CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A Version Oriented Parallel Asynchronous Evolution Strategy for Deep Learning

Master's thesis in Computer Science and Engineering

MYEONG-JIN JANG

Department of Computer Science and Engineering
THE CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

A Version Oriented Parallel Asynchronous Evolution Strategy for Deep Learning

MYEONG-JIN JANG

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

MYEONG-JIN JANG

Supervisor: Philippas Tsigas, Department of Computer Science and Engineering
Examiner: Marina Papatriantafilou, Department of Computer Science and Engineering

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in $\LaTeX$
Gothenburg, Sweden 2021

MYEONG-JIN JANG
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

In this work we propose a new parallel asynchronous Evolution Strategy (ES) that outperforms the existing ESs, including the canonical ES and steady-state ES. ES has been considered a competitive alternative solution for optimizing neural networks in deep reinforcement learning, instead of using an optimizer and a back-propagation function. In this thesis, three different ES systems were implemented to compare the performances of each ES implementation. Two ES systems were implemented based on existing ES systems, which are the canonical ES and steady-steady ES, respectively. Lastly, the last ES system is the proposed ES system called Version Oriented Parallel Asynchronous Evolution Strategy (VOPAES). The canonical ES replaces all population individuals at each generation, whereas the steady-state ES replaces only the weakest population with the newly created one. By replacing all population individuals, the canonical ES could optimize the network faster than the steady-state ES. However, it requires synchronization which might increase CPU idle time. On the contrary, a parallel steady-state ES does not require synchronization, but its learning speed could be slower than the parallel canonical ES one. Therefore, we suggest VOPAES as an advanced ES solution that takes the benefits of both the parallel canonical ES and the parallel steady-state ES system. The test results of this work demonstrated that the canonical ES system can be implemented asynchronously using versions. Moreover, by merging the benefits, VOPAES could decrease CPU idle time and maintain high optimization accuracy and speed as the parallel canonical ES system. In conclusion, VOPAES achieved the fastest training speed among the implemented ES systems.

Keywords: Reinforcement Learning, Parallelism, Evolution Strategy, Back-propagation, Asynchronous, Optimization

Acknowledgements

I believe that there is no self-made man in the world. To this point, I am genuinely grateful that I could meet my supervisor and examiner through this course resulting in so many meaningful lessons.

Myeong-jin Jang, Gothenburg, October 2021

Contents

List of Figures

List of Tables

1 Introduction

Long optimization time in the training of deep learning is one of the challenging aspects in the development process of deep learning. One of the reasons for the long optimization time in the training is that the optimizing process generally works serially rather than parallel. Since, generally, an optimizer optimizes a neural network by using multiple functions including a back-propagation and a back-propagation function calculates the gradient at each neural network layer serially from the last layer to the first layer. Therefore, optimizing neural networks with a back-propagation function can be less scalable when the neural network becomes more extensive and has more layers (more parameters) [1]. In this case, using a scalable optimizing method could be a viable solution to reduce the long optimization time in the training.

Evolution Strategy (ES) could be a competitive optimization solution for deep learning due to the reason that ESs could be implemented as parallel without additional modifications [2, 3, 4]. ESs are optimization techniques that were first developed by P.Bienert, I.Rechenberg, and H.P.Schwefel in the mid-1960s [5] and the details of these techniques will be discussed in Section 2.3. Since we found ESs are interesting alternative optimization methods, we decided to explore various existing ES systems and improve them.

We researched previous parallel ES-related studies at the beginning of this work and found several related ones. One [6] of the studies that we found was that Mario Srouji and Karan Dhabalia conducted at the Department of Electrical and Computer Engineering of Carnegie Mellon University [6]. In their study [6], only several parallel synchronous ES implementations were dealt with. Since we found asynchronous solutions could be more efficient than synchronous solutions, we investigated further in other exiting asynchronous ES, and steady-state ES was found [5]. Therefore, we firstly decided to implement both canonical parallel synchronous ES and the steady-state asynchronous parallel ES as reference ES systems.

During the initial tests, we found that the benefits of the parallel synchronous canonical ES system and the parallel asynchronous steady-state ES system lie in different places. On the one hand, the parallel synchronous ES system could more accurately optimize neural networks than the parallel asynchronous steady-state ES system. However, the parallel synchronous canonical ES system increases CPU idle time [2, 7, 8, 9]. On the contrary, the parallel asynchronous steady-state ES could reduce CPU idle time, nevertheless its optimization speed is slower than the parallel synchronous canonical ES system [9].

Since, the parallel canonical ES system replaces all the current networks with the newly created ones, whereas the parallel steady-state ES system only replaces the weakest network with the newly created one. Therefore, the parallel canonical ES system could more accurately optimize a network than the parallel steady-state ES. However, synchronization is necessary to update every network without creating generation gaps, which decrease the optimization robustness [9] and the synchronization process rises the CPU idle time. On the contrary, a parallel steady-state ES could work asynchronously without creating generation gaps [9, 5] because generation does not exist in a steady-state ES. However, steady-state ES could not optimize accurately as the parallel synchronous canonical ES systems, although it could increase the CPU utilization.

As a direction to improve the existing ES systems, we investigated how to make the parallel canonical ES system work asynchronously. By doing that, we considered it is possible to take all benefits from the parallel synchronous canonical ES and the parallel asynchronous steady-state ES. To merge the benefits of both ES systems, we used multiple versions that show which generation the combined neural networks belong to. Consequently, these versions enable the parallel canonical ES to work asynchronously without creating generation gaps, since generation gaps can be prevented by referencing several versions in the recombination. For instance, if different generations coexist in the networks of the current elites, the networks showing different versions from the first elite's version could be excluded. Thus, using versions can efficiently prevent occurring generation gaps. We named this ES system VOPAES, and, in conclusion, VOPAES succeeded in decreasing the training time and increasing CPU utilization by reducing each worker's waiting time (CPU Idle time). The test results and their detailed implementation will be discussed and analyzed in Sections 4 and 3, respectively.

1.1 Goals

This study aimed to improve the learning speed of ES by studying existing parallel ES implementations, which are the canonical ES and steady-state ES. Thus, the training durations of various parallel ES systems were compared in this study.

1.2 Metrics of Interest

In machine learning, there are three learning phases, which are training, validation, and test. In this work, only training-related metrics were dealt with, because the purpose of this study was to improve the existing ESs and they are optimization methods in the training (optimizing) phase alone. Therefore, training duration is the sole evaluation metric. Training duration is the period it takes from the start of the training until the termination criterion is met and a reward percentage is the termination criterion in this work. A reward percentage is a percentage of the accumulated rewards during the whole episode in the training data to be a good evaluation metric for optimizing the neural network. In this study, if one prediction (action) is correct, then one reward will be provided. On the contrary, if the action is incorrect, then zero will be given. If the training data contains stock prices for 100 days and an agent could precisely predict 50 days, then 50 percent will be the reward percentage from the training. In other words, if an agent tends to achieve higher reward percentages during the training, it indicates that the agent is learning properly. Therefore, reward percentages are reasonable metrics to evaluate the training speeds of the ES systems.

1.3 Challenges

The parallel synchronous canonical ES system showed several strengths compared to the parallel asynchronous steady-state ES system. For instance, its training speed was more rapid than the parallel asynchronous steady-state ES. However, its synchronization tended to increase CPU idle time compared to the asynchronous steady-state system. Because in synchronization systems, the threads that already finished their task should wait until the last thread completes its assigned task. Unfortunately, synchronization in a parallel canonical ES is necessary, although it increases CPU idle time. Because otherwise, generation gaps could occur [9, 5], and generation gaps decrease the optimization robustness [9]. Thus, a canonical ES system is generally implemented as a parallel synchronous system. Hence, we set the challenge of this work as enabling the parallel canonical ES to work asynchronously with high optimization robustness.

1.4 Motivation

The motivation of this work was to enable the canonical ES system to work asynchronously but prevent generation gaps at the same time by implementing the recombination in a coordinated fashion. By doing so, maintaining the training speed and decreasing CPU idle time could be achievable. As a solution, we used versions that show in which generation each combined neural network is to prevent generation gaps in the recombination phase when the parallel canonical ES system works asynchronously. As a result, the ES implementation could work asynchronously and could succeed in preventing generation gaps.

1.5 Related Work

There have been multiple previous studies about ESs in deep reinforcement learning. Recent research [6] that Mario Srouji and Karan Dhabalia studied showed how parallel synchronous ES implementations could be implemented and compared their differences. Some studies also support that ES can be a competitive alternative solution for an optimizer and a back-propagation function in deep reinforcement learning. For instance, Salimans et al. proved that using 1440 parallel cores with a simple ES algorithm could solve MuJoCo humanoid in 10 minutes [2]. Also, Kyunhyun Lee et al. in 2021 supported that ES could increase the deep reinforcement learning speed by enabling concurrent execution [7]. Alfredo V. Clemente et al. studied efficient parallel ES implementations and significantly reduced training time [10]. Other studies studied asynchronous ES. Eric O. Scott et al. showed that asynchronous parallel ES implementations could decrease the CPU idle time than synchronous parallel ES implementations [8]. Dario Izzo et al. also studied a similar asynchronous ES algorithm called an asynchronous island model [11]. Different algorithmic mechanisms of ES were studied by Nils Müller et al. [4]. Patryk Chrabaszcz et al. benchmarked the canonical ES for Playing Atari [3]. Matthew A. Martin et al. dealt with asynchronous parallel evolutionary algorithms [12].

2 Background

There are three subsections in this chapter. Firstly, the basic concepts regarding deep learning including what neural networks are, how its learning or training process works, and how back-propagation functions work are described. The second section is about ESs, including the two existing popular ES methods and some important terminologies. Lastly, the final section describes the general information of the parallel programming.

2.1 Deep Learning

Deep learning is one of the subareas of artificial intelligence. It requires a neural network to extract features from the given input data. There are several learning methods, including supervised learning, unsupervised learning, and reinforcement learning. Particularly, reinforcement learning was used in this work. In the subsections written below, the basic terminologies of deep learning including neural networks, optimization (learning), and back-propagation functions are explained. Knowing about regression analysis, chain rule, cost function, gradient descent, and stochastic gradient descent is helpful to understand how back-propagation functions are utilized in the optimization process of deep learning and why the back-propagation works serially.

2.1.1 Neural Networks

Neural networks are computing systems inspired by the animals' neural networks in their brains. Neural networks consist of multiple interconnected neurons and each neuron contains a weight and bias. These weights and biases are one of the parameters that determine the output of the network. When the input data is fed into the first (input) layer, the input data is processed until the data reaches the last (output) layer. In this process, the weights and biases in the network vitally affect the outcome. A simple diagram for helping readers to understand what a neural network is shown in Figure 2.1.

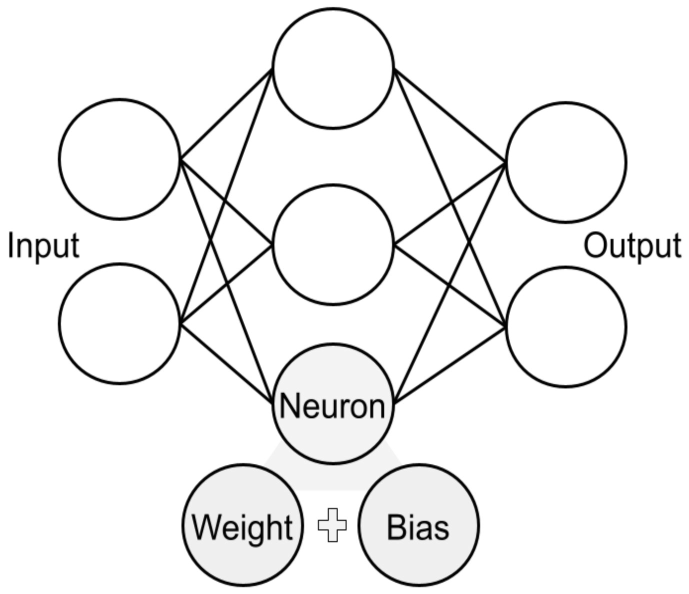

Figure 2.1: Simple neural network diagram: This diagram depicts a neural network consisting of multiple interconnected neurons. Each neuron has a its own weight and bias and all the weights and biases in a network are parameters that determine the output values. Input in the diagram indicates the input layer, while output indicates the output layer.

