Performance Modelling For Scalable Deep Learning

T Kavarakuntla

PHD 2023

Performance Modelling For Scalable Deep Learning

Tulasi Kavarakuntla

A thesis submitted in partial fulfilment of the requirements of

Manchester Metropolitan University

for the degree of Doctor of Philosophy

Department of Computing and Mathematics

Manchester Metropolitan University

2023

Contents

List of figures

List of publications

1. T. Kavarakuntla, L. Han, H. Lloyd, A. Latham, and S. B. Akintoye, “Performance analysis of distributed deep learning frameworks in a multi-gpu environment,” in 2021 20th International Conference on Ubiquitous Computing and Communications (IUCC/CIT/DSCI/SmartCNS), IEEE, 2021, pp. 406–413, London.

2. T. Kavarakuntla, L. Han, H. Lloyd, A. Latham, A. Kleerekoper, and S. B. Akintoye, “A Generic Performance Model for Deep Learning in a Distributed Environment, ” in 2022 IEEE Symposium Series on Computational Intelligence (SSCI). IEEE, 2022, pp.191.

3. T. Kavarakuntla, L. Han, H. Lloyd, A. Latham, A. Kleerekoper, and S. B. Akintoye, “A generic performance model for deep learning in a distributed environment,” submitted at IEEE access arXiv preprint arXiv:2305.11665, 2023 (under review).

Abstract

Performance modelling for scalable deep learning is very important to quantify the efficiency of large parallel workloads. Performance models are used to obtain run-time estimates by modelling various aspects of an application on a target system. Designing performance models requires comprehensive analysis in order to build accurate models. Limitations of current performance models include poor explainability in the computation time of the internal processes of a neural network model and limited applicability to particular architectures.

Existing performance models in deep learning have been proposed, which are broadly categorized into two methodologies: analytical modelling and empirical modelling. Analytical modelling utilizes a transparent approach that involves converting the internal mechanisms of the model or applications into a mathematical model that corresponds to the goals of the system. Empirical modelling predicts outcomes based on observation and experimentation, characterizes algorithm performance using sample data, and is a good alternative to analytical modelling. However, both these approaches have limitations, such as poor explainability in the computation time of the internal processes of a neural network model and poor generalisation. To address these issues, hybridization of the analytical and empirical approaches has been applied, leading to the development of a novel generic performance model that provides a general expression of a deep neural network framework in a distributed environment, allowing for accurate performance analysis and prediction. The contributions can be summarized as follows:

In the initial study, a comprehensive literature review led to the development of a performance model based on synchronous stochastic gradient descent (S-SGD) for analysing the execution time performance of deep learning frameworks in a multi-GPU environment. This model's evaluation involved three deep learning models (Convolutional Neural Networks (CNN), Autoencoder (AE), and Multilayer Perceptron (MLP)), implemented in

three popular deep learning frameworks (MXNet, Chainer, and TensorFlow) respectively, with a focus on following an analytical approach. Additionally, a generic expression for the performance model was formulated, considering intrinsic parameters and extrinsic scaling factors that impact computing time in a distributed environment. This formulation involved a global optimization problem with a cost function dependent on unknown constants within the generic expression. Differential evolution was utilized to identify the best fitting values, matching experimentally determined computation times. Furthermore, to enhance the accuracy and stability of the performance model, regularization techniques were applied. Lastly, the proposed generic performance model underwent experimental evaluation in a real-world application. The results of this evaluation provided valuable insights into the influence of hyperparameters on performance, demonstrating the robustness and applicability of the performance model in understanding and optimizing model behavior.

Acknowledgements

Firstly, I would like to sincerely thank my Director of Studies Prof. Liangxiu Han for the extensive amount of assistance provided throughout the entire PhD, and her unwavering support and guidance, this research would not have been possible. Her dedication and commitment to my academic journey have been invaluable.

I would like to sincerely thank my supervisor, Dr. Huw Lloyd, for the extensive amount of assistance provided throughout the entire PhD and without whom I probably would not even be doing this PhD, as well as Dr. Annabel Latham, Dr. Samson Akintoye and Dr. Anthony Kleerekoper for the extremely valuable assistance and feedback provided during this process.

I would like to thank Dr. Sravanthi Sashikumar for her valuable support to get admission in MMU university. I would like to thank my Government for providing sponsorship that has allowed me to pursue my studies at MMU University.

Finally, most important thanks go to my parents, family members, and especially my husband Dr. Subramanyam Vasanthapalli, for his constant support, understanding, and affectionate encouragement. Additionally, I extend my greatful thanks to my children, Bhavagna and Kethan, for their unwavering love and support throughout my journey.

Chapter 1

Introduction

This chapter presents an overview of the background and motivation of this research introducing performance modelling in scalable deep learning in a distributed environment.

1.1 Background and Motivation

Deep learning [1], a branch of machine learning [2], has seen widespread adoption and emerged as a fundamental technology across diverse domains such as computer vision, natural language processing, speech recognition, autonomous driving, recommendation systems, healthcare diagnostics, and financial forecasting. The use of deep neural network architectures, such as Googlenet [3], ResNet [4], VGG net [5], and Deep CNN [6], has propelled advancements in these domains. However, the training and deployment of these architectures require substantial computational resources. Training with a large amount of data requires a parallelised and distributed environments employing techniques such as data parallelism, model parallelism, pipeline parallelism, and hybrid parallelism.

Performance modelling [7] [8] plays a critical role in optimizing the efficiency and performance of these large-scale parallel workloads in deep learning. Performance models are used to obtain run-time estimates by modelling various aspects of an application on a target system. However, accurate performance modelling is a challenging task. Existing performance models are broadly categorised into two methodologies: analytical modelling and empirical modelling. Analytical performance modeling [9] offers a transparent and systematic approach to unraveling the intricate workings of these models and their interactions with the surrounding system components. In analytical modeling, a model's or an

application's internal mechanism is converted into a mathematical model corresponding to the system's goals, which can significantly expedite the creation of a performance model for the intended system.

There are several significant existing works in the field of analytical performance modeling for deep learning. Yan *et al.* [10] developed a performance model to evaluate the impact of partitioning and resourcing decisions on the overall performance and scalability of distributed system architectures using the Adam DL framework. Kim *et al.* [11] conducted a comparative analysis of deep learning frameworks in single and multi-GPU environments, exploring the performance implications of different convolution algorithms. Qi *et al.* [12] proposed a performance model named Paleo, which considered communication schemes, network architecture, and parallelization strategies to predict deep neural network performance. Castello *et al.* [13] developed an analytical model to evaluate the scalability of data parallelism and model parallelism for distributed deep learning training of convolutional neural networks. Jia *et al.* [14] focused on analyzing the impact of network topologies, communication patterns, and batch sizes on the performance and consistency of distributed deep learning applications. However, despite the valuable insights provided by these analytical models, they have limitations in terms of generalization and explainability. The reliance on simplifications and assumptions can hinder their ability to generalize well to diverse deep learning architectures, datasets, and hardware configurations. As a result, the analytical estimates may not accurately reflect the real-world performance scenarios. Additionally, the lack of explainability in these models limits understanding of the underlying mechanisms and decision-making processes of deep learning systems. Although they can predict performance, they often fall short in providing clear explanations of the factors influencing the outcomes.

Empirical modelling [15] is a good alternative to analytical models. In this approach, modelling predicts the outcome of an unknown set of system parameters based on observation and experimentation. It characterises an algorithm's performance across problem instances and parameter configurations based on sample data. Empirical models predict the output of a new configuration on the target machine. Several empirical modeling studies have contributed to the field of performance modeling in deep learning. Yufei *et al.* [16] developed a performance model for FPGA-based accelerators, achieving close predictions to actual test results but lacking explainability. Z. Lin *et al.* [17] proposed a model con-

sidering network topology and communication patterns, demonstrating higher accuracy in predicting training time but with limited generalizability. Andre Viebke [18] focused on predicting execution time on Intel's Many Integrated Core (IMIC) architectures with high accuracy, although lacking generalizability and detailed explanation. Rakshith *et al.* [19] evaluated the performance of the Horovod framework for image classification tasks, providing optimization recommendations but using specific experimental configurations. These studies have provided valuable insights into predicting and optimizing performance in various deep learning scenarios. However, it is important to acknowledge the common limitations in these empirical modeling approaches. One limitation is the lack of generalizability, as many of these models are evaluated on specific architectures, datasets, or experimental configurations, which may not fully represent the diversity of real-world scenarios. Additionally, the level of explainability in these models are limited, making it challenging to gain a comprehensive understanding of the underlying mechanisms driving performance.

To address the limitations of both analytical and empirical performance modeling in deep learning performance estimation, it is critical to develop a comprehensive performance model that can overcome these challenges.

1.2 Aims and Objectives

The main aim of this study is to develop a generic performance model for scalable deep learning system in a distributed environment, achieving both generalizability and explainability of internal processess of deep neural networks. The study will employ both analytical and empirical approaches to model the impact of various internal and external parameters, including filter size, pooling size, batch size, and number of GPUs, on the performance of the system. The proposed performance model will be able to optimize the performance of distributed deep learning systems and provide insights into the factors that affect system performance in a distributed environment. Towards this aim, the following objectives have been identified:

- Objective 1: Conduct a comprehensive review of the literature on neural network models, with attention to distributed and parallelized algorithms run on different

frameworks. By critically evaluating the existing research, this objective aims to establish a comprehensive understanding of the current state-of-the-art in performance models in deep learning and identify potential research gaps and areas for further investigation.

- Objective 2: Develop a performance model based on synchronous stochastic gradient descent (S-SGD) to analyze the execution time performance of deep learning frameworks in a multi-GPU environment and evaluate the model using three deep learning models (Convolutional Neural Networks, Autoencoder, and Multilayer Perceptron), each implemented in three frameworks (MXNet, Chainer, and Tensorflow) respectively. Additionally, consider load imbalance factors that may affect the scalability of deep learning models.
- Objective 3: Develop a generic performance model considering the influence of intrinsic parameters and extrinsic scaling factors that affect computing time in a distributed environment and formulate the generic expression as a global optimization problem using regularization on a cost function written in terms of the unknown constants in the generic expression. The model has to be solved using differential evolution to find the best-fitting values to match experimentally determined computation times. This type of generic performance model is a novel contribution in the deep neural networks domain.
- Objective 4: Apply the developed performance model to a real world application and analyze how far the performance model can be feasible in deep neural network domain.

Each of these objectives contributes to the overall aim of this thesis, which is to develop a performance model for scalable deep learning in a distributed environment using deep learning frameworks. The performance model developed in this thesis can be used to improve the speed and efficiency of deep learning in a distributed environment through parallelization and distributed strategies. It is designed with generalizability in mind and prioritizes explainability, making it a valuable tool for practitioners in the field of deep learning.

1.3 Contributions

In the course of this work, a number of original contributions have been made to the field of performance modelling of deep learning. These contributions are:

- Conducted a comprehensive review of the literature on neural network models, with attention to distributed and parallelized algorithms run on different frameworks. Most typical deep learning models such as Multi-layer Perceptron model, convolutional neural network and Autoencoder were included in the study, along with related work on performance modelling.
- Developed a performance model to analyze iteration time layerwise in a deep neural network in various frameworks using synchronous stochastic descent algorithm. Built a performance model based on synchronous stochastic gradient descent (S-SGD), to analyze the execution time performance of deep learning frameworks in a multi-GPU environment. Considered load imbalance factor and mini-batch time (time taken to divide mini-batches) and evaluated the model using three deep learning models (Convolutional Neural Networks, Autoencoder, and Multilayer Perceptron), each implemented in three frameworks (MXNet, Chainer, and Tensorflow) respectively. Using experimental data, analyze the effect of load imbalance on the scalability of deep learning models, concluding that it is an important contribution to parallel inefficiency.
- Developed a generic expression for a performance model considering the influence of intrinsic parameters and extrinsic scaling factors that affect computing time in a distributed environment. Thereafter, formulated this as a global optimization problem using regularization on a cost function in terms of the unknown constants in the generic expression, and formulated the problem as a global optimization task. Solved the optimization problem using differential evolution to find the best-fitting values to match experimentally determined computation times.
- Applied the developed performance model to a real-time application and analyzed how far the performance model can be generalizable and explainable for a complex and large dataset, demonstrating that the performance model has generalization and explainability.

1.4 Thesis Structure

Chapter 2 provides a comprehensive literature review study in relation to deep learning architectures, distributed deep learning, deep learning frameworks, performance modelling (focusing on deep learning in a distributed environment) and the differential evolution algorithm utilised to fulfil the aims of the thesis.

Chapter 3 introduces a performance model based on synchronous stochastic gradient descent (S-SGD) to analyse the execution time performance of deep learning frameworks in a multi-GPU environment and evaluated the model using three deep learning models (Convolutional Neural Networks, Autoencoder and Multilayer Perceptron), each implemented in three frameworks (MXNet, Chainer and Tensorflow) respectively.

Chapter 4 presents a generic expression for a performance model considering the influence of intrinsic parameters and extrinsic scaling factors that affect computing time in a distributed environment. It formulates the generic expression as a global optimization problem using a cost function written in terms of the unknown constants, and solves it using differential evolution to find the best fitting values to match experimentally determined computation times. Compared differential evolution using with and without regularization techniques, and the results found that differential evolution using regularization gives a generalized model with improved performance. Also, the predictive performance is comparable to black box machine learning models.This type of generic performance model is a novel contribution in the deep neural networks domain.

Chapter 5 discusses the performance of the proposed model training a 3D-ResAttNet architecture, on a popular, complex and large dataset i.e., ADNI dataset using PyTorch deep learning framework in a multi-GPU environment. The results shown that the performance model can be generalizable and explainable for a complex and large dataset and proved that the performance model has generalization and explainability.

Chapter 6 concludes the thesis with a general discussion of the results obtained in the previous chapters, as well as discussing ideas for further work.

Chapter 2

Literature review

In this chapter, a survey is presented on research regarding deep learning, distributed deep learning, performance modeling of deep learning, differential evolution, and regularization techniques.

2.1 Deep Neural Networks

Deep learning refers to a subset of machine learning techniques that revolve around training artificial neural networks with multiple layers [1]. In traditional machine learning, algorithms typically work with shallow architectures that have only a few layers. In contrast, deep learning involves the use of neural networks with three or more layers, allowing for a greater capacity to learn complex representations and patterns in data.

Artificial Neural Networks (ANNs) [20], also known as neural networks, are interconnected networks of artificial neurons that process and learn from input data, inspired by the structure and function of the human brain. It is comprised of interconnected artificial neurons organized in layers, with weights and activation functions determining information flow and computations. Neural networks are trained to acquire knowledge from data, allowing them to generate predictions or make decisions by recognizing learned patterns and relationships.

Deep learning encompasses both supervised and unsupervised learning techniques [21] and finds applications in various tasks, including pattern analysis in data and classification. Since industrial applications of deep learning started around 2010, deep learning has

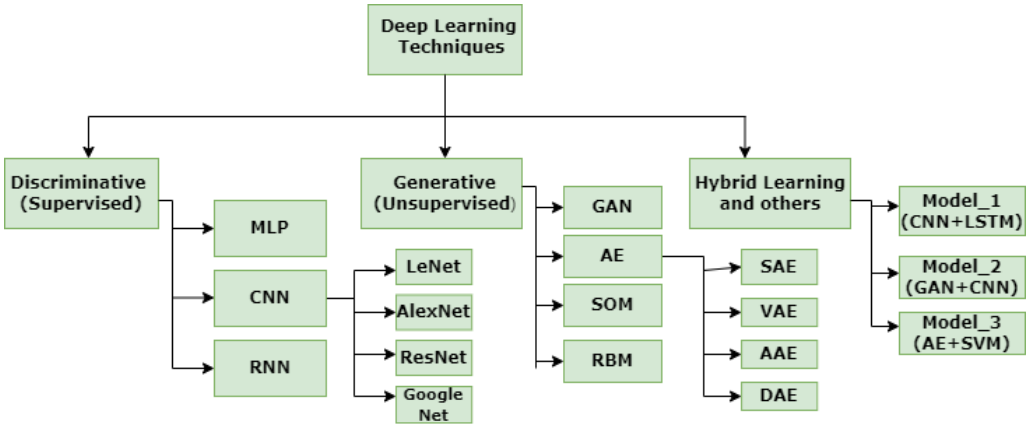


Figure 2.1. Classification of deep learning techniques [26]

Figure Legend

Here, MLP-Multilayer Perceptron, CNN-Convolution Neural Network, RNN-Recurrent Neural Network, LeNet-Simple Convolutional Neural Network, AlexNet-Name of a Convolutional Neural Network, ResNet- Residual Network, GoogleNet- Deep Covolutional Neural Network, GAN- Generative Advarsarial Networks, AE- Autoencoder, SOM- Self-Organising Map, RBM- Restricted Boltzman Machine, SAE- Staked Autoencoder, VAE- Variational Autoencoder, AAE- Adversarial Autoencoder, DAE- Denoising Autoencoders, LSTM- Long Short Term Memory, SVM- Support Vector Machine.

become an increasingly significant field of research within the machine learning community. Over the past few years, numerous deep learning models have been developed and explored. The deep learning techniques are classified into three types such as Discriminative (supervised), Generative (unsupervised), Hybrid learning and others. The classification of the deep learning methods are shown in the Figure 2.1. The deep learning models used in this research are Multilayer Perceptron (MLP) [22], Convolutional Neural Network (CNN) [23] / ResNet [24], and Autoencoder (AE) [25].

2.1.1 Multilayer Perceptron

A feedforward neural network, also known as a Multilayer Perceptron (MLP) [22], is a type of artificial neural network. It operates by allowing information to flow unidirectionally, progressing from the initial layer to the final layer without any loops or cycles. The MLP is commonly used as a supervised learning algorithm and is applied to tasks such as classification and regression [27]. The MLP architecture, as shown in the Figure 2.2, may be broken down into three primary parts:

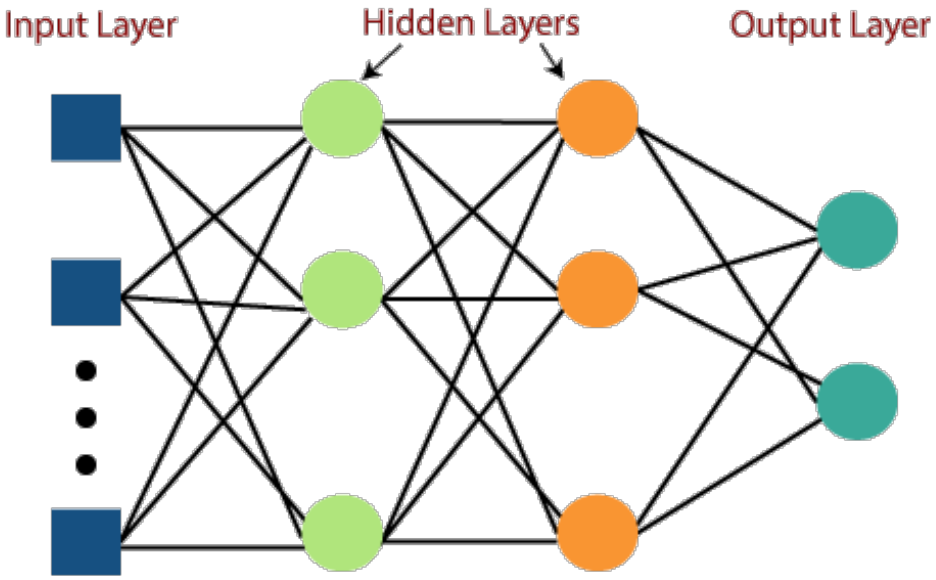


Figure 2.2. Multilayer Perceptron

1. Input Layer: The input layer acts as the initial component of the MLP, responsible for receiving the input data and transmitting it to the subsequent layer. The dimensionality of the input data determines the number of neurons in the input layer.

2. Hidden Layers: The intermediate layers of the MLP, known as hidden layers, play a significant role in processing input data and extracting meaningful features. Multiple neurons are present in each hidden layer, and they form connections with all the neurons in the subsequent layer.

3. Output Layer: The output layer represents the final layer of the MLP, responsible for

producing the conclusive output. In the case of classification tasks, the total number of unique classes determines the number of neurons in the output layer. In a regression task, it is determined by the dimensionality of the output data.

The neurons in each layer are connected using weighted connections, and each neuron performs a weighted sum of its inputs succeeded by a non-linear activation function. This function's role is to introduce non-linearity into the network, which enables it to learn complex non-linear relationships [28] in the input data. The weights of the MLP are learned using the backpropagation algorithm, which involves propagating the loss backwards through the neural network. This allows for the identification of each node's contribution to the loss and subsequent adjustment of the weights to minimize it. During backpropagation, the neural network's weights are adjusted based on the loss function gradient, iteratively optimizing performance by minimizing the loss. It's important to note that the optimization process doesn't directly assign higher or lower weights based on error rates; instead, it aims to find weight values that optimize the network's overall performance. The loss function evaluates the variance between the expected and actual outputs and is often minimized using optimization algorithms like Stochastic Gradient Descent (SGD) [29].

2.1.2 Convolutional Neural Networks

Convolutional Neural Networks (CNN) [30][31][32][33] are highly popular and widely employed as deep learning models for medical imaging, autonomous driving, bio-metric authentication and large-scale image classification and recognition. Convolutional networks consist of convolution, max-pooling, and fully-connected layers stacked on top of one another as shown in Figure 2.3.

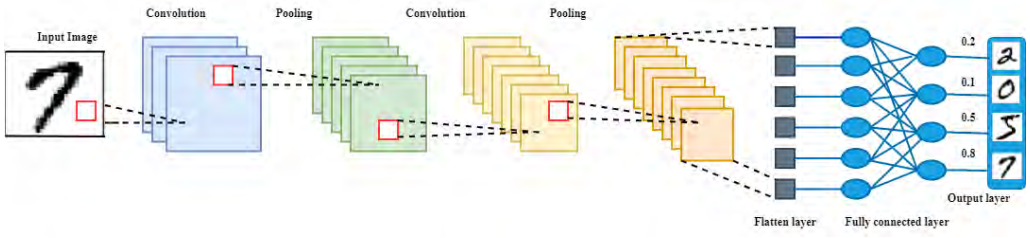


Figure 2.3. Architecture of LeNet-5 [34].

1. Convolution Layer: Convolutional Neural Networks (CNNs) employ a primary layer known as the convolutional layer that utilizes filters to create feature maps, indicating the presence of specific features in the input image. To incorporate nonlinearity and enhance the network's capacity for learning complex patterns, nonlinear activation functions like ReLU are commonly applied within the CNN architecture.
2. Pooling Layer: The pooling layers down sample the data, keeping feature maps to a reasonable size as the number of features increases and allowing succeeding convolution layers to view a broader spatial extent of the inputs.
3. Fully Connected Layer: The output is passed through fully connected layers that perform a linear transformation of feature maps, followed by another nonlinear activation function. The final fully connected layer utilizes the softmax function to generate a probability distribution over the possible classes.

The parameters of the network, including the filters and weights, are learned through back propagation. The CNN architecture facilitates the extraction of a hierarchy of representations from the input image. The lower layers of the network focus on detecting elementary features, while the higher layers progressively capture more intricate and sophisticated features and relationships.

ResNet

The ResNet (Residual Network) [24] is a specific architectural design for deep neural networks that effectively addresses the problem of vanishing gradients. The vanishing gradient problem occurs in very deep neural networks when the gradient signal diminishes significantly during backpropagation, leading to slow convergence. To overcome this issue, normalization techniques and residual connections are employed in the ResNet architecture. The Residual connections allow the network to skip over some layers during the forward pass, ensuring that the gradient signal has a direct path to propagate through the network. The ResNet architecture successfully addresses the vanishing gradient problem through the utilization of residual connections, as depicted in Figure 2.4.

The ResNet architecture is constructed by integrating a series of convolutional layers, pooling layers, batch normalization layers, and fully connected layers. The network re-

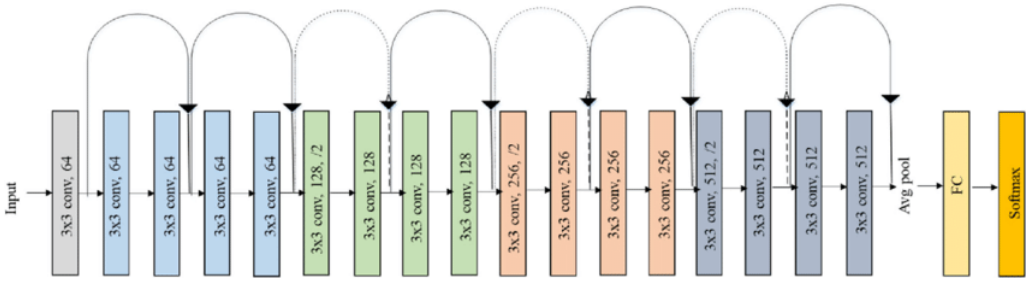


Figure 2.4. ResNet Architecture. Image source [35]

ceives an RGB image input with dimensions of 224x224 pixels. The network has the following layers:

1. Input Layer: The input layer is a 7x7 convolutional layer with stride 2 that takes the input image. Here, stride refers to the step size at which the convolution or pooling operation is applied to the input data.

2. Max Pooling Layer: Following the initial layer, a 3x3 max pooling layer with a stride of 2 is applied. This pooling layer downsamples the image, resulting in a reduction in size by a factor of 4.

3. Residual Blocks: Within the network, there are multiple residual blocks that are comprised of two or more convolutional layers and a connection mechanism. The inclusion of this connection mechanism, known as a shortcut connection, empowers the network to learn residual functions that facilitate the bypassing of specific layers during the training process.

4. Global Average Pooling Layer: Subsequent to the residual blocks, a global average pooling layer is introduced. This layer computes the average value over the spatial dimensions of the feature maps, resulting in a global representation of the input.

5. Fully Connected Layer: It is an extensively connected layer that employs the softmax activation function to generate the ultimate probabilities for each class.

ResNet is a family of convolutional neural network architectures, with notable variants including ResNet-50, ResNet-101, and ResNet-152. These variations are distinguished by their depths, consisting of 50, 101, and 152 layers, respectively. Pre-training on extensive

datasets such as ImageNet has been conducted on these architectures. ResNet models provide flexibility and can be customized or utilized as feature extractors for a wide range of computer vision applications, including object detection, image classification, and image segmentation.

For example, the ResNet architecture was used to train the performance model on the Alzheimer's dataset. While models like AlexNet [36], VGGNet [5], and GoogLeNet [3] have demonstrated success in image classification tasks, they have comparatively fewer layers than ResNet. Alzheimer's disease classification [37] might benefit from the increased depth and capacity to capture intricate patterns offered by ResNet's architecture. In addition, ResNet's skip connections and residual blocks enable it to learn residual features effectively, which is crucial when dealing with complex datasets. This can help in capturing and representing intricate patterns in brain images that may be important for Alzheimer's disease classification. ResNet has established itself as a leading performer across diverse computer vision benchmarks and challenges [38], indicating its effectiveness in capturing complex patterns and achieving high classification accuracy. This success might make it a preferred choice in the research community, especially when aiming for top performance.

2.1.3 Autoencoder

Autoencoders [39] are a class of unsupervised learning techniques that leverage neural networks to derive data representations. The architecture of the network comprises a compression layer that induces a compressed version of the input. Compression and subsequent reconstruction become difficult when input features are independent of each other. However, if there is any pattern or structure in the data, such as interdependence between input features, the network can learn and utilize this structure while passing the input through the compression layer.

The architecture of an autoencoder, as shown in the Figure 2.5, consists of two main stages: encoding and decoding. The encoder is the first part of the autoencoder that takes the input data and compresses it into a lower-dimensional representation. It consists of several layers of neural networks, each with a set of learnable parameters that convert

the input information into a latent representation. The final layer of the encoder is the bottleneck layer, which produces the latent representation or latent code.

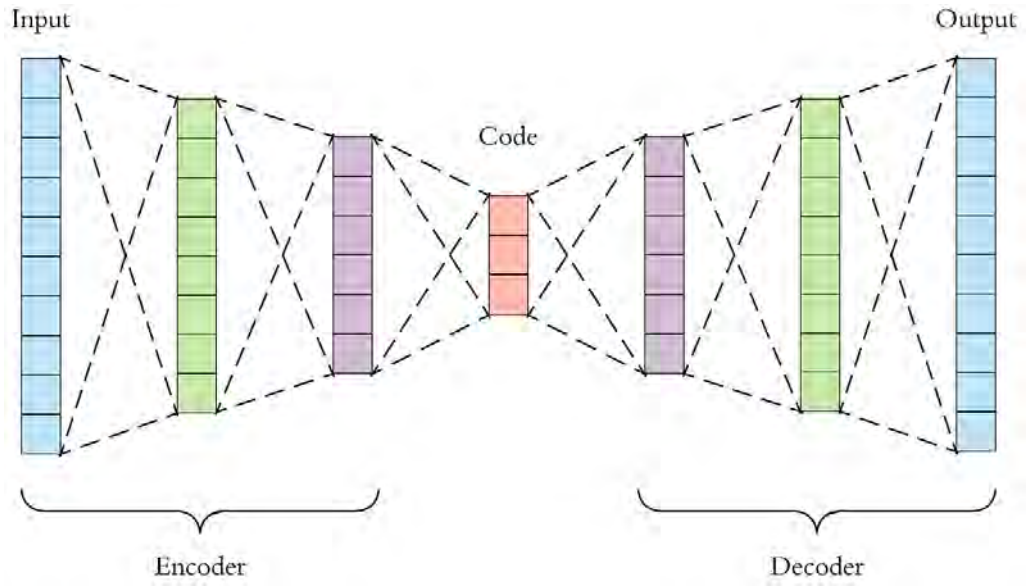


Figure 2.5. Autoencoders Architecture.

The decoder is the second part of the autoencoder that takes the latent representation and reconstructs the original input data. It consists of several layers of neural networks that take the latent representation and convert the latent representation into its initial input data format. The ultimate layer of the decoder produces the final output, which should match the input data as closely as possible.

The autoencoder undergoes unsupervised learning, where it is trained on input data to minimize the difference between the reconstructed output and the original input. Typically, to compute the reconstruction error, a loss function is applied such as Mean Squared Error (MSE) [40] or Binary Cross-Entropy (BCE) [41].

The architecture of an autoencoder can be modified to improve its performance or to solve specific tasks such as anomaly detection, data compression, and image denoising. Variants of autoencoders such as Convolutional Autoencoder (CAE) [42] and Recurrent Autoencoder (RAE) [43] can be used for processing image and sequential data. A Denoising Autoencoder (DAE) [44] can be used to remove noise from the input data, and a Variational Autoencoder (VAE) [45] can be used for generative modeling and data gener-

ation, respectively.

2.2 Parallel and Distributed Deep Learning

The field of distributed deep learning [46] has been a significant area of research for many years with wide variety of real-time and practical applications in every sector. Typically, deep learning applications have to deal with big data, parameter storage and computation power.

- Big Data: In industry, the ImageNet dataset [47] is a prime example of large-scale data, typically consists of 155 terabytes of data. For example, Instagram users upload 4.7 billion images per day, and deep learning models are used to analyze and process these images to gain insights into user preferences, interests, and behaviors.
- Parameters Storage: Deep learning models are built with numerous layers which contains a few hundreds to 2 billion parameters. It needs 0.1-8GiB merely to store the model (which is stored in memory rather than on the hard drive when in use). In this case, the model's training phase typically uses more memory, this poses a great challenge in terms of model fitting on a single conventional PC.
- Computation Power: Recognizing and categorizing objects in high-resolution images using a deep neural network requires a substantial amount of computational resources. For instance, the ImageNet dataset is a prevalent standard for image recognition, and it encompasses 14,197,122 images with a 224x224 pixel resolution. Based on factors such as the intricacy of the model and the accessible computing infrastructure, training a deep learning model on this dataset could take up to several weeks or months. This poses a significant challenge to organizations or researchers with restricted access to high-performance computing resources.

To conclude, the complex nature of deep learning requires high computational power, efficient data processing and storage, and distributed infrastructure for training large models with significant amounts of data. Due to the extensive nature of this process, parallelization is necessary to distribute the workload across multiple compute nodes equipped with multiple GPUs. To ensure the effectiveness of parallelization, advanced distributed

optimization strategies must be implemented, leveraging parallelism approaches to harness the full potential of computational power. Parallelization and distributed training are vital for efficiently training large-scale deep learning models by dividing the workload across multiple computing resources, enabling faster processing and scalability to handle extensive datasets. Data parallelism, model parallelism, pipeline parallelism, and hybrid parallelism are the different parallelization methods used to achieve parallelization in deep learning tasks.

2.2.1 Data Parallelism

Data parallelism [48] refers to dividing the training data into non-overlapping subsets and distributing them across multiple machines. Each machine performs computations locally by utilizing a complete model, but communication is necessary between the computing nodes to merge the gradients and update the model weights. Advantage of data parallelism is speedup, better cost per performance in the long run and applicable to any DL model architecture. To expedite deep learning, researchers usually utilize multiple graphics processing units (GPUs) allocated per computing node for reducing training time. Deep learning using data-parallel processing can be categorized as either synchronous or asynchronous based on the timing of the aggregation process.

In the synchronous data parallelism approach, once the gradients are computed on each GPUs, they are combined through averaging or summation, followed by an aggregation operation to adjust the training parameters. For the exchange of data required for accumulation within GPUs, Distributed TensorFlow [49] is a commonly adopted distributed deep learning framework. It utilizes the parameter-server approach to collect the computed gradients from each GPU's memory or the CPU's memory as depicted in Figure 2.6 (a). This method ensures synchronous sharing of the same parameters among all GPUs, thereby maintaining training accuracy. However, using this scheme in a heterogeneous computing system where some GPUs may have lower processing performance than others can lead to degraded system performance.

The Asynchronous approach [50] offers a viable solution to the aforementioned problem by allowing the GPU to bypass synchronization during each iteration, thus avoiding the need to wait for slower GPUs. However, this approach may lead to reduced training

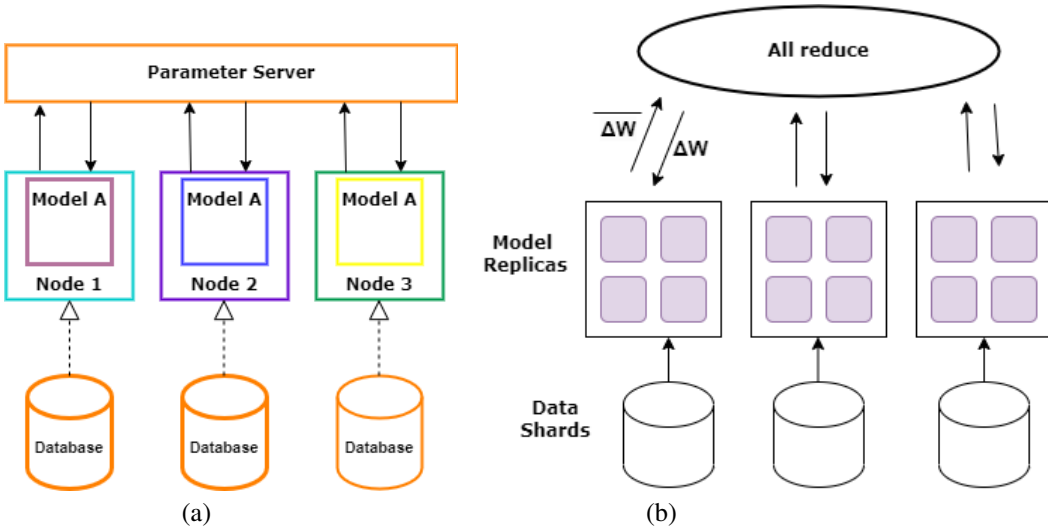


Figure 2.6. Parallel computing approaches (a) Parameter server approach, and (b) All reduce architecture

accuracy as each GPU can use different parameters during each iteration. Furthermore, this method has the potential to create network congestion by considering I/O activities on individual devices, including parameter servers responsible for performing aggregations.

The All-reduce scheme, unlike the parameter-server approach [51], enables direct parameter and gradient exchange among GPUs for aggregation as illustrated in Figure 2.6 (b). It relies on distributed I/O across devices, allowing decentralized communication with both synchronous and asynchronous methods. Synchronous All-reduce ensures consistency by waiting for completion, while asynchronous allows independent updates, reducing synchronization overhead but risking inconsistency. All-reduce adoption mitigates potential parameter-server bottlenecks, providing communication flexibility for distributed training.

The decentralized architecture [52], works without parameter server. Instead, the GPUs establish direct communication to exchange parameter updates by means of an all-reduce operation. The topology of the workers plays a critical in this operation. In a fully connected network, the communication overhead can be significant since the gradients are communicated to all other workers. To address this issue, a commonly utilized solution is to implement a ring topology known as ring-all reduce. Horovod [52] is a distributed framework utilized for the training of deep learning models in popular deep learning frameworks such as TensorFlow, Keras, PyTorch, and Apache MXNet. The primary objective of

Horovod is to optimize the efficiency of distributed deep learning by utilizing inter-GPU communication via ring reduction. The framework's proficiency lies in its ability to handle extensive datasets while retaining shorter training times, achieved through standardized distributed TensorFlow techniques.

Baidu [53] presented their implementation of ring all-reduce and demonstrated a draft implementation of it by creating a fork of TensorFlow. Uber also uses the ring-all reduce algorithm to improve inter GPU communication through NCCL (NVIDIA Collective Communications Library) which allows for faster and more efficient distributed training in TensorFlow. In contrast to the asynchronous methodology, which updates weights independently without synchronization among workers, ring-all reduce performs parallel stochastic gradient descent (SGD) [54] [55] [56] [57]. To assess the effectiveness of data parallelism on multi-GPU nodes, the widely recognized Alexnet architecture [58] was employed, resulting in a speedup of approximately 2.2x with 4 GPUs compared to a single GPU.

2.2.2 Model Parallelism

Model parallelism [59] is another common parallelization strategy. This approach involves dividing the model among multiple GPUs, with each worker responsible for computing a subset of layers. By doing so, parameter synchronization between workers is eliminated, but data transfers are necessary between adjacent layers assigned to different workers. Model parallelism is particularly beneficial when dealing with models that are too large to be accommodated on a single GPU or Tensor Processing Unit (TPU). However, if the number of nodes is too high, communication overhead can significantly reduce network performance.

Project adam [60] introduced a deep learning training system that supports model parallelism, enabling efficient and scalable training processes. The system places significant emphasis on optimizing computation and communication to enhance overall efficiency and scalability. To achieve this, *adam* utilized a parameter server architecture, Within high traffic execution paths, the system adopts lock-free data structures for queues and hash tables to ensure efficient operations. The purpose of this is to accelerate network processing, update, and storage I/O operations. Model training on a machine involves multi-threaded

training, enabling the acceleration of training processes by accessing and updating shared model weights locally without the use of locks. This optimization approach is similar to the *hog-wild* system [61]. *Hog-wild* is a strategy for distributed computing utilized in the training of deep learning models. The combination of centralized and distributed methods in the DistBelief framework creates a hybrid parallel system that effectively capitalizes on their respective advantages. Through the strategic use of hybrid parallelism, DistBelief achieves improved scalability and accelerated deep network training, optimizing resource utilization and enhancing performance on large-scale machine learning tasks. This integration enables efficient processing and coordination across multiple devices, contributing to the framework's efficacy in handling complex and computationally intensive tasks.

2.2.3 Pipeline Parallelism

Pipeline parallelism is an approach that integrates the concepts of data parallelism and model parallelism. A model is divided into layers, and each node is assigned a specific layer to work on. Additionally, the data is split into smaller subsets, which are then propagated through the pipeline to subsequent workers for processing. Gpipe [62] is a pipeline parallelism library designed to enable the scaling of any network expressed as a sequence of layers. This library offers the necessary adaptability to effectively scale networks to extremely large sizes. Gpipe achieves this through the utilization of a batch splitting pipelining algorithm, resulting in nearly linear speedup when distributing a model across multiple accelerators. An important advantage of Gpipe is its ability to train expansive neural networks for various tasks with distinct network architectures. For instance, Gpipe successfully trained an Amoeba Net model with 557 million parameters for Image Classification, achieving an impressive top-1 accuracy of 84.4% on the ImageNet-2012 dataset. Furthermore, in Multilingual Neural Machine Translation (MNMT), Gpipe trained a single Transformer model with 6 billion parameters and 128 layers using an extensive multilingual dataset encompassing over 100 languages. In terms of speed, Gpipe outperformed all other NLP models. The successful application of Gpipe demonstrates its exceptional capability in facilitating the training of massive neural networks across a diverse array of domains.

2.2.4 Hybrid Parallelism

Hybrid parallelism refers to when the DL model are complex and composed of many different layers then it mixes data, model and pipeline parallelism depending on the complexity of a model. The DistBelief framework [63] is a hybrid parallel system that incorporates two distinct optimization techniques: Downpour SGD and Sandblaster, for online and batch optimizations. Downpour SGD adopts a centralized parameter server and employs an asynchronous stochastic gradient descent method, ensuring efficient synchronization and updates. In contrast, Sandblaster takes a different approach by utilizing distributed batch optimization procedures in conjunction with a decentralized implementation of the L-BFGS (Limited-Broyden–Fletcher–Goldfarb–Shanno) algorithm [64]. The combination of these approaches creates a hybrid parallel system within the DistBelief framework, utilizing both centralized and distributed methods. By incorporating hybrid parallelism, DistBelief achieves enhanced scalability and accelerated deep network training, leading to efficient resource utilization and improved performance on large-scale machine learning tasks.

2.3 Deep Learning Frameworks

Deep learning (DL) frameworks provide a high-level programming interface for constructing, training, and evaluating deep neural networks. In this thesis, the following deep learning frameworks have been utilized: TensorFlow [65], MXNet [66], Chainer [67], and PyTorch [68], to assess the performance of the models.

2.3.1 TensorFlow

Google introduced TensorFlow in November 2015 as a platform for building and constructing DL implementations. TensorFlow [65] is an opensource, scalable and versatile software library for numeric and conventional mathematical computations using dataflow graphs [69]. TensorFlow is capable of using many threads, enabling multi-core processors to be utilised effectively. Moreover, it contains GPU implementations that use NVIDIA

CUDA-based Deep Neural Network (cuDNN) [70], enabling the effective use of one (or more) GPUs on a single node.

