LOTUS: learning-based online thermal and latency variation management for two-stage detectors on edge devices

Yifan Gong*[1], Yushu Wu*[1], Zheng Zhan*[1], Pu Zhao[1], Liangkai Liu[2], Chao Wu[1], Xulong Tang[3], Yanzhi Wang[1]
*Equal Contribution, [1]Northeastern University, [2]University of Michigan, [3]University of Pittsburgh

ABSTRACT

Two-stage object detectors exhibit high accuracy and precise localization, especially for identifying small objects that are favorable for various edge applications. However, the high computation costs associated with two-stage detection methods cause more severe thermal issues on edge devices, incurring dynamic runtime frequency change and thus large inference latency variations. Furthermore, the dynamic number of proposals in different frames leads to various computations over time, resulting in further latency variations. The significant latency variations of detectors on edge devices can harm user experience and waste hardware resources. To avoid thermal throttling and provide stable inference speed, we propose LOTUS, a novel framework that is tailored for two-stage detectors to dynamically scale CPU and GPU frequencies jointly in an online manner based on deep reinforcement learning (DRL). To demonstrate the effectiveness of LOTUS, we implement it on NVIDIA Jetson Orin Nano and Mi 11 Lite mobile platforms. The results indicate that LOTUS can consistently and significantly reduce latency variation, achieve faster inference, and maintain lower CPU and GPU temperatures under various settings. Our code is available at [link].

1 INTRODUCTION

With the breakthrough of deep neural networks (DNNs), the object detection task has gained rapid development and wide utilization. There is a growing need to run detection models on widespread edge devices for tasks like autonomous vehicles, drone-based environmental monitoring, or mobile inventory management in retail. Among the detection models, two-stage detectors offer improved detection by refining region proposals and enhancing object classification with robustness in detecting objects of different sizes, which is especially helpful in scenarios where high precision is required. Though enjoying these advantages, they are typically more computation intensive, consuming high power during on-device inference, leading to rapidly decreasing battery and increasing temperature. Managing the thermal and latency for two-stage detectors on edge devices is thus a challenging yet urgent problem.

To reduce heat generation and power consumption, Dynamic Voltage and Frequency Scaling (DVFS) is proposed by adjusting CPU or GPU voltage-frequency (VF) levels at runtime. However, conventional DVFS is designed for the operating system kernel and is thus application-agnostic. Furthermore, CPU and GPU have separate VF scaling algorithms, incurring inefficiency of resource utilization within a limited power budget. Besides, although DVFS can reduce power consumption [17], it does not guarantee to solve the overheating problem [15] on edge devices. For edge devices such as mobile phones without active cooling methods such as fan control, if the device temperature goes above a threshold, thermal throttling will be activated to decrease the frequency to a very low level for reducing the temperature. Though the latest work zTT [8] manages to address the thermal throttling problem with a joint VF scaling for CPU and GPU, its direct application for two-stage detectors fails to perform well due to the ineffective design without incorporating the characteristics of two-stage detection models.

To overcome the limitations of prior works, we propose LOTUS, a learning-based online thermal and latency variation management framework tailored for two-stage detectors on edge devices. To improve user experience, our objective is to reduce the latency variation and keep the temperature as low as possible through DVFS. We start by formulating the problem mathematically, then conduct in-depth analysis and profiling of two-stage detection models to obtain their characteristics. It turns out that besides the influence of frequency, the dynamic proposal numbers obtained by the Region Proposal Network (RPN) varying across different images also cause significant latency variations. It makes our problem more challenging as the model latency is affected by both external and internal factors. To tackle this issue, we leverage the DRL approach that can handle dynamic and complex environments with varying images and CPU/GPU temperatures. The DRL agent, powered by a single deep Q-network working at different model widths, provides two chances for frequency scaling during each image frame inference. To train the Q-network effectively, LOTUS keeps two separate experience replay buffers. Furthermore, an additional ϵ_t-greedy cool-down action selection is introduced to avoid thermal issues at early DRL training while allowing the agent to learn how to handle high-temperature cases gradually. To show the effectiveness of LOTUS, we implement it on NVIDIA Jetson Orin Nano and Mi 11 Lite, and the results demonstrate that LOTUS can achieve better performance for both static and dynamic environment settings. We summarize our contributions as follows.

- We analyze and profile the performance and characteristics of two-stage detection models, and find out that the first stage is the main contributor to the latency while the second stage significantly contributes to the large runtime variation due to the dynamic proposal numbers.
- Based on the analysis of the two-stage detection models, we propose LOTUS, a systematic learning-based framework to achieve online thermal and latency variation management tailored for two-stage detectors. LOTUS jointly adjusts the CPU and GPU frequency twice for each image inference with specialized design of the deep Q-network, experience replay buffer, and ϵ_t-greedy cool-down action selection.
- We implement LOTUS on different edge devices, including NVIDIA Jetson Orin Nano and Mi 11 Lite. The results indicate that LOTUS achieves faster inference speed (by up to 30.8%

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06
https://doi.org/10.1145/3649329.3657310

improvement) with reduced variation (by up to 72.8%), better thermal management, and a higher ratio of images meeting the latency constraint (by up to 43.8%) than baselines under both static and dynamic environments.

