Real-Time Systems
https://doi.org/10.1007/s11241-024-09418-4

Inference serving with end-to-end latency SLOs over dynamic edge networks

Vinod Nigade[1] · **Pablo Bauszat**[1] · **Henri Bal**[1] · **Lin Wang**[1]

Accepted: 25 October 2023

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 for end-to-end inference latency service-level objectives (SLO). Jellyfish handles the network variability by utilizing 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 continuously map users and adapt DNNs at runtime, so that we fulfill latency SLOs while maximizing the overall inference accuracy. We further investigate *dynamic* DNNs, i.e., DNNs that encompass multiple architecture variants, and demonstrate their potential benefit through preliminary experiments. 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.

Keywords Inference serving · DNN adaptation · Data adaptation · Dynamic edge networks · Dynamic DNNs

1 Introduction

In the past decade, modern applications such as augmented reality, intelligent personal assistants, and autonomous driving (Liu et al. 2019; Braun et al. 2019; Ali et al. 2020; Ahmad et al. 2020) have proliferated. A considerable number of these applications are based on deep learning (DL) inference, e.g., analyzing continuous

✉ Vinod Nigade
v.v.nigade@vu.nl

[1] Vrije Universiteit Amsterdam, Amsterdam, The Netherlands

Published online: 06 February 2024

video streams to understand the environment with pretrained deep neural networks (DNNs) (Ananthanarayanan et al. 2017). Employing sophisticated learning techniques (He et al. 2016; Bochkovskiy et al. 2020), these DNNs typically demand intensive computations, making them hard to deploy on mobile and IoT devices due to the limited capability of these devices. Ongoing research efforts enable the deployment of large DNNs on end devices with limited capabilities through model compression (e.g., using quantization (Zhou et al. 2018), model pruning (Zhang et al. 2018b), or knowledge distillation (Hinton et al. 2015)). Despite recent advances, compressed DNNs still experience significant accuracy loss and require meticulous tweaks (Cheng et al. 2018; Wang et al. 2019). In addition, the compressed DNNs are challenging to fit on end devices such as micro-controllers with only a few kilobytes of memory and low-power consumption, severely limiting the working set size for storing DNN parameters and inference latency (Svoboda et al. 2022). 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 (Bhardwaj et al. 2022).

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 (Crankshaw et al. 2017), Nexus (Shen et al. 2019), Clockwork (Gujarati et al. 2020), and INFaas (Romero et al. 2021a). The general goal is to achieve resource efficiency and/or guarantee inference latency (e.g., serving requests within 100ms (Gujarati et al. 2020)), 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 IoT devices need to travel through a (wireless) network before they arrive at the edge server. Such a network typically shows high performance variability (Huang et al. 2012; Xu et al. 2020), 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 zoo (collection) of DNNs with

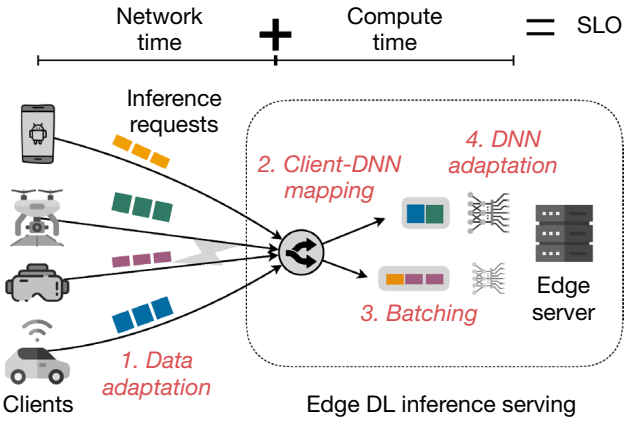


Fig. 1 Collective DNN adaptation for timely edge DL inference serving

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 (Crankshaw et al. 2017; Shen et al. 2019). 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[1]), 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 (Aarts and Korst 1990). 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.

Jellyfish requires a zoo (collection) of DNN variants with different accuracy-latency tradeoff profiles to perform DNN adaptation. Commonly, a zoo of DNNs is generated using a collection of *static* DNN models that have a fixed computational graph and parameters during inference (often referred to as the *bag-of-models* approach). Alternatively, we can employ a single *dynamic* DNN that has a dynamic computational graph that enables partial execution and virtually embeds multiple DNN variants in one base DNN. Each of these strategies has its advantages and disadvantages. We can easily construct static

[1] The DNN instance represents one instance of a particular DNN variant selected by the Jellyfish scheduler from the zoo of DNNs to deploy it on one GPU for inference.

DNN variants by downloading different pretrained models or by training different architectures separately with standard methodologies. However, the static approach introduces DNN adaptation overhead during runtime (further explained in Sect. 2.4). In contrast, dynamic DNNs incur negligible adaptation overhead and enable on-the-fly adaptation. However, designing and training dynamic DNNs is challenging, especially for real-world vision tasks like object detection. In this paper, we propose a greedy instance prefetching strategy to reduce DNN adaptation overhead when using static DNNs. We further explore dynamic DNNs as an alternative to the bag-of-models approach for inference serving.

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.
4. We present ideas for optimizing the DNN execution to avoid DNN switching (adaptation) costs on GPUs and enable highly efficient execution for batched inferences by leveraging dynamic DNNs. Furthermore, we integrate dynamic DNNs in Jellyfish and conduct preliminary experiments to demonstrate the effectiveness of dynamic DNNs in inference serving systems.

Paper structure The rest of the paper is structured as follows: Sect. 2 introduces the background and motivates our work. Section 3 presents the overall system design. Section 4 describes the formulation of the collective DNN adaptation problem and our proposed scheduling algorithms. Section 5 presents a design sketch for leveraging dynamic DNNs to efficiently switch between DNNs and perform batched inferences. Section 6 discusses our implementation in detail. Section 7 contains the evaluation of Jellyfish and a discussion of the results. Section 8 presents the preliminary evaluation of leveraging dynamic DNNs in Jellyfish. Section 9 discusses limitations and future work. Section 10 describes related work. Section 11 provides a final conclusion.

Extended version This paper significantly extends our prior conference paper, "Jellyfish: Timely Inference Serving for Dynamic Edge Networks", published at the 43rd IEEE Real-Time Systems Symposium (RTSS) (Nigade et al. 2022). This paper includes the following key additions:

- A new Sect. 2.4, explaining techniques for generating DNN variants and highlighting their advantages and disadvantages for DNN inference serving.

- System details for online latency budget estimation in Sect. 3.3.
- Addition of a new table that lists all the notations used in the paper (Table 1).
- A new part in Sect. 4.3 that describes parameter selection policies for the simulated annealing algorithm.
- A new Sect. 5 that presents our design sketch for leveraging dynamic DNNs to optimize DNN adaptation overhead and batched inference execution using techniques like network pruning and early-exit.
- Inclusion of implementation details concerning client-side decisions and overhead in Sect. 6.
- Detailed elaboration of anomalies and new observations found in end-to-end performance analysis (see Sect. 7.2).
- Addition of new experimental results (Table 2) comparing Jellyfish with a baseline under cases with heterogeneous clients.
- A new Sect. 8 that presents the preliminary evaluation of dynamic DNNs in Jellyfish, including the impact on DNN adaptation and the performance of the early-exit technique.
- Discussion and limitations in Sect. 9.1 for supporting accuracy constraints, providing clients-side adapters, and handling unreliable communication networks.
- A new Sect. 9.2 that discusses future work and the limitations of leveraging dynamic DNNs.
- Inclusion of related work on dynamic DNNs for a comprehensive overview.

These additions contribute to a more thorough and detailed analysis of the Jellyfish framework to further improve its overall performance.

2 Background and motivation

2.1 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 (Liu et al. 2019; Ali et al. 2020; Ahmad et al. 2020). A DNN consists of multiple layers. To achieve high accuracy, DNNs employ an increasing number of layers (He et al. 2016; Szegedy et al. 2017), 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 (Liu et al. 2019; Ali et al. 2020).

DL inference serving on servers has been extensively studied recently (Crankshaw et al. 2017; Shen et al. 2019; Gujarati et al. 2020; Romero et al. 2021a). Applications based on DL inference typically require some form of latency guarantee, often specified as a service-level objective (SLO), to ensure the

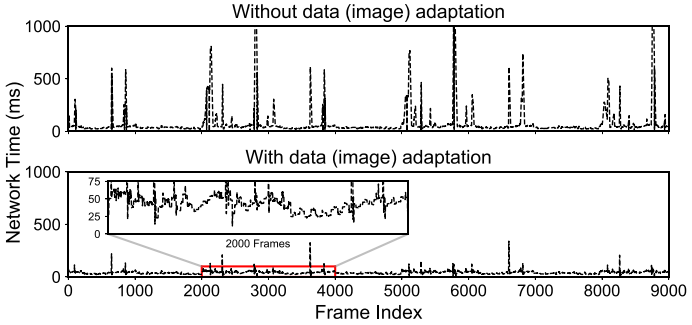


Fig. 2 Network time for sending JPEG images with adaptive resolutions over an LTE network (bandwidth shown in Fig. 14)

usefulness of the inference result. For example, digital assistance like Amazon Alexa dictates that the tail latency is constrained within 200–300ms (Chen et al. 2017). 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.

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.

2.2 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 (Zhang et al. 2020b; Rusci et al. 2020). The idea has been applied in several DL inference serving systems, such as ALERT (Wan et al. 2020), 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 (Lee and Nirjon 2020; Kannan and Hoffmann 2021). Overall, DNN adaptation techniques

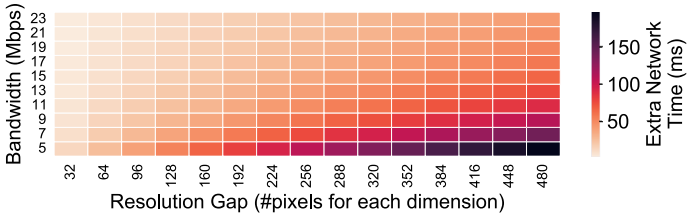


Fig. 3 The extra network time spent when a client sends JPEG images at a resolution larger than the input size of the DNN on the server under varying bandwidth conditions (covering the bandwidth range of a real-world LTE network). The resolution gap is defined as the chosen client-sending resolution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

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 (Zhang et al. 2018a; Ran et al. 2018), 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. Figure 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.

2.3 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 (Zhang et al. 2018a; Ran et al. 2018; Wan et al. 2020; Lee and Nirjon 2020). 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 (Dai et al. 2016). 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 (Zhang et al. 2018a; Ran et al. 2018; Du et al. 2020; Nigade et al. 2021). 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 (Nigade et al. 2021; Jiang et al. 2021). 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.

2.4 Techniques for generating DNN variants

Inference serving systems typically deploy functionally-equivalent DNN variants and switch between them at runtime to trade accuracy off for execution latency. In the following, we will discuss techniques for generating DNN zoos using *static* and *dynamic* DNNs and highlight their advantages and disadvantages for DNN inference serving. Note that Jellyfish and its scheduling algorithms are generally applicable regardless of the techniques used for generating DNN variants.

2.4.1 Static DNNs

Static DNNs have a fixed computational graph and set of parameters. They follow the same computational path to process each input and get the prediction output. Many serving systems (Wan et al. 2020; Romero et al. 2021a) use a set of *static* DNN models with different latency-accuracy tradeoff profiles, also known as the bag-of-models technique. One benefit of this technique is that we can easily generate a set (bag) of DNN variants by downloading off-the-self pretrained DNNs from the publicly available model hosting hubs. These DNN variants can have different architectures (e.g., varying numbers of layers and filters) and are trained separately. This technique, therefore, leads to different parameters for every DNN variant. Before a DNN variant can be used for inference, it needs to be loaded into the GPU memory. Depending on the parameter size of DNNs, the number of DNNs in the set, and the memory capacity of the GPU, not all DNN variants may fit on the GPU at once. As a result, the DNN variants need to be continuously swapped in and out of the GPUs following the adaptation decisions made by the inference serving scheduler.