2.1.2 Optimization or Learning

The optimization or learning process in deep learning is to find a proper set of network parameters that produce the desired output. When the input data is fed into the first layer of the network, the network processes the data through the neurons at each layer until the output layer. When the output comes out at the output layer, it is possible to calculate the difference between the desired output and the output calculated via the network. This difference is called a loss and a loss function is used to calculate losses. If the parameters of the network are highly optimized, then the loss tends to be small. On the contrary, if the parameters of the network are less optimized, then the loss becomes greater. Therefore, optimizing the parameters according to the calculated losses is the learning process in deep learning and it is based on a regression analysis. This process is depicted in Figure 2.2.
An optimizer with a back-propagation is utilized to optimize the parameters of a neural network. An optimizer is an algorithm that uses a loss function to calculate losses, determining the rate of learning and back-propagation. The back-propagation is a tool to iteratively calculate the gradient of each layer [1, 13] by the chain rule. If the learning rate is set high, then this change can be relatively radical. If it is low, then the changes can be relatively slow.

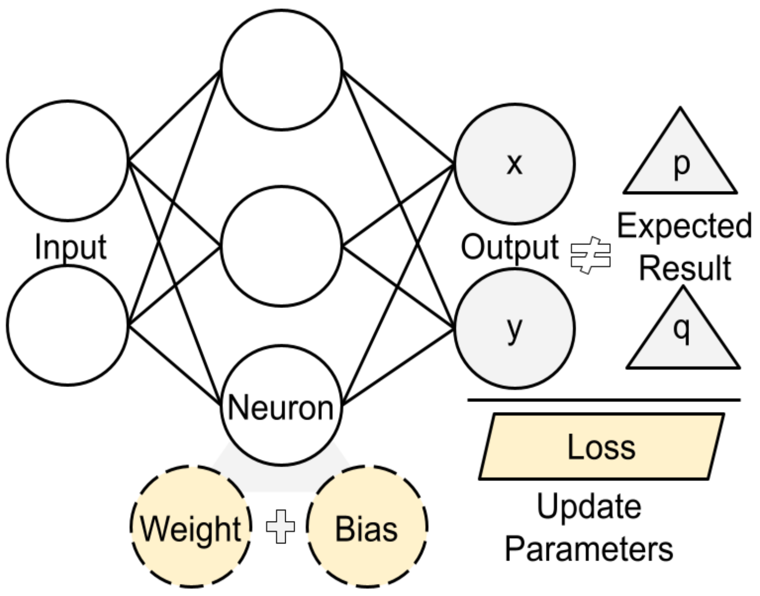

Figure 2.2: Simple neural network learning diagram: This diagram describes how a loss is calculated. When the input comes to the first layer (input layer), the data is processed from the first layer to the last layer, which is the output layer. In this example, the output is depicted as x and y in the last layer and the expected results are described as p and q, which are different from the output. These differences are calculated as a loss. This loss will be utilized to optimize the parameters of the network by the optimizer.

2.1.2.1 Regression Analysis

Regression analysis is a statistical method to find the relationship between a dependent variable and multiple independent variables and regression analysis is utilized in the learning process of deep learning. In a mathematical function, a dependent variable could be the output whereas an independent variable could be the input. Regression is generally utilized in machine learning to check how well a machine can predict compared to the expected output. One standard regression analysis is linear regression that finds the closest line to the output. An example of linear regression is depicted in Figure 2.3.

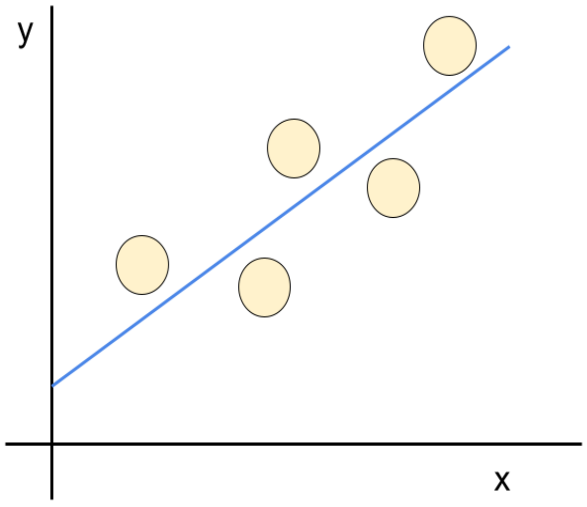

Figure 2.3: Example of linear regression: Dependent variable y and independent variable x are shown, and the yellow dots are the output. The blue line is located the closest distance from the yellow dots, and finding this linear line is the linear regression of regression analysis.

2.1.2.2 Chain Rule

The chain rule is a formula to calculate the derivative of a composite function. When a composite function consists of multiple differentiable functions, then the relationship between the first input independent variable and the output of the composite function can be calculated by multiplying the sequential derivatives that are linked directly, from back to forward. For example, if $f(x) = y, g(y) = t, f(g(y)) = z(t)$, then the derivative of the output value with respect to a change in x can be calculated as

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

. This chain rule is utilized in a back-propagation function to calculate the gradients from the last layer to the first layer serially.

2.1.2.3 Cost Function or Loss function

A cost function is to calculate how the expected results and actual values are different. In optimization, decreasing the cost function is the ultimate direction of the optimization.

2.1.2.4 Gradient Descent

Gradient descent is an iterative optimization method to find a local minimum point of the loss function by using the first derivative instead of calculating all output values of the cost function. Thus, it can find a local minimum value of a cost function efficiently. Some Back-propagation functions use gradient descent to optimize a neural network efficiently.
If the predicted value consists of multiple variables, then each variable's derivative can be calculated, and they are called gradients. An example is shown in Figure 2.4.

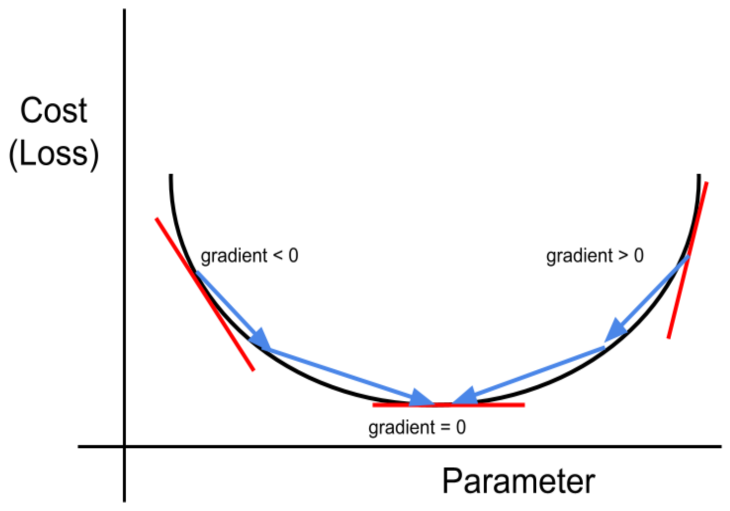

Figure 2.4: Gradient descent example: This figure shows how gradient descent works. The curved line is the output of a cost function, and the first derivative is the gradient. Using the first derivative, it tends to find the point that the gradient is close to zero as much as possible. When the gradient is negative, then the parameter can be increased. On the contrary, if the gradient is positive, then the parameter can be decreased.

2.1.2.5 Stochastic Gradient Descent

It is an iterative method for optimization by calculating the stochastic gradient instead of the sum of the gradients calculated from the entire data. It iteratively calculates a mini-batch input data and optimizes the object. Therefore, it reduces the computational burden, although its convergence rate can be lower. Stochastic Gradient Descent is used for optimization in machine learning due to its efficiency.

2.1.2.6 Optimizer

An optimizer is a set of functions and algorithms to optimize a neural network in Machine Learning. It regulates a learning rate and utilizing functions including a back-propagation function and a loss function to optimize a neural network as desired. Stochastic Gradient Descent is one of the optimizers.

2.1.2.7 Back-propagation

A back-propagation is a function to calculate each layer's gradient iteratively from the last layer to the first layer by the chain rule. It efficiently calculates the gradient by using gradient methods including gradient descent or stochastic gradient descent.

2.2 Reinforcement Learning Framework

RL is one of the learning methods of Machine Learning. In RL, the necessary components are an environment, state, agent, and reward. An agent takes an action after receiving a state from an environment, then a reward will be given to the agent according to the choice. The state is the current situation provided by the environment and it can be either dependent on the action or not. For example, if reinforcement learning is about predicting stock prices, then the state can be a price, which could be independent of the agent's action. Whereas, if reinforcement learning is about walking, then the state can be the current position and the machine's posture information, which are highly dependent on the agent's action. This cycle will be reinforced until the termination criterion is met. This RL framework is depicted in Figure 2.5.

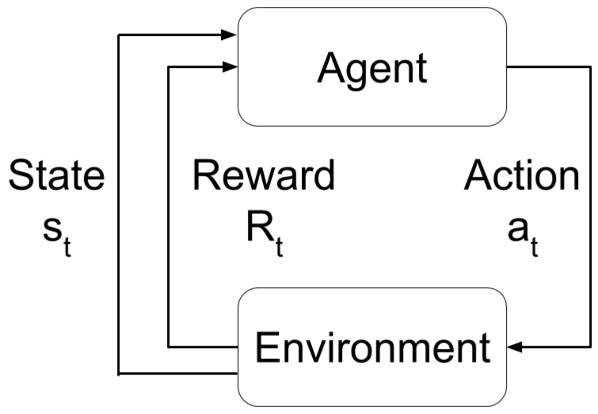

Figure 2.5: RL framework and components: This diagram depicts the framework of RL and the components including agent, environment, action, reward, and state are shown. When a state is provided to the agent, the agent chooses an action. A reward will be given to the agent accordingly. This cycle is repeated, and the agent learns by receiving rewards.

2.2.1 Agent

An agent is an object that decides an action at each time and learns iteratively.

2.2.2 Environment

In reinforcement learning, an environment is where an agent learns its prediction ability through the repeated cycles of making actions and receiving rewards and the rewards are given according to its correctness of the current actions. An environment provides state to an agent and the agent perceives the given state to predict the next action.

2.2.3 State

A state is the modified data gained from an environment, and an agent processes it. An agent in an environment perceives its current situation as a given state and decides how to react by selecting an action and will receive a reward after every action is made.

2.2.4 Reward

A reward gives an agent a sense of how well the decision was made. If the current action results somewhere close to the expected output, then the reward tends to be positive or big. On the contrary, if the current action creates an output far from the expected one, then the reward tends to be negative or small. However, how to calculate rewards is a design choice.

2.3 Evolution Strategies

ES [14, 15] are competitive optimization techniques, and their ideas came from evolution in nature. This optimization techniques were developed in the beginning 1960s by Ingo Rechenberg, Hans-Paul Schwefel, and others [5]. These techniques use mutation, recombination, and selection to optimize objects iteratively.

2.3.1 Terminologies

2.3.1.1 Generation

In ES, one cycle of its execution is called generation. This execution generally consists of selection, recombination (combining), mutation, and evaluation of its fitness.

2.3.1.2 Parent & Offspring

An existing population is required to create a new population, and they are called parent and offspring, respectively.

2.3.1.3 Mutation & Noise

Mutation is a method to create slightly different new neural networks from its parent by adding a noise which is usually a randomly distributed value between 0 and 1. Randomly distributed values are created as many as the number of neurons in a network then added to the parameters of the neurons. Therefore, slightly different multiple neural networks are created after mutation.

2.3.1.4 Evaluation Fitness

Once a new neural network is created, after the mutation process of an offspring, the evaluation fitness is conducted to check the performance of the new neural network.

2.3.1.5 Elite & Selection

After evaluating a neural network's fitness, a group of neural networks showing the highest performance is selected. This group is called the elite.

2.3.1.6 Recombination or Combining

Recombination is a process to create a new neural network by combining the elite's neural networks. Since the elite is the group that shows the highest performance, it is expected to create a better neural network by combining them iteratively.

2.3.1.7 Generation Gap

In one generation of ES, the mutation is conducted by population. Since they used the same neural network for the mutation, the difference between offspring's neural networks is bounded in a specific range. Thus, a neural network that is created in an ES can converge to an optimized network. If different generation's neural networks are combined, it is difficult to converge a neural network to an optimized network. This problem is called the generation gap [9, 5].

2.3.2 Common ES types

2.3.2.1 The Canonical ES

There are two canonical ES versions [5]. The only difference between them is where they select the elite. One selects the elite from the offspring, whereas the other selects the elite from both parents and their offspring. In this work, only the first case (selecting an elite from offspring) was used, described in Figure 2.6. Also, Figure 2.7 shows how the canonical ES replaces their new mutated network with the existing networks at every generation.

