Extra-Deep: Automated Empirical Performance Modeling for Distributed Deep Learning

Marcus Ritter
Department of Computer Science
Technical University of Darmstadt
Darmstadt, Germany
marcus.ritter@tu-darmstadt.de

Felix Wolf
Department of Computer Science
Technical University of Darmstadt
Darmstadt, Germany
felix.wolf@tu-darmstadt.de

ABSTRACT

With the rapidly increasing size and complexity of DNNs, equally sophisticated methods are needed to train them efficiently, including distributed training and various model/hybrid parallelism approaches. Even though developers heavily rely on state-of-the-art frameworks such as PyTorch and TensorFlow, these provide little insight into an application's training behavior at scale, leading to latent performance bottlenecks and inefficient training configurations. We propose Extra-Deep, an automated empirical performance modeling approach for distributed deep learning to model performance metrics, such as the training time, as a function of the applications' configuration parameters. We leverage the created models to analyze a training task's performance, scalability, efficiency, and cost. Gathering empirical measurements of full training runs is very laborious and costly. Therefore, we employ an efficient sampling strategy that reduces the profiling time for the required empirical measurements by, on average, about 94.9%. Using our sampling strategy, we can analyze the performance behavior and identify cost-effective training configurations even for large-scale and long-running applications. We evaluated our approach on three parallelization strategies, with four DNN models and five datasets. The results show that Extra-Deep has an average prediction accuracy of 93.6% when compared to empirical results.

CCS CONCEPTS

• **Computing methodologies** → **Modeling methodologies**; *Machine learning*; Parallel computing methodologies; Distributed computing methodologies.

KEYWORDS

Deep learning, data parallelism, model parallelism, performance modeling, performance analysis, high performance computing

ACM Reference Format:
Marcus Ritter and Felix Wolf. 2023. Extra-Deep: Automated Empirical Performance Modeling for Distributed Deep Learning. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3624062.3624204

SC-W 2023, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0785-8/23/11.
https://doi.org/10.1145/3624062.3624204

1 INTRODUCTION

Deep learning (DL) has become increasingly popular in many major research areas, often achieving new state-of-the-art results [6, 7]. Due to its ability to solve complex problems the number of possible applications is constantly increasing. At the same time, the trained models become ever larger and more complex, which leads to a continually growing computational overhead. In the field of natural language processing (NLP), for instance, model size has been increasing at least tenfold each year [6, 10, 29]. One example of this trend is OpenAIs GPT NLP model. While GPT-2 had 1.5 billion parameters [29] its successor GPT-3 features 175 billion parameters [6], requiring hundreds of GPUs and several days for training. In general, today's state-of-the-art models [6, 10] exploit the capabilities of GPUs to perform simultaneous computations in order to distribute the training processes and significantly accelerate machine learning operations. Furthermore, they far exceed the size of single GPU memory, making model parallelization and distributed training indispensable. However, huge models are only one reason for distributed training. Other causes include the acceleration of the training process utilizing data parallelism, training on large datasets, or very compute-intensive models.

1.1 Motivation and Challenges

To distribute the training process, application developers mostly use a combination of deep learning frameworks such as TensorFlow and Horovod [32]. Together they enable the distribution of the training process among nodes, utilizing all available CPUs and GPUs while also managing inter-node communication via MPI. Since individual training runs for models such as GPT-3 cost millions of dollars, developers try to make the training process as efficient as possible, optimally utilizing accelerators such as GPUs or TPUs. In general, the efficient distribution of a deep learning application is a difficult task. Therefore, many deep learning applications suffer from latent scalability bottlenecks, including inefficient input pipelines, insufficiently optimized network architectures, inappropriate model parameters, large synchronization overheads, or inefficient training configurations [14, 27, 37]. Hence, there are a variety of approaches, such as CRADLE [26] or DeepDiagnosis [36] that try to debug and localize bugs in deep learning codes.

Even though deep learning frameworks such as TensorFlow, PyTorch, and Horovod come with designated performance profiling tools [2, 3, 32], they provide little to no insight into an application's distributed training performance and efficiency at different scales. The same applies to Nsight Systems [23], a profiling and visualization tool from NVIDIA based on the CUDA Profiling Tools Interface

(CUPTI) [22]. It provides a more general performance analysis approach for all kinds of applications utilizing NVIDIA GPUs but also does not allow any assumptions on application behavior at scale or performance extrapolation.

Besides these profiling tools, there are a variety of more general approaches, focusing on studying the computation requirements, data reuse, and memory access patterns in deep learning applications [16, 18, 30]. Though, their approach is too general to extract useful performance insights for specific application codes. Again others use analytical modeling to predict the training performance of an application. Qi et al., for example, proposed PALEO, an analytical performance model that models the expected scalability and performance of a deep learning system [28]. Similarly Kahira et al. proposed ParaDL, a tool that uses a combination of analytic modeling and empirical parameterization to analyze the computational, communication, and memory requirements of convolutional neural networks (CNNs) to understand the trade-offs between different parallelism approaches on performance and scalability [17]. However, these approaches require expert knowledge and manual analysis, which significantly limits their code coverage and the amount of insights on application behavior they can provide [11, 13, 15, 24].

Moreover, there are a variety of empirical modeling approaches such as Extra-P [31]. Extra-P creates performance models based on a set of small-scale performance experiments, enabling an automated performance and scalability analysis. However, these approaches cannot analyze deep learning codes [5], or applications utilizing GPUs [8, 9]. Again other approaches often focus on creating performance models to optimize a specific application scenario, such as CNNs [12, 19, 25], or standard processes and algorithms found in almost all types of deep learning applications, such as stochastic gradient descent (SGD) [34, 35].

Consequently, application developers often lack the necessary tools, expert knowledge, or performance insights to optimize their deep learning codes and identify cost-effective training configurations. Without reliable information providing insights into an application's performance at different scales, it is generally unclear: **Q1.** How long it takes to train a specific DNN per epoch with a given resource allocation? **Q2.** How the training time per epoch and efficiency changes depending on the training configuration, e.g., the number of MPI ranks or used GPUs? **Q3.** If the application suffers from any latent performance or scalability bottlenecks? **Q4.** How much the training of a specific DNN will cost per epoch for a given training configuration? **Q5.** What the most efficient training configuration is considering a particular budget or time frame?

1.2 Contribution

In this paper, we present Extra-Deep, our novel automated empirical performance modeling framework based on Extra-P, to analyze the training performance, scalability, efficiency, and cost of distributed deep learning applications. Figure 1 shows an overview of the proposed framework and performance analysis process. Using only a few small-scale performance experiments, Extra-Deep automatically models performance metrics, such as the application runtime as a function of the applications' execution parameters, e.g., the number of MPI ranks, without requiring expert knowledge or manual analysis. Extra-Deep creates models for individual application kernels, such as CUDA kernels, and application models, e.g., describing the training time per epoch broken down by training phases (computation, communication, memory operations). Besides modeling the application runtime, our framework enables the analysis of various other performance metrics, such as the number of visits per kernel or the number of transferred bytes for memory operations. Furthermore, it supports today's most commonly used parallel strategies, including data, model, and hybrid parallelism.

Large-scale deep learning applications often require several hours to train a single epoch. Profiling an entire training run is, therefore, very expensive in terms of time and cost and, in most cases, not practical. Thus, we developed an efficient sampling strategy that effectively reduces the necessary profiling time of the required small-scale performance experiments, making the measurement of full training runs redundant. Consequently, Extra-Deep enables a performance analysis even for large-scale deep learning applications requiring several hours of training per epoch. Our main contributions are as follows:

- We propose Extra-Deep, an automatic empirical modeling framework to analyze the training performance, scalability, efficiency, and cost of distributed deep learning tasks. training configurations for specific applications.
- We propose an efficient measurement sampling strategy for distributed DL applications, which reduces the necessary profiling time by about 94.9% on average, making the measurement of full training runs redundant.
- We show that Extra-Deep accurately predicts the training performance (93.6% average prediction accuracy) by conducting a wide range of experiments for different DNN architectures, data sets, HPC systems, and parallel strategies.
- An application case study that outlines the performance insights Extra-Deep can provide while also functioning as a best practice guide for developers attempting to analyze the performance of their deep learning code.

The remainder of this paper is organized as follows. After providing an overview of our novel modeling framework, we outline Extra-Deep's performance analysis process, efficient measurement sampling strategy, and model creation in Section 2. We then provide an in-depth explanation of how we analyze application scalability, efficiency, and cost to identify cost-effective training configurations in Section 3. To outline the insights Extra-Deep can provide, we provide a case study on CIFAR-10 as a running example throughout Section 2-3, which also functions as a best practice guide for application developers. In Section 4, we evaluate the accuracy and predictive power of our modeling framework on three parallelization strategies, four DNN models, and five datasets, followed by a discussion of the results and comparison to related work. Finally, we provide a conclusion in Section 5.

