

nnWeb: Towards efficient WebGPU-based DNN inference via automatic collaborative offloading

Jiawei Liu^a, Bing Dong^a, Weilong Wang^a, Muhan Yuan^a, Borui Li^a*, Zhao-Dong Xu^b, Shuai Wang^a

^a School of Computer Science and Engineering, Southeast University, Nanjing, 211189, China

^b School of Civil Engineering, Southeast University, Nanjing, 211189, China

ARTICLE INFO

Keywords:

Model partitioning
WebGPU
Mobile computing
Cloud computing

ABSTRACT

In-browser neural network inference offers the promise of cross-platform AI applications, but faces severe latency and energy challenges on resource-constrained devices. In this paper, we present nnWeb, a WebGPU-based in-browser neural network inference framework with optimized latency and energy efficiency. nnWeb dynamically partitions neural network and facilitates the collaborative offloading between client browser and server. nnWeb operates in two phases: (1) *layer-wise isolation-based profiling*, which is used to predict per-layer execution latency and energy on heterogeneous hardware; and (2) *asynchronous execution-based DNN partitioning*, which continuously monitors network bandwidth and device load to select the optimal partition point using WebGPU's native pipeline parallelism, minimizing total latency or energy consumption by solving a closed-form optimization at runtime. Extensive evaluation on various in-browser AI models and networking conditions shows that nnWeb achieves an average reduction of 30% to 52% in total inference latency compared with static partitioning. Moreover, nnWeb realizes energy savings ranging from 11.3% to 44.0% in contrast to standalone browser inference.

1. Introduction

With the rapid growth in both the scale of deep learning models and the diversity of deployment scenarios, in-browser deployment of neural networks [1,2] is becoming increasingly prevalent. In-browser inference enables AI models to run on any device equipped with a modern web browser, ranging from desktops to tablets, smartphones, and even lightweight embedded platforms. This inference paradigm supports applications such as intelligent search [3], recommendation systems [4], and automatic translation [5] across heterogeneous hardware and operating systems.

Due to the interactive nature of the browser, its responsiveness and energy consumption are critical to end-user Quality of Experience (QoE). This importance leads to an increased attention from both industry and academia [6,7]. This problem is further exacerbated as the model sizes and computation demands increase continuously. While high-performance servers can execute Deep Neural Network (DNN) models efficiently, they typically incur high energy consumption [8]. In contrast, deploying the same models on client devices with limited computational and memory resources often results in significant latency [9].

To mitigate these limitations, various in-browser inference acceleration techniques have been explored. These methods aim to improve both responsiveness and energy efficiency by applying model compression strategies such as pruning [10], quantization [11], and operator fusion, together with the use of specialized hardware like GPUs [12]. However, in-browser inference must be implemented through cross-platform graphics APIs, such as WebGL [13], which limits the applicability of the existing optimizations. Moreover, existing in-browser inference frameworks [1,2,14–16] focus mainly on optimizing the inference on independent device. These works neglect the opportunity to take advantage of the Web for collaborative computing.

Fortunately, the proliferation of WebGPU [17] offers new opportunities for efficient and fast in-browser inference. As the latest W3C standard, WebGPU exposes a lower-level, more efficient GPU interface than WebGL. By providing native resource binding and finer-grained pipeline control, WebGPU reduces runtime overhead and enhances both memory throughput and parallelism, making it well suited for high-performance scientific computing and deep learning inference directly on the browser. Moreover, inherent network connectivity of browsers opens up the possibility of collaborative inference, whereby

* Corresponding author.

E-mail address: libr@seu.edu.cn (B. Li).

<https://doi.org/10.1016/j.comnet.2025.111489>

Received 8 May 2025; Received in revised form 12 June 2025; Accepted 18 June 2025

Available online 1 July 2025

1389-1286/© 2025 Elsevier B.V. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

a single DNN can be partitioned and executed across multiple browser instances [17,18].

However, optimizing inference with WebGPU presents two fundamental challenges. First, WebGPU significantly accelerates model inference, reducing execution time from the order of seconds to milliseconds (see Section 3.2). A key challenge is how to precisely profile WebGPU-based in-browser AI inference at the millisecond level. Second, WebGPU adopts an asynchronous execution paradigm that decouples model loading from inference [19]. How to fully exploit this paradigm to further reduce latency and energy consumption during collaborative inference remains an open research problem.

To tackle these challenges, we present nnWeb, an adaptive, layer-wise model partitioning system that dynamically offloads DNN computations between client and server. nnWeb operates in two key phases: (1) *Layer-wise isolation-based profiling*. We build predictive performance profiling models for each DNN layer type (e.g., convolution, pooling, fully connected) across heterogeneous devices. This comprehensive profiling enables nnWeb to capture the performance characteristics of different layers under diverse conditions, thus providing a basis for the client and server to make the browser's fine-grained partition decision at the same time in the subsequent stages. (2) *Asynchronous execution-based DNN partitioning*. Leveraging these predictive models together with real-time measurements such as network bandwidth and device workload, nnWeb determines the optimal partition points within the DNN model. Once the partition point is selected, both the client and the server are informed, allowing the server to proactively prepare its portion of the model by utilizing WebGPU's asynchronous execution APIs. When the client transfers intermediate activations, the server can immediately proceed with inference, effectively overlapping initialization and computation in order to minimize end-to-end latency and energy consumption.

We implement nnWeb and conduct extensive evaluation with prevalent AI models and various network conditions. Results show that nnWeb achieves an average reduction of 30%–52% in total inference latency compared with static partitioning. Moreover, nnWeb realizes energy savings ranging from 11.3% to 44.0% in contrast to standalone browser inference. The key contributions of our work are as follows:

- We propose nnWeb, an adaptive model partitioning system that dynamically offloads DNN computations between client browser and server browser to minimize total latency or energy consumption.
- We design a performance modeling approach that accurately predicts the execution time and resource requirements of different layers on various devices.
- We validate the effectiveness of nnWeb across multiple network conditions and hardware configurations, demonstrating its broad applicability in diverse application domains, including computer vision, natural language processing, and speech recognition.

The remainder of this paper is organized as follows. Section 2 provides the background and motivation for our work, discussing the trade-offs between browser-based and server-based inference and the layer-level characteristics of neural network models. Section 3 presents the design of nnWeb, including the offline profiling phase and the online optimization system. Section 4 evaluates the performance of nnWeb across various network conditions, hardware configurations, and application domains, demonstrating its effectiveness in reducing total latency and optimizing energy consumption. Section 5 discusses the limitations of nnWeb and potential directions for future research. Finally, Section 6 reviews related works in the field, and Section 7 concludes the paper.

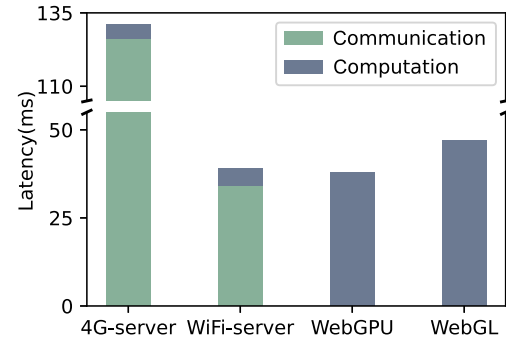


Fig. 1. The comparison of model inference based on different configurations.

2. Background and motivation

Intelligent web applications increasingly rely on in-browser neural network inference to deliver low-latency, privacy-preserving AI services across heterogeneous devices. However, running complex DNNs entirely on either the client or the server alone entails significant trade-offs: server-side inference incurs communication delays, while client-side inference is constrained by limited compute and memory. In this section, we first examine the fundamental latency and resource trade-offs of collaborative offloading between browsers and servers, and then derive layer-wise insights that motivate a dynamic partitioning strategy.

2.1. Client-server trade-offs in collaborative DNN offloading

Neural network inference is traditionally performed on cloud servers [8], leveraging powerful hardware but suffering from high transmission latency, especially problematic for real-time applications. In contrast, running inference directly in client browsers reduces communication delays but is constrained by limited local resources, often resulting in unacceptable latency. To balance these trade-offs, collaborative offloading has emerged as a promising solution. It leverages the strengths of both client-side and server-side execution by dynamically partitioning DNN models, allowing the client browser to handle early layers while the server processes the deeper stages of inference.

When evaluating the client-server browser inference trade-off, multiple dimensions warrant consideration. For the purpose of illustration in this section, we specifically use latency as an example to explain the impact on application performance. It should be noted that this represents just one aspect of a comprehensive evaluation that should also incorporate factors such as energy efficiency, computational resource allocation, and cost-effectiveness.

To quantitatively analyze these trade-offs, we conduct performance evaluations using MobilenetV1 [20], a lightweight convolutional neural network for image classification. Our experimental setup comprised:

- Client device: Laptop with integrated GPU (Intel UHD Graphics 730)
- Server device: High-performance machine with discrete GPU (NVIDIA RTX 3060)

As illustrated in Fig. 1, our results demonstrate that inference based on server browser exhibited end-to-end latencies of 131 ms (4G) and 39 ms (WiFi). Notably, while server computation latency measured only 5 ms, transmission latency dominated the total delay, particularly in 4G environments. In contrast, the client browser based implementations using WebGPU and WebGL achieved latencies of 38 ms and 47 ms respectively, demonstrating that while client inference eliminates communication overhead, it incurs higher computational latency compared to server browser based inference.

These findings highlight the fundamental trade-off between communication and computation latency in neural network inference:

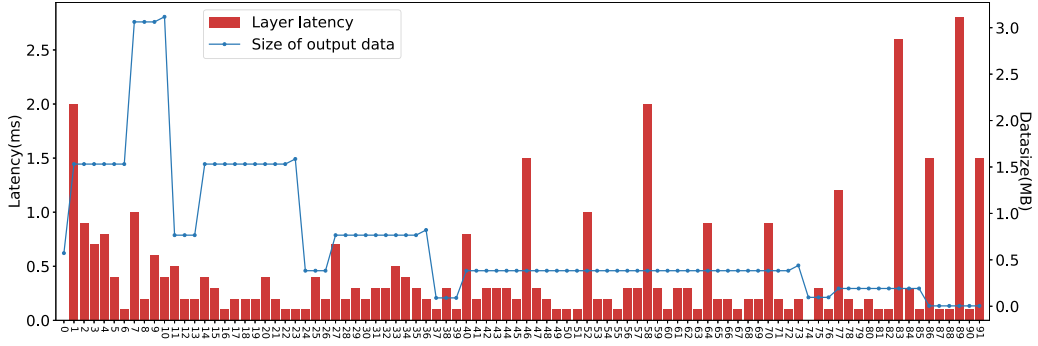


Fig. 2. Per-layer inference time (red bars) and output data size (blue markers) after processing each layer in MobilenetV1. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

- Server browser: Superior computational capability but constrained by transmission delays.
- Client browser: Eliminates transmission costs but limited by local resource constraints.

This dichotomy motivates our investigation of a hybrid inference framework that strategically combines the advantages of both approaches, which will be introduced in §3. By implementing dynamic computation offloading between client and server environments, we aim to optimize overall latency and enhance application performance, particularly for real-time use cases.

While our analysis focuses specifically on latency as an exemplar, similar trade-off considerations apply to other critical factors including energy consumption. The principles underlying our hybrid approach can be extended to address these additional dimensions, underscoring the importance of developing adaptive inference solutions for intelligent applications, which will be verified in the experiment in Section 4.6.

2.2. Layer-wise partitioning insights for neural network models

In Section 2.1, we have established the theoretical basis for partitioning AI models across client and server environments. In this section, we explore the advantages of such partitioning paradigms through layer-level analysis of output data size and computational latency, aiming to optimize task distribution between the two platforms. We evaluate the data and computational characteristics of each layer in MobilenetV1 to determine the feasibility and efficiency of partitioning neural network computations. By analyzing layer-specific attributes, we can strategically allocate processing layers to either the client or the server, thereby optimizing end-to-end performance.

Per-layer Latency Analysis. As depicted in Fig. 2, the red bars sequentially represent the execution time of each layer within MobilenetV1 when implemented on WebGPU, and names of each layer in MobilenetV1 can be found in Table 1. Convolution layers, encompassing both pointwise and depthwise convolutions, emerge as the primary computational bottleneck, accounting for over 80% of the total execution time. Notably, pointwise convolution layers, despite their relatively low computational complexity, frequently exhibit higher latency than standard convolutional layers. This discrepancy arises from their complex memory access patterns, which introduce additional overhead. Conversely, batch normalization and global average pooling layers execute significantly faster. These findings underscore the diverse computational demands across different layer types, a critical consideration when devising partitioning paradigm.

