ADaPS: Adaptive Data Partitioning to Parallelize CNN Inference on Resource-Constrained Hardware

Jaume Mateu Cuadrat
Seoul National University
Seoul, South Korea
jaume@csap.snu.ac.kr

Bernhard Egger
Seoul National University
Seoul, South Korea
bernhard@csap.snu.ac.kr

Abstract

The growing adoption of AI applications has led to an increased demand for deploying neural networks on diverse device platforms. However, even modest networks now require specialized hardware for efficient execution due to their rising computational cost. To address this, distributed execution across connected, resource-constrained devices is gaining importance. While prior work relies on empirical models or supports limited partitioning, we present ADaPS, a novel framework for distributing Convolutional Neural Networks (CNNs) inference workloads across heterogeneous embedded devices. Our analytical model partitions the height and width dimensions of 4D tensors and explores layer fusion opportunities, accounting for compute, memory, and communication constraints. ADaPS efficiently explores the vast partitioning space using a tree-based hybrid optimization algorithm combining Alpha-Beta pruning and dynamic programming. Evaluations on multiple CNNs and device configurations show that ADaPS is able to improve inference latency by up to 1.2× on average while significantly reducing data transfers compared to state-of-the-art methods.

CCS Concepts: • **Computing methodologies** → *Distributed artificial intelligence*; *Parallel programming languages*; **Game tree search**; **Heuristic function construction**.

Keywords: Convolutional network, data partitioning, min-max, parallelization

ACM Reference Format:
Jaume Mateu Cuadrat and Bernhard Egger. 2025. ADaPS: Adaptive Data Partitioning to Parallelize CNN Inference on Resource-Constrained Hardware. In *Proceedings of the 26th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '25), June 16–17, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3735452.3735532

LCTES '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1921-9/25/06
https://doi.org/10.1145/3735452.3735532

1 Introduction

Initially valued for their object detection capabilities in images and video streams, CNNs now power critical applications across diverse domains including medical diagnostics [3], intelligent surveillance systems [16], and advanced image processing [2]. Over the past decade, neural networks have evolved into massive architectures that require billions of parameters and compute operations. As these networks continue to grow in complexity, the challenge of efficient hardware mapping extends beyond small devices to increasingly affect mainstream computing platforms.

For edge devices and mid-range consumer CPUs, even established networks such as ResNet50 [5] and VGG16 [19] already strain hardware resources due to their substantial memory requirements compared to more efficient architectures like MobileNet [6]. The problem is further exacerbated by newer architectures such as NASNet [26] and Inception v4 [21], which feature complex Directed Acyclic Graph (DAG) with multiple execution paths, significantly expanding possible execution orders and complicating optimization.

This paper specifically addresses the needs of end-users and small-to-medium enterprises that require low-latency inference capabilities for relatively large networks on commodity hardware. In these scenarios, batch sizes are typically limited, and minimizing inference latency is the primary concern. Existing approaches to this challenge generally fall into two categories: model partitioning and data partitioning. Model partitioning distributes different layers of the CNN across multiple devices [4, 10, 13, 22], while data partitioning divides the input tensors within individual layers across devices [7, 11, 12, 20, 23–25]. A number of implementation strategies have been proposed. Some techniques employ a master node for coordination [11, 20, 25], others determine partitioning during compilation [7, 12], and more sophisticated approaches consider both device heterogeneity and inter-device communication costs [23, 24].

While data partitioning can effectively reduce inference latency, determining the optimal division strategy across the tensor dimensions remains a critical challenge. CNN tensors can potentially be partitioned along four dimensions: input channels, output channels, height, and width. Most existing approaches, including this work, focus on height and width partitioning due to its generality.

Table 1. Comparison of related work. *H*, *W*, and *C* denote partitioning in the *H*eight, *W*idth, and *C*hannel dimension.

Paper	Search Type	Heterogeneous Devices	Partitioning Directions	Communication	Layer Fusion
MoDNN [11]	Layer by layer	No	H or W	Centralized	No
DeepThings [25]	Fixed partition	No	H or W	Centralized	No
DeeperThings [20]	Fixed partition	No	C	Centralized	Yes
CoEdge [23]	Layer by layer	Yes	H or W	Centralized	No
DeepSlicing [24]	Grouped layers	Yes	W	Centralized	Yes
EdgeFlow [7]	Analytical model	Yes	H	Decentralized	Yes
BBGraP [12]	Fixed partition	No	H or W or C	Decentralized	No
ADaPS (this work)	Analytical model	Yes	H or W	Decentralized	Yes

The presented framework, ADaPS, builds upon and extends state-of-the-art approaches through several key innovations. ADaPS automates the exploration of a comprehensive search space across all supported tensor dimensions for heterogeneous device configurations, while identifying optimal partitioning strategies for each layer that simultaneously minimize data transfer overhead and redundant computations. The framework employs advanced search techniques inspired by Minmax [17] and Alpha-beta pruning [8] game search algorithms to efficiently navigate the complex solution space. Furthermore, ADaPS incorporates detailed network characteristics (bandwidth, latency) and device-specific computational capabilities to guide optimization decisions, enabling support for flexible hardware combinations.

To demonstrate the practical applicability of ADaPS in real-world heterogeneous edge environments, we evaluate the performance of the presented framework across two to four-devices configurations comprising various Raspberry Pi boards.

In summary, the contributions of this work are

- A flexible technique for inference workload partitioning along a tensor's height and width dimensions into partitions for heterogeneous edge device configurations.
- An efficient search space exploration mechanism using a specialized tree structure and pruning methods that significantly reduce optimization time.
- A exploration of opportunities for layer fusion among the partitions that considers the end-to-end characteristics of a CNN architecture to maximize performance.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 introduces the key concepts underlying our implementation. Section 4 describes the ADaPS framework and its operation in detail. Section 5 evaluates our approach against state-of-the-art methods, and Section 6 concludes this work.

