Multiplying Matrices Without Multiplying (ICML-2021)

D. Blalock (MosaicML, MIT CSAIL) and J. Guttag (MIT CSAIL)

Jungtaek Kim

jtkim@postech.ac.kr

POSTECH
Pohang 37673, Republic of Korea
https://jungtaek.github.io

September 24, 2021

Introduction

- [Blalock and Guttag, 2021]
- Matrix multiplication is among the most fundamental subroutines used in machine learning and scientific computing.
- As a result, there has been a great deal of work on
  1. implementing high-speed matrix multiplication libraries,
  2. designing custom hardware to accelerate multiplication of certain classes of matrices,
  3. speeding up distributed matrix multiplication,
  4. designing efficient Approximate Matrix Multiplication (AMM) algorithms,

  under various assumptions.
- In this paper, the authors focus on the AMM task *under the assumptions that the matrices are tall, relatively dense, and resident in a single machine's memory*.

Introduction

- This setting arises naturally in machine learning and data mining when one has a data matrix $\boldsymbol{A}$ whose rows are samples and a linear operator $\boldsymbol{B}$ one wishes to apply to these samples.
- $\boldsymbol{B}$ could be a linear classifier, linear regressor, or an embedding matrix, among other possibilities:

$$\mathbf{t} = \boldsymbol{A}\boldsymbol{B}. \tag{1}$$

- As a concrete example, consider the task of approximating a softmax classifier trained to predict image labels given embeddings derived from a neural network.
- Here, the rows of $\boldsymbol{A}$ are the embeddings for each image, and the columns of $\boldsymbol{B}$ are the weight vectors for each class.

Approximating Softmax Classifiers

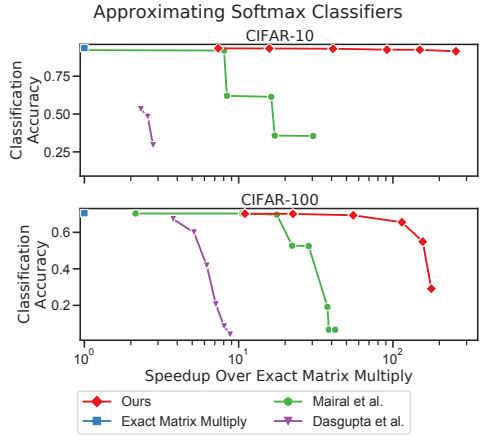

Figure 1: Ours achieves a dramatically better speed-accuracy tradeoff than the best existing methods when approximating two linear classifiers.

Introduction

- Traditional AMM methods construct matrices $\boldsymbol{V}_A, \boldsymbol{V}_B \in \mathbb{R}^{D \times d}$, $d \ll D$ such that

$$\boldsymbol{AB} \approx (\boldsymbol{AV}_A)(\boldsymbol{V}_B^\top \boldsymbol{B}). \tag{2}$$

- Often, $\boldsymbol{V}_A$ and $\boldsymbol{V}_B$ are sparse, embody some sort of sampling scheme, or have other structure such that these projection operations are faster than a dense matrix multiply.

Multiply-ADDitioN-IESS, a.k.a. MADDNESS

- It employs a nonlinear preprocessing function and reduces the problem to table lookups.
- Moreover, in the case that $\boldsymbol{B}$ is known ahead of time, which happens when applying a trained linear model to new data, among other situation, MADDNESS does not require any multiply-add operations.
- MADDNESS is most closely related to vector quantization methods used for similarity search.
- However, instead of using an expensive quantization function that requires many multiply-adds, we introduce a family of quantization functions that require no multiply-adds.

Problem Formulation

- Let $\boldsymbol{A} \in \mathbb{R}^{N \times D}$ and $\boldsymbol{B} \in \mathbb{R}^{D \times M}$ be two matrices, with $N \gg D \geq M$.
- Given a computation time budget τ, this task is to construct three functions $g(\cdot)$, $h(\cdot)$, and $f(\cdot, \cdot)$, along with constants α and $\boldsymbol{\beta}$, such that

$$\|\alpha f(g(\boldsymbol{A}), h(\boldsymbol{B})) + \boldsymbol{\beta} - \boldsymbol{AB}\|_F < \varepsilon(\tau)\|\boldsymbol{AB}\|_F, \tag{3}$$

  for as small an error $\varepsilon(\tau)$ possible.
- The constants α and $\boldsymbol{\beta}$ are separated from $f(\cdot, \cdot)$ so that $f(\cdot, \cdot)$ can produce low-bitwidth outputs even when the entries of $\boldsymbol{AB}$ do not fall in this range.
- We assume the existence of a training set $\tilde{\boldsymbol{A}}$, whose rows are drawn from the same distribution as the rows of $\boldsymbol{A}$.
- This is a natural assumption in the case that rows of $\boldsymbol{A}$ represent examples in training data, or structured subsets thereof (such as patches of images).

