Lightweight and Accurate DNN-Based Anomaly Detection at Edge

Zhang, Qinglong ; Han, Rui; Xin, Gaofeng; Liu, Chi Harold; Wang, Guoren; Chen, Lydia Y.

DOI
10.1109/TPDS.2021.3137631

Publication date
2022

Document Version
Final published version

Published in
IEEE Transactions on Parallel and Distributed Systems

Lightweight and Accurate DNN-Based Anomaly Detection at Edge

Qinglong Zhang, Rui Han, Gaofeng Xin, Chi Harold Liu, *Senior Member, IEEE*, Guoren Wang, *Senior Member, IEEE*, and Lydia Y. Chen, *Senior Member, IEEE*

Abstract—Deep neural networks (DNNs) have been showing significant success in various anomaly detection applications such as smart surveillance and industrial quality control. It is increasingly important to detect anomalies directly on edge devices, because of high responsiveness requirements and tight latency constraints. The accuracy of DNN-based solutions rely on large model capacity and thus long training and inference time, making them inapplicable on resource strenuous edge devices. It is hence imperative to scale DNN model sizes in correspondence to the run-time system requirements, i.e., meeting deadlines with minimal accuracy losses, which are highly dependent on the platforms and real-time system status. Existing scaling techniques either take long training time to pre-generate scaling options or disturb the unsteady training process of anomaly detection DNNs, lacking the adaptability to heterogeneous edge systems and incurring low inference accuracies. In this article, we present LightDNN to scale DNN models for anomaly detection applications at edge, featuring high detection accuracies with lightweight training and inference time. To this end, LightDNN quickly extracts and compresses blocks in a DNN, and provides large scaling space (e.g., 1 million options) by dynamically combining these compressed blocks online. At run-time, LightDNN predicts the DNN's inference latency according to the monitored system status, and optimizes the combination of blocks to maximize its accuracy under deadline constraints. We implement and extensively evaluate LightDNN on both CPU and GPU edge platforms using 8 popular anomaly detection workloads. Comparative experiments with state-of-the-art methods show that our approach provides 145.8 to 0.56 trillion times more scaling options without increasing training and inference overheads, thus achieving as much as 15.05% increase in accuracy under the same deadlines.

Index Terms—Anomaly detection, edge inference, DNN, model scaling, predictable latency

1 Introduction

The fast development of Internet of Things (IoT) and deep learning technology leads to the emergence of anomaly detection applications at the edge of network [1]. Deep neural networks (DNNs) have become ubiquitous in these applications [2], [3], as accurate detection requires effectively extracting features from images [4], [5], [6], [7], [8] or videos [6], [9], [10], [11]. Existing DNN-based anomaly detection techniques focus on improving detection accuracies [7], [8] or dealing with complex images or videos [6], [7], [8]), both of which require DNNs of increasing complexities. Performing anomaly detection at edge thus need to meet time requirements, which are natural to many anomaly detection systems for identifying outliers in real-time, while enjoying the high accuracy brought by complex DNNs. This requires precisely scaling the model sizes in correspondence to the dynamics of available resources, so as to meet the inference deadline with *small overheads and minimal accuracy losses*. However, applying existing scaling techniques, that is, layer removing [12], [13], [14], [15], [16], nested networks [17], [18], [19], [20], FLOP scaling [21], [22], [23], [24], [25], and Neural architecture search (NAS) [26], [27], [28], to anomaly detection systems, faces two key challenges in practice.

First of all, existing techniques employ *the model-grained scaling mechanism, which needs extremely long time to provide sufficient scaling options*. Specifically, layer removing, FLOP scaling, and NAS techniques generate multiple models of different sizes. When handling complex DNNs, these techniques take hours to generate one model even using powerful GPU servers. Nested network techniques generate a multi-capacity model (e.g., switchable model [19], [20]) to provide large scaling space. However, these techniques either consume large memories (e.g., 1k scaling options takes GBs of memory in US-Net [29]) or use time-consuming architecture search algorithm (e.g., searching 1k models takes more than 10 days in FN^3 [20]). Hence existing techniques can only support a small number of descendant models, which cause two major problems in scaling:

Low Accuracies. Descendant models provide a coarse-grained differentiation of resources, thus leaving considerably proportions of resource unused. We tested two anomaly detection workloads: GANomaly [7] on the Coil100 image dataset and ResNet-18 [10] on the UCSD-peds1 video dataset. Taking the filtering pruning technique as an example, Fig. 1 shows that to meet the deadline constraint, the

- *Qinglong Zhang, Rui Han, Gaofeng Xin, Chi Harold Liu, and Guoren Wang are with the Beijing Institute of Technology, Beijing 100811, P.R. China. E-mail: {3120211050, hanrui, wanggr}@bit.edu.cn, 2361417120@qq.com, liuchi02@gmail.com.*
- *Lydia Y. Chen is with TU Delft, 2628 CD Delft, The Netherlands. E-mail: lydiaychen@ieee.org.*

Manuscript received 2 Sept. 2021; revised 15 Nov. 2021; accepted 14 Dec. 2021. Date of publication 29 Dec. 2021; date of current version 23 May 2022.
This work was supported in part by the National Natural Science Foundation of China under Grants 61872337 and 62132019, in part by the National Research and Development Program of China under Grant 2019YQ1700, and in part by Swiss National Science Foundation NRP75 under Grant 407540_167266.
(Corresponding author: Rui Han.)
Recommended for acceptance by A. J. Peña, M. Si, and J. Zhai.
Digital Object Identifier no. 10.1109/TPDS.2021.3137631

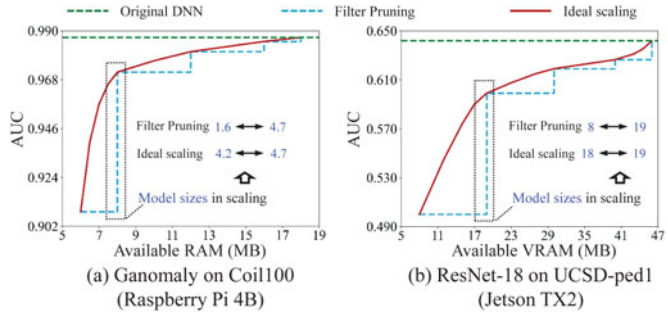


Fig. 1. Comparison of accuracy losses between model-grained scaling and ideal scaling.

used DNN incurs considerably *lower accuracies (AUC)* compared to that of the ideal scaling case. This is because in the ideal case, the DNN can be scaled at a much finer granularity, thus guaranteeing model size precisely fits the available memory.

High Scaling Overheads. In an online scaling, most of existing techniques need to exchange an entire DNN and thus result in high paged in and page out overheads. In particular, some nested network techniques (e.g., FN^3 [20] and OFA [19]) fail to support part-exchanged model scaling due to their irregular weight sharing mechanism. Some other nested network techniques (e.g., NestDNN [17] and US-Net [18]) can exchange part of the DNN in scaling, but they still cause high overheads because they need extra expensive operations to incorporate the newly added part to the current DNN.

Moreover, existing techniques depend on *the unsteady training process of anomaly detection and thus further worsen accuracy in model scaling*. Specifically, many DNN-driven anomaly detection applications employ *generative adversarial network (GAN)-based model training* [30] or *self-training* [10] to enable unsupervised anomaly detection. Both methods are difficult to train because their training process is unstable and easy to be unbalanced, e.g., one of models has already converged but others haven't. However, when generating descendant models, existing scaling techniques such as nested network and FLOP scaling need to introduce extra operations to such unstable training processes, which may severely degrade model accuracy.

In this paper, we present LightDNN, a lightweight and accurate scaling framework that is designed fundamentally based on the block structure (e.g., ResBlock in ResNet [31]) widely exist in DNNs of today's anomaly detection systems. The key idea of LightDNN is to identify a finite number of blocks in a DNN, generate each block's descendant/compressed blocks with short training time, and combine these blocks to provide large scaling space. Moreover, the block generation process is independent of the GAN-based training and self-training process, hence it works well for a wide range of anomaly detection applications. The concrete contributions of this work are as follows:

Block-Grained Scaling Mechanism for Large Scaling Space and Small Overheads. The limitation of *model-grained* scaling is rooted in the constraint that the number of scaling options equals to the number of descendant models and hence they are not scalable. LightDNN proposes a novel block training approach that extracts blocks from a DNN and train their descendant blocks independent of the whole network. This allows the training to be completed fast and by just generating dozens of descendant blocks for a DNN, the combination of these blocks brings massive choices of descendant models. At run-time, LightDNN employs an optimizer to select the combination of (descendant) blocks that minimize accuracy losses. The scaling decision is then conducted by only changing a small proportion of descendent blocks with small overheads.

Independence of the Unsteady Training Process in Anomaly Detection. Rather than modifying the training process, LightDNN starts from a trained DNN model and identifies its blocks. By taking each block as the teacher model, LightDNN employs model compression techniques (e.g., filter pruning [32]) to transform the block into descendant blocks of different sizes and trains these blocks using supervised and steady SGD-based method. This allows LightDNN work well for DNNs in anomaly detection applications.

Implementation and Evaluation. We implement a system prototype of LightDNN and conduct extensive evaluation against the state-of-the-art techniques, i.e., layer removing, nested network, and FLOP scaling. Using 8 popular image and video anomaly detection workloads, we summarize of evaluation results as follows: *(i) Extensible in terms of architecture*. The evaluation is performed on both CPU and GPU edge platforms: one Raspberry Pi 4B with ARM architecture designed for low overhead edge systems, and one Jetson TX2 with 256-core NVIDIA Pascal^{TM} GPU architecture. *(ii) Low-overhead offline training and online scaling*. LightDNN provides 145.8 to 0.56 trillion times more scaling options using shorter model generation time, while reducing online scaling overheads by 76.06%. *(iii) Precise latency prediction and optimized online scaling*. Under different workloads, available memories and performance interferences, LightDNN achieves an average prediction error of 5.14%, much lower than that (35.49%) of existing approaches. Our prediction-based scaling optimizer is able to improve accuracy by 12.26% on average compared to layer removing and nested network techniques under the same deadlines, by 17.84% compared to FLOP scaling techniques under the same computational costs, and by 15.05% overall.

2 Background and Related Work

2.1 DNN-Based Anomaly Detection

Anomaly detection, or outlier detection, finds objects with unexpected behaviors. With the prosperity of IoT technology, anomaly detection systems are widely deployed on edge devices. High-dimensional data such as image and video widely exists in today's anomaly detection applications. For such an issue, deep learning based anomaly detection techniques are increasingly used and this work focuses on two prevalent DNN-driven anomaly detection applications.

Image Anomaly Detection. Two reconstruction-based DNNs are used to handle image data: autoEncoder [4] and GAN, which compute abnormal scores according to the errors between reconstructed images and original images. Specifically, *reconstruction-based methods* initially use autoEncoder to improve accuracy from two aspects: improving the network structure [33] and using latent vector of autoEncoder [34]. GPND is then used to force the latent vectors to

be close to normal distribution [5]. GANomaly [7] is a representative technique that combines GAN and autoEncoder, and uses autoEncoder as GAN's discriminator in model training.

Video Anomaly Detection. This application uses both reconstruction-based and classification-based DNNs. Classification-based algorithms train anomaly detection classifiers based on self-supervised learning [10], [11]. For example, self-supervised algorithms (or non-supervised learning algorithms) first use pre-trained Yolov3 to get pseudo-labels/objects, and then apply convolutional neural networks (e.g., ResNet-18 or ResNet-50 [10]) to extract features, and finally train classifiers based on fully connected networks or convolutional neural networks (CNNs). These methods are widely used to detect anomaly video, because they do not need to label every frame of videos manually.

Most image and video anomaly detection applications use *GAN-based training and self-training* methods to construct their DNNs. Specifically, the GAN-based method uses two or more models to learn the data distribution of normal samples through the adversarial learning between them. GAN is difficult to train because the process of adversarial learning is sensitive and easy to be unbalanced, e.g., one of them has already converged but others haven't [30]. Similarly, in self-training [10], the model learns from the pseudo-labels updated by itself and updates the label through the prediction result of itself. The network predictions are sometimes incorrect, which imports incorrect pseudo-labels. Overfitting these labels will cause "confirmation bias" [35] and affect the accuracy of the model.

