

UNIVERSITY OF FRIBOURG

MASTER THESIS

Non-Differentiable Function Approximation: Beyond Neural Networks

Author:
Corina Masanti

Supervisor:
Dr. Giuseppe Cuccu

Co-Supervisor:
Prof. Dr. Philippe
Cudré-Mauroux

January 01, 1970

eXascale Infolab
Department of Informatics

Abstract

Corina Masanti

Non-Differentiable Function Approximation: Beyond Neural Networks

Neural networks as generic function approximators can solve many challenging problems. However, they can only be applied successfully for a suited problem structure. In specific, neural networks require differentiability. But there are many areas where calculating an accurate gradient is non-trivial, including problems in Reinforcement Learning (RL). In contrast, Black-Box Optimization (BBO) techniques are less limiting. 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 RL. This thesis aims to achieve good results with a function approximator other than neural networks. I analyze promising models optimized with BBO methods.

Problem -> Solution -> Results
TODO: describe models and results

Keywords: Black-Box Optimization, Reinforcement Learning

Contents

Abstract	iii
1 Introduction	1
1.1 General Notation	1
1.2 Reinforcement Learning Control Problems	1
1.3 Classic Reinforcement Learning	2
1.3.1 Neural Networks as Models	3
1.4 OpenAI Gym	4
1.5 Direct Policy Search	4
1.6 Neuroevolution	4
1.7 Black-Box Optimization	4
1.7.1 Random Weight Guessing	5
1.7.2 Evolution Strategies	5
1.7.3 Covariance Matrix Adaptation Evolution Strategy	5
1.8 Research Questions	6
2 Method	7
2.1 Benchmarks in Reinforcement Learning	7
2.1.1 Impact of Bias	9
2.2 OpenAI Gym Environments	10
2.2.1 CartPole	11
2.2.2 Acrobot	12
2.2.3 MountainCar	13
2.2.4 MountainCarContinuous	14
2.2.5 Pendulum	14
2.3 Visualization	15
2.4 Alternative Models	15
2.4.1 Polynomials	16
2.4.2 Splines	18
2.4.3 Binary Trees	18
3 Experiments	19
3.1 Experiments	19
3.1.1 Reproduction of RWG-Paper	19
3.1.2 Comparison of Alternative Models to Neural Networks	19
3.1.3 Analysis of the Impact of Bias	20
3.2 Results	21
3.2.1 Reproduction of RWG-Paper	21
3.2.2 Polynomial Model	21
3.2.3 Binary Tree Model	24
3.2.4 Bias Investigation	26
3.3 Discussion	30

4 Conclusion	31
4.1 Conclusion	31
4.2 Future Work	31
Bibliography	33
5 Appendix	35
5.1 Additional Plots	35

List of Figures

1.1	Interaction loop between an RL agent and the environment	2
1.2	Sketch of a Neural Network	3
2.1	Reproduced Plots	9
2.2	Impact of Bias	10
2.3	Illustration of the environment CartPole	11
2.4	Illustration of the environment Acrobot	12
2.5	Illustration of the environment MountainCar	13
2.6	Illustration of the environment Pendulum	14
2.7	Visualization of results	15
2.8	Upper and lower bound	17
2.9	Sigmoid function	17
3.1	Results for all classic control environments using neural networks without bias	22
3.2	Results for the discrete classic control environments using the polynomial model P_1 without bias	23
3.3	Results for the discrete classic control environments using the polynomial model P_1 with bias	24
3.4	Results for the classic control environments using binary trees without bias	25
3.5	Results for all classic control environments using neural networks with bias	27
3.6	Results of tripling the number of weights for neural networks	28
3.7	Results of altering the number of layers for neural networks	29
3.8	Results of experiment 2: neurons	29

Chapter 1

Introduction

1.1 General Notation

First, I am defining the notation I used throughout this thesis. For the scalars, I used regular, normal-weight variables such as x . Vectors are represented by bold lower-case variables like \mathbf{x} . Tensors and matrices are denoted by bold upper-case characters, such as \mathbf{X} . In summary:

x	: Scalar
\mathbf{x}	: Vector
\mathbf{X}	: Matrix or tensor

Useful? Everything covered?

1.2 Reinforcement Learning Control Problems

mention environment, reward

In Reinforcement Learning (RL), we have an agent and an environment. The agent can take on actions to interact with the environment. The environment, on the other hand, always holds a certain state that can be altered by the actions of the agent. The goal of the agent is to learn a policy π that drives the system into the desired state by taking the right actions. For example, the goal might be to solve skill games like Ball-in-a-Cup or Kendama (Kober, Mohler, and Peters, 2010). However, we do not have data beforehand as we do in supervised learning problems. We only get a scalar reward signal that indicates how well the agent performs. The goal of the agent is to maximize the cumulative rewards received by the environment. Therefore, the agent needs to learn optimal behavior through trial-and-error interaction with the environment. Figure 1.1 illustrates the loop between the agent and the environment. The agent has received the observation o_t from the environment and therefore knows the current state of it. The agent then performs some action a_t in the environment. The environment changes according to the received action and returns an observation o_{t+1} that indicates the current state of the environment and a reward r_{t+1} . One such interaction between the agent and the environment represents one *timestep*.

The rewards correspond to an underlying reward function that is specific to the environment. The reward function is a function that maps the current state of the environment, the current action, and the subsequent state into a scalar value. The function can be deterministic or it can have a random component making it stochastic. Depending on the problem, the reward can be sparse or dense. Sparse reward

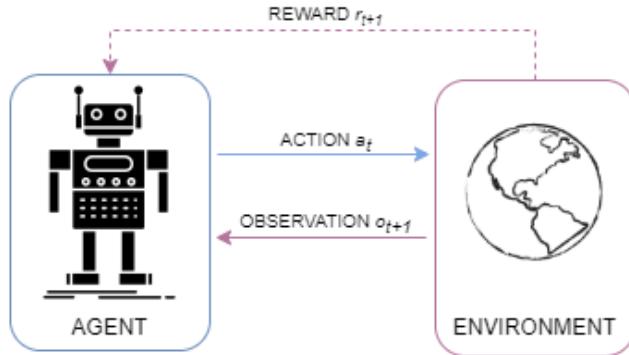


FIGURE 1.1: Interaction loop between an RL agent and the environment. At timestep t , the agent receives the observation o_t from the environment. The agent decides to select action a_t and interacts with the environment. The environment changes and returns the next observation o_{t+1} and the reward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

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 huge challenge since we cannot know which particular actions led to success or failure. Optimization problems with this setting are generally difficult. Dense reward environments return a meaningful reward at almost every time step.

Learning a policy that maximizes the returned rewards is generally non-trivial. The agent can only learn through trial and error. If the policy is not explorative enough, we might get stuck in a local optimum and never reach the goal. However, if there is too much exploration we loose reward. Thus, there needs to be a balance between exploration and exploitation. In addition, often there is 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).

Include MDP? Talk about sample efficiency? Talk about value function already? Talk about model-based and model-free?

1.3 Classic Reinforcement Learning

Classic RL framework and approach

We can solve RL problems with methods based on value functions or methods based on policy search. The value function is a prediction of future reward of landing on a specific state. It indicates how good or bad being in a certain state is. With methods based on value functions, the agent chooses the best action in the state. 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).

Include Q-Learning, reward shaping? show more the difference between this and dps

1.3.1 Neural Networks as Models

Neural networks are machine learning algorithms inspired by the functionalities of the human brain. They consist of connected neurons or nodes that imitate the biological neurons of the brain sending signals to each other. A neuron receives one or multiple input values and calculates one or multiple output values. Figure [insert figure] illustrates one neuron. The neuron receives three input values and calculates one output value. For the calculation of the output, the neuron uses the assigned weights w . Thus, the output represents the sum of weighted inputs.

