Autonomous Soaring Policy Initialization by 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

Erklärung iii

Kurzfassung / Abstract v

List of Figures ix

List of Tables xi

Nomenklatur xiii

1. Introduction 1

2. Reinforcement Learning 3
- 2.1. Markov Decision Process 3
- 2.2. The Agent - Environment System 3
- 2.3. Model Based and Model Free Learning 4
- 2.4. Agent 5
- 2.5. Return 6
- 2.6. Policy 6
- 2.7. Value Functions 7
- 2.8. The Bellman Equation 8

3. Dynamic Programming 11
- 3.1. Approximate Dynamic Programming 13
- 3.2. Reward Function 13
- 3.3. The Principle of Optimality 13
- 3.4. Types of Dynamic Programming Algorithms 14
  - 3.4.1. Policy Evaluation 14
  - 3.4.2. Policy Iteration 15
  - 3.4.3. Value Iteration 17
- 3.5. The Contraction Mapping Theorem 17

4. Function Approximation 21
- 4.1. Tables 21
- 4.2. Artificial Neural Networks 22
- 4.3. Supervised Learning 26

4.4. Optimization Techniques 27
4.4.1. Gradient Descent 27
4.4.2. Stochastic Gradient Descent 28
4.4.3. The ADAM Algorithm 28
4.5. Overfitting 29

5. Trajectory Optimization with Policy Iteration 31
5.1. 2D Environment 31
5.1.1. Policy Representation 31
5.1.2. Equations of Motion 31
5.1.3. Discretization of the State- and Action Space 33
5.2. 3D Environment 34
5.2.1. Discretization of the state and action space 34

6. Results 35
6.1. 2D Policy Iteration 35
6.2. 3D Scenarios 37

7. Discussion 39

A. Appendix A: Glider Parameters and Scenario Data 41

B. Appendix B: Computer Configuration 43

C. Appendix C: Table Representation of the Value Function and Policy 45

Literaturverzeichnis 47

List of Figures

2.1. The Agent-Environment-System [6] 4

3.1. Classification of Machine Learning algorithms 11
3.2. Figure 3.1 von Markus 12
3.3. The Policy Iteration Algorithm 16

4.1. A neuron from an artificial neural network [10] 23
4.2. Activation functions [10] 24
4.3. The MLP used for the policy [10] 25

5.1. Angles in three-dimensional flight. 32
5.2. The discretized state space. For x_3 and z_4, the grid for the speed vector is drawn. 33

6.1. Trajectory and angle of attack after 100 Value Iterations 37
6.2. Trajectory and angle of attack after 100 iterations of Generalized Policy Iteration with 100 evaluation steps in each iteration 38

List of Tables

5.1. Grid parameters for trajectory optimization 34

6.1. Comparison of flight- and computation-time for OC, PI and VI 36
6.2. Comparison of flight- and computation-time for OC, PI and VI 38

A.1. glider data 41
A.2. scenario data 41

B.1. computer specifications 43

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
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

$\boldsymbol{b}$		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	$\left[\frac{\mathrm{kg}}{\mathrm{m}^3}\right]$	air density

a		action
m	[kg]	glider mass
o		observation
Q		Action Value Function
q	$\left[\frac{\mathrm{N}}{\mathrm{m}^2}\right]$	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. The basic idea is to use the knowledge about the glider dynamics to pre-train a policy. After that, a reinforcement learning 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. Reinforcement Learning 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 interacting 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 very successfully applied Trust Region Policy Optimization (TRPO) and Asynchronous Advantage Actor Critic (A3C) to a trajectory optimization problem[10]. 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 [10] 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. 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. 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 [7](todo: hier noch das Kapitel einfügen). 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 5.1.3 and 5.2.1. 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 (c.f. section 6.2) 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 6.1 are also relevant.

Essentially, a pilot who is confronted with unknown updrafts would at first also rely on his experience from prior flights, be it through calm air or not.

2. Reinforcement Learning

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).

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

A Markov Decision Process is also 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 2.7 If π^* is found for a specific MDP, the MDP is solved.

2.2. The Agent - Environment System

In Reinforcement Learning scenarios, an agent interacts with its environment and thereby tries to maximize some sort of reward. By interacting with its environment, it learns how to behave optimally with respect to maximizing a 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 belonging to the next state s_{t+1}. The process is repeated, then. Figure 2.1 illustrates what happens at each time step.

