CacheInf: Collaborative Edge-Cloud Cache System for Efficient Robotic Visual Model Inference

Anonymous Author(s)
Submission Id: 841

Abstract

Visual information processing is crucial for mobile robots performing tasks such as navigation, manipulation, and human-robot interaction. However, limited computational power for local computation and unstable wireless network bandwidth for computation offloading to a GPU server on these robots lead to slow visual model inference, hindering real-time responsiveness and increasing energy consumption. Existing caching mechanisms designed for fixed edge devices reduce computation by reusing cached activations (computation intermediates) by aligning the activations to the detected movements on the input images, but they face challenges on mobile robots due to frequent camera perspective changes that accumulates error when reusing the cached activation, which leads to degraded inference accuracy or frequent discarding and recomputing the cached activations that increases latency.

We propose CacheInf, a high-performance edge-cloud caching system for efficient visual model inference on mobile robots. Instead of wholly reusing or recomputing the activations, CacheInf selectively reuses a portion of the cached activations frame and recomputes the others, which minimizes required computation, accelerates both local processing and computation offloading, and mitigates error accumulation in activations, achieving an optimal trade-off between inference accuracy and speed. Evaluation on various visual models and wireless network environments shows CacheInf reduced end-to-end inference latency by up to 74.26% and reduced average energy consumption for inference by up to 81.21% compared with the baselines.

Keywords: Edge computing

1 Introduction

Visual information is vital for various robotic tasks deployed on mobile edge devices (mobile robots), such as navigation [15], manipulation [3], and human-robot interaction [25]; and as a major visual information processing method, fast, accurate and energy-efficient visual model inference (typically convolution neural networks [10, 19, 24], CNN) is important for the robotic tasks to timely respond to environment changes. Unfortunately, the mobile robot typically has limited computational power and limited and unstable wireless network bandwidth [28], which slow down the inference speed via both local computation (complete all inference calculations on the edge device itself) and computation offloading (offload the calculations to remote GPU servers). Thus the mobile robot often suffers the problem of slow visual model inference that interferes the robotic task performance (e.g., 1.01 seconds in human pose estimation [10] or 0.62 seconds in surrounding occlusion prediction [22]) and the prolonged inference latency naturally increases energy consumption each inference.

To address this problem, we seek enlightenment from previous work targeting fixed-position edge devices (e.g., smart cameras) that employ caching mechanisms [4, 5, 7] to reduce the computational burden of visual model inference. They are based on the fact that the visual models extensively use operators (e.g., convolution) whose computation results (i.e. activations) are spatially correlated to the input image: the value of each pixel on the activations is dominated by a block of the input image (i.e., receptive field) determined by the model architecture [4]. Given a continuous stream of images, these methods identify the movements of the receptive fields and interpolate (reuse) the cached activations of the corresponding operator, eliminating the need for computing activations for all related operators (including both the current and previous ones). When the movement recognition fails, a full local computation (recompute) is performed on the input image to obtain the latest activations by recomputing all the operators.

Introducing a caching mechanism that reuses previous activations for visual model inference on mobile robots has the potential to eliminate activation computation and reduce data transmission volume in computation offloading (since only the receptive fields with recognized movements need to be recomputed and transmitted); however, the frequent camera perspective movements [5, 7] inherent to mobile robots create a ***dilemma*** between inference accuracy and inference speed when applying such caching mechanisms. Changes in camera perspective typically bring new scenes or changed occlusion into the images, which cannot be covered by receptive field movements (recognition failure). To cope with this situation, the above methods can only either ignore certain recognition failure at the cost of severely degraded inference accuracy (25.89% lower in [7]), or frequently execute full local computation or full transmission of the input images during computation offloading, sacrificing inference speed. Consequently, existing caching mechanisms struggle to strike a balance between inference accuracy and speed,

regardless of whether they employ local computation or computation offloading.

The key reason for the dilemma faced by existing caching methods is that they involve deciding whether to reuse or recompute the activations at the granularity of the whole input frame, which lacks a tradeoff space between inference accuracy and speed. When continuously reusing activations computed with a reference frame, the difference between the new frames and the reference frame accumulates, causing either degraded accuracy or triggering a full recomputation to eliminate errors in activations. However, with the ability to tradeoff between accuracy and inference speed (for example, partially reusing and partially recomputing the activations), we can selectively compute on the blocks with the most significant differences while reusing the computation results of the others, mitigating the accumulation of errors while accelerating both local computation and computation offloading via caching.

To bridge this gap, in this paper, we propose CacheInf, a high-performance collaborative edge-cloud cache system for efficient robotic visual model. Given a continuous stream of visual input in a robotic visual task, CacheInf selectively reuses a portion of the cached activations of a reference frame while recomputes the rest based on statistical metrics such as the mean square error of pixels in corresponding receptive fields between the reference frame and the current frame. In this way, the amount of required computation is minimized which accelerates both local computation and computation offloading and the error accumulation in activations is mitigated, achieving the optimal tradeoff between inference accuracy and inference speed for collaborative edge-cloud visual model inference on the mobile robot.

The first challenge of the the design of CacheInf is to minimize computation using cache mechanism without compromising the inference accuracy. While prioritizing recomputation on receptive fields with most appearance difference seems effective, the background also gradually changes in appearance along a video stream with perspective changes, which attracts wasteful recomputation on these areas. We observe that a moving targeted object tend to have different movements compared with the background (otherwise it is stationary and the corresponding cached activations can be directly reused). Thus, instead of appearance-based metrics, we analyze the movements of the sub-pixel blocks of a receptive field (internal movements) and prioritize the recomputation on receive fields with highest variance in internal movements, reducing unnecessary computation while maintaining high inference accuracy.

The second challenge is to minimize the overall inference latency considering the interaction between the mobile robot and the GPU server. One major obstacle is that switching between local computation and computation offloading under the caching mechanism (e.g., when wireless network bandwidth changes) typically requires costly one full local recomputation or full transmission of the input image, and such cost hinders a greedy scheduling method from adopting such switching to gain further overall acceleration. To overcome this issue, we schedule between local computation and computation offloading by looking ahead several steps with the predicted future wireless network bandwidth and possible computation reduction based on previous records, to minimize the overall inference latency.

We implement CacheInf based on python and pytorchh [13] integrated with self-implemented C++ CUDA extensions [1]. For the tail cases where we need to fully offload an input image to the GPU server, we integrate a state-of-the-art offloading method called Hybrid-Parallel (HP) [18] which mitigates the caused latency. Our baselines include HP, a state-of-the-art cache-based computation reduction methods called EVA2 [4], EVA2 integrated with HP and local computation. We evaluated CacheInf on a four-wheel robot equipped with a Jetson NX Xavier [11] that is capable of computing locally with its low-power-consumption GPU. The offloading GPU server is a PC equipped with an Nvidia 2080ti GPU. Our datasets include the standard datasets of video frames of DAVIS [14] captured by a handheld camera and our self-captured video frames using sensors on our robot. Extensive evaluation over various visual models [10, 20, 22] and wireless network bandwidth circumstances shows that:

- CacheInf is fast and accurate. Among the baselines that maintains high accuracy, CacheInf reduced the end-to-end inference time by 34.29% to 74.26% with only 3.43% to 4.12% accuracy reduction.
- CacheInf saves energy. Among the baselines that maintains high accuracy, CacheInf reduced the average energy consumed to complete inference on each image by 23.89% to 81.21%.
- CacheInf is easy to use. The CacheInf pipeline can be a drop-in integration to the existing CNN-based visual models to accelerate their inference.

