Context-aware Image Compression Optimization for Visual Analytics Offloading

ABSTRACT

Convolutional Neural Networks (CNN) have given rise to numerous visual analytics applications in Internet of Things (IoT) environments. Visual data is typically captured by IoT cameras and then live streamed to edge servers for analytics due to the prohibitive cost of running CNN on computation-constrained IoT end devices. To guarantee low-bandwidth and low-latency visual analytics offloading and accurate visual analytics, the key lies in image compression that minimizes the amount of visual data to offload. Despite the wide adoption, JPEG standard and traditional image compression do not address the accuracy of analytics tasks, leading to ineffective compression for visual analytics offloading. Although recent machine-centric image compression techniques leverage sophisticated neural network models or hardware architecture to support the accuracy-bandwidth trade-off, they introduce excessive latency in the visual analytics offloading pipeline. This paper presents CICO, a Context-aware Image Compression Optimization framework to achieve low-bandwidth and low-latency visual analytics offloading. CICO contextualizes image compression for offloading by employing easily-computable low-level image features to understand the importance of different image regions for a visual analytics task. Accordingly, CICO is able to optimize the trade-off between compression size and analytics accuracy. Extensive results from real-world experiments demonstrate that CICO reduces the bandwidth consumption of existing compression methods by up to 40% under a comparable analytics accuracy. In terms of the low-latency support, CICO achieves up to a 2x speedup over state-of-the-art compression techniques.

1 INTRODUCTION

With the advancement in Convolutional Neural Networks (CNN) [13, 17, 34], visual analytics tasks (herein referred to as vision apps) such as human face recognition [31], pedestrian detection [6], or traffic monitoring [23] have been deployed in Internet of Things (IoT) environments. Typically, visual data is captured by the cameras of IoT end devices, e.g., underwater sensor nodes [21], and then live streamed to edge servers for analysis due to the computation constraints of the IoT end devices and the prohibitive cost of running CNN models on these end devices.

To guarantee the performance of vision apps in IoT systems, the network bandwidth required for visual analytics offloading must be minimized because of the challenging network conditions in IoT environments. For example, capturing and offloading images in drone object detection requires us to minimize the offloading bandwidth since the network connection between the drone and edge server can be highly dynamic or even intermittent. Moreover, the latency of the whole visual analytics offloading pipeline, from encoding to decoding, must be minimal to support time-sensitive vision apps. For example, during victim search in a fire incident, images of the firefighting site should be sent to the command center for analysis as soon as possible so that commanders can guide the rescue operation effectively.

The key to achieving the *low-bandwidth* and *low-latency* visual analytics offloading is to minimize the amount of visual data to offload through image compression. Well-known image compression standards such as JPEG [41] and JPEG2000 [38] focus on improving the visual quality of the reconstructed images under limited network bandwidth. However, they are not able to consider the analytics accuracy when they are applied to image offloading in vision apps.

Machine-centric image compression [9] has been proposed to address this limitation by both enhancing the accuracy of vision apps and minimizing the size of the data to be offloaded. CNN-driven compression [2, 3, 33, 43] is one category of such techniques. These methods employ CNN models to encode an image into a vector at the IoT end device for offloading and use generative models at the server to reconstruct the image. They can compress images into smaller sizes than traditional image compression standards while preserving the quality of reconstructed images. However, these approaches usually require heavy computation power (e.g., GPU) to perform encoding (on the IoT end device) and/or decoding (on the edge server) through sophisticated CNN models [2, 3, 33], which could incur excessive end-to-end latency in the offloading pipeline for vision apps. The other category of machine-centric compression – server-driven compression [9, 25] compresses images for offloading adaptively based on the information sent from the edge server that indicates the importance of image regions. Nevertheless, the server feedback introduces an additional delay before the data can be compressed for offloading. If the delay is significant, the regions of interest (ROI) sent by the edge server can deviate from the ROI currently captured and the compression performance will degrade.

In this paper, we remedy the aforementioned issues of existing image compression techniques by proposing *CICO*, Context-aware Image Compression Optimization. CICO is a lightweight framework that contextualizes and optimizes image compression for low-bandwidth and low-latency visual analytics offloading in vision apps. As low-level image features such as STAR [1] and FAST [35] reflect high-level image semantics that is of interest to the vision apps, CICO learns such a relationship and utilizes it to identify the importance of different image regions for a vision app. Accordingly, CICO optimizes the trade-off between compression size and analytics accuracy. By putting the compression of each image region under a vision app into a *context*, CICO is able to minimize the required network bandwidth for visual analytics offloading while preserving the analytics accuracy. By employing image features that can be computed efficiently in the runtime, CICO allows images to be compressed, offloaded, and reconstructed in a minimal end-to-end latency. To the best of our knowledge, CICO is the first compression framework that achieves low-bandwidth and low-latency visual analytics offloading while ensuring analytics accuracy.

Realizing CICO requires us to overcome two challenges.

1. How to make the relationship between image features and image compression learnable? The basic principle of CICO is that the image region with a higher density of important image features should have a higher compression quality, i.e., less information loss. To achieve this goal, design choices like 1) the significance of different features in a particular vision app and 2) the mapping from the feature density to the compression quality have to be made. We innovatively propose the *context-aware compression module* (CCM) within the CICO framework that models the above design choices into learnable parameters (referred to as the *configuration*). The CCM is a generic module that can be built on top of any other compression methods such that the compression methods will fit a vision app in a better way.

2. How to conduct the learning in order to compress images? An essential step in CICO is to make the CCM aware of and optimized for the target vision app. To this end, we model the selection of the configuration of the CCM into a multi-objective optimization (MOO) problem. The variable is the configuration and the objectives are 1) maximizing the analytics accuracy regarding the vision app, e.g., the top-1 accuracy for image classification and the mean average precision (mAP) for object detection [**?**], and 2) minimizing the size of data to be offloaded. Solving the MOO problem means deriving its *Pareto front*, which is non-trivial because of the infinite design space of the configuration and the costly evaluation of a configuration. We address these issues with the *compression optimizer* (CO) within the CICO framework that optimizes the choice of configurations and efficiently evaluates each configuration. The CO finds offline the optimal set of configurations for the CCM in a reasonable amount of time.

We evaluate CICO by focusing on two vision apps (image classification and object detection) and two IoT end devices (Raspberry Pi 4 Model B and Nvidia Jetson Nano) in two network environments (WiFi and LTE networks). By comparing CICO with traditional JPEG standard and a CNN-based compression method [43], our extensive results demonstrate that CICO improves the accuracy-bandwidth trade-offs of JPEG and CNN-based encoders and achieves a lower end-to-end latency and higher processing speed for visual analytics offloading. Specifically, CICO reduces the size of offloaded images compressed by existing compression techniques by up to 40% while reaching a comparable analytics accuracy. In terms of the support for low-latency vision apps, CICO achieves up to a 2× speedup over state-of-the-art compression techniques.

The contribution of this paper is summarized as follows.

- We propose CICO, a novel and lightweight framework that contextualizes and optimizes the image compression for low-bandwidth and low-latency offloading in vision apps.
- We model and solve the image compression as an MOO problem offline, which allows online compression to be context-aware with minimal impact on the latency.
- We optimize JPEG and a CNN-based encoder with CICO and conduct extensive evaluations to validate the low-bandwidth and low-latency benefits of CICO.

For the remainder of this paper, we first discuss the motivation and the related work in Section 2. Then, we present an overview of the system architecture in Section 3. Two key components in CICO, the context-aware compression module, and the compression optimizer are detailed in Section 4 and Section 5, respectively. CICO is evaluated in Section 6, which is followed by the discussion in Section 7 and the conclusion in Section 8.

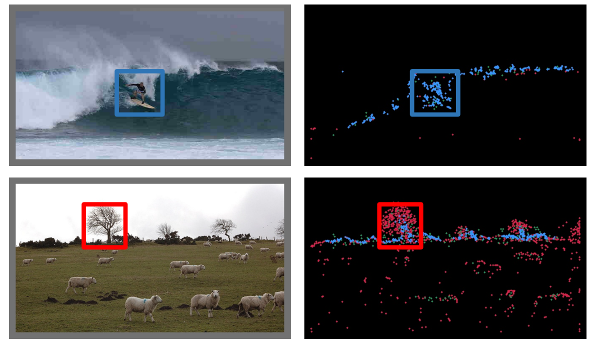


Figure 1: Low-level image features indicate different ROI.

