QuadraLib: A Performant Quadratic Neural Network Library for Architecture Optimization and Design Exploration

Zirui Xu [1] Fuxun Yu [1] Jinjun Xiong [2] Xiang Chen [1]

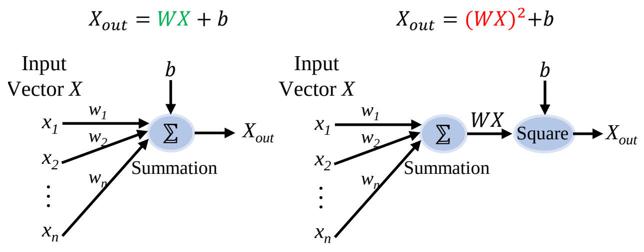
Abstract

The significant success of Deep Neural Networks (DNNs) is highly promoted by the multiple sophisticated DNN libraries. On the contrary, although some work have proved that Quadratic Deep Neuron Networks (QDNNs) show better non-linearity and learning capability than the first-order DNNs, their neuron design suffers certain drawbacks from theoretical performance to practical deployment. In this paper, we first proposed a new QDNN neuron architecture design, and further developed *QuadraLib*, a QDNN library to provide architecture optimization and design exploration for QDNNs. Extensive experiments show that our design has good performance regarding prediction accuracy and computation consumption on multiple learning tasks.

1 Introduction

The availability of easy-to-use Deep Learning (DL) frameworks/libraries, ranging from the early days' Theano (Bergstra et al., 2010), Caffee (Jia et al., 2014), Torch (Paszke et al., 2019) to the recent Tensorflow (Abadi et al., 2016), PyTorch (Paszke et al., 2019), MxNet (Chen et al., 2015), etc. have undoubtedly contributed to the popularity of DL and the proliferation of many successful deep neural networks (DNNs) for a variety of applications, such as *VGGNet* (Simonyan & Zisserman, 2014) and *ResNet* (He et al., 2016a) for image classification, *Faster RCNN* (Ren et al., 2015) and *SSD* (Liu et al., 2016) for object detection, and *BERT* (Devlin et al., 2019) and *GPT-3* (Brown et al., 2020) for natural language modeling. The success and wide adoption of these DNN models in turn raise many new requirements for these DL frameworks, thus pushing the continued development of new functionalities inside these DL frameworks. This has created a positive feedback cycle for the unparalleled prosperity of the entire DL ecosystem, and has become a new source of inspiration for innovation for the entire machine learning community.

One recent notable progress in improving DNN's learning capacity is the proposal of Quadratic Deep Neural Networks (QDNNs). In contrast to the existing DNNs where each neuron is represented by a linear combination of its inputs X and weight parameters W (i.e., a first-order polynomial form) as shown in Fig. 1 (a), the QDNN's neurons are represented by a second-order polynomial of inputs X and weight parameters W as shown in Fig. 1 (b), thus improving DNN models' learning capacity (Milenkovic et al., 1996; Redlapalli et al., 2003; Zoumpourlis et al., 2017; Ganesh et al., 2017; Fan et al., 2018; Jiang et al., 2019; Brooks et al., 2019; Mantini & Shah, 2021; Chrysos et al., 2020; Bu & Karpatne, 2021). Compared to the linear neurons, the benefits of QDNNs stem from the unique characteristics of the second-order polynomial form: (1) stronger non-linearity, hence improved capability for feature extraction (Jiang et al., 2019), and (2) higher model efficiency as QDNN can approximate polynomial decision boundaries using smaller network depth/width (Chrysos et al., 2020; Bu & Karpatne, 2021). Moreover, with such non-linearity, replacing ReLU with quadratic layer will significantly reduce the computation cost in many Privacy-Preserving Machine Learning (PPML) protocol designs (Mishra et al., 2020; Gilad-Bachrach et al., 2016; Chou et al., 2018; Garimella et al., 2021). In spite of these great potentials, the number of follow-up works in QDNNs is, however, substantially smaller than the traditional first-order DNNs.



(a) Linear Neuron (b) A Typical Quadratic Neuron

Figure 1. From Linear Neuron to Quadratic Neuron

When we look closely at the evolution path for the first-order DNNs' development as shown in Fig. 2, it has seen a similar issue. For example, although Convolutional Neural

[1]George Mason University, VA, USA [2]University at Buffalo, NY, USA. Correspondence to: Xiang Chen <xchen26@gmu.edu>, Jinjun Xiong <jinjun@buffalo.edu>.

Proceedings of the 5th MLSys Conference, Santa Clara, CA, USA, 2022. Copyright 2022 by the author(s).

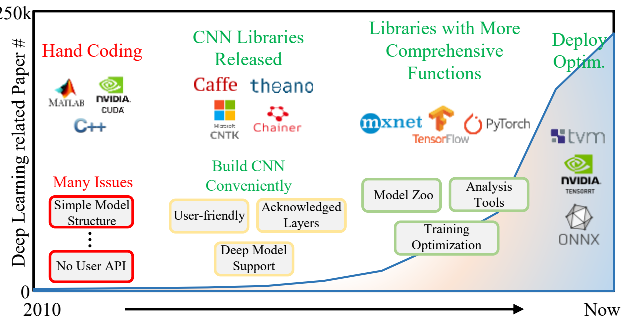


Figure 2. DNN Library Timeline and Paper Number

Networks (CNNs) were proposed as early as 1990s, there were few follow-up works until around 2012. The major roadblock was the productivity issue as researchers during that period had to build models by writing customized software with low-level primitives (such as C/C++ and Nvidia CUDA). Starting around 2012, DNN development started to ramp up with the introduction of DNN libraries and frameworks as pioneered by Theano (Bergstra et al., 2010), Torch (Collobert et al., 2011), and Chainer (Tokui et al., 2015) etc. These works provided a user-friendly APIs to include a set of pre-defined DNN layers and models for network construction, and the number of DNN-based research has since increased significantly as represented by the volume of published research papers around DNNs. The diversity of DNN/CNN models raised the need for even more productive tools and libraries, which incentivised companies to open-source frameworks like Tensorflow (Abadi et al., 2016), Pytorch (Paszke et al., 2019), etc. These libraries further provided optimized training/inference computation to ensure fast and optimal convergence of new DNN models. Moreover, some model visualization tools were developed to enable in-depth analyses of DNN models, such as TensorBoard (Abadi et al., 2016). In summary, as demonstrated in Fig. 2, a set of well-defined libraries and tools really enabled the research community to fully understand the power of first-order DNN by facilitating their fast model exploration, thus speeding up their innovations in designing more powerful DNN models for the past decade.

Inspired by this historical success for the first-order DNN development, we believe a similar effort is needed for the second-order DNNs, i.e., QDNNs, so that we can enable more researchers to better explore various QDNN models and understand QDNNs' strengths and weakness. The contributions of our work are as follows:

- We first categorized the existing QDNN design into four types based on their neuron architecture and then comprehensively analyzed the drawbacks of each design type from multiple perspectives.
- Based on the drawbacks analysis, we proposed a new quadratic neuron architecture. The effectiveness and efficiency of the proposed neuron is further analyzed from the theoretical aspect.
- We developed a second-order DNN library based on PyTorch, called *QuadraLib*, that not only supports flexible model construction for various QDNNs but also optimizes memory overhead for QDNN training.
- We conducted a systematic study of our proposed QDNN model on various learning tasks, including image classification, image generation and object detection. The extensive experimental results validate its superiority compared to the first-order DNNs and the existing QDNN models.