Background: Product Quantization

- Product quantization (PQ) [Jégou et al., 2011] is a classic vector quantization algorithm for approximating inner products and Euclidean distances.
- PQ serves as the basis for nearly all vector quantization methods.
- The basic intuition behind PQ is that $\boldsymbol{a}^\top \boldsymbol{b} \approx \hat{\boldsymbol{a}}^\top \boldsymbol{b}$, where $\|\hat{\boldsymbol{a}} - \boldsymbol{a}\|$ is small but $\hat{\boldsymbol{a}}$ has special structure allowing the product to be computed quickly.
- In somewhat more detail, PQ consists of the following:
  1. Prototype Learning,
  2. Encoding Function $g(\boldsymbol{a})$,
  3. Table Construction $h(\boldsymbol{B})$,
  4. Aggregation $f(\cdot, \cdot)$.

Background: Product Quantization

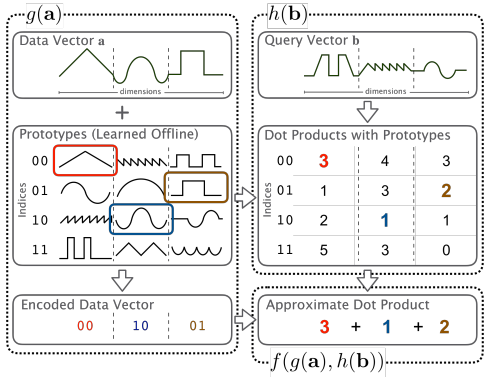

Figure 2: Product Quantization. The $g(\cdot)$ function returns the index of the most similar prototype to the data vector a in each subspace. The $h(\cdot)$ function computes a lookup table of dot products between the query vector b and each prototype in each subspace. The aggregation function $f(\cdot, \cdot)$ sums the table entries corresponding to each index.

Background: Prototype Learning

- Let $\tilde{\boldsymbol{A}} \in \mathbb{R}^{N \times D}$ be a training set, K be the number of prototypes per subspace, C be the number of subspaces, and $\{\mathcal{J}^{(c)}\}_{c=1}^{C}$ be the mutually exclusive and collectively exhaustive sets of indices associated with each subspace.
- The training-time task of PQ is to learn C sets of prototypes $\boldsymbol{P}^{(c)} \in \mathbb{R}^{K \times |\mathcal{J}^{(c)}|}$ and assignments $\boldsymbol{z}^{(c)}$ such that:

$$\sum_{i=1}^{N} \sum_{c=1}^{C} \sum_{j \in \mathcal{J}^{(c)}} \left(\tilde{\boldsymbol{A}}_{ij} - \boldsymbol{P}^{(c)}_{\boldsymbol{z}^{(c)},j} \right)^2, \tag{4}$$

  is minimized.
- It does this by running K-means separately in each subspace $\mathcal{J}^{(c)}$ and using the resulting centroids and assignments to populate $\boldsymbol{P}^{(c)}$ and $\boldsymbol{z}^{(c)}$.

Background: Encoding Function

- Given the learned prototypes, PQ replaces each row $\boldsymbol{a}$ of $\boldsymbol{A}$ with the concatenation of its C K-means centroid assignments in each of the C subspaces.
- Formally,

$$g^{(c)}(\boldsymbol{a}) = \arg\min_{k} \sum_{j \in \mathcal{J}^{(c)}} \left(\boldsymbol{a}_j - \boldsymbol{P}_{k,j}^{(c)} \right). \tag{5}$$

Background: Table Construction

- Using these same prototypes, PQ constructs a lookup table $h^{(c)}(\boldsymbol{b}) \in \mathbb{R}^K$ in each of the C subspaces for each column $\boldsymbol{b}$ of $\boldsymbol{B}$, where

$$h^{(c)}(\boldsymbol{b})_k = \sum_{j \in \mathcal{J}^{(c)}} \boldsymbol{b}_j \boldsymbol{P}^{(c)}_{k,j}. \tag{6}$$

Background: Aggregation

- Given the encoding of $\boldsymbol{a}$ and the lookup tables for $\boldsymbol{b}$, the product can be approximated as

$$\boldsymbol{a}^\top \boldsymbol{b} = \sum_{c=1}^{C} {\boldsymbol{a}^{(c)}}^\top \boldsymbol{b}^{(c)} \approx \sum_{c=1}^{C} h^{(c)}(\boldsymbol{b})_k, \tag{7}$$

  where $k = g^{(c)}(\boldsymbol{a})$.

