UNIVERSITY OF CALIFORNIA,
IRVINE

Deep Learning Performance Optimization via Model Parallelization

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Engineering

by

Ahmad Albaqsami

Dissertation Committee:
Professor Nader Bagherzadeh, Chair
Professor Jean-Luc Gaudiot
Professor Chen-Yu (Phillip) Sheu

2019

DEDICATION

To my wife, Zainab, for all her patience and support
To my children, Abdullah, Mohammad, and Amatallah, for all their love
To my parents, Yasmeen and Salman, for all their encouragement

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Page

LIST OF ALGORITHMS

LIST OF ABBREVIATIONS

API	Application Programming Interface
AI	Artificial Intelligence
ANN	Artificial Neural Network
BO	Bayesian Optimization
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DL	Deep Learning
DNN	Deep Neural Network
HEFT	Heterogeneous Earliest Finish Time
GA	Genetic Algorithm
GBR	Gradient Boosting Regressor
GPU	Graphics Processing Unit
ML	Machine Learning
SGD	Stochastic Gradient Descent
TF	TensorFlow

ACKNOWLEDGMENTS

"The most complete gift of God is a life based on knowledge."

-Ali Ibn Abu Talib

Firstly and most importantly, I would like to thank my creator for all the blessing that are bestowed on me on a daily basis.

I want to thank my advisor, Professor Nader Bagherzadeh, for his guidance and support throughout the PhD years and believing in me. I would like to thank him for lifting me up and giving me the encouragement to persevere. I thank him for his wisdom and knowledge. I want to extend my thanks to my committee members Professor Jean-Luc Gaudiot and Professor Chen-Yu (Phillip) Sheu for taking the time out of their busy lives to provide their insightful feedback.

I would like to thank my wife Zainab Albaghli for her patience and amazing support. She held the fort taking care of our three wonderful kids, Abdullah, Mohammad, and Amatallah, while I pursued my PhD.

I would like to thank my parents for instilling in me the love of knowledge.

I want to thank my colleagues at the Advanced Computer Architecture Group for their insightful discussions and feedback. I would like to extend a special thank you to my labmate Maryam S. Hosseini for all the discussions and work we did together.

I would like to thank the support of The Public Authority for Applied Education and Training, Kuwait, for the opportunity to study at the University of California Irvine and pursue a PhD.

CURRICULUM VITAE

Ahmad Albaqsami

EDUCATION

Doctor of Philosophy in Computer Engineering **2019**
University of California, Irvine *Irvine, California*

Master of Science in Computer Engineering **2010**
University of Southern California *Los Angeles, California*

Bachelors of Science in Computer Engineering **2003**
Pennsylvania State University *University Park, Pennsylvania*

Bachelors of Science in Electrical Engineering **2003**
Pennsylvania State University *University Park, Pennsylvania*

RESEARCH EXPERIENCE

Graduate Research Assistant **2014–2019**
University of California, Irvine *Irvine, California*

TEACHING EXPERIENCE

Instructor **2010–2014**
The Public Authority for Applied Education and Training (PAAET) *Shuwaikh, Kuwait*

Professional EXPERIENCE

Telecom Engineer **2006–2008**
Nokia Networks *Murgab, Kuwait*

Telecom Engineer **2005–2006**
Motorola *Murgab, Kuwait*

Telecom Engineer **2003–2005**
Future Communications Company International *Farwaniya, Kuwait*

PUBLICATIONS

Ahmad Albaqsami, Maryam S Hosseini and Nader Bagherzadeh, "HTF-MPR: A Heterogeneous TensorFlow Mapper Targeting Performance using Genetic Algorithms and Gradient Boosting Regressors", in 2018 Design, Automation Test in Europe Conference Exhibition (DATE)

Zana Ghaderi, Nader Bagherzadeh and **Ahmad Albaqsami**, "STABLE: Stress-aware boolean matching to mitigate BTI-induced SNM reduction in SRAM-based FPGAs", in IEEE Transactions on Computers (Volume: 67, Issue: 1, Jan. 1 2018)

Arquimedes Canedo, Zhi Zhang, **Ahmad Albaqsami**, Jiang Wan and Mohammad Abdullah Al Faruque, "Maintaining the design intent in the synthesis of 3-d and 1-d system models using constraints", in IEEE Systems Journal (Volume: 12, Issue: 2, June 2018)

SOFTWARE

itermore `https://github.com/atinzad/itermore`
Python PyPI package implements grouping permutations

ABSTRACT OF THE DISSERTATION

Deep Learning Performance Optimization via Model Parallelization

By

Ahmad Albaqsami

Doctor of Philosophy in Computer Engineering

University of California, Irvine, 2019

Professor Nader Bagherzadeh, Chair

In recent years, machine learning (ML) and, more noticeably, deep learning (DL), have become increasingly ubiquitous. Applications of these technologies are being seen in many fields, including health care, manufacturing, and end-consumer services. In terms of deployment, deep neural networks (DNNs) are found in consumer devices, small internet-of-things devices, embedded in vehicles, and on a large scale in data centers and servers. The trend indicates that the use of DL in *smart* applications will continue to increase in the coming years.

As the name suggests, *learning* is an integral part of the functionality of DNNs, whether this learning takes place off-line before deployment, or happens in real time while the DNN is carrying out its assigned task. As part of the learning process, *training* is required to set the parameters, also known as *weights*, of the DNN in order to achieve high accuracy in the assigned task. Without training, the DNN is rendered useless, given that the parameters are not set correctly. It has been shown that this training process requires large amounts of data and a high number of training iterations for the DNN model to be effective. The weights are updated in each iteration based on the subset of the training data provided. The training process has proven to be a challenge given the long timescale involved. The amount of training data, the number of weights, and the computational complexity of updating those

weights are all factors that contribute to this challenge. One way to reduce training time is to allocate processes to a multitude of processors, thus achieving some sort of sub-optimal parallelism. One approach is to have this decision be carried out by ML or DL experts. The problem with this is the absence of concrete information to ensure the best decision is taken: the time it takes for a particular process to run on a particular processor, and the costs of inter-communication between processors, are in fact unknown. Even with the intuition of an expert in this domain, a sub-optimal solution that outperforms a single-processor use case is not achieved.

In this dissertation, a hybrid-based multi-step optimization framework is presented. The framework explores the vast design space of mapping processes to processors. The search and evaluation are conducted in real time while training the DNN. In the first stage of the framework we compare the algorithmic intuitive approach with the Bayesian optimization (BO) approach. In the second stage of the framework, we create a predictive function for the performance of a single iteration of training, comparing the accuracy of different predictive functions created by different ML algorithms. The developed predictive model is then used as a surrogate function when identifying the best mapping. This stage in the search applies genetic algorithms (GA). An adaptive feature is also presented and tested for responsiveness to any changes that affect the performance of the training in the system.

We also present heterogeneous earliest finish time (HEFT): a deterministic approach to mapping. In addition, we present the concept of *node splitting*, which refers to the *computational graph* of the DNN being split in order to accommodate a higher level of model parallelism. It is noted that this would also affect the accuracy of the DNN, since the *hyperparameters* are affected.

The framework and methodologies were evaluated in real, non-simulated systems using wall-clock time. The DNNs were built using Google's ML/DL library, *TensorFlow* (TF).

Chapter 1

Introduction

Machine learning (ML), and more recently *deep learning* (DL), have been utilized as powerful tools in many fields including computer vision [12], finance [31], recommender systems [15], search engines [53], and games [65]. ML is a dominant branch of artificial intelligence (AI) based on the idea that a system uses algorithms to *learn* from data, identify patterns, and make decisions, rather than being explicitly programmed via a *rule-based* approach. This paradigm shift in AI has required systems to become engaged in learning.

Deep neural networks (DNNs) and DL algorithms [59] are the latest iterations of ML toolsets. In their simplest form, they are inspired by the human brain, and thus they have been developed to mimic the human brain's *synapses* and *neurons*. The neurons in this case are the functions, while the synapses are the connections that carry the information from one neuron to the next. The strength of the connections, also known as the synaptic *weights*, dictate how the neural network performs and functions. The values of these weights in turn need to be learned via training. This is achieved based on *experience*, analogous to *data* in ML terms. The amount of time it takes to train a neural network model is very strongly correlated to the amount of data, the number of weights that need to be adjusted, and how

much adjustment is required to attain the target level of accuracy. This process needs to be accelerated. To address this problem, an understanding of what DNNs are, how they are represented in software, and how they are dealt with in hardware is required. In addition, we need to understand the learning process, as well as how DL algorithms in DNNs are different from rule-based algorithms.

1.1 Artificial Neural Networks

In its simplest form, an artificial neural network (ANN) consists of weights and neurons. The structure of an ANN consists of layers, where the output of the neurons of one layer connects to the neurons of the next layer. The directed connections are the weights, which are multiplied by the output of the neuron. A layer is a collection of neurons, and each neuron is an activation function that applies to the sum of all its inputs. It is also noted that the neurons in a particular layer all use the same type of activation function. An example of a three-layer ANN is shown in Figure 1.1.

Equation 1.1 shows the computation output of the three nodes in the hidden layer from Figure 1.1.

$$\begin{aligned} z_1 &= \sigma_1(x_1 w_{11}^{[1]} + x_2 w_{21}^{[1]}) \\ z_2 &= \sigma_1(x_1 w_{12}^{[1]} + x_2 w_{22}^{[1]}) \\ z_3 &= \sigma_1(x_1 w_{13}^{[1]} + x_2 w_{23}^{[1]}) \end{aligned} \tag{1.1}$$

In $w_{bc}^{[a]}$; a is the layer, b is the starting node and c is the end node. z_d is the resulting output from $\sigma_d(.)$. Equation 1.2 shows the the computation output of the model depicted in Figure 1.1.

$$\hat{y} = \sigma_2(z_1 w_{11}^{[2]} + z_2 w_{21}^{[2]} + z_3 w_{31}^{[3]}) \tag{1.2}$$

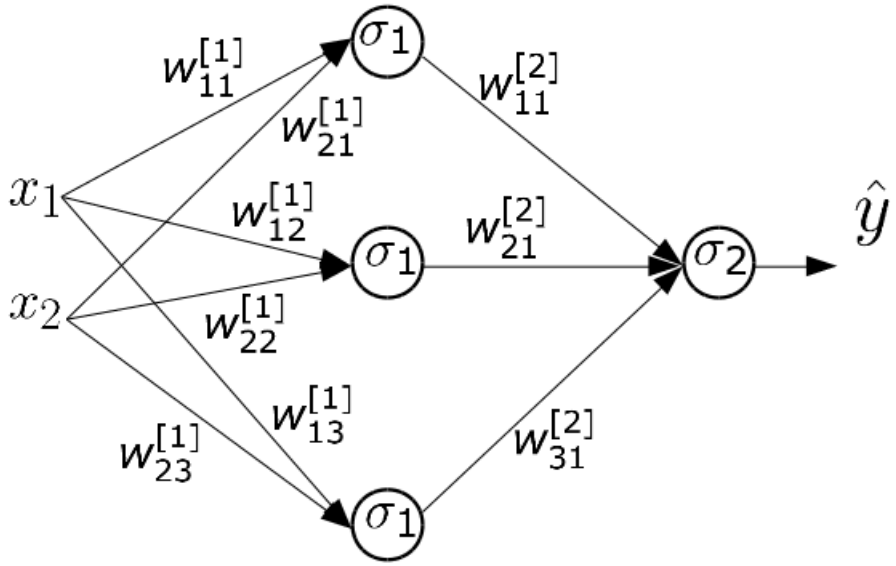


Figure 1.1: A simple three layer neural network. Input is a vector of size two containing x_1 and x_2 and output is a scalar $\hat{y}$. The activation functions are σ_1 in the hidden layer and σ_2 in the output. The computation of the model is shown in Equations 1.1 and 1.2.

An effectively functioning ANN would produce a $\hat{y}$ that is close or equal to y given any x. There are two requirements in this case: a well-structured computational model, and the correct parameter values. The computational model structure is static, and its components are referred to as *hyperparameters*. The parameter values are tunable while the hyperparameters are not. Learning is the process by which the parameters, in this case w, are tuned. Hyperparameters are not affected by the training process. See Figure 1.2, showing part of a DNN, where ws are tunable and the hyperparameters; size of layer, number of layers, and activation functions, are not.

1.2 Deep Learning

DL is the process of updating the weights, or parameters, of a DNN, resulting in the DNN being able to take in an input and predict an output that is the same as the expected output. This can be described as a mapping process from a set of inputs to a set of desired outputs.

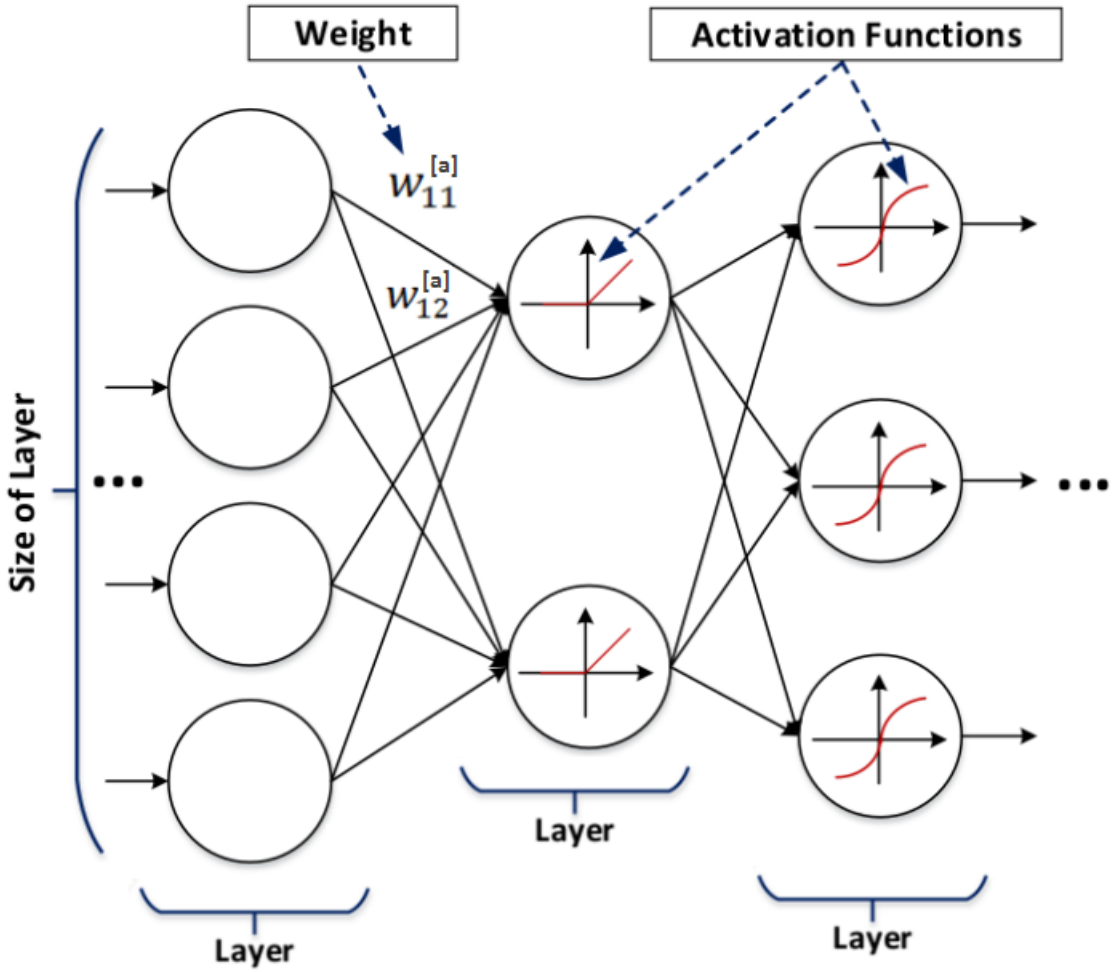


Figure 1.2: An intersection of a DNN. The value of each of the weights $w_{ij}^{[k]}$ may change during training. The number of nodes and type of activation function does not change.

In its most recognized form, learning requires a large dataset of input and output pairs. In each iteration, the DL process works to minimize the error between the predicted output and the desired output of a particular input. The weights are updated in response to the error observed between the desired and predicted outputs. This is an iterative process, as in each iteration the weights are updated in order to reduce the observed error.

Prior to *deploying* any ML model to be used for *inference*, the model should be trained. In the case of *supervised learning*, the input x and the intended output y are provided in order to subsequently produce a trained model f_{NN}. It is intended that for any given input data

point $x^{(i)}$ (where x is the input vector and (i) is a instance of data in the dataset where $i \in N_{dataset}$ and $N_{dataset}$ is the size of the dataset), $f_{NN}(x^{(i)}) \longrightarrow \hat{y}^{(i)}$ where the intention is $\hat{y}^{(i)} \approx y^{(i)}$. An untrained model would most likely produce a $\hat{y}$ that is not close to the intended y. To evaluate this discrepancy a loss function L is used. One such metric in DNNs is the *L2 norm* [18] loss function, shown in Equation 1.3.

$$L(y, \hat{y}) = \sum_{i=1}^{N} (y^{(i)} - \hat{y}^{(i)})^2 \tag{1.3}$$

In the case of *classification* problems, where the output is a category rather than a specific number, the *cross-entropy* function is used:

$$L(y, \hat{y}) = -\frac{1}{n} \sum_{i} ln\left(\frac{e^{y^{(i)}}}{\sum_j e^{\hat{y}^{(j)}}}\right) \tag{1.4}$$

The objective of training is to *find* weights that would result in a f_{NN} that is $L \approx 0$. The process of searching, via training, for the correct parameter values is complex and takes a long time. The search space of the error function is non-convex given the nature of the activation functions [11]. A well-known method for training DNNs is stochastic gradient descent (SGD) [11]. SGD provides the direction for $\hat{y}$ to be closer to y. The weights are changed, which affects $\hat{y}$. This change is brought about using a technique known as *backpropagation* [57]. Simply put, backpropagation adjusts the weights to produce a $\hat{y}$ closer to y. Therefore, in each training iteration, using a subset or *mini-batch* of input data x, L is assessed. Backpropagation updates the weights, and then the process is repeated. This whole process in DNNs is what is known as DL [46, 60]. The following Figures illustrate a single training iteration where backpropagation is used on an ANN with four layers: Figure 1.3 shows the structure of the ANN and the parameters (the weights ws that will be updated); Figure 1.4 shows, in three consecutive steps, the forward propagation; Figure 1.5 shows the error updates (also in three steps starting from the d_y), and lastly, Figure 1.6 shows the weight updates.

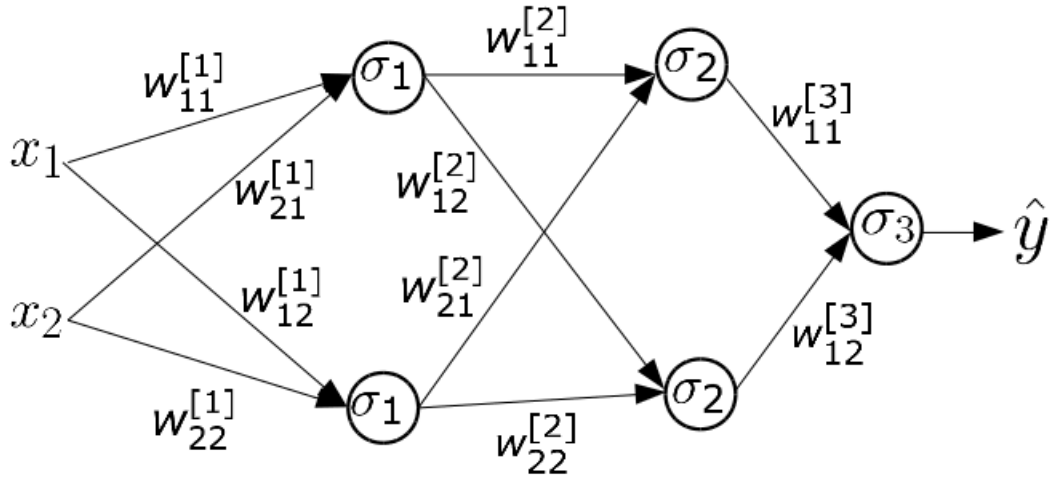


Figure 1.3: Four-layer ANN with input vector of size 2 (x_1 and x_2); number of weights is ten; three types of activation functions (σ_1,σ_2 and σ_3), and one output $\hat{y}$. Figures 1.4, 1.5, and 1.6 illustrate a single iteration of training where the weights, ws, are updated and modified.

The number of cycles, or training iterations, is related to many factors including the dataset size, the number of *features* per data point, the desired accuracy, etc. The duration of a single training iteration is correlated to the number of layers in the DNN and, less strongly, to the number of weights. The reason for the number of layers being a more significant factor in the duration of an iteration is that the update processes for all weights within a layer take place in parallel via *vectorization*. This parallelism is utilized within a single device (e.g. graphics processing unit [GPU]), assuming that the weights within a layer can fit into the devices memory.

1.3 ANN to Computational Graph using TensorFlow

A number of software libraries have been developed to assist practitioners in building and training DNNs. TensorFlow [3, 2] (TF), a computational graphs and numerical models library developed by Google, is a prominent example of such tools. The Application Programming Interface (API) makes it possible for data scientists and ML practitioners to build

and train large models with large datasets in a distributed system without prior knowledge of the hardware architecture. Keras [17], a high-level API, runs TensorFlow as its de facto library. Although it is easier and faster to develop DNNs in Keras, it does not allow for higher levels of control, such as assigning devices to specific operations. This has led to TF being the optimal choice when dealing with parallelism.

Figure 1.7 shows the pictorial translation of a simple ANN to a TF graph. TF uses the computational graph approach: each *node* in the TensorFlow graph G is an operation op_i, and the *vertices* are *tensors*; i.e., multi-dimensional matrices.

The API employed by TF makes it an easy task for the programmer to assign devices (GPUs, central processing units [CPUs], etc.) to operations. If no explicit assignment is made, all operations will be assigned to a single GPU, provided GPU-enabled TF is installed.

When it comes to DL, GPUs have been more desirable than Multi-core CPUs. The main usage of CPUs is to process complex computations given the architecture of CPUs, while GPUs contain many but simple cores that can handle many simple computations in parralel. Therefore when it comes to parallelism, high bandwidth and given the simple but many arithmetic computations, the default has been to use GPUs in DNNs and DL.

The GPU homogeneous *mapping* (where a single GPU is mapped to every operation in an ANN TF computational graph) is not optimal when it comes to performance. Even though a single GPU performs well when it comes to intra-parallelism, there is more potential for performance optimization when using multiple devices and multiple types of devices. How this mapping is supposed to operate is non-intuitive; thus, expert placed mappings will not always result in improved performance. This is due to the unknown costs that result from inter-communication costs of operations mapped to different devices. This, in turn, makes the performance of the ANN with respect to mapping a black box: we cannot deterministically predict the outcome of the ANNs performance.

1.4 Dissertation Contribution

Mapping operations to devices in a computational graph is an *NP-hard* problem [56, 40, 70]; thus, the challenge of finding a mapping that outperforms the homogeneous mapping is a non-trivial one. This dissertation provides a methodology that searches the design space and finds a mapping, or several candidate mappings, that could outperform homogeneous mapping. The methodology can succeed where an expert would fail, given the narrow candidate solution space. The methodology is graph-agnostic and system-agnostic, meaning that neither the types and number of devices, nor the structure and size of the graph, will affect the method of search: it will, therefore, depend on the observations made in each training iteration.

The time taken per training iteration is a restrictive aspect given the large exploration area of the design space. A way to mitigate this part of the methodology is to build a predictive model that is used as part of the exploration, rather than the search being completely dependent on observations. The construction of such a model and the analysis of different types of suitable models is analyzed in this dissertation.

In this dissertation, the different optimization methods are shown and analyzed in each stage of the method's pipeline.

Overall, this work provides a rich analysis of methods for searching a solution space for a sub-optimal operation to device mapping that best utilizes computational graph parallelization to speed up DNN training. This in return helps ML practitioners to speed up the training process with out affecting the accuracy of their DNN models.

The contributions of this dissertation can be summarized as follows:

- Presenting a methodology to concurrently train and search for improved mapping performance in a TF computational graph.
- Developing and presenting an algorithmic approach to mapping that is part of the larger pipeline.
- Incorporating a Bayesian optimization (BO) method to search for the best possible mapping.
- Developing an ML pipeline to build a predictive model for the execution time, or *makespan*, of the TF computational graph.
- Analyzing and showcasing different ML approaches to building an ML predictive model of the makespan.
- Incorporating Genetic algorithms (GA) with the predictive model to search for the best makespan.
- Analyzing different initial populations for the GA and their effects on the search.
- Presenting a self-correcting approach i.e. an *adaptive* approach, to correcting the mapping that reacts to system-level changes.
- Presenting a methodology for mapping using a greedy approach that incorporates the heterogeneous earliest finish time (HEFT) algorithm.
- Presenting a method for splitting computational nodes, to achieve higher parallelism and therefore provide more options for partitioning the graph.

1.5 Dissertation Organization

The dissertation is organized as follows. Chapter 2 summarizes the different methods for increasing the performance of DNNs. These performance enhancing methods will be categorized as those that affect accuracy and those that do not.

Chapter 3 presents the computational graph used in TF and the precursor to the optimization problem. This chapter also introduces the notations used to tackle the problem of mapping, makespan valuation, and the search for the mapping that produces the desired makespan.

Chapter 4 presents methods for generating mappings. Bayesian optimization (BO) and its modification for generating mapping is presented. The expert-intuition-centric approach of determining mappings is also discussed, and is referred to as the algorithmic approach. Results are presented and shown.

Chapter 5 presents the ML approaches and identifies the final incorporated ML approach. We present feature extraction, determining the accuracy of the predictors, and the motivations behind using certain ML algorithms to build the makespan predictor.

Chapter 6 describes our frameworks Heterogenous TensorFlow Mapper known as HTF-MPR, and it's succesor Adaptive HTF-MPR. How GA is used with the surrogate function is described, as well as the adaptivity feature that is used in Adaptive HTF-MPR. The results of HTF-MPR and Adaptive HTF-MPR conclude the chapter.

Chapter 7 presents the application of HEFT in determining the best mapping, laying out how to establish the inputs to HEFT. The limitations of HEFT are also discussed, as well as why HTF-MPR does not have such limitations.

Chapter 8 presents work carried out in pruning the DNN for the purpose of increasing the potential for parallelism. Recommendations for future work in the area of searching the mapping space are also discussed.

