Pervasive and Mobile Computing 83 (2022) 101594

Contents lists available at ScienceDirect

Pervasive and Mobile Computing

journal homepage: www.elsevier.com/locate/pmc

Memory-aware and context-aware multi-DNN inference on the edge

Bart Cox [a,*], Robert Birke [b], Lydia Y. Chen [a]

[a] TU Delft, Van Mourik Broekmanweg 6, Delft, 2628 XE, The Netherlands
[b] ABB Corporate Research Switzerland, Segelhofstrasse 1K, Baden-Daetwil, 5405, Switzerland

ARTICLE INFO

Article history:
Received 16 September 2021
Received in revised form 16 February 2022
Accepted 11 April 2022
Available online 18 April 2022

Keywords:
Multiple DNNs inference
Average response time
Edge devices
Memory-aware scheduling

ABSTRACT

Deep neural networks (DNNs) are becoming the core components of many applications running on edge devices, especially for real time image-based analysis. Increasingly, multi-faced knowledge is extracted by executing multiple DNNs inference models, e.g., identifying objects, faces, and genders from images. It is of paramount importance to guarantee low response times of such multi-DNN executions as it affects not only users quality of experience but also safety. The challenge, largely unaddressed by the state of the art, is how to overcome the memory limitation of edge devices without altering the DNN models. In this paper, we design and implement MASA, a responsive memory-aware multi-DNN execution and scheduling framework, which requires no modification of DNN models. The aim of MASA is to consistently ensure the average response time when deterministically and stochastically executing multiple DNN-based image analyses. The enabling features of MASA are (i) modeling inter- and intra-network dependency, (ii) leveraging complimentary memory usage of each layer, and (iii) exploring the context dependency of DNNs. We verify the correctness and scheduling optimality via mixed integer programming. We extensively evaluate two versions of MASA, context-oblivious and context-aware, on three configurations of Raspberry Pi and a large set of popular DNN models triggered by different generation patterns of images. Our evaluation results show that MASA can achieve lower average response times by up to 90% on devices with small memory, i.e., 512 MB to 1 GB, compared to the state of the art multi-DNN scheduling solutions.

1. Introduction

Edge devices, such as cameras, wearables, and sensors, are becoming an integral part of our daily life and increasingly draw on deep neural networks (DNNs) for sophisticated real-time image-based analysis. Instagram [1] enables real-time knowledge extraction upon captures of images by running the *inference* of convolutional neural networks (CNNs) – one of the most widely adopted type of DNN for image processing – directly on users' devices [2,3]. Autonomous vehicles and surveillance systems are other examples that need to execute DNN inferences to recognize the surroundings and facilitate on-line decision making. It is imperative to ensure the responsiveness of DNN inference, i.e., low response time, even on edge devices for the quality of experience as well as safety.

To extract the numerous information embedded in a single image, inference with multiple different DNNs needs to be executed — greatly increasing the computational overhead and risk of unresponsiveness. Let us take as an example

* Corresponding author.
E-mail addresses: b.a.cox@tudelft.nl (B. Cox), robert.birke@ch.abb.com (R. Birke), lydiaychen@ieee.org (L.Y. Chen).

https://doi.org/10.1016/j.pmcj.2022.101594

identifying the number of females in an image. To complete this inference task [4], a first DNN model (i.e. FaceNet) identifies the number and location of faces. Then, a second DNN model (i.e. GenderNet) classifies faces into male and female. If an image does not contain any faces, GenderNet can be skipped. The execution flow of multiple DNNs naturally depends on the analysis and the images. We term multi-DNN inference job the analysis of the same image processed by a set of DNN models. In addition to the number of DNN models within a job, the performance of DNN inference highly depends on how images are generated, e.g., periodically or stochastically. More predictable the arrival patterns and sizes of inference jobs, more straightforward to manage the response time [5]. Big inference jobs composed of a high number of large DNN models can easily increase the response time of small inference jobs that are unfortunately scheduled behind.

In contrast to desktop and server-grade systems, edge devices are known to be constrained in resources, both CPU and memory. At the same time, the high accuracy of complex DNNs comes at the cost of intensive memory footprint and computation cost. Typical CNN models [6] can take up hundreds of MB memory to store tens of millions model parameters. Executing the convolution layers of CNNs is CPU intensive, whereas computing the fully-connected layers is more memory intensive due to the high number of model weights. It is no mean feat to run multi-DNN inference jobs on edge devices.

To turn DNNs edge inference from infeasible to reality, the related studies take two orthogonal directions: (i) minimizing the resource demands of DNNs, and (ii) exploring complementary DNNs resource patterns. The former [7,8] aim to compress and prune models for *individual* networks — achieving a calculable trade-off with accuracy. However, the main focus is on single-DNN and provides no additional support for multi-DNN. To execute multi-DNN inference jobs, existing DNN frameworks, e.g., PyTorch [9], and Caffe [10], only execute one model at a time, sequentially completing all DNNs. The later take a step beyond single-DNN by considering complementary resource usages [7] or intra-network dependencies [8] without altering the DNN structure and degrading model accuracy. They shed light on how multi-DNN inferences can be accelerated on devices with narrow resources. However, more realistic workloads, e.g., images are generated stochastically, and dynamic multi-DNNs execution flows, are neglected.

In this paper, we design Masa, a highly responsive and memory-aware multi-DNN execution framework. It is an on-device middleware. We aim to achieve low average response times for extracting multiple information from captured images on edge devices. To efficiently multiplex the limited CPU and memory resources across DNN models, Masa uniquely considers inter- and intra-DNN dependency. For intra-DNN dependency, we model the resource demands in per layer granularity. Masa opportunistically schedules the loading and execution of each layer as to leverage complimentary resource usages of different layers, i.e., interleaving the loading and execution of convolution layers (CPU intensive) and fully-connected layers (memory intensive). To handle the challenging stochastic workloads, Masa greedily executes DNN layers by the principle of smallest memory first followed by their generation order, avoiding the performance penalty of memory swap. For inter-DNN dependency, we explicitly consider the image context and factor in its impact in the execution flow. For instance, the execution of GenderNet in the previous example of identifying females in an image hinges on the inference outcome of FaceNet. Masa can conservatively load and execute the dependent DNN only upon receiving the positive signal from its predecessor. This is the context-aware Masa, which can significantly save resources compared to the context-oblivious Masa [11]. We first analytically show that memory-aware scheduling is probably NP-hard, and then evaluate the optimality of the proposed scheduling heuristics via mixed integer programming.

Masa has two key components: an offline network preparator and an online scheduler. The network preparator splits each DNN by layers and estimates each layer peak memory demand. The scheduler handles layer loading and execution of all DNNs by dynamically checking the inter-network dependency, available memory, and estimated memory demands. We implement Masa on top of Caffe. We extensively evaluate Masa on a large set of multi-DNN inference scenarios and a representative set of hardware, namely Raspberry Pi And VM configurations. Our evaluation shows that the memory-aware features of Masa significantly reduces the response times compared to the state of the art multi-DNN engines.

The contributions of Masa are multifold. First of all, Masa is a first of its kind memory-aware and context-aware multi-DNN execution middleware on edge devices (Section 4). Second, the scheduler of Masa can simultaneously consider complimentary resource patterns of DNN layers and handle inter-DNN and intra-DNN dependencies when facing stochastically and deterministically generated images (Section 4.3). Third, Masa is extensively proven to minimize the average response times of multi-DNN inferences for real-world applications executed on resource-constrained edge devices (Section 5). Fourth, Masa is analytically validated via mixed-integer programming (Section 3), showing close to optimal performance with the advantage of scalability and agility required by real world problems (Section 5.6).

2. Preliminary and motivation

CNNs are one of the main types of DNNs for image-based inference. They combine convolution and pooling layers for feature extraction and complementary fully-connected layers for classification [6]. Fig. 1 presents an exemplary architecture. CNNs are inspired by the organization of the visual cortex. Individual neurons answer to stimuli only in a restricted region of the visual field named receptive field. Multiple such fields overlap to cover the entire visual area.

Convolutional layers divide the image in receptive fields from which they extract features via convolution with a filter matrix shared across all receptive fields. Initial layers capture low-level features, e.g., edges, color. Subsequent layers combine low-level characteristics into higher-level features which allow the network to gain a thorough understanding

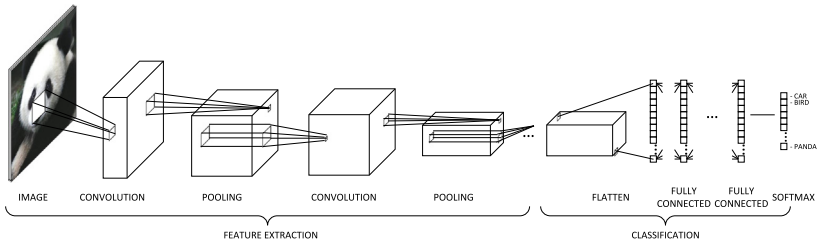


Fig. 1. Exemplary CNN architecture comprising convolutional, pooling and fully-connected layers. Flatten reshapes the input to a 1D array. Softmax performs the final classification.