In a neural network, the neurons 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 1.2 shows a sketch of a network with three hidden layers that are fully connected. Each neuron holds an associated weight and threshold. It receives

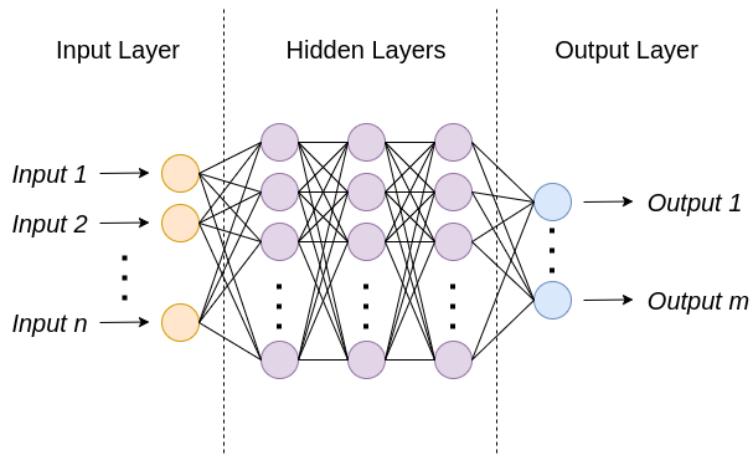


FIGURE 1.2: **Sketch of a neural network.** The image shows a sketch of a neural network with three fully connected hidden layers. The network expects n inputs and has m possible outputs.

an input and outputs the sum of weighted inputs. If this sum is greater than the associated threshold, the neuron gets activated.

A large part of the success of neural networks can be traced to the use of *backpropagation*. Backpropagation uses information from the previous epoch (i.e. iteration) to adjust the weights of the network using the error gradient.

A neural network with one or more hidden layers defines a non-linear function. It is theoretically able to represent any function. Thus, neural networks are *generic function approximators*. In addition, neural networks are highly expressive and flexible. They are applied successfully in many areas including in RL problems. Neural networks can be used to approximate a value function or a policy function. Deep RL combines RL with the methods from deep learning. It has been used in many applications including robotics, video games, computer vision (François-Lavet et al., 2018).

Should NN be explained in more detail? Write more about DRL, include recent results, advantages, disadvantages

1.4 OpenAI Gym

mention as example of control problems, that it's a common benchmark in the literature, provides environments

OpenAI Gym¹ is a toolkit that provides environments that represent general RL problems. They are easy to use and serve as benchmark problems with varying difficulty and challenges.

Explain more, compare with Section 2.2 to not repeat

1.5 Direct Policy Search

mention it as an alternative to Deep RL, mention it uses networks to approximate policy rather than value function

An alternative to the methods described so far is direct policy search. Instead of approximating a value function, we approximate the policy directly with an independent function approximator with its own parametrization. Using a value function can work well but there are some limitations. Studies suggest that approximating a policy can be easier to use and deliver better results (Sutton et al., 1999; Anderson, 2000). Here we have two approaches: gradient-based and gradient-free methods.

Explain more, go more into advantages, find more papers with comparison

1.6 Neuroevolution

special case of DPS: NN as policy approximator, train with evo

Neuroevolution is a special case of direct policy search. A neural network is used to approximate the policy and optimized with evolutionary algorithms. It was shown that genetic algorithms can work well even in hard settings (Such et al., 2017).

Explain more

1.7 Black-Box Optimization

but what is evo? here's a primer on black box optimization

In Black-Box Optimization (BBO), the problem structure, as well as the model, remains unknown. BBO methods optimize a parametrization without any constraints on the problem, model or solution needed. Thus, there is no constraint on differentiability or convexity. The optimization is done solely based on a score or cumulative reward, which makes these methods directly applicable to Reinforcement Learning (RL) problems.

Explain more

¹<https://www.gymlibrary.ml/>

1.7.1 Random Weight Guessing

Randomization and RWG (uniform distribution)

Random Weight Guessing (RWG) is the simplest BBO method. With RWG, we repeatedly randomly initialize the weights of a model until the resulting model classifies all training instances correctly. Then we test the model on a separate test set (Schmidhuber, Hochreiter, and Bengio, 2001). RWG is not a reasonable learning algorithm, but it can be used as an analysis method (Oller, Glasmachers, and Cuccu, 2020).

Explain more, include limitations (expensive algorithms)

1.7.2 Evolution Strategies

ES-(1+1) (normal, adapt mean)

Evolution Strategies (ES) belong to the class of evolutionary algorithms and are 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. In each generation new individuals (candidate solutions) are generated using the current parents. We continue with the next generation until a stopping criteria is met. ES are multi-agent algorithms which means that they generate multiple parametrizations that explore a different path or area in the optimization space independently. This exploration prevents us from landing 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 performed had a better fitness score. Algorithm 1 shows this process in pseudocode.

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

```

1: Parameter:  $\sigma > 0$ 
2: Initialization:  $x = x_0 \in \mathbb{R}^d$ 
3: while not done do
4:   Sample  $x' \sim \mathcal{N}(x, \sigma^2 I)$ 
5:   if  $f(x') \leq f(x)$  then
6:      $x \leftarrow x'$ 
7: Return  $x$ 
```

Explain more

1.7.3 Covariance Matrix Adaptation Evolution Strategy

CMA-ES (which is a ES-(mu,lambda)) (normal, adapt mean, adapt covariance, large pop)

Covariance Matrix Adaptation Evolution Strategy (CMA-ES) is a method for non-linear, non-convex optimization problems in continuous domain. For each iteration

(generation), a population of new individuals is generated by sampling a multivariate normal distribution. After each iteration, the mean and the covariance matrix is updated (Hansen, 2016).

Include formulas from tutorial

1.8 Research Questions

Start from the experiments. They are your answer. Which question do they answer? These are your research questions.

In the experiments section, mention explicitly which question each answers

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 function approximators other than neural networks compare with the latter?
2. What impact has the bias on the performance of the model?
 - (a) Does increasing the number of weights worsen the performance of the model?
 - (b) Does a neural network with more than two hidden layers yield worse scores?
 - (c) Does the neural network's performance suffer from an increase in the number of neurons?
 - (d) Does the difficulty of the environment affect the observed effect of the bias?

Move to experiments? Or formulate questions concerning bias later? Maybe divide first question into multiple smaller ones.

Chapter 2

Method

2.1 Benchmarks in Reinforcement Learning

When developing a novel algorithm, it is important to compare our results with existing models. For this evaluation, we need standard benchmark problems. These are a set of standard optimization problems. OpenAI Gym is a toolkit created for exactly this scenario. As mentioned in Section 1.4, it contains a collection of benchmark problems with various levels of difficulty. However, not all benchmark problems are meaningful for the evaluation of an algorithm. If a problem is too trivial to solve, the results do not reflect the quality of the model adequately. We do not need to put a large amount of effort into the creation of a complex model for an easy-to-solve task.

In the paper *Analyzing Reinforcement Learning Benchmarks with Random Weight Guessing* (Oller, Glasmachers, and Cuccu, 2020), the authors analyze and visualize the complexity of standard RL benchmarks based on score distribution. They tested their approach on the five Classic Control benchmark problems from the OpenAI Gym interface: CartPole, Acrobot, Pendulum, MountainCar, and MountainCarContinuous. Given an RL environment, the authors conducted a fixed series of experiments. For these experiments, 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), and 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. The evaluation of the benchmark problems should be as objective as possible and should not include bias in the data. To achieve this, the authors did not include any learning opportunities for the network models. Instead, they chose the network weights i.i.d. from the standard normal distribution $\mathcal{N}(0, 1)$ with Random Weight Guessing (RWG). This approach assures randomness and no directed learning. The goal of the paper was not to further analyze the network models but to investigate the benchmark problems themselves. With this in mind, they initialized 10^4 samples ($N_{samples} = 10^4$) with different random weights. The number of samples would be too large for a reasonable learning strategy. However, the large number of samples serves a different purpose than optimizing the results. Instead, the aim is to draw statistical conclusions. Each of these samples of a neural network represents a controller that maps observations to actions in the environment. Later in this thesis, I will explore function approximators other than neural networks representing the controller. In the paper, the authors tested the controllers for each environment during 20 independent episodes ($N_{episodes} = 20$). For each episode, they saved the score in the score tensor S . Algorithm 2 illustrates the procedure with pseudocode.