While aforementioned studies with promising results in detection anomaly, their models need to be supported by computation capacity, being CPU or GPU, with which the changing available resources on edge devices may not be equipped. *It thus calls for a solution to efficiently scale up or down existing anomaly detection models to achieve the full promise of accurate DNNs.*

2.2 Related Work

When running complex DNN models in the cloud [36] or on the edge [37], [38], three categories of scaling techniques can be applied to trade off inference latency and accuracy.

Layer removing techniques apply structured pruning techniques to trade off inference latency and model accuracy in anomaly detection systems. Specifically, they first generate a list of descendant models of different sizes and then dynamically select one of them according to the available resources and the deadline constraint at run-time. Commonly used structured pruning techniques include filter pruning [32], and low rank decomposition [39], and knowledge distillation [40]. However, these techniques suffer from long re-training (fine-tuning) time in descendant model generation. For example, when training ResNet-18 on ImageNet [41], both filter pruning and low rank decomposition take an average of 7 and 8 hours to generate a descendant model on a GPU server with 48-GB Quadro RTX 8000 graphics card. This means scaling techniques can only have a small number of scaling options (e.g., 5 descendant blocks [17], [32]). The large gaps among these models may incur large accuracy losses.

Nested network techniques [17], [18], [19], [20] aim to design compact modules suitable for running on edge devices. They train all descendant models jointly by inserting some extra operations into each batch, e.g., sampling sub network or clearing parts of gradient [17]. Although these techniques can generate hundreds to trillions of descendant models for standard SGD-based DNNs (e.g., DNNs in image classification and object detection applications), they have poor effects in anomaly detection systems. This is because the extra operations in the training may result in the difficulty of convergence in GAN-based training [30] and the confirmation bias in self-training [35], thus degrading model accuracy. In particular, in US-Net [18], FN^3 [20], and OFA [19] proposes "progressive shrinking training" which causes large accuracy losses.

FLOP Scaling Techniques. Progressive inference techniques (MSDNet [21] and Hardware-Aware Progressive Inference (HAPI) [22]) use several points to support the early exist of inference if a pre-defined classification accuracy is met. Block-skipping techniques (BlockDrop [23] and ConvNet-AIG [24]) dynamically skip unimportant convolutional layers or blocks to reduce computational costs. However, both types of techniques employ unique and complex training processes and loss functions, and they are inapplicable to DNNs in anomaly detection applications (the training cannot converge). Channel gating (CGNet [25]) is the only technique that can be used in anomaly detection applications and it dynamically finds a DNN's unimportant parts and skips them in inference.

3 LightDNN

To enable the predictable latency and high accuracy in DNN scaling, two basic research questions need to be answered in LightDNN: (1) how to efficiently train and profile the descendant blocks for a DNN (Section 3.2); (2) how to perform fine-grained scaling of the DNN using these blocks with both high accuracy and small overheads (Section 3.2).

3.1 Overview

Fig. 2 shows the LightDNN architecture, which is split into an offline stage and an online stage.

The *offline* stage consists of two steps: block generation and profiling. Given a trained DNN with n blocks, the *block generation* step first transforms each block into a list of descendant blocks. Traditional model compression techniques re-train a compressed model from scratch and may suffer from long training time. In contrast, LightDNN re-trains a descendant block based on the original block for two reasons. First, the training only uses the block's intermediate data and avoids the unnecessary computations on other parts of the network. Second, the block and all its descendant blocks have the same input and output channels in the DNN. This allows LightDNN to incorporate any of these descendant blocks into the network without extra operations.

Subsequently, the *block profiling* step generates profiles for all blocks and descendant blocks, including their inference latencies, accuracies and memory footprints. Note that in latency profiling, LightDNN only measures the percentage of reduced latency compared to that of the original/uncompressed block. This is because the latency of

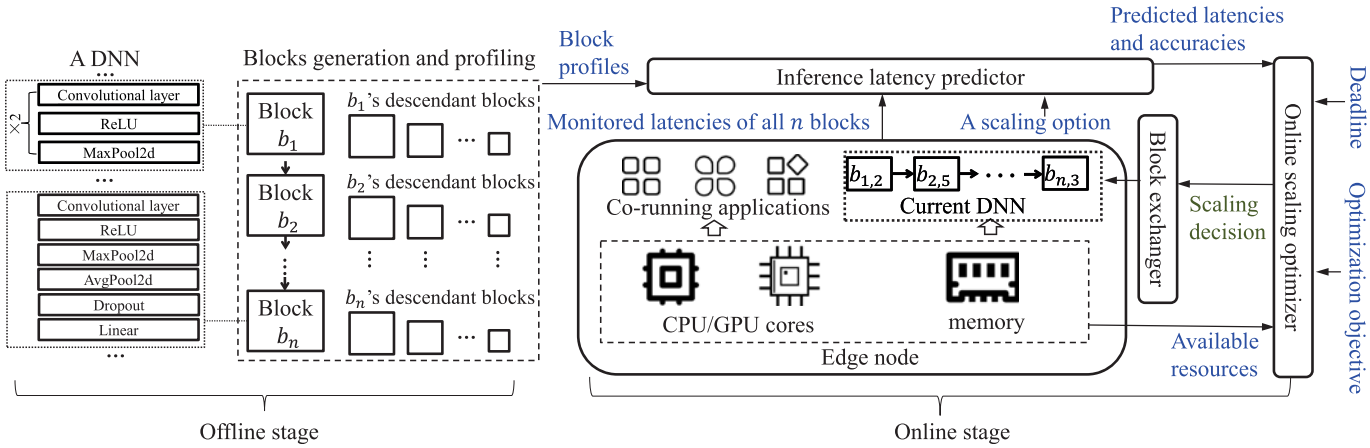


Fig. 2. LightDNN architecture.

processing the same block also depends on the system status, e.g., this latency increases when the system resource is saturated. The profile hence records the reduction percentage in order to be applicable to different performance interference scenarios.

At the *online* stage, three components work together to provide fine-grained and lightweight scaling. Once any change in available resources or performance interference is detected for the DNN, the *online scaling optimizer* finds the optimal scaling option that maximizes the inference accuracy under a given deadline and outputs it as the scaling decision. In optimization, the *inference latency predictor* takes a scaling option as input and outputs the DNN's predicted latency according the latest system status (that is, the monitored latencies of all n blocks). The *block exchanger* conducts this scaling decision that corresponds to a combination of the (descendant) blocks in the DNN and only switches replaced blocks to reduce page in/page out overheads.

3.2 System Model

This work considers a DNN that can be decomposed into a sequence of n blocks: $\{b_1, \ldots, b_n\}$. Each block is one part of the network and consists of multiple layers such as convolutional layers, nonlinearity such as a ReLU, and pooling layers (e.g., max pooling layer) [42]. In the DNN, these blocks take most of computational costs and memory footprints, and hence they determine the inference latency of an anomaly detection system. For example, Fig. 2 illustrates two blocks (b_1 and b_n) in a DNN and both blocks contain convolutional layers and other layers. We note that most of the DNNs applied in anomaly detection systems (examples are GANomaly [7], OGNet [8], and ResNet [31]) have similar architectures of convolutional layers in their blocks.

Descendant Block. In LightDNN, each block b_i corresponds to a set of n_i descendant blocks of smaller sizes ($1 \leq i \leq n$ and $n_i > 1$). A descendant block $b_{i,j}$ ($0 \leq j \leq n_i$ and $b_{i,0}$ is the original block b_i) is characterized by:

$$b_{i,j} := (\mathcal{S}_{i,j}, \mathcal{T}_{i,j}, \mathcal{A}_{i,j}, \mathcal{B}_{i,j}^{(k)}). \tag{1}$$

- $\mathcal{S}_{i,j}$: the decreased memory footprint of $b_{i,j}$ ($\mathcal{S}_{i,0} = 0$).
- $\mathcal{T}_{i,j}$: the percentage of processing latency reduction in $b_{i,j}$ ($\mathcal{T}_{i,0} = 0$).
- $\mathcal{A}_{i,j}$: the accuracy loss of $b_{i,j}$ ($\mathcal{A}_{i,0} = 0$).
- $\mathcal{B}_{i,j}^{(k)}$: whether descent block $b_{i,j}$ is switched into the model in kth block-grained scaling ($k \geq 0$ and $\mathcal{B}_{i,j}^{(k)} \in \{0, 1\}$). $\mathcal{B}_{i,j}^{(0)}$ represents the initial state before scaling.

Layered Structure in DNN Scaling. In block-grained scaling, LightDNN guarantees that the DNN's layer structure keeps unchanged. First, a descendant block's decision variable $\mathcal{B}_{i,j}^{(k)}$ is either 0 (not selected) or 1 (selected). Second, for each block b_i, only one of its n_i descendant blocks can be selected

$$\forall\ 1 \leq i \leq n, \sum_{j=0}^{n_i} \mathcal{B}_{i,j}^{(k)} = 1. \tag{2}$$

3.3 Training Process Independent Block Generation

Block Generation. The *block generation* step first extracts blocks from the original trained DNN and applies pruning filter techniques to transform each block into a list of descendant blocks. These blocks are then re-trained to recover their functionalities based on the their *training environment*. As illustrated in Fig. 3, a block's training environment corresponds to its input and output channels in the DNN. Using this environment, the standard stochastic gradient descent (SGD) method is applied to quickly train a descendant block. At each training iteration, a mini-batch $\mathcal{D}$ is sampled from the training dataset. For a descendant block

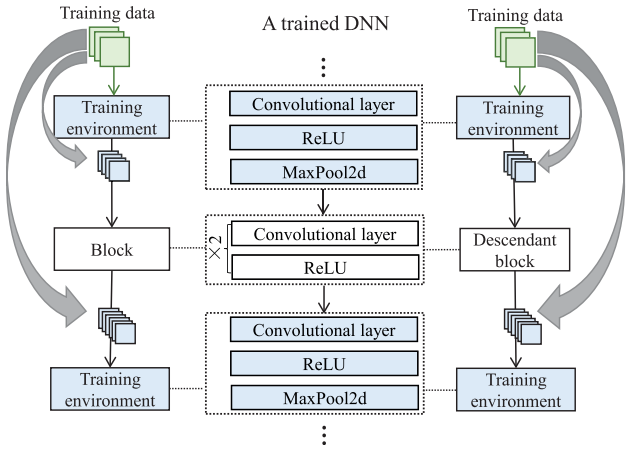


Fig. 3. Training environment in block generation.

$b_{i,j}$, the loss $\mathcal{L}$ is defined as

$$\begin{aligned}
&\mathcal{F}_{input} = (b_{i-1} \circ \cdots \circ b_1 \circ b_0)(\mathcal{D}) \\
&\mathcal{F}_{output} = b_i(\mathcal{F}_{input}) \\
&\mathcal{F}^*_{output} = b_{i,j}(\mathcal{F}_{input}) \\
&\mathcal{L} = \|\mathcal{F}_{output} - \mathcal{F}^*_{output}\|_2^2,
\end{aligned} \tag{3}$$

where $\circ$ represents the composite function, i.e., $(f \circ g)(x) = f(g(x))$. The calculation of loss has three steps. Step 1 calculates the intermediate data (i.e., block b_i's input $\mathcal{F}_{input}$) in the original model. In practice, we can collect this from a forward process in inference. Given input $\mathcal{F}_{input}$, step 2 calculates the outputs $\mathcal{F}_{output}$ and $\mathcal{F}^*_{output}$ of b_i and $b_{i,j}$, and computes the difference of these two outputs as loss $\mathcal{L}$. Finally, step 3 back propagates the loss $\mathcal{L}$ to update $b_{i,j}$'s parameters.