2 Related Work

Recent research has focused on improving inference latency through three primary approaches: data partitioning, model partitioning, and optimization techniques (Table 1). *Model partitioning* has emerged as the prevalent strategy, with significant contributions from works such as Combining [13], and Coin [22], and other notable systems [4, 10]. Other techniques explore *data partitioning*, including MoDNN [11], DeepThings [25], DeeperThings [20], CoEdge [23], DeepSlicing [24], EdgeFlow [7], and BBGraP [12]. These approaches offer various strategies for distributing the computational workload across multiple devices, with a particular emphasis on resource-constrained platforms where traditional inference methods prove impractical due to memory and computational limitations

2.1 Centralized vs. Decentralized Approaches

Distributed CNN inference across multiple devices requires coordination to ensure a correct execution order and data synchronization. These coordination mechanisms generally fall into two categories. In **centralized approaches**, a designated master device orchestrates synchronization among all participating devices in the network. The master device dynamically creates tasks and distributes them to idle devices, ensuring a balanced workload distribution and maximizing resource utilization. This paradigm offers simplified coordination but may introduce a bottleneck and a single point of failure at the master device. **Decentralized approaches**, on the other hand, distribute the responsibility for synchronization and correctness to the individual devices. While this eliminates the overhead associated with a master device, this approach suffers from a significantly increased complexity. The compiler must incorporate performance models to calculate a partitioning scheme that balances execution equally on all devices, considering device performance and network bandwidth. Recent works that employ a decentralized approach include EdgeFlow [7] and BBGraP [12].

2.2 Search Space Exploration and Layer Fusion

The partitioning of CNN workloads across multiple devices presents a computationally intensive optimization challenge with an exponentially growing search space. Sophisticated exploration strategies are necessary to find the best divisions that balance the computational load while minimizing the

communication overhead, particularly as the number of devices and the network complexity increases. In other words, the efficiency of a partitioning heavily depends on the refinement of the search space exploration techniques. Earlier works, such as MoDNN [11] and CoEdge [23], optimize each layer individually and synchronize after each division on the master device. More advanced techniques, such as those employed by DeepThings [25] and DeeperThings [20], minimize data synchronization *within* layers while still synchronizing *between* layers. DeeperThings extends this concept by exploring both input and output channel partitioning. This is particularly beneficial for fully-connected layers towards the end of the network where the weights are significantly larger than the feature maps and spatial dimensions become indivisible.

In distributed inference, layer fusion is critical to achieving maximum performance because it minimizes inter-device data movements. Existing approaches, however, exhibit significant limitations. BBGraP [12] implements data partitioning layer-by-layer, dividing the output based solely on the number of available devices, but fails to consider device-specific parameters or capabilities. DeepSlicing [24] advances the state-of-the-art by searching across multiple fused layers. The search is limited to a fixed number of layers rather than optimizing the partitioning strategy itself. EdgeFlow [7] addresses the data transfer overhead between devices through a linear optimization problem that considers only the previous layer to reduce inter-layer data transfers. However, this approach introduces restrictions on output patterns, as it prohibits data overlaps essential for convolutional operations where stride and width parameters naturally create input overlaps. Consequently, if the previous layer cannot produce overlapped outputs, EdgeFlow cannot perform layer fusion, resulting in increased data transfers between the devices. ADaPS addresses these limitations by implementing a comprehensive approach to layer fusion that considers the entire CNN architecture while supporting flexible partitioning along multiple tensor dimensions. ADaPS dynamically optimizes the partitioning strategies based on computational capabilities and communication constraints, enabling the generation of more efficient execution plans on heterogeneous hardware configurations.

3 CNN Computational Graph Partitioning

This section introduces the concepts and formal definitions of our partitioning framework. We first categorize CNN operators according to their partitioning characteristics and then present the mathematical formulations to determine the input requirements based on the partitioned outputs.

3.1 Operator Classification

When partitioning CNN workloads, different operators exhibit distinct behaviors that affect how they can be divided efficiently. Building on formulations from EdgeFlow [7] and BBGraP [12], our framework classifies operators into three types based on their mathematical properties and partitioning constraints. This classification is shown in Table 2. Type 1 operators comprise operators with weights, padding, or stride parameters, such as convolutional and pooling layers. These operators require special handling during partitioning due to their neighborhood dependencies, i.e., output elements depend on multiple adjacent input elements. Type 2 operators include element-wise operators, such as multiplication or addition, and activation functions that can be partitioned without introducing additional dependencies except the direct input-output mappings. Type 3 operators are operators that transform tensor dimensions or aggregate information across dimensions. Operators of this type are not partitioned by our framework.

Table 2. Classification of CNN Operators

Classification	Operators
Type 1	Conv, DWConv, Pool, AvgPool, MaxPool
Type 2	Mul, Add, HardSwish, HardSigmoid, BatchNorm, Relu, GeMM
Type 3	Concat, Reshape, ReduceMean, Flatten, Dropout, Fully-connected (FL)

3.2 Partition Representation and Tensor Indexing

Our framework adopts the NCHW tensor representation convention, where N represents the batch dimension, C denotes channels, and H and W specify the height and width dimensions, respectively. The origin (index 0) is located at the top-left corner of the tensor. Partitions are defined using interval notation across these dimensions.

Each partition is represented as a collection of intervals in one or more dimensions. For example, a height partition denoted as H:[0,2)[2,4) indicates two intervals: the first spanning height indices 0 to 2, and the second covering indices 2 to 4. The left value in each interval specifies the starting index (inclusive), while the right value indicates the ending index (exclusive). When partitioning occurs in only one dimension, each interval can be directly mapped to a specific device.

3.3 Partitioning Concepts

Figure 1 illustrates two core concepts in our partitioning approach using representative examples from both Type 1 (top) and Type 2 (bottom) operators: dependencies and computation overlaps. The Type 1 example shows a convolutional operation with padding, while the Type 2 example demonstrates an element-wise operation. **Dependencies** (shown in blue in Figure 1) represent values (output activations of the preceding layer) that must be transferred from one device to another, creating synchronization points in

the execution flow. These data dependencies directly impact communication overhead and are a primary optimization target in our framework. **Computation overlaps** (highlighted in yellow in Figure 1) occur when identical computations are performed on multiple devices. These duplicated calculations represent a trade-off between computation redundancy and communication cost. For operators in Type 1, overlap can reduce communication requirements by eliminating dependencies. On the other hand, for operators in Type 2, data overlap typically only increase computational load without yielding any communication benefits.

3.4 Partition Selection Strategy

The effectiveness of a partitioning strategy depends on balancing the computational load across all devices while minimizing the communication overhead. For operators of Type 1, ADaPS calculates the optimal overlap to reduce communication costs, especially for operators with large kernel sizes. For Type 2 operators, we minimize overlaps since they provide no communication benefit. This classification-based approach allows our framework to make intelligent partitioning decisions that are tailored to the characteristics of each operator, leading to more efficient distributed execution across heterogeneous computing resources.

