DICE: Dynamic In-Situ Control for Edge-based Applications

Yongtao Yao
University of Delaware
Newark, DE 19716, USA
yongtao@udel.edu

Liping Julia Zhu
Meta Platforms, Inc.
Menlo Park, CA, USA
juliazhu@meta.com

Weisong Shi
University of Delaware
Newark, DE 19716, USA
weisong@udel.edu

***Abstract*—This paper focuses on addressing computational constraints and energy limitations prevalent in edge-based applications through an innovative approach, dynamic in-situ control for edge-based applications (DICE). DICE capitalizes on the burgeoning trend in vehicle sensor technologies, such as camera, Radar, and LiDAR, which are becoming increasingly powerful and capable of performing pre-processing computations. DICE introduces a concept of "downstream offloading", which distinguishes it from traditional offloading approaches that typically offload computational tasks from edge devices to more powerful Edge Servers. In contrast, DICE offloads part of the computational tasks from the Edge Server to the sensor itself, thereby optimizing data processing at the source and reducing the volume of data transmission required.This approach not only addresses the latency bottleneck frequently encountered in energy-intensive neural networks but also enhances the efficiency of data processing by selectively filtering out non-critical frames based on event-triggering mechanisms. DICE leverages the unique strengths of portable devices such as smartwatches and smartphones, even with their inherent computational and power limitations. The framework consists of an adaptive control layer for dynamic task allocation and an application layer designed to deploy quantized models on System on Chips (SoCs) like TinyML, thereby improving the efficiency of AI-driven applications while conservatively utilizing energy. This system proposes a sustainable, energy-efficient pathway for future edge-based applications.**

***Index Terms*—smart sensor, dynamic control, edge collaboration**

I. INTRODUCTION

The advent of high-bandwidth, low-latency communication technologies such as 5G and WiFi6 is unlocking potential for edge-based applications, particularly in mobile scenarios. Alongside the deployment of novel edge computing products, these next-generation communication technologies promise to transform the human-computer interaction experience. Edge computing offers a practical solution to meet the low latency requirements of such applications. In this study, we introduce a task offloading framework based on TinyML, aiming to accelerate data processing on edge-based devices. However, in contrast to conventional offloading approaches that offload computation tasks to a powerful Edge Server, our framework facilitates "downstream offloading", shifting part of the computations from the Edge Server to the sensor device itself. This new approach capitalizes on the computational capabilities of modern sensors in vehicles, optimizing data processing at the source, reducing data transmission volume, and thus mitigating latency.

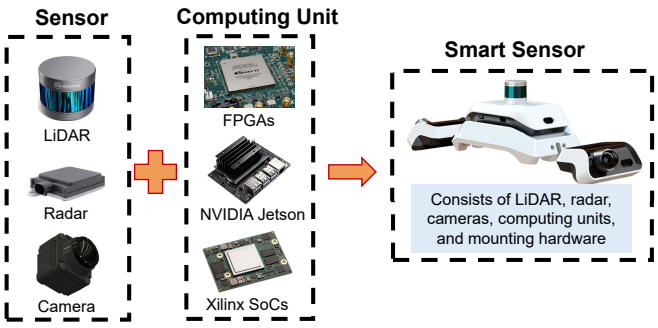

Fig. 1. Smart Sensor.

Connected and Autonomous Vehicle: The combination of communication technologies, robotics, and edge computing [37] has catalyzed significant advancements in the field of autonomous driving. Rigorous research and development initiatives have expedited the emergence of connected and autonomous vehicles (CAVs), as evident from high-profile developments like Tesla's Autopilot [10], Google's Waymo [9], and Baidu's Apollo [44]. Serving as an optimal edge computing platform [28], CAVs support various intelligent applications like real-time remote diagnostics [31] and advanced driver assistance [21]. These innovations are driven by extensive vehicle data generated by multiple onboard sensors, such as cameras, radar, and LiDAR. These integral CAV components are projected to generate approximately 40 terabytes of data for every eight hours of driving—equivalent to the data output of nearly 3,000 people [38]. It's estimated that by 2025, about 470 million CAVs will be operating worldwide, generating an astounding 280 petabytes of data [24]. This development signifies a notable milestone in the application of edge computing in the transportation sector, ushering in a new era of data-driven vehicular innovation.

Computation Intensive Services: While most deep neural networks (DNNs) are designed with a focus on boosting accuracy, this often results in substantially increased model complexity. Current state-of-the-art networks, such as Inceptionv4 [40] and ResNet-50 [2], may contain dozens or even hundreds of layers, each layer requiring millions of matrix

multiplications, to achieve superior accuracy. The computational intensity of these DNN models presents significant challenges when deployed on CAVs with limited computation resources. Despite the increasing computational capabilities of CAV platforms [5], [6], they struggle to keep up with users' growing demands for more resource-intensive applications, such as intersection analysis [22] and driver behavior detection [25]. Hence, our proposed downstream offloading approach under DICE framework becomes increasingly important, ensuring a range of intelligent services (DNN models) can be deployed and executed effectively and dynamically on a single resource-constrained CAV computation platform and CAV fleets.

The DICE framework embraces the concept of a Smart Sensor, as outlined in Figure 1. This Smart Sensor is a holistic integration of diverse sensory inputs and computational units, extending its potential to address a wider array of tasks. Our particular implementation in DICE employs a combination of a camera and a Raspberry Pi as a Smart Sensor, where the Raspberry Pi performs a range of functions from Data Preprocessing to Object Detection. Such a configuration underlines the flexibility of the DICE framework, showcasing its potential to adapt to diverse edge computing scenarios by accommodating various sensor and computing unit combinations. In our study, we emphasize the ability of the DICE framework to significantly minimize latency and energy consumption during neural network computations on edge devices. To demonstrate this, we implement the framework on devices of different computational capabilities, namely a Raspberry Pi and a Google Pixel 4. In typical scenarios, lightweight algorithms deployed on the Raspberry Pi prove sufficient for performing tasks such as pedestrian detection in blind spots and stop sign verification. However, in more complex scenarios demanding greater computational prowess, DICE implements a "downstream offloading" process where the data is transferred to the more powerful Google Pixel 4. This process embodies the adaptive nature of DICE, ensuring optimal performance regardless of the computational constraints of the edge device.

This research makes several important contributions to the field:

- *Downstream offloading with tiny ML*: We implement deep learning algorithms with Tiny ML on embedded devices, using Raspberry Pi and a cell phone as an EdgeServer to enhance detection model accuracy, with the novelty of offloading part of computations from the EdgeServer to the Raspberry Pi.
- *Edge computing-assisted driving*: We enhance pedestrian and road sign detection in challenging scenarios, using deep learning algorithm and distance estimation algorithms, thereby facilitating edge computing-assisted driving with the help of our "downstream offloading" concept.
- *Estimation of pedestrian and traffic sign distances*: We apply distance estimation algorithms to determine pedestrian and traffic sign locations and distances, aiding in safer driving.
- *Collaboration for edge devices*: We utilize Raspberry Pi and cell phone to deploy and execute detection models, with the cell phone acting as an edge server to run high-accuracy models when necessary, highlighting the model's portability and versatility in our "downstream offloading" setting.

This research capitalizes on the synergy between Tiny ML and edge computing, specifically within embedded devices, to optimize resource allocation and enhance model accuracy for real-world applications. Our work employs dynamic control mechanisms to determine when and how devices are used for specific computational tasks, emphasizing adaptability and efficiency in handling edge computing tasks. We hope this paper provides a roadmap for applying Tiny ML and edge computing in autonomous driving and other complex scenarios, thereby contributing to the expanding body of knowledge in this field.