2 RELATED WORK

Existing DVFS techniques. DVFS is a common technique that dynamically adjusts the VF level of a processor for energy efficiency and thermal management. The VF scaling algorithm in a DVFS implementation, also known as governor, is typically provided by the processor manufacturer and controlled by the operating system. Examples of default governors in Linux are `ondemand` [12] and `interactive` [1], which adjust CPU frequency based on predefined CPU utilization levels. Linux also employs a simplified version of `ondemand`, called `simple_ondemand`, for GPU control. While these governors aim to ensure stable performance and reduce power consumption, they have two limitations. First, they do not consider application performance. Second, having separate governors for CPU and GPU hampers efficient resource utilization within a limited power budget. Recently, a series of methods [2, 3] are proposed to design governors for various applications to tackle these limitations, yet the tasks are significantly different from deep learning (DL) tasks. Furthermore, most DVFS implementations still suffer from overheating problems. For devices like mobile phones, if the device temperatures reach a certain threshold, thermal throttling occurs, which significantly degrades application performance. zTT [8] takes both thermal throttling issues and DL-based applications into consideration. However, it suffers from a significant performance drop if directly applied to two-stage detectors. Latest work [9] designs the governor from another perspective with multiple concurrent tasks, which is orthogonal to our design.

Object Detection. There are two mainstreams of DNN-based object detection models, i.e., one-stage detectors and two-stage detectors. One-stage detectors perform object detection in a single pass without explicitly proposing regions. YOLO [13] is one of the representative model that predicts bounding boxes and class probabilities directly from the full image. Though one-stage detectors are computationally efficient, they usually achieve lower accuracy compared to two-stage detectors. Two-stage detectors propose region proposals before performing object classification and refinement. This approach has shown high accuracy in object detection and are more suitable to various application on edge devices that requires high precision such as perception in autonomous vehicles, suspicious activities detection in restricted areas, and environmental monitoring by drones. Faster R-CNN [14] introduces a RPN to generate region proposals and a Fast R-CNN network for object classification and bounding box regression. Mask R-CNN [7] extends Faster R-CNN to include an additional branch for instance segmentation, achieving state-of-the-art results in object detection and segmentation tasks. R-FCN [4] proposes a region-based fully convolutional network, which enables efficient region-wise computation for object detection.

3 MOTIVATIONS

Thermal Issues. With the advancement of edge devices, the capabilities of their processors have significantly increased, packing more power into compact spaces. However, this progress comes with the challenge of managing the heat generated during intensive computations. With the slim design of mobile phones and edge devices, thermal management becomes crucial to prevent severe

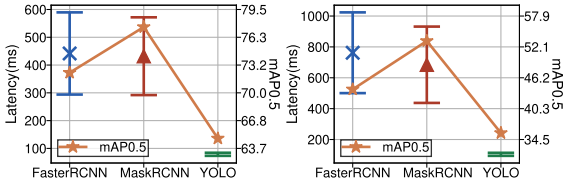


Figure 1: The mean and variation of inference latency and precision for two-stage detectors (FasterRCNN, MaskRCNN) and one-stage detector (YOLOv5) on different datasets.

performance degradation in such demanding scenarios. Multiple factors influence the temperature characteristics of these devices such as environmental temperature, on-device applications, thermal coupling among processors and battery. The mutual effects of these factors are too complicated to be characterized precisely, making the thermal management problem challenging.

Challenges of Two-Stage Detection Models. Compared to one-stage detectors, two-stage detectors allow for more refined region proposals and improved object classification, resulting in enhanced detection performance [4, 7, 10, 13, 14], as shown in Fig. 1 with a higher mAP. Specifically, they exhibit robustness in detecting objects of varying sizes with the region proposal stage to identify potential object regions regardless of their size. This is especially beneficial in cases with objects of significant size variations in the same image, such as small objects in high resolution images obtained from aerial surveillance.

Though enjoying these benefits, the practical deployment of two-stage detection models on edge devices faces the challenges of unstable inference. To demonstrate the issue, we show the mean and variation of the inference latency for representative one-stage detector YOLOv5 [16], and two-stage detectors including FasterRCNN [14] and MaskRCNN [7] in Fig. 1. The mean and variation are calculated by the inference time on each dataset. On the KITTI dataset, FasterRCNN, MaskRCNN, and YOLOv5 exhibit inference variation of 134.10ms, 194.64ms, and 2.75ms, respectively. Similarly, on the VisDrone2019 dataset, the inference variation for these detectors are 198.85ms, 200.18ms, and 10.38ms. The latency variation for two-stage detectors is significantly greater than one-stage detectors. The unstable inference speed not only harms the user experience, but also wastes a lot of hardware resources [5, 6]. In applications involving object tracking, navigation, or control systems, large latency variations can negatively impact the accuracy of tracking and control algorithms. Systems that integrate multiple sensors or modalities face challenges in maintaining synchronization when the latency of different sources varies significantly.