4 The ADaPS Framework

ADaPS introduces a novel search space exploration methodology that automatically identifies optimal partitioning strategies across heterogeneous devices. The framework considers both computational capabilities and communication constraints, enabling effective workload distribution while minimizing data transfer overhead.

During its search, ADaPS considers each device's processing speed, bandwidth constraints, and data transfer considerations. Unlike related work, ADaPS recursively explores optimal partitioning for layer fusion across the entire CNN, not just adjacent layers.

4.1 Search Space Exploration

Previous works have shown limitations in adaptability. BBGraP [12] employs naive partitioning without considering hardware specifications. EdgeFlow [7], on the other hand, restricts its search to immediate layer pairs and prohibits overlapped output partitions. ADaPS addresses these limitations through a sophisticated search approach that considers the heterogeneous hardware environment. The search space exploration engine comprises four integrated components:

1. **Path Analysis**: This component handles networks with multiple paths and shortcuts by decomposing complex graphs into manageable sub-blocks. This transforms a multi-path problem into a collection of single-path segments while maintaining partitioning consistency.
2. **Partition Generation**: Rather than exhaustively enumerating all possible combinations, this component implements a procedural system that dynamically creates specific partitioning configurations on demand. ADaPS stores only essential parameters and is able to recreate any partition in linear time.
3. **Performance Evaluation**: For each candidate partition, a performance model computes the expected execution time by considering the computational workload, communication volume, and memory requirements of the partition. This enables a quantitative comparison between different partitioning strategies.
4. **Layer Fusion Optimization**: This component searches for optimal divisions between consecutive layers while dynamically updating previously determined partitions. The algorithm can revise earlier decisions when downstream layers reveal more efficient global configurations, determining when fusion benefits outweigh the costs.

In the following, we describe each of these components in detail.

4.2 Path Analysis

Modern neural networks frequently feature complex topologies with multiple execution paths, skip connections, and merge points. This presents a significant challenge for partitioning optimization, as naively traversing the graph could lead to inconsistent partitioning decisions or inefficient search patterns. ADaPS addresses these challenges by decomposing the network graph into manageable sub-blocks before calculating the partitions.

To ensure efficient exploration of the partitioning search space, we establish the following constraints for sub-block formation:

- Each sub-block must form a directed acyclic graph (DAG) with exactly one entry point and one exit point
- A sub-block may contain layers that originally had multiple inputs or outputs in the complete network
- When encountering a layer with multiple outgoing connections, we terminate the current sub-block and initiate new sub-blocks for each output path
- When encountering a layer with multiple incoming connections, we remove previously traversed inputs from consideration and terminate the current sub-block

Figure 3 illustrates this decomposition process. Starting with the complete network graph (Figure 3(a)), the algorithm identifies layers with multiple inputs or outputs. When a layer with multiple inputs is encountered (Figure 3(b)), the algorithm terminates the current sub-block and removes the traversed connection from further consideration. This

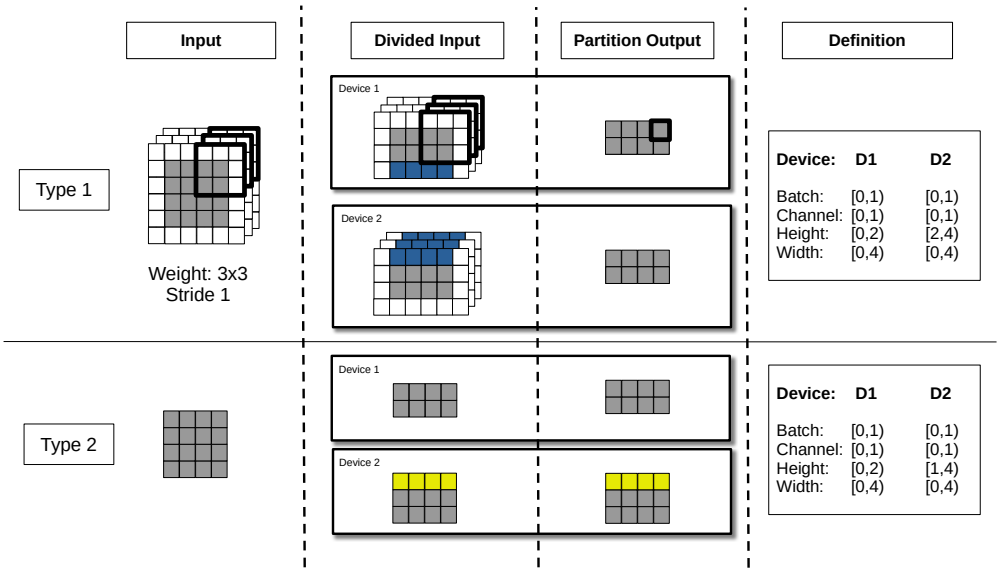


Figure 1. Illustration of partitioning effects across different operator types. Type 1 operators (top) benefit from computational overlaps by reducing data dependencies (blue regions), while Type 2 operators (bottom) experience only redundant computation (yellow regions) without communication benefits.

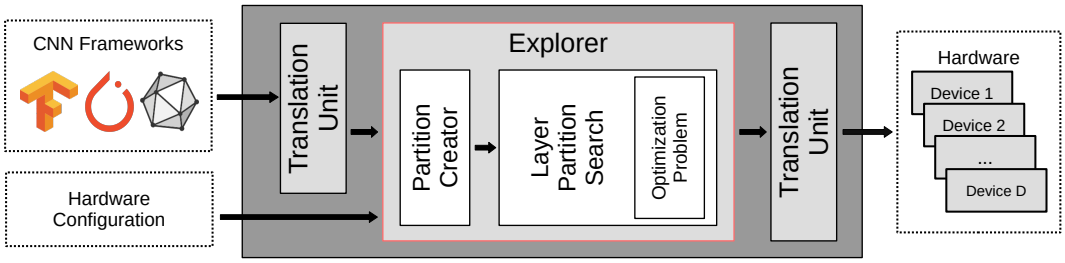


Figure 2. ADaPS architecture highlighting the search space exploration module (in red) within the optimization pipeline.

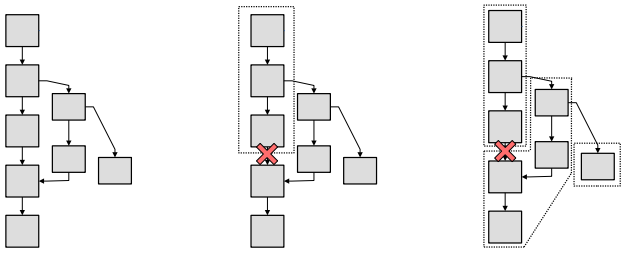
process continues until the entire network has been decomposed into a collection of single-entry, single-exit sub-blocks (Figure 3(c)).

