Adaptive Scheduling for Edge-Assisted DNN Serving

Jian He
UT-Austin
Austin, USA
jianhe@cs.utexas.edu

Chenxi Yang
UT-Austin
Austin, USA
cxyang@cs.utexas.edu

Zhaoyuan He
UT-Austin
Austin, USA
zyhe@cs.utexas.edu

Ghufran Baig
UT-Austin
Austin, USA
ghufran@cs.utexas.edu

Lili Qiu
UT-Austin
Austin, USA
lili@cs.utexas.edu

Abstract—**Deep neural networks (DNNs) have been widely used in various video analytic tasks. These tasks demand real-time responses. Due to the limited processing power on mobile devices, a common way to support such real-time analytics is to offload the processing to an edge server. This paper examines how to speed up the edge server DNN processing for multiple clients. In particular, we observe batching multiple DNN requests significantly speeds up the processing time. Based on this observation, we first design a novel scheduling algorithm to exploit the batching benefits of all requests that run the same DNN. This is compelling since there are only a handful of DNNs and many requests tend to use the same DNN. Our algorithms are general and can support different objectives, such as minimizing the completion time or maximizing the on-time ratio. We then extend our algorithm to handle requests that use different DNNs with or without shared layers. Finally, we develop a collaborative approach to further improve performance by adaptively processing some of the requests or portions of the requests locally at the clients. This is especially useful when the network and/or server is congested. Our implementation shows the effectiveness of our approach under different request distributions (*e.g.*, Poisson, Pareto, and Constant inter-arrivals).**

Index Terms—**deep neural networks (DNNs), real-time video analytics, batching DNN requests, scheduling algorithm**

I. Introduction

Motivation: Deep neural networks (DNNs) are widely used in many applications, including autonomous driving, cognitive assistance, video surveillance and AR/VR. Many applications demand real-time inference. Existing works speed up DNN inference in two ways. One way is to train simpler models [26] to reduce computation overhead or use compressed DNNs (*e.g.*, [15], [37]). While significant progress has been made in this front (*e.g.*, MobileNet [20], [53] and ShuffleNet [40], [66]), depending on the model complexity and learning tasks [5], [59], [64], it is not always feasible to achieve real-time guarantees on the mobile devices. A complementary way for the mobile systems is to offload expensive DNN execution to edge servers (*e.g.*, [50], [35]). Edge server is often preferred due to lower network latency. Collaborative DNN [32], [21] leverages the client's computation resources to further reduce processing time. How to efficiently serve many requests on a server and support collaborative DNN execution poses an interesting system challenge, especially for edge servers with limited memory and computational resources. Thus, we explore how to efficiently serve many clients on an edge server. This capability has significant implication on the viability of many mobile applications, including autonomous driving, smart homes, surveillance, as it is common for multiple mobile devices to share one edge server to run DNNs for their analytic tasks.
Opportunities and challenges: A natural way to serve multiple inference requests (*e.g.*, different cameras in the setting of video surveillance or autonomous driving) is to run requests in FIFO. Batching these requests together significantly enhances efficiency due to coalesced memory access (*e.g.*, each weight in a DNN is loaded to the cache only once and used for all input data rather than loaded for each input data separately [28], [22]). Our measurement on a server with an Nvidia Tesla P100 GPU shows it takes around 24 ms for GoogleNet [56] to process one request, and 132 ms to process 10 requests one by one. In comparison, it takes only 28 ms to process 10 requests as a batch, which is only slightly higher than running one request.

It is necessary to have a high enough request rate to create a batch. This is common since video analytic tasks require high frame rate. For example, video surveillance requires 15 frames/second (FPS) [19] on each camera and many cameras are deployed across the environment. A typical experimental automated-driving vehicle consists of ten or more cameras monitoring different fields of view, orientations, and near/far ranges. Each camera generates images at 10-40 FPS depending on its function [63]. Moreover, an edge server can serve requests from different users, organizations, or vehicles, which further increases the request rates. These requests go through one or more DNNs, and some of these DNNs can be shared.

Batching benefit has been widely recognized, but designing schedulers to exploit these benefits in DNN job processing is underexplored. Batching in DNN processing is different from general job batching because the latter batches either entire jobs or nothing, while DNN jobs go through well-defined layers and batching can take place in multiple layers (*i.e.*, partial batching). Scheduling DNN jobs can make a big difference on the batching opportunities, hence the performance.

Designing batch-aware scheduling algorithms is challenging. Simply batching all or a fixed number of jobs is not always desirable since it requires earlier requests to wait to form a large batch and the batching benefit may vary significantly with the job arrival rate and the type of DNN layers. This motivates us to develop scheduling algorithms to support various objectives.
Our approach: We develop a novel scheduling algorithm for requests using the same DNN as the same video analytics task typically uses the same DNN. For example, image classification uses GoogleNet [56], object detection uses SSD [38], and image segmentation uses FCN [39]. Initially, without memory constraints, we develop a dynamic programming algorithm to minimize completion time for a given set of requests. Then, considering memory constraints that limit batch sizes, we incorporate request deadlines and maximize the on-time ratio of jobs while leveraging batching benefits.

We further generalize our algorithm to handle requests that do not run the same DNN. In particular, we consider two scenarios: (i) requests that use different DNNs without shared layers and (ii) requests that go through multiple DNNs and some of these DNNs are shared. Both scenarios are common. For example, some clients may run image classification using ResNet [17] while other clients may run image segmentation using FCN [39]. This falls into (i). A common scenario for (ii) arises when requests go through multiple DNNs and some of these DNNs are shared. For example, video prediction and segmentation (*e.g.*, SDCNet [51] and RTA [24]) share the same optical flow DNN model but use different DNNs for the remaining processing. In this case, the requests at the optical flow DNN can be batched together. Similarly, human pose estimation usually consists of multi-person detection using a shared model (*e.g.*, Faster RCNN [52]) and predicting each person's pose using different models (*e.g.*, IEF [4] and G-RMI [47]). Therefore, the requests at the object detection DNN can be batched together. We design scheduling algorithms for both (i) and (ii). We extend our scheduling algorithm to support DNNs with different numbers of shared layers.

Finally, we consider that a client can process some requests to further reduce the request completion time. We develop two offloading algorithms that consider network delay along with server and client processing time to adaptively determine whether to offload and how much to offload. Our client-side optimization is inspired by [32] but differs in that we consider the server's batching benefit to maximize the efficiency.