Memory Footprint and Accuracy Profiling. The *block profiling* step can directly calculate a block b_i or its descendant block $b_{i,j}$'s memory footprint according to its model size, and estimate $b_{i,j}$'s decreased memory footprint $\mathcal{S}_{i,j}$. In contrast, obtaining $b_{i,j}$'s accuracy loss is not trivial because the block incurs different accuracy losses when combined with other blocks of different sizes. The profiling step hence considers v representative model sparsities/sizes in profiling a descendant block $b_{i,j}$ and calculates its accuracy loss as the average accuracy loss of the k profiles.

Latency Profiling. On edge devices, the processing latency of a block is considerably influenced by its co-running applications due to their resource contention and performance interference. Random system background activities such as system maintenance or garbage collection also influence the processing latency. This means the absolute value of a descendant block's latency dynamically changes under different performance interferences. Typically, larger performance interferences incurs longer latencies and hence this descendant block brings larger reductions in latency in profiling. However, at run-time, the performance interference dynamically changes and it is infeasible to estimate a descendant block's latency reduction at the offline stage. For such an issue, the profiling step predicts the *percentage of latency reduction* for each descendant block. This prediction approach is inspired by one key intuition: *a block's processing latency is proportional to its block size, namely its number of parameters*. In other words, the ratio of two blocks latencies can be calculated offline according to their block sizes. Let s_i and $s_{i,j}$ be the block sizes of block b_i and its descendant block $b_{i,j}$, the percentage of latency reduction $b_{i,j}$ is calculated as $\mathcal{T}_{i,j} = \frac{s_{i,j}}{s_i}$. Note that the above calculation of latency reduction works for both a block's ideal latency (i.e., its minimum latency without performance interference) and latencies under different interferences.

3.4 Block-Grained DNN Scaling

In this section, we first show how to pack all requirements and constraints of an online optimization into a strict mathematical optimization problem and solve it efficiently on resource-constrained devices (Section 3.4.1). We then explain how to obtain the accurate online profiles required in this solving process (Section 3.4.2) and how to perform the block exchange in online scaling (Section 3.4.3).

3.4.1 Online Scaling Optimizer

At run-time, the optimizer finds the combination of (descendant) blocks for a DNN to minimize its accuracy loss. Equation (4) defines the optimization objective o_k in the kth scaling ($k \geq 1$) as the accuracy loss caused by the descendant blocks

$$\begin{aligned}
&o_k = \min_{\mathcal{B}} \sum_{i=1}^{n} \sum_{j=0}^{n_i} \mathcal{A}_{i,j} \cdot \mathcal{B}^{(k)}_{i,j} \\
&s.t. \\
&t^{(k)}_{predict} \leq t_{deadline} \\
&s_{DNN} - \sum_{i=1}^{n} \sum_{j=0}^{n_i} \mathcal{S}_{i,j} \cdot \mathcal{B}^{(k)}_{i,j} \leq s_{max} \\
&\forall\ 1 \leq i \leq n, \sum_{j=0}^{n_i} \mathcal{B}^{(k)}_{i,j} = 1, \quad \mathcal{B}^{(k)}_{i,j} \in \{0, 1\}.
\end{aligned} \tag{4}$$

The optimization has three constraints: (1) *deadline constraint*. given a scaling option, the predicted latency of the DNN (Equation (5)) should be lower than the deadline $t_{deadline}$. (2) *Memory constraint*. Let s_{DNN} be the memory footprint before scaling and $\sum_{i=1}^{n} \sum_{j=0}^{n_i} \mathcal{S}_{i,j} \cdot \mathcal{B}^{(k)}_{i,j}$ be the decreased memory footprint in scaling down or the minus of increased memory footprint in scaling up. The second constraint states that the memory footprint after scaling should be smaller than the maximal available memory s_{max}. (3) *Structure constraint*. The DNN's layer structure keeps unchanged in scaling. That is, a descendant block $b_{i,j}$ can only be replaced by one of the descendant blocks generated from the same block b_i.

The optimization in Equation (4) can be seen as an Integer Linear Programming (ILP) problem because its decision variable $\mathcal{B}^{(k)}_{i,j}$ can only be 0 or 1. LightDNN solves this ILP problem with two phases. First, it relaxes the integer constraint of ILP and transforms it into a linear programming (LP) problem p^* that can be quickly solved. Second, it employs the branch and bound method to recursively search the optimal solution of ILP with polynomial time complexity. The basic idea of the branch and bound method is to construct a tree that uses p^*'s optimal solution q^* as the root node, and recursively traverses its branch nodes until the optimal solution of ILP is found. ILP problem is NP-complete and can't be solved in polynomial time. LightDNN also adds an early stopping threshold σ such that the search process is completed if the found optimal solution is close enough to the actual optimal solution. This guarantees the availability and efficiency of this algorithm on resource-constrained edge devices.

In the above optimization, a descendant block is either used or not in the DNN. This descendant block represents a portion of its original block and conceptually, massive descendant blocks can be generated to provide a fine-grained differentiation of the original block. However, our approach only generates a finite number (e.g., 5) of descendant blocks for each original block for two reasons. First, the accuracy and latency gaps of descendant blocks are small. For instance, when generating 5 descendant blocks for ResNet-18, the accuracy gap of two consecutive blocks is less than 0.006 and their latency gap is less than 0.06 seconds (on Raspberry Pi). This means 5 descendant blocks

provide a sufficiently fine granularity in accuracy-latency trade-offs. Second, a DNN consists of a few to dozens of blocks. Even if each block only has several descendant blocks, the combination of these blocks provide thousands to millions of scaling options.

As with existing techniques, we do not support online block generation because re-training is required to recover the functionality of the pruned block. Therefore, it's impractical to switch just a part of a block into the model (i.e., generating a smaller block based on this existing block on the fly and switching it into the model). We just switch it completely or not.

Actually, this simple strategy is sufficient to provide an accurate and seamless trade-off between inference accuracy and latency because of the following two reasons:

1) There're significant differences in performance between a few (e.g., 5) descendant blocks of an original block. For instance, in ResNet-18, $b_{8,1}$ provides 3.99% of model size reduction and 0.60% of accuracy loss, while $b_{8,2}$ provides 7.97% of model size reduction and 1.43% of accuracy loss. The accuracy and seamlessness of LightDNN will not increase linearly as the number of descendant blocks increases, that is, marginal effects will occur when generating excessive descendant blocks. Unnecessary descendant blocks just add more offline costs without adequate contribution to model performance.
2) For a single block, we just choose it or not. However, this block-level binary choice can generate model-level seamless scaling by compositions of several blocks, so LightDNN can provide accurate model scaling according to the latency requirements and memory constraints.

3.4.2 Inference Latency Predictor

In Equation (4), the inference latency prediction is based on the generated descendant blocks and their profiles and it faces two challenges at run-time. First, the co-running applications and background activities continuously change on the edge node, and hence the performance interference they cause dynamically changes. Second, the combination of different descendant blocks creates a large space of candidate scaling options, and it is infeasible to test each option online. LightDNN solves these challenges by dividing the prediction into *two steps*. Step 1 predicts the latencies of the *uncompressed* DNN and its *original* blocks by estimating the performance interference according to the monitored latencies in the current DNN. Step 2 takes a scaling option (that is, a combination of descendant blocks) as input and outputs the predicted latency if this scaling is conducted.

Specifically, *step 1* predicts the latencies of the uncompressed DNN and blocks by taking five inputs: (1) t_i^m is the monitored latency of block b_i in the current DNN. The latencies of all n blocks represent the latest performance interference in the system; (2)$\mathcal{B}_{i,j}^{(k-1)}$ denotes whether block $b_{i,j}$ is used ($\mathcal{B}_{i,j}^{(k-1)}$=1) or not ($\mathcal{B}_{i,j}^{(k-1)}$=0) by the current DNN; (3) t^* is the DNN's *ideal* latency without performance interference; (4) t_i^* is block b_i's *ideal* latency without performance interference; (5) $\mathcal{T}_{i,j}$. The first two inputs are decided by run-time system and DNN scaling status and the last three inputs are from block profiles. Step 1 outputs the uncompressed DNN's *predicted* latency ($t^{(k)}$) and each block b_i's *predicted* latency ($t_i^{(k)}$) in the kth scaling.

Algorithm 1 details the process of this step. It employs an interference factor α_i to reflect the performance interference to each block b_i (line 1). This factor is calculated as the predicted latency $t_i^{(k)}$ of block b_i (lines 2 to 7) divided by b_i's ideal latency t_i^* without performance interference (line 8). Subsequently, the algorithm identifies and removes the outliers of these interference factors (lines 10 to 18). Let μ and σ be the mean and standard deviation of the n factors, an factor α_i is an outlier if $|\alpha_i - \mu| \geq 3\sigma$. Finally, the algorithm uses the remaining factors to calculate the interference factor $\hat{\alpha}$ of the DNN (line 19) and predicted latency $t^{(k)}$ for the whole network (line 20).

Algorithm 1. Block Latency Prediction for the kth Scaling

Require: t_i^m, $\mathcal{B}_{i,j}^{(k-1)}$, t^*, t_i^*, $\mathcal{T}_{i,j}$ $(1 \leq i \leq n, 1 \leq j \leq n_i)$
1: Set $\alpha_i = 0$ for $1 \leq i \leq n$;
2: **for** (i = 1; $i \leq n$; i++) **do**
3: **for** (j = 0; $j \leq n_i$; j++) **do**
4: **if** $\mathcal{B}_{i,j}^{(k-1)}$ **then**
5: $t_i^{(k)} = t_i^m$ / (1 - $\mathcal{T}_{i,j}$), break;
6: **end if**
7: **end for**
8: $\alpha_i = \frac{t_i^{(k)}}{t_i^*}$;
9: **end for**
10: $\mu = \frac{\sum_{v=1}^{n} \alpha_v}{n}$;
11: $\sigma = \sqrt{\frac{\sum_{v=1}^{n} (\alpha_k - \mu)^2}{n}}$
12: $\hat{\alpha} = 0$, $v = 0$;
13: **for** (i = 1; $i \leq n$; i++) **do**
14: **if** $|\alpha_i - \mu| < 3\sigma$ **then**
15: $\hat{\alpha} += \alpha_i$;
16: $v++$;
17: **end if**
18: **end for**
19: $\hat{\alpha} /= v$;
20: $t^{(k)} = t^* \cdot \hat{\alpha}$;
21: return $t^{(k)}$, $t_1^{(k)}$ to $t_n^{(k)}$.

Given a candidate scaling option represented by $\mathcal{B}_{i,j}^{(k)}$ ($k \geq 1$, $1 \leq i \leq n$, and $0 \leq j \leq n_i$), *step 2* predicts the latency of the DNN as the latency $t^{(k)}$ of the uncompressed DNN (namely the longest latency) minus the reduced latencies because of the compressed/descendant blocks selected in the scaling

$$t_{predict}^{(k)} = t^{(k)} - \sum_{i=1}^{n} \sum_{j=0}^{n_i} t_i^{(k)} \cdot \mathcal{T}_{i,j} \cdot \mathcal{B}_{i,j}^{(k)}. \quad (5)$$

3.4.3 Block Exchanger

For a DNN model of n blocks, let $\mathcal{B}_{i,j}^{(k-1)}$ and $\mathcal{B}_{i,j}^{(k-1)}$ $(1 \leq i \leq n$ and $1 \leq j \leq n_i)$ be its choice of blocks before and after the kth scaling. The block exchanger is designed to conduct the optimal scaling decision with two steps. First, the exchanger scans $\mathcal{B}_{i,j}^{(k-1)}$ and $\mathcal{B}_{i,j}^{(k)}$ simultaneously. If $\mathcal{B}_{i,j}^{(k-1)} = 0$ and $\mathcal{B}_{i,j}^{(k)} = 1$, the block $b_{i,j}$ will be loaded from the disk into the memory immediately. Second, once the new block is

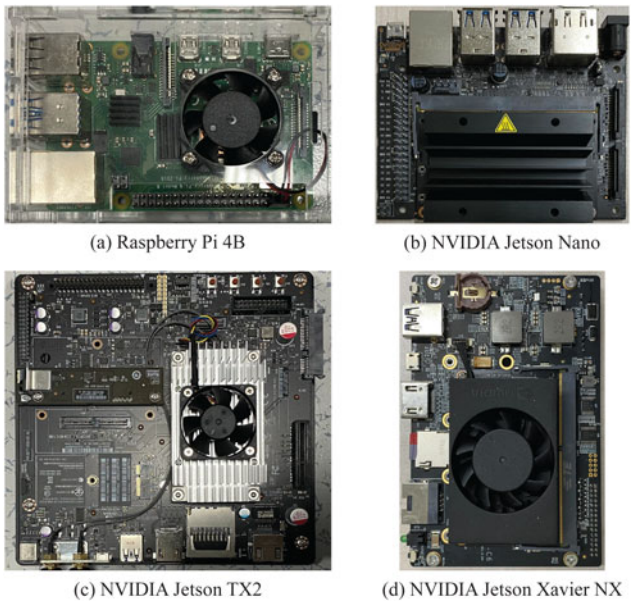


Fig. 4. Illustration of four edge platforms in evaluation.

loaded, it will be switched into the model to replace the corresponding old block. The memory space occupied by old replaced blocks will be paged out naturally by the garbage collection (GC).