We plan to open source our *QuadraLib* to the community with a simple-to-use installation instruction. We hope this will stimulate the community's interests in conducting more active research around QDNNs so that we can jointly expand the arsenal of the existing DNN models to include higher-order neuron networks and also provide potential benefits to PPML society.

2 Drawbacks of the Existing QDNN Neuron Architecture Design

We first reviewed the current QDNN works in the last decades based on their corresponding key quadratic neuron computation formats and summarized them in Table. 1. According to the way that how the second-order term of input X is introduced in each quadratic neuron, we can divide the current QDNN works into four types as shown in table: for $\mathbb{T}1$ design, each input X with size d will conduct out-product with $d \times d$ full-rank weight matrix[1]; for $\mathbb{T}2$ design, the second-order term is realized by directly squaring each input X; for $\mathbb{T}3$ design, the second-order term is coming from the square of a first-order neuron. Therefore, the size of weight parameter vector W_a in both $\mathbb{T}2$ and $\mathbb{T}3$ keeps unchanged compared to the original first-order neuron; in $\mathbb{T}4$ quadratic neuron, the second-order term is calculated by the Hadamard Product of the two first-order neurons with different sets of weight parameters.

Most previous QDNN works focused on theoretically proving the stronger non-linearity and approximation ability of the single quadratic neuron, therefore they only applied QDNNs on some very simple learning tasks with small network structures, such as one fully connection layer for XOR Gate simulation (Fan et al., 2018; Cheung & Leung, 1991; Redlapalli et al., 2003). However, by investigating the computation pattern of these existing QDNN works from the angle of practical usage, we identified several problems.

P1 ***Approximation Capability Issue:*** This issue is from $\mathbb{T}2$ and $\mathbb{T}3$ due to the insufficient trainable parameters. Specifically, as Table. 1 shows, compared to $\mathbb{T}1$ and $\mathbb{T}4$, $\mathbb{T}2$ and $\mathbb{T}3$ QDNN design only have one set of weight parameters

[1] The original format of the proposed quadratic neuron formulation in (Cheung & Leung, 1991; Milenkovic et al., 1996) is $\sum_{i=1}\sum_{j=1} W_{ij}x_ix_j$, which mathematically equals to X^TWX.

Table 1. The Overview of Current QDNN Works

Type	Neuron Format	Work Reference	Computation Complexity		Model Structure†	Library Usage	Issue
$\mathbb{T}1$	$f(X) = X^T W_a X + W_b X$	(Cheung & Leung, 1991)	$\mathcal{O}(n^2+n)$	$\mathcal{O}(n^2+n)$	1-layer	×	P2 P3 P4
		(Zoumpourlis et al., 2017)			1-layer	Torch	
$\mathbb{T}1$	$f(X) = X^T W_a X$	(Redlapalli et al., 2003)	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	1-layer	Matlab	P2 P3 P4
		(Jiang et al., 2019)			6-layer	TensorFlow	
		(Mantini & Shah, 2021)			4/18-layer	×	
$\mathbb{T}2$	$f(X) = W_a X^2$	(Goyal et al., 2020)	$\mathcal{O}(2n)$	$\mathcal{O}(n)$	2-layer	×	P1 P3
$\mathbb{T}3$	$f(X) = (W_a X)^2$	(DeClaris & Su, 1991)	$\mathcal{O}(2n)$	$\mathcal{O}(n)$	1-layer	×	P1 P3
$\mathbb{T}4$	$f(X) = (W_a X) \circ (W_b X)*$	(Bu & Karpatne, 2021)	$\mathcal{O}(3n)$	$\mathcal{O}(2n)$	5/10-layer	TensorFlow	P3
$\mathbb{T}1\&2$	$f(X) = X^T W_a X + W_b X^2$	(Milenkovic et al., 1996)	$\mathcal{O}(n^2+2n)$	$\mathcal{O}(n^2+n)$	1-layer	×	P2 P3 P4
$\mathbb{T}2\&4$	$f(X) = (W_a X) \circ (W_b X) + W_c X^2$	(Fan et al., 2018)	$\mathcal{O}(5n)$	$\mathcal{O}(3n)$	2-layer	Matlab	P3
Ours	$f(X) = (W_a X) \circ (W_b X) + W_c X$	This work	$\mathcal{O}(4n)$	$\mathcal{O}(3n)$	Various	*QuadraLib*	-

1. $X^T = \{x_1, x_2, ..., x_d\}$ is neuron's input and $f(X)$ represents neuron's output. For simplicity, we ignore bias b here.
2. *:in (Bu & Karpatne, 2021), it only introduces the quadratic ResNet design. We list the design's general format.
3. $\circ$ represents Hadamard product. †: The second-order layer in the QDNN.

while didn't introduce extra trainable parameters. According to (Radosavovic et al., 2019), the approximation capability is highly related to the model capacity, namely, with a larger amount of trainable parameters, the model will show a higher learning performance potential. In that case, $\mathbb{T}2$ and $\mathbb{T}3$ model design will have lower theoretical learning performance (such issue will be further validated with numerical experiments in Section 5.2).

P2* *Computation Complexity Issue: This issue is from $\mathbb{T}1$ since it introduces a full-rank weight matrix in each neuron, which is a 4D array $W \in R^{C*r^4*N*C}$ (where C, N, and r represent the input channel number, filter number, and kernel size). Therefore, if we let r^2 equals to n, the time or space complexity of $\mathbb{T}1$ neuron is at least $\mathcal{O}(n^2)$. Such complexity will exponentially increase with a larger model structure regarding depth and width, easily introducing out-of-memory issue during model training. For example, in (Mantini & Shah, 2021), the parameter size of the original first-order *ResNet* is only 0.2M while it dramatically increases to 128M for QDNN with same structure.

P3* *Converge Performance Issue: This issue is from $\mathbb{T}1$ to $\mathbb{T}4$ since we found that the second-order term in QDNNs will introduce critical gradient vanishing issue, impairing QDNN training convergence. Here, we use $\mathbb{T}4$ design $(W_a X)(W_b X)$ with plain network structure as an example. Assume X^k represents the input of the k^{th} layer. During back-propagation, the gradient for X_k can be formulated as:

$$g(X^k) = \frac{\partial \ell}{\partial X^L} \cdot \frac{\partial X^L}{\partial X^k} = \frac{\partial \ell}{\partial X^L} \cdot \prod_{i=k+1}^{L} X^i \prod_{i=k+1}^{L} ((W_a^i)^2 + (W_b^i)^2) \quad (1)$$

where ℓ indicates the loss value. We ignored activation function here since their gradients equal to one (when input larger than zero) during back-propagation. It is easily to find that: contrast to the traditional first-order DNN which only include weight parameters in their gradients, gradients in QDNNs contain both weights $\prod_{i=k+1}^{L}((W_a^i)^2 + (W_b^i)^2)$ and outputs from other layers $\prod_{i=k+1}^{L} X^i$. Since activation X_i objects to norm distribution $X_i \sim N(0,1)$, $\prod_{i=k+1}^{L} X^i$ is obviously much smaller than 1. Moreover, with depth increase, $\prod_{i=k+1}^{L} X^i$ will gradually approach to zero, causing gradient vanishing. It should be noted that the same issue occurs for other kinds of second-order term design as well, which proves that it is the common issue for all QDNNs.