The unstable latency problems for two-stage detectors are raised from two aspects. First, two-stage detection models are more **computationally intensive**, thus causing a higher risk of overheating. To mitigate the thermal issue, certain frequency adjustment is leveraged at the cost of varying inference time. Second, two-stage methods generate a **dynamic** number of **object proposals** for different image frames, thus incurring more uncertainty for the computation counts and thus inference speed. On the contrary, the feature maps in one-stage models are used to generate a static number of anchors or default boxes.

4 LOTUS FRAMEWORK DESIGN

4.1 Problem Formulation

4.1.1 Problem Formulation. It is necessary yet challenging to manage the heat, especially for the intensive DNN computations on edge

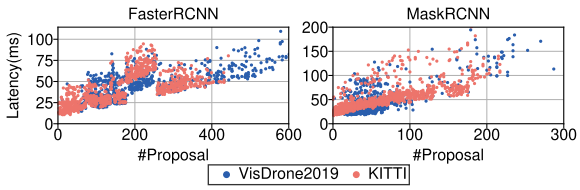


Figure 2: Inference latency of the second stage for different numbers of proposals on FasterRCNN and MaskRCNN.

devices. At runtime, DVFS that adjust the CPU and GPU frequency (f^{cpu} & f^{gpu}) is an efficient method to decrease heat production from circuits and manage the inference speed dynamically. To provide better user experience with more stable inference, there are three requirements for the inference latency l_i of the i^{th} image: (i) l_i should be as small as possible for fast responsiveness; (ii) l_i is better to meet the latency constraint L posed by the applications, i.e., $l_i < L$; (iii) l_i should maintain a small latency variation under various environments. Besides, to prevent the device from overheating that causes thermal throttling, the CPU temperature T_i^{cpu} and GPU temperature T_i^{gpu} should not exceed the pre-defined threshold value T^{thres}. Thus, l_i, T_i^{cpu}, and T_i^{gpu} are all highly influenced by f^{cpu} and f^{gpu}. Scaling the frequency appropriately is a crucial problem to guarantee better user experience. More specifically, the optimization problem can be formulated mathematically as:

$$\begin{aligned} \min \sum_{i=1}^{I} & l_i + \alpha \cdot (l_i - \bar{l})^2 + \beta \cdot U(l_i - L) \\ s.t. \quad & T_i^{cpu} \leq T^{thres}, \forall i \\ & T_i^{gpu} \leq T^{thres}, \forall i \end{aligned} \tag{1}$$

where $\bar{l}$ represents the mean of the latency, U is the step function with a value of 1 for inputs greater than 0, and 0 otherwise, α and β are two hyperparameters adjusting their relative importance.

4.1.2 Two-Stage Detection Models. Understanding the object detection algorithms is the base to solve the optimization problem in Eq. (1). Besides pre- and post-processing, the inference of a two-stage detector mainly has four parts. As shown in Figure 3, it begins with a backbone such as ResNet-50 to extract the features. Following the backbone is the RPN that draws candidates of objects based on features to creates proposals. The second stage begins with Region of Interest (RoI) pooling to create proposal feature maps with feature maps and region proposals. The results are then fed into the classifier, which utilizes fully connected layers to determine the object class of each proposal and establish a fixed bounding box.

The dynamic number of proposals generated in the first stage leads to dynamic computation counts in following steps, and thus more significant inference time variation compared to one-stage models with a static number of anchors/default boxes for detection. To verify this, we demonstrate the correlation between the latency of the second stage and the number of proposals in Fig. 2 by setting the CPU and GPU frequency at a fixed level. As observed from the figure, on both datasets, a large number of proposals obtained by the first stage corresponds to a longer latency for the second stage.

4.2 Lotus Framework with Frequency Scaling

We make the following observations. (i) As the backbone and RPN have a fixed amount of computations, the latency for these parts is influenced by the hardware execution environment (frequency).

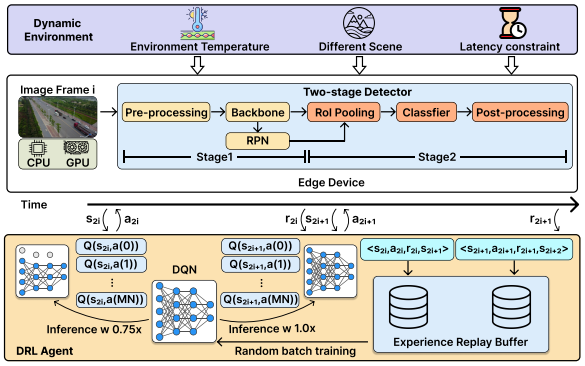


Figure 3: Overview of Lotus.