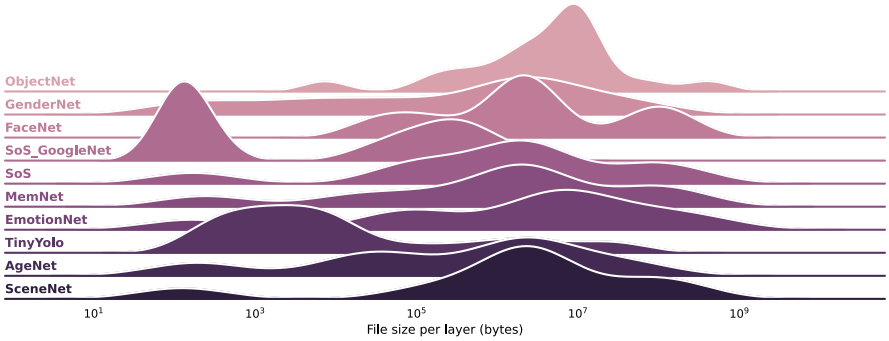


Fig. 2. Layer size distribution for different networks. The *y*-axis shows the density of the distribution. The more layers of a certain size are in a model, the higher the density.

of the images. Pooling layers apply aggregation functions, e.g. `max`, on fields of convolved features. This reduces noise and extracts dominant positional and rotational invariant features. Fully-connected layers learn non-linear combinations of extracted high-level features and classify the image using the `softmax`. Different CNN architectures vary the number and hyper-parameters of these layers. Fig. 2 depicts examples for different CNNs.

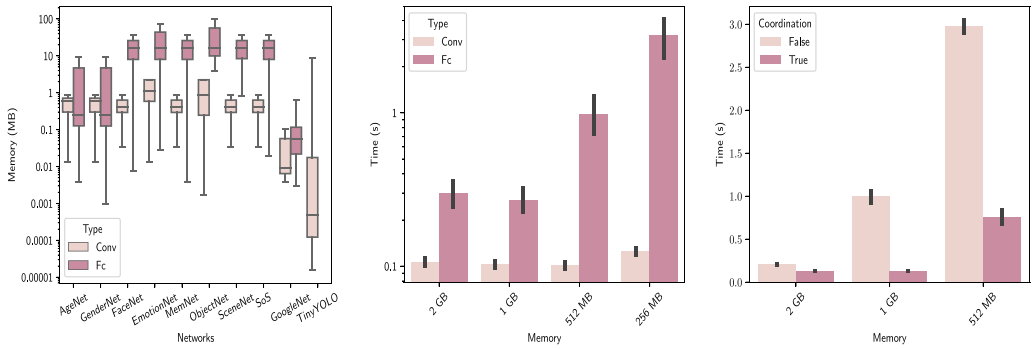
2.1. Memory differs for networks and their layers

The CNNs runtime memory requirements vary greatly. Traditionally DNNs are loaded in their entirety (bulk) before being executed [9,10] favoring throughput over memory usage. Larger architectures with more and bigger layers require more memory.Table 1 shows examples of peak memory demands for different CNNs using Caffe (see Section 5.1 for details) ranging from 183 MB to 892 MB.

To lower peak memory requirements, especially to run large architectures on memory-constrained (edge) devices, we modify Caffe to allow fine-grained control on when layers are loaded and executed. However, memory usage can still significantly differ between layers. Fig. 3(a) summarizes the per-layer peak memory distribution for all CNNs in Table 1. For each network and layer type the box shows the 25th, 50th, and 75th quartiles while the whiskers extend to the rest of the distribution. We skip pooling layers since they have negligible memory costs. For most networks, fully-connected layers have 10× higher memory usage. This is due to the high number of weights, which grows quadratically with the layer sizes, i.e., equal to the product of fully-connected and its input layer sizes. Convolutional layers only define few weights since the filter matrix is shared across all receptive fields. An exception is TinyYOLO using only convolutional layers. A memory-aware scheduler needs to balance the memory usage with preloading layers for consistent performance.

2.2. Memory matters — intra-network

The memory demands of modern CNNs can easily exceed the available memory, especially on resource-constrained edge devices, deteriorating inference responsiveness. Swapping allows OSs to execute programs with peak memory requirements that exceeds the available physical RAM by offloading memory pages to the swap space on disk. When a program accesses a memory page in the swap space a page fault is generated and the program execution is halted until the memory page has been restored to RAM. Since disk is orders of magnitude slower than RAM, programs can incur significant slowdowns based on the frequency of page faults. We demonstrate such an effect by bulk loading SceneNet as an example. Fig. 3(b) shows its mean execution time split per layer type under diminishing available memory. We repeat each experiment 10 times. Error bars report the standard deviation. The peak memory usage of SceneNet is 892 MB. If the available memory is above this value, i.e., 2 GB and 1 GB RAM, execution time remains unaffected. Lower values of



(a) Layer type peak memory distribution. (b) Diminishing available memory. (c) Lack of multi-DNN coordination.

Fig. 3. Memory usage: differences and impact on inference times of single- and multi-DNN.

available memory results in longer execution time as swapping kicks in. The higher the overcommitted memory the more page faults are generated, resulting in a higher execution speed penalty. With 256 MB RAM, the execution time is 7.2× slower compared to 1 GB RAM. From the split layer statistics, one can see that the lower the memory requirements are the lower the slowdown is because the probability of a page fault is lower. With 512 MB RAM, the fully-connected layers are already affected significantly (2.5× slowdown), while the effect on convolution layers is (still) negligible. Overall this underlines how memory-awareness matters for good performance.

2.3. Memory matters - inter-networks

Multi-DNN inference easily exacerbates the detrimental effects of insufficient memory. Each network increases memory usage. Without coordination, this increases the probability of page faults which negatively affect execution time. We exemplify this effect by running two instances of the same network (MemNet) to ease comparison. Each instance is pinned to a separate core, i.e., the instances share memory but no compute resources. We first run the two instances in parallel (without any coordination). Then we run one instance after the other (naive coordination). We repeat each experiment 10 times. Fig. 3(c) compares the mean execution time of one MemNet instance across the two scenarios. Error bars indicate the standard deviation. Each MemNet instance uses 880 MB peak memory. When memory is sufficient, i.e., 2 GB RAM, both cases have similar execution times. The no-coordination case is slightly slower due to contention on other resources, i.e., mainly loading the weights from disk. However with multiple networks and no-coordination memory becomes a bottleneck already at 1 GB RAM. Here the no-coordination case is 7.8× slower. With 512 MB RAM, both cases are degraded but no-coordination is still 3.9× worse. Naive coordination does better but potentially misses out on complementary resource usages. This highlights the increased challenge and need for coordination when doing multi-DNN inference.

3. Constraint programming model

The layer scheduling can be modeled as a combinatorial optimization problem with constraints, i.e. Constraint Satisfaction Problem [12]. Concretely, given the set of layer tasks T we want to find an ordering which minimizes the makespan.

3.1. Formal definition

Let $T = \{t_1, t_2, \ldots, t_n\}$ be the set of n tasks to be scheduled across a set of k workers $W = \{w_1, w_2, \ldots, w_k\}$. As tasks are not fixed to any worker, we need to keep track of task-worker-assignment relation with the variable X. $X_{t,w}$ is 1 if the task t is assigned to the worker w and 0 otherwise. Each task t_i has a completion time $c = s_i + p_i$ given by a task start time s_i plus a task processing time $p_i \geq 0$. The scheduler tries to decide the starting times $S = s_i$ and assignments X of all tasks such that the completion time of the last task is minimized. From this we define formally the optimization problem as:

$$\min_{S,X} \quad \max_{t_i \in T} c_i \tag{1a}$$

$$\text{s.t.} \quad \sum_{w \in W} X_{t_i,w} = 1, \ \forall t_i \in T \tag{1b}$$

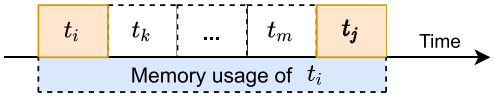


Fig. 4. Example how the memory allocated by a loading task t_i must be retained because the execution task t_j expects it to be available.

$$[s_i, c_i] \cap [s_j, c_j] = \emptyset,\ \forall t_i, t_j \in T : i \neq j, X_{t_i,w} = X_{t_j,w} = 1, w \in W \tag{1c}$$

$$c_i \leq s_j,\ \forall t_i, t_j \in T : \phi(t_j, t_i) = 1 \tag{1d}$$

$$M_{tasks}(\tau_l) + M_{locked}(\tau_l) \leq M_{system}, \forall \tau_l \tag{1e}$$

where:

$$M_{direct}(\tau_l) = \sum_{t_i \in T} \psi(\tau_l, s_i, c_i) m_i^{direct} \tag{2}$$

$$M_{locked}(\tau_l) = \sum_{t_i, t_j \in T : \theta(t_j, t_i) = 1} \psi(\tau_l, s_i, c_j) m_{j,i}^{locked} \tag{3}$$

3.2. Task relation constraints

We do not want to load or execute a layer multiple times. Constraint (1b) ensures that each task is assigned exactly once. An assigned task occupies a worker for a time interval specified by the tuple $[s_i, c_i]$. Since each worker can handle one task at a time, all intervals on the same worker must be non-overlapping. Constraint (1c) ensures this by requiring that the intersection of all intervals belonging to a given worker is the empty set. Tasks can have dependencies on other tasks, i.e. task t_i must end before task t_i starts if task t_j depends on task t_i. We model these dependencies via the dependency relation function $\phi(t_i, t_j)$:

$$\phi(t_j, t_i) = \begin{cases} 1 & \text{if task } t_j \text{ depends on } t_i \\ 0 & \text{otherwise} \end{cases}$$

Constraint (1d) ensures that the schedule respects these dependencies.