Matrix multiplication serves as a fundamental operation upon which neural networks and other machine learning models heavily rely. Its computational simplicity and inherent suitability for parallelization make it an efficient choice for various computational tasks in the field of machine learning. Alongside these computational benefits, distributed training plays a crucial role in training deep learning models across multiple machines. By harnessing the power of distributed training, it becomes possible to achieve faster training times and train larger models that go beyond the memory capacity restrictions of a single machine.

Moreover, the TensorFlow API provides the *tf.distribute.strategy*, which offers a convenient solution for distributing training tasks and enables parallelization across multiple GPUs. This approach significantly accelerates the training process while effectively utilizing the combined memory resources of multiple GPUs, allowing for efficient utilization of larger models. The *TensorFlow.distribute.strategy API* encompasses a range of distribution strategies to cater to different requirements. For instance, the *tf.distribute.MirroredStrategy* creates replicas on each GPU, ensuring that all variables are mirrored across all replicas. In contrast, the *tf.distribute.experimental.CentralStorageStrategy* places all variables on the CPU while duplicating operations across all GPUs. Additionally, the *tf.distribute.experimental.ParameterServerStrategy* designates machines as workers and parameter servers, effectively distributing the workload accordingly.

2.3.2 MxNet

Apache introduced MXNet [66] in November 2015 as a platform for training and deploying DL implementations. It's a combination of declarative and imperative programming styles [71], and it can derive gradients by using auto differentiation. MXnet is optimized for memory and compute efficiency and runs on distinct heterogeneous systems like mobile devices to distributed GPU clusters. MXNet facilitates distributed training, enabling the utilization of multiple devices to accelerate model training. MXNet supports data, model and hybrid parallelism.

For distributed training, MXNet provides several APIs. Gluon API is a high-level interface for building neural networks. The gluon.data.DataLoader class provides a fast and efficient method for loading and distributing data across several devices using a communication library such as MPI [72] or NCCL [73]. And also MXNet provides a kvstore module [74] that enables easy synchronization of the parameters across the devices during training.

The MXNet framework works efficiently with Parameter Server (PS) and all-reduce architectures [75] for gradient synchronization, utilizing TCP (Transmission Control Protocol) or RDMA (Remote Direct Memory Access) [76], reduced job training time and improved resource efficiency in resource performance modeling [77]. Additionally, the developers of MXNet have created an open-source framework, called SOCKEYE [78], which outperformed seven current NMT (neural machine translation toolkit) [78] based on four deep learning backends on WMT (Workshop on Statistical Machine Translation) [79] tasks with minimal setup or hyperparameter tuning.

2.3.3 Chainer

Chainer, an open-source deep learning framework, was developed at Preferred Networks, Inc., and released to the public in 2015 [67]. Chainer provides several methods to perform distributed training, together with data parallelism, model parallelism, and the parameter server approach. Chainermn, a module within Chainer, enables both data parallelism and model parallelism. Through the utilization of data parallelism, the input data is partitioned across numerous devices or nodes, enabling each individual device or node to process a specific portion of the data. Gradients are accumulated and synchronized across the devices or nodes to update the model parameters effectively. Alternatively, model parallelism entails the distribution of model parameters among devices or nodes, where each device or node undertakes computations for a specific section of the model. Chainermn provides convenient APIs to facilitate the implementation of both data parallelism and model parallelism.

Moreover, Chainer is equipped with the capability to support the parameter server approach through its integration with the chainercv module. In this approach, the parameters of the model are centrally stored on a dedicated server. Each worker node participating in

the distributed training retrieves the necessary parameters from the server, performs gradient computations, and updates the parameters accordingly. The parameter server approach offers notable advantages, particularly for models characterized by a substantial number of parameters. By efficiently distributing the parameter updates and computations, Chainer facilitates efficient training across distributed environments.

To perform distributed training in Chainer, you need to set up a distributed environment with multiple nodes or GPUs. You can then use the APIs provided by the chainermn or chainercv module to perform data parallelism, model parallelism, or parameter server training.

2.3.4 Pytorch

PyTorch [80] is a deep learning framework known for its dynamic computational graph and smooth integration with Python. It offers developers the flexibility to define and modify computational graphs in real-time, enhancing the ease of model development and debugging. At the heart of PyTorch lies the *torch.Tensor* class, which serves as the foundation for constructing neural networks and represents multi-dimensional arrays. Through the utilization of the *torch.nn module*, developers can easily create intricate network architectures by employing predefined layers and activation functions.

The automatic differentiation [81] engine of *PyTorch, torch.autograd*, is responsible for computing gradients automatically. This functionality simplifies model training by enabling techniques like backpropagation. Additionally, PyTorch seamlessly integrates with CUDA for GPU acceleration, resulting in faster training and inference for large-scale models. The framework excels in parallelization, providing support for data parallelism, model parallelism, and distributed training. This empowers developers to take advantage of multiple GPUs or machines, facilitating parallel data processing and the distribution of model components. As a consequence, PyTorch significantly enhances performance and scalability.

PyTorch's distributed training capabilities offer significant advantages when it comes to scaling up deep learning tasks. The *torch.nn.parallel.DistributedDataParallel* (DDP) module simplifies distributed training by incorporating features such as data partitioning,

gradient synchronization, and communication among nodes in the distributed system. This streamlined approach allows developers to efficiently train models across multiple machines, effectively utilizing computational resources and managing larger datasets. With PyTorch's robust support for distributed training, models can be trained faster and scaled to handle complex tasks through the utilization of both data and model parallelism.

2.4 Performance Modelling in Deep Learning

Performance modeling [7] is a fundamental technique used to create an abstract representation or model of a system, enabling an in-depth understanding and prediction of its behavior and performance characteristics. By carefully capturing crucial aspects of the system, including its structure, components, and workload patterns, performance modeling facilitates the simulation and analysis of system performance under diverse conditions. It serves as a critical tool for making informed decisions pertaining to system design, optimization strategies, and resource allocation. Through the use of mathematical or computational models, researchers and analysts can explore a multitude of scenarios, accurately predict performance metrics, and effectively assess the impact of changes or enhancements. Performance modeling encompasses various forms, such as analytical models, simulations, queuing models, and statistical models, accommodating different system types and complexities. Leveraging performance modeling techniques, it becomes possible to assess system scalability, identify performance bottlenecks, and guide optimization efforts, leading to significant improvements in overall system performance and efficient resource utilization.

Performance modelling involves prediction – estimating the performance of a new system, the impact of change on an existing system, or the impact of a change in workload on an existing system [82]. Existing performance modelling of deep learning frameworks can be generally segmented into two categories:

1. Analytical Modelling (AM).
2. Empirical Modelling (EM).

2.4.1 Analytical Modelling of Deep Learning

This subsection provides the existing works developed in a distributed environment using analytical modelling. Analytical modelling uses a transparent approach to convert a model's or an application's internal mechanism into a mathematical model corresponding to the system's goals, which can significantly expedite the creation of a performance model for the intended system. The existing analytical modelling works investigated deep learning performance modelling and scaling optimisation in distributed environment [10], asynchronous GPU processing based on mini-batch SGD [83], efficient GPU utilisation in deep learning [84], comprehensive analysis and comparison of the performance of deep learning frameworks running on GPUs [85] [86].

Yan *et al.* [10] developed performance model to evaluate the impact of partitioning and resourcing decisions on the overall performance and scalability of distributed system architectures' using a DL framework Adam [60]. In addition, the performance model was also used to guide the development of a scalability optimizer that quickly selects the optimal system configuration for reducing DNN training time. However, the model can only be applied to specific DL systems, particularly when it has parameter servers and synchronous weights between worker nodes dynamically.

Heehoon Kim *et al.* [11] evaluated five popular deep learning frameworks TensorFlow [65], CNTK [87], Theano [88], Caffe-MPI [89] and Torch [90] in terms of their performance in both single and multi-GPU environments. In this work, each framework incorporated and compared different convolution algorithms, such as Winograd, General Matrix Multiplication (GEMM), Fast Fourier Transformation (FFT), and direct convolution algorithms, in terms of layered-wise analysis and execution time. The results have shown that FFT and Winograd algorithms surpass the GEMM and other convolution algorithms. However, the convolution algorithms used by the frameworks provided poor explainability regarding their internal operations.

Qi *et al.* [12] proposed an analytical performance model named Paleo, predicting the deep neural network performance by considering communication schemes, network architecture and parallelization strategies. The results demonstrated that hybrid parallelism performed much better than data parallelism while training the Alexnet model. However,

the model did not consider other factors affecting the overall performance of a model, such as memory usage, data transfer, or communication overhead in distributed environments.

Castello *et al.* [13] developed an analytical model to evaluate the scalability of data parallelism and model parallelism for distributed deep learning training of deep convolutional neural networks (CNNs). The analysis considers various factors, including batch size, computational performance of processing units, memory bandwidth of processing units, network link bandwidth, and cluster dimension. The analysis utilizes analytical performance models that can simulate both the CNN model's organization and the distributed platform's hardware configuration. How ever, the model does not discuss the practical implications of the findings, such as how to optimize the performance of a real-world system based on the model predictions.

Jia *et al.* [14] focused on analyzing the impact of various factors such as network topology, communication patterns, and batch sizes on the performance and consistency of distributed deep learning applications. The exploration of different network topologies, including ring, mesh, and star, revealed varying impacts on the performance and consistency of distributed deep learning applications. Specifically, the ring topology demonstrated the best performance, followed by the mesh and star topologies. Additionally, the analysis of various communication patterns, such as all-to-all, random, and broadcast, indicated that the all-to-all communication pattern exhibited the best performance, followed by random and broadcast patterns. Moreover, employing larger batch sizes could effectively enhance the performance of decentralized deep learning systems by mitigating communication overhead. They found that using techniques such as Batch-based Consistency Control (BCC) and Model Parallelism with Weight Stashing (MP-WS) improved consistency in the training process.However, the paper only considers synchronous training and does not explore the performance or consistency of asynchronous training methods.

Sean Mahon *et al.* [91] provided a comprehensive analysis of the factors that influence the performance of distributed and scalable deep learning, including the size of the model and dataset, communication overhead, network bandwidth, and hardware configuration. They presented two communication methods: parameter server and all-reduce. Comparing the performance of these approaches on different deep learning models, they found that the choice of communication method can significantly impact performance. In some

cases, the parameter server approach exhibits better performance with a large number of unreliable and less powerful machines, while the AllReduce method works better with a small number of fast devices in a controlled environment with strong connected links.

Shi *et al.* [86] developed a performance model to evaluate the performance of various distributed deep learning frameworks (TensorFlow, CNTK, MXnet and Caffe) with deep convolutional neural networks (Alexnet, ResNet and GoogleNet models) on the multi-GPU environment. They measured training time, memory usage, and GPU utilization and compared the frameworks in terms of training time and resource utilization. However, they did not provide a breakdown of the time to divide the mini-batch into smaller batches or measure the load imbalance factor, which are critical factors that could significantly affect the training efficiency and performance in a parallel computing environment. Kavarakuntla *et al.* [92] extended the Shi analytical performance model to evaluate the runtime performance of deep learning frameworks (TensorFlow, MXnet and Chainer) with the CNN, MLP and AN models running in the multi-GPU environment. The extended model considered the load imbalance factor and made a layer-wise analysis of a neural network, providing a more comprehensive evaluation of the frameworks' performance. The experimental results showed that the load balance is an influential factor affecting the system performance.

2.4.2 Empirical Modelling of Deep Learning

Empirical modeling presents a strong alternative to analytical models, offering advantages such as greater flexibility, realism, accuracy, and the ability to derive data-driven insights for complex real-world scenarios. Empirical modelling builds models through observation and experimentation, which is antithetic to analytical modelling. In this approach, empirical modelling predicts the outcome of an unknown set of system parameters based on observation and experimentation. It characterises an algorithm's performance across problem instances and/or parameter configurations based on sample data. Empirical models predict the output of a new configuration on the target machine. By using the empirical modelling approach, the existing works investigated new collective communication techniques in distributed environment.

Oyama *et al.* [83] proposed a performance model for asynchronous stochastic gradient

descent-based deep learning systems on GPU-based supercomputers. The model accurately predicted time to sweep the dataset, mini-batch size, and staleness with average errors of 5%, 9%, and 19%, respectively, for various CNN architectures on different GPU-based supercomputers. It utilized small empirical models and provided precise probability distributions of crucial performance metrics. However, the study's limitations include a focus solely on weight synchronization among GPUs and data-parallelism, neglecting other parallelization strategies such as parameter servers or model-parallel approaches. Additionally, the study did not explore potential communication overhead and network latency challenges, which could significantly impact the performance of distributed deep learning systems.

Yufei *et al.* [16] established a performance model for estimating resource consumption and performance efficiency of field-programmable gate array (FPGA) based accelerators for CNN inference. The model was applied to the design phase to find and explore optimal design options for these accelerators. The model focused on several key performance metrics, including Dynamic Random Access Memory (DRAM) efficiency, response time, and PE utilization. The evaluation results showed that the model's predictions closely matched the actual test results obtained on FPGAs for CNN inference, with predictions typically within a factor of three of the actual results. However, the model has poor explainability, that did not discuss the impact of the proposed methodology on the overall design process, including design time and design complexity. The methodology may introduce additional complexity or design constraints that could limit the potential benefits of FPGA-based acceleration for CNN inference.

Z.Lin *et al.* [17] proposed a performance prediction model for distributed deep learning on GPU clusters that considering both the network topology and communication patterns of the trained deep learning model. The communication and computation times for each layer in a deep neural network were included in the model. The model was evaluated on several deep learning benchmarks and showed that it achieved higher accuracy in predicting training time than existing models. The model can also be used to optimize the performance of distributed deep learning by finding the optimal configuration of GPU nodes and reducing the training time. However, the assessment of the proposed model was confined to three different GPU clusters, potentially limiting its generalizability to other GPU clusters or distributed DL architectures.

Andre Viebke *et al.* [18] developed an empirical performance model for predicting the execution time of deep learning models, specifically CNNs and RNNs, on Intel's Many Integrated Core (IMIC) architectures. The model considered various factors such as the number of cores, memory bandwidth, and communication overhead, which are unique features of the IMIC architecture. The proposed performance model was evaluated by comparing its predictions with experimental data. The experimental data was obtained from running several deep learning models on Intel's Xeon Phi and Knights Landing platforms [93]. The results demonstrated that the proposed model achieved high prediction accuracy, with an average error of less than 5% for both training and inference phases. However, the model's generalizability is limited and there is a lack of explanation for its high accuracy.

Rakshith *et al.* [19] presented an empirical study of the performance of Horovod, a distributed deep learning framework, for image classification tasks. They evaluated the performance of Horovod on two popular image datasets, CIFAR-10 and ImageNet, using a cluster of machines with varying numbers of GPUs. They also compared the performance of Horovod to other distributed deep learning frameworks, such as TensorFlow and PyTorch, and found that Horovod achieved better performance in certain scenarios. They provided recommendations for optimizing the performance of Horovod on large-scale image datasets, such as using efficient data loading and preprocessing techniques, and optimizing the communication and synchronization between the machines. However, the experimental configuration utilized in the research does not accurately reflect real-world situations in which the underlying hardware and network setups may differ substantially.

Most recently, a new approach named the hybrid model has been proposed [94] by combining the elements of analytical modeling and empirical modeling for better performance prediction developed in other fields. Inspired by this idea, a model is proposed that gives insights into the intrinsic parameters' performance and scalability of the extrinsic parameters.

2.4.3 Conclusions from Previous Studies and Introduction of Differential Evolution

In summary, the existing literature on performance modeling of deep learning frameworks in distributed environments was explored. The studies presented a diverse range of ap-

proaches, including analytical modeling and empirical modeling, to predict and understand the performance characteristics of distributed deep learning systems. These works have significantly contributed to the understanding of the system behavior, scalability, communication patterns, and resource utilization in various distributed settings. However, the existing studies do have certain limitations. For instance, some of the analytical models are limited to specific distributed deep learning systems, and their generalizability to other architectures or GPU clusters may be constrained. Additionally, empirical models may not provide sufficient explainability for some of their predictions, limiting insights into the internal mechanisms of deep learning models.

To address these limitations and to enhance our understanding of distributed deep neural networks, proposed a generic performance model of an application in a distributed environment with a generic expression of the application execution time that considers the influence of both intrinsic factors/operations (e.g., algorithmic parameters/internal operations) and extrinsic scaling factors (e.g., data chunks or batch size). Formulating it as a global optimization problem with regularization, we solve it using the differential evolution algorithm—a robust optimization technique. This approach aims to find the best-fit values of the constants in the generic expression, matching the experimentally determined computation time.

2.5 Differential Evolution

Differential Evolution (DE) [95] is an evolutionary optimization algorithm widely used in various fields for solving optimization problems such as motor fault diagnosis [96], structure prediction of materials [97], automatic clustering techniques [98], community detection [99], learning applications [100] and so on. It was introduced by Storn and Price in 1997. The algorithm functions on a group of potential solutions referred to as individuals or vectors, which form a population. Each individual represents a potential optimization solution [101], and the algorithm strives to progressively refine these solutions through iterative steps. The procedure of the Differential Evolution algorithm as follows:

1. Initialization: A population of individuals is randomly generated, with each individ-

ual having a set of parameter values that represent a potential solution.

2. Mutation: DE introduces diversity into the population by generating new candidate solutions through mutation. For each individual in the population, a mutation operation is performed by combining the parameter values of three randomly selected individuals (known as target, donor, and base vectors) using a mutation factor. This generates a new trial vector.

3. Crossover: The trial vector undergoes the crossover operation, where its parameters are merged with those of the target vector. The determination of parameter inheritance from either vector is governed by a crossover probability [102].

4. Selection: The trial vector undergoes assessment using an objective function to quantify its fitness or quality. If the trial vector demonstrates superior performance compared to the target vector, it displaces the target vector in the subsequent generation. Conversely, if the trial vector fails to outperform, the target vector retains its place within the population.

5. Termination: The algorithm continues to iterate through the mutation, crossover, and selection steps until a specific termination condition is satisfied [101]. This condition may involve reaching a predetermined fitness level, achieving convergence within the population, or exceeding a maximum number of iterations.

Differential Evolution is known for its simplicity, efficiency, and ability to handle optimization problems with non-linear, non-differentiable, and noisy objective functions. It is similar to other evolutionary algorithms such as Genetic Algorithm (GA) [103] by applying mutation, crossover, and selection operators to determine the population toward better solutions. In contrast to the Genetic Algorithm (GA), the Differential Evolution (DE) algorithm imparts mutation to each individual while transferring them to the next generation. In the mutation procedure of DE, for each solution, three more individuals are picked from the population, and as a consequence, a mutated individual is produced. On the other hand, GAs typically use mutation, crossover, and selection operators on pairs of individuals to generate new offspring for the next generation. This means that in GAs, only two individuals are involved in producing a new offspring, while in DE, three individuals play a role in the mutation process. It is determined based on the fitness value whether or

not the first individual selected will be replaced. In differential evolution, the crossover is not the primary operation, as it is in the genetic algorithm. In recent times, several works have been proposed to use DE for neural network optimization [104] [105] [106].

2.6 Regularization

Regularization techniques are utilized in machine learning to address the issue of overfitting [107]. By preventing models from fitting the training data too closely, regularization helps improve their generalization capability. Regularization achieves this by imposing additional constraints during training, promoting simplicity and reducing model complexity [108]. It discourages the over-reliance on individual features [109] and handles multicollinearity, a situation where features are highly correlated [110]. By tuning a regularization parameter, such as λ, the trade-off between fitting the training data and model complexity can be optimized [111]. For L1 regularization, loss can be calulated as,

$$Loss = DataLoss + \lambda * \sum |coefficient| \tag{2.1}$$

here, λ (lambda) is the regularization parameter that controls the strength of the penalty term. The higher the value of λ, the stronger the regularization, leading to a sparser model with more coefficients close to zero. The absolute value function ensures that the penalty is proportional to the magnitude of the coefficients, promoting sparsity in the model. For L2 regularization, loss can be calculated as,

$$Loss = DataLoss + \lambda * \sum \left|coefficient^2\right| \tag{2.2}$$

here, λ (lambda) is the regularization parameter controlling the penalty term's strength. Higher λ leads to stronger regularization, resulting in smaller coefficients in the model. The squared term in L2 regularization ensures a balanced shrinkage of coefficients, preventing any single coefficient from dominating the model's prediction.

Regularization techniques thus play a crucial role in enhancing the robustness, interpretability, and performance of machine learning models by preventing overfitting and improving generalization to unseen data [112].

There are two main regularization techniques that have emerged as pivotal tools for enhancing performance models by mitigating overfitting issues and improving generalization [113]. The first technique is L1 regularization, also known as Lasso regularization. It is widely used to induce sparsity in model coefficients by adding a penalty term to the loss function [114]. This encourages less influential features to shrink towards zero, reducing variance and enhancing model interpretability. L1 regularization has demonstrated its effectiveness in various domains, such as image recognition [115], natural language processing [116], and financial forecasting [117]. The second technique is L2 regularization, commonly referred to as Ridge regularization. It is prominent for mitigating excessive reliance on individual features. By incorporating a squared penalty term into the loss function [118], L2 regularization maintains small yet non-zero coefficients for all features, leading to more robust and less sensitive performance models [119].

The selection of the weight parameter, which governs the regularization strength, assumes paramount importance in regularization techniques. Finding the optimal λ value is crucial in striking the right balance between the model's alignment with the training data and its level of complexity. Common methodologies, such as cross-validation [120] and grid search [121], are employed to identify the lambda value that maximizes model generalization. These techniques involve the systematic evaluation of model performance for various λ values, enabling the selection of an optimal value that facilitates superior generalization capability.

In summary, regularization techniques, including L1 regularization (Lasso) and L2 regularization (Ridge), offer robust mechanisms for reducing variance, enhancing model interpretability, and optimizing the lambda parameter in performance modeling tasks. The extensive utilization and demonstrated effectiveness of these techniques across diverse domains underscore their significance in elevating the performance of models and addressing the challenges associated with overfitting.

2.7 Summary

The literature survey on deep learning architectures, distributed deep learning, deep learning frameworks, performance modelling, regularization techniques, and the differ-

ential evolution algorithm has unveiled the crucial roles of these elements in scalable deep learning in distributed environments. The survey highlighted the significance of distributed deep learning, which efficiently processes large datasets by leveraging the computational power of distributed systems. Given the exponential growth of data, distributed deep learning has become essential for handling the computational and memory requirements of modern AI models.

Additionally, various deep learning frameworks: TensorFlow, PyTorch, and MXNet, offer robust tools for implementing deep learning models efficiently, contributing significantly to their widespread adoption across industries and academia.

Moreover, the literature review emphasized the importance of performance modelling for scalable deep learning in distributed environments. Optimizing deep learning models for various parameters through performance modelling enables efficient resource utilization, reduces training time, and enhances overall system performance. Previous studies have demonstrated the value of analytical and empirical modelling techniques in predicting system behaviour, identifying performance bottlenecks, and guiding optimization efforts.

Furthermore, the differential evolution algorithm was discussed as a powerful evolutionary optimization technique used in various fields, including deep learning. By iteratively refining potential optimization solutions through mutation, crossover, and selection steps, the differential evolution algorithm effectively addresses complex optimization problems with non-linear, non-differentiable, and noisy objective functions.

Chapter 3

Performance Analysis of Distributed Deep Learning Frameworks in a Multi-GPU Environment

The work presented in this chapter was summarized in a paper presented at IUCC-2021, IEEE 20th International conference on ubiquitous computing and communications [122].

This chapter presents an experimental analysis and performance model for assessing deep learning models, including Convolutional Neural Networks (CNNs), Multilayer Perceptron (MLP), and Autoencoder, on the three frameworks: TensorFlow, MXNet, and Chainer, in a multi-GPU environment. These frameworks provide the basic building blocks for designing effective neural network models for various applications such as computer vision, speech recognition, and natural language processing. Factors that influence the performance of these frameworks were analyzed by computing their running time in the proposed model while taking the load imbalance factor into account. The evaluation results highlight significant differences in the scalability of the frameworks and the importance of load balance in parallel distributed deep learning.

3.1 Background and Motivation

Existing works have investigated various aspects of deep learning performance modeling on distributed systems [10], including predicting the performance of asynchronous

stochastic gradient descent [83], analytical models for estimating the optimal use of GPU resources for deep learning [84], and evaluating and benchmarking the performance of deep learning frameworks on GPUs [85] [86]. In this study, the work presented in [86] is extended to analyze and refine some parts of the model by further dividing the timings for stages of the training. The study also considers the effect of load imbalance on the performance of three distributed deep learning frameworks (TensorFlow, MXNet, and Chainer) with Convolutional Neural Network (CNNs), Multilayer Perceptron (MLP), and Autoencoder (AE) models. The evaluation is conducted in the context of a single node, multi-GPU system. The contributions outlined in this chapter are:

- Different from the existing works, the performance model built based on synchronous stochastic gradient descent (S-SGD) analyzes the execution time performance of deep learning frameworks in a multi-GPU environment, taking into account the load imbalance factor and mini-batch time (time taken to divide mini-batches). The model evaluates three deep learning models (Convolutional Neural Networks, Autoencoder, and Multilayer Perceptron), each implemented in three frameworks (MXNet, Chainer, and Tensorflow) respectively.
- Using experimental data, the effect of load imbalance on the scalability of deep learning models was analyzed, concluding that it is an important contribution to parallel inefficiency.

3.2 The Proposed Performance Model

3.2.1 Preliminaries

The notations used in the experiment are shown in Table 3.1.

Table 3.1. Notation used in this chapter are (after [86])

Symbol	Description
N_g	Number of total GPUs
t_{iter}	An Iteration time
t_{io}	I/O time of an iteration
t_{h2d}	Communication time between CPU and GPU of an Iteration
t_{md}	Time for dividing batches into mini-batches
t_f	Forward operation time of an iteration
t_b	Backward operation time of an iteration
$t_{f_i}^{(l)}$	Time taken by i^{th} GPU for l^{th} layer in forward operation
$t_{b_i}^{(l)}$	Time taken by i^{th} GPU for l^{th} layer in backward operation
t_{c_i}	Time taken by i^{th} GPU for computing gradients aggregation
t_u	Model update time of an iteration
t_c	Gradients aggregation time of an iteration

3.2.2 Mini-batch stochastic gradient descent(SGD)

Let's consider an L-layered DNN model, which is trained iteratively on a GPU using mini-batch SGD. Each iteration consists of five steps:

1. Fetch a training data mini batch from either internal or external disk t_{io};
2. Transfer the training data from CPU memory to GPU memory through PCIe t_{h2d};
3. Perform feed-forward calculations layer by layer by using GPU kernels $t_{f_i}^{(l)}$;
4. Use backward propagation for gradients computation from Layer L to Layer 1 $t_{b_i}^{(l)}$;
5. Calculate average gradients and update the model t_u.

An iteration time can be expressed as:

$$t_{iter} = t_{io} + t_{h2d} + t_f + t_b + t_u = t_{io} + t_{h2d} + \sum_{l=1}^{L} t_f^{(l)} + \sum_{l=1}^{L} t_b^{(l)} + t_u \tag{3.1}$$

3.2.3 Synchronous stochastic gradient descent (S-SGD) using multiple GPUs

In comparison with the SGD, S-SGD consists of six steps. The steps 1 to 4 are similar to the SGD. The fifth step is gradient aggregation, and the sixth step is updating the model. The iteration time of the S-SGD implementation can be represented as:

$$t_{iter} = t_{io} + t_{h2d} + \sum_{l=1}^{L} t_f^{(l)} + \sum_{l=1}^{L} t_b^{(l)} + \sum_{l=1}^{L} t_c^{(l)} + t_u \tag{3.2}$$

In the single GPU environment, $\sum_{l=1}^{L} t_c^{(l)} = 0$.

3.2.4 The Proposed Performance Model based on S-SGD

In this chapter, different from the existing works [86], a performance model of S-SGD is built with the inclusion of two new parameters: time taken to divide the batch into mini-batches and maximum time taken by GPU, taking load imbalance factor into account. The importance of the selection of these two parameters are to analyse and to refine the parts of the model by further dividing the timings for stages of the training.

Assume that a machine contains N_g GPUs. Given the model to be trained, each GPU will individually keep a complete set of model parameters, although parameter values are identical and synchronised across GPUs. For an example, Figure 3.1 describes the work-flow of the performance model when $N_g = 4$. In general, the model works as discussed in section 3.2.3 using multiple GPUs. Thus, the proposed performance model of training DNNs with S-SGD in the TensorFlow, MXNet, and Chainer frameworks is developed.

Here, S-SGD executes feed-forward and backward propagation simultaneously on each GPU with the same model and distinct training datasets. The time taken for dividing each batch into mini-batches and the maximum time taken by each GPU in forward processing are considered. By substituting these two parameters in the modeling function, the iteration time t_{iter} for the S-SGD implementation can be represented as follows:

$$t_{iter} = t_{io} + t_{h2d} + t_{md} + max_{i\epsilon[1,N_g]}(\sum_{l=1}^{L} t_{f_i}^{(l)} + \sum_{l=1}^{L} t_{b_i}^{(l)} + \sum_{l=1}^{L} t_{c_i}^{(l)}) + t_u \tag{3.3}$$

In the single GPU environment, $\sum_{i=1}^{L} t_c^l = 0$. The time of an iteration can be written as:

$$t_{iter} = t_{io} + t_{h2d} + t_{md} + \sum_{l=1}^{L} t_f^{(l)} + \sum_{l=1}^{L} t_b^{(l)} + t_u \tag{3.4}$$

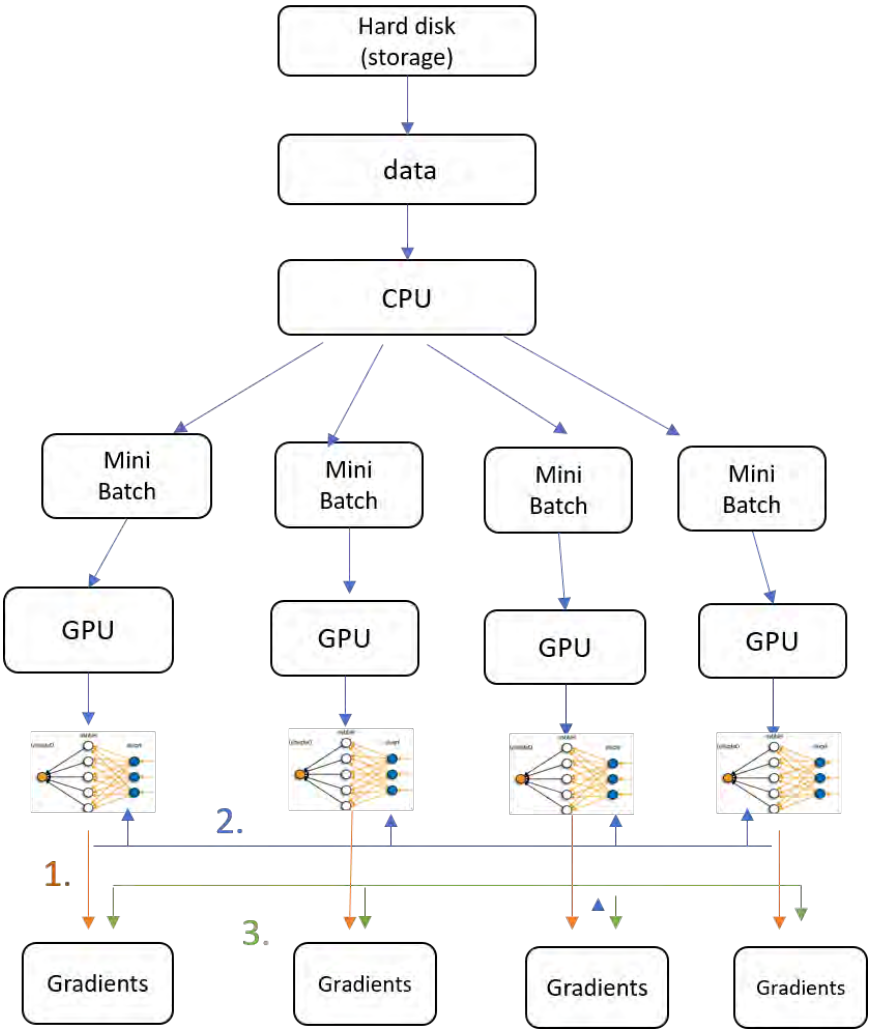


Figure 3.1. Workflow of the model: (1) loss and gradient computation, (2) gradient aggregation, and (3) parameter update

Now consider the effects of optimization strategies [123], which make use of task parallelism, which are found in the existing deep learning frameworks. Two possible optimization opportunities can be noticed [123]. Initially, parallelizing data reading tasks with the computing tasks effectively hides the time cost of disk I/O. Secondly, gradient commu-

nication tasks with the backpropagation computing tasks can be parallelized. In the case of overlapping I/O with computation, the first step is frequently processed with multiple threads, allowing the I/O time of a new iteration to overlap with the computing time of the preceding iteration [124]. In such a manner, computing in the following iteration can begin immediately after the model is completed. Thus, the average iteration time of pipelined SGD is calculated as:

$$t_{iter} = \max(t_f + t_b + t_u \ , \ t_{io} + t_{h2d}) \tag{3.5}$$

In a scenario where the gradient communication overlaps with the computation, the gradient communication could be re-programmed to run concurrently with the backpropagation steps. Therefore, the overheads of I/O and gradient communications need to be reduced to achieve good performance and scalability, Let t'_{iter} and t'_{io} represent the iteration time and I/O times respectively on N_g GPUs. The speedup of using N_g GPUs is the given by:

$$S = N_g \frac{t_{iter}}{t'_{iter}} \tag{3.6}$$

Accounting for the optimizations described above, the expression can now be written as:

$$S = N_g \frac{\max\left\{t_{io} + t_{h2d} \ , \ t_f + t_b + t_u\right\}}{\max\left\{t'_{io} + t_{h2d} + t_{md} \ , \ t_f + t_b + t_c\right\}} \tag{3.7}$$

The speedup, denoted by S, is calculated as the ratio of the maximum time taken in the single-GPU case to the maximum time taken in the multiple-GPU case. The numerator represents the maximum time for a single GPU, considering the time for I/O and communication (represented by $t_{io} + t_{h2d}$) and the time for forward and backward computations along with model updates (represented by $t_f + t_b + t_u$). The denominator represents the maximum time for multiple GPUs, considering the time for I/O, communication, and dividing batches into mini batches (represented by $t_{io} + t_{h2d} + t_{md}$), and the time for forward, backward computations, and gradients aggregation (represented by $t_f + t_b + t_c$). To summarize, this equation quantifies the speedup achieved by using multiple GPUs compared to a single GPU, considering all the relevant factors in a multi-GPU environment.

3.3 Experiments

In this section, the experimental environment is described, and the results of experiments are presented to investigate the running time performance of DNN models and frameworks, and how communication tasks affect the scalability of S-SGD.

3.3.1 Experimental Setup

Initially, the hardware specifications conducted in the experiments are defined. A single node with three GPUs was used. GPU@ GEFORCE RTX 2080, CPU@ 2.60 GHZ 2.81 GHZ, and Memory (RAM) - 16.0 GB. Software used for the experimentation are TensorFlow version-2.1.0, MXNet version -1.6.0, Chainer version-7.4.0, python version-3.6.9, CUDA version-10.2. and operating system- Linux. Nsight profiler [125] was used to find the running time performance of GPU activity. The Nsight profiling tool collects and views profiling data through the command-line. It provides valuable insights into various CUDA-related activities on both the CPU and the GPU, such as kernel execution, memory transfers, memory set, CUDA API calls, and performance measures for CUDA kernels.

Furthermore, the time duration of an iteration for processing a mini-batch of input data is measured to evaluate the execution performance. Three Neural Network models, i.e., the Multilayer Perceptron (MLP), Convolutional Neural Network (CNN), and Autoencoder (AE) models, are chosen for evaluation. The models are trained on the MNIST dataset on three frameworks, i.e., TensorFlow [65], MXNet [66], and Chainer [67], by applying distributed and parallel training. The MNIST dataset contains 70,000 images of ten handwritten digits and is divided into training and test datasets. The training dataset has 60,000 images, while the test dataset contains 10,000 images. All the two datasets have 10 classes, the 10 numerical digits. In the experiment, two epochs are run, the result of the first epoch is discarded since this will include some setup, which is not representative of the average training load over a long run. Each iteration time is recorded and averaged over the second epoch, calculating the mean and standard deviation of each time.

3.3.2 Performance Metrics

The speedup and load imbalance factor are selected as performance metrics for run time evaluation on three different frameworks. The speedup is defined in Equation 3.7. The load imbalance factor for a set of parallel process which execute in times $t_1 \ldots t_N$ is defined as

$$LIF = N \cdot \frac{max_{i\epsilon[1\ldots N]} t_i}{\sum_{i=1}^{N} t_i} \tag{3.8}$$

$LIF = 1$ corresponds to a perfectly balanced load, whereas for imbalanced loads, $LIF > 1$.

The experimental evaluation is focused on two goals below.

- The first experimental goal is to investigate running time performance of each model using different frameworks in a multi-GPU environment.
- The second experimental goal is to investigate how load imbalance factor of each model under different computing nodes/GPU affects the computing efficiency.

3.4 Results and Analysis

This section illustrates the running performance followed with analysis based on the performance modelling of Chainer, MXNet and TensorFlow in training CNN, MLP and Autoencoder models in a multiple GPU environment.

3.4.1 Single GPU

Initially, the performance results obtained on a single GPU are described. The average time taken by a framework to complete one iteration during training evaluates the framework's performance. Consequently, a comparison of the time spent on each step of SGD becomes possible. The timings are provided in Table 3.2 and illustrated in Figures 3.2, 3.3, and 3.4. The results of each phase will be discussed in the following sections.

Table 3.2. Measured time of SGD phases on single GPU. All times are given in seconds, as the mean and standard deviations over all iterations in a single epoch of training.

CNN	Chainer	MXNet	TensorFlow
t_{io}	0.0004±0.00002	0.0002±0.00005	0.0006±0.00008
t_{h2d}	0.0383±0.0054	0.0201±0.0027	0.0212±0.0023
t_{md}	0.0006±0.00003	0.0003±0.00001	0.0005±0.00002
$\sum_{i=1}^{l} t_{f_i}^l$	0.0663±0.0031	0.0307±0.0073	0.3489±0.0729
$\sum_{i=1}^{l} t_{b_i}^l$	0.0594±0.0030	0.1347±0.0040	0.1151±0.0170
t_u	0.2365±0.0194	0.1564± 0.0514	0.2636±0.0469
t_{iter}	0.4009±0.0240	0.3421±0.0354	0.7494 ±0.1391

MLP	Chainer	MXNet	TensorFlow
t_{io}	0.0001±0.000018	0.0005±0.00008	0.0003±0.000025
t_{h2d}	0.0331±0.0062	0.0182±0.00078	0.0199±0.0035
t_{md}	0.0006±0.00001	0.0003±0.00003	0.0005±0.00008
$\sum_{i=1}^{l} t_{f_i}^l$	0.0523±0.0067	0.1034 ±0.0082	0.0576±0.0045
$\sum_{i=1}^{l} t_{b_i}^l$	0.0481±0.0280	0.1754±0.0187	0.1680 ±0.0134
t_u	0.4533±0.0095	0.2054±0.00099	0.5985±0.0089
t_{iter}	0.5869±0.0575	0.3992±0.0591	1.4371±0.0597

AN	Chainer	MXNet	Tensorflow
t_{io}	0.0004±0.00005	0.0001±0.00003	0.0005±0.00008
t_{h2d}	0.0316±0.0026	0.0185±0.0090	0.0215±0.0030
t_{md}	0.0006±0.00003	0.0003±0.00001	0.0005±0.00008
$\sum_{i=1}^{l} t_{f_i}^l$	0.1388±0.0045	0.1322±0.0064	0.1595±0.0072
$\sum_{i=1}^{l} t_{b_i}^l$	0.1421± 0.0056	0.2265± 0.0076	0.4274±0.0103
t_u	0.3675±0.0201	0.3287±0.0307	0.3765 ±0.0215
t_{iter}	0.6804±0.0328	0.706±0.0258	0.9854±0.0320

In the initial phase of the performance model, all three frameworks have multiple threads to read data from the CPU memory to the GPU. By observing the results in Table 3.2, therefore, evident that for all frameworks, the I/O time remains small. In the second phase, after reading of data from disk to memory, the data should be transmitted to the GPU for training. The tested environment uses PCIe to connect the CPU and GPU, which provides a total bandwidth of 11 GB/sec. From the results in Table 3.2, apparent that Chainer typically exhibits a higher memory copy time than both TensorFlow and MXNet.

In the third phase (t_{md}), the three frameworks differ in the data distribution to GPUs. In the Chainer framework, the data batch divided into multiple batches on the GPU, whereas in the MXNet and Tensorflow frameworks, batches divided into mini-batches on the CPU and then transferred to the GPUs dynamically. As a result, the Chainer framework takes

Table 3.3. Gradient aggregation time in seconds in the multi-GPU experiments.

Network	Framework	t_{comm}	
		2 GPUs	3 GPUs
CNN	Tensorflow	0.3945	0.4017
	MXNet	0.3245	0.3415
	Chainer	0.3106	0.3404
MLP	Tensorflow	0.3024	0.4145
	MXNet	0.3156	0.2569
	Chainer	0.2945	0.2345
Autoencoder	Tensorflow	0.7187	0.7199
	MXNet	0.3565	0.3698
	Chainer	0.4563	0.4583

0.3s longer compared to MXNet and TensorFlow.

n the forward phase, it can be seen that while the results are comparable in the case of the Autoencoder and MLP models, in the CNN model, Tensorflow significantly slower than both the MXNet and Chainer frameworks. MXNet's performance good in the forward phase due to its usage of auto symbolic differentiation and imperative programmimg [66]. In the case of the CNN, both Chainer and MXNet able to auto-tune to determine the optimal convolutional algorithms for convolutional layers, but TensorFlow does not allow the convolution techniques to be customized. TensorFlow uses the Winograd algorithm, which in some situations may be suboptimal. Considering the CNN model, MXNet makes use of GEMM-based convolution, which results in 0.05s less in the forward phase and up to 0.15s more in the backward phase. Chainer employs the FFT technique [11], which results in a forward phase that 0.06s higher and 0.1s less in the backward phase.

