SmartLite: A DBMS-based Serving System for DNN Inference in Resource-constrained Environments

Qiuru Lin
The State Key Laboratory of Blockchain and Data Security, Zhejiang University
qiurulin@zju.edu.cn

Sai Wu
The State Key Laboratory of Blockchain and Data Security, Zhejiang University
wusai@zju.edu.cn

Junbo Zhao
Zhejiang University
j.zhao@zju.edu.cn

Jian Dai
Alibaba Group
yiding.dj@alibaba-inc.com

Meng Shi
Zhejiang University
konicy@zju.edu.cn

Gang Chen
Zhejiang University
cg@zju.edu.cn

Feifei Li
Alibaba Group
lifeifei@alibaba-inc.com

ABSTRACT

Many IoT applications require the use of multiple deep neural networks (DNNs) to perform various tasks on low-cost edge devices with limited computation resources. However, existing DNN model serving platforms, such as TensorFlow Serving and TorchServe, are resource-intensive and require high-performance GPUs that are often not available on low-cost edge devices. In this paper, we propose SmartLite, a lightweight DBMS that addresses these challenges by storing the parameters and structural information of neural networks as database tables and implementing neural network operators inside the DBMS engine. SmartLite quantizes model parameters as binarized values, applies neural pruning techniques to compress the models, and transforms tensor manipulations into value lookup operations of the DBMS to reduce computation overhead. Experimental results show that SmartLite requires 98% less memory while achieving about a 134% performance speedup compared to TorchServe. Our proposed solution addresses the challenges of running multiple DNN models on low-cost edge devices and provides a significant contribution to the field of IoT applications.

PVLDB Reference Format:
Qiuru Lin, Sai Wu, Junbo Zhao, Jian Dai, Meng Shi, Gang Chen, and Feifei Li. SmartLite: A DBMS-based Serving System for DNN Inference in Resource-constrained Environments. PVLDB, 17(3): 278 - 291, 2023.
doi:10.14778/3632093.3632095

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/lynn2089/SmartLite.

1 INTRODUCTION

Most edge devices host an embedding database system to support basic data management and analytical tasks. For such devices, in-database machine/deep learning (in-DB ML/DL) inference [39, 40, 47, 59] enables timely and in-place predictive analysis, and eliminates the cost of data migration between database systems and DL platforms [15, 41, 46, 54]. Databases that can support multi-modal tasks, such as video databases [38] and text databases [63], are attracting the interest of researchers. An illustrative analytical SQL query, such as Q1, can be provided to demonstrate the scenario in which multiple in-DB DL models serve in the video database.

```
Q1: SELECT COUNT(*) FROM key_frame K
    WHERE M1(K.imageID)='rainy'
    AND K.date>'2023-01-01' AND K.date<'2023-05-01'
    GROUP BY M2(M3(K.imageID));
```

In Q1, $M1$ denotes a weather classification model, $M2$ is a vehicle classification model used to classify the types of vehicles in the image and $M3$ is a vehicle detection model to determine whether there are any vehicles present in the image. It can greatly facilitate data management if the in-DB DL model can perform inference without the need to transfer data to a separate DL platform. However, implementing in-DB DL systems on terminal units with constrained resources, such as low-end CPUs and limited memory, presents challenges in terms of computation and storage.

To support such multi-modal queries, loosely-coupled solutions, such as MADlib [32], Microsoft MLS [4], MindsDB [5], and SQLFlow [8], only require minor changes to the DBMS and are suitable for devices with sufficient resources. However, unlike personal computers or servers, edge devices or embedding units often have limited resources, making it hard to support conventional full-fledged platforms (e.g., TensorFlow Serving [55] and TorchServe [9]). Despite these challenges, in-DB DL systems remain an attractive option for many applications.

Another recent approach is the *tightly-coupled solution*, where the DL models are stored as relational tables, the computations of DL operations are converted into relational algebra operations, and the analytical queries are fully processed inside the database [39, 47, 58]. However, the relational data model and relational algebra operations are not well-suited to handle large tensors. Empirical results reported in [47] suggest that the tightly-coupled solution may result in sub-optimal performance compared to the loosely-coupled approach, especially when computation overhead dominates the total cost. In fact, typical neural operators such as convolution, max pooling, average pooling, and full connection are reported to be 30% to 70% slower than their corresponding implementations in PyTorch (more details in Section 5.4).

Proceedings of the VLDB Endowment, Vol. 17, No. 3 ISSN 2150-8097.
doi:10.14778/3632093.3632095

Our aim is to optimize the tightly-coupled solution's benefits, enabling DBMS to support individual model inferences without GPUs or powerful hardware while closing the performance gap with DL platforms. To achieve it, we present the **SmartLite**, which extends existing DBMSs to support DL inference tasks under resource-constrained settings. We demonstrate how to model neural operators in a more concise and database-friendly way, and how to perform neural network computations inside a relational database in an efficient and resource-saving manner. We propose three major designs in SmartLite.

The first design we propose is ***DBMS-oriented parameter-driven model simplification***. We observe that slow computation (i.e., model inference) and high resource consumption primarily result from floating-point calculations, which CPUs are not optimized for. A trained neural network typically contains millions to billions of floating-point parameters that represent pre-trained weights. The computation involving these large float tensors is generally very slow in a relational database [25, 26, 53]. To reduce the cost of computation and storage, we propose to simplify the floating-point parameters using the *neural network quantization* technique. Specifically, SmartLite applies the quantization to convert floating-point numbers to binary numbers. Meanwhile, SmartLite transforms the model operations on the floating-point numbers into bit-level operations such as BitCount or XOR. Finally, SmartLite establishes lookup tables to accelerate the combination of bit-level operators.

The second design we propose is ***model structure optimization***. After reducing the cost of floating-point computations, we find that the performance bottleneck has shifted to a large number of bit-level computations when performing inferences on binarized networks [23, 44, 45, 60]. In practice, a model gradually converges in the last few epochs, during which most parameter values remain unchanged, while some frequently flip between 0 and 1. This suggests that, under the current neural network structure, those parameters may not contribute to the inference performance of the model. Therefore, we define a probabilistic model to describe the effectiveness of parameters in a given bit-length and prune the corresponding bit-level computation. It allows SmartLite to discard many lookup operations when they have a negligible effect on the inference accuracy, thus optimizing the neural network structure.

The third design is ***in-DB model sharing and scheduling***. As many IoT applications deploy multiple DL models on a single edge device, we explore organizing multiple in-DB models within a running SmartLite instance. Notably, many models follow a pre-training and fine-tuning paradigm. Consequently, we can map a relational table with pre-trained parameter values into logical blocks, allowing reuse of these blocks for various fine-tuned models derived from the same pre-trained model, thereby reducing data redundancy. In resource-constrained settings lacking sufficient physical memory to hold all in-DB models, we introduce a scheduling algorithm to determine which parts of the used models to load into memory. The primary goal of the scheduling algorithm is to minimize the swapping cost incurred during in-DB model serving.

In summary, we make the following contributions: (1) We present a lightweight system, SmartLite, that reduces the computational complexity of neural network operators inside the DBMS. SmartLite supports serving multiple models simultaneously under the resource-constrained setting. (2) SmartLite applies binarized quantization to reduce the parameter size of a trained neural network and compresses the parameters at the bit-level. (3) SmartLite maps the computation of matrix multiplication into table lookup, a more friendly operation for the database, discards the redundant data while training, and reuses the pre-trained parameter blocks in its memory. (4) We experimentally demonstrate that SmartLite achieves 134% speedup compared to the mainstream ML framework PyTorch for serving multiple neural models. The size of the model storage is 98% smaller than PyTorch.

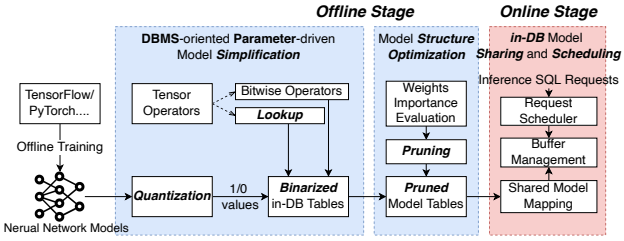

Figure 1: Main modules of SmartLite.

This paper is organized as follows. We present the architecture of SmartLite in Section 2 and introduce our approaches in Section 3. We discuss how SmartLite serves inferences of multiple models in Section 4. The experimental performance of different implementations is evaluated in Section 5. We review the related work in Section 6 and conclude the paper in Section 7.

2 SMARTLITE: OVERVIEW

In this section, we give an overview of SmartLite and briefly introduce the workflow of our in-DB model inference process. Note that SmartLite is not designed to support in-DB model training due to the limited computational resources of the edge devices. We leave the resource-intensive training process to DL frameworks on the cloud server. SmartLite does not rely on any assumption of the underlying DBMSs. However, tensor manipulations involved in DL inference require column-wise processing, and hence, for performance considerations, current SmartLite is implemented on top of ClickHouse [2], a columnar DBMS.

As shown in Figure 1, the processing is split into the offline stage and the online stage. In the offline stage, we export the trained model from Tensorflow/PyTorch into SmartLite and then apply the quantization to transform the float parameters into 0-1 bit parameters, significantly reducing the size of the neural model. Both the model structure information and model parameters are maintained as tables of SmartLite. Correspondingly, the tensor operators, such as tensor multiplication, can be implemented as bitwise operators, which are well supported by modern DBMS. To further improve the inference performance, we combine bottleneck operators, such as BitCount and XOR, into a specific lookup operation (see Section 3 for details), dramatically reducing the computation complexity. Note that in our experiments, the quantization may cause a performance drop in accuracy of less than 6%. However, it can effectively reduce the computation overhead and enable real-time inference for devices with limited computation capability.[1]

[1]The evaluation results are presented in Section 5.6.

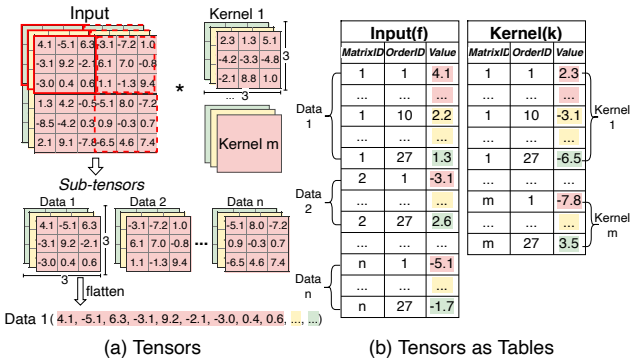

Figure 2: Tensors as tables: a simple solution.