Data Size Variation. The blue markers in Fig. 2 denote the output data size for each layer, which serves as the input to the subsequent layer. Early-stage convolution layers, particularly pointwise convolutions, generate substantial output data (as indicated by the tallest bars).

Table 1
Layers of MobilenetV1.

Index	Layer name	Index	Layer name	Index	Layer name	Index	Layer name
0	input	24	dw4	48	pw7_relu	72	pw11_relu
1	conv1	25	dw4_bn	49	dw8	73	pad12
2	bn1	26	dw4_relu	50	dw8_bn	74	dw12
3	relu1	27	pw4	51	dw8_relu	75	dw12_bn
4	dw1	28	pw4_bn	52	pw8	76	dw12_relu
5	dw1_bn	29	pw4_relu	53	pw8_bn	77	pw12
6	dw1_relu	30	dw5	54	pw8_relu	78	pw12_bn
7	pw1	31	dw5_bn	55	dw9	79	pw12_relu
8	pw1_bn	32	dw5_relu	56	dw9_bn	80	dw13
9	pw1_relu	33	pw5	57	dw9_relu	81	dw13_bn
10	conv2	34	pw5_bn	58	pw9	82	dw13_relu
11	dw2	35	pw5_relu	59	pw9_bn	83	pw13
12	dw2_bn	36	pad6	60	pw9_relu	84	pw13_bn
13	dw2_relu	37	dw6	61	dw10	85	pw13_relu
14	pw2	38	dw6_bn	62	dw10_bn	86	gap2d
15	pw2_bn	39	dw6_relu	63	dw10_relu	87	reshape1
16	pw2_relu	40	pw6	64	pw10	88	dropout
17	dw3	41	pw6_bn	65	pw10_bn	89	conv4
18	dw3_bn	42	pw6_relu	66	pw10_relu	90	reshape2
19	dw3_relu	43	dw7	67	dw11	91	Argmax
20	pw3	44	dw7_bn	68	dw11_bn		
21	pw3_bn	45	dw7_relu	69	dw11_relu		
22	pw3_relu	46	pw7	70	pw11		
23	conv3	48	pw7_bn	71	pw11_bn		

This is attributed to the application of hundreds of convolution kernels on input feature maps to extract discriminative features. The output size remains unchanged after batch normalization and activation layers. However, depthwise convolution layers drastically reduce data size, achieving up to a 6× reduction. Finally, the dense layer condenses the output to a single classification label. These trends in data size variation are instrumental in assessing communication overhead, a pivotal factor in optimizing partitioning paradigm.

Through comprehensive analysis of per-layer latency and data size variation, we derive the following insights: Each layer's computational and data profiles are inherently dependent on its type and position within the model. While later layers typically exhibit higher computational latency, their associated data sizes are significantly smaller compared to the input layer. This inverse relationship suggests that front-end layers process larger data volumes, whereas back-end layers impose heavier computational demands. These findings present strategic opportunities for partitioning neural network computations between the client browser and server browser. Specifically, offloading front-end layers to the client browser can minimize communication costs, while executing back-end layers in the server browser leverages its superior computational resources. These observations underscore the need for a nuanced approach to partitioning AI models, considering both computational latency and communication overhead.

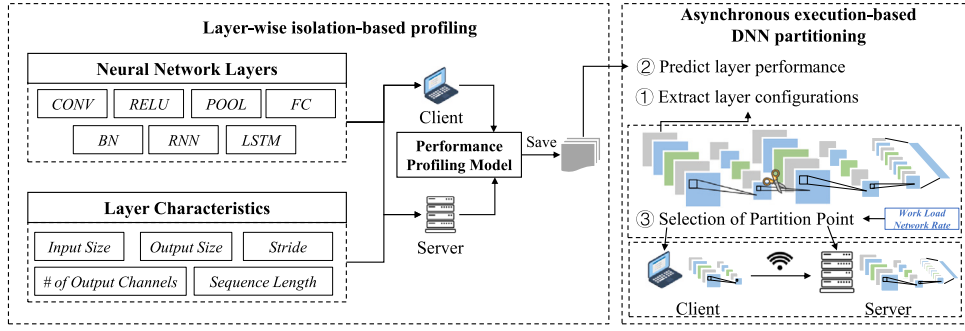


Fig. 3. nnWeb overview. Layer-wise isolation-based profiling is conducted offline to collect and analyze detailed characteristics of different neural network models. At runtime, asynchronous execution-based DNN partitioning continuously monitors the current wireless network status and the available computational resources on both the client and the server sides. Based on the profiling database and real-time conditions, it intelligently determines the optimal partitioning point for model inference.

3. Methods

The optimal partitioning point of the model depends on the neural network's topology, which is reflected in the variations of computation and data size at each layer (Section 2.2). Additionally, even for the same neural network structure, dynamic factors such as the state of wireless networks can influence the optimal partitioning point. Due to these dynamic factors, an automated system is required to intelligently select the optimal point for model partitioning to optimize total latency or energy consumption on mobile devices. To fulfill this requirement, we have designed an intelligent model partitioning framework, nnWeb. The nnWeb framework comprises two key components: layer-wise isolation-based profiling and asynchronous execution-based DNN Partitioning, both of which are designed to efficiently manage the execution of partitioning in intelligent applications. Fig. 3 illustrates the design of nnWeb.

3.1. Layer-wise isolation-based profiling

With the acceleration provided by WebGPU, the inference latency of neural networks has been dramatically reduced from the order of seconds to milliseconds (detailed in Section 3.2), making it comparable to communication latency. Moreover, modern neural networks consist of diverse layer types, each exhibiting unique characteristics such as varying input/output dimensions and distinct computational behaviors. To accurately predict the latency of each layer—and consequently determine the optimal partitioning point—we must model individual layers along with their specific parameters.

To this end, layer-wise isolation-based profiling is performed offline to systematically collect and analyze the detailed performance characteristics of each layer. We vary the parameters of individual layers and measure their execution times to capture performance patterns. Regression models are then trained to estimate execution latency on both the client and the server sides.

Consider a multi-layered model with M sequentially arranged layers $\mathcal{L} = \{L_0, L_1, \dots, L_{M-1}\}$, where each layer L_i processes intermediate data of size $S_i \in \mathbb{R}^+$. Let d_c and d_s denote the client and server devices, respectively. The execution latency of layer L_i on device $d \in d_c, d_s$ follows the statistical model:

$$\mathcal{T}_{L_i}^{(d)} = f_{L_i}^{(d)}(\mathbf{x}_{L_i}, \mathbf{y}_d) + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2), \quad (1)$$

where $f_{L_i}^{(d)}$ represents the device-specific regression function, $\mathbf{x}_{L_i}$ denotes layer configuration parameters (e.g., input dimensions, kernel size), and $\mathbf{y}_d$ captures device-dependent variables (e.g., browser load level). The Gaussian error term ϵ accounts for measurement noise and unmodeled effects. We systematically characterize the parametric space for each layer type as follows:

Convolution Layer. In the realm of convolution operations, where the kernel size is $K \times K$, the stride is S , and the number of output

channels is C_{out} , the parameters are meticulously selected to capture the layer's essence. We denote these parameters as $\mathbf{x}_{conv} = [\text{GFLOPs}, C_{in}, (\frac{K}{S})^2 C_{out}]^T$. Here, GFLOPs serves as a quantifier of the model's computational demand, C_{in} represents the count of input channels, and $(\frac{K}{S})^2 C_{out}$ encapsulates the spatial and channel capacity influenced by the kernel and stride. This parametrization not only reflects the layer's computational intensity but also its feature extraction capabilities, offering a comprehensive view of the convolutional process.

Fully Connected Layer. For a fully connected layer, the parameters are determined by considering the dimensions of input and output features, represented as D_{in} and D_{out} , as well as the counts of input and output neurons, denoted by N_{in} and N_{out} . Consequently, the selected parameters are $\mathbf{x}_{fc} = [D_{in}, D_{out}, N_{in}, N_{out}]^T$. This parametrization captures the layer's role in learning intricate patterns through its dense connections, bridging the gap between input representations and output predictions. It is worth noting that the parametrization approach for **argmax** and **softmax** layers follows a similar rationale, focusing on the transformation from input to output dimensions.

Pooling Layer. The pooling layer's parameters encompass the pooling type and the dimensions of the input and output feature maps. Let $\mathcal{P} \in \{\text{max}, \text{avg}\}$ symbolize the pooling type, with D_{in} and D_{out} representing the sizes of the input and output features, respectively. The parameters are thus defined as $\mathbf{x}_{pool} = [\mathcal{P}, D_{in}, D_{out}]^T$. This configuration not only signifies the pooling mechanism but also the spatial transformation it imparts, playing a pivotal role in downsampling and accentuating salient features within the data.

Activation Layer. Within the activation layer, there exists a direct mapping between input and output. The parameters are determined by the type of activation function $\sigma(\cdot)$, the number of output neurons N_{out} , and the spatial extent of the feature maps S_{out} . Therefore, the selected parameters are $\mathbf{x}_{act} = [\sigma, N_{out}, S_{out}]^T$. These parameters influence the layer's capacity to introduce non-linearity into the model, thereby enhancing its ability to approximate complex functions. It is important to highlight that **Normalization layers** utilize a comparable parametrization, with σ being substituted by various normalization techniques such as batch normalization.

Embedding Layer. The embedding layer serves as a critical component in processing discrete textual inputs by mapping them into continuous vector representations. As the computational demand of this layer intensifies with the length of the input data, the selected parameters are the input sequence length L , embedding dimension D , and vocabulary size V . Thus, $\mathbf{x}_{emb} = [L, D, V]^T$ encapsulates key aspects affecting the embedding process. The embedding dimension D determines the richness of the vector representation, while the vocabulary size V dictates the scope of unique tokens that can be effectively mapped, both of which are essential for capturing semantic meanings in natural language processing tasks.

Recurrent Layer. The recurrent layer stands out for its flexibility in accommodating diverse types of recurrent cell architectures, including Recurrent Neural Networks (RNN) [21], Long Short-Term Memory (LSTM) networks [22], and Gated Recurrent Units (GRU) [23]. These distinct cell types entail different parameter configurations, which in turn result in varying computational loads and memory capacities. For instance, LSTM layers typically impose a heavier computational load compared to RNN layers due to their sophisticated gating mechanisms that regulate information flow. In this research, the input shape, the dimension of the hidden state, and the specific recurrent kernel type are selected as input variables. For RNN cells $\mathcal{R} \in \{\text{RNN}, \text{LSTM}, \text{GRU}\}$, input shape D_{in} , and hidden state size H_{hid} , the parameters are $\mathbf{x}_{rnn} = [D_{in}, H_{hid}, \mathcal{R}]^T$. This parametrization effectively captures the recurrent layer's dynamic behavior, its capacity to retain temporal information, and its computational requirements, making it a vital component in sequence modeling tasks.

By training the prediction models on devices with abundant samples and then transferring them to target devices for testing, the sample efficiency of these prediction models can be greatly improved [24]. This approach allows us to implement a performance prediction model for each layer, creating a platform to analyze the execution time of each layer of the network across different devices, i.e. client and server, without running the model.

3.2. Asynchronous execution-based DNN partitioning

In traditional DNN partitioning strategies, initialization (i.e. model loading and data preparation) and execution form a strictly sequential pipeline, as illustrated in Fig. 4(a). Latency in any stage propagates through the entire workflow, exacerbating both overall response time and energy consumption [14]. In contrast, nnWeb leverages WebGPU to enable asynchronous execution between initialization and inference phases, as shown in Fig. 4(b), thereby eliminating unnecessary waiting and improving overall responsiveness. Moreover, WebGPU significantly reduces the execution time of in-browser AI inference. Fig. 4(c) presents a comparison between the end-to-end latency of a traditional partitioning strategy without WebGPU acceleration and that of the proposed nnWeb framework. In the absence of WebGPU, the combined inference and communication latency is approximately 1.5s, whereas with WebGPU acceleration, the process can be completed in approximately 27.3 ms. At the millisecond level, the proportion of communication overhead in the overall latency becomes more pronounced. As a result, accurately predicting the inference time of each DNN layer becomes increasingly important to enable effective trade-offs between computation and communication in terms of both latency and energy consumption. This necessitates the use of profiling models introduced in the previous section to guide the selection of optimal partitioning points.

It is important to note that in our inference scenario, model loading is considered a one-time cost during the startup phase and is thus excluded from per-inference latency assessments [25]. Research in this domain typically focuses on steady-state performance, where the dominant factors are the per-inference execution time and communication latency [26]. This design assumption allows profiling and optimization efforts to concentrate on the long-term behavior of the system, rather than transient initialization overhead.

