Autonomous Soaring Policy Initialization through Dynamic Programming

Benjamin Rothaupt

Masterarbeit

2018

Universität Stuttgart
Institut für Flugmechanik und Flugregelung

Erklärung

Hiermit versichere ich, dass ich diese Masterarbeit selbständig mit Unterstützung der Betreuer angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Die Arbeit oder wesentliche Bestandteile davon sind weder an dieser noch an einer anderen Bildungseinrichtung bereits zur Erlangung eines Abschlusses eingereicht worden.

Ich erkläre weiterhin, bei der Erstellung der Arbeit die einschlägigen Bestimmungen zum Urheberschutz fremder Beiträge entsprechend den Regeln guter wissenschaftlicher Praxis[1] eingehalten zu haben. Soweit meine Arbeit fremde Beiträge (z.B. in der Form von Bilder, Zeichnungen, Textpassagen etc.) enthält, habe ich diese Beiträge als solche gekennzeichnet (Zitat, Quellenangabe) und eventuell erforderlich gewordene Zustimmungen der Urheber zur Nutzung dieser Beiträge in meiner Arbeit eingeholt. Mir ist bekannt, dass ich im Fall einer schuldhaften Verletzung dieser Pflichten die daraus entstehenden Konsequenzen zu tragen habe.

Ort, Datum	Benjamin Rothaupt

Betreuer: M. Sc. Stefan Notter

[1] Nachzulesen in den DFG-Empfehlungen zur „Sicherung guter wissenschaftlicher Praxis“ bzw. in der Satzung der Universität Stuttgart zur “Sicherung der Integrität wissenschaftlicher Praxis und zum Umgang mit Fehlverhalten in der Wissenschaft“

Kurzfassung

Testtext Testtext 123

Abstract

Dies ist der englische Abstract.

Contents

List of Figures

List of Tables

Nomenclature

Abbreviations

ANN	Artificial Neural Network
CPU	central processing unit
DP	Dynamic Programming
GPI	Generalized Policy Iteration
GPU	graphics processing unit
iFR	Institute for Flight Mechanics and Control
MDP	Markov Decision Process
MLP	multi layer perceptron
MSE	mean squared error
OC	optimal control
OPI	Optimistic Policy Iteration
PE	Policy Evaluation
PI	Policy Iteration
RAM	random access memory
RL	Reinforcement Learning
SL	Supervised Learning
VI	Value Iteration
VRAM	video random access memory

Latin Letters

$\bar{e}$	mean error
$\boldsymbol{b}$	bias
$\boldsymbol{W}$	weights
Δt	time step
$\mathcal{A}$	action space
$\mathcal{N}$	Gaussian Distribution
$\mathcal{P}$	State Transition Probability

$\mathcal{S}$	state space	
ρ	$\left[\frac{\text{kg}}{\text{m}^3}\right]$	air density
a	action	
e	error	
m	[kg]	glider mass
o	observation	
Q	Action Value Function	
q	$\left[\frac{\text{N}}{\text{m}^2}\right]$	dynamic pressure
r	reward	
s	state	
V	State Value Function	
D	drag	
L	lift	

Greek Letters

α	[rad]	angle of attack
$\boldsymbol{\theta}$	parameter vector	
γ	discount factor	
Λ	aspect ratio	
μ	mean	
ϕ	[rad]	flight path angle
π	policy	
σ	standard deviation	
e	oswald efficiency factor	

Indices

$*$	optimal
π	w.r.t. policy π
T	terminal
t	time step
D	drag
L	lift

1. Introduction

The goal of this work is to find an optimal trajectory of a glider moving through calm air and reaching a given distance in minimal time. The resulting optimal policy can be used as a starting point for training the glider on scenarios with different wind conditions utilizing other algorithms like Reinforcement Learning (RL). The basic idea is to use the knowledge about the glider dynamics to pre-train a policy. After that, an RL algorithm is used to train the policy on scenarios with the agent facing a stochastic environment.

In soaring competitions, there are two main goals that pilots must be able to achieve in order to be successful: exploiting updrafts to gain potential energy and covering a given distance in minimal time. These are two conflicting tasks. The longer a pilot decides to stay within an updraft, the more energy can he harvest. The time spent there however increases the total flight time which interferes with the other goal. Generally, the locations where updrafts occur are not known a-priori. So a pilot has to react spontaneously to changes in his vicinity in order to harvest as much energy as possible, which takes a lot of experience. Automation of the optimal-flying task is therefore challenging.

Another topic, where using energy from updrafts can be helpful, is electric flying. Electric aircraft have been an important research topic in recent years. They can operate environmentally friendly and are easy to maintain. Their range and endurance is - however - limited by the energy density of current batteries. This means they either have to land frequently to recharge or be equipped with solar panels on their wings which weigh them down and are difficult to maintain. Electric UAVs suffer from the same deficiency. They can theoretically be used for a wide range of applications from gathering scientific data to surveillance with battery capacity being their only limiting factor. The exploitation of thermal updrafts is one way to increase the endurance of an aircraft without changing anything on the airframe itself, which makes it relatively cheap to implement.

Reacting to changes in the environment is a topic that is inherently covered by various Machine Learning algorithms. In fact, the most challenging and therefore interesting and powerful aspect of *learning* is gaining the ability to generalize and behave reasonably in new situations, such as unknown updraft distributions. This sets Machine Learning apart from traditional optimal control algorithms that rely on precise knowledge of the underlying problem.

Finding an optimal flight path with respect to varying mission goals in unknown atmosphere is a challenging task. While the dynamics of the aircraft itself are known and understood, the occurrence of updrafts cannot be predicted with sufficient accuracy. RL algorithms are very good at finding an optimal path through an unknown environment. They require no prior knowledge and learn from experience they gather through inter-

acting with the environment. The stochastics subject to the likelihood of occurence of thermal updrafts and their specific characteristics are hard to model. Sampling from experience constitutes a way to learn how to act, while facing (hidden) stochastics within the environment. This way, RL algorithms can deal with (apparently) random updrafts efficiently.

Markus Zuern applied Trust Region Policy Optimization (TRPO) and Asynchronous Advantage Actor Critic (A3C) to a trajectory optimization problem [12]. In his work, both algorithms deal with the complete environment, i.e. the glider dynamics and the stochastic wind distribution, at the same time. They require no knowledge about the environment, nor do they try to learn a model. Instead, they directly optimize a policy. In TRPO, there does not even exist a value function that represents known information about the MDP. Although the results in [12] are remarkable, they do not exploit the fact that one part of the trajectory optimization MDP is known a priori: the glider dynamics when flying through calm air.

The goal of this work is to utilize this knowledge by applying Dynamic Programming to calculate a time-optimal trajectory through calm air by finding an optimal policy and then following it. The resulting policy is then trained with unknown wind distributions. This is similar to the behavior one would expect from a human pilot. If he/she is confronted with unknown updrafts, he/she would at first also rely on his experience from prior flights, be it through calm air or not. Dynamic Programming (DP) is one of the most popular algorithms for MDPs with a state space that is not too large. Typically, MDPs with up to a few millions of states can be solved with DP algorithms.

Dynamic Programming is a general term that summarizes a certain kind of algorithms. These algorithms can be used to solve problems that can be split into subproblems. The optimal solution of each subproblem can then be used to recompose the optimal solution to the whole problem. The underlying principle is called the *principle of optimality*. The term was first used by Richard E. Bellman in 1954 [3]. He proved that various optimization problems could be solved efficiently by making use of the principle of optimality instead of traditional optimal control theory. In his book *Applied Dynamic Programming*, Bellman covered numerous examples of optimal control problems and shows how they can be solved by Dynamic Programming [4]. If the subproblems are similar, DP is especially effective. DP however suffers from the "curse of dimensionality", that is in large or continuous state or action spaces it quickly becomes inefficient [9]. The obvious approach is to discretize the state space and deal with a smaller number of states. If done properly, this yields an approximation of the exact optimal solution. The achieved approximate solution is close to the exact one if the discretization is sufficiently fine. Like in many technical topics, there is a tradeoff between precision and cost.

In this work, a time optimal flight path through calm air is calculated with dynamic programming. First, the state and action space are discretized in sections 4.1.4 and 4.2.1. A policy iteration (PI) algorithm is then used to calculate optimal paths in the vertical plane and in a 3D environment. The usefulness of the results from the 3D scenario is obvious. If airspace restrictions or topography restrict the horizontal trajectory, any 3D

scenario degenerates to a 2D optimization of a vertical trajectory along the predefined ground trace. Therefore, the results of optimization in the vertical plane shown in section **??** are also relevant.

The first part of this thesis presents the general concepts and terms of Reinforcement Learning (cf. chapter 2) and the theory of Dynamic Programming and the algorithms used in this work (cf. chapter 2.11). Chapter 3 gives a brief overview of artificial neural networks (ANNs) and how training an ANN works. The trajectory optimization problem is described in chapter 4 and the results are presented and compared to an optimal control result in chapter 5. The last chapter contains a brief discussion of the results and how using an ANN, that has been pre-trained by DP, for RL affects the results.

2. Solution Methods for Markov Decision Processes

2.1. Markov Decision Process

Path planning through calm air can be done analyzing how a glider behaves in calm air. Flying a glider can be regarded as a decision process where the pilot decides what control surface position is suitable for the current situation. By varying, for example, the elevator position, the pilot can control the speed of the glider and thereby the flight path angle. In reality, this process is time-continuous. Computers - however - can only deal with discrete processes. Therefore it is common practice to introduce a time step Δt. At each moment in time $t_k = t_0 + k\Delta t$, an elevator position is chosen and is kept constant until t_{k+1}. The smaller the time step, the better the approximation of a continuous process. Each state s_t in this decision process has the Markov property, i.e. the information needed to calculate s_{t+1} is fully contained in s_t. In other words, the probability of reaching a state s_{t+1} by taking an action a_t only depends on the state s_t. The predecessor states of s_t are irrelevant for $P(s_{t+1}|s_t, a_t)$. A decision process where the Markov property applies to all states is called a Markov Decision Process (MDP).

$$P(s', r|s, a) = \mathbb{P}[s_{t+1} = s', r_{t+1} = r|s_t = s, a_t = a] \tag{2.1}$$
$$= \mathbb{P}[s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, s_{t-2}, a_{t-2}, ...] \tag{2.2}$$

2.2. The Principle of Optimality

According to Bellman's principle of optimality, the solution of some optimization problems can be put together from the solution of subproblems. This is the basis for all dynamic programming algorithms. If an agent chooses the optimal action in a state s_t and all states it visits afterwards (s_k, $k = t+1, t+2, ..., T$) , the resulting trajectory is the optimal one from s_t to the terminal state s_T.

The principle of optimality can be expressed by the Bellman Equation, which is shown in section 2.9. Prior to introducing the Bellman Equation, a few necessary concepts and terms are explained in the following sections.

(Bsp. Fibonacci Folge und/oder shortest path)

2.3. The Agent - Environment System

In Reinforcement Learning scenarios, an agent interacts with its environment and thereby learns how to maximize some sort of scalar reward signal. The agent aims to maximize its total reward by choosing - at each time step - the best possible action a from a set of actions $\mathcal{A}$. The agent has no prior knowledge about the environment and about how to behave in an optimal way. Instead, it learns by interacting with its environment and the information it receives during training. Before each step, the agent is in a state s_t, chooses an action a_t and receives a scalar reward r_t and an observation o_t representing the next state s_{t+1}. The process is repeated, then. Figure 2.1 illustrates what happens at each time step. Note that, unlike in this thesis, a timestep is denoted by an index k instead of t. The observation o_t can just be s_{t+1}. However, in partially observable environments, it might only be a part of s_{t+1} or any information regarding the true state s_{t+1}. In this work, o_t is equal to s_{t+1} (i.e. the agent has complete knowledge about his state at any time).

The reward r_t the agent receives at each step is determined by a reward function $\mathcal{R}(s, a)$. The reward function maps from states (and sometimes actions) to a scalar reward and characterizes the mission target. The choice of a reward function is crucial for the success of any RL-based trajectory optimization algorithm. The only way to "tell" the agent, what his goal is, is by defining the reward function.

Generally, a reward function should guide the agent to the target. There are, however, restrictions on what a reward function should contain. If too much information about the problem is put into the reward function, this might lead to a suboptimal result. If, for example, a user trying to solve an MDP thinks a specific terminal state might be better than the others, he might be tempted to increase the final reward for reaching this - seemingly optimal - terminal state. But any (unnecessary) assumption, that is put in the reward function, can restrict the agent's ability to find an optimal trajectory on his own. A false assumption can sometimes even jeopardize convergence of the whole algorithm. Therefore, a good reward function should only give a reward for reaching a target state and penalize every step the agent does, if time is crucial. It can also penalize any step that leads to an undesired terminal state (in this work, touching the ground is such a case). This way, the agent can find a good trajectory without restrictions. More on the reward function used here can be found in section 4.1.3.

2.4. Model Based and Model Free Learning

MDP solution methods can be divided into model-free and model-based methods. In model-based methods, the agent uses a model to predict the reaction of the environment to any of his actions. An environment model typically takes a state-action pair and returns the next state and next reward. If the environment is discrete and stochastic,

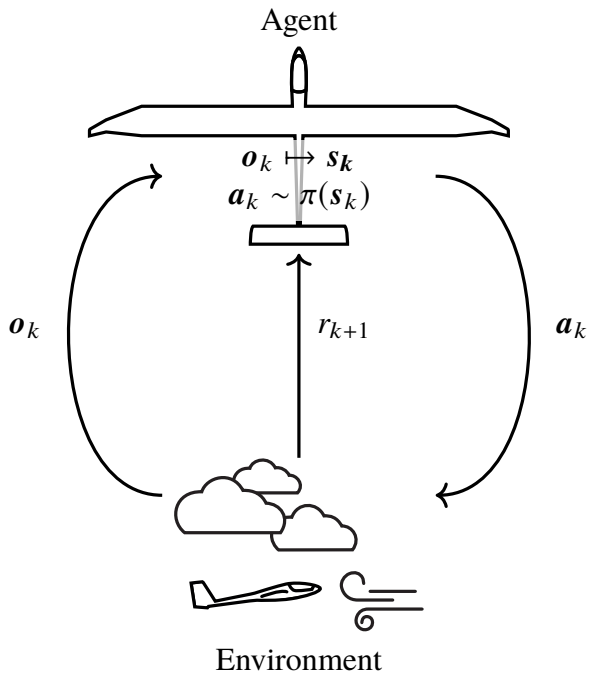

Figure 2.1.: The Agent-Environment-System [8]

there are multiple possible next states and rewards.[1] A *sample model* returns one of the possibilities, whereas a *distribution model* returns some representation of all possible next states and rewards. To distinguish experience from a model from real experience, the results from a model are referred to as *simulated experience* [11, section 8.1].

Regardless of the type, all models approximate what might happen to the agent in future time steps. Whether the experience gathered through a model is useful obviously depends on its quality. Every model adds complexity to an algorithm. The more complexity, the higher the cost of implementing and updating it. Model free methods are generally simpler to implement and more scalable.

In model free solution methods, there is no model of the environment. The agent is not explicitly aware of the environment behavior. Instead, the policy (and value function) are directly affected by the experience of the agent. Sutton and Barto refer to model free methods as some sort of "trial and error" (c.f. [11, section 1.5]). If the goal is to maximize the expected return, a model might not be needed at all. In TRPO, for example, the policy is optimized by maximizing the expected return with respect to the policy parameters θ by sampling trajectories through the state space and utilizing a gradient based optimization method. This is simpler than learning a model of the environment and using it to plan.

A model of the environment is - in principle - not required for successful training. It

[1] In a continuous stochastic environment, there would be a distribution of successor states and rewards.

can however be used to gather simulated experience (c.f. section 2.4) and therefore reduce training time on a real agent like a robot. Although imperfect, information from a model can be useful in such cases where training time is restricted which particularly holds true for robots flying outdoors.