Our block exchanger is lightweight for two reasons. First, traditional layer removing and FLOP scaling techniques need to exchange the entire DNN in a scaling. In contrast, our exchanger only switches a small number of blocks that donot exist in the scaling decision (i.e., where $\mathcal{B}_{i,j}^{(k-1)} \neq \mathcal{B}_{i,j}^{(k)}$), thus considerably reducing the page-in and page-out overheads.

Second, nested techniques such as NestDNN construct a multi-capacity model to incorporate all descendant models and thus also exchange a part of the DNN in a scaling. However, these techniques need extra operations: when some pages are loaded into memory, new layers needs to be constructed using both model weights (whose can be more than dozens of MBs in large networks) and APIs of deep learning libraries. These operations consume extra resources and time in scaling. In contrast, our exchanger needs no extra operations, because LightDNN maintains the same training environment for a block and all its descendant blocks, and hence allows the direct switching of them via simple assignment API provided by the programming language (e.g., $setattr()$ in Python).

4 Evaluation

In this section, we test the full implementation of LightDNN on top of PyTorch [43] with an extensive set of evaluations.

4.1 Experimental Setup

Testbeds. We choose both CPU and GPU edge platforms imposing different architectural features to showcase the cross-platform nature of LightDNN when it comes to hardware. We use Raspberry Pi 4B with 4 1.5GHz Cortex-A72 cores (ARM v8) and 4 GB memory; and NVIDIA Jetson TX2 with 256-core NVIDIA Pascal^{TM} GPU architecture and 8 GB memory. We also discuss the scalability of our approach using two Jetson platforms: Jetson Nano has 4 ARM Cortex-A57 cores and 4GB memory, and Jetson Xavier NX has 6 NVIDIA Carmel ARM v8.2 cores and 8GB memory. Fig. 4 illustrates the four edge platforms. All platforms run Linux Ubuntu 18.04 LTS.

DNN Models and Datasets, and Applications. To evaluate LightDNN's generalization capability on different anomaly detection tasks, we selected 8 workloads in two most important applications. Table 1 summarizes the dataset characteristics and the division of nomal/abnomal classes in these workloads.

- *Image anomaly detection*. This type of tasks usually employs reconstruction-based DNNs. Without loss of generality, we select three popular DNN models (GANomaly [7], GPND [5], and OGNet [8]) and four image datasets: Coil100 [44], EMNIST [45], Caltech-256 [46], and ImageNet [41].
- *Video anomaly detection*. We test classification-based DNNs for this type of tasks. We select four classification-based DNNs (ResNet-18, ResNet-50 [10], AlexNet and VGG16 [10], [11]) and one most commonly used video dataset (UCSD-ped1 [47]). UCSD-peds1 contains 34 training and 36 testing video samples.

Evaluation Metrics. Similar to most anomaly detection applications, the inference accuracy is measured by Receiver Operating Characteristic (ROC) Curve (AUC). This metric represents the probability that a randomly

TABLE 1
A Summary of Eight Anomaly Detection Workloads

Model	Dataset	#Data points	Abnormal classes
Ganomaly	Coil100	train: 5400 test normal: 1080 test abnormal: 720	Normal classes: 10th class abnormal classes: 50th-59th class
Ganomaly	EMNIST	train:345426 test normal:57527 test abnormal:411302	Normal classes: all letter abnormal classes: all digits
GPND	Caltech256	train:1798 test normal:455 test abnormal:455	Normal classes: 250th, 144th, 252th, 22th, 67th class abnormal classes: 256th class
OGNet	ImageNet2012	train:78055 test normal:25499 test abnormal:25499	Normal classes: 0th-79th class abnormal classes: 500th-519th class
ResNet-18 ResNet-50 AlexNet VGG16	UCSD-Peds1	train:14000 test normal:2800 test abnormal:1400	Abnormal classes: non pedestrian entitiesand anomalous pedestrian motion patterns

TABLE 2
Comparison of Offline Preparation Time Between LightDNN and Baseline Techniques

		Ganomaly	Ganomaly	*GPND*	OGNet	ResNet-18	ResNet-50	AlexNet	VGG16
Dataset		Coil-100	EMNIST	Caltech256	ImageNet2012	UCSD-Peds1			
#Blocks in a DNN		6	6	4	6	8	16	3	6
Filter Pruning	#scaling options	5	5	5	5	5	5	5	5
	time(h)	0.35	2.5	1.2	42	0.9	1.9	0.5	3.3
Low Rank Decomposition	#scaling options	5	5	5	5	5	5	1	1
	time(h)	0.3	2.1	0.96	40	0.66	1.5	0.1	0.6
Knowledge Distillation	#scaling options	5	5	5	5	5	5	5	5
	time(h)	0.25	1.4	0.9	33.6	0.7	1.8	0.45	2.7
CGNet	#scaling options	4	4	3	2	3	3	4	2
	time(h)	0.56	4.0	2.16	50.4	1.38	2.85	0.98	3.33
NestDNN	#scaling options	5	5	5	5	5	5	5	5
	time(h)	0.83	6.2	1.27	53	0.92	2.2	0.62	4
US-Net	#scaling options	256	256	512	512				
	time(h)	0.21	1.5	1.08	33.6				
FN^3	#scaling options	6.05e23	2.77e23	5.40e16	7.25e24				
	time(h)	0.35	2.5	1.8	63				
OFA	#scaling options	1953125	1953125	15625	9765625	10000	6.56e7	3125	1.2e9
	time(h)	18.25	123.1	802.5	199.4	51.5	89.7	58.55	200.7
LightDNN	#scaling options	**46656**	**46656**	**1296**	**46656**	**1679616**	**2.82e12**	**729**	**531441**
	time(h)	**0.08**	**4.2**	**3.24**	**4.5**	**0.42**	**2.4**	**0.33**	**2.665**

chosen positive test point (whose actual class is positive) will have a greater chance of being predicted as positive than a randomly chosen negative test point.

Compared Baselines. We implement and compare three categories of state-of-the art DNN scaling techniques from the literature. (1) *Layer removing* techniques including Filter pruning [32], low rank decomposition [39], and knowledge distillation [40]. (2) *Nested network* techniques including NestDNN [17], US-Net [18], FN^3 [20]. OFA [19]. (3) *FLOP scaling* techniques. Among five techniques (MSDNet [21], HAPI [22], BlockDrop [23], ConvNet-AIG [24], and CGNet [25]), only CGNet is applicable to GAN-based training and self-training in anomaly detection applications.

4.2 Evaluation of Offline and Online Overheads

4.2.1 Evaluation of Model and Block Generation

In the baseline approaches, the re-training of descendant models takes most of the generation part. Such re-training (or fine-tuning) is necessary to boost accuracy of descendant models. In LightDNN, the generation time is the summation of pre-training and profiling time of descendant blocks.

Comparison to Layer Removing and FLOP Scaling Techniques. For each DNN, we generate 5 descendant models in the layer removing techniques and 5 descendant blocks for each block in LightDNN. When applying to anomaly detection applications, the training is difficult to converge in the FLOP scaling technique (CGNet), so less numbers of descendant models can be generated. To make our comparisons fair, the re-training of each descendant model or block using the same training data, hyperparameter settings (the maximal epoch is 20), and platform (a GPU server with 48-GB Quadro RTX 8000 Graphics Card). Table 2 lists comparison results. We can see that LightDNN uses the shortest generation time in most of the cases because it only needs to train small blocks rather than the entire network. The only exception is the GPND model, because the inference in this model needs extra operations on latent representation such as SVD decomposition, thus causing much longer inference/profiling time. Moreover, the baseline approaches provide 1 to 5 optional models in scaling (low rank decomposition cannot compress AlexNet and VGG16 on the UCSD-Peds1 dataset). In contrast, LightDNN provides 729 to 2.82 trillion scaling options depending on the number of blocks in these DNNs.

Comparison to Nested Network Techniques. Following the above experimental settings, Table 2 shows that NestDNN takes long time to generate 5 nested descendant models. The other three techniques generate a large number of options using similar training time as LightDNN, but they have three restrictions in practical online scaling. First, US-Net and FN^3 are only applicable to half of anomaly detection applications. Second, switchable BN [29] is used in US-Net to support multiple nested descendant networks, and its memory footprint becomes extremely large when the number of descendant models is large. For example, the model size of US-Net on Ganomaly will double when the number of online descendant networks is 1k (e.g., 1k scaling options), and will reach GBs when it provides 50k scaling options. This means on resource-constrained edge devices, only small scaling options can be provided by US-Net. Finally, the number of scaling options in FN^3 is limited due to its time-consuming architecture search algorithm. For example, an accuracy test of GPND takes 3 minutes, which means it will take hundreds of days if we need search 10k GPND models in online scaling. For OFA, the search is based on an accuracy estimation model, whose training data needs long time to correct. For example, preparing 16k training data for GPND takes more than one month.

TABLE 3
The Average Accuracy Estimation Loss of LightDNN in Eight Workloads

Model	Dataset	Average accuracy estimation loss
Ganomaly	Coil100	0.46%
Ganomaly	EMNIST	0.18%
GPND	Caltech256	0.66%
OGNet	ImageNet2012	0.15%
ResNet18	UCSD-Peds1	0.51%
ResNet50		0.84%
AlexNet		0.24%

Evaluation of Estimation Model. Let a_{est} be a DNN's *estimated* accuracy, which is calculated using the accuracy of the original DNN and the profiled accuracy losses of descendant blocks. Let a_{true} be the DNN's *actual* accuracy, which is calculated using the test dataset. The precision of accuracy loss estimation is measured as the percentage of difference between these two accuracies: $\frac{|a_{true}-a_{est}|}{a_{true}}$.

For each DNN, we test multiple available memories within the range of its minimal and maximal model sizes, and the interval of two consecutive memories is 0.5 MB. We test an average of 100 memories for one DNN and report the average value: a smaller percentage means a higher precision of accuracy loss estimation. The evaluation results in Table 3 shows that for all DNNs, the percentage is smaller than 1% (the average percentage is 0.5%). That is, when LightDNN constructs DNNs of different sizes under different available memories, the accuracy loss estimation is precise to support scaling decisions.

4.2.2 Evaluation of Energy Consumption in Scaling

Another key overhead of scaling is the energy consumption cased by model switching when the available resource changes. Fig. 5 shows the comparison results of different approaches by testing 8 workloads on Raspberry Pi 4B and Jetson TX2. We used UNI-T UT658 power monitor to measure the energy consumptions. As expected, the energy consumed by *layer removing* and *FLOP scaling* techniques is the largest. This is because filter pruning/knowledge distillation, low rank decomposition, and CGNet directly replace the entire DNN to a smaller or larger one in a scaling, thus incurring the highest switching overheads. Take VGG16 an an example, the average page in and page out size is 261 MBs in filter pruning, 151 MBs and 117 MBs in OFA and NestDNN, and only 12 MBs in LightDNN.