(ii) The latency of the second stage is determined not only by the hardware environment, but also the number of proposals from the first stage that varies across images. To provide a stable inference speed for two-stage detectors, the frequency adjustment problem can be decomposed to *when* and *how* to adjust the frequency.

For the *when* problem, we first perform a detailed profiling of the inference latency with fixed frequency. We observe that the latency of the first stage (including the preprocessing, backbone and RPN) takes about 80% of the entire model latency, while the latency of the ROI, classifier, and post-processing takes about 20%. Meanwhile, the runtime variation is mainly introduced by the second stage. E.g., for the FasterRCNN on KITTI, the runtime variation of its second stage can be up to 159.5ms, where its entire inference is merely 441.9ms on average. We can conclude that with fixed frequency, the first stage is the main contribution for latency while the second stage significantly contributes to the large runtime variation.

Thus, for the *when* problem, though scaling the frequency at the beginning of an image inference [8] is the most standard method, it is not suitable for two-stage detectors, since it is not able to effectively reduce the latency variation from the second stage. Besides, scaling the frequency at the start of the second stage solely is not desirable either, as it can hardly reduce the latency with the first stage as the main contributor. Thus, based on the above discussion, the first *when* question can be addressed by allowing two frequency decisions for each image inference, i.e., at the start of an image inference and after the creation of the proposals. Furthermore, the overhead of frequency scaling is negligible with dozens of microseconds (compared to the DNN latency) on edge devices.

For the next *how* problem, directly solving the optimization problem as in Eq. (1) is infeasible. The frequency scaling problem is conducted in a dynamic and complex environment with varying images and CPU/GPU temperature. The frequency to be taken not only depends on the current image and system information, but also the frequency decisions made in the past. Current frequency decision also affects the thermal situation in the near future. These features make our problem suitable to leverage DRL, where an agent learns how to make appropriate decisions through interaction with a dynamic environment.

Based on the above discussions, we illustrate the framework overview of Lotus in Fig. 3. Lotus is mainly composed of a DRL-based agent interacting with the environment and providing two actions for each image frame. At the start of an image inference, it performs frequency scaling based on the current status. Next at the end of the first stage with number of proposals available, it can further adjust the frequency for the optimization objectives.

4.3 DRL Design

We address the optimization of problem in Eq. (1) with a DRL approach, and carefully design the state, action, and reward of the DRL agent in Lotus, as specified below.

4.3.1 Action. The DRL agent provides two separate actions a_{2i} and a_{2i+1} for the inference of the i^{th} image frame. The first action a_{2i} is given at the beginning of the i^{th} image frame while the second action a_{2i+1} is taken after the inference of the RPN. The two-action design is proposed to tackle the characteristics of the two-stage detection models as mentioned in Sec. 4.2. The action set for both actions is the same. For a device equipped with M CPU frequency levels and N GPU frequency levels, the entire action space contains $M \times N$ different action choices. It is specifically defined as $\mathbf{a} = \{< f^{cpu,1}, f^{gpu,1} >, \cdots, < f^{cpu,m}, f^{gpu,n} >, \cdots, < f^{cpu,M}, f^{gpu,N} >\}$, where $< f^{cpu,m}, f^{gpu,n} >$ corresponds to scaling the CPU frequency to the m^{th} level and GPU frequency to the n^{th} level, respectively.

4.3.2 State. Different from the action design that is the same for a_{2i} and a_{2i+1}, the state differs for two consecutive time steps. The state s_{2i} observed at the beginning of the inference of the i^{th} image is formulated as a tuple with six elements represented as $\{S_{2i}, T_{2i}^{cpu}, T_{2i}^{gpu}, f_{2i}^{cpu}, f_{2i}^{gpu}, \Delta L_{2i}\}$. S_{2i} indicates the current stage, T_{2i}^{cpu} and T_{2i}^{gpu} represent the current CPU and GPU temperature, and f_{2i}^{cpu} and f_{2i}^{gpu} correspond to the current CPU and GPU clock frequency level. ΔL_{2i} indicates the remaining time to meet the latency constraint. As for the state s_{2i+1}, it is formulated with a seven element tuple $\{S_{2i+1}, T_{2i+1}^{cpu}, T_{2i+1}^{gpu}, f_{2i+1}^{cpu}, f_{2i+1}^{gpu}, \Delta L_{2i+1}, P_{2i+1}\}$. The additional observation dimension P_{2i+1} is the number of proposals for the i^{th} image. P_{2i+1} is obtained based on the feature map from the backbone and the region proposals from the RPN, thus is only available at time step $2i+1$ for the i^{th} image frame.

