Dynamic Resource Allocation for Deadline-Constrained Neural Network Training

Luciano Baresi, Marco Garlini, Giovanni Quattrocchi
Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria
{name.surname}@polimi.it

***Abstract*—Neural networks (NN) serve as the backbone for various applications, including computer vision, speech recognition, and natural language processing. Due to their iterative nature, training NNs is a highly compute-intensive task that is typically executed using a statically allocated set of devices (e.g., CPUs or GPUs). This static allocation prevents adjusting priorities, making it impossible to reassign resources to urgent tasks, and potentially causing high-priority training jobs to miss their expected completion times.**

This paper proposes *DECOR-NN* (DEadline COnstrained Resource allocation for Neural Networks), a control mechanism for NN training that dynamically allocates resources according to a user-defined deadline (i.e., a Service Level Agreement), ensuring that the training phase completes within the specified time. The solution leverages control theory and has been developed on top of PyTorch, a widely-used framework for training NNs. *DECOR-NN* dynamically allocates either GPUs or fractions of CPUs to meet user deadlines and also allows users to modify the deadline at runtime to accommodate changes in job priorities. A comprehensive empirical evaluation using three benchmark applications demonstrates that *DECOR-NN* successfully completes training jobs with an average deviation from the deadline of only 1.75%.

***Index Terms*—Neural Networks, Dynamic Resource Allocation, GPU, Control Theory, PyTorch**

I. Introduction

Neural networks (NN) have become the cornerstone of modern artificial intelligence, underpinning a wide array of applications such as computer vision [1], speech recognition [2], and natural language processing [2]. These applications rely on the ability of NN to learn from data through a process known as training, which involves adjusting the network's parameters to minimize a loss function. Once training is completed, models are embedded into services and operate in inference mode, providing real-time predictions in response to user requests.

The training process is iterative and computationally intensive, often requiring substantial computational resources and time [3]. Having Service Level Agreements (SLAs) for NN training is crucial to ensure predictable and timely completion of training tasks. Such requirements are usually encoded as deadlines, that is, the maximum allowed execution time to complete the training job [4]. These deadlines allow for better resource planning and management, enabling organizations to meet specific performance and delivery expectations [5].

Existing industrial frameworks for training NN, such as PyTorch[1] and TensorFlow[2] enable users to allocate a set of available devices, such as CPUs or GPUs, before starting the training phase. However, they do not allow for setting deadlines and the allocation is only static. Static allocation cannot be altered at run-time, meaning that once devices are assigned to a task, they remain occupied for the entirety of the execution. If we think about a platform that allows users to train their NN on a shared infrastructure, static allocation prevents any adjustment in priority for a training job, making it impossible to reallocate resources dynamically. As a result, if a high-priority training job is scheduled to be executed, it cannot reclaim the allocated devices, potentially leading to deadline violations.

Moreover, NN-enabled applications could be executed on both powerful cloud nodes equipped with high-end GPUs and also on resource-contrained devices with only CPUs available. For example, *TinyML* approaches focus on optimizing the models to allow low-power embedded devices to run intelligent NN-based applications [6]. The heterogeneous nature of available devices, models, and applications further complicates the accurate prediction of the resources required to meet the SLAs, making static allocation even less suitable for ensuring the efficient execution of training jobs.

Some approaches in the literature [7], [8], [9] focus on managing NN-enabled applications in *inference mode*, allocating CPU or GPU resources – either statically or dynamically – to meet defined Service Level Agreements on response time. However, handling NN training presents different challenges. Although the inference mode is interactive and requests can be served in the order of hundreds of milliseconds, the training process is long lasting, and each execution requires substantial computational resources and time [3]. Few approaches in the literature have focused on handling the training phase for cloud environments [10] or edge devices (i.e., *edge learning*) [11], [12], [13]. For example, the approaches presented by Gu et al. [14] and Shi et al. [15] allow software engineers to set deadlines to constrain NN training jobs. However, these approaches primarily focus on optimizing the scheduling of training jobs while relying solely on static resource allocation. To our knowledge, no existing solution comprehensively addresses the dynamic allocation of CPU and GPU resources to effectively meet training deadlines.

To address these limitations, we propose *DECOR-NN* (DEadline COnstrained Resource allocation for Neural Networks), a novel mechanism based on control theory to dynamically allocate resources for deadline-constrained training jobs. *DECOR-NN* provides a dynamic allocation strategy for the NN-based

[1]https://pytorch.org
[2]https://www.tensorflow.org

application in training mode, which adjusts the allocation of resources in real time based on the current state of the training process and the remaining time until the deadline. Thus, *DECOR-NN* allows to balance resource availability and training speed, ensuring timely completion without overburdening the computational infrastructure. Additionally, *DECOR-NN* allows users to modify deadlines at run-time, providing flexibility in managing training jobs. This enables the system to adapt to evolving requirements, accelerating or decelerating the training processes as needed. Finally, a prototype of *DECOR-NN* has been built on top of PyTorch, a widely-used framework for training NN. The system is capable of allocating either GPUs or CPUs, providing a flexible solution to meet deadlines across different types of devices.

To validate *DECOR-NN*, we conducted a comprehensive empirical evaluation using three benchmark applications run on Amazon Web Services (AWS). The results demonstrate that *DECOR-NN* can successfully terminate training jobs within the requested deadlines, maintaining acceptable margins of error, even when deadlines are changed at runtime.

In summary, our contributions are as follows.

- **Dynamic Resource Allocation for NN training:** we introduce *DECOR-NN*, a mechanism to dynamically allocate CPU or GPU resources to meet NN training deadlines.
- **Runtime Deadline Adjustments:** *DECOR-NN* allows users to modify deadlines on the fly, providing flexibility in the management of training jobs.
- **Heterogeneous Device Support:** we integrated *DECOR-NN* into PyTorch, enabling the use of GPUs and CPUs to accommodate a wide range of devices.

The rest of this paper is organized as follows. Section II introduces the problem addressed by our work and a motivating scenario. Section III introduces the high-level model of *DECOR-NN*, while Section IV discusses its implementation. Section V presents the empirical evaluation, while Section VI presents the threats to the validity. Section VII surveys the related work, and Section VIII concludes the paper.

II. Problem Statement and Motivating Scenario

The proliferation of Deep Learning technologies has led to their integration into a diverse array of applications that rely on complex NN to implement intelligent behavior. Managing the training processes of these NN in heterogeneous and dynamic environments presents significant challenges, primarily due to the dynamic nature of workloads, varying priority levels, and the need for efficient resource utilization. Traditional static resource allocation strategies are inadequate in addressing these challenges, as they cannot adapt to the fluctuating demands of intelligent applications in real-time. This rigidity often results in an inefficient use of computational resources, increased operational costs, and the inability to meet stringent deadlines essential for the reliable performance of AI-driven systems [16].

To illustrate these challenges, consider a smart city ecosystem comprising various mobile clients, including smart buildings and mobile phones. Smart buildings continuously monitor environmental conditions and adjust energy consumption to maintain optimal comfort levels for occupants. This requires frequent updates to their predictive models based on real-time data, necessitating on-demand training and fine-tuning of NN. Mobile phones, on the other hand, run applications that personalize user experiences by fine-tuning models based on individual usage patterns, while ensuring data privacy by processing information locally. In this scenario, different applications generate training jobs with varying priorities and deadline constraints.

