Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Adaptive Goal-Directed Behavior using Planning with Reinforcement Rewards

Bachelorthesis
at Research Group Knowledge Technology, WTM
Prof. Dr. Stefan Wermter

Department of Informatics
MIN-Faculty
Universität Hamburg

submitted by
Joshua Schimmelpfennig
Course of study: Human-Computer-Interaction
Matrikelnr.: 6813643
on
11.06.2019

Examiners: Dr. Cornelius Weber
Dr. Sven Magg

Abstract

This thesis explores look-ahead planning in a continuous, open-horizon task aimed towards reinforcement learning algorithms. Look-ahead planning optimization uses a world-model to predict future states given a sequence of actions and uses reinforcements from those states to optimize the actions with gradient descent. The goal of this thesis is to show the capabilities of the algorithm for reinforcement learning, represented by the pendulum swing-up task. Our results show that the current algorithm is not capable of solving the swing-up task reliably. However, it can hold the pendulum once it is upright. Further, we found that the algorithm is not suited for agents with constrained action-spaces, except when the goal can be reached directly.

Zusammenfassung

Diese Arbeit untersucht den Planungsalgorithmus "look-ahead planning" in einer offenen Aufgabe im kontinuierlichen Raum, die auf Reinforcement Learning Algorithmen ausgerichtet ist. Die Planoptimierung verwendet ein Weltmodell, um mit dem Plan, einer Sequenz an Aktionen, zuknftige Zustnde vorherzusagen und diese zu nutzen, um Reinforcements berechnet. Die Reinforcements werden benutzt, um die Aktionen mit Gradient Descent zu optimieren. Das Ziel dieser Arbeit ist es, die Eignung dieses Planungsalgorithmuses fr Reinforcement Learning Probleme zu zeigen, reprsentiert durch eine Pendulum Swing-Up Aufgabe. Unsere Ergebnisse zeigen, dass der aktuelle Algorithmus nicht in der Lage ist, die Pendulum Swing-Up Aufgabe zuverlssig zu lsen. Der Algorithmus kann das Pendel jedoch halten, wenn dieses bereits aufrecht steht. Weiterhin haben wir festgestellt, dass der Algorithmus nicht fr Agenten mit eingeschrnkten Handlungsrumen geeignet ist, auer wenn das Ziel direkt erreicht werden kann.

Contents

List of Figures

Chapter 1

Introduction

Programming a robot to walk is a difficult task and can benefit from machine learning algorithms, in which agents learn to interact with their environment and to generalize from specific solutions, as seen in the stanford littledog research by Zico Kolter and Ng [13]. Hess et al. [5] show a very promising algorithm, with which humanoid agents can learn to traverse difficult terrain and even jump over obstacles. This reinforcement learning (RL) algorithm uses model networks to map the environment's observable states onto motor controls of the agent, acting in the environment. This entails a reinforcement, prompting the model to adapt the network and thus optimize the reinforcement. This causes the network to learn a model of the world and achieve the goal.

The drawback of model-free reinforcement learning is that agents have to execute an action first in order to adapt their action policy, making costly trial-and-error runs necessary. If agents had the ability to expect the results of their actions and refine them before acting in the environment, they could improve their learning performance and allow them to navigate through previously unseen territory while inferring knowledge from previously seen actions. This behavior is adaptive since the agent's trajectory can be altered if the agent receives new information, for example, visual input about an obstacle the agent is about to collide with.

Planning can achieve this, allowing the agent to plot its actions into the future and use the anticipated world states to modify its plan regarding the goal [11]. In its essence, planning is a search in state or plan state [10] allowing the agent to optimize its path through the scenarios state and action space. Planning needs to train a representation of the world in order to infer knowledge from it. Once a world model is trained, it can be used to generalize to states the agent has not yet seen [7].

A candidate for this task is look-ahead planning by Thrun et al. [11]. Here, an agent imagines how its action will impact future states and use that knowledge to tweak this plan, using reinforcement function as a value of those states. This work shows that an agent controlling a virtual robot arm can successfully intercept a ball rolling parallel to it, requiring the agent to move the arm in a trajectory perpendicular to the trajectory of the ball.

The validity of this method is shown by Otte et al. [6], where they use a similar

approach to solve a locomotion task in 2D-space, even allowing the agent to plot trajectories around concave objects.

Zhou et al. [12] show a modern implementation of this algorithm, using it in a combined algorithm approach to reach a given goal state.

In this thesis, we want to further explore this algorithm. We are especially interested in the performance of the algorithm in the domain of continuous, open-horizon tasks since in a potential robot application, states in the real world are continuous and the agent should not stop walking and collapse once it reaches a possible goal state. We aim to show the potential of this algorithm by exploring and testing an implementation of the plan optimization for a representative reinforcement learning task, showing that the agent is capable of solving this task and further research into it is warranted.

Chapter 2

Related Work

The concept of planning allows an agent to infer knowledge from the future state it predicts. Thrun et al. [11] use a planning model termed "look-ahead planning" to let a robot arm intercept a rolling ball, this work is the basis for this thesis. Alongside planning, the prediction model used contains an experience and a world model. The world model predicts, given a state and an action, future states necessary for the planning, while the experience model learns the results of the plan optimization itself, thus being able to predict optimal steps in the future. For planning, a plan optimizer refines a proposed sequence of actions (the plan) progressively, using the world model to predict future states as the effects of the actions and modify the sequence with gradient descent in action space. It was shown that the planning algorithm, referred to for the remainder of this thesis as "look-ahead planning", is able to successfully solve the robot reaching task. Additionally, look-ahead planning performed remarkably on an open horizon pole-balancing task, where the agent was able to hold the pole up for 20.000 steps. Though the pole-balancing task has an open horizon, it only has a binary action space. Further, planning is most interesting in cases where the goal is not directly reachable, for example in the mountain car task.

Using the same concept as look-ahead planning, Otte et al. [6] show a more recent but similar approach. They use LSTM-like recurrent neural networks (RNNs) for online inference. Using the RNNs temporal context, the algorithm projected the current state into the future, calculating the difference between the predicted and the desired goal state. This difference is in turn optimized using back-propagation through time with respect to the actions. This allows an agent to plot paths around concave objects and successfully reaching the goal state. They also showed the capability of this algorithm for long-term inference by combining short- and long-term planning due to long-term planning being too imprecise on its own. However, the algorithm was not applied to an open-horizon task.

A recent implementation of look-ahead planning can be found in Zhou et al. [12], where it is used in a hybrid planning strategy. Look-ahead planning is used on a global scale to generate trajectories, moving a robotic agent into the proximity of a goal state. This planning algorithm is only used on a global scale because model-based forward planning accumulates errors while predicting, especially if doing

so for long plans. The world model used for the planning algorithm predicted the activation of biologically inspired place and head-direction cells. In this cell space, the planning was executed. Compensating for the error of the look-ahead planning, they include a second planning strategy for short ranges, comparable to the ensemble approach in [6].

The second system uses visual recognition to reach a target with finer accuracy when the scale requires higher precision. The experiments compare look-ahead planning with a fixed and dynamic plan length. In the former strategy, the last state resulting from the plan execution is used for optimization, while the dynamic plan length strategy dynamically increases the plan length by chaining the fixed length strategy.

The chaining is continued until the distance between resulting and desired goal state increases again, meaning that the agent overshot the target (and especially for the coarser strategy, having reached the point at which to switch to the finder strategy). They conclude that a plan with fixed length is harder to optimize if initialized badly, due to vanishing gradient from only using the last state after plan execution. This was not the case with the dynamic plan length since all previous actions have been optimized in a previous iteration, thus only needing slight adjustments. They also discover that the planning performance deteriorates with plan length, possibly due to the accumulation error in the chain of world models. Though the algorithm was not implemented in an open-horizon task, it shows that it works for this use case.

Further examining the possibilities of world models, which is an important component in look-ahead planning, Ha and Schmidhuber [4] use a world model in combination with a compact policy to solve locomotion tasks. The world model used consists of an RNN combined with a mixture density network. It is a distributional world model and computes a probability vector for the next state given the current state, the planned action and the hidden state of the RNN. This network allowed for a compact controller consisting of a single-layer linear model, mapping the environment's current state and the world model's hidden state to an action.

The goal of the agent was to optimize its expected cumulative reward during a single rollout. Additionally, the world model was trained to predict state changes independently of the controller. Thus the compact controller could be trained with a more exotic approach, in this case, evolution strategies. Combined with an autoencoder, this setup successfully solved OpenAI's car racing environment and appears to be the first architecture to do so. Concentrating most of the agent's complexity inside a differentiable world model also allows for common GPU acceleration, which is also important for look-ahead planning, as here the plan optimization works online.

In the locomotion domain, Heess et al. [5] use deep reinforcement learning with distributed policy gradient optimization to learn a complex and robust locomotion behavior. Learning with a rich environment yet a simple reward function, they show that a walker is able to traverse diverse environments. The simple reward function based on forward progress was specifically used to encourage the emergence of robust locomotion behavior learned from the rich environment, thus giving an

agent the ability to traverse more general and complex environments. This also reduces the possibility of accidentally overfitting the reward function to the task, which would cause an algorithm to become brittle. The reward function depends on the speed, distance traveled and whether the torso of the walker was upright. The reinforcement function that this thesis uses, is an example for future locomotion reinforcement function and could be used when testing look-ahead planning in the walking domain.

In comparison to the experience network used in look-ahead planning, Pascanu et al. [7] introduce an alternative approach to plan creation. Here, an agent can imagine the resulting state and reward following a variety of actions in consecutive iterations. Since the states are saved in a history, the agent can then decide to continue exploring a certain path, creating a tree of states, results and rewards. This tree can then be used to choose an optimal path, by selecting a desired state in the path and using the path between it and the root, the current state in the world, as the plan. This construction technique can be adapted to look-ahead planning.

Silver et al. [9] use a single, differentiable architecture for planning. This architecture consists of a deep neural network with a Markov reward process as the recurrent core and rolls it out multiple times to make the predictions. Interestingly, the model was not handcrafted to use the actual representation, but can instead have its own representation. Using a different representation is comparable to Zhou et al. [12] however, they used a specific representation of biological cells, where the model can take on any possible form. This is possible because the authors integrated both training the model and planning with regards to the overarching objective of the agent. The only constraint for the model is that the trajectories found through the model produce results consistent with the agent's environment. It can be used to further improve look-ahead planning in the future.

In summary, we will explore and test look-ahead planning by Thrun et al. [11] in an open-horizon task. The original implementation already shows that the algorithm works for an open-horizon task, but not for a continuous action space and a planning task with a not directly accessible goal state. Otte et al. [6] and Zhou et al. [12] works use comparable approaches or look-ahead planning explicitly and show that this approach should work. Yet, they did not show this for a task with an open horizon. Ha and Schmidhuber [4] show the capabilities of the world model and that it is possible to focus computation in it. Heess et al. [5] show agents learning difficult locomotion tasks. The reinforcement function used here can be adapted to be used with look-ahead planning. Further potential algorithm optimizations can be seen in Pascanu et al. [7] and Silver et al. [9], showing different approaches to plan and world model creation respectively.

Chapter 3

Review of Current Research

3.1 World Model

As mentioned in chapter 2, the world model is one of the main components of look-ahead planning and necessary for planning generally [11]. A world model is an approximation of the environment that allows an agent to predict how the environment will respond to its action. Given a state and an action, the world model returns predictions about the impact of the action on the state. In a distribution model, the model predicts probabilities for all states given an action, wherein a sample model, the world model predicts a single state [10]. Depending on the context, the distribution model can be more useful in a discreet space and the sample model in a continuous one. Importantly, a distribution model can always be used to sample a single state but will be more difficult to train. For example, a distribution model works well in a continuous state space when combined with an autoencoder [4].

In our case, feedforward multilayer perceptrons are sufficient. If temporal effects should be considered, a recurrent neural network is needed [11].

3.2 Planning

3.2.1 General Planning

Planning describes the process of developing and executing a plan. From a given initial state, an agent tries to reach a goal by executing a list of actions in its plan. During planning, an agent has to consider its position in the environment, the range of actions it can perform and ultimately the goal it aims to reach. For example, a deterministic forward planner with discreet actions and states could treat this problem as a search in a state-space graph, where the goal would be represented by a goal state to reach. This process can be modeled as a stationary (time-independent) Markov decision process. More on classical planning can be found in Poole and Mackworth's Artifical Intelligence. [3]

3.2.2 Look-Ahead Planning

This thesis focuses on look-ahead planning introduced by Thrun et. al. [11], therefore the algorithm is summarized below.