4.3.3 Reward. We propose a specialized reward design for Lotus. To achieve the goal of stable inference for two-stage detectors, Lotus tries to optimize the latency while avoiding thermal throttling. Specifically, the reward is defined as $r = r^{time} + \lambda r^{temp}$, where r^{time} stands for the reward for latency while r^{temp} is the reward for temperature. Specifically, r^{time} is formulated as

$$r_i^{time} = \begin{cases} \tanh(\Delta L_i) + \frac{1}{1+\sigma_n(\Delta L_i)}, & if\ \Delta L_i > 0; \\ p\Delta L_i, & otherwise, \end{cases} \tag{2}$$

where $\tanh(\Delta L_i)$ ensures fast inference and $\frac{1}{1+\sigma_n(\Delta L_i)}$ constrains the latency variation, with $\sigma_n(\Delta L_i)$ representing the standard deviation calculated from the n most recent images. If $\Delta L_i < 0$, it means a violation of the time constraint, thus a penalty multiplier p is applied. As for the temperature reward, it is defined as

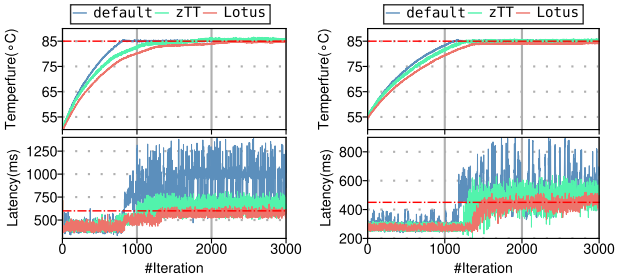
$$r_i^{temp} = \begin{cases} 1, & if\ T^{cpu} \le T^{thres}\ and\ T^{gpu} \le T^{thres}; \\ -p, & otherwise, \end{cases} \tag{3}$$

where a positive reward is received as long as both T^{cpu} and T^{gpu} are below the device throttling temperature T^{thres}. However, when either temperature exceeds the throttling temperature T^{thres}, a penalty multiplier p is applied.

4.3.4 Specialized DRL Agent Design for Two-Stage Detection Models. Lotus resorts to a model-free DRL approach and leverages the popular Deep Q-Networks (DQN) algorithm [11]. The fundamental concept of Q-learning is based on the notion that if we had a function Q^* capable of predicting the expected return when taking a specific action in a given state, it is easy to derive a policy that maximizes the rewards by simply following $\pi^*(s) = \arg\max_a Q^*(s,a)$. As DNNs can work as universal function approximators, the agent learns to approximate the optimal Q-value function $Q^*(s,a)$ with DNNs in the DQN algorithm. Different from other settings, the observations of Lotus for state s_{2i} and s_{2i+1} are different for the inference of the i^{th} image. This indicates that there should be two sets of Q-value functions corresponding to the two different state-action pairs for each image. The most straightforward approach to tackle this is to leverage two separate DNNs (i.e., two Q-networks) for the approximation. But this **separation cuts off the correlations** between the two actions for the inference of the same image frame.

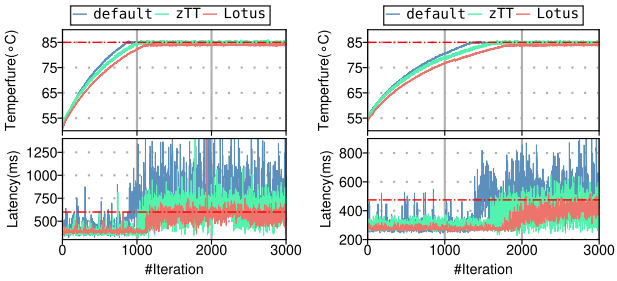
To deal with this problem, we investigate whether we can keep only one DNN for the Q-value approximation for these two sets of state-action pairs. We observe that the major difference between the first state-action pair $(s_{2i}.a_{2i})$ and second state-action pair $(s_{2i+1}.a_{2i+1})$ for the i^{th} image frame is the presence of the proposal number P_{2i+1}, which can only be obtained after the inference of the first stage. Thus, we design the Q-network to be executed with two different widths (number of active channels)) $[\alpha\times, 1.0\times]$ configurations. The Q-value for the first state-action pair $(s_{2i}.a_{2i})$ is computed by only the first $\alpha\times$ (e.g., $0.75\times$ or 75%) channels in each layer, while the Q-value of the second state-action pair $(s_{2i+1}.a_{2i+1})$ is obtained by executing the Q-network with full network width for each layer (i.e., $1\times$), as shown in Fig. 3. With this design, the Q-value computations **share major parameters and computations**. Two separate experience replay buffers are leveraged to store the observed transitions, i.e., one is to store transitions $< s_{2i}, a_{2i}, r_{2i}, s_{2i+1} >$ at $2i$ time step while the other is to store $< s_{2i+1}, a_{2i+1}, r_{2i+1}, s_{2i+2} >$ for $2i+1$. During training, a batch of random samples is selected in the corresponding experience replay buffer to update the Q-network. In particular, at time step $2i$, the sampled transitions are used to update the Q-network with $\alpha\times$ width, while the remaining weights are not updated.