3.3. Simple memory model

Tasks execute three types of actions: layer loading, executing, and unloading. Both layer loading and executing result in an increase of memory usage. Loading parameters moves them from disk to memory. Likewise, layer execution stores the intermediate variables and results in memory. On the contrary, layer unloading once completed frees up the memory. We use a simple memory model which assigns to each task t_i a direct memory usage of $m_i^{direct} \geq 0$.

Every task uses memory to execute. We must guarantee that at any point in time the memory usage does not exceed the available memory pool of the system M_{system}. The total memory used at each point in time τ is given by sum of the memory usages m_i of all running tasks t_i at time τ. To know if a task is running at time τ, Eq. (2) uses the indicator function $\psi(\tau, s_i, c_i)$:

$$\psi(\tau, s_i, c_i) = \begin{cases} 1 & s_i \leq \tau \leq c_i \\ 0 & \text{otherwise.} \end{cases}$$

To be able to efficiently solve the problem we further discretize the time as τ_l.

3.4. Locked memory model

Modeling the used memory at time τ_l with only the direct memory underestimates the memory requirements. Tasks which belong to the same layer are highly coupled to each other, as can be seen in Fig. 4. The ordering between loading, execution and unloading tasks related to a given layer exists because the execution task uses the memory allocated by the loading task for the parameters from disk. Hence some tasks use not only their own memory but also parts of the memory allocated by other tasks. $\theta(t_i, t_j)$ captures this relationship:

$$\theta(t_j, t_i) = \begin{cases} 1 & \text{if task } t_j \text{ requires access to memory allocated by task } t_i \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

To incorporate the locked memory model into the our optimization problem we introduce a new interval parameter. Given a task t_j using memory allocated by t_i the locked memory interval is defined by $[s_i, c_j]$, where s_i and c_j are the start and completion time of the interval where the memory shared between tasks t_i and t_j is in use. This memory usage of

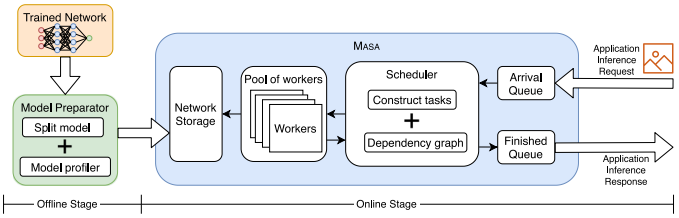


Fig. 5. Architecture of MASA.

such an interval is denoted by the locked memory $m_{j,i}^{locked}$ locked from the ith task to the jth task. Eq. (3) computes the total locked memory M_{locked} as the sum of the locked memory of all intervals active at time τ using again the indicator function ψ.

Constraint (1e) completes the problem definition ensuring that at any time τ_l the total memory given by the sum of M_{direct} and M_{locked} is below the system limit. Using this problem we evaluate the optimality of MASA in Section 5.6.

4. MASA

We design MASA, a memory-aware multi-DNN inference framework on edge devices. MASA can efficiently process jobs composed of images and DNNs and responsively provide requested analyses, e.g., inferring faces and genders. We first describe the architecture, core components, and implementation, then a detailed memory-aware scheduling problem analysis — demonstrating its hardness.

4.1. Architecture overview

MASA is composed of an offline model preparator and an online scheduler as shown in Fig. 5. The offline inputs are the candidate set of trained DNN models and the online inputs are the images and specific networks. Following the observations of 2.1, MASA models each layer of networks as a task in a dependency graph such that those tasks can be scheduled with high flexibility. For instance, when a new inference task of high priority arrives, the conventional execution of the entire networks can either complete the task or abort it — limited flexibility. In contrast, MASA can switch models by temporarily suspending the execution of low priority tasks or run them concurrently due to such a finer-granularity task modeling.

Model Preparator. All trained DNNs are first processed offline by the model preparator, from model downloading, splitting, to profiling. Each DNN model gets split by layer, and their corresponding weight parameters are stored in separate files. We then profile the active memory usage of each layer by offline execution of the DNNs. The details of the profiling steps are provided in Section 4.2.

Scheduler. The scheduler is responsible for constructing two types of tasks for each network layer, namely loading model parameters from disk into memory and execution of such a layer. We abbreviate them as *loading* and *execution* tasks.

The scheduler constructs the dependency graph for tasks of each DNN and then distributes those tasks to free workers based on their dependency graph. Loading tasks have to be strictly executed prior to the execution tasks of the same layers. More importantly, the scheduler dynamically monitors the memory usages and estimated memory requirements for tasks of all DNNs that are ready. The estimated memory requirement is obtained from the profiler. The detailed algorithms of the scheduler are described in Section 4.3. When all tasks from a DNN are completed, the inference results are sent as a response to the application.

4.2. Memory profiler

In order to make memory-aware scheduling decisions, MASA needs accurate estimates of the memory requirements per layer. An underestimation of the memory usage causes the scheduler to over commit its memory budget and thereby likely causing paging and slowdowns. An overestimation of the memory usage can lead to under utilization of precious resources.

The memory profiler aims to precisely estimate the peak memory usage per layer which demands a finer granularity than existing well-known off-the-shelf profilers. We hence resort to query the kernel via the `/proc/self/statm` interface to track the RAM usage of the process. Specifically, we execute each layer of a network in isolation by a single worker during profiling. To track the memory usage from the beginning of a layer task to its end, we spawn a parallel thread which continuously probes the consumed memory. The mean time between probing of the profiler thread is 7.12 ± 5.5 µs, which is significantly lower than the execution time of the smallest layer considered in our evaluation. Table 1 lists the storage space of each network and peak run-time memory usage. One can see that the active memory usage is multiple times higher than the storage space.

Algorithm 1: MASA scheduling algorithm

```
1  Function ValidTask(task):
2      c ← BusyWorkersCount()                          // busy worker count
3      M ← GetFreeMemorySpace()                        // available memory
4      return c == 0 ∨ task.req_memory ≤ M
5  Function ScheduleTask(w, tasks):
6      for task ∈ tasks do
7          M ← GetFreeMemorySpace()                    // available memory
8          if ValidTask(task) then
9              M = M − task.req_memory
10             w.Assign(task)
11             return True
12     end
13     return False
14 Procedure Masa()
15     W ← IdleWorkers()                               // set of idle workers
16     for w ∈ W do
17         exec, load ← ReadyTasks()                   // ready tasks by type
18         if not ScheduleTask(w, exec) then
19             ScheduleTask(w, load)
20     end
21     return
```

4.3. Scheduler

The scheduler sends the loading and execution tasks of all layers of DNNs that are requested for specific images to workers. It ensures that the memory occupancy is within the space limit. The scheduler follows two principles: (i) the task dependency, i.e., loading task of layer j should be completed before starting its execution task, and (ii) a hybrid order of layer type and memory constraint within and across multiple DNNs. As such, MASA incorporates the memory dependency at the inter- and intra-network levels.

The loading and execution tasks of all layers across all specified DNNs are kept sorted into two groups: the waiting and ready group. Tasks start in the waiting group. Once all their dependencies are satisfied they are moved to the ready group. Dependencies are resolved based on the dependencies graphs defined by the multi-DNN job and each DNN model. Tasks in the ready group are sorted to optimize memory consumption in a greedy fashion. Alg. 1 summarizes the scheduler pseudocode. Execution tasks are prioritized over loading tasks since they free up memory upon completion. Tasks are sorted by increasing memory consumption. Function *ReadyTasks()* on line 17 lists tasks in ascending order of memory usage. When one or more workers are available, tasks are pulled from the ready group in order, checking each time if their estimated required memory exceeds the available memory. If not, the task is started via the function *Assign()* on line 10; otherwise the next task is checked. In case none of the ready tasks fits the available memory and all workers are idle, we forgo the memory constraint to allow one task to run to ensure progress (line 4). In this case we start the task with the smallest required memory of the preferred type, i.e., execution before loading.