After the authors obtained the scores, they calculated the mean performance over all episodes from a sample and its variance. These statistics are significant insights. They can reveal how stable the network models are in completing a given task. A

Algorithm 2 Evaluation process taken from Oller, Glasmachers, and Cuccu, 2020

```

1: Initialize environment
2: Create array  $S$  of size  $N_{architectures} \times N_{samples} \times N_{episodes}$ 
3: for  $n = 1, 2, \dots, N_{samples}$  do
4:   Sample NN weights randomly from  $\mathcal{N}(0, 1)$ 
5:   for  $e = 1, 2, \dots, N_{episodes}$  do
6:     Reset the environment
7:     Run episode with NN
8:     Store accrued episode reward in  $S_{a,n,e}$ 
```

low mean value suggests that, in general, the network cannot 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. Therefore, we strive for a high value for the mean and a low value for the variance. However, training a network with random weight guessing should generally not result in 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 ranked the samples according to their mean scores. They then visualized their results with three plots: a log-scale histogram of the mean scores, a scatter plot of the sample scores over their rank, a scatter plot of score variance over the mean score.

I reproduced the results of the authors following the mentioned methodology. My findings for the environment `CartPole` are displayed in Figure 2.1. The plots illustrate the results for each of the three network architectures. Each row shows the histogram of the mean score values in the left image, the scatter plot of all scores over their rank in the image in the middle, and the scatter plot of the score variance over the mean score in the right image for a specific network architecture. There are few differences, but overall all network architectures deliver similar insights. The histogram plots show that the majority of networks receive a low score. Since the weights of the networks were chosen with RWG, this is rather unsurprising. But there is still a significant amount of networks that were able to achieve a high mean value or even the maximum value of 200. With a score of 200, the network was able to solve the task each episode. Therefore, the network could reliably solve the task without any learning technique involved. This should not be the case for a complex task. Furthermore, in the scatter plot in the middle, we can see that the line plot of the mean scores is a continuous increasing line without any jumps. Thus, a suited RL algorithm should generally be able to learn the task incrementally without converging into a local optimum. At the top of the scatter plot, we can see quite a few data points with a score of 200 that have a relatively low mean score. This indicates that a network that generally performs poorly can still solve the task with the right initialization conditions. Lastly, in the scatter plot on the right, we can see the distribution of the variance according to the mean value. On the left side, we have low scores of variance corresponding with a low mean value. These networks were consistently unable to achieve a high score. Without any training involved, we can expect most networks to be in this area. However, in the middle of the plot, the data points are spread out. For a high variance, the scores of a network differ highly from the mean value. Thus, we might get lucky and receive a high score depending on initialization conditions, but we might as well get a low score. These networks are inconsistent and unstable. On the right side of the scatter plot, we can see that

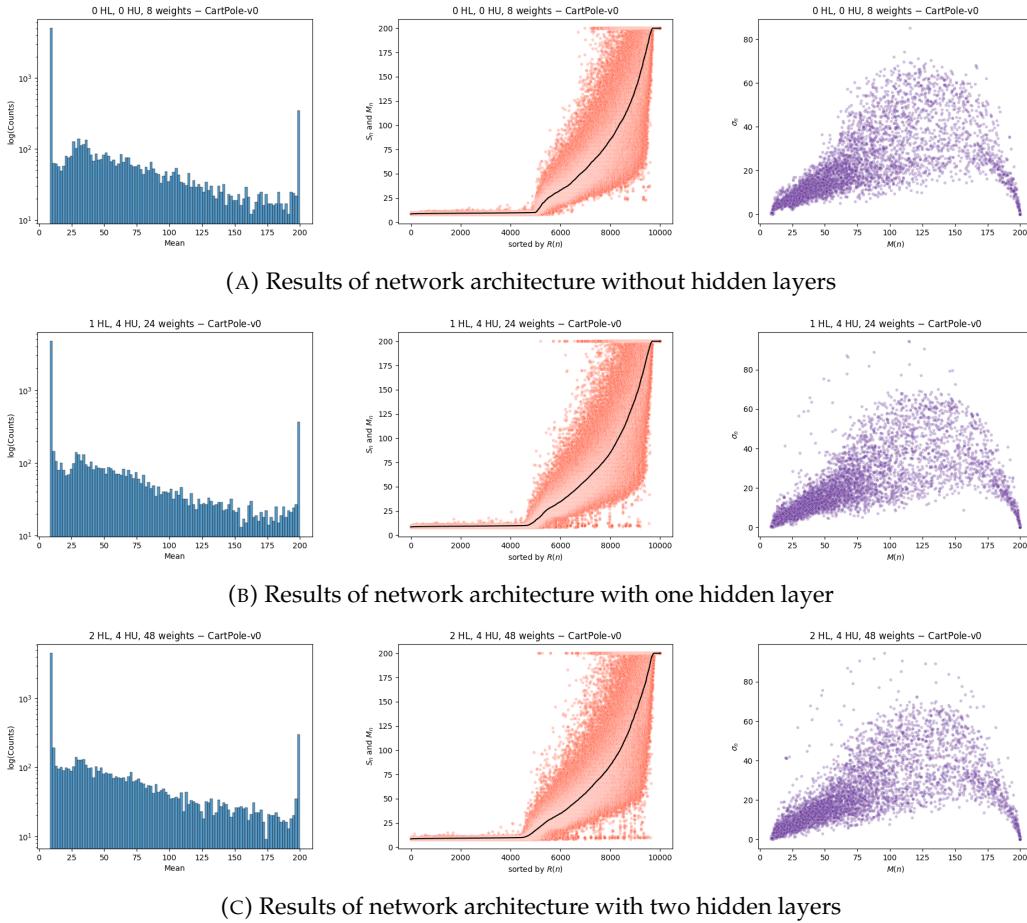


FIGURE 2.1: Results of the benchmark evaluation. The plots show a log-scale histogram of the mean scores (left images), a scatter plot of the sample scores over their rank (middle images), a scatter plot of score variance over the mean score (right image). As expected with RWG, most networks were not able to solve the given task. However, there is still a significant amount of samples achieving a mean score of 200. That suggests that the environment is trivial to solve.

the data points with a high mean value are mostly of low variance. Thus, to achieve a high mean value, the network needs consistency.

2.1.1 Impact of Bias

Interestingly, the usage of the bias had a relatively large impact on the performance of the network in my experiments. Without bias, the networks seem to achieve overall better scores. All plots in Figure 2.1 illustrate the results without bias. For comparison, Figure 2.2 shows the results of a network with two hidden layers with the same configurations as before but this time including bias. As we can see, the networks with bias connections have a much lower score in general. The 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. Or in other words: the

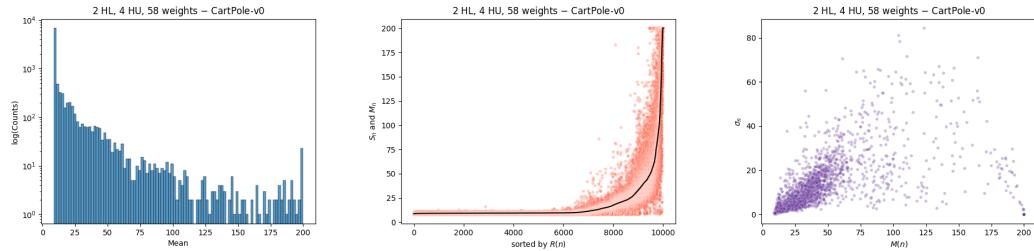


FIGURE 2.2: **Impact of bias.** The figures show the performance of a network with two hidden layers with the same settings as before but here we include bias. We can clearly see that the network with bias connection performs much worse than the one without.

more possibilities we have to falsely guess a weight, the higher the probability to fail. 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 in general. Randomly guessing the parameters of a simple model has a higher chance to result 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 can be approximated. 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. As the paper showed, 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. However, 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. But 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.

The experiments following this thought process are described in Chapter 3.

How much should be included here and how much in experiments/visualization? Rename section name?

2.2 OpenAI Gym Environments

OpenAI Gym offers five classic control environments ready to use: CartPole, Acrobot, MountainCar, MountainCarContinuous, and Pendulum. The initial state of each environment is stochastic. Out of all environments OpenAI Gym provides, the classic control environments are considered as easier ones to solve. Each environment has an observation and an action space. An agent performs some action from the action space and observes how the environment state changes.

2.2.1 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 track. Fig. 2.3 shows one possible state of this problem.

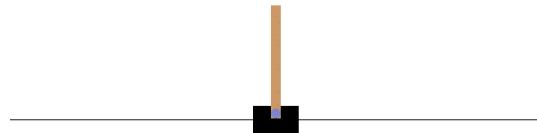


FIGURE 2.3: **Illustration of the environment CartPole.** The image shows one possible state of the CartPole environment.

