

# UNIVERSITY OF FRIBOURG

## BACHELOR THESIS

---

# Thesis Title

---

*Author:*  
David Gauch

*Supervisor:*  
Prof. Dr. Philippe  
Cudré-Mauroux  
Giuseppe Cuccu

January 01, 1970

eXascale Infolab  
Department of Informatics



# Abstract

David Gauch

*Thesis Title*

Neural networks have become a powerful tool for machine learning, particularly with the large amount of data now available. Deep learning has proven to be an effective method for many applications in various domains, but neural networks have also been used in the field of reinforcement learning to solve non-continuous control tasks. This approach is counterintuitive, since neural networks are continuous. The hypothesis is that discontinuous models such as binary trees would have an advantage in addressing these tasks. To test this hypothesis, a new function has been introduced that allows the tree to grow dynamically when tasks are too complex to be solved by its current structure. This is an important step towards architecture search for binary trees, and has been tested on different benchmarks with promising results. The results indicate that a growing binary tree could be an efficient model for solving many control tasks. Overall, the study provides insights into the use of discontinuous models for reinforcement learning, and the potential benefits of using dynamic structures such as growing binary trees for efficient and effective learning.

**Keywords:** reinforcement learning, black-box optimization, binary trees, architecture search



# Contents

<b>Abstract</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Neural Networks . . . . .	1
1.2 Binary trees . . . . .	2
1.3 Architecture search . . . . .	3
1.4 Motivation and Hypothesis for Discontinuous Models in Reinforcement Learning . . . . .	4
1.5 Contribution . . . . .	5
<b>2 Reinforcement learning basics</b>	<b>7</b>
2.1 Reinforcement learning and continuous control . . . . .	7
2.1.1 Reinforcement learning paradigm . . . . .	7
2.1.2 Continuous control . . . . .	7
2.2 Classical reinforcement learning . . . . .	8
2.3 Direct Policy Search . . . . .	10
2.4 Neuroevolution . . . . .	10
2.5 Black-Box Optimization . . . . .	11
2.5.1 Random weight guessing . . . . .	11
2.5.2 Evolution strategies . . . . .	12
2.5.3 Covariance Matrix Adaption Evolution Strategy . . . . .	12
2.6 Benchmarks for reinforcement learning control problems . . . . .	13
2.6.1 OpenAI Gym . . . . .	13
<b>3 Method</b>	<b>17</b>
3.1 Model . . . . .	17
3.1.1 Node module . . . . .	17
3.1.2 Functions module . . . . .	17
3.1.3 BTTree module . . . . .	18
3.2 Environments . . . . .	21
3.2.1 Lunar Lander . . . . .	21
Action space . . . . .	21
Observation space . . . . .	22
Rewards . . . . .	23
3.2.2 Bipedal Walker . . . . .	24
Action space . . . . .	24
Observation space . . . . .	24
Rewards . . . . .	25
3.3 Fitness module . . . . .	26
3.4 Env_model_interface module . . . . .	26
3.5 Challenges . . . . .	27

<b>4 Experiments</b>	<b>29</b>
4.1 Scientific experiments . . . . .	29
4.2 Experimental Design and Implementation . . . . .	29
4.2.1 Visualization . . . . .	30
4.3 Results . . . . .	30
4.3.1 Lunar Lander . . . . .	30
4.3.2 Bipedal walker . . . . .	31
<b>5 Conclusion</b>	<b>35</b>
5.1 Conclusion . . . . .	35
5.2 Future Work . . . . .	35
<b>Bibliography</b>	<b>37</b>

# List of Figures

1.1	Neural network and its components . . . . .	2
1.2	Binary tree . . . . .	3
1.3	Non-continuous control task . . . . .	5
2.1	Main interaction of the agent and the environment in reinforcement learning . . . . .	8
2.2	Optimization of a 2D problem with CMA-ES . . . . .	14
2.3	Atari games of OpenAI Gym . . . . .	15
3.1	Components of a single-node tree . . . . .	18
3.2	Components of a tree with three nodes . . . . .	19
3.3	Addition of two nodes in a binary tree with the <code>add_node</code> function . . . . .	21
3.4	Different states of the Lunar Lander environment . . . . .	23
3.5	Bipedal walker performing on both environment versions . . . . .	25
4.1	Plots of the lunar lander experiment . . . . .	31
4.4	Different states of the Bipedal walker environment where the walker got stuck in local optima . . . . .	33



## Chapter 1

# Introduction

Machine learning, particularly deep learning using neural networks, has greatly benefitted from the availability of large amounts of data. Neural networks have demonstrated significant potential in solving a wide range of tasks. By examining the potential of binary trees, this thesis aims to provide insights into alternative models that could complement or even surpass the performance of traditional neural networks in certain reinforcement learning scenarios.

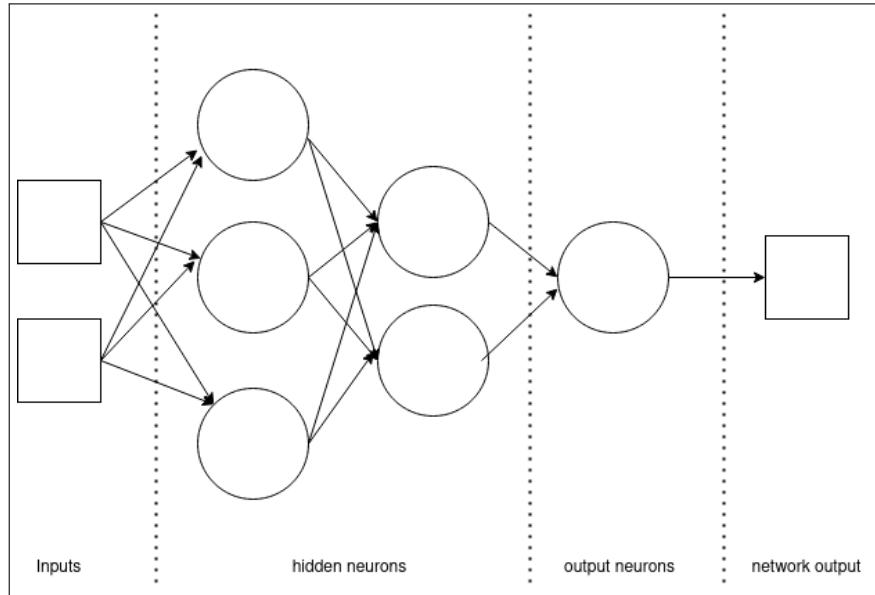
## 1.1 Neural Networks

Neural networks are a type of machine learning that takes inspiration from the structure and function of neurons in the human brain. The model is composed of layers of interconnected artificial neurons, which process and transmit information. In neural networks, the input data is first passed through the first layer, and then each subsequent layer receives the output from the previous layer as input.

Neural networks learn by adjusting the connections, or weights, between neurons based on the input data and the desired output. The output of a single neuron is calculated by combining the inputs from the previous layer with the corresponding weights, and adding a bias term if applicable. This weighted sum is then passed through an activation function, which transforms the sum into the output of the neuron. Common activation functions include the sigmoid function, hyperbolic tangent ( $\tanh$ ), and rectified linear unit (ReLU). The choice of activation function depends on the specific task and the architecture of the network.

A neural network can be represented as a computational graph that combines a series of simple functions to produce complex and high-dimensional representations, which capture the underlying function that enables the network to make accurate predictions. This ability to learn multiple levels of abstraction through function composition is one of the key strengths of neural networks and sets them apart from traditional linear models. Figure 1.1 illustrate a human-readable way to represent neural networks.

The most widely used optimization algorithm for the learning process is gradient descent, which helps find the optimal weights by minimizing the error between the network's predictions and the actual output. Backpropagation is a powerful tool for efficiently calculating the gradient of the loss function with respect to the weights. This is done through a forward pass, where the predicted outputs and intermediate node values are determined, followed by a backward pass, where the gradient of the loss function with respect to each weight is calculated using the chain rule of calculus. The gradient is then used to update the weights through gradient descent until the weights converge to values that minimize the loss function. For backpropagation to be effective, the activation functions of the artificial neurons must be continuous and easily differentiable (Goodfellow, Bengio, and Courville, 2016).



**FIGURE 1.1: Neural network** process incoming input data through a structure consisting of two hidden layers, one with three neurons and one with two, and an output layer with a single neuron. The output produced by the network is determined by passing it through an activation function, which represents the final output of the network.

The structure of a neural network, including the number of layers and number of neurons per layer, significantly impacts its ability to solve tasks. A larger number of layers allows for the approximation of more complex functions, but can also result in overfitting, where the model performs well on training data but poorly on new data. The field dedicated to finding optimal structures is referred to as neural architecture search.

## 1.2 Binary trees

Trees are a type of data structure that are commonly used in computer science and mathematics. They consist of nodes, which are connected by edges. Trees are special because they are a type of graph that is undirected, connected, and acyclic, meaning that nodes are connected by edges, but there are no loops or cycles in the graph.

In a tree, each node is either a parent or a child. The top node, with no parent, is called the root, while the bottom nodes with no children are called leaves. The distance from the root node determines the level of the node, with the root at level 0 and its children at level 1, and so on. Nodes on the same level are called siblings.

Binary trees are a specific type of tree in which every node, except for leaves, has at most two children, which are called the left and right nodes. Binary trees are easy to understand and visualize, making them useful for a variety of applications. An example of a binary tree is shown in Figure 1.2.

In addition to binary trees, there are other types of trees that are commonly used in computer science and data structures. One example is the "n-ary" tree, which allows nodes to have any number of children. Another example is the "balanced" tree, which is designed to keep the tree's height as small as possible, while still allowing for efficient searches and insertions. Balanced trees come in several different

varieties, such as red-black trees and B-trees, each with their own specific balancing algorithms and performance characteristics (Goodrich, n.d.).

For this project, a specific type of binary tree will be used, in which all nodes except the leaves have exactly two children, rather than at most two children like in classical binary trees.

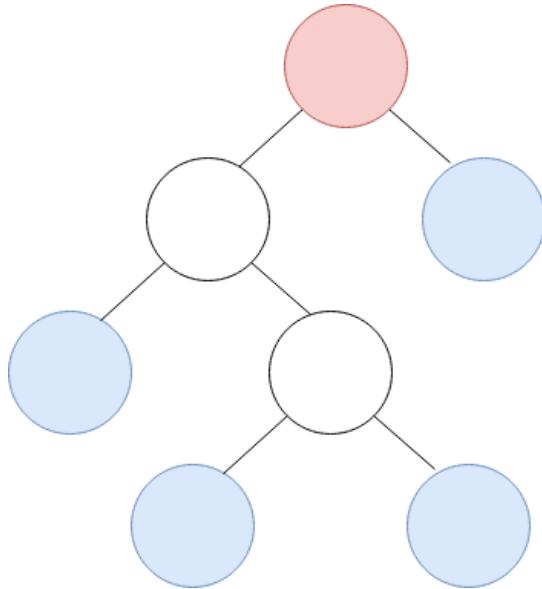


FIGURE 1.2: **Binary tree** where the blue nodes illustrate leafs and the red node is the root of the tree

Binary trees offer several advantages over other data structures. One of the main advantages is their ability to efficiently search, insert, and delete elements. Another advantage is their simplicity and interpretability, as the structure of a binary tree is easy to understand and visualize. This makes it easier to understand how decisions are made and identify errors in the model. In the context of reinforcement learning, binary trees can provide better approximation of discontinuities by allowing for different actions depending on the chosen path. This will be one of the motivations of the project discussed later.