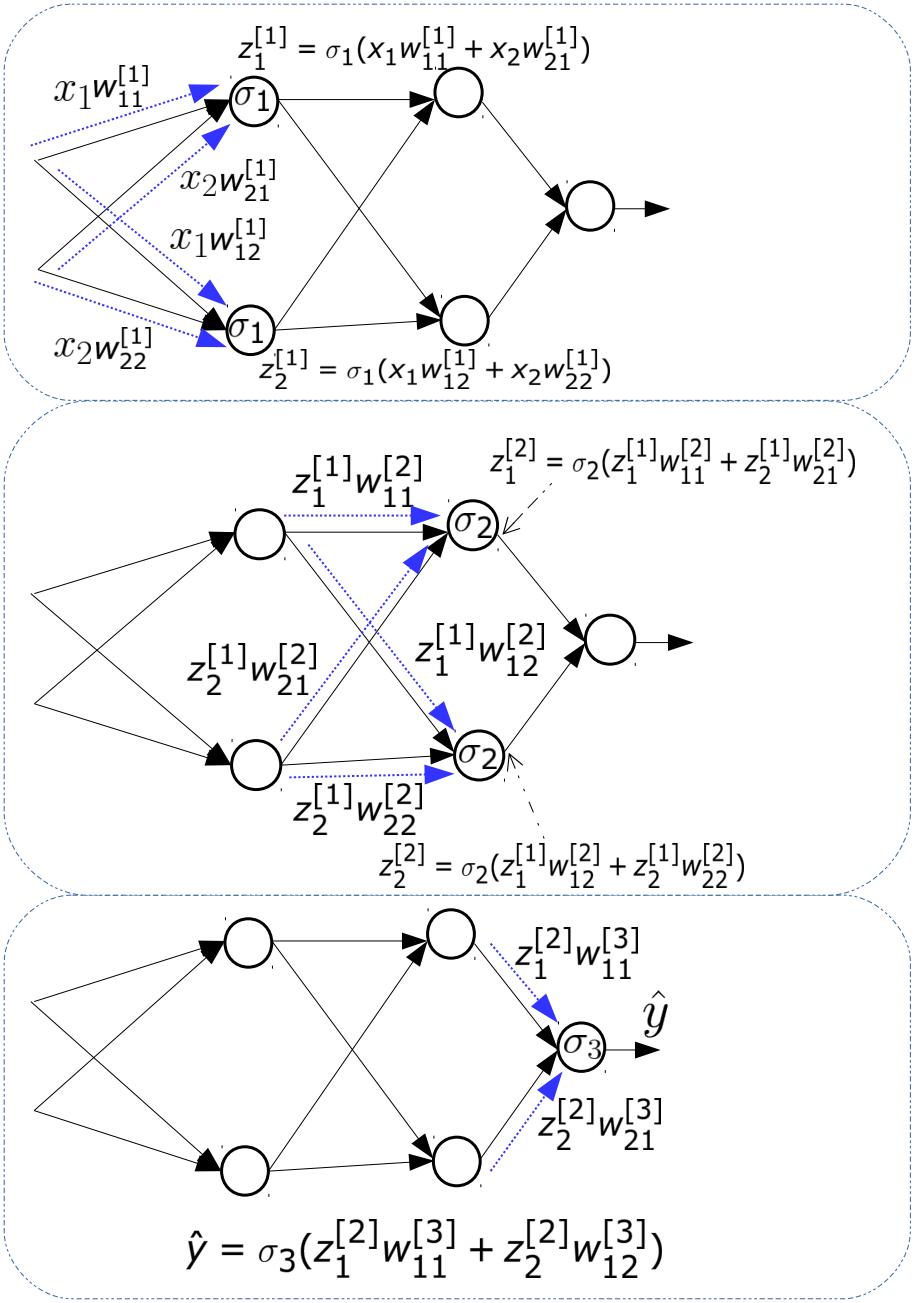


Figure 1.4: Forward propagation. First, the input is multiplied by $w_x^{[1]}$ weights as shown (top). The activation function σ_1 is applied to the two sums. The resultants, $z_1^{[1]}$ and $z_2^{[1]}$ are multiplied and the new results are propagated to the next layer, where σ_2 is applied to the sums (middle). Finally, the sum of the resultants is calculated and applied with an activation function σ_3, resulting in the output $\hat{y}$ (bottom).

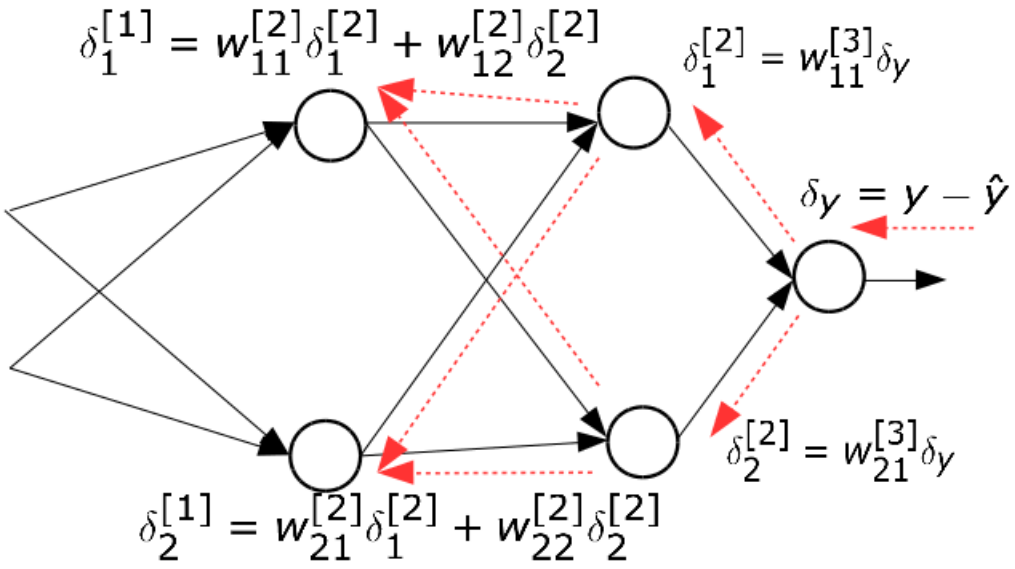


Figure 1.5: The error is updated. Note that δ_y can be a mean square; for simplicity it is shown as $y - \hat{y}$. The error values are updated and propagated up to $\delta_i^{[1]}$.

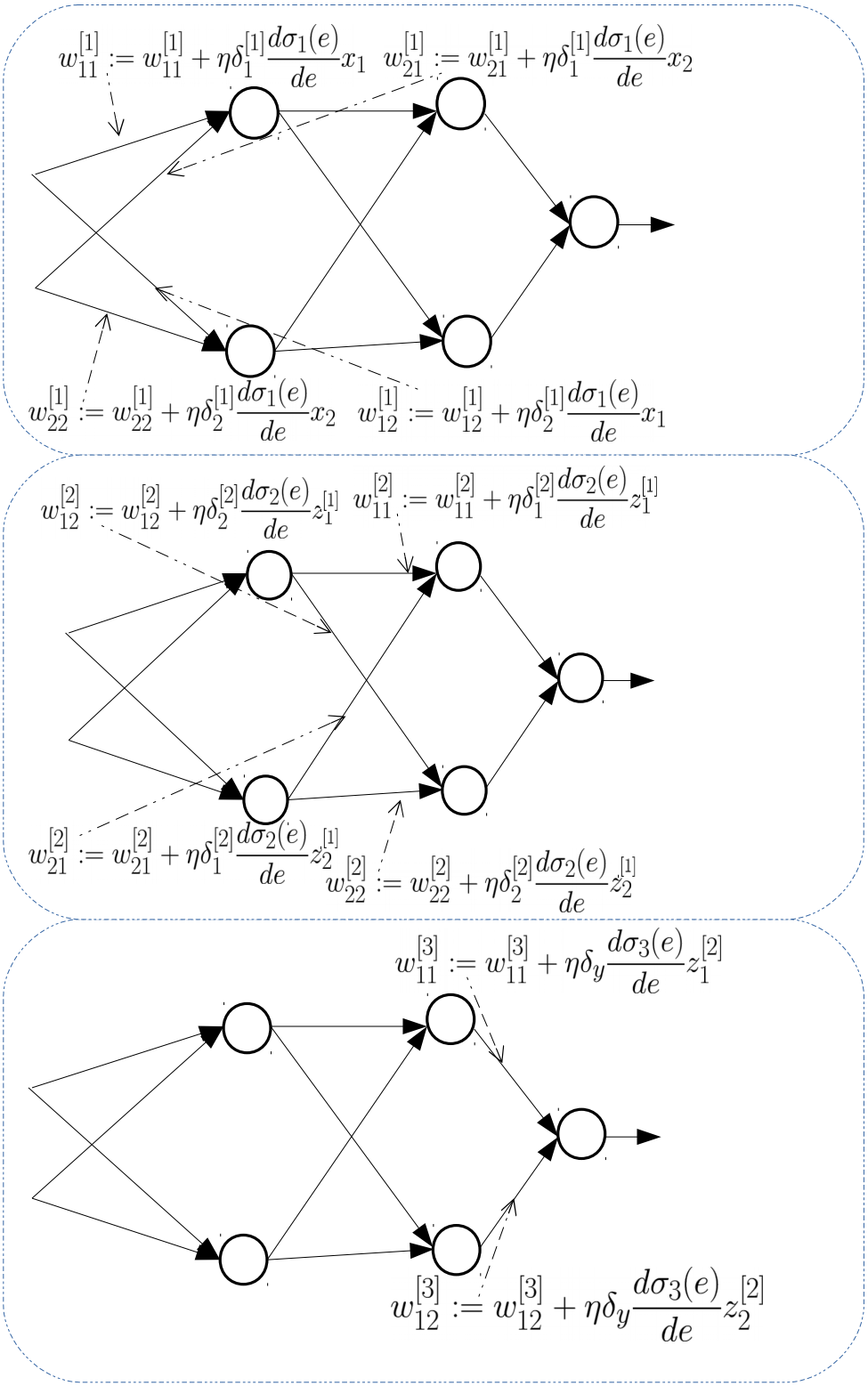


Figure 1.6: Using the calculated errors (δs) the weights are updated, starting from the weights $w_i^{[1]}$ and ending with the weights $w_i^{[3]}$.

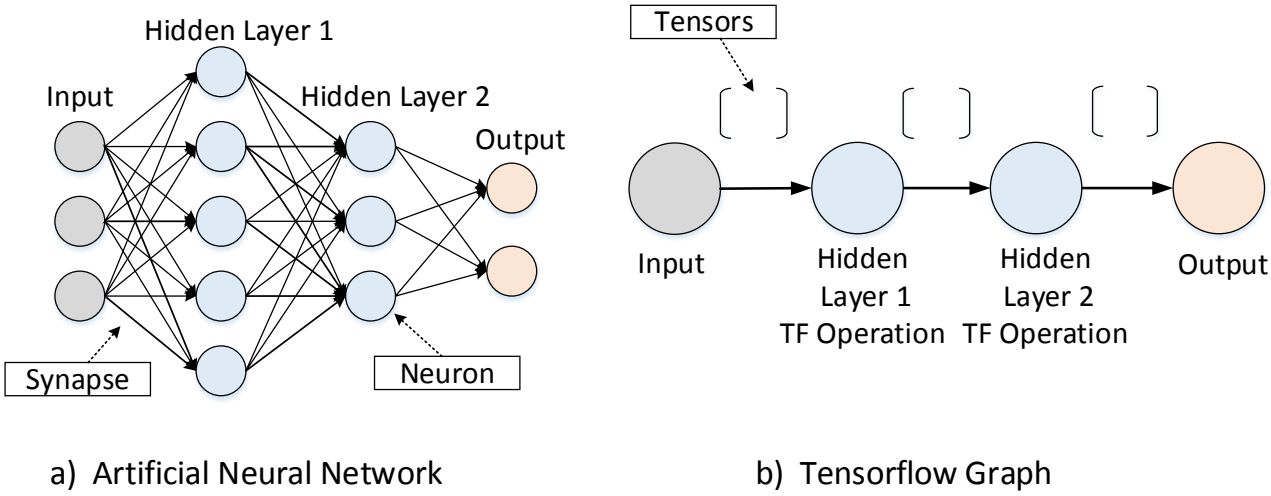


Figure 1.7: ANN and its equivalent TF graph.

Chapter 2

Related Work

With regard to DNN performance, and more specifically training performance, several approaches have been considered in industry and academia. The proposed approaches may be split into categories: those that affect the accuracy of the DNN, and those that do not. The approaches that affect the accuracy change the structure of the DNN, while those with no effect on accuracy keep the structure intact and achieve performance improvements in other ways, mainly based on resource management.

2.1 Structural Changes to DNNs

Historically, DNNs have been known to be large and therefore *over-parameterized* [20], indicating that there is room for improvement via size reduction. Size reduction of DNNs has several benefits: it saves space in terms of storage; reduces computation, thus resulting in reduced latency and power consumption; and requires less transfer of parameters to and from memory. Structural changes with the objective of reducing size can be based on two types of approaches: reduction of parameters, and lowering of precision weights.

2.1.1 Pruning

Some of the earliest work in reducing the number of weights, also known as *pruning*, are Optimal Brain Damage [45], where a local error function is used to determine which weights to remove, and Optimal Brain Surgeon [29], which improves on and further reduces the number of weights achieved by [45]. The development of convolutional neural networks (CNNs) [42], which use the convolutional layer, is a major contributor to the speeding up of DNNs in both training and inference, as well as accuracy. This is due to convolutional layers having significantly lower numbers of weights than fully connected layers. Other pruning methods target more specific types, or parts, of DNNs, while some use different decision criteria: [30] prunes *channels* or *feature maps* in a convolutional layer via statistical selection methods based on regression. The purpose of these approaches is to accelerate the training process.

Chapter 8 touches on pruning in relation to partitioning to achieve parallelism for the sake of accelerated training in DNNs. This indicates that pruning is one of the set of tools available for working toward acceleration.

2.1.2 Lower Precision

Reducing the precision to a lower bit size has the effects of reducing computation and reducing space. A particularly aggressive approach to compression is taken in BinaryConnect [19], where the values of the weights are a single bit and are trained as such. A less aggressive approach is taken in [75], where the weights are restricted to -1, 0, and +1, while [71] uses two-bit precision. In [51], three-bit precision is used via base-2 logarithmic representation. In [28], the authors reduce the number of parameters via *quantization*, another form of lowering the precision by having weights share values.

2.2 Resource Management

One key consideration when carrying out structural changes is that they can affect accuracy. However, when resource management is applied, it does not affect either the functionality or the accuracy of the DNN; therefore, in many instances, it is less invasive.

2.2.1 System Level

One approach within this category of accelerating the training of DNNs is to partition a DNN along many GPUs. This is known as *model parallelism*, as mentioned above. Extensive research has been conducted in this domain: using commodity off-the-shelf high-performance computing, research by [14] uses a cluster of GPUs to train one billion parameters. Similarly, Adam [16] presents and implements a distributed system to train two billion parameters, where the system is designed to accelerate the training. In Tetris [24], the researchers propose a hardware architecture that implements scheduling as well as partitioning of tensors in a 3D memory. In [34], each layer of the DNN uses a different parallelization strategy, while in FlexFlow [35], the parallelization search is expanded to different dimensions using samples, operations, and attributes, as well as parameters using simulation. In all of the works mentioned so far in this section, hardware and large distributed systems have been designed and used for the sole purpose of accelerating the training process.

2.2.2 Semi-automatic Partitioning

Automatic partitioning of computational graphs is not a recent phenomenon, but is relatively new in relation to DNNs. Early works from Bell labs [39] partition by investigating the *edges* or links between the operations. A similar approach but with different cost evaluations is carried out in [22]. In [36], meta-heuristic approaches are used to partition computational

graphs, while [27] tackles application-specific cases using mathematical and optimization methods.

In terms of DNNs, Google Brain [49] tackles the graph partitioning problem using reinforcement learning (RL) as an optimization method; however, manual work is involved since many of the operations in this are grouped and therefore share a device. In Tofu [73], automatic partitioning occurs while requiring input from the domain expert. The domain expert in this case is required to communicate with the system using a descriptive language.

In Chapter 7 we present a static heuristic approach to creating a mapping that partitions the DNN.

2.3 Automatic Partitioning

There are many options for accelerating the training of DNNs. These are not mutually exclusive and may work in conjunction with one another. The route of model parallelism through partitioning is the least invasive since it does not affect the accuracy of the DNN.

Work using partitioning, and more specifically automatic partitioning, in DNNs is relatively recent. Many trials still require some input or intervention from a domain expert to ensure the most effective partitioning of the DNN. This dissertation presents our work, which achieves speeding up of the search due to modeling the makespan (single training iteration) prediction, as well as adaptability due to device and system changes.

Chapter 3

Computational Model

3.1 TensorFlow and Computational Graphs

TF uses the computational graph approach: each *node* in the TensorFlow graph G is an operation op_i, and the *vertices* are *tensors*, i.e. multi-dimensional matrices. Figure 3.1 illustrates a *directed* graph in TF. The *dataflow* graph in Figure 3.1 shows the dependencies: certain operations will not execute unless all data dependencies are executed. Let $G = (Op, E)$, where $Op = \{op_1, op_2, ..op_{N_{op}}\}$ are the operations, and $E = \{(op_a, op_b), (op_c, op_d)...\}$ are the *directed edges* where $e_i = (op_a, op_b)$ is a tensor from $op_a \longrightarrow op_b$ and $a \neq b$. Note that the TF graph G is assumed to be an *acyclic* dataflow graph.

The order in which operations are executed is determined by the TF scheduler. The *Distributed Master* evaluates the TF Graph G's nodes; i.e., the *Worker Services* schedules the operations according to the Distributed Master's request.

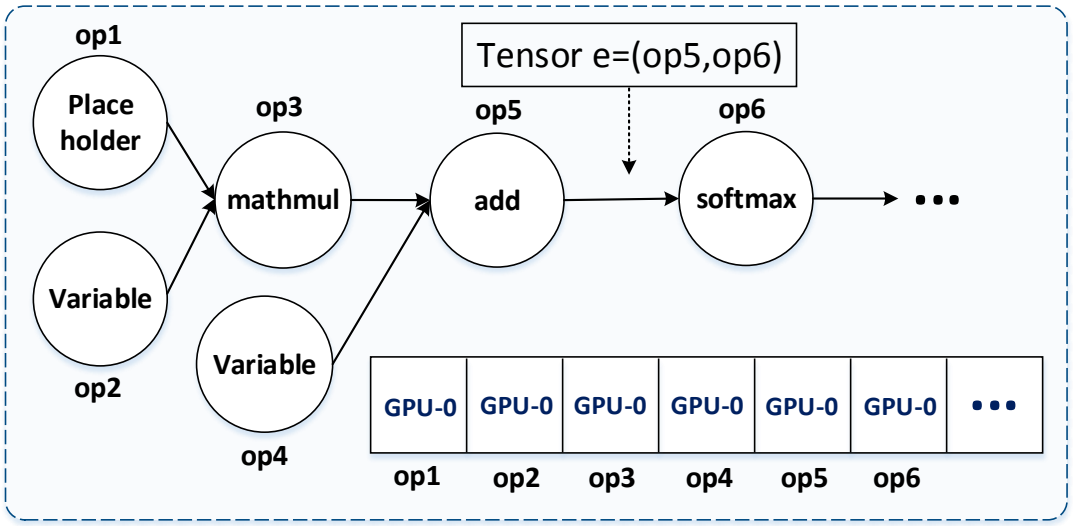


Figure 3.1: Homogeneous mapping: All the operations, by default, are mapped to GPU-0. See Code 3.1.

Code 3.1: NN model in TF (default homogeneous mapping). See Figure 3.1.

```
import tensorflow as tf

op1=tf.placeholder(tf.float32,[None, 784])

op2=tf.Variable(tf.zeros([784,10]))

op3=tf.matmul(op1, op2)

op4=tf.Variable(tf.zeros([10]))

op5=tf.add(op3,op4)

op6=tf.nn.softmax(op5)

...
```

In TF, the programming paradigm requires a construction of a model (graph) where the hyperparameters are set before the model is run. A model run could either be for training or inference. Code 3.1 shows a model construction of Figure 3.1, while code 3.2 shows the script required to run the model. A *Session* is created and the last operation in G, in this case op_6, is passed on as a parameter to the Session.

A single run would provide the actual *makespan* of the graph. The makespan is the time it takes to complete one *iteration* (i.e. run) of the graph.

Code 3.2: Run model in Session

```
...
sess = tf.Session()
sess.run(op6,feeddict=...)
...
```

3.2 Mapping

Mapping, also referred to as device placement, is the assignment of an operation to a device. By default, TF maps all the operations to a single device. If GPU-enabled TF is installed and the hardware is supported, then all the operations in a TF graph are mapped to a single GPU; otherwise, all the operations are mapped to a CPU. A workaround to defining your own mapping is to use the *tf.device* directive. An illustration of a mapped TF graph and its accompanying code is shown in Figure 3.2.

Given a set of devices $D = \{d_1, d_2, ..d_{N_D}\}$, and a set of operations $Op = \{op_1, op_2, ...op_{N_{op}}\}$, a particular mapping is defined as $m_i = \{(op_1, d_x), (op_2, d_y)...(op_{N_op}, d_z)\}$ (where d_x,d_y,d_z denote any device since the device-to-operation mapping is one-to-many). Note that the size of m_i,$|m_i| = |Op| = N_{op}$. The mapping of devices to operations is one-to-many, meaning that several operations could be mapped to a single device in any particular mapping, while the opposite is not true (see Figure 3.2).

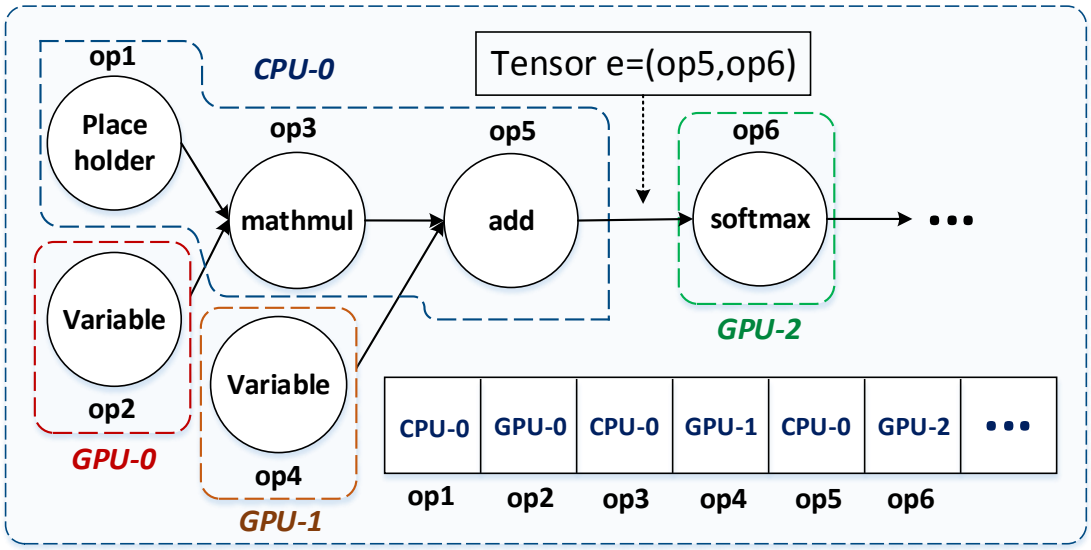


Figure 3.2: Heterogeneous mapping: $op1$,$op3$ and $op5$ are mapped, i.e. assigned, to $CPU-0$, $op2$ to $GPU-0$, $op4$ to $GPU-1$, and $op6$ to $GPU-2$. See Code 3.3.

Code 3.3: Addition of tf.device

```
import tensorflow as tf

with tf.device('/cpu:0'):

        op1=tf.placeholder(tf.float32,[None, 784])

with tf.device('/gpu:0'):

        op2=tf.Variable(tf.zeros([784,10]))

with tf.device('/cpu:0'):

        op3=tf.matmul(op1, op2)

with tf.device('/gpu:1'):

        op4=tf.Variable(tf.zeros([10]))

with tf.device('/cpu:0'):

        op5=tf.add(op3,op4)

with tf.device('/gpu:2'):

        op6=tf.nn.softmax(op5)

...
```

In our notation, $m_{TF} = m_{gpu-0} = m_2$ is the GPU-0 mapping, which is the default TF mapping (as shown in Figure 3.1), while $m_{cpu} = m_1$ is the *homogeneous* CPU mapping and $m_{gpu-1} = m_3$ is the homogeneous GPU-1 mapping. The rest of the mappings, $m_i | i > 3$, are different *heterogeneous* mappings (an example of a heterogeneous mapping is shown in Figure 3.2). No two mappings are the same; i.e., $m_i \neq m_j | i \neq j$ where $m_i, m_j \in M$.

3.3 Optimization

The objective is to find a mapping m that results in a faster execution time than the default TF mapping, thus speeding up the overall training time. The optimization problem is therefore:

$$m^\star = \arg\min_{m \in M} f_t(m) \tag{3.1}$$

where $f_t(m)$ is the *makespan* (execution time) of a single training iteration of the TF graph using mapping m. m^* is any mapping that outperforms the TF mapping. Note that possible mappings of G are represented by M, which has a size of $|M| = N_D^{N_{op}}$, where N_D and N_{op} are the number of devices and operations, respectively. The search for an optimal mapping in the search space is thus considered an *NP-hard* [56, 40, 70] problem. The mapping problem could be reduced to *NP-Complete* by relaxing the condition; i.e., by finding a mapping that outperforms the default homogeneous TF mapping rather than finding the global optimal mapping ($M^* \subset M$ and $m^* \in M^* | f_t(m^*) < f_t(m_{TF})$). Some of the characteristics of the makespan $f_t(m)$ are as follows:

- It is a *continuous* function; i.e., the execution time is a real number.
- The input data m represent a *tuple* of *categorical* data; i.e., the values of the operations are device labels which are discrete.

- A single evaluation is *expensive*, meaning that an actual run of the graph has to occur to find the makespan value.
- It is a *black-box* function; i.e., its structure is unknown (not convex, not linear, etc.).
- Given that it is a black-box function, it is thus *non-differentiable*. Neither the first- nor the second-order derivative may be utilized.

For this process to be worthwhile, the whole training time needs to be shorter than the training time of the default homogeneous TF mapping:

$$F_t(\pi, oh_\pi) < F_t(\pi_{m_{TF}}, 0) \tag{3.2}$$

where $F_t(\pi, oh_\pi)$ is the sum of execution times plus overhead given a *policy* of mappings $\pi = m_a, m_b, ...$ Note that generating such policy is an overhead, represented by oh_π. $F_\tau(\pi_{m_{TF}}, 0)$ is the TF total training time with a policy of using a single type of mapping, which is a homogeneous mapping, m_{TF}, and no search overhead.

$$F_t(\pi_{m_{TF}}, 0) = \sum_{i=1}^{I} f_t(m_{TF}) \tag{3.3}$$

where I is the number of training iterations it takes to reach the final desired model f_{NN}. Regardless of policy used, and as long as the number of iterations is I, the desired f_{NN} is unchanged. Figure 3.3 illustrates this concept of training where the path is set regardless of the value of t.

3.4 Experimental Setup

For evaluation purposes, a multi-core CPU (Intel(R) Core(TM) i7-7700 CPU 3.60Ghz) and 2 GPUs (Nvidia GeForce GTX 1050 Ti) are used. For implementation, we use Python 2.7.15

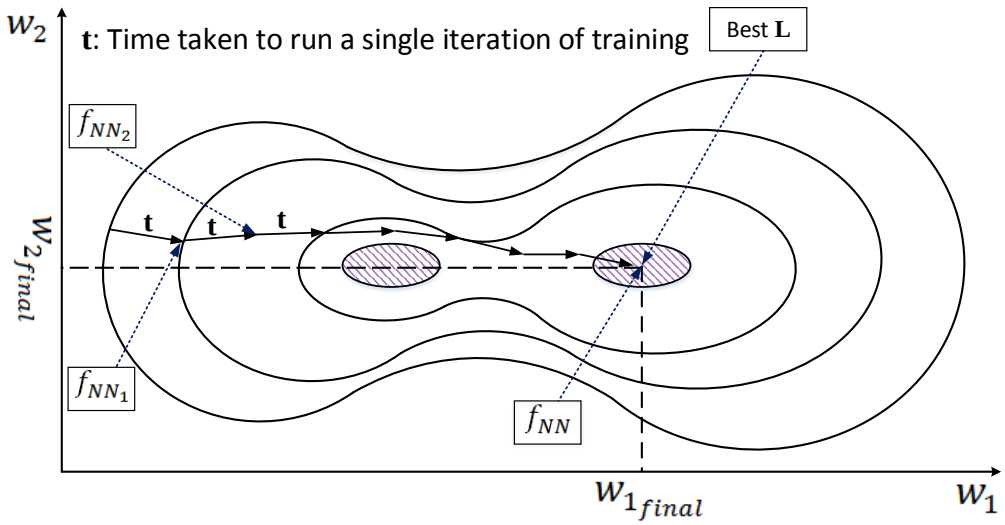


Figure 3.3: The gradient path of the model. t does not change in each iteration (as long as device mapping does not change). The values of w_1 and w_2 change in each iteration as part of the training process. f_{NN} is the final trained DNN that has the best possible values for the parameters given the training method. The mapping does not affect f_{NN}, just how fast the process of reaching this value is.

with the Anaconda bundled package of libraries. The benchmarks are implemented in GPU-supported TensorFlow 1.9.0, running on CUDA 9.1 and CuDNN v7.1. The gradient boosting regressor (GBR) makespan predictive model (described in Chapter 6) is implemented using scikit-learn 0.19.1 [54]. The Bayesian optimizer (described in Chapter 5) is implemented using Hyperopt 0.2 [9].

To evaluate the proposed method, three state-of-the-art benchmarks are run using HTF-MPR, Adaptive HTF-MPR, and the default TF mapper. Table 3.1 shows the benchmark list, the number of eligible operations for mapping, and the number of training iterations per benchmark.

Benchmark	Mappable Operations	Total Operations	Training Iterations
MNIST Softmax	10	99	60K
ALEXNET	54	294	500K
VGG-16	69	376	500K

Table 3.1: Benchmarks.