4.4. Implementation

We altered the Caffe framework to support layer-by-layer loading and execution of DNNs. This enables partial execution of DNNs and optimization of (multi-DNN) inference on edge devices. We term this new framework EDGECAFFE.[1]

Pool of workers. Layer tasks are computed by workers. The scheduler assigns task to free workers from the pool. Once completed the worker returns to the pool.

Scheduler. EDGECAFFE coordinates the work between the workers via the scheduler. The scheduler pulls multi-DNN jobs from the arrival queue, queries the model information from the network storage, builds the necessary layer handling tasks and resolves the dependency graph. This allows to assign tasks such that the correct network layers are present for computation without violating any dependency. Completed jobs are pushed to the finished queue.

Network Storage. Trained DNNs are saved on disk in the Network Storage. A prepared network consists of split model files containing one or a few layers each, as well as a model description. Other components can fetch a (subset of) model from disk through the Network Storage API by providing the model name. **Arrival and Finish Queue**. The arrival queue stores multi-DNN job requests to be processed by the scheduler. Once completed the reference results are pushed to the finish queue ready to be consumed by the application(s).

[1] https://github.com/bacox/edgecaffe.

4.5. Extension to context-aware network execution

Complex inference tasks require multiple networks to extract all the relevant information. These networks often depend on each other. For example an object detection network first detects if there are any humans in an image before a separate network can detect human specific features like age or gender on the detected humans. This creates a cascading relationship between networks within an inference task. With this relationship in mind, it does not make sense to run all the networks on all the inputs all the time. In our example, the age and gender estimation networks can be skipped when no humans are detected. Hence the execution of downstream network(s) is conditioned on the result of the upstream network(s). We term this context-aware execution of networks.

The implementation of Masa in Section 4.4 does not consider any inter-network dependencies other than the fact that all the networks share the same memory resource. In order to support context-aware execution we modify Masa to leverage the domain knowledge of the user to define the inter-network dependencies. The user describes how the networks depend on each other via two additional network properties. The first one specifies the upstream network(s) and the second the conditions that determine the necessity of execution. With this we can for example express that FaceNet depends on TinyYolo and that the output label of TinyYolo needs to have at least one human in order for the execution of FaceNet to be required. Masa uses the defined inter-network dependency to ensure that during scheduling the start of the execution of TinyYolo precedes the start of the execution of FaceNet. Moreover, we added support to abort networks that already started execution but are deemed unnecessary due to the output of an upstream network during run-time. Aborted networks are automatically transferred to Masa 's finished queue. Any downstream networks are aborted as well due to the fact that the necessity of execution is a transitive property.

We consider two policies for context-aware network execution. The first, called CA-wait, naively waits for upstream networks to finish before deciding if execution of any downstream network is necessary. The second, called CA-preempt first starts all upstream networks but allows any downstream network to start in parallel if the resources allow for it. If the outcome of any upstream network causes the execution of a downstream network to be unnecessary, this already running downstream network is aborted. The intuition behind is that this speeds up the total execution in scenarios where there is a high probability of a positive output of upstream networks since we avoid idle waiting of parallel threads. For example with the inter-network relations described in Fig. 9(b), if a dataset has a high number of images with people, CA-preempt is expected to be faster on average because we have to run all the networks with high probability.

5. Evaluation

This section presents our in-depth evaluation of Masa using real-world DNN applications on representative edge devices.

5.1. Experimental setup

Edge Devices. We consider RaspberryPi (termed RPi) as representative edge devices because of its wide adaptability and ease of programming. Specifically, we select three configurations of RPi: (i) RPi 3B+ equipped with Cortex-A53 (1.4 GHz) and 1 GB memory, (ii) RPi 4B with Cortex-A72 (1.5 GHz) and 2 GB memory, and (iii) RPi 4B with Cortex-A72 (1.5 GHz) and 4 GB memory. To emulate the scenario of multiple applications on edge devices, we only consider memory sizes of 512 MB, 1 GB and 2 GB in the following evaluations. Each RPi is equipped with a SanDisk Extreme 64 GB microSD card for storage.

Multi-DNN Jobs. We consider 9 types of DNNs (AgeNet, GenderNet, FaceNet, SoS, GoogleNet, TinyYOLO, EmotionNet, MemNet, and SceneNet) to analyze images from the EDUB-Seg dataset [13,14]. We note that as the models are not altered, e.g., compressed or pruned, the model accuracy is not impacted. Each network conducts various kinds of image analysis, e.g., inferring age, gender, and salient objects, and differs in structure and size. Table 1 summarizes all considered networks listing the disk space and active memory usage obtained from our profiler. EmotionNet is the largest network in terms of absolute storage space, i.e., in the order of 380 MB. SceneNet is the largest network in terms of memory usage, i.e., in the order of 890 MB. Networks listed in Table 1 are commonly used for multi-DNN inference applications like lifelogging [7]. We emulate scenarios of multi-DNN inference by executing multiple of such DNN models on periodic and stochastic image arrivals. The specific composition and number of DNN models are given in each subsection. For periodical case, single images arrive with a fixed interval. For stochastic case, both inter-arrival times and number of DNNs per image follow normal and uniform distribution, respectively.

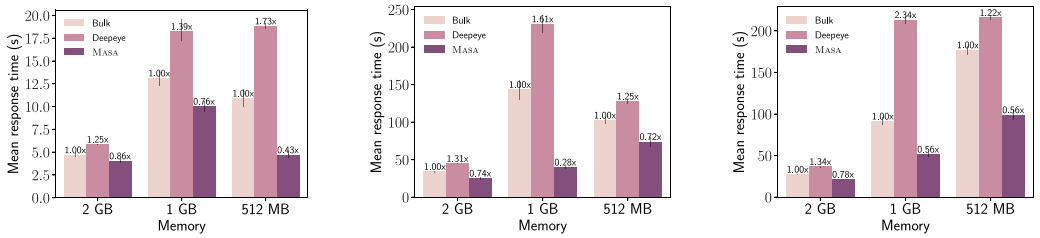
Performance Measures. We aim to optimize average response times for multi-DNN jobs. The response time is measured from the moment the image arrives till the time all DNN models complete execution. For deterministic arrivals, we also set deadlines which are imperative for safety critical systems. All values presented are averaged over 150 images and more than 450 DNN inferences.

Comparison. We compare Masa against `Bulk` and `DeepEye` [7]. `Bulk` executes one DNN at a time by first loading all layers at once and then executing them. `DeepEye` interleaves the execution and loading of convolution and fully-connected layers. Different from Masa, `DeepEye` acts on the granularity of layer type, instead of individual layers and is not aware of their memory usages by loading one type of network layers at once. We implement both approaches on EdgeCaffe and use two worker threads.

Table 1
Overview of used DNNs in the experimental setup.

Network[a]	Inference	Storage space (MB)	Memory usage (MB)	Layers		
				conv	fc	total
AgeNet [4]	Age	44	183	3	3	19
GenderNet [4]	Gender	44	186	3	3	19
FaceNet [15]	Face	217	875	5	3	23
SoS [16]	Saliency	218	875	5	3	21
GoogleNet [16]	Saliency	23	404	22	2	151
TinyYOLO [17]	Object	62	263	9	–	39
EmotionNet [18]	Emotion	378	761	5	3	22
MemNet [19]	Memorability	217	880	5	3	22
SceneNet [20]	Object	221	892	5	3	23

[a]The architecture for SoS, GoogleNet, TinyYOLO, EmotionNet and MemNet are AlexNet, GoogleNet, Darknet, VGG-S and Hybrid-CNN respectively.



(a) Small networks (b) Mixed networks (c) Lifelogging

Fig. 6. Average response times of deterministic image arrivals for `Bulk`, `DeepEye` and Masa.

5.2. Deterministic image analysis

We consider three types of workload scenarios that analyze periodically images: (i) three small DNNs: AgeNet, GenderNet and TinyYOLO, (ii) three mixed DNNs both small and large: AgeNet, EmotionNet, and FaceNet, and (iii) lifelogging scenario where 5 DNNs automatically annotate captured images: TinyYOLO, EmotionNet, MemNet, SceneNet, and SoS. The DNNs used in the lifelogging scenario are typical networks used to give useful annotations to images captured during a lifelogging activity [7]. The images arrive every 20, 200, 200 s for small, mixed and lifelogging scenarios, respectively, and the analysis of multi-DNN needs to be completed before the arrival of the next job.