Additionally, the computational resources available across these devices range from powerful GPUs in cloud servers to limited CPUs in edge devices, further complicating efficient resource allocation. The dynamic nature of these workloads, where training demands can surge unpredictably based on real-time data and application requirements, requires a flexible and adaptive resource management system. Such a system must intelligently allocate and adjust resources in real-time to accommodate high-priority tasks while efficiently utilizing available computational power to handle lower-priority jobs without compromising overall system performance. In this context, energy efficiency is a critical concern, especially for edge devices operating on limited power sources [17]. Balancing the need for rapid training with energy conservation requires sophisticated mechanisms that can dynamically scale resource usage based on current demands and priorities.

Another crucial aspect is the ability to dynamically adjust deadlines. For example, in a smart building, a training job for the energy optimization model might be running with a standard deadline. However, if the system detects an increase in resource contention, such as during peak operational hours when HVAC systems and other energy-intensive devices are heavily used, the training job deadline can be extended to temporarily reduce resource usage. This ensures that critical real-time operations maintain priority while still allowing the model to complete its training efficiently. In the case of a mobile app recommendation system, training jobs may initially be scheduled with a fixed deadline to update models regularly. However, if unexpected cost constraints arise, such as a sudden spike in cloud infrastructure prices, the system could extend the deadline to shift resource usage to off-peak hours, minimizing costs without significantly affecting the quality of recommendations. In contrast, if resource availability increases unexpectedly, the deadline could be shortened to leverage additional capacity and accelerate model update.

Handling such dynamic deadline changes introduces additional layers of complexity. The resource management system must not only allocate resources efficiently under normal conditions, but also quickly adapt to shifting priorities and updated deadlines without causing significant disruptions or delays. This requires a self-adaptive mechanism capable of monitoring the current state of training processes and reallocating resources in real-time to meet new deadlines. Static resource allocation strategies lack the responsiveness needed to handle these scenarios, leading to potential deadline violations,

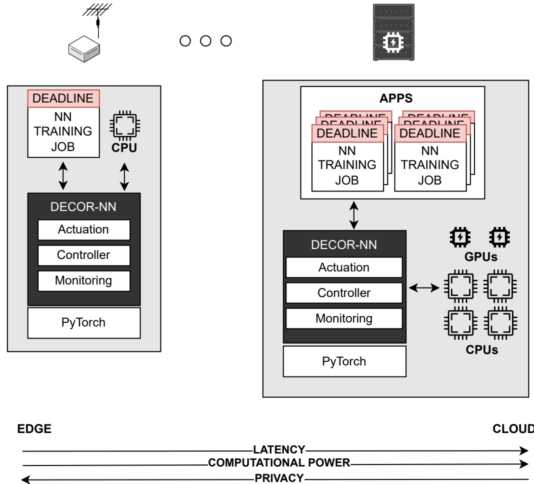

Fig. 1. *DECOR-NN* deployed on an edge and cloud nodes.

inefficient resource usage, and compromised performance of intelligent applications.

Addressing these multifaceted challenges calls for an innovative approach to ensure that training jobs are completed within their specified deadlines, adapt to varying priority levels, and optimize resource utilization across heterogeneous devices. *DECOR-NN* represents a significant first step in this direction. By enabling real-time adjustments to both CPU and GPU allocations, it ensures efficient resource utilization while adhering to deadlines for single jobs. This sets the groundwork for future extensions that can support the execution of multiple concurrent jobs on a shared infrastructure. Such extensions will be able to fully address the scenarios envisioned, enabling dynamic and efficient resource management in complex and heterogeneous systems.

III. *DECOR-NN*

DECOR-NN is a solution for dynamic resource allocation in NN training, with the aim of addressing the aforementioned challenges.

Figure 1 depicts *DECOR-NN* deployed on two nodes that represent varying levels of computational capacity: an edge node equipped with a single CPU (on the left) and a powerful cloud node featuring four CPUs and two GPUs (on the right). On the edge node, *DECOR-NN* handles a single NN training job running exclusively on the CPU. On the cloud node, *DECOR-NN* manages multiple training jobs, each with distinct deadlines with the presence of GPUs, enabling accelerated processing for some of the training jobs.

The figure says that *DECOR-NN* is composed of three main components that work on top of an existing framework for training NNs (e.g., PyTorch): (i) a *monitoring system* that measures the progress of training jobs, and (ii) a *controller*, that computes the computational resources required to finish before the deadline, and (iii) an actuation system that takes the output of the controller and actuates it in the underlying infrastructure. Although the first two components are implementation-independent and described in the rest of this section, the actuator depends on the underlying framework and the type of training hardware used (i.e., a CPU or a GPU). Thus, we provide further details on the actuation system in Section IV where we describe our prototype.

DECOR-NN is designed to manage resources with the finest possible granularity, allowing CPU cores to be allocated in fractions. Although mechanisms for GPU usage with fine granularity exist, such as MIG (Multi-Instance GPU) and MPS (Multi-Process Service) [18], these approaches are either limited to specific GPU models or primarily handle memory allocation, rather than offering true fractional computational resources (more details on this topic are provided in Section IV-B). Control granularity is thus heterogeneous and depends on the resources at hand: fractions for CPUs and units for GPUs.

Given the highly dynamic nature of the execution environment, the control system also needs to be responsive to changes in the foreseen progress rate and the originally requested deadline. This flexibility mimics real-world scenarios where training priorities can shift, requiring the system to allocate fewer resources for extended deadlines and more resources for shorter ones. By dynamically allocating resources and allowing one to adjust deadlines, *DECOR-NN* ensures it can handle varying demands and maintain efficient resource usage in highly dynamic environments.

Finally, *DECOR-NN* has been designed to complete the training within the specified time frame, but without allocating too many resources. If set deadlines are feasible, *DECOR-NN* allocates resources to complete the training task as close as possible to the deadline, avoiding premature completions and over-allocation of resources.

A. *Monitoring*

In NN training, data are typically processed in cycles called *epochs*. An epoch refers to a complete pass through the entire training dataset. Training an NN usually involves multiple epochs in which the model is iteratively learning from the data. To manage memory and computational resources efficiently, the dataset is also divided into smaller subsets known as *batches*. Each batch contains a fixed number of samples from the dataset. Instead of processing the entire dataset at once, the training process feeds the NN with these batches and updates its parameters incrementally after each batch [19].

DECOR-NN defines the progress at time k as the number of processed batches at k with respect to the total amount of batches needed to complete the training job:

$$a_{\%}(k) = 100 \cdot \frac{processedBatches(k)}{totalBatches} \tag{1}$$

where $totalBatches$ is defined as:

$$totalBatches = epochs \cdot (n/batchSize) \tag{2}$$

where $epochs$ is the number of epochs, n is the total number of samples in the dataset used in the training, and $batchSize$ is the size of each batch.

The total number of batches during training is directly influenced by batch size, which is a user-defined parameter. This parameter should be carefully set to allow the controller to make fine-grained adjustments and maintain accurate control over the training process. A smaller batch size results in a higher number of batches, enabling more frequent progress updates and providing the controller with more opportunities to react to changes effectively. On the other hand, a larger batch size reduces the number of batches, causing progress updates to be less frequent and larger in percentage. This limits the controller's ability to make precise adjustments, potentially leading to inefficient training and resource allocation [20].

In contrast, increasing $epochs$ means that the dataset will be processed more times, potentially leading to better model performance, as the network has more opportunities to learn from the data. However, this also increases the number of iterations, and thus the training time.

B. Controller

The controller is the component responsible for determining the resources to allocate at each time step k to meet established deadlines. *DECOR-NN* allows users to set how conservative they want to be with their training deadlines. The $desiredResponseTime$, that is, the target training time for the controller, is calculated by multiplying the $deadline$ (the maximum allowed training time) by a parameter $\alpha \in (0, 1]$. A higher α means that $desiredResponseTime$ will be very close to $deadline$, while a lower α allows for some buffer time to complete the training process in advance to avoid missing the deadline due to unexpected delays. For example, setting $\alpha = 0.95$ means that the desired response time is 95% of the actual deadline. This 5%-buffer provides a safety margin to accommodate any unforeseen delays or fluctuations during training and reduces the risk of missing the deadline.

