Q-gym: An Equality Saturation Framework for DNN Inference Exploiting Weight Repetition

Cheng Fu

Hanxian Huang

Bram Wasti

Chris Cummins

Riyadh Baghdadi

Kim Hazelwood

Yuandong Tian

Jishen Zhao

Hugh Leather

cfu@ucsd.edu

hah008@ucsd.edu

bwasti@fb.com

cummins@fb.com

baghdadi@mit.edu

kimhazelwood@fb.com

yuandong@fb.com

jzhao@ucsd.edu

hleather@fb.com

UC San Diego

UC San Diego

UC San Diego

La Jolla, CA, USA

La Jolla, CA, USA

La Jolla, CA, USA

Menlo Park, CA, USA

Menlo Park, CA, USA

Menlo Park, CA, USA

Menlo Park, CA, USA

Menlo Park, CA, USA

Menlo Park, CA, USA

ABSTRACT

The high computation cost is one of the key bottlenecks for adopting deep neural networks (DNNs) in different hardware. When client data are sensitive, privacy-preserving DNN evaluation method, such as homomorphic encryptions (HE), shows even more computation cost. Prior works employed weight repetition in quantized neural networks to save the computation of convolutions by memorizing or arithmetic factorization. However, such methods fail to fully exploit the exponential search space from factorizing and reusing computation. We propose Q-gym, a DNN framework consisting of two components. First, we propose a compiler, which leverages equality saturation to generate computation expressions for convolutional layers with a significant reduction in the number of operations. Second, we integrate the computation expressions with various parallelization methods to accelerate DNN inference on different hardware. We also employ the efficient expressions to accelerate DNN inference under HE.

Extensive experiments show that Q-gym achieves 19.1% / 68.9% more operation reductions compared to SumMerge and original DNNs. Also, computation expressions from Q-gym contribute to 2.56× / 1.78× inference speedup on CPU / GPU compared to OneDNN and PyTorch GPU on average. For DNN evaluation under HE, Q-gym reduces the homomorphic operations by 2.47× / 1.30× relative to CryptoNet and FastCryptoNet for HE tasks with only 0.06% accuracy loss due to quantization.

CCS CONCEPTS

- Computing methodologies → Parallel algorithms.

ACM Reference Format:


1 INTRODUCTION

Deep Neural Networks (DNNs) are becoming the de-facto solutions for various computer vision tasks [14, 33]. However, due to the large computation and storage cost of convolutional layers, DNNs are difficult to be integrated with many computation environments, such as mobile devices. To mitigate the computation and storage constraints for efficient DNN deployment, mainstream approaches include Quantizing DNNs [66, 68, 70] and leveraging DNN sparsity [23, 35, 36].

One key characteristic of quantized DNNs (QNNs) is weight repetition, i.e., weights in different layers are repetitions of a small number of Q unique ones (e.g., {-0.18, 0, 0.18} and Q = 3). Leveraging such a regularity, a state-of-the-art approach SumMerge [43] reduces computation by two measures: i) Factorization – for example, factorizing a dot-product \( ax + by + az + aw \) to be \( a(x + z + w) + by \) and ii) Computation reuse – for instance, when computing two dot products \( r_0 = ax + ay + az + aw \) and \( r_1 = bx + ay + bz + bw \), SumMerge first calculates a \( \text{partial sum} \ e_0 = x + z + w \) and reuses it in the downstream computations \( r_0 = a(e_0 + ay), r_1 = b(e_0 + ay) \).

While SumMerge indeed reduces the computation, it may not fully exploit weight repetition. As shown in Figure 1, while SumMerge leverages partial sums, it cannot reuse \( \text{partial products} \) (e.g., \( by, cz \), leading to sub-optimal reduction. The underlying reason is straightforward: finding the best strategy to reduce the cost of an arithmetic expression is a combinatorial optimization problem, and the simple greedy heuristics leveraged by SumMerge may be insufficient.
In this paper, we propose Q-gym, a framework to accelerate DNN inference by fully exploiting weight repetition. Q-gym comes with two components: a compiler that finds smart ways to reduce addition and multiplication counts in DNN evaluation, and a set of downstream tasks that leverage the compiled output to reduce its wall-clock time.

Q-gym’s Compiler. To reuse both partial sum and product results, our compiler aggressively explores the search space by leveraging a data-structure called e-graph [15, 38] that represents a large number of equivalent expressions in a compact manner (Figure 2). Thanks to its compact representation, the search space is easier to navigate. To identify the optimal solution in the search space, our compiler performs three steps. (i) Following equality saturation [63], we expand the e-graph by repeatedly applying a set of rewrite rules (Table 2). (ii) Then, we extract the optimal computation expressions using integer linear programming (ILP) [52]. (iii) With large convolutional layers, e-graph expansion hardly saturates. In order to fully explore the search space with such layers, we employ a pulsed searching [27] algorithm, which iteratively applies the first two steps until convergence of the computation cost. In the simple example illustrated in Figure 1, our compiler identifies its optimal solution. For large convolution layers, Q-gym can still find more optimized solutions compared to the greedy-based method.

Another technique for reducing the cost of convolutions is Winograd [31] which uses fast FFT to reduce computation operations in convolutional layers. Although this can have a significant effect on the number of operations, it does not take advantage of weight reuse and so does not achieve optimal reduction.

In this paper, we propose Q-gym, a framework to accelerate DNN inference by fully exploiting weight repetition. Q-gym comes with two components: a compiler that finds smart ways to reduce addition and multiplication counts in DNN evaluation, and a set of downstream tasks that leverage the compiled output to reduce its wall-clock time.

Q-gym’s Compiler. To reuse both partial sum and product results, our compiler aggressively explores the search space by leveraging a data-structure called e-graph [15, 38] that represents a large number of equivalent expressions in a compact manner (Figure 2). Thanks to its compact representation, the search space is easier to navigate. To identify the optimal solution in the search space, our compiler performs three steps. (i) Following equality saturation [63], we expand the e-graph by repeatedly applying a set of rewrite rules (Table 2). (ii) Then, we extract the optimal computation expressions using integer linear programming (ILP) [52]. (iii) With large convolutional layers, e-graph expansion hardly saturates. In order to fully explore the search space with such layers, we employ a pulsed searching [27] algorithm, which iteratively applies the first two steps until convergence of the computation cost. In the simple example illustrated in Figure 1, our compiler identifies its optimal solution. For large convolution layers, Q-gym can still find more optimized solutions compared to the greedy-based method.

We also discover that the input activations overlap during the ‘sliding’ computation of weight kernels. That means computations can be reused between the sliding convolutions. With the proposed powerful searching algorithm, our compiler identifies a better solution by leveraging an extended search space compared with SumMerge [43] and Winograd [31].