Next, in the backward phase, MXNet is slower than the TensorFlow and Chainer frameworks. The values of t_f and t_b differ in performance due to the differing use of the cuDNN API. cuDNN may have different performance depending on the parameters used. Some factors that affect performance are: data layout, implicit matrix multiplications, dimension quantization techniques, convolution parameters such as batch size, Height and width filtersize, channels in and out (NHWC, NCWH), and strides. For example, in MXNet and Chainer, the NCHW data layout is used, whereas TensorFlow has NHWC layout, which acts as a performance factor.

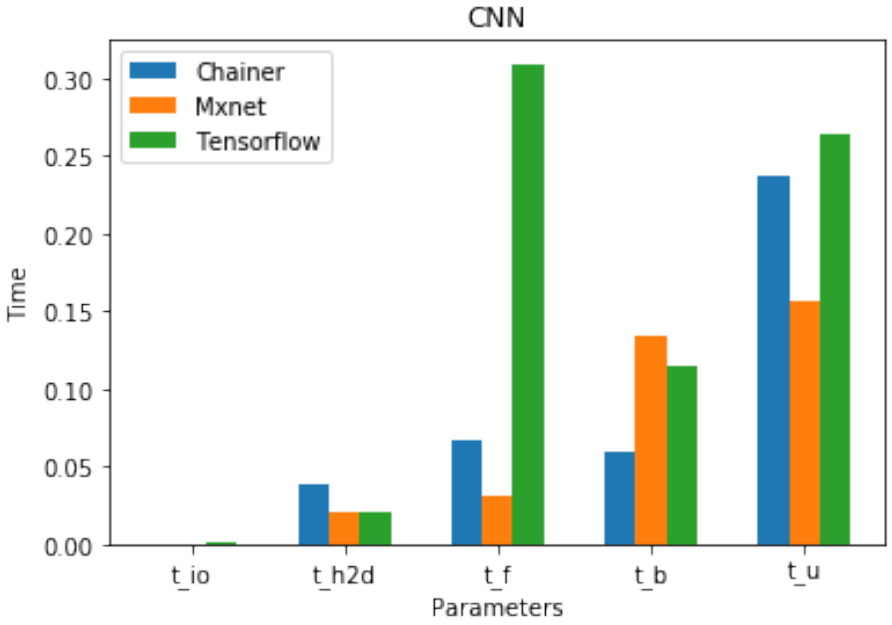


Figure 3.2. Iteration times on a single GPU for the CNN model

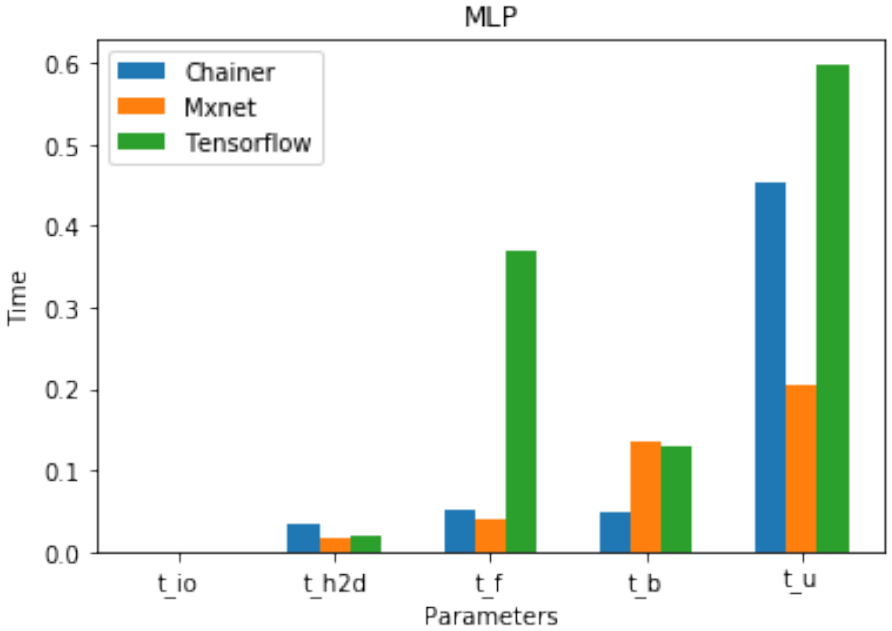


Figure 3.3. Iteration times on a single GPU for the MLP model

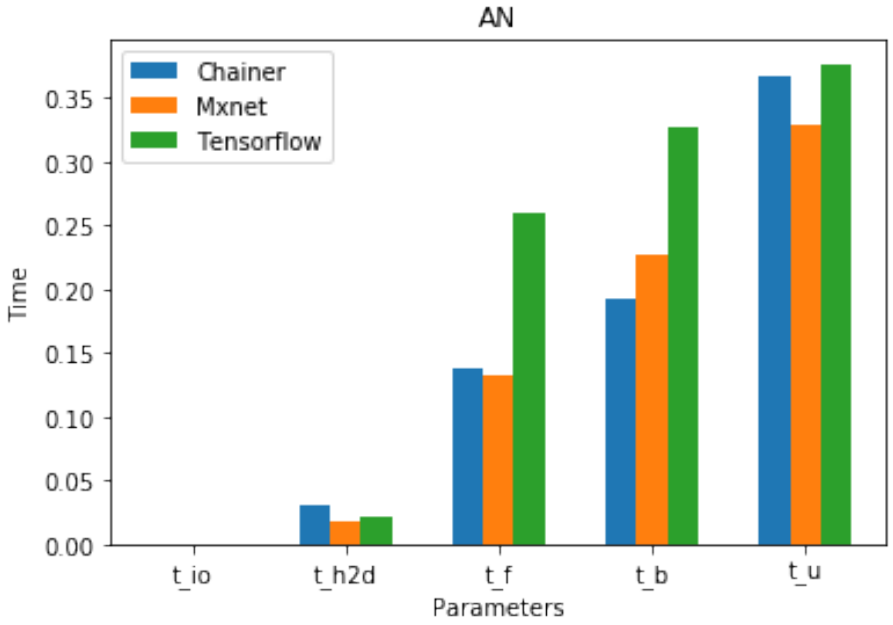


Figure 3.4. Iteration times on a single GPU for the Autoencoder model

3.4.2 Multi-GPU

In multi-GPU testing, the mini-batch was scaled with the number of GPUs, with each GPU having the same dataset. As the number of GPUs increases, data communication overhead increases due to the data aggregation process between devices. Measurements of this time, t_{comm}, are give in Table 3.3. Figures 3.5, 3.6 and, 3.7 shows the results for the speedup when running on two and three GPUs, and the breakdown of the timings in terms of the performance model are shown in Figures 3.8 to 3.13.

From Figures 3.5, 3.6, and 3.7 it can be observed that MXNet achieves linear scaling from one to three GPUs, while Chainer achieves speeds 0.2X less than MXNet. From Figures 3.8 to 3.13, it is evident that the data aggregation time t_a in MXNet is less than in the TensorFlow and Chainer frameworks. Here, MXNet parallelizes the gradient aggregation with back propagation i.e., after the gradients of the current layer(l_i) are computed, the preceding layer (l_{i-1}) of backward propagation can be performed without latency. As a result, gradient computation of (l_{i-1}) is parallelized with gradient aggregation of l_i. Thus, following computing layers can hide much of the synchronisation overhead of gra-

dients. As a result, MXNet has less aggregation time and good scalability compared to other frameworks. TensorFlow implements S-SGD differently. It has no parameter server and uses peer-to-peer memory access if it is compatible with the hardware topology. Each GPU receives gradients from other GPUs, averages them, and updates the model when the backward propagation completes, even from the decentralised method. In this process, the model update t_u and backward propagation has no computation overlap, which led to the observed relatively poor scaling performance in TensorFlow.

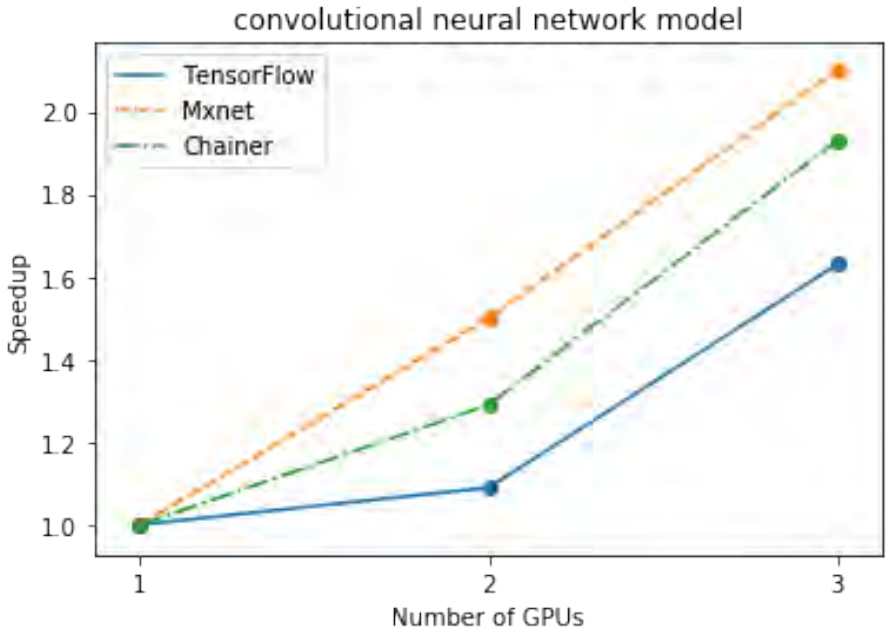


Figure 3.5. Measured speedup for the three frameworks on different numbers of GPUs for the CNN model

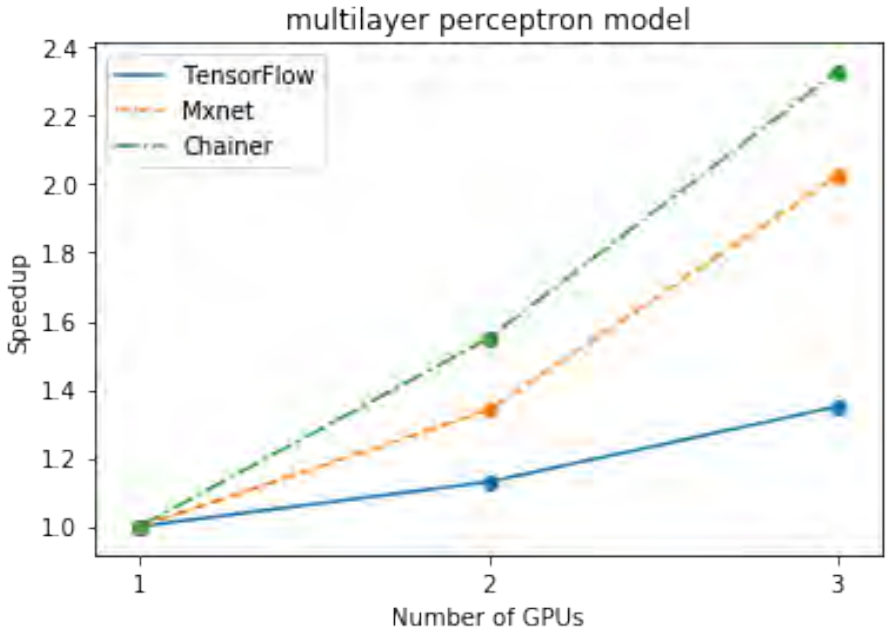


Figure 3.6. Measured speedup for the three frameworks on different numbers of GPUs for the MLP model.

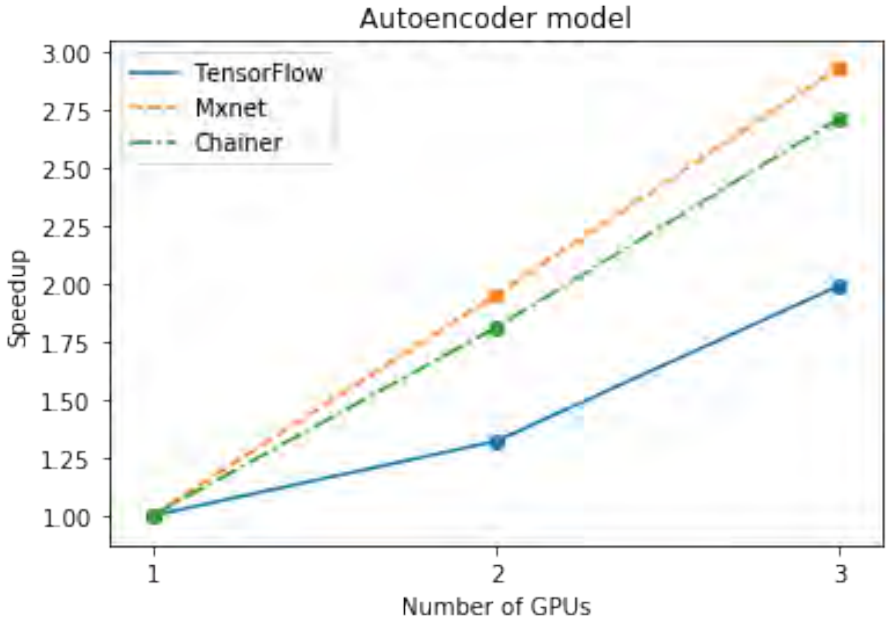


Figure 3.7. Measured speedup for the three frameworks on different numbers of GPUs for the Autoencoder model.

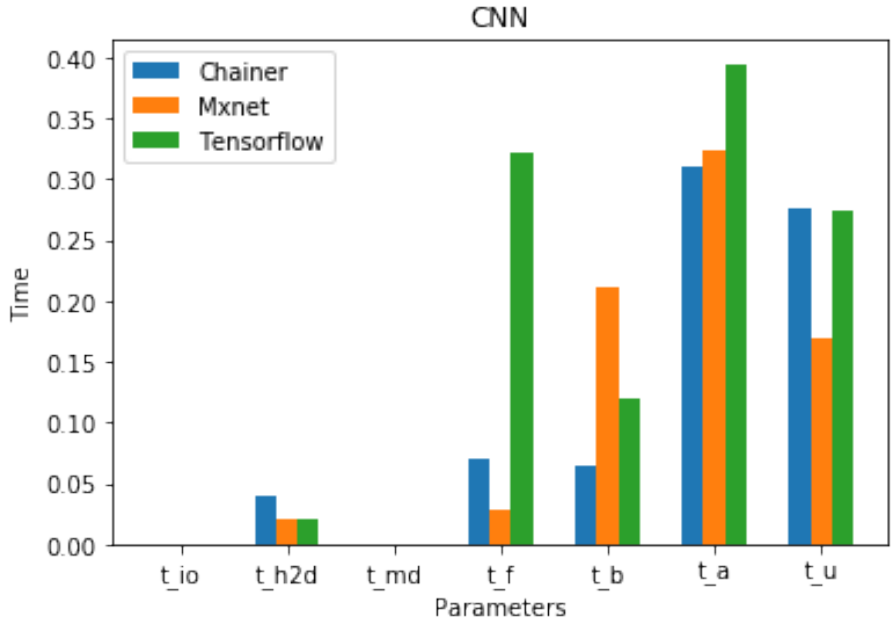


Figure 3.8. Iteration time on multiple GPUs. Results for two GPUs for the CNN model

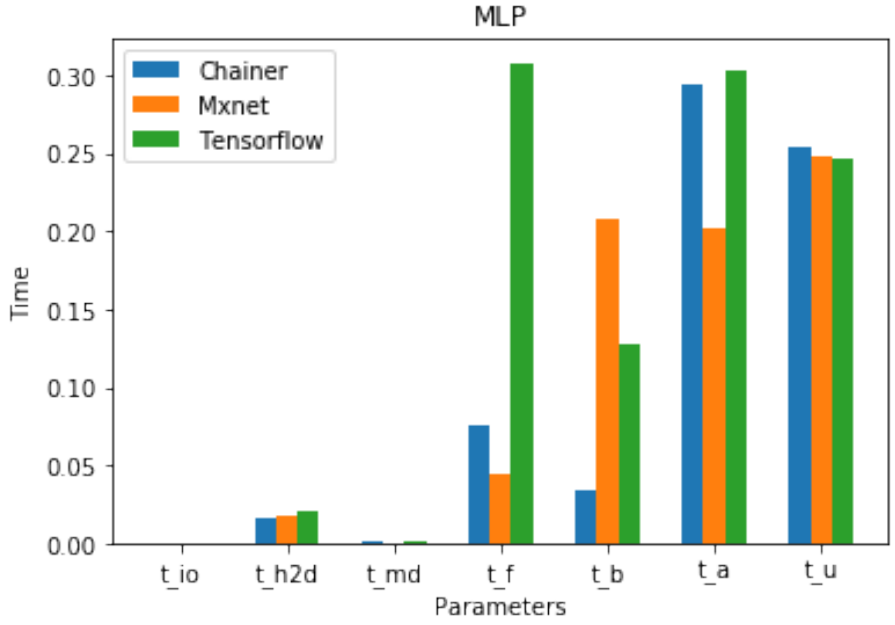


Figure 3.9. Iteration time on multiple GPUs. Results for two GPUs for the MLP model.

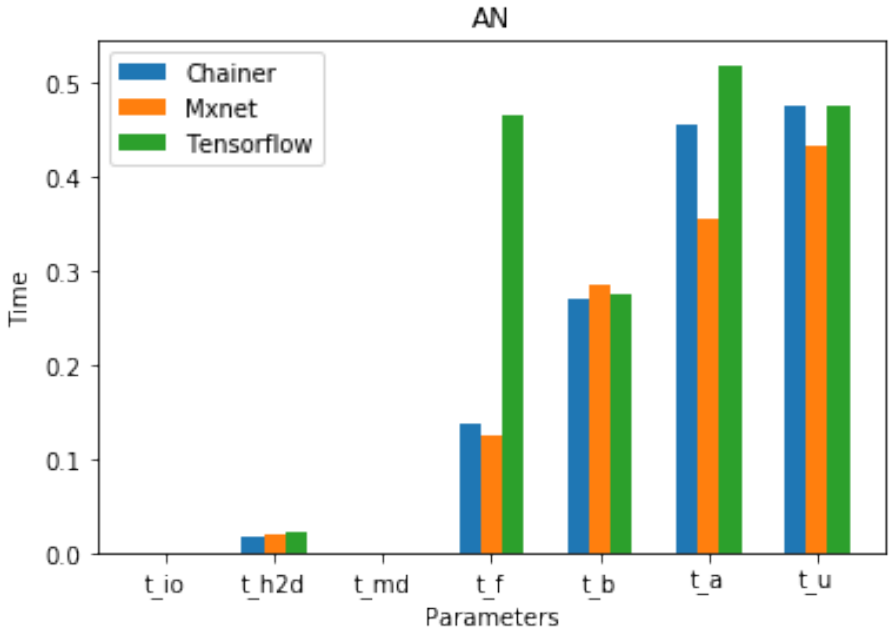


Figure 3.10. Iteration time on multiple GPUs. Results for two GPUs for the Autoencoder model.

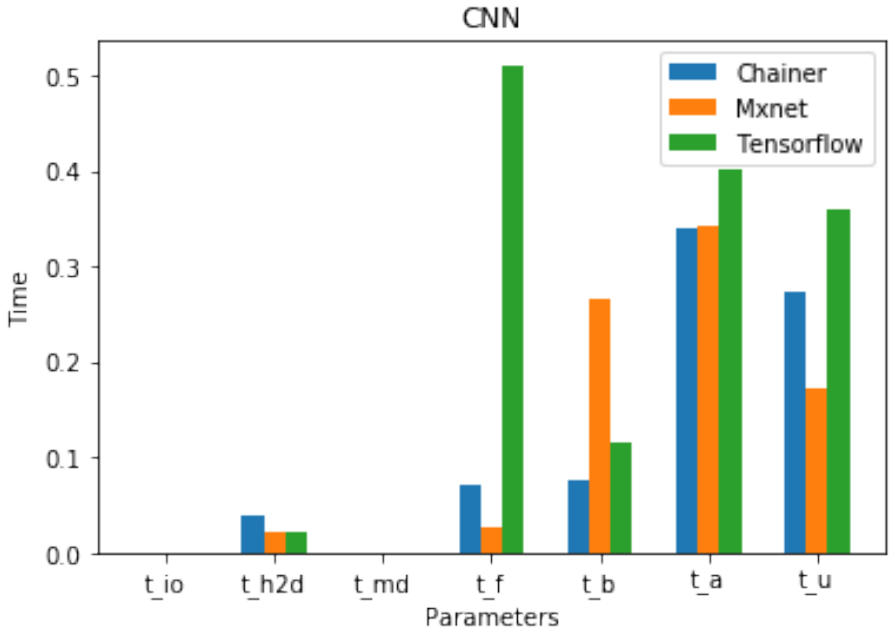


Figure 3.11. Iteration time on multiple GPUs. Results for three GPUs for the CNN model

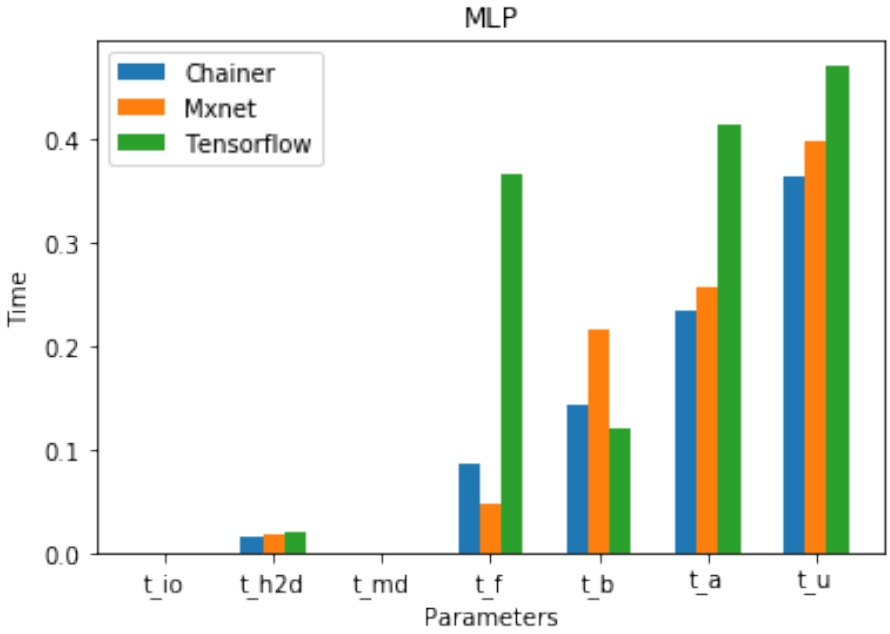


Figure 3.12. Iteration time on multiple GPUs. Results for three GPUs for the MLP model.

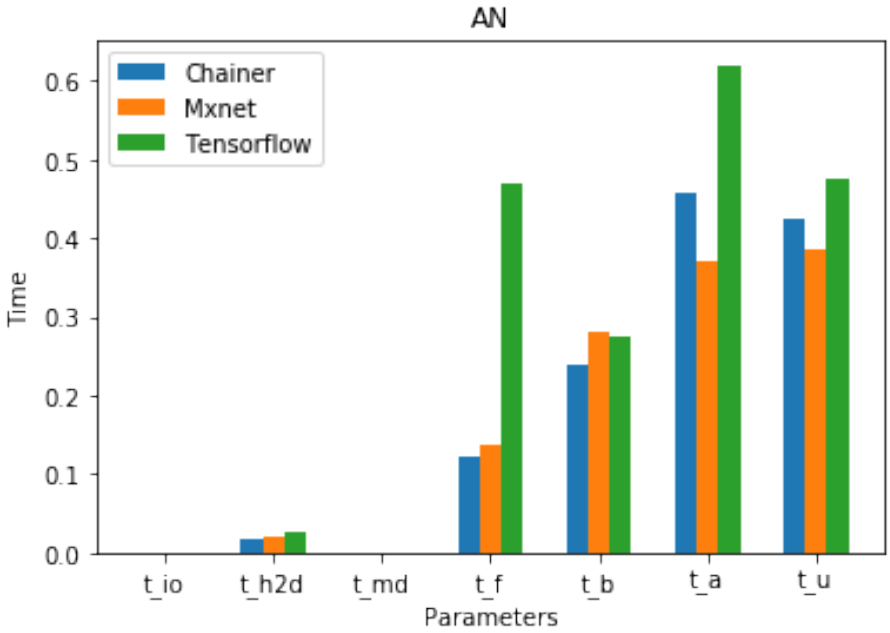


Figure 3.13. Iteration time on multiple GPUs. Results for three GPUs for the Autoencoder model.

3.4.3 Load Imbalance Factor

Load balancing [126] [127] in a parallel system plays a major role in determining scalability. A load imbalance occurs when work is distributed unevenly among workers. Here, the *Load Imbalance Factor* for each neural network model in each deep learning framework has been calculated based on Equation 3.8.

From the results in Table 3.4, it is clear that all three frameworks are not well balanced, since in all cases the load imbalance factor is greater than one. Qualitatively, the higher values of load imbalance correspond to the lower speedups and degraded scalability, as shown in Figures 3.6, 3.7, and 3.8. For example, in the case of Tensorflow, poor scalability is accompanied by relatively high values of the load imbalance factor.

Table 3.4. Load Imbalance Factor

Network	Framework	Load Imbalance Factor	
		2 GPUs	3 GPUs
TensorFlow	CNN	1.15	1.23
	MLP	1.189	1.20
	AN	1.175	1.27
Chainer	CNN	1.025	1.052
	MLP	1.032	1.043
	AN	1.152	1.202
MXNet	CNN	1.013	1.030
	MLP	1.015	1.079
	AN	1.142	1.213

Here, further linear regression analysis is presented to understand how the load imbalance factor contributes to parallel inefficiency, according to the equation below:

$$t = \beta_0 + \beta_1 N_g + \beta_2 l_f + \epsilon \tag{3.9}$$

where N_g and l_f represent the number of GPUs and load imbalance factor respectively, t is the total execution time of an epoch, β_0, β_1, β_2 are the regression coefficients and ϵ represents a random value indicating the error in each observation of t. The values of β_0, β_1, β_2 should be chosen to minimise the sum of squared prediction errors.

The following values for the coefficients in the nine cases are found and shown in Table 3.5.

Table 3.5. Regression Coefficients and R^2 Values for Each Model and Framework.

Model	Deep learningFramework	β_0	β_1	β_2	R^2
CNN	**Tensorflow**	**0.00001**	**40.5588**	**369.4848**	**0.9904**
MLP	**Tensorflow**	**0.00001**	**16.1671**	**245.9177**	**0.9963**
AN	**Tensorflow**	**0.00002**	**49.1037**	**398.6701**	**0.9960**
CNN	**MXnet**	**0.00001**	**5.57483**	**287.4133**	**0.9901**
MLP	**MXnet**	**0.00002**	**28.9346**	**326.7283**	**0.9987**
AN	**MXnet**	**0.00002**	**33.1037**	**398.6701**	**0.9903**
CNN	**Chainer**	**0.00001**	**9.00791**	**286.8447**	**0.99**
MLP	**Chainer**	**0.00001**	**0.1584**	**292.1287**	**0.9963**
AN	**Chainer**	**0.000068**	**25.584**	**260.263**	**0.9965**

where $t^{'}$ is the computed prediction execution time as a function of the number of GPUs and the load imbalance factor. The coefficients of determination, R^2, are computed to further investigate the impact of the number of GPUs and load imbalance factor on execution time. R^2 represents the proportion of the variance in execution time that is predicted from both number of GPUs and load imbalance factor. It is defined as follows:

$$R^2 = 1 - \frac{SS_{rss}}{SS_{tss}} \tag{3.10}$$

where SS_{rss} and SS_{tss} are the residual sum of squares and the total sum of squares. They are defined as:

$$SS_{rss} = \sum (t - t^{'})^2 \tag{3.11}$$

and,

$$SS_{tss} = \sum (t - \bar{t})^2 \tag{3.12}$$

R^2 values for (CNN, MLP, AE) TensorFlow are found to be (0.9904, 0.9963, 0.9960), for MXNet (0.9901, 0.9987, 0.9903), and for Chainer (0.99, 0.9963, 0.9965). These results imply that the regression forecasts are accurate in predicting the relationship between execution time and load imbalance factor. As the value of R^2 increases, the model's fit to the training data becomes more accurate and precise. The results confirm the importance of load balancing to achieve scalability in distributed deep learning.

3.5 Summary

The performance of different deep learning frameworks over different deep learning neural networks has been evaluated in terms of scalability in a multi-GPU environment, taking into account a range of factors affecting performance, including load imbalance. The existing performance model [86] has been further extended with the inclusion of two new parameters: time taken to divide the batch into mini-batches and the maximum time taken by GPU. The proposed performance model was built to measure the performance of different deep learning framework implementations which include TensorFlow, MXNet and Chainer frameworks on three models: Convolutional neural network, Multilayer perceptron and Autoencoder models, in a multi-GPU environment. The experimental results have shown that MXNet and Chainer have better scalability compared to TensorFlow for all three models. Moreover, the analysis of the load imbalance factor has shown that load balancing is a contributing factor to scalability in distributed deep learning, and high load imbalance is strongly correlated with poor scalability in the experiments. However, the performance model could not provide deeper insights into where the load imbalance arises. This motivates the development of a more detailed performance model which is fit to the performance data using a global optimization algorithm, and this will be covered in the next chapter.

Chapter 4

A Generic Performance Model for Deep Learning in a Distributed Environment

The work presented in this chapter was summarized in a paper presented at the IEEE SSCI 2022 conference, held in Singapore.

To address the limitations mentioned in previous chapters, the objective in this chapter is to build a performance model that quantifies the efficiency of large parallel workloads. This motivates the development of a more detailed performance model, which fits experimental performance data using a global optimization algorithm. Existing performance models in deep learning are broadly categorised into two methodologies: analytical modelling and empirical modelling as defined in Chapter 2 in sections 2.4.1 and 2.4.2. Analytical modelling uses a transparent approach to convert the model's or applications' internal mechanisms into a mathematical model corresponding to the system's goals, which can significantly expedite the creation of a performance model for the intended system. Empirical modelling predicts the outcome of an unknown set of system parameters based on observation and experimentation. It characterises an algorithm's performance across problem instances and/or parameter configurations based on sample data. These models predict the output of a new configuration on the target machine. In this chapter, the hybridization of the analytical model and empirical modeling serves as an inspiration. Here, a novel generic performance model is proposed that provides a general expression in terms of intrinsic and extrinsic factors of a deep neural network framework in a distributed environment, which gives accurate performance predictions. The contributions outlined in this chapter are:

- Developed a generic expression for a performance model considering the influence of intrinsic parameters and extrinsic scaling factors that affect computing time in a distributed environment.
- Formulated the generic expression as a global optimization problem using regularization on a cost function in terms of the unknown constants in the generic expression, which has been solved using differential evolution to find the best fitting values to match experimentally determined computation times.
- Evaluated the proposed model in three deep learning frameworks, i.e., TensorFlow, MXNet, and PyTorch, to demonstrate its performance efficiency.

4.1 The Proposed Generic Performance Model

Given an application consisting of a number of processes in a distributed environment, the execution time of the application can be considered from two levels: 1) Execution time of internal processes of the application (for example, intrinsic parameters of the application); and 2) External scaling factors that affect the computing efficiency (such as a number of machines/processors or data chunks or batch size). A generic performance model for computing total computational time (t) per iteration of an application can be described as follows:

$$t(I, E) = t_I(I) f_E(E) + C \tag{4.1}$$

Here, intrinsic parameters are represented as I, E represents extrinsic parameters, t_I represents the computation time of the processes affected by intrinsic parameters, f_E represents extrinsic scaling factors that affect the computing performance, and C is a constant. In general, I and E are vectors in which each element is a hyperparameter of the deep learning model such as a filter size (intrinsic) or batch size (extrinsic).

In the model, the internal time t_I is represented as a sum of terms in powers of the

components of I:

$$t_I = \sum_{i=1}^{n} a_i I_i^{p_i} \tag{4.2}$$

Basically, intrinsic parameters represent model parameters of the deep neural network, as shown in Fig.4.1. In equation (4.2), the coefficients a_i relate to the relative importance of the processes, and the powers p_i relate to the computational complexity.

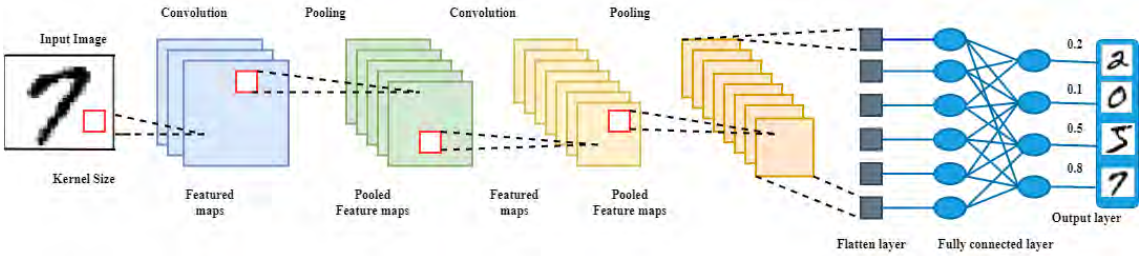


Figure 4.1. Internal processes involved in a convolutional neural network.

The external factors are related to scaling, and these appear in the model as multiplicative terms with different powers in the computation of the external scaling factor f_E, which is given by:

$$f_E = \prod_{j=1}^{m} E_j^{q_j} \tag{4.3}$$

Here, the powers q_j give information about scalability. By substituting the t_I and f_E in equation (4.1), the computational time (t) is given as follows:

$$t(I, E, x) = \left(\sum_{i=1}^{n} a_i I_i^{p_i} \right) \prod_{j=1}^{m} E_j^{q_j} + C \tag{4.4}$$

which is now written as a function of I, E and x, where

$$x = \left\{ a_1, ..., \ a_{n_I}, p_1, ... p_{n_I}, q_1, ... q_{n_E}, c \right\} \in \mathbb{R}^M \tag{4.5}$$

Here, x is a vector formed by combining a, p, q and constant coefficient C. In equa-

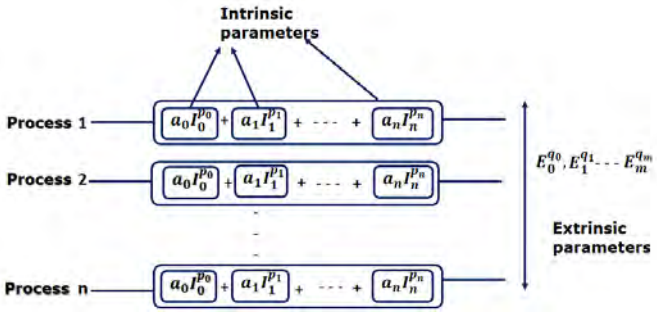


Figure 4.2. Functional diagram of proposed performance model.

tion (4.4), the intrinsic parameters I and extrinsic parameters E are the known input values. a, p, q and coefficient C are unknown constants. The functional diagram of the proposed performance model is shown in Figure 4.2.

The optimal values of these unknown constants (total: $M = 2n_I + n_E + 1$) are computed using the differential evolution algorithm. The aim is to find the best-fitting values of these constants, by fitting the model to experimentally measured execution times obtained with different values of the internal and external parameters I and E. Before going into the cost function formulation of the differential evolution algorithm [128], the general methodology for obtaining the experimental data is described. For every possible combination of values of intrinsic and extrinsic parameters, there will be too many combinations for an exhaustive grid search. Therefore, random sampling has been applied to ensure that every hyperparameter in the population has an equal opportunity of being selected for obtaining measured times. The methodology used for measured time is the time taken for an iteration of an epoch. The iteration time is computed as the difference between an iteration's end time and starting time.

The experimental data for fitting the model comprises N measurements with randomly selected values of the parameters. The values of the intrinsic parameters are denoted by

$$I_{i,k}, \quad i \in [1, n_I], \;\; k \in [1, N] \tag{4.6}$$

where i indexes the components of the vector of parameters, and K refers to a given ob-

servation in the experiment. Similarly, the extrinsic parameters are denoted by

$$E_{j,k}, \quad j \in [1, n_E], \;\; k \in [1, N] \tag{4.7}$$

The measured time for observation k is

$$t_k, \;\; k \in [1, N] \tag{4.8}$$

Here, i, j denote the input feature indices. $k \in [1, N]$ indicate the sample index in dataset D. N is the number of input samples in D.

4.1.1 Global Optimisation Using Differential Evolution

Given the generic expression as shown in equation (4.4), as mentioned in the earlier subsection, the best fit values of a, p, q, and C are found by minimizing a cost function. The cost function is formulated as the mean absolute difference between the predicted execution time and the actual measured times as follows:

$$f(x) = \frac{1}{N} \sum_{k=1}^{N} \left| t_k - \hat{t_k}(I_K, E_k, x) \right| \tag{4.9}$$

where N as number of data samples, t_k is the measured time, $\hat{t_k}$ is the predicted time derived from equation (4.4).

To solve the above optimization problem, the differential evolution algorithm (DE) is used with the cost function in equation (4.4). The mean absolute error between the predicted times of the model and the measured times is minimized, resulting in a value of the vector x which represents the best fitting model. Recall that this vector encodes the coefficients and powers of the terms in the model due to each of the hyperparameters; these can then be used to make predictions of the execution time for any set of values of the intrinsic and extrinsic parameters and furthermore, inspection of these coefficients can provide insight into the relative importance and computational complexity of the internal processes, as well as the scalability of the external processes. For this work, the DE implementation from the *scipy* python package is used, with default values of the hyperparameters. Limits

of $(0 \ldots 1000)$ are enforced for constants and coefficients (a, C) and $-5 \ldots 5$ for powers (p, q).

4.1.2 Regularization

A globally optimized, unconstrained model may be prone to overfitting or producing unstable solutions with high parameter variance. To address these issues, a regularization term is introduced to the cost function. Regularization achieves the best fit by incorporating a penalizing term in the cost function, which assigns a higher penalty to complex curves. Thus, the motivation to apply regularization to the performance model arises. Generally, regularization can be defined as:

$$f_{\text{reg}}(x) = f(x) + \lambda.L \tag{4.10}$$

where λ controls the bias-variance trade-off, and L is some measure of the complexity of the model. In this study two types of regularization techniques have been used. The first one Lasso regression (L1) form, and the second Ridge regression (L2) form. Firstly, L1 regularization, also called a lasso regression, adds the absolute value of the magnitude of the coefficient as a penalty term to the loss function. The L1 regularization solution is sparse. Secondly, L2 regularization, also called ridge regression, adds the squared magnitude of the coefficient as the penalty term to the loss function, and its solution is non-sparse. In L1 regularization, L1 (Lasso) shrinks the less important features coefficient to zero, thus removing some features altogether. L1 works well for feature selection in case there is a huge number of features. In L2 regularization, it adds a penalty as model complexity increases. The regularization parameter λ penalizes all the parameters except the intercept to ensure the model generalizes the data and avoids overfitting. Ridge regression adds the squared magnitude of the coefficient as a penalty term to the loss function. Both kinds of regularization have been applied to the performance model. Now, the cost function for optimization using both L1 and L2 regularizations is as follows:

$$f_{L_1}(x) = \frac{1}{N} \sum_{k=1}^{N} \left| t_k - \hat{t_k} \right| + \lambda. \sum_{k=1}^{N} |x| \tag{4.11}$$

$$f_{L_2}(x) = \frac{1}{N}\sum_{k=1}^{N}|t_k - \hat{t_k}| + \lambda.\sum_{k=1}^{N}x^2 \tag{4.12}$$

Here, applying the regularization term λ controls the bias-variance trade-off in the internal processes.

4.2 Experimental Evaluation

To evaluate the performance of the proposed model, the approach has been applied to three deep learning frameworks (TensorFlow, PyTorch, and MXNet) and extensive experiments have been conducted. The proposed performance evaluation approach is assessed by modeling distributed training of a CNN architecture on a multi-GPU system. The main goal is to investigate how well the predicted execution time fits the experimentally measured time.

4.2.1 System Configuration

The experiments are implemented on a single node containing three GEFORCE RTX 2080 GPUs, each with 2.60 GHz speed and 16 GB GPU RAM. The node also consists of a 2.81 GHz speed CPU machine, 25 Gbps network bandwidth and a CUDA-10.2 with a Linux operating system. Furthermore, the node consists of various software configurations/installations, including PyTorch 1.2.0, Torchvision 0.4.0, Python 3.6, TensorFlow 2.1.0 and MXNet 1.6.0.

4.2.2 Dataset and Model Selection

For three deep learning frameworks, a CNN architecture, LeNet-5, was selected, which Yann LeCun proposed in 1998 as a general common neural network structure for handwritten font recognition. It consists of two convolutional layers, two fully-connected layers, pooled layers for cross-combination and an output layer that predicts values via the fully connected layer. Besides, LeNet-5 works well with handwritten datasets [129], it also

reduces the number of parameters and can automatically learn features from raw pixels [130].

LeNet-5 is trained on three popular datasets, MNIST, fashion-MNIST, and CIFAR-10, using three popular deep learning frameworks: TensorFlow, PyTorch, and MXNet, in a multi-GPU system. Fashion-MNIST [131] serves as a direct drop-in replacement for the original MNIST dataset for benchmarking machine learning algorithms. Each example is a 28x28 grayscale image associated with a label from 10 classes. The CIFAR-10 dataset [132] contains 60,000 images with 32×32 pixels. The images are classified into ten classes - aeroplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck; each has 6,000 images.

4.2.3 Performance Metrics

The scalability and Mean Absolute Percentage Error (MAPE) are selected as performance metrics for run-time evaluation on three different frameworks. The scalability is measured in the powers of external parameters as shown in equation (4.3). The MAPE can be defined as:

$$MAPE = \frac{1}{n} \sum_{i=1}^{n} \left| \frac{y_i - x_i}{x_i} \right| \times 100 \tag{4.13}$$

where y_i = the measured value, x_i = the predicted value, n = total number of data points.

The choice of using MAPE is motivated by its lower sensitivity to outliers. The presence of these outliers in the results justifies the selection of MAPE as the evaluation metric. Additionally, minimizing the mean absolute error between the predicted times of the model and the measured times results in a value of the vector x which represents the best-fitting model. The scaling parameters are used in the proposed evaluation model to evaluate the performance of the deep learning frameworks.

4.2.4 Experiments

The experiment is designed to accomplish the following three goals:

1. The performance evaluation of deep learning frameworks were conducted using the proposed performance model with and without regularization. Specifically, the proposed performance model was applied to three deep learning frameworks: TensorFlow, MXnet, and PyTorch, under two circumstances, with and without regularization.
2. To investigate regularization to find the best value of lambda, and which of L1 and L2 performs better.
3. Comparison of the proposed model with existing black-box machine learning models. The proposed model and the two widely used models, random forest regression, and support vector regressor, were compared to demonstrate their performance and interpretability.