For more information on how machine learning algorithms can be classified, please refer to chapter 2.11, which also contains one classification example in Fig. 2.2.

2.5. Agent

The agent in an MDP consists of at least one of the following parts.

- Policy:
  A mapping from states to actions that tells the agent what to do (c.f. section 2.7)
- Value Function:
  A mapping from states (or state-action pairs) to values (c.f. section 2.8)
- Model:
  A model of the environment that might be updated by the agent's experience

Most agents implement a policy. They also can have a value function, like in actor-critic algorithms, where the value function is used for so called bootstrapping [2].

$$\mathbb{E}[G_t|s_t = s] = \mathbb{E}[r_{t+1} + \gamma G_{t+1}|s_t = s, a_t = a] \tag{2.3}$$
$$= \mathbb{E}[r_{t+1} + \gamma V(s_{t+1})|s_t = s, a_t = a] \tag{2.4}$$

Instead of sampling a complete episode, bootstrapping makes it possible to learn from incomplete episodes.

2.6. Return

The return G_t at a time step t is the cumulative reward r_t at each step from the state s_t onwards until reaching a terminal state:

$$G_t = \sum_{k=0}^{T-t-1} \gamma^k r_{t+k+1} \tag{2.5}$$

[2] Bootstrapping is a general term, that applies to methods, that make use of previous results in order to obtain an new result. In this case, it means utilizing the Bellman Expectation Equation to estimate the return G_t by doing a one step lookahead. The state value $V(s)$ is then updated utilizing $V(s')$, which can be seen as a "previous" result.

If $\gamma \in (0, 1]$ is close to zero, immediate rewards are weighted stronger. If γ is close to one, rewards in the distant future are weighted likewise, making decisions more far sighted. In infinite horizon problems, where there is no terminal state, γ must be less than one in order to avoid infinite returns. In finite horizon (i.e. episodic) MDPs, γ is usually close to, or exactly one.

Each decision at a given state s_t also affects what the next state s_{t+1} is and therefore the next reward. More on the reward and what role it plays in Dynamic Programming is explained in section 4.1.3.

2.7. Policy

In reinforcement learning problems, a policy is usually the part that represents the agent. It contains all the information needed to decide what action to choose in each state the agent visits. A policy is a mapping from states to actions.

Generally, a policy is a mapping from states to actions. A policy can also be stochastic, in which case it maps from states and actions to a number between 0 and 1, which represents the probability to pick a when in s.

$$\pi : \mathcal{S} \times \mathcal{A} \mapsto [0, 1] \tag{2.6}$$

If the policy is deterministic, only one a_t out of $\mathcal{A}$ has a probability of one with all other actions assigned to probability zero. In any policy, the sum of all possible action-probabilities must be one.

$$\pi(a|s) = \mathbb{P}[a_t = a|s_t = s] \tag{2.7}$$

This yields equations 2.8 and 2.9 for the state transition probability and the reward function under policy π. The reward function is explained in section 4.1.3.

$$\mathcal{P}_\pi(s, s') = \sum_a \pi(a|s)\mathcal{P}(s'|s, a) \tag{2.8}$$

$$\mathcal{R}_\pi(s) = \sum_a \pi(a|s)\mathcal{R}(s, a) \tag{2.9}$$

At each timestep, the agent receives a state observation o_{t-1} from the environment, passes it to the policy and the policy returns an action a_t. This action is taken and the agent gets a new observation o_t and a reward r_t. This process is repeated until the agent reaches a terminal state.

A policy can be implemented in many different ways. The simplest option is a table where each field contains the action for one state or - if the policy is stochastic - all possible actions and their respective probabilities. Other examples include linear combinations of the inputs and linear combinations of basis functions. The most popular way to implement a policy is - however - through an artificial neural network (ANN). More information about neural networks is given in chapter 3.

2.8. Value Functions

Some RL-algorithms try to find a policy directly from experience. Others, like Q-learning, additionally utilize a so called value function. A value function maps from states or state-action pairs to their values. In this context, the value of a state or an action is a measure of how useful it is for the agent to visit the respective state or choosing the respective action. There are two types of value functions, state value functions and action value functions. The state value $V(s)$ is the expected total return from state s_t onwards until the episode terminates. The action value also takes into account what the agent chooses to do in state s_t. It is the expected total return from state s_t when choosing action a_t.

$$V(s) = \mathbb{E}[G_t|s_t = s] \tag{2.10}$$

$$Q(a|s) = \mathbb{E}[G_t|s_t = s, a_t = a] \tag{2.11}$$

In the following sections, the action space is regarded finite for simplicity. The state value function 2.10 can be expressed in terms of the action value function 2.11:

$$V_\pi(s) = \sum_a \pi(a|s) Q_\pi(s, a) \tag{2.12}$$

Similarly, the action value function can be expressed by means of the state value function:

$$Q_\pi(s, a) = \mathcal{R}(s, a) + \gamma \sum_{s'} \mathcal{P}(s'|s, a) V_\pi(s') \tag{2.13}$$

If an agent has found the optimal value function V^* or Q^* of an MDP, the optimal policy π^* can be derived directly by acting greedily with respect to the value function at each state.

Like a policy, a value function may be implemented as a table, a linear combination of the input features, a combination of basis functions, or an artificial neural network.

2.9. The Bellman Equation

One way to find an optimal trajectory is by utilizing a value function $V(s)$, which maps from a state (or a state-action pair) to a state (or action) value. It tells the agent how good it is to be in a specific state in terms of the expected return. The optimal value function tells the agent the maximum return he can collect from every state.

According to the Bellman Expectation Equation, every trajectory through the state space can be decomposed into two parts: the next step and the rest of the path. This principle also applies to value functions, which are introduced in the following sections.

The state value function (c.f. Eq. 2.10) can be written as a weighted sum of the action values, each multiplied with the probability to take the respective action according to

the policy. Replacing $Q_\pi(s,a)$ in equation 2.12 with equation 2.13 yields equation 2.15, which is known as the *Bellman Expectation Equation*:

$$V_\pi(s) = \sum_a \pi(a|s) Q_\pi(s,a) \tag{2.14}$$

$$= \sum_a \pi(a|s) \sum_r \sum_{s'} \mathcal{P}(s', r|s, a)[r + \gamma V_\pi(s')] \tag{2.15}$$

In a deterministic environment, equation 2.15 becomes

$$V_\pi(s) = \sum_a \pi(a|s)[r + \gamma V_\pi(s')] \tag{2.16}$$

The Bellman Expectation Equation can be used to update a state value by looking at the state values of its successor states and weighting them by the probabilities to reach each successor state (c.f. equations 2.29 and 2.30).

Instead of a weighted sum of rewards and successor values, like in equation 2.15, one can take only the action that has the maximum action value. The resulting state value $V_*(s)$ is called the *optimal state value*. It is the maximum expected return from state s onwards. This yields the following equations.

$$V_*(s) = \max_{a \in \mathcal{A}(s)} Q_{\pi_*}(s,a) \tag{2.17}$$

$$= \max_a \mathbb{E}_{\pi_*}[G_t | s_t = s, a_t = a] \tag{2.18}$$

$$= \max_a \mathbb{E}_{\pi_*}[r_{t+1} + \gamma G_{t+1} | s_t = s, a_t = a] \tag{2.19}$$

$$= \max_a \mathbb{E}[r_{t+1} + \gamma V_*(s_{t+1}) | s_t = s, a_t = a] \tag{2.20}$$

$$= \max_a \sum_{s',r} \mathcal{P}(s', r|s, a)[r + \gamma V_*(s')] \tag{2.21}$$

Again, if the environment is deterministic, equation 2.21 can be simplified:

$$V_*(s) = \max_a [r + \gamma V_*(s')] \tag{2.22}$$

All equations from 2.17 to 2.22 are alternative ways to express the *Bellman Optimality Equation*. It states that the value of a state under an optimal policy must equal the expected return for the best action in that state [11]. Intuitively it is obvious that the optimal policy must pick the best action in each state. In Value Iteration, the *Bellman Optimality Equation* is used as an update rule. See chapter 2.11.2 for more details.

2.10. Solving an MDP

A Markov Decision Process is a series of decision problems, where each decision has an impact on the total return the agent receives. In most RL-problems, the goal is to

maximize the total return from a given initial state. This is done by trying to find the optimal policy π^* which yields the maximum expected return from each state. π^* can be achieved directly or by finding the optimal value function, i.e. the mapping from each state or state-action-pair to its true value V^*. The definition of a state value is given in section 2.8 If π^* is found for a specific MDP, the MDP is solved.

2.11. Dynamic Programming

empty

Figure 2.2.: Classification of Machine Learning algorithms

This thesis deals with Dynamic Programming for policy optimization. More specifically, policy iteration and value iteration are applied to a trajectory optimization problem. In a prior work on this topic, TRPO and A3C were applied to a 2D trajectory optimization problem [12]. Both algorithms deal with the complete environment, i.e. the glider dynamics and the stochastic wind distribution, at the same time. Whereas the wind distribution is unknown a priori, the glider dynamics are well known. Therefore, a DP algorithm can be applied to any scenario with calm air. The resulting optimal policy π^*_{calm} then can be used as a starting point for TRPO in scenarios with unknown wind conditions.

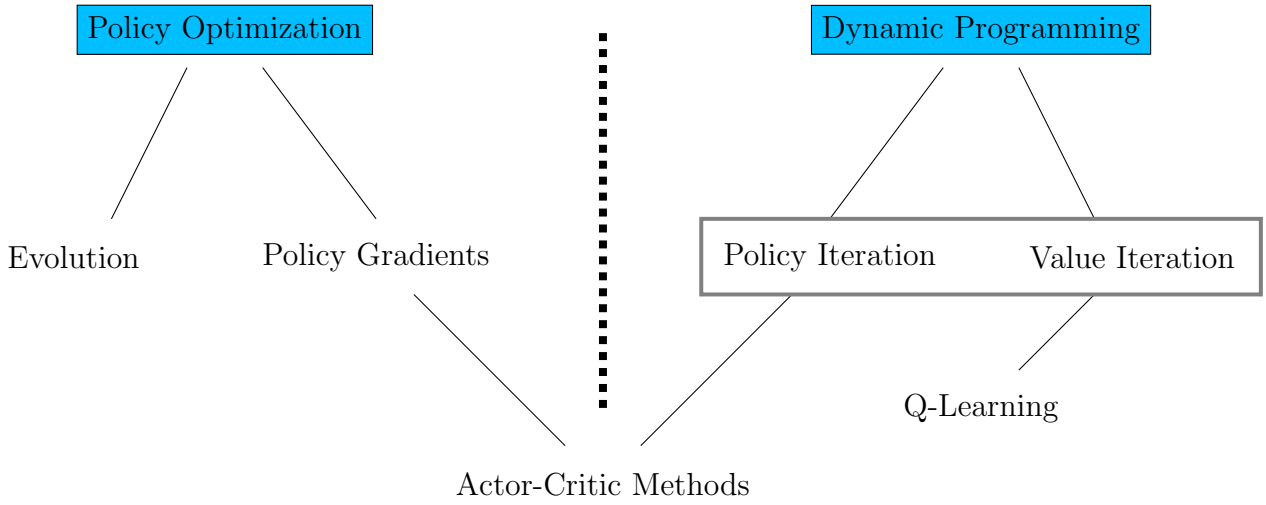

Figure 2.3.: MDP solution methods, adopted from [2]]

Dynamic programming is a general term, that is used for algorithms that solve a specific type of problem. Those problems can be decomposed into subproblems and solved by composing the solutions of those subproblems. If the subproblems are very similar, the

solution of one subproblem can be used to calculate another one. In such cases, dynamic programming is applicable. In MDPs, these subproblems are similar in the sense that the Bellman Expectation Equation and the Bellman Optimality Equation hold true in all states, i.e. each state (and action) value is connected to its successor values by the Bellman Equations.

Also, the optimal action is found in all states the same way, as can be seen in section 2.11.2. If the algorithm is successful, the policy is optimal according to section 2.2. Unlike with TRPO, optimization is not done through sampling complete trajectories, but by iterating the state-values and policy for each state independently. After one sweep over the state space, the value function and policy are updated and the process is repeated.

In theory, starting TRPO or A3C with a policy that is already optimal with respect to the scenario in calm air should accelerate training with updrafts because the information about how to behave in a situation with no wind is already incorporated in the policy.

2.11.1. Approximate Dynamic Programming

Dynamic programming has been utilized to solve problems in various disciplines since its invention. Most of the applications were in finance, computer science and control theory. The latter is also the topic of this thesis. Unfortunately, the descriptions of the relationship between RL and DP vary from discipline to discipline and seem inconsistent at first. Therefore, this section contains some information about important keywords and methods in DP and RL and tries to bring the different viewpoints of control theory, computer science and finance together. For each discipline, the relevant nomenclature is shown and some sources containing further information are given. Then, the seemingly inconsistent viewpoints are brought together.

The term Approximate Dynamic Programming (ADP) often comes up in the context of machine learning. (Daran tippe ich gerade, mir geht es vor allem um den Aufbau von Kapitel 2.)

2.11.2. Types of Dynamic Programming Algorithms

Policy Evaluation

In Policy Evaluation, the goal is to find an approximation of the value function V_π that corresponds to a given policy π. The value of a state is defined as the expected total return from that state onwards until the episode terminates. Obviously, the expectation in equation 2.25 depends on the actions that are taken at each step and therefore on the policy that is evaluated. The state value that corresponds to a specific policy π is denoted by $V_\pi(s)$.

$$V_\pi(s_t) = \mathbb{E}[G_t|s_t = s] \tag{2.23}$$
$$= \mathbb{E}\left[\sum_{k=0}^{T-t-1} \gamma^k r_{t+k+1}|s_t = s\right] \tag{2.24}$$
$$= \mathbb{E}[r_{t+1} + \gamma^1 r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + ... + \gamma^{T-t-1} r_T|s_t = s] \tag{2.25}$$

With a stochastic discrete policy, equation 2.25 becomes

$$V_\pi(s_t) = \sum_a \pi(a|s)(\mathcal{R}(s,a) + \gamma V_\pi(s')) \tag{2.26}$$

Recall the definition of a state value $V(s) = \mathbb{E}[G_t|s_t = s]$ from Eq. 2.10. This equation can be used as a mapping to update the value of s_t with the expected return according to the current policy.

$$V_\pi(s_t) \leftarrow\!\shortmid \mathbb{E}[\sum_{k=0}^{T-t-1} \gamma^k r_{t+k+1}|s_t = s] \tag{2.27}$$

In Eq. 2.25, all terms expect the first one can be replaced by the expected return from s_{t+1} onwards:

$$V_\pi(s_t) \leftarrow\!\shortmid \mathbb{E}[r_{t+1} + \gamma G_{t+1}|s_t = s] \tag{2.28}$$
$$= r_{t+1} + \gamma V_\pi(s_{t+1}) \tag{2.29}$$

With a discrete stochastic policy, equation 2.29 becomes

$$V_\pi(s_t) \leftarrow\!\shortmid \sum_a \pi(a|s)(r_t + \gamma V_\pi(s')) \tag{2.30}$$

In a state space with a finite set of states $\mathcal{S}$, writing down Eq. 2.30 for each state results in a system of $|\mathcal{S}|$ linear equations that can be solved for $V(s)$. If the number of states is very large, this is not very efficient.

Instead, 2.25 is usually solved iteratively. The simplest way is to perform a one step lookahead from each state to get r_t and $V(s_{t+t})$ from the current estimate of the state value function. Once all values are calculated, the values of all states are updated simultaneously [3]. Unlike other RL-algorithms, the state values are not updated along a trajectory. Instead, each state is updated independently.

[3] There are variants of Policy Iteration, where the updates are performed asynchronously. This means that, for example, all neighbors of a terminal state are updated first. After that, the states, that lie next to these states, are updated, and so on. They usually converge faster, but at the expense of implementation complexity.

