Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Mathematics

Neural Network Optimization using Parallel Computing

Saumya Bhatnagar
16338296

November 12, 2018

A Master's Degree Project submitted in partial fulfilment
of the requirements for the degree of
M.Sc. (High-Performance Computing)

Declaration

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at `http://www.tcd.ie/calendar`.

I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at `http://tcd-ie.libguides.com/plagiarism/ready-steady-write`.

Signed: _____ Date: _____

Abstract

The rising importance of Artificial Intelligence and Machine Learning has demanded data scientists and researchers to discover new age computer systems that are capable of self-learning, reduce human effort and achieve high accuracy while performing tasks. The development of Artificial Neural Networks; a biologically inspired computing system, has emerged as one of the most effective solution to achieve this. The challenge, however, is optimizing Artificial Neural Networks to achieve high accuracy and speed. Given the vast selection of configuration variables (aka hyper-parameters), optimizing Neural Networks becomes a challenging and complex task. Various solutions to automate this task have since emerged. This thesis seeks to explore and compare the effectiveness of various methods that automate the Neural Networks optimizing process. An in-depth comparison between Genetic Algorithm, Parallel Genetic Algorithm and Quantum inspired Genetic Algorithm to optimize Neural Networks to achieve high accuracy and speed in processing Cifar-10 dataset has been presented in this thesis.

Acknowledgements

I would like to express my deepest appreciation to all those who provided me the possibility to complete this thesis. To begin, I would like to thank my dissertation supervisor, Dr. Darach Golden for his continual help, guidance and enthusiasm throughout the course of this research. It was through his supervision that I developed deep curiosity to dwell deeper into understanding Neural Networks and research effective new ways to optimize them. I would also like to thank my parents who taught me to strive for nothing short of 'excellence'. This has been a period of intense learning, both academically and personally. Your contributions were invaluable.

Contents

List of Figures

1 INTRODUCTION

1.1 Motivation

The rise of the Fourth Industrial Revolution is characterized by the emergence of a fusion of technologies which are blurring the boundaries between the physical, digital, and biological realms (1). Technologies such Artificial Intelligence, Machine Learning/Neural Networks and Quantum Computing have already begun to disrupt existing markets and value networks. The rising demand for these 'new age' technologies has enticed data scientists to focus efforts on developing more efficient accurate computing systems that have a wide application base. The advent of Neural Networks, a computing system that mimics the biological neural system of humans(2), has perhaps, been among the most important developments in the recent past.

The direct application of human nervous system into pattern recognition, aka "perception", has had a profound influence on the evolution of machine learning and artificial intelligence. Neural networks have been used to solve a wide spectrum of problems. Businesses and governmental organizations have used Neural Networks to facilitate sales forecasting, vehicular traffic identification, customer research, risk management and developing games for commercial use. Furthermore, this new age computing system has successfully applied to solve vehicular traffic identification, classifying images for human-computer interactions and for natural language processing, etc. The benefits of Neural Networks model are undeniable. However, a major factor limiting the use of Neural Networks in many cases, is the problem of achieving high accuracy and speed. The selection of configuration variables, a.k.a. "hyper-parameters", helps optimize the accuracy and efficiency. A Neural Network continually attempts to readjust the hyper-parameters to fetch best results. Thus, hyper-parameter optimization is central to Neural Network modeling. Some business bet on hand tuning by experts. Researchers, however, have embarked on a journey to find method for automating the process. A remarkable upgrade over the hand tuning method is the use of Genetic Algorithm (GA). Much similar to Neural Networks, the inspiration for GA finds its genesis in the biological sphere. GA uses a hybrid model of random search and generations. In this thesis, various concepts from GA such as exploitation of parents' features from previous

generation to cross-breed, exploration in terms of randomly initializing population from the sample set based on the fitness of the population group & mutation has been taken.

However, a major limitation of using GA is that the distant result of thousands of generations says very little about the effectiveness of the approach. To tackle the challenge of high computation cost of implementing evolutionary based optimization methods (e.g. in this thesis – genetic algorithm), parallel computing sweeps across the generations with minimal time, by creating parallel instances of the tasks. Depending on the job and resources, parallel instructions can be extended among nodes, cores or threads. Parallel computing has a strategic advantage of interdependence and/or independence of the parallel extension thus used. Parallel computing concepts can be borrowed by any programming language for staging concurrency.

Irrespective of python's versatile nature, with diverse libraries, innovative user interfaces, python has a reputation for being slow and unfit for High Performance Computing. Nonetheless, python being a competent language, has practical experience for scientific computing tasks. Fortunately, this research would comprehend the compatibility of python with HPC. Apart from multithreading and multiprocessing, python for parallel computing also characterizes MPI based parallel processing. Based on MPI-2 C++ bindings, module Mpi4py is python interface to MPI. (3). Expedition of deep learning contextually categorizing the hyper-parameters, adaptation of genetic algorithm for optimization, and erudition of Mpi4py, are therefore the key motivating factors behind this project.

1.2 Research Question

The rising importance of Neural Networks has undoubtedly caught the attention of an increasing number of data scientists and practitioners to find new application bases for the same. However, one of the major constraints during implementation is the tuning of hyper-parameters. Optimization is central to achieving high efficiency and accuracy of results. Traditionally, hand/manual tuning of hyper-parameters had been the norm. However, in an attempt to find more effective methods for optimization, data scientists have zealously focused efforts on finding newer methods for achieving high accuracy and speed. This has given rise to several advancements in automated hyper-parameter tuning methods. While a host of optimization algorithms/methodologies exist, vast existing body of research shows GA(4) to be substantially more accurate and efficient in tuning Neural Networks.

One major limitation, however, has been that Genetic Algorithms are slow. GA requires tens of thousands of generations to be run (which has proved to be a tenuous task). Implementation of Parallel computing could potentially be a solution to solve this

problem.

The research question is:
How effective is Mpi4py(3) based GA for optimizing Neural Networks to achieve both high accuracy and high efficiency on the Cifar-10 dataset?
While improvising Neural Network, exclusion of probable values for hyper-parameters may dominate poor performance. This thesis seeks to explore a methodology with all probable values in search space domain, hybridizing with concurrent scaling so as to avoid losing out on latency.

1.3 Research Objectives

As highlighted in the previous sections, the overall purpose of this study is to present a comparison between the different ways of optimizing and parallelizing massive amounts of data using aggregated computing power. This facilitates in analyzing the performance of Mpi4py parallel programs and finally, determine what the best practices to be employed are. A overview of the goals and objectives of this research are presented below.

1. To explore the impact of parallel computing
   (a) Present a comparison between Sequential and Parallel Computing.
   (b) Compare Quantum inspired algorithm with Parallel Algorithm.
2. To identify the best methods to parallelize computation process. Provide a comprehensive comparison between Mpi4py for Single Population Fine Grained Algorithms and Multiple-population Coarse Grained Algorithms (Island Model)
   (a) Conduct performance analysis of the time taken to process the data
   (b) Conduct performance analysis of accuracy achieved
3. To highlight the impact of using different combinations of hyperparameters (using cifar-10 as input) on:
   (a) accuracy
   (b) time taken

   The (a) Activation function (b) Optimizer (c) Number of Neurons (d) Number of layers and (e) Dropout have been taken as variables (hyperparameters).
4. Overall, this thesis seeks to provide a recommendation for the best set of hyperparameters for initialization for cifer-10 dataset.

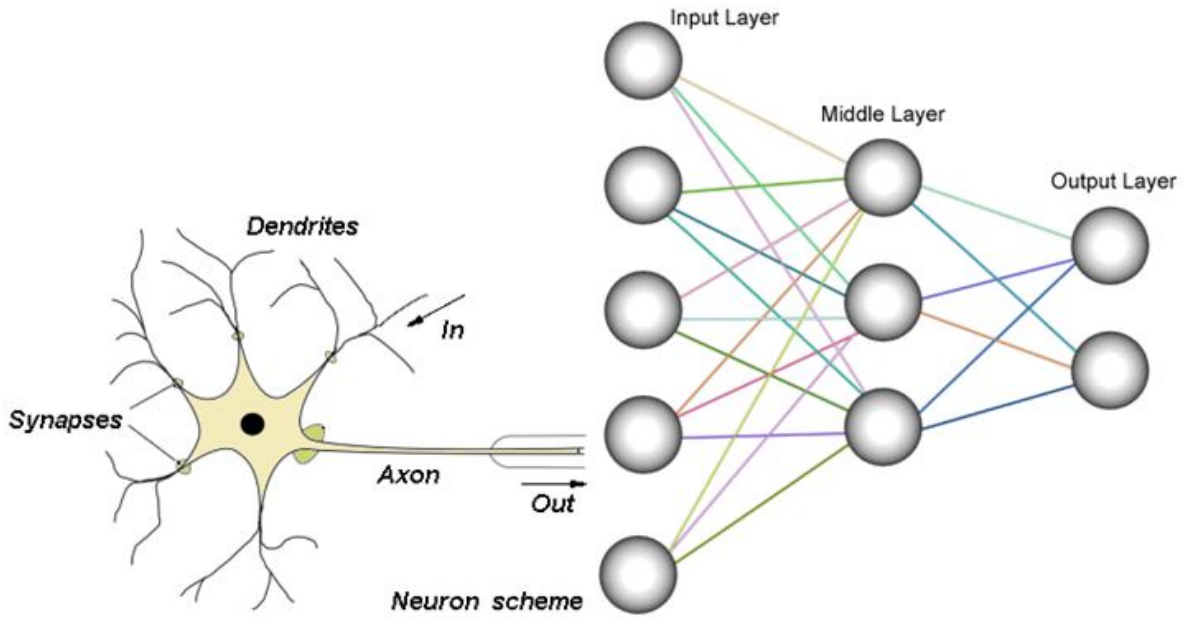

Figure 1.1: Biological vs Artificial Neuron, source

1.4 Theory

1.4.1 Aspects of Neural Network

With th advent of Neural Networks, there has been a resurgence in the field of High Performance Computing(2). Neural Networks have become of high importance to Machine Learning practitioners due to the wide spread computational application possibilities. Neural Networks can be used to rapidly provide a collectively-computed solution (a digital output) to a problem on the basis of analog input information (5). Exploring the definition of Neural Networks provides a deeper insight onto how the computational system operates and provides promising results.

By definition, Neural Network computing architectures mimic the neuronal structure of the mammalian cerebral cortex.

A Neural Network consists of a number of interconnected processing elements, commonly referred to as 'neurons' (6). The 'neurons' are arranged into multiple 'layers'. Neurons within each layer are connected to and interact with neurons in the subsequent layers via 'weighted connections'. The 'weighted connections' determine the level of influence between the interconnected neurons. A pictorial presentation of Neural Networks is presented in Figure 1.1.

As depicted in the Figure 1.2, Neural Networks consist of multiple layers. Signals pass from the input layer to the output layer. Data is presented to the input layer and the output layer

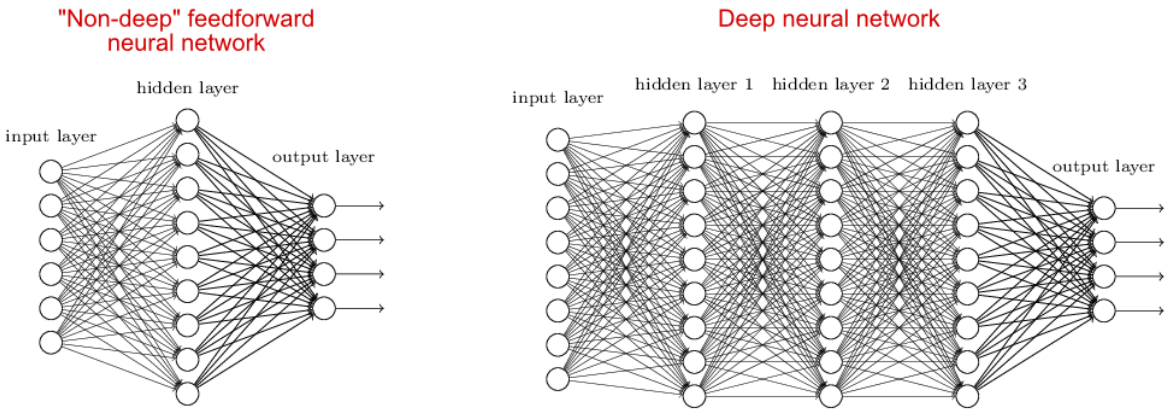

Figure 1.2: Simple Neural Network vs Deep Learning Neural Network, source

holds the final responses. The "hidden layers" in between help represent and commute the association between the layers.

One of the major benefits of Neural Networks is that they are capable of learning by adapting to the association of weights. Two strategies for training Neural Networks exist; **(i) Backward Propagation and (ii) Feedforward Neural Network training**. Webros's(7) backward propagation algorithm significantly accelerated the training of Multilayer Networks by distributing the error term (or inaccuracy of fitting the Neural Network) back up through the layers. The error terms shrink exponentially as they propagate through layers. Deep Feedforward networks, also often called Neural Networks or Multi-layer perceptron (MLPs), are the quintessential deep-learning models. Feedforward Networks form the basis of many important commercial applications. For example, the Convolutional Neural Network (CNN) used for object recognition from photos are a specialized type of Feedforward network(8). Both MLPs and CNNs come within Deep Learning. Multilayer Perceptrons (MLPs) represent the most general Feedforward Neural Network model possible; they are organized in layers, such that every neuron within a layer receives its own copy of all the outputs of the previous layer as its input. The major difference between MLPs and CNNs is that CNN are less densely connected. This helps CNN scale down on the number of parameters through sharing of weights, conserving the information. Spatially close inputs are assumed to be correlated. For this reason, the accuracy gained through the use of CNN outweighs the accuracy derived through the use of MLP. The graphical representation of MLP and CNN is shown in Figure 1.3 and 1.4.

Hyper-parameters are variables which determine how the network is structured. In Machine Learning, Hyper-parameter Tuning or Hyper-parameter Optimization is the task of selecting a set of optimal parameters for solving an ML problem using a learning algorithm. Given the large number of 'hyper-parameters' which exist, choosing the appropriate architecture for achieving accurate results has remained as a subject of concern. Although

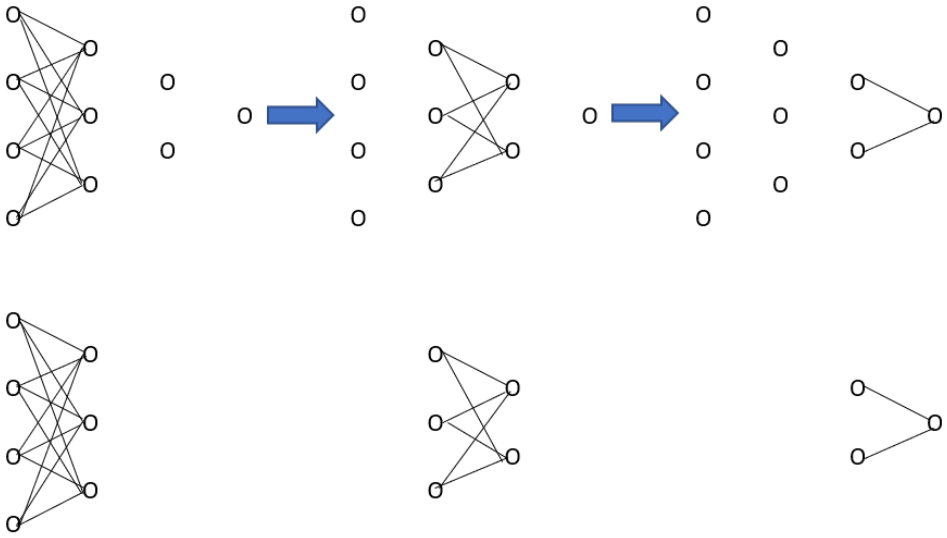

Figure 1.3: Graphical Depiction of Multilayer Perceptrons

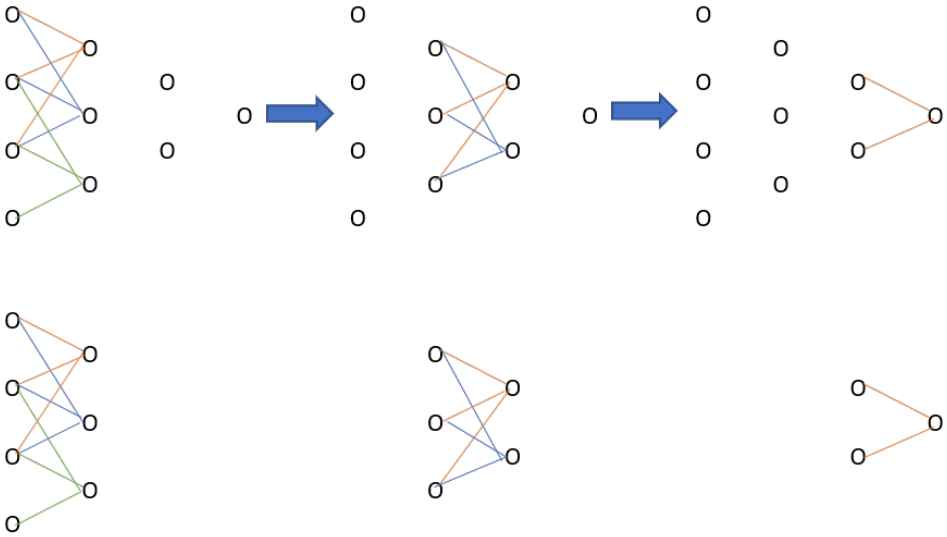

Figure 1.4: Graphical Depiction of Convolutional Neural Networks

many parameters exist, five parameters have been selected as configuration variables for the purpose of testing the results in this thesis. The structure of Artificial Neural Network is directly influenced by these parameters. Also, these parameters determine how the network is trained to achieve accurate results (thus, they are set before optimizing the weights and bias). A few other examples of parameters include learning rate and loss function.