The paper is organized as follows: Section II provides a literature review, presenting related work in the fields of Tiny ML, edge computing, and autonomous vehicles, and highlighting relevant methodologies and their implications. Section III outlines the algorithms and techniques used in this research, detailing the application of Tiny ML and the concept of "downstream offloading" in edge computing. Section IV presents the experimental setup, the execution of the models, and the results obtained, providing empirical validation of our research. Section V offers an in-depth analysis and interpretation of the results, drawing from both theoretical and practical perspectives to understand their implications. Finally, Section VI concludes the paper, summarizing our findings and suggesting potential avenues for future research in the field.

II. RELATED WORK

Algorithms in computing systems play a vital role in sensing, perception, localization, prediction, and control. This section outlines the latest progress in critical algorithmic fields: object detection, Augmented Reality and distance estimation, and task offloading.

1) Object detection: Object detection, especially under challenging conditions, is critical to the real-world application of deep learning for autonomous vehicles (AVs) [29]. Object detection algorithms' development generally goes through two phases: the conventional object detection phase, and the deep learning-supported object detection phase [46]. Traditional methods like Viola Jones Detectors [42], Histogram of Oriented Gradients (HOG) [4], and Deformable Part-based Model (DPM) [7] provided valuable insights. Contemporary deep learning-based methods, such as RCNN series [11], [12], [17], [35], Single Shot MultiBox Detector (SSD) series [8], [26], and You Only Look Once (YOLO) series [20], [33], [34], further evolved detection methods, balancing speed and accuracy [23], [26].

2) Augmented reality: Augmented Reality (AR) on mobile platforms enables the real-time integration of virtual objects, or holographic content, within 3D real-world environments. This technology necessitates an image recognition system that

can accurately identify objects in the camera view with low latency, a challenge heightened by image distortion issues [27]. Liu *et al.* responded to this challenge by proposing CollabAR, a framework for edge-assisted, collaborative image recognition. The CollabAR system, comprising a distortion-tolerant image recognizer, a correlated image finder, and a multi-view ensembler, is deployed on the edge server to manage the tradeoff between recognition accuracy and system latency. Complementing the server, the client runs an anchor-based pose estimation module for tracking mobile device positioning. Implemented on a variety of commercial devices and evaluated on two multi-view image datasets, CollabAR achieved over 96% accuracy in recognizing severely distorted images, while reducing system latency to 17.8ms. In AR, holograms placed by the user are expected to remain stationary relative to real-world objects, but inaccuracies in environment mapping and device positioning often lead to positional errors. Han *et al.* addressed this issue by proposing SceneIt, a mobile AR visual environment rating system based on hologram position error prediction [16]. Utilizing custom scene characterization metrics and considering complex scene attribute interactions, SceneIt was shown to predict positional error severity accurately and efficiently, enhancing mobile AR applications. Further, in an edge-assisted network scenario, Scargill *et al.* proposed Intelli-AR, a set of intelligent preloading algorithms to optimize transmission efficiency [36]. Using a Markov decision process to model device motion trajectories, Intelli-AR adaptively learns optimal preloading policies, dramatically improving successful preloads. Multi-user AR, which allows multiple users to view common virtual objects in a shared location, also necessitates network communication to synchronize virtual object positioning across displays. Ran *et al.* sought to fill the knowledge gap in this area with SPAR, a system that correlates communication data with latency and object positioning in AR displays [32]. Their findings underscored the importance of effective communication strategies, new metrics for updating virtual objects and scene observation, and tools for automatic quantification of inadvertent spatial-temporal changes in virtual object positioning. Deployed on an Android smartphone running open source AR, SPAR reduced latency and spatial inconsistencies by up to 55% and 60%, respectively.

3) Distance estimation: The concept of distance estimation is an integral component of effective object detection, as outlined in the Dist-YOLO architecture [41]. This novel system extends the prediction vectors generated by the original You Only Look Once (YOLO) model by incorporating distance information, thereby creating a more precise object detection model. Furthermore, Dist-YOLO is paired with a fitting distance loss function to optimize the performance of the system. The research showcases the superior accuracy of Dist-YOLO in identifying bounding boxes compared to the original YOLO model, without any additional demand on the backbone's capacity. Moreover, it proves the efficacy of using a monocular camera integrated with Dist-YOLO in precisely estimating the distance of an object.

4) Task offloading: DICE significantly differs from traditional task offloading approaches in its dynamic and adaptive nature. Traditional task offloading methods typically follow a rigid pattern of assigning tasks to available resources without any sophisticated decision-making process [13], [18], [43]. This approach can lead to inefficient resource utilization and suboptimal performance, particularly in scenarios where the task demands or environmental conditions are variable. In contrast, DICE applies a more intelligent approach that considers the "When" and "How" to control a computational task. It uses real-time data and context awareness to make dynamic decisions about which tasks to execute and where to execute them within the network, thus optimizing both the performance and resource utilization. Instead of a static offloading strategy, DICE offers a dynamic control mechanism for edge computing, which makes it more flexible and efficient in handling a variety of application scenarios.

III. DICE FRAMEWORK

A. Edge Device and Edge Server

Edge devices and edge servers represent the two primary components in the DICE Framework, each serving specific roles and maintaining a symbiotic relationship that promotes efficient processing and overall system performance.

Edge Device: In the context of the DICE framework, an edge device is typically an augmented reality (AR) device with limited computational resources. The edge device captures data from its surroundings using various sensors, executes lightweight models deployed on it, and performs functions necessary for real-time operations such as object recognition, pedestrian detection, and road sign detection. Given the resource constraints, the edge device aims to achieve the lowest possible energy consumption without compromising its real-time performance and response capabilities.

Edge Server: The edge server possesses more robust computational resources and serves to complement the edge device's capabilities. It assists in the execution of more complex tasks that might be beyond the processing capacity of the edge device. To achieve this, the edge server accommodates more resource-hungry models, and takes over processing tasks from the edge device when necessary, under the guidance of the adaptive control layer. By doing so, the edge server aids in achieving higher system throughput and ensuring service quality.

The relationship between the edge device and the edge server is a crucial aspect of the DICE Framework. It's a dynamically orchestrated collaboration, regulated by the adaptive control layer, which determines how and when the edge server should assist the edge device based on current workload, network conditions, and available resources. The interaction between the edge device and the edge server allows the DICE Framework to optimise energy consumption and computational efficiency, ensuring that the system can handle both simple and complex tasks effectively.

B. Adaptive Control Layer

The Adaptive Control Layer (ACL) is a critical part of the DICE Framework as it regulates the dynamic allocation of computing tasks between the edge device and the edge server. The aim is to optimize energy consumption and ensure system performance in real-time by considering the current workload, available resources, and network conditions.

The ACL implements a decision-making algorithm that estimates the energy consumption and performance if a task were to be processed on the edge device or offloaded to the edge server. This estimation is performed using a cost function that combines the potential energy usage and latency associated with both options.

