UNI FR
UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

eXascale Infolab

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

Boulevard de Pérolles 90 • 1700 Fribourg • Switzerland
phone +41 (26) 300 84 65 • diuf-secr@unifr.ch • www3.unifr.ch/inf

Abstract

David Gauch

Thesis Title

Write the thesis abstract here. Should be between half-a-page and one page of text, no newlines.

Keywords: keywords, list, here

Contents

List of Figures

Chapter 1

Introduction

1.1 Reinforcement learning

Reinforcement learning is a type of machine learning that focuses on training agents to make decisions in dynamic environments in order to maximize a reward signal. It is distinct from other paradigms 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. In addition to being a problem that can be addressed with specific solution methods, reinforcement learning is also the field of study that examines this problem and its potential solutions. Overall, the goal of reinforcement learning is to teach agents to make optimal decisions in order to achieve a desired outcome or reward.(Sutton and Barto, 2018).

Classical reinforcement learning involves the interaction between two main components: an environment and an agent. The environment is represented by a current state that provides information about the "world" in which the agent is located, and the agent is the individual who performs actions within this environment. In each interaction, the agent receives observations based on the current state of the environment and selects an action to take. This action is then transmitted to the environment, which updates its internal state and provides feedback to the agent in the form of observations and a reward. The reward signal indicates whether the action was suitable for completing the task, while the observations provide an overview of the updated environment. Figure 1.1 illustrates one timestep of interaction between the agent and the environment.

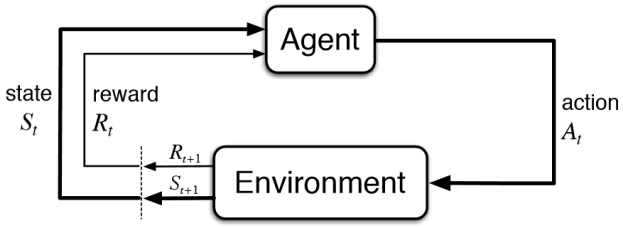

FIGURE 1.1: **Main interaction of the agent and the environment in reinforcement learning.** At the beginning (timestep t) the agent gets the observation S_t and the reward R_t from the environment. The agent performs then action A_t and sends it to the environment. The environment changes its state 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. 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 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 π, starting from state s. One way to compute the value function is using the Bellman equation, which expresses the value of a state in terms of the values of its successors(Barron and Ishii, 1989). However, 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 π 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.

While reinforcement learning has been effective in solving a range of tasks, it has also encountered challenges in real-world applications (Zhu et al., 2020). In this thesis, however, the paradigm is sufficient for addressing the desired tasks. Overall, reinforcement learning offers a powerful tool for training agents to make decisions in dynamic environments and optimize for a given reward signal.

1.2 Black-Box Optmization

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

1.2.1 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 (Salimans et al., 2017). 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.

1.2.2 Covariance Matrix Adapation Evolution Strategy

1.3 Binary trees

Trees are a commonly used data structure in mathematics and computer science that are composed of nodes and edges. A tree is an undirected, connected, acyclic graph, which means that it consists of a set of nodes that are connected by edges, but there are no loops or cycles in the graph. In a tree, a node that connects other nodes is called a parent node, and the nodes that are connected to it are called children nodes. The node at the top of the tree, which has no parent nodes, is called the root node, and the nodes at the bottom of the tree, which have no children, are called leaf nodes. The levels of a tree are determined by the distance from the root node, with the root node being at level 0 and the nodes connected to it being at level 1, and so on. Nodes on the same level are called sibling nodes.

Binary trees are a special type of tree in which all nodes except for the leaf nodes have at most two children nodes. These children nodes are typically referred to as the left node and the right node, respectively. An example of a binary tree is shown in Figure 1.2.

