TESTUDO: Collaborative Intelligence for Latency-Critical Autonomous Systems

Mohanad Odema, Luke Chen, Marco Levorato, and Mohammad Abdullah Al Faruque, *Senior Member, IEEE*

***Abstract*—Edge computing is to be widely adopted for autonomous systems (AS) applications as compute-intensive processing tasks can be offloaded to compute-capable servers located at the edge of the network infrastructure. Given the critical nature of numerous AS applications, their tasks are mostly governed by strict execution deadlines to alleviate any safety concerns from delayed responses. Although wireless link uncertainty has prompted recent works to designate redundant local execution as an offloading fail-safe to ensure these deadlines are met, frequent invocation of such fail-safe mechanisms can potentially undermine the extent of performance gains from offloading. In this paper, we thoroughly analyze how redundant execution overheads can influence the overall performance. Then, we present TESTUDO, a methodology to optimize the energy consumption for latency-sensitive AS applications employing collaborative edge computing. Primarily, our methodology encompasses two main stages: (*i*) Designing processing pipelines supporting optimal offloading points and fail-safe integration using modular design techniques, and (*ii*) Developing a context-aware adaptive runtime solution based on Deep Reinforcement Learning to adapt the mode of operation according to the wireless network status. Our experiments for end-to-end control and object detection use-cases have shown that TESTUDO achieved energy gains reaching up to 31% and 13.4% (15.9% and 5.3% on average) for the former and latter, respectively, while incurring little-to-no degradation in prediction scores ($< 1\%$ change) from state-of-the-art strategies.**

***Index Terms*—Edge Computing, Collaborative Intelligence, Latency-Critical, Autonomous Systems,**

I. INTRODUCTION AND RELATED WORKS

THe wide-scale deployment of autonomous systems (AS) is edging ever closer to becoming reality given the massive prospect of benefits from applications such as autonomous driving and unmanned aerial vehicles (UAV). Hence, machine learning – and deep neural networks (DNNs) in particular – has been at the forefront of algorithmic techniques applied for primary processing given their exceptional performances on crucial autonomy-related tasks. However, the challenge has always been to map these compute-intensive algorithms onto the *relatively* resource-constrained AS computing platforms. For instance, the continuously growing computational demands has led the newer generation of autonomous driving systems (ADS) hardware platforms, the Nvidia AGX Orin, to incur a thermal design power (TDP) rating of 800 Watts [1] – a $3.2\times$ increase from its predecessor, the Nvidia Drive PX2 [2], which can reduce a vehicle's driving range by a factor up to 11.5% [3]. In the case of UAVs, physical form constraints can limit hosting powerful computing platforms, and consequently limiting their applications' scope [4].

This work was partially supported by the National Science Foundation (NSF) under award CCF-2140154. Mohanad Odema, Luke Chen, and Mohammad Abdullah Al Faruque are with the Department of Electrical Engineering & Computer Science, University of California at Irvine, CA 92697 USA (emails: {modema, panwangc, alfaruqu}@uci.edu). Marco Levorato is with the Computer Science Department, Donald Bren School of Information and Computer Sciences, University of California at Irvine, CA 92697 USA (email: levorato@uci.edu).

To address this, *edge computing* has been presented as promising solution for processing burden of compute-intensive tasks can be delegated to compute-capable servers located at the edge of the network infrastructure. Albeit similar to conventional cloud computing [5]–[8], the rationale behind adopting edge computing hinges on having the compute servers positioned within close proximity from where the data is initially generated, firmly mitigating the effects of propagation delays [9]. To further bound the impact of transmission overheads, researchers have proposed to shrink the data prior to transmission, most prominently through the application of *in-model compression* techniques that map the high-dimensional raw input data to low scale representations before the offloading point with little-to-no impact on the performance accuracy [10], [11]. What's more, the promise of higher bandwidths from the forthcoming V2X and 5G wireless technologies is poised to further diminish the effect of added transmission latencies.

Despite all these efforts, the volatile nature of wireless links can cause additional delays that can compromise the operational integrity of such latency-critical autonomous systems. Specifically, abrupt interruptions and micro-scale delays from a multitude of dynamic factors such as path propagation losses, network congestion, server queuing delays, and even the motion characteristics of the AS itself, can accumulate to cause a notable impact on the overall processing latency, especially precarious for autonomous systems control applications whose nominal safety relies on providing outputs within tens of milliseconds [12]. Inspired by the *redundancy* policies of safety-critical systems [13], recent edge computing works have embraced a similar idea to address response time uncertainty through redundant execution schemes, in which an additional execution pipeline – usually on the local platform – is activated whenever server response time is predicted to peak due to wireless channel impairments, which we denote here by *offloading fail-safe* mechanisms [4], [14]–[16]. Unlike conventional fail-safe approaches that apply redundancy for *functional* safety, that is, to ensure a system is immune of software bugs and/or hardware failures [12], [17], *offloading fail-safe* are more concerned with safety from a *nominal* sense, in which correct and timely outputs are to be provided every time-step even if the wireless channel is impaired.

On the same subject, although fail-safe techniques target handling extreme rare cases of operation in which deviation from normal operation can occur, divergent cases in wireless networks are more likely to occur considering the highly dynamic environment of mobile autonomous systems employing edge computing, which makes the *offloading fail-safe* subclass liable to more frequent triggering compared to others. Specifically, navigation mechanics and wireless links' fragility can combine to introduce subtle additional delays that can accumulate faster than channel quality estimates are updated at the AS, possibly causing sub-optimal offloading decisions, where despite ensuring critical deadlines are met, the extent of performance gains is likely to be affected, instilling doubts on the efficacy of offloading in the first place and whether local execution presented the better execution option in retrospect. In summary, we find collaborative intelligence approaches for latency-sensitive AS suffering from the following limitations:

1) The overhead of incorporating offloading fail-safe techniques for reliable offloading is largely understudied, that is, repeated invocations of these secondary routines can limit the desired degree of performance efficiency given how dynamic wireless networks can be.
2) Despite redundant execution and reactivation of local resources being adopted for offloading reliability, they are not characterized as part of performance modelling.
3) Setting the reactivation of local compute resources as the offloading fail-safe guarantees meeting execution deadlines regardless of the wireless state. However, additional incurred costs in terms of energy consumption are overlooked for the most part.

A. Motivational Case Study

We present an example on how offloading fail-safe routines can affect performance efficiency in a real-world scenario, we analyze how the end-to-end control self-driving offloading solution presented in [14] would perform under different wireless condition states, where autonomous driving systems (ADS) are required to complete processing and analysis of the collected input data within 100 ms of their acquisition. Primarily, this constraint is inspired by how numerous self-driving datasets collect frames at a sensor sampling frequency of 10 Hz [3], [12]. As a way to promote energy efficiency without compromising the execution deadlines, [14] proposed an optimal offloading strategy to select between local or remote execution based on estimates of the experienced data rates. Additionally, reactivation of local computing resources is also designated as the fail-safe when the slack time for receiving results back from the edge server expires. For this motivational example, we implemented a pipeline for processing 720p camera inputs using a ResNet-18 [18] on an NVIDIA Drive PX2 [2] – used by Tesla for their Autopilot ADS [19]. We also collected a trace of 500 LTE data rate samples so as to compare the overall energy consumption under their offloading strategy against continuous local execution.

As illustrated in Figure 1, we evaluate the efficacy of their offloading strategy under both ideal and challenging wireless conditions compared to that of pure local execution. By challenging wireless conditions, we mean the scenarios in which the intricate wireless channel impairments – e.g., sudden obstruction of line-of-sight during navigation – accumulate to delay the response times of the edge server beyond the critical threshold (i.e., the 100 ms deadline) after offloading has already been attempted. We define a parameter called *failure rate* which denotes the amount of times the offloading

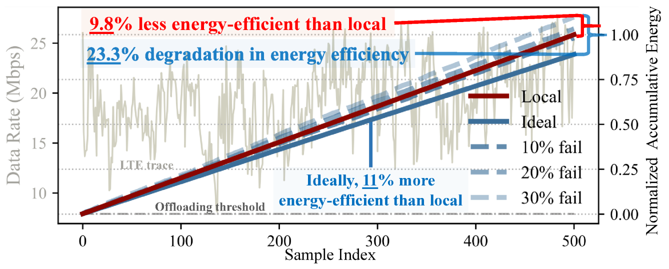


Fig. 1. Cumulative Energy consumption for processing a 720p input image given the Nvidia Drive PX2 ADS, the optimal offloading strategy in [14], and the collected LTE data rate trace in the background for different failure rates. Energy values are normalized with respect to that from local execution.

fail-safe is invoked instantiating the reactivation of the local computing resources for the given wireless data rate trace. As shown, offloading under ideal conditions is 1.12× more energy-efficient compared to local execution over the same LTE trace. However, it can be observed that the performance deviates from the ideal when the percentage of failure rate increases. For instance at 30%, not only does energy consumption increase by 23.3% over the ideal case, but also by a factor of 9.8% over pure local execution. It should be noted that it is for the purpose of this motivation that we analyze the effects of relatively high failure rates given the short trace of samples. In practice, controllers are likely to detect high failure rates (as 30%), and revert back to pure local execution. In the experiments section, we place emphasis on more realistic failure rate settings (as low as 1%) over longer sample traces.

Summary and conclusions from observations: In most cases, offloading strategies developed for latency-critical AS are predominantly adequate to operate under specific expected behaviors of the wireless network. However, delay variations experienced in real-world scenarios can have a profound impact on performance, potentially causing fail-safe triggering that hampers the overall solution's effectiveness. Therefore, given the variations in networking infrastructure from one region to the other, as well as the highly dynamic nature of the communication channel, edge server load, and urban navigation, which all exhibit complex spatio-temporal distributions that influence the overall delay, a different approach is needed.

B. Problem and Research Challenges

Achieving resource-efficient and reliable edge computing for AS instigates addressing the following key challenges:

1) How to account for fail-safe integration during the design stage so as to minimize their energy consumption overhead without compromising the solution's utility?
2) How to effectively adapt the runtime operational policy according to an *all-encompassing* view of the wireless condition state, and its impact on processing efficiency?
3) How to ensure that the adopted optimizations – at both design and deployment – would maintain the degree of robustness required by the critical AS applications?

