Utilizing state-of-art NeuroES and GPGPU to optimize Mario AI

An experimental performance evaluation

Hasse Lövgren

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden
This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Engineering: Game and Software Engineering. The thesis is equivalent to 20 weeks of full-time studies.

Contact Information:
Author(s):
Hasse Lövgren
E-mail: hala09@student.bth.se

University advisor:
Prof. Johan Hagelbäck
Dept. Computer Science & Engineering

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden
Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57
Abstract

Context. Reinforcement Learning (RL) is a time consuming effort that requires a lot of computational power as well. There are mainly two approaches to improving RL efficiency, the theoretical mathematics and algorithmic approach or the practical implementation approach. In this study, the approaches are combined in an attempt to reduce time consumption.

Objectives. We investigate whether modern hardware and software, GPGPU, combined with state-of-art Evolution Strategies, CMA-Neuro-ES, can potentially increase the efficiency of solving RL problems.

Methods. In order to do this, both an implementational as well as an experimental research method is used. The implementational research mainly involves developing and setting up an experimental framework in which to measure efficiency through benchmarking. In this framework, the GPGPU/ES solution is later developed. Using this framework, experiments are conducted on a conventional sequential solution as well as our own parallel GPGPU solution.

Results. The results indicate that utilizing GPGPU and state-of-art ES when attempting to solve RL problems can be more efficient in terms of time consumption in comparison to a conventional and sequential CPU approach.

Conclusions. We conclude that our proposed solution requires additional work and research but that it shows promise already in this initial study. As the study is focused on primarily generating benchmark performance data from the experiments, the study lacks data on RL efficiency and thus motivation for using our approach. However we do conclude that the GPGPU approach suggested does allow less time consuming RL problem solving.

Keywords: Reinforcement Learning, Evolution Strategies, GPGPU, Artificial Neural Networks
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Basic three-layered ANN</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Artificial neuron activation scheme</td>
<td>11</td>
</tr>
<tr>
<td>2.3</td>
<td>Hyperbolic tangent sigmoid function</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>RL problem example, Maze</td>
<td>13</td>
</tr>
<tr>
<td>2.5</td>
<td>General Evolutionary Algorithm flow-chart</td>
<td>16</td>
</tr>
<tr>
<td>2.6</td>
<td>Mario AI sensor grid</td>
<td>20</td>
</tr>
<tr>
<td>2.7</td>
<td>NDRange hardware abstraction model in OpenCL</td>
<td>24</td>
</tr>
<tr>
<td>3.1</td>
<td>High level architecture diagram</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>C++ Executable diagram</td>
<td>29</td>
</tr>
<tr>
<td>5.1</td>
<td>Graph of the average total execution time</td>
<td>41</td>
</tr>
<tr>
<td>5.2</td>
<td>Graph of the average MLP execution time per frame</td>
<td>42</td>
</tr>
<tr>
<td>6.1</td>
<td>Average total MLP execution time percentage</td>
<td>49</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Mario AI MLP type table ........................................... 21
3.1 Table of arguments .................................................. 30
4.1 Ten distinct experiment setups .................................... 38
5.1 Prestudy results ....................................................... 40
5.2 Averaged total execution time of experiments .................. 42
5.3 Averaged MLP execution time per frame ......................... 43
5.4 Accumulated reward from primary experiment ................. 44
6.1 T-test between CPU and GPU on 'total execution time' data 47
6.2 T-test between CPU and GPU on 'MLP execution time' data 50
Chapter 1

Introduction

The research conducted in this study is targeted at the Artificial Intelligence (AI) branch called Machine Learning (ML) and more specifically the types of ML called Reinforcement Learning (RL) and Evolutionary Learning (EL). Any unfamiliar or technical term encountered in this introductory paragraph will be thoroughly explained in section 2.1 Background.

By experimental performance evaluation, this study attempt to provide statistical data with which to demonstrate the somewhat overlooked potential of utilizing Graphics Processing Units (GPUs) in solving RL problems faster and more efficiently on inexpensive PCs.

To accomplish this, a direct policy search approach to RL is taken, while value function approaches are rejected. A policy is basically a potential solution to the RL problem. Using a direct policy search instead of a value function approach roughly means that instead of estimating expected rewards and following a path based on maximum expected reward, as one would using a value function approach, one instead tries different policies, searching directly for a solution in the policy space, and analyze variables that are more important or dominant given a certain situation. A direct policy search method would then instead make modifications to those variables and test another, similar, policy to see if the reward is higher or lower, thus deeming the modifications made as good or bad. RL and direct policy search is explained in greater detail in subsection 2.1.4. The reason for using direct policy search in this study is that it allows for simple batch learning.

Batch learning is a necessity for efficient GPU utilization as GPUs implement a very different architecture than CPUs, which immensely favor parallelism through the principle of Single Instruction Multiple Data (SIMD). Studies have shown GPUs to be significantly faster at propagating Multi Layer Perceptrons (MLPs) than traditional CPUs [13, 15]. An MLP is a feed-forward Artificial Neural Network (ANN) and is used as a representation of policies in this study. However, most studies are concerned with propagating one single MLP for Supervised Learning tasks where the MLP tend to be larger and more complex (i.e. have a higher problem dimension) than what is recommended for most RL cases. In
Chapter 1. Introduction

In this study, the proposed solution will instead utilize GPGPU to propagate several smaller MLPs in a batch in parallel. Each MLP in this batch represents one specific policy and also one individual in a population needed by the chosen policy search method.

The favored policy search method adopted in this study is an Evolution Strategy (ES) called the Covariance Matrix Adaptation ES (CMA-ES). An ES is a kind of Evolutionary Algorithm (EA) that is designed to optimize variables after how nature optimize life using evolution. Hence, an ES has a population of individuals that are selected to survive or die depending on their performance in a certain task. The surviving individuals breed new individuals that are similar to their parents. This evaluation, selection and breeding is done in iterations called generations and will over time evolve the population as a whole, producing individuals that are optimized in performing a certain task. In this study, the CMA-ES will be used as a Neuro evolution Strategy (NeuroES) to evolve and optimize the weights of the MLPs, thus searching directly in policy space for better solutions. An ES is called a NeuroES when it is applied to Neuroevolution, which it is when evolving weights in MLPs [10]. Hence in this study, when applied for RL, the CMA-ES will be referred to as the CMA-NeuroES. Note that nothing differentiate the CMA-ES and the CMA-NeuroES except for their areas of application, the internal algorithms still work exactly the same.

NeuroES and the CMA-ES are explained in more detail in subsections 2.1.5 & 2.1.6.

The CMA-ES was selected for its promising potential in optimizing solutions to noisy multimodal problems [6], which is just what many practical RL problems are. Noise is simply unwanted information that might disturb the learning and optimization of the algorithm. The CMA-ES has also been shown to outperform other direct policy search methods such as policy gradient methods (PGMs) when optimizing solutions in RL problems with much noise [7] [8]. In order for the CMA-ES to cope well with noise and multimodality it requires increasingly larger population sizes as the size of the problem dimension increase. Problem dimension simply translates to MLP size in ANN-based RL agents. This fact goes well in hand with the requirements of efficient SIMD, which is another very important reason for selecting the CMA-ES.

The CMA-ES motivates the use of large enough population sizes to be able to efficiently utilize a GPU, assuming the problem dimension of the RL-problem is high enough to require it. Thus, in order to perform valid performance tests and extract valuable data, a multimodal and noisy RL benchmark, operating in high problem dimensions, is required. Common RL benchmarks such as the (double) cart-pole or the mountain car have rather small problem dimensions and are easily solved with any in a range of RL approaches, without the need for batch-learning and GPU utilization. A more difficult, noisy and multimodal benchmark,
operating in high problem dimensions, is Togelius's Mario AI benchmark, an extension of Markus Persson's Infinite Mario Bros. The Mario AI benchmark is thus exceptional for the purpose of this study.

In short, this study's main focus is to compile and analyze performance data on one CPU-only and one GPU-aided solution by performing repeated Mario AI benchmark tests. To do this, however, the two solutions must also be implemented. Both solutions will use the exact same code when running the Mario AI benchmark and updating the MLP weights with the CMA-ES (i.e. optimizing or "learning"). The only differing piece of code is the MLP-propagating code where the CPU solution will propagate one MLP at the time and the GPU-aided solution will propagate all MLPs at once, in parallel, on multiple GPUs.

1.1 Problem Statement

Problem
The problem dealt with in this study is the fact that attempting to solve complex RL problems is a computationally intensive task that requires immense computational power and, consequently, a lot of time as well.

No matter what approach is taken, be it value function-oriented or direct policy search-oriented, training a RL agent to perform some complex task will require a large number of trial-and-error sequences to get it right.

Aim
The aim of this study is to provide statistical data that indicate that utilizing the computational power of GPUs could be a valuable remedy for the high computational intensity required when attempting to solve complex RL problems. While the use of GPUs will not decrease the number of necessary computations, it will unload the CPU and perform faster and more efficient computations, thus reducing the time required to solve a RL problem.

In order to compile such statistical data, and to help guide the research toward the aim, a number of concrete objectives have been established. These objectives, in order of execution, are the following.

Objectives

Prestudy: Implement a framework that allow the CMA-NeuroES to optimize MLP weights in the Mario AI benchmark. In this framework, implement a solution that propagates MLPs sequentially. Test the solution on the Mario AI benchmark in accordance with the test performed by Togelius et al. in [24]. This framework and sequential solution will be used as a test to verify that the CMA-ES can be successfully applied as a NeuroES to the Mario AI benchmark. Additionally the framework will be used in
future objectives to verify and test that the parallel GPU solution executes properly.

**GPU Utilization:** Implement a solution that utilize GPUs to propagate the MLPs for all Mario individuals of a population, in parallel every frame, as they traverse the level.

**Benchmark:** Design, setup and run experiments from which to extract statistical performance data from both the parallel GPU implementation as well as the sequential CPU implementation.

**Research questions**
The one primary research question this study attempt to answer concerns the relative difference in performance between a CPU-only and a GPU-aided implementation, following the ultimate aim of the study. However there are two secondary research questions associated with this study as well.

The secondary research questions are only concerned with the efficiency of using the CMA-NeuroES to solve the Mario AI benchmark problem, one of which is answered in the prestudy while the other can be answered with data acquired when performing experiments to answer the primary research question. These secondary research questions are of little relevance for the aim of this study but may strengthen some motives for choosing the CMA-NeuroES as the direct policy search method.

The research questions are defined as follows:

**Primary RQ**

**RQ1** What relative difference in performance, between a CPU and a GPU implementation of feed-forward MLP can be observed when attempting to solve the Mario AI RL problem with the CMA-NeuroES?

**Secondary RQs**

**RQ2** How well does the CMA-NeuroES perform in the Mario AI benchmark running the exact same test as described by Togelius et al. in [24]?

**RQ3** What correlation between population size, problem dimension and max fitness can be established when using the CMA-NeuroES to attempt to solve the Mario AI benchmark problem?

**Relevance**
The reason the results of this study are relevant is directly associated with the importance of time as a resource. Utilizing GPUs for computations may lead to increased time efficiency in solving RL problems. Increased time efficiency translates to solving the same RL problem, with equal results, in less time. Potentially
increasing the time efficiency could benefit fields such as games and robotics as well as other fields in which solving RL problem is a common task.