In the experiment, the distributed training of LeNet-5 on MNIST, fashion-MNIST, and CIFAR-10 datasets is performed using different deep learning frameworks such as TensorFlow, MXNet, and PyTorch. The values of the experimental training parameters are created by applying random sampling on a set of intrinsic and extrinsic parameters and its corresponding average training time taken by a deep CNN architecture per iteration. Table 4.1 shows intrinsic and extrinsic parameters and their possible values. The intrinsic parameters are the model's hyperparameters, including kernel size, pooling size, activation function, etc. The number of GPUs and the batch size are extrinsic factors since these affect the scaling over multiple processes.

The experiments involve repeated trials in which the time for a single training iteration is measured using randomly selected intrinsic and extrinsic parameter values. 1500 trials were conducted to prepare a dataset of 1500 data samples. For each sample, three iterations are run with the same parameter values, and the median value of the measured time is taken. The experimental data for 1000 trials are used to fit the performance model or train the standard black-box models for comparison. The remaining 500 are used to evaluate the trained and fitted models. Finally, the experimental parameters are used to build three

Table 4.1. Parameters of the performance model, with ranges of values sampled in the experiments.

Index	Name	Set of possible values considered
Intrinsic parameters		
1	Kernel size	{2,3,4,5}
2	Pooling size	{2,3,4,5}
3	Activation function	{Relu, Tanh, Sigmoid}
4	Optimizer	{Adam, SGD}
5	Image_dataset_name	{MNIST,Fashion-MNIST,CIFAR-10}
6	Number of filters	{4,8,16,32,64}
7	Learning rate	{0.1,0.01,0.001,10^{-4}, 10^{-5}, 10^{-6}}
8	Padding_mode	{valid, same}
9	Stride	{1,2,3}
10	Dropout probability	{0.2,0.5,0.8}
Extrinsic parameters		
11	Number of GPUs	{1,2,3}
12	Batch size	{8,16,32,64,128}

performance evaluation models, such as the Differential evolution (DE) algorithm and two standard black-box models. Each fit is run ten times with different random seeds to obtain the mean and standard deviation for each of the fitted parameters. The performance of these models and their corresponding results are explained in the subsequent subsections.

4.3 Results and Analysis

This section shows the results of the proposed performance model for three well-known deep neural networks, i.e., TensorFlow, MXNet and PyTorch. The performance model is evaluated and compared with two standard black-box regression models: Support vector regressor [133] and random forest regressor [134]. Tables 4.2 and 4.3 compares internal parameters and scalability in various frameworks, respectively.

4.3.1 Performance Evaluation of Deep Learning Frameworks using the Proposed Performance Model without regularization

The differential evolution algorithm is applied to the proposed model and evaluated using the three deep learning frameworks. The actual execution time for training the model

using the three frameworks is recorded and predicted execution times are also generated. Figures 4.3, 4.4, and 4.5 shows the scatter graph of the predicted execution times from the proposed model plotted against the actual execution time for the test dataset. The linear fit to the straight line determines how well the model can predict unseen configurations. Best fit constant coefficients for all frameworks are shown in Table 4.2.

Each fit is run ten times with different random seeds to obtain the mean and standard deviation for each of the fitted parameters. The results show stable and consistent fits for the extrinsic parameters and the additive constant C, suggesting that the scalability results are accurate. There is a higher variance in the intrinsic parameters. Table 4.2 shows that the model gives broadly consistent performance for the constant coefficients, representing the relative importance of the process controlled by categorical parameters. For example, for the activation function coefficients, Adam has a large constant and takes more time than SGD in PyTorch and TensorFlow frameworks. In MXNet, SGD takes maximum time. And also in terms of padding, *same* parameter will take more time than *valid* parameter. However, comparisons of these parameters should only be made *within* a framework since the absolute values will be affected by the scaling behaviour. However, the high variance in some intrinsic parameters suggests that some work needs to be done in extracting insights from these parameters, which will be addressed in the next section.

Table 4.2. Derived intrinsic and extrinsic parameters from the differential evolution-optimized performance model for the three deep learning frameworks. Parameters are given as the mean and standard deviation over ten fits. Here a and p represent coefficients and powers respectively of a term representing an intrinsic parameter, where as q is power in a multiplicative term representing an extrinsic (scaling) parameter.

	MXNet		**PyTorch**		**TensorFlow**	
Intrinsic parameters	a	p	a	p	a	p
Filter size	554.87 ± 311.73	-4.06 ± 0.53	423.36 ± 256.88	-2.88 ± 1.04	346.73 ± 216.24	-3.22 ± 0.78
Kernel size	10.57 ± 7.05	-4.10 ± 0.70	168.54 ± 123.27	-2.34 ± 1.82	54.78 ± 32.91	-4.00 ± 1.41
Pool size	18.08 ± 5.17	-4.21 ± 0.46	209.14 ± 186.87	-3.31 ± 0.92	79.45 ± 53.53	-3.48 ± 1.33
Learning rate	459.50 ± 258.52	3.68 ± 0.62	489.52 ± 221.63	3.21 ± 0.70	458.34 ± 278.03	3.26 ± 0.91
Stride	17.29 ± 6.12	-0.83 ± 0.23	140.64 ± 138.62	-0.63 ± 0.58	29.00 ± 14.54	-1.85 ± 0.90
Dropout probability	1.79 ± 0.75	2.24 ± 1.62	437.06 ± 184.32	1.80 ± 1.66	10.23 ± 9.51	1.87 ± 1.62
Same	2.50 ± 0.97	-	11.02 ± 5.09	-	6.14 ± 1.54	-
Valid	1.56 ± 0.96	-	0.77 ± 1.81	-	1.61 ± 2.24	-
Sigmoid	23.25 ± 10.23	-	475.92 ± 139.65	-	251.57 ± 122.01	-
Relu	21.90 ± 10.40	-	475.56 ± 137.27	-	255.93 ± 122.35	-
Tanh	23.14 ± 10.30	-	444.48 ± 138.25	-	254.28 ± 121.27	-
MNIST	35.75 ± 12.81	-	815.62 ± 69.44	-	232.24 ± 108.77	-
Fashion-MNIST	35.94 ± 12.75	-	815.68 ± 68.39	-	231.33 ± 109.93	-
CIFAR-10	18.57 ± 12.68	-	308.73 ± 53.32	-	124.56 ± 108.01	-
SGD	16.68 ± 10.10	-	361.65 ± 130.64	-	158.74 ± 109.25	-
Adam	16.85 ± 10.32	-	720.15 ± 123.99	-	168.55 ± 108.67	-
Extrinsic parameters	q		q		q	
Batchsize	-0.99 ± 0.003		-1.13 ± 0.01		-1.35 ± 0.08	
No. of GPUs	-0.99 ± 0.004		-1.029 ± 0.001		-0.74 ± 0.001	
Constant term	C		C		C	
	3.703 ± 0.017		12.677 ± 0.038		1.930 ± 0.122	

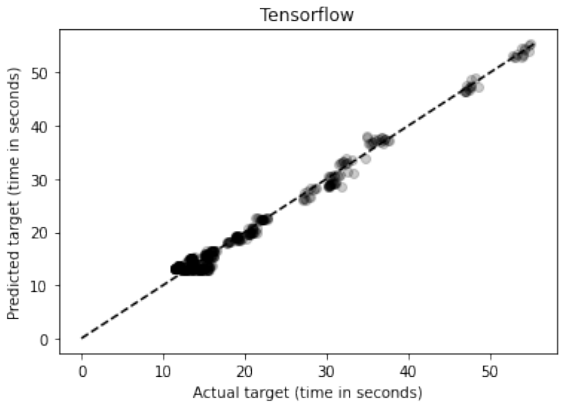


Figure 4.3. The proposed performance model predicted and measured times in TensorFlow deep learning frameworks using differential evolution algorithm.

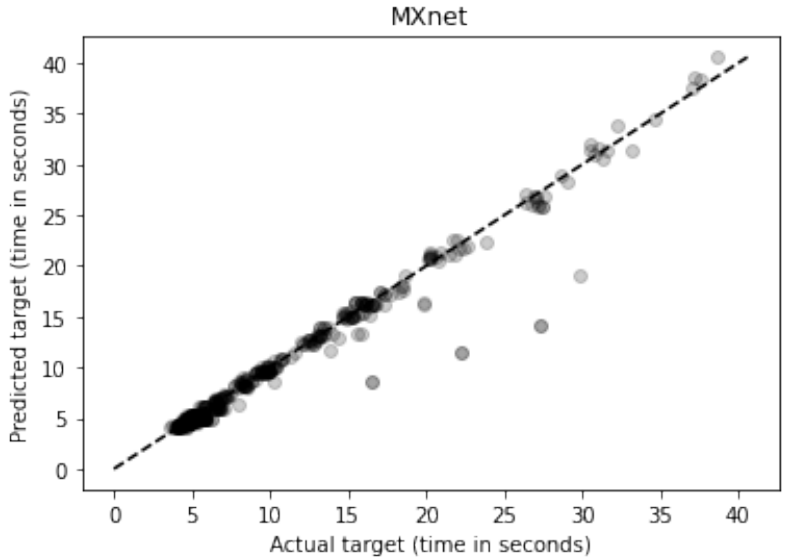


Figure 4.4. The proposed performance model predicted and measured times in MXNet deep learning frameworks using differential evolution algorithm.

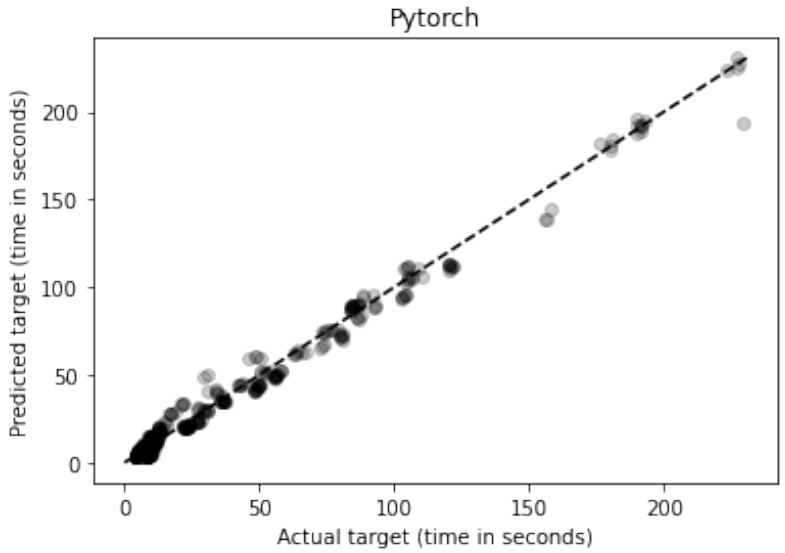


Figure 4.5. The proposed performance model predicted and measured times in PyTorch deep learning frameworks using differential evolution algorithm.

4.3.2 Performance Evaluation of Deep Learning Frameworks using the Proposed Performance Model using regularisation

The proposed performance model was applied to three deep learning frameworks using the differential evolution algorithm with regularization. Actual execution times for training the model using the three frameworks were recorded, and predicted execution times were generated. Scatter graphs of the predicted execution times from the proposed model plotted against the actual execution times are shown in Figures 4.6, 4.7, and 4.8. The linear fit to the straight line assesses the model's ability to predict unseen configurations. Best fit constant coefficients for all frameworks are presented in Table 4.3.

The results demonstrate stable and consistent fits for the extrinsic parameters and the additive constant C, suggesting accurate scalability results. The use of regularization reduces the higher variance in the intrinsic parameters. The model yields consistent performance for the constant coefficients, representing the relative importance of the process controlled by categorical parameters. For example, in PyTorch and TensorFlow frameworks, the activation function coefficients show that Adam has a large constant and takes more time than SGD, while in MXnet, SGD takes the most time. The categorical parameter *padding* with two possible values *valid* and *same* shows better performance for the *same* mode. However, it is important to note that comparisons of these parameters should be made *within* a framework, as absolute values may be influenced by the scaling behavior.

The constant term C, which were derived from the model fitting process. The intrinsic parameters, represented by coefficients a and powers p, signify the relationship between various model attributes (e.g., filter size, kernel size, pool size) and the computation time. On the other hand, the extrinsic parameters, denoted by the power q, describe the influence of external factors, such as batch size and the number of GPUs, on the computation time. The standard deviations associated with the constant term C are relatively small, indicating consistent measurements of its effect and reducing the likelihood of random variations. The constant term C plays a fundamental role in the performance model, capturing the baseline computation time that remains unaffected by changes in the model's parameters and input data. It accounts for aspects of computation that are independent of the model architecture and hyperparameters. The negative values of the extrinsic parameters q indicate that increasing the batch size and utilizing more GPUs lead to reduced

computation time, signifying improved scalability and computational efficiency.

However, the constant term's values demonstrate variations across different deep learning frameworks, suggesting possible framework-specific effects on the baseline computation time. Larger C values imply limited opportunities for further performance optimization through parameter tuning. Consequently, such scenarios may lead to less efficient scaling and diminished performance gains when utilizing larger batch sizes or additional GPUs.

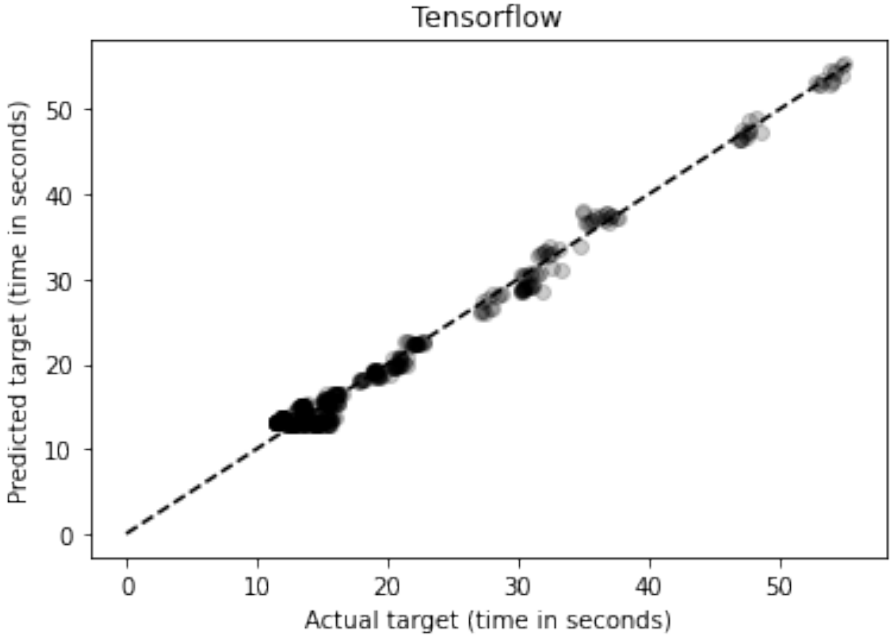


Figure 4.6. The proposed performance model predicted and measured times in TensorFlow deep learning frameworks using differential evolution algorithm using regularization.

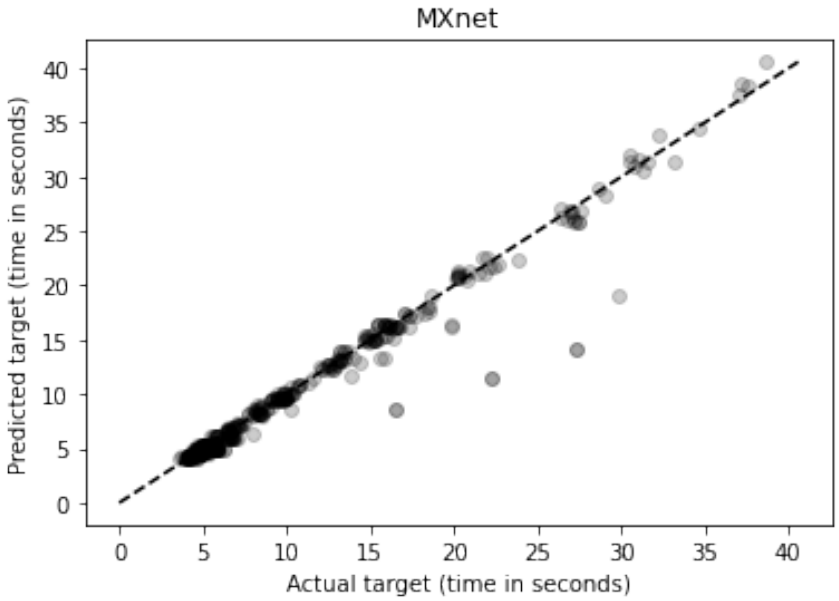


Figure 4.7. The proposed performance model predicted and measured times in MXnet deep learning frameworks using differential evolution algorithm using regularization.

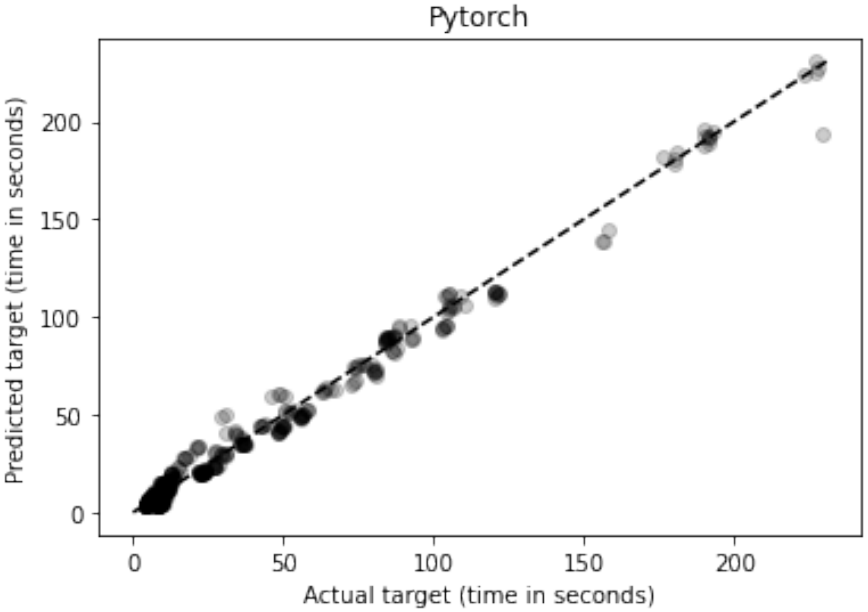


Figure 4.8. The proposed performance model predicted and measured times in PyTorch deep learning frameworks using differential evolution algorithm using regularization.

Table 4.3. Derived intrinsic and extrinsic parameters from the differential evolution-optimized performance model for the three deep learning frameworks. Parameters are given as the mean and standard deviation over ten fits. Here a and p represent coefficients and powers respectively of a term representing an intrinsic parameter, where as q is power in a multiplicative term representing an extrinsic (scaling) parameter.

	Mxnet		**PyTorch**		**TensorFlow**	
Intrinsic parameters	a	p	a	p	a	p
Filter size	6.27 ± 0.59	0.36 ± 0.01	6.07 ± 1.59	0.89 ± 0.05	8.39 ± 0.37	0.77 ± 0.01
Kernel size	4.44 ± 0.65	0.50 ± 0.04	4.84 ± 1.90	2.02 ± 0.24	6.59 ± 0.29	2.04 ± 0.03
Pool size	4.69 ± 0.33	0.52 ± 0.03	3.23 ± 0.83	1.55 ± 0.45	6.70 ± 0.67	1.98 ± 0.05
Learning rate	3.62 ± 0.41	-0.04 ± 0.003	3.75 ± 1.70	-0.27 ± 0.02	4.40 ± 0.60	-0.22 ± 0.007
Stride	4.51 ± 0.40	-0.99 ± 0.11	2.92 ± 1.54	-0.83 ± 1.42	4.13 ± 0.59	2.46 ± 0.10
Dropout probability	4.20 ± 0.67	-0.35 ± 0.05	35.92 ± 1.15	-5.00 ± 0.00	4.46 ± 0.43	-1.94 ± 0.07
Same	2.66 ± 0.51	-	2.08 ± 0.84	-	1.90 ± 0.58	-
Valid	1.50 ± 0.43	-	-0.57 ± 1.56	-	0.49 ± 0.71	-
Sigmoid	2.18 ± 0.45	-	2.32 ± 1.15	-	1.41 ± 0.41	-
Relu	1.52 ± 0.33	-	3.21 ± 1.35	-	1.85 ± 0.64	-
Tanh	2.29 ± 0.39	-	2.93 ± 1.93	-	1.99 ± 0.71	-
MNIST	5.48 ± 0.52	-	3.37 ± 1.35	-	1.99 ± 0.72	-
Fashion-MNIST	7.73 ± 0.36	-	3.42 ± 1.56	-	2.28 ± 0.72	-
CIFAR-10	1.00 ± 0.02	-	1.89 ± 1.00	-	1.63 ± 0.69	-
SGD	2.31 ± 0.36	-	2.16 ± 1.00	-	1.73 ± 0.46	-
Adam	1.78 ± 0.41	-	3.42 ± 1.45	-	2.01 ± 0.85	-
Extrinsic parameters	q		q		q	
Batchsize	-0.87 ± 0.005		-1.00 ± 0.007		-1.19 ± 0.01	
No. of GPUs	-1.07 ± 0.007		-1.01 ± 0.004		-0.74 ± 0.005	
Constant term	C		C		C	
	3.45± 0.024		1.03 ± 0.07		12.62 ± 0.05	

4.3.3 Comparison of the Proposed Performance Model with Machine Learning Models

The proposed model has been compared with two standard black box models, i.e., random forest regressor and support vector regressor. Generally, the random forest regressor has better prediction accuracy due to its ensemble learning shown in Figures 4.9, 4.10, and 4.11. The result shows a good linear fit compared to the differential evolution algorithm. However, the drawback of the random forest regressor is that it cannot give any insights into its internal working mechanism. Support vector regressor is a non-parametric technique that uses kernel functionality to model complex relationships. It can be effective in high-dimensional spaces. Support vector regressor predicted and measured times are shown in Figures 4.12, 4.13, and 4.14. Support vector regressor excels in handling intricate and non-linear patterns in the data, leading to potentially complex models that

require longer computation times during prediction. Additionally, the use of kernel functions in Support vector regressor, which maps the data to higher-dimensional spaces, can contribute to increased computation time, especially in high-dimensional datasets. However, the result shows a poor fit for all the deep learning frameworks compared with the random forest regressor and differential evolution algorithm. Furthermore, the sensitivity of SVR to outliers in the data might cause it to adapt to these extreme instances, resulting in longer prediction times for certain data points. Note that the performance of the proposed model is slightly inferior to random forest. However, the proposed model can provide insights into the internal behaviour and scalability, which are impossible with a black box model such as random forest.

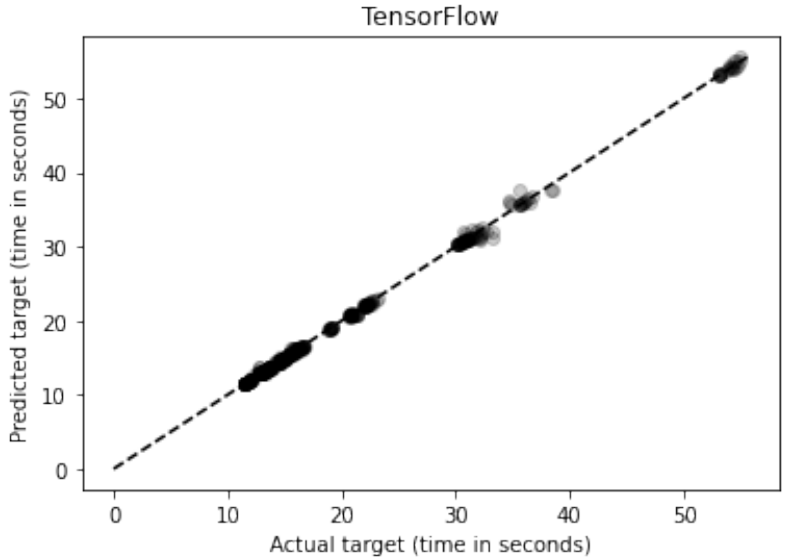


Figure 4.9. Random forest regressor predicted and measured times in TensorFlow deep learning frameworks using differential evolution algorithm.

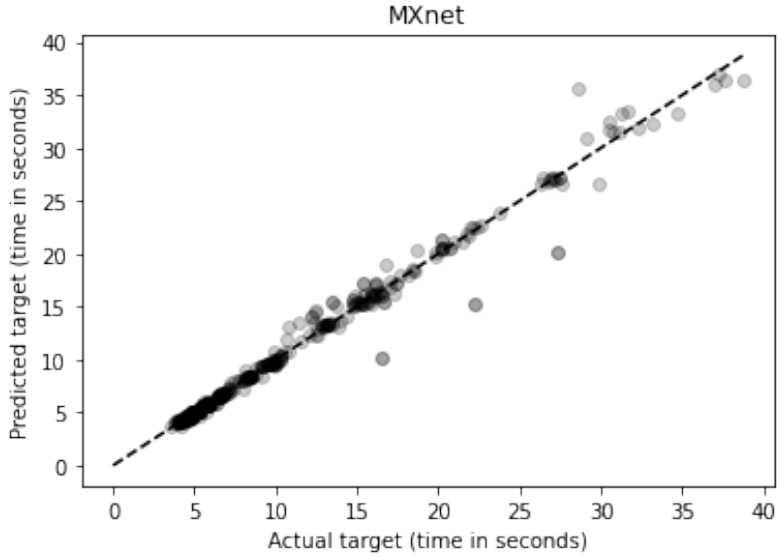


Figure 4.10. Random forest regressor predicted and measured times in MXNet deep learning frameworks using differential evolution algorithm.

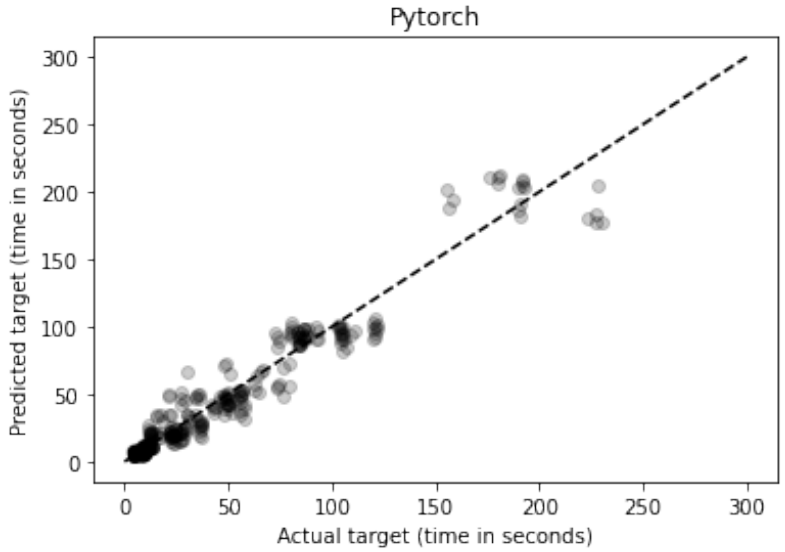


Figure 4.11. Random forest regressor predicted and measured times in PyTorch deep learning frameworks using differential evolution algorithm.

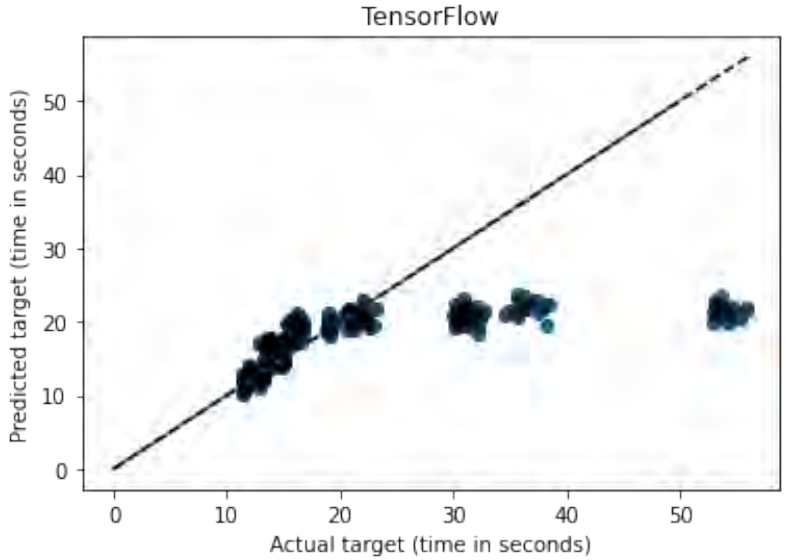


Figure 4.12. Support vector regressor predicted and measured times in TensorFlow deep learning framework.

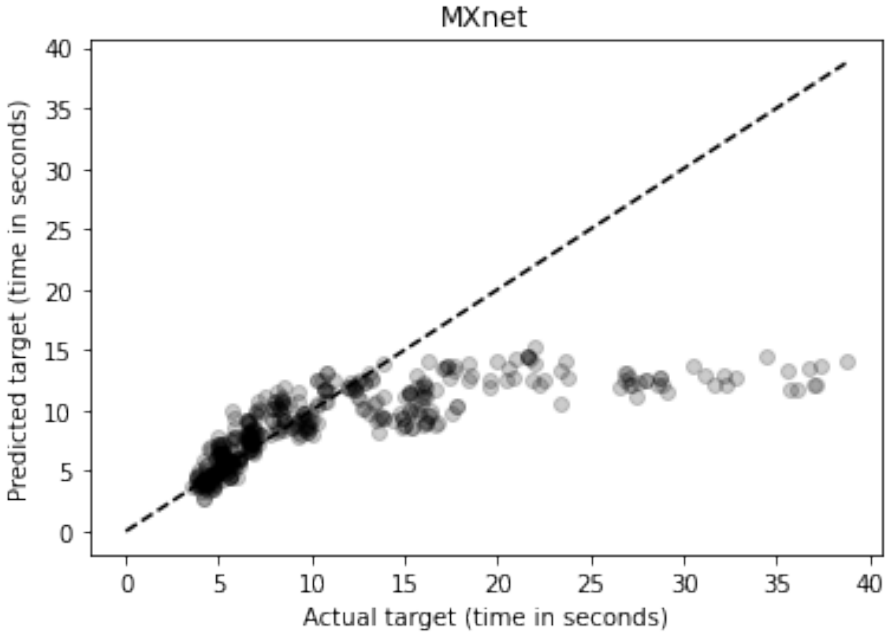


Figure 4.13. Support vector regressor predicted and measured times in MXNet deep learning framework.

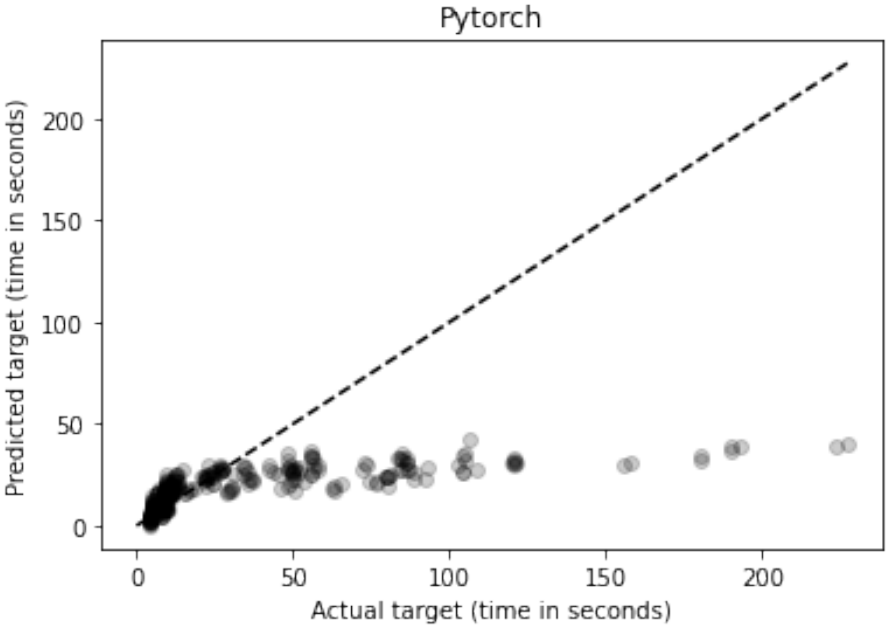


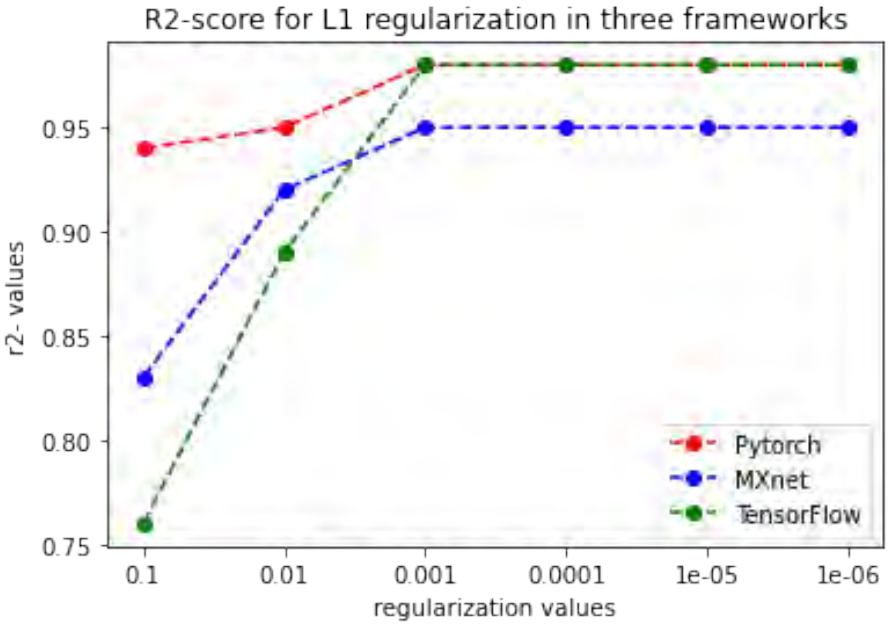
Figure 4.14. Support vector regressor predicted and measured times in PyTorch deep learning framework.

4.3.4 Evaluation of Regularization

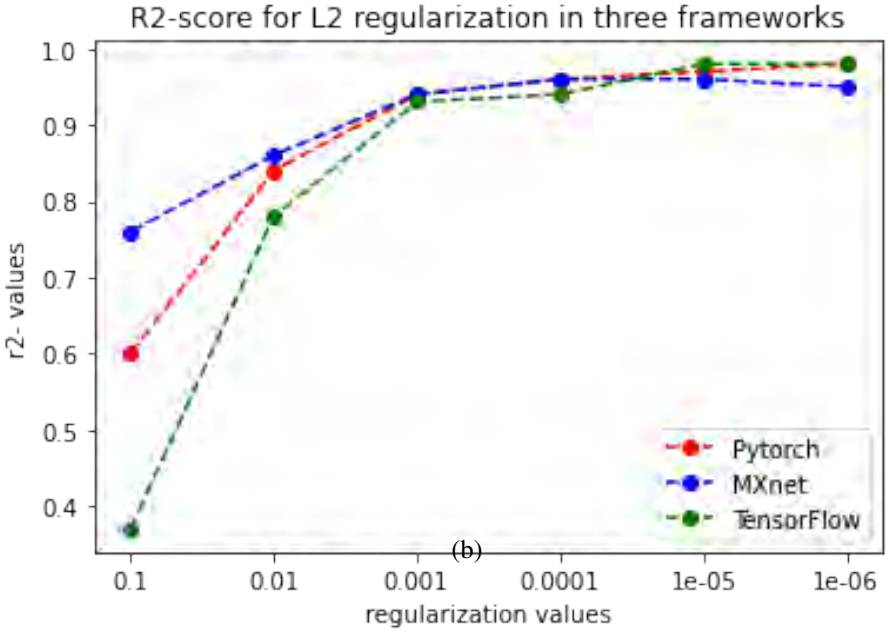
Regularization was applied to the cost function of the proposed performance model to optimize the vector constants and mitigate high variance in the intrinsic parameters of three deep learning frameworks. Specifically, both L1 and L2 regularization were employed, and their results were compared. The MAPE, MSE, and RMSE results favored L2 regularization, as depicted in Table 4.4. Consequently, L2 regularization was considered appropriate for the performance model. Various regularization parameter values were applied in logarithmic scale for L1 and L2 to identify the optimal value of the λ parameter. Figures 4.15 (a) and 4.15 (b) indicated a decline in the R2 score when the λ value exceeded 0.001. Notably, the model demonstrated robust fits at λ = 0.001 and offered consistent performance for the constant coefficients, representing the relative importance of the processes controlled by categorical parameters. Additionally, in Table 4.3, consistent performance was observed for the constant coefficients. The model coefficients plotted against the regularization parameter are depicted in Figures 4.16 (a) and (b). Constant coefficients of intrinsic parameters were plotted in Figure 4.16 (a), while power coefficients of intrinsic parameters were plotted in Figure 4.16 (b). Furthermore, coefficients of categorical intrinsic parameters were presented in Figure 4.17 (a), and powers of extrinsic parameters were shown in Figure 4.17 (b). Based on these findings, it was concluded that L2 regularization with a lambda value of 0.001 proved to be the most effective regularization strategy for the performance model.

Table 4.4. L1 and L2 regularization results in terms of Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE)

	L1 Regularization			L2 Regularization		
	MXnet	PyTorch	TF	MXnet	PyTorch	TF
MAPE	8%	29%	13%	7%	27%	10%
MSE	105.74	450.93	220.16	103.37	443.35	201.81
RMSE	10.28	17.52	14.83	10.16	17.02	14.20



(a)



(b)

Figure 4.15. Effect of regularization. (a) R2 values with different regularization values in three different frameworks using L1 regularization and (b) R2 values with different regularization values in three different frameworks using L2 regularization.

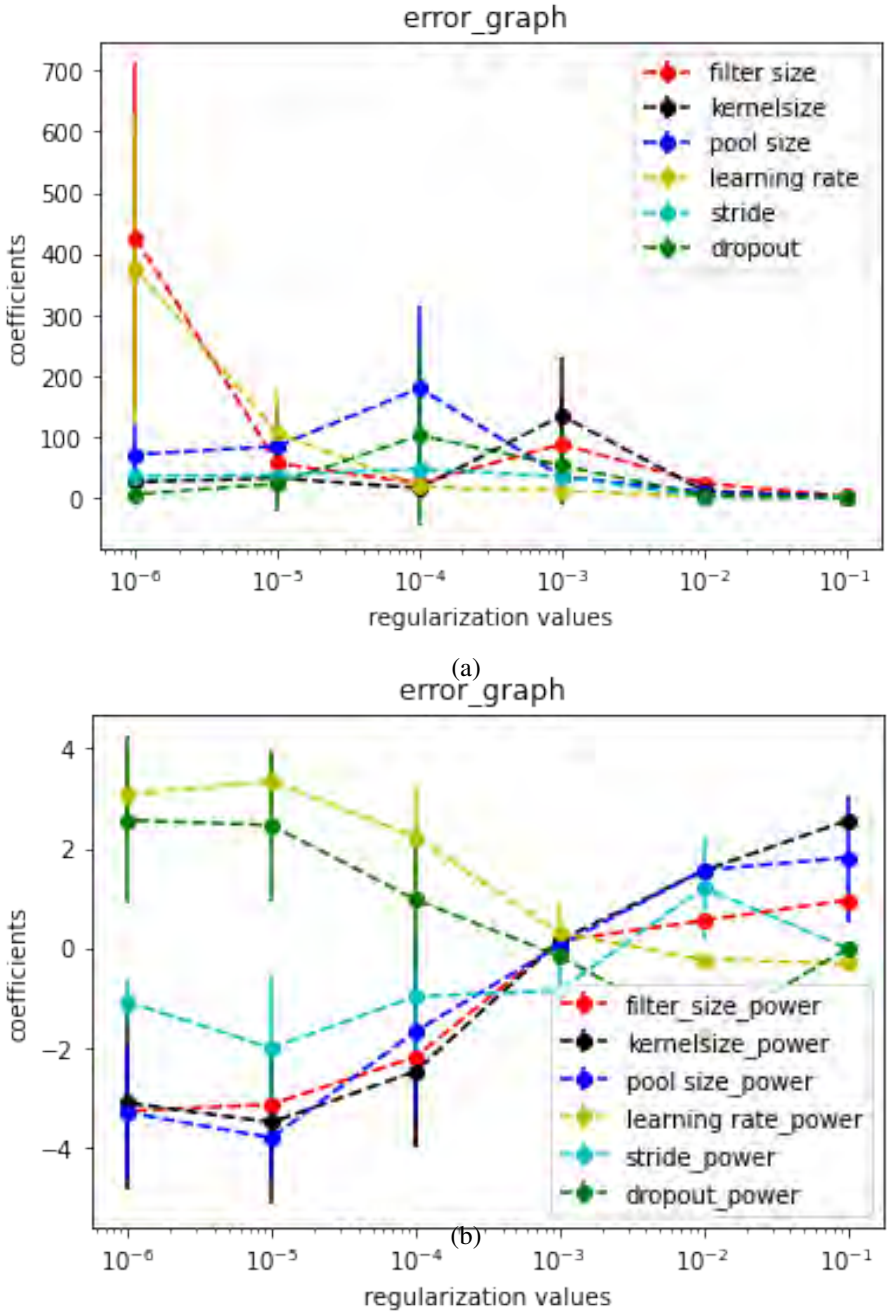


Figure 4.16. Effect of regularization, with model coefficients plotted against regularization parameter. Constant coefficients of intrinsic parameters are plotted in (a), the power coefficients of intrinsic parameters are shown in (b)

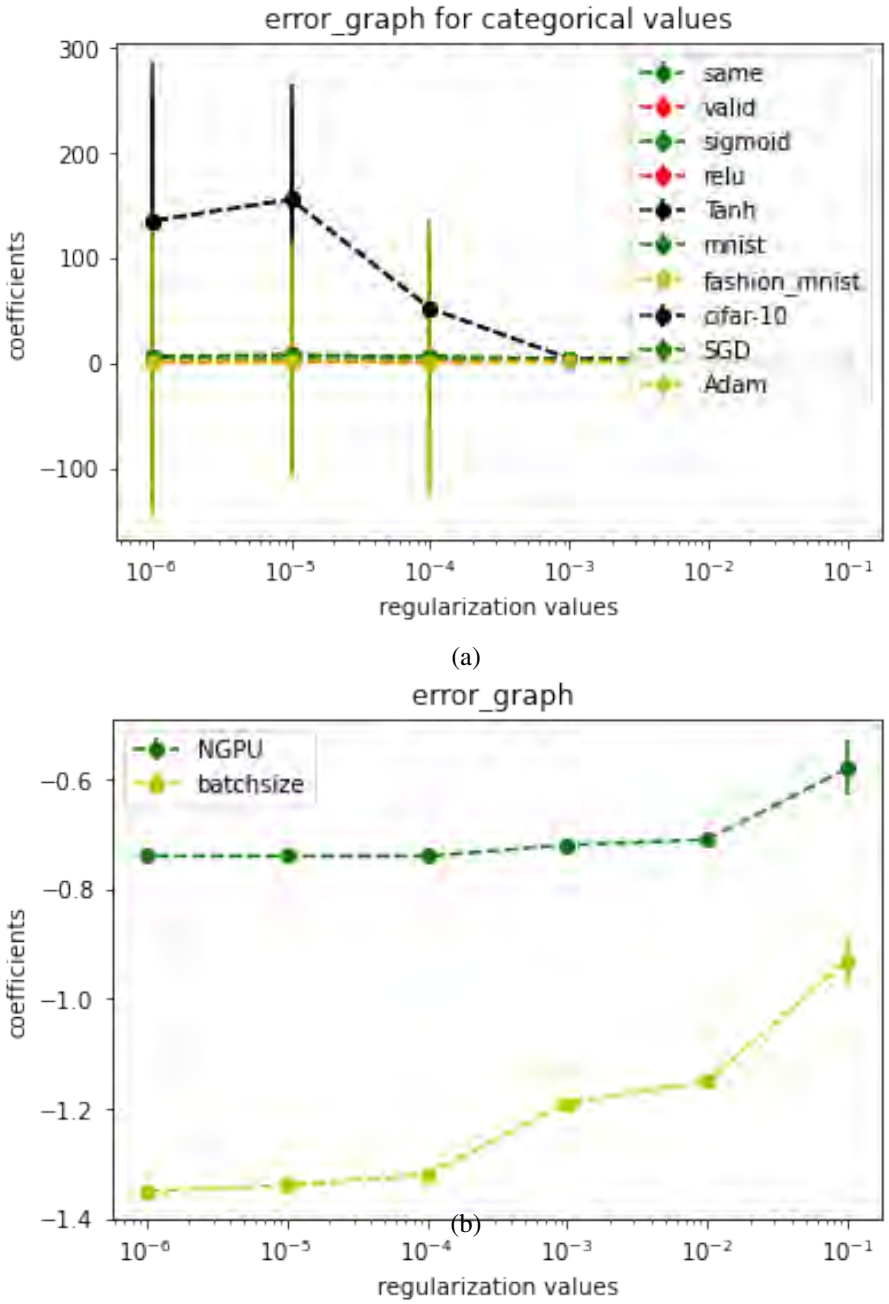


Figure 4.17. Effect of regularization, with model coefficients plotted against regularization parameter. coefficients of categorical intrinsic parameters in (a) and with powers of extrinsic parameters in (b).