4.3.5 ϵ_t-greedy cool-down action selection. Thermal issues are inevitable during the DRL training and inference phases. Unlike larger devices (such as servers and desktops) with fans or powerful cooling systems, heat can not be effectively dissipated on edge devices. The ineffective heat dissipation leads to delayed responsiveness for DRL actions, yielding difficulty in the DQN algorithm with ϵ-greedy that explores a random action with probability ϵ. To avoid choosing higher frequency when the device is already overheated (T^{cpu} or T^{gpu} is higher than T^{thres}), zTT [8] introduces a *cool-down action* that specifically selects a random CPU and GPU frequency which is lower than the current status in such circumstances. However, this random action prevents the agent from learning reasonable action selections when the temperature is high, thus the agent is not able to provide good action selection in such cases. To avoid this issue, Lotus introduces ϵ_t-greedy cool-down action selection, where ϵ_t is initialized between $[0, 1]$. When the device is overheated, Lotus selects a random frequency lower than the current setting by the probability ϵ_t. Otherwise, the action is selected according to the output of the Q-network. Each time the cool-down action is triggered, the probability ϵ_t decays sinusoidally as the agent accumulates more experience in handling the overheating case. Implementing ϵ_t-greedy cool-down action selection prevents severe performance drops due to thermal issues in the early training phase, promoting



(a) VisDrone2019 dataset (b) KITTI dataset

Figure 4: Comparison on Jetson Orin Nano with FasterRCNN. Red dashed lines indicate the throttling bound and latency constraint.



(a) VisDrone2019 dataset (b) KITTI dataset

Figure 5: Evaluation on Jetson Orin Nano with MaskRCNN. Red dashed lines indicate the throttling bound and latency constraint.

smoother convergence of the Q-network. Simultaneously, it enables the agent to gradually improve its ability to manage overheating scenarios throughout the training process.

4.4 Lotus Implementation

Lotus is implemented in Python for two edge devices. (i) NVIDIA Jetson Orin Nano with a 6-core Arm Cortex-A78AE v8.2 64-bit CPU (1.5GHz), 1024-core NVIDIA Ampere architecture GPU with 32 Tensor Cores (625MHz), and an 8GB 128-bit LPDDR5 memory. (ii) Mi 11 Lite equipped Qualcomm SnapDragon 780G chipset with a Qualcomm Kryo 670 Octa-core CPU (1×2.40 GHz Cortex-A78 & 3×2.22 GHz Cortex-A78 & 4×1.90 GHz Cortex-A55) and a Qualcomm Adreno 642 GPU. Fig. 3 illustrates an overview of Lotus implementation. The Lotus agent is executed with an NVIDIA 2080Ti GPU and controls the CPU and GPU frequency of client devices. Data between the client (edge device) and agent is transmitted through the `socket`. Note that data can also be transmitted via wired or wireless networks to adapt to various scenarios. The observation of each state is directly collected through the `sysfs` in the Linux kernel and Android kernel for (i) and (ii), respectively.

4.4.1 Lotus Training. The Q-network architecture is designed as a 4-layer MLP that works at two widths [0.75×, 1×]. The Q-network in Lotus agent selects actions based on typical ϵ-greedy and ϵ_t-greedy cool-down action selection proposed in Sec. 4.3.5. The agent collects observation samples, calculates rewards, and stores samples in the two sets of experience replay buffers. Then, the Q-network is trained for 10,000 iterations using Adam optimizer with $\beta_1 = 0.9, \beta_2 = 0.99$, and the learning rate is set to 0.01 with cosine decay.

4.4.2 Overhead analysis. The overhead incurred by the Lotus is trivial. It comes from executing the Q-network of the agent and data transmission between the agent and the client device. The Q-network latency is benchmarked on a desktop with an NVIDIA GeForce 2080Ti GPU, which only consumes 0.42ms on average. The data transmission via `socket` takes 1.92ms/message. Thus, the overall overhead of Lotus is 8.52ms/inference, which is marginal compared to the inference latency of the two-stage detection model.

Table 1: Quantitative results on Jetson Orin Nano. The $\bar{l}$ and σ_l indicate the mean and standard deviation of latency. R_L indicates the ratio of meeting the preset latency constraint (satisfaction rate).

Detector	Method	KITTI			VisDrone2019		
		$\bar{l}$(ms)↓	σ_l(ms)↓	R_L ↑	$\bar{l}$(ms)↓	σ_l(ms)↓	R_L ↑
FasterRCNN	default	434.6	139.8	51.4%	686.0	241.1	29.4%
	zTT	363.7	85.6	55.5%	577.6	167.5	46.3%
	Lotus	**343.2**	**68.6**	**66.5%**	**523.5**	**102.9**	**71.1%**
MaskRCNN	default	443.9	148.0	59.8%	768.4	260.4	39.0%
	zTT	408.3	111.7	87.1%	584.3	114.2	50.1%
	Lotus	**388.5**	**88.9**	**95.2%**	**531.4**	**70.7**	**74.9%**

5 EXPERIMENTS

We implement Lotus on Jetson Orin Nano and Mi 11 Lite to demonstrate its advantages of maintaining stable DNN inference while preventing overheating.