There are a few ways in which these fields could benefit from increased time efficiency. It could allow for the ability to solve slightly bigger and slightly more complex RL problems without consequently increasing the execution time. Or it could allow a solution to be more accurately optimized, perhaps leading to better results, without consequently increasing the execution time. Or at the very least, it could simply allow the solution to be found in less time than previously.

1.2 Report Overview

This section is dedicated to briefly explaining the chapters of this study to give a general overview of the report.

Chapter 1, Introduction has introduced the area of this study as well as the problem statement and general aim. It has also introduced the objectives to follow and the research questions to ultimately answer.

Chapter 2, Background and Related Work describes all the areas that are important to understand and appreciate the research conducted in this study. This chapter is very long and there are a lot of sections covered in sufficient depth. Any section in this chapter that the reader might feel they are already sufficiently knowledgeable about can be skipped. It also covers Related Work in a section, explaining what previous work this study is based on.

Chapter 3, Implementation is dedicated to explain the solution that is designed and implemented as an objective necessary to complete in order to answer the research questions and reach the aim of the study. This section is technical and does require the reader to be familiar with the sections covered in the Background chapter.

Chapter 4, Experimental Performance Evaluation describes the various experiments and experimental methods used in this study in order to properly answer the research questions. This is regarded as the primary method chapter, although the previous chapter, Implementation, is also partly a method required to reach the aim.

Chapter 5, Results presents the results from the experimental performance evaluations run in the manner defined in the previous chapter. This chapter is short and straightforward with only a few words disclosing important fact and explaining how to interpret the results.
Chapter 1. Introduction

Chapter 6, Analysis looks at the results with more depth and analyze what the data produced in the experiments actually tell us. The research questions are also answered in this chapter with regards to the analysis.

Chapter 7, Conclusion and Future Work finally concludes the entire study with some brief statements about the results, analysis, research questions and aim. Furthermore, future work is presented in the areas in which some uncertainties still remain or where improvements could be made.
Chapter 2

Background and Related work

2.1 Background

The following subsections will present essential background information, with adequate depth, in all areas that are crucial to understanding the research performed in this study. Any knowledge in any of those areas that is not essential to understanding what this study aims to demonstrate or answer, will not be provided. For such in depth, fundamental, knowledge the reader will be referred to suggested literature on which the specific section is based.

Any of the subsections can be skipped if the reader regards herself as well informed on the given subject.

2.1.1 AI & Machine Learning (ML)

Artificial Intelligence, or AI, is an extremely broad field containing very many subfields in a number of disciplines. Many of these subfields are further categorized into branches of that specific subfield. AI is, in rugged terms, the idea of man-made, constructed, intelligence; a vastly intriguing subject spanning many disciplines of academia. It is hard to capture the full scope of what AI is in a few sentences, therefore the focus will be exclusively on the AI subfield relevant for this study, namely Machine Learning (ML). For more information about AI, the book Artificial Intelligence: A Modern Approach, by Stuart Russel and Peter Norvig [18] has a short but excellent introductory chapter on AI that mention the philosophy and other foundations of AI.

ML is a broad field in and of itself, although being a subfield of AI. ML, like AI, is a field that is studied within several academic disciplines such as statistics, mathematics, engineering and computer science [19]. In this study, the relevant discipline is computer science and thus the information presented will be that of ML in computer science.

ML is similar to human learning which is based on interactions, feedback, memory and experience. When applying learning on machines such as computers, advanced and carefully designed algorithms are used. The algorithms are designed to adapt behaviour and improve based on experience.
ML is commonly categorized into four types, the two latter of which are especially relevant in this study and have their own subsections in this Background section. However the types often borrow methods and algorithms from each and are sometimes used in unison thus blurring the lines somewhat between them. The ML types are the following [16]:

**Supervised Learning**
In this category the learning agent is provided with a training set that include both the question and answer, i.e. inputs and expected outputs. Based on this training set the algorithms in supervised learning generalizes to respond correctly to additional (outside of the training set) inputs, thus having learned. This is the most common technique for ML and the premier choice in situations where large sets of complete training data is available such as, for instance, when trying to recognize handwriting [20]. However such is not always the case.

**Unsupervised Learning**
This category differs from supervised learning in that no answer, i.e. expected outputs, are provided to the learning agent. The agent is thus required to distinguish patterns in the input data and categorize it according to the patterns, thus having learned potential data categories. This kind of learning is commonly used on data with presumed, but maybe not apparent, clusters.

**Reinforcement Learning**
Reinforcement learning is similar to both supervised and unsupervised learning in the sense that it receives feedback in a manner similar to supervised learning with the exception that the feedback is not specified in advance through a training set but rather learned through repeated interaction and experience. Reinforcement learning has been successfully applied to a number of fields, and has most notably been applied to games[23] and robotics[14].

**Evolutionary Learning**
The fourth category is modeled after biological evolution. The idea is to breed offspring based on fitness of individuals to optimize behaviour or structure or what ever necessary. The individuals are placed in an experimental environment and their performance is measured through a fitness function. The individuals with the best fitness is selected to "breed" and the next population of individuals are based on this elite group that achieve the best fitness score in a specific task.

In this study we will use an NeuroEvolution Strategy (NeuroES), which is a optimization technique belonging to the Evolutionary Learning category, to solve
a Reinforcement Learning problem. What this means and how it is accomplished is explained in the following subsections.

2.1.2 Artificial Neural Networks (ANN)

An ANN is, as the name suggests, a network connecting a cluster of artificial neurons. ANNs are modeled after how natural brains work with interconnected brain cells (neurons) firing signals to one another in a vastly complex network to make out patterns from sensory input. However, unlike natural brains, consisting of several billion neurons, an ANN seldom consist of even a thousand artificial neurons. A neuron can be in two states, active or inactive, just like a bit can be 0 or 1.

![Basic three-layered ANN](image)

Figure 2.1: Basic three-layered ANN.

The circles represent artificial neurons in their respective layers. Each arrow represents a unique connection weight between two neurons in consecutive layers. Note that a neuron is connected to every single neuron in the subsequent layer.

ANNs have one input layer of neurons, one or several hidden layers and finally one output layer. Each pair of consecutive layers (e.g. input-hidden, hiddenX-hiddenY, hidden-output) are connected through weighted connections where every neuron in layer A is connected to every neuron in layer B, each connection represented by its own unique weight, see Figure [2.1]. Please note that not all
kinds of ANNs work and look like this but the lesser complex ones, which are of relevance in this study, do.

In short, ANNs will first get input in one end, the input layer. The neurons in the input layer then fire some of the neurons depending on what the input looks like, which in turn may cause the next layer to fire some of its neurons. This continues for every layer until finally an output is produced at the other end of the ANN, the output layer. Thus the output will be different depending on the given input. A more detailed explanation of the inner workings of the kind of ANN used in this study, is presented in the next subsection, 2.1.3, The Multi-Layer Perceptron.

One of the key aspects of ANNs are their ability to adapt behaviour over time to produce different output given the same input as earlier, in order to ultimately improve on their behaviour. This means an ANN can, with enough time, be optimized or trained to act in a predetermined fashion. To modify the behaviour of an ANN the only necessary action is to alter the weighted connections between neurons. The most common way of doing this is called backpropagation. Backpropagation roughly means traversing the ANNs connection weights from back to front, updating the weights it traverses based on some error value calculated between the given output and the expected output. Backpropagation is not used in this study and is therefore irrelevant which is why no thorough explanation is presented. In this study we will instead attempt to adapt the behaviour of ANNs by modifying their connection weights using NeuroEvolution Strategies, more on this in subsection 2.1.5.

ANNs are used in RL as a function approximator. This means that the ANN serves to approximate the best output to produce given a limited number of input neurons. Due to the limitation of input neurons, only a finite (and rather small) number of distinct states can be represented. A state in this case is the array of input neurons altogether. For example, if an ANN has eight input neurons it can represent 256 ($2^8$) distinct states. More on why function approximators are necessary and how they are used to solve Reinforcement Learning problems in subsection 2.1.4. For a more in depth, mathematical and algorithmic explanation on ANNs, please refer to chapter 18.7 in [18].

2.1.3 The Multi-Layer Perceptron (MLP)

There are a number of different types of ANNs such as Recurrent Neural Networks, Modular Neural Networks and Feedforward Neural Networks to name a few. The feedforward neural network is arguably the least complex type of ANN. One kind of feedforward neural network is the Multi-Layer Perceptron (MLP), which is the kind used in this study. The MLP works like the simple network briefly explained in subsection 2.1.2, ANN, that propagate from one layer to the next, beginning at the input layer, propagating layers sequentially in order, until some output is produced in the final output layer. It is called a feedforward network because it
propagates in one direction only and contains no loops or feedbacks. This means information from layer B will only be fed into layer C and never to its own layer or the preceding layer A. The name Multi-Layer Perceptron stems from the fact that it has multiple layers and the algorithm is based on the perceptron algorithm, invented by Frank Rosenblatt [17].

When propagating an MLP to produce output from some arbitrary input you have to start at the input layer. The input layer is a little bit different than the other layers of an MLP as the neuron states (active or inactive) in this layer are only dependent on some given information. Such information is commonly extracted from the state of the problem which is to be solved. The information is represented by an array of real-valued ones or zeros in which each element corresponds to a single neurons state in the input layer. If the first value is 1, the first neuron in the input layer will be regarded as active while it would be regarded as inactive if the value was 0.

The neuron states in all subsequent layers, that is all hidden layers and the output layer, will be determined based on the neurons’ states in the directly preceding layer. For these layers, all neurons’ states are determined by calculating the sum of all neuron and connection-weight products, separately, that are connected to the particular neuron for which the state is to be determined. This means inactive neurons will not have an impact on any other neuron since the product of its weighted connection value and zero will be zero \((x * 0 = 0)\). This summation of weighted neurons is called the transfer function and is depicted in Figure 2.2

![Figure 2.2: Single artificial neuron activation scheme. The steps that determine the state of one neuron.](image)

Directly preceding the transfer function is the activation function. The purpose of the activation function is to determine whether or not the neuron will fire
or not. A commonly used activation function is the hyperbolic tangent sigmoid and is depicted in Figure 2.3. The value from the transfer function is fed into the activation function to finally determine the given neuron’s state. To supplement the activation function, a threshold value is added. The threshold simply determines what value ultimately needs to be exceeded in order for the neuron to fire, thus becoming active. Would the input from the transfer function, after being fed through the activation function, not exceed the threshold, the neuron would remain inactive.

![Figure 2.3: Hyperbolic tangent sigmoid function.](image)

Finally, it should be mentioned, that for algorithmic purposes, one so called bias-weight is always included when computing a neuron’s activation beyond the neuron’s connection weights. This bias can simply be viewed as a neuron from the preceding layer that is always firing, independently of input or other neurons’ states.

For more on MLPs, please refer to chapter 18.7 in [18] and/or chapter 3 in [16].

### 2.1.4 Reinforcement Learning (RL)

The following subsection is a very simplified and narrowed down explanation of RL. RL is in and of itself, like mostly anything related to ML, a grand subject with many concepts, methods, algorithms and technical terms associated with it. The purpose of this subsection is to briefly introduce the concepts that are necessary to understand and appreciate the work done in this particular study. The book Reinforcement Learning: An Introduction [21], on which this entire
subsection is based, is recommended reading for anyone interested in a more in
depth, mathematical and algorithmic understanding of RL.