Q-gym’s Downstream Tasks. With the reduced DNN operations generated by our compiler, we can accelerate multiple downstream tasks, such as DNN evaluation on CPU/GPU and DNN evaluation under Homomorphic Encryption (HE) [19].

We propose an acceleration scheme to exploit the parallelism and locality in CPU/GPU to couple with Q-gym’s efficient expressions. In particular, by replacing the inner-dot product of a convolution layer with our expressions, we can exploit the reduction in operations to yield speedup. With different configurations in parallelization, our acceleration scheme can fully utilize the computation resource available. Our acceleration scheme can also effectively integrate with other parallelization schemes, such as loop tiling, vectorization, and multithreading.

We also propose to use Q-gym’s efficient expressions for DNN evaluation under HE for preserving data privacy. It is compatible with mainstream HE libraries while significantly reducing the evaluation overhead.

Experiments. Extensive experiments show that our framework effectively accelerates various DNN applications. For QNN where the number of unique weight \( \left( Q \right) \leq 12 \), Q-gym reduces the number of FLOPs \(( \#FLOPs) \) by 68.9% / 19.1% compared to naïve QNN / SumMerge on average. On CPU \(( Q \leq 3 \) ), Q-gym achieves a speedup of 1.83× / 2.56x compared to SumMerge / OneDNN [2]. On GPU \(( Q \leq 3 \) ), Q-gym achieves a speedup of between 1.64× / 1.78× relative to SumMerge / PyTorch GPU [40]. For DNN evaluation under HE, Q-gym shows 59.5% / 22.9% HE operation reduction compared to CryptoNet [19] and Faster CryptoNet [13] with only -0.6% accuracy reduction due to quantization.

Summary. We summarize our contributions as follows:

- We propose Q-gym, a framework for quantized neural networks that can yield efficient computation dataflow for various downstream applications.
- Q-gym leverages an iterative searching algorithm, e-graph representation, ILP formulation, and elaborated search space to accelerate QNN inference.
- We implement back-end frameworks of Q-gym for multiple sub-domains for QNNs deployment tasks, such as CPU / GPU inference and HE applications. We combine Q-gym with other acceleration schemes and HE protocols.
- We corroborate Q-gym’s general applicability and superior performance across various quantized DNN models.

Q-gym is the first framework to fully exploit weight repetition to accelerate privacy-preserving DNN inference. Also, this is the first work that applies the idea of equality saturation to simplify the arithmetic of DNN computation.
2 BACKGROUND

Background on CNN. While Q-gym is broadly applicable to any DNNs that can be represented by dot-products, we focus on deep convolutional neural networks [21, 32] (referred to as DNNs in the rest of the paper) as examples in this paper to make our proposal more concrete.

DNNs are commonly used for image classification [21, 30], object detection [48, 49], and image segmentation [50]. A convolutional layer implements a set of weight kernels to detect features in the input image. A weight kernel is defined by a set of weights, $W$, and a bias term, $B$. Each convolutional layer applies $K$ kernels of dimension $R \times S \times C$ on the input with dimension $H \times W \times C$, resulting in output feature maps with dimension $(H - R + 1) \times (W - S + 1) \times K$. $C$ denotes the number of input channels. $H$ and $W$ denote input height and weight. $R \times S$ denotes kernel shape. Formally, suppose the input activation is $IA$ and weight is $L$, then the output activation $O$ of this convolution operation is the dot product between input $IA$ and parameters $L$, added by the bias $B$, and followed by a non-linear activation function $g()$. For a layer with a unit stride, this can be represented as:

$$O(x, y, k) = g(B + \sum_{c=0}^{C-1} \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} L(r, s, c, k) \cdot IA(x + r, y + s, c))$$ (1)

For a quantized DNN (QNN), we denote the number of unique weights as $Q$. For a sparse DNN, we denote the sparsity ratio – the ratio between the number of non-zero weights and the number of parameters – as $Sp$.

Background on Homomorphic Encryptions. A broad range of applications, such as medical [26, 45], and fraud detection [5, 41], require privacy and confidentiality in client data. Homomorphic Encryption (HE) [18] is an ideal solution for such applications. When using HE for DNN inference, the Model vendors provide DNN model $M$ and evaluation function $f$ that computes $\hat{y}$ which satisfies $E(x_i) \circ M = E(f(x_i, M, k_{pub}))$. After the cloud completes the model inference, the client can decrypt $(D)$ the result using the private key $k_{pri}$ through Eq.(2). In this way, both parties can keep their data private during the DNN evaluation.

$$f(x_i, M) = D(E(f(x_i, M, k_{pub}), k_{pri})).$$ (2)

The key bottleneck of applying HE on DNN is that homomorphic multiplications and additions are extremely computationally expensive [13, 25]. An evaluation of CryptoNet [19] requires 250 seconds on the MNIST dataset. The inference time of DNN under HE is proportional to the number of HE operations [13] (Detailed in Sec. 5.2).

In this paper, we identify that exploiting weight repetition can greatly reduce the computation overhead for popular HE protocols. The efficient evaluation function $f$ from Q-gym reduces the number of HE operations by up to $2.47 \times 1.30$ compared to the state-of-the-art methods (FastCryptoNet/CryptoNet).

3 MOTIVATION

In this section, we discuss the motivations for proposing Q-gym.

Weight Quantization is Not Fully Explored. Weight quantization is a widely-used method to reduce the storage and computation overhead for deploying DNNs (See Section 7). Most existing works for QNN acceleration, such as BitFusion [54] and FLIM [58], leverage the shorter bit width of weights to accelerate each multiplication and addition. However, we identify that weight repetition features in QNNs are not fully explored for computation reduction in existing works. Specifically, we can bypass computations by factorizing the computation or reusing partial results. (Figure 1).

Previous Works Employing Weight Repetition is Sub-optimal. Activation group reuse (AGR) [22] is the first method that exploits weight repetition for efficient inference. The dot-product between input and each weight kernel will be factorized into expressions with reduced multiplications (Figure 1(a)). The operands of factorized expressions formed activation groups and common sub-expressions between groups can be shared to reduce computation. Starting from the first activation group, AGR greedily extracts the maximum overlapping term between the first and the rest of the terms in the activation groups. Then, it recursively searches the second term and so on.

SumMerge [43] also adopts factorization and activation grouping schemes. Different from AGR, SumMerge (1) iterates all activation groups pairs and generates shared ‘sub-computation’ terms between activation groups in the activation grouping phase and (2) employs a ‘maxscore’ – defined as the number of times each ‘sub-computation’ term appears across activation groups – to determine the sub-computation terms to be reused across. SumMerge iterates between (1) and (2) until no shared sub-computation terms are left.

