3D Soaring through unknown Atmosphere

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

Nomenklatur

Abkürzungen

Lateinische Buchstaben

Griechische Buchstaben

Indizes

1. Introduction

In autonomous soaring, it is vital to gather as much energy from the atmosphere as possible in order to stay aloft for long time periods. Generally, the locations where updrafts occur are not known a-priori. So a pilot has to react spontaneously on changes in his vicinity in order to harvest as much energy as possible, which takes a lot of experience.

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, 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.

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. The task of finding an optimal path in moving air can be split into finding an optimal path in calm air and dealing with the unknown updrafts separately. 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 simulations. This makes them well suited for learning how to react optimally to random updrafts.

Finding an optimal path through calm air can be done more efficiently, utilizing the knowledge about how a glider behaves in calm air. Flying a glider can be regarded as a time-discrete decision process where, at each time step t, the pilot selects a control surface position. The states s_t in this decision process have the markov property, i.e. the information needed to calculate s_{t+1} is fully contained in s_t. A decision process where the markov property applies to all states is called a Markov Decision Process (MDP). Dynamic Programming is one of the most popular machine learning algorithms for such Markov Decision Processes.

Dynamic Programming is a general term that applies to 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", in large or continuous state spaces it quickly becomes inefficient. The obvious approach is to dis-

cretize the state space and deal with a smaller number of states. If done properly, this yields an approximation of the exact optimal solution. Depending on the problem and the discretization, the achieved approximate solution is more or less close to the exact one. 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. A policy iteration 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. The results from the 2D scenario are also useful where the ground trace of a trajectory is dictated by boundary conditions such as topography or airspace restrictions.

2. Reinforcement Learning

2.1. 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 a scalar reward function. 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 in the environment, 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} and the process is repeated. By interacting with the environment, the agent learns which action is optimal for each state it visits.

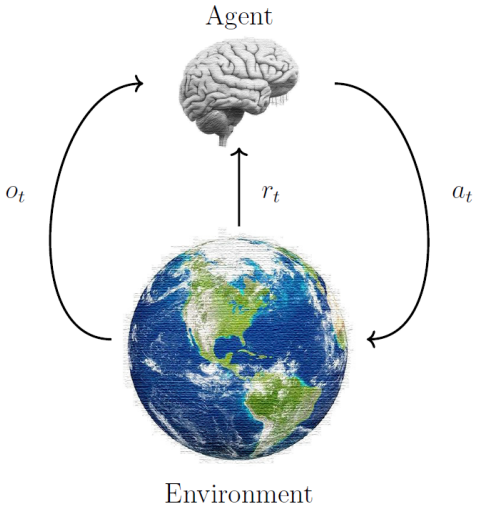

Figure 2.1.: The agent-environment-system [3]

2.2. 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:

$$G_t = \sum_{k=0}^{T-t-1} \gamma^k r_{t+k+1} \tag{2.1}$$

If $\gamma \in [0, 1]$ is close to zero, immediate rewards are weighted more. If γ is close to one, rewards in the distant future are weighted more, 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.

2.3. Markov Decision Process

A Markov Decision Process is a series of decision problems where each decision has an impact on the final reward the agent receives. In an MDP, the probability of reaching a state s_{t+1} only depends on the state s_t. The predecessors of s_t are irrelevant for $P(s_{t+1}|s_t, a_t)$.

$$P(s', r|s, a) = \mathbb{P}[s_{t+1} = s', r_{t+1} = r|s_t = s, a_t = a] \tag{2.2}$$
$$= \mathbb{P}[s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, s_{t-2}, a_{t-2}, ...] \tag{2.3}$$

Every decision process where this probability does not only depend on the direct predecessor can be made markovian by defining additional state variables representing the states that came before. This extended current state contains all relevant information for the transition to s_{t+1}.

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^*. If π^* is found for a specific MDP, the MDP is solved.

