TabConv: Low-Computation CNN Inference via Table Lookups

Neelesh Gupta[*]
University of Southern California
Los Angeles, California, USA
neeleshg@usc.edu

Narayanan Kannan[*]
University of California, Los Angeles
Los Angeles, California, USA
nkann19@ucla.edu

Pengmiao Zhang[*][†]
University of Southern California
Los Angeles, California, USA
pengmiao@usc.edu

Viktor Prasanna
University of Southern California
Los Angeles, California, USA
prasanna@usc.edu

ABSTRACT

Convolutional Neural Networks (CNNs) have demonstrated remarkable ability throughout the field of computer vision. However, CNN inference requires a large number of arithmetic operations, making them expensive to deploy in hardware. Current approaches alleviate this issue by developing hardware-supported, algorithmic processes to simplify spatial convolution functions. However, these methods still heavily rely on matrix multiplication, leading to significant computational overhead. To bridge the gap between hardware, algorithmic acceleration, and approximate matrix multiplication, we propose *TabConv*, a novel, table-based approximation for convolution to significantly reduce arithmetic operations during inference. Additionally, we introduce a priority masking technique based on cosine similarity to select layers for table-based approximation, thereby maintaining the model performance. We evaluate our approach on popular CNNs: ResNet-18, ResNet-34, and NetworkIn-Network (NIN). TabConv preserves over 93% of the original model's performance while reducing arithmetic operations by 36.5%, 25.8%, and 99.4% for ResNet-18 on CIFAR-10, CIFAR-100, and MNIST, respectively, 35.6% and 99.3% for ResNet-34 on CIFAR-10 and MNIST, and 98.9% for NIN on MNIST, achieving low-computation inference.

CCS CONCEPTS

• **Computing methodologies → Neural networks**; • **Theory of computation → Numeric approximation algorithms**.

KEYWORDS

convolutional neural network, table lookup, product quantization

ACM Reference Format:
Neelesh Gupta, Narayanan Kannan, Pengmiao Zhang, and Viktor Prasanna. 2024. TabConv: Low-Computation CNN Inference via Table Lookups. In *21st ACM International Conference on Computing Frontiers (CF '24), May 7–9, 2024, Ischia, Italy*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3649153.3649212

[*]These authors contributed equally.
[†]Corresponding author.

1 INTRODUCTION

Convolutional Neural Networks (CNNs) are ubiquitous in the field of computer vision. They are widely adopted as solutions to tasks in image classification [1, 2], object detection [3], semantic segmentation [4, 5], and many others [6]. This utility has fueled concerted efforts to improve CNN inference, yielding new models that surpass benchmarks. Unfortunately, these improvements in performance have resulted in significant increases in computational cost.

The continued expansion of CNN model complexity has caused an explosion in both the number of operations and amount of time required to process data and make predictions [7–9]. Moreover, this issue has all but rendered the traditional, multi-core CPU obsolete as a deep learning platform, thus mandating the usage of other dedicated hardware devices such as GPUs [10] and FPGAs [11] to power model operations. However, they too face their own unique set of challenges in energy consumption and a lack of standardization, respectively. Given these issues, much research into mitigating CNN computational costs focuses not on hardware, but on optimizing functions of the model itself.

Many of these efforts can be summarized as attempts to simplify the convolution function through mapping the process to an already well-optimized operation, such as matrix multiplication (MM) [12, 13]. Though they achieve a definite reduction in complexity, these methods leave ample room for improvement. High computational costs remain a bottleneck for inference latency, even when limited to refined processes. To combat this, techniques in approximating MM have been incorporated into Deep Neural Network (DNN) workflows, including those that simplify computation [14] and those that replace it entirely [15–17]. A promising approach involves the use of Product Quantization (PQ) to map MMs to table lookups [18]. However, existing works [15, 19] map only the MM of the last linear layer in a DNN to a table and suffer a large dropoff in accuracy when attempting more.

Motivated by shortcomings in both hardware and algorithm-based acceleration as well as the promise of table-based computation, we introduce *TabConv*, a novel low-computation CNN approximation based on table lookups that significantly reduces MMs in a CNN model while maintaining its performance. To achieve this, we follow a three-step process that integrates aspects of prior works with our own contributions: 1) Taking a trained CNN, converting all instances of convolution into MMs, 2) Mapping these MMs to table lookups based on product quantization, 3) Employing a novel priority masking approach to determine which layers should retain their exact computations rather than table-based approximations.

We solve two key problems during the development of TabConv-based CNN approximations. 1) The issue of mapping entire neural network layers with various operations–We introduce novel table approximation solutions for convolution, linear operations, batch normalization, and activation functions. 2) The compounding increase in error by layer once mapped to a table-based approximation–We combat this issue using a novel priority masking method that identifies which layers are best left in neural network form.

We summarize our main contributions below:

- We propose TabConv, a novel low-computation CNN approximation through table lookups, which significantly reduces the arithmetic operations required during inference while maintaining the CNN model performance.
- We design tabular primitives for operations in CNNs models, including convolution, batch normalization, linear operations, and activation functions.
- We propose a novel priority masking strategy that selectively converts certain layers in a model to table lookups while retaining other layers as exact calculations, balancing the trade-off between computation and accuracy.
- We evaluate TabConv on popular CNNs ResNet-18, ResNet-34, and NetworkInNetwork (NIN). Results show that while preserving higher than 93% of the original model performance, TabConv reduces 36.5%, 25.8%, and 99.4% of arithmetic operations for ResNet-18 on CIFAR-10, CIFAR-100, and MNIST, reduces 35.6% and 99.3% of arithmetic operations for ResNet-34 on CIFAR-10 and MNIST, and reduces 98.9% of arithmetic operations for NIN on MNIST.

2 BACKGROUND

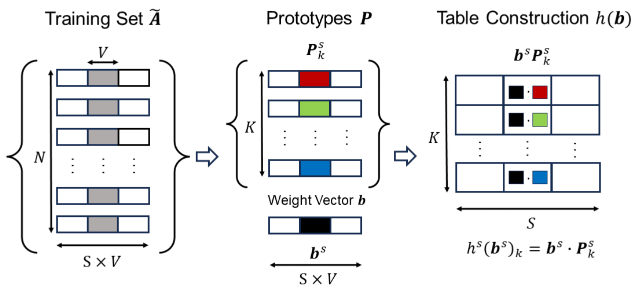
2.1 Product Quantization

Our approach is built upon the Product Quantization (PQ) algorithm [18], notable for its use in accelerating and approximating vector inner products through quantization and precomputation. In general, given an arbitrary vector $\mathbf{a} \in \mathbb{R}^D$ drawn from a training set $\tilde{\mathbf{A}} \in \mathbb{R}^{N \times D}$ with N samples, and a fixed weight vector $\mathbf{b} \in \mathbb{R}^D$, PQ generates a quantized approximation $\hat{\mathbf{a}}$ of $\mathbf{a}$ such that $\hat{\mathbf{a}}^\top \mathbf{b} \approx \mathbf{a}^\top \mathbf{b}$. In order to carry out this process, D-dimensional $\mathbf{a}$ is split into S disjoint, V-dimensional subspaces, within each of which K quantized subvectors are learned as prototypes. Note that since $\hat{\mathbf{a}}$ is quantized and $\mathbf{b}$ is fixed, their corresponding inner product is easily precomputed and reused in a query. Figure 1 provides an overview of the PQ process, while a detailed description follows below.