In look-ahead planning, plan optimization modifies actions with respect to reinforcements the agent expects to receive after each step through the plan. Reinforcements are explained in Section 3.3.

The plan optimization is performed by using gradient descent on reinforcements the agent imagines to receive. The agent does this by executing the plan in its mind, using its knowledge of the world to predict what states and reinforcements will result from a given initial state and plan. The agent then executes the first action of the plan to act in the world. This process is repeated for each step the agent takes.

Generally, look-ahead planning can be separated into two phases. In the first phase, a new plan is generated using a heuristic or inference on an experience network. During planning, a good heuristic is to use the plan constructed for the previous step as a basis for the next one, disregarding the action executed in the previous step. Doing this, a new action needs to be added to the plan, which is determined by said heuristics or the experience network. In the second phase, the plan is optimized iteratively using gradient descent in action space.

The experience model is an additional network trained during planning, which learns to predict the planning results. The model learns a strong policy with regards to the current state and future reinforcements, allowing it to predict optimal actions needing minimal adjustments. The experience model is not in the scope of this thesis and will not be used. Instead, a heuristic introduced later in chapter 4, will be implemented.

In order for an agent to achieve planning, a world model (see section 3.1) is required [11]. Depending on the task, the world model can be recurrent, while its task is to approximate the world the agent acts in. The model learns to predict future states and reinforcements, given a state and an action and is trained by comparing its predictions to actual states and reinforcements, generated by the agent acting in the environment.

To do the planning, an agent feeds an action into its world model and receive a prediction of the resulting state and reinforcement. The resulting state can be combined with another action and again, be used to predict yet another state and reinforcement. Here, the prediction of future states given a sequence of action, or a plan, is referred to as unrolling the plan with the world model. Doing so while iterating over a plan will build a chain of states/reinforcements and world models, generating a utility measure for each action in each state. These reinforcements are reduced by a differentiable function into a single measurement, E_{reinf}, measuring the difference between the predicted and desired reinforcements. E_{reinf} is defined as follows:

$$E_{reinf} \equiv \frac{1}{2} \sum_{r=1}^{N} \sum_{k} g_k(\tau) \cdot (\text{reinf}'_k - \text{activation}_k(\tau))^2, \tag{3.1}$$

where N is the number of actions, τ is the current time step, k is the number of reinforcement output units of the world model, reinf'_k is the desired reinforcement and g_k the weight of the reinforcement with respect to τ and k [11]. This transforms the task of optimizing a plan into a task of minimizing E_{reinf}.

Since the world model, states and the reduction function are differentiable, gradients of the actions with regards to E_{reinf} can be calculated and used to optimize the plan iteratively using gradient descent. As shown in the original work, the computation of the gradients can be achieved with a feed-forward algorithm, or alternatively with backpropagation. With these gradients, the plan can be improved either until convergence or for a predetermined amount of iterations. This whole process is repeated for each step.

To summarise this section, look-ahead planning works by the agent imagining the results of its planned actions using a world model and iteratively improving those actions by applying gradients of the actions with regard to the expected reinforcement to those actions.

3.3 Reinforcements

Reinforcements here are adapted from the reward signal in reinforcement learning. The reward is a numerical value that the agent receives for each timestep. In reinforcement learning, it is the agent's goal to maximize this value. This turns the reward into a measure of the quality of the state the agent is in and can be compared to pleasure and pain in a biological system. Thus, the reward maps the problem that the agent is tasked with solving onto a numerical value, becoming the problems feature that the agents see. Further, the reward becomes the basis for changing action policies if the reward is lower than expected, causing the agent to learn a policy which solves the task. [10]

Chapter 4

Approach

4.1 Research Objective

As mentioned in Chapter 2, reinforcement learning is very successful in the realm of locomotion. However, without planning the agent can only react to stimuli he receives at the moment. This might lead it into a local minima or causes it to choose paths with high initial but poor reinforcements later on, though better options are available. Being able to see into the future could allow an agent to better approximate the value function, even if it is not explicitly present in the environment. We used the look-ahead planning algorithm by Thrun et al., introduced and explained in Chapter 2 and 3.2.2. In short, look-ahead planning chains world-models and their predictions together and, given an initial state and a list of actions, creates a list of resulting states and reinforcements. While the predicted states are used to predict the next states, the reinforcements are aggregated into reinforcement energy using Equation (4.1). Here, reinforcement energy is the total reinforcement the agent expects to receive when executing the plan. The actions can then be optimized with regards to this reinforcement energy, creating an optimal plan.

Look-ahead planning was shown to work in continuous state spaces with finite tasks, as shown in the original implementation [11] and in newer approaches, as seen in [12] or [6]. However, the abilities of look-ahead planning have not yet been shown in a continuous open-ended task, in which the agent never truly reaches a goal state. Due to the task structure, a continuous control task, where the agent holds a joint at a desired angle, look-ahead planning should perform well in it. Thus the research question arises, whether Thrun's look-ahead planning works in navigating through continuous, open-ended tasks. To show the validity of the algorithm, we aim to solve a representative reinforcement learning task, namely the classical swing-up task. This will show that, in principle, the algorithm can solve reinforcement learning task and that future research into this topic is worthwhile.

To show that the algorithm can achieve the task, it will be separated into two sub-tasks. If an agent cannot hold the controlled object in its desired state, it will never be able to perform well in the whole task, even if it can manage to swing the pendulum up. It has to be shown first, that look-ahead planning can hold the

pendulum up for a certain amount of time. Secondly, the agent needs to swing the pendulum up.

If both tasks can be fulfilled, it can be assumed that look-ahead planning is a good candidate for continuous control tasks and should be researched further.

4.2 Frameworks

This section describes the two main frameworks used for this thesis, namely Google's TensorFlow with Eager Execution and OpenAi's Gym.

4.2.1 TensorFlow Eager Execution

TensorFlow [1] is a framework for machine learning which allows users to describe a model by defining a graph. This model can then be trained using a variety of built-in functions, optimizer, loss functions, metrics to be logged and more. This can all be done in the high-end API Keras, which allows creating models in a pythonic approach by using objects. Alternatively, the functionalities of TensorFlow allow the user to customize the lower-level functions of this framework. Recently, an eager execution mode was introduced to this framework.

Generally, eager execution refers to the execution order used by a programming language. Eagerly executing programming paradigms evaluate an expression when it is presented to the interpreter, which is in contrast to lazy execution, where statements are only executed once their value is requested by the program. Intended as another high-end API, eager execution basically turned TensorFlow into a general math framework. For example, TensorFlow with eager execution can be used just as one would use NumPy, only with the use of tensors. Additionally, TensorFlow has integrated GPU-acceleration, meaning Tensor calculations can be offloaded to ones GPU. Though primarily intended as a high-end API, TensorFlow eager execution allows the user access to all the functionality needed to do machine learning from scratch, making it excellent for researching novel approaches and testing unconventional learning paradigms. What ties eager execution together is the implementation of the auto-differentiation functionality for variables, the main feature of standard TensorFlow. In eager execution, auto-differentiation is modeled in the GradientTape object. While active, the GradientTape will write every function applied to a predetermined variable to the tape, allowing it to reverse the functions when calculating the gradients. Partial derivatives can be calculated by defining the derivation of a variable with respect to another, for example, the derivative of model weights with respect to the loss value.

Keras is an object-oriented wrapper for the graph definition of TensorFlow. Using Keras, one can easily define a sequential network simply by calling a model object with functions, which will add another layer to the network. If one would want a more sophisticated model, Keras has a fully functional API for that as well. Importantly, Keras is available in eager execution, meaning a model can be

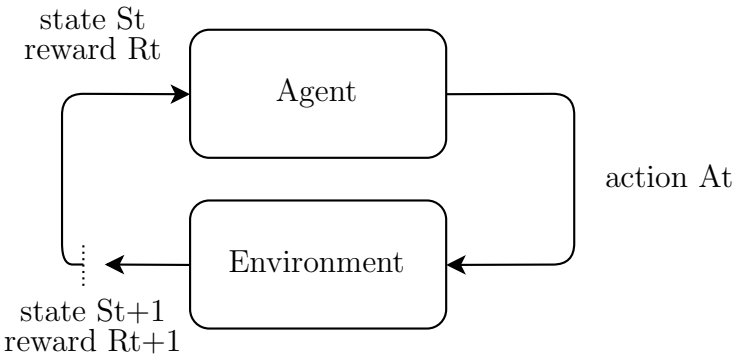


Figure 4.1: The agent-environment interaction in Gym

easily written using Keras simple model structure and then be trained using eager execution methods.

4.2.2 OpenAI Gym

Gym [2] is an environment created to train and compare reinforcement learning algorithms. The idea behind Gym is to develop a universal interface for different problems an algorithm might face, allowing for the environments defined in Gym to be easily interchangeable and extended by users.

The agent-environment interaction is the same as in a classic Markov decision process [10] and can be seen in Figure 4.1. In it, an agent sends the action it wants to execute based on the state and its policy, to the environment. The environment, in turn, answers with the resulting state of the action, the reinforcement the agent receives for that action. Additionally, in Gym, the environment also sends a boolean flag whether the task was solved, in the environment used this has no effect. This forms a cycle of action and reward, allowing reinforcement learning agents to learn the reinforcement function with time. One initialization and consecutive actions in the environment will be referred to as a rollout. Lastly, OpenAI Gym offers a leaderboard where users can compare their algorithms and are encouraged to add write ups or their GitHub repositories for their implementation, so that other users might peer review them, promoting the idea of open source and shared learning.

Interesting for a planning agent is the environment, in which a solution to the problem is counter-intuitive. One such environment is the pendulum task, in which an agent must swing and hold a frictionless pendulum up. An example of this pendulum can be seen in Figure 4.2. The difficulty is, that the pendulum cannot be swung up simply by following the reinforcement curve. It is not possible for the agent to swing the pendulum up in a single push, the pendulum needs to build up speed before it can be swung up. To solve this task, an agent needs to accept a high initial loss in order to receive optimal reinforcement later on. Simply following the reinforcement function would instead lead the agent into a local optimum, in which the gravitational force and force of the agent would be in equilibrium, but the pendulum could not move up any further.

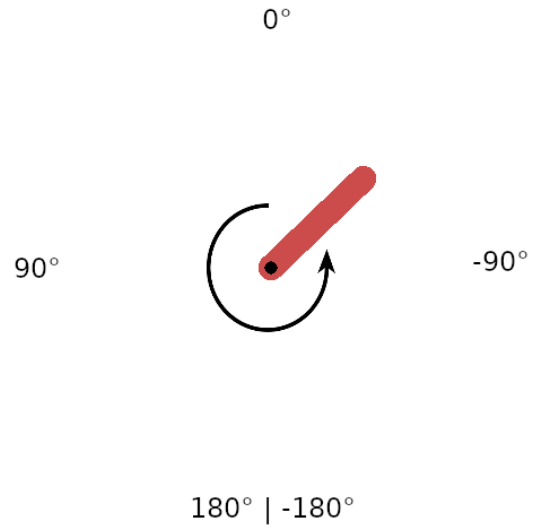


Figure 4.2: Example of the Pendulum environment with angles for θ. As seen from 0°, θ becomes smaller if the pendulum turns counter-clockwise and bigger if the pendulum turns clockwise. θ is 360° if the pendulum does a full counter-clockwise rotation, starting form 0°.

4.3 Implementation

This section describes how we implemented look-ahead planning, what challenges we faced and how those where solved. We will give a broad overview of this project first so that readers familiar with the algorithm can skip to the next section. After that, the main components are introduced, after which general concepts and difficulties the implementation needs to solve are explained.