To this end, an asynchronous execution-based DNN partitioning strategy is adopted to optimize the execution of multi-layered models by determining the optimal split point between local and remote processing. Our primary objective is to minimize the total execution time or energy consumption by efficiently distributing the computational workload between the browser and the server. This distribution is influenced by several factors, including the data size of each layer, performance prediction models for both local and remote inference, the current network transmission rate, and the local device load. The pseudocode of the algorithm is outlined in Algorithm 1, and the following section provides a detailed delineation of the algorithm's steps:

Algorithm 1 Asynchronous Execution-Based DNN Partitioning

```

1: Input:
   - Model layers  $\mathcal{L} = \{L_0, L_1, \dots, L_{M-1}\}$  and layer parameters  $\mathbf{x}_{L_i}$ 
   - Regression models  $f_{L_i}^{(d)}$  for each layer
   - Network transmission rate  $W$  and local workload  $\mathbf{y}_{load}$ 
   - Device power consumption  $P_{dc}, P_{ds}, P_{net}$ 
   - Optimization objective  $O \in \{\text{latency}, \text{energy}\}$ 
2: Output: Optimal partition index  $j^*$  and strategy type
3:  $T_{ds} \leftarrow \sum_{i=0}^{M-1} f_{L_i}^{(ds)}(\mathbf{x}_{L_i}) + \frac{S_0}{W}$ 
4:  $E_{dc} \leftarrow \sum_{i=0}^{M-1} f_{L_i}^{(dc)}(\mathbf{x}_{L_i}, \mathbf{y}_{load}) \cdot P_{dc}$ 
5: for  $j = 0$  to  $M - 1$  do
6:    $T_{client} \leftarrow \sum_{i=0}^j f_{L_i}^{(dc)}(\mathbf{x}_{L_i}, \mathbf{y}_{load})$ 
7:    $T_{net} \leftarrow \frac{S_j}{W}$ 
8:    $T_{server} \leftarrow \sum_{i=j+1}^{M-1} f_{L_i}^{(ds)}(\mathbf{x}_{L_i})$ 
9:    $T_{total}(j) \leftarrow T_{client} + T_{net} + T_{server}$ 
10:   $E_{client} \leftarrow \sum_{i=0}^j f_{L_i}^{(dc)}(\mathbf{x}_{L_i}, \mathbf{y}_{load}) \cdot P_{dc}$ 
11:   $E_{comm} \leftarrow 2 \cdot T_{net} \cdot P_{net}$ 
12:   $E_{server} \leftarrow \sum_{i=j+1}^{M-1} f_{L_i}^{(ds)}(\mathbf{x}_{L_i}) \cdot P_{ds}$ 
13:   $E_{total}(j) \leftarrow E_{client} + E_{comm} + E_{server}$ 
14: end for
15: if  $O = \text{Latency}$  then
16:    $j^* \leftarrow \arg \min_j T_{total}(j)$ 
17:    $O_{total}(j^*) \leftarrow T_{total}(j^*), O_{\Pi} \leftarrow T_{ds}$ 
18: else if  $O = \text{Energy}$  then
19:    $j^* \leftarrow \arg \min_j E_{total}(j)$ 
20:    $O_{total}(j^*) \leftarrow E_{total}(j^*), O_{\Pi} \leftarrow E_{dc}$ 
21: end if
22: if  $O_{\Pi} \leq O_{total}(j^*)$  then
23:   return Full  $\Pi$  execution
24: else
25:   return Co-processing at layer  $j^*$ 
26: end if

```

The algorithm begins by initializing the necessary parameters $\mathbf{x}_{L_i}$ and variable $\mathbf{y}_d$. Here, $\mathbf{x}_{L_i} \in \{\mathbf{x}_{conv}, \mathbf{x}_{fc}, \mathbf{x}_{pool}, \mathbf{x}_{act}, \mathbf{x}_{emb}, \mathbf{x}_{rnn}, \dots\}$ represents parameters of layers such as convolutional and fully connected layers. $\mathbf{y}_d$ includes the local load level $\mathbf{y}_{load}$ on the browser side, which can be used to ensure accurate performance predictions. Performance profiling models $f_{L_i}^{(d)}$ are employed to estimate both the computational latency of executing individual layers on both client and server browsers and the associated energy consumption during their execution. Beyond predicting execution time, these models are extended to account for energy efficiency—a critical consideration in optimizing model partitioning strategies, particularly for resource-constrained mobile devices with limited battery capacity. Energy consumption is quantified by multiplying the layer's execution time by the device's power draw during that period. This dual objective enables our framework to simultaneously evaluate latency and energy efficiency, thereby facilitating more informed decisions in distributed neural network partitioning.

To begin with, it is posited that all computational processes are executed on the server, which serves as the foundational benchmark for the latency threshold of the system. Subsequently, the algorithm calculates the aggregate time required, a process that includes the execution latency of each layer on the server as well as the transmission delay for the initial data set. When estimating data transmission latency, potential discrepancies between the transmission rate of the client and the reception rate of the server are taken into account. However, given the relatively modest sizes of models utilized within the client-server architecture, a simplifying assumption is adopted, positing symmetry in communication between the browser and server. More precisely, it is assumed that the data transmission rate from the client

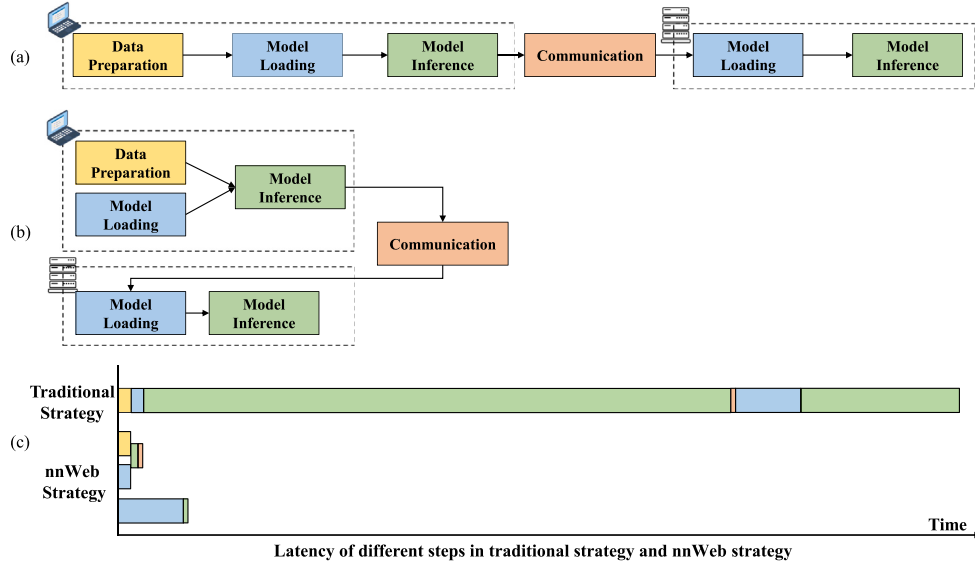


Fig. 4. MobilenetV1 partitioning strategies. (a) Traditional partitioning strategy [14], with a combined inference and communication latency of 1.5 s, which follows a sequential pipelining approach; (b) nnWeb partitioning strategy under WebGPU, with a combined inference and communication latency of 27.3 ms, which employs asynchronous execution; (c) Latency comparison of the two strategies along the same timeline.

to the server and the reception rate at the server from the client are equivalent, both being designated as the network transmission rate W . This assumption facilitates a streamlined analysis while maintaining a reasonable approximation for practical scenarios involving such architectures. The total time assumed for operations running entirely on the server (denoted as **server-only**), designated as T_{server} , serves as a critical baseline for subsequent analysis and optimization endeavors:

$$T_{d_s} = \sum_{i=0}^{M-1} f_{L_i}^{d_s}(\mathbf{x}_{L_i}) + T_{ori}^{net}, \quad (2)$$

where $T_{ori}^{net} = S_0/W$ denotes the data transmission delay incurred when transmitting the original data to the server without processing through L_0 .

Similarly, the total energy consumption for operations running entirely in the client browser (denoted as **client-only**), designated as E_{d_c} , can be estimated as:

$$E_{d_c} = \sum_{i=0}^{M-1} f_{L_i}^{(d_c)}(\mathbf{x}_{L_i}) \times P_{d_c}, \quad (3)$$

where P_{d_c} is the power consumption of the client during computation.

After determining the server-only execution latency T_{d_s} and client-only energy consumption E_{d_c} , we compare them with our proposed partitioning strategy. Specifically, to rigorously identify the optimal partitioning strategy between client and server resources, the proposed algorithm conducts a systematic evaluation of computational latency and energy consumption.

From the perspective of **latency optimization**, for the i th layer, the algorithm calculates the predicted latency for executing layer L_i on the browser, denoted as $\mathcal{T}_{L_i}^{d_c} = f_{L_i}^{d_c}(\mathbf{x}_{L_i}, \mathbf{y}_{load})$. Here, $\mathbf{y}_{load}$ represents the local load level on the client device, which affects execution latency due to resource contention and computational throttling. In contrast, server execution latency for layer L_i , $\mathcal{T}_{L_i}^{d_s} = f_{L_i}^{d_s}(\mathbf{x}_{L_i})$, is modeled without an explicit load parameter, as server environments typically maintain stable load conditions. For each potential partitioning point $j \in \{0, 1, \dots, M-1\}$, the algorithm calculates the total latency T_{total} as follows:

$$T_{total}(j) = \underbrace{\sum_{i=0}^j \mathcal{T}_{L_i}^{d_c}}_{\text{Client processing}} + \underbrace{\mathcal{T}_{L_j}^{net}}_{\text{Data communication}} + \underbrace{\sum_{i=j+1, i < m}^{M-1} \mathcal{T}_{L_i}^{d_s}}_{\text{Server processing}}, \quad (4)$$

where the first term in Eq. (4) represents the cumulative latency for executing layers L_0 to L_j on the client browser; $\mathcal{T}_{L_j}^{net} = S_j/W$ denotes the data communication latency incurred when transferring intermediate outputs between the j th and $(j+1)$ th layers across the network; the last term represents the cumulative latency for executing layers L_{j+1} to L_{M-1} on the server browser.

From the perspective of **energy optimization**, the algorithm estimates the energy consumption for each layer on both the client and server, while also incorporating the energy consumed during data transmission. For the client and server, energy consumption $E_{L_i}^{d_c}$ and $E_{L_i}^{d_s}$ are derived by multiplying the layer latency by the power consumption of client and server P_{d_c} and P_{d_s} respectively. Data communication energy is computed as $2 \times T_{L_j}^{net} \times P_{net}$, where P_{net} represents the power consumption of the network interface, and the factor of 2 accounts for both sending and receiving data. Thus, for each potential partitioning point $j \in \{0, 1, \dots, M-1\}$, the algorithm calculates the total energy consumption E_{total} as follows:

$$E_{total}(j) = \underbrace{\sum_{i=0}^j \mathcal{T}_{L_i}^{d_c} \times P_{d_c}}_{\text{Client energy}} + \underbrace{2 \times \mathcal{T}_{L_j}^{net} \times P_{net}}_{\text{Communication energy}} + \underbrace{\sum_{i=j+1}^{M-1} \mathcal{T}_{L_i}^{d_s} \times P_{d_s}}_{\text{Server energy}}. \quad (5)$$

The optimal partitioning index j^* is then determined through:

$$j^* = \arg \min_{j \in \{0, \dots, M-1\}} O_{total}(j), \quad (6)$$

$$\text{Partitioning Strategy} = \begin{cases} \text{Full } \Pi \text{ execution} & \text{if } O_{\Pi} \leq O_{total}(j^*) \\ \text{Co-processing at layer } j^* & \text{otherwise,} \end{cases} \quad (7)$$

where $O_{total}(j)$ represents the optimization objective, which can be either $T_{total}(j)$ for latency optimization or $E_{total}(j)$ for energy optimization. The choice of $O_{total}(j)$ depends on the specific optimization goal: $O_{total}(j) = T_{total}(j)$ for time optimization and $O_{total}(j) = E_{total}(j)$ for energy optimization. O_{Π} denotes the metric corresponding to the chosen optimization objective (either T_{d_s} or E_{d_c}), and Π corresponding to the choosing between client or server. This strategy ensures that the optimal demarcation point between local and remote processing is identified based on the prioritized optimization goal.