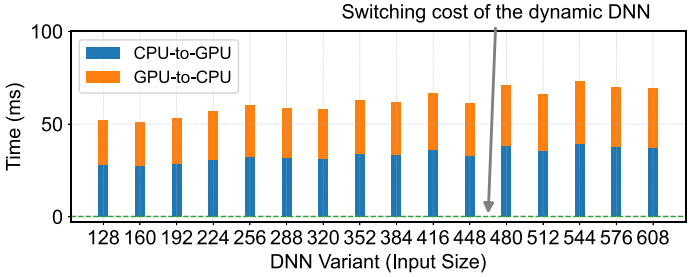
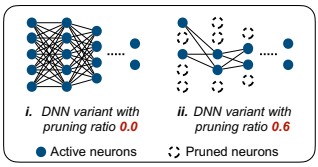


Fig. 4 The DNN switching cost from CPU-to-GPU (swapped in) and GPU-to-CPU (swapped out) for 16 static DNN variants (their parameter sizes range in [132.66, 215.94] MB) compared to the dynamic DNN when measured in isolation without any interference (due to contention on the PCIe) bus from parallel inference workloads on an NVIDIA RTX2080Ti GPU. Dynamic DNN incurs almost negligible (0.17 ms) switching costs. The x-axis denotes the input size of DNN variants — the smaller the DNN input size, the lower the latency and accuracy

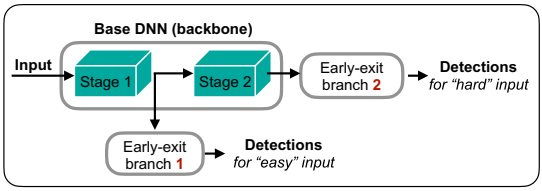
The DNN switching cost in terms of time can be quite high (hundreds of milliseconds) and may lead to a significant delay in DNN adaptation. Such a delay can further result in a *mismatch* between the input data size and the expected DNN variant, which ultimately leads to latency SLO violations and reduced accuracy.

Figure 4 shows the switching cost of different DNN variants (whose parameter sizes range in [132.66, 215.94] MB) when swapped in and out of the GPU without considering the interference (due to contention on the PCIe) from the parallel inference workload. We generate the static DNN variants and the dynamic DNN using the OFA-ResNet50 (Cai et al. 2020) and DETR (Carion et al. 2020) models (explained in Sect. 8.1). We observe an increasing trend, albeit weak, in the switching cost as DNN size increases, particularly for CPU-to-GPU transfer. The difference (variance) in switching costs across the different DNN variants remains minimal. This is because our current hardware setup uses a PCIe 3.0 x16 link to the NVIDIA RTX2080Ti, which offers a high theoretical bandwidth of approximately 16GB/s. Hence, for a narrow range of relatively small parameter sizes [132.66, 215.94] MB, the data transfer time does not exhibit significant variation. Nevertheless, even in this ideal scenario with minimal interference, the switching cost is considerable.

Jellyfish employs a DNN prefetching (caching) technique to alleviate the switching cost issue (see Sect. 4.4). The idea is to keep prefetching a few extra DNN variants neighboring (in size) to the currently active DNN variant in the hope that the scheduler will select the next active DNN variant from the neighbors of the current ones that are cached on the GPU. However, prefetching techniques can struggle in highly dynamic environments where adaptation decisions are mostly irregular. In addition, even when the DNN variant cache hit ratio is high, we must continuously swap in and out the DNN variants that are neighbors to the active DNN variant to keep the DNN cache active on the GPU, which inevitably interferes with the ongoing inference process due to contention on the data transfer link (such as PCIe).



(a) DNN variants with a neuron pruning method



(b) Early-exit DNN with two exit branches

Fig. 5 An illustration of dynamic DNNs with two popular methods: **a** network (neuron) pruning, **b** early-exit

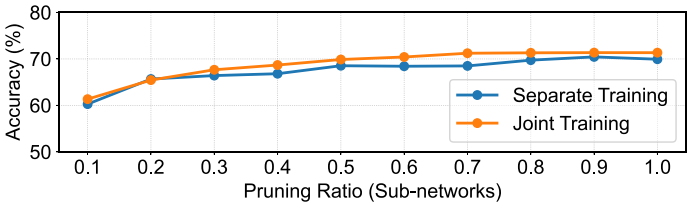


Fig. 6 The accuracy comparison of sub-networks (generated with different pruning ratios) trained separately and jointly. The accuracy curve of all sub-networks embedded in one big base DNN network and trained jointly is close to that of sub-networks trained separately (with separate parameter sets). Here, the architecture of all sub-networks used in the experiment is similar to the AlexNet architecture employed in SubFlow (Lee and Nirjon 2020) and trained on the CIFAR10 classification dataset (Krizhevsky et al. 2009)

To avoid the switching cost altogether, we study *whether it is possible to embed all DNN variants as sub-networks in one big base DNN network (sharing a common parameter set) and enable the switching between sub-networks dynamically on-the-fly.*

2.4.2 Dynamic DNNs

To overcome the limitations of the bag-of-models technique, we investigate the family of DNN architecture called *dynamic* DNNs. Recently, many works have proposed to enable the dynamic (partial) execution of DNN models during runtime to improve computational efficiency without incurring a significant loss in accuracy (Han et al. 2022). One of the primary methods for dynamic DNN execution is through *online network pruning*, as depicted in Fig. 5a. In online network pruning, a less important (or redundant) portion of the DNN network is pruned or skipped during the execution, expecting it to improve computational efficiency. For example, we can prune less important activations (neurons) from the layer's output or we can prune weights, channels in the filter, or the whole filter itself from the parameter set. The decision to prune and select a sub-network for the dynamic execution mostly depends on the *changing content* of the input. Although this approach aims to maintain the same accuracy for sub-networks compared to the bigger base DNN, it cannot guarantee

latency-bounded execution per input request (a prerequisite in many inference serving systems (Gujarati et al. 2020)).

SubFlow (Lee and Nirjon 2020) and Heo et al. (Heo et al. 2020) propose to execute sub-networks within latency bounds depending on the changing compute resource or time budgets, but by trading accuracy for latency guarantees. One of the main drawbacks of such an approach is a significant loss in accuracy when sub-networks are not retrained. If we retrain sub-networks separately, it generates a separate parameter set for every sub-network, and thus, we arrive at the same problem of high switching costs as in bag-of-models. OFA (Cai et al. 2020) and DRESS (Qu et al. 2022) propose joint training for all sub-networks together, avoiding different parameter sets for different sub-networks. Figure 6 shows the accuracy comparison between sub-networks (generated with different pruning ratios) when trained separately and jointly. The accuracy curve for sub-networks trained jointly is close to that of sub-networks trained separately for the classification task. The joint training thus can help us embed all sub-networks in one big base DNN. However, as many inference serving systems require latency-bounded execution, we have to forgo the sub-network selection technique based on the input content, especially in the batching scenario where inputs in the batch can have different content. That means we miss an opportunity to optimize the sub-network selection for accuracy.

Another method to enable dynamic DNN execution is to exit early from the DNN (a.k.a. *early-exit DNNs*, as illustrated in Fig. 5b), when the input data is easy to infer, thereby amortizing the computational cost (Teerapittayanon et al. 2016). To that end, we need to place exit branches at intervals along the base DNN to decide and exit when partial execution up to the exit point is confidently accurate in its prediction. The exit decision from the early-exit DNN primarily depends on the input content (difficulty level) and less on the latency constraint. Therefore, we cannot optimally utilize early-exit DNNs for latency-bounded execution due to their coarse-grained execution choices (only a few exit branches are available). However, the batched inference scenario that is typical in inference serving systems, provides opportunities to utilize early-exit DNNs. When some input requests in the batch exit early, the computational need of those exited input requests becomes zero in the subsequent execution, allowing more compute resources and thus faster completion opportunity for the remaining requests in the batch.

While dynamic DNNs can offer benefits to inference serving systems, training them is still a non-trivial, resource- and time-consuming process, especially when combining network pruning and early-exit techniques. Furthermore, all existing works on early-exit focus primarily on the classification task (Teerapittayanon et al. 2016; Laskaridis et al. 2020b), sometimes on the segmentation (Li et al. 2017), or text generations tasks (Schwartz et al. 2020), and rarely on the object detection task which is an important fundamental task in many video analytics applications. In Sect. 5, we sketch a design for leveraging dynamic DNNs for the object detection task to improve the performance of batched inference and DNN adaptation. Note that, while object detection is chosen here as an application task, Jellyfish is equally applicable to many other tasks.

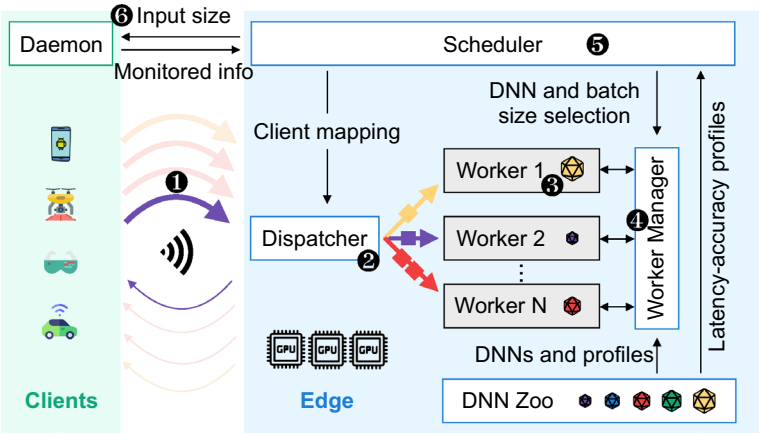


Fig. 7 An overview of the Jellyfish system architecture

3 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.

3.1 Overview

An overview of the Jellyfish system architecture is shown in Fig. 7. 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 preprocessing 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).

3.2 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 (Shen et al. 2019; Gujarati et al. 2020)). We exclusively employ GPUs for the DNN inference task analogous to other serving systems (Shen et al. 2019; Gujarati et al. 2020). 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 Sect. 4.4).

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 Sect. 4.

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 (Han et al. 2016), early-exit (Laskaridis et al. 2020a), and neural architecture search (Cai et al. 2020). We leverage the bag-of-models[2] technique to select a list of pretrained 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.

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 (Du et al. 2020; Laskaridis et al. 2020a; Ran et al. 2018; Yin et al. 2015).

3.3 Online latency budget estimation

Our scheduling algorithm requires an estimate of the compute (latency) time budget for every client to optimally select and map DNN models. As discussed above, the total time budget (i.e., end-to-end latency SLO) is composed of the time spent on the network (*network time*) and the time required for inference request handling (*compute time*). This relationship can be expressed as:

$$compute\_time = SLO - network\_time$$

We then use the compute time as the *latency budget* available for request handling on the server, which includes request queuing, input data preprocessing, and DNN execution. We also factor in other constants or measurable delays, such as the dispatch time between the dispatcher and the worker units. As the time spent by requests on the communication network is variable and depends on the real-time network bandwidth (see Fig. 2), the latency budget is highly dynamic and, thus, needs to be computed *online* and *continuously* adjusted.