4.3.1 Overview

The algorithm implemented consists of three major parts, the environment, the world model and the plan optimizer. The basic interaction between them can be seen in Figure 4.3, where a snapshot during a rollout can be seen. The environment gives an agent feedback for its action. Given an action, it returns the next state from which a reinforcement can be derived. Using this new state, the agent calculates the trajectory it wants to take using the plan optimizer with the world model and returns the next action that should be executed in the environment. This cycle will be repeated for the current rollout, limited only by time or steps since the environment can never actually be solved. The agent is separated into two parts, the plan optimizer, and the world model. When the agent receives a new state, the world model is used to predict future state transitions according to the plan, basically imagining the reaction of the environment if it would execute the plan. The reaction, or resulting states, is used to calculate how the actions need to change to allow the plan to be optimal. This process is done in the plan optimizer

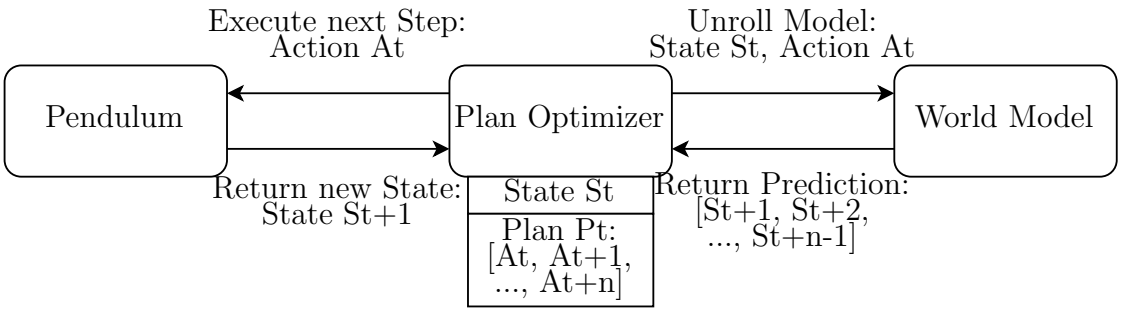


Figure 4.3: Basic interaction-cycle of the three major modules in our implementation.

using Gradient Descent on the plan. This allows the agent to iteratively improve the plan.

In this implementation, the experience network of the original model by Thrun et al. [11] is not used due to not being in the scope of this thesis. Furthermore, since we want to analyze the optimization of the plan optimizer, the world model does not predict the reinforcement, as is done in the original implementation. This is done to control for any noise the reinforcement prediction might create.

4.3.2 Pendulum

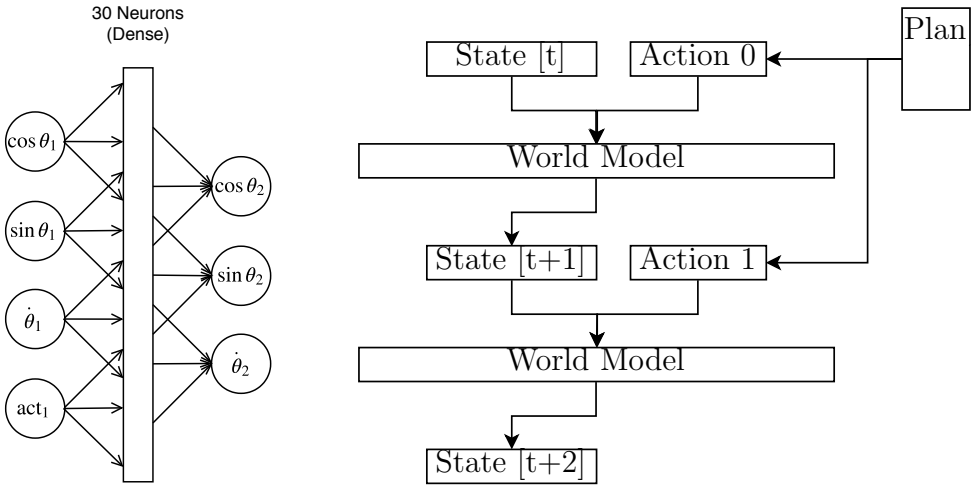
The Pendulum is an environment in Gym, which consists of a friction-less pendulum fixed to a single point. Figure 4.2 is an example of how it looks like. The agent's task is to hold the pendulum upright for as long as possible, as there is no determined goal state. The agent can apply torque to the pendulum, however, it can not swing the pendulum up by only pushing constantly into one direction. Instead, the pendulum would get stuck at the equilibrium point where the torque is no longer stronger than the gravitational pull.

Internally, the states of the environment are encoded as the angle theta θ of the pendulum and its speed thetadot ($\dot{\theta}$). Externally, the environment only sends the sine and cosine of θ, as well as $\dot{\theta}$ to the agent. This results in the following state: $(\cos\theta, \sin\theta, \dot{\theta})$

As seen in Figure 4.3, the project consists of three main components. Being responsible for communication with the Gym environment, the Pendulum module contains a class that serves as a front-end to it.

Generally, to keep results comparable, we did not want to directly manipulate states of the environment. However, to explicitly examine how the algorithm behaves in certain conditions, it is necessary to bridge the environment's random initialization and set θ or $\dot{\theta}$ explicitly. The class mentioned previously can set and update states in the environment as well as return the internal values of the environment, making analyses of the behavior of the algorithm less cumbersome.

The Pendulum module also contains some helper functions, like initialization functions for single actions or whole plans. Another function can execute a plan in the environment and return the visited states. Lastly, this module contains a



(a) The structure of the world model. The input layer takes the three attributes of the environment's state and an action to pass it through a dense layer with 30 neurons. The three output neurons return the values for the attributes of the next state.

(b) Example of the plan unrolling using the world model. Here, the plan has two actions which are fed sequentially through the world model using the current state prediction.

Figure 4.4: The architecture of the world model.

generator function for the training of the world model, yielding training data for a single rollout with a predetermined step length.

4.3.3 World Model

As mentioned in Chapter 2, planning needs a way of looking into the future, allowing the agent to have an expectation of how the environment will respond to the actions it executes. This expectation of future states can be modeled with a world model. Since the accuracy of the plan directly depends on the quality of the prediction, it is important that this model is accurate. Furthermore, since an error in a prediction will be kept and used for the next prediction, the world model will ultimately accumulate and carry the error of every consecutive prediction.

In our implementation, the world model was built using the sequential Keras API mentioned in Section 4.2.1. As seen in Figure 4.4a, the model has a single hidden dense layer with 30 neurons, with 4 input neurons for a state and action tuple and 3 output neurons for the next state prediction. Since the state attribute $\dot{\theta}$ already encodes temporal effects, no recurrent network is needed for this task.

The neurons use ReLU as an activation function in the hidden layer. This is chosen to try and combat the previously discovered effects of vanishing gradients. As will be mentioned later, the effect of vanishing gradients is connected to the plan length, resulting in a deeper chain of world models when unrolling the plan.

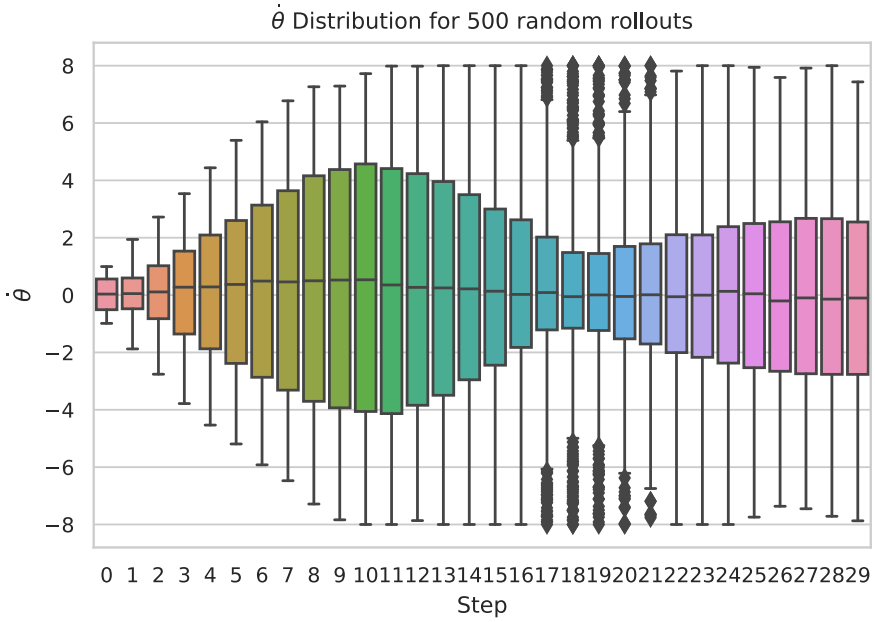


Figure 4.5: The distribution of $\dot{\theta}$ for 500 rollouts over 30 consecutive steps as boxplots. $\dot{\theta}$ is initialized randomly by the environment in [-1, 1] and its domain is [-8, 8].

For training, RMSE (root mean-squared-error [3]) was used as the error function. This is because the RMSE scales the values of MSE back into the normal state space. Scaling values back is useful, the MSE for attributes with domains in [0, 1] tends to be very small, due to the squaring In contrast, the MSE tends to become very large for values larger than 1. Since we do not scale our input, $\dot{\theta}$ would over-proportionally influence the error and thus the gradients, ignoring the error of sine and cosine.

The model was trained with backpropagation using the GradientTape and eager execution. Training data can be continuously generated using the generator function (see Section 4.3.2) of the pendulum module, supplying the world model with new data for every iteration. The training data returned by the trainer is a tuple of features and labels, where the feature is a single tuple of the state and action and the labels are the state as returned by the environment for that input. The data for an iteration is 15 consecutive steps of a single rollout, batched together into a single mini-batch.

Since the environment initializes $\dot{\theta}$ only between -1 and 1, a set of consecutive steps is needed to let the model see higher pendulum speeds. If the model never sees the behavior of the world at high speeds, it has to rely on generalizations made from lower speeds. Furthermore, the model never sees the boundaries of the domain of $\dot{\theta}$ and could, together with the generalization, predict values far outside of it.

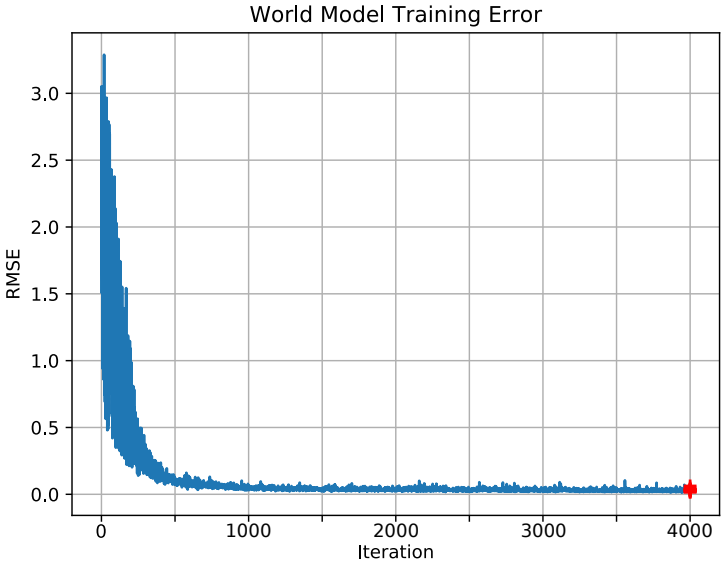


Figure 4.6: The RMSE values for the mini-batches the world model achieved during training. The training was done for 4000 iterations with 15 consecutive steps in the mini-batch each. 10 test errors were calculated after the iterations and marked with red crosses.

However, we do not want to have too many steps, since the probability with each step rises that gravity causes the pendulum to fall down and biases the model towards seeing the pendulum fall or swing at 180° (the bottom of the pendulum). As shown in Figure 4.5, 15 steps contain both high and low values for $\dot{\theta}$. The model was then trained with one rollout as a mini-batch, with the 15 consecutive steps as a mini-batch. This was done for 4000 rollouts, whereas the model usually converges within 1000 steps. The model was trained with the Adam optimizer, since adaptive optimizer account for sparse data and Adam is a generally well performing, adaptive optimizer [8]. The convergence of the models RMSE can be seen in Figure 4.6.

Figure 4.7 shows the predictive accuracy of the world model. In Figure 4.7a, the model predicted 100 states into the future from 500 random starting states, producing the error curves seen in the visualization. The ranges for the state attributes are discussed in Section 4.3.3. The cyan curve is the mean of the error curves. Noteworthy are the exploding errors when the model approaches predictions closer to the 100 step mark, where the model is likely to predict values outside of the state attributes' domains. As seen in Figure 4.7b, a 95% confidence interval is drawn around the mean of the RMSE curves, seen in the previous figure. The figure shows, that the prediction becomes inaccurate after 20 consecutive predictions into the future. However, Figure 4.7c shows how the true and predicted states actually change over consecutive steps. Here, the cyan curve is the RMSE of the prediction as compared to the truths. This figure shows that the $\dot{\theta}$ prediction

has, due to its domain, the biggest influence on the RMSE. Furthermore, it shows that the high RMSE, also seen in the previous two figures, is partly caused by an accumulated error in the swinging frequency, shifting the $\dot{\theta}$ curve to be faster.