However, SumMerge is inefficient in the following aspects. i) SumMerge is still a greedy heuristic. Extracting the sub-computation term with the best ‘maxscore’ in each iteration may not be the optimal solution. ii) Factorization can guarantee an upper bound on multiplications ($Q \cdot K$) for the convolution layer $L$. However, the search space of SumMerge is limited, as it only searches for common sub-expressions between activation groups. In other words, SumMerge does not fully exploit the search space of potential computation reuse. For example, it does not reuse the computed result to bypass partial products (e.g., by cz in Figure 1). Also, with the increase of $Q$, the performance of SumMerge drops drastically (Section 6.1) as the size of each activation group reduces. iii) With the increasing size of weight kernels, the computation of ‘maxscore’ is extremely expensive and incurs a large compilation time. In Figure 1, SumMerge and AGR yield the same results. In Table 1, we summarize the comparisons between Q-gym and AGR/SumMerge.

Applying Weight Repetition in Acceleration is Non-trivial. As shown in Eq.(1), the computation of a convolution layer involves 6 loops ($H, W, R, S, C, K$), and how to parallelize the computation has been a long-time problem [12, 46, 54]. The most common techniques

<table>
<thead>
<tr>
<th>Methods</th>
<th>Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evaluated Hardware</td>
<td>CPU</td>
</tr>
<tr>
<td>HE Application</td>
<td>✓</td>
</tr>
<tr>
<td>Temporal Reuse</td>
<td>✓</td>
</tr>
<tr>
<td>Bypass partial product</td>
<td>✓</td>
</tr>
<tr>
<td>Average FLOPs Reduction</td>
<td>&lt;49.8%</td>
</tr>
<tr>
<td>Max CPU/GPU Speedup</td>
<td>-</td>
</tr>
</tbody>
</table>
involves 1) using loop tiling [54] to get better data locality, 2) Multi-threading and 3) SIMD vectorization.

These techniques cannot be directly used in our scenario as Q-gym changes the computation dataflow between the input and weight kernels. In this paper, we carefully design our parallelization scheme so that we can combine the efficient arithmetic compiled from Q-gym with the common acceleration methods on general-purpose hardware (Detailed in Sec. 5.1).

4 COMPILER DESIGN

4.1 Overview

As discussed in Section 3, the previous method of SumMerge [43] with a greedy heuristics cannot fully explore the search space and fails to maximize computation reuse. We propose Q-gym that leverages equality saturation (Section 4.2) to resolve the limitations of SumMerge. Q-gym’s compiler incorporates a two-phase design: exploration and extraction. During the exploration stage (Section 4.3), Q-gym applies a set of rewrite rules on the computation expressions represented in an efficient data structure that can compactly represent a large number of equivalent expressions. For the extraction stage (Section 4.4), Q-gym formulates the selection of expressions as an ILP problem and finds more low-cost solutions compared to the greedy method. To handle an even larger search space, Q-gym also uses a pulsed searching algorithm [27] that iterates the exploration and extraction until convergence (Section 4.5). We also observe that inputs are overlapped during the sliding window computation of output. As such, we propose a temporal reuse search space that can further reduce the operations (Section 4.5).

4.2 Equality Saturation

Equality saturation [57, 62] is a method to resolve the exponential time and space requirement of traditional graph rewriting. Leveraging the efficient data structure e-graph underlying the equality saturation, the rewriting can be applied to the e-graph simultaneously without interfering with each other. Also, the e-graph is a compact representation of equivalent representations, which resolves the memory space limitation in traditional rewriting. In this paper, we propose to use the idea of equality saturation to reduce QNN operations. Many other applications leveraging equality saturation are discussed in Section 7.

In this subsection, we introduce the e-graph and the rewriting rules applied in Q-gym.

E-graphs. An e-graph is a data structure that compactly represents equivalent expressions. An e-graph contains a set of e-classes and a set of e-nodes. Each e-class represents a set of equivalent terms that can be computed from any of its e-node children. Each root e-class represents the final computed term from the input and a weight kernel.

Figure 2(a) shows an e-graph and the representation of an input \([a, b]\) and two simple weight kernels \([x, x]\) and \([x, y]\). The expressions that compute these computations are given at the bottom of the graph, defining e8 and e6 respectively. After applying rule rewriting (Table 2) to saturation (defined in Section 4.3), the e-graph will be expanded to Figure 2(b).

### Table 2: Rewrite Rules used in Q-gym for DNN arithmetic simplification. \(e_i\) denotes an e-class.

<table>
<thead>
<tr>
<th>Description</th>
<th>Source</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mult Commutative</td>
<td>(e_i \times e_j)</td>
<td>(e_j \times e_i)</td>
</tr>
<tr>
<td>Add Commutative</td>
<td>(e_i + e_j)</td>
<td>(e_j + e_i)</td>
</tr>
<tr>
<td>Add Associative I</td>
<td>((e_i + e_j) + e_k)</td>
<td>(e_i + (e_j + e_k))</td>
</tr>
<tr>
<td>Add Associative II</td>
<td>(e_i + (e_j + e_k))</td>
<td>((e_i + e_j) + e_k)</td>
</tr>
<tr>
<td>Mult Distributive</td>
<td>((e_i + e_j) \times e_k)</td>
<td>(e_i \times e_k + e_j \times e_k)</td>
</tr>
<tr>
<td>Mult Factorize</td>
<td>(e_i \times e_k + e_j \times e_k)</td>
<td>((e_i + e_j) \times e_k)</td>
</tr>
</tbody>
</table>

In Q-gym, an e-node is one operator (i.e., ‘*’ and ‘+’) associated with two operands. The input value of each e-node’s operand is a child e-class. Formally,

(i) An e-graph represents a term if any of its e-classes do.

(ii) An e-class represents a term if any of its equivalent child e-nodes do. All terms represented by an e-class are equivalent.

(iii) An e-node \(f(c_0, c_1)\) represents a term \(f(e_0, e_1)\) if each e-class \(e_i\) represents \(c_i\).

In Figure 2(b), ‘e13’ is an e-class that represents both \(b + a\) and \(a + b\). In Figure 2(a), the e-node marked in red represents the term \(x + a\) as e-classes \(e_0\) and \(e_1\) represent \(x\) and \(a\).

Rewrite Rules. In equality saturation, we expand the e-graph using a set of rewrite rules given in Table 2. The set of rewrite rules states the equivalence between each pair of computation arithmetic expressions. During e-graph exploration, rewrite rules will be applied to the e-graph. Specifically, we search for a pattern (source pattern) in the input e-graph that is equivalent to another subgraph pattern (target pattern). Q-gym begins rewriting the e-graph after all the applicable rules are searched.

Note that some of the rewrite rules do not create any new e-class to the e-graph (e.g. the commutative rules). However, they may allow other rules to be applied in the computation (e.g. factorization). Also, these rules increase the connections between e-nodes and e-classes, which enables the extraction phase to find better expressions for computation reuse.