In control terms, the setpoint is the target value that a system aims to maintain through its control mechanisms. The setpoint $a^{sp}_{\%}$ of *DECOR-NN* is a ramp that represents an ideal linear progress from time 0 (progress 0%) to time $desiredResponseTime$ (progress 100%), that is:

$$a^{sp}_{\%}(k) = 100 \cdot \frac{elapsedTime(k)}{desiredResponseTime} \tag{3}$$

For example, this means that when 40% of the available time has passed, the expected progress in terms of executed batches over the total should also be 40%. This ensures that the training process is on track to meet the deadline and maintains an ideally constant pace throughout the duration.

The controller is based on control theory and is activated every T_s seconds, a user-defined parameter that represents the duration of each discrete control step k. It exploits a Proportional-Integral (PI) model [21] defined as follows:

$$\begin{cases} e(k) &= a^{sp}_{\%}(k) - a_{\%}(k) \\ i(k) &= i(k-1) + \eta \cdot e(k-i) \\ p(k) &= K \cdot e(k) \\ d(k) &= K \cdot i(k) + p(k) \end{cases} \tag{4}$$

At each control step k, the controller model computes the error $e(k)$, which is the difference between the desired progress $a^{sp}_{\%}(k)$ and the actual progress $a_{\%}(k)$ (see Equation 1). The integral term $i(k)$ accumulates the past errors, weighted by η, reflecting the cumulative deviation from the setpoint over time. The tuning parameter $\eta \in (0, 1)$ adjusts the controller behavior. Lower values of η make the system more stable, meaning that it is less likely to experience fluctuations or become unstable, thereby preventing the progress of the training job from oscillating around the set point. However, this also means that it will take longer for the system to recover from any disturbances or deviations from the desired progress.

The proportional term $p(k)$ is directly proportional to the current error, scaled by the tuning parameter K, that is, the gain of the controller. The gain determines how strongly the controller responds to the error between the desired and actual progress. A higher K means that the controller will make more significant adjustments to the number of devices allocated in response to any given error, leading to faster corrections. Conversely, a low K results in smaller adjustments, making the controller response more gradual.

The controller then combines these terms to determine the control signal $d(k)$, which adjusts the number of devices allocated (either CPU cores or GPUs). The proportional term $p(k)$ responds to the current error, while the integral term $i(k)$ accounts for the accumulated past errors.

The controller includes an anti-windup mechanism, which ensures that the number of allocated devices remains within a user-defined minimum and the maximum number of available devices. In addition, a quantization mechanism is implemented to allocate devices in discrete steps. Thus, the number of devices d' to actuate is defined as follows:

$$d'(k) = max(d_{min}, min(d_{max}, q \cdot \left\lceil \frac{d(k)}{q} \right\rceil)) \tag{5}$$

where $d(k)$ is the initial control signal, d_{min} and d_{max} are the minimum and maximum number of devices (e.g., the total amount of CPU cores or GPU available in the host machine), and q is the quantization parameter. Given that different device types provide different allocation granularity —fractions for CPUs and whole units for GPUs— we set the value of q to 0.05 for CPUs and 1 for GPUs. Based on our experiments (see Section V), a value of 0.05 for CPUs is low enough to allow for fine-grained control over resource allocation and provides precise adjustments. In contrast, 1 for GPUs reflects the fact that they can only be allocated in units.

To ensure that the controller function correctly, T_s should be set to a value larger than the actuation time needed to re-allocate devices. This way, we ensure that each re-allocation be completed before the next allocation occurs to prevent overlapping re-allocations.

Note that the presented controller does not natively support executions with both CPUs and GPUs simultaneously for the same training job. However, a system can be engineered to provide such support by using two separate instances of the controller, one for GPUs and one for CPUs, both sharing the same deadline and setpoint. This system should employ a load balancer to direct a smaller portion of batches to CPUs and a larger portion to faster GPUs, optimizing the use of available resources.

IV. Prototype

We developed a prototype[3] on top of PyTorch, a widely used framework that offers an easy-to-use Python interface for training NN [22].

DECOR-NN extends PyTorch by allowing users to set a deadline when submitting a training job and by implementing dynamic resource allocation (not supported by PyTorch).

Although CPU and GPU executions share the same control model presented in Section III, their implementation differs due to the distinct characteristics of these devices and may require the use of specific PyTorch components.

A. DECOR-NN for CPUs

When a user schedules a training job for execution on a CPU, *DECOR-NN* wraps the PyTorch application (i.e., the user code and the PyTorch runtime) in a Docker[4] container. The use of containers allows one to isolate the training process from other running applications but also to control its CPU allocation at runtime (i.e., vertical scalability). This dynamic allocation is achieved by adjusting two parameters of the Linux Completely Fair Scheduler (CFS)[5]: CPU_{quota} and CPU_{period}. Within each CPU_{period} (defaulting to $100ms$), a group of tasks can use up to CPU_{quota} CPU time before being throttled. To allocate n cores, CPU_{quota} must be set as follows:

$$CPU_{quota} = n \cdot CPU_{period} \tag{6}$$

Note that n can be a fractional number, allowing for finer allocation granularity when controlling the progress of the training phase. For example, if n is set to 1.5 and CPU_{period} is 100 ms, CPU_{quota} would be 150 ms, meaning the process can use 1.5 CPU cores. This finer granularity enables more precise adjustments to resource allocation and enhances control over the training process.

The monitoring system is wrapped within the container, while the controller and actuator run outside the container in a separate process. This setup allows the actuator to change the resources allocated to the PyTorch application container, as Docker commands cannot be executed from within the container itself. Variable n in Equation 6 is set to be equal to the controller output $d'(k)$ (see Equation 5) where the number of devices is interpreted as the number of cores and CPU_{period} remains at its default value (100 ms). As mentioned above, the cores are quantized with a precision of 0.05 cores, which allows for a precise resource allocation.

The communication between the controller and the other components in the CPU approach is unidirectional: The monitoring system updates the progress as soon as a batch of data is processed and notifies the controller by sharing a volume with the host machine running the containers. At each control step k, the actuator enforces the new allocation using the following Docker command: `docker update --cpu-quota=((d'(k) * 100000)) pytorch_app_name`

This command dynamically adjusts the CPU resources allocated to the training container, and the actuation delay is almost immediate, typically within hundreds of milliseconds. Given the size of the batches used in the experiments and the fast actuation time of Docker, the control period T_s has been set to 1 second.

B. DECOR-NN for GPUs

Although the control mechanism remains unchanged, working with GPUs presents more challenges. The main challenge lies in enabling efficient use of fractional GPUs, so that *DECOR-NN* could work in a way similar to that presented for CPUs. GPUs are not inherently designed for dynamic reallocation or fine-grained partitioning. Several approaches attempt to address this challenge, but each comes with limitations that prevents their usage in *DECOR-NN*.

MIG [23] provides strict isolation by dividing a GPU into fixed (non-fractional) independent partitions. This ensures predictable performance, but lacks flexibility, as partitions cannot be resized dynamically at run-time. Moreover, MIG is only available on NVIDIA Ampere GPUs and newer architectures. MPS [24], on the other hand, allows for dynamic sharing of GPU resources across multiple CUDA processes. Although it provides more flexibility than MIG, MPS does not guarantee strict isolation, leading to potential interference between processes and performance variability which could affect *DECOR-NN*'s control system.