4.3.4 Plan Creation

In the original implementation by Thrun et al. [11], new actions are proposed by an experience network, which learned to predict the results of planning. The experience network is not examined in this thesis, therefore a heuristic is used instead. After an action is expanded, a neutral action is added to the back of the plan. This action is the median of the action space, an action with value zero, equivalent of telling the agent to do nothing. In contrast to a random initialization, this function avoids the worst case of initializing an action whereas its optimal value is on the opposite side of the action space. With the zero function, in the worst case, actions only have to be optimized through half of the action space. Further, the initial plan, the plan with which the plan optimizer starts in a new rollout, also consists of zero actions.

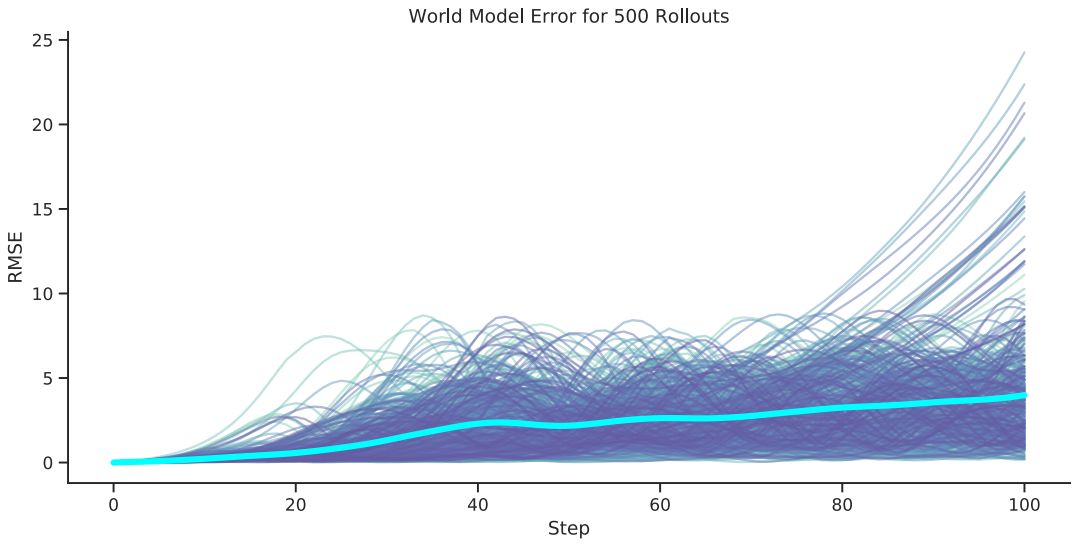
4.3.5 Plan Optimizer

The plan optimizer is the core of the system. We implemented it as an object which holds its current plan and can be updated with a new state, it then optimizes a new plan, with the previous plan as a heuristic for the next plan. After an action is taken, we add new actions with a value of zero, to the end of the plan. The action is at the median of the action space. We do this in order to keep the plan length the same. The environment interprets this as an action with no force. With an initial action of zero, the optimizer can tune it in either direction efficiently and suffers less from a bad initialization, as could be the case if random variables were used.

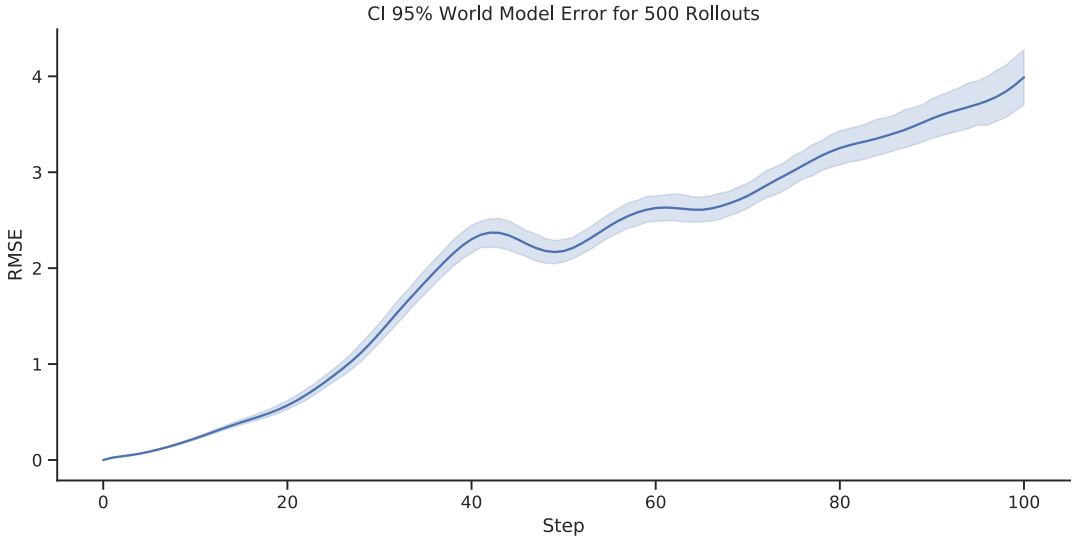
The process of optimizing the plan will be explained for a single iteration and can be seen as exemplary in Figure 4.4b, with a plan length of two. First, the plan is unrolled using the world model by combining the current state of the agent in the environment with the first action of the plan. The world model predicts the next state given those values. The state prediction is then combined with the next action of the plan and used as the next input of the world model.

This is done until all actions from the plan are used, creating a chain of world models and state-action combinations. Essentially, this combination of state predictions and actions as the next input for the world model turns this structure into a layer itself, with the identity function as the activation function. This creates a deep network for the prediction of the n-th environment state, with our actions as weights in the network.

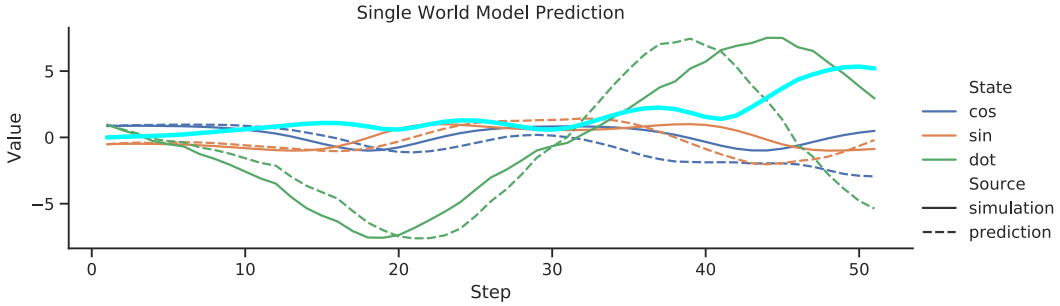
We have turned the task of optimizing a plan to a task of optimizing certain weights in a neural network. Using the reinforcement function from section 4.3.6, we calculate the reinforcement of each predicted state and combine them into a single reinforcement energy value E_{reinf}.



(a) The RMSE graphs for 500 rollouts with 100 steps world model prediction, with the mean RMSE in cyan.



(b) The 95% confidence interval of the RMSE for 500 rollouts with 100 steps world model prediction.



(c) The change in states of the true state (filled) and the world model prediction (dashed) for 50 time steps, starting from the same initial state. The RMSE is marked as a filled cyan curve.

Figure 4.7: World Model prediction accuracy.

However, here we do not use the original function for E_{reinf} (3.1). The reasons for this are, that we calculate the reinforcement from the state prediction and do not let the model predict it, thus we do not have any output units for the reinforcement. Furthermore, we do not weigh the reinforcement here, but let any weighting with regard to time happen inside the implemented optimizer if the particular optimizer implements it. The desired reinforcement in Equation (3.1) is in our case, as seen in section 4.3.6, zero. This leaves us with the following equation for E_{reinf}:

$$E_{reinf} \equiv \sum_{r=0}^{N} -(\text{reinforcement}(\text{prediction}(\text{act}_i))),$$

where N is the number of actions in the plan,act_i is the action at index i in the plan, prediction() the prediction function for an action, reinforcement() the reinforcement function for a given state. The result of the reinforcement function is negated, because our reinforcement function only returns losses (see section 4.3.6) and we need positive values to minimize with Gradient Descent.

Using TensorFlow eager execution's GradientTape introduced in section 4.2.1, we can now calculate all actions' gradients with regards to this reinforcement energy, giving us the changes we need to apply to our actions. Using the GradientDescentOptimizer implemented in TensorFlow, we update our actions with a learning rate of $\lambda = 0.5$, which shows the best convergence for the plan, as seen in section 4.3.8. Now, the plan needs to be unrolled again in the next iteration to further improve it.

When the optimization is done, the plan optimizer returns the next action of the plan, which can, in turn, be given to the environment to execute and return the next state.

4.3.6 Reinforcement Function

The reinforcement function is the core of reinforcement learning and in this environment a numerical number denoting the current value of the state the agent is in. The goal of the agent is to optimize the reinforcement it accumulates during a single rollout. An additional challenge in the pendulum environment is that the true states of the pendulum are hidden and only the sine, cosine and speed of theta θ are given to the agent. The original reinforcement is as follows:

$$-(\theta^2 + 0.1 \cdot (\dot{\theta})^2 + 0.001 \cdot a^2), \tag{4.1}$$

here θ is the hidden angle and a is the action taken. Calculating the reinforcement from the state instead of predicting it, makes the implementation more difficult. This is because the reinforcement function is needed to calculate the predicted reinforcement while planning, where no environment is present to extract the true values from. Additionally, θ cannot be inferred easily, since the world model could predict values not in the domain of cosine and sine, which is why inverse functions like arccos cannot be applied here without risking a crashing agent.

Because of this, we decided to use a heuristic to approximate the reinforcement function. In this case, the cosine prediction as a substitute for theta is the candidate of choice, as its extrema are aligned with the extrema of (4.1). The used reinforcement function has the form:

$$-((\cos(\theta)-1)^2+0.1\cdot(\dot{\theta})^2+0.001\cdot a^2). \tag{4.2}$$

There are two big drawbacks to this approach, which might reduce the performance of the algorithm. Firstly, in the original function, a slope towards the minimum exists, with a steep inner angle at the maximum, whereas the cosine has the form of a wave, with shallow gradients at the extrema. Secondly, if the world model would encode a looping rotation by multiplying the θ predictions with an integer factor (which could be syntactically correct), the cosine of θ is no longer in its domain and this influences our approximation of the reinforcement function dramatically. The optimum is morphed to the sides of the original minimum, creating a small indentation, starting from the optimum in the curve. This can cause the algorithm to optimize the actions towards angles next to, but not at $\theta = 0$. Thus, the only benefit of cosine over arccos is, that the program does not crash fatally if the prediction moves out of its domain.

4.3.7 Plan Length

The Plan length describes, how far an agent can look into the future. Here, we trade computational time for plan depth, which might hold critical information to sway the agent's decision. This is especially true since the agent has to accept higher initial reinforcement for an optimal reinforcement later on. Therefore, to solve the swing up, the agent has to imagine reaching the optimum and staying there. If the planning length cannot be extended significantly beyond reaching the optimum, the agent might decide against swinging up and instead move towards the equilibrium mentioned in Section 4.3.2.

There are two more constraints that need to be considered. The first one is computation time. Since the plan in combination with the world model is unraveled linearly during planning, GPU acceleration is not effective. Actually, GPU acceleration might even reduce performance, caused by the overhead of moving operations to the GPU. Furthermore, each action considers reinforcements with regards to the reinforcement energy E_{reinf} and because the loss is considered for each action, the computational time rises exponentially with each new action.

Secondly, due to this algorithm's known problem with vanishing gradients, it has to be shown in an experiment, that gradients from the later actions can actually reach and influence the last action.

From this, experiment conditions for the planning can be derived. One question to answer is, whether the algorithm works better when the earlier or later reinforcements are weighed as more important, giving the algorithm either a short- or a farsighted focus. The idea is that the first actions receive gradients from all other reinforcements, whereas the later, in extreme the last action, only receive

gradients with respect to a much smaller amount of reinforcements, or only one in the extreme case.

Another reason for weighing reinforcements is an effect, which will call gradient conflict. It is a derivative of the credit assignment problem, and explained in section 4.3.8.