2 MOTIVATION AND RELATED WORK

2.1 Motivation

Low-level image features (referred to as features) abstract image information and are highly related to the vision app. They could provide the context to enhance image compression in a lightweight manner if used appropriately. In essence, features are calculated by making a binary decision at every pixel on whether it meets a certain criterion, e.g., STAR [1], FAST [35], and ORB [36]. Our observation is that different features indicate different ROIs. As shown in Figure 1, we apply three feature extraction methods, FAST (red points), STAR (green points), and ORB (blue points), to two images. The first column shows the original image and the second column shows the detected feature points. For the image in the first row, the image area with a high density of ORB feature points contains the person who is surfing. For the image in the second row, the image area with a high density of FAST feature points contains the tree. These results confirm that low-level image features correlate to high-level vision apps. More importantly, unlike computation-intensive CNN features [2, 33], these features can be detected in a lightweight manner. Given a target vision app, we expect that the compression algorithm can learn to locate ROI (i.e., the context) by using these features and perform low-bandwidth and low-latency image compression accordingly.

2.2 Image Compression

2.2.1 Traditional Image Compression. Traditional image compression techniques like JPEG [41], JPEG2000 [38] and WebP [15] aim at preserving the visual quality of images. JPEG divides the image into 8×8 macroblocks and operates on the YUV components of them. It mainly consists of three steps, 1) discrete cosine transform (DCT) that extracts DCT coefficients from the YUV components, 2) quantization that divides DCT coefficients in all macroblocks by a quantization table and rounds results to integers, and 3) entropy encoding that applies Huffman coding to the quantized DCT coefficients. Quantization is the step that determines the compression quality of JPEG images. WebP is similar to JPEG in the sense that it also operates on macroblocks and involves DCT, quantization, and entropy encoding. WebP improves on JPEG via predictive coding that uses information in neighboring macroblocks to predict a macroblock. Unlike these techniques that focus on visual quality, CICO focuses on maximizing the accuracy regarding vision apps and minimizing the data to be offloaded.

2.2.2 Machine-Centric Image Compression. Machine-centric image compression techniques can be categorized into CNN-driven compression and server-driven compression.

CNN-driven compression. The autoencoders [2, 3, 33] employ a CNN model to encode an image into a vector and use another CNN model to reconstruct the image from the vector. The autoencoder is able to compress images into a much smaller size than traditional compression techniques, e.g., JPEG. However, their encoding part demands sophisticated CNN models to extract latent features from the image, which places a drastic computation burden on end devices with limited computation capabilities. To deal with this problem, DeepCOD [43] proposes an "imbalanced" autoencoder that consists of a lightweight encoder and a relatively more complex decoder. The limitation of DeepCOD is that heavy computation capability, e.g., GPUs such as Nvidia Titan V and Nvidia GeForce GTX Titan X, are required at the edge server to reconstruct images in real time.

Server-driven compression. Serve-driven compression has been proposed to exploit the server-side ROI feedback to drive spatial quality adaptation at the end devices [9, 25]. The limitation is that the additional delay introduced by device-server communication can lead to excessive end-to-end latency and hamper the spatial quality adaptation. There are also approaches [42] that utilize features of interest provided by scientists to heuristically partition and compress data. However, it is difficult to find the best configuration for this heuristic approach or generalize it to compress a different type of data.

Unlike these CNN-driven and server-driven compression techniques that bring unacceptable end-to-end latency for visual analytics offloading, CICO seeks for a lightweight compression algorithm that would result in a minimal latency in the offloading pipeline. Furthermore, CICO adopts a more generalizable approach that models image compression into an MOO problem and searches for the optimal configuration on the Pareto front without any other prior domain knowledge.

2.3 Multi-Objective Optimization

The multi-objective optimization (MOO) problem targets at the configuration denoted by $\boldsymbol{\theta} = (\theta_1, ...\theta_k) \in \Psi \subseteq \mathbb{R}^k$, where k is the dimension of the configuration and Ψ is the set of all feasible configurations (also known as the design space) in the MOO problem. The goal of the MOO problem is to find configurations that maximize m objective functions, i.e.,

$$\max_{\boldsymbol{\theta} \in \Psi} \boldsymbol{f}(\boldsymbol{\theta}) = (f_1(\boldsymbol{\theta}), ..., f_m(\boldsymbol{\theta})) \subseteq \mathbb{R}^m, \tag{1}$$

where $m = 1, 2, ...$

In the case of $m = 1$, the configurations $\boldsymbol{\theta} \in \Psi$ can be easily ordered according to the objective function $\boldsymbol{f}(\boldsymbol{\theta})$. When $m >= 2$, the *dominance relation* is introduced to partially order configurations in the design space. We say $\boldsymbol{\theta}$ is dominated by $\boldsymbol{\theta}'$ when

$$\boldsymbol{\theta} \prec \boldsymbol{\theta}' = \begin{cases} \theta_i \leq \theta_i' & \forall i = 1, ..., m \\ \theta_i < \theta_i' & \exists i = 1, ..., m \end{cases} \tag{2}$$

If a configuration is not dominated by any other feasible configuration, this configuration is *Pareto optimal*. There exists a set of Pareto optimal configurations Ω such that

$$\Omega = \{\boldsymbol{\theta} | \neg \exists \boldsymbol{\theta}' s.t. \boldsymbol{\theta} \prec \boldsymbol{\theta}', \boldsymbol{\theta}' \in \Psi\}. \tag{3}$$

Ω is also called the *exact Pareto front* of Ψ, which is the solution for the MOO problem. Additionally, any subset $\hat{\Omega} \subseteq \Psi$ is an *approximate Pareto front*. Due to the difficulty in finding the exact Pareto front for certain problems. The goal becomes finding the approximate Pareto front $\hat{\Omega}$, which is as close as possible to the exact Pareto front Ω.

Practical problems like the design of embedded systems [4, 30] and neural network architectures [26, 39] have been modeled and solved as the MOO problem. The main challenge is the large design space, which makes exhaustive search expensive. To address this issue, *design space exploration* (DSE) approaches have been proposed to explore the design space efficiently, which are categorized into heuristics-based and model-based approaches.

Heuristics-based DSE approaches exploit domain knowledge to remove sub-optimal configurations [14, 19], identify the importance of parameters in the configuration [11], or guide the direction of the exploration of configurations [8, 30].

Model-based DSE approaches assume little prior knowledge about the MOO problem but build models to assist DSE, e.g., Non-dominated Sorting Genetic Algorithm II (NGSA II) [7] and Multi-objective Bayesian Optimization (MOBO) [12, 40].

In this paper, we take the first attempt to model image compression in CICO into an MOO problem that simultaneously optimizes the accuracy of the vision app and the offloading bandwidth.

3 SYSTEM OVERVIEW

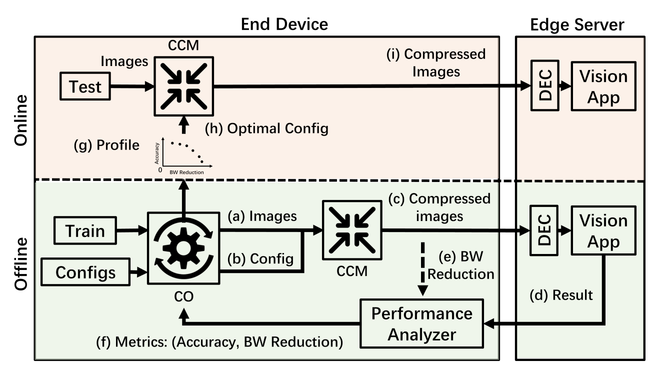


Figure 2: System Architecture

As shown in Figure 2, the architecture of CICO can be split into the offline profiling stage and the online compression stage.

3.1 Offline Profiling Stage

In the offline profiling stage, the *compression optimizer* (CO) interacts with the *context-aware compression module* (CCM) and the vision app to establish the profile for online compression in the following five steps.

1. Initialization. The CO first samples a set of raw images (a) from the training data and selects a configuration (b) to be evaluated.

