Jellyfish: Timely Inference Serving for Dynamic Edge Networks

Vinod Nigade, Pablo Bauszat, Henri Bal, Lin Wang
Vrije Universiteit Amsterdam

***Abstract*—While high accuracy is of paramount importance for deep learning (DL) inference, serving inference requests on time is equally critical but has not been carefully studied especially when the request has to be served over a dynamic wireless network at the edge. In this paper, we propose Jellyfish—a novel edge DL inference serving system that achieves soft guarantees on end-to-end inference latency often specified as a service-level objective (SLO). To handle the network variability, Jellyfish exploits both data and deep neural network (DNN) adaptation to conduct tradeoffs between accuracy and latency. Jellyfish features a new design that enables collective adaptation policies where the decisions for data and DNN adaptations are aligned and coordinated among multiple users with varying network conditions. We propose efficient algorithms to dynamically adapt DNNs and map users, so that we fulfill latency SLOs while maximizing the overall inference accuracy. Our experiments based on a prototype implementation and real-world WiFi and LTE network traces show that Jellyfish can meet latency SLOs at around the 99th percentile while maintaining high accuracy.**

I. Introduction

In the past decade, modern applications such as augmented reality, intelligent personal assistants, and autonomous driving [1]–[4] have proliferated. A considerable number of these applications are based on deep learning (DL) inference, e.g., analyzing continuous video streams to understand the environment with pre-trained deep neural networks (DNNs) [5]. Employing sophisticated learning techniques [6], [7], these DNNs typically demand intensive computations, making them hard to deploy on mobile and IoT devices due to the limited capability of these devices. Therefore, DL inference for mobile and IoT applications is often offloaded to a more powerful nearby computing platform such as edge servers equipped with high-end accelerators like GPUs or TPUs [8].

Handling DL inference requests is generally referred to as *inference serving*, where requests are scheduled to computing resources (e.g., GPUs). Then, the corresponding DNN is loaded on the computing resources to execute the request, taking the data associated with the request as input. DL inference serving has been extensively studied recently with frameworks including Clipper [9], Nexus [10], Clockwork [11], and INFaas [12]. The general goal is to achieve resource efficiency and/or guarantee inference latency (e.g., serving requests within 100ms [11]), as typically specified in the service-level objective (SLO) of modern applications.

Despite the enormous efforts, virtually all existing DL inference serving systems focus on the server part, leaving out the network part when specifying the SLO. However, inference requests with input data generated by mobile or

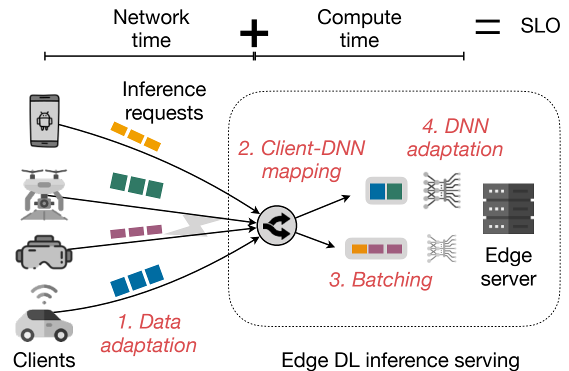


Fig. 1: Collective DNN adaptation for timely edge DL inference serving.

IoT devices need to travel through a (wireless) network before they arrive at the edge server. Such a network typically shows high performance variability [13], [14], causing variable delays in network transmission for inference requests. Hence, SLOs for mobile and IoT applications should be specified end-to-end, covering both the network and compute parts. Being agnostic to the network time, edge DL inference serving systems risk ending up with insufficient time to process the request (e.g., under poor network conditions), leading to SLO violations. Therefore, considering network time and end-to-end SLOs poses new challenges and calls for new designs for timely edge DL inference serving for mobile and IoT applications.

In this paper, we propose Jellyfish—a novel framework for timely inference serving at the edge, *aiming to guarantee the end-to-end SLO* while achieving high inference accuracy. Jellyfish relies on two adaptation strategies to achieve tradeoff between accuracy and latency: data adaptation to adjust the input data size and DNN adaptation to switch between DNNs. Jellyfish features a new design that enables *collective adaptation* policies. More specifically, Jellyfish aligns the data and DNN adaptation decisions for each client and coordinates the adaptation decisions among multiple clients by provisioning a pool of DNNs with different latency-accuracy tradeoff profiles to serve the requests from these clients collectively. One major benefit of such a design is the potential of leveraging request batching—a known technique for improving resource efficiency in DL inference serving [9], [10]. The higher resource efficiency in Jellyfish translates into more room for inference accuracy improvements under latency constraints, but at the cost of more complex scheduling decision-making that involves multiple steps, as depicted in Fig. 1.

Jellyfish addresses the scheduling challenges with a set of efficient algorithms. Particularly, given a collective DNN

adaptation decision (i.e., a selected set of DNN instances), Jellyfish first solves the client-DNN mapping problem by applying dynamic programming. The client-DNN mapping algorithm also leverages batching to the maximum and outputs the corresponding request batching decision for each DNN instance. Upon system status changes, Jellyfish employs a separate procedure to adapt (select) DNNs incrementally based on simulated annealing [15]. In addition, Jellyfish adopts a greedy DNN instance prefetching strategy to reduce DNN adaptation overhead on the server. Finally, Jellyfish keeps informing each client about the input size of the DNN to which they are mapped, so that the client performs data adaptation by sending inference requests with that particular data size.

In developing Jellyfish, we make the following contributions:

1) We present Jellyfish, a new DL inference serving system for dynamic edge networks based on the idea of collective DNN adaptation, aiming to achieve soft SLO guarantees.
2) We formulate the collective DNN adaptation problem considering the latency constraints, and propose efficient algorithms for dynamic client-DNN mapping, request batching, and DNN selection.
3) We design and implement a prototype for Jellyfish and demonstrate its effectiveness by conducting extensive experiments for popular video analytics inference tasks with real-world network traces. Our results show that Jellyfish can meet the SLO of inference requests around 99% of the time while maintaining high accuracy.

II. Background and Motivation

A. DL Inference Serving

Today, DL-based mobile and IoT applications like augmented reality and intelligent personal assistants rely on deep neural networks (DNNs) to complete inference tasks like object detection and speech recognition [1], [3], [4]. A DNN consists of multiple layers. To achieve high accuracy, DNNs employ an increasing number of layers [6], [16], leading to unprecedented computing demands for DNN execution. However, mobile and IoT devices are typically resource-constrained, incapable of completing DL-based inference on time with state-of-the-art DNNs. Furthermore, battery life is usually a big concern for these devices. Hence, DL inference tasks are often offloaded to more powerful computing platforms such as edge servers equipped with high-end GPUs and TPUs [1], [3].

DL inference serving on servers has been extensively studied recently [9]–[12]. Applications based on DL inference typically require some form of latency guarantee, often specified as a service-level objective (SLO), to ensure the usefulness of the inference result. For example, digital assistance like Amazon Alexa dictates that the tail latency is constrained within 200–300ms [17]. Current inference serving frameworks like Nexus and Clockwork focus mainly on meeting SLOs via inference request scheduling, leveraging the high predictability of DNN execution time. However, most of these frameworks assume that a fixed SLO is specified for the DNN execution part and optimize only the inference serving time towards this SLO.

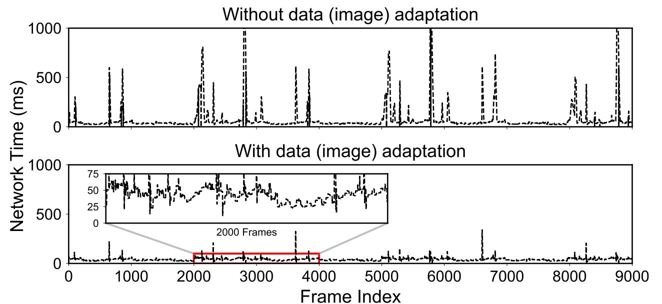


Fig. 2: Network time for sending JPEG images with adaptive resolutions over an LTE network (bandwidth shown in Fig. 10).

We argue that this is insufficient for inference serving at the edge for mobile and IoT applications. Typically, inference requests with input data (e.g., an image) issued by mobile or IoT devices travel through a dynamic (wireless) network (e.g., WiFi or cellular) before they reach the edge server. As a result, the time left for computing (i.e., inference serving) on the server can experience significant variations due to the variable network time caused by the variable network performance (see Fig. 2). Consequently, the application SLO should be defined *end-to-end*, including both the *network* and *compute* time. Ideally, edge DL inference serving for mobile and IoT applications should consider jointly the network and compute parts in the pipeline and be adaptive to network dynamics.

B. Adaptation Techniques for Inference Serving Systems

DNN adaptation. The idea of DNN adaptation is to choose between functionally-equivalent DNNs with different latency-accuracy tradeoff profiles. Generally, this can be achieved by two approaches: (1) DNN switching relies on a set of DNNs optimized with different depths, widths, or numerical precision offline [18], [19]. The idea has been applied in several DL inference serving systems, such as ALERT [20], where the DNN is switched continuously at runtime to meet latency, accuracy, and energy constraints. (2) Dynamic DNNs enable the partial execution of the DNN (e.g., a sub-network or early-exit) at runtime depending on the changing input data content or resource availability [21], [22]. Overall, DNN adaptation techniques are agnostic to the variable network time, making them, when applied alone, ineffective for end-to-end latency SLO guarantees over a dynamic edge network.

Data adaptation. When the input data for the DNN has to be transferred over a dynamic network [23], [24], data adaptation (e.g., changing the image resolution) can be used to reduce the input data size to avoid network bottlenecks (w.r.t. throughput), at the cost of reduced inference accuracy. To illustrate the power of data adaptation, we perform an experiment where we stream JPEG images over a dynamic LTE network and adapt the image resolution to ensure a stable throughput. Fig. 2 shows that data adaptation can help to smooth out the big spikes in the network time for each image, but still, significant variability can be observed. This shows that data adaptation, while beneficial, is not enough on its own to deal with tight end-to-end latency SLO requirements.