Suppose E_{local} and $E_{offload}$ denote the estimated energy consumption of executing the task locally on the edge device and offloading it to the edge server, respectively. Similarly, L_{local} and $L_{offload}$ represent the expected latencies. The cost function C_{local} for local execution and $C_{offload}$ for offloading can be calculated as:

$$\begin{aligned} C_{\text{local}} &= \alpha E_{\text{local}} + \beta L_{\text{local}} \\ C_{\text{offload}} &= \alpha E_{\text{offload}} + \beta L_{\text{offload}} \end{aligned} \tag{1}$$

where α and β are weighting coefficients that reflect the importance of energy consumption and latency, respectively. The decision to execute locally or offload is then determined by:

$$\text{execute\_locally} = C_{\text{local}} < C_{\text{offload}} \tag{2}$$

This decision-making process is repeated in real-time, adjusting to dynamic changes in workload, network conditions, and available resources. The workings of the ACL can be outlined as follows:

Algorithm 1 Adaptive Control

```
while True do
    current_workload = get_current_workload()
    available_resources = get_available_resources()
    network_conditions = get_network_conditions()

    E_local = estimate_local_energy()
    L_local = estimate_local_latency()
    E_offload = estimate_offload_energy()
    L_offload = estimate_offload_latency()

    C_local = alpha*E_local + beta*L_local
    C_offload = alpha*E_offload + beta*L_offload

    if C_local < C_offload then
        execute_task_locally()
    else
        execute_task_on_edge_server()
    end if
end while
```

The Adaptive Control Layer, as presented, provides a dynamic, real-time strategy for balancing energy consumption and latency in the DICE Framework. It maximizes the computational capabilities of both the edge device and the edge server, enhancing the overall system's efficiency and performance.

1) Estimation of Local Energy and Latency: The energy consumption and latency of running the application locally on the edge device are estimated in the Adaptive Control Layer (ACL) of the DICE framework. The aim is to determine whether running the application locally is more energy-efficient and less time-consuming than offloading it to the edge server.

The local energy consumption is dependent on several factors, including the computational complexity of the application, the power profile of the device, and the current state of the device's resources. Thus, the local energy can be approximated as follows:

$$E_{\text{local}} = P_{\text{comp}} \times T_{\text{local}} \tag{3}$$

where E_{local} is the local energy, P_{comp} is the computational power consumed by the device (which is specific to the device and can be obtained from device specifications or profiling), and T_{local} is the local latency or the time taken to run the application locally.

The local latency T_{local} can be estimated based on the computational complexity of the application and the current load on the device's processor. It can be expressed as:

$$T_{\text{local}} = \frac{C_{app}}{R_{\text{device}}} \tag{4}$$

where C_{app} is the computational complexity of the application (measured in FLOPs or another suitable metric), and R_{device} is the current computational resource available on the device (measured in FLOPs/sec or another corresponding metric).

2) Estimation of Offload Energy and Latency:: The energy and latency of offloading the application to the edge server are also estimated in the ACL. The aim here is to determine whether offloading the application to the edge server is more energy-efficient and less time-consuming than running it locally on the edge device.

The energy consumption for offloading the application comprises the energy required to transmit the data to the edge server and the energy to receive the processed data from the edge server. The offload energy can be approximated as:

$$E_{\text{offload}} = E_{\text{trans}} + E_{\text{recv}} \tag{5}$$

where E_{trans} and E_{recv} represent the energy consumption for data transmission and reception, respectively. These can be calculated based on the size of the data to be transferred, the transmission/reception power of the device, and the transmission/reception rates.

The offload latency comprises the time to transmit the data to the edge server, the time for the server to process the data, and the time to receive the processed data from the server. It can be approximated as:

$$T_{\text{offload}} = T_{\text{trans}} + T_{\text{server}} + T_{\text{recv}} \tag{6}$$

where T_{trans}, T_{server}, and T_{recv} represent the latency for data transmission, server processing, and data reception, respectively. These can be calculated based on the size of the data to be transferred, the transmission/reception rates, and the server processing speed. More details for the energy and latency estimation can be found in Appendix A. This pseudocode first computes the local and offload latency and energy by calling relevant functions. It then makes the decision whether to run the application locally or offload it to the server based on the estimated values. The decision process can be made more complex by incorporating other factors, such as current device load, network conditions, and user preferences.

3) Execute Task on Edge Server: The underlying principle revolves around decision-making on whether to process tasks locally or offload them to the edge server, based on the current workload, energy, latency estimates, and the network state. The function takes into account these factors, then determines and initiates the task offloading if deemed beneficial.

The communication between the local edge device and the edge server can be implemented using a client-server model over a high-speed network protocol such as 5G or Wi-Fi 6, which can significantly reduce communication latency and provide high data rates. When the offloading decision is made, the edge device encapsulates the relevant input data into packets and sends them to the edge server through the established high-speed connection. The edge server, upon receiving the data, unpacks it, processes the received task, and sends the results back to the edge device. To ensure high communication efficiency, several techniques can be applied:

- *Data compression technique*: it's used before sending data to the edge server, reducing the amount of data to be transferred and thus, saving bandwidth and reducing transmission latency.
- *Concurrent data transmission*: Multiple data streams are transmitted concurrently to fully utilize the network capacity.
- *Error control mechanisms*: Mechanisms such as Automatic Repeat reQuest (ARQ) are used to handle possible transmission errors and ensure data integrity.

While these methods can enhance the efficiency of communication, it's vital to evaluate the associated overheads. The decision to offload tasks should not only consider the computation latency but also take into account the communication overhead incurred by the task offloading.

C. Application Layer

The Application Layer (AL) is the second primary component of the DICE Framework. This layer focuses on optimizing AI-based applications for AR devices and their deployments on edge devices. It adopts quantized tiny models, which are lightweight deep learning models with reduced computational complexity and energy consumption. The Application Layer also supports event-driven triggers, thus ensuring applications remain responsive while minimizing energy usage.

Quantized tiny Models: Tiny models are pruned and quantized versions of full-scale deep learning models. Pruning involves reducing the size of the model by removing less critical connections or parameters, while quantization means reducing the precision of the model's numerical parameters. The tiny models in the Application Layer of the DICE framework are deployed on edge devices, striking a balance between model accuracy and computational efficiency.

Algorithm 2 Adaptive Control

```
# Define the original model
M = get_original_model()

# Prune the original model to create the tiny model
P = prune(M)

# Quantize the pruned model for further optimization
Q = quantize(P)

 # Deploy the tiny model on the edge device
deploy_model(Q)

while True do
    event = check_for_event()
    if event then
        # Execute the tiny model
        output = execute_model(Q)
        handle_output(output)
    end if
end while
```

Model pruning and quantization are pivotal techniques in optimizing deep learning models for deployment on resource-constrained edge devices. Pruning eliminates unnecessary parameters in a trained model that contribute minimally to the final prediction, thus reducing model size and computational requirements, making it more efficient [15]. Quantization, on the other hand, decreases the numerical precision of the model's weights and activations, resulting in significantly lower storage and computational needs without significant loss in accuracy [14]. Together, these techniques allow for the creation of compact, efficient models that maintain a balance between computational requirements and accuracy, suitable for deployment on edge devices. The *pruning* and *quantization* processes can be expressed with the following equations. If M represents the original model and P represents the pruned model, the pruning operation is defined as:

$$\mathrm{P} = prune(\mathrm{M}) \tag{7}$$

where the *prune* operation reduces the size of M while maintaining a similar level of accuracy. The quantized model Q is obtained from the pruned model as:

$$\mathrm{Q} = quantize(\mathrm{P}) \tag{8}$$

where the *quantize* operation reduces the precision of P's numerical parameters, leading to a model that is computationally less complex.