C. Novel Contributions

To address the aforementioned challenges, our main contributions in this paper are listed as follows:

1) We present TESTUDO, a methodology for implementing dynamic neural network solutions for latency-critical

AS employing remote edge computing. As far as our knowledge goes, TESTUDO is the first to present a multi-branch neural network model distributed across both the local platform and server, serving both the primary and secondary offloading fail-safe processing routines while maintaining the nominal safety for the AS application.

2) At the design stage, TESTUDO adopts block-wise Neural Architecture Search (NAS) as a modular design approach to design the multi-branch neural network. In particular, knowledge distillation (KD) methods are applied to optimize the design the various computing blocks constituting the different execution paths of the dynamic neural network to efficiently integrate both features of split-computing and early-exiting.
3) For the deployment phase, TESTUDO enacts a deep reinforcement learning (DRL) approach to select for each time window the processing pipeline that maximizes prediction quality and energy efficiency given the execution deadlines. Our novel contribution here lies in leveraging the abstract information generated each time window within the processing domain for the DRL's input state observation in the following time step. This aids in discerning the underlying *contextual* and *temporal* correlations existing in the data stream, and consequently estimate the input sample complexity and network stability status.
4) Evaluation of TESTUDO on the use-cases of end-to-end control in ADS and UAVs' object detection has demonstrated that it improves on energy efficiency compared to the state-of-the-art edge offloading strategies – reaching up to 31% with an average of 15.9% for the former, and 13.4% with an average of 5.3% for the latter with virtually no impact on models' utility.

II. Problem Formulation

In this section, we formulate our problem to minimize the energy consumption footprint for the latency-critical AS applications employing collaborative intelligence. Since we are engaging this problem from the perspective of the edge device, i.e., the AS platform itself, our analysis is performed under the assumption that the AS has already established connection with the edge server from which it gets the strongest reception signal – which in most cases is the one closest geographically. From here, our formulation relies on modelling the experienced latency and energy consumption at the edge device when operating within the vicinity of the edge server's coverage area. Furthermore, we rely on a simple model of channel failures – which anyway captures channel correlation – to obtain a clear performance evaluation.

Formally, multiple execution strategies, M, can be supported when edge computing is provided for autonomous systems, comprising at least a basic local routine in addition to another remote execution mode [4]. Each operational mode $m \in M$ can possess its own unique execution path and performance overhead. Thus for each m, we can breakdown the key performance metrics; end-to-end latency, L^m_{total}, and energy consumption, E^m_{total}, into the following components:

$$L^m_{total} = L^m_{exec} + L^m_{comm} + L^m_{ser} \tag{1}$$

$$E^m_{total} = E^m_{exec} + E^m_{comm} + E^m_{idle} \tag{2}$$

where L^m_{exec} and L^m_{ser} represent the local and edge server's execution latencies, respectively. Similarly, E^m_{exec} and E^m_{idle} correspond to the energy consumption of the local computing platform during execution and idle states, with the latter being reached whenever the system is neither processing nor transmitting data – as in waiting for the results from the edge server. L^m_{comm} and E^m_{comm} represent the respective latency and energy for communication which can be given as:

$$L^m_{comm} = L^m_{Tx} + L^m_{Rx} + \delta(L_{prop}, L_{queue}) \tag{3}$$

$$E^m_{comm} = E^m_{Tx} + E^m_{Rx} \tag{4}$$

where L^m_{Tx}, L^m_{Rx}, E^m_{Tx}, and E^m_{Rx} are the transmission latencies and energy consumption in the uplink and downlink. δ represents a random function that captures the additional uncertain latencies that may be experienced by an AS in the deployment environment. We characterize two dominant factors influencing δ: propagation delays, L_{prop}, and queuing delays, L_{queue}. The latter directly represents the randomness associated with how occupied the remote edge server's queues are, which in turn translates into additional waiting times until the offloaded task is dispatched for processing. In terms of the former, the causes for propagation delays for AS can be broken down into two bilateral sources: (*i*) the nature of radio waves propagation that incurs path losses due to diffraction, reflection, and other effects in the deployment environment, where the edge device can suffer degradation in the received signal strength as a result of obstructed line-of-sight or multi-path fading, also translating into additional delays due to the longer round-trip times and/or re-transmissions; and (*ii*) The unique motion characteristics of an AS which further exacerbate the path loss effects as a result of the movement patterns, speed, orientation, antenna alignments, proximity, etc. Given how the modelling dynamics of these factors can be extremely challenging to solve in real-time considering the milliseconds operational scale of AS, we abstract all these factors into the random function δ as our main focus is the end-to-end latency, irrespective of the true instantaneous causes of such additional delays. We further denote the respective transmission sizes during upload and download as a and b and define the transmission overheads as follows:

$$L^m_{Tx} = \frac{a^m}{\phi_u},\ E^m_{Tx} = P_{Tx} \cdot L^m_{Tx} \tag{5}$$

$$L^m_{Rx} = \frac{b^m}{\phi_d},\ E^m_{Rx} = P_{Rx} \cdot L^m_{Rx} \tag{6}$$

in which ϕ_u, ϕ_d, P_{Tx}, and P_{Rx} are the data rates and transmission power estimates at the local platform during upload and download, respectively. In practice, the upload and download transmission sizes, a^m and b^m, are the ones after post-processing distributed across several data frames, where each possesses header information as additional data specific to the networking protocol besides the application payload. Nevertheless, header sizes can be negligible when dealing with high-fidelity data (e.g., image representations), and from this point forward, our analysis of the transmission overheads will be based on the true feature data sizes without networking overheads, following the practice in relevant edge computing works [5], [6], [10], [14]

From here, the selection process of an edge computing mode m at runtime needs to factor two essential requisites: (*i*) *Reliability*; in the sense that critical hard execution deadlines associated with the AS application must be met regardless of the selected execution path or wireless network state. We formally give the reliability constraint as $L^{m}_{total} <= T$, with T being the critical deadline, i.e., execution window length. (*ii*) *Robustness*; through ensuring that any selected operational mode does not degrade prediction quality compared to a canonical computing baseline – as in pure local execution. To elaborate, a mode m can be more energy-efficient through instantiating simpler local computing modules, leading to smaller evaluations of L_{exec} and E_{exec} in (1) and (2). The caveat, however, is that some inputs may experience higher prediction errors when processed by the simpler model compared to the full-sized baseline model. Formally, we can define this error increase as Δerr^{m} = err^{m} - err^{base}, and the robustness constraint to be $\Delta err^{m} < err_{th}$, where err_{th} is a predefined positive tolerance threshold (err_{th}=0 indicates the extreme case where no increase in error is tolerated). Hence, we can optimize the AS edge computing operation for energy efficiency through defining the following objective for every time window of duration T as follows:

$$\min_{m \in M} \; E^{m}_{total}, \;\; \text{s.t.} \;\; L^{m}_{total} <= T, \; \Delta err^{m} < err^{th} \tag{7}$$

where the goal is to identify an optimal edge computing operational mode that provides the lowest energy footprint as defined in (2), subject to both the *reliability* and *robustness* constraints as defined above.

Modeling fail-safe offloading: When remote edge computing is supported, we can break down a neural network model into two parts: a *head* portion which comprises the network layers that precede the offloading point (if any) on the local edge platform, and a *tail* which constitutes parts of network past the offloading point which are deployed on the edge server. In the case of direct input offloading, the entire neural network model is deployed on the edge server as a tail model with no layers assigned to the local device. When it comes to latency-critical AS applications, reactivation of local computing resources need to be incorporated as a recovery routine to account for network uncertainty when server responses in the corresponding time window are perceived to peak beyond a critical threshold. This means some form of the *tail* model needs to be deployed on the local edge device, leading the definition of L_{exec} in (1) to become:

$$L_{exec} = \begin{cases} L_H, & \text{if } L_H + L_{comm} + L_{ser} < T - L_{FS} \\ L_H + L_{FS}, & \text{otherwise} \end{cases} \tag{8}$$

where L_H is the execution latency for the *head* portion of a model on the local platform prior to offloading whereas L_{FS} is the execution latency component for the offloading fail-safe.

Given the lack of emphasis on how often does L_{exec} in (8) evaluate to the second case during offloading – which is dependent on the randomness induced by δ in (3), the projected performance gains from offloading might be overly optimistic. TESTUDO aims to remedy this deficiency by *actively* considering the impact of the offloading fail-safe invocation frequencies on the overall performance to guide both the *design* and *deployment* stages of the edge computing solution. In Figure 2, we depict an instance of the final system model rendered with execution blocks distributed between the local platform and the edge server, supporting multiple potential operational modes that instigate the need for a learning-based approach for solving (7). More details are provided on the design and functionality of each component in the next section.

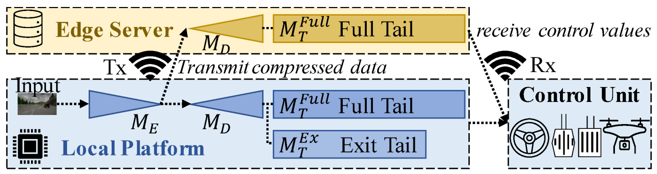


Fig. 2. An example end-to-end control system architecture rendered through TESTUDO supporting reliable edge computing for autonomous systems. Offloading point is placed following the $\mathcal{M}_E$ component. The aggregation of $\mathcal{M}_D$ and either of $\mathcal{M}_T^{Full}$ or $\mathcal{M}_T^{Ex}$ can be used as the offloading fail-safe.

III. SYSTEM DESIGN

We first describe the processing pipeline classes for AS, detail our proposed design approach supporting optimal offloading points, and discuss how to implement the various processing components using modular design techniques.

A. End-to-end Processing Pipelines in Autonomous Systems

For AS, there are two primary approaches to implement end-to-end control pipelines:

Imitation learning: The approach instigates a model learning how to *imitate* human experts' behavior with regards to a specific control task (e.g., self-driving) [20], where a model can learn through supervised learning to minimize a loss function between its predictions and ground-truth values. Mainly, there are two primary components: (*i*) *Perception*; to perceive events occurring in the environment, sensing modalities are provided to the AS through sensory equipment (e.g., vision through mounted cameras) to abstract higher state representations from the collected data through a processing model (e.g., DNN) [18], and (*ii*) *Control*; concatenated at the end of the perception pipeline to receive its outputs – in addition to any available control inputs (e.g., turn left signal) – and translate them into the necessary control outputs.