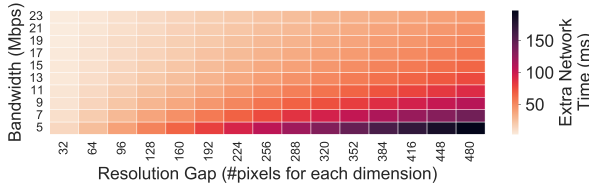


Fig. 3: The extra network time spent when a client sends JPEG images at a resolution larger than the input size of the DNN on the server under varying bandwidth conditions (covering the bandwidth range of a real-world LTE network). The resolution gap is defined as the chosen client-sending resolution minus the resolution expected by the DNN on the server.

C. Limitations of Existing Approaches

We identify the following two major limitations of existing approaches when used for inference serving with end-to-end latency SLOs under highly dynamic edge networks.

Misaligned adaptation decisions. Existing works mostly focus on either data or DNN adaptation [20], [21], [23], [24]. When simply combined, they could produce misaligned adaptation decisions, leading to suboptimal performance. For example, when the network condition is good, input data adaptation may choose a high resolution for the image data that is sent over the network. However, if the DNN running on the server expects a much lower resolution for its input due to a low compute time budget, the received image has to be downscaled before being served. This leads to resource waste in terms of both network time and bandwidth. To quantify this effect, Fig. 3 shows the extra network transmission time due to misaligned adaptation decisions, where up to 150ms of extra time is unnecessarily consumed simply for network transmission when the chosen input data size is larger than the input size of the chosen DNN. Conversely, if the chosen data size is smaller than the input size of the chosen DNN, the data has to be upscaled when reaching the server, which potentially affects the DNN accuracy adversely [25]. To ensure the end-to-end latency SLO, the decisions for the two adaptation strategies need to be aligned.

Uncoordinated adaptations for multiple clients. Many existing works on adaptive inference focus on a single-client setup where the adaptation is applied to a single inference pipeline [23], [24], [26], [27]. Although such a setup could be simply replicated across multiple clients, we argue that such a design would lead to poor resource efficiency, which is detrimental to the resource-limited edge environment. Without coordination among the adaptation for different clients, the server would need to instantiate a large number of DNN instances, each for a client and possibly in a different size. Further, batching of inference requests from multiple clients would be prohibited, leading to poor resource efficiency especially when the inference request rate for each client is low. To avoid these issues, the adaptations for multiple clients need to be coordinated holistically.

None of the existing works are able to overcome these limitations simultaneously [27], [28]. We argue that a *collective adaptation* approach that holistically aligns and coordinates DNN and data adaptation decisions for multiple clients is required to address the aforementioned challenges.

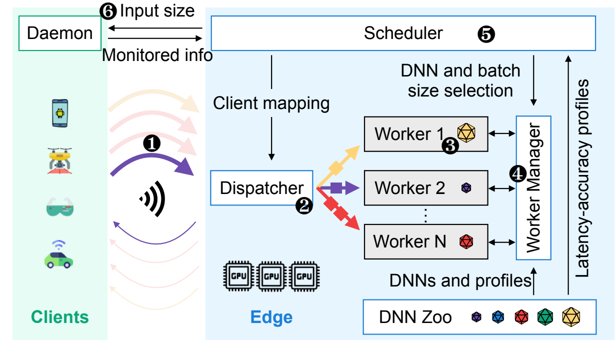


Fig. 4: An overview of the Jellyfish system architecture.

III. Jellyfish Design

Jellyfish's primary goal is to serve all the inference requests from multiple clients over the network and meet the request deadlines as defined by their SLOs. In this section, we discuss the architecture, general workflow, and main components of Jellyfish, which work in tandem to achieve the goal.

A. Overview

An overview of the Jellyfish system architecture is shown in Fig. 4. Jellyfish supports multiple clients simultaneously, and its major components are located on the edge side. When the clients ❶ send the requests to the edge over the network, the dispatcher component takes the client-DNN mapping from the scheduler and ❷ distributes the requests to workers running the expected DNN. Each worker is a separate process (on one or more edge servers) holding some GPU resources to ❸ serve inference requests with the batch size selected by the scheduler. The worker manager ❹ deploys DNNs (stored in the DNN zoo) to the workers following the DNN selection decision by the scheduler. The scheduler provides the intelligence of Jellyfish, where it takes the latency-accuracy profiles from the DNN zoo and the monitored information from the client daemon as input, and ❺ runs our scheduling algorithms periodically to decide the client-DNN mapping, DNN selection (adaptation), and batch size for each worker. The scheduler then ❻ informs all the clients about the input size of their mapped DNNs to start sending new requests at that particular input size (i.e., data adaptation *aligned* with DNN adaptation).

While Jellyfish is an edge-centric inference serving system, it requires some basic support (as daemons) from clients: (1) a metadata exchange mechanism (piggy-backed on the normal inference requests/responses) for sharing client side monitored information including the inference request rate and estimated network bandwidth, and the input size dictated by the client-DNN mapping from the scheduler, (2) a request pre-processing mechanism that adjusts the data to match the DNN input size or to the maximum possible size when matching exactly the DNN input size is impossible due to poor network conditions.

The end-to-end latency consists of two parts: network time (request and response) and compute time on the edge (for request dispatching and handling, request preprocessing if any, queuing, and DNN execution).

B. System Components

Dispatcher. The dispatcher distributes inference requests from clients to their respective workers. It first fetches the client-DNN mapping from the scheduler and then redirects the requests to the workers running the corresponding DNNs. The dispatcher also handles all the connections to clients and includes service endpoints to interact with the clients, e.g., to (de)register clients in the system.

Worker. Each worker is statically allocated on one GPU and maintains a local queue to buffer incoming requests. The worker process batches requests (resizing them if needed) in the queue and sends the request batches to the DNN deployed on the GPU for execution. The worker also implements a lazy dropping policy at the queue where requests that are too late to be processed by the current DNN will be dropped directly without further processing (similar to [10], [11]). We exclusively employ GPUs for the DNN inference task analogous to other serving systems [10], [11]. Our system and algorithms equally apply to CPUs or other accelerators, provided that the predictability and stability of inference latencies hold.

Worker manager. The worker manager is responsible for deploying and adapting DNNs on the workers. Supplied with the DNN selection decision made by the scheduler, the worker manager fetches the DNN from the DNN zoo and loads the DNN (moving from the host memory to the GPU memory) on the GPU of the worker. The worker manager also instructs the worker about the batch size to use with the deployed DNN. Upon receiving new decisions from the scheduler, the worker manager swaps out the current DNNs and loads the new DNNs. However, swapping DNNs on the GPU can be time-consuming and cause delays in DNN updates. To alleviate this issue, we preload a set of DNNs that are neighboring (in input size) the currently selected DNN (see §IV-D).

Scheduler. The scheduler provides the intelligence of the system by making the adaptation decisions. The goal of the scheduler is to maximize the overall accuracy while meeting the latency SLOs for all clients. The scheduler continuously collects and maintains the following information: client state (i.e., request rate, SLO, and bandwidth estimation), edge state (currently deployed DNNs and client-DNN mapping), and DNN profiles from the DNN zoo. The scheduler then feeds such information to a set of scheduling algorithms periodically (or upon system state changes) to (re)generate decisions for DNN selection, batch size, and client-DNN mapping. We layout the detail of the scheduling algorithms in §IV.

DNN zoo. The DNN zoo keeps a set of DNNs with different input sizes for the same DL inference task, enabling latency-accuracy tradeoffs in DNN adaptation. To generate these DNNs with varying architectures and input sizes, there exist several techniques, such as bag of models [29], early-exit [30], and neural architecture search [31]. We leverage the bag of models[1] technique to select a list of pre-trained DNNs. We sort DNNs in increasing order of their input sizes. After sorting, we expect the accuracies of these DNNs to follow an increasing order; otherwise, we simply remove the DNNs with lower accuracy but a larger input size. We profile (and store) the latency and accuracy of these DNNs for different batch sizes.

[1] A collection of functionally-equivalent DNNs with varying architectures and latency-accuracy tradeoff profiles.

Client daemon. The client runs a daemon process to collect local metadata (e.g., request rate, bandwidth estimation, and SLO) to share with the scheduler on the edge. Upon the transfer of each inference request, the client daemon estimates the network bandwidth for that request. To this end, we can employ the online network bandwidth estimation techniques used in recent works [24], [26], [30], [32].

IV. Scheduling Algorithms

In this section, we provide the formulation of the scheduling problem and present our algorithm design.

A. Problem Formulation

Suppose the DNN zoo holds a set of diverse DNNs denoted by $\mathbb{M} = \{m_1, m_2, ..., m_M\}$. Each DNN $m_j \in \mathbb{M}$ is associated with profiles including inference latency $l_j(b)$, throughput $t_j(b) = b/l_j(b)$, and expected accuracy a_j, where $b \in [1...B]$ is the batch size bounded by a given integer B. We enumerate the DNNs in set $\mathbb{M}$ in the increasing order of the inference latency. Similar to other works [20], [24], [33], we assume that a smaller DNN (i.e., with smaller input size) has lower inference latency, but also lower expected accuracy. The accuracy of DNNs can be modeled as a non-decreasing function of the DNN size [34]. The inference latency can be modeled as an increasing function of the DNN size and the batch size. When the batch size increases, the inference latency grows sub-linearly, leading to increased throughput with diminishing returns at larger batch sizes [35].