1.4.2 Survival of the fittest

The previous section had introduced Convolutional Neural Network Model. This section seeks to provide an overview of the optimization technique used with brief detailing on the concepts of Genetic Algorithm (GA) that are used.

Genetic Algorithms (GA) is an evolutionary algorithm inspired by Charles Darwin's (1859)(9) Theory on Natural Selection and Genetic Mutation. Darwin's Theory of Natural Evolution points to premise that all species of organisms arise and develop through the mechanism of natural selection and genetic mutation. Organisms develop through a series of small, inherited variations that eventually improve the organism's ability to compete, survive, and reproduce. This is the basis of GA - a population-based metaheuristic optimization algorithm. Thus, Genetic algorithms can be applied to generate high quality solutions to optimization and search problems by relying on bio-inspired operators such as mutation, crossover and selection(10).

In this thesis, the optimization technique is based on Natural Selection, Cross-Breeding over generations. The optimization technique - **EETO (Exploration & Exploitation Trade-off based Optimization) method** makes use of the concepts inspired by GA to optimize the Neural Net as detailed below:

1. Initialization of Parents' Genes: In the first step, random candidate solutions (parents' features/ chromosomes) are generated. These chromosomes are further used to breed/produce new populations.
2. Iterative Generations: GA takes an iterative process where a sequence of outcomes is continuously developed until an optimal solution is provided. A population within each iteration is referred to as a 'generation'.
3. Fitness Evaluation: The 'quality of the solutions' is represented as 'fitness'. Candidates of solutions are developed through evolution at each generation. The relative 'fitness' of the candidate solutions is expected to be higher through passing generations.
4. Breeding: Breeding takes place after 'Fitness Evaluation' and involves two concepts:
   (a) **Exploitation**: This step is based on 'Natural Selection'. Set of candidates are selected within each 'generation' based on 'fitness'. Relatively 'fitter' solutions

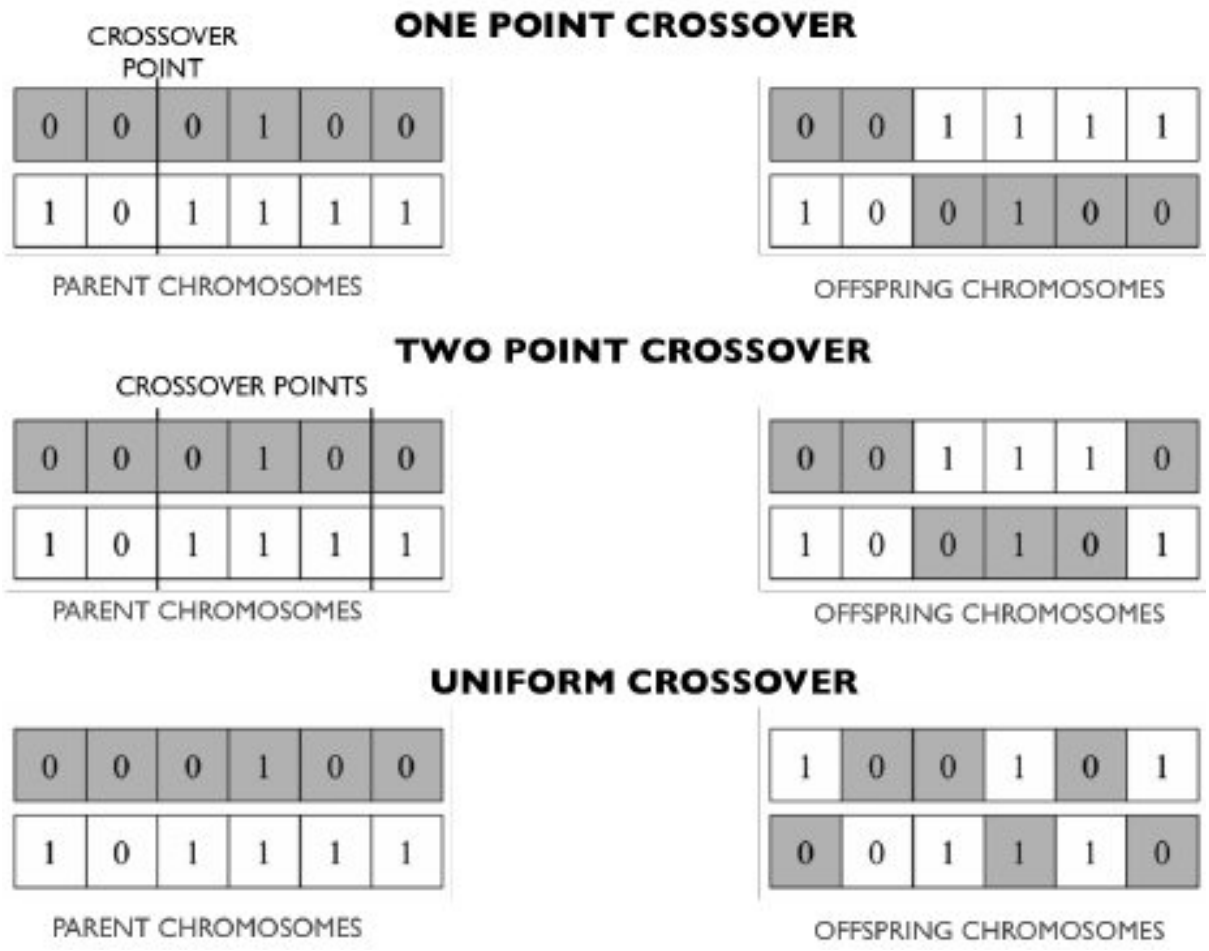

Figure 1.5: Types of crossover, source

are selected from previous generations. During 'Crossover', the algorithm exploits the features of the selected parents to generate a new offspring. There are multiple methods of crossovers. In a single point crossover, only one crossover point is selected in contrast with multi-crossover point. This thesis uses Uniform Crossover. A Uniform Crossover point randomly selects from the parents. In contrast, under Arithmetic Algorithm, arithmetic operations are performed to produce off-springs. A graphical representation of these algorithms is presented in Figure 1.5.

(b) **Exploration**: This step involves killing/deleting the least fitting individual and re-initializing the individual randomly ensuring exploration to get out of local maxima and produce optimal solutions by reaching global maxima.

5. Mutation: Each candidate solution has a set of properties (its chromosomes or genotype) which can be mutated and altered; traditionally, solutions are represented in binary as strings of 0s and 1s, but other encodings are also possible(11). It is important noting that the mutation rate can influence the convergence. A sub-optimal

mutation rate could lead to a 'Genetic Drift'. A Genetic drift is the change in the frequency of an existing gene variant (allele) in a population due to random sampling(12) of the off-springs (candidate solutions). Genetic Drift plays a mirror role in the evolution process. In case of slow mutation rate, Genetic Variation is likely to be reduced, causing reduced incremental rate of 'average fitness' through passing generations. On the contrary, a suboptimal 'high mutation rate' increases the risk of good solutions being lost.

6. Termination: The final step is termination. Termination is normally achieved when the iterative process can no longer continue. In this thesis, the termination circumstances that are considered are listed below:
   (a) Number of generation constraints
   (b) In case if outcome satisfactory meets the minimum criteria.

1.4.3 Quantum Computation

In classical computers information is stored in bits(0 or 1). However, in a quantum computer, the information is stored in qubit, building block of a quantum computer, which is a spin-1/2 of particle such as a proton, neutron, or electron. So, a quantum memory register is composed of several spin-1/2 particles, or qubits. A qubit stores a superposition of 0 and 1, after measurement the state collapses from it's superposition state to one of these two states.

While in a superposition state, a qubit can take any value between 0 and 1. This increased sample space is the key to evaluate quantum inspired genetic algorithm.

In this thesis, a new methodology has been tried-
QEETO: Quantum inspired Exploration vs Exploitation based Optimization method. The EETO method has been inspired by beauty of superposition from Quantum Mechanics. From a random theta, a new position of Bloch sphere is taken and using the resulting value the hyper-parameter is defined.

1.5 Overview

The previous subsections of this thesis have been successful in providing foundation for this research paper by defining the research question, describing the motivation for the topic and discussing the goals and objectives of this research. Important terms such as

'hyperparameters', 'Neural Networks' and Genetic Algorithms have been defined and discussed.

The following sections aim to build on this foundation and develop on the existing body of knowledge on Neural Networks. This section seeks to provide a structural overview of the thesis.

Chapter 2 reviews the existing body of literature associated with Neural Networks and provides an overview of the latest developments taking place. Finally, a comparison between the various methods of tuning Neural Networks, including Grid Search, Random Search, Bayesian Search, Gradient Search and Evolutionary Search have been presented in this chapter.

Chapter 3 discusses the design/structure of the algorithm used. Details on the programming language (python) and library, SLURM, various TCHPC clusters and their compatability with Machine Learning library "keras/tensorflow" has been compared, parallel computing and details on the methodology for the research have been illustrated in this section.

Chapter 4 has been dedicated to discussing the research findings. 'Implementation and Data Analysis' process of MPi4py for optimizing Neural Networks using the optimization method is comprehensively discussed in this chapter. This section talks about the steps in getting the software up and running. This section provides data analysis of the finding of the sequential code, various parallel codes and the quantum inspired algorithm. **Chapter 6** concludes the thesis.

2 STATE OF THE ART

2.1 A review of Neural Networks Optimization Procedure

The previous sections have highlighted that the accuracy of results achieved through Neural Networks is influenced by hyper-parameter selection. In evaluating accuracy, Neural Network Optimization remains as a crucial task. This section reviews the models/approaches to automate hyper-parameter tuning process.

2.1.1 Types of tuning:

The different methods for optimizing hyper-parameters are detailed below.

Hand Tuning

This is a non-automated tuning method. Under Hand Tuning method, the design process initiated by designing a simple network, either by directly applying architectures that have shown successes for previous/similar problems or by trying hyper-parameter values that generally seem to be effective in bringing satisfactory results.

Grid Search Method

The Grid Search Method is among the most widely used strategies for hyper-parameter tuning(13). This method involves remunerating all the possible candidates for the solution and systematically analyzing whether each candidate satisfies the problem statement. Grid Search algorithms are generally measured by Cross Validation in the training set(14) and measured by evaluation in a Handout Validation set(15).

Since all the possible combinations of parameters are checked, the Grid Search method is highly exhaustive in nature. Assuming 5 hyper-parameters are taken with 7 values in each,

all possible combinations; 7^5 (16,807) combinations would be checked. The number of nodes/cores required would have to be 7^5 (over 16,000). This is beyond the capacity of the cluster 'Boyle' (taking into account the busyness of the cluster). For this reason, the Grid Search method has not been employed as a part of this thesis.

A Gaussian process analysis of the function from hyper-parameters to validation set performance reveals that for most data sets only a few of the hyper-parameters really matter, but that different hyper-parameters are important on different data sets. This phenomenon makes grid search a poor choice for configuring algorithms for new data sets(13). The random Search method to be discussed in next section overcomes this limitation.

Random Search (RS) Method

The Random Search Method randomly selects hyper-parameter combinations rather than performing an exhaustive search. A statistical distribution for hyper-parameters can be randomly sampled from. Since, Random Search optimization methods don't require the gradient of the problem to be optimized, Random Search Method can be used on functions that are neither differentiable nor continuous. Thus, a large class of optimization problems can be handled by random search techniques(16).

Bayesian Optimization Method

This optimization method typically works by assuming the unknown function is sampled from a Gaussian process and maintains a posterior distribution for this function as observations are made(17). Bayesian optimization method builds a probabilistic model of the function mapping from the hyper-parameter values to objectives assessed on a Validation Set. Since they work on Sequential model-based optimization (SMBO), Bayesian Optimization methods can obtain high quality results with minimal human effort, though then the chances of getting into the local maxima-minima may increase.

The majority of automatic hyper-parameter tuning mechanisms are sequential optimization methods: the result of each training run with a particular set of hyper-parameters is used as knowledge to inform the subsequent hyper-parameters searched(18). Various other have attempted to use Bayesian Optimization method to train neural networks(19).

However, performing the task sequentially makes the process prohibitively slow. Number of methods have emerged to tackle this issue and speed up the process(20). It has been found that the speed of the Bayesian Optimization can be increased by parallelizing the process(21).

Scikit-learn or Auto-sklearn (built on top of Scikit-learn) features packages for the aforementioned Bayesian Optimization methods(22).

Gradient based optimization

Gradient based optimization methods are based on iterative gradient descent scheme. The method involves computation of gradient with respect to configuration variables and later optimizing the variables using gradient descent(23). "hypergrad" is a Python package for differentiation with respect to hyper-parameters (24).

Evolution-based optimization

Evolutionary optimization uses evolutionary algorithm to search the space of hyperparameters for a given algorithm(25).

The various evolutionary based algorithms involve DEvol, DEAP, TPOT and PBT, and are discussed in the next section.

2.1.2 Evolutionary based optimization – A perspective

Genetic Algorithms reiterate the principles of natural evolution in a stylized way, applied to a problem, they do not work on one particular solution but on populations of solutions, which evolve until they converge at levels regarded as optimal(26). GAs have been successfully implemented in diverse areas to improve operational efficiency and to provide efficient solutions to real world problems. For example, in the banking and finance sector, (26) finds that GAs have proven to be a very effective instrument in computing the risk of insolvency. Genetic Algorithms have also been successfully implemented in robotics. GA has been used for autonomous robot navigation(27). Furthermore, the last few years have seen important advances in the use of Genetic Algorithms to address challenging optimization problems in industrial engineering(28). This section digs into the various genetic-based algorithms used in the market.

DEvol and DEAP

DEvol (Deep Neural Network Evolution)(29) and DEAP(30) both play with the model-based population of Neural Network genome. A pool of networks is selected having different set of configuration variables, and then the generations of breeding and natural selection follows.

KERAS library was first ever used for Genetic Algorithm in Neural Networks in DEvol research. However, DEvol doesn't support parallel computation. Since, training hundreds or thousands of different models to evaluate the fitness of each is not always feasible, DEvol is considered computationally expensive and time consuming.

High Performance Computing allows scientists and engineers to solve complex science, engineering, and business problems using applications that require high bandwidth, enhanced networking, and very high compute capabilities. Looking at the parallel methods implemented using Evolutionary Algorithms for Neural network Optimization, the DEAP framework supports parallelism using multiprocessing.

TPOT

In their influential article, Olsen et. al. (2016) report the development of an evolutionary algorithm called 'Tree-based Pipeline Optimization Tool' (TPOT). TPOT uses a version of genetic programming to automatically design and optimize a series of data transformations and machine learning models that attempt to maximize the classification accuracy for a given supervised learning data set(31). TPOT automatically detects all possible pipelines to find the best fit for the data.

A typical Neural Network application requires practitioners to apply the appropriate data pre-processing, feature engineering, feature extraction, and feature selection methods that make the dataset amenable for machine learning. TPOT implements four main types of pipeline operators: feature selection, preprocessors, decomposition, and models(32).