### 1.3 Architecture search

Neural networks have seen significant advancements in recent years, due in large part to the availability of vast amounts of data. Deep neural networks have demonstrated their ability to solve a wide range of problems, such as image recognition and translation. However, finding the optimal number of layers and nodes, referred to as the architecture, for a neural network to efficiently solve a problem is still a challenging task. One approach is to try different architectures and evaluate their performance, which can be time and effort consuming when dealing with deep, complex neural networks as the structure needs to be designed by hand. Another approach is to use neural architecture search (NAS), which automates the process of finding an appropriate architecture for a specific task. Researchers continue to investigate effective methods for architecture search, with evolutionary algorithms, reinforcement learning, gradient-based optimization, or a combination of these techniques, being commonly used to explore the space of possible architectures (Elsken, Metzen, and Hutter, n.d.).

Until now, neural architecture search algorithms tend to be slow and expensive due to the need to train a large number of candidate networks to inform the search process. The paper by Mellor et al., n.d. showed that a possible solution to speed up this process would be to perform neural architecture search without any network training. The authors implemented a search algorithm called "NASWOT," which only makes observations on the initial untrained networks in the scope of convolutional networks.

The concept of automatically searching for structures that improve model performance without manual intervention can also be applied to binary tree models. A complex reinforcement learning task would usually require a larger binary tree, as this would increase the search space for policies and, thereby, improve the probability of finding good solutions. To achieve this, it is important to adapt the size of the tree according to the complexity of the task. By dynamically incrementing the size of the tree, various strategies can be implemented. The first decision is where to add or remove new nodes in the tree and how many to add. For example, nodes could be added from left to right until the level is full and then proceed to the next layer, or they could be added to a randomly selected leaf, as is the case in this project. Another decision is when to change the size of the tree dynamically. If the tree structure changes too rapidly, the algorithm may not have enough time to search for solutions in the actual search space, whereas waiting too long could result in the algorithm getting stuck in a local optimum and wasting time. Therefore, it is important to strike a balance and change the size of the tree at the right moment. Additionally, it is important to decide which functions the newly created nodes should have. Should they be the same as their parent nodes, or should they be different? Finally, another surprising consideration is whether the newly added nodes should be leaves, which is not the case in this project, as you will see later.

All of these decisions and more must be made and possibly compared when deciding which node-adding strategy to implement. It is important that, even if the tree structure changes, the mathematical equation of the tree remains invariant, which can be a challenging aspect to implement and can also limit the strategies that can be used.

## 1.4 Motivation and Hypothesis for Discontinuous Models in Reinforcement Learning

In the context of this project, the observation is that current models are typically continuous, but many control tasks are not. Neural networks for example need continuous functions in backpropagation. The basic idea behind backpropagation is to adjust the weights of the network by calculating the gradient of the loss function with respect to the weights. This is done by propagating the error backwards through the network, hence the name "backpropagation". The goal is to minimize the error of the network, the next time it makes a prediction and it is repeated for many iterations until the model converges to a set of weights that minimize the error. An example of a non-continuous problem this is swing up cartpole. This task consists of a pendulum fixed to a cart with a fixed joint above and a loose joint below, which is swung up in a first step and then stabilized in the second step. Figure 1.3 illustrates this problem. When attempting to approximate the function that models this task, it becomes apparent that the function will have a discontinuity. Due to the two distinct tasks involved, the agent must be able to recognize when the first

task is complete and the second one begins. In the real world, there are many control problems that are not continuous. The hypothesis is that discontinuous models would have an advantage in addressing these tasks. Binary trees should allow for the approximation of discontinuities. This is achieved by using a hyperplane to partition the observation space, and depending on the chosen subtree (by going to the left or right child), a different policy will be used. To test this hypothesis, multiple individuals can be evaluated in the environment (by running them through the fitness function) and their performance analyzed, along with other metrics such as the number of individuals that successfully solve the task. Hyperparameters also play an important role in improving the performance of individuals in the environment.

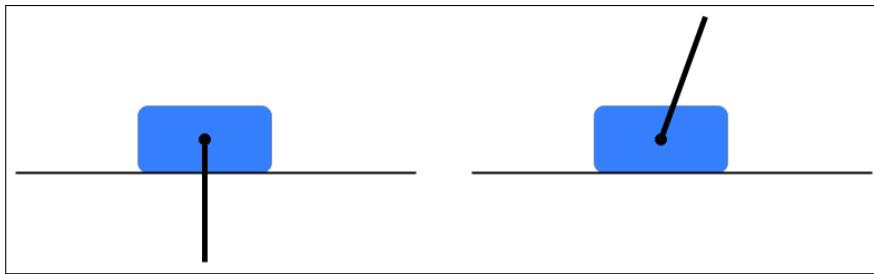


FIGURE 1.3: **Cart-pole swing up problem** On the left side, the initial state of the problem is depicted where the pole is pointing downwards and needs to be swung up by moving the blue cart on the horizontal line. On the right side, the cart has successfully managed to swing the pole up and now needs to stabilize it vertically.

## 1.5 Contribution

This thesis builds on the prior work "Alternative Models for Direct Policy Search in Reinforcement Learning Control Problems" (Masanti, n.d.), which recognizes the potential of using binary trees to address reinforcement learning challenges. The main contribution of this thesis is to enhance the efficiency of the previous model by introducing a novel function that dynamically increases the size of the binary tree. This new function represents a crucial step towards enabling architecture search for binary trees and has the potential to significantly improve their performance in solving reinforcement learning tasks. It is important to note, however, that only one possible tree-growing strategy will be tested, and no comparison of different structures will be made. The goal is solely to increase the tree size until the given task can be solved.

The thesis included a code structure for binary tree models, serving as a basic starting point. The first step was to make the code work. Once the binary tree model was functional, a basic architecture search technique was implemented, and its effectiveness was evaluated on several reinforcement learning problems.



## Chapter 2

# Reinforcement learning basics

## 2.1 Reinforcement learning and continuous control

### 2.1.1 Reinforcement learning paradigm

A learning paradigm is a formal description of a framework that enables the learning process by defining the sources of information to learn from, establishing criteria for assessing the effectiveness of a learning solution, and identifying the available resources that can enhance the learning process. Essentially, a learning paradigm provides a formal description of the underlying principles and assumptions that guide how we learn and improve our knowledge and skills.

The paradigm involves the interaction between two main components: an agent and an environment. The environment represents the "world" in which the agent operates and provides information about its current state. The agent, on the other hand, is responsible for taking actions within the environment. It acts on the environment by performing available actions and controls, perceives and assesses the changes occurring in the environment via the senses at its disposal, and engages in a feedback loop that informs and modifies its future behavior. Each interaction between the agent and the environment follows a sequence: the agent receives observations based on the current state of the environment, selects an action to take, and transmits that action to the environment. In response, the environment updates its internal state and provides feedback to the agent in the form of updated observations and a reward signal. The reward signal indicates the success or suitability of the agent's action in completing a task, while the updated observations provide information about the new state of the environment. The reward function, also called the fitness, is the only signal required for this learning method to improve and estimate how good a solution is. Figure 2.1 depicts a single timestep of interaction between the agent and the environment.

Reinforcement learning is different from other machine learning approaches such as supervised learning, which uses labeled data to predict outputs for unseen data, and unsupervised learning, which seeks to find patterns in unlabeled data. Reinforcement learning is both a problem that can be addressed with specific solution methods and a field of study that examines the problem and its potential solutions. Unlike supervised learning, there are no labeled data available in reinforcement learning, so the agent learns by maximizing its performance based on the reward signal (Sutton and Barto, 1998)

### 2.1.2 Continuous control

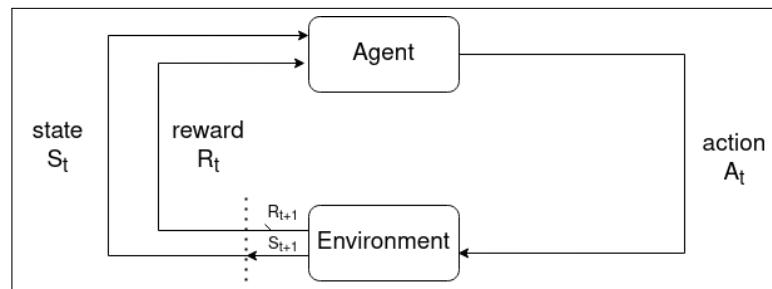
A control problem involves a dynamic system described by state variables, and the goal is to determine a strategy that leads the system to its desired target state. The

agent, which takes actions and is part of the environment, sends actions to determine the future behavior of the system. In continuous control problems, the system is observable at all times, and there is continuous interaction between the agent and the environment. An example is self-driving cars, which require continuous adaptation of behavior to drive optimally. To simplify the analysis of dynamic systems, time is often discretized into time-steps. This approach provides a way to break down the system's behavior into manageable intervals that can be analyzed and optimized. However, in practice, the accuracy of the analysis is often limited by the control frequency, which is the rate at which the system's inputs can be adjusted. If the control frequency is sufficiently high, the discretization of time becomes less critical, as the system's behavior can be accurately captured by the rate at which the inputs are adjusted. Therefore, the use of time-steps provides a useful tool for simplifying the analysis of dynamic systems, but the accuracy of the analysis ultimately depends on the system's control frequency (Franklin, Powell, and Emami-Naeini, 2014).

Both continuous control and reinforcement learning aim to design systems with richly structured perception, perform planning and control that adapt effectively to environmental changes, and exploit safeguards in the face of new scenarios (Recht, 2018).

## 2.2 Classical reinforcement learning

Some challenges cannot be solved through traditional problem-solving methods or supervised learning algorithms, and reinforcement learning is necessary to tackle them effectively. The Classical Reinforcement Learning framework is a comprehensive system designed with the sole purpose of enabling optimal interactions between agents and their environment. The framework is founded on the principle of trial-and-error learning, where the agent learns through experience by interacting with the environment, and receiving feedback in the form of rewards or penalties. The framework is structured to optimize the agent's behavior, allowing it to learn the best actions to take in any given situation. This broad framework is applicable across a range of contexts, and it has been successfully employed in various fields such as robotics, game-playing, and autonomous vehicles, among others.



**FIGURE 2.1: Main interaction of the agent and the environment in reinforcement learning.** At the beginning (timestep  $t$ ) the agent gets the observation  $S_t$  and the reward  $R_t$  from the environment. The agent performs then action  $A_t$  and sends it to the environment. The environment changes its state and returns a new observation  $S_{t+1}$  and a new reward  $R_{t+1}$ .

In reinforcement learning, the policy is a key element of the framework that determines the actions an agent should take in different states of the environment. The policy is represented by a mapping from states to actions, and it can be either

deterministic or stochastic. Deterministic policies specify a single action to take in each state, while stochastic policies specify probabilities for different actions to occur. The reward an agent receives depends on the chosen policy, and the sequence of states reached by the agent is called a Markov chain. In the Classical Reinforcement Learning framework, all reinforcement learning algorithms describe the problem as a Markov chain, which captures the essential properties of the agent's environment. This mathematical concept models systems that change over time in a way that depends only on the current state and not on the history of past states(Sutton and Barto, 2018). The algorithms then analyze the interactions between the agent and its environment, focusing on observations, actions, and rewards. The rewards are often modeled to simplify the policy, which is the strategy that the agent uses to decide on its actions based on the current state of the environment. The modeling of rewards can be done using various techniques, including value function or q-function, among others.