The contribution of this paper is twofold: 1. a new caching mechanism for visual model inference on mobile devices which selectively reuse and recompute fractions of cached activations to best tradeoff between inference accuracy and computation reduction; 2. a scheduling mechanism designed to optimize the overall inference latency considering the interaction between the edge (the mobile robot) and the cloud (the GPU server to offload computation to) And the resulting system, CacheInf, optimally reduces visual model inference latency and energy consumption on the mobile robot. The accelerated visual model inference and the reduced power consumption will make real-world robots more performant on various robotic tasks and nurture more visual models to be deployed in real-world robots. The source code and evaluation logs of CacheInf is available at https://github.come/eurosys25paper841/CacheInf.

The rest of this paper is organized as follows. Chapter two introduces background and related work. Chapter three gives an overview of CacheInf and Chapter four presents its detailed design. Chapter five describes the implementation. Chapter six presents our evaluation results and Chapter seven concludes.

2 Background

2.1 Vision tasks on robots

Vision tasks play a crucial role in enabling robots to perceive, understand, and interact with the environment. Visual information is essential for various robotic tasks, such as object recognition [6], navigation [15], manipulation [3], and human-robot interaction [25]. The rapid advancements in machine learning, particularly deep learning, have revolutionized the field of computer vision and have been widely adopted in robotic applications, which form the foundation for many high-level robotic tasks.

However, the deployment of visual models on resource-constrained robots poses significant challenges. Visual models often require significant computational resources and memory, which may not be readily available on robots, especially in mobile and embedded systems. Furthermore, real-time performance is critical for many robotic tasks, as robots need to process and respond to visual information quickly to ensure safe and effective operation. Therefore, fast visual model inference becomes a key requirement for the successful deployment of deep learning models in robotic applications. Addressing these challenges is essential for enabling robots to effectively perceive, understand, and interact with their environment in real time, paving the way for more intelligent and autonomous robotic systems.

2.2 Visual Models

Convolutional layers [12] have become a fundamental building block in visual models, leading to significant breakthroughs in various computer vision tasks. Inspired by the biological structure of the visual cortex [21], these layers apply learnable filters to the input image, performing convolution operations to produce feature maps that highlight the presence of specific patterns at different spatial locations (i.e., local operators). This enables deep learning models to capture translation-invariant features and learn hierarchical representations [9], with early layers learning low-level features like edges and corners, and deeper layers learning more complex patterns and object parts. As a result, deep convolutional neural networks (CNNs) have achieved state-of-the-art performance in various vision applications, image classification [16], object detection [6], and semantic segmentation [23], due to their ability to effectively capture and learn spatial hierarchies of features from raw input images. As the field of computer vision continues to evolve, convolutional layers are expected to remain a crucial component in the development of advanced models for understanding and analyzing visual data.

2.3 Resource Limitations of Robots

To demonstrate the instability of wireless transmission in real-world situations, we conducted a robot surveillance experiment using four-wheel robots navigating around several given points at 5-40cm/s speed in our lab (indoors) and campus garden (outdoors), with hardware and wireless network settings as described in Sec. 6.6. We saturated the wireless network connection with iperf [2] and recorded the average bandwidth capacity between these robots every 0.1s for 5 minutes.

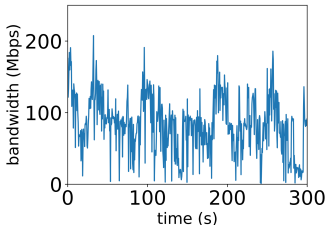


Figure 1. The instability of wireless transmission between our robot and a base station in robotic IoT networks.

The results in Fig. 1 show average bandwidth capacities of 93 Mbps and 73 Mbps for indoor and outdoor scenarios, respectively. The outdoor environment exhibited higher instability, with bandwidth frequently dropping to extremely low values around 0 Mbps, due to the lack of walls to reflect wireless signals and the presence of obstacles like trees between communicating robots, resulting in fewer received signals compared to indoor environments. This limitation on the wireless network bandwidth on the robot poses significant challenges for the efficient and reliable computation offloading of robots in real-world scenarios, particularly in outdoor environments where the instability of wireless networks is more pronounced.

2.4 Related Work

Edge-Cloud Collaborative Inference expedites the overall inference process by leveraging a GPU server to handle a portion of the computational workload of the robot. The DSCCS approach [8] views the visual model in a layer-wise perspective and focuses on model-layer-level scheduling (layer partitioning) for rapid inference; Hybrid-Parallel [18] further offers a more fine-grained control by partitioning and scheduling the computation within local operators, so that the robot can compute on a portion of the input on local operators while at the same time transmitting the result of the input to the GPU server. It enhances parallelism and further accelerates inference. However, despite the advancements in these offloading techniques, the limited bandwidth still poses a bottleneck for data transmission, which our caching

mechanism effectively mitigates and achieves a significant improvement in inference performance.

Existing caching methods (e.g., EVA2 [5], DeepCache [27]) designed for real-time vision tasks on edge devices leverage temporal redundancy in video streams to avoid unnecessary computation on most frames. EVA2 proposes a new algorithm called activation motion compensation, which adapts to visual changes by detecting changes in the visual input and incrementally updating previously computed activations. DeepCache, on the other hand, discovers temporal locality at the input of the model by exploiting the video's internal structure and propagates regions of reusable results into the model's internal structure. However, despite their achievements in accelerating inference, these methods face a trade-off between inference accuracy and speed. They must either compromise on accuracy by ignoring certain recognition failures or sacrifice inference speed by frequently executing full local computation or transmitting entire input images during computation offloading. Moreover, these algorithms are designed for fixed-position devices with a stationary perspective (e.g., smart cameras), where the primary changes in the input images are due to object movement (e.g., people, vehicles). In contrast, mobile robot scenarios introduce additional challenges as the background also changes due to perspective shifts, leading to more frequent triggering of full local computation in these methods and further exacerbating the challenge of striking an optimal balance between accuracy and speed.

3 System Overview

3.1 Working Environment

We assume that the working environment of CacheInf is a mobile robot performing robotic tasks in the real world which requires seamless real-time visual model inference on the continuous image stream captured from the on-board camera, to achieve real-time response to various environment changes. The robot itself is equipped with low-power-consumption gpu to perform slow and comparatively high energy consumption local visual model inference so as to guarantee its worst case performance; it has wireless network access to a remote powerful GPU server that provides opportunities of acceleration via computation offloading, but the connection suffers from limited and unstable wireless network bandwidth.