We implement our approach on an edge server with an Nvidia Tesla P100 GPU and 16GB GPU memory. We implement a client on the Nvidia Jetson Nano with a 128-core Maxwell GPU and 4 GB memory, which has been widely used as a client platform (*e.g.*, [23], [14]). Video frame transmissions are generated using WiFi and LTE packet traces.

Our main contributions are as follows:

- We design a batching-aware DNN scheduling algorithm to efficiently serve requests for same DNNs, which is flexible to different objectives (*e.g.*, min completion time, max on-time ratio). Our schemes significantly reduce the completion time and improve the system capacity (*i.e.*, max number of concurrent serving requests) by 20%-400% over baselines when serving a single DNN. When maximizing the on-time ratio, our scheme improves the system capacity by $> 22\%$ over the Earliest Deadline First (EDF) with batching strategy.
- We extend our algorithm to support multiple DNNs with different numbers of shared layers. Our scheme improves the system capacity by more than 200% over baselines.
- We enable collaborative DNN execution at the client side to speed up processing. The client can process some requests locally to reduce the server load. Collaborative execution further improves the system capacity by $> 67\%$ over our optimized server only strategy. We implement our approaches on commodity hardware to demonstrate effectiveness.

II. Related Work

Speeding up client-side DNN execution: One way to speed up DNN inference is to train simpler DNNs (*e.g.*, MobileNet [20], [53], SqueezeNet [26], ShuffleNet [40], [66]) or compress DNNs [34], [61]. Despite significant advances, several important learning tasks cannot run on mobile devices in real time (*e.g.*, semantic segmentation, activity recognition, super-resolution video reconstruction). DeepMon [25] and DeepCache [62] reuse pre-cached intermediate DNN output to avoid redundant computation for same input. NestDNN [10] adaptively prunes filters from convolutional layers to reduce the computation demand when the available resource is insufficient. These approaches speed up DNN execution at the cost of degrading accuracy. Deepeye [41] and uLayer [33] speed up DNN execution by using both CPU and GPU. We mainly focus on exploiting batching benefits in GPU.

Adjusting DNN configurations: The computational cost of DNNs depends on the input data size (*e.g.*, the resolution and frame rate of the input video for video analytics tasks). Existing approaches, such as Deepdecision [50], DARE [36], Chameleon [30]), optimize latency by adaptively adjusting the video resolution and frame rate according to the network condition and server workload. These approaches speed up the processing but reduce the accuracy. It also needs an accurate model that captures the relationship between the accuracy and computational cost, which is challenging.

Collaborative DNN execution: Collaborative processing leverages the client's computation resources to further reduce processing time. Instead of offloading all computation, several works (*e.g.*, [32], [21]) partition DNN processing between the client and server. Partitioning has to be done carefully since intermediate results tend to be larger than the input data. Our work complements the existing work by developing batching-aware collaborative processing to exploit batching benefits.

Batching user requests: [2], [22], [54], [31], [6], [13] batch DNN inferences to improve completion time. Nexus [54] develops a new bin packing based algorithm to batch entire jobs Our work advances state-of-the-art as follows: (i) it introduces layer-wise batching-aware scheduling to create more batching opportunities in a single DNN, (ii) it enables batching for multiple DNNs with shared layers, and (iii) it optimizes different performance objectives (*e.g.*, latency and on-time ratio). In Sec. V, we show our system reduces the completion time by 25-32% over Nexus when running SDCNet and RTA. We explicitly optimize completion time or the number of tardy jobs for layer-wise batch-aware scheduling whereas LazyBatching [6] blindly batches requests as long as they do not violate SLA, however batching may not always improve performance and should be used strategically. Clockwork [13] does not consider layer-wise batching, which results in a much lower batching opportunity than ours. Mainstream [29] and MCDNN [16] re-train the existing DNNs so that they have some common layers. Requests can be batched at those common layers to reduce processing delay. We do not modify the DNNs and use scheduling to increase the batching opportunities.

Scheduling: There are several dynamic programming based scheduling algorithms (*e.g.*, [3], [49]). Different from the existing works, which treat each request as a single process, we develop scheduling algorithms for DNNs where each request involves multiple layers and our scheduler determines the order of requests and their corresponding layers to process. [57], [8], [11] batch only requests arriving at the same time and improve resource utilization across multiple GPUs. Our work supports more fine-grained layer-wise batching. Least Laxity [65] assigns a priority to a job based on its running time, but the job's actual running time may be different if batched with other jobs. In comparison, we use the running time under batching for scheduling, which is more accurate.

III. Our Approach

In this section, we first formulate the scheduling problem to minimize the completion time of a single DNN and present our dynamic programming algorithm (Sec. III-A). We then consider how to maximize the job on-time ratio (Sec. III-B). We further generalize to multiple DNNs with or without shared layers (Sec. III-C). Finally, we extend it to support collaborative DNN execution, where clients can process portions of the DNN requests locally (Sec. III-D).

A. Completion Time of One DNN

We develop a scheduling algorithm to optimize the completion time of a given set of requests. When a batch of requests finishes running a layer and new requests arrive, we re-compute the schedule for the updated set of requests. A unique aspect of batching-aware DNN scheduling is that requests are processed according to the DNN layer structure and can be batched with other requests only at the same layer.

1) Problem Formulation

Let N denote the number of layers in a DNN, R denote the set of requests in the system, a_i denote the i-th request's arrival time, c_i denote the i-th request's completion time. Any active request stays in GPU memory till it completes. Our goal is to minimize the total completion time.

Let l_i denote the layer at which the request i currently resides. $l_i = 1$ indicates the request i is waiting to run the first layer. Let $h_k(b)$ denote the running time of layer k for a batch size b. Let B denote the upper bound of the batch size at a layer due to limited GPU memory. We assume $h_k(b)$ has the following property: $h_k(b_1 + b_2) \leq h_k(b_1) + h_k(b_2)$ when $b_1 + b_2 \leq B$. $h_k(b) = +\infty$ if the batch size $b > B$. In our system with 16 GB GPU memory, $B = 90$ is sufficient for even the largest model: SSD. We estimate B by pre-allocating memory based on the maximum number of requests across all DNN layers. This estimation is conservative since different layers may have different number of requests. Dynamically adjusting memory bound for different layers can reduce memory requirement with considerable overhead due to frequent memory reallocation across layers. So we leave it to the future work.