RL follows a simple framework with three utterly essential elements: states, ac-
tions and rewards. Through interactions between said elements, using clever
algorithms and methods, RL agents attempt to solve problems, i.e. achieve a
long-term goal. What this means is that a RL agent is given a long-term goal to
achieve within an environment of which it may initially have little or no informa-
tion. With the long-term goal as the strict aim, the agent must interact with and
learn about the environment in order to adapt its behaviour to finally, at some
point, achieve said goal. It quickly becomes apparent that such RL problems can,
in many cases, be very complex and hard to solve. A rather simple and classic
example is to have an agent solve a maze, as depicted in Figure 2.4

In the maze example, the entire picture represents the environment while
each square represents a state. The ability to move between squares, i.e. moving
up, down, left or right, would represent the actions and performing one of the
actions would mean a state transition. Finally the rewards are represented by red
and green colored squares, which means the agent is only given a reward when
reaching one of the states associated with a reward.

![Figure 2.4: Example of a simple RL problem.](image)

**Markov Decision Processes**
The example RL problem depicted in Figure 2.4 is also a Markov Decision
Chapter 2. Background and Related work

Process (MDP). An MDP is a central concept in RL as it enables mathematical modelling of decision making in stochastic environments. If an environment is stochastic it means that uncertainty exists when performing actions and that any action has a probability of leading to any state. MDPs include the three key elements of RL: states, actions and rewards, and also a probability feature in order to model such decision making.

An MDP consists of four sets, \((S, A, R_a, P_a)\), that together make up the entire MDP and contain all the information necessary to find an optimal solution. \(S\) denote the set of all possible states, \(s \in S\). \(A\) denote the set of all possible actions, \(a \in A\). \(R_a(s, s')\) denotes the set of all expected rewards to be received when transitioning from any state \(s \in S\) to any other state \(s' \in S\). \(P_a(s, s')\) denote the set of all probabilities that taking any action \(a \in A\) in any state \(s \in S\) will lead to any other state \(s' \in S\).

Any MDP where the rewards (\(R_a(s, s')\)) and probabilities (\(P_a(s, s')\)) are unknown can be called a RL problem, solving such an MDP will consequently also solve the RL problem.

To find a solution to an MDP is to find the optimal policy \(\pi(s)\). A policy is a decision maker function for the MDP that produces an action \(a \in A\) to perform given any state \(s \in S\) from the set of all possible states \(S\). The optimal policy is the one that returns optimal reward when traversing the MDP, that is the policy that generates more reward than any other. It is clear from this that any MDP can have a set of policies that produce different actions given different states which would naturally return different amounts of reward.

Function Approximators

Modelling an MDP based on a RL problem can be achieved by storing the sets \((S, A, R_a, P_a)\) in organized look-up tables. However these sets, and consequently the tables, do very quickly become unmanageable in size if the environment is large and/or the number of possible actions are many. For such complex problems, function approximators are commonly used to attempt to generalize the sets. A common function approximator, used in this study to approximate policies, is Artificial Neural Networks (see subsection 2.1.2). Using an ANN as a policy means that the ANNs input neurons are represented by a state \(s \in S\) and the output neurons will represent an action \(a \in A\) to perform in state \(s\). One specific policy's unique representation is an array of ANN weights. Adapting (or learning) to perform the right actions given a specific state means altering those weights to produce satisfying results that ultimately solve the RL problem.

Methods

When solving RL problems and MDPs there are a few commonly mentioned approaches. Arguably the most common and researched approaches are the value function approaches. Value function approaches tries to find the optimal policy
by continuously estimating and updating a set (or function) of expected reward returned per state $V^\pi(s)$ or state-action pair $Q^\pi(s, a)$ for some policies, this is done using some clever algorithms that will not be discussed. Using this expected reward an agent can make qualified choices on which state or state-action pair is most desirable from its current state. Some examples of value function methods are the Monte Carlo method and the Temporal Difference method. Temporal Difference can solve RL problems continuously while Monte Carlo can only solve RL problems episodically. In very rough terms continuous learning means that an agent could in theory solve the RL problem in only one very long trial while episodic learning requires the agent to execute a new trial at regular intervals when encountering a stop-criteria, such as reaching a goal or running out of time. Value function approaches are merely mentioned here because of their major significance to RL in general and in order to differentiate it from the direct policy search approach used in this study. However, they are not relevant for the work done in this study other than for background purposes. The interested reader is referred to [21].

Another approach, which is used in this study, is called direct policy search. As the name suggests, this approach boils down to simply searching for an optimal policy directly in the set, or in a subset, of all possible policies in the policy space. This is, however, somewhat more of an optimization rather than learning but it can still be efficiently used to solve RL problems and MDPs [22, 7]. Common direct policy search methods are gradient-based, and are called Policy Gradient Methods (PGMs). However in this study, a particular kind of evolutionary algorithm called an Neuroevolution Strategy (NeuroES) is used as a direct policy search method.

2.1.5 Neuroevolution Strategies (NeuroES)

In order to understand what a NeuroES is and how it works, a few preliminaries are required. As NeuroES is a kind of Evolution Strategy (ES) and an ES in turn is a kind of Evolutionary Algorithm (EA), both of those concepts will be explained first in proper order. The information in this section is based heavily on the book on AE Eiben and JE Smith; Introduction to Evolutionary Computing [3].

**Evolutionary Algorithms (EA)**

In the computer science spectrum of ML, a kind of optimization based on evolution exists. It is called Evolutionary Computing (EC) and belongs in the Evolutionary Learning category of ML (explained in subsection 2.1.1). Finally, as a subset to EC, there exists a few Evolutionary Algorithms, or EAs. Some examples of EAs include Genetic Algorithms, Evolutionary Programming, Genetic Programming and Evolution Strategies [3]. What all these algorithms have in
common is that they are all fundamentally designed and implemented to resemble biological processes observed in nature, or more specifically, evolution. The main principle of evolution is natural selection and is commonly explained as 'survival of the fittest'. EAs are designed on this very same principle.

In an EA, a population of several individuals is spawned, or initialized if it is the first iteration. Each individual in the population is evaluated to produce a fitness, which is a measurement of how well adapted an individual is to solve its given task. An individual can be represented by anything from a single number to a complex RL agent and the evaluation function can, similarly, be of entirely contrasting sophistication. For instance the evaluation can be as simple as a linear equation from 0 to 255, where the individual get a higher fitness when closer to 0, or as sophisticated as rewarding a complex RL agent with higher fitness for using proper grammar and semantics when communicating. When representing an individual it is important to differentiate the genotype from phenotype. A genotype is the encoding of an individual used by the EAs while the phenotype is the "real world" representation of a genotype. For example, the phenotype is a number between 0 and 255 in the "real world" while it would likely be represented as an eight bit array for the EA, thus the numbers genotype is an array of bits. As for the more complex example, the communicating agent, the phenotype is the agent itself while the genotype could be for instance an array of real values representing every single weight in an ANN. The representation and evaluation function is thus entirely based on the problem and environment to which the
After evaluating the entire population, some individuals are selected to become parents to offspring, the new individuals of the population. The selection of these individuals is called the parent selection mechanism and its purpose is to select the most appropriate parents for the offspring. Always selecting the parents with the best fitness might not be the best strategy as that might lead to over-adapting and getting stuck in what is called a local optima. A local optima can be explained as a pseudo-optimal state where the population is maximally optimized in a very narrow area and thus never evolves to explore additional options that could further maximize their fitness. After selecting the parents, the offspring is created by mutation and recombination. Mutation means that the offspring is created very similar to its parent but with some slight alterations to its genotype. In a recombination, two parents’ genotypes are merged into one or two offspring genotypes, similar to natural breeding. These offspring are then also evaluated to produce their own respective fitnesses.

At last, the population is reduced to its original size by survivor selection. Selecting the survivors of the population is done to sort out the fit from the lesser fit individuals, like natural selection. The fit individuals survive and go on to form the population of the next generation while the unfit does not and are removed entirely from the algorithm. From this stage, the EA starts looping. It loops until some condition is met, for instance an average fitness of the population exceeds a threshold, or the number of maximum iterations has been exceeded. Note that this means that it is the population as a unit, and not the individuals, that is actually being optimized. Individuals, after being spawned from mutation or recombination, are merely kept or replaced, never altered. The general process of an EA is presented as a flow-chart in Figure 2.5 and as a pseudo-code algorithm in Algorithm 1.

**Algorithm 1:** General EA as pseudo-code

```
Data: Population genotypes
Result: Adapted population genotypes
1 randomly spawn initial population;
2 set termination conditions;
3 while no condition is met do
4    evaluate individuals in population;
5    select appropriate parents;
6    mutate and recombine parents;
7    spawn new individuals/offspring;
8    evaluate new individuals;
9    select surviving individuals;
10   population = surviving individuals;
11 end
```
Evolution Strategies (ES)

An Evolution Strategy is much like an EA as it utilizes an evolution-like loop and operates on entire populations. What differs an ES from other EAs is mainly the distinct way in which an ES performs mutation.

In an ES, the genotypes of individuals are mutated based on a normal distribution that requires two parameters, mean and variance, \( N(\xi, \sigma) \). The genotypes are represented as a real-valued vector (\( \tilde{x} = \langle x_1, \ldots, x_n \rangle \)) in ESs. The mean parameter (\( \xi \)) is always set to 0 in ESs. The variance parameter (\( \sigma \)) is also referred to as step-size when working with ESs. When mutating a genotype each element in the real-valued vector is simply incremented by a value attained from the normal distribution \( N(0, \sigma) \), as such:

\[
\tilde{x} = \langle x_1, \ldots, x_n \rangle \quad x'_i = x_i + N(0, \sigma)
\]

Interestingly, this step-size parameter (\( \sigma \)) is part of the genotype itself in an ES. Thus the step-size undergoes mutation and recombination as well. This is referred to as self-adaptation and is the signature of ESs. Self-adaptation roughly means that a population will not only become increasingly adapted to its environment with every iteration but it will also become increasingly good at producing valuable offspring. In the most simple case, any genotype contain only one step-size element but they can also contain \( n \) step-size elements, as many as the dimensions of the basic genotype vector.

1-step: \( \tilde{x} = \langle x_1, \ldots, x_n, \sigma_1 \rangle \)  
\( n \)-step: \( \tilde{x} = \langle x_1, \ldots, x_n, \sigma_1, \ldots, \sigma_n \rangle \)

If the genotype is \( n \)-step, each basic element (\( x \)) is incremented by a value from a normal distribution with its corresponding step-size value. The mutation instead looks as follows:

\[
\tilde{x} = \langle x_1, \ldots, x_n, \sigma_1, \ldots, \sigma_n \rangle \quad x'_i = x_i + N(0, \sigma_i)
\]

When mutation occurs on a genotype, the step-size elements are mutated first by a different equation, after which the basic elements are mutated using the newly mutated step-size element.

A more in depth explanation can be found in chapter 4 of Introduction to Evolutionary Computing [3].

A Neuroevolution Strategy (NeuroES) is an ES used for Neuroevolution purposes [11]. Neuroevolution is a term used in Evolutionary computing when an evolutionary algorithm is used to train or optimize ANNs. This means that a NeuroES works in the exact same way as a standard ES but the basic elements of the genotype vector (\( x \in \tilde{x}_i \)) are all represented as weights in an ANN as a phenotype.