P4* *Implementation Feasibility Issue: Implementation feasibility indicates whether the designed QDNN is development-friendly with the current DNN libraries. As shown in Table. 1, only a few of the existing QDNN works mentioned that they could be implemented on some DNN libraries, *e.g.*, Matlab and Torch, while most of them didn't specify the capability of being supported by the current DNN libraries. Specifically, the currently DNN frameworks are all specifically designed for the first-order DNNs with formation $X^T W$, where W is $n \times 1$ when assuming the size of X is $n \times 1$. On the contrary, $\mathbb{T}1$ neuron has a bilinear formation $X^T W X$ where the size of W is $n \times n$. Therefore, such neuron cannot be simply composed by two consecutive linear neurons. We need to rewrite the entire convolution operation to add extra multiplication between W and the second X. Therefore, such type of neurons are not implementing-friendly.

P5* *Structure Design Issue: The existing QDNN works suffer from two sub-problems in terms of QDNN structure

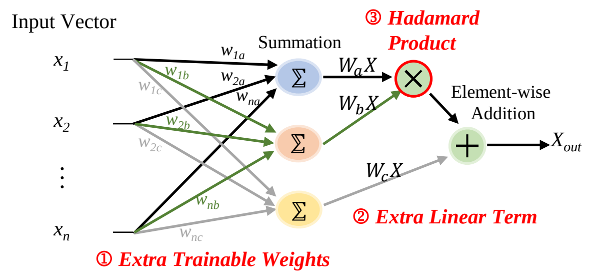


Figure 3. The New Quadratic Neuron Design

design: first, since they propose distinct quadratic neuron architectures and most of are not capable for the current DNN libraries, it is difficult to reproduce these QDNN design; second, most of them merely leverage shallow model structures *e.g.*, one or two layers, to verify the designed neuron capability, while such shallow model structures are not practicable for common-used learning applications such as image classification or object detection. However, identifying an optimal network structure for a given learning scenario usually needs to introduce significant design efforts, such as Network Architecture Search (NAS) in the first-order DNN design.

***P6* Memory Usage Issue:** As Table. 1 shows, some existing QDNN design such as (Cheung & Leung, 1991; Micikevicius et al., 2017; Fan et al., 2018) introduce more intermediate parameters during training. During training process, QDNNs needs to be executed forwardly to get loss and other intermediate parameters such as activation. These intermediate parameters needs to be cached in computing unit's memory (*e.g.* GPU) until finishing gradients computing during backward, thereby will introduce high memory cost, eventually causing memory inefficient-usage problem.

In the rest of this paper, we addressed the above six problems from two main perspectives:

- From theoretical **neuron architecture optimization perspective**, we proposed a new quadratic neuron design to solve problems *P1* to *P4* (Section 3).
- From practical **model construct and training optimization perspective**, we proposed *QuadraLib*, which can generate optimal QDNN model structure for the practical deployment (*P5*) and address the inefficient memory usage problem (*P6*) (Section 4).

3 QDNN Neuron Architecture Optimization

3.1 New Neuron Architecture Design

Through the above analysis, we identified that the existing QDNN works show one or more drawbacks due to their neuron computation format design. Fortunately, from the above performance analysis, we can obtain several ***design insights:*** 1) The second-order term in QDNN is expected to introduce extra trainable parameters instead of just using the original one set of weight parameters in the first-order neuron. 2) Besides second-order term, the new quadratic neuron should include other term to prevent the gradient vanishing issue. 3) Since the original out-product between weight matrix and input vector introduces intensive computation workload as well as memory cost, replacing it with Hardmard product can significantly improve designed neurons' computation efficiency. 4) To achieve similar implementation feasibility as the first-order DNNs, the proposed quadratic neuron should be easily assembled via multiple first-order neurons.

Based on the above design insights, we optimize the second-order term and introduce new linear term into quadratic neuron design, and propose a new quadratic neuron format (as shown in Fig. 3), which can comprehensively address the identified neuron design issue in the current QDNN works. The proposed new quadratic neuron is formulated as:

$$f(X)=(W_aX)\circ(W_bX)+(W_cX). \tag{2}$$

It should be noticed here, our proposed neuron design is similar to (Fan et al., 2018). (Fan et al., 2018) mainly focus on improving the approximation capability by introducing two types of second-order term. However, as discussed in *P3*, such design will amplify gradient vanishing issue, hurting its converge performance. On the contrary, we address this issue by replacing W_cX^2 with a linear term W_cX. We will further analyze the superiority of our design in terms of approximation capability, converge performance, computation complexity, and implementation feasibility.

3.2 Theoretical Performance Analysis

Extra Weights and Linear Term for Approximation Capability (P1) Improvement: The approximation capability improvement of our design is achieved from two aspects: first, similar to $\mathbb{T}4$ design, our design introduces two sets of weight to form the second-order term, providing extra trainable parameters (shown as ① in Fig. 3). Second, we add an extra linear term in our design (shown as ② in the figure). Specifically, let's use a L-layer QDNN with plain structure (e.g. *VGG-16*) as an example. Such model can be formulated without/with linear term W_cX^k as:

$$Y=\prod_{l=1}^{L}W_a^lW_b^lX^{2^L}\longrightarrow\ Y=\sum_{i=1}^{2^L}\prod_{l=1}^{L}W_a^lW_b^lW_c^lX^i, \tag{3}$$

where $\prod_{l=1}^{L} W_a^lW_b^l$ represents the product over a set of weight W_a^l and W_b^l. The first high-order polynomial only includes a 2^L order term regarding input X. According to Chebyshev approximation (Hernández, 2001), the approximation error introduced by such polynomial is approached to a X^{2^L-1} term. On the contrary, the second polynomial includes multiple terms with X from the first-order to 2^L order, which means its approximation error is approached to zero. Therefore, by adding linear term, our neuron design can significantly improve approximation capability.

Hadamard Product for Computation Complexity (P2) Optimization: In order to avoid huge computing workload and memory coast of $\mathbb{T}1$ design, the original out-product between weight matrix and input vector is replaced by a Hadamard product (③ in figure), which will significantly improve designed neurons' computation efficiency from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$ regarding time/space complexity. In Section 3.3, we will further discuss how we boost computation efficiency for QDNN during training through removing unnecessary intermediate parameters.

Linear Term for Converge Performance (P3) Enhancement: As discussed in Section 2.2, all kinds of second-order terms will introduce critical gradient vanishing issue, decreasing training effectiveness. On the contrary, we identified that the added first-order term (②) in our neuron design can solve this issue. In the first-order DNN model, we leverage identity mapping ($h(X^{k-1})$ (He et al., 2016b)) to provide an extra term $\frac{\partial \ell}{\partial X^L}$ (the gradient of loss to the last layer) during each layer's gradient calculation. Such extra term ensures that information can propagate directly to each shallow layers. We found that the first-order term $W^c x^k$ in our design not only can improve neuron approximation capability, but also can work as the identity mapping to solve the gradient vanishing issue. Specifically, the quadratic layer with our design neuron is: $X^{k+1} = (W_a^k X^k W_b^k X^k) + W_c^k X^k$. During back-propagation, the gradient of X^k is:

$$\frac{\partial X^{k+1}}{\partial X^k} = X^k(W_a^2 + W_b^2) + W_c, \tag{4}$$

where the additional term W_c is the main contribution for the gradient calculation in the original first-order DNNs. It can cooperate with Batch Normalization layer and activation function (*e.g.* ReLU) to prevent gradient vanishing issue[2].