2) Scheduling Algorithm

Unbounded Batch Size: We derive the following property for the optimal schedule using the FIFO scheduling (*i.e.*, the job arriving first finishes first though not necessarily first served due to batching). For example, we can serve the second arriving request and then batch with the first request so both requests finish together.

Lemma 1: For a set of requests R without the batch size bound, if we schedule a request, it will run till completion before running another request that arrives later assuming FIFO.

Proof sketch: If we schedule a request, it will be better to run this request till completion before running others. Switching to running other requests before finishing processing the ongoing requests that have already started causes all requests to have a longer processing time. Refer to [1] for our proof.

Based on the above Lemma, we have the following policy when there is no limit on the batch size. If we schedule a request to run, it will batch all the requests that arrive earlier. This policy has several advantages. First, it ensures earlier requests will finish no later than later requests, which improves fairness and avoids starvation. Second, we can develop a dynamic programming algorithm to minimize the completion time. The algorithm picks a few splitting points, divides a request sequence into segments based on those points, and runs the requests segment by segment. Note that the segment means a sequence of requests batched together before running till the end. The segments run in the order of their arrival time (*i.e.*, the segment involving the requests that arrive earlier is executed earlier). Within each segment, the latest requests are processed first and batched together with the earlier requests when they meet at the same layer. Each segment runs without stopping in the middle according to the Lemma 1.

For example, as shown in Figure 1, there are requests A, B, C, D, E, in the system. We split them into two segments: segment 1 involving requests A and B, and segment 2 involving requests C, D, and E. We first run segment 2, where request C runs from layer l_3 to layer l_4 alone, then batched with request D and runs till layer l_5, where it is batched further with request E and runs till the end. Then we run segment 1, where request A runs alone from layer l_1 to layer l_2, and is batched together with request B and runs till the end.

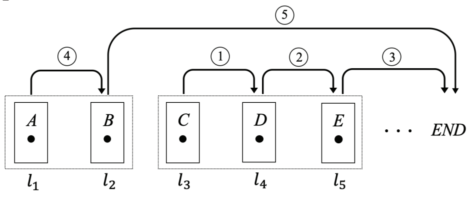


Fig. 1. An example of request segments.

Dynamic programming: As we can see from the above example, how to split requests into multiple segments has significant impact on the performance. Our dynamic programming algorithm selects the splitting points to minimize the completion time. We sort the requests in an increasing order of their arrival time such that the arrival time of requests j and i, denoted as a_j and a_i respectively, satisfies $a_j \leq a_i$ for any $j \leq i$. Since we process requests in the FIFO manner, we have $l_j \geq l_i$. Let $min\_cost(i)$ denote the minimum cost of running all requests from i to 1 under all possible ways of splitting, and $cost(i, j)$ denote the cost of running request i and batched with all requests from i to j along the way till completion without splitting. $cost(i, j)$ can be computed by summing up the running time of each layer starting from layer

l_i weighted by $active(j)$, the number of active jobs before request j finishes including those that finish together with request j. That is,

$$cost(i,j) = active(j) \times \sum_{k=l_i..N} h_k(count(i,l_i)) \qquad (1)$$

where l_i is the layer at which request i resides, and $count(i,l)$ denotes the number of requests that the request i is batched together till it reaches the layer l including the request i and the existing requests at the layer l. The reason behind Equation 1 is that all jobs finishing together with request j need to wait till they all finish. This includes running time from layer l_i till layer N, where the cost of running a layer depends on the number of requests at the layer. $min\_cost(i)$ can be recursively computed as follows.

$$min\_cost(i) = min_{j=1..i}(min\_cost(j-1) + cost(i,j)) \qquad (2)$$

Intuitively, to compute the minimum cost of running requests i through 1 till completion, we search for the best splitting point j where the requests from i to j run together in one segment and incurs $cost(i,j)$, and the cost of running requests from 1 to $j-1$ is computed recursively by considering all possible ways of splitting.

Based on Equation 2, we implement a dynamic programming algorithm that computes a table of size $|R|$, where the i-th entry stores $min\_cost(i)$. We sort all requests in terms of their arrival, where request 1 arrives the earliest. We initialize $min\_cost(0) = 0$ and set $min\_cost(1)$ to the cost of running request 1 by itself till completion. Then we add $min\_cost(2)$, which considers running the request 1 by itself and then the request 2 versus batching the request 2 with the request 1. Similarly, we compute $min\_cost(3)$, which is the minimum of running the first two requests using all possible splittings and then running request 3 by itself, running the first request by itself and batching the requests 2 and 3, and batching the requests 1, 2, 3. As we can see, computing one table entry incurs $O(|R|)$ cost and there are $O(|R|)$ entries. So the overall time complexity is $O(|R|^2)$.

Theorem 1: Our dynamic programming algorithm minimizes completion time for requests using the same DNN among all FIFO schedules when there is no memory bound.

Proof: The property of the optimal FIFO schedule proved in Lemma 1 indicates the request sequence is divided into segments and the segment having the earliest arriving request will run first. The remaining problem is how to find the splitting points for the segments. Our recursive search enumerates all splitting points for FIFO schedules. Hence it yields the lowest completion time among all FIFO schedules. Our evaluation *considers* memory constraints, and our scheme may not be the optimal in this case but still performs well.

Speedup computation: The complexity of our dynamic programming is $O(|R|^2)$. When the number of requests is large, the dynamic programming incurs substantial delay. To further speed it up, we treat all requests at the same layer as one unit: batching all or no requests from a given layer. This reduces the complexity to $O(N^2)$, where N is the number of layers. In practice, only the layers that currently have requests matter, which is even smaller.

We further speed up by clustering layers into fewer groups. Let G denote the number of groups. This will reduce the complexity to $O(G^2)$. In our evaluation, we divide the layers into 5 groups, where each group has close to 1/5 running time of the entire DNN. Our results show that clustering layers speeds up our dynamic programming algorithm with only small impact on the performance (*e.g.*, clustering layers reduces running time from 13ms to 2ms when computing schedule for 500 requests while yielding similar performance).

Incremental update: The schedule is subject to change upon finishing processing one or more requests at a layer and the arrival of a new request. In this case, we run our scheduling algorithm on CPU in parallel to processing the DNN requests on GPU. If the earlier request that changes the layer moves from layer l_1 to layer l_2 since the last schedule update, we reuse the table entries after layer l_2 at the time of last computation and re-compute the remaining entries as well as adding a new entry $min\_cost(i)$ for the new request.