Each set of state values approximates the true value function of the problem better than the previous set of estimates. Every state value converges to the true state value under the given policy. It is not efficient to wait until the values have fully converged, i.e. the true value function V_π is reached. Instead the process is repeated until the maximum absolute change in values lies beneath a certain threshold ϵ_V. This maximum absolute change can be expressed by the supremum-norm $||(\cdot)||_\infty$ (c.f. section 2.11.3). Algorithm 1 shows a possible implementation of iterative policy evaluation. In practice, there can be an upper bound for the number of evaluation steps.

$$||V_k - V_{k-1}||_\infty < \epsilon_V \tag{2.31}$$

Algorithm 1 Iterative policy evaluation

```
function Policy Evaluation
    Initialize V(s) = 0 ∀ s ∈ S and load policy π.
    while true do
        for all s ∈ S do
            V_{π,new}(s) ↤ r + γ V_{π,old}(s')
        end for
        if ||V_new − V_old||_∞ ≤ ε_V then
            break
        end if
    end while
end function
```

Policy Iteration

Policy iteration is an iterative way to calculate an estimate of the optimal policy of an MDP. Starting with an arbitrary policy and value function, one iteration of policy evaluation is performed to get an estimate of V_π. After that, after that, the policy is replaced by the greedy policy w.r.t. to the new value function estimate V_π. This alternating process of Policy Evaluation and Policy Improvement is repeated until the policy is satisfactory.

$$a_{\text{greedy}}(s_t) = \underset{a}{\text{argmax}}[Q(s_t, a_t)] \tag{2.32}$$

For a deterministic, greedy policy π_{greedy}, this yields:

$$\pi(a_{\text{greedy}}|s_t) = \mathbb{P}[a_t = a_{\text{greedy}}(s_t)|s_t] \tag{2.33}$$
$$= 1 \tag{2.34}$$

and for a stochastic policy π:

$$\mathbb{E}[\pi(a_t|s_t)] \leftarrow\!\shortmid\, a_{\text{greedy}} \tag{2.35}$$

As mentioned in subsection 2.11.2, each policy evaluation step usually ends if the maximum difference between two subsequent values of the same state are sufficiently close.

There is a version of Policy Iteration where only one value update is done before updating the policy. This algorithm is called Optimistic Policy Iteration (OPI). Although the first estimate of the value function is only a rough approximation, OPI also converges to the optimal value function and policy. The following diagram illustrates the policy iteration scheme.

$$\pi_0 \overset{evaluate}{\longrightarrow} V_1 \overset{improve}{\longrightarrow} \pi_1 \overset{evaluate}{\longrightarrow} V_2 \overset{improve}{\longrightarrow} \pi_2 \overset{evaluate}{\longrightarrow} \ldots \overset{evaluate}{\longrightarrow} V_* \overset{improve}{\longrightarrow} \pi_*$$

Another way to picture policy iteration is shown in figure 2.4. Every PI-step brings the Value Function $V(s)$ closer to $V_*(s)$ and the policy π closer to π_*.

Policy iteration can be seen as two competing processes. On one hand, every policy evaluation step changes v, which leads to a change in the greedy policy. The previous greedy policy is therefore no longer greedy after the evaluation step. On the other hand, every policy improvement step makes the policy greedy w.r.t the current value function. This typically changes the value function, so the value function before the policy update is no longer the correct one (cf. [11, section 4.6]).

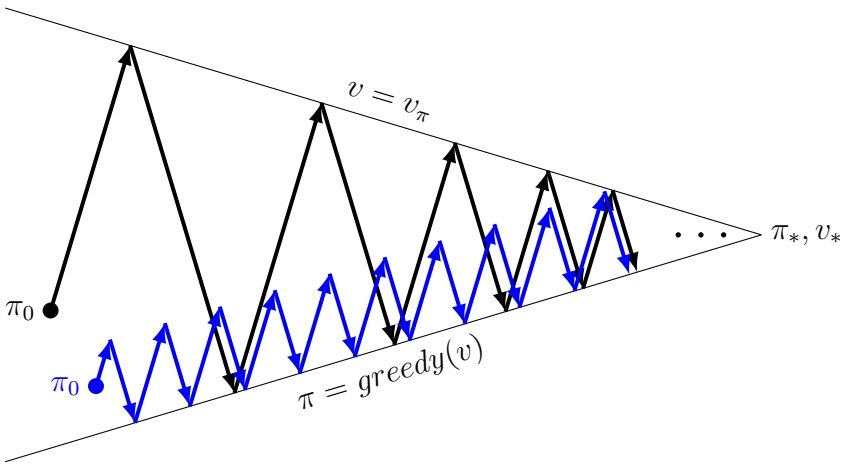

Figure 2.4.: The policy iteration algorithm (adopted from [11, section 4.6]

The black arrows in Fig. 2.4 refer to a policy iteration algorithm, that calculates v_π exactly in every evaluation step. This typically takes long for one iteration, but requires less iterations than optimistic policy iteration. The latter is what the blue arrows refer to. Each policy evaluation step only yields a rough estimate of v_π and takes less time than exact evaluation. OPI however usually takes more iterations than exact PI.

The performance of both algorithms on the trajectory optimization problem is compared in chapter 5.

In a discrete MDP, there is a finite number of states and actions. At each state $s \in \mathcal{S}$, the agent can choose the action a from a set $\mathcal{A}$ of possible actions. In such a scenario, there exist $|\mathcal{A}|^{|\mathcal{S}|}$ different policies. It therefore takes $|\mathcal{A}|^{|\mathcal{S}|}$ iterations to find the optimal policy assuming no policy occurs twice. If that was the case, this policy would have to occur in two subsequent iterations and therefore be the fixed point π_* of the policy sequence generated by PI.

As the results in chapter 5 show, PI converges much faster in practice than the upper bound $|\mathcal{A}|^{|\mathcal{S}|}$ may suggest. Algorithm 2 shows one way to implement generalized policy iteration. The algorithm alternates between PE and policy improvement until none of the actions change from one iteration to the next. The pseudo code for optimistic policy iteration can be found in algorithm 4 in appendix A.4.

Value Iteration

Value iteration is similar to policy iteration, but instead of calculating a new policy at each iteration step, the state values $V(s)$ are directly updated with the maximum possible successor value that is achievable starting from s. This yields a sequence of value functions, each one being a better estimate of V_* than its predecessor. Each of the intermediate value functions is abstract in so far as there does not have to exist an explicit policy corresponding to it (c.f. [10, lecture 3]). In a possibly discounted MDP with a deterministic environment, the update rule for each state value can be seen in equation 2.36.

$$V(s_t) \leftarrow\!\shortmid \max_a [r_{t+1} + \gamma V(s_{t+1}) | s_t = s] \tag{2.36}$$

In principle, this is equivalent to synchronous OPI, where after one state value update at each state, the policy is replaced by the greedy policy with respect to the new value function. The only difference is, that value iteration does not output an intermediate policy at each iteration. Instead, it only iterates value functions.

$$V_1 \longrightarrow V_2 \longrightarrow V_3 \longrightarrow ... \longrightarrow V_*$$

In practice, a stopping criterion is introduced to limit calculation time. In this work, iteration is stopped if $||V_k - V_{k-1}||_\infty < \epsilon_V$, identical to equation 2.31. The last iterate V_n is considered an approximation of V_*. If V_n is close enough to the true optimal value function V_*, the (approximately) optimal policy can be calculated in the end by acting greedily with respect to V_n. Recall that, if Value Iteration is stopped too early, there might not even be a real policy that corresponds to the last value function iterate V_n [10, lecture 3]. A possible implementation of value iteration is shown in algorithm 3.

2.11.3. The Contraction Mapping Theorem

A contraction mapping $f : M \to M$ on a metric space M has the following property:

$$|f(x_1) - f(x_2)| \leq \gamma |x_1 - x_2| \tag{2.37}$$

with $x_1, x_2 \in M$ and $\gamma \in [0, 1)$. In this context, $|(\cdot)|$ denotes an arbitrary metric on M.

The term contraction reflects the fact that, graphically speaking, a contraction mapping reduces the distance between x_1 and x_2.

Discounted MDPs

Policy Evaluation in discounted MDPs is a contraction mapping. This result can be obtained in few steps. The mapping in equation 2.30 and 2.38 is known as the *Bellman Operator*. It maps from a state value function to a state value function. If the Bellman Operator is applied multiple times, this can be expressed by $T^n(v)$.

$$T_\pi(v(s')) = \mathcal{R}_\pi + \gamma \mathcal{P}_\pi v(s') \tag{2.38}$$

If the Bellman Operator is applied to two state value functions $V_1(s)$ and $V_2(s)$, it reduces the distance between the two in value function space. This distance can be expressed by the supremum norm, which is defined as follows:

$$||V||_\infty = \max_{s \in \mathcal{S}} |V(s)| \tag{2.39}$$

The supremum norm of $V_1 - V_2$ is therefore

$$||V_1(s) - V_2(s)||_\infty = \max_{s \in \mathcal{S}} |V_1(s) - V_2(s)| \tag{2.40}$$

which is the biggest absolute difference of values of the same state in the state space. If the Bellman Operator is applied to both V_1 and V_2, the contraction property proof is straightforward. For simplicity, the arguments of $V_1(s)$ and $V_2(s)$ are omitted in the following equations.

$$\begin{aligned}
||T_\pi(V_1) - T_\pi(V_2)||_\infty &= ||(\mathcal{R}_\pi + \gamma \mathcal{P}_\pi V_1) - (\mathcal{R}_\pi + \gamma \mathcal{P}_\pi V_2)||_\infty && (2.41)\\
&= ||\gamma \, \mathcal{P}_\pi \, (V_1 - V_2)||_\infty && (2.42)\\
&\leq ||\gamma \mathcal{P}_\pi|| V_1 - V_2||_\infty||_\infty && (2.43)\\
&\leq ||\gamma|| V_1 - V_2||_\infty||_\infty && (2.44)\\
&= \gamma ||V_1 - V_2||_\infty && (2.45)
\end{aligned}$$

Whenever s' happens to be the state where V_1 and V_2 differ the most, equality holds in line 2.43. If not, applying the supremum norm to $V_1(s') - V_2(s')$ increases the value of the expression. $\mathcal{P}_\pi$ must always be smaller than or equal to one. Similar to line 2.43, equality in line 2.44 holds only if $\mathcal{P}_\pi = 1$. Replacing $\mathcal{P}_\pi$ by one in line 2.43 yields line 2.44. The left side of line 2.41 must thus be less than line 2.45 which proves the contraction property of the Bellman Operator in discounted MDPs.

Undiscounted MDPs

An MDP, where the discount factor γ is one, is called an undiscounted MDP. In such an MDP, the Bellman Operator looks like in equation 2.46.

$$T_\pi(v(s)) = \mathcal{R}_\pi + \mathcal{P}_\pi V(s') \tag{2.46}$$

If γ is one, the above proof of the contraction property does not hold anymore. It can however be proven, that the Bellman Operator still contracts. In the undiscounted case, this takes more than one step. Equation 2.47 shows the n-step contraction property. For simplicity, the index π and the argument s are omitted.

$$||T^{(n)}(V_1) - T^{(n)}(V_2)||_\infty \leq ||V_1 - V_2||_\infty \tag{2.47}$$

Equation 2.47 means, that applying the Bellman Operator n times to two value functions brings them closer together in value function space. As a result, the Bellman Operator also leads to $V_1 \approx V_2$ eventually also in undiscounted MDPs.

A proof, that policy iteration and value iteration also converge in undiscounted MDPs, can be found in [],[] and [] (die trage ich in der nächsten Fassung noch ein).

2.12. Exploiting the specific Problem Structure

According to [9], there is no such thing as a universal dynamic programming algorithm that is capable of solving various problems efficiently. Instead, making use of the structure of a problem is often vital in order to reduce computation time or make DP feasible in the first place. The following paragraph illustrates how this can be done in the case of path planning through value iteration.

In numerics, making use of new information as soon as it is available often reduces calculation time significantly. In value iteration, this can mean to update a state value as soon as a new value has been calculated. This way, subsequent calculations can make use of more recent data than in the case of synchronous updates[4]. The most efficient way to exploit the problem structure in this work (cf. 4) is to go backwards through the state space. The states directly next to the terminal states are updated first and simultaneously. After that, the layer[5] next to these states is updated and so on. This way, the final reward can propagate to the start state in one iteration, i.e. one backward pass through the state space. As the value iteration algorithm in this work also considers movement (i.e. movement away from the goal), it typically takes about a thi

[4]Recall that, in synchronous DP, the state values are updated at the same time after one complete sweep over the state space. If the start state s_0 is 50 steps away from the target state s_T, it takes 50 iterations (i.e. complete sweeps over the state space) to find a trajectory from s_0 to s_T.

[5]In this context, one layer is one set of points that have the same x-coordinate. This set forms a vertical "layer" of states that have the same distance to the goal.

Algorithm 2 Generalized policy iteration

function GENERALIZED POLICY ITERATION
  Initialize $V(s) = 0 \; \forall \; s \in \mathcal{S}$
  Load arbitrary initial policy π_0.
  Initialize m, ϵ_V
  while true **do**
    function POLICY EVALUATION
      while true **do**
        for all $s \in \mathcal{S}$ **do**
          $V_{\pi,new}(s) \leftarrow\!\shortmid r + \gamma V_{\pi,old}(s')$
        end for
        if $||V_{new} - V_{old}||_\infty \leq \epsilon_V$ **then**
          break
        end if
      end while
    end function
    function POLICY IMPROVEMENT
      for all $s \in \mathcal{S}$ **do**
        sample m actions $a_n(s)$
        for all a_n **do**
          $Q(s, a_n) = r + \gamma V(s')$
        end for
        $a_{greedy}(s) = \underset{a_n}{\text{argmax}}[Q(s, a_n)]$
      end for
    end function

    train policy on $a_{greedy,new}$ to obtain π_{new}
    if $||a_{greedy,new} - a_{greedy,old}||_\infty = 0$ **then**
      break
    end if
  end while

end function

Algorithm 3 Value iteration

function VALUE ITERATION
  Initialize $V(s) = 0 \; \forall \; s \in \mathcal{S}$
  Initialize m, ϵ_V
  while true **do**
    for all $s \in \mathcal{S}$ **do**
      sample m actions $a_n(s)$
      for all a_n **do**
        $Q(s, a_n) = r + \gamma V(s')$
      end for
      $V_{new}(s) = \max_{a_n}[Q(s, a_n)]$
    end for
    if $||V_{new} - V_{old}||_\infty \leq \epsilon_V$ **then**
      break
    end if
  end while
  function POLICY IMPROVEMENT
    for all $s \in \mathcal{S}$ **do**
      sample m actions $a_n(s)$
      for all a_n **do**
        $Q(s, a_n) = r + \gamma V(s')$
      end for
      $a_{greedy}(s) = \operatorname{argmax}_{a_n}[Q(s, a_n)]$
    end for
  end function
  train policy on the latest $a_{greedy,new}$ to obtain an estimate of π_*
end function

3. Function Approximation

A common problem in engineering is developing a model from measurement data. Such a model makes it possible to generalize from the given data to new data, for which a real measurement does not exist. The process of developing a model is often combined with function approximation. The goal here is to find a mapping as a linear combination of elements, which constitute a basis from functions, that approximates a given behavior, also called a target function, as close as possible.

Mathematically, function approximation is the process of finding a mapping from inputs x to outputs y. Usually, a set $\mathcal{X} = \{x_{fit,1}, x_{fit,2}, ...\}$ of inputs and a set $\mathcal{Y} = \{y_{fit,1}, y_{fit,2}, ...\}$ corresponding desired outputs is given and the goal is to find a function that returns a value close to $y_{fit,i}$ for each $x_{fit,i}$.