The value function is a key concept in reinforcement learning that allows us to evaluate the effectiveness of different policies. $V_\pi(s)$  defines the expected total reward that an agent can expect to receive by following the policy  $\pi$ , starting from state  $s$ . One way to compute the value function is using the Bellman equation (2.1),

$$V_\pi(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} P_{s,s'}^{\pi(s)} V_\pi(s') \quad (2.1)$$

which expresses the value of a state in terms of the values of its successors(Barron and Ishii, 1989).  $R(s, \pi(s))$  describes the reward obtained by doing the action chosen by  $\pi(s)$  in state  $s$ .  $\gamma$  enables to give more or less importance to the rewards that occur later.  $P_{s,s'}^{\pi(s)}$  describes the probability of reaching  $s'$ , after executing the action chosen by  $\pi(s)$  in the state  $s$  and  $V_\pi(s')$  stands for the future reward collected by the agent following policy  $\pi$  starting from the state  $s'$ . It is also important to note, the Bellman equation does not have a closed-form solution, which makes it challenging to compute the value function in practice. The value function can be used to define an optimal policy, which is the policy that is expected to maximize the reward over time. Another way to analyze policies is using the Q-function. $Q_\pi(s, a)$  is defined as the expected total reward acquired by the agent following policy  $\pi$  starting from state  $s$  and taking action  $a$ . The Q-function can be related to the value function through the equation  $V_\pi(s) = Q_\pi(s, \pi(s))$ . A common method for finding the optimal Q-function is Q-learning, which is an iterative process that updates the Q-function based on experience (Watkins and Dayan, 1992).

Reinforcement learning is well-suited for autonomous systems that learn to achieve a desired outcome through trial and error. However, this paradigm presents a unique challenge that is not encountered in supervised or unsupervised learning: balancing exploitation and exploration. Exploitation refers to the process of repeating actions that have resulted in positive rewards in the past, in order to maximize the cumulative reward. On the other hand, exploration involves trying new actions in order to potentially discover higher rewards and avoid getting stuck in a local optimum. Finding the right balance between these two approaches is crucial for the success of the learning process. There are a lot of strategies for this including  $\epsilon$  – greedy selection and  $Q$  – learning, but still a lot of research is done to find more effective solutions (Coggan, n.d.).

Reinforcement learning has been effective in solving a range of tasks, ranging from simple games to complex real-world problems in fields such as robotics, games

and autonomous driving. However, it has also encountered challenges in these real-world applications (Zhu et al., 2020). Nevertheless, the paradigm is sufficient for addressing the desired tasks in this thesis.

In conclusion, reinforcement learning offers a powerful tool for training agents to make decisions in dynamic environments and optimize for a given reward signal. It can effectively solve a range of problems while also presenting the unique challenge of balancing exploitation and exploration.

## 2.3 Direct Policy Search

Direct policy search is a method for solving reinforcement learning problems. It is a type of policy search algorithm that aims to directly optimize the policy, as opposed to value-based methods which use a value function approximation.

One of the primary benefits of direct policy search is its ability to handle high-dimensional and continuous action spaces, making it well-suited for problems with complex and continuous control tasks, such as robotics and autonomous systems.

To improve performance and better understand how network complexity contributes to representing the policy, the network in direct policy search can be separated into two components: one for learning intermediate representations of the input, and another for learning the policy (Cuccu, Togelius, and Cudre-Mauroux, 2019). This separation enables the use of smaller networks dedicated to policy learning.

Direct policy search can be implemented using various techniques, including gradient-based optimization methods and evolutionary algorithms. The choice of method depends on the specific problem and constraints and may involve trade-offs between computational efficiency and solution quality.

## 2.4 Neuroevolution

Neuroevolution is a technique that leverages black-box optimization, usually evolutionary algorithms, to determine the parameters of a neural network. The fitness function assesses the network by transforming the individual into weight matrices and inserting them into the network. Neuroevolution brings both the advantages and drawbacks of evolutionary algorithms to the challenge of finding a suitable neural network model.

In reinforcement learning, neuroevolution can be utilized for direct policy search, eliminating the need for supervised learning. The only necessary information is the reward function, not labeled actions. While neuroevolution for direct policy search in reinforcement learning has its advantages, there are also some limitations to consider. Firstly, these algorithms are computationally expensive and may not be as efficient as gradient-based algorithms. Additionally, since the performance of randomized algorithms depends on random events such as mutations, their performance can vary significantly across runs and there are no guarantees. Furthermore, neuroevolution algorithms only use the cumulative reward at the end of an episode and miss the correspondence between individual actions and per-step rewards.

For this project, the same concept is applied but using binary trees instead of neural networks. A proposal for the name of this method is "Treevolution".

## 2.5 Black-Box Optimization

In mathematics, optimization refers to the process of finding the maximum or minimum value of an objective function. Neural networks, for example, try to find the best weights for approximating an underlying function using techniques such as backpropagation and gradient descent. However, these techniques require knowledge of the derivative of the function, which may not always be available or may be too complex to compute (Schaul, n.d.). Black-box optimization is a method that does not rely on any assumptions about the function or its properties, and can be used to optimize any function approximator. It is based on a feedback score similar to reinforcement learning, and the parameter set is improved based on this score (Anderson, 1995).

Black-box optimization methods are generally less efficient than traditional techniques such as gradient descent because they do not take advantage of information about the structure of the function being optimized. This means they must explore a larger space of possible solutions, which can be time-consuming. However, black-box optimization methods can be effective in situations where the function being optimized is highly complex or has a large number of variables, and traditional methods may not be applicable. They are also flexible and can be applied to a wide range of problems without requiring any knowledge of the function being optimized.

In optimization problems, techniques typically converge to a single optimum, which is referred to as unimodality. On the other hand, multimodality refers to the presence of multiple distinct optima in the objective function, which is more common in real-world applications. Solving multimodal problems requires exploration in addition to the exploitation used in single-optimum problems. For example, gradient descent only uses exploitation and can only find another local optimum through exploration by restarting with a different initialization.

Black-box optimization methods, which do not depend heavily on knowledge of the function, can be well-suited for handling multimodal problems because they can explore a larger space of possible solutions. However, one challenge in multimodal landscapes is avoiding getting stuck in a local optimum before reaching the global optimum. A solution to this challenge is to generate multiple viable parametrizations, each exploring a different area in the optimization space. This technique gives a better understanding of the landscape and provides direction for where the most improvement can be obtained. An example of a method for generating parametrizations with improving scores is evolutionary algorithms.

### 2.5.1 Random weight guessing

The simplest version of an optimizer is to randomly select the set of weights, also known as random initialization, and keep always the best performing individuals. Algorithm 1 shows this procedure. This procedure is called random weight guessing. By randomly initializing the weights, the network is able to explore a wide range of possible solutions, increasing the chances of finding a good global minimum. Even with its simple implementation, random weight guessing has shown some great results in Classic Control benchmarks from the OpenAI Gym (Oller, Glasmachers, and Cuccu, 2020). It's important to note that the initialization of the weights can have a significant impact on the performance of the neural network. Thus, the chosen range for the randomly selected weights will have an impact on the result. Choosing a suitable range for the randomly selected weights is important to ensure that the network can learn useful features and avoid getting stuck in a

poor local minimum during the training process. However, it should be noted that random weight guessing will have some limitations with complex problems due to the large search space, because of the large number of possible weight combinations, which can make it difficult for the algorithm to find the global optimum.

---

**Algorithm 1** random weight guessing algorithm
 

---

```

1: best_ind  $\leftarrow$  None            $\triangleright$  No best performing individual at the beginning
2: best_fit  $\leftarrow -\infty$        $\triangleright$  fitness of best performing individual
3: while stopping criterion not reached do
4:   generate a population randomly
5:   evaluate fitness of each individual
6:   if fitness of individual  $>$  best_fit then
7:     best_ind  $\leftarrow$  individual
8:     best_fit  $\leftarrow$  fitness of individual
```

---

### 2.5.2 Evolution strategies

Evolution strategies (ES) is a class of evolutionary algorithms that is specialized for optimization of continuous variables. Inspired by natural evolution, ES is a black-box optimization algorithm that uses a process of mutation and selection to search for good solutions to a given problem. In the main loop of the algorithm, new individuals are created by mutating the parent individuals of the current generation. An individual in the context of ES refers to a specific set of parameters being optimized by the algorithm. A population is a group of individuals being considered by the algorithm at a given time, and a generation refers to one iteration of the main loop. The fitness of an individual is a measure of its performance or quality, based on the feedback score provided by the algorithm.

The main loop of the ES algorithm consists of creating new individuals from the parent individuals of the current generation, evaluating their fitness, and selecting the best-performing individuals to be the parent individuals for the next generation. This process continues until a sufficient solution is found, as determined by a stopping criterion. Algorithms differ in the number of offsprings created per generation, the number of selected individuals for the next generation, and how the mutation process is performed. Other than gradient-descent based methods, ES generates multiple individuals and by that explores different areas or paths of the optimization space independently, which can be beneficial for avoiding local optima and solving real-world problems that may require sophisticated exploration mechanisms. It is important to note that the efficiency of Evolution Strategies highly depends on factors like the population size, or the mutation and selection methods used. To maximize their performance, experimenting on this factors with different configuration setting might be useful(Salimans et al., 2017).

### 2.5.3 Covariance Matrix Adaption Evolution Strategy

CMA-ES (Covariance Matrix Adaptation Evolution Strategy) is a stochastic optimization algorithm that is used to optimize complex non-linear functions. It is a derivative-free optimization method that is particularly well-suited for high-dimensional problems. The algorithm works by maintaining a distribution of candidate solutions (i.e. a population of possible solutions) and adapts the distribution based on the performance of the solutions. It is based on the evolution strategy algorithm and uses

a covariance matrix to adapt the distribution. The algorithm iteratively updates the distribution until it converges to a solution that is close to the global optimum (Akimoto et al., 2012).