2.4. Policy

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

$$\pi : \mathcal{S} \times \mathcal{A} \mapsto [0, 1] \tag{2.4}$$

The policy is a probability distribution over actions given states and returns - at 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. In a stochastic policy, the sum of all possible action-probabilities must be one.

$$\pi(a|s) = \mathbb{P}[a_t = a|s_t = s] \tag{2.5}$$

This yields equations 2.6 and 2.7 for the state transition probability and the reward function under policy π.

$$\mathcal{P}_\pi(s, s') = \sum_a \pi(a|s)\mathcal{P}(s'|s, a) \tag{2.6}$$

$$\mathcal{R}_\pi(s) = \sum_a \pi(a|s)\mathcal{R}(s, a) \tag{2.7}$$

At each timestep, the agent passes its state observation to the policy and the policy returns an action. This action is taken and the agent gets a new observation. 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.

2.5. Value Functions

Some RL-algorithms try to find a policy directly from experience. Others, like A3C, additionally utilize a so called value function. A value function maps from states or actions 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.

$$V(s) = \mathbb{E}[G_t|s_t = s] \tag{2.8}$$

$$Q(a|s) = \mathbb{E}[G_t|s_t = s, a_t = a] \tag{2.9}$$

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 can be implemented with a table, linear combinations of the inputs, basis functions and an artificial neural network.

2.6. The Bellman Expectation Equation

According to the Bellman Expectation Equation, every path 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 2.8 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:

$$V_\pi(s) = \sum_a \pi(a|s) Q_\pi(s,a) = \sum_a \pi(a|s) \sum_r \sum_{s'} \mathcal{P}(s', r|s, a)[r + \gamma V_\pi(s')] \tag{2.10}$$

In a deterministic environment, equation 2.10 becomes

$$V_\pi(s) = \sum_a \pi(a|s)[r + \gamma V_\pi(s')] \tag{2.11}$$

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 (see equations 4.6 and 4.7).

3. Function Approximation

A common problem in all engineering topics 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 and fitting it to given data is called function approximation. It is essential for topics like image recognition, but also regression.

Mathematically, function approximation is the process finding a model function that maps from input variables 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 gives a value close to $y_{fit,i}$ for each $x_{fit,i}$.

In the context of machine learning, artificial neural networks are commonly used for classification, clustering and regression. In classification, the neural network is trained on a set of samples, each belonging to a category which is supplied to the network. As there typically is a finite number of categories, the output of the network is discrete. The training aims to enable the network to put samples it has not been trained on in the correct category. In regression, the network is trained to approximate a function that maps from the input(-vector) to a continuous output(-vector). In this work, for example, the policy is represented as an artificial neural network. In clustering, the network is given a set of training samples without explicitly knowing what cluster they belong to. The training goal is for the network to find appropriate clusters and put the samples into the right cluster so that similar samples are in the same cluster.

3.1. Artificial Neural Networks

The idea behind artificial neural networks is to mimic the structure of a human brain. 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 3.1. It takes an arbitrary number of inputs, multiplies each input by the corresponding input weight, adds a bias and applies to the result a nonlinear function, the activation function. Multiple activation functions are common, but they all are somewhat s-shaped. Examples include the step function, ($f(x) = 1$ if $x \geq x_0$, $f(x) = 0$ otherwise), the rectifier nonlinearity ($f(x) = max(0, x)$), the sigmoid function ($f(x) = 1/(1 - e^{-x})$) and the tanh-activation function ($f(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$).