There are two types of function approximation. In some cases, knowledge about the specific problem can help to find suitable base functions. If, for example, a mapping is known to be linear ($y = a_0 + a_1 x$), the function approximation process degenerates to finding the two coefficients a_0 and a_1. The same applies to quadratic, trigonometric or exponential target functions or linear combinations of them. If, however, the target function is not known, function approximation is more elaborate. One can try multiple target functions or combine them in order to find one, that fits best, or use a *universal function approximator* like an artificial neural network.

3.1. Tables

The simplest way to express a discrete mapping is by a table where each input/output pair is represented by a cell of the table. The number of inputs and corresponding outputs in a table is inherently finite. The set of states and actions in a continuous MDP is however infinite, so the policy and value functions must also be continuous. If a discrete mapping like a table is used as a representation of a policy or value function in a continuous MDP, it will, at some point, be confronted with unknown inputs. The inability to deal with unknown inputs is a big disadvantage of tables. There are, however, ways to overcome this weakness.

Nearest Neighbor

The most straightforward way to obtain an output for an unknown input is to look for the table entry that is closest to the unknown input. This yields a piecewise constant mapping from inputs to outputs and enables the table to generalize. Whether this suffices

as an approximation for a continuous mapping f heavily depends on the properties of f . If there are large differences between the output of two subsequent inputs, the error can be very high.

Linear Interpolation

Another way to obtain outputs for unknown inputs is to interpolate linearly. In a first step, the two nearest known inputs x_0 and x_1 and their respective outputs y_0 and y_1 are looked up (note that $x_0 \leq x \leq x_1$ in this case). In the second step, the output is calculated by the following formula:

$$y_{interp} = y_0 \, (1 - \gamma) + y_1 \, \gamma, \tag{3.1}$$

where $\gamma = \frac{x - x_0}{x_1 - x_0}$.

One advantage of linear interpolation of table values is that the output is at least continuous, though not differentiable at the known input points.

3.2. Artificial Neural Networks

The structure of artificial neural networks resembles the structure of a human brain according to the current state of research [7, section 1.1]. The human brain consists of a very high number of neurons, each receiving signals from other neurons, processing them and itself sending a signal to other neurons if certain conditions are met. Each neuron is very simple, it is the high number of them and the fact that they all work in parallel that makes the human brain so powerful.

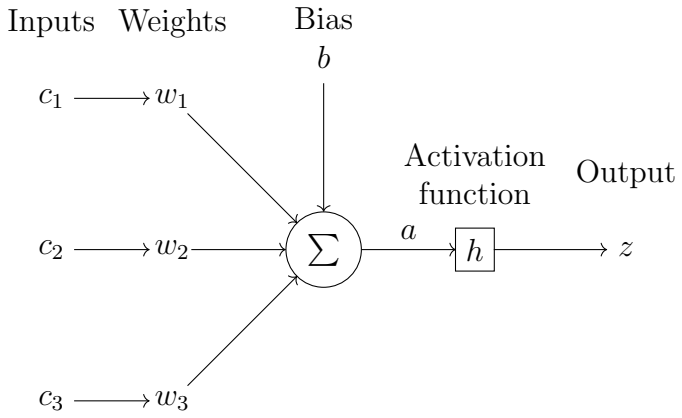

Figure 3.1.: A neuron in an artificial neural network

A neuron in an artificial neural network looks like what is depicted by Fig. 3.1. It takes an arbitrary number of inputs c_i, multiplies each input by the corresponding input weight

w_i, adds a bias and applies to the result a a non-linear function, the so called activation function h, yielding z, the neuron output. This output can either serve as an input to another neuron or as the network output. Multiple activation functions are common, most of which are somewhat s-shaped. These activation functions map the set of real numbers to the range between -1 and 1 or sometimes between 0 and 1, limiting the neuron output and therefore its influence on the output of the ANN network. Examples include the step function, ($z = 1$ if $x \geq x_0$, $z = 0$ otherwise), the rectifier nonlinearity ($h(x) = \max(0, x)$), the sigmoid function ($h(x) = 1/(1 + e^{-x})$) and the $\tanh(\cdot)$-activation function ($h(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$). Nonlinearity is essential here, because if linear activation functions are used, each neuron output is a linear function of its inputs. Any combination of linear mappings is itself linear. So if a deep neural network has only linear activation functions, the ANN output is always a linear function of the inputs, regardless of the number of hidden layers. Therefore, a deep neural network, where all activation functions are linear, can be replaced with a network, that has only one hidden layer (or something that is even simpler). The computational expense of dealing with a deep ANN is - however - very high compared to one with a single hidden layer. A deep ANN with only linear activation functions therefore does not make sense.

In an artificial neural network, the neurons are arranged in layers. The ANNs in this work are all fully connected feed forward networks, also known as Multi Layer Perceptrons (MLPs). Each neuron in a given layer k gets an input from each neuron in the previous layer $k - 1$ and outputs a signal to each neuron in the next layer $k + 1$ (c.f. Fig. 3.1).

If a policy or value function is represented by an artificial neural network (ANN), the output for each state (or state-action-pair) is determined by the weights and biases of the network and the activation functions. As the latter are usually given or predetermined (and not trained), only the weights and biases are usually regarded as parameters. One set of weights and biases forms the *parameter vector* θ of the neural network. If, for example, a stochastic policy is parametrized by θ and represented by a neural net with k layers, the notation is as follows:

$$\pi_\theta(a|s) = \mathbb{P}[a|s, \theta] \tag{3.2}$$

with

$$\theta = \{\boldsymbol{W}^{(1)}, \boldsymbol{b}^{(1)}, \boldsymbol{W}^{(2)}, \boldsymbol{b}^{(2)}, ..., \boldsymbol{W}^{(k)}, \boldsymbol{b}^{(k)}\} \tag{3.3}$$

The policy ANN does not necessarily need to output the action directly. Instead, most RL-related policies are stochastic, i.e. the neural net outputs the mean and standard deviation of a normal distribution:

$$\pi_\theta(\mu_\theta, \sigma_\theta|s) = \frac{1}{\sqrt{2\pi}\sigma_\theta} e^{-\frac{(a-\mu_\theta)^2}{2\sigma_\theta^2}} \tag{3.4}$$

The action a is then drawn from $\mathcal{N}(\mu_\theta(s), \sigma_\theta^2(s))$. In this work, the policy is represented with a neural network that has two outputs, the mean and standard deviation. Dynamic

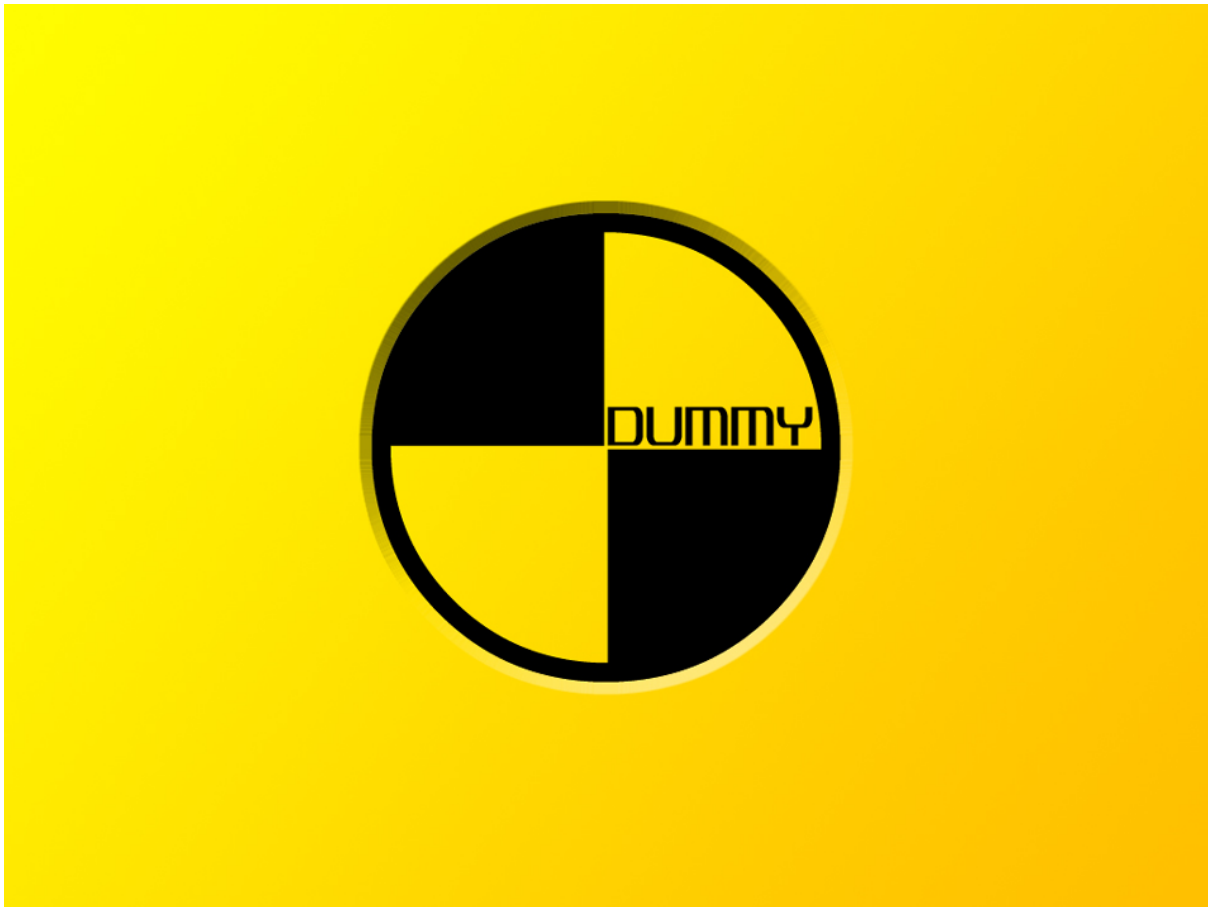

Figure 3.2.: Activation functions [12]

Programming is used only to find the optimal mean values. The standard deviation is set to 0.5. This serves as an intial solution for the policy net to be trained by an RL method.

Input Standardization

All training algorithms that are used in this work rely on the partial derivative of a loss function with respect to the network weights and biases. As can be seen in Eq. 3.8 and the following equations, they perform a step in (an approximation of) the current gradient direction. This gradient heavily depends on the network inputs. If some of the inputs have much larger values than the rest, this can result in some weights being updated faster than others, which is not desirable. Therefore, the inputs to a neural network should have the same scale. In this work, the inputs are scaled to a normal distribution with $\mu = 0$ and $\sigma = 1$. Note that μ_t and σ_t are not the same as μ and σ. μ_t and $\log(\sigma_t)$ are the outputs of the policy MLP at timestep t.

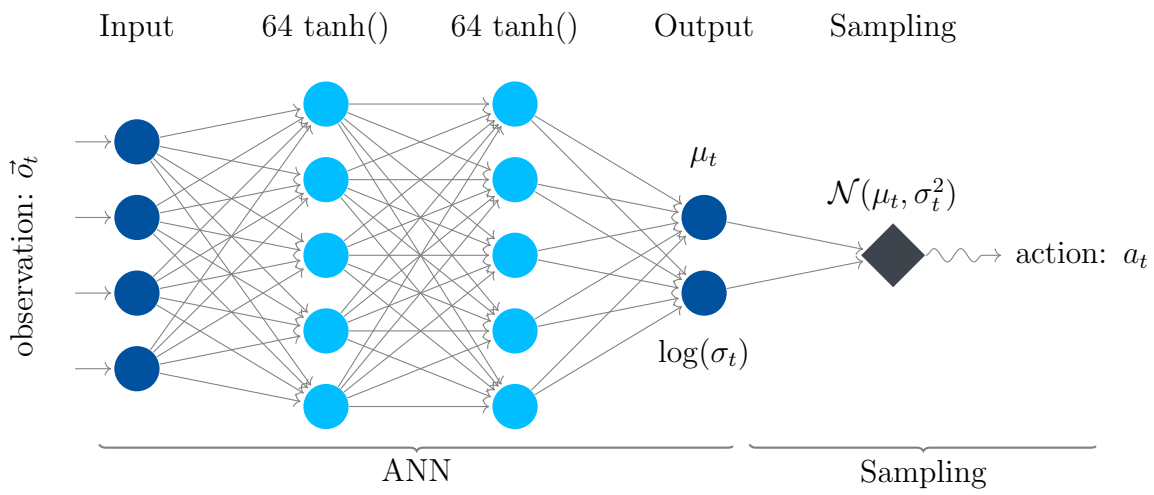

Figure 3.3.: The policy MLP [12]

$$\hat{x} = \frac{x - \mu}{\sigma} \tag{3.5}$$

The specific values for μ and σ for each of the scenarios are presented in table A.1.

Action Scaling

In an artificial neural network, every neuron can output a value between -1 and 1. Therefore, the output of a neural network is limited by the number of neurons in the last hidden layer. Obtaining a large output requires a lot of neurons. On the other hand, lots of neurons favor overfitting and increase training time. If the desired outputs are too small, on the other hand, then so are the neuron outputs. The tanh(·) activation function that is used for the policy is approximately linear in this area. Therefore, the whole network behaves almost linearly if it is trained to output values close to zero. With linear mapping being possible to approximate by an ANN without any hidden layer, there would not be any benefit of applying a deep net. The most important advantage of deep neural networks is however their ability to represent highly complex and a priori unknown behavior.

Both problems can be avoided if outputs lie between -1 and 1. This way, it is unlikely that any of the activation functions reach their saturation limit, nor is the network restricted to linear behavior. Therefore, the neural network is usually trained to output values between -1 and 1 and these values are then scaled to the desired magnitude by multiplying them with a scaling factor. Another way to mitigate this problem is choosing activation functions that are also nonlinear if the input and/or output is close to zero.

Weight initialization

Training an artificial neural network is an iterative process. Therefore it requires initial values for the weights and biases. As training a neural network through supervised learning comes down to solving an optimization problem iteratively, training success highly depends on the initial weights.

The two most popular ways to initialize neural network weights are the Glorot uniform initialization and the Glorot normal distribution. According to the Glorot uniform initialization, the weights in one layer must be distributed uniformly within $\pm\sqrt{\frac{6}{n_{in}+n_{out}}}$ where n_{in} and n_{out} are the numbers of neurons in the prior and current layer, respectively.

The Glorot normal initialization states that the weights must be initialized according to a normal distribution with zero mean and a variance of $\sigma = \sqrt{\frac{2}{n_{in}+n_{out}}}$.

In this work, all neural networks are initialized according to the Glorot uniform distribution.

3.3. Supervised Learning

All machine learning algorithms can be divided into two categories: supervised learning and unsupervised learning. In supervised learning, the agent is given the desired action for each state explicitly, so it knows what it is supposed to do. In unsupervised learning, however, the agent does not know what the best action is, so it has to find out itself. One way of doing that is from moving in the state space and learning from experience, for example by applying a reinforcement learning algorithm.

In the supervised learning case, training is done via minimizing a loss function that depends on the desired action and the actual action taken by the agent. A common loss-function is the mean squared error (MSE) :

$$\bar{e} = \frac{1}{N}\sum_{n=1}^{N} e_n \tag{3.6}$$

$$= \frac{1}{N}\sum_{n=1}^{N}[\frac{1}{2}\,(a_n(s) - a_{fit,n})^2] \tag{3.7}$$

The goal of supervised learning is to minimize the difference between $a_{n,fit}$ and $a_n(s)$ in each state. If $a_n(s)$ is the output of a function approximator, minimizing L_{MSE} is equivalent to fitting the function approximator to the desired outputs.

One advantage of the application of a loss function is that numerous iterative optimization algorithms have been developed to solve such a problem.

3.4. Optimization Techniques

As mentioned before, supervised learning is usually done by defining a loss function that depends on the network weights and biases and finding a local or global minimum numerically.