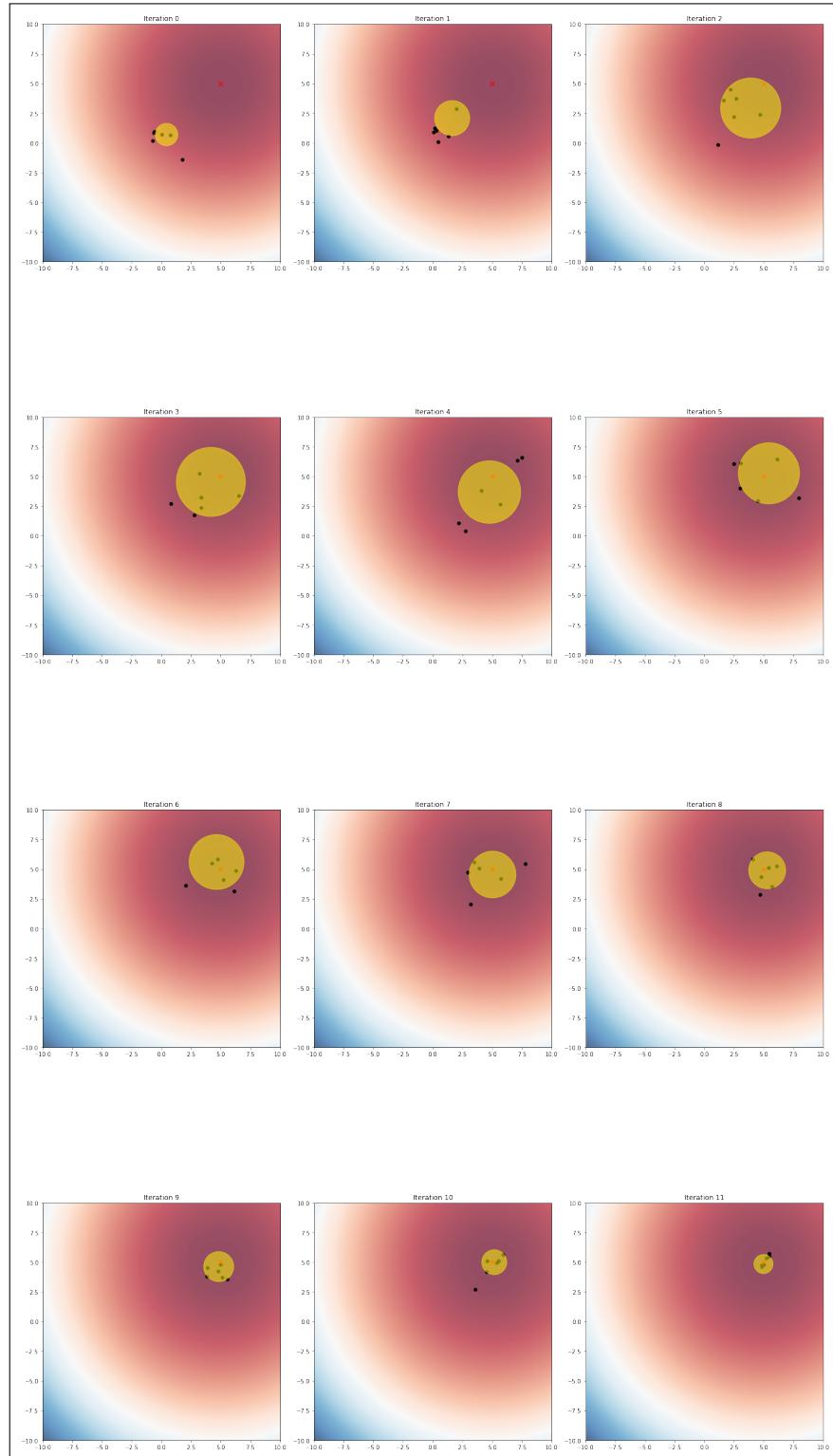
The algorithm has several hyperparameters that can be adjusted to optimize its performance and highly influence its efficiency. Some of the most important ones include: population size, step size (represented by "sigma" parameter), number of generations, number of parents (represented by "mu" parameter), and so on. The "mu" parameter represents the number of solutions (or parents) that are selected from the population to generate the next generation of solutions and it determines the balance between exploration and exploitation in the search process. The "sigma" parameter represents the step size of the search, it controls the scale of the search and determines how far the algorithm moves away from the current best solution in each generation, it also adjusts the standard deviation of the multivariate normal distribution that guides the search. CMA-ES is a robust optimization algorithm that is widely used. Figure 2.2 illustrates the evolution of the search distribution for CMA-ES on a simple quadratic function, which is a minimization task. The function is defined as  $(x - 5)^2 + (y - 5)^2$ . The background of the plot indicates good solutions with dark red colors and less good solutions with lighter colors. The red cross represents the optimal solution at coordinates (5, 5). The CMA-ES algorithm starts with a standard deviation of one and an initial population initialized with zeros. In the early stages of the optimization process, the search distribution will grow, leading to a high level of exploration. The search distribution is represented by a yellow ellipse, where the center of the ellipse is the mean of the current solutions and the width and height of the ellipse are twice the standard deviation of the search distribution. As the optimization progresses, the search distribution becomes more focused on finding the best individuals and converging towards the global optimum.

## 2.6 Benchmarks for reinforcement learning control problems

A benchmark environment in the context of reinforcement learning (RL) is a standard and well-defined scenario that serves as a reference point for evaluating and comparing the performance of different RL algorithms. These environments usually provide a clear definition of the state space, action space, reward structure, and other problem specifications. By using common benchmark environments, researchers and practitioners can easily compare the performance of their algorithms to those of others, and determine which algorithms are best suited for specific types of RL problems. A common benchmark environment is OpenAI Gym, which will be used in this project.

### 2.6.1 OpenAI Gym

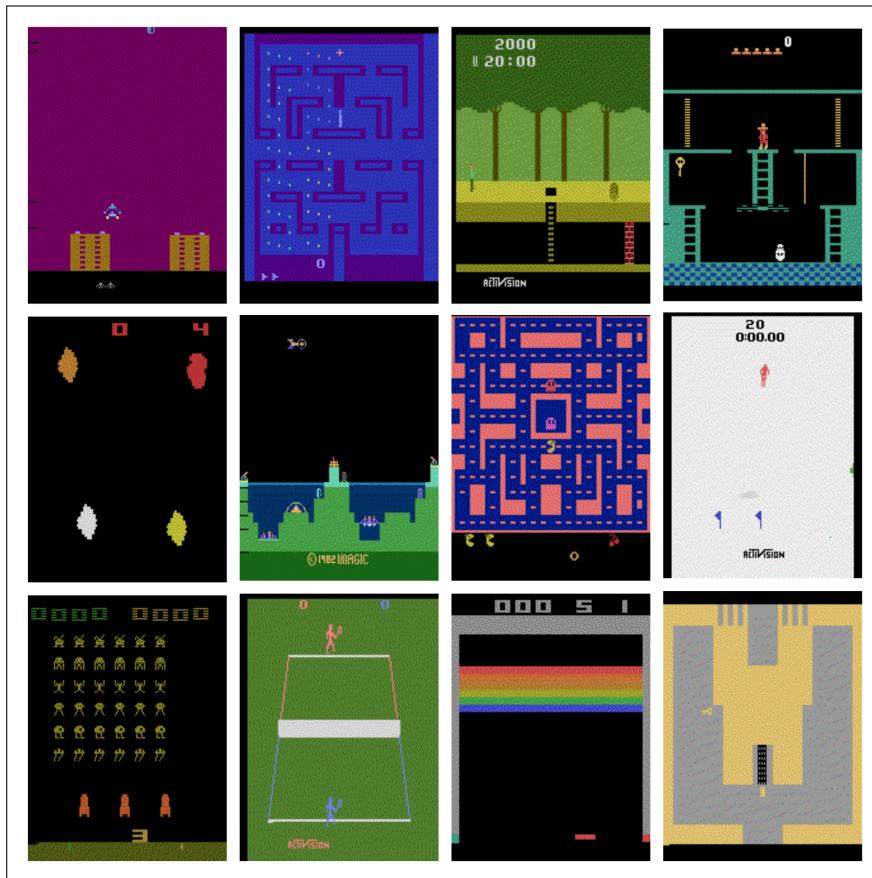
Open AI Gym is a toolkit that provides a variety of environments for developing and comparing reinforcement learning algorithms. One of its main advantages is that it uses the same interface for every task which enables an easy comparison and reproduction of results. It offers a range of environments for training agents, including classical control problems, Atari games, and physics simulations which vary in difficulty. OpenAI Gym offers tools for evaluating and visualizing the performance of the algorithms such as pre-built plotters and metrics. All of this gives big advantages for the research community in the field of reinforcement learning (*OpenAI Gym Beta 2016*). There are many different categories of environments available. The less



**FIGURE 2.2: Optimization of a 2D problem with CMA-ES** Illustration of a population reaching the global optimum in twelve generations. The background displays the fitness landscape, with red colors indicating higher scores. The red cross indicates the optimal score. The population is represented by black dots, and the yellow ellipse represents the search distribution.

complex environments are the classical control problems and can usually be solved rapidly. The environments used in the context of this project are of the category of Box2D environments. Those problems are harder to solve and are highly configurable. Another category are Atari games, which are a collection of classic video games from the 1980s that were released for the Atari 2600 console. These games are relatively simple by modern standards, but they are still challenging for machine learning algorithms because they require the agent to learn to make decisions in a complex and dynamic environment.

Some of the Atari games included in OpenAI Gym are Pong, Breakout, Space Invaders, and Pac-Man. These games have become popular benchmarks for reinforcement learning algorithms because they are simple enough to be used as a starting point for research, but complex enough to pose a challenge(Brockman et al., 2016). Figure 2.3 illustrates some of the Atari games available in OpenAI Gym.



**FIGURE 2.3: Some Atari games of OpenAI Gym** Illustration of a subset of the Atari environments available in OpenAI Gym. The represented environments from left to right and from the top to the bottom are: Air Raid, Alien, Pitfall, Montezuma Revenge, Asteroids, Atlantis, Ms Pacman, Skiing, Space Invaders, Breakout and Adventure

To start working with the toolkit, the first step is to generate an instance of a specified environment. This can be done with the predefined function `gym.make()` to which we pass the name of the environment we want to generate as parameter. The environment can then be stored as a variable and can be reset to its initial state with the `reset()` function which is typically done at the beginning of an episode and gives out the observations of the current state and some extra information. The observation is often used to get an action from a model which is then passed as

argument to the predefined function *step()*. This function returns the next state, the reward obtained, a boolean indicating whether the episode is over and some extra information too. These are just some basic functions that enable to start developing and evaluating reinforcement learning algorithms with the help of OpenAI Gym.

## Chapter 3

# Method

This chapter presents the core contribution of the project, focusing on the binary tree model with a particular emphasis on two key functions. The efficiency of the model was tested on two OpenAI Gym environments, which will be discussed. Additionally, some basic reinforcement learning components and their adaptations for this project will be presented. The code for this chapter is written in Python and can be found on Github<sup>1</sup>.

### 3.1 Model

The use of binary trees in this project offers a unique perspective in model architecture and decision-making, as well as potentially improved performance and interpretability compared to traditional neural network models. Additionally, the binary tree model does not require back-propagation or continuous functions, making it a highly efficient and effective solution. The black-box optimization techniques used further enhance the efficiency and effectiveness of the model.

#### 3.1.1 Node module

A node in a binary tree is composed of a pointer that points to its parent node, a function from a defined function class, and an assigned weight that adjusts the importance of the decision or computation made at that node. Additionally, the node has two pointers, one to its left child and one to its right child, that are used to traverse the tree and make decisions based on the input data and the functions applied at each node. Figure 3.1 illustrates a binary tree with a single node.

#### 3.1.2 Functions module

Each node in the binary tree contains a function that is used to make decisions or perform computations based on the input data. In the project, three function types were implemented: constant, linear, and perceptron.

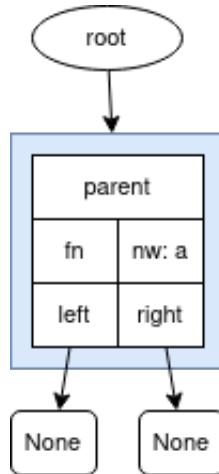
The constant function returns the weights as output regardless of the input values, while the linear function returns the dot product of the weights and the observations it received as input. The perceptron uses the sigmoid activation function

$$\frac{1}{(1 + \exp(-x))} \quad (3.1)$$

to transform the dot product between the weights and observations (denoted as  $x$ ) into a scalar value that can be used to perform computations.

---

<sup>1</sup>[https://github.com/DavidGauc/btree\\_model/](https://github.com/DavidGauc/btree_model/)



**FIGURE 3.1: Components of a single-node tree** Representation of a binary tree with a single node. The *root* pointer points to the node, and the *parent* pointer points to nothing as it is a single-node tree. The main components of the node are a function *fn* that is implemented in the function module, an amount of weights *nw* it contains, and two pointers to its left and right child (*left*, *right*) which point to None in this case.