Modular Pipelines: This is the standardized approach for implementing industry-grade processing pipelines for AS [3], [4], [12], [21], which relies on having independent modules placed at different parts of the computing pipeline, each receiving the partial outputs from the preceding module(s) for processing to provide new partial outputs for the subsequent module(s) until the final control unit outputs are generated, where every module is responsible for a specific learnable task – as how the outputs from perception and localization modules in an ADS are provided to a planning module [22]. Since perception constitutes the bulk of the processing load [3], directing offloading optimizations towards its modules can maximize performance gains across the entire pipeline.

B. In-Model Compression for Split Computing

One prominent approach for achieving efficient split-computation between the local platform and the edge server is the application of in-model compression to obtain optimal offloading points. Formally, a DNN model $\mathcal{M}$ can be split into two parts: a head $\mathcal{M}_H$ and tail $\mathcal{M}_T$ to be deployed on the local and edge server platforms, respectively. The direct approach to select the splitting layer, ℓ, has been to identify the layer at which the output $z_\ell = \mathcal{M}_H(x)$ becomes smaller in size than the input x to decrease transmission overhead. Oftentimes, this criterion is only met at the latter layers for many DNN architectures, which leads to increased local computation [5], [8]. Instead, recent split computing works proposed the notion of in-model compression through a *bottleneck* [11], in which a modified model version $\mathcal{M}'$ would comprise 3 sections: $\mathcal{M}_E$, $\mathcal{M}_D$, and $\mathcal{M}_T$. Submodels $\mathcal{M}_E$ and $\mathcal{M}_D$ represent a specialized form of an encoder-decoder architecture replacing the original $\mathcal{M}_H$. From here, $\mathcal{M}_E$ would serve as the new head $\mathcal{M}'_H$ while the concatenation of $\mathcal{M}_D$ and $\mathcal{M}_T$ would be deployed on the edge server. Conceptually, $\mathcal{M}_E$ is introduced to obtain the compressed form $z'_\ell = \mathcal{M}_E(x)$ prematurely in the network to realize an early optimal offloading point – *bottleneck* – within $\mathcal{M}'$. $\mathcal{M}_D$ on the other hand serves two purposes: (*i*) ensuring that $z' = \mathcal{M}_D(\mathcal{M}_E(x))$ maintains the same spatial dimensions as the original input to $\mathcal{M}_T$, and (*ii*) minimizing the loss incurred by $\mathcal{M}'$ due to the proposed structural modifications of $\mathcal{M}_E$ and $\mathcal{M}_D$. In terms of the latter, techniques inspired by knowledge distillation (KD) have shown tremendous promise in maintaining the accuracy of $\mathcal{M}'$ on par with that of the original $\mathcal{M}$ [10], [14].

C. Blockwise Neural Architecture Search

In this part, we propose to *branch out* an a supplementary model from the primary processing pipeline to be leveraged for both the *main* and *secondary* execution routines as a more energy-efficient alternative. To achieve this, this branched model would comprise *simpler* computing modules than those of the main processing model, reducing the latency and energy consumption overheads from local execution. We denote this simpler model as the *early-exit model*. Figure 2 depicts the early-exit model in the final system architecture as the aggregation of $\mathcal{M}_{\mathcal{D}}$ and $\mathcal{M}_T^{\mathcal{EX}}$, following the shared $\mathcal{M}_{\mathcal{E}}$.

Given their simpler composition, early-exit models are not as accurate as their primary counterparts, and hence, they are only invoked when the input sample belongs to the distribution of canonical samples exhibiting low complexity features. Since numerous AS applications belong to the class of regression problems (e.g., predicting control outputs), conventional approaches for deciding on the the early-exiting decision based on classification confidence estimates are not directly applicable [23]. Instead, we propose to implement the early exit using a modular approach, namely blockwise neural architecture search (NAS) whose advantages are fourfold: (*i*) A modular approach aids in identifying which blocks are the most *sensitive* to alterations with regards to the task at hand, allowing optimizations to be targeted towards the *less-critical* blocks, (*ii*) Customization of search blocks is supported, enabling the inclusion of desired edge computing

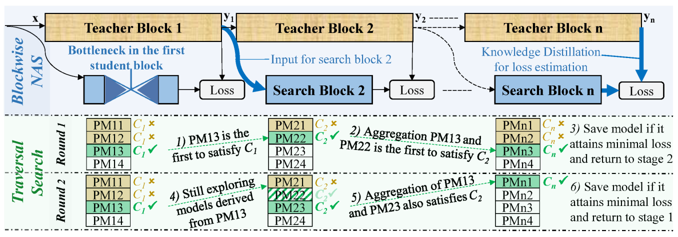

x
Teacher Block 1
y_1
Teacher Block 2
y_2
Teacher Block n
y_n
Blockwise NAS
Bottleneck in the first student block
Input for search block 2
Knowledge Distillation for loss estimation
Loss
Search Block 2
Loss
Search Block n
Loss
Traversal Search
Round 1
PM11 C_1 ✗
PM12 C_1 ✗
PM13 C_1 ✓
PM14
1) PM13 is the first to satisfy C_1
PM21 C_2 ✗
PM22 C_2 ✓
PM23
PM24
2) Aggregation PM13 and PM22 is the first to satisfy C_2
PMn1 C_n ✗
PMn2 C_n ✗
PMn3 C_n ✓
PMn4
3) Save model if it attains minimal loss and return to stage 2
Round 2
PM11 C_1 ✗
PM12 C_1 ✗
PM13 C_1 ✓
PM14
4) Still exploring models derived from PM13
PM21 C_2 ✗
PM22 C_2 ✓
PM23 C_2 ✓
PM24
5) Aggregation of PM13 and PM23 also satisfies C_2
PMn1 C_n ✓
PMn2
PMn3
PMn4
6) Save model if it attains minimal loss and return to stage 1

Fig. 3. Blockwise NAS for edge computing (*top*) and a walk-through example for the traversal search (*bottom*). PM is for partial model and its indices are for the stage and the PM's ranking based on the loss defined in (12).

features as incorporating a bottleneck in the first search block (see Figure 3), (*iii*) The rendered simpler execution path can be leveraged for energy efficiency along both the primary and fail-safe execution paths with minimal impact on the model utility, and (*iv*) A student model's accuracy is not necessarily bound by that of the teacher.

Blockwise NAS using Knowledge Distillation: Neural Architecture Search (NAS) is an established method to automate DNN model design through identifying architecture α^* that achieves the best performance on a target task. Typically, a NAS search space is defined as a large supernet $\mathcal{A}$ with shared parameter weights W, and $\alpha^* \in \mathcal{A}$ is a subnet within. To manage the colossal search overheads, the approach in [24] proposed to divide the search space $\mathcal{A}$ into smaller successive independent supernets $\mathcal{A}_i$ with each block i possessing its shared weights W_i, leading to an exponential reduction in the search space size and the overall design turnaround time. Thus, given inputs X and ground truth values Y, α^* is formed by aggregating N subnets from the search blocks which satisfy:

$$\alpha^* = \arg\min_{\alpha \in \mathcal{A}} \sum_{i=1}^{N} \mathcal{L}_{val}(W_i^*(\alpha_i), \alpha_i; y_{i-1}, y) \tag{9}$$

$$s.t.\ W_i^* = \min_{W_i} \ \mathcal{L}_{train}(W_i, \mathcal{A}_i; y_{i-1}, y_i) \tag{10}$$

where y_{i-1} and y_i represent the inputs and ground truth labels for search block i, respectively. Practically, pre-trained DNN models on the same task can be leveraged as *teachers* to obtain y_i and y_{i-1} from their intermediate data representations at different stages, which allows guiding the search process for each search block i. In words, the main building blocks constituting a DNN architecture, such as the 4 primary blocks of stacked layers in a ResNet architecture [18], are designated as separate teacher blocks, each with its input and output representations utilized as guides for the corresponding search block, as depicted in Figure 3 (top). Therefore using knowledge distillation (KD), the training and validation loss estimates, $\mathcal{L}_{train}$ and $\mathcal{L}_{val}$, between block predictions $\hat{y_i}(\cdot)$ and the teacher ground truth values can be given by:

$$\mathcal{L}_{train}(W_i, \mathcal{A}_i; y_{i-1}, y_i) = \frac{1}{K} ||y_i - \hat{y_i}(y_{i-1})||^j \tag{11}$$

$$\mathcal{L}_{val}(W_i, \mathcal{A}_i; y_{i-1}, y_i) = \frac{1}{K \cdot \sigma^j(y_i)} ||y_i - \hat{y_i}(y_{i-1})||^j \tag{12}$$

in which K is the number of output neurons, $\sigma(y_i)$ is the standard deviation of y_i, and j is for the function degree. The loss estimate in $\mathcal{L}_{val}$ is normalized relative to the corresponding $\sigma^j(y_i)$ to ensure fairness since feature map sizes can differ from one candidate partial model to the other within a search block. Without any loss in generality, we found for experiments that setting j to 2 for $\mathcal{L}_{train}$ (Mean Squared Error) and to 1 for $\mathcal{L}_{val}$ (Mean Absolute Error) worked well.

Model Aggregation under Constraint: After the initial search process has concluded, partial model rankings are rendered for each search block according to $\mathcal{L}_{val}$. If there are no target performance constraints, then the top-ranking partial models from each block can be concatenated to construct the complete DNN model However, as the goal here is to obtain more efficient computational blocks for the early exit, a target performance constraint (e.g., latency) denoted by C_{target} needs to be satisfied. To avoid the prohibitive act of evaluating each possible combination of partial models, we construct a lookup table for the performance costs of each candidate operation within a search block (which in the case of latency are obtained through hardware measurements). Then, we can estimate the maximum allowable cumulative performance cost for each block C_i as:

$$C_i = \sum_{n=1}^{i} cost_n = C_{target} - \sum_{n=i+1}^{N} min\_cost_n \qquad (13)$$

where min_cost_n is the minimum cost for a partial model at block n estimated from the pre-calculated lookup table. Once each block's maximum cost C_i has been estimated using (13), a traversal search can be performed starting from the first search block, and *recursively* going through the partial models of the subsequent blocks as long as the corresponding C_i constraints are satisfied. In other words, the testing of subsequent blocks is skipped if the current partially constructed model at block i has a cumulative performance cost that exceeds C_i. Furthermore, once a model satisfying the constraint has been identified, the search returns to the previous block to avoid testing inferior models [24]. A walk-through example for this traversal search is provided in Figure 3 (bottom).

