UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

eXascale Infolab

UNIVERSITY OF FRIBOURG

MASTER THESIS

Alternative Models for Direct Policy Search in Reinforcement Learning Control Problems

Author:
Corina Masanti

Supervisors:
Dr. Giuseppe Cuccu
Prof. Dr. Philippe Cudré-Mauroux

August 11, 2022

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

Corina Masanti

Alternative Models for Direct Policy Search in Reinforcement Learning Control Problems

Neural networks as generic function approximators can solve many challenging problems. Regardless, they can only be applied successfully for a suited problem structure. A large part of the success of neural networks can be attributed to the efficiency of the backpropagation algorithm, which requires labeled data to estimate accurate error gradients. However, there are many areas where accessing labeled data is non-trivial, including problems in reinforcement learning. In contrast, black-box optimization techniques are less limiting and can be used to optimize the parameters of any function approximator. They presume no constraints on the problem structure, the model, or the solution. With this flexibility, we can study alternative models to neural networks that are yet unexplored in the context of reinforcement learning. This thesis aims to achieve comparable results with a function approximator other than neural networks. To this end, I analyzed the performance of polynomial and binary tree models on reinforcement learning benchmark problems. I compared these results to those of neural networks. The results show that the binary tree model can produce results comparable to neural networks, and even outperforms them, while providing essential advantages. The simple structure of the binary tree model makes hyperparameter tuning straightforward and provides high interpretability.

Keywords: black-box optimization, reinforcement learning, binary trees, neural networks, random weight guessing

Contents

List of Figures

Chapter 1

Introduction

1.1 Reinforcement Learning Control Problems

Control problems appear in the context of both reinforcement learning and control theory. Thus, we often see vocabulary specific to control theory in the context of reinforcement learning. A control problem, generally explained, is a dynamic system described by state variables. Controls (actions) determine how the system will behave in the future. The goal of a control problem is to work out a strategy that causes the system to terminate in the desired target state. Reinforcement learning and approaches in control theory offer a framework to solve such control problems (Buşoniu et al., 2018). Even though the two fields embark on the same problem, the research communities remained disjoint primarily, leading to different approaches to the same problem. Reinforcement learning often makes model-free predictions from data, whereas control theory often defines well-specified models (Recht, 2018). This thesis will focus on reinforcement learning frameworks to solve control problems.

Reinforcement learning is an area of machine learning that involves learning the optimal behavior to maximize a reward signal inside a dynamic environment. Trial-and-error interactions with the environment mark the learning process. The learner does not know which actions lead to which rewards until he tries them (Kaelbling, Littman, and Moore, 1996). Reinforcement learning differs from supervised learning, where learning is achieved through a training set with labeled examples. Each sample of the training data represents a description of the situation and a label that marks how the system should react. The learning process aims to generalize the optimal actions given a specific situation so that the system acts correctly in situations outside the training set.

In reinforcement learning, there are no labels available beforehand. In addition, we can also not classify reinforcement learning as unsupervised learning, where we typically try to find a hidden structure in a set of unlabeled data. With reinforcement learning, we do not try to find some structure in the data, but solely try to maximize the reward signal. Therefore, reinforcement learning is considered a third main machine learning paradigm alongside supervised and unsupervised learning.

One additional challenge often overlooked in other branches of machine learning is a focus on the trade-off between *exploration* and *exploitation*. For a high reward, the reinforcement learning agent should take an action that it has tried in the past and was effective. It should therefore exploit the knowledge gained previously. However, to discover this beneficial action, it has to explore actions it has not taken before (Sutton and Barto, 2018).

The agent's goal in a reinforcement learning problem is to drive the system into the desired state by taking the correct sequence of actions. For example, the goal might be to solve non-trivial skill games like Ball-in-a-Cup or Kendama (Kober,

Mohler, and Peters, 2010). The agent's behavior at a particular time is defined by the policy π. A policy is a mapping from each state s of the environment to each action a the agent can perform in that specific state (Sutton and Barto, 2018). We can write π as

$$\pi : \mathcal{S} \rightarrow \mathcal{A},$$

where $\mathcal{S}$ is defined as the set of states and $\mathcal{A}$ is the set of actions. Policies can be stochastic or deterministic. Deterministic policies assign each state to a specific action. Stochastic policies define a probability distribution over $\mathcal{S}$. The underlying goal in reinforcement learning is for the agent to learn or approximate the optimal policy which determines the best action to take in a particular state.

To overcome the mentioned difficulties unique to reinforcement learning, we need to experiment with alternative models and strategies than those used for supervised or unsupervised learning.

1.2 Classic Reinforcement Learning

In reinforcement learning, we have an agent (also called a controller) and an environment. The agent can take on actions to interact with the environment. On the other hand, the environment always holds a specific state that is impacted by the agent's actions. As mentioned, the goal is to find the optimal policy while only receiving a scalar reward signal that indicates how well the agent performs. The agent's goal is to maximize the cumulative rewards received by the environment. Therefore, the agent needs to learn optimal behavior only through trial-and-error interaction with the environment. This aspect makes the problem structure much more challenging.

Figure 1.1 illustrates the loop between the agent and the environment. The agent receives the observation o_t from the environment based on the environment's current state. However, not all the information about the environment may be present in the observation. If there is missing information about the environment's current state, the environment is only partially observable. If the observation contains full knowledge, the environment is fully observable (Dong et al., 2020). The agent then

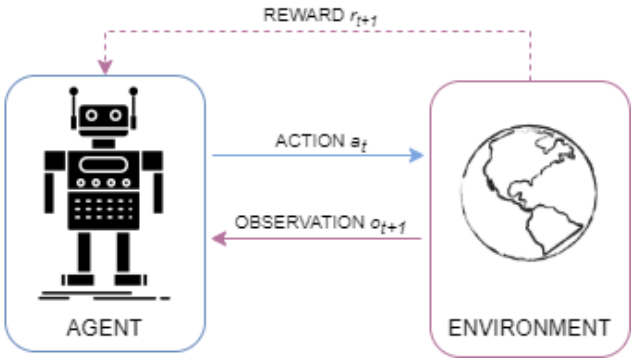


FIGURE 1.1: **Interaction loop between an reinforcement learning agent and the environment.** At timestep t, the agent receives the observation o_t from the environment. The agent decides to select action a_t and interacts with the environment. The enviornment changes and returns the next observation o_{t+1} and the reward r_{t+1} to the agent.

performs some action a_t in the environment. The environment's state changes according to the received action and returns an observation o_{t+1} based on the updated

state of the environment and a reward r_{t+1}. One interaction between the agent and the environment represents one *timestep*.

The rewards are computed by an underlying reward function specific to the environment. The reward function $\mathbf{R}$ is a function that maps the state of the environment and the action of the agent into a scalar value. Formally, we can write $\mathbf{R}$ as:

$$\mathbf{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R},$$

The reward function can be deterministic or can contain a random component making it stochastic. Depending on the problem, the reward can be sparse or dense. Sparse reward means that the reward function returns zero in most of its domain and only returns a meaningful reward in a few rare settings or at the end of the experiment. That introduces a major challenge since we cannot know which actions lead to success or failure. Optimization problems with this setting are generally challenging to solve. Dense reward environments return a meaningful reward at every step, making the optimization process more manageable.

Learning a policy that maximizes cumulative rewards is generally non-trivial. If the learning algorithm is not explorative enough, it might get stuck in a local optimum and never reach the goal. However, if there is too much exploration, we receive less overall reward. Thus, there needs to be a balance between exploration and exploitation. In addition, there is often a time delay for the reward. The consequence of an action might not be immediately visible to the agent. This problem is known as the credit-assignment problem in the literature (Sutton and Barto, 2018).

We can solve reinforcement learning problems with methods based on value functions or policy search. Methods based on policy search directly search for the optimal policy π^* and do not need to maintain a value function model (Arulkumaran et al., 2017). For methods based on value functions, the state-value function $\mathcal{V}$ defines the expected return of each possible state. It indicates how good or bad being in a particular state is. The agent chooses the best action based on the value of the states. The value of state s under policy π is defined by $\mathcal{V}_\pi(s)$ and indicates the agent's expected total reward following policy π starting from state s (Sutton and Barto, 2018).

Value functions induce a partial order on policies. Policy π is better than policy π' if and only if $\mathcal{V}_\pi(s) \geq \mathcal{V}_{\pi'}(s)$ applies to all states $s \in S$. The optimal policy π^* leads to the optimal state-value function:

$$\mathcal{V}_{\pi^*}(s) = \max_\pi V_\pi(s)$$

To find π^*, we can use the *Bellman optimality equation* which divides the value function into the immediate reward and the discounted future reward. The optimal value function then results in:

$$\mathcal{V}_{\pi^*}(s) = \max_{a \in \mathcal{A}} \mathbf{R}(s, a) + \gamma \sum_{s' \in S} \mathcal{P}^a_{s,s'} \mathcal{V}_{\pi^*}(s')$$

The discount factor γ determines the importance given to the immediate reward and the future rewards. $\mathcal{P}$ is the transition function where $\mathcal{P}^a_{s,s'}$ defines the probability of moving from state s to state s' while using action a. There are many methods, for example value iteration or policy iteration, to solve this equation and find or approximate the optimal policy.

Another function that can be used to evaluate the quality of a policy is the Q function. It is closely related to $\mathcal{V}$, but instead of mapping a state to a value, it maps

state-action pairs to a value. More formally, we can write the relationship between the two functions as

$$\mathcal{V}_\pi(s) = \mathcal{Q}_\pi(s, \pi(s)).$$

The process of finding the optimal mapping $\mathcal{Q}^*$ and the optimal policy with the help of $\mathcal{Q}$ is called Q-learning.

As interest in neural networks and deep learning increased, deep reinforcement learning, which uses a neural network to estimate policies or value functions, attracted more attention.

1.2.1 Neural Networks as Models

Artificial neural networks are machine learning algorithms inspired by the functionalities of the human brain. They are applied successfully in many areas including in reinforcement learning problems. In 1958, Frank Rosenblatt implemented the perceptron, which represents a simplified version of a biological neuron (Rosenblatt, 1958). It was an early type of what later should be called artificial neural networks. On this idea, the concept was born and further developed throughout the years.

Neural networks in the modern sense consist of connected nodes that imitate the biological neurons of the brain, sending signals to each other. A node receives one or multiple input values and calculates its output value. The information is then passed onto the next node. Figure 1.2 illustrates the structure of one node. To calculate the

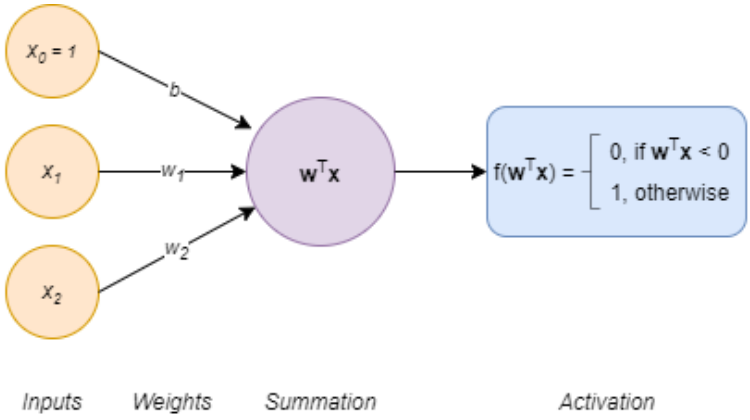


FIGURE 1.2: **Illustration of an artificial neuron.** The image illustrates how an artificial neuron works. The neuron receives two input values x_1, x_2 and one bias term b. It first calculates the weighted sum of the inputs. The activation function then defines the neuron's output passed to the next neuron.

output value, it receives two input values x_1, x_2 and one bias term b. We often use a bias, allowing us to make affine transformations to the data. The node uses the assigned weights $\mathbf{w}$ to calculate the weighted sum of input values. In more detail, the calculation is the following:

$$\mathbf{w}^T\mathbf{x} = w_1x_1 + w_2x_2 + b$$

For convenience, the following images will not show the bias term explicitly.

The activation function f then defines the output value that should be passed into the next neuron. The example shown in Figure 1.2 uses a step function with a threshold of 0. Many other functions can be used as activation functions. Another

commonly used function is the logistic function. Figure 1.3 shows the formula and a plot of it in 2D. The logistic function is a mathematical function that maps an arbi-

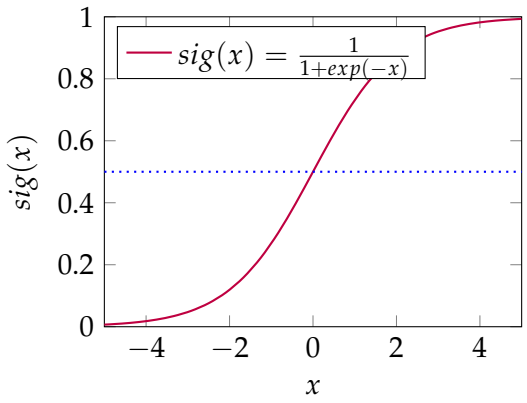


FIGURE 1.3: **Logistic function.** The function maps an arbitrary input space into the range between 0 and 1. The output of the function can be interpreted as a probability. The sigmoid function helps scale data into a meaningful value.

trary input space into an output space with a small range, for example, 0 and 1. The function has a characteristic S-shaped curve.

In a neural network, the nodes are arranged in connected layers. A network has at least an input and an output layer. We can further expand it with one or more hidden layers. Depending on the model, the connectivity between the layers differs. Figure 1.4 shows a sketch of a network with three hidden layers that are fully connected. A neural network with one or more hidden layers is theoretically able to

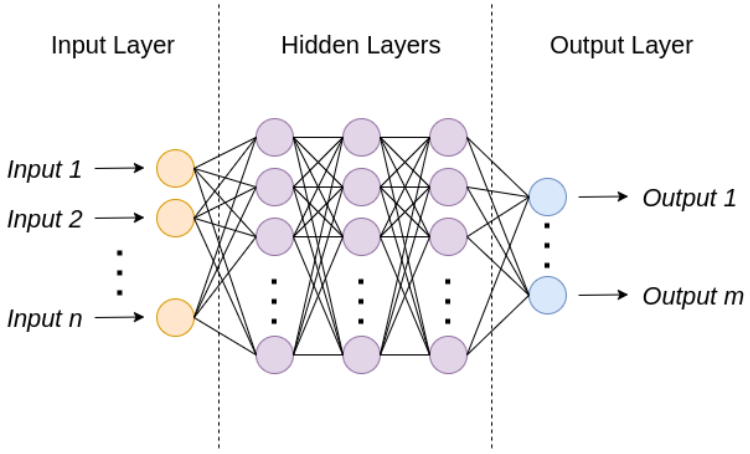


FIGURE 1.4: **Sketch of a neural network.** The image shows a sketch of a neural network with three fully connected hidden layers. The network expects n inputs and has m outputs.

represent any function. Thus, neural networks are generic function approximators.

Deep reinforcement learning combines reinforcement learning with the methods of deep learning. Deep neural networks have multiple hidden layers, allowing them to represent data with multiple levels of abstractions (LeCun, Bengio, and Hinton, 2015). Deep reinforcement learning has been used successfully in many applications, including robotics, video games, and computer vision (François-Lavet et al., 2018).

The main advantage is the scalability of deep neural networks in a high dimensional space (Dong et al., 2020).

A large part of the success of neural networks and deep learning can be traced back to the use of *backpropagation*. Backpropagation uses information from the previous epoch (i.e., iteration) to adjust the network weights using the error gradient. For the backpropagation algorithm to be efficient, it relies on calculating accurate gradients. In fields like reinforcement learning, where we only have a reward signal that might be time delayed, this is a non-trivial task. In addition, there is a risk of getting stuck in a local optimum (Ha, 2017).

In the context of reinforcement learning, neural networks as function approximators can be used to approximate the optimal policy π^*, the optimal state value function $\mathcal{V}^*$, and the optimal action-value function $\mathcal{Q}^*$ (Golovin et al., 2017). An important advantage over other methods like dictionary-based approaches is the generalization: the network can return the expected reward for all actions, even those yet unexplored.