Thus, when using an ANN to approximate policies in RL 2.1.4, the set of all these policies can be directly searched for a better solution using NeuroES. Any policy in this set would be represented by a unique real-valued vector. This
Chapter 2. Background and Related work

means that NeuroESs can be used as a direct search policy method for solving RL problems. It has shown potential in a number of studies [7, 8, 9, 10].

2.1.6 Covariance Matrix Adaptation ES (CMA-ES)

The CMA-ES is a state-of-art ES designed and developed by Nikolaus Hansen. It works similar to other ESs but has been extended with a covariance matrix to allow for what is called correlated mutations. It is also, unlike other ESs, completely derandomized. In rough terms, the covariance matrix allows for better adaptation of the step-size parameter, which in turn allows for better and more efficient adaptation of the population as a whole. Note that the CMA-ES can, like any other ES, be used as a NeuroES. It is then referred to as the CMA-NeuroES [10]. As the technical details of exactly how the algorithm works is vastly redundant for this study, a few references to studies using the CMA-ES in practice will be provided instead.

The CMA-ES has been successfully implemented in various studies, some notable ones with regards to Reinforcement Learning are: [7, 9, 10, 12, 19]. The CMA-ES has also been thoroughly tested and compared to some other algorithms [6, 11, 8].

A thorough study [6] shows the correlation between population size, dimensionality of the problem and efficiency in approximating multimodal functions. It is shown that the CMA-ES deals with multimodality in higher dimensions more efficiently with increased population sizes.

For the technically and algorithmically interested, a fairly straightforward tutorial of the CMA-ES has been compiled by Hansen [5].

2.1.7 Mario AI

Mario AI is a RL benchmark developed by Togelius et.al. [24]. It is an extended version of Markus Persson’s Infinite Mario Bros (IMB) and is developed in Java. Both IMB and Mario AI are public domain which means that all resources, both source code, graphics and other media are publicly available for non-commercial use.

IMB is a clone of the well known game Super Mario Bros and use the same game mechanics. The goal of IMB is to traverse the level from the start at the far left all the way to the goal at the far right. To reach the goal, Mario must run or walk rightwards while jumping over gaps and hills while also avoiding the enemies scattered across the level. Additionally, Mario has to reach the goal before the 200 second time limit has run out. The challenge of reaching the goal is also the RL problem that constitute the foundation of the Mario AI benchmark.

https://mojang.com/notch/mario/
One of the key features of IMB is the random level generation that creates different levels (environments) based on a random seed and a level difficulty value. For the Mario AI benchmark, this enables the RL problem to be more generalized. This means that the RL problem need not just be defined as being able to finish one particular, static, level but rather be proficient in finishing any level of a certain difficulty, regardless of differences in environment.

Mario AI extends IMB by allowing easy implementation of AI controllers that will automatically control Mario in the game. These AI controllers are used in AI agents that can probe the environment for information in order to produce an action. While Mario AI allows many different kinds of AI controllers, this subsection will only focus on the MLP-based controllers and how they work since all other kinds are irrelevant to this study.

The environment surrounding Mario can be probed using differently sized sensor grids. The sensor grid extends in all directions with Mario in the center grid square. There are grids of three sizes in Mario AI, Small, Medium, and Large. The small sized grid extends one square in all directions, forming a box of nine squares including the one occupied by Mario. The medium sized grid extends an additional square in all directions and the large sized grid extends it one further. The three different grid sizes are color coded in Figure 2.6.

The sensor grids are used to probe the environment surrounding Mario for enemies and terrain. In order to probe for both enemies and terrain, two grids...
are necessary, one for the terrain ($\tilde{t}$) and one for the enemies ($\tilde{e}$). Note that coins are not probed for and is thus ignored. These grids are represented by an array of booleans (True/False) where each square is one element in the array. If any square of the grid is occupied by terrain, the corresponding boolean element in the array will be set to True, while if the square is unoccupied the element is set to False. Additionally, Mario’s current state can be fetched to find out whether Mario is on the ground and whether he can jump or not. Both of these attributes are, like any square in the grid, represented by a boolean value of True or False. As an example, the small terrain and enemy grids in Figure 2.6 as well as Mario’s state, would be represented as follows:

$$\tilde{t}_s = \langle False, False, False, False, False, True, True, True \rangle$$

$$\tilde{e}_s = \langle False, False, False, False, False, False, False, False \rangle$$

$$\text{isMarioOnGround} = True \quad \text{mayMarioJump} = True$$

For each size of grid (S, M, L) there is a corresponding MLP agent. An MLP agent of type Small uses sensor grids of type Small. The corresponding sensor grid sizes applies to Medium and Large agents as well. An MLP agent is simply an agent that use a feed-forward MLP (see subsections MLP 2.1.3 & RL 2.1.4) to decide what action ($a \in A$) to take given a specific state ($s \in S$). A state ($s$) in such an agent is represented by the two grid arrays, $\tilde{t}$ and $\tilde{e}$, as well as Mario’s ground and jump state. Note that the last element of the state representation is the bias for the MLP and is always set to 1.0.

$$s = \langle \tilde{t}, \tilde{e}, \text{isMarioOnGround}, \text{mayMarioJump}, 1.0 \rangle$$

From this, it is clear that the size of the states will differ since $\tilde{t}$ and $\tilde{e}$ varies in size depending on the MLP agent type. Naturally, this also means that the input layer of the MLP in the agent will be different depending on the type. Note that the boolean elements of the state representation is converted to floating points (1.0 for True and 0.0 for False) before making up the input layer of the MLP. As an additional consequence, the different types of MLPs will have different number of weights and thus operate in different problem dimensions. Other than that, nothing differentiate the MLP agents from one another, they work exactly the same and all have ten neurons in one hidden layer and 5 neurons in the output layer. Table 2.1 shows the type of MLP agents’ important numbers.

<table>
<thead>
<tr>
<th>MLP agent type</th>
<th>grid size</th>
<th># neurons</th>
<th># weights</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$t$</td>
<td>$\tilde{e}$</td>
<td>$\text{in}$</td>
</tr>
<tr>
<td>Small</td>
<td>9</td>
<td>9</td>
<td>21</td>
</tr>
<tr>
<td>Medium</td>
<td>25</td>
<td>25</td>
<td>53</td>
</tr>
<tr>
<td>Large</td>
<td>49</td>
<td>49</td>
<td>101</td>
</tr>
</tbody>
</table>

Table 2.1: Mario AI MLP agent numbers by type.
For the MLP agent to move Mario toward the goal, the MLP need to propagate its inputs (the state representation) through the neural network, thus deciding what action to take.

In Table 2.1, the number of output neurons is set to five (5) for any MLP type. This is interpreted as one action per firing output neuron. These five actions correspond to actions that can be performed on a classic NES controller in the Super Mario Bros. game. Thus, the output neurons correspond to the controllers buttons: Left, Right, Down, A and B. Pressing Up does nothing in Super Mario Bros and is thus excluded from the actions list. Note that Mario can perform multiple actions at once, thus the output layer can have more than one firing neuron at the time.

The actions will do the same things their corresponding buttons would in the original Super Mario Bros. game. Left will make Mario go left, right will make Mario go right and down will make Mario duck. The A button initiates a jump and the jump will be higher depending on if the button is held or not. The B button will shoot fire (if possible) and initiate running mode. Mario will continue to run as long as the B button is held.

The MLP agents propagate their MLP once every frame in order to produce an action to take. The Mario AI updates 15 times for every "in game" second which means that with a 200 second time limit, there are a maximum of 3000 frames in any trial. Of course the number of frames can be a lot fewer if Mario falls into a pit, dies by an enemy or finishes the level.

If an agent successfully traverse an entire level and reach the goal, this agent can be regarded as having completed the RL problem that is Mario AI. To make the RL problem more generalized and a lot harder however, the agent should be able to complete a majority of the levels it encounters.

In addition to the AI controllers just explained, Mario AI also allows for data gathering. The most important data to extract from the Mario AI benchmark is the "distance traveled" value as that is the most important value when it comes to measuring how well a RL agent has performed. This distance value can be used as a fitness value in any Evolutionary Algorithm, including Neuroevolution Strategies.

2.1.8 General-Purpose GPU (GPGPU)

General-purpose computing on Graphics Processing Units, or GPGPU, is a state of art technology that allow developers to harness the power of many GPU cores simultaneously. GPGPU applications execute in massive parallelity on modern graphics cards that have a vast array of compute units, i.e. cores[2]. The GPGPU system's architecture is inherited from the common graphics card architecture which relied on multiple processors to calculate pixel values for games and renderings. However it has now been generalized in a way that allow for a broader
range of applications to harness the GPU power through fully programmable cores. To access the cores a developer has to use an API such as CUDA, OpenCL or Direct Compute. All three of these APIs work fundamentally the same and have a similar framework.

Below is a description of the primary components of the general architecture of the GPGPU model. The following components of the general architecture of GPGPU applies to both Direct Compute and CUDA as well as OpenCL but some technical terms are based on OpenCL wording since it is the API used in the implementation in this study. The information below is based on W. Hwu and D. Kirk's book Programming Massively Parallel Processors: A Hands-on Approach [11].

Host
In GPGPU the CPU is called the Host. It does not only refer to the CPU as in the hardware processor but more in terms of the application running on the CPU which at some time in execution access the GPGPU API (i.e OpenCL) to perform massively parallel execution. The host initiates all setup variables for execution, such as which device to use, how many work items should execute per work group, what kernel to execute and also manages memory to make sure the kernel execute on the correct data. All these terms will be explained below. The memory copy from Host to Device can in some cases be a major bottleneck as the memory buses are comparatively slow.

Device
The Device is an abstract name for any device capable of executing OpenCL code in parallel. Although it is possible to run OpenCL code on a multicore CPU, the device almost exclusively refer to a graphics card. On the device of choice is where the massively parallel executions will be performed. As different graphics cards have different hardware, i.e. more or less memory, few or several cores etc. this abstraction is of major importance for GPGPU. It also means that maximum efficiency will be achieved with different setup variables depending on the hardware, i.e. graphics card.

Kernel
The software running on the device is called a Kernel in GPGPU. It is coded in a C-like high-level syntax and compiled by the host before execution. This is where all instructions that will run on the device need to be expressed in kernel code. The kernel instructions (after compilation) are run simultaneously on all work items in any work group which mean that the code has to be very considerate of parallel work items to be optimally efficient. For instance, branching is usually not very good as those work items that do not branch need to halt execution and wait.
NDRange, work groups and work items
The NDRange, work groups and work items in GPGPU refer to components of the hardware abstraction API that is OpenCL. The components are depicted in Figure 2.7. The NDRange contain all work groups and each of the work groups have a multiple of work items. The setup variables initiated by the Host is what will determine the NDRange’s representation in terms of how many work groups will exist and how many work items each one such work group will have. When a kernel is set to execute, it is executed on the NDRange based on the NDRange setup.

The work groups of the NDRange are executed on any processor on the device and scheduling is entirely up to the device. Any device processor has a number of cores, it is on these cores the work items execute given instructions from the compiled kernel.

Each work item will be assigned a unique ID based on what work group it belongs to and its relative location in that work group. This ID is needed to distinguish for instance which memory address this particular work item is to execute on. Furthermore, the work items have access to different kinds of memory banks.