Each instance of the function class contains the number of inputs and outputs, the weights used by the function, and the number of times the function has been activated. The weights can be learned or fixed, and the input and output values can take a specific range.

For nodes that are not leaf nodes in the tree, it is convenient to use linear functions, as their output will be a scalar value useful for the traversal of the tree. The details of how the functions, weights, and pointers are used to make decisions and perform computations in the tree can be found in 3.1.3.

### 3.1.3 BTree module

This module implements the binary tree model. A binary tree is composed of interconnected nodes, each of which holds a function. The illustration in Figure 3.2 shows a binary tree with a root node and two child nodes. A binary tree is a structure made up of linked nodes that contain functions that are used to make decisions or perform computations based on input data. Each node in the tree has a function, which can be one of the types implemented in the project: constant function, linear function, or perceptron. It also has two child nodes that can be either leaf nodes or internal nodes.

The process of using the tree to make decisions or perform computations is referred to as activation. The *activate* function represented as pseudo code in the Algorithm 2 starts at the root of the tree and navigates through the links between nodes based on the output of the current node's function. When it reaches a leaf node, it returns the output of that node's function as the final output of the tree.

The construction of the tree involves determining the decision points in the tree, which are selected based on the input data and the problem to be solved. The tree can be trained and updated by adjusting its functions, weights, and links between nodes.

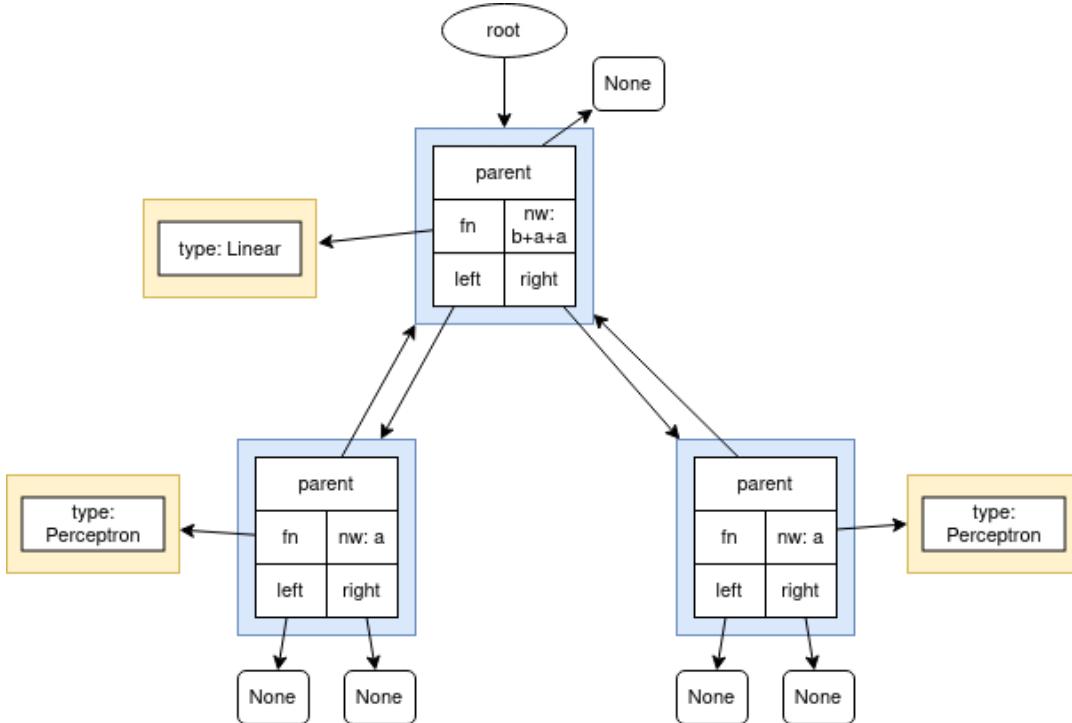


FIGURE 3.2: **Components of a tree with three nodes** Representation of a tree with a root node that contains a linear function and two child nodes with perceptrons. Each node displays its pointer to the corresponding function instance (represented by yellow blocks) and the connections to each other. The number of weights  $nw$  of the parent node is incremented by the number of weights of its children nodes.

Compared to other models like neural networks and decision trees, binary trees have both advantages and limitations. One advantage is interpretability, as the decision points and functions used in the tree can be explained. However, the efficiency of the tree is sensitive to its size, which can be a limitation. This can be mitigated through pruning techniques or other methods that optimize the tree structure.

---

**Algorithm 2** activate function
 

---

```

1: function ACTIVATE(obs)
2:   node  $\leftarrow$  self.root                                 $\triangleright$  starting from the root
3:   while True do
4:     if node is a leaf then return node.fn(obs)       $\triangleright$  output of node's function
5:     else if node.fn(obs)  $\geq 0$  then
6:       node  $\leftarrow$  node.left                           $\triangleright$  go to the left child node
7:     else
8:       node  $\leftarrow$  node.right                          $\triangleright$  go to the right child node
  
```

---

The difficulty of tasks can vary, making it necessary to have the ability to adjust the size of the tree accordingly. For instance, simpler problems like the Cartpole in OpenAI Gym can be solved with a smaller tree, while more complex problems demand a larger tree. Increasing the size of the tree enables more complex decision-making and can enhance the model's performance. However, it also increases the risk of overfitting.

In this project, a function has been implemented to dynamically increase the size

of the tree as the complexity of the problem increases. This process of finding an optimal structure is referred to as architecture search. The implemented function demonstrates only one of many possible techniques for growing the tree structure. This technique is a simple approach and will add only a minimal amount of complexity while maintaining the model's invariance. The function operates by randomly selecting a leaf node and adding two new nodes to it and procedes as follow:

- It begins by traversing the tree randomly until it reaches a leaf node. The pseudo code shown in Algorithm 3 illustrates the implementation of the *pick\_random\_leaf* function which does this job
- It verifies if the selected leaf is the root of the tree or not
- If it is, a new parent node is created
- If not, the leaf's parent node is duplicated to create a new parent node
- The new parent node is then set as the parent of the current leaf node and its duplicate.

The new nodes are added to the tree in such a way that the current leaf node's relative position to its previous parent node remains unchanged. For more details about the implementation the pseudo code of the function is shown in Algorithm 4.

---

**Algorithm 3** pick\_random\_leaf function

---

```

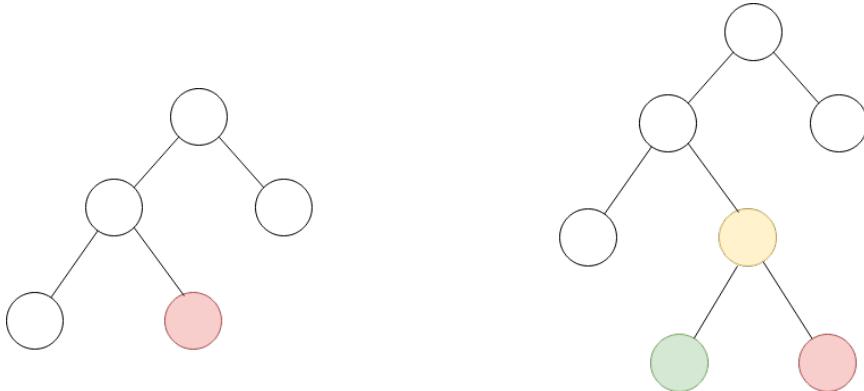
function PICK_RANDOM_LEAF
    current  $\leftarrow$  self.root                                 $\triangleright$  starting from the root
    last_direction  $\leftarrow$  None
    while current is not the leaf do
        if random  $\geq$  0.5 then                       $\triangleright$  random is between 0 and 1
            current  $\leftarrow$  current.left                   $\triangleright$  go to the left child node
            last_direction  $\leftarrow$  'left'                 $\triangleright$  store relative position of current node to its
        parent
        else
            current  $\leftarrow$  current.right               $\triangleright$  go to the right child node
            last_direction  $\leftarrow$  'right'
    return (current, last_direction)

```

---

It is important to note that the function does not add two child nodes directly to the leaf node, but instead adds one node as the parent of the leaf node and the other as its sibling. As shown in Figure 3.3, this is an example of adding two nodes to a binary tree. Figure 3.3a shows a binary tree with five nodes before using the node adding strategy. Figure 3.3b then shows the binary tree incremented by two new node with their position in the tree. After the addition of these new nodes, all the links in the tree must be updated, and information regarding the number of weights and the number of descendants must be refreshed. To achieve this, the function employs a process of propagating the information from the current leaf node to the root of the tree.

The function for expanding the size of the tree adapts to the complexity of the problem by growing based on a stagnation threshold, which is determined by a lack of improvement in the score for a certain number of steps. A larger tree has a greater search space, which enables it to handle more complex problems. However, it also increases computational time, making it crucial to strike a balance between expanding the tree too rapidly or too slowly.



(A) Illustration of the initial binary tree before the addition of the new nodes. The red node represents the randomly selected leaf node, from which the node addition process will start.

(B) Representation of the tree after adding two nodes using the `add_node` function. The yellow node is the newly created parent node and the green node is the sibling of the previously existing red node. It is important to note that the red node retains its relative position to the parent node

FIGURE 3.3: Addition of two nodes in a binary tree with the `add_node` function

## 3.2 Environments

For this project, two environments from the Box2D category of OpenAI Gym were utilized<sup>2</sup>. These environments are more complex than the "Classical Control" problems and offer greater configurability. Box2D is a 2D physics engine designed for games that enables objects to move in a realistic manner, enhancing game interactivity(*Box2D: Overview* n.d.).

### 3.2.1 Lunar Lander

The Lunar Lander environment simulates a scenario where a rocket must land between two flags on the surface of the Moon. The rocket has three engines that can either be fired at full speed or turned off. This environment is available in both a continuous and a discrete version. In this project, the discrete version was utilized. For the continuous version to work the output needs to be normalized in order to give out values in the range of -1 and 1. The figure shown in Figure 3.4 depicts the various states that the rocket can be in during the landing process.

#### Action space

The environment has four available actions: do nothing, fire the left engine, fire the right engine, or fire the main engine pointing downwards. The strength at which the engines fire cannot be adjusted and is fixed, resulting in a discrete action space with a dimension of 4. In practice the action with the biggest value obtained through the model is chosen and a number from 0 to 3 is sent to the environment.