In the offline process, we also inherit the idea of network pruning [14, 44, 45] to discard unnecessary parameters. Namely, if parameters do not show a strong correlation with the final prediction result, we can avoid the computation involving them. In this way, we sacrifice 1%-3% (tunable) accuracy to achieve a runtime improvement of over 200%. [2] In the online process, SmartLite receives real-time prediction requests for multiple models. However, due to a limited buffer, only a few of them can be fully maintained in memory. We observe that many neural models share similar neural structures and only refine the last few layers. Therefore, SmartLite identifies the shared parts of models and creates a mapping table to avoid buffer redundancy of neural parameters. To avoid repetitively shuffling data in/out of memory, a scheduling algorithm is employed to maximize the likelihood that requests can be fully processed in memory. Figure 1 illustrates our key three optimization techniques that are applied to the offline and online stages. In what follows, we delve into the details of the two stages and explain how we support in-DB real-time inference with constrained resources.

3 OFFLINE PROCESSING STAGE

In the offline stage, we import the trained neural models (parameters and neural structures) into SmartLite. A straightforward storage approach is introduced to maintain tensors as tables and correspondingly, we transform tensor computations into SQL queries. To reduce the computation and storage overhead, we apply two optimization techniques: DBMS-oriented parameter-driven model simplification and model structure optimization.

3.1 Tensors as Tables

Tensors are the main citizens of neural network computations. In SmartLite, both input requests and neural models are represented as tensors. Therefore, we first discuss how to maintain tensors in a relational database.

3.1.1 Storage. In a typical convolutional neural network (CNN), both the input tensor and the weight tensor (or kernel tensor in the convolution layer) usually have four dimensions: $N \times H \times W \times C$, representing batch size, height, width, and channels, respectively. A straightforward idea is to transform these tensors into multiple one-dimensional arrays and maintain them as one-column tables. Tensor multiplication, also known as MatMul, is considered the most fundamental and costly operation in CNN [11, 62]. It is composed of dot product operations between multiple sub-tensors. To represent the MatMul in the database, we transform the sub-tensors that perform the dot product operations, rather than the entire tensors, into multiple one-dimensional arrays and store them as tables.

Figure 2a shows a specific case of MatMul, where it is repetitively invoked to compute the convolution. Each convolution operation can be seen as a dot product operation between sub-tensors. We divide the tensors into multiple sub-tensors based on how we calculate the dot product calculation. In Figure 2a, a sub-tensor is a tensor of dimensions $1 \times 3 \times 3 \times 3$ that satisfies the compatibility condition for MatMul with the kernel tensor. To denote the structure of sub-tensors, auxiliary indices can be added to the tables as indicators. For instance, as demonstrated in Figure 2b, the input has n sub-tensors and there are m kernel tensors. To preserve this structure of the neural network and locate the relevant elements involved in a MatMul computation, we generate MatrixID and OrderID in the table. MatrixID is used to mark the sub-tensor IDs and OrderID is used to mark the order of the elements in the sub-tensors. This results in an input table with $n * 3 * 3 * 3$ tuples and a kernel with $3 * 3 * 3 * m$ tuples.

3.1.2 Computation. After we maintain tensors as tables, we can leverage the DBMS to manipulate them. In fact, SQL is a very powerful coding language and most tensor manipulations, such as MatMul, SUM, MIN/MAX, and Count, can be directly processed via SQL queries. We use MatMul as an example to illustrate how SQL can be applied to process tensors and hence, perform the model inference. As mentioned in the previous storage, in the calculation process of MatMul, the value of each position in the result matrix is actually the dot product between sub-tensors. Essentially, the MatMul operation of the jth sub-tensors can be denoted as follow:

$$MatMul(F_j, K_j) = \sum_{i=0}^{size(F)} (F_j[i] * K_j[i]) \quad (1)$$

where $F_j[i]$ and $K_j[i]$ are the ith element of the sub-tensor F_j and the sub-tensor K_j, respectively.

The storage of F_j and K_j in the database is illustrated in Figure 2b. To achieve MatMul by the SQL, we establish connections between elements sharing the same OrderID through INNER JOIN. The associated sub-tensors F_j and K_j are then aggregated using GROUP BY with matching MatrixIDs and subjected to the MatMul operation. The SQL statement can be described as follows:

```
SELECT SUM(F.value * K.value)
FROM F INNER JOIN K
ON F.OrderID = K.OrderID
GROUP BY F.MatrixID, K.MatrixID;
```

Other popular neural network operators based on tensor manipulations can be translated into SQL queries as well and we illustrate the idea in the following sections. The straightforward approach, although simple and intuitive, is not efficient for the DBMS, which is designed to support relational data, not arrays. Therefore, we propose two optimizations to make it more compatible with DBMS.

[2] The tuning knob for accuracy/runtime is presented in Section 5.5.5.

3.2 Parameter-driven Model Simplification

3.2.1 Parameters Quantization And Collective Bit-Storage. Default neural models are trained with float weights, incurring high computation and storage overheads. To address the issue, we apply neural network quantization[19, 69] to simplify neural models in the offline stage. We apply XNOR-Net [26, 53, 56, 66] to binarize a neural model with only 0 and 1 as parameter values. Figure 3 shows a binary storage table of the input table in Figure 2. We use one bit-array as a tuple to represent all binary values of a sub-tensor because of its compact representation. Auxiliary indices are not needed in this case, since we can perform 1-to-1 mapping. Thus, tensor manipulations over the binary tables can be implemented as bitwise operators of the DBMS on corresponding bit-arrays.

We propose a new bit-oriented storage format for neural model layers, which overcomes the limitation of DBMS in supporting bit-arrays of varying lengths. The proposed format is depicted in Figure 4, where a bit-oriented table file is created for each layer. Data in each storage file is partitioned into equal-sized (4KB) data blocks, each comprising a 32-bit header and a set of 256-bit data chunks. The header comprises 16 bits for the width/height of the tensor and 16 bits for the number of data chunks. In the data blocks corresponding to each neural model layer, the data chunks constitute all the tensors required for computations. The data of the tensor is stored in data chunks sequentially. Furthermore, these data chunks are organized as the columns of binary tables within the DBMS. During the computational process, SmartLite executes queries for different neural model layers by parsing and locating the positions of the relevant data blocks in the storage files and subsequently retrieving the contents of the headers and data chunks. The tensor format is explained below:

Typical Weight Tensor Let $M_w = w \times h \times c$ denote the weight tensor. M_w can be split into $\lceil \frac{whc}{256} \rceil$ chunks. A large tensor may span several data blocks and if there is extra space left, SmartLite uses "0" as the padding bit.

Kernel Tensor Unlike other weight tensors, the kernel tensor is usually compact and has equal width and height. Let κ and c denote the kernel width and the number of channels, respectively. The kernel tensor has $\kappa \times \kappa \times c$ bits and SmartLite splits it into $\kappa \times \kappa$ bit-arrays. A 256-bit data chunk can store up to $\lfloor \frac{256}{\kappa^2} \rfloor$ bit-arrays.

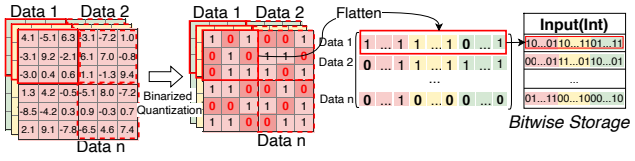

Figure 3: Binary storage table.

The proposed SmartLite maintains two types of tables for a neural model. A meta table records all layers of the model and how they are connected, while the pure data table represents the weight tensors of a layer. Since tensors need to be executed in a certain order during the calculation process, we have strict requirements for the data layout and order in the storage data table. With the help of header information, SmartLite can locate all bit-arrays of either a typical weight tensor or a kernel tensor quickly. The binary table structure allows for bitwise operations on any tensor, such as bitwise XOR and AND, without any parsing or transforming overhead, as demonstrated in Section 3.2.2.

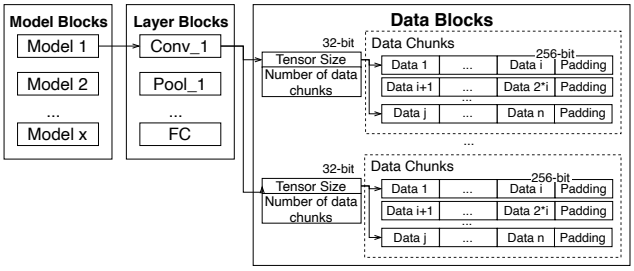

Figure 4: : Binary storage file for neural models.

3.2.2 Computation over Binary Table. By transforming a tensor into a binary table, we can perform operations on the tensors as bitwise operations on the tables. We have:

$$MatMul(F, K) \approx \beta\alpha MatMul(f, k) \quad s.t.\ F, K \in \mathbb{R}^n,\ f, k \in \{0, 1\}^n \tag{2}$$

where f and k are the binarized versions of F and K generated by the XNOR-Net, respectively. β and α are the scale factors and are determined during the training from XNOR-Net. The theoretical analysis of Equation 2 on binarization is discussed in [56]. Note that the error difference between before and after binary quantization is very large, so the scaling factors need to be introduced for the corresponding correction. We can use the offline trained scaling factor as the known parameters. So we discard them from our discussion. Putting the bitwise operations into Equation 2, $MatMul(f, k)$ can be rewritten by the following equation [21]:

$$MatMul(f, k) = -2 * BitCount(XOR(f, k)) + size(f) \tag{3}$$

where *BitCount* returns the number of 1s in a bit-array. *XOR* is the function of exclusive bitwise *OR* and *size(f)* returns the tensor size. Using tensor mathematical operations, SmartLite can support various neural network modules (such as Max Pooling, Average Pooling, Batch Normalization, ReLU, Sign, tanh, etc.) in a binarized manner via bitwise operations such as *BitCount* and *XOR*. For example, Convolution, Full Connection, and MLP are implemented as MatMul while Max Pooling, Average Pooling, tanh, and Sign are single-tensor transformations. ReLU is a value filter. We have provided the SQL statements for implementing these neural modules in Table 1, omitting details for the sake of brevity. The Basic Block of ResNet is the most complex module we demonstrate, requiring a combination of Convolution, ReLU, and module connections.

3.2.3 Performance Optimization. After profiling, it has been discovered that the *BitCount* function is the most expensive and frequently used operation. This function counts the number of 1s in a bit-array, and scanning the whole data incurs a cost of $O(len)$ (where len is the length of the bit-array), which can make it the most time-consuming part in model inference. To address this issue, SmartLite

Table 1: The SQL statement of neural network operators.