D. Deployment Hierarchy

Figure 2 illustrates how the final rendered computing modules are to be distributed across the local platform and the edge server for the deployment stage. As shown, $\mathcal{M}_E$ with the optimal offloading point from the first student block is placed locally to be shared by all possible execution paths. Conversely, $\mathcal{M}_D$ is replicated across both execution domains, with the subsequent computing blocks from the original *teacher* model concatenated at the end of $\mathcal{M}_D$ forming the *full* tail model, $\mathcal{M}_T^{Full}$. Furthermore, the local domain possesses an extra local early exit tail $\mathcal{M}_T^{Ex}$ following $\mathcal{M}_D$ as well which is constructed from the remaining *student* blocks, which can be invoked for the *primary* or *fail-safe* operation. All execution paths converge to supply inputs for the following module in the AS pipeline until final predictions are mapped onto physical control outputs. From this arrangement, an ample decision space of operational modes is presented that can be exploited to maximize resource efficiency based on the corresponding deployment conditions.

IV. Runtime Adaptation for Deployment Stage

The challenge for an AS employing edge computing during deployment is to maintain an efficient and reliable operation in

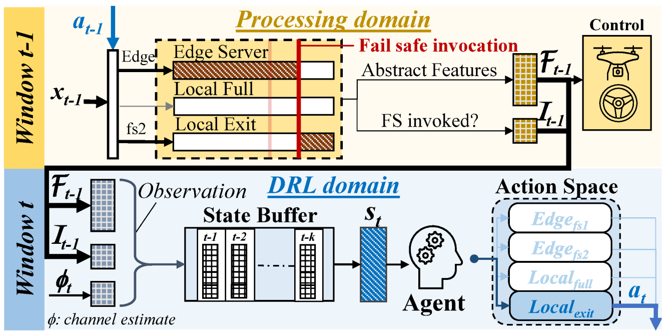


Fig. 4. Information flow between the processing and DRL domains. Mainly, the DRL reuses abstract features and the fail-safe invocation flag from the processing domain at $t-1$ to guide its decision for the window t.

face of the wireless channel uncertainty which can negatively affect performance. To realize a control performance close to the optimal, we propose a learning-based approach that can capture the underlying distributions of the dynamic parameters characterized by complex spatio-temporal patterns so as to tune the operational control knobs at runtime accordingly.

With this setting, we pinpoint 2 crucial guiding principles for any runtime solution implementation: (*i*) it should be light-weighted so that the AS would not incur additional excessive processing overheads while being able to capture the desired contextual information from the environment, and (*ii*) the solution must maintain the application integrity with regards to latency, that is, the mode selection decision should be made available before execution paths diverge.

A. Deep Reinforcement Learning (DRL) Solution

We propose a deep reinforcement learning (DRL) solution to extract contextual knowledge and discern temporal patterns within both the collected data streams (e.g., mounted camera feed) and network conditions to determine the best execution strategy for an AS. The novelty of our DRL implementation is that it follows the aforementioned guiding principles as follows: (*i*) Since AS applications process data samples collected at high sampling frequencies, Strong temporal correlations exist between *successive* samples, and thus the abstract representation obtained from the processing pipeline at time window $t-1$ can be leveraged for constructing the contextual observation of the following time window t, and (*ii*) leveraging abstract representations imply that no exhaustive processing is needed by the DRL, allowing the solution to be compact as desired. Figure 4 illustrates this sequence of information passing over successive windows $t-1$ and t, where the abstract data representations $\mathcal{F}_{t-1}$ and the offloading fail-safe invocation status $\mathcal{I}_{t-1}$ represent the information of interest for the observation at window t as they reflect the data sample's complexity as well as the stability of the wireless network. Formally, the constituents of the DRL solution are as follows:

State Space: the observation at time t is given as $o_t = \{\phi_t, \mathcal{F}_{t-1}, \mathcal{I}_{t-1}\}$, where ϕ_t represents the corresponding probed channel capacity, while $\mathcal{F}_{t-1}$ and I_{t-1} are the respective abstract feature representations and the fail-safe invocation flag at $t-1$. Given how ϕ_t (in Mbps) sustained at the AS can be approximated based on the prior experienced end-to-end latencies and transmission data sizes, it can fail to capture the intricate variations in the networking environment that may delay server response times, and cause the reactivation of the local resources as the secondary execution routines. Thus, through monitoring the fail-safe invocation status, I_{t-1}, we can gain insight into the immediate state of the network in terms of stability to account for the possible lags of ϕ_t. We also include a state buffer that enables stacking observation o_t with the prior k observations to form the final state representation $s_t = \{o_t, o_{t-1}, .., o_{t-k}\}$. The rationale behind having this state buffer is to add another dimension to the temporal correlations between different observations if needed by the target application without complicating the DRL implementation, i.e., keeping it as a lightweight solution.

Action Space: The action space A defines all possible actions that can be taken given a state $s \in S$. In the case of edge computing for AS, A represents the set of all potential execution strategies which we define as $A = \{Edge_{fs1}, Edge_{fs2}, Local_{full}, Local_{exit}\}$, where $Edge_{fs1}$ and $Edge_{fs2}$ are the decisions to execute at the edge server with the distinction in the subscripts, $fs1$ and $fs2$, reflecting the offloading fail-safe choices of $\mathcal{M}_T^{Full}$ or $\mathcal{M}_T^{Ex}$, respectively (see Section III-D). $Local_{full}$ and $Local_{exit}$ are the respective full and early-exit local execution strategies (recall the definition of the early-exit model in Section III-C)

Agent: Through a Q function that can estimate the value of actions for any state s, a DRL agent can identify the optimal action at each t, $a_t = \arg\max_{a \in A} Q_\pi(s_t)$, given s_t under a learnt policy π. To better manage the continuous state space of our environment, we adopt a Deep Q-Learning approach to approximate Q_π by a policy network that is trained to maximize a reward function, $\mathcal{R} = f(\lambda, E_{total}, error)$, which directs the network to select actions that minimize the *error* and energy consumption E_{total} given the trade-off parameter λ as both metrics are impacted by the choice of execution strategy. Specifically, we intend for the DRL agent to learn a function that selects the most convenient action $a \in A$ depending on the perceived scene complexity and the networking conditions, where our rationale is to set the λ parameter to a value that allows the DRL agent to learn a policy that selects the most energy efficient action iff the scene exhibits low complexity characteristics (encoded through the $error$ metric), or else an action that instantiates the full model pipeline will be chosen. E_{total} constitutes the local and transmission overheads as defined in Equation 2, whereas $error$ can be evaluated as the estimated loss function from the already trained full model predictions – that is, the Mean Absolute Error (MAE) between *predictions* and the true labels from a supervised learning dataset. We remark that our interest here is only in the *evaluations* of the loss function to compute the DRL reward for the agent's training. The full and exit models are already trained by this point. Without loss of generality, we employ a Double Deep Q-Network (DDQN) [25] and define $\mathcal{R}$ as:

$$R = -\lambda * E_{total} + -(1-\lambda) * error \qquad (14)$$

B. Training Environment for the Agent

To train the policy network, we need to capture the wireless network uncertainty within the DRL training process through estimates of the channel capacity, ϕ, and abrupt interruptions that trigger the fail-safe represented by the flag, I, as follows:

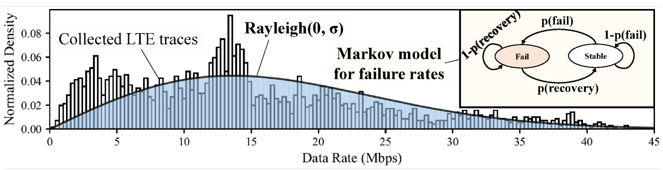


Fig. 5. Fitting rayleigh distribution to collected network traces (*left*), and the markov model for failure rates (*right*)

Channel Modeling: Through collecting traces of channel capacity ϕ, we are able to fit them into a Rayleigh distribution – widely adopted for wireless communications. This would enable approximating channel capacity estimates as independent and identically distributed (i.i.d.) random variables sampled for the DRL training as $\phi \sim Rayleigh(0, \sigma)$, with σ being the scale factor. Figure 5 depicts how we fit a Rayleigh distribution for the LTE traces collected for our experiments.

Failure Rates: Our aim here is to imitate potential random network fluctuations experienced within the AS environment (recall δ in Equation 3) within the DRL environment, where the end-to-end latency when offloading can be pushed beyond the pre-specified deadlines, triggering the secondary local execution routine and changing the system state into what we denote as a *failure state*. From here, we characterize the rate of moving to this *failure state* as failure rates within the emulation environment, and leverage a Markov model comprising two states – *stable* and *fail*. The reasons for adopting this Markov approach are two-fold: (*i*) our primary concern is whether *failure state* has been accessed or not, irrespective of which random delay component from δ caused it, and (*ii*) In actual deployment scenarios, it is difficult to discern in real-time the cause of this transition considering the milliseconds operational scale.

In Figure 5, the transition probabilities between the two states are given by $p(fail)$ and $p(rec)$, with the former capturing the overall failure rate over the duration of operation, whereas the latter is for the probability of recovery and returning to the stable state. $p(rec)$ is defined independent of $p(fail)$ to maintain a degree of randomness over the duration the system remains in a *failure state*, that is, for how many successive time windows does the system incur invocations of the secondary execution routines until it transitions back to the stable state. Consequently, the temporal patterns of network quality degradation experienced in the runtime environment due to motion, terrain and surrounding object variation can be emulated. Hence, we set $p(rec)$ to 0.5 representing an equally likely chance to recover from or remain in the *failure state*.