The network time is mostly dominated by request transfer time from the client to the server, as the size of the input data is typically much larger than the response (i.e., the inference result). To compute the network time for a client, we estimate its real-time network upload bandwidth (throughput) and calculate the time per request as:

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

Table 1 List of notations

Symbol	Description
$\mathbb{G}$	Set of GPU workers
$\mathbb{M}$	Set of diverse DNN models
s_j	Input size of DNN m_j
a_j	Expected accuracy of DNN m_j
$l_j(b)$	Inference latency of DNN m_j for batch size b
$t_j(b)$	Throughput of DNN m_j for batch size b
$\mathbb{C}$	Set of clients
O_i	Pre-specified latency SLO (ms) of client c_i
λ_i	Request rate of client c_i
L_{ij}	Latency (compute) budget of client c_i for DNN m_j
s_i	Request (data) size of client c_i
W_i	Estimated network bandwidth of client c_i
x_{ijk}	Binary decision variable to denote if a client c_i is mapped to DNN m_j deployed on GPU worker g_k
b_k	Integer decision variable to denote the batch size for the selected DNN on worker g_k
z_{kj}	Auxiliary decision variable to denote the selection of DNN m_j on GPU worker g_k

$$network\_time = \frac{s}{W} + RTT \tag{1}$$

Here, s is the input data size (after compression) and W is the network throughput. RTT is the round-trip propagation delay that can be measured using existing tools like `ping`. To efficiently estimate the network bandwidth over time, we can leverage the bandwidth estimation techniques adopted in several recent works such as DDS (Du et al. 2020), SPINN (Laskaridis et al. 2020a), DeepDecision (Ran et al. 2018), or FastMPC (Yin et al. 2015). We will detail our implementation on bandwidth estimation further in Sect. 6.

4 Scheduling algorithms

In this section, we provide the formulation of the scheduling problem and present our algorithm design. Table 1 contains a list of notations used in the problem formulation.

4.1 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 (Ran et al. 2018; Wan et al. 2020; Zhang et al. 2020a), 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 (Wang et al. 2020). 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 (Kannan et al. 2019).

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 (Yu et al. 2022), where each instance is treated as a separate worker. The DNN execution time is highly predictable (Gujarati et al. 2020), 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 Sect. 3.2.

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:

$$(P_1) \max_{\{x,b\}} \quad \sum_{i,j,k} a_j \cdot \lambda_i \cdot x_{ijk} \tag{2}$$

$$\text{s.t.} \quad \sum_{j,k} x_{ijk} = 1, \forall i \tag{3}$$

$$\sum_j z_{kj} \leq 1, \forall k \tag{4}$$

$$z_{kj} \geq x_{ijk}, \forall i, j, k \tag{5}$$

$$\sum_{j,k} x_{ijk} \cdot 2l_j(b_k) \leq \sum_{j,k} x_{ijk} \cdot L_{ij}, \forall i \tag{6}$$

$$\begin{aligned} &\sum_{i,j} x_{ijk} \cdot \lambda_i \leq \sum_j z_{kj} \cdot t_j(b_k), \forall k \\ \text{vars} \quad &x_{ijk}, z_{kj} \in \{0,1\}, b_k \in [1...B] \end{aligned} \tag{7}$$

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. (2) 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. (3). Equation (4) captures that at most one DNN is selected for each worker. Equation (5) guarantees that all clients are mapped to the same and correct DNN when they are mapped to the same worker. Equation (6) 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 mentioned in Sect. 3.3 (i.e., by 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 (Shen et al. 2019). Thus, the latency (compute) budget should be at least *twice* the DNN execution time. Equation (7) 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.

4.2 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. (2). 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.

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. (6).

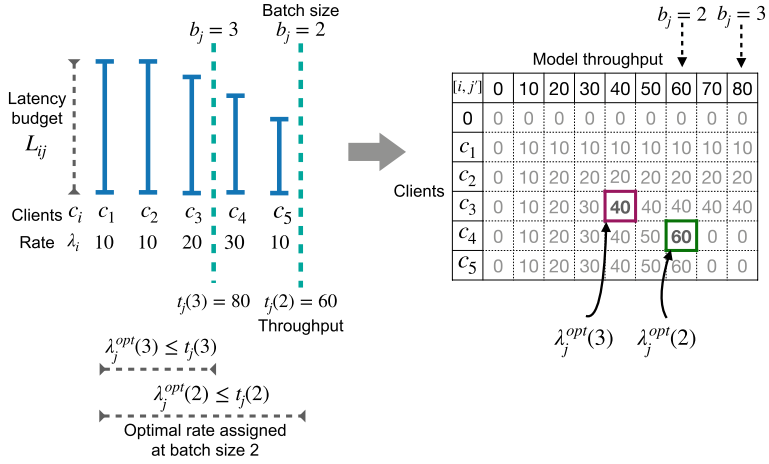
We observe that, given a fixed batch size, the client-DNN mapping problem reduces to a standard 0-1 knapsack problem, where we treat clients as items, the request rates of clients as weights and values, and the maximum throughput of the DNN for the given batch size as the knapsack capacity. The problem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

Algorithm 1: Client-DNN Mapping

```
    Function MapClients(clients, dnn_models):
 1: sort dnn_models in descending order of their accuracies
 2: map ← { }
 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 (⟨ model, clients', batch_size ⟩)
 7:     clients ← clients − clients'
 8: return map

    Function FindOptimalClients(model = ⟨l_j, t_j⟩, 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 ⟨L_ij, λ_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 (clients + 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'
```

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. 8. We then recursively start computing the cell values (aggregate request rate) for each row representing clients in descending order of their latency budget. For each client (row),



$[i,j']$	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. 8 An illustrative example to show the DP-based algorithm

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. 8 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 Sect. 7.6, 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.

4.3 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.

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. (2). 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) (Aarts and Korst 1990), 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: 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

best_models $\leftarrow$ $previous\_models$
best_mapping $\leftarrow$ `MappingAlgo` ($clients$, best_models)
for mode in [DEGRADE, UPGRADE] **do**
    $T \leftarrow T_0$
    mapping $\leftarrow$ best_mapping, dnn_models $\leftarrow$ best_models
    while not `Stop`(mapping, mode) **and** $T > T_{min}$ **do**
        models$'$ $\leftarrow$ `NeighborsGenerator` (dnn_models, mode)
        mapping$'$ $\leftarrow$ `MappingAlgo` ($clients$, models$'$)
        if `Better`(mapping$'$, best_mapping, mode) **then**
            best_mapping $\leftarrow$ mapping$'$, best_models $\leftarrow$ models$'$
        diff $\leftarrow$ mapping$'$.metric $-$ mapping.metric
        if diff > 0 **or** **exp**$(-\text{diff}/T) >$ `rand`$(0, 1)$ **then**
            mapping $\leftarrow$ mapping$'$, dnn_models $\leftarrow$ models$'$
        $T \leftarrow T \cdot \alpha$
return best_models

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**.

Simulated annealing parameters Although the SA framework has been widely used, applying it 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, and stopping condition. We determine the SA parameters in Algorithm 2 for DNN selection as follows:

- *Stopping Condition (Line 6)*: One crucial property of SA is that it offers a good tradeoff between exploration and exploitation controlled by the temperature T and its temperature reduction strategy (a.k.a. the cooling schedule). In the UPGRADE mode, we stop the exploration when the temperature falls below the minimum temperature T_{min}. However, in the DEGRADE mode, we stop much earlier when **O1** is satisfied.
- *Neighbors Generator (Line 7)*: To effectively explore the search space, we generate neighboring states (i.e., DNN set) based on the operational mode. In DEGRADE mode, we decide randomly whether to downgrade the DNN size in the set (state) by one step or keep it the same. In the UPGRADE mode, we also add a random decision to upgrade the DNN by one step.
- *Better Solution (Line 9)*: The function named `Better()` defines the quality of the current solution. In the UPGRADE mode, the solution is better if it does not violate **O1** and has higher accuracy. Here, we define accuracy as the weighted average of all clients' inference accuracy (determined by the DNN's profiled accuracy to which it is mapped, otherwise zero), where the clients' frame rate determines the weight (see Eq. (2)). In the DEGRADE mode, the solution is better if it improves **O1**.
- *Acceptance Probability (Line 12)*: Accepting a weak solution is determined by the difference (named `diff`) between the weak solution and the current solution. In the DEGRADE mode, the `diff` is simply the difference between the value **O1** of two solutions. In the UPGRADE mode, if the new solution does not violate **O1**, the `diff` is between the value **O2** of the two solutions; otherwise, the solution is dropped. The probability of acceptance also depends on the value of temperature T that decreases over iterations by a reduction factor α.

The effectiveness of SA depends on the initial value of T, the minimum temperature T_{min}, and the temperature reduction factor α. In our simulation, we found the initial value of $T = 0.0125$, $T_{min} = 0.0005$ and $\alpha = 0.99$ offers decent performance. Note that the accuracy value of a solution is in the range [0, 1]. Intuitively, it means that

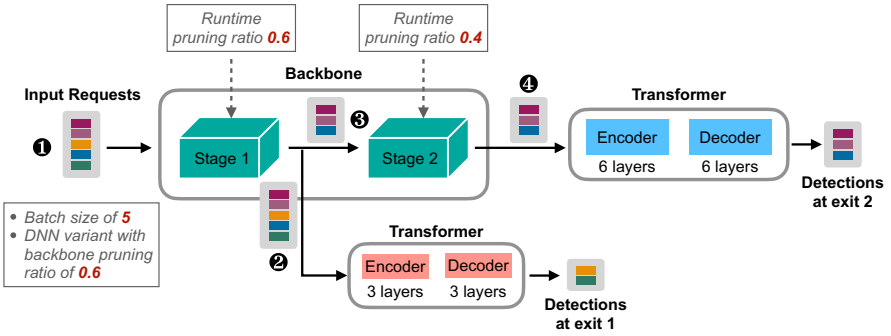


Fig. 9 Dynamic DNN based on a transformer-based vision model (e.g., DETR) to generate different DNN variants by pruning the backbone network with different pruning ratios. The early-exit branches optimize, in the case of batched inference, latency and accuracy by enlarging (i.e., reducing the pruning ratio) the backbone stage for inference requests in the remaining batch

we accept the weak solution with high probabilities when the absolute `diff` value is lower than the T value.

4.4 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.

5 Design sketch for dynamic DNNs

In this section, we sketch a design for leveraging *dynamic* DNNs for the object detection task by combining the best of the two methods used for creating dynamic DNNs: (a) *network pruning* for creating fine-grained DNN variants (sub-networks) to switch between them dynamically on-the-fly, avoiding the DNN switching cost and (b) *early-exit* for optimizing the performance of batched inference. The performance gain of early-exit manifests in three aspects: (1) Inference requests that exit early with high confidence reduce the inference latency of these requests. (2) When a portion of the batch exits early, the batch size drops, and thus, more

compute resources can be allocated to the requests remaining in the batch, leading to faster completion of these requests. (3) The availability of extra compute resources resulting from early-exit creates the opportunity of reducing the pruning ratio of the remaining layers in the DNN variant (i.e., enlarging the subsequent layers). The enlarged part of the DNN for the remaining requests in the batch can potentially contribute to accuracy improvement. Figure 9 shows our proposed dynamic DNN for the object detection task, and we explain the inference workflow in the following parts.

5.1 Dynamic DNN creation for object detection

Many works on dynamic DNN execution, especially on early-exit DNNs, have focused on a simple classification task and rarely on complex regression tasks like object detection (Han et al. 2022). One of the reasons is that typical object detection models have a highly compute-intensive object detection head (for region proposals, classification and regression) (Ren et al. 2017) and are tightly integrated with the feature extractor in the case of single-stage object detectors (Bochkovskiy et al. 2020). Thus, adding multiple heavy object detection heads as exit branches defeats the primary objective of saving computations.

Recently, transformer-based vision models such as DETR (Carion et al. 2020) and Deformable DETR (Zhu et al. 2021) have become increasingly popular for object detection. These models use popular DNN architectures like ResNet to extract image features and pass these features through the transformer block (containing an encoder-decoder module) to output detections. Fortunately, the execution time of the transformer block is relatively small compared with their feature extractors (called the backbone DNN) (Sreedhar et al. 2022; Samplawski and Marlin 2021). Therefore, we can add lighter transformer blocks as exit branches at critical points in the backbone DNN, thus enabling effective early-exit for object detection. Furthermore, we use network pruning techniques on the backbone DNN (i.e., the feature extractor) to create DNN variants, thereby realizing our idea of dynamic object detection DNN model with early-exit functionality.

