. This implies that

$$\|\hat{\boldsymbol{a}}^\top \boldsymbol{b}\| \leq \frac{C\sigma_A \|\boldsymbol{b}\|_2}{2\sqrt{\lambda}} \tag{38}$$

and therefore

$$\frac{-C\sigma_A \|\boldsymbol{b}\|_2}{2\sqrt{\lambda}} \leq \hat{\boldsymbol{a}}^\top \boldsymbol{b} \leq \frac{C\sigma_A \|\boldsymbol{b}\|_2}{2\sqrt{\lambda}}. \tag{39}$$

For each of the C codebooks, this means that the value to be quantized lies in the interval $[\frac{-\sigma_A \|\boldsymbol{b}\|_2}{2\sqrt{\lambda}}, \frac{\sigma_A \|\boldsymbol{b}\|_2}{2\sqrt{\lambda}}]$ of width $\frac{\sigma_A \|\boldsymbol{b}\|_2}{\sqrt{\lambda}}$. Because MADDNESS quantizes the lookup tables such that largest and smallest entries for any row of $\boldsymbol{P}$ are linearly mapped to 255.5 and -0.5,[4] respectively, the worst-case quantization error is when the quantized value lies exactly between two quantization levels. We therefore need to compute the largest possible gap between a value and its quantization. Using 256 quantization levels, the largest possible gap is $1/(256/.5) = 1/512$ of the interval width. Multiplying by the above interval width yields a maximum quantization error for a given codebook of $\frac{\sigma_A \|\boldsymbol{b}\|_2}{512\sqrt{\lambda}}$. Because the errors in each subspace may not agree in sign, their sum is an upper bound on the overall quantization error. □

At this point, we have all of the pieces necessary to prove a generalization guarantee for `Maddness-Regress` save

[4]We use 255.5 and -0.5 rather than 255 and 0 because the latter only guarantees that a point is within $1/510$ of the interval width, not $1/512$. This is not an important choice and either option would be fine.

one: a theorem linking the norms of the various vectors and matrices involved to a probabilistic guarantee. Kakade et al. (2009) provide such a gaurantee, based on Rademacher complexity (Bartlett & Mendelson, 2002).

Theorem F.1 ((Kakade et al., 2009), Corollary 5)**.** *Let $\mathcal{F} = \{\boldsymbol{w}^\top \boldsymbol{x} : \|\boldsymbol{w}\|_2 \leq W\}$ be the class of linear functions with bounded L_2 norms, let $\mathcal{S}$ be a set of n samples drawn i.i.d. from some distribution $\mathcal{D}$ over the L_2 ball of radius X, and let $\mathcal{L}(f), f \in \mathcal{F}$ be a loss function with Lipschitz constant L. Then for any $0 < \delta < 1$, it holds with probability at least $1 - \delta$ over the sample $\mathcal{S}$ that*

$$\mathbb{E}_{\mathcal{D}}[\mathcal{L}(f)] \leq \mathbb{E}_{S}[\mathcal{L}(f)] + \frac{LXW}{\sqrt{2n}} \left(8 + \sqrt{-log(\delta)}\right). \tag{40}$$

We can now obtain our desired guarantee for the regression step.

Lemma F.6 (Generalization Error of `Maddness-Regress`)**.** *Let $\mathcal{D}$ be a probability distribution over $\mathbb{R}^D$ and suppose that* MADDNESS *is trained on a matrix $\tilde{\boldsymbol{A}} \in R^{N \times D}$ whose rows are drawn independently from $\mathcal{D}$ and with maximum singular value bounded by σ_A. Let C be the number of codebooks used by* MADDNESS *and $\lambda > 0$ the regularization parameter used in the ridge regression step. Further let $g(\boldsymbol{a})$ be a fixed (data-independent) function and $\mathcal{L}(\boldsymbol{a}, \boldsymbol{b}) \triangleq |\boldsymbol{a}^\top \boldsymbol{b} - f(g(\boldsymbol{a}), h(\boldsymbol{b}))|$. Then for all vectors $\boldsymbol{b}$, any vector $\boldsymbol{a} \sim \mathcal{D}$, and any $0 < \delta < 1$, we have with probability at least $1 - \delta$ that*

$$\begin{aligned}\mathbb{E}_{\mathcal{D}}[\mathcal{L}(\boldsymbol{a}, \boldsymbol{b})] \leq \mathbb{E}_{\tilde{\boldsymbol{A}}}[\mathcal{L}(\boldsymbol{a}, \boldsymbol{b})] + \frac{C\sigma_A \|\boldsymbol{b}\|_2}{512\sqrt{\lambda}} + \\ \frac{C\sigma_A \|\boldsymbol{b}\|_2}{2\sqrt{2n\lambda}} \left(8 + \sqrt{-log(\delta)}\right).\end{aligned} \tag{41}$$

Proof. The output of `Maddness-Regress` can be decomposed into

$$\hat{\boldsymbol{y}} \triangleq f(g(\boldsymbol{a}), h(\boldsymbol{b})) = \boldsymbol{g}^\top \boldsymbol{P}\boldsymbol{b} + \varepsilon + \zeta \tag{42}$$

where $\boldsymbol{g} = g(\boldsymbol{a})$, $\boldsymbol{P}$ is the matrix of prototypes, ε is data-independent noise from the averaging process[5], and ζ is noise from quantizing the lookup table entries. By Lemma F.5, $\zeta \leq \frac{C\sigma_A \|\boldsymbol{b}\|_2}{512\sqrt{\lambda}}$ (accounting for the second term in Equation 41). We therefore need only obtain a guarantee for $|\boldsymbol{g}^\top \boldsymbol{P}\boldsymbol{b} - \boldsymbol{a}^\top \boldsymbol{b}|$. Defining $\boldsymbol{w} \triangleq \boldsymbol{P}\boldsymbol{b}$, we see that `Maddness-Regress` is a linear model, and therefore subject to Theorem F.1. Given an upper bound on the Lipschitz constant of the loss, a bound on the L_2 norm of $\boldsymbol{g}$, and a bound on the L_2 norm of $\boldsymbol{w}$, we can apply this theorem. The Lipschitz constant for the absolute loss is 1. The L_2 norm of $\boldsymbol{g}$ is exactly C. The L_2 norm of $\boldsymbol{w}$ can be bounded as

$$\|\boldsymbol{w}\|_2 = \|\boldsymbol{P}\boldsymbol{b}\|_2 \leq \|\boldsymbol{P}\|_\infty \|\boldsymbol{b}\|_2 \leq \frac{\sigma_A \|\boldsymbol{b}\|_2}{2\sqrt{\lambda}} \tag{43}$$

using Lemma F.3. □

Using this lemma, the proof of Theorem 4.1 is immediate; we begin with Lemma F.6 and simply union bound over all $2^{C(4\lceil \log_2(D) \rceil + 120)}$ hypotheses from Lemma F.1.

[5]We continue to make the assumption that the least significant bits of the lookup table entries are independent Bernoulli(0.5) random variables, which is nearly true in practice. Even if this assumption does not hold, this noise does not contribute to the generalization gap unless it differs between train and test sets.