DeepC is an open-source compiler and inference framework aimed at bringing the power of deep learning to resource-

constrained hardware devices. It provides a platform for running deep learning models on microcontrollers and edge devices, enabling edge computing applications to leverage the power of artificial intelligence. DeepC supports an array of deep learning models and provides functionalities for model optimization, such as quantization and layer fusion. These optimizations are critical for deploying deep learning models on edge devices with limited computational resources and memory. The compiler converts high-level language representation of a model into efficient machine code, which can be directly executed on edge devices. DeepC offers a seamless way to deploy and run optimized deep learning models on edge devices, facilitating the development of efficient edge computing applications. In our DICE framework, we use DeepC for model optimization in the Application Layer, ensuring that the AI-driven applications run efficiently on edge devices.

Event-driven triggers:

Central to the DICE framework is an event-driven mechanism that informs the selection of key frames for downstream offloading. Within this process, the Raspberry Pi conducts an initial preprocessing of the video stream, utilizing a Tiny ML model deployed on the device to classify each frame. Frames that include a target object—whether a pedestrian or a traffic sign—are identified as key frames and subsequently transmitted to the Edge Server for further analysis. Conversely, non-key frames, which do not contain identified targets, are not selected for transmission, optimizing bandwidth and computation resources.

This event-driven approach enables the DICE framework to focus computational resources on frames that are most likely to provide valuable information. By narrowing the scope of analysis in this way, the system can perform more in-depth processing without overwhelming the computational capabilities of the edge device or overloading the communication link between the device and the Edge Server. Additionally, the flexibility of the DICE framework allows for the definition of a broader set of events that could trigger the downstream offloading of data for additional processing. For instance, potential events could include instances when the camera is obstructed by a small insect or when the camera malfunctions. By training the Tiny ML model on the Raspberry Pi to recognize such events, the system can respond more effectively to a wider array of scenarios, enhancing its overall performance and utility.

The event-driven mechanism embedded within the DICE framework thus represents a versatile and efficient approach to managing the computational and communication demands of edge computing applications. By intelligently directing resources based on the event-driven detection of key frames, the DICE framework can maximize system performance, providing a robust platform for the real-time processing and analysis of data in edge-based applications.

The pseudocode of the Application Layer is presented in Algorithm 2. The Application Layer, as described, enables the DICE Framework to deploy optimized deep learning models on edge devices and execute them based on event-driven triggers. This design enhances application responsiveness and efficiency, facilitating a smooth user experience even in resource-constrained settings.

IV. EXPERIMENTAL DESIGN AND METHODOLOGY

In this study, we introduce a system that synergistically integrates edge computing and AR technology to facilitate automated driving. This system is devised to detect pedestrians and stop signs, estimate their distance, and deliver timely alerts to the driver, thereby helping to prevent traffic accidents and violations. This section elucidates the experimental design behind our proposed system, which encompasses three main components: an edge computing module, an AR display module, and a pedestrian and stop sign detection module.

The edge computing module is entrusted with processing the raw sensor data extracted from the vehicle's cameras. The AR display module takes charge of superimposing virtual information onto the real-world scene. The pedestrian and stop sign detection module is tasked with detecting pedestrians and stop signs and estimating their distance. To assess our system's performance, we devised a series of experiments encompassing various driving scenarios. Specifically, we constructed three distinct scenarios to emulate different traffic conditions: urban, suburban, and rural. Each scenario includes a combination of different road conditions, traffic densities, and weather conditions.

For data collection, we outfitted a test vehicle with multiple cameras. The data, which included RGB and depth information, was captured at a rate of 10 frames per second. The gathered data underwent pre-processing to eliminate noise or outliers before being divided into training and testing sets.

Our pedestrian and stop sign detection module employed a deep learning-based approach, where two separate models were trained for pedestrian detection and stop sign detection, respectively. The training data was generated by manually labeling the pedestrians and stop signs in the collected data. We evaluated the detection module's performance using standard metrics such as precision, recall, and F1 score.

To appraise the entire system's performance, we quantified the accuracy and latency of the pedestrian and stop sign detection and AR display modules. Furthermore, we assessed the system's capability to deliver timely alerts to drivers and its effectiveness in reducing traffic accidents or violations.

Our work introduces a Raspberry Pi-based pedestrian and stop sign detection system, where the lightweight model can be executed on the Raspberry Pi to detect pedestrians and stop signs in blind areas during driving. Under typical conditions, the system doesn't detect pedestrians or stop signs in 80% of the scenarios. However, when the system suspects the presence of pedestrians or stop signs, it triggers data offloading to a smartphone for detection. The smartphone, acting as the Edge Server, hosts a more powerful model, thereby enabling more accurate detection. The experiments employed YOLOv3 and SSD models for training and evaluated the models' performance in lab and real-world scenarios. We also assessed the

models' detection accuracy, resource utilization, and overall performance to confirm their feasibility. The experimental results demonstrated that the system could accurately detect pedestrians and stop signs in diverse scenarios while maintaining excellent performance. The system's application is anticipated to offer drivers a safer and more convenient driving experience.

A. Experimental Platform Description

In our experimental setup, we link the Raspberry Pi with a High Definition (HD) Web Camera. We conceptualize the Raspberry Pi as the computational module of the Web Camera. This approach aims to simulate the functions of novel camera technologies, furnishing capabilities such as data preprocessing and filtering. By employing this configuration, we attempt to demonstrate the potential of advanced camera systems in offloading computational tasks, underscoring the role of downstream offloading in optimizing the operation of sensor-driven systems.

Fig. 2. Experiment Platform.

The Unzano HD800 video camera was utilized as the primary data acquisition device in our edge-based experimental setup due to its range of features that met our research requirements. The camera's USB connectivity facilitated easy integration with our Raspberry Pi platform, while its CMOS photo sensor technology provided high-resolution, clear imaging data essential for accurate detection and classification tasks. The camera's compact design, H.264 support, digital-camera, and hd-movie capabilities also made it an efficient choice for our setup. Its lightweight and portable design, black color, indoor and outdoor usage specifications, and audio recording capabilities added to the robustness of our experimental platform.

We utilized a Raspberry Pi 4 Model B as the Edge device. The device incorporates a Broadcom BCM2711B0 quad-core A72 (ARMv8-A) 64-bit 1.5GHz processor and a Broadcom VideoCore VI GPU. With its dual 2.4 GHz and 5 GHz 802.11b/g/n/ac wireless LAN and Gigabit Ethernet connectivity, it boasts 4 GB of LPDDR4 SDRAM and microSD storage. It also includes Bluetooth 5.0 and Bluetooth Low Energy (BLE) capabilities. With its compact size and considerable computing power, this device exhibits high scalability for executing detection models. We also paired the device with a 4000mAh battery pack to ensure extended usage time.

The EdgeServer in our experiment is the Google Pixel 4, which is powered by a Qualcomm Snapdragon 855 processor and an Adreno 640 GPU. This device possesses 6GB RAM and 128GB storage. Owing to its high computing power and portability, this device can effectively serve as a potent EdgeServer for running high-precision inspection models. To prolong detection periods, the device is fitted with a 2800mAh battery.

For the AR device, we deployed the Samsung Galaxy Watch 4, an advanced smartwatch model announced in August 2021. Its dimensions are 44.4 x 43.3 x 9.8 mm, and the weight varies between 30.3 g (for the 44mm model) and 25.9 g (for the 40mm model). The watch incorporates a super AMOLED display of 1.4 inches with a resolution of 450 x 450 pixels. It runs on Android Wear OS with One UI Watch 3 and has an Exynos W920 (5 nm) chipset. Furthermore, it boasts 16GB internal storage with 1.5GB RAM. The smartwatch also features several sensors, including accelerometer, gyro, heart rate, and barometer, thereby offering an ideal platform for AR applications. The smartwatch is powered by a Li-Ion 361 mAh battery, ensuring prolonged usage.