To ensure real-time computation of schedule, we consider up to the first 500 active requests in the system. These requests should also be the ones that occupy the last several layers. The remaining requests will be considered in the future round. The intuition is that the requests arriving late cannot be scheduled immediately since the system already has many requests to process. We can delay calculating the schedule for these requests till the system is close to run them. Scheduling 500 requests takes less than 2ms.

Bounded Batch Size: When the maximum batch size is bounded by B, we still apply the above dynamic algorithm to find a schedule. The only modification is to set $cost(i)$ to $+\infty$ if the number of requests between the request $i \in R$ and $j \in R$ is larger than the bound B and we stop searching the splitting points whose cost are already $+\infty$.

B. Maximize On-Time Ratio of One DNN

So far, we focus on minimizing the completion time. Next we explore minimizing the number of jobs that miss their deadlines. (*i.e.*, tardy jobs). We develop two algorithms.

Our first algorithm modifies EDF, which optimally minimizes tardy jobs on preemptive uniprocessors without batching. To harness batching benefits while satisfying the deadline, we develop the following modified EDF. We sort all jobs by their deadlines and pick the job from the head of the sorted list, and add it to the current batch if all scheduled jobs still satisfy their deadlines. If not, we will go to the next job and add it to the batch if the deadline of all jobs in the batch is honored, and iterate till the end of the list. With this, we honor the deadlines as much as we can while opportunistically batching more jobs.

The modified EDF is not optimal with batching, so we develop a dynamic programming algorithm in Section III with two modifications: (i) we change the goal to minimize # of tardy jobs, and (ii) we drop the jobs that missed their deadline. The output schedule gives the estimated completion time of each ongoing job, and we drop jobs that cannot meet deadlines. While Lemma 1 no longer holds when minimizing the number of tardy jobs, the algorithm significantly outperforms the above modified EDF in our evaluation since it explicitly considers the impact of batching if the jobs satisfy the deadline.

C. Schedule Multiple DNNs

Multiple DNNs without shared layers: We schedule multiple non-shared layer DNNs across M DNNs using dynamic programming. Then we enumerate and select the permutations of DNNs the one that yields the smallest completion time. Enumerating all permutations of DNNs is affordable since the number of commonly used DNNs (M) is small. Running a DNN means running all requests of that DNN till completion. In the DNN permutations, we only consider model-wise permutations – we run all requests of a DNN before we start scheduling requests for another DNN. Since we re-compute the optimal schedule whenever a batch of requests finishes running a layer and new requests arrive, the optimized permutation may change over time to take into account the new requests. The scheduling algorithm outputs the order of the requests to serve until existing requests execute a layer and a new request arrives, in which case the schedule is re-computed based on the latest input. Therefore, requests from different DNNs can be served in an interleaved manner.

Multiple DNNs with shared layers: Next we consider requests that go through multiple DNNs and some of them can be shared. For example, the video prediction and segmentation tasks both use FlowNet2 [27] to compute the inter-frame optical flow and then use SDCNet [51] and RTA [24], respectively, for the remaining processing. In this case, requests for these two different tasks can be batched at FlowNet2.

The schedules for individual DNN are computed with a similar strategy. When computing the completion time, we ensure all requests belonging to the m-th DNN run till completion and they will be batched with any requests arriving earlier (including those belonging to other DNNs at the shared layers) up to the bound B to maximize the batching benefit. When multiple DNNs are loaded to the GPU memory, B is set to the total GPU memory used by all DNNs. We then derive the completion time for different orders of running DNNs and select the permutation with the lowest completion time.

Incremental update: The schedule is subject to change upon (i) a batch of requests finishing running a layer and the arrival of a new request, or (ii) a request moving across the boundary between shared and non-shared layers. Whenever any such event occurs, we re-run our scheduling algorithm on CPU in parallel to DNN execution on GPU. As before, we reuse the previous table as much as possible. We first find the earliest request that changes the layer since the last schedule update. Say the request moves from layer i_1 to layer i_2. We re-use the table entries after layer l_2 at the time of last schedule update. Reusing table entries allows a quick update of decision (*e.g.*, within 2ms for 500 requests).

D. Collaborative DNN Execution

The GPUs on mobile devices are generally less powerful than those at the server. Mobiles also adjust the GPU speed to save power. For example, the Nvidia Jetson Nano device can run VGG16 at 4fps and 11fps when it is at 5W and 20W mode. Despite slower processing speed than the server, it can be beneficial for the mobile devices to process some requests locally when the server is overloaded. We develop two collaborative DNN execution strategies: (i) binary offloading (*i.e.*, a client either processes or offloads an entire request), and (ii) partial offloading (*i.e.*, a client processes the first few layers and offload the rest to the server).

Binary offloading: The mobile device can process the request locally if it the device can finish the request within the deadline. Otherwise, it compares the local vs. remote processing time (including network delay) and picks a lower one so that the job can finish faster. Local processing time can be simply estimated using measurement. Our evaluation uses the average running time of the DNN across 100 runs as the estimate. Remote processing time is the sum of the network delay and server processing time, where the network delay is estimated based on the transmission size and network throughput using exponential weighted moving average (EWMA) with a weight of 0.3 on a new delay sample and the server processing time is determined using the above dynamic programming algorithm.

Partial offloading: To improve efficiency, we allow clients to process their requests locally until the server is available, and then offloads the remaining processing. We call this partial offloading and it further reduces the server load by processing the first few layers on the client side. The server is allowed to estimate and inform the client of the time required to finish the new and existing requests. This is easy since the client can perform one-time profiling [32] of popular DNNs to estimate the processing time of various layers.

To transmit the intermediate results to the server, we compress the output from the client after processing some layers so that it can be sent efficiently to the server. We observe that video compression is much faster than file compression schemes (*e.g.*, gzip), owing to available hardware accelerators, so we use lossless video compression H.264. It takes 1.5ms to compress at the client and 0.6ms to decompress at the server. In order to use H.264, we quantize the intermediate output from the client to INT8 and feed the output to the server. Running quantization incurs considerable overhead, so we run quantized models to remove the need of quantizing the input or output while speeding up DNN processing. Running quantized models yield little degradation in the accuracy [7], [60].

Even with compression, the intermediate data size for the first few layers (which can finish on the client without incurring large delay) is larger than the original image size. Therefore, when the server is powerful as in our evaluation, partial offloading is attractive only for fast networks, such as 5G. On the other hand, when the server is slow (*e.g.*, IoT devices), partial offloading is useful even on a slow network as in [32]. Our approach automatically makes offload decisions based on the server, client, and network speed.