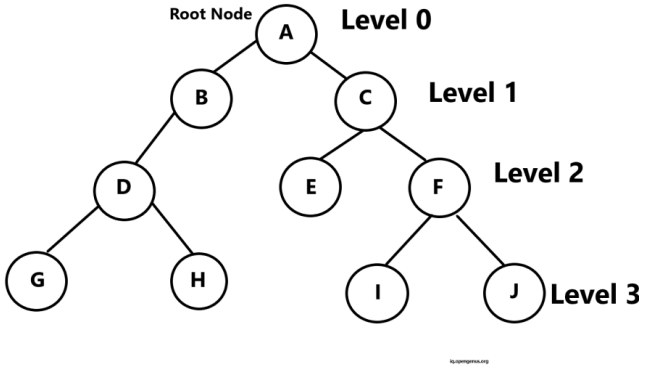

FIGURE 1.2: **Binary tree with letters representing the data**

1.4 Architecture search

Neural networks have seen significant improvements in recent years, particularly with the abundance of data available. Deep neural networks are now able to solve a wide range of problems such as image recognition and translations. However, finding the optimal number of layers and nodes (referred to as the architecture) for a neural network to solve a problem efficiently remains a challenging task. One approach is to try different architectures and evaluate their performance. This can be difficult when dealing with deep neural networks with high complexity. Another approach is to use neural architecture search, which automates the process of finding an appropriate architecture for a specific task. Research is ongoing to find effective methods for architecture search. Currently, evolutionary algorithms, reinforcement learning, and gradient-based optimization, or a combination of these techniques, are commonly used to search the space of possible architectures (Elsken, Metzen, and Hutter, n.d.).

1.5 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.

1.6 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 is 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).

To start working with the toolkit, the first step is to generate an instance of a specified envrionment. 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 wheter 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 2

Method

2.0.1 Experiment of this project

In the context of this project, the observation is that current models are typically continuous (due to back-propagation), but many control tasks are not. An example of this is the problem of the pendulum 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. When attempting to approximate the function that models this task, it becomes apparent that the function is not continuous. 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. To test this hypothesis, multiple individuals can be evaluated in the environment (by going through the fitness function) and analyzing their performance, the number of individuals that solve the task, and other metrics. Hyperparameters also play an important role in improving the performance of individuals in the environment.

2.1 Fitness

The fitness class serves as an interface between the model and the environment. It implements the control loop at its core, through the fit function. The control loop essentially evaluates how well a set of weights performs in an environment by selecting actions, as determined by the model, and outputting a score based on its performance.

Algorithm 1 fit

```
reset the environment and get the observations
set the weights of the individual in the model
score = 0
done = False
for number of step nsteps do
    get the action given from the model and the actual observation
    execute an action step and get the state of the environment
    increment the score with the obtained reward from the action step
    if done then
        break
return score
```

2.2 Environments

For this project, two environments of the *Box2D* category of OpenAI Gym were used[1]. These environments are more complex than the "Classical Control" problems and are highly configurable. Box2D is a 2D physics engine for games that can be used to make objects move in a realistic way and make the game more interactive.(*Box2D: Overview* n.d.).

2.2.1 Lunar Lander

The Lunar Lander environment consists of a rocket attempting to land between two flags on the surface of the Moon. The rocket can use three engines, which can be fired at full speed or turned off. The environment has both a continuous and a discrete version. For this project, the discrete version was used. Figure 2.1 illustrates the different states in which the rocket can be during the landing.

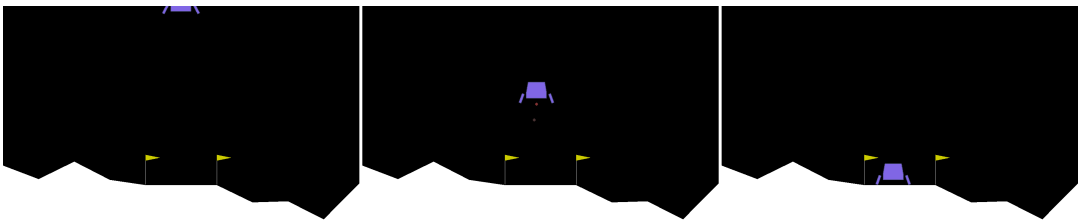

FIGURE 2.1: **Different states of the Lunar Lander environmment** The left image illustrates the rocket at the beginning of the task. The image in the middle shows the rocket firing its main engine in order to adjust its trajectory and the right image shows a successful landing between the two flags.