Operators	SQL Statement	Description [3]
MatMul	SELECT -2*BitCount(XOR(f.value,k.value)) +size(XOR(f.value,k.value)) FROM Input f INNER JOIN Kernel k;	Refer to Section 3.1.
Convolution(Conv)		Refer to Section 3.1.
Full Connection(FC)		$y = fk^T$
MLP		$\{FC_1, FC_1, ..., FC_n\}$
Max Pooling	SELECT BitCount(f.value)>0?1:0 FROM Input f;	$max(f)$
Average Pooling	SELECT BitCount(f.value)/size(f.value) FROM Input f;	$avg(f)$
Batch Normalization	SELECT (f.value-(SELECT AVG(f.value) FROM Input f)) / (SELECT sqrt(varPop(f.value)+0.00001) FROM Input f) FROM Input f;	$\frac{f-avg(f)}{\sqrt{var(f)+0.00001}}$
ReLU	SELECT f.value>0?f.value:0 FROM Input f;	$max(0, f)$
Sign	SELECT sign(f.value) FROM Input f;	$sgn(f)$
tanh	SELECT (exp(f.value)-exp(-f.value)) / (exp(f.value)+exp(-f.value)) FROM Input f;	$\frac{exp(f)-exp(-f)}{exp(f)+exp(-f)}$
Basic Block in ResNet [31]	SELECT f".value+f.value FROM(SELECT (-2*BitCount(XOR(f.value,k.value)) +size(XOR(f.value,k.value))) AS value FROM (SELECT value>0?value:0 AS value FROM (SELECT (-2*BitCount(XOR(f.value,k.value)) + size(XOR(f.value,k.value))) AS value FROM Input f INNER JOIN Kernel1 K)) AS f INNER JOIN Kernel2 k) as f", Input f;	$\{Conv_1, Conv_2, ReLU, FC\}$

accelerates the *BitCount* computation using lookup tables, which is explained in this subsection.

A possible way to speed up the processing of enumerable input combinations is to compute and store the corresponding results in a lookup table offline and search the lookup table during online processing, reducing the cost from $O(len)$ to $O(1)$ [52]. We notice that SmartLite applies *BitCount* normally after multiplying two bit-arrays f and k in Table 1. So we extend the lookup scheme. For two bit-arrays of equal-size len, SmartLite maintains a lookup table LP with 2^{2*len} elements. For example, let "10110000" and "11100111" be the binary representations of f and k, respectively. Then, $f \times k$ produces a bit-array "10100000", and the concatenation of "10110000" and "11100111" (i.e., 10110000⊕11100111) gives "1011000011100111" which is 45287 (decimal value of 1011000011100111). Therefore, SmartLite has an entry $BitCount[45287] = 2$ (number of 1s in "10100000") in its lookup table LP. We maintain all possible results in LP for future calculation of the bitwise operations. If SmartLite receives the query $BitCount(10110000, 11100111)$ again, it returns 2 directly by checking the LP table.

The approach can be further extended to support more complex combinations of multiple bitwise operations. For example, SmartLite can merge the *BitCount* lookup entry with the *XOR* operator to create a new lookup entry, denoting the result of *BitCount(XOR())*. For the same two bit-arrays as our previous example, their *XOR* result is "01010111", and SmartLite makes a new entry $BitCount(XOR[45287]) = 5$ corresponding to the result of *BitCount* of "01010111". In this way, one lookup completes the operation $BitCount(XOR(f, k))$, which reduces the processing overhead. Formally, the lookup table of $BitCount(XOR[b(f), b(k)])$ returns the *BitCount* results for $XOR(b(f), b(k))$, where $b(f)$ and $b(k)$ are the binary representation of the bit-array f and k, respectively.

However, the above lookup table is not scalable for some cases that involve two extremely large tensors. For example, the last full connection layer of VGG16 [61] has $len = 4096$. It is impossible to maintain a lookup table of size 2^{8192} in SmartLite. To solve this issue, SmartLite splits $b(f)$ into $x = \lceil \frac{len}{m} \rceil$ equal-size bit-arrays: $b_1(f), \ldots, b_x(f)$, where $b_i(f)$ has a length of m. For any two bit-arrays f and k, both the computed results of *BitCount* and *XOR* over the whole bit-arrays are equal to the sum of computed results of *BitCount* and *XOR* over the sub-bit-arrays. That is, SmartLite can get the results of $BitCount(b(f) \times b(k))$ and $BitCount(XOR(b(f), b(k)))$ from the split bit-arrays. Formally, we have:

$$BitCount(b(f) \times b(k)) = \sum_{i=1}^{x=\lceil \frac{len}{m} \rceil} BitCount(b_i(f) \times b_i(k)) \quad (4)$$

$$BitCount(XOR(b(f), b(k))) = \sum_{i=1}^{x=\lceil \frac{len}{m} \rceil} BitCount(XOR(b_i(f), b_i(k)) \quad (5)$$

In this way, SmartLite can perform one large table lookup operation as x small ones. Moreover, SmartLite only needs to keep a lookup table of size 2^{2m} for all bit-arrays because they share the same table if they have the same length.

One special case is the convolution layer, where input tensors are multiplied with some fixed kernels iteratively. Since only a limited number of kernels are involved, SmartLite does not need to maintain all possible combinations. Instead, we adopt an "inverted" lookup approach. For each kernel k_i, SmartLite creates a lookup table LP^i. Given an input with the binary representation "1011000", $LP^i[176]$ returns the result of $BitCount(XOR(10110000, b(k_i)))$. Formally, we can initialize LP for a convolution layer as follows:

$$LP^i[j] = BitCount(XOR(b(j), b(k_i))) \quad (6)$$

where $b(j)$ represents the binary array of integer j. Then, we can convert all computations involving k_i into a lookup of LP^i. Suppose a convolution layer has n_k kernels. SmartLite only needs to maintain n_k lookup tables. In this way, SmartLite dramatically reduces the sizes of the lookup tables from 2^{2*len} to $n_k \times 2^{len}$. For example, for a convolution layer in VGG16 with 128 kernels of size 3×3, SmartLite only needs to keep 128 lookup tables of size 2^9, which costs about 64KB in total storage. Different from one big lookup table for all layers, the inverted kernel lookup tables are associated with each convolution layer. In other words, we can maintain multiple sets of the lookup tables, one for each convolution layer.

3.3 Model Structure Optimization

After transforming the *BitCount* and *BitCount(XOR())* operations into efficient memory lookups, the performance of SmartLite during neural inference is mainly determined by the number of lookups. Existing works [14, 50, 57, 60] show that some neural structures are redundant and do not contribute to the inference performance. Hence, network pruning is applied to compress neural models, effectively reducing the processing overhead. During the training process, we find that some weights frequently flip from 0 to 1 and vice versa in the last few epochs, indicating that the model cannot generate a consistent value for those weights [44]. In these epochs, the accuracy of the model improves gradually (if not over-fitting). This observation suggests that the values of those weights do not

[3]The more specific definitions of neural operators and the implementations of PyTorch functions can be obtained from: https://pytorch.org/docs/stable/nn.html

contribute to the performance of the model. As such, SmartLite may remove those weights without hurting the accuracy during the neural model inference.

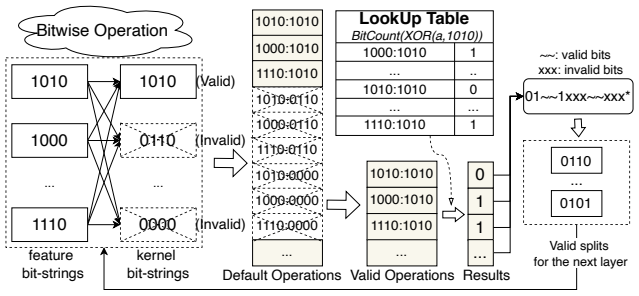

Figure 5: Workflow of neural network pruning.

Figure 5 illustrates the method employed by SmartLite, which involves dividing input and kernel tensors into m-bit splits and utilizing a lookup table to replace costly BitCount operations. SmartLite maintains a frequency histogram of each bit in the kernel, indicating the number of times the bit flips during training. This histogram, which is a by-product of the training process, can be obtained by analyzing the training log. We define the threshold θ as the trade-off between the computation overhead and accuracy. If the bit flips for more than θ number of epochs, we mark it as "invalid". Existing works [27, 45, 65] demonstrate the locality property of neural network weights. Effective weights are grouped together and can be partitioned into a finite number of clusters. Similarly, weights that do not contribute to the model inference are also clustered, i.e., the whole bit-array. This observation suggests that if all bits in the split bit-array are invalid, we can mark the whole bit-array as invalid.

In Figure 5, we ignore any splits that are marked as "invalid" and omit any intermediate results involving invalid splits during MatMul computation. By doing so, we effectively decrease the number of lookups required during MatMul. The resulting tensors are then forwarded to the next layer as input to continue the cascading pruning, further reducing the overhead.

Pruning performance analysis. The neural network pruning is determined by two parameters, m and θ. m is the length of the split bit-array. The pruning threshold θ is used as the tuning knob to adjust the model's runtime and accuracy. We demonstrate the effect of θ on the pruning ratio and accuracy in Section 5.5.5. Here, we briefly discuss the effect of m. A larger m results in fewer lookup operations, but leads to a lower probability of detecting an invalid split. We use CNN and fully connected network (FCN) as our examples to illustrate the effect. Other neural structures follow a similar analysis.

Let P denote the probability of a bit being invalid. We apply the beta-binomial distribution (BBD) to model the distribution of P and the correlations between bits. Therefore, we estimate the probability of all m bits being invalid as:

$$P(x = m, \alpha, \beta) = \frac{\Gamma(m+\alpha)\Gamma(\alpha+\beta)}{\Gamma(m+\alpha+\beta)\Gamma(\alpha)} \tag{7}$$

where α and β are parameters of the beta distribution. Γ denotes the gamma function.

Let f_{in}, f_{out}, and n_w denote the sizes of the input tensor, the output tensor, and the kernel tensor, respectively. The number of invalid splits is estimated as $\frac{n_w}{m}P(x = m, \alpha, \beta)$. For FCN, we have:

$$f_{out} = \lceil \frac{f_{in}}{m} \rceil \lceil \frac{n_w}{m} \rceil P(x = m, \alpha, \beta) \times m^2 \tag{8}$$

For CNN, the size of the kernel is $\kappa \times \kappa$. Suppose we have $f_{in} = h \times h \times c$ and $f_{out} = h' \times h' \times c'$. The number of expected channels can be denoted as follows [44]:

$$c' = \lceil \frac{n_w}{\kappa^2} \rceil P(x = m, \alpha, \beta) \tag{9}$$

The width/height of f_{out} is estimated as:

$$h' = \lfloor \frac{h - \kappa + 2p}{s} \rfloor + 1 \tag{10}$$

where p and s represent the padding and stride of the CNN [51].

$P(x = m, \alpha, \beta)$ can be estimated using the sampling approach and is monotonically decreasing in terms of m. To achieve the optimal pruning effect, we can obtain the m to minimize f_{out}:

$$f'_{out} = 0 \tag{11}$$

In a real-world scenario, a neural network normally consists of various types of neural layers (e.g., Convolution+ReLU+BN), making it difficult to estimate a global optimal m. Instead, we choose to generate an optimal m for each individual layer according to the sampling results during the model training, although it may result in a sub-optimal solution for the whole network.