In this work, the Adam algorithm is used for training the policy to output the desired action at each point in the state space. As most optimization algorithms, Adam is gradient-based. Therefore, some insight into the general ideas behind gradient based numeric optimization is useful for understanding Adam. Sections 3.4.1 and 3.4.2 explain briefly how gradient based optimization works in general. In section 3.4.3, Adam is described and explained.

All presented algorithms produce a series of points in the space that the network weights and biases establish. The point at the kth optimization step is defined by a distinct set of weights and biases θ, similar to coordinates in the state space (c.f. Eq. 3.3).

3.4.1. Gradient Descent

In gradient descent, the next point θ_{k+1}, i.e. the next set of weights and biases, is reached by computing the gradient of the loss function with respect to the network weights, $\nabla_{\theta_k} L_k$, and performing one step in the direction of steepest descent. θ_{k+1} is calculated by the following equation:

$$\theta_{k+1} = \theta_k - \nabla_{\theta_k} \bar{e}_k \cdot \alpha \tag{3.8}$$

with

$$\nabla_{\theta_k} \bar{e}_k = \frac{1}{N} \cdot \sum_{n=1}^{N} \nabla_{\theta_k} e_{n,k} \tag{3.9}$$

$$= \frac{1}{N} \cdot \sum_{n=1}^{N} [(a_n(s) - a_{n,fit}) \cdot \nabla_{\theta_k} a_n(s)] \tag{3.10}$$

and an arbitrary step size α.

The choice of α has a big effect on the convergence of the optimization algorithm. If α is too small, each θ_{k+1} is close to the respective θ_k so it takes many steps to find a minimum. If α is too big, the algorithm might not converge at all.

Ordinary gradient descent methods are not suitable for optimizing the output of large artificial neural networks because computing the gradient $\frac{\partial L_k}{\partial \theta_k}$ for all the weights and biases is very computation-intensive and time-consuming.

3.4.2. Stochastic Gradient Descent

As mentioned before, the output of a neural network cannot be optimized efficiently with a gradient descent algorithm. One way to bypass this weakness is Stochastic Gradient

Descent (SGD). If the loss function is a sum like in Eq. 3.7, it is sufficient to calculate the gradient of only one summand of the loss function, i.e. only consider one random training sample at each step k, and perform a small step in the corresponding direction:

$$\nabla_{\theta_k} \bar{e}_k \approx \nabla_{\theta_k} e_{n,k} \tag{3.11}$$

$$\theta_{k+1} = \theta_k - \nabla_{\theta_k} e_{n,k} \cdot \alpha \tag{3.12}$$

If this is done for all samples successively, it also converges to the loss from Eq. 3.7. Thanks to the faster computation of $\nabla_{\theta_k} L_{n,k}$, SGD converges faster than ordinary gradient descent although Eq. 3.11 is only a rough estimate of the true gradient. (todo: Beleg und Erklärung)

3.4.3. The Adam Algorithm

Basic Stochastic Gradient Descent has a fixed stepsize α. Most advanced optimization algorithms have a variable stepsize and an update direction that depends on the previous updates to accelerate convergence. One example is to add a momentum term to the gradient from Eq. 3.12. This affects the step size and the direction of the update.

It takes into account the previous update and changes the direction of the next update. Like a ball rolling down a slope, the algorithm does not immediately change direction if the gradient of the slope changes direction or stop if the gradient is zero. If there is a plateau in the loss function or a weak local minimum, ordinary algorithms tend to get stuck there. With momentum, the risk of getting stuck is reduced.

The Adam-algorithm utilizes a modified momentum term for better convergence. Equation 3.13 shows one update step, the parameters $\hat{m}_t$ and $\hat{v}_t$ are calculated with equations 3.14 and 3.15.

$$\theta_{k+1} = \theta_k - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{3.13}$$

with

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} = \frac{\beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t}{1 - \beta_1^t}, \tag{3.14}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} = \frac{\beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2}{1 - \beta_2^t} \tag{3.15}$$

The algorithm utilizes the first and second moments of the previous gradients to calculate the update. Consider a pair of m_t and v_t that follow recursively from m_{t-1} and v_{t-1} according to equations 3.14 and 3.15. For the calculation of each m_t and v_t, the previous value m_{t-1} and the current gradient g_t are used. m_t and v_t are biased estimates of the first and second moments, there is a correction term in each of the denominators of equations 3.14 and 3.15 to account for that. See [6] for more details.

3.5. Overfitting

It is seldom desirable to minimize the loss function to its absolute minimum value on the training data. At first, this might sound counterintuitive. But apart from the additional time it takes to get there, it increases the chances of running into problems.

When dealing with a continuous state space, it is common practice to discretize it and deal with the discretized state space instead. While a continuous state space consists of an infinite number of states, the number of states in the discretized state space is finite. This reduces computation time and makes some algorithms applicable in the first place. If an appropriate grid is used (i.e. if it is dense enough in relevant areas of the state space), the solution is likely to be close to the solution of the continuous problem. The more grid points, the closer the solution is to the continuous solution.

If, however, a function approximator like an ANN is optimized on the discretized state set, it is possible that, although it might fit the training data very closely, it may not generalize well, i.e. give unexpected outputs when fed with states that have not been used in training. This problem is called *overfitting*. It especially occurs if at least one of the following conditions is met:

- Noisy data (every data point differs slightly from the expected value)
- Limited dataset (only a few points to fit to)
- High number of parameters (many degrees of freedom)[1]

Overfitting is an issue in many function- and curve-fitting applications. Fig. 3.4 shows overfitting in case of a high order polynomial. Although it is much simpler than a neural network, the problems are similar.

The plot shows two polynomials that have been fitted to a set of seven points. The blue line represents a quadratic polynomial, the red line belongs to a 10th order polynomial. A nth order polynomial has n+1 parameters and can therefore be fit exactly to n+1 points. The quadratic polynomial only has three parameters and can therefore not pass through all seven points exactly. In such cases, fitting is done by minimizing the mean quadratic fitting error:

$$\bar{e}_{fit} = \frac{1}{m}\sum_{i=1}^{m}[y_i - y_{i,fit}]^2 \tag{3.16}$$

$$= \frac{1}{m}\sum_{i=1}^{m}[y_i - \sum_{n}(a_n\, x^n)]^2 \tag{3.17}$$

where m is the number of given points.

[1] i.e. lots of neurons in the case of an ANN.

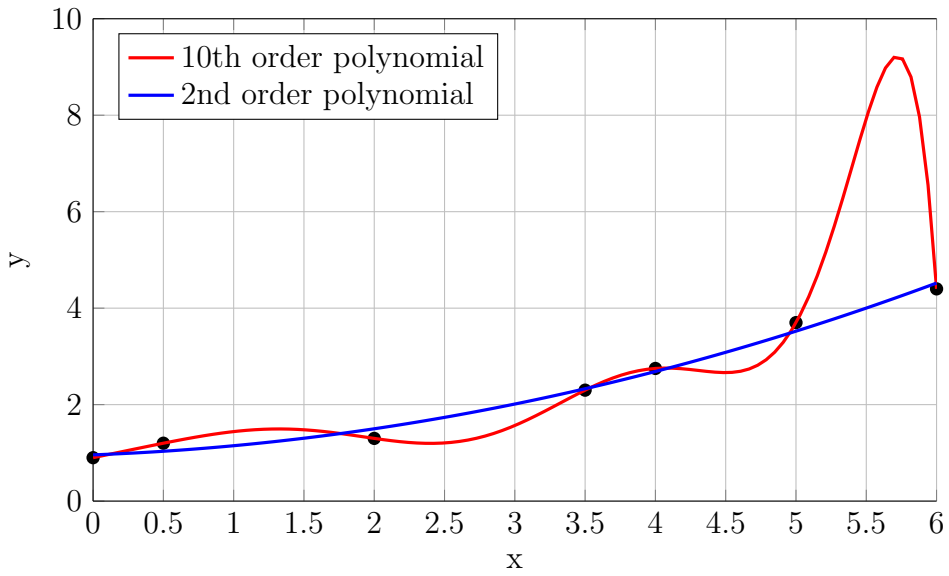

Figure 3.4.: Overfitting in case of a high order polynomial and noisy data

The 10th order polynomial has eleven parameters. It is therefore possible to fit the function exactly to the seven input points. This can be done by solving a system of linear equations where each point contributes one equation. In the case of the 10th order polynomial, there are eleven parameters. If this polynomial is fitted to seven points, there are seven equations and eleven parameters to be determined. The corresponding system of equations is called *underdetermined*. Four of the parameters have to be set prior to solving the linear system. Their choice is arbitrary, which can lead to a shape like the red plot. Although the seven conditions at the points are met exactly by the red line, the blue plot intuitively seems like a more appropriate approximation.

Of course, any high order polynomial can also be fitted to a set of points by minimizing 3.17. Unfortunately, this can also lead to overfitting, because it uses the same information (input point coordinates). Therefore, it will eventually find a set of parameters that fits the points exactly and is likely to have the same oscillations as the red line shown in Fig. 3.4.

In any application, where there are more parameters than points to fit, overfitting can be an issue. Any function approximator should therefore only have as many parameters as necessary. Any additional parameter can increase the risk of overfitting. There are several other precautions that can be taken in order to avoid overfitting. Some of them are presented in the following paragraphs.

One way to deal with overfitting is splitting the state set into a set of training data and a set of validation data. After training on the training dataset, the value of the loss function on the training data is compared to the value of the loss function on the validation data. If - from one training step to the next - the training loss decreases while

the validation loss does not, overfitting is likely. This can be used as a criterion to stop the training. Fig. 3.5 shows how training loss and validation loss typically evolve during training. After iteration 63, only the training loss decreases. At this point, training should be stopped in order to avoid overfitting.

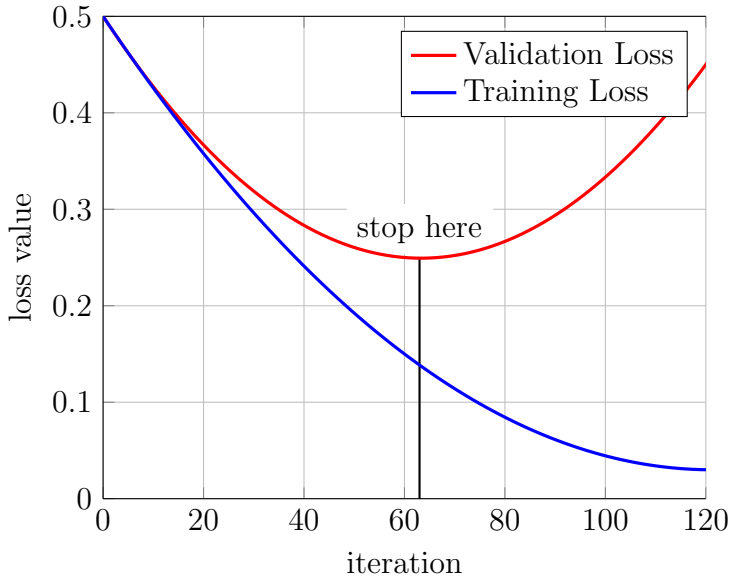

Figure 3.5.: Stop criterion to avoid overfitting

The training set can also be split into three datasets, one for training, one for validation during training, and the third for post-training overfitting detection. Typically, the third dataset contains approximately 20% of the samples, the remaining 80% are divided into training and validation data at the same ratio, so the parameter updates are done with 64% of the total sample count, and 16% are used to check for overfitting during training.

Another way of preventing overfitting is to penalize large weights and biases in the hidden layers. If the weights and biases have high values, they can cause similar behavior to what is shown in Fig. 3.4. Penalizing large weights and biases typically smoothes out the neural network output, thus averting oscillations.

In this work, there are two datasets. At the beginning of training, 80% of the data are chosen randomly and used for training. The remaining 20% are used to detect overfitting during training according to what is depicted in Fig. 3.5. Note that it typically takes more iterations until overfitting occurs than in the previous example. Large weights and biases are also penalized. This is done by adding a term to the loss function from Eq. 3.7, yielding Eq.3.18. Note that, in Eq. 3.18, the set of parameters θ is interpreted as a vector that just contains all weights and biases of the hidden layers as a list to simplify the notation of the sum in the last term.

$$\bar{e} = \frac{1}{N} \sum_{n=1}^{N} [\frac{1}{2} (a_n(s) - a_{fit,n})^2] + a_\theta \sum_{i=1}^{I} \theta_i^2, \tag{3.18}$$

where I is the number of scalar weights and biases in the hidden layers and θ_i is the ith parameter of the ANN. A value of 0.06 for a_θ has proven to suppress overfitting without affecting the neural network output too much.

4. Trajectory Optimization with Dynamic Programming

4.1. 2D Environment

In the context of this work, the aircraft is treated as a mass point. Rotational dynamics are neglected. This is sufficient for trajectory optimization algorithms. See [5] for more details.

In this scenario, the glider moves in the geodetic vertical plane. Its control is the angle of attack α. When α is increased, the lift L increases. This results in an increase of γ, as can be seen in equation 4.4. This way the agent can control its velocity vector and therefore its position. All parameters of the glider are shown in appendix A.1. Any 3D-Scenario where the horizontal trajectory is restricted by obstacles or the optimal ground trace is obvious can be regarded as a 2D problem. The only control variable to be optimized is then the angle of attack or an equivalent quantity (e.g. vertical acceleration).

4.1.1. Policy Representation

As the Policy Iteration algorithm is running on a rectangular grid and the states that are updated do not change, a table that stores the greedy action for every state is sufficient for the Policy Iteration algorithm. Once the optimal policy π^*_{calm} is found, a neural network is trained to approximate the optimal action for each of the states on the grid. This policy is then used to do TRPO.

4.1.2. Equations of Motion

The physical state of the glider in the geodetic vertical plane consists of the position ${}_g\boldsymbol{r} = [x, z]^\top$ and velocity ${}_g\boldsymbol{v_K} = [{}_gu_K, {}_gw_K]^\top$. Apart from the geodetic reference frame (index g), a body fixed frame (index f), a trajectory fixed frame (index k) and an aerodynamic frame (index a) are used to describe the glider dynamics. In the vertical plane, coordinates can be transformed between the frames by a rotation around the y-axis. Figure 4.1 illustrates the transformation angles that connect the reference frames. Note that, in this work, there is no wind. $\vec{v}_W$ and α_W are therefore both zero and the aerodynamic and trajectory fixed frames are the same.

In the vertical plane, there are 4 state variables, x, z, $u = \dot{x}$ and $w = \dot{z}$, each with respect to a geodetic reference frame that has its origin at a point on the earth surface.

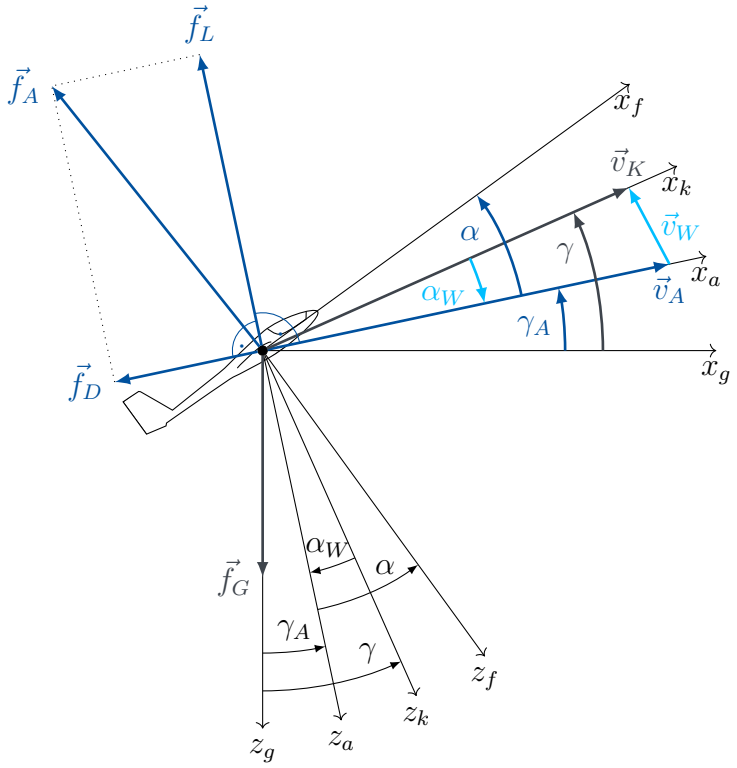

Figure 4.1.: Reference frame transformation [8]

The aerodynamic forces are often expressed in an aerodynamic reference frame. The coordinate transformations u and w can also be expressed by V and γ.