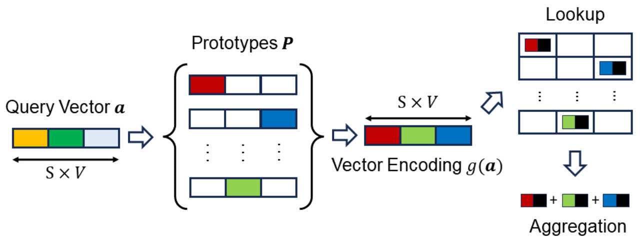
2.1.1 Training. The PQ training process includes both a prototype learning phase and table construction phase.

Prototype Learning (p): Consider $\tilde{\mathbf{A}}^\mathbf{s} \in \mathbb{R}^{N \times V}$, the vectors of the s-th subspace of $\tilde{\mathbf{A}}$. In the Prototype Learning phase, K prototypes, $\mathbf{P}_k^s$ (where k is the index of the prototypes within the s-th subspace), of $\tilde{\mathbf{A}}^\mathbf{s}$ are learned by minimizing the distance between the vectors of $\tilde{\mathbf{A}}^\mathbf{s}$ and their nearest corresponding prototype $\mathbf{P}_k^s$. The process is formulated in Equation 1 below.

$$p^c(\tilde{\mathbf{A}}) \triangleq \arg\min_P \sum_s \sum_i \left\|\tilde{\mathbf{A}}_i^s - \mathbf{P}_k^s\right\|^2 \tag{1}$$



(a) During PQ training, prototypes within each subspace are learned and their dot products with weight vector b are precomputed for storage in a table.



(b) During PQ query, the query vector is encoded to find the indices of its nearest prototypes, whose dot products are then looked up in the table and subsequently aggregated to yield the final result.

Figure 1: Training and query of product quantization.

Table Construction (h): Next, we construct a table with entries consisting of inner products between prototypes $\mathbf{P}_k^s$ and the weight vector $\mathbf{b}^s$ where s signifies the vector belonging to the s-th subspace. The function $h^s(\mathbf{b})_k$ describes the sk-th entry of the table.

$$h^s(\mathbf{b})_k \triangleq \mathbf{b}^{s\top} \cdot \mathbf{P}_k^s \tag{2}$$

2.1.2 Query. The query process eliminates the need for multiplication operations in the inner product calculation by encoding the query vector to its nearest prototype, looking up its corresponding table entries, and aggregating to yield the final result.

Vector Encoding (g): For arbitrary query vector $\mathbf{a}$, $g^s(\mathbf{a})$ locates its closest prototype $\mathbf{P}_k^s$ in each subspace s by finding the index k with minimal distance to $\mathbf{a}^s$. The function, as formulated below, outputs a set of indices representing the encoded vector of $\mathbf{a}$.

$$g^s(\mathbf{a}) \triangleq \arg\min_k \left\|\mathbf{a}^s - \mathbf{P}_k^s\right\|^2 \tag{3}$$

Lookup and Aggregation (f): After using the encoded indices to lookup the precomputed values, the corresponding entries by subspace are aggregated through the following function $f(\cdot,\cdot)$, yielding a final approximation for $\mathbf{a}^\top \mathbf{b}$.

$$f(\mathbf{a}, \mathbf{b}) = \sum_s h^s(\mathbf{b})_k, k = g^s(\mathbf{a}) \tag{4}$$

Thus, the actual dot product operation in $\mathbf{a}^\top \mathbf{b}$ is avoided by approximating the result through table lookups. We use locality sensitive hashing [15] for encoding and parallel summation for aggregation, resulting in significantly lower complexity than the dot product, especially for large dimensional vectors.

2.2 Im2col Convolution

The im2col method [12, 13] transforms the convolution function into a general instance of matrix multiplication (MM) by converting both the input image and kernel into patch matrices. An outline of this process is depicted in Figure 2. For simplicity's sake, we consider the case when stride is 1 and padding is 0.

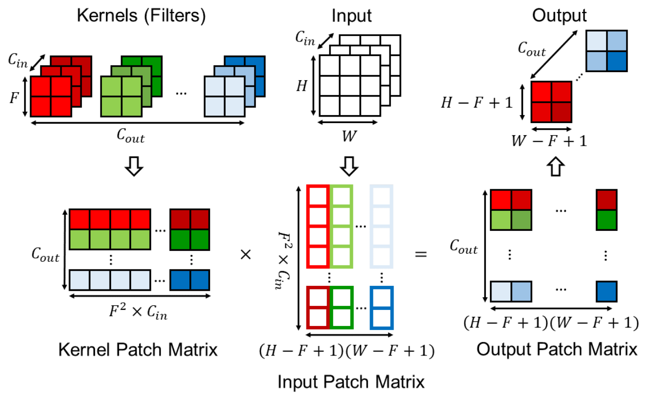


Figure 2: Outline of the im2col Method.

Input Patch Matrix. Given an input image with C_{in} channels of size $H \times W$ and C_{out} kernels (filters) of size $F \times F$, the input patch matrix is formed by taking kernel-sized patches of each input channel and flattening them into column vectors which are then concatenated to form a new matrix of size $F^2C_{in} \times (H-F+1)(W-F+1)$.
Kernel Patch Matrix. The kernel patch matrix is formed by reshaping each channel of each kernel into row vectors which are then concatenated similarly as above. The result is of size $C_{out} \times F^2C_{in}$.

We apply im2col as an intermediate step to transform the convolution function into an MM operation. Following this conversion, we replace subsequent MMs with approximate table lookups.

3 RELATED WORK

3.1 CNN Acceleration

There exists a rich body of literature on techniques to accelerate CNNs. Of particular note are those utilizing algorithmic processes, model compression methods, and hardware implementation. Many algorithm-based methods implement a mapping of the convolution function to an instance of matrix multiplication (MM), including the following processes: the im2col transformation [12, 13, 20], kn2row transformation [21], Winograd Minimal Filtering Method [22], Toeplitz matrix conversion [23], and Fast Fourier Transform convolutions [24, 25]. Model compression techniques like pruning [26, 27] and quantization [28, 29] reduce redundancy and model size to help performance. Acceleration through hardware entails the usage of GPUs with high performance computing capability [30, 31], and energy efficient FPGAs offering parallel acceleration tailored for CNNs [32, 33]. While these methods are successful in accelerating computation, they heavily feature MM. Eliminating these costly operations offers a new opportunity for advancement in CNN acceleration. In this paper, we introduce a novel approach that maps CNN to a series of fast tabular lookups. Further hardware implementation facilitates a parallel process with significantly reduced computational cost.