The reward r_t the agent receives at each step is determined by a reward function $\mathcal{R}(s,a)$. 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 give the agent a positive reward for actions that bring him closer to his goal and penalize actions that reduce his chances of reaching it. If an action takes the agent to a terminal state that is not the goal, this action should yield a big penalty. Penalties are a means of keeping the agent away from states that are not desirable, such as unwanted terminal states or states in their direct vicinity. More on the reward function used here can be found in chapter 3

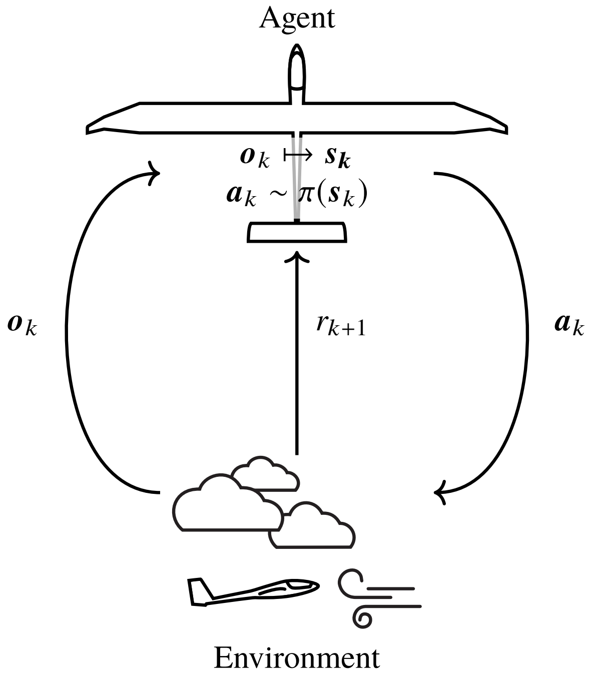
Agent
$\boldsymbol{o}_k \mapsto \boldsymbol{s}_k$
$\boldsymbol{a}_k \sim \pi(\boldsymbol{s}_k)$
$\boldsymbol{o}_k$
r_{k+1}
$\boldsymbol{a}_k$
Environment

Figure 2.1.: The Agent-Environment-System [6]

2.3. 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 stochastic, there are multiple possible next states and rewards. A *sample model* returns one of the possibilities, whereas a *distribution model* returns some representation of all possible next states and rewards.

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. To distinguish experience from a model from real experience, the results from a model are referred to as *simulated experience*[9]. Figure 3.2 shows some popular Machine Learning algorithms and classifies them.

2.4. 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 2.6)
- Value Function:
  A mapping from states (or state-action pairs) to values (c.f. section 2.7)
- Model:
  A model of the environment that might be updated by the agent's experience

Most agents have a policy. They also can have a value function, like in actor-critic algorithms where the value function is used for bootstrapping. This means utilizing the Bellman Expectation Equation to estimate the return G_t by doing a one step lookahead and use the sum of the immediate reward and the successor state value.

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

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

A model of the environment is - in principle - not required for successful training. It can however be used to gather simulated experience 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.

2.5. 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:

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

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 3.2

2.6. 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.

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

The policy is a probability distribution over actions given states and returns - for each state - the probability of taking an action a_t. If the policy is deterministic, only one a_t out of $\mathcal{A}$ has a probability of one with all other actions assigend 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 **??**.

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

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

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.

2.7. 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.

2.8. The Bellman Equation

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. Equivalently, every state value can be split into the value of the next step and the successor state value.

The state value function 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*:

$$\begin{aligned} V_\pi(s) &= \sum_a \pi(a|s) Q_\pi(s,a) && (2.14) \\ &= \sum_a \pi(a|s) \sum_r \sum_{s'} \mathcal{P}(s', r|s,a)[r + \gamma V_\pi(s')] && (2.15) \end{aligned}$$

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 3.8 and 3.9).

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.

$$\begin{aligned} V_*(s) &= \max_{a \in \mathcal{A}(s)} Q_{\pi_*}(s,a) && (2.17) \\ &= \max_a \mathbb{E}_{\pi_*}[G_t | s_t = s, a_t = a] && (2.18) \\ &= \max_a \mathbb{E}_{\pi_*}[r_{t+1} + \gamma G_{t+1} | s_t = s, a_t = a] && (2.19) \\ &= \max_a \mathbb{E}[r_{t+1} + \gamma V_*(s_{t+1}) | s_t = s, a_t = a] && (2.20) \\ &= \max_a \sum_{s',r} \mathcal{P}(s', r|s,a)[r + \gamma V_*(s')] && (2.21) \end{aligned}$$

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 [9]. 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 3.4.3 for more details.