The dynamics in the vertical plane are given by the following equations. They are taken from [5].

$$\dot{x} = V\ cos(\gamma) \tag{4.1}$$

$$\dot{z} = -V\ sin(\gamma) \tag{4.2}$$

$$\dot{V} = -\frac{D + g\ cos(\gamma)}{m} \tag{4.3}$$

$$\dot{\gamma} = \frac{L}{V\ m} - \frac{g\ cos(\gamma)}{V} \tag{4.4}$$

where $L = q\ c_L\ S$, $D = q\ c_D\ S$ with $q = \frac{\rho}{2}\ V^2$ and $V = \sqrt{u^2 + w^2}$.

4.1.3. Reward Function

As shown in Fig. 2.1, the agent interacts with his environment by, at each time step, picking an action a_t and receiving an observation o_t and a reward r_t. In RL, the reward function is the only means to give the agent information about what he should or should not do.

The reward function in this work is as follows:

$$\mathcal{R}(s,a) : \mathcal{S} \mapsto \mathbb{R} : r_t = \begin{cases} -\Delta t & \text{if } s_{t+1} \neq s_{T+} \\ -\Delta t + \frac{d}{10} & \text{if } s_{t+1} = s_{T+} \end{cases} \tag{4.5}$$

In Fig. 4.2, all state transitions, where the agent crosses a thick line (i.e. leaves the admissible part of the state space), lead to termination of the episode. If the agent hits one of the black thick lines, he gets a final reward of 0. Only passing the green line earns him a final reward of $\frac{d}{10}$.

where s_{T+} is a terminal state that is also a target state. There are terminal states s_{T-} where the agent is not meant to go. These terminal states are those where the agent touches the ground before reaching his target range. If the agent reaches his goal, he receives a final reward r_T according to the second line of Eq. 4.5.

It is possible to penalize undesired behavior of the agent by defining a large negative reward if, for example, the agent touches the ground before reaching the target distance or flies too far in the wrong direction. This is, however, not necessary, if the final reward is sufficiently high. The important thing here is, that reaching a target state must always yield a higher return than ending an episode on the ground from any state $s \in \mathcal{S}$.[1]

4.1.4. Discretization of the State- and Action Space

The state space is discretized with a rectangular grid, i.e. it is replaced by a set of points $s_i = [x_k, z_l, u_m, w_n]^\top$ evenly distributed across all dimensions.

Fig. 4.2 illustrates the discretization of the state space. The agent starts its flight at the top left corner (at the glider symbol) with zero velocity (hence the green dot on the speed grid). For the grid cell at x_4 and z_3, the grid for the speed vector is drawn. All of the cells in the position grid contain a speed grid, like the small one in Fig. 4.2.

In the 2D-scenario, the action space is one-dimensional, with the only action being the angle of attack α. For the policy improvement step, a finite number of evenly spaced actions is sampled from the infinite set of possible actions between $\alpha_{min} = 0$ and $\alpha_{max} = 0.2$. Assuming that the mapping from actions to returns is continuous, the action that yields the maximum expected return out of the sampled actions is an approximation of the true greedy action.

[1] This is true in the scenarios in chapter 5. In scenario 1, for example, the agent has to cover a horizontal distance of 500m. This takes him between 19.7s (c.f. table 5.2) and approx. 40s and earns him a time penalty of up to -40. With a final reward of $\frac{d}{10} = +50$, the resulting total reward is still positive and therefore sufficiently high for the agent to find the goal.

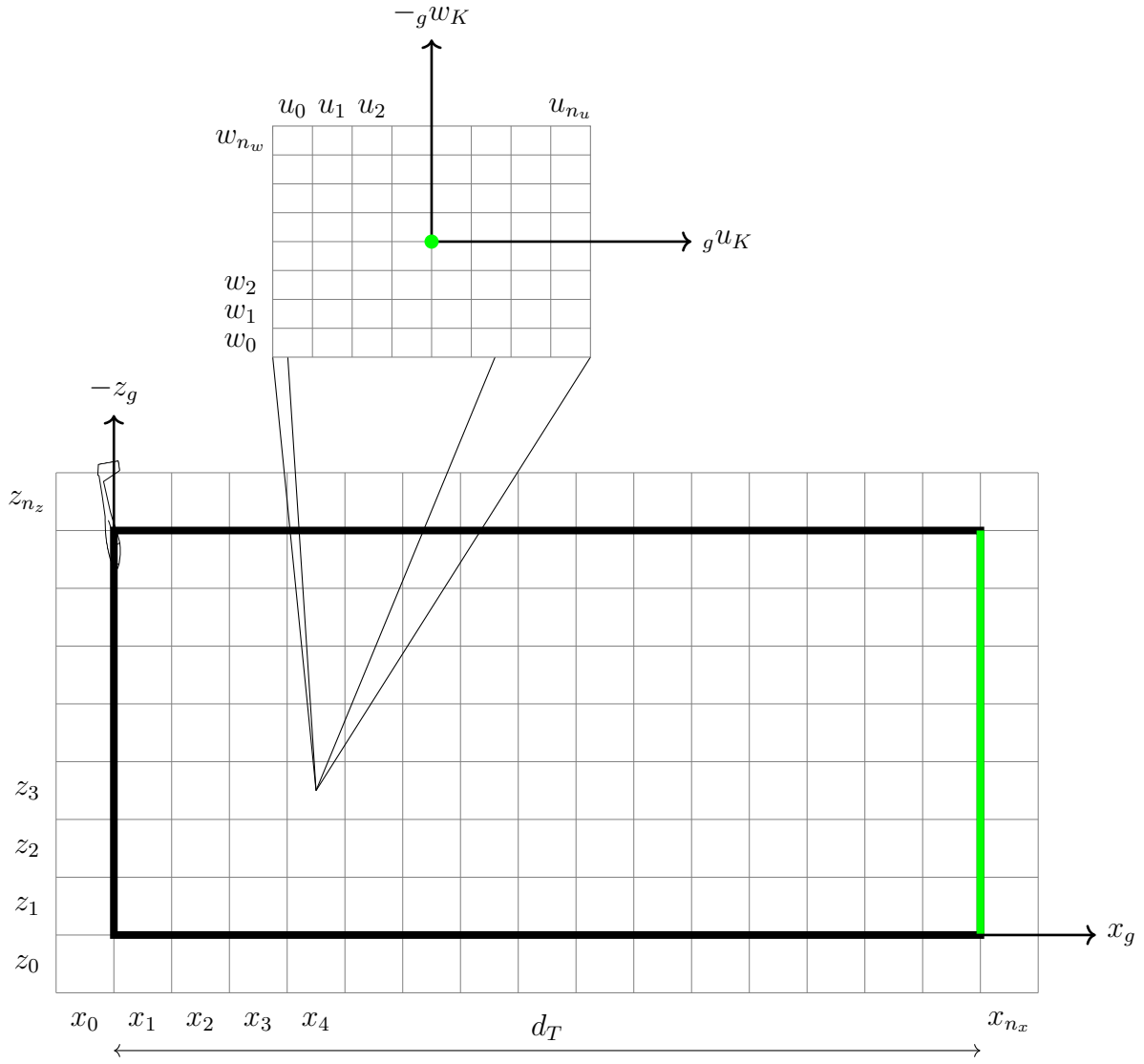

Figure 4.2.: The discretized state space in all 2D scenarios.

distance	500 m	1000 m	2000 m
n_x	52	102	202
n_z	52	52	52
n_u	8	8	8
n_w	8	8	8

Table 4.1.: Grid parameters for trajectory optimization

By discretization, the continuous trajectory optimization problem is made time- and space-discrete. For the purpose of keeping calculation time low, discretization should be as coarse as possible without affecting the result too much. A sample time Δt of 1 s is used for all scenarios. With a horizontal speed of $15 - 25$ $\frac{\text{m}}{\text{s}}$, the agent covers about $15 - 25$ m within one time step. The horizontal separation of grid points should be about 10m. This way, the agent can skip one grid cell, if its velocity is sufficient. Grid cells, that are closer to the goal, usually have a higher state value. With the possibility to skip a cell, the agent is thus encouraged to fly at higher speeds. For the different scenarios, this yields the grid resolutions shown in table 4.1. In theory, more horizontal grid layers can yield a better solution. But because the tabular VI data is used to train a neural network, which "smoothes" out the given data, more than 52 layers (i.e. 2m between cells vertically) have proven not to improve the result significantly.

4.2. 3D Environment

In the 3D scenario, the agent is put in a rectangular 3D state space and has to find a goal. Like in 2D, the goal is to reach the far end of the state space from his initial state. To make things more interesting, the direction the agent is flying in the beginning varies. In 3D, the agent can change the flying direction by performing a half loop, a horizontal curve, or a combination of both. What is optimal depends on the specific case.

In this three-dimensional environment, the state vector consists of three cartesian coordinates ${}_{\boldsymbol{g}}\boldsymbol{r} = [x, y, z]^\top$ and three cartesian velocities ${}_{\boldsymbol{g}}\boldsymbol{v}_{\boldsymbol{K}} = [{}_g u_K, {}_g v_K, {}_g w_K]^\top$. The goal is for the agent to reach a given distance in minimal time. The controls are the angle of attack α and the bank angle μ. The angle of attack works like in a 2D environment, whereas the bank angle lets the agent change his heading in order to reach the goal.

4.2.1. Discretization of the state and action space

Discretization is done like in the 2D scenario, which results in cuboids for the position and velocity.

5. Results

5.1. 2D Dynamic Programming

5.1.1. Optimal Control Benchmark

In the 2D scenario, three Dynamic Programming algorithms are compared:

- Generalized policy iteration (cf. section 2.11.2)
- Optimistic policy iteration (cf. section 2.11.2)
- Value iteration (cf. section 2.11.2)

The agent faces three different scenarios, each with calm air. Table 5.1 shows the distance to cover, start state, and the mean and standard deviation for state normalization (cf. 3.2) for each scenario. For each scenario, there is a separate mean μ and standard deviation σ for each coordinate. Therefore, each column contains a vector of means and standard deviations.

scenario	1	2	3
distance d_T	500m	1000m	2000m
start state s_0	$s_0 = [0, 100, 0, 0]^\top$	$s_0 = [0, 100, 0, 0]^\top$	$s_0 = [0, 100, 0, 0]^\top$
normalization μ	$[250, 50, 10, 5]$	$[500, 50, 10, 5]$	$[1000, 50, 15, 1]$
normalization σ	$[150, 30, 5, 2]$	$[300, 30, 5, 2]$	$[600, 30, 6, 3]$

Table 5.1.: scenario data

Recall that policy iteration algorithms produce a sequence of value-functions and policies that approximate V_* and π_*. Each iterate V_{k+1} and π_{k+1} is closer to V_* and π_* than its predecessor V_k and π_k. In the GPI algorithm, the iterative evaluation stops, if the values of all states change by less than the reward for one time step Δt within one evaluation step. When this point is reached, only the time penalty propagates through the network. At states s_t, where the agent only moves slowly, chances are that the successor state s_{t+1} is (roughly) equal to s_t (and therefore $V(s_t) = V(s_{t+1})$). In such cases, the state value $V(s_t)$ is overwritten with $r_t + V(s_t)$ at each iteration. This goes on indefinitely if PE is not stopped manually. Every policy iteration where the evaluation step is repeated multiple times obviously takes longer than an optimistic policy iteration. The whole policy

iteration algorithm has converged if - at the last policy improvement step - none of the actions have changed. This means that all actions were already optimal with respect to the current value function as well as the previous one.

Value iteration only iterates state value functions. It combines one step of policy evaluation and policy improvement in every iteration. One value iteration therefore usually takes less time than one step of policy iteration. If a close approximation of V_* is found, a policy can be derived by acting greedily with respect to the last value function iterate.

In scenario 1, the agent has to cover 500 m while flying through calm air. Table 5.2 shows the flight times from the start-state to the goal after policy optimization with GPI, OPI and VI. As a benchmark, the optimal control is shown in the first column. Calculation time is the time it took to obtain the given trajectories and control sequences. Note that lower calculation and flight times are better. As mentioned before, value iteration takes the least time for one iteration and converges before OPI and GPI.

Although all three DP algorithms theoretically take up to $|\mathcal{A}|^{|\mathcal{S}|}$ iterations to converge (c.f. section 2.11.2), they are known to typically converge after only few iterations. This is also the case in this scenario.

	OC	GPI	OPI	VI
iterations (-)	-			7
calculation time (h:min)	0:15			2:04
flight time (s)	19.7			20.0

Table 5.2.: Flight- and computation-times for OC, GPI, OPI and VI in scenario 1

All algorithms yield similar flight times that are slightly higher than the OC result. As can be seen in table 5.2, VI takes the least calculation time. As all algorithms converge to a reasonable solution, both policy iteration algorithms are not considered for scenarios 2 and 3, because VI converges faster and calculation time gets more critical at higher distances, i.e. more grid points.

Figures 5.1, 5.2 and 5.3 show the results of generalized policy iteration, optimistic policy iteration and value iteration, respectively. As a benchmark, every figure also contains the optimal trajectory and control sequence.

The value iteration algorithm clearly achieves the best result with respect to flight time. Within the first 30m, the angle of attack is a little higher than optimal, leading to a trajectory that lies above the optimal one. Nevertheless, the flight time is only 0.3s higher than the benchmark. Because the state values are updated

Figure 5.1 shows the agent's trajectory optimized with GPI. Optimization took approximately two hours (c.f. table 5.2). Qualitatively, the control sequence is very similar to the optimal control sequence. Within the first 50m of horizontal movement, there is some stutter in the control sequence which increases flight time slightly.

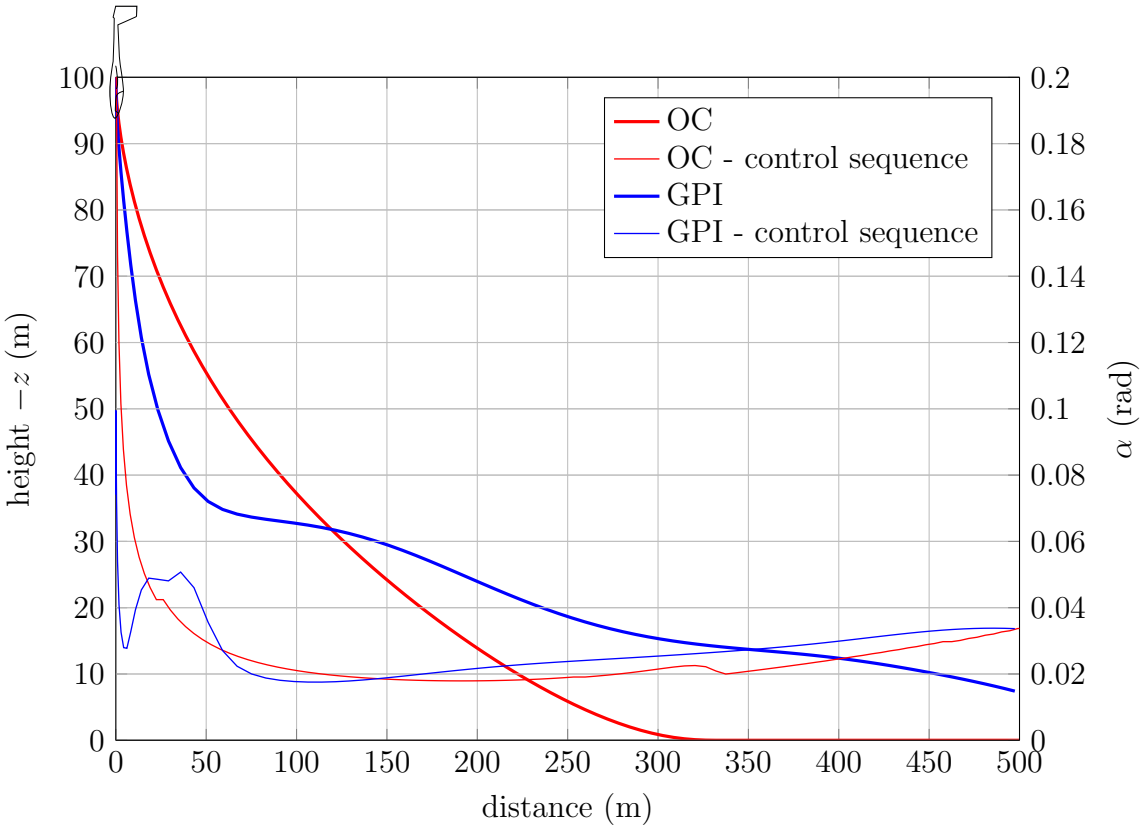

Figure 5.1.: Results of generalized policy iteration (blue) and optimal control (red) in scenario 1

In the upper left corner, a small part of the state grid is drawn. The grid elements have their original size with respect to the height and the distance.