Once the search is complete, TPOT provides a python code for the best possible pipeline available for the data set. A pictorial depiction of the same is shown in Figure 2.1

TPOT uses an evolutionary algorithm to automatically design and optimize a series of standard machine learning operations (i.e., a pipeline) that maximize the final classifier's accuracy on a supervised classification dataset by using Genetic Algorithms and Pareto Optimization as depicted in Figure 2.2.

PBT

In the seminal research on Neural Network training(18), PBT or Population Based Training applies a hybrid model of random search and hand-tuning. Randomly initiated networks will be bred, trained and the hyper-parameters of the best performing network will be exploited by the rest of the networks. The process will continue till the number of generations are exhausted or the desired fitness of the network is achieved.

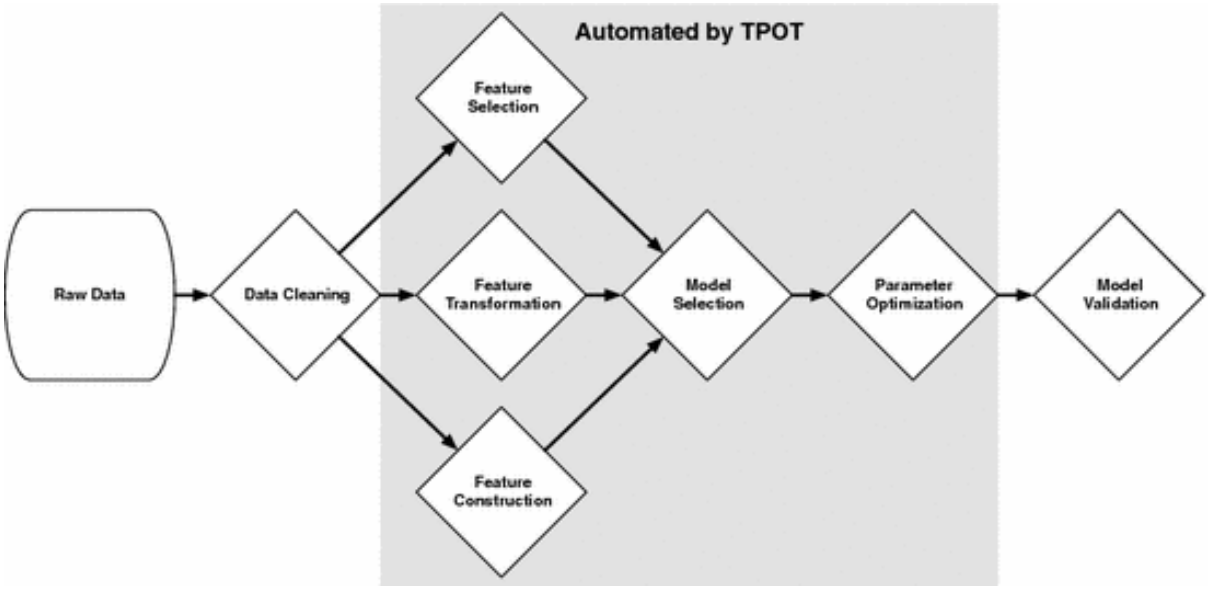

Figure 2.1: Machine Learning Process using TPOT, source

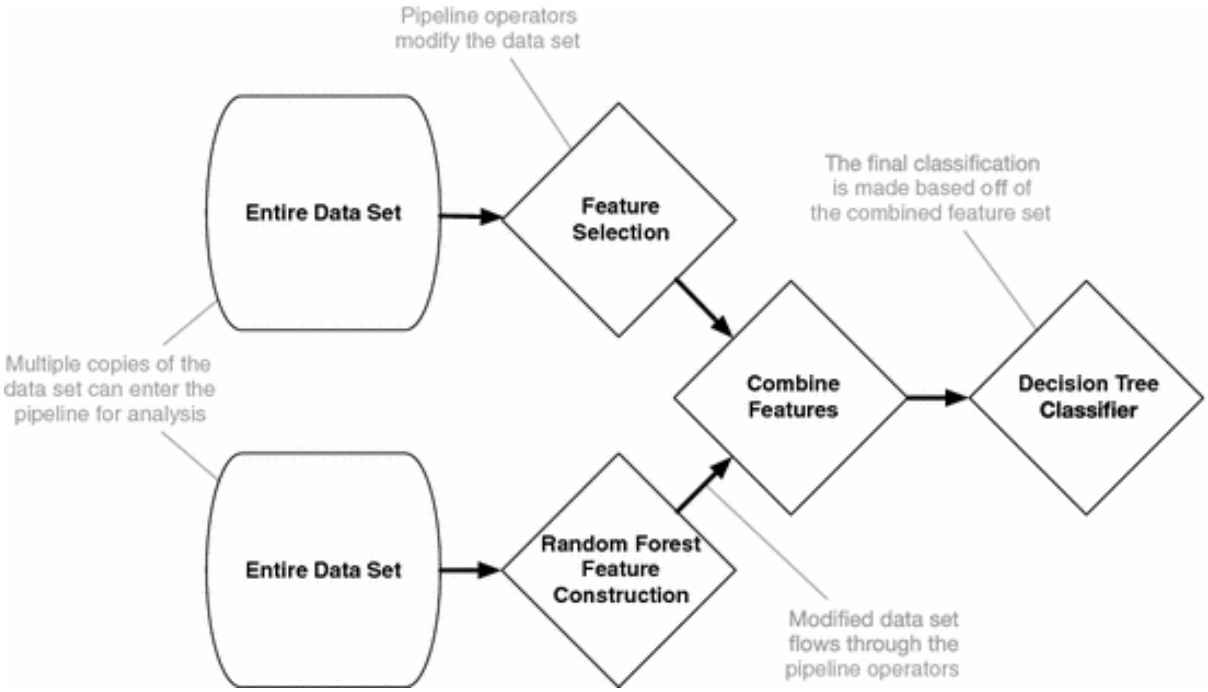

Figure 2.2: TPOT, source

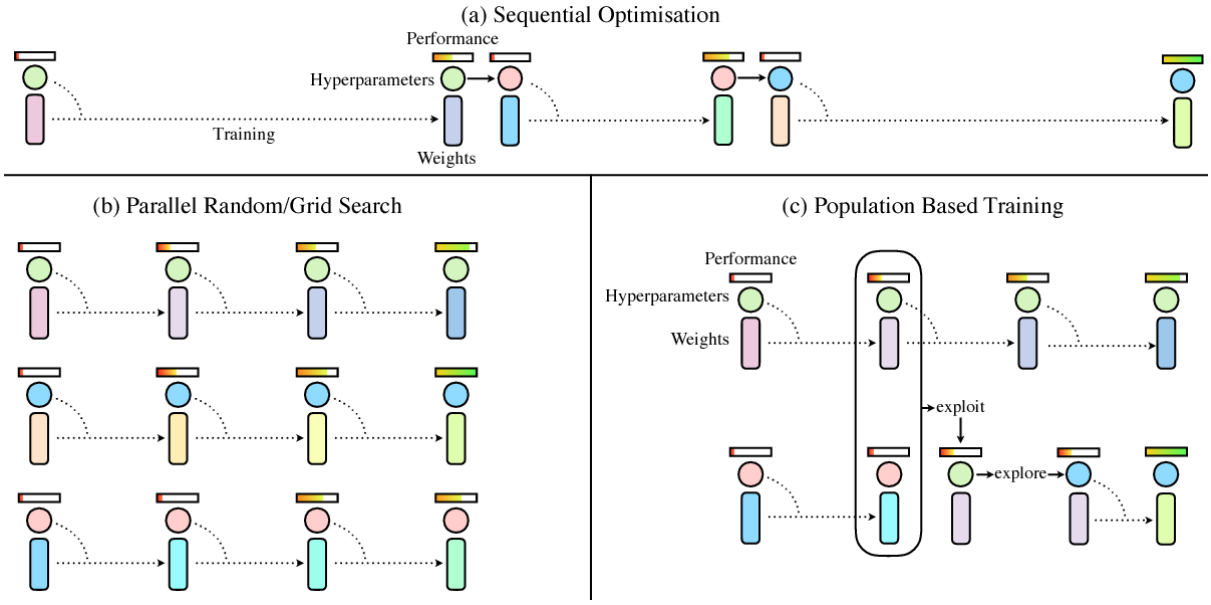

Figure 2.3: Population Based Training of Neural Networks, source

2.1.3 Quantum Inspired Genetic algorithm

Due to seminal article showing polynomial run time of integer factorization(33), researchers have been taking interest in emulating quantum computer on a classical machine. A result of which is Quantum Inspired Genetic Algorithm(34).

Hybrid Genetic Algorithm (HGA)(35) is a GA that combines quantum operators (measure, quantum chromosomes, etc.) with classical genetic operators (crossover and mutation). HGA has been used for global optimization problems previously(36)(37)(38).

2.1.4 Overview

This section illustrated an overview of the emerging wisdom surrounding the development and the wide application base for Deep Neural Networks hyper-parameter optimization. The various methods by which Neural Networks are being adapted have been comprehensively discussed in this section.

One issue that was completely overlooked in discussed implementations is the possibility of reaching local maxima instead of global maxima. Clearly this is a very important concern. Exploiting the hyper-parameters of the network with better accuracy, would impede the usage of hyper-parameters giving lower accuracy. This limits the possibility for achieving the best results.

3 DESIGN

3.0.1 The framework

The scope of this study is to access best set of hyper-parameters and train the neural network dataset. Previous section discussed software tools and algorithms that are currently in use by machine learning researchers. In this section, a panorama of the algorithm is discussed. The remaining sections, section 3.2, 3.3, 3.4 and 3.5 talk about the next stage of the panorama – design language and libraries, parallel computing and architectures.

The algorithm is designed to quickly get to the speed of neural network with the best possible accuracy; from first principles in Neural Network and Genetic Algorithm, all the way to discussions of some of the intricate details in Parallel Processing using MPI, with the purposes of achieving respectable performance on a machine learning benchmark: CIFAR10(classification of small images across 10 distinct classes: plane, car, bird, cat, deer, dog, frog, horse, ship and truck).

The model does so by selecting various hyper-parameters with different possible value combinations, followed by initiating the set of networks using those selected values and using the algorithm to crossbreed networks to compute the relative importance of parameter set to maximize the accuracy.

With this, we will have acquired the fundamental structure of the model, to apply the algorithm to the interesting and difficult problem of fine tuning neural network parameters, with the trick of a parallelism to make it better than the initial baseline serial model (in terms of time and memory usage)

The architectures of the basic Optimization Algorithm - 'EETO' and the optimization algorithm 'EETO' for Neural Network Tuning are shown in Figure 3.1 and 3.2 respectively. A parallel implementation has also been applied which will be discussed in later section. Moreover, a parallel hybrid model -Island Model inspired by Island Model - Genetic algorithm is also implemented(39). The structure of the Island model is shown in Figure 3.3