2. Image Compression. The selected images (a) are compressed by the CCM based on the selected configuration (b). Then, the compressed images (c) will be offloaded to the edge server over the network.

3. Image Processing. After receiving the compressed images (c), the edge server decodes them, performs analysis via CNN models, and sends back the result (d).

4. Metrics Collection. The performance analyzer calculates the accuracy based on the received result (d) and measures the amount of image data reduced for offloading (referred to as the bandwidth reduction for clarity) (e). The metrics (f), including the accuracy and the bandwidth reduction, are sent to the CO.

5. Optimization. The CO receives metrics (f) of the configuration (b) and learns to select the next configuration based on the historical performance of all selected configurations.

The profile (g) consists of explored configurations that are Pareto optimal in terms of accuracy and bandwidth reduction. In other words, the profile is the approximate Pareto front on the training data.

3.2 Online Compression Stage

In the online compression stage, the CCM selects the optimal configuration (h) from the profile (g) based on the bandwidth condition of the end device and the accuracy requirement. The configured CCM compresses images from the testing data and generates compressed images (i). Then, the compressed images are offloaded, decoded, and processed by CNN models in the edge server.

In the following, we present details of the context-aware compression module and the compression optimizer in Section 4 and Section 5, respectively.

4 CONTEXT-AWARE COMPRESSION

The context-aware compression module (Figure 3) consists of feature extraction, context derivation and base compression. Feature extraction and context derivation exploit low-level image features in the input image to derive the context that drives adaptive compression of the base compression.

Feature extraction. Low-level feature extraction distills information from input images efficiently. We start with a set of low-level image features represented by $\Gamma = \{F^{(1)}, ..., F^{(M)}\}$, where $F^{(j)}$ is the j-th image feature and M is the number of classes of features. Common image feature extraction such as STAR [1], FAST [35], and ORB [36] can be applied to the input image I for extracting feature points.

Context Derivation. Context derivation translates low-level image features to the context, which is performed in the following three steps.

1) Tiling. By spatially dividing a raw image I into N equal-sized tiles, where each tile is indexed by $i \in \{1, ..., N\}$, we can get the *vector of feature density* $\boldsymbol{d}^{(j)} = (d_1^{(j)}, ..., d_N^{(j)})$ for the j-th feature, $j = 1, ..., M$. $d_i^{(j)}$ represents the feature density of the j-th feature in the i-th tile. Note that $\sum_{i=1}^{N} d_i^{(j)} = 1$, $j = 1, ..., M$.
2) Weight. We define the *vector of weighted density* $\boldsymbol{\rho}$ to represent the weighted density contributed by all features in each tile, i.e., $\boldsymbol{\rho} = (\rho_1, ..., \rho_N)$. The vector of weights $\boldsymbol{\alpha} = (\alpha_1, ..., \alpha_M)$ describes the importance of different features. The weighted density is calculated as the dot product of the vector of feature density and the vector of weights, i.e., $\rho_i = \sum_{j=1}^{M} \alpha_j d_i^{(j)}$, $i \in \{1, ..., N\}$. Note that $\rho_i \in [0, 1]$ and $\sum_{i=1}^{N} \rho_i = 1$.
3) Nonlinear. We use the *nonlinear function* $g(\cdot; \boldsymbol{\beta})$ defined on $[0, 1]$ to map the vector of weighted density $\boldsymbol{\rho}$ to the vector of compression quality $\boldsymbol{\eta} = (\eta_1, ..., \eta_N)$, where $\eta_i = g(\rho_i; \boldsymbol{\beta}) \in [0, 1]$ indicates the compression quality of the i-th tile. $\boldsymbol{\beta}$ is a hyperparameter. A higher compression quality implies less information loss after compression.

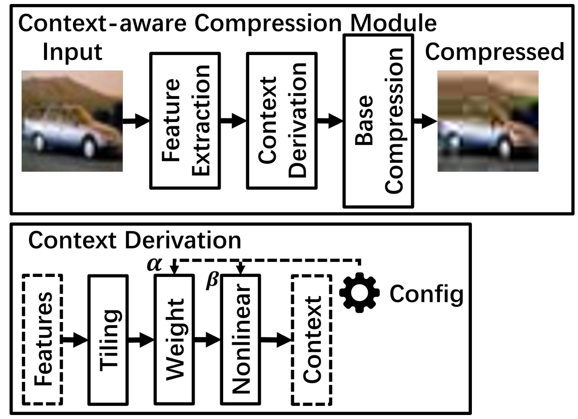


Figure 3: Context-aware Compression Module

Base compression. Base compression utilizes the context to perform adaptive compression with an existing compression method, e.g., JPEG. Specifically, we apply the existing compression method to different tiles in the image I based on the compression quality in that tile. For example, different quantization tables in JPEG can be selected for a tile based on its compression quality. The base compression is denoted by $I' = C(I; \boldsymbol{\eta})$, where C represents the compression operation.

The *compression configuration* is $\boldsymbol{\theta} = (\boldsymbol{\alpha}, \boldsymbol{\beta})$. For clarity, the derivation of the context can be treated as a mapping ξ from the input image I to the compression quality $\boldsymbol{\eta}$, i.e., $\eta = \xi(I; \boldsymbol{\theta})$. Finally, the CCM can be expressed as

$$I' = C(I; \xi(I; \boldsymbol{\theta})). \tag{4}$$

5 COMPRESSION OPTIMIZER

The compression optimizer consists of the exploration optimizer and the data sampler, as shown in Figure 4. The exploration optimizer generates configurations to be evaluated based on the accuracy and bandwidth reduction of previously evaluated configurations. The data sampler randomly samples a subset of the data for each evaluation. We will first formulate image compression via the

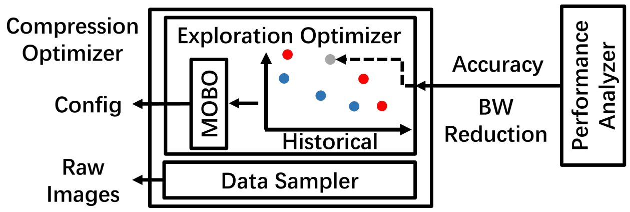


Figure 4: Compression Optimizer

CCM into an MOO problem and discuss challenges. Then, we detail how the challenges are addressed by the exploration optimizer and the data sampler.

5.1 Problem Formulation

With Equation 4, the CCM can transform an image dataset $\mathcal{D}$ into a compressed image dataset $\mathcal{D}^{'} = \{I^{'}|I^{'} = C(I;\xi(I;\boldsymbol{\theta})), I \in \mathcal{D}\}$, which will be sent to the edge server, decoded and processed by CNN models. The configuration affects metrics like the *accuracy* v regarding the vision app and the *bandwidth reduction* r.

The accuracy is calculated based on the result returned by the vision app ((d) in Figure 2) and the ground truth. For simplicity, it is represented by $v = \mathcal{V}(\boldsymbol{\theta};\mathcal{D}) = \mathcal{V}(\boldsymbol{\theta})$, where $\mathcal{V}$ is an abstraction for accuracy metrics like the top-1 accuracy and the mAP.

The bandwidth reduction is calculated by $r = 1 - \frac{\sum_{I^{'} \in \mathcal{D}^{'}} |I^{'}|}{\sum_{I \in \mathcal{D}} |I|} = \mathcal{R}(\boldsymbol{\theta};\mathcal{D}) = \mathcal{R}(\boldsymbol{\theta})$, where $|\cdot|$ represents the size of an image. A higher value of r means a smaller size after compression and more loss of information.

We aim at finding configurations that maximize both the accuracy and the bandwidth reduction, which can be formulated into a multi-objective optimization (MOO) problem as in Equation 5.

$$\max_{\boldsymbol{\theta} \in \Psi} f(\boldsymbol{\theta}) = (\mathcal{V}(\boldsymbol{\theta}), \mathcal{R}(\boldsymbol{\theta})) \subseteq \mathbb{R}^2, \tag{5}$$

where $\Psi = \{\boldsymbol{\theta} \in \mathbb{R}^M | \theta_i \in [0,1], i = 1, ..., M\}$ is the design space. The goal of the compression optimizer is to find the approximate Pareto front $\hat{\Omega} \subseteq \Psi$ of the MOO problem defined in Equation 5.