3. Dynamic Programming

Figure 3.1.: Classification of Machine Learning algorithms

This thesis deals with Dynamic Programming for policy optimization. Markus Zuern very successfully applied Trust Region Policy Optimization (TRPO) and Asynchronous Advantage Actor Critic (A3C) to a 2D trajectory optimization problem[10]. In his work, 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} (todo: dies überall übernehmen) then can be used as a starting point for TRPO in scenarios with unknown wind conditions.

Figure 3.2.: Figure 3.1 von Markus

The term Dynamic Programming is used for optimization problems that 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 is connected to its successor state(s) by one of the Bellman Equations.

Also, the optimal action is found in all states the same way, as can be seen in section 3.4.2. If the algorithm is successful, the policy is optimal according to section 3.3. 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 "saved" in the policy.

3.1. Approximate Dynamic Programming

(hier noch einfügen, wie DP mit RL zusammenhängt)

3.2. 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 and should not do.

The reward function in this work is as follows:

$$\mathcal{R}(s,a) : \mathcal{S} \mapsto \mathbb{R} : r_t = \begin{cases} -\Delta t & \text{if } s_{t+1} \neq s_{T+} \\ -\Delta t + \frac{d}{10} & \text{if } s_{t+1} = s_{T+} \end{cases} \tag{3.1}$$

(todo: abbildung dazu?)

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. 3.1.

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. If the agent has to cover a distance of 500 m, he receives a final reward of $+50$. with a mean ground speed of about $15 - 20\frac{\mathrm{m}}{\mathrm{s}}$, covering 500 m takes between 25 and 33.3 s. If the agent gets -1 reward per second it takes him to reach 500 m, the expected return from a state far away from the goal can still get up to about $33.3 \cdot (-1) + 50 = 16.7$. This would not work if the final reward was only $+\frac{\mathrm{d}}{20}$. 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 $s \in \mathcal{S}$.

3.3. 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 term *principle of optimality* was first used by Richard E. Bellman in 1954 [1]. 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 [2].

3.4. Types of Dynamic Programming Algorithms

3.4.1. 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 3.4 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 $V_\pi(s)$.

$$\begin{aligned} V_\pi(s_t) &= \mathbb{E}[G_t | s_t = s] && (3.2) \\ &= \mathbb{E}\left[\sum_{k=0}^{T-t-1} \gamma^k r_{t+k+1} | s_t = s\right] && (3.3) \\ &= \mathbb{E}[r_{t+1} + \gamma^1 r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + ... + \gamma^{T-t-1} r_T | s_t = s] && (3.4) \end{aligned}$$

With a stochastic discrete policy, equation 3.4 becomes

$$V_\pi(s_t) = \sum_a \pi(a|s)(\mathcal{R}(s,a) + \gamma V_\pi(s')) \tag{3.5}$$

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.

$$V_\pi(s_t) \mapsfrom \mathbb{E}[\sum_{k=0}^{T-t-1} \gamma^k r_{t+k+1} | s_t = s] \tag{3.6}$$

In Eq. 3.4, all terms expect the first one can be replaced by the expected return from s_{t+1} onwards:

$$\begin{aligned} V_\pi(s_t) \mapsfrom &= \mathbb{E}[r_{t+1} + \gamma G_{t+1} | s_t = s] && (3.7) \\ &= r_{t+1} + \gamma V_\pi(s_{t+1}) && (3.8) \end{aligned}$$

With a discrete stochastic policy, equation 3.8 becomes

$$V_\pi(s_t) \hookleftarrow \sum_a \pi(a|s)(r_t + \gamma V_\pi(s')) \tag{3.9}$$

In a state space with a finite set of states $\mathcal{S}$, writing down Eq. 3.9 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, 3.4 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. Unlike most RL-algorithms, the state values are not updated along a trajectory. Instead, each state is updated independently.[1]

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 3.5)

$$||V_k - V_{k-1}||_\infty < \epsilon_V \tag{3.10}$$

3.4.2. 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, a new policy is generated that chooses the greedy action with respect to V_π in each state. This alternating process of Policy Evaluation and Policy Improvement is repeated until the policy is satisfactory.