First-order Neuron Combination for Implementation Feasibility (P4) Improvement: In order to have high implementation feasibility, our designed quadratic neuron is the combination of three traditional first-order neurons (shown in Fig. 3): the first two conduct Hadamard Product and then the product will sum with the third one. Different with $\mathbb{T}1$ design that need to create the customization convolution operation, all the operations defined in our neuron are supported by the current DNN libraries, without introducing extra implementation efforts. Moreover, these operations already be optimized by the DNN libraries from compiler-level, guaranteeing an optimal computing performance.

[2] It may still meet model degeneration issue (Orhan & Pitkow, 2017) for deep plain network structure, which can be further solved by using our model construction method or *ResNet* structure.

4 QuadraLib for QDNN Design Exploration

In the last section, we proposed a new quadratic neuron design to address previous drawbacks (*P1* to *P4*) from theoretical aspect. According to library capability analysis in the last column of Table. 1, there lacks a library to support the implementation of all kinds of the existing QDNNs. To further address (*P5* *P6*) and conduct in-depth numerical analysis as well as achieve design exploration on QDNNs, we proposed *QuadraLib*, *an extended PyTorch-based Python library to provide a set of complementary components for QDNNs.*

4.1 Library Overview

Fig. 4 shows our *QuadraLib*'s design overview:

- In Model Level, *QuadraLib* defines a set of *encapsulated layer modules* to support all existing quadratic neuron types. Based on that, *QuadraLib* provides QDNN model construction flow and design insights to guarantee a decent accuracy performance of the built QDNN model. Furthermore, an auto-builder is proposed to extend QDNN design by converting any first-order model to a corresponding QDNN version.
- In Training/Inference Level, *QuadraLib* leverages a memory profiler[3] to monitor the memory cost of the generated QDNN model during the training phase. A *Quadratic Optimizer* is developed that applied a *hybrid back-propagation scheme* to QDNN model training, which further enhances the training efficiency of QDNNs (Section 4.3).
- In Application Level, *QuadraLib* also provides model analysis tools such as *activation and weight/gradient distribution visualization*. Leveraging such visualization tools, we reveal the distinct feature extraction capability of QDNN vs. first-order DNNs (Section 5).

4.2 QDNN Model Construction

As aforementioned, the existing QDNN works fail to provide a unified neuron design, QDNN structure design, as well as implementations (P5). *QuadraLib* addresses this issue by providing a series of QDNN model construction help: first, it defines the encapsulated quadratic layer modules as the fundamental units of model construction. Next, we can either use the quadratic layer module to construct a full QDNN model from scratch or combine it with other first-order layer modules in the pre-defined construction functions. Furthermore, we proposed an auto-builder which leverages knowledge of the existing first-order DNN to extend the QDNN model structure design.

[3] Currently, we use default memory profiler in Pytorch.

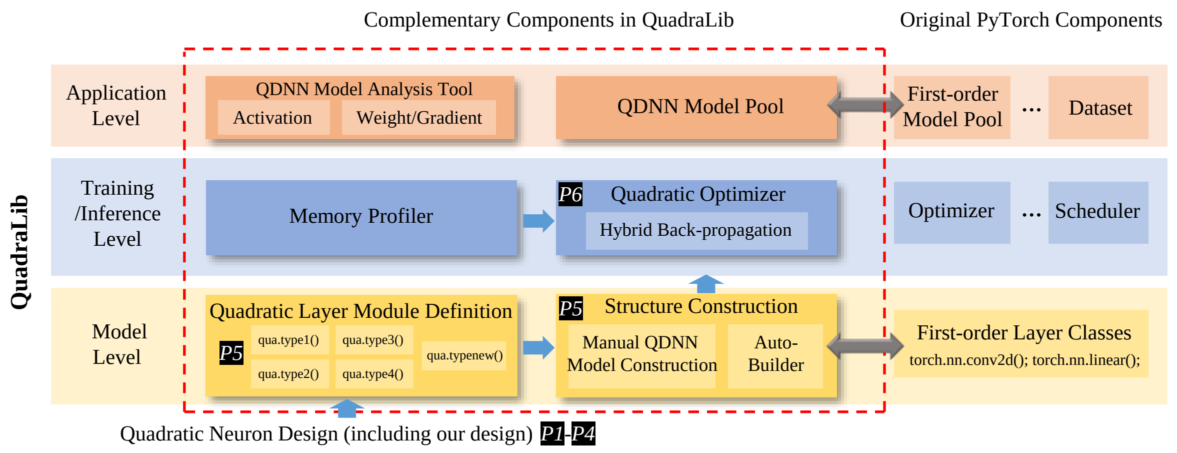


Figure 4. QuadraLib Architecture Design

Encapsulated Quadratic Layer Module: Similar to construct the first-order DNNs in PyTorch, QDNN construction is expected to be assembled by various quadratic layer modules. Therefore, *QuadraLib* first defines and encapsulates the quadratic layer modules as fundamental units to alleviate the manual QDNN construction complexity.

Specifically, various quadratic layers including our proposed one are written as new $nn.module$ separately. Once creating a QDNN, the desired quadratic layer can be introduced as $qua.type\#()$ (here,# represents different types of quadratic layer). Since our layer module is inherited from PyTorch, it supports flexible layer parameters setting such as kernel-size, padding/stride, depth-wise/point-wise. By doing that, the customized quadratic layer and the original first-order layer share the same instantiation and computation graph definition methodology, thereby can easily replace any original first-order layer module in a given model to create a QDNN version.

Manual QDNN Model Construction: Since our proposed quadratic layer modules share the same characteristics with the ordinary first-order layer modules in PyTorch, the model construction process in *QuadraLib* is aligned with PyTorch, which is user-friendly. Specifically, during model construction, we first created a structure configuration file. By defining the model structure parameters such as depth and width, such configuration file can guide the structure design. Here are some insights for QDNN model design during configuration file definition: 1) since quadratic neuron has higher capability, the depth of QDNN can be reduced, thereby decrease the model computation cost and also avoids the potential gradient vanishing or model degeneration issues discussed before; 2) since second-order term will generate extreme values, batch-normalization layer is significantly important for QDNN to regulate the output activation values; 3) QDNNs with small network structures don't need activation functions (*e.g.* ReLU) due to the high capability of quadratic neuron. However, when QDNN depth increases, activation functions are important since they can prevent gradient vanishing as discussed in Section 3.2. Secondly, once we determined configuration file, we will insert quadratic layer modules into a construction function and the entire QDNN model can be defined as a layer sequence. Here we demonstrated a code piece example of our construction function with *VGGNet* structure in the following figure.

```
// A code piece example of our
    construction function.
import QuadraNeuron as quad
def model_construct(self, cfg):
    ...
  for v in cfg:
    layers += [qua.type1(in_channels, v),
        nn.ReLU()]
    in_channels = v
  return nn.Sequential(layers)
```

It is easily to see that our quadratic layer module can directly inserted into construction function with other layer types (*e.g.* $nn.ReLU()$), which is flexible and convenient. At last, users can easily import the construction function into their code and instantiate a corresponding QDNN model.