2 THE EXTRA-DEEP FRAMEWORK

In this Section, we introduce Extra-Deep [1], our automated empirical performance modeling framework specifically tailored for distributed deep learning applications. To allow a better understanding of its functionality, we first outline the basic performance analysis

process. We then take an in-depth look at its efficient measurement sampling strategy, which tremendously reduces the required measurement overhead compared to measuring full training runs, and enables us to analyze the performance of long-running and large-scale applications. Subsequently, we outline how we create empirical performance models for distributed training at application and kernel level, describing the applications computational, communication, and memory footprint while supporting different types of parallel training strategies. Finally, we provide a case study on CIFAR-10 as a running example throughout the sections 2-3 to guide application developers in analyzing the performance behavior of their codes by answering the in Section 1.1 formulated questions **Q1** to **Q5**. This case study is followed by an extensive evaluation of our approach in Section 4.

For this purpose we created a simple distributed deep learning benchmark using TensorFlow and Horovod that trains a ResNet-50 on the CIFAR-10 dataset. The CIFAR-10 dataset is a subset of the 80 million tiny images dataset, containing 60 000 32x32 color images in 10 classes, with 6 000 images per class. We trained the ResNet-50 from scratch using data parallelism, weak scaling, and a batch size of 256 per rank. To scale the problem size accordingly, we multiply the size of the training dataset by the number of MPI ranks and perform some basic data augmentation. We then shard the dataset by the number of MPI ranks and shuffle it so that each worker gets a different but equal-sized piece of the dataset for training. To conduct the necessary empirical performance measurements, we used the DEEP cluster described in Table 1. Thus, each MPI rank (worker) has its own dedicated GPU. However, this also means no communication between GPUs via NCCL.

Extra-Deep is based upon Extra-P [31], an automated empirical performance modeling tool for HPC applications that enables application developers to quickly analyze the performance behavior of their program code without requiring manual analysis or expert knowledge. It is a powerful and widely used tool in the HPC community and, therefore, the ideal starting point for providing similar support for distributed deep learning applications. However, a major restriction of Extra-P is that it only enables the performance analysis of applications that use the bulk synchronous parallel (BSP) model. Today many deep learning applications also use the asynchronous parallel (ASP) model. Furthermore, it does not support strong scaling, modeling derived application metrics, (repetitive) application phases, or different parallelization strategies such as data, tensor, or pipeline parallelism. Finally, it does not support the most commonly used DL implementation frameworks, such as TensorFlow or PyTorch, nor can it leverage performance measurements from application kernels executed on a GPU. Consequently, Extra-P does not meet the requirements for the performance analysis of distributed deep learning applications.

We follow the principle of software reusability and, therefore, decided to significantly extend Extra-P's data processing, aggregation, modeling, and analysis capabilities to fit the requirements, rather than developing a new performance analysis tool from scratch. First, we introduce a new logic to read and aggregate performance measurements of kernels executed on GPUs including support for the two major DL libraries, TensorFlow and PyTorch. Second, we expand Extra-P's support for other parallel programming models to enable an analysis for data, model, and hybrid parallel training.

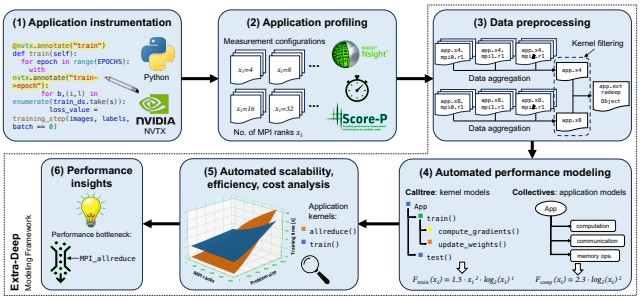


Figure 1: Overview of the Extra-Deep modeling framework including the employed performance analysis process, and the used measurement toolchain.

Third, we define novel performance functions to model metrics, such as the training time per epoch spent for computation, communication, and memory operations as a function of the applications' execution parameters. Fourth, we leverage the created models to perform further analysis, e.g., analyzing the application's scalability, efficiency, and cost to identify cost-efficient training configurations.

The general focus of our approach is on the performance analysis of distributed deep learning applications running on small to large-scale clusters. Therefore, we do not consider serial or applications that utilize only a single node, including cases of model parallel training that use several GPUs but only a single process. Finally, Extra-Deep supports weak as well as strong scaling scenarios.

2.1 Extra-Deep's Performance Analysis Process

Figure 1 outlines Extra-Deep's basic performance analysis process, including the toolchain we propose to collect the required empirical performance measurements. The numbers (**1-5**) in the blue oblong-shaped boxes describe the different analysis steps followed by their outputs displayed by the gray rectangles.

(1) Application instrumentation: To analyze an application's performance, we first instrument all high-level code, i.e., code that the developer has written. This enables us to not only analyze framework-level code such as TensorFlow or CUDA API function calls, which are automatically covered by many profiling tools such as Nsight Systems [23] or Score-P [4] but also investigate the performance of user-defined functions. We instrument an application using Extra-Deep's built-in automated instrumentation tool that uses static code analysis to instrument the code using NVIDIAs Tools Extension Library (NVTX) [21]. As almost all of todays deep learning codes are written in Python, we only support Python files.

(2) Application profiling: Subsequently, we profile the instrumented application using NVIDIA's profiling tool Nsight Systems to conduct a series of performance experiments that are the empirical measurement base for modeling. Using our efficient measurement sampling strategy (see Section 2.2), measuring five training and validation steps from two epochs of training is enough for modeling and to capture all performance-relevant kernels of the application. A training step is one gradient update of the neural network in which a predefined batch (number of samples) of the training data set is processed. A validation step evaluates the networks accuracy,

e.g., to predict a target variables value, using a batch of the validation data set without updating the networks' gradients. However, one can measure more steps, epochs, or the entire application runtime to further increase model accuracy. The first epoch acts as a warm-up round for the training process, and its measurements are not used for modeling. Many deep learning frameworks, such as TensorFlow, perform initialization and optimization operations during the first few training steps. Thus, one will encounter high variations in the measured performance metric's values. For our analysis, we measure the runtime and the number of visits for each instrumented function. This includes CUDA kernels, memset, memcopy, and NCCL operations on the GPU, as well as CUDA API, cuBLAS, cuDNN, MPI, OS, and user-defined function calls on the CPU. For the memory operations, we additionally measure the number of transferred bytes. We measure these application kernels for each MPI rank of an application configuration, where an application configuration corresponds to a specific set of execution parameters, such as the number of MPI ranks or the problem size, that are used when running the application. Alternatively, Extra-Deep supports measurements from other profiling tools such as Score-P, or any CUPTI-based performance profiler.

(3) Data preprocessing: Profiling an application with our efficient measurement sampling strategy enables us to create accurate performance models using measurements from only five training steps from two training epochs. However, this requires some preprocessing and aggregation of the collected measurement data before we can create performance models. As this step is essential for our modeling approach to reduce the required measurement overhead and enable the analysis of long-running and large-scale applications, it is described in detail in Section 2.2.

(4) Performance modeling: After data preprocessing, we employ Extra-Deep to automatically create kernel and application models for a variety of different metrics, such as the application runtime or the number of transferred bytes. A kernel model describes the performance of an individual application kernel, e.g., for a CUDA kernel executed on GPU during training. An application model, on the other hand, for example, describes the application's training time or cost per epoch. For a detailed explanation of the model creation process, see Section 2.3.

(5) Performance analysis: Next, we leverage the created models to perform further analysis, investigating the application's performance behavior at different scales and identify cost-effective training configurations, which is described in detail in Section 3.

(6) Performance insights: Finally, one can exploit the performance insights found by automated analysis to fix or improve the application code. Subsequently, one would profile the application again and run another performance analysis to verify if the made changes had the desired effect.

2.2 Efficient Measurement Sampling

Large-scale deep learning applications often require several hours to train a single epoch. Profiling an entire training run is, therefore, very expensive in terms of time and cost and, in most cases, not practical. Thus, we developed a simple heuristic sampling strategy to efficiently conduct the necessary measurements for a performance analysis while minimizing their overhead and cost.