While our decomposition approach effectively reduces the complexity of the partitioning search problem, it does not incorporate cost-based optimization for the sub-block boundaries themselves. This simplification may occasionally result in sub-optimal decomposition patterns. However, the substantial reduction in search space complexity outweighs potential inefficiencies at sub-block boundaries. It is important to note that this decomposition does not constrain layer fusion opportunities within each sub-block. The subsequent optimization phases may still identify and implement fusion across any layers within a sub-block, regardless of their position or connectivity pattern. Furthermore, while layers at sub-block boundaries are not candidates for cross-boundary fusion, their individual partitioning strategies remain fully optimizable.

4.3 Partition Creation

The search space of one layer's partitioning grows exponentially with the number of devices. If no constraints are applied, there are $O(M^D)$ possible partitionings, where M represents the output feature map dimension size and D is the number of deployed devices. Layers are independent, yielding a search space complexity of $O(M_l^D)\ \forall l \in Layers$. Exhaustively evaluating all possible partitions is impossible; therefore, we employ an adaptive thresholding approach to constrain the search space while preserving the exploration of the most promising partitioning configurations.

4.3.1 Bound Determination for Partition Sizes. To effectively prune the search space, we first establish minimum and maximum partition sizes for each device. The minimum partition size for device d is calculated using Equation 1, where out_size represents the output feature map dimension to be partitioned, and $speed_d$ denotes the processing



(a) Initial network graph (b) Multiple-input layer identification (c) Final sub-block decomposition

Figure 3. Path analysis process for network decomposition. Dotted rectangles represent identified sub-blocks, and crossed connections indicate removed paths during the decomposition process.

capability of device d relative to all devices in the system.

$$min\_size_d = \lfloor out\_size \cdot \frac{speed_d}{\sum_{i=1}^{D} speed_i} \rfloor \quad (1)$$

The processing speed metrics ($speed_d$) are derived from empirical performance measurements, following methodologies established in EdgeFlow [7] and DeepSlicing [24]. We develop a linear regression model based on benchmarking data collected using PyTorch's [15] profiling tools. This model provides a reliable approximation of relative processing speeds across heterogeneous devices.

The minimum size constraint ensures a computational load distribution proportional to each device's processing capability in an ideal scenario that ignores communication overhead. Partition sizes smaller than this minimum threshold would likely create performance imbalances, resulting in suboptimal execution as detailed in the optimization formulation in Section 4.4.

The maximum size of a partition is identical for all devices and ensures that no single device monopolizes the entire workload.

$$max\_size_d = out\_size - (D - 1) \quad (2)$$

For example, in a scenario with output size 224 and four devices, the first device cannot process the entire range from 0 to 224, as this would leave no meaningful work for the remaining devices. Similarly, the last device cannot start its partition at index 0 for the same reason.

4.3.2 Search Space Pruning with Adaptive Thresholding.

After the minimum and maximum bounds are established, we implement adaptive thresholding according to Equation (3):

$$search\_size_d = (max\_size_d - min\_size_d) \cdot thrs \quad (3)$$

Where $thrs$ represents a tunable threshold parameter between 0 and 1 that controls the aggressiveness of search space pruning. This approach allows us to restrict the search on the most promising region between the minimum size (based on computational balance) and maximum size (constrained by work distribution requirements).

The threshold mechanism provides a configurable trade-off between exploration thoroughness and computational efficiency. Higher threshold values explore a larger portion of the partition size range, potentially discovering more optimal configurations at the cost of increased search time. Lower threshold values concentrate on partitions closer to the computational balance point, accelerating the search process while potentially missing some optimization opportunities. An evaluation of the threshold's effect is provided in the result section (Table 6).

Importantly, this thresholding approach preserves the potential for layer fusion by maintaining a reasonable range of partition sizes rather than forcing a single fixed partitioning strategy. Exploring diverse partitioning configurations remains essential for identifying fusion opportunities across multiple layers.

4.4 Optimization Problem

This subsection describes our methodology for identifying optimal partitioning strategies through a formalized optimization problem. The approach incorporates three key factors: computational workload distribution, inter-device data transfer, and management of overlapped computational regions. The following notations are used during this subsection:

- Indices s and e denote the start and end of partition intervals (e.g., for interval $[3, 7)$, $s = 3$ and $e = 7$)
- $p_{l,d,s/e}$: l layer index, d device, s start/e end of partition.

4.4.1 Computational Workload Estimation.

The computational workload distribution across heterogeneous devices is calculated by Equation (4) by computing a vector of estimated execution times $\mathbf{t}$ for each device d, based on the assigned output size and device-specific processing capabilities. We utilize the same $speed_d$ values as introduced in Section 4.3.

$$t_d = \lfloor out\_size / speed_d \rfloor \quad \forall d \in [1, D] \quad (4)$$

Equation (5) identifies the computational bottleneck by determining t_{comp}, the maximum execution time among all devices. This represents the slowest device that others must wait for to maintain synchronization.

$$t_{comp} = \max(\mathbf{t}) \quad (5)$$

Example 4.1. Consider two partitioning strategies: $p_{l,1} = [0, 3)[3, 6)[6, 9)$ and $p_{l,2} = [0, 2)[2, 3)[3, 9)$, where all devices have equal processing capabilities. In $p_{l,2}$, two devices ($device_1$ and $device_2$) complete significantly earlier but remain idle while waiting for $device_2$. Conversely, in $p_{l,1}$, the

slowest device finishes earlier than in $p_{l,2}$. Our model penalizes $p_{l,2}$ to avoid excessive idle time across devices.

4.4.2 Data Transfer Overhead. After determining the computational costs, we calculate the data transfer overhead between adjacent layers.