4.3.5 Scalability Analysis for the Regularized model

The scaling power values in the table represent how the execution time changes with the number of GPUs as shown in Table 4.3. Here, q is power in a multiplicative term representing an extrinsic parameter. The extrinsic parameter coefficients are consistent in the proposed performance model with and without regularization. As shown in Table 4.5, -1 indicates ideal scaling, in which case the time is inversely proportional to the number of GPUs. The coefficients in PyTorch and MXnet framework show better scaling performance than TensorFlow. In TensorFlow, the value -0.73 is less than -1, which indicates sub-optimal scaling.

Table 4.5. nGPUs scaling power in various frameworks. Here nGPUs represent number of GPUs.

Frameworks	**nGPUs scaling power**
TensorFlow	-0.73
MXnet	-1.01
PyTorch	-1.02

4.4 Summary

In this work, a generic performance model for deep learning applications in a distributed environment was developed. The model takes into account both intrinsic and extrinsic factors that affect performance and scalability. It is formulated as a global optimization problem that utilizes regularization on a cost function and employs the differential evolution algorithm to find the best-fit values of the constants in the generic expression.

The proposed model was evaluated on three popular deep learning frameworks, namely TensorFlow, MXnet, and PyTorch, demonstrating accurate performance predictions and interpretability. Additionally, experimental results indicated that MXnet and PyTorch exhibited better scalability performance than TensorFlow.

Moreover, the proposed method, when coupled with regularization, successfully optimized the vector constants and reduced high variance in intrinsic parameters. Notably, the model can be applied to any distributed deep learning framework without requiring

changes to the code, and it provides insights into the factors influencing deep learning application performance and scalability.

Chapter 5

Case Study: Performance Analysis of a 3D-ResAttNet Model for Alzheimer's Diagnosis from 3D MRI Images

Building upon the previous chapter, this section aims to apply the proposed performance model to a realistic use case: a large 3D Explainable Residual Self-Attention Convolutional Neural Network (3D-ResAttNet) model trained on the Alzheimer's Disease Neuroimaging Initiative (ADNI) dataset (http://adni.loni.usc.edu). The primary objective involves evaluating the effectiveness and accuracy of the model in predicting the execution time of deep learning frameworks. Additionally, this chapter expands the analysis to a specific deep learning model, namely the 3D-ResAttNet architecture, implemented using the PyTorch framework within a multi-GPU environment. By harnessing the strengths of PyTorch, including its dynamic computation graph, user-friendly syntax, and robust community support, this study endeavors to provide a comprehensive assessment of the performance model's suitability for analysing a large production model in the realm of medical imaging.

Alzheimer's disease is a prevalent neurodegenerative disorder that presents significant challenges in accurate diagnosis. Early detection is crucial for timely intervention and effective management. However, the current diagnostic process lacks definitive tests, resulting in interpretation variability and delays. Therefore, there is a pressing need for reliable and efficient methods to improve early detection and accurate diagnosis of Alzheimer's disease.

Structural magnetic resonance imaging (MRI) has emerged as a valuable tool for analyzing brain structure and detecting atrophy patterns associated with Alzheimer's disease. By examining changes in brain morphology and volume, structural MRI enables the localization of regions of atrophy and improves diagnostic accuracy. This imaging modality offers insights into the underlying pathological changes in Alzheimer's disease, aiding in the understanding of disease progression.

However, traditional methods for atrophy localization and Alzheimer's disease diagnosis have limitations in terms of accuracy and interpretability. To address these limitations, advanced computational techniques are required. Xin Zhang *et al.* [37] proposed a novel 3D residual self-attention deep neural network architecture specifically designed for joint atrophy localization and Alzheimer's disease diagnosis using structural MRI data in their study. The paper introduces a model that overcomes the limitations of existing approaches and offers the potential for accurate localization of atrophy regions and interpretable diagnostic outcomes.

The proposed model by Xin Zhang is of significant importance as it addresses the limitations of existing approaches in Alzheimer's disease diagnosis. It offers the potential for accurate localization of atrophy regions and provides interpretable results, thereby aiding in the diagnosis and understanding of the disease. By implementing this model on the Alzheimer's disease dataset, the objective of the research is to evaluate the performance model's effectiveness on a modern research issue of representative size and complexity.

5.1 Performance Model

The formulation of performance model has been explained in the previous chapter 4 in the section 4.1.

The proposed performance model is as follows:

$t(I, E, x) = \left(\sum_{i=1}^{n} a_i I_i^{p_i}\right) \prod_{j=1}^{m} E_j^{q_j} + C$

Here $x = \left\{a_1, ..., \ a_{n_I}, p_1, ... p_{n_I}, q_1, ... q_{n_E}, c\right\} \in \mathbb{R}^M$ is a vector formed by combining a, p, q and coefficient C.

5.1.1 System Configuration

Implemented the experiments on a single node containing three GEFORCE RTX 2080 GPUs, each with 2.60 GHz speed and 16 GB GPU RAM, to study the performance of the generic model using PyTorch framework. The node also consists of a 2.81GHz speed CPU machine, 25Gbps network bandwidth and a CUDA-10.2 with Linux operating system. Furthermore, the node consists of various software configuration/ installation include PyTorch 1.2.0, Torchvision 0.4.0, Python 3.6.

5.1.2 Dataset and Model

Evaluate the proposed performance model using the 3D-ResAttNet-18 model trained on the Alzheimer's Disease Neuroimaging Initiative (ADNI) dataset in a multi-GPU system. The dataset includes MRI scans from 1407 subjects across ADNI-1, ADNI-2, and ADNI-3 datasets. These subjects were categorized into three groups - Alzheimer's Disease (AD), Mild Cognitive Impairment (MCI), and Normal Control (NC) - based on standard clinical criteria, such as Mini-Mental State Examination (MMSE) scores and Clinical Dementia Rating (CDR). MCI subjects were further divided into two subgroups, progressive MCI (pMCI) and stable MCI (sMCI), for the prediction of MCI conversion. pMCI represents subjects who progressed to AD within 36 months of their baseline visit, while sMCI comprises subjects who remained diagnosed with MCI. The 3D-ResAttNet model was trained using the ADNI-1 dataset, which contains 431 scans and served as the primary data source for training in this study.

Additionally, Figure 5.1 illustrates the architecture of the 3D-ResAttNet-18 model used in the evaluation [37]. A 3D explainable residual attention network (3D ResAttNet), a deep convolutional neural network incorporating self-attention residual mechanism and explainable gradient-based localization class activation mapping (GradCAM) for AD diagnosis (see Figure 5.1). The design rationale includes: 1) Efficient training and performance enhancement with the residual mechanism, addressing gradient-related challenges and preserving global features; 2) Learning long-range dependencies with the self-attention mechanism to capture crucial information effectively; 3) Utilizing gradient-based localization class activation mapping (GradCAM) to provide visual explanations of Alzheimer's

disease predictions.

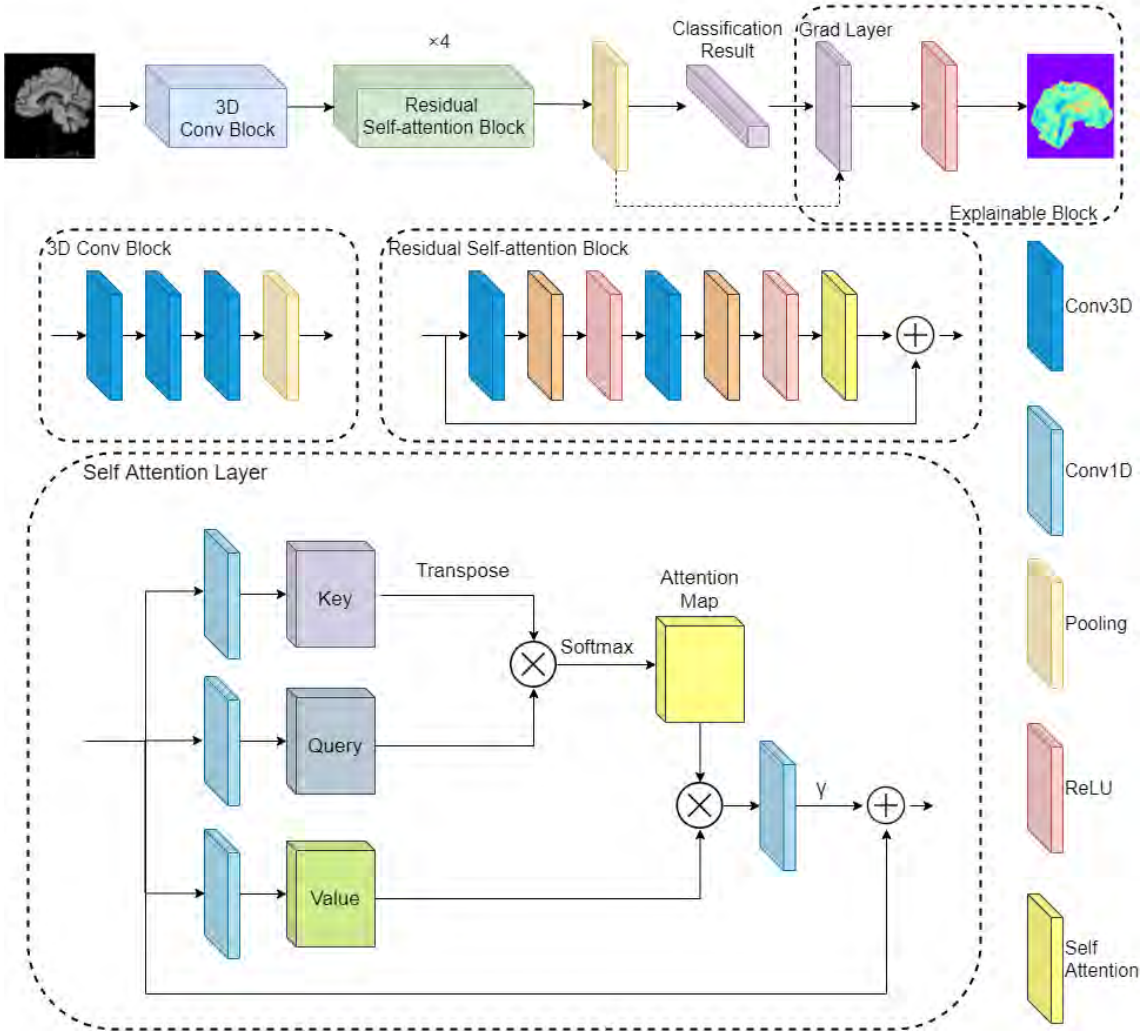


Figure 5.1. The architecture of 3D residual attention deep Neural Network. Image source [37]

To train the 3D-ResAttNet-18 model, utilize the ADNI-1 dataset as the primary source of data, which includes 431 scans . The ADNI-1 dataset comprises T1-weighted MR images acquired using both 1.5T and 3T scanners. The dataset used in the study includes magnetic resonance (MR) images acquired using 1.5 Tesla (1.5T) and 3 Tesla (3T) scanners. T1-weighted imaging is a specific imaging technique that provides high-resolution anatomical information of the brain. The 1.5T and 3T refer to the magnetic field strengths of the scanners used to acquire the images. The original dataset is in Neuroimaging Informatics Technology Initiative (NIfTI) format, undergoes preprocessing to correct spatial distortions caused by gradient nonlinearity and B1 field inhomogeneity. This involves AC-PC (Anterior Commissure - Posterior Commissure) correction, intensity correction [135], and skull stripping [136]. MIPAV(Medical Image Processing, Analysis, and Visualization) is used for AC-PC correction, while FSL (FMRIB Software Library v6.0) is employed for skull stripping. The sMRIs are aligned with the Colin27 template using a line align registration strategy in FSL, eliminating global linear differences and achieving consistent spatial resolution. These preprocessing steps ensure accurate analysis of the brain images as shown in Figure 5.2.

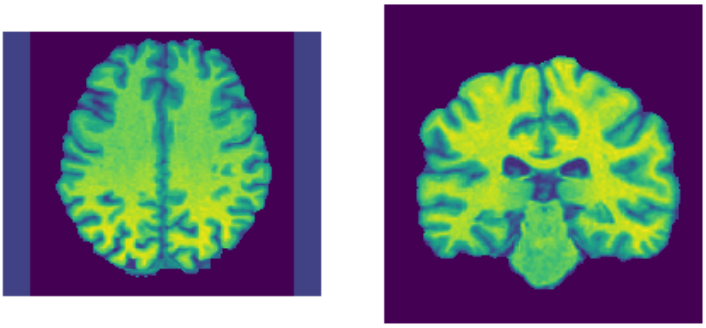


Figure 5.2. The first image is normal control and second image has Alzheimer's disease.

The overall utilization of the 3D-ResAttNet-18 architecture trained on the ADNI dataset provides a robust framework for the evaluation, allowing us to assess the performance model's effectiveness in predicting execution time for deep learning frameworks on complex medical image data.

5.1.3 Performance Metrics

The scalability and Mean Absolute Percentage Error (MAPE) are selected as performance metrics for run-time evaluation on PyTorch deep learning framework. The scalability is measured in the powers of external parameters as shown in chapter 4, section 4.1, equation (4.3). The MAPE metric employed to evaluate the accuracy of this fit and the overall quality of the performance model as shown in equation (4.13).

5.1.4 Experiments

The experiment is designed to accomplish the following two goals:

1. Performance evaluation of the proposed performance model on the 3D-ResAttNet architecture implemented with PyTorch deep learning framework.
2. Comparison between the proposed model and black box machine learning model.

For the first evaluation, performed the distributed training of 3D-ResAttNet on Alzheimer's dataset using PyTorch deep learning framework. The experimental training parameters are created by applying random sampling on a set of intrinsic and extrinsic parameters and its corresponding average training time taken by a 3D-ResAttNet architecture per iteration. Table 5.1 shows intrinsic and extrinsic parameters and their possible values. The intrinsic parameters are the model's hyperparameters, including kernel size, pooling size, activation function, etc. The number of GPUs and the batch size are extrinsic factors since these affect the scaling over multiple processes. The experiments involve several trials where the time for a single training iteration measured using randomly selected values of the intrinsic and extrinsic parameters. 1500 trials were conducted to prepare a dataset of 1500 data samples. The experimental data for 900 trials are used to fit the performance model or train the standard black box models for comparison. The remaining 600 are used to evaluate the test and validation models. For the second evaluation, the experimental parameters are used to build performance evaluation model, random forest regressor algorithm model. The performance of these models and their corresponding results are explained in the subsequent subsections.

Table 5.1. Parameters of the performance model, with ranges of values sampled in the experiments.

Index	Name	Set of possible values considered
Intrinsic parameters		
1	Kernel size	{2,3,4,5}
2	Pooling size	{1}
3	Activation function	{Relu, Tanh, Sigmoid}
4	Optimizer	{Adam, SGD}
5	Image_dataset_name	{Alzheimers dataset}
6	Number of filters	{16,32,64}
7	Learning rate	{0.1,0.01,0.001,10^{-4}, 10^{-5}, 10^{-6}}
8	Padding_mode	{valid, same}
9	Stride	{2}
Extrinsic parameters		
11	Number of GPUs	{1,2,3}
12	Batch size	{1,2,3,4}

5.2 Results and Analysis

This section shows the results of the proposed performance model using PyTorch framework and compared with standard machine learning algorithm, random forest regressor. Table 5.2 shows internal parameters and scalability in PyTorch framework. Table 5.3 shows mean absolute error values on predictions of the performance models, respectively.

5.2.1 Performance Evaluation of the Proposed Performance Model on the 3D-ResAttNet Architecture Implemented with PyTorch Deep Learning Framework

Applied the proposed performance model using regularization to the 3D-ResAttNet architecture implemented with PyTorch deep learning framework. The actual execution times for training the model using the deep learning framework were recorded and predicted execution times also generated. Figure 5.3 shows the scatter graph of the predicted execution times from the proposed model plotted against the actual execution time. The linear fit to the straight line determines how well the model can predict unseen configurations. The best fit constant coefficients for all frameworks are shown in Table 5.2.

Table 5.2. Derived intrinsic and extrinsic parameters from the differential evolution-optimized performance models for the three deep learning frameworks. Parameters are given as the mean and standard deviation over ten fits. a and p represent coefficients and powers, respectively, of a term representing an intrinsic parameter, whereas q is power in a multiplicative term representing an extrinsic (scaling) parameter.

	PyTorch	
Intrinsic parameters	a	p
Filter size	75.65 ± 7.38	0.21 ± 0.01
Kernel size	68.24 ± 4.32	0.32 ± 0.02
Learning rate	60.69 ± 5.40	-0.04 ± 0.002
Sigmoid	27.74 ± 5.00	-
Relu	15.56 ± 3.61	-
Tanh	19.96 ± 3.66	-
SGD	30.92 ± 4.12	-
Adam	36.30 ± 3.73	-
Extrinsic parameters	q	
Batchsize	-0.03 ± 0.002	
No. of GPUs	-0.94 ± 0.02	
Constant term	C	
	75.87± 6.35	

The results show stable and consistent fits for the extrinsic parameters and the additive constant C, suggesting that the scalability results are accurate. The higher variance in the intrinsic parameters are reduced by using regularization. From Table 5.2, it becomes apparent that the model consistently demonstrates internal consistency, with relatively small variances. Because the differences between the categorical parameters are of the order of (or) greater than the standard deviation. For example, for the activation function coefficients, which is categorical with three possible values *Relu* and *Tanh* and *Sigmoid* parameters. *Sigmoid* takes more time than *Relu* and *Tanh* mode. In terms of optimizers, Adam has a large constant and takes more time than SGD. Insights about the time complexity of the internal processes can be gained by considering the powers in the generic expression. Notably, the learning rate exhibits a power value that is remarkably close to zero, indicating that the learning rate does not significantly impact the overall performance.

Additionally, the extrinsic parameter considered is batch size. In terms of batch size, the power is close to zero, indicating that batch size has a minimal effect on performance. This finding contradicts the previous chapters where batch size was considered important. Consequently, smaller batch sizes can be utilized, which typically enhance classification

accuracy, without compromising training time performance. The consistent and stable fits observed for the extrinsic parameters and the additive constant C validate the accuracy of the scalability results obtained from the differential evolution optimization process.

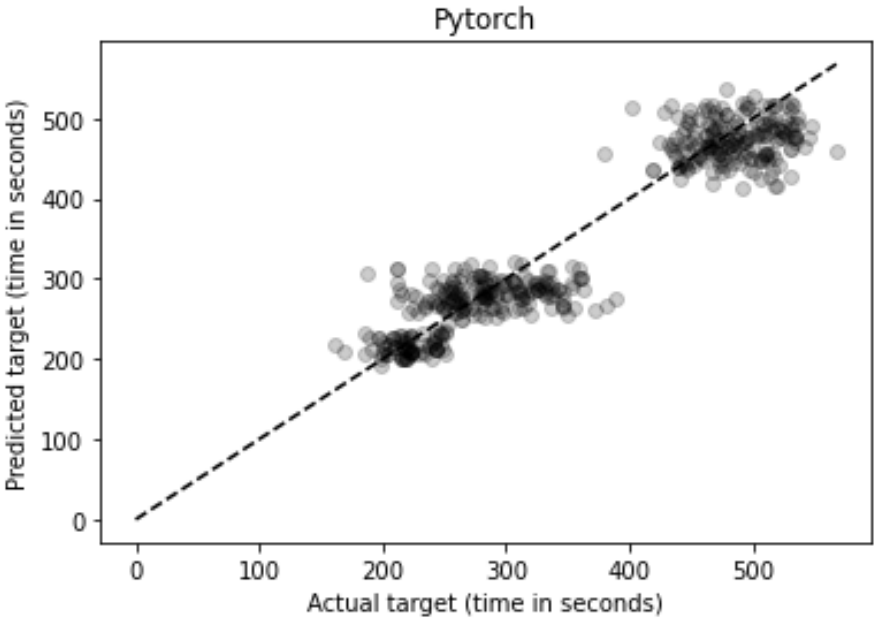


Figure 5.3. The proposed performance model predicted and measured times in PyTorch deep learning frameworks using differential evolution algorithm.

5.2.2 Comparison of the Proposed Performance Model with Random Forest

Compared the proposed model with the standard algorithm in machine learning, random forest regressor. Generally, the random forest regressor has better prediction accuracy due to its ensemble learning shown in Figure 5.4. The result shows a good linear fit compared to the differential evolution algorithm. However, the drawback of the random forest regressor is that it cannot give any insights into its internal working mechanism. Evaluate the fits using the mean absolute percentage error between predicted execution time and actual times, as shown in Table 5.3. Note that the performance of the proposed model is exactly the same as that of random forest regressor. However, the proposed model can provide insights into the internal behaviour and scalability, which are impossible with a black box model such as random forest.

Table 5.3. Mean absolute percentage error on predictions of the performance models on the 300 instances in the evaluation dataset in seconds.

	PyTorch
Differential evolution	0.07
Random forest	0.07

Table 5.4. nGPUs scaling power in various frameworks. Here nGPUs represent number of GPUs.

Framework	**nGPUs scaling power**
PyTorch	-0.94 ± 0.02

By observing the coefficients, the PyTorch framework show better scaling performance. As shown in Table 5.4, -0.94 indicates close to ideal scaling, in which case the time is inversely proportional to the number of GPUs.

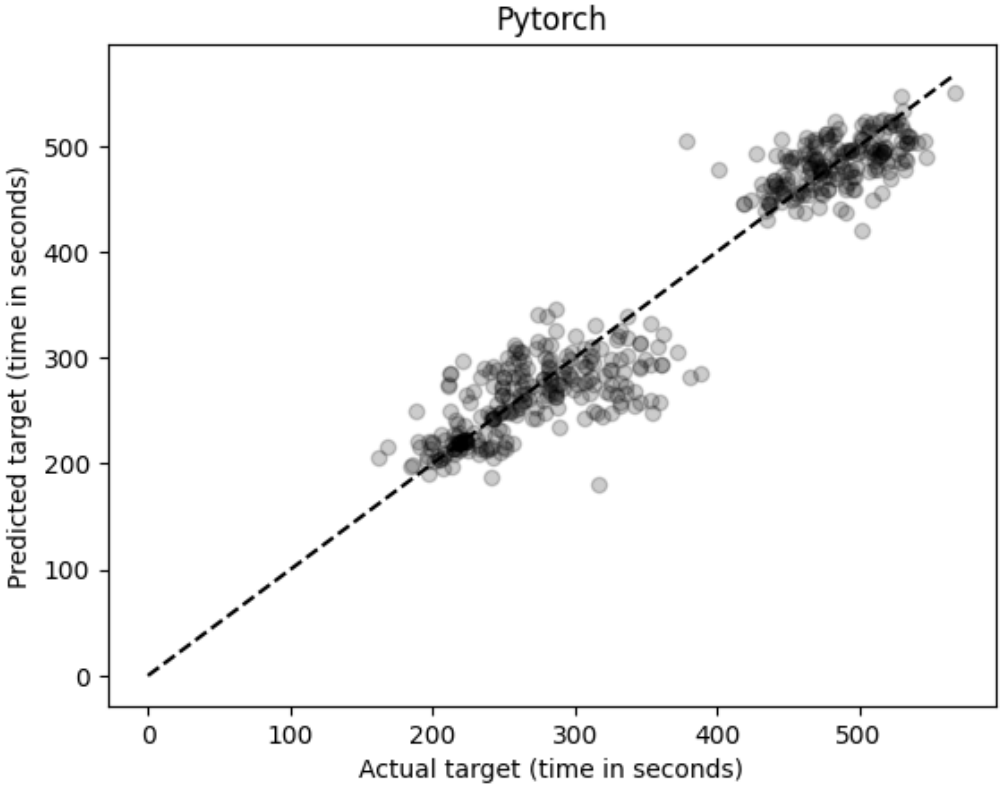


Figure 5.4. Random forest regressor predicted and measured times in PyTorch deep learning frameworks using differential evolution algorithm.

5.3 Summary and Discussion

Evaluated the proposed work by analyzing the intrinsic parameters' performance and scalability of the popular deep learning framework PyTorch, by training a 3D-ResAttNet architecture on the popular Alzheimer's dataset in a multi-GPU environment. The experimental results show that the proposed method can be applied to a distributed deep learning framework without instrumenting the code. Argue that the proposed performance model shows promise as a generic tool which balances the accuracy of prediction with insight into the underlying system. The scalability behavior is well captured by the model; however, there is good stability in the fits to the internal parameters, which describes insights of the internal processes.

In the study, the performance model was applied to a realistic use case involving a large dataset of 3D images. These 3D images pose a challenge for deep learning models due to their memory-intensive nature. However, the performance model proved valuable in gaining insights into the model's performance characteristics. One significant finding is the accurate determination of scalability in relation to GPU count, which exhibited good scalability in the case. Additionally, discovered that batch size has minimal impact on scalability, providing practitioners with an important insight for optimizing predictive accuracy and runtime performance.

It is worth noting that both the proposed model and the random forest regressor model showed higher errors compared to the models in previous chapters. This suggests the presence of increased noise and variance in the data, possibly resulting from the influence of large images on performance. Factors like cache performance may become more critical with such large datasets, leading to unpredictable variations in training time that cannot be solely attributed to hyperparameters. Despite these challenges, predictions were achieved with a mean absolute percentage error of 7%, highlighting the effectiveness of the approach. Furthermore, the analysis yielded valuable insights into the impact of hyperparameters on performance, reinforcing the robustness and applicability of the performance model in understanding and optimizing model behavior.

Chapter 6

Conclusion And Future Work

Performance modelling for scalable deep learning is very important to quantify the efficiency of large parallel workloads. Performance models are used to obtain run-time estimates by modelling various aspects of an application on a target system. Designing performance models requires comprehensive analysis in order to build accurate models. Limitations of current performance models include poor explainability and limited applicability to particular architectures. Existing performance models in deep learning have been proposed, which are broadly categorised into two methodologies: analytical modelling and empirical modelling. Analytical modelling utilizes a transparent approach that involves converting the internal mechanisms of the model or applications into a mathematical model that corresponds to the goals of the system. Empirical modelling predicts outcomes based on observation and experimentation, characterizes algorithm performance using sample data, and is a good alternative to analytical modelling. However, both these approaches have limitations such as poor explainability and poor generalisation.

In this work, applied hybridization of the analytical and empirical approaches developed a novel generic performance model that provides a general expression in terms of a deep neural network framework in a distributed environment that gives accurate performance. The contributions can be summarized as follows:

1. A comprehensive literature review was conducted using a systematic methodology, and a performance model was built based on synchronous stochastic gradient descent (S-SGD) to analyze the execution time performance of deep learning frameworks in a multi-GPU environment. The model was evaluated using three deep learning models (Convolutional Neural Networks, Autoencoder, and Multilayer Perceptron), each

implemented in three frameworks (MXNet, Chainer, and TensorFlow) respectively. This initial study follows more closely the analytical approach. Factors influencing the performance of deep learning frameworks were analyzed by computing the running time of each framework in the proposed model, considering the load imbalance factor. The results have shown that MXNet and Chainer have better scalability compared to TensorFlow for all three models. Moreover, the analysis of the load imbalance factor has shown that load imbalancing is a contributing factor to scalability in distributed deep learning, and high load imbalance is strongly correlated with poor scalability in the experiments. However, the performance model could not provide deeper insights into where the load imbalance arises. This motivates the development of a more detailed performance model fitted to the performance data using a global optimization algorithm. The findings and methodology were subsequently published in [122].

2. A performance model was developed to quantify the efficiency of large parallel workloads. In this work, a generic performance model of an application in a distributed environment was proposed, with a generic expression of the application execution time that considers the influence of both intrinsic factors/operations (e.g. algorithmic parameters/internal operations) and extrinsic scaling factors (e.g. the number of processors, data chunks and batch size). The problem was framed as a global optimization task with and without using regularization, and a cost function and differential evolution algorithm were employed to determine the optimal values of the constants in the generic expression. The proposed model was evaluated on three deep learning frameworks (i.e., TensorFlow, MXnet, and PyTorch). The experimental results show that the proposed model can provide accurate performance predictions and interpretability. In addition, the proposed work could be applied to any distributed deep neural network without instrumenting the code and provides insight into the factors affecting performance and scalability.

   Applied the regularisation on the cost function to the proposed performance model to optimise the vector constants and reduce high variance in intrinsic parameters in three deep learning frameworks. Both kinds of regularizations were applied to the model, and the results of L1 and L2 regularizations were compared. It was found that L2 regularization is more appropriate for the performance model. L2 regularization

was considered appropriate for the performance model. Various regularization parameter values were applied in logarithmic scale to find a better value for λ. The R2 score was used for evaluation, and it was observed that the R2 score deteriorates when the λ value exceeds 0.001. When $\lambda = 0.001$, the model fits well, and it provides consistent performance for the constant coefficients, representing the relative importance of the process controlled by categorical parameters. Furthermore, the model gives consistent performance for the constant coefficients in all the frameworks. The performance model using regularization is a generalized model with optimized good fits in all the frameworks. Overall, the proposed performance model using regularization provides accurate and stable fits and represents the relative importance of different process parameters in distributed environments. Additionally, this work has been submitted to IEEE Access [137] (under review).

3. An experimental evaluation was conducted for the proposed generic performance model applied to a real-world application. The evaluation involved applying the performance model to a realistic use case that utilized a large dataset of 3D images, specifically the ADNI dataset. These 3D images present a challenge for deep learning models due to their memory-intensive nature. Nevertheless, the performance model proved to be valuable in gaining insights into the performance characteristics of the model. One significant finding from the evaluation is the accurate determination of scalability in relation to GPU count. The model exhibited good scalability as the number of GPUs was increased. Additionally, it was observed that batch size had minimal impact on scalability. This finding provides practitioners with an important insight for optimizing both predictive accuracy and runtime performance. Overall, the experimental evaluation of the proposed generic performance model in a real-world scenario involving 3D images has yielded valuable insights, particularly in terms of scalability and the impact of batch size.

6.1 Future Works

For future work, there are several avenues to explore based on the findings and limitations of the current research:

Further investigation of load imbalance: Although the initial study identified load imbalance as a contributing factor to scalability in distributed deep learning, the specific sources of load imbalance were not deeply explored. Future work can focus on analyzing and addressing load imbalance issues in more detail to gain deeper insights and improve scalability.

Refinement of the performance model: While the proposed performance model showed promise in predicting performance and providing interpretability, there is still a room for improvement. Future work can focus on refining the model by modifying the generic expression to allow for more complex interactions between the terms or exploring alternative optimization algorithms to enhance stability and accuracy of the fits. The regularization term added to the cost function treats all parameters in the same way, however, there are two types of parameters: powers and coefficients. In the future, investigate treating these terms differently could be worthwhile, as in some cases, such as the Alzheimer's model in which training times are of the order of hundreds of seconds, one would expect these parameters to have widely differing magnitudes. The powers should always remain of order unity, while the coefficients may be orders of magnitude larger. A more sophisticated scheme would allocate different values of the regularization constant to the two classes of parameters.

Extending the model to other applications and frameworks: The current research evaluated the performance model on three deep learning frameworks (TensorFlow, MXnet, and PyTorch). Future work can expand the scope by applying the model to other distributed deep neural network frameworks to assess its generalizability and effectiveness across different applications.

Integration of the performance model into practical systems: The developed performance model can be integrated into real-world systems to guide decision-making processes and optimize the performance of large parallel workloads. Future work can explore the practical implementation of the model in distributed deep learning environments and evaluate its impact on system performance and efficiency.

Exploring Symbolic Regression for Improved Generic Expressions: Investigating the use of symbolic regression to attempt to discover better generic expressions for use in the model. Symbolic regression is an emerging area of machine learning in which genetic

programming techniques are used to find the algebraic expressions which can accurately model a dataset. Observing the results of symbolic regression on performance data may lead to the design of expressions which better capture the underlying processes.

By addressing these areas in future research, it is possible to advance the field of performance modeling for scalable deep learning and contribute to the development of efficient and effective systems for large-scale parallel workloads.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[2] M. I. Jordan and T. M. Mitchell, "Machine learning: Trends, perspectives, and prospects," *Science*, vol. 349, no. 6245, pp. 255–260, 2015.

[3] M. Al-Qizwini, I. Barjasteh, H. Al-Qassab, and H. Radha, "Deep learning algorithm for autonomous driving using googlenet," in *2017 IEEE intelligent vehicles symposium (IV)*, IEEE, 2017, pp. 89–96.

[4] F. He, T. Liu, and D. Tao, "Why resnet works? residuals generalize," *IEEE transactions on neural networks and learning systems*, vol. 31, no. 12, pp. 5349–5362, 2020.

[5] U. Muhammad, W. Wang, S. P. Chattha, and S. Ali, "Pre-trained vggnet architecture for remote-sensing image scene classification," in *2018 24th International Conference on Pattern Recognition (ICPR)*, IEEE, 2018, pp. 1622–1627.

[6] N. Aloysius and M. Geetha, "A review on deep convolutional neural networks," in *2017 international conference on communication and signal processing (ICCSP)*, IEEE, 2017, pp. 0588–0592.

[7] S. Pllana, S. Benkner, F. Xhafa, and L. Barolli, "Hybrid performance modeling and prediction of large-scale computing systems," in *2008 International Conference on Complex, Intelligent and Software Intensive Systems*, IEEE, 2008, pp. 132–138.

[8] Z. Zhong, V. Rychkov, and A. Lastovetsky, "Data partitioning on heterogeneous multicore and multi-gpu systems using functional performance models of data-parallel applications," in *2012 IEEE international conference on cluster computing*, IEEE, 2012, pp. 191–199.

[9] C. Janiesch, P. Zschech, and K. Heinrich, "Machine learning and deep learning," *Electronic Markets*, vol. 31, no. 3, pp. 685–695, 2021.

[10] F. Yan, O. Ruwase, Y. He, and T. Chilimbi, "Performance modeling and scalability optimization of distributed deep learning systems," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2015, pp. 1355–1364.

[11] H. Kim, H. Nam, W. Jung, and J. Lee, "Performance analysis of cnn frameworks for gpus," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, 2017, pp. 55–64.

[12] H. Qi, E. R. Sparks, and A. Talwalkar, "Paleo: A performance model for deep neural networks," 2016.

[13] A. Castelló, M. Catalán, M. F. Dolz, J. I. Mestre, E. S. Quintana-Ortí, and J. Duato, "Performance modeling for distributed training of convolutional neural networks," in *2021 29th Euromicro international conference on parallel, distributed and network-based processing (PDP)*, IEEE, 2021, pp. 99–108.

[14] D. Jia, M. P. Saha, J. Bhimani, and N. Mi, "Performance and consistency analysis for distributed deep learning applications," in *2020 IEEE 39th International Performance Computing and Communications Conference (IPCCC)*, IEEE, 2020, pp. 1–8.

[15] M. Dungey*, R. Fry, B. González-Hermosillo, and V. L. Martin, "Empirical modelling of contagion: A review of methodologies," *Quantitative finance*, vol. 5, no. 1, pp. 9–24, 2005.

[16] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "Performance modeling for cnn inference accelerators on fpga," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 4, pp. 843–856, 2019.

[17] Z. Lin, X. Chen, H. Zhao, Y. Luan, Z. Yang, and Y. Dai, "A topology-aware performance prediction model for distributed deep learning on gpu clusters," in *2020 IEEE International Conference on Big Data (Big Data)*, IEEE, 2020, pp. 2795–2801.

[18] A. Viebke, S. Pllana, S. Memeti, and J. Kolodziej, "Performance modelling of deep learning on intel many integrated core architectures," in *2019 International Conference on High Performance Computing & Simulation (HPCS)*, IEEE, 2019, pp. 724–731.

[19] R. Rakshith, V. Lokur, P. Hongal, V. Janamatti, and S. Chickerur, "Performance analysis of distributed deep learning using horovod for image classification," in *2022 6th International Conference on Intelligent Computing and Control Systems (ICICCS)*, IEEE, 2022, pp. 1393–1398.

[20] J. J. Hopfield, "Artificial neural networks," *IEEE Circuits and Devices Magazine*, vol. 4, no. 5, pp. 3–10, 1988.

[21] J. Sresakoolchai and S. Kaewunruen, "Railway defect detection based on track geometry using supervised and unsupervised machine learning," *Structural health monitoring*, vol. 21, no. 4, pp. 1757–1767, 2022.

[22] Y. Huang, "Advances in artificial neural networks–methodological development and application," *Algorithms*, vol. 2, no. 3, pp. 973–1007, 2009.

[23] L. Alzubaidi, J. Zhang, A. J. Humaidi, A. Al-Dujaili, Y. Duan, O. Al-Shamma, J. Santamaría, M. A. Fadhel, M. Al-Amidie, and L. Farhan, "Review of deep learning: Concepts, cnn architectures, challenges, applications, future directions," *Journal of big Data*, vol. 8, pp. 1–74, 2021.

[24] Z. Wu, C. Shen, and A. Van Den Hengel, "Wider or deeper: Revisiting the resnet model for visual recognition," *Pattern Recognition*, vol. 90, pp. 119–133, 2019.

[25] P. Vincent, "A connection between score matching and denoising autoencoders," *Neural computation*, vol. 23, no. 7, pp. 1661–1674, 2011.

[26] I. H. Sarker, "Deep learning: A comprehensive overview on techniques, taxonomy, applications and research directions," *SN Computer Science*, vol. 2, no. 6, p. 420, 2021.

[27] M. Rocha, P. Cortez, and J. Neves, "Evolution of neural networks for classification and regression," *Neurocomputing*, vol. 70, no. 16-18, pp. 2809–2816, 2007.

[28] M. Gardner and S. Dorling, "Statistical surface ozone models: An improved methodology to account for non-linear behaviour," *Atmospheric Environment*, vol. 34, no. 1, pp. 21–34, 2000.

[29] B. Barak, B. Edelman, S. Goel, S. Kakade, E. Malach, and C. Zhang, "Hidden progress in deep learning: Sgd learns parities near the computational limit," *Advances in Neural Information Processing Systems*, vol. 35, pp. 21 750–21 764, 2022.

[30] L. O. Chua and T. Roska, "The cnn paradigm," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 40, no. 3, pp. 147–156, 1993.

[31] E. Maggiori, Y. Tarabalka, G. Charpiat, and P. Alliez, "Convolutional neural networks for large-scale remote-sensing image classification," *IEEE Transactions on geoscience and remote sensing*, vol. 55, no. 2, pp. 645–657, 2016.

[32] Y. Han, J. Kim, and K. Lee, "Deep convolutional neural networks for predominant instrument recognition in polyphonic music," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 25, no. 1, pp. 208–221, 2016.

[33] S. Shao, R. Yan, Y. Lu, P. Wang, and R. X. Gao, "Dcnn-based multi-signal induction motor fault diagnosis," *IEEE Transactions on Instrumentation and Measurement*, vol. 69, no. 6, pp. 2658–2669, 2019.

[34] W. Rawat and Z. Wang, "Deep convolutional neural networks for image classification: A comprehensive review," *Neural computation*, vol. 29, no. 9, pp. 2352–2449, 2017.