4 ONLINE INFERENCE STAGE

One advantage of SmartLite is that it can simultaneously support multiple models in a lightweight manner, compared to conventional ML/DL frameworks such as TensorFlow Serving and TorchServe. Tensors of the pre-trained models are maintained as database tables in SmartLite, which uses the memory mapping strategy to reduce the I/O overhead. This allows SmartLite to serve multiple neural models and provides the capability of real-time inferences on edge devices. There are two main optimizations applied: model sharing and model scheduling.

4.1 In-DB Model Mapping and Sharing

We observe that a widely adopted strategy for building neural models is "transfer and refine". A basic model pre-trained with large datasets is reused as shared blocks by keeping all weights except the last few layers unchanged. We only re-train the last 2-3 layers for predictions as private blocks. For instance, VGG is a common basic model that can be refined to support tasks such as classification [48], object detection [30], and style transfer [64]. This approach motivates our model-sharing strategy.

Figure 6 illustrates the model sharing in SmartLite, where all data blocks are mapped to a memory buffer and the LRU strategy is applied to select a victim block if the buffer is full. SmartLite maintains a meta table for each model, recording how its parameters are stored. When serving a model, we first check the meta table to retrieve the weights of the next layer. If the corresponding data blocks are buffered in memory, we initiate the computation directly. Otherwise, the lazy loading process similar to the demand-paging mechanism is invoked, which only loads data blocks if necessary.

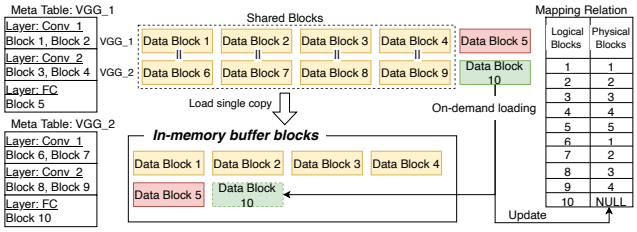

Figure 6: Model sharing with mapped memory.

To keep track of how blocks are buffered, we maintain a mapping table for logical blocks to physical blocks. The request of accessing a data block from the serving process is translated into operations on a physical block. There are three possible access patterns. First, we have already set up the mapping relation and the request is transformed into memory access. Second, if we find that the mapped physical blocks are identical (e.g., block 1 and block 6 in Figure 6) because the corresponding models share the same basic structure, we modify the mapping and refer to the same physical block, reducing the redundancy. Finally, if the mapping is not established, we load a data block from the disk into the buffer. If the buffer is already full, the LRU algorithm is applied to pick a victim and replace it with a new block. Last but not least, the related records in the mapping table are updated as well.

4.2 Model Scheduling

In a heavily loaded scenario, the process of adaptive loading can result in frequent migration of data blocks between memory and disk. This problem can be formally defined as follows.

Let N denote the maximum number of buffered blocks and M denote the number of pre-trained neural network models. We define the set of models as $S_\chi = \chi_i = (b_1, \ldots, b_{g(i)}) \mid 0 \le i < M$, where model χ_i is comprised of blocks b_1 to $b_{g(i)}$. Let T^i represent the buffer status at time i, such that $T^0(b_j) = 1$ if block b_j is present in memory at time i, and 0 otherwise. The buffer imposes a limit on the number of the buffered blocks, such that $\sum_{\forall j} T^i(b_j) \le N$. At time $i+1$, if the buffer has been updated, the cost is estimated as:

$$C_{i \to i+1} = \sum_{\forall j} |T^{i+1}(b_j) - T^i(b_j)| \tag{12}$$

Our target is to generate an inference sequence that achieves the least loading costs:

$$\underset{\forall \pi}{\text{argmin}} \left(\sum_{i=0}^{M-1} C_{\pi(i) \to \pi(i+1)} \right) \tag{13}$$

π represents a permutation of the model order, indicating the sequential buffering of model blocks. As there are exponential possibilities, this problem can be considered a special case of the online knapsack problem with removable cost [28, 34, 35], particularly in the real scenario where requests are received in a streaming manner. N represents the budget, and each model χ_i is considered an item with weight $g(i)$ and benefit r_i, where r_i is the number of requests for model χ_i. If the buffer is full and new requests come in, we must remove some models from the buffer to accommodate the new ones, incurring a removable cost $C_{i \to i+1}$. The problem is further complicated by the fact that some models share the same data blocks, resulting in dynamically changing model weights. It is worth noting that there exists no deterministic online algorithm with a constant competitive ratio for the general removable online knapsack problem.

Algorithm 1: Online scheduling strategy.

Input: requests $R = \{(\chi_i, r_i) | 0 \le i < M\}$ at time t
r_i is the number of the requests for model χ_i
Let Hashmap $H(G_x)$ return all requests of models in G_x
for $x = 0$ *to* $L-1$ **do**
  $H(G_x) += \sum_{\chi_i \in G_x} R[\chi_i].r_i$
  $v_x = \frac{H(G_x)}{E^t(G_x)}$
end
Let G_{max} be the group with max v_x
if G_{max} *is not maintained in memory* **then**
  Remove groups with minimal v_x from memory until all models in G_{max} can be held in memory
  update the corresponding mapping table T
end
Process requests of models in G_{max}
$H(G_x) = 0$

We extend the greedy algorithm of [29]. The main idea is to process requests for a group of models sharing the same basic structure simultaneously. Suppose we have L basic neural structures, all models are grouped based on their neural structures : $\{G_0, ..., G_{L-1}\}$. For each model χ, we split its data blocks into two categories: $B^\chi_{basic} = \{b_1, ..., b_x\}$ and $B^\chi_{tune} = \{b'_1, ..., b'_y\}$. The set B^χ_{basic} is shared between all models with the same basic structure, while B^χ_{tune} denotes χ's private blocks. We define $\bar{T}^i(G_j)$ to return whether any model of G_j is maintained in the buffer at time i. We further define the cost of group G_j at time i as follows:

$$E^i(G_j) = \begin{cases} B^{\chi_0}_{basic} + \sum_{\forall \chi \in G_j} (B^\chi_{tune}) & \text{if } \bar{T}^{i-1}(G_j) == 0 \\ \sum_{\forall \chi \in G_j} (B^\chi_{tune}) & \text{otherwise} \end{cases} \tag{14}$$

where χ_0 is a random model in G_j. Then, our idea is presented in Algorithm 1. The algorithm first counts the number of requests for all basic neural structures at time t and then evaluates the benefit by dividing the number of requests by the cost. For the neural group G_j with the maximal benefit, SmartLite schedules it with the highest priority and serves all requests for models in the group.

5 EVALUATIONS

5.1 Experimental Setup

Hardware. To evaluate the performance of SmartLite, we conduct experiments on an edge device equipped with an ARMv8 CPU (AArch84, 8 cores @ 1.8GHz) and 16GB of memory. This type of CPU is commonly used in embedded systems due to its low energy consumption and compact size.

Metrics. We measure the performance of SmartLite using six metrics: the maximum number of models that can be loaded into memory, total inference runtime of a batch (in seconds), prediction

Table 2: Overall evaluation of different models on edge devices.

Devices	Models	Metrics	PyTorch	MADlib	DL2SQL	Default SmartLite	SmartLite(lookup)	SmartLite(prune)	Maximum Improvement
AArch64	VGG16	*Time(s)*	12.573	21.326	45.296	22.412	15.291	**10.888**	4.16×
		Memory Usage(MiB)	1197.751	1045.802	3177.058	950.441	953.328	**655.984**	4.84×
		Accuracy(%)	90.37%	90.37%	90.37%	90.37%	**90.37%**	89.67%	-0.70%
	ResNet18	*Time(s)*	4.827	10.274	15.785	12.544	10.324	**4.471**	3.53×
		Memory Usage(MiB)	236.587	378.304	987.461	167.029	170.453	**136.558**	7.23×
		Accuracy(%)	87.22%	87.22%	87.22%	87.22%	**87.22%**	84.61%	-2.61%
	AlexNet	*Time(s)*	6.053	20.796	21.146	13.373	9.836	**5.205**	4.06×
		Memory Usage(MiB)	302.935	807.856	1321.375	287.103	289.605	**215.855**	6.12×
		Accuracy(%)	88.84%	88.84%	88.84%	88.84%	**88.84%**	86.73%	-2.11%
	TextCNN	*Time(s)*	0.528	12.939	3.008	0.551	0.363	**0.302**	42.84×
		Memory Usage(MiB)	62.648	199.128	744.894	44.864	45.082	**30.628**	24.32×
		Accuracy(%)	74.86%	74.86%	74.86%	74.86%	**74.86%**	72.14%	-2.72%
X86_64	VGG16	*Time(s)*	9.865	34.793	41.316	17.644	11.559	**7.352**	5.62×
		Memory Usage(MiB)	1142.921	1146.011	2509.782	541.449	544.566	**322.523**	7.78×
	ResNet18	*Time(s)*	5.441	17.884	24.719	11.908	6.801	**4.781**	5.17×
		Memory Usage(MiB)	571.622	611.352	934.367	91.316	92.488	**64.269**	14.54×
	AlexNet	*Time(s)*	8.596	27.241	29.599	19.875	10.341	**5.802**	5.10×
		Memory Usage(MiB)	697.276	772.032	1078.144	165.075	166.539	**92.961**	11.60×
	TextCNN	*Time(s)*	1.206	24.411	2.382	0.323	0.295	**0.165**	147.95×
		Memory Usage(MiB)	34.148	270.683	506.757	25.898	27.546	**11.402**	44.44×

accuracy (as a percentage), memory usage (in MiB), read/write rate (in MiB/s), and throughput (in rows/s). The default batch size used in our experiments is 64.

Threads. By default, we use the maximum number of threads (10) for metric measurements in all experiments except for the one that compares the effect of varied numbers of threads.

Compared Methods. For comparison purposes, we use PyTorch (including TorchServe), a popular DL framework, as a baseline implementation. We also compare our approach to a loosely-coupled solution called MADlib, which supports in-DB inference with the support of TensorFlow. Additionally, we utilize DL2SQL [47] as a direct comparison to the tightly-coupled approach. DL2SQL translates the neural operators into SQL queries and relies on the DBMS to optimize the queries. To maintain tensors in relational tables, DL2SQL adopts two strategies: Column-oriented Matrix Multiplication (CoMM) and Multi-Column based Matrix Multiplication (MCMM). CoMM follows the same storage and computation strategy as shown in Figure 2b. MCMM is similar to CoMM, except that it aligns all values of different channels in one row.