Action Space: The CartPole environment accepts one action per step and has a discrete action space of 0, 1. The action indicates the direction in which we push the cart by a fixed force. 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.** List of all possible actions for the CartPole environment.

Observation Space: For CartPole, we receive four observations that inform us about the position and velocity of the cart as well as the pole. Table 2.2 shows all observations with the range of the 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 keep the pole upright as long as possible. Thus, we get a positive reward of +1 for each step that does not meet the termination criteria. An episode terminates when either the pole angle is $\pm 12^\circ$, the cart position is ± 2.4 , or the episode length is greater than 200 (500 for v1).

2.2.2 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 a certain height. Fig 2.4 shows one possible state for this problem with the fixed height indicated by a horizontal line.

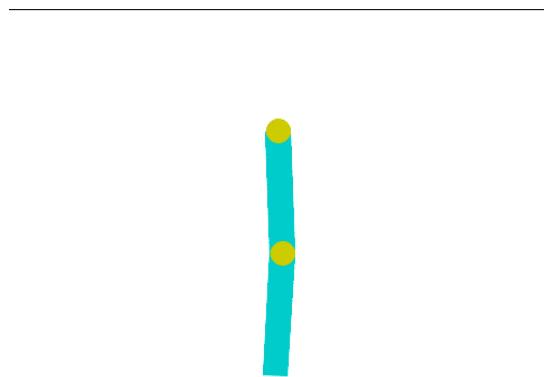


FIGURE 2.4: **Illustration of the environment Acrobot.** The image shows one possible state of the Acrobot environment.

Action Space: For Acrobot, the action space is discrete and deterministic and represents 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.** List of all possible actions for the CartPole environment.

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. theta1 is the angle of the first joint, where an angle of 0 indicates 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.

Reward: The goal is to reach the target height with as little steps as possible. Thus, each step that does not reach the goal receives a negative reward of -1. Reaching the target height results in a reward of 0. The reward threshold is -100.

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.

2.2.3 MountainCar

For the MountainCar problem, we need to maneuver a car up a hill to its target. For this, we need to strategically accelerate the car to reach the goal. This problem first appeared in Andrew Moore's PhD Thesis *Efficient memory-based learning for robot control* (Moore, 1990). There are two versions of this problem, one with a discrete action space and one with a continuous action space. This version has a discrete action space. Fig 2.5 shows one possible initial state of this environment.

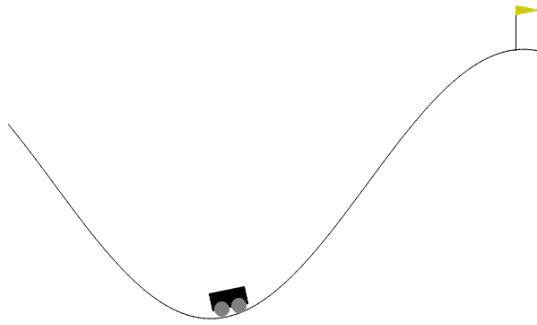


FIGURE 2.5: **Illustration of the environment MountainCar.** The image shows one possible state of the MountainCar environment.

Action Space: The action space consists of three discrete actions. We can either accelerate the car to the right or to the left or we do not interact at all. The actions are further described in Table 2.5

Action	Description	Value	Unit
0	Accelerate to the left	Inf	position(m)
1	Don't accelerate	Inf	position(m)
2	Accelerate to the right	Inf	position(m)

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

Observation Space: The environment gives us two observations with information about the position and the velocity of the car. They are described in Table 2.6.

Reward: To solve the environment, the car has to reach the flag on the right hill

Observation	Min	Max	Unit
Position of the car along the x-axis	-Inf	Inf	position(m)
Velocity of the car	-Inf	Inf	position(m)

TABLE 2.6: **Observation space of the environment MountainCar.**
Description of all observations for the MountainCar environment.

as fast as possible. For every step in which the agent cannot reach the target, he receives a negativ reward of -1.

2.2.4 MountainCarContinuous

MountainCarContinuous represents the same problem as MountainCar but with a continuous action space.

Action Space: This environment acccepts one continuous action per step. The action represents the directional force on the car and is clipped in the range [-1, 1] and multiplied by a power of 0.0015.

Observation Space: The observation space is the same as in MountainCar. Table 2.6 holds the important information.

Reward: Each step in which the car does not reach the target, we receive a negative reward of $-0.1 * \text{action}^2$ to penalise actions of large magnitude. If the goal is reached, we receive a positive reward of +100.

2.2.5 Pendulum

In this environment, 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 free end to swing it to an upright position.



FIGURE 2.6: **Illustration of the environment Pendulum.** The image shows one possible state of the Pendulum environment.

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.** List of all possible actions for the Pendulum environment.

Observation Space: The environment returns three observations.

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.

talk about difficulty / what the environment tests, check if versions are correct, not happy with presentation of this section, reward for Pendulum missing

2.3 Visualization

For the visualization of the results, I am using similar plots as Oller, Glasmachers, and Cuccu, 2020. The type of plots I will be using are already presented in Figure 2.1. However, I will mainly be using the scatter plots shown in the middle. Figure 2.7 shows an example of such a plot.

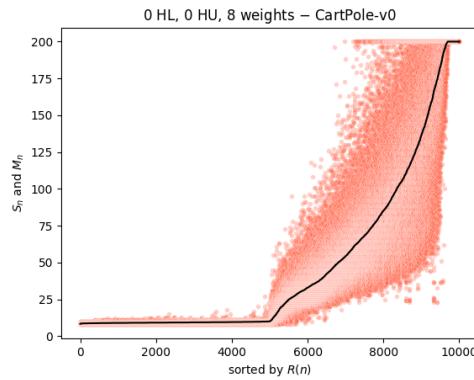


FIGURE 2.7: **Visualization of results.** Example for the visualization of the conducted experiments.

Rethink paper section, present plots here for the first time?

2.4 Alternative Models

We can represent many problems in machine learning as a function mapping an input space into an output space. The output space or *search space* denotes the space of

all feasible solutions. The underlying function is the output of the learning process of an algorithm and is often called the *target function*. Optimally, we would derive the formula of the function explicitly. However, even though we suspect that such a function exists, we usually do not have enough information to derive a formula directly. Thus, we aim to approximate the target function with *function approximators* by using the available data. In general, we can apply any function approximator. However, each function approximator has its advantages and limitations. For example, some may be prone to local optimum. Depending on the task, this could prevent us from finding a good solution.

2.4.1 Polynomials

In mathematics, a polynomial is the 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

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

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 $\mathbf{x} = [x_0, \dots, x_m]^T$ being a vector and of degree n can be represented by

$$p(\mathbf{x}) = \sum_{i=0}^n \mathbf{w}_i^T (x_k^i)_{k \in I} \in F^n[\mathbf{x}], \quad \mathbf{w}_0, \dots, \mathbf{w}_n \in F^n, \quad I = \{0, \dots, m\}.$$

Polynomials are relatively simple mathematical expressions and offer some significant advantages: their derivative and indefinite integral are easy to determine and are also polynomial. 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 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.