B. Dataset Selection

DARK FACE: DARK FACE dataset provides 6,000 real-world low light images captured during the nighttime, at teaching buildings, streets, bridges, overpasses, parks etc., all labeled with bounding boxes for of human face, as the main training and/or validation sets. We also provide 9,000 unlabeled low-light images collected from the same setting. Additionally, we provided a unique set of 789 paired low-light/normal-light images captured in controllable real lighting conditions (but unnecessarily containing faces), which can be used as parts of the training data at the participants' discretization. There will be a hold-out testing set of 4,000 low-light images, with human face bounding boxes annotated.

Nighttime Vehicle Detection Dataset: This dataset is collected for vehicle classification in darkness collected and labeled in the Autonomous Robots Lab at the University of Nevada, Reno. This dataset contains 10913 gray-scale images of night-time images of roads. The images all have dimensions of 1280 x 1024 pixels (width x height). Not all images in the dataset contain vehicles, though the majority (90%) do. The data was collected using a PointGrey Chameleon 3 Grayscale camera (CM3-U3-13S2C-CS-BD).

NightOwls Dataset: Pedestrian detection at night from a RGB camera is an under-represented yet very important problem, where current state-of-the-art vision algorithms fail. Computer vision methods for detection at night have not received much attention, despite the fact they are a critical building block of many systems such as safe and robust autonomous cars. The NightOwls Dataset focuses on pedestrian detection at night. It consists of 279,000 fully-annotated images in 40 video

sequences recorded at night, captured by an industry-standard camera.

C. Testbed Setup

This study proposes a system that leverages the synergies of edge computing and AR technologies for automated driving assistance, specifically for detecting pedestrians and traffic signs, and consequently providing timely warnings to drivers. The proposed system has been deployed on three main devices - a Raspberry Pi 4 Model B as the Edge device, a Google Pixel 4 as the EdgeServer, and a Samsung Galaxy Watch 4 as the AR device. This section outlines the detailed design and methodology of the experiment conducted to evaluate the efficacy and performance of the proposed system.

1) System Deployment: A lightweight detection model was deployed on the Raspberry Pi 4 Model B (Edge device) to continuously scan the surroundings for potential targets i.e., pedestrians or traffic signs. The tiny model was designed for lower computational complexity, thereby enabling it to run efficiently on the Raspberry Pi while conserving energy.

Upon the detection of a potential target, the system triggers a more sophisticated and high-precision model on the Google Pixel 4 (EdgeServer) for an in-depth classification of the target, and to estimate its distance from the vehicle. The EdgeServer device has more computational capabilities and can handle the higher complexity of the detection model.

The Samsung Galaxy Watch 4 (AR device) is used to notify the driver of potential hazards, using an intuitive AR interface. Once the system on the EdgeServer finalizes the classification and distance estimation, an alert is sent to the AR device, providing the driver with a timely warning.

2) Experimental Scenarios: The experimental validation of the proposed system was conducted under various driving scenarios, to simulate different traffic situations. These scenarios were designed to test the system in different environments, including urban, suburban, and rural areas, with varying traffic densities and weather conditions.

3) Performance Metrics: The system's performance was evaluated using several metrics, focusing on detection accuracy, resource utilization, system latency, and power consumption. The detection accuracy of both the lightweight model and the high-precision model were evaluated. The resource utilization on both Edge device and EdgeServer were also measured, taking into account the CPU and memory usage. The system latency, including the detection time and the communication latency between the Edge device and the EdgeServer, was also evaluated. The power consumption of the Edge device during the whole process was also monitored.

4) Comparative Study: To further validate the performance of our DICE system, we compare its performance with traditional cloud-based solutions and other existing edge computing models. The comparative study would focus on the same performance metrics as described above.

This experimental design aims to provide a comprehensive evaluation of the DICE system in various realistic driving scenarios, thereby highlighting its effectiveness and benefits in assisting automated driving and enhancing traffic safety.

D. Experiment Methodology

1) Building a traffic signs & pedestrian classifier: In our DICE experimental setting, we constructed a robust and sophisticated Convolutional Neural Network (CNN) model to classify traffic signs and pedestrians. The model was built using TensorFlow's Keras library and was organized in a sequential manner, implying that each layer of the model is in sequence and directly connected to the layers adjacent to it. The architecture of the model is outlined as follows: The model begins with a convolutional layer having 64 filters, each with a size of 3x3, an input shape of 32x32x3, and using the Rectified Linear Unit (ReLU) activation function. This layer is followed by a max pooling operation with a pool size of 2x2, which reduces the spatial dimensions of the output from the previous layer. After this, a batch normalization operation is applied to standardize and stabilize the learning process. Two additional convolutional blocks with a similar architecture are included, with the number of filters increased to 128 and 256 respectively, to allow the model to learn more complex representations. After the convolutional operations, the output is flattened to transform the 3D output to 1D. Then, the model goes through five dense layers with 512, 256, 128, 64, and 32 units respectively, all using ReLU activation functions. Each dense layer is accompanied by a dropout layer with a rate of 0.2, which randomly sets a fraction of the input units to 0 during training time, thereby preventing overfitting. Finally, a dense output layer with five units is included using a softmax activation function, which gives the output as a probability

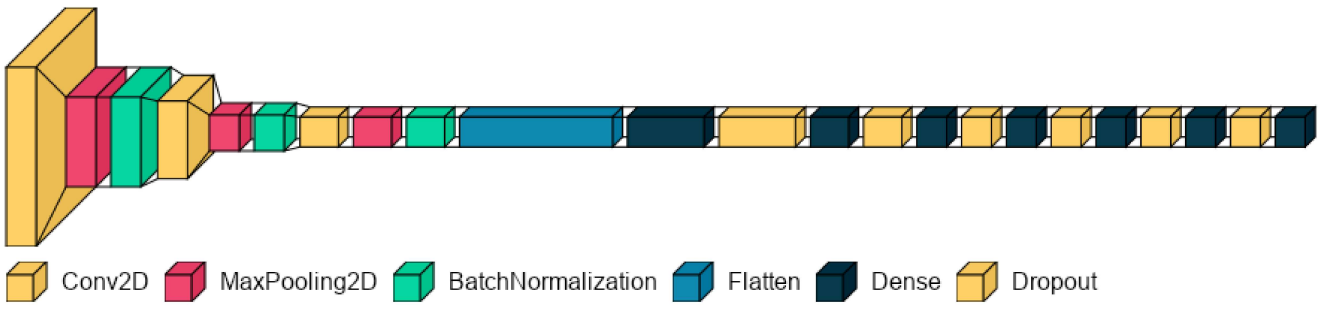

Fig. 3. Traffic signs & pedestrian classifier.

distribution over the five classes (presumably corresponding to different traffic signs and pedestrian situations). This model was compiled with the Adam optimizer, which adapts the learning rate during training, a categorical cross-entropy loss function, and accuracy as the evaluation metric. The resulting CNN model is deep, adaptable, and capable of identifying and distinguishing between different traffic signs and pedestrians effectively, playing a key role in our DICE experiments.

2) Object detector: We utilize YOLOv3-Tiny [1], a faster, resource-efficient version of the full YOLOv3 model, suitable for real-time object detection. Its streamlined structure incorporates fewer convolutional layers, optimizing speed and performance on less powerful devices like Raspberry Pi. Despite this reduction, YOLOv3-Tiny maintains respectable accuracy and excels in real-time processing, making it ideal for initial detection of pedestrians and traffic signs in our system.