3.2 Approximate Matrix Multiplication

Techniques in approximate matrix multiplication use algorithmic methods to simplify computation. For example, sampling input matrices [14], finding sketches of matrices [34, 35], and random projection to lower dimensional subspaces [36, 37] all attempt to reduce the number of rows or columns being operated on. Other approaches go further by replacing MM outright through techniques in hashing, averaging, logarithm computation, and designing distributed algorithms [15, 38–40]. Notably, the Product Quanitzation (PQ) algorithm [18] is used to convert traditional instances of MM into a series of tabular lookups [15]. Our work presents a comprehensive design methodology and fine-tuning process that both maintains and accelerates performance for applying PQ to CNNs.

4 APPROACH

4.1 Problem Definition

Our objective is to refine CNN inference by employing table lookups to approximate its computations. Let $\mathcal{M}$ be a CNN model characterized by its parameters $\boldsymbol{\theta}$ where $\mathcal{M}(\mathbf{x};\boldsymbol{\theta})$ represents the model's output given input $\mathbf{x}$. Our objective is to construct a table-based approximation $\mathcal{T}$ with parameters $\boldsymbol{\phi}$ such that $\mathcal{T}(\mathbf{x};\boldsymbol{\phi})$ closely approximates the output of $\mathcal{M}$. We formalize this as follows:

$$\min_{\boldsymbol{\phi}} \left[\frac{1}{N} \sum_{i=1}^{N} \|\mathcal{M}(\mathbf{x}_i;\boldsymbol{\theta}) - \mathcal{T}(\mathbf{x}_i;\boldsymbol{\phi})\|^2 \right] \tag{5}$$

We aim to minimize the discrepancy between the outputs of the conventional and the table-based models, ensuring that the latter is able to reduce arithmetic operations while maintaining accuracy.

4.2 Overview of TabConv

For a given CNN-based model, we generate its approximation, a TabConv-based model, through a three-step strategy, as depicted in Figure 3, including: 1) Converting convolution operations within a model into matrix multiplications (MM) format; 2) Mapping the resulting MMs to table lookups; 3) Employing a novel priority masking strategy to strike a balance between accuracy and computation. Next, we introduce the detailed workflow.
Input: CNN-based model. The input to the TabConv process is a CNN-based model that has already been trained for a specific task. This model likely consists of convolutional layers, batch normalization, linear layers, and activation functions, depending on the architecture and task it was designed for.
Step 1: Converting to MMs. In the initial step, the key operations within the original CNN-based model are transformed into the format of MMs. For convolution operations, we use im2col [12] as described in Section 2.2. For batch normalization operations, we fold the operation into the im2col MMs by merging the normalized weights into the convolution weights (Section 4.3.4). This conversion facilitates the subsequent steps by providing a more structured and efficient representation of the operations involved in the model.

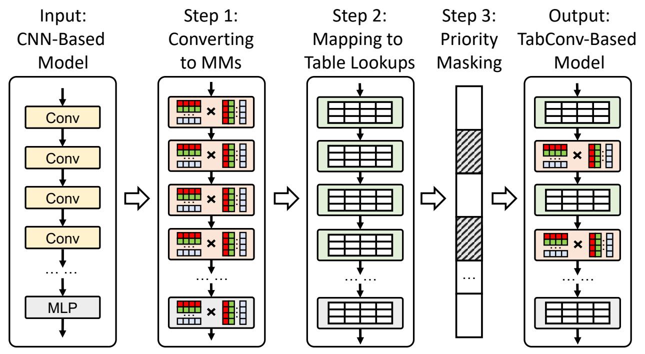

Input: CNN-Based Model
Step 1: Converting to MMs
Step 2: Mapping to Table Lookups
Step 3: Priority Masking
Output: TabConv-Based Model
Conv
Conv
Conv
Conv
MLP

Figure 3: Workflow of converting a CNN-based model to the proposed TabConv-based model.

Step 2: Mapping to table lookups. The resulting MMs are then mapped to table lookups based on product quantization (Section 2.1). We quantize the input data of each CNN and linear layer into a fixed size of prototypes (Section 4.3), precompute the dot product of the quantized vectors and the layer weights, and store the results in a table. In inference, we lookup the layer results from the closest prototypes to the input vectors, avoiding any MM operations.
Step 3: Priority masking. In deep CNNs, increasing the number of layers mapped to table lookups can lead to degraded approximation performance and accuracy, as deeper layers are likely to compound existing input errors. To address this, we introduce a novel priority masking method, employing a "similarity drop" metric to quantify differences between table and original CNN layers (Section 4.3.5).
Output: TabConv-based model. The output of this process is the TabConv-based model, an approximation of the original CNN-based model. This TabConv-based model closely approximates the predictive performance of the original model while significantly reduces the number of arithmetic operations involved in inference.

4.3 Mapping CNN to Table Lookups

We map operations in CNN to table lookups based on im2col and product quantization. In the following, we describe the process of constructing table CNN operations and how the table-based inference eliminates MM operations.

4.3.1 Table Construction for Convolution. Figure 4 illustrates the table construction for precomputed convolutional results. The im2col algorithm reshapes convolutional layer inputs from $C_{in} \times H \times W$ to $HW \times F^2C_{in}$ (assuming with paddings). This process creates a 2D matrix of dimensions $NHW \times F^2C_{in}$ by combining N data points from the training dataset. The F^2C_{in} dimension is divided into S subspaces, each learning K prototypes through methods like unsupervised clustering or locality-sensitive hashing [15]. After learning the K prototypes for each subspace, dot products are computed between these prototypes and the corresponding subspace of the kernel patch matrix, reshaped from $C_{out} \times F^2C_{in}$ kernel weights. This results in C_{out} sub-tables, one for each output channel, storing dot products between prototypes and channel weights with each entry representing the product for a subspace.

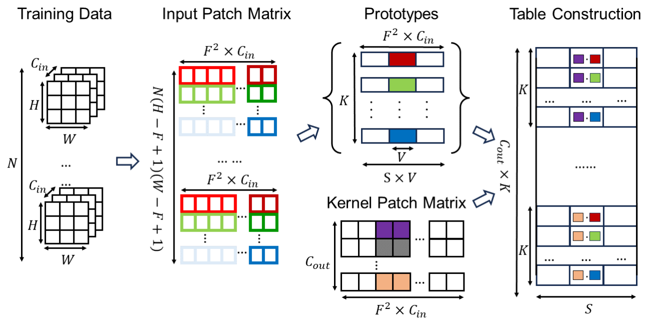


Figure 4: Table construction for convolutional operation.