Challenges. A naive implementation of the compression optimizer can follow these steps to find the approximate Pareto front:

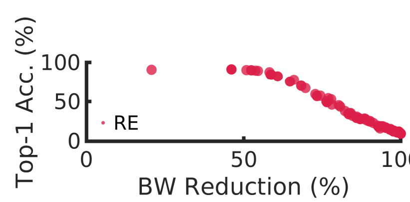
1) draw a random set of configurations from the design space, where each configuration is sampled with the same probability, i.e., randomized exploration (RE),
2) evaluate each configuration over the whole dataset to obtain objectives, i.e., the accuracy and the bandwidth reduction, and
3) find the Pareto front of explored configurations.

However, there are two problems with this naive implementation: 1) *exploration inefficiency*: the infinite design space makes it challenging for RE to obtain a good approximate Pareto front, and 2) *evaluation inefficiency*: it is time-consuming to evaluate objectives over the whole dataset.

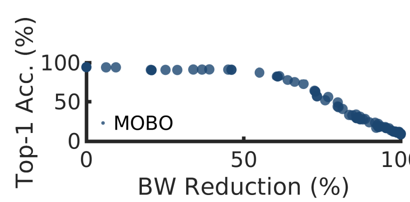
5.2 Exploration Optimizer

Exploration inefficiency. To understand the exploration inefficiency problem, we conduct a preliminary experiment to investigate the offline profiling regarding the vision app based on image classification. It is implemented with Meta Pseudo Labels (MPL) [32], the state-of-the-art image classification method, to classify images in the CIFAR10 dataset [22]. The base compression encodes and decodes the image with the linear interpolation method, which is implemented with the *resize()* function in OpenCV [5]. A lower compression quality means a smaller size after encoding and more information loss. The whole training set of CIFAR10 is used to evaluate objectives, and RE is first adopted to select 100 configurations from the design space. The configurations explored by RE are presented in Figure 5(a), where each point represents the performance of a configuration (top-1 accuracy, bandwidth reduction). We can notice that the configurations on the Pareto front are unevenly sampled. Almost all explored configurations result in a bandwidth reduction over 40% while only one configuration results in a lower bandwidth reduction (roughly 20%). Configurations resulting in lower rates and higher accuracy are rarely explored.

Challenges. We are trying to solve the design space exploration (DSE) problem, which aims at pruning unwanted configurations. Though it has been studied in the design of embedded systems [4, 30] and neural network architectures [26, 39], the design space in these problems is mostly finite, and heuristics can be exploited to solve it. Our problem, however, has an infinite number of configurations, and there is a lack of knowledge of the impact of different knobs in the configuration. AWStream [44] has proposed to scale RE with up to 30 GPUs, but this is not affordable for everyone.



(a) Randomized Exploration



(b) MOBO

Figure 5: Explored configurations by RE and MOBO.

Solution. We address this problem with multi-objective Bayesian optimization (MOBO) [12]. We first set the maximum number of iterations of the algorithm. MOBO models the objectives, i.e., the accuracy and the bandwidth reduction, as drawn from the Gaussian process distribution to capture their relationship with the configuration and to accommodate the noise at the same time. MOBO optimizes the choice of configurations based on the historical performance of all selected configurations such that it 1) correctly locates the Pareto optimal configurations (i.e., red points instead of blue points in Figure 4) and 2) evenly samples Pareto optimal configurations (the gray point in Figure 4). Algorithm 1 illustrates how MOBO is utilized in the design space exploration. We first set the maximum number of iterations N. Then, we initialize the set of Pareto optimal configurations Ω to empty (line 1). Next, we start a loop to iterate over different configurations with MOBO. In this loop, MOBO chooses a configuration $\boldsymbol{\theta}$ based on the history of explored configurations and their performance (line 4). With the chosen configuration $\boldsymbol{\theta}$, we can obtain its performance v and r by running compression and the vision app (line 5). If the chosen configuration is not dominated by any other configurations in the set Ω, we add this configuration to Ω (line 6). Finally, we add the configuration and its performance to the history H. Figure 5(b) demonstrates the optimized exploration achieved by MOBO, where the configurations are more evenly distributed and closer to the exact Pareto front.

Algorithm 1 Design Space Exploration with MOBO

Require: The maximum number of iterations N

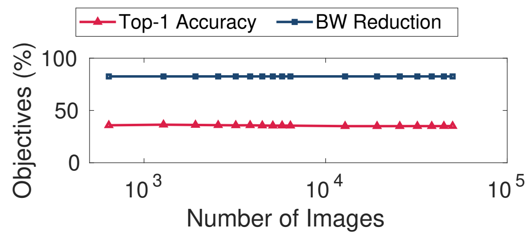
1: $\Omega \leftarrow \{\}$
2: $H \leftarrow \{\}$
3: **for** $k = 1, k{+}{+}, k < N$ **do**
4: $\quad \boldsymbol{\theta} \leftarrow \mathrm{MOBO}(H)$
5: $\quad v \leftarrow \mathcal{V}(\boldsymbol{\theta}), r \leftarrow \mathcal{R}(\boldsymbol{\theta})$
6: $\quad$ **if** $\neg\exists \boldsymbol{\theta}' \in \Omega$, s.t., $\theta \prec \boldsymbol{\theta}'$ **then**
7: $\qquad \Omega \leftarrow \Omega \bigcup \{\boldsymbol{\theta}\}$
8: $\quad H \leftarrow H \bigcup \{(\boldsymbol{\theta}, v, r)\}$

5.3 Data Sampler

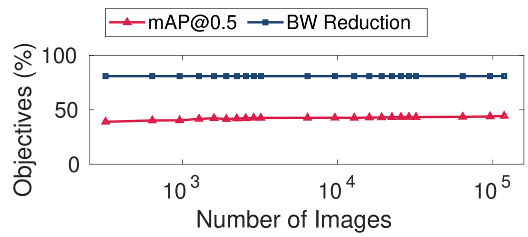
Evaluation inefficiency. To understand the evaluation inefficiency problem, we simulate a vision app that runs YOLOv5 [18], the state-of-the-art object detection technique, on COCO2017 [24], a large-scale dataset for object detection. COCO2017 contains 118, 287 images on its training set, where the objects would take over 5 hours to be detected with an Nvidia RTX 2080 GPU. This indicates that we need over 5 hours to evaluate a single configuration, which is not acceptable considering that finding a good approximate Pareto front usually requires hundreds or even thousands of evaluations. The question is, *do we really need to use the whole dataset to evaluate a single configuration*?

Observations. We conduct an experiment to investigate how the objectives would respond to the change in the size of the dataset. We randomly select a subset of configurations $\mathcal{A} \subseteq \Psi$ and a subset of data $\hat{\mathcal{D}} \subseteq \mathcal{D}$. The accuracy and the bandwidth reduction averaged over configurations in $\mathcal{A}$ can be calculated by $\bar{v} = \frac{1}{|\mathcal{A}|}\sum_{\boldsymbol{\theta}\in\mathcal{A}} \mathcal{V}(\boldsymbol{\theta}; \hat{\mathcal{D}})$ and $\bar{r} = \frac{1}{|\mathcal{A}|}\sum_{\boldsymbol{\theta}\in\mathcal{A}} \mathcal{R}(\boldsymbol{\theta}; \hat{\mathcal{D}})$, respectively. By varying the size of $\hat{\mathcal{D}}$ (referred to as the sampling size), we collect the average values of objectives using different sampling sizes. The results for two vision apps based on image classification (with MPL on CIFAR10) and object detection (with YOLOv5 on COCO2017) are shown in Figure 6(a) and Figure 6(b), respectively. We observe that although there are more than 50k images in CIFAR10 and more than 100k images in COCO2017, the objectives quickly converge and stabilize when the sampling size reaches several thousand.

Solution. Based on this observation, we configure the data sampler to randomly sample 100×32 images in the evaluation of each



(a) Image Classification



(b) Object Detection

Figure 6: Objectives vs. The Sampling Size.

configuration for both image classification and object detection, which significantly accelerates the compression optimizer.

6 EVALUATION

6.1 Methodology

Applications. We evaluate CICO on two vision apps – image classification (CLS) and object detection (DET), respectively.