For the experiments with a polynomial model in Chapter 3, I used two architectures P_1 and P_2 . The first model P_1 consists of one polynomial for each possible action in a discrete action space. The input of the model is the observation from the environment. The dimension of the weight vectors is according to the dimension of the input vector. 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:

$$\begin{aligned} p_0(\mathbf{x}) &= \sum_{i=0}^n \mathbf{w}_i^T (x_k^i)_{k \in I} \in \mathbb{R}, & \mathbf{w}_0, \dots, \mathbf{w}_3, \mathbf{x} \in \mathbb{R}^4, & I = \{0, 1, 2, 3\} \\ p_1(\mathbf{x}) &= \sum_{i=0}^n \hat{\mathbf{w}}_i^T (x_k^i)_{k \in I} \in \mathbb{R}, & \hat{\mathbf{w}}_0, \dots, \hat{\mathbf{w}}_3, \mathbf{x} \in \mathbb{R}^4, & I = \{0, 1, 2, 3\} \end{aligned}$$

In the formulas, n denotes the degree of the polynomial. In my experiments, I tested polynomials of degrees 1, 2, and 3. The output of the polynomials has no reasonable upper and lower limit, as illustrated in Figure 2.8. That makes it harder to interpret the results reasonably. So, I scaled the outputs with a sigmoid function. A sigmoid function is a mathematical function that maps an arbitrary input space into

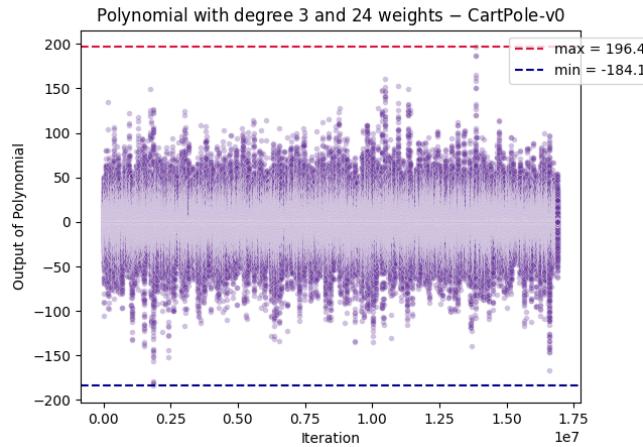


FIGURE 2.8: **Upper and lower bound.** The figure shows each output of the polynomial functions described above. As we can see, the functions are not well bound, and there are quite a few outliers. That makes it hard to interpret the output sensibly.

an output space with a small range, for example, 0 and 1. The function has a characteristic S-shaped curve. We can interpret the output space of the sigmoid function as a probability. In this case, we search for the probability that a specific action is the reasonable one given an observation \mathbf{x} . Thus, for our example with the CartPole environment, we can interpret $\text{sig}(p_0)$ as the probability that action 0 is the correct one and $\text{sig}(p_1)$ 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:

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

For the sigmoid function, I used the logistic sigmoid function. The formula and a plot of the function in 2D are shown in Figure 2.9.

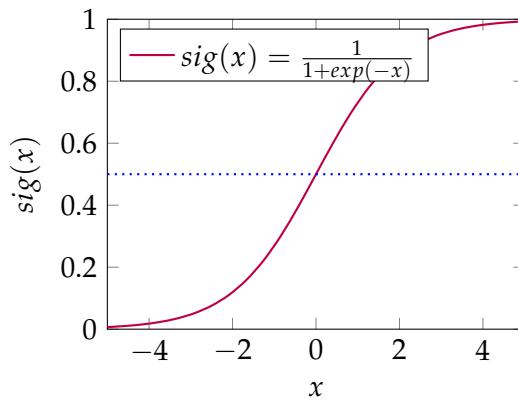


FIGURE 2.9: **Sigmoid function.** The figure shows a plot of the logistic sigmoid function. The function maps an arbitrary input space into the range between 0 and 1. The output of the function can be interpreted as a probability. It is useful to scale data into a meaningful value.

The second model P_2 is constructed similarly to P_1 , but it only consists of one polynomial instead of one for each possible action. For the CartPole environment,

this means P_2 consists of:

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

Analogous to P_1 , I tested the polynomial $p(\mathbf{x})$ with degrees 1, 2, and 3. So, $n \in \{1, 2, 3\}$. In addition, I again used the logistic sigmoid function to scale the output of the polynomial. However, the output of P_2 is determined by a fixed threshold instead of comparing multiple polynomials. Putting this into a formula for the CartPole environment, we get:

$$P_2(\mathbf{x}) = \begin{cases} 1 & \text{if } \text{sig}(p(\mathbf{x})) > 0.5, \\ 0 & \text{otherwise.} \end{cases}$$

Include theorems (Taylor, Weierstass)

Explain difference between approximation and interpolation

Does F^n make sense?

2.4.2 Splines

General splines, adjustments

Splines can also be used as function approximators. However, with splines we do not aim to approximate the whole function but rather return a piecewise approximation.

First was sind splines, dann herleitung in this case.

2.4.3 Binary Trees

A binary tree is a fairly simple data structure in computer science. A binary tree consists of nodes that each have at most two children, referred to as left child and right child.

Herleitung von splines erklären -> vielleicht in vorheriger subsection? Explain continuous envs.

Chapter 3

Experiments

This chapter aims to answer the research questions formulated in Section 1.8. 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 Experiments

This section describes the experiments I used in order to investigate and answer the formulated research questions. We can break down the experiments into three categories: reproduction of the results of the paper *Analyzing Reinforcement Learning Benchmarks with Random Weight Guessing*, comparison of the alternative models described in Section 2.4 to neural networks, and an analysis of the impact of bias in these settings.

3.1.1 Reproduction of RWG-Paper

For a starting point, I reproduced the results of the paper *Analyzing Reinforcement Learning Benchmarks with Random Weight Guessing* discussed in Section 2.1. 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 2 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 guideline to judge the effectiveness of the alternative models in comparison with neural networks.

Section title good?

3.1.2 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 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 P_2 and tested them 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 2.2. 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 2.1. 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 3 shows an overview of the described procedure. As we can see, the procedure is very similar to the one shown in Algorithm 2.

Algorithm 3 Procedure for alternative models using RWG

```

1: Initialize environment
2: Initialize model
3: Create array  $S$  of size  $N_{samples} \times N_{episodes}$ 
4: for  $n = 1, 2, \dots, N_{samples}$  do
5:   Sample model weights randomly from  $\mathcal{N}(0, 1)$  or  $U(a, b)$ 
6:   for  $e = 1, 2, \dots, N_{episodes}$  do
7:     Reset the environment
8:     Run episode with model
9:     Store accrued episode reward in  $S_{n,e}$ 
```

Include splines or not?

3.1.3 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 2.4. 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:

$$\text{Triple number of weights:} \quad \mathbf{w}_i = \mathbf{w}_{i1} + \mathbf{w}_{i2} + \mathbf{w}_{i3}$$

- (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.

Are these adaptions "extreme" enough?

3.2 Results

3.2.1 Reproduction of RWG-Paper

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.

3.2.2 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.