To accommodate fluctuations in network transmission rates and local computational workloads, the nnWeb partitioning algorithm dynamically updates the model split point based on Eqs. (4) to (6),

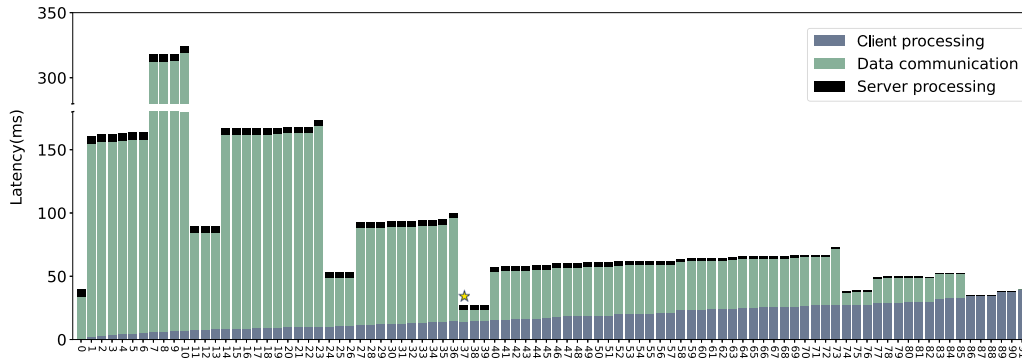


Fig. 5. The total latency of MobilenetV1 when choosing different partition points. The partition points for best latency are each marked by $\star$.

thereby maintaining optimal partitioning decisions under varying runtime conditions. This adaptability ensures that inference performance remains robust across heterogeneous and dynamic execution environments.

Furthermore, to fully leverage the asynchronous execution paradigm provided by WebGPU, the predicted partition point is proactively communicated to both the client and the server at runtime. This design enables parallel initialization on both sides, effectively eliminating the dependency whereby inference execution must wait for model initialization to complete. In particular, the server can begin loading its portion of the model without waiting for input data from the client, thereby reducing idle time and improving system-wide efficiency in terms of both latency and energy consumption. This asynchronous execution-based DNN partitioning strategy adopted by nnWeb ensures flexibility and minimizes overall execution time by thoroughly evaluating and selecting the most efficient processing configuration—whether it involves full offloading to the server or collaborative execution with an optimal partitioning point j^* .

4. Evaluation

4.1. Experimental setup and configuration

We build a prototype of nnWeb by creating TensorFlow.js [27] instances as the foundational infrastructure for both the client and server-side components. We implement a client-server interface using Websockets, a transmission control protocol (TCP) network protocol that enables bidirectional communication between the browser and server. To enable dynamic selection of partition points, the AI model can be loaded on both the client and server. We evaluate nnWeb under a WiFi network environment with a 100Mbps bandwidth, using two types of GPU backends: an integrated GPU (Intel UHD Graphics 730, 75 W) and a discrete GPU (NVIDIA GTX 1650 Ti, 75 W). The communication module, which includes wireless transmission and the WebSocket communication stack, contributes an additional power consumption of approximately 15 W during active data exchange.

4.2. Latency reduction analysis

In collaborative computing between mobile devices and servers, selecting an optimal model partitioning point is crucial for improving computational efficiency. Our previous analysis in Section 2.2 shows that different layers have varying computation and communication demands: front-end layers process large data with low computation intensity, while back-end layers handle smaller data but need high computation power. Based on these findings, this section explores how different partitioning points affect the total latency of MobilenetV1 when offloading computations from the browser to the server via WiFi, aiming to find the best partitioning strategy that balances communication and computation resources.

Table 2

Benchmark specifications.

APP	Network	Input	Layers
NLP	Spam detection	Word vectors	6
ASR	Speech commandsV1	Speech commands	23

Fig. 5 depicts the total latency of MobilenetV1 across various partitioning points, with each bar representing the latency corresponding to a specific partition point. Specifically, the leftmost bar signifies cloud-only processing, where all layers are executed on the server browser. As the partition point migrates to the right, an increasing number of layers are processed locally on the client browser, with the rightmost bar denoting client-only processing.

The experimental results unveil that partitioning at a middle layer effectively equilibrates data communication overhead and the exploitation of the server computational power. For instance, the partitioning point *dw6* attains the lowest latency by leveraging the computing capabilities of both the client and server. This strategy enhances latency by 31% in comparison to cloud-only processing and by 30% relative to client-only processing.

Based on the experimental evidence, we conclusively establish that partitioning at the layer granularity markedly ameliorates latency by balancing the trade-offs between communication and computation. For MobilenetV1 with WiFi communication, the optimal partitioning point resides within the middle layers of the model. At this juncture, the amalgamated benefits of diminished data communication and efficient server-side computation are optimized, culminating in the most efficient overall computational process.

4.3. Generalization across application domains

Neural network model partitioning across different computing platforms is crucial for balancing computational efficiency and communication overhead in modern AI system design. While prior analyses on Computer Vision (CV) models have shown the importance of considering layer-type characteristics and model topology for determining optimal partitioning strategies, the unique demands of various application domains require a more comprehensive exploration of partitioning approaches. To demonstrate the wide applicability of our partitioning strategy, we extend our exploration to two additional intelligent applications: Natural Language Processing (NLP) and Automatic Speech Recognition (ASR). Our aim is to show that the partitioning approach can be effectively applied beyond CV models to other domains, despite their distinct computational and data features. Detailed specifications of the two benchmark models from the TensorFlow.js library are provided in Table 2. The partitioning for each model begins at the first non-input layer and ends at the final layer, and we use a wireless network setup to evaluate the total latency.

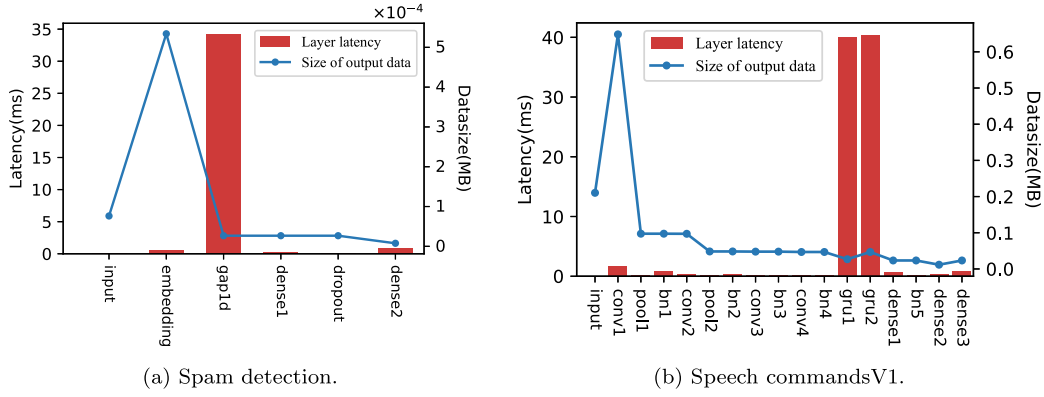


Fig. 6. The per-layer latency on the WebGPU and size of data after each layer's execution.

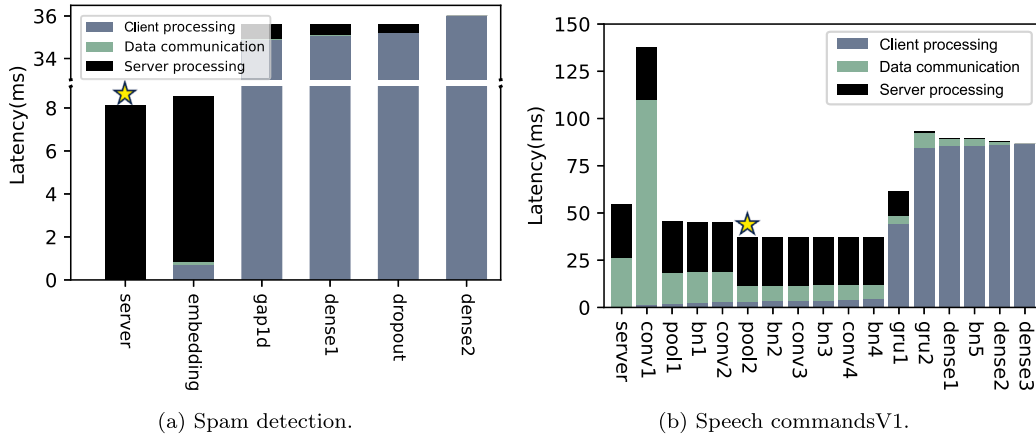


Fig. 7. The total latency when choosing different partition points.

NLP applications. The layer decomposition for NLP models is presented in Fig. 6(a). Contrary to CV models, NLP models typically integrate inputs with temporal dependencies, which involve embedding layers and RNN layers. Owing to their sequential nature, these layers are computationally intensive. However, the shorter input sequences in NLP models offer prospects for partitioning near initial stages. This implies that partitioning strategies can be tailored to exploit the unique attributes of NLP models, such as their temporal dependencies and shorter input sequences. As depicted in Fig. 7(a), the optimal partition point for NLP models is inclined to be nearer to the input layers, where the computational load can be effectively distributed between the browser and the server.

ASR applications. The layer decomposition for ASR models is shown in Fig. 6(b). These models integrate LSTM [22] layers and GRU [23] layers, both of which are computationally intensive. Similar to NLP models, the shorter input sequences in ASR applications permit partitioning near initial stages. This observation further underpins the adaptability of our partitioning approach, as it highlights the capacity to adjust to diverse model architectures and input characteristics. Fig. 7(b) illustrates the optimal partition point for ASR models, which aligns with the trend seen in NLP applications.

Our investigation across CV, NLP, and ASR models reveals that optimal partitioning strategies are strongly influenced by model architectures and input characteristics:

- In CV models, convolutional layers typically expand data dimensionality, whereas global average pooling layers substantially reduce it. Models with similarly sized fully connected and activation layers exhibit negligible variation in per-layer latency and

data size. Critically, the optimal partition point in CV models depends on their topological structure, with mid-layer partitioning often proving most effective.

- NLP models, in contrast, process temporally dependent inputs through embedding and recurrent layers, which impose high computational demands due to sequential processing. Their shorter input sequences facilitate early-stage partitioning, with optimal split points occurring closer to the input layers. This strategy effectively balances computational load between client and server.
- ASR models demonstrate analogous behavior, employing compute-intensive GRU and LSTM layers. Similar to NLP models, their shorter input sequences enable favorable partitioning near the initial layers, reinforcing the observed trend in sequence-based architectures.

These findings not only demonstrate the versatility of our approach across multiple domains but also reveal the need for a dynamic partitioning system that can adapt to the specific characteristics of each model.

4.4. Multi-model validation and benchmarking

Building upon the exploration of partitioning strategies across diverse application domains in Section 4.3, this section is designed to further verify and consolidate the findings. Specifically, experiments in Section 4.3 revealed that different model architectures and input characteristics significantly influence optimal partitioning strategies, with CV models benefiting from middle layer partitioning, and NLP and ASR models favoring partitioning near the initial layers due to their temporal dependencies and shorter input sequences. To ensure