![Figure 2.7: NDRange hardware abstraction model in OpenCL](image)
Figure courtesy of W. Hwu and D.Kirk [11]

Each work item has its own private register space. This registry memory is very limited but very fast. Beside the registers, each work item also has a larger amount of local memory that is also private to each individual work item but significantly slower. For work items to be able to co-operate within the work
group, they have access to a *shared memory*. All work items, regardless of their work group also have access to a *global memory* which can also be accessed by the host. The global memory is where the data will exist initially as it is the only space the host can access. Data therefore has to be copied into the more efficient memory banks on the device, this is controlled by the kernel.

### 2.2 Related Work

The foundation of this study is based on Nikolaus Hansen’s work on the CMA-ES, Christian Igel and Verena Heidrich-Meisner’s work on the CMA-NeuroES as a direct policy search alternative and also the work of Julian Togelius et al. on the Mario AI benchmark. The following paragraphs inform of the work of these researchers and attempt to position this study in relation to said work [6, 5, 7, 8, 9, 10, 24].

In addition to assembling some of their work, i.e. using the CMA-NeuroES on the Mario AI benchmark, this study attempts to demonstrate the possibility to, and potential benefits of, integrating GPGPU with the CMA-NeuroES, especially for complex RL problems.

Nikolaus Hansen is the designer of the CMA-ES algorithm, on which he has also conducted many studies centered on its behaviour and performance.

One of those studies [6] is of much value in motivating this research as it demonstrates the CMA-ES’s ability to converge to a solution in multimodal problem environments, by increasing the population size ($\lambda$). Without that knowledge, there would be no point in conducting this research at all, as it is the primary motivator to using large enough population sizes ($\lambda$) to efficiently utilize GPGPU.

Additionally, Hansen has compiled a brilliant tutorial on the CMA-ES and how to practically use it [5]. On his personal page on the *Laboratoire de Reserche en Informantique* website[^1], source code for the CMA-ES is available in many different programming languages. The tutorial, as well as the source code, was unquestionably of great value when conducting this research.

Verena Heindrich-Meisner and Christian Igel have conducted some highly relevant research on using the CMA-ES as a direct search policy method for solving RL problems [7, 8, 9, 10]. Their research show that the CMA-NeuroES can, very efficiently, optimize solutions to noisy ANN-based RL problems.

Any direct policy search method could have been used instead of the CMA-NeuroES in this study to optimize solutions in a parallel manner. However, any other method would not be as easily motivated as the CMA-NeuroES is for a GPGPU-aided solution, due to its also very promising performance in multimodal problem environments in high dimensional spaces. As has been mentioned before,

[^1]: https://www.lri.fr/~hansen/
the GPGPU-solution need to operate on more complex (higher dimension) RL problems in order to efficiently utilize the computational power of the GPUs.

Thus the work of Heindrich-Meisner and Igel, along with the work of Hansen, strengthen the motivation to conduct the research in this study. Their combined work is primarily focused on the CMA-ES’ algorithmic efficiency as an ES and a Neuro-ES. Such efficiency is often measured in number of evaluations or number of iterations and is compared to other RL-methods or ESs.

This study has a different focus and is primarily concerned with measuring and ultimately increasing the computational efficiency of performing some of the calculations necessary in the CMA-ES in as little time as possible. Thus the measurement will be based on execution time and the comparison will be between different implementations of the very same algorithm, one sequential (CPU) and one parallel (GPU).

Julian Togelius et al. are behind the development of the Mario AI benchmark [24] in which the solution is to be tested. The benchmark is very important to this study as it is more complex than common RL benchmarks and can run on varying dimensions through different MLP sizes. Both of these key aspects of the Mario AI benchmark are prerequisites to being able to run comparative experiments between a sequential CPU solution and a parallel GPGPU solution.
Chapter 3

Implementation

In order to answer the research questions posed in this study, some quite extensive preliminary work is required. Most of this preliminary work consist of implementing and assembling the various parts that make up the solutions that the experimental performance evaluation is to be performed on. In accordance with the objectives, the majority of this implementation was done very early on in the study, during the Prestudy objective. Everything apart from the GPGPU-aided MLP-propagation explained in section 3.2 was implemented as part of the prestudy objective. The implementation was developed on and for the GNU/Linux OS.

The following sections will explain the development of the experimental framework, the additional GPGPU-aided solution as well as the scripts performing and assembling the experimental performance evaluation. The experimental method and the experiments along with explanatory statements are presented in the next chapter, Experimental Performance Evaluation.

3.1 Developing the Framework

3.1.1 High Level Architecture

Developing the complete framework means that all the different source code and APIs needs to be assembled. The primary components are the Mario AI benchmark, the CMA-NeuroES algorithm and its evolutionary loop as well as the MLP propagation. The MLP propagation part will have to utilize the GPGPU API OpenCL for the parallel solution that will be developed at a later stage. Consequently, an issue with code languages exists between these primary components as the Mario AI and the CMA-NeuroES is developed in Java while the OpenCL API is written in C++. This issue will be resolved by integrating the Java Native Language (JNI) library into the framework. Thus the framework will be a C++ executable that propagate MLPs and handles the evolutionary loop but also calls functions on, and shares data with a Java Virtual Machine (JVM) that handle the Mario AI and the CMA-NeuroES. For architectural reasons, both the sequential and the parallel MLP propagating components will be implemented in C++. This also mitigates any potential difference in execution time that might
had been present if one solution was written in Java and the other in C++.

Figure 3.1 shows the primary components of the experimental framework.

![Figure 3.1: High level architecture diagram of the experimental framework.](image)

As Figure 3.1 shows, some components execute within a JVM. The JVM is controlled by the C++ executable and in order to pass data and invoke function calls between the JVM and the executable, a JNI adapter was developed. The JNI adapter makes it possible for the C++ executable to invoke functions in the JVM as well as send data to it or request data from it. One half of the JNI adapter is written in C++ and the other half is written in Java. The functions on the C++ side of the adapter only serve as proxy functions to invoke their respective functions on the Java side of the adapter. The Java side JNI adapter then informs the various Java components to execute as they should. Invocations in the opposite direction, from Java to C++, is not possible nor necessary with this solution.

### 3.1.2 The C++ Executable

The C++ executable is the main application. It contains all components necessary to execute a single evolutionary run, producing a set of optimized weight values for an MLP that can be used to automatically control an agent in the Mario AI benchmark. Along with the weights, performance data measured in milliseconds (ms) are also produced. This subsection will focus on the evolutionary loop, the MLP-propagation and the benchmarking of various important sections of code.
A detailed explanation of the inner workings of the Java based components will not be presented. However the Mario AI benchmark and how any Evolutionary Strategy (including the CMA-ES) is used for Neuroevolution has already been explained in the Background and Related work chapter.

Figure 3.2 depicts a mid level view of the C++ executable. The very first, top left, component is initialization and setup. This component sets up both the C++ executable and the JVM for execution. The input arguments to this component are specified when starting the executable, if no arguments are supplied, standard values are used. The available arguments are listed in Table 3.1.
Once initialized, the evolutionary loop starts executing. The evolutionary loop is the core of the entire experimental framework as this is where the MLP weights are being optimized to complete the Mario AI RL problem.

During the evolutionary loop, a benchmarking component will keep track of the total execution time (in milliseconds) of the evolutionary loop, the total execution time of only the MLP-propagating section as well as the number of frames (in the Mario benchmark) that has been executed. The number of frames are also the number of times the MLPs have been propagated, which is why it is of value. These benchmarking values will make up the core data set for the results of the experiments.

The evolutionary loop will run until any stop condition is fulfilled. The stop conditions are either ‘--maxGen’, ‘--maxEval’ or ‘--stopGoal’. It may also terminate if the CMA-NeuroES experiences issues, these issues are explained in the Results (5) chapter. The evolutionary loop executes as follows (see Figure 3.2 for context):

"Get Next Population" retrieves new MLP weights for each and every individual in the population. All MLP weights are retrieved in one batch. This is done only once every iteration of the evolutionary loop as the weights does not change until the population has been evaluated and the CMA-NeuroES has been updated. Each unique set of these MLP weights is the genotypes of one Mario AI MLP agent.

"Evaluate population" is a loop in itself that iterates once for every frame of the Mario AI benchmark run. As explained in the background chapter on Mario AI (see 2.1.7), there are a maximum of 3000 frames due to the time limit in the Mario AI RL problem. In this loop, every Mario
Chapter 3. Implementation

instance set up in the JVM will update one frame per iteration. There are as many Mario instances as there are individuals in the population, namely as many as specified by `-popSize` (λ). Each phenotype Mario MLP agent instance has its own unique corresponding set of genotype MLP weights, as acquired in the previous task, "Get Next Population". Most of the tasks performed in this loop are executed in the JVM and merely invoked in the C++ executable. When "Get States" is invoked, the JVM probes the scene of every single individual Mario instance and returns all these states to the C++ executable. Each state, corresponding to the input neurons of their respective MLPs, are fed into the MLPs and propagated in the "Propagate MLPs" task, this is done in C++ and not in the JVM. During this task, depending on the `-gpgpu` argument, the MLPs will be propagated sequentially or in parallel. The implementational difference between the two solutions is presented in section 3.2. After propagating the MLPs, their resulting actions are sent to the JVM by invoking the "Set Actions" task. In the JVM, every Mario instance performs its given action and returns to the C++ Executable whether or not their Mario instance is still alive. If no Marios are alive, the evaluation is complete. If the evaluation is complete, the next task, "Update CMA-NeuroES", is performed, otherwise the "Evaluate population"-loop executes the next iteration.

"Update CMA-NeuroES" is rather self-explanatory and only invokes an update on the CMA-NeuroES in the JVM component. The effect of this is parent selection, recombination and mutation of the most fit individuals and the breeding of a new population of MLP weights, ready to be retrieved for the next iteration of the Evolutionary loop. If the CMA-NeuroES encounters a stop condition, the C++ executable will be notified and will invoke a "Save Data" command containing the benchmark data. It is then up to the JVM to make sure everything is properly saved. If this happens, the evolutionary loop is finished and after making sure everything is saved, the application will terminate. An optimized genotype consisting of a number of MLP weights has been created. The weights as well as the evolutionary run’s performance data is saved.

3.1.3 The Java Virtual Machine

The Java Virtual Machine (JVM) is a component owned by the C++ executable. Within this JVM, one single Java object exists for the C++ executable to interact with. This object is the JNI adapter. The JNI adapter in turn has a number of Mario AI benchmark objects as well as one CMA-NeuroES object. It is the job of the JNI adapter to handle these objects when commands are invoked from the C++ executable. Every Mario AI benchmark instance has its own Mario MLP agent as well as its own level environment in which the Mario MLP agent
attempts to reach the goal. The type of MLP agent, the level environments random seed value and the difficulty value is set by the input arguments --mlpType, --rndSeed and --difficulty. The CMA-NeuroES algorithm is set up by the values specified by the input arguments --weightFile, --popSize, --maxGen, --maxEval and --stopGoal. The latter three of which are only stop conditions, dictating when the evolutionary loop will stop executing and thus the CMA-NeuroES will stop updating. The first input argument is used to initialize the CMA-NeuroES with an already optimized set of weights for further optimization and the --popSize determines population size (λ), i.e. the number of individuals the CMA-NeuroES’ will select parents and survivors from. --popSize also determines how many Mario benchmarks will exist.