Algorithm: Algorithm 1 describes how to generate the temporally correlated states to train the DRL. The first component, *correlated inputs* in *line 2*, represents a batch of consecutive data points that share a temporal relation (e.g., consecutive frames from a camera), which is needed by the DRL to learn how to estimate the data complexity for a sample i using the abstract representation of the preceding sample, $\mathcal{F}_{i-1}$. In *lines 4-8*, the corresponding data rate estimate, ϕ_i, and failure state, m_i are sampled and used to construct the corresponding observation, which can be aggregated with the prior k observations to construct an ensemble state. With these states, a typical DRL training procedure in *lines 9-15* is performed where the agent interacts with the environment through exploration and exploitation, and samples experiences from its replay buffer to train its policy network until convergence [25].

V. Experiments and Results

A. Experimental Setup

We evaluate TESTUDO on two AS applications instantiating different requirements and computational demands: end-to-end control for autonomous driving and object detection in modular UAV pipelines. Our setup is detailed as follows:

Datasets: We use the Carla conditional imitation learning dataset [26] and the Au-Air UAV dataset for low altitude traffic surveillance [27] – the former instantiating the imitation learning class of problems while the latter is for object detection as part of a modular processing pipeline. The Carla dataset has been collected using the CARLA urban driving simulator [28] and is divided into 657,800 and 74,800 respective training and validation image frames, each of which is coupled with control/sensor outputs. The Au-Air dataset contains 32,823 labeled frames extracted from 8 recorded video clips with 132,034 object instances belonging to 8 object categories related to traffic surveillance. To arrange frames as *correlated inputs* (see Algorithm 1), groups of temporally correlated frames – 200 for Carla and 60 for Au-Air – are aggregated into separate clips as inputs. With this arrangement, we divide the Au-Air dataset into 90% training and 10% validation sets.

Benchmarking: We compare TESTUDO against Sage [14] and Hydra [4] strategies employing collaborative edge computing with offloading fail-safe measures – the former for end-to-end control in ADS while the latter for UAV navigation. In brief, Sage's operational principle is to determine each time window, t, whether to execute locally or offload to an edge server based on whichever action is perceived to be more energy-efficient given a critical latency threshold. Their fail-safe is to re-invoke the remainder of the full local execution pipeline whenever the threshold $T - L_{FS}$ is reached based on corresponding estimates of the data rate, ϕ. As for Hydra, it employs a more lenient deadline policy that can tolerate missing δ successive deadlines. Specifically, it entails *two operational modes* when connected to a single remote edge server: (*i*) *Performance (P);* where computation is delegated to the server for energy efficiency as long as the execution deadline, T, is met, and (*ii*) *Reliability (R)*; where the local pipeline is activated as a fail-safe alongside the remote execution pipeline when δ successive deadlines are missed. Once the the latency of the remote server execution falls below T once more, Hydra switches back to *Performance (P)* mode. We follow their specifications and perform our analysis using 720×1280 (720p) and 360×640 image sizes for the respective ADS and UAV experiments, and set their critical deadlines, T, to 100 and 150 ms, respectively.

Blockwise NAS: We train and use ResNet-50 and ResNet-18 architectures [18] as the teacher models. These architectures comprise ResNet backbones, supplemented with relevant backends according to the AS application – conditional imitation learning component for vehicle control [26] and the region proposal network components from Faster R-CNN for object detection [29]. To facilitate upcoming comparisons, we keep the architectural parameters of the earliest student block with the bottleneck in accordance with the specifications in [10],

Algorithm 1: DRL Training Environment

```
Input: Temporal constant: $k$, Rayleigh distribution scale: $\sigma$
Initialize $O\_queue(k), S\_list, m_0$
// construct temporal states
for $correlated\_inputs$ in $dataset$ do
    for $i = 1$ to $len(correlated\_inputs)$ do
        $\phi_i \sim Rayleigh(0, \sigma)$                    // sample $\phi$
        $m_i \leftarrow Markov(m_{i-1})$                  // failure state
        $O\_queue.push(\phi_i, \mathcal{F}_{i-1}, m_{i-1})$   // moving window
        if $len(O\_queue) >= k$ then
            $S\_list.append(\forall observation \in O\_queue)$

// training procedure
for $t$ to $len(S\_list) - 1$ do
    // exploit/explore based on $\epsilon$-decay
    $a_t = \arg\max_{a \in A} Q_\pi(s_t)$ or random action
    Take action $a_t$, observe reward $r_t$ and next_state $s_{t+1}$
    $e_t = (s_t, a_t, r_t, s_{t+1})$              // experience tuple
    $Store\_Experience(e_t)$                 // store experience
    $e_j = Sample\_Experiences()$             // random batches
    $Q_{loss} = DDQN(e_j)$                   // calculate loss
    $Q_\pi \leftarrow Q_{loss}$                  // update policy network
```

TABLE I
SEARCH SPACES FOR THE EARLY-EXITS. EXPANSION RATIO IS A MULTIPLIER FOR SHRINKING #CHANNELS WITHIN STUDENT BLOCKS

	#Student Blks	#Layers	Kernel	Expansion Ratio
ResNet-18	1	2,3	3,5,7	$\frac{1}{4}$, $\frac{3}{8}$
ResNet-50	2	2,3,4	3,5,7	$\frac{1}{8}$, $\frac{1}{4}$

where we use 3 and 6 output channels at the bottleneck offloading point ($\mathcal{M}_E$ output) for the respective applications, leading to data transmission sizes $64\times$ and $32\times$ less than that of the inputs, respectively (We refer the interested reader to [10] for more details). These student blocks with bottlenecks are trained using blockwise KD from the first two corresponding ResNet teacher blocks. The remaining two ResNet teacher blocks are used to guide the blockwise NAS search for implementing the early-exit model with the search space descriptions provided in Table I. Thanks to the modularity, we only needed to run 3 search epochs for each teacher-student combination, with a learning rate of 2×10^{-3} and a batch size of 4, taking less than a day in total on a desktop machine with Nvidia 2070 GPU. For the final traversal search, we characterize C_{target} as the execution latency of the local exit model, and we set it to $0.8\times$ that of the local full model. The best performing model from the NAS satisfying C_{target} is rendered as the exit model.

Scoring Metrics: End-to-end vehicle control is evaluated based on the Mean Absolute Error (MAE) between model predictions and ground truths for control outputs – steering, acceleration pedal, and braking pedal angles. For scoring models on object detection, we use the prominent mean Absolute Precision (mAP) with an intersection over union (IoU) threshold of 0.5 following the Au-Air dataset paper [27].

Performance Evaluation: We use an Nvidia Drive PX2 AutoChauffeur and an Nvidia Jetson Nano as our hardware computing platforms for the ADS and UAV, respectively. To maximize hardware performance efficiency and benchmark local execution latency overheads, we leverage the Nvidia TensorRT library [30] to compile our DNN models as highly optimized inference engines on both experimental Nvidia platforms. Mainly, we keep the default TensorRT settings in which each first visible GPU on the respective platform is designated as the target inference unit – that is, the integrated Pascal-based GPU on the TegraX2 SoC for the Drive PX2 and the Maxwell-based GPU for the Jetson Nano. We ensure that no other applications are utilizing the hardware resource during the benchmarking process. Additionally, in order to estimate the local execution power, P_{exec}, we adopt a system-level approach and monitor the difference in consumed power on the Drive PX2 during *execution* and *idle* times, which on average reaches $\approx$ 7 W when our deployed DNNs are invoked on the target onboard GPU. For the Jetson Nano, we set P_{exec} to 4 W in conjunction with a similar analysis performed in [4]. Thus, we can have a common ground for directly comparing the execution and offloading overheads without the effects of the usual 'on' power. Predominantly, we can now evaluate the local energy consumption overheads $E_{exec} = L_{exec} \cdot P_{exec}$ in equation (2) using the aforementioned values. Whereas for E_{comm}, we follow the practice adopted by the relevant related collaborative intelligence works [5]–[7], [14], and use the 4G LTE and WiFi data transfer power consumption models in [31] to obtain estimates for the communication power, P_{comm}.

DRL: Our DRL model comprises 5 fully connected layers. The first 4 layers take as input the abstract features, $\mathcal{F}_{t-1}$, map them to a lower-dimensional representation, before concatenating them with $\mathcal{I}_{t-1}$ and ϕ_t to be inputted to the final layer. In terms of added cost, the transmission overhead of $\mathcal{F}_{t-1}$ from the edge server back to the local DRL domain – if needed – is negligible and can be aggregated with the returned results. For perspective, $\mathcal{F}_{t-1}$ size is $\approx$ 0.5 kB for the ADS application incurring a transmission cost of $<$ 1 ms at 5 Mbps. In terms of the DRL architecture itself, it occupies $\approx$ 0.6 MB (40$\times$ less than our smallest processing model) and completes execution in $<$ 1 ms, satisfying the requirement to provide an output decision prior to the offloading point at the encoder's output $\mathcal{M}_E(\cdot)$ – which approximately takes 12 ms. We also performed empirical evaluations for the DRL's agent performance at $k = 1, 2$, and found that for both our experimental datasets, setting $k = 1$ worked fine.