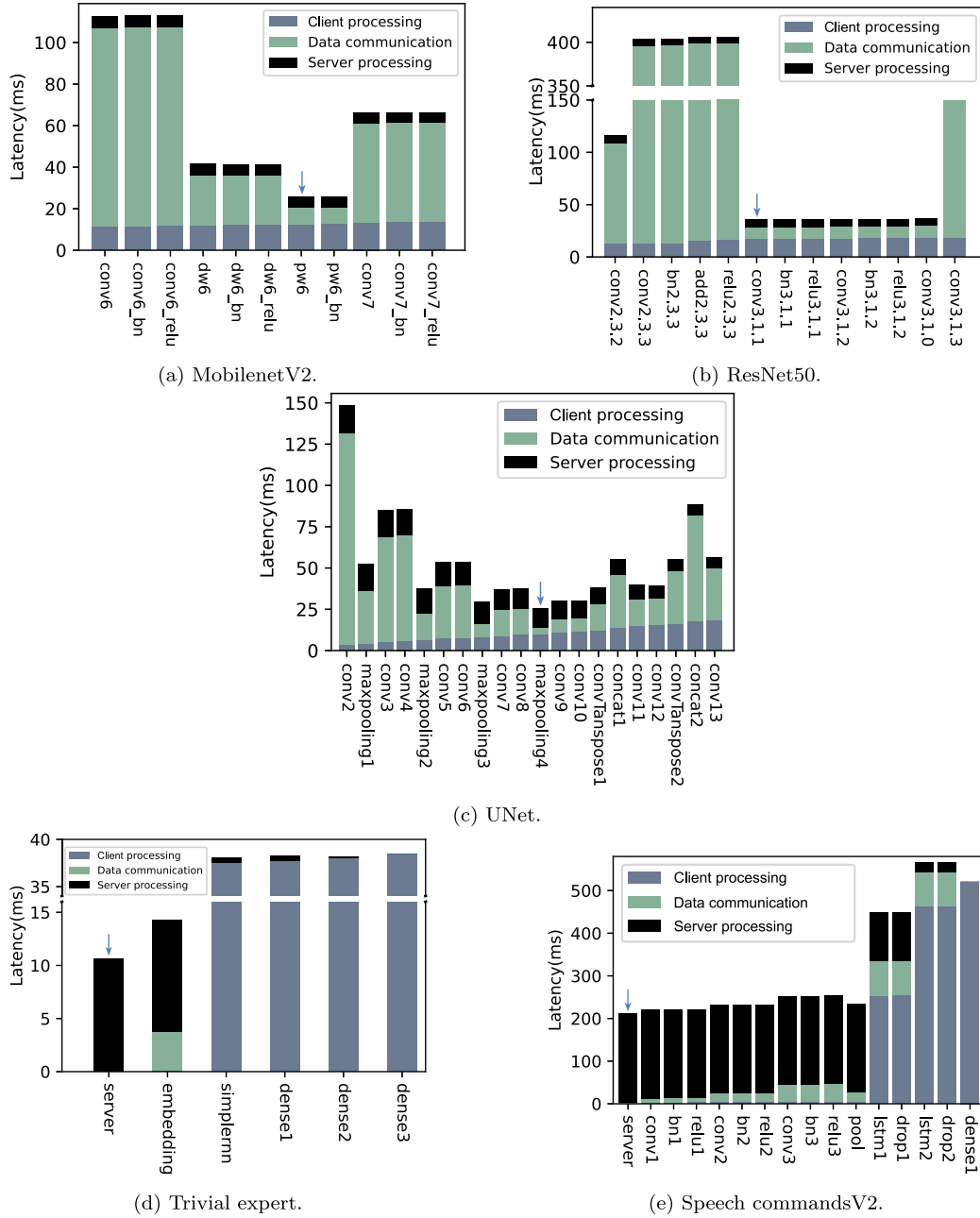


Fig. 8. Predicted partition points by nnWeb with WiFi on integrated graphics. The blue arrow is the optimal partition points predicted by nnWeb. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 3

Benchmark specifications.

APP	Network	Input	Layers
CV	MobilenetV2	Image	156
CV	ResNet50	Image	177
CV	UNet	Image	31
NLP	Trivial expert	Word vectors	5
ASR	Speech commandsV2	Speech commands	17

the reliability and validity of these insights, we employ four distinct models in Table 3 to rigorously assess the accuracy of nnWeb predictions regarding optimal partition points under real-world conditions, particularly using WiFi on integrated graphics.

Fig. 8 shows the comparison between the predicted partition points of nnWeb and the actual measured total latency. The experimental

results demonstrate the high prediction accuracy and reliability of nnWeb. As shown in Fig. 8(a) and (b), for CV models, communication latency is the dominant factor in total latency, significantly outweighing layer latency, while the partition point generates minimal output data. Fig. 8(c) and (d) show that for NLP and ASR layer models, which are characterized by small input data and long RNN layer execution times, server-side inference emerges as the optimal solution.

In Fig. 9, we compare the average communication latency and output data size at the partition point with the average values of all layers before and after it. For CV models such as ResNet50, communication latency constitutes a substantial portion of the total latency. Notably, in MobilenetV2, although the partition point's latency is not the smallest compared to the average latencies before and after, the data output at the partition point is minimal. This observation aligns with the nnWeb prediction in Fig. 8, where the communication time at the optimal partition point for CV models is relatively small compared

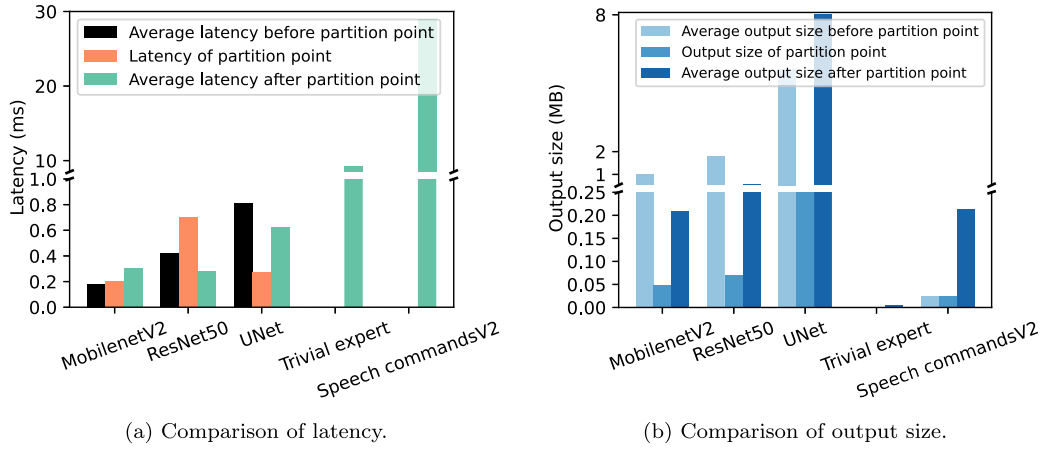


Fig. 9. Comparing execution time and output data size at partition point with average execution time and output data size before and after the partition point.

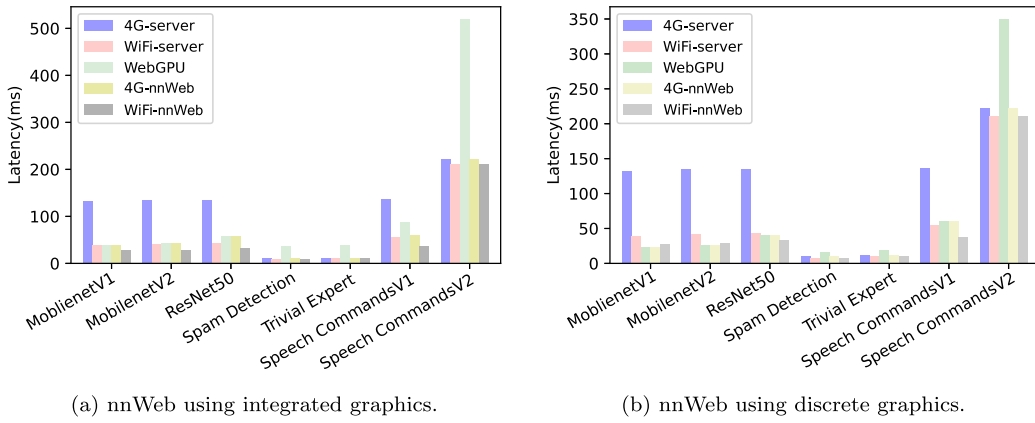


Fig. 10. Latency speedup achieved by nnWeb normalized to the status quo approach.

to other partition points. For NLP and ASR models, given the minimal input data size and the relatively large execution time of RNN layers, server-side inference emerges as the optimal choice. This conclusion also corresponds to the optimal partition point predicted by nnWeb in Fig. 8.

4.5. Assessment of latency optimization in nnWeb

In this section, we evaluate the performance of nnWeb in optimizing total latency across diverse configurations and application domains. The assessment is conducted by analyzing the partition point selection and the corresponding latency improvements achieved by nnWeb in comparison to existing approaches.

Partition point selection. Table 4 presents the partition point choices for latency optimization across 32 configurations, encompassing seven models from TensorFlow.js model library [27], two wireless network types, and two GPU backends. nnWeb consistently identifies the optimal partition points for minimizing total latency across all permutations of models and configurations. For instance, in CV models such as MobilenetV1 and MobileNetV2, nnWeb selects middle-layer partition points (e.g., *dw6*, *pw6*) under WiFi conditions, aligning with the findings in Section 4.4 that middle-layer partitioning is beneficial for CV models. In contrast, for NLP and ASR models, nnWeb predominantly opts for server-side inference due to their computationally intensive layers and shorter input sequences, which corroborates the earlier observations in Section 4.4.

Latency improvement. Fig. 10 illustrates the latency improvements achieved by nnWeb relative to current methods across seven benchmarks on WiFi and 4G networks. The results demonstrate that

Table 4

nnWeb's prediction partition points for best total latency.

Models	Integrated graphics		Discrete graphics	
	4G	WiFi	4G	WiFi
MobilenetV1	Argmax	dw6	Argmax	dw6
MobilenetV2	Argmax	pw6	Argmax	pw6
ResNet50	Argmax	conv3.1.1	Argmax	conv3.1.1
Spam detection	Server	Server	Server	Server
Trivial expert	Server	Server	Server	Server
Speech commandsV1	conv1	bn4	bn4	dense3
Speech commandsV2	Server	Server	Server	Server

nnWeb significantly reduces total latency across diverse application domains and hardware configurations. For CV applications, nnWeb achieves substantial latency acceleration, particularly when configured with high-end GPUs. For NLP applications, nnWeb delivers significant latency reductions under WiFi conditions by leveraging the computational power of the server. In the case of ASR applications, nnWeb identifies that performing the entire model on the server yields the optimal latency performance. Across all validated configurations, nnWeb's chosen inference modes result in time savings of up to 30% on the cloud server and 52% on the browser, with peak improvements of 33% and 69%, respectively.

Comparing nnWeb to baseline. To further validate the effectiveness of nnWeb, we compare its hierarchical performance prediction capabilities with those of Neurosurgeon [18] on the VGG model. In comparison to Neurosurgeon, our profile model has been significantly enhanced through the incorporation of a more extensive range of layer types and a substantially larger volume of input data. This refinement

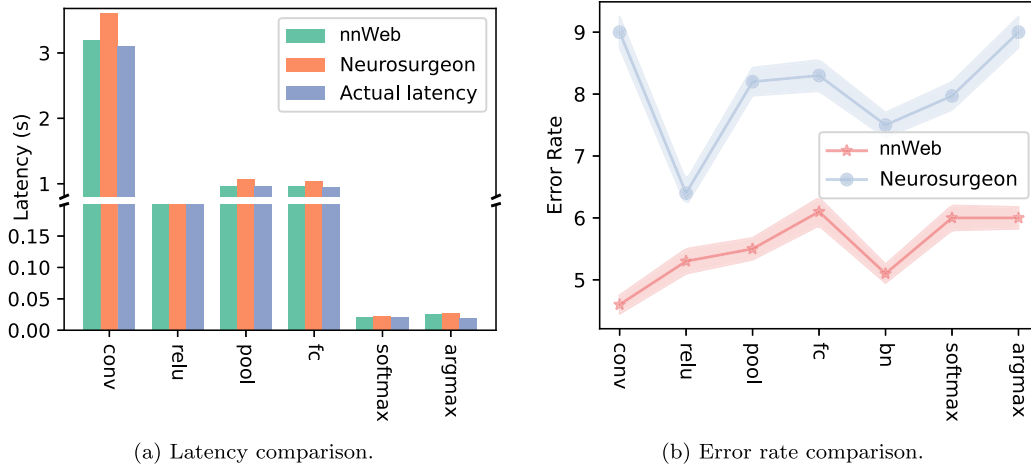


Fig. 11. Comparison of nnWeb and Neurosurgeon [18]. (a) Comparison of total latency grouped by layer. (b) Average error rates across layers.

results in markedly improved prediction accuracy. Neurosurgeon relies on the conventional heterogeneous computing framework of mobile device CPUs/GPUs and cloud server resources. This reliance inherently imposes limitations on its scalability and cross-platform capabilities when contrasted with nnWeb, which is built upon the WebGPU foundation. Furthermore, nnWeb adopts an asynchronous pipeline during both the model loading and inference phases, leveraging the asynchronous execution capabilities of WebGPU. This stands in contrast to Neurosurgeon's sequential pipeline approach and thereby offers greater efficiency in task execution.

We aggregate predicted latency by layer categories and compare it with actual latency. As shown in Fig. 11(a), the predicted execution times generated by nnWeb are significantly closer to the actual measured times than those predicted by Neurosurgeon. Fig. 11(b) displays the average error rates for several layers on the test model. The simpler regression prediction model employed by Neurosurgeon, which uses fewer input variables, produces larger prediction errors. In contrast, the enhanced regression prediction model developed for nnWeb, which incorporates additional input variables and categories, yields smaller prediction errors. These results underscore the superior accuracy of nnWeb in performance prediction and further validate its reliability as a dynamic partitioning system.

The experimental evidence highlights the ability of nnWeb to dynamically adapt partitioning strategies based on model architecture, input characteristics, and network conditions. This adaptability is critical for optimizing computational efficiency and communication overhead in modern AI systems. The consistent performance improvements and prediction accuracy observed across diverse configurations validate the effectiveness of nnWeb as a dynamic partitioning system capable of addressing the unique demands of various application domains.