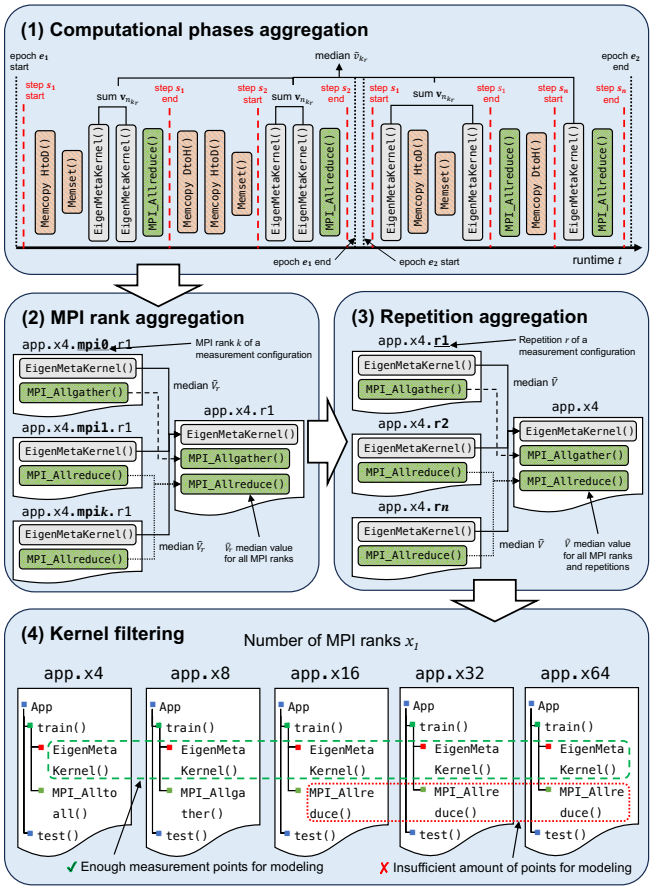


Figure 2: Overview of Extra-Deep's efficient sampling strategy (1), data preprocessing and aggregation logic (2)-(4).

To measure large-scale deep learning applications, we take advantage of the repetitive nature of their computational phases. In general, the distributed training of a DNN can be divided into the following six phases: I/O, data pre-processing, the forward pass where the input data is passed through the entire network, backpropagation to compute the gradients, the gradient exchange, and finally the update of the networks' weights. With a few exceptions, such as I/O, which can be done at once when initializing the program, if the entire training data set fits into memory, these phases are repeated for each training step of an epoch and as many epochs until model convergence is reached.

Since each training step executes almost the exact same operations, it is sufficient to profile only a small number of steps to analyze and then extrapolate the found performance behavior for an entire epoch. For large-scale applications, the training phase usually accounts for most application runtime. Therefore, one can easily measure the other phases, such as the initialization or the validation, without requiring a significant measurement overhead by simply reducing the number of epochs and training steps to one. To extrapolate the performance observations from a few training steps to an entire epoch, we inject NVTX marks into the training

step and epoch callback functions of our benchmark codes during application instrumentation. This will produce a timestamp indicating the start and end of each training step s and epoch e during profiling. This approach works for all major deep learning frameworks, such as TensorFlow or PyTorch. When reading the measurement data, Extra-Deep automatically identifies all function calls executed during a training step or between two steps.

$$\sum_{u=1}^{o} v_u = \mathbf{v}_{n_{k_r}} \tag{1}$$

Step (1) in Figure 2 illustrates how we utilize the NVTX marks to identify which function calls belong to a particular training epoch and step. To extrapolate the performance of an application kernel, such as the `EigenMetaKernel`, we first sum all metric values of all its executions during each training step s_n indicated by s_{nstart} and s_{nend} so that $\mathbf{v}_{n_{k_r}}$ represents the total metric value of the kernel for the step n, the MPI rank k, and the repetition r as shown in Equation 1. The variable v is universal and can hold the value of different metrics, such as the runtime, the number of visits per kernel, or the number of transferred bytes. The variable u represents the index of the measured values, o equals the total number of measured kernel executions, n is the training step index, and r is the index for the potential repetition of this measurement configuration. Then we calculate the median value $\tilde{v}_{k_r}$ of all steps s_n for each MPI rank k and measurement repetition r using the individual step's sum $\mathbf{v}_{n_{k_r}}$. Subsequently, we use the median values $\tilde{v}_{k_r}$ from each rank to compute the median metric value $\tilde{V}_r$ of all processes, as illustrated by step (2) in Figure 2. Next, we aggregate the measured values over the measurement repetitions as shown in step (3) of Figure 2. Therefore, we calculate the median metric value $\tilde{V}$ of all processes and repetitions by using the median metric values from each repetition $\tilde{V}_r$. Since some of the application kernels might be executed asynchronously, i.e., they could fall in between two steps, for example, s_{1end} and s_{nstart}, as shown in step (1) of Figure 2, we additionally aggregate their values by performing steps (1)-(3) just like for the other kernels that are executed during a training step.

Before creating a model, we have to ensure that this kernel is also found in at least four other application configurations (five in total) to fulfill the minimum modeling requirements of our approach [31], which are outlined in detail in Section 2.3. An application configuration or measurement point is defined by the values of the application's execution parameters, e.g., the number of MPI ranks x_1. To obtain enough data for modeling our approach requires performance measurements, e.g., of the runtime of the `EigenMetaKernel` for at least five different measurement points $x_1 = \{4, 8, 16, 32, 64, \dots\}$, as highlighted in step (4) of Figure 2. If the kernel appears in less than five of the applications' configurations, no model will be created due to an insufficient amount of data. If a kernel is only found in a single training step or MPI rank, it usually indicates that it is irrelevant to the application's performance.

To create application models, e.g., to model the training time per epoch, we use a slightly different aggregation process. First, we perform the same aggregation for each application kernel as described in steps (1)-(3). Second, we categorize the kernels by their type, i.e., computation, communication, or memory operation. Third, we sum all $\tilde{V}$ of each kernel from the same category with each other. As a result, we receive three values describing the total metric value $\tilde{V}_{comp}$ for the computation, $\tilde{V}_{comm}$ for the communication, and $\tilde{V}_{mem}$ for the memory operations and use these values for modeling. In this case, all kernel executions are essential for our model. Thus, we do not perform any kernel filtering.

For our CIFAR-10 case study we first instrument the application code by utilizing Extra-Deep's builtin instrumentation tool. We then profile five training steps from two epochs of training using the efficient measurement sampling strategy. For profiling we use Nsight Systems and measure NVTX, MPI, cuDNN, cuBLAS, and OS function calls as well as memory operations. We then use the data aggregation capabilities of our tool to convert the measurement output of Nsight Systems into one Extra-Deep object per application configuration, as shown in Figure 2.

2.3 Creating Empirical Performance Models

A performance model is simply a function that describes how the performance of a program, expressed in terms of a metric such as execution time, changes, as execution parameters, such as the number of processes, change. By conducting a series of experiments with varying execution parameters, we obtain an empirical dataset to create performance models of an application. The performance measurements in this set reflect the changes in application performance as execution parameters x_m change, where m is the number of parameters, such that Extra-Deep can discover the underlying function and automatically create a performance model. Furthermore, each of these experiments represents a specific application configuration determined by the used execution parameters, which we refer to as a measurement point $P(x_1, x_2, \dots, x_m)$.

To create performance models, Extra-Deep requires measurements of at least five different application configurations per parameter x_1 to be modeled, while the other parameters x_m are held constant [31]. Though, this is the absolute minimum requirement. As one can almost fit any function through two measurement points, we need at least five points to accurately differentiate between logarithmic, linear, and polynomial complexity. The more measurement points per parameter, the more accurately we can model the found performance behavior. The closer the measurement points are to the desired execution scale, the better the predictive power of the created models. To obtain the required measurements for modeling, we vary the application's configuration parameters, e.g., the number of MPI ranks $x_1 = \{4, 8, 16, 32, 64, \dots\}$, or the batch size $x_2 = \{32, 64, 128, 256, 512, \dots\}$. However, we do not consider hyper-parameters, parameters that are used to control the learning process, such as the learning rate or activation function. We model only parameters that directly influence the performance behavior of the training process in terms of metrics such as the runtime.

We define a measurement point $P(x_1, x_2, \dots, x_m)$ as a unique configuration of the applications execution parameters x_m, where m is the number parameters considered for modeling. The notation $\mathbf{x}_1 = \{4, 8, 16, 32, 64\}$ represents a set of possible parameter values for x_1, resulting in the measurement points $\mathbf{P}(\mathbf{x}_1) = \{P(4), P(8), P(16), P(32), P(64)\}$. A set of measurement points with several parameters is simply noted as $\mathbf{P}(\mathbf{x}_1, \mathbf{x}_2)$, where $\mathbf{x}_1 = \{4, 8\}$ and $\mathbf{x}_2 = \{32, 64\}$, will lead to the measurement points: $P(4, 32)$, $P(4, 64)$, $P(8, 32)$, and $P(8, 64)$.

For our case study, we focus on modeling the effects of the number of MPI ranks x_1 on the application's training performance.