B. End-to-end Vehicle Control Experiments

Architectures Evaluation: We compare different ResNet architectures with regards to the experienced MAE and latency to process input images of 720p resolution in Table II. We denote our trained baselines – the teachers – as *ResNet*$_{base}$, the primary model with the distilled bottleneck as *ResNet*$^{full}_{w/BN}$, and the exit model from the block-wise NAS as *ResNet*$^{exit}_{w/BN}$. Our trained baseline *ResNet-50*$_{base}$ model achieved an average MAE of 0.033 compared to the 0.032 achieved by the version in [14], whereas its latency processing overhead reached 140.16 ms – exceeding the 100 ms response requirement in [3], [12]. Once the bottleneck structure has been integrated, the execution latency drops to 89.66 ms for the *ResNet-50*$^{full}_{w/BN}$ version. Even more so, the average MAE drops to 0.03, asserting how student architectures are not bounded by their teacher's performance. The top-performing model with an early-exit tail incurred a latency execution overhead of 70.91 ms, satisfying C_{target} as defined above. The average MAE for the exit model *ResNet-50*$^{exit}_{w/BN}$ is slightly more than the full version reaching around 0.033. Similar trends were

TABLE II
COMPARING BLOCKWISE MODELS AND THEIR BASELINES FOR THE CARLA SELF-DRIVING DATASET ON THE NVIDIA PX2 PLATFORM

Model	MAE$\times 10^{-3}$				720p input latency(ms)
	Steer	Acc.	Brake	Avg.	
ResNet-50 [14]	26.0	51.4	18.0	31.8	–
ResNet-50$_{base}$	26.0	53.6	19.8	33.1	140.16
ResNet-50$^{full}_{w/BN}$	25.7	45.8	18.9	**30.1**	**89.66**
ResNet-50$^{exit}_{w/BN}$	25.9	52.2	19.5	**32.5**	**70.91**
ResNet-18$_{base}$	25.9	51.5	19.4	32.3	60.3
ResNet-18$^{full}_{w/BN}$	20.1	43.3	18.0	**27.1**	**41.94**
ResNet-18$^{exit}_{w/BN}$	25.9	54.5	19.0	**33.1**	**33.48**

observed for the ResNet-18 architectures, where the average MAE dropped from *ResNet-18*$_{base}$ to *ResNet-18*$^{full}_{w/BN}$ (0.032 to 0.027), and increased back for *ResNet-18*$^{exit}_{w/BN}$ to 0.033. The full and exit ResNet model are to be arranged as described in Section III-D, forming, in turn, the DRL action space.

DRL Performance Evaluation: Firstly, we associate the *robustness* error parameters in Equation (7) with the MAE evaluations, and through empirical evaluations on the training dataset, we find that setting err^{th} to 0.02 offers good energy optimization opportunities from leveraging the secondary exit path with little-to-no effect on MAE. In accordance with this setting, we define a reference optimal strategy to select the most energy-efficient action as long as the MAE difference between the full and exit models is less than 0.02 (i.e, $\Delta MAE < 0.02$), or else actions instantiating the full model pipeline would be selected. In Figure 6, we compare the energy consumption at runtime of our proposed DRL solution against other strategies: *Local*, *Sage* [14], and the reference *optimal* strategy. All energy evaluations are normalized with respect to that of pure local execution, and we repeat the analysis for two DRL policies (at $\lambda = 0.01$ and 0.05) trained at k=1 for each ResNet architecture. These λ values were chosen after performing some parametric sweeps on both the logarithmic and linear scales, with the purpose of prioritizing robustness over energy efficiency as stated in Section IV-A. Given LTE communication and failure rates of 1%, 10%, and 20%, we observe that the policies learned by our DRL solutions consistently outperform both the local and Sage strategies in terms of energy efficiency across all scenarios. Most notably, the energy consumption for the ResNet-50 at $\lambda = 0.05$ dropped by up to 31% and 23.3% from SAGE for the 1% and 20% failure rates, respectively.

For the ResNet-50s, mapping policies learned by the DRL lowered the average experienced MAE across the 3 failure rates (by a factor up to 1.2%) from the other two strategies that only utilize *ResNet-50*$^{full}_{w/BN}$. This points out how the DRL not only learns how to leverage the exit pipeline for energy efficiency, but also to improve its predictions on a *per-sample basis*, exploiting $\mathcal{F}_{t-1}$ to determine which samples are simple enough to map to the early-exit model. For the ResNet-18 architectures, the energy consumption still drops even for the MAE-oriented policy at $\lambda = 0.01$ by respective 1.7%, 9.6%, and 12.8%. However, the larger discrepancy in MAE between the full and exit ResNet-18 models (see in Table II) makes it harder for the DRL to decrease MAE through the learned mapping policies. Still at $\lambda = 0.01$, the DRL is capable of maintaining the MAE difference from strategies that only

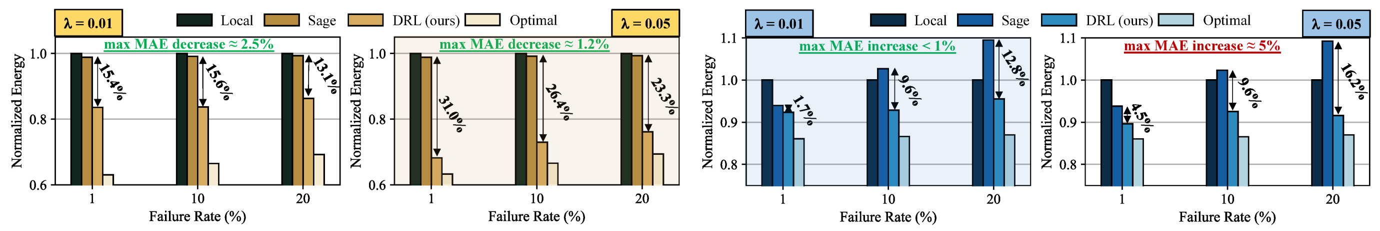


Fig. 6. Benchmarking in terms of energy savings given when LTE is supported for ResNet-50 (*brown*) and ResNet-18 (*blue*). Evaluations are normalized with respect to that of local execution. Shaded backgrounds indicate the better λ options for MAE – evaluated on the Carla imitation learning dataset [26].

utilize *ResNet-18*$^{full}_{w/BN}$ within $< 1\%$ for all failure rates.

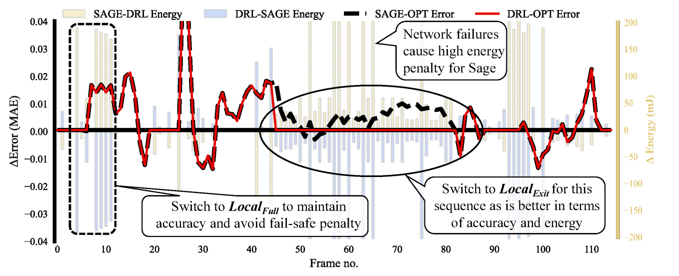


Fig. 7. Comparing the DRL and Sage behaviors over a sequence of correlated frames in terms of difference in MAE from the reference optimal policy. The background bar plot shows their the per-frame energy consumption difference.

Policy Analysis: We perform a closer in-depth analysis of the policy learned by the DRL for the ResNet-18 at $\lambda = 0.01$ for the 20% failure rate scenario. In Figure (7), we compare the behavior of the DRL policy against that of Sage over a clip sequence of correlated input frames from Carla's evaluation dataset. Each frame is associated with a corresponding ϕ and failure state m from the DRL emulation environment (as defined in Section IV-B). All MAE estimates are referenced against that of the optimal policy (OPT) defined above and depicted by the solid horizontal black line, whereas the bar plot in the background indicates the energy consumption difference between Sage and the DRL for every indexed frame (as shown in the legend). For instance, a negative value for DRL-SAGE at a certain frame indicates that DRL is more energy efficient by the difference amount in mJ. Throughout the trace, we observe that the DRL is able to recognize patterns of abrupt network interruptions, reverting to local execution to avoid the fail-safe high energy penalties. On top of that, the period (frames 45-80) over which the DRL behavior is the same as the optimal demonstrates how it also learned to exploit the local exit to improve on both the error and energy objectives.
We also breakdown the action selection frequency of the DRL for the ResNet-18 at $\lambda = 0.01$ in Figure 8 over the different failure rates. As illustrated, Higher failure rates lead the DRL to opt for more local execution actions. Despite this, the ratio between attempted offloads that suffered an offloading fail-safe invocation by both the DRL and Sage almost remains the same across the failure rates at around $\approx 5\times$ less.

Analysis using WiFi: We repeat our experiments for the more power-efficient WiFi technology [31]. In Table III, the energy gains for the ResNet-50 architecture remain substantial reaching 36.98% and 19.85% over Sage while dropping the MAE by 1.68% and 1.25% for the 1% and 20% failure rates, respectively. For the ResNet-18, the MAE-energy trade-off problem becomes more complicated as their slimmer struc-

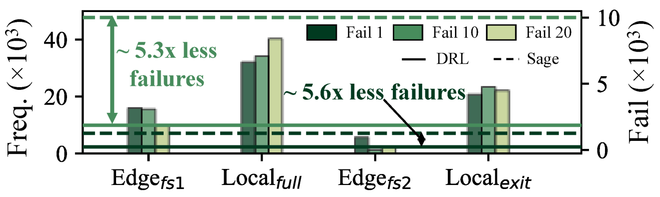


Fig. 8. DRL Action selection frequency for the ResNet-18 at λ=0.01 over the Carla evaluation dataset [26] and the same sequence of sampled ϕ estimates.

ture and the negligible WiFi transmission overhead massively shrink the energy penalty for the early-exit fail-safe, making it a more appealing action choice, and subsequently leading MAE to increase (as the 2.22% increase for $\lambda = 0.1$ at 20% case). At ($\lambda = 0.01$), the MAE difference at 20% failure to $< 1\%$ at the expense of a 0.92% increased energy consumption over Sage. From here, we can deduce that the extent of performance gains is mainly dependent on the underlying model structure and the supported wireless technology.

C. Object Detection for UAV Experiments

Architectures Evaluation: We again train two baseline ResNet architectures, ResNet$_{base}$, to guide the design process of the primary, ResNet$^{full}_{w/BN}$, and early-exit, ResNet$^{exit}_{w/BN}$, models for object detection. In Table IV, we compare the mAP evaluations on the Au-Air dataset as well as the latency execution overhead on the Jetson Nano for 360×640 inputs. Typically, with each added optimization, the model execution overhead is lowered, reaching primary/exit latency combinations of 98.84/78.96 ms and 42.21/31.38 ms for the ResNet-50 and ResNet-18, respectively. In terms of mAP, we first remark for perspective that the mAP values estimates for models in the original Au-Air dataset paper [27] ranged between 22∼30%. Our distilled *ResNet-50*$^{full}_{w/BN}$ reached 29.9% mAP – improving over its baseline teacher (which achieved 28.8% – which indicates how *ResNet-50*$^{full}_{w/BN}$ generalizes better with regards to the object detection task, whereas the mAP of the exit model *ResNet-50*$^{exit}_{w/BN}$ is slightly less at 27.1%. For the ResNet-18, the mAP evaluations degrade slightly with each incorporated optimization – from 27.3% to 26.8% to 26.3%.