$$a_{\text{greedy}}(s_t) = \underset{a}{\text{argmax}}[Q(s_t, a_t)] \tag{3.11}$$

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

$$\begin{align} \pi(a_{\text{greedy}}|s_t) &= \mathbb{P}[a_t = a_{\text{greedy}}(s_t)|s_t] \tag{3.12} \\ &= 1 \tag{3.13} \end{align}$$

and for a stochastic policy π:

---

[1] 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.

$$\mathbb{E}[\pi(a_t|s_t)] \mapsfrom a_{\text{greedy}} \tag{3.14}$$

As mentioned in subsection 3.4.1, 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. Equation 3.4.2 illustrates the policy iteration scheme.

$$\pi_0 \xrightarrow{evaluate} V_1 \xrightarrow{improve} \pi_1 \xrightarrow{evaluate} V_2 \xrightarrow{improve} \pi_2 \xrightarrow{evaluate} \dots \xrightarrow{evaluate} V_* \xrightarrow{improve} \pi_*$$

Another way to picture Policy Iteration is shown in figure 3.3. Every PI-step brings the Value Function $V(s)$ closer to $V_*(s)$ and the policy π closer to π_*.

Figure 3.3.: The Policy Iteration Algorithm

The performance of both algorithms on the glider-problem is compared in chapter 6.

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 6 show, PI and VI converge much faster in practice than the upper bound $|\mathcal{A}|^{|\mathcal{S}|}$ may suggest.

3.4.3. 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 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. [8], lecture 3). In a possibly discounted MDP with a deterministic environment, the update rule for each state value can be seen in equation 3.15.

$$V(s_t) \hookleftarrow \max_a [r_{t+1} + \gamma V(s_{t+1}) | s_t = s] \tag{3.15}$$

In principle, this is equivalent to 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 bound calculation time. In this work, iteration is stopped if $||V_k - V_{k-1}||_\infty < \epsilon_V$, identical to equation 3.10. 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.

3.5. The Contraction Mapping Theorem

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

$$|f(x_1) - f(x_2)| \leq \gamma |x_1 - x_2| \tag{3.16}$$

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 3.9 is known as the *Bellman Operator* $T_\pi(v(s')) = \mathcal{R}^\pi + \gamma \mathcal{P}^\pi v(s')$. 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 $T^n(v)$.

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:

$$||V||_\infty = \max_{s \in \mathcal{S}} |V(s)| \tag{3.17}$$

The supremum norm of $V_1 - V_2$ is therefore

$$||V_1(s) - V_2(s)||_\infty = \max_{s \in \mathcal{S}} |V_1(s) - V_2(s)| \tag{3.18}$$

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.

$$\begin{aligned} ||T^\pi(V_1) - T^\pi(V_2)||_\infty &= ||(\mathcal{R}^\pi + \gamma \mathcal{P}^\pi V_1) - (\mathcal{R}^\pi + \gamma \mathcal{P}^\pi V_2)||_\infty && (3.19) \\ &= ||\gamma \; \mathcal{P}^\pi \; (V_1 - V_2)||_\infty && (3.20) \\ &\leq ||\gamma \mathcal{P}^\pi ||V_1 - V_2||_\infty||_\infty && (3.21) \\ &\leq ||\gamma ||V_1 - V_2||_\infty||_\infty && (3.22) \\ &= \gamma ||V_1 - V_2||_\infty && (3.23) \end{aligned}$$

Whenever s' happens to be the state where V_1 and V_2 differ the most, equality holds in line 3.21. 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 3.21, equality in line 3.22 holds only if $\mathcal{P}^\pi = 1$. Replacing $\mathcal{P}^\pi$ by one in line 3.21 yields line 3.22. The left side of line 3.19 must thus be less than line 3.23 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 3.24.

$$T^\pi(v(s)) = \mathcal{R}^\pi + \mathcal{P}^\pi V(s') \tag{3.24}$$

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 3.25 shows the n-step contraction property. For simplicity, the index π and the argument s are omitted.

$$||T^{(n)}(V_1) - T^{(n)}(V_2)||_\infty \leq ||V_1 - V_2||_\infty \tag{3.25}$$

Equation 3.25 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 in undiscounted MDPs.

The following proof is more sophisticated than for discounted MDPs. (todo: Proof)

4. 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 where 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 of x_{fit} and corresponding y_{fit} 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.