Figure 9 shows our proposed DNN architecture based on vision transformers for object detection combining network pruning and early-exit techniques. In the figure, the Jellyfish scheduler ❶ selects a DNN variant with an initial pruning ratio of 0.6 (fraction of DNN to be pruned) to be deployed on a GPU for serving input requests with a batch size of 5, given a fixed compute time budget. The batched input goes through stage 1 (a block of consecutive layers) of the backbone network. The output feature from the first stage ❷ moves to the first transformer block (i.e., exit branch 1). At exit branch 1, the *easy-to-predict* requests from the batch might exit confidently, leading to a decrease in batch size at stage 2. As the batch size decreases in stage 2, allowing for more compute resources per inference request in the batch, we can reduce the pruning ratio of stage 2 (i.e., enlarging the backbone network in this stage) without affecting the overall batch execution time. Even if we do not enlarge the network in stage 2, it can still accelerate the execution of the remaining batch. The remaining batch then ❸ passes through (the *enlarged* version

of) stage 2 of the backbone. Finally, the remaining batch features ❹ are fed into the second transformer block (i.e., exit block 2) to get the predictions for the remaining requests in the batch. In practice, we use a “transformer" head, but as a concept, it is a more general “classifier" or “prediction" head that does not necessarily has to be a transformer.

The above design allows smooth runtime adaptation of DNN variants with negligible costs and accelerated execution of batched inference. In addition, we expect a gain in overall accuracy for a fraction of the requests (the ones that do not exit-early) due to the enlargement of stage 2 (with a new and low pruning ratio of 0.4) compared to passing the whole batch through a static DNN variant with a high pruning ratio of 0.6.

A major challenge is how to decide when to exit early, a non-trivial problem for the object detection task. We explain our approach to designing an early-exit decision-making module in Sect. 5.3.

5.2 DNN training

Training our dynamic DNN with the network pruning and early exit functionality involves a multi-step process, as it should train multiple exit branches. We build our training process on top of existing multi-step training strategies adopted in the early exit paradigm (Laskaridis et al. 2021; Matsubara et al. 2023). Our multi-step training process works as follows: In the first step, the main exit branch is trained end-to-end together with the backbone DNN. As the backbone DNN contains different DNN variants generated with different pruning ratios, we must jointly train all the DNN variants, similar to DRESS (Qu et al. 2022). That means the main exit branch should be trained on features from all DNN variants, and we should combine (aggregate) the loss values for all the DNN variants when back-propagating. Once the main exit branch (on all DNN variants) and the backbone DNN are trained, we train the remaining early-exit branches in the second step. In this step, we train only the respective exit branch and freeze other parts of the backbone DNN. Similar to the first step, we must train exit branches on features from all the backbone DNN variants.

5.3 Early-exit decision making

Once we have our fully trained dynamic DNN, the question is *how to decide when to exit from a particular exit branch*. Compared with the classification task, the decision to exit from a particular exit branch is non-trivial and is an under-explored topic for regression tasks like object detection (Laskaridis et al. 2021). For classification tasks, the simple approach is to compare the top class score or entropy of the `softmax` output vector of the exit branch against a predefined threshold to make an exit decision (Teerapittayanon et al. 2016). However, for object detection, we need to examine and quantify various factors, including the object class scores of bounding boxes, the bounding box area and the number of bounding boxes recalled

correctly, thus making it hard to formulate the exit decision-making strategy. The class score of bounding boxes is relatively easy to measure and quantify using similar approaches to the classification task. However, estimating the accuracy of bounding boxes and the number of bounding boxes recalled is a hard problem in the absence of the ground truth.

Our approach is to use a *learning-based* module, e.g., a lightweight binary classifier. The classifier takes some extracted features (intermediate or output features) from the exit branch and outputs a scalar value (in [0, 1]), indicating the confidence in the outputs of the exit branch. A high value indicates that the output (prediction) of the exit branch is likely accurate. The real challenge is training such a binary classifier. To address this challenge, we propose to create a training dataset of positive and negative samples based on the loss value between the exit branch output and the ground truth. We can use the same loss function used in training the full DNN model to compute the loss value. The samples with a loss value lower than a predefined threshold contribute to positive samples, and to negative samples otherwise. Once the training dataset is created, the learning-based module can be trained like any other binary classifier.

Our learning-based approach appears promising in making exit decisions. In our preliminary experiment with the object detection model (ResNet50-DETR) and two-exit branches, the main branch achieves an accuracy of 0.623 mAP (mean average precision) on the COCO-val2017 image dataset (Lin et al. 2014), and exit-branch 1 achieves 0.401 mAP when all validation samples exit from the respective branch. With the oracle that makes an early-exit decision reasonably accurate, the overall mAP of the early-exit DETR is 0.594, where 35.76% of samples exit from exit-branch 1. The oracle is based on the dataset of positive and negative samples generated using the loss values and a predefined loss threshold (as described above). Our learning-based decision-making module achieves 0.587 mAP for 30.94% of samples exiting from the first exit branch, demonstrating the high potential of enabling an early-exit functionality for the object detection task.

5.4 Dynamic DNN summary

To summarise, dynamic DNNs generated with a combination of network pruning and early-exit functionality have the potential to avoid the DNN switching cost by embedding all DNN variants as sub-networks in one shared base DNN. The early-exit functionality can be used for efficient execution of the batched inference thanks to the reduced computation for the remaining batch with a reduced number of inference requests, or it can be used to improve accuracy by enlarging the DNN for the remaining batch. Specifically, a transformer-based vision model enables the early-exit functionality for the object detection task attributed to their lightweight detection head. In Sect. 8, we demonstrate the applicability of our design to Jellyfish through preliminary evaluations.

6 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 Sect. 7.5).

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 (PyTorch 2022). 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.

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 (Yin et al. 2015)) of

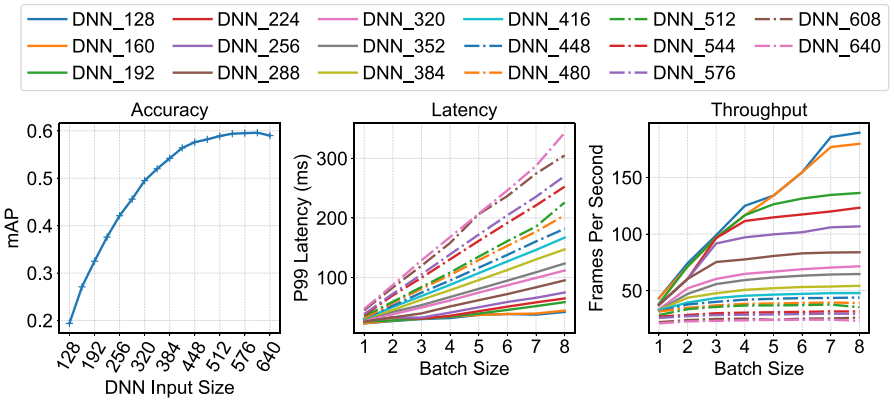


Fig. 10 The DNN performance profiles

the client's bandwidth over the past one second as the estimated bandwidth for the client.

Client-side decisions and overhead We implement a very lightweight process (in terms of compute, memory and network usage) on clients to apply decisions (i.e., data size to use) received from the scheduler running on the server and only perform lightweight data resizing and bandwidth estimation, which is typical for adaptive-video-analytics systems (Zhang et al. 2018a; Ran et al. 2018; Laskaridis et al. 2020a). Furthermore, Jellyfish imposes a negligible network overhead by piggybacking small metadata (a few Kbps) on the video data.

7 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?

7.1 Methodology

DNN zoo We employ a well-known pretrained YOLOv4 DNN architecture (Bochkovskiy et al. 2020) and use its Pytorch-YOLOv4 implementation (Tianxiaomo 2020) 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 (Lin et al. 2014) and use the standard comparison metric called mean average precision (mAP) to rank the DNNs. Figure 10 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) 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 (Du et al. 2020), 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, which is consistent with earlier works like VideoStorm (Zhang et al. 2017), Chameleon (Jiang et al. 2018), AWStream (Zhang et al. 2018a), and DDS (Du et al. 2020). We exclude missed frames as it is hard to quantify their impact on the user application. Similar to DeepDecision (Ran et al. 2018) and DDS (Du et al. 2020), 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.

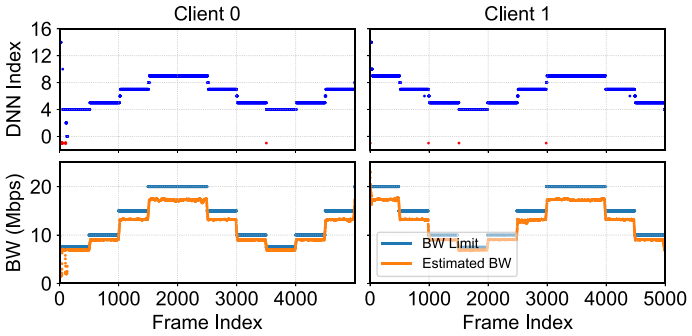


Fig. 11 DNN selection for each client in a setting (two clients, 25 FPS, 100ms SLO) under a synthetic network trace. Red dots indicate dropped frames

7.2 End-to-end performance

We first analyze the end-to-end performance of Jellyfish under a synthetic network trace. Following AWStream (Zhang et al. 2018a), 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. (6)). 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.

DNN adaptation Fig. 11(bottom) depicts the estimated bandwidth values close to the actual bandwidth limits, displaying the accuracy of our bandwidth estimation. Figure 11(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.

In Fig. 11(bottom), we observe that the error gap between *estimated* and *actual* bandwidth values increases with the actual bandwidth. Two main factors influence this error gap: (a) As we use the Linux `tc` utility to replay network bandwidth traces, the difference between the *effective* bandwidth achieved by the `tc` utility and the actual bandwidth widens with higher actual bandwidth values, leading to an increased error gap in the bandwidth estimation. However, we expect this behavior to be absent in real-world scenarios. (b) Our bandwidth estimation technique relies on the acknowledgment-based mechanism that introduces extra *host delays* (1-3 ms) on the server side. As the actual bandwidth increases, the impact of these

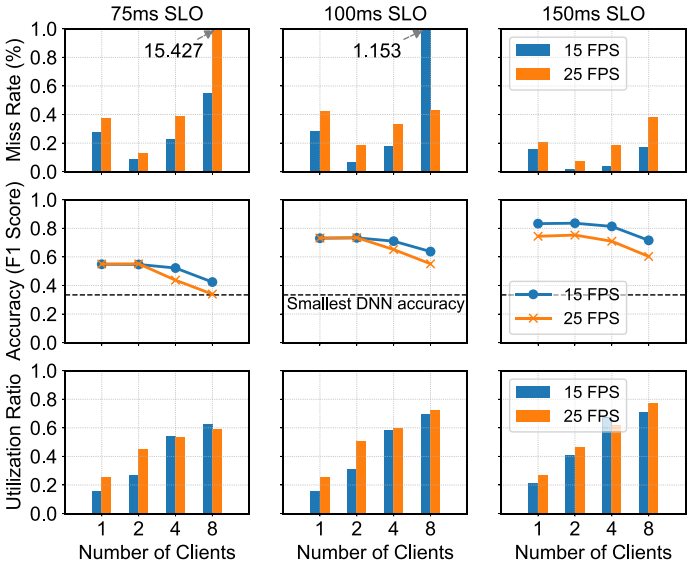


Fig. 12 The miss rate, accuracy and worker utilization of Jellyfish for varying SLOs, request rates, and numbers of clients under a synthetic network trace. We truncate the Y-axis(top) to 1% to focus on the extremely low values ($\leq$ 1%) observed in almost all experimental settings

host delays on network time estimation becomes more significant, resulting in a wider gap between estimated and actual bandwidth values. Note that for real-world network traces, the estimated bandwidth values may not match *precisely* the actual bandwidth values since we use the harmonic mean of frequently changing past bandwidth values.