The set of workers performing DL inference is represented by $\mathbb{G} = \{g_1, g_2, ..., g_K\}$. We assume each worker exclusively occupies one GPU to run the DNN to serve inference requests. More fine-grained GPU sharing mechanisms such as NVIDIA multi-process service (MPS) or multi-instance GPU (MIG) can also be employed [36], where each instance is treated as a separate worker. The DNN execution time is highly predictable [11], so we use DNN latency profiles obtained offline for online latency prediction.

Suppose the system is serving a set of clients given by $\mathbb{C} = \{c_1, c_2, ..., c_N\}$. Each of the clients c_i generates inference requests with input size s_i at rate λ_i. Both the set of clients and the request rate can be time-varying; for the ease of expression, we omit the time index in the notation. Each client will be mapped to a worker on the edge side and inference requests from this client are sent to that particular worker. The client also specifies the SLO, i.e., the end-to-end inference latency, as O_i. The network bandwidth at client c_i is denoted by W_i, which is estimated by the client daemon as discussed in §III-B.

The scheduling problem of Jellyfish aims to find the *optimal multiset of DNNs* to be deployed on the workers, the *client-DNN mapping*, and the *batch size* for each worker, so as to maximize the expected accuracy of all served inference requests. We introduce a binary decision variable $x_{ijk} \in \{0, 1\}$ to denote

if a client c_i is mapped to DNN m_j deployed on worker g_k and an integer decision variable $b_k \in [1...B]$ to denote the batch size for the selected DNN on worker g_k. We also introduce an auxiliary decision variable $z_{kj} \in \{0, 1\}$ denoting the selection of DNN m_j on worker g_k. The scheduling problem can be formulated with the following integer program:

$$
\begin{aligned}
(P_1) \quad \max_{\{x,b\}} \quad & \textstyle\sum_{i,j,k} a_j \cdot \lambda_i \cdot x_{ijk} && (1)\\
\text{s.t.} \quad & \textstyle\sum_{j,k} x_{ijk} = 1, \forall i && (2)\\
& \textstyle\sum_{j} z_{kj} \leq 1, \forall k && (3)\\
& z_{kj} \geq x_{ijk}, \forall i, j, k && (4)\\
& \textstyle\sum_{j,k} x_{ijk} \cdot 2l_j(b_k) \leq \sum_{j,k} x_{ijk} \cdot L_{ij}, \forall i && (5)\\
& \textstyle\sum_{i,j} x_{ijk} \cdot \lambda_i \leq \sum_j z_{kj} \cdot t_j(b_k), \forall k && (6)\\
\text{vars} \quad & x_{ijk}, z_{kj} \in \{0, 1\}, b_k \in [1...B]
\end{aligned}
$$

The aim is to maximize the overall accuracy by serving requests with more accurate DNNs, given all requests are served within their SLOs. Thus, Eq. (1) defines the overall accuracy metric as the objective to maximize. Each client is mapped to only one DNN and one worker as specified in Eq. (2). Eq. (3) captures that at most one DNN is selected for each worker. Eq. (4) guarantees that all clients are mapped to the same and correct DNN when they are mapped to the same worker. Eq. (5) enforces the latency constraint specified with respect to the edge-side latency (compute) budget L_{ij} when mapping client c_i to DNN m_j. The *latency budget* can be calculated as $O_i - s_i/W_i$ (i.e., subtracting *network time* from SLO). We cap the queueing delay for an inference request on the edge side at the DNN execution time $l_j(b_k)$ (representing the worst case), which is also used in [10]. Thus, the latency (compute) budget should be at least *twice* the DNN execution time. Eq. (6) guarantees that the DNN m_j on worker g_k has adequate throughput capacity to support the aggregate request rate of all the mapped clients.

The above problem is hard to solve and existing solvers for mixed-integer linear program (MILP) cannot handle it in reasonable time (e.g., within a second). Our MILP implementation of the problem in CPLEX takes around 20s to 15min time with 4 threads for finding the optimal solution for a representative setup of 4 workers, 16 clients, and 16 DNNs with a maximum batch size of 12. To handle the complexity, we propose to tackle the problem by splitting it into two sub-problems: (1) client-DNN mapping and (2) DNN selection. We optimize each sub-problem iteratively to improve the overall accuracy objective without violating the latency SLO constraint.

B. Client-DNN Mapping

We first discuss the client-DNN mapping problem, which later serves as a building block for the DNN selection problem. The goal is to map the set of clients to a given set of DNN instances, optimizing the overall accuracy as defined in Eq. (1). Our client-DNN mapping algorithm is based on the key observation that the overall accuracy is maximized when the larger DNNs (more accurate ones) are assigned with higher aggregate request rates. We adopt a greedy approach where we first find clients and map them to the largest DNN to ensure the maximum possible aggregate request rate. Then, we repeat the same for the remaining clients and DNNs in descending order of DNN size (i.e., accuracy). The above process is listed in the `MapClients` function in Algorithm 1.

Algorithm 1: Client-DNN Mapping

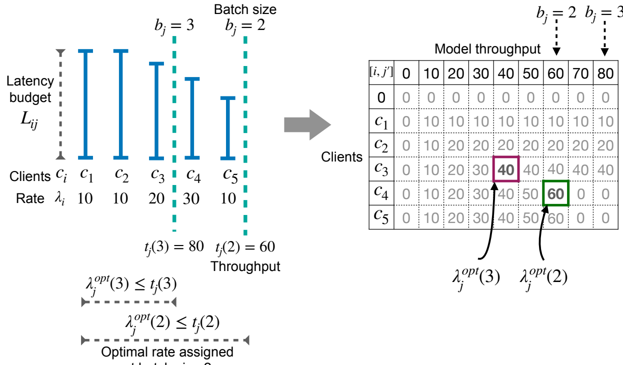
```
Function MapClients(clients, dnn_models):
1:  sort dnn_models in descending order of their accuracies
2:  map    f g
3:  for model in dnn_models do
        // break if clients is empty
4:      clients'    FindOptimalClients(model, clients)
5:      batch_size    model.checkAndAssign (clients')
6:      map.append (h model, clients', batch_size i)
7:      clients    clients    clients'
8:  return map

Function FindOptimalClients(model = hl_j, t_j i, clients):
9:  sort clients in descending order of latency (compute) budget for model
10: dp_mat    NULL
11: best_cell    (0, 0), best_value    0
12: h    GCD of all possible client rates
13: for hL_ij, λ_i i  in  clients do
14:     B_i    argmax_b (2l_j(b)    L_ij)
        // break if B_i = 0 as further clients are not satisfied
15:     K_i    floor(t_j(B_i)/h)
16:     if dp_mat = NULL then
17:         Alloc int array of size (jclientsj + 1, K_i + 1) with zeros
18:     for k = 1 to K_i do
19:         dp_mat[i, k]    dp_mat[i    1, k]
20:         w_k    k  h
21:         if λ_i    w_k then
22:             k'    (w_k    λ_i)/h
23:             v    λ_i + dp_mat[i    1, k']
24:             if v > dp_mat[i    1, k] then
25:                 dp_mat[i, k]    v
26:             if v > best_value then
27:                 best_cell    (i, k) , best_value    v
    // Perform standard backtracing from (row, col)    best_cell
       adding row into clients' list
28: return clients'
```

Now, the problem becomes *how to find a subset* of clients with the maximum possible request rate for a given DNN while meeting the SLOs of all these clients that may have diverse request rates and latency budgets. The key for solving this problem is to decide what batch size to use for the DNN as it dictates the maximum inference throughput. Using small batch sizes reduces the throughput, thus limiting the aggregate request rate; if we opt for large batch sizes to ensure enough throughput, the inference latency increases, thus challenging the SLOs of the assigned clients as specified in Eq. (5).

We observe that, given a fixed batch size, the client-DNN mapping problem reduces to a standard 0-1 knapsack problem, where we treat clients as items, the request rates of clients as weights and values, and the maximum throughput of the DNN for the given batch size as the knapsack capacity. The problem can be solved by existing algorithms, but we still need to iterate over all possible batch sizes, which is time-consuming.

Dynamic programming. We propose an efficient solution based on dynamic programming (DP) to find the optimal client-DNN mapping for a given DNN across all possible batch sizes in one shot, as listed in function `FindOptimalClients` in Algorithm 1. The idea is to enumerate all possible aggregate request rates that can be assigned to the DNN up to a maximum throughput value at the largest batch size possible and use them as columns in the DP matrix, as depicted in Fig. 5. We then recursively start computing the cell values (aggregate request



$[i,j']$	0	10	20	30	40	50	60	70	80
0	0	0	0	0	0	0	0	0	0
c_1	0	10	10	10	10	10	10	10	10
c_2	0	10	20	20	20	20	20	20	20
c_3	0	10	20	30	40	40	40	40	40
c_4	0	10	20	30	40	50	60	0	0
c_5	0	10	20	30	40	50	60	0	0

Fig. 5: An illustrative example to show the DP-based algorithm.

rate) for each row representing clients in descending order of their latency budget. For each client (row), we identify the largest batch size for which the latency constraint is satisfied and use it to identify the largest enumerated column in the DP matrix (lines $13-15$) up to which the cell values are computed, and the remaining cell values are kept zero. Line $18-27$ covers the standard DP iteration for a row (client). Finally, we perform a standard backtracing from the best cell (maximum aggregate value) to find the optimal subset of clients.

Example. Fig. 5 illustrates a simple example of mapping five clients to DNN m_j. The DNN at batch size $b_j = 3$ can satisfy the latency constraint of three clients, $c1$, $c2$ and $c3$. Whereas at batch size $b_j = 2$, the DNN can satisfy two more clients, $c4$ and $c5$. At batch size $b_j = 3$, the theoretical throughput $t_j(3)$ of the DNN is 80 and all three clients with aggregate request rate 40 can be assigned to this DNN m_j. Therefore, the optimal request rate assigned to the DNN at batch size three is 40, denoted by $\lambda_j^{opt}(3)$. However, at batch size $b_j = 2$, the theoretical throughput $t_j(2)$ of the DNN is 60. Here, multiple subsets of clients are possible, e.g., one subset is $\{c1, c2, c3, c5\}$ and another is $\{c1, c2, c4, c5\}$ with aggregate request rate of 50 and 60, respectively. Therefore, the optimal request rate assigned to the DNN at batch size two is 60, denoted by $\lambda_j^{opt}(2)$. Finally, the optimal request rate assigned to the DNN is $\max(\lambda_j^{opt}(3), \lambda_j^{opt}(2))$, i.e., 60 at batch size $b_j = 2$ with clients $\{c1, c2, c4, c5\}$.