In deep learning approaches in supervised learning, a training set with labeled data is available. In contrast, reinforcement learning requires dynamical learning based on the continuous feedback of the environment. For value-based methods like Q-learning, we can iteratively adapt the value function based on the received reward. However, in policy-based approaches, we need to evaluate the policy somehow. In supervised learning, we would use backpropagation. However, we do not have a training set in reinforcement learning and cannot guarantee differentiability. There are several methods to still use neural networks as policy approximators, one of them is *Neuroevolution*, which will be discussed in Section 1.5

1.3 OpenAI Gym

The research community requires high-quality benchmarks to compare algorithms and models applied to problems in reinforcement learning. There were already a few released benchmarks like the Arcade Learning Environment (Bellemare et al., 2013) or the RLLab benchmark for continuous control (Duan et al., 2016). OpenAI Gym[1] combined the elements of these existing benchmarks in one toolkit. It provides a collection of tasks (called *environments*) with varying levels of difficulties and challenges. The interface is the same for all environments, making it very convenient to use. The environments represent general reinforcement learning control problems. To interact with an environment, we first have to initialize it. We can do this with the predefined function `make()`. The following Python code shows the initialization of the environment `CartPole`:

```
1 import gym
2 env = gym.make("CartPole-v0")
```

In the setting of reinforcement learning, we assume that an agent is interacting with the dynamic environment. In each step, the agent can take some action and receive a reward and a new observation from the environment. The environment predefines both the action and observation space. In practice, an agent returns an action `a0` with which we call the function `step()` of the environment. The next code snippet shows an example of this behavior:

[1] https://www.gymlibrary.ml/

```
observation, reward, done, info = env.step(a0)
```

The information returned by the environment applies to the current timestep. The variable `observation` contains an element of the observation space, `reward` includes the amount of reward as a result of the agent's action, `done` is a boolean value that indicates whether the episode has ended or not, and `info` is a dictionary that contains auxiliary diagnostic information. OpenAI Gym focuses on episodic reinforcement learning, where the agent acts in a series of episodes with a fixed length. The initial state of the environment in each episode is sampled randomly. An episode ends when specific termination criteria are met. For example, when the task is solved or when we max out on timesteps (Brockman et al., 2016).

1.4 Direct Policy Search

Although using value functions to solve reinforcement learning problems has worked well in many applications, there are limitations. If the state space is not entirely observable, convergence problems can arise (El-Fakdi, Carreras, and Palomeras, 2005). Additionally, in traditional value-based methods like Q-learning or temporal difference learning, the value function is computed iteratively with the help of bootstrapping. That often results in a bias in the quality assessment of state-action pairs for continuous state spaces. Another disadvantage of these methods is that the value functions are often discontinuous (Deisenroth, Neumann, Peters, et al., 2013).

Studies suggest that directly approximating a policy can be easier and delivers better results (Cuccu, Togelius, and Cudré-Mauroux, 2018; Sutton et al., 1999; Anderson, 2000). This alternative method to methods based on value functions is called *direct policy search*. With direct policy search, the algorithm directly searches in the search space of policy representations discarding any value function (Wierstra, 2010). Instead of approximating a value function, we approximate the policy directly with an independent function approximator. Thus, we directly approximate the action-state mapping.

1.5 Neuroevolution

Neuroevolution is a method that we can use for direct policy search. As discussed in Section 1.2.1, optimizing neural networks as policy approximators results in some problems. One solution is using evolutionary algorithms to optimize the network parameters instead of backpropagation. With these algorithms, estimating accurate gradients becomes obsolete, and the risk of being stuck in a local optimum is reduced. Evolutionary algorithms only require a measure of the network's performance, making them directly applicable for reinforcement learning problems. Thus, we can use it more widely in various problem classes. Such et al., 2017 showed that genetic algorithms, a subgroup of evolutionary algorithms, work well for the parameter search of neural networks. Even in complex reinforcement learning settings, including Atari and humanoid locomotion, they could achieve results at the scale of traditional deep reinforcement learning frameworks.

1.6 Black-Box Optimization

In black-box optimization, the problem structure, as well as the model, is unknown to the learning algorithm. Those methods optimize a parametrization without any constraints on the problem, model or solution needed. Unlike the gradient-based learning techniques there is no constraint on the model or error differentiability. The optimization is solely based on a score or cumulative reward, which makes these methods directly applicable to reinforcement learning problems. Since there are no constraints imposed on the architecture or properties of the model, we can use any function approximator we find suitable for a problem and optimize it with black-box optimization techniques.

Given an objective function $f(x)$, black-box optimization techniques aim to find the solution x^* that optimizes f using the least number of calls. Depending on the problem structure, this can be a maximization or minimization problem (Ollivier et al., 2017). The problem might be subject to some constraints formulated as equality or inequality equations. We can use black-box optimization algorithms to find the best parameters for any system as long as a function can measure its performance (Golovin et al., 2017). The parameterization is commonly referred to as the *individual* and the function is called *fitness function* in the context of black-box optimization. For reinforcement learning problems, the agent's performance is measured by the reward function. Thus, we can use this information to apply black-box optimization techniques where the cumulative reward determines the individual's fitness.

1.6.1 Random Weight Guessing

Random Weight Guessing (RWG) is the simplest black-box optimization method. Algorithm 1 shows the procedure with pseudocode. RWG repeatedly randomly ini-

Algorithm 1 RWG as described in Schmidhuber, Hochreiter, and Bengio, 2001

```
repeat
    Randomly sample model weights
until model corretly classifies all training sequences
return model weights
```

tializes the weights of a model maintaining the best so far, monotonically improving performance.

Schmidhuber, Hochreiter, and Bengio, 2001 were able to show that simple RWG can even outperform previously proposed more complex algorithms. Naturally, RWG is not especially performant. A problem might require many free parameters or high weight precision in practical applications. In these cases, RWG becomes infeasible for finding a suitable parametrization within a reasonable time. A more appropriate learning technique is suggested for real-life applications. Nevertheless, the results suggest that RWG can be used as an analysis method. Oller, Glasmachers, and Cuccu, 2020 also used an RWG algorithm to analyze the complexity of some basic OpenAI Gym benchmark problems.

1.6.2 Evolution Strategies

Evolution Strategies (ES) belong to the class of evolutionary algorithms best suited for continuous optimization. They are directly inspired by the idea of evolution in nature and use the concept of mutation and selection. ES are implemented in a loop

where one iteration is called a generation. New individuals (candidate solutions) are generated using the current parents in each generation. We continue with the next generation until a stopping criterion is met. ES are multi-agent algorithms that generate multiple parametrizations that independently explore a different path or area in the optimization space. This exploration prevents us from getting stuck in a local optimum.

(1 + 1)-ES: The population model (1 + 1) means that we maintain only one individual. For the next generation, we generate one new offspring as a variation of the parent, drawn from the normal distribution. For this, we adapt the mean of the distribution according to the parent. Then we keep either the parent or the child based on which had a better fitness score. Algorithm 2 shows this process in pseudocode.

Algorithm 2 (1 + 1)-ES in d dimensions

Parameter: $\sigma > 0$
Initialization: $x = x_0 \in \mathbb{R}^d$
while not done **do**
    Sample $x' \sim \mathcal{N}(x, \sigma^2 I)$
    if $f(x') \leq f(x)$ **then**
        $x \leftarrow x'$
return x

1.6.3 Covariance Matrix Adaptation Evolution Strategy

The Covariance Matrix Adaptation Evolution Strategy (CMA-ES) is an advanced (μ, λ)-ES that adapts not only the mean of the multivariate normal distribution but also updates the covariance matrix. Since we are in the field of black-box optimization techniques, there are no constraints on the problem, model, or solution of the problem. CMA-ES is a method for non-linear, non-convex optimization problems in a continuous domain. It has been successful in many applications with ill-conditioned objective functions and is considered the industry standard. The following technical information has mainly been summerized from Hansen, 2016.

Sampling of the population: For each iteration of the CMA-ES algorithm, a population of new individuals is generated by sampling a multivariate normal distribution. Formally, for generation $g + 1$, the population sampling can be written as

$$\mathbf{x}_k^{(g+1)} \sim \mathbf{m}^{(g)} + \sigma^{(g)} \mathcal{N}(\mathbf{0}, \mathbf{C}^{(g)}) \qquad \text{for} \quad k = 1, ...\lambda,$$

where $\mathbf{x}_k^{(g+1)}$ symbolizes the k-th individual from generation $g + 1$, $\mathbf{m}^{(g)}$ denots the mean value of the distribution at generation g, $\sigma^{(g)}$ is the standarddeviation at generation g, $\mathbf{C}^{(g)}$ is the covariance matrix at generation g, and λ is the size of the population. After the sampling of generation $g + 1$, we need to adapt the mean and the covariance matrix for the sampling of the next generation. Thus, we need to calculate $\mathbf{m}^{(g+1)}$ and $\mathbf{C}^{(g+1)}$.

Moving the mean: At the end of each iteration, we need to update the mean of the distribution. The new mean $\mathbf{m}^{(g+1)}$ is a weighted average of μ selected individuals from the generation $g + 1$. Thus, we select μ individuals from $\mathbf{x}_1^{(g+1)}, ..., \mathbf{x}_\lambda^{(g+1)}$. The

formula to update the mean for the next generation is the following:

$$\mathbf{m}^{(g+1)} = \mathbf{m}^{(g)} + c_m \Sigma_{i=1}^{\mu} w_i (\mathbf{x}_{i:\lambda}^{(g+1)} - \mathbf{m}^{(g)})$$

$$\Sigma_{i=1}^{\mu} w_i = 1, \quad w_1 \geq w_2 \geq ... \geq w_\mu > 0$$

$c_m \leq 1$ is the learning rate of the algorithm and is usually chosen as $c_m = 1$. μ is the population size with λ being the population size of the parent population with $\mu \leq \lambda$. The weights $w_{i=1,...,\mu} \in \mathbb{R}_{>0}$ are positive coefficients used for the recombination process. The variable $\mathbf{x}_{i:\lambda}^{(g+1)}$ symbolizes the i-th best indivdual from the $(g+1)$-th population in relation to its corresponding fitness score.

Adaption of the covariance matrix: In addition to updating the mean of the search distribution, we also need to adapt the covariance matrix. Calculating the adapted covariance matrix is significantly more complex than updating the mean and takes the most computational effort. More details can be found in Hansen, 2016.

1.7 Research Questions

The main goal of this thesis is to compare alternative models to neural networks. To analyze and compare the different models and their architectures, I formulated the following research questions:

1. How do neural networks compare with other function approximators?
2. What advantages and disadvantages can we see with other function approximators?
3. How does the bias influence the performance of the model?

Chapter 2

Method

2.1 OpenAI Gym Environments

OpenAI Gym offers five environments ready to use in their *classic control* suite: `CartPole`, `Acrobot`, `MountainCar`, `MountainCarContiuous`, and `Pendulum`. The initial state of each environment is stochastic. Out of all environments OpenAI Gym provides, the classic control environments are considered the easier ones to solve while still being used as baselines in multiple applications. Each environment defines an observation space and an action space. An agent performs some action from the action space and receives an observation describing how the environment's state changed. The information about the environments for this section was taken from the online documentation[1].

2.1.1 CartPole

The `CartPole` environment corresponds to the pole-balancing problem formulated in *Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems* (Barto, Sutton, and Anderson, 1983). A pole is attached to a cart that the agent can move along a one-dimensional rail. Fig. 2.1 shows two possible states of this problem. In the image on the left, the pole is perfectly balanced and upright. The right image shows the pole tilted to the left. Thus, the cart should in principle move to the left to correct the angle of the pole.

FIGURE 2.1: **Illustration of the environment CartPole.** The image shows two possible states of the `CartPole` environment. The left image shows a perfectly balanced pole. In the right image, the pole tilts to the left.

Action Space: The `CartPole` environment accepts one action per time step and has a discrete action space of {0, 1}. The action indicates the direction in which we push

[1] https://www.gymlibrary.ml/

the cart by a fixed force, left and right respectively. The actions are described in Table 2.1.

Action	Description
0	Push cart to the left
1	Push cart to the right

TABLE 2.1: **Action space of the environment CartPole.** Description of all possible actions for the `CartPole` environment.

Observation Space: For `CartPole`, we receive four observations informing us about the position and velocity of the cart and the pole. Table 2.2 shows all observations with the range of the corresponding values.

Observation	Min	Max
Cart Position	-4.8	4.8
Cart Velocity	-Inf	Inf
Pole Angle	~ -0.418 rad (-24°)	~ 0.418 rad (24°)
Pole Angular Velocity	-Inf	Inf

TABLE 2.2: **Observation space of the environment CartPole.** Description of all observations for the `CartPole` environment.

Reward: The goal of this task is to maintain the pole upright. Thus, we get a positive reward of +1 for each step that does not meet the termination criteria. An episode terminates when the pole angle is $\pm 12°$, the cart position is ± 2.4 (hits the end of the rail), or the episode length is higher than 200 (500 for `v1`).

2.1.2 Acrobot

The environment `Acrobot` is based on the paper *Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding* (Sutton, 1995) and the book *Reinforcement Learning: An Introduction* (Montague, 1999). The system contains a chain with two links, with one end of the chain fixed. The goal is to swing the free end of the chain upwards until reaching a certain height. Fig 2.2 shows in the left image one possible state for this problem with the fixed height indicated by a horizontal line. The right image of the figure illustrates one possible state that reached the target height.

Action Space: For `Acrobot`, the action space is discrete and deterministic, representing the torque applied to the joint between the two links, including the loose end. Table 2.3 shows a description of all possible actions.

Action	Description	Unit
0	apply -1 torque to the actuated joint	torque (N m)
1	apply 0 torque to the actuated joint	torque (N m)
2	apply 1 torque to the actuated joint	torque (N m)

TABLE 2.3: **Action space of the environment Acrobot.** Description of all possible actions for the `Acrobot` environment.

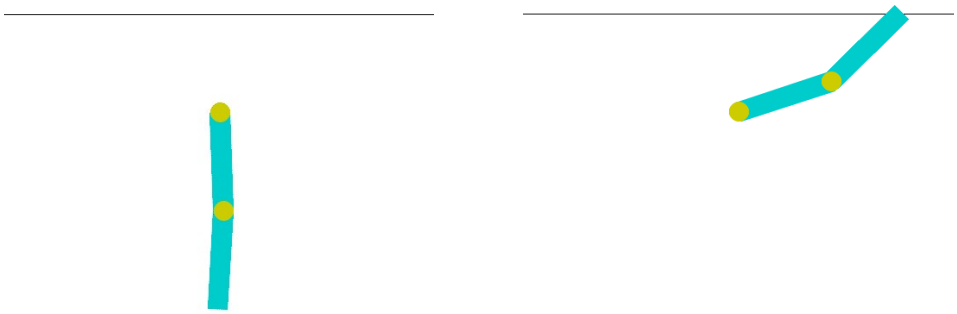


FIGURE 2.2: **Illustration of the environment Acrobot.** The image on the left shows one possible state of the environment `Acrobot`. The image on the right shows one possibility of reaching the target height.

Observation Space: `Acrobot` returns six observations that inform us about the two rotational joint angles and their angular velocities. Table 2.4 contains a description for each observation. Here, `theta1` is the angle of the first joint, where an angle of 0 indicates that the first link is pointing directly downwards; `theta2` is relative to the angle of the first link. An angle of 0 corresponds to having the same angle between the two links.

Observation	Min	Max
Cosine of `theta1`	-1	1
Sine of `theta1`	-1	1
Cosine of `theta2`	-1	1
Sine of `theta2`	-1	1
Angular velocity of `theta1`	~ -12.567 (-4 * π)	~ 12.567 (4 * π)
Angular velocity of `theta2`	~ -28.274 (-9 * π)	~ 28.274 (9 * π)

TABLE 2.4: **Observation space of the environment Acrobot.** Description of all observations for the `Acrobot` environment.

Reward: The goal of this task is to reach the target height with as few steps as possible. Thus, each step that does not attain the goal receives a negative reward of -1. Reaching the target height results in a reward of 0. An episode ends after 500 time steps. `Acrobot` is considered an unsolved environment, meaning there is no reward threshold at which we would consider it solved.