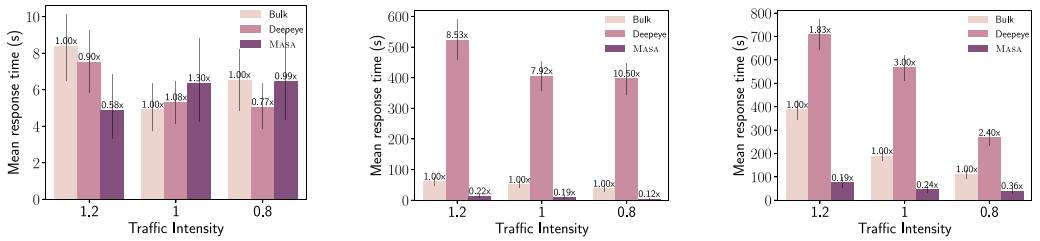
We summarize all three scenarios in Fig. 6. For small and mixed networks, Masa achieves the lowest response time, trailed by `Bulk` and `DeepEye` across all combinations of workload scenarios and RPi configurations, as shown in Figs. 6(a) and 6(b). The performance of `DeepEye` often has the highest response times because multiple DNNs are greedily loaded into memory and cause costly memory swap operations. Meanwhile, due to consideration of intra-network dependency, Masa outperforms `Bulk` by at least 30%, in case of 512 MB memory.

Another observation worth mentioning is the trend of performance gain of Masa. It has significant performance gains on devices with smaller memory, i.e., 512 MB, whereas the performance gain on 2 GB memory is less significant. Moreover, the relative performance gains of Masa against other approaches is the highest on small devices and homogeneous DNNs, compared to large devices and mixed workloads. This can be explained by the balanced task times of execution and loading and avoidance of memory swapping. Fig. 6(c) summarizes the average response time of the lifelogging application that emulates a real-life application, executing 5 DNN inferences for every captured image. Similar trends as for the previous two scenarios can be observed: Masa can effectively allocate the limited worker threads and memory to minimize response time. The average response time of Masa with 1 GB memory is similar to `Bulk` and `DeepEye` running on 2 GB memory. In other words, Masa can achieve almost 50% resource saving without degrading response times.

5.3. Stochastic image analysis

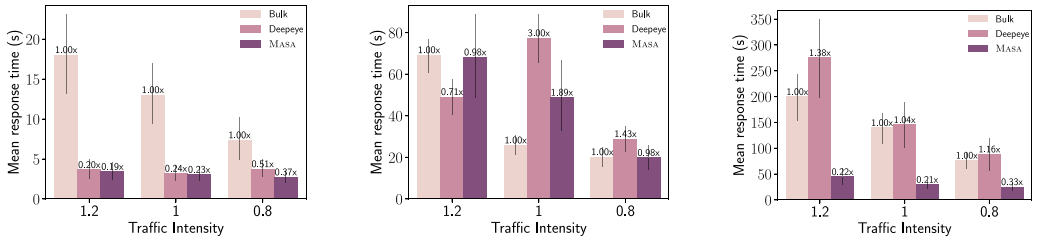
Here, we consider stochastic image analyses, where either the image arrivals or the number of DNN inferences, or both are generated stochastically. This workload scenario is more complex than the periodic cases due to the intricate intra-network dependency and heterogeneity of inference jobs.

Specifically, two types of stochastic scenarios are evaluated with increasing randomness. *Scenario I*: a single DNN inference randomly drawn from the nine listed in Table 1 is requested upon the arrival of images following a normal distribution with mean equal to the sum of the average execution times of the candidate models and standard deviation of 200 ms. Due to the high variance of inter-arrival times, multiple DNNs need to be processed at the same time. *Scenario II*: images arrive periodically but the composition of multiple DNNs is randomly requested following a uniform distribution.



(a) 2 GB memory (b) 1 GB memory (c) 512 MB memory

Fig. 7. Average response times of stochastic scenario I for `Bulk`, `DeepEye` and Masa.



(a) 2 GB memory (b) 1 GB memory (c) 512 MB memory

Fig. 8. Average response times of stochastic scenario II for `Bulk`, `DeepEye` and Masa.

For each scenario, we evaluate Masa on three memory configurations and three traffic intensities. We normalize the traffic intensity with respect to the mean of the execution time of all networks in the scenario, as such values of 0.8, 1.0 and 1.2 are evaluated. The higher the value of traffic intensity becomes, the more challenging it is to achieve low response times.

Figs. 7 and 8 summarize the average response times of the aforementioned stochastic scenarios and the relative performance gain with respect to `Bulk` and `DeepEye`. Similar trends as the deterministic case can be observed: Masa achieves the lowest response times for small memory and high traffic intensity. In terms of absolute values, here the relative gains are significantly higher than for the deterministic scenarios which only need to cope with predictable workloads.

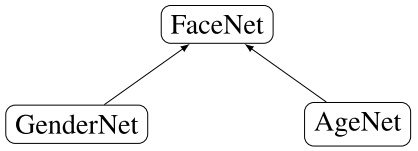
Scenario I. In Fig. 7, one can clearly see that the performance gains of Masa against `Bulk` and `DeepEye` are more pronounced for higher traffic intensity and lower memory. In the cases of 1 GB and 512 MB memory (shown in Figs. 7(b) and 7(c)), we reduce the average response times up to 90%, compared to the second best policy. In the case of 2 GB memory shown in Fig. 7(a), we are slightly worse especially for the lighter traffic intensity. This can be explained by the fact that under light traffic greedy loading algorithms like `Bulk` and `DeepEye` are better to recover from the impact of memory paging than under heavy traffic. Moreover, we would like to point out that the average response times of `Bulk` and `DeepEye` increase from around 6 up to several hundreds seconds when memory space is reduced from 2 GB to 512 MB. In contrast, Masa can show only a 1.5X increase of the response times relative to the memory reduction, i.e., from 5 seconds at 2 GB up to 60 s at 512 MB.

Scenario II. In Fig. 8 we can see that Masa is still the best performing policy, trailed by `DeepEye` then `Bulk`. As this scenario is slightly more predictable than scenario I due to the periodic image arrivals, the performance gain is slightly lower than scenario I. `DeepEye` is able to manage the response time by greedily interleaving the model loading and execution. Here, Masa can maintain close to superlinear ratio between average response times and available memory, i.e., from 3 s at 2 GB up to 30 s at 512 MB at a traffic intensity of 1.

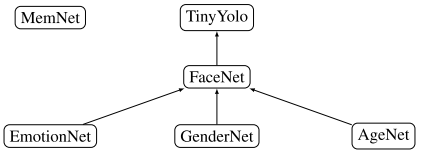
Masa is able to achieve such remarkable results due to its intelligent memory management and interleaved execution of DNN layers. We also evaluate the effectiveness of Masa on different number of worker threads. Masa can robustly ensure low response times, whereas `Bulk` and `DeepEye` cannot properly take advantage of multiple worker threads. Increasing the number of workers to a value larger than 2 allows Masa to better interleave tasks when multiple DNNs are executed concurrently. This flexibility in scaling is not seen in `Bulk` and `DeepEye`. For example, doubling the number of workers from 2 to 4 under a random high traffic intensity and 3 small DNNs, Masa reduces the mean response time by 79% compared to 15% and 9% achieved by `DeepEye` and `Bulk`, respectively. Due to the space limit, we skip the detailed presentation of such results.

5.4. Context-aware image analysis

To evaluate the benefits of context-aware network execution we consider the two inter-network dependency scenarios shown in Fig. 9. The first scenario, termed light (see Fig. 9(a)), uses FaceNet to detect faces and infer subsequently the

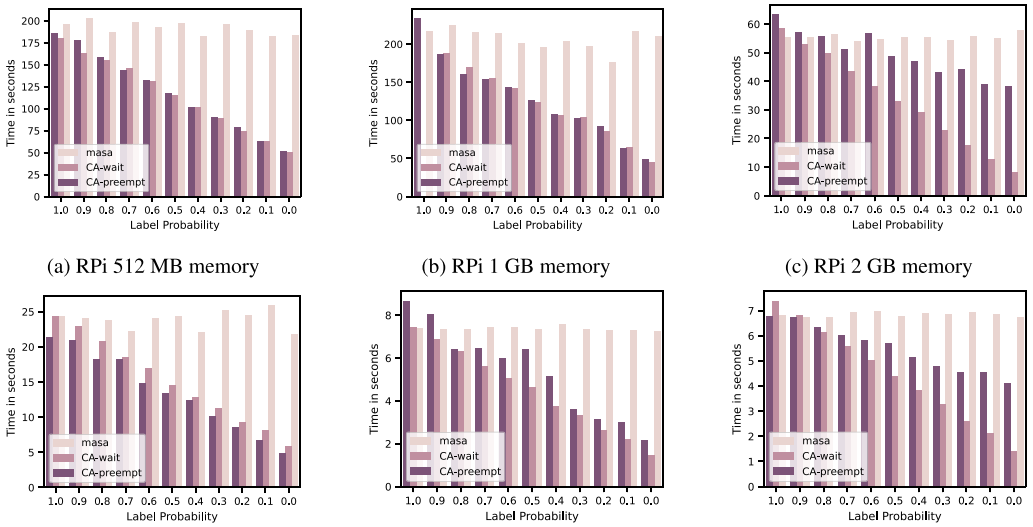


(a) Light workload: EmotionNet, GenderNet, and AgeNet only runs if FaceNet detects a face.