Optimality. For a specific DNN, the DP-based solution is optimal. However, when mapping clients, multiple cells with the maximum aggregate request rate may exist in the DP table. We then choose the mapping randomly, and this might affect the optimality of the overall solution. As shown in §VI-F, our approach (together with DNN selection) is near-optimal.

Time complexity. In the worst case, the step size h (Line 12) in DNN throughput enumeration is one, and therefore, the total number of columns in the DP matrix is equal to the maximum throughput in the DNN Zoo (t_{max}). The asymptotic complexity for mapping clients to GPU workers becomes $O(|\mathbb{G}| \cdot |\mathbb{C}| \cdot t_{max})$, where $\mathbb{G}$ and $\mathbb{C}$ are the set of workers and clients.

C. DNN Selection

Once the client-DNN mapping is in place, the next question is *how to select the optimal (multi-)set of DNNs*, where the size of the set is equal to the number of workers.

Algorithm 2: DNN Selection Based on SA

Data: Client-DNN mapping function MappingAlgo, clients list $clients$, previous DNNs list $previous_models$, initial temperature T_0, min temperature T_{min}, temperature reduction ratio α

Result: near-optimal list of DNNs

1: best_models $previous_models$
2: best_mapping MappingAlgo ($clients$, best_models)
3: **for** mode in [DEGRADE, UPGRADE] **do**
4: T T_0
5: mapping best_mapping, dnn_models best_models
6: **while not** Stop(mapping, mode) **and** $T > T_{min}$ **do**
7: models^0 NeighborsGenerator (dnn_models, mode)
8: mapping^0 MappingAlgo ($clients$, models^0)
9: **if** Better(mapping^0, best_mapping, mode) **then**
10: best_mapping mapping^0, best_models models^0
11: diff mapping^0.metric mapping.metric
12: **if** diff > 0 **or exp**(diff$/T$) $>$ rand$(0, 1)$ **then**
13: mapping mapping^0, dnn_models models^0
14: T T α
15: **return** best_models

Finding the optimal set from the large space of size $\binom{|\mathbb{M}|+|\mathbb{G}|-1}{|\mathbb{G}|}$ to serve multiple clients that have varying characteristics like different SLOs, request rates, and network conditions, is nearly impractical using an exhaustive search. There are two criteria for optimality: (**O1**) the fraction of the total number of clients that can be mapped to the selected DNN set, (**O2**) the average accuracy improvement as defined in Eq. (1). To compute these metrics, we use client-DNN mapping (Algorithm 1) as a building block for every candidate DNN set. The exhaustive search thus becomes even more expensive.

Simulated annealing. We choose to use simulated annealing (SA) [15], a local search technique based on random walks that avoid being stuck in local optima when exploring the solution state space. SA accepts weak solutions with some probability defined by a parameter named *temperature T*. The acceptance probability is high initially due to the high temperature; it decreases with the decrease of the temperature.

Algorithm 2 depicts our iterative SA algorithm that performs collective DNN adaptation. We start the SA process by mapping clients (using Algorithm 1) to some previous or initial set of DNNs. In our implementation, the initial (i.e., bootstrap) set of DNNs contains the smallest-size DNN instances from the DNN zoo. Unlike in conventional SA, we have two modes of operation, namely DEGRADE and UPGRADE. We first start the DNN's exploration in DEGRADE mode, meaning we reduce the DNN size to generate the next state of neighboring DNNs. This is to first serve the minimum number of clients, for satisfying the optimality criteria **O1**. If we may repeat, the degraded DNNs have lower latency and higher throughput, therefore, improves the possibility of serving more clients. As soon as **O1** is satisfied, we switch the state (DNNs) exploration to UPGRADE mode. Here, the idea is to select mainly the larger DNNs to improve the accuracy objective (i.e., optimality criteria **O2**) without violating **O1**.

Although the SA framework has been widely used, applying them in practice is highly problem-specific due to a non-standard approach of selecting the algorithm parameters such as the neighbors' generator function, acceptance probability, stopping condition, etc. Details on how to determine the SA parameters for DNN selection are provided in the appendix of the full version of the paper [37].

D. DNN Update

Once the set of DNNs is selected for the current clients, the DNNs must be loaded onto the workers. However, loading a DNN on a GPU can incur a considerable time overhead due to the launching of CUDA kernels, transfer of DNN parameters, etc. To mitigate this issue, we prefetch DNNs on GPUs. More specifically, we employ a prefetching technique based on the nearest-neighbor policy where we pre-load DNNs neighboring the currently loaded one. When a new set of DNNs is selected, we order and match the new set to the old set such that the distance between the enumerated DNNs is minimized, so as to benefit most from prefetching. This problem is similar to the well-known stable marriage problem, and we solve it by sorting the old and new sets in decreasing order of the DNN size. We then assign the workers running DNNs from the previous set to the new DNNs in an element-by-element fashion.

V. Implementation

We implement a Jellyfish system prototype (around 4K lines of Python code) using the Pytorch framework for DNN inference on GPUs. We also provide simulation scripts to test the performance of our scheduling algorithms for different DNNs, clients, and GPU configurations. The source code is publicly available at: https://github.com/vuhpdc/jellyfish.

Hardware setup. We carry out parts of our experiments on a server equipped with an Intel Core i9-10980XE CPU (36 cores), 128GB DRAM, and two GPUs (NVIDIA RTX2080Ti), running Ubuntu 18.04. We then use another server equipped with an Intel Core i7-8700K CPU (12 cores) and 32GB DRAM to emulate multiple clients, ensuring that compute, memory, and network bandwidth are not the bottleneck. The original bandwidth between these two servers is 1Gbps. We use the Linux `tc` utility to control clients' bandwidth and replay real-world network traces. For large-scale experiments, we use AWS instances to deploy the clients and GPU workers (see §VI-E).

Software details. We expose Jellyfish service APIs through standard gRPC calls and use a bidirectional stream mechanism to handle continuous request-response client streams. Currently, the dispatcher module includes a multi-threaded gRPC server. The scheduler module runs in a separate process at a periodic interval of half a second unless specified explicitly. Each worker runs two processes: one to receive requests and load DNNs and the other as a DNN executor running with the highest priority. The communication between processes on the same machine is done through Python `SimpleQueue` (i.e., a Pipe) and PyZMQ over TCP on the distributed server. For stable and deterministic performance, we disable NVIDIA's cuDNN optimisations and control randomness with manual seed values [38]. For frame (image) compression, we use a JPEG encoding scheme with a high compression level to trade a slight degradation in analytics accuracy for speed. Furthermore, we hold all DNNs in the DNN zoo in memory to avoid disk IO overhead.

Placement of system components. On a stand-alone, multi-GPU server, the dispatcher, scheduler, and each worker run in their own processes on the same machine. Therefore, the server should have sufficient download network bandwidth, enough CPU cores to run each process on dedicated cores, sufficient DRAM to hold the DNN zoo and store incoming requests (aggregate of all clients). On distributed servers, the dispatcher and scheduler processes run on a front-end machine, whereas workers run on separate machines where GPUs are installed. The front-end machine should be a powerful server to handle connections from multiple clients. The network between front-end and worker machines should not be a bottleneck and should have predictable dispatch latency.

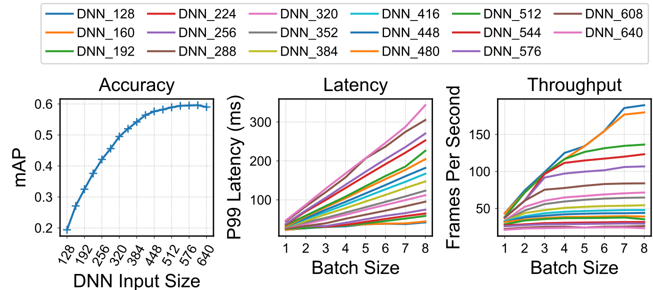


Fig. 6: The DNN performance profiles.

Bandwidth estimation. We implement a separate acknowledge mechanism for inference requests so that clients can estimate their network bandwidth per request by measuring the request input data size and the smoothed round-trip latency. The scheduler then uses the harmonic mean (following prior work [32]) of the client's bandwidth over the past one second as the estimated bandwidth for the client.

VI. Performance Evaluation

We perform extensive experiments for real-world scenarios using object detection inference tasks. We demonstrate the effectiveness of Jellyfish by answering the following questions:

Q1 Can Jellyfish fulfill its goal under variable network conditions and diverse client characteristics?

Q2 How well does Jellyfish perform compared with other DNN inference scheduling algorithms?

Q3 How well does Jellyfish perform on large-scale setups?

A. Methodology

DNN zoo. We employ a well-known pre-trained YOLOv4 DNN architecture [7] and use its Pytorch-YOLOv4 implementation [39] for the object detection task. Importantly, YOLOv4 supports different DNN input (frame) sizes by resizing the network configuration and using the same weight parameters across all resized networks. We choose 17 different DNN configurations whose input sizes (in both dimensions) range from 128 to 640 with a step size of 32, indexed from 0 to 16. We discard the DNN of size 640 as it has lower accuracy than size 608, but has higher latency for execution.