2.1.3 MountainCar and MountainCarContiuous

The environments `MountainCar` and `MountainCarContiuous` are two different versions of the same problem. Both environments share the same observation space. The difference between the two is that `MountainCar` has a discrete action space, and `MountainCarContinuous` has a continuous action space. The goal of the task is to maneuver a car up a mountain to its target. However, the car's engine is not strong enough to drive up the slope in one push. To achieve the goal, we need to accelerate the car strategically. An agent should learn to swing from left to right to gain momentum and eventually reach the target. This task is rather difficult to learn since

we counterintuitively need to move in the opposite direction of the target to swing up. The problem first appeared in Andrew Moore's PhD thesis *Efficient memory-based learning for robot control* (Moore, 1990). Fig 2.3 shows one possible state in the left image and the target state of the two environments in the right image.

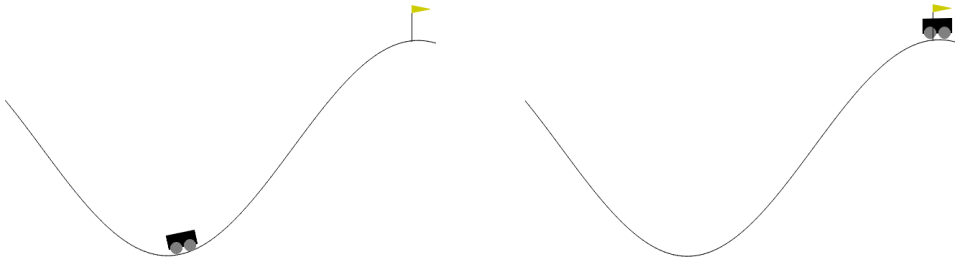


FIGURE 2.3: **Illustration of the environments MountainCar and MountainCarContinuous.** The image shows one possible state of the environments `MountainCar` and `MountainCarContiuous` on the left side. The image on the right shows the target state of the two environments.

Action Space: The action space for the environment `MountainCar` consists of three discrete actions. We can accelerate the car to the right or the left, or we do not interact with it. The actions are further described in Table 2.5. On the other hand, the

Action	Description
0	Accelerate to the left
1	Don't accelerate
2	Accelerate to the right

TABLE 2.5: **Action space of the environment MountainCar.** Description of all possible actions for the `MountainCar` environment.

environment `MountainCarContiuous` accepts one continuous action per step. The action represents the directional force on the car and is clipped in the range of [-1, 1] and multiplied by a power of 0.0015.

Observation Space: The environments `MountainCar` and `MountainCarContiuous` share the same observation space. They return two observations with information about the car's position and velocity. The observations are further described in Table 2.6.

Observation	Min	Max
Position of the car along the x-axis	-1.2	0.6
Velocity of the car	-0.07	0.07

TABLE 2.6: **Observation space of the environment MountainCar.** Description of all observations for the `MountainCar` and `MountainCarContiuous` environments.

Reward: To solve the task, the car has to reach the flag on the right hill as fast as possible. Each time step in which the agent cannot get to the target, it receives

a negative reward of -1 for the environment `MountainCar`. The task is considered solved when getting an average reward of at least -110.0 over 100 consecutive trials. For the environment `MountainCarContiuous`, for each step in which the car does not reach the target, we receive a negative reward of

$$r = -0.1 * a^2,$$

where r is the reward and a is the action. Like that, actions of large magnitude are penalized. When reaching the goal, we receive a positive reward of +100. The environment `MountainCarContiuous` is considered solved when getting an average reward of 90.0 over 100 consecutive trials.

2.1.4 Pendulum

For the environment `Pendulum`, the system consists of a pendulum attached at one end to a fixed point and the other end being free. The goal is to apply torque on the fulcrum to swing it to an upright position. This task is known as the inverted pendulum swing-up problem and is based on the classic problem in control theory. Figure 2.4 shows one possible state of the environment in the left image and the target state in the right one.

FIGURE 2.4: **Illustration of the environment Pendulum.** The left image shows one possible state of the `Pendulum` environment. In the right image, the pendulum is in the upright position, which denotes the target state.

Action Space: The environment accepts one action per step. The action represents the torque applied to the free end of the pendulum.

Action	Description	Min	Max
0	Torque	-2.0	2.0

TABLE 2.7: **Action space of the environment Pendulum.** Description of all possible actions for the `Pendulum` environment.

Observation Space: The environment returns three observations. The first two observations represent the x- and y-coordinate of the pendulum's free end in a cartesian coordinate system. The variable `theta` defines the angle in radians. The third observation is the angular velocity of the free end of the pendulum.

Observation	Min	Max
x = cos(`theta`)	-1.0	1.0
y = sin(`theta`)	-1.0	1.0
Angular Velocity	-8.0	8.0

TABLE 2.8: **Observation space of the environment Pendulum.** Description of all observations for the `Pendulum` environment.

Reward: The reward function is defined as $r = -(theta^2 + 0.1 * theta\_dt^2 + 0.001 * torque^2)$. Regardless of the received reward, an episode ends after 200 time steps. `Pendulum` is an unsolved environment meaning that it has no reward threshold at which we consider it solved.

2.2 Analysis and Visualization

This thesis aims to deliver a proper analysis of different kinds of models in reinforcement learning. For this, we need appropriate methods and visualizations guaranteeing objectivity in evaluation.

2.2.1 Reproducing Literature Results

Oller, Glasmachers, and Cuccu, 2020 deliver an excellent foundation for analysis in their paper *Analyzing Reinforcement Learning Benchmarks with Random Weight Guessing*. The authors analyzed and visualized the complexity of standard reinforcement learning benchmarks based on score distribution.

They tested their approach on the five classic control benchmark problems from the OpenAI Gym interface discussed in Section 2.1. The authors conducted a fixed series of experiments with each environment. They used three neural network architectures ($N_{architectures} = 3$):

- a network without any hidden layers (0 HL, 0 HU),
- a network with a single hidden layer of 4 units (1 HL, 4 HU),
- a network with two hidden layers of 4 units each (2 HL, 4 HU).

With these, they cover a variety of network models that are suited to solve the given tasks.

For the analysis, the authors did not train the network models. Instead, they chose the network weights from the standard normal distribution $\mathcal{N}(0, 1)$ with RWG. This approach assures randomness and no directed learning. They initialized 10^4 samples ($N_{samples} = 10^4$) with different random weights. The authors chose such a large number of samples to draw meaningful statistical conclusions from their experiments. Each sample represents a neural network controller that maps observations to actions in the environment. The authors tested the controllers for each environment during 20 independent episodes ($N_{episodes} = 20$). They saved the score

Algorithm 3 Evaluation process taken from Oller, Glasmachers, and Cuccu, 2020

```
Initialize environment
Create array S of size N_architectures × N_samples × N_episodes
for n = 1, 2, ..., N_samples do
    Sample NN weights randomly from 𝒩(0, 1)
    for e = 1, 2, ..., N_episodes do
        Reset the environment
        Run episode with NN
        Store episode reward in S_{a,n,e}
```

in the score tensor S for each episode. Algorithm 3 illustrates the procedure with pseudocode.

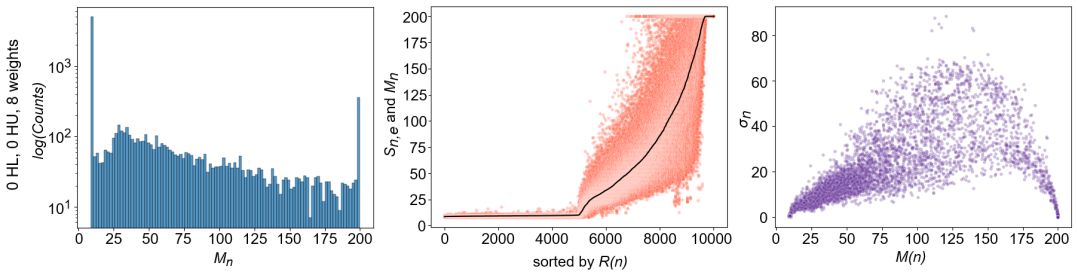
The authors calculated the mean performance over all episodes from a sample and its variance for the analysis. These statistics can reveal how successful and stable the network models are in completing a task. A low mean value suggests that the controller struggles to complete the task. The variance gives us further insight into the score distribution. It illustrates how spread out the scores are from their respective mean score. A high value means that we have high variability. A controller is valuable if it can solve a specific task reliably and stable: a high value for the mean and a low value for the variance.

However, initializing a network with random weights should not create a stable controller. If this is the case, we can assume that the task to solve was too trivial and is not valuable for evaluation measurements. In the illustrations of the paper, the authors visualized their results with three plots: a log-scale histogram of the mean scores, a scatter plot of the sample scores, and a scatter plot of score variance over the mean score.

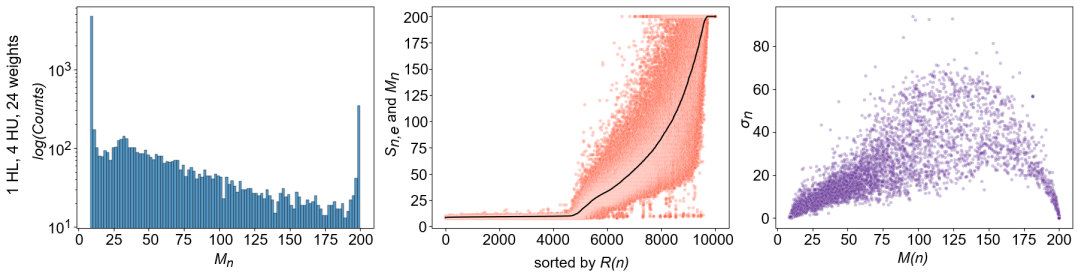
Reproduction of the results: I reproduced the results of the authors following the mentioned methodology. My findings for the environment `CartPole` are displayed in Figure 2.5. The plots illustrate the results for each of the three network architectures. Each row shows the histogram of the mean score values in the first column, the scatter plot of all scores over their rank in the second column, and the scatter plot of the score variance over the mean score in the third column for the corresponding network architecture. There are few differences, but overall all network architectures deliver similar insights.

The histogram plots show that more than 10^3 networks on the logarithmic scale receive a low score. A significant number of networks could still achieve a high mean value or the maximum value of 200. With a score of 200, the network solved the task in each episode. Therefore, the network could reliably solve the environment. Furthermore, in the scatter plots in the second column of Figure 2.5, we can see that the line plot of the mean scores is a continuous increasing line. A suited learning algorithm should generally be able to learn the task incrementally.

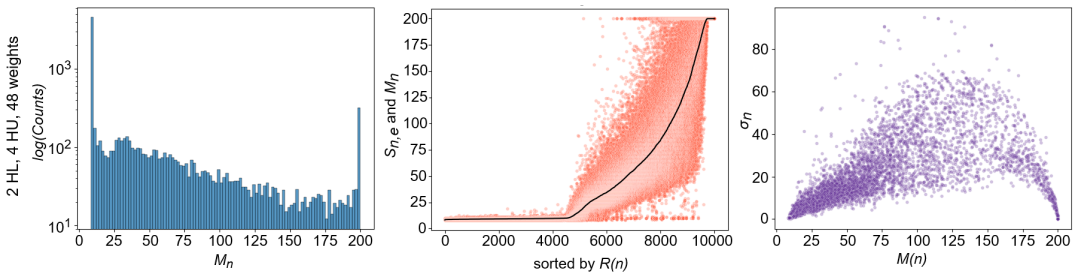
At the top of the scatter plots in the second column of Figure 2.5, we can see quite a few data points with a score of 200 that have a relatively low mean score. This behavior indicates that a network that generally performs poorly can still solve the task with the right initialization conditions. Lastly, in the scatter plots in the last column in Figure 2.5, we can see the distribution of the variance according to the mean value. On the left inside the scatter plots, we have low scores of variance corresponding with a low mean value. These networks were consistently unable to



(A) Results of the network architecture without hidden layers



(B) Results of the network architecture with one hidden layer



(C) Results of the network architecture with two hidden layers

FIGURE 2.5: **Results of the benchmark evaluation.** The plots show a log-scale histogram of the mean scores (left images), a scatter plot of the sample scores over their rank (middle images), and a scatter plot of score variance over the mean score (right image). As expected with RWG, most networks could not solve the given task. However, a significant number of samples still achieve a mean score of 200. That suggests that the environment is trivial to solve.

achieve a high score. We can expect most networks to be in this area without any training. However, the data points are spread out in the middle of the plot. For a high variance, the scores of a network differ highly from the mean value. Receiving a high score depends highly on the initialization conditions, meaning networks with this architecture are inconsistent and unstable. On the right inside the scatter plot, we can see that the data points with a high mean value are primarily of low variance. Thus, to achieve a high mean value, the network needs consistency.

Impact of Bias: Unexpectedly, but following the findings in Oller, Glasmachers, and Cuccu, 2020, the usage of the bias had a relatively significant impact on the performance of the network in my experiments. Without bias, the networks seem to achieve better scores overall. All plots in Figure 2.5 illustrate the results without bias. For comparison, Figure 2.6 shows the results of a network with the same configurations as before, but this time including bias having a much lower score. The

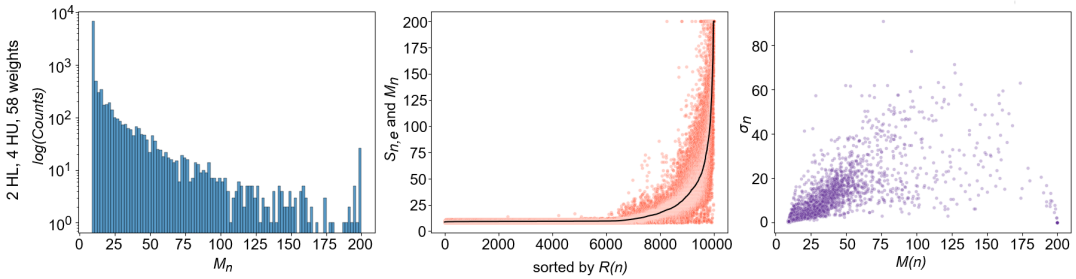


FIGURE 2.6: **Impact of bias.** The figures show the performance of a network with two hidden layers with the same settings as before, but here we include bias. We can see that the network with bias connections performs much worse than those without bias connections.

number of networks that can consistently solve the task also decreases significantly. In the paper, the authors noted that the probability mass of top-performers generally increases when dropping the bias connections for all tested environments. Thus, this is not an isolated observation. However, they did not investigate this behavior further as it was not the focus of their paper.

One possible explanation could be that guessing additional weights might be fatal for achieving a good score. In other words: the more possibilities we have to guess a weight incorrectly, the higher the probability of failing. To test this hypothesis, we can alter the number of weights and compare the results. With an increased number of weights, we would expect the networks to perform worse than before. However, it could also be that the number of weights is not as impactful as the complexity of the model. Randomly guessing the parameters of a simple model has a higher chance of resulting in a good (simple) model than guessing the parameters of a more complex model.

The complexity of the network architecture gives an upper bound for the function that we can approximate. A network with high complexity maps into a larger search space with more complex functions. Since the size of the search space increases, there are also more possible samples that fail to solve the task. The paper showed that a simple model is sufficient to solve these environments. A complex model is oversized for our purpose here. Thus, randomly guessing a simple model can yield a model with good performance with enough attempts.

It is unlikely to randomly guess a complex model that performs well without any training involved. To test this hypothesis, we can increase the complexity of a network by varying the number of hidden layers or the number of neurons in a hidden layer. With increased complexity, we would expect the networks to perform worse than before.

Another interesting aspect would be to inspect the role of the bias in connection with the environments. A simple model is sufficient to solve a simple task. Nevertheless, what if the environment is more difficult to solve? In that case, the undersized complexity would limit us from finding an appropriate solution as the search space is not large enough for this scenario. Therefore, including bias should improve the results. That shows the importance of neural architecture search, a research field aiming to automate neural networks' architectural design process (Kyriakides and Margaritis, 2020).

Next to the analysis of the models, Chapter 3 also contains the experiments following this thought process.