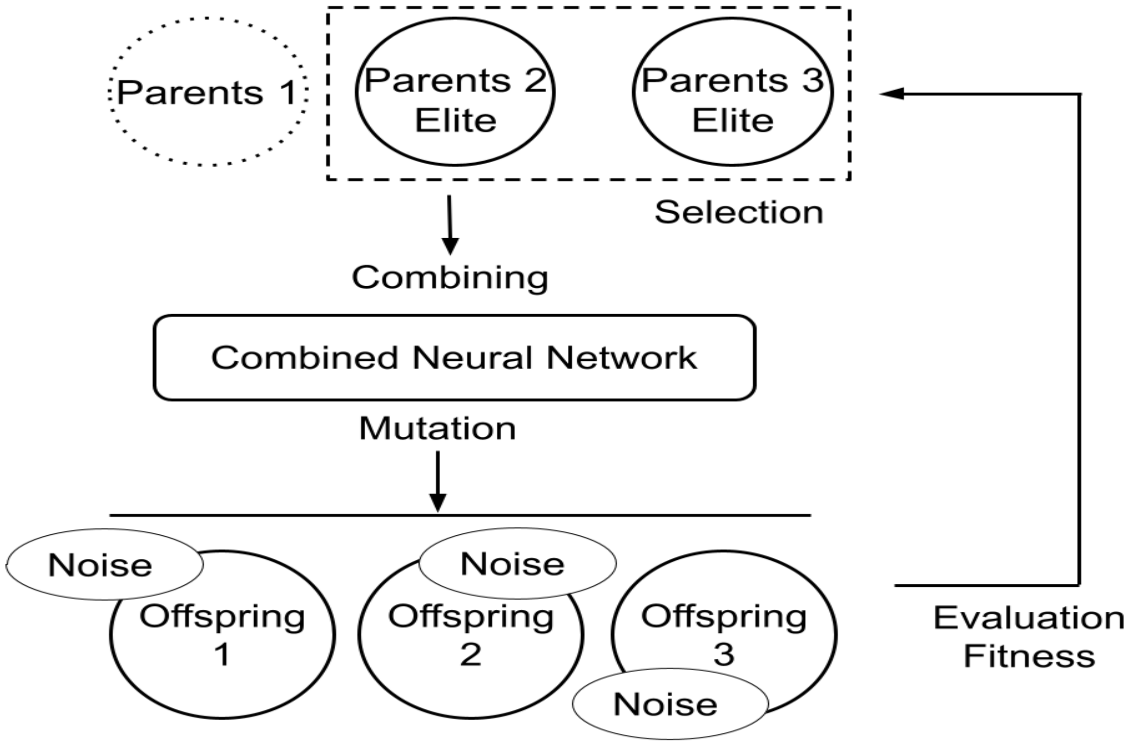

Figure 2.6: Canonical ES process diagram: This diagram describes the generation cycle of the canonical ES after the initializing process, which includes creating population individuals and distributing an initial network to the created population individuals. Firstly, a selection is conducted to select the elites. These chosen elites are to be recombined, and a mutation is executed to create offspring for the next generation. Each offspring's neural network is mutated with a noise, which is a randomly distributed value between zero to one. So, in the figure, three different noises are added into the networks of offspring of the current generation. This cycle is repeated until the termination criterion is met.

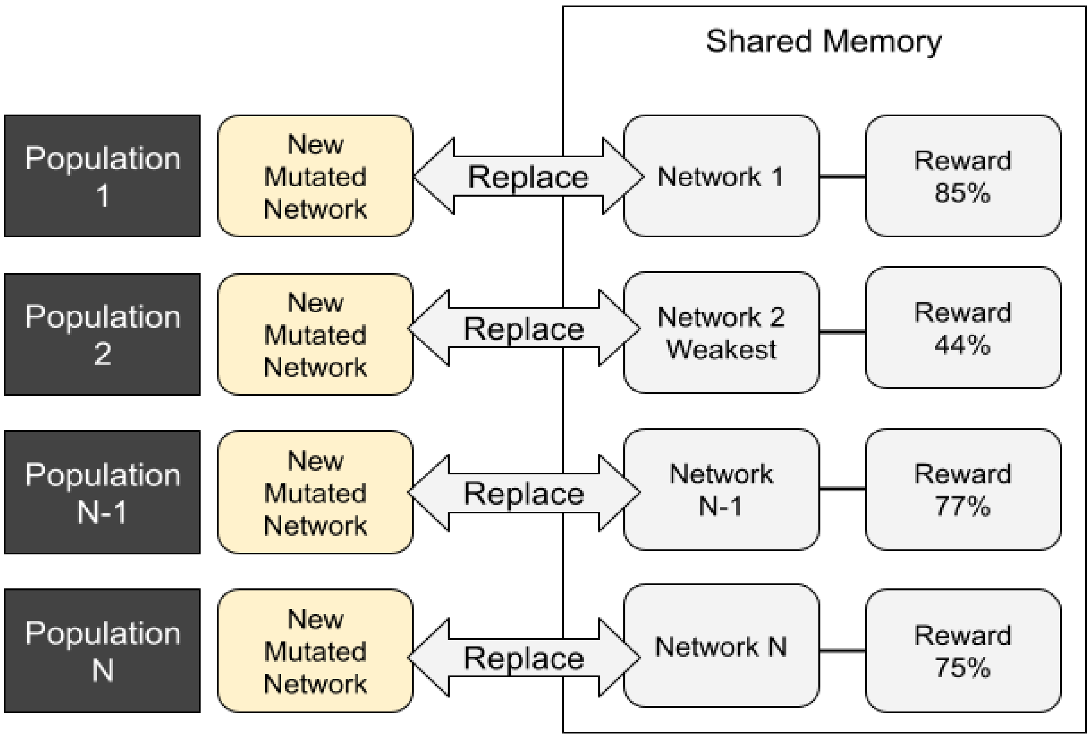

Figure 2.7: Canonical ES network update process: This diagram describes how networks are updated in the canonical ES system. All the current population individuals must be replaced with the newly created population individuals at every generation cycle.

2.3.2.2 Steady-State ES

Steady-state ES is a variant of popular ES methods. Only one offspring is created at each generation cycle, and the offspring replaces the worst network that shows the worst fitness result. The populations are updated steadily, but steady-state ES can prevent generation gaps from occurring [9, 5]. Therefore, steady-state ES is generally utilized for parallel asynchronous implementations. [9, 5].

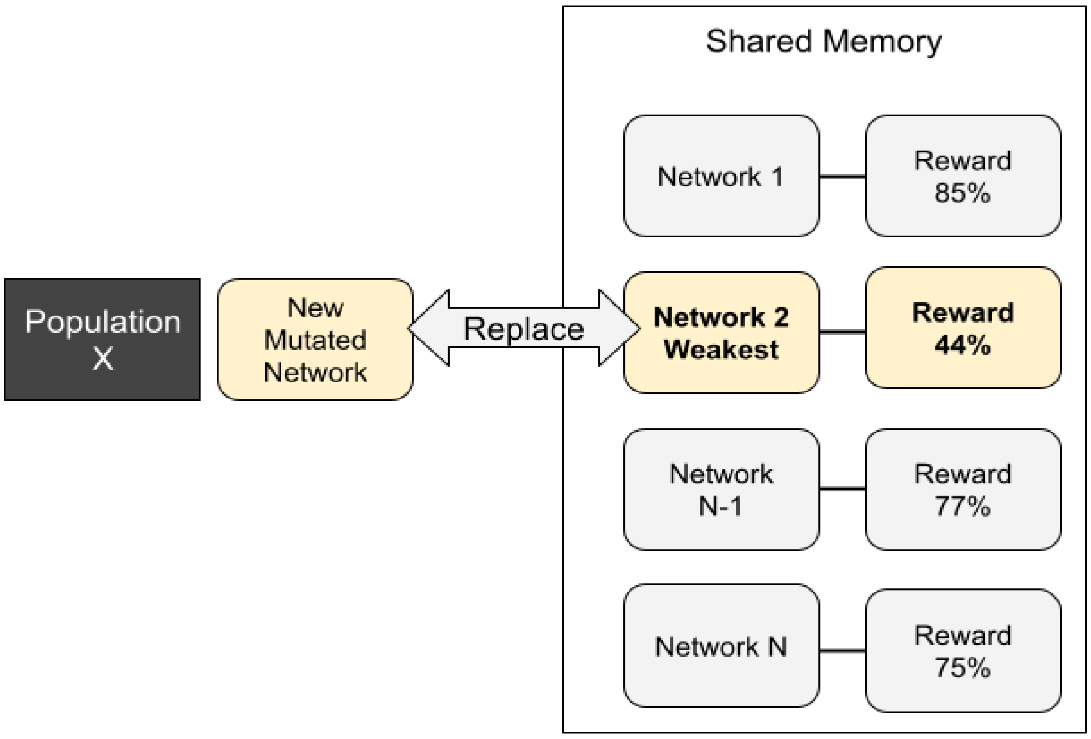

Figure 2.8: Steady-state ES network update process: This diagram depicts how networks are updated at every generation cycle. Each population creates a new mutated network, and the newly create network is to replace the weakest network that shows the lowest fitness results (reward percentage). Because it only replaces one network at one generation, it can prevent generation gaps. However, the evolution speed is relatively slow compared to the canonical ES.

2.4 Parallel Programming

In this section, parallel-related terminologies are briefly explained. Mainly, threads are used to implement parallel ES systems, and the Pthreads library in C++ is utilized for that.

2.4.1 Threads

A thread is the slightest instruction flow. Threads share data, including global variables, and this makes cooperating between threads easier.

2.4.2 Processes

A process is an executing program that is operated by one or multiple threads.

2.4.3 Multitasking

Multitasking is a computing skill that enables the concurrent execution of multiple tasks over a specific time. It improves the efficient use of computer hardware to perceive that multiple tasks are executing simultaneously.

2.4.4 Problem decomposition

2.4.4.1 Task parallelism

A task is divided and executed by multiple cores.

2.4.4.2 Data parallelism

Data is divided and executed by multiple cores.

2.4.5 Critical Section

The critical section is the data where multiple threads can access and change. This section is critical since when more than two threads change the current value, it causes inconsistency.

2.4.6 Mutual Exclusion

Mutual exclusion is a property that ensures consistency in critical sections.

2.4.7 Concurrency

Concurrency is the skill of enabling multiple threads or processes to be executed in parallel simultaneously.

2.4.8 Multithreading

Multithreading is the ability to enable multiple threads to work concurrently.

2.4.9 Pthreads Library and its calls

POSIX threads or Pthreads is an execution model that provides multiple functions and variables to facilitate parallel implementations. Some Pthread related calls that are used in this study are written below.

2.4.9.1 Pthread_create

A function to create a thread in the calling process.

2.4.9.2 Pthread_join

A function to have the calling process or thread wait until a designated group of threads is terminated.

2.4.9.3 Pthread_exit

A function to terminate a calling thread.

2.4.9.4 Pthread_cond_wait

A function to have a calling process or thread waits until pthread_cond_signal or pthread_cond_broadcast is called.

2.4.9.5 Pthread_cond_signal

A function to wake up a designated process or thread waiting after calling pthread_cond_wait.

2.4.9.6 Pthread_cond_broadcast

A function to wake up all processes or threads that are waiting after calling pthread_cond_wait.

2.4.9.7 Mutex

Mutexes are mutual exclusion locks designed to protect shared data or other resources accessed by multiple processes or threads simultaneously.

2.4.9.8 Condition Variable

Condition variables are to provide ways to have processes or threads wait until some conditions are met.

3
Method

This chapter will discuss the applied reinforcement learning framework and the details of the implemented three ES systems. There are three methods that were used in this work. Each method consists of the same applied reinforcement learning framework with their own implemented ES system. By applying the same reinforcement learning framework, it was able to compare the training speeds of the three implemented ES systems. The following sections 3.1, 3.2.1, 3.2.2, and 3.2.3 in the chapter describe the details of each implementation and the applied reinforcement learning framework.