Equations (6) and (7) calculate the data transfer requirements at the start and the end of each partition interval, applying constraints to prevent illogical negative transfer values.

$$data_{l,d,s} = \begin{cases} 0 & \text{if} \quad p_{l+1,d,s} > p_{l,d,s} \\ \min(p_{l,d,s}, p_{l+1,d,e}) - p_{l+1,d,s} & \text{o/w} \end{cases} \tag{6}$$

$$data_{l,d,e} = \begin{cases} 0 & \text{if} \quad p_{l+1,d,e} < p_{l,d,e} \\ p_{l+1,d,e} - \max(p_{l+1,d,s}, p_{l,d,e}) & \text{o/w} \end{cases} \tag{7}$$

Example 4.2. For this example, we consider two layers division partitions at $d = 1$: where $p_{l,1} = [3, 6)$ and $p_{l+1,1} = [2, 3)$, the application of Equations (6) and (7) yields $data_{l,1,s} = 1$ and $data_{l,1,e} = 0$.

The total data transfer time (t_{data}) is calculated by summing individual transfer requirements and dividing by the bandwidth capacity (BW_n) for each device:

$$t_{data} = \sum_{d=1}^{D} (data_{l,d,s} + data_{l,d,e})/BW_d \tag{8}$$

4.4.3 Overlapped Computation Overhead. As explained in Section 3, certain operators require duplicate computation across multiple devices when partitioned. Equation (10) calculates this overlapped computation overhead (t_{ov}), accounting for processing speed differentials across devices.

$$r_{ov} = \begin{cases} p_{l,d,e} - p_{l,d+1,s} & d = 1 \\ p_{l,d,s} - p_{l,d-1,e} & d = D \\ p_{l,d,s} - p_{l,d-1,e} + p_{l,d,e} - p_{l,d+1,s} & 1 < d < D \end{cases} \tag{9}$$

$$t_{ov} = \sum_{d=1}^{D} \frac{r_{ov}}{speed_d} \tag{10}$$

Note in Equation (9), that for the first and the device ($d = 1$ and $d = D$, respectively), only one of the sides of the partitions are considered, as these devices are not positioned between two other devices.

4.4.4 Total Execution Time. The total execution time ($score$) for a candidate partitioning strategy is given by the sum of the components:

$$score = t_{comp} + t_{data} + t_{ov} \tag{11}$$

It is important to note that our current model does not consider potential overlap between computation and data transfer operations. This simplification allows us to directly sum the three time components to obtain the total execution time for each layer. While more sophisticated pipelining techniques could theoretically improve performance further and are left for future work, this additive model provides a robust approximation for comparing candidate partitioning strategies, with lower scores indicating superior configurations.

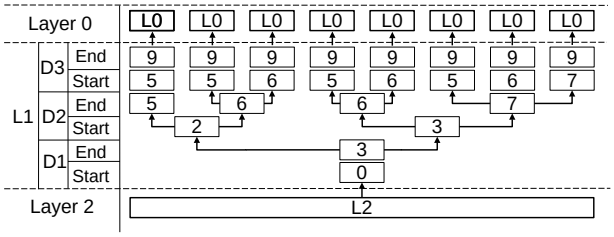


Figure 4. Search space exploration example. Tree nodes prefixed with 'L' represent layer structures, 'D' indicates device index, and Start/End denote partition interval boundaries.

The calculated $score$ serves as the primary metric during partition selection in Section 4.5, guiding the identification of optimal partitioning strategies for each layer in the CNN architecture.

4.5 Search Space Exploration

This section describes our approach to efficiently exploring the vast partitioning search space while identifying near-optimal solutions. We present a tree-based search algorithm enhanced with pruning techniques and dynamic programming to significantly reduce computational complexity.

4.5.1 Tree-Based Representation. We employ a hierarchical tree structure to represent the partitioning search space, enabling the application of Alpha-Beta pruning techniques. Figure 4 illustrates this representation, where each layer node encodes a specific division of the layer's tensor along the selected dimension. Each layer node contains three essential components:

- **Layer:** Represents the CNN layer being partitioned, with boundaries from 0 to the layer's dimension size in the selected direction.
- **Device:** Indicates the specific device assignments for a partition segment. For example, in Figure 4, device 1 the left-most assigned partition [2, 5).
- **Partition Interval:** Defines the start and end indices of each partition segment within the tensor dimension.

The organization of tree nodes within the tree significantly impacts search efficiency. To leverage the convex nature of our optimization function, we order tree nodes to represent monotonically increasing partition sizes when traversing from left to right. For example, in Figure 4, partitions progress from [0, 3) to [0, 4), incrementing the partition size. We apply a similar ordering to starting points within each device. This structured ordering enables more effective pruning during the search.

4.5.2 Evaluation Methodology.

Our algorithm employs a depth-first search to reach terminal nodes, followed by backpropagation to evaluate partitioning decisions. The evaluation of layer partitioning configurations is governed by Equation (12), which addresses both current layer optimization and interactions with adjacent layers.

$$r_l = \begin{cases} opt(div_{0,out}, div_{1,in}) & l = 0 \\ opt(div_{l,out}, \emptyset) + r_{l-1} & l = L-1 \\ opt(div_{l,out}, div_{l+1,in}) + r_{l-1} & 0 < l < L-1 \end{cases} \quad (12)$$

Equation (12) comprises two key components. The first component, opt, evaluates the relationship between the input division, $div_{l+1,in}$, (derived from the previous layer's output) and the current layer's output division, $div_{l,out}$. In Figure 4, this corresponds to analyzing partitions from Layers 1 and 2 using the optimization formulation from Section 4.4. If no previous layer exists ($l = L-1$), the data transfer component from Equation (8) is omitted.

The second component r considers the impact of the current partitioning decision on subsequent layers. For example, when evaluating Layer 1, we incorporate results from Layer 0. This recursive formulation enables the discovery of layer fusion opportunities across multiple layers, not just adjacent pairs.

The evaluation results for each unique partitioning configuration are stored to prevent redundant computation, significantly reducing the overall search time.

4.5.3 Dynamic Programming Optimizations.

To accelerate search space exploration, we implement two dynamic programming techniques:

Memoization of Evaluation Results: As noted in Section 4.5.2, identical partitioning configurations may be encountered multiple times during the search. We cache evaluation results for each unique configuration, retrieving them when the same configuration is encountered again rather than recalculating.

Sub-Tree Reuse: This technique identifies and reuses pruned branches across different parts of the search tree when all preceding node values match. Figure 5 illustrates this concept, where branches sharing the same previous starting node values can reuse identical sub-trees. This approach preserves the convexity properties established by our node ordering, ensuring consistent evaluation patterns while eliminating redundant exploration.

4.5.4 Alpha-Beta Pruning Implementation.

The efficiency of our search algorithm is substantially enhanced through a modified Alpha-Beta pruning approach. By leveraging the sorted nature of our search tree (Section 4.5.1), we can eliminate entire branches that cannot improve upon the best solution found so far.