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

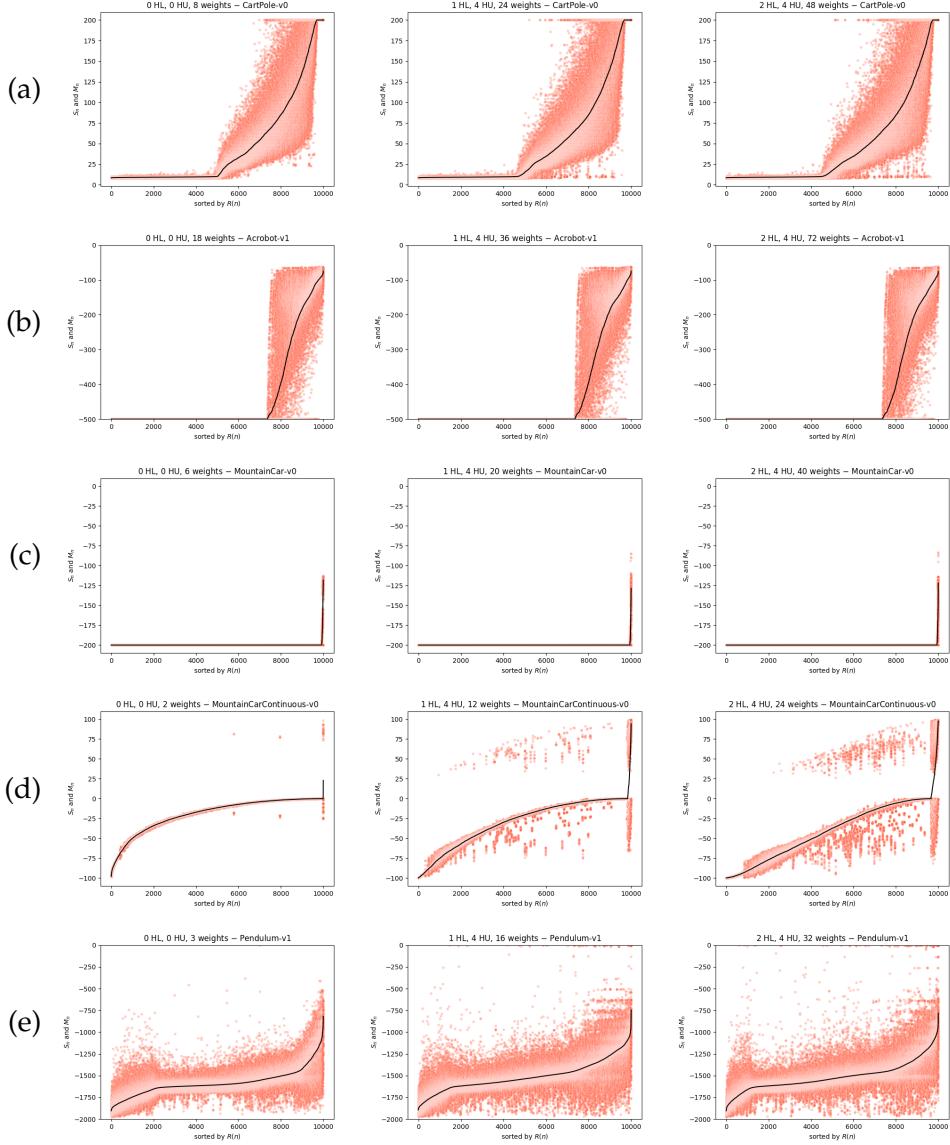


FIGURE 3.1: Results for all classic control environments using neural networks without bias. Each row shows the results of one environment, whereas the columns represent the different network architectures. These plots are a reproduction of the results of the RWG-paper. My results are consistent with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

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 a tiny boost 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, respectively, degree 3.

For the environment Acrobot, the lineplot stays at -500 until around rank 7'500

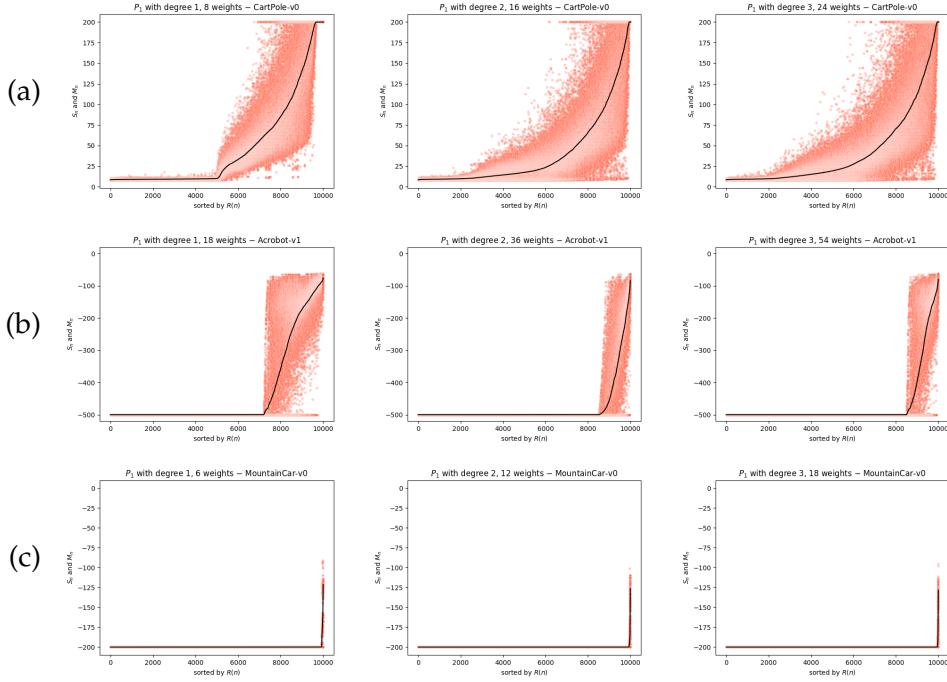


FIGURE 3.2: Results for the discrete classic control environments using the polynomial model P_1 without bias. Each row shows the results of one environment. The columns represent the different degrees of the polynomials. In these plots, the model P_1 was used. The results aligned in the first column 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.

for the linear model. 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. This observation cannot be made for the neural networks.

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.

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 the ones without using bias in Figure 3.2 shows a significant difference. The bias influences the performance of the model negatively. Thus, the observation about the bias that 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.

Add missing plots for bias. Include plots of P_2 as well (failed for MountainCar)?

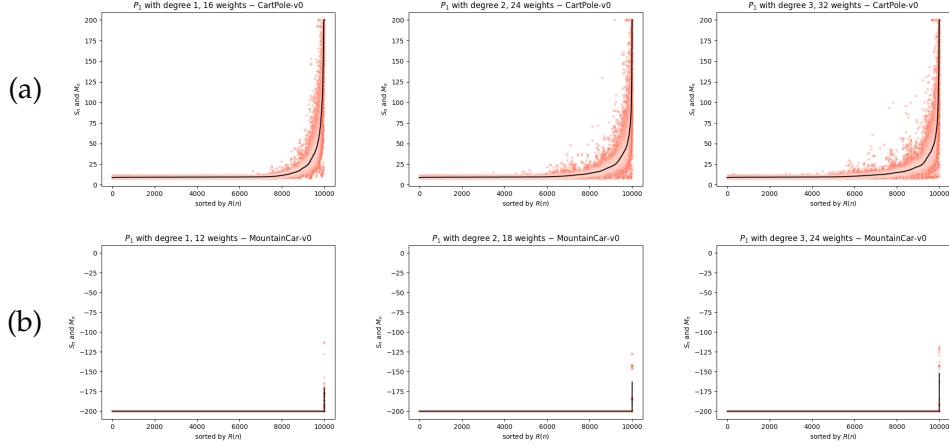


FIGURE 3.3: Results for the discrete classic control environments using the polynomial model P_1 with bias. Each row shows the results of one environment. The columns represent the different degrees of the polynomials. For these plots, the polynomial model P_1 was used. Overall, the model performs noticeably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

3.2.3 Binary Tree Model

For the binary trees, I tested three models, one with 1 node, one with 4 nodes, and one with 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.

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.

For the environment Acrobot, the plots do not show a significant difference between the two models. Overall, we can say that both models reach similar scores and mean values. However, the scores seem to be more spread out for the binary trees than for the neural networks, resulting in a higher variance.

The MountainCar environment is more challenging for a learning algorithm. Comparing the two models, binary trees and neural networks, we can see that the binary trees are more likely to reach a better score. However, the risk of being stuck in a