Name(s)	Operation Type	count
y_2	tf.add	1
y_1	tf.matmul	1
y	tf.nn.softmax	1
cross_entropy	tf.nn.softmax_cross_entropy_with _logits	1
x, y_	tf.placeholder	2
reduce_mean	tf.reduce_mean	1
train_step	tf.train.GradientDescentOptimizer	1
W, b	tf.Variable	2

Table 3.2: MNIST Softmax distribution of operations and labels(names) of said operations.

Unigine's SuperPostion benchmarking tool [67] is used to stress-test the system in order to test out the adaptive feature of Adaptive HTF-MPR.

3.4.1 MNIST Softmax

The *MNIST Softmax* used in our evaluation is a simple TF implementation [8] that trains a classifier for a ten-digit grayscale image dataset MNIST [47]. The dataset contains 60,000 training and 10,000 testing images. Each image is 28x28 grayscale and, as the dataset suggests, the classifier has ten classes. Figure 3.4 shows the graph representation.

Given that there are only ten *mappable operations* (operations in the computational graph that are explicitly mentioned in the Python TF code), the total number of possible mappings in this case is $N_D^{N_{OP}} = 3^{10}$. With this small number of mappings, it is possible to generate and evaluate the whole search space; therefore, a brute-force analysis can be conducted to find the *global optimal* mapping. In future chapters, we will compare the m^* of HTF-MPR, Adaptive-HTFMPR, m_{TF}, and the global optimal. In addition, we will compare F_t (see Equation 3.2).

The distribution of operations is shown in Table 3.2:

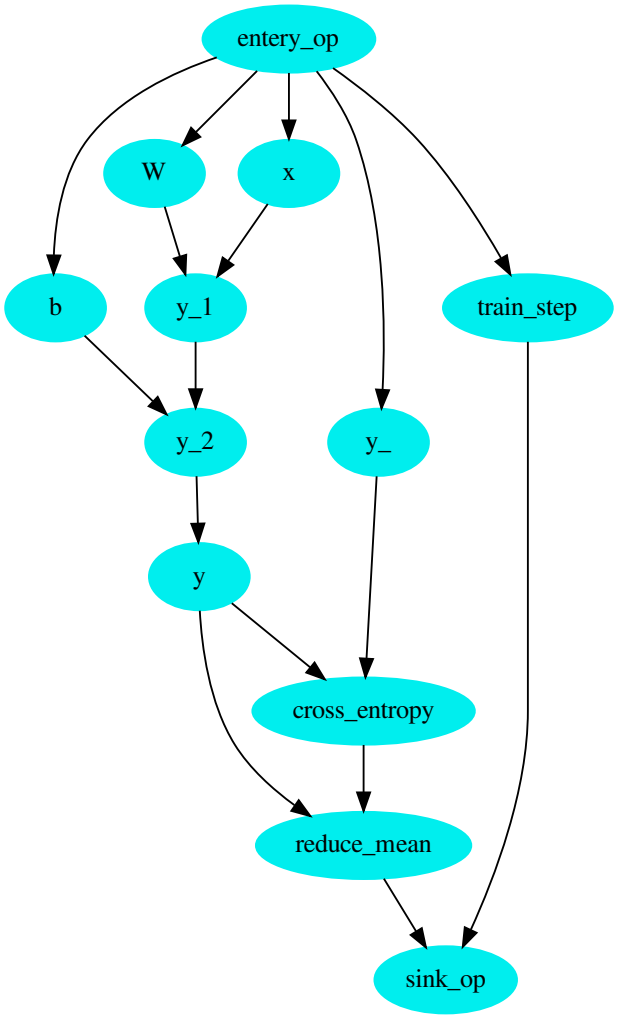


Figure 3.4: MNIST Softmax computational graph. There are ten mappable operations. The entry_op and sink_op are virtual operations and are not mapped to any device.

3.4.2 AlexNet and VGG-16

AlexNet [43] and VGG-16 [64] are deep CNNs that are designed to classify images from the ImageNet [58] dataset. The ImageNet training dataset contains 1.2 million labeled images of 1000 labels, i.e. classifications. The input to the neural network is a three-channel rescale image with a resolution of 224x224x3. HTF-MPR and Adaptive HTF-MPR are tested on both neural networks to gauge and evaluate the speeding up, by comparing the respective $f_t(m^*_{htf.mpr})$ and $f_t(m^*_{A.htf.mpr})$. In addition, the total training time of $F_t(\pi_{htf.mpr}, oh)$ and $F_t(\pi_{A.htf.mpr}, oh)$ are measured.

For VGG-16, we run 500,000 training iterations with a batch size of 32 per iteration. For AlexNet, we run 500,000 training iterations with a batch size of 64 per iteration. The computational graphs for VGG-16 and AlexNet are shown in Figures 3.5 and 3.6, respectively.

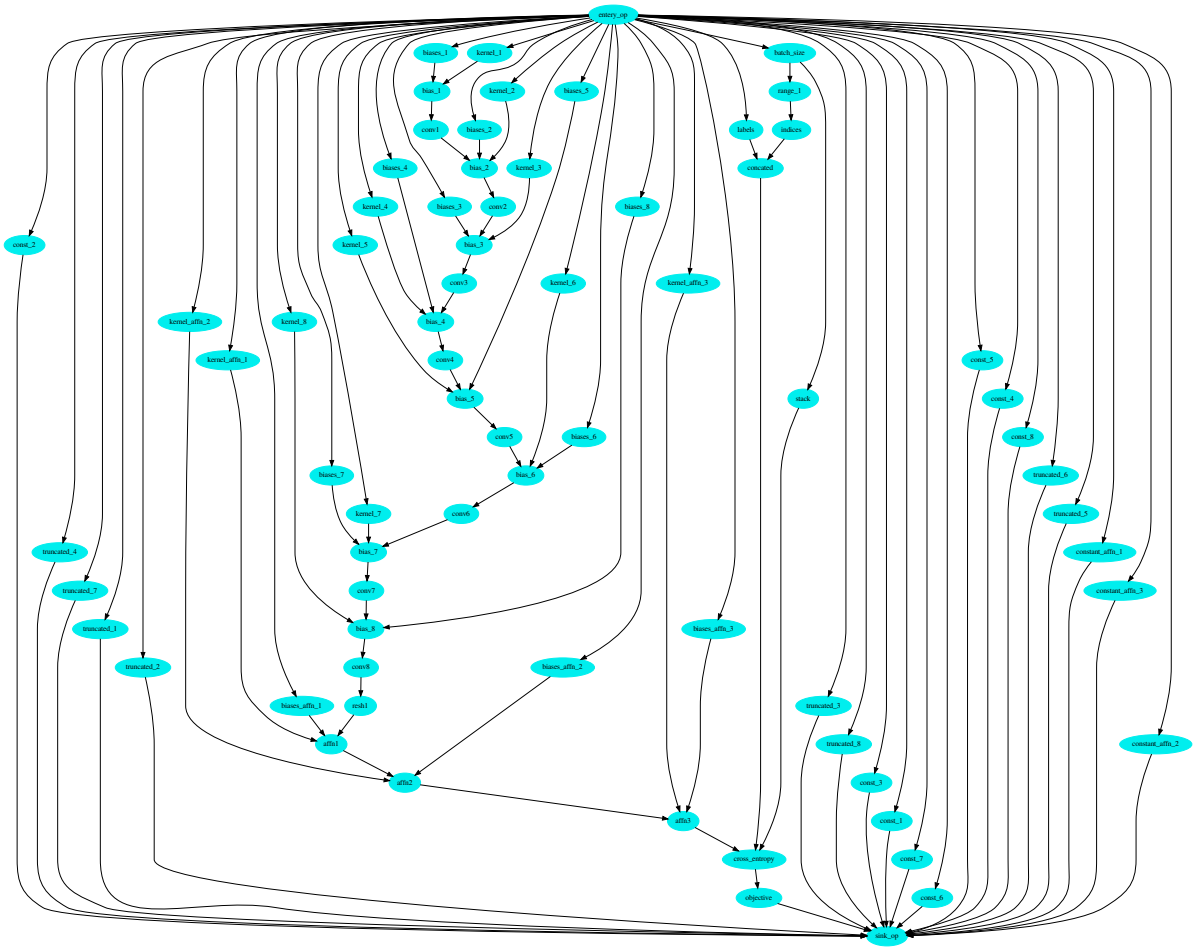


Figure 3.5: VGG-16 computational graph. There are 69 mappable operations. The source and sink are virtual operations and are not mapped to any device.

Table 3.3 shows the operation count for VGG-16, while Table 3.4 shows the operation count for AlexNet.

Operation Type	count
tf.concat	1
tf.constant	11
tf.expand_dims	2
tf.nn.relu	8
tf.nn.relu_layer	3
tf.nn.softmax_cross_entropy_with_logits	1
tf.range	1
tf.reduce_mean	1
tf.reshape	9
tf.size	1
tf.stack	1
tf.truncated_normal	8
tf.Variable	22

Table 3.3: VGG-16 distribution of operations.

Operation Type	count
tf.concat	1
tf.constant	8
tf.expand_dims	2
tf.nn.relu	5
tf.nn.relu_layer	3
tf.nn.softmax_cross_entropy_with_logits	1
tf.range	1
tf.reduce_mean	1
tf.reshape	6
tf.size	1
tf.stack	1
tf.truncated_normal	8
tf.Variable	16

Table 3.4: AlexNet distribution of operations.

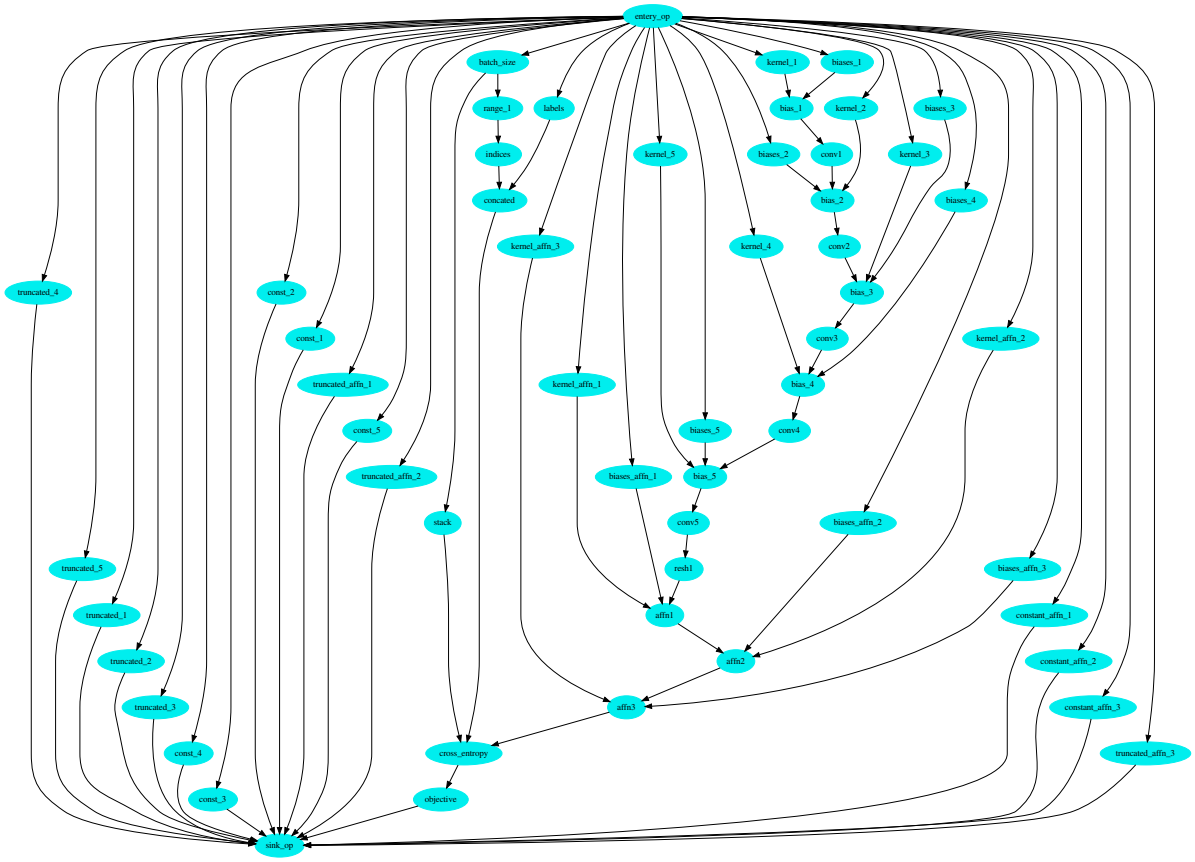


Figure 3.6: AlexNet computational graph. There are 54 mappable operations. The source and sink are virtual operations and are not mapped to any device.

3.5 MNIST Softmax Brute-Force Analysis

In this section, we evaluate the whole space design of mappings for MNIST Softmax. This is in order to provide a general idea of the distribution of makespans, and of which mappings are optimal.

We have generated all $N_D^{N_{op}} = 3^{10} = 59049$ mappings for MNIST Softmax. Figure 3.7 shows part of the distribution of the makespan, as well the average makespan and the makespan of the m_{GPU-0} homogeneous mapping. The makespan values extend to approximately 0.02 seconds. Makespan distribution beyond 0.002 is not shown in the figure. The three highest-

performing mappings and the three least effective mappings are shown in Figure 3.8, along with the makespan values of the respective mappings.

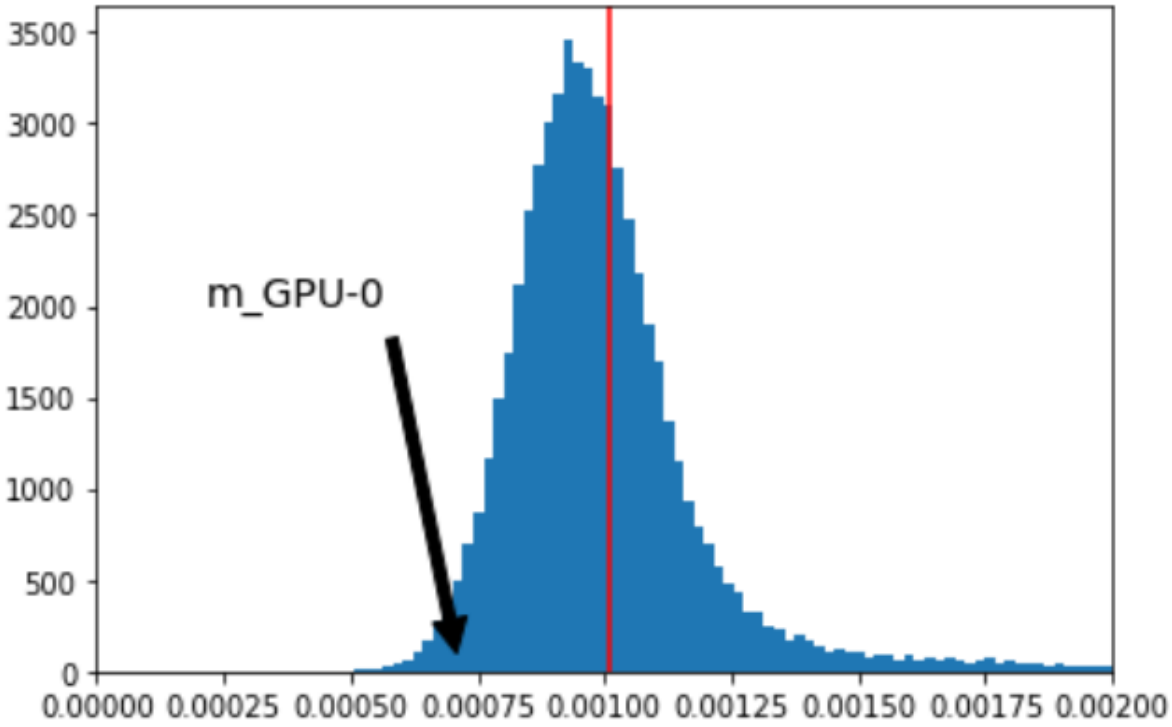


Figure 3.7: MNIST Softmax makespan distribution: x-axis shows the makespan and y-axis shows the count for that makespan. The mean of the distribution is shown by the red vertical line. Note that the figure caps at 0.002s, but the distribution has a long tail that extends to 0.02s. Approximately 5% of mappings outperform the default TF m_{GPU-0} mapping in the MNIST Softmax case given the current state of the TF software.

3.5.1 Analysis

Due to the small size of the model, CPU resources are used for the majority of the operations, rather than GPU resources. The brute-force analysis in this case does not show that CPU allocation is better, but rather indicates that a mix of resources is most effective and that the mix is non-intuitive, meaning that an expert would not have devised the mapping that is indicated in Figure 3.8. Thus, this analysis presents a justification for using a meta-heuristic approach to discovering and searching mappings.

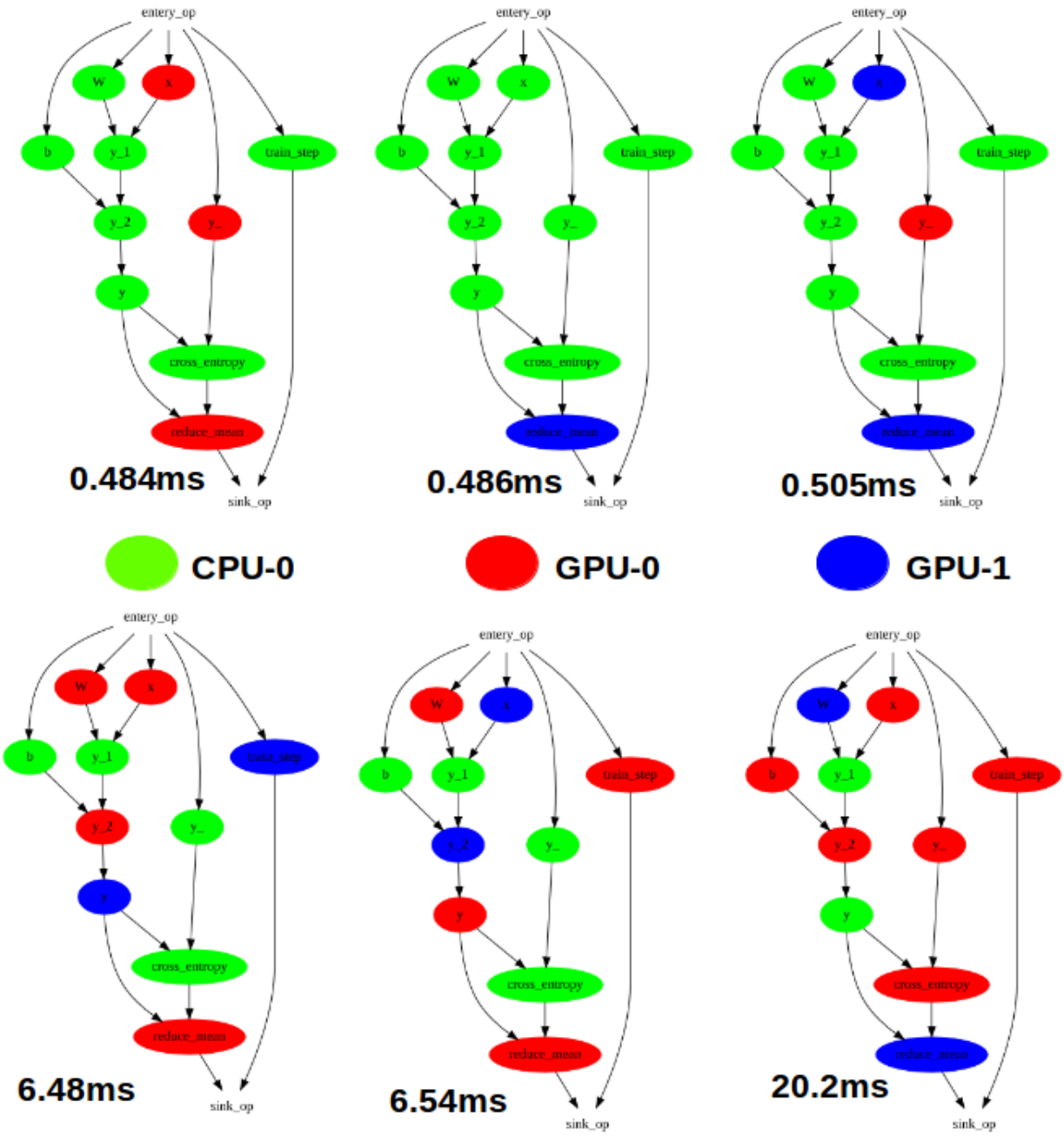


Figure 3.8: The three mappings at the top of the figure are the top three mappings in terms of makespan. The topmost has seven operations mapped to CPU-0, and three operations mapped to GPU-0, with a makespan of 0.484 ms per iteration. The three at the bottom of the figure are the least effective mappings: the worst has a makespan of 20.2 ms, with two operations mapped to CPU-0, six operations mapped to GPU-0, and two operations mapped to GPU-1. The mapping m_{GPU-0} has a makespan $f_t(m_{GPU-0})$ =0.72 ms. Note that the worst mappings change devices after each operation, incurring high communication costs as overheads.

Chapter 4

Generating Initial Mappings

The importance of the initial mappings is two-fold; one is for building the predictive model and is therefore used as the training dataset (described in a chapter 5), the other is for usage in the genetic algorithm (GA) applied in both HTF-MPR and Adaptive HTF-MPR (described in Chapter 6).

4.1 Algorithmic Mapping

Algorithmic mapping, as the name suggests, is a way of generating mappings that is done in a methodological way. An illustrative example of the mapping types generated by the algorithmic approach is shown in Fig 4.1. The initial mapping types provide variety, are useful in training the fitness predictive model, and are promising initial starting points for the GA. One of the mappings is the default mapping of heterogeneous (GPU support) TF (all GPU-0). The other mapping is the default mapping for non-GPU supported TF (all CPU-0).

Briefly, the initial mappings used in the algorithmic approach are generated with the characteristics listed below.

- **Homogeneous mapping**: A single device for all operations. Figure 4.1 shows an example of homogeneous mapping. The number of mappings is proportional to the number of devices, i.e. N_D. (Figures 4.1**a** and **b**).
- **Longest path mapping**: A single device is mapped to the operations making up the longest path in the graph, while the other operations are mapped to different devices than the one on the longest path.(Figures 4.1**c** and **d**)
- **Random homogeneous path mapping**: A single device is mapped to a non-longest single path, while the rest of the operations are mapped to different devices than the one on the designated path. (Figure 4.1**e**).
- **Color mapping**: Whenever possible, no two connected operations should be mapped to the same device. Note that, in terms of makespan, this would result in the worst possible performance; however, this approach is used for variety in the training dataset used for building the ML predictive model. (Figure 4.1**f**).

A high level of variety in the initial mappings leads to a more versatile generalized predictive model, but a less accurate model on any given concentrated region where the search takes place. An alternative method, that of Bayesian Optimization (BO), will be presented in the subsequent section.

In cases of homogeneous mapping, all operations are assigned to one device, eliminating any type of communication cost. This is beneficial when communication costs are relatively high compared to computational costs, and work best when all operations, and therefore parameters, fit into the device's memory. Later chapters will indicate that even small models such as MNIST Softmax will have better mappings that are not homogeneous. Intuitively, the

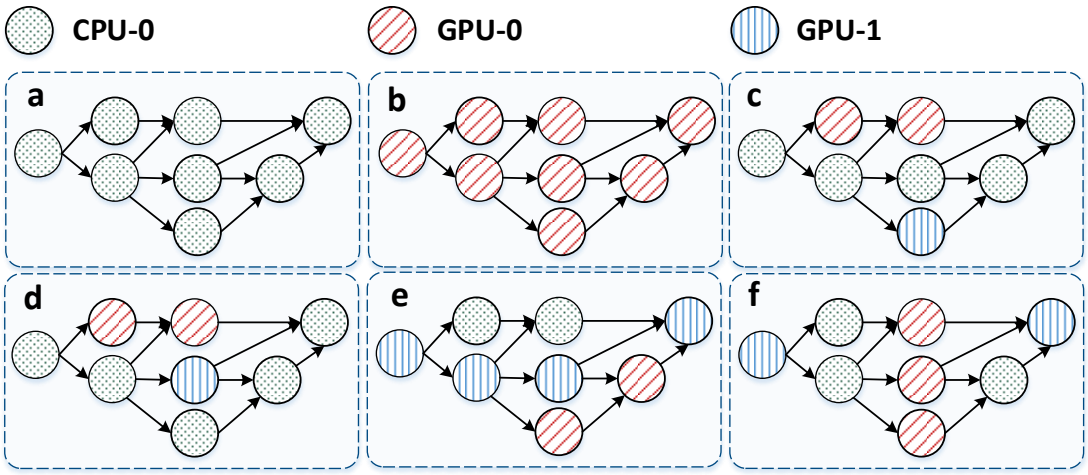


Figure 4.1: Examples of some initial mappings: **a** and **b** are homogeneous (single device), **c** and **d** are longest paths, **e** is non-longest path, and **f** is color mapped.

justification of such mapping is considered to be reasonable due to the lack of communication cost. The default mapping in GPU-enabled TF is a homogeneous GPU-0 mapping, and non-GPU-enabled TF uses a homogeneous CPU mapping.

The intuition behind the longest path approach is that all the dependent operations in the longest path should use one device while other operations use other devices to utilize parallelization. This would reduce the cost of inter-device communication in the longest path and, since parallelization is not achievable due to dependencies, there is no opportunity cost to consider.

Given that the communication costs are unknown and that the computational cost per operation is also hidden information, it could be that a non-longest path (where length is measured by number of operations in the path of the graph) would best be made homogeneous due to the path having the highest latency.

With color mapping, the inter-communication tax is insured to be taken. Color-mapped mappings are expected to have the worst makespans. The justification of their use in this case is to add variety for the dataset that is used in the ML process to create the predictive model.

4.2 Bayesian Optimization

Bayesian optimization (BO) [55] is based on *Bayesian reasoning*, where the reconstruction of the objective function $f_t(m)$ is updated based on new evidence; i.e., due to evaluation of new data points in $f_t(m)$. The higher the number of data points evaluated, the closer the *surrogate function* becomes to $f_t(m)$. The *Tree Parzen Estimator* (TPE) [10, 9] is one of the methods for constructing the surrogate function. The target of the Bayesian optimizer is to find the data point (input) that would result in the minimum of the function. This is achieved by choosing the next input to be evaluated according to the surrogate function and past results. The surrogate function is described by a probabilistic model approach:

$$P(t|m) \sim \mathcal{N}(\mu(m), \sigma(m)^2) \tag{4.1}$$

where $\mathcal{N}(\mu, \sigma^2)$ is the *normal distribution*, with μ as the expected *mean* function and σ^2 as the expected *variance* function.

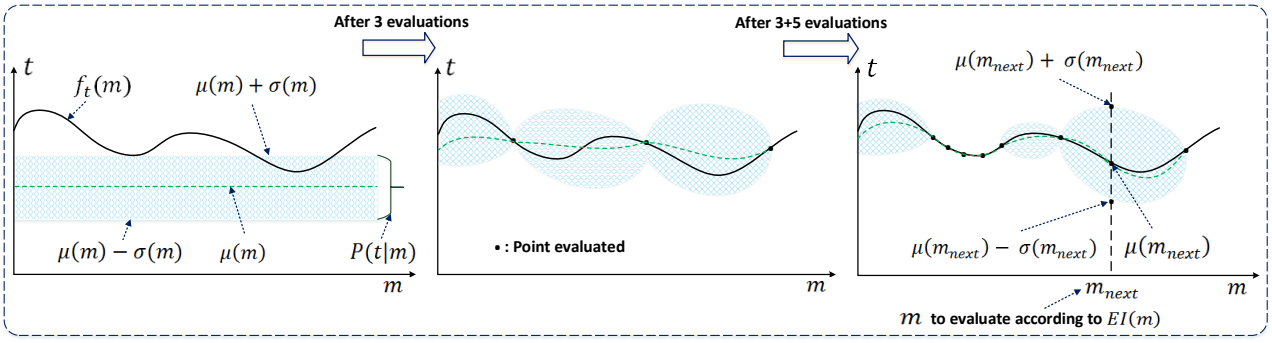


Figure 4.2: Bayesian optimization general method.

The surrogate function is optimized via Bayesian methods by selecting an m that will perform well on $P(t|m)$. Figure 4.2 shows a general overview of how an increased number of evaluations affects $P(t|m)$. As an overview, the steps taken by the Bayesian optimizer are:

1. Build $P(t|m)$ according to the already evaluated $f_t(m)$. In our case, the homogeneous mappings m_1,m_2,..m_{N_D} and their results on $f_t(m)$ result in an initial $P(t|m)$.