2.2.2 Visualization

For the visualization of the results, I am also following Oller, Glasmachers, and Cuccu, 2020. Figure 2.5 already presented the type of plots used for the visualization in Chapter 3. However, I only use the scatter plots in the middle image because it transparently illustrates a high amount of information. Figure 2.7 shows an example of such a plot. The different architectures of the model are aligned in the columns,

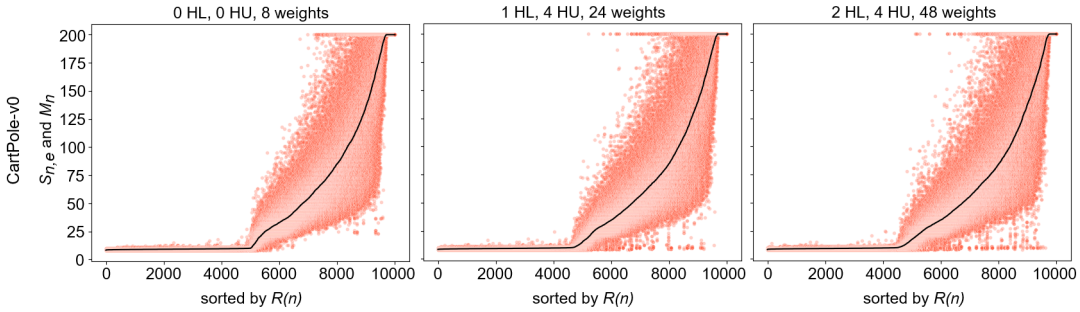


FIGURE 2.7: **Visualization of results.** The image shows an example of the visualization of the conducted experiments. The scatter plot represents the individual sample scores $S_{n,e}$ for episode e over the rank $R(n)$ of the sample n. The overlaid line plot consists of each sample's mean scores M_n.

and the row represents the environment. In this case, I am using three neural network architectures and the `CartPole` environment. In each image, the samples are sorted according to their mean score for better analysis. The position of a sample n within the sorted list is referred to as its rank $R(n)$. In the plot, the samples are aligned according to their rank on the x-axis. Each red dot represents the individual sample score $S_{n,e}$ for sample n in episode e. The overlaid line plot represents the mean score M_n of sample n over all episodes. Thus, the line plot is a naturally increasing curve of the mean scores. The scatter plot indicates the variety of scores for one sample. The title of the plot informs us which model and how many weights were used. In addition, the corresponding OpenAI gym environment is declared.

2.3 Alternative Models

This section describes the models I used in the experiments. Since we are using black-box optimization techniques, we have no constraints on the model. Therefore, we can use any function approximator. Note that we are in the field of direct policy search, where we directly search in the search space of policies for a suited approximation.

2.3.1 Polynomials

Polynomials are equations in the form of a sum of powers in one or more variables multiplied by constant coefficients. Given a field F, a polynomial in variable x with coefficients in F is a formal expression denoted by

$$f(x) = \sum_{i=0}^{n} a_i x^i \in F[x], \qquad a_0, ..., a_n \in F, \;\; i \in \mathbb{N},$$

where $F[x]$ represents the set of all such polynomials (Fischer, 2014, p. 61). The above formula shows the one dimensional case. For the multi-dimensional case, x and the coefficients are vectors instead of scalars. Thus, a polynomial $p(\mathbf{x})$ with $\mathbf{x} = [x_0, ..., x_m]^T$ being a vector and of degree n can be represented by

$$p(\mathbf{x}) = \sum_{i=0}^{n} \mathbf{w_i}^T (x_k^i)_{k \in I} \in F^n[\mathbf{x}], \qquad \mathbf{w_0}, ...\mathbf{w_n} \in F^n, \ I = \{0, ..., m\}.$$

Polynomials are relatively simple mathematical expressions and their derivative and indefinite integral are easy to determine and are also polynomials. Due to their simple structure, polynomials can be valuable to analyze more complex functions using polynomial approximations. *Taylor's theorem* tells us that we can locally approximate any k-times differentiable function by a polynomial of degree k. We call this approximation *Taylor's polynomial*. Furthermore, the *Weierstrass approximation theorem* says that we can uniformly approximate every continuous function defined on a closed interval by a polynomial. Other applications of polynomials are *polynomial interpolation* and *polynomial splines*. Polynomial interpolation describes the problem of constructing a polynomial that passes through all given data points. Polynomial splines are piecewise polynomial functions that can be used for spline interpolation. Spline interpolation is a form of interpolation where splines are used as interpolants.

Construction of the model: For the experiments with the polynomial model in Chapter 3, I constructed the model P_1. It consists of one polynomial mapping from the observaction space to the action space of the reinforcement learning environment for each possible action in a discrete action space. The dimension of the weight vectors is according to the dimension of the input vector. For example, for the environment `CartPole` with the discrete action space $\{0, 1\}$ and observation $\mathbf{x} = [x_0, x_1, x_2, x_3]^T$, this means that P_1 consists of two polynomials:

$$p_0(\mathbf{x}) = \sum_{i=0}^{n} \mathbf{w_i}^T (x_k^i)_{k \in I} \in \mathbb{R}, \qquad \mathbf{w_0}, ..., \mathbf{w_3}, \mathbf{x} \in \mathbb{R}^4, \ I = \{0, 1, 2, 3\}$$

$$p_1(\mathbf{x}) = \sum_{i=0}^{n} \hat{\mathbf{w}}_\mathbf{i}^T (x_k^i)_{k \in I} \in \mathbb{R}, \qquad \hat{\mathbf{w}}_\mathbf{0}, ..., \hat{\mathbf{w}}_\mathbf{3}, \mathbf{x} \in \mathbb{R}^4, \ I = \{0, 1, 2, 3\}$$

In the formulas, n denotes the degree of the polynomial. For the experiments, I used polynomials of degrees one, two, and three. However, the output of the polynomials is unbound, as illustrated in Figure 2.8. That makes it harder to interpret the results as probabilities. So, I scaled the outputs with the logistic sigmoid function. We can interpret the output space of the sigmoid function as a probability, as discussed in Section 1.2.1. In this case, we search for the probability that a specific action is chosen given an observation $\mathbf{x}$. Thus, for our example with the `CartPole` environment, we can interpret $sig(p_0)$ as the probability that action 0 is the correct one and $sig(p_1)$ as the probability that action 1 is the correct one. Putting this thought into a formula for P_1 and the `CartPole` environment, we get:

$$P_1(\mathbf{x}) = \begin{cases} 1 & \text{if } sig(p_1(\mathbf{x})) > sig(p_0(\mathbf{x})) \,, \\ 0 & \text{otherwise} \,. \end{cases}$$

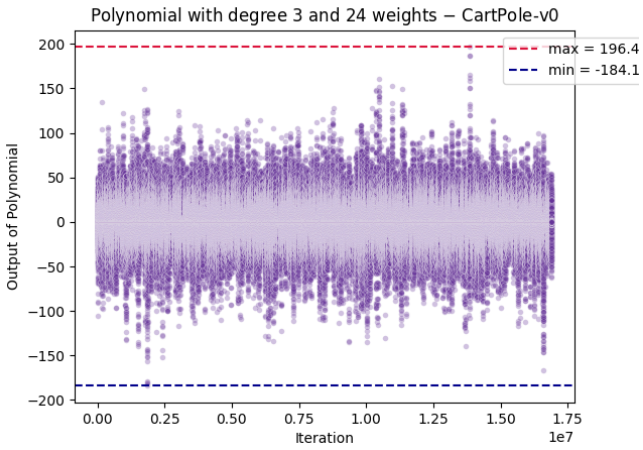


FIGURE 2.8: **Upper and lower bound.** The figure shows each output of the polynomial functions described above. As we can see, the functions are not well bound, and there are quite a few outliers. That makes it hard to interpret the output sensibly.

2.3.2 Binary Trees

A binary tree is a common data type in computer science with a relatively simple design: a hierarchical tree structure consisting of a set of connected nodes. A node is a structure that can hold data and connections to other nodes, also called links or edges. Nodes in a binary tree have exactely two child nodes. A node's child nodes are those connected to it and situated beneath it in the tree. The node is then called the parent node in the perspective of its child nodes. Figure 2.9 illustrates an example of a binary tree with seven nodes. The topmost node is called the root node.

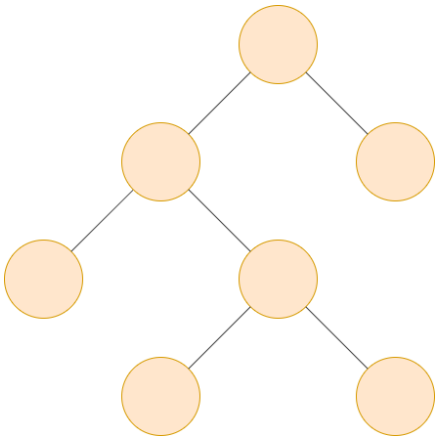


FIGURE 2.9: **Example of a binary tree.** The figure shows an example of a binary tree consisting of seven nodes.

By design, there exists a path from the root node to every other node of the tree. The nodes which do not have child nodes are called leaf nodes.

We can use binary trees as decision trees for machine learning problems. Decision trees represent a predictive model to come to conclusions about a set of observations. For each node, we assess a feature, and a child node is selected accordingly. This process starts at the root node, and we repeat it until reaching a leaf node. The leaf node then holds the value of the target variable. In classification problems, the

target variable takes on a value inside a discrete set of values. The leaves then represent the class labels. The target variable can also take on a constant value inside a specified range for regression problems.

The approach to decision trees is simple and intuitive when comparing them with other algorithms in machine learning. Decision trees split the input space into multiple partitions. When the dataset is splittable, it works well with fast predictions. Tree-based models like C4.5 which is an extension of the earlier ID3 algorithm and CART became quite popular (Quinlan, 2014; Breiman et al., 2017). C4.5 was even ranked one of the top ten algorithms in data mining (Wu et al., 2008). Another advantage of decision trees is their interpretability since they implement rules humans can easily analyze. There were also approaches combining neural networks and decision trees. Yang, Morillo, and Hospedales, 2018 introduced a neural network-based tree model DNDT which performed better for some datasets than neural networks while providing an interpretable decision process. Despite the success of decision trees in supervised learning problems, decision trees are not commonly used in reinforcement learning. Reinforcement learning introduces many challenges, including that the model has to constantly adapt as the interaction between the agent and the environment continues. Silva et al., 2020 approached this challenge by allowing for a gradient update over the entire tree.

Construction of the model: Since this study aims to experiment with models that will be optimized with black-box optimization techniques, we can use a simple binary tree model where each node holds three properties: `values`, `left`, `right`. The variables `left` and `right` contain the two child nodes. If they are empty, the node is a leaf. The variable `values` holds the model's output if the node is a leaf. Otherwise, it contains the weights used for the decision strategy. Figure 2.10 illustrates the decision process of the binary tree model for the environment `CartPole`. The illus-

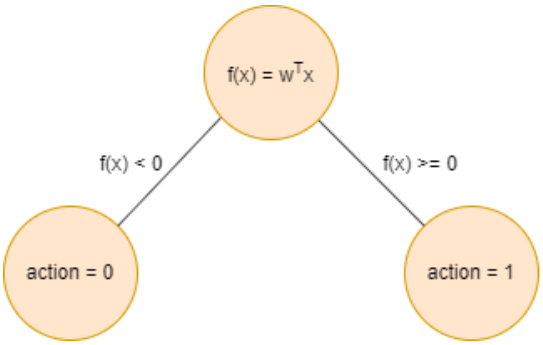


FIGURE 2.10: **Illustration of binary tree model.** The figure illustrates the binary tree model with the environment `CartPole`.

trated model has one node for the decision finding and two leaf nodes containing the two possible discrete actions, 0 and 1. The dot product defines the decision-making process. For the environment `CartPole` with observations $\mathbf{x} = [x_0, x_1, x_2, x_3]^T$, this results in the following formula:

$$f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} = w_0x_0 + w_1x_1 + w_2x_2 + w_3x_3$$

The variable w represents the weights of the model. The actions of the leaves are fixed, whereas an optimization technique optimizes the weights of the nodes involved in the decision-making process.

The implementation is straightforward for environments like `CartPole` where we only have two possible discrete actions. However, for an action space with more than two actions possible, the binary tree still has the same structure, but the decision is fixed between the minimum action and the maximum action. For example, for the environment `Acrobot`, the minimum action is to apply -1 torque to the joint, and the maximum action is to apply 1 torque to the joint. In this case, we discard the action of applying 0 torque to the joint. Similarly, for environments with a continuous action space, we again take the minimum and the maximum of the value of the action. For the environment `Pendulum`, this would be -2.0 and +2.0.

The only hyperparameter we have with this model is the number of nodes for the decision process. That makes hyperparameter tuning much more straightforward than it would be for other models like neural networks. Note that the leaves do not count as nodes for the model's architecture since each path in the tree has to result in a final decision between two actions. The leaf nodes are mandatory in that sense and thus cannot be altered in their number.

While simplistic, this model is easily extendable, for example, by having an equation decide the action in a leaf rather than a hardcoded action. Most importantly, architecture search using standard tree operators should be much more straightforward.

This thesis constitutes an initial feasibility study. So, further extensions are left to future work.

Chapter 3

Experiments

This chapter aims to answer the research questions formulated in Section 1.7. First, I describe which experiments I conducted. Next, I show the results of these experiments. Finally, I discuss my findings further in the last section.

3.1 Reproduction of Previous Results

For a starting point, I reproduced the results of the paper *Analyzing Reinforcement Learning Benchmarks with Random Weight Guessing* (Oller, Glasmachers, and Cuccu, 2020) discussed in Section 2.2. I used the same methodology as the authors of the paper did. In specific, I used three network architectures: a network without any hidden layers (0 HL, 0 HU), a network with a single hidden layer of 4 units (1 HL, 4 HU), and a network with two hidden layers of 4 units each (2 HL, 4 HU). I used the same procedure as explained in pseudocode in Algorithm 3 using RWG. I tested the three neural networks for all five classic control environments provided by the OpenAI Gym interface: `CartPole`, `Acrobot`, `Pendulum`, `MountainCar`, and `MountainCarContinuous`. For the next experiments, the reproduced results serve as a baseline to judge the effectiveness of the alternative models in comparison with neural networks.

3.1.1 Results

Figure 3.1 shows the results for all five classic control environments using neural networks without bias. For each plot, the samples are ranked according to their mean score and aligned on the $x-$axis according to their rank. The scatter plots show all scores of the samples, whereas the lineplot illustrates the mean of each sample over all episodes. Each column shows a different network architecture. The first column shows the results of the networks without hidden layers, the second column the networks with one hidden layer, and the last column the networks with two hidden layers. The rows are dedicated to the different classic control environments.

These results are consistent with the ones from the paper *Analyzing Reinforcement Learning Benchmarks with Random Weight Guessing*. In the subsequent sections, I will refer to these plots to compare the effectiveness of other models or configurations to these neural network architectures.