3.2 Architecture of CacheInf

CacheInf consists of four major components: Collaborative Offload Scheduler, Cache Analyzer, Cache Combiner, and Recomputation Input Constructor. Figure 2 describes the runtime workflow of CacheInf. Collaborative Offload Scheduler functions both at the initialization stage and at runtime and we exclude the initialization stage in Figure 2 for simplicity. For simplicity, we call the Recomputation Input Constructor and Cache Combiner for activations together with the visual model inference pipeline as the CacheInf Executor which resides on both the robot and the GPU server.

3.2.1 Collaborative Offload Scheduler. During the initialization stage of the robotic task and CacheInf is granted access to a visual model and an initial input image. Collaborative Offload Scheduler first profiles the visual model based on this initial input and its execution statistics (e.g., execution time of each operator and the receptive field) on both the robot and the server. Then it finds the operator involved in the visual model that should cache its computation results (activations) and computes for an optimized computation and offloading plan for each possible situation including different wireless network bandwidth and different ratios of recomputation. To operator to cache its computation results is typically the last operator that preserves the spatial information of the input images (i.e., convolution, pooling, element-wise addition/subtraction, etc), which is typically the operator before operators such as flatten and matrix multiplication.

At runtime Collaborative Offload Scheduler collects the running statistics such as wireless network bandwidth and the ratio of recomputation and at a fixed interval (e.g., several inference) rearrange the placement of computation (either locally compute or offload the computation to the GPU server) according to the precomputed schedule accordingly for optimal inference acceleration. The rearrangement can also be triggered when extreme conditions are detected, such as almost zero wireless network bandwidth or a full recomputation on the input image is required.

3.2.2 Cache Analyzer. Given a new input image, Cache Analyzer contrast it with the cached input image to find the movements of the receptive fields of the cached activations. Since the receptive fields of different pixels on the activations often overlaps as shown in Fig. 3, we first divide the whole input image into different small pixel blocks and calculate the movements of each pixel blocks (represented by motion vectors) via pixel block matching. The movements of the pixel blocks within a receptive field are then averaged as the movement of the receptive field. This process is similar to the motion detection process of the modern video encoding, but we decided not to leverage the existing video encoder to accelerate this process, since it incurs other calculation for video encoding that is unnecessary for our system and brings extra overhead. We also turn down other solutions in embedded systems [4] since they require specially designed hardwares. Instead, we leverage the onboard GPU where we implement a CUDA-accelerated pixel block matching algorithm (three-step block matching algorithm) to accelerate this process and the computation is lightweight (execution time estimated to be around 2 to 3 ms) compared with the visual model inference.

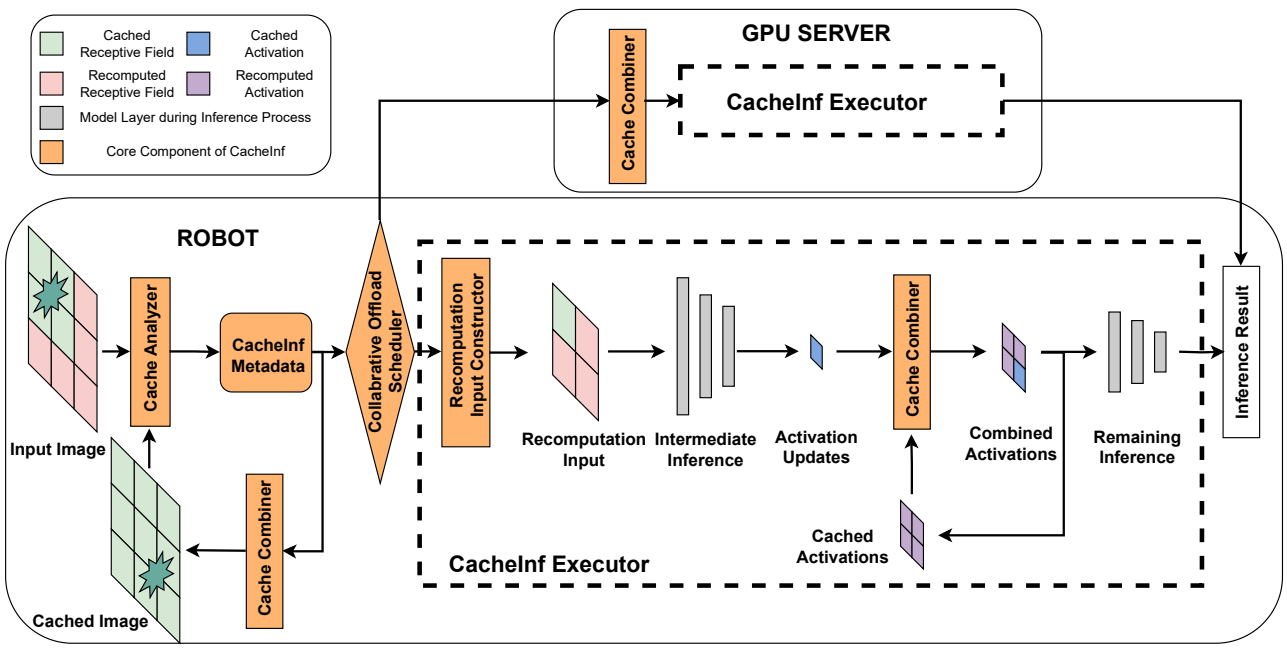


Figure 2. Architecture and workflow of CacheInf.

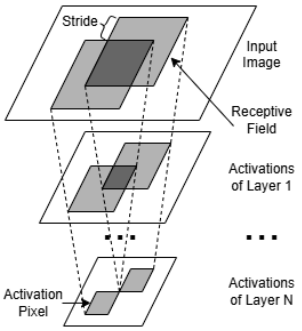


Figure 3. Receptive fields and their dominated pixels on the activations.

After the above process, we obtain the motion vectors of both the small pixel blocks on the input image and the receptive fields of the activations. During combining the motion vectors of small pixels blocks involved in a receptive field, we also compute the variance of the motion vectors within a receptive field (internal variance) as hints for selecting receptive fields to recompute. Besides, during the block matching process we also computed appearance-based metrics such as mean square error of pixel values on the input image and we use these metrics to sparsely select a small portion of pixels on the input image with the highest appearance difference which is then used to update the cached input image. These selected pixels (compensation pixels) are meant to limit the difference between the updated cached input image and the actual input image on the GPU server. All these data are aggregated and referred to as CacheInf metadata in Fig. 2 as the transmission. Note that these data are much smaller than the input image in size. For example, when the small pixel blocks is a square consisting of HXH pixels and the input image shape is MXN, the resulting number of motion vectors or variance of these small pixel blocks on the input image will be $\frac{M}{H}X\frac{N}{H}$. Together with the compensation pixels to update the cached input image, the possible transmission data volume to the GPU server is greatly reduced.