<sup>2</sup><https://www.gymlibrary.dev/environments/box2d/>

---

**Algorithm 4** add\_node function

---

```

function ADD_NODE
    current, last_direction = pick_random_leaf()  $\triangleright$  go to a leaf (current) randomly
    if current is the root then
        create new_parent node with a linear function
        take over the number of activations from current to new_parent
        last_direction  $\leftarrow$  'left'  $\triangleright$  doesn't matter if its the root
        self.root  $\leftarrow$  new_parent  $\triangleright$  new_parent is the root
    else
        copy current's parent into new_parent node
        take over the number of activations from current's parent to new_parent
        create copy_node  $\triangleright$  copy of current
        if last_direction is 'right' then  $\triangleright$  maitain relative position to parent node
            new_parent.right  $\leftarrow$  current  $\triangleright$  set current as right child of new_parent
            new_parent.left  $\leftarrow$  copy_node
        else if last_direction is 'left' then
            new_parent.left  $\leftarrow$  current
            new_parent.right  $\leftarrow$  copy_node
        else
            raise error
        Set new_parent's parent to current's parent  $\triangleright$  fix parent links
        Set current's parent to new_parent
        Set copy_node's parent to new_parent
        if new_parent is not the root then
            if last_direction is 'right' then
                set new_parent as right child of its parent node
            else if last_direction is 'left' then
                set new_parent as left child of its parent node
            else
                raise error
        if new_parent is not the root then
            parent_iter  $\leftarrow$  parent of new_parent  $\triangleright$  starting point for progating up
            while True do
                number of wheights of parent_iter is incremented by wheights of newly created nodes
                number of nodes of parent_iter is incremented by nodes of newly created nodes
                if parent_iter is root then
                    break
                else
                    Set parent_iter to its parent  $\triangleright$  Go one node upwards
            Add to the number of weights of new_parent the number of weights of its two children nodes
            Add to the number of nodes of new_parent the number of nodes of its two children nodes

```

---

**Observation space**

The observation space for the Lunar Lander contains eight values. Two of them are booleans that indicate whether the corresponding leg of the lander is touching the

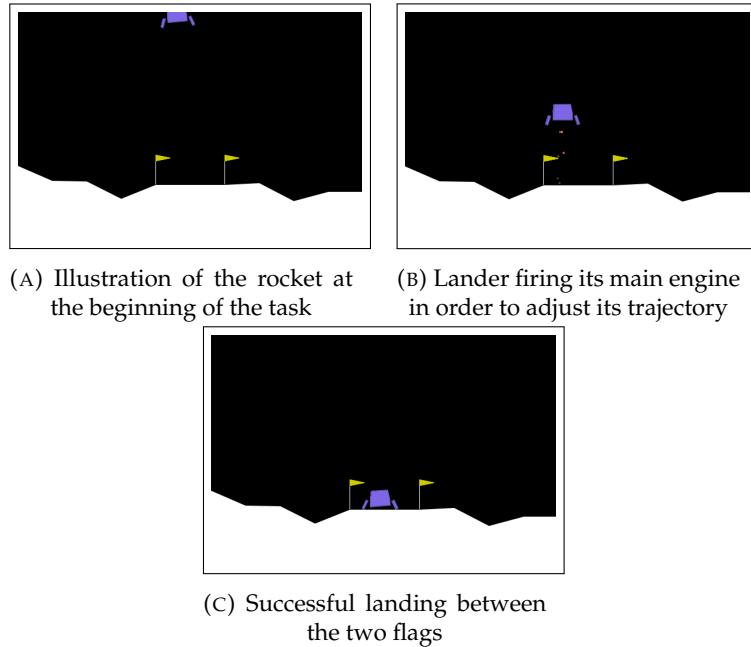


FIGURE 3.4: Different states of the Lunar Lander environment

Moon's surface or not, while all the other values are continuous.

TABLE 3.1: Observation values for the Lunar lander

	Min	Max
coordinates of the lander in x	-1.5	1.5
coordinates of the lander in y	-1.5	1.5
linear velocity in x	-5.0	5.0
linear velocity in y	-5.0	5.0
angle	-3.14	3.14
angular velocity	-5.0	5.0
left leg touching ground	0	1
right leg touching ground	0	1

## Rewards

For the agent in the Lunar Lander environment, it receives a reward for successfully landing on the landing pad starting from the top of the screen. The reward points in the default implementation are calculated as follow:

- $-100 \times \sqrt{state[0] \times state[0] + state[1] \times state[1]}$ : This calculates a penalty for the horizontal position and velocity of the lander, where  $state[0]$  and  $state[1]$  are the normalized horizontal position and velocity, respectively. The closer the lander is to the center of the viewport, the closer the value of  $state[0]$  will be to 0, and the less penalty it will incur. The penalty is scaled by -100 to make it a significant factor in the reward.
- $-100 \times \sqrt{state[2] \times state[2] + state[3] \times state[3]}$ : This calculates a penalty for the vertical position and velocity of the lander, where  $state[2]$  and  $state[3]$  are

the normalized vertical position and velocity, respectively. The calculation is similar to the one for the horizontal position.

- $-100 \times |state[4]|$ : This calculates a penalty for the angle of the lander, where  $state[4]$  is the angle of the lander with respect to the vertical axis. The more the lander is tilted, the higher the penalty.
- $10 \times state[6]$ : This adds a bonus for having the first leg of the lander in contact with the ground, where  $state[6]$  is equal to 1 if the first leg is in contact, and 0 otherwise. The bonus is scaled by 10 to make it a relatively small factor in the reward.
- $10 \times state[7]$ : This adds a bonus for having the second leg in contact with the ground. The calculation is similar to the one for the first leg.
- $-0.30 \times main\_engine$ : This calculates a penalty for each frame the main engine is firing. In the case of a discrete action space  $main\_engine$  is either 1 or 0.
- $-0.03 \times side\_engine$ : This calculates a penalty for each frame the a side engine is firing. In the case of a discrete action space  $side\_engine$  is either 1 or 0.

An additional reward of -100 or +100 points for crashing or landing safely respectively is obtained at the end of the episode. The final reward is the sum of all of these terms. The design of the reward encourages the agent to land the lander safely on the landing pad with minimum velocity, at a suitable angle, and with both legs in contact with the ground.

The task is considered solved when the agent accumulates a total reward of 200 points or higher. It's worth noting that the exact rewards given for each action, state or event are not fixed and can be adjusted to fine-tune the agent's behavior, based on the specific implementation of the environment.

### 3.2.2 Bipedal Walker

This environment simulates a two-legged robot attempting to walk as far as possible on uneven terrain. There are two versions available: a "normal" version (Figure 3.5a) and a more challenging "hardcore" version (Figure 3.5b) which includes obstacles. The robot is composed of a hull and two legs, each with two joints, one connecting to the hull and the other allowing the leg to bend. Figure 3.5 illustrates the robot in action on both versions.

#### Action space

The actions of the bipedal walker are continuous, with four actions available, corresponding to the motor speed values of each joint. The values range from -1 to 1 and determine the movement and stability of the robot. It's possible to adjust the range of values in different implementations of the environment.

#### Observation space

The observation space for the bipedal walker has a dimension of 24 and consists of continuous values, as well as a few boolean values that indicate whether the legs are in contact with the ground or not. The observation space includes information such as the angle, angular velocity, linear velocity, and position of the torso of the robot. It's worth noting that the exact position of the robot is not explicitly stated in the

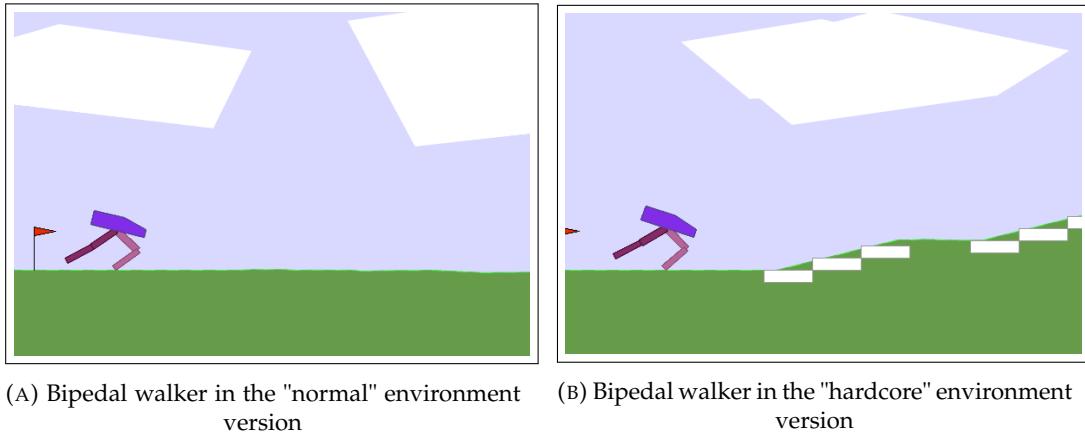


FIGURE 3.5: Bipedal walker performing on both environment versions

observation space, but it can be derived from other observations, such as the linear and angular velocities of the joints.

TABLE 3.2: Observation values for the Bipedal Walker

	Min	Max
hull angle speed	-3.14	3.14
angular velocity	-5.0	5.0
horizontal speed	-5.0	5.0
vertical speed	-5.0	5.0
position of joints	-3.14	3.14
joints angular speed	-5.0	5.0
left leg contact with ground	0	1
right leg contact with ground	0	1
10 lidar rangefinder measurements	-1.0	1.0

## Rewards

A reward is given to the robot when it is able to move forward without falling. Falling is defined as the hull touching the ground (horizontal position less than 0) and it is penalized by -100 points. If the bipedal walker reaches the end of the environment, it accumulates 300 points. The episode is also terminated if the horizontal position of the walker is greater than the length of the terrain. The default calculations for the reward are following:

- $130 \times pos[0]/SCALE$ : This encourages the agent to move forward.  $pos[0]$  is the normalized horizontal position of the walker and  $SCALE$  is a normalization factor that enables to receive 300 points on completion of the task.
- $-5.0 \times |state[0]|$ : This calculates a penalty for deviating from keeping the head straight.  $state[0]$  is the normalized angular velocity of the walker's head. The more the walker's head deviates from being straight, the higher the penalty.
- $-0.00035 \times MOTORS\_TORQUE \times np.clip(|a|), 0, 1)$ : This calculates a penalty for the use of motor torque by the agent the calculation is done for each motor

of the walker. *MOTORS\_TORQUE* is a constant that represents the maximum torque that a motor can apply. The larger the torque applied by a motor, the larger the penalty. The use of *np.clip* ensures that the torque used is clipped to the range [0, 1].

The "normal" version is considered solved when 300 points are earned within 1600 time steps. For the "hardcore" version, the same amount of points has to be earned within 2000 time steps. The goal is to achieve the highest possible reward while avoiding falling and moving as efficiently as possible. Like for the Lunar Landar, the values can be adjusted to fine-tune the agent's behaviour.

### 3.3 Fitness module

The fitness module acts as a bridge between the model and the environment, making it a crucial component in reinforcement learning algorithms. It provides the mechanism for feedback and evaluation of the agent's performance, allowing it to learn and optimize its behavior. The *fit* function, at the core of the fitness module, implements the control loop, evaluating the quality of the agent's actions based on the rewards provided by the environment.

The efficiency of the agent is calculated by summing the rewards received over the course of a single episode, which is given as the output of the *fit* function. This score is used to guide the learning process and update the agent's parameters, helping it to continually improve its performance. The pseudo code in Algorithm 5 illustrates the basic elements of the *fit* function.

---

#### Algorithm 5 fit function

---

```

function FIT(ind)
    reset the environment and get the observations
    set the weights of ind in the model
    score  $\leftarrow$  0
    done  $\leftarrow$  False                                 $\triangleright$  False is a boolean
    for number of step nsteps do
        action  $\leftarrow$  env_model_interface.get_action(model,obs)
        get new state of environment after executing action step
        increment score with the obtained reward from the action step
        if done then
            break
    return score                                 $\triangleright$  in our case  $-score$  as we use CMA-ES

```

---

### 3.4 Env\_model\_interface module

Depending on whether the environment has a discrete or continuous action space, the module needs to make some adjustments through its two functions, *is\_env\_discr\_or\_cont* and *get\_action*.

The function *is\_env\_discr\_or\_cont* determines whether the action space of the given environment is discrete or continuous. If it is discrete, the number of actions required for the output of the binary tree is set to the number of actions in the action space. If the action space is continuous, the number of actions is determined, and a lambda function is created to rescale the action within the action boundaries. This

function assumes that the output of the 'action' is between zero and one, as is the case for a logistic function, for example.