For CLS, we apply Meta Pseudo Labels (MPL) [32] to classify the CIFAR10 dataset [22]. The CIFAR10 dataset contains 60,000 color images. Each image in the dataset is labeled with one of 10 classes. The goodness metric we adopt is the top-1 accuracy. The CIFAR10 dataset is divided into a training set of 50,000 images and a test set of 10,000 images.

For DET, we apply YOLOv5 [18] to detect objects in the COCO2017 dataset [24]. COCO2017 contains over 120k color images. Each image contains one or multiple objects from 91 categories. The goodness metric we adopt is the mean average precision (mAP). Specifically, we use mAP@0.5 as the metric, which means a bounding box is correct if the intersection over union (IoU) of it and the ground truth is over 0.5. The COCO2017 dataset is divided into a training set of 118,287 images and a test set of 5,024 images.

Hardware. We include two models of IoT end devices – Raspberry Pi 4 Model B and Nvidia Jetson Nano. Raspberry Pi 4 Model B (denoted by RPi) is equipped with a Quad-core Cortex-A72 CPU @ 1.5GHz. Nvidia Jetson Nano (denoted by Nano) is equipped with a Quad-core Cortex-A57 CPU @ 1.5GHz. We also include two types of edge servers. One configuration is a Linux desktop equipped with an Intel Core i9-8950HK CPU @ 2.90GHz ×8 (denoted by i9). The other configuration is a Linux desktop equipped with an Intel Core i7-9700K CPU @ 3.60GHz×12 (denoted by i7). The edge servers are connected to the campus network via a 1Gbps cable. The end devices are connected to the Internet via WiFi or LTE, as detailed below.

Networking. We consider two real-world network conditions in the evaluation – WiFi and LTE. For WiFi, we adopt the 802.11ac standard with a frequency of 5 GHz and a bandwidth of 450Mbps. For LTE, we choose 4G LTE with an upload bandwidth of 50Mbps.

6.2 CICO settings

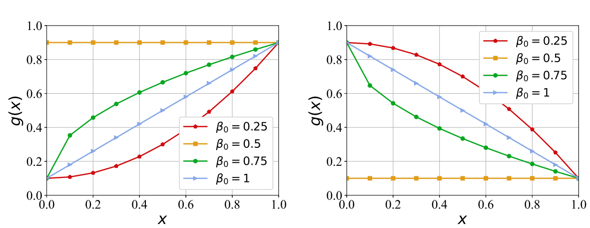
Base compression algorithm. We optimize two base compression algorithms with CICO, i.e., a traditional compression technique and a CNN-based compression technique. For the traditional compression technique, we adopt JPEG [16], the de facto standard for image compression. For the CNN-based compression technique, we adopt the encoder in DeepCOD [43] (denoted by CNN). The image is compressed by a single-layer CNN, a quantization layer, and an entropy encoding layer. In DeepCOD, the image is reconstructed using a sophisticated CNN model consisting of residual networks and self-attention networks. To allow DeepCOD to run on our edge servers with CPU, we adapt the decompression by applying operations of compression in the reverse order, i.e., the decoder of entropy encoding, dequantization, and up-sampling (linear interpolation). The proposed compression techniques are denoted by CICO-J and CICO-C. For CICO-J, we apply JPEG compression to tiles with the quantization table of each tile selected based on the CICO-derived compression quality. A higher value of the compression quality means smaller values in the quantization table and less information loss. For CICO-C, we apply single-layer convolution to tiles with the stride of the convolution kernel, which is equal to the size of the kernel, chosen based on the compression quality of the tile. A higher value of the compression quality means a smaller stride of the kernel and less information loss.

Low-level image feature. Considering the running time and the performance of different features in image classification and object detection, we use FAST [35], SIFT [28], and good features to track [37] in image classification, and STAR [1], FAST and ORB [36] in object detection.

Nonlinear function. The nonlinear function in the context-aware compression module (Figure 3) is defined as shown in Equation 6.

$$g(x;\boldsymbol{\beta}) = \begin{cases} \beta_1 + (\beta_2 - \beta_1) * x^{2\beta_0 - 1} & \beta_0 \in [0.5, 1] \\ \beta_1 + (\beta_2 - \beta_1) * x^{\frac{1}{1-2\beta_0}} & \beta_0 \in [0, 0.5), \end{cases} \tag{6}$$

where $x \in [0,1]$ and $\boldsymbol{\beta} = (\beta_0, \beta_1, \beta_2)$, $\beta_k \in [0,1]$, $k = 0, 1, 2$. Figure 7 shows the shape of the nonlinear function under different configurations of $\boldsymbol{\beta}$.



(a) $\beta_1 = 0.1, \beta_2 = 0.9$ (b) $\beta_1 = 0.9, \beta_2 = 0.1$

Figure 7: Illustration of the nonlinear function.

6.3 Accuracy-Bandwidth Trade-off

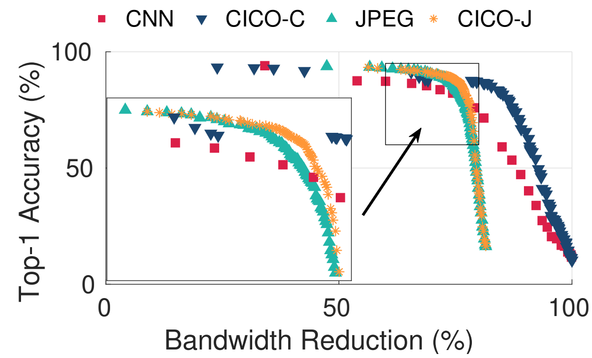


Figure 8: Accuracy-bandwidth trade-offs (CLS).

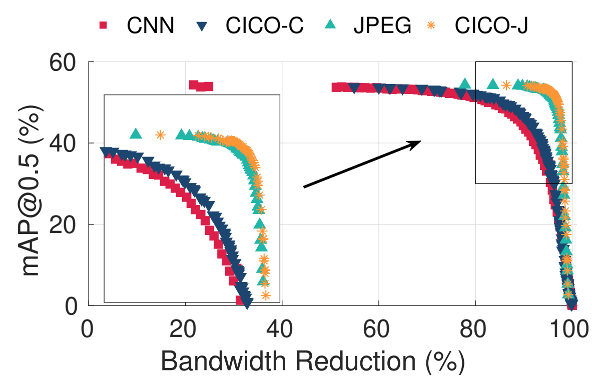
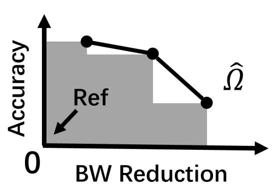


Figure 9: Accuracy-bandwidth trade-offs (DET).

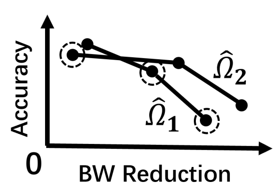
In this subsection, we evaluate the accuracy-bandwidth trade-offs of different approaches. Figure 8 and Figure 9 show the accuracy-bandwidth trade-offs evaluated with MPL and YOLOv5, respectively, where each point represents the bandwidth reduction and the top-1 accuracy/mAP@0.5 of a configuration. We observe that CICO-J and CICO-C outperform JPEG and the CNN-based encoder, respectively (curves higher in the figure). For example, in Figure 8, compared to the bandwidth reduction of 76.9% and the top-1 accuracy of 79.8% achieved by the CNN-based encoder, CICO-C can achieve a bandwidth reduction of 86.1% and a top-1 accuracy of 79.9%. In other words, CICO reduces the size of compressed images by around 40% over the CNN encoder at the same level of top-1 accuracy. This is because CICO optimizes the accuracy-bandwidth trade-off while considering the spatial differentiation among different image regions. However, this is not addressed in the existing methods. On one hand, JPEG does not consider the analytics accuracy in the design space. On the other hand, the CNN encoder is essentially a fixed-length encoder that does not address ROI because the convolution is equally applied to different image regions.