2. Then, the Bayesian optimizer chooses the next m that would be *assumed* to perform well on $P(t|m)$.

3. The chosen m is evaluated with $f_t(m)$.

4. $P(t|m)$ is updated based on the results of m on $f_t(m)$.

Steps 2–4 are repeated several times. The reason for the use of $f_t(m)$ rather than $f'_t(m)$ (the predictive model and less costly function) is that BO is an expensive approach; specifically, the construction of $P(t|m)$ from history, and selection of the next m to evaluate, are expensive. Therefore, the BO would perform well on expensive functions such as $f_t(m)$, given that the whole Bayesian process is expensive. Using $f'_t(m)$ in the BO would not be beneficial with regard to time. BOs are expensive in terms of computation time, yet they require fewer calls to the objective function compared to other optimizers, since they apply *reason* in deciding what to evaluate next; i.e., they use $P(t|m)$, to choose the next m to evaluate.

To decide on which m to evaluate next (step 3), a utility function known as the *acquisition function* [62] is used:

$$EI(m) = E[\max_{m}(0, f_t(m) - f_t(m_{best}))] \tag{4.2}$$

$$m_{next} = \arg\max_{m} EI(m) \tag{4.3}$$

EI is the *expected improvement*, a type of acquisition function. m_{best} is the *current* best solution, while m_{next} is the next m that would be evaluated. $EI(m)$ is analytically evaluated as follows:

$$EI(m) = \begin{cases} (\mu(m) - f_t(m_{best}))\Phi(Z) & \\ +\sigma(m)\phi(Z) & \sigma(m) > 0 \\ 0 & \sigma(m) \leq 0 \end{cases} \tag{4.4}$$

$$\text{where } Z = \frac{\mu(m) - f_t(m_{best})}{\sigma(m)} \tag{4.5}$$

where $\mu(m)$ and $\sigma(m)$ are the *mean* and the *standard deviation* of the distribution of $P(t|m)$ at point m, respectively (as was mentioned in Equation 4.1), while Φ and ϕ are the *cumulative distribution function* and *probability density function* of the *normal distribution*, respectively. Note that the acquisition function is less costly in terms of computation compared to $f(m)$; i.e., $\mu(m)$ and $\sigma(m)$ are very inexpensive to evaluate. EI displays a high value if the evaluated m is in a known neighborhood that outperforms m_{best} (high $\mu(m)$), or if we evaluate in an unknown territory (high $\sigma(m)$). Both approaches; *exploitation* (high $\mu(m)$) and *exploration* (high $\sigma(m)$)are used. For categorical data [25], which is the case with the mapping where the values are devices, the best way to construct the probabilistic surrogate function, and thus to evaluate and search, is to use TPE. TPE is used by *constructing* (step 4) the surrogate function $P(t|m)$ by applying *Bayes rule*:

$$P(t|m) = \frac{P(m|t)P(t)}{P(m)} \tag{4.6}$$

where $P(m|t)$ is the probability of a mapping m given an actual makespan t.

$$P(m|t) = \begin{cases} l(m) & t < t_{th} \\ g(m) & t \geq t_{th} \end{cases} \tag{4.7}$$

t_{th} is the makespan threshold of the two distributions. Note that $l(m)$ and $g(m)$ both have normal distributions. With that said, EI would be:

$$EI(m) = \frac{l(m)}{g(m)} \tag{4.8}$$

A selection strategy would be to select m more toward the $l(m)$ distribution, given $\arg\max_m EI(m)$. As the Bayesian optimizer progresses through higher number of iterations, EI converges more toward exploitation than exploration, given that $P(t|m)$ becomes closer to $f_t(m)$. An overview of the Bayesian optimizer is shown in Figure 4.2.

4.3 Results

We generate 700 initial mappings using BO, GA, algorithmic, and random approaches. The mappings generated by the BO approach (a), the algorithmic approach (b),the GA approach (c), and the random approach (d) are compared (see Figure 4.3). Note that the number of training runs per mapping is equal to five in our case: this is in order to mitigate the time overhead incurred by reconstructing the graph with a different mapping.

In each case, we show the final distribution of makespans for the mappings generated by the different methods and show the final average of each method. Figure 4.4 and Figure 4.5 show the distribution for VGG-16 and AlexNet, respectively, with N=700. Note that the BO method has an overall lower mean and the distribution is skewed to lower makespans.

Figure **??** show further insight into how the averages of the makespan distributions change with time. The default mapping in this case is most effective, for now, given that this is the first stage. The BO approach shows steady improvement, meaning that better mappings are found with each iteration. A similar, but slower, trend is displayed in the GA method results. The algorithmic method starts off with relatively good mappings, but does not show much improvement with each iteration (although it does find a good mapping later on, which is not shown by the latest average but can be observed in the latest minimum figure): this is indicative of running out of good mapping *ideas*, where *intuition* does not pan out much further.

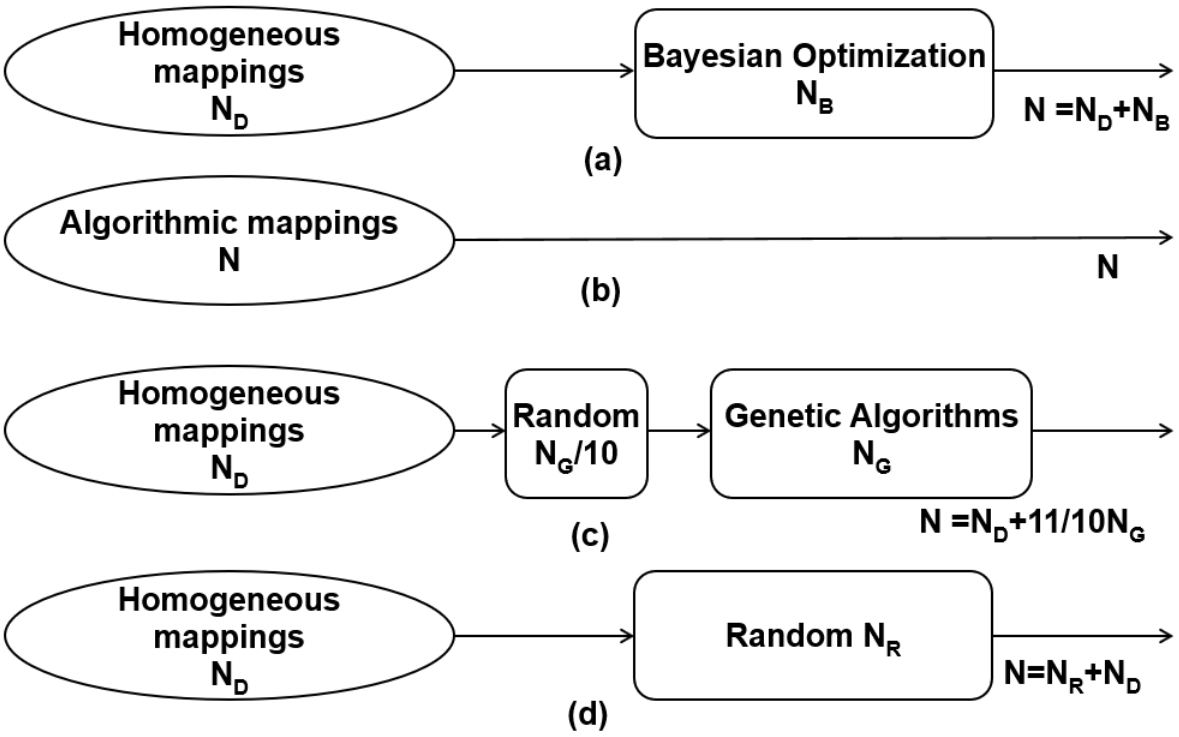


Figure 4.3: Configurations of initial mappings. **a)** is the Adaptive HTF-MPR approach (described in a later chapter) N_B is the number of mappings generated by the BO while N_D is the initial homogenous mapping (which is also used as a starting point for the BO). **b)** is the HTF-MPR approach (described in a later chapter). **c)** is using GA as the initial mappings where N_G is the number of mappings generated by the GA. Finally, **d)** uses the Random approach where N_R is the number of mappings generated randomly. Note that N is equal for all configurations. The number of mappings generated is N=700 in each case.

Figure 4.7 shows the latest minimum at each iteration. Note that Genetic, Algorithmic and Bayesian all eventually converge within the same neighborhood. Genetic seems to get there quicker while Algorithmic, and Bayesian get there at a later on iteration.

Important considerations for training time include not only the final makespan that is achieved, i.e. $f_t(m^*)$, but the whole process $F_t(\pi, oh_\pi)$. Figure 4.8 shows the overall time taken to carry out the first stage (before the ML stage and running the GA with the predictive model function). Note that in the initial stage the default outperforms the other methods; however, when used in conjunction with the later stages, a better mapping is found and

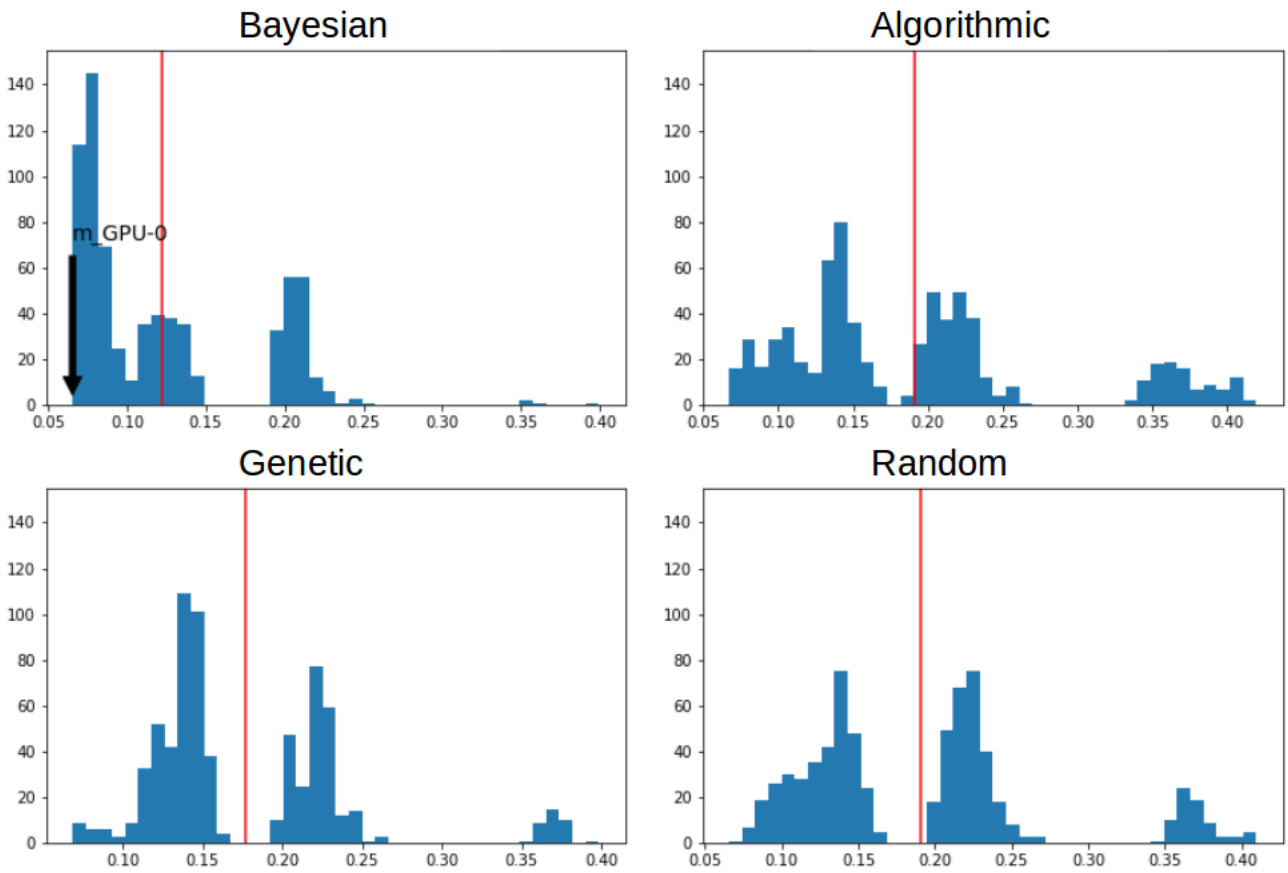


Figure 4.4: Makespan distribution for **VGG-16**. The x-axis is the makespan (seconds) and the y-axis is the count of mappings. The vertical red line indicates the average of the distribution. In the Bayesian figure, the makespan of the TF default mapping is indicated with a black arrow labeled m_{GPU-0}.

therefore a faster overall training time is achieved (as shall be shown in subsequent chapters regarding HTF-MPR and adaptive HTF-MPR).

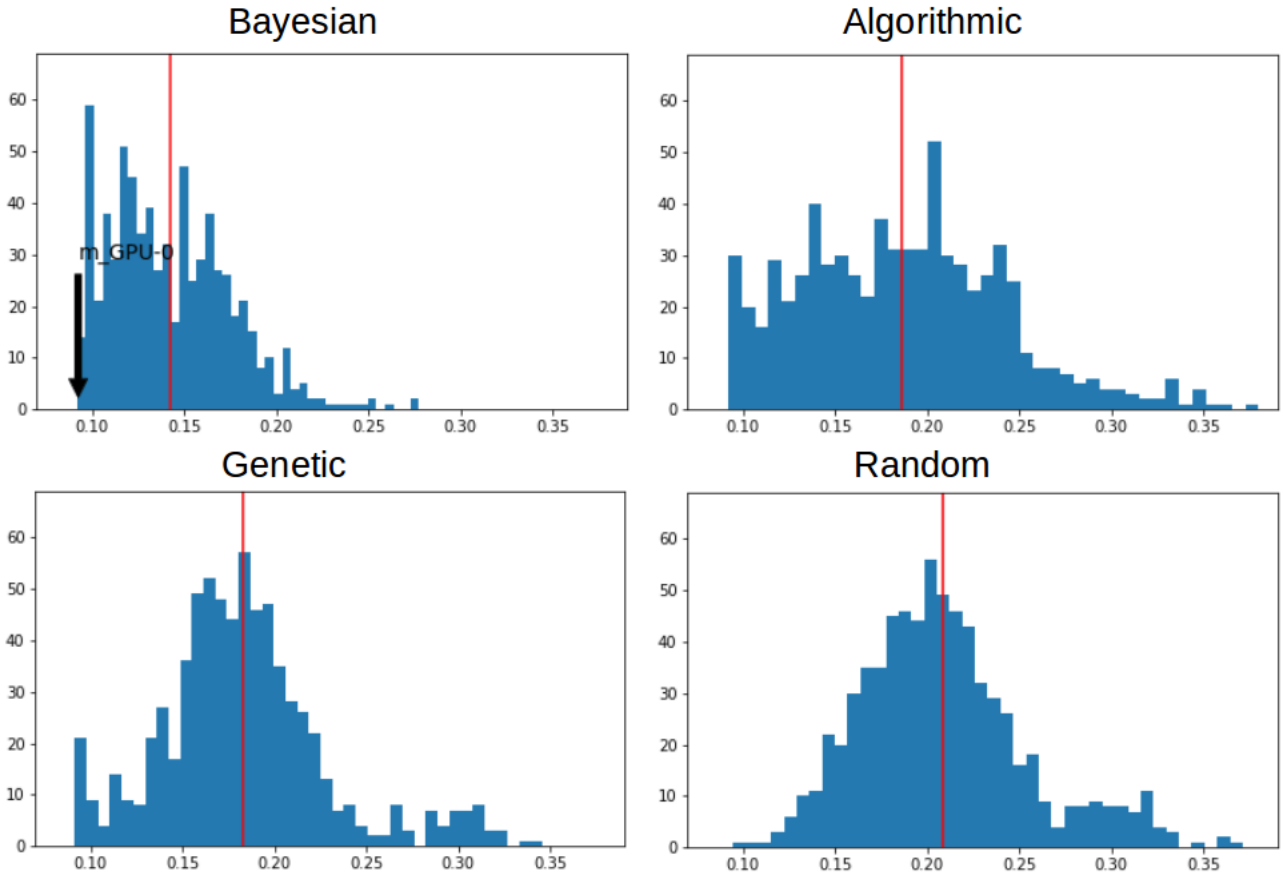


Figure 4.5: Makespan distribution for **AlexNet**. The x-axis is the makespan (seconds) and the y-axis is the count of mappings. The vertical red line indicates the average of the distribution. In the Bayesian figure, the makespan of the TF default mapping is indicated with a black arrow labeled m_{GPU-0}.

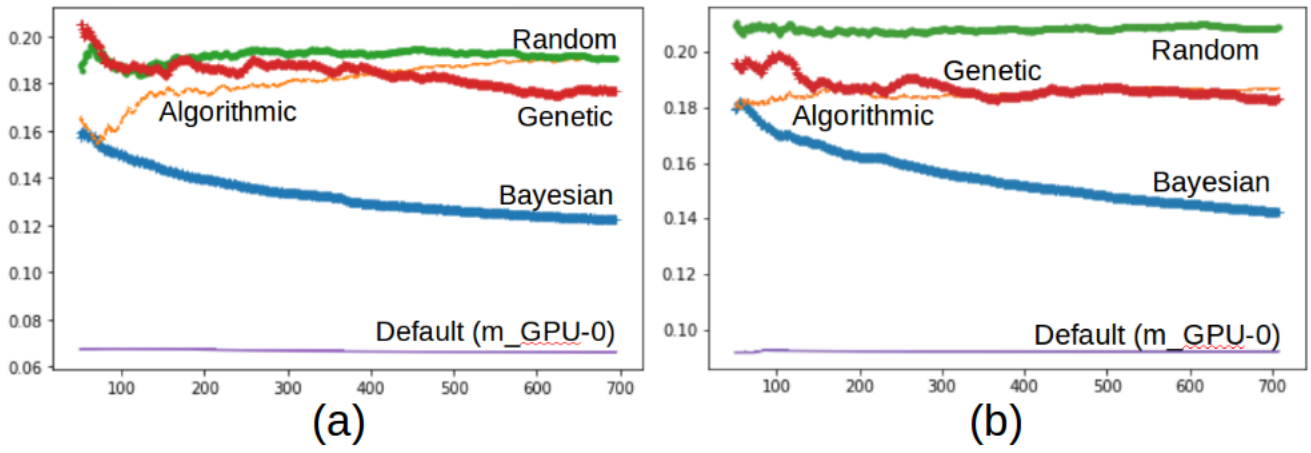


Figure 4.6: The latest average with each iteration for a) **VGG-16** and b) **Alexnet**. The x-axis shows iteration count, while the y-axis shows the average makespan (seconds). Note that the plot starts from iteration 50. the Bayesian improves with each iteration, same goes for the GA method.

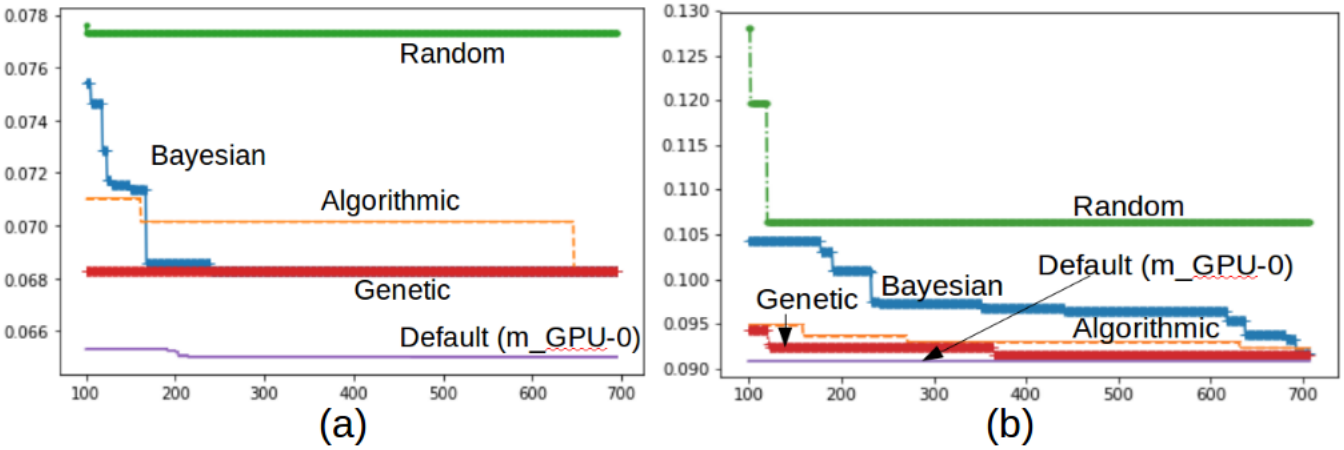


Figure 4.7: The latest minimum with each iteration for a) **VGG-16** and b) **Alexnet**. The x-axis shows iteration count, while the y-axis shows the minimum makespan (seconds). Note that the plot starts from iteration 100.

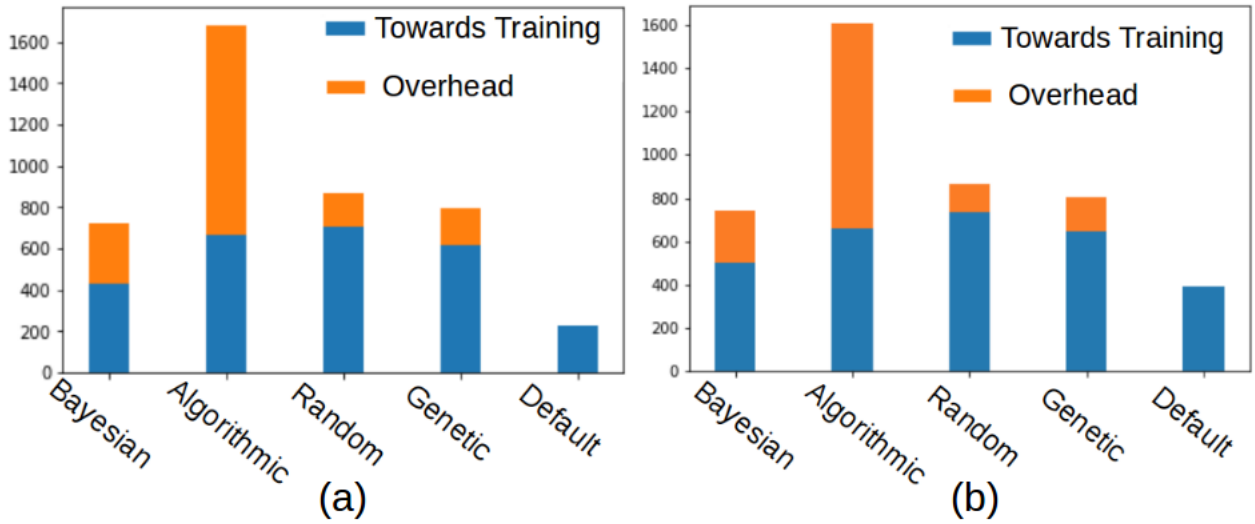


Figure 4.8: Total duration of first stage (Figure 4.3) for a) **VGG-16** and b) **Alexnet**. The total time is the sum of the overhead due to search and reconstruction of the graph with each new mapping, and the actual run of the f_{NN}, which contributes to the reduction of number of training iterations left. With the default TF model, there is no reconstruction of the graph as the mapping is constant; thus, there is no overhead. Note that with N=700, there are five training iterations per evaluated mapping; therefore, the figure shows the time for 700x5=3,500 training iterations of f_{NN}.

Chapter 5

Makespan Predictive Model

In this chapter, ML is introduced in relation to *regression* problems, and the different fundamental ML algorithms relating to regression are explained. Ensemble methods, the basis of gradient boosting regressors (GBR), is introduced to properly understand the GBR algorithm. The pipeline for the predictive model is showcased and the results of different ML models are compared with GBR.

The purpose of a makespan predictor $f'_t(m)$ is to speed up the overall training time F_t. With a reliable makespan predictor, it is possible to perform a search on M (the mapping solution space) in a fraction of the time that is required when using the results of $f_t(m)$; i.e., the actual run. The training set for building the predictor is $\{(m_i, t_{m_i})\}_{i=1}^{N}$.

5.1 ML in Regression

There are different ML algorithms that deal with regression problems: the class of problems where the output is a continuous real number. Each of these ML algorithms have benefits and drawbacks, and there is no one ML algorithm that is an all-encompassing solution to any

regression problem. Note that the makespan prediction is solved using a *supervised* learning technique, meaning that the true output t that results from $f_t(m)$, of a given m, is known and used to build the model $f'_t(m)$ in hopes of a resulting t' that is close to t.

5.2 Linear and Polynomial

The simplest type of ML model is a linear model. Assuming that m contains a single variable, the relationship then becomes:

$$t' = wm \tag{5.1}$$

where w is the parameter that needs to be tuned in order to have the relationship fit; i.e., to reduce the loss error.

$$L(t, t') = \sum_{i=1}^{N_m} (t^{(i)} - t'^{(i)})^2 = \sum_{i=1}^{N_m} (t^{(i)} - f'_t(m_i))^2 = \sum_{i=1}^{N_m} (t^{(i)} - wm_i)^2 \tag{5.2}$$

where the desire is to find a w that minimizes L. Note that N_m is the number of parameters in the model.

$$w^\star = \arg\min_{w \in R} L(t, t') \tag{5.3}$$

This can clearly be extended to a multi-variable input, meaning m now has multiple variables or multiple *features*. In this case, the ML model is still linear, since the output variable(s) are a linear combination of the input features. In *polynomial* regression ML models, the concept is the same, except that the model is a non-linear combination of the input features. Linear regression ML models work well in cases of non-complex relations between input and output, while polynomial regression models are very hard to design; i.e., it is complex to design the nonlinear functions that must be used. Gradient descent is used to update the $w \in W$ values, where W start at a random point and the values are then updated iteratively.

5.3 Ensemble Models

Ensembling is the process of combining multiple models, which may be referred to as sub-models. Each of these ML sub-models learns and updates its parameters in order to reduce the errors in a given training dataset. This is done prior to the models being combined into the ensemble. In regression, the output prediction of the sub-models can be combined in a simple form either by *averaging*, or by taking a *weighted* average of the outputs. This form of combining or ensembling is referred to as *bagging* in the literature. Figure 5.1 illustrates the process by which an ensemble is constructed using bagging. *Boosting* is another form of ensembling. Unlike bagging, the sub-model is trained using the entire dataset. Then, the weak dataset predictions are trained on subsequent sub-models. Boosting is thus a sequential method. Figure 5.2 illustrates the process by which an ensemble is constructed using boosting.

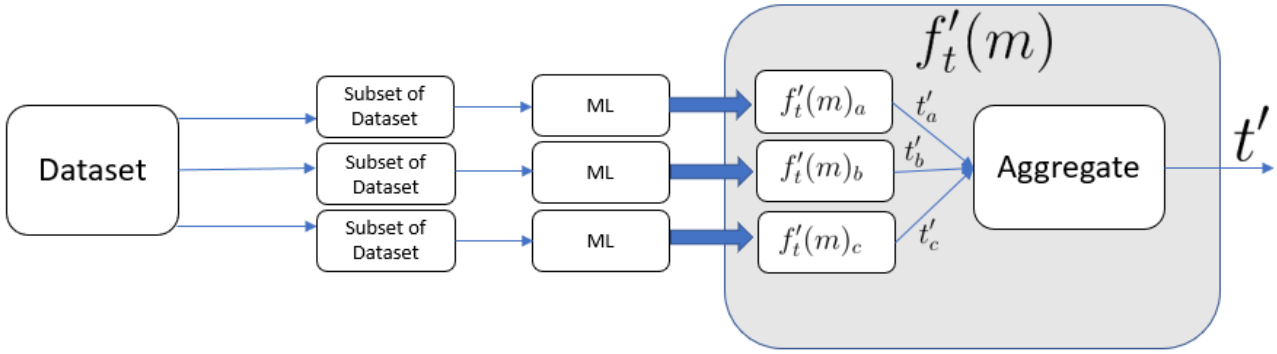


Figure 5.1: Ensemble created using bagging method. The sub-models ($f'_t(m)_a$, $f'_t(m)_b$, $f'_t(m)_c$) are trained in parallel. Each sub-model uses a different subset of the dataset to train.