QDNN Auto-Builder: For an existing learning task, manually designing a QDNN model from scratch needs a lot of prior domain experience and can involve significant effort, such as detector backbones. Therefore, besides to manually construct QDNN from scratch, another effective approach for efficient QDNN construction is to leverage the existing first-order DNN model pools, which already include various sophisticated pre-defined first-order DNN structure for different learning tasks. Therefore, *QuadraLib* provides a QDNN auto-builder to convert the first order DNN to a corresponding QDNN without designing from scratch.

To guarantee the converted QDNN model has optimal performance (*e.g.* prediction accuracy and model depth), we conducted two operations: layer replacement and heuristic-based layer reduction. Specifically, we first iteratively re-

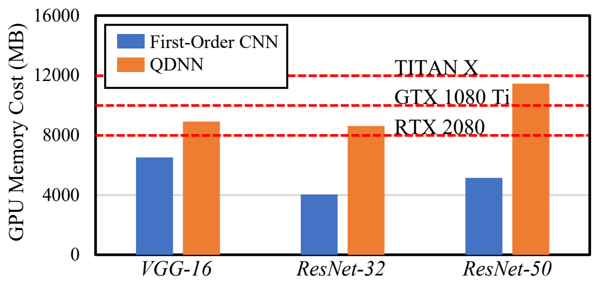


Figure 5. GPU Memory Cost Evaluation

placed the first-order layer module in the original construction function with our quadratic layer module from shallow layer to deep layer. Secondly, we identified which layers in the converted model can be removed to achieve a more compacted model structure. In order to achieve that, we used layer performance indicator from (Xu et al., 2019), which can be defined as:

$$RI = \frac{P(M_{par})P(T_{lat})}{\triangle Acc}, \tag{5}$$

where $P(M_{par})$ and $P(T_{lat})$ represent the ratio of the target layer to the entire model in terms of parameter amount and computation workload. $\triangle Acc$ represents the accuracy drop when removing the target layer. The layer with high parameter size and computation workload while low accuracy contribution will have a high RI value, which means that it can be removed firstly during our heuristic-based layer reduction. Based on the above two operations, auto-builder can finalize the optimal model structure for QDNN, providing extra model structure sources for QDNN model pool in Application Level of *QuadraLib*.

4.3 QuadraLib Training Optimization

Based on the analysis in Section 2, we identified the current QDNNs still suffer from inefficient memory usage issue (P6) since they will generate more intermediate parameters during training phase. *QuadraLib* address this problem in Training/Inference Level via proposing a quadratic optimizer. Specifically, the model will be first evaluated by a *memory profiler* to exam whether it is under potential out-of-memory risk. Fig. 5 shows the GPU memory usage of the first-order DNNs and $\mathbb{T}$2&4 QDNN (Fan et al., 2018) monitored by our *memory profiler*. The two evaluated models share the same network structure and batch size is set as 512. It is clearly to see that the first-order DNN training can be satisfied by most widely-used GPUs while QDNNs with same layers will introduce more memory cost and may cause out-of-memory issue during training phase. Once *memory profiler* identified inefficiency memory usage issue, *QuadraLib* will optimize the training via a proposed hybrid back-propagation scheme.

Hybrid Back-propagation for Memory Efficiency: In the most DNN libraries, gradient calculation during back-propagation is supported by Auto-Differentiation (AD) with

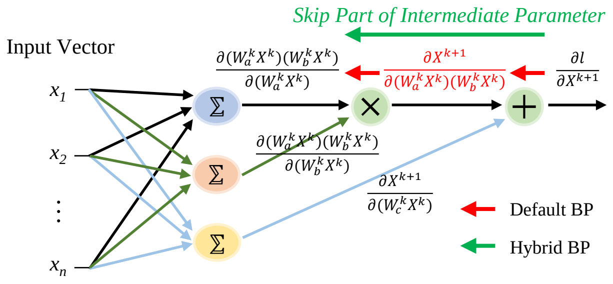

Skip Part of Intermediate Parameter
Input Vector
x_1
x_2
⋮
x_n
$\frac{\partial(W_a^kX^k)(W_b^kX^k)}{\partial(W_a^kX^k)}$
$\frac{\partial X^{k+1}}{\partial(W_a^kX^k)(W_b^kX^k)}$
$\frac{\partial l}{\partial X^{k+1}}$
$\frac{\partial(W_a^kX^k)(W_b^kX^k)}{\partial(W_b^kX^k)}$
$\frac{\partial X^{k+1}}{\partial(W_c^kX^k)}$
Default BP
Hybrid BP

Figure 6. Back-propagation in QDNN Training

reverse-mode (Baydin et al., 2018), which means the partial derivatives will backwardly go through each layer from output to input. The red arrow in Fig. 6 shows the default back-propagation process for our quadratic layer. Assume we want to update W_a^k, its gradient can be calculated according to the chain rule as:

$$\frac{\partial \ell}{\partial W_a^k} = \frac{\partial \ell}{\partial X^{k+1}} \cdot \frac{\partial X^{k+1}}{\partial (W_a^k X^k)(W_b^k X^k)} \cdot \frac{\partial (W_a^k X^k)(W_b^k X^k)}{\partial (W_a^k X^k)} \cdot \frac{\partial (W_a^k X^k)}{\partial W_a^k}. \tag{6}$$

During this process, all the intermediate parameters needs to be stored on GPU memory. Compared to AD, Symbolic Differentiation (SD) can derive the symbolic expression of each parameter's partial gradients before the back-propagation. Therefore, training process doesn't need to save every intermediate parameters on the memory. For example, Eq. 6 can be simplified as:

$$\frac{\partial \ell}{\partial W_a^k} = \frac{\partial \ell}{\partial X^{k+1}} \cdot (W_b^k X^k) \cdot X^k, \tag{7}$$

which means that we don't need intermediate term $\frac{\partial X^{k+1}}{\partial (W_a^k X^k)(W_b^k X^k)}$ during SD process. Therefore, we proposed a hybrid back-propagation scheme to decrease the memory usage during QDNN training, which is the combination of AD and SD. Indicated by green arrow in Fig. 6, for each quadratic layer, when obtaining input values and output values after forward process, we use SD to calculate the corresponding gradients of weight parameters and input during back-propagation. Therefore, only necessary intermediate parameters will be stored before the backward. On the other hand, since the gradients of other layers such as batch normalization are not easy to formulate in SD, we can still leverage AD to calculate their values.

Quadratic Optimizer Implementation: In the original PyTorch library, we can simply leverage *model.backward()* to conduct AD with gradient calculation. In order to achieve our hybrid back-propagation, we need to modify the backward computing flow inside each quadratic layer with SD. Therefore, we provide another version of our quadratic layer. Instead of be defined as *nn.module* which can only rely the default AD, we build it by inheriting *torch.autograd.Function* so that we can manually define both the forward and backward computing process inside the quadratic layer as individual *def forward* and *def backward*. Meanwhile, in *def forward*, we use gradient checkpointing

Table 2. Performance Evaluation of Various Quadratic Neuron Designs with Deep Network Structures.