DNN profiles. We profile the DNNs (accuracy and latency) using the COCO-val2017 image dataset [40] and use the standard comparison metric called mean average precision (mAP) to rank the DNNs. Fig. 6 shows the DNN profiles used in the evaluation. From the throughput profile, we see that for the majority of the DNNs, the curve starts plateauing at around batch size 8. Furthermore, we use the 99th-percentile (P99)

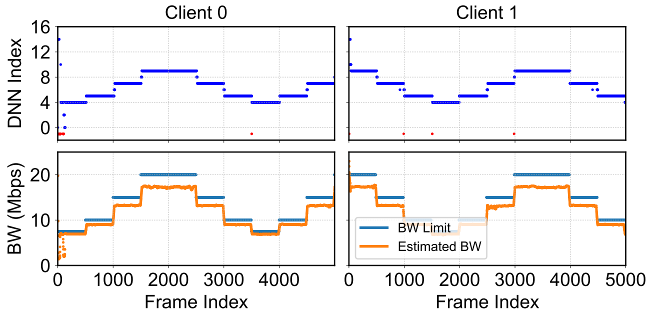


Fig. 7: DNN selection for each client in a setting (two clients, 25 FPS, 100ms SLO) under a synthetic network trace. Red dots indicate dropped frames.

latency profile to keep SLO violations low. Unlike the average latency profile, the P99 latency profile curve may not follow the non-decreasing trend (as required in our algorithms) due to high tail variations. Thus, our latency estimator adjusts the values by conservatively allocating the higher latency values of smaller DNNs to larger DNNs.

Video datasets. We evaluate our system on the vehicle detection task on highways identifying classes such as "cars", "buses", "motorbikes", and "trucks". Like in DDS [26], we pick three publicly available 10min traffic videos (around 9K frames each) at 720p resolution. We extract and replay video frames at different frame rates to generate requests for clients.

Evaluation metrics. We evaluate the system using the following performance metrics:

Miss rate: The miss rate describes the fraction of frames that have missed their SLOs or have been dropped early in the pipeline due to SLO violations.

Analytics accuracy: We use the *F1 score* (a harmonic mean of precision and recall) with IoU (intersection over union) of 0.5 as a metric to quantify analytics accuracy. We exclude missed frames as it is hard to quantify their impact on the user application. Similar to DeepDecision [24] and DDS [26], we use the detection results of the DNN whose input size is equal to that of the original video as ground truth.

Worker Utilization: The worker utilization is the fraction of the total time during which workers are busy executing the DNN inferences on GPUs.

B. End-to-End Performance

We first analyze the end-to-end performance of Jellyfish under a synthetic network trace. Following AWStream [23], we periodically set each client's bandwidth to a value from the ordered set $\{20, 15, 10, 7.5\}$Mbps and keep each value for 20 seconds. We test with $\{1, 2, 4, 8\}$ concurrent clients and draw their SLOs from the set of $\{75, 100, 150\}$ milliseconds (ms) and request rates from the set of $\{15, 25\}$ FPS. The smallest DNN in the DNN zoo has P99 latency of 23ms for batch size 1. Thus, we have a lower limit of 75ms (instead of 50ms) in the SLOs set for each client because the minimum time budget for computing on the server must be 46ms (twice the latency of the smallest DNN, see Eq. (5)). Next, we start clients sequentially with a small random wait (in $[1, 10]$s) between two clients, mimicking random client arrivals/departures and creating a random requests arrival pattern. The clients replay the same network trace but start from random points to avoid a lock-step behavior. We run each experiment for three iterations and report the mean value. Note that mostly two parallel DNNs are selected on two GPUs for serving clients at any given moment.

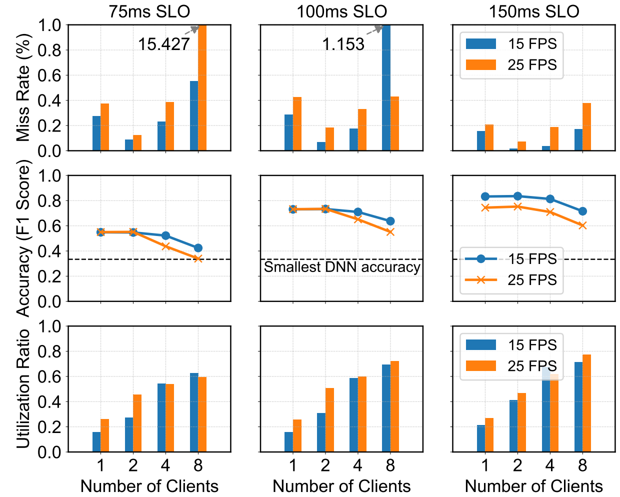


Fig. 8: The miss rate, accuracy and worker utilization of Jellyfish for varying SLOs, request rates, and numbers of clients under a synthetic network trace.

DNN adaptation. Fig. 7(bottom) depicts the estimated bandwidth values close to the actual bandwidth limits, displaying the accuracy of our bandwidth estimation. Fig. 7(top) illustrates DNN selection decisions for each client in a setting with two clients. It shows that larger DNNs are selected when the bandwidth is higher and vice-versa, implying DNN adaptation.

Miss rate. Fig. 8(top) shows that the overall miss rate is less than 1% for almost all settings. We make three observations: **(a)** The miss rate for settings with 150ms SLO is the lowest due to the high compute time budget available on the edge server. **(b)** The miss rate is relatively high (1.153%) for the setting with 100ms SLO, 15 FPS, and 8 clients. Here, the scheduler often selects two DNNs: a small one with a moderate batch size (e.g., DNN index 3 with batch size 3, throughput around 90 FPS) and a large one with a small batch size (e.g., DNN index 9 with batch size 1, throughput around 32 FPS). Hence, many clients (e.g., around 6 with an aggregate request rate of 90 FPS) are served by the small DNN. Thus, the small DNN is heavily loaded and the requests served by this DNN are more sensitive to the micro-bursts of requests created by nonuniform request arrival. **(c)** The miss rate is unacceptable (15.427%) for the setting with 75ms SLO, 25 FPS, and 8 clients, which represents an overloaded situation. To support the aggregate request rate of 200 FPS of 8 clients with 75ms SLO, the scheduler must select the smallest DNN on each GPU with batch size 4 (the DNN latency and throughput being 32ms and 124 FPS, respectively). That means the clients require a time budget of at least 64ms for computing, which is impossible when the client's bandwidth is low (i.e., 7.5Mbps). Thus, the scheduler often selects the batch size of 4 and 3 on each GPU, with the total inference throughput being slightly lower than the aggregate request rate, leaving one client *unmapped* to any

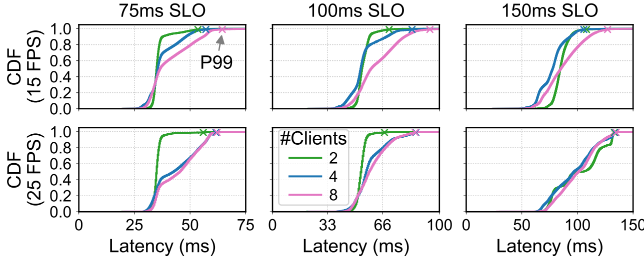


Fig. 9: End-to-end latency CDF for varying SLOs, request rates, and numbers of clients under a synthetic network trace.

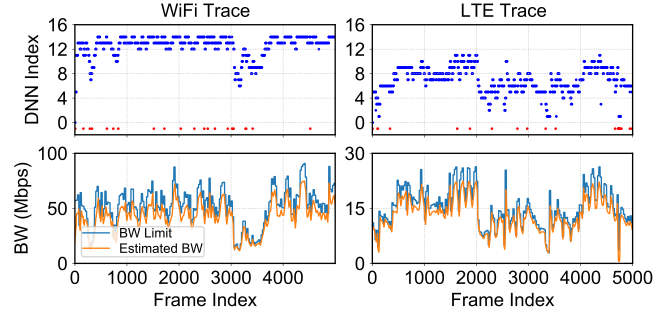


Fig. 10: Illustration of DNN adaptation for a client in a setting (2 clients, 25 FPS, 100ms SLO) with WiFi and LTE traces. Red dots mark dropped frames.

of the DNNs. Overall, *Jellyfish delivers an extremely low miss rate ($\leq 1\%$) when the system is not overloaded.*

Analytics accuracy. Fig. 8(middle) shows the accuracy of all settings. The settings with one or two clients have similar accuracy since requests are served with similar DNNs. The accuracy decreases when the aggregate request rate increases. Here, the aggregate request rate can increase when the number of clients or their frame rate increases. In this case, the scheduler has to lower the DNNs sizes to support the higher request rates. However, with larger SLOs, the scheduler can select larger DNNs when possible. Consequently, the accuracy at 150ms SLO for all settings is higher than that at 100ms or 75ms SLOs. Overall, for all settings, the accuracy achieved by Jellyfish is much higher than that achieved by the smallest DNN (i.e., the DNN likely to be deployed directly on client devices), *demonstrating the benefits of offloading inference tasks to the edge server, albeit dynamic networks*.

Worker utilization. Fig. 8(bottom) shows the aggregate worker utilization ratio. The utilization is lower for settings with fewer clients and lower FPS due to lower aggregate request rates and larger arrival times between requests. Once the system becomes more saturated with more clients and higher SLOs, the utilization increases (up to 75%) because Jellyfish tends to select larger batch sizes and DNNs, thus increasing the compute usage. The experiment confirms that *Jellyfish's low miss rates are not at the cost of reduced worker utilization*.