4.6. Assessment of nnWeb's energy optimization

In accordance with Eq. (5) from Section 3.2, we further evaluate the partitioning strategy based on energy consumption metrics. We continue to utilize the benchmarks from Table 3. Fig. 12 illustrates a comparison between the predicted partition points of nnWeb and the actual measured total energy consumption. It can be observed that, in contrast to the latency perspective where communication latency holds a significant proportion, from the energy consumption perspective, communication energy consumption becomes the least critical factor. This indicates that when optimizing the end-to-end performance of a system, it is essential to select appropriate optimization metrics based on specific scenarios.

Furthermore, in Fig. 12, MobilenetV2, ResNet50, and Trivial expert tend to operate primarily on the server. This is attributed to the fact

that these models have a relatively low total computational load, and the server GPU demonstrates a significantly higher energy efficiency compared to the integrated graphics card, making server operation the optimal energy efficiency trade-off.

For UNet and Speech commandsV2, they choose to operate solely on the client browser, indicating that in these types of models, migrating a layer to the server results in an increase in server energy consumption that exceeds the reduction in local energy consumption. This may be because these two types of models have a high computational load and feature a large number of concatenation or loop structures in their latter halves. Although there is a slight decline in computational performance on the low-power local hardware, it can achieve a more energy-saving effect.

Moreover, the energy consumption for each application under client-only, server-only, and optimally partitioned configurations is reported in Fig. 12 and summarized in Table 5. The table presents the total energy consumption at the optimal partition points, along with the corresponding reduction in energy usage compared to the client-only and server-only baselines. Results show that, by identifying optimal partition points using nnWeb, total system energy consumption can be reduced by 11.29% to 43.98%.

4.7. Robustness under local load and network variations

In this section, we examine the robustness of nnWeb under varying local load and network conditions. The analysis focuses on two key aspects: the impact of local load variations and the system's response to network bandwidth fluctuations.

Impact of local load variations. Fig. 13(a) illustrates how nnWeb adapts to varying levels of local load. As the local load increases, nnWeb dynamically adjusts its partitioning strategy. The results indicate that when the local load is high, executing all layers of the model on the server becomes the optimal choice. This observation highlights nnWeb's ability to balance computational efficiency and communication overhead based on load demands.

Adapting to network fluctuations. Fig. 13(b) depicts the fluctuations in wireless bandwidth over time (top black line) and compares the total latency of cloud-only processing versus nnWeb (bottom lines). The experiment reveals that server-only processing is highly susceptible to network variations, leading to significant latency increases during low-bandwidth phases. In contrast, nnWeb demonstrates robustness against network fluctuations by adaptively adjusting its partitioning choices and data transfer volumes based on available bandwidth. This adaptability ensures consistent performance even under varying network conditions.

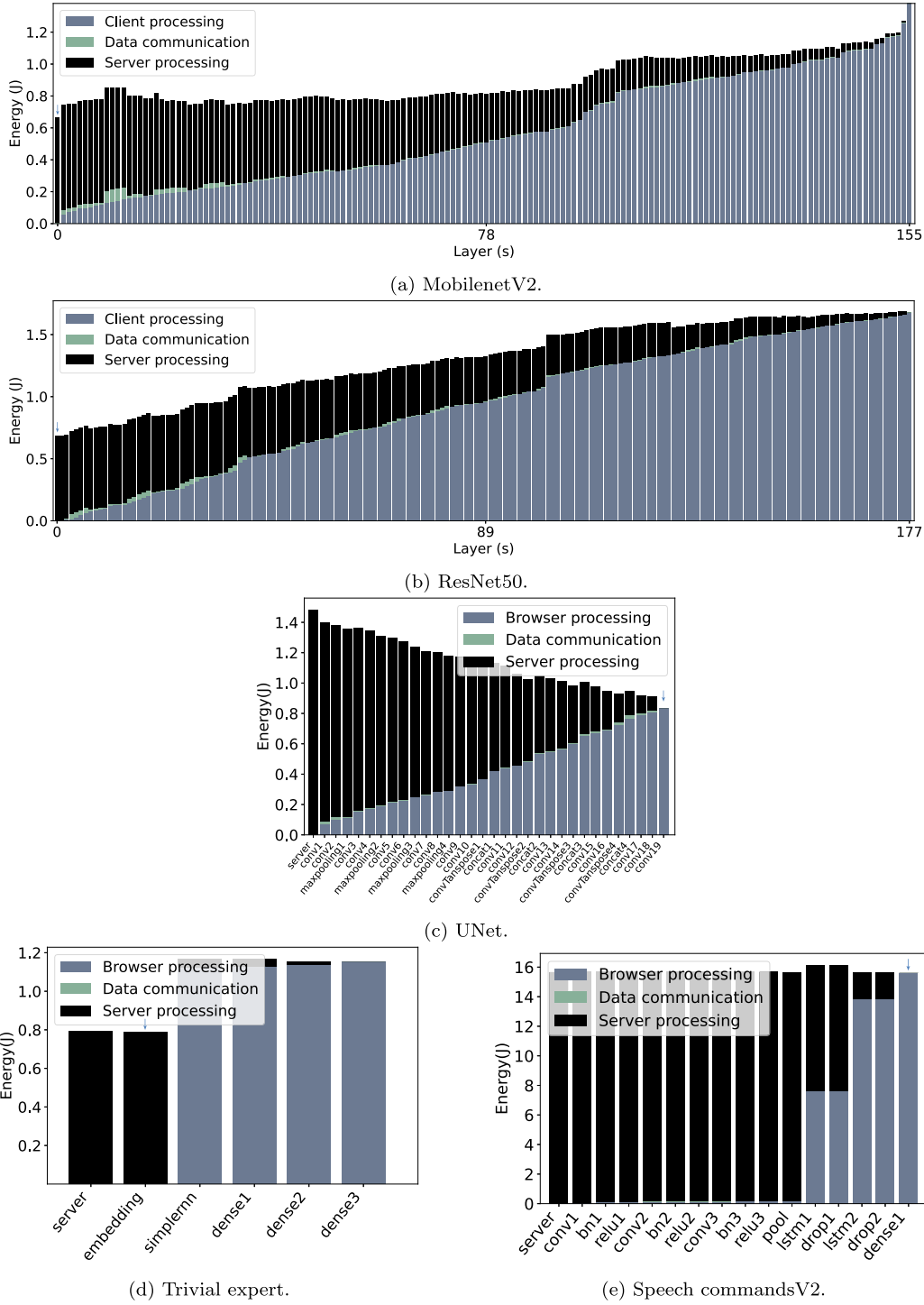


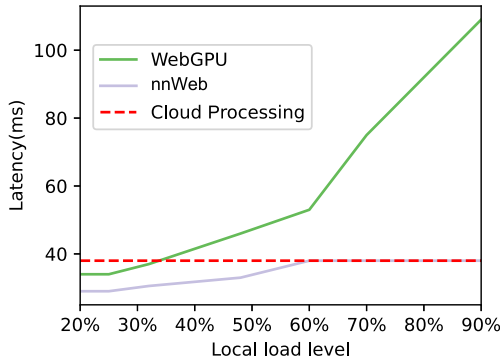
Fig. 12. nnWeb's prediction partition points with WiFi on integrated graphics. The blue arrow is the optimal partition points predicted by nnWeb. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Adapting to various network environments. As shown in Table 4, nnWeb selects different partition points for CV models under varying network conditions, whereas the partition points for ASR and NLP models remain unchanged. This observation aligns with the findings in Section 4.4, where it was demonstrated that, for CV models, communication latency is the dominant contributor to total system latency. Consequently, nnWeb dynamically adjusts the model partition point according to the current network conditions to minimize total latency.

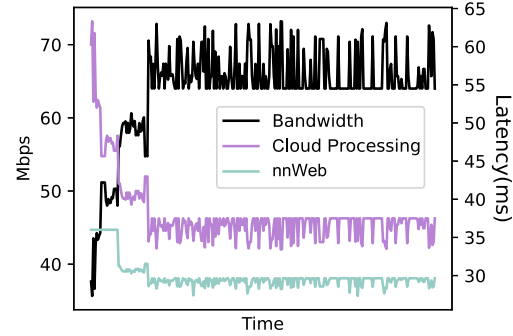
In contrast, for ASR and NLP models, the proportion of communication latency is significantly smaller compared to computation latency. As a result, changes in network bandwidth have minimal impact on their optimal partitioning decisions. To further validate this behavior, we select three representative computer vision models—MobilenetV2, ResNet50, and U-Net—and evaluated nnWeb's partitioning decisions under three network conditions (4G, WiFi and 5G), with bandwidth values referenced from Rochman et al. [28]. The results are illustrated in Fig. 14. As the available bandwidth increases, the proportion of

Table 5
Energy consumption reduction of partition points compared with web-only or server-only.

APP	Client-only (J)	Server-only (J)	Partition point and energy (J)	Reduction (compared to client-only)	Reduction (compared to server-only)
MobilenetV2	0.8534	0.6633	Server (0.6633)	22.28%	–
ResNet50	0.7731	0.6858	Server (0.6858)	11.29%	–
UNet	0.8322	1.4856	conv19 (0.8322)	–	43.98%
Trivial expert	1.1550	0.7950	Embedding (0.7909)	31.53%	0.52%
Speech commandsV2	15.6150	15.6678	dense1 (15.6150)	–	0.34%



(a) nnWeb adjusts partition based on local load levels.



(b) The impact of wireless bandwidth variance.

Fig. 13. The impact of local load variations and network fluctuations on nnWeb. (a) Local load impact. (b) Network fluctuation impact.

communication latency decreases significantly. Accordingly, nnWeb tends to offload a greater portion of computation to the server. This trend is most evident in the case of the U-Net model. Since the impact of communication latency becomes negligible at higher bandwidths, and the server typically offers superior computational resources compared to the client, fully delegating model inference to the server becomes the optimal strategy for minimizing total system latency.

The experimental results highlight the effectiveness of nnWeb in dynamically adapting to variations in local load and network conditions. By intelligently adjusting partitioning strategies, nnWeb is able to maintain optimal performance under diverse operational scenarios, thereby reinforcing its value as a dynamic partitioning system for modern AI applications.

5. Discussion

In this study, we have introduced nnWeb, an innovative dynamic partitioning system designed to optimize collaborative computing between mobile devices and servers. Our experimental findings provide compelling evidence of the efficacy of nnWeb in reducing latency and energy consumption. However, further exploration reveals several areas that warrant deeper investigation.

5.1. Multi-objective optimization

While nnWeb has achieved remarkable results in optimizing latency and energy consumption, the current research is predominantly focused on single-objective optimization, either latency or energy. In practical applications, system designers are often required to balance multiple

objectives, such as latency, energy consumption, and hardware costs. At present, nnWeb's optimization strategy is limited to considering only latency and energy, with other critical factors such as hardware costs and model accuracy loss remaining unaddressed. Future research should explore multi-objective optimization methodologies to accommodate the diverse and often competing demands encountered in real-world scenarios.

To address these challenges, several multi-objective optimization techniques can be employed. For instance, Pareto optimization algorithms and evolutionary algorithms have been extensively applied in distributed computing systems and resource allocation problems [29]. These methods are particularly effective in identifying Pareto-optimal trade-offs among conflicting objectives such as latency, energy consumption, and computational resource utilization. When applied to nnWeb, these methods enable the systematic exploration of solution spaces to balance workload distribution, communication overhead, and service quality requirements. By leveraging evolutionary mechanisms, they can dynamically adapt to heterogeneous resource constraints and stochastic workload patterns, thereby optimizing end-to-end performance while maintaining robustness in dynamic edge-cloud environments.