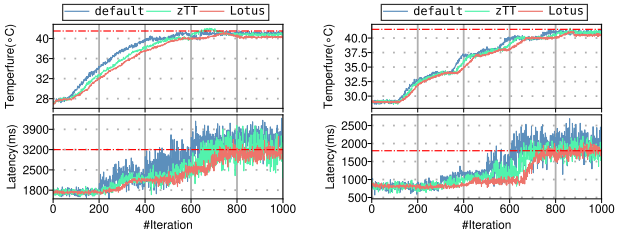
5.1 Experiment Setup

5.1.1 Baselines. We compare with the default governors and the latest learning-based thermal management governor zTT [8]. The default governors for the CPU and GPU on NVIDIA Jetson Orin Nano are `schedutil` and `nvhost_podgov`, respectively. Mi 11 Lite adopts `schedutil` and `msm-adreno-tz` as its governor to control its CPU and GPU frequency. The state-of-the-art zTT [8] scales CPU and GPU frequency jointly with a DRL approach.

5.1.2 Evaluation Environments. We evaluate Lotus in comparison with baselines across diverse settings that simulate real-world usage environments. We test on the KITTI and VisDrone2019 datasets, which are commonly used in autonomous vehicles and drones. The FasterRCNN and MaskRCNN are selected as the on-device two-stage detectors. These two well-known models are the basis of succeeding two-stage detectors. Note that different latency constraints in Lotus are applied for different datasets and models due to their varied computation demands. We also test in both static and dynamic external environments for practical considerations.

5.2 Experiment Results

5.2.1 Lotus on Static External Environments. The static external environment is defined by the normal 25°C indoor environment without external cooling systems. We compare the device temperature (averaged between the CPU temperature and GPU temperature) and detector latency by executing the detector 3,000 iterations on the device (each iteration corresponds to processing one image). For NVIDIA Jetson Orin Nano, the results are shown in Fig. 4 for FasterRCNN, Fig. 5 for MaskRCNN, and Tab. 1. We can observe that Lotus consistently achieves better performance by maintaining a lower device temperature, faster average inference speed, smaller latency variation, and higher ratio of meeting preset time constraints (i.e., higher satisfaction rate). Specifically, for MaskRCNN on VisDrone2019, Lotus reduces the latency by 30.8% compared to the default and 9.1% compared to zTT, with a lower device temperature consistently. Meanwhile, the Lotus latency variation measured by standard deviation is reduced by 72.8% and 38.1% compared to the



(a) VisDrone2019 dataset (b) KITTI dataset

Figure 6: Evaluations on Mi 11 Lite with FasterRCNN. Red dashed lines indicate the throttling bound and latency constraint.

Table 2: Quantitative results on Mi 11 Lite 5G. The $\bar{l}$ and σ_l indicate the mean and standard deviation of latency. R_L indicates the ratio of meeting the preset latency constraint (satisfaction rate).

Detector	Method	KITTI			VisDrone2019		
		$\bar{l}$(ms)↓	σ_l(ms)↓	R_L ↑	$\bar{l}$(ms)↓	σ_l(ms)↓	R_L ↑
FasterRCNN	default	1377.5	525.1	70.9%	2728.0	761.5	63.3%
	zTT	1260.9	448.2	83.3%	2509.7	649.3	79.7%
	Lotus	**1185.8**	**429.9**	**89.7%**	**2421.0**	**558.7**	**92.5%**
MaskRCNN	default	1652.1	781.8	61.3%	3241.9	725.5	40.1%
	zTT	1582.7	610.5	79.8%	2972.5	621.7	59.4%
	Lotus	**1429.5**	**552.3**	**91.5%**	**2649.5**	**591.2**	**83.8%**

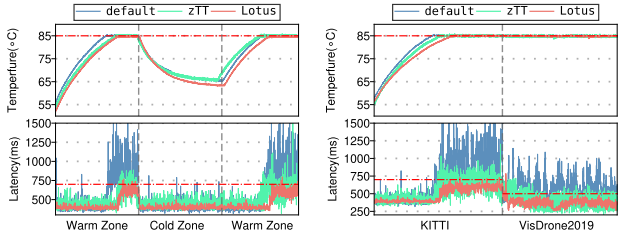
two baselines. Furthermore, Lotus improves the satisfaction rate by 35.9% and 24.8% compared to the two baselines.

The results on Mi 11 Lite are shown in Fig. 6 and Tab. 2. Similar to Jetson Orin Nano, Lotus achieves better performance than all baselines across all measures. For instance, Lotus decreases the latency variation by 29.4% and 9.5% for MaskRCNN on KITTI. The ratio of images meeting time constraints are improved to 92.5% by Lotus for FasterRCNN on VisDrone2019. Based on the above results, Lotus is effective in thermal and latency variation management.

5.2.2 Lotus on Dynamic External Environments. The external environment of the devices may change dynamically. Thus, to exhibit the robustness of Lotus across various environments, we test the edge device with diverse environmental changes.