In addition, we observe that, by comparing the curves of CICO-J and CICO-C versus JPEG and CNN respectively, CICO demonstrates more improvement near the center of the curve while less improvement at both ends of the curve. The reason is that when the bandwidth reduction is close to the lower bound or upper bound of the base compression, CICO tends to choose configurations that assign the highest or the lowest compression quality to all tiles, respectively. Near the center of the curve, CICO can reassign and adapt the compression quality of tiles in a more effective way to improve the accuracy-bandwidth trade-off.

To statistically evaluate the improvement in the accuracy-bandwidth trade-off, i.e., the Pareto front on the test data with the optimal configuration, we introduce two metrics: *hypervolume* and *coverage* [29]. Hypervolume $\mathcal{H}$ measures the area dominated by a Pareto front with respect to a reference point. In our evaluation, the reference point is set to $(0, 0)$. Figure 10(a) shows the hypervolume of a Pareto front consisting of 3 configurations, which is represented by the area of the gray regions. A higher value in the hypervolume indicates a better accuracy-bandwidth trade-off. Coverage $\chi(\hat{\Omega}_1, \hat{\Omega}_2)$ calculates the percentage of configurations in $\hat{\Omega}_1$ that is dominated by $\hat{\Omega}_2$. We say a configuration is dominated by a Pareto front if any configuration on that Pareto front dominates the configuration. In Figure 10(b), the configurations dominated by $\hat{\Omega}_1$ or $\hat{\Omega}_2$ are surrounded by dashed circles. We can find $\chi(\hat{\Omega}_1, \hat{\Omega}_2) = 2/3$. and $\chi(\hat{\Omega}_2, \hat{\Omega}_1) = 1/3$. A higher coverage implies a relatively better performance in the accuracy-bandwidth trade-off.



(a) Hypervolume



(b) Coverage

Figure 10: Illustration of Metrics.

Table 1 shows the hypervolume of different approaches. The hypervolume of CICO-J and CICO-C outperforms that of JPEG and the CNN-based encoder in image classification and object detection, respectively. The coverage of different pairs of approaches are presented in Table 2 (MPL) and Table 3 (YOLOv5). It is shown that the accuracy-bandwidth trade-offs of CICO-J (CICO-C) dominate most (over 70%) of that of JPEG (CNN).

Overall, CICO improves the accuracy-bandwidth trade-off of JPEG and the CNN-based encoder in vision apps of both image classification and object detection. This is mainly attributed to the data sampler and exploration optimizer of CICO that were introduced in Section 5.

Table 1: Hypervolume.

	JPEG	CNN	CICO-J	CICO-C
CLS	0.736	0.79	0.737	0.847
DET	0.531	0.506	0.532	0.509

Table 2: Coverage $\chi(\hat{\Omega}_1, \hat{\Omega}_2)$ (CLS).

$\hat{\Omega}_1$ \ $\hat{\Omega}_2$	JPEG	CNN	CICO-J	CICO-C
JPEG	0	24	5.1	11.4
CNN	79.1	0	30.8	8.6
CICO-J	<u>81.4</u>	28.0	0	11.4
CICO-C	76.7	<u>84.0</u>	62.8	0

Table 3: Coverage $\chi(\hat{\Omega}_1, \hat{\Omega}_2)$ (DET).

$\hat{\Omega}_1$ \ $\hat{\Omega}_2$	JPEG	CNN	CICO-J	CICO-C
JPEG	0	85.2	0	89.4
CNN	3.6	0	4.3	18.8
CICO-J	<u>92.7</u>	83.6	0	89.4
CICO-C	3.6	<u>70.5</u>	4.3	0

6.4 End-to-end Analysis

In this subsection, we conduct an analysis to show how CICO affects the end-to-end performance of visual analytics offloading, including the end-to-end offloading latency and the system processing speed. To study the impact of CICO on different hardware architectures, we build four hardware architectures based on the choices of the IoT end devices (RPi and Nano) and the edge servers (i9 and i7). Each hardware architecture integrates different IoT end devices and edge servers, which are denoted by RPi+i9, RPi+i7, Nano+i9, and Nano+i7. For a fair comparison of the end-to-end performance, we make sure the difference of accuracy between two compression approaches is less than 1% in both image classification and object detection. In image classification, the top-1 accuracy of all approaches is configured to near 85%. In object detection, the mAP@0.5 of all approaches are configured to be near 50%.

The end-to-end offloading latency consists of the encoding latency (enc), the network transmission latency (net), and the decoding latency (dec). Figure 11 and Figure 12 present the end-to-end offloading latency in image classification using WiFi and LTE, respectively. Figure 13 and Figure 14 present the end-to-end offloading latency in object detection using WiFi and LTE, respectively. We observe that the end-to-end offloading latency is reduced by CICO for CNN and JPEG in most hardware architectures and network conditions. For example, Figure 14 shows that CICO reduces the end-to-end latency for the CNN encoder and JPEG by 35% and 15%, respectively. This is because CICO significantly reduces the network transmission latency by optimizing the compression algorithm and achieving a higher bandwidth reduction at a similar analytics accuracy. The overhead of the CICO computation is the slightly increased encoding and decoding latency. However, as can be seen from the figures, the computation cost of utilizing low-level image features introduced by CICO is negligible in general.

A few exceptions are found when using CICO to compress images for offloading in WiFi. In these cases, the end-to-end offloading latency is several milliseconds higher in CICO (e.g., Figure 11). The reason for the increased latency is that the image size (32×32) is relatively small while the network bandwidth in our ideal office WiFi (several hundred Mbps) is significantly high. As a result, the reduced network transmission latency is not sufficient to compensate for the encoding/decoding latency added by CICO. However, we point out that this phenomenon is unlikely to happen in more realistic situations in practice where the IoT environment has significantly lower and unstable bandwidth (similar to or worse than LTE) and the image data to be offloaded are generally larger. We will also show in the following that such minor latency discrepancy does not affect the expedited performance of the whole CICO offloading pipeline.

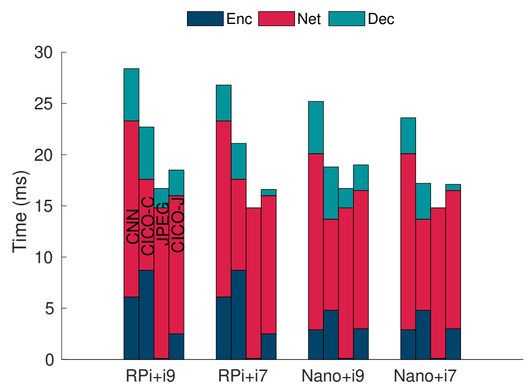


Figure 11: End-to-end offloading latency using WiFi (CLS).

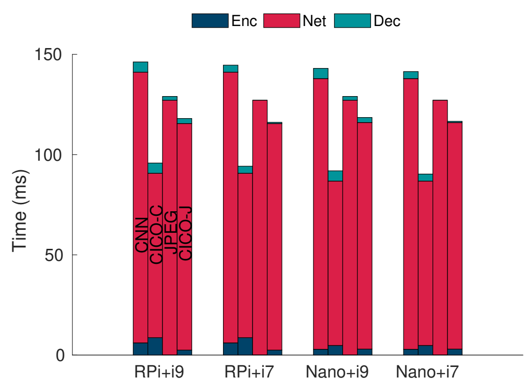


Figure 12: End-to-end offloading latency using LTE (CLS).

Since component-wise and end-to-end latency evaluate the performance of a system rather than the quality of service that can be delivered by a system, we evaluate the end-to-end processing speed to examine the quality of service of the visual analytics offloading. The processing speed is determined by the highest component latency among encoding, network transmission, and decoding. Unlike the absolute numbers of latency, the processing speed provides users and system designers an intuitive way to understand how CICO can achieve ultra-fast visual analytics offloading compared to state-of-the-art compression techniques. Figure 15 and Figure 16 demonstrate the processing speed in image classification using WiFi and LTE, respectively. Figure 17 and Figure 18 demonstrate the processing speed in object detection using WiFi and LTE, respectively.

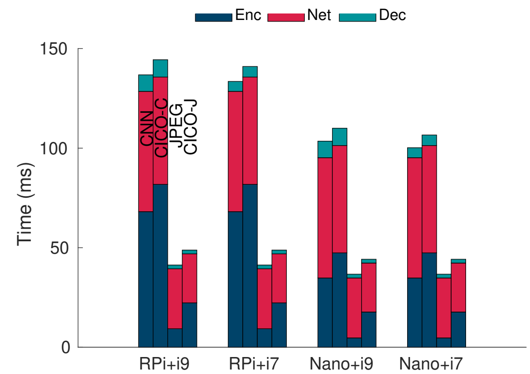


Figure 13: End-to-end offloading latency using WiFi (DET).