When the error gap is large, Jellyfish adopts a conservative approach. It assumes that requests will spend more time on the network, resulting in low latency budgets on the server. As a result, the Jellyfish scheduler selects relatively smaller DNNs, slightly degrading the accuracy while maintaining the low miss rates.

Miss rate Fig. 12(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. On the contrary, and perhaps counterintuitive, for the setting with 25 FPS and with the same SLO and number of clients as in the 15 FPS setting, the miss rates are slightly lower. This is because Jellyfish often selects smaller DNNs with higher throughput to handle higher aggregate request rates (i.e., 200 FPS vs. 120 FPS). As a result, selecting

smaller DNN models for the same latency SLO provides more slack in the latency budget per request on the server. This extra slack enables requests to wait longer in the queue without being dropped, thereby helping to alleviate the issue with non-uniform 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 of the DNNs. Overall, *Jellyfish delivers an extremely low miss rate* ($\leq 1\%$) *when the system is not overloaded.*

Furthermore, we expect the miss rates (and accuracy) for the 1 client and 2 clients settings to be comparable due to the similarity in the distributions of DNNs selected by the scheduler. However, in our current Python-based implementation, the miss rate for the 1 client setting is *slightly* higher. In the 1 client setting, only one GPU (and the associated GPU worker) is active at any given moment, as dictated by the scheduler. We observe that when a GPU worker on our multi-GPU setup is *not* active for an extended period of time, the few subsequent inference requests on that GPU worker experience occasional delays. These delays occur during the retrieval of inference requests from the dispatch queue, fetching active model information from the loader thread, or executing the initial DNN inference on the GPU after a prolonged idle interval. We attribute these delays to the internal operations of commodity servers, which employ best-effort process/queue management and scheduling. While we observe these delays often in the 1 client setting, they are rare and have minimal impact, causing only slightly higher miss rates. Such occasional delays are one of the reasons that providing hard guarantees for the end-to-end latency SLO is challenging, especially on commodity setups. As discussed in Sect. 9.1, we expect system maintainers to tune the system for maximum predictability and stability when deploying Jellyfish-like serving systems.

Analytics accuracy Fig. 12(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. 12(bottom) shows the aggregate worker utilization ratio. The utilization is lower for settings with fewer clients and lower FPS due to lower

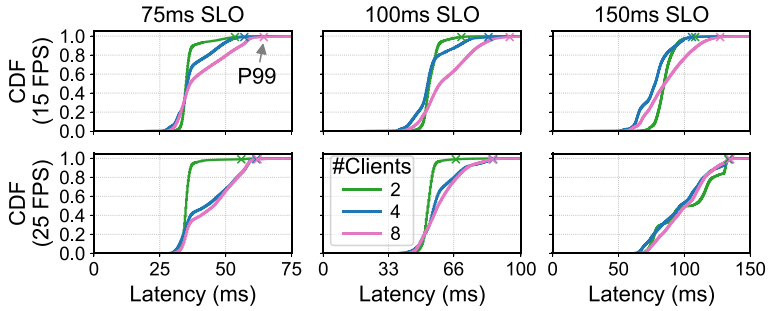


Fig. 13 End-to-end latency CDF for varying SLOs, request rates, and numbers of clients under a synthetic network trace

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.

Furthermore, there are some settings where the utilization can be *slightly* lower for higher aggregate request rates. For instance, in a specific setting with 15 FPS, 150ms SLO, and 4 clients, the scheduler often selects large DNNs with a small batch size (e.g., DNNs with indices 11, 9, and 7 and batch size 1). In contrast, for the setting with 25 FPS, 150ms SLO, and 4 clients, the scheduler selects relatively smaller DNNs with large batch sizes (e.g., DNNs with indices 7, 5, and 4 and batch sizes 3, 4, and 5, respectively), mainly to achieve higher throughput. Because of the smaller DNNs in the latter setting, there is more *slack* in the latency budget per request on the server since the number of clients and SLOs are the same. Therefore, the DNN executor can afford to wait longer for the requests to arrive without violating the latency constraint, allowing it to fill the batch with the desired size. This longer waiting time results in the DNN executor doing relatively less work and slightly more waiting compared to the setting with 15 FPS. Hence, in this particular case, the utilization could be slightly lower. Overall, the experiment confirms that *Jellyfish 's low miss rates are not at the cost of reduced worker utilization*.

End-to-end latency Fig. 13 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)*.

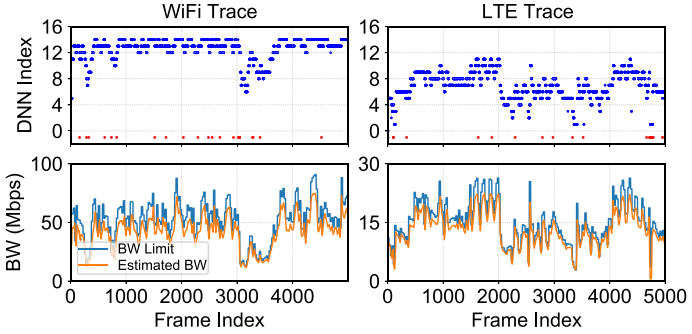


Fig. 14 Illustration of DNN adaptation for a client in a setting (2 clients, 25 FPS, 100ms SLO) with WiFi and LTE traces. Red dots mark dropped frames

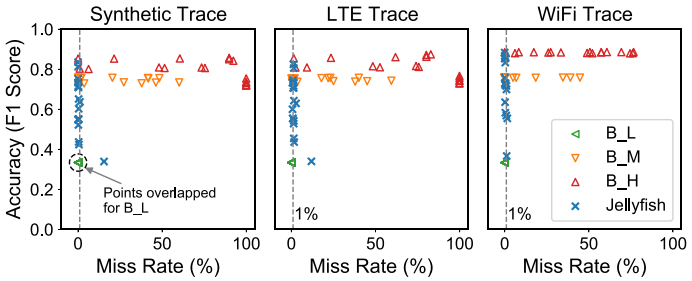


Fig. 15 Comparison of Jellyfish with baselines. B_M and B_H have excessive SLO violations making them ineffective, while B_L suffers from low accuracy

7.3 Comparison with the baselines

Baseline Inspired by Clockwork (Gujarati et al. 2020), 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.

Table 2 Performance of Jellyfish and B_M with heterogeneous clients

Network Trace	#Clients	Jellyfish		B_M	
		Miss Rate (%)	F1 Score	Miss Rate (%)	F1 Score
Synthetic	4	0.182	0.5737	15.510	0.7418
	8	0.648	0.5459	35.006	0.7440
LTE	4	0.952	0.5810	11.782	0.7457
	8	1.618	0.5456	31.936	0.7462
WiFi	4	0.622	0.6747	0.278	0.7577
	8	0.825	0.6264	23.17	0.7579

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 (Zhang et al. 2018a) (for the frame resolution) and offers optimal adaptation because the DNN accuracy is a monotonic function of the frame resolution (see Fig. 10).

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 Sect. 7.2, 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 4 G/LTE downlink bandwidth trace (van der Hooft et al. 2016) downscaled by a factor of two to represent the uplink bandwidth (Huang et al. 2012). The WiFi trace has a higher median bandwidth value (53.1 vs. 22.8 Mbps) than the LTE trace, exhibiting high and low bandwidth environments. In addition, the WiFi trace has a much narrower spread (19.6 vs. 38.51 interquartile range) than the LTE trace, indicating that the LTE network is a highly dynamic and hostile environment. Fig. 14(bottom) shows the estimated and actual bandwidth values. Figure 14(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. 15 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 clients) where at least one client cannot be mapped to any DNN (explained in Sect. 7.2). 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. Table 2 shows the performance of Jellyfish and B_M for heterogeneous clients under three different network traces. 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. 15 for homogeneous clients. Similar results hold for the synthetic and WiFi network traces. Note that as we consider the F1 score of only the processed requests, the accuracy of B_M is higher than Jellyfish but at the cost of extremely high miss rates.

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

7.4 Performance of joint adaptation

In Sect. 7.3, 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

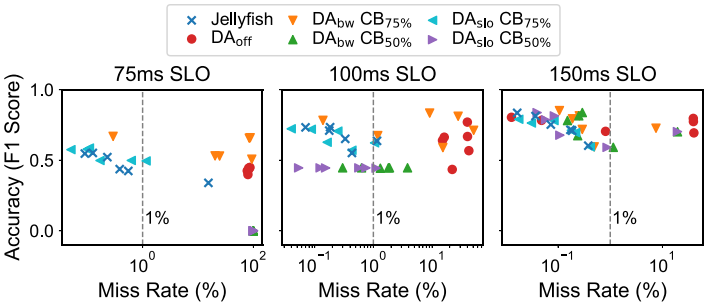


Fig. 16 The impact of the three data adaptation strategies on Jellyfish 's performance under a synthetic network trace. The label DA means data adaptation, and $CB_{x\%}$ means $x\%$ of SLO allocated as a compute time budget. The x-axis is in log scale

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 tradeoff 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 (Zhang et al. 2018a). 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.

Similar to Sect. 7.2, we use 18 experimental settings on synthetic network trace for performance comparison.

Results and discussion We show the results in Fig. 16. **(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

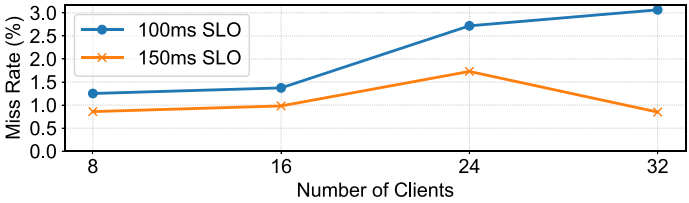


Fig. 17 The miss rate on 8 GPUs for varying numbers of clients operating at 15 FPS with the dynamic LTE trace

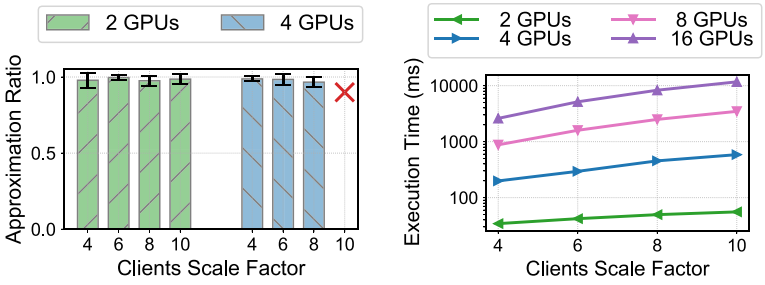
75ms SLO which results in a 100% miss rate. **(c)** For SLO-aware data adaptation (DA_{slo}), the miss rates are comparable to Jellyfish, but the accuracy is significantly lower for a compute time budget of 50% of the end-to-end SLO ($CB_{50\%}$). The accuracy of DA_{slo} is on par with Jellyfish for 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 (Dai et al. 2016). 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*.

7.5 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. 10. 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.

Figure 17 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



(a) Approximation ratio (mean) (b) Execution time in log scale

Fig. 18 Performance of Jellyfish scheduler for varying #GPUs and #clients. Here, the #clients is a product of Clients Scale Factor (x-axis) and #GPUs

scheduler may then assign many clients to the smaller DNNs. As mentioned in Sect. 7.2, 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)*.

7.6 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. 10. We run around 100 problem instances for each setting. The solution quality of each algorithm is measured by the accuracy objective defined in Eq. 2. We then use the approximation ratio between our algorithm and the MILP algorithm as a comparative metric.