3.1 Applied Reinforcement Learning Framework

One reinforcement learning framework was used with the implemented ES systems and predicting stock prices was the subject. In this applied framework, there are several major components including an agent, action, state, and reward, and they interact in an environment. In an episode, the agent chooses an action (prediction stock price changes) after receiving a state from an environment. Then a reward is given to the agent according to the previous action. If a prediction (action) is correct, then a positive reward is given to the agent so that the agent could learn how to predict precisely. In detail, increase, decrease, and stay are the actions that the agent could choose at one time. If the price increases more than one percent than the previous day's close price, then the correct action is an increase. If the target price decreases more than one percent than the previous day's close price, then the correct action is a decrease. Otherwise, the correct action is a stay. Generally, an environment is where an agent interacts with it and learns its prediction ability through multiple episodes by making actions and receiving rewards and states. In this study, stock price prediction is the subject of reinforcement learning. So, Apple's stock prices were used as states in the environment and the historical accumulated stock prices of Apple were achieved from Yahoo Finance. A state is given to the agent as a vector that consists of the price differences between 50 current and previous prices in a row. In case not enough previous prices exist, then zeros are used as the non-existing close prices. For example, if the agent predicts the price of the first day in the training dataset, then the state is a vector that consists of the differences between 50 zeros since the training dataset does not contain the previous prices. When the agent predicts correctly, then one is given as a reward. Otherwise, zero is given. Figure 3.1 describes how the applied RL framework works.

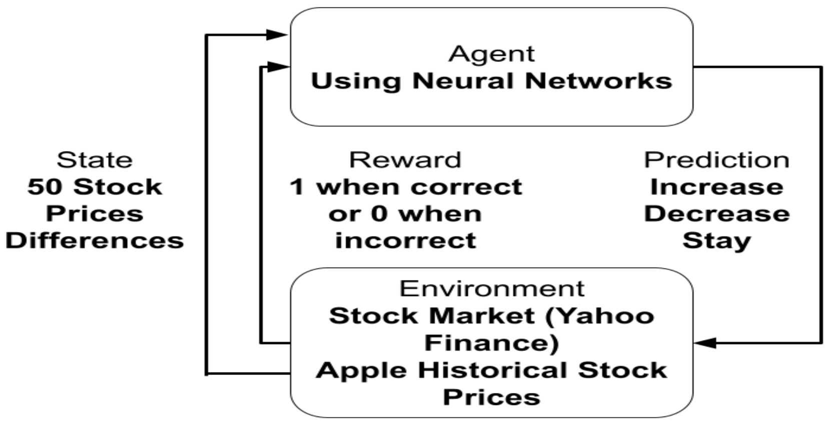
Agent
Using Neural Networks
State
50 Stock Prices Differences
Reward
1 when correct or 0 when incorrect
Prediction
Increase
Decrease
Stay
Environment
Stock Market (Yahoo Finance)
Apple Historical Stock Prices

Figure 3.1: Applied RL framework: This diagram shows how the applied reinforcement learning framework works. First, the environment provides a state to the agent. The agent predicts whether the next day's stock prices will increase, decrease, or stay. If the prediction is correct, then one will be provided as a reward. Otherwise, zero will be given to the agent. This cycle continues until the termination criterion is met.

3.2 Implemented ES systems

In this thesis, three ES systems were implemented. The first and second implementations are based on existing ES systems, the canonical ES and steady-state ES, respectively. The last one is a newly suggested ES system: VOPAES. Subsections from 3.2.1 to 3.2.3 describe the details of each ES system's implementations.

3.2.1 Parallel Synchronous Canonical ES

This ES system is based on the existing ES called the canonical ES. In this implementation, synchronization is required to prevent generation gaps and this work is conducted by a master process. Not only synchronization, but the master process also conducts selecting elites and recombination, since these tasks require one process to be done. On the other side, there are tasks that can be executed in parallel, and they are mutation and evaluating its fitness. In this work, multithreading was utilized for task parallelism. Therefore, several workers (threads) are created as many as the number of population individuals at every generation cycle by the master. In this parallel synchronous ES system, the number of the population is fixed to 12, since it showed the highest performance in the initial tests. Pthread calling, including pthread_join, pthread_create, and pthread_exit, was utilized to facilitate synchronization. The master process and workers should share some data including the state of neural networks, their fitness results, and recombined networks and the sharing is conducted via shared variables. The shared variables are the critical sections, and they are protected by a mutex to maintain consistency. In detail, the main process creates threads using the pthread_create function, then waits by calling the prhead_join function until every thread terminates. The created twelve threads evaluate their network's fitness and save the fitness results with the mutated neural network before terminating themselves. After all threads are terminated, the master process starts to rank the networks by their fitness results to select elites. Then, the master process creates a new neural network which is the average value of the selected elites' networks, and this task is called recombination. This cycle is repeated until the test results become equal or greater than the termination criterion. This cycle is described in Figure 3.2, and a pseudo-code of the parallel synchronous canonical ES is written in Table 3.1.

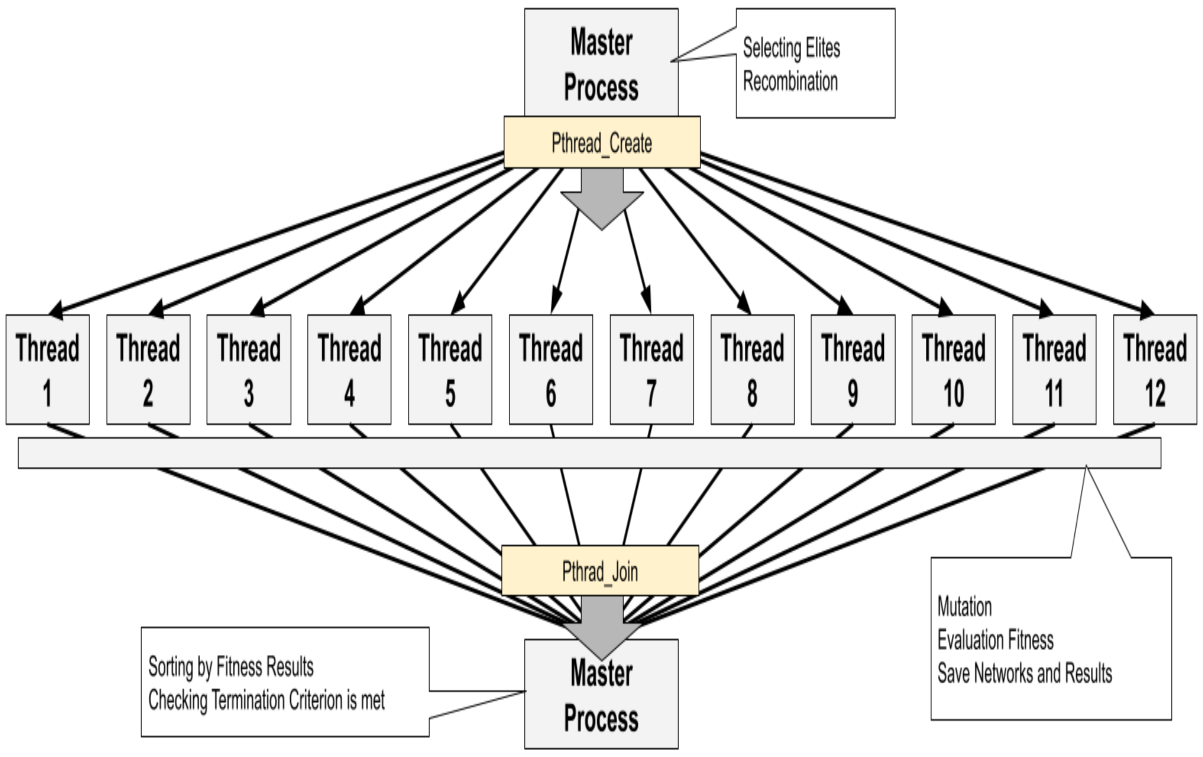

Figure 3.2: Parallel synchronous canonical ES: This diagram depicts how the parallel synchronous ES works in parallel and synchronously with multiple threads using pthread functions. At each generation cycle, the master creates as many threads (workers) as the populations, which is 12. Then, the master process waits until the created threads complete their tasks by calling the pthread_join function. Meanwhile, the created workers individually conduct their assigned tasks and terminate themselves by calling the pthread_exit function. When every created thread successfully terminates itself, the master process could proceed with its task. This cycle continues until the termination criterion is met.

Algorithm 1 Parallel Synchronous Canonical ES

```
1: Input: initial parameters $\theta$, elite number $en$, worker number $n$
2: Initialize: $n$ workers and initial parameters $\theta_o$
3: while termination criterion is not met do
4:     Mutation sample $\epsilon_1, ...\epsilon_n \sim$ N(0, I)
5:     Evaluation Fitness compute returns $F_i = F(\theta_t + \sigma\epsilon_i)$ for $i = 1, ..., n$
6:     Synchronization
7:     Selection sort, rank, select elites
8:     Recombination Set $\theta_{t+1} \leftarrow \theta_t + \frac{1}{en}\sum_{i=1}^{en} \epsilon_i$
9:      t = t+1
10: end while
```

Table 3.1: Pseudo-code of the canonical ES

3.2.2 Parallel Asynchronous steady-state ES

The parallel asynchronous steady-state ES system was implemented based on the existing ES, called steady-state ES. Since the parallel steady-state ES does not require synchronization, the master process was not needed as it was required in the previous implementation. Thus, each population individually conducts its tasks by multithreading and the entire training is conducted until the termination criterion is met. However, the population individuals still need to share some data, including neural networks and networks' fitness results. The sharing was conducted via shared variables and a mutex was utilized to provide mutual exclusion in the critical sections. The tasks that are conducted in one generation cycle are loading the current data, selecting, recombination, mutation, evaluating network fitness, and saving them into shared variables. In this ES system, one newly mutated network replaces the weakest network that shows the lowest fitness result (reward percentage) at each generation cycle, and 12 threads individually conduct the replacing work via the shared variables. For instance, when a worker(thread) could access the shared variables after achieving the mutex, the thread loads the current data into its memory and returns the mutex. The thread continues its own tasks with the data in its memory unless the termination criterion is met. Therefore, other threads are able to access the data before the thread finishes its task, which rises efficiency. When a worker completes creating a new mutated network and evaluating its fitness result, the worker immediately tries to access the shared variables once again to save the network and its fitness result. The generation cycles in the entire training repeat until the termination criterion is met. Figure 3.3 describes how threads conduct their work individually without having the master by utilizing the shared and individual variables. While Table 3.2 shows a pseudo-code of the parallel asynchronous steady-state ES.