DRL Performance Evaluation: Once more, we associate the *robustness* condition error parameters with the MAE loss, and also find that setting err^{th} to 0.02 also works for this set of experiments. We train a DRL policy at $\lambda = 0.5$ on the object detection losses and compare it against *Sage*, *Hydra*, and *local* strategies with regards to the change in mAP and energy consumption in Table V. As all the competitor strategies employ a single model pipeline, we first demonstrate – prior to assessing energy gains – how the input mapping function

TABLE III
REDUCTION IN MAE AND ENERGY COMPARED TO OTHER STRATEGIES GIVEN THE NVIDIA PX2/WIFI SETUP USING THE CARLA DATASET [26]

Model	Fail (%)	λ	MAE Red. %		Energy Red. %	
			[14]	local	[14]	local
ResNet50	1	0.1	**1.68**	1.68	**36.98**	60.02
	20	0.05	**1.25**	1.25	**19.85**	39.51
ResNet18	1	0.8	**-0.85**	-0.85	**0.42**	35.19
	20	0.1	-2.22	-2.22	3.37	19.04
	20	0.01	**-0.30**	-0.30	**0.92**	16.95

TABLE IV
COMPARING MODELS PERFORMANCE ON THE AU-AIR OBJECT DETECTION DATASET FOR UAV [27] ON THE NVIDIA JETSON NANO

Model	Metric	ResNet$_{base}$	ResNet$^{full}_{w/BN}$	ResNet$^{exit}_{w/BN}$
ResNet50	mAP	28.8	**29.9**	**27.1**
	Lat. (ms)	122.06	**98.84**	**78.96**
ResNet18	mAP	27.3	**26.8**	**26.3**
	Lat. (ms)	55.99	**42.21**	**31.38**

learnt by the DRL to execution pipelines can improve mAP scores over strategies that employ a single execution pipeline – be it the *full* or *exit* model. As shown in Table V, the DRL's dynamic input mapping consistently improves mAP scores compared to any strategy employing a single *exit* pipeline. However, mAP percentage changes are minute compared to the *full* model. For instance, the DRL offered slight mAP improvements (up to 0.54%) for the ResNet-18 architecture and sustained minor drops (reaching -0.66%) for the ResNet-50 as the failure rate increased. The increase or decrease in mAP score can be attributed to the difference in prediction scores between the *full* and *exit* pipelines in each architecture (see Table IV), as closer mAP scores between both ResNet-18 model allowed the DRL policy to better converge on the *error* objective, and improve the average mAP scores. Even for the ResNet-50, the fact that the max mAP drop is 0.66% out of a maximum possible of 9.4% (the %drop in mAP from ResNet$^{full}_{w/BN}$ to ResNet$^{exit}_{w/BN}$ in Table IV) indicates how the DRL still managed to learn an effective mapping function.

For energy consumption comparisons, we first reiterate that Hydra [4] employs a soft-deadline policy that tolerates missing δ consecutive T deadlines before activating the local pipeline alongside the remote one. The local pipeline is deactivated once more when the total execution latency drops below T. In the table, our DRL solution for the most part outperforms Hydra at δ=1 by up to 10.59% and 13.42% for the ResNet-50 and ResNet-18, respectively, except for the ResNet-50 1% failure rate case, in which our DRL is 7.75% less efficient. This is attributed to two things: (*i*) the fail-safe execution overhead in the ResNet-50 is high, and (*ii*) At lower failure rates, Hydra scarcely invokes redundant local executions. Nonetheless, their soft deadlines policy caused 38 out of the 3240 validation frames to miss their deadlines. As the deadline constraint gets looser (δ=2), more frames miss their deadlines, and more energy consumption overhead is incurred by the DRL relative to Hydra. Compared to Sage, our energy gains are not as much as in the previous experiment because of the more relaxed T=150 ms, which has facilitated both more offloads and more slack time for receiving server responses before triggering the fail-safe, making Sage a viable option here at low failure rates.

Policy Analysis: In Figure 9, we take a more in-depth look at how our learnt DRL policy behaves compared to Hydra ($\delta = 1$) for the ResNet-18 over a sequence of 6 correlated frames from the evaluation dataset at 10% failure rate. For each frame, we show the mAP scores for the full and exit models to gain a better insight into the DRL's choices. At first, both the DRL and Hydra are opting for their offloading mode as it represents the most energy-efficient option. For the following frame at t=1, the network starts to experience interruptions, which initially led the DRL to incur an additional energy penalty due to the fail-safe invocation, whereas Hydra missed its deadline and switched to Reliability (*R*) mode for the following frame at t=2. Through observing ($\mathcal{F}_{t-1}$), the DRL can discern input scene complexity, and understands from the first two frames that the exit pipeline can serve as a better candidate for this corresponding input stream, which we observe for t=2 as it attempts offloading once more but with the exit model as the fail-safe. Seeing network interruptions persist, the DRL designates the $Local_{exit}$ as the primary mode of operation to maximize energy efficiency. Once connectivity is restored, both strategies switch back to offloading modes. All in all, our DRL achieved 168 mJ net energy gains over Hydra and an average mAP of 81.9% compared to the 81.7%.

VI. DISCUSSION

A. Overall Findings

Understanding the expected deployment conditions, integrating the offloading fail-safe overheads as part of the performance models, and encoding network disparities through a metric like failure rates enables designing a more effective run-time solution that is capable of better management of energy resources. From our experiments, although TESTUDO generally achieved better performance efficiency for edge computing AS applications with hard execution deadlines, less-critical applications can benefit from more tolerant strategies, such as Hydra, employing softer deadline policies that accept missing one or two deadlines before invoking redundant execution. Additionally, despite the changes in prediction errors being mild ($< 1\%$ at the worst), the DRL solution can be curtailed for absolute robustness – that is, no mAP changes – through reducing its action space to offloading decisions only.

We also find that the blockwise NAS is extremely efficient in designing both the primary and exit pipelines due to the dramatically reduced search spaces which rendered exit models of performances close to their primary counterparts. In summary, we found that the degree of effectiveness of our DRL solution dependent on combinations of (*i*) input sizes; for offloading is more relevant as the size of an input (processing load) increases, (*ii*) Choices of T and L_{FS}; for larger fail-safe overheads close to T (as in T=100 for the ResNet-50) makes local execution strategies much more appealing, (*iii*) Hardware; for local execution overheads are affected accordingly, and (*iv*) wireless technology; for some technologies incur a more power-efficient data transfer than others [31]. For example, the 720p inputs, ResNet18, T=100, Nvidia PX2, and WiFi combination in the end-to-end vehicle control experiments did not offer any improvements over the state-of-the-art, unlike other combinations.

TABLE V
COMPARING THE % CHANGE IN MAP AND ENERGY CONSUMPTION FOR THE DRL ON THE OBJECT DETECTION TASK FOR THE AU-AIR DATASET [27] FOR JETSON NANO AND WIFI. HYDRA [4] ALLOWS SUCCESSIVE δ DEADLINES TO BE MISSED BEFORE REDUNDANTLY ACTIVATING THE LOCAL PIPELINE.

Model	Fail (%)	mAP Inc. %		Energy Red. %				Missed Deadlines	
		Full	Exit	Hydra [4]		Sage [14]	local	Hydra [4]	
				$\delta = 1$	$\delta = 2$			$\delta = 1$	$\delta = 2$
ResNet50	1	**0.03**	2.0	**-7.75**	-13.55	-8.94	383.02	**38**	60
	10	**-0.66**	0.85	**3.66**	-26.87	-2.25	180.23	**257**	422
	20	**-0.65**	0.53	**10.59**	-23.45	3.40	117.57	**440**	720
ResNet18	1	**-0.0**	0.07	**1.58**	-0.55	0.25	125.19	**34**	54
	10	**0.54**	0.15	**10.56**	-4.21	2.16	72.85	**288**	450
	20	**0.23**	0.35	**13.42**	-7.39	1.79	48.16	**463**	671

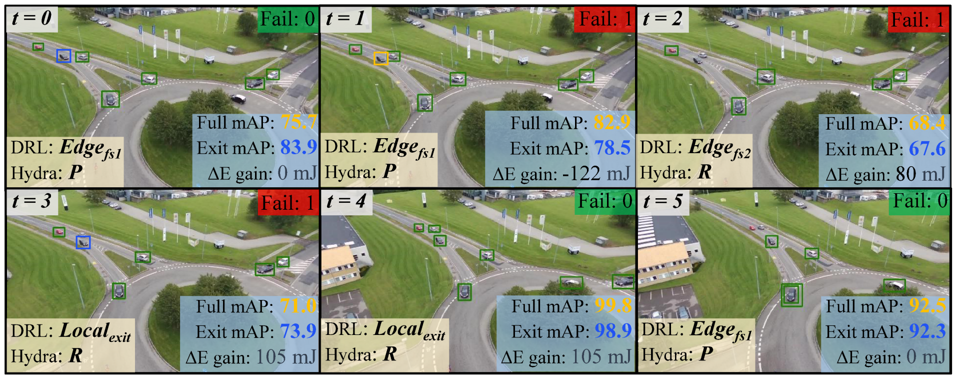


Fig. 9. Comparing DRL and Hydra decisions over a sequence of 6 correlated frames from the runtime evaluation dataset using Au-Air [27] and the network failure state for ResNet-18's full and exit models. ΔE gain is the per-frame energy consumption difference between Hydra and the DRL. Green bounding boxes indicate objects detectable by both models, blue ones are for objects detected by the exit only, and yellow ones are for objects detected by the full only.

B. Study Limitations and Future Works

We performed our analysis in this paper using models from the existing wireless infrastructure – WiFi and 4G LTE. However, we concur that evaluations using 5G and V2X communication protocols are still needed as they will be instrumental in the real-world adoption of connected and autonomous systems [9]. Given such scenarios, the problem can be further scaled to the effective management of an assembly of AS (e.g., a fleet of connected autonomous vehicles) – possibly entailing different dynamics of communication. Also relevant is how the extent of energy gains is dependent on how the local execution power P_{exec} is evaluated, which was through a system-level perspective in this work. A more granular approach could lead to more insights on the effectiveness of our approach.