Approximation ratio Fig. 18a 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. 18b, 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.*

7.7 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.*

8 Preliminary evaluation with dynamic DNNs

After demonstrating the effectiveness of Jellyfish with the bag-of-models technique, we now evaluate *dynamic* DNNs in Jellyfish. We implement a dynamic DNN for the real-world object detection task using a transformer-based vision model called DETR (Carion et al. 2020). The architecture of the DETR model consists of two parts, the backbone part (e.g., ResNet50, a convolutional neural network) extracting features and the transformer part (as object detection head) predicting bounding boxes and classes from the features extracted by the backbone.

8.1 Evaluation setups

DNN variants In our experiment, we replace the *static* ResNet50 backbone part of the DETR model with the *dynamic* OFA-ResNet50 (once-for-all architecture) (Cai et al. 2020), which allows us to switch between different sub-networks of the OFA-ResNet50 *dynamically* on-the-fly. Similar to Sect. 7, we consider 16 DNN variants or sub-networks in OFA-ResNet50, supporting 16 different input (data) sizes (in both spatial dimensions) from 128 to 608 with a step size of 32. We then add one transformer block as an object detection head to all 16 OFA-ResNet50 sub-networks (backbones), generating 16 OFA-ResNet50-DETR variants. The latency and throughput profiles of every OFA-ResNet50-DETR variant are similar to the profiles of YOLOv4 DNNs (see Sect. 7.1). That means the latency of a smaller DNN size (smaller input size) is lower than that of a bigger DNN size. To profile the latency of a DNN variant in the dynamic DNN for a particular batch size, we consider the worst-case execution latency which is when all the requests in the batch exit from

the final exit branch. In addition, we assume that the accuracy characteristics of our 16 OFA-ResNet50-DETR variants are similar to that of YOLOv4 DNNs, meaning that the accuracy increases monotonically with the DNN input size. Note that, as benchmarked in (Sreedhar et al. 2022; Samplawski and Marlin 2021), the DETR model's backbone part is relatively more expensive in computation (60%-90% of the total execution time) than the transformer part (10%-40% of the total execution time) depending on the input size and batch size used. Hence, we do not consider a dynamic architecture for the transformer block in this work.

We compare the performance of dynamic DNN variants with their *static* counterparts. The static DNN variants are equivalent to the dynamic DNN variants (sub-networks) in terms of model architecture (backbone and transformer block) and parameters, except that they have only one main exit branch (transformer block). The size of the parameters for each of the 16 static DNN variants ranges in [132.66, 215.94] MB, with a median value of 167.68 MB. In comparison, the size of the parameters of the dynamic DNN (embedding 16 or more sub-networks) is only 262.70 MB, which is 10 times lower than the total size of the 16 static DNN variants and just 56.6% larger than the median value for static DNN variants. Even if we add an early exit branch to the dynamic DNN variant, the size of the parameters is only 287.01 MB, making the dynamic DNN a highly *parameter-efficient collection* of DNN variants. Furthermore, the execution latencies of dynamic DNN variants are nearly identical to that of their static counterparts. The median value of the difference between the execution latencies of dynamic DNN variants and static DNN variants across all the batch sizes is just 0.817 ms with an interquartile range of 1.32 ms. For the execution latencies of dynamic DNN variants with the early-exit branches, see Sect. 8.3.

Implementation and setup We integrate our OFA-ResNet50-DETR dynamic DNN into Jellyfish and perform preliminary experiments. Similar to Jellyfish, we implement the components of our dynamic DNN in Python and PyTorch framework for DNN execution, as DETR (Carion et al. 2020) and OFA (Cai et al. 2020) implementations are also publicly available in the same frameworks. We use the same hardware and software setups for the server and clients as described in Sect. 6, except that we use only one GPU on the server because we are primarily interested in the DNN switching cost and the early-exit performance of DNNs selected on a GPU and to avoid the influence of DNN selection decisions made across GPUs by the Jellyfish scheduler in the case of multi-GPU setup. In addition, we use one traffic video from the video dataset (see Sect. 7.1) and replay video frames at different frame rates as the client requests. As we primarily focus on the execution efficiency of the dynamic DNN variants, we do not train the OFA-ResNet50-DETR variants and consider it as future work.

8.2 Impact on DNN adaptation

To evaluate the performance of dynamic DNNs with respect to adaptation, we test different experimental settings with {1, 2, 4} clients, SLOs from the set of {50, 100, 150} milliseconds (ms), request rates (frames per second) from the set of

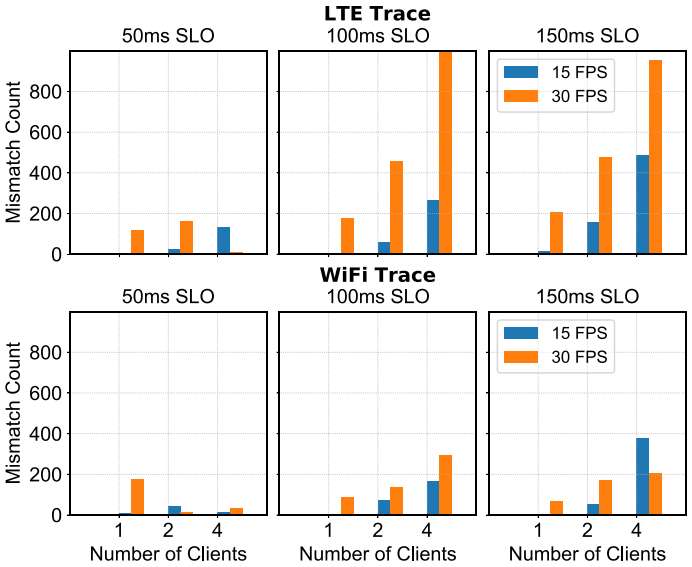


Fig. 19 Number of mismatches that occur when the newly arrived frames from clients are executed with the old DNN variant due to the delay in switching to the new active DNN variant. Here, we see that static DNN variants can lead to a significant number of mismatches, whereas dynamic DNN has zero mismatches in almost all cases (thus skipped in the figure)

{15, 30} FPS. To emulate dynamic network conditions, we use the two real-world network traces (i.e., LTE and WiFi) as mentioned in Sect. 7.3. Finally, we disable the DNN caching mechanism of Jellyfish.

Evaluation metric It is non-trivial to isolate and measure the impact of DNN adaptation through main metrics (i.e., the accuracy and miss rates, as defined in Sect. 7.1) because of the interplay between various factors that affect these metrics. For example, a slight difference in the latency profiles between static and dynamic DNNs can lead to the selection of different DNN variants for execution, affecting the accuracy and/or end-to-end latency of clients. Hence, we define a specific metric called *mismatch count*. The mismatch count metric indicates the number of newly arrived inference requests (or frames) executed using the old DNN variant due to the delay in the DNN adaptation process. Such a mismatch may negatively affect the miss rate when new frames are executed with the old but bigger DNN variant, and it may degrade accuracy when new frames are executed with the old but smaller DNN variant.

Figure 19 shows the number of mismatches observed when static DNNs are used. We do not plot numbers for the dynamic DNNs, as all values are almost always zero. In the case of static DNNs, the larger SLOs have a higher mismatch count because they provide more room for DNN adaptation. The mismatch count is further exacerbated when the aggregate request rate increases due to the increase in the number of clients or their inference request rates. Furthermore, the mismatch count is higher under the LTE trace because of the low bandwidth values (median value 14.07 Mbps) and comparatively large variations.

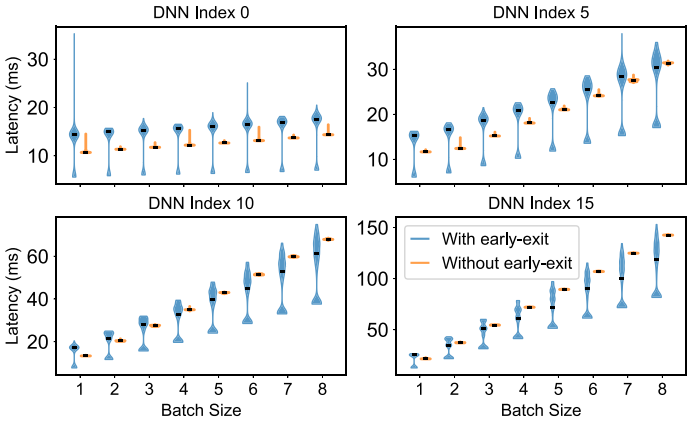


Fig. 20 The performance of DNN variants with early-exit technique compared to the no early-exit DNN variants. We sample four DNN variants uniformly from the set of 16 DNN variants

On the other hand, the mismatch count for dynamic DNNs is zero for almost all settings. When the mismatch count is not zero, the value is small and negligible compared to that with the static DNNs. For example, for the worst-case setting with 4 clients, 150 ms SLO and 30 FPS under the LTE trace, the mismatch count for dynamic DNNs is only 64, whereas the mismatch count for static DNNs is 953. The small mismatch count in dynamic DNNs can be attributed to the way Jellyfish handles pending requests in the queue before adapting it to the new active DNN variant. Specifically, Jellyfish waits for sufficient requests to arrive (to fill the batch) before it can drain the pending requests with the old DNN variant, leading some new requests to be executed with the old DNN variant.

In summary, the negligible mismatch count shows that the dynamic DNN *quickly* adapts to the DNN variant desired by the scheduler of inference serving systems and removes the need for DNN caching.

8.3 Early-exit performance

We add a small transformer block (an object detection head) with only three encoder and decoder layers as an exit branch to the dynamic OFA-ResNet50-DETR model (see Fig. 9). This exit branch is inserted after the second ResNet block of the OFA-ResNet50-DETR DNN and is reused for all 16 DNN variants of the dynamic DNN. To simulate different early-exit scenarios, we assign exit probabilities to the first exit branch of 16 DNN variants uniformly from the synthetic set {0.25, 0.30, 0.35, 0.40}, indicating the percentage of requests exiting from that branch. The smaller (bigger) DNN variants have lower (higher) exit probabilities.

We first evaluate the execution latency of four DNN variants sampled uniformly from the OFA-ResNet50-DETR model. The exit probabilities of these four DNN variants at the first exit branch are 0.25, 0.30, 0.35, 0.40, respectively. We run each DNN variant for eight different batch sizes and for 1K iterations per batch size.

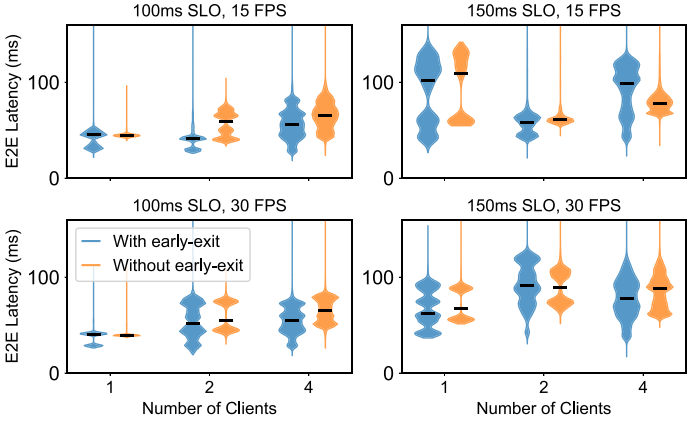


Fig. 21 End-to-end latency observed when DNN variants are deployed with and without early-exit technique for varying SLOs, request rates (FPS) and numbers of clients under a WiFi network trace

The violin plot in Fig. 20 shows the performance of four sampled DNN variants with and without the early-exit technique. DNN variants without the early-exit branch have steady execution latency and, thus, have only one density area centred around a very narrow range. On the other hand, DNN variants with the early-exit branch have two density areas displaying the latency timings of requests exiting at two exit branches. The median latency is higher for smaller DNNs or smaller batch sizes because of the extra time required to execute the additional exit branch. For bigger DNNs and larger batch sizes, the median execution latency of the early-exit DNN variant is lower than that without early-exit. This is because bigger DNNs have higher exit probabilities, and thus more requests exit at the first branch, causing a significant reduction in the overall batch inference time. Further, even under low exit probabilities (such as 0.30 for DNN index 5), the reductions are noteworthy at large batch sizes. This demonstrates a potential for a substantial reduction in batch inference time even when only a few requests exit (low exit probability) with large batch sizes, thereby amortizing the overall inference serving latency.