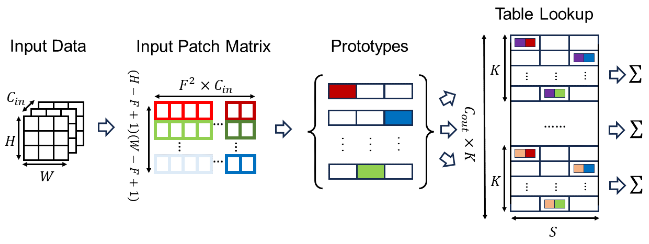


Figure 5: Table lookup for convolutional operation inference.

4.3.2 Table Lookups for Convolution Inference. Figure 5 shows the convolutional layer inference using the constructed table via lookup operations. Incoming query data (either an input image or a CNN layer input data) is reshaped to a patch matrix following the rule of im2col to a 2D matrix. Each row is split into S subspaces. Then, for each subvector in a subspace, we find the corresponding prototype by running the trained clustering or hashing function. All the $HW \times S$ subvectors are independent and can run the prototype matching in parallel. Using the index of the matched prototypes, the precomputed dot product for each output channel can be directly acquired by looking up from the C_{out} trained tables. The output lookup operations are also independent between output channels and can work in parallel. Finally, the subspace is aggregated through a simple sum operation, avoiding all MMs in convolution inference.

4.3.3 Linear Operation via Table Lookups. A linear operation is commonly used as the final classifier at the end of a CNN model. It transforms input $\mathbf{x}$ into an output $\mathbf{y}$ through a linear transformation, defined by the equation:

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b} \tag{6}$$

where $\mathbf{W}$ is the layer's weight matrix and $\mathbf{b}$ is the layer's biases. Figure 6 shows the process of constructing tables for a linear layer. For a training set of N inputs with input dimension D_{in}, we divide D_{in} into S subspaces, each with K prototypes. The linear layer weights are split similarly. We create a table with D_{out} sub-tables of $K \times S$ entries by storing dot product results between weight and prototype subvectors. To include the bias from the linear layer, we add it to a table column, ensuring its inclusion during the final aggregation in inference, as depicted in Figure 7. The linear layer inference then relies solely on prototype matching via hashing and table lookups, similar to table-based convolution.

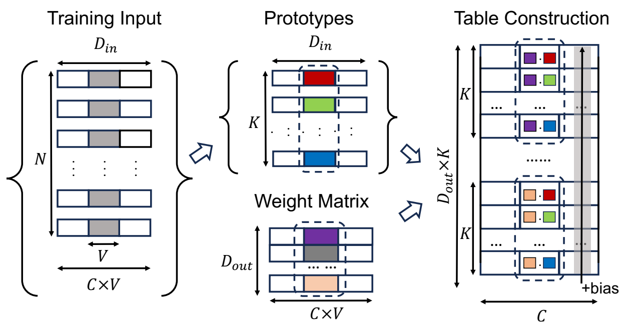


Figure 6: Table construction for linear operation.

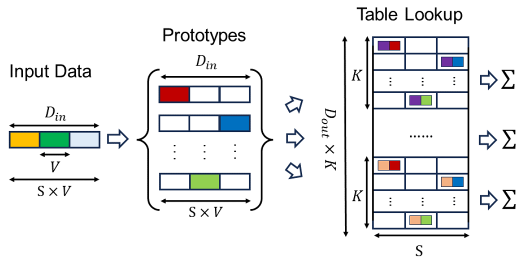


Figure 7: Table lookup for for linear layer inference.

4.3.4 Folding Batch Normalization. Batch normalization is widely used to speed up training and provide regularization for deep CNNs [41]. While batch normalization helps convergence, it requires an inference calculation after each convolution. To further reduce operations in a CNN model, we fold batch normalization operations into the convolution process, merging these two layers by transforming the weights and bias using the following equations:

$$\begin{aligned}\mathbf{W}' &= \mathbf{W}\frac{\gamma}{\sqrt{\sigma^2+\epsilon}}\\ \mathbf{b}' &= \frac{\gamma}{\sqrt{\sigma^2+\epsilon}}(\mathbf{b}-\mu)+\beta\end{aligned} \tag{7}$$

where γ and β are trained parameters, σ^2 is the variance, μ is the mean, and ϵ is a small constant.

4.3.5 Activation Function. The most commonly used non-linear activation function in CNNs is ReLU [42]. This step is combined with aggregation after table lookups, as shown in Equation 8.

$$f_{act}\left(\mathbf{a},\mathbf{b}\right) = \max\{0, \sum_{s} h^{s}\left(\mathbf{b}\right)_{k}\}, k = g^{s}\left(\mathbf{a}\right) \tag{8}$$

4.4 Priority Masking

When mapping more CNN layers to table lookups, the approximation error accumulates, and the prediction performance drops. To address this, we propose a novel priority masking strategy to retain exact operations crucial for maintaining model performance.

We introduce a priority masking rate $M \in [0,1]$ to adjust the masking, defined as the rate between the number of layers to be masked (not mapped to table) and the total number of model layers.

Algorithm 1 Priority Masking

```
1:  Input: Trained $N$-layer CNN model $\mathcal{M}$
2:  Input: Trained table-based approximation model $\mathcal{T}$
3:  Input: Training input data $\mathcal{D}$
4:  Input: Priority masking rate $M$
5:  Initialize: Layer similarity list $Sim$ of size $N$
6:  Initialize: Similarity difference list $SimDrop$ of size $N-1$
7:  Initialize: Priority list $Priority$ of size $N-1$
8:  Initialize: Mask list $Mask$ of size $N$, initialized as 1
9:  for $i$ in 0 to $N-1$ do                ▷ Get layer-wise similarity drop
10:     $\mathbf{y_i} \leftarrow \mathcal{M}[0:i](\mathcal{D})$
11:     $\hat{\mathbf{y}}_\mathbf{i} \leftarrow \mathcal{T}[0:i](\mathcal{D})$
12:     $Sim[i] \leftarrow S_C(\mathbf{y_i}, \hat{\mathbf{y}}_\mathbf{i})$
13:     if $i > 0$ then:
14:         $SimDrop[i] \leftarrow (Sim[i] - Sim[i-1], i)$
15:     end if
16: end for
17: $Priority \leftarrow$ Sorted($SimDrop$, Reverse=True)
18: for $i$ in 0 to $\lfloor M(N-1)\rceil$ do          ▷ Mask list update
19:     j = $Priority$[i][1]
20:     $Mask$[j] = 0
21: end for
22: for $i$ in 0 to $N-1$ do                ▷ Retrain using mask list
23:     if $Mask[i] = 0$ then
24:         $\mathcal{T}[i] \leftarrow \mathcal{M}[i]$
25:     else
26:         $\mathcal{T}[i] \leftarrow$ Retrain $(\mathcal{T}[i])$
27:     end if
28: end for
```

Given a CNN model $\mathcal{M}$, its table-based approximation $\mathcal{T}$, and a priority masking rate M, we calculate the cosine similarity S_C of each layer's output to compute the similarity drop between consecutive layers. We sort the similarity drop list in descending order, giving layers with the most significant drops in S_C the highest priority for masking. Based on this priority list, we update the mask list to 0 for the layer indices holding the $\lfloor M(N-1)\rceil$ largest drops in similarity. We then retrain the table-based model based on the mask list. Retraining the tables is necessary because the new architecture alters the input distribution to layers, affecting prototype matching and potentially amplifying existing input errors.