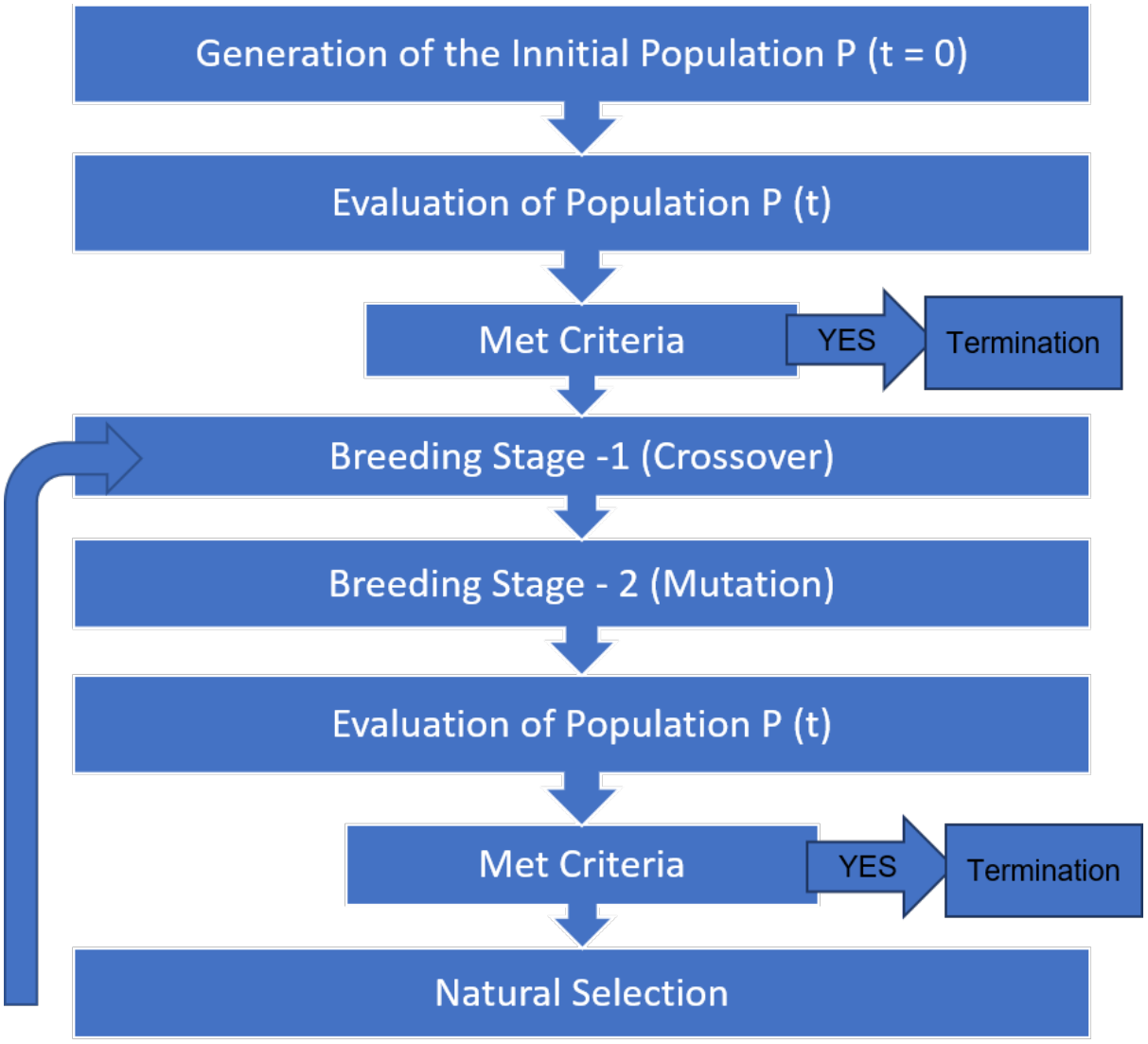

Figure 3.1: Architecture Optimization Algorithm

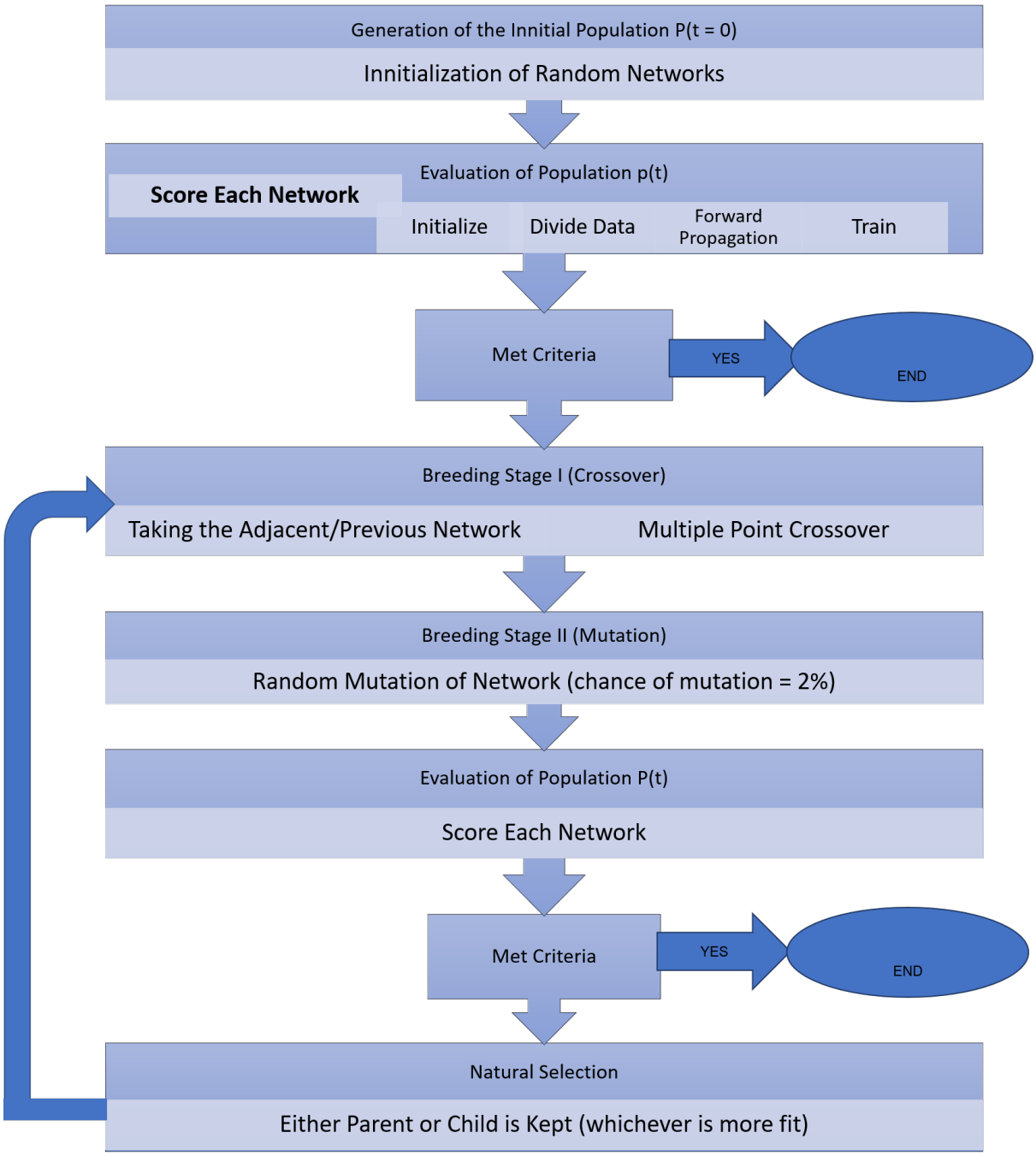

Figure 3.2: Architecture for EETO for Neural Network Tuning

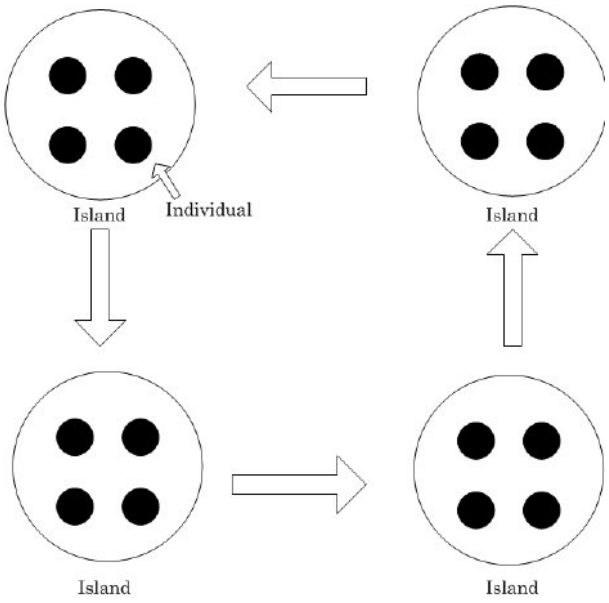

Figure 3.3: Island Model, source

3.0.2 Software development

Python comes with a very active user community and is overtaking in the race of popular programming languages. Even in HPC it has successfully found its way. With the advent of scientific python, python has become a successful contender(40). The language is not only clean, easy to implement but also comes with a vast number of open source libraries. The growing presence of libraries like cython, parallelpython(pp), Numba, Numpy, Mpi4py, makes python a reasonable choice for HPC. "Intel Parallel Studio XE 2017" Distribution for Python release claims to have shown a near native C performance with compilers and library packages optimized for the Intel architecture. Python libraries such as Numpy can make use of multithreading through calls to the Intel Threading Building Blocks (Intel TBB) library. Additionally, Intel Vtune provides line-by-line source code profiling to help find and correct issues causing performance hot spots or bottlenecks in Python source.

In the race to be the fastest while PyPy has proven itself to be more than 500% times faster than C version of python(CPython) (41), still there are algorithms where it is slower than equivalent code in C, involving a tradeoff between time and memory. For parts taking longer than average time, a python interface to C can be created, where data is passed from python and the routine is executed w/o any python overheads. These tools for easier interfacing include PySwip, pyrex and Cython. Although PySwip or Swi-Prolog offers a variety of development environment, multi-threading, comprehensive low-level interface to C, the memory usage of PySwip extensively using pointers slows down the system heavily. On the other hand, Pyrex which claims to be as fast as C language(42), comes with a bunch of limitations, as basic as not being able to access functions like local() and global()(43). A

more recent version - Cython compiler gives C-like performance along with Python bindings for many C and C++ libraries. But Cython is slowed down by Global Interpreter Lock (GIL). The lock keeps the Python virtual machine single-threaded, use of Python threads don't experience any concurrency unless I/O is involved (which frees the GIL). Fortunately, there are ways to overcome GIL mechanism viz. Multiprocessing and Mpi4py by offering a process-based parallelism using subprocesses instead of threads(44).

Multiprocessors make use of shared memory, employ parallel processing techniques which limit their relevance to SMP-based hardware. However, Mpi4py would involve distributed memory. Mpi4py is cluster based and offers high scalability due to the relative absence of shared resources(45). As this thesis aims to exploit multiple nodes of a cluster, mpi4py is being used to leverage parallel computing. Since commands used from Mpi4py module support both mpirun and mpiexec, any of the Mpi execution can be used for running.

Next view of the "neural network" panorama, is the usage of Keras for Convolution Neural Network (CNN). Keras integrates with lower-level deep learning languages (in particular - TensorFlow) and has built-in support for multi-GPU data parallelism. Not only Keras use dozens of hyper-parameters, which are critical for the thesis, it comes with many datasets, models and layers, which can be imported into the code using simple import commands(46).

The complexity of these models could easily be extended beyond these capabilities to include any parameters included in Keras, allowing the creation of more complex architectures.

Handling of SLURM workload manager

Last but not the least, the codes are scheduled using SLURM workload manager. SLURM comes with handy commands (view SLURM documentation using "man <command>") each for scheduling, cancelling, viewing the status and possible starting time of jobs/nodes and managing usage of the node memory, accessing idle nodes, etc which are efficiently applied in this thesis. A refine built feature of this software is that depending on free window on a node, the code can be scheduled for the available time on that node and upon timeout, can be resumed (requires manual changes by the user in SLURM script and MAIN source file script) on another next available node.

3.0.3 Leveraging Parallel Computing

Welcome to the third (and final) aspect in a series of designing of the algorithm. This section will, for the most part, assume familiarity with the previous two in the series. Parallel computing is a type of computation in which many calculations or the execution of processes

are carried out concurrently(47). With the industry-wide switch to multicore and manycore architectures, parallel computing has become the only venue in sight for continued growth in application performance(48). Increasingly, parallel processing is being seen as a cost-effective method for the fast solution of computationally large and data-intensive problems(49). Thus, the same has been applied in this thesis.

Parallel Algorithm primarily deals with dividing the networks into different processors to increase the efficiency of performing tasks and to reduce the time taken. However, various parallelization methods exist for dividing the populations to maximize efficiency. While implementing EETO parallelism In this thesis, analogy has been drawn from two major types of Parallel Genetic Algorithm (PGA): **Single-population fine-grained Genetic Algorithms**, and **Multiple-population Coarse-grained Genetic Algorithms**. These PGAs differ in terms of parallelization. A further description of these PGAs and the usage of each of the PGA is detailed below.

1. Single Population Fine Grained Algorithms (SPFGA) Fine Grained Algorithm consists of multiple individuals evolving independently of each other. Each civilization consists of only one individual. Fine Grained Algorithm prevents premature convergence since each individual evolves independently. Further, this also provides the benefit of greater diversity. For these reasons, this type of algorithm supposedly outperforms all the other types of algorithms when dealing with high dimensional variable spaces(50). Fine-grained PGA is more suitable for custom hardware design(51). Since each population has a single individual, a simple processor core is suitable for performing the tasks.

2. Multiple-population Coarse Grained Algorithms (MPCGA) The coarse-grained (distributed) parallel algorithms assume the division of a large population into several subpopulations which are processed concurrently on different processing nodes(52). They are also known as multi-deme GAs. Sub-populations evolve independently with only occasional exchange of individuals between them. Independent evolution allows for greater diversity and prevents premature convergence (similar to Fined Grained Populations).

The architecture showing parallel implementation has been shown in Figure 3.4.

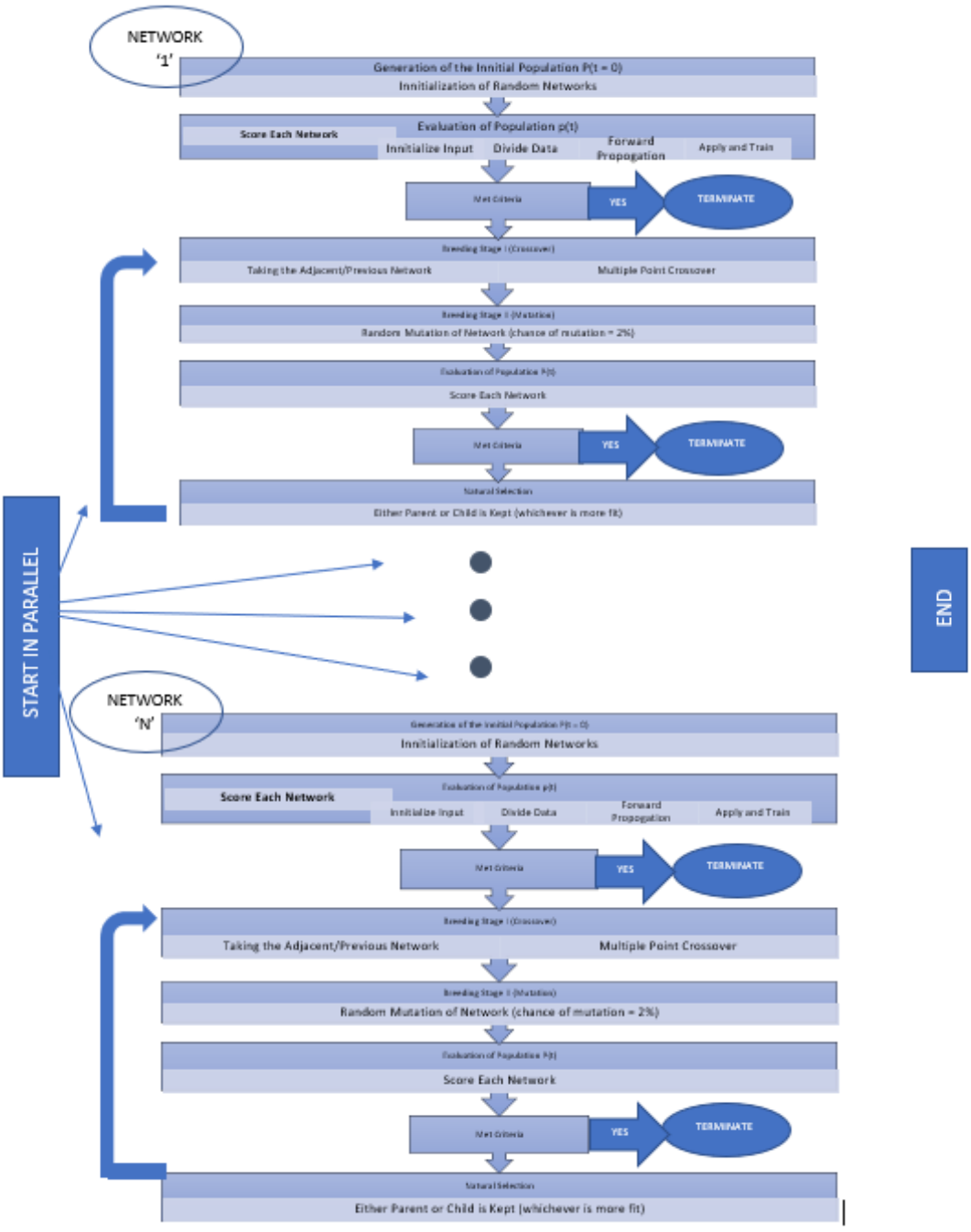

Figure 3.4: Architecture for Parallel Neural Network Tuning

3.1 HPC Cluster Architecture: Clusters, Installations

3.1.1 Development: Using Chuck Cluster

The journey started with installations of softwares and libraries on chuck for development. Starting with python 3.7 which was not officially supported till 2018 June end, python 3.5 was the next version tried for this thesis, switching back and forth with python 2.7(supports best for tensorflow – dependency for Keras).

As discussed in the previous article, the main library used for neural networks training, i.e. keras was installed using pip3, which installed the latest version of keras(2.2.2). Since it used Tensorflow backend, the next challenge was to install Tensorflow. Tensorflow works on 64 bit systems. And the clusters' architecture used in TCHPC are 64bit. Somehow, the tensorflow latest version was throwing error: "Operation not permitted", so switching to older version of tensorflow, i.e. tensorflow1.5 worked. Next step was to run the code, but that introduced to the next pitfall which was Binary incompatibility from numpy. Error: "numpy.dtype size changed, may indicate binary incompatibility" The solution is to check what packages depend on numpy and find out against what numpy version they were built. The error was a result of importing Keras that was compiled against a newer version of numpy than is installed. Instead, installation of numpy1.14.5 worked. For that first the newer version of numpy is to be uninstalled and then the older version is to be installed. Now the binaries in the dependent packages are compatible.

The summary of the steps is shown in the sequence graph in Figure 4.1.

All these installed adding '–user' at the end to install it in the home directory.

3.1.2 Deployment Trial 1: Using Lonsdale Cluster

The major problem faced was inability of getting Keras to work on Lonsdale or Kelvin. Following the steps taken while assessing the resources of the cluster Chuck, the tensorflow installation step threw error:

> "`/usr/lib64/libstdc++.so.6: version 'GLIBCXX_3.4.14' not found`"

Digging further into the same showed, Chuck is on RHEL 7x and Lonsdale is on RHEL 6x. And it turns out that they have different GLIBC. Default GLIBC on Lonsdale is 2.12. However, Tensorflow won't work on anything except the absolute latest hardware. Tensorflow needs GLIBC >= 2.17 So, to be able to get Tensorflow on Lonsdale, the steps to be taken included:

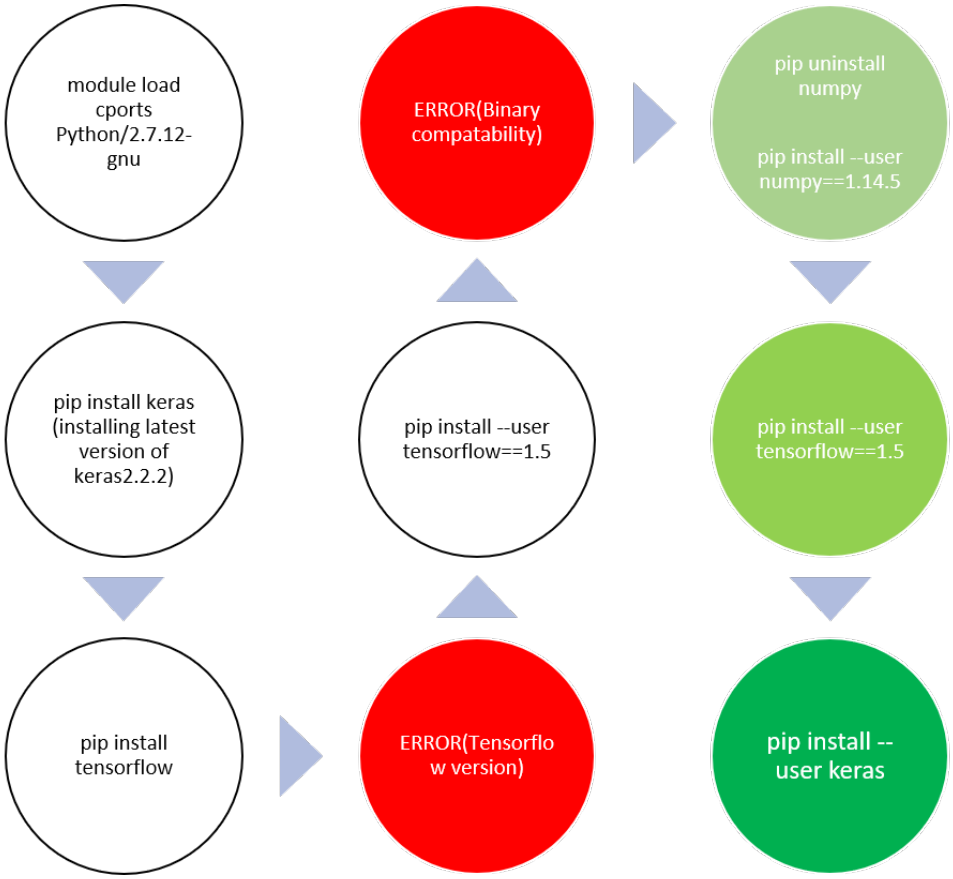

Figure 3.5: Steps on TCHPC Chuck Cluster

1. Installation of a new gcc version,
2. then compile bazel (a tool to compile tensorflow).

3.1.3 Deployment Trial 2: Using Kelvin Cluster

The Kelvin Cluster has RHEL6.10 and the concern of GLIBC not supporting Tensorflow holds while using Kelvin Cluster. Solution to this would be to get to the cluster which has higher version Scientific Linux

3.1.4 Deployment Trial 3: Using Boyle Cluster

While there was a mismatch of the version and backend software requirements on Lonsdale and Kelvin, Boyle has the same Scientific Linux release as Chuck and thus, promises implementation of code.