As illustrated in Figure 6, after evaluating several partitioning possibilities, our algorithm identifies and prunes branches that cannot yield better results, significantly reducing search time. The pruning propagates up through previous branches to the layer's root node, maximizing the elimination of unnecessary computation.

It is important to note that when evaluating multiple layers with Equation (12), this pruning approach might occasionally exclude potentially superior solutions in specific edge cases. However, this represents a necessary trade-off to make the vast search space computationally feasible. In practice, our experimental results demonstrate that the algorithm consistently finds high-quality solutions despite this theoretical limitation.

5 Experimental Evaluation

5.1 Experimental Setup

To rigorously evaluate ADaPS, we conducted experiments on resource-constrained edge devices using the hardware configurations detailed in Table 3. Our implementation leverages PyTorch [15] and its Gloo backend for distributed communication primitives. For comparing inference latency across frameworks, we used PyTorch's built-in profiling tools to ensure accurate timing measurements.

We selected a diverse set of CNN architectures to thoroughly evaluate our framework: AlexNet [9], VGG16 [19], MobileNetV2 [18], InceptionV3 [21], and HardcoreNAS [14]. These networks were chosen to represent a spectrum of complexity, from relatively simple architectures to those with challenging multi-path structures. All network models were obtained from ONNX Zoo [1].

To explore different network bandwidths, we test each configuration at 10 Mbps and 100 Mbps. This allowed us to evaluate the frameworks' sensitivity to communication constraints. We compared ADaPS against two state-of-the-art decentralized frameworks: BBGraP [12] and EdgeFlow [7]. All reported results are normalized to EdgeFlow's performance for consistent comparison. The partitioning strategies were computed once per network and hardware configuration. The threshold was set such that the total runtime of ADaPS' search algorithm stayed below five minutes (15% for all networks except VGG with a threshold of 5 percent) on single core of a AMD Ryzen 7 7700X processor running at 5.4 GHz. All reported performance results represent averages across 100 independent runs on real hardware.

5.2 Performance in Heterogeneous Environments

The most significant advantage of ADaPS is its ability to efficiently distribute workloads across heterogeneous devices. Figure 7 shows the performance comparison for a two-node heterogeneous configuration (one Raspberry Pi 5B and one Raspberry Pi 4B). ADaPS outperforms both BBGraP

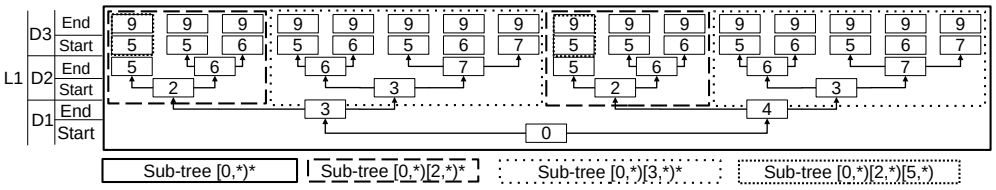


Figure 5. Visual representation of sub-tree reuse in dynamic programming optimization. Identical sub-trees (highlighted) are computed once and reused across branches with matching preceding conditions.

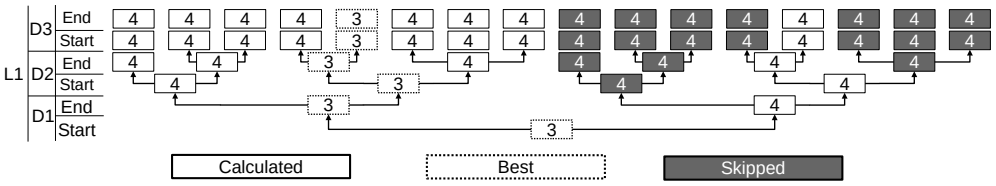


Figure 6. Visual representation of Alpha-Beta branch pruning. Values inside nodes represent optimization function results, with shaded nodes indicating pruned search paths that cannot improve upon the current best solution.

Table 3. Hardware Configurations for Experimental Evaluation

Platform	Devices	Processor	Cores/Device	Memory
Raspberry Pi 4B	2	Cortex-A72 (1.5GHz)	8	8 GiB
Raspberry Pi 5B	4	Cortex-A76 (2.4GHz)	8	8 GiB

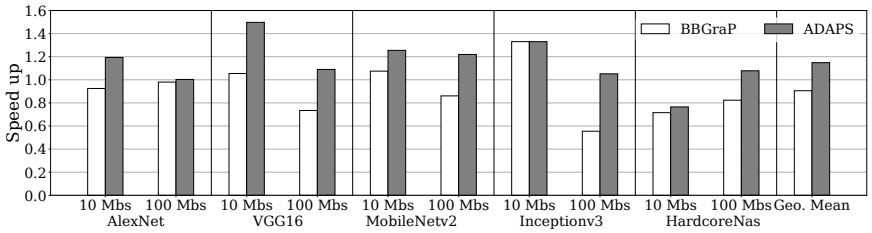


Figure 7. Two-node heterogeneous configuration performance comparison (RP 5B × 1, RP 4B × 1). Results normalized to EdgeFlow (lower is better).

and EdgeFlow across most network architectures and bandwidth configurations, with an average improvement of approximately 30% for networks without complex multi-path structures. This performance advantage stems from ADaPS's ability to adapt to device capabilities and optimize partitioning decisions accordingly. For simpler network architectures like AlexNet and VGG16, ADaPS achieves particularly impressive gains, with performance improvements of up to 54% compared to EdgeFlow at higher bandwidth settings. The improvements are more pronounced at 100 Mbps than at 10 Mbps for most networks, indicating that ADaPS effectively leverages available bandwidth to optimize data transfer patterns.

For more complex architectures with multiple execution paths (InceptionV3 and HardcoreNAS), the performance improvements are more modest but still significant. In these cases, the challenge of coordinating multiple execution paths while maintaining efficient data transfer becomes more pronounced, particularly at lower bandwidth settings. Nevertheless, ADaPS still matches or exceeds the performance of BBGraP for these networks and significantly outperforms EdgeFlow.

5.3 Data Transfer Efficiency

A critical factor in ADaPS's performance advantage is its ability to minimize inter-device data transfer through intelligent partitioning decisions. Table 4 presents the data reduction comparison between ADaPS (`AD`), BBGraP (`BB`), and EdgeFlow for the two-node heterogeneous configuration (one Raspberry Pi 5B, one Raspberry Pi 4B device). Lower values indicate less communication overhead relative to EdgeFlow.