3.3 Cache Combiner

Cache Combiner resides both on the robot and the GPU server to maintain their individual cached input image and also the activations. Upon reception of the motion vectors of small pixel blocks and compensation pixels, Cache Combiner moves the pixel blocks on the cached input image according to the motion vectors and merge the compensation pixels into the moved image, so that the cached input image keeps tracks with the actual current input image in appearance. This is important for the subsequent inference based on CacheInf, because if the difference between the cached input image and the next input image is large, the appearance-based block matching algorithm used in Cache Analyzer will become less effective.

For the activations, the aggregated motion vectors from the receptive field for the activations, the activation matrix itself and the partially computed results on the receptive fields for the pixels on the activations can be equivalently

viewed as an image with motion vectors adn compensation pixels. Thus, we scale the the aggregated receptive field motion vectors to the size of the activations and then apply the above process on the activations to update the activations.

3.4 Recomputation Input Constructor

Recomputation Input Constructor resides both on the robot and the GPU server. When the Collaborative Offload Scheduler decides on the placement of computation, the Recomputation Input Constructor on the corresponding end (either robot or the server) collects pixels blocks on the cached input image updated by Cache Combiner to form the partial recomputation input. During construction, it prioritizes receptive fields with higher internal variance of the involved pixel blocks on the input image, which is computed in the previous stage. Specifically, Recomputation Input Constructor first selects the receptive field with highest internal variance, and then it repeatedly peeks the internal variance of the surrounding receptive fields of the selected and extends the selected area to cover these receptive fields if their internal variance is close to the selected one within a threshold. In this way, we constructed a dense and smaller input for the visual model to reduce computation burden.

4 Design

4.1 Cache Analyzer, Cache Combiner and Recomputation Input Constructor

As described above, Cache Analyzer extracts the motion vectors of small pixel blocks of the input image and combines (average) the the motion vectors of small pixel blocks involved in a receptive field as the motion vector for the receptive field of the cached activations.

$$\begin{aligned} \min_{i,j} \quad & \frac{(P_{k,l} - P'_{i+k,j+l})^2}{n} \\ s.t. \quad & 0 \le k \le M \\ & 0 \le l \le N \\ & 0 \le i+k \le M \\ & 0 \le j+l \le N \end{aligned} \tag{1}$$

The motion vectors can be computed as in Eq. 1 which is a block matching process that minimizes mean square error (MSE) between small pixels blocks in the cached input image and the new input image, where $P_{k,l}$ is a small pixel block indexed by (k, l) in the cached input image and $P'_{i+k,j+l}$ is a small pixel block on the new input image with offset as (i, j) and (i, j) is the motion vector of $P_{k,l}$. Both images are of size $M * N$ and each pixel block consist of n pixels.

When the above computation process finishes, we obtain the final minimized MSE of each small pixel blocks and high MSE indicates that a pixel block on the cached input image cannot finely match on the new input image, and thus we select such pixel blocks with MSE above a threshold as sparse update (compensation pixels) to the cached input image. Inversely, Cache Combiner moves pixel blocks on the cached image and the cached activations according to the motion vectors and merge the compensation pixels to the cached image and the cached activations.

The variance of motion vectors of small pixel blocks in a receptive field (internal variance) is computed as in Eq. 2

$$var_i = \frac{1}{m} \sum_j (MSE_{max} - MSE_j) \cdot (mv_j - \hat{mv}) \tag{2}$$

where the j indexed small pixel block resides in the receptive field indexed by i. MSE_{max} is the maximal mean square error of small pixel blocks within the receptive field and mv_j and $\hat{mv}$ are motion vectors indexed by j and the average motion vector of motion vectors in the receptive field. The internal variance of a receptive field describes how the pixel blocks in this receptive field diverges in movement which distinguishes moving objects instead of stationary background captured from a moving camera and the existence of new information. Note that the factor $(MSE_{max} - MSE_j)$ is used to filter out motion vectors caused by pixel blocks on the cached input image which cannot finely match on the new input image since these pixel blocks tend to generate random motion vectors.

With the calculated receptive field internal variance, Recomputation Input Constructor collects pixels on the updated cached input image from the receptive field with the highest internal variance and its surrounding receptive fields that meets a internal variance threshold. The resulting sub-image will be fed to the subsequent inference pipeline. Note that the inference process will be intercepted by Cache Combiner when the selected operator with its output activations cached, where Cache Combiner merge the computed partial activations back to the cached activations and then the subsequent calculation will be conducted on the full-sized updated cached activations.

4.2 Collaborative Offload Scheduler

We define the ratio of pixel blocks on the input image whose computed MSE is above a threshold (mismatched) as c and we define the computation time before the cached operator as T_r for the robot and T_s and the execution time of the rest operators as T'_r and T'_s. We suppose the size of the collected recomputation input R by Recomputation Input Constructor and the transmission data volume V of CacheInf metadata are both proportional to c, and the proportional counterparts are referred to as R_c and V_c. We assume T_r and T_s are proportional to the recomputation input size R_c. The major goal of Collaborative Offload Scheduler is to statically find an optimal placement of computation for a period of given times (n) of inference for each pair of wireless network bandwidth b, r and the current placement of computation p, supposing that the bandwidth and r will hold during during the period.

Algorithm 1: Collaborative Offload Schedule

Input: Wireless network bandwidth b; ratio of mismatched pixel blocks c; current computation placement p; consideration period n

Output: The static schedule of computation placement under b, c, p, *schedule*

1 $robot\_time = 0$
2 $server\_time = 0$
3 $r = R_c/R$
// First inference time cost
4 **if** p == *robot* **then**
5 $\quad server\_time = V/b + T_s + T_s'$
6 $\quad robot\_time = T_r * r + T_r'$
7 **end**
8 **else**
9 $\quad robot\_time = T_r + T_r'$
10 $\quad server\_time = V_c/b + T_s * r + T_s'$
11 **end**
// Subsequent inference time cost
12 $server\_time+ = (n-1) * (V_c/b + T_s * r + T_s')$
13 $robot\_time+ = (n-1) * (T_r * r + T_r')$
14 **if** $robot\_time > server\_time$ **then**
15 $\quad$ schedule = server
16 **end**
17 **else**
18 $\quad$ schedule = robot
19 **end**
20 **return** *schedule*

As shown in Algo. 1, we estimate the time cost of switching the computation placement from current p to either the robot and the GPU server for n inference. If the current computation placement is at the robot, switching to the GPU server will incur full transmission of the input data and full computation on the visual model (line 5); if and the GPU server side, switching to the robot will incur full computation on the visual model (line 9). If current computation placement remains, for the robot side, each inference incurs time cost of visual model inference proportional to c (line 6 and 13) and for the server side, each inference incurs time cost of both transmission and visual model inference (line 10 and 12). Finally we select the computation placement with least estimated inference time for n inference as the scheduled computation placement.