The function `get_action` is used by the `fit` function to retrieve the action from the model. It takes the model and the current observation of the environment as inputs. The function first obtains the action through activation of the binary tree. If the environment is determined to be continuous by `is_env_discr_or_cont`, the action is rescaled and returned as the output. This means that all outputs of the binary tree are used and they correspond to the signal to send to each control, which requires regression to approximate an action based on the given outputs. If the environment is discrete, the function selects the action with the highest activation by examining the leaf reached by the activation of the binary tree. The procedure is outlined in the pseudocode in Algorithm 5.

---

**Algorithm 6** `get_action` function

---

```
function GET_ACTION(model, obs)
    action  $\leftarrow$  model.activate(obs)            $\triangleright$  action obtained from model activation
    if environment type is continuous then
        rescale the action within the action boundaries
    else if environment type is discrete then
        pick action with highest activation
    else
        raise Error
    return action
```

---

## 3.5 Challenges

- The main difficulty was encountered in implementing the `add_node` function. The goal is to generate a model that is functionally equivalent to the previous best performing model but still capable of improvement. This is achieved by adjusting more weights. If improvement is not achieved, the `add_node` function can be called again until the critical complexity is reached and better scores are obtained. When adding new nodes, it's important to maintain the direction chosen during activation and the number of times each node's function was activated before the node addition. This means that if a node's function chose to go to the left child node during activation, the new larger tree should also go the same way, and the activation should lead to the same leaf but within a larger tree. The code memorizes the last direction chosen when randomly going through the tree, and when adding new nodes, the randomly selected leaf keeps its relative position to its new parent nodes.
- Another challenge was ensuring that each node always knew the number of weights it and its descendants had. This information is crucial for the functions within the nodes. When adding new nodes to a tree, the number of weights for all ancestor nodes of the newly added nodes is no longer correct, as they must now include the number of weights of those newly added nodes. The same holds true for the number of nodes, which also needs to be updated. The number of nodes for a given node corresponds to the number of descendants of that node, including itself. In the implementation of the `add_node` function, this information was propagated upward through the tree starting from the parent node of the one of the two newly inserted nodes (the one that is set as

parent of the leaf which was chosen randomly). The number of weights and nodes of the newly inserted nodes was added to the current node iteratively. Finally, the number of weights and nodes of the new parent node, which was not included in the loop, was also increased by the number of weights of its two children.

# Chapter 4

# Experiments

In this chapter, the objective is to address the research questions of this study through a series of experiments. The experiments performed in this study involve solving the "Lunar Lander" and "Bipedal Walker" control tasks from the OpenAI Gym library using a binary tree model. The results obtained from these experiments are analyzed and discussed in terms of the efficiency of the binary tree model in solving these control tasks.

## 4.1 Scientific experiments

Basic scientific experiment involves several steps that ultimately lead to a conclusion based on the observations made. The first step is to observe a phenomenon and formulate a hypothesis about how or why it works. Next, an experiment is designed and executed to validate or disprove the hypothesis. A crucial step is to then analyze and interpret the data obtained from the experiment, and finally, to draw a conclusion. It is important to note that in computer science, the process can be more complex because the experimentation often involves creating something that did not exist previously. However, the same scientific methods must still be applied to study and understand the newly created system.

## 4.2 Experimental Design and Implementation

The initial structure of the tree was set to a single-node tree, with either a perceptron or constant function. The `add_node` function was implemented to allow the tree to have linear functions in its decision-making nodes (all except the leaves) and either perceptrons or constants at its leaves when the tree has more than one node. The experiment was optimized using random weight guessing and CMA-ES. Although random weight guessing showed promising results, especially in the early stages of solving the environments, the analysis will focus on CMA-ES as it showed more potential for these problems. A target score was set for each environment, and the experiment would stop once this target score was reached (the target score has to be negative for CMA-ES). The `add_node` function was called when a stagnation threshold was reached, which grew proportionally to the tree size to allow for rapid exploration of the optimal solution in smaller trees, and slower exploration as the tree grew and the search space became larger. As CMA-ES works with a covariance matrix, its size was adjusted when nodes were added to the tree, with a new matrix created and the best-performing individual reset. This allowed us to see how the individual with the best fitness evolved with changes to the tree structure. The control loop searched for individuals that performed well with the current tree size,

and if the threshold was reached, it would grow the tree and continue until the target score, indicating that the environment was solved, was reached. The values for specific tasks can be adjusted in configuration files.

#### 4.2.1 Visualization

To evaluate the performance of the binary tree in solving the environments, two plots were used. First, a score over generations line plot was used to indicate the current best performing individual. The scores were negated to have positive scores on the plot, as CMA-ES requires negative results. This plot shows if the tree with increased size tends to have more individuals reaching high scores.

The second plot is a log-scale histogram of the mean scores. For this, the mean score of each population is calculated. This plot shows if the model overall tends to have more individuals achieving high scores or not.

### 4.3 Results

The experiments for the lunar lander were all run on an Acer Spin SP513-52N with an Intel Core i7-8550U CPU and 7.7 GB RAM, as they were solved rapidly. No other programs were run at the same time during the execution of the experiments. The bipedal walker experiments were run on a remote server of the University of Fribourg as the time to run them was longer. The server has 256 processors of the brand AMD with each having 64 CPU cores and 251 GB RAM. Both plots described in 4.2.1 were used to analyze the performance of the models solving the two environments.

#### 4.3.1 Lunar Lander

The Lunar Lander was solved rapidly using the binary tree implementation, with a target score of 270 (-270 with CMA-ES) chosen to indicate a smooth landing between the two flags in the environment. The initial tree structure, a single node tree with a perceptron function, was found to be too simple to solve the task, although it could solve the task in some cases. The use of one linear function as the root and two perceptrons as child nodes was found to be effective in reaching the target score when using CMA-ES as the optimizer. However, this was not the case with constant functions for the leaves, where a larger structure was required to achieve the same result. The number of steps, which indicates the maximum number of steps the agent can execute per episode, was set to 300.

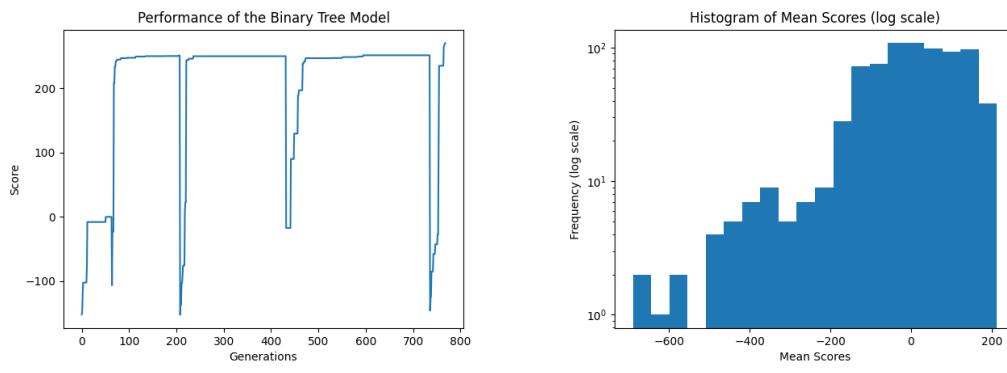
For the CMA-ES optimizer, the initial standard deviation (also called *sigma*) was set to 0.3, and the starting point (also called *mu*) is an array with a size equal to the number of weights in the binary tree, with their values chosen randomly between zero and one. The optimum is suggested to lie within  $\mu \pm 3 * \sigma$ , according to the documentation<sup>1</sup>. Furthermore, an option was set to determine the maximum number of iterations done by the optimizer. This value allows the tree to grow whenever the maximum number of iterations is reached. As explained before, the number of iterations should be small for a small tree structure with few weights and longer for a larger tree. This was achieved by setting the maximum number of iterations to be the multiplication of the number of weights of the current tree and a scalar. The scalar for this experiment was set to two and is called the *stag\_step*.

---

<sup>1</sup>[https://cma-es.github.io/apidocs-pycma/cma.evolution\\_strategy.CMAEvolutionStrategy.html](https://cma-es.github.io/apidocs-pycma/cma.evolution_strategy.CMAEvolutionStrategy.html)

Figure 4.1a shows the evolution of the best-performing individual over the generations in a line plot. The large score reductions in the plot indicate the use of the `add_node` function, which increases the size of the binary tree. With the increasing tree size, both the best-performing individual and the covariance matrix of CMA-ES are reset. This means that the best score obtained by an individual is reset to minus infinity, and once a population passes through the experiment, the score of its best-performing individual is overtaken as the best score. This method enables us to see if the new structure of the tree reaches high scores quicker than the preceding one. The plot shows that the initial tree structure, a single-node tree with a perceptron function, achieves low scores. However, by adding two nodes to the tree, which means having a linear function as the root and two child nodes with perceptrons, scores over 200 points are reached. Also, the steepness of the slope that shows the rapidity at which those scores were reached for tree sizes with more than one node is similar. This shows that further increases (after having a tree with three nodes) in the size of the binary tree do not significantly increase the scores and the execution time to reach high scores. The figure also shows that the single-node tree was searched for optimal solutions over a few generations, and that the number of generations searching for solutions increases with the size of the tree. This can be seen by the distance between the depressions of the graph. The bars in the graph that result from these separations represent the evolution of the best scores obtained with a certain tree structure.

Figure 4.1b shows a log-scaled histogram of the mean scores achieved. The y-axis shows the mean scores, and the x-axis shows the number of populations that achieved these scores in log-scale. The mean is calculated over 15 individuals representing one population. The plot shows that relatively few populations achieve low scores in this environment. Most populations have a score of about 0 points, although populations with a high mean remain high. The graph shows the mean of the populations over all tree sizes.



(A) **Score over generations lineplot of the lunar lander environment** Evolutions of the best performing individual in the lunar lander environment with an increasing binary tree size over the generations

(B) **log-scale histogram of the mean scores obtained in the lunar lander environment** The scores where obtained with a growing binary tree until one individual obtained a score of at least 270 points

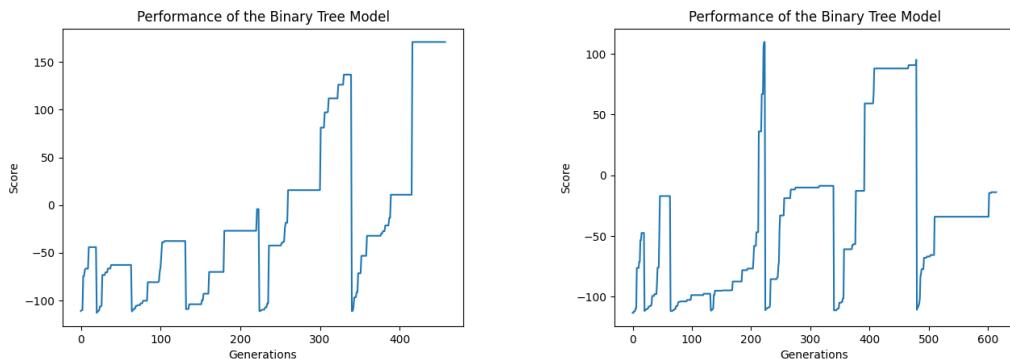
FIGURE 4.1: Plots of the lunar lander experiment

### 4.3.2 Bipedal walker

The Bipedal Walker was not solved with the current implementation, as 300 (-300 for CMA-ES) points were not obtained throughout the experiment. However, a larger

tree seemed to find more individuals with high fitness. In this experiment, the initial structure showed better results when starting with a constant as the single node's function rather than with a perceptron. The maximum number of steps was set to 1600. For the CMA-ES optimizer, the same values were chosen for the parameters as for the Lunar Lander experiment. The *stag\_step* variable, explained in 4.3.1, was set to 0.2 in this case.