3.2 Comparison of Alternative Models to Neural Networks

This experiment aims to provide a comparison of alternative models to the commonly used neural network model. I selected a few promising models and analyzed them with the help of the classic control environments. Analogous to the neural

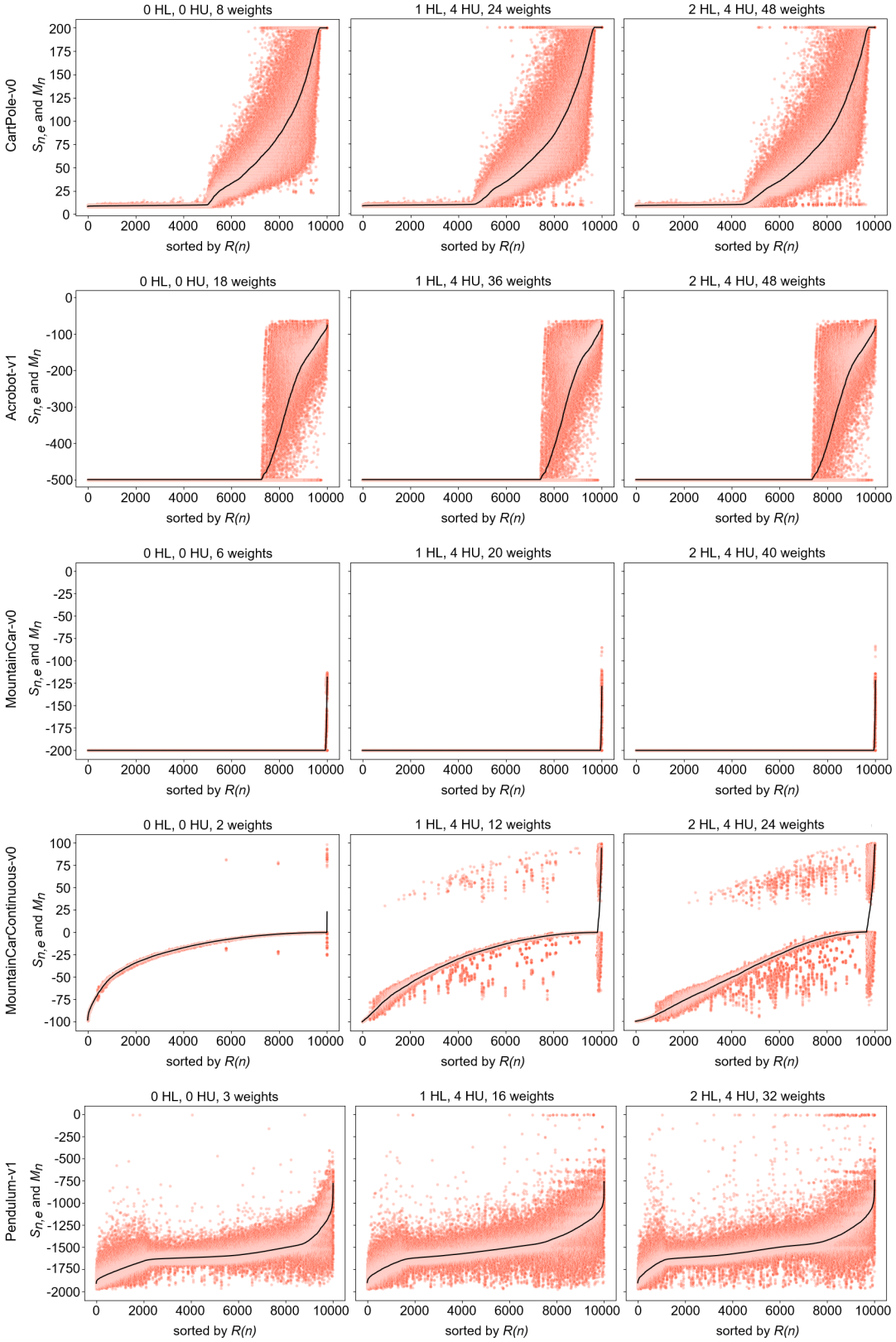


FIGURE 3.1: **Results for all classic control environments using neural networks without bias.** Each row shows the results of one environment, whereas the columns represent the different network architectures. These plots are a reproduction of the results of the RWG-paper. My results are consistent with those shown in the paper.

networks, I used three architectures for each model with increasing complexity. I conducted the following experiments:

(a) In the first experiment, I took the polynomial model P_1 and tested it on the discrete classic control environments. I used polynomials of degrees 1, 2, and 3.

(b) In the second experiment, I tested the binary tree model on all five classic control environments. I used binary trees with 1, 4, and 8 nodes.

I described the models in more detail in Section 2.1. For the experiments, I used the same procedure as in *Analyzing Reinforcement Learning Benchmarks with Random Weight Guessing* with only slight adaption. Thus, we can directly compare the results of the alternative models to those of neural networks. Another important aspect of this procedure is that there is no learning involved. For their paper, the authors were interested in the complexity of the environment, whereas I aim to find out more about the nature of the models. The classic control environments are relatively easy to solve, as explained in Section 2.2. Therefore, we expect some controllers to solve the task even without any learning involved.

For the experiments, first, I initialized the environment. Second, I initialized the respective model. Then, I drew the weights of the model from the standard normal distribution $\mathcal{N}(0,1)$ or the uniform distribution $U(a,b)$ depending on the model and environment used. Each of these instances of the model represents a sample. In total, I used $10'000$ samples ($N_{samples}$). Finally, I ran 20 episodes ($N_{episodes}$) with each sample for an environment and stored the respective score as an entry of the score tensor S. Algorithm 4 shows an overview of the described procedure. As we can see, the procedure is very similar to the one shown in Algorithm 3.

Algorithm 4 Procedure for alternative models using RWG

```
Initialize environment
Initialize model
Create array S of size N_samples × N_episodes
for n = 1, 2, ..., N_samples do
    Sample model weights randomly from 𝒩(0,1) or U(a,b)
    for e = 1, 2, ..., N_episodes do
        Reset the environment
        Run episode with model
        Store accured episode reward in S_{n,e}
```

3.3 Analysis of the Impact of Bias

The authors of the paper *Analyzing Reinforcement Learning Benchmarks with Random Weight Guessing* indicate that using bias worsens the performance of neural networks in their experimental setting. This experiment serves to analyze this behavior. I used the same procedure as explained in the paper with the same three neural network architectures: a network without any hidden layers (0 HL, 0 HU), a network with a single hidden layer of 4 units (1 HL, 4 HU), and a network with two hidden layers of 4 units each (2 HL, 4 HU). First, I reproduced the results for the networks with bias to confirm their findings. Then, I varied the number of (a) weights, (b) hidden layers, and (c) neurons for a neural network. To get insights into whether this observation is specific to neural networks, I additionally used the polynomial model for

the experiment concerning the variation of the weights of the model. In more detail, the experiments I conducted in this matter are the following:

(a) In the first experiment, I only changed the number of weights for each of the three networks and the polynomial model P_1 described in Section 2.3. I tripled the number of weights for all models. To achieve this, I constructed one weight $\mathbf{w_i}$ out of three weights by addition:

$$\text{Triple number of weights:} \qquad \mathbf{w_i} = \mathbf{w_{i1}} + \mathbf{w_{i2}} + \mathbf{w_{i3}}$$

(b) In the second experiment, I tested the neural network models with different numbers of hidden layers. Each layer still has the same number of hidden neurons as before. Thus, the networks have four hidden units for each layer. I analyzed a network with 4 hidden layers, one with 6, and one with 8.

(c) In the last experiment concerning this subject, I varied the number of hidden neurons for a network. Here, I only used a network with two hidden layers. For the number of hidden neurons, I chose 5, 8, and 10.

3.4 Polynomial Model

Figure 3.2 shows the results for the discrete classic control environments using the polynomial model P_1 without bias. The visualization is identical to the one I used for the neural networks in Figure 3.1. The rows show the results for the different environments, and the columns show the polynomial with degrees 1, 2, and 3. Looking at the first column, we can see a striking resemblance to the performance of the neural network without hidden layers shown in Section 3.1 for all three environments. This observation is not entirely a surprise since the neural network without hidden layers represents a linear controller. However, the second and last columns show different results for the environments `CartPole` and `Acrobot`.

CartPole: For the environment `CartPole`, we can see that the curve of the mean score stays low until around 5′000 but then goes up relatively steeply. That means the linear model fails around 50% to guess the action correctly. However, after that, we have a high probability to achieve a good score or even solve the task entirely during multiple episodes. There are also quite a few samples that could solve the task each time, as indicated by the short straight black line at the top of the plot. Looking at the polynomials with degree 2 for `CartPole`, we can see that the scores increase already at around 2′000, but the slope is less steep than for the linear model. Therefore, we start earlier to get meaningful results, but the scores increase only slowly. This aspect is significant for a learning algorithm. In addition, there are fewer samples that could solve the task for each episode than there are for the linear model. Furthermore, the variance is higher for the polynomials with a higher degree compared to the linear model. If we look at the results of the polynomials with degree 3, we can see that there is only minor improvement compared to the polynomials of degree 2. The scores are overall slightly higher, but the slope is similar to before. The considerable difference lies between the polynomials with degree 1 and polynomials with degree 2 and degree 3.

Acrobot: For the environment `Acrobot`, the polynomial model with degree one delivers a score distribution indistinguishable from the one of the neural network

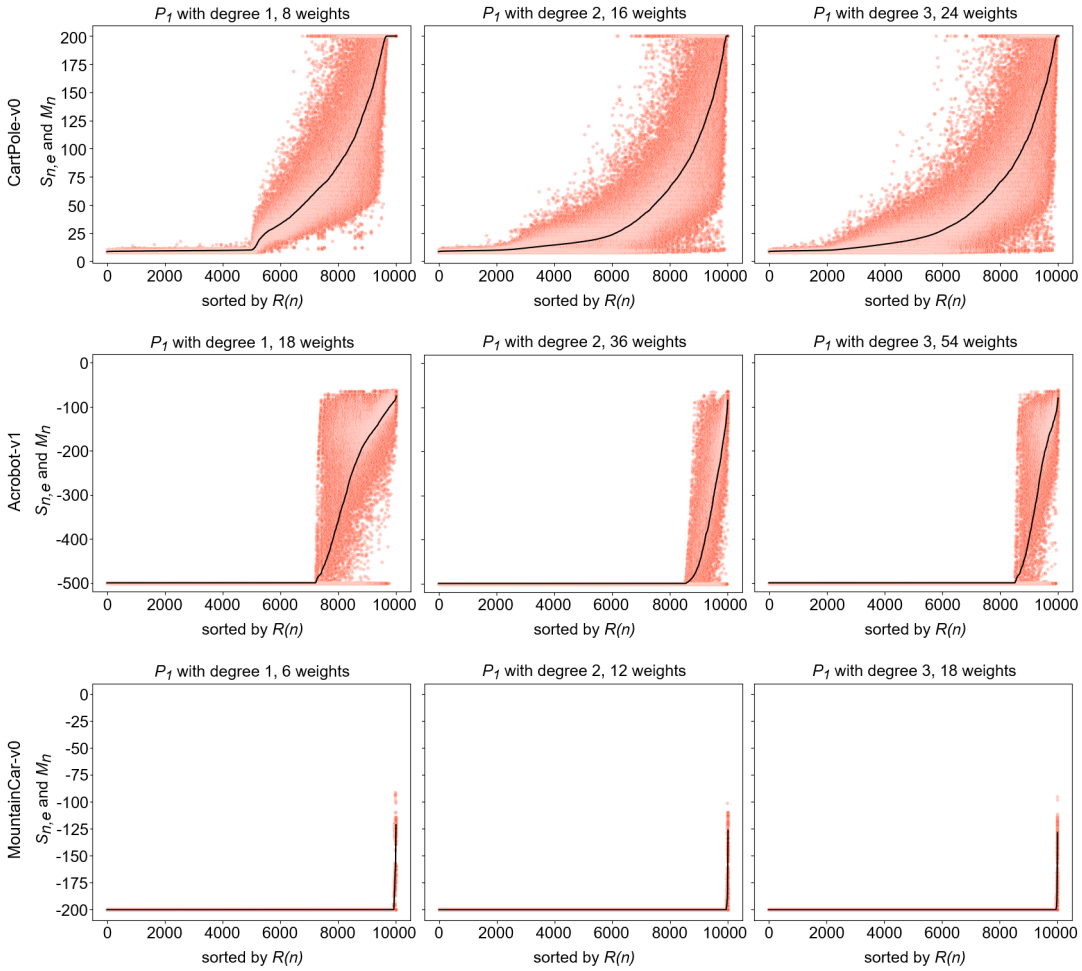


FIGURE 3.2: **Results for the discrete classic control environments using the polynomial model without bias.** Each row shows the results of one environment. The columns represent the different degrees of the polynomials. In these plots, the model P_1 was used. The results aligned in the first column are almost indistinguishable from the ones of the neural networks without hidden layers. Nonetheless, the environments `CartPole` and `Acrobot` deliver different results for a higher degree of the polynomial.

without hidden layers. The lineplot stays at -500 until around rank 7'500. Then it goes up steeply. Thus, we have a large plateau of minimum scores before we reach scores over -500. That opposes a higher difficulty level for the learning algorithm than `CartPole`. But with RWG, we can still reach relatively high scores, and the lineplot is continuous. Interestingly, the polynomial models with a higher degree performed noticeably worse. We cannot observe this behavior with neural networks.

MountainCar: The environment `MountainCar` is more challenging to solve than the other environments, as we can see by the results of both the neural networks and the polynomial model. Both models show very similar findings. There is a large fitness plateau, and only a few samples can reach a score higher than the minimum score of -200. That applies to both the linear models and the ones with higher complexity.

Bias: Figure 3.3 shows the results of using the polynomial model P_1 to solve the discrete classic control environments using bias. Comparing these results with those

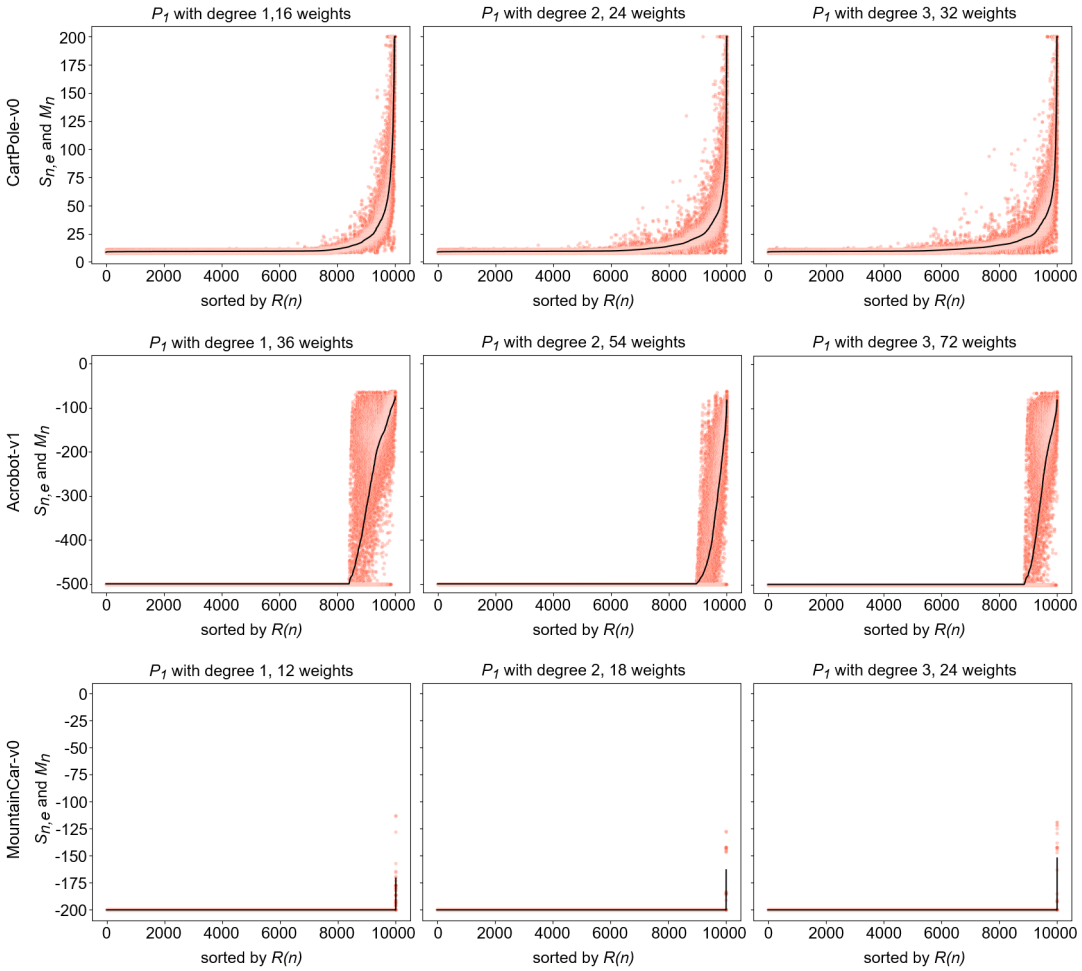


FIGURE 3.3: **Results for the discrete classic control environments using the polynomial model with bias.** Each row shows the results of one environment. The columns represent the different degrees of the polynomials. For these plots, the polynomial model P_1 was used. Overall, the model performs noticeably worse with bias as opposed to without bias.

without using bias in Figure 3.2 shows a significant difference. The bias influences the performance of the model negatively. The individual and mean scores are visibly lower for all three environments for all three configurations of the model. Thus, the observation about the bias the authors made in the paper *Analyzing Reinforcement Learning Benchmarks with Random Weight Guessing* is not specific to neural networks but seems to be more of a general factor. Understanding the reasons behind it could be the focus of a furture study.

3.5 Binary Tree Model

For the binary trees, I tested three models with 1, 4, and 8 nodes. Since the previous experiments suggest that using bias has a negative effect, the following experiments do not include bias. Figure 3.4 shows the results for all five classic control environments with the binary tree model.