Note that the above process is conducted offline during the system initialization stage. At runtime, as the robot collects the input images and estimates the wireless network bandwidth, Collaborative Offload Scheduler maintains an estimation of b and c and every n inference update the computation placement by querying the precomputed schedule. Computation placement to either the robot or the GPU server will cause the CacheInf metadata computed by the previous stages to be transmitted to either the CacheInf Executor at the robot side or at the GPU server side for inference result computation. If the placement is at the GPU server side, the inference results will be transmitted back to the robot as show in Fig. 2.

4.3 End-to-End Runtime

Finally, we are combining all the above components to schedule for computation and offloading in a cache-aware way to optimize end-to-end inference latency for robotic visual models. With Hybrid-Parallel integrated, cache can exists partially both at the robot and the server and we analyze the cache ratio on robot r_c and the cache ratio r_s on server by enquiry the current cached pixels with the previous slice of input (i.e., x_i). For an x_i, we define the minimum portion of locally executed input of its parent operators (i.e., operators whose output is the input of o_i) as x_i' and different between x_i indicates offloading to/from the server. For every operator $o_i \in \boldsymbol{O}$ involved in a visual model, we define its finishing time since the first operator starts executing as t_i^c on the robot (c means client) and t_i^s on server.

We can have the finish time of each operator on the robot and the server as the following, where $D(o_i, x_i' - x_i, r)$ is the data volume needed to be transmitted at operator o_i with cache ratios r_c and r_s and b is the estimated bandwidth:

$$t_i^c = \begin{cases} t_{i-1}^c + T_c(o_i, x_i, r_c - R(\hat{\boldsymbol{O}}_c, o_i)), & 1 \le i \le n \wedge x_i \le x_i' \\ max(T_c(o_i, x_i, r_c - R(\hat{\boldsymbol{O}}_c, o_i)) + \\ \quad t_{i-1}^c, \ \frac{1}{b} D(o_i, x_i' - x_i, r) + t_i^s), & 1 \le i \le n \wedge x_i > x_i' \end{cases}$$

$$t_i^s = \begin{cases} t_{i-1}^s + T_s(o_i, 1 - x_i, r_s - R(\hat{\boldsymbol{O}}_c, o_i)), & 1 \le i \le n \wedge x_i \ge x_i' \\ max(T_s(o_i, 1 - x_i, r_s - R(\hat{\boldsymbol{O}}_c, o_i)) + \\ \quad t_{i-1}^s, \ \frac{1}{b} D(o_i, x_i' - x_i, r_s) + t_i^c), & 1 \le i \le n \wedge x_i < x_i' \end{cases}$$

The first rows of the above two equations describe the scenarios where either the robot or the server does not need to receive data from the opposite side and thus the finishing time of this operator only depends on its local execution time. The second rows instead describe the opposite scenarios, where either the robot or the server needs to receive data from the opposite side (e.g., $x_i > x_i'$ for the robot) and have to wait until the same operator to finish computing at the opposite side and then be transmitted at bandwidth b.

With the above statements, optimizing the end-to-end inference latency for the visual model with cache enabled at a given bandwidth b and cache ratios r_c and r_s is to solve

$$\begin{aligned} & \min_{\boldsymbol{X}} t_n^c \\ s.t. \quad & x_1 = x_n = 1 \\ & \forall j \in \boldsymbol{O}_{nonlocal}, x_j (mod \ 1) = 0 \\ & \forall j \notin \hat{\boldsymbol{O}}_c, x_j = x_j' \end{aligned} \tag{3}$$

In Equation 3, the first two constraints ensure that inference output will finally be located at the robot and non-local operators will always have full input; the third constraint ensures that offloading will not happen within an operator whose cached is merged into the cache of other operators, since we cannot recover the operator's whole feature map.

The resulting algorithms of CacheInf at both the robot and the server are presented in Algorithm 2 and Algorithm 3. Line 1 to 3 in Algorithm 2 and Line 1 to 4 in Algorithm 3 profile the model at both the robot and the server and compute a schedule as described in Section 4.3, where the computation is located on the server to speed up computation. The rest of Algorithm 3 is basically mirrored from that of Algorithm 2 and thus we focus on Algorithm 2 for simplicity.

Line 6 to 8 in Algorithm 2 identifies the reusable cache by matching features between the input image I and its cached counterpart $Cache[1]$ and gets the homography matrix M and the sparse uncached input that needs to be recomputed. After communicating info of bandwidth, cache ratio and homography matrix with the server, we query the *Schedule* to get input ratio x and parent operator input ratio x' as described in Section 4.3. The we start executing each operator o_i involved in the model sequentially. We recover the whole input by combining sparse input *inp* with cache for non-local operators or gather extra pixels from cache for *inp* for sparse local operator computation at cached operators at line 12 to 19. When offloading is required to accelerate inference, we send a slice of *inp* to the server or merge received partial input from the server to *inp* at line 20 to 26. When *inp* is finally ready and not empty, we execute the operator o_i with inp where we choose the sparse local operator for sparse input and choose the original operator for dense input at line 27 to 34.

Algorithm 2: CacheInfClient

Input: A continual sequence of video images $\boldsymbol{I}$; DNN model M
Output: The inference results *ret* on each image in $\boldsymbol{I}$

```
// profile
1  T_c, U_c = Profile(M)
2  Send(M, T_c, U_c)
3  Schedule, Ô_c = Receive()
4  Cache = InitCache(Ô_c)
   // inference
5  foreach I in I do
6      b = EstimateBandwidth()
7      r_c, r_s = AnalyzeCacheRatio(I, Cache[1])
8      inp, M = IdentifyCache(I, Cache[1])
9      Send(b, r_c, r_s, M)
10     x, x' = Schedule[b, r_c, r_s]
11     foreach i = 1, 2, ..., n do
12         if i ∈ O_nonlocal and x_i > 0 and IsSparse(inp)
            then
13             inp = DenseRecover(inp, Cache[i], M)
14             UpdateCache(Cache[i], inp, M)
15         end
16         else if x'_i > 0 and i ∈ Ô_c then
17             inp = SparseGather(inp, Cache[i], M)
18             UpdateCache(Cache[i], inp, M)
19         end
20         if x_i < x'_i then
21             inp, inp' = Slice(inp, x_i, x'_i)
22             Send(inp')
23         end
24         else if x_i > x'_i then
25             inp = Merge(inp, Receive())
26         end
27         if x_i > 0 then
28             if IsSparse(inp) then
29                 inp = SparseExecute(o_i, inp)
30             end
31             else
32                 inp = Execute(o_i, inp)
33             end
34         end
35     end
36     ret[I] = inp
37 end
38 return ret
```

5 Implementation

We implemented CacheInf with python, pytorch [13] and C++ CUDA [1] on Ubuntu20.04. The major components of CacheInf, namely Cache Analyzer, Cache Combiner and Recomputation Input Constructor are implemented via highly optimized CUDA programming and exposed as python extensions for easy to use. We recorded that executing these components on our edge robot equipped with Jetson Xavier NX board took on average 4.3ms.