Nested network techniques construct multi-capacity models to incorporate multiple descendant models and thus considerably reduces the page-in and page-out sizes in model switching. However, the multi-capacity models also incur extra operations in scaling: when some pages are loaded into memory, new layers needs to be constructed using extra operations. In contrast, *LightDNN* provides a lightweight scaling mechanism with twofold meanings. First, LightDNN only needs to exchange some blocks rather than the entire network, thus saving page-in and page-out overheads like nested network techniques. Second, the newly added blocks can be directly combined into the model without extra operations. This is because in scaling, LightDNN only exchanges a descendant block to another descendant block belonging to *the same original block*. In other words, LightDNN maintains the same input and output channels (i.e., training environment) for both descendant blocks in the network, thereby allowing the switching of them without extra operations. The results in Fig. 5 verify

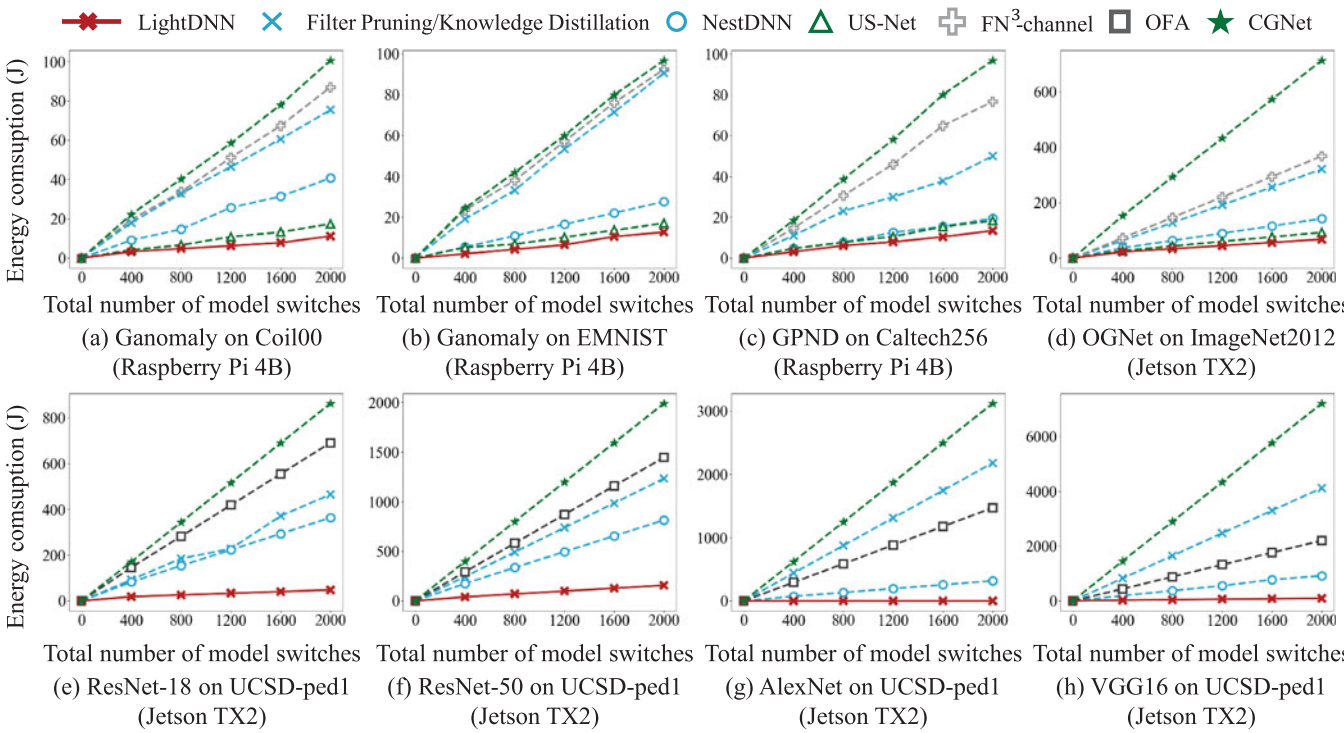


Fig. 5. Comparison of energy consumptions in model scaling.

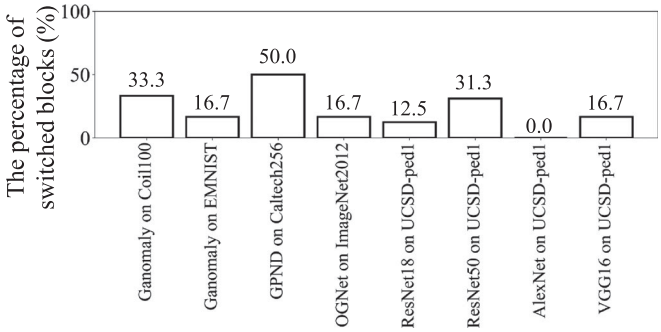


Fig. 6. The percentage of switched blocks in LightDNN when model scaling.

the above claims: LightDNN reduces energy consumptions by 68.16% and 83.03% on Raspberry Pi 4B and Jetson tx2, and by 71.94% overall.

Percentage of Switched Blocks. To support the above claim, we demonstrate the percentage of switched blocks in Fig. 6. The result shows that all models need to switch less than 50% of blocks to construct the new DNNs when the available memory changes. When considering all cases, LightDNN switches an average of 22.15% of blocks. Note that no block is switched in AlexNet, because the scaling optimizer finds that the smallest AlexNet model has the highest accuracy and tends to maintain this model under different available memories.

4.3 Evaluation of Inference Accuracy in Online Scaling

4.3.1 Evaluation of Inference Latency Prediction

The effectiveness of DNN scaling is considerably impacted by the accuracy of its latency predictor. To evaluate this accuracy, we ran each DNN model under variations of both memory resources and performance interferences. Specifically, 10 memories are tested for each workload. The lower and upper bounds of these memories are set as the minimum and maximum required resources to run the smallest and largest models in the baseline approaches. The DNN's co-running applications also continuously change to provide varying performance interferences. For each workload, the test was repeated 100 times for consistency and the distribution of prediction errors is reported using box plots.

Fig. 7 compares the prediction errors between LightDNN and seven baseline approaches (CGNet is not reported because this technique only scales FLOPs but does not change inference). This error is defined as the *percentage* of decreased or increased value in the predicted latency when comparing to the actual latency on the edge node. From Fig. 7, we can see that in most of the cases, the average prediction errors of our approach are smaller than 6%, indicating the latency predictor keeps a good track of the system status and reduced latency due to model scaling. In contrast, model-grained scaling approach can only selected one of the pre-generated models according the available memory and unaware the impact of performance interference, hence incurring large prediction errors between 20% to 50%. In all cases, LightDNN reduces prediction errors are much lower (6.90x lower) than those of all model-grained scaling approaches.

Discussion of Deadline Violation. In this evaluation, we test ResNet-50 (the DNN with the largest number of LightDNN blocks) on the GPU platform (Jetson TX2) and GPND on the CPU platform (Raspberry Pi). For each model and platform, we perform online scaling 100 times under various performance interferences and report the distribution of the 100 actual latencies in Fig. 8. The results show that the actual latencies of both models are approximately normally

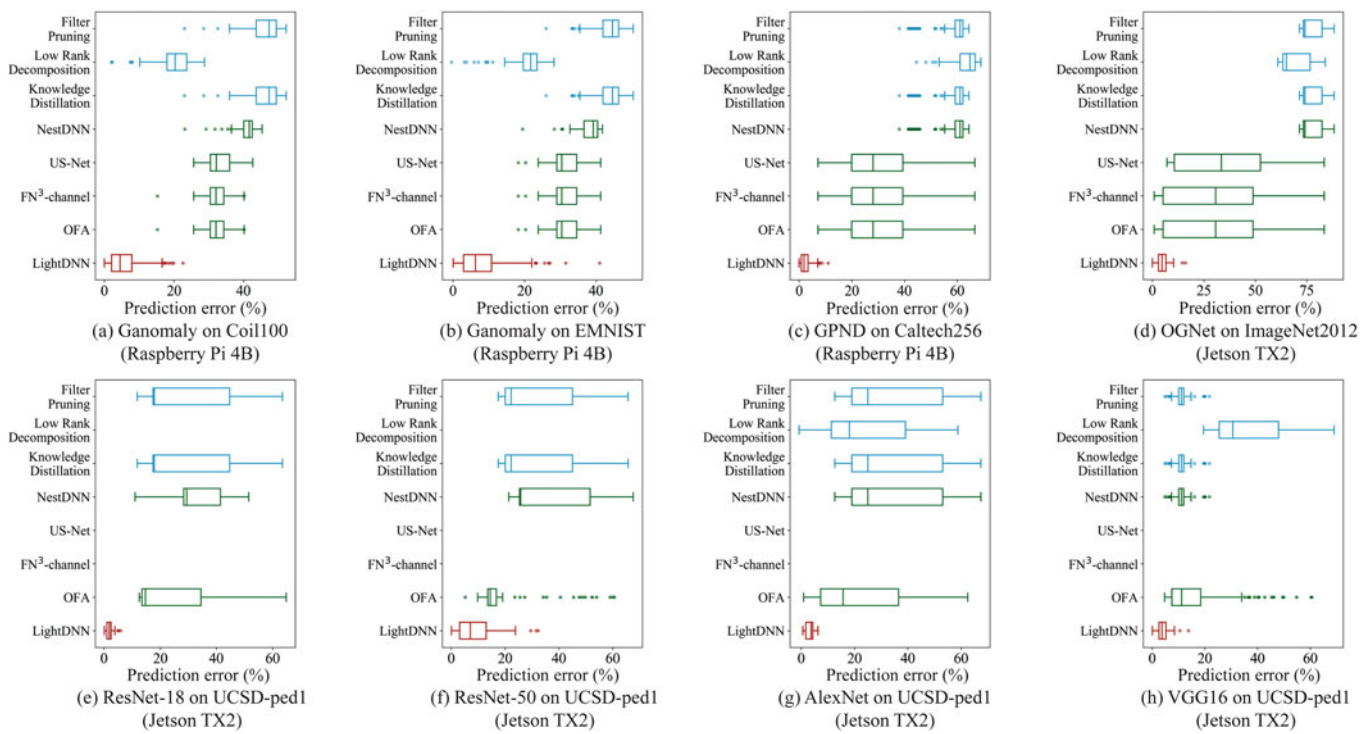


Fig. 7. Comparison of prediction errors of inference latency between LightDNN and baseline approaches.

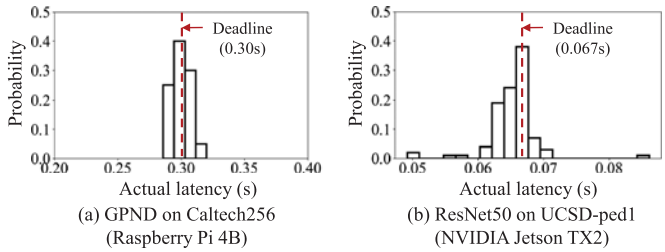


Fig. 8. The relationship between the deadline and the actual latencies in the CPU and GPU platform.

distributed around the given deadline, and most of the latencies are close to the deadline. This means the latency estimation in LightDNN is precise: the actual latencies are only slightly smaller or larger than the deadline. Specifically, 97.5% of the cases in GPND are distributed within 5% of the deadline, and 95% of the cases in ResNet-50 are distributed within 10% of the deadline. The estimation error on the GPU platform is higher because this platform has higher parallelism and thus higher uncertainties in inference. We note that anomaly detection systems usually apply a soft deadline constraint. That is, on a resource-sharing edge device, occasionally exceeding the deadline is allowed in anomaly detection. Our approach can also strictly meet the deadline constraint by setting a slightly shorter deadline in estimation.

4.3.2 Evaluation of DNN Scaling

Fig. 9 illustrates the comparison between LightDNN and *layer removing and nested network techniques* across 8 DNN workloads under deadline constraints, which are set to 25 ms for ResNet and AlexNet, 180 ms for VGG 16, 150 ms for Ganomaly and OGNet, and 300 ms for GPND (this model needs extra computations in addition to the model itself). In LightDNN, the stopping threshold of scaling optimization is set to 0.005 and all the scaling optimization decisions are made within 1 seconds. We have two key observations from the result.