4.3.8 Plan Convergence and Gradient Conflict

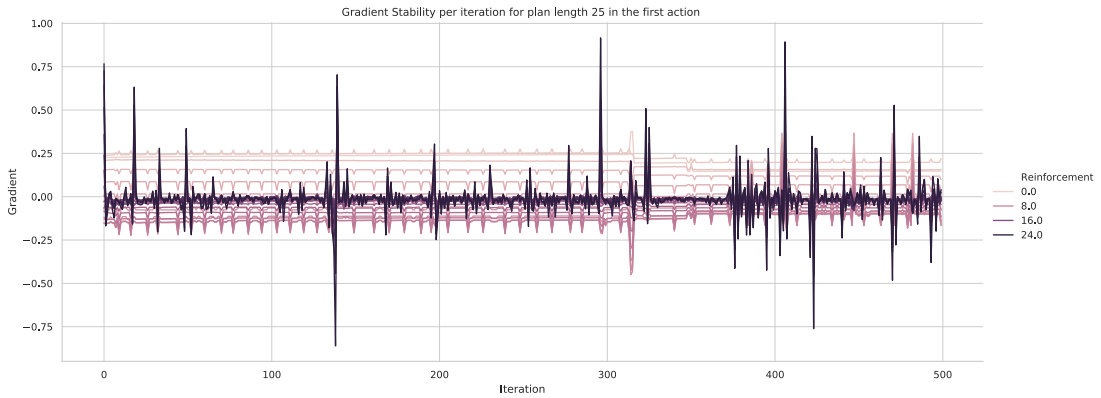
If we want to solve the environment, we need planning to improve our plan. In normal networks, we can check whether the error converges to zero, as we did with the world model in Figure 4.6, where we expected the training error of the world model to converge to zero. The same is applicable to the reinforcement energy E_{reinf} of the plan optimizer. In look-ahead planning, we expect E_{reinf} to converge towards the most optimal value that the agent can reach, since this is the one we are optimizing. We also expect the gradients to converge towards zero, as the actions in the plan get closer towards their optimal value.

However, these gradients are usually hidden from us, if we optimize E_{reinf} only. To obtain those gradients, we instead optimized the plan towards each reinforcement first.

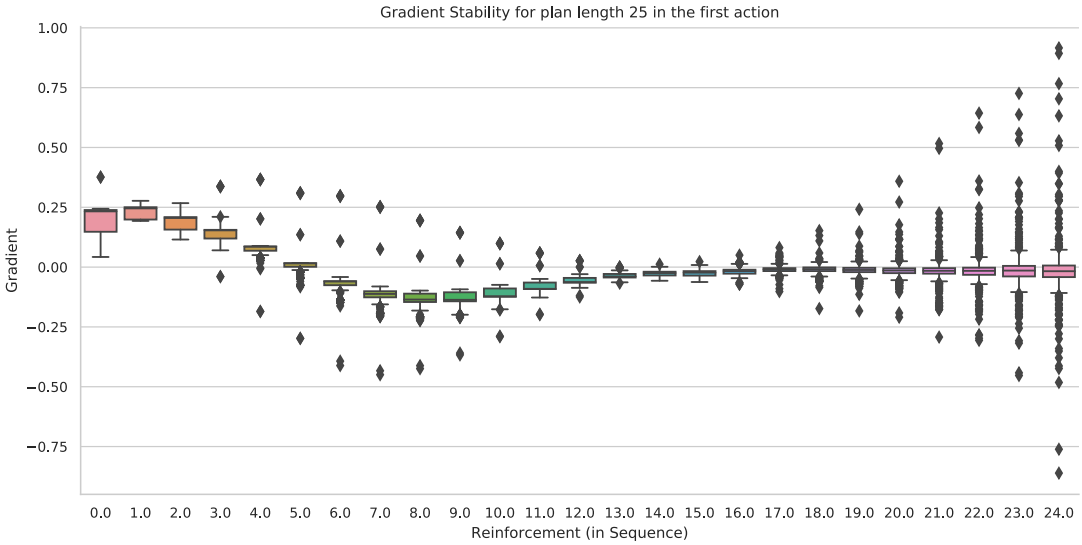
As can be seen in Figure 4.8a, the gradients caused by different reinforcements do not converge strongly. Also, later reinforcements (the darker ones in the figure), cause greater spikes and noise, indicating a potential exploding gradient problem. Furthermore, the noise occurs for all reinforcements synchronize at the same time, increasing in size with the depth of the plan, while the gradients themselves do not converge towards zero. The noise can potentially indicate that the sum of gradients actually converges to zero and the noise is the current plan being at a borderline to another close plan.

Potentially exploding gradients can also be seen in Figure 4.8b. Here, the distribution of gradients that the first action receives from each reinforcement is shown as a list of boxplots. The data was collected during a single rollout from a given starting state, with a radians for θ of 2 (approximately 115°) and $\dot{\theta}$ with a value of 4. From this point onward, the plan was refined for 500 planning iterations or 500 cycles in the plan optimizer. It can also be seen, that the gradients of the first five reinforcements are positive, whereas later gradients are usually negative, trying to optimize the plan in a different direction, conflicting with each other. We will refer to such gradients as conflicting gradients. Summing conflicting gradients equally will, at best, result in minimal gradients, and in the worst case, result in no optimization whatsoever. Slight changes in the reinforcements can have relatively large impacts on the resulting value that is applied to the action. Weighing the reinforcement values differently, for example preferring later values, might, therefore, improve performance. Therefore, the weighing of gradients is a condition in the experiments later on.

In Figure 4.9 graphs can be seen, when optimizing the plans with regards to E_{reinf} as a whole, instead of each reinforcement separately. Figure 4.9a shows again, the gradients that the first action of the plan receives. Instead of different



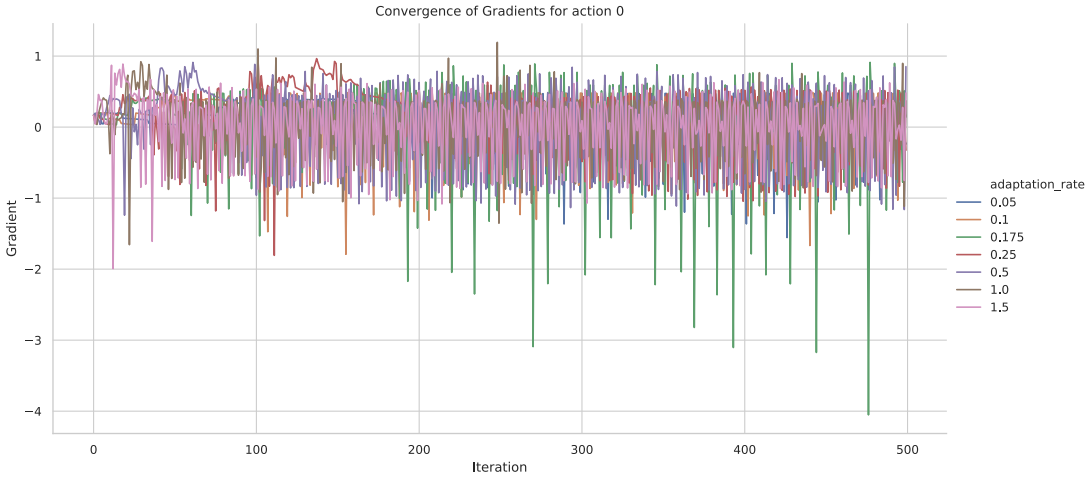
(a) Stability of gradients, where the plan was refined for 500 planning iterations (see Section 4.3.5), with a plan length of 25. Visualized are the gradients that the first action receives from every reinforcement caused by each of the following actions from the planned 25 steps.



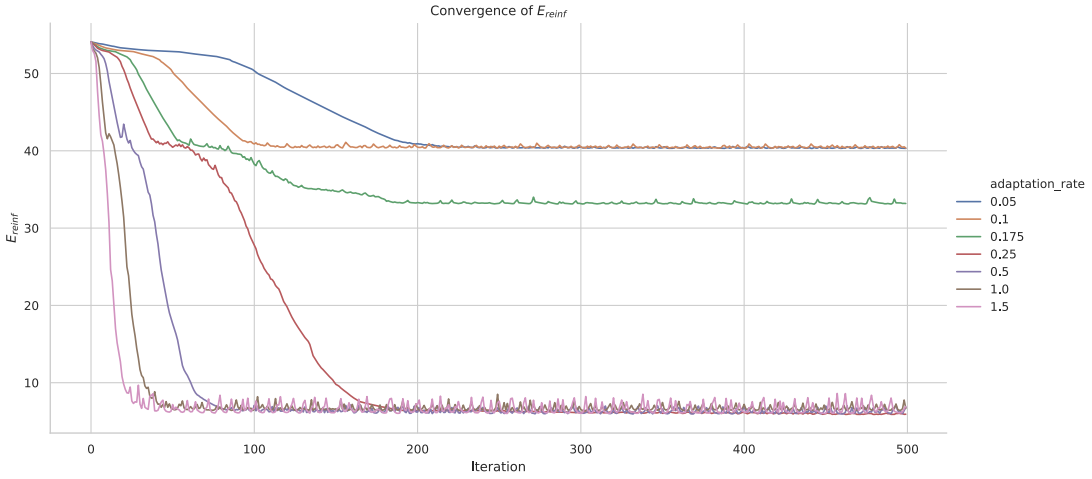
(b) Boxplot of gradient distributions the first action received from all future reinforcements caused by actions in the plan. The plan was refined for 500 planning iterations with a plan length of 25.

Figure 4.8: These convergence graphs were created when optimizing each of the reinforcements separately and thus getting access to the gradients. θ is initialized with a radians of 2 (approximately 115°) and $\dot{\theta}$ with a speed of 4.

gradient sources, different adaptation rates of the gradients (learning rate but here used in the planning context) are shown. As seen in Figure 4.8a, here too the gradients oscillate, do not converge strongly and contain noise. Figure 4.9b shows the change in the actual reinforcement energy E_{reinf} for different adaptation rates. Interestingly, all curves show a settle-point at around 40, whereas adaptation rates greater than 0.1 will continue converging towards another optimum. When settling, all curves experience noise. This raises the question, whether lower adaptation rates can reach an optimal plan. Furthermore, if the optimizer returns a plan with actions that are still in the action space, or if overfitting the plan will make it contain unreachable actions.



(a) Gradients the first action receives with respect to the reinforcement energy.



(b) The convergence of the reinforcement energy E_{reinf} for different adaptation rates.

Figure 4.9: These convergence graphs where created when optimizing the whole reinforcement energy E_{reinf} instead of each reinforcement separately.

4.3.9 Point of no return

Generally, the point of no return describes a point at which turning back is no longer possible, viable or safe. In our case, the point of no return describes the angle at which the pendulum can no longer be held up by the agent. At this point, the agent is no longer able to counteract gravity with his torque on a still pendulum, the pendulum gains no speed. The agent is expected to model this point and, from it onward, swing the pendulum through the bottom and up the other side again. Additionally, this is also the angle until which an agent could possibly hold the pendulum up since any speed towards higher angles would mean that the speed is greater than zero and the moving pendulum would fall down definitely. Unless the pendulum is swung up, in which case it is about to cross the point of no return. To acknowledge for this point, it is necessary to know where it is. We tested this prior

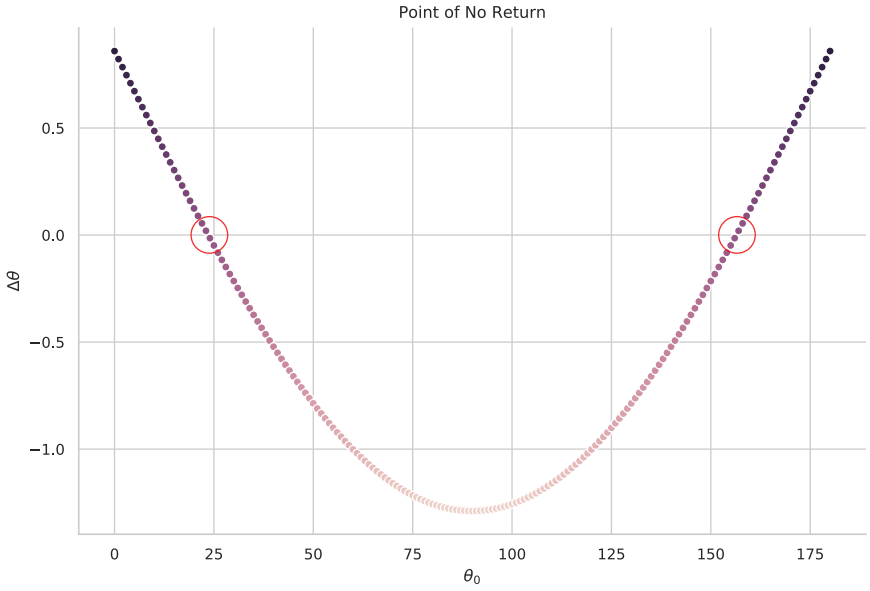


Figure 4.10: Changes in $\Delta\theta$ when initializing with with no speed and applying maximum torque towards the top. Marked are the points where the agent's torque and the gravity are equal. The left marker, closes to $\theta = 0$, shows the point of no return.

to the experiments by initializing a pendulum with no initial speed for every angle in one half of the pendulum, as both sides are symmetrical. We then logged how much the angle of the pendulum changes, if full force towards the top (or degree 0) is applied. If the difference is positive, the agent's torque successfully counteracted gravity. If it is negative, it did not and the pendulum moved towards the bottom.