Lonsdale/ Kelvin:

1. Scientific Linux release 6.10
2. ldd (GNU libc) 2.12

Boyle/ Chuck:

1. Scientific Linux release 7.5 (Nitrogen)
2. ldd (GNU libc) 2.17

A brief of the HPC clusters(54) is as follows:

Title	Boyle	Kelvin	Lonsdale
Server	Scientific Linux release 7.5 (Nitrogen)	Scientific Linux release 6.10 (Carbon)	Scientific Linux release 6.10 (Carbon)
Kernel Version	3.10.0-862.3.3.el7.x86_64	2.6.32-754.3.5.el6.x86_64	2.6.32-754.3.5.el6.x86_64
GCC version	Red Hat 4.8.5	Red Hat 4.4.7	Red Hat 4.4.7
Ldd Gnu LIBC	2.17	2.12	2.12
Processor Type	Intel	Intel	Opteron
RAM	4.6TB	2.4TB	3.2TB
Interconnect	Mellanox ConnectX-3 QDR	Qlogic Infiniband QDR	Infiniband DDR
Theoretical Peak Performance	43.378TF	12.76TF	11.33TF
Linpack Score	33.536TF		8.9TF
Number of Nodes	33	100	154
ArchitecturE	64	64	64
Ram/node	64	24	16
RAM	4.6TB	2.4TB	3.2TB
Num cores	804	1200	1243
Number of sockets/node	2	2	2
Number of cores per socket	8	6	4
Clock Speed	2.00GHz	2.66GHz	2.30GHz
Available to	Selected Groups	Irish Researchers	TCD Researchers

Side-stepping re-installations of any library and preservation of the work done on compatibility testing of libraries, one of the four tools viz. Virtualenv, Native pip, Docker, or Anaconda can be used. This thesis makes use of Anaconda. Anaconda or conda is a mechanism to keep the dependencies required by different projects in separate places, making it reliable and easy for using various libraries. Conda installs TensorFlow and all the required and/or dependent packages. Activating the environment will activate all the packages installed in that environment. The environment packages can be viewed using "conda list". Installing TensorFlow with pip in a conda environment may cause issues. The conda installation would take precendent over the pip installation. This means that after importing tensorflow in python, "tensorflow.__version__" would return the version installed through conda. Last but not the least is installation of Mpi4py, which can be installed using conda install command.

Recap of configurations:

Server: Scientific Linux release 7.5 (Nitrogen)

conda 4.5.10

Python 2.7.15 :: Anaconda, Inc.

numpy 1.15.0

Keras 2.2.2

Tensorflow 1.5

Mpi4py 2.0.0

gcc: gcc (GCC) 4.8.5 20150623 (Red Hat 4.8.5-28)

ld: ldd (GNU libc) 2.17

3.2 Installation steps

See Figure 4.2 for a detailed introduction of how the installation and code works and some basic usage.

3.3 Introduction to the code for Simulation

The sections explains the pseudo code for the various algorithms used.

Pseudo Codes Neural Network Tuning - NNT

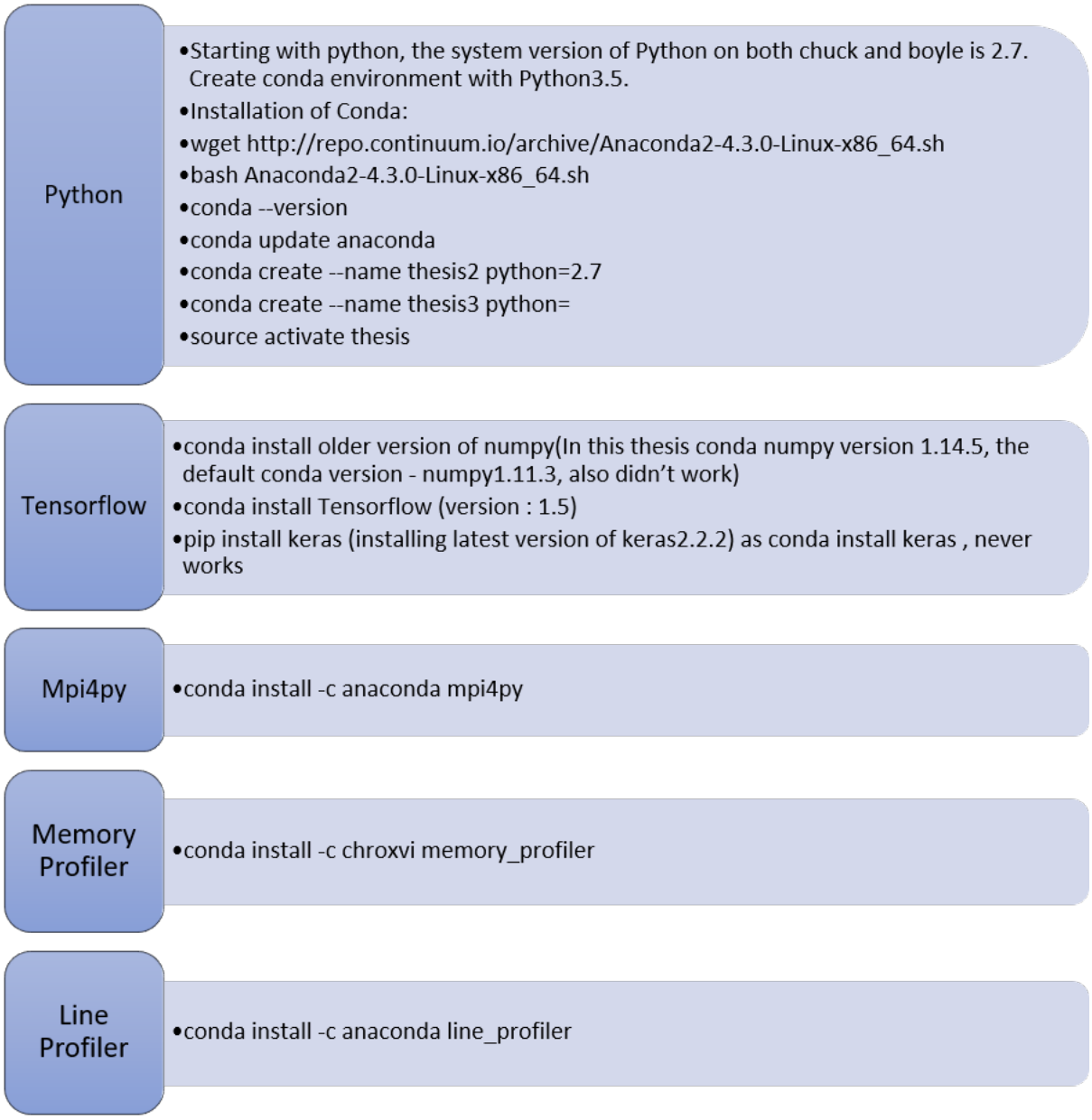

Figure 3.6: Steps to install libraries

1. start
    (a) Initialize Input
    (b) Define Input data
    (c) Define output data
2. Divide data into training and test data
    (a) In this Thesis, Cifar10 dataset is considered, with total 60k data points
    (b) test data = 10k data points (15
    (c) Training data=50k data points (85
3. Define layers, neuron units in each layer, activation function, optimizer and dropout
4. Fit the model
    (a) Feed Forward neural Network
5. Compile the model
6. Make total error = 0
7. Train the neural network
8. Evaluate the fitness of the network
9. Total error += Error for each output neuron
10. If total error < target error => Terminate, else
    (a) Continue Step 6

Pseudo Codes Sequential Algorithmic for Neural Network Tuning

1. Initialize population or network
2. Train each network
3. Evaluate the fitness of each network
4. Check the criteria
    (a) number of generations
    (b) Fitness achieved
5. If achieved then terminate, else
    (a) Breed Networks
    (b) Train each network

(c) Evaluate fitness of each network

(d) Select the best networks for next generations

(e) Continue Step 4

Pseudo Codes Parallel Algorithm for Neural Network Tuning

1. Initialize population or network (one per processor)
2. Train network on each processor
3. Evaluate the fitness of each network on the processors
4. Check the criteria
   (a) number of generations
   (b) Fitness achieved
5. If achieved then terminate, else
   (a) Breed Networks
      i. Crossover (take network from previous processor for crossover)
      ii. Mutation (Chance of mutation is considered 2
   (b) Train each network
   (c) Evaluate fitness of each network
   (d) Select the best networks for next generations
   (e) Continue Step 4

Pseudo Codes Parallel Island Model based Algorithm for Neural Network Tuning

1. Initialize islands
2. Initialize island population
3. Train island networks
4. Evaluate the fitness of each network on the processors
5. Check the criteria
   (a) number of generations
   (b) Fitness achieved
6. If achieved then terminate, else

(a) Breed Networks

i. Crossover (take network from previous processor within the island for crossover)

ii. Mutation (Chance of mutation is considered 2

(b) Train each network

(c) Evaluate fitness of each network

(d) Select the best intra-island network for next generations

(e) Continue Step 4

7. If generation number is a multiple of 10, then

(a) Breed Networks

i. Inter-island Crossover

ii. Mutation (Chance of mutation is considered 2

(b) Train each network

(c) Evaluate fitness of each network

(d) Select the best networks for next generations

(e) Continue Step 4

Pseudo Codes for Quantum Inspired Algorithm for Neural Network Tuning

1. Initialize quantum population vector Q
2. Train each network in the Q vector
3. Evaluate the fitness of each network in the Q vector
4. Check the criteria

(a) number of generations

(b) Fitness achieved

5. If achieved then terminate, else

(a) Breed Networks (Quantum Crossover using superposition)

(b) Train each network

(c) Evaluate fitness of each network

(d) Select the best networks for next generations

(e) Continue Step 4

Github Code repository Links:

Neural Network Optimization Github Repository

3.4 Sequential Concern

SSE Speed Limitations

Careful observation in the output file suggests that if TensorFlow is built from source it can be faster on the machine. The error is as follows: "The TensorFlow library wasn't compiled to use SSE instructions, but these are available on your machine and could speed up CPU computations" And it requires bazel build, which has gcc build constraints

Training Speed Constraint

In this thesis, optimization of Neural Network parameters requires faster computation of the evaluation function. Since with generations during optimization, multiple populations are evaluated and replaced on a generational basis, the algorithm poses memory and time constraints. As we have seen in the above cluster comparison, Boyle has limited nodes and using only one OMP THREAD drags the whole training process of one genome to an hour. If one evaluation takes 1 hour, then for over 1000 population, evaluations can be deadening to the cluster. In this thesis, generations are constrained to the limit of 30

4 IMPLEMENTATION AND DATA ANALYSIS

Exploration vs Exploitation based Optimization (EETO) Method has been implemented in Sequential Code. The same has been parallelized using Mpi4py. Next, the population initialization method of EETO has been inspired using concepts of Quantum Mechanics. The implementation is discussed in detail in below subsections.

In all the methods, the search space has been kept constant. A group of 5 hyper-parameters has been considered: activation functions, optimizers, hidden layers, nodes in hidden layers and dropout. Six activation functions included are: sigmoid, elu, selu, relu, tanh, hard_sigmoid. In addition to this, optimizers include sgd, adagrad, adadelta, adam, adamax, nadam. Hidden layers range from 1-15. Nodes/neurons range from 4-128. Dropouts range from 0.1 to 0.5.

In all the methods, the networks are initialized using random sets of configuration variables. These networks then exchange hyper-parameters with each others(exploitation). The networks get the change to mutate @30%. This process is called as breeding. Based on fitness values, the network with least fitness value is killed/deleted and re-initialized (Exploration).

4.1 Genetics Inspired Exploration vs Exploitation based Optimization for Neural Network Tuning

4.1.1 Implementation

Several implementations of the same have been done using different number of processes (5 implementation using network size of 4, 6, 8, 10, 12 respectively)

Requirements:

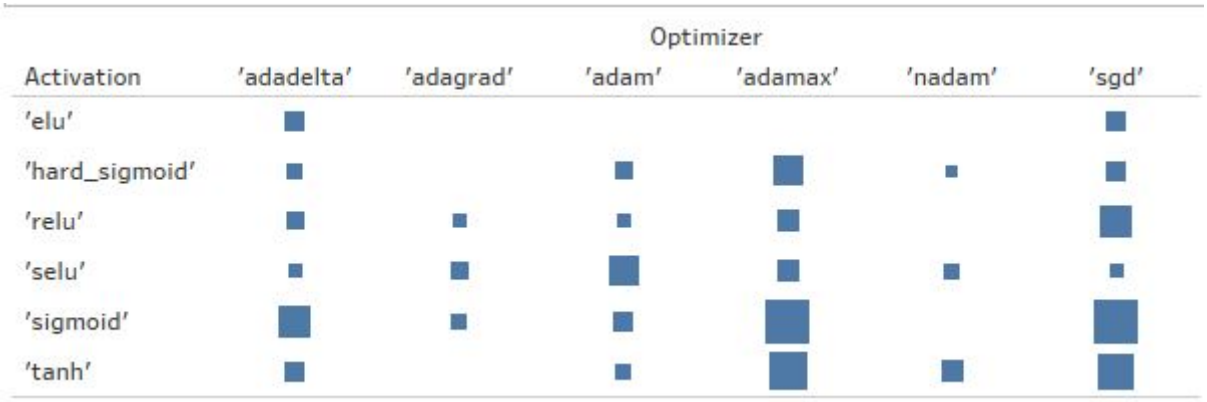

Figure 4.1: Cross tab of activation functions and optimizers tested

1. Server: Scientific Linux release 7.5 (Nitrogen)
2. conda 4.5.10
3. Python 2.7.15 :: Anaconda, Inc.
4. numpy 1.15.0
5. Keras 2.2.2
6. Tensorflow 1.5
7. gcc: gcc (GCC) 4.8.5 20150623 (Red Hat 4.8.5-28)
8. ld: ldd (GNU libc) 2.17

Usage:

```
python snnt.py
```

This saves the output in a log file.