We now evaluate the performance by incorporating early-exit DNN variants in Jellyfish. We test different experimental settings with varying numbers of clients, SLOs, and request rates under the WiFi trace. We choose WiFi trace as it has higher bandwidth (median value 53.1Mbps) than the LTE one, allowing the scheduler to select bigger DNN variants that can benefit from the early-exit performance.

Figure 21 shows the end-to-end latency per request with and without the early-exit branch. In many settings, the median end-to-end latency in the case of early-exit DNN variants is lower than that of the case without the early-exit branch. However, we observe that this does not always indicate that the early-exit technique offers low end-to-end latency. This is because the 16 DNN variants have different latency profiles with and without the early-exit branch. The early-exit DNN variants have higher latency profile values due to the additional time spent at the first early-exit branch. As a result, for the early-exit case, the scheduler may select smaller DNN variants leading to lower execution latency. Hence, the selection of smaller DNN variants

can also potentially lower the median value, making it hard to isolate the actual reason for the improvement. We leave the detailed analysis to future exploration.

9 Further discussion and limitations

9.1 Jellyfish

This section discusses the limitations and directions for future work to improve Jellyfish further.

Request rate adaptation Similar to Chameleon (Jiang et al. 2018) and DeepDecision (Ran et al. 2018), 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.

Accuracy optimization Jellyfish aims to optimize for accuracy by selecting the best possible DNN variants from the collection (bag) of diverse DNNs for the current system conditions. Therefore, we expect that the *effective* accuracy achieved by the clients (assuming they are successfully mapped to DNNs) will be *bounded*. The smallest DNN and the largest DNN in the collection determine the minimum and maximum accuracy values. The degradation of accuracy depends on the distribution (spread) of accuracy values (profiles) among the DNN models in the collection. When the minimum accuracy (determined by the smallest DNN model) is *not* acceptable to the broad range of users, the system designer can attempt to optimize the serving by improving the minimum accuracy and narrowing down the spread in the accuracy by choosing the collection of DNN variants prudently. One can select the collection of fine-grained DNN models - easily achievable using dynamic DNNs - to increase the space of DNN models and narrow the accuracy spread.

Accuracy constraint Jellyfish can easily support the addition of constraints on the desired *minimum* accuracy per client. Our scheduler can be extended to support such accuracy constraints by setting the latency budgets of clients to zero for DNNs whose accuracy is lower than the desired value. However, clients may not be mapped to any DNN variant during runtime if they provide a greater desired value on the accuracy constraint. When clients cannot be mapped to any DNN

due to either accuracy constraints or stricter latency or extremely low bandwidth conditions, we must implement a component on the client side to execute requests locally, provided that clients have sufficient compute resources for inference with optimized/compressed DNNs.

Client adapter As mentioned in Sect. 3.2, Jellyfish requires client support to facilitate data adaptation, bandwidth estimation, and metadata piggybacking, which is typical for adaptive-video-analytics systems. Hence, providing client-side adapters or skeleton code for multiple languages will help ease the application development.

Unreliable communication network In this paper, we do not explicitly tackle the unreliability of the communication network. However, the client and server communication in our current implementation is handled using gRPC. We, therefore, rely on the gRPC-TCP's reliability feature to manage message loss/drop conditions. There can be delays in message delivery which can delay the data adaptation process with the correct size. Such delays can then lead to a mismatch between the data size and the DNN size on the server, subsequently impacting the application performance. However, we anticipate that these delays will be considered when estimating the bandwidth, and therefore the Jellyfish scheduler will handle such conditions in the next scheduling round. Additionally, we piggyback the metadata (e.g., data/input size) on every response to the client. As a result, clients can recover from unreliable conditions as soon as the connection stabilizes.

9.2 Dynamic DNNs

In this section, we discuss the limitations and future work of our proposal on leveraging dynamic DNNs to avoid DNN switching costs and improve the performance of batched inference.

Controlled experiments We evaluated our proposal for combining the network pruning and early-exit techniques in the Jellyfish framework. However, as discussed in the evaluation section, it is hard to isolate the actual reason for the performance improvement because multiple factors in Jellyfish affect the final outcome. Therefore, it is worth evaluating our proposal in other serving systems where more controlled experiments can be conducted more conveniently.

Joint DNN training As our preliminary evaluation is focused primarily on the execution efficiency of the proposed approach, we did not train all the DNN variants and their early-exit branches. It is important to study the effectiveness of the training strategy proposed in Sect. 5.2. Such a study will highlight the challenges, complexity and cost involved in training jointly all the variants. Such a study will also help us in understanding the accuracy and exit probabilities achieved at each exit branch.

Early-exit decision making As discussed in Sect. 5.3, we evaluated our learning-based module for exit decision-making exclusively on one DNN variant with two exit branches, where 30.94% of requests exit at the first exit branch with a minimal drop in accuracy. It is important to study how well our learning-based module performs across all DNN variants.

Early-exit APIs Widely used deep learning frameworks like PyTorch and TensorFlow do not provide any APIs for the early-exit implementation and their batched inference execution. Instead, the current APIs execute all requests through the entire network and exit them simultaneously. Therefore, it is necessary to explore the API specifications needed for early-exit networks so that any arbitrary requests can exit in the middle of the network with low performance overhead.

Accuracy gains through enlargement On one hand, the early-exit technique offers accelerated execution of batched inference. On the other hand, it allows the remaining requests in the batch to use more compute resources that could then be used to enlarge the remaining portion of the DNN variant in an effort to increase accuracy. Therefore, it is worth investigating the gains in accuracy achieved through the enlargement of the portion of the DNN.

10 Related Work

Adaptive video analytics systems Recent works such as VideoStorm (Zhang et al. 2017), AWStream (Zhang et al. 2018a), Chameleon (Jiang et al. 2018), DeepDecision (Ran et al. 2018), JCAB (Wang et al. 2020), DDS (Du et al. 2020), and SPINN (Laskaridis et al. 2020a) 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 (Zeng et al. 2020). 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 (Crankshaw et al. 2017) provides an easy-to-use abstraction layer for low-level deep learning frameworks. Nexus (Shen et al. 2019) aims to optimize serving throughput without SLO violations. Clockwork (Gujarati et al. 2020) 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 (Crankshaw et al. 2020), Llama (Romero et al. 2021b), and FA2 (Razavi et al. 2022) optimize the serving of complex DNN pipelines. INFaas (Romero et al. 2021a) automates the hardware and model-variant selection and deployment through managed services. Model-Switching (Zhang et al. 2020a) 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 (Olston et al. 2017), Torch Serve (Pytorch 2021), and Triton Inference Server (NVIDIA 2021) 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 (Nigade et al. 2021) or optimize resources with relatively lenient latency constraints (i.e., 1–5s) (Jiang et al. 2021). In contrast, Jellyfish maximizes inference accuracy with *millisecond-level* latency SLO targets given a highly dynamic network.

Dynamic DNNs Recently, neural architecture search (NAS) based methods such as ProxylessNAS (Cai et al. 2019), MnasNet (Tan et al. 2019), and OFA (Cai et al. 2020) have been proposed to generate dynamic (or static) sub-networks by searching through the large architecture space. These methods complement our idea of using multiple DNN variants (sub-networks) for DNN adaptation in inference serving systems, provided they maintain efficient parameter sharing and architecture switching between the sub-networks. While previous works like BranchyNet (Teerapittayanon et al. 2016) have focused on improving computational efficiency in DNNs through early-exit techniques, they are limited to tasks such as classification (Teerapittayanon et al. 2016; Laskaridis et al. 2020b), segmentation (Li et al. 2017), or text generation (Schwartz et al. 2020). Our design of the dynamic DNN extends the early-exit technique to the object detection task by utilizing a lightweight transformer-based exit branch. Additionally, the performance of batched inference in the context of early-exit networks has not been extensively studied, except for a few recent exceptions like DVABatch (Cui et al. 2022), PAME (Zhang et al. 2022), and Fluid Batching (Kouris et al. 2022). So far, not much work has been done on combining the network pruning and early-exit techniques. Recently, Görmez and Koyuncu (2022) evaluated pruning methods for early-exit networks to generate a smaller yet static early-exit network. To the best of our knowledge, the idea of combining these two techniques within a serving system to achieve smooth DNN adaptation with efficient batched inference is novel. Moreover, using transformer-based vision models opens up the possibility of applying this combination to the object detection task.

11 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. We also present ideas to design and integrate dynamic DNNs in Jellyfish to accommodate more efficient DNN switching and batched execution. Our preliminary evaluation demonstrates the potential of integrating dynamic DNNs to further enhance Jellyfish.

Acknowledgements We would like to thank the editor and anonymous reviewers for their valuable comments and suggestions. This work is part of the Efficient Deep Learning (EDL) programme (grant number P16-25), funded by the Dutch Research Council (NWO), and the Real-Time Video Surveillance Search project (grant number 18038), also supported by the Dutch Research Council (NWO). We would also like to thank NVIDIA for the generous donation of two A30 Tensor Core GPUs through their academic hardware grant program.

References

Aarts EHL, Korst JHM (1990) Simulated annealing and Boltzmann machines—a stochastic approach to combinatorial optimization and neural computing. Wiley, New Jersey

Ahmad F, Qiu H, Eells R et al (2020) Carmap: fast 3d feature map updates for automobiles. USENIX NSDI, Santa Clara, pp 1063–1081

Ali AJB, Hashemifar ZS, Dantu K (2020) Edge-slam: edge-assisted visual simultaneous localization and mapping. ACM MobiSys, New York, pp 325–337

Ananthanarayanan G, Bahl P, Bodík P et al (2017) Real-time video analytics: the killer app for edge computing. Computer 50:58–67

Bhardwaj R, Xia Z, Ananthanarayanan G et al (2022) Ekya: continuous learning of video analytics models on edge compute servers. USENIX NSDI, Santa Clara, pp 119–135

Bochkovskiy A, Wang C, Liao HM (2020) Yolov4: optimal speed and accuracy of object detection. arXiv arXiv:2004.10934

Braun M, Mainz A, Chadowitz R et al (2019) At your service: designing voice assistant personalities to improve automotive user interfaces. ACM CHI, Boston, p 40

Cai H, Zhu L, Han S (2019) Proxylessnas: direct neural architecture search on target task and hardware. ICLR, Vienna

Cai H, Gan C, Wang T et al (2020) Once-for-All: train one network and specialize it for efficient deployment. ICLR, Vienna