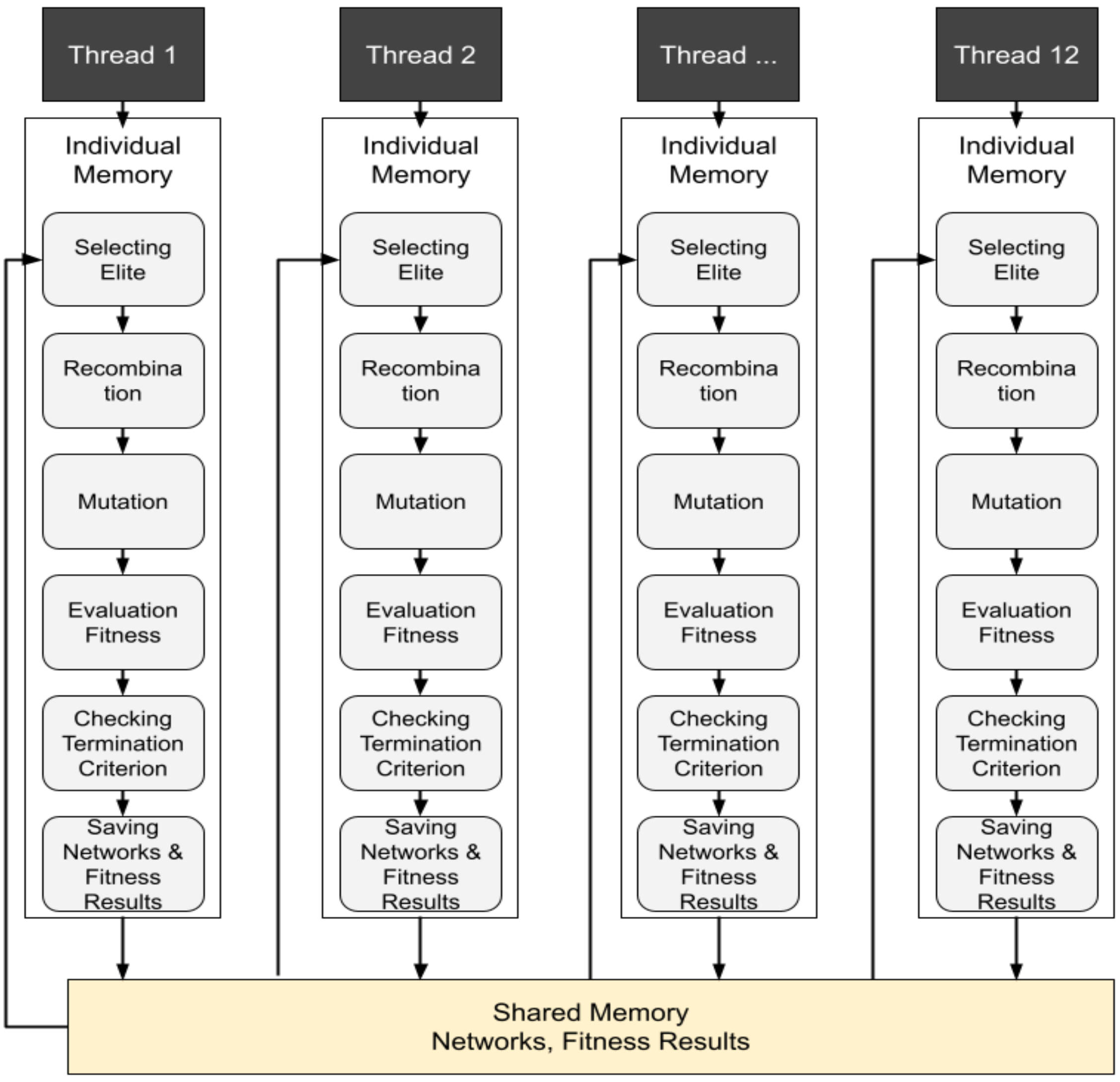

Figure 3.3: Parallel asynchronous steady-state ES: This diagram shows how each worker(thread) conducts its tasks. Workers are created and individually repeat their generation cycles. They work individually and separately but their latest networks and fitness results are shared via shared variables. When a worker finishes its tasks, the worker immediately tries to get access to the shared variables. Once the worker achieves a mutex, it loads the current saved networks and their fitness results to its memory. Then it returns the mutex and begins conducting tasks including selecting, recombination, mutation, as it is written in the diagram. This cycle continues until the termination criterion (reward percentage) is met.

Algorithm 2 Parallel Asynchronous Steady-State ES

```
1: Input: initial parameters $\theta$, elite number $en$, worker number $n$
2: Initialize: $n$ workers and initial parameters $\theta_o$
3:     Mutation sample $\epsilon_1, ...\epsilon_n \sim$ N(0, I)
4:     Evaluation Fitness compute returns $F_i = F(\theta_t + \sigma\epsilon_i)$ for $i = 1, ..., n$
5: while termination criterion is not met do
6:     Selection sort, rank, select elites and the weakest population individual
7:     Recombination Set $\theta_{recombined} \leftarrow \frac{1}{en}\sum_{i=1}^{en}\theta_{i_{(t)}}$
8:     Mutation Set $\theta_{mutated} \leftarrow \theta_{recombined} + \epsilon \sim$ N(0, I)
9:     Evaluation Fitness compute returns $F(\theta_{mutated})$
10:    Replace newly created network replaces the weakest network
11:      t = t+1
12: end while
```

Table 3.2: Pseudo-code of the steady-state ES

3.2.3 Proposed Approach (Contribution): VOPAES

VOPAES is the proposed ES that works in parallel and asynchronously. On the one hand, VOPAES is similar to the canonical ES since it replaces all the population at each generation cycle. On the other hand, VOPAES is different from the parallel synchronous canonical ES because each worker separately conducts its tasks without the master process. That means VOPAES is an asynchronous version of the canonical parallel ES and the purpose of this solution is to reduce the CPU idle time but maintain the learning speed as fast as the canonical synchronous parallel ES. Thus, the most part of the implementation is similar to the previous two implementations. Workers(threads) share data including the latest networks, fitness results, and versions through the shared variables that are protected by a mutex. Once the current data is loaded to each thread's individual memory, then the threads start conducting their work separately. This design decision was made to reduce the memory bottleneck problem. The selection and recombination are conducted after loading the shared variables into the worker's own memory. However, in order to making the canonical parallel ES system as an asynchronous system, we need to solve the generation gap problem. The solution or contribution of this work that we suggested is attaching a version that shows the network's generation to each network so that the canonical parallel ES system can work asynchronously without generation gaps. In detail, the best elite which shows the highest fitness result is chosen, and the best elite's version becomes the criterion version. This chosen criterion version is the criterion that decides whether the remaining networks are included or excluded in recombination. As a result, only the networks that have the version with the criterion version can be included in the recombination process. If the version is not identical to the criterion version, the related network is discarded. For instance, if the best elite's version is four and the second-best elite's version is three, the second-best elite' network is not combined since it has a different version. Then, recombination (combining) is executed among the elites who have the identical version, and the next generation's neural network is created. The version is incremented by one from the best elite's network version. Thus, in this case, the new network's version becomes five. Now mutation and evaluation of its fitness are conducted. The fitness result, the mutated neural network, and its version are saved in the shared variables after achieving the mutex. Finished populations do not wait until other populations complete these tasks and immediately start to achieve the mutex for the following generation. This generation cycle continues until the termination criterion is met. Figure 3.4 depicts how VOPAES works in parallel and asynchronously, while Figure 3.5 focuses explaining how the versions are used for preventing generation gaps in the recombination. Also, a pseudo-code of VOPAES is written in Table 3.3.

Algorithm 3 Proposed Approach (Contribution): VOPAES

```
1: Input: initial parameters $\theta$, elite number $en$, worker number $n$, version $v$, recombine number $rn$
2: Initialize: version $v$, recombine number $rn$, $n$ workers, initial parameters $\theta_o$
3: while termination criterion is not met do
4:     Mutation sample $\epsilon_1, ...\epsilon_n \sim$ N(0, I)
5:     Evaluation Fitness compute returns $F_i = F(\theta_t + \sigma\epsilon_i)$ for $i = 1, ..., n$
6:     Loading Load the current networks from the shared variables into the individual memory
7:     Selection sort, rank, select elites
8:     Version Check Set $v \leftarrow v_{(HighestElite)}$, $rn \leftarrow 0$, $\theta_{sum} \leftarrow 0$
9:     Recombination
10:            for $r(rank) = 0, 1, 2, ..., en$ do
11:            if $v_r == v$
12:                $rn$=$rn$+1
13:                $\theta_{sum} = \theta_{sum} + \theta_r$
14:            end if
15:        end for
16:        Set $\theta_{t+1} \leftarrow \frac{1}{rn}\theta_{sum}$
17:        Saving Save $\theta_{t+1}$ into the shared variables
18: t = t+1
19: end while
```

Table 3.3: Pseudo-code of VOPAES

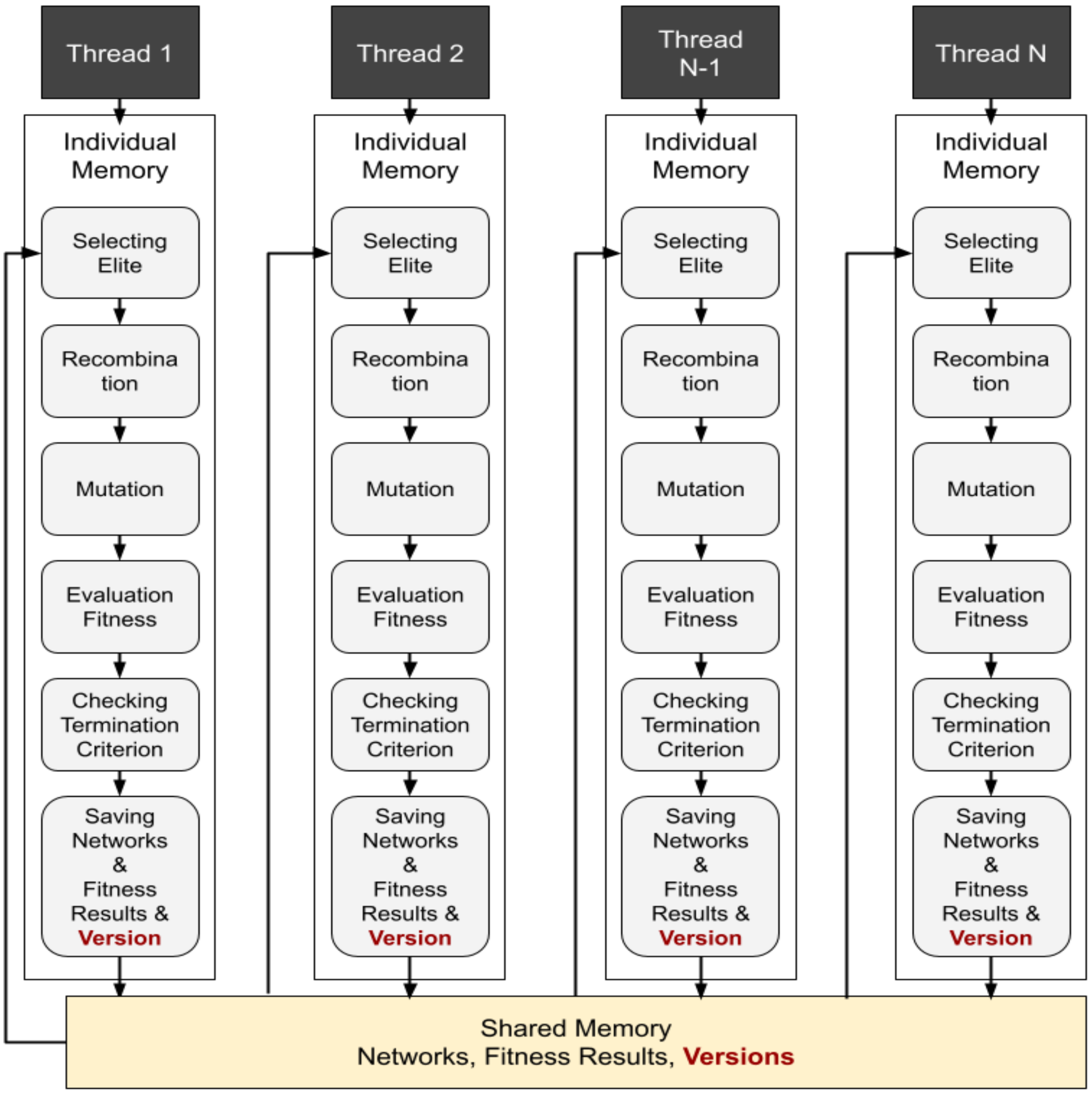

Figure 3.4: VOPAES diagram: This diagram depicts how VOPAES works parallel but asynchronously. In the beginning, threads (workers) are created once and receive a randomly created initial neural network. Then, the threads (workers) begin conducting their tasks. The versions in the diagram are highlighted as reading color in order to stress the difference between the parallel asynchronous steady-state ES system. This cycle continues until one of the threads reaches the termination criterion (reward percentage).