4.3 Exploration Phase

As discussed in Section 4.1, in the first stage of equality saturation, we expand the e-graph using a set of rewrite rules. For example, the exploration phase expands the e-graph from Figure 2(a) to Figure 2(b).

An e-graph is initialized from the input expressions (Figure 2(a)). Then, we search the source pattern of each rewrite rule in the e-graph. If a match is found, it returns the e-class and corresponding substitution (target pattern) to the ‘match list’ \(U\).

Once all the (e-class, substitution) pairs are found, we begin to modify the e-graph in the ‘match list’ simultaneously. Each pair of modifications (i.e., e-class and substitution pair) may create a new e-class/e-node in the graph unless the e-class/e-node already exists.

An e-graph is called saturated if no more e-class/e-node can be created. The exploration phase will stop when one of the following three conditions is met: 1) The e-graph is saturated. 2) The maximum exploration iterations limit \(\text{max}_\text{expl}_{\text{iter}}\) is reached. 3) The maximum e-node limit \(\text{max}_\text{enode}\) is reached. Note that the e-graph for Q-gym is finite though possibly very large. Long-running explorations are prevented by \(\text{max}_\text{expl}_{\text{iter}}\) and \(\text{max}_\text{enode}\).

The pseudo-code of the exploration phase is shown in Algorithm 1.
Algorithm 1 Equality Saturation.

Input: Starting e-graph \( G \); A set of rewrite rules \( R \); Max e-node limit \( \text{max}_{\text{node}} \); Max exploration iterations \( \text{max}_{\text{explore \_iter}} \)

Output: Explored e-graph \( G \); Nodes selected \( I_{\text{node}} \); Objective value cost.

1: \( i \leftarrow 0 \)
2: while \( i < \text{max}_{\text{explore \_iter}} \) do
3: \( G' \leftarrow G, i \leftarrow i + 1, U \leftarrow \emptyset \) // match list
4: \( U \leftarrow U \cup \text{Rule \_Searching}(G, r_j) \) for \( r_j \) in \( R \)
5: \( G \leftarrow \text{Applying \_Rules}(G, u_j) \) for \( u_j \) in \( U \)
6: if \( G \equiv G' \) or \( \text{num\_node}(G') \geq n_{\text{max}} \) then
7:    break; // Saturate or max node limitation reached
8: end if
9: end while
10: \( G, I_{\text{node}}, \text{cost} \leftarrow \text{Extraction}(G) \)
11: \text{return} \( G, I_{\text{node}}, \text{cost} \)

4.4 Extraction Phase

After the exploration phase, the selection of different e-nodes or e-classes from the root node may yield different computation expressions (e.g., Extraction #1 and #2 in Figure 2(b)). The extracted expressions are equivalent to the input term which is guaranteed by the feature of e-graph. The goal of the extraction phase is to select the best-represented term for each root e-class according to a cost model.

Many extraction methods for equality saturation have been proposed, such as greedy algorithms [39] or ILP solutions [60, 61]. One of the key differences between simplifying DNN arithmetic and previous works of equality saturation (Section 7) is that multiple root nodes co-exist during the extraction phase. That is because each weight kernel \((k, k \in \{0, \ldots, K\})\) in layer \( L \) yields different output activation results \((O(x, y, k))\).

In this section, we discuss two strategies to extract low-cost expressions from an expanded e-graph.

Notations. Let \( i \) be an e-class where \( i = 0, \ldots, N - 1 \). Let \( m \) be an e-node where \( m = 0, \ldots, M - 1 \). Let \( m.\text{op} \) denote the operator of the e-node and \( m.\text{r}_0 \) / \( m.\text{r}_1 \) denote the operands of the e-node. Let \( \text{root} \) be the set of root e-classes.

Let \( e_i \) and \( n_m \) denote a binary integer which indicates if the corresponding e-node \( i \) and e-class \( m \) is selected or not.

Let \( m.\text{r}_0 \) denote the child e-class of operand \( \text{r}_0 \) in e-node \( m \), and let \( \text{child}(i) \) denote the set of child e-nodes of e-class \( i \), i.e., \( \{m|m \in \text{child}(i)\} \).

Cost Model. In this paper, we define our cost model for an e-node \( m \) as Eq.(3).

\[
\text{cost}(m) = \begin{cases} 
C_{\text{add}}, & \text{if } m.\text{op} = + \\
C_{\text{mult}}, & \text{if } m.\text{op} = \times \\
0, & \text{otherwise}
\end{cases}
\] (3)

Here, \( C_{\text{mult}} \) and \( C_{\text{add}} \) are constant values. \( m.\text{op} \) indicates the operator associates with the e-node \( m \). On an Intel i7-8700K CPU, the latency ratio between floating-point (FP) multiplication and FP addition is 3:5. And their reciprocal throughput ratio is 1:2 [17]. Assuming the computation units are fully-pipelined, we choose \( C_{\text{mult}} = 2 \) and \( C_{\text{add}} = 1 \) to make the generated arithmetic expressions more suitable for the deployed hardware. We report complexity for DNN computation as FLOPs, i.e. the sum of additions and multiplications, which are the most commonly used metrics to evaluate the computation complexity of DNN models.

Note that a more complicated and accurate cost model (e.g., dynamic cost) may better characterize the deployed hardware and it is left for future work.

Greedy Extraction Algorithm. We first experiment with a greedy extraction strategy. Q-gym computes the subgraph cost of each e-node using the cost model given in Eq.(3). For each e-class, Q-gym selects the e-node with the smallest subgraph cost. Then, the computation expression for each root e-class is determined.

Note that greedy extraction is not guaranteed to extract the graph with the minimum cost. That is because the e-nodes are selected locally in each eclass to minimize the subgraph cost and it fails to consider the potential computation sharing between e-nodes whose subgraphs may be overlapped.
When the size of a weight kernel is large, the e-graph can hardly optimize the cost of arithmetic operations:

\[
\text{Minimize: } f(x) = \sum_{m=0}^{M-1} \text{cost}(m) \cdot n_m
\]  

The constraints of the ILP are listed below:

1. All the root e-classes (root) should be included as they represent the final computation results:

\[ e_i = 1, \forall i \in \{i | i \text{ is root}\} \]  

2. Also, we need constraints to say that if an e-class i is included, then at least one of its child e-nodes is included. Otherwise, the term represented by the selected e-class cannot be reached.

\[ e_i \leq \sum_{j \in \text{child}(i)} n_m \]  

3. For each e-node m selected, the child e-class of each operand must be included. This guarantees that the input to the e-node is not empty.

\[ n_m \leq e_{m,r0}, \quad n_m \leq e_{m,r1} \]

To recap, the ILP optimization problem can be defined as minimizing Eq.(4) which is subject to constraints Eq.(5 - 7).