End-to-end latency. Fig. 9 shows the end-to-end latency CDF for all settings except for a setting with 1 client, which performs similarly to the setting with 2 clients. We see that the median latency increases for all clients when the SLO increases as the scheduler selects larger DNNs. For example, the median latency is 53.77ms with two clients, 15 FPS, and 100ms SLO, whereas it is 84.62ms with 15 FPS and 150ms SLO. Although the median latency is much lower (queuing time is assumed equal to the DNN inference time), the P99 latency is close to the SLO, especially for settings with many clients where the queuing time is high. This confirms *the importance of using higher (e.g., P99) DNN latency profiles and the assumption of worst-case queuing time for low miss rates.*

In a nutshell, **Jellyfish can fulfill the goal of delivering low miss rates while maintaining high accuracy (Q1).**

C. Comparison with the Baselines

Baseline. Inspired by Clockwork [11], we implement a fine-grained baseline scheduler based on the *earliest deadline first* (EDF) policy. The idea is to deploy a static DNN on all GPUs and schedule requests with the earliest deadline without preemption on the next available GPU worker. Similar to Clockwork, for batching requests adaptively, we maintain a global priority queue per batch size where new requests are added to every batch queue. The priority of a request in a batch queue is determined by the earliest time to schedule the request at the respective batch size. We then schedule requests from each batch queue with a sufficient number of requests by iterating through batch queues in the decreasing order of batch size. The requests are dropped from the particular batch queue when the time budget is insufficient for DNN execution. Here, the system design and implementation are similar to our setup except for the scheduling logic, as we focus our comparison on scheduling algorithms. More importantly, our network time estimation helps to compute the variable *network time* and the variable *compute time budget* per request on the server, which is then used as a *deadline* for the EDF policy.

Further, we enable *data adaptation* on clients. The adaptation policy picks the maximum possible frame resolution below the input size of the chosen DNN to maintain a stable network throughput. This policy is similar to AWStream [23] (for the frame resolution) and offers optimal adaptation because the DNN accuracy is a monotonic function of the frame resolution (see Fig. 6).

We compare Jellyfish with three variants of the baseline by deploying the lowest (B_L), middle (B_M), and highest (B_H) DNNs in terms of accuracy (also size). Similar to §VI-B, we test around 18 experimental settings with a combination of {2, 4, 8} concurrent clients, {75, 100, 150}ms SLO and {15, 25} FPS. Along with a synthetic network trace, we also compare the performance on two real-world network traces: a WiFi network trace and a 4G/LTE downlink bandwidth trace [41] downscaled by a factor of two to represent the uplink bandwidth [13]. Fig. 10(bottom) shows the estimated and actual bandwidth values. Fig. 10(top) shows the DNN selection decisions for one client under the two real-world network traces, indicating that Jellyfish adapts *quickly* to bandwidth changes.

Results and discussion. Fig. 11 shows the performance of Jellyfish against the baselines under three different network traces. Jellyfish *consistently* has the overall miss rates below 1% for the synthetic and WiFi trace and below 1.5% for the LTE trace except for one setting (75ms SLO, 25 FPS, and 8

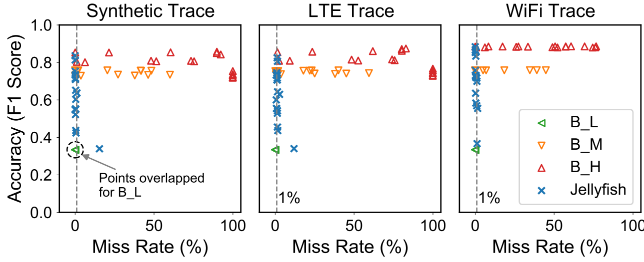


Fig. 11: Comparison of Jellyfish with baselines. B_M and B_H have excessive SLO violations making them ineffective, while B_L suffers from low accuracy.

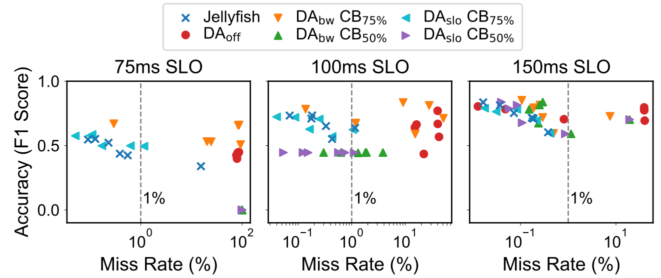


Fig. 12: The impact of the three data adaptation strategies on Jellyfish's performance under a synthetic network trace. The label DA means data adaptation, and CB$_{x\%}$ means $x\%$ of SLO allocated as a compute time budget. The x-axis is in log scale.

clients) where at least one client cannot be mapped to any DNN (explained in §VI-B). Jellyfish achieves decent accuracy for settings with large SLOs and low aggregate request rates under the WiFi trace, on par with B_H. This is because the bandwidth values are generally high (median value 53.1Mbps) under WiFi, leaving a large compute time budget on the server. For B_L, the network is never a bottleneck because the smallest frame size, 128×128 at 25 FPS, needs only 2Mbps bandwidth. Besides, two B_L instances can support the aggregate request rate in all settings. Thus, *the miss rates for B_L are negligible but at the cost of the lowest accuracy*. B_H is the worst in terms of inference throughput and bandwidth requirement (26Mbps). Hence, *B_H has the highest miss rate for almost all settings*.

B_M has better performance than other baselines but fails to provide consistency like Jellyfish does for all settings. B_M has high miss rates in the following two cases: **(a) Low SLOs and low request rates:** Clients need around 7Mbps to send frames at the desired size (354×354) and 15 FPS. Clients do not face any network bottleneck, especially under synthetic and WiFi traces, and thus can always send frames at the desired size. Yet, sending at the desired frame size results in significant network time (up to 60ms), leaving a very small compute time budget on the server, especially when the bandwidth drops below 10Mbps. Hence, the miss rates are around 40%, *indicating the necessity of aligning the data and DNN adaptation decisions*. On the other hand, clients sending at 25 FPS need about 11Mbps, and therefore, clients would lower frame sizes (data adaptation) to maintain stable network throughput. Due to the data adaptation, the network time is significantly reduced, leaving enough compute time budget on the server for the inference. **(b) High aggregate request rates:** The scheduler has to increase the batch size to support many clients (or their high aggregate frame rates), but at increased compute time, which hurts settings without sufficient SLOs. Therefore, B_M has high miss rates for 4 and 8 clients with SLOs under 150ms.

Furthermore, as we consider the F1 score of only the processed requests, the accuracy of B_M and B_H is higher than Jellyfish in some settings but at the cost of *extremely high* miss rates. Note that the gap in accuracy between Jellyfish and baselines B_M and B_H decreases when the SLO increases as the scheduler tends to select larger DNNs.

Heterogeneous clients. We also experiment with heterogeneous clients, i.e., clients with varying combinations of request rates (FPS) and SLOs in one setting. Under the LTE trace, the baseline B_M has a miss rate of 11.78% for 4 clients and 31.94% for 8 clients. In contrast, Jellyfish has a miss rate of 0.95% for 4 clients and 1.62% for 8 clients, in line with the results in Fig. 11 for homogeneous clients. Similar results hold for the synthetic and WiFi network traces (see Appendix C in the full version [37]).

In summary, **Jellyfish consistently outperforms baselines in terms of miss rates and maximizes the accuracy whenever a larger compute time budget is available (Q2).**

D. Performance of Joint Adaptation

In §VI-C, we see that the miss rates are significantly higher for the baselines that do not perform DNN adaptation, even when using data adaptation. We now investigate the impact of joint adaptation, i.e., the combination of data and DNN adaptation. To this end, we enable or disable the two system adaptation components independently and analyze the impact of each combination on the overall Jellyfish performance. For the data adaptation, we further consider three scenarios for which we provide modified implementations:

No data adaptation (DA$_{off}$), i.e., simply streaming data from clients at a predefined fixed size. Specifically, we choose the input size of the middle DNN, i.e., 354×354, which provides a good trade-off between bandwidth requirement and accuracy. Here, the scheduler knows the data size and treats it as a constant during DNN adaptation.

Default data adaptation (DA$_{bw}$) with typical network bandwidth awareness to maintain stable network throughput [23]. Here, the current network condition is considered but no knowledge about the DNN adaptation component is provided. In this scenario, we have to *statically* allocate some percentage of the end-to-end SLO as a *compute* time budget for the DNN adaptation. For our experiments, we choose 50% and 75% heuristically. We cannot allocate 25% of SLO as a compute time budget because no DNNs are feasible to execute for the 75ms and 100ms SLO settings.

SLO-aware data adaptation (DA$_{slo}$) that optimizes the data adaptation strategy to also consider the network time budget. Here, the data adaptation is aware that a part of the end-to-end SLO has been *statically* allocated for the DNN adaptation. Hence, it attempts to deliver the data to the server in the remaining time to achieve low miss rates considering the network time budget in addition to the current network bandwidth.

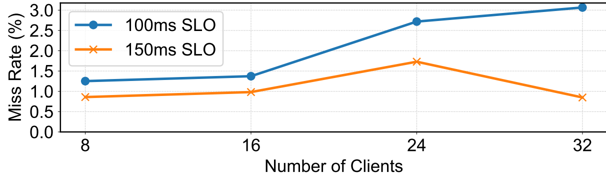
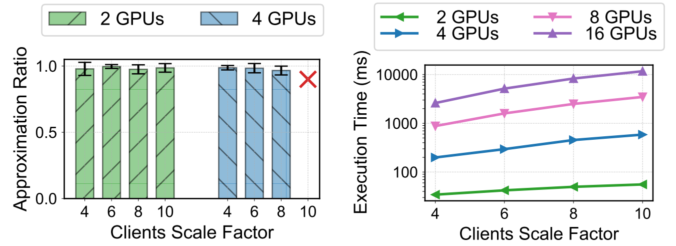


Fig. 13: The miss rate on 8 GPUs for varying numbers of clients operating at 15 FPS with the dynamic LTE trace.



(a) Approximation ratio (mean)

(b) Execution time in log scale

Fig. 14: Performance of Jellyfish scheduler for varying #GPUs and #clients. Here, the #clients is a product of Clients Scale Factor (x-axis) and #GPUs.