4.1.2 Analysis

The fitness values for various hyper-parameters are analyzed

Number of networks initialized = 4, Processors used = 1

A crossstab of the tested combinations of hyper-parameters is shown below. It can be easily seen from Figure 4.1 that while evaluating hyper-parameter combinations for training out of 6 activation functions, 'hard_sigmoid' and 'elu' could not be tested with optimizer 'adagrad', implying that not all the functions could be tested

Next, a heat map of hyper-parameters' set and the fitness thusly achieved has been created as shown in Figure 4.2. From this it can be concluded that fitness is highly related

HyperParameters	
'elu','adadelta',6,4,0.40	17.7
'elu','adadelta',6,128,0.10	43.5
'elu','sgd',1,64,0.10	41.0
'elu','sgd',6,16,0.5	18.2
'hard_sigmoid','adadelta',1,64,0.40	40.2
'hard_sigmoid','adam',1,64,0.25	42.7
'hard_sigmoid','adam',6,64,0.40	10.0
'hard_sigmoid','adamax',1,4,0.20	30.3
'hard_sigmoid','adamax',1,4,0.25	27.0
'hard_sigmoid','adamax',1,16,0.20	37.7
'hard_sigmoid','adamax',1,128,0.40	44.0
'hard_sigmoid','nadam',15,64,0.10	10.0
'hard_sigmoid','nadam',15,64,0.40	10.0
'hard_sigmoid','sgd',1,4,0.40	23.3
'hard_sigmoid','sgd',1,64,0.20	38.2
'relu','adadelta',3,64,0.25	26.6
'relu','adadelta',12,64,0.10	29.7
'relu','adagrad',1,8,0.10	17.7
'relu','adagrad',1,8,0.20	15.2
'relu','adam',1,4,0.20	10.0
'relu','adam',1,8,0.20	10.0
'relu','adam',12,4,0.20	10.0
'relu','adamax',1,128,0.25	39.5
'relu','adamax',12,64,0.10	33.8
'relu','sgd',1,4,0.40	25.0
'relu','sgd',1,64,0.25	43.4
'relu','sgd',3,64,0.40	27.8
'relu','sgd',3,128,0.25	41.0
'relu','sgd',12,4,0.5	10.0

HyperParameters	
'relu','sgd',12,64,0.10	19.8
'selu','adadelta',3,16,0.40	33.3
'selu','adagrad',9,64,0.20	35.6
'selu','adagrad',15,128,0.25	20.1
'selu','adam',1,64,0.25	46.0
'selu','adam',1,128,0.25	47.5
'selu','adam',3,16,0.20	38.3
'selu','adam',12,128,0.40	17.0
'selu','adamax',1,4,0.10	33.5
'selu','adamax',1,128,0.10	46.9
'selu','nadam',1,16,0.25	38.9
'selu','sgd',1,64,0.25	35.3
'sigmoid','adadelta',1,16,0.10	37.3
'sigmoid','adadelta',1,64,0.10	43.3
'sigmoid','adadelta',1,128,0.25	44.5
'sigmoid','adadelta',3,8,0.40	18.7
'sigmoid','adadelta',6,128,0.25	10.0
'sigmoid','adagrad',1,64,0.10	41.2
'sigmoid','adam',1,8,0.40	28.1
'sigmoid','adam',3,128,0.25	40.2
'sigmoid','adamax',1,16,0.10	39.8
'sigmoid','adamax',1,64,0.10	45.4
'sigmoid','adamax',1,128,0.5	43.8
'sigmoid','adamax',1,128,0.10	48.1
'sigmoid','adamax',1,128,0.25	47.1
'sigmoid','adamax',1,128,0.40	46.2
'sigmoid','adamax',9,16,0.25	10.0
'sigmoid','adamax',9,128,0.25	20.9
'sigmoid','adamax',12,128,0.10	10.0

HyperParameters	
'sigmoid','adamax',12,128,0.25	10.0
'sigmoid','sgd',1,4,0.40	28.5
'sigmoid','sgd',1,16,0.40	32.2
'sigmoid','sgd',1,64,0.10	36.6
'sigmoid','sgd',1,64,0.25	37.8
'sigmoid','sgd',1,128,0.10	38.3
'sigmoid','sgd',1,128,0.25	37.8
'sigmoid','sgd',1,128,0.29	37.1
'sigmoid','sgd',1,128,0.40	37.5
'sigmoid','sgd',6,128,0.5	10.0
'tanh','adadelta',1,4,0.40	28.1
'tanh','adadelta',1,128,0.20	38.3
'tanh','adam',1,64,0.25	38.2
'tanh','adamax',1,16,0.20	35.6
'tanh','adamax',1,64,0.25	39.9
'tanh','adamax',1,64,0.40	40.9
'tanh','adamax',1,128,0.20	42.1
'tanh','adamax',6,64,0.25	35.8
'tanh','adamax',9,128,0.10	38.2
'tanh','nadam',1,16,0.25	29.1
'tanh','nadam',3,16,0.5	10.0
'tanh','nadam',3,128,0.10	32.4
'tanh','nadam',12,16,0.25	10.0
'tanh','sgd',1,8,0.25	33.7
'tanh','sgd',1,64,0.25	41.6
'tanh','sgd',1,128,0.10	40.5
'tanh','sgd',1,128,0.25	40.8
'tanh','sgd',3,64,0.25	39.1
'tanh','sgd',15,128,0.40	10.7

Figure 4.2: Sampled hyper-parameters' set;
Column 1 (from Left to Right): Activation function, Optimizer, Layers, Neurons, Dropout;
Column 2: Maximum Fitness for the hyper-parameter combination;

to the number of neurons.

A more detailed plot is as follows (Figure 4.3). The size of the squares denote the fitness. While increment in number of neurons increase the fitness, decrement in number of layers help improvise the fitness.

With 4 random networks initialization of best combination of hyper-parameters is highly unlikely, since total possible combinations of hyper-parameters is tens of thousands. However, if with the help of HPC, speed could be achieved then the same could be run for thousands of generations and with higher number of networks increasing the possibility of a better accuracy network. To test the same, networks of size 6, 8, 10 and 12 have been run in sequential and parallel. The next section discussed the effectiveness of the same.

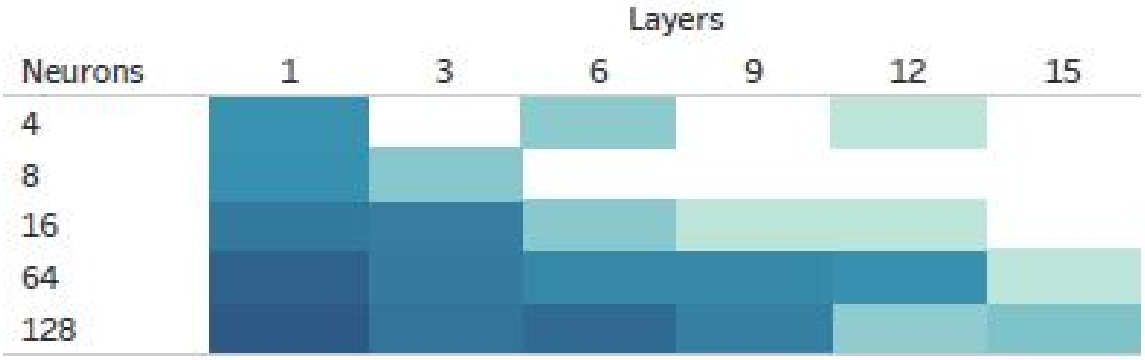

Figure 4.3: Fitness Heat Map - neurons va layers

4.2 PGA Inspired Exploration vs Exploitation based Optimization for Neural Network Tuning

4.2.1 Implementation

As discussed in the previous section, two parallel software implementations of the algorithm are employed for development. Keeping everything constant, the implementations differ in the arrangement of the networks. Exploring the same space, both the implementations use Mpi4py for parallelism. Since the networks being evaluated at processors are not equal in size, MPI Barrier is used.

1. In Single Population Fine Grained Algorithms (SPFGA), the serial code is run using multiple processors. Every processor initiates one network. For crossover, network of the previous rank is taken. The bred child is mutated considering 30% mutation chance. In this analysis, multiple codes are run using 4, 6, 8, 10, 12 processors.

   In this method, a send and receive model of GA is used. For each node, one network is initiated. Assuming more nodes ensures increased input combinations, will result in increased possibility of getting desired accuracy in the initial stages. Safety flags like are used to ensure MPI initialization has been successful. Breeding of one child per processor occurs. Considering processors structured in circular fashion, Crossover takes network from previous processor using MPI Non-Blocking Communication.

2. Multiple-population Coarse Grained Algorithms (MPCGA) model uses the same concept as stated above. However, multiples island are initialized with population in each island. Two types of breeding takes place, inter-island(after every 5 generations) and intra-island. In this, population size of 4 in each island with 3 islands and total of 12 processes is run. Multiple islands of population are created viz. multiple MPI

communicators for sets of networks are created.

Using the same hypothesis as previously used, "assuming more nodes ensures increased input combinations, will result in increased possibility of getting desired accuracy in the initial stages", for each node, one network is initiated. These networks are then split into communicators. Breeding of one child per processor occurs, however crossover is intra-communicator, implying that same circular fashion of processors for every communicator is assumed and the processors are mated with the previous processors. After every "x"(in this thesis, x is kept 5) number of generations, inter-communicator crossover occurs in which best of every communicator/island is computed at every node using MPI gather and MPI scatter commands and is sent to the other islands. This process continues till either required accuracy is achieved or the limit for the number of generations have been crossed.

Requirements:

1. Server: Scientific Linux release 7.5 (Nitrogen)
2. conda 4.5.10
3. Python 2.7.15 :: Anaconda, Inc.
4. numpy 1.15.0
5. Keras 2.2.2
6. Tensorflow 1.5
7. gcc: gcc (GCC) 4.8.5 20150623 (Red Hat 4.8.5-28)
8. ld: ldd (GNU libc) 2.17
9. Mpi4py 2.0.0

Usage:

```
mpiexec -n 7 python dnnt.py
mpiexec -n 7 python innt.py
```

This saves the output in log files. The code can be run in accordance with the availability of cluster nodes

4.2.2 Comparative Analysis

A comparison of fitness evolution with generations is drawn for various ranks as shown below. Starting with analysis on code using 4 networks initialized on 4 processors in parallel. The below image (Figure 4.4) shows that there is very slight difference in the fitness

between two methods. Figure 4.4 shows the fitness achieved by various implementations. The graphs in the left hand side are for sequential method, the graphs in the right hand side are the parallel method. Both the methods are run for various network sizes (4, 6, 8, 10, 12). The graph shows the line chart for fitness conversion with respect to the number of generations passed. In both sequential and parallel implementations, the fitnesses are converging to a value of 50% However, the accuracy that serial code taken hours to achieve can be gained in minutes time running parallel code.

This shows that increasing the number of ranks/networks has no impact on fitness, however, increasing the ranks for the analysis does help in getting the fitness sooner.

Figure 4.5 shows the area graph of the time taken by various implementations. The presentation follows the same structure as the fitness graph in Figure 4.4. Comparing the same with graphs plotted for the serial implementation, it can be easily concluded that time taken to train each network in parallel is less than the time taken while training in sequential. The reason behind such behavior is division and ease of access of RAM for each network. Keras add sessions everytime a model is trained. The reduction of time at generation level is due to division of tasks in parallel reducing the amount of time taken as per “Law of division of labor”

In sequential graph, the time taken increases with generations. In the beginning it was believed that this is due to memory constraints as the python library that is used for optimization "keras", saves the models and the hyper-parameters, parameters and other required variables for further cross-checking.

Island Model

The same is tested creating Island model. Three islands of size 4 each have been created. Below is the graph (Figure 4.6) at generation level, comparing the time taken by the Island model implementation vs the time taken by the sequential and the parallel implementations to achieve the same fitness. One of the Islands Island group 0 out of the three Islands (Island Group 0, 1 and 2) have been plotted against Sequential and parallel with network size of 4 each.

From the above table it can be inferred that implementation of the island model resulted is fruitful in terms of time taken.

However, the graph of the sequential implementation is alarming as the time taken is increasing with generations.

Memory Limits

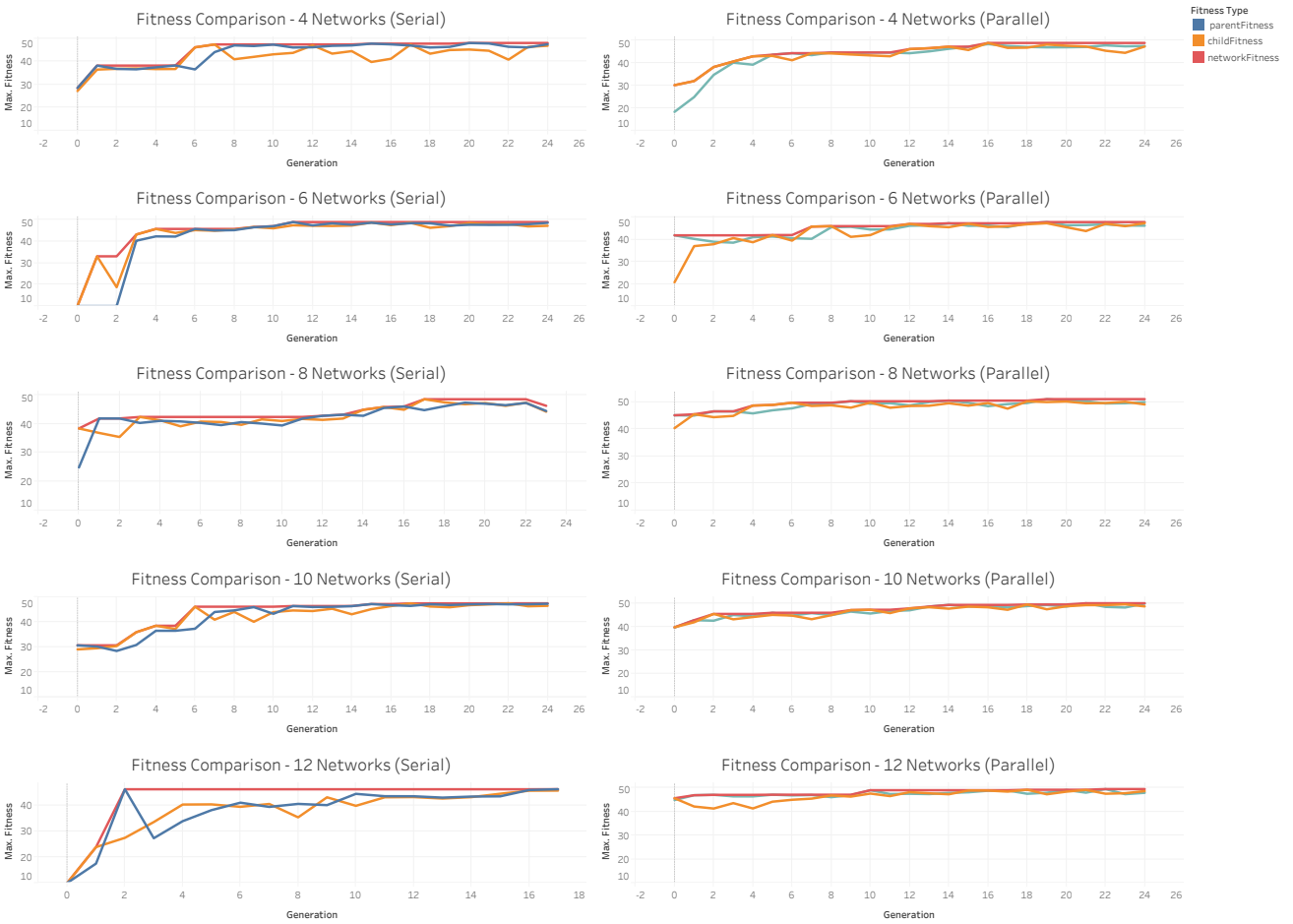

Figure 4.4: Fitness comparison - Sequential vs Parallel Implementation;
Left Hand Side: Fitness results for Sequential Implementation with different network sizes;
Right Hand Side: Fitness results for Parallel Implementation with different network sizes;
Top to bottom: Fitness evaluation graphs for Network sizes 4, 6, 8, 10, 12;
Fitness are being captured at generation level;

Figure 4.5: Time comparison - Sequential vs Parallel Implementation
Left Hand Side: Results for time taken in Sequential Implementation with different network sizes;
Right Hand Side: Results for time taken in Parallel Implementation with different network sizes;
Top to bottom: Graph depicting Time Taken for Network sizes 4, 6, 8, 10, 12;
Area graph for Time taken is shown at generation level;