The two major components, the Mario AI benchmark and the CMA-NeuroES, are both merely integrated ‘as is’ into this framework and is the work of Hansen\[5\] and Togelius et. al. [24]. However some minor changes were made to the Mario AI benchmark in order to allow multiple instances of Mario AI to be run at the same time with different Mario agents and in different level environments. Furthermore the "native" MLP agents in Mario AI are not used seeing as the MLPs are propagated in the C++ side of the framework. Instead, a simple interface MLP agent is used that is supplied with an action for every frame through the JNI adapter rather than actually propagating an MLP itself. This action is of course received from the C++ executable after the MLPs have been propagated with the states as input, thus the change is non-existent in terms of behaviour.

### 3.2 Utilizing GPGPU

The MLP execution is the only major task executed in C++ except for the Evolutionary loop control. The way it is developed does have some issues like for instance that the MLPs are propagated even though the corresponding Mario agent is already dead and does not need any further actions. In the parallel solution this is a must, in the sequential solution this could have been avoided by just propagating the ones that are still alive. However, this solution does not, and that is something that needs to be taken into account when analyzing the results. For the sake of the MLP execution time difference, this is the fair comparison, as both solutions will propagate the same number of MLPs each frame. However, for the sake of the total execution time difference, the sequential version would most likely have benefited to some extent from not propagating unnecessary MLPs.

Just before the MLPs are propagated during a frame, all states are fetched from the JVM. The states are represented as one vector containing all inputs from

\[5\] https://www.lri.fr/~hansen/cmaes_inmatlab.html#java

\[24\] http://julian.togelius.com/mariocompetition2009/marioai.zip
every Mario instance. The number of states are the same as the population size, \( \lambda \). The size of each such state, i.e. the number of input neurons per MLP, is dependent on what type of MLP agent is used. The number of inputs can be either 21, 53 or 101, as is shown in Table 2.1 and explained back in the Mario AI background section (see 2.1.7). The states vector is represented as follows, with \( \lambda \) denoting population size, \( n \) denoting the number of inputs and \( x \) denoting an individual input neuron:

\[
\tilde{S} = \langle x_{11}, x_{12}, ..., x_{1n}, x_{21}, x_{22}, ..., x_{2n}, ......, x_{\lambda 1}, x_{\lambda 2}, ..., x_{\lambda n} \rangle
\]

**Algorithm 2: Sequentially propagating multiple MLPs**

**Data:** States vector \( \vec{S} \)

**Data:** MLP weights vector \( \vec{W} \)

**Result:** Actions vector \( \vec{A} \)

1. for \( i = 0; i < \lambda; i++ \) do
2.  state = \( \vec{S}[i] \); //Segmenting the state vector
3.  weights = \( \vec{W}[i] \); //Fetching corresponding weights
4.  //Propagating the state with corresponding weights
5.  actions = Propagate(weights, state);
6.  \( \vec{A}[i].append(\text{actions}); \) //Append actions to Actions vector
7. end

In the sequential MLP propagating solution, the states vector is segmented into \( \lambda \) states that are propagated one by one, sequentially, inside a loop from 0 to \( \lambda \). This solution is presented as pseudo-code in Algorithm 2.

What this does is ordinary feed forward propagation on a number of MLPs in a row. In contrast, the parallel solution uses no loop and simply propagate all the MLPs at once, as is shown in the very simple pseudo-code below.

**Algorithm 3: Propagating multiple MLPs in parallel**

**Data:** States vector \( \vec{S} \)

**Data:** MLP weights vector \( \vec{W} \)

**Result:** Actions vector \( \vec{A} \)

1. \( \vec{A}[i] = \text{Propagate}(\vec{W}, \vec{S}); \) //Propagating the entire batch at once

This parallel solution uses OpenCL for the "Propagate" function. Note that this solution lets OpenCL handle the number of work items per work group etc. The reason for that is to allow experiments to be run on different hardware devices and not have the setup specialized to a particular device. However, experiments in this study were only run on one device due to resource and time constraints. The parallel solution requires two different kernels. These two kernels only propagate the MLPs one layer. This means that the kernels will have to be run twice. The
first time for the initial layer, input to hidden. The second time for the final layer, hidden to output.

The first kernel is used to multiply the input values to their corresponding weights for all the MLPs. Each work item in this kernel will multiply one input value and one weight value in one of the MLPs. Since all weights need to be multiplied with an input neuron, the number of work items are $numWeights \times n \times \lambda$. $numWeights$ denote the number of weights in the MLP and is, like $n$, different depending on what type of MLP is propagated (see Table 2.1). The number of weights is calculated by multiplying the number of input neurons with the number of hidden neurons, and adding that number to the product of the number of hidden and the number of output neurons, e.g. $numIn \times numHidden + numHidden \times numOut$.

The second kernel has fewer work items. This kernel only has one work item for every neuron in layer B, times the number of MLPs ($\lambda$) of course. In this kernel, each work item adds together the multiplied weights that is connected to its corresponding neuron. Lastly each work item run the activation function, the hyperbolic tangent, on this sum. The first layer is now propagated and this entire process will be executed one more time on the final layer to produce the output.

The procedure is very similar to the sequential solution, which naturally also propagate one layer at the time. The main difference is that the parallel solution can use many parallel work items in order to speed up the process. However there is of course a loss in terms of overhead to copying the states to the OpenCL device for each frame. There is also overhead in setting/switching the kernels four times for each frame. A smaller, but perhaps notable overhead, is also introduced when copying the MLP weights to the OpenCL device once every iteration of the evolutionary loop.

The kernel code is presented in Appendix A. Note however that the kernels are compiled once it is known what type of MLP is going to be used, which means that some variables, like $numInput$, $numHidden$ and $numWeights$ are constants and entered into the kernel code just before compiling it. The kernels in the code presented in Appendix A are the two kernels for propagating the first layer, input to hidden.

### 3.3 Running Experiments

In order to automatically run multiple evolutionary runs, Python scripts were written for the two experiments. The only purpose of these scripts is to automize the entire experiment process. The scripts will not only execute the evolutionary runs (the C++ Executable) with the right parameters but will also be responsible for collecting and visualizing the experiment data that is produced.

When a solution needs to be tested on multiple levels with different level random seed and difficulty values, a Java jar file will be run. This jar file simply
executes one single run of a level based on three input arguments; where to load the weights from and two integer values for the difficulty and level random seed. This jar file can also be used to visualize a single Mario run in order to verify that the MLP agents are acting as they are expected to by using the -v argument. When the jar has finished executing it prints the MLP agent’s achieved fitness.

The python scripts as well as the C++ and Java source code are available online\textsuperscript{3}.

\textsuperscript{3}http://www.hal90.se/misc/source.tar.gz/
Chapter 4

Experimental Performance Evaluation

This chapter is dedicated to explaining the method by which the experiments of this study were conducted. In order to answer both the primary RQ1 and the secondary RQ3, a number of experimental performance evaluations had to be conducted. The experiments were conducted in the framework on the two comparable solutions; the sequential and the parallel. All experiments were conducted multiple times in order to get as accurate averages as possible and to eliminate as much noise in the data as possible. The purpose of the experimental performance evaluation is to generate statistical data that can be visualized in graphs or presented as tables, in order to analyze and draw conclusions based on it.

The variables of interest in these experiments are primarily problem dimension (MLP size/type), population size and execution time. A relation between execution time, MLP type and population size should be made clear from the experiments for both the sequential and the parallel solution. Furthermore, to answer the secondary RQs, the relation between fitness (accumulated reward), MLP type and population size should, if possible, be established.

Most experimentation was conducted during the "Benchmark" objective but a minor experiment was also conducted at the end of the prestudy in order to establish that the CMA-NeuroES can be used on the Mario AI RL problem. Conducting this minor experiment will also provide research data to answer RQ2.

All experiments were conducted in a GNU/Linux OS.

4.1 The Prestudy Experiment

The prestudy experiment was conducted right after the experimental framework had been developed. GPGPU utilization was not yet implemented. This experiment is based entirely on the experiment Togelius et.al. conducted on the Mario AI benchmark as an initial test [24]. The only thing differing between this experiment and Togelius’ is the choice of training algorithm. While they use a μ+λ NeuroES and the HyperGP[1] neuroevolution/genetic programming algorithm to train MLPs and Simple Recurrent Networks (SRNs), this experiment used the CMA-NeuroES to train MLPs only.
Chapter 4. Experimental Performance Evaluation

The experiment was conducted in the following way. The population size is 100 and one evolutionary run, i.e. one experiment trial, evolves for 100 generations. Every individual Mario agent in this population has its own set of MLP weights. All these individuals will try to complete the same level, using the same level random seed. This random seed remains the same throughout the entire evolutionary run for every individual. The difficulty of the level starts at 0 and is incremented by 1 if any individual during a generation manages to reach the goal. This means that the level may change between generations. After an entire evolutionary run of 100 generations, the most fit individual from the last generation is chosen to be tested for generalization. This individual is then evaluated on 4000 unique levels where its accumulated reward is checked. The reward is a value based on how far Mario has traveled. The farther the distance, the higher the reward. The levels are of difficulty 0, 3, 5 and 10. For each difficulty, 1000 levels will be tested, each with a unique random seed. These 1000 random seeds are always the same, in this experiment it is simply the values 1 to 1000. This experiment was conducted six times, in accordance with [24], and each experiment uses a different random seed for the evolutionary run. The accumulated reward in the varying difficulties were lastly averaged over all results and presented in a table with comparing results from Togelius' experiment. The data from this experiment is meant to answer RQ2.

In accordance with Togelius' experiment, this entire experiment was conducted on all three types of MLPs, Small, Medium and Large.

4.2 The Primary Experiment

This experiment was designed to answer both RQ1 and RQ3 but with primary focus on RQ1. In order to answer RQ1, the experiment was conducted on both the sequential CPU solution and the parallel GPGPU solution. The CMA-NeuroES was tested in the Mario AI benchmark in a number of different setups. The setups were chosen in order to establish a correlation between population size, problem dimension and fitness as well as a correlation between population size, problem dimension and execution time for both the parallel and sequential solution. Each run of the experiment will provide data to answer both RQ1 and RQ3. However for RQ3, whether or not the experiment was performed sequentially or in parallel is redundant information and will not be considered.

A single experimental run consist of ten (10) evolutionary runs with some static and some variable parameters. The static parameters that stay the same between all experiments are: the maximum number of generations (−maxGen) and the difficulty of the Mario levels (−difficulty). These parameters are set to 300 and 5, respectively. The choice to evolve for 300 generations was made with the number of experimental runs in mind. In an attempt to get more data points and
thus eliminate some noise in the data, primarily concerning RQ1, the limit of 300
generations was set in order to allow the completion of at least 20 experimental
runs for each solution within the time frame of this thesis project. The value of
the difficulty parameter is primarily motivated by the results from the prestudy.