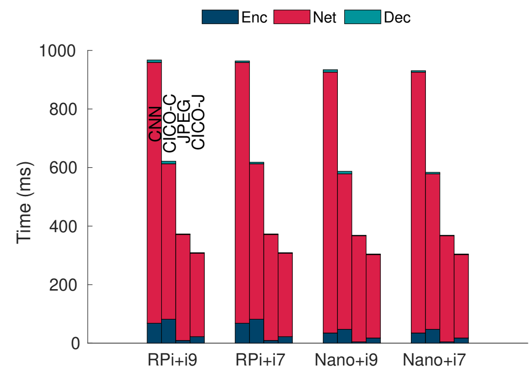


Figure 14: End-to-end offloading latency using LTE (DET).

Comparing the processing speed with and without CICO, we can find that the processing speed has been significantly improved by CICO in different hardware architectures and network conditions. We observed up to a 2× speed up of the visual analytics offloading pipeline among all these scenarios. The results of end-to-end processing speed confirm that CICO is faster and more appropriate than existing compression techniques for time-sensitive vision apps that require a higher frame processing rate in visual analytics offloading.

In sum, CICO reduces the end-to-end offloading latency and improves the processing speed for JPEG and the CNN-based encoder in most hardware architectures and network conditions.

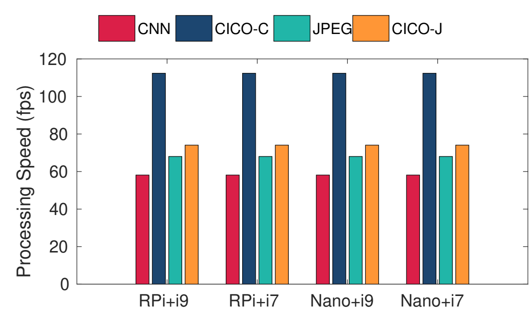


Figure 15: Processing speed using WiFi (CLS).

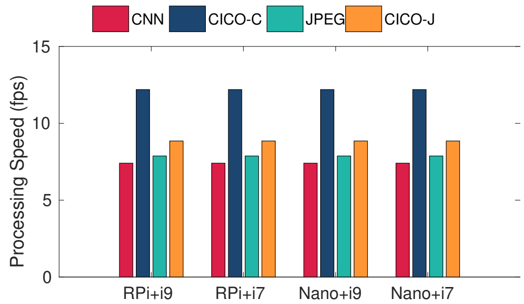


Figure 16: Processing speed using LTE (CLS).

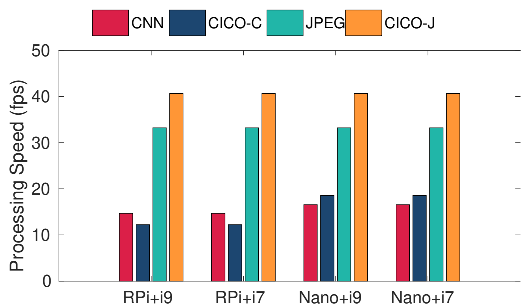


Figure 17: Processing speed using WiFi (DET).

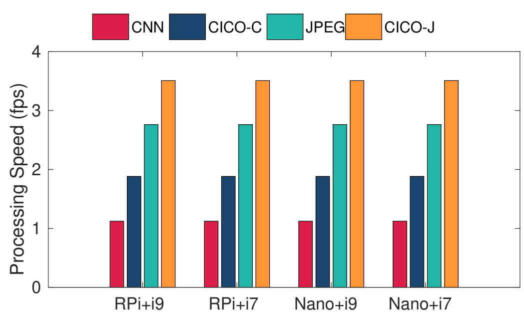


Figure 18: Processing speed using LTE (DET).

6.5 Profiling Cost

The profiling involves running the offline profiling stage (Figure 2) for two base compression modules on two applications, CLS and DET, which results in four offline profiling stages. We set the number of configurations to explore to be 500. As discussed in Section 5.3, 100×32 samples will be used for each configuration. In each offline profiling stage, a total of $500 \times 100 \times 32 = 1,600,000$ images will be encoded, transmitted, decoded, and processed by the application. The profiling is performed on a Linux server equipped with two Nvidia GeForce RTX 2080 GPUs. For image classification, the offline profiling for each compression approach takes about 20 hours. For object detection, the profiling for each compression approach takes about 40 hours. Our proposed offline profiling method allows CICO to learn from the images and the vision apps in a reasonable period of time.

6.6 Profiling Error

To demonstrate the difference of the profile obtained using the training data and the performance of it on the test data, we introduce the *profiling error*. It is defined as the absolute difference in the accuracy (or the bandwidth reduction) of the configuration measured with the training data and the test data. The profiling error is averaged over all configurations on the profile (of CICO-J and CICO-C for two vision apps) and shown in Table 4. We can notice that the profiling errors of the bandwidth reduction and the accuracy are small in general. This indicates that system designers can choose the configuration on the profile to optimize the utilization of the bandwidth resource on the IoT end device.

Table 4: Profiling Error.

	Accuracy	BW Reduction
CLS	$2.7(\pm 1.8)\%$	$0.026(\pm 0.049\%)$
DET	$4.3(\pm 1.5)\%$	$0.34(\pm 0.23)\%$

7 DISCUSSION

Choice of the low-level image features. One advantage of our approach is that the system designer does not have to understand how each type of low-level image feature affects the overall compression performance. Instead, our approach automatically learns how to exploit different low-level image features in image compression. The system designer only needs to include a few well-known features [1, 28, 35–37] and make sure the running time of the optimized compression approach, which includes the time spent in feature extraction and compression, is acceptable. For example, the running time should be kept under 33 ms for real-time applications at an offloading speed of 30 fps.

Choice of the nonlinear function. The nonlinear function models the relationship between the feature density and the compression quality. We selected the one in Equation 6 to strike a trade-off between training complexity and compression performance. The nonlinear function can be defined in other forms as long as it maps a density value in $[0, 1]$ to a compression quality value in $[0, 1]$. More parameters could be included in the nonlinear function to allow our approach to better model the relationship between the feature density and the compression quality. However, the downside is that the design space of our system would be larger, which would take longer for the compression optimizer to learn the optimal set of parameters.

Choice of the base compression module. The choice of the compression method is generally flexible. It can be any traditional, e.g., JPEG, or machine learning-based, e.g., DeepCOD, compression method. The base compression method would need to be configured in a way that it could compress different image tiles with different qualities. The other consideration is that an excessively complicated compression method should not be used because the benefits introduced by CICO in bandwidth reduction and network latency reduction might be offset by the additional delay incurred in the encoding and decoding modules.

The vision-based application. In addition to image classification and object detection, our approach is generic and can be applied to other vision-based applications like car counting [27] and action detection [20]. As long as a vision app explicitly gives out a metric that can evaluate the performance of an image dataset, CICO can be used to learn the dataset and enhance the visual analytics offloading performance.

8 CONCLUSION

We present CICO, a novel compression framework that contextualizes and optimizes image compression for visual analytics offloading in IoT. CICO is the first low-bandwidth and low-latency compression framework that optimizes the accuracy and the bandwidth in visual analytics offloading. The compression problem is formulated as an MOO problem and the Pareto front of the MOO problem is approximated by an MOBO-based exploration optimizer and an efficient data sampler. We evaluate the performance of CICO in extensive experimental settings. Our results show that, compared to state-of-the-art compression approaches, CICO elevates the accuracy-bandwidth trade-off and the end-to-end quality of service of visual analytics offloading in IoT.

REFERENCES

[1] Motilal Agrawal, Kurt Konolige, and Morten Rufus Blas. 2008. Censure: Center surround extremas for realtime feature detection and matching. In *European Conference on Computer Vision*. Springer, 102–115.

[2] Eirikur Agustsson, Michael Tschannen, Fabian Mentzer, Radu Timofte, and Luc Van Gool. 2019. Generative adversarial networks for extreme learned image compression. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 221–231.