Temperature changes. An edge device typically works under varying temperatures. For instance, a mobile device may switch between the warm indoor and the cold outdoor frequently in the winter. A drone operating in open airspace can experience very different outside temperatures due to the altitude and weather conditions. The change of environmental temperature challenges the robustness of Lotus. To verify its robustness against temperature fluctuating, we define two temperature zones: (i) warm zone (25°C) (ii) cold zone (0°C). The device is placed at different temperature zones during the inference. Fig. 7a shows the results of changing temperature zones when using the MaskRCNN detector on VisDrone2019. We can see that Lotus can smoothly and quickly adapt to the temperature changes with consistently lower temperature, faster inference, smaller latency variation, and higher satisfaction rate. With lower outside temperature and better cooling conditions, Lotus seeks to keep low latency and variations. With higher temperature, Lotus tends to prevent overheating while meeting the latency constraint.

Domain changes. It is common for a device to carry multiple tasks on different domains. For example, the search and rescue drone is expected to detect vehicles on the street and identify missing persons in the wild. Furthermore, the domain change is usually



(a) Temperature changes (b) Domain changes

Figure 7: Performance when the environment changes. Red dashed lines indicate the throttling bound and latency constraint.

accompanied with different time constraint settings as different tasks usually have different requirements. Here, we switch the dataset from KITTI to VisDrone2019 during inference, as shown in Fig. 7b. The results indicate that Lotus performs better than baselines when the task domain switches with a more stable inference and thermal management. We can conclude that Lotus can better adapt to environmental changes and utilize hardware resources.

6 CONCLUSION

This paper proposes Lotus, a framework that tackles the challenges of unstable inference and thermal issues on edge devices for two-stage detectors by dynamically scaling CPU and GPU frequencies via DRL. Experiments demonstrate consistent improvements in latency and temperature control, making Lotus a promising solution for stable and efficient edge-based applications.

ACKNOWLEDGEMENT

This work is partly supported by the Army Research Office/Laboratory via grant W911-NF-20-1-0167 to Northeastern University, National Science Foundation CCF-1937500, and CNS-1909172.

REFERENCES

[1] Dominik Brodowski, Nico Golde, Rafael J Wysocki, *et al.* 2013. Cpu frequency and voltage scaling code in the linux (tm) kernel. *Linux kernel documentation* (2013), 66.
[2] Yonghun Choi, Seonghoon Park, Hojung Cha. 2019. Graphics-aware power governing for mobile devices. In *MobiSys*.
[3] Po-Kai Chuang, Ya-Shu Chen, Po-Hao Huang. 2017. An adaptive on-line CPU-GPU governor for games on mobile devices. In *ASP-DAC*.
[4] Jifeng Dai, Yi Li, Kaiming He, *et al.* 2016. R-FCN: Object Detection via Region-based Fully Convolutional Networks. In *NeurIPS*.
[5] Yifan Gong, Zheng Zhan, Pu Zhao, *et al.* 2022. All-in-one: A highly representative dnn pruning framework for edge devices with dynamic power management. In *ICCAD*.
[6] Yifan Gong, Pu Zhao, Zheng Zhan, *et al.* 2023. Condense: A Framework for Device and Frequency Adaptive Neural Network Models on the Edge. In *DAC*.
[7] Kaiming He, Georgia Gkioxari, Piotr Dollár, *et al.* 2017. Mask R-CNN. In *ICCV*.
[8] Seyeon Kim, Kyungmin Bin, Sangtae Ha, *et al.* 2021. zTT: Learning-based DVFs with zero thermal throttling for mobile devices. In *MobiSys*.
[9] Chengdong Lin, Kun Wang, Zhenjiang Li, *et al.* 2023. A Workload-Aware DVFS Robust to Concurrent Tasks for Mobile Devices. In *MobiCom*.
[10] Tsung-Yi Lin, Piotr Dollár, Ross B. Girshick, *et al.* 2017. Feature Pyramid Networks for Object Detection. In *CVPR*.
[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, *et al.* 2015. Human-level control through deep reinforcement learning. *nature* 518, 7540 (2015), 529–533.
[12] Venkatesh Pallipadi Alexey Starikovskiy. 2006. The ondemand governor. In *Proceedings of the linux symposium*, Vol. 2. 215–230.
[13] Joseph Redmon, Santosh Divvala, Ross B. Girshick, *et al.* 2016. You Only Look Once: Unified, Real-Time Object Detection. In *CVPR*.
[14] Shaoqing Ren, Kaiming He, Ross B. Girshick, *et al.* 2015. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *TPAMI* (2015).
[15] Krishna Sekar. 2013. Power and thermal challenges in mobile devices. In *MobiCom*.
[16] Ultralytics. 2021. YOLOv5: A state-of-the-art real-time object detection system. https://docs.ultralytics.com.
[17] Yushu Wu, Yifan Gong, Zheng Zhan, *et al.* 2023. MOC: Multi-Objective Mobile CPU-GPU Co-Optimization for Power-Efficient DNN Inference. In *ICCAD*.