Table 4. Comparison of transferred data volume for a two-node heterogeneous setup (1x RP 5B, 1x RB 4B),normalized to EdgeFlow. BB denotes BBGraP, AP represents ADaPS.

Network	Bandwidth (Mbps)	Data Reduction Factor	
		BB	AD
AlexNet	10	1.45	**0.73**
	100	1.45	**0.96**
VGG16	10	1.08	**0.46**
	100	1.08	**0.61**
MobileNetV2	10	**0.50**	0.52
	100	**0.51**	0.53
InceptionV3	10	2.18	**0.97**
	100	2.17	**0.98**
HardcoreNAS	10	1.71	**0.94**
	100	1.71	**0.95**

ADaPS achieves the lowest communication overhead for most networks, with particularly impressive reductions for VGG16 (54-39% reduction compared to EdgeFlow). The strong correlation between data reduction and inference speedup confirms that ADaPS's superior partitioning decisions effectively minimize the communication bottleneck. For MobileNetV2, BBGraP achieves slightly better data reduction, but the difference is marginal (0.50 vs. 0.52), explaining the comparable performance between the two frameworks for this specific network.

5.4 Scaling to Four-Node Configurations

To evaluate ADaPS's scalability, we extended our experiments to a four-node heterogeneous configuration comprising two Raspberry Pi 5B and two Raspberry Pi 4B devices. As shown in Figure 8, ADaPS maintains its performance advantage at this larger scale, though with somewhat reduced margins compared to the two-node configuration.

With four nodes, ADaPS achieves an average improvement of approximately 15% compared to EdgeFlow and BBGraP. Interestingly, the performance pattern with respect to bandwidth changes: in the four-node configuration, lower network speeds (10 Mbps) often yield better relative performance than higher speeds (100 Mbps). This result stems from how increased bandwidth affects the optimization landscape. At higher bandwidths, the algorithm creates more synchronization points, which can lead to inefficiencies when scaled to more devices.

Table 5 shows the data reduction comparison for the four-node configuration. ADaPS again achieves the lowest communication overhead for most networks, though the margins are smaller compared to the two-node case. VGG16 and AlexNet show the most significant data reduction (19% and 28% respectively at their best settings), which directly correlates with their superior inference performance. This confirms that ADaPS's optimization strategy remains effective even at larger scales.

Table 5. Comparison of transferred data volume for a four-node heterogeneous setup (2x RP 5B, 2x RB 4B) of BBGraP (BB) and ADaPS (AP) relative to EdgeFlow.

Network	Bandwidth (Mbps)	Data Reduction Factor	
		BB	AD
AlexNet	10	1.05	**0.88**
	100	0.86	**0.72**
VGG16	10	0.98	**0.81**
	100	0.98	**0.81**
MobileNetV2	10	**0.92**	0.93
	100	**0.71**	0.72
InceptionV3	10	1.22	**1.09**
	100	1.29	**1.10**
HardcoreNAS	10	1.08	**1.02**
	100	1.04	**0.98**

5.5 Impact of Network Architecture

Our results reveal important insights about how network architecture affects partitioning performance:

- **Linear architectures** (AlexNet, VGG16) benefit most significantly from ADaPS's optimization approach, with consistent performance improvements across all configurations.
- **Efficient architectures** (MobileNetV2) show more modest gains, likely because they already minimize computational and memory requirements, leaving less room for optimization through partitioning.
- **Complex multi-path architectures** (InceptionV3, HardcoreNAS) present greater challenges for all partitioning frameworks, particularly at lower bandwidths. Nevertheless, ADaPS still outperforms competing approaches for these networks in most configurations.

5.6 Pruning results

The search space for assigning partitions of a neural network layer to D heterogeneous devices is combinatorially large and cannot be exhaustively explored; heuristics are therefore essential. The presented ADaPS technique first estimates a partitioning based on device performance, followed by aggressive pruning. The initial partitioning provides a strong initial guess. A threshold parameter thr defines the search range around this estimate: given an initial guess p, the search spans $[p - thr \cdot p, p + thr \cdot p]$. While the threshold parameter allows for tuning the trade-off between search quality and runtime, the resulting space remains exponentially large.

Pruning further reduces this space to a manageable size. Table 6 quantifies the impact of pruning on the search space for a single convolutional layer of size 224 and a varying

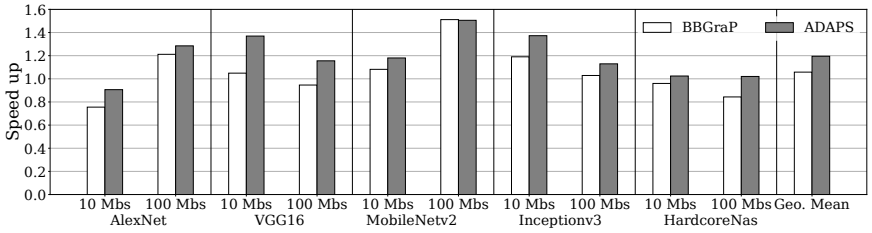


Figure 8. Performance comparison with four-node heterogeneous configurations (2x Raspberry Pi 5B, 2x Raspberry Pi 4b). Results normalized to EdgeFlow (lower is better).

Table 6. Effect of pruning threshold on search space exploration for a single convolutional layer of size 224.

# Devices		Threshold			
		5%	10%	15%	20%
2	Search space (log10)	2.5	3.1	3.4	3.6
	Visited	26	48	70	92
	Pruning factor (log10)	1.1	1.4	1.5	1.7
4	Search space (log10)	5.7	7.4	8.2	9
	Visited	389	1355	2600	4622
	Pruning factor (log10)	3.1	4.2	4.8	5.3
8	Search space(log10)	9.8	13.3	15.5	17.2
	Visited	966	3417	7344	12747
	Pruning factor (log10)	6.8	9.8	11.7	13.1
16	Search space (log10)	17.1	21.1	26.6	28.7
	Visited	2501	5174	16528	43121
	Pruning factor (log10)	13.7	17.4	22.4	24.1

numbers of devices. Pruning becomes increasingly effective with a larger threshold and more devices, as it eliminates infeasible candidates early in the search tree. As shown in Section 4.5.4, ADaPS halts exploration of subtrees that cannot yield better solutions, exploiting the tendency of high-quality solutions to occur early. The pruning benefit multiplies across layers, scaling exponentially with the network depth.