5.2. Reinforcement learning integration

The prediction model of nnWeb is trained using an offline approach, which relies on pre-collected data to model layer latency and energy consumption. While this method provides a solid foundation, it may not fully adapt to the dynamic variations encountered during actual operation. The inherent limitation of offline training is its inability to

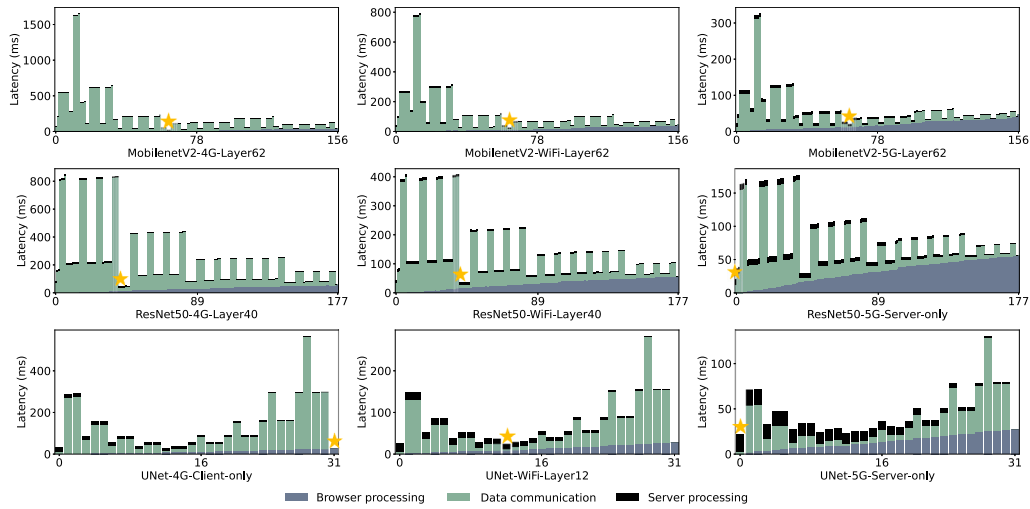


Fig. 14. Performance evaluation of nnWeb utilizing various models under different network conditions, highlighting optimal partition points (layer index or client-only/server-only) marked by asterisks.

capture the nuanced changes that occur during system operation, which can lead to discrepancies between predicted and actual performance.

To mitigate this issue, the integration of reinforcement learning (RL) into nnWeb's framework is proposed. RL has demonstrated remarkable adaptability in dynamic systems by interacting with the environment to adjust strategies in real-time [30]. For example, RL has been successfully applied for real-time optimization in network traffic management [31] and autonomous driving systems [32]. By incorporating RL into nnWeb, the system could dynamically refine its partitioning strategies based on real-time operational conditions, thereby enhancing both the robustness and predictive accuracy of the system.

5.3. Attention-based model architectures

Current research in neural network partitioning predominantly addresses conventional architectures such as CNNs, ResNets, and RNNs, which exhibit sequential and largely linear layer structures. These traditional models lend themselves well to established partitioning paradigms due to their inherent structural simplicity. However, the rapid evolution of neural network architectures has introduced significantly more complex, non-linear designs, most notably Transformer-based models [33] incorporating sophisticated attention mechanisms.

Despite their widespread adoption and superior performance across various domains, the partitioning of Transformer architectures remains understudied. Unlike their CNN or RNN counterparts, Transformers present unique challenges for model partitioning due to their non-sequential organization, dynamic computational graphs, and the complex interdependencies introduced by self-attention mechanisms. These architectural innovations fundamentally alter the computational characteristics and memory access patterns, rendering traditional partitioning approaches suboptimal. Consequently, to ensure the sustained relevance of nnWeb in the face of evolving model architectures, there exists a critical need for novel partitioning strategies specifically designed to address the distinctive requirements of modern attention-based architectures.

6. Related works

Deep learning in browser. Web browsers have emerged as a pivotal platform for the widespread adoption of deep learning (DL) technologies, offering significant advantages over traditional native applications. Unlike native applications, which require installation and

upgrades, web browsers eliminate these barriers, lowering the threshold for end-users and accelerating the deployment of DL services. Moreover, in-browser inference avoids transmitting user data to remote servers, thereby preserving user privacy and attracting substantial research attention [15,34–36]. In-browser inference has empowered various innovative functionalities in web applications. Examples include facial contour analysis for optimized online meetings, face detection, age estimation based on facial images, and music generation according to user interactions [37]. These applications highlight the potential of in-browser inference to enhance user experiences across diverse domains.

However, despite these advantages, in-browser inference faces notable performance challenges. Studies indicate that compared to local platforms, in-browser inference exhibits significantly higher latency on both CPU and GPU, averaging 16.9× and 4.9× higher, respectively. Additionally, the memory requirements for in-browser inference can be extremely demanding, sometimes exceeding the memory footprint of the DL models themselves by 334.6×, which brings a significant energy cost.

To address these limitations, researchers have explored various optimization strategies to enhance the performance and efficiency of in-browser inference. Neural network compression techniques, such as pruning, quantization, and distillation, have been employed to reduce large-scale models into smaller, more efficient versions suitable for edge devices. Furthermore, the integration of hardware accelerators like GPUs, ASICs, and FPGAs has bolstered cloud computing capabilities. Despite these advancements, existing techniques still fall short of fully meeting the stringent requirements for inference accuracy and efficiency in browser platforms.

In addition to the work on the utilization and efficiency of data center systems [38–44], various algorithm optimization techniques have been proposed. For instance, neural network compression methods [45,46] prune large-scale neural network models into smaller ones with lower computational demands, effectively enabling deployment on edge devices. Furthermore, to enhance cloud computing capabilities, a variety of accelerators such as GPUs, ASICs, and FPGAs have been introduced for data centers [47]. However, these techniques are unlikely to address the direct needs of all existing browser platforms, particularly the challenge of accurate and efficient model inference.

Computation offloading. Computation offloading has emerged as a critical strategy for enhancing the performance of resource-constrained devices by leveraging remote computational resources. This technique involves transferring computationally intensive tasks from edge devices to powerful servers or cloud platforms, thereby reducing local

processing burdens and improving overall system efficiency. However, the effectiveness of offloading is heavily influenced by factors such as network latency, bandwidth constraints, and dynamic resource availability. Among these, transmission latency stands out as one of the most significant factors affecting computation offloading and serves as a core metric for evaluating network quality of service [48].

The primary objectives of computation offloading are to minimize end-to-end latency, reduce energy consumption, and optimize resource utilization. For latency-sensitive applications such as real-time object detection or speech recognition, offloading can significantly reduce inference time by balancing local and remote computations [49,50]. Energy efficiency is also a critical concern for battery-powered devices, as offloading energy-intensive tasks to remote servers can extend operational lifetimes [51].

Offloading strategies can be broadly categorized into two types: full offloading and partial offloading. Full offloading transfers the entire task to a remote server, which is most suitable when data transmission costs are negligible compared to local processing delays [52]. Partial offloading is a strategy that partitions computational tasks between edge devices and remote servers, aiming to optimize both communication and computation costs [53]. Various algorithms have been proposed to enhance application performance through computation offloading, with partial offloading being particularly effective in scenarios where balancing local and remote resources is critical [54–57]. A notable example of this approach is Neurosurgeon, a system developed by Kang et al. [18]. Neurosurgeon demonstrates the practical implementation of partial offloading by automatically dividing neural network models between mobile devices and servers at the granularity of individual layers. This method not only reduces communication overhead but also ensures efficient utilization of computational resources on both ends, highlighting the potential of partial offloading in optimizing performance for complex applications.

7. Conclusion

In this paper, we introduce nnWeb, a comprehensive framework designed to expedite browser-based deep neural network (DNN) inference via synergistic client-server execution. By constructing precise, layer-level performance and energy models through offline profiling and harnessing WebGPU's asynchronous APIs to overlap initialization and computation, nnWeb fundamentally reshapes the latency profile of web inference. The framework dynamically adjusts to fluctuating network conditions and heterogeneous device capabilities, achieving up to 30% latency reduction on cloud servers and up to 52% in browsers, while decreasing overall system energy consumption by 11.29% to 43.98%. nnWeb exhibits robust performance across diverse application domains—including computer vision, natural language processing, and speech recognition. Future work will investigate multi-objective optimization, the integration of reinforcement learning for continuous adaptation, and support for non-sequential model architectures to further extend nnWeb's versatility. Collectively, nnWeb lays a solid foundation for the efficient deployment of AI applications in heterogeneous environments.

CRediT authorship contribution statement

Jiawei Liu: Writing – review & editing, Writing – original draft, Validation, Methodology. **Bing Dong:** Writing – original draft, Investigation. **Weilong Wang:** Visualization, Validation, Formal analysis. **Muhan Yuan:** Investigation, Data curation. **Borui Li:** Supervision, Conceptualization. **Zhao-Dong Xu:** Supervision, Resources. **Shuai Wang:** Supervision, Resources, Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We thank all the reviewers for their valuable comments. This work is partially supported by the National Natural Science Foundation of China (Grants No. 62272098, 62302096, and U24B20152), Central University Basic Research Fund of China (Grants No. 2242024K40019), Natural Science Foundation of Jiangsu Province (Grants No. BK2023 0813), Zhishan Young Scholar Program of Southeast University (Grants No. 2242024RCB0041), and the Major Project of Fundamental Research on Frontier Leading Technology of Jiangsu Province (Grants No. BK20222006).

Data availability

Data will be made available on request.

References

- [1] M. Hidaka, Y. Kikura, Y. Ushiku, T. Harada, WebDNN: Fastest DNN execution framework on web browser, in: Proceedings of the 25th ACM International Conference on Multimedia, in: MM '17, Association for Computing Machinery, New York, NY, USA, 2017, pp. 1213–1216, <http://dx.doi.org/10.1145/3123266.3129394>.
- [2] H.-A. Goh, C.-K. Ho, F.S. Abas, Front-end deep learning web apps development and deployment: A review, *Appl. Intell.* 53 (12) (2023) 15923–15945, <http://dx.doi.org/10.1007/s10489-022-04278-6>.
- [3] K. Sanderson, AI science search engines are exploding in number — Are they any good? *Nature* 616 (7958) (2023) 639–640, <http://dx.doi.org/10.1038/d41586-023-01273-w>.
- [4] Y. Zhang, Browser-oriented universal cross-site recommendation and explanation based on user browsing logs, in: Proceedings of the 8th ACM Conference on Recommender Systems, in: RecSys '14, Association for Computing Machinery, New York, NY, USA, 2014, pp. 433–436, <http://dx.doi.org/10.1145/2645710.2653367>.
- [5] T. Jiang, W. Li, J. Wang, X. Wang, Using artificial intelligence-based online translation website to improve the health education in international students, in: 2021 2nd International Conference on Artificial Intelligence and Education (ICAIE), 2021, pp. 25–28, <http://dx.doi.org/10.1109/ICAIE53562.2021.00012>.
- [6] J. Wen, Z. Chen, X. Jin, X. Liu, Rise of the planet of serverless computing: A systematic review, *ACM Trans. Softw. Eng. Methodol.* 32 (5) (2023) 131:1–131:61, <http://dx.doi.org/10.1145/3579643>.
- [7] D. Gu, Y. Zhao, Y. Zhong, Y. Xiong, Z. Han, P. Cheng, F. Yang, G. Huang, X. Jin, X. Liu, ElasticFlow: An elastic serverless training platform for distributed deep learning, in: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, in: ASPLOS 2023, Association for Computing Machinery, New York, NY, USA, 2023, pp. 266–280, <http://dx.doi.org/10.1145/3575693.3575721>.
- [8] N.F. Syed, M. Ge, Z. Baig, Fog-cloud based intrusion detection system using recurrent neural networks and feature selection for IoT networks, *Comput. Netw.* 225 (2023) 109662, <http://dx.doi.org/10.1016/j.comnet.2023.109662>.
- [9] K. Wang, C.-M. Chen, M.S. Hossain, G. Muhammad, S. Kumar, S. Kumari, Transfer reinforcement learning-based road object detection in next generation IoT domain, *Comput. Netw.* 193 (2021) 108078, <http://dx.doi.org/10.1016/j.comnet.2021.108078>.
- [10] Y. Jiang, S. Wang, V. Valls, B.J. Ko, W.-H. Lee, K.K. Leung, L. Tassiulas, Model pruning enables efficient federated learning on edge devices, *IEEE Trans. Neural Netw. Learn. Syst.* 34 (12) (2023) 10374–10386, <http://dx.doi.org/10.1109/TNNLS.2022.3166101>.
- [11] W.-x. Liu, D.-x. Chen, M.-q. Tan, K.-y. Chen, Y. Yin, W.-L. Shang, J. Li, J. Cai, Model parameter prediction method for accelerating distributed DNN training, *Comput. Netw.* 255 (2024) 110883, <http://dx.doi.org/10.1016/j.comnet.2024.110883>.
- [12] J.C. Borromeo, K. Kondepu, N. Andriolli, L. Valcarenghi, FPGA-accelerated SmartNIC for supporting 5G virtualized radio access network, *Comput. Netw.* 210 (2022) 108931, <http://dx.doi.org/10.1016/j.comnet.2022.108931>.
- [13] S. Chickerur, S. Balannavar, P. Hongekar, A. Prerna, S. Jituri, WebGL vs. WebGPU: A performance analysis for web 3.0, *Procedia Comput. Sci.* 233 (2024) 919–928, <http://dx.doi.org/10.1016/j.procs.2024.03.281>.
- [14] Y. Huang, X. Qiao, S. Dustdar, Y. Li, AoDNN: An auto-offloading approach to optimize deep inference for fostering mobile web, in: IEEE INFOCOM 2022 - IEEE Conference on Computer Communications, 2022, pp. 2198–2207, <http://dx.doi.org/10.1109/INFOCOM48880.2022.9796763>.