(b) Heavy workload: extends the light workload. FaceNet only runs if TinyYolo detects people. In addition MemNet runs independently.

Fig. 9. DNN relationship diagrams used to evaluate the CA-WAIT and CA-PREEMPT.



(a) RPi 512 MB memory (b) RPi 1 GB memory (c) RPi 2 GB memory

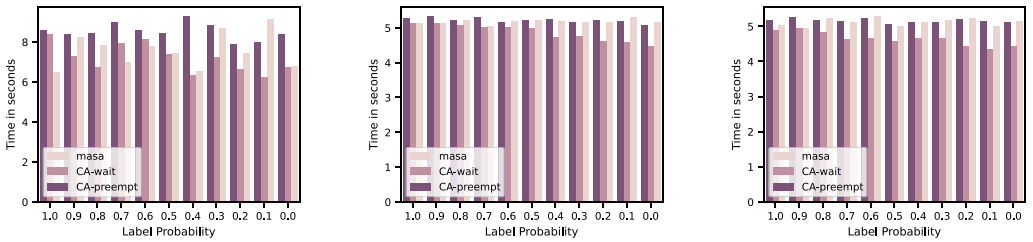
(d) VM 512 MB memory (e) VM 1 GB memory (f) VM 2 GB memory

Fig. 10. Heavy scenario: average makespan under label probabilities varying from 0 (downstream networks never executed) to 1 (downstream networks always executed) and on different platforms (top row: RPi, bottom row: VM) and available memory.

age and gender via AgeNet and GenderNet. Here the execution of AgeNet and GenderNet is dependent on the result of FaceNet. The second scenario, termed heavy, extends the light scenario with three additional networks (see Fig. 9(b)). We condition the execution of FaceNet on the presence of people as detected by TinyYolo and additionally infer emotions from detected faces via EmotionNet. Moreover, MemNet independently assigns a memorability score to each input. Compared to the light scenario, the added downstream network (EmotionNet) and the deeper dependency graph of the heavy scenario lead to a higher difference in computational load between positive and negative outcomes of any upstream network. This is especially true for TinyYolo located at the root of the dependency graph. In addition, MemNet increases the base system load.

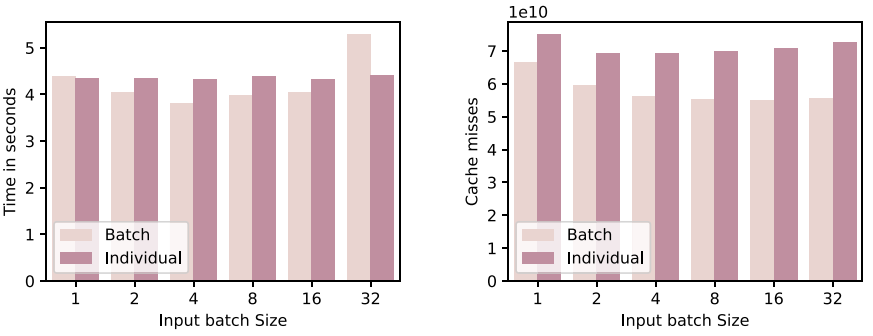
We evaluate both scenarios on the Raspberry Pis and virtual machines with different amounts of memory. To determine the impact of the context-aware execution we fix the label probability of the root network to given values. A label probability of 0.4 means that 40% of the outputs of the root network causes the downstream networks to be executed. Instead, the remaining 60% of the outputs of the root network causes the downstream networks to be skipped or aborted. To warrant the same behavior for both scenarios we ensure that the intermediate FaceNet always provides positive output, i.e. if executed will always trigger the execution of its downstream networks too. We repeat the experiment for different label probabilities and compare the two context-aware execution policies CA-WAIT and CA-PREEMPT against the normal MASA algorithm.

Heavy scenario. Fig. 10 clearly shows that both CA-WAIT and CA-PREEMPT outperform MASA under almost all platform configurations and label probabilities. As expected the gap with respect to MASA becomes larger the lower the label probability is since downstream networks are executed with lower frequency reducing linearly the average makespan. One can further see that CA-PREEMPT falls behind CA-WAIT with lower label probability. We impute this fact to the interference between parallel execution of networks. The early started downstream networks slow down the upstream networks even if they are aborted later on. The effect varies with the available memory. Since we take explicitly into account only the available free memory, with higher amounts of memory CA-PREEMPT allows more networks to be executed concurrently.



(a) VM 512 MB memory (b) VM 1 GB memory (c) VM 2 GB memory

Fig. 11. Light scenario: average makespan under label probabilities varying from 0 (downstream networks never executed) to 1 (downstream networks always executed).



(a) Per input inference latency (b) Cache misses

Fig. 12. Effect of input batching on MASA with different batch sizes and 1 GB memory VM.

Thereby it increasingly slows down the execution of the first network. Finally, similar trends are observed across both RPis or VMs only on different time schedules due to the faster VM hardware.

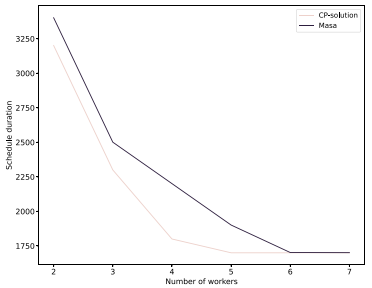
Light scenario. Using a lighter workload the difference between CA-WAIT and MASA becomes less clear. Here the potential gain from not executing the downstream networks is reduced due to the lower height and number of nodes in the dependency tree, and the fact that AgeNet and GenderNet are rather lightweight networks. Hence, context aware execution has no clear advantage over regular execution of MASA. This is clearly shown in Fig. 11. For brevity the RPi results are not shown, but the same observations as for the heavy scenario hold.

5.5. Batching of inputs

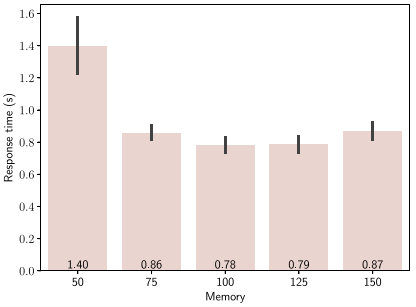
The previous sections considered inference tasks where a single input was given at a time. The execution of multiple input for the same set of networks can be executed as a batch. This allows to pipeline the processing and discount the network loading operation across multiple inputs increasing the processing throughput. However, at the same time it grows the memory footprint of the inference task to store the additional intermediate layer outputs. We compare the results of batched against individual arrivals using a 2-network inference task under the same number of arrivals. For example, with a batch size of 8, the batched task consists of 1 arrivals of 8 inputs (1×8) while the individual task consists of 8 arrivals of 1 input (8×1). MASA is not able to split a network in sub-layer components and thus a network executed with batched inputs will always have a larger memory footprint. Fig. 12(a) shows the average per input inference time. On the one hand batched inputs profit from reusing loaded layers across multiple inputs which reduces the number of cache misses. This can be seen in Fig. 12(b). On the other hand the increasing memory footprint of larger batch sizes obliges MASA to increasingly delay layer tasks until enough free memory is available. Overall, as batch sizes increase, this leads first to better then to worse per input inference times than individual arrivals with batch size 8 giving the best results. With 32 inputs, the more fine grained control over the memory usage given by individual arrivals overcomes any batching benefit.

5.6. Optimality of MASA

We evaluate the optimality of MASA using the multi-network scheduling problem defined in Section 3. Due to the complexity of the optimization problem, we can only find the optimal solution for small scale problems. In particular,

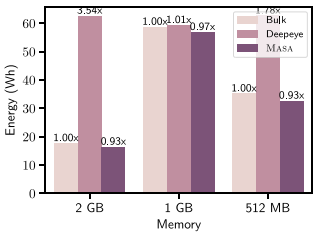


(a) Multi model scheduling of two 4-layers networks. MASA approximates the optimal CP-schedule when the number of workers increases.

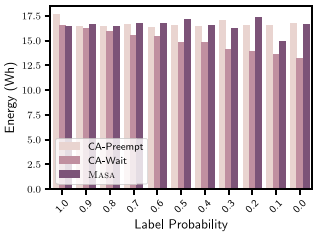


(b) Effect of under-/ over-estimation of required memory for a network. 50%, 75%, 125%, and 150% represent the percentage of memory values from the profiler.