Therefore, we measure five different application configurations that we will use for modeling $\mathbf{P}(\mathbf{x}_1)$ with the parameter values $\mathbf{x}_1 = \{2, 4, 6, 10, 12\}$ and 12 additional measurement points $\mathbf{P}^+(\mathbf{x}_1^+)$, with the parameter values $\mathbf{x}_1^+ = \{14, 16, 18, 20, 24, 28, 32, 36, 40, 48, 56, 64\}$, which we will use for evaluation. In addition we repeat the measurements for each of these configurations five times to investigate the effects of system noise on model creation.

2.3.1 Creating Models for Individual Kernels. To create a performance model of a single application kernel, e.g., a CUDA kernel such as the `EigenMetaKernel`, we need to extrapolate the performance behavior found in the aggregated measurement data. The extrapolation is necessary as we only profile a small number of training steps and not the entire training process. Since every epoch consists of a particular and fixed number of training and validation steps, the total metric value for the whole epoch of the `EigenMetaKernel` $F_{kernel} = f_m(x_1, \ldots, x_m)$, that can be described as a function of the applications' configuration parameters x_m, can be calculated from the aggregated measurement data.

Therefore, we first define the following important variables: the batch size per worker B, the number of MPI ranks x_1, the number of samples in the training data set D_t, the number of samples in the validation data set D_v, the number of training steps per epoch n_t, the degree of data parallelism G, and the degree of model parallelism M. These analytical values are easy to identify for each deep learning application and have to be provided only once at the start of the modeling process. The rest of the process is fully automated. The variables G and M play an essential role by controlling the degree of data and model parallelism, enabling our approach to adapt the extrapolation methodology to the employed parallel strategy.

$$n_t = \lfloor (D_t/(G/M))/B \rfloor \tag{2}$$

$$n_v = \lfloor (D_v/(G/M))/B \rfloor \tag{3}$$

$$F_{kernel} = n_t \cdot \tilde{V}_{t_{kernel}} + n_v \cdot \tilde{V}_{v_{kernel}} \tag{4}$$

$$F_{kernel}(x_1, \ldots, x_m) = \sum_{k=1}^{h} c_k \cdot \prod_{l=1}^{m} x_l^{i_{kl}} \cdot log_2^{j_{kl}}(x_l) \tag{5}$$

Using these definitions, the number of training steps per epoch n_t can be calculated with Equation 2. Similarly, the number of validation steps n_v can be calculated with Equation 3. Subsequently, the total metric value of the `EigenMetaKernel` per epoch F_{kernel} can be calculated using Equation 4, where $\tilde{V}_{t_{kernel}}$ and $\tilde{V}_{v_{kernel}}$ are the median metric values of all MPI ranks of the kernel per training/validation step. This universal formula works for all performance metrics, e.g., application runtime, the number of visits, or the number of transferred bytes.

After calculating the derived metric value from the measured values, the next step is to create a performance model using the derived metric. To create a performance model, Extra-P expresses the effect of one or several parameters x_m on performance as a sum of terms consisting of products of polynomial and logarithmic expressions, where h is the maximum number of terms allowed per parameter. Therefore, first, a search space of possible model hypotheses is generated by instantiating the performance model normal form (PMNF), which is shown in Equation 5, with different exponents (i, j) chosen from a predefined set of exponents, e.g., $I = \{0, 1, 2\}$ and $J = \{0, 1\}$. The values of these sets can be adjusted to define the search space for the models. On the other hand, the parameters' values are bound by the user and system limitations, e.g., the maximum number of MPI ranks available on the given system. Subsequently, the coefficients c_k of the hypothesis are calculated using linear regression. Finally, the best model is identified using cross-validation, choosing the hypothesis with the smallest symmetric mean absolute percentage error (SMAPE).

Since Extra-Deep is based upon Extra-P, we employ its core modeling methodology. However, instead of modeling the measured values directly, we create performance models using the previously described derived metric values $F_{kernel}(x_m)$ by inserting them into the PMNF as shown in Equation 5. We apply this modeling methodology for all kernel models, including CUDA kernels, memset, memcopy, and NCCL operations, user-defined functions covered by the NVTX instrumentation, OS library, cuBLAS, cuDNN, and MPI function calls.

As it is generally unclear how many epochs a deep learning application has to be trained to reach the state of convergence, the performance models created by Extra-Deep always describe the application's performance for a specific metric and time frame, e.g., per epoch. A key strength of our approach is that we employ all information contained in the measurements for model creation. Even though collecting the empirical data requires some measurement overhead, it enables us to capture all relevant aspects of an application's performance, such as I/O, performed memory techniques, computation, and communication (inter/intra node), including the parallel strategy used for training. Herefore, we do not need to define complicated analytical models for each parallel strategy. In fact, the definition of our derived metric in combination with the PMNF, as shown in Equation 5, is sufficient to model all of the previously described effects on performance and support different types of parallel strategies. For this work, we evaluated our approach for data, tensor, and pipeline parallelism.

2.3.2 Creating Application Models. To create application models, e.g., to model a total metric value such as the application runtime per epoch, we apply the same methodology as for individual kernel models. First, we calculate the number of training n_t and validation steps n_v per epoch. Next, we aggregate the metric values of all kernels performing some kind of computation $\tilde{V}_{comp}$, communication $\tilde{V}_{comm}$, or memory operations $\tilde{V}_{mem}$ during training or validation. Then, the total metric value per epoch F_{epoch} can be calculated using Equation 6. Subsequently, we again create a search space of hypotheses by instantiating the PMNF with the derived metric value as shown in Equation 7, calculate the coefficients c_k, and identify the model with the best fit via the SMAPE metric.

$$F_{epoch} = n_t \cdot (\tilde{V}_{t_{comp}} + \tilde{V}_{t_{comm}} + \tilde{V}_{t_{mem}}) + n_v \cdot (\tilde{V}_{v_{comp}} + \tilde{V}_{v_{comm}} + \tilde{V}_{v_{mem}}) \tag{6}$$

$$F_{epoch}(x_m) = \sum_{k=1}^{h} c_k \cdot \prod_{l=1}^{m} x_l^{i_{kl}} \cdot log_2^{j_{kl}}(x_l) \tag{7}$$

$$F_{comp} = n_t \cdot \tilde{V}_{t_{comp}} + n_v \cdot \tilde{V}_{v_{comp}} \tag{8}$$

$$F_{comm} = n_t \cdot \tilde{V}_{t_{comm}} + n_v \cdot \tilde{V}_{v_{comm}} \tag{9}$$

$$F_{mem} = n_t \cdot \tilde{V}_{t_{mem}} + n_v \cdot \tilde{V}_{v_{mem}} \tag{10}$$

Similarly, as for the application runtime per epoch, we can model the total metric value for all computation F_{comp}, communication F_{comm}, and memory operations F_{mem} separately as shown in Equations 8-10. Hence, we simply aggregate all relevant kernels' metric

values executed during training $\tilde{V}_{t_{comp}}$ and validation $\tilde{V}_{v_{comp}}$ and multiply them with the number of training n_t, validation steps n_v.

Using the described approach, we create a performance model for our CIFAR-10 benchmark to model the training time per epoch as a function of the number of MPI ranks x_1. With the resulting model $T_{epoch}(x_1) = 158.58 + 0.58 \cdot x_1^{2/3} \cdot \log_2(x_1)^2$, we can easily predict the training time per epoch for any configuration and answer the in Section 1.1 defined question **Q1.** *How long does it take to train the ResNet-50 per epoch with a given resource allocation?* For a resource allocation of 40 MPI ranks, for example, a training time of 352.37 seconds per epoch is required.

3 ANALYZING APPLICATION PERFORMANCE

To gain further insights into an application's performance behavior at different scales, we leverage the created models to investigate the scalability, efficiency, and cost of the entire application as well as individual kernels. Afterwards, we outline how we use the gathered information to identify cost-effective training configurations.

3.1 Training Scalability and Bottlenecks

To identify potential bottlenecks in the application code, we employ the created runtime models and rank them by their growth trends according to the Big O notation. For example, we take the runtime models for all CUDA kernels executed on the GPU during training. By ranking them according to their growth trends, we can identify the functions that will become the performance bottleneck during training of a specific application configuration. Subsequently, one can investigate if it is possible to optimize the pinpointed kernels, e.g., using tensor fusion or other measures.