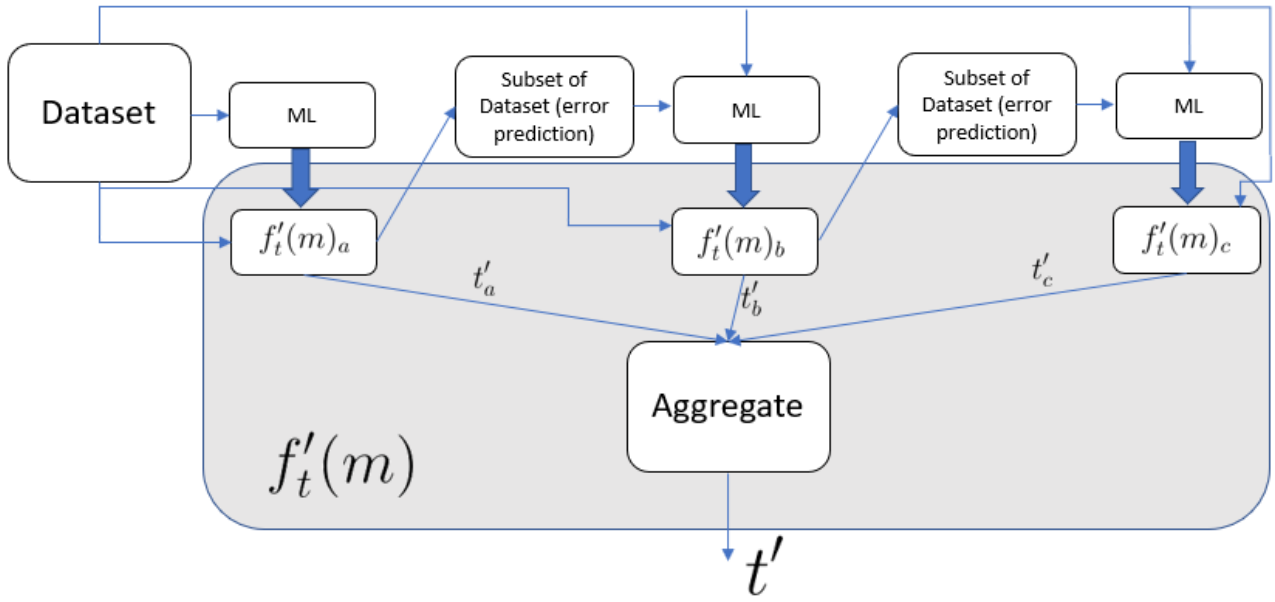


Figure 5.2: Ensemble created using boosting method. The sub-models are trained in sequence. The sub-dataset of erroneous predictions from $f'_t(m)_a$ is added to the training dataset to train the model that produces $f_t(m)_b$.

5.4 Encoding

The purpose of encoding is to change the features of data points from one representation to another. We demonstrate two types of encodings: one-hot encoding and integer encoding. Each feature input is henceforth referred to as a variable.

In *one-hot encoding*, a variable is *expanded* to multiple variables, each taking on a value of either 0 or 1. Exactly one of the expanded variables from the original variable is assigned a value of 1, while the rest are set to 0. In the case of a mapping m, the the size of m, or the number of variables of m, without the one-hot encoding is $|m| = N_{op}$. If one-hot encoding is applied then the size would be the multiple of the number of values each operation would take; i.e., the number of devices. More formally, $|m^{one-hot}| = N_{op}\dot{N}_D$. Figure 5.3 illustrates the difference between *integer encoding*, as is applied in HTF-MPR, and one-hot encoding, as is applied in Adaptive HTF-MPR.

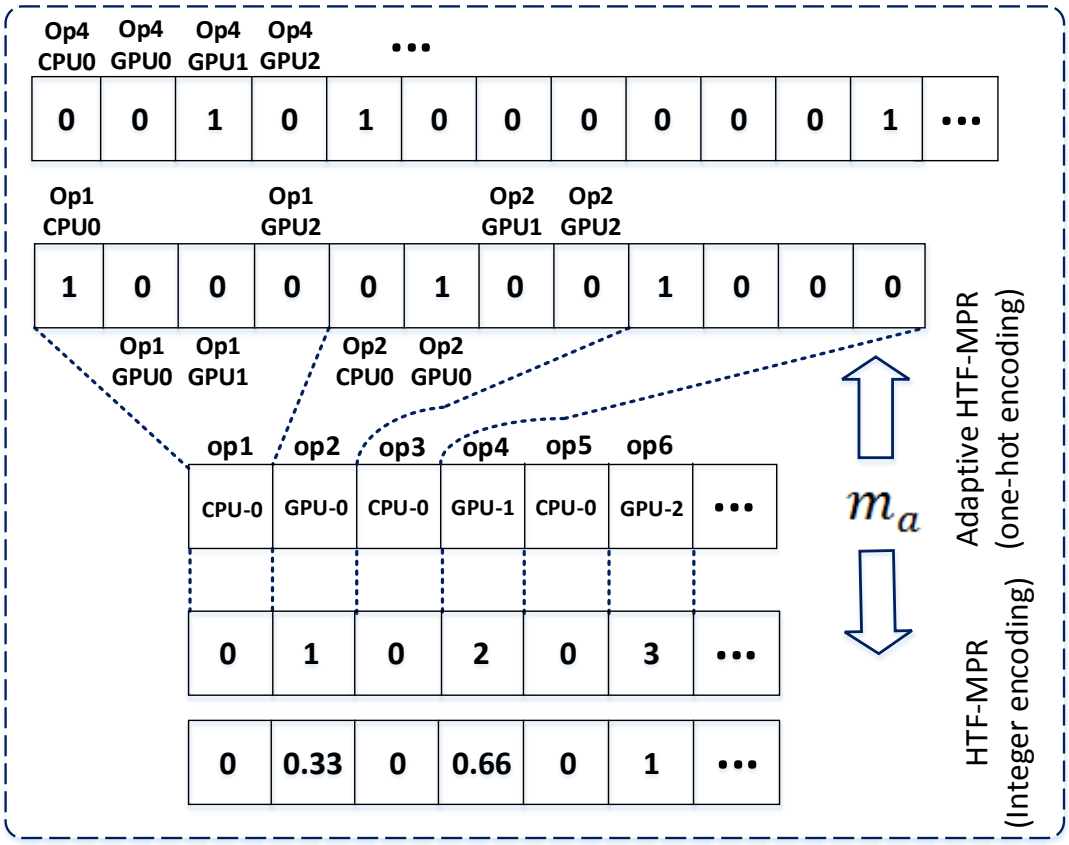


Figure 5.3: Encoding: m_a is encoded using integer encoding, where CPU-0$\longrightarrow$ 0, GPU-0$\longrightarrow$ 1, GPU-1$\longrightarrow$ 2, and GPU-2$\longrightarrow$ 3. The integers are then *normalized*. The top part illustrates one-hot encoding, where *dummy variables* are used. This increases the number of *features*; in this case, a single variable is expanded to four, since there are four devices. Note that CPU-0$\longrightarrow$ 1000, GPU-0$\longrightarrow$ 0100, GPU-1$\longrightarrow$ 0010, and GPU-2$\longrightarrow$ 0001.

Categorical variables have nominal values, meaning that their values have a qualitative property rather than a quantitative property. Integer encoding (as is applied in HTF-MPR, see Figure 5.3) assumes *order*; that is, numbers have an order in relation to each other. Thus, CPU-0 does not have a closer relationship to GPU-0 than it does to GPU-1. If the integer 0 is assigned to CPU-0, 1 is assigned to GPU-0, 2 is assigned to GPU-1, etc., we have implicitly assigned relations. These relations have an effect when used mathematically in ML models. Since no ordinal relationship between the devices exists, one-hot encoding is more suitable.

5.5 Gradient Boosting Regression

Based on investigation of several ML algorithms, the *gradient boosting regression* (GBR) [23] algorithm outperforms others tested in terms of the *Kendall tau rank distance* [41] metric. GBR consists of weak learners that are assembled together and made into an ensemble of a strong prediction model. This ensembling of weak learners occurs after each iteration where the new weak learner improves upon the whole predictive model; therefore, the weights, as well as the hyperparameters, are adjusted during training:

$$f'_{t,k+1}(m) = f'_{t,k}(m) + h(m) = t_m \tag{5.4}$$

where $f'_{t,k+1}(m)$ is the makespan predictor at step $k+1$ of its training, which is made up of the previous predictor $f'_{t,k}(m)$ and an estimator $h(m)$. Therefore, the final makespan predictor $f'_t(m)$ is made up of many weak predictors:

$$f'_t(m) = \sum_{j=1}^{n} h_j(m)\gamma_j + const. \tag{5.5}$$

where n is the total number of training iterations required to construct the predictive model. γ is an optimized coefficient that is multiplied by the weak learner i.e. the weight of the learner. Training happens in an incremental manner, initially set as:

$$f'_{t,0} = \arg\min_{\gamma} \sum_{i=1}^{N} L(t_i, \gamma) \tag{5.6}$$

where $f_t(m) \longrightarrow t_m$, and during the training of the predictive model. N mappings are used as input X, and N timings are used as the output Y (see Figure 6.1, *ML algorithm for training*). Subsequently, the makespan predictive model is updated by computing the *residual*:

$$r_j(m_i) = -\left[\frac{\delta L(t_{m_i}, f'_{t,j-1}(m_i))}{\delta f'_{t,j-1}(m_i)}\right], \text{for } i = 1, ..N \tag{5.7}$$

Then, the *base learner*, i.e. estimator $h_j(m)$, is constructed using the residual $r_j(m)$ and input m. Therefore, the training set for $h_j(m)$ is $\{(m_i, r_j(m_i)\}_{i=1}^{N}$. The γ_j is then updated:

$$\gamma_j = \arg\min_{\gamma} \sum_{i=1}^{N} L(t_i, f'_{t,j-1}(m_i) + \gamma h_j(m_i)) \tag{5.8}$$

The model is then updated as demonstrated in Equation 5.4:

$$f_{t,j}(m) = f_{t,j-1}(m) + \gamma_j h_j(m) \tag{5.9}$$

This whole process is repeated n times, resulting in a final predictive model $f'_t(m)$, which is used by the GA.

5.6 Metrics

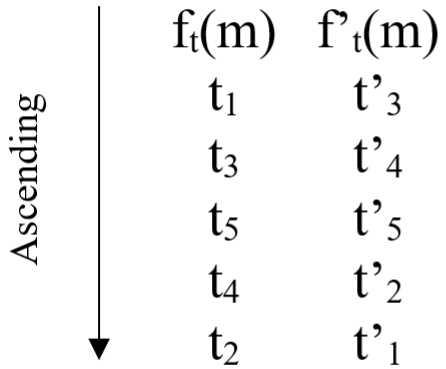
Given two mappings m_a and m_b, the *Kendall number* is calculated as follows:

$$k(t_a, t_b, t'_a, t'_b) = \begin{cases} 1, & \text{if } t_a < t_b \text{ and } t'_a > t'_b. \\ 1, & \text{if } t_a > t_b \text{ and } t'_a < t'_b. \\ 0, & \text{otherwise.} \end{cases} \tag{5.10}$$

where $f_t(m_a) \longrightarrow t_a$ and $f_t(m_b) \longrightarrow t_b$ are the actual makespans of mapping m_a and m_b, respectively, and $f'_t(m_a) \longrightarrow t'_a$ and $f'_t(m_b) \longrightarrow t'_b$ are the predicted makespans of m_a and m_b, respectively. A value of **1** indicates a mismatch in the pair-wise order between the actual and the predictive makespans, and **0** indicates a preserved ordering. The normalized Kendall tau ranking distance is thus:

$$K_{norm} = \sum_{i} \sum_{j<i} \frac{2 \cdot k(t_i, t'_i, t_j, t'_j)}{N(N-1)} \tag{5.11}$$

Figure 5.4 shows an example of five mappings m_1, m_2, m_3, m_4, m_5, where the actual makespans are $t_1 < t_3 < t_5 < t_4 < t_2$ and the predicted makespans are $t'_3 < t'_4 < t'_5 < t'_2 < t'_1$.



$f_t(m)$	$f'_t(m)$	k
$t_1 < t_2$	$t'_1 > t'_2$	1
$t_1 < t_4$	$t'_1 > t'_4$	1
$t_1 < t_5$	$t'_1 > t'_5$	1
$t_1 < t_3$	$t'_1 > t'_3$	1
$t_3 < t_2$	$t'_3 < t'_2$	0
$t_3 < t_4$	$t'_3 < t'_4$	0
$t_3 < t_5$	$t'_3 < t'_5$	0
$t_5 < t_2$	$t'_5 < t'_2$	0
$t_5 < t_4$	$t'_5 > t'_4$	1
$t_4 < t_2$	$t'_4 < t'_2$	0

Figure 5.4: An example of the Kendall values for five makespans. The resulting $K_{norm} = 0.5$.

The K-fold method used to validate the predictive model is shown in Figure 5.5. The mappingsfitness pairings are shuffled and then partitioned into k parts. The predictive model $f'_t(m)_i$ is trained using all the partitions except for partition i. Partition i is then used as a validation measure to observe the normalized Kendall tau ranking distance. This process is repeated k times, using a different partition i for the validation each time. Figure 5.5 shows how this process is carried out.

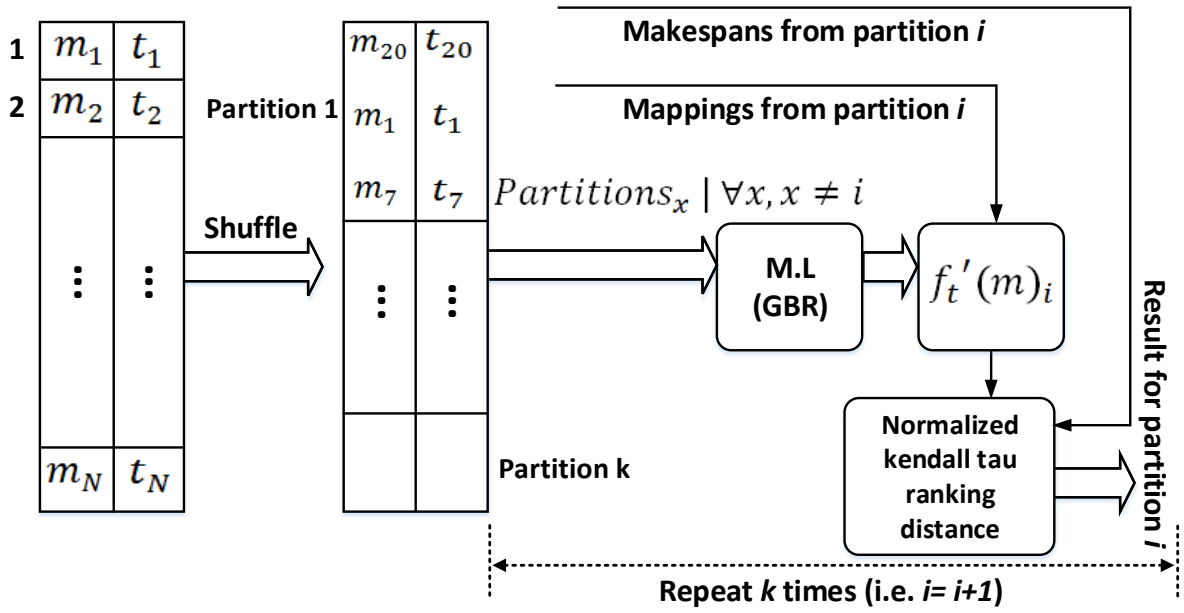


Figure 5.5: K-fold method of validation. The mappings (input) and the makespan timings (labels) are shuffled, then split into k parts. A partition is selected to be the test dataset, while the rest of the partitions are used for training the model using GBR. The resulting predictive model is then tested using the test dataset partition. The Normalized Kendall tau ranking is noted and the process is repeated, with a different partition used as the test dataset each time. Note that the use of m_{20}, m_1, and m_7 is arbitrary for illustrative purposes to indicate that the dataset is shuffled.

5.7 Results

We have applied k-fold cross validation across a number of ML algorithms. The number of mappings used is different for each benchmark, while the value of $k = 5$ is used for all benchmarks:

benchmark	$\mathbf{M}_p$ Size	M Size	Training Size	Testing Size
ALEXNET	105	55	84	21
MNIST softmax	135	10	108	27
VGG-16	105	69	84	21

The average results are shown in Fig 5.6. GBR outperforms all other ML algorithms across the board. The reason why decision tree, Adaboost and GBR outperform the other ML models considerably is that $f_t(m)$ is not only nonlinear but discontinuous; therefore, it requires a tree-like and ensemble-like model structure.

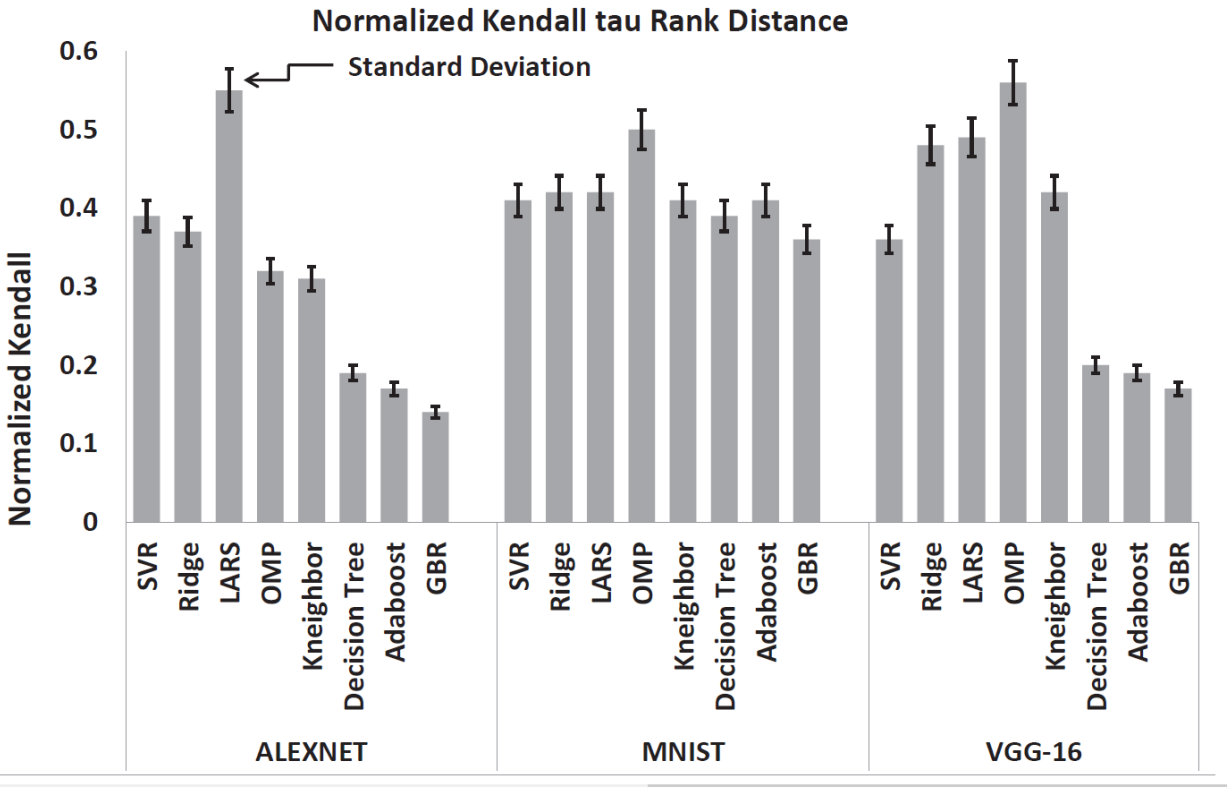


Figure 5.6: Predictive model performance using k-fold (k=5) and different ML algorithms. The chart shows the average results from five runs and includes the standard deviation of the five runs. **SVR**: Support Vector Regression; **Ridge**: Ridge Regression; **LARS**: Least Angle Regression; **OMP**: Orthogonal Matching Pursuit; **Kneighbor**: Regression-based on *k-nearest* neighbors.

The results are shown in Figure 5.7. We compare the use of one-hot encoding and integer encoding, where the training dataset is generated either by BO (as in the case of Adaptive HTF-MPR) or an algorithmic approach (as in the case of HTF-MPR) both generation methods were described in the previous chapter. Note that in each case the one-hot encoding outperforms the integer encoding. The performance will affect how many mappings will be

chosen for evaluation; that is, the *top K* mappings after the GA stage (to be described in Chapter 6).

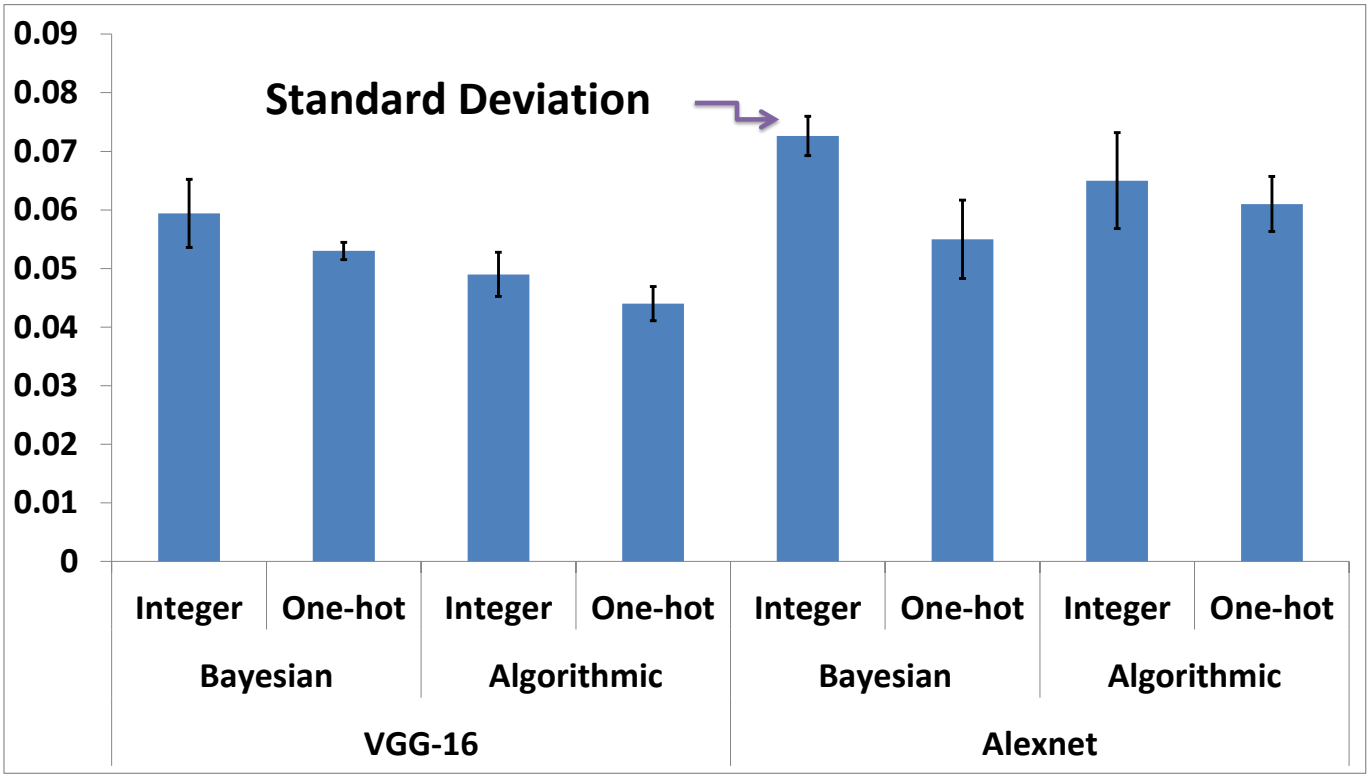


Figure 5.7: K-fold results. The y-axis is the normalized Kendall, where a lower number indicates a lower error rate. Note that N=700 (number of mappings) and K=5 (number of folds). The bar indicates the average of the five runs (normalized Kendall of five tested partitions) and the standard deviation shown is due to the difference of the five runs.

Chapter 6

Heterogeneous TensorFlow Mapper

In this Chapter, HTF-MPR and Adaptive HTF-MPR are explained. The point of using these frameworks is to find a sub-optimal mapping that outperforms TF's mapping. The duration of the whole search process should outperform TF default mapping (where the TF default does not have the overhead of searching a sub-optimal mapping).

6.1 HTF-MPR

HTF-MPR [4] is a framework that finds a better *device-to-operations* mapping in order to speed up execution times of TF computational graphs. Mappings are evaluated by measuring the execution's runtime using a particular mapping; i.e., $f_t(m) \longrightarrow t_m$. Once a reasonably sized sample of mappings and their speeds is collected, a *predictive model* $f'_t(m)$ is produced. The reason for the use of a predictive model, rather than the actual run, is that the makespan of a certain mapping is returned almost *750 times faster*. In other words, 750 mappings can be analyzed using the makespan predictive model compared to one mapping using the actual run. Note that there are accuracy issues with the predictive model, as is the case with any

model; therefore, the best mappings according to $f'_t(m)$ are run again to evaluate their $f_t(m)$. An overview of HTF-MPR is shown in Figure 6.1.

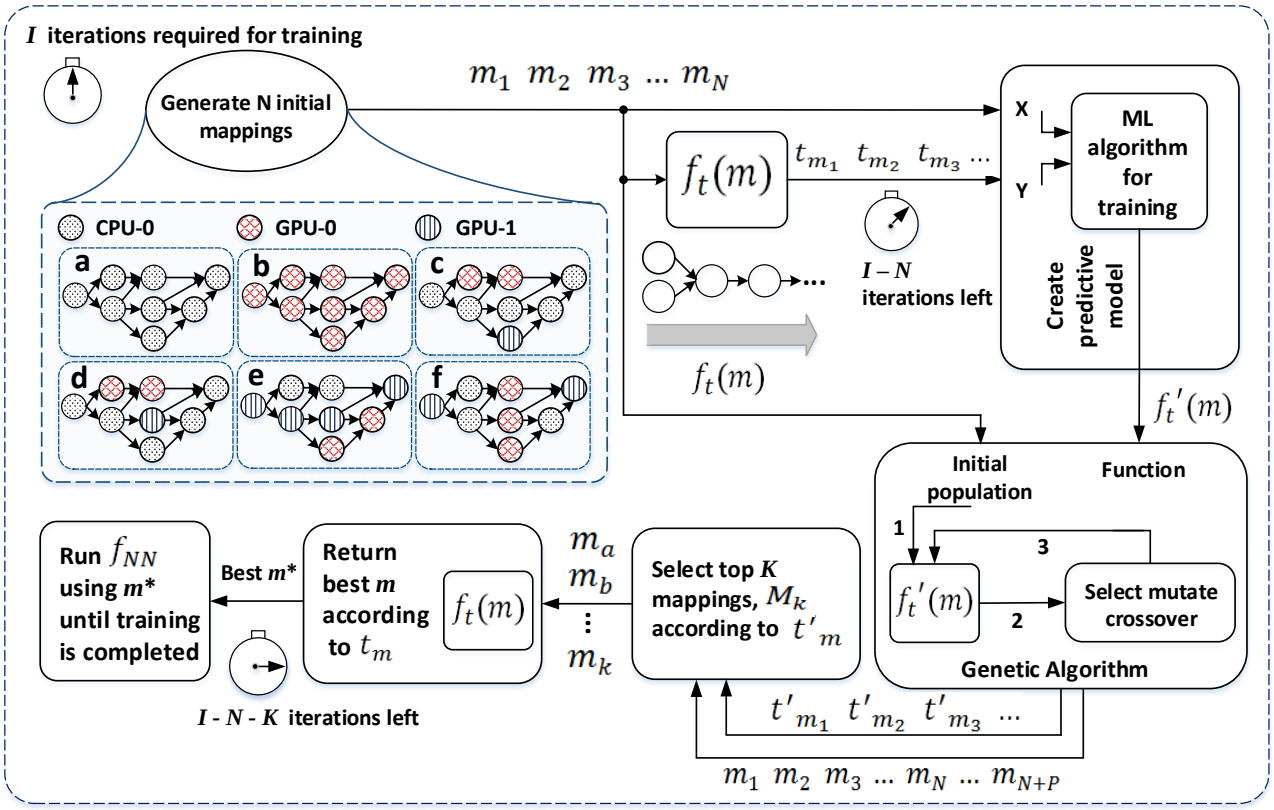


Figure 6.1: HTF-MPR overview: **1.** N initial mappings are generated (Chapter 4). **2.** These mappings are then run on the TF graph, where their makespans, $f_t(m) \longrightarrow t_m$, are recorded. The number of iterations left to train the model (and therefore get it closer to the final model f_{NN}) is $I - N$. **3.** The input data X and output data Y are used to construct the predictive model (Chapter 5). **4.** The predictive model as well as the mappings are provided to the GA (Subsection 6.1.1). **5.** Top mappings are selected according to the predicted makespans (Subsection 6.1.2) . **6.** The top mappings are then run on the TF graph to obtain actual makespans $f_t(m)$. The number of training iterations is advanced by K (the number of top mappings), thus reducing the required runs to $I - N - K$. **6.** Finally, the top mapping, m^*, is found and used for the rest of the training; i.e., for $I - N - K$ iterations.