Fig. 4. Object distance estimation.

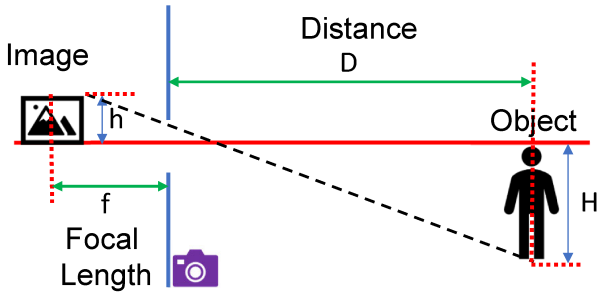

Fig. 5. Object distance estimation.

3) Object distance estimation: We implement a monocular camera model for object distance estimation [3], capitalizing on the principle of similar triangles in the pinhole camera model. This principle dictates a one-to-one relationship between an object and its image. As shown in Fig. 5, we derive a relationship involving the known parameters: the camera's focal length (f), the height of the object in the image plane (h), the height of the object in the object plane (H), and an unknown parameter, the distance from the camera to the object (D). The formulas we obtain are:

$$\frac{h(pixel)}{f(pixel)} = \frac{H(cm)}{D(cm)}$$
$$f = h \times \frac{D}{H} \tag{9}$$
$$D = H \times \frac{f}{h}$$

To establish the focal length, we employed a practical scenario. A reference object of known height was chosen, and the height (h) of the reference object's image on the image plane was determined post-segmentation. By adjusting the distance (D) from the camera to the object to a specific value, we were able to substitute these parameters into the second equation of Equation 9 to calculate the focal length (f) in pixels. Subsequently, we were able to estimate the distance D using the second formula of Equation 9.

V. EVALUATION AND DISCUSSION

We evaluate the performance of the DICE framework in terms of energy consumption, throughput, and latency. Our experimental results show that DICE significantly reduces energy consumption while maintaining high throughput and low latency for AI-driven applications on Edge devices.

Our experimental design aims to illustrate the efficacy of the DICE framework in a simulated real-world scenario: a Raspberry Pi is utilized as the computational module of a camera, where it hosts a Tiny model for pedestrian and traffic sign classification. The mobile phone, acting as the Edge Server, is responsible for further data analysis, specifically for object detection tasks. An adaptive control layer determines the execution location for the distance estimation algorithm. Finally, a smartwatch is used to receive the final detection results and issue alerts.

We conduct a performance evaluation of the DICE framework, focusing on energy consumption, throughput, and latency. Our experimental setup is designed to show that DICE can significantly reduce energy consumption while maintaining high throughput and low latency, which are critical factors for AI-driven applications on edge devices.

TABLE I
PERFORMANCE ON THE EDGE.

	CPU Usage	Mem Usage	Latency
Idle	0.2%	494 MB	-
deepC	25%	495.4 MB	0.64s
Non-Tiny model	33.6%	620.9 MB	0.15s

Table I presents the performance characteristics of the edge-based computation under different operational conditions: Idle, running deepC (the deep learning model used in our framework), and running a Non-Tiny model for comparison.

In the Idle state, CPU usage is minimal, at 0.2%, with a memory footprint of 494 MB. As expected, there is no measurable latency, as no computational tasks are being performed.

When running deepC, the CPU usage increases to 25%, which is a significant but manageable load, demonstrating the efficiency of the deepC model on the edge device. The memory usage sees a minor increase to 495.4 MB, illustrating the model's low memory footprint. The latency for processing each frame is measured at 0.64 seconds, which is reasonably low and suitable for real-time applications. In contrast, when running the Non-Tiny model, CPU usage rises to 33.6%, indicating that more computational resources are required. Correspondingly, the memory usage increases significantly to 620.9 MB, a clear demonstration of the heavier resource requirement of the Non-Tiny model. However, it is noteworthy that the latency for the Non-Tiny model is 0.15 seconds, lower than that of deepC. This could be due to the Non-Tiny

model's design being optimized for speed, despite its higher resource usage. This table illustrates the resource efficiency of the deepC model compared to a Non-Tiny model, underlining the benefits of using such compact, lightweight models in edge-based computations. The latency comparison also opens up a trade-off discussion between resource usage and latency, offering insight into the factors that must be considered when choosing or designing models for deployment on edge devices.

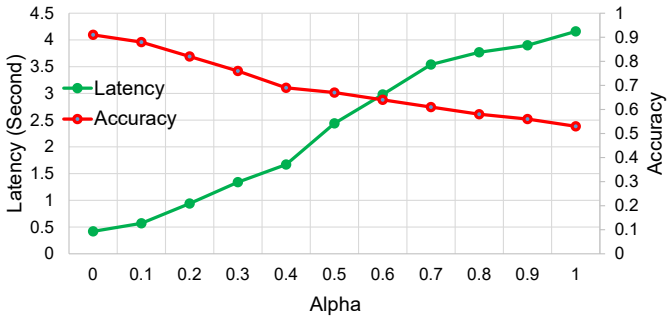

Fig. 6. Dynamic Control Experiment: Balancing Accuracy and Latency in the DICE Framework.

The experiment shown in Fig. 6 was conducted to examine the dynamic control capabilities of the DICE framework. Both Raspberry Pi and Google Pixel 4 were deployed with object detection models. However, Raspberry Pi had an optimized and lighter model while Google Pixel 4 ran a higher-accuracy model. By adjusting the alpha parameter in line 11 and 12 of Algorithm 1 (with beta set as 1-alpha), we were able to measure the average accuracy and latency jointly achieved by both devices under different alpha values.

As alpha increased from 0 to 1, the accuracy decreased from 0.91 to 0.53. This downward trend in accuracy was expected due to the higher reliance on the lower-accuracy model on Raspberry Pi as alpha increased. The results validate DICE's capability of offering dynamic control in balancing performance (accuracy) and computational load (latency). On the other hand, latency increased from 0.42 to 4.16 as alpha went up. This was due to the reduced processing power from the Raspberry Pi, which led to an increase in latency as alpha increased. Hence, it's noteworthy that there is a trade-off between accuracy and latency depending on the alpha parameter.

These results showcase the efficacy of DICE in managing task offloading between devices with disparate computing capabilities. This illustrates the adaptability of DICE in real-world edge computing scenarios, effectively providing an adjustable knob to balance between accuracy and latency based on system requirements.

Table II outlines the performance characteristics of the DICE framework across different event loads, ranging from 0% to 100%. The table presents measurements of CPU utilization, memory usage, uplink data rate, and latency.

At the lowest level of event detection (0%), the CPU utilization is 51.42%, and the memory usage is 37.20%, with

TABLE II
DATA FRAME CONSUMPTION EFFICIENCY OF DICE AND PERFORMANCE ON EDGESERVER.