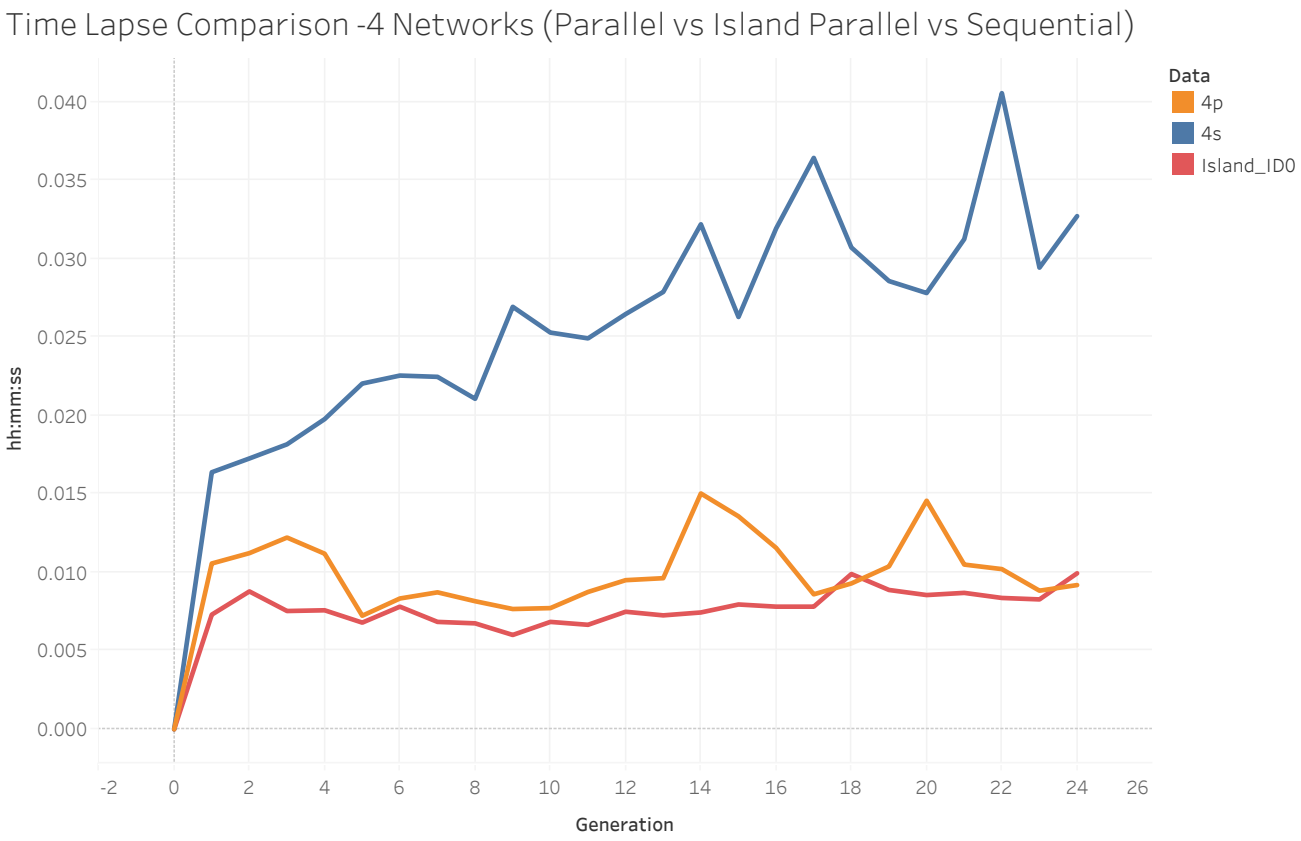

Figure 4.6: Time comparison - Sequential vs Parallel vs Island Implementation;
Legend Decode:
4s -> 4 Networks Genetics Inspired Algorithm Data;
4p -> 4 Networks Parallel Algorithm Data;
Island_ID0 -> 4 Networks Genetic Algorithm Island Algorithm Data using only one Island group with ID "0"

Furthermore, the efficiency and speed-up graphs are plotted as shown in Figure 4.7.

This graph shows the super-linearity of the parallel implementation. So, an analysis on memory leak was done.

A sequential code with networks size 4 was run and snapshots of the objects released were captured with time difference of barely some minutes. Some object types with greater increments are shown in Figure 4.8.

Despite the fact that python comes with it's own garbage collector, this was a result of the circular reference issue, thus preventing the garbage collector from freeing the objects held in memory.The garbage collector works on the principle of reference counting algorithm. Reference cycles can occur in container objects (lists, dictionaries, classes, tuples). The re-referencing of tuples and dictionaries of dictionaries used while saving the hyper-parameter sets were left untracked.

From Figure 4.8, it can be easily concluded that the object type "dict", "tuple" are problematic To further validate the same, the total number of objects and the object sizes have been looked upon as shown in Figure 4.9

TCHPC-Boyle cluster has 32 compute nodes, providing memory/storage resources to virtual machine instances. In this thesis, most of the codes ran in boyle nodes 0-10 which have 16 cores.

Structure of Boyle cluster nodes				
Memory at node and core level for each partition				
Nodes	Cores	Partition	Memory	Memory per Node
13	28	compute	128000	4571
8	28	compute	256000	9143
10	16	compute	64000	4000
1	28	compute	128000	4571
20	240	mic	7697	32
1	28	long	256000	9143

While creating the neural network models every generation, the keras library stores the model creation session. Because of storing all these sessions, SLURM output shows memory error.

An analysis which was previously carried out: Parallel code using 28 cores was run on one of the 28core nodes. Memory usage of the code was more than 4.5G. Error:

“`slurmstepd: Exceeded step memory limit at some point.`”

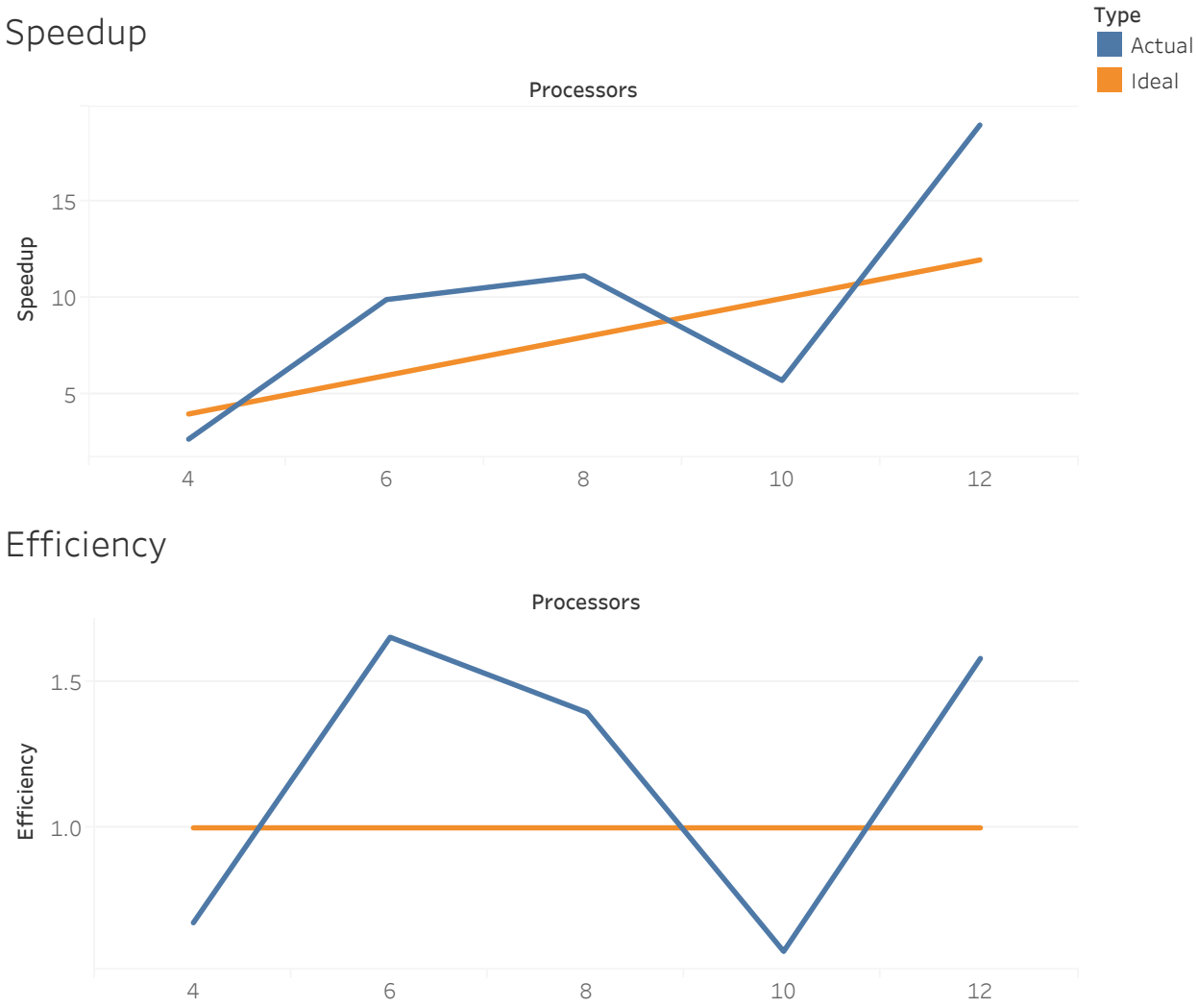

Figure 4.7: Speed-up and Efficiency

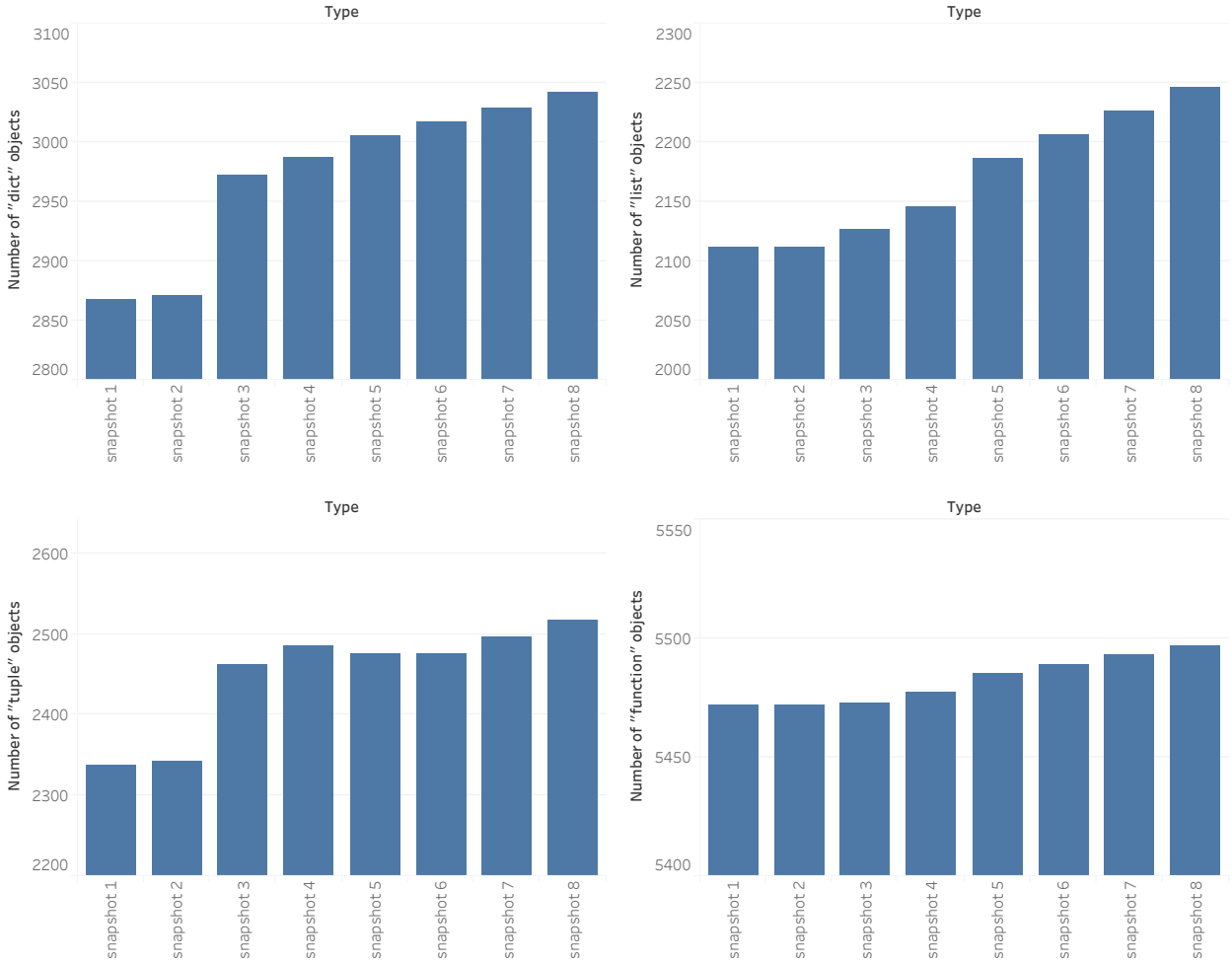

Figure 4.8: Memory leak Snapshots

types	Running for 1 generation		Running for 5 generations		Times increament in number of objects
	# objects	total size	# objects	total size	
<class 'numpy.ndarray	3	117.23 MB	3	117.23 MB	NA
<class 'tuple	62126	5.69 MB	482777	44.19 MB	7.8
<class 'list	44141	4.30 MB	250113	24.86 MB	5.7
<class 'dict	10997	2.52 MB	87719	20.02 MB	8.0
<class 'int	83692	2.23 MB	630694	16.84 MB	7.5
<class 'tensorflow.core.framework.node_def_pb2.NodeDef			28057	2.35 MB	Internally saved from first generation in keras, and then incremented with generations
<class 'tensorflow.python.framework.tensor_shape.Dimension			29688	1.59 MB	
<class 'tensorflow.python.framework.ops.Tensor			29401	1.57 MB	
<class 'tensorflow.python.framework.tensor_shape.TensorShape			29361	1.57 MB	
<class 'tensorflow.python.framework.ops.Operation			28057	1.50 MB	
<class 'str'	14656	1.05 MB	17258	1.24 MB	1.18

Figure 4.9: Memory leak with generations

The solution is to clear session after usage of every model. In addition to this, although python uses automatic garbage collectors, reclaiming memory rendered useless by the code, in the code hyper-parameters are given by dictionary of a dictionary of a dictionary, which is run for all networks over number of generations, so to avoid unnecessary filled memory, the variables should be deleted or replaced with "None" value.

4.3 Quantum Genetics Inspired Genetics Inspired Exploration vs Exploitation based Optimization for Neural Network Tuning

A comparison of fitness with evolving generations can be seen in Figure 4.10. The highest fitness achieved is 39.18% (10% lesser than what was achieved without Quantum Inspired Algorithm). Thus proving that initializing the networks with more randomized values did not turn out fruitful.

The graph shows the best fitness achieved for different network types at generation level. With 4 networks, the best fitness is not achieved even after 25 generations, but is continuously improving, thereby demanding more generations to be run. The graph proves that with increment in number of networks the the chances of getting better fitness in initial generations increases.

A quick comparison has been done taking 4 networks for the parallel implementation done in previous section vs the implementation done using Quantum Inspired Algorithm for time taken as well as the fitness evolution as shown in Figure 4.11

It can be easily inferred that time taken in Quantum Inspired Algorithm increases with generation as a result of memory leak. In case of parallel code, the time taken increases and reduces thereby contributing to more number of cores being used and increment in time is a result of heavier network (network demanding more computation because of hyper-paramenter selection) being trained.