[3] Johannes Ballé, Valero Laparra, and Eero P Simoncelli. 2016. End-to-end optimized image compression. *arXiv preprint arXiv:1611.01704* (2016).

[4] Giovanni Beltrame, Luca Fossati, and Donatella Sciuto. 2010. Decision-theoretic design space exploration of multiprocessor platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 29, 7 (2010), 1083–1095.

[5] G. Bradski. 2000. The OpenCV Library. *Dr. Dobb's Journal of Software Tools* (2000).

[6] Zhaowei Cai, Mohammad Saberian, and Nuno Vasconcelos. 2015. Learning complexity-aware cascades for deep pedestrian detection. In *Proceedings of the IEEE International Conference on Computer Vision*. 3361–3369.

[7] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation* 6, 2 (2002), 182–197.

[8] Robert P Dick and Niraj K Jha. 1997. MOGAC: A multiobjective genetic algorithm for the co-synthesis of hardware-software embedded systems. In *iccad*, Vol. 97. 522–529.

[9] Kuntai Du, Ahsan Pervaiz, Xin Yuan, Aakanksha Chowdhery, Qizheng Zhang, Henry Hoffmann, and Junchen Jiang. 2020. Server-Driven Video Streaming for Deep Learning Inference. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 557–570.

[10]]pascal-voc-2012 M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. [n. d.]. The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results. http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html.

[11] William Fornaciari, Donatella Sciuto, Cristina Silvano, and Vittorio Zaccaria. 2002. A sensitivity-based design space exploration methodology for embedded systems. *Design Automation for Embedded Systems* 7, 1 (2002), 7–33.

[12] Paulo Paneque Galuzio, Emerson Hochsteiner [de Vasconcelos Segundo], Leandro dos Santos Coelho, and Viviana Cocco Mariani. 2020. MOBOpt — multi-objective Bayesian optimization. *SoftwareX* 12 (2020), 100520. https://doi.org/10.1016/j.softx.2020.100520

[13] Ross Girshick. 2015. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*. 1440–1448.

[14] Tony Givargis, Frank Vahid, and Jörg Henkel. 2001. System-level exploration for pareto-optimal configurations in parameterized systems-on-a-chip. In *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No. 01CH37281)*. IEEE, 25–30.

[15] Google. 2020. *A new image format for the Web*. https://developers.google.com/speed/webp

[16] The Independent JPEG Group. 2014. *libjpeg*. https://github.com/LuaDist/libjpeg

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[18] Glenn Jocher, Alex Stoken, Jirka Borovec, NanoCode012, ChristopherSTAN, Liu Changyu, Laughing, tkianai, Adam Hogan, lorenzomammana, yxNONG, AlexWang1900, Laurentiu Diaconu, Marc, wanghaoyang0106, ml5ah, Doug, Francisco Ingham, Frederik, Guilhen, Hatovix, Jake Poznanski, Jiacong Fang, Lijun Yu, changyu98, Mingyu Wang, Naman Gupta, Osama Akhtar, PetrDvoracek, and Prashant Rai. 2020. *ultralytics/yolov5: v3.1 - Bug Fixes and Performance Improvements*. https://doi.org/10.5281/zenodo.4154370

[19] Eunsuk Kang, Ethan Jackson, and Wolfram Schulte. 2010. An approach for effective design space exploration. In *Monterey Workshop*. Springer, 33–54.

[20] Okan Köpüklü, Xiangyu Wei, and Gerhard Rigoll. 2019. You only watch once: A unified cnn architecture for real-time spatiotemporal action localization. *arXiv preprint arXiv:1911.06644* (2019).

[21] N Krishnaraj, Mohamed Elhoseny, M Thenmozhi, Mahmoud M Selim, and K Shankar. 2020. Deep learning model for real-time image compression in Internet of Underwater Things (IoUT). *Journal of Real-Time Image Processing* 17, 6 (2020), 2097–2111.

[22] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[23] Yuanqi Li, Arthi Padmanabhan, Pengzhan Zhao, Yufei Wang, Guoqing Harry Xu, and Ravi Netravali. 2020. Reducto: On-Camera Filtering for Resource-Efficient Real-Time Video Analytics. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 359–376.

[24] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. 2014. Microsoft coco: Common objects in context. In *European conference on computer vision*. Springer, 740–755.

[25] Luyang Liu, Hongyu Li, and Marco Gruteser. 2019. Edge assisted real-time object detection for mobile augmented reality. In *The 25th Annual International Conference on Mobile Computing and Networking*. 1–16.

[26] Zhichao Lu, Ian Whalen, Vishnu Boddeti, Yashesh Dhebar, Kalyanmoy Deb, Erik Goodman, and Wolfgang Banzhaf. 2019. Nsga-net: neural architecture search using multi-objective genetic algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 419–427.

[27] Thomas Moranduzzo and Farid Melgani. 2013. Automatic car counting method for unmanned aerial vehicle images. *IEEE Transactions on Geoscience and Remote Sensing* 52, 3 (2013), 1635–1647.

[28] Pauline C Ng and Steven Henikoff. 2003. SIFT: Predicting amino acid changes that affect protein function. *Nucleic acids research* 31, 13 (2003), 3812–3814.

[29] Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. 2009. Respir: A response surface-based pareto iterative refinement for application-specific design space exploration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28, 12 (2009), 1816–1829.

[30] Maurizio Palesi and Tony Givargis. 2002. Multi-objective design space exploration using genetic algorithms. In *Proceedings of the tenth international symposium on Hardware/software codesign*. 67–72.

[31] Omkar M Parkhi, Andrea Vedaldi, and Andrew Zisserman. 2015. Deep face recognition. (2015).

[32] Hieu Pham, Qizhe Xie, Zihang Dai, and Quoc V Le. 2020. Meta pseudo labels. *arXiv preprint arXiv:2003.10580* (2020).

[33] Aaditya Prakash, Nick Moran, Solomon Garber, Antonella DiLillo, and James Storer. 2017. Semantic perceptual image compression using deep convolution networks. In *2017 Data Compression Conference (DCC)*. IEEE, 250–259.

[34] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 779–788.

[35] Edward Rosten and Tom Drummond. 2006. Machine learning for high-speed corner detection. In *European conference on computer vision*. Springer, 430–443.

[36] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. 2011. ORB: An efficient alternative to SIFT or SURF. In *2011 International Conference on Computer Vision*. 2564–2571. https://doi.org/10.1109/ICCV.2011.6126544

[37] Jianbo Shi et al. 1994. Good features to track. In *1994 Proceedings of IEEE conference on computer vision and pattern recognition*. IEEE, 593–600.

[38] Athanassios Skodras, Charilaos Christopoulos, and Touradj Ebrahimi. 2001. The jpeg 2000 still image compression standard. *IEEE Signal processing magazine* 18, 5 (2001), 36–58.

[39] Sean C Smithson, Guang Yang, Warren J Gross, and Brett H Meyer. 2016. Neural networks designing neural networks: multi-objective hyper-parameter optimization. In *Proceedings of the 35th International Conference on Computer-Aided Design*. 1–8.

[40] Shinya Suzuki, Shion Takeno, Tomoyuki Tamura, Kazuki Shitara, and Masayuki Karasuyama. 2020. Multi-objective bayesian optimization using pareto-frontier entropy. In *International Conference on Machine Learning*. PMLR, 9279–9288.

[41] Gregory K Wallace. 1992. The JPEG still picture compression standard. *IEEE transactions on consumer electronics* 38, 1 (1992), xviii–xxxiv.

[42] Chaoli Wang, Hongfeng Yu, and Kwan-Liu Ma. 2009. Application-driven compression for visualizing large-scale time-varying data. *IEEE Computer Graphics and Applications* 30, 1 (2009), 59–69.
[43] Shuochao Yao, Jinyang Li, Dongxin Liu, Tianshi Wang, Shengzhong Liu, Huajie Shao, and Tarek Abdelzaher. 2020. Deep compressive offloading: speeding up neural network inference by trading edge computation for network latency. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*. 476–488.
[44] Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A Lee. 2018. Awstream: Adaptive wide-area streaming analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 236–252.