6 Evaluation

6.1 Evaluation Settings

Testbed. We conducted experiments on a four-wheeled robot and an air-ground robot. Both robots are equipped with a Jetson Xavier NX [11] 8G onboard computer with cuda acceleration capability and a MediaTek MT76x2U USB wireless network interface card for wireless connectivity. The Jetson Xavier NX is connected to a Leishen N10P LiDAR, an ORBBEC Astra depth camera and an STM32F407VET6 controller via USB serial ports, which are managed and driven using ROS Noetic. The GPU server used in our experiments is equipped with an Intel(R) i5 12400f CPU @ 4.40GHz and an

Algorithm 3: CacheInfServer

```
// profile and compute schedule at the server
M, T_c, U_c = Receive()
T_s, U_s = Profile(M)
Schedule, Ô_c = ComputeSchedule(T_s, U_s, T_c, U_c)
Send(Schedule, Ô_c)
Cache = InitCache(Ô_c)
// inference
while True do
    b, r_c, r_s, M = Receive()
    x, x' = Schedule[b, r_c, r_s]
    foreach i = 1, 2, ..., n do
        if i ∈ O_nonlocal and x_i < 1 and IsSparse(inp) then
            inp = DenseRecover(inp, Cache[i], M)
            UpdateCache(Cache[i], inp, M)
        end
        else if x'_i < 1 and i ∈ Ô_c then
            inp = SparseGather(inp, Cache[i], M)
            UpdateCache(Cache[i], inp, M)
        end
        if x_i > x'_i then
            inp, inp' = Slice(inp, x_i, x'_i)
            Send(inp')
        end
        else if x_i < x'_i then
            inp = Merge(inp, Receive())
        end
        if x_i < 1 then
            if IsSparse(inp) then
                inp = SparseExecute(o_i, inp)
            end
            else
                inp = Execute(o_i, inp)
            end
        end
    end
end
```

NVIDIA GeForce GTX 2080 Ti 11GB GPU, connected to our robot via Wi-Fi 6 over 80MHz channel at 5GHz frequency.

Workload. We chose two real-world visual robotic applications as our major workloads: 1. Kapao [10] depicted in Figure 4, a convolution-neural-network-based (CNN-based) real-time people key point detection applications used to guide our four-wheeled robot to track and follow a walking people; 2. AGRNav [22] depicted in Figure 5, a CNN-based autonomous air-ground robot navigation application that predicts unobserved obstacles by semantic prediction on point clouds and optimizes the navigation trajectory for the air-ground robot. We also verified CacheInf's performance on a broader range of visual models common to mobile devices: VGGNet [17], ConvNeXt [24], RegNet [26] using their default implementation of torchvision [20].

Dataset. For AGRNav we used the officially available sequence of point clouds input [22] and for Kapao and the rest of the models from torchvision, we used the DAVIS [14] dataset which are sequences of video images of people and animals doing different activities captured using hand-held cameras.

Experiment Environments. The experiments across all systems and all workloads were conducted with the robot stationary and the charger plugged in. We emulate the mobile indoor environment for the experiments by replaying the wireless network bandwidth limits recorded where the robot was moving in our office with desks and separators interfering wireless bandwidth. We adopt such setting to maintain stable computation performance of the robot across different experiments while ensuring that the setting reflects the limited computation resources and limited and unstable wireless network bandwidth that hinder fast visual model inference on the mobile robot.

Baselines. We selected a state-of-the-art (SOTA) cached-based computation reduction method called EVA2 [4] and a state-of-the-art (SOTA) computation offloading method, Hybrid-Parallel [18] (referred to as HP) as the major baselines. EVA2 reuses the cached activations when the detected MSE of the input image compared with the cached input image is below a threshold or wholly recomputes otherwise. HP splits the input image in parallel computes and offloads different splits of the input to accelerate computation offloading.

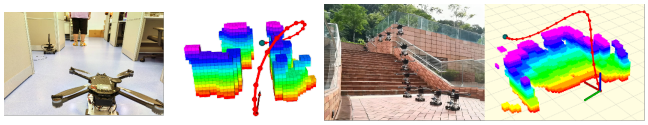
We integrated EVA2 with HP to enable computation offloading by default for fair comparison, and we alter its MSE threshold to two levels high and low to emulate different choices when EVA2 is applied to moving cameras: high MSE threshold means recomputation will be less likely to be triggered for more activation reusing; low MSE threshold will more frequently trigger recomputation for higher accuracy. We also compared its performance with low MSE threshold when computing locally. These results in three evaluation items: EVA2(L), EVA2(H)-HP and EVA2(L)-HP for local version of EVA2 with low MSE threshold, EVA2 with HP and high MSE threshold and EVA2 with HP and low MSE threshold.

We also show the performance of direct local computation of the applications to demonstrate the effectiveness of all these systems. We refer to CacheInf as Ours in the tables. Each result in the tables is followed with standard deviation ($\pm n$).

The evaluation questions are as follows:

(a) Targeted people **(b)** Robot moving trajectory

Figure 4. A real-time people-tracking robotic application on our robot based on a state-of-the-art human pose estimation visual model, Kapao [10].



(a) Indoors **(b)** Outdoors

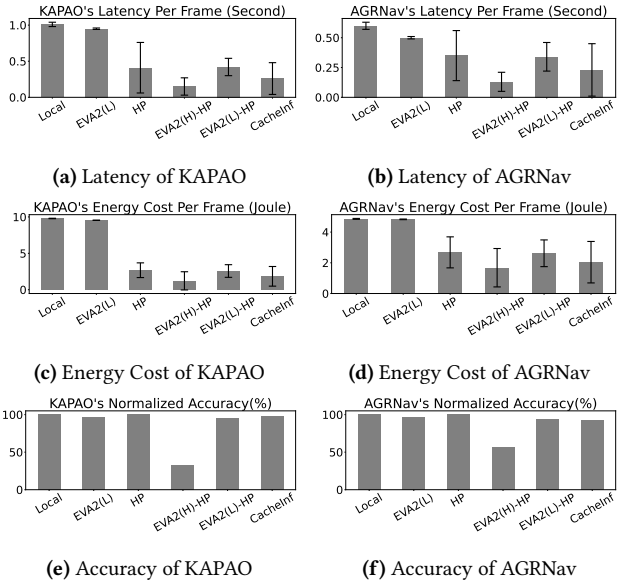
Figure 5. By predicting occlusions in advance, AGRNav [22] gains an accurate perception of the environment and avoids collisions, resulting in efficient and energy-saving paths.

- RQ1: How much does CacheInf benefit real-world robotic applications in terms of inference time, inference accuracy and energy consumption?
- RQ2: How does CacheInf perform on more models common to mobile devices?
- RQ3: How is the above gain achieved in CacheInf and what affects it?
- RQ4: The limitations and potentials of CacheInf.