	VGG-8		VGG-16		ResNet-32	
	Train	Test	Train	Test	Train	Test
$\mathbb{T}2$	91%	81%	10%	10%	99%	89%
$\mathbb{T}3$	93%	85%	10%	10%	99%	89%
$\mathbb{T}4$	94%	85%	10%	10%	99%	89%
$\mathbb{T}4$+Identity	95%	86%	99%	91%	99%	90%
Ours	**97%**	**88%**	**99%**	**94%**	**99%**	**93%**

(*torch.utils.checkpoint.checkpoint*) to make sure quadratic layer will not store these intermediate parameters after the forward process.

Besides model construction and training optimization, *QuadraLib* also provides other components in Application Level. *e.g.* model analysis tools (discussed in Section 5.)

5 Experimental Evaluation

In this section, we first conduct experiments to evaluate the effectiveness of our design component in *QuadraLib*, including the new quadratic neuron design and hybrid-BP based quadratic optimizer. Overall experiments are then conducted on several representative deep learning tasks (Image Classification, Object Detection, GAN-based Image Generation) to demonstrate the generality and scalability of *QuadraLib* in various of tasks. For Image Classification, three model structures are used as testing targets, including *VGG-16* (Simonyan & Zisserman, 2014), *ResNet-32* (He et al., 2016a), *MobileNetV1* (Howard et al., 2017). Meanwhile, three datasets are included: *CIFAR-10*, *CIFAR-100*, and *Tiny-ImageNet*. For Image generation, one baseline network structure, namely, *SNGAN* (Miyato et al., 2018) is evaluated on *CIFAR-10*. We then convert original *SNGAN* into our QDNN and compare its performance with baseline and other state-of-the-art works. For Object Detection, Single Shot MultiBox Detector (SSD) (Liu et al., 2016) framework with *PASCAL VOC2007* and *PASCAL VOC2012* dataset are used as test model and corresponding dataset.

5.1 Design Components Evaluation

Convergence Benefits of New Neuron Design: We first evaluated our new neuron design. The results are shown in Table. 2. Specifically, we use $\mathbb{T}2$, $\mathbb{T}3$, $\mathbb{T}4$ and $\mathbb{T}4$+Identity ($W_aXW_bX + X$) as baselines[4] and tested different neuron designs in *VGG-8*, *VGG-16*, and *ResNet-32* on *CIFAR-10*. Based on the results, we can found that for plain structures (VGGs), $\mathbb{T}2$,$\mathbb{T}3$,$\mathbb{T}4$ ***fail to achieve convergence once the depth is increased from 8 to 16***. Only by adding identity mapping or liner term in the quadratic neuron can help the model to achieve convergence.

[4]$\mathbb{T}1$ is not practical to deployment with deep network structure due to high memory cost.

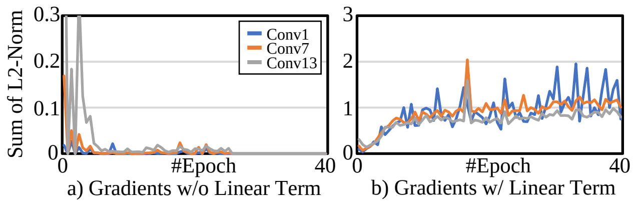


Figure 7. Gradient Evaluation for QDNN with *VGG-16* Structure

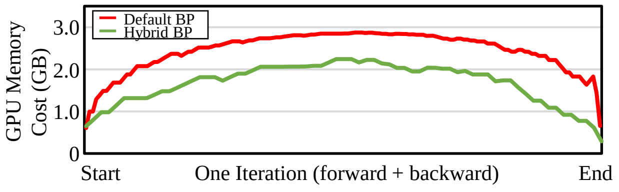


Figure 8. GPU Memory Cost Results with Hybrid-BP

By contrast, our neuron design with linear term shows consistently better convergence performance with higher accuracy. Moreover, Fig. 7 shows the gradient value comparison for $\mathbb{T}3$ design without/with linear term, which is measured by our gradient distribution visualization toolset. We can easily find: without linear term or identity mapping function, gradients in the shallow layer (Conv1) quickly approach to zeros in the first few epochs and thereby weight parameters stop updating. By contrast, our neuron design by adding the linear term solves the gradient vanishing issue, enabling gradients to keep as meaningful values during training, thus achieving better training effectiveness.

Memory Savings of Hybrid-BP Quadratic Optimizer: We then evaluate the performance of our quadratic optimizer by hybrid back-propagation on a ConvNet with 5 convolution layers. The batch size is set as 256 and the input size is 32×32. We leverage *torch.cuda.memory_allocated()* to measure the GPU memory occupied by QDNN training including forward and backward. The evaluation results are shown in Fig. 8. First, the memory cost during forward is constantly increased (indicated by red line) since intermediate parameters are gradually stored and they will be release from GPU memory during backward. Therefore, the required minimum GPU memory space is around 3GB. By contrast, when using our hybrid back-propagation (indicated by green line), the required GPU memory cost decreases to 2.2 GB, which shows a 26.7% memory cost saving and higher training memory efficiency.

5.2 Evaluation for Image Classification

Experiment Setup: For all three datasets, we use SGD optimizer and CosineAnnealing scheduler (Loshchilov & Hutter, 2016) with an initial learning rate of 0.1 and the training epoch is set as 200. The batch size for *CIFAR* and *Tiny-ImageNet* is set as 256 and 128, respectively. The first-order DNNs are used as baselines and two other state-

Table 3. The Performance Evaluation on *CIFAR*

	Model	#Layer/#Block	#Param	Train Time/Batch	Train Memory	Test Time/Batch	Accuracy (CIFAR-10)	Accuracy (CIFAR-100)
VGG-16	First-order†	13 CL*	1.47E+7	28ms	4.4GB	9ms	93.01%	73.11%
	(Fan et al., 2018)	7 CL	1.20E+7	56ms	3.3GB	19ms	92.48%	72.47%
	(Bu & Karpatne, 2021)	7 CL	0.80E+7	43ms	3.2GB	14ms	93.22%	72.33%
	QuadraNN (no auto-builder)	13 CL	4.41E+7	93ms	6.3GB	42ms	93.31%	73.23%
	QuadraNN	7 CL	1.20E+7	53ms	3.3GB	18ms	**94.03%**	**73.72%**
ResNet-32	First-order	BS*:[5, 5, 5]	4.80E+5	28ms	2.9GB	11ms	91.80%	68.59%
	(Fan et al., 2018)	BS:[2, 2, 2]	3.93E+5	33ms	2.4GB	16ms	91.92%	67.83%
	(Bu & Karpatne, 2021)	BS:[2, 2, 2]	2.62E+5	31ms	2.1GB	14ms	91.23%	67.70%
	QuadraNN (no auto-builder)	BS: [5, 5, 5]	14.24E+7	89ms	5.3GB	33ms	91.98%	69.98%
	QuadraNN	BS:[2, 2, 2]	3.93E+5	32ms	2.2GB	15ms	**92.38%**	**70.10%**
MobileNet-V1	First-order	13 DW‡	4.22E+6	27ms	4.8GB	10ms	**92.97%**	**70.35%**
	(Fan et al., 2018)	8 DW	3.53E+6	35ms	3.6GB	11ms	91.59%	68.31%
	(Bu & Karpatne, 2021)	8 DW	2.97E+6	25ms	3.2GB	8ms	92.18%	69.77%
	QuadraNN (no auto-builder)	13 DW	12.66E+7	112ms	7.6GB	47ms	91.81%	69.93%
	QuadraNN	8 DW	3.53E+6	32ms	3.3GB	10ms	92.44%	70.03%

First-order†: the original *VGG-16* with linear neurons. CL*: Convolution layers. DW‡: a pair of depth-wise and point-wise convolution layers.
BS*: block numbers. For example, [5, 5, 5] means the network has 3 blocks and each block includes 5 residual connections.