To further analyze application scalability, we introduce an additional performance metric, the speedup Δ, that quantifies the change in training performance for different application configurations. More specifically, it quantifies the gain or loss in training performance in percent by analyzing the change in runtime between a chosen initial measurement point P_1 and a second point P_k of the set $P(\mathbf{x}_1)$. P_1 is always the first measurement point from the set $P(\mathbf{x}_1)$ created from the parameter-value series $\mathbf{x}_1$, where x_1 is the number of MPI ranks. P_k is the point with the kth value in the series $\mathbf{x}_1 = (x_{1,1}, x_{1,2}, \ldots, x_{1,k})$. Since P_1 functions as the baseline for the calculation, the speedup for this point is always 0%. If $k = 1$ then $\Delta_{P_k} = 0\%$. For the measurement points P_k, the speedup can be calculated with Equation 11 if $k > 1$, where $T_1 = F_{kernel}(x_{1,1})$ is the derived baseline runtime of a kernel, an application phase, such as computation, or the entire application configuration $P_1(x_{1,1})$, and $T_k = F_{kernel}(x_{1,k})$ the runtime of the configuration $P_k(x_{1,k})$. By calculating the speedup for each point in the set $P(\mathbf{x}_1)$, we have enough data points to create a model for the speedup of an individual kernel, such as a CUDA kernel or the training time per epoch, as a function of the applications configuration parameters.

$$\Delta_{P_k} = (T_1 - T_k)/(T_1/100) \tag{11}$$

$$\Delta_{kernel}(x_m) = \sum_{k=1}^{h} c_k \cdot \prod_{l=1}^{m} x_l^{i_{kl}} \cdot log_2^{j_{kl}}(x_l) \tag{12}$$

Therefore, we instantiate the PMNF with the calculated speedups as shown in Equation 12 and create a model as previously described. By ranking the created models by their achieved speedup, this metric allows developers to easily identify the functions that benefit the most or least from scaling up the application.

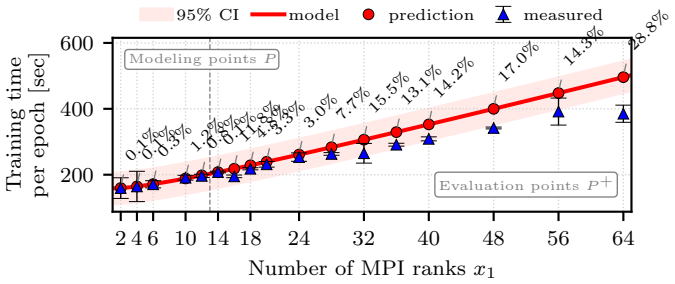


Figure 3: Training time per epoch breakdown of our model in comparison with measured runs. The vertical line indicates the separation between modeling and evaluation points.

Following this methodology, we can investigate the second question for our case study **Q2.** *How does the training performance change depending on the application configuration?* Figure 3 outlines the training time per epoch compared to the measured values and plots the created runtime model T_{epoch}. It shows how the training time per epoch changes as a function of the number of MPI ranks x_1. Since we used weak scaling for the performance experiments, we should ideally see a constant function, indicating that our code scales perfectly. Instead, one can observe that the training time increases with each additional node used. Overall the accuracy of the created model is very high, with a prediction error ranging between 0.1% and 1.2%. The model's predictive power is similarly high, reaching a maximum prediction error of 28.8% for the largest evaluation point at 64 MPI ranks. Another indication for the high prediction accuracy is that, besides for the last three evaluation points, all actual measured runtime values lie within the 95% confidence interval of our model. As shown by the error bars in Figure 3 the run-to-run variation between measurements with the same configuration is quite low, ranging between 0.6% and 13.9%, enabling us to create this accurate model. Furthermore, one can observe that the run-to-run variation increases the larger x_1, which makes it more difficult to predict the training time for an extrapolation point the further it is away from the set used for modeling.

Next, we investigate **Q3.** *Does the application suffer from any latent performance or scalability bottlenecks?* The biggest scalability bottleneck of the training process is the communication overhead required to exchange data and the DNNs gradients after each batch update. The more nodes are used for training, the higher the overhead and the more vulnerable the training process becomes to network/system noise and other contingencies. We identified this bottleneck by creating performance models for all application kernels and ranking them by their growth trends. The model of the communication time per epoch $T_{comm}(x_1) = 30.14 + 3.56 \cdot x_1^{2/3} \cdot 0.78 \cdot \log_2(x_1)^1$, which includes all MPI communication such as allreduce, allgather or broadcast, outlines how high the overhead actually is. The required communication time per training epoch increases from 34.41 seconds for two nodes to 296.57 seconds for 64 nodes.

$$\epsilon = \Delta_a / \Delta_t \qquad \Delta_t = (x_{1,k} - x_{1,1})/(x_{1,1}/100) \tag{13}$$

$$o = x_1 \cdot \varrho \qquad C_{kernel}(x_m) = T_{kernel}(x_m) \cdot o \tag{14}$$

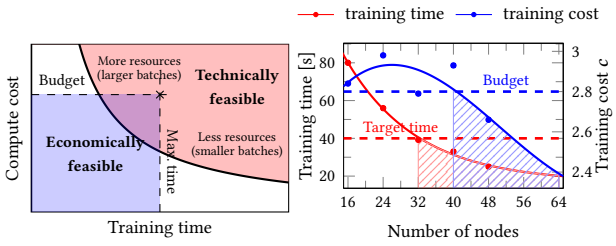
3.2 Parallel Training Efficiency

When scaling a distributed deep learning application, one must consider the diminishing returns of adding more resources for training. Therefore, Extra-Deep enables developers to analyze the parallel efficiency of their code for a specific application configuration $P(x_m)$. The parallel efficiency ϵ is defined as the ratio of the true speedup Δ_a to the theoretical speedup Δ_t, where Δ_t assumes that there is no parallelization overhead. Therefore, we can calculate the parallel efficiency using Equation 13. To identify the true speedup, we employ our in Section 3.1 described speedup model $\Delta_{kernel}(x_m)$, to calculate Δ_a for the point $P(x_m)$. The theoretical speedup can be calculated by quantifying the change of the number of MPI ranks x_1 from the initial measurement point P_1 to the point $P_k(x_1)$, which we want to analyze. Again we assume that P_1 is the first measurement point created from the parameter-value series $\mathbf{x}_1 = (x_{1,1}, x_{1,2}, \ldots, x_{1,k})$, and P_k it the kth point. The efficiency for the baseline application configuration P_1 is always 100%. The Δ_t for a point $P_k(x_1)$ can then be calculated with Equation 13. When we calculate the parallel efficiency for enough data points, we can create a performance model that describes the applications or a kernels efficiency as a function of its configuration parameters $\epsilon_{kernel}(x_m)$, following the same process as we used for the speedup.

3.3 Cost-Effective Training Configurations

An improved training performance by utilizing either more system resources, or accelerators such as GPUs, always comes at a certain price. Therefore, analysis is required to decide if this increase in training cost is worth the gain in performance on a case-by-case basis. To analyze the trade-off between the achieved training performance and its cost, we first define the training cost $C(x_m)$ of an individual kernel, an application phase, or the entire application for a specific application configuration $P(x_m)$ as the number of used CPU core hours. We calculate the training cost as shown in Equation 14, where $T_{kernel}(x_m)$ is the runtime (per epoch) of an application kernel, e.g., the `EigenMetaKernel` in seconds for the configuration $P(x_m)$, and o represents the total number of used CPU cores by all MPI ranks x_1. To obtain $T_{kernel}(x_m)$, we simply use the runtime model created by Extra-Deep and evaluate it for the configuration $P(x_m)$. The total number of CPU cores o is calculated by multiplying x_1 with the number of utilized CPU cores per MPI rank ϱ. If necessary, one can easily translate this universal definition into a monetary representation by multiplying $C(x_m)$ with the average cost per core hour. On the systems we used for our analysis, as on most of today's HPC systems, the utilized GPUs are not separately billed for. Hence, the cost for the used CPU core hours includes the cost of the GPUs. If this is not the case, Extra-Deep allows the user to specify a custom formula for the cost calculation.

Coming back to our case study, we are now able to answer question **Q4.** *How much does the training of the ResNet-50 cost per epoch for a given training configuration?* Following the described procedure, we create a cost model $C_{epoch}(x_1) = 0.082 \cdot x_1^{1.62}$ that describes the applications training cost per epoch. Using this model



(a) Tradeoff training time and cost (b) Cost effectiveness analysis

Figure 4: (a) Overview of the tradeoff between training time and computational cost [20]. (b) Example of the identification of cost-effective training configurations for strong scaling.

one can easily estimate the cost for distributed training, e.g., using 32 nodes, which would amount to 22.49 core hours.

Based on the previous definitions of the application speedup, efficiency, and cost, we can identify the most cost-effective training configuration for a specific goal, e.g., for a fixed computational budget of 10 000 CPU core hours or a maximum training time of ten hours. In a weak scaling scenario, this is rather simple. Using the runtime and cost models created by Extra-Deep, we can identify the in practice valid values for the applications' configuration parameters by restricting them to an interval where the training time and the cost are smaller or equal to the set limits (budget/runtime). In this case, the configuration with the smallest resource allocation will always be the one with the lowest cost and the highest parallel efficiency. This case also applies to our case study. Hence, we can answer question **Q5.** *What is the most cost-effective training configuration considering a particular budget or time frame?* Since we use weak scaling, the most cost-effective training configuration is the one with the smallest number of MPI ranks $x_1 = 2$.