Data and Query Sets. We train and evaluate the performance of SmartLite by training 200 models using VGG16, ResNet18, AlexNet, and TextCNN as the basic neural structures and four widely adopted datasets: Mnist [6], CIFAR10 [1], ImageNet [3], and MR [7], respectively. Of these, Mnist, CIFAR10, and ImageNet are image datasets and MR is a text dataset. We target four tasks: image classification, object detection, text classification, and text sentiment analysis. And for each task, we randomly slice a portion of the corresponding dataset, train a model offline, and export it into SmartLite for online inference. We repeat the process and train 50 models for each structure. Inference queries, normally denoted as (d, m_i, t_j), are randomly generated. Here, d represents a piece of data from the test sets of the four datasets, while m_i and t_j represent the labels of base models and task types. Given m_i and t_j, we can obtain the candidate model to process the request. It is worth noting that for inference, d is translated into a tensor based on its pixel values and maintained as the table in SmartLite.

5.2 Overall Performance

In these experiments, we aim to demonstrate the general performance of all different approaches. To show the effectiveness of different optimization approaches of SmartLite, we create three versions. The default SmartLite does not employ the lookup and pruning optimizations. SmartLite(lookup) only adopts the lookup transformation optimization, and SmartLite(prune) applies both techniques. To ensure a fair comparison, all approaches, including PyTorch, MADlib, and DL2SQL, report their results after applying binarized quantization on neural networks. The effect of binarized quantization will be shown individually in Table 3.

In addition to the results on the AArch64-based device, we include results on an X86_64 server (10 cores @ 2.2GHz, 20GB memory) for comparison purposes. In the last column of Table 2, we add the maximum improvement of SmartLite (prune) compared to the worst performances to facilitate intuitive analysis.

Table 2 shows that during the inference process for all models, SmartLite (prune) exhibits significant advantages for model inference compared with other all implementations, at the cost of a small decrease in accuracy. As shown in the last column of Table 2, SmartLite (prune) achieves a 4× to 100× improvement in prediction latency and memory consumption compared to existing solutions, while resulting in an accuracy degradation of only 0.7%, 2.61%, 2.11%, and 2.72% for VGG16, ResNet18, AlexNet, and TextCNN, respectively. The result trends on the AArch64 and X86_64 are consistent, demonstrating the stability of our approach across different hardware settings. We also observe that SmartLite (lookup) significantly reduces inference time compared to SmartLite (default), albeit with a small increase in memory consumption due to the lookup tables. Moreover, our model pruning strategy in SmartLite (prune) effectively reduces memory consumption and inference time compared to SmartLite (lookup).

5.3 Performance of Multi-model Serving

One advantage of SmartLite is its capability of hosting multiple models with constrained resources. We evaluate the performances of

multi-model serving in different hardware settings using AArch64 and X86_64 and present the results in Figure 7a-d. Results of DL2SQL are not included because of its long inference time, which makes it not feasible for real-time predictions.

Figure 7a displays the memory usage difference among the three methods on the AArch64-based device. We observe that MADlib's memory usage is more than 43 times that of SmartLite, making it a highly resource-intensive approach. Due to the limitation of thread quantity, the growth trend of MADlib's memory usage slows down after reaching a certain upper limit, but the running time increases significantly. PyTorch exhibits a similar situation. The memory usage of PyTorch is more than 50 times that of SmartLite. For the corresponding number of models, on average, SmartLite's execution speed is 1.21 times and 1.34 times that of MADlib and PyTorch, respectively, as shown in Figure 7b.

Figure 7c and 7d illustrate the performance on the X86_64-based device. Figure 7c shows the memory usage of three implementations for serving the same number of models. MADlib and PyTorch use 37 times and 25 times more memory than SmartLite, respectively. The difference in memory usage increases as the number of loaded models increases until the thread quantity reaches the limit. Our model scheduling and sharing strategy saves memory significantly. Figure 7d compares the inference performance of MADlib, PyTorch, and SmartLite for multi-model serving. SmartLite is faster than MADlib and PyTorch in inference speed, with up to 3.21× and 3.17× speedup for the same number of loaded models. The speed gap increases as the number of models increases.

Figure 7e illustrates the maximum number of models that MADlib, PyTorch, and SmartLite can load with different memory sizes. We can see that MADlib and PyTorch consume larger memory. In contrast, SmartLite can load more models into memory, scaling up to 28 times more than MADlib and PyTorch for the 8G memory size. Meanwhile, SmartLite can control the number of buffered models by utilizing memory size as a tuning knob.

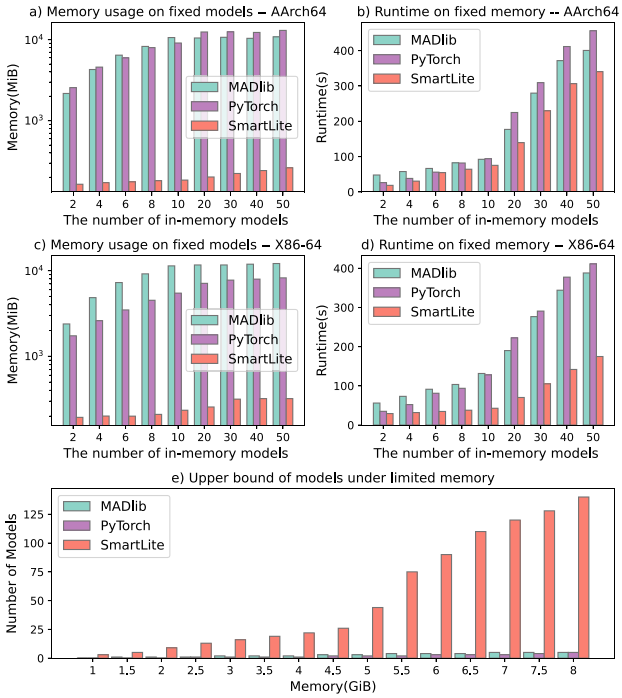

Figure 7: Performance comparisons of multi-model serving.

5.4 Effect of Binarized Quantization

We compare the performance of the floating-point models and the quantized models using PyTorch and the database. We compare the running time of SmartLite with other implementations and use the speed-up ratio as a measure. We evaluate four operators: convolution, maximum pooling, average pooling, and full connection used by the ResNet18. We fix the batch size of input data to 16. Due to the extremely low overhead of pooling one batch, we cannot measure its performance accurately enough with our database. Therefore, for Mnist and ImageNet, we measure the time for 500 batches of data to pass through the pooling layer and the full connection layer. For CIFAR10, the number of batches for the pooling operator is 500, and for the full connection operator, it is 10,000.

Table 3 presents the results of the performance of various processing methods. The findings indicate that SmartLite outperforms other implementations by achieving a 9.25× speedup for the convolutional operator. Moreover, SmartLite accelerates the maximum pooling operator up to 20.27× and achieves a 13.47× faster performance on average-pooling than other implementations. For the full connection, SmartLite demonstrates a remarkable 26× performance improvement. As expected, binarized operations are generally faster than floating-point operations. However, the analysis reveals that the database provides better support for binarized operations than floating-point ones. Hence, the advantage of binarized computing is more suitable for the database. Notably, the migration from floating-point to binarized computing results in a much larger performance improvement for the database than PyTorch.

5.5 Detailed Analysis of Different Modules

5.5.1 Different Neural Operators. We evaluate the performance of SmartLite regarding both the entire neural network and frequently-used neural network operators. Specifically, we evaluate the convolutional operator, the fully connected operator, the max-pooling operator, and the average-pooling operator of ResNet18 on the CIFAR10 dataset. To evaluate the convolutional operator, we set the output channel to 80 and the kernel size to $3 \times 3 \times 3$.

Figure 8 demonstrates the impact of different quantized inference implementations on query processing time. The horizontal axis represents the batch size, and the different curves represent various implementations. The vertical axis represents the query processing time. The results indicate that SmartLite, optimized by the lookup operation, achieves a performance improvement of up to 34× on the convolutional operator. Specifically, compared to PyTorch, SmartLite runs, on average, 303% faster. These findings suggest that our proposed implementation of bitwise operation has an advantage in terms of data computation parallelism. Despite the complexity of the full connection computation, SmartLite still outperforms other methods in terms of runtime. In particular, SmartLite is up to 63× faster than CoMM. Moreover, SmartLite is

Table 3: Comparison of different processing: PyTorch-float is the floating-point inference on PyTorch; DL2SQL is the floating-point inference in database; PyTorch-binary is the quantized inference on PyTorch.

Datasets	Implements	Convolution		Max Pooling		Average Pooling		Full Connection	
		Time(s)	*Acceleration*	*Time(s)*	*Acceleration*	*Time(s)*	*Acceleration*	*Time(s)*	*Acceleration*
Mnist	PyTorch-float	0.021	1.75×	0.063	7.00×	0.041	1.58×	0.015	3.00×
	DL2SQL	0.065	5.42×	0.057	6.33×	0.231	8.88×	0.031	6.20×
	PyTorch-binary	0.016	1.33×	0.054	6.00×	0.035	1.35×	0.012	2.40×
	SmartLite	**0.012**	**1.00×**	**0.009**	**1.00×**	**0.026**	**1.00×**	**0.005**	**1.00×**
CIFAR10	PyTorch-float	0.037	9.25×	0.027	13.5×	0.017	2.83×	0.155	25.83×
	DL2SQL	0.032	8.00×	0.018	9.00×	0.043	7.17×	0.081	13.50×
	PyTorch-binary	0.009	2.25×	0.019	9.50×	0.015	2.50×	0.009	1.50×
	SmartLite	**0.004**	**1.00×**	**0.002**	**1.00×**	**0.006**	**1.00×**	**0.006**	**1.00×**
ImageNet	PyTorch-float	0.015	3.75×	0.136	12.36×	0.121	3.18×	0.021	3.00×
	DL2SQL	0.018	4.50×	0.223	20.27×	0.512	13.47×	0.036	5.14×
	PyTorch-binary	0.009	2.25×	0.124	11.27×	0.096	2.53×	0.018	2.57×
	SmartLite	**0.004**	**1.00×**	**0.011**	**1.00×**	**0.038**	**1.00×**	**0.007**	**1.00×**

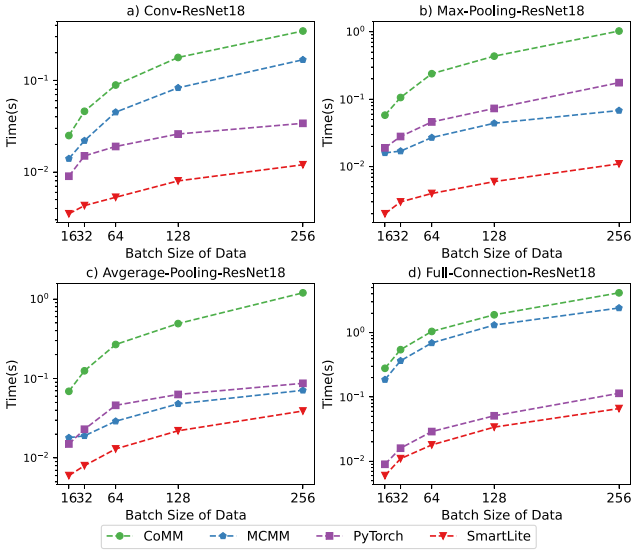

Figure 8: Running time of different implementations.