This selective masking ensures critical features are preserved, maintaining the accuracy of the model's inference while still benefiting from the efficiency of table lookups where appropriate.

4.5 Complexity Analysis

We analyze the computational complexity of our approach by examining arithmetic operations–an established indicator of inference latency [22, 24]–and storage costs.

4.5.1 Arithmetic Operations. Arithmetic operations result from the following two processes: vector encoding g to get table indices and aggregation f after the table lookup to output results. For a convolution of an input with C_{in} channels of size $H\times W$, stride S, and padding P with C_{out} kernels of size $F \times F$, it follows that g results in $H'W'S\log_2(K)$ operations and f results in $H'W'C_{out}log_2(S)$ operations, where $H' = \frac{H-F+2P}{T}+1$ and $W' = \frac{W-F+2P}{T}+1$. The

Table 1: Complexity analysis of TabConv and state-of-the-art acceleration methods.

	Arithmetic Operations (MFLOPs)		Storage Cost (MBytes)	
Approach	Expression*	Example†	Expression	Example
im2col [12]	$C_{out}H'W'(2F^2C_{in}-1)$	5.105	$C_{in}C_{out}F^2d$	0.036
kn2row [21]	$C_{out}H'W'(2F^2C_{in}-1)$	5.105	$C_{in}C_{out}F^2d$	0.036
Toeplitz [23]	$C_{out}H'W'(2F^2C_{in}-1)$	5.105	$C_{in}C_{out}F^2d$	0.036
FFT [24]	$\beta C_{in}C_{out}H'W'+\alpha^2C_{out}H'W'\left(\frac{1}{2}C_{in}+\frac{13}{16}C_{out}\right)$	0.272	$\beta C_{in}C_{out}F^2d+\alpha^2C_{out}d\left(\frac{1}{2}C_{in}+\frac{3}{2}C_{out}\right)$	0.032
TabConv	$H'W'\left[S\log_2(K)+C_{out}\log_2(S)\right]$	0.081	$H'W'S\log_2(K)+C_{out}SKd$	16.02

$* H' := \frac{H-F+2P}{T}+1, W' := \frac{W-F+2P}{T}+1, \beta := 1-\alpha^2$

† Example based on the first convolution in ResNet-18: $H=32, W=32, F=7, P=3, T=2, C_{in}=3, C_{out}=64, \alpha=0.5, S=8, K=8192, d=4$

total number of arithmetic operations (FLOPs) is then:

$$H'W'\left[S\log_2(K)+C_{out}\log_2(S)\right] \tag{9}$$

4.5.2 Storage Cost. Storage costs consist of the vector encoding results and table entries, where one index of an encoded prototype incurs a cost of $\log_2(K)$ bytes and we denote the data byte-length of a precomputed entry as d. The actual prototypes need not be stored, as we use the encoded indices to look up table results directly. Considering the same convolution setup and parameters as in Section 4.5.1, there are $H'W'$ prototype indices and $C_{out}SK$ total table entries. Thus, the resulting storage in bytes is:

$$H'W'S\log_2(K)+C_{out}SKd \tag{10}$$

4.5.3 Comparison with State-of-the-Art. We compare our approach to various widely used, well-optimized algorithmic acceleration methods, each of which is outlined below.

- **im2col** [12]: Im2col, as seen in Section 2.2, transforms convolution into matrix multiplication by creating patch matrices from the input image and kernels.
- **kn2row** [21]: Kn2row turns each $F \times F$ convolution into F^2 1×1 convolutions by shifting the final feature map.
- **Toeplitz** [23]: Toeplitz convolutions convert the input image to a Toeplitz matrix by unrolling kernel-sized patches.
- **Fast Fourier Transform (FFT)** [24]: FFT convolutions use Fourier transforms to compute large kernels' convolutions.

Table 1 describes the number of FLOPs and parameter counts for these methods alongside our TabConv implementation. C_{in} and C_{out} represent the number of input and output channels of an arbitrary image, respectively, while H and W signify its dimensions, F denotes kernel size, T stride, P padding, and d the data byte length (H' and W' are the output dimensions, calculated using the expressions outlined in $*$ in the Table legend). Additionally, $\alpha \in [0,1]$ in the FFT row is a parameter that varies by layer and kernel size [24], and S and K in the TabConv row are the number of subspaces and prototypes per subspace, respectively.

We provide example FLOPs and storage calculations for implementation in the first convolution of ResNet-18, with parameters as stated in †. Our work results in a 98.4% decrease in MFLOPs compared to im2col, kn2row, and Toeplitz, and a 70.2% decrease in MFLOPs compared to FFT. The configuration $S=8$, $K=8192$ in this case incurs large storage costs. Using $S=2$, $K=2048$ would increasingly reduce MFLOPs while requiring a lesser 29.3x storage compared to other approaches. Though storage cost is non-negligible, the number of arithmetic operations is largely reduced.

5 EXPERIMENTS

5.1 Experimental Setup

5.1.1 Models. We apply our approach, TabConv, to three distinct CNN models for evaluation, including ResNet-18, ResNet-34, and NetworkInNetwork (NIN). The models complexity of accuracy performance on a variety of datasets are shown in Table 2. The models, selected for their varied architectural features, are as follows:

- **ResNet-18** [7]: ResNet-18 is a part of the Residual Network family with 18 layers organized into blocks, allowing for apt training and better performance by using skip connections.
- **ResNet-34** [7]: ResNet-34 extends ResNet-18 by increasing its depth to 34 layers. It leverages extended residual blocks and skip connections to best capture complex features.
- **Network In Network (NIN)** [43]: NIN integrates micro networks as multilayer perceptrons into convolutional layers. We implement a 9 layer NIN for evaluation.

Table 2: Complexity and accuracy of models we implemented.