- [15] F. Jia, S. Jiang, T. Cao, W. Cui, T. Xia, X. Cao, Y. Li, Q. Wang, D. Zhang, J. Ren, Y. Liu, L. Qiu, M. Yang, Empowering in-browser deep learning inference on edge through just-in-time kernel optimization, in: Proceedings of the 22nd Annual International Conference on Mobile Systems, Applications and Services, in: MOBISYS '24, Association for Computing Machinery, New York, NY, USA, 2024, pp. 438–450, <http://dx.doi.org/10.1145/3643832.3661892>.
- [16] M. Salazar, D. Bandara, G. Bejarano, Evaluating model compression techniques for on-device browser sign language recognition inference, in: 2024 IEEE XXXI International Conference on Electronics, Electrical Engineering and Computing (INTERCON), 2024, pp. 1–7, <http://dx.doi.org/10.1109/INTERCON63140.2024.10833501>.
- [17] GPU for the web community group, 2017, <https://www.w3.org/community/gpu/>.
- [18] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, L. Tang, Neurosurgeon: Collaborative intelligence between the cloud and mobile edge, SIGARCH Comput. Arch. News 45 (1) (2017) 615–629, <http://dx.doi.org/10.1145/3093337.3037698>.
- [19] Z. Chen, Y. Ma, H. Shen, M. Liu, Weinfer: Unleashing the power of WebGPU on LLM inference in web browsers, in: The Web Conference 2025, 2025.
- [20] A.G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, H. Adam, MobileNets: Efficient convolutional neural networks for mobile vision applications, 2017, <http://dx.doi.org/10.48550/arXiv.1704.04861>, arXiv:1704.04861.
- [21] J. Shen, F. Liu, Y. Tu, C. Tang, Finding gene network topologies for given biological function with recurrent neural network, Nat. Commun. 12 (1) (2021) 3125, <http://dx.doi.org/10.1038/s41467-021-23420-5>.
- [22] J. Wang, K. Liu, H. Li, LSTM-based graph attention network for vehicle trajectory prediction, Comput. Netw. 248 (2024) 110477, <http://dx.doi.org/10.1016/j.comnet.2024.110477>.
- [23] E. Mushtaq, A. Zameer, R. Nasir, Knacks of a hybrid anomaly detection model using deep auto-encoder driven gated recurrent unit, Comput. Netw. 226 (2023) 109681, <http://dx.doi.org/10.1016/j.comnet.2023.109681>.
- [24] Y. Akhauri, M. Abdelfattah, On latency predictors for neural architecture search, Proc. Mach. Learn. Syst. 6 (2024) 512–523.
- [25] T. Guo, Cloud-based or on-device: An empirical study of mobile deep inference, in: 2018 IEEE International Conference on Cloud Engineering (IC2E), 2018, pp. 184–190, <http://dx.doi.org/10.1109/IC2E.2018.00042>.
- [26] K. Kojas, A survey of serverless machine learning model inference, 2023, <http://dx.doi.org/10.48550/arXiv.2311.13587>, arXiv:2311.13587.
- [27] TensorFlow.js Models, <https://www.tensorflow.org/js/models>.
- [28] M.I. Rochman, V. Sathya, D. Fernandez, N. Nunez, A.S. Ibrahim, W. Payne, M. Ghosh, A comprehensive analysis of the coverage and performance of 4G and 5G deployments, Comput. Netw. 237 (2023) 110060, <http://dx.doi.org/10.1016/j.comnet.2023.110060>.
- [29] L.K. Tartibu, Multi-Objective Optimization Techniques in Engineering Applications: Advanced Methods for Solving Complex Engineering Problems, in: Studies in Computational Intelligence, vol. 1184, Springer Nature Switzerland, Cham, 2025, <http://dx.doi.org/10.1007/978-3-031-81237-8>.
- [30] J. Wang, H. Xia, L. Xu, R. Zhang, K. Jia, DRL-based latency-energy offloading optimization strategy in wireless VR networks with edge computing, Comput. Netw. 258 (2025) 111034, <http://dx.doi.org/10.1016/j.comnet.2025.111034>.
- [31] S. Choi, S. Choi, G. Lee, S.-G. Yoon, S. Bahk, Deep reinforcement learning for scalable dynamic bandwidth allocation in RAN slicing with highly mobile users, IEEE Trans. Veh. Technol. 73 (1) (2024) 576–590, <http://dx.doi.org/10.1109/TVT.2023.3302416>.
- [32] J. Wu, C. Huang, H. Huang, C. Lv, Y. Wang, F.-Y. Wang, Recent advances in reinforcement learning-based autonomous driving behavior planning: A survey, Transp. Res. Part C: Emerg. Technol. 164 (2024) 104654, <http://dx.doi.org/10.1016/j.trc.2024.104654>.
- [33] X. Qin, W. Zhu, Q. Hu, Z. Zhou, Y. Ding, X. Gao, R. Gu, DenseNet-transformer: A deep learning method for spatial-temporal traffic prediction in optical fronthaul network, Comput. Netw. 253 (2024) 110674, <http://dx.doi.org/10.1016/j.comnet.2024.110674>.
- [34] L. Quan, Q. Guo, X. Xie, S. Chen, X. Li, Y. Liu, Towards understanding the faults of JavaScript-based deep learning systems, in: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, in: ASE '22, Association for Computing Machinery, New York, NY, USA, 2023, pp. 1–13, <http://dx.doi.org/10.1145/3551349.3560427>.
- [35] D. Tian, H. Shen, Y. Ma, Parallelizing DNN inference in mobile web browsers on heterogeneous hardware, in: Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services, in: MobiSys '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 519–520, <http://dx.doi.org/10.1145/3498361.3538763>.
- [36] Q. Zhang, X. Li, X. Che, X. Ma, A. Zhou, M. Xu, S. Wang, Y. Ma, X. Liu, A comprehensive benchmark of deep learning libraries on mobile devices, in: Proceedings of the ACM Web Conference 2022, in: WWW '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 3298–3307, <http://dx.doi.org/10.1145/3485447.3512148>.
- [37] Q. Wang, S. Jiang, Z. Chen, X. Cao, Y. Li, A. Li, Y. Ma, T. Cao, X. Liu, Anatomizing deep learning inference in web browsers, ACM Trans. Softw. Eng. Methodol. 34 (2) (2025) 47:1–47:43, <http://dx.doi.org/10.1145/3688843>.
- [38] C.-H. Hsu, Y. Zhang, M.A. Laurenzano, D. Meisner, T. Wenisch, J. Mars, L. Tang, R.G. Dreslinski, Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting, in: 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), 2015, pp. 271–282, <http://dx.doi.org/10.1109/HPCA.2015.7056039>.
- [39] J. Hauswald, M.A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R.G. Dreslinski, T. Mudge, V. Petrucci, L. Tang, J. Mars, Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers, in: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, in: ASPLOS '15, Association for Computing Machinery, New York, NY, USA, 2015, pp. 223–238, <http://dx.doi.org/10.1145/2694344.2694347>.
- [40] Q. Chen, H. Yang, J. Mars, L. Tang, Baymax: QoS awareness and increased utilization for non-preemptive accelerators in warehouse scale computers, SIGPLAN Not. 51 (4) (2016) 681–696, <http://dx.doi.org/10.1145/2954679.2872368>.
- [41] Y. Zhang, D. Meisner, J. Mars, L. Tang, Treadmill: Attributing the source of tail latency through precise load testing and statistical inference, SIGARCH Comput. Arch. News 44 (3) (2016) 456–468, <http://dx.doi.org/10.1145/3007787.3001186>.
- [42] A. Jain, M.A. Laurenzano, L. Tang, J. Mars, Continuous shape shifting: Enabling loop co-optimization via near-free dynamic code rewriting, in: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2016, pp. 1–12, <http://dx.doi.org/10.1109/MICRO.2016.7783726>.
- [43] M.A. Laurenzano, Y. Zhang, J. Chen, L. Tang, J. Mars, PowerChop: Identifying and managing non-critical units in hybrid processor architectures, SIGARCH Comput. Arch. News 44 (3) (2016) 140–152, <http://dx.doi.org/10.1145/3007787.3001152>.
- [44] V. Petrucci, M.A. Laurenzano, J. Doherty, Y. Zhang, D. Mossé, J. Mars, L. Tang, Octopus-man: QoS-driven task management for heterogeneous multicores in warehouse-scale computers, in: 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), 2015, pp. 246–258, <http://dx.doi.org/10.1109/HPCA.2015.7056037>.
- [45] T. Zhang, S. Ye, K. Zhang, J. Tang, W. Wen, M. Fardad, Y. Wang, A systematic DNN weight pruning framework using alternating direction method of multipliers, in: Proceedings of the European Conference on Computer Vision (ECCV), 2018, pp. 184–199.
- [46] X. Xie, K.-H. Kim, Source compression with bounded DNN perception loss for IoT edge computer vision, in: The 25th Annual International Conference on Mobile Computing and Networking, in: MobiCom '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 1–16, <http://dx.doi.org/10.1145/3300061.3345448>.
- [47] J. Hauswald, Y. Kang, M.A. Laurenzano, Q. Chen, C. Li, T. Mudge, R.G. Dreslinski, J. Mars, L. Tang, DiINN and tonic: DNN as a service and its implications for future warehouse scale computers, SIGARCH Comput. Arch. News 43 (3S) (2015) 27–40, <http://dx.doi.org/10.1145/2872887.2749472>.
- [48] X. Fan, X. Deng, Y. Xia, L. Yi, L.T. Yang, C. Zhu, Tensor-based confident information coverage reliability of hybrid internet of things, IEEE Trans. Mob. Comput. 23 (3) (2024) 2171–2185, <http://dx.doi.org/10.1109/TMC.2023.3250508>.
- [49] F. Wang, M. Zhu, M. Wang, M.R. Khosravi, Q. Ni, S. Yu, L. Qi, 6G-enabled short-term forecasting for large-scale traffic flow in massive IoT based on time-aware locality-sensitive hashing, IEEE Internet Things J. 8 (7) (2021) 5321–5331, <http://dx.doi.org/10.1109/JIOT.2020.3037669>.
- [50] F. Sufyan, A. Banerjee, Computation offloading for distributed mobile edge computing network: A multiobjective approach, IEEE Access 8 (2020) 149915–149930, <http://dx.doi.org/10.1109/ACCESS.2020.3016046>.
- [51] J. Zhao, W. Li, H. Liu, P. Yu, H. Li, Z. Wen, An enhanced genetic algorithm for computation task offloading in MEC scenario, Int. J. Wirel. Mob. Comput. 25 (2) (2023) 118–127, <http://dx.doi.org/10.1504/IJWMC.2023.133059>.
- [52] F. Sufyan, A. Banerjee, Comparative analysis of network libraries for offloading efficiency in mobile cloud environment, Int. J. Adv. Comput. Sci. Appl. 10 (2019) <http://dx.doi.org/10.14569/IJACSA.2019.0100272>.
- [53] S. Khan, Z. Jiangbin, M. Irfan, F. Ullah, S. Khan, An expert system for hybrid edge to cloud computational offloading in heterogeneous MEC-MCC environments, J. Netw. Comput. Appl. 225 (2024) 103867, <http://dx.doi.org/10.1016/j.jnca.2024.103867>.
- [54] B. Li, W. Dong, Edge-centric programming for IoT applications with automatic code partitioning, IEEE Trans. Comput. 71 (10) (2022) 2408–2422, <http://dx.doi.org/10.1109/TC.2021.3129367>.
- [55] B. Li, W. Dong, EdgeProg: Edge-centric programming for IoT applications, in: 2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS), 2020, pp. 212–222, <http://dx.doi.org/10.1109/ICDCS47774.2020.00038>.
- [56] B. Li, W. Dong, Y. Gao, WiProg: A WebAssembly-based approach to integrated IoT programming, in: IEEE INFOCOM 2021 - IEEE Conference on Computer Communications, 2021, pp. 1–10, <http://dx.doi.org/10.1109/INFOCOM42981.2021.9488424>.
- [57] B. Li, H. Fan, Y. Gao, W. Dong, Bringing webassembly to resource-constrained IoT devices for seamless device-cloud integration, in: Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services, in: MobiSys '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 261–272, <http://dx.doi.org/10.1145/3498361.3538922>.