Fig. 5.3 shows the trajectory and control sequence after only 7 value iterations. The algorithm work themselves through the state space backwards, updating the state values close to the target first and moving backwards from there (cf. 2.12). This increases convergence speed significantly. The number of required iterations is also independent of the number of states with this method. This is not the case with synchronous updates (i.e. updating all state values at once as soon as there is a new value for all states).

Figure 5.4 shows in blue the trajectory in scenario 2 after 7 value iterations. The thick line is the glider trajectory $[x(t), z(t)]^T$, the thin line is the control sequence $[x(t), \alpha(t)]^T$ The agent is dropped at the start state with zero velocity. As can be seen, both the policy from VI and the optimal control (red) direct the agent towards the target.

Both algorithms yield policies that are able to find the goal from various initial states.

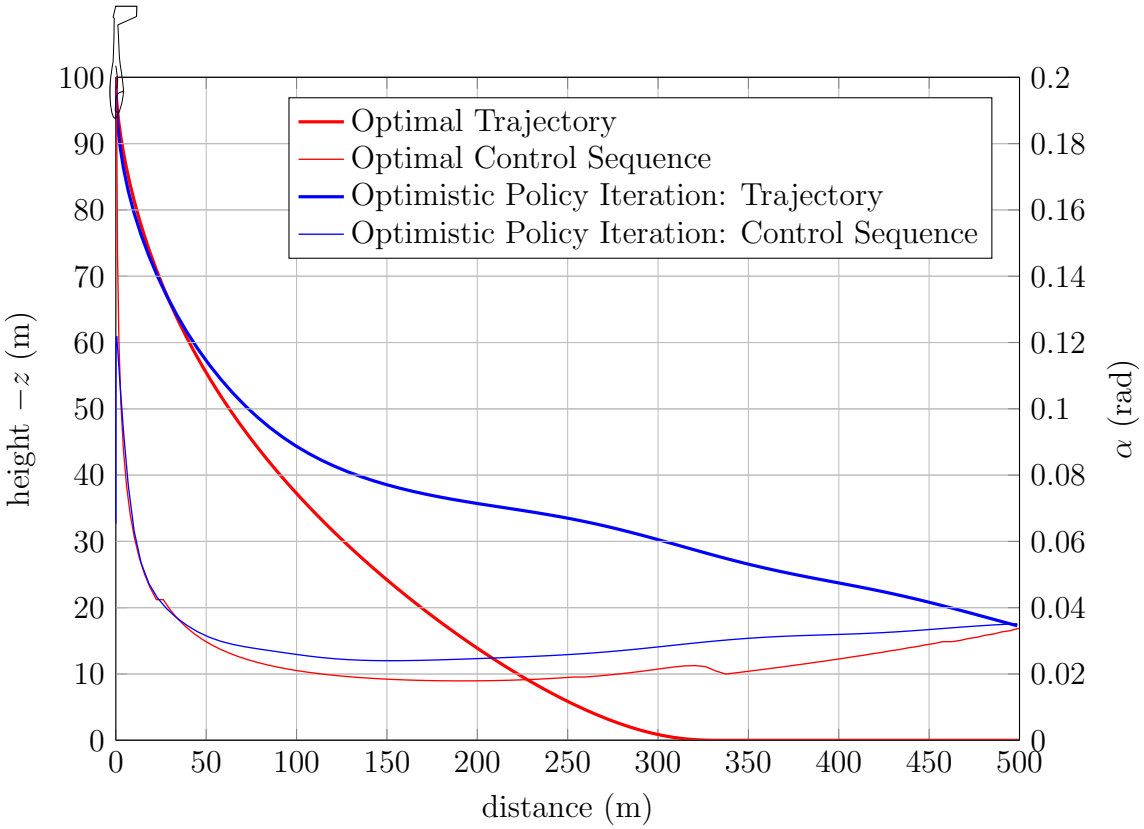

Figure 5.2.: Results of optimistic policy iteration (blue) and optimal control (red) in scenario 1

5.1.2. Policy Initialization for TRPO

In the previous section, it was shown that value iteration is a viable alternative for optimal control in certain path planning scenarios. The goal of this section is to find out whether value iteration can also serve to accelerate convergence of a TRPO algorithm.

The parameters of a neural network are usually initialized randomly according to a specific probability distribution (cf. section 3.2). The idea behind policy initialization by DP is to use a set of weights, that have already been optimized in a similar scenario, in order to speed up convergence of a subsequent training procedure.

The TRPO training procedure differs from the one in dynamic programming. Instead of sweeping over the complete state space and performing a one step lookahead at each point, TRPO samples complete trajectories. A specific number of trajectories

The performance of TRPO can either be measured by flight times (like in section 5.1.1) or by the average and maximal return

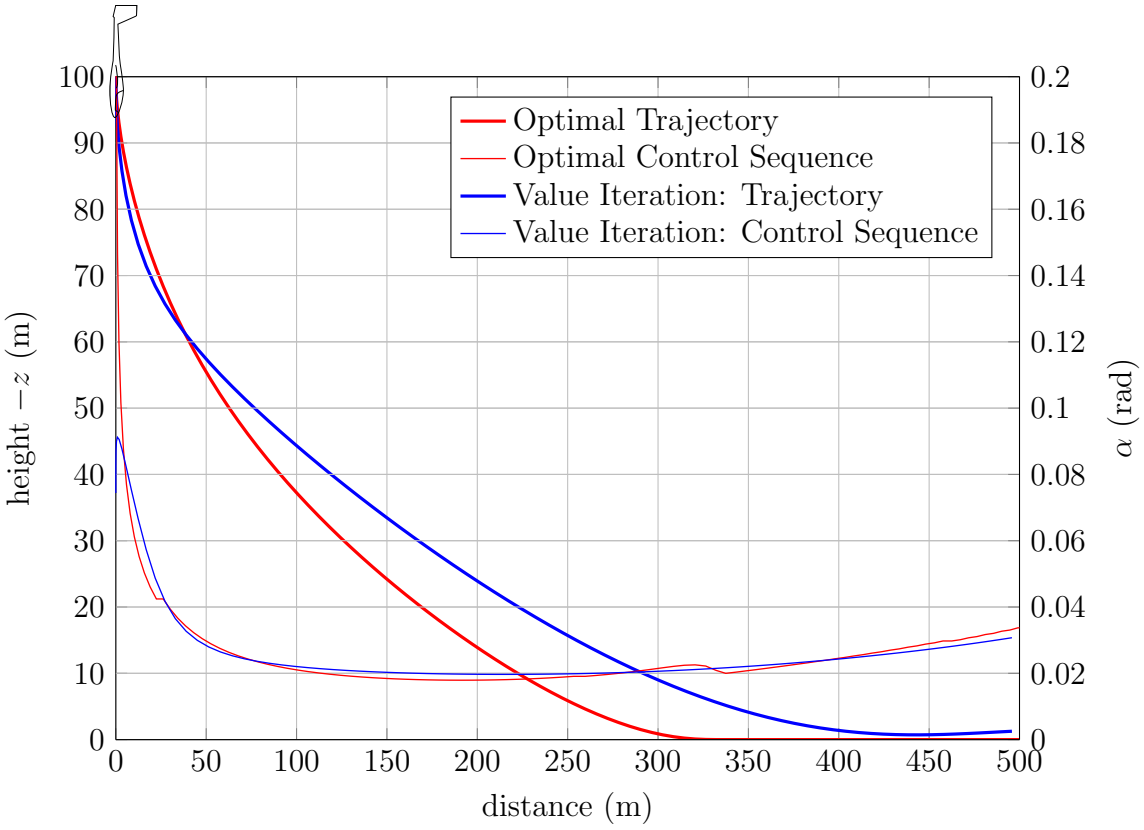

Figure 5.3.: Results of value iteration (blue) and optimal control (red) in scenario 1

algorithm	OC	VI
approx. time per iteration (s)	-	
iterations (-)	-	43
calculation time (h:min)	1:31	
flight time (s)	47.0	50.4

Table 5.3.: Flight- and computation-times for OC, OPI and VI in scenario 2

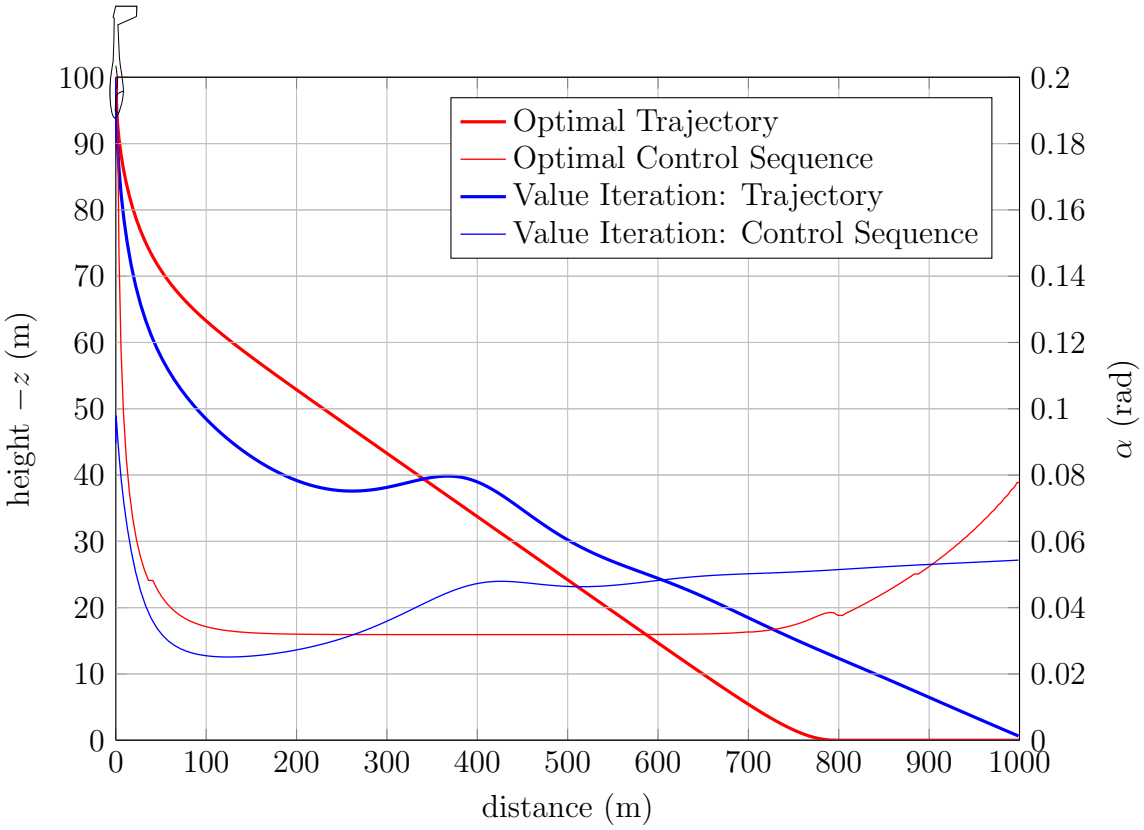

Figure 5.4.: Results of value iteration (blue) and optimal control (red)