	Complexity		Accuracy		
Model	MFLOPs	Size (MB)	C10	C100	MN
ResNet-18 [7]	37.67	47.76	0.844	0.493	0.983
ResNet-34 [7]	75.49	87.19	0.821	0.515	0.986
NIN [43]	223.90	3.78	0.851	0.319	0.879

5.1.2 Datasets. TabConv is assessed on three datasets, consisting of varying image complexities and class diversities, as follows:

- **CIFAR-10 (C10)** [44] with 60K images and 10 classes.
- **CIFAR-100 (C100)** [45] with 60K images and 100 classes.
- **MNIST (MN)** [46] with 70K images and 10 classes.

5.1.3 Metrics. To comprehensively evaluate our approach against standard CNN performance, we consider the following metrics:

- **Accuracy:** determined by the percentage of test images correctly predicted as their true class, measuring the model's ability in prediction.

- **Arithmetic operations in FLOPs:** the number of floating-point operations (FLOPs) needed for a single inference, measures a model's computational complexity.
- **Storage cost in Bytes:** the number of bytes required to store the model, assessing the memory footprint of the model.

5.2 Evaluation of Table Lookup Mapping

We thoroughly investigate the performance of mapping CNNs to table lookups by tuning the following configurations: number of subspaces S, number of prototypes per subspace K, and priority masking rate M. Based on a case configuration $S = 8, K = 8192$ (8K), and $M = 0.4$, we explore the design space of each dimension through variable control, as shown in Table 3, Table 4, and Table 5.

Table 3: Accuracy when varying the number of subspaces S.

			ResNet18			ResNet34			NIN		
S	K	M	C10	C100	MN	C10	C100	MN	C10	C100	MN
1	8K	0.4	0.540	0.134	0.970	0.586	0.108	0.972	0.164	0.032	0.646
2	8K	0.4	0.542	0.116	0.980	0.636	0.130	0.974	0.274	0.038	0.716
4	8K	0.4	0.634	0.162	0.974	0.698	0.126	0.986	0.298	0.048	0.804
8	8K	0.4	0.696	0.172	0.982	0.728	0.164	0.986	0.472	0.064	0.812
16	8K	0.4	0.766	0.218	0.984	0.780	0.204	0.986	0.612	0.118	0.822

Table 4: Accuracy when varying the number of prototypes K per subspace.

			ResNet-18			ResNet-34			NIN		
S	K	M	C10	C100	MN	C10	C100	MN	C10	C100	MN
8	1K	0.4	0.684	0.162	0.978	0.736	0.132	0.978	0.336	0.040	0.726
8	2K	0.4	0.686	0.186	0.980	0.710	0.142	0.978	0.366	0.060	0.774
8	4K	0.4	0.722	0.180	0.982	0.724	0.154	0.980	0.422	0.064	0.806
8	8K	0.4	0.696	0.172	0.982	0.728	0.164	0.986	0.472	0.064	0.812
8	16K	0.4	0.684	0.136	0.984	0.718	0.110	0.978	0.512	0.060	0.816

Table 5: Accuracy when varying the priority masking rate M.

			ResNet-18			ResNet-34			NIN		
S	K	M	C10	C100	MN	C10	C100	MN	C10	C100	MN
8	8K	0	0.248	0.068	0.964	0.284	0.058	0.964	0.276	0.042	0.818
8	8K	0.2	0.590	0.130	0.976	0.416	0.096	0.986	0.372	0.038	0.814
8	8K	0.4	0.696	0.172	0.982	0.728	0.164	0.986	0.472	0.064	0.812
8	8K	0.6	0.840	0.254	0.988	0.806	0.220	0.988	0.474	0.186	0.838
8	8K	0.8	0.822	0.492	0.984	0.820	0.408	0.986	0.500	0.228	0.852

In Table 3, we examine how changing the number of subspaces (S) from 1 to 16 affects the accuracy of table-based model inferences. For ResNet-18, the accuracy drop on CIFAR-10, CIFAR-100, and MNIST is 0.226, 0.084, and 0.014 respectively. Similarly, for ResNet-34 and NIN across these datasets, we observe accuracy drops of 0.194, 0.096, 0.014, and 0.448, 0.086, 0.176 respectively with decreasing S. The results indicate that more subspaces enhance the approximation capability of table-based models.

In Table 4, we explore the influence of varying the number of prototypes per subspace (K) from 1K to 16K on model accuracy. Notably, in ResNet-18 on CIFAR-10, the performance drops by 0.038 when K changes from 4K to either 1K or 16K, suggesting an K for certain cases. While increasing K to 16K maximizes performance in specific scenarios, such as ResNet-18 for MNIST and NIN for CIFAR-10 and MNIST, too many prototypes can sometimes result in lower accuracy. Thus, carefully calibrating K is crucial, especially when operations are more straightforward to approximate. Despite this, increasing K provides higher accuracy in most cases even when the best accuracy is not from the maximum value of K.

Table 5 shows the impact of varying the priority masking rate (M) over 0, 0.2, 0.4, 0.6, and 0.8's effect on the accuracy of TabConv inference. When decreasing M from 0.8 to 0, ResNet-18 on CIFAR-10, CIFAR-100, and MNIST has an accuracy drop of 0.574, 0.424, 0.020 respectively. For ResNet-34 on CIFAR-10, CIFAR-100, and MNIST, table-based models undergo an accuracy drop of 0.536, 0.350, and 0.022 respectively. For NIN on CIFAR-10, CIFAR-100, and MNIST, table-based models undergo an accuracy drop of 0.224, 0.186, and 0.034 respectively. In particular, the accuracy drops for the MNIST dataset are low and tell that when models are able to achieve certain performance on specific datasets, increasing M does not help as much as it would on harder to approximate data.

5.3 Evaluation of Priority Masking

In evaluating the priority masking technique, we methodically replace 20%, 40%, 60%, and 80% of matrix multiplication operations in the forward pass with exact neural network counterparts based on cosine similarity, prioritizing layers with significant similarity reductions ($M = 0.2, 0.4, 0.6$, and 0.8). This strategy helps us to measure the effects of substituting matrix operations with exact calculations on a per-layer basis, highlighting key trade-offs between accuracy, computational complexity, and storage requirements.

In Figure 8, we present the curve of S_C for all models and all datasets with $S = 8$ subspaces and $K = 8K$ prototypes. For ResNet-18 on CIFAR-10 dataset, the full table average layer-wise S_C drop is 0.402. However, this decreases significantly to 0.289, 0.245, 0.182, and 0.019 as the masking rate (M) is increased to $0.2, 0.4, 0.6$, and 0.8, respectively. Furthermore, S_C in the last layer increases from 0.602 for the full table to 0.735, 0.793, 0.851, and 0.989 for varying values of M. This trend holds for all models over all datasets but not to the same severity. Since degree of approximation's closeness impacts the final accuracy, we want to minimize layer-wise drops in S_C as well as maximize S_C in last layer. In the case of NIN, applying more aggressive masking percentages does not improve accuracy nor cosine similarity to the same degree. We reason this is due to varying model architectures since NIN does not contain a linear layer, making final layer approximations more difficult.