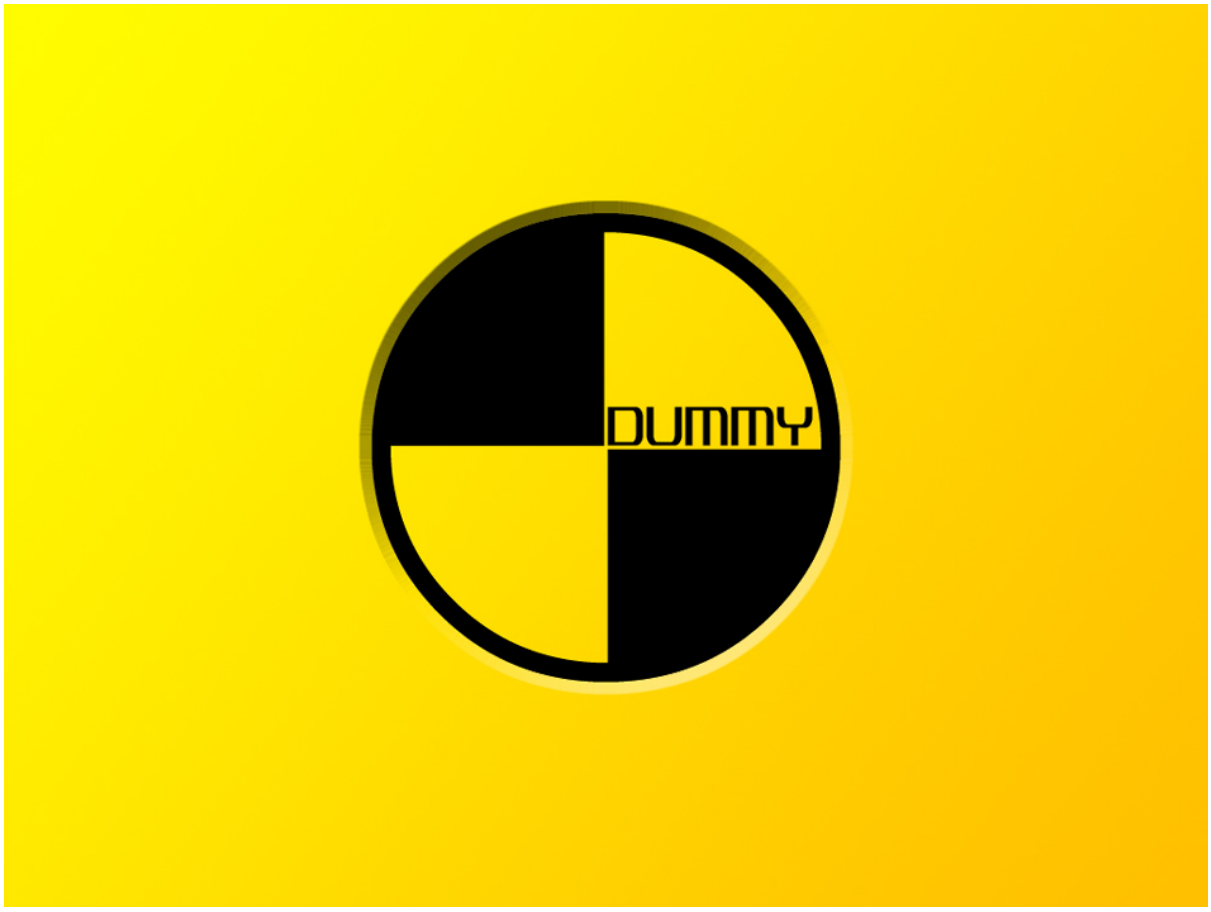

Figure 3.1.: A neuron from an artificial neural network [3]

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 3.1).

Modern computer GPUs also have a high number of processing units and are therefore well suited for parallelized calculations.