Carion N, Massa F, Synnaeve G et al (2020) End-to-end object detection with transformers. ECCV, Glasgow, pp 213–229
Chen Z, Hu W, Wang J et al (2017) An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance. ACM/IEEE SEC, Wilmington, p 14:1-14:14
Cheng Y, Wang D, Zhou P et al (2018) Model compression and acceleration for deep neural networks: the principles, progress, and challenges. IEEE Signal Process Mag 35:126–136
Crankshaw D, Wang X, Zhou G et al (2017) Clipper: a low-latency online prediction serving system. USENIX NSDI, Santa Clara, pp 613–627
Crankshaw D, Sela G, Mo X et al (2020) InferLine: latency-aware provisioning and scaling for prediction serving pipelines. ACM SoCC, Seattle, pp 477–491
Cui W, Zhao H, Chen Q et al (2022) DVABatch: diversity-aware multi-entry multi-exit batching for efficient processing of DNN services on gpus. USENIX ATC, Boston, pp 183–198
Dai D, Wang Y, Chen Y et al (2016) Is image super-resolution helpful for other vision tasks? IEEE WACV, Snowmass, pp 1–9
Du K, Pervaiz A, Yuan X et al (2020) Server-driven video streaming for deep learning inference. ACM SIGCOMM, New York, pp 557–570
Görmez A, Koyuncu E (2022) Pruning early exit networks. CoRR arXiv:2207.03644
Gujarati A, Karimi R, Alzayat S et al (2020) Serving dnns like clockwork: performance predictability from the bottom up. USENIX OSDI, Berkeley, pp 443–462
Han S, Shen H, Philipose M et al (2016) MCDNN: an approximation-based execution framework for deep stream processing under resource constraints. ACM MobiSys, New York, pp 123–136
Han Y, Huang G, Song S et al (2022) Dynamic neural networks: a survey. IEEE Trans Pattern Anal Mach Intell 44:7436–7456
He K, Zhang X, Ren S et al (2016) Deep residual learning for image recognition. IEEE CVPR, Seattle, pp 770–778
Heo S, Cho S, Kim Y et al (2020) Real-time object detection system with multi-path neural networks. IEEE RTAS, San Antonio, pp 174–187
Hinton GE, Vinyals O, Dean J (2015) Distilling the knowledge in a neural network. CoRR arXiv:1503.02531
Huang J, Qian F, Gerber A et al (2012) A close examination of performance and power characteristics of 4g LTE networks. ACM MobiSys, New York, pp 225–238
Jiang J, Ananthanarayanan G, Bodík P et al (2018) Chameleon: scalable adaptation of video analytics. ACM SIGCOMM, New York, pp 253–266
Jiang J, Luo Z, Hu C et al (2021) Joint model and data adaptation for cloud inference serving. IEEE RTSS, Houston, pp 279–289
Kannan T, Hoffmann H (2021) Budget rnns: multi-capacity neural networks to improve in-sensor inference under energy budgets. IEEE RTAS, San Antonio, pp 143–156
Kannan RS, Subramanian L, Raju A et al (2019) GrandSLAm: guaranteeing slas for jobs in microservices execution frameworks. ACM EuroSys, New York, p 34:1-34:16
Kouris A, Venieris SI, Laskaridis S, et al (2022) Fluid batching: Exit-aware preemptive serving of early-exit neural networks on edge npus. CoRR arXiv:2209.13443
Krizhevsky A, Hinton G, et al (2009) Learning multiple layers of features from tiny images. CoRR
Laskaridis S, Venieris SI, Almeida M et al (2020) SPINN: synergistic progressive inference of neural networks over device and cloud. ACM MobiCom, Los Cabos, p 37:1-37:15
Laskaridis S, Venieris SI, Kim H et al (2020) HAPI: hardware-aware progressive inference. IEEE/ACM ICCAD, NewYork, pp 1–9
Laskaridis S, Kouris A, Lane ND (2021) Adaptive inference through early-exit networks: design, challenges and directions. ACM EMDLMobiSys, New York, pp 1–6
Lee S, Nirjon S (2020) SubFlow: a dynamic induced-subgraph strategy toward real-time DNN inference and training. IEEE RTAS, San Antonio, pp 15–29
Li X, Liu Z, Luo P et al (2017) Not all pixels are equal: difficulty-aware semantic segmentation via deep layer cascade. IEEE CVPR, Seattle, pp 3193–3202
Lin T, Maire M, Belongie SJ et al (2014) Microsoft COCO: common objects in context. ECCV, Zurich, pp 740–755
Liu L, Li H, Gruteser M (2019) Edge assisted real-time object detection for mobile augmented reality. ACM MobiCom, Los Cabos, p 25:1-25:16
Matsubara Y, Levorato M, Restuccia F (2023) Split computing and early exiting for deep learning applications: survey and research challenges. ACM Comput Surv 55:1–30

Nigade V, Winder R, Bal HE et al (2021) Better never than late: timely edge video analytics over the air. ACM SenSys, New York, pp 426–432
Nigade V, Bauszat P, Bal H et al (2022) Jellyfish: timely inference serving for dynamic edge networks. IEEE RTSS, Houston, pp 277–290
NVIDIA (2021) NVIDIA Triton Inference Server. https://developer.nvidia.com/nvidia-triton-inference-server
Olston C, Fiedel N, Gorovoy K, et al (2017) Tensorflow-serving: flexible, high-performance ML serving. arXiv arXiv:1712.06139
Pytorch (2021) TorchServe. https://pytorch.org/serve/
PyTorch (2022) Reproducibility. https://pytorch.org/docs/stable/notes/randomness.html
Qu Z, Sarwar SS, Dong X, et al (2022) DRESS: dynamic real-time sparse subnets. CoRR arXiv:2207.00670
Ran X, Chen H, Zhu X et al (2018) DeepDecision: a mobile deep learning framework for edge video analytics. IEEE INFOCOM, New Jersey, pp 1421–1429
Razavi K, Luthra M, Koldehofe B et al (2022) FA2: fast, accurate autoscaling for serving deep learning inference with SLA guarantees. IEEE RTAS, San Antonio, pp 146–159
Ren S, He K, Girshick RB et al (2017) Faster R-CNN: towards real-time object detection with region proposal networks. IEEE Transactions on Pattern Analysis and Machine Intelligence. IEEE, New Jersey, pp 1137–1149
Romero F, Li Q, Yadwadkar NJ et al (2021) INFaaS: automated model-less inference serving. USENIX ATC, Boston, pp 397–411
Romero F, Zhao M, Yadwadkar NJ, et al (2021b) Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines. arXiv arXiv:2102.01887
Rusci M, Capotondi A, Benini L (2020) Memory-driven mixed low precision quantization for enabling deep network inference on microcontrollers. MLSys, Austin, pp 326–335
Samplawski C, Marlin BM (2021) Towards transformer-based real-time object detection at the edge: a benchmarking study. IEEE MILCOM, Boston, pp 898–903
Schwartz R, Stanovsky G, Swayamdipta S et al (2020) The right tool for the job: matching model and instance complexities. ACL, Dublin, pp 6640–6651
Shen H, Chen L, Jin Y et al (2019) Nexus: a GPU cluster engine for accelerating DNN-based video analysis. ACM SOSP, New York, pp 322–337
Sreedhar K, Clemons J, Venkatesan R, et al (2022) Enabling and accelerating dynamic vision transformer inference for real-time applications. CoRR arXiv:2212.02687
Svoboda F, Fernández-Marqués J, Liberis E et al (2022) Deep learning on microcontrollers: a study on deployment costs and challenges. In: Yoneki E, Nardi L (eds) EuroSys'22. ACM EuroMLSys, New York, pp 54–63
Szegedy C, Ioffe S, Vanhoucke V et al (2017) Inception-v4, inception-ResNet and the impact of residual connections on learning. AAAI, Washington, pp 4278–4284
Tan M, Chen B, Pang R et al (2019) Mnasnet: platform-aware neural architecture search for mobile. IEEE CVPR, Long Beach, pp 2820–2828
Teerapittayanon S, McDanel B, Kung HT (2016) Branchynet: fast inference via early exiting from deep neural networks. ICPR, New York, pp 2464–2469
Tianxiaomo (2020) Pytorch-yolov4. https://github.com/Tianxiaomo/pytorch-YOLOv4
van der Hooft J, Petrangeli S, Wauters T et al (2016) Http/2-based adaptive streaming of HEVC video over 4g/lte networks. IEEE Commun Lett 20:2177–2180
Wan C, Santriaji MH, Rogers E et al (2020) ALERT: accurate learning for energy and timeliness. USENIX ATC, Boston, pp 353–369
Wang E, Davis JJ, Zhao R et al (2019) Deep neural network approximation for custom hardware: where we've been, where we're going. ACM Comput Surv 52:40:1-40:39
Wang C, Zhang S, Chen Y et al (2020) Joint configuration adaptation and bandwidth allocation for edge-based real-time video analytics. IEEE INFOCOM, New York, pp 257–266
Xu D, Zhou A, Zhang X et al (2020) Understanding operational 5G: a first measurement study on its coverage, performance and energy consumption. ACM SIGCOMM, New York, pp 479–494
Yin X, Jindal A, Sekar V et al (2015) A control-theoretic approach for dynamic adaptive video streaming over HTTP. ACM SIGCOMM, New York, pp 325–338

Yu F, Wang D, Shangguan L, et al (2022) A survey of multi-tenant deep learning inference on GPU. arXiv:2203.09040
Zeng X, Fang B, Shen H et al (2020) Distream: scaling live video analytics with workload-adaptive distributed edge intelligence. ACM SenSys, New York, pp 409–421
Zhang H, Ananthanarayanan G, Bodík P et al (2017) Live video analytics at scale with approximation and delay-tolerance. USENIX NSDI, Berkeley, pp 377–392
Zhang B, Jin X, Ratnasamy S et al (2018) AWStream: adaptive wide-area streaming analytics. ACM SIGCOMM, New York, pp 236–252
Zhang T, Ye S, Zhang K et al (2018) A systematic DNN weight pruning framework using alternating direction method of multipliers. ECCV, Munich, pp 191–207
Zhang J, Elnikety S, Zarar S et al (2020) Model-switching: dealing with fluctuating workloads in machine-learning-as-a-service systems. USENIX HotCloud, Berkeley
Zhang X, Lu H, Hao C et al (2020) SkyNet: a hardware-efficient method for object detection and tracking on embedded systems. MLSys, Austin, pp 216–229
Zhang S, Cui W, Chen Q et al (2022) PAME: precision-aware multi-exit DNN serving for reducing latencies of batched inferences. ACM ICS, New York, pp 1–12
Zhou Y, Moosavi-Dezfooli S, Cheung N et al (2018) Adaptive quantization for deep neural network. AAAI, Washington, pp 4596–4604
Zhu X, Su W, Lu L et al (2021) Deformable DETR: deformable transformers for end-to-end object detection. ICLR, Vienna

Vinod Nigade is a postdoctoral researcher in the High Performance Distributed Computing group at Vrije Universiteit Amsterdam. His research spans various facets of computer systems, such as networked systems, machine learning systems, video analytics systems, and distributed systems of intermittent [battery-less] devices. He holds a PhD and MSc from Vrije Universiteit Amsterdam and received an Outstanding Paper Award at IEEE RTSS 2022. Vinod also has four years of industrial experience in computer systems, predominantly in distributed and data storage systems.

Pablo Bauszat received the Ph.D. degree in computer science from TU Braunschweig, Germany in 2015. He was a Post-Doctoral Researcher at the Computer Graphics and Visualization Group, Delft University of Technology, The Netherlands, and a Software Engineer at Google Research Zurich, Switzerland. Currently, he is a Scientific Programmer in the High Performance Distributed Computing Group at the Vrije Universiteit Amsterdam, The Netherlands. He published in a variety of research domains including computer graphics and visualization, computer vision, machine learning, and networking.

Henri Bal is a full professor of Computer Science at the Vrije Universiteit, where he leads the High Performance Distributed Computing group. He is a member of the Academia Europeana and winner of the Euro-Par 2014 Achievement Award. He has been Program Chair of CCGrid and HPDC and PC member of numerous conferences.

Lin Wang is currently a Full Professor and Head of the Chair of Computer Networks in the Department of Computer Science at Paderborn Univesity. He received his Ph.D. degree from the Institute of Computing Technology, Chinese Academy of Sciences in 2015. Previously, he held positions at Vrije Universiteit Amsterdam, TU Darmstadt, SnT Luxembourg, and IMDEA Networks Institute. He is broadly interested in networked systems, with a focus on in-network computing and intermittently-powered IoT systems. He has received a Google Research Scholar Award, an Outstanding Paper Award of RTSS 2022, a Best Paper Award of IPCCC 2023, and an Athene Young Investigator Award of TU Darmstadt. He is a Senior Member of IEEE.