This means that the angle at which the difference in theta first becomes negative is the point of no return. The result can be seen in Figure 4.10, where the angle 24° , with a $\Delta\theta$ of -0.014474, is the first angle with a negative $\Delta\theta$. This means the point of no return is in between angles 23° and 24° .

4.4 Experiments

Following the research objective, its sub-tasks and the discussion provided in the implementation, two experiments can be deduced.

4.4.1 Conditions

As discussed in section 4.3.8, weighing the gradients might improve planning performance, therefore we test the weighing of gradients as additional conditions. The weighing of the gradients follows the linear function $\text{grad}_i \cdot \frac{\text{i}+1}{N}$, where N is the amount of gradients, i the index of the gradient, determined by the action that caused this gradient and grad_i, the gradient. This function is used to weigh gradi-

ents in the condition, in which later reinforcements are preferred. To prefer earlier reinforcements, we simply invert the weighing term to $\text{grad}_i \cdot (1 - \frac{\text{i}+1}{N})$.

We will refer to these conditions as 'none', 'first' and 'last', depending on which reinforcements they prefer.

4.4.2 Pendulum Balance

The first experiment, pendulum balance, aims to show that an agent can hold the pendulum up in the upright position. In section 4.3.9, the point of no return in the environment was determined, at which we expect an agent to no longer be able to hold the pendulum up. We consider this task to be solved if the agent can hold the pendulum at an angle higher than 23 degrees for 50 timesteps.

First, we aim to show this for the 'none' condition only. For this, the pendulum is initialized with no initial acceleration, at angles with a five degrees increment ranging from -30 to 30 degrees, including the points of no return at both sides. Both sides are included to ensure that the agent works on both sides of the pendulum. As seen in section 4.3.8, multiple adaptation rates converge well, but to different values for E_{reinf}. For this experiment use a gradient adaptation rate of 0.5, with 100 iterations when optimizing the plan. As no far look-ahead is needed, we use a plan length of 10.

Next, we want to show this in combination with the weighing of gradients. Here, we increase the angle density to 2 degrees to a total of 31 angles between -30° to 30°, but keep adaptation rate and a number of iterations the same.

4.4.3 Pendulum Swing-Up

The second experiment focuses on the swing-up task. Here, all previous conditions compete against a random agent, which samples actions from the action space $[-2, 2]$ uniformly. The goal of the second experiment is to measure the overall performance in the complete swing-up task, measured by the accumulated reinforcements during different rollouts. The agents perform for 100 timesteps, with a more conservative gradient adaptation rate of 0.1, 100 iterations of plan optimization and 25 steps of look-ahead planning. With a plan length of 25, it is not guaranteed that the agent can see the goal in its prediction. However, this should not be necessary as the reinforcement function only is used to optimize the plan.

Chapter 5

Results

5.1 Pendulum Balance

The results of the first experiment can be seen in Figure 5.1. First, we tested how the algorithm performs without weighing the reinforcements, with initial angles between -30° and 30° in 5° steps. Each angle was tested for 66 times. This can be seen in Figure 5.1a, where all 66 rollouts are plotted. Interestingly, the variance and standard deviation of both θ and $\dot{\theta}$ is 0°. This means that there is no noise in the environment and that the optimizer always finds the same plan. Out of the 13 angles tested, the four angles beyond the point of no return, that are -30°, -25°, 25° and 30°, failed the task as expected.

Figures 5.1b - 5.1d contain the results for the second part of the experiment, where the conditions 'none', 'first' and 'last' are executed with a 2° precision between each initial angle. In total, 31 angles were tested for each condition. Since we have previously discovered the deterministic nature of the environment, the experiments were run five times to validate that finding earlier in the experiment.

In Figure 5.1b, the results for the 'none' condition can be seen. Similar to the first experiment of this condition, pendulums at angles beyond the point of no return could not be held up. Surprisingly, the agent failed asymmetrically at 22°, but not at -22°. Similar results can be seen for the 'first' condition in Figure 5.1c. The agent in the 'last' condition, seen in Figure 5.1d, failed at more angles than the previous conditions. Specifically, the agent failed at angles less than -18° and greater than 16°, repeating the asymmetry seen in the previous conditions. This might be due to how the linear weighing is calculated (see Section 4.4.1), which values the reinforcement from the last action with 1, but the first action with 0. This is not a problem for the 'first' condition, as this means that the last step is simply ignored. In this task, however, shortsightedness is most important and ignoring the results of the first step might be fatal, especially when the following steps are only weighed slightly more.

Surprisingly, in all conditions, the angle of the pendulums seem not to converge to 0°, but to a value closer to -5° θ. This is can be caused by the reinforcement function described in Section 4.3.6, where we needed to approximate the reinforcement

functions as the world model did not predict the hidden states of the environment. Alternatively, the world model could potentially have a misaligned.

Another aspect that has to be taken into account is that this experiment uses a more liberal gradient adaptation rate of 0.5, which causes the reinforcement energy E_{reinf} to converge very fast, as seen in Figure 4.9b. However, this figure also shows that different adaptation rates find different optima. It could be possible, that higher adaptation rates create plans that in theory minimize E_{reinf}, but in actuality are not executable by the agents.

This idea is supported by the fact, that neither the original look-ahead planning from Thrun et al. [11] nor this implementation have any way of limiting the optimization with respect to the agent's constraints in the real world. It could certainly be possible, that plans are created that expect the agent to move the pendulum into the optimal position in the first step and then to never move the pendulum again. In the Pendulum Balance task, this would not be a problem since the goal state is so close and it would benefit algorithms that focus on the first action, as here convergence of the plan actions would be faster, compared to the 'last' condition. In the Swing-Up task, however, this could remove the agent's ability to plan around the obstacle of not being able to swing past the point of gravitational and the agent's forces equilibrium, if the plan simply optimizes through it.

As we did not test this behavior of the algorithm, we can not give definite answers about how often this happens. After a small investigation though, we can confirm that it does happen, as the following plan shows, which was optimized for the first step of an agent with a pendulum initialized at angle 20 and the other parameters were kept the same from the experiment:

$$[-5.053009, -1.5466716, -1.5755008, -1.5546209, ..., -1.1432618]$$

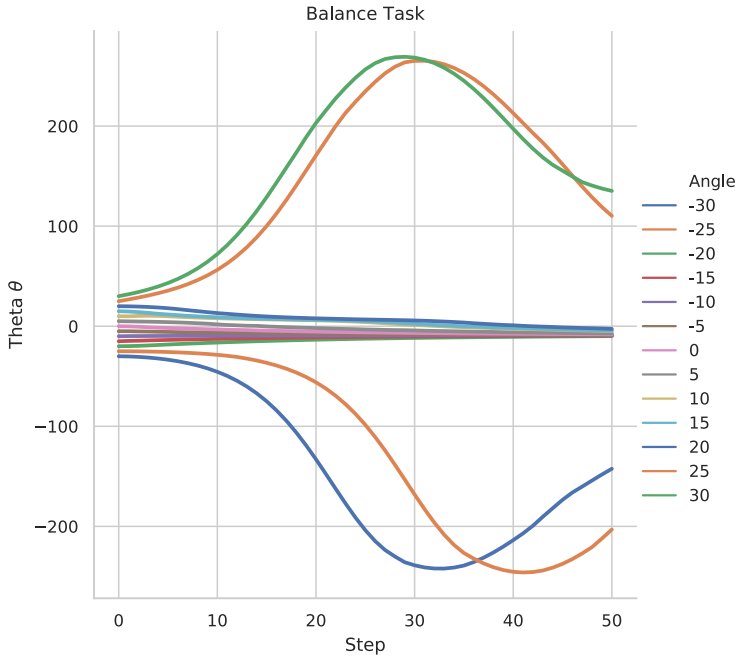
Here, the first action is -5, which is definitely outside of the agent's capabilities, when its action space is only [-2, 2].

In summary, agents using look-ahead planning are able to hold the pendulum up. They can save the pendulum at angles closer to the point of no return if they fully value the reinforcement of the next action, independent of how strong they weigh later actions. If agents ignore the first reinforcement and instead focus on later states, they are not able to save pendulums closer to the points of no return. This shows, that this algorithm can at least hold the pendulum upwards for 50 consecutive timesteps if it manages to swing the pendulum up into such a position. This is questionable, as we also discovered a critical flaw in the plan optimization of the agent, namely ignoring the capabilities and constraints an agent faces in the real world when refining plans it should execute.

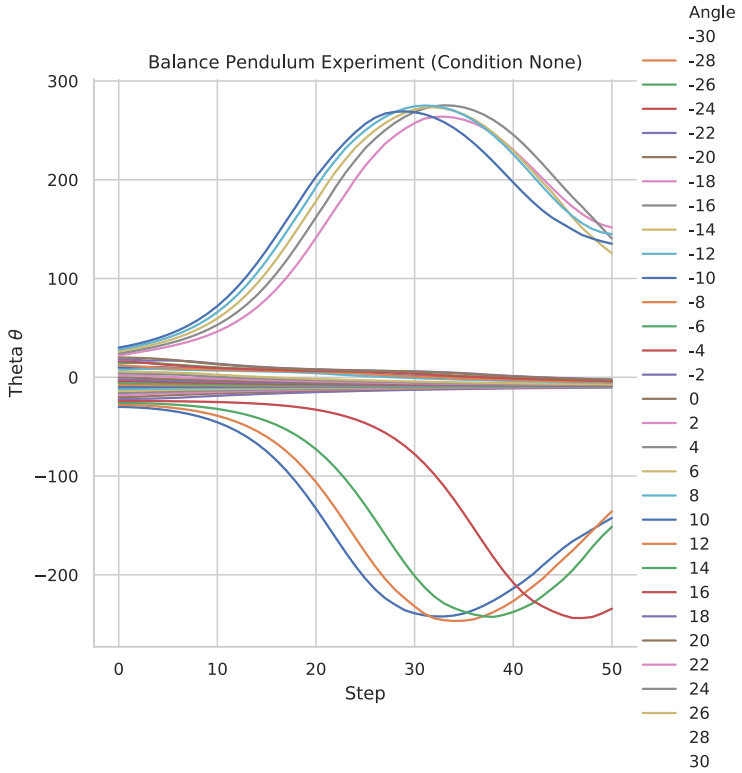
5.2 Pendulum Swing-Up

In the second experiment, we want to measure the agent's overall performance in the environment. We compare the mean accumulated reinforcement achieved at the end of multiple rollouts, for the three conditions and a random agent. In

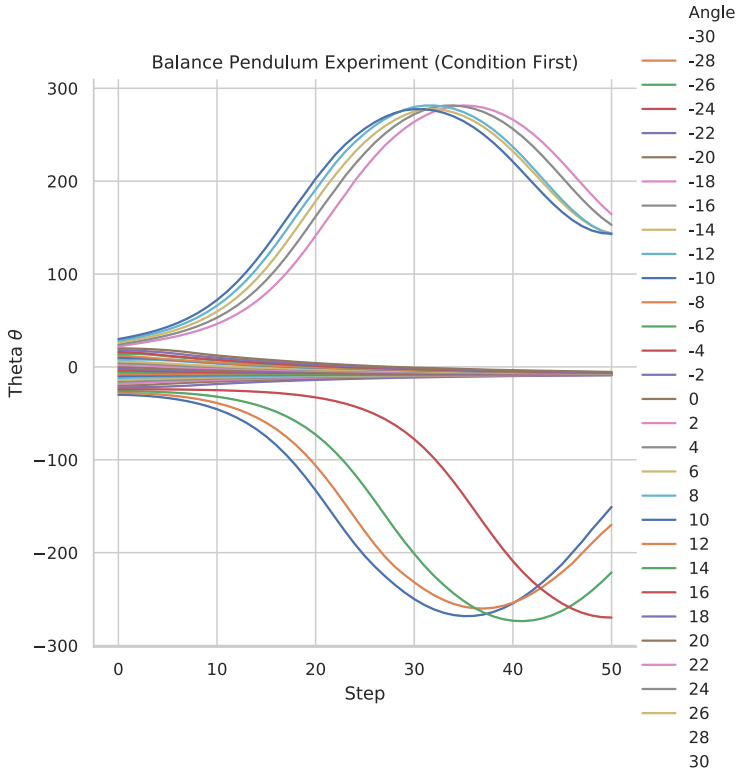
Figure 5.1: The results for the Pendulum Balance experiment.