Although ADaPS does not guarantee globally optimal solutions, it consistently identifies high-quality configurations within exponential search spaces in reasonable time. The experimental results in this section confirm that ADaPS outperforms the state-of-the-art baselines across diverse architectures and hardware setups.

6 Conclusion

We present ADaPS, a framework for efficient CNN inference on resource-constrained heterogeneous hardware. Our approach combines game-theoretic techniques (Minmax and Alpha-beta pruning) with dynamic programming optimizations to comprehensively explore partitioning strategies across the entire network architecture. Unlike previous methods that optimize layers individually or in adjacent pairs, ADaPS performs a holistic analysis that enables discovery of effective layer fusion opportunities while minimizing both computational imbalance and communication overhead. The experimental evaluation demonstrates that ADaPS consistently outperforms state-of-the-art approaches, improving inference latency by up to 30% for some architectures and maintaining an average improvement of 1.2× across all tested networks. These gains are accompanied by significant reductions in inter-device data transfer of up to 54% compared to existing methods. ADaPS's ability to automatically tailor partitioning decisions to specific hardware capabilities proves particularly valuable as network complexity increases.

Acknowledgments

We thank the anonymous reviewers for their helpful feedback. This work was funded, in parts, by the Neural Processing Research Center (NPRC) of the Samsung Advanced Institute of Technology (SAIT), and by the Korean National Research Foundation through grants 21A20151113068 (BK21 Plus for Pioneers in Innovative Computing - Dept. of Computer Science & Engineering, SNU), 10077609 (MOTIE/KEIT), and RS-2023-00302083 (MSIT). ICT at Seoul National University provided research facilities for this study. Bernhard Egger is the corresponding author for this work.

References

[1] Junjie Bai, Fang Lu, Ke Zhang, et al. 2019. ONNX: Open Neural Network Exchange. https://github.com/onnx/onnx.

[2] Ahmed A Elngar, Mohamed Arafa, Amar Fathy, Basma Moustafa, Omar Mahmoud, Mohamed Shaban, and Nehal Fawzy. 2021. Image classification based on CNN: a survey. *Journal of Cybersecurity and Information Management* 6, 1 (2021), 18–50.

[3] Eyad Elyan, Pattaramon Vuttipittayamongkol, Pamela Johnston, Kyle Martin, Kyle McPherson, Chrisina Jayne, Mostafa Kamal Sarker, et al. 2022. Computer vision and machine learning for medical image analysis: recent advances, challenges, and way forward. *Artificial Intelligence Surgery* 2 (2022).

[4] Myeonggyun Han, Jihoon Hyun, Seongbeom Park, Jinsu Park, and Woongki Baek. 2019. MOSAIC: Heterogeneity-, Communication-, and Constraint-Aware Model Slicing and Execution for Accurate and Efficient Inference. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 165–177. doi:10.1109/PACT.2019.00021

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[6] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[7] Chenghao Hu and Baochun Li. 2022. Distributed inference with deep learning models across heterogeneous edge devices. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 330–339.

[8] Donald E. Knuth and Ronald W. Moore. 1975. An analysis of alpha-beta pruning. *Artificial Intelligence* 6, 4 (1975), 293–326. doi:10.1016/0004-3702(75)90019-3

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger (Eds.), Vol. 25. Curran Associates, Inc.

[10] Nicholas D Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. 2016. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. IEEE, 1–12.

[11] Jiachen Mao, Xiang Chen, Kent W Nixon, Christopher Krieger, and Yiran Chen. 2017. Modnn: Local distributed mobile computing system for deep neural network. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 1396–1401.

[12] Jaume Mateu Cuadrat, Daon Park, and Bernhard Egger. 2022. A Black-Box Graph Partitioner for Generalized Deep Neural Network Parallelization. In *International Conference on the Economics of Grids, Clouds, Systems, and Services*. Springer, 132–140.

[13] Svetlana Minakova, Erqian Tang, and Todor Stefanov. 2020. Combining task-and data-level parallelism for high-throughput CNN inference on embedded CPUs-GPUs MPSoCs. In *Embedded Computer Systems: Architectures, Modeling, and Simulation: 20th International Conference, SAMOS 2020, Samos, Greece, July 5–9, 2020, Proceedings 20*. Springer, 18–35.

[14] Niv Nayman, Yonathan Aflalo, Asaf Noy, and Lihi Zelnik-Manor. 2021. HardCoRe-NAS: Hard Constrained diffeRentiable Neural Architecture Search. arXiv:2102.11646 [cs.LG]

[15] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[16] Mehdi Rafiei, Jenni Raitoharju, and Alexandros Iosifidis. 2023. Computer vision on x-ray data in industrial production and security applications: A comprehensive survey. *Ieee Access* 11 (2023), 2445–2477.

[17] Ronald L Rivest. 1987. Game tree searching by min/max approximation. *Artificial Intelligence* 34, 1 (1987), 77–96.

[18] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2019. MobileNetV2: Inverted Residuals and Linear Bottlenecks. arXiv:1801.04381 [cs.CV]

[19] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[20] Rafael Stahl, Alexander Hoffman, Daniel Mueller-Gritschneder, Andreas Gerstlauer, and Ulf Schlichtmann. 2021. DeeperThings: Fully distributed CNN inference on resource-constrained edge devices. *International Journal of Parallel Programming* 49 (2021), 600–624.

[21] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander Alemi. 2017. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 31.

[22] K Vanishree, Anu George, Srivatsav Gunisetty, Srinivasan Subramanian, Shravan Kashyap, and Madhura Purnaprajna. 2020. CoIn: Accelerated CNN Co-Inference through data partitioning on heterogeneous devices. In *2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS)*. IEEE, 90–95.

[23] Liekang Zeng, Xu Chen, Zhi Zhou, Lei Yang, and Junshan Zhang. 2020. Coedge: Cooperative dnn inference with adaptive workload partitioning over heterogeneous edge devices. *IEEE/ACM Transactions on Networking* 29, 2 (2020), 595–608.

[24] Shuai Zhang, Sheng Zhang, Zhuzhong Qian, Jie Wu, Yibo Jin, and Sanglu Lu. 2021. Deepslicing: collaborative and adaptive cnn inference with low latency. *IEEE Transactions on Parallel and Distributed Systems* 32, 9 (2021), 2175–2187.

[25] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. 2018. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2348–2359.

[26] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. 2018. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 8697–8710.

Received 2025-03-21; accepted 2025-04-21