Action space

The environment has four actions that can be used. It can either do nothing or fire with the engine on the left, on the right or the main engine, which points downwards. The power at which the engines fire cannot be adjusted (it can only be turned on or off). This means the action space is of dimension 4 and discrete.

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

Rewards

For the agent, starting from the top of the screen and successfully landing on the landing pad, it receives a reward of 100-140 points. If the rocket crashes, it receives a negative reward of -100 points. Touching the legs of the lander to the ground gives an additional reward of +10 points per leg and firing the engine gives a small penalty. The task is considered solved when the agent receives a total reward of 200 points or higher.It's worth noting that the rewards points may vary depending on

[1] https://www.gymlibrary.dev/environments/box2d/

Observation	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

the specific implementation of the environment, and that the exact points given for each action, state or event are not fixed values but can be adjusted to fine-tune the agent's behavior.

2.2.2 Bipedal Walker

This environment simulates a two-legged robot attempting to walk as far as possible on uneven terrain. Two versions are available: a "normal" version and a more challenging "hardcore" version 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 2.2 illustrates the robot in action on both versions.

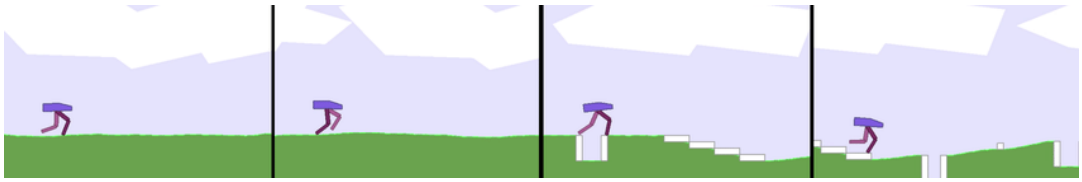

FIGURE 2.2: **Bipedal walker walking in both versions of the environment** The left two frames illustrate the bipedal walker in different positions in the "normal" version of the environment. The right two frames show the robot attempting to walk over the obstacles in the "hardcore" version.

Action space

The actions of the bipedal walker are continuous. There are four actions that can be performed, one for each joint. The values range between -1 and 1 and indicate the motor speed values of the corresponding joint. Each action affects the movement and stability of the robot, allowing it to walk or fall. The range of values can be adjusted depending on the implementation of the environment.

Observation space

The observation space for the bipedal walker is of dimension 24 and contains continuous values, except for the booleans that indicate whether the legs are touching the ground or not. The observation space contains information such as the angle and angular velocity of each joint, the linear velocity, and the position of the torso, among others. Note that the position of the robot is not explicitly provided in the observation space, but it can be inferred from the other observations, such as the linear and angular velocities of the joints.

Observation	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 and it is penalized by -100 points. If the bipedal walker reaches the end of the environment, it accumulates 300 points. Each time the robot moves its joints, it also receives a small penalty. 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.

2.3 Model

n this project, binary trees are employed as a model architecture instead of traditional neural networks. The optimization of these models is done using black-box optimization techniques. One of the advantages of using binary trees is their interpretability, as it is easier to understand the logic behind the decision-making process and explain the reasoning behind the model's predictions.

2.3.1 Node

A node in the binary tree is composed of a pointer that points to its parent node, a function that is part of the function class. The function applied at the node is used to make a decision or perform a computation based on the input data. An amount of weight the node contains, which is used to adjust the importance of the decision or computation made at that specific node. Lastly, two pointers, one to its left child and the other to its right child, which are used to navigate through the tree and make decisions based on the input data and the function applied at each node. Figure 2.3 illustrates a tree with a single node.