Furthermore, the values of T used here represent worst-case bounds as industry standards employ tighter bounds for safety – as strict as 30 Hz ($\sim$33 ms) for an ADS. From our analysis, ResNet-18 derived architectures are more poised to operate around these tighter constraints. In a similar vein, although TESTUDO has shown promising performance gains, real-world experiments are still needed for more tangible experiences of network instability due to the various dynamic factors (e.g., motion characteristics), which are extremely unpredictable and hard to model accurately. We also remark that relatively small datasets have limited the DRL exposure to more experiences of diverse contextual information. Also, more sophisticated hardware platforms can instigate more complex architectures and multi-sensory pipelines that require further evaluations on end-to-end workloads in terms of offloading key kernels – potentially benefiting more from offloading optimizations [16].

On another note, our analysis assumes that the AS always operates within the vicinity of the original edge server it delegated its processing to. In real-world settings, the AS may be moving away from the edge server and entering into the domain of a different edge server. In such cases, the abstract random function, δ, can be a substandard characterization of these motion aspects, as δ merely abstracts random latencies with no consideration to their source, the spatio-temporal aspects of AS motion, or even the inner tidings at the edge server side in terms of task mapping, hand-offs and relays. One way that can aid in addressing this is through monitoring the Received Signal Strength Index (RSSI) periodically at the local AS platform, and involving it in the state observation for the DRL as indicators of the mobility patterns. Still, experiments need to be performed on both the simulation and real-world settings to evaluate the efficacy of such approaches. We leave these problems for future research works to address.

VII. CONCLUSIONS

We presented TESTUDO, a methodology for reliable and efficient edge computing for AS operating under stringent execution deadlines. Firstly, TESTUDO encompasses a modular approach for designing efficient DNN computing pipelines for edge computing supporting optimal offloading points and fail-safe mechanisms using blockwise NAS and KD techniques. Then at runtime, a DRL solution is provided to adapt the execution mode according to the sample complexity and the network status. Our experiments for end-to-end control and object detection have demonstrated that TESTUDO achieved energy gains up to 31% and 13.4% with averages of 15.9% and 5.3%, respectively, with $<$1% change in prediction scores.

REFERENCES

[1] S. Abuelsamid, "Nvidia Cranks Up And Turns Down Its Drive AGX Orin Computers," *Forbes*, Jun 2020. [Online]. Available: https://www.forbes.com/sites/samabuelsamid/2020/05/14/nvidia-cranks-up-and-turns-down-its-drive-agx-orin-computers

[2] "All new Teslas are equipped with NVIDIA's new Drive PX 2 AI platform," https://electrek.co/2016/10/21/all-new-teslas-are-equipped-with-nvidias-new-drive-px-2-ai-platform-for-self-driving, 2016.

[3] S.-C. Lin, Y. Zhang, C.-H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars, "The architectural implications of autonomous driving: Constraints and acceleration," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 751–766.

[4] D. Callegaro, S. Baidya, and M. Levorato, "Dynamic distributed computing for infrastructure-assisted autonomous uavs," in *ICC 2020-2020 IEEE International Conference on Communications (ICC)*. IEEE, 2020.

[5] Y. Kang *et al.*, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge." Association for Computing Machinery, 2017.

[6] B. Zamirai, S. Latifi, P. Zamirai, and S. Mahlke, "Sieve: Speculative inference on the edge with versatile exportation," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[7] D. J. Pagliari, R. Chiaro, Y. Chen, S. Vinco, E. Macii, and M. Poncino, "Input-dependent edge-cloud mapping of recurrent neural networks inference," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[8] M. Odema, N. Rashid, B. U. Demirel, and M. A. Al Faruque, "LENS: Layer Distribution Enabled Neural Architecture Search in Edge-Cloud Hierarchies," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 403–408.

[9] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang, and W. Shi, "Edge computing for autonomous driving: Opportunities and challenges," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1697–1716, 2019.

[10] Y. Matsubara, D. Callegaro, S. Baidya, M. Levorato, and S. Singh, "Head network distillation: Splitting distilled deep neural networks for resource-constrained edge computing systems," *IEEE Access*, vol. 8, pp. 212 177–212 193, 2020.

[11] A. E. Eshratifar, A. Esmaili, and M. Pedram, "BottleNet: A Deep Learning Architecture for Intelligent Mobile Cloud Computing Services," in *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2019, pp. 1–6.

[12] S. Baidya, Y.-J. Ku, H. Zhao, J. Zhao, and S. Dey, "Vehicular and edge computing for emerging connected and autonomous vehicle applications," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[13] S. Liu, J. Tang, Z. Zhang, and J.-L. Gaudiot, "Computer architectures for autonomous driving," *Computer*, vol. 50, no. 8, pp. 18–25, 2017.

[14] A. Malawade, M. Odema, S. Lajeunesse-DeGroot, and M. A. Al Faruque, "SAGE: A Split-Architecture Methodology for Efficient End-to-End Autonomous Vehicle Control," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 5s, pp. 1–22, 2021.

[15] D. Callegaro, M. Levorato, and F. Restuccia, "SeReMAS: Self-Resilient Mobile Autonomous Systems Through Predictive Edge Computing," in *IEEE Intl. Conf. on Sensing, Communication and Networking*, 2021.

[16] L. Chen, M. Odema, and M. A. A. Faruque, "Romanus: Robust task offloading in modular multi-sensor autonomous driving systems," *arXiv preprint arXiv:2207.08865*, 2022.

[17] Y. C. Yeh, "Triple-triple redundant 777 primary flight computer," in *1996 IEEE Aerospace Applications Conference. Proceedings*, vol. 1. IEEE, 1996, pp. 293–307.

[18] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2016, pp. 770–778.

[19] (2021) Tesla Autopilot. [Online]. Available: https://www.tesla.com/autopilot

[20] J. Hawke, R. Shen, C. Gurau, S. Sharma, D. Reda, N. Nikolov, P. Mazur, S. Micklethwaite, N. Griffiths, A. Shah *et al.*, "Urban driving with conditional imitation learning," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2020, pp. 251–257.

[21] "Udacity. An Open Source Self-Driving Car," https://www.udacity.com/self-driving-car, 2022.

[22] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada, "An open approach to autonomous vehicles," *IEEE Micro*, vol. 35, no. 6, pp. 60–68, 2015.

[23] S. Teerapittayanon, B. McDanel, and H.-T. Kung, "Branchynet: Fast inference via early exiting from deep neural networks," in *2016 23rd International Conference on Pattern Recognition (ICPR)*. IEEE, 2016.

[24] C. Li, J. Peng, L. Yuan, G. Wang, X. Liang, L. Lin, and X. Chang, "Block-wisely supervised neural architecture search with knowledge distillation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 1989–1998.

[25] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016.

[26] F. Codevilla, M. Müller, A. López, V. Koltun, and A. Dosovitskiy, "End-to-end driving via conditional imitation learning," in *2018 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2018, pp. 4693–4700.

[27] I. Bozcan and E. Kayacan, "Au-air: A multi-modal unmanned aerial vehicle dataset for low altitude traffic surveillance," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, 2020.

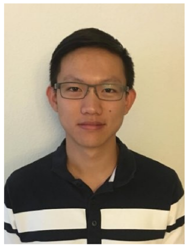
[28] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "Carla: An open urban driving simulator," in *Conference on robot learning*. PMLR, 2017, pp. 1–16.

[29] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," *Advances in neural information processing systems*, vol. 28, 2015.

[30] https://developer.nvidia.com/tensorrt.

[31] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck, "A close examination of performance and power characteristics of 4g lte networks," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*, 2012, pp. 225–238.

Mohanad Odema received the B.Sc. degree in Communications and Electronics Engineering and the M.Sc. degree in Computer Engineering from Ain Shams University, Cairo, Egypt in 2014 and 2018, respectively. He is currently pursuing the Ph.D. degree in Computer Engineering with the University of California at Irvine (UCI). His current research interests are focused on design methodologies for robust and efficient edge computing using dynamic neural network architectures, especially for autonomous systems and mobile health applications.



Luke Chen (Graduate Student Member, IEEE) received his B.S. (2019) in Electrical and Computer Engineering from the University of California Irvine (UCI) in Irvine, CA, USA, where he is currently pursuing a Ph.D. degree in Electrical and Computer Engineering with a focus in Computer Engineering at UCI. His current research explores dynamic neural networks, edge-cloud-split architectures, and distributed computing for wearables and autonomous systems.

Marco Levorato is an Associate Professor in the Computer Science department at UC Irvine. He completed the PhD in Electrical Engineering at the University of Padova, Italy, in 2009. Between 2010 and 2012, he was a postdoctoral researcher with a joint affiliation at Stanford and the University of Southern California. His research interests are focused on distributed computing over unreliable wireless systems, especially for autonomous vehicles and healthcare systems. His work received the best paper award at IEEE GLOBECOM (2012). In 2016 and 2019, he received the UC Hellman Foundation Award and the Dean mid-career research award, respectively. His research is funded by the National Science Foundation, the Department of Defense, Intel and Cisco.

Mohammad Abdullah Al Faruque (Senior Member, IEEE) received his B.Sc. degree in Computer Science and Engineering (CSE) from Bangladesh University of Engineering and Technology (BUET) in 2002, and M.Sc. and Ph.D. degrees in Computer Science from Aachen Technical University and Karlsruhe Institute of Technology, Germany in 2004 and 2009, respectively. Mohammad Al Faruque is currently with the University of California Irvine (UCI), a Full Professor and directing the Embedded and Cyber-Physical Systems Lab. Before he was with Siemens Corporate Research and Technology in Princeton, NJ. Prof. Besides 120+ IEEE/ACM publications in the premier journals and conferences, Prof. Al Faruque holds 11 US patents. Prof. Al Faruque is currently serving as the associate editors of the ACM Transactions on Design Automation on Electronics and Systems and the IEEE Design and Test. He is an IEEE senior member and an ACM senior member. He is also the IEEE CEDA Distinguished Lecturer for 2022-2023.