	Idle	0% events	30% events	50% events	100% events
CPU Utilization	16.53	51.42	77.50	83.36	89.61
Memory	25.50%	37.20%	45.90%	49.50%	61.40%
Uplink (kB/s)	3.10	3.10	43.39	76.23	177.07
Latency (s)	-	0.25	2.48	3.83	4.16

an uplink data rate of 3.10 kB/s and a latency of 0.25s. These relatively modest resource allocations reflect the light computational load associated with processing frames that do not contain a target object. As the percentage of events increases to 30%, we observe a substantial increase in CPU utilization, rising to 77.50%, and memory usage, up to 45.90%. The uplink data rate also increases notably to 43.39 kB/s due to the increased number of key frames being offloaded for further analysis. Correspondingly, the system latency increases to 2.48s due to the added computational and transmission burdens. The trends of increasing CPU utilization, memory usage, uplink data rate, and latency continue as the event load increases to 50% and 100%. This trend is expected, as higher event loads correspond to a larger number of key frames, resulting in increased computational demand and data transmission. At the highest event load (100%), the CPU utilization reaches 89.61%, memory usage is at 61.40%, uplink data rate soars to 177.07 kB/s, and the latency increases to 4.16s. While these resource allocations and latency are considerable, they reflect the system's capability to handle heavy computational loads without crashing or overrunning memory capacity.

Overall, the DICE framework exhibits a robust performance and adaptability in response to varying event loads. The results demonstrate the ability of the DICE framework to allocate system resources effectively based on the event-driven detection of key frames and to maintain operational efficiency despite increasing computational and communication demands.

DICE's key frame extraction mechanism exhibits several advantages over the approaches discussed in the [30] and [19]. The critical distinction is that DICE uses an event-driven mechanism to identify key frames, unlike traditional methods, which rely primarily on temporal or visual features. This allows DICE to adapt to the changing content of the video stream more dynamically and reduce the computation and network load by transmitting only the most significant frames. Additionally, unlike other approaches that perform key frame extraction at the edge or cloud server, DICE performs this operation directly on the device (in our case, the Raspberry Pi). This means that DICE can reduce the amount of data that needs to be transmitted over the network, thereby reducing network bandwidth requirements, latency, and energy consumption.

Yang's work focuses on optimizing the allocation of computational resources to various tasks in an edge computing

TABLE III
COMPARISON OF DICE WITH OTHER WORKS

Feature/Work	Generic Framework for Task Offloading [30]	Optimized Key Frame Extraction [19]	Resource Allocation for Task Offloading [45]	Motion-Based Key Frame Extraction [39]	DICE
Tailored for AR Devices	No specific focus on AR devices	No specific focus on AR devices	No specific focus on AR devices	No specific focus on AR devices	Yes
Event-Driven Mechanism	Absent	Absent	Absent	Absent	Yes
Network Bandwidth, Latency, and Power Efficiency	Standard	Standard	Improved through optimized resource allocation	Standard	Improved through event-driven frame selection and on-device processing
Key Frame Extraction Location	Not applicable	Performed at the edge or cloud server	Not applicable	Performed at the edge or cloud server	Performed directly on the device

network [45]. While these methods can offer improved performance in general edge computing applications, they don't account for the specific demands of AR devices, nor do they typically consider the unique opportunities provided by on-device processing. DICE, on the other hand, is specifically designed for AR applications and takes full advantage of on-device processing capabilities. Sujatha *et al.* uses motion detection or optical flow algorithms to select key frames in video processing applications [39]. While such methods can be effective for certain tasks, they generally require significant computational resources and are typically performed on powerful edge or cloud servers. By contrast, DICE uses a more lightweight, event-driven approach to key frame selection, which can be performed directly on the AR device and provides more efficient use of network and computational resources.

The DICE framework presents a significant step forward in edge computing, enabling a more effective use of available resources while maximizing the accuracy and responsiveness of applications. By considering the inherent heterogeneity of edge devices, DICE provides a powerful tool for harnessing the full potential of edge computing, leading to safer and more efficient applications. The flexibility and adaptability of DICE ensure that it can be effectively utilized across a wide range of edge computing scenarios, highlighting its potential for widespread adoption.

VI. CONCLUDING REMARKS

In this study, we introduced the Dynamic In-Situ Control for Edge-based Applications (DICE), a novel framework that leverages the emerging concept of downstream offloading. DICE targets vehicular applications and harnesses the increasing computational capabilities of vehicular sensors, such as cameras, radars, and LiDARs. Unlike traditional offloading approaches that offload computation from edge devices to more powerful servers, DICE pushes a part of the computation downstream to the sensor level. This approach enables sensors to pre-process and filter data based on event-driven triggers, reducing unnecessary data transmission and increasing the efficiency of the overall system. DICE was implemented using a TinyML-based model on a Raspberry Pi, acting as a simulated computational module for a high-definition web camera. The framework was tested in a variety of scenarios, demonstrating its potential for enhancing the performance of edge-based devices. By optimally allocating computational tasks between edge devices and sensors, DICE successfully reduced latency and conserved energy. DICE represents a significant step forward in the application of edge computing in vehicular scenarios, proving the viability and efficiency of downstream offloading. The implications of this research extend beyond vehicular applications, offering insights for the broader development of energy-efficient, edge-based applications.

VII. FUTURE WORK

There are two primary areas in which we aim to extend this work. Firstly, our current classification and object detection models sometimes struggle to perform well under challenging conditions, such as during nighttime or in poorly lit environments. The tiny models we employ do not deliver high accuracy under these circumstances. Consequently, we will consider developing specific training routines for these types of scenarios to enhance the precision of our models. This adaptation could prove crucial for a range of real-world applications that involve non-optimal conditions. Secondly, our current focus is largely confined to classification and object detection. In future work, we aim to explore other use cases such as tracking. For instance, we can design caching mechanisms within the camera's computation module to expedite subsequent tracking tasks. These enhancements could yield significant improvements in the overall efficiency and performance of our system, opening up new possibilities for utilizing downstream offloading in increasingly complex applications. Our ongoing and future research will continue to probe these boundaries, contributing to a growing understanding of the intersection between edge computing, Tiny ML, and real-world, sensor-driven applications.

ACKNOWLEDGEMENTS

This work is supported in part by the National Science Foundation (NSF) grants #2140346.

REFERENCES

[1] P. Adarsh, P. Rathi, and M. Kumar, "Yolo v3-tiny: Object detection and recognition using one stage improved model," in *2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS)*, 2020, pp. 687–694.

[2] T. Akiba, S. Suzuki, and K. Fukuda, "Extremely large minibatch sgd: Training resnet-50 on imagenet in 15 minutes," *arXiv preprint arXiv:1711.04325*, 2017.

[3] Y.-T. Cao, J.-M. Wang, Y.-K. Sun, and X.-J. Duan, "Circle marker based distance measurement using a single camera," *Lecture Notes on Software Engineering*, vol. 1, no. 4, p. 376, 2013.

[4] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*, vol. 1. IEEE, 2005, pp. 886–893.

[5] Z. Dong, Y. Gu, J. Chen, S. Tang, T. He, and C. Liu, "Enabling predictable wireless data collection in severe energy harvesting environments," in *2016 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2016, pp. 157–166.

[6] Z. Dong, Y. Gu, L. Fu, J. Chen, T. He, and C. Liu, "Athome: Automatic tunable wireless charging for smart home," in *Proceedings of the Second International Conference on Internet-of-Things Design and Implementation*, 2017, pp. 133–143.

[7] P. Felzenszwalb, D. McAllester, and D. Ramanan, "A discriminatively trained, multiscale, deformable part model," in *2008 IEEE conference on computer vision and pattern recognition*. IEEE, 2008, pp. 1–8.

[8] C.-Y. Fu, W. Liu, A. Ranga, A. Tyagi, and A. C. Berg, "Dssd: Deconvolutional single shot detector," *arXiv preprint arXiv:1701.06659*, 2017.

[9] S. Gibbs, "Google sibling waymo launches fully autonomous ride-hailing service," *The Guardian*, vol. 7, 2017.