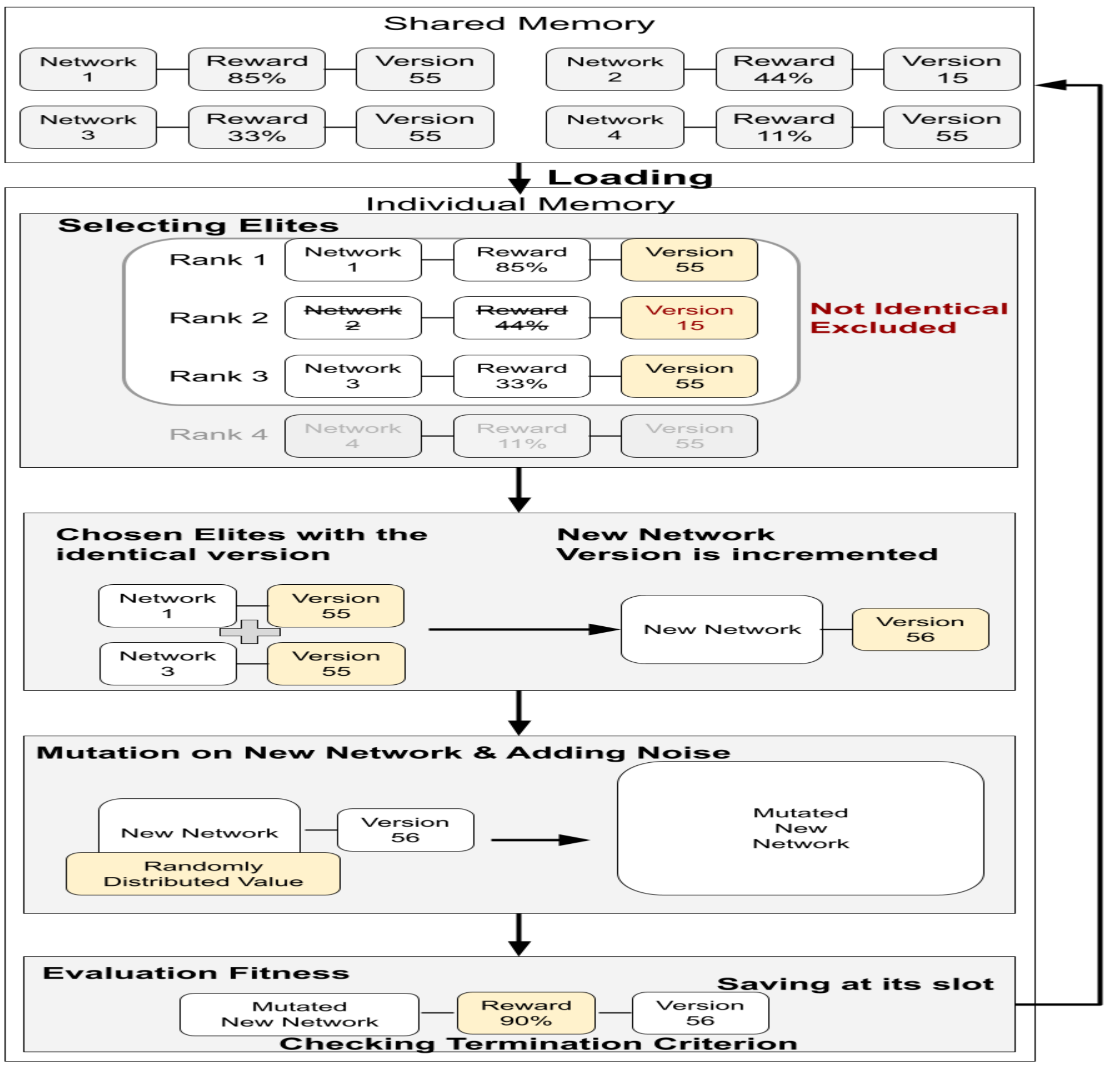

Figure 3.5: VOPAES example detail diagram: This diagram shows how the versions facilitate to prevent generation gaps in recombination. After loading the latest networks, versions, and fitness results, the selection begins at a worker's individual memory. The networks are now ranked (sorted) by their fitness results (reward percentages), and the current elites are selected. Then checking versions is conducted among the chosen elites. For example, in this diagram, the second-ranked network two is excluded in this diagram although if it shows the second-highest fitness result. Because it has a different version from the first one, this version checking process prevents generation gaps from occurring. Hence, only networks one and three are left to be combined, and the new combined network's version is incremented by one. If network two had the same version, 55, all elites, including network two, would be recombined. Then the mutation, which is adding a randomly distributed value to the combined network, is conducted. After evaluating its fitness, the new fitness result will be checked to be equal to or greater than the termination criterion (reward percentage). The fitness result, the network, and version are to be saved in the shared variables.

4
Results

In this chapter, data collection, test environment, and test results are described. The scope of this work was to study how to decrease the training duration of ES. Thus, the test results focused on comparing the optimization performances of the implemented ES systems during its training. Three tests were conducted; the first test was to validate that those three implemented ES systems could optimize neural networks during training. In this case, if the reward percentages of the implemented ES systems tend to increase during the training, it is validated that the learning process properly works. Therefore, the reward percentages in a 40-minute period were recorded and compared in Figure 4.1. The second test was designed to measure the average duration of each ES system. In this test, the termination criterion was set high so that the training duration could extend. The last experiment was to analyze how VOPAES could decrease the total training duration compared to other ES systems. Comparing the number of epochs per second of the implemented ES systems could determine that the suggested solution, VOPAES, is either effective or efficient.

4.1 Data (Historical Stock Prices) Collection

Accumulated historical stock price data were required as input data for this project. Yahoo Finance, which provides multiple American Companies' daily stock prices, was chosen as the input data source, and Apple's stock prices were particularly selected. Provided data from Yahoo Finance consists of seven categories: data, open price, high price, low price, close price, adjacent price, and volume. Yet, only close values, which are the last transacted price of a stock before the market officially closes, were used for stock price prediction.

4.2 Test Environment

A personal computer was utilized during the project. The specification of the personal computer and development software environment is described in Table 4.1.

CPU	Intel Core i7 Processor 8565U - 4 cores (8 threads) - 1.8 GHz up to 4.6GHz 8MB L3 Cache
Memory	12GB LPDDR3
OS	Linux Ubuntu 20.04 LTS
CMake	3.16.3
Virtual Box	6.1
Tools	Visual Studio Code, Google Colab
Programming Language	C++, Python (for only plotting)

Table 4.1: Test Environment

4.3 Test 1: Reward Percentage Increase in 40 minutes

This test validated that the implemented three ES systems could optimize their neural networks by observing the increase of reward percentages. To precisely compare the results, the same training time, which is 40 minutes, was applied to every ES system for training the same data (Apple's stock prices for one year). The criterion for determining whether the action is an increase, decrease, or stay is one percent difference. Hereafter, this term is used as the action determining criterion. For instance, if the close price of the next day increases more than one percent than the close price of today, then an increase is the correct prediction. On the contrary, if the close price of the next day decreases more than one percent than the close price of today, then a decrease is the correct prediction. Otherwise, a stay is the precise one. First, VOPAES showed the highest increase ratio of the reward percentage during the test. The synchronous parallel ES of the canonical ES followed the first one. The asynchronous ES using steady-state ES showed the lowest performance in increasing reward percentage during the training. The test result showing how each ES system optimizes its neural network is depicted in Figure 4.1.

As a result, the reward percentages of the three ES systems reached almost the same point at around five minutes after the test began. Then, the percentage rewards of the parallel synchronous canonical ES and VOPAES went up more rapidly than the parallel asynchronous steady-state ES. Eventually, VOPAES could reach the highest reward percentage among the implemented ES systems at the end of the training. As Figure 4.1 shows, three systems could optimize the neural networks. Because the reward percentages tended to increase, although some fluctuations existed. Another point of the results is that the initial reward percentage of the parallel asynchronous steady-state ES is the lowest among the ES implementations, whereas the reward percentages of the other two ES systems started higher, which is at about 54 percent. Since the parallel asynchronous steady-state ES creates only one mutated network while the others produce 12 mutated networks at the beginning of the training. Thus, the possibility to find a higher reward percentage amongst the population of the first generation was higher in the parallel synchronous canonical

ES and VOPAES than the parallel asynchronous steady-state ES.

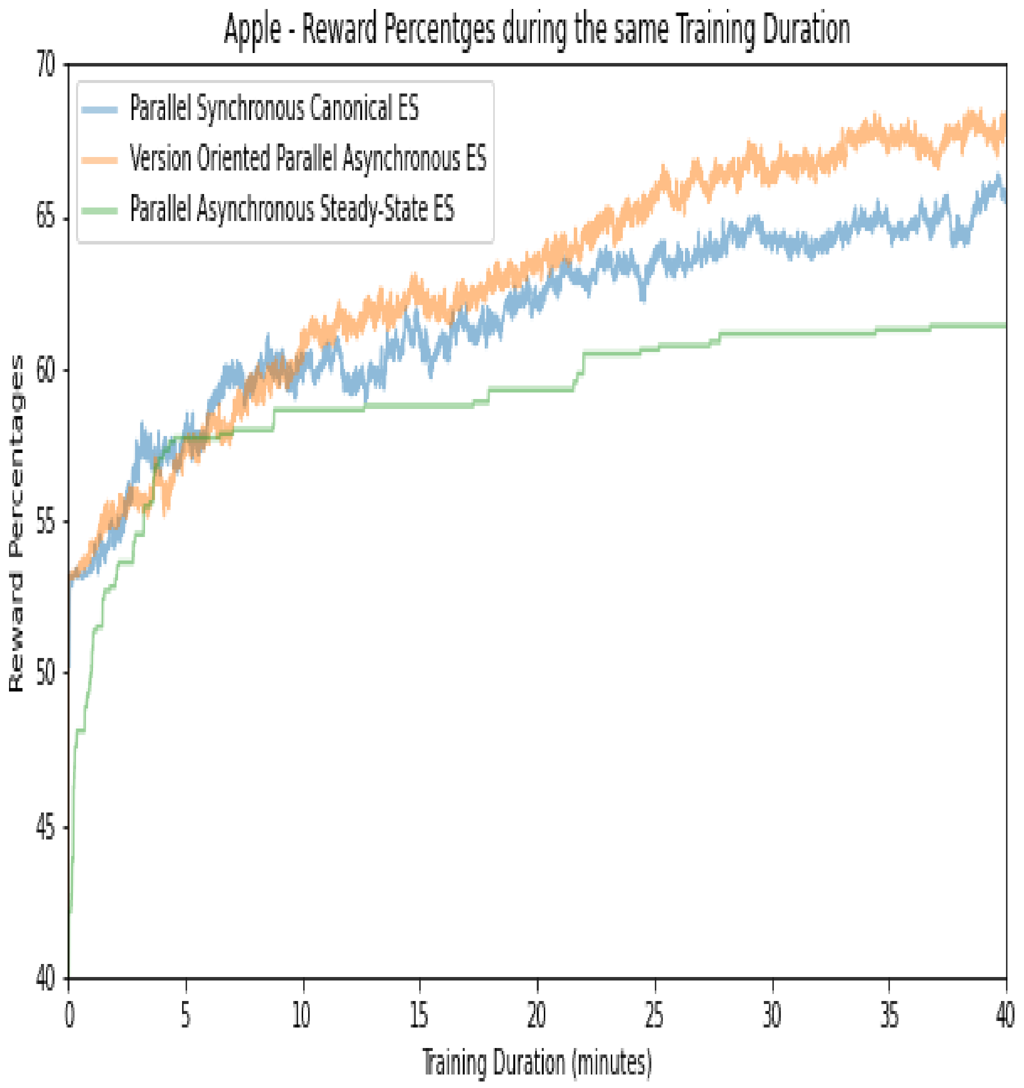

Figure 4.1: Test result 1 - Reward percentage changes in 40 minutes: This figure shows the reward percentage of each ES system during the same training duration.

4.4 Test 2: Eleven Tests for Average Training Duration

This test was conducted to show the average training duration of each ES system. Because ES is a randomized optimizing solution, and the training duration could vary. Despite that 90 percent could be a high termination criterion, which could cause the overfitting problem, the termination criterion was set at 90 percent. Because the scope of this study is finding a solution that increases the learning speed of ES rather than finding better solutions for stock price prediction. if the termination criterion is too low, it might be challenging to see the differences between the three ES systems. For instance, at the first test, VOPAES's reward percentage was even lower than the others until 10 minutes, nevertheless, it mostly outperformed the others after 10 minutes until the end of the training. For the same reason, more extensive data was used as the input data for this test. The training data for this test is Apple's historical stock price between 2011 to 2014 and, 90 percent was set as the termination criterion.

As a result, the parallel asynchronous steady-state ES could not reach the termination criterion, whereas the other ES systems could reach the termination criterion. Therefore, hereafter only the parallel synchronous canonical ES and VOPAES will be compared. The average training duration of the parallel synchronous ES was 80206 seconds, while for VOPAES it was 36446 seconds. VOPAES could reduce the training duration by approximately 55 percent than the parallel synchronous canonical ES in this test. The median and distribution of the training duration for the eleven tests are shown in Figure 4.2. It is also seen that the distribution of VOPEAS is narrower than the parallel synchronous canonical ES's.