First, LightDNN consistently achieves higher accuracies than baseline approaches under different available RAM or VRAM resources across all 8 workloads. This result indicates that our block-grained scaling is able to deliver state-of-the-art inference accuracy under a given memory budget and deadline constraint. This is because based on the precise prediction of inference latencies, LightDNN is able to use the largest possible model to improve inference accuracy. Overall, LightDNN increases the inference accuracy by 12.26% compared to those of baselines.

Second, we observe that when the available memory becomes smaller, LightDNN achieves more improvements in inference accuracy. On average, it achieves 14.96% higher accuracy when the available memory is smaller than 30MB. This is because the scaling optimizer in LightDNN can find the blocks having the largest influence on accuracy while first scaling down less important blocks. Small descendant models in LightDNN hence benefits from these important blocks while other baseline models do not.

Moreover, we compare LightDNN with *FLOP scaling techniques* under the same computational costs. Fig. 10 shows that LightDNN consistently archives higher accuracies for all workloads. This because FLOP scaling techniques reduce computational costs by skipping parts of a DNN. In contrast, our approach trades off accuracy and FLOPs by selecting descendant blocks of different sizes, and each descendant block can be viewed as an approximation of the original block and its accuracy loss is small. We can also observe that our approach provide much large scaling space than CGNet. Overall, our approach increases accuracies by an average of 17.84%.

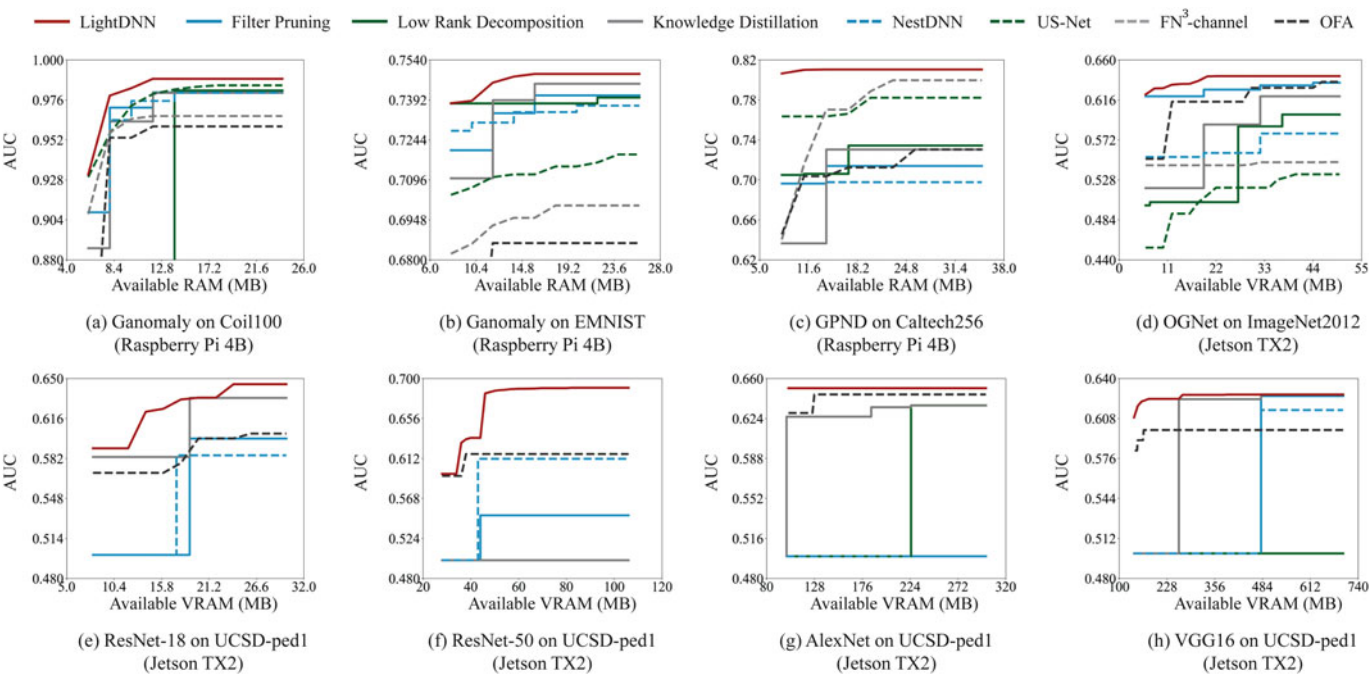


Fig. 9. Comparison of inference accuracies between LightDNN and layer removing and nested network techniques.

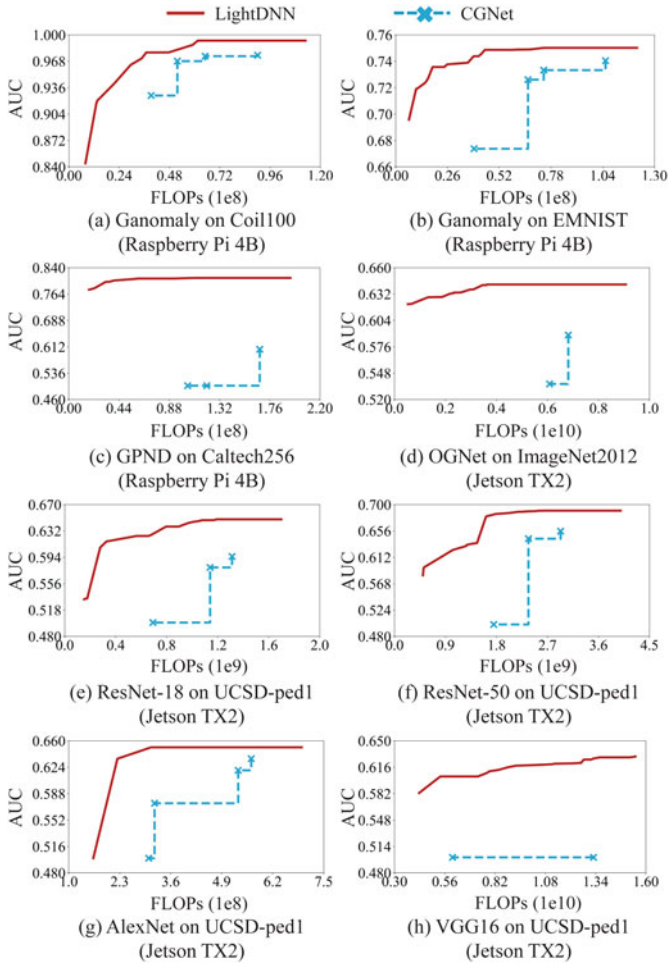


Fig. 10. Comparison of inference accuracies between LightDNN and FLOP scaling techniques.

Discussion of Stopping Threshold σ. In scaling, the value of σ determines both optimization time and inference accuracy. In this evaluation, we take ResNet-50 as an example and test its scaling optimization on Raspberry Pi (the less powerful edge platform). Six values of σ ranging from 0 to 1e-3 are tested. For each value, we sample 100 different memory constraints and deadlines to construct 100 different optimization problems. Table 4 reports each threshold's accuracy gap to the actual optimal solution (without early stopping) and the average optimization time. We can see that a larger value of σ brings faster optimization time by sacrificing negligible accuracy. For example, when $\sigma = 1e-3$, the optimization time is nearly 30% shorter than the time when $\sigma = 0$, while only causing 0.1% accuracy drop. Hence in experiments, the stopping threshold is set to 0.005 in order to provide fast optimization time and high accuracy.

TABLE 4
The Impact of σ on Optimization Accuracy and Time

σ	The accuracy gap to the optimal solution	Average optimization time (s)
0	0%	0.2151
1e-7	0%	0.2123
1e-6	0%	0.1955
1e-5	0%	0.1942
1e-4	0.023%	0.1777
1e-3	0.070%	0.1521

TABLE 5
Comparsion of the Number of Scaling Options, Searching Time, and Training Time Between LightDNN and NAS Techniques

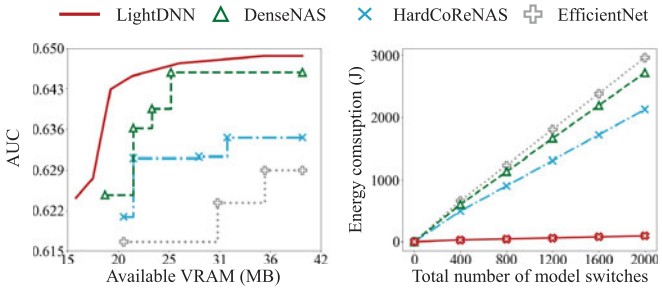
	#scaling options	searching time (h)	training time (h)
EfficientNet	5	0	402.5
DenseNAS	5	0.64	2.33
HardCoRe-NAS	5	0.05	1
LightDNN	**1679616**	**0**	**0.42**

4.4 Comparison to EfficientNet and NAS Techniques

In this section, we compare LightDNN with EfficientNet [48] and two latest NAS techniques (DenseNAS [27] and HardCoRe-NAS [28]): (1) EfficientNet: the network's width, depth, and input resolution are defined by three equations: α^{φ}, β^{φ}, and γ^{φ}, in which α, β, and γ are determined by the grid search. Subsequently, to provide multiple optional models in scaling, this technique generates optimal model architectures by setting different values of φ. (2) HardCoRe-NAS and DenseNAS build a super network as the entire search space, and search and generate multiple optimal sub-networks from it. Specifically, DenseNAS needs training in the searching process, and the searched sub-network requires re-training from scratch. HardCoRe-NAS fully trains the super network before searching, and the searched sub-network needs fine-tuning.

Evaluation Settings. We compare all techniques using the video anomaly detection application (the UCSD-Peds1 dataset). The three NAS techniques search and generate 5 descendant models based on MobileNetV2 [49]. For EfficientNet, the parameters α, β, γ are set to 1.2, 1.1, and 1.15. This is reasonable because EfficientNet obtains these values from ImageNet and proves their generality to other DNNs. For LightDNN, the descendant blocks are learned from blocks in ResNet-18.

Overheads in Offline Model Searching and Generation. Table 5 lists the number of scaling options and their generation time, including searching time and training time. We can see that among four techniques, LightDNN provides much larger scaling options while using less generation time. This is because for each descendant model, the three NAS techniques needs expensive searching and retraining time. EfficientNet needs negligible searching time because it only searches one parameter (φ). LightDNN also needs no search time and its re-training of blocks is much quicker than the re-training of entire DNNs in NAS techniques. The combination of these blocks provides large scaling space. In contrast, to provide the same number of scaling options, EfficientNet, DenseNAS, and HardCoRe-NAS take 2.11×10^8, 1.05×10^6, and 3.53×10^5 GPU hours, respectively.



(a) Comparison of inference accuracies

(b) Comparison of energy consumptions

Fig. 11. Comparison of inference accuracies and energy consumption between LightDNN and NAS techniques.

Accuracy and Energy Consumption in Online Scaling. Fig. 11 compares the inference accuracies and energy consumptions of LigthDNN and the three NAS techniques on Jetson TX2 (the deadline is 150ms). Fig. 11a shows that LightDNN achieves the highest under different available memories. EfficientNet results in the lowest accuracy because it only uses one parameter in network search and hence its optional networks have the largest differences. Fig. 11b shows that LightDNN significantly reduces energy consumption in scaling (reducing by 95.9% in average). This is because the NAS techniques perform model scaling via switching the entire networks. The average page in and page out size is 26MBs in EfficientNet, 25.5MBs in DenseNAS and HardCoReNAS, while this size is only 10MBs in LightDNN. Another reason of high scaling overhead in the NAS techniques is that a MobileNet-based network has much more network layers than a standard CNN (e.g., ResNet-18) when both networks have the same model size, thus incurring larger model initialization overheads.