[10] S. Gillmore and N. L. Tenhundfeld, "The good, the bad, and the ugly: Evaluating tesla's human factors in the wild west of self-driving cars," in *Human Factors and Ergonomics Society Annual Meeting*, 2020.

[11] R. Girshick, "Fast r-cnn," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1440–1448.

[12] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 580–587.

[13] H. Guo, J. Liu, and J. Lv, "Toward intelligent task offloading at the edge," *IEEE Network*, vol. 34, no. 2, pp. 128–134, 2019.

[14] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International conference on machine learning*. PMLR, 2015, pp. 1737–1746.

[15] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, 2015, pp. 1135–1143.

[16] Y. Han, Y. Chen, R. Wang, J. Wu, and M. Gorlatova, "Intelli-ar preloading: A learning approach to proactive hologram transmissions in mobile ar," *IEEE Internet of Things Journal*, 2022.

[17] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2961–2969.

[18] A. Islam, A. Debnath, M. Ghose, and S. Chakraborty, "A survey on task offloading in multi-access edge computing," *Journal of Systems Architecture*, vol. 118, p. 102225, 2021.

[19] Z.-g. Jiang and X.-t. Shi, "Application research of key frames extraction technology combined with optimized faster r-cnn algorithm in traffic video analysis," *Complexity*, vol. 2021, pp. 1–11, 2021.

[20] Joseph Redmon and Ali Farhadi, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.

[21] V. K. Kukkala, J. Tunnell, S. Pasricha, and T. Bradley, "Advanced driver-assistance systems: A path toward autonomous vehicles," *IEEE Consumer Electronics Magazine*, vol. 7, no. 5, pp. 18–25, 2018.

[22] K. Lee, J. Flinn, and B. D. Noble, "Gremlin: Scheduling interactions in vehicular computing," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, ser. SEC '17. New York, NY, USA: ACM, 2017, pp. 4:1–4:13. [Online]. Available: http://doi.acm.org/10.1145/3132211.3134450

[23] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2980–2988.

[24] L. Liu, S. Lu, R. Zhong, B. Wu, Y. Yao, Q. Zhang, and W. Shi, "Computing systems for autonomous driving: State-of-the-art and challenges," *IEEE Internet of Things Journal*, 2020.

[25] L. Liu, X. Z. M. Qiao, and W. Shi, "Safeshareride: Edge-based attack detection in ridesharing services," in *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*. Boston, MA: USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/hotedge18/presentation/liu

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

[27] Z. Liu, G. Lan, J. Stojkovic, Y. Zhang, C. Joe-Wong, and M. Gorlatova, "Collabar: Edge-assisted collaborative image recognition for mobile augmented reality," in *2020 19th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. IEEE, 2020, pp. 301–312.

[28] S. Lu, Y. Yao, and W. Shi, "Collaborative learning on the edges: A case study on connected vehicles," in *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*, 2019.

[29] C. Michaelis, B. Mitzkus, R. Geirhos, E. Rusak, O. Bringmann, A. S. Ecker, M. Bethge, and W. Brendel, "Benchmarking robustness in object detection: Autonomous driving when winter is coming," *arXiv preprint arXiv:1907.07484*, 2019.

[30] A. Naouri, H. Wu, N. A. Nouri, S. Dhelim, and H. Ning, "A novel framework for mobile-edge computing by optimizing task offloading," *IEEE Internet of Things Journal*, vol. 8, no. 16, pp. 13 065–13 076, 2021.

[31] S. Orf, M. R. Zofka, and J. M. Zöllner, "From level four to five: Getting rid of the safety driver with diagnostics in autonomous driving," in *2020 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*. IEEE, 2020, pp. 19–25.

[32] X. Ran, C. Slocum, Y.-Z. Tsai, K. Apicharttrisorn, M. Gorlatova, and J. Chen, "Multi-user augmented reality with communication efficient and spatially consistent virtual objects," in *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, 2020, pp. 386–398.

[33] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[34] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 7263–7271.

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

[36] T. Scargill, S. Hurli, J. Chen, and M. Gorlatova, "Demo: Will it move? indoor scene characterization for hologram stability in mobile ar," *Proceedings of ACM HotMobile*, vol. 2021, 2021.

[37] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE internet of things journal*, vol. 3, no. 5, pp. 637–646, 2016.

[38] L. Sidi and S. Weisong, "The emergence of vehicle computing," *IEEE Internet Computing*, 2021.

[39] C. Sujatha and U. Mudenagudi, "A study on keyframe extraction methods for video summary," in *2011 International Conference on Computational Intelligence and Communication Networks*. IEEE, 2011, pp. 73–77.

[40] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Thirty-first AAAI conference on artificial intelligence*, 2017.

[41] M. Vajgl, P. Hurtik, and T. Nejezchleba, "Dist-yolo: Fast object detection with distance estimation," *Applied Sciences*, vol. 12, no. 3, p. 1354, 2022.

[42] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*, vol. 1. IEEE, 2001, pp. I–I.

[43] L. Wang, Q. Zhang, Y. Li, H. Zhong, and W. Shi, "Mobileedge: Enhancing on-board vehicle computing units using mobile edges for cavs," in *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2019, pp. 470–479.

[44] K. Xu, X. Xiao, J. Miao, and Q. Luo, "Data driven prediction architecture for autonomous driving and its application on apollo platform," in *2020 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2020, pp. 175–181.

[45] S. Yang, "A joint optimization scheme for task offloading and resource allocation based on edge computing in 5g communication networks," *Computer Communications*, vol. 160, pp. 759–768, 2020.
[46] Z. Zou, Z. Shi, Y. Guo, and J. Ye, "Object detection in 20 years: A survey," *arXiv preprint arXiv:1905.05055*, 2019.

APPENDIX

A. The energy and latency estimation

```
# Define the computational complexity of the application and the device resources
C_app = get_app_complexity()
R_device = get_device_resources()

# Estimate local latency and energy
T_local = estimate_local_latency(C_app, R_device)
E_local = estimate_local_energy(T_local)

# Define the data size, transmission/reception power and rates, and server processing speed
Data_size = get_data_size()
P_trans, P_recv = get_device_power_profile()
Rate_trans, Rate_recv = get_trans_recv_rates()
Server_speed = get_server_speed()

# Estimate offload latency and energy
T_trans, T_recv = estimate_trans_recv_latency(Data_size, Rate_trans, Rate_recv)
T_server = estimate_server_latency(Data_size, Server_speed)
T_offload = T_trans + T_server + T_recv

E_trans, E_recv = estimate_trans_recv_energy(Data_size, P_trans, P_recv, Rate_trans, Rate_recv)
E_offload = E_trans + E_recv
# Functions to estimate local and offload latencies and energies
def estimate_local_latency(C_app, R_device):
    return C_app / R_device

def estimate_local_energy(T_local):
    P_comp = get_device_power_comp()
    return P_comp * T_local

def estimate_trans_recv_latency(Data_size, Rate_trans, Rate_recv):
    T_trans = Data_size / Rate_trans
    T_recv = Data_size / Rate_recv
    return T_trans, T_recv

def estimate_server_latency(Data_size, Server_speed):
    return Data_size / Server_speed

def estimate_trans_recv_energy(Data_size, P_trans, P_recv, Rate_trans, Rate_recv):
    T_trans, T_recv = estimate_trans_recv_latency(Data_size, Rate_trans, Rate_recv)
    E_trans = P_trans * T_trans
    E_recv = P_recv * T_recv
    return E_trans, E_recv
```