### 4.5 Pulsed e-graph Searching

When the size of a weight kernel is large, the e-graph can hardly reach saturation during the exploration phase due to hardware memory limitations. To resolve this issue, we propose an efficient searching algorithm by iteratively conducting exploration and extraction. Specifically, Q-gym expands the e-graph until the number of e-nodes or exploration steps reached its predefined limits (i.e., max\_enode, max\_explore\_iter) and extracts the lowest-cost computation expressions using the ILP or greedy methods. Next, Q-gym will explore a new e-graph starting from the last generated computation expressions as shown in Algorithm 2. Our results show the pulsed searching algorithm can significantly reduce the computation cost throughout iterations (Section 6.1).

**Algorithm 2**: Pulsed e-graph Searching

**Input**: Input expressions expr; Set of rewrite rules \( R \); Max node count max\_enode; Max searching epochs Epochs

**Output**: Output expressions expr\_o; Final e-graph \( G' \); Selected node list \( L'_{\text{node}} \)

1. cost’ \( \leftarrow \infty \)
2. \( G \leftarrow \text{Initialize\_Egraph(expr\_p)} \) // Initialize e-graph \( G \)
3. for \( i \) in 0, \ldots, Epochs do
   4. \( G, L_{\text{node}}, \text{cost} \leftarrow \text{Equality\_Saturation}(G, R, \text{max\_enode}) \)
   5. \( G \leftarrow \text{Rebuild\_eGraph}(G, L_{\text{node}}) \) // Remove all unselected nodes
   6. if \( \text{cost} \leq \text{cost'} \) do \( G', L'_{\text{node}} \leftarrow G, \text{cost}, L_{\text{node}} \), end for
   7. expr\_o \( \leftarrow \text{rebuild\_expressions}(G', L'_{\text{node}}) \) // rebuild output expressions from e-graph
8. return expr\_o, \( G', L'_{\text{node}} \)

**Figure 3**: An illustration of temporal reuse search space. \( tpr \) denotes computations that are explored together. The gray area is the overlapping area of inputs across timesteps. The red / green / orange squares denote the same convolutional layer in different time steps.

**Temporal search space**. Q-gym also observes that when \( R = S \geq 2 \), a convolutional layer, the input activation overlaps in temporal dimensions when computing \( O(x, y, k) \) and \( O(x, y+1, k) \). That means computation reuse can also be applied between the computation of different \( O \). As shown in Figure 3, if Q-gym searches two continuous convolution operations together, Q-gym can potentially find better computation expressions with lower cost in QNNs. We define the number of convolution operations searched together as timesteps \( tpr \).

### 5 Q-GYM’S DOWNSTREAM TASKS

To exploit the efficient computation expressions generated by our compiler, we test Q-gym’s performance on various downstream DNN applications.

#### 5.1 Accelerating CNN on CPU and GPU Systems

The generated efficient expressions from Q-gym can be used to accelerate CNN inference on CPU/GPU. In this subsection, we detail how to use the generated expressions to yield speedup in DNN inference on CPU/GPU.

**Mapping Expressions to Code**. As shown in Eq.(1), the convolution operation iterates over 6 dimensions \((H, W, R, S, C, K)\). In Q-gym, we unroll the inner for-loop over \((R, S, C, K)\) dimensions. Specifically, the computation over \((R, S, C, K)\) for weight layer \( L \) yields \( K \) output results (i.e., \( O(x, y, k) \), \( k \in \{0, ..., K\} \)). We replace the unrolled computations with Q-gym’s efficient expressions through a common function call. The weight value is hardcoded in the function; so we don’t need to load weights during the inference. The output \( K \) results from naive loops over \((R, S, C, K)\) and Q-gym’s function call are identical. Note that the convolutions across input dimensions \( H, W \) are still repeated in each iteration, so the computation of different output pixels shares the same efficient expressions. No code generation compiler is used.

**Combination of Q-gym with Parallelism Methods**. By treating the computations along \((R, S, C, K)\) as an atomic function, we can easily adapt the efficient computation expressions with (i) loop tiling, (ii) multithreading, and (iii) vectorization, respectively. In Figure 4, we illustrate how to combine the computation of our generated expressions with (i)(ii)(iii).
We identify that the low-cost computation arithmetic from Q-gym’s compiler significantly reduces the evaluation time of DNN under homomorphic encryption (HE) as discussed in Section 2. The generated computation dataflow can be easily combined with different HE libraries (i.e., HELib [4], SEAL [53]) by simply replacing the evaluation function \( f \) with the expressions compiled from Q-gym without any modifications to the encryption protocol. Note that Q-gym does not increase the depth of multiplication in HE; so it will not affect the correctness of the output result. We evaluate Q-gym’s efficient expressions using two popular HE protocols (BGV [10] / BFV [16]). We also verify the correctness of the Q-gym’s expressions using HElib/SEAL. HElib/SEAL with BGV/BFV schemes are also used to implement the CryptoNet/FastCryptoNet, respectively.

Note that the runtime of DNN inference under HE is linear to the number of HE operations. Specifically, there are four types of HE operations in DNN inference: (i) plaintext (PT)-ciphertext (CT) addition, (ii) ciphertext-ciphertext addition, (iii) plaintext-ciphertext multiplication, and (iv) ciphertext-ciphertext multiplication. Because the wall-clock time for executing different types of HE operations is highly dependent and is relevant to other settings (e.g., key size, machine settings), in this paper, we use the number of HE operations (HOPs) for comparison. This metric is commonly used for other privacy-preserving DNN methods, such as FastCryptoNet [15].

### 6 Evaluation

In this section, we evaluate the performance of our compiler design in reducing QNN computations (Section 6.1) and the performance of Q-gym in accelerating various DNN applications (Section 6.2).

#### 6.1 Algorithm Analysis

**Experimental Setup.** We implement the algorithm of Q-gym in Rust from egg [62], an open-source equality saturation library. During the e-graph extraction phase, we use Gurobi [1] as the ILP solver. We also reimplemented SumMerge in Python with loop parallelism as the baseline. The compilation time is tested on Intel Core i7-8700K CPU with 6 physical cores. This machine has 32/32 KiB of L1 instruction/data cache, 256KiB of L2 cache per core, and 12 MiB of shared L3 cache (Detailed in Table 3).

To measure the performance of Q-gym in reducing computations, we use the ratio between the number of reduced FLOPs and the total number of FLOPs (\( \text{ops} \)) as the evaluation metric. For comparison purposes, we conduct experiments with Q-gym that use the greedy method during extraction (denoted as Q-gym\(_{gd} \)). Also, Q-gym without the pulsed searching algorithm is denoted as Q-gym\(_{-p} \) (i.e., \( \text{max}_{\text{epoch}} = 1 \)).