5.4 Evaluation of Complexity

We comprehensively evaluate table-based models' complexity and tradeoffs, noting our work is a time-space tradeoff approach. Figure 9 shows the trade off between storage and number of operations for table-based models under varying values of M. For example in

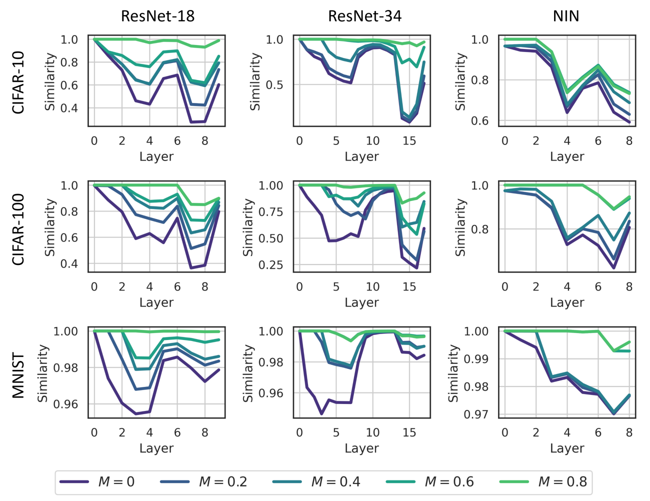


Figure 8: Layer-wise cosine similarity (S_C) when varying M.

CIFAR-10, ResNet-18 shows a FLOPs decrease from 39.5 million to as low as 260,000 in the process of converting the full CNN to tables. For ResNet-34, the reduction is from 79.1 million to around 388,000 FLOPs. The NIN model shows a significant drop from 234.8 million to about 2.5 million FLOPs. In terms of storage, ResNet-18's memory footprint goes from 11.7 MB to 343.0 MB when converting all operations to table lookups. ResNet-34's memory footprint goes from 21.8 MB to 594.5 MB when converting all operations to table lookups. NIN's memory footprint goes from 997.0 KB to 88.6 MB when converting all operations to table lookups.

5.5 Evaluation of Overall Performance

We introduce three metrics to evaluate the relative performance against standard CNNs: R_A, R_F, and R_S where R_A measures the accuracy ratio, R_F measures the ratio of total operations, and R_S measures the storage cost ratio between TabConv and CNNs.

In Table 6, we summarize the selection of priority masking rate M and TabConv overall performance given the lower bounds of R_A at 0.9, 0.8, and 0.7. For a bound $R_A \geq 0.9$, TabConv retains at least 93% of the original model's accuracy ($R_A \geq 0.93$), achieving a reduction in FLOPs of 36.5%, 25.8%, and 99.4% for ResNet-18 on CIFAR-10, CIFAR-100, and MNIST, respectively. It also reduces FLOPs by 35.6% and 99.3% for ResNet-34 on CIFAR-10 and MNIST, respectively, and achieves a 98.9% reduction in FLOPs for NIN on MNIST. When attaining over 80% of initial CNN accuracy, TabConv is able to reduce 58.6% and 60.5% of total operations during inference for ResNet-18 and ResNet-34 on CIFAR-10 respectively. When attaining over 70% of CNN accuracy, TabConv is able to reduce 77.2% of FLOPs for ResNet-18 on CIFAR-10, reduce 36.3% FLOPs for ResNet-34 on CIFAR-100, and reduce 10.8% FLOPs for NIN on CIFAR-100.

TabConv-based models exhibit a space-time tradeoff, necessitating on average a 32.0×, 33.4×, and 29.8× increase in memory footprint to retain 90%, 80%, and 70% accuracy, respectively, compared to CNN-based models. In general, our work effectively reduces the number of operations required for CNN inference while maintaining model accuracy, albeit with increased storage costs.

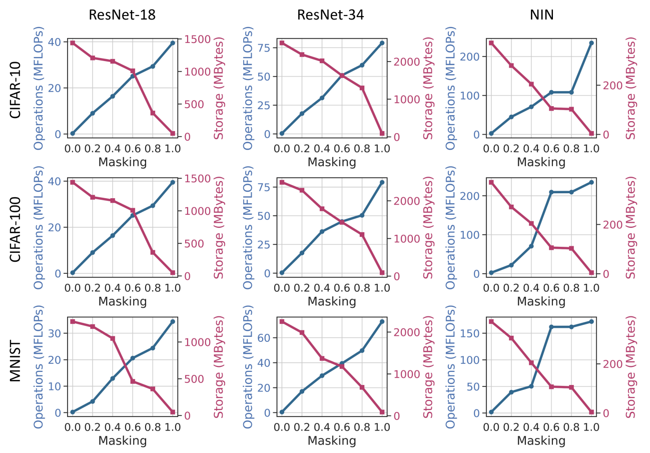


Figure 9: Operations and storage costs when varying M.

Table 6: TabConv masking selection and performance when maintaining thresholds of original CNN accuracy.

		ResNet-18			ResNet-34			NIN		
Bound	Res	C10	C100	MN	C10	C100	MN	C10	C100	MN
0.9	M	0.6	0.8	0	0.6	-	0	-	-	0
	R_A	0.995	0.998	0.981	0.982	-	0.984	-	-	0.931
	R_F	0.636	0.742	0.006	0.644	-	0.007	-	-	0.011
	R_S	20.61	7.377	27.26	17.81	-	25.21	-	-	94.80
0.8	M	0.4	0.8	0	0.4	-	0	-	-	0
	R_A	0.825	0.998	0.981	0.887	-	0.984	-	-	0.931
	R_F	0.414	0.742	0.006	0.395	-	0.007	-	-	0.011
	R_S	23.62	7.377	27.26	22.08	-	25.21	-	-	94.80
0.7	M	0.2	0.8	0.0	0.4	0.8	0	-	0.8	0
	R_A	0.700	0.998	0.981	0.887	0.792	0.984	-	0.715	0.931
	R_F	0.228	0.742	0.006	0.395	0.637	0.007	-	0.892	0.011
	R_S	24.64	7.38	27.26	22.08	12.09	25.21	-	26.03	94.80

6 CONCLUSION

We proposed *TabConv*, a novel CNN approximation based on converting key operations to table lookups. The key steps in our process include taking a trained CNN and converting all instances of convolution to matrix multiplication, mapping these to table lookups via product quantization, and employing a priority masking technique to identify which layers should be replaced by table approximation and which should retain their original form. Our TabConv-based model significantly reduces arithmetic operations while maintaining performance. In future work, we plan to optimize the table-based approximation through developing automatic configuration tools for table structure, quantized table entries for compression, and better hashing functions for prototype matching.