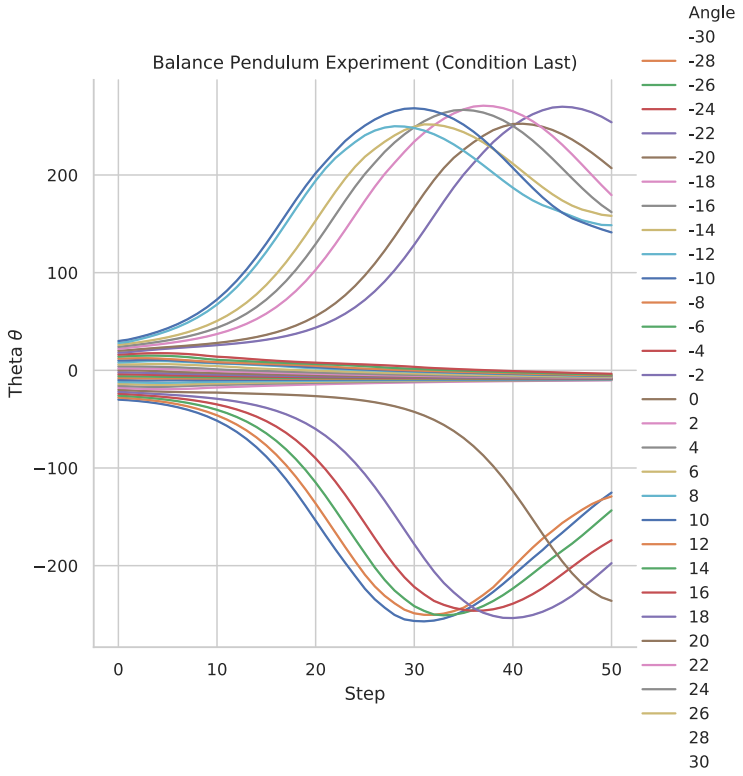
(a) Results for the Pendulum Balance experiment with no weighing of reinforcements. The initial angles for the agent are in $[-30, 30]$, with a separation of 5°. For each angle, 66 rollouts are plotted in the figure.



(b) Results for the Pendulum Balance experiment with no weighing of reinforcements, but with initial angles for the agent in $[-30, 30]$, with a separation of 2°.



(c) Pendulum Balance Experiment with weighing of earlier Reinforcements, but with initial angles for the agent in $[-30, 30]$, with a separation of 2°.



(d) Pendulum Balance Experiment with weighing of later reinforcements, but with initial angles for the agent in $[-30, 30]$, with a separation of 2°.

Strategy	Mean	Var	Std
none	**-577.537**	74915.938	273.708
first	**-573.788**	45020.458	212.180
last	-686.737	26312.005	162.210
random	-619.800	**24504.078**	**156.538**

Table 5.1: Results for the first part of the experiment, using a plan length of 10. Shown is the distribution of the accumulated reinforcement after 60 rollouts. Best results are marked bold.

Strategy	Mean	Var	Std
none	-644.750	7043.627	83.926
first	-613.978	6348.385	79.677
last	-643.694	**4132.110**	**64.281**
random	**-590.275**	22557.090	150.190

Table 5.2: Results for the second part of the experiment, where the agents used a plan length of 25. The distribution of the accumulated reinforcement after 25 rollouts is shown. Best results are marked bold.

the first part, we use 10 as the plan length, the same as in the first experiment. In the second part, we increase the plan length to 25, expecting an increase in performance in the planning conditions.

5.2.1 Results for Plan Length 10

Table 5.1 shows the results for the first part of this experiment. For this part of the experiment, 60 rollouts with random initialization were calculated for each condition and the agents had a plan length of 10. In this part of the experiment, the 'none' and 'first' condition scored the best mean accumulated reinforcements, with agents in the 'first' condition being marginally ahead. The relative difference between agents in the 'first' condition and random agents is approximate -8.02%. Followed by the random agents and agents in the 'last' condition scored last.

Surprisingly, the variance and standard deviation of agents in the planning conditions are higher than for random agents, with agents in the 'none' condition showing the highest variance. This means, that on average the agents in the 'none' and 'first' condition score better accumulated reinforcements than random agents, but can be way higher in some cases.

Figure 5.2a shows mean reinforcements the agents received at different timesteps. The first dip after the initial reinforcement of 0 is interesting, as it gives an insight into what state the agents were initialized in. Here, the 'none' agents initialized in states with better and 'last' agents condition in states with worse reinforcement.

Furthermore, the next few steps show sine-like curves, indicating that on average a fall was one of the first patterns seen. Also, none of the curves approach the optimum of zero reinforcement, indicating that the agents were not able to swing the pendulum up reliably, which might have been caused by the inability to plan around the local optima, found in the previous experiment in Section 5.1. The agents in the 'none' and 'last' condition seem to have, on average converged to different local optima after step 40.

How the reinforcements accumulate on average is shown in Figure 5.2b. The 'first' and 'none' agents seem to behave similar, same for 'last' and random agents. Agents in the 'last' conditions fall behind the random agents at later steps, breaking away after around 40 steps, which could be connected to the convergence in the previous figure.

In summary, the agents in the 'first' and 'none' condition were marginally better than random agents but did not manage to reliably converge the pendulums angle towards zero reinforcement.

5.2.2 Results for Plan Length 25

The results for the second part of the Pendulum Swing-Up experiment can be seen in Table 5.2. Here, the agent has a plan length of 25 and the results were calculated during 25 random rollouts. Compared to the first part of this experiment in Section 5.2.1, fewer rollouts were used due to increased computation time due to the plan length.

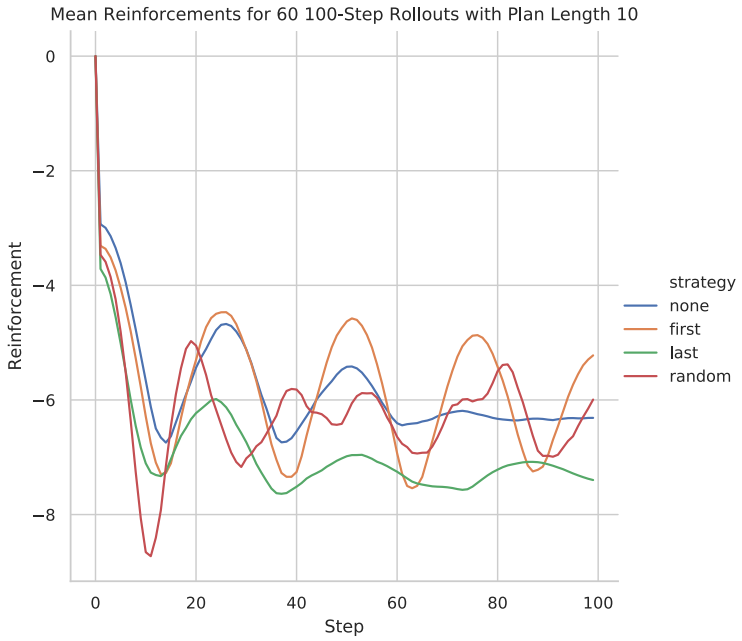
When comparing the different conditions, all conditions scored worse mean accumulated reinforcements than the random agent. Second, best is the agent which preferred the reinforcements of earlier predictions and the agents in 'none' and 'last' condition tie last. However, the relative difference between the random and 'first' condition is only approximately -4.02%. The 'first' condition scored better than the two other planning conditions, which is consistent with the previous experiment.

When we look at the consistency of the agents, the results change. Unexpectedly, the accumulated reinforcements of the random agents vary stronger than that of agents in the planning conditions. This is in contrast to the findings of the first part of this experiment. It can be seen in both the variance and standard deviation. The agents in the 'last' condition scored the least varying results, ahead of agents in the 'first' condition, who are slightly better than agents in the 'none' condition.

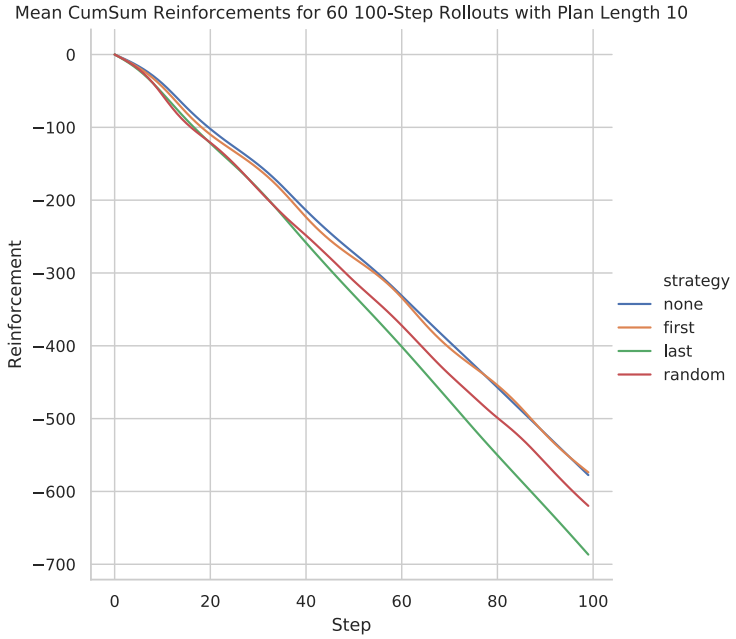
The mean reinforcement that the agents receive during each timestep can be seen in Figure 5.3a. As every agent started with 0 reinforcement we can again compare the average reinforcements for the first action to compare initializations. This can be seen best after the first step when the values break off from the vertical line connecting the curve to the initial zero reinforcements. Random agents have, on average, less reinforcement from the first step than any of the other conditions. Also, the agents in the 'first' condition initialized on average the worst.

When looking at all steps, sine-curve like patterns can be seen, especially for random agents. This could indicate that the plans did not converge successfully

Figure 5.2: Results of the Pendulum Swing-Up experiment for agents in conditions with a plan length of 10.

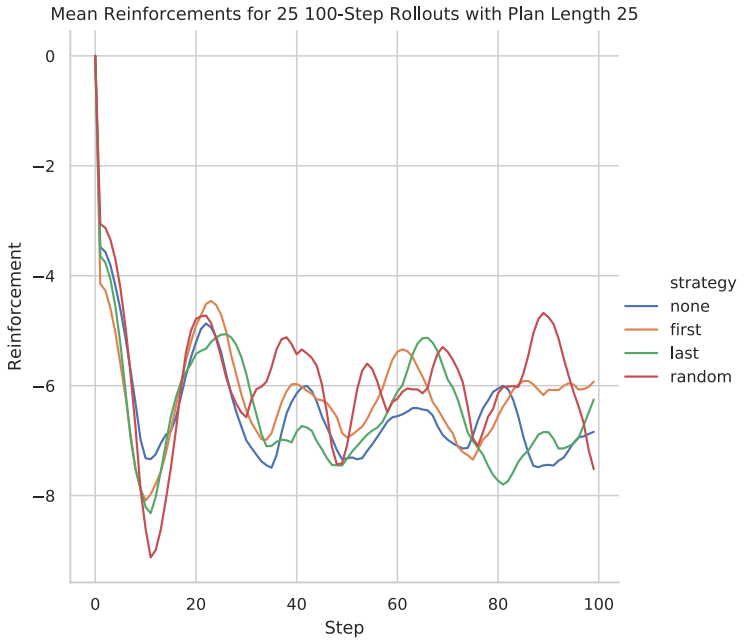


(a) Mean reinforcement the agents in different conditions received per timestep. The data was collected during 60 rollouts with random state initialisation.