4.1. 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.

A written version of the value function table and the table containing the greedy action for each state can be found in appendix **??**.

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 quite 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:

$$y_{interp} = y_0 \cdot (1 - \gamma) + y_1 \cdot \gamma \tag{4.1}$$

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.

4.2. Artificial Neural Networks

The structure of artificial neural networks resembles the structure of a human brain according to the current state of research [5, Chapter 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.

A neuron in an artificial neural network looks like what is depicted by Fig. 4.1. 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 nonlinear function, the so called activation function h, yielding z, the neuron output. This output can either serve as an input to another neuron or the network output. Multiple activation functions are common, but they all are somewhat s-shaped. Most 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-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, does not make sense.

Figure 4.1.: A neuron from an artificial neural network [10]

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 $k+1$ (see figure 4.1).

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 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:

$$\pi_\theta(a|s) = \mathbb{P}[a|s, \theta] \tag{4.2}$$

with

Figure 4.2.: Activation functions [10]

$$\theta = \{\boldsymbol{W}^{(1)}, \boldsymbol{b}^{(1)}, \boldsymbol{W}^{(2)}, \boldsymbol{b}^{(2)}, ..., \boldsymbol{W}^{(k)}, \boldsymbol{b}^{(k)}\} \tag{4.3}$$

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:

$$\pi_\theta(\mu_\theta, \sigma_\theta | s) = \frac{1}{\sqrt{2\pi}\sigma_\theta} e^{-\frac{(a-\mu_\theta)^2}{2\sigma_\theta^2}} \tag{4.4}$$

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 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.

Figure 4.3.: The MLP used for the policy [10]

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 4.7 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 $\sigma = 1$:

$$\hat{x} = \frac{x - \mu}{\sigma} \tag{4.5}$$

The specific values for μ and σ for each of the scenarios are presented in table A.2.

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, then so are the neuron outputs. The $tanh(\cdot)$ 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 and linear behavior can also be obtained without any hidden layers. The most important advantage of deep neural networks is however their ability to represent highly nonlinear 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 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.

4.3. 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):

$$L = \frac{1}{N} \cdot \sum_{n=1}^{N} L_n = \frac{1}{N} \cdot \sum_{n=1}^{N} [\frac{1}{2} \cdot (a_n(s) - a_{n,fit})^2] \tag{4.6}$$

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.

4.4. 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 4.4.1 and 4.4.2 explain briefly how gradient based optimization works in general. In section 4.4.3, 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 k-th optimization step is defined by a distinct set of weights and biases θ, similar to coordinates in the state space (c.f. Eq. 4.3).

4.4.1. 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:

$$\theta_{k+1} = \theta_k - \nabla_{\theta_k} L_k \cdot \alpha \tag{4.7}$$

with

$$\nabla_{\theta_k} L_k = \frac{1}{N} \cdot \sum_{n=1}^{N} \nabla_{\theta_k} L_{n,k} \tag{4.8}$$

$$= \frac{1}{N} \cdot \sum_{n=1}^{N} [(a_n(s) - a_{n,fit}) \cdot \nabla_{\theta_k} a_n(s)] \tag{4.9}$$

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.

4.4.2. 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. 4.6, 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:

$$\nabla_{\theta_k} L_k \approx \nabla_{\theta_k} L_{n,k} \tag{4.10}$$

$$\theta_{k+1} = \theta_k - \nabla_{\theta_k} L_{n,k} \cdot \alpha \tag{4.11}$$

If this is done for all samples successively, it also converges to the loss from Eq.4.6. Thanks to the faster computation of $\nabla_{\theta_k} L_{n,k}$, SGD converges faster than ordinary gradient descent although Eq.4.10 is only a rough estimate of the true gradient.

4.4.3. 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.4.11. 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 4.12 shows one update step, the parameters $\hat{m}_t$ and $\hat{v}_t$ are calculated with equations 4.13 and 4.14.

$$\theta_{k+1} = \theta_k - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{4.12}$$

with

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} = \frac{\beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t}{1 - \beta_1^t}, \tag{4.13}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} = \frac{\beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2}{1 - \beta_2^t} \tag{4.14}$$

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 4.13 and 4.14. 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 4.13 and 4.14 to account for that. See [4] for more details.

4.5. 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 the training data is noisy, i.e. each data point differs slightly from the expected value, and if a high number of parameters is trained with a limited dataset.