Without specification, we set \( \text{max}_{\text{node}} = 10^7 \), epochs \( \text{max}_{\text{epoch}} \) to 10, \( \text{max}_{\text{explore iter}} = 8 \), and set the temporal reuse steps (tpr) to 0 for Q-gym. For ILP extraction time limitation, we set it to \( \sqrt{K \cdot S \cdot C \cdot K} \) seconds which is relative to the number of weight kernels in the convolutional layer.

**Comparison of Reduced Operations.** Figure 5 compares the reduction of FLOPs between Q-gym, Q-gym\(_{gd} \), and SumMerge. We evaluate the algorithms on QNNs where the weights are generated synthetically using a uniform distribution. We assume none of the weights are 0 in the synthesized layers.

The result shows that Q-gym achieves significant computation reductions compared to SumMerge on quantized DNN with different Q. On average, Q-gym shows 19.1% / 15.5% more computation...
Figure 5: Comparison of reduced operations (ops) between (a) Q-gym (b) Q-gym\text{gd} (c) SumMerge over different quantization schemes (Q) and layer sizes (R \cdot S \cdot C \cdot K).

Table 3: CPU/GPU configurations.

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel i7-8700K, 6 physical Cores, 3.7 GHz, 1 sockets</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>NVIDIA 2080, 8GB main memory, 1024 maximum threads</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>32KiB 8-way I$, 32KiB 8-way D$, private</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>256KiB, 16-way, private</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>12MiB, 11-way, shared</td>
</tr>
<tr>
<td>TLB</td>
<td>L1D 4-way 64 entries, L1I 8-way 128 entries STLB 12-way 1536 entries</td>
</tr>
<tr>
<td>DRAM</td>
<td>DDR4, 32GB, 2666MHz, 2 sockets, 6 channels per socket</td>
</tr>
<tr>
<td>Kernel</td>
<td>Linux 5.4.0</td>
</tr>
<tr>
<td>Software</td>
<td>GCC 7.1, PyTorch 1.4.0+cuDNN, CUDA 11.2, Rust 1.56.1</td>
</tr>
</tbody>
</table>

reduction than SumMerge in Figure 5. Q-gym also achieves a more substantial operation reduction compared to Q-gym\text{gd}. As discussed in Section 4.4, using an ILP solver can find a better solution with a lower computation cost compared to a greedy heuristic.

Q-gym\text{gd} can also outperform SumMerge. On average, Q-gym\text{gd} shows 3.6% more computation reduction across all the kernels tested in Figure 5 (2.3% / 8.3% on average for Q = 2 / Q = 3). This is because e-graph exploration covers a larger search space compared to searching the common expressions between activation groups.

Compilation Time of Q-gym's Algorithm. We evaluate the time for Q-gym to compile a given weight layer into efficient computation expressions. Figure 6 shows the compilation time of Q-gym. With the increasing weight dimension and unique weights (Q), SumMerge takes a much longer compilation time than Q-gym. That is because SumMerge checks the overlapping of terms between all activation groups for computing the 'maxscore' (Section 3). Also, the 'maxscore' is computed many times until no overlapping between activation groups is found.

With the increase of Q, the compilation time of Q-gym does not change a lot due to the e-node/explore step limitation (max\_node, max\_explore\_iter) and max epochs (max\_epochs) for pulsed searching. To apply Q-gym in runtime compilation techniques, the users can choose Q-gym\text{gd} which is much faster compared to Q-gym and SumMerge.

Sensitivity to the Number of Unique Weights (Q) and Layer Size. With a growing number of weights, the performance of Q-gym also increases (Figure 5). When Q = 12, the computation reduction ratio of Q-gym increases from 38.3% to 62.9% with the growing size of weights. For SumMerge, the performance eventually converges at ~50%. This means the greedy heuristic cannot fully exploit the computation reuse.

With the growing number of unique weights (Q), Q-gym always shows more computation reduction compared to SumMerge. Specifically, for Q = 2 / Q = 3 / Q = 12, Q-gym shows 21.3% / 27.6% / 11.2% more computation reduction compared to SumMerge. Note that SumMerge’s performance decreases with the increase of Q. This is because the activation groups in SumMerge are factorized into smaller sets when Q is large, offering fewer opportunities for computation reuse.

Sensitivity to Temporal Reuse Steps (tpr). Figure 7 shows a sensitivity analysis of tpr on different QNN layers. With larger R, more inputs are overlapped across time steps and achieve lower computation cost when tpr \geq 1. For weight layers where (R,S) is (3,3), (5,5), (7,7), Q-gym where tpr = 3 shows 4.7, 6.0%, and 6.2% more computation reduction compares to Q-gym where tpr = 0. With the increasing tpr, the overlapping area of input activations also increases (Figure 3).

Comparison between Q-gym and Winograd. We also compare Q-gym with temporal reuse against Winograd, a commonly used technique that leverages fast FFT to reduce computation operations in convolutional layers. The downside of Winograd is that the transformation overhead is not negligible, i.e., the storage overhead of transformation matrices and the computation of input, weight transformations, and inversions. Also, Winograd has a theoretical boundary of 75% computation reduction (factor of 4). In this comparison, we did not take into consideration the extra computation for input, weight transformation, and inversion for Winograd. Thus, the baseline is stronger than what happens in practice.
Analysis of Q-gym on group convolutions. We also verify if \(\mbox{Q-gym with temporal reuse to Winograd (denoted as 'wino')}. We assume the input tile size \((H, W)\) for Winograd is \((4, 4)\), \((12, 12)\), \((25, 25)\) for weights \((R, S)\) \((3, 3)\), \((5, 5)\), \((7, 7)\), respectively. The corresponding \(\alpha\) are \(2.25 / 10.51 / 19.61\) in Winograd [31].

On average, Q-gym shows 24.2% more computation reduction compared to Winograd as shown in Figure 7. Note that Q-gym can be combined with Winograd and we leave it as future work.

Analysis of Q-gym on group convolutions. We also verify if Q-gym is still working in group convolution layers [51, 59, 67]. Traditional convolution layers have a weight dimension \((R, S, C, K)\) as given in Eq.(1). With group convolution where the grouping factor is \(g\), the channel of each weight kernel would be \(C_g = C/g\). The \(K\) kernels will be separated into \(g\) groups and each group has \(K_g = K/g\) weight kernels, i.e., each group has a dimension of \((R, S, C_g, K_g)\). When \(g = C\), the convolution layer will become a depth-wise convolution layer. During group convolution, the input activation can be reshaped into \((H, W, C_g, L_g)\). Each group of kernels \((R, S, C_g, K_g)\) handles different groups of the input activations \((R, S, C_g)\) in the same fashion given in Eq.(1). As such, Q-gym can still handle group convolutions when \(R \times S \times C_g \times K_g > 1\). But with the increase of \(g\), the search space decreases, giving Q-gym less opportunity to reuse computations. As shown in Figure 8, when \(Q = 3\), the operation reduction drops from 77.1% of traditional convolution to 38.9% for depth-wise convolution layers.