For a strong scaling scenario, however, this determination is more complicated. In practice, one must distinguish between what is technically possible and economically feasible [20]. Figure 4a highlights the tradeoff between training time and compute cost, where we have a fixed training budget and a target time in which we need to finish the training process. In this case, valid training configurations can only be found where both areas overlap with each other. Figure 4b shows a more concrete example, where we set a maximum training time of 40 seconds and a training budget of 2.8 core hours. The hatched areas under the curves outline the in practice valid training configurations, i.e., the number of nodes meeting one or both set targets. Next, we use the created efficiency model to identify the configuration with the highest parallel efficiency in these intervals, which is the most cost-effective training configuration for this scenario.

4 EVALUATION

To evaluate Extra-Deep, we conducted various performance experiments for different training tasks, datasets, parallelization strategies, DNN architectures, and evaluation systems. We analyzed the created models in terms of their two most essential aspects: the achieved model accuracy and their predictive power. We define

Table 1: Overview of the systems and their hardware configuration that were used for the evaluation of Extra-Deep.

Name	System hardware
DEEP	75 nodes, 1x Intel Xeon Cascade Lake Silver 4215 CPU (8 cores, 16 threads), 48 GB DDR4 RAM (2 400 MHz), InfiniBand EDR (100 GBit/s), 1x Nvidia V100 GPU, without NCCL support
JURECA	192 nodes, 2x AMD EPYC 7742 CPUs (128 cores), 512 GByte DDR4 RAM, 2x InfiniBand HDR (NVIDIA Mellanox Connect-X6), 4x Nvidia A100 GPUs each, with NCCL support

model accuracy as a measure of how well a model fits the data points used for its creation. Therefore, we calculate the model accuracy as the percentage error of the predicted value compared to the measured metric value at each modeling point. We define predictive power as the extrapolation accuracy for a measurement point outside the parameter value range used for modeling. To determine the extrapolation accuracy for an evaluation point, we compare the model's prediction result with the actual measured metric value and calculate the percentage error. After outlining the evaluation methodology, we provide an analysis of the results, followed by a discussion of our observations.

4.1 Evaluation Methodology

Selected datasets & DNN architectures: To show that our approach supports a variety of deep learning applications such as image, speech, or NLP tasks, we choose five standard deep learning datasets commonly used for benchmarking: CIFAR-10, CIFAR-100, ImageNet, IMDB, and Speech Commands. We used the following DNN architectures for the benchmarks: a CNN with ten hidden layers (SpeechCommands), an NNLM (IMDB), a ResNet-50 (CIFAR-10, CIFAR-100), and an EfficientNet-B0 (ImageNet).

Application benchmarks: For our analysis, we created five synthetic benchmarks written in Python, one for each dataset. Each of these applications performs I/O, e.g., to load the input data or for checkpointing, performs data preprocessing, trains a DNN in parallel, and validates the training progress at the end of each epoch. In addition, we created three different implementations per benchmark, one for each parallel strategy we investigated. First, we employ pure data parallelism, currently the most frequently used parallel strategy for distributed DNN training. Second, we employ two forms of hybrid parallelism: a combination of data and model parallelism: tensor parallelism, and pipeline parallelism. Since pure model parallelism is executed in a serial fashion, we do not consider it for this analysis. The data parallel implementation uses TensorFlow and Horovod, while the tensor parallel code uses Mesh-tensorflow [33], and the pipeline parallel code uses PyTorch and Horovod. We instrument the codes using Extra-Deep's built in instrumentation tool and the NVTX library.

Evaluation environments: The performance experiments were conducted on two different supercomputers: the Deep (Extreme Scale Booster) and the JURECA (DC Module) system at Jülich Supercomputing Centre (JSC). Table 1 outlines their hardware configurations in detail. On both systems, we were able to access a maximum of 64 nodes at once.

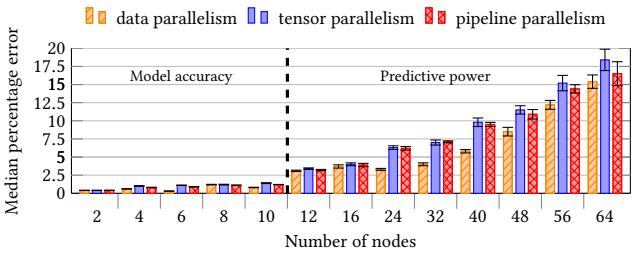


Figure 5: Comparison of the model accuracy (2-10 nodes) and predictive power (12-64 nodes) for the training time per epoch models for different parallel strategies on JURECA. The bars show the MPE of all benchmarks. The error bars show the 95% confidence intervals of the reported MPEs.

Experiment configuration: To evaluate the accuracy and predictive power of the created models, we focus on modeling their performance as a function of the resource allocation, i.e., the number of MPI ranks x_1. To facilitate a more accessible analysis and increase the paper's readability, we focused on the detailed evaluation of one model parameter, including all of Extra-Deep's features, rather than discussing several model parameters. However, as hyperparameters such as the learning rate or activation function play an important role in training, we conducted several test runs for each benchmark to select an adequate set of values to ensure high GPU utilization. In addition, we naturally adjust those values as we scale the number of MPI ranks according to the utilized parallel strategy. Finally, we run each performance experiment twice, once using weak scaling and once using strong scaling. To conduct the required empirical measurements for the experiments on the Deep system, we used the parameter-value set $\mathbf{x}_1 = \{2, 4, 6, 8, 10\}$ for x_1 to obtain five points $\mathbf{P}(\mathbf{x}_1) = \{P_1, \ldots, P_5\}$ for modeling. We then measured eight additional points $\mathbf{P}^+ = \{P_1^+, \ldots, P_8^+\}$ using $\mathbf{x}_1^+ = \{12, 16, 24, 32, 40, 48, 56, 64\}$ to evaluate the predictive power of the created models. For the experiments on the JURECA system, we used the parameter-value sets $\mathbf{x}_1 = \{8, 16, 24, 32, 40\}$ for $P(\mathbf{x}_1)$ and $\mathbf{x}_1^+ = \{12, 48, 64, 96, 128, 160, 192, 224, 256\}$ for $\mathbf{P}^+(\mathbf{x}_1^+)$.

4.2 Extra-Deep's Accuracy and Predictive Power

In the following Section, we outline the evaluation results of Extra-Deep regarding its predictive power and model accuracy for different parallel strategies, system architectures and communication patterns, and application types and DNN architectures. We then examine the required profiling overhead with and without our efficient measurement sampling strategy. Finally, we discuss further evaluation results. The figures present the combined results of both our weak and strong scaling experiments.

4.2.1 Parallel strategies. First, we investigated the capability of our approach to accurately predict the training time per epoch for different types of parallel strategies. Therefore, we calculate each benchmark's percentage error at the modeling and evaluation points for the training time per epoch models for all parallel strategies. We then calculate the median percentage error (MPE) for all benchmarks per strategy from these values. As shown by

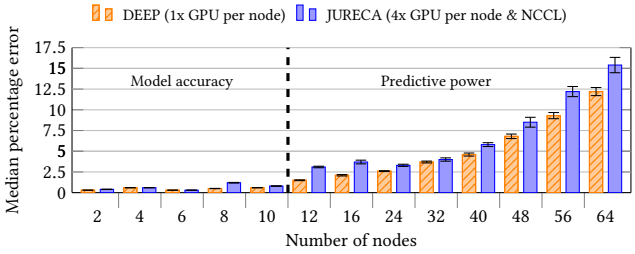


Figure 6: Comparison of the model accuracy (2-10 nodes) and predictive power (12-64 nodes) for the training time per epoch models for data parallelism. The bars show the MPE of all benchmarks for each system.

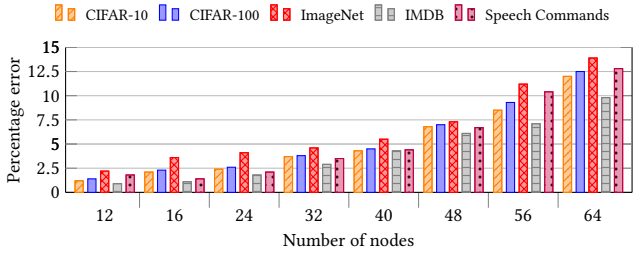


Figure 7: Comparison of the predictive power for the runtime per epoch models for data parallel training per benchmark on DEEP. The bars show the percentage error of the models for the evaluation points.

the results in Figure 5 the model accuracy is very high for all strategies with an MPE ranging between 0.4% and 1.4%. The predictive power of the created models for the evaluation points (12 to 64 nodes) is, as expected, decreasing linearly. The further away the extrapolation point, the greater the MPE. Furthermore, the results show that the tensor and pipeline parallelism models are slightly less accurate. However, this is also expected as forms of hybrid parallelism are more challenging to predict due to their much more complex communication and synchronization patterns. The overall results suggest that our approach provides accurate predictions for all parallel strategies with a maximum error of 18.4% for tensor parallelism and 64 nodes. For this experiment, the degree of data parallelism G was defined as $G = x_1$ (4× the number of nodes), and the degree of model parallelism M was set as $M = 1$ for the data parallel benchmarks, while $G = \frac{x_1}{4}$ (no. nodes) and $M = 4$ for the benchmarks using tensor, pipeline parallelism.