Our offloading strategies can be extended to incorporate the client's energy (*e.g.*, a client considers local processing as an option only when its battery power exceeds a threshold. Otherwise, it always offloads to the server).

IV. System Implementation

Our high-level system architecture is shown in Fig. 2.

A. Server Implementation

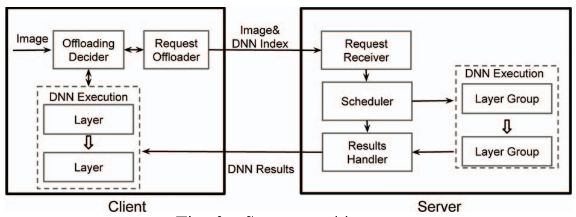


Fig. 2. System architecture.

The server uses Pytorch [48] to run DNNs on Nvidia Tesla P100. It keeps track of requests at each layer and executes the scheduling algorithms implemented in Python. We revise the Pytorch DNN API to run a specified set of requests through specified layers. We use CUDA synchronization [9] before running a group so that GPU does not have any other active threads. We also use CUDA synchronization before we start running the next group so that all GPU threads for the existing group of layers have already been completed. When we create a large batch, we also need to copy the input for each request to a continuous GPU memory block. The memory manager allocates a memory block when forming a batch and releases that block when the batch finishes running the next layer. Our scheduling algorithm requires the running time of each layer as the input. We perform one-time profiling for the running time of each layer in various DNNs by varying the batch size.

B. Collaborative DNN Execution

We implement our client on Nvidia Jetson Nano [44]. Our system can run on any device (*e.g.*, IoT devices, etc). Less powerful mobile devices tend to offload all DNN tasks, while more powerful devices (*e.g.*, Nvidia Jetson Nano/TX2) process more requests locally. The client generates requests, which consist of images, arrival time, and the DNN to use. As described in Section III-D, the client determines whether to offload the current request. If the request runs locally, the client runs some or all layers in the DNN using TensorRT [45], which is compatible with Pytorch. If the request requires complete offloading, the client will transmit the JPEG [58] image and index of the DNN to the server via TCP. The server loads the DNN requested by the client to the memory, uses the JPEG library from OpenCV [46] to decompress the image, and feeds the decompressed data to the DNN as the input. If the request requires partial offloading, the client transmits intermediate results compressed by H.264 along with the next layer index in the DNN to the server, and the server decompresses them and finishes the remaining processing. The intermediate results consist of a sequence of feature maps, each of which is a gray-scale image. In both cases, upon finishing the DNN processing, the server uses the TCP to transmit results to the client.

V. Evaluation

A. Evaluation Methodology

DNN request traces: We generate DNN requests using a Poisson process by default. The average arrival rate is $10-350$ requests/sec. Each experiment runs 5000 requests. We also try Pareto and deterministic inter-arrival time to understand the impact of different request arrival patterns.

Network traces: We use the packet traces [12] collected from LTE uplink connections. Each one lasts for around 20min. The throughput is within 4–20Mbps. We use them to generate the reception timestamp of requests at the server. We only use transmission delay as the propagation delay to the edge server is negligible compared with the DNN processing time [42].

Image traces: We use video traces from the dataset MOT16 [43]. The images are resized to 240×240, which is the input resolution of the pre-trained DNNs used in our experiments. Our system can run DNNs with any input resolution. The relative performance of different algorithms remains the same when the image resolution and GPU memory increase by the same amount. We use JPEG to compress images. The size of images varies from 0.12Mbits to 0.33Mbits.

DNNs: We evaluate popular DNNs for different analytics tasks: VGG16 [55], ResNet50 [18] and GoogleNet [56] for classification, SSD [38] for object detection, SDCNet [51] for video prediction, and RTA [24] and FCN [39] for video segmentation. We load all the models to the memory at the beginning, so there is no overhead of model loading when we switch DNNs when running multiple DNNs.

Performance metrics: We use three metrics: (i) completion time: the time duration from request generation to getting the DNN execution results at the client. The completion time captures the end-to-end latency of a request, which includes the latency of every step the request goes through in our system, including running the scheduling algorithm and performing memory copy. (ii) ratio of on-time requests: the ratio of requests that meet the user's deadline, and (iii) capacity: the maximum request rate at which the on-time ratio is above 90%. We can easily see the system capacity from the on-time ratio graphs by looking for the load beyond which the on-time request ratio falls below 90%. The default deadline is 300ms and 150ms for evaluations with and without collaborative execution, respectively. We also vary the deadline to understand its impact.

Algorithms: We compare our algorithms with two baselines: (i) *No-Batch*, which runs all requests one by one, (ii) *Batch*, which sorts the requests in an increasing order of their arrival time and batches all requests starting from the first one up to the bound B. *Batch* and Our algorithms run on the modified version of PyTorch to support batching at different layers. Our modified PyTorch has little overhead: its Batch=1 version is equivalent to No-Batch and takes only 3ms longer.

Testbed: We develop a testbed to evaluate the performance of various algorithms. We generate requests from multiple clients using a single Linux machine based on real traces. We run DNN requests on the edge server in real time, which means that our results include all system overheads, including memory copying, thread switching and overheads related to monitoring and dynamically changing the PyTorch execution graph. For collaborative execution, we use one real client to run all the client-side components on Nvidia Jetson Nano and send requests over WiFi. Requests from other clients are generated by a Linux machine according to the real traces collected from

Nvidia Jetson Nano. *All evaluation results are from our testbed.*

B. Minimize One DNN Completion Time

We compare the performance of serving requests for a single DNN in our system. We vary the request rate from 10 to 150 requests per second (req/sec). The request deadline is 150 ms.

Fig. 3 shows our algorithm achieves the highest system capacity for all DNNs. The capacity of our algorithm is 120, 160, 320, 70 and 110 for VGG16, ResNet50, GoogleNet, FCN and SSD, respectively. The corresponding numbers are 100, 110, 90, 50 and 90 for the *Batch* strategy, and are 50, 50, 50, 30, 70 for *No-Batching*. Our algorithm improves system capacity over *Batch* by 20%, 36%, 67%, 40%, 22% for VGG16, ResNet50, GoogleNet, FCN and SSD, respectively; the corresponding improvement over *No-Batch* is 140%, 200%, 400%, 40% and 57%. The system capacity improvement of our algorithm comes from strategically harnessing the batching benefits. ResNet50 and GoogleNet have more system capacity than the other DNNs due to more batching benefits.