Note that most of the depth-wise or group convolutions are implemented together with traditional convolutional layers [51, 59, 67]. Also, these models are typically not quantized due to a large accuracy drop.

Effectiveness of Pulse Searching Algorithm. Figure 9 shows the operation reductions over different epochs during Q-gym’s pulse searching. On average, Q-gym with pulsed e-graph searching can reduce the computation cost by 9.9% compared to Q-gym−p. For small kernels, e.g., \((1, 1, 32, 32)\), the computation reduction of pulse searching is limited due to the relatively small search space. For large kernels, pulse searching shows substantial performance improvement over Q-gym−p. Note that from epochs 5 to 7, the number of operations is still reducing across different test cases. However, overall the number of reduced FLOPs becomes negligible compared to the total FLOPs.

6.2 Evaluation of Q-gym’s Downstream Tasks

Experimental Setup. We evaluate the performance of Q-gym on an Intel Core i7-8700K CPU (The same machine as Section 6.1) and an NVIDIA 2080 GPU (1024 max threads), respectively. For CPU and GPU baselines, we choose to compare against the expressions generated from SumMerge. We also compare against the state-of-the-art DNN compiler (OneDNN [2]) for Intel CPU and Pytorch GPU (v1.4.1) [40] which has a carefully designed cuDNN/CUDA (v11.2) back-end for performance purposes. All the weights and input activations are 32-bit floating-point numbers. For all comparisons, we set the batch size to be 1 to accommodate the common configurations used in real-time systems. The implementation methods of Q-gym for CPU/GPU are described in Section 5.

Performance of Q-gym on CPU. Figure 10 shows the per-layer speedup with different Qs on the CPU. All experiments assume an input of dimensions of \(H = W = 48\). We enable multithreading (MT) for both Q-gym and OneDNN. For MT implementation in Q-gym, we set \(N_{thr} = 12\) and \(K_{thr} = 1\) across all layers tested to maximize CPU resource utilization. For OneDNN settings, we scan the number of threads from 1 to 12 for each layer and select the best-performed setting. For SumMerge, we apply the same MT implementation as Q-gym (Section 5) with the computation expressions generated from SumMerge. For vectorization, we unroll the loop along \(H\) dimensions (unrolled factor = 8).

When \(Q \leq 3\), Q-gym shows 2.56x / 1.83x speedup compared to OneDNN and SumMerge on average across different convolution layers. Note that \(Q \leq 3\) are common settings for QNNs (Table 4). We observe that when the \(Q\) gets larger, the speedup of Q-gym
Table 4: Comparison between Q-gym and SumMerge/OneDNN on CPU performance. ∘ TTQ is a sparse and quantized model (one of the three values is 0. ¶ VGG-small is a derivative architecture based on VGG [56]. ‡ ’−Ops Gap’ denotes the gap of computation reduction ratio (−ops) between Q-gym and SumMerge. The models can be found in the repository [64, 69].

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DNN Arch</td>
<td></td>
<td>Full Quantized %</td>
<td>CPU MT Speedup</td>
<td>CPU MT Speedup</td>
</tr>
<tr>
<td>AlexNet</td>
<td>3</td>
<td>ImageNet</td>
<td>TTQ [70]</td>
<td></td>
</tr>
<tr>
<td>ResNet-18</td>
<td>2</td>
<td>CIFAR-100</td>
<td>LS-1 [42]</td>
<td></td>
</tr>
<tr>
<td>ResNet-18</td>
<td>3</td>
<td>CIFAR-100</td>
<td>LS-T [42]</td>
<td></td>
</tr>
</tbody>
</table>

Figure 10: Per-layer speedup (higher is better) comparison on CPU. The tuple \((R, S, C, K)\) denotes the size of different convolution layers. For OneDNN implementation, we use the official performance profiling tool [3].

Figure 11: Per-layer speedup (higher is better) comparison on GPU.

Figure 12: Per-layer GOPS and peak GOPS on CPU across different convolutional layers. The tuple \((R, C)\) denotes the size of the kernel \((R, S, C, K)\) where \(R = S, C = K\).

Q-gym shows an average speedup of \(1.92x / 1.64x\) compared to PyTorch when \(Q = 2 / Q = 3\). Also, Q-gym shows an average speedup of \(1.63x\) when \(Q \leq 3\) over the computation expression from SumMerge. With the increasing size of weight layers, the performance of Q-gym drops faster than PyTorch due to the larger instruction size and memory footprint. Also, for the convolution layer where \(H\) and \(W\) are small, Q-gym parallelizes on the \(K\) dimension to maximize the usage of GPU resources. This will reduce the search space of Q-gym’s compilation phase and reduce the opportunity for computation reuse.
Performance of Q-gym for HE. As mentioned in Section 5.2, we compare DNN inference under HE using the homomorphic operations (HOPs) following previous works [13]. Due to the expensive computation cost of HOP on real hardware, the inference time of HE applications is linear to the number of HOPs regardless of the hardware or software parallelization schemes.

For the baseline comparison, we choose CryptoNet [19], a dense DNN model that is trained on MNIST [32]. FastCryptoNet [13], a follow-up work of CryptoNet that leverages sparsity to bypass HOPs. We use a trained dense QNN and a trained sparse QNN to compare with the baselines separately. The model architectures are the same as FastCryptoNet (a slight variant of CryptoNet). We apply Q-gym to compile the models for HOPs comparison.

As is shown in Table 5 and 6, Q-gym achieves 59.5% and 22.9% HOPs reduction relative to CryptoNet and FastCryptoNet. Meanwhile, the models trained using INQ achieve almost the same accuracy (-0.11%/0.02%) compared to CryptoNet / FastCryptoNet. Besides, Q-gym reduces the PT-CT multiplication by 10.8% / 1.87x which is the computation bottleneck of DNN inference under HE.

Deploying Q-gym on Small Devices. During code generation, Q-gym unrolls the inner convolution loops and replaces them using efficient computation expressions. As such, the compiled binary size is larger than the one using naive loops. For small architectures (e.g., mobile devices) with a small instruction cache, the large number of instructions may decrease the performance of Q-gym. On those devices, Q-gym can be employed using an alternative option by leaving the computation expressions in the "interpreted form", i.e., we load the expressions during inference and compute the output accordingly, which is the way implemented in SumMerge [43]. Q-gym would then still be more efficient than the "interpreted" SumMerge due to better performance in reducing operations. On our experiment architectures, Q-gym’s deploying method is better and we use the same way to evaluate SumMerge in this paper.