[35] F. Ramzan, M. U. G. Khan, A. Rehmat, S. Iqbal, T. Saba, A. Rehman, and Z. Mehmood, "A deep learning approach for automated diagnosis and multi-class classification of alzheimer's disease stages using resting-state fmri and residual neural networks," *Journal of medical systems*, vol. 44, pp. 1–16, 2020.

[36] T. Shanthi and R. Sabeenian, "Modified alexnet architecture for classification of diabetic retinopathy images," *Computers & Electrical Engineering*, vol. 76, pp. 56–64, 2019.

[37] X. Zhang, L. Han, W. Zhu, L. Sun, and D. Zhang, "An explainable 3d residual self-attention deep neural network for joint atrophy localization and alzheimer's disease diagnosis using structural mri," *IEEE journal of biomedical and health informatics*, vol. 26, no. 11, pp. 5289–5297, 2021.

[38] S. Alyamkin, M. Ardi, A. C. Berg, A. Brighton, B. Chen, Y. Chen, H.-P. Cheng, Z. Fan, C. Feng, B. Fu, *et al.*, "Low-power computer vision: Status, challenges, and opportunities," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 411–421, 2019.

[39] Y. Wang, H. Yao, and S. Zhao, "Auto-encoder based dimensionality reduction," *Neurocomputing*, vol. 184, pp. 232–242, 2016. [Online]. Available: `https://api.semanticscholar.org/CorpusID:207111259`.

[40] H. Marmolin, "Subjective mse measures," *IEEE transactions on systems, man, and cybernetics*, vol. 16, no. 3, pp. 486–489, 1986.

[41] U. Ruby and V. Yendapalli, "Binary cross entropy with deep learning technique for image classification," *Int. J. Adv. Trends Comput. Sci. Eng*, vol. 9, no. 10, 2020.

[42] D. Holden, J. Saito, T. Komura, and T. Joyce, "Learning motion manifolds with convolutional autoencoders," in *SIGGRAPH Asia 2015 technical briefs*, 2015, pp. 1–4.

[43] S. E. Otto and C. W. Rowley, "Linearly recurrent autoencoder networks for learning dynamics," *SIAM Journal on Applied Dynamical Systems*, vol. 18, no. 1, pp. 558–593, 2019.

[44] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol, "Extracting and composing robust features with denoising autoencoders," in *Proceedings of the 25th international conference on Machine learning*, 2008, pp. 1096–1103.

[45] J. An and S. Cho, "Variational autoencoder based anomaly detection using reconstruction probability," *Special lecture on IE*, vol. 2, no. 1, pp. 1–18, 2015.

[46] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–43, 2019.

[47] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, Ieee, 2009, pp. 248–255.

[48] H. Zhang, Y. Li, Z. Deng, X. Liang, L. Carin, and E. Xing, "Autosync: Learning to synchronize for data-parallel distributed deep learning," *Advances in Neural Information Processing Systems*, vol. 33, pp. 906–917, 2020.

[49] C. Jia, J. Liu, X. Jin, H. Lin, H. An, W. Han, Z. Wu, and M. Chi, "Improving the performance of distributed tensorflow with rdma," *International Journal of Parallel Programming*, vol. 46, pp. 674–685, 2018.

[50] K. A. Alnowibet, I. Khan, K. M. Sallam, and A. W. Mohamed, "An efficient algorithm for data parallelism based on stochastic optimization," *Alexandria Engineering Journal*, vol. 61, no. 12, pp. 12 005–12 017, 2022.

[51] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016, pp. 1–16.

[52] A. Balu, Z. Jiang, S. Y. Tan, C. Hedge, Y. M. Lee, and S. Sarkar, "Decentralized deep learning using momentum-accelerated consensus," in *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2021, pp. 3675–3679.

[53] S. Shi, Q. Wang, X. Chu, B. Li, Y. Qin, R. Liu, and X. Zhao, "Communication-efficient distributed deep learning with merged gradient sparsification on gpus," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, IEEE, 2020, pp. 406–415.

[54] X. Luo, W. Qin, A. Dong, K. Sedraoui, and M. Zhou, "Efficient and high-quality recommendations via momentum-incorporated parallel stochastic gradient descent-based learning," *IEEE/CAA Journal of Automatica Sinica*, vol. 8, no. 2, pp. 402–411, 2020.

[55] D. Lee, N. He, P. Kamalaruban, and V. Cevher, "Optimization for reinforcement learning: From a single agent to cooperative agents," *IEEE Signal Processing Magazine*, vol. 37, no. 3, pp. 123–135, 2020.

[56] P. Yang, "Prls-inves: A general experimental investigation strategy for high accuracy and precision in passive rfid location systems," *IEEE Internet of Things Journal*, vol. 2, no. 2, pp. 159–167, 2014.

[57] Y. Bao, Y. Peng, Y. Chen, and C. Wu, "Preemptive all-reduce scheduling for expediting distributed dnn training," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, IEEE, 2020, pp. 626–635.

[58] Q. Wang, J. Zhao, D. Gong, Y. Shen, M. Li, and Y. Lei, "Parallelizing convolutional neural networks for action event recognition in surveillance videos," *International Journal of Parallel Programming*, vol. 45, pp. 734–759, 2017.

[59] M. S. Patil and S. Chickerur, "Study of data and model parallelism in distributed deep learning for diabetic retinopathy classification," *Procedia Computer Science*, vol. 218, pp. 2253–2263, 2023.

[60] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 571–582.

[61] C. Noel and S. Osindero, "Dogwild!-distributed hogwild for cpu & gpu," in *NIPS Workshop on Distributed Machine Learning and Matrix Computations*, 2014, pp. 693–701.

[62] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: Generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 1–15.

[63] G. Heigold, E. McDermott, V. Vanhoucke, A. Senior, and M. Bacchiani, "Asynchronous stochastic optimization for sequence training of deep neural networks," in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2014, pp. 5587–5591.

[64] J. Jiang, X. Feng, Z. Hu, X. Hu, F. Liu, and H. Huang, "Medical image fusion using transfer learning and l-bfgs optimization algorithm," *International Journal of Imaging Systems and Technology*, vol. 31, no. 4, pp. 2003–2013, 2021.

[65] B. Pang, E. Nijkamp, and Y. N. Wu, "Deep learning with tensorflow: A review," *Journal of Educational and Behavioral Statistics*, vol. 45, no. 2, pp. 227–248, 2020.

[66] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[67] S. Tokui, R. Okuta, T. Akiba, Y. Niitani, T. Ogawa, S. Saito, S. Suzuki, K. Uenishi, B. Vogel, and H. Yamazaki Vincent, "Chainer: A deep learning framework for accelerating the research cycle," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 2002–2011.

[68] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, *et al.*, "Pytorch distributed: Experiences on accelerating data parallel training," *arXiv preprint arXiv:2006.15704*, 2020.

[69] K. Wongsuphasawat, D. Smilkov, J. Wexler, J. Wilson, D. Mane, D. Fritz, D. Krishnan, F. B. Viégas, and M. Wattenberg, "Visualizing dataflow graphs of deep learning models in tensorflow," *IEEE transactions on visualization and computer graphics*, vol. 24, no. 1, pp. 1–12, 2017.

[70] D. Strigl, K. Kofler, and S. Podlipnig, "Performance and scalability of gpu-based convolutional neural networks," in *2010 18th Euromicro conference on parallel, distributed and network-based processing*, IEEE, 2010, pp. 317–324.

[71] D. K. Gifford and J. M. Lucassen, "Integrating functional and imperative programming," in *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, 1986, pp. 28–38.

[72] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel computing*, vol. 22, no. 6, pp. 789–828, 1996.

[73] A. A. Awan, C.-H. Chu, H. Subramoni, and D. K. Panda, "Optimized broadcast for deep learning workloads on dense-gpu infiniband clusters: Mpi or nccl?" In *Proceedings of the 25th European MPI Users' Group Meeting*, 2018, pp. 1–9.

[74] B. Lv, B. Liu, F. Liu, N. Xiao, and Z. Chen, "Rm-kvstore: New mxnet kvstore to accelerate transfer performancewith rdma," in *2018 IEEE Symposium on Computers and Communications (ISCC)*, IEEE, 2018, pp. 00 236–00 242.

[75] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed dnn training acceleration," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 16–29.

[76] P. W. Frey and G. Alonso, "Minimizing the hidden cost of rdma," in *2009 29th IEEE International Conference on Distributed Computing Systems*, IEEE, 2009, pp. 553–560.

[77] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: An efficient dynamic resource scheduler for deep learning clusters," in *Proceedings of the Thirteenth EuroSys Conference*, 2018, pp. 1–14.

[78] S. A. Mohamed, A. A. Elsayed, Y. Hassan, and M. A. Abdou, "Neural machine translation: Past, present, and future," *Neural Computing and Applications*, vol. 33, pp. 15 919–15 931, 2021.

[79] A. Lopez, "Statistical machine translation," *ACM Computing Surveys (CSUR)*, vol. 40, no. 3, pp. 1–49, 2008.

[80] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[81] C. Guo, Y. Qiu, J. Leng, C. Zhang, Y. Cao, Q. Zhang, Y. Liu, F. Yang, and M. Guo, "Nesting forward automatic differentiation for memory-efficient deep neural network training," in *2022 IEEE 40th International Conference on Computer Design (ICCD)*, IEEE, 2022, pp. 738–745.

[82] S. Pllana, S. Benkner, F. Xhafa, and L. Barolli, "Hybrid performance modeling and prediction of large-scale computing systems," in *2008 International Conference on Complex, Intelligent and Software Intensive Systems*, 2008, pp. 132–138. DOI: 10.1109/CISIS.2008.20.

[83] Y. Oyama, A. Nomura, I. Sato, H. Nishimura, Y. Tamatsu, and S. Matsuoka, "Predicting statistics of asynchronous sgd parameters for a large-scale distributed deep learning system on gpu supercomputers," in *2016 IEEE International Conference on Big Data (Big Data)*, IEEE, 2016, pp. 66–75.

[84] M. Song, Y. Hu, H. Chen, and T. Li, "Towards pervasive and user satisfactory cnn across gpu microarchitectures," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2017, pp. 1–12.

[85] S. Shi, Q. Wang, P. Xu, and X. Chu, "Benchmarking state-of-the-art deep learning software tools," in *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, IEEE, 2016, pp. 99–104.

[86] S. Shi, Q. Wang, and X. Chu, "Performance modeling and evaluation of distributed deep learning frameworks on gpus," in *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*, IEEE, 2018, pp. 949–957.

[87] D. S. Banerjee, K. Hamidouche, and D. K. Panda, "Re-designing cntk deep learning framework on modern gpu enabled clusters," in *2016 IEEE international conference on cloud computing technology and science (CloudCom)*, IEEE, 2016, pp. 144–151.

[88] C. Boufenar and M. Batouche, "Investigation on deep learning for off-line handwritten arabic character recognition using theano research platform," in *2017 Intelligent Systems and Computer Vision (ISCV)*, IEEE, 2017, pp. 1–6.

[89] A. A. Awan, K. Hamidouche, J. M. Hashmi, and D. K. Panda, "S-caffe: Co-designing mpi runtimes and caffe for scalable deep learning on modern gpu clusters," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2017, pp. 193–205.

[90] R. Collobert, S. Bengio, and J. Mariéthoz, "Torch: A modular machine learning software library," Idiap, Tech. Rep., 2002.

[91] S. Mahon, S. Varrette, V. Plugaru, F. Pinel, and P. Bouvry, "Performance analysis of distributed and scalable deep learning," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, pp. 760–766. DOI: 10.1109/CCGrid49817.2020.00-13.

[92] T. Kavarakuntla, L. Han, H. L. MIEEE, A. L. SMIEEE, and S. B. Akintoye, "Performance analysis of distributed deep learning frameworks in a multi-gpu environment," in *2021 IEEE 20th Intl Conf on Ubiquitous computing and communications(IUCC-2021), The 4th Intl Conf on Data science and Computational Intelligence(DSCI-2021)*, 2021.

[93] Y. S. Shao and D. Brooks, "Energy characterization and instruction-level energy model of intel's xeon phi processor," in *International Symposium on Low Power Electronics and Design (ISLPED)*, IEEE, 2013, pp. 389–394.

[94] D. Didona, F. Quaglia, P. Romano, and E. Torre, "Enhancing performance prediction robustness by combining analytical modeling and machine learning," in *Proceedings of the 6th ACM/SPEC international conference on performance engineering*, 2015, pp. 145–156.

[95] R. Storn, "On the usage of differential evolution for function optimization," in *Proceedings of North American Fuzzy Information Processing*, 1996, pp. 519–523. DOI: 10.1109/NAFIPS.1996.534789.

[96] C.-Y. Lee and C.-H. Hung, "Feature ranking and differential evolution for feature selection in brushless dc motor fault diagnosis," *Symmetry*, vol. 13, no. 7, 2021.

DOI: 10.3390/sym13071291. [Online]. Available: https://www.mdpi.com/2073-8994/13/7/1291.

[97] X. Chen, S. Song, J. Ji, Z. Tang, and Y. Todo, "Incorporating a multiobjective knowledge-based energy function into differential evolution for protein structure prediction," *Information Sciences*, vol. 540, pp. 69–88, 2020.

[98] S. Saha and R. Das, "Exploring differential evolution and particle swarm optimization to develop some symmetry-based automatic clustering techniques: Application to gene clustering," *Neural Comput. Appl.*, vol. 30, no. 3, pp. 735–757, Aug. 2018, ISSN: 0941-0643. DOI: 10.1007/s00521-016-2710-0. [Online]. Available: https://doi.org/10.1007/s00521-016-2710-0.

[99] Y.-H. Li, J.-Q. Wang, X.-J. Wang, Y.-L. Zhao, X.-H. Lu, and D.-L. Liu, "Community detection based on differential evolution using social spider optimization," *Symmetry*, vol. 9, no. 9, 2017. DOI: 10.3390/sym9090183. [Online]. Available: https://www.mdpi.com/2073-8994/9/9/183.

[100] M. Baioletti, A. Milani, and V. Santucci, "Learning bayesian networks with algebraic differential evolution," in *Parallel Problem Solving from Nature – PPSN XV*, Cham: Springer International Publishing, 2018, pp. 436–448.

[101] M. F. Ahmad, N. A. M. Isa, W. H. Lim, and K. M. Ang, "Differential evolution: A recent review based on state-of-the-art works," *Alexandria Engineering Journal*, vol. 61, no. 5, pp. 3831–3872, 2022. DOI: https://doi.org/10.1016/j.aej.2021.09.013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S111001682100613X.

[102] A. Qi, D. Zhao, F. Yu, A. A. Heidari, H. Chen, and L. Xiao, "Directional mutation and crossover for immature performance of whale algorithm with application to engineering optimization," *Journal of Computational Design and Engineering*, vol. 9, no. 2, pp. 519–563, 2022.

[103] A. Anwaar, A. Ashraf, W. H. K. Bangyal, and M. Iqbal, "Genetic algorithms: Brief review on genetic algorithms for global optimization problems," *2022 Human-Centered Cognitive Systems (HCCS)*, pp. 1–6, 2022.

[104] M. S. AbouOmar, Y. Su, H. Zhang, B. Shi, and L. Wan, "Observer-based interval type-2 fuzzy pid controller for pemfc air feeding system using novel hybrid neural network algorithm-differential evolution optimizer," *Alexandria Engineering Journal*, vol. 61, no. 9, pp. 7353–7375, 2022.

[105] N. Ikushima, K. Ono, Y. Maeda, E. Makihara, and Y. Hanada, "Differential evolution neural network optimization with individual dependent mechanism," in *2021 IEEE Congress on Evolutionary Computation (CEC)*, IEEE, 2021, pp. 2523–2530.

[106] R. A. Venkat, Z. Oussalem, and A. K. Bhattacharya, "Training convolutional neural networks with differential evolution using concurrent task apportioning on hybrid cpu-gpu architectures," in *2021 IEEE Congress on Evolutionary Computation (CEC)*, IEEE, 2021, pp. 2567–2576.

[107] T. Tušar, K. Gantar, V. Koblar, B. Ženko, and B. Filipič, "A study of overfitting in optimization of a manufacturing quality control procedure," *Applied Soft Computing*, vol. 59, pp. 77–87, 2017.

[108] B. Reineking *et al.*, "Constrain to perform: Regularization of habitat models," *Ecological Modelling*, vol. 193, no. 3-4, pp. 675–690, 2006.

[109] K. Zhou, Z. Liu, Y. Qiao, T. Xiang, and C. C. Loy, "Domain generalization: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022.

[110] J. Y.-L. Chan, S. M. H. Leow, K. T. Bea, W. K. Cheng, S. W. Phoong, Z.-W. Hong, and Y.-L. Chen, "Mitigating the multicollinearity problem and its machine learning approach: A review," *Mathematics*, vol. 10, no. 8, p. 1283, 2022.

[111] C. Ou, H. Zhu, Y. A. Shardt, L. Ye, X. Yuan, Y. Wang, and C. Yang, "Quality-driven regularization for deep learning networks and its application to industrial soft sensors," *IEEE Transactions on Neural Networks and Learning Systems*, 2022.

[112] L. Zhang, X. Wang, D. Yang, T. Sanford, S. Harmon, B. Turkbey, B. J. Wood, H. Roth, A. Myronenko, D. Xu, *et al.*, "Generalizing deep learning for medical image segmentation to unseen domains via deep stacked transformation," *IEEE transactions on medical imaging*, vol. 39, no. 7, pp. 2531–2540, 2020.

[113] R. de Albuquerque Teixeira, A. P. Braga, R. H. Takahashi, and R. R. Saldanha, "Improving generalization of mlps with multi-objective optimization," *Neurocomputing*, vol. 35, no. 1-4, pp. 189–194, 2000.

[114] L. Tian, Z. Wang, W. Liu, Y. Cheng, F. E. Alsaadi, and X. Liu, "An improved generative adversarial network with modified loss function for crack detection in electromagnetic nondestructive testing," *Complex & Intelligent Systems*, pp. 1–10, 2022.

[115] L. Zhang, M. Yang, and X. Feng, "Sparse representation or collaborative representation: Which helps face recognition?" In *2011 International conference on computer vision*, IEEE, 2011, pp. 471–478.

[116] N. Bacanin, M. Zivkovic, F. Al-Turjman, K. Venkatachalam, P. Trojovskỳ, I. Strumberger, and T. Bezdan, "Hybridized sine cosine algorithm with convolutional neural networks dropout regularization application," *Scientific Reports*, vol. 12, no. 1, p. 6302, 2022.

[117] Y. Chen, J. Guo, J. Huang, and B. Lin, "A novel method for financial distress prediction based on sparse neural networks with l 1/2 regularization," *International Journal of Machine Learning and Cybernetics*, vol. 13, no. 7, pp. 2089–2103, 2022.

[118] Y. Tian, D. Su, S. Lauria, and X. Liu, "Recent advances on loss functions in deep learning for computer vision," *Neurocomputing*, vol. 497, pp. 129–158, 2022.

[119] A. Pandey and A. Kumar, "Deep features based automated multimodel system for classification of non-small cell lung cancer," in *2022 IEEE Delhi Section Conference (DELCON)*, IEEE, 2022, pp. 1–7.

[120] Y. Kono and M. Koizumi, "Model life extension for continuous process: Non-invasive correction of model-plant mismatch with regularization," in *2023 European Control Conference (ECC)*, IEEE, 2023, pp. 1–8.

[121] K. Zhou, Q. Zhang, and J. Li, "Tsvmpath: Fast regularization parameter tuning algorithm for twin support vector machine," *Neural Processing Letters*, vol. 54, no. 6, pp. 5457–5482, 2022.

[122] T. Kavarakuntla, L. Han, H. Lloyd, A. Latham, and S. B. Akintoye, "Performance analysis of distributed deep learning frameworks in a multi-gpu environment," in *2021 20th International Conference on Ubiquitous Computing and Communications (IUCC/CIT/DSCI/SmartCNS)*, IEEE, 2021, pp. 406–413.

[123] G. Niu, X. Li, X. Wan, X. He, Y. Zhao, X. Yi, C. Chen, L. Xujun, G. Ying, and M. Huang, "Dynamic optimization of wastewater treatment process based on novel multi-objective ant lion optimization and deep learning algorithm," *Journal of Cleaner Production*, vol. 345, p. 131 140, 2022.

[124] S. Hooshmand, P. Abedin, M. O. Külekci, and S. V. Thankachan, "I/o-efficient data structures for non-overlapping indexing," *Theoretical Computer Science*, vol. 857, pp. 1–7, 2021.

[125] *Nsight systems*, `https://developer.nvidia.com/nsight-systems`, Accessed: 2021.

[126] F. M. Talaat, H. A. Ali, M. S. Saraya, and A. I. Saleh, "Effective scheduling algorithm for load balancing in fog environment using cnn and mpso," *Knowledge and Information Systems*, vol. 64, no. 3, pp. 773–797, 2022.

[127] E. Gures, I. Shayea, M. Ergen, M. H. Azmi, and A. A. El-Saleh, "Machine learning based load balancing algorithms in future heterogeneous networks: A survey," *IEEE Access*, 2022.

[128] K. Fleetwood, "An introduction to differential evolution," in *Proceedings of Mathematics and Statistics of Complex Systems (MASCOS) One Day Symposium, 26th November, Brisbane, Australia*, 2004, pp. 785–791.

[129] S. Park, J. Lee, and H. Kim, "Hardware resource analysis in distributed training with edge devices," *Electronics*, vol. 9, no. 1, p. 28, 2020.

[130] A. Khan, A. Sohail, U. Zahoora, and A. S. Qureshi, "A survey of the recent architectures of deep convolutional neural networks," *ArXiv*, vol. abs/1901.06032, 2019. [Online]. Available: `http://arxiv.org/abs/1901.06032`.

[131] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: A novel image dataset for benchmarking machine learning algorithms," *ArXiv*, vol. abs/1708.07747, 2017.

[132] F. O. Giuste and J. C. Vizcarra, "Cifar-10 image classification using feature ensembles," *ArXiv*, vol. abs/2002.03846, 2020.

[133] Y. Essam, Y. F. Huang, J. L. Ng, A. H. Birima, A. N. Ahmed, and A. El-Shafie, "Predicting streamflow in peninsular malaysia using support vector machine and deep learning algorithms," *Scientific Reports*, vol. 12, no. 1, p. 3883, 2022.

[134] W. Lin, Z. Wu, L. Lin, A. Wen, and J. Li, "An ensemble random forest algorithm for insurance big data analysis," *Ieee access*, vol. 5, pp. 16 568–16 575, 2017.

[135] J. G. Sled, A. P. Zijdenbos, and A. C. Evans, "A nonparametric method for automatic correction of intensity nonuniformity in mri data," *IEEE transactions on medical imaging*, vol. 17, no. 1, pp. 87–97, 1998.

[136] Y. Wang, J. Nie, P.-T. Yap, F. Shi, L. Guo, and D. Shen, "Robust deformable-surface-based skull-stripping for large-scale studies," in *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2011: 14th International Conference, Toronto, Canada, September 18-22, 2011, Proceedings, Part III 14*, Springer, 2011, pp. 635–642.

[137] T. Kavarakuntla, L. Han, H. Lloyd, A. Latham, A. Kleerekoper, and S. B. Akintoye, "A generic performance model for deep learning in a distributed environment," *arXiv preprint arXiv:2305.11665*, 2023.

Appendix A

Paper: Performance analysis of distributed deep learning frameworks in a multi-gpu environment

2021 20th International Conference on Ubiquitous Computing and Communications (IUCC/CIT/DSCI/SmartCNS)

Performance Analysis of Distributed Deep Learning Frameworks in a Multi-GPU Environment

Tulasi Kavarakuntla, Liangxiu Han, Huw Lloyd, MIEEE, Annabel Latham, SMIEEE, Samson B. Akintoye
Dept. of Computing and Mathematics, Manchester Metropolitan University, Manchester, UK
Tulasi.Kavarakuntla@stu.mmu.ac.uk, {L.Han, Huw.Lloyd, A.Latham, S.Akintoye}@mmu.ac.uk

***Abstract*—Deep Learning frameworks, such as TensorFlow, MXNet, Chainer, provide many basic building blocks for designing effective neural network models for various applications (e.g. computer vision, speech recognition, natural language processing). However, run-time performance of these deep learning frameworks varies significantly even when training identical deep network models on the same GPUs. This study presents an experimental analysis and performance model for assessing deep learning models (Convolutional Neural Networks (CNNs), Multilayer Perceptrons (MLP), Autoencoder) on three frameworks: TensorFlow, MXNet, and Chainer, in a multi-GPU environment. We analyse factors that influence these frameworks' performance by computing the running time of each framework in our proposed model, taking load imbalance factor into account. The evaluation results highlight significiant differences in the scalability of the frameworks, and the importance of load balance in parallel distributed deep learning.**

***Index Terms*—Deep Learning; GPUs; SGD and synchronous SGD; Deep Learning Frameworks, Load imbalance factor.**

I. Introduction

With the available computational power such as GPU, Deep learning (DL) [1], as a subset of machine learning based on artificial neural networks, has attracted much attention due to its nature in discovering correlation structure in data in an unsupervised fashion, which has led to its popularity among the many domains such as image classification, speech recognition, computer vision and natural language processing. However, training a deep learning model is a challenging task due to many constraints such as large data instances and high dimensionality, model complexity and inference time, and model selections. For instance, there are a million parameters defining a deep learning model, which requires large amounts of data to learn from it and is a computationally intensive process. Especially, when the data size and the deep learning models become larger and more complicated, training a model within a considerate period usually demands more hardware memory and computing power such as parallel and distributed computing [2] [3] [4] including data parallelism [5], model parallelism [6], pipeline parallelism [7] and hybrid parallelism [8]. Recently, various distributed deep learning frameworks such as Caffe-MPI [9], TensorFlow [10], MXNet [11], Chainer [12], CNTK [13]) have been proposed, which provide basic building blocks for designing effective neural network models for targeted applications. However, run-time performance of these deep learning frameworks varies significantly even when training identical deep network models on the same GPUs. Existing works have investigated deep learning performance modelling on distributed systems [14], asynchronous stochastic gradient descent performance prediction [15], and analytical models for estimating the optimum utilisation of GPU resources [16] for deep learning, and performance evaluation and benchmarking of deep learning frameworks on GPUs [17] [18]. In this study, we extend the work presented in [18] to analyse the performance of three distributed deep learning frameworks(TensorFlow, MXNet and Chainer) with Convolutional Neural Networks (CNNs), Multilayer perceptrons (MLP) and Autoencoder within a multi-GPU environment. Our contributions include:

- Different from the existing works, by taking account of load imbalance factor and mini-batch time (time taken to divide mini-batches), we build a performance model based on synchronous stochastic gradient descent (S-SGD) to analyse the execution time performance of deep learning frameworks in a multi-GPU environment, and evaluate the model using three deep learning models (Convolutional Neural Networks, Autoencoder and Multilayer Perceptron), each implemented in three frameworks (MXNet, Chainer and Tensorflow) respectively.
- Using our experimental data, we analyze the effect of load imbalance on the scalability of deep learning models, concluding that it is in important contribution to parallel inefficiency.

The remainder of the paper is organised as follows. Section II reviews relevant related work. In section III we develop our performance model based on S-SGD. Section IV presents experiments and analysis on a range of DNN frameworks and models. In Section V, we summarize our conclusions and discuss future work.

II. Related Work

This section presents an overview of the existing performance models used in distributed systems. A performance model [19] [20] provides insight into an implementation's behaviour in future execution contexts and is used to evaluate development-stage design and infrastructure investment decisions.

Yan *et al.* [14] developed performance modelling for exploring the design space and to identify effective system configurations that reduces elapsed time between iterations on the training data. The results shown that error rates of less than

978-1-6654-6667-7/21/$31.00 ©2021 IEEE

DOI 10.1109/IUCC-CIT-DSCI-SmartCNS55181.2021.00071

25% enable them to define and differentiate between desirable and undesirable system parameter combinations.

Oyama et al. [15] developed a performance model for SPRINT, an Asychronous SGD based deep learning system based on mini-batch SGD, running on GPU. The model included the key parameters in asynchronous SGD training are mini-batch size and gradient staleness. There were no other parallelism types in the performance model, as weights were directly synchronised between GPUs. The findings showed that the ASGD deep learning system SPRINT's performance model effectively predicted sweeping time, mini-batch size, staleness, and probability distributions of the fundamental parameters on two GPU-based supercomputers. The prediction model was then used to assess deep learning's scalability for future hardware architectures.

Lee *et al.* [21] used CNN models to perform image recognition, implementing AlexNet on five different frameworks, which include CNTK [13], Caffe-MPI [9], Theano [22], Torch [23], and TensorFlow [10], and assessed the GPU performance characteristics. Each framework includes a variety of convolution algorithms. They performed a comparison based on the performance of some convolution algorithms such as the Winograd method, GEMM, FFT and direct convolution. Scaling DNNs in a single node with multiple GPUs is essential. As a result, they examined the factors that contributed to their overhead when parallelizing the data. The results indicated that by simply altering the framework's options, the training speed could be increased by a factor of two without modifying any source code.

Qi *et al.* [24] proposed a performance model known as Paleo, which combine parallelization strategies, communication schemes, and network architecture to forecast the deep neural networks training performance. In training the AlexNet model, the results showed that hybrid parallelism outperformed data parallelism. Paleo has been compared in several communication schemes, including OneToAll, Tree AllReduce, and Butterfly AllReduce.

Yufei *et al.* [25] established a performance model for estimating resource consumption and performance efficiency on FPGAs, that was applied to the design phase to find and explore optimal design options. The authors mainly focused on DRAM efficiency, response time, and PE utilization. The evaluation results showed that the model's predictions are quite closely match (within a factor of three) the actual test results obtained on field programmable gate arrays.

Andre Viebke [26] investigated performance prediction accuracy using three alternative CNN models on an Intel Xeon Phi Processor. These two parameterized performance models estimated training convolutional neural networks' execution time. The first performance model used minimal parameter estimate approaches. The second model estimated sequential work by measuring forward and backward propagation. The results showed that the first model's average performance prediction accuracy was 4% higher than the second model.

Shi *et al.* [18] created performance models to assess the performance of a variety of distributed deep learning frameworks (such as CNTK or MXnet) with Alexnet, GoogleNet and ResNet models on GPU computing platforms. They developed models for SGD in single-GPU, multi-GPU, and distributed cluster systems. Through experimental analysis, identified overheads and limitations that could be further optimized in terms of system configuration.

TABLE I: Notation used in this paper (after [18])

Symbol	Description
N_g	Number of total GPUs
t_{iter}	An Iteration time
t_{io}	I/O time of an iteration
t_{h2d}	Communication time between CPU and GPU of an Iteration
t_{md}	Time for dividing batches into mini-batches
t_f	Forward operation time of an iteration
t_b	Backward operation time of an iteration
$t_{f_i}^{(l)}$	Time taken by i^{th} GPU for l^{th} layer in forward operation
$t_{b_i}^{(l)}$	Time taken by i^{th} GPU for l^{th} layer in backward operation
t_{c_i}	Time taken by i^{th} GPU for computing gradients aggregation
t_u	Model update time of an iteration
t_c	Gradients aggregation time of an iteration

In this work, we develop a performance model based on that of [18], and evaluate it in the context of a single node, multi-GPU system. Different from the existing works, we refine some parts of the model by further dividing the timings for stages of the training, and also consider the effect of load imbalance on the performance. We analyse the running performance of Convolutional Neural Network, Multilayer Perceptron and Autoencoder models on three different frameworks respectively.

III. The Proposed Performance Model

A. Preliminaries

For convenience and easy reference, the notations used here follow the notations in [18].

1) Mini-batch SGD: Let consider an L-layered DNN model, which is trained iteratively on a GPU using mini-batch SGD. Each iteration consists of five steps: 1) Fetch a training data mini batch from either internal or external disk; 2)Transfer the training data from CPU memory to GPU memory through PCIe ; 3) Perform feed-forward calculations layer by layer by using GPU kernels; 4) Use backward propagation for gradients computation from Layer L to Layer 1; 5) Calculate average gradients and update the model.

An iteration time can be expressed as:

$$t_{iter} = t_{io}+t_{h2d}+t_f+t_b+t_u = t_{io}+t_{h2d}+\sum_{i=1}^{l} t_f^l+\sum_{i=1}^{l} t_b^l+t_u \quad (1)$$

2) S-SGD using multiple GPUs: In comparison with the SGD, S-SGD consists of six steps. The 1st - 4th steps are similar to the SGD. The 5th step is gradient aggregation, and

the sixth step is updating the model. The iteration time of the S-SGD implementation can be represented as:

$$t_{iter} = t_{io} + t_{h2d} + \sum_{i=1}^{l} t_f^l + \sum_{i=1}^{l} t_b^l + \sum_{i=1}^{l} t_c^l + t_u \quad (2)$$

In the single GPU environment, $\sum_{i=1}^{l} t_c^l = 0$.

B. The Proposed Performance Model based on S-SGD

In this work, different from the existing works [18], we build a performance model of S-SGD by inclusion of two new parameters: time taken to divide the batch into mini-batches and maximum time taken by GPU, taking load imbalance factor into account.

Assume that a machine contains k GPUs. Given the model to be trained, each GPU will individually keep a complete set of model parameters, although parameter values are identical and synchronised across GPUs. For an example, Figure 1 describes the workflow of the performance model when $k = 4$. In general, the model works as discussed in section III-A2 using multiple GPUs. Thus, we develop our proposed performance model of training DNNs with S-SGD in the TensorFlow, MXNet and Chainer frameworks.

Here, S-SGD executes feed-forward and backward propagation simultaneously on each GPU with the same model and distinct training datasets. We consider the time taken for dividing each batch into mini-batches and we also consider the maximum time taken by each GPU in forward processing. By substituting these two parameters in our modelling function, the iteration time t_{iter} for the S-SGD implementation can be represented as follows:

$$t_{iter} = t_{io} + t_{h2d} + t_{md} + max_{i\epsilon(1,n)}(\sum_{i=1}^{l} t_f^l + \sum_{i=1}^{l} t_b^l + \sum_{i=1}^{l} t_c^l) + t_u \quad (3)$$

In the single GPU environment, $\sum_{i=1}^{l} t_{c_i}^l = 0$. The time of an iteration can be written as:

$$t_{iter} = t_{io} + t_{h2d} + t_{md} + \sum_{i=1}^{l} t_f^l + \sum_{i=L}^{1} t_b^l + t_u \quad (4)$$

We now consider the effects of optimization strategies, which make use of task parallelism, which are found in the existing deep learning frameworks. We can notice two possible optimization opportunities. Initially, we can parallelize data reading tasks with the computing tasks, which effectively hide the time cost of disk I/O. Secondly, gradient communication tasks with the back propagation computing tasks can be parallelized. In the case of overlapping I/O with computation, the first step is frequently processed with multiple threads, allowing the I/O time of a new iteration to overlap with the computing time of the preceding iteration. In such a manner, computing in the following iteration can begin immediately after the model is completed. Thus, the average iteration time of pipelined SGD is calculated as;

$$t_{iter} = \max(t_f + t_b + t_u, t_{io} + t_{h2d} + t_{md}) \quad (5)$$

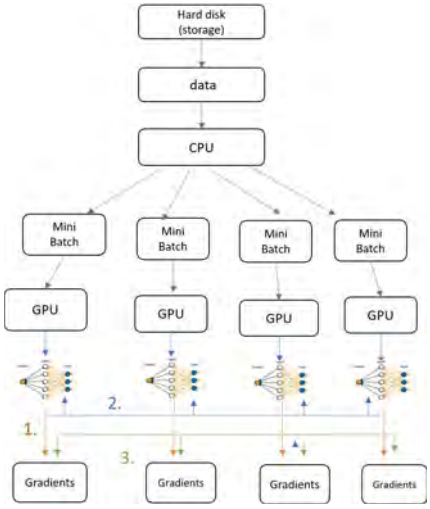


Fig. 1: Workflow of the model: (1) loss and gradient computation, (2) gradient aggregation, (3) parameter update

In a scenario where the gradient communication overlaps with the computation, the gradient communication could be re-programmed to run concurrently with the backpropagation steps. Therefore, the overheads of I/O and gradient communications need to be reduced to achieve good performance and scalability, Let t'_{iter} and t'_{io} represent the iteration time and I/O times respectively on N_g GPUs. The speedup of using N_g GPUs is the given by

$$S = N_g \frac{t_{iter}}{t'_{iter}} \quad (6)$$

Accounting for the optimizations described above, we can now write this as:

$$S = N_g \frac{\max\{t_{io} + t_{h2d}, t_f + t_b\}}{\max\{t'_{io} + t_{h2d}, t_f + t_b + t_c\}} \quad (7)$$

IV. Experiments

In this section, we describe our experimental environment and present the results of experiments to investigate the running time performance of DNN models and frameworks, and how communication tasks affect the scalability of S-SGD.

A. Experimental Setup

Initially, we define the hardware specification conducted in our experiments. We used a single node with three GPUs. GPU@ GEFORCE RTX 2080, CPU@ 2.60 GHZ 2.81GHZ and Memory (RAM)- 16.0GB. Software used for the experimentation are TensorFlow version-2.1.0, MXnet version -1.6.0, Chainer version-7.4.0, python version-3.6.9, CUDA version-10.2. and operating system- Linux. We used Nsight profiler [27] to find the running time performance of GPU activity.

Furthermore, we measure the time duration of an iteration for processing a mini-batch of input data to evaluate the

execution performance. Here we choose three Neural Network models i.e., the Multilayer Perceptron model (MLP), Convolutional Neural Network model (CNN) and Autoencoder model. The models are trained on the MNIST dataset on three frameworks i.e., TensorFlow [10], MXNet [11] and Chainer [12] by applying distributed and parallel training. The MNIST dataset contains 70,000 images of ten handwritten digits and is divided into training and test datasets. The training dataset has 60,000 images, while the test dataset contains 10,000 images. All the two datasets have 10 classes, the 10 numerical digits. In our experiment, we run two epochs and discard the result of the first epoch, since this will include some setup which is not representative of the average training load over a long run. We recorded each iteration time and average these over the second epoch, calculating the mean and standard deviation of each time.

B. Performance Metrics

The speedup and load imbalance factor are selected as performance metrics for run time evaluation on three different frameworks. The speedup is defined in Equation 7. The load imbalance factor for a set of parallel process which execute in times $t_1 \dots t_N$ is defined as

$$LIF = \frac{\max_{i\in[1\dots N]} t_i}{(1/N)\sum_{i=1}^{N} t_i}. \quad (8)$$

$LIF = 1$ corresponds to a perfectly balanced load, whereas for imbalanced loads, $LIF > 1$.

The experimental evaluation is focused on two goals below.

- The first experimental goal is to investigate running time performance of each model using different frameworks in a multi-GPU environment.
- The second experimental goal is to investigate how load imbalance factor of each model under different computing nodes/GPU affects the computing efficiency.

C. Results and Analysis

This section illustrates the running performance followed with analysis based on the performance modelling of TensorFlow, MXNet and Chainer in training CNN, MLP and Autoencoder models in a multiple GPU environment.

1) Single GPU: Initially, we describe the performance results obtained on a single GPU. The average time taken by a framework to complete one iteration during training is used to evaluate the framework's performance. As a result, we can compare the time spent on each step of SGD. The timings are given in Table II and shown graphically in Figure 2. The results of each phase will be discussed in the following sections.

In the initial phase of the performance model, all three frameworks have multiple threads to read data from the CPU memory to the GPU. By observing the results in Table II we see that for all frameworks the I/O time is small. In the second phase, after the reading of data from disk to memory, the data should be transmitted to the GPU for training purpose. Our tested environment uses PCIe to connect the CPU and GPU, which provides a total bandwidth of 11 GB/sec. From the

TABLE II: Measured time of SGD phases on single GPU. All times are given in seconds, as the mean and standard deviations over all iterations in a single epoch of training.

CNN	Chainer	MXNet	TensorFlow
t_{io}	0.0004±0.00002	0.0002±0.00005	0.0006±0.00008
t_{h2d}	0.0383±0.0054	0.0201±0.0027	0.0212±0.0023
t_{md}	0.0006±0.00003	0.0003±0.00001	0.0005±0.00002
$\sum_{i=1}^{l} t_{f_i}^l$	0.0663±0.0031	0.0307±0.0073	0.3489±0.0729
$\sum_{i=1}^{l} t_{b_i}^l$	0.0594±0.0030	0.1347±0.0040	0.1151±0.0170
t_u	0.2365±0.0194	0.1564± 0.0514	0.2636±0.0469
t_{iter}	0.4009±0.0240	0.3421±0.0354	0.7494 ±0.1391

MLP	Chainer	MXNet	TensorFlow
t_{io}	0.0001±0.000018	0.0005±0.00008	0.0003±0.000025
t_{h2d}	0.0331±0.0062	0.0182±0.00078	0.0199±0.0035
t_{md}	0.0006±0.00001	0.0003±0.00003	0.0005±0.00008
$\sum_{i=1}^{l} t_{f_i}^l$	0.0523±0.0067	0.1034 ±0.0082	0.0576±0.0045
$\sum_{i=1}^{l} t_{b_i}^l$	0.0481±0.0280	0.1754±0.0187	0.1680 ±0.0134
t_u	0.4533±0.0095	0.2054±0.00099	0.5985±0.0089
t_{iter}	0.5869±0.0575	0.3992±0.0591	1.4371±0.0597

AN	Chainer	MXNet	Tensorflow
t_{io}	0.0004±0.00005	0.0001±0.00003	0.0005±0.00008
t_{h2d}	0.0316±0.0026	0.0185±0.0090	0.0215±0.0030
t_{md}	0.0006±0.00003	0.0003±0.00001	0.0005±0.00008
$\sum_{i=1}^{l} t_{f_i}^l$	0.1388±0.0045	0.1322±0.0064	0.1595±0.0072
$\sum_{i=1}^{l} t_{b_i}^l$	0.1421± 0.0056	0.2265± 0.0076	0.4274±0.0103
t_u	0.3675±0.0201	0.3287±0.0307	0.3765 ±0.0215
t_{iter}	0.6804±0.0328	0.706±0.0258	0.9854±0.0320

TABLE III: Gradient aggregation time in the multi-GPU experiments

Network	Framework	t_{comm}	
		2 GPUs	3 GPUs
CNN	Tensorflow	0.3945	0.4017
	MXNet	0.3245	0.3415
	Chainer	0.3106	0.3404
MLP	Tensorflow	0.3024	0.4145
	MXNet	0.3156	0.2569
	Chainer	0.2945	0.2345
Autoencoder	Tensorflow	0.7187	0.7199
	MXNet	0.3565	0.3698
	Chainer	0.4563	0.4583

results in Table II, we see that Chainer typically has higher memory copy time than both TensorFlow and MXNet.