MADDNESS

- PQ and its variants yield a large speedup with $N, M \gg D$.
- However, we require an algorithm that only needs $N \gg M, D$, a more relaxed scenario common when using linear models and transforms.
- In this setting, the preprocessing time $g(\boldsymbol{A})$ can be significant, since ND may be similar to (or even larger than) NM.
- To address this case, the authors introduce a new $g(\boldsymbol{A})$ function that yields large speedups even on much smaller matrices.
- The main idea behind our function is to determine the "most similar" prototype through locality-sensitive hashing [Indyk and Motwani, 1998].

Hash Function Family

- The family of hash functions in this paper is balanced binary regression trees, with each leaf of the tree acting as one hash bucket.
- The leaf for a vector x is chosen by traversing the tree from the root and moving to the left child if the value x_j at some index j is below a node-specific threshold v, and to the right child otherwise.
- To enable the use of SIMD instructions, the tree is limited to 16 leaves and all nodes at a given level of the tree are required to split on the same index j.

MADDNESSHASH

Algorithm 1 MADDNESSHASH

Input: vector $\boldsymbol{x}$, split indices $j^1, \ldots, j^4$, split thresholds $\boldsymbol{v}^1, \ldots, \boldsymbol{v}^4$
$i \leftarrow 1$ // node index within level of tree
for $t \leftarrow 1$ **to** 4 **do**
  $v \leftarrow \boldsymbol{v}_i^t$ // lookup split threshold for node i at level t
  $b \leftarrow x_{j^t} \geq v \; ? \; 1 : 0$ // above split threshold?
  $i \leftarrow 2i - 1 + b$ // assign to left or right child
end for
return i

Learning the Hash Function Parameters

- The split indices j^1, j^2, j^3, j^4 and split thresholds $\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3, \boldsymbol{v}_4$ are optimized on the training matrix $\tilde{\boldsymbol{A}}$ using a greedy tree construction algorithm.
- To describe this algorithm, we introduce the notion of a bucket $\mathcal{B}_i^t$, which is the set of vectors mapped to node i in level t of the tree.
- The root of the tree is level 0 and $\mathcal{B}_1^0$ contains all the vectors.
- The sum of squared errors (SSE) loss is used:

$$\mathcal{L}(j, \mathcal{B}) = \sum_{\boldsymbol{x} \in \mathcal{B}} \left(x_j - \frac{1}{|\mathcal{B}|} \sum_{\boldsymbol{x}' \in \mathcal{B}} x'_j \right)^2, \tag{8}$$

$$\mathcal{L}(\mathcal{B}) = \sum_j \mathcal{L}(j, \mathcal{B}). \tag{9}$$

Adding The Next Level to the Hashing Tree

Algorithm 2 Adding The Next Level to the Hashing Tree

Input: buckets $\mathcal{B}_1^{t-1}, \ldots, \mathcal{B}_{2^{t-1}}^{t-1}$, training matrix $\tilde{\boldsymbol{A}}$
// greedily choose next split index and thresholds
$\hat{\mathcal{J}} \leftarrow$ `heuristic_select_idxs`$(\mathcal{B}_1^{t-1}, \ldots, \mathcal{B}_{2^{t-1}}^{t-1})$
$l^{min}, j^{min}, \boldsymbol{v}^{min} \leftarrow \infty$, NaN, NaN
for $j \in \hat{\mathcal{J}}$ **do**
  $l \leftarrow 0$ // initialize loss for this index to 0
  $\boldsymbol{v} \leftarrow [\,]$ // empty list of split thresholds
  for $i \leftarrow 1$ **to** 2^{t-1} **do**
    $v_i, l_i \leftarrow$ `optimal_split_threshold`$(j, \mathcal{B}_i^{t-1})$
    `append`$(\boldsymbol{v}, v_i)$ // append threshold for bucket i
    $l \leftarrow l + l_i$ // accumulate loss from bucket i
  end for
  if $l < l^{min}$ **then**
    $l^{min} \leftarrow l, j^{min} \leftarrow j, \boldsymbol{v}^{min} \leftarrow \boldsymbol{v}$ // new best split
  end if
end for
// create new buckets using chosen split
$\mathcal{B} \leftarrow [\,]$
for $i \leftarrow 1$ **to** 2^{t-1} **do**
  $\mathcal{B}_{below}, \mathcal{B}_{above} \leftarrow$ `apply_split`$(v_i^{min}, \mathcal{B}_i^{t-1})$
  `append`$(\mathcal{B}, \mathcal{B}_{below})$
  `append`$(\mathcal{B}, \mathcal{B}_{above})$
end for
return $\mathcal{B}, l^{min}, j^{min}, v^{min}$