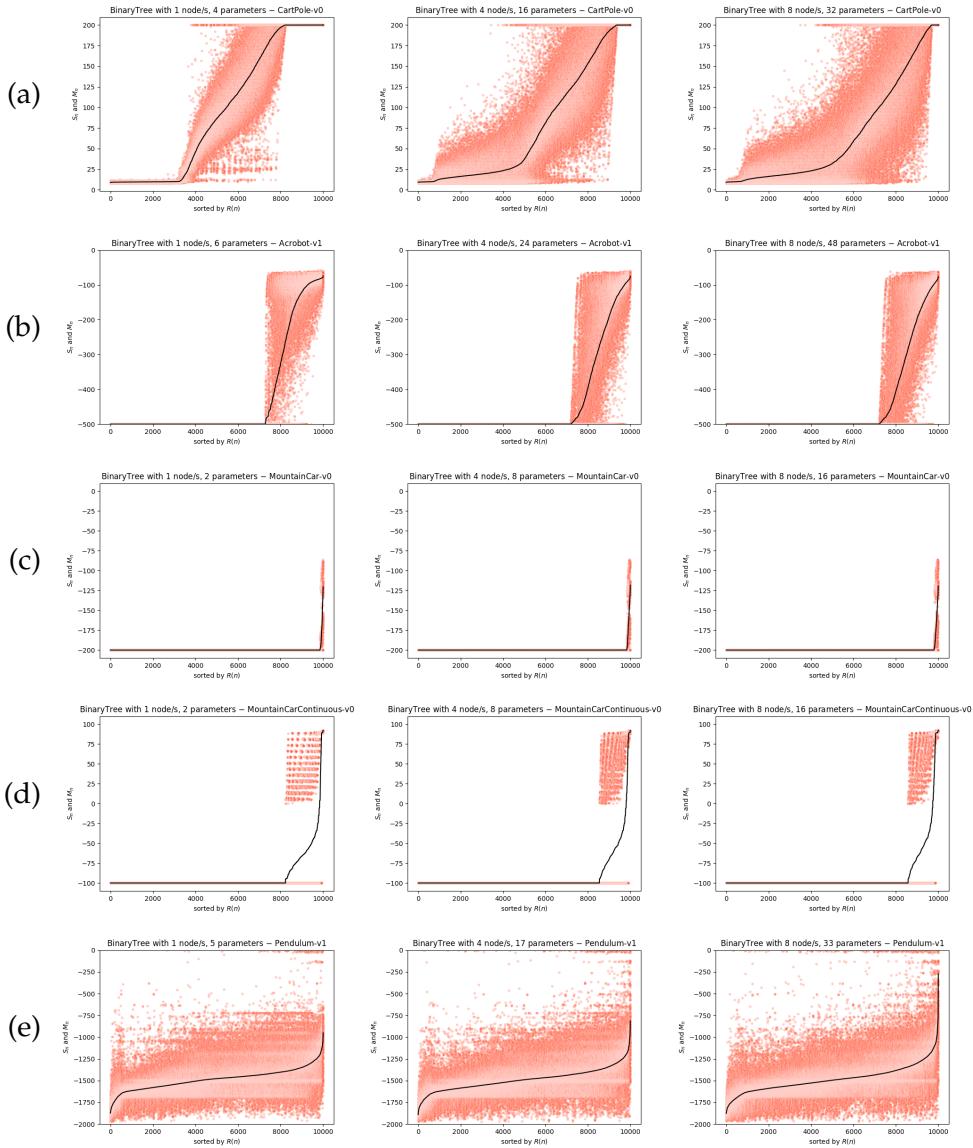


FIGURE 3.4: Results for the classic control environments using binary trees without bias. Each row shows the results of one environment. The columns represent the different configurations for the binary tree models. Even with only one node, the results are comparable to those from neural networks or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

fitness plateau remains. In addition, the scores are spread wider for the binary tree than for the neural network.

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 for the mean scores. The binary trees have a lot of low performers. However, there are a few samples that are able to reach a positive score. The neural network without hidden layers could not achieve these high mean scores. The considerable difference between the two models could be caused by the binary tree still 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.

For the Pendulum environment, we can see that the binary trees perform increasingly better with more nodes. The difference between the two models, neural network and binary tree, is not huge. However, we can see that using neural networks results in a few higher mean scores. But with the binary trees, we have a slightly higher score in the middle area. Additionally, the binary trees show a higher variance than the neural networks. 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 produced by neural networks.

Explain how model handled continuous environments in models section.

3.2.4 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 3.5 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 slightly worse with the increased complexity of the model. Interestingly, for the environment Acrobot, the network without hidden layers is resistant 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 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.

To further investigate this behavior, Figure 3.6 shows the results of tripling the number of weights for each network architecture for all classic control environments. Comparing these plots to those in Figure 3.1, we can barely see a difference. The slope of the mean values looks the same. The data points do not show different behavior. Thus, we can conclude that increasing the number of weights does not impact the achieved score of the model significantly.

Figure 3.7 shows the results of altering the number of layers in a network.

Figure 3.8 shows the results of increasing the number of neurons in each hidden layer for a network with two hidden layers. Comparing the plots with the ones in Figure 3.1, the results do not show much difference. The slope is a little less steep with increased neurons.

Restructuring, adjust text, exchange plots. Does it make sense to do all envs. in this experiments?

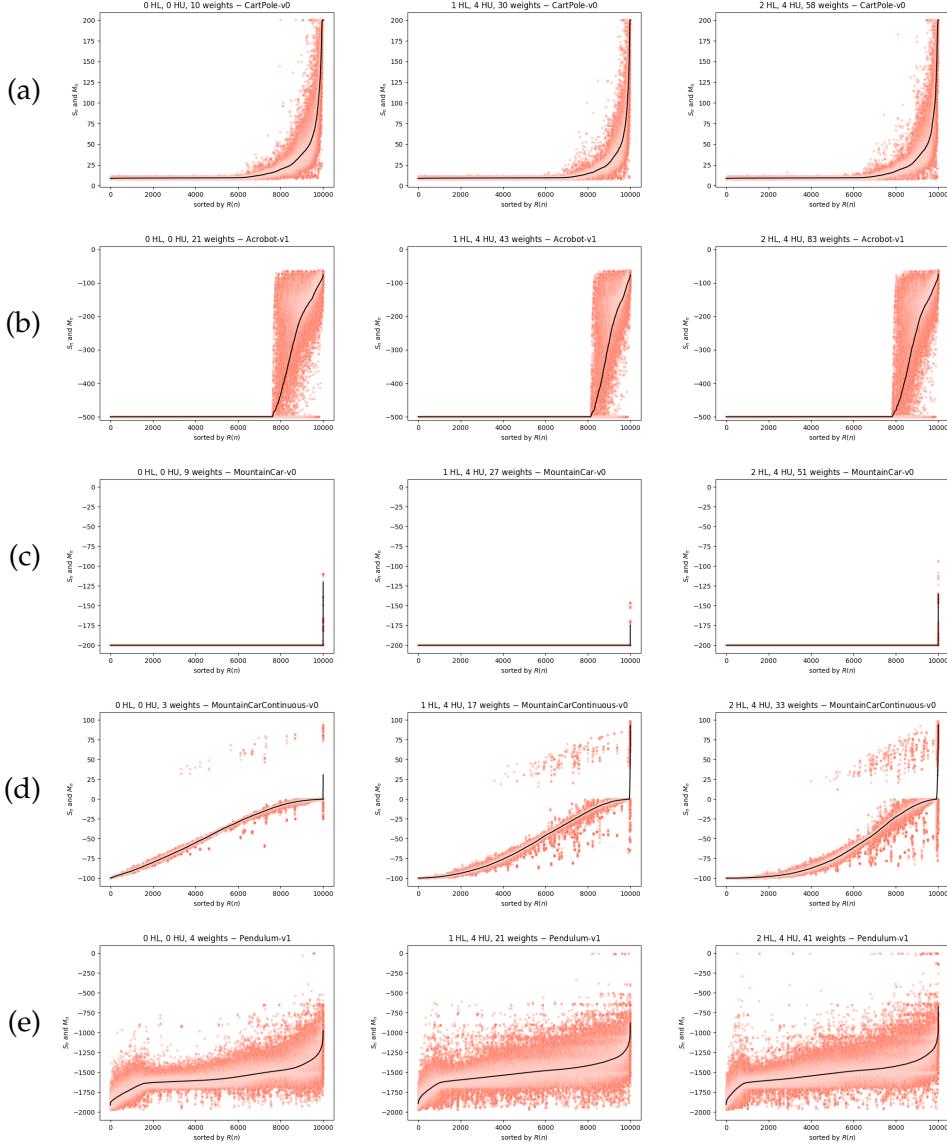


FIGURE 3.5: Results for all classic control environments using neural networks with bias. Each row shows the results of one environment, whereas the columns represent the different network architectures. For most environments, we can see that introducing a bias has a negative effect on the score distribution.