2.3.2 Functions

Each node in the binary tree structure contains a function that is used to make decisions or perform computations based on the input data. In this project, three function types were implemented: a constant function, a linear function, and a perceptron. The constant function returns the weights as output, regardless of the input values. The linear function returns the dot product of the weights and the observations it received as input. The perceptron uses a specific activation function, in this

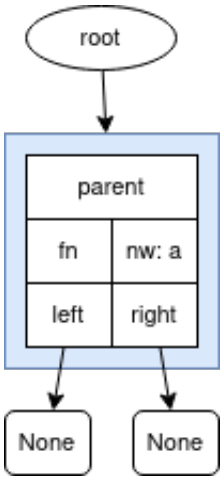

FIGURE 2.3: **Components of a single-node tree** This represents a binary tree with a single node. The *root* pointer points to the node, and the *parent* pointer points to nothing as it is a single-node tree. The main components of the node are a function fn that is implemented in the function class, 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.

case the sigmoid,

$$1 \div (1 + \exp(-x)) \tag{2.1}$$

to transform the dot product between the weights and the observations (noted as x) into a scalar value that can be used to perform computations. Each instance of the function class contains a number of inputs and outputs, the weights used by the function, and the number of times the function was activated. The weights can be learned or fixed, and the input and output values can take a specific range. Note that for nodes in the tree that are not leaf nodes, it is convenient to use linear functions as the output will be a scalar value that is useful for the traversal of the tree. The specific details of how the functions, weights and pointers are used to make decisions or perform computations in the tree, can be found in the activate function in 2.3.3.

2.3.3 Binary tree

A binary tree is made up of linked nodes that contain functions. Figure 2.4 illustrates a binary tree with a root node and two child nodes. A binary tree is a structure composed of linked nodes that contain functions that are used to make decisions or perform computations based on the 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, perceptron) and two child nodes that can be either leafs or internal nodes.

The process of using the tree to make decisions or perform computations is known as activation. The activate function starts from the root of the tree and navigates through the tree by following the links between the nodes based on the output of the function of the current node. When it reaches a leaf node, it returns the output of the function of that node as the final output of the tree.

The construction of the tree is done by deciding the decision points in the tree, these decision points are chosen based on the problem to solve and the input data. The tree can be trained and updated by adjusting the functions, weights, and links between the nodes.

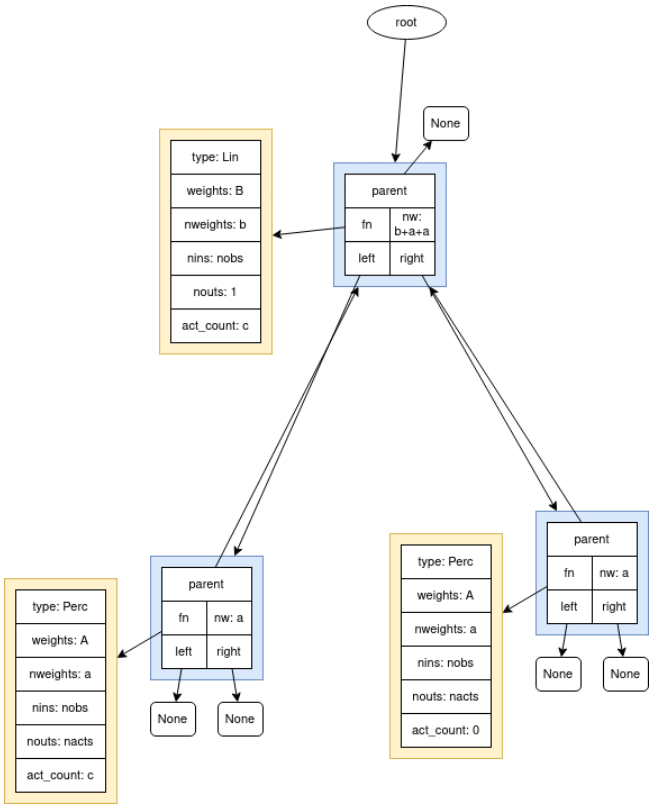

FIGURE 2.4: **Components of a tree with three nodes** Representation of a tree with a root node containing a linear function and two child nodes with perceptrons. Each node shows its pointer to the corresponding function instance (yellow blocks) and the links to each other. The *n* before variables or values stands for "number," thus *nobs* means the number of observations, for example.