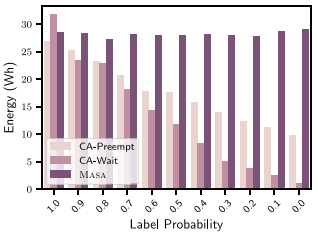
Fig. 13. Optimality (a) and sensitivity to memory estimates (b) of MASA.



(a) [Wh] for three small DNN.



(b) [Wh] for context aware light scen.



(c) [Wh] for context aware heavy scen.

Fig. 14. Energy analysis: MASA, CA-WAIT, and CA-PREEMPT.

we consider herein scheduling two 4-layers networks, i.e. total of 24 tasks, with no inter-network dependencies. We find the optimal solution using the solver from Google OR-Tools and compare against the makespans achieved by MASA for different numbers of workers. Fig. 13(a) shows that MASA is able to quite closely achieve the optimal makespan under all number of workers (worst optimality ratio is 1.22 with 4 workers). When the number of workers increases MASA is even able to equal the optimal solution. More workers simplifies the problem by giving MASA more opportunities to choose the better tasks.

5.7. Sensitivity Analysis of Memory Estimate

Here, we test the robustness of MASA in response to inaccurate memory estimates. To such an end, we multiply the profiled memory requirements of all layers in AgeNet by 0.5 and 0.75 to show under-estimation and by 1.25 and 1.5 to show over-estimation of memory. We run a batch of 6 AgeNet networks with a repetition of 20 image arrivals. We summarize the results in Fig. 13(b). The under-estimated memory values cause the algorithm to load too many networks at the same time while the over-estimated memory values cause the algorithm to under-utilize the available memory. In the case of severely underestimation (the 0.5 multiplier), the average response times increase from 0.8 to 1.4 s as well as the standard deviation. Overall, conservative estimation of memory (even up to 50% higher than actual values) only incur small percentage increases of response times, e.g. 11.5% for 50% overestimation.

5.8. Energy consumption

We conclude our evaluation with an analysis of energy efficiency. Energy efficiency can be critical for battery operated edge devices. We monitor the power consumption of the RPi device. As the on-board tools of the RPi are unfit for accurately measuring power, we use an external power monitor (Joy-IT JT-TC66C) to measure the voltage and current drawn from the power supply and derive the corresponding energy consumption in terms of Wh.

Fig. 14(a) compares MASA against `Bulk` and `DeepEye` for the deterministic three small DNN inference described in Section 5.2 with 150 input images. The energy values are on a par with the measured response times. As a result, the energy consumption of MASA is the lowest. MASA is explicitly designed to increase the responsiveness of DNN inference on edge. This leads to shorter inference times which reduce the energy consumption. Next, we compare MASA against the two types of context-aware execution. Figs. 14(b) and 14(c) show the energy usages for the light and heavy workload scenario described in Section 5.4 under decreasing label probabilities. One can clearly see that CA-WAIT and CA-PREEMPT benefit

Table 2
Overview of prior-art.

Method	Multi-DNN	Stochasticity	No DNN similarity	Memory aware	Architecture	
					GPU/NPU/TPU	CPU
MCDNN [23]	✔	✗	✗	✔	✔	✔
DeepEye [7]	✔	✗	✔	✗	✗	✔
NeuOS [8]	✔	✗	✔	✗	✔	✔
NestDNN [24]	✔	✗	✗	✔	✗	✔
DeepMon [25]	✗	✗	–	✔	✔	✗
PatDNN [26]	✗	✗	–	✔	✔	✔
DeepCache [27]	✗	✗	–	✔	✔	✔
DART [28]	✔	✗	✔	✗	✔	✔
PREMA [29]	✗	✗	✔	✗	✔	✗
Layerweaver [30]	✔	✗	✔	✔	✔	✗
AI-MT [31]	✔	✗	✗	✗	✔	✗
Lee et al. [32]	✔	✗	✗	✔	✔	✔
Wang et al. [33]	✗	✗	✔	✗	✗	✔
Hu et al. [34]	✗	✗	✔	✗	✗	✔
MASA	✔	✔	✔	✔	✗	✔

from the conditional inference execution of dependent DNNs compared to MASA. Moreover, CA-WAIT benefits from its conservative nature compared to CA-PREEMPT, by avoiding wasting energy on preempted computations. Consequently, CA-WAIT outperforms both CA-PREEMPT and MASA when the label probability decreases. The gains are higher in the heavy scenario due to its increased computational inference load. On average for all possible label probabilities, CA-WAIT outperforms MASA by 10% in the light scenario and CA-WAIT and CA-PREEMPT outperform MASA by 54% and 37% respectively in the heavy scenario.

5.9. Limitations

MASA assumes that trained models can be split into independent stages, e.g. DNN layers, which fit the available memory. Models that cannot be split or have layers larger than the available memory, will have a lowered response time performance. Models where the input of a layer depends on multiple preceding layers, e.g. shortcuts in Residual Networks, also increases the peak memory usage of stages which can reduce the effectiveness of MASA. In the worst case, the performance of MASA defaults to the same level as the baseline `Bulk`. However, given the non-uniformity of the memory usage of the layers in a DNNs (Fig. 2), the reduction in effectiveness will mostly affect the execution of the few large layers and not the entirety of the model. A possible solution to reduce the memory footprint of the model stages (sub-layers) can be using adaptive DNN partitioning [21,22]. The offline model preparation currently uses only the model properties ignoring the properties of the target hardware device. While this makes it possible to perform the model preparation in an offline fashion, it can create limitations if the chosen model split is not compatible with the target hardware. Taking the properties of the target hardware into account would give the end-user more flexibility when it comes to deploying pre-trained models to edge devices. In future work this could be incorporated into MASA.

6. Related work

We compare MASA with state-of-the-art DNN inference frameworks under two aspects, memory awareness and multi-DNN focus, summarized in Table 2.

Memory-aware DNN Inference. As the size and complexity of DNN applications have grown, their need for computing resources and memory has increased tremendously. It is particularly vital to manage and control the execution of DNNs on edge devices that have limited RAM. Existing solutions resort to model compression [35,36], quantization. [37], or network pruning [26,35] to reduce the memory demands of DNNs. DeepMon [25] and DeepCache [27] are mobile deep learning inference frameworks which accelerate CNNs execution using cache mechanisms for processing convolutional layers. Jiang et al. [37] use graph optimizations and quantization to increase the performance of CNNs in terms of inference speed. Yang et al. [35] propose a mixed pruning method with a minimal accuracy penalty, which reduces the number of parameters in a DNN by removing redundant layers. PatDNN [26] is a pattern-based DNN pruning framework, including compressed weight storage, register load redundancy elimination, and parameter auto-tuning. Dynamic DNNs are used to adaptively skip a subset of computations to stay within the available memory or time budget while minimizing the degradation of inference accuracy [33,34]. The common theme of aforementioned approaches is to tradeoff the accuracy for the resource efficiency for a single DNN, whereas MASA manages the memory requirements of multiple DNNs without altering the network structure and its resource demands.

Multiple DNNs. Running multiple models on the same device at the same time or sharing resources with other processes is an effective but challenging way to multiplex the limited available resources. Several recent studies [7,8, 24,38,39] propose solutions to efficiently run multi-DNN inference such that energy consumption, response time and

accuracy can be optimized. NeuOS [8] minimizes a three-dimensional space, including latency, accuracy, and energy at the layer granularity for multi-model execution. DeepEye [7] and Layerweaver [30] use interleaving of layers to optimize the execution of multiple DNNs. [38] dynamically determines which DNN should be selected to process a given input by considering the desired accuracy and inference time. NestDNN [24] allows concurrent model execution by scaling down resource demands by using multi-capacity models and surrendering accuracy. MCDNN [23] and AI-MT [31] rely on model sharing of variants with the same base model to efficiently run the same model type with different tasks. DART [28] groups a subset of DNN layers (task-level stage) and assigns them to workers of CPUs and GPUs to minimize task response times by balancing tasks over resources. PREMA [29] enables multi-network execution on Neural Processing Units (NPU) by preempting low priority tasks in favor of high priority tasks. [32] proposes a weights virtualisation scheme that enables efficient packing of multiple DNNs in a single memory space. MemA [40] compares the effect of the execution time of a multi-DNN inference in the context of different scheduling algorithms but has difficulties to adapt to unexpected changes in the available memory space. The aforementioned studies accelerate inference time of multi-DNNs by leveraging the similarity across DNNs to reduce the computation, without being aware of their active memory usages. Different from them, the proposed Masa dynamically schedules DNNs according to the active memory status and does not rely on the similarity of DNN structures.