3.2. 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

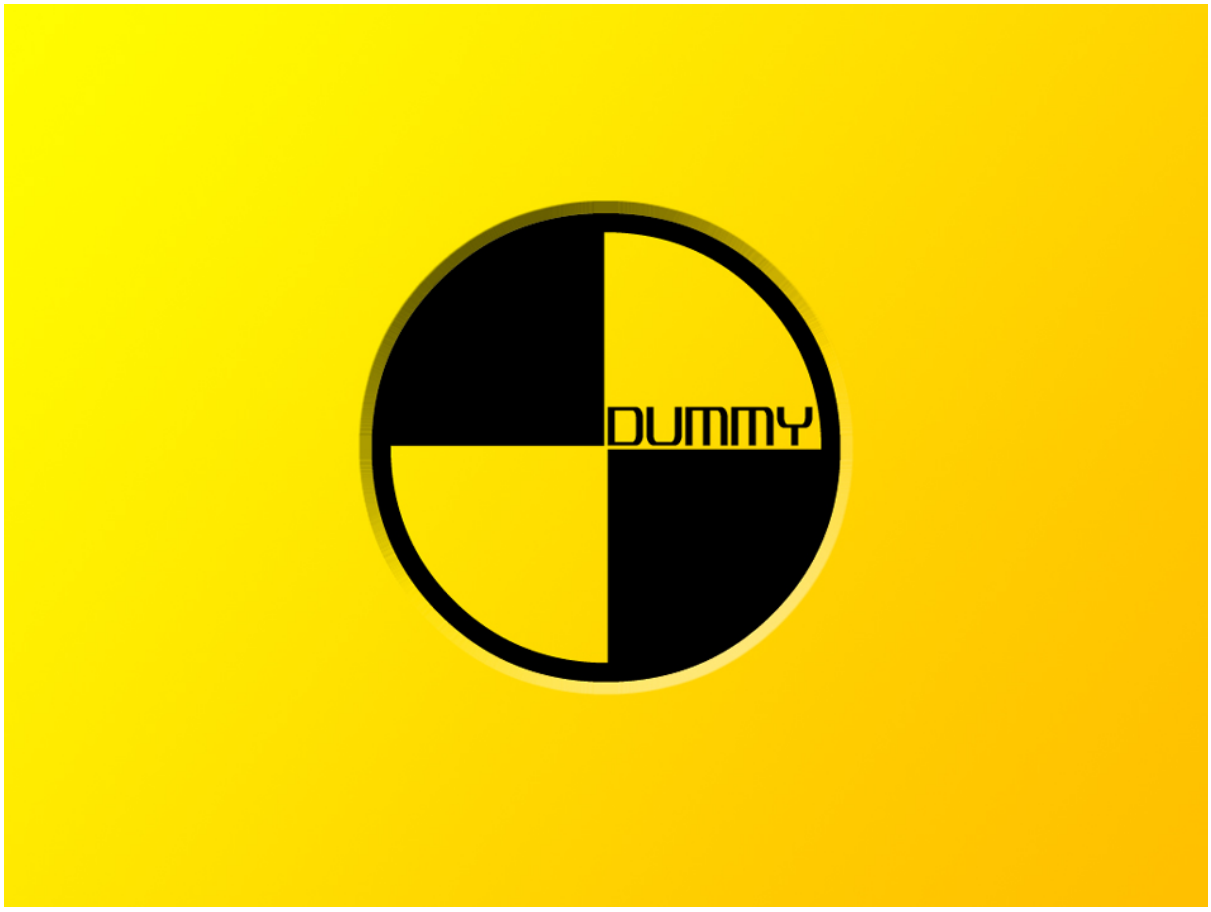

Figure 3.2.: The MLP used for the policy [3]

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):

$$L = \frac{1}{N} \cdot \sum_{n=1}^{N} L_n = \frac{1}{N} \cdot \sum_{n=1}^{N} [\frac{1}{2} \cdot (a_n(s) - a_{n,desired})^2] \tag{3.1}$$

The goal of supervised learning is to minimize the difference between $a_{n,desired}$ 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. If L_{MSE} is minimal, $a_n(s)$ is as close as possible to $a_{n,desired}$ in most states.

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

3.3. 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. Some of the most popular algorithms are presented in the following sections. All presented algorithms produce a series of points in the space created by the network weights and biases. For simplicity, only weights are considered in this chapter. The point at the k-th optimization step is defined by a distinct set of weights and biases W^k, similar to coordinates in the state space.

3.3.1. Gradient Descent

In gradient descent, the next point W^{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, $\nabla_{W^k} L^k$, and performing one step in the direction of steepest descent. W^{k+1} is calculated by the following equation:

$$W^{k+1} = W^k - \nabla_{W^k} L^k \cdot \alpha \tag{3.2}$$

with

$$\nabla_{W^k} L^k = \frac{1}{N} \cdot \sum_{n=1}^{N} \nabla_{W^k} L_n^k = \frac{1}{N} \cdot \sum_{n=1}^{N} [(a_n(s) - a_{n,desired}) \cdot \nabla_{W_k} a_n(s)] \tag{3.3}$$

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 W_{k+1} is close to the respective W_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 $\frac{\partial L_k}{\partial W_k}$ for all the weights and biases is very computation-intensive and time-consuming.

3.3.2. 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 is Stochastic Gradient Descent. If the loss function is a sum like in eq. 3.1, 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 opposite direction:

$$\nabla_{W^k} L^k \approx \nabla_{W_k} L_n^k \tag{3.4}$$

$$W^{k+1} = W^k - \nabla_{W_k} L_n^k \cdot \alpha \tag{3.5}$$

If this is done for all samples successively, it also converges to the loss from eq.3.1. Thanks to the faster computation of $\nabla_{W^k} L_n^k$, SGD converges faster than ordinary gradient descent although eq.3.4 is only a rough estimate of the true gradient.