Similar to §VI-B, we use 18 experimental settings on synthetic network trace for performance comparison.

Results and discussion. We show the results in Fig. 12. **(a)** For no data adaptation (DA_{off}), the miss rates are extremely high in almost all settings as expected. **(b)** For default data adaptation (DA_{bw}), the miss rates are lower than DA_{off} for settings with higher SLOs. However, compared to Jellyfish, DA_{bw} still has higher miss rates and lower accuracy, especially for settings with lower SLOs (75ms and 100ms). Note that when the compute time budget is 50% of the SLO, no DNN is selected for the 75ms SLO which results in a 100% miss rate. **(c)** For SLO-aware data adaptation (DA_{slo}), the miss rates are comparable to Jellyfish, but the accuracy is significantly lower for a compute time budget of 50% of the end-to-end SLO ($CB_{50\%}$). The accuracy of DA_{slo} is on par with Jellyfish for a compute time budget of 75% of the SLO ($CB_{75\%}$). In the case of DA_{slo} and $CB_{75\%}$, the frames are streamed at a lower resolution (due to a low network time budget) and upscaled on the server for serving with bigger DNNs. While the task we consider in the experiments (vehicle detection) is not obviously sensitive to quality degradation from frame upscaling, that behaviour may not hold for other tasks (e.g., semantic segmentation), DNN architectures, and data content [25]. Furthermore, the accuracy depends on the *manual* selection of a *static* budget allocation (50% or 75%) between data and DNN adaptation, and the optimal value can be hard to decide in practice. Jellyfish *automatically* and *dynamically* allocates the time budget between data and DNN adaptation.

In summary, joint adaptation is crucial for achieving low miss rates with optimal accuracy—**Jellyfish's dynamic allocation of time budget between data and DNN adaptation and alignment of adaptation decisions allow for a consistently high performance without manual system configurations.**

E. Large-Scale Setup

We also evaluate Jellyfish on a large-scale distributed cloud setup. Specifically, we run the dispatcher on an AWS compute instance `c5.9xlarge`, 8 workers on 8 `g4dn.2xlarge` instances equipped with T4 NVIDIA GPUs and 8 `t3.2xlarge` instances to emulate up to 32 clients. Here, we test Jellyfish with varying numbers of clients for {100, 150}ms SLOs and 15 FPS on the LTE trace. We choose an FPS of 15 to support a large number of clients without introducing a throughput bottleneck on the server and to offer enough leeway for DNN adaptation. The latency profile patterns remain proportional to the one in Fig. 6. We use only the smallest ten DNNs since larger DNNs have much lower throughput, making them inefficient in this setup. Note that T4 GPUs have a low power limit of 70W. Therefore, even after fixing the clock values, power throttling leads to rather unstable inference timings, which can negatively affect the performance of Jellyfish.

Fig. 13 shows that the miss rates are less than 1.73% for an SLO of 150ms and 3% for 100ms. For 100ms SLO and 32 clients, the scheduler selects relatively smaller DNNs than 24 clients to support the aggregate request rate (480 FPS). The scheduler may then assign many clients to the smaller DNNs. As mentioned in §VI-B, many clients assigned to the same DNN might distort the uniformity of the request arrival pattern and thus lead to increased request misses when the inference timings are unstable. However, the miss rate improves with the increase of the SLO (e.g., 150ms), due to increased compute time budget that can mask the unstable timings. We observe no particular trend in the miss rate when the number of clients increases as the miss rate depends on the complex dynamics of client characteristics and DNN performance profiles. Overall, **Jellyfish achieves miss rates within the acceptable range ($1-3\%$), even on a large-scale setup (Q3).**

F. Scheduler Performance

We evaluate the Jellyfish scheduler performance through simulations, comparing it with the optimal MILP algorithm. We run the algorithms with multiple settings spanning {2, 4, 8, 16} GPUs and the number of clients with a factor of {4, 6, 8, 10} times the number of GPUs. Each client randomly draws its request rate from {10, 15, 25} and SLO from {75, 100, 150}ms and its bandwidth is chosen uniformly at random from the interval [7.5, 50) Mbps. We use the same DNN profiles as depicted in Fig. 6. We run around 100 problem instances for each setting. The solution quality of each algorithm is measured by the accuracy objective defined in Eq. 1. We then use the approximation ratio between our algorithm and the MILP algorithm as a comparative metric.

Approximation ratio. Fig. 14a shows the mean approximation ratio for 2 and 4 GPUs. The MILP algorithm could not return enough optimal solutions for settings with {8, 16} GPUs and 40 clients on 4 GPUs, even after specifying the time limit of 30 minutes for each problem instance. It can be observed that *our scheduling algorithm is near-optimal*, with an approximation ratio ranging from 0.966 to 0.996.

Execution time. As depicted in Fig. 14b, our naive Python implementation of the Jellyfish scheduler has a sub-second execution time for up to 8 GPUs and clients scale factor of 6. With the increase of the GPUs and the clients scale factor, the

execution time increases almost linearly. Overall, *it is practical to run our scheduler at a high frequency for handling high network dynamics in typical edge scenarios.*

G. DNN Prefetching Performance

We also analyze the effectiveness of the DNN prefetching strategy. We consider the same settings under the two real-world network traces, where DNNs must be adapted more often to handle frequently changing bandwidth. In this case, the DNN hit ratio is around 92.37% when five DNNs (out of 16) and 83.61% when only three DNNs are prefetched at a time. On our setup, such a hit ratio translates to a maximum gain of 3% in processing requests precisely with the newly selected DNN. The gain is not high due to the minimal cost of moving DNNs on our GPU setup (150-200ms). However, we anticipate the gain to be substantial for large state-of-the-art DNNs. *The high hit ratio confirms the effectiveness of the nearest-neighbor prefetching and our DNN update method.*

VII. Related Work

Adaptive video analytics systems. Recent works such as VideoStorm [42], AWStream [23], Chameleon [43], DeepDecision [24], JCAB [34], DDS [26], and SPINN [30] have proposed adaptive solutions for networked video analytics. Their main goal is to schedule bandwidth efficiently or save energy by means of trading accuracy for resource efficiency. However, meeting latency SLOs in an end-to-end fashion has not been the main goal or even considered. Data adaptation is applied in DeepDecision and JCAB, with theoretical frameworks for adapting input video configurations (such as frame resolution and rate). Although JCAB considers a multi-client scenario (despite simulation-based evaluation), none of them consider the multi-client, multi-GPU serving scenario for a holistic DNN adaptation. The problem of resource allocation and workload partitioning between multiple clients (smart cameras) and an edge cluster in video surveillance systems has been addressed by Distream [44]. Unlike Jellyfish, however, Distream does not account for variable edge network conditions and millisecond-level SLOs, thus limiting its applicability for the highly dynamic scenarios we consider in this paper.

Inference serving systems. Clipper [9] provides an easy-to-use abstraction layer for low-level deep learning frameworks. Nexus [10] aims to optimize serving throughput without SLO violations. Clockwork [11] leverages the predictable performance of the DNNs, considers the SLO guarantees on the server, and maps requests to the desired model, but does not utilize DNN adaptation. Inferline [45], Llama [46], and FA2 [47] optimize the serving of complex DNN pipelines. INFaas [12] automates the hardware and model-variant selection and deployment through managed services. Model-Switching [33] proposes to scale DNNs (up and down) instead of scaling resources in the case of fluctuating workload. None of these cloud-based solutions consider the impact of the dynamic edge network on the end-to-end latency. These serving systems do not consider the client conditions and perform client data adaptation to reduce network transmission time and effectively increase the compute time budget on the server-side. Further, since many of these serving systems are designed for different objectives (e.g., resource optimization), it is non-trivial to incorporate network variation, data/DNN adaptation dependencies, and collective adaptation in them without fundamental changes.

While enterprise-grade serving systems such as TensorFlow Serving [48], Torch Serve [49], and Triton Inference Server [50] support best-effort inference batching, they do not have latency guarantees as a first-class service feature, let alone considering client network conditions. Integrating our scheduler logic into these systems is an interesting direction for future work. Jellyfish bridges the gap between adaptive video analytics systems and inference serving systems.

Joint adaptation. Recent works have also argued for joint data and DNN adaptation. However, they either focus on a single-client setup [27] or optimize resources with relatively lenient latency constraints (i.e., 1–5s) [28]. In contrast, Jellyfish maximizes inference accuracy with *millisecond-level* latency SLO targets given a highly dynamic network.

VIII. Discussion and Limitations

Request rate adaptation. Similar to Chameleon [43] and DeepDecision [24], Jellyfish does not adapt the request (frame) rate and we consider it as future work. The plan is to decouple the request rate adaptation decision from the server-side scheduling and leave the decision up to the client. Such an approach may help with Jellyfish scalability.

Predictability. Generally, we assume that DNN inference latency is predictable and invariably remains stable. Yet, in practice, especially on commodity hardware and software, it is hard to maintain stable performance without having a detailed understanding of the system's internals. We expect the service providers to tune the system in favor of stability than speed.

Latency budget estimation. Our latency (compute) budget estimation currently depends on predicting accurately the client's bandwidth and the data size of the video frames. With image encoding such as JPEG and PNG, the compressed size depends on the changing content of the image, which affects the estimation of network time. We plan to explore the more advanced bandwidth estimation techniques and frame/video compression scheme with a constant compression ratio.

IX. Conclusion

Jellyfish is an edge-centric DL inference serving system that provides soft guarantees for end-to-end latency SLOs specified over the variable network transmission and DNN inference time. Jellyfish employs efficient algorithms for client-DNN mapping and DNN selection, enabling collective system adaptation by aligning data and DNN adaptation decisions and coordinating adaptation decisions for multiple clients. Our evaluation based on a system prototype with real inference tasks and real-world network traces confirms that Jellyfish *consistently* achieves *extremely* low latency SLO violations while maintaining high accuracy.