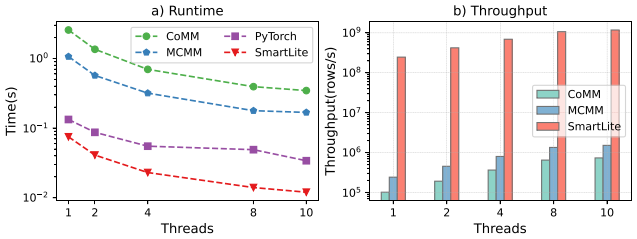

Figure 9: Effect of the number of threads.

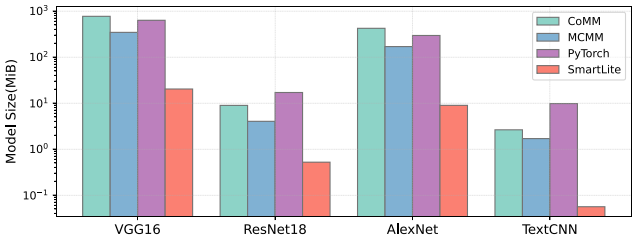

Figure 10: Model size comparison.

16× faster than PyTorch in max-pooling and 353% faster in average-pooling. These results demonstrate the effectiveness and efficiency of SmartLite in various neural network operators.

5.5.2 Thread Scalability. To assess the stability of various quantized inference methods in the database, we vary the number of threads and measure the runtime and throughput of the convolutional operator implemented by different methods. PyTorch is used as the baseline for runtime comparison. Throughput is defined as the number of rows written to the table per second. Figure 9a depicts the thread scalability of various quantized inference methods on CIFAR10. Among the different methods, SmartLite performs the best under various thread settings, demonstrating the effectiveness of bitwise operation optimization. Specifically, when the number of threads increases from 1 to 10, SmartLite reduces its runtime by 625% compared to PyTorch, indicating its efficient utilization of thread resources. Throughput is one of the criteria for measuring database performance. Figure 9b demonstrates that the throughput of different methods increases as the number of threads increases. Notably, the throughput of SmartLite is 1609× higher than that of CoMM, the simplest implementation, when the number of threads is 10. The results indicate that SQL statements of SmartLite are more conducive to leveraging the vectorized execution of the database.

5.5.3 Model Size. As shown in Figure 10, SmartLite achieves a significantly lower model size than the other methods. For VGG16, SmartLite reduces the model size by 38×, 17×, and 31× compared to CoMM, MCMM, and PyTorch, respectively. Similarly, for ResNet18, SmartLite compresses the model size by about 17×, 8×, and 32× compared to CoMM, MCMM, and PyTorch, respectively. The model size

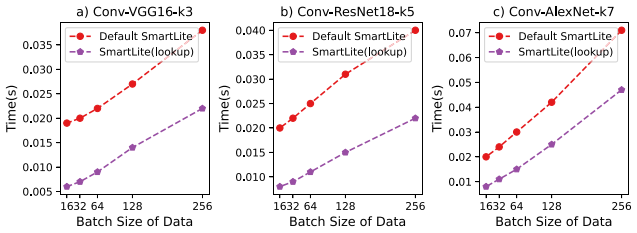

Figure 11: Effect of lookups.

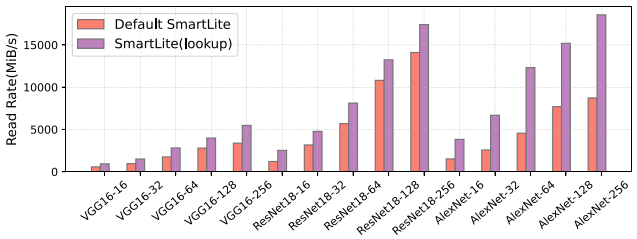

Figure 12: Read performance of convolution operator.

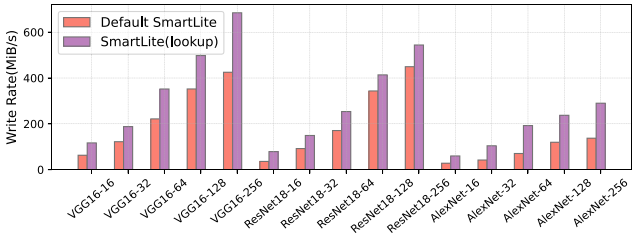

Figure 13: Write performance of convolution operator.

of AlexNet in SmartLite is 33× smaller than that in PyTorch. Compared to CoMM and MCMM, SmartLite shrinks the size of AlexNet by 47× and 19×, respectively. Furthermore, TextCNN stored in SmartLite is 47×, 30×, and 174× smaller than those stored in CoMM, MCMM, and PyTorch, respectively. These results demonstrate that SmartLite is efficient in utilizing limited storage resources.

5.5.4 Lookup Table for Bitwise Count. We use SmartLite with the default bitwise operation (*Default SmartLite*) as the baseline method and compare it with SmartLite based on the lookup operation (*SmartLite(lookup)*). Our implementation of SmartLite is conducted on the CIFAR10, using the VGG16, ResNet18, and AlexNet networks. We select one kernel from each of these three networks to ensure a fair comparison. The input size is fixed at $3 \times 32 \times 32$, while the kernel size of VGG16, ResNet18, and AlexNet are $3 \times 3 \times 3$, $3 \times 5 \times 5$, and $3 \times 7 \times 7$, respectively. We denote these kernels as $conv - vgg16 - k3$, $conv - ResNet18 - k5$, and $conv - AlexNet - k7$, where $conv$ indicates the operator type, $vgg16$ indicates the network type and $k3$ indicates the convolutional kernel size.

As depicted in Figure 11, our experiments demonstrate that SmartLite with the lookup operation achieves a significant speedup compared to the default bitwise operation. Specifically, with a 3×3 convolutional kernel, SmartLite with the lookup operation achieves a speedup of up to 2.17×. For a 5×5 convolutional kernel, the lookup operation reduces the running time by 50% compared to the original bitwise operation. Moreover, with a 7×7 convolutional kernel, the lookup operation achieves a maximum 176% acceleration in the query speed than that of the original SmartLite. Our experimental results also indicate that a larger batch size of the input results in a greater advantage of the optimization, which demonstrates the stability of the lookup operation. Next, we compare the read rates and the write rates between different convolutional kernel sizes and batch sizes. Figures 12 and 13 present the results, where the number at the end of each horizontal label denotes the batch size, e.g., 32 in VGG16-32. Across all scenarios, *SmartLite(lookup)* outperforms all other methods in terms of speed. Specifically, for the read rate, the advantage of *SmartLite(lookup)* becomes more pronounced as the batch size increases, reaching a maximum of 212% of the *Default SmartLite* method. Regarding the write rate, *SmartLite(lookup)* also demonstrates a significant performance improvement, reaching 211% of *Default SmartLite* when the batch size is 256.

5.5.5 Network Pruning in SmartLite. We study the effects of two tuning parameters: pruning threshold θ and bit-array length m. Specifically, we utilize the CIFAR10 dataset and ResNet18 in our experiments. In addition, users can control the pruning ratio by adjusting the pruning threshold while keeping the bit-array length m fixed, thus adjusting the trade-off between running time and accuracy. As illustrated in Figure 14, the accuracy exhibits a similar trend across different m as the pruning ratio increases. Moreover, An increase in the pruning threshold θ leads to a decrease in the pruning ratio. The impact of pruning threshold θ on the pruning ratio is more pronounced for smaller bit-array length m.

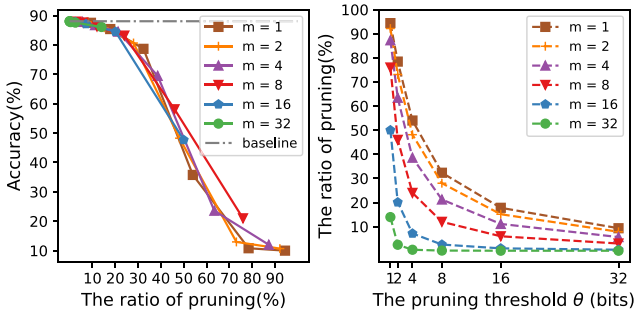

Figure 14: Effect of the tunable parameters.

5.6 Effect on Accuracy

We also design experiments to evaluate the accuracy of in-DB quantized models. We select four neural networks, namely VGG16, ResNet18, AlexNet, and TextCNN, and compute their Top-1 classification accuracy in each epoch. Specifically, we test VGG16 on Mnist, ResNet18 on CIFAR10, AlexNet on ImageNet, and TextCNN on MR, respectively. Figure 15 shows the evaluation results of the floating-point models and the quantized models on the four datasets. The results indicate that for VGG16, the quantized model reduces the training accuracy by 1.65% and lowers the testing accuracy by

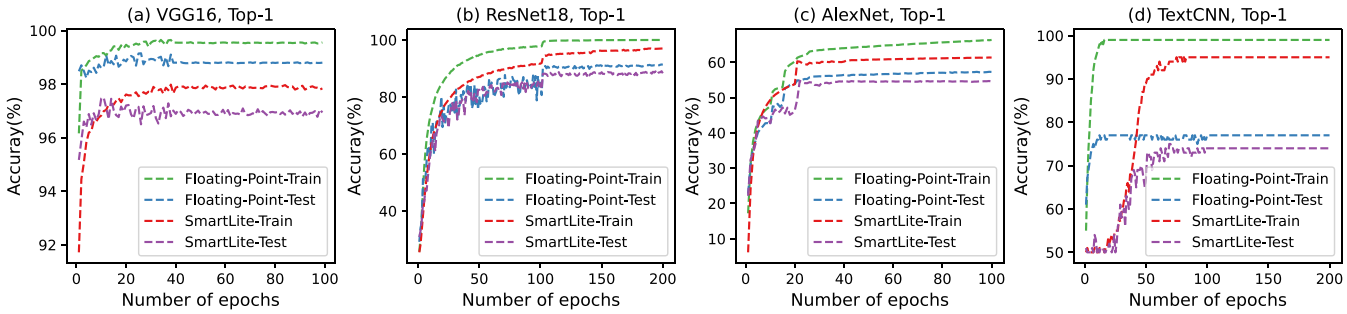

Figure 15: Accuracy comparison between the floating-point neural network and the quantized neural network.