GPU Multi-Stream [25] improves intra-process parallelism by enabling concurrent execution of execution graphs within a single process. While this can enhance resource utilization, such an approach requires precise prior knowledge of GPU occupancy, which is not possible in dynamic execution environments.

To avoid performance issues due to inefficient partitioning and to provide a highly compatible solution, *DECOR-NN* instead works with multiple independent GPUs in parallel. This means that, *DECOR-NN* allows to allocate single physical or virtual GPUs and not fractions of them. Our solution exploits data parallelism, that is, unlike having a sequential execution of batches, multiple batches of the same training job are allocated in parallel to different GPUs [26].

Integration with PyTorch multi-GPU paradigms. PyTorch provides various tools to use multiple GPUs in parallel. *DataParallel* uses multi-threading to split batches of data

[3]The code and all the experiments are available at https://zenodo.org/records/12818562

[4]https://www.docker.com

[5]https://docs.kernel.org/scheduler/sched-design-CFS.html

and send them to different GPUs. Although it may be not optimal in terms of efficiency due to the Global Interpreter Lock (GIL) in Python, which only allows one thread to execute at a time, *DataParallel* provides a straightforward method to parallelize data management among GPUs on a single machine. Unlike *DataParallel*, *DistributedDataParallel* creates a separate process for each GPU and avoids the performance drop caused by the GIL and is the paradigm recommended by the PyTorch documentation[6]. It also supports multi-node distributed training of a model. *DECOR-NN* supports both *DataParallel* and *DistributedDataParallel*. This design decision is grounded in the needs and mechanics of dynamic resource allocation. *DataParallel* and *DistributedDataParallel* paradigms operate under the data parallelism model [27], where the training data is divided into smaller batches that are processed in parallel among multiple GPUs. This model aligns well with the dynamic resource allocation of *DECOR-NN*, which adjusts the allocation of resources based on training progress and computational demands.

Dynamic GPU allocation. For training jobs scheduled for GPU execution, *DECOR-NN* does not utilize containers. To support multiple GPUs, PyTorch requires users to add an additional component to the NN (i.e., a layer) for the configuration of batching and GPU parallelism. *DECOR-NN* incorporates a custom actuator (a Python module) that manages resource allocation by integrating with the supported PyTorch parallelism paradigms. Such component requires to specify the amount of devices to be allocated for training the NN and any change to this necessitates saving, modifying, and reloading the model. *DECOR-NN*'s actuator operates on top of this component so that it synchronizes threads or processes each time a different GPU allocation is enacted to ensure that model updates are properly handled.

Since the actuation of GPU allocations is managed within the application context (and not externally as for containerized CPU executions), the communication between controller and the PyTorch application (i.e., the user-defined training code), along with the monitoring system and the actuator, is bidirectional, as the application itself needs to be aware of re-allocation requests to synchronize the training cycle.

Moreover, because the controller must read the current progress of the training phase and compute the number of GPU devices every T_s, it cannot stop and wait for synchronization. The time to reallocate GPUs is not constant and depends on different factors such as GPUs waiting for other GPUs to finish processing their batch. Therefore, the system may still be in the synchronization phase when T_s has already passed, causing the controller to decide on another GPU allocation before the previous one is enacted.

To address this issue, *DECOR-NN* tracks both the last successful allocation and the last attempted allocation. When the controller needs to update the number of devices, it checks if the last attempted allocation was successful. If so, the controller would work as expected. If the allocation is still pending, the controller produces the previous, not yet actuated allocation, and the integral contribution $i(k)$ is not altered (see Equation 4), that is, $i(k) = i(k-i)$.

V. Experiments

The evaluation aims to answer these research questions.

- **RQ1:** Can *DECOR-NN* meet deadlines efficiently when using CPUs?
- **RQ2:** Can *DECOR-NN* meet deadlines efficiently when using GPUs?
- **RQ3:** How does *DECOR-NN* behave when deadlines are changed at run-time?

To evaluate *DECOR-NN*, we conducted a series of tests using different models and multiple deadlines to thoroughly assess *DECOR-NN*. We selected three benchmark NNs, namely ResNet-50 [28], *VGG-19* [29], and *Inception v3* [30], due to their widespread use and different model architectures. To run each set of experiments, the model was initially trained using all available devices (either CPUs or GPUs according to the experiment) to determine the minimum training time. This value was then multiplied by a deadline coefficient, d_c, to generate the deadlines for testing *DECOR-NN*. For example, with $d_c = 1.5$, the experiment's deadline is 50% longer than the minimum training time.

We executed the experiments three times for each model and for each deadline. The chosen values for d_c were 1.0, 1.5, and 1.8. 1.0 means testing *DECOR-NN* under tight deadlines that are only feasible with nearly all available devices , while the higher values allowed for more relaxed deadlines. In all the experiments, we set $\alpha = 1$ (see Section III-B) to let *DECOR-NN* target exactly the input deadline.

During the experiment execution, we measured the precision of *DECOR-NN* as the error percentage $\epsilon^{\%}$ with respect to the deadline, that is:

$$\epsilon^{\%} = 100 \cdot (trainingTime - deadline)/deadline \quad (7)$$

This means that the negative values of $\epsilon^{\%}$ reflect runs that finished before the target time; positive values are violations of the deadline.

In particular, we measured $\epsilon^{\%}_{|\mu|}$, that is, the average of the absolute values of $\epsilon^{\%}$ across the different repetitions, $\epsilon^{\%}_{MAX}$, that is, the maximum error (i.e., the run with the greatest violation), and $\epsilon^{\%}_{MIN}$, that is, the minimum error (i.e., the run with the highest anticipation). Note that minimum or maximum errors equal 0% indicate, respectively, that the training never finished early or late. Moreover, we also collected μ_d, that is, the average number of devices allocated (either CPUs or GPUs according to the experiment).

A. CPU-based training (RQ1)

To test *DECOR-NN* on CPUs we used an AWS m4.2xlarge instance, which is equipped with 8 virtual 2.4 GHz Intel Xeon Broadwell E5-2686 v4 CPUs. The control period was set to 1 second ($T_s = 1$). The results of this set of experiments are summarized in Table I.

[6]https://pytorch.org/docs/stable/generated/torch.nn.DataParallel.html

TABLE I
CPU-BASED RESULTS.

MODEL	$\mathbf{d_c}$	$\mu_\mathbf{d}$	$\epsilon_{\lvert\mu\rvert\%}$	$\epsilon^{\%}_{\mathbf{MIN}}$	$\epsilon^{\%}_{\mathbf{MAX}}$
Inception v3	1.0	7.83	0.79%	−0.72%	1.2%
Inception v3	1.5	2.88	0.43%	−0.43%	0%
Inception v3	1.8	2.32	0.47%	−0.54%	0%
ResNet50	1.0	7.93	3.78%	0%	5.67%
ResNet50	1.5	2.60	0.43%	−0.48%	0%
ResNet50	1.8	2.42	0.34%	−0.4%	0%
VGG-19	1.0	7.87	0.93%	0%	1.1%
VGG-19	1.5	2.76	0.49%	−0.49%	0%
VGG-19	1.8	2.27	0.47%	−0.51%	0%

TABLE II
GPU-BASED RESULTS.