3.3.3. Momentum

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.3.5. 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.

3.3.4. The ADAM Algorithm

The ADAM-algorithm utilizes a modified momentum term for better convergence. Equation 3.6 shows one update step, the parameters $\hat{m}_t$ and $\hat{v}_t$ are calculated with equations 3.7 and 3.8.

$$W_{k+1} = W_k - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{3.6}$$

with

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} = \frac{\beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t}{1 - \beta_1^t} \tag{3.7}$$

and

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} = \frac{\beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2}{1 - \beta_2^t} \tag{3.8}$$

The algorithm utilizes the first and second moments of the previous gradients to calculate the update. 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 3.7 and 3.8 to account for that. See [2] for more details.

3.4. Overfitting

It is not always desirable to minimize the loss function to the absolute minimum of the loss function on the training data. 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 only 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 a policy is trained 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 datapoint differs slightly from the expected value, and if a high number of parameters is trained with a limited dataset.

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 or even increases, overfitting is likely. This can be used as a criterion to stop the training.

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.

4. Dynamic Programming

The term Dynamic Programming is used for optimization problems that can be decomposed into subproblems and solved by solving those subproblems. The solution to the original problem is then recomposed from the solutions of the 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.

4.1. The Principle of Optimality

According to the 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 also in every successor state it visits s_{t+k} ($k \in [1, T-t]$), the resulting trajectory is the optimal one from s_t to the terminal state.

4.2. Types of Dynamic Programming Algorithms

4.2.1. Policy Evaluation

In Policy Evaluation, the goal is to find 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.

$$V(s_t) = \mathbb{E}[G_t | s_t = s] = \mathbb{E}[\sum_{k=0}^{T-t-1} \gamma^k r_{t+k+1} | s_t = s] \tag{4.1}$$

With a stochastic discrete policy, equation 4.1 becomes

$$V(s_t) = \sum_a \pi(a|s)(\mathcal{R}(s,a) + \gamma V_\pi(s')) \tag{4.2}$$

Equation 4.1 can be used to update the value of s_t with the expected return according to the current policy.

$$V(s_t) \leftarrow\!\rightarrow \mathbb{E}[G_t|s_t = s] = \mathbb{E}[\sum_{k=0}^{T-t-1} \gamma^k r_{t+k+1}|s_t = s] \tag{4.3}$$

$$= \mathbb{E}[r_{t+1} + \gamma^1 r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + ... + \gamma^{T-t-1} r_T|s_t = s] \tag{4.4}$$

In 4.1, all terms expect the first one can be replaced by the expected return from s_{t+1}:

$$V(s_t) \leftarrow\!\rightarrow = \mathbb{E}[r_{t+1} + \gamma G_{t+1}|s_t = s] \tag{4.5}$$

$$= \mathbb{E}[r_{t+1} + \gamma V(s_{t+1})|s_t = s] \tag{4.6}$$

With a discrete stochastic policy, equation 4.6 becomes

$$V_{new}(s_t) \leftarrow\!\rightarrow \sum_a \pi(a|s)(r_t + \gamma V_\pi(s')) \tag{4.7}$$

In a state space with a finite set of states $\mathcal{S}$, writing down 4.7 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, 4.1 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 at the same time.

This process is repeated until the maximum change in values lies beneath a certain threshold ϵ.

4.2.2. 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.

$$a_{greedy}(s_t) = \underset{a}{argmax}[Q(s_t, a_t)] \tag{4.8}$$

For a deterministic policy π, this yields:

$$\pi(a_{greedy}|s_t) = \mathbb{P}[a_t = a_{greedy}(s_t)|s_t] \leftarrow\!\rightarrow 1 \tag{4.9}$$

and for a stochastic policy π:

$$\mathbb{E}[\pi(s_t)] \leftarrow\!\rightarrow a_{greedy} \tag{4.10}$$

4.2.3. 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 not necessarily corresponding to a policy, but each one being a better estimate of v^* than its predecessor.

$$V(s_t) \leftarrow\!\shortmid \max[r_{t+1} + \gamma V(s_{t+1}) | s_t = s] \tag{4.11}$$

4.3. The Contraction Mapping Theorem

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

$$|f(x_1) - f(x_2)| \leq \alpha |x_1 - x_2| \tag{4.12}$$

with $x_1, x_2 \in M$ and $\alpha \in [0, 1)$. Instead of $|x_1 - x_2|$ and $|f(x_1) - f(x_2)|$, any measure of the distance between x_1 and x_2 or $f(x_1)$ and $f(x_2)$ can be used.