Moreover, our scheduling algorithm not only improves capacity but also the completion time. It cuts down the completion time by up to 53% over *Batch* when the request rate is below the capacity of the *Batch* strategy and by up to 29% over the *No-Batch* strategy when the request rate is below the capacity of the *No-Batch* strategy. Our scheduling algorithm also improves the on-time ratio. The improvement is up to 69% over *Batch* and 37% over *No-Batch*.

Comparison with Nexus: Fig. 4 compares our approach with Nexus [54] using requests with a Deterministic inter-arrival time for VGG16. Our approach reduces the completion time over Nexus by 10.5% when the request rate is 95. Our benefit is even larger in multiple DNNs shown in Sec. V-D.

C. Maximize On-Time Ratio

Next, we consider minimizing # of tardy jobs. We compare three schemes: (i) EDF, (ii) Our-Time, our algorithm that minimizes completion time, (iii) Our-Tardy, our algorithm that maximizes on-time ratio. The request deadline is 150ms. All schemes drop jobs that have passed their deadline. Fig. 5 shows all schemes have close to 100% on-time ratio when the request rate is low; Our-Tardy yields the highest on-time ratio as request rate increases. For ResNet, the system capacity of EDF, Our-Time, and Our-Tardy is 80, 80 and 100. For VGG, the corresponding numbers are 90, 100 and 110. Our-Tardy improves EDF and Our-Time by 22-25% and 10-22%. Our-Time can satisfy more requests' deadline than EDF even though it does not explicitly consider the deadlines. This is because Our-Time minimizes the completion time, which indirectly reduces tardy jobs. EDF is less effective than Our-Time since the batching benefit is not considered. Scheduling jobs by the deadline order only may reduce the batching opportunity, which causes higher running time and more tardy jobs.

We evaluate the on-time ratio by varying the deadline and request rates. The system capacity of Our-Tardy is 110 when the deadline is 150ms. Increasing deadline allows more requests to be served on time. All schemes have close to 100% on-time ratio when the request rate is 100 and the deadline is higher than 200ms. When the request rate is 120 and the deadline is higher than 200ms, both Our-Time and Our-Tardy improve EDF by around 50%. When the deadline is below 200ms, both of them improve the on-time ratio by more than 20% over EDF. Therefore, our algorithm is effective in maximizing on-time ratio even if requests have different deadlines.

D. Performance for Multiple DNNs

DNNs without shared layers: Fig. 6(a) shows the performance when the requests are equally split between two DNNs without shared layers. When serving ResNet50 and GoogleNet, the system capacity of *No-Batch*, *Batch* and our algorithm are 30, 110 and 190. This is 500% and 73% improvement over *No-Batch* and *Batch*. We also run VGG16 and GoogleNet. The capacity of *No-Batch*, *Batch* and our algorithm are 50, 90 and 150, which is 200% and 67% improvement over *No-Batch* and *Batch*. Our algorithm performs the best even for DNNs without shared layers by increasing batching opportunities in the same DNN. If the request rate is below 110, the on-time ratios for *Batch* and our algorithm are 83% and 96%, when serving ResNet50 and GoogleNet. We observe similar pattern when serving VGG16 and GoogleNet. Even if the request rate is within the system capacity of both our algorithm and *Batch*, our algorithm can still achieve higher on-time request ratio due to faster processing rate.

DNNs with shared layers: Fig. 6(b) shows the performance when all requests go through the same optical flow model – FlowNet2 and then are equally split between SDCNet and RTA. The system capacity are 20, 60 and 90 for *No-Batch*, *Batch* and our algorithm. So our approach yields $4.5\times$ and $1.5\times$ system capacity of *No-Batch* and *Batch*, respectively.

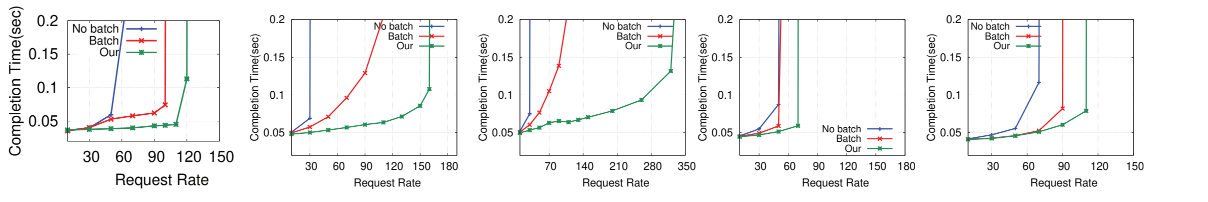
Since these two models are more time-consuming than others, we set the request deadline to 300ms for the evaluation. The average completion time is 0.232sec and the average ratio of on-time requests is 98% for our algorithm when the request rate is within the capacity. Without batching requests at the shared layers, the system capacity remains the same but the average completion time increases to 0.314sec, which is 35% higher than enabling batching. Its on-time request ratio reduces to 42%. Thus, batching at the shared layers is beneficial.

Comparison with Nexus [54]: We compare our approach with Nexus [54] when the requests all go through FlowNet2 and then equally split between SDCNet and RTA. When the request rate is 70, the following table shows our approach reduces the completion time by 27%, 32%, 25% under Poisson, Pareto, and Constant inter-arrival time, respectively.

Completion time (sec).	Poisson	Pareto	Constant
Ours	0.124	0.157	0.109
Nexus	0.158	0.207	0.136

E. Collaborative DNN Execution

In our collaborative algorithm, a request is offloaded only when local execution is too slow to meet its deadline. The default deadline is 300ms. The server runs Our-Tardy to maximize the on-time ratio. In our experiments, all the client requests run either VGG16 or FCN. We vary the number of clients and each client generates requests according to a Poisson



(a) VGG16 (b) ResNet50 (c) GoogleNet (d) FCN (e) SSD

Fig. 3. Completion Time for a single DNN.