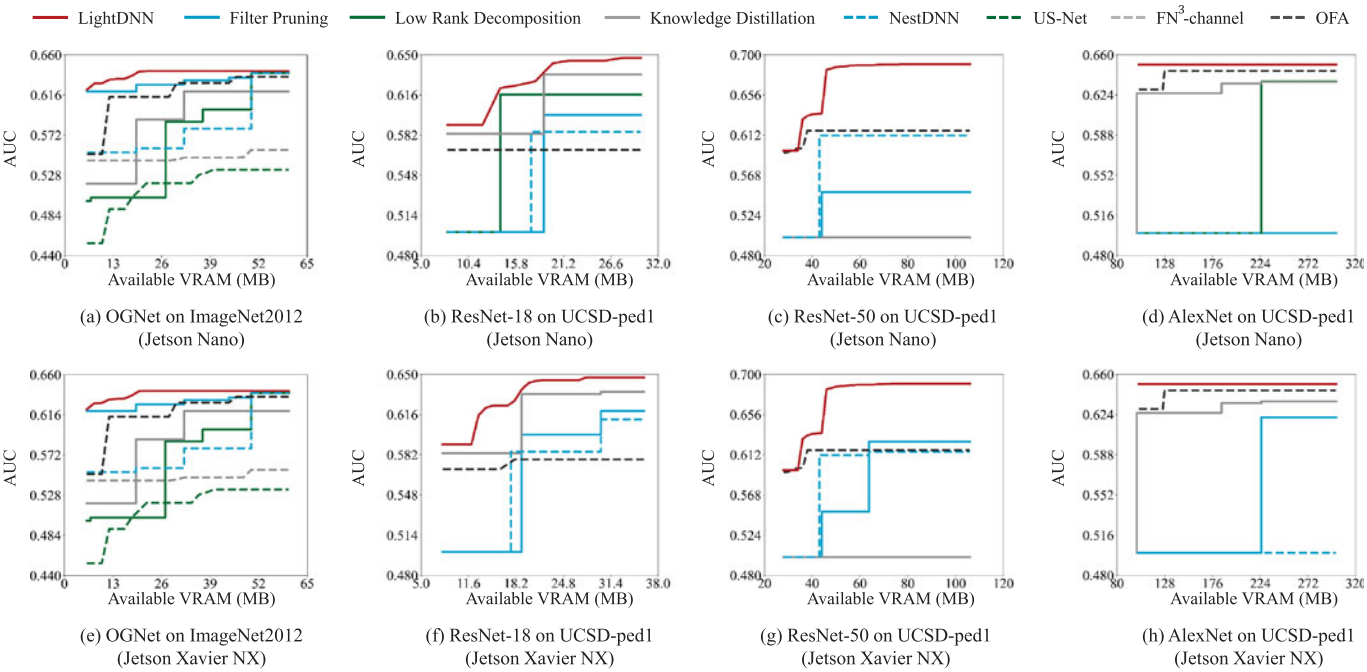
4.5 Discussions of Applicability of LightDNN

Generality of LightDNN. LightDNN represents the first framework that supports block-grained scaling of DNNs for anomaly detection systems. In this work, we selected GAN-based models (GANomaly, OGNet, and GPND) and self-training models (AlexNet, VGG and ResNet) as representative DNNs to implement and evaluate our approach. Nested network and FLOP scaling techniques result in poor accuracies because they introduce extra operations into the training process, thus amplifying the instability in GAN-based training and the number of wrong pseudo-labels in self-training. LightDNN works well for all these models because it generates descendant blocks based on *trained* DNNs and finetunes them in another training environment, hence it is independent of the unstable training process of the original DNN.

LightDNN can be generalized to support a wide range of prevalent DNNs with convolutional layers. In such a DNN, its blocks can be identified and transformed into descendant blocks to support block-grained scaling. Examples include WideResNet [50], feature map exploitation (SENet18 [51]), attention (Residual Attention Network (RAN) [52]), and lightweight DNN (MobileNet [49]). Our approach is inapplicable when any filter in the DNN cannot be removed, e.g., this DNN only has one filter in the extreme case.

Discussion of Other Powerful NVIDIA Jetson Platforms. We added two NVIDIA Jetson platforms to show the scalability of LightDNN: NVIDIA Jetson Nano and Xavier NX. According to the performance difference of these two platforms, we set the deadlines on Jetson Nano and Jetson Xavier NX as twice and two-thirds of the deadline on Jetson TX2, respectively. The evaluation follows the settings of Section 4.3.1 and VGG16 is not tested because its size exceeds the memory capacity of Jetson Nano.

Fig. 12 demonstrates the inference accuracies of the eight techniques under different available memories. We can see



(a) OGNet on ImageNet2012 (Jetson Nano)

(b) ResNet-18 on UCSD-ped1 (Jetson Nano)

(c) ResNet-50 on UCSD-ped1 (Jetson Nano)

(d) AlexNet on UCSD-ped1 (Jetson Nano)

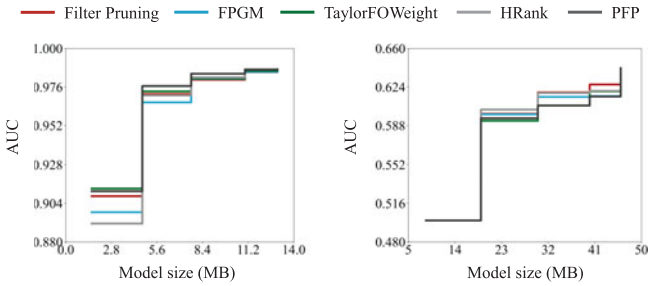
(e) OGNet on ImageNet2012 (Jetson Xavier NX)

(f) ResNet-18 on UCSD-ped1 (Jetson Xavier NX)

(g) ResNet-50 on UCSD-ped1 (Jetson Xavier NX)

(h) AlexNet on UCSD-ped1 (Jetson Xavier NX)

Fig. 12. Comparison of inference accuracies between LightDNN and layer removing and nested network techniques in NVIDIA Jetson Nano and NVIDIA Jetson Xavier NX.



(a) Ganomaly on Coil100 (Raspberry Pi 4B)

(b) ResNet-18 on UCSD-ped1 (Jetson TX2)

Fig. 13. Implementation of LightDNN using 5 model compression techniques.

LightDNN achieves the highest accuracies in all cases. The result indicates that LightDNN has a good scalability across various constraints and devices of different computation resources. In contrast, the seven compared techniques show unstable and inconsistent performances across different models and datasets, because they rely on the unstable GAN and self-learning training processes. For example, low rank decomposition achieves high accuracy on Jetson Nano (Fig. 12b) but fails to provide any available model that satisfies the deadline on Jetson Xavier NX (Fig. 12f).

Applicability to Model Compression Techniques. LightDNN currently uses filter pruning [32] to compress blocks and our approach can be implemented using other model compression techniques. First, Filter Pruning via Geometric Median (FPGM) [53] directly calculates filters' importances and removes unimportant ones to compress a network. Second, data-dependent techniques such as TaylorFOWeight [54], High-Rank Feature Map (HRank) [55], and Provable Filter Pruning (PFP) [56] use training samples to estimate a filter's importance. We implemented all the above five compression techniques and tested Ganomaly and ResNet-18 as examples. The results in Fig. 13 show that these techniques produce models of similar accuracies, because the descendant blocks generated by different compression techniques learn from the same original block in our approach.

Applicability to Deep Learning Platforms. LightDNN is currently implemented in PyTorch Mobile [43] to support deep learning models at edge. It is convenient to extend LightDNN to support other mainstream DL platforms such as TensorFLow [57] and Caffe [58]. In LightDNN, the block extraction and switching can be implemented using dozens of lines of code in these platforms; the block training uses the standard filter pruning technique; and both the profiling and the online scaling of blocks are independent of the underlying platforms.

5 Conclusion

This paper presents the design, implementation and evaluation of LightDNN, a framework that enables lightweight and accurate scaling of DNNs for anomaly detection systems. Our approach can provide large scaling space for a DNN by generating descendant blocks for its blocks offline. Based on the precise perdition of the DNN's inference latency at run-time, LightDNN can search the optimal combination of these blocks to maximize accuracy under deadline constraints. Extensive evaluation results prove the efficacy and practicality of LightDNN.

Acknowledgments

The authors would like to thank the anonymous reviewers for careful review of our article. Rui Han is the corresponding author.

References

[1] B. R. Kiran, D. M. Thomas, and R. Parakkal, "An overview of deep learning based methods for unsupervised and semi-supervised anomaly detection in videos," *J. Imag.*, vol. 4, no. 2, 2018, Art. no. 36.

[2] R. Chalapathy and S. Chawla, "Deep learning for anomaly detection: A survey," 2019, *arXiv:1901.03407v2*.

[3] R. Han *et al.*, "SlimML: Removing non-critical input data in large-scale iterative machine learning," *IEEE Trans. Knowl. Data Eng.*, vol. 33, no. 5, pp. 2223–2236, May 2021.

[4] J. An and S. Cho, "Variational autoencoder based anomaly detection using reconstruction probability," *Special Lecture IE*, vol. 2, no. 1, pp. 1–18, 2015.

[5] S. Pidhorskyi, R. Almohsen, and G. Doretto, "Generative probabilistic novelty detection with adversarial autoencoders," in *Proc. 32nd Int. Conf. Neural Inf. Process. Syst.*, 2018, pp. 6823–6834.

[6] D. Gong *et al.*, "Memorizing normality to detect anomaly: Memory-augmented deep autoencoder for unsupervised anomaly detection," in *Proc. IEEE/CVF Int. Conf. Comput. Vis.*, 2019, pp. 1705–1714. [Online]. Available: https://doi.org/10.1109/ICCV.2019.00179

[7] S. Akcay, A. A. Abarghouei, and T. P. Breckon, "GANomaly: Semi-supervised anomaly detection via adversarial training," in *Proc. Asian Conf. Comput. Vis.*, C. V. Jawahar, H. Li, G. Mori, and K. Schindler, Eds., vol. 11363, 2018, pp. 622–637. [Online]. Available: https://doi.org/10.1007/978–3-030-20893-6_39

[8] M. Z. Zaheer, J. Lee, M. Astrid, and S. Lee, "Old is gold: Redefining the adversarially learned one-class classifier training paradigm," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2020, pp. 14171–14181. [Online]. Available: https://doi.org/10.1109/CVPR42600.2020.01419

[9] M. Hasan, J. Choi, J. Neumann, A. K. Roy-Chowdhury, and L. S. Davis, "Learning temporal regularity in video sequences," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 733–742. [Online]. Available: https://doi.org/10.1109/CVPR.2016.86

[10] M. Georgescu, A. Barbalau, R. T. Ionescu, F. S. Khan, M. Popescu, and M. Shah, "Anomaly detection in video via self-supervised and multi-task learning," 2020, *arXiv:2011.07491*.

[11] G. Pang, C. Yan, C. Shen, A. van den Hengel, and X. Bai, "Self-trained deep ordinal regression for end-to-end video anomaly detection," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2020, pp. 12170–12179. [Online]. Available: https://doi.org/10.1109/CVPR42600.2020.01219

[12] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," in *Proc. Int. Conf. Learn. Representations*, 2015, pp. 1–14.

[13] Y. H. Oh *et al.*, "A portable, automatic data qantizer for deep neural networks," in *Proc. 27th Int. Conf. Parallel Archit. Compilation Techn.*, 2018, pp. 1–14.

[14] H. Li, C. Hu, J. Jiang, Z. Wang, Y. Wen, and W. Zhu, "JALAD: Joint accuracy-and latency-aware deep structure decoupling for edge-cloud execution," in *Proc. IEEE 24th Int. Conf. Parallel Distrib. Syst.*, 2018, pp. 671–678.

[15] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing energy-efficient convolutional neural networks using energy-aware pruning," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 5687–5695.

[16] B. Reagen *et al.*, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit.*, 2016, pp. 267–278.