Optimizing the Prototypes

- Instead of suggesting prototypes or table optimizations based on knowledge of $\boldsymbol{B}$, the authors introduce a method to choose prototypes such that $\tilde{\boldsymbol{A}}$ can be reconstructed from its prototypes with as little squared error as possible.
- Let $\boldsymbol{P} \in \mathbb{R}^{KC \times D}$ be a matrix whose diagonal blocks of size $K \times |\mathcal{J}^{(c)}|$ consist of the K learned prototypes in each subspace c.
- The training matrix $\tilde{\boldsymbol{A}}$ can be approximately reconstructed as $\tilde{\boldsymbol{A}} = \boldsymbol{G}\boldsymbol{P}$, where $\boldsymbol{G}$ serves to select the appropriate prototype in each subspace.
- Rows of $\boldsymbol{G}$ are formed by concatenating the one-hot encoded representations of each assignment for the corresponding row of $\tilde{\boldsymbol{A}}$.
- For example if a row were assigned prototypes $\langle 3\ 1\ 2 \rangle$ with $K = 4$, $C = 3$, its row in $\boldsymbol{G}$ would be $\langle 0010\ 1000\ 0100 \rangle \in \mathbb{R}^{12}$.
- Finally, it is solved with ridge regression: $\boldsymbol{P} = (\boldsymbol{G}^\top \boldsymbol{G} + \lambda \boldsymbol{I})^{-1} \boldsymbol{G}^\top \tilde{\boldsymbol{A}}$.

Fast 8-bit Aggregation

- Let $\boldsymbol{T} \in \mathbb{R}^{M \times C \times K}$ be the tensor of lookup tables for all M columns of $\boldsymbol{B}$.
- Given the encodings $\boldsymbol{G}$, the function $f(\cdot, \cdot)$ is defined as

$$f(g(\boldsymbol{A}), h(\boldsymbol{B}))_{n,m} = \sum_{c=1}^{C} \boldsymbol{T}_{m,c,k}, \tag{10}$$

  where $k = g^{(c)}(\boldsymbol{a}_n)$.

- We propose an alternative that sacrifices a small amount of accuracy for a significant increase in speed.
- Instead of using addition instructions, we use averaging instructions, such as `vpavgb` on x86 or `vrhadd` on ARM.

Theoretical Guarantees

Theorem 4.1 (Generalization Error of MADDNESS)**.** *Let $\mathcal{D}$ be a probability distribution over $\mathbb{R}^D$ and suppose that* MADDNESS *is trained on a matrix $\tilde{\boldsymbol{A}} \in \mathbb{R}^{N \times D}$ 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*

$$\mathbb{E}_{\mathcal{D}}[\mathcal{L}(\boldsymbol{a}, \boldsymbol{b})] \leq \mathbb{E}_{\tilde{\boldsymbol{A}}}[\mathcal{L}(\boldsymbol{a}, \boldsymbol{b})] + \frac{C \sigma_A \|\boldsymbol{b}\|_2}{2\sqrt{\lambda}} \left(\frac{1}{256} + \frac{8 + \sqrt{\nu(C, D, \delta)}}{\sqrt{2n}} \right) \tag{10}$$

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

$$\nu(C, D, \delta) \triangleq C(4 \lceil \log_2(D) \rceil + 256) \log 2 - \log \delta. \tag{11}$$

How Fast is MADDNESS?

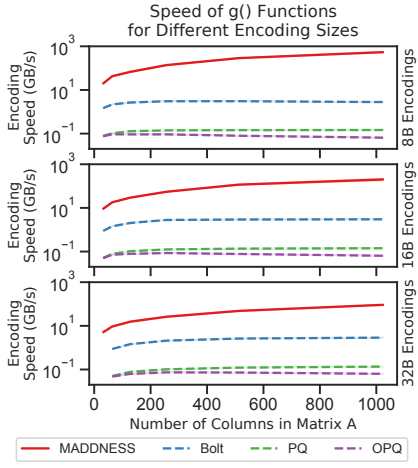

Figure 3: MADDNESS encodes the $\boldsymbol{A}$ matrix orders of magnitude more quickly than existing vector quantization methods.

How Fast is MADDNESS?