6.1.1 Search with Genetic Algorithm

Genetic algorithms (GAs) [50] are a type of optimization technique that is *metaheuristic*, meaning they are designed to work on non-deferential and non-linear search spaces [26]. They

are known for solving *task-mapping* [33] problems. In HTF-MPR, the GA uses $f'_t(m)$ as the inverse *fitness* of a particular *solution*, i.e. mapping. The fitness of a solution is proportional to the likelihood of its being chosen as one of the parents to generate a new solution. This new solution is assessed using $f'_t(m)$ and added to the *population*. The search process is shown in the GAs part of Figure 6.1. Initially, the algorithmically generated mappings are provided to the GA, where the t'_m of each mapping is calculated. Then, two parents with a probability proportional to the fitness are selected; i.e., the inverse of t'_m, whereby these two parents generate a new mapping via *crossover*. In our approach we use two methods for crossover (Figure 6.2 and Figure 6.3). Figure 6.2 shows a *stochastic* approach to generating a new mapping. The fitness of the parent dictates the percentage of operation mapping that the new mapping will inherit from that parent. Figure 6.3 shows another approach, where

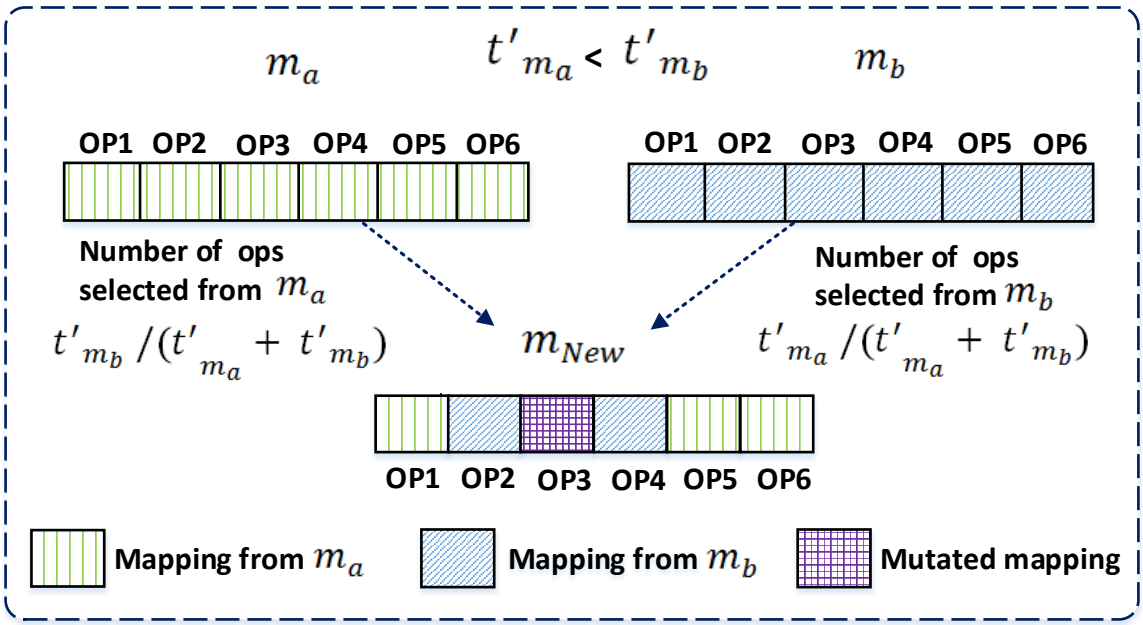


Figure 6.2: Crossover using a stochastic method whereby the number of mappings taken from a particular parent is relative to how fit the parent is. In this case, m_a is more fit than m_b given the lower predicted makespan; i.e., $t'_{m_a} < t'_{m_b}$. Therefore, more operation mappings are copied from m_a than m_b. Some operations mappings also go through *mutation*, meaning they are not copied from either parent. In this example, *op*3 has been mutated.

the crossover points dictate the number of newly generated mappings. For example, if there were two crossover points, then *at most* six new mappings would be generated from the

parents (see Figure 6.3). If there were three crossover points, then *at most* 14 new generated mappings would occur; i.e., at most $2^{N_c+1} - 2$ generated mappings, where N_c is the number of crossover points.

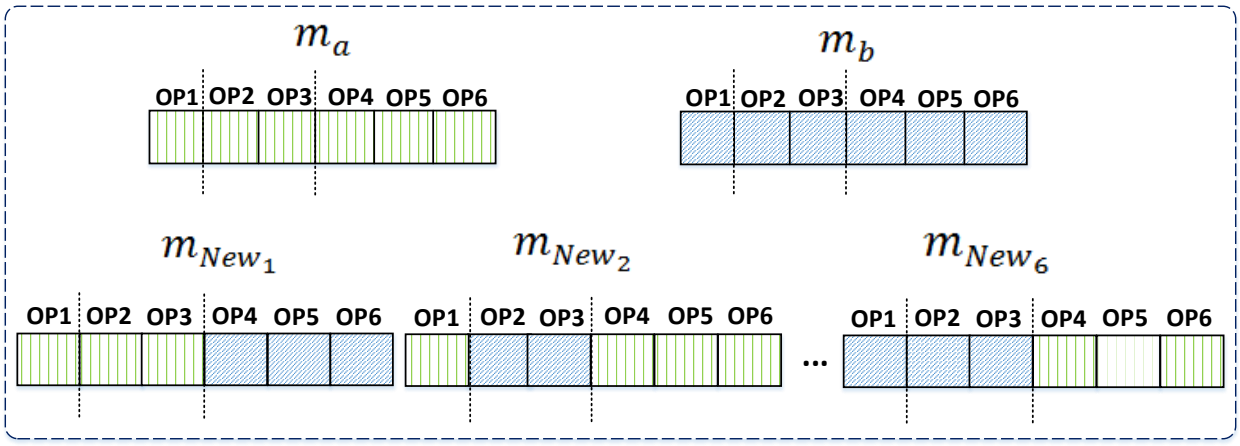


Figure 6.3: Crossover using crossover points. In this example, six new mappings are generated from the parents m_a and m_b.

6.1.2 Final Selection

P new mappings, and their predicted makespans t'_m, are generated by the GA as shown in Figure 6.1. These mappings are then sorted in t'_m ascending order. The top K mappings are then selected and run on the TF graph to obtain the actual makespans t_m. The top mapping m^*, according to t_m, is then run until the training of the TF graph is completed, thus reaching the final state of the model f_{NN}. Note that the training advances when the model is run. During a run, regardless of the mapping used, there is the added benefit of acquiring $f_t(m)$ while not affecting the path of the training. This indicates that the final destination of f_{NN} is the same, with the only difference being how quickly a given model can reach it.

6.2 Adaptive HTF-MPR

6.2.1 Overview

Adaptive HTF-MPR uses a similar methodology to HTF-MPR with some modifications (see Figure 6.4). One modification is the introduction of the BO [55] step using $f_t(m)$ as the function for performance evaluation. The aim of applying BO is to find the locale, or neighborhood, of the best mappings via intelligent search. The resulting mappings are then used to construct the makespan predictive model $f'_t(m)$. Another modification is the removal of the initial mappings using the algorithmic approach. This is due to using the results of the BO as input to the ML to create the makespan predictive model as well as the initial population for the GA. The input or initial starting point for BO consists of the homogeneous mappings.

6.2.2 Adaptivity

The training time for some state-of-the-art neural networks could reach hundreds of thousands of iterations [74], with each iteration taking a certain period of time depending on the employed hardware and the batch size of the input data. There is no guarantee that either the state or performance of the system will remain consistent throughout the training; i.e., parts of the system's hardware, CPUs or GPUs, could have different loads at different times due to external processes. This would affect the makespan and thus the training time. To combat this, the makespan time has to be monitored. The monitoring module would detect any drastic divergence from the average performance level, whether representing improvement or degradation. If one of the system components (i.e., the load of one of the devices) were to increase or decrease, it would affect $f_t(m)$ and thus changes what could be considered m^*. In this paper, among other things, we add a monitoring mechanism and

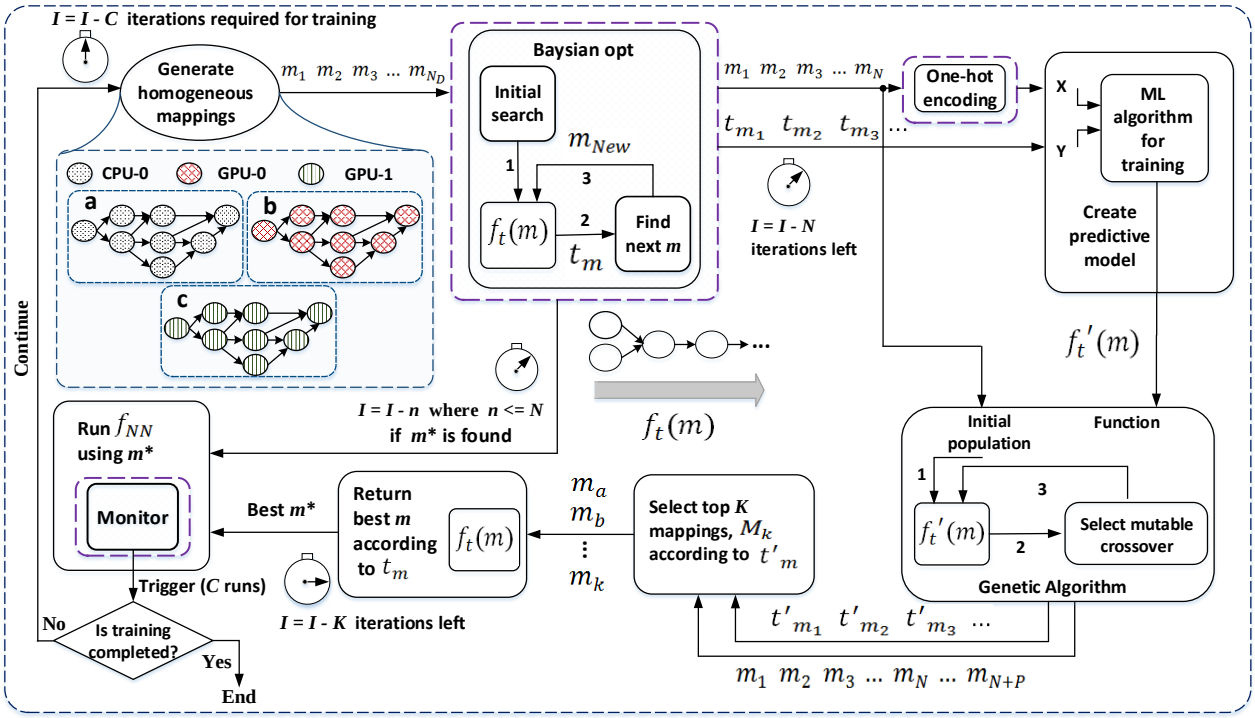


Figure 6.4: Adaptive HTF-MPR overview: **1.** N initial mappings are generated using BO (Chapter 4). **2.** Mappings are then run on the TF graph where their makespans, $f_t(m) \longrightarrow t_m$, are recorded. The number of iterations left to train the model (and therefore get it closer to the final model f_{NN}) is $I - N$. **3.** Input data X are turned to one-hot encoding (Chapter 5) and makespan predictive model is constructed (Chapter 5). **4.** GA is run (Subsection 6.1.1) until population size reaches P. **5.** Top mappings are selected according to the predicted makespans. **6.** The top K mappings are then run on the TF graph to obtain the actual makespans $f_t(m)$. The number of training iterations is advanced by K, thus reducing the required runs to $I - N - K$. **6.** The top mapping, m^*, is identified and used for the rest of the training. The monitor triggers a rerun of the process if required.

a way to deal with and *adapt* to these changes in the system. Our Adaptive HTF-MPR takes corrective measures to find a new m^* once the monitoring module sets a trigger. Our monitoring module works with both gradual, slow changes [72] and abrupt changes [6].

6.2.3 Training the Predictive Model

As in HTF-MPR, we train a surrogate function $f'_t(m)$ to be used in the GA. Using the mappings generated by the BO evaluations, we train in order to create a makespan predictive

model $f'_t(m)$ using GBR as explained in Chapter 5. Note that the two main differences between the predictive model used in HTF-MPR and that of Adaptive HTF-MPR are:

- The training dataset uses mappings that are skewed more toward better performing makespans:

$$\sum_{m \in M_{Bayesian}} f_t(m) < \sum_{m \in M_{initial}} f_t(m) \tag{6.1}$$

  where $M_{Bayesian}$ and $M_{initial}$ are the mappings generated by the Bayesian optimizer (Adaptive HTF-MPR) and the initial mappings (HTF-MPR), respectively. In addition, $|M_{Bayesian}| = |M_{initial}|$, so as to make the comparison from Equation 6.1 fair.

- One-hot encoding is used rather than normalized integer encoding. This is a better fit given that the values in the mapping are non-ordinal categorical data.

6.2.4 Genetic Algorithm Search

As in HTF-MPR, GA is used to search for an optimal mapping that outperforms TF's default GPU homogeneous mapping, using the makespan predictive model $f'_t(m)$ as the surrogate function to evaluate performance of a given solution. The differences here are that:

- The initial population consists of the mappings from the Bayesian optimizer, meaning a more concentrated search space.

- A makespan predictive model $f'_t(m)$ that is designed to work well within the neighborhood of the search space of the initial population.

6.2.5 Adaptive Run

During the run on m^*, both the average and the standard deviation are taken for a window size of Q iterations of $f_t(m^*)$. If, after the Q iterations, $f_t(m^*)$ changes to become higher or lower than βx of the standard deviation, this causes a trigger to occur. The trigger would start the Adaptive HTF-MPR process again. Note that the number of iterations left for training and reaching the final trained model f_{NN} would be reduced (see Figure 6.4).

Initialization:
set the mean:
$\mu_{win} = \frac{1}{Q}\sum_{i=1}^{Q} f_t(m^*)_i$;
set the standard deviation:
$\sigma_{win} = \sqrt{\frac{\sum_{i=1}^{Q}(f_t(m^*)_i - \mu_{win})^2}{Q}}$;
set upper-bound:
P_trigger = $\mu_{win} + \beta\sigma_{win}$;
set lower-bound:
N_trigger = $\mu_{win} - \beta\sigma_{win}$;
start location of monitoring: i = Q + 1;
Trigger=False;

while f_{NN} *still training* **do**
  Advance f_{NN} training;
  i=i+1;
  if *N_trigger*$< f_t(m^*)_i <$ *P_trigger* **then**
    Trigger=True;
    Break from while loop;
  end
end
if *Trigger* **then**
  run Adaptive HTF-MPR on f_{nn} from iteration i
end

Algorithm 1: Monitoring algorithm: The average makespan of each run is taken for a window size of Q. The standard deviation is recorded, and the triggers are set. While running the neural network on mapping m^*, we check the current makespan. If the makespan is above the P_trigger or lower than the N_trigger, a trigger is set and Adaptive HTF-MPR is run again.

A trigger indicates that there has been a change in the hardware state; either a drop or an improvement in performance (both could be either gradual or abrupt). In either case this would require a reassessment of the values of $f_t(m)$ and therefore a search for a new m^*. Algorithm 1 shows the monitoring mechanism.

6.3 Results

6.3.1 Genetic Algorithm on Predictive Model

In this section the results of the GA on the predictive model are presented. The factors that are essential in evaluating the performance of this part are the following:

- The time it takes to search using the GA on the makespan predictive model $f'_t(m)$. The time is indicated by T_{N+P}, while the size of the search is $N + P$.
- The results of the search. The first occurrence of a mapping that has a makespan $f_t(m^*)$ better than the default mapping $f_t(m_{TF})$ makespan.
- What number of evaluations K, using $f_t(m)$, are needed to find the best possible makespan $f_t(m^{**})$ among the ensuing GA results. Number of evaluations is correlated to time T_K. Note that $f_t(m_{TF}) > f_t(m^*) \geq f_t(m^{**})$.

	Model	$N+P$	T_{N+P}	K	T_K	i^*	$\frac{f_t(m_{TF})}{f_t(m^*)}$	$rank'(m^*)$	$rank(m^*)$	i^{**}	$\frac{f_t(m_{TF})}{f_t(m^{**})}$	$rank'(m^{**})$
1	Bayesian	10,000	25.7s	1,000	770s	1021	1.04	6	12	7836	1.205	16
2	Bayesian	100,000	1012s	1,000	768s	74611	1.04	1	521	10026	1.209	9
3	Bayesian	10,000	25.36s	100	77.6s	7428	1.036	2	37	7290	1.201	69
4	Algorithmic	10,000	22.22s	1,000	769s	2935	1.04	4	89	9342	1.19	58
5	Algorithmic	100,000	1099s	1,000	743s	8002	1.038	7	51	10247	1.204	81

Table 6.1: GA results using predictive model $f'_t(m)$ on **AlexNet**.

	Model	$N+P$	T_{N+P}	K	T_K	i^*	$\frac{f_t(m_{TF})}{f_t(m^*)}$	$rank'(m^*)$	$rank(m^*)$	i^{**}	$\frac{f_t(m_{TF})}{f_t(m^{**})}$	$rank'(m^{**})$
1	Bayesian	100,000	1014s	1,000	752s	1	1.00	1	1	1	1.0	1
2	Bayesian	150,000	1806s	1,000	765s	100179	1.06	56	3	130775	1.14	87
3	Algorithmic	100,000	920s	1,000	698s	1	1.00	112	1	1	1.00	112
4	Algorithmic	150,000	1846s	1,000	703s	1	1.00	18	1	1	1.00	18

Table 6.2: GA results using predictive model $f'_t(m)$ on **VGG-16**.

As indicated in [21], the initial population is an important metric for the GA. Table 6.1 and Table 6.2 show that the initial populations generated by the Bayesian optimizer outperform

the algorithmic initial population in both instances. Regarding the size of the search, 10,000 searches in the GA and 100 evaluations prove sufficient to identify the best mapping in AlexNet. For VGG-16, the search space is larger; therefore, 150,000 searches are required m_{TF}.

6.3.2 Run and Adaptivity

In this section the full runs of the TF default mapper, the HTF-MPR, and the Adaptive HTF-MPR are presented. In addition, a stress test is applied on the system and the changes of the makespan are observed. We see how Adaptive HTF-MPR reacts and how it affects the overall training time. The total training times for VGG-16 and AlexNet are shown in Figure 6.5. The overhead with AlexNet is low due to the fact that the GA part is not run for long (only 10,000 mappings). The GA slows down over time and does not have a linear relationship with number of iterations, as can be seen from Table 6.1: when comparing 10,000 runs and 100,000 runs, the increase in T_{N_P} is 40x, while the number of GA iterations increases by only 10x.

Figure 6.6 shows what happens to the makespan when a high load is applied.

We apply a high load on GPU-0 for a 30-minute duration. The makespan per iteration is shown in Figure 6.7. The performance of the predictor worsens when Adaptive HTF-MPR is triggered (due to the GPU-0 high load). The reason for the low performance of the predictor is the high variance of the makespan (see Figure 6.6).

Depending on the load duration and how sporadic the load is, the adaptive component performs accordingly. In the case of high variance (sporadic load) the makespan predictor is unable to generate a single-point prediction. In case of bumps over or below the P_trigger and N_trigger, respectively, Adaptive HTF-MPR would perform as usual.

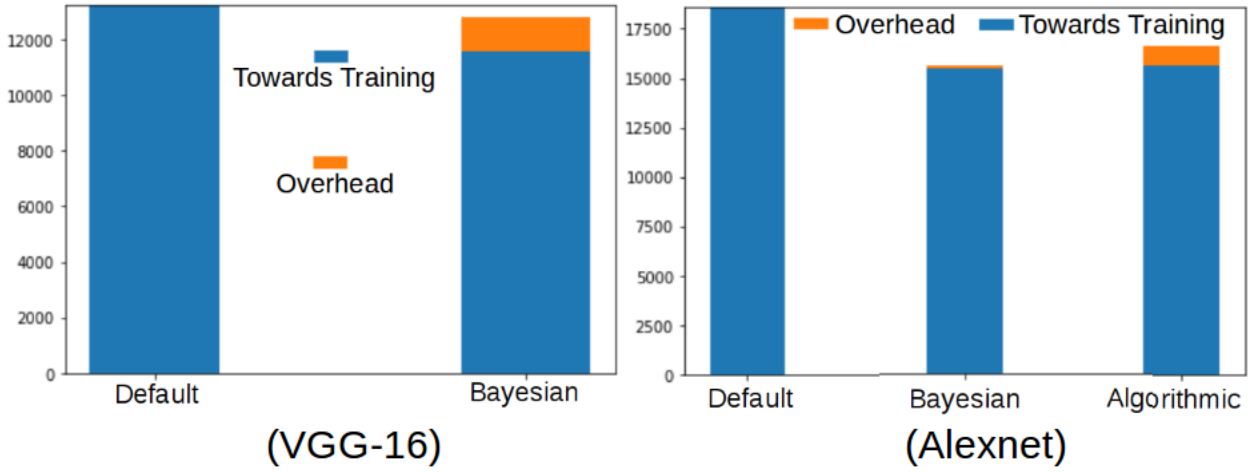


Figure 6.5: Total training time (minutes). The Bayesian Optimization approach (Adaptive HTF-MPR) improved the overall time by **3.5%** in VGG-16 and **18.7%** in Alexnet. The overhead in the Bayesian accounts 9.5% of the whole process in VGG-16 while it accounts for 1.1% in Alexnet. Note that the Algorithmic did not find a better mapping for VGG-16 as shown in Table 6.2. As for Alexnet, the overall improvement was by 12% and the overhead accounts for 5.6% using the Algorithmic approach.

6.4 Conclusion of Previous Chapters

We have presented HTF-MPR and Adaptive HTF-MPR approaches to optimizing the mapping of devices to operations in order to improve performance. The proposed frameworks use algorithmic and BO processes, respectively. A predictive model is applied in the GA to search for a mappings, which outperforms the TF default mapping overall. The predictive model is trained using algorithmic and BO approaches, respectively, resulting in mappings and makespan observations. The predictive model is constructed using GBR. Experimental results show a substantial overall speeding up for the investigated benchmarks. In addition, we have presented our analysis of the solution space using the small benchmark MNIST-Softmax. We have observed that only a small percentage5%of mappings outperform the default TF mapping, indicating that it is difficult to identify a successful search scheme for a large computational graph. The proposed search technique has been shown to be capable of finding a mapping that outperforms the default TF mapping. We have also presented the adaptive mechanism and explored how it reacts when the system experiences stress.

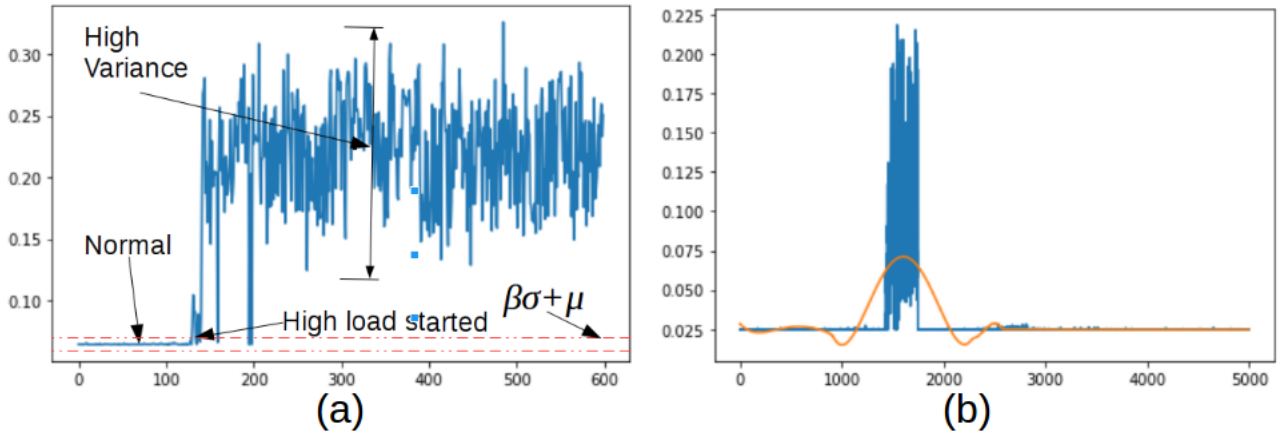


Figure 6.6: The TF default mapping on: a) **VGG-16**; b) **AlexNet**. The y-axis is the makespan and the x-axis represents the iterations number. The makespan changes when there is a high load (using Unigine's SuperPostion benchmarking tool [67]) on the GPU. The red line shows the threshold for when Adaptive HTF-MPR would be triggered if the default mapping were also the m^* mapping. $\beta = 10$ in this case. The higher the *Beta* coefficient the less sensitive to changes Adaptive HTF-MPR is. Note that different loads have been used in both instances. In addition, the load has high variance in this case.

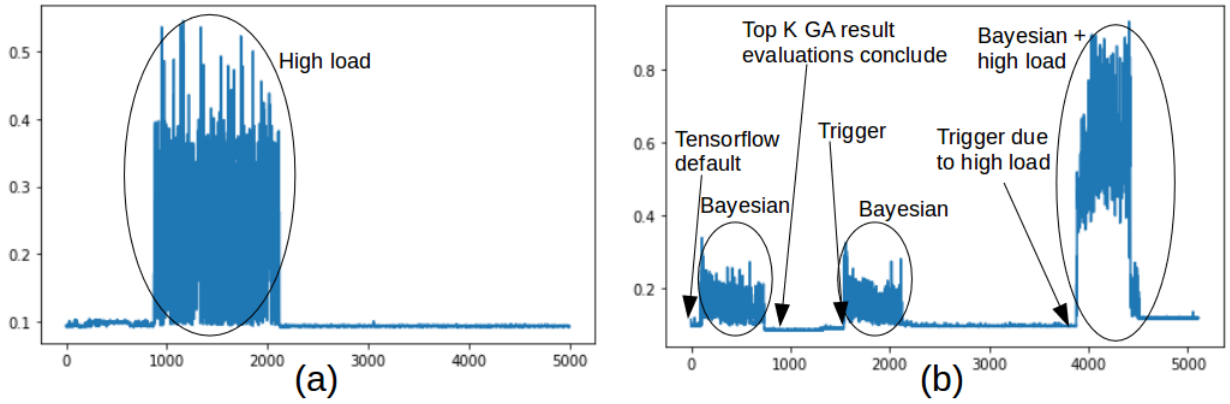


Figure 6.7: AlexNet makespan at each iteration: **a) without**; and **b) with** Adaptive HTF-MPR. Note that GA happens offline (meaning that it does not contribute to the advancement of the training step) and therefore is not shown. The top K of the resulting GA results are run on f_{NN} and therefore are shown. In this case K=100. The high load is applied for 30 minutes in both cases.

Chapter 7

Static Mapping Method

In this chapter, we tackle the mapping optimization problem using methods borrowed from static-scheduling algorithms, more specifically *heterogeneous earliest finish time* (HEFT).

7.1 List Scheduling

In static task-scheduling, and more specifically in list-based scheduling, there are two main phases [7]:

1. The ordered list of processes phase (in our case TF operations), where operations are mapped according to their priority levels.
2. The selection of resources (i.e., devices) per process, where the resource allocation is set in the order determined by item 1.

In our list-based scheduling framework, we apply the HEFT algorithm. In this static-list-based task-scheduling algorithm, two forms of information are required as input to the scheduler:

- A table of execution times that shows the execution time for each operationdevice pair.
- The dependency graph in the *directed acyclic graph* (DAG), which shows the operation dependencies (i.e., the successor of each operation).