[17] B. Fang, X. Zeng, and M. Zhang, "NestDNN: Resource-aware multi-tenant on-device deep learning for continuous mobile vision," in *Proc. 24th Annu. Int. Conf. Mobile Comput. Netw.*, New York, NY, USA, 2018, pp. 115–127. [Online]. Available: https://doi.org/10.1145/3241539.3241559

[18] J. Yu and T. S. Huang, "Universally slimmable networks and improved training techniques," in *Proc. IEEE/CVF Int. Conf. Comput. Vis.*, 2019, pp. 1803–1811. [Online]. Available: https://doi.org/10.1109/ICCV.2019.00189
[19] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han, "Once-for-all: Train one network and specialize it for efficient deployment," in *Proc. Int. Conf. Learn. Representations.*, 2020. [Online]. Available: https://openreview.net/forum?id=HylxE1HKwS
[20] Y. Cui *et al.*, "Fully nested neural network for adaptive compression and quantization," in *Proc. 29th Int. Joint Conf. Artif. Intell. 17th Pacific Rim Int. Conf. Artif. Intell.*, 2020, pp. 2080–2087. [Online]. Available: https://doi.org/10.24963/ijcai.2020/288
[21] G. Huang, D. Chen, T. Li, F. Wu, L. Van Der Maaten, and K. Q. Weinberger, "Multi-scale dense convolutional networks for efficient prediction," 2017, *arXiv:1703.09844*.
[22] S. Laskaridis, S. I. Venieris, H. Kim, and N. D. Lane, "HAPI: Hardware-aware progressive inference," in *Proc. 39th Int. Conf. Comput.-Aided Des.*, 2020, pp. 1–9.
[23] Z. Wu, T. Nagarajan, A. Kumar, S. Rennie, L. S. Davis, K. Grauman, and R. Feris, "BlockDrop: Dynamic inference paths in residual networks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 8817–8826.
[24] A. Veit and S. Belongie, "Convolutional networks with adaptive inference graphs," in *Proc. Eur. Conf. Comput. Vis.*, 2018, pp. 3–18.
[25] W. Hua, Y. Zhou, C. De Sa , Z. Zhang, and G. E. Suh, "Channel gating neural networks," 2018, *arXiv:1805.12549*.
[26] M. Tan and Q. V. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *Proc. 36th Int. Conf. Mach. Learn.*, Long Beach, California, USA, 2019, vol. 97, pp. 6105–6114. [Online]. Available: http://proceedings.mlr.press/v97/tan19a.html
[27] J. Fang, Y. Sun, Q. Zhang, Y. Li, W. Liu, and X. Wang, "Densely connected search space for more flexible neural architecture search," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Seattle, WA, USA, 2020, pp. 10625–10634.
[28] N. Nayman, Y. Aflalo, A. Noy, and L. Zelnik, "HardCoRe-NAS: Hard constrained differentiable neural architecture search," in *Proc. 38th Int. Conf. Mach. Learn.*, vol. 139, 2021, pp. 7979–7990.
[29] J. Yu, L. Yang, N. Xu, J. Yang, and T. S. Huang, "Slimmable neural networks," in *Proc. Int. Conf. Learn. Representations.*, 2019, pp. 1–12. [Online]. Available: https://openreview.net/forum?id=H1gMCsAqY7
[30] S. Sinha, Z. Zhao, A. Goyal, C. Raffel, and A. Odena, "Top-k training of gans: Improving GAN performance by throwing away bad samples," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2020, pp. 1–14. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/a851bd0d418b13310dd1e5e3ac7318ab-Abstract.html
[31] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.
[32] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient ConvNets," in *Proc. Int. Conf. Learn. Representations*, 2016, pp. 1–13.
[33] D. T. Nguyen, Z. Lou, M. Klar, and T. Brox, "Anomaly detection with multiple-hypotheses predictions," in *Proc. IEEE Int. Conf. Mach. Learn.*, vol. 97, 2019, pp. 4800–4809. [Online]. Available: http://proceedings.mlr.press/v97/nguyen19b.html
[34] D. Abati, A. Porrello, S. Calderara, and R. Cucchiara, "Latent space autoregression for novelty detection," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2019, pp. 481–490.
[35] E. Arazo, D. Ortego, P. Albert, N. E. O'Connor, and K. McGuinness, "Pseudo-labeling and confirmation bias in deep semi-supervised learning," in *Proc. Int. Joint Conf. Neural Netw.*, 2020, pp. 1–8. [Online]. Available: https://doi.org/10.1109/IJCNN48605.2020.9207304
[36] R. Han, C. H. Liu, S. Li, S. Wen, and X. Liu, "Accelerating deep learning systems via critical set identification and model compression," *IEEE Trans. Comput.*, vol. 69, no. 7, pp. 1059–1070, Jul. 2020.
[37] R. Han, S. Li, X. Wang, C. H. Liu, G. Xin, and L. Y. Chen, "Accelerating gossip-based deep learning in heterogeneous edge computing platforms," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 7, pp. 1591–1602, Jul. 2020.
[38] R. Han *et al.*, "Accurate differentially private deep learning on the edge," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 7, pp. 1591–1602, Jul. 2021.
[39] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, "Compression of deep convolutional neural networks for fast and low power mobile applications," in *Proc. Int. Conf. Learn. Representations*, 2015, pp. 1–16.
[40] Y. Tian, D. Krishnan, and P. Isola, "Contrastive representation distillation," in *Proc. Int. Conf. Learn. Representations*, 2020.
[41] Downsampled imagenet datasets. Accessed: Dec. 30, 2020. [Online]. Available: http://image-net.org/download-images
[42] S. Liu and W. Deng, "Very deep convolutional neural network based image classification using small training sample size," in *Proc. 3rd IAPR Asian Conf. Pattern Recognit.*, 2015, pp. 730–734.
[43] Pytorch. Accessed: Jan. 01, 2021. [Online]. Available: https://pytorch.org/
[44] The columbia university image library(coil-100) dataset. Accessed: Oct. 01, 2020. [Online]. Available: https://www1.cs.columbia.edu/CAVE/software/softlib/coil-100.php
[45] The emnist dataset. Accessed: Mar. 28, 2019. [Online]. Available: https://www.nist.gov/itl/products-and-services/emnist-dataset
[46] The caltech 256 dataset. Accessed: Oct. 02, 2019. [Online]. Available: http://www.vision.caltech.edu/ImageDatasets/Caltech256/
[47] Ucsd-peds1 dataset. Accessed: Feb. 27, 2013. [Online]. Available: http://www.svcl.ucsd.edu/projects/anoma-ly/dataset.html
[48] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *Proc. 36th Int. Conf. Mach. Learn.*, 2019, pp. 6105–6114.
[49] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 4510–4520.
[50] S. Zagoruyko and N. Komodakis, "Wide residual networks," in *Proc. Brit. Mach. Vis. Conf.*, Sep. 2016, pp. 87.1–87.12.
[51] J. Hu, L. Shen, S. Albanie, G. Sun, and E. Wu, "Squeeze-and-excitation networks," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 42, no. 8, pp. 2011–2023, Aug. 2020. [Online]. Available: https://doi.org/10.1109/TPAMI.2019.2913372
[52] F. Wang *et al.*, "Residual attention network for image classification," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 3156–3164.
[53] Y. He, P. Liu, Z. Wang, Z. Hu, and Y. Yang, "Filter pruning via geometric median for deep convolutional neural networks acceleration," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2019, pp. 4340–4349.
[54] P. Molchanov, A. Mallya, S. Tyree, I. Frosio, and J. Kautz, "Importance estimation for neural network pruning," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2019, pp. 11264–11272.
[55] M. Lin, R. Ji, Y. Wang, Y. Zhang, B. Zhang, Y. Tian, and L. Shao, "HRank: Filter pruning using high-rank feature map," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2020, pp. 1529–1538.
[56] L. Liebenwein, C. Baykal, H. Lang, D. Feldman, and D. Rus, "Provable filter pruning for efficient neural networks," 2019, *arXiv:1911.07412*.
[57] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. 12th USENIX Conf. Oper. Syst. Des. Implementation.*, 2016, pp. 265–283.
[58] Caffe. Accessed: Feb. 13, 2020. [Online]. Available: http://caffe.berkeleyvision.org/

Qinglong Zhang is currently working toward the master's degree with the School of Computer Science and Technology, Beijing Institute of Technology. His research interests include edge intelligence and deep learning applications.

Rui Han received the MSc (Hons.) degree in 2010 from Tsinghua University, China, and the PhD degree from Imperial College London, U.K., in 2014. He is currently an associate professor with the School of Computer Science and Technology, Beijing Institute of Technology, China. He has authored or coauthored more than 40 papers in the areas of his research fields including papers at MobiCOM, the *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, *IEEE Transactions on Knowledge and Data Engineering*, INFOCOM, and ICD. His research interests include system optimization for cloud data center workloads (in particular highly parallel services and deep learning applications) and CS.

Gaofeng Xin is currently working toward the undergraduate degree with the School of Computer Science and Technology, Beijing Institute of Technology. His research interests include optimization of big data system for machine learning and deep learning workloads.

Chi Harold Liu (Senior Member, IEEE) received the BEng degree in electronic and information engineering from Tsinghua University, China, in 2006 and the PhD degree in electronic engineering from Imperial College, U.K., in 2010. He is currently a full professor and the vice dean of the School of Computer Science and Technology, Beijing Institute of Technology, China. Before moving to academia, he was with IBM Research, China, as a staff researcher and the project manager from 2010 to 2013, a postdoctoral researcher with Deutsche Telekom Laboratories, Germany, in 2010, and a research staff member with IBM T. J. Watson Research Center, USA, in 2009. He has authored or coauthored more than 100 prestigious conference and journal papers, owned 26 EU/U.K./U.S./Germany/Spain/China patents, and the book editor of 11 books published by Taylor & Francis Group, USA, and China Machine Press, China. His research interests include big data analytics, mobile computing, and machine learning. He was the associate editor for the *IEEE Transactions on Network Science and Engineering*, an area editor of *KSII Transaction on Internet and Information Systems*, the symposium chair of IEEE ICC 2020 on Next Generation Networking, and the (lead) guest editor of the *IEEE Transactions on Emerging Topics in Computing* and the *IEEE Sensors Journal*. He was also the general chair of the IEEE SECON'13 Workshop on IoT Networking and Control, IEEE WCNC'12 Workshop on IoT Enabling Technologies, and ACM UbiComp'11 Workshop on Networking and Object Memories for IoT. He was a consultant to Asian Development Bank, Bain and Company, and KPMG, USA, and the peer reviewer for Qatar National Research Foundation, National Science Foundation, China, Ministry of Education and Ministry of Science and Technology, China. He is a fellow of IET, British Computer Society, and Royal Society of Arts. He was the recipient of IBM First Plateau Invention Achievement Award in 2012, Excellent Editor Award 2021 for *IEEE Transactions on Network Science and Engineering*, ACM SigKDD'21 Best Paper Runner-up Award, and IEEE DataCom'16 Best Paper Award.

Guoren Wang (Senior Member, IEEE) received the BSc, MSc, and PhD degrees from the Department of Computer Science, Northeastern University, Shenyang, China, in 1988, 1991, and 1996, respectively. He is currently a full professor and the director of Institute of Data Science and Knowledge Engineering, Department of Computer Science and Technology, Beijing Institute of Technology, Beijing, China. He was an assistant president with Northeastern University, China. He has authored or coauthored more than 150 research papers in top conferences and journals, including the *IEEE Transactions on Knowledge and Data Engineering*, ICDE, SIGMOD, VLDB, etc. His research interests include XML data management, query processing and optimization, bioinformatics, high-dimensional indexing, parallel database systems, and cloud data management.

Lydia Y. Chen (Senior Member, IEEE) received the BA degree from National Taiwan University, and the PhD from Pennsylvania State University. She is currently an associate professor with the Department of Computer Science, Technology University Delft. Prior to joining TU Delft, she was a research staff member with the IBM Zurich Research Lab from 2007 to 2018. She has authored or coauthored more than 80 papers in journals, including the *IEEE Transactions on Distributed Systems*, *IEEE Transactions on Service Computing*, and conference proceedings, e.g., INFOCOM, Sigmetrics, DSN, and Eurosys. Her research interests include dependability management, resource allocation and privacy enhancement for large scale data processing systems and services. She was the recipient of the best paper awards at CCgrid15 and eEnergy15.