In the third phase (t_{md}), the three frameworks differ in the data distribution to GPUs. In the Chainer framework, the data batch is divided into multiple batches in the GPU whereas in the MXNet and Tensorflow framework, batches are divided into mini-batches on the CPU and then transferred to the GPUs dynamically. As a result the Chainer framework takes 0.3s higher compared to MXNet and TensorFlow.

In the forward phase, we can see that while the results are comparable in the case of the Autoencoder and MLP models, in the CNN model, Tensorflow is significantly slower than both the MXNet and Chainer frameworks. MXNet's performance is good in the forward phase due to its usage of auto symbolic differentiation and imperative programmimg [11]. In the case of the CNN, both Chainer and MXNet are able to autotune to determine the optimal convolutional algorithms for convolutional layers, but TensorFlow does not allow the

convolution techniques to be customized. TensorFlow uses the Winograd algorithm, which in some situations may be suboptimal. Considering the CNN model, MXNet makes use of GEMM-based convolution, which results in 0.05s less forward phase and up to 0.15s more backward phase. Chainer employs the FFT technique [21], which results in a forward phase that is 0.06s higher and 0.1s less in the backward phase.

Next in the backward phase, MXNet is slower than the TensorFlow and Chainer frameworks. The values t_f and t_b are different in performance due to differing use of the cuDNN API. cuDNN may have different performance depending on the parameters that are used. Some factors that affect performance are: Data Layout, Implicit matrix multiplications, Dimension quantization techniques, Convolution parameters such as Batch size, Height and width filtersize, channels in and out (NHWC, NCWH) and strides. For example, in MXNet and Chainer, the NCHW data layout are used whereas TensorFlow has NHWC layout which acts as a performance factor.

2) Multi-GPU: In a multi-GPU per node testing, we scaled the mini-batch with the number of GPUs. Each GPU has the same dataset. As the number of GPUs increases, data communication overhead increases due to the data aggregation process between devices. Our measurements of this time t_{comm} are give in Table III. Figure 3 shows the results for the speedup when running on two and three GPUs, and the breakdown of the timings in terms of the performance model are shown in Figure 4.

From Figure 3, we see that MXNet achieves linear scaling on one to three GPUs, while Chainer achieves speeds 0.2X less than MXNet. From Figure 4, we see that the data aggregation time t_a in MXNet is less than in the TensorFlow and Chainer frameworks. Here, MXNet parallelizes the gradient aggregation with back propagation i.e., after the gradients of the current layer(l_i) are computed, the preceding layer (l_{i-1}) of backward propagation can be performed without latency. As a result, gradient computation of (l_{i-1}) is parallelized with gradient aggregation of l_i. Thus, following computing layers can hide much of the synchronisation overhead of gradients. As a result, MXNet has less aggregation time and good scalability compared to other frameworks. TensorFlow implements S-SGD differently. It has no parameter server and uses peer-to-peer memory access if it is compatible with the hardware topology. Each GPU receives gradients from other GPUs, averages them, and updates the model when the backward propagation completes, even from the decentralised method. In this process, the model update t_u and backward propagation has no computation overlap, which led to the observed relatively poor scaling performance in TensorFlow.

3) Load Imbalance Factor: Load balancing in a parallel system plays a major role in determining scalability. A load imbalance occurs when work is distributed unevenly among workers. Here we have calculated the *Load Imbalance Factor* for each neural network model in each deep learning framework based on Equation 8.

From the results in Table IV, it is clear that all three frameworks are not well balanced, since in all cases the load imbalance factor is greater than one, and in some cases significantly greater. Qualitatively, we see that the higher values of load imbalance correspond to the lower speedups, and degraded scalability, see Figure 3. For example, in the case of Tensorflow, we see that poor scalability is accompanied by relatively high values of the load imbalance factor.

TABLE IV: Load Imbalance Factor

Network	Framework	Load Imbalance Factor	
		2 GPUs	3 GPUs
TensorFlow	CNN	1.15	1.23
	MLP	1.189	1.20
	AN	1.175	1.27
Chainer	CNN	1.025	1.052
	MLP	1.032	1.043
	AN	1.152	1.202
MXNet	CNN	1.013	1.030
	MLP	1.015	1.079
	AN	1.142	1.213

Here we also present further linear regression analysis to understand how load imbalance factor contributes to parallel inefficiency, according to the equation below:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \epsilon \tag{9}$$

where x_1 and x_2 represent the number of GPUs and load imbalance factor respectively, y is the total execution time of an epoch, $\beta_0, \beta_1, \beta_2$ are the regression coefficients and ϵ represents a random value indicating the error in each observation of y. The values of β_0,β_1, and β_2 should be chosen to minimise the sum of squared prediction errors.

We find the following values for the coefficients in the nine cases. For CNN model using TensorFlow,

$$y^{'} = 0.00001 - 40.5588\bar{X}_1 + 369.4848\bar{X}_2 \tag{10}$$

For MLP model using TensorFlow,

$$y^{'} = 0.00001 - 16.1671\bar{X}_1 + 245.9177\bar{X}_2 \tag{11}$$

For AN model using TensorFlow,

$$y^{'} = 0.00002 - 49.1037\bar{X}_1 + 398.6701\bar{X}_2 \tag{12}$$

For CNN model using MXNet,

$$y^{'} = 0.000011 - 5.57483\bar{X}_1 + 287.4133\bar{X}_2 \tag{13}$$

For MLP model using MXNet,

$$y^{'} = 0.00002 - 28.9346\bar{X}_1 + 326.7283\bar{X}_2 \tag{14}$$

For AN model using MXNet,

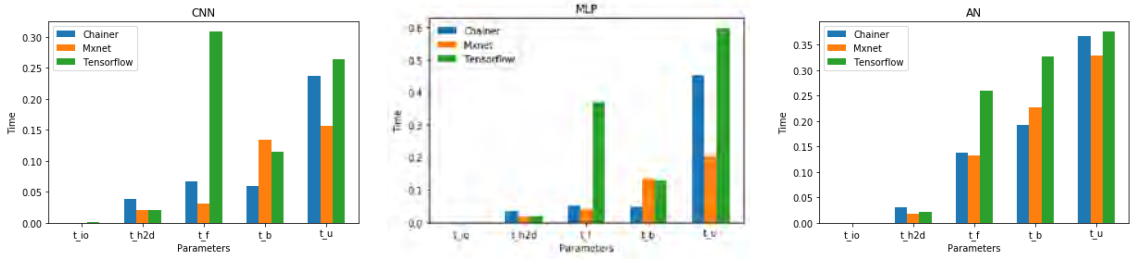
$$y^{'} = 0.00002 - 33.1037\bar{X}_1 + 398.6701\bar{X}_2 \tag{15}$$

For CNN model using chainer,

$$y^{'} = 0.00001 - 9.00791\bar{X}_1 + 286.8447\bar{X}_2 \tag{16}$$

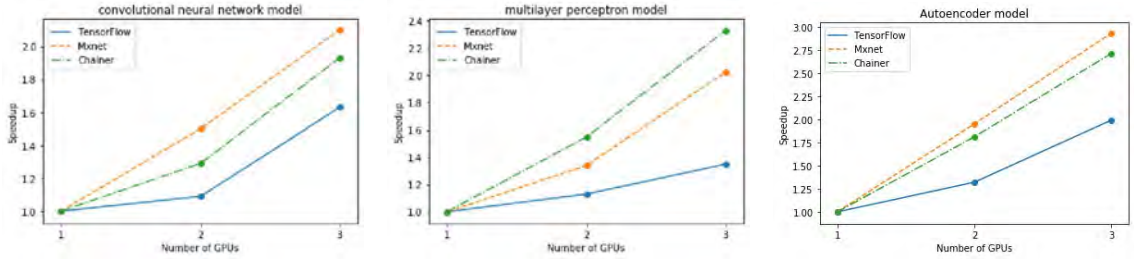
For MLP model using chainer,

$$y^{'} = 0.000016 - 10.1584\bar{X}_1 + 292.1287\bar{X}_2 \tag{17}$$



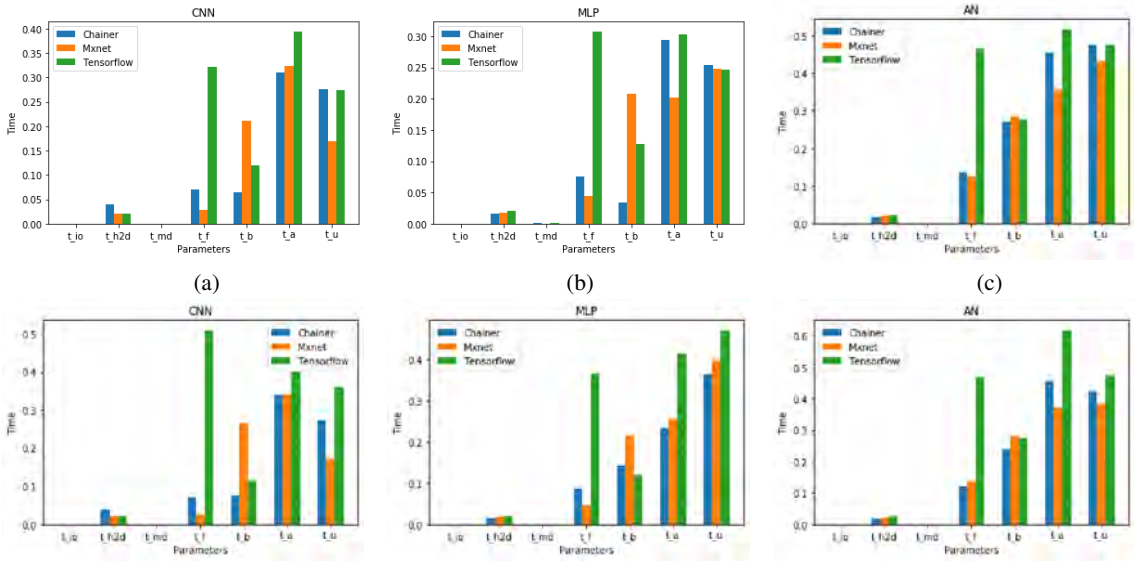
(a) (b) (c)

Fig. 2: Iteration times on a single GPU for (a) CNN (b) MLP and (c) Autoencoder models



(a) (b) (c)

Fig. 3: Measured speedup for the three frameworks on different numbers of GPUs for the three DNN models (a) CNN, (b) MLP and (c) Autoencoder.



(a) (b) (c)

(d) (e) (f)

Fig. 4: Iteration time on multiple GPUs. Results for two GPUs are shown in (a), (b), (c) for CNN, MLP and Autoencoder. Results for three GPUs are in (d), (e) and (f).

For AN model using chainer,

$$y^{'} = 0.000068 - 25.584\bar{X}_1 + 260.263\bar{X}_2 \qquad (18)$$

where $y^{'}$ is the computed prediction execution time as a function of the number of GPUs and the load imbalance factor.

We compute their coefficients of determination, R^2, to further investigate the impact of number of GPUs and load imbalance factor on execution time. R^2 represents the proportion of the variance in execution time that is predicted from both number of GPUs and load imbalance factor. It is defined as follows:

$$R^2 = 1 - \frac{SS_{rss}}{SS_{tss}} \tag{19}$$

where SS_{rss} and SS_{tss} are the residual sum of squares and the total sum of squares. They are defined as:

$$SS_{rss} = \sum (y - y^{'})^2 \tag{20}$$

and,

$$SS_{tss} = \sum (y - \bar{y})^2 \tag{21}$$

We find R^2 values for (CNN, MLP, AN) TensorFlow of (0.9904, 0.9963, 0.9960), for MXNet of (0.9901, 0.9987, 0.9903) and for Chainer of (0.99, 0.9963,0.9965). These results imply that the regression forecasts are accurate in predicting the relationship between execution time and load imbalance factor. As the value of R^2 increases, the model's fit to the training data becomes more accurate and precise. The results confirm the importance of load balancing to achieve scalability in distributed deep learning.

V. Conclusion and future work

We have evaluated the performance of different deep learning frameworks over different deep learning neural networks in terms of scalability in a multi-GPU environment, taking into account a range of factors affecting performance, including load imbalance. We have further extended an existing performance model [18] based on synchronous-S-SGD with the inclusion of two new parameters: time taken to divide the batch into mini-batches and maximum time taken by GPU. The proposed performance model was built to measure the performance of different deep learning framework implementations which include TensorFlow, MXNet and Chainer frameworks on three models: Convolutional neural network, Multilayer perceptron and Autoencoder models, in a multi-GPU environment. The experimental results have shown that MXNet and Chainer have better scalability compared to TensorFlow for all three models. Moreover, our analysis of the load imbalance factor has shown that load balancing is a contributing factor to scalability in distributed deep learning, and high load imbalance is strongly correlated with poor scalability in our experiments. Future work will probe the reason for the load imbalance in these cases, with the aim of discovering optimal parameters to keep the load balanced.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[2] Y. Ko, K. Choi, J. Seo, and S.-W. Kim, "An in-depth analysis of distributed training of deep neural networks," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 994–1003.

[3] S. Shi, Q. Wang, K. Zhao, Z. Tang, Y. Wang, X. Huang, and X. Chu, "A distributed synchronous sgd algorithm with global top-k sparsification for low bandwidth networks," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 2238–2247.

[4] Y. Kim, H. Choi, J. Lee, J.-S. Kim, H. Jei, and H. Roh, "Efficient large-scale deep learning framework for heterogeneous multi-gpu cluster," in *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, 2019, pp. 176–181.

[5] B. Shinde and S. T. Singh, "Data parallelism for distributed streaming applications," in *2016 International Conference on Computing Communication Control and automation (ICCUBEA)*, 2016, pp. 1–4.

[6] M. H. Mofrad, R. Melhem, Y. Ahmad, and M. Hammoud, "Studying the effects of hashing of sparse deep neural networks on data and model parallelisms," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, 2020, pp. 1–7.

[7] N. Takisawa, S. Yazaki, and H. Ishihata, "Distributed deep learning of resnet50 and vgg16 with pipeline parallelism," in *2020 Eighth International Symposium on Computing and Networking Workshops (CANDARW)*, 2020, pp. 130–136.

[8] K.-N. Joo and C.-H. Youn, "Accelerating distributed sgd with group hybrid parallelism," *IEEE Access*, vol. 9, pp. 52 601–52 618, 2021.

[9] A. A. Awan, K. Hamidouche, J. M. Hashmi, and D. K. Panda, "S-caffe: Co-designing mpi runtimes and caffe for scalable deep learning on modern gpu clusters," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2017, pp. 193–205.

[10] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[11] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[12] S. Tokui, R. Okuta, T. Akiba, Y. Niitani, T. Ogawa, S. Saito, S. Suzuki, K. Uenishi, B. Vogel, and H. Yamazaki Vincent, "Chainer: A deep learning framework for accelerating the research cycle," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 2002–2011.

[13] A. A. Awan, K. Hamidouche, A. Venkatesh, and D. K. Panda, "Efficient large message broadcast using nccl and cuda-aware mpi for deep learning," in *Proceedings of the 23rd European MPI Users' Group Meeting*, 2016, pp. 15–22.

[14] F. Yan, O. Ruwase, Y. He, and T. Chilimbi, "Performance modeling and scalability optimization of distributed deep learning systems," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2015, pp. 1355–1364.

[15] Y. Oyama, A. Nomura, I. Sato, H. Nishimura, Y. Tamatsu, and S. Matsuoka, "Predicting statistics of asynchronous sgd parameters for a large-scale distributed deep learning system on gpu supercomputers," in *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 2016, pp. 66–75.

[16] M. Song, Y. Hu, H. Chen, and T. Li, "Towards pervasive and user satisfactory cnn across gpu microarchitectures," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 1–12.

[17] S. Shi, Q. Wang, P. Xu, and X. Chu, "Benchmarking state-of-the-art deep learning software tools," in *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*. IEEE, 2016, pp. 99–104.

[18] S. Shi, Q. Wang, and X. Chu, "Performance modeling and evaluation of distributed deep learning frameworks on gpus," in *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*. IEEE, 2018, pp. 949–957.

[19] S. Pllana, S. Benkner, F. Xhafa, and L. Barolli, "Hybrid performance modeling and prediction of large-scale computing systems," in *2008 International Conference on Complex, Intelligent and Software Intensive Systems*. IEEE, 2008, pp. 132–138.

[20] T. Fahringer, S. Pllana, and J. Testori, "Teuta: Tool support for performance modeling of distributed and parallel applications," in *International Conference on Computational Science*. Springer, 2004, pp. 456–463.

[21] H. Kim, H. Nam, W. Jung, and J. Lee, "Performance analysis of cnn frameworks for gpus," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2017, pp. 55–64.
[22] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky *et al.*, "Theano: A python framework for fast computation of mathematical expressions," *arXiv e-prints*, pp. arXiv–1605, 2016.
[23] R. Collobert, S. Bengio, and J. Mariéthoz, "Torch: a modular machine learning software library," Idiap, Tech. Rep., 2002.
[24] H. Qi, E. R. Sparks, and A. Talwalkar, "Paleo: A performance model for deep neural networks." in *ICLR (Poster)*, 2017.
[25] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "Performance modeling for cnn inference accelerators on fpga," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 4, pp. 843–856, 2019.
[26] A. Viebke, S. Pllana, S. Memeti, and J. Kolodziej, "Performance modelling of deep learning on intel many integrated core architectures," in *2019 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2019, pp. 724–731.
[27] "Nsight systems," https://developer.nvidia.com/nsight-systems, accessed: 2021.

Appendix B

Paper: A Generic Performance Model for Deep Learning in a Distributed Environment

A Generic Performance Model for Deep Learning in a Distributed Environment

Tulasi Kavarakuntla, Liangxiu Han, Huw Lloyd, MIEEE, Annabel Latham, SMIEEE, Anthony Kleerekoper, Samson B. Akintoye

Dept. of Computing and Mathematics, Manchester Metropolitan University, Manchester, UK

Tulasi.Kavarakuntla@stu.mmu.ac.uk, {L.Han, Huw.Lloyd, A.Latham, Akleerekoper, S.Akintoye}@mmu.ac.uk

***Abstract*—Performance modelling of a deep learning application is essential to improve and quantify the efficiency of the model framework. However, existing performance models are mostly case-specific with limited capability of applying to new deep learning frameworks/applications. In this paper, we propose a generic performance model of an application in a distributed environment with a generic expression of the application execution time that considers the influence of both intrinsic factors/operations (e.g. algorithmic parameters/internal operations) and extrinsic scaling factors (e.g. data chunks or batch size). We formulate it as a global optimisation problem and solve it based on a cost function and differential evolution algorithm to find the best fit values of the constants in the generic expression to match the experimentally determined computation time. We have evaluated the proposed model on three deep learning frameworks (i.e., TensorFlow, MXnet, and Pytorch). The experimental results show that the proposed model can provide accurate performance predictions and interpretability. In addition, the proposed work can be applied to any distributed deep neural network without instrumenting the code and has better functionalities like explaining internal parameters performance and scalability.**

I. Introduction

Deep neural networks are effective tools for unsupervised data exploration to discover correlation structures. Deep neural network architectures necessitate the use of high computational resources for training a large amount of data requires a parallelised and distributed environment. Performance models are used to obtain run-time estimates by modelling various aspects of an application on a target system. However, accurate performance modelling is a challenging task. Existing performance models in deep learning have been proposed, which are broadly categorised into two methodologies: analytical modelling and empirical modelling. Analytical modelling relies on the white-box approach. Empirical modelling is a good alternative to analytical models and predicts the outcome of an unknown set of system parameters based on observation and experimentation. However, the current methods in analytical modelling and empirical modelling are the poor presentation of the underlying internal operations and lack of providing an unbiased experimental study in the distributed environment. We are inspired by the hybridisation of the analytical model and empirical modelling. We developed a novel generic performance model in a distributed environment that gives accurate performance predictions and applicable to any distributed deep neural network without instrumenting the code and having better functionalities like explaining intrinsic parameters performance and scalability that provides added value in the field.

II. Related Work

Performance modelling involves prediction – estimating the performance of a new system or the impact of a change in workload on an existing system. Existing performance modelling of deep learning frameworks can be broadly divided into two categories: 1) Analytical modelling, 2) Empirical modelling.

1) Analytical Performance Modelling: This subsection provides the existing works developed in a distributed environment using analytical modelling. Yan *et al.* [1] evaluated the effects of partitioning and resourcing decisions on distributed system architectures. Qi *et al.* [2] proposed an analytical performance model named Paleo, predicting the deep neural network performance by considering communication schemes, network architecture and parallelization strategies. Shi *et al.* [3] developed an analytical performance model to evaluate various distributed deep learning framework performance.

2) Empirical Modelling: Empirical modelling builds models through observation and experimentation, which is antithetic to analytical modelling. Woo et al. [4] developed a data-driven model to evaluate collective communication techniques in a distributed environment. A new approach named hybrid model has been proposed [5] by combining the elements of analytical modelling and empirical modelling for better performance prediction developed in other fields. Unlike the existing works, we developed a generic expression applicable to any distributed deep neural network without instrumenting the code and having better functionalities like explaining internal parameters performance and scalability.

III. Methodology

A. The Proposed Generic Performance Model

Given an application consisting of a number of processes in a distributed environment, the computing efficiency, which is also known as the execution time of the application, can be considered from two levels: 1) Execution time of internal processes of the application; and 2) External scaling factors that affect the computing efficiency. A generic performance

model for computing total computational time(t) per iteration of an application can be described as follows:

$$t(I, E, x) = (t_I)\, f_E + C \tag{1}$$

We represent internal time as t_I and can be represented as:

$$t_I = \sum_{i=1}^{n} a_i I_i^{p_i} \tag{2}$$

In (1), t_I represents the time affected by intrinsic parameters, f_E represents extrinsic scaling factors that affect the computing performance, and C is a constant. We represent the individual processes as a polynomial in terms of the internal parameters. In (2), the coefficients a_i relate to the relative importance of the processes, and the powers p_i relate to the computational complexity.
The external scaling factor f_E, which can be represented as:

$$f_E = \prod_{j=1}^{m} E_j^{q_j} \tag{3}$$

Here, the powers q_j gives information about scalability. By substituting the t_I and f_E in (1), the computational time (t) is given as follows:

$$t(I, E, x) = \left(\sum_{i=1}^{n} a_i I_i^{p_i} \right) \prod_{j=1}^{m} E_j^{q_j} + C \tag{4}$$

Here $\boldsymbol{x} = \{a_1, ..., a_{n_I}, p_1, ... p_{n_I}, q_1, ... q_{n_E}, c\} \in \mathbb{R}^M$ is a vector formed by combining $\boldsymbol{a}, \boldsymbol{p}, \boldsymbol{q}$ and coefficient $\boldsymbol{C}$. In (4), $\boldsymbol{a}, \boldsymbol{p}, \boldsymbol{q}$ and coefficient $\boldsymbol{C}$ are unknown constants. We compute the optimal values of these unknown constants (total: $M = 2n_I + n_E + 1$) using the differential evolution algorithm [6], on the basis of experiments which measure training time for different values of the intrinsic and extrinsic parameters. We have:

$$\text{Intrinsic parameters: } I_{i,k}, \quad i \in [1, n_I], \;\; k \in [1, N] \tag{5}$$

$$\text{Extrinsic parameters: } E_{j,k}, \quad j \in [1, n_E], \;\; k \in [1, N] \tag{6}$$

$$\text{measured time-per-iteration : } t_k, \;\; k \in [1, N]. \tag{7}$$

Here, i, j are denoting the input feature indices. $k \in [1, N]$ indicate the sample index in dataset $\boldsymbol{D}$. N is the number of input samples in $\boldsymbol{D}$. Given the generic expression as shown in (4), we formulate the cost function as the mean absolute difference between predicted execution time and the actual measured times. We solve this optimization problem by using the Differential Evolution algorithm (DE) [6], a population-based evolutionary algorithm for continuous optimization.

TABLE I: Mean absolute error on predictions of the performance models on the 500 instances in the evaluation dataset in seconds.

	TensorFlow	**MXnet**	**Pytorch**
DE	0.90	0.55	3.55
RF	0.15	0.39	6.42
SVM	4.92	3.17	15.44

B. Results and Analysis

This section shows the results of the proposed performance model for three well-known deep neural networks, i.e., TensorFlow, MXnet and Pytorch and compared with the two standard machine learning algorithms, which include support vector machine and random forest regressor. We recorded the training performance of a convolutional neural network on image classification problems using the MNIST, MNIST-fashion and CIFAR-10 datasets. We took 1500 samples with randomly chosen values of the parameters, using 1000 samples to fit the performance models, retaining 500 for evaluation. Table I shows mean absolute error values in seconds on predictions of the performance models. Table II shows scalability in various frameworks, respectively.

TABLE II: nGPUs scaling power in various frameworks. Here nGPUs represent number of GPUs.

Frameworks	**nGPUs scaling power**
TensorFlow	-0.74
MXnet	-0.99
Pytorch	-1.02

By observing the coefficients, the Pytorch and MXnet framework show better scaling performance than TensorFlow. As shown in Table II, -1 indicates ideal scaling, in which case the time is inversely proportional to the number of GPUs. In TensorFlow, the value -0.73 indicates sub-optimal scaling.

IV. Conclusion

In this work, we have developed a generic performance model for global optimisation problem using a differential evolution algorithm that gives insights into internal processes' performance and scaling factors and evaluated on three deep learning frameworks i.e., TensorFlow, MXnet and Chainer. The experimental results show that the proposed method can be applied to any distributed deep learning framework without instrumenting the code.

References

[1] F. Yan, O. Ruwase, Y. He, and T. Chilimbi, "Performance modeling and scalability optimization of distributed deep learning systems," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2015, pp. 1355–1364.

[2] H. Qi, E. R. Sparks, and A. Talwalkar, "Paleo: A performance model for deep neural networks," 2016.

[3] S. Shi, Q. Wang, and X. Chu, "Performance modeling and evaluation of distributed deep learning frameworks on gpus," in *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*. IEEE, 2018, pp. 949–957.

[4] J. Woo, H. Choi, and J. Lee, "Empirical performance analysis of collective communication for distributed deep learning in a many-core cpu environment," *Applied Sciences*, vol. 10, no. 19, p. 6717, 2020.

[5] D. Didona, F. Quaglia, P. Romano, and E. Torre, "Enhancing performance prediction robustness by combining analytical modeling and machine learning," in *Proceedings of the 6th ACM/SPEC international conference on performance engineering*, 2015, pp. 145–156.

[6] R. Storn and K. Price, "Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces," *J. of Global Optimization*, vol. 11, no. 4, pp. 341–359, dec 1997. [Online]. Available: https://doi.org/10.1023/A:1008202821328

Appendix C

Paper: A Generic Performance Model for Deep Learning in a Distributed Environment

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

A Generic Performance Model for Deep Learning in a Distributed Environment

TULASI KAVARAKUNTLA[1], LIANGXIU HAN[1]*, HUW LLOYD, MIEEE[1], ANNABEL LATHAM, SMIEEE[1], ANTHONY KLEEREKOPER[1], AND SAMSON B. AKINTOYE[1]

[1]Department of Computing and Mathematics, Manchester Metropolitan University, UK (e-mail: Tulasi.Kavarakuntla@stu.mmu.ac.uk; l.han@mmu.ac.uk; Huw.Lloyd@mmu.ac.uk; A.Latham@mmu.ac.uk; Akleerekoper@mmu.ac.uk; s.akintoye@mmu.ac.uk)

* Corresponding author: L. Han (e-mail: l.han@mmu.ac.uk).

ABSTRACT Performance modelling of a deep learning application is essential to improve and quantify the efficiency of the model framework. However, existing performance models are mostly case-specific, with limited capability for the new deep learning frameworks/applications. In this paper, we propose a generic performance model of an application in a distributed environment with a generic expression of the application execution time that considers the influence of both intrinsic factors/operations (e.g. algorithmic parameters/internal operations) and extrinsic scaling factors (e.g. the number of processors, data chunks and batch size). We formulate it as a global optimization problem and solve it using regularization on a cost function and differential evolution algorithm to find the best-fit values of the constants in the generic expression to match the experimentally determined computation time. We have evaluated the proposed model on three deep learning frameworks (i.e., TensorFlow, MXnet, and Pytorch). The experimental results show that the proposed model can provide accurate performance predictions and interpretability. In addition, the proposed work can be applied to any distributed deep neural network without instrumenting the code and provides insight into the factors affecting performance and scalability.

INDEX TERMS Deep Learning, Performance modelling, Optimization, Differential evolution.

I. INTRODUCTION

Deep neural networks are effective tools for unsupervised data exploration to discover correlation structures. As a result, they are widely used in computer vision, self-driving cars, medical image analysis, video games, and online self-service applications. However, deep neural network architectures such as GoogLeNet [1], ResNet [2], VGG Net [3], and Deep Convolutional Neural Networks (CNN) [4] necessitate the use of high computational resources. Training with a large amount of data requires a parallelised and distributed environment, primarily data parallelism, model parallelism, pipeline parallelism, and hybrid parallelism. Performance modelling is essential in quantifying the efficiency of large parallel workloads. Performance models are used to obtain run-time estimates by modelling various aspects of an application on a target system. However, accurate performance modelling is a challenging task. Existing performance models are broadly categorised into two catergories: analytical modelling and empirical modelling. Analytical modelling uses a transparent approach to convert a model's or an application's internal mechanism into a mathematical model corresponding to the system's goals, which can significantly expedite the creation of a performance model for the intended system. The existing analytical modelling works investigated deep learning performance modelling and scaling optimization in distributed environments [5], asynchronous GPU processing based on mini-batch SGD [6], efficient GPU utilisation in deep learning [7], comprehensive analysis and comparison of the performance of deep learning frameworks running on GPUs [8], [9]. However, the major limitation of these works is the poor presentation of the underlying internal operations (i.e., areas of the features' space or specific workload conditions) in the distributed environment.

Empirical modelling is a good alternative to analytical models, which predicts the outcome of an unknown set of system parameters based on observation and experimentation. It characterises an algorithm's performance across problem instances and parameter configurations based on sample data. Existing works investigated deep convolutional neural networks using asynchronous stochastic gradient descent techniques in a distributed environment [6]. Nevertheless, the existing empirical modelling methods are still facing a

challenge on how to provide an unbiased experimental study in a distributed environment using GPUs.

Thus, inspired by the hybridisation of the analytical and empirical approaches, this paper proposes a novel generic performance model that provides a general expression of intrinsic and extrinsic factors of a deep neural network framework in a distributed environment for accurate performance prediction and interpretability. Especially, the proposed model is applicable to any to any distributed deep neural network without instrumenting the code, which furthermore allows for explaining intrinsic parameters' performance and scalability, providing added value in the field. Our contributions include the following:

- We have developed a generic expression for a performance model considering the influence of intrinsic parameters and extrinsic scaling factors that affect computing time in a distributed environment.
- We have formulated the generic expression as a global optimization problem using regularization on a cost function in terms of the unknown constants in the generic expression, which we have solved using differential evolution to find the best fitting values to match experimentally determined computation times.
- We have evaluated the proposed model in three deep learning frameworks, i.e., TensorFlow, MXnet, and Pytorch, to demonstrate its performance efficiency.

The remainder of the paper is organised as follows. Section II discusses related work of the existing performance models in a distributed environment. In section III, we discuss research methodology, e.g., problem description, the proposed performance model. Section IV discusses an experimental evaluation of the effectiveness of our proposals. Finally, section V concludes the paper and highlights the future works.

II. RELATED WORKS

This section provides an overview of the existing performance models in a distributed computing environment and differential evolution as a solution to the optimization problem.

A. EXISTING PERFORMANCE MODELS

Performance modelling involves prediction of the performance of a system, the impact of change on an existing system, or the impact of a change in workload on an existing system [10], [11]. Existing performance modelling of Deep Learning (DL) frameworks can be broadly divided into two categories: 1) Analytical modelling and 2) Empirical modelling.

1) Analytical Performance Modelling of DL frameworks

Yan et al. [5] developed a performance model to evaluate the effect of the partitioning and resourcing decisions on the distributed system architectures' overall performance and scalability using a DL framework Adam [12]. In addition, the performance model was also used to guide the development of a scalability optimizer that quickly selects the optimal system configuration for reducing DNN training time. However, the model can only be applied to specific DL systems, particularly when it has parameter servers and synchronous weights between worker nodes dynamically.

Qi et al. [13] developed an analytical performance model named Paleo, predicting the deep neural network performance by considering communication schemes, network architecture and parallelization strategies. The results demonstrated that hybrid parallelism performed much better than data parallelism while training the Alexnet model. However, the model did not consider other factors affecting the overall performance of a model, such as memory usage, data transfer, or communication overhead in distributed environments.

Heehoon Kim et al. [14] evaluated five popular deep learning frameworks TensorFlow [15], CNTK [16], Theano [17], Caffe-MPI [18] and Torch [19] in terms of their performance in both single and multi-GPU environments. In this work, each framework incorporated and compared different convolution algorithms, such as Winograd, General Matrix Multiplication (GEMM), Fast Fourier Transformation (FFT), and direct convolution algorithms, in terms of layered-wise analysis and execution time. The results have shown that FFT and Winograd algorithms surpass the GEMM and other convolution algorithms. However, the convolution algorithms used by the frameworks provided poor explainability regarding their internal operations.

Shi et al. [9] proposed a performance model to evaluate various distributed deep learning frameworks' performance with different convolutional neural networks in the multi-GPU environment. They measured training time, memory usage, and GPU utilization and compared the frameworks in terms of training time and resource utilization. However, they did not provide a breakdown of the time to divide the mini-batch into smaller batches or measure the load imbalance factor, which are critical factors that could significantly affect the training efficiency and performance in a parallel computing environment. Kavarakuntla et al. [20] extended Shi's analytical performance model to evaluate the deep learning frameworks' run-time performance with the autoencoder, multilayer perceptron and convolutional neural network models in the GPU cluster environment. The extended model considered the load imbalance factor and made a layer-wise analysis of a neural network, providing a more comprehensive evaluation of the frameworks' performance. The experimental results showed that the load balance is an influential factors affecting the system performance.

However, the models mentioned above have poor explainability and were developed for specific architectures, which were not generic and couldn't be applied to a wide range of neural networks.

2) Empirical Modelling of DL frameworks

Empirical modelling builds models through observation and experimentation, which is antithetical to analytical modelling.

Oyama et al. [6] proposed a performance model for predicting the statistics of an asynchronous stochastic gradient descent-based deep learning system, potentially improving the model's performance by optimizing the hyperparameters, such as gradient staleness and mini-batch size. They did not consider parallelization methods and applied direct weights synchronized among GPUs. The study results showed that the proposed method could predict the statistics of asynchronous SGD parameters, including mini-batch size, sweeping dataset time, staleness, and probability distributions of these essential parameters. However, the work did not address the issue of communication overhead and network latency, which could significantly affect the performance of distributed deep learning systems.

Rakshith et al. [21] presented an empirical study of the performance of Horovod, a distributed deep learning framework, for image classification tasks. They evaluated the performance of Horovod on two popular image datasets, CIFAR-10 and ImageNet, using a cluster of machines with varying numbers of GPUs. They also compared the performance of Horovod with other distributed deep learning frameworks, such as PyTorch and TensorFlow, and found that Horovod achieved better performance in certain scenarios. They provided recommendations for optimizing the performance of Horovod on large-scale image datasets, such as using efficient data loading and preprocessing techniques, and optimizing the communication and synchronization between the machines. However, the experimental configuration utilized in the research might not accurately reflect real-world situations in which the underlying hardware and network setups may differ substantially.

Lin et al. [22] considered the network topology and communication patterns to train deep learning models on GPU clusters. The model included the communication and computation times for each layer in a deep neural network and used a prediction model that is more sophisticated than a simple linear regression approach to predict the total training time. The model was evaluated on several deep learning benchmarks and showed that it achieved higher accuracy in predicting training time than existing models. The model could also be used to optimize the performance of distributed deep learning by finding the optimal configuration of GPU nodes and reducing the training time. However, the assessment of the proposed model was confined to three distinct GPU clusters, potentially limiting its generalizability to other GPU clusters or distributed DL architectures.

Most recently, inspired by the concept of combining elements of analytical modelling and empirical modelling for better performance prediction developed in other fields [23], we [24] developed a generic performance model for deep learning applications in distributed environments, which offers the advantage of applicability to various deep learning frameworks. However, its performance is sub-optimal and lacks comprehensive analysis and experimental evaluation.

To address the above limitations, in this paper, we have implemented the model that gives insights into the intrinsic parameters' performance and scalability of the extrinsic parameters for more accurate performance prediction. Our proposed model in this paper provides a generic expression applicable to any distributed deep neural network without instrumenting the code and enabling functionality such as explaining internal parameters' performance and scalability. The detailed method is described in section III below.

B. DIFFERENTIAL EVOLUTION

Differential Evolution (DE) was developed by Storn et al. [25] as an algorithm to solve various complex optimization problems such as motor fault diagnosis [26], structure prediction of materials [27], automatic clustering techniques [28], community detection [29], learning applications [30] and so on. The algorithm achieves optimal solutions by maintaining a population of individual solutions and employing a distinct process to generate new offspring through the combination of existing solutions. Those offspring exhibiting superior objective values are retained in subsequent iterations of the algorithm, thereby enhancing the individual's new objective value and subsequently assimilating them into the population. Conversely, if the newly acquired objective value fails to surpass existing solutions, it is promptly disregarded. This iterative process continues until a specific termination condition is met, ensuring the algorithm's convergence [31]. It shares similarities with other evolutionary algorithms, such as the Genetic Algorithm (GA) [32], wherein mutation, crossover, and selection operators are employed to steer the population towards increasingly favourable solutions. In contrast to the genetic algorithm, the differential evolution algorithm imparts mutation to each individual while transferring them to the next generation. In its mutation procedure, for each solution, three more individuals are picked from the population, and as a consequence, a mutated individual is produced. It is determined based on the fitness value whether or not the first individual selected will be replaced. In differential evolution, the crossover is not the primary operation, as it is in the genetic algorithm. In recent times, several works have been proposed to use DE for neural network optimization [33], [34], [35].

However, none of the works mentioned above used DE to analyse and evaluate the performance of deep neural networks with many processes in a distributed environment with the goal of finding the best-fit values by minimising the regularised cost function.

III. METHODOLOGY

A. THE GENERIC PERFORMANCE MODEL

Given an application consisting of a number of processes in a distributed environment, the execution time of the application can be considered from two levels: 1) Execution time of internal processes of the application (for example, intrinsic parameters of the application); and 2) External scaling factors that affect the computing efficiency (such as a number of machines/processors or data chunks or batch size). A generic

performance model for computing total computational time (t) per iteration of an application can be described as follows:

$$t(I, E) = t_I(I) f_E(E) + C \tag{1}$$

Here, we represent intrinsic parameters as I, E represents the extrinsic parameters, t_I represents the computation time of the processes affected by intrinsic parameters, f_E represents extrinsic scaling factors that affect the computing performance, and C is a constant. In general, I and E are vectors in which each element is a hyperparameter of the deep learning model such as a filter size (intrinsic) or batch size (extrinsic).

In our model, we represent the internal time t_I as a sum of terms in powers of the components of I:

$$t_I = \sum_{i=1}^{n} a_i I_i^{p_i} \tag{2}$$

Basically, intrinsic parameters represent model parameters of the deep neural network, as shown in Fig.1. In equation (2), the coefficients a_i relate to the relative importance of the processes, and the powers p_i relate to the computational complexity. The external factors are related to scaling, and these appear in the model as multiplicative terms with different powers in the computation of the external scaling factor f_E, which is given by:

$$f_E = \prod_{j=1}^{m} E_j^{q_j} \tag{3}$$

Here, the powers q_j give information about scalability. By substituting the t_I and f_E in equation (1), the computational time (t) is given as follows:

$$t(I, E, x) = \left(\sum_{i=1}^{n} a_i I_i^{p_i} \right) \prod_{j=1}^{m} E_j^{q_j} + C \tag{4}$$

which we now write as a function of I, E and x where $\boldsymbol{x} = \{a_1, ..., \ a_{n_I}, p_1, ... p_{n_I}, q_1, ... q_{n_E}, c\} \in \mathbb{R}^M$ is a vector formed by combining $\boldsymbol{a}, \boldsymbol{p}, \boldsymbol{q}$ and the constant coefficient $\boldsymbol{C}$. In (4), the intrinsic parameters I and extrinsic parameters E are the known input values. $\boldsymbol{a}, \boldsymbol{p}, \boldsymbol{q}$ and coefficient $\boldsymbol{C}$ are unknown constants. The functional diagram of the proposed performance model is shown in Fig. 2.

We compute the optimal values of these unknown constants (total: $M = 2n_I + n_E + 1$) using the differential evolution algorithm. The aim is to find the best-fitting values of these constants, by fitting the model to experimentally measured execution times obtained with different values of the internal and external parameters I and E. Before going to the cost function formulation of the differential evolution algorithm [36], we describe the general methodology for obtaining the experimental data. For every possible combination of values of intrinsic and extrinsic parameter there will, in general, be too many combinations for an exhaustive grid search. Therefore, we have applied random sampling to ensure that every hyperparameter in the population has an equal opportunity of being selected for obtaining measured times. Here, the methodology used for measured time is the time taken for an iteration of an epoch. We compute the iteration time as the difference between an iteration's end time and starting time.

The experimental data for fitting the model comprises N measurements with randomly selected values of the parameters. We denote the values of the intrinsic parameters by

$$I_{i,k}, \quad i \in [1, n_I], \ \ k \in [1, N] \tag{5}$$

where i indexes the components of the vector of parameters, and k refers to a given observation in the experiment. Similarly, the extrinsic parameters are denoted by

$$E_{j,k}, \quad j \in [1, n_E], \ \ k \in [1, N] \tag{6}$$

The measured time for obervation k is

$$t_k, \ \ k \in [1, N]. \tag{7}$$

Here, i, j denote the input feature indices. $k \in [1, N]$ indicate the sample index in dataset $\boldsymbol{D}$. N is the number of input samples in $\boldsymbol{D}$.