7. Conclusion

Motivated by the importance and emerging trend of conducting real-time image analyses at edge devices, we design and implement Masa, a responsive memory-aware and context-aware multi-DNN execution middleware. Masa carefully models and manages the run-time memory usage of convolutional and fully-connected layers of all DNNs. As such, it leverages the complimentary CPU/memory usage patterns and dependencies of layers within and across networks to efficiently schedule multiple DNN inferences simultaneously. We evaluate Masa on comprehensive workload scenarios, i.e., combinations of 9 different DNNs on three hardware configurations of Raspberry Pi and VMs. Our results show that Masa can achieve optimality ratios close to one and significantly reduce the average response times of multi-DNN inferences by up to 90%, compared to state-of-art multi-DNN solutions. Masa is particularly effective for challenging scenarios where the available memory is low, e.g., 512 MB to 1 GB memory, and multiple DNN inferences are conducted on stochastically captured images. For future work, we will focus on additionally considering energy efficiency and extend Masa to applications with different and mixed types of DNNs.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is partially supported by the SNSF NRP75 project 167266.

References

[1] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, et al., Machine Learning at Facebook: Understanding Inference at the Edge, in: IEEE HPCA, 2019, pp. 331–344.
[2] M. Valueva, N. Nagornov, P. Lyakhov, G. Valuev, N. Chervyakov, Application of the residue number system to reduce hardware costs of the convolutional neural network implementation, Math. Comput. Simulation 177 (2020) 232–243.
[3] V.A. Sindagi, V.M. Patel, A survey of recent advances in CNN-based single image crowd counting and density estimation, Pattern Recognit. Lett. 107 (2018) 3–16.
[4] G. Levi, T. Hassner, Age and gender classification using convolutional neural networks, in: IEEE CVPR Workshops, 2015, pp. 34–42.
[5] M. Harchol-Balter, Performance Modeling and Design of Computer Systems: Queueing Theory in Action, Cambridge University Press, 2013.
[6] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016.
[7] A. Mathurz, N.D. Lanezy, S. Bhattacharyaz, A. Boranz, C. Forlivesiz, F. Kawsarz, DeepEye: Resource efficient local execution of multiple deep vision models using wearable commodity hardware, in: ACM MobiSys, 2017, pp. 68–81.
[8] S. Bateni, C. Liu, NeuOS: A Latency-Predictable Multi-Dimensional Optimization Framework for DNN-driven Autonomous Systems, in: USENIX ATC, 2020, pp. 371–385.
[9] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al., Pytorch: An imperative style, high-performance deep learning library, in: NIPS, 2019, pp. 8026–8037.
[10] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, T. Darrell, Caffe: Convolutional architecture for fast feature embedding, in: ACM Multimedia, 2014, pp. 675–678.
[11] B. Cox, J. Galjaard, A. Ghiassi, R. Birke, L.Y. Chen, Masa: Responsive Multi-DNN Inference on the Edge, in: IEEE PerCom, 2021, pp. 1–10.
[12] N. Sadeh, M.S. Fox, Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem, Artificial Intelligence 86 (1) (1996) 1–41.
[13] M. Dimiccoli, M. Bolaños, E. Talavera, M. Aghaei, S.G. Nikolov, P. Radeva, Sr-clustering: Semantic regularized clustering for egocentric photo streams segmentation, Comput. Vis. Image Underst. 155 (2017) 55–69.
[14] E. Talavera, M. Dimiccoli, M. Bolanos, M. Aghaei, P. Radeva, R-clustering for egocentric video segmentation, in: Iberian Conference on Pattern Recognition and Image Analysis, Springer, 2015, pp. 327–336.
[15] S.S. Farfade, M.J. Saberian, L.-J. Li, Multi-view face detection using deep convolutional neural networks, in: ACM ICMR, 2015, pp. 643–650.

[16] J. Zhang, S. Ma, M. Sameki, S. Sclaroff, M. Betke, Z. Lin, X. Shen, B. Price, R. Mech, Salient object subitizing, in: IEEE CVPR, 2015, pp. 4045–4054.
[17] J. Redmon, A. Farhadi, YOLO9000: better, faster, stronger, in: CVPR, 2017, pp. 7263–7271.
[18] G. Levi, T. Hassner, Emotion Recognition in the Wild via Convolutional Neural Networks and Mapped Binary Patterns, in: ACM ICMI, 2015, pp. 503–510.
[19] A. Khosla, A.S. Raju, A. Torralba, A. Oliva, Understanding and predicting image memorability at a large scale, in: IEEE ICCV, 2015, pp. 2390–2398.
[20] B. Zhou, A. Lapedriza, J. Xiao, A. Torralba, A. Oliva, Learning deep features for scene recognition using places database, in: NIPS, 2014, pp. 487–495.
[21] C. Hu, W. Bao, D. Wang, F. Liu, Dynamic adaptive DNN surgery for inference acceleration on the edge, in: IEEE INFOCOM 2019-IEEE Conference on Computer Communications, IEEE, 2019, pp. 1423–1431.
[22] W. He, S. Guo, S. Guo, X. Qiu, F. Qi, Joint DNN partition deployment and resource allocation for delay-sensitive deep learning inference in IoT, IEEE Internet Things J. 7 (10) (2020) 9241–9254.
[23] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, A. Krishnamurthy, MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing Under Resource Constraints, in: ACM MobiSys, 2016, pp. 123–136.
[24] B. Fang, X. Zeng, M. Zhang, NestDNN: Resource-aware multi-tenant on-device deep learning for continuous mobile vision, in: ACM MobiCom, 2018, pp. 115–127.
[25] L.N. Huynh, Y. Lee, R.K. Balan, Deepmon: Mobile gpu-based deep learning framework for continuous vision applications, in: ACM MobiSys, 2017, pp. 82–95.
[26] W. Niu, X. Ma, S. Lin, S. Wang, X. Qian, X. Lin, Y. Wang, B. Ren, PatDNN: Achieving real-time DNN execution on mobile devices with pattern-based weight pruning, in: ACM ASPLOS, 2020, pp. 907–922.
[27] M. Xu, M. Zhu, Y. Liu, F.X. Lin, X. Liu, DeepCache: Principled cache for mobile deep vision, in: ACM MobiCom, 2018, pp. 129–144.
[28] Y. Xiang, H. Kim, Pipelined Data-Parallel CPU/GPU Scheduling for Multi-DNN Real-Time Inference, in: IEEE RTSS, 2019, pp. 392–405.
[29] Y. Choi, M. Rhu, Prema: A predictive multi-task scheduling algorithm for preemptible neural processing units, in: IEEE HPCA, 2020, pp. 220–233.
[30] Y.H. Oh, S. Kim, Y. Jin, S. Son, J. Bae, J. Lee, Y. Park, D.U. Kim, T.J. Ham, J.W. Lee, Layerweaver: Maximizing Resource Utilization of Neural Processing Units via Layer-Wise Scheduling, in: IEEE HPCA, 2021, pp. 584–597.
[31] E. Baek, D. Kwon, J. Kim, A multi-neural network acceleration architecture, in: ACM/IEEE ISCA, 2020, pp. 940–953.
[32] S. Lee, S. Nirjon, Fast and scalable in-memory deep multitask learning via neural weight virtualization, in: ACM MobiSys, 2020, pp. 175–190.
[33] Y. Wang, J. Shen, T.-K. Hu, P. Xu, T. Nguyen, R. Baraniuk, Z. Wang, Y. Lin, Dual dynamic inference: Enabling more efficient, adaptive, and controllable deep inference, IEEE J. Sel. Top. Signal Process. 14 (4) (2020) 623–633.
[34] T. Hu, T. Chen, H. Wang, Z. Wang, Triple Wins: Boosting Accuracy, Robustness and Efficiency Together by Enabling Input-Adaptive Inference, in: ICLR, 2020.
[35] W. Yang, L. Jin, S. Wang, Z. Cu, X. Chen, L. Chen, Thinning of convolutional neural network with mixed pruning, IET Image Process. 13 (5) (2019) 779–784.
[36] D.C. Mocanu, E. Mocanu, P. Stone, P.H. Nguyen, M. Gibescu, A. Liotta, Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science, Nature Commun. 9 (1) (2018).
[37] Z. Jiang, T. Chen, M. Li, Efficient deep learning inference on edge devices, in: ACM SysML, 2018.
[38] V.S. Marco, B. Taylor, Z. Wang, Y. Elkhatib, Optimizing deep learning inference on embedded systems through adaptive model selection, ACM Trans. Embed. Comput. Syst. 19 (1) (2020) 1–28.
[39] M. LeMay, S. Li, T. Guo, Perseus: Characterizing performance and cost of multi-tenant serving for cnn models, in: IEEE IC2E, 2020, pp. 66–72.
[40] J. Galjaard, B. Cox, A. Ghiassi, L.Y. Chen, R. Birke, MEMA: Fast Inference of Multiple Deep Models, in: IEEE PerCom Workshops, 2021, pp. 281–286.