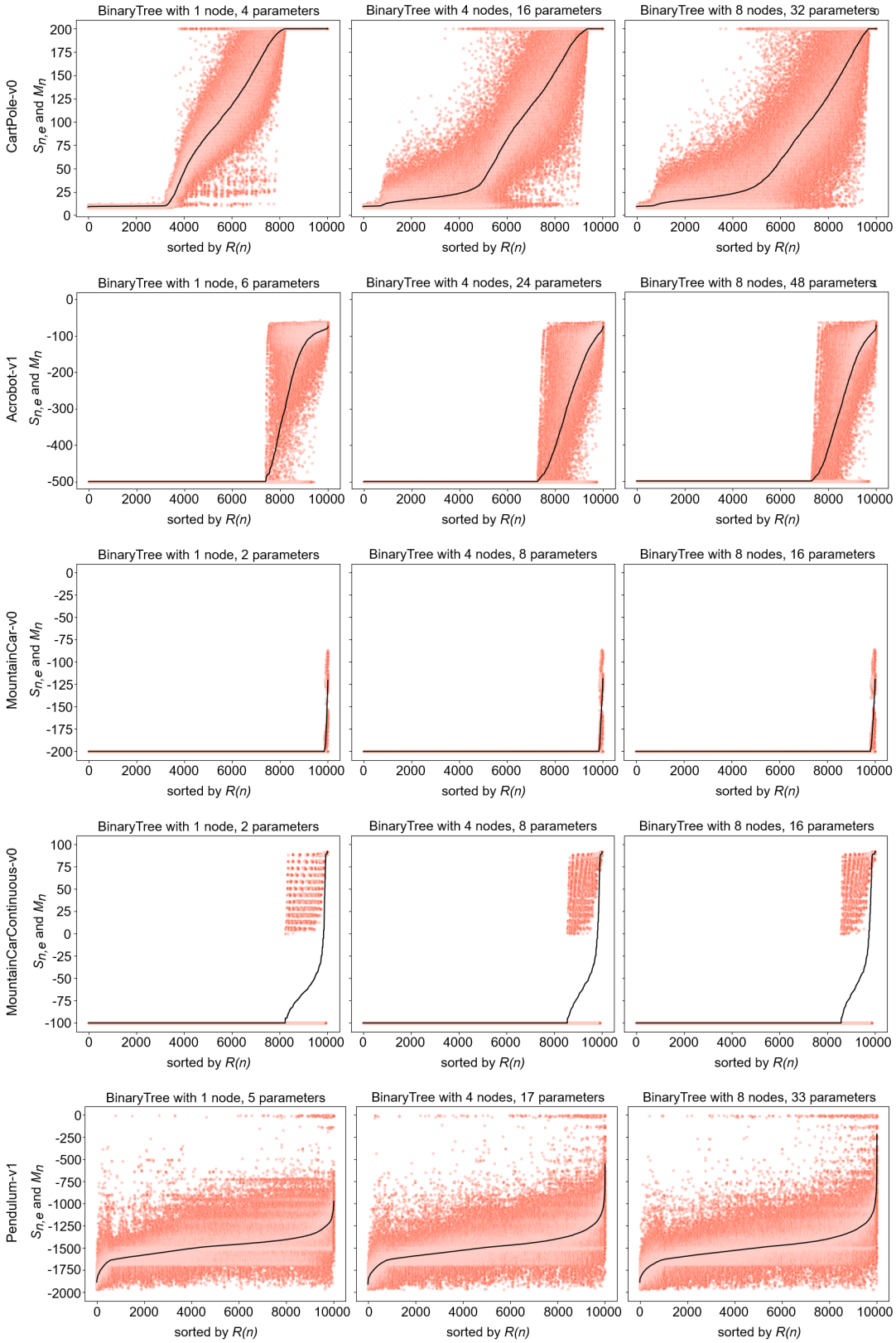


FIGURE 3.4: **Results for the classic control environments using binary trees without bias.** Each row shows the results of one environment. The columns represent the different configurations for the binary tree models. Even with only one node, the results are comparable to those from neural networks or outperform them.

CartPole: Comparing the results using binary trees with those using neural networks shown in Figure 3.1, we can see a few differences. For the environment `CartPole`, we can see that we generally reach better scores with the binary trees. Comparing the neural network without hidden layers with the binary tree with only one node, we have a similar curve, but the lineplot representing the mean scores sets off much earlier at around 3'500 for the binary tree than the one for the neural network, which sets off at around 5'000. In addition, the binary tree has a higher chance of reaching a maximal mean score, indicated by the straight black line at the top right. For the more complex architectures, it gets more interesting.

The binary tree with four nodes can already raise the mean value under 1'000 samples. That tells us that the problem of the fitness plateau is reduced significantly. Thus, when using a learning algorithm, the binary tree has a better chance of solving the problem without getting stuck in a fitness plateau. The chance of reaching a maximal mean score decreases with adding more nodes to the binary tree. That could be caused by the increasing complexity of the binary tree, which makes it harder to guess the weights correctly. Furthermore, the scores are more spread out for the binary trees than for the neural networks, indicating a higher variance.

Acrobot: Comparing the results of the binary tree model with one node with the neural network without hidden layers for the environment `Acrobot`, the binary tree produces a better score distribution. Although the fitness plateau remains to the same extent, the individual scores are overall higher, and the slope of the mean scores has improved, demonstrating a steeper slope. The neural networks with higher complexity seem to slightly outperform the binary trees with four and eight nodes. The scores are slightly higher, and the slope of the mean scores is a bit steeper. However, the difference is not massive. Overall, we can say that both models reach similar scores and mean values.

MountainCar: Comparing the two models, binary trees and neural networks, for the environment `MountainCar`, we can see that they reach similar mean scores. However, the binary trees are more likely to reach more and higher individual scores. The risk of being stuck in a fitness plateau remains for the binary trees. In addition, the scores are spread wider for the binary tree than for the neural network.

MountainCarContinuous: The results for the environment `MountainCarContinuous` look very different for the binary trees than for the neural networks. Neural networks show a slowly increasing curve with a peak after 9'500-9'900 samples for the mean scores. The binary trees have a lot of low performers displayed as a plateau of minimum scores. We cannot see this bad behavior in the results of the neural network models. However, there are a few samples that can reach a positive score. The neural network without hidden layers could not achieve these high mean scores.

This behavior is likely caused by the binary tree outputting discrete values instead of continuous ones, as it is the case for neural networks. The actions are fixed with the current implementation of the binary tree model. The two possible actions are the minimum and maximum of the action defined by the environment. In the case of `MountainCarContiuous`, this results in either -1 or 1. However, experimenting with this action pair shows us some interesting results for the environment `MountainCarContiuous`. Fiugure 3.5 shows the results of using the action pair 0 and 1 instead of -1 and 1. There is a significant improvement in the model's performance.

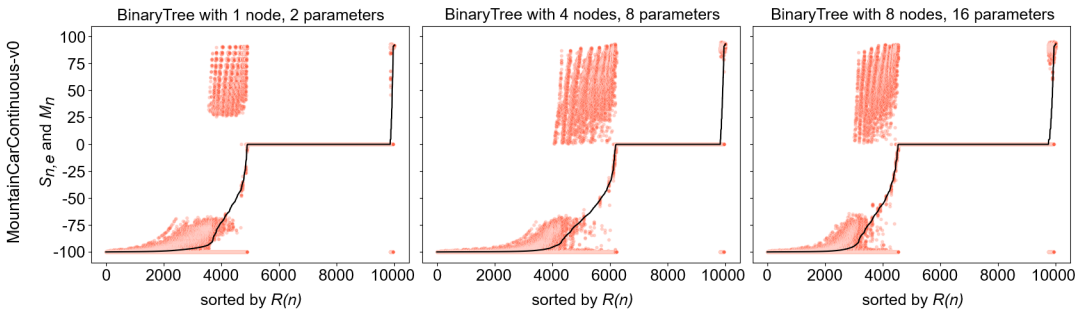


FIGURE 3.5: **MountainCarContiuous using the binary tree model with actions 0 and 1.** The figures show the results of the environment `MountainCarContiuous` using binary trees with fixed actions 0 and 1 as opposed to -1 and 1. The model's performance increased significantly compared to the results in Figure 3.4.

There are still some low performers, but the score distribution improved noticeably. However, many samples produce a zero score, resulting in a large fitness plateau.

Pendulum: For the `Pendulum` environment, we can see that the binary trees perform increasingly better with more nodes. The binary tree with one node reaches a mean score of around -1'500 in the large middle area. These scores are slightly higher than those of the neural network without hidden layers in the middle area. However, the neural network reached a higher maximum mean score than the binary tree. With an increased number of nodes, the binary tree model even outperforms the neural networks with remarkably higher mean scores. Even though the actions produced by the binary trees are discrete instead of continuous, the curve of the mean scores looks very similar to the one of neural networks.

3.6 Bias Investigation

In their paper, Oller, Glasmachers, and Cuccu (2020) mentioned that using bias negatively affects the score distribution leading to fewer top performers and generally lower scores. Figure 3.6 shows my results for all classic control environments with neural networks without using bias. Comparing these plots with the ones without using bias in Figure 3.1, we can see the negative impact the bias has on the effectiveness of the model. For the environment `CartPole`, the scores are overall lower and get slighly worse with the increased complexity of the model. Interestingly, for the environment `Acrobot`, the network without hidden layers is resistent to the negative impact of the bias. However, the network with one hidden layer results in lower scores with bias. The network with two hidden layers with has slightly better results than the one with one hidden layer but still performs worse than the one without bias. The same observation can be made for the environment `MountainCar`. For the environment `MountainCarContinuous`, we can see a negative effect on the score distribution for all three network architectures. The environment `Pendulum` does not seem to suffer much when introducing bias. For all three architectures, there is not much difference visible. The scores are generally lower and produce less top performers when using bias. Therefore, we can confirm the interesting observation of the authors, namely, that using bias is counterproductive in this experimental setting.

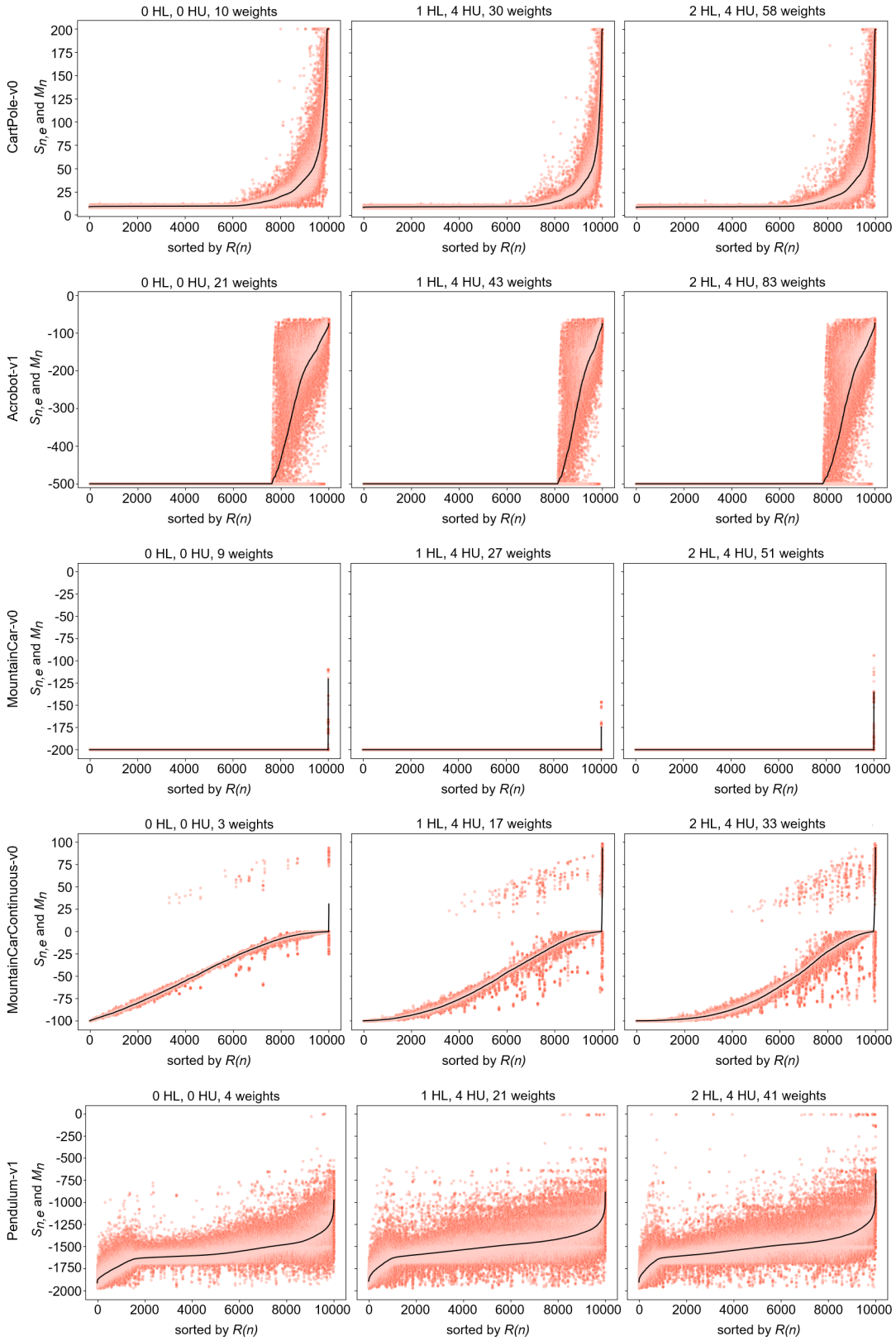


FIGURE 3.6: **Results for all classic control environments using neural networks with bias.** Each row shows the results of one environment, whereas the columns represent the different network architectures. For most environments, we can see that introducing a bias has a negative effect on the score distribution.

Altering the number of weights: To further investigate this counterintuitive behavior, Figuer 3.8 shows the results of the number of weights for the polynomial model, and Figure 3.8 shows the results of tripling the number of weights for each network architecture. Comparing the results of the polynomial models, we cannot

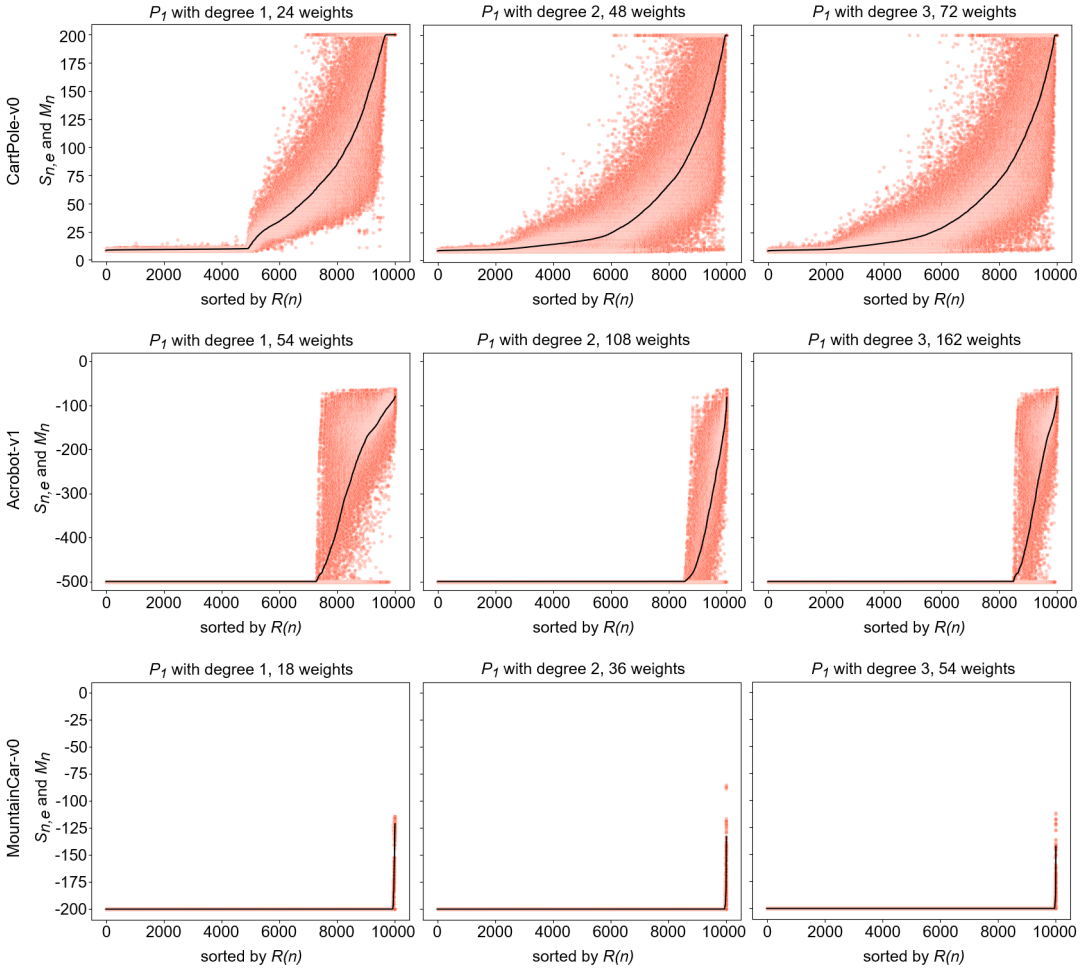


FIGURE 3.7: **Results of tripling the number of weights for the polynomial model.** The figure shows the results of tripling the number of weights for the polynomial model for the three discrete classic control environments. Despite the increased number of weights, there are only slight differences visible in the plots of the environment `MountainCar` compared to the ones shown in Figure 3.2. `Acrobot` and `CartPole` look indistinguishable from the other results.

see a difference for the environments `CartPole` and `Acrobot` comparing these results with those shown in Figure 3.2. The environment `MountainCar` shows only slight differences.