The variable parameters are population size (\texttt{--popSize}) and MLP type (\texttt{--mlpType}). The MLP type represent the problem dimension as the different
types of MLPs have different input layer sizes. Only the Medium and the Large
MLP types were used in an experiment due to complications with generalization
for the Small MLP. The population size variable is set to a value from 100 to
500, with 100 incremental steps. Thus the ten evolutionary runs that make up a
single experimental run are the following:

<table>
<thead>
<tr>
<th>Population size:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Medium MLP:</td>
</tr>
<tr>
<td>100 200 300 400 500</td>
</tr>
<tr>
<td>Large MLP:</td>
</tr>
<tr>
<td>100 200 300 400 500</td>
</tr>
</tbody>
</table>

Table 4.1: Table displaying the ten different experiment setups.

The choice of these variables are primarily based on \cite{6} where population sizes
beyond 1000 are tested. However since the Mario AI benchmark is not as quick
to execute as merely calculating a single function (as in Hansen’s case), the very
large population sizes are unfortunately not testable within the time frame of
this thesis project. Though since the dimension of the Medium and Large MLPs
are far greater than the dimensions tested by Hansen, the very large, 1000+,
population sizes could be of interest in future work.

In addition to these parameters, the level random seed for every individual in
the population is changed with each new generation. The individuals will thus
try to complete different levels. This means that the population is no longer op-
timizing on a single level but is instead optimizing on the RL problem of learning
to complete Mario levels. However, the random seeds are capped between 1 and
100 in order to, more easily, allow for some potential pattern recognition. The
grand motive behind changing the random seed value between evaluations is to
try and allow for better generalization.

Twenty (20) of these experiments were conducted in both for sequential and
the parallel solutions. This resulted in 400 (10 * 20 * 2) data points, 200 for each
solution, to help answer RQ1.

As for RQ3, each of the 40 evolved populations from the experiments was tested in
a similar manner as was done in the prestudy experiment. Each population’s mean
genotype (MLP weight vector) was evaluated on the 100 levels the population had
been optimized on, which are all levels from random seed 1 to 100, on difficulty 5.
The mean accumulated reward for each of the ten distinct evolutionary runs (see
Table 4.1 can then be established and analyzed. The reward was determined by how far a distance Mario has traveled on the level when the trial was over.
Chapter 5

Results

Table 5.1 shows the results from the prestudy experiment. This data is primarily used to answer [RQ2], but was also used to motivate some parameter decisions for the primary experiment, such as difficulty for the benchmarks. In this table, Togelius’ results are displayed as well, for comparative purposes. The three bottom lines are the actual results of the experiment conducted for the prestudy.

The data in the columns "Level" to "10" are average values from six separate experimental runs with the exact same arguments except for level random seed. The Level column represent the maximum level the population reached in the experiment. The last four columns represent the average rewards the agents/controllers received on the 4000 test levels conducted after the evolutionary run.

<table>
<thead>
<tr>
<th>Controller</th>
<th>Level</th>
<th>0</th>
<th>3</th>
<th>5</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small MLP</td>
<td>3</td>
<td>1784</td>
<td>719</td>
<td>606</td>
<td>531</td>
</tr>
<tr>
<td>Medium MLP</td>
<td>0.83</td>
<td>864</td>
<td>456</td>
<td>410</td>
<td>377</td>
</tr>
<tr>
<td>Large MLP</td>
<td>0.5</td>
<td>559</td>
<td>347</td>
<td>345</td>
<td>300</td>
</tr>
<tr>
<td>Small SRN</td>
<td>2.83</td>
<td>3050</td>
<td>995</td>
<td>834</td>
<td>300</td>
</tr>
<tr>
<td>Medium SRN</td>
<td>1.83</td>
<td>1625</td>
<td>670</td>
<td>576</td>
<td>512</td>
</tr>
<tr>
<td>Large SRN</td>
<td>0.25</td>
<td>757</td>
<td>440</td>
<td>408</td>
<td>373</td>
</tr>
<tr>
<td>Small HyperGP SRN</td>
<td>1.87</td>
<td>2365</td>
<td>780</td>
<td>695</td>
<td>585</td>
</tr>
<tr>
<td>Medium HyperGP SRN</td>
<td>2.13</td>
<td>2352</td>
<td>786</td>
<td>679</td>
<td>545</td>
</tr>
<tr>
<td>Large HyperGP SRN</td>
<td>1.25</td>
<td>2314</td>
<td>633</td>
<td>588</td>
<td>534</td>
</tr>
<tr>
<td>CMA-NeuroES Small MLP</td>
<td>6.33</td>
<td>1297</td>
<td>1643</td>
<td>1408</td>
<td>1133</td>
</tr>
<tr>
<td>CMA-NeuroES Medium MLP</td>
<td>4.83</td>
<td>1029</td>
<td>1181</td>
<td>1131</td>
<td>846</td>
</tr>
<tr>
<td>CMA-NeuroES Large MLP</td>
<td>2.66</td>
<td>903</td>
<td>785</td>
<td>734</td>
<td>641</td>
</tr>
</tbody>
</table>

Table 5.1: Table containing results from the prestudy. The nine topmost lines are results from Togelius’ tests in [24], while the three bold lines at the bottom represent the results from our experiments.

In the same columns, a higher reward value represents getting further along the level, thus being better optimized to solve levels. One level is somewhere between 3500 and 4000 units long which thus represents the highest reward achievable.
Accumulating a value of that magnitude in this experiment would represent finishing every single one of the 1000 levels the agent was tested on.

5.1 The Primary Experiment

**Primary data.** For the primary experiment, the hardware on which they were conducted are of interest and importance. All experimental trials were run on the same household PC in a 64 bit GNU/Linux OS. The CPU is an x86_64 AMD Phenom II X6 1090T 3.2 Ghz with 6 cores, however only one core was utilized and no CPU multi-threading was utilized in any way. The graphics card, or GPGPU device, is a Radeon HD 5800. This GPGPU device is eligible with OpenCL 1.2 which is the version used in this study.

![Figure 5.1: Graph of the average total execution time of the experiments of all 20 different configurations. Values are presented in Table 5.2](image)

Figure 5.1: Graph of the average total execution time of the experiments of all 20 different configurations. Values are presented in Table 5.2.
### Table 5.2: Table of the average total execution time of the experiments of all 20 different configurations.

The values in this table are the data points in Figure 5.1 including standard deviations.

* Note that these values are averaged on fewer experiments than the supposed 20.

**Figure 5.1 & 5.2** visualizes the performance data resulting from the primary experiments.

![MLP execution time per frame](image)

**Figure 5.2:** Graph of the average MLP execution time per frame for all of the 20 different experiment configurations. Values are presented in Table 5.3.
Chapter 5. Results

The blue colored lines represent the results from the sequential experimental runs while the red colored lines represent the results from the parallel GPGPU runs. The solid lines represent the experimental runs using the large MLP type with the high problem dimension $N=1060$. The dotted lines represent the experimental runs using the medium MLP type with a smaller problem dimension, $N=580$.

<table>
<thead>
<tr>
<th>Config.</th>
<th>Milliseconds (ms) &amp; Standard Deviation (sd)</th>
<th>Population size ($\lambda$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ms</td>
<td>sd</td>
</tr>
<tr>
<td>CPU</td>
<td>1060</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.63</td>
<td>0.03</td>
</tr>
<tr>
<td>GPU</td>
<td>1060</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.63</td>
<td>0.04</td>
</tr>
<tr>
<td>CPU</td>
<td>580</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.28</td>
<td>0.01</td>
</tr>
<tr>
<td>GPU</td>
<td>580</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.54</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Table 5.3: Table of the average MLP execution time per frame for all of the 20 different experiment configurations. The values in this table are the data points in Figure 5.2 including standard deviations.

Due to premature termination, most of the experiments using a medium MLP type with a population size of 500 were removed from the dataset. The premature termination means that these experiments did not run for 300 iterations. This means that the data points for medium MLP (dotted lines) on $\lambda=500$ are averaged on a lot fewer experiments than the supposed 20. These premature terminations has a big effect on the results of the total execution time depicted in Figure 5.1. It is evident from the fact that blue dotted line breaks the trend by suddenly, at $500\ \lambda$, have a somewhat lower total execution time compared to the red dotted line.

However, the data for MLP execution time per frame is produced by dividing the total MLP execution time with the number of frames that the experiment executed. Since the number of frames are also lowered at an equal rate to the execution time these premature terminations does not affect these results in any major way.

Below each graph is also a table (see Table 5.2 & 5.3) with the values printed in text for a detailed clarification of the results.

Secondary data. The following results have been produced by testing each of the MLPs that has been evolved during the primary experiments. The data in Table 5.4 only concerns the secondary RQ3. Note that CPU and GPU experiments are no longer separate, thus every data point is averaged from 40 evolved MLPs. Reward in this table is presented in the same way that the prestudy table,
max reward would be somewhere around 3500 to 4000. The higher reward the
more optimized the MLP agent. Also note that these results are based on only
the one hundred first random seeds (1 - 100) and not the one thousand first, as
it was in the prestudy results table.

<table>
<thead>
<tr>
<th>Population size (λ)</th>
<th>Medium MLP</th>
<th>Large MLP</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1754</td>
<td>1552</td>
</tr>
<tr>
<td>200</td>
<td>1769</td>
<td>1780</td>
</tr>
<tr>
<td>300</td>
<td>1766</td>
<td>1756</td>
</tr>
<tr>
<td>400</td>
<td>1743</td>
<td>1737</td>
</tr>
<tr>
<td>500</td>
<td>1731*</td>
<td>1735</td>
</tr>
</tbody>
</table>

Table 5.4: Averaged accumulated reward for each of the ten different experiment
configurations.

* Note that this value is averaged on fewer experiments than the supposed 40.
Chapter 6

Analysis

6.1 Prestudy

In this section, the data produced in the prestudy objective will be analyzed in order to show successful accomplishment of the objective and what the results can help us answer. Naturally, this section will primarily concern [RQ2].

By being able to run proper experiments as well as produce and present valid data, it is evident that the objective has been successfully accomplished.

From Table 5.1 it is clear that the CMA-NeuroES performs very well, for all MLP types, in comparison to the solutions tested in [24]. As for the Level-column, every MLP type outclass the other solutions by reaching much higher levels in the 100 generations. This results in a somewhat lower, but still good, reward when tested in levels of difficulty 0 but a comparatively much better score on difficulties 3, 5 and 10. This is likely due to the CMA-NeuroES adapting and optimizing to complete harder levels (difficulty 3-5) for additionally many generations compared to the easier levels (difficulty 0-2).

However there is an apparent issue with generalization seeing as the controllers have finished several levels and increased the level difficulty which requires a reward of somewhere between 3500 and 4000. Togelius notes this as well:

—“However, there are problems with generalization. Controllers that have managed to progress to clear levels of difficulty 2 or 3 have problems with clearing levels of the same difficulty other than the particular level they were trained on, and often even fail to clear level 0.” [24]

Comparing the three CMA-NeuroES solutions to each other it is clear that the Small MLP type performs exceptionally well while the Large MLP has not been optimized as well. The Medium MLP performs better than the Large but worse than the Small. It is very likely that this outcome is a consequence of increased problem dimension. The same pattern is clear from Togelius’ experiments and is noted as follows:

— “The problems of spatial reach were not solved by simply adding more inputs, representing a larger part of the environment, to the standard neural network. Indeed, it seems that simply adding more inputs decreases the evolvability of the controllers, probably due the added epistasis of high-dimensional search spaces.” [24]
Arguably, however, the added inputs may serve evolvability better if the RL problem was more complex and the evolutionary run was allowed to evolve for several more generations.