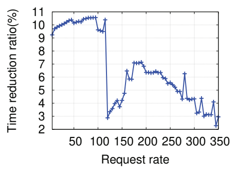
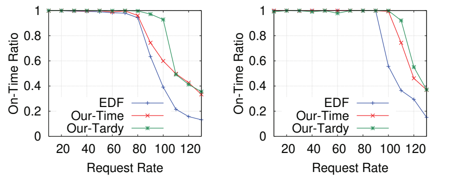
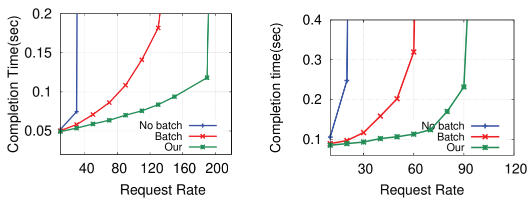


Fig. 4. Comparison with Nexus for VGG16.



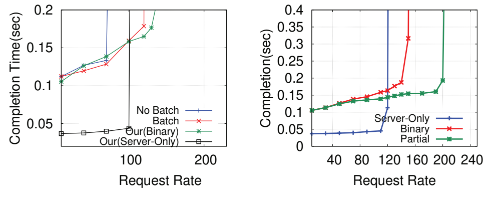
(a) ResNet (b) VGG

Fig. 5. Maximize on-time ratio for ResNet and VGG.



(a) w/o share (ResNet, GoogleNet) (b) w/ share (SDCNet, RTA)

Fig. 6. 2 DNNs with and without shared layers.



(a) Binary offloading (b) Partial offloading

Fig. 7. Collaborative execution for VGG16.

process with a mean arrival rate of 10 req/sec. Among those clients, one client is running all the client-side components on Jetson Nano and communicates with the server via WiFi, while the others are simulated to generate requests to the server.

Binary offloading: Fig. 7(a) shows the performance of VGG16 with binary offloading. The ratio of requests offloaded to server is 0.48, 0.53 and 0.60 for *No-Batch*, *Batch* and our algorithm, respectively. Clients offload more requests in our scheme due to its faster server processing. Our binary offloading runs requests locally as long as they can finish within the deadline. In this case, the completion time of the local requests may increase. For example, Fig. 7(a) shows that the completion time is around 0.125 sec for VGG16 and 0.136 sec for FCN, which is higher than the server processing time in Fig. 3(a) and 3(d). This is acceptable since the requests still finish in time and client processing saves network and server cost.

With binary offloading, the system capacity in VGG16 is 70, 120 and 140 for *No-Batch*, *Batch* and our algorithm. The corresponding numbers are 30, 50 and 70 for FCN. Binary offloading improves the system capacity of serving VGG16 by 25% and 17% for *Batch* and our algorithm. When serving FCN, the capacity improvement of *Batch* and our algorithm is 40% and 29%. By running some requests locally on the client, our scheme has a higher capacity. *No-Batch* has the same capacity w/ and wo/ binary offloading for VGG16 because the client is not fast enough to reduce the server's queue.

Partial offloading: The completion time includes client processing time, network transmission time, and server processing time. Due to the relatively large intermediate results, we multiply the throughput by 10× so that it is closer to that in the 5G networks. The client runs quantized DNN layers, uses H.264 to compress intermediate results, and transmits data over WiFi. Fig. 7(b) shows the performance of partial offloading when running VGG16. The binary offloading improves the system capacity from 120 in the server-only scheme (Sec. III-A) to 140, and partial offloading further improves the capacity to 200, which translates to 67% increase in the system capacity over the server-only scheme. For FCN, the binary offloading improves the system capacity from 70 to 90, and partial offloading further improves to 120, out-performing the server-only scheme by 71%, since the client can more often perform local processing.

VI. Conclusion

We develop batch-aware DNN scheduling for edge servers. It supports (i) different optimization objectives: minimizing completion time or maximizing job on-time ratio, (ii) requests using the same or different DNNs with or without shared layers, and (iii) collaborative DNN execution to further reduce processing delay by adaptively running some or portions of requests locally at the client side. Our extensive evaluation demonstrates its effectiveness.

References

[1] https://osf.io/r7wsc/?view_only=0daed714f6264c32a2f039a2eff9bd6b.

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In {*OSDI*} *16)*, pages 265–283, 2016.

[3] P. Brucker, A. Gladky, H. Hoogeveen, M. Y. Kovalyov, C. N. Potts, T. Tautenhahn, and S. L. Van De Velde. Scheduling a batching machine. *Journal of scheduling*, 1(1):31–54, 1998.

[4] J. Carreira, P. Agrawal, K. Fragkiadaki, and J. Malik. Human pose estimation with iterative error feedback. In *CVPR*, 2016.

[5] L.-C. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam. Encoder-decoder with atrous separable convolution for semantic image segmentation. In *ECCV*, pages 801–818, 2018.

[6] Y. Choi, Y. Kim, and M. Rhu. Lazy batching: An sla-aware batching system for cloud machine learning inference. In *HPCA*. IEEE, 2021.

[7] Y. Choukroun, E. Kravchik, F. Yang, and P. Kisilev. Low-bit quantization of neural networks for efficient inference. In *ICCVW*. IEEE, 2019.

[8] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, 2017.

[9] Cuda runtime API. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html.

[10] B. Fang, X. Zeng, and M. Zhang. Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision. In *MobiCom*, 2018.

[11] Z. Fang, T. Yu, O. J. Mengshoel, and R. K. Gupta. Qos-aware scheduling of heterogeneous servers for inference in deep neural networks. In *CIKM*, pages 2067–2070. ACM, 2017.

[12] S. Fouladi, J. Emmons, E. Orbay, C. Wu, R. S. Wahby, and K. Winstein. Salsify: low-latency network video through tighter integration between a video codec and a transport protocol. In {*NSDI*} *18*, 2018.

[13] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace. Serving {DNNs} like clockwork: Performance predictability from the bottom up. In *OSDI 20*, pages 443–462, 2020.

[14] R. Hadidi, J. Cao, Y. Xie, B. Asgari, T. Krishna, and H. Kim. Characterizing the deployment of deep neural networks on commercial edge devices. In *IISWC*, 2019.

[15] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[16] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *MobiSys*. ACM, 2016.

[17] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.

[18] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, June 2016.

[19] J. Honovich. Frame rate guide for video surveillance. https://ipvm.com/reports/frame-rate-surveillance-guide.

[20] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[21] C. Hu, W. Bao, D. Wang, and F. Liu. Dynamic adaptive dnn surgery for inference acceleration on the edge. In *INFOCOM 2019*. IEEE, 2019.

[22] Y. Hu, S. Rallapalli, B. Ko, and R. Govindan. Olympian: Scheduling gpu usage in a deep neural network model serving system. In *Proceedings of the 19th International Middleware Conference*. ACM, 2018.