The binary tree has some advantages and limitations when compared to other models such as neural networks and decision trees. One of the advantages is that it can be interpretable, as the decision points and functions used in the tree can be explained. However, one of the limitations is that it can be sensitive to the size of the tree, making the efficiency of the tree depend on the size of the tree. This can be overcome by applying pruning techniques or other methods to optimize the tree structure. The difficulty of tasks can vary, making it important to have the ability to

Algorithm 2 *activate* function

```
starting from the root
while true do
    if node is a leaf then return output of node's function
    else if output of node's function > 0 then
        go to the left child node
    else
        go to the right child node
```

adjust the size of the tree accordingly. For instance, simpler problems like the Cartpole in OpenAI Gym can be solved with a small tree, while more complex problems require a larger tree. Increasing the size of the tree allows for more complex decision making and can improve the model's performance. However, it also increases

the risk of overfitting. In this project, a function has been implemented to automatically increase the size of the tree as the difficulty of the problem increases. This process of finding an optimal structure is known as architecture search. The implemented function illustrates only one of many possible techniques for increasing the tree structure.

The function works by randomly selecting a leaf node and adding two new nodes to it. It starts by traversing the tree randomly until it reaches a leaf node. Then, it checks if the selected leaf is the root of the tree or not. If it is the root, a new parent node is created, otherwise, the leaf's parent node is copied to create a new parent node. The new parent node is then set as the parent of the current leaf node and its copy. The new nodes are added to the tree in such a way that the relative position of the current leaf node to its previous parent node remains the same.

It is important to note that the function does not add two child nodes to the leaf node, but one new node is added as the parent of the leaf node and the other as its sibling. Figure 2.5 illustrates an example of adding two nodes to a binary tree. After the addition of the new nodes, all the links in the tree need to be fixed and the information about the number of weights and the amount of descendants needs to be updated. To accomplish this, the function uses a propagation of the information from the current leaf node to the root.

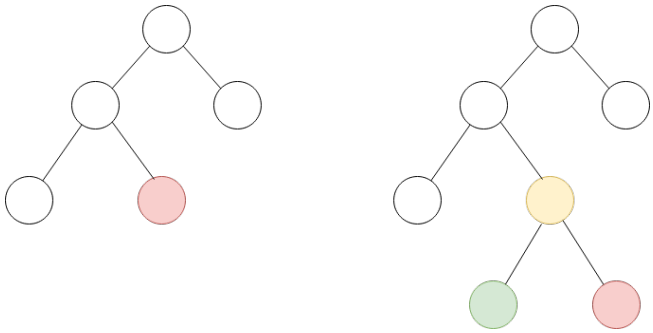

FIGURE 2.5: **Addition of two nodes in a binary tree** The left tree shows the initial tree before the addition of the nodes. The red node represents the randomly chosen leaf from which the node addition will follow. The tree on the right represent the tree after addign two nodes with the add_node function. The node in yellow is the newly created parent node and the green node is the sibling of the previously existing red node. Notice that the red node keeps its relative position to the parent node.

Algorithm 3 *add_node* function

```
    pick a random leaf
    if leaf is a root then
3:      create new_parent node (root) with a linear function
    else
        copy leafs parent into new_parent node
6:  create copy_node (copy of leaf)
    set copy_node and leaf as children of new_parent
    fix links
9:  propagate the count of new nodes and their weights up the tree
```

The function for increasing the size of the tree allows for the tree to adapt to the difficulty of the problem by growing according to a stagnation threshold, where the score of the individual does not improve for a certain number of steps. A larger tree has a larger search space, which allows it to solve more complex problems. However, it also increases the computation time. Therefore, it is important to find the right balance between growing the tree too quickly or too slowly.

Chapter 3

Experiments

3.1 Results

Bibliography

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).

Elsken, Thomas, Jan Hendrik Metzen, and Frank Hutter (n.d.). "Neural Architecture Search: A Survey". en. In: ().

OpenAI Gym Beta (Apr. 2016). en. URL: https://openai.com/blog/openai-gym-beta/ (visited on 01/08/2023).

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 (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.