ACKNOWLEDGMENT

This work has been supported by the U.S. National Science Foundation (NSF) under grant CNS-2009057 and SaTC-2104264, as well as the DEVCOM Army Research Lab (ARL) under grant W911NF2220159.

REFERENCES

[1] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, pp. 211–252, 2015.

[2] H. Touvron, A. Vedaldi, M. Douze, and H. Jégou, "Fixing the train-test resolution discrepancy," *Advances in neural information processing systems*, vol. 32, 2019.

[3] R. Girshick, "Fast r-cnn," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1440–1448.

[4] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3431–3440.

[5] Y. Yuan, X. Chen, and J. Wang, "Object-contextual representations for semantic segmentation," in *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part VI 16*. Springer, 2020, pp. 173–190.

[6] P. Zhang, A. Srivastava, A. V. Nori, R. Kannan, and V. K. Prasanna, "Fine-grained address segmentation for attention-based variable-degree prefetching," in *Proceedings of the 19th ACM International Conference on Computing Frontiers*, ser. CF '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 103–112. [Online]. Available: https://doi.org/10.1145/3528416.3530236

[7] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[8] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2015.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[10] H. Kim, H. Nam, W. Jung, and J. Lee, "Performance analysis of cnn frameworks for gpus," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2017, pp. 55–64.

[11] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," *IEEE Micro*, vol. 35, no. 3, pp. 10–22, 2015.

[12] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[13] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *Tenth international workshop on frontiers in handwriting recognition*. Suvisoft, 2006.

[14] P. Drineas and R. Kannan, "Fast monte-carlo algorithms for approximate matrix multiplication," in *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 2001, pp. 452–459.

[15] D. Blalock and J. Guttag, "Multiplying matrices without multiplying," in *International Conference on Machine Learning*. PMLR, 2021, pp. 992–1004.

[16] M. Elhoushi, Z. Chen, F. Shafiq, Y. H. Tian, and J. Y. Li, "Deepshift: Towards multiplication-less neural networks," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2021, pp. 2359–2368.

[17] P. Zhang, N. Gupta, R. Kannan, and V. K. Prasanna, "Attention, distillation, and tabularization: Towards practical neural network-based prefetching," 2024.

[18] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 1, pp. 117–128, 2010.

[19] T. Yu, J. Yuan, C. Fang, and H. Jin, "Product quantization network for fast image retrieval," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 186–201.

[20] C. Salvador Rohwedder, J. P. Labegalini de Carvalho, J. Amaral, G. Araujo, G. Colmenares, and A. Wang, "Pooling acceleration in the davinci architecture using im2col and col2im instructions," 06 2021, pp. 46–55.

[21] A. Vasudevan, A. Anderson, and D. Gregg, "Parallel multi channel convolution using general matrix multiplication," in *2017 IEEE 28th international conference on application-specific systems, architectures and processors (ASAP)*. IEEE, 2017, pp. 19–24.

[22] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 4013–4021.

[23] Y. Chen, Y. Xie, L. Song, F. Chen, and T. Tang, "A survey of accelerator architectures for deep neural networks," *Engineering*, vol. 6, no. 3, pp. 264–274, 2020.

[24] L. Chi, B. Jiang, and Y. Mu, "Fast fourier convolution," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 4479–4488. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/2fd5d41ec6cfab47e32164d5624269b1-Paper.pdf

[25] J. H. Ko, B. Mudassar, T. Na, and S. Mukhopadhyay, "Design of an energy-efficient accelerator for training of convolutional neural networks using frequency-domain computation," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[26] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang, "Pruning and quantization for deep neural network acceleration: A survey," *Neurocomputing*, vol. 461, pp. 370–403, 2021.

[27] V. Radu, K. Kaszyk, Y. Wen, J. Turner, J. Cano, E. J. Crowley, B. Franke, A. Storkey, and M. O'Boyle, "Performance aware convolutional neural network channel pruning for embedded gpus," in *2019 IEEE International Symposium on Workload Characterization (IISWC)*, 2019, pp. 24–34.

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

[29] A. Ruospo, E. Sanchez, M. Traiola, I. O'Connor, and A. Bosio, "Investigating data representation for efficient and reliable convolutional neural networks," *Microprocessors and Microsystems*, vol. 86, 08 2021.

[30] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra *et al.*, "Can fpgas beat gpus in accelerating next-generation deep neural networks?" in *Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays*, 2017, pp. 5–14.

[31] S. Rovder, J. Cano, and M. O'Boyle, "Optimising convolutional neural networks inference on low-powered gpus," 2019.

[32] K. Abdelouahab, M. Pelcat, J. Serot, and F. Berry, "Accelerating cnn inference on fpgas: A survey," 2018.

[33] L. Sterpone, S. Azimi, and C. De Sio, "Cnn-oriented placement algorithm for high-performance accelerators on rad-hard fpgas," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–13, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10314789

[34] Q. Ye, L. Luo, and Z. Zhang, "Frequent direction algorithms for approximate matrix multiplication with applications in cca," *computational complexity*, vol. 1, no. m3, p. 2, 2016.

[35] Y. Mroueh, E. Marcheret, and V. Goel, "Co-occurring directions sketching for approximate matrix multiply," in *Artificial Intelligence and Statistics*. PMLR, 2017, pp. 567–575.

[36] A. Magen and A. Zouzias, "Low rank matrix-valued chernoff bounds and approximate matrix multiplication," in *Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete Algorithms*. SIAM, 2011, pp. 1422–1436.

[37] M. B. Cohen, J. Nelson, and D. P. Woodruff, "Optimal approximate matrix product in terms of stable rank," *arXiv preprint arXiv:1507.02268*, 2015.

[38] D. P. Francis and K. Raimond, "A practical streaming approximate matrix multiplication algorithm," *Journal of King Saud University-Computer and Information Sciences*, vol. 34, no. 1, pp. 1455–1465, 2022.

[39] M. S. Kim, A. A. Del Barrio, H. Kim, and N. Bagherzadeh, "The effects of approximate multiplication on convolutional neural networks," *IEEE Transactions on Emerging Topics in Computing*, vol. 10, no. 2, pp. 904–916, 2021.

[40] R. Li, Y. Xu, A. Sukumaran-Rajam, A. Rountev, and P. Sadayappan, "Efficient distributed algorithms for convolutional neural networks," *CoRR*, vol. abs/2105.13480, 2021. [Online]. Available: https://arxiv.org/abs/2105.13480

[41] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*. pmlr, 2015, pp. 448–456.

[42] A. F. Agarap, "Deep learning using rectified linear units (relu)," *arXiv preprint arXiv:1803.08375*, 2018.

[43] M. Lin, Q. Chen, and S. Yan, "Network in network," 2014.

[44] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research)." [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html

[45] ——, "Cifar-100 (canadian institute for advanced research)." [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html

[46] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.