MODEL	$\mathbf{d_c}$	$\mu_\mathbf{d}$	$\epsilon_{\lvert\mu\rvert\%}$	$\epsilon^{\%}_{\mathbf{MIN}}$	$\epsilon^{\%}_{\mathbf{MAX}}$
Inception v3	1	4.00	2.48%	0%	2.59%
Inception v3	1.5	2.89	0.42%	−0.29%	0.55%
Inception v3	1.8	2.29	0.93%	0%	1.01%
ResNet50	1	4.00	5.04%	0%	5.99%
ResNet50	1.5	2.80	1.25%	0%	1.99%
ResNet50	1.8	2.30	0.83%	0%	0.85%
VGG-19	1	4.00	2.53%	0%	2.62%
VGG-19	1.5	2.94	0.30%	0%	0.35%
VGG-19	1.8	2.43	0.27%	0%	0.39%

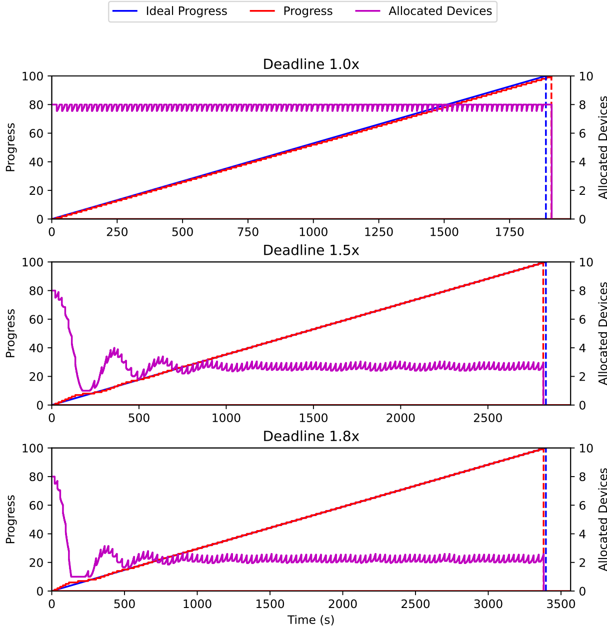

Fig. 2. *VGG-19* Progress on CPU.

For $d_c = 1.0$, *DECOR-NN* allocated a high number of CPU cores close to the maximum available to meet tight deadlines. Inception v3, ResNet50, and VGG-19 required an average of 7.83, 7.93, and 7.87 cores, respectively. The absolute average percentage errors of 0.79% and 0.93% for Inception v3 and VGG-19 indicate minor deadline violations. However, we observed a larger error (3.78%) for ResNet50. These errors can be attributed to the controller that did not reach a steady state immediately but transitioned across different states where not all available cores were fully utilized. Since $d_c = 1.0$ represents the tightest feasible deadline, these violations occur due to necessary adjustments and transitions during the initial phase. Moreover, ResNet50 tends to be more sensitive to batch size adjustments, leading to less efficient resource utilization under strict time constraints [31].

After relaxing the deadline ($d_c = 1.5$), the average number of used CPUs decreased significantly, demonstrating *DECOR-NN*'s ability to scale down resources while keeping deadlines. Inception v3, ResNet50, and VGG-19 showed average allocations of 2.88, 2.60, and 2.76 devices, with low absolute average percentage errors of $0.43\%, 0.43\%$, and 0.49%. Violations ($\epsilon^{\%}_{MAX}$) are zero for all models showing *DECOR-NN*'s precision and efficiency in resource allocation under less strict deadlines.

Similarly, for $d_c = 1.8$, *DECOR-NN* further reduced device allocation, with Inception v3, ResNet50, and VGG-19 requiring 2.32, 2.42, and 2.27 devices on average. The absolute average percentage errors remained low at 0.47%, 0.34%, and 0.47%, respectively. These minimal errors confirm *DECOR-NN*'s ability to optimize resources effectively while adhering to the desired progress rate. In general, *DECOR-NN* shows robust performance across different models and deadlines. It efficiently allocates resources to meet tight deadlines with minimal error and scales down effectively for relaxed deadlines, maintaining high accuracy and reliability. The consistent results across various models highlight *DECOR-NN*'s robustness and capability to handle dynamic training environments.

To better visualize the precision of *DECOR-NN*, Figure 2 shows the evolution of progress and the core allocation of *VGG-19*. The three charts, from top to bottom, represent runs with d_c equal to 1, 1.5, and 1.8. These charts demonstrate how *DECOR-NN* gracefully reaches a steady state in terms of core allocation after an initially more fluctuating phase. In all three charts, the ideal linear progress rate ($a^{sp}_{\%}(k)$) and the actual progress rate ($a_{\%}(k)$) almost overlap, showcasing *DECOR-NN*'s ability to minimize errors. This behavior is positively influenced by the fine-grained actuation that allows for fractional CPU core allocations.

B. GPU-based training (RQ2)

To test *DECOR-NN* on GPUs we used an AWS instance of a g4dn.12xlarge, which is equipped with 4 NVIDIA T4 Tensor Core GPUs, and its results are listed in Table II. For these experiments, we used DataParallel as PyTorch paradigm, and a sample time of $T_s = 5s$, to avoid too frequent actuations that may introduce delays.

For $d_c = 1.0$, *DECOR-NN* consistently allocated the maximum available GPUs to meet tight deadlines (constant

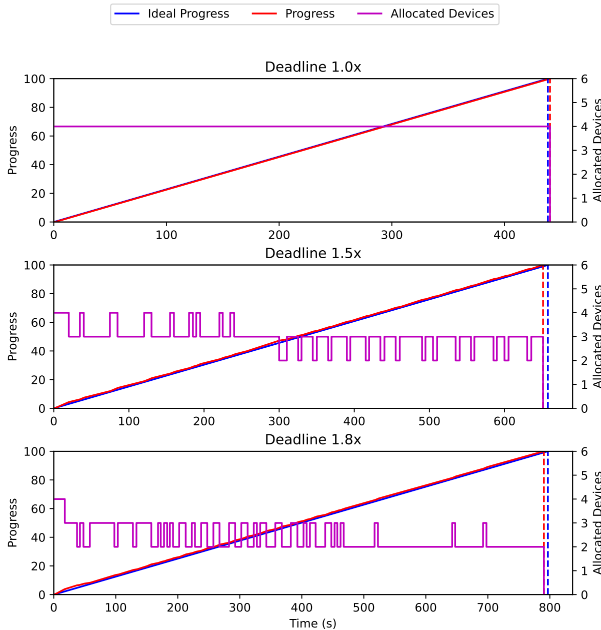

Fig. 3. *VGG-19* Progress on GPU.

allocation of 4 devices, as expected, in all applications). The absolute average percentage errors were 2.48%, 5.04%, and 2.53%, respectively, confirming the behavior observed in the CPU-based training.

When the deadline coefficient was relaxed to $d_c = 1.5$, the average number of allocated GPUs decreased, demonstrating *DECOR-NN*'s ability to scale down resource usage while maintaining deadline adherence. Inception v3, ResNet50, and VGG-19 had average allocations of 2.89, 2.80, and 2.94 GPUs, respectively. The absolute average percentage errors were, respectively, 0.42%, 1.25%, 0.30%, reflecting the balance between reduced resource usage and maintaining performance. There were minor deadline violations ($\epsilon^{\%}_{MAX}$) with delays equal to 2.59%, 1.99%, 0.35%, respectively.

Similarly, for $d_c = 1.8$, *DECOR-NN* slightly reduced GPU allocation, with Inception v3, ResNet50, and VGG-19 requiring 2.29, 2.30, and 2.43 GPUs on average. The absolute average percentage errors were 0.93%, 0.83%, and 0.27%, respectively.