Comparing the plots in Figure 3.8 to those in Figure 3.1, we cannot see a huge difference. The score distribution of the environments `CartPole` and `Acrobot` look indistinguishable from those presented in Figure 3.1. We can see slight differences in the plots for the environment `MountainCar`. Increasing the number of weights for the network without hidden layers results in a few better individual scores without much difference in the mean score distribution. We see slightly better mean scores for the network with one hidden layer, whereas the network with two exhibits slightly worse mean scores. For the environment `MountainCarContinuous`, the mean scores are lower compared to the networks with more weights. However,

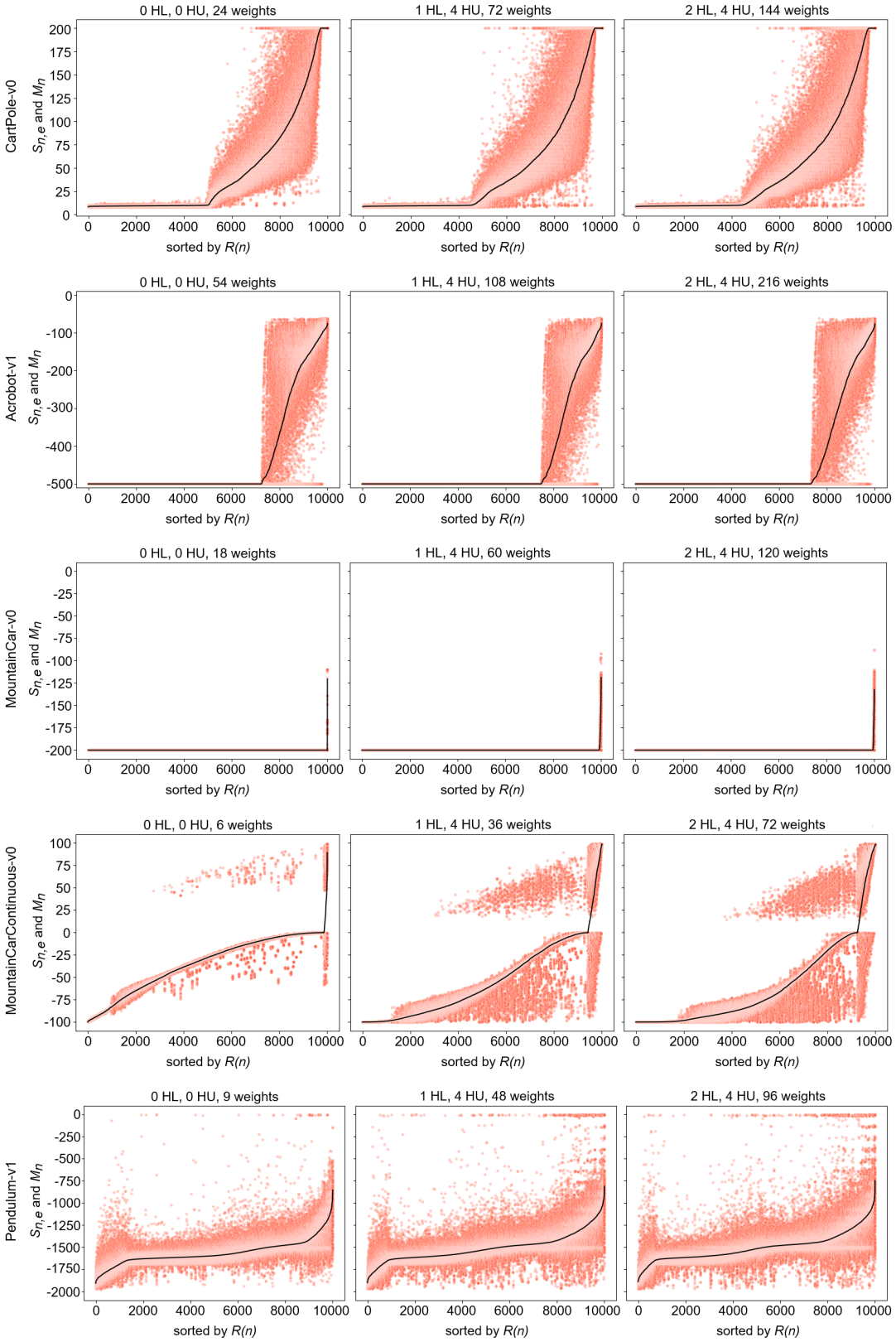


FIGURE 3.8: **Results of tripling the number of weights for neural networks.** The figure shows the results of tripling the number of weights for the three neural network architectures for the classic control environments. Despite the increased number of weights, there are only slight differences visible in the plots compared to the ones shown in Figure 3.1 for all environments with the exception of `MountainCarContinuous`.

there are more top performers. This is especially visible for the network without hidden layers.

In addition, the scores are more spread out. There are a few more samples that reach a higher score for the environment `Pendulum` with the network without hidden layers. However, the networks with one and two hidden layers deliver similar results to those shown in Figure 3.1 for the environment `Pendulum`. To summarize, altering the number of weights for a network can produce a slight difference in the results for some environments. But these differences are not as dramatic as we have seen for the networks without using bias.

Altering the number of layers: Figure 3.9 shows the results of altering the number of layers in a network. Each layer still has the same amount of neurons, namely four neurons. For the environment `CartPole`, the lineplot of the mean scores seems to start earlier to rise above the minimum score, but the slope is also slightly less steep with an increasing number of hidden layers. The environment `Acrobot` does not seem to be affected anyhow by the change in the number of layers. The plots look indistinguishable. The mean scores for the environment `MountainCar` seem to be best for the network with four hidden layers. The scores tend to get lower with an increased number of layers. For the environment `MountainCarContinuous`, the slope of the mean scores changes. The scores get generally slightly lower with more layers. Finally, we cannot see a big difference in the plots for the environment `Pendulum` by adding more layers. There may be a few more top performers with more hidden layers, but the difference is not significant and is not reflected in the mean score distribution.

Altering the number of neurons: Figure 3.10 shows the results of increasing the number of neurons in each hidden layer for a network with two hidden layers. We cannot see much difference in the plots for the environments `CartPole` and `Acrobot`. For both, the score distribution seems to be unaffected by the increased number of neurons. The mean scores for the environment `MountainCar` first decrease with five and eight neurons. Then with ten neurons, we achieve higher scores again. For the environment `MountainCarContinuous`, the plots look similar to the ones in Figure 3.9 with an increased number of layers. We can see a slight decrease in the scores with increased neurons which is less extreme than when we increased the number of layers. With the environment `Pendulum`, we can reach higher mean scores with an increased number of neurons. Otherwise, the slope of the mean scores and the overall score distribution looks very similar.

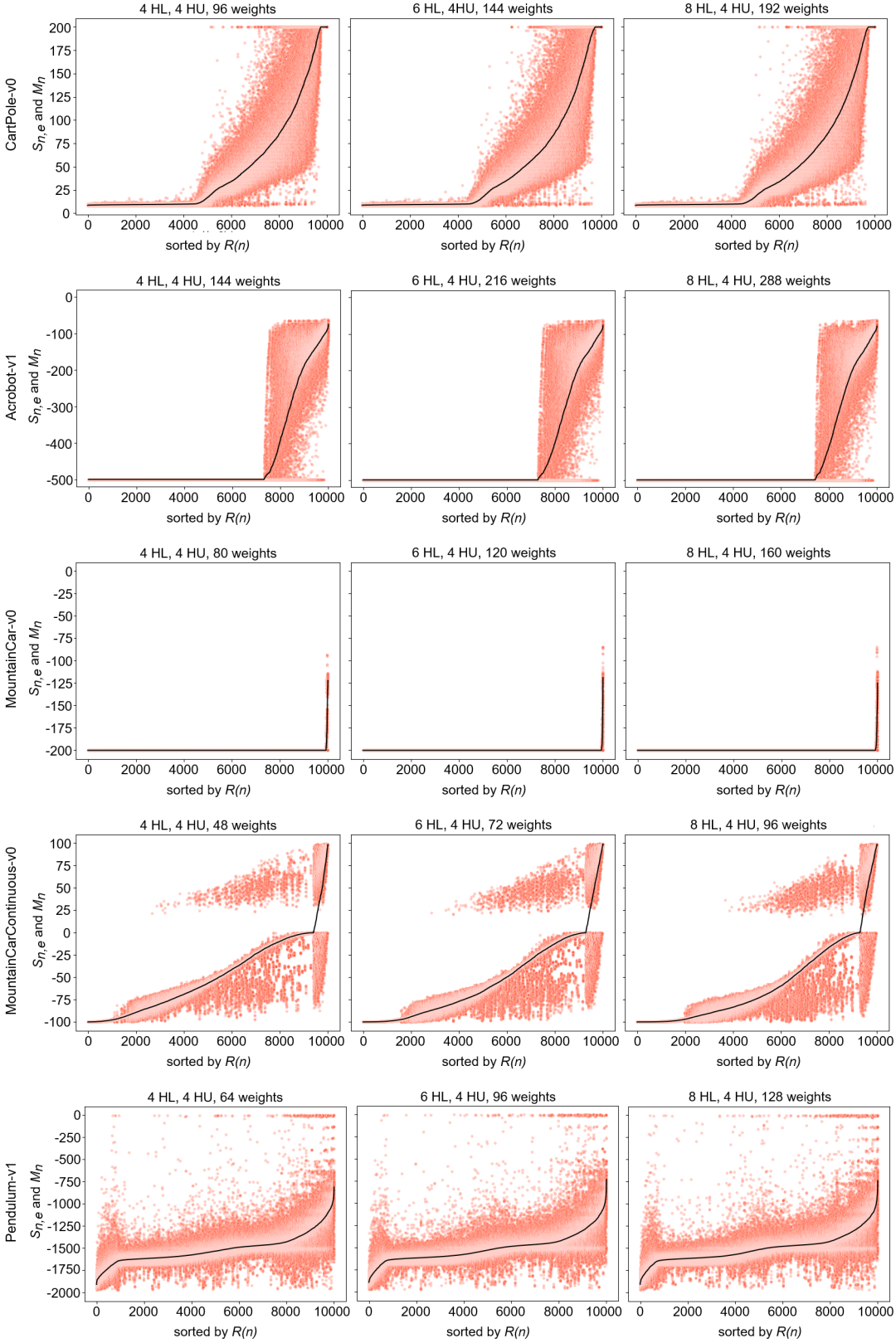


FIGURE 3.9: **Results of altering the number of layers for neural networks.** The plots show the results of increasing the number of hidden layers in a neural network gradually. The score distribution changes with an increased number of layers for some environments. However, the changes are less extreme compared to the results of the networks that used bias.

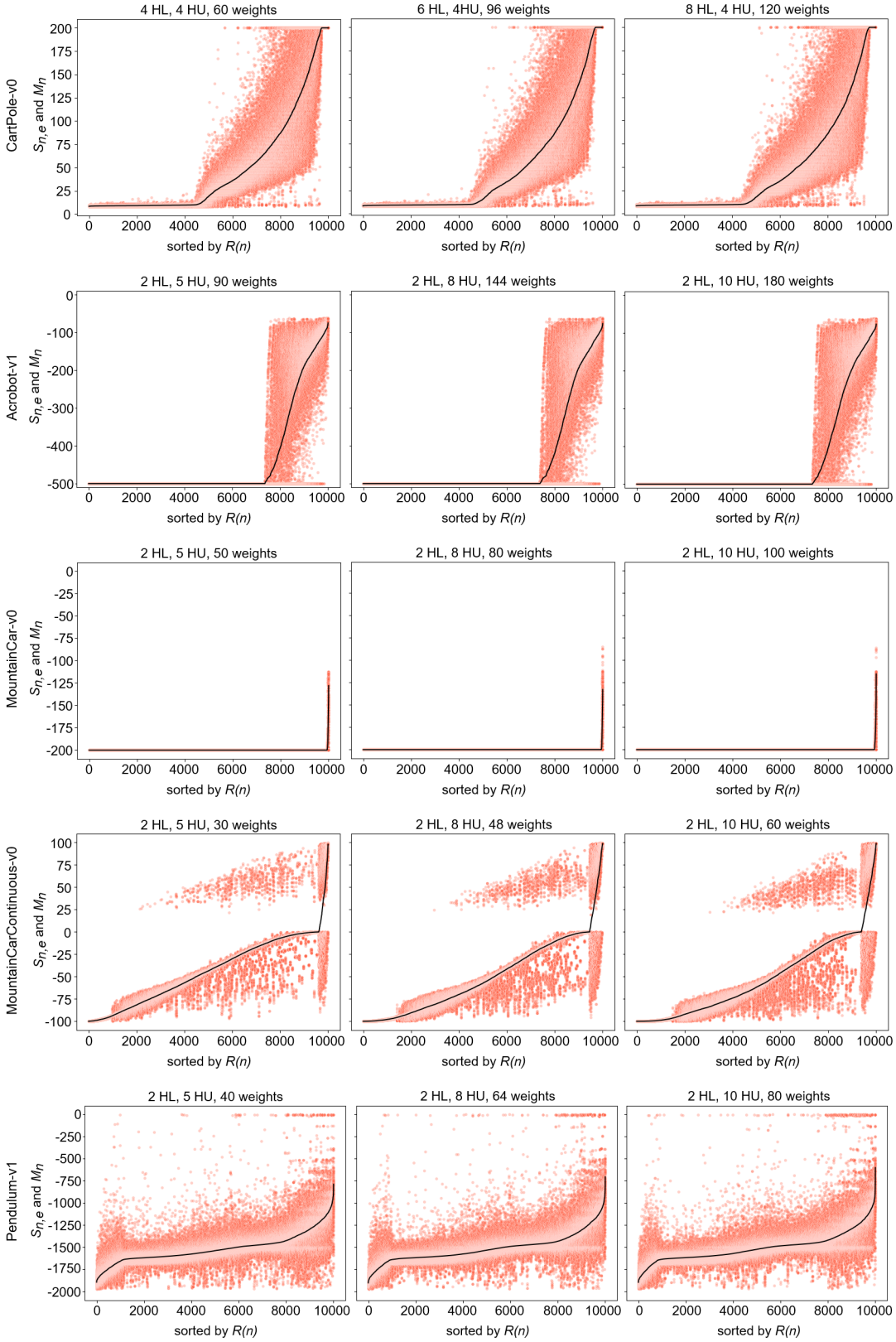


FIGURE 3.10: **Results of altering the number of neurons for neural networks.** The plots show the result of increasing the number of neurons for each layer in a network with two hidden layers. For some environments, there is a change in the mean scores. However, the changes are not extreme.

3.7 Discussion

This chapter delivered a direct comparison between neural networks and alternative models: polynomials and binary trees. In addition, I investigated the impact of bias in these settings. The subsequent paragraphs aim to discuss the results to prepare for the conclusions and proposed future work in the next chapter.