7 RELATED WORK

This section discusses previous works related to our study.

Quantized Neural Networks. Neural network quantization [64, 68, 70] is a promising technique to compress and accelerate DNNs. The reduced bit width enables low-bit width multiplication [54] to speed up the inference. Jung et al. introduced parameterized quantization intervals and optimized them to minimize task loss [24]. Han et al. used k-means clustering as a method of quantization to share weights [20]. Zhou et al. proposed a rule-based non-uniform quantization by leveraging a logarithmic distribution [68]. [64, 70] optimize the quantization clipping range during model training. Beyond the computer vision tasks, quantization can also be applied to BERT [55, 65] for NLP tasks. Q-gym can be applied to all these quantization models to reduce the computations of QNNs.

Equality Saturation Applications. The key idea of Q-gym is equality saturation [57, 60]. It has also been widely used in various domains, such as simplifying CAD design, rewriting DNN architectures, improving the accuracy of floating-point expressions and doing semantic code search [37, 39, 44, 63]. In this paper, we contribute to the iterative searching algorithm and elaborated search space to better reduce the computation in DNN. This is also the first work that applies equality saturation to accelerate homomorphic encryption for quantized DNNs inference.

Table 5: A breakdown of HOPs for each layer between Q-gym and CryptoNet. The dense model for Q-gym uses 2-bit INQ [68] where \( Q = 4 \). The model architecture (same as FastCryptoNet) for Q-gym is a slight variant of CryptoNet. We set \( tpr = 0 \) for all layers’ compilation. Fully-connected (FC) layers can be treated as special convolutional layers (conv) where \( R, S, H, W \) are 1. ‘Act’ denotes activation layers.

<table>
<thead>
<tr>
<th>Layer (op)</th>
<th>Conv-1</th>
<th>Act-1</th>
<th>Pool-1</th>
<th>Conv-2</th>
<th>Act-2</th>
<th>FC-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv-1</td>
<td>42,757</td>
<td>845</td>
<td>6,845</td>
<td>309,905</td>
<td>100</td>
<td>1990</td>
</tr>
<tr>
<td>Act-1</td>
<td>845</td>
<td></td>
<td>6,845</td>
<td>845</td>
<td>100</td>
<td>990</td>
</tr>
<tr>
<td>Pool-1</td>
<td>6,845</td>
<td></td>
<td>1,520</td>
<td>845</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conv-2</td>
<td>309,905</td>
<td>1,520</td>
<td>154,350</td>
<td>845</td>
<td>100</td>
<td>990</td>
</tr>
<tr>
<td>Act-2</td>
<td>100</td>
<td></td>
<td>100</td>
<td>100</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>FC-1</td>
<td>1990</td>
<td>10</td>
<td>990</td>
<td>990</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>612,129</td>
<td>2,205</td>
<td>312,137</td>
<td>296,842</td>
<td>296,842</td>
<td>945</td>
</tr>
</tbody>
</table>

Accuracy
99.17

Table 6: Comparison between Q-gym and FastCryptoNet. As FastCryptoNet is a sparse model \( sp = 6.63\% \) and applied 2-bit INQ quantization [68], Q-gym uses a sparse and quantized model \( (Q = 4) \) for evaluation accordingly. Sparse ratio \( sp \) is set to be the same as Fast CryptoNet.

<table>
<thead>
<tr>
<th>Layer (op)</th>
<th>Conv-1</th>
<th>Act-1</th>
<th>Pool-1</th>
<th>Conv-2</th>
<th>Act-2</th>
<th>FC-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv-1</td>
<td>8,619</td>
<td>5,070</td>
<td>6,845</td>
<td>22,950</td>
<td>600</td>
<td>477</td>
</tr>
<tr>
<td>Act-1</td>
<td>5,070</td>
<td>1,690</td>
<td>1,690</td>
<td>1,690</td>
<td>1,690</td>
<td>1,690</td>
</tr>
<tr>
<td>Pool-1</td>
<td>6,845</td>
<td>1,690</td>
<td>1,690</td>
<td>1,690</td>
<td>1,690</td>
<td>1,690</td>
</tr>
<tr>
<td>Conv-2</td>
<td>22,950</td>
<td>1,690</td>
<td>10,850</td>
<td>10,850</td>
<td>10,850</td>
<td>10,850</td>
</tr>
<tr>
<td>Act-2</td>
<td>600</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>FC-1</td>
<td>477</td>
<td>427</td>
<td>427</td>
<td>427</td>
<td>427</td>
<td>427</td>
</tr>
<tr>
<td>Total</td>
<td>247,864</td>
<td>3,995</td>
<td>3,995</td>
<td>3,995</td>
<td>3,995</td>
<td>3,995</td>
</tr>
</tbody>
</table>

Accuracy
99.06
Acceleration of Homomorphic Encryptions for DNNs. Since the first Fully HE scheme was proposed by Gentry et al. [18], many acceleration methods for FHE have been proposed, such as Levelled Somewhat HE (SWHE) [7, 9, 10]. Many advances [8, 16, 25] leverage SWHE to do privacy-preserving DNN inference. Cryptoneto [19] is the first work to use SWHE for DNN inference. FastCryptoneto [13] leverages DNN model sparsity to accelerate CryptoNet. Many other works have aimed at improving the computation efficiency of non-linear layers [34] or computing SWHE in a SIMD fashion [25]. These methods are orthogonal to the computation reduction generated from Q-gym and can be combined to further improve the efficiency of HE DNN inference. Note that Q-gym does not tamper with the security of HE. The input is kept private all the time and no information is leaked.

8 CONCLUSION

This paper proposes Q-gym, a DNN framework that accelerates quantized DNNs by exploiting the weight repetition characteristic. Q-gym proposes a compiler and a set of acceleration schemes for various DNN applications. For the compiler, Q-gym employs the idea of equality saturation and represents the DNN computation into an e-graph and applies a set of rewrite rules to explore equivalent expressions. After the exploration, we extract the lowest-cost computation expressions from the e-graph by formulating the selection of nodes in the e-graph into an integer linear programming problem. By iteratively conducting exploration and extraction and leveraging a temporal search space, we can further reduce the computation operations compared to previous works. Leveraging the reduced computation, Q-gym can accelerate a set of DNN applications. We build QNN inference kernels on CPU and GPU with carefully designed parallelization schemes and combine the efficient expressions from Q-gym with multi-threading, vectorization, and loop tiling. Q-gym also proposes to combine the reduced expressions into DNN inference flow under homomorphic encryption. Experiments show significant speedups and operation reductions in those applications compared to the state-of-the-art methods.

REFERENCES


PACT ’22, October 10–12, 2012, Chicago, IL Fu, et al.