6.2 End-to-End Performance

Fig. 6 shows the end-to-end statistics of inference latency, energy consumption per inference and inference accuracy. As for inference accuracy, we use the inference results of local computation as the groundtruth to collect inference accuracy for the other items and normalized the inference accuracy against the accuracy. From Fig. 6, we can learn that CacheInf greatly reduces end-to-end inference latency and energy consumption per inference while maintaining high accuracy. Among the baselines that preserve high inference accuracy, CacheInf reduced the inference latency by 36.59% to 74.26% for Kapao and 34.29% to 61.67% for AGRNav; CacheInf reduced the energy consumption per inference by 28.41% to 81.21% for Kapao and 23.89% to 58.02% for AGRNav. While EVA2(H) greatly reduced inference latency and energy consumption, the lowered accuracy makes it unusable for real-world applications.

EVA2(L) and EVA2(L)-HP each achieved similar performance as local computation and HP, which indicates that the high MSE due to perspective changes makes direct cached activation reuse impossible and the frequent triggering of recomputation. In these cases, the overhead of cache analysis computation outweighs the possible gain of rare reusing



(a) Latency of KAPAO **(b)** Latency of AGRNav

(c) Energy Cost of KAPAO **(d)** Energy Cost of AGRNav

(e) Accuracy of KAPAO **(f)** Accuracy of AGRNav

Figure 6. The performance of KAPAO and AGRNav on various systems.

and caused almost no gain compared with wholly local computation and wholly offloading, which makes EVA2(L)-HP even incurred high inference latency than the pure offloading method of HP. In contrast, the ability of CacheInf to reuse cached activations whose receptive fields contain less new information (less internal variance) and partially recomputes the others with more new information (high internal variance) enables it to reduce inference latency while maintaining high inference accuracy across these applications.

6.3 Performance on Various Common Models

The above conclusions can be further validated by results of a wider range of visual models in Table 1 and Table 2. Across different visual models, CacheInf reduced the inference latency by 13.4% to 43.6% indoors and 13.1% to 45.9% outdoors, and it results in the reduction in energy consumed per inference to be 11.1% to 46.7% indoors and 9.5% to 42.2% compared with the baselines. Note that although CacheInf's gain is still evident, the lower bound of CacheInf's gain decreased on these models compared with Kapao and AGRNav; the reason could be that these models are less computation-intensive, which can be implied from their shorter time for local computation compared with Kapao and AGRNav. When inference of a visual models is not computation-intensive, the gain of using sparse local operators in CacheInf will be limited since execution of each local operator will no longer be the bottleneck. In terms of GPU memory consumption, CacheInf increased GPU memory consumption by 3.2% to 24.8% compared with No Cache, while reducing 12.8% to 39.5% GPU memory consumption compared with Cache All.

Model(number of parameters)	Local computaion time/ms	System	Transmission time/ms	Inference time/ms	Percentage(%)
RegNet(54M)	175.0(±23.6)	EVA2(L)-HP	47.6(±47.8)	77.8(±39.3)	61.22
		HP	49.6(±21.7)	55.0(±24.8)	90.18
		CacheInf	44.2(±27.7)	45.3(±35.0)	97.57
VGG19(143M)	118.0(±18.9)	EVA2(L)-HP	38.9(±47.1)	65.2(±28.1)	59.75
		HP	44.8(±20.9)	47.6(±18.1)	94.15
		CacheInf	37.8(±31.2)	41.1(±20.3)	94.26
ConvNeXt(197M)	316.7(±31.0)	EVA2(L)-HP	56.0(±36.1)	79.2(±35.9)	70.72
		HP	56.4(±34.7)	59.7(±26.6)	94.43
		CacheInf	40.4(±37.8)	44.7(±33.3)	90.38

Table 1. Average transmission time, inference time, percentage that transmission time accounts for of the total inference time of common visual models in different environments with different systems.

	System	Power consumption(W)	Energy consumption(J) per inference
RegNet	Local	9.0(±0.3)	1.37(±0.02)
	EVA2(L)-HP	6.04(±1.88)	0.43(±0.16)
	HP	5.24(±1.43)	0.29(±0.08)
	Ours	5.20(±1.51)	0.24(±0.08)
VGG19	Local	9.78(±0.34)	0.95(±0.02)
	EVA2(L)-HP	6.82(±2.10)	0.36(±0.18)
	HP	6.51(±1.74)	0.31(±0.08)
	Ours	6.70(±1.88)	0.27(±0.10)
ConvNeXt	Local	9.92(±0.38)	3.12(±0.03)
	EVA2(L)-HP	4.86(±0.44)	0.31(±0.05)
	HP	4.57(±0.23)	0.27(±0.01)
	Ours	5.26(±0.40)	0.24(±0.05)

Table 2. The power consumption against time (Watt) and energy consumption per inference (Joule) of common visual models in different environments with different systems.

6.4 Breakdown

System	MSE	Recomputation Ratio
CacheInf	22022.6	34.6%
EVA2(L)-HP	22792.4	95.3%
EVA2(H)-HP	22785.4	3.4%

Table 3. The average observed MSE of different cache-based systems and the ratio of recomputation in Kapao

Tab. 3 presents the average observed MSE of the small pixel blocks when applying the block matching algorithm in the Kapao application. We mark the recomputation ratio as the ratio of size of the input fed to the inference pipeline for each inference. For reusing or recomputing cases in EVA2, recomputation ratio is 0. or 1. We can observe that all the systems observed high level of MSE during the inference (note that the pixel value in our experiment ranges from 0 to 255) while three systems apply different strategies: EVA2(L)-HP almost recomputes on every input image due to the frequent triggering of the MSE threshold while EVA2(H)-HP almost never recomputes. There could be a tradeoff threshold that helps EVA2 integrated with HP to achieve a medium performance between EVA2(L)-HP and EVA2(H)-HP; however, a such threshold would most likely results in degradation in both inference latency and inference accuracy. CacheInf endured this high level of MSE and managed to only compute on the partially selected shard of the input image, which reduced inference latency while maintaining high inference accuracy.

6.5 Ablation Study

System	Inference Latency (s)	Energy Consumption (J)
CacheInf	0.26	1.84
CacheInf-Local	0.76	6.34
CacheInf-Full	0.56	3.43

Table 4. The inference latency and energy consumption of CacheInf when disabling offloading (CacheInf-Local) and partial recomputation (CacheInf-Full) in Kapao

Tab. 4 shows the ablation study of CacheInf. We disabled offloading in CacheInf-Local and disabled partial recomputation in CacheInf-Full (only execute full recomputation) and execute the same experiments in Kapao. Both cases caused the performance gain to degrade, and disabling offloading caused the performance gain to degrade in a greater extend, which implies that an important portion of CacheInf's gain comes from transmission data volume reduction. The reason of this is twofold: 1. the computation power of the robot is evidently weaker than the GPU server; 2. the computation time

reduction is not proportional to the size of the input image, resulting in sub-optimal scheduling. However, CacheInf-Local still evidently reduced inference time compared to the simple local computation cases in Fig. 6, indicating the advantages of partial recomputation mechanism in CacheInf.