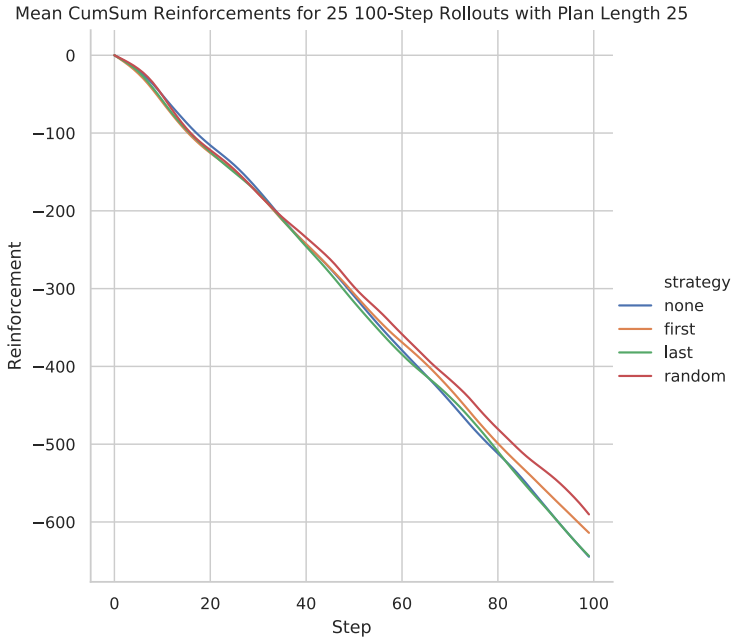


(b) Mean accumulation of reinforcements the agents received per timestep. The final mean sum is used to measure the performance of a given condition. The data was collected during 60 rollouts with random state intialisation.

Figure 5.3: Results of the Pendulum Swing-Up experiment for agents in conditions with a plan length of 25.



(a) Mean reinforcement the agent received per timestep for the four conditions. The data was collected during 25 rollouts with a random initialisation each.



(b) Mean accumulated reinforcement the agent received after each step during a rollout for the four conditions. The reinforcements accumulate with every step the agent takes in the environment, where the final accumulated reinforcement is used to measure the agents performance. The data was collected during 25 rollouts with random initialisation.

and their states were mostly dependent on the movement of the pendulum itself. Furthermore, the planning conditions did not, on average, converge towards zero, meaning they were not successful at swinging up the pendulum consistently.

The mean cumulative sum of reinforcements during each timestep is plotted in Figure 5.3b. Here we can see that on average, the mean cumulative sum of reinforcements does not vary much between conditions, which is supported by the relative small lead the random condition has against the other conditions in mean accumulated reinforcement seen in Table 5.2.

In summary, the agents in the planning conditions scored worse than random agents and were not able to converge towards the optimum.

Chapter 6

Conclusion

6.1 Summary of Results

The goal of this work is to explore the capabilities of look-ahead planning in continuous, open-ended reinforcement learning tasks. The exploration is done in the pendulum swing-up environment as a representative task for general reinforcement learning, from which to generalize the agent's potential. To show this, we both explore the implementation and analyze experiments in this environment.

To answer our initial research question from Section 4.1, we conclude that look-ahead planning, in its current implementation, is not capable of solving the pendulum swing-up task, thus not proving its abilities for other reinforcement learning tasks, as Section 5.2 shows. Here, look-ahead planning is only marginally better than a random agent, and only if it operates with a small plan length. Still, we could not show that the agent is capable of swinging the pendulum up reliably, as seen in Figure 5.2a and 5.3a.

Crucially, we discovered a major problem of the current implementation in Section 5.1, rendering look-ahead planning ineffective for our use case. The optimization of look-ahead planning is in not constrained and will, given enough iterations, over-optimize plans to a point where an agent with constrained action-space or any real-world agent, cannot possibly execute the plan.

However, potential can be seen in Section 5.1, showing that look-ahead planning is able to balance the pendulum successfully for at least 50 time-steps and that the principal approach is merited.

Since we could not show the success of look-ahead planning in the swing-up experiment, we can not give a conclusion to the gradient conflict problem mentioned in Section 4.3.8. The results show, that the preferring earlier actions, resulting states and reinforcements return better or equal results to no preferring, as seen in Tables 5.1 and 5.2. Here, preferring later states and assigning zero value to the first action was worse or at best equal to no preferring. However, as the swinging-up could not be shown, these results might be caused by the agents getting closer to the local optimum, namely the equilibrium point of agent force and environment gravity, thus the results are inconclusive to the actual performance of the weighing.

As other work has reported in Chapter2, we too have find problems with our gradients, as seen in Figure 4.3.8. Instead of vanishing gradients, we see exploding gradients, another deep-learning gradient problem. This does make sense, however, as unrolling the plan basically creates a deep network, with each action adding another layer of the world model and resulting state. Basically, the resulting states and action pairs are fully connected layer with no bias and the identity function as the activation function. Thus, the longer the plan, the deeper the network and the more prominent deep-learning problems become.

Another critical aspect we found during implementation is seen in Section 4.3.7. For each step, the plan needs to be unrolled and optimized for multiple iterations until E_{reinf} converges (see Figure 4.9b). Since unrolling the plan and optimizing it has to happen sequentially, modern methods of GPU acceleration cannot be applied. As the optimization happens online, while the agent acts in the environment, the agent is either highly constrained in the planning depth and amount of plan optimizations it can do, or in the time it needs between each step.

6.2 Contribution

This work contributes a first implementation of look-ahead planning based on the work of Thrun et al. [11] in python with TensorFlow eager execution and OpenAI Gym for continuous, open-horizon, reinforcement learning tasks. The modularity of the modules allows for expansions into different environments and adaptations to the algorithm. Further, the implementation is discussed and explored in Section 4.3, explaining the adaptations of the original implementation to fit the given task in detail.

Additionally, we designed and conducted two experiments to analyze the behavior of the algorithm in different scenarios. We show firstly, that the current implementation is not able to swing the pendulum up reliably and secondly, that the algorithm is capable of holding the pendulum up once it reaches an upward position.

Lastly, we discovered and explained a weakness of the algorithm, that is one of the possible reasons it fails to succeed in our task and that makes it ineffective for different tasks involving agents with limiting action capabilities, such as real-life agents.

6.3 Future Work

Following from the findings in Section 6.1, if this implementation is to be applied in other continuous control tasks with limited actions, the problem of over-optimization due to a missing optimization limiter should be focused on first, as this currently breaks the algorithm for the tasks discussed in this thesis.

Further work could explore different parts of the original algorithm [11], that are not part of this thesis, especially an annealing factor for the plan optimization,

forcing the plan to converge towards a sequence, and the experience model, learning to predict the planning result. The experience network is part of the plan creation topic, which this thesis does not cover, but could be interesting for future work. Other than the experience network and plan optimization with gradient descent, a tree exploration in state space with actions might find good plans, such as in the work of Pascanu et al. [7], where a tree is explored to find the optimal plan.

This would tie together with the next aspect, the computational load online. During a rollout, the agents have to unroll the model multiple times to optimize the plan. Some of this load would be relieved by an experience network, but the optimization would have to stop on plan convergence. Another architecture could also help with that, where the agent only plans until the action it does becomes instinct and only uses the planning to navigate through unknown territory, which would be similar to the experience network. Additionally, newer optimizer like the adam optimizer could be examined in future work.

Considering the experiments we conducted in Section 4.4, with the results in 5, future work could examine the algorithm in the swing-up experiments for longer timesteps, giving it a chance to randomly reach the goal state. Additionally, we only conducted experiments for two plan lengths, but an analysis of the plan length's effect on the accumulated reinforcement could hold interesting results.

In Section 4.4.1 we explained the conditions we used in the experiment, trying to tackle the gradient conflict problem, weighing the plan linearly. As the agents in the 'first' condition did perform better or equal to agents with no weighing, it might be worthwhile to improve this. One way would be to use the error of the world model for consecutive predictions and using the inverse of the error as a weighing function, reflecting the known uncertainty of the world model onto the reinforcements.

The world model could be further tweaked too. In environments with more attributes in each world state, a model that learns trajectories through latent state space, as seen in Silver et al. [9], might be helpful. One way would be to quantitatively analyze the effect of different activation functions on the predictive ability, such as elu or softsign. In this thesis, we used the raw state as input, but this might cause the model to weigh the higher values form the thetadot $\dot{\theta}$ attribute as more important, one could thus look into normalizing the input before using it with the model. Another approach that would become necessary in an environment that does not encode temporal effects would be the use of a recursive neural network (RNN), which can be seen in the work of Ha and Schimdhuber [4]. Future work could try different world model architecture, b Though the deep-learning problems encountered in this thesis, exploding gradients in this case should be kept in mind.

Lastly, future work could try this algorithm in a different environment with a more accessible reinforcement function or use a world model that too predicts the reinforcement, as done in the original implementation [11].

Bibliography

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," mar 2016. [Online]. Available: http://arxiv.org/abs/1603.04467

[2] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," jun 2016. [Online]. Available: http://arxiv.org/abs/1606.01540

[3] A. M. David Poole, *Artificial Intelligence: Foundations of Computational Agents, 2e.* Cambridge University Press, 2017. [Online]. Available: http://artint.info/

[4] D. Ha and J. Schmidhuber, "World Models," mar 2018. [Online]. Available: http://arxiv.org/abs/1803.10122

[5] N. Heess, D. TB, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, S. M. A. Eslami, M. Riedmiller, and D. Silver, "Emergence of Locomotion Behaviours in Rich Environments," *arXiv preprint arXiv:1707.02286*, 2017. [Online]. Available: http://arxiv.org/abs/1707.02286

[6] S. Otte, T. Schmitt, K. Friston, and M. V. Butz, "Inferring Adaptive Goal-Directed Behavior Within Recurrent Neural Networks," in *International Conference on Artificial Neural Networks*, 2017, pp. 227–235.

[7] R. Pascanu, Y. Li, O. Vinyals, N. Heess, L. Buesing, S. Racanière, D. Reichert, T. Weber, D. Wierstra, and P. Battaglia, "Learning model-based planning from scratch," jul 2017. [Online]. Available: http://arxiv.org/abs/1707.06170

[8] S. Ruder, "An overview of gradient descent optimization algorithms," sep 2016. [Online]. Available: http://arxiv.org/abs/1609.04747

[9] D. Silver, H. van Hasselt, M. Hessel, T. Schaul, A. Guez, T. Harley, G. Dulac-Arnold, D. Reichert, N. Rabinowitz, A. Barreto, and T. Degris, "The Predictron: End-To-End Learning and Planning," dec 2016. [Online]. Available: http://arxiv.org/abs/1612.08810

[10] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2017. [Online]. Available: http://incompleteideas.net/book/bookdraft2018jan1.pdf%0Ahttp://incompleteideas.net/sutton/book/bookdraft2017june.pdf

[11] S. Thrun, K. Moller, and A. Linden, "Planning with an Adaptive World Model," in *Advances in Neural Information Processing Systems (NIPS)*, 1991, pp. 450–456. [Online]. Available: http://papers.nips.cc/paper/365-planning-with-an-adaptive-world-model.pdf

[12] X. Zhou, C. Weber, C. Bothe, and S. Wermter, "A hybrid planning strategy through learning from vision for target-directed navigation," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 11140 LNCS. Springer, Cham, oct 2018, pp. 304–311. [Online]. Available: http://link.springer.com/10.1007/978-3-030-01421-6_30

[13] J. Zico Kolter and A. Y. Ng, "The stanford littledog: A learning and rapid replanning approach to quadruped locomotion," *International Journal of Robotics Research*, vol. 30, no. 2, pp. 150–174, feb 2011. [Online]. Available: http://journals.sagepub.com/doi/10.1177/0278364910390537

Erklärung der Urheberschaft

Hiermit versichere ich an Eides statt, dass ich die vorliegende Bachelorthesis im Studiengang Human-Computer-Interaction selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel - insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ort, Datum Unterschrift

Erklärung zur Veröffentlichung

Ich stimme der Einstellung der Bachelorthesis in die Bibliothek des Fachbereichs Informatik zu.

Ort, Datum Unterschrift