Comparison of polynomial model with neural networks: As we have seen in the last section, the polynomial model can deliver comparable results to neural networks for discrete environments. The results of the environment `MountainCar` with the polynomial model look almost indistinguishable from those of the neural network model. For the environment `Acrobot` the polynomial model with degree delivers very similar results as the neural network without hidden layers. With the higher complexity of the model, neural networks outperform the polynomial model.

The polynomial of degree one results in the same score distribution as the neural network without hidden layers. The score distribution differs noticeably from neural networks with higher model complexity. The mean scores show the same trend for all neural network architectures, which cannot be applied to the polynomial model.

With the linear polynomial model, we have a 50% chance of failing, but the probability of actually solving the task for all episodes is higher than for the polynomials with a higher degree. That means with the linear model or neural networks, we have a larger fraction of samples that can solve the environment independently of the initialization conditions. We could assume that this is an important aspect for such a model and choose the polynomial with degree 1 over one with a higher degree. We should remind ourselves that these experiments are rather unusual for an application since there is no learning involved and the number of samples is huge.

In a standard application, we would use some kind of training and want the model to sequentially improve its performance. When using a learning algorithm, we would not prefer the polynomials with degree one or the neural networks as we might get stuck in a fitness plateau when the algorithm has no method of dealing with this behavior.

The results of the polynomial model are comparable to those of neural networks. This applies only for the three discrete environments since the polynomial model is not constructed to deal with a continuous action space.

Comparison of binary tree model with neural networks: Despite the simple structure of the binary tree model, it delivers comparable results to neural networks and even outperformed them. The binary tree model significantly outperforms neural networks for the environment `CartPole` for all three architectures. The binary tree with only one node shows a similar slope for the mean scores as the network without hidden layers, but it visibly reduces the plateau of minimum scores. The binary trees with higher complexity even manages to remarkably shrink the plateau, which none of the three architectures of the neural network did on that scale.

The environment `MountainCar` yields similar mean scores for the binary trees and the neural networks. There is a large plateau of minmum scores for both models. When comparing the three configurations of the two models with each other, we can see that the binary tree model reaches more and higher individual scores than the neural network models. The results of the two models are similar for the two models for the environment `Acrobot`. The binary tree with one node outperforms the neural network without hidden layers whereas the complexer neural networks outperform the binary trees with four and eight nodes.

Even though the binary tree model at its current implementation outputs only discrete action values, the results for the continuous environment `Pendulum` are comparable to those of neural networks. The binary tree with eight nodes even managed to outperform neural networks. For the environment `MountainCarContinuous`, neural networks delivered better results. Although the binary tree model with one node reached significantly higher mean scores than neural networks, it exhibits a large plateau of minimum scores that we do not see for the neural networks. The networks with higher complexity reached identical high mean scores.

The considerable difference in the score distribution between the two models could be caused by the binary tree producing discrete actions instead of continuous ones, as neural networks did. For the binary tree model, the action is either -1 or 1, limiting the output space significantly. The environment `MountainCarContiuous` seems more sensitive to these discrete action outputs than the environment `Pendulum`. The experiments with the choice of actions resulted in a much better score distribution for the action pair 0 and 1.

The binary tree model produced a similar score distribution as neural networks did. For some environments, the binary trees even outperform neural networks. The environment `MountainCarContinuous` seems more difficult for the binary trees to learn than for neural networks.

Impact of bias: Including bias has a significant negative impact on almost all of the five classic control environments. For the environment `Acrobot` and `Pendulum`, the difference between using bias and not using bias seems less extreme than for the other environments depending on the number of layers used.

To further investigate this behavior, I subsequently changed three aspects of the model's architecture. First, I increased the number of weights of the networks. Despite the remarkably increased number of weights, the results show only slight differences in the score distribution. The exception is the environment `MountainCarContinuous`, where the mean scores are remarkably lower. Except for the environment `MountainCarContinuous`, we can conclude that the increased number of weights does not impact the score distribution significantly.

Second, I increased the number of hidden layers. Some environments show slightly different results like `CartPole` and `MontainCar`. However, the changes are less significant than in the experiments with bias connections. The exception is again `MountainCarContinuous`, where the difference is noticeably more extreme with an increased number of layers.

Finally, I increased the number of neurons in each hidden layer. In the results of this experiment, some changes in the score distribution are visible. The mean scores of the environments `Pendulum` and `MountainCar` show slight differences. The environment `MountainCarContinuous` shows a similar picture as before.

In summary, no other aspect impacts the performance of the model as negative as using bias. The exceptions are the environment `MountainCarContinuous`, where we see even more extreme differences when changing the network architecture, and `Pendulum`, which already shows smaller differences than the other environments when using bias versus not using bias. It seems that this behavior is specific to the bias in the setting of RWG for the environments `CartPole`, `Acrobot`, and `MountainCar`. In addition, we have seen that the environment `MountainCarContinuous` generally reacts sensitive to a change in the model's architecture, showing a similar image as when using bias. In contrast, `Pendulum` does not seem to be significantly affected by a change in the architecture. But as we have seen, the bias also affects it less.

Chapter 4

Conclusion

This work's driving question was whether we could find alternative models for direct policy search whose performance is comparable to neural networks, which were already successfully applied as policy approximators in neuroevolution. To investigate how these models compare to neural networks and what advantages they have, two research questions were introduced in Section 1.7. In addition, one research question was formulated concerning the architecture of a model for additional analysis.

RQ1: How do function approximators other than neural networks compare with the latter? I directly compared the alternative models, the polynomial and binary tree models, to neural networks. The polynomial model produced results comparable to those of neural networks. For the environment `CartPole`, it delivered a more desirable score distribution for polynomials of degrees two and three. We would still prefer neural networks to tackle the problem of the `Acrobot` environment. The environment `MountainCar` seems to produce the same score distribution regardless of the model used. We cannot conclude that using the polynomial model robustly produces better results or results comparable to neural networks.

The binary tree model could deliver comparable results and even outperformed neural networks several times. The simplicity of the model did not limit the model's performance. The opposite is the case. It significantly outperformed neural networks for the environment `CartPole`. Since `CartPole` represents a relatively simple problem, the simple structure of binary trees is advantageous compared to the complex architecture of neural networks. Neural networks have proven that they are capable of solving many challenging problems. Finding a good strategy with neural networks seems less efficient for a more straightforward task.

The search space of the model can explain these results. A more complex model like neural networks with multiple hidden layers has a much larger search space in which they search for a good strategy for a given task. A large search space is a disadvantage when the task is easier to solve. We must carefully select our approach and adapt to the complexity of the task we want to solve.

The environment `MountainCar` again produces a similar score distribution. The mean values seem to be identical. There are a few higher individual scores for the binary tree model. As it comes from the construction of the environment, the problem implemented in `MountainCar` is hard to solve by RWG. All of the tested models showed similar findings. The large fitness plateau remains for all of them. as it comes from the construction of the environment. To solve the problem, we need a learning algorithm high in exploration. For this use case, techniques in black-box optimization offer a good framework. Training a model with a learning algorithm prone to getting stuck in a local optimum will likely not result in a robust model that can solve the task. Also for the environment `Acrobot`, the results of the two models

look very similar. Although there are a few differences, both models seem equally suited to solve this task with the proper learning technique. Since a fitness plateau is present, an explorative learning algorithm should produce the best results.

Although the binary tree model only outputs discrete actions that are fixed, the results for the environment `Pendulum` can compare to those of neural networks. Both models seem to be suited to solve the task reliably with an appropriate learning algorithm. In this case, the danger of being stuck in a local optimum is less prevalent than for the environment `Acrobot`. Thus, the learning algorithm does not have to be as explorative.

The environment `MountainCarContiuous` delivered counterintuitive results. The restriction of the binary tree model to only output the two actions -1 and 1 from the otherwise continuous action space negatively influenced the model's performance. Changing the actions from -1 and 1 to 0 and 1 resulted in a significantly better score distribution. Since the action defines the directional force applied to the car, a value between 0 and 1 means that we apply no force at all or actively push the car to the right in the direction of the target. Thus, the results show that the model does not necessarily have to learn to accelerate the car to the left and right to solve the task.

To conclude these findings, the binary tree model looks promising for direct policy search problems in reinforcement learning. Combined with a suited learning technique, it should robustly produce results comparable to those of neural networks.

RQ2: What advantages and disadvantages can we see with other function approximators? With neural networks, we have many hyperparameters that need fine-tuning. Hyperparameters such as the activation function, number of hidden layers and neurons influence the network structure significantly. Parameters such as the learning rate, number of epochs, and batch size, on the other hand, influence the training of the network. For both the polynomial model and the binary tree model, we have only one parameter, namely the degree of the polynomial and the number of nodes of the binary tree. Thus, finding a suitable configuration of the architecture of the alternative is straightforward and does not need much effort as opposed to neural network models. In addition, the binary tree model offers much more insight into decision-making than the neural network model. To comprehend how a neural network makes a decision is very hard. The binary tree model offers much more interpretability in that sense.

RQ3: How does the bias influence the performance of the model? Adding bias had a negative impact on all discrete models for the polynomial model and on almost all five classic control environments for the neural network model. In my experiments with the number of weights, the number of layers, and the number of neurons, none of these changes influenced the model's performance as much as the bias did. There seems to be something specific about the use of bias that changes the performance for these environments in the setting of RWG. Why this is the case needs further research. However, there are two exceptions. The environment `MountainCarContiuous` was very reactive to each change in the architecture or the number of weights. The environment `Pendulum` did not show a significant difference when using bias vs. when not using bias. A change in the architecture or the number of weights affected it similarly.

4.1 Future Work

There is much further research that can be done in this area. This work represents the first step in the direction of alternative models like the binary tree model. This thesis showed that the binary tree model looks promising and can produce results comparable to neural networks while maintaining a simple architecture and interpretability. However, it would be interesting how we can adapt the model to work better on specific problems. The current implementation of the model can undoubtedly be improved. For continuous action spaces, we can adapt the leaf nodes to be included in the learning algorithm or implement a function instead of outputting a fixed action.

So far, no practical learning technique has been involved. It would be interesting to see how the model performs with state-of-the-art continuous black-box optimization optimizers such as CMA-ES replacing RWG. Additionally, the model should be tested in more challenging benchmark problems like the environment `BipedalWalker` also included in the set of environments provided by the OpenAI Gym interface.

Finally, it would be interesting to see how other models perform in this setting. With techniques in black-box optimization, there are numerous possibilities.

Bibliography

Anderson, Charles W (2000). "Approximating a policy can be easier than approximating a value function". In: *Computer Science Technical Report*.

Arulkumaran, Kai et al. (2017). "Deep Reinforcement Learning: A Brief Survey". In: *IEEE Signal Processing Magazine* 34.6, pp. 26–38. DOI: 10.1109/MSP.2017.2743240.

Barto, Andrew G., Richard S. Sutton, and Charles W. Anderson (1983). "Neuronlike adaptive elements that can solve difficult learning control problems". In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-13.5, pp. 834–846. DOI: 10.1109/TSMC.1983.6313077.

Bellemare, Marc G et al. (2013). "The arcade learning environment: An evaluation platform for general agents". In: *Journal of Artificial Intelligence Research* 47, pp. 253–279.

Breiman, Leo et al. (2017). *Classification and regression trees*. Routledge.

Brockman, Greg et al. (2016). *OpenAI Gym*. eprint: arXiv:1606.01540.

Buşoniu, Lucian et al. (2018). "Reinforcement learning for control: Performance, stability, and deep approximators". In: *Annual Reviews in Control* 46, pp. 8–28.

Cuccu, Giuseppe, Julian Togelius, and Philippe Cudré-Mauroux (2018). "Playing Atari with Six Neurons". In: *CoRR* abs/1806.01363. arXiv: 1806.01363. URL: http://arxiv.org/abs/1806.01363.

Deisenroth, Marc Peter, Gerhard Neumann, Jan Peters, et al. (2013). "A survey on policy search for robotics". In: *Foundations and Trends® in Robotics* 2.1–2, pp. 1–142.

Dong, Hao et al. (2020). *Deep Reinforcement Learning*. Springer.

Duan, Yan et al. (2016). "Benchmarking deep reinforcement learning for continuous control". In: *International conference on machine learning*. PMLR, pp. 1329–1338.

El-Fakdi, Andres, Marc Carreras, and Narcís Palomeras (2005). "Direct Policy Search Reinforcement Learning for Robot Control." In: *CCIA*, pp. 9–16.

Fischer, Gerd (2014). *Lineare Algebra*. Springer.

François-Lavet, Vincent et al. (2018). "An introduction to deep reinforcement learning". In: *Foundations and Trends® in Machine Learning* 11.3-4, pp. 219–354.

Golovin, Daniel et al. (2017). "Google vizier: A service for black-box optimization". In: *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pp. 1487–1495.

Ha, David (2017). "A Visual Guide to Evolution Strategies". In: *blog.otoro.net*. URL: https://blog.otoro.net/2017/10/29/visual-evolution-strategies/.

Hansen, Nikolaus (2016). "The CMA evolution strategy: A tutorial". In: *arXiv preprint arXiv:1604.00772*.

Kaelbling, Leslie Pack, Michael L Littman, and Andrew W Moore (1996). "Reinforcement learning: A survey". In: *Journal of artificial intelligence research* 4, pp. 237–285.

Kober, Jens, Betty Mohler, and Jan Peters (2010). "Imitation and reinforcement learning for motor primitives with perceptual coupling". In: *From motor learning to interaction learning in robots*. Springer, pp. 209–225.

Kyriakides, George and Konstantinos G. Margaritis (2020). "An Introduction to Neural Architecture Search for Convolutional Networks". In: *CoRR* abs/2005.11074. arXiv: 2005.11074. URL: https://arxiv.org/abs/2005.11074.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). "Deep learning". In: *nature* 521.7553, pp. 436–444.

Montague, P Read (1999). "Reinforcement learning: an introduction, by Sutton, RS and Barto, AG". In: *Trends in cognitive sciences* 3.9, p. 360.

Moore, Andrew William (1990). "Efficient memory-based learning for robot control". In.

Oller, Declan, Tobias Glasmachers, and Giuseppe Cuccu (Apr. 16, 2020). "Analyzing Reinforcement Learning Benchmarks with Random Weight Guessing". In: *arXiv:2004.07707 [cs, stat]*. arXiv: 2004.07707. URL: http://arxiv.org/abs/2004.07707 (visited on 12/07/2021).

Ollivier, Yann et al. (2017). "Information-geometric optimization algorithms: A unifying picture via invariance principles". In: *Journal of Machine Learning Research* 18.18, pp. 1–65.

Quinlan, J Ross (2014). *C4. 5: programs for machine learning*. Elsevier.

Recht, Benjamin (2018). "A tour of reinforcement learning: The view from continuous control". In: *arXiv preprint arXiv:1806.09460*.

Rosenblatt, Frank (1958). "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6, p. 386.

Schmidhuber, Jürgen, Sepp Hochreiter, and Yoshua Bengio (2001). "Evaluating benchmark problems by random guessing". In: *A Field Guide to Dynamical Recurrent Networks*, pp. 231–235.

Silva, Andrew et al. (2020). "Optimization methods for interpretable differentiable decision trees applied to reinforcement learning". In: *International conference on artificial intelligence and statistics*. PMLR, pp. 1855–1865.

Such, Felipe Petroski et al. (2017). "Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning". In: *arXiv preprint arXiv:1712.06567*.

Sutton, Richard S (1995). "Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding". In: *Advances in Neural Information Processing Systems*. Ed. by D. Touretzky, M.C. Mozer, and M. Hasselmo. Vol. 8. MIT Press.

Sutton, Richard S and Andrew G Barto (2018). *Reinforcement learning: An introduction*. MIT press.

Sutton, Richard S et al. (1999). "Policy gradient methods for reinforcement learning with function approximation". In: *Advances in neural information processing systems* 12.

Wierstra, Daniel (2010). "A Study in Direct Policy Search". PhD thesis. Technische Universität München.

Wu, Xindong et al. (2008). "Top 10 algorithms in data mining". In: *Knowledge and information systems* 14.1, pp. 1–37.

Yang, Yongxin, Irene Garcia Morillo, and Timothy M Hospedales (2018). "Deep neural decision trees". In: *arXiv preprint arXiv:1806.06988*.