1.87%. For ResNet18, the training accuracy drops by 2.97%, and the testing accuracy decreases by 3.11%. For TextCNN, neural network quantization brings about a 3.98% drop in training accuracy and a 3.03% drop in test accuracy. Even for AlexNet, which has the largest dataset, the training accuracy difference is 5.03%, while the testing accuracy difference is 3.76%. As shown in Figure 15, quantization leads to some loss of accuracy. This is an inevitable consequence since quantization reduces the precision of computation. However, given the memory saving and speedup in inference, this trade-off can be acceptable for resource-constrained devices.

6 RELATED WORK

In this section, we briefly introduce related research lines.

Neural Network Quantization: Neural network quantization is to reduce the memory and computation consumption of neural networks while keeping their accuracy as much as possible. Courbariaux et al. [20] find that very low precision multipliers are sufficient to train neural networks. Courbariaux et al. [21] introduce a method to train binarized neural networks by computing the gradients of parameters with binarized weights and activations. Rastegari et al. [56] propose Binary-Weight-Networks and XNOR-Net. The multiplication of XNOR-Net is implemented by XNOR operation with a scaling factor. Banner et al. [12] quantize model parameters, activations, and layer gradients to 8-bit and propose Range Batch-Normalization to tolerate higher noise. They also propose a 4-bit post-training quantization approach [13] to avoid retraining CNN models. ZeroQ [16] proposes a zero-shot quantization framework that uses mixed-precision quantization. ZAQ [49] takes a novel two-level discrepancy modeling and optimizes the quantized model with adversarial learning. Considering different statistical properties of neural gradients, Chmiel et al. [18] optimize the floating-point format and scale of the gradients and set sparsity thresholds for gradient pruning.

Optimization for database and models: Directly supporting DL in a database normally does not result in satisfactory performance. BitWeaving [43] utilizes bit-level parallelism in modern processors to speed up scanning in main memory. ROVEC [42] avoids unnecessary datatype casting and utilizes block-wise statistics to improve SIMD-based evaluation parallelism. Jiang et al. [37] speed up query execution by filtering unnecessary encoded data and vectorizing execution based on SIMD. These methods are not designed to optimize the computation of in-DB model inference. As for neural network implementation, BNN Pruning [44] prunes the binarized neural network by using weight flipping frequency to judge whether the weight is important for accuracy. Jiang et al. [45] prunes weights by randomly assigning values to high-dimensional bit-sliced data and comparing accuracy differences. But these studies cannot reduce bit-array computation in a database. SLIDE [17, 22] enables neural networks to be trained on CPUs efficiently through parallelism and LSH based sparsification. Unlike SLIDE, which uses hash lookups to obtain active neurons for training, SmartLite obtains computation results for inference directly through one lookup table.

In-DB ML/DL: In-DB ML training and inference becomes a hot topic in recent years. MadLib [32], Vertica-ML [24], CORE [67], Zebra [68], and DB4ML [36] combine the deployment of neural networks and ML in a database through UDFs. These approaches are easy to deploy. Different from these approaches, DL2SQL [47], AC/DC [39], and LMFAO [58] implement ML/DL by aggregation queries. For specific unstructured datasets queries, tspDB [10] is a real-time time series prediction system integrating with PostgreSQL. Focus [33] indexes video by using a specialized neural network to detect objects and clusters similar objects to avoid redundant queries. CORE [67] packages proxy models into UDFs and accelerates inference by exploiting predicate correlation. These in-DB ML/DL approaches do not actually reduce the computational complexity of the model. Differently, SmartLite considers both the storage cost and computational complexity of the model itself, which greatly reduces the overhead of model inference.

7 CONCLUSION

In this paper, we introduce SmartLite, a lightweight DBMS designed to efficiently process tasks associated with neural network models. SmartLite turns complex neural networks into binary-valued relational tables and converts the computation over neural networks to relational operations. As such, SmartLite reduces memory usage and maintains accurate results over limited resources. Several optimization techniques are devised to speed up the in-DB model computation, including lookup tables, block-sharing, and multi-model scheduling based on shared costs. Our experiments demonstrate that SmartLite performs well on various metrics.

ACKNOWLEDGMENTS

This work was supported by the Key Research Program of Zhejiang Province (Grant No. 2023C01037).

REFERENCES

[1] 2023. CIFAR10. Retrieved November 16, 2023 from https://www.cs.toronto.edu/~kriz/cifar.html

[2] 2023. ClickHouse. Retrieved November 16, 2023 from https://clickhouse.com/

[3] 2023. ImageNet. Retrieved November 16, 2023 from https://www.image-net.org/

[4] 2023. Microsoft SQL Server Machine Learning Services. Retrieved November 16, 2023 from https://microsoft.github.io/sql-ml-tutorials

[5] 2023. MindsDB. Retrieved November 16, 2023 from https://mindsdb.com

[6] 2023. Mnist. Retrieved November 16, 2023 from http://yann.lecun.com/exdb/mnist/

[7] 2023. MR. Retrieved November 16, 2023 from https://www.cs.cornell.edu/people/pabo/movie-review-data/

[8] 2023. SQLFlow. Retrieved November 16, 2023 from https://sql-machine-learning.github.io

[9] 2023. TorchServe. Retrieved November 16, 2023 from https://github.com/pytorch/serve

[10] Anish Agarwal, Abdullah Alomar, and Devavrat Shah. 2020. tspDB: Time Series Predict DB. In *NeurIPS 2020 Competition and Demonstration Track, 6-12 December 2020, Virtual Event / Vancouver, BC, Canada (Proceedings of Machine Learning Research)*, Hugo Jair Escalante and Katja Hofmann (Eds.), Vol. 133. PMLR, 27–56. http://proceedings.mlr.press/v133/agarwal21a.html

[11] Andrew Anderson, Aravind Vasudevan, Cormac Keane, and David Gregg. 2017. Low-memory GEMM-based convolution algorithms for deep neural networks. *CoRR* abs/1709.03395 (2017). arXiv:1709.03395 http://arxiv.org/abs/1709.03395

[12] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. 2018. Scalable methods for 8-bit training of neural networks. *Advances in neural information processing systems* 31 (2018).

[13] Ron Banner, Yury Nahshan, and Daniel Soudry. 2019. Post training 4-bit quantization of convolutional networks for rapid-deployment. *Advances in Neural Information Processing Systems* 32 (2019).

[14] Davis W. Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John V. Guttag. 2020. What is the State of Neural Network Pruning?. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze (Eds.). mlsys.org. https://proceedings.mlsys.org/book/296.pdf

[15] Beatriz Blanco-Filgueira, Daniel García-Lesta, Mauro Fernández-Sanjurjo, Víctor M. Brea, and Paula López. 2019. Deep Learning-Based Multiple Object Visual Tracking on Embedded System for IoT and Mobile Edge Computing Applications. *IEEE Internet Things J.* 6, 3 (2019), 5423–5431. https://doi.org/10.1109/JIOT.2019.2902141

[16] Yaohui Cai, Zhewei Yao, Zhen Dong, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. 2020. ZeroQ: A Novel Zero Shot Quantization Framework. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[17] Beidi Chen, Tharun Medini, James Farwell, Sameh Gobriel, Tsung-Yuan Charlie Tai, and Anshumali Shrivastava. 2020. SLIDE : In Defense of Smart Algorithms over Hardware Acceleration for Large-Scale Deep Learning Systems. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze (Eds.). mlsys.org. https://proceedings.mlsys.org/book/306.pdf

[18] Brian Chmiel, Liad Ben-Uri, Moran Shkolnik, Elad Hoffer, Ron Banner, and Daniel Soudry. 2020. Neural gradients are near-lognormal: improved quantized and sparse training. *arXiv preprint arXiv:2006.08173* (2020).

[19] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett (Eds.). 3123–3131. https://proceedings.neurips.cc/paper/2015/hash/3e15cc11f979ed25912dff5b0669f2cd-Abstract.html

[20] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2014. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024* (2014).

[21] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830* (2016).

[22] Shabnam Daghaghi, Nicholas Meisburger, Mengnan Zhao, and Anshumali Shrivastava. 2021. Accelerating SLIDE Deep Learning on Modern CPUs: Vectorization, Quantizations, Memory Optimizations, and More. In *Proceedings of Machine Learning and Systems 2021, MLSys 2021, virtual, April 5-9, 2021*, Alex Smola, Alex Dimakis, and Ion Stoica (Eds.). mlsys.org. https://proceedings.mlsys.org/paper/2021/hash/3636638817772e42b59d74cff571fbb3-Abstract.html

[23] Tim Dettmers and Luke Zettlemoyer. 2019. Sparse Networks from Scratch: Faster Training without Losing Performance. *CoRR* abs/1907.04840 (2019). arXiv:1907.04840 http://arxiv.org/abs/1907.04840

[24] Arash Fard, Anh Le, George Larionov, Waqas Dhillon, and Chuck Bear. 2020. Vertica-ML: Distributed Machine Learning in Vertica Database. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 755–768. https://doi.org/10.1145/3318464.3386137

[25] Giorgia Fiscaletti, Marco Speziali, Luca Stornaiuolo, Marco D. Santambrogio, and Donatella Sciuto. 2020. BNNsplit: Binarized Neural Networks for embedded distributed FPGA-based computing systems. In *2020 Design, Automation & Test in Europe Conference & Exhibition, DATE 2020, Grenoble, France, March 9-13, 2020*. IEEE, 975–978. https://doi.org/10.23919/DATE48585.2020.9116220

[26] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. 2021. A Survey of Quantization Methods for Efficient Neural Network Inference. *CoRR* abs/2103.13630 (2021). arXiv:2103.13630 https://arxiv.org/abs/2103.13630

[27] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating Large-Scale Inference with Anisotropic Vector Quantization. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event (Proceedings of Machine Learning Research)*, Vol. 119. PMLR, 3887–3896. http://proceedings.mlr.press/v119/guo20h.html

[28] Xin Han, Yasushi Kawase, and Kazuhisa Makino. 2014. Online Unweighted Knapsack Problem with Removal Cost. *Algorithmica* 70, 1 (2014), 76–91. https://doi.org/10.1007/s00453-013-9822-z

[29] Xin Han, Yasushi Kawase, and Kazuhisa Makino. 2015. Randomized algorithms for online knapsack problems. *Theor. Comput. Sci.* 562 (2015), 395–405. https://doi.org/10.1016/j.tcs.2014.10.017

[30] Md Foysal Haque, Hye-Youn Lim, and Dae-Seong Kang. 2019. Object Detection Based on VGG with ResNet Network. In *International Conference on Electronics, Information, and Communication, ICEIC 2019, Auckland, New Zealand, January 22-25, 2019*. IEEE, 1–3. https://doi.org/10.23919/ELINFOCOM.2019.8706476

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 770–778. https://doi.org/10.1109/CVPR.2016.90

[32] Joseph M. Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. 2012. The MADlib Analytics Library or MAD Skills, the SQL. *Proc. VLDB Endow.* 5, 12 (2012), 1700–1711. https://doi.org/10.14778/2367502.2367510