[23] Z. Hu, A. B. Tarakji, V. Raheja, C. Phillips, T. Wang, and I. Mohomed. Deephome: Distributed inference with heterogeneous devices in the edge. In *EMDL*, 2019.

[24] P.-Y. Huang, W.-T. Hsu, C.-Y. Chiu, T.-F. Wu, and M. Sun. Efficient uncertainty estimation for semantic segmentation in videos. In *ECCV*, 2018.

[25] L. N. Huynh, Y. Lee, and R. K. Balan. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In *MobiSys*, pages 82–95. ACM, 2017.

[26] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[27] E. Ilg, N. Mayer, T. Saikia, M. Keuper, A. Dosovitskiy, and T. Brox. Flownet 2.0: Evolution of optical flow estimation with deep networks. In *CVPR*, pages 2462–2470, 2017.

[28] Inference: The next step in gpu-accelerated deep learning. https://devblogs.nvidia.com/inference-next-step-gpu-accelerated-deep-learning/.

[29] A. H. Jiang, D. L.-K. Wong, C. Canel, L. Tang, I. Misra, M. Kaminsky, M. A. Kozuch, P. Pillai, D. G. Andersen, and G. R. Ganger. Mainstream: Dynamic stem-sharing for multi-tenant video processing. In *ATC*, 2018.

[30] J. Jiang, G. Ananthanarayanan, P. Bodik, S. Sen, and I. Stoica. Chameleon: scalable adaptation of video analytics. In *Sigcomm*. ACM, 2018.

[31] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. Noscope: optimizing neural network queries over video at scale. *Proceedings of the VLDB Endowment*, 10(11):1586–1597, 2017.

[32] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *SIGARCH*, volume 45, pages 615–629. ACM, 2017.

[33] Y. Kim, J. Kim, D. Chae, D. Kim, and J. Kim. μlayer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization. In *EuroSys*, page 45. ACM, 2019.

[34] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *IPSN*. IEEE Press, 2016.

[35] L. Liu, H. Li, and M. Gruteser. Edge assisted real-time object detection for mobile augmented reality. In *MobiCom*. ACM, 2019.

[36] Q. Liu and T. Han. Dare: Dynamic adaptive mobile augmented reality with edge computing. In *ICNP*, pages 1–11. IEEE, 2018.

[37] S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. Du. On-demand deep model compression for mobile devices: A usage-driven model selection framework. In *MobiSys*, pages 389–400. ACM, 2018.

[38] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. Ssd: Single shot multibox detector. In *ECCV*. Springer, 2016.

[39] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In *CVPR*, pages 3431–3440, 2015.

[40] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *ECCV*, 2018.

[41] A. Mathur, N. D. Lane, S. Bhattacharya, A. Boran, C. Forlivesi, and F. Kawsar. Deepeye: Resource efficient local execution of multiple deep vision models using wearable commodity hardware. In *MobiSys*, 2017.

[42] R. McLaughlin. 5g low latency requirements. https://broadbandlibrary.com/5g-low-latency-requirements/.

[43] A. Milan, L. Leal-Taixé, I. Reid, S. Roth, and K. Schindler. Mot16: A benchmark for multi-object tracking. *arXiv preprint arXiv:1603.00831*, 2016.

[44] Nvidia jeston nano developer kit. https://developer.nvidia.com/embedded/jetson-nano-developer-kit.

[45] Nvidia tensorrt. https://developer.nvidia.com/tensorrt.

[46] Opencv. https://opencv.org.

[47] G. Papandreou, T. Zhu, N. Kanazawa, A. Toshev, J. Tompson, C. Bregler, and K. Murphy. Towards accurate multi-person pose estimation in the wild. In *CVPR*, pages 4903–4911, 2017.

[48] 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 *Neurips*, 2019.

[49] C. N. Potts and M. Y. Kovalyov. Scheduling with batching: A review. *European journal of operational research*, 120(2):228–249, 2000.

[50] X. Ran, H. Chen, X. Zhu, Z. Liu, and J. Chen. Deepdecision: A mobile deep learning framework for edge video analytics. In *INFOCOM*, 2018.

[51] F. A. Reda, G. Liu, K. J. Shih, R. Kirby, J. Barker, D. Tarjan, A. Tao, and B. Catanzaro. Sdc-net: Video prediction using spatially-displaced convolution. In *ECCV*, pages 718–733, 2018.

[52] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Neurips*, 2015.

[53] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018.

[54] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram. Nexus: a gpu cluster engine for accelerating dnn-based video analysis. In *SOSP*, pages 322–337. ACM, 2019.

[55] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[56] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *CVPR*, 2015.

[57] X. Tang, P. Wang, Q. Liu, W. Wang, and J. Han. Nanily: A qos-aware scheduling for dnn inference workload in clouds. In *HPCC/SmartCity/DSS*, pages 2395–2402. IEEE, 2019.

[58] G. K. Wallace. The jpeg still picture compression standard. *IEEE transactions on consumer electronics*, 38(1):xviii–xxxiv, 1992.

[59] H. Wang, V. Bhaskara, A. Levinshtein, S. Tsogkas, and A. Jepson. Efficient super-resolution using mobilenetv3. In *Computer Vision–ECCV 2020 Workshops*. Springer, 2020.

[60] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius. Integer quantization for deep learning inference: Principles and empirical evaluation. *arXiv preprint arXiv:2004.09602*, 2020.

[61] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng. Quantized convolutional neural networks for mobile devices. In *CVPR*, pages 4820–4828, 2016.

[62] M. Xu, M. Zhu, Y. Liu, F. X. Lin, and X. Liu. Deepcache: principled cache for mobile deep vision. In *MobiCom*, pages 129–144. ACM, 2018.

[63] M. Yang, S. Wang, J. Bakita, T. Vu, F. D. Smith, J. H. Anderson, and J.-M. Frahm. Re-thinking cnn frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge. In *RTS*, 2019.

[64] C.-L. Zhang, X.-X. Liu, and J. Wu. Towards real-time action recognition on mobile devices using deep models. *arXiv preprint arXiv:1906.07052*, 2019.

[65] W. Zhang, S. Teng, Z. Zhu, X. Fu, and H. Zhu. An improved least-laxity-first scheduling algorithm of variable time slice for periodic tasks. In *ICCI*, pages 548–553, 2007.

[66] X. Zhang, X. Zhou, M. Lin, and J. Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *CVPR*, 2018.