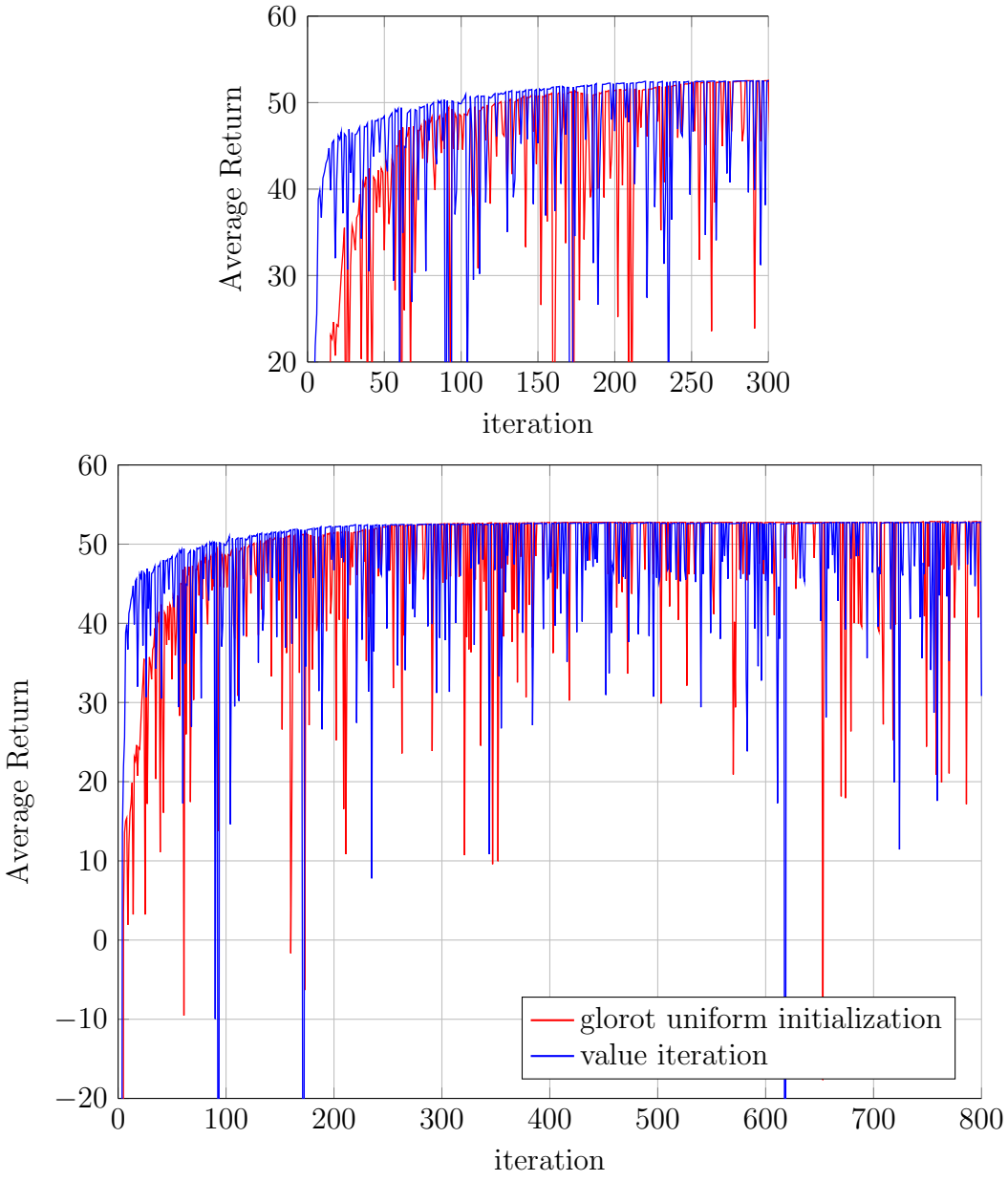

Figure 5.5.: Average return per iteration with and without policy initialization

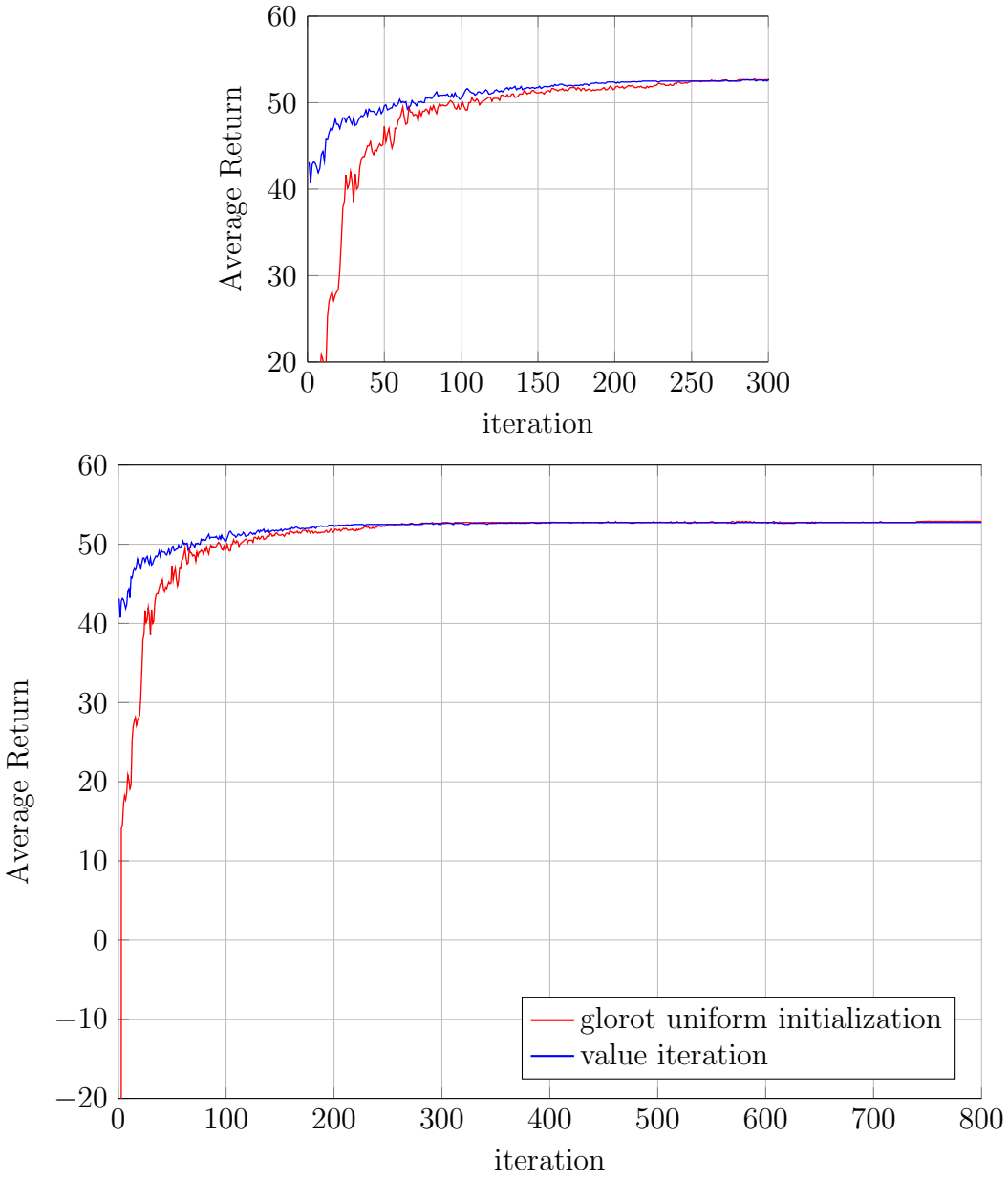

Figure 5.6.: Maximum return per iteration with and without policy initialization

6. Discussion

In this work, three dynamic programming algorithms have been implemented and used for glider trajectory optimization in a calm environment. Their results were compared to an optimal control result and it has been investigated, if pre-training with DP helps convergence of a TRPO algorithm. The three algorithms all converged to a near-optimal solution in a scenario where 500m were to cover and the initial state was known a priori. Value iteration took the least time and was therefore chosen for studies in scenarios with distances of 1000m and 2000m. Results showed that TRPO performance is improved by DP. But if the pre-training time had been used for more TRPO iterations instead, the total calculation time would have been even lower.

As for the performance of PI and VI, the latter has several advantages. It is easier to implement, because it takes less steps than PI. This also makes it easier to optimize VI for a specific problem structure. In this case, performing VI layer by layer only took minor changes to the code, whereas PI took more adjustment in order to be performed this way. Because VI takes less steps for a state value update, each value iteration also takes less time in total (cf. appendix A.2).

Based on the results in chapter 5, one could argue, that dynamic programming does not make sense in this context. With a sufficiently fine state grid, even value iteration takes eight times as long as the optimal control benchmark in scenario 1 and only yields a suboptimal result in all scenarios. If compared to TRPO, a policy optimization algorithm (cf. [12]), it performs worse with respect to calculation time as well as flight time. It also cannot deal with unknown upwinds, which TRPO can.

A big advantage of DP is however, that it inherently covers the whole state space. If the result of one of the DP algorithms is used to train an artificial neural network, an agent utilizing it can get to the goal from every point in the state space[1]. This is not true for TRPO and OC, which both optimize a specific trajectory from a specific start state to the goal. As a tool for policy initialization, DP is especially helpful, if the start state (and therefore the approximate trajectory) is not known a priori.

Even if the start state is known, DP has an impact on the results of TRPO. As the results in Fig. 5.6 show, pre training with a DP algorithm increases the initial average and maximum return per iteration for a TRPO algorihm. If the stopping criterion of TRPO is reaching a specific average return (i.e. a specific average flight time), Fig. 5.5 shows that this goal can be reached earlier than with standard TRPO and a Glorot-initialized policy ANN. For example, a pre-trained policy net reaches an average return of 50 after only 80

[1]This is only true, if reaching the goal is physically possible at all. A glider can, for example, not reach a point that is 1000m away, if its altitude is insufficient and there are no upwinds.

iterations, while a Glorot-initialized policy net takes about 130 iterations. Whether this higher return in early TRPO iterations justifies the additional expense of performing a DP pre-training, depends on the performance of the TRPO algorithm and can therefore not be answered generally. A DP result can also be calculated only once and then stored and used for multiple TRPO calculations afterwards (e.g. with random updrafts or horizontal winds). This way, the time DP takes carries less weight if compared to TRPO without pre-training.

Another topic in section 5.1.2 is, whether policy optimization by TRPO distorts the policy outputs from DP. As can be seen in Fig. **??**, this is (not) the case. This means that an agent with a pre-trained policy still finds the goal after policy optimization through TRPO. A combination of both algorithm retains the advantages of DP and TRPO, making DP a viable method to enhance TRPO performance and results.

A. Appendix

A.1. Glider Parameters

mass m	$3.366kg$
wing area S	$0.568m^2$
aspect ratio Λ	10.2
oswald efficiency factor e	0.9
c_{D0}	0.015

Table A.1.: glider data

A.2. Computer Configuration and Implementation

A.2.1. Computer Configuration

Table A.2 contains the technical data of the system used for calculations.

CPU	Intel i7-7700HQ @ 2.80 GHz (4 physical cores, 8 threads)
RAM	16 GB DDR3 RAM @ 3200 MHz
GPU	Nvidia Geforce GTX 1050 Ti (4 GB VRAM, 768 CUDA cores)
OS	Ubuntu 16.04 LTS

Table A.2.: computer specifications

All algorithms were implemented in Python (Version 3.5), utilizing the rllab framework [1]. The policy ANN used in the simulations was handled by the GPU using CUDNN, Theano and Lasagne. All other code including the tables representing the greedy actions and state values was executed with eight parallel processes on the CPU.

The rllab framework already contains a step-method that lets the agent make a step through the state space. The original step method takes the current state s_t of the agent, which is stored as a class attribute, and an action a_t, which is passed in the function call. It returns the next state s_{t+1}, the reward r_{t+1} and a flag telling the agent whether he has reached a terminal state. This step-method has been modified so it also takes the current state s_t as an argument. This way, a one step lookahead can be performed from any state regardless of the state that is currently stored in the class attribute.

A.2.2. Computational Subtleties of PI and VI

As mentioned in chapter 6, a value iteration takes less time than a policy iteration. This is due to a few subtle differences between VI and PI. In every Policy Iteration step, the following actions are performed for each state:

- Update the state value by performing a one step lookahead with the current greedy action (cf. Eq. 2.29).
- Sample a discrete set of actions $\mathcal{A}_D$ and perform a one step lookahead for each action. This yields an estimate of $Q(s, a)$ for each $a \in \mathcal{A}_D$.
- Pick the action that yields the maximum expected return out of $\mathcal{A}_D$ according to Eq. 2.32.

The most computationally expensive part here is the third item, which involves performing multiple one step lookaheads, picking the largest action value from a list, finding the respective list index and picking the greedy action from $\mathcal{A}_D$. Another disadvantage

is that PI requires more memory, because the greedy actions have to be stored explicitly for each iteration i, as they represent π_i which is required for iteration $i+1$.

In value iteration, every iteration step involves the following actions:

- Sample a discrete set of actions $\mathcal{A}_D$ and perform a one step lookahead for each action. This yields an estimate of $Q(s,a)$ for each $a \in \mathcal{A}_D$.
- Pick the maximum action value out of all $Q(s,a)$ from step one and save it as the new state value of s_t (cf. Eq. 2.36[1]

Both algorithms perform multiple one step lookaheads. Value iteration, however, takes less time. Unlike policy iteration, it does not calculate each greedy action explicitly. Instead, it only picks the maximum achievable state value. Only after convergence of VI, one step of policy improvement is performed with the last state value function iterate to achieve a policy (cf. section 2.11.2). In total, value iteration is about 20 % faster than policy iteration.

[1]Note that, in this case, $\max_a[r_{t+1} + \gamma V(s_{t+1})|s_t = s]$ is equivalent to $\max_a[Q(s,a)]$.

A.3. Table Representation of the Value Function and Policy

The state value function and the greedy action for each state on the grid is saved in a four-dimensional table, which has been implemented in the context of this work. The tablee is implemented as a python class and contains methods to read data from a table cell and update cell contents. Because it also contains information about the state grid itself, it can transform a physical state into a set of table indices and back. This makes its usage in the DP algorithms simple. Every cell can either be accessed by an array of four physical coordinates $[x, z, u, w]$ or four indices $[i_x, i_z, i_u, i_w]$. Unlike two-dimensional tables, a 4D table cannot be visualized easily. Fig. A.1 visualizes how a four-dimensional table works by means of multiple 2D tables.

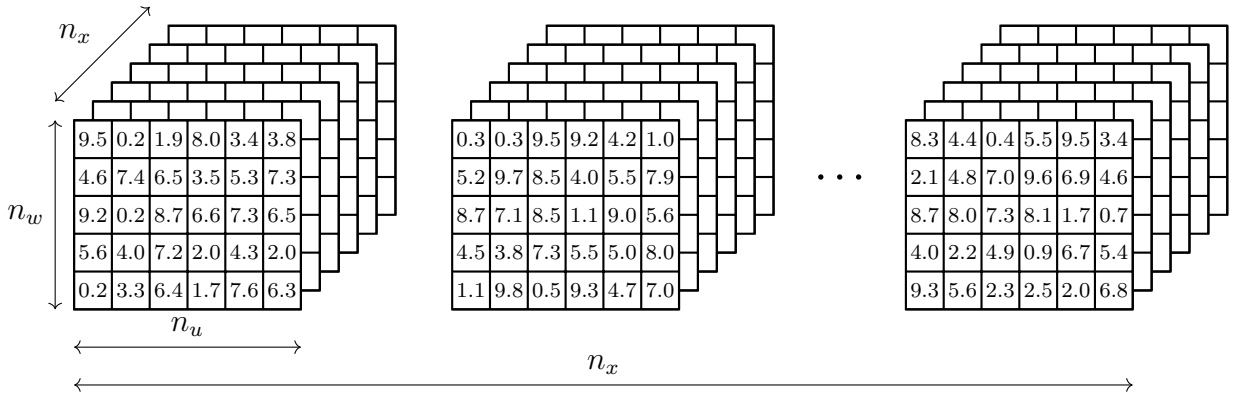

Figure A.1.: A four-dimensional table.

A 3D table can be considered as a stack of 2D tables. Likewise, a 4D table can be considered as a stack of 3D tables.

A.4. Optimistic Policy Iteration

Algorithm 4 Optimistic Policy Iteration

function OPTIMISTIC POLICY ITERATION

Initialize $V(s) = 0 \; \forall \; s \in \mathcal{S}$
Load arbitrary initial policy π_0.
Initialize m, ϵ_V
while true **do**

function POLICY EVALUATION
for all $s \in \mathcal{S}$ **do**
$V_{\pi,new}(s) \leftarrow\!\shortmid r + \gamma V_{\pi,old}(s')$
end for
end function

function POLICY IMPROVEMENT
for all $s \in \mathcal{S}$ **do**
sample m actions $a_n(s)$
for all a_n **do**
$Q(s, a_n) = r + \gamma V(s')$
end for
$a_{greedy}(s) = \underset{a_n}{\text{argmax}}[Q(s, a_n)]$
end for
end function
Train policy on $a_{greedy,new}$ to obtain π_{new}

if $||a_{greedy,new} - a_{greedy,old}||_\infty = 0$ **then**
break
end if
end while

end function

Bibliography

[1] *rllab-website*. https://rllab.readthedocs.io/en/latest/. Version: 2018. – visited on August 17, 2018

[2] Abbeel, P. ; Schulmann, J. : *Deep Reinforcement Learning through Policy Optimization*. https://people.eecs.berkeley.edu/~pabbeel/nips-tutorial-policy-optimization-Schulman-Abbeel.pdf. Version: 2016

[3] Bellman, R. E.: The Theory of Dynamic Programming. (1954)

[4] Bellman, R. E.: Applied Dynamic Programming. (1962)

[5] Fichter, W. ; Grimm, W. : *Flugmechanik*. Shaker Verlag, 2009

[6] Kingma, D. P. ; Ba, J. : Adam: A Method for Stochastic Optimization. In: *CoRR* abs/1412.6980 (2014). http://arxiv.org/abs/1412.6980

[7] Kriesel, D. : *A Brief Introduction to Neural Networks* availableathttp://www.dkriesel.com

[8] Notter, S. ; Zuern, M. ; Gross, P. ; Fichter, W. : Reinforced Learning to Cross-Country Soar in the Vertical Plane of Motion. (2018)

[9] Powell, W. B.: *Approximate dynamic programming: solving the curses of dimensionality*. Wiley-Interscience (Wiley series in probability and statistics). – XVI, 460 Seiten S. http://swbplus.bsz-bw.de/bsz275179400cov.htm. – ISBN 978–0–470–17155–4. – UB Vaihingen

[10] Silver, D. : *UCL Course on Reinforcement Learning*. 2015

[11] Sutton, R. S. ; Barto, A. G.: *Reinforcement Learning - An Introduction*. The MIT Press, 2018

[12] Zuern, M. : *Autonomous Soaring through Unknown Atmosphere, Master Thesis*. Institute for Flight Mechanics and Control, Stuttgart University, 2017