X. Acknowledgement

We would like to thank the anonymous reviewers for their valuable comments and suggestions. We would also like to thank Guilherme Henrique Apostolo for proofreading the paper. This work is part of the Efficient Deep Learning (EDL) programme (grant number P16-25), financed by the Dutch Research Council (NWO).

References

[1] L. Liu, H. Li, and M. Gruteser, "Edge assisted real-time object detection for mobile augmented reality," in *ACM MobiCom*, 2019, pp. 25:1–25:16.

[2] M. Braun, A. Mainz, R. Chadowitz, B. Pfleging, and F. Alt, "At your service: Designing voice assistant personalities to improve automotive user interfaces," in *ACM CHI*, 2019, p. 40.

[3] A. J. B. Ali, Z. S. Hashemifar, and K. Dantu, "Edge-slam: edge-assisted visual simultaneous localization and mapping," in *ACM MobiSys*, 2020, pp. 325–337.

[4] F. Ahmad, H. Qiu, R. Eells, F. Bai, and R. Govindan, "Carmap: Fast 3d feature map updates for automobiles," in *USENIX NSDI*, 2020, pp. 1063–1081.

[5] G. Ananthanarayanan, P. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha, "Real-time video analytics: The killer app for edge computing," *Computer*, vol. 50, no. 10, pp. 58–67, 2017.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE CVPR*, 2016, pp. 770–778.

[7] A. Bochkovskiy, C. Wang, and H. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," *arXiv*, 2020.

[8] R. Bhardwaj, Z. Xia, G. Ananthanarayanan, J. Jiang, N. Karianakis, Y. Shu, K. Hsieh, V. Bahl, and I. Stoica, "Ekya: Continuous learning of video analytics models on edge compute servers," in *USENIX NSDI*, 2022.

[9] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *USENIX NSDI*, 2017, pp. 613–627.

[10] 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 *ACM SOSP*, 2019, pp. 322–337.

[11] 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 *USENIX OSDI*, 2020, pp. 443–462.

[12] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "INFaaS: Automated model-less inference serving," in *USENIX ATC*, 2021, pp. 397–411.

[13] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck, "A close examination of performance and power characteristics of 4g LTE networks," in *ACM MobiSys*, 2012, pp. 225–238.

[14] D. Xu, A. Zhou, X. Zhang, G. Wang, X. Liu, C. An, Y. Shi, L. Liu, and H. Ma, "Understanding operational 5G: A first measurement study on its coverage, performance and energy consumption," in *ACM SIGCOMM*, 2020, pp. 479–494.

[15] E. H. L. Aarts and J. H. M. Korst, *Simulated annealing and Boltzmann machines - a stochastic approach to combinatorial optimization and neural computing*. Wiley, 1990.

[16] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, Inception-ResNet and the impact of residual connections on learning," in *AAAI*, 2017, pp. 4278–4284.

[17] Z. Chen, W. Hu, J. Wang, S. Zhao, B. Amos, G. Wu, K. Ha, K. Elgazzar, P. Pillai, R. L. Klatzky, D. P. Siewiorek, and M. Satyanarayanan, "An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance," in *ACM/IEEE SEC*, 2017, pp. 14:1–14:14.

[18] X. Zhang, H. Lu, C. Hao, J. Li, B. Cheng, Y. Li, K. Rupnow, J. Xiong, T. S. Huang, H. Shi, W. W. Hwu, and D. Chen, "SkyNet: a hardware-efficient method for object detection and tracking on embedded systems," in *MLSys*, 2020.

[19] M. Rusci, A. Capotondi, and L. Benini, "Memory-driven mixed low precision quantization for enabling deep network inference on microcontrollers," in *MLSys*, 2020.

[20] C. Wan, M. H. Santriaji, E. Rogers, H. Hoffmann, M. Maire, and S. Lu, "ALERT: accurate learning for energy and timeliness," in *USENIX ATC*, 2020, pp. 353–369.

[21] S. Lee and S. Nirjon, "SubFlow: A dynamic induced-subgraph strategy toward real-time DNN inference and training," in *IEEE RTAS*, 2020, pp. 15–29.

[22] T. Kannan and H. Hoffmann, "Budget rnns: Multi-capacity neural networks to improve in-sensor inference under energy budgets," in *IEEE RTAS*, 2021, pp. 143–156.

[23] B. Zhang, X. Jin, S. Ratnasamy, J. Wawrzynek, and E. A. Lee, "AWStream: adaptive wide-area streaming analytics," in *ACM SIGCOMM*, 2018, pp. 236–252.

[24] X. Ran, H. Chen, X. Zhu, Z. Liu, and J. Chen, "DeepDecision: A mobile deep learning framework for edge video analytics," in *IEEE INFOCOM*, 2018, pp. 1421–1429.

[25] D. Dai, Y. Wang, Y. Chen, and L. V. Gool, "Is image super-resolution helpful for other vision tasks?" in *IEEE WACV*, 2016, pp. 1–9.

[26] K. Du, A. Pervaiz, X. Yuan, A. Chowdhery, Q. Zhang, H. Hoffmann, and J. Jiang, "Server-driven video streaming for deep learning inference," in *ACM SIGCOMM*, 2020, pp. 557–570.

[27] V. Nigade, R. Winder, H. E. Bal, and L. Wang, "Better never than late: Timely edge video analytics over the air," in *ACM SenSys*, 2021, pp. 426–432.

[28] J. Jiang, Z. Luo, C. Hu, Z. He, Z. Wang, S. Xia, and C. Wu, "Joint model and data adaptation for cloud inference serving," in *IEEE RTSS*, 2021, pp. 279–289.

[29] 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 *ACM MobiSys*, 2016, pp. 123–136.

[30] S. Laskaridis, S. I. Venieris, M. Almeida, I. Leontiadis, and N. D. Lane, "SPINN: synergistic progressive inference of neural networks over device and cloud," in *ACM MobiCom*, 2020, pp. 37:1–37:15.

[31] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han, "Once-for-All: Train one network and specialize it for efficient deployment," in *ICLR*, 2020.

[32] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli, "A control-theoretic approach for dynamic adaptive video streaming over HTTP," in *ACM SIGCOMM*, 2015, pp. 325–338.

[33] J. Zhang, S. Elnikety, S. Zarar, A. Gupta, and S. Garg, "Model-switching: Dealing with fluctuating workloads in machine-learning-as-a-service systems," in *USENIX HotCloud*, 2020.

[34] C. Wang, S. Zhang, Y. Chen, Z. Qian, J. Wu, and M. Xiao, "Joint configuration adaptation and bandwidth allocation for edge-based real-time video analytics," in *IEEE INFOCOM*, 2020, pp. 257–266.

[35] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang, "GrandSLAm: Guaranteeing slas for jobs in microservices execution frameworks," in *ACM EuroSys*, 2019, pp. 34:1–34:16.

[36] F. Yu, D. Wang, L. Shangguan, M. Zhang, C. Liu, and X. Chen, "A survey of multi-tenant deep learning inference on GPU," *arXiv*, vol. abs/2203.09040, 2022.

[37] V. Nigade, P. Bauszat, H. Bal, and L. Wang, "Jellyfish: Timely inference serving for dynamic edge networks (extended version with appendix)," https://research.vu.nl/en/publications/jellyfish-timely-inference-serving-for-dynamic-edge-networks, 2022.

[38] PyTorch, "Reproducibility," https://pytorch.org/docs/stable/notes/randomness.html, (Accessed on Jan 31, 2022).

[39] Tianxiaomo, "Pytorch-yolov4," https://github.com/Tianxiaomo/pytorch-YOLOv4, 2020.

[40] T. Lin, M. Maire, S. J. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: common objects in context," in *ECCV*, vol. 8693, 2014, pp. 740–755.

[41] J. van der Hooft, S. Petrangeli, T. Wauters, R. Huysegems, P. Rondao-Alface, T. Bostoen, and F. D. Turck, "Http/2-based adaptive streaming of HEVC video over 4g/lte networks," *IEEE Commun. Lett.*, vol. 20, no. 11, pp. 2177–2180, 2016.

[42] H. Zhang, G. Ananthanarayanan, P. Bodík, M. Philipose, P. Bahl, and M. J. Freedman, "Live video analytics at scale with approximation and delay-tolerance," in *USENIX NSDI*, 2017, pp. 377–392.

[43] J. Jiang, G. Ananthanarayanan, P. Bodík, S. Sen, and I. Stoica, "Chameleon: scalable adaptation of video analytics," in *ACM SIGCOMM*, 2018, pp. 253–266.

[44] X. Zeng, B. Fang, H. Shen, and M. Zhang, "Distream: scaling live video analytics with workload-adaptive distributed edge intelligence," in *ACM SenSys*, 2020, pp. 409–421.

[45] D. Crankshaw, G. Sela, X. Mo, C. Zumar, I. Stoica, J. Gonzalez, and A. Tumanov, "InferLine: latency-aware provisioning and scaling for prediction serving pipelines," in *ACM SoCC*, 2020, pp. 477–491.

[46] F. Romero, M. Zhao, N. J. Yadwadkar, and C. Kozyrakis, “Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines,” *arXiv*, 2021.
[47] K. Razavi, M. Luthra, B. Koldehofe, M. Mühlhäuser, and L. Wang, “FA2: Fast, accurate autoscaling for serving deep learning inference with SLA guarantees,” in *IEEE RTAS*, 2022, pp. 146–159.
[48] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke, “Tensorflow-serving: Flexible, high-performance ML serving,” 2017.
[49] Pytorch, “TorchServe,” https://pytorch.org/serve/, (Accessed on October 06, 2021).
[50] NVIDIA, “NVIDIA Triton Inference Server,” https://developer.nvidia.com/nvidia-triton-inference-server, (Accessed on October 06, 2021).