6.6 Discussion

While CacheInf ultimately reduced inference latency and energy consumption while maintaining high inference accuracy, it is still fundamental trading off between inference latency and inference accuracy which can be seen from the slightly degraded inference accuracy. This is a common limitation of the cache-based visual model inference acceleration methods and such mild inference accuracy degradation is not likely to degrade the application performance. Besides, with the limitation of pixel block matching algorithms, in extreme situations there could be no detected reusable area on the input image and in this case CacheInf will degrade to full recomputation with the selected recomputation area expand to the whole size of the image. As for future work, we are planning to evaluate the performance of CacheInf in a broader range of datasets, environments and visual models to further examine its performance.

7 Conclusion

In this paper, we present CacheInf, a collaborative edge-cloud cache system for efficient robotic visual model inference. Based on the continuity of visual input of robots in the field and the local operators commonly used in visual models, we introduce cache mechanism to visual model inference in CacheInf. By reusing computation results of similar local geometries between consecutive inputs, CacheInf accelerates visual model inference by reducing both local computation time and transmission time when offloading computation to the GPU server, while maintaining high inference accuracy. The more real-time visual model inference on robots enabled by CacheInf will nurture more visual models to be deployed in real-world robots.

References

[1] CUDA c++ programming guide.

[2] iPerf - Download iPerf3 and original iPerf pre-compiled binaries.

[3] Belhassen Bayar and Matthew C Stamm. Constrained convolutional neural networks: A new approach towards general purpose image manipulation detection. *IEEE Transactions on Information Forensics and Security*, 13(11):2691–2706, 2018.

[4] Mark Buckler, Philip Bedoukian, Suren Jayasuriya, and Adrian Sampson. EVA²: Exploiting temporal redundancy in live computer vision. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 533–546. ISSN: 2575-713X.

[5] Lukas Cavigelli, Philippe Degen, and Luca Benini. CBinfer: Change-based inference for convolutional neural networks on video data.

[6] Reagan L Galvez, Argel A Bandala, Elmer P Dadios, Ryan Rhay P Vicerra, and Jose Martin Z Maningo. Object detection using convolutional neural networks. In *TENCON 2018-2018 IEEE Region 10 Conference*, pages 2023–2027. IEEE, 2018.

[7] Loc N. Huynh, Youngki Lee, and Rajesh Krishna Balan. DeepMon: Mobile GPU-based deep learning framework for continuous vision applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 82–95. ACM.

[8] Huanghuang Liang, Qianlong Sang, Chuang Hu, Dazhao Cheng, Xiaobo Zhou, Dan Wang, Wei Bao, and Yu Wang. Dnn surgery: Accelerating dnn inference on the edge through layer partitioning. *IEEE transactions on Cloud Computing*, 2023.

[9] Chao Ma, Jia-Bin Huang, Xiaokang Yang, and Ming-Hsuan Yang. Hierarchical convolutional features for visual tracking. In *Proceedings of the IEEE international conference on computer vision*, pages 3074–3082, 2015.

[10] William McNally, Kanav Vats, Alexander Wong, and John McPhee. Rethinking keypoint representations: Modeling keypoints and poses as objects for multi-person human pose estimation. In *European Conference on Computer Vision*, pages 37–54. Springer, 2022.

[11] NVIDIA. The world's smallest ai supercomputer. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-series/, 2024.

[12] Keiron O'shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

[13] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[14] F. Perazzi, J. Pont-Tuset, B. McWilliams, L. Van Gool, M. Gross, and A. Sorkine-Hornung. A benchmark dataset and evaluation methodology for video object segmentation. In *Computer Vision and Pattern Recognition*, 2016.

[15] Lingyan Ran, Yanning Zhang, Qilin Zhang, and Tao Yang. Convolutional neural network-based robot navigation using uncalibrated spherical images. *Sensors*, 17(6):1341, 2017.

[16] Waseem Rawat and Zenghui Wang. Deep convolutional neural networks for image classification: A comprehensive review. *Neural computation*, 29(9):2352–2449, 2017.

[17] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[18] Zekai Sun, Xiuxian Guan, Junming Wang, Haoze Song, Yuhao Qing, Tianxiang Shen, Dong Huang, Fangming Liu, and Heming Cui. Hybrid-parallel: Achieving high performance and energy efficient distributed inference on robots, 2024.

[19] Sasha Targ, Diogo Almeida, and Kevin Lyman. Resnet in resnet: Generalizing residual architectures. *arXiv preprint arXiv:1603.08029*, 2016.

[20] torchvision. torchvision — Torchvision 0.13 documentation. https://pytorch.org/vision/0.13/, 2024.

[21] Bryan Tripp. Approximating the architecture of visual cortex in a convolutional network. *Neural computation*, 31(8):1551–1591, 2019.

[22] Junming Wang, Zekai Sun, Xiuxian Guan, Tianxiang Shen, Zongyuan Zhang, Tianyang Duan, Dong Huang, Shixiong Zhao, and Heming Cui. Agrnav: Efficient and energy-saving autonomous navigation for air-ground robots in occlusion-prone environments. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2024.

[23] Panqu Wang, Pengfei Chen, Ye Yuan, Ding Liu, Zehua Huang, Xiaodi Hou, and Garrison Cottrell. Understanding convolution for semantic segmentation. In *2018 IEEE winter conference on applications of computer vision (WACV)*, pages 1451–1460. Ieee, 2018.

[24] Sanghyun Woo, Shoubhik Debnath, Ronghang Hu, Xinlei Chen, Zhuang Liu, In So Kweon, and Saining Xie. Convnext v2: Co-designing and scaling convnets with masked autoencoders. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16133–16142, 2023.

[25] Min Wu, Wanjuan Su, Luefeng Chen, Zhentao Liu, Weihua Cao, and Kaoru Hirota. Weight-adapted convolution neural network for facial expression recognition in human–robot interaction. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 51(3):1473–1484, 2019.

[26] Jing Xu, Yu Pan, Xinglin Pan, Steven Hoi, Zhang Yi, and Zenglin Xu. Regnet: self-regulated network for image classification. *IEEE Transactions on Neural Networks and Learning Systems*, 2022.

[27] Mengwei Xu, Mengze Zhu, Yunxin Liu, Felix Xiaozhu Lin, and Xuanzhe Liu. Deepcache: Principled cache for mobile deep vision. In *Proceedings of the 24th annual international conference on mobile computing and networking*, pages 129–144, 2018.

[28] Xinlei Yang, Hao Lin, Zhenhua Li, Feng Qian, Xingyao Li, Zhiming He, Xudong Wu, Xianlong Wang, Yunhao Liu, Zhi Liao, et al. Mobile access bandwidth in practice: Measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 114–128, 2022.