Dependencies are easily extracted since the TF graph is already implemented in a DAG fashion. With regard to the execution times, these need to be *extracted* and *captured* for every device–operation pair.

7.2 Task Mapping

The fact that no changes may occur after the TF graph is run implies that the use of static scheduling [44] is required. List scheduling [44] is a type of static scheduling that consists of two main steps: a prioritization step and a device selection step. Many types of list-scheduling algorithms are available [7, 66, 48]. One approach that has been thoroughly tested and fits our need for static scheduling on DAGs, which are heterogeneous both in operations and in devices, is the HEFT algorithm [68]. In terms of the construction of scheduling and mapping, HEFT is a fast heuristic approach: it is greedy and works well with DAG [13].

In order to utilize HEFT, two pieces of information are required: the operation dependencies that are represented by the DAG, and the execution time of these operations on every device available. In our case, the execution time of an operation on each device is unknown a priori.

7.3 Execution Time

As discussed in the previous section, list scheduling requires the execution time of each device–operation pair. One way to obtain the execution time is by using the analytical model approach. This requires considerable information, some of which is platform-dependent (i.e., based on hardware, devices, type of device, and inner workings of the operations) [5]. Although the analytical approach gives a better prediction model [5], it requires a lot of unknown information that is not easy to obtain in a heterogeneous environment.

Another way to obtain the execution time is to build a prediction model for the device–operation pairs. In order to use the execution time prediction model, inputs must be given to the model; i.e., input to features. Features in this case could be the number of tensors that are inputs, the dimensions of the input, and the sizes of each dimension. In addition, more features can be constructed via polynomial feature extraction. Using ML approaches, the models can be constructed ahead of time and any new TF graph would fetch the appropriate set of device–operation models to obtain approximate values of execution times. Note that feature values, per TF operation type, must be extracted in this case to be fed to the model. In addition, the learning process would require the generation of many and varied data points that need to be run on TF operations in order to train each and every device–operation model. Such an approach would not guarantee variable data-point generation that would span the range of values, and thus would not guarantee the proper training set for the execution-time prediction model.

Another approach, and that which we follow, is to run each operation with its actual argument values. This obtains the execution time for the deviceoperation pair with the exact requested values, and does so for each and every deviceoperation pair. Although the cost of carrying out this extraction is repeated for every TF model, it is more accurate and hassle-free, as there is no need to figure out what the parameters are or to generate data points.

In addition, given that neural networks, in this case TF models, are run multiple times in magnitudes of at least hundreds of thousands when training the models, and unknown times when a model is run in an inference manner, the overhead of running the operation to obtain the execution time is negligible.

7.3.1 Execution Time Calculation

For every device–operation pair, a *session* run is made in order to obtain the runtime of said operation in said device (see Figure 7.1). Once the session-time matrix is completed, the execution calculation is applied to each device–operation pair t_{τ_i,d_x}. This calculation is given according to the adjacency list, where the inverse of the adjacency list would provide all required incoming operations:

$$t_{\tau_i,d_x} = s_{\tau_i,d_x} - \sum_{\tau_j \in pred(\tau_i)} s_{\tau_j,d_x} \tag{7.1}$$

An example of an adjacency list (represented in Figure 7.1 as a directed graph) follows:

operation	successor(s)
τ_c	τ_b
τ_b	τ_d
τ_a	τ_d
τ_d	...

The inverse of which is:

operation	predecessor(s)
τ_d	τ_b τ_c
τ_b	τ_c
τ_a	...
τ_c	...

This is used for the calculation of the execution time for each operation using the session time: an example for t_{τ_d} is given in Figure 7.1.

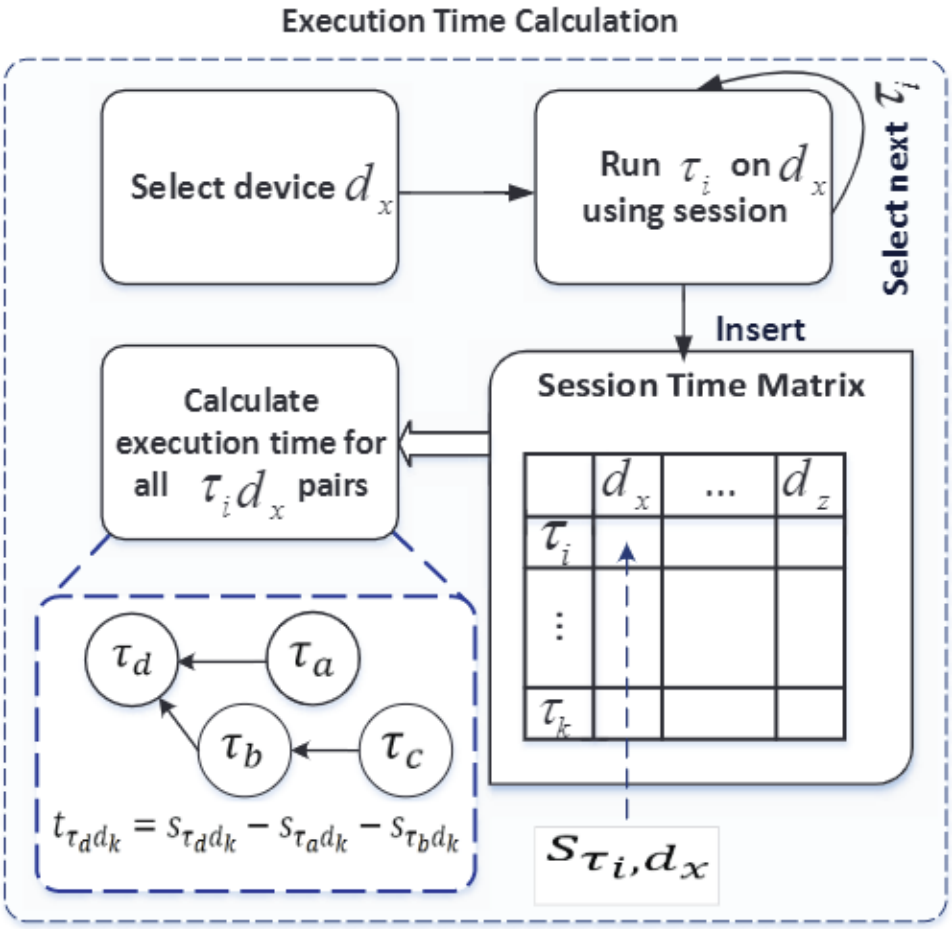


Figure 7.1: Execution-time calculation

The number of runs required to determine each deviceoperation execution time is VXD. Thus, the overhead of using TF-Mapper is directly proportional to the number of TF operations in the model and the number of devices in the system.

7.4 HEFT Scheduling for Mapping

Once the execution-time matrix is complete, it is used alongside the adjacency list (which describes the DAG) in the scheduling algorithm. HEFT [69] is chosen as our scheduling algorithm based on [13]. The computational complexity of HEFT is $O(V^2D)$ [63], while that of a brute-force approach is $O(D^V)$. HEFT has two main phases. Firstly, the upward *rank* of each operation is determined, so the operations can be scheduled according to their priority levels: the largest corresponds to the highest priority. Note that if an operation has no inputs and is considered the source of the DAG, it will have the largest rank. The rank calculation is determined as follows:

$$rank_u(\tau_i) = \overline{t_{\tau_i}} + \max_{\tau_j \in succ(\tau_i)}(rank(\tau_j)) \tag{7.2}$$

where $\overline{t_i}$ is the average execution time of the operation over all devices (i.e., the average of the row in the execution-time matrix). $succ(\tau_i)$ are all the successors of τ_i in the DAG. In this situation, the maximum rank between all successors is added to the calculation. Note that the use of communication costs is not considered when calculating the rank [63].

The aim of the scheduler is to reduce the completion time or *makespan*. In the case of neural networks, a single run of the backpropagation is considered to represent the DAGs makespan without completion; i.e., the completion occurs only when the DAG is run multiple times (rather than once). Thus, in our case, the *makespan* is a single run.

Once the rank of each operation has been determined, the second phase starts. The scheduler conducts its operation selection phase in descending order according to rank [69]. The device $d_x \in D$ that would be selected is the device that would provide the earliest finish time (EFT):

$$EFT(\tau_i, d_x) = t_{\tau_i, d_x} + EST(\tau_i, d_x) \tag{7.3}$$

where EST is the earliest start time given by:

$$EST(\tau_i, d_x) = max\{available(d_x), \max_{\tau_k \in pred(\tau_i)}(EFT(\tau_k))\} \tag{7.4}$$

$available(d_x)$ is the earliest time when d_x is available to run τ_i, while the inner max term represents the moment at which all the predecessor operations have completed their execution. Thus, the earliest finish time depends on the max of device availability or predecessor operation completion. Note that:

$$EST(\tau_{entery}, d_x) = 0 \tag{7.5}$$

This indicates that the highest-ranked operation, which is the entry operation in the DAG, has an earliest start time of 0.

7.5 Experimental Results

7.5.1 Experimental Setup

To test our framework, we use a system that consists of a multi-core CPU (Intel(R) Core(TM) i7-4770K CPU 3.50Ghz) and GPU (Nvidia GeForce GTX 770). The TF version used is 0.12 with GPU capability (using Nvidia CUDA 8.0 and cuDNN v5), and the Python version is 2.7.12. The following benchmarks are tested:

- AlexNet, a CNN used to classify images.
- MNIST Softmax classifier, a very simple image classifier.
- VGG-16, a CNN used on ImageNet.

The benchmarks are run with TF, without any forced mapping, and the total execution time of each benchmark is recorded. The same benchmarks are run through TF-Mapper to obtain the mappings, which are then run using the resulted mappings. Total execution time of each benchmark and the processing time of TF-Mapper are recorded. Also, as an extra measure for comparison, we randomly generate a mapping and apply it to the benchmark in order to compare results (and determine that the proper mapping via the scheduling algorithm is contributing to speeding up the process). Below is a summary of the benchmarks. Note that the learning process is an iterative process; thus, the DAG is run several times. The majority of operations are not handled within TF-Mapper. This is because these operations are generated by TF; thus, the user may not assign a mapping via a *tf.device*. With these hidden operations, TF handles the mapping via its default mechanism.

7.5.2 Results

Given the overhead of running TF-Mapper, an average speed-up of **1.05** is still accomplished with AlexNet, **1.35** with MNIST-Softmax classifier, and **1.11** with VGG-16. The default TF device mapper (as of version 0.12), using TensorFlow-GPU, maps the majority, or in some cases, all of the operations to the GPU.

Excluding the overhead time of the TF-Mapper, and basing our results on the mapping resulting from this, the average speed-up values are **1.16** for AlexNet, **1.85** for MNIST-Softmax classifier, and **1.16** for VGG-16.

The majority of the process time in TF-Mapper is taken up by the **execution-time calculation** stage. More specifically, a session run is required per TF operation in each device, in order to capture the session time. The time taken by the **HEFT scheduling algorithm** is insignificant compared to the **execution-time calculation**: less than 1% of the time is taken up by the scheduling stage. An increased number of devices would thus mean an

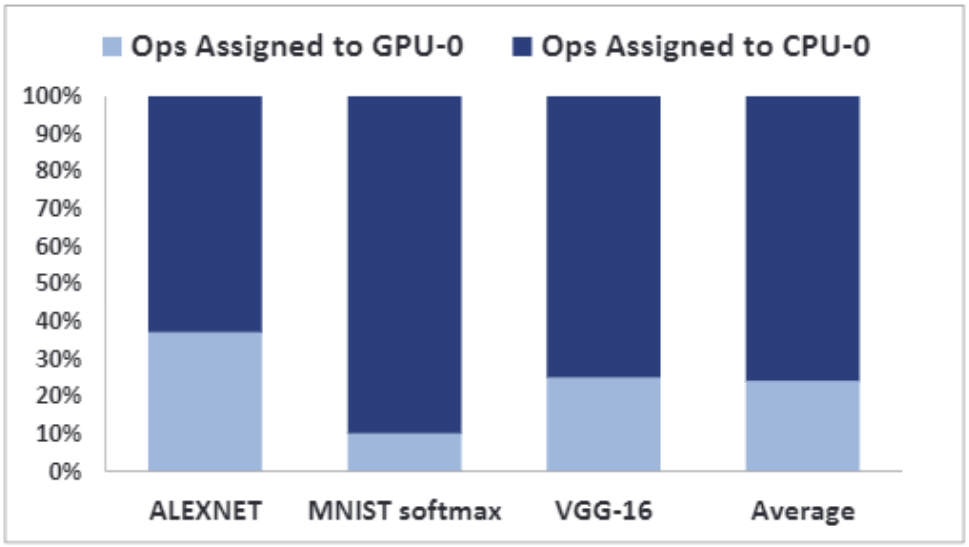


Figure 7.2: Distribution deviceoperation mapping from TF-Mapper

increase in the overhead time of TF-Mapper. Note that random mapping affects the performance negatively, indicating that proper mapping via HEFT scheduling contributes to speeding up the overall process.

With regard to the distribution of device mapping, the exclusive use of GPUs in TensorFlow-GPU default mapper is not always ideal. This is due to different systems running different types of GPU hardware (some are high-end while others are low-end). In addition, the states of the devices (i.e., CPU(s) and GPU(s)) at any given time are unknown and contribute to the overall performance. Thus, a check on the execution time per deviceoperation pair is required before (within a short period of time) running a benchmark.

According to the execution times and scheduling algorithm, TF-Mapper does not favor GPU for every operation, as can be seen by the distribution of the mapping (see Figure 7.2). With such mapping the performance improves (see Figure 7.3).

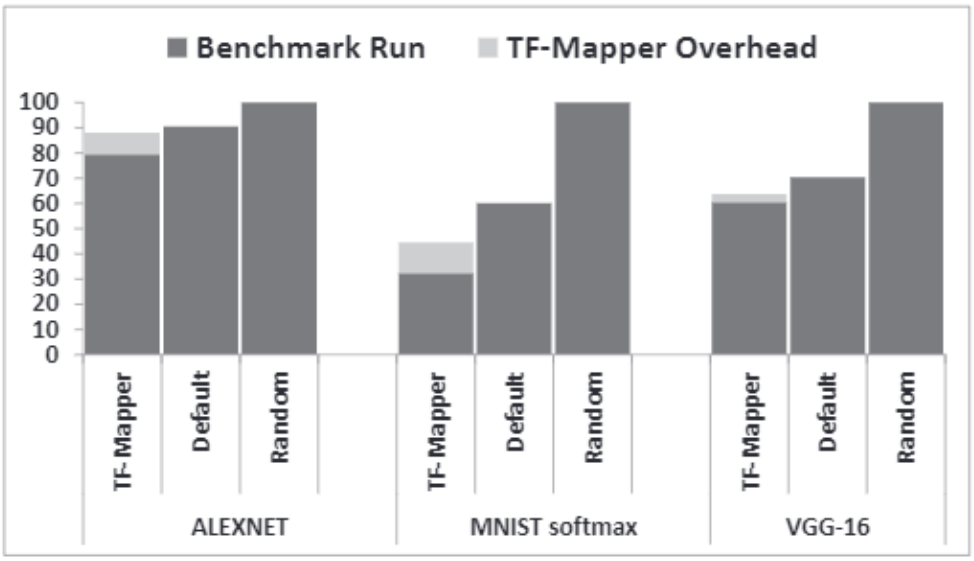


Figure 7.3: Comparison of relative execution times

7.6 Conclusions

In this paper, we have presented our TF-Mapper framework, which is used to optimize the mapping of devices to TF operations. The TF-Mapper extracts the execution time of each deviceoperation pair. Using the HEFT scheduling algorithm, it finds an optimal deviceoperation mapping, providing better performance than the one carried out by the default TF mapper. This method works when using two devices, but would fail with any more than two. A more effective approach for three or more devices would be to use HTF-MPR or Adaptive HTF-MPR.

Chapter 8

Operation Split for higher Parallelism

In this chapter, operations in the graph will be divided into sub-operations. In other words, the NN will be partitioned in order to make more use of inter-parallelism (parallelism between devices, as opposed to parallelism within a single device such as a GPU). The methodology [61] presented will therefore increase the operations but decrease the computation time required for each operation. In order to split the operations within a fully connected NN layer, it is necessary to *prune* some weights; i.e., to reduce the number of parameters. The method of removal and the choice of which parameters to remove are of importance in this case.

8.1 System Model

Our framework targets neural networks that have some or all of their *nodes* fully connected to the subsequent nodes. The *set* of starting nodes, $\boldsymbol{N_{initial}}$, is *fully* connected to the subsequent nodes $\boldsymbol{N_{final}}$; i.e., the model has *fully connected layers*. A link, which is a parameter, is a connection represented by $\boldsymbol{L_{ij}}$, where $\boldsymbol{i}$ is the starting node number and, $\boldsymbol{j}$

is the connected node number within a layer. The link's value (i.e., the parameter's weight) is represented by $\boldsymbol{w_{i,j}}$. $\boldsymbol{L_{i,j} = 0}$ if the link is pruned, and if not, $\boldsymbol{L_{i,j} = 1}$. Note that $\boldsymbol{w_{i,j}}$ may contain any value. The set of weights, $\boldsymbol{W_i}$, consists of links, $\boldsymbol{L_i}$, that connect between the set of nodes $\boldsymbol{N_i}$ and $\boldsymbol{N_j}$. Figure 8.1a shows an example of a fully connected layer of size $\boldsymbol{6 \times 8}$. Figure 8.1b shows the matrix representation of the fully connected layer. Figure 8.1c indicates the weight matrix of the fully connected layer. The *connectedness number*, $\boldsymbol{C}$, is simply;

$$\boldsymbol{C} = \sum_{\boldsymbol{i=1}}^{|\boldsymbol{N_{initial}}|} \sum_{\boldsymbol{j=1}}^{|\boldsymbol{N_{final}}|} \boldsymbol{L_{i,j}} \tag{8.1}$$

A fully connected layer is annotated as $\boldsymbol{C_{full}}$, thus;

$$\boldsymbol{C_{full}} = |\boldsymbol{N_{initial}}| \times |\boldsymbol{N_{final}}| \tag{8.2}$$

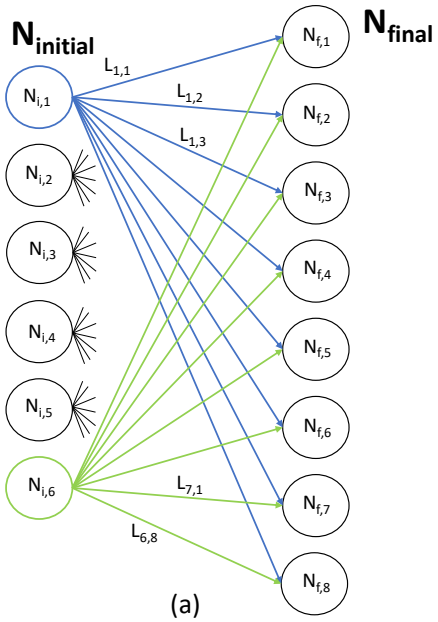
Therefore, the *connectedness ratio*, $\boldsymbol{R}$, is:

$$\boldsymbol{R} = \frac{\boldsymbol{C}}{\boldsymbol{C_{full}}} \tag{8.3}$$

Figure 8.2 shows an example of a two-partition pruning of the fully connected layer from Figure 8.1. Figure 8.3 visually illustrates the partitions of Fig 8.2 and the reduction of the number of weights due to that partitioning. Given that there are $\boldsymbol{|P|}$ partitions, where a $\boldsymbol{P_x} \in \boldsymbol{P}$, then any given $\boldsymbol{N_{initial,j}} \in \boldsymbol{P_x}$ will not be in any other partition. The same applies to nodes in $\boldsymbol{N_{final,i}}$. More formally:

$$\boldsymbol{\{P_i, P_j \in P | i \neq j, P_i \cap P_j = \emptyset\}} \tag{8.4}$$

Equation 8.4 is the constraint of the groupings of nodes in $\boldsymbol{N_{initial}}$ and $\boldsymbol{N_{final}}$. That is, once a particular node is in a particular partition, it cannot be a member of another partition.



$L_{1,1}$	$L_{1,2}$	$L_{1,3}$	$L_{1,4}$	$L_{1,5}$	$L_{1,6}$	$L_{1,7}$	$L_{1,8}$
$L_{2,1}$	$L_{2,2}$	$L_{2,3}$	$L_{2,4}$	$L_{2,5}$	$L_{2,6}$	$L_{2,7}$	$L_{2,8}$
$L_{3,1}$	$L_{3,2}$	$L_{3,3}$	$L_{3,4}$	$L_{3,5}$	$L_{3,6}$	$L_{3,7}$	$L_{3,8}$
$L_{4,1}$	$L_{4,2}$	$L_{4,3}$	$L_{4,4}$	$L_{4,5}$	$L_{4,6}$	$L_{4,7}$	$L_{4,8}$
$L_{5,1}$	$L_{5,2}$	$L_{5,3}$	$L_{5,4}$	$L_{5,5}$	$L_{5,6}$	$L_{5,7}$	$L_{5,8}$
$L_{6,1}$	$L_{6,2}$	$L_{6,3}$	$L_{6,4}$	$L_{6,5}$	$L_{6,6}$	$L_{6,7}$	$L_{6,8}$

(b)

$w_{1,1}$	$w_{1,2}$	$w_{1,3}$	$w_{1,4}$	$w_{1,5}$	$w_{1,6}$	$w_{1,7}$	$w_{1,8}$
$w_{2,1}$	$w_{2,2}$	$w_{2,3}$	$w_{2,4}$	$w_{2,5}$	$w_{2,6}$	$w_{2,7}$	$w_{2,8}$
$w_{3,1}$	$w_{3,2}$	$w_{3,3}$	$w_{3,4}$	$w_{3,5}$	$w_{3,6}$	$w_{3,7}$	$w_{3,8}$
$w_{4,1}$	$w_{4,2}$	$w_{4,3}$	$w_{4,4}$	$w_{4,5}$	$w_{4,6}$	$w_{4,7}$	$w_{4,8}$
$w_{5,1}$	$w_{5,2}$	$w_{5,3}$	$w_{5,4}$	$w_{5,5}$	$w_{5,6}$	$w_{5,7}$	$w_{5,8}$
$w_{6,1}$	$w_{6,2}$	$w_{6,3}$	$w_{6,4}$	$w_{6,5}$	$w_{6,6}$	$w_{6,7}$	$w_{6,8}$

(c)

Figure 8.1: a) Example of a model representation of a fully connected layer. b) shows the connection's representation in matrix form. Note that in the above case, $\boldsymbol{C = C_{full} = 6 \times 8 = 48}$ and $\boldsymbol{R = 1}$. c) represents the values of the weights (values of the links) represented in matrix form.

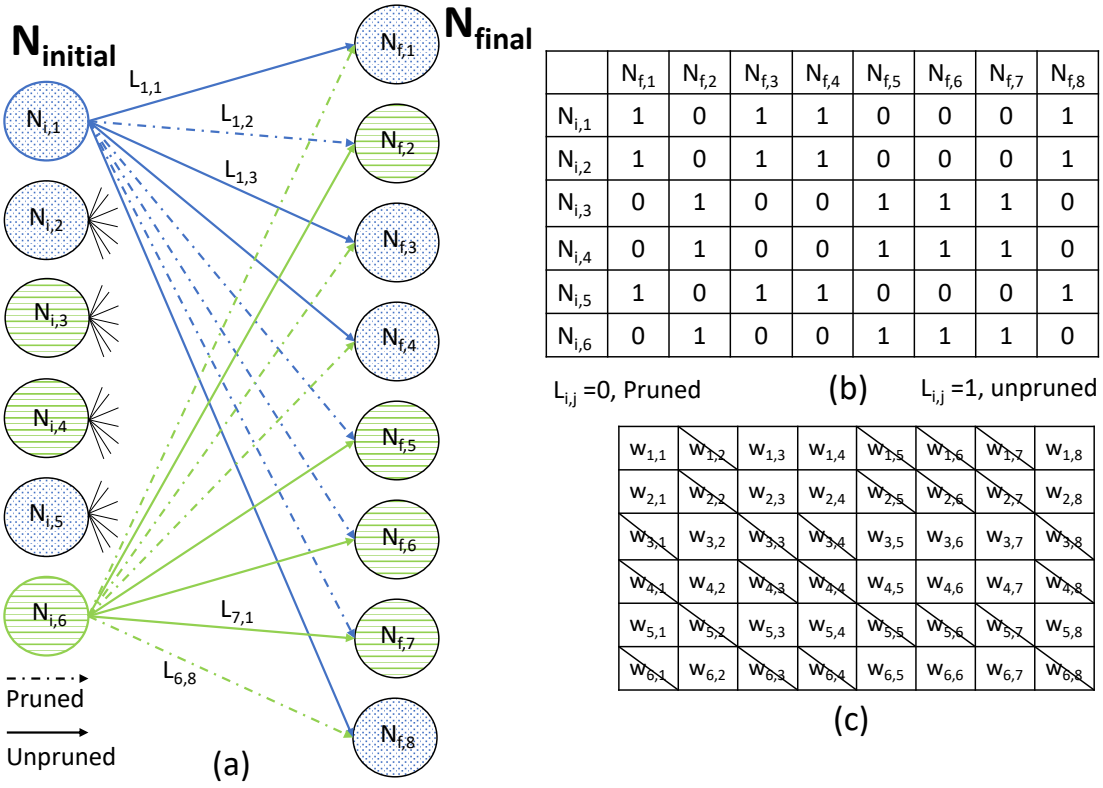
Another way of stating this is:

$$\boldsymbol{N_i \in P_n} \text{ Then } \boldsymbol{N_i \notin P_m, \forall m \neq n} \tag{8.5}$$

Note that there is an upper, $\left\lceil \frac{|\boldsymbol{N_{initial}}|}{|\boldsymbol{P}|} \right\rceil$, and lower, $\left\lfloor \frac{|\boldsymbol{N_{initial}}|}{|\boldsymbol{P}|} \right\rfloor$, bound to the number of $\boldsymbol{N_{initial,i}}$ nodes that are members of a partition $\boldsymbol{P_n}$. The same is true for $\boldsymbol{N_{final,i}}$ nodes. In addition, the number of partitions that contain the upper limit is $|\boldsymbol{N_{initial}}| \mod |\boldsymbol{P}|$, while the number that contain the lower limit is $|\boldsymbol{P}| - (|\boldsymbol{N_{initial}}| \mod |\boldsymbol{P}|)$. As an example, if $|\boldsymbol{N_{initial}}| = \boldsymbol{22}$ and $|\boldsymbol{P}| = \boldsymbol{5}$ (i.e., number of partitions), then an example of partition sizes for $\boldsymbol{N_{initial}}$, ignoring $\boldsymbol{N_{final}}$, would be

$$(|\boldsymbol{P_1}|, |\boldsymbol{P_2}|, |\boldsymbol{P_3}|, |\boldsymbol{P_4}|, |\boldsymbol{P_5}|) = (\boldsymbol{4, 5, 4, 4, 5})$$

. Therefore, the example suggests that there are three partitions of size four and two partitions of size five. This bound description also applies to $\boldsymbol{N_{final}}$.



	$N_{f,1}$	$N_{f,2}$	$N_{f,3}$	$N_{f,4}$	$N_{f,5}$	$N_{f,6}$	$N_{f,7}$	$N_{f,8}$
$N_{i,1}$	1	0	1	1	0	0	0	1
$N_{i,2}$	1	0	1	1	0	0	0	1
$N_{i,3}$	0	1	0	0	1	1	1	0
$N_{i,4}$	0	1	0	0	1	1	1	0
$N_{i,5}$	1	0	1	1	0	0	0	1
$N_{i,6}$	0	1	0	0	1	1	1	0

Figure 8.2: a) indicates what links are to be pruned from the fully connected layer; b) shows the connection's representation, with 0 representing the absence of a link. Note that in the above case, $\boldsymbol{C = C_{full} = 12}$ and $\boldsymbol{R = 0.5}$.