B. GLOBAL OPTIMIZATION USING DIFFERENTIAL EVOLUTION

Given the generic expression as shown in equation (4), as mentioned in earlier sub section, we find the best fit values of $\boldsymbol{a}, \boldsymbol{p}, \boldsymbol{q}$ and C by minimizing a cost function. We formulate the cost function as the mean absolute difference between the predicted execution time and the actual measured times as follows:

$$f(x) = \frac{1}{N} \sum_{k=1}^{N} \left| t_k - \hat{t}_k(I_k, E_k, x) \right| \tag{8}$$

where N as number of data samples, t_k is the measured time, $\hat{t}_k$ is the predicted time derived from equation 4.

To solve the above optimization problem, we have used the differential evolution algorithm (DE) using the cost function in equation 4. The mean absolute error between the predicted times of the model and the measured times is minimized, resulting in a value of the vector x which represents the best-fitting model. Recall that this vector encodes the coefficients and powers of the terms in the model due to each of the hyperparamters; these can then be used to make predictions of the execution time for any set of values of the intrinsic and extrinsic paramters and furthermore, inspection of these coefficients can provide insight into the relative importance and computational complexity of the internal processes, as well as the scalability of the external processes. For this work, we use the DE implementation from the scipy python package, with default values of the hyperparameters. We enforce limits of $(0 \ldots 1000)$ for constants and coefficients (a, C) and $-5 \ldots 5$ for powers (p, q).

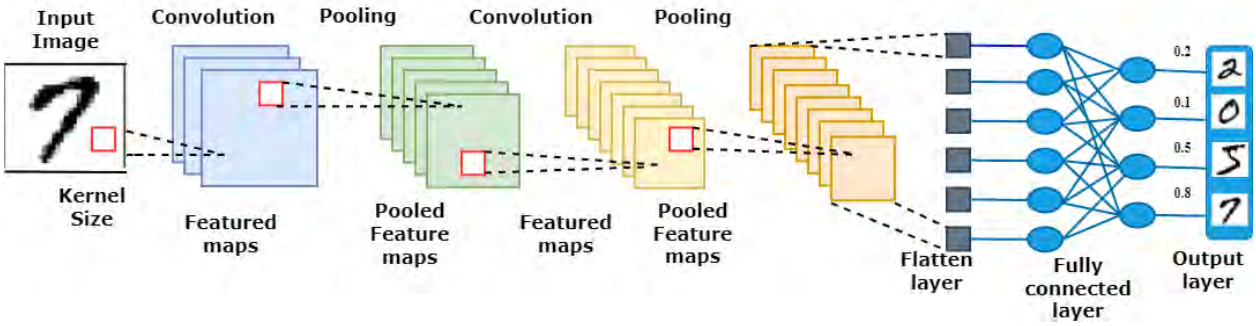


FIGURE 1: Internal processes involved in a convolutional neural network.

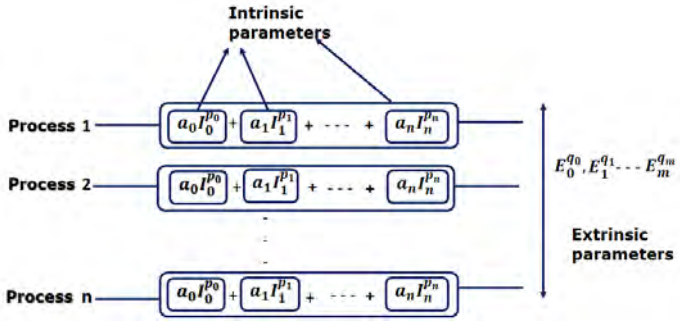


FIGURE 2: Functional diagram of proposed performance model.

C. REGULARIZATION

A globally optimized, unconstrained model may be prone to overfitting or producing unstable solutions with high parameter variance. To address these issues, we introduce a regularization term to the cost function. Regularization achieves the best fit by introducing a penalizing term in the cost function, which assigns a higher penalty to complex curves. So, we are motivated to apply regularization to our performance model. Generally, regularization can be defined as:

$$f_{\text{reg}}(x) = f(x) + \lambda.L \tag{9}$$

The parameter λ controls the balance between bias and variance in the model, where L represents the model's complexity. There are two regularization techniques: (a) Lasso regression (L1) and (b) Ridge regression (L2). L1 regularization, also known as lasso regression, includes the absolute value of the coefficient magnitude as a penalty in the loss function. The resulting solution from L1 regularization is sparse, meaning it tends to eliminate less important features by setting their coefficients to zero. This is useful for feature selection when dealing with a large number of features. On the other hand, L2 regularization, or ridge regression, incorporates the squared magnitude of the coefficient as a penalty in the loss function. The solution obtained from L2 regularization is non-sparse and penalizes the model's complexity. The regularization parameter λ penalizes all parameters except the intercept, ensuring that the model generalizes the data and avoids overfitting. Ridge regression uses the squared magnitude of the coefficient as the penalty term. We have applied both L1 and L2 regularizations to the performance model. The cost function for optimization with both L1 and L2 regularizations is as follows:

$$f(x) = \frac{1}{N}\sum_{k=1}^{N}\left|t_k - \hat{t}_k(I_k, E_k, x)\right| + \lambda.\sum_{k=1}^{N}|x| \tag{10}$$

$$f(x) = \frac{1}{N}\sum_{k=1}^{N}\left|t_k - \hat{t}_k(I_k, E_k, x)\right| + \lambda.\sum_{k=1}^{N}|x|^2 \tag{11}$$

Here, applying the regularization term λ reduces the bias-variance trade-off in the internal processes.

IV. EXPERIMENTAL EVALUATION

To evaluate the performance of the proposed model, we have applied our approach to three popular deep learning frameworks (TensorFlow, PyTorch and MxNet) and conducted extensive experiments. We assess the proposed performance evaluation approach by modelling distributed training of a CNN architecture on a multi-GPU system. The main goal is to investigate how well the predicted execution time fits the experimentally measured time.

A. SYSTEM CONFIGURATION

We implement the experiments on a single node containing three GEFORCE RTX 2080 GPUs, each with 2.60 GHz speed and 16 GB GPU RAM. The node also consists of a 2.81 GHz speed CPU machine, 25 Gbps network bandwidth and a CUDA-10.2 with a Linux operating system. Furthermore, the node consists of various software configurations/installations, including PyTorch 1.2.0, Torchvision 0.4.0, Python 3.6, TensorFlow 2.1.0 and MXnet 1.6.0.

B. DATASET AND MODEL SELECTION

For three deep learning frameworks, we selected a CNN architecture, LeNet-5, which Yann LeCun proposed in 1998 as a general common neural network structure for handwritten font recognition. It consists of two convolutional layers, two fully-connected layers, pooled layers for cross-combination and an output layer that predicts values via the fully connected layer. Besides, LeNet-5 works well with handwritten

datasets [37], it also reduces the number of parameters and can automatically learn features from raw pixels [38].

We train LeNet-5 on three popular datasets, MNIST, fashion-MNIST and CIFAR-10, using TensorFlow, PyTorch and MxNet, in a multi-GPU system. MNIST [39], [40] is a database of handwritten digits derived by the National Institute of Standards and Technology (NIST) for learning techniques and pattern recognition methods with a little effort on pre-processing and formatting. It contains 60,000 training and 10,000 testing images, divided into four files: training set images, testing set images, training set labels and testing set labels. Each image has 28 x 28 pixels. Fashion-MNIST [41] replaces the MNIST, where each image has a 28x28 grayscale and is associated with a label from 10 classes. The CIFAR-10 dataset [42] comprises 60,000 images, classified into ten classes - aeroplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. Each image has 32×32 pixels, and each classification has 6,000 images.

C. PERFORMANCE METRICS

The scalability and the mean absolute percentage error (MAPE) are selected as performance metrics for run-time evaluation on three different frameworks. Scalability is measured in the powers of external parameters as shown in (3). The MAPE can be defined as:

$$MAPE = \frac{1}{N}\sum_{k=1}^{N}\frac{|t_k - \hat{t_k}|}{t_k} \tag{12}$$

where t_k = the measured value, $\hat{t_k}$ = the predicted value, n = the total number of data points.

The experimental evaluation aims to evaluate the proposed performance model and find the best-fit values using the differential evolution algorithm. The MAPE is used to evaluate the closeness of this fit and the quality of the performance model. The scaling parameters are used in our proposed model to evaluate the performance of the deep learning frameworks.

D. EXPERIMENTS

We have conducted a set of experiments to evaluate the proposed model from the following aspects:

1) Performance evaluation of deep learning frameworks using the proposed performance model with and without regularization. Specifically, we have applied the proposed performance model to three deep learning frameworks: TensorFlow, MXnet, and PyTorch, under two circumstances, with and without regularization.
2) Comparison of the proposed model with the existing black-box machine learning models. We have also compared our proposed model with two widely used models including Random Forest Regression [43] and Support Vector Machine [44], and demonstrated its performance and interpretability.

In our experiment, we have performed the distributed training of LeNet-5 on MNIST, fashion-MNIST and CIFAR-10 datasets using the three deep learning frameworks. The values of the experimental training parameters are created by applying random sampling on a set of intrinsic and extrinsic parameters and its corresponding average training time taken by a deep CNN architecture per iteration. Table 1 shows intrinsic and extrinsic parameters and their possible values. The intrinsic parameters are the model's hyperparameters, including kernel size, pooling size, activation function, etc. The number of GPUs and the batch size are extrinsic factors since these affect the scaling over multiple processes.

TABLE 1: Parameters of the performance model, with ranges of values sampled in the experiments.

Index	Name	Set of possible values considered
Intrinsic parameters		
1	Kernel size	{2,3,4,5}
2	Pooling size	{2,3,4,5}
3	Activation function	{Relu, Tanh, Sigmoid}
4	Optimizer	{Adam, SGD}
5	Image_dataset_name	{MNIST,Fashion-MNIST,CIFAR-10}
6	Number of filters	{4,8,16,32,64}
7	Learning rate	{0.1,0.01,0.001,10^{-4}, 10^{-5}, 10^{-6}}
8	Padding_mode	{valid, same}
9	Stride	{1,2,3}
10	Dropout probability	{0.2,0.5,0.8}
Extrinsic parameters		
11	Number of GPUs	{1,2,3}
12	Batchsize	{8,16,32,64,128}

The experiments involve several trials in which we measure the time for a single training iteration using randomly selected intrinsic and extrinsic parameter values. We conduct 1500 trials to prepare a dataset of 1500 data samples. For each sample, we run three iterations with the same parameter values, and take the median value of the measured time. The experimental data for 900 trials are used to fit our performance model or train the standard black-box models for comparison. The remaining 600 are used to test and validate models. Finally, the experimental parameters are used to build three performance evaluation models, such as the Differential evolution (DE) algorithm with and without using regularization models and two standard black-box models. We run each fit ten times with different random seeds to obtain the mean and standard deviation for each of our fitted parameters. The performance of these models and their corresponding results are explained in the subsequent subsections.

E. RESULTS AND ANALYSIS

This section shows the results of our proposed performance model for three popular deep neural networks, i.e., TensorFlow, MXnet, and PyTorch. We evaluate the performance model with and without regularization and compare it with standard black-box regression models such as Support Vector Machine and Random Forest Regressor (RF). Tables 2 and 3 compare intrinsic parameters and scalability in various frameworks with and without using regularization. Table 4 shows mean absolute percentage error values on predictions of the performance models using L1 and L2 regularization.

TABLE 2: Derived intrinsic and extrinsic parameters from the differential evolution-optimized performance models for the three deep learning frameworks. Parameters are given as the mean and standard deviation over ten fits. a and p represent coefficients and powers, respectively, of a term representing an intrinsic parameter, whereas q is power in a multiplicative term representing an extrinsic (scaling) parameter.

	Mxnet		**Pytorch**		**TensorFlow**	
Intrinsic parameters	a	p	a	p	a	p
Filter size	554.87 ± 311.73	-4.06 ± 0.53	423.36 ± 256.88	-2.88 ± 1.04	346.73 ± 216.24	-3.22 ± 0.78
Kernel size	10.57 ± 7.05	-4.10 ± 0.70	168.54 ± 123.27	-2.34 ± 1.82	54.78 ± 32.91	-4.00 ± 1.41
Pool size	18.08 ± 5.17	-4.21 ± 0.46	209.14 ± 186.87	-3.31 ± 0.92	79.45 ± 53.53	-3.48 ± 1.33
Learning rate	459.50 ± 258.52	3.68 ± 0.62	489.52 ± 221.63	3.21 ± 0.70	458.34 ± 278.03	3.26 ± 0.91
Stride	17.29 ± 6.12	-0.83 ± 0.23	140.64 ± 138.62	-0.63 ± 0.58	29.00 ± 14.54	-1.85 ± 0.90
Dropout probability	1.79 ± 0.75	2.24 ± 1.62	437.06 ± 184.32	1.80 ± 1.66	10.23 ± 9.51	1.87 ± 1.62
Same	2.50 ± 0.97	-	11.02 ± 5.09	-	6.14 ± 1.54	-
Valid	1.56 ± 0.96	-	0.77 ± 1.81	-	1.61 ± 2.24	-
Sigmoid	23.25 ± 10.23	-	475.92 ± 139.65	-	251.57 ± 122.01	-
Relu	21.90 ± 10.40	-	475.56 ± 137.27	-	255.93 ± 122.35	-
Tanh	23.14 ± 10.30	-	444.48 ± 138.25	-	254.28 ± 121.27	-
MNIST	35.75 ± 12.81	-	815.62 ± 69.44	-	232.24 ± 108.77	-
Fashion-MNIST	35.94 ± 12.75	-	815.68 ± 68.39	-	231.33 ± 109.93	-
CIFAR-10	18.57 ± 12.68	-	308.73 ± 53.32	-	124.56 ± 108.01	-
SGD	16.68 ± 10.10	-	361.65 ± 130.64	-	158.74 ± 109.25	-
Adam	16.85 ± 10.32	-	720.15 ± 123.99	-	168.55 ± 108.67	-
Extrinsic parameters	q		q		q	
Batchsize	-0.99 ± 0.003		-1.13 ± 0.01		-1.35 ± 0.08	
No. of GPUs	-0.99 ± 0.004		-1.029 ± 0.001		-0.74 ± 0.001	
Constant term	C		C		C	
	3.703 ± 0.017		12.677 ± 0.038		1.930 ± 0.122	

TABLE 3: Derived intrinsic and extrinsic parameters from the differential evolution-optimized performance models for the three deep learning frameworks using L2 regularization. Parameters are given as the mean and standard deviation over ten fits. a and p represent coefficients and powers, respectively, of a term representing an intrinsic parameter, whereas q is power in a multiplicative term representing an extrinsic (scaling) parameter.

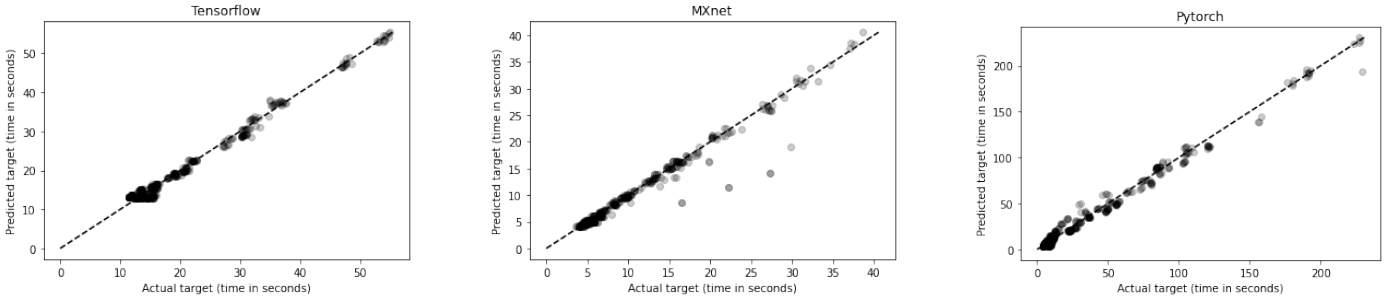
	Mxnet		**Pytorch**		**TensorFlow**	
Intrinsic parameters	a	p	a	p	a	p
Filter size	6.27± 0.59	0.36 ± 0.01	6.07 ± 1.59	0.89 ± 0.05	8.39 ± 0.37	0.77 ± 0.01
Kernel size	4.44 ± 0.65	0.50 ± 0.04	4.84 ± 1.90	2.02 ± 0.24	6.59 ± 0.29	2.04 ± 0.03
Pool size	4.69 ± 0.33	0.52 ± 0.03	3.23 ± 0.83	1.55 ± 0.45	6.70 ± 0.67	1.98 ± 0.05
Learning rate	3.62 ± 0.41	-0.04 ± 0.003	3.75 ± 1.70	-0.27 ± 0.02	4.40 ± 0.60	-0.22 ± 0.007
Stride	4.51 ± 0.40	-0.99 ± 0.11	2.92 ± 1.54	-0.83 ± 1.42	4.13 ± 0.59	2.46 ± 0.10
Dropout probability	4.20 ± 0.67	-0.35 ± 0.05	35.92 ± 1.15	-5.00 ± 0.00	4.46 ± 0.43	-1.94 ± 0.07
Same	2.66 ± 0.51	-	2.08 ± 0.84	-	1.90 ± 0.58	-
Valid	1.50 ± 0.43	-	-0.57 ± 1.56	-	0.49 ± 0.71	-
Sigmoid	2.18 ± 0.45	-	2.32 ± 1.15	-	1.41 ± 0.41	-
Relu	1.52 ± 0.33	-	3.21 ± 1.35	-	1.85 ± 0.64	-
Tanh	2.29 ± 0.39	-	2.93 ± 1.93	-	1.99 ± 0.71	-
MNIST	5.48 ± 0.52	-	3.37 ± 1.35	-	1.99 ± 0.72	-
Fashion-MNIST	7.73 ± 0.36	-	3.42 ± 1.56	-	2.28 ± 0.72	-
CIFAR-10	1.00 ± 0.02	-	1.89 ± 1.00	-	1.63 ± 0.69	-
SGD	2.31 ± 0.36	-	2.16 ± 1.00	-	1.73 ± 0.46	-
Adam	1.78 ± 0.41	-	3.42 ± 1.45	-	2.01 ± 0.85	-
Extrinsic parameters	q		q		q	
Batchsize	-0.87 ± 0.005		-1.00 ± 0.007		-1.19 ± 0.01	
No. of GPUs	-1.07 ± 0.007		-1.01 ± 0.004		-0.74 ± 0.005	
Constant term	C		C		C	
	3.45± 0.024		1.03 ± 0.07		12.62 ± 0.05	

TABLE 4: L1 and L2 regularization results in terms of the mean absolute percentage error (MAPE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE)

	L1 Regularization			L2 Regularization		
	MXnet	PYtorch	TF	MXnet	PYtorch	TF
MAPE	8%	29%	13%	7%	27%	10%
MSE	105.74	450.93	220.16	103.37	443.35	201.81
RMSE	10.28	17.52	14.83	10.16	17.02	14.20

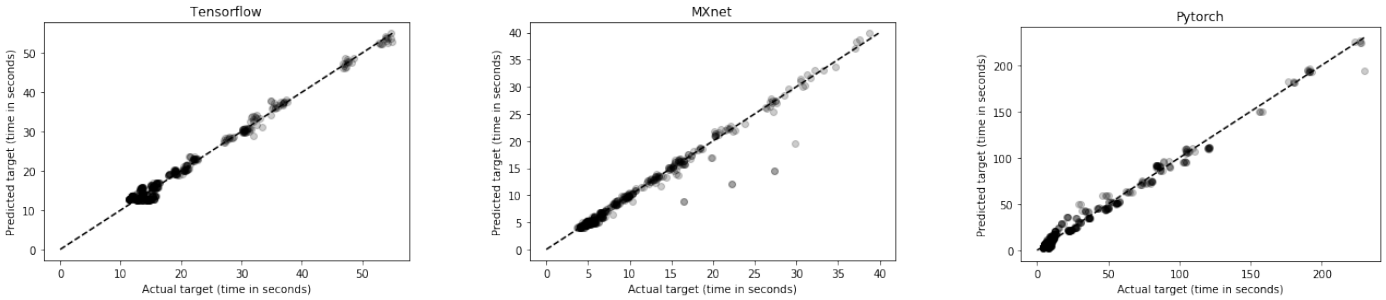
1) Performance Evaluation of Deep Learning Frameworks using the Proposed Performance Model without regularization

We have applied the differential evolution algorithm to our proposed model and evaluated it using the three deep learning frameworks. The actual execution time for training the model using the three frameworks is recorded, and predicted



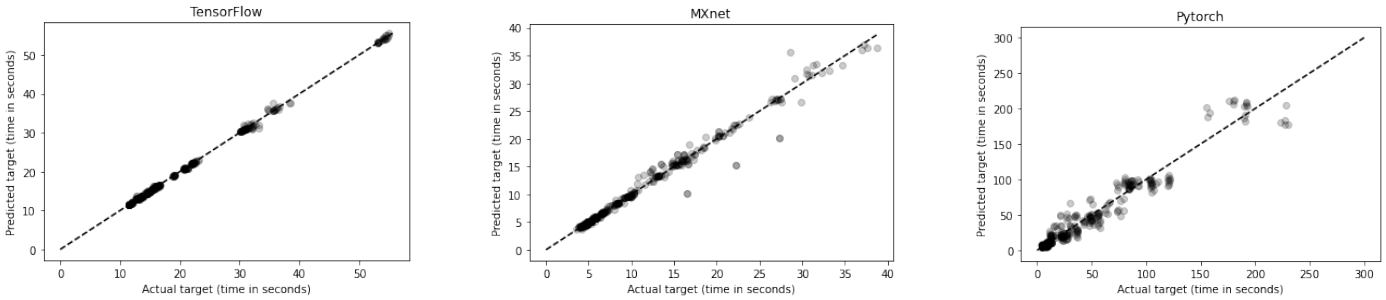
(a) TensorFlow (b) MXnet (c) Pytorch

FIGURE 3: The proposed performance model prediction and the actual measured times in three deep learning frameworks without regularization



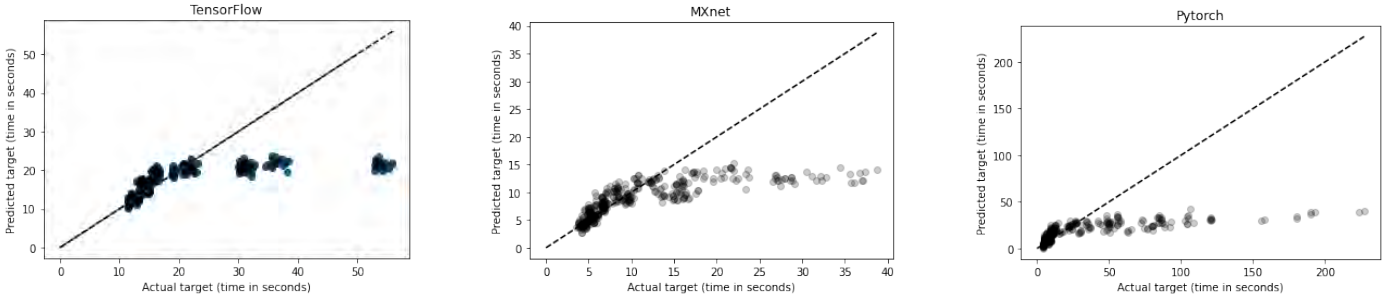
(a) TensorFlow (b) MXnet (c) Pytorch

FIGURE 4: The proposed performance model prediction and the actual measured time in three deep learning frameworks with regularization



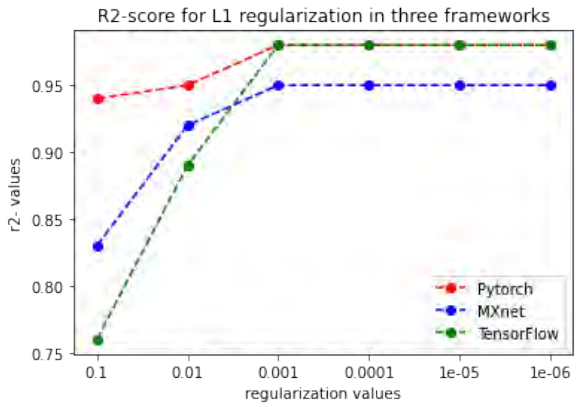
(a) TensorFlow (b) MXnet (c) Pytorch

FIGURE 5: Random forest regressor prediction and the actual measured time in three deep learning frameworks

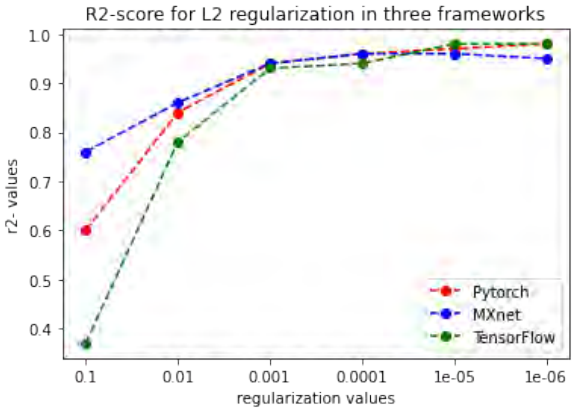


(a) TensorFlow (b) MXnet (c) Pytorch

FIGURE 6: Support vector machine prediction and the actual measured times in three deep learning frameworks

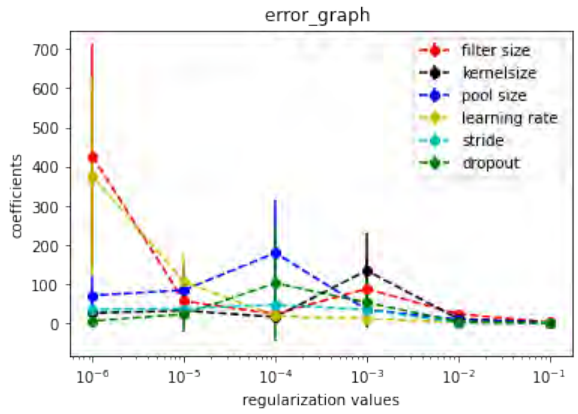


(a) R2 values with different regularization values in three different frameworks using L1 regularization

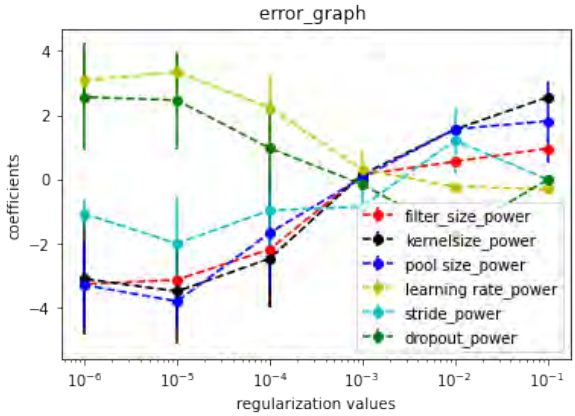


(b) R2 values with different regularization values in three different frameworks using L2 regularization.

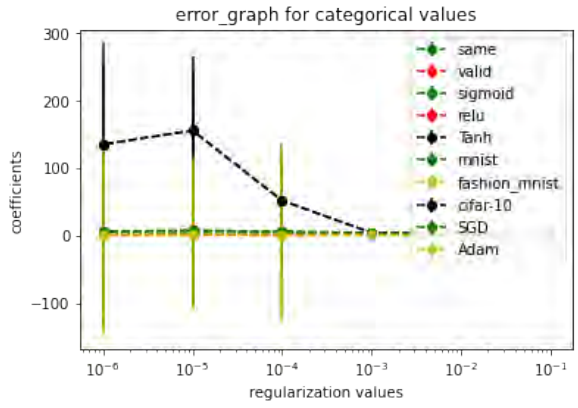
FIGURE 7: Effect of regularization.



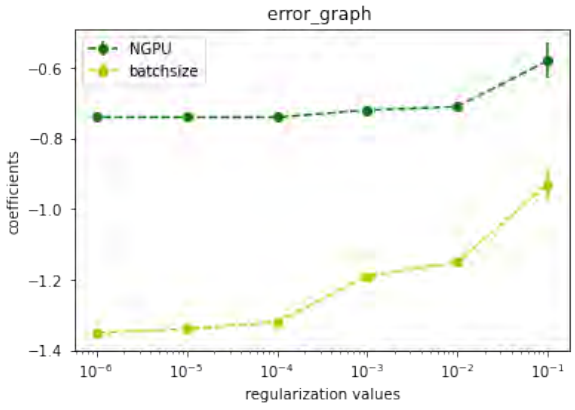
(a) TensorFlow



(b) MXnet



(c) Pytorch



(d) Pytorch

FIGURE 8: Effect of regularization, with model coefficients plotted against regularization parameter. Constant coefficients of intrinsic parameters are plotted in (a), the power coefficients of intrinsic parameters are shown in (b), coefficients of categorical intrinsic parameters in (c), with powers of extrinsic parameters in (d).

execution times are also generated. Fig.3 shows the scatter graph of the predicted execution times from the proposed model plotted against the actual execution time for the test dataset. The linear fit to the straight line determines how well the model can predict unseen configurations. We find the best fit constant coefficients for all frameworks are shown in Table 2.

The results show stable and consistent fits for the extrinsic

parameters and the additive constant C, indicating that the scalability results are accurate. The higher variances in the intrinsic parameters are reduced by using regularization. Table 2 shows that the model gives broadly consistent performance for the constant coefficients, representing the relative importance of the process controlled by categorical parameters. For instance, Adam has a large constant for the activation function coefficients and takes more training time than SGD in Pytorch and TensorFlow frameworks, while SGD has the highest training time with the MXnet framework. The padding parameter, which is categorical with two possible values valid and same. same shows better performance for the valid mode.

2) Performance Evaluation of Deep Learning Frameworks using the Proposed Performance Model with regularization

We have applied regularization to the cost function to the proposed performance model to optimize the vector constants and reduce high variance in intrinsic parameters in three deep learning frameworks. We applied both regularizations to our model and compared the results of L1 and L2. The MAPE, MSE, and RMSE results are better with L2 regularization, as shown in Table 4. We therefore consider L2 regularization appropriate for our performance model and applied various regularization parameter values in logarithmic scale in L1 and L2 to find the best value of the λ parameter. In Figures 7(a) and 7(b), we see that the R2 score deteriorates when the λ value is higher than 0.001. For instance, when $\lambda = 0.001$, the model fits well, and the model gives broadly consistent performance for the constant coefficients and represents the relative importance of the process controlled by categorical parameters. Furthermore, in Table 3, we can see that the model gives consistent performance for the constant coefficients, representing the relative importance of the processes controlled by categorical parameters. The results show that the performance model using regularization is a generalised model with optimized good fits in all the frameworks. For example, for padding coefficients, same parameter takes more training time than valid parameter in all frameworks. For activation function coefficients, Tanh takes more training time than Relu and Sigmoid in MXnet and TensorFlow frameworks, while Relu takes maximum time with Pytorch. Also, in terms of dataset coefficients, the Fashion-MNIST dataset takes more training time than MNIST and CIFAR-10 datasets in all three frameworks.

3) Comparison of the Proposed Performance Model with Black Box Models

We have compared the proposed model with two standard black box models, i.e., Random Forest Regressor and Support Vector Machine. Generally, the random forest regressor has better prediction accuracy due to its ensemble learning shown in Fig.5. The result shows a good linear fit compared to the differential evolution algorithm with and without regularization. However, the drawback of the random forest regressor is that it cannot give any insights into its internal working mechanism. Support vector machine regression is a non-parametric technique because it depends on kernel functionality. It is more productive in high-dimensional spaces. Fig.6 shows the predicted and measured times of the support vector machine. The result shows a poor fit for all the deep learning frameworks compared with the random forest regressor and differential evolution algorithm with and without regularization. We evaluate the fits using the mean absolute percentage error between predicted execution time and actual times, as shown in Table 5. Note that the performance of our proposed model is slightly inferior to the random forest. However, the proposed model can provide insights into the internal behaviour and scalability, which are impossible with a black box model such as a random forest.

TABLE 5: Mean Absolute Percentage Error (MAPE) on predictions of the performance models on the 300 instances in the evaluation dataset in seconds.

	TensorFlow	**MXnet**	**Pytorch**
Differential evolution	5%	5%	12%
Differential evolution with regularization	10%	7%	14%
Random forest	0.7%	3%	23%
Support vector machine	16%	21%	54%

TABLE 6: nGPUs scaling power in various frameworks, nGPUs represent number of GPUs.

Frameworks	**nGPUs scaling power**
TensorFlow	-0.74
MXnet	-0.99
Pytorch	-1.02

4) Scalability Analysis

Observing the coefficients q from Table 2 and Table 3, where q is power in a multiplicative term representing an extrinsic parameter, we see that the extrinsic parameter coefficients are consistent in the proposed performance model with and without regularization. As shown in Table 6, -1 indicates ideal scaling, in which case the time is inversely proportional to the number of GPUs. The coefficients in Pytorch and MXnet frameworks show better scaling performance than TensorFlow. In TensorFlow, the value -0.73 is less than -1, indicating sub-optimal scaling.

V. CONCLUSION AND FUTURE WORKS

In this work, we have developed a generic performance model for deep learning applications in a distributed environment with a generic expression of the application execution time that considers the influence of both intrinsic and extrinsic factors. We also formulated the proposed model as a global optimization problem and solved it using regularization on a cost function and differential evolution algorithm

to find the best-fit values of the constants in the generic expression. The proposed model has been evaluated on three popular deep learning frameworks: TensorFlow, MXnet, and Pytorch, and shown to provide accurate performance predictions and interpretability. Also, the experimental results show that MXnet and Pytorch exhibit better scalability performance than TensorFlow. Furthermore, the proposed method with regularization has been found to optimize the vector constants and reduce high variance in intrinsic parameters. The model can be applied to any distributed deep learning framework without requiring any changes to the code and can offer insight into the factors affecting deep learning application performance and scalability. Future work may include evaluating the model's performance on various deep learning frameworks to assess its generalisation capability.

REFERENCES

[1] P. Ballester and R. M. Araujo, "On the performance of googlenet and alexnet applied to sketches," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[2] S. Targ, D. Almeida, and K. Lyman, "Resnet in resnet: Generalizing residual architectures," *arXiv preprint arXiv:1603.08029*, 2016.

[3] L. Wang, S. Guo, W. Huang, and Y. Qiao, "Places205-vggnet models for scene recognition," *arXiv preprint arXiv:1508.01667*, 2015.

[4] N. Aloysius and M. Geetha, "A review on deep convolutional neural networks," in *2017 international conference on communication and signal processing (ICCSP)*. IEEE, 2017, pp. 0588–0592.

[5] F. Yan, O. Ruwase, Y. He, and T. Chilimbi, "Performance modeling and scalability optimization of distributed deep learning systems," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2015, pp. 1355–1364.

[6] Y. Oyama, A. Nomura, I. Sato, H. Nishimura, Y. Tamatsu, and S. Matsuoka, "Predicting statistics of asynchronous sgd parameters for a large-scale distributed deep learning system on gpu supercomputers," in *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 2016, pp. 66–75.

[7] M. Song, Y. Hu, H. Chen, and T. Li, "Towards pervasive and user satisfactory cnn across gpu microarchitectures," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 1–12.

[8] S. Shi, Q. Wang, P. Xu, and X. Chu, "Benchmarking state-of-the-art deep learning software tools," in *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*. IEEE, 2016, pp. 99–104.

[9] S. Shi, Q. Wang, and X. Chu, "Performance modeling and evaluation of distributed deep learning frameworks on gpus," in *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*. IEEE, 2018, pp. 949–957.

[10] S. Pllana, S. Benkner, F. Xhafa, and L. Barolli, "Hybrid performance modeling and prediction of large-scale computing systems," in *2008 International Conference on Complex, Intelligent and Software Intensive Systems*, 2008, pp. 132–138.

[11] T. Fahringer, S. Pllana, and J. Testori, "Teuta: Tool support for performance modeling of distributed and parallel applications," in *International Conference on Computational Science*. Springer, 2004, pp. 456–463.

[12] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 571–582.

[13] H. Qi, E. R. Sparks, and A. Talwalkar, "Paleo: A performance model for deep neural networks," 2016.

[14] H. Kim, H. Nam, W. Jung, and J. Lee, "Performance analysis of cnn frameworks for gpus," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2017, pp. 55–64.

[15] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "{TensorFlow}: A system for {Large-Scale} machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[16] F. Seide and A. Agarwal, "Cntk: Microsoft's open-source deep-learning toolkit," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 2016, pp. 2135–2135.

[17] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky *et al.*, "Theano: A python framework for fast computation of mathematical expressions," *arXiv e-prints*, pp. arXiv–1605, 2016.

[18] A. A. Awan, K. Hamidouche, J. M. Hashmi, and D. K. Panda, "S-caffe: Co-designing mpi runtimes and caffe for scalable deep learning on modern gpu clusters," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2017, pp. 193–205.

[19] R. Collobert, S. Bengio, and J. Mariéthoz, "Torch: a modular machine learning software library," Idiap, Tech. Rep., 2002.

[20] T. Kavarakuntla, L. Han, M. Huw Lloyd, S. Annabel Latham, and S. B. Akintoye, "Performance analysis of distributed deep learning frameworks in a multi-gpu environment," in *2021 IEEE 20th Intl Conf on Ubiquitous computing and communications(IUCC-2021), The 4th Intl Conf on Data science and Computational Intelligence(DSCI-2021)*, 2021.

[21] R. Rakshith, V. Lokur, P. Hongal, V. Janamatti, and S. Chickerur, "Performance analysis of distributed deep learning using horovod for image classification," in *2022 6th International Conference on Intelligent Computing and Control Systems (ICICCS)*. IEEE, 2022, pp. 1393–1398.

[22] Z. Lin, X. Chen, H. Zhao, Y. Luan, Z. Yang, and Y. Dai, "A topology-aware performance prediction model for distributed deep learning on gpu clusters," in *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 2020, pp. 2795–2801.

[23] D. Didona, F. Quaglia, P. Romano, and E. Torre, "Enhancing performance prediction robustness by combining analytical modeling and machine learning," in *Proceedings of the 6th ACM/SPEC international conference on performance engineering*, 2015, pp. 145–156.

[24] T. Kavarakuntla, L. Han, H. Lloyd, A. Latham, A. Kleerekoper, and S. B. Akintoye, "A Generic Performance Model for Deep Learning in a Distributed Environment, " in *2022 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, 2022, pp. 191.

[25] R. Storn and K. Price, "Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces," *J. of Global Optimization*, vol. 11, no. 4, pp. 341–359, dec 1997. [Online]. Available: https://doi.org/10.1023/A:1008202821328

[26] C.-Y. Lee and C.-H. Hung, "Feature ranking and differential evolution for feature selection in brushless dc motor fault diagnosis," *Symmetry*, vol. 13, no. 7, 2021. [Online]. Available: https://www.mdpi.com/2073-8994/13/7/1291

[27] W. Yang, E. M. D. Siriwardane, R. Dong, Y. Li, and J. Hu, "Crystal structure prediction of materials with high symmetry using differential evolution," *Journal of Physics: Condensed Matter*, vol. 33, 2021.

[28] S. Saha and R. Das, "Exploring differential evolution and particle swarm optimization to develop some symmetry-based automatic clustering techniques: Application to gene clustering," *Neural Comput. Appl.*, vol. 30, no. 3, pp. 735–757, aug 2018. [Online]. Available: https://doi.org/10.1007/s00521-016-2710-0

[29] Y.-H. Li, J.-Q. Wang, X.-J. Wang, Y.-L. Zhao, X.-H. Lu, and D.-L. Liu, "Community detection based on differential evolution using social spider optimization," *Symmetry*, vol. 9, no. 9, 2017. [Online]. Available: https://www.mdpi.com/2073-8994/9/9/183

[30] M. Baioletti, A. Milani, and V. Santucci, "Learning bayesian networks with algebraic differential evolution," in *Parallel Problem Solving from Nature – PPSN XV*. Cham: Springer International Publishing, 2018, pp. 436–448.

[31] M. F. Ahmad, N. A. M. Isa, W. H. Lim, and K. M. Ang, "Differential evolution: A recent review based on state-of-the-art works," *Alexandria Engineering Journal*, vol. 61, no. 5, pp. 3831–3872, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S111001682100613X

[32] D. Liu, D. Hong, S. Wang, and Y. Chen, "Genetic algorithm-based optimization for color point cloud registration," *Frontiers in Bioengineering and Biotechnology*, vol. 10, 2022. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fbioe.2022.923736

[33] M. Baioletti, G. Di Bari, A. Milani, and V. Poggioni, "Differential evolution for neural networks optimization," *Mathematics*, vol. 8, no. 1, 2020. [Online]. Available: https://www.mdpi.com/2227-7390/8/1/69

[34] N. Ikushima, K. Ono, Y. Maeda, E. Makihara, and Y. Hanada, "Differential evolution neural network optimization with individual dependent mecha-

nism," in *2021 IEEE Congress on Evolutionary Computation (CEC)*, 2021, pp. 2523–2530.

[35] R. A. Venkat, Z. Oussalem, and A. K. Bhattacharya, "Training convolutional neural networks with differential evolution using concurrent task apportioning on hybrid cpu-gpu architectures," in *2021 IEEE Congress on Evolutionary Computation (CEC)*, 2021, pp. 2567–2576.

[36] K. Fleetwood, "An introduction to differential evolution," in *Proceedings of Mathematics and Statistics of Complex Systems (MASCOS) One Day Symposium, 26th November, Brisbane, Australia*, 2004, pp. 785–791.

[37] S. Park, J. Lee, and H. Kim, "Hardware resource analysis in distributed training with edge devices," *Electronics*, vol. 9, no. 1, p. 28, 2020.

[38] A. Khan, A. Sohail, U. Zahoora, and A. S. Qureshi, "A survey of the recent architectures of deep convolutional neural networks," *ArXiv*, vol. abs/1901.06032, 2019. [Online]. Available: http://arxiv.org/abs/1901.06032

[39] Y. LeCun, C. Cortes, and Christopher, "The mnist database of handwritten digits," 2020. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[40] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, pp. 141–142, 2012.

[41] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *ArXiv*, vol. abs/1708.07747, 2017.

[42] F. O. Giuste and J. C. Vizcarra, "Cifar-10 image classification using feature ensembles," *ArXiv*, vol. abs/2002.03846, 2020.

[43] T. K. Ho, "Random decision forests," in *Proceedings of 3rd international conference on document analysis and recognition*, vol. 1. IEEE, 1995, pp. 278–282.

[44] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.