Table 4. The Performance of *VGGNet* on *Tiny-ImageNet*

Model	#Layer	Accuracy
First-order	13 CL	62.38%
QuadraNN	7 CL	**62.66%**
QuadraNN (no ReLu)	7 CL	59.43%

Table 5. The QDNN Performance Evaluation for Image Generation

Model	IS(↑)	FID(↓)
CQFG(Lucas et al., 2019)	8.10	18.60
EBM(Du & Mordatch, 2019)	6.78	38.2
SNGAN(Miyato et al., 2018)	8.06±0.10	19.06±0.50
PolyNet (Chrysos et al., 2020)	8.49±0.11	16.79±0.81
QuadraNN	**8.58±0.07**	**16.53±0.73**

IS: Inception Score. FID: Frechet Inception Distance.

of-the-art QDNN designs are evaluated, which are indicated as Design 1 (Fan et al., 2018) and Design 2 (Bu & Karpatne, 2021). The model generated from the proposed auto-builder is name as "**QuadraNN**". In order to validate auto-builder's effectiveness, we naively replace the first-order DNN with quadratic neuron in each layer and name as "QuadraNN (no auto-builder)". All tests are running 10 times and we calculate the mean values.

Results Analysis: The evaluation results are shown in Table. 3. We can found that our design (QuadraNN) can always show better accuracy performance than baseline and other designs on both *CIFAR-10* and *CIFAR-100*. Specifically, compared to the original first-order DNN with *VGG-16* structure, QuadraNN only needs 7 convolution layers, which achieves 18.36% parameter size and 11.30% GPU memory cost reduction. Meanwhile, QuadraNN shows 1.01% and

a) CIFAR-10 Images Generated By SNGAN
b) CIFAR-10 Images Generated By QuadraNN
Figure 9. Image Generation Example

0.61% accuracy improvement. Moreover, if we directly generate QDNN without applying converter, the model size is significant large with higher consumption and latency. When using converter, we can achieve 4×, 3×, 1.5× saving in terms of parameter size, training time cost, and memory occupancy. For *ResNet-32* structure, QuadraNN only need 2 skipping connections structure in each residual block, thereby decrease 23.90% parameter size and 6.89% GPU memory cost. On the contrary, QuadraNN shows 0.58% and 1.51% accuracy improvement. Similarly, using our converter can achieve 4×, 5×, and 1.6× saving in terms of parameter size, training time cost, and memory occupancy. For *MobileNet-V1* structure, QuadraNN only needs 8 pair of depth-wise/point-wise convolution layers (DWs) while the original first-order DNN needs 13 DWs. Therefore, QuadraNN achieves 31.25% GPU memory saving while achieve a comparable accuracy performance than the first-order DNN. Compared to QuadraNN (no converter), our model also shows better performance with 4×, 3×, and 2× saving in terms of parameter size, training time cost, and memory occupancy. Moreover, when increasing the image size, QuadraNN still can achieve the better perfor-

Table 6. The Detection Performance Evaluation on PASCAL VOC2007

Model	Pre-trained	plane	bike	bird	boat	bottle	bus	car	cat	chair	⋯	sheep	sofa	train	TV	**Total**
1st order	×	0.63	0.61	0.33	0.37	0.12	0.64	0.73	0.59	0.27	⋯	0.51	0.52	0.67	0.53	0.52
QuadraNN	×	0.76	0.80	0.66	0.65	0.36	0.82	0.84	0.84	0.53	⋯	0.70	0.77	0.86	0.71	**0.73**
1st order	√	0.78	0.83	0.76	0.72	0.45	0.87	0.86	0.88	0.59	⋯	0.79	0.79	0.86	0.74	0.76
QuadraNN	√	0.78	0.84	0.78	0.73	0.47	0.88	0.86	0.89	0.63	⋯	0.79	0.82	0.89	0.76	**0.78**

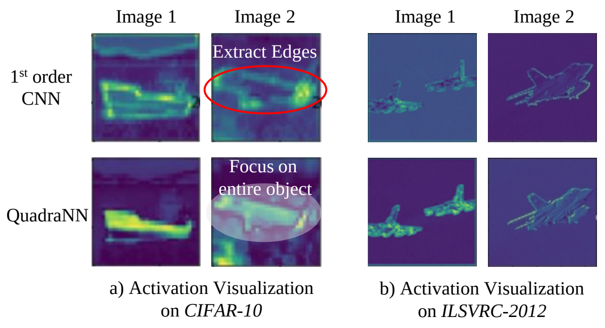


Image 1
Image 2
Image 1
Image 2
1st order CNN
Extract Edges
QuadraNN
Focus on entire object
a) Activation Visualization on *CIFAR-10*
b) Activation Visualization on *ILSVRC-2012*

Figure 10. Activation Attention via Our Visualization Tool

mance trade-off between efficiency and accuracy than the first-order DNNs, which is shown in Table. 4.

5.3 Evaluation for Image Generation

Experiment Setup: We adopt *SNGAN* (Miyato et al., 2018) as the baseline structure and test it on *CIFAR-10*. The original generator in *SNGAN* has 3 residual blocks. For our QuadraNN, we convert each convolution layer in generator into our quadratic layer. The structure of discriminator and other hyper-parameters are same as settings in (Miyato et al., 2018). Three others state-of-the-art works are compared (Lucas et al., 2019; Du & Mordatch, 2019; Chrysos et al., 2020). Similar to (Chrysos et al., 2020), each network was run for 10 times and their mean and variance are recorded. We use Inception Score (IS) (Salimans et al., 2016) and Frechet Inception Distance (FID) (Heusel et al., 2017) as performance metrics. A higher IS score and a lower FID score indicate a better generation performance.

Results Analysis: The results are shown in Table. 5. Compared to the baseline *SNGAN* and other first-order DNNs, QDNN models (PolyNet and Ours) demonstrate a higher IS but lower FID scores, which indicates that they have performance superiority. Moreover, since our quadratic neuron design has a theoretical higher approximation capability, thereby shows a better performance than PolyNet.

5.4 Evaluation on Object Detection

Experiment Setup: As aforementioned, we use SSD as object detector. The backbone of SSD is *VGG-16*. We use VOC2007 trainval and VOC2012 trainval as training set and use VOC2007 as test set. The initial learning rate is 10^{-3} and will decrease 10 times at iteration [80000, 100000]. In the original SSD, backbone network *VGG-16* use a pre-trained model parameters from *ILSVRC-2012*. In order to comprehensively evaluate the detection performance of QuadraNN, we have two settings: 1) We initialize both first-order and second-order *VGG-16* with Kaiming initialization (He et al., 2015) without pre-training. 2) We initialize the model parameters by copying the parameter from pre-trained QuadraNN on *ILSVRC-2012*.