(todo: hier noch ein Bild mit einem overgefitteten Polynom einfügen)

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.

(todo: Bild von overfitting einfügen)

The training set can also be split into three datasets, one for training, one for validation during training, and the third one to check for overfitting after training. 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 in the hidden layers.

5. Trajectory Optimization with Policy Iteration

5.1. 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 [3] 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 5.4. This way the agent can control its velocity vector and therefore its position. All other parameters of the glider are shown in appendix A. 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).

5.1.1. 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 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.

5.1.2. Equations of Motion

The physical state of the glider in the geodetic vertical plane consists of the position $_{\boldsymbol{g}}\boldsymbol{r} = [x, z]^T$ and velocity $_{\boldsymbol{g}}\boldsymbol{v} = [_g u_K, {_g w_K}]^T$. 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 5.1 illustrates the transformation angles that connect the reference frames.

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

Figure 5.1.: Angles in three-dimensional flight.

The dynamics in the vertical plane are given by the following equations. They are taken from [3].

$$\dot{x} = V\ cos\phi_K \tag{5.1}$$

$$\dot{z} = -V\ sin\phi_K \tag{5.2}$$

$$\dot{V} = -\frac{D + g\ cos\phi_K}{m} \tag{5.3}$$

$$\dot{\phi}_K = \frac{L}{V\ m} - \frac{g\ cos\phi_K}{V} \tag{5.4}$$

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

5.1.3. 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 $s_n = [x_i, z_j, u_k, w_l]$ evenly distributed across all dimensions.

Figure 5.2.: The discretized state space. For x_3 and z_4, the grid for the speed vector is drawn.

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.

By discretization, the continuous trajectory optimization problem is made time- and space-discrete. For the purpose of keeping calculation time low, a sampling rate Δt of $1s$ is used. With a speed of $15 - 25\frac{\mathrm{m}}{\mathrm{s}}$, the agent covers about $15 - 25\mathrm{m}$ within one time step. The grid points should be separated by approximately that distance in order to

distance	$500m$	$1000m$	$2000m$
n_x	25	50	100
n_z	20	20	20
n_u	10	10	10
n_w	10	10	10

Table 5.1.: Grid parameters for trajectory optimization

make tabular solution methods feasible. For the different scenarios, this yields the grid resolutions shown in table 5.1.

5.2. 3D Environment

todo

5.2.1. Discretization of the state and action space

todo

6. Results

6.1. 2D Policy Iteration

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

- Generalized policy iteration with up to 100 evaluation iterations before each improvement step
- Optimistic policy iteration
- Value iteration

As OPI and VI are technically equivalent (c.f. section 3.4.3), the interesting part here is which algorithm approximates the optimal policy best in minimum time. In the GPI method, the iterative evaluation stops, if the values of all states change by less than the reward for one time step Δt from one evaluation step to the next. 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.

The Policy Iteration has converged if - at the last Policy Improvement step - none of the actions was changed. This means that all actions were already optimal with respect to the current value function.