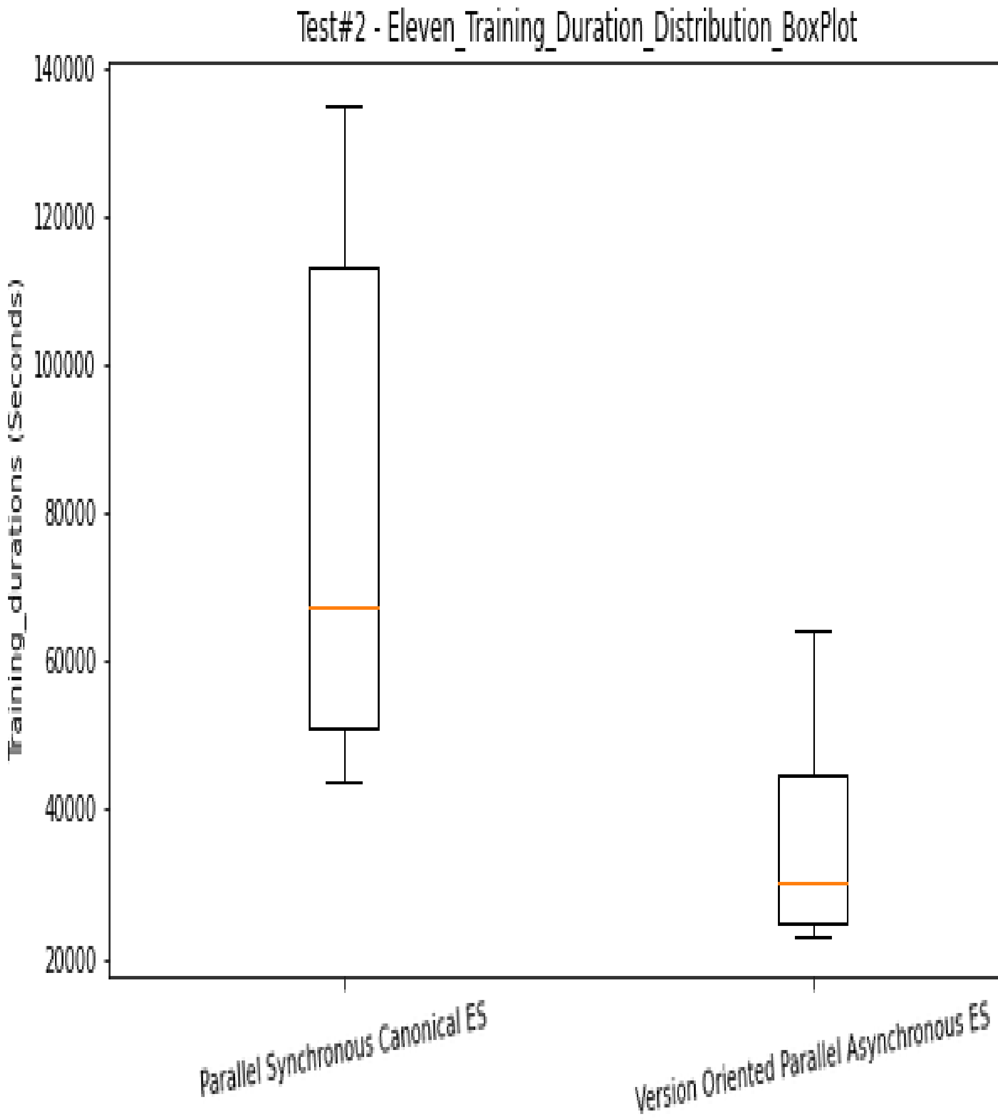

Figure 4.2: Test result 2 (The training is for reaching 90% reward percentage) - Training duration distribution and its median value in the eleven training: This figure shows the median and distribution of the eleven training duration of the parallel synchronous canonical ES and VOPAES, respectively.

4.5 Test 3: A test for measuring the required number of epochs

The final test aimed to analyze how VOPAES could outperform the parallel synchronous canonical ES. Since it was not certain how VOPAES could outperform the others. The reason could be either its efficiency or effectiveness. Therefore, the required epoch numbers per second in both parallel synchronous canonical ES and VOPAES were analyzed. For example, if VOPAES could use fewer epochs to optimize a neural network than the parallel synchronous canonical ES, then it indicates VOPAES works more effectively than the parallel synchronous canonical ES. In contrast, if VOPAES requires more epochs, but processes them in a shorter time, then that indicates VOPAES works more efficiently than the parallel synchronous canonical ES. The same termination criterion, the training data, and the action determining criterion were applied as Test 2. In VOPAES, each thread recorded its time per ES cycle by itself and summed up manually after training to measure the whole number of epochs conducted. While, in the parallel synchronous canonical ES, the master process recorded the time per ES cycle alone, and the total number of epochs was calculated by multiplying the number of epochs recorded by the master process to the number of workers because they work synchronously. For instance, if the master conducted 10 epochs and there are 12 workers, then the total epochs executed is 120 epochs.

Figure 4.3 shows the distribution of execution time per generation cycle of each thread. It is seen that the median and distribution of each thread's execution time per ES cycle were even. The medians are roughly 0.7 seconds, and 75 percent of data are distributed between 0.75 seconds and 0.6 seconds.

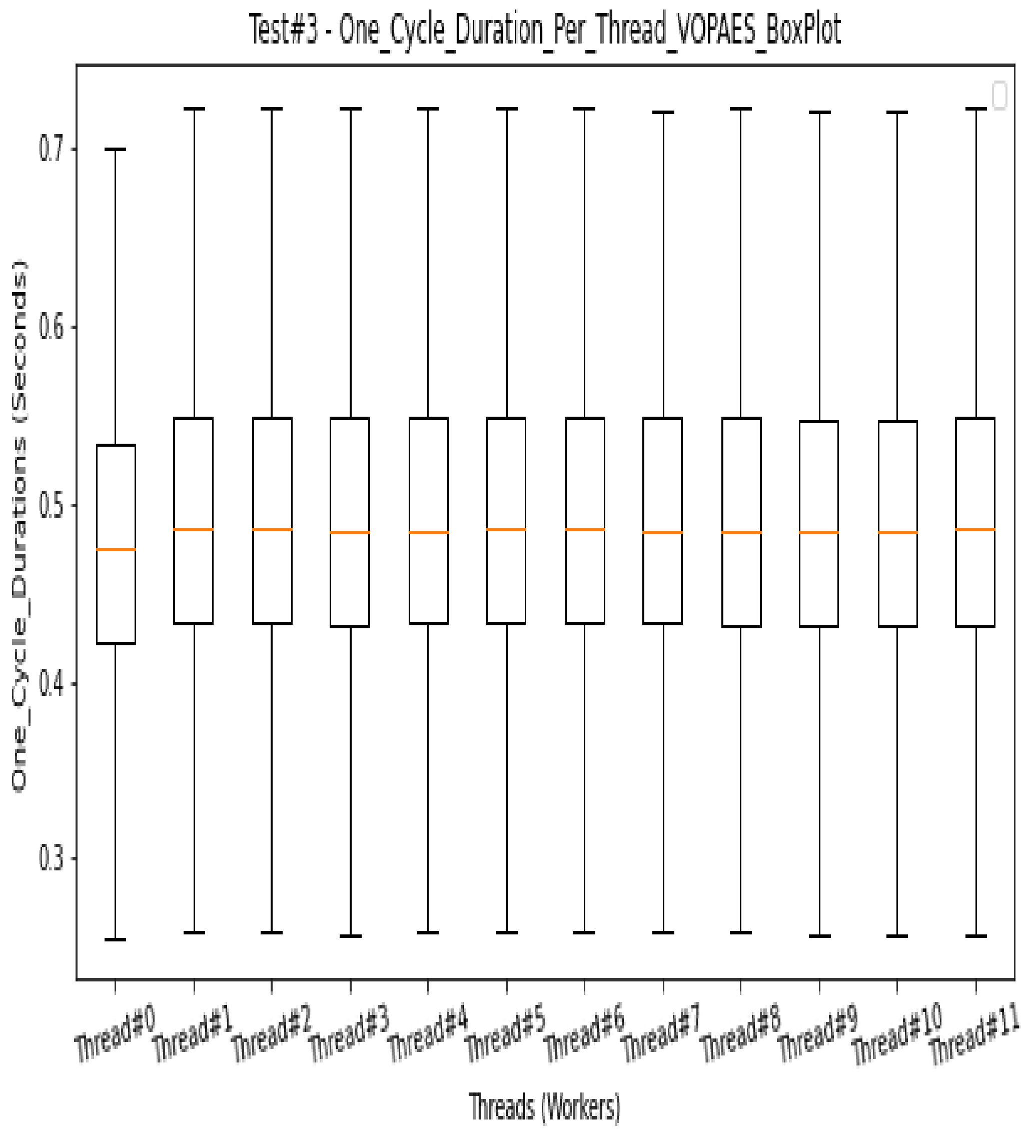

Figure 4.3: Test result 3 - Distribution box plot per thread in VOPAES: this figure shows the distribution and the median execution time per ES generation in VOPAES.

Furthermore, one cycle duration of the parallel synchronous canonical ES and VOPAES are compared and shown in Figure 4.4. It was expected that VOPAES could decrease the duration per generation cycle compared to the duration per generation cycle of the synchronous ES system since it does not require synchronization. However, it turned out that they both show similar times per one generation cycle. The medians of both ES systems are at around 4.8 seconds and the range of the distribution is between approximately 0.25 seconds to 0.71 seconds. In addition, it is seen that most data, which is three-quarters, is ranged between 0.43 seconds and 0.54 seconds. The reason why the one-cycle durations of VOPAES are similar to the ones of the parallel synchronous canonical ES is considered as that each thread in VOPAES requires completing more tasks per generation cycle compared to the parallel synchronous canonical ES. Hence, the two systems showed similar duration per generation cycle even VOPAES could decrease CPU idle time by working asynchronously.

Lastly, the number of epochs per training duration in both ES systems were analyzed and described as a bar chart in Figure 4.5. The parallel synchronous canonical ES and VOPAES showed very similar results at approximately 25 times. However, VOPAES showed a slightly lower number of epochs per training duration, around two percent. This test result indicates that VOPAES works more effectively compared to the parallel synchronous canonical ES. Since VOPAES could decrease the average training duration, but the required number of epochs per training duration is slightly fewer than the parallel synchronous canonical ES. This test result is analyzed in Section 5.1.3.

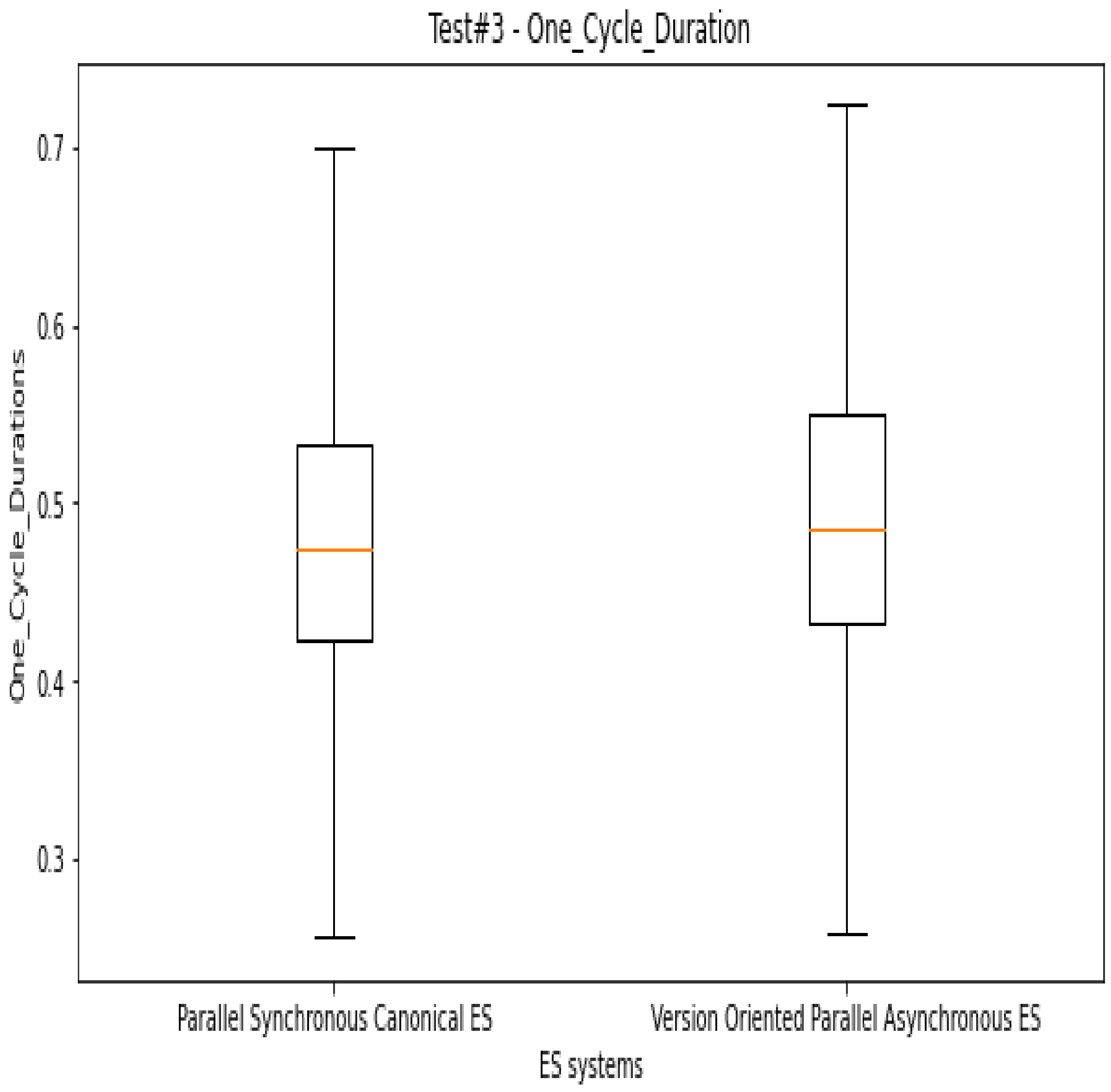

Figure 4.4: Test result 3 - Distribution box plot for one generation cycle duration: this figure shows the distribution of one-cycle durations and its median in VOPAES and the parallel synchronous canonical ES, respectively.