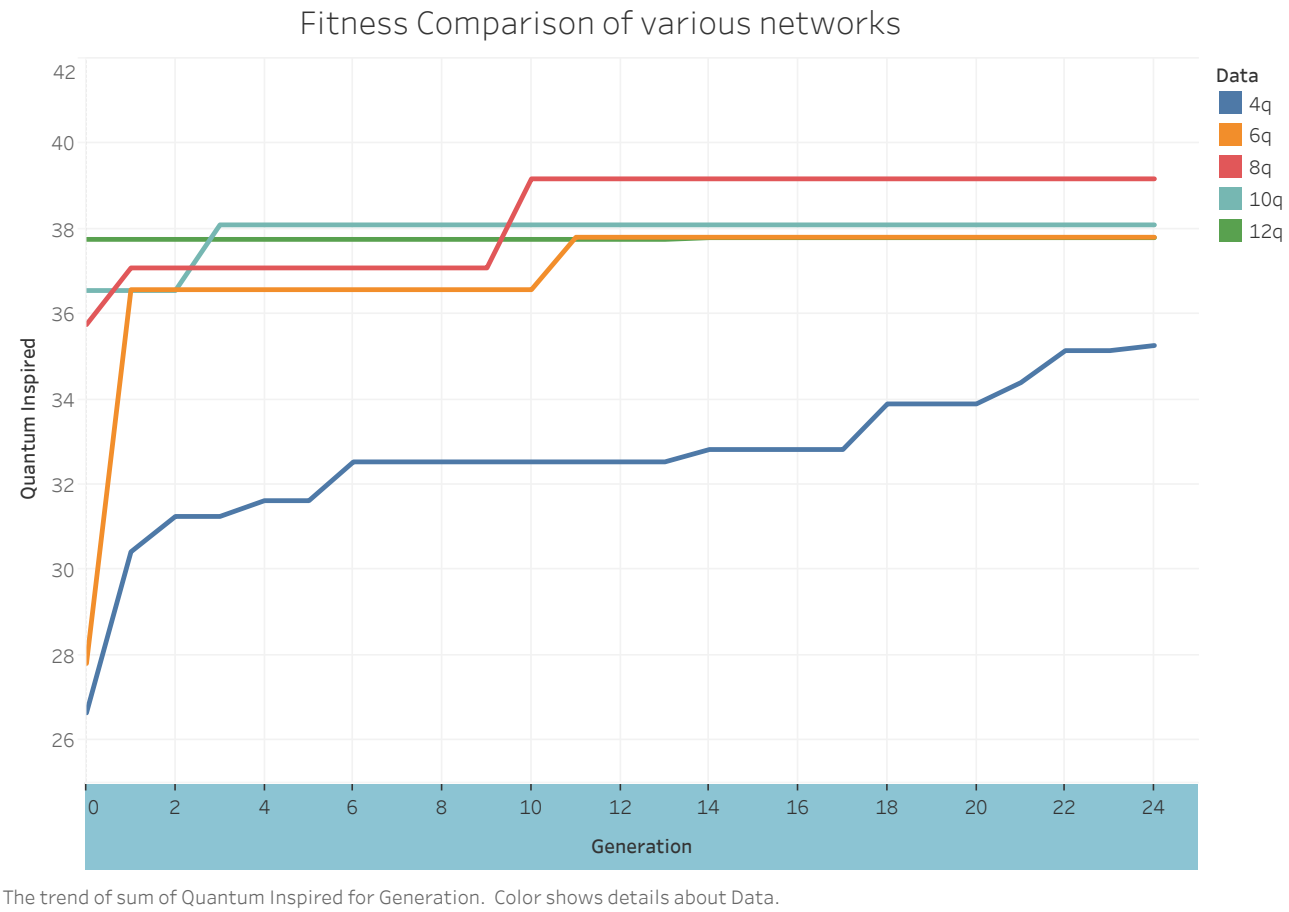

Figure 4.10: Best fitness achieved - for every generation for various network sizes;
Legend Decode:
4q -> 4 Networks Quantum Inspired Algorithm Data;
6q -> 6 Networks Quantum Inspired Algorithm Data;
8q -> 8 Networks Quantum Inspired Algorithm Data;
10q -> 10 Networks Quantum Inspired Algorithm Data;
12q -> 12 Networks Quantum Inspired Algorithm Data;

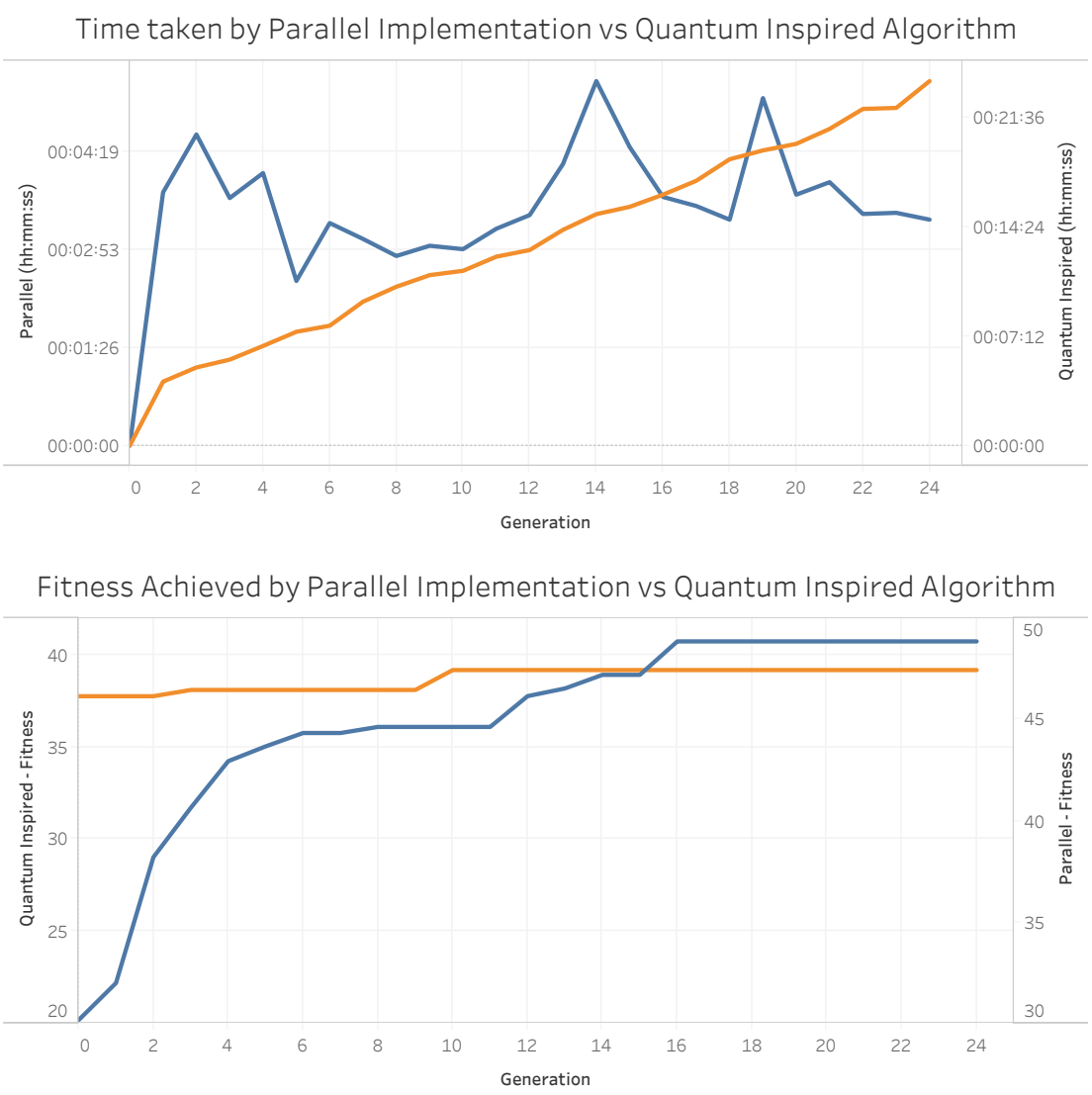

Figure 4.11: The graph shows time taken and fitness comparison taking 4 networks
a. the time taken in both Parallel vs Quantum Inspired Algorithm implementations
b. best fitness for every generation in both Parallel vs Quantum Inspired Algorithm implementations

5 Summary

In this thesis, Cifar10 image classification dataset is used. Four optimization methods have been implemented to get the best set of hyper-parameters, which are:

1. Sequential Genetic Inspired Algorithm
2. Single Population Fine Grained Algorithms
3. Multiple-population Coarse Grained Algorithms (Island Model)
4. Quantum Inspired Algorithm

These methods have been run and tested on various network sizes (4, 6, 8, 10 and 12).

Overall, highest efficiency fetched is 51.14% from parallel code giving best set of hyper-parameters as follows:

adamax (Optimizer) - elu (Activation function) - 128 nodes - 3 layers - 0.1 dropout

This accuracy is a result of parent fitness. Also, there is a minute difference between the efficiencies of parallel and serial methods with best efficiency achieved from running serial code to be 49.05 using hyper-parameter which similar to the overall best hyper-parameter set as:

adamax (Optimizer) - elu (Activation function) - 128 nodes - 9 layers - 0.1 dropout.

The difference is in the number of layers, which is another interesting result: the decrease in layers and the increase in neurons in hidden layers are two major factors resulting in higher frequencies.

There have been cases where child fitness is higher than it's parent's thereby confirming the chances of better exploration and getting out of the local maxima. However, the best

efficiencies from all the methodologies sequential (49.05%), parallel (51.14%) and quantum inspired (39.18%) code have been achieved as a result of parent fitness.

Also, networks despite same hyper-parameter configurations take different time in training resulting in different accuracy/efficiency. However, the difference in time and accuracy is not high and can be ignored (time difference 4-5 minutes in serial code, accuracy difference 1-2%)

Different sets of hyper-parameters took different time e.g. hyper-parameter set with larger number of nodes (say 128) resulted in increased time taken to train the network as compared to the set with smaller number of nodes(say 4).

Although, python supports garbage collection, the parallel code has shown super-linear speed-up. This is because of huge memory leak: 50MB in 5 generations. The same if computed for hundreds of generations will fail.

In all the algorithms, it was seen that crossover and mutation steps took seconds to complete, on the contrary, fitness evaluation using keras took several minutes to complete.

Furthermore, out of all the implementations parallel code is time and resource efficient (island model being better of the two parallel implementations). The Parallel Algorithms have resulted in decreased time taken for the computationally expensive neural network tuning.

Bibliography

[1] Klaus Schwab. The 4th industrial revolution. In *World Economic Forum. New York: Crown Business*, 2016.

[2] Richard Lippmann. An introduction to computing with neural nets. *IEEE Assp magazine*, 4(2):4–22, 1987.

[3] Lisandro D Dalcin, Rodrigo R Paz, Pablo A Kler, and Alejandro Cosimo. Parallel distributed computing using python. *Advances in Water Resources*, 34(9):1124–1139, 2011.

[4] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[5] John J Hopfield and David W Tank. "neural" computation of decisions in optimization problems. *Biological cybernetics*, 52(3):141–152, 1985.

[6] Anthony TC Goh. Back-propagation neural networks for modeling complex systems. *Artificial Intelligence in Engineering*, 9(3):143–151, 1995.

[7] Paul Werbos. Beyond regression:" new tools for prediction and analysis in the behavioral sciences. *Ph. D. dissertation, Harvard University*, 1974.

[8] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[9] Charles Darwin. *On the origin of species, 1859*. Routledge, 2004.

[10] Melanie Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[11] Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[12] Joanna Masel. Genetic drift. *Current Biology*, 21(20):R837–R838, 2011.

[13] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[14] Chih-Wei Hsu, Chih-Chung Chang, Chih-Jen Lin, et al. A practical guide to support vector classification. 2003.

[15] Davide Chicco. Ten quick tips for machine learning in computational biology. *BioData mining*, 10(1):35, 2017.

[16] Francisco J Solis and Roger J-B Wets. Minimization by random search techniques. *Mathematics of operations research*, 6(1):19–30, 1981.

[17] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[18] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, et al. Population based training of neural networks. *arXiv preprint arXiv:1711.09846*, 2017.

[19] Niranjan Srinivas, Andreas Krause, Sham M Kakade, and Matthias Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. *arXiv preprint arXiv:0912.3995*, 2009.

[20] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *IJCAI*, volume 15, pages 3460–8, 2015.

[21] Javier González, Zhenwen Dai, Philipp Hennig, and Neil Lawrence. Batch bayesian optimization via local penalization. In *Artificial Intelligence and Statistics*, pages 648–657, 2016.

[22] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, pages 2962–2970, 2015.

[23] Jan Larsen, Lars Kai Hansen, Claus Svarer, and M Ohlsson. Design and regularization of neural networks: the optimal use of a validation set. In *Neural Networks for Signal Processing [1996] VI. Proceedings of the 1996 IEEE Signal Processing Society Workshop*, pages 62–71. IEEE, 1996.

[24] Dougal Maclaurin, David Duvenaud, and Ryan Adams. Gradient-based hyperparameter optimization through reversible learning. In *International Conference on Machine Learning*, pages 2113–2122, 2015.

[25] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[26] Franco Varetto. Genetic algorithms applications in the analysis of insolvency risk. *Journal of Banking & Finance*, 22(10-11):1421–1439, 1998.

[27] Theodore W Manikas, Kaveh Ashenayi, and Roger L Wainwright. Genetic algorithms for autonomous robot navigation. *IEEE Instrumentation & Measurement Magazine*, 10 (6), 2007.

[28] M Gen and R Cheng. Genetic algorithms and engineering design wiley j new york google scholar. 1997.

[29] Joe Davison. devol. `https://github.com/joeddav/devol`, 2017.

[30] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.

[31] Randal S Olson, Ryan J Urbanowicz, Peter C Andrews, Nicole A Lavender, Jason H Moore, et al. Automating biomedical data science through tree-based pipeline optimization. In *European Conference on the Applications of Evolutionary Computation*, pages 123–137. Springer, 2016.

[32] Randal S Olson, Nathan Bartley, Ryan J Urbanowicz, and Jason H Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 485–492. ACM, 2016.

[33] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[34] Mihai Udrescu, Lucian Prodan, and Mircea Vlăduțiu. Implementing quantum genetic algorithms: a solution based on grover's algorithm. In *Proceedings of the 3rd Conference on Computing Frontiers*, pages 71–82. ACM, 2006.

[35] Ling Wang, Fang Tang, and Hao Wu. Hybrid genetic algorithm based on quantum computing for numerical optimization and parameter estimation. *Applied Mathematics and Computation*, 171(2):1141–1156, 2005.

[36] José Fernando Gonçalves, Jorge José de Magalhães Mendes, and Mauricio GC Resende. A hybrid genetic algorithm for the job shop scheduling problem. *European journal of operational research*, 167(1):77–95, 2005.

[37] Dong Hwa Kim, Ajith Abraham, and Jae Hoon Cho. A hybrid genetic algorithm and bacterial foraging approach for global optimization. *Information Sciences*, 177(18): 3918–3937, 2007.

[38] Yi-Tung Kao and Erwie Zahara. A hybrid genetic algorithm and particle swarm optimization for multimodal functions. *Applied Soft Computing*, 8(2):849–857, 2008.

[39] Erick Cantú-Paz. A survey of parallel genetic algorithms. *Calculateurs paralleles, reseaux et systems repartis*, 10(2):141–171, 1998.

[40] Richard Friedman. Intel releases optimized python for hpc, Feb. 2017.

[41] How fast is pypy? `http://speed.pypy.org/`. Accessed: 2018-09-07.

[42] Pyrex documentation. `https://wiki.python.org/moin/Pyrex`, . Accessed: 2018-09-07.

[43] Limitations. `https://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/version/Doc/Manual/Limitations.html`, . Accessed: 2018-09-07.

[44] Python 3.4.9 documentation, the python standard library, concurrent execution, multiprocessing — process-based parallelism. `https://docs.python.org/3.4/library/`, Aug. 2018.

[45] Lisandro Dalcın, Rodrigo Paz, Mario Storti, and Jorge D'Elıa. Mpi for python: Performance improvements and mpi-2 extensions. *Journal of Parallel and Distributed Computing*, 68(5):655–662, 2008.

[46] François Chollet et al. Keras. `https://keras.io`, 2015.

[47] George S Almasi and Allan Gottlieb. Highly parallel computing. 1988.

[48] Wen-mei Hwu. What is ahead for parallel computing. *Journal of Parallel and Distributed Computing*, 74(7):2574–2581, 2014.

[49] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing: design and analysis of algorithms*, volume 400. Benjamin/Cummings Redwood City, 1994.

[50] Jian-Ming Li, Xiao-Jing Wang, Rong-Sheng He, and Zhong-Xian Chi. An efficient fine-grained parallel genetic algorithm based on gpu-accelerated. In *Network and parallel computing workshops, 2007. NPC workshops. IFIP international conference on*, pages 855–862. IEEE, 2007.

[51] AL Marakeby. Analysis of md5 algorithm safety against hardware implementation of brute force attack international journal of advanced research. *Computer and Communication Engineering*, 2(9):3332–3335, 2013.

[52] Mohamed Wahib, Asim Munawar, Masaharu Munetomo, and Kiyoshi Akama. Optimization of parallel genetic algorithms for nvidia gpus. In *Evolutionary Computation (CEC), 2011 IEEE Congress on*, pages 803–811. IEEE, 2011.

[53] W. Liu and B. Schmidt. Parallel pattern-based systems for computational biology: A case study. *IEEE Transactions on Parallel & Distributed Systems*, 17:750–763, 08 2006. ISSN 1045-9219. doi: $10.1109/\mathrm{TPDS}.2006.109$. URL doi.ieeecomputersociety.org/10.1109/TPDS.2006.109.

[54] Research it, cluster. https://www.tchpc.tcd.ie/resources/clusters. Accessed: 2018-09-07.