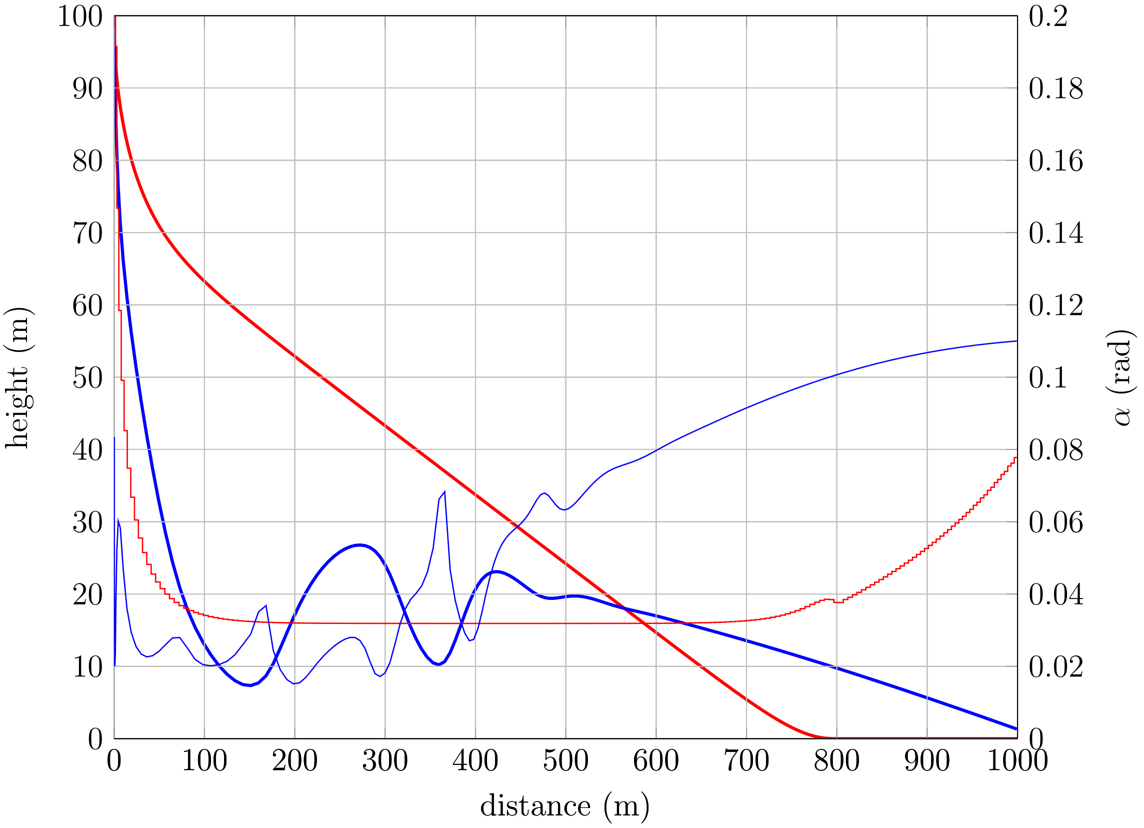

Figure 6.1 shows in blue the trajectory in scenario 1 after 100 iterations of Optimistic Policy Iteration. The thick line is the glider trajectory $[x(t), z(t)]^T$, the thin line is the control sequence $[x(t), \alpha(t)]^T$ The agent is dropped at the start state with zero velocity. As can be seen, both the policy from OPI and the optimal control (red) direct the agent towards the target.

Table 6.2 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.

distance d_T	500m	500m	500m	500m
algorithm	OC	OPI	GPI	VI
calculation time (s)	901.3		7245	3746
flight time (s)	19.67		21.2	21.4

Table 6.1.: Comparison of flight- and computation-time for OC, PI and VI

None of the DP algorithms reaches the optimal flight time of $19.67s$. Each of the policies can however reach the target state $[d_T, z(T), u(T), w(T)]^T$ from any arbitrary

Figure 6.1.: Trajectory and angle of attack after 100 Value Iterations

point in the state space. The optimal control is only valid if the glider starts at the state from which it was calculated.

Although all algorithms theoretically take up to $|\mathcal{A}|^{|\mathcal{S}|}$ iterations to converge (c.f. section 3.4.2), they are known to typically converge after surprisingly few iterations.

During each Policy Iteration step with GPI, it takes between 60 and 70 policy evaluation steps until approximate convergence. Therefore each policy iteration takes significantly longer in the GPI case than with optimistic policy iteration where evaluation is stopped after only one evaluation step.

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

6.2. 3D Scenarios

Figure 6.2.: Trajectory and angle of attack after 100 iterations of Generalized Policy Iteration with 100 evaluation steps in each iteration

distance d_T	1000m	1000m	1000m	1000m
algorithm	OC	OPI	GPI	VI
calculation time (s)	5484.5			
flight time (s)	47.04			

Table 6.2.: Comparison of flight- and computation-time for OC, PI and VI

7. Discussion

Platzhalter...

A. Appendix A: Glider Parameters and Scenario Data

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

scenario	1	2	3
distance d_T	500m	1000m	2000m
start state s_0	$s_0 = [0, 100, 0, 0]^T$	$s_0 = [0, 100, 0, 0]^T$	$s_0 = [0, 100, 0, 0]^T$
normalization μ			
normalization σ			

Table A.2.: scenario data

B. Appendix B: Computer Configuration

Table B.1 contains the technical data of the system used for calculations. All calculations were performed in phyton. The policy ANN used in the simulations was handled by the GPU using CUDNN, Theano and Lasagne. All other code was executed with eight parallel processes on the CPU.

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 B.1.: computer specifications

C. Appendix C: Table Representation of the Value Function and Policy

Bibliography

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

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

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

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

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

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

[7] 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

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

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

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