To better visualize the results of *DECOR-NN*, we present in Figure 3 the evolution of progress and GPU allocation for the application *VGG-19*. The three charts, from top to bottom, represent runs with d_c values of 1, 1.5, and 1.8. Similarly to the CPU experiments, the ideal linear progress rate ($a^{sp}_{\%}(k)$) and the actual progress rate ($a_{\%}(k)$) almost overlap, demonstrating the precision of *DECOR-NN* in following the desired progress rate. However, due to the coarser granularity of GPU allocation, the charts exhibit more fluctuations in the number of GPUs allocated. This is likely because the optimal resource allocation often falls between two integer values. Despite these fluctuations, *DECOR-NN* efficiently meets the target deadlines with a very small margin of error, showcasing its effectiveness in managing GPU resources.

TABLE III
DECOR-NN RESULTS ON CPU, DEADLINE CHANGED AT 30% PROGRESS.

MODEL	$\mathbf{d_c}$	$\mu_\mathbf{d}$	$\epsilon_{\lvert\mu\rvert\%}$	$\epsilon^{\%}_{\mathbf{MIN}}$	$\epsilon^{\%}_{\mathbf{MAX}}$
Inception v3	$1.5 \rightarrow 1.2$	3.92	2.10%	0%	2.28%
Inception v3	$1.8 \rightarrow 1.44$	3.85	1.16%	0%	1.16%
ResNet50	$1.5 \rightarrow 1.2$	3.95	1.47%	0%	2.28%
ResNet50	$1.8 \rightarrow 1.44$	3.89	1.01%	0%	1.47%
VGG-19	$1.5 \rightarrow 1.2$	3.91	2.30%	0%	2.47%
VGG-19	$1.8 \rightarrow 1.44$	3.84	1.38%	0%	1.53%

TABLE IV
DECOR-NN RESULTS ON GPU, DEADLINE CHANGED AT 30% PROGRESS.

MODEL	$\mathbf{d_c}$	$\mu_\mathbf{d}$	$\epsilon_{\lvert\mu\rvert\%}$	$\epsilon^{\%}_{\mathbf{MIN}}$	$\epsilon^{\%}_{\mathbf{MAX}}$
Inception v3	$1.5 \rightarrow 1.2$	3.35	1.40%	-1.60%	0%
Inception v3	$1.8 \rightarrow 1.44$	2.73	1.19%	-1.26%	0%
ResNet50	$1.5 \rightarrow 1.2$	3.40	2.13%	-2.24%	0%
ResNet50	$1.8 \rightarrow 1.44$	2.78	1.65%	-1.72%	0%
VGG-19	$1.5 \rightarrow 1.2$	3.42	0.94%	-0.97%	0%
VGG-19	$1.8 \rightarrow 1.44$	2.80	0.80%	-0.89%	0%

C. Variable Deadlines (RQ3)

We also tested *DECOR-NN* under conditions in which the deadline changes dynamically during the training phase. Specifically, we adjusted the deadline to 80% of its original value once the ideal progress reached 30%. The chosen d_c values were the same as those used in the static deadline tests, excluding $d_c = 1.0$, since lowering the deadline beyond this point would render it unfeasible. The updated d_c values after the change are reflected in the d_c columns in Table III and IV.

For the CPU approach, *DECOR-NN* effectively handled dynamic deadlines without any violations. The performance showed a slight increase in the absolute average percentage error compared to the static case, which is expected as the controller had to quickly adapt to the new, stricter deadline. Despite the abrupt change, the maximum error observed was within a reasonable margin, demonstrating *DECOR-NN*'s robustness in managing dynamic deadlines.

In the GPU approach, no deadlines were violated, following a similar pattern to the static deadline case in terms of earliness. The errors related to earliness were slightly larger than in the static case, which is reasonable since the controller needed to recover abruptly and had less time to adapt to the new deadline. The maximum error while arriving early was 2.24%.

Figures 4 and 5 illustrate the behavior of *DECOR-NN* using a sample of *VGG-19*, trained with $d_c = 1.8$, which was adjusted at run-time to $d_c = 1.44$. These figures show that once the perturbation is detected, there is a spike in the number of allocated devices, followed by a convergence towards a new stable number of devices if the remaining time permits. This behavior highlights *DECOR-NN*'s ability to dynamically

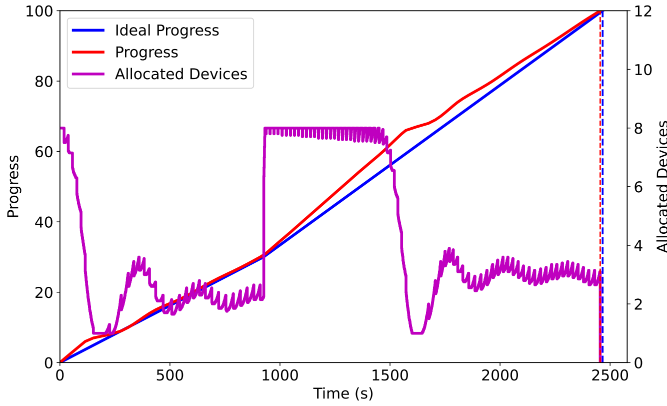

Fig. 4. *VGG-19* Progress on CPU, perturbed.

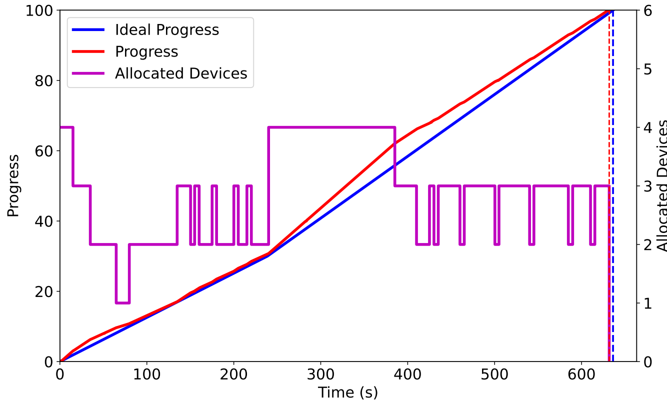

Fig. 5. *VGG-19* Progress on GPU, perturbed.

reallocate resources in response to changing deadlines while minimizing deviations from the target.

Overall, the results demonstrate that *DECOR-NN* is capable of effectively handling dynamic deadlines with minimal errors and no deadline violations. The system's ability to quickly adapt to new deadlines while maintaining performance underscores its robustness and flexibility in dynamic training environments.

VI. Threats to Validity

Although our evaluation of *DECOR-NN* demonstrates encouraging results, it is essential to acknowledge potential limitations and discuss how we addressed them [32].

Internal Validity. A potential threat arises from the limited set of models and deadline coefficients tested. We focused on three well-known architectures (ResNet-50, VGG-19, and Inception v3), each with distinct complexities and resource demands, to ensure a representative set of scenarios. Although this choice may not reflect all possible architectures or workload conditions, these models are widely adopted benchmarks. Similarly, we selected a set range of deadline coefficients. While not exhaustive, this range allows to capture both tight and relaxed scenarios demonstrating how *DECOR-NN* is able to effectively control the speed of training without wasting resources.

Early stopping, i.e., stopping training once predefined conditions are met, can impact our ability to accurately assess the progress of a training job. If a condition is met, the job may finish earlier than expected. Although this would not result in any deadline violations, it could lead to suboptimal resource utilization (i.e., allocation of more resources than necessary). In the future, we plan to mitigate this by exploring the use of symbolic execution and predictive models to understand the likelihood of meeting one of these conditions [33].

External Validity. One aspect of our evaluation that can influence external validity is the relatively short duration of the training processes that we tested, on the order of minutes. Under such conditions, the initial period, during which *DECOR-NN* is still adjusting to the current training dynamics, can represent a significant portion of the overall execution time. This initial phase acts as a transition period, where the system moves from an initial allocation of resources to a stable configuration that closely tracks the desired progress rate. In longer training sessions (e.g., hours or days), the same transition phase would occupy a smaller fraction of the total execution time. As a result, any temporary imprecision or overhead incurred during this initial adjustment would weigh less heavily on the overall outcome. Thus, we expect that for longer training jobs, the errors reported for $d_c = 1$, which is the tightest deadline setting, would actually be lower, as the vast majority of the training period would occur under stable, steady-state conditions.