Togelius also mentions that “The problems with generalization might be solved through using new seeds for every evaluation, though that will lead to problems with noise...”. Changing the seed between evaluations would make the RL problem more complex and could arguably, in turn, motivate using the Medium and Large MLP types despite their lower score in this prestudy experiment. The cost of this is likely more noise in the RL problem for the CMA-NeuroES to cope with. However, Heindrich-Meisner and Igel argues that “They usually outperform gradient-based approaches in the presence of noise and on multi-modal objective functions...” concerning ESs as direct policy search methods in RL problems[7]. Seeing as the CMA-ES is a state-of-art ES that has been thoroughly tested on high dimensional multi-modal functions [6], changing the seed between evaluations is easily motivated for the primary benchmark experiment.

6.2 GPU Utilization and Benchmarking

This section will analyze the CPU and GPU experiment results produced in the primary experiment. Most importantly, the performance evaluation data between sequential and parallel execution will be analyzed but also the generalized reward performance to answer RQ3. As with the prestudy, it is evident from the fact that data has been properly produced, that both objective two and three has been successfully accomplished.

The analysis will begin with the generalized reward performance since it is closely related to the results and analysis of the prestudy experiments.

Secondary RQ. In this paragraph the results in Table 5.4 will be analyzed. Those results are primarily important when answering the secondary RQ, RQ3. The purpose of RQ3 is mainly to attempt to motivate the use of high population sizes in high problem dimensions for solving complex RL problems with ESs.

As Table 5.4 suggests, there is no notable difference between the two MLP types with any of the five different population sizes. This indicates that the results are likely to have been produced by chance and what the numbers represent most likely some average reward that is very easily accumulated in a number of levels in the Mario AI benchmark by just running rightwards and jumping occasionally.

The only notable difference in the table is the Large MLP type with a population size of 100. This particular configuration performs exceptionally bad and accumulates almost 200 less reward than any other configuration. Looking at the results from the prestudy experiment (Table 5.1) gives some hint as to why this might be. In the prestudy the population size is 100 and in 100 generations
Chapter 6. Analysis

it does not manage to complete levels of difficulty level 2 on average. In this generalized experiment, the difficulty was set to 5. Of course there is also the major difference and much higher difficulty of solving the very complex task of solving levels in general compared to just solving a single level, as was done in the prestudy.

Ultimately it is clear that additional work is necessary in order to properly motivate using the CMA-NeuroES on such complex RL problems with MLPs in such high problem dimensions. Thus, [RQ3] will unfortunately remain unanswered.

**Primary RQ.** In this paragraph, the performance in terms of execution time is analyzed in order to answer [RQ1]. Despite the lack of motivation for using the CMA-NeuroES to evolve Mario AI controllers/agents in as high problem dimensions and with as large population sizes as this experiment has attempted, the performance results show some difference between CPU and GPU solutions. Thus, this study has still successfully reached the set aim by accomplishing the three objectives and by producing as well as presenting performance data indicating that a GPU solution may in fact remedy some of the computational intensity of solving complex RL problems. The fact that the Mario AI RL problem used in the experiments was not solved is unfortunate but still secondary to the performance results.

As for [RQ1], which is the primary research question we need to answer in order to claim success in reaching the aim, it is answered in two parts. The first part concerns the performance data on the entire evolutionary run. The data is presented in Figure 5.1 & Table 5.2. In addition to running the experiments, a T-test has been done on the data points. The T-test is two-sided and unpaired and the data is assumed to be normally distributed around its mean value. This assumption is based on the sole fact that they are measurements of execution time. The result of the T-test on the 'total execution time' data is presented in Table 6.1.

<table>
<thead>
<tr>
<th>Test relation</th>
<th>Population size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU 1060 rel. GPU 1060</td>
<td>✔</td>
</tr>
<tr>
<td>CPU 580 rel. GPU 580</td>
<td>✔</td>
</tr>
</tbody>
</table>

Table 6.1: T-test with $\alpha = 0.05$ on 'total execution time' data.

The data is presented in Figure 5.1 & Table 5.2

* Not available due to varying number of data points.
For these tests, the CPU and GPU results were tested against one another with the two different problem dimensions (MLP types) separately. The test indicates that the experiment data is reliable and did not occur simply by chance. However there are some points that are not statistically significant. These points represent ‘cross-over’ points in the data where the two solutions switch place horizontally and the one that was performing best in the previous population size is now performing worst. Thus, when crossing paths, the different solutions are very close in performance and so the T-test fails to recognize any statistically significant difference. Comparing Table 6.1 to Figure 5.2 demonstrates this effectively. In the corresponding figure, the comparison between the solid lines represents the first line in the T-test table while the dotted lines’ comparison correspond to the second line in the table.

Looking at the performance data it is clear that the higher the population size is and the larger the problem dimension is, the larger the relative difference in performance between the parallel and sequential solution is as well. This ultimately answers the primary RQ, RQ1. However, further analysis can be made. The very high standard deviations and thus error bars in this data set are likely to have occurred simply because of the random nature of an evolutionary run. There is no way to control how many frames each evaluation will execute and if all MLP agents (Marios) fail to complete the level before the maximum of 3000 frames has passed, the evaluation is complete. Thus, the number of executed frames, and therefore also the total time of execution, varies a great deal between evolutionary runs.

As the only section of the code that has been parallelized is the MLP execution time, it is of much interest to analyze this data as well. Figure 6.1 depicts how much time, in percent, is spent on executing MLPs in any experimental run. It strongly indicates that the percentage decreases for the GPU solution when the population size is increased, for both MLP types, but plate out at 400 λ with no further decrease. The CPU solution appears entirely opposite and it appears that the percentage MLP execution time is increasing slightly with increased population size. The difference is however not very big seeing as the Large GPU is just below 40% and the Large CPU is around 46% – 47%.
Furthermoe, looking at the MLP execution time per frame, a more distinct relative difference is evident. This data is presented in Figure 5.2 with corresponding table. The graph very clearly demonstrate a difference between the parallel and the sequential solutions where the parallel executes in significantly less time than the sequential. This is especially true for the higher population and problem dimension sizes.

As with the ‘total execution time’ data, T-tests have been made in the same fashion for the MLP execution time data set. The results are displayed in Table 6.2 and shows that a statistically significance difference exists between most relative data sets. Like previously, the comparisons that are not statistically significant represent cross-over points in the graph.
Chapter 6. Analysis

<table>
<thead>
<tr>
<th>Test relation</th>
<th>Population size</th>
<th>Statistically significant?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100</td>
<td>200</td>
</tr>
<tr>
<td>CPU 1060 rel. GPU 1060</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CPU 580 rel. GPU 580</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 6.2: T-test with $\alpha = 0.05$ on 'MLP execution time' data.
The data is presented in Figure 5.2 & Table 5.3.
* Not available due to varying number of data points.

The relation between the two GPU solutions are also interesting to analyze, where there is little difference in terms of execution time per frame for any of the population sizes. This indicates that the size of the problem dimension and population is secondary to some other factor that limits the decrease of MLP execution time. A qualified guess of what this factor may be is that it is related to time consuming context switches between the OpenCL kernels for the different passes. Additionally, and in conclusion, it is important to disclose that Linux is known to have flawed and substandard support for OpenCL and graphics card utilization in general. Furthermore, the GPGPU device (graphics card) that the experiments were run on does not have hardware support for working with double precision floating points which was used in the implementation. It does however have software support but as a consequence this lack of hardware support is likely to have had a significant impact on the execution time of the GPGPU kernels. However, as previously mentioned, the actual time spent inside the GPGPU might have been secondary to the time spent on context switches when changing kernels.
Chapter 7

Conclusions and Future Work

In conclusion, this study has reached the set aim by answering both the primary research question, $RQ_1$, as well as at least one secondary research question, $RQ_2$. Through the empirical statistical data produced by experimental performance evaluation, a clear difference in performance can be observed between the two solutions. This difference seem to benefit the GPU solution and increasingly so when population size ($\lambda$) and problem dimension is increased as well. The one, conventionally sequential, solution simply executes every single MLP one after another on the CPU while the other, parallel, solution attempts to batch-propagate MLPs in order to achieve better performance. The design and implementation of this parallel solution was also part of the problem statement and objectives and was ultimately a prerequisite to being able to perform the experimental performance evaluation to compare the given solutions. A lot of time was invested in that design and implementation which fortunately, judging from the results, was not in vain.

Unfortunately, however, $RQ_3$ will have to remain unanswered seeing as no clear correlation could be observed between population size and problem dimension with regards to maximum fitness. As was discussed in earlier chapters, there are arguably a few possible reasons that this information escapes our experimental results. Possibly, the population sizes are too small and/or the number of iterations are insufficient for the CMA-NeuroES to start recognizing patterns in the input layer. This might also be an issue related to how the state representation, as a grid surrounding the agent, is designed in the Mario AI benchmark. Furthermore it is also likely that the very random and perhaps all too noisy state space is responsible. Meaning that perhaps the fact that every individual in the population is optimizing on a different random seed for every new generation simply creates an incredible and unmanageable amount of noise. There is also the possibility that solving such complex RL problems through the use of NeuroES is inconceivable.

However in order to know with absolute certainty what the issues are, additional work needs to be done. This future work would include running experiments on several more generations, trying larger population sizes. Also instead of supplying each individual with a different random seed for every new generation,
give all of them the same random seed but change this one random seed with every new generation. Basically do further investigation on an algorithmic level in order to find more optimal configurations for solving the Mario AI RL problem. This includes investigating why the Small (and Medium on high $\lambda$) MLP appears to behave divergently in the CMA-NeuroES in a complex and general RL problem, thus terminating prematurely.

Additional work could also be done with regards to the performance when utilizing GPGPU to propagate multiple MLPs at once. The most obvious future work would be to test the solution on additional hardware and in other operating systems such as Microsoft Windows. This would potentially rule out the issues experienced in this study regarding double precision floating points and bad OpenCL driver support. Furthermore, additional optimization and/or rewriting of the OpenCL kernels could likely be done in order to avoid the presumed overheads from multiple context switching.
References


References


Appendix A

Kernel code

OpenCL kernel code for propagating input to hidden layer. Do note that numInput, numHidden, and numWeights are all constants that are added in before compiling the code. When propagating the hidden to output layer, the numInput constant is simply set to the value of numHidden and the numHidden constant is set to the value of numOutput. That is the only difference between propagating the two layers.

```
__kernel void mulWeightPassA(
    __global double * weights,
    __global double * in,
    __global double * tmpWeights )
{
    uint globalID = get_global_id(0);
    uint threadID = globalID%numInput*numHidden;
    uint blockID = (globalID-threadID)/numInput*numHidden;

    uint weightIndex = blockID*numWeights + threadID;
    uint neuronIndex = threadID/numHidden + blockID*numInput;

    tmpWeights[weightIndex] = weights[weightIndex]*in[neuronIndex];
}

__kernel void activationPassB(
    __global double * tmpWeights,
    __global double * out )
{
    uint globalID = get_global_id(0);
    uint threadID = globalID%numHidden;
    uint blockID = (globalID-threadID)/numHidden;

    double weight = 0;
    uint idx = blockID*numWeights + threadID;
    for(int i = 0; i < numInput; i++)
        weight += tmpWeights[idx + i*numHidden];
    out[globalID] = tanh(weight);
}
```