Results Analysis: Table. 6 shows the evaluation results. For situation of training without pre-training, QuadraNN demonstrate approximate 50% detection performance improvement than the ordinary first-order DNN. Such performance can even compete the first-order DNN with pre-training model parameters. For pre-training setting, QuadraNN still shows a better detection performance with 0.02 mAP improvement. We believe the superiority of QDNN in object detection is due to the difference characteristic of feature extraction between first-order layer and second-order layer. In order to validate our assumption, we leverage our activation attention visualization tool to compare the feature extraction results for the first layer on *CIFAR-10* and *ILVSRC-2012*, which is shown Fig. 10. We can get an interesting finding: the traditional first-order convolutional layer more focuses on edges, including edges of objects and background. On the contrary, quadratic layer can ***extract the entire objects*** (shown as white circle area). This feature extraction characteristic will highly contribute to object detection task.

6 Conclusion and Discussion

In this paper, we comprehensively reviewed the existing QDNN architecture and proposed a new quadratic neuron design with theoretical effectiveness and efficiency analysis. We further developed "*QuadraLib*", a QDNN library to support QDNN design with model structure construction and training optimization. The extensive experiments from multiple learning tasks demonstrate that our designed QDNN has superiority compared to the first-order DNNs and other existing QDNN works. Furthermore, from our theoretical analysis and numerical experiments, we believed that QDNNs show a significant potential on learning tasks which highly depends on extracted objects, such as object detection, segmentation, and position recognition. By applying QDNNs on these scenarios, the recognition process will more focus on the important objects while ignoring the unimportant background, thus is expected to achieve a better performance improvement gain.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pp. 265–283, 2016.

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., and Siskind, J. M. Automatic differentiation in machine learning: a survey. *Journal of machine learning research*, 18, 2018.

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. Theano: A cpu and gpu math compiler in python. In *Proc. 9th Python in Science Conf*, volume 1, pp. 3–10, 2010.

Brooks, D., Schwander, O., Barbaresco, F., Schneider, J.-Y., and Cord, M. Second-order networks in pytorch. In *International Conference on Geometric Science of Information*, pp. 751–758. Springer, 2019.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

Bu, J. and Karpatne, A. Quadratic residual networks: A new class of neural networks for solving forward and inverse problems in physics involving pdes. *arXiv preprint arXiv:2101.08366*, 2021.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

Cheung, K. and Leung, C. Rotational quadratic function neural networks. In *[Proceedings] 1991 IEEE International Joint Conference on Neural Networks*, pp. 869–874. IEEE, 1991.

Chou, E., Beal, J., Levy, D., Yeung, S., Haque, A., and Fei-Fei, L. Faster cryptonets: Leveraging sparsity for real-world encrypted inference. *arXiv preprint arXiv:1811.09953*, 2018.

Chrysos, G. G., Moschoglou, S., Bouritsas, G., Panagakis, Y., Deng, J., and Zafeiriou, S. P-nets: Deep polynomial neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 7325–7335, 2020.

Collobert, R., Kavukcuoglu, K., and Farabet, C. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

DeClaris, N. and Su, M.-c. A novel class of neural networks with quadratic junctions. In *Conference Proceedings 1991 IEEE International Conference on Systems, Man, and Cybernetics*, pp. 1557–1562. IEEE, 1991.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, 2019.

Du, Y. and Mordatch, I. Implicit generation and generalization in energy-based models. *arXiv preprint arXiv:1903.08689*, 2019.

Fan, F., Cong, W., and Wang, G. A new type of neurons for machine learning. *International journal for numerical methods in biomedical engineering*, 34(2):e2920, 2018.

Ganesh, Y., Singh, R. P., and Murthy, G. R. Pattern classification using quadratic neuron: An experimental study. In *2017 8th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pp. 1–6. IEEE, 2017.

Garimella, K., Jha, N. K., and Reagen, B. Sisyphus: A cautionary tale of using low-degree polynomial activations in privacy-preserving deep learning. *arXiv preprint arXiv:2107.12342*, 2021.

Gilad-Bachrach, R., Dowlin, N., Laine, K., Lauter, K., Naehrig, M., and Wernsing, J. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International conference on machine learning*, pp. 201–210. PMLR, 2016.

Goyal, M., Goyal, R., and Lall, B. Improved polynomial neural networks with normalised activations. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8. IEEE, 2020.

He, K., Zhang, X., Ren, S., and Sun, J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.

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

He, K., Zhang, X., Ren, S., and Sun, J. Identity mappings in deep residual networks. In *European conference on computer vision*, pp. 630–645. Springer, 2016b.

Hernández, M. Chebyshev's approximation algorithms and applications. *Computers & Mathematics with Applications*, 41(3-4):433–445, 2001.

Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B., and Hochreiter, S. Gans trained by a two time-scale update rule converge to a local nash equilibrium. *Advances in neural information processing systems*, 30, 2017.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pp. 675–678, 2014.

Jiang, Y., Yang, F., Zhu, H., Zhou, D., and Zeng, X. Nonlinear cnn: improving cnns with quadratic convolutions. *Neural Computing and Applications*, pp. 1–10, 2019.

Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., and Berg, A. C. Ssd: Single shot multibox detector. In *European conference on computer vision*, pp. 21–37. Springer, 2016.

Loshchilov, I. and Hutter, F. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

Lucas, T., Shmelkov, K., Alahari, K., Schmid, C., and Verbeek, J. Adversarial training of partially invertible variational autoencoders. In *INNF'19-Workshop on Invertible Neural Nets and Normalizing Flows*, pp. 1–14, 2019.

Mantini, P. and Shah, S. K. Cqnn: Convolutional quadratic neural networks. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pp. 9819–9826. IEEE, 2021.

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

Milenkovic, S., Obradovic, Z., and Litovski, V. Annealing based dynamic learning in second-order neural networks. In *Proceedings of International Conference on Neural Networks (ICNN'96)*, volume 1, pp. 458–463. IEEE, 1996.

Mishra, P., Lehmkuhl, R., Srinivasan, A., Zheng, W., and Popa, R. A. Delphi: A cryptographic inference service for neural networks. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pp. 2505–2522, 2020.

Miyato, T., Kataoka, T., Koyama, M., and Yoshida, Y. Spectral normalization for generative adversarial networks. In *International Conference on Learning Representations*, 2018.

Orhan, A. E. and Pitkow, X. Skip connections eliminate singularities. *arXiv preprint arXiv:1701.09175*, 2017.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32:8026–8037, 2019.

Radosavovic, I., Johnson, J., Xie, S., Lo, W.-Y., and Dollár, P. On network design spaces for visual recognition. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 1882–1890, 2019.

Redlapalli, S., Gupta, M. M., and Song, K.-Y. Development of quadratic neural unit with applications to pattern classification. In *Fourth International Symposium on Uncertainty Modeling and Analysis, 2003. ISUMA 2003.*, pp. 141–146. IEEE, 2003.

Ren, S., He, K., Girshick, R., and Sun, J. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28:91–99, 2015.

Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., and Chen, X. Improved techniques for training gans. *Advances in neural information processing systems*, 29:2234–2242, 2016.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Tokui, S., Oono, K., Hido, S., and Clayton, J. Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, volume 5, pp. 1–6, 2015.

Xu, Z., Yu, F., Liu, C., and Chen, X. Reform: Static and dynamic resource-aware dnn reconfiguration framework for mobile device. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pp. 1–6, 2019.

Zoumpourlis, G., Doumanoglou, A., Vretos, N., and Daras, P. Non-linear convolution filters for cnn-based learning. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 4761–4769, 2017.