8.2 Partition Pruning Overview

There are two types of objectives of *partition pruning*: pruning with the objective of having balanced partitions, and pruning with the objective of having the least absolute weight loss. The second objective guarantees a smaller loss of accuracy, while the first allows for maximum parallelism. Note that the number of parameters pruned is directly related to the number

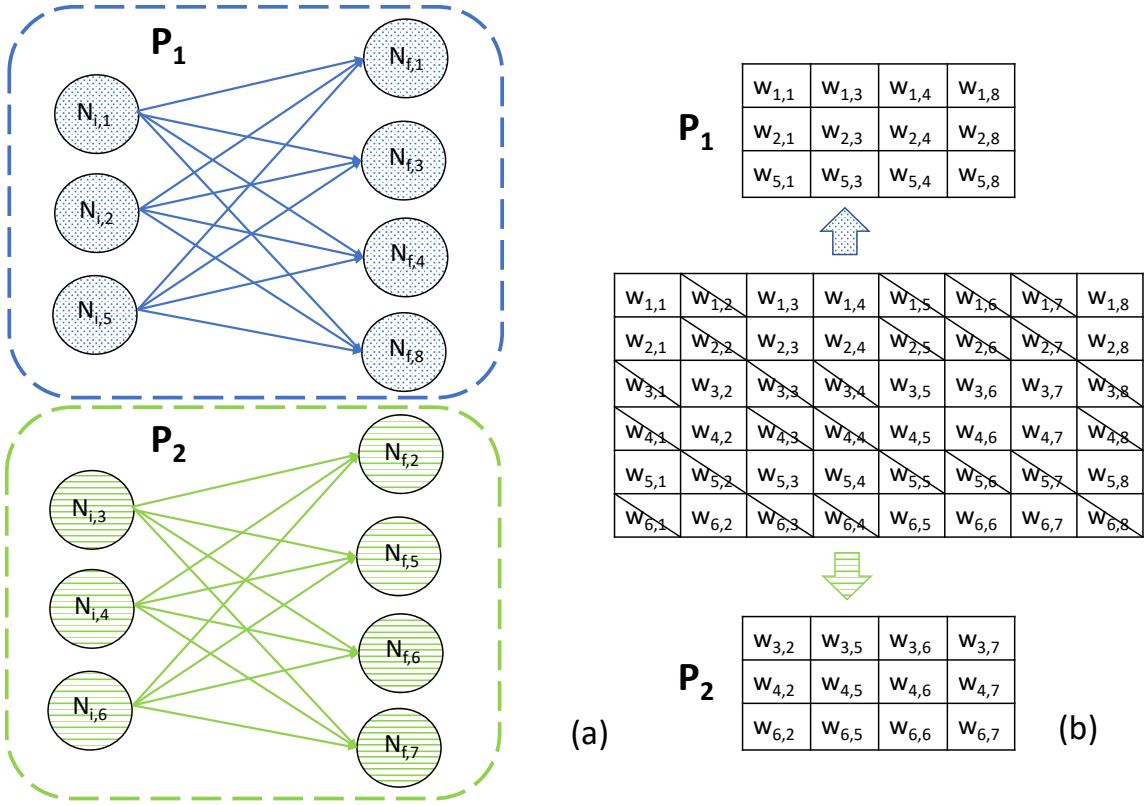


Figure 8.3: a) the resulting partitions show full independence; b) shows the reduction of parameters resulting from the two-partition targeted pruning.

of partitions desired. The connectedness ratio, in relation to the number of partitions, is $\boldsymbol{R}_{|\boldsymbol{P}|} = \frac{\mathbf{1}}{|\boldsymbol{P}|}$. Thus, for a given $|\boldsymbol{P}|$, partition pruning will find the following:

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & |\boldsymbol{C}_{\boldsymbol{full}} \sum |\boldsymbol{w}_{\boldsymbol{i},\boldsymbol{j}}| - \sum \boldsymbol{x}_{\boldsymbol{i},\boldsymbol{j}} |\boldsymbol{w}_{\boldsymbol{i},\boldsymbol{j}}|| \\ \text{subject to} \quad & \boldsymbol{x}_{\boldsymbol{i},\boldsymbol{j}} = \mathbf{0} \text{ or } \mathbf{1} \\ & \sum \boldsymbol{x}_{\boldsymbol{i},\boldsymbol{j}} = \boldsymbol{R}_{|\boldsymbol{P}|} \boldsymbol{C}_{\boldsymbol{full}} \\ & \{\boldsymbol{P}_{\boldsymbol{m}}, \boldsymbol{P}_{\boldsymbol{n}} \in \boldsymbol{P} | \boldsymbol{n} \neq \boldsymbol{m}, \boldsymbol{P}_{\boldsymbol{m}} \cap \boldsymbol{P}_{\boldsymbol{n}} = \emptyset\} \end{aligned}$$

From the objective function, we determine which $\mathbf{1} - \boldsymbol{R}_{|\boldsymbol{P}|} \boldsymbol{C}_{\boldsymbol{full}}$ parameters are pruned for a particular fully connected layer while minimizing the cumulative weight loss.

8.3 Input/Output

The input to the partition pruning algorithm is a matrix representation, $\boldsymbol{W_{fc,i}}$, of the targeted fully connected layer, $\boldsymbol{i}$. This is exemplified in Figure 8.1c. Note that the fully connected layer is assumed and asserted to be trained; that is, the parameters have the correct values for the targeted neural network's base accuracy. In a fully connected layer, every element of the matrix $\boldsymbol{L_{fc,i}}$ is 1 (see Equation 8.2). After partition pruning, the output will be $\boldsymbol{L_{part,i}}$ and the sum of all its elements would be $\boldsymbol{RC_{full}}$. This is exemplified in Figure 8.2b.

8.4 Methodology

This section presents the methodology of selecting the links to prune, taking into consideration the partitioning. The example of $|\boldsymbol{N_{initial}}| = \mathbf{7}$, $|\boldsymbol{N_{final}}| = \mathbf{10}$, and $|\boldsymbol{P}| = \mathbf{3}$, will be used to describe the process. Figure **??** shows an overview of the methodology and where partition pruning resides.

8.4.1 Start: Selection of $N_{initial,i}$, and $N_{final,j_1,j_2..}$:

In the first stage, a row in the matrix is *randomly* selected; that is, a random $\boldsymbol{N_{initial,i}}$ is selected for processing. Note that currently $|\boldsymbol{P_n}| = \mathbf{0}$ for all $\boldsymbol{n}$, because no pair of nodes has joined a partition. After choosing an $\boldsymbol{N_{initial,i}}$, a set of $\boldsymbol{N_{final}}$ nodes is chosen, and in this case, the set size is $\left\lceil \frac{|N_{final}|}{|P|} \right\rceil$. The node $\boldsymbol{N_{initial,i}}$, and the nodes $\boldsymbol{N_{fianl,j_1,j_2..}}$, are chosen to be part of the first partition, $\boldsymbol{P_1}$. Those selected will have their $\boldsymbol{L_{i,j}} = \mathbf{1}$, while those not selected will have their $\boldsymbol{L_{i,j'}} = \mathbf{0}$. Note that the links selected have the *highest magnitudes* (refer to Figure 8.4a as an example). Figure 8.4b illustrates an example of the changes in

values and a pictorial representation of the first partition.

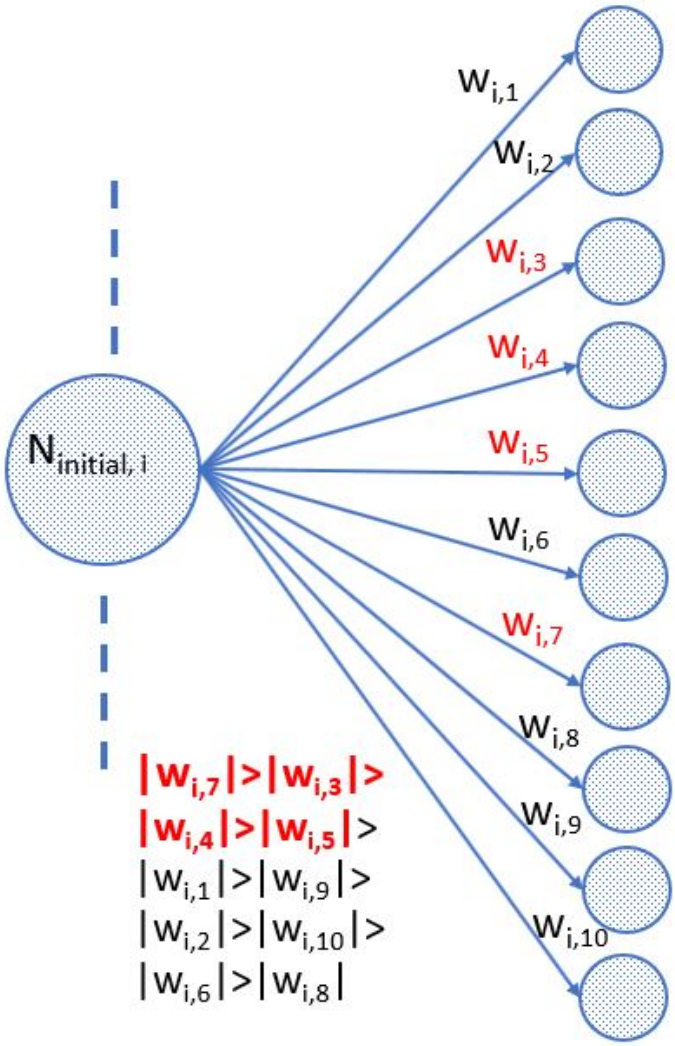


Figure 8.4: Random selection of $\boldsymbol{N_{initial,i}}$, where $\boldsymbol{i = 4}$ in this example. The top four weights, in terms of magnitude, are $\boldsymbol{w_{i,7}, w_{i,3}, w_{i,4}}$, and $\boldsymbol{w_{i,5}}$ in descending order. Note that its top *four* because of the upper bound, $\boldsymbol{\lceil |N_{final}|/|P| \rceil = \lceil 10/3 \rceil = 4}$ b) $\boldsymbol{P_1}$, after partitioning, contains four nodes (the limit) from $\boldsymbol{N_{final}}$, and one node from $\boldsymbol{N_{initial}}$. The $\boldsymbol{L}$ matrix is updated for row i=4.

8.4.2 Non-Start: Selection:

Moving forward, another $\boldsymbol{N_{initial,i}}$ node is selected at random. The highest non-partition members, $\boldsymbol{w_{i,j}}$s, are sorted from the highest to the lowest magnitude, as was carried out previously. The sum of the highest upper bound (or a lower bound if all upper bound partitions are fulfilled) is compared with the sum of the magnitude of partition-member weights/links that still have capacity (as per the upper and lower bounds of the number of nodes of type $\boldsymbol{N_{initial}}$).

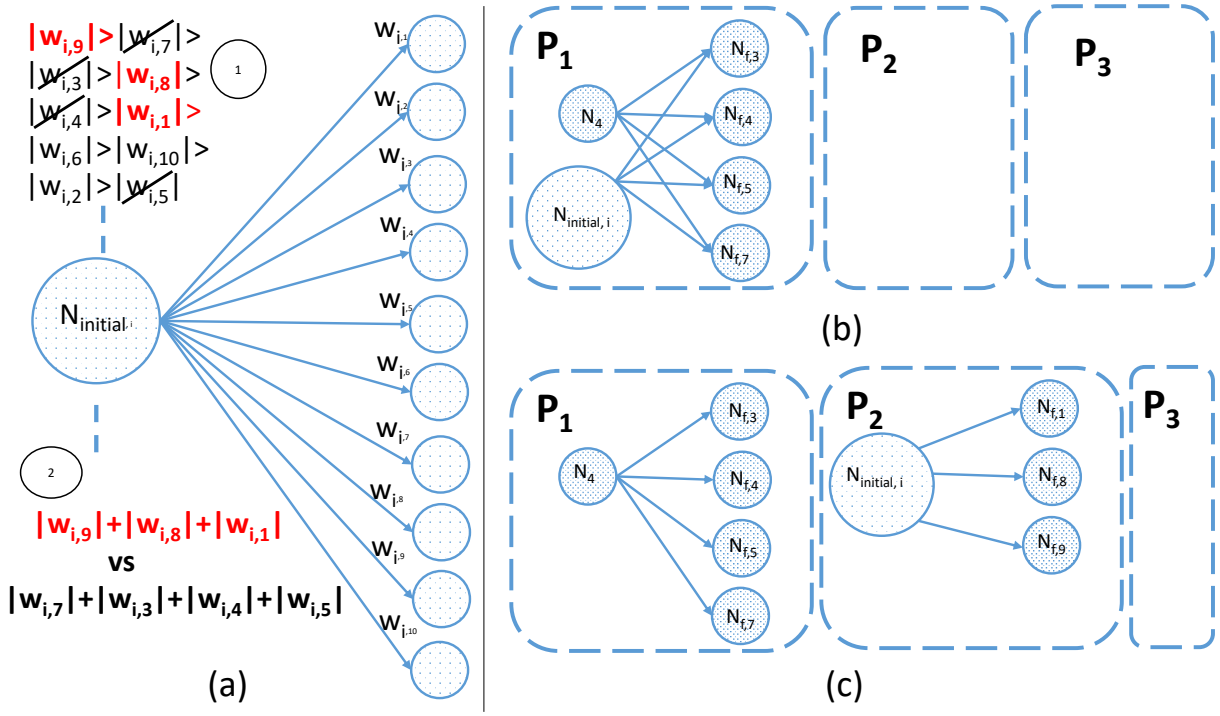


Figure 8.5: Second random selection of $\boldsymbol{N_{initial,i}}$ (where $\boldsymbol{i \neq 4}$). The top three weights (1) in terms of magnitude that are non-partition members are $\boldsymbol{w_{i,9}, w_{i,8}, w_{i,1}}$, in descending order. Note that its top *three* due to the capacity for $\boldsymbol{N_{final}}$ node type is $\boldsymbol{(P_1, P_2, P_3) = (4, 3, 3)}$ b) shows the situation in case of $\boldsymbol{|w_{i,7}| + |w_{i,3}| + |w_{i,4}| + |w_{i,5}| > |w_{i,9}| + |w_{i,8}| + |w_{i,1}|}$. c) is the case scenario.

8.4.3 End and Try Again:

This process is repeated until every partition $\boldsymbol{P_m}$ is at capacity in terms of $\boldsymbol{N_{initial}}$ nodes and $\boldsymbol{N_{final}}$ nodes. Note that the partitioning is dependent on which row, i.e $\boldsymbol{N_{initial,i}}$, is selected at each iteration. Once the process is completed, the weight loss is recorded.

8.5 Conclusion

One of the main reasons for operation splitting via partition pruning is to increase the parallelism of the execution. One potential future direction for this work would be to include sub-operations that are mapped, thus increasing the mapping search space. An even further development would be to consider the different iterations of splits that could occur and within those different mapping search spaces. With pruning, the number of parameters would decrease drastically and the accuracy would thus change. The approach taken had minimal affect on the accuracy which was the intention by removing those parameters that had the smallest affect due to their small magnitude.

Chapter 9

Conclusion and Future Work

Machine-learning (ML) applications are ever-increasing in todays world, from the health care industry to space exploration and consumer goods. Moreover, the applications are getting more and more complex. With increasing complexity, high-parameter and interconnected models are required. A notable structure is the deep neural network (DNN) model. Indeed, these parameters need tuning, which requires a large amount of data, which itself requires training. This training is referred to as deep learning (DL). DL is computationally intensive and requires a lot of time to complete.

This dissertation tackles the time aspect, specifically how it is possible to reduce execution time via parallelism by virtue of resource management. This can be achieved via the optimal or sub-optimal allocation of processors to process, referred to as mapping. The number of possible mappings is large, which is problematic if an optimal mapping is to be found manually.

To understand the viability of non-manual mapping, we investigated a small model by testing every possible mapping. We observed that the absolute best mapping is non-intuitive, indicating that the function of time to mapping is not straight forward.

We, therefore, devised an automated framework to search for a sub-optimal mapping. This framework uses an array of approaches and tools. We used Bayesian optimization (BO) to initiate the search space while simultaneously running the training of the DNN. While training the DNN, observations of the execution times were made in order to direct the BO with respect to the best performing mappings. We compared different initial search methods, which are also referred to as initial mapping generations, finding that the BO outperformed the other approaches in the initial search task.

Given that the search space is large, running the DNN to gather observations would be likely to hinder the speed of the search. To remedy this, we trained a function (model) using the gradient-boosting regressor (GBR) ML algorithm to predict the observed makespans, depending on the particular mapping. It takes the predictive model orders of magnitude less time to calculate the makespan than it takes to observe makespan via actual DNN run, thus expanding the horizons of the search function.

In order to benefit from the speed of the predictive function, a lightweight optimization search algorithm needs to be used. BO, although effective, is best utilized with heavy functions (such as the actual run of the DNN), which defeats the purpose of using it with light functions (such as the predictive model). Thus, a genetic algorithm (GA) was used, which is much faster when it comes to generating new mappings from parent mappings (as was explained in Chapter 6).

Finally, once the top predicted observations are found, the corresponding mappings are used on the DNN in order to find the actual best, rather than the predicted best, since actual observations and predicted observations may be different.

An adaptive mechanism is added to the framework in order to continuously monitor the performance of the system in terms of execution time. In the case of any changes in the execution time, a new search for a better mapping occurs.

This framework has shown that the performance is better than that of the homogeneous default mapping of the system. In addition, it outperformed intuitive-based mapping.

We also introduced a non-search-based mapping methodology in Chapter 7. Here, we took the lead from scheduling algorithms, where device-operation times were observed, and communication costs ignored. This method would not work well with many devices or large computational graphs. Thus, its use is limited and indicates that the execution time of the computational graph is a black box.

As a precursor to future work, we developed a method to partition a DNN via pruning the links between layers; i.e., we reduced the number of parameters in a partition-centric way and removed those that affected the accuracy of the DNN the least.

9.1 Future Work

There are different paths that may be taken to expand the work of this dissertation, all of which target improving the performance of the execution time of the DNN during training. Therefore, the different approaches tackle the same problem from different angles, thereby improving the accuracy of the predictive model and its recovery; i.e., the adaptive mechanism.

9.1.1 Deep Probabilistic Modeling

Given that when the system is unstable, or less so when even stable, the makespan for any particular mapping will not give the same value at each run. This is due to unaccounted elements in the system. Rather than having a predictive model that provides a strict value, the makespan predictive model provides a range of values or a probability of the values. Note that the probabilistic [52] approach is an inherent part of machine learning.

9.1.2 Graph Convolutional Networks

When the mapping is represented, it is done so in flat terms. This means that the connections and their relations are lost. To remedy this issue, we may borrow ideas from convolutional neural networks (CNNs). When CNNs take an image as an input, the proximity of the pixels are taken into consideration, demonstrating that an image is nothing but a rigid rectangular graph comprised of pixels that link each other. Rather than flatten the computational graph, we keep its structure intact. Indeed, graph convolutional networks have proven their effectiveness in many graph applications [32].

9.1.3 Principle Component Analysis

By using principle component analysis (PCA) [37] as a statistical method to reduce the dimensionsand thus the size of feature setused in creating the makespan predictive model, we can essentially improve the accuracy of the predictor. Moreover, we can extend the usage of PCA on the GA. Indeed, Other methods of dimensional reduction and feature selection could be explored [1] beyond PCA.

9.1.4 Reinforcement Learning

In the adaptivity part of adaptive HTF-MPR, where the system is monitored for any performance changes, reinforcement learning (RL) [38] can be used. When a trigger occurs, HTF-MPR performs an incremental search using RL methods, rather than starting the search over from scratch, which saves time. In other words, rather than run the HTF-MPR all over again from the start, RL continues the search and utilizes past data.

9.1.5 Split Node

As discussed in the conclusion of Chapter 8, research could continue with respect to how to further utilize concurrency, as well as with respect to making parallelisms available by partitioningpartitioning being an operation wherein a node is split into smaller operations that deal with smaller tensors. The draw back here is that the accuracy of the neural network might be affected. Luckily, however, most networks (such as VGG-16 and AlexNet) are over-parameterized, leaving potential for reduction of paramters in this regard.

Bibliography

[1]

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, 2016. USENIX Association.

[3] M. Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[4] A. Albaqsami, M. S. Hosseini, and N. Bagherzadeh. Htf-mpr: A heterogeneous tensorflow mapper targeting performance using genetic algorithms and gradient boosting regressors. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 331–336, March 2018.

[5] M. Amarís, R. Y. de Camargo, M. Dyab, A. Goldman, and D. Trystram. A comparison of gpu execution time prediction using machine learning and analytical modeling. In *Network Computing and Applications (NCA), 2016 IEEE 15th International Symposium on*, pages 326–333. IEEE, 2016.

[6] S. Aminikhanghahi and D. J. Cook. A survey of methods for time series change point detection. 51(2):339–367, 2016.

[7] H. Arabnejad and J. G. Barbosa. List scheduling algorithm for heterogeneous systems by an optimistic cost table. *IEEE Transactions on Parallel and Distributed Systems*, 25(3):682–694, 2014.

[8] T. T. Authors. mnist classifier using softmax in tensorflow. `https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/tutorials/mnist/`, 2017.

[9] J. Bergstra, D. Yamins, and D. D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages I–115–I–123, 2013.

[10] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 2546–2554. Curran Associates, Inc., 2011.

[11] L. Bottou. Large-scale machine learning with stochastic gradient descent. In Y. Lechevallier and G. Saporta, editors, *Proceedings of COMPSTAT'2010*, pages 177–186, Heidelberg, 2010. Physica-Verlag HD.

[12] G. Bradski. Learning-based computer vision with intels open source computer vision library. *Intel Technology Journal*, 9, 05 2005.

[13] L.-C. Canon, E. Jeannot, R. Sakellariou, and W. Zheng. Comparative evaluation of the robustness of dag scheduling heuristics. In *Grid Computing*, pages 73–84. Springer, 2008.

[14] B. Catanzaro. Deep learning with cots hpc systems. 2013.

[15] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, R. Anil, Z. Haque, L. Hong, V. Jain, X. Liu, and H. Shah. Wide & deep learning for recommender systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, DLRS 2016, pages 7–10, New York, NY, USA, 2016. ACM.

[16] 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)*, pages 571–582, Broomfield, CO, 2014. USENIX Association.

[17] F. Chollet. keras. `https://github.com/charlespwd/project-title`, 2015.

[18] C. Cortes, M. Mohri, and A. Rostamizadeh. L2 regularization for learning kernels. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, UAI '09, pages 109–116, Arlington, Virginia, United States, 2009. AUAI Press.

[19] M. Courbariaux, Y. Bengio, and J.-P. David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, pages 3123–3131, Cambridge, MA, USA, 2015. MIT Press.

[20] M. Denil, B. Shakibi, L. Dinh, M. A. Ranzato, and N. de Freitas. Predicting parameters in deep learning. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 2148–2156. Curran Associates, Inc., 2013.

[21] P. A. Diaz-Gomez and D. F. Hougen. Initial population for genetic algorithms: A metric approach. In *GEM*, pages 43–49, 2007.

[22] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *19th Design Automation Conference*, pages 175–181, June 1982.

[23] J. H. Friedman. Stochastic gradient boosting. *Computational Statistics Data Analysis*, 38(4):367 – 378, 2002. Nonlinear Methods and Data Mining.

[24] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis. Tetris: Scalable and efficient neural network acceleration with 3d memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 751–764, New York, NY, USA, 2017. ACM.

[25] E. C. Garrido-Merchn and D. Hernndez-Lobato. Dealing with categorical and integer-valued variables in bayesian optimization with gaussian processes, 05 2018.

[26] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[27] L. Hagen and A. B. Kahng. New spectral methods for ratio cut partitioning and clustering. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(9):1074–1085, Sep. 1992.

[28] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[29] B. Hassibi, D. G. Stork, and G. J. Wolff. Optimal brain surgeon and general network pruning. In *IEEE International Conference on Neural Networks*, pages 293–299 vol.1, March 1993.

[30] Y. He, X. Zhang, and J. Sun. Channel pruning for accelerating very deep neural networks. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 1398–1406, Oct 2017.

[31] J. B. Heaton, N. G. Polson, and J. H. Witte. Deep learning in finance, 2016.

[32] M. Henaff, J. Bruna, and Y. LeCun. Deep convolutional networks on graph-structured data. *CoRR*, abs/1506.05163, 2015.

[33] J. H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, Cambridge, MA, USA, 1992.

[34] Z. Jia, S. Lin, C. R. Qi, and A. Aiken. Exploring hidden dimensions in accelerating convolutional neural networks. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 2274–2283, Stockholmsmssan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[35] Z. Jia, M. Zaharia, and A. Aiken. Beyond data and model parallelism for deep neural networks. *CoRR*, abs/1807.05358, 2018.

[36] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation; part i, graph partitioning. *Operations Research*, 37(6):865–892, 1989.

[37] I. Jolliffe. *Principal Component Analysis*, pages 1094–1096. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[38] L. P. KAELBLING. Reinforcement learning : A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[39] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307.

[40] D. E. Knuth. Postscript about np-hard problems. *SIGACT News*, 6(2):15–16, Apr. 1974.

[41] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, pages 1137–1143, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[42] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[43] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[44] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999.

[45] Y. Le Cun, J. S. Denker, and S. A. Solla. Optimal brain damage. In *Proceedings of the 2Nd International Conference on Neural Information Processing Systems*, NIPS'89, pages 598–605, Cambridge, MA, USA, 1989. MIT Press.

[46] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521:436–44, 05 2015.

[47] Y. LeCun and C. Cortes. MNIST handwritten digit database. 2010.

[48] G. Liu, K. Poh, and M. Xie. Iterative list scheduling for heterogeneous computing. *Journal of Parallel and Distributed Computing*, 65(5):654 – 665, 2005.

[49] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pages 2430–2439. JMLR.org, 2017.

[50] M. Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[51] D. Miyashita, E. H. Lee, and B. Murmann. Convolutional neural networks using logarithmic data representation. *CoRR*, abs/1603.01025, 2016.

[52] K. P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[53] H. Niu, I. Keivanloo, and Y. Zou. Learning to rank code examples for code search engines. *Empirical Software Engineering*, 22(1):259–291, Feb 2017.

[54] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[55] M. Pelikan, D. E. Goldberg, and E. Cantú-Paz. Boa: The bayesian optimization algorithm. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 1*, GECCO'99, pages 525–532, 1999.

[56] M. Pinedo and K. Hadavi. Scheduling: Theory, algorithms and systems development. In *Operations Research Proceedings 1991*, pages 35–42. Springer, 1992.

[57] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Neurocomputing: Foundations of research. chapter Learning Representations by Back-propagating Errors, pages 696–699. MIT Press, Cambridge, MA, USA, 1988.

[58] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, and et. al. Imagenet large scale visual recognition challenge. *arXiv preprint arXiv:1409.0575*, 2014.

[59] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85117, Jan 2015.

[60] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[61] S. Shahhosseini, A. Albaqsami, M. Jasemi, S. Hessabi, and N. Bagherzadeh. Partition pruning: Parallelization-aware pruning for deep neural networks. *CoRR*, abs/1901.11391, 2019.

[62] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, Jan 2016.

[63] K. R. Shetti, S. A. Fahmy, and T. Bretschneider. Optimization of the heft algorithm for a cpu-gpu environment. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2013 International Conference on*, pages 212–218. IEEE, 2013.

[64] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[65] G. Synnaeve, N. Nardelli, A. Auvolat, S. Chintala, T. Lacroix, Z. Lin, F. Richoux, and N. Usunier. Torchcraft: a library for machine learning research on real-time strategy games. *CoRR*, abs/1611.00625, 2016.

[66] X. Tang, K. Li, G. Liao, and R. Li. List scheduling with duplication for heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 70(4):323 – 329, 2010.

[67] U. S. B. tool. SuperPosition Software. `https://benchmark.unigine.com/superposition/`, 2017.

[68] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Task scheduling algorithms for heterogeneous processors. In *Proceedings of the Eighth Heterogeneous Computing Workshop*, HCW '99, pages 3–, Washington, DC, USA, 1999. IEEE Computer Society.

[69] H. Topcuoglu, S. Hariri, and M.-y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, 13(3):260–274, 2002.

[70] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science (Vol. A): Algorithms and Complexity*. MIT Press, Cambridge, MA, USA, 1990.

[71] G. Venkatesh, E. Nurvitadhi, and D. Marr. Accelerating deep convolutional networks using low-precision and sparsity. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2861–2865, March 2017.

[72] M. Vogt and H. Dette. Detecting gradual changes in locally stationary processes. *Ann. Statist.*, 43(2):713–740, 04 2015.

[73] M. Wang, C.-c. Huang, and J. Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 26:1–26:17, New York, NY, USA, 2019. ACM.

[74] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer. Imagenet training in minutes. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, pages 1:1–1:10, New York, NY, USA, 2018. ACM.

[75] C. Zhu, S. Han, H. Mao, and W. J. Dally. Trained ternary quantization. *CoRR*, abs/1612.01064, 2016.