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

Policy Evaluation is a contraction mapping.

(...hier den Beweis für Policy Evaluation Gleichungen von Silver einfügen)

5. Trajectory Optimization with Policy Iteration

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 known information about the glider dynamics to pre-train a policy that is - in the next step - improved by reinforcement learning.

5.1. Glider Representation

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 [1] for more details.

5.1.1. Maximum Range in a given Configuration

An airplane with no propulsion system moving through calm air reaches its maximum range when flying at the maximum ratio of lift to drag. Lift and drag can be expressed as the product of dynamic pressure q, a reference area S and a dimensionless coefficient c_l or c_d:

$$L = q \cdot S \cdot c_l = \frac{\rho}{2} v^2 \cdot S \cdot c_L \tag{5.1}$$

$$D = q \cdot S \cdot c_d = \frac{\rho}{2} v^2 \cdot S \cdot c_D \tag{5.2}$$

$$\frac{L}{D} = \frac{c_L}{c_D} \tag{5.3}$$

The optimal lift to drag ratio is also the optimal ratio of c_L and c_D:

$$(\frac{L}{D})_{max} = (\frac{c_L}{c_D})_{max} = (\frac{c_L}{c_D})_{opt} \tag{5.4}$$

With a symmetric drag polar model, this yields:

$$c_{L,opt} = c_L((\frac{c_L}{c_D})_{opt}) = \sqrt{\frac{c_{d0}}{k}} \tag{5.5}$$

and

$$\alpha_{opt} \approx \frac{\Lambda + 2}{2\pi\Lambda} \cdot c_{L,opt} \tag{5.6}$$

A policy that is trained to return α_{opt} for every state can be used as a starting point for policy optimization with a Dynamic Programming algorithm.

5.1.2. Horizontal Flight

todo

5.1.3. Following the Line of Sight Towards the Target

At each state in the state space, the line of sight from the agent to the target can be compared to the

5.2. 2D Environment

In this scenario, the glider moves in the geodetic vertical plane. Its control is the angle of attack α. All other parameters of the glider are shown in appendix (verweis!).

5.2.1. Equations of Motion

In the vertical plane, there are 4 state variables, x, z, $u = \dot{x}$ and $w = \dot{z}$. u and w can also be expressed by V and γ.

The dynamics in the vertical plane are given by the following equations.

$$\dot{x} = V \cdot cos\gamma \tag{5.7}$$

$$\dot{z} = -V \cdot sin\gamma \tag{5.8}$$

$$\dot{V} = -\frac{D + g \cdot cos\gamma}{m} \tag{5.9}$$

$$\dot{\gamma} = \frac{A}{V \cdot m} - \frac{g \cdot cos\gamma}{V} \tag{5.10}$$

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

5.2.2. 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.

In the 2D-scenario, the action space is one-dimensional, the only action is 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 reward out of the sampled actions is an approximation of the true greedy action.

5.3. 3D Environment

5.3.1. Equations of Motion

In three dimensional state space, the dynamics of a mass-point are as follows.

$$\dot{x} = V \cdot cos(\gamma) \cdot cos(\chi) \tag{5.11}$$

$$\dot{y} = V \cdot cos(\gamma) \cdot sin(\chi) \tag{5.12}$$

$$\dot{z} = V \cdot sin(\gamma) \tag{5.13}$$

$$_g\dot{V} = \tag{5.14}$$

$$\dot{\gamma} = \tag{5.15}$$

$$\dot{\chi} = \tag{5.16}$$

=¿MP Modell

5.3.2. Discretization of the state and action space

todo

6. Results

6.1. 2D Policy Iteration

6.2. 3D Scenarios

7. Zusammenfassung und Ausblick

Platzhalter...

A. Appendix A: Glider Parameters

B. Appendix B: Computer Configuration

Table (...) contains the technical data of the system used for this work.

Bibliography

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

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

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