4.2.2 System architecture & communication patterns. Figure 6 outlines the result of the second experiment, quantifying the ability of our approach to deal with different types of system architectures and communication paradigms. As for the parallel strategies analysis, we again calculated the MPE for the training time per epoch models for all benchmarks and systems. The experiment was conducted on both evaluation systems using data parallelism. The GPU accelerated nodes of Deep have only one GPU per node and do not support NCCL communication. The GPU accelerated nodes of JURECA have four GPUs per node and support NCCL. For both systems, the degree of data parallelism was set to the number of MPI ranks ($G = x_1$) to utilize all available GPUs. The results shown in Figure 6 demonstrate that our approach can accurately predict the impact of intra-, inter-node, and NCCL communication on the application's performance behavior. As for the different parallel strategies, the model accuracy is very high, with an MPE ranging between 0.3% and 1.2%. The prediction error increases with each evaluation point reaching at most 15.4% for JURECA and 64 nodes. As expected, the models for JURECA are slightly less accurate, which can be explained by the increased complexity of predicting inter-node and NCCL communication between the GPUs.

4.2.3 Application types & DNN architectures. To demonstrate that our approach works for different types of deep learning applications, as well as DNN architectures, the third experiment of the evaluation investigates the differences in model accuracy and predictive power of the created training time per epoch models for the different benchmarks. Figure 7 outlines its results as percentage errors for each benchmark. As the model accuracies are very similar to the previously shown results, with a percentage error ranging between 0.4% and 1.4%, we have refrained from plotting them explicitly. However, there are minor but apparent differences between the benchmarks for predictive power. Again the percentage error is increasing steadily for all benchmarks as the number of nodes grows. While the models for the IMDB benchmark are the most accurate, the ones for ImageNet are the worst, with a maximum difference in the percentage error of 4.1% for 64 nodes. The IMDB dataset has only 50 000 samples, which is relatively small compared to the ImageNet dataset, which has more than 1.2 million samples requiring a much higher computational effort for training. Another significant difference is the DNN architecture used for training. The EfficientNet-B0 used for ImageNet is much larger and more complex than the NNLM employed for IMDB. Therefore, the results in Figure 7 clearly show that the training process's complexity and overhead significantly impact our approach's model accuracy. However, the results, a percentage error of at most 13.9% for ImageNet and 64 nodes, show that our method accurately predicts the performance of large-scale applications.

4.2.4 Profiling overhead. To quantify the effectiveness of our efficient measurement sampling strategy, we measured the profiling overhead and the execution time for several epochs of training for all benchmarks using data parallelism and 64 nodes on Deep. We then calculated the median execution time and the median profiling time per epoch for each benchmark. Moreover, we disregarded application kernels not executed during training as they contribute only a small amount to the overall application runtime. The results of this experiment are outlined in Figure 8. Without our sampling strategy it is necessary to profile entire training epochs resulting in a very large measurement overhead as shown by the bars with the line pattern in Figure 8, which would make the analysis of large-scale deep learning applications such as GPT-3 too expensive to be practical. Compared to the results with the applied sampling strategy, the average measurement overhead is reduced by about 94.9%. Furthermore, the results show that the strategy is especially effective for large and long-running benchmarks such as ImageNet and less effective for short-running benchmarks such as IMDB. The

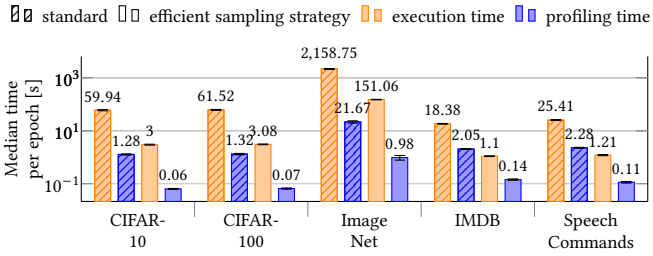


Figure 8: Comparison of the standard profiling overhead and execution time per epoch vs. our efficient measurement sampling strategy for data parallel training.

Table 2: Further evaluation results for other types of models and metrics. The table shows the average MPE for all benchmarks and both evaluation systems using data parallelism.

		Evaluation Points (Number of Nodes)						
Model type	**Metric**	**24**	**32**	**40**	**48**	**56**	**64**	**Model no.**
CUDA	time	1.4%	3.3%	3.0%	5.6%	10.2%	15.6%	845
kernels	visits	0.5%	1.1%	1.4%	2%	1.8%	3.1%	845
NVTX	time	1.4%	3.3%	3.0%	5.6%	10.2%	15.6%	115
func.	visits	0.5%	1.1%	1.5%	2.3%	4.5%	3.2%	115
OS func.	time	2.3%	1.8%	4.5%	7.8%	10.5%	13.6%	230
cuBLAS	time	4.1%	7.2%	9.1%	10.4%	14.3%	18.3%	50
cuDNN	time	5.9%	8.9%	7.6%	11.3%	14.2%	18.9%	40
MPI	time	3.9%	5.1%	7.2%	10.1%	15.2%	22.4%	35
Memory	time	2.5%	3.3%	2.2%	4.8%	5.5%	7.9%	25
ops.	bytes	1.4%	1.7%	2.3%	4.4%	6.5%	7.2%	25

profiling overhead of our approach amounts to an average of about 5.4% of the total execution time per epoch over all benchmarks. This percentage does not change when profiling with or without the efficient sampling strategy, as we effectively only reduce the number of profiled epochs and training steps. The measurement overhead per step/epoch, however, is still the same.

4.2.5 Further evaluation results. Table 2 describes the results of our remaining performance experiments, outlining the enormous number of performance models we evaluated. It shows that the prediction accuracy of Extra-Deep is even better for models of individual application kernels such as the instrumented NVTX functions. The highlighted results outline the most important findings of the experiments. First, we found that for all model types, the number of visits is generally easier to predict than the runtime. For the CUDA kernels, for example, the MPE for the number of visits is only 3.1% compared to 15.6% for the runtime models for 64 nodes. Furthermore, we found that the runtime of the MPI functions is generally harder to predict, which is outlined by an MPE of 3.9% for 24 nodes and 22.4% for 64 nodes. Finally, we found that our predictions for the runtime and the number of transferred bytes for the memory operations are exceptionally accurate, indicated by an MPE of 7.9% and 7.2% for 64 nodes.

4.3 Discussion of the Evaluation Results

The outlined results, an average model accuracy of 97.6% and an average prediction accuracy of 93.6%, emphasize that our approach accurately predicts the performance behavior of distributed deep learning applications, independently of application type, DNN architecture, system hardware, and utilized parallel strategy. These averages are calculated using the model and prediction accuracies from all the models that we created while conducting our performance experiments, evaluated at an evaluation point four times the scale than the ones used for modeling.

If the MPE of the created models for 64 nodes seems high for some of the performance models, one must consider that the parameter values used for generating the models are more than six magnitudes smaller. The source of the prediction error is simply the effect of trying to predict behavior at a much larger scale and the effects of system noise, such as OS noise or concurrently running jobs, on the performance measurements. Depending on the system architecture, hardware, and configuration, run-to-run variations of 15% or more are common, when measuring an application with the exact same set of execution parameters. On the measurements conducted for our evaluation we found an average run-to-run variation of about 12.6% on DEEP and 17.4% on JURECA. Thus, prediction errors for 64 nodes between 15-20% are a desirable outcome.

Furthermore, the presented results depict the worst-case scenario, where one uses only a minimal and very cheap (starting point) set of measurements for model creation. Consequently, this error can be drastically reduced by measuring one or two additional measurement points closer to the evaluation target. A limitation of our approach is that we can not predict behavior that is not present in the conducted performance measurements. Communication algorithms and performed memory techniques might change depending on the application scale. Therefore, a clear expectation of the model's target scale helps to identify the correct application configurations for profiling. A prediction of an applications training time per epoch for 1 024 MPI ranks x_1 based on the measurement points $P(\mathbf{x}_1)$, where $\mathbf{x}_1 = \{2, 4, 6, 8, 10\}$ is simply unrealistic. Though, a prediction with $\mathbf{x}_1 = \{8, 16, 32, 64, 128\}$ is possible. Using our efficient sampling strategy, measuring additional points close to the desired prediction scale is very cheap, which makes it impossible to miss important application behavior.