Construct Validity. We assessed *DECOR-NN* using error percentages as primary metrics to evaluate its timeliness and efficiency. Although these metrics focus on the core objective of achieving predictable training completion times, they do not encompass all potential operational considerations (e.g., cost, energy consumption, or model accuracy). However, our central aim is to provide software engineers with a tool that simplifies the management and control of training durations. By demonstrating minimal deviations from the set dealines, we effectively show that *DECOR-NN* can achieve deterministic training times, making it easier for engineers to plan and adjust their development cycles. This initial focus on timing serves as a clear and quantifiable baseline from which more comprehensive evaluations can be performed, incorporating additional metrics and perspectives, in future work.

Conclusion Validity. Our conclusions rely on multiple runs and averaged results to minimize the influence of transient fluctuations in cloud environments. Although factors like resource contention and network variability remain outside our direct control, our methodology was selected, using repeated runs and standard workloads, to reduce random noise and highlight stable trends. The reported minimum and maximum values are always close to the average, demonstrating stable performance and low variability across multiple experimental runs. This consistency suggests that *DECOR-NN* can reliably meet its time objectives, minimizing uncertainty, and making training durations more predictable.

VII. Related Work

Different works tackle the problem of NN training from various perspectives, often emphasizing specific aspects such as scheduling or infrastructure optimization [10]. However, dynamic resource allocation is rarely the primary focus in these approaches.

In particular, Gu et al. [14] propose ElasticFlow, a multi-tenant oriented platform for NN to handle deadline constraints. ElasticFlow focuses primarily on adapting the scheduling of training jobs to meet predefined deadlines. They employ an optimization problem to ensure that the jobs are completed within the set time frames. However, they assume a static resource allocation model: Once GPUs are assigned to a training job, they remain fixed throughout the process. This static allocation does not allow for on-the-fly adjustments if conditions change, such as altering priorities, modifying deadlines, or reallocating resources as other training jobs complete or start. In contrast, our approach supports dynamic resource allocation. This means that both CPU and GPU resources can be adjusted in real-time, enabling the system to respond to evolving requirements. Moreover, deadlines can be changed at run-time to let higher-priority jobs reclaim resources. Since these solutions tackle different but complementary aspects of the same problem, they can work together to form a more comprehensive solution. In the future, we plan to integrate *DECOR-NN* into ElasticFlow, combining its scheduling capabilities with our dynamic resource allocation mechanism to achieve both efficient scheduling and deadline-driven adaptive training.

Similarly, Filippini et al. [34] present ANDREAS a system for scheduling and resource allocation of NN training running on GPU-powered clusters. It approaches the problem by jointly optimizing runtime performance and energy consumption to reduce costs. As ElasticFlow, ANDREAS only provides static resource allocation and only focuses on GPU-enabled nodes, while *DECOR-NN* provides dynamic resource allocation for both CPU- and GPU-equipped machines.

Gao et al. [35] formulate an optimization problem for Edge-Cloud systems which handles both deadline requirements and job mapping, solvable through a heuristic algorithm or through integer linear programming. However, it only handles feasible deadlines, does not support deadline changes at runtime, and statically assigns a set of resources to complete the job without considering how close the deadline is.

Baresi et al. [36] propose an implementation of Apache Spark [37] that implements a control mechanism for big-data applications in a cluster, which also includes a supervisor that ensures a fair distribution of resources between concurrent jobs. This proposal, though it supports dynamic allocation of resources, contrary to *DECOR-NN* does not support PyTorch and GPU dynamic allocation. Their follow-up work [8] tackles NN-enabled applications and their dynamic resource allocation, but only focuses on the inference mode.

Dimopoulos et al. [38] propose Justice, a different deadline-sensitive system for resource negotiators such as Mesos [39] and YARN [40]. Justice statically allocates resources when a new job is scheduled, and determines the allocated amount based on data obtained for previous jobs. This approach does not support GPUs and does not support deadline changes at run-time.

Singh et al. [41] propose a method for federated learning where resource allocation cost is part of an optimization problem that balances it along with training time and accuracy, since more training rounds equal longer execution time and better accuracy. While this optimizes multiple parameters, it does not allow one to choose a deadline and thus training time is not controllable.

Ghasemi et al. [42] present a novel approach to optimize the execution of NNs on edge devices with energy constraints and deadlines. The proposed solution dynamically partitions the computation between an energy-constrained edge device and a more powerful cloudlet, aiming to minimize the edge device's energy consumption while ensuring the workload is completed within the specified deadline. The method accounts for the performance and power profiles of the devices, communication delays, and the overhead of parallel execution. A heuristic algorithm based on Lagrangian relaxation is used to solve the constrained shortest-path problem, significantly reducing decision-making overhead compared to traditional Integer Linear Programming approaches. This work focuses primarily on model parallelism, where the layers of the NN are partitioned across different devices or nodes, while *DECOR-NN* leverages data parallelism by distributing different chunks of the input dataset across multiple devices. *DECOR-NN* is particularly effective for handling large datasets, as it optimizes resource utilization through data distribution. In contrast, model parallelism is more suitable for very large NN, where the model itself is too large to fit on a single device.

VIII. Conclusions and Future Work

As applications increasingly integrate NN that require resource-intensive and time-consuming training processes, effective resource management becomes essential. In this paper, we present *DECOR-NN*, a dynamic resource allocation solution designed to train NN within specified deadlines. Our evaluation demonstrates that deadline violations are rare when submitted deadlines exceed the minimum feasible duration, with a maximum error rate of 2.51%.

Our future work includes extending our evaluation to incorporate *DistributedDataParallel* for GPUs, developing control mechanisms to manage multiple concurrent training jobs, and implementing support for both early-stopping and fractional GPU allocations. Moreover, we plan to conduct an in-depth evaluation of the impact of *DECOR-NN* on energy consumption and environmental footprint.

Acknowledgements

This work was supported by project EMELIOT, funded by MUR under the PRIN 2020 program (Contract 2020W3A5FY), and by project 3A-Italy Circular and Sustainable Made in Italy - MICS (3A-ITALY) CUP under Grants D43C22003120001 and PE00000004.

References

[1] R. Szeliski, *Computer Vision: Algorithms and Applications*. Springer Nature, 2022.

[2] D. W. Otter, J. R. Medina, and J. K. Kalita, "A Survey of the Usages of Deep Learning for Natural Language Processing," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 2, pp. 604–624, 2021.

[3] R. Livni, S. Shalev-Shwartz, and O. Shamir, "On the Computational Efficiency of Training Neural Networks," *Advances in Neural Information Processing Systems*, vol. 27, p. 855–863, 2014.

[4] W. Gao, Z. Ye, P. Sun, Y. Wen, and T. Zhang, "Chronus: a Novel Deadline-aware Scheduler for Deep Learning Training Jobs," in *Proc. of the ACM Symposium on Cloud Computing*, 2021, pp. 609–623.

[5] G. Russo Russo, V. Cardellini, and F. Lo Presti, "Hierarchical Autoscaling Policies for Data Stream Processing on Heterogeneous Resources," *Transactions on Autonomous Adaptive Systems*, vol. 18, no. 4, Oct. 2023.

[6] L. Dutta and S. Bharali, "TinyML Meets Iot: a Comprehensive Survey," *Internet of Things*, vol. 16, p. 100461, 2021.