[33] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodík, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. 2018. Focus: Querying Large Video Datasets with Low Latency and Low Cost. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 269–286. https://www.usenix.org/conference/osdi18/presentation/hsieh

[34] Sungjin Im, Ravi Kumar, Mahshid Montazer Qaem, and Manish Purohit. 2021. Online Knapsack with Frequency Predictions. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan (Eds.). 2733–2743. https://proceedings.neurips.cc/paper/2021/hash/161c5c5ad51fcc884157890511b3c8b0-Abstract.html

[35] Kazuo Iwama and Guochuan Zhang. 2007. Optimal Resource Augmentations for Online Knapsack. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, 10th International Workshop, APPROX 2007, and 11th International Workshop, RANDOM 2007, Princeton, NJ, USA, August 20-22, 2007, Proceedings (Lecture Notes in Computer Science)*, Moses Charikar, Klaus Jansen, Omer Reingold, and José D. P. Rolim (Eds.), Vol. 4627. Springer, 180–188. https://doi.org/10.1007/978-3-540-74208-1_13

[36] Matthias Jasny, Tobias Ziegler, Tim Kraska, Uwe Röhm, and Carsten Binnig. 2020. DB4ML - An In-Memory Database Kernel with Machine Learning Support. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 159–173. https://doi.org/10.1145/3318464.3380575

[37] Hao Jiang and Aaron J. Elmore. 2018. Boosting Data Filtering on Columnar Encoding with SIMD. In *Proceedings of the 14th International Workshop on Data Management on New Hardware* (Houston, Texas) *(DAMON '18)*. Association for Computing Machinery, New York, NY, USA, Article 6, 10 pages. https://doi.org/10.1145/3211922.3211932

[38] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. NoScope: Optimizing Deep CNN-Based Queries over Video Streams at Scale. *Proc. VLDB Endow.* 10, 11 (2017), 1586–1597. https://doi.org/10.14778/3137628.3137664

[39] Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2018. AC/DC: In-Database Learning Thunderstruck. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning, DEEM@SIGMOD 2018, Houston, TX, USA, June 15, 2018*, Sebastian Schelter, Stephan Seufert, and Arun Kumar (Eds.). ACM, 8:1–8:10. https://doi.

org/10.1145/3209889.3209896

[40] Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2020. Learning Models over Relational Data Using Sparse Tensors and Functional Dependencies. *ACM Trans. Database Syst.* 45, 2 (2020), 7:1–7:66. https://doi.org/10.1145/3375661

[41] He Li, Kaoru Ota, and Mianxiong Dong. 2018. Learning IoT in Edge: Deep Learning for the Internet of Things with Edge Computing. *IEEE Netw.* 32, 1 (2018), 96–101. https://doi.org/10.1109/MNET.2018.1700202

[42] Meng Li, Zheyu Miao, Di Wu, Feifei Li, Sheng Wang, Wei Cao, Zhi Qiao, Bin Yu Ruan, Kun Yu Liang, Jun Xin Yang, Haipeng Dai, and Guihai Chen. 2021. ROVEC: Runtime Optimization of Vectorized Expression Evaluation for Column Store. *IEEE Transactions on Knowledge and Data Engineering* (2021), 1–1. https://doi.org/10.1109/TKDE.2021.3124669

[43] Yinan Li and Jignesh M. Patel. 2013. BitWeaving: fast scans for main memory data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 289–300. https://doi.org/10.1145/2463676.2465322

[44] Yixing Li and Fengbo Ren. 2020. BNN Pruning: Pruning Binary Neural Network Guided by Weight Flipping Frequency. In *21st International Symposium on Quality Electronic Design, ISQED 2020, Santa Clara, CA, USA, March 25-26, 2020*. IEEE, 306–311. https://doi.org/10.1109/ISQED48828.2020.9136977

[45] Yixing Li, Shuai Zhang, Xichuan Zhou, and Fengbo Ren. 2020. Build a compact binary neural network through bit-level sensitivity and data pruning. *Neurocomputing* 398 (2020), 45–54. https://doi.org/10.1016/j.neucom.2020.02.012

[46] Ji Lin, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han. 2020. MCUNet: Tiny Deep Learning on IoT Devices. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/86c51678350f656dcc7f490a43946ee5-Abstract.html

[47] Qiuru Lin, Sai Wu, Junbo Zhao, Jian Dai, Feifei Li, and Gang Chen. 2022. A Comparative Study of in-Database Inference Approaches. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 1794–1807. https://doi.org/10.1109/ICDE53745.2022.00180

[48] Shuying Liu and Weihong Deng. 2015. Very deep convolutional neural network based image classification using small training sample size. In *3rd IAPR Asian Conference on Pattern Recognition, ACPR 2015, Kuala Lumpur, Malaysia, November 3-6, 2015*. IEEE, 730–734. https://doi.org/10.1109/ACPR.2015.7486599

[49] Yuang Liu, Wei Zhang, and Jun Wang. 2021. Zero-Shot Adversarial Quantization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 1512–1521.

[50] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. 2017. ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*. IEEE Computer Society, 5068–5076. https://doi.org/10.1109/ICCV.2017.541

[51] Wei Ma and Jun Lu. 2017. An Equivalence of Fully Connected Layer and Convolutional Layer. *CoRR* abs/1712.01252 (2017). arXiv:1712.01252 http://arxiv.org/abs/1712.01252

[52] Kinga Marton, Bolba Raluca, and Alin Suciu. 2017. Counting bits in parallel. In *2017 16th RoEduNet Conference: Networking in Education and Research (RoEduNet), Tg. Mures, Romania, September 21-23, 2017*. IEEE, 1–6. https://doi.org/10.1109/ROEDUNET.2017.8123743

[53] Bradley McDanel, Surat Teerapittayanon, and H. T. Kung. 2017. Embedded Binarized Neural Networks. In *Proceedings of the 2017 International Conference on Embedded Wireless Systems and Networks, EWSN 2017, Uppsala, Sweden, February 20-22, 2017*, Per Gunningberg, Thiemo Voigt, Luca Mottola, and Chenyang Lu (Eds.). Junction Publishing, Canada / ACM, 168–173. http://dl.acm.org/citation.cfm?id=3108031

[54] Mehdi Mohammadi, Ala I. Al-Fuqaha, Sameh Sorour, and Mohsen Guizani. 2018. Deep Learning for IoT Big Data and Streaming Analytics: A Survey. *IEEE Commun. Surv. Tutorials* 20, 4 (2018), 2923–2960. https://doi.org/10.1109/COMST.2018.2844341

[55] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. TensorFlow-Serving: Flexible, High-Performance ML Serving. *CoRR* abs/1712.06139 (2017). arXiv:1712.06139 http://arxiv.org/abs/1712.06139

[56] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*. Springer, 525–542.

[57] Alex Renda, Jonathan Frankle, and Michael Carbin. 2020. Comparing Rewinding and Fine-tuning in Neural Network Pruning. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=S1gSj0NKvB

[58] Maximilian Schleich and Dan Olteanu. 2020. LMFAO: An Engine for Batches of Group-By Aggregates. *Proc. VLDB Endow.* 13, 12 (2020), 2945–2948. https://doi.org/10.14778/3415478.3415515

[59] Maximilian Schleich, Dan Olteanu, Mahmoud Abo Khamis, Hung Q. Ngo, and XuanLong Nguyen. 2019. Learning Models over Relational Data: A Brief Tutorial. In *Scalable Uncertainty Management - 13th International Conference, SUM 2019, Compiègne, France, December 16-18, 2019, Proceedings (Lecture Notes in Computer Science)*, Nahla Ben Amor, Benjamin Quost, and Martin Theobald (Eds.), Vol. 11940. Springer, 423–432. https://doi.org/10.1007/978-3-030-35514-2_32

[60] Yujun Shi, Li Yuan, Yunpeng Chen, and Jiashi Feng. 2021. Continual Learning via Bit-Level Information Preserving. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021, virtual, June 19-25, 2021*. Computer Vision Foundation / IEEE, 16674–16683. https://doi.org/10.1109/CVPR46437.2021.01640

[61] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1409.1556

[62] Paul Springer and Paolo Bientinesi. 2018. Design of a High-Performance GEMM-like Tensor-Tensor Multiplication. *ACM Trans. Math. Softw.* 44, 3 (2018), 28:1–28:29. https://doi.org/10.1145/3157733

[63] Matthias Urban and Carsten Binnig. 2023. Towards Multi-Modal DBMSs for Seamless Querying of Texts and Tables. *CoRR* abs/2304.13559 (2023). https://doi.org/10.48550/arXiv.2304.13559 arXiv:2304.13559

[64] Pei Wang, Yijun Li, and Nuno Vasconcelos. 2021. Rethinking and Improving the Robustness of Image Style Transfer. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021, virtual, June 19-25, 2021*. Computer Vision Foundation / IEEE, 124–133. https://doi.org/10.1109/CVPR46437.2021.00019

[65] Yikai Wang, Yi Yang, Fuchun Sun, and Anbang Yao. 2021. Sub-bit Neural Networks: Learning to Compress and Accelerate Binary Neural Networks. In *2021 IEEE/CVF International Conference on Computer Vision, ICCV 2021, Montreal, QC, Canada, October 10-17, 2021*. IEEE, 5340–5349. https://doi.org/10.1109/ICCV48922.2021.00531

[66] Olivia Weng. 2021. Neural Network Quantization for Efficient Inference: A Survey. *CoRR* abs/2112.06126 (2021). arXiv:2112.06126 https://arxiv.org/abs/2112.06126

[67] Zhihui Yang, Zuozhi Wang, Yicong Huang, Yao Lu, Chen Li, and X. Sean Wang. 2022. Optimizing Machine Learning Inference Queries with Correlative Proxy Models. *Proc. VLDB Endow.* 15, 10 (2022), 2032–2044. https://www.vldb.org/pvldb/vol15/p2032-yang.pdf

[68] Tianhuan Yu, Zhenying He, Zhihui Yang, Fei Ye, Yuankai Fan, Yinan Jing, Kai Zhang, and X Sean Wang. 2022. Zebra: A novel method for optimizing text classification query in overload scenario. *World Wide Web* (2022), 1–27.

[69] Jianhao Zhang, Yingwei Pan, Ting Yao, He Zhao, and Tao Mei. 2019. daBNN: A Super Fast Inference Framework for Binary Neural Networks on ARM devices. In *Proceedings of the 27th ACM International Conference on Multimedia, MM 2019, Nice, France, October 21-25, 2019*, Laurent Amsaleg, Benoit Huet, Martha A. Larson, Guillaume Gravier, Hayley Hung, Chong-Wah Ngo, and Wei Tsang Ooi (Eds.). ACM, 2272–2275. https://doi.org/10.1145/3343031.3350534