In comparison to other modeling approaches, Extra-Deep works independently of application type or used parallel strategy. Approaches such as PALEO [28] or ParaDL [17] provide accurate analytical models and performance predictions, however, they require expert knowledge, manual analysis, and are limited to application level models. Whereas Extra-Deep automatically creates performance models for each instrumented application kernel. A detailed side-by-side comparison of its prediction accuracy with other analytical/empirical modeling approaches is, therefore, currently not possible, as there is no other tool that can automatically produces models for all kernels of an application.

5 CONCLUSION

We presented Extra-Deep, our novel modeling framework, to analyze the training performance and identify cost-effective training configurations for distributed deep learning. Using only a few small-scale performance experiments, it automatically creates kernel and application models for all instrumented application functions without requiring expert knowledge or manual analysis. Thus, enabling

developers to analyze application scalability, efficiency, and cost. Our evaluation showed that it accurately predicts performance metrics, such as runtime, as a function of an application's execution parameters, e.g, the number of MPI ranks, for different parallel training strategies, reaching an average prediction accuracy of 93.6%. Using an efficient sampling strategy that reduces the profiling time for the required empirical measurements by, on average, about 94.9%, compared to measuring full training runs, it enables an automated performance analysis even for long-running and large-scale applications requiring several hours of training per epoch.

ACKNOWLEDGMENTS

This work was funded by the German Research Foundation (DFG) – Project No. 449683531. Moreover, it obtained funding from the European Commission and the German Federal Ministry of Education and Research (BMBF) under the EuroHPC Programmes DEEP-SEA (GA No. 955606, BMBF funding No. 16HPC015) and ADMIRE (GA No. 956748, BMBF funding No. 16HPC006K), which receive support from the European Union's Horizon 2020 programme and DE, FR, ES, GR, BE, SE, GB, CH (DEEP-SEA) and DE, FR, ES, IT, PL, SE (ADMIRE), respectively. Furthermore, the authors gratefully acknowledge the German Federal Ministry of Education and Research (BMBF) and the Hessian Ministry for Science and the Arts (HMWK) for supporting this work as part of the NHR funding and for the computing time provided on the high-performance computer Lichtenberg at the NHR Center TU Darmstadt. Finally, the authors feel indebted for further computing time on the supercomputers DEEP and JURECA at Jülich Supercomputing Centre.

REFERENCES

[1] 2023. Extra-Deep. https://github.com/extra-p/extradeep
[2] 2023. PyTorch Profiler. https://pytorch.org/docs/stable/profiler.html
[3] 2023. TensorFlow Profiler. https://www.tensorflow.org/guide/profiler
[4] Dieter an Mey, Scott Biersdorf, Christian Bischof, Kai Diethelm, Dominic Eschweiler, Michael Gerndt, Andreas Knüpfer, Daniel Lorenz, Allen Malony, Wolfgang E Nagel, et al. 2011. Score-P: A unified performance measurement system for petascale applications. In *Competence in High Performance Computing 2010*. Springer, 85–97.
[5] Nicola Bombieri, Federico Busato, and Franco Fummi. 2016. A fine-grained performance model for GPU architectures. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1267–1272.
[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
[7] Bo Chen, Jiewei Cao, Alvaro Parra, and Tat-Jun Chin. 2019. Satellite pose estimation with deep landmark regression and nonlinear pose refinement. In *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*. 0–0.
[8] Diego Didona, Francesco Quaglia, Paolo Romano, and Ennio Torre. 2015. Enhancing performance prediction robustness by combining analytical modeling and machine learning. In *Proceedings of the 6th ACM/SPEC international conference on performance engineering*. ACM, 145–156.
[9] Dmitry Duplyakin, Jed Brown, and Robert Ricci. 2016. Active learning in performance analysis. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 182–191.
[10] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *The Journal of Machine Learning Research* 23, 1 (2022), 5232–5270.
[11] Jiazhen Gu, Huan Liu, Yangfan Zhou, and Xin Wang. 2017. Deepprof: Performance analysis for deep learning applications via mining gpu execution patterns. *arXiv preprint arXiv:1707.03750* (2017).
[12] Md Shahriar Iqbal, Lars Kotthoff, and Pooyan Jamshidi. 2019. Transfer Learning for Performance Modeling of Deep Neural Network Systems. In *USENIX Conference on Operational Machine Learning (OpML 19)*. 43–46.
[13] Matthijs Jansen, Valeriu Codreanu, and Ana-Lucia Varbanescu. 2020. DDLBench: Towards a Scalable Benchmarking Infrastructure for Distributed Deep Learning. In *IEEE/ACM Fourth Workshop on Deep Learning on Supercomputers*. IEEE, 31–39.
[14] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. 2021. The symptoms, causes, and repairs of bugs inside a deep learning library. *Journal of Systems and Software* 177 (2021), 110935.
[15] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. *Proceedings of Machine Learning and Systems* 1 (2019), 1–13.
[16] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.
[17] Albert Njoroge Kahira, Truong Thao Nguyen, Leonardo Bautista Gomez, Ryousei Takano, Rosa M Badia, and Mohamed Wahib. 2021. An oracle for guiding large-scale model/hybrid parallel training of convolutional neural networks. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*. 161–173.
[18] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. 2019. Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 754–768.
[19] Ying-Chiao Liao, Chuan-Chi Wang, Chia-Heng Tu, Ming-Chang Kao, Wen-Yew Liang, and Shih-Hao Hung. 2020. PerfNetRT: Platform-Aware Performance Modeling for Optimized Deep Neural Networks. In *International Computer Symposium (ICS)*. IEEE, 153–158.
[20] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. 2018. An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162* (2018).
[21] NVIDIA. 2020. NVIDIA Tools Extension Library (NVTX). https://docs.nvidia.com/nsight-visual-studio-edition/2020.1/nvtx/index.html
[22] NVIDIA. 2023. CUDA Profiling Tools Interface. https://developer.nvidia.com/cuda-profiling-tools-interface
[23] NVIDIA. 2023. Nsight Systems. https://developer.nvidia.com/nsight-systems
[24] Ziqian Pei, Chensheng Li, Xiaowei Qin, Xiaohui Chen, and Guo Wei. 2019. Iteration time prediction for CNN in multi-GPU platform: modeling and analysis. *IEEE Access* 7 (2019), 64788–64797.
[25] Damiano Perri, Paolo Sylos Labini, Osvaldo Gervasi, Sergio Tasso, and Flavio Vella. 2019. Towards a learning-based performance modeling for accelerating deep neural networks. In *International Conference on Computational Science and Its Applications*. Springer, 665–676.
[26] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1027–1038.
[27] Sarunya Pumma, Min Si, Wu-Chun Feng, and Pavan Balaji. 2019. Scalable deep learning via I/O analysis and optimization. *ACM Transactions on Parallel Computing (TOPC)* 6, 2 (2019), 1–34.
[28] Hang Qi, Evan R Sparks, and Ameet Talwalkar. 2017. Paleo: A performance model for deep neural networks. In *International Conference on Learning Representations*.
[29] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.
[30] Md Aamir Raihan, Negar Goli, and Tor M Aamodt. 2019. Modeling deep learning accelerator enabled gpus. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 79–92.
[31] Marcus Ritter, Alexander Geiß, Johannes Wehrstein, Alexandru Calotoiu, Thorsten Reimann, Torsten Hoefler, and Felix Wolf. 2021. Noise-Resilient Empirical Performance Modeling with Deep Neural Networks. In *Proceedings of the 35th International Parallel and Distributed Processing Symposium (IPDPS), Portland, Oregon, USA*. IEEE, 23–34.
[32] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).
[33] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. 2018. Mesh-tensorflow: Deep learning for supercomputers. *Advances in neural information processing systems* 31 (2018).
[34] Shaohuai Shi, Qiang Wang, and Xiaowen Chu. 2018. Performance modeling and evaluation of distributed deep learning frameworks on gpus. In *IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, (DASC)*. IEEE, 949–957.
[35] Alexander Ulanov, Andrey Simanovsky, and Manish Marwah. 2017. Modeling scalability of distributed machine learning. In *33rd International Conference on Data Engineering (ICDE)*. IEEE, 1249–1254.
[36] Mohammad Wardat, Breno Dantas Cruz, Wei Le, and Hridesh Rajan. 2022. DeepDiagnosis: automatically diagnosing faults and recommending actionable fixes in deep learning programs. In *Proceedings of the 44th International Conference on Software Engineering*. 561–572.
[37] Ming Yan, Junjie Chen, Xiangyu Zhang, Lin Tan, Gan Wang, and Zan Wang. 2021. Exposing numerical bugs in deep learning via gradient back-propagation. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 627–638.