[7] G. R. Russo, D. Ferrarelli, D. Pasquali, V. Cardellini, and F. L. Presti, "Qos-aware Offloading Policies for Serverless Functions in the Cloud-to-edge Continuum," *Future Generation Computer Systems*, vol. 156, pp. 1–15, 2024.

[8] L. Baresi, D. Y. X. Hu, G. Quattrocchi, and L. Terracciano, "NEPTUNE: a Comprehensive Framework for Managing Serverless Functions at the Edge," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 19, no. 1, pp. 1–32, 2024.

[9] J. P. K. S. Nunes, S. Nejati, M. Sabetzadeh, and E. Y. Nakagawa, "Self-adaptive, Requirements-driven Autoscaling of Microservices," in *Symposium on Software Engineering for Adaptive and Self-managing Systems (SEAMS)*, 2024, pp. 168–174.

[10] Z. Ye, W. Gao, Q. Hu, P. Sun, X. Wang, Y. Luo, T. Zhang, and Y. Wen, "Deep Learning Workload Scheduling in Gpu Datacenters: a Survey," *ACM Computing Surveys*, vol. 56, no. 6, pp. 1–38, 2024.

[11] Y. Sun, W. Shi, X. Huang, S. Zhou, and Z. Niu, "Edge Learning With Timeliness Constraints: Challenges and Solutions," *IEEE Communications Magazine*, vol. 58, no. 12, pp. 27–33, 2020.

[12] C. Prigent, A. Costan, G. Antoniu, and L. Cudennec, "Enabling Federated Learning Across the Computing Continuum: Systems, Challenges and Future Directions," *Future Generation Computer Systems*, pp. 767–783, 2024.

[13] Q. Liang, W. A. Hanafy, A. Ali-Eldin, and P. Shenoy, "Model-driven Cluster Resource Management for AI Workloads in Edge Clouds," *Transactions Autonomous Adaptive Systems*, vol. 18, no. 1, Mar. 2023.

[14] D. Gu, Y. Zhao, Y. Zhong, Y. Xiong, Z. Han, P. Cheng, F. Yang, G. Huang, X. Jin, and X. Liu, "ElasticFlow: an Elastic Serverless Training Platform for Distributed Deep Learning," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 266–280.

[15] W. Shi, S. Zhou, Z. Niu, M. Jiang, and L. Geng, "Joint Device Scheduling and Resource Allocation for Latency Constrained Wireless Federated Learning," *IEEE Transactions on Wireless Communications*, vol. 20, no. 1, pp. 453–467, 2020.

[16] H. Zheng, K. Xu, M. Zhang, H. Tan, and H. Li, "Efficient Resource Allocation in Cloud Computing Environments Using AI-driven Predictive Analytics," *Applied and Computational Engineering*, vol. 82, pp. 17–23, 2024.

[17] Y. Jia, Z. Huang, J. Yan, Y. Zhang, K. Luo, and W. Wen, "Joint Optimization of Resource Allocation and Data Selection for Fast and Cost-efficient Federated Edge Learning," *Transactions on Cognitive Communications and Networking*, pp. 1–13, 2024.

[18] S. Jain, I. Baek, S. Wang, and R. Rajkumar, "Fractional GPUs: Software-based Compute and Memory Bandwidth Reservation for Gpus," in *Real-time and Embedded Technology and Applications Symposium (RTAS)*, 2019, pp. 29–41.

[19] Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[20] Y. Bengio, "Practical Recommendations for Gradient-based Training of Deep Architectures," 2012. [Online]. Available: https://arxiv.org/abs/1206.5533

[21] O. Hanif and V. Kedia, "Evolution of Proportional Integral Derivative Controller," in *Int. Conf. on Recent Innovations in Electrical, Electronics and Communication Engineering*, 2018, pp. 2655–2659.

[22] A. Paszke *et al.*, *PyTorch: an Imperative Style, High-Performance Deep Learning Library*. Curran Associates Inc., 2019.

[23] "Multi-Instance GPU documentation," https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html, last accessed: 2024-10-13.

[24] "Multi-Process Service documentation," https://docs.nvidia.com/deploy/mps/index.html, last accessed: 2024-10-13.

[25] N. Corporation, *CUDA Multi-Stream Best Practices*, 2023, accessed: 2024-11-22. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#multiple-streams

[26] Hugging Face, *Model Parallelism*, https://huggingface.co/docs/transformers/v4.17.0/en/parallelism, 2022.

[27] Y. E. Wang, C.-J. Wu, X. Wang, K. Hazelwood, and D. Brooks, "Exploiting Parallelism Opportunities With Deep Learning Frameworks," *ACM Transactions on Architecture and Code Optimization*, vol. 18, no. 1, pp. 1–23, 2020.

[28] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," 2015. [Online]. Available: https://arxiv.org/abs/1512.03385

[29] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-scale Image Recognition," 2015. [Online]. Available: https://arxiv.org/abs/1409.1556

[30] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision," 2015. [Online]. Available: https://arxiv.org/abs/1512.00567

[31] S. L. Smith, P. Kindermans, and Q. V. Le, "Don't decay the learning rate, increase the batch size," *CoRR*, vol. abs/1711.00489, 2017. [Online]. Available: http://arxiv.org/abs/1711.00489

[32] C. Wohlin, M. Höst, and K. Henningsson, "Empirical Research Methods in Web and Software Engineering," in *Web Engineering*, 2006, pp. 409–430.

[33] L. Baresi, G. Denaro, and G. Quattrocchi, "Symbolic Execution-driven Extraction of the Parallel Execution Plans of Spark Applications," in *Proc. of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 246–256.

[34] F. Filippini, D. Ardagna, M. Lattuada, E. Amaldi, M. Riedl, K. Materka, P. Skrzypek, M. Ciavotta, F. Magugliani, and M. Cicala, "Andreas: Artificial Intelligence Training Scheduler for Accelerated Resource Clusters," in *International Conference on Future Internet of Things and Cloud*. IEEE, 2021, pp. 388–393.

[35] C. Gao, A. Shaan, and A. Easwaran, "Deadline-constrained Multi-resource Task Mapping and Allocation for Edge-Cloud Systems," in *Global Communications Conference*. IEEE, 2022, pp. 5037–5043.

[36] L. Baresi, A. Leva, and G. Quattrocchi, "Fine-grained Dynamic Resource Allocation for Big-data Applications," *IEEE Transactions on Software Engineering*, vol. 47, no. 8, pp. 1668–1682, 2021.

[37] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proc. of the 2nd Conf. on Hot Topics in Cloud Computing*. USENIX, 2010, p. 10.

[38] S. Dimopoulos, C. Krintz, and R. Wolski, "Justice: a Deadline-aware, Fair-share Resource Allocator for Implementing Multi-analytics," in *Int. Conf. on Cluster Computing* . IEEE, 2017, pp. 233–244.

[39] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center," in *Symposium on Networked Systems Design and Implementation*. USENIX, 2011.

[40] V. K. Vavilapalli *et al.*, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *Proc. of the Annual Symposium on Cloud Computing*. ACM, 2013.

[41] A. K. Singh and K. Khoa Nguyen, "Joint Selection of Local Trainers and Resource Allocation for Federated Learning in Open Ran Intelligent Controllers," in *IEEE Wireless Communications and Networking Conf.*, 2022, pp. 1874–1879.

[42] M. Ghasemi, S. Heidari, Y. G. Kim, C.-J. Wu, and S. Vrudhula, "Energy-efficient, Delay-constrained Edge Computing of a Network of Dnns," *IEEE Transactions on Computers*, pp. 1–13, 2024.