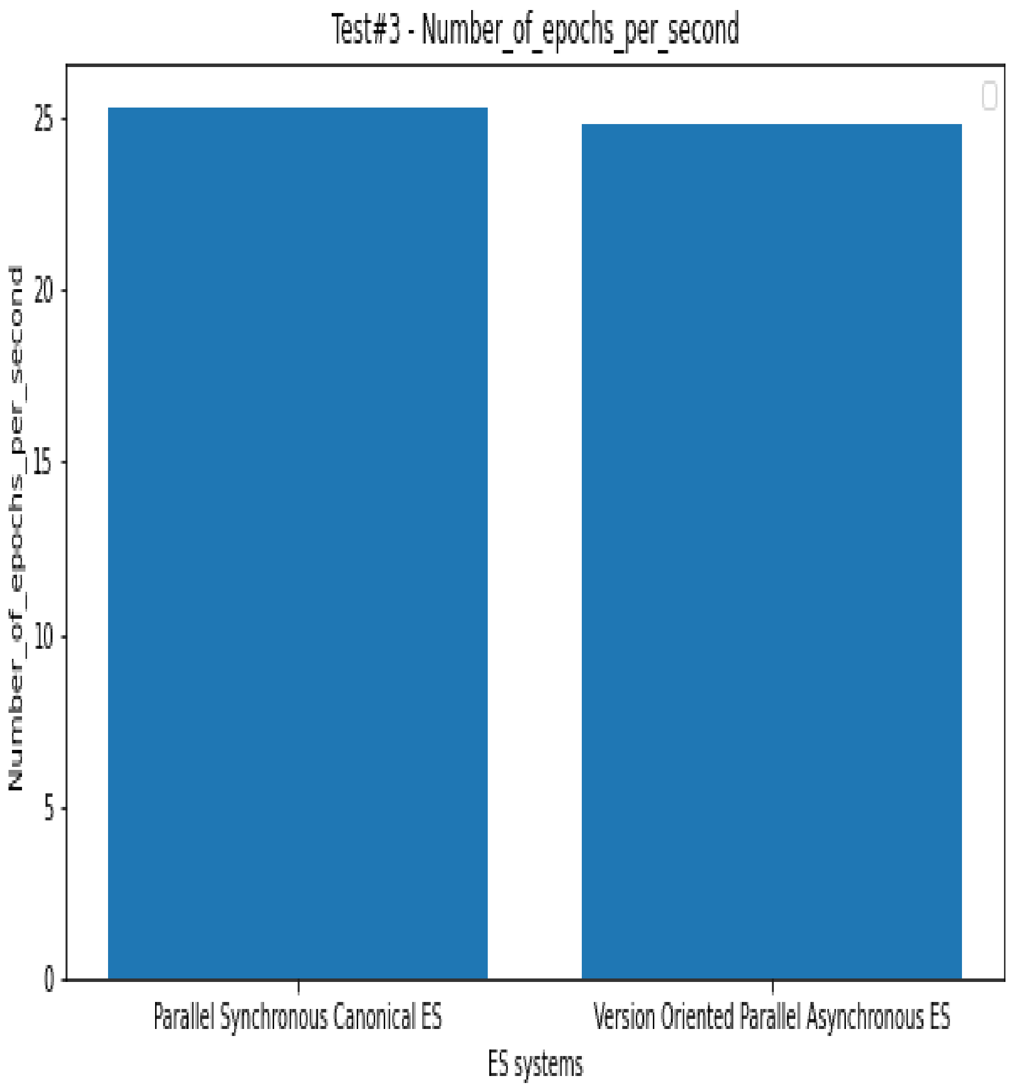

Figure 4.5: Test result 3 - Bar plot for the number of epochs per second: This bar plot shows the total number of epochs divided by the total training duration. As a result, parallel synchronous canonical ES and VOPAES both had very similar numbers of epochs per second, which is at around 25 times. The test results indicate that VOPAES could be a more effective optimization solution than parallel synchronous canonical ES.

5

Discussion & Conclusion

In this chapter, the discussion and conclusion will be discussed.

5.1 Discussion

In this section, lessons learned through the study, future work with follow-up questions, and contributions are discussed.

5.1.1 Lessons Learned

Three characteristics of parallel ES systems have been empirically studied through this study. Firstly, the parallel synchronous canonical ES showed a faster optimizing speed compared to the parallel asynchronous steady-state ES. Secondly, the parallel asynchronous steady-state ES system could not optimize a neural network as much as the parallel synchronous canonical ES system did. Lastly, generation gaps could be prevented by using versions that show generations so that VOPAES could optimize a neural network and the parallel synchronous ES system and even could decrease the average training duration by around 55 percent in the second test result.

5.1.2 Follow-up Questions & Future Work

A possible further test could be testing VOPAES with an increased number of cores. In this study, only 12 workers were used. It is expected that the benefits of VOPAES could be maximized by increasing the number of cores and workers. Furthermore, VOPAES can be utilized in other deep learning methods, such as supervised learning and unsupervised learning to check its compatibility and efficiency.

5.1.3 Contributions

The main contribution of this work is in increasing the learning speed and decreasing CPU idle of a parallel asynchronous canonical ES. The test results demonstrated that generation gaps in a parallel asynchronous canonical ES can be preventable by using versions. Moreover, the results of tests 2 and 3 showed that VOPAES could work more effectively than the parallel asynchronous canonical ES. For instance, in test 2, VOPAES could decrease the training duration by almost 54 percent than the parallel asynchronous canonical ES with a similar number of required epochs during the tests.

There are two reasons why VOPAES could decrease the training duration by almost 54 percent despite that the duration per epoch is extremely close to the parallel canonical ES's. Firstly, VOPAES has higher diversity in the parameter pool than the parallel canonical ES. Since, in VOPAES, each worker saves the current networks in the shared variables to its own memory, and they individually produce a new mutated network based on the saved networks in their separated memory. If the diversity increases, then the possibility to find the desired parameter set goes up as well. This is the same reason why the training time tends to decrease when the number of workers increases as long as the system provides enough capacity including the number of cores.

Secondly, VOPAES could reverse the previous generation if the latest generation of the network shows worse performance than the previous one and previous generation networks which show higher performance exists in its elite set. This was possible because that VOPAES is an asynchronous system. On the contrary, the parallel canonical ES cannot reverse the current generation even the latest one shows lower performance than the previous one. During the tests, it was detected that the highest reward percentage in the latest generation of the parallel canonical ES is lower than the highest reward percentage during the whole training. Because even the newly mutated network was created based on the best parameters in the previous generation, the mutation process could decrease the performance of the newly created one and it costs a considerable amount of time. Unlikely, VOPAES could save the networks that were created in previous generations, and if the newly created one could not perform better than them, VOPAES can use the previous networks.

However, each worker in VOPAES requires to work more than the workers in the parallel canonical ES. For example, each thread individually conducts recombination, mutation. Furthermore, they require to load the current networks in the shared variables after gaining the mutex. Therefore, the duration per epoch is similar to the parallel canonical ES even the time for synchronization is reduced.

5.2 Conclusion

In conclusion, VOPAES showed higher learning speed among the implemented synchronous and asynchronous ES during the tests. Specifically, the average training duration of VOPAES was approximately 55 percent lower than the synchronous parallel ES system in the second test. Furthermore, test three was conducted to analyze that VOPAES is either effective or efficient by calculating the number of epochs per second. As a result, it was found that VOPAES works more effectively than the parallel synchronous canonical ES since both ES system showed almost the same number of epochs per second, but VOPAES could decrease the training duration.

Bibliography

[1] S. Haykin, *Neural Networks and Learning Machines*, ser. Neural networks and learning machines. Prentice Hall, 2009, no. v. 10. [Online]. Available: https://books.google.co.kr/books?id=K7P36lKzI_QC

[2] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution strategies as a scalable alternative to reinforcement learning," 2017.

[3] P. Chrabaszcz, I. Loshchilov, and F. Hutter, "Back to basics: Benchmarking canonical evolution strategies for playing atari," *CoRR*, vol. abs/1802.08842, 2018. [Online]. Available: http://arxiv.org/abs/1802.08842

[4] N. Müller and T. Glasmachers, "Challenges in high-dimensional reinforcement learning with evolution strategies," in *Parallel Problem Solving from Nature – PPSN XV*, A. Auger, C. M. Fonseca, N. Lourenço, P. Machado, L. Paquete, and D. Whitley, Eds. Cham: Springer International Publishing, 2018, pp. 411–423.

[5] H. Beyer, "Evolution strategies," *Scholarpedia*, vol. 2, no. 8, p. 1965, 2007, revision #193589.

[6] M. Srouji and K. Dhabalia, "Parallel Price Prediction Using Reinforcement Learning," Master's thesis, Carnegie Mellon University, the USA, 2017.

[7] K. Lee, B.-U. Lee, U. Shin, and I. S. Kweon, "An efficient asynchronous method for integrating evolutionary and gradient-based policy search," 2021.

[8] E. O. Scott and K. A. De Jong, "Understanding simple asynchronous evolutionary algorithms," in *Proceedings of the 2015 ACM Conference on Foundations of Genetic Algorithms XIII*, ser. FOGA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 85–98. [Online]. Available: https://doi.org/10.1145/2725494.2725509

[9] E. S. Scott and K. De Jong, "Understanding simple asynchronous evolutionary algorithms," 01 2015.

[10] A. V. Clemente, H. N. Castejón, and A. Chandra, "Efficient parallel methods for deep reinforcement learning," 2017.

[11] D. Izzo, M. Rucinski, and C. Ampatzis, "Parallel global optimisation meta-heuristics using an asynchronous island-model."

[12] M. A. Martin, A. R. Bertels, and D. R. Tauritz, "Asynchronous parallel evolutionary algorithms: Leveraging heterogeneous fitness evaluation times for scalability and elitist parsimony pressure," in *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO Companion '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1429–1430. [Online]. Available: https://doi.org/10.1145/2739482.2764718

[13] I. Goodfellow, Y. Bengio, and A. Courvile, *Deep Learning.* The MIT Press, 2016.

[14] W. Vent, “Rechenberg, ingo, evolutionsstrategie — optimierung technischer systeme nach prinzipien der biologischen evolution. 170 s. mit 36 abb. frommann-holzboog-verlag. stuttgart 1973. broschiert,” *Feddes Repertorium*, vol. 86, no. 5, pp. 337–337, 1975. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/fedr.19750860506

[15] J. Born, “Schwefel, h.-p., numerische optimierung von computer-modellen mittels der evolutionsstrategie. mit einer vergleichenden einführung in die hill-climbing- und zufallsstrategien. (isr 26) basel-stuttgart, birkhäuser verlag 1977. 390 s., sfr. 48,–.” *ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik*, vol. 60, no. 5, pp. 272–272, 1980. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/zamm.19800600516