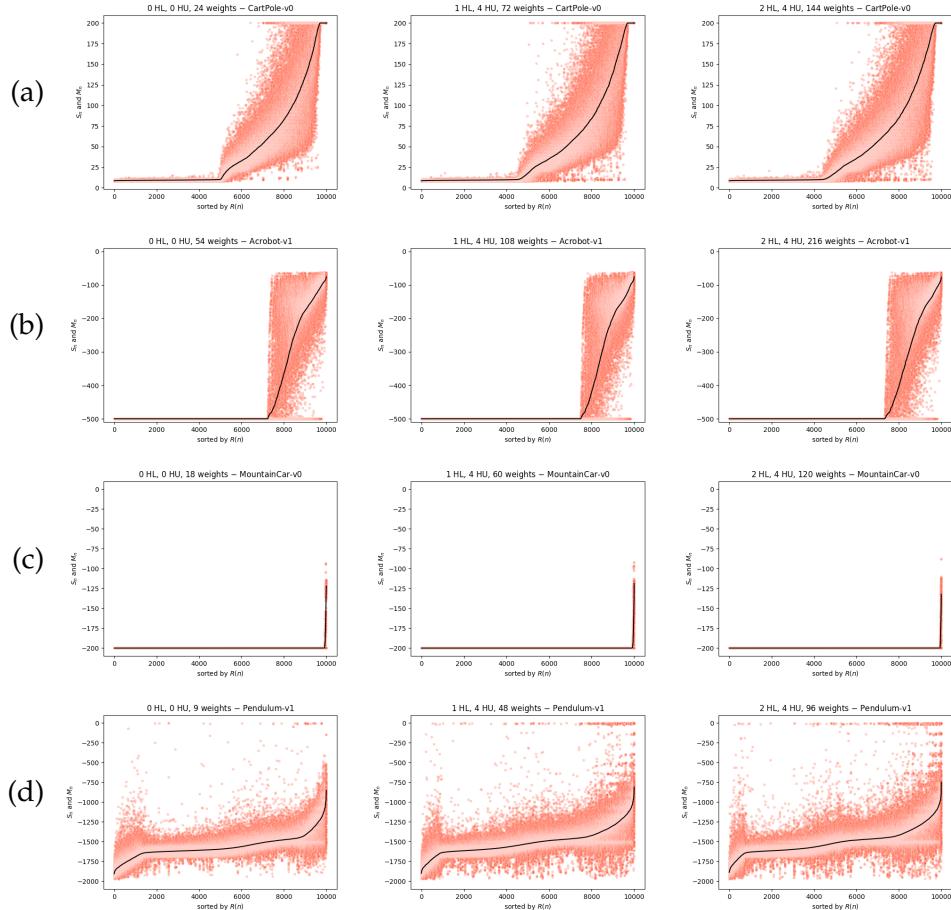


FIGURE 3.6: Results of tripling the number of weights for neural networks. The figure shows the results of tripling the number of weights for the three neural network architectures for the classic control environments. Despite the increased number of weights, there is no visible difference in the plots compared to the ones shown in Figure 3.1.

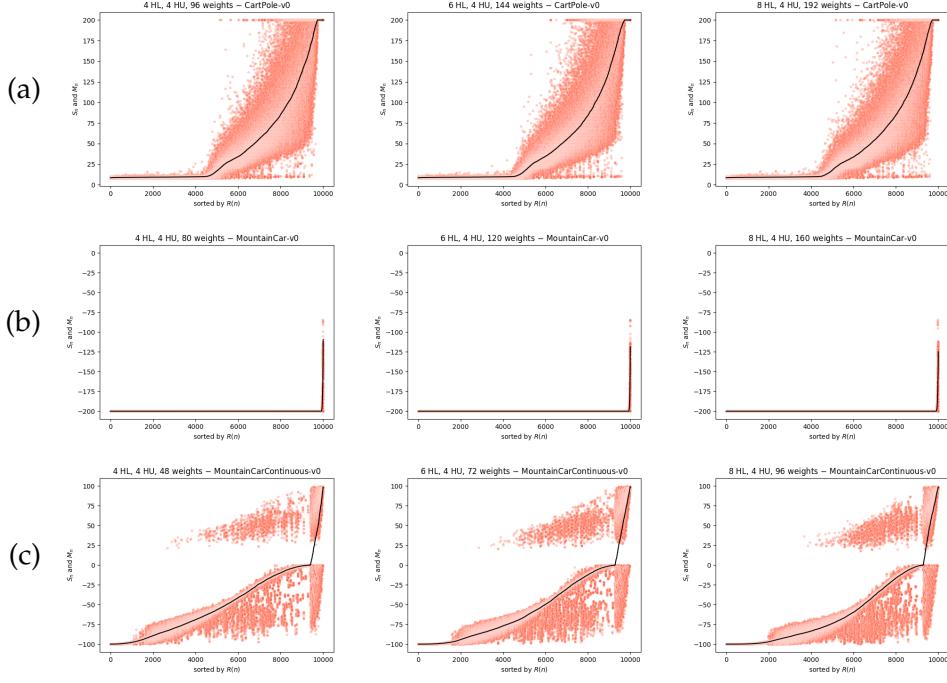


FIGURE 3.7: Results of altering the number of layers for neural networks. The plots show the result of increasing the number of hidden layers gradually.

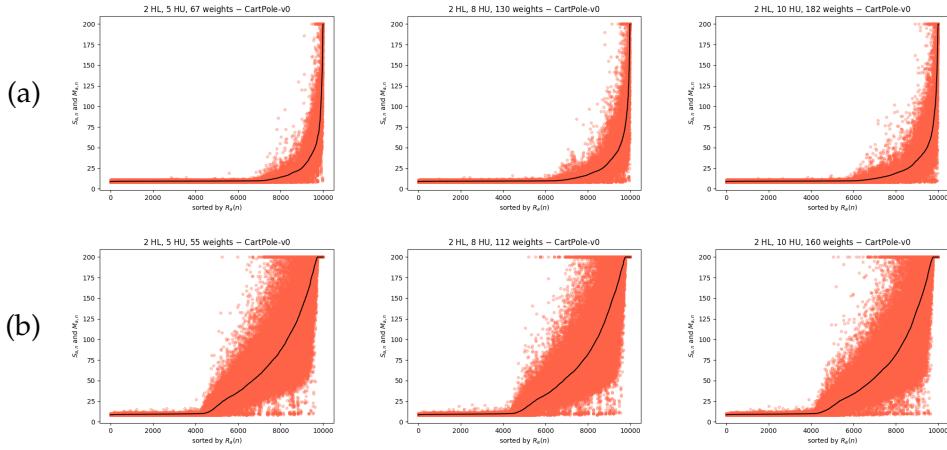


FIGURE 3.8: Results of experiment 2: neurons. The plots show the result of increasing the number of neurons for each layer in a network with two hidden layers. The plots in row (a) represent the networks with bias connections, the ones in row (b) represent the networks without bias connections. We can see a slight improvement in row (a).

3.3 Discussion

This chapter delivered a direct comparison between neural networks and alternative models like polynomials and binary trees. In addition, I investigated the impact of bias in these settings.

Comparison of polynomial model with neural networks: The polynomial model can definitely deliver comparable results to neural networks for the 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 with a higher complexity of the model, neural networks outperform the polynomial model. Nonetheless, for the CartPole environment, the plots differ from neural networks. The mean scores show the same trend for all neural network architectures. This 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 that with the linear model or neural networks, we have a larger fraction of samples that can solve the environment independently of the initialization conditions. At first glance, 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. However, 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. Considering this aspect, when using a learning algorithm, we would not prefer the polynomials with degree 1 as we might get stuck in a fitness plateau when the algorithm has no method of dealing with this behavior.

Comparison of binary tree model with neural networks: Despite the simple structure of the binary tree model, it could deliver comparable results or even outperform neural networks. The binary tree model outperformed neural networks for the environments CartPole and MountainCar. The results are similar for the two models for the environment Acrobot. 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. For the environment MountainCarContinuous, neural networks delivered better results.

Impact of bias: No other aspect impacted the performance of the model as much negatively as using bias. It seems that this behavior is specific to the bias in the setting of RWG.

Explain more.

Chapter 4

Conclusion

4.1 Conclusion

- Summarize the thesis, this time with a focus on the results obtained
- Add what deductions you have from the discussion above

In this work, I delivered a direct comparison of neural networks to the polynomial and binary tree models. I was able to show that especially the binary tree model can deliver comparable results with a simpler structure and less needed parameters. The last aspect makes the process of parameter tuning straightforward.

Explain more. Answer research questions.

4.2 Future Work

- Future work
- Fourier? You should add your own ideas rather than mine :)
- Particularly, please do not mention Bezier yet :) that was in confidence
- Harder benchmarks
- Ways to complexify the function from the b-tree

The continuation of this work includes...

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](https://doi.org/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](https://doi.org/10.1109/TSMC.1983.6313077).
- 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.
- Hansen, Nikolaus (2016). "The CMA evolution strategy: A tutorial". In: *arXiv preprint arXiv:1604.00772*.
- 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.
- 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](https://arxiv.org/abs/2004.07707). URL: <http://arxiv.org/abs/2004.07707> (visited on 12/07/2021).
- 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.
- 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. URL: <https://proceedings.neurips.cc/paper/1995/file/8f1d43620bc6bb580df6e80b0dc05c4Paper.pdf>.
- 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.

Chapter 5

Appendix

5.1 Additional Plots