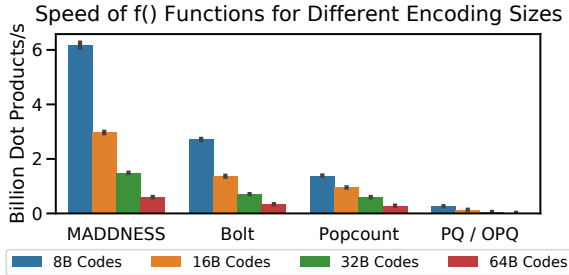

Figure 4: Given the preprocessed matrices, MADDNESS computes the approximate output twice as fast as the fastest existing method.

Softmax Classifier

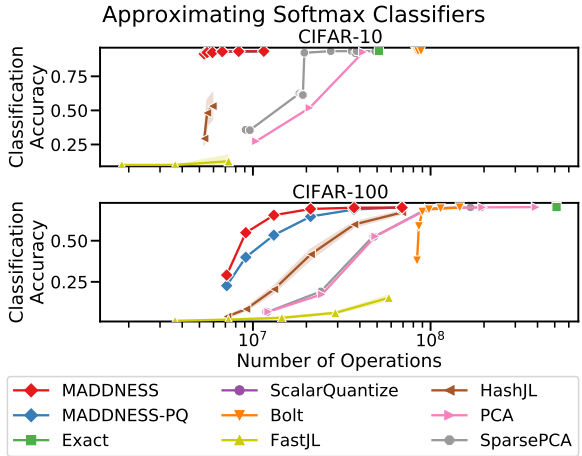

Figure 5: MADDNESS achieves a far better speed- accuracy tradeoff than any existing method when approximating two softmax classifiers.

Kernel-Based Classification

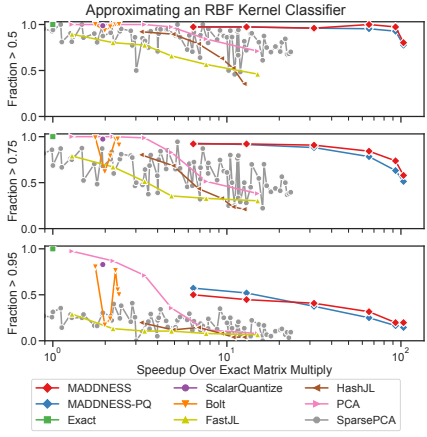

Figure 6: Fraction of UCR datasets for which each method preserves a given fraction of the original accuracy. MADDNESS enables much greater speedups for a given level of accuracy degradation.

Image Filtering

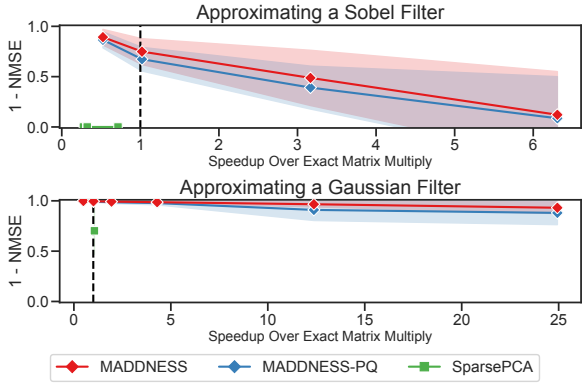

Figure 7: Despite there being only two columns in the matrix $\boldsymbol{B}$, MADDNESS still achieves a significant speedup with reasonable accuracy. Methods that are Pareto dominated by exact matrix multiplication on both tasks are not shown; this includes all methods but MADDNESS and SparsePCA.

Discussion and Conclusion

- MADDNESS achieves order-of-magnitude speedups compared to existing AMM methods
- It achieves up to two-order-of-magnitude speedups compared to the dense baseline.
- It also compresses matrices by up to three orders of magnitude.
- These results are evaluated on a CPU, and are obtainable only when there is a training set for one matrix.
- The authors also claim superior performance only when one matrix is larger than the other, and both matrices are tall–the regime wherein our extremely fast (but less accurate) encoding function is beneficial.

Discussion and Conclusion

- The authors have not demonstrated results using GPUs or other accelerators.
- They also have not evaluated a multi-CPU-threaded extension of this algorithm.
- They have not demonstrated results using convolutional layers in neural networks, or results accelerating full networks.

References I

D. Blalock and J. Guttag. Multiplying matrices without multiplying. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 992–1004, Virtual, 2021.

P. Indyk and R. Motwani. Approximate nearest neighbor: Towards removing the curse of dimensionality. In *Proceedings of the Annual ACM Symposium on Theory of Computing (STOC)*, pages 604–613, 1998.

H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2011.