To assess the impact of tree size on performance, the experiment was run for 45 minutes. Figure 4.2a shows the line plot of scores over generations. Similar to the Lunar Lander, large reductions in the plot indicate an addition of nodes to the tree. With small trees, the individuals perform poorly (less than zero points). From the fifth bar onwards (which corresponds to a binary tree with nine nodes, as we start with one node and always add two nodes with the *add\_node* function), the score of the best-performing individuals with the model increases rapidly, achieving scores over 100 points. However, it is important to note that this is not always the case. Some experiments achieve lower scores even with big tree structures. For example, when the experiment was run for one hour, the results in Figure 4.2b showed different results. The fourth and sixth bars indicate very high scores (seven and eleven nodes), but the fifth and seventh bars show that the individuals had scores of less than zero points (nine and thirteen nodes). This shows that an increasing size of the tree does not necessarily mean that individuals with high scores will be generated. In the case of this example, the highest scores were obtained with a tree structure of seven nodes.

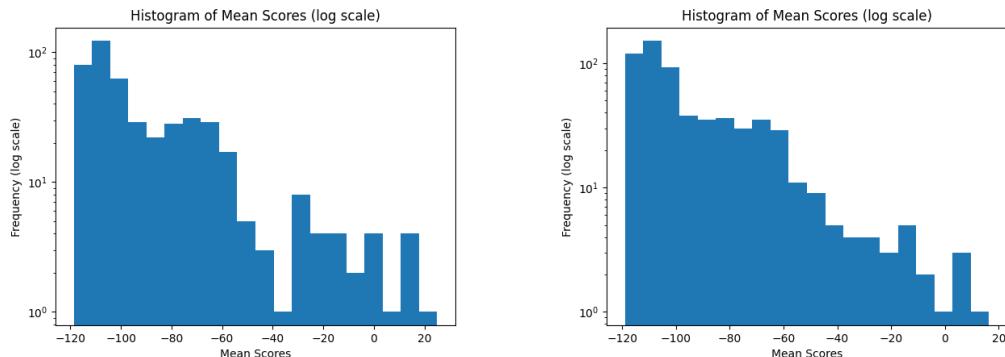


(A) **Score over generations lineplot of the bipedal walker** Evolutions of the best performing individual in the bipedal walker environment run for 45 minutes with an increasing binary tree size over the generations

(B) **Score over generations lineplot of the bipedal walker** Evolutions of the best performing individual in the bipedal walker environment run for one hour with an increasing binary tree size over the generations

The log-scaled histogram in Figure 4.3a provides the mean scores of the populations, as calculated over 15 individuals. The experiment was run for 45 minutes. The histogram shows that, given the complexity of the environment, a large proportion of populations had low mean scores. Most of the means were below zero points and the amount decreased as the mean scores increased. There were no populations with a mean score over 20 points after 45 minutes of the experiment, which indicates that the task is still far from being solved (the target score is 300 points). The results of the same histogram after running the experiment for one hour (Figure 4.3b) showed similar results.

It is important to note that all of these experiments were run using the default reward functions, without fine-tuning. In the case of the more complex Bipedal Walker environment, this led to suboptimal results. The walker often became stuck in local



(A) **log-scale histogram of the mean scores obtained in the bipedal walker environment run for 45 minutes** The scores were obtained with a growing binary tree for one hour even if the environment was not solved

(B) **log-scale histogram of the mean scores obtained in the bipedal walker environment run for 1 hour** The scores were obtained with a growing binary tree for one hour even if the environment was not solved

optima, which prevented it from exploring better ways of moving. For example, the walker often remained balanced on its two legs without falling, as shown in Figure 4.4. This is because it does not receive a large penalty for remaining in that state, whereas taking a step forward, which could lead to learning a more efficient way to walk, would result in a fall and a penalty.

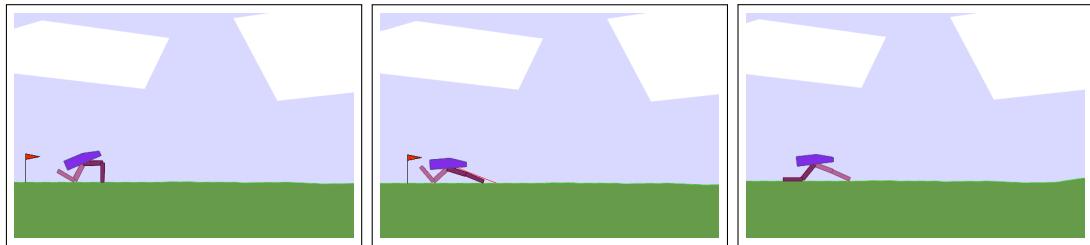


FIGURE 4.4: Different states of the Bipedal walker environment where the walker got stuck in local optima



## Chapter 5

# Conclusion

### 5.1 Conclusion

In this work, we improved the binary tree model as an alternative for neural networks in solving reinforcement learning problems by adding a function that enables trees to grow dynamically depending on the complexity of the task to solve. The first goal of the project was to make the model work, which was a success. The newly implemented function that should enable the model to increase the size of the tree by randomly selecting the place to add new nodes was also implemented successfully.

The new model showed that it could solve the *LunarLander* environment with discrete actions rapidly. However, for the *BipedalWalker* environment, which has a continuous action space, it had more difficulties than expected. The task often got stuck with relatively low scores. It is, however, a comprehensive approach due to the simplicity of the `add_node` function, which is only a first step toward architecture search for binary trees.

### 5.2 Future Work

The continuation of this work includes fitness shaping as a first step to see if the "Bipedal Walker" task can be solved with the current implementation of the binary tree. It would also be interesting to investigate the performance or the model on other environments to evaluate the robustness of the model in solving reinforcement learning problems and could provide valuable insights into its capabilities and limitations. Furthermore, a crucial aspect would be to explore other methods for architecture search using binary trees. The scope of this project is limited as it can only grow the tree by randomly selecting the place to insert new nodes, the addition of new nodes is done when a linear threshold is reached based on the size of the tree, the functions of the nodes are set equally for all nodes of the tree (only distinguishing between leaf and non-leaf nodes), etc. All these aspects could be modified and tested to improve architecture search for binary trees. Also comparing the performance of the binary tree model to that of traditional neural networks on various tasks could provide a better understanding of the potential advantages and disadvantages of using binary trees as an alternative.

Finally, binary trees remain one possibility for an alternative to neural networks. It would be interesting to research other models that address the current limitations of neural networks.



# Bibliography

- Akimoto, Youhei et al. (Dec. 2012). "Theoretical Foundation for CMA-ES from Information Geometry Perspective". en. In: *Algorithmica* 64.4, pp. 698–716. ISSN: 0178-4617, 1432-0541. DOI: 10.1007/s00453-011-9564-8. URL: <http://link.springer.com/10.1007/s00453-011-9564-8> (visited on 01/24/2023).
- Anderson, James A. (1995). *An Introduction to Neural Networks*. en. Google-Books-ID: \_ib4vPdB76gC. MIT Press. ISBN: 978-0-262-51081-3.
- Barron, E.N. and H. Ishii (Sept. 1989). "The Bellman equation for minimizing the maximum cost". en. In: *Nonlinear Analysis: Theory, Methods & Applications* 13.9, pp. 1067–1090. ISSN: 0362546X. DOI: 10.1016/0362-546X(89)90096-5. URL: <https://linkinghub.elsevier.com/retrieve/pii/0362546X89900965> (visited on 12/16/2022).
- Box2D: Overview* (n.d.). URL: <https://box2d.org/documentation/index.html> (visited on 01/20/2023).
- Brockman, Greg et al. (June 2016). *OpenAI Gym*. en. arXiv:1606.01540 [cs]. URL: <http://arxiv.org/abs/1606.01540> (visited on 02/28/2023).
- Coggan, Melanie (n.d.). "Exploration and Exploitation in Reinforcement Learning". en. In: () .
- Cuccu, Giuseppe, Julian Togelius, and Philippe Cudre-Mauroux (Mar. 2019). *Playing Atari with Six Neurons*. en. arXiv:1806.01363 [cs, stat]. URL: <http://arxiv.org/abs/1806.01363> (visited on 02/10/2023).
- Elsken, Thomas, Jan Hendrik Metzen, and Frank Hutter (n.d.). "Neural Architecture Search: A Survey". en. In: () .
- Franklin, Gene F., J. Da Powell, and Abbas Emami-Naeini (2014). *Feedback Control of Dynamic Systems*. 7th. USA: Prentice Hall Press. ISBN: 0-13-349659-7.
- Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (Nov. 2016). *Deep Learning*. en. Google-Books-ID: omivDQAAQBAJ. MIT Press. ISBN: 978-0-262-33737-3.
- Goodrich, Michael T (n.d.). "Data Structures and Algorithms in Python". en. In: () .
- Masanti, Corina (n.d.). "Alternative Models for Direct Policy Search in Reinforcement Learning Control Problems". en. In: () .
- Mellor, Joseph et al. (n.d.). "Neural Architecture Search without Training". en. In: () .
- Oller, Declan, Tobias Glasmachers, and Giuseppe Cuccu (Apr. 2020). *Analyzing Reinforcement Learning Benchmarks with Random Weight Guessing*. en. arXiv:2004.07707 [cs, stat]. URL: <http://arxiv.org/abs/2004.07707> (visited on 01/24/2023).
- OpenAI Gym Beta* (Apr. 2016). en. URL: <https://openai.com/blog/openai-gym-beta/> (visited on 01/08/2023).
- Recht, Benjamin (Nov. 2018). *A Tour of Reinforcement Learning: The View from Continuous Control*. en. arXiv:1806.09460 [cs, math, stat]. URL: <http://arxiv.org/abs/1806.09460> (visited on 02/10/2023).
- Salimans, Tim et al. (Sept. 2017). *Evolution Strategies as a Scalable Alternative to Reinforcement Learning*. en. arXiv:1703.03864 [cs, stat]. URL: <http://arxiv.org/abs/1703.03864> (visited on 12/19/2022).
- Schaul, Tom (n.d.). "Studies in Continuous Black-box Optimization". en. In: () .

- Sutton, Richard S. and Andrew G. Barto (Feb. 1998). *Reinforcement Learning: An Introduction*. en. Google-Books-ID: U57uDwAAQBAJ. MIT Press. ISBN: 978-0-262-30384-2.
- (Nov. 2018). *Reinforcement Learning, second edition: An Introduction*. en. Google-Books-ID: uWV0DwAAQBAJ. MIT Press. ISBN: 978-0-262-35270-3.
- Watkins, Christopher J. C. H. and Peter Dayan (May 1992). “Q-learning”. en. In: *Machine Learning* 8.3, pp. 279–292. ISSN: 1573-0565. DOI: 10 . 1007/BF00992698. URL: <https://doi.org/10.1007/BF00992698> (visited on 12/16/2022).
- Zhu, Henry et al. (2020). “THE INGREDIENTS OF REAL-WORLD